

## **1 Цель работы**

Научиться реализовывать алгоритмы поиска.

## **2 Задание**

Вариант 4

**Задание на реализацию:**

### **Задание 2:**

Реализуйте структуру данных «Массив», элементами которого выступают экземпляры класса Car (минимум 10 элементов), содержащие следующие поля (марка, VIN, объем двигателя, стоимость, средняя скорость). Добавьте метод для любой метод сортировки и метод бинарного поиска по полю «VIN». При вызове поиска убедитесь, что элементы структуры данных отсортированы, в ином случае – выбросите исключение.

### **Задание 5:**

Реализуйте структуру данных «Массив», элементами которого выступают экземпляры класса Book (минимум 10 элементов), содержащие следующие поля (автор, издательство, кол-во страниц, стоимость, ISBN). Добавьте метод для любой метод сортировки и метод скачкообразного поиска по полю «автор». При вызове поиска убедитесь, что элементы структуры данных отсортированы, в ином случае – выбросите исключение.

## **3 Теория по заданиям**

**Бинарный поиск (Binary Search)** - это алгоритм поиска используется в отсортированном массиве путем многократного деления интервала поиска пополам.

Для этого алгоритма структура данных, должна быть отсортирована.

Этапы алгоритма:

- Находится средний индекс;
- Значение среднего индекса сравнивается с значением, который надо найти;
- При нахождении, возвращает индекс;
- Если не нашёл и ищущее значение меньше значения среднего индекса то берётся диапазон левой стороны от среднего индекса;
- Если не нашёл и ищущее значение больше значения среднего индекса то берётся диапазон правой стороны от среднего индекса;
- Цикл этапов повторяется с новым диапазоном до тех пор пока не найдёт индекс или диапазон выйдет за границы.

Пример работы алгоритма бинарного поиска на рисунке 3.1.

```
Search index of element = 4
left: 0 | center: 4 | right: 9
[10, 9, 8, 7, 6, 5, 4, 3, 2, 1]
4 < 6
left: 5 | center: 7 | right: 9
[5, 4, 3, 2, 1]
4 > 3
left: 5 | center: 5 | right: 6
[5, 4]
4 < 5
left: 6 | center: 6 | right: 6
[4]
4 == 4
index of element = 6
```

Рисунок 3.1 – Пример работы бинарного поиска на массиве из чисел

**Скачкообразный поиск (Jump Search)** – это алгоритм поиска определённого значения в отсортированном массиве путём перепрыгивания через определённый шаг:

Этапы алгоритма:

- Шаг прыжка = корень длины массива;
- Проходить цикл с шагом прыжка и если попало значение, то возвращаем индекс;
- Если значение в цикле с шагами больше значения определённого шага, то выходим и запоминаем диапазон от индекса прошлого шага до индекса нынешнего шага;
- Ищем значение Линейным поиском (проход циклом по заданному диапазону, ища значения под каждым индексом, пока не дойдёт до конца);
- Если пройдя все циклы не нашёл, то возвращаем -1.

Пример работы алгоритма скачкообразного поиска на рисунке 3.2.

```
Search index of element = 6
[10, 9, 8, 7, 6, 5, 4, 3, 2, 1]
jump: 3
7[3] | 4[6] | 6 > 4[6] |
linear search from: 3 to: 6
7[3] | 6[4] | <-
index of element = 4
```

Рисунок 3.2 - Пример работы скачкообразного поиска на массиве из чисел

#### 4 Код программы для задания 2

Структура данных Массив – Array для задания 2 и задания 5 со всеми методами.

