1 Цель работы

Научиться реализовывать такие базовые структуры данных, как: очередь, дерево и куча.

2 Задание

Вариант 4

Задание на реализацию:

Задание 4:

Реализуйте структуру данных «Двоичное дерево поиска», элементами которой выступают экземпляры класса Car (минимум 10 элементов), содержащие следующие поля (марка, VIN, объем двигателя, стоимость, средняя скорость), где в качестве ключевого элемента при добавлении будет выступать стоимость. Структура данных должна иметь возможность сохранять свое состояние в файл и загружать данные из него. Также реализуйте 2 варианта проверки вхождения элемента в структуру данных.

Задание 10:

Реализуйте структуру данных «Минимальная куча» на основе односвязного списка, элементами которой выступают экземпляры класса Car (минимум 10 элементов), содержащие следующие поля (марка, VIN, объем двигателя, стоимость, средняя скорость), где в качестве ключевого элемента при добавлении будет выступать стоимость. Структура данных должна иметь возможность сохранять свое состояние в файл и загружать данные из него. Также реализуйте 2 варианта проверки вхождения элемента в структуру данных.

3 Теория по заданиям

Дерево двоичного поиска (BST) — это двоичное дерево, в котором каждый узел в левом поддереве меньше корня, а каждый узел в правом поддереве больше корня. Свойства дерева двоичного поиска являются рекурсивными

- Левое поддерево узла содержит только узлы с ключами, меньшими, чем ключ узла.
- Правое поддерево узла содержит только узлы с ключами, превышающими ключ узла.
- Левое и правое поддеревья также должны быть деревьями двоичного поиска.
- Не должно быть повторяющихся узлов

Дерево двоичного поиска можно визуализировать как на рисунке 3.1.

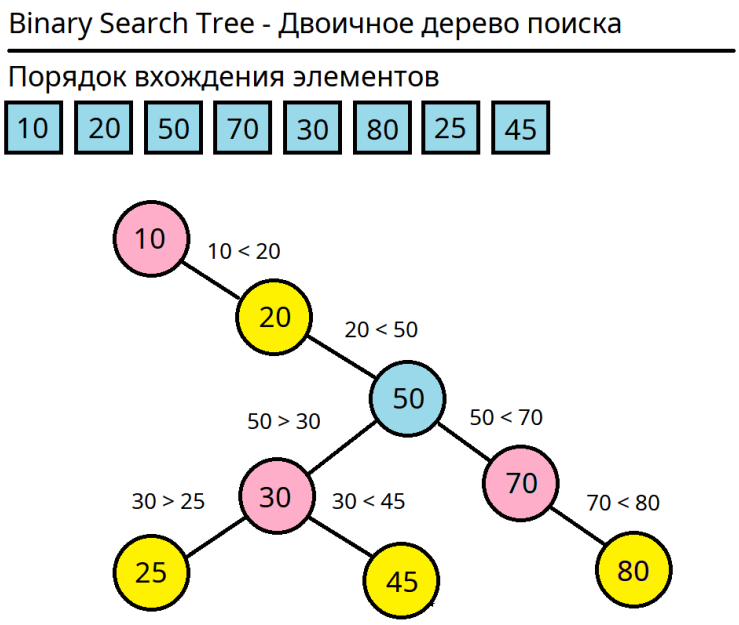

Рисунок 3.1 – Односвязный список

Минимальная куча на основе односвязный список - это куча, где корневой элемент минимален, а каждый родительский узел меньше или равен своим потомкам. Который вместо массива принимает односвязный список.

4 Код программы для задания 4

Класс машины для 4 и 10 задания:

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

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

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

    # магический метод - меньше <
    def __lt__(self, other: Self) -> bool:
        return self.cost < other.cost

    # магический метод - равно ==
```

```
    def __eq__(self, other: Self) -> bool:
        return self.cost == other.cost

    # магический метод - сложение +
    def __add__(self, other: Self) -> float:
        return self.cost + other.cost

    # магический метод - вычитание -
    def __sub__(self, other: Self) -> float:
        return self.cost - other.cost

    # магический метод - умножение *
    def __mul__(self, other: Self) -> float:
        return self.cost * other.cost

    # магический метод - деление //
    def __truediv__(self, other: Self) -> float:
        return self.cost / other.cost

    # магический метод - деление на цело //
    def __floordiv__(self, other: Self) -> float:
        return self.cost // other.cost

    # магический метод - вывода
    def __str__(self) -> str:
        return f"{self.mark} | {self.cost}"


# 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", 10.5, 2_500_000, 110.5),
    Car("Jeep", "1J8GR48K25C547741", 12.3, 2_111_000, 120),
    Car("Chery", "LVVDB11B6ED069797", 8, 1_800_500, 123),
    Car("Ford", "1FACP42D3PF141464", 30, 4_003_900.5, 180),
    Car("Tesla", "5YJ3E1EA8KF331791", 24.3, 3_060_000, 161.4),
    Car("Lada", "XTAKS0Y5LC6599289", 16, 2_044_000, 160),
    Car("Ravon", "XWBJA69V9JA004308", 18.2, 3_000_000, 152)
]
```

Путь файла с Двоичного дерева поиска– structuredata/binarysearchtree.py

```
from temp.car import Car
import json
from typing import Generic, Optional, TypeVar
from dataclasses import dataclass

T = TypeVar("T")
```

```python
# класс Узла для Двоичного дерева поиска,
# в котором хранится элемент и ссылки на None или на левый и правый
Узел
@dataclass
class Node(Generic[T]):
    key: T
    left: Optional['Node[T]'] = None
    right: Optional['Node[T]'] = None


# Класс Двоичного дерева поиска
class BinarySearchTree(Generic[T]):
    # Хранит путь (начальный узел) Двоичного дерева поиска
    def __init__(self) -> None:
        self.root: Optional[Node[T]] = None

    # Вставка по ключу
    def insert(self, key: T) -> None:
        # если пусто, то создаём узел
        if self.root is None:
            self.root = Node(key)
            return
        root = self.root
        while root is not None:
            if key == root.key:
                return
            # если в значение ключа больше то в правую ветвь
            if key > root.key:
                if root.right is None:
                    root.right = Node(key)
                    return
                root = root.right
            else:
                # иначе в значение ключа меньше то в левую ветвь
                if root.left is None:
                    root.left = Node(key)
                    return
                root = root.left
        return

    # Находит по ключу узел из Двоичного дерева поиска
    def find(self, key: T) -> Optional[Node[T]]:
        return self.recursive_find(self.root, key)

    # (Рекурсия) Возвращает узел по заданному пути ключа и ключу
    def recursive_find(self, root: Optional[Node[T]], key: T) ->
Optional[Node[T]]:
        # если нету продолжения в узле ключа, возвращаем место, на
котором остановились или если нашёл ключ
        if root is None or key == root.key:
```

```
            return root

        # рекурсия для прохождения по всем узлам если значение ключа
больше чем у узле, то вправо иначе влево
        if key > root.key:
            return self.recursive_find(root.right, key)
        return self.recursive_find(root.left, key)

    # (Рекурсия) Возвращает предыдущий узел ключа (который имеет
ссылку на ключ) и True|False (флаг) для понимания справа или слева
    def recursive_prefind(self, root: Optional[Node[T]], key: T) ->
tuple[Optional[Node[T]], bool]:
        # если нету продолжения в узле ключа, возвращаем место, на
котором остановились или если ключ равен (если выбран начальный ключ)
        if root is None or key == root.key:
            return root, False
        # Если у предыдущего ключа справа ключ, то возвращаем узел и
True (ключ справа)
        if root.right is not None and key == root.right.key:
            return root, True
        # Если у предыдущего ключа слева ключ, то возвращаем узел и
False (ключ слева)
        if root.left is not None and key == root.left.key:
            return root, False

        # рекурсия для прохождения по всем узлам если значение ключа
больше чем у узле, то вправо иначе влево
        if key > root.key:
            return self.recursive_prefind(root.right, key)
        return self.recursive_prefind(root.left, key)

    # Возвращает максимальное значение из узел Двоичного дерева поиска
    def max(self) -> Optional[T]:
        return self.get_max(self.root)

    # Возвращает минимальное значение из узел Двоичного дерева поиска
    def min(self) -> Optional[T]:
        return self.get_min(self.root)

    # Возвращает с заданным путём(узлом) максимальное значение из узел
(по правой ветке)
    def get_max(self, xroot: Optional[Node[T]]) -> Optional[T]:
        root = xroot
        if root is None:
            return
        while root.right is not None:
            root = root.right
        return root.key

    # Возвращает с заданным путём(узлом) минимальное значение из узел
(по левой ветке)
```

```
    def get_min(self, xroot: Optional[Node[T]]) -> Optional[T]:
        root = xroot
        if root is None:
            return
        while root.left is not None:
            root = root.left
        return root.key