Путь файла с Массивом– structuredata/array.py

```
import ctypes
from typing import Generic, TypeVar, Iterator

T = TypeVar("T")

class Array(Generic[T]):
    # основная переменная - вместимость массива:
    def __init__(self, capacity: int) -> None:
        self.length: int = 0
        self.capacity: int = capacity
        # создаём массив на основе Си
        self.arr: ctypes.Array[T] = (capacity * ctypes.py_object)()

    # возвращает длину
    def get_length(self) -> int:
        return self.length

    # возвращает вместимость
    def get_capacity(self) -> int:
        return self.capacity

    # получить значение по индексу
    def get(self, index: int) -> T:
        # если индекс вне диапазона длины массива, то ошибка
        if index >= self.get_length() or index < 0:
            raise "Index is out of range"
        return self.arr[index]

    # магический метод - получение значения по индексу
    def __getitem__(self, index: int) -> T:
        # если индекс вне диапазона длины массива, то ошибка
        if index >= self.get_length() or index < 0:
            raise "Index is out of range"
        return self.arr[index]

    # выделяем больше места в массиве (увеличивает вместимость)
    def expand_capacity(self, cap: int) -> None:
        size = cap
        if size <= 0:
            size = 1
        # создаёт новый массив с новой вместимостью
        new_arr: ctypes.Array[T] = (size * ctypes.py_object)()
        # передаются все значения прошлого массива
        for i in range(self.get_length()):
            new_arr[i] = self.get(i)
```

```

        # передаём новые данные массив и вместимость
        self.arr: ctypes.Array[T] = new_arr
        self.capacity = size

# добавляет в конец массива элемент
def append(self, elem: T) -> None:
    # если вместимость кончается, то увеличиваем
    if self.length == self.capacity:
        self.expand_capacity(self.capacity * 2)
    # добавляем элемент
    self.arr[self.length] = elem
    self.length += 1

# вставляет элемент на заданную позицию в массив
def insert(self, elem: T, index: int) -> None:
    if index > self.length or index < 0:
        raise "Index is out of range"
    # если вместимость кончается, то увеличиваем
    if self.length == self.capacity:
        self.expand_capacity(self.capacity * 2)
    for i in range(self.length, index - 1, -1):
        if index == 0 and i - 1 < 0:
            self.arr[0] = elem
            continue
        self.arr[i] = self.arr[i - 1]
    self.arr[index] = elem
    self.length += 1

# Магический метод замены значения по индексу
def __setitem__(self, index: int, elem: T) -> None:
    # если индекс вне диапазона длины массива, то ошибка
    if index >= self.get_length() or index < 0:
        raise "Index is out of range"
    self.arr[index] = elem

# удаление по индексу - возвращает удалённый элемент
def remove(self, index: int) -> T:
    # проверка индекса
    if index >= self.get_length() or index < 0:
        raise "Index is out of range"

    elem: T = self.get(index)
    # сдвиг значений по индексам влево на 1 шаг от индекса до
длины - 1
    for i in range(index, self.get_length() - 1):
        self.arr[i] = self.arr[i + 1]

    self.length -= 1
    return elem

# очистка (пустой массив)

```

```

def clear(self) -> None:
    self.length = 0
    self.arr: ctypes.Array[T] = (self.capacity *
ctypes.py_object)()

# Сортировка вставками - по убыванию
def sort_insertion(self) -> None:
    # если массив пуст, то ничего не делается
    if self.length <= 0:
        return
    # от 0 до длины массива, i - индекс значения, который будем
вставлять
    for i in range(self.length):
        # вставка self.arr[i] за j - индексом значения, отсчёт
идет справа налево
        for j in range(i - 1, -1, -1):
            # Вставка происходит если значение индекса i будет
больше значение индекса j вместе с (вставка в начало j == 0 или если
соседнее значение больше чем вставляемое значение)
            if self.arr[i] > self.arr[j] and (j == 0 or
self.arr[i] < self.arr[j - 1]):
                # Вставка значения - копия в нужную позицию
                self.insert(self.arr[i], j)
                # Удаление значения со старой позиции
                self.remove(i + 1)
                # выходим из цикла, так как вставка уже произошла
                break

# Проверка - отсортирован ли массив по убыванию
def is_sorted(self) -> bool:
    for i in range(self.length - 1):
        if self.arr[i] < self.arr[i + 1]:
            return False
    return True

# Алгоритм бинарного поиска (условие - сортировка по убыванию),
если не нашёлся элемент то возвращает -1
def search_binary(self, elem:T) -> int:
    # Проверка отсортированности массива. если нет, то выдаёт
исключение
    if not self.is_sorted():
        raise ValueError("Array is not sorted")

    # диапазон [левый индекс, правый индекс]
    left, right = 0, self.get_length() - 1
    # пока правый индекс больше или равно левому индексу, то
продолжает искать
    while right >= left:
        # индекс центра (деление на цело - округляет в меньшую
сторону)
        center = (right + left) // 2

```

```

        # если значение больше центрального, то берём левую
сторону от центра
        if elem > self.arr[center]:
            right = center - 1
        elif elem < self.arr[center]:
            # если значение меньше центрального, то берём правую
сторону от центра
            left = center + 1
        else:
            # иначе - нашёл и возвращает индекс значения
            return center
    # если не нашёл, то -1
    return -1

    # Алгоритм скачкообразного поиска (условие - сортировка по
убыванию), если не нашёлся элемент то возвращает -1
    def search_jump(self, elem:T) -> int:
        # Проверка отсортированности массива. если нет, то выдаёт
исключение
        if not self.is_sorted():
            raise ValueError("Array is not sorted")

        # диапазон, в котором будет поиск через скачки
        left, right = 0, self.get_length()
        # шаг прыжка - корень от шага прыжка
        jump = int(self.get_length() ** 0.5)
        # если левая часть равна, то возвращаем (чтобы если после
прыжка индекс стал меньше, можно было вернуться назад i - jump)
        if self.arr[left] == elem:
            return left

        # выполняем прыжки и сравнения
        for i in range(left + jump, right, jump):
            # если равен, то нашли
            if elem == self.arr[i]:
                return i
            # если больше, то берём отрезок от предыдущего прыжка до
нынешнего (выход из цикла)
            elif elem > self.arr[i]:
                left = i - jump
                right = i
                break
        else:
            # чтобы с начало не начинать линейный поиск, с последнего
прыжка до конца
            left = i

        # линейный поиск с предыдущего прыжка до последнего прыжка /
или последний прыжок до конца (если прыжки дошли до конца)
        for i in range(left, right):
            if elem == self.arr[i]:

```

```

        return i
    # если не нашёл, то -1
    return -1

    # магический метод итерирования - возвращает значение
    def __iter__(self) -> Iterator[T]:
        for i in range(self.get_length()):
            yield self.get(i)

    # магический метод вывода класса массива в формате [знач1, знач2,
    ... знач N]
    def __str__(self) -> str:
        text = ""
        for i in range(self.get_length()):
            text += str(self.get(i)) + ", "
        if text == "":
            return "[]"
        return f"[{text[:-2]}]"

    # магический метод предназначен для машинно-ориентированного
    вывода
    def __repr__(self) -> str:
        return self.__str__()

```

Для аннотации типа берём из библиотеки **typing** функции - Generic, Optional, TypeVar, Iterator:

- **TypeVar** – любой тип переменной(если задаётся int то всегда будет только int если str то str и т.д.); (универсальный тип);
- **Generic** – Универсальный класс (Абстрактный базовый класс для универсальных типов.) чтобы использовать нужны TypeVar, только аргументы этого типа. (показывает, что за тип без него(Generic) тип не показывает);
- **Optional** – может хранить либо None либо узел списка (T);
- **Iterator** – указывает на то, что объект имеет реализацию метода iter.

Методы класса Массив:

- **\_\_init\_\_(capacity)** – принимает в себя вместимость, создаёт Массив и заполняет его пустотой;
- **get\_length()** – возвращает длину;
- **get\_capacity()** – возвращает вместимость;
- **get(index)** – получить значение по индексу;
- **\_\_getitem\_\_(index)** – магический метод - получение значения по индексу;

- **expand\_capacity(cap)** – выделяем больше места в массиве (увеличивает вместимость);
- **append(elem)** – добавляет в конец массива элемент;
- **insert(elem, index)** – вставляет элемент на заданную позицию в массив;
- **\_\_setitem\_\_(index, elem)** – Магический метод замены значения по индексу;
- **remove(index)** – удаление по индексу - возвращает удалённый элемент;
- **clear()** – очистка (пустой массив);
- **sort\_insertion()** – Сортировка вставками - по убыванию;
- **is\_sorted()** – проверка - отсортирован ли массив по убыванию;
- **search\_binary(elem)** – алгоритм бинарного поиска (условие - сортировка по убыванию), если не нашёлся элемент то возвращает -1;
- **search\_jump(elem)** – алгоритм скачкообразного поиска (условие - сортировка по убыванию), если не нашёлся элемент то возвращает -1;
- **\_\_iter\_\_()** – магический метод итерирования - при итерировании будет возвращать значение по индексу;
- **\_\_str\_\_()** – магический метод вывода класса массива в формате [знач1, знач2, ... знач N];
- **\_\_repr\_\_()** – магический метод предназначен для машинно-ориентированного вывода.

### Класс Car(Машины) для 2 задания:

Путь файла класса Car и примерами элементов этого класса (element) – temp/car.py

```
from typing import Self, Optional
from dataclasses import dataclass
from functools import total_ordering

# дополняет недостающие методы за счёт других (изменяются >=, <=, !=, >)
@total_ordering
@dataclass
class Car:
    mark: str
    vin: str
    engine_capacity: int
    cost: int
    average_speed: float

    # флаг - переключатель основного параметра - по умолчанию cost
    def __post_init__(self) -> None:
        self.flag = "cost"
```



```

# задать основной параметр
def set_flag(self, text: str) -> None:
    if text in ["mark", "vin", "capacity", "cost", "speed"]:
        self.flag = text

# получить основной параметр или получить параметр
def get_flag(self, text: Optional[str] = None) -> str | float:
    tx = self.flag
    if text is not None:
        tx = text
    if tx == "mark":
        return self.mark
    elif tx == "vin":
        return self.vin
    elif tx == "capacity":
        return self.engine_capacity
    elif tx == "cost":
        return self.cost
    elif tx == "speed":
        return self.average_speed
    return self.cost

# магический метод - меньше <
def __lt__(self, other: Self) -> bool:
    other_any = other.get_flag(self.flag)
    return self.get_flag() < other_any

# магический метод - равно ==
def __eq__(self, other: Self) -> bool:
    other_any = other.get_flag(self.flag)
    return self.get_flag() == other_any

# магический метод - сложение +
def __add__(self, other: Self) -> str | float:
    other_any = other.get_flag(self.flag)
    return self.get_flag() + other_any

# магический метод - вычитание -
def __sub__(self, other: Self) -> float:
    other_any = other.get_flag(self.flag)
    return self.get_flag() - other_any

# магический метод - умножение *
def __mul__(self, other: Self) -> str | float:
    other_any = other.get_flag(self.flag)
    return self.get_flag() * other_any

# магический метод - деление /
def __truediv__(self, other: Self) -> float:
    other_any = other.get_flag(self.flag)