    # Метод удаления по ключу
    def remove(self, key: T) -> None:
        # находим узел ключа
        root = self.find(key)
        # если не был найден, то прекращаем удаление
        if root is None:
            return
        # предыдущий узел, справа? (да/нет)
        pre_root, is_right = self.recursive_prefind(self.root, key)
        # у узла есть ветвления влево, вправо?
        left, right = root.left is not None, root.right is not None
        # нету слева и нету справа
        if not(left) and not(right):
            if pre_root == root:
                self.root = None
            # если ключ справа, иначе слева - присваиваем пустоту None
            if is_right:
                pre_root.right = None
            else:
                pre_root.left = None
        elif left and not(right):
            # (нету ветки справа) если ключ справа, иначе слева -
присваиваем левую ветку узла
            if is_right:
                pre_root.right = root.left
            else:
                pre_root.left = root.left
        elif not(left) and right:
            # (нету ветки слева) если ключ справа, иначе слева -
присваиваем правую ветку узла
            if is_right:
                pre_root.right = root.right
            else:
                pre_root.left = root.right
        else:
            # ветки есть слева и справа - ищем у узла по левой ветки
его максимум, а по правой ветки его минимум
            mx, mn = self.get_max(root.left), self.get_min(root.right)
            # Минимальный ключ - Делаем сравнение по модулю (у кого
разница меньше с значением ключа, того заменяем на место удаления)
            key_min = min([mx, mn], key=lambda x: abs(key - x))
            # Предыдущий узел минимального ключа, справа? (да/нет)
```

```
            pop_root, pop_is_right = self.recursive_prefind(root,
key_min)

            # значение узла минимального ключа
            popkey: Optional[T] = None
            if pop_is_right:
                # присваиваем ключ для замены и чистим путь
                popkey = pop_root.right
                pop_root.right = None
            else:
                popkey = pop_root.left
                pop_root.left = None

            # делаем замену на другой ключ
            if popkey.right is not None:
                root.right = popkey.right
            if popkey.left is not None:
                root.left = popkey.left

            if root == pop_root:
                if pop_is_right:
                    mn = self.get_min(root.right)
                    if mn is not None:
                        ro = self.find(mn)
                        ro.left = popkey.left
                else:
                    mx = self.get_max(root.left)
                    if mx is not None:
                        ro = self.find(mx)
                        ro.right = popkey.right
            root.key = popkey.key

    # Шаг слева для метода рисования простого графика дерева (путь,
шаг слева, список чисел шагов) - возвращает максимальный левый шаг
    def steps_left(self, root: Optional[Node[T]], step:int=0,
m:list[int]=[]) -> int:
        if root is None:
            return max(m)
        m += [step]
        self.steps_left(root.right, step - 1, m)
        self.steps_left(root.left, step + 1, m)
        return max(m)

    # Рисует простой график дерева
    def graph_print_tree(self, root: Optional[Node[T]], step:int,
n:int=0, k:int=0) -> Optional[Node[T]]:
        if root is None:
            return None
        # значение ключа
        tx = root.key.__str__()
        # шаг слева с табуляцией
```

```
        print(f"{(step - k) * "\t" } {n * "+"} [ {tx} ] {k * "-"}")
        self.graph_print_tree(root.left, step, n - 1, k + 1)
        self.graph_print_tree(root.right, step, n + 1, k - 1)

    # (возвращает список ключей) Предпоследовательный обход - обход
начиная с корня сначала слева, а потом справа (сохраняет все значение
ключей в список)
    def list_preorder_tree(self, root: Optional[Node[T]],
n:list[T]=list()) -> list:
        if root is None:
            return n
        n += [root.key]
        self.list_preorder_tree(root.left, n)
        self.list_preorder_tree(root.right, n)
        return n

    # (возвращает список ключей) Последовательный обход - обход по
порядку
    def list_inorder_tree(self, root: Optional[Node[T]],
n:list[T]=list()) -> list:
        if root is None:
            return n
        self.list_inorder_tree(root.left, n)
        n += [root.key]
        self.list_inorder_tree(root.right, n)
        return n

    # (возвращает список ключей) Последующий обход - обход сначала
слева, а потом справа заканчивая корнем
    def list_postorder_tree(self, root: Optional[Node[T]],
n:list[T]=list()) -> list:
        if root is None:
            return n
        self.list_postorder_tree(root.left, n)
        self.list_postorder_tree(root.right, n)
        n += [root.key]
        return n

    # Количество ключей
    def get_size(self) -> int:
        if self.root is None:
            return 0
        return len(self.list_preorder_tree(self.root, []))