```

```

        return self.get_flag() / other_any

# магический метод - деление на цело //
def __floordiv__(self, other: Self) -> int | float:
    other_any = other.get_flag(self.flag)
    return self.get_flag() // other_any

# магический метод - вывода
def __str__(self) -> str:
    return str(self.get_flag())

# магический метод предназначен для машинно-ориентированного
вывода
def __repr__(self) -> str:
    return str(self.get_flag())

# 10 раличных модель машин (10 элементов для дерева)
element = [
    Car("Toyota", "JTJHY00W004036549", 10, 2_000_000, 150),
    Car("Lexus", "JTJHT00W604011511", 20, 2_104_000, 140),
    Car("Hyundai", "KMFGA17PPDC227020", 15, 3_003_000, 160.8),
    Car("Haval", "LGWFF4A59HF701310", 10, 2_500_000, 110.5),
    Car("Jeep", "1J8GR48K25C547741", 12, 2_111_000, 120),
    Car("Chery", "LVVDB11B6ED069797", 8, 1_8000_500, 123),
    Car("Ford", "1FACP42D3PF141464", 30, 4_003_900, 180),
    Car("Tesla", "5YJ3E1EA8KF331791", 24, 3_060_000, 161.4),
    Car("Lada", "XTAKS0Y5LC6599289", 16, 2_044_000, 160),
    Car("Ravon", "XWBJA69V9JA004308", 18, 3_900_000, 152)
]

```

Декоратор **@dataclass** – убирает лишние методы (автоматизирует) `__init__`, `__repr__`, `__str__` и `__eq__` вместо этого можно сразу аннотацию писать

Например было `__init__(self, name) -> self.name = name`, станет `name: str`

`__post_init__(self)` - метод, который срабатывает после `__init__`, здесь можно инициализировать переменные и всё, что можно было в `__init__`.

Декоратор **@total\_ordering** заполняет не достающие методы по уже существующем например по `__lt__` и `__eq__`, задаст остальные методы сравнения.

Методы `set_flag(self, text)` задаёт основное значение по названию (по умолчанию – стоимость), `get_flag(self)` даёт основное значение.

Метод массива бинарного поиска (`search_binary`) с проверкой на отсортированный массив (`is_sorted`):

```

# Проверка - отсортирован ли массив по убыванию
def is_sorted(self) -> bool:

```

```

        for i in range(self.length - 1):
            if self.arr[i] < self.arr[i + 1]:
                return False
        return True

    # Алгоритм бинарного поиска (условие - сортировка по убыванию),
    # если не нашёлся элемент то возвращает -1
    def search_binary(self, elem:T) -> int:
        # Проверка отсортированности массива. если нет, то выдаёт
        # исключение
        if not self.is_sorted():
            raise ValueError("Array is not sorted")

        # диапазон [левый индекс, правый индекс]
        left, right = 0, self.get_length() - 1
        # пока правый индекс больше или равно левому индексу, то
        # продолжает искать
        while right >= left:
            # индекс центра (деление на цело - округляет в меньшую
            # сторону)
            center = (right + left) // 2
            # если значение больше центрального, то берём левую
            # сторону от центра
            if elem > self.arr[center]:
                right = center - 1
            elif elem < self.arr[center]:
                # если значение меньше центрального, то берём правую
                # сторону от центра
                left = center + 1
            else:
                # иначе - нашёл и возвращает индекс значения
                return center
        # если не нашёл, то -1
        return -1

```

#### 4.1 Тесты

Код теста:

```

from structuredata.array import Array
# element - содержит 10 различных элементов машин (10 различных Car)
from temp.car import Car, element

ex_car1 = Car("Haval 0.2v", "LGUFF4A59HF507891", 7, 3_890_000, 88)
ex_car2 = Car("Haval 0.3v", "1J8GR48K25C547741", 6, 3_000_100, 58)
same_cost = Car("Haval 0.4v", "LHHFF4G67HF777001", 5, 3_890_000, 180)

cap = 10
arr = Array(cap)
ind = 4

```

```

sorted_vin = sorted([i.vin for i in element])
sorted_vin.reverse()

# добавление в конец
def test_append():
    for elem in element:
        elem.set_flag("vin")
        arr.append(elem)

    for i in range(len(element)):
        assert element[i] == arr[i]

# проверка функций на get_ дающие значение
def test_gets():
    assert arr.get_capacity() == cap
    assert arr.get_length() == len(element)

    for i in range(len(element)):
        assert element[i] == arr.get(i)

# вставка с проверка размера
def test_resize_capacity():
    new_cap = cap * 2
    arr.append(element[2])
    assert arr.get_capacity() == new_cap

# вставка
def test_insert():
    arr.insert(ex_car1, ind)
    assert arr[ind] == ex_car1
    assert arr[ind - 1] == element[ind - 1]
    assert arr[ind + 1] == element[ind]
    arr.insert(ex_car2, 0)
    assert arr[0] == ex_car2
    assert arr[1] == element[0]
    arr.insert(same_cost, arr.get_length())
    assert arr[arr.get_length() - 1] == same_cost

# задаёт элемент по индексу
def test_set_elem():
    arr[0] = element[-1]
    assert arr[0] == element[-1]

# удаление и очистка
def test_remove_and_clear():
    assert arr.remove(0) == element[-1]
    assert arr.remove(arr.get_length() - 1) == same_cost
    assert arr.remove(arr.get_length() - 1) == element[2]
    assert arr.remove(ind) == ex_car1
    assert arr.get_length() == len(element)

```

```

# проверка сортировки вставками
def test_sort_insertion():
    arr.sort_insertion()
    for i in range(len(element)):
        assert arr[i].get_flag() == sorted_vin[i]

# проверка бинарного поиска
def test_search_binary():
    for i in element:
        nnn = arr.search_binary(i)
        assert arr[nnn] == i

# проверка бинарного поиска - не найден и другой элемент, но один и
# тот же vin
def test_search_binary_unfind_and_same():
    same_cost.set_flag("vin")
    ex_car2.set_flag("vin")
    assert arr.search_binary(same_cost) == -1
    assert arr.search_binary(ex_car2) ==
sorted_vin.index(element[4].vin)

# проверка бинарного поиска
def test_search_binary_except():
    arr.clear()
    for elem in element:
        elem.set_flag("vin")
        arr.append(elem)
    try:
        arr.search_binary(elem)
    except ValueError:
        assert True
        return
    assert False

# проверка вывода
def test_str():
    arr.sort_insertion()
    list_str = "[" + ", ".join(map(str, sorted_vin)) + "]"
    assert list_str == str(arr)

# проверка работы итерации
def test_iteration():
    n = 0
    for elem in arr:
        assert elem.get_flag() in sorted_vin
        assert elem.get_flag() == sorted_vin[n]
        n += 1

```

Результат пройденных тестов (рисунок 4.1).

```

PS D:\Saves\GUAP\SEM3\AuSD\laba5> pytest test_2.py
===== test session starts =====
platform win32 -- Python 3.12.7, pytest-8.3.3, pluggy-1.5.0
rootdir: D:\Saves\GUAP\SEM3\AuSD\laba5
collected 12 items

test_2.py ..... [100%]

===== 12 passed in 0.02s =====

```

Рисунок 4.1 – результат проверки теста 2 задания

Все 12 тестов были пройдены успешно.

## 4.2 Бенчмарк

Был произведён бенчмарк бинарного поиска на количество значений.

Код программы:

```

from structuredata.array import Array
from temp.car import Car, element
from timeit import timeit
from random import choice, randint

times1 = ""

for su in range(1, 5):
    # количество элементов
    k1 = 10 ** su
    table = Array(k1)
    dit = []
    marks = [i.mark for i in element]
    for i in range(k1):
        dit += [Car(
            mark=choice(marks),
            vin="".join(choice("0123456789ABCDEFGHIJKLMNOPQRSTUVWXYZ"))
        )]
    for i in range(17)):
        engine_capacity=randint(50, 400),
        cost=randint(900_000, 6_000_000),
        average_speed=randint(70_00, 250_00)/100
    )]

    for x in dit:
        x.set_flag("vin")
        table.append(x)

    table.sort_insertion()
    find_elem = choice(dit)
    code1 = """table.search_binary(find_elem)"""

    time1 = timeit(code1, number=20, globals={"find_elem": find_elem,
"table": table})