    # Методы с приставкой print_ - выводы обходов
    def print_preorder_tree(self) -> None:
        spis = map(str, self.list_preorder_tree(self.root, []))
        print("Предпоследовательный обход: " + ", ".join(spis))

    def print_inorder_tree(self) -> None:
        spis = map(str, self.list_inorder_tree(self.root, []))
```

```
        print("Последовательный обход: " + ", ".join(spis))

    def print_postorder_tree(self) -> None:
        spis = map(str, self.list_postorder_tree(self.root, []))
        print("Последующий обход: " + ", ".join(spis))

    # Сохранение состояния Двоичного дерева поиска в json формате
    def save(self, path: str) -> None:
        with open(path, "w", encoding="UTF-8") as file_save:
            json.dump(self.__dict__, file_save, ensure_ascii=False,
default=str)

    # Загрузка состояния Двоичного дерева поиска в json формате

    def load(self, path: str) -> None:
        with open(path, "r", encoding="UTF-8") as file_load:
            self.root = eval(json.load(file_load)["root"])

    # Графический вывод через магический метод
    def __str__(self) -> str:
        k = self.steps_left(self.root)
        self.graph_print_tree(self.root, k)
        return ""
```

Класс BinarySearchTree хранит начальный узел Node или ничего None - root (корень, из которого сторится дальнейшее дерево):

- **insert(key)** - вставка по ключу;
- **find(key)** - находит по ключу узел из Двоичного дерева поиска;
- **recursive_find(root, key)** - возвращает узел по заданному пути ключа и ключу через рекурсию;
- **recursive_prefind(root, key)** - возвращает предыдущий узел ключа (который имеет ссылку на ключ) и True|False (флаг) для понимания справа или слева через рекурсию;
- **max()** - возвращает максимальное значение из узел Двоичного дерева поиска;
- **min()** - возвращает минимальное значение из узел Двоичного дерева поиска;
- **get_max(xroot)** - возвращает с заданным путём(узлом) максимальное значение из узел (по правой ветке);
- **get_min(xroot)** - возвращает с заданным путём(узлом) минимальное значение из узел (по левой ветке);
- **remove(key)** - удаляет по ключу;
- **steps_left(root, step=0, m=[])** - шаг слева для рисования простого графика дерева;

- **graph_print_tree(root, step, n=0, k=0)** - рисует простой график дерева через рекурсию;
- **list_preorder_tree(root, n=[])** - возвращает список ключей предпоследовательного обхода;
- **list_inorder_tree(root, n=[])** - возвращает список ключей последовательного обхода;
- **list_postorder_tree(root, n=[])** - возвращает список ключей последующего обхода;
- **get_size()** - возвращает количество ключей в двоичном дереве поиска;
- **print_preorder_tree()** - вывод предпоследовательного обхода;
- **print_inorder_tree()** - вывод последовательного обхода;
- **print_postorder_tree()** - вывд последующего обхода;
- **save(path)** - сохраняет состояние Двоичного дерева поиска в json формате;
- **load(path)** - загружает состояние Двоичного дерева поиска в json формате;
- **__str__()** - графический вывод через магический метод вывода.

4.1 Тесты

Код теста:

```
from structuredata.binarysearchtree import BinarySearchTree
# element - содержит 10 различных элементов машин (10 различных Car)
from temp.car import Car, element
import os.path

ex_car1 = Car("Haval 0.2v", "LGUFF4A59HF507891", 7, 3_890_000, 88)
same_cost = Car("Haval 0.3v", "LHHFF4G67HF777001", 5, 3_890_000, 180)
binseatre = BinarySearchTree()


# вставка
def test_insert():
    for i in element:
        binseatre.insert(i)
    btree = binseatre.list_preorder_tree(binseatre.root)
    for it in element:
        assert it in btree

# вставка с таким же ключём и проверка размера
def test_insert_same_and_size():
    binseatre.insert(ex_car1)
    size1 = binseatre.get_size()
    btree_pre1 = binseatre.list_preorder_tree(binseatre.root)
    binseatre.insert(same_cost)
    size2 = binseatre.get_size()
    btree_pre2 = binseatre.list_preorder_tree(binseatre.root)
```

```
    assert btree_pre1 == btree_pre2
    assert size1 == size2
    binseatre.root = None
    assert binseatre.get_size() == 0

# удаление
def test_remove():
    for i in element:
        binseatre.insert(i)
    btree = binseatre.list_preorder_tree(binseatre.root)
    binseatre.insert(ex_car1)
    binseatre.remove(ex_car1)
    btree2 = binseatre.list_preorder_tree(binseatre.root)
    assert btree == btree2