```

```

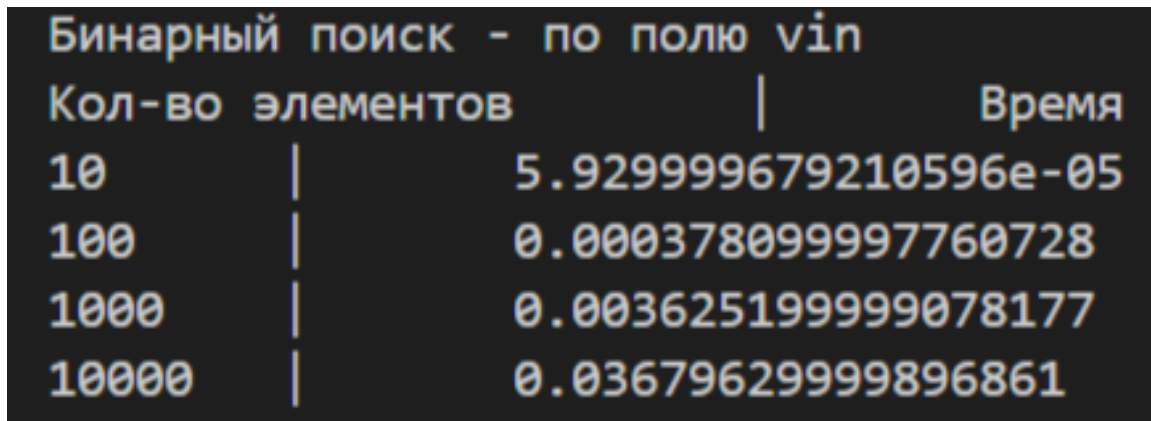
table.clear()

times1 += f"{k1}\t|\t{time1}\n"

print("Бинарный поиск - по полю vin")
print(f"Кол-во элементов\t|\tВремя")
print(times1)

```

Вывод в консоли изображено на рисунке 4.2.



```

Бинарный поиск - по полю vin
Кол-во элементов      |      Время
10                    | 5.929999679210596e-05
100                   | 0.000378099997760728
1000                  | 0.003625199999078177
10000                 | 0.03679629999896861

```

Рисунок 4.2 – вывод в консоли времени в секунды

Можно заметить, что количество элементов влияет на время каждого алгоритма бинарного поиска.

Кол-во элементов	Время в секундах
10	0.0000593
100	0.0003781
1000	0.0036252
10000	0.0367963

Таблица 4.1 – Бинарный поиск

## 5 Код программы для задания 5

### Класс Book(Книги) для 5 задания:

Путь файла класса Book и примерами элементов этого класса (element\_book) – temp/book.py

```

from typing import Self, Optional
from dataclasses import dataclass
from functools import total_ordering

# дополняет недостающие методы за счёт других (изменяются >=, <=, !=, >)
@total_ordering

```

```

@dataclass
class Book:
    author: str
    publisher: str
    pages: int
    cost: int
    isbn: int

    # флаг - переключатель основного параметра - по умолчанию cost
    def __post_init__(self) -> None:
        self.flag = "cost"

    # задать основной параметр
    def set_flag(self, text: str) -> None:
        if text in ["author", "publisher", "pages", "cost", "isbn"]:
            self.flag = text

    # получить основной параметр или получить параметр
    def get_flag(self, text: Optional[str] = None) -> str | float:
        tx = self.flag
        if text is not None:
            tx = text
        if tx == "author":
            return self.author
        elif tx == "publisher":
            return self.publisher
        elif tx == "pages":
            return self.pages
        elif tx == "cost":
            return self.cost
        elif tx == "isbn":
            return self.isbn
        return self.cost

    # магический метод - меньше <
    def __lt__(self, other: Self) -> bool:
        other_any = other.get_flag(self.flag)
        return self.get_flag() < other_any

    # магический метод - равно ==
    def __eq__(self, other: Self) -> bool:
        other_any = other.get_flag(self.flag)
        return self.get_flag() == other_any

    # магический метод - сложение +
    def __add__(self, other: Self) -> str | int | float:
        other_any = other.get_flag(self.flag)
        return self.get_flag() + other_any

    # магический метод - вычитание -
    def __sub__(self, other: Self) -> int | float:

```



```

        other_any = other.get_flag(self.flag)
        return self.get_flag() - other_any

# магический метод - умножение *
def __mul__(self, other: Self) -> str | int | float:
    other_any = other.get_flag(self.flag)
    return self.get_flag() * other_any

# магический метод - деление /
def __truediv__(self, other: Self) -> float:
    other_any = other.get_flag(self.flag)
    return self.get_flag() / other_any

# магический метод - деление на цело //
def __floordiv__(self, other: Self) -> int | float:
    other_any = other.get_flag(self.flag)
    return self.get_flag() // other_any

# магический метод - вывода
def __str__(self) -> str:
    return str(self.get_flag())

# магический метод предназначен для машинно-ориентированного
вывода
def __repr__(self) -> str:
    return str(self.get_flag())

# 10 раличных модель машин (10 элементов для дерева)
element_book = [
    Book("Л. Н. Толстой", "Эксмо", 1000, 472, 9785699120147),
    Book("О. Л. Ершова", "Эксмо", 520, 473, 9785699867935),
    Book("Роберт Джордан", "Азбука", 33, 401, 9785677520647),
    Book("В. Д. Афиногенов", "Вече", 45, 277, 9785697425819),
    Book("Люси Мод Монтгомери", "Азбука", 145, 370, 9785696450122),
    Book("М. Ю. Лермонтов", "Азбука", 76, 320, 9785699224456),
    Book("Н. Н. Телупова", "Эксмо", 80, 250, 9785693322112),
    Book("С. Н. Зигуненко", "Вече", 60, 501, 9785664550121),
    Book("И. Е. Забелин", "Азбука", 120, 399, 9785693124755),
    Book("В. Д. Афиногенов", "Вече", 230, 410, 9785699673146)
]

```

Методы `set_flag(self, text)` задаёт основное значение по названию (по умолчанию — стоимость), `get_flag(self)` даёт основное значение.

Метод массива скачкообразного поиска (`search_jump`) с проверкой на отсортированный массив (`is_sorted`):

```

# Проверка - отсортирован ли массив по убыванию
def is_sorted(self) -> bool:

```

```

        for i in range(self.length - 1):
            if self.arr[i] < self.arr[i + 1]:
                return False
        return True

    # Алгоритм скачкообразного поиска (условие - сортировка по
    убыванию), если не нашёлся элемент то возвращает -1
    def search_jump(self, elem:T) -> int:
        # Проверка отсортированности массива. если нет, то выдаёт
        исключение
        if not self.is_sorted():
            raise ValueError("Array is not sorted")

        # диапазон, в котором будет поиск через скачки
        left, right = 0, self.get_length()
        # шаг прыжка - корень от шага прыжка
        jump = int(self.get_length() ** 0.5)
        # если левая часть равна, то возвращаем (чтобы если после
        прыжка индекс стал меньше, можно было вернуться назад i - jump)
        if self.arr[left] == elem:
            return left

        # выполняем прыжки и сравнения
        for i in range(left + jump, right, jump):
            # если равен, то нашли
            if elem == self.arr[i]:
                return i
            # если больше, то берём отрезок от предыдущего прыжка до
            нынешнего (выход из цикла)
            elif elem > self.arr[i]:
                left = i - jump
                right = i
                break
        else:
            # чтобы с начало не начинать линейный поиск, с последнего
            прыжка до конца
            left = i

            # линейный поиск с предыдущего прыжка до последнего прыжка /
            или последний прыжок до конца (если прыжки дошли до конца)
            for i in range(left, right):
                if elem == self.arr[i]:
                    return i
            # если не нашёл, то -1
            return -1

```

## 5.1 Тесты

Код теста:

```
from structuredata.array import Array
```

```

# element_book - содержит 10 различных элементов книг (10 различных
Book)
from temp.book import Book, element_book

ex_book1 = Book("П. Л. Нимский", "Эксмо", 520, 555, 9780000000000)
ex_book2 = Book("Люси Мод Монтгомери", "Эксмо", 120, 55,
9780000000001)
same_cost = Book("И. Д. Тимухин", "Азбука", 520, 255, 9780000000002)

cap = 10
arr = Array(cap)
ind = 4

sorted_author = sorted([i.author for i in element_book])
sorted_author.reverse()

# добавление в конец
def test_append():
    for elem in element_book:
        elem.set_flag("author")
        arr.append(elem)

    for i in range(len(element_book)):
        assert element_book[i] == arr[i]

# проверка функций на get_ дающие значение
def test_gets():
    assert arr.get_capacity() == cap
    assert arr.get_length() == len(element_book)

    for i in range(len(element_book)):
        assert element_book[i] == arr.get(i)

# вставка с проверка размера
def test_resize_capacity():
    new_cap = cap * 2
    arr.append(element_book[2])
    assert arr.get_capacity() == new_cap

# вставка
def test_insert():
    arr.insert(ex_book1, ind)
    assert arr[ind] == ex_book1
    assert arr[ind - 1] == element_book[ind - 1]
    assert arr[ind + 1] == element_book[ind]
    arr.insert(ex_book2, 0)
    assert arr[0] == ex_book2
    assert arr[1] == element_book[0]
    arr.insert(same_cost, arr.get_length())
    assert arr[arr.get_length() - 1] == same_cost

```

```

# задаёт элемент по индексу
def test_set_elem():
    arr[0] = element_book[-1]
    assert arr[0] == element_book[-1]

# удаление и очистка
def test_remove_and_clear():
    assert arr.remove(0) == element_book[-1]
    assert arr.remove(arr.get_length() - 1) == same_cost
    assert arr.remove(arr.get_length() - 1) == element_book[2]
    assert arr.remove(ind) == ex_book1
    assert arr.get_length() == len(element_book)

# проверка сортировки вставками
def test_sort_insertion():
    arr.sort_insertion()
    for i in range(len(element_book)):
        assert arr[i].get_flag() == sorted_author[i]

# проверка скачкообразного поиска
def test_search_jump():
    for i in element_book:
        nnn = arr.search_jump(i)
        assert arr[nnn] == i

# проверка скачкообразного поиска - не найден и другой элемент, но
# один и тот же author
def test_search_jump_unfind_and_same():
    same_cost.set_flag("author")
    ex_book2.set_flag("author")
    assert arr.search_jump(same_cost) == -1
    assert arr.search_jump(ex_book2) ==
sorted_author.index(element_book[4].author)

# проверка скачкообразного поиска
def test_search_jump_except():
    arr.clear()
    for elem in element_book:
        elem.set_flag("author")
        arr.append(elem)
    try:
        arr.search_jump(elem)
    except ValueError:
        assert True
        return
    assert False

# проверка вывода
def test_str():
    arr.sort_insertion()
    list_str = "[" + ", ".join(map(str, sorted_author)) + "]"

```

```

    assert list_str == str(arr)

# проверка работы итерации
def test_iteration():
    n = 0
    for elem in arr:
        assert elem.get_flag() in sorted_author
        assert elem.get_flag() == sorted_author[n]
        n += 1

```

Результат пройденных тестов (рисунок 5.1).

```

PS D:\Saves\GUAP\SEM3\AuSD\laba5> pytest test_5.py
===== test session starts =====
platform win32 -- Python 3.12.7, pytest-8.3.3, pluggy-1.5.0
rootdir: D:\Saves\GUAP\SEM3\AuSD\laba5
collected 12 items

test_5.py ..... [100%]

===== 12 passed in 0.02s =====

```

Рисунок 5.1 – результат проверки теста 5 задания

Все 12 тестов были пройдены успешно.

## 5.2 Бенчмарк

Был произведён бенчмарк скачкообразного поиска на количество значений.

Код программы:

```

from structuredata.array import Array
from temp.book import Book
from timeit import timeit
from random import choice, randint
from string import ascii_letters

times1 = ""

for su in range(1, 5):
    # количество элементов
    k1 = 10 ** su
    table = Array(k1)
    dit = []
    for i in range(k1):
        dit += [Book(
            author="".join(choice(ascii_letters) for i in
range(randint(5, 30))),
            publisher="".join(choice(ascii_letters) for i in
range(randint(5, 30))),
            pages=randint(10, 1000),

```

```

        cost=randint(100, 10000),
        isbn=int("".join(str(choice(range(10))) for i in
range(13)))
    )]

    for x in dit:
        x.set_flag("author")
        table.append(x)

    table.sort_insertion()
    find_elem = choice(dit)
    code1 = """table.search_jump(find_elem)"""

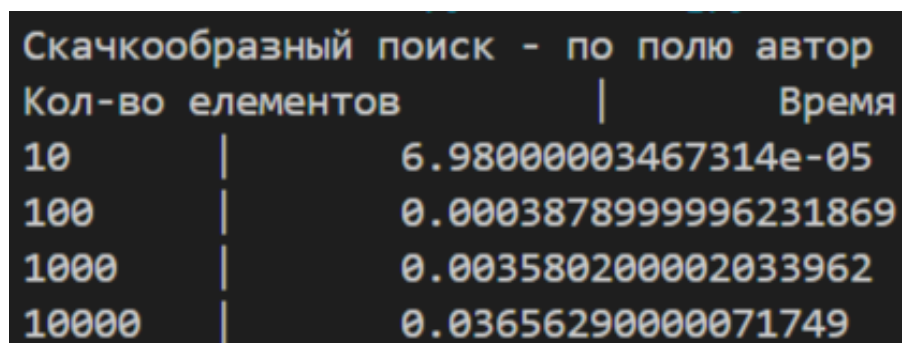
    time1 = timeit(code1, number=20, globals={"find_elem": find_elem,
"table": table})
    table.clear()

    times1 += f"{k1}\t|\t{time1}\n"

print("Скачкообразный поиск - по полю автор")
print(f"Кол-во элементов\t|\tВремя")
print(times1)

```

Вывод в консоли изображено на рисунке 5.2.



```

Скачкообразный поиск - по полю автор
Кол-во элементов      |      Время
10      |      6.98000003467314e-05
100     |      0.0003878999996231869
1000    |      0.003580200002033962
10000   |      0.03656290000071749

```

Рисунок 5.2 – вывод в консоли времени в секунды

Кол-во элементов	Время в секундах
10	0.0000698
100	0.0003879
1000	0.0035802
10000	0.0365629

Таблица 5.1 – Скачкообразный поиск

По полученным результатам алгоритмов поиска, можно понять, что оба алгоритма: Бинарный и Скачкообразный поиск, почти одинаковы по скорости. Но всё же Бинарный оказался на немного быстрее чем Скачкообразный поиск.

## **Выводы**

В данной лабораторной работе я создала структуру данных Массив с применением таких алгоритмов поиска как:

- Бинарный поиск;
- Скачкообразный поиск;

Научилась применять аннотацию типов с помощью библиотеки `typing`. А также провела тесты своей структуры данных с алгоритмами поиска вместе с бенчмарком. Бенчмарк показал, что для массива хорошо подходят оба алгоритма поиска, за счёт их лёгкости и простоты.