# сохранение состояния в файл (проверка существует ли файл)
def test_save():
    path = "test_json1.json"
    binseatre.save(path)
    assert os.path.exists(path)

# загрузка файла в пустое дерево
def test_load():
    path = "test_json1.json"
    btree_pre = binseatre.list_preorder_tree(binseatre.root)
    btree_post = binseatre.list_postorder_tree(binseatre.root)
    btree_in = binseatre.list_inorder_tree(binseatre.root)
    binseatre.root = None
    binseatre.load(path)
    b1 = binseatre.list_preorder_tree(binseatre.root)
    b2 = binseatre.list_postorder_tree(binseatre.root)
    b3 = binseatre.list_inorder_tree(binseatre.root)
    assert b1 == btree_pre
    assert b2 == btree_post
    assert b3 == btree_in

# проверка нахождения ключа
def test_find():
    nen = binseatre.find(same_cost)
    assert nen is None
    assert binseatre.find(element[3]).key == element[3]

# Проверка на максимальное значение и минимальное
def test_max_min():
    costs = [i for i in element]
    mx, mn = max(costs), min(costs)
    assert mx == binseatre.max()
    assert mn == binseatre.min()
```

Результат пройденных тестов (рисунок 4.1).

```
PS D:\Saves\GUAP\SEM3\AuSD\laba3> pytest test_4.py
========================= test session starts =========================
platform win32 -- Python 3.12.7, pytest-8.3.3, pluggy-1.5.0
rootdir: D:\Saves\GUAP\SEM3\AuSD\laba3
collected 7 items

test_4.py .......                                                [100%]

========================== 7 passed in 0.02s ==========================
PS D:\Saves\GUAP\SEM3\AuSD\laba3>
```

Рисунок 4.1 – результат проверки теста 4 задания

Все 7 тестов были пройдены успешно.

4.2 Бенчмарк

Был произведён бенчмарк на вместимость и количество значений.

Код программы:

```
from structuredata.binarysearchtree import BinarySearchTree
from temp.car import Car, element
from timeit import timeit
from random import choice, randint

times1 = ""
times2 = ""

for su in range(1, 6):
    table = BinarySearchTree()
    # колличество элементов
    k1 = 10 ** su
    dit = []
    marks = [i.mark for i in element]
    for i in range(k1):
        dit += [Car(
            mark=choice(marks),
            vin="".join(choice("0123456789ABCDEFGHJKLMNPRSTUVWXYZ")
for i in range(17)),
            engine_capacity=randint(5_00, 40_00)/100,
            cost=randint(900_000 * 100, 6_000_000 * 100)/100,
            average_speed=randint(70_00, 250_00)/100
        )]

    for x in dit:
        table.insert(x)
    for x in dit:
        table.remove(x)
    code1 = """for x in dit:
        table.insert(x)
```

```
    """
    code2 = """for x in dit:
        table.remove(x)
    """
    time1 = timeit(code1, number=1, globals={"dit": dit, "table":
table})
    time2 = timeit(code2, number=1, globals={"dit": dit, "table":
table})
    times1 += f"        {k1}            |          {time1}\n"
    times2 += f"        {k1}            |        {time2}\n"

print("Вставка")
print(f"Кол-во елементов    |   Время")
print(times1)
print("Удаление")
print(f"Кол-во елементов    |   Время")
print(times2)
```

Вывод в консоли изображено на рисунке 5.2.

```
Вставка
Кол-во елементов    |   Время
        10          |          9.099996532313526e-06
        100         |          0.00014010000450070947
        1000        |          0.002137100003892556
        10000       |          0.03328459999465849
        100000      |          0.49338829999032896

Удаление
Кол-во елементов    |   Время
        10          |        2.0799998310394585e-05
        100         |        0.0002150999935111031
        1000        |        0.0016226999869104475
        10000       |        0.019884699999238364
        100000      |        0.1284691000037128
```

Рисунок 4.2 – вывод в консоли времени в секунды

Можно заметить, что количество элементов влияет на время обработки Двоичного дерева поиска.

Кол-во элементов (вместимость)	Время
10	0.00000909999532313526
100	0.00014010000450070947

1000	0.002137100003892556
10000	0.03328459999465849
100000	0.49338829999032896

Таблица 4.1 – Вставка в Двоичном дереве поиска

Кол-во элементов (вместимость)	Время
10	0.00002079999810394585
100	0.0002150999935111031
1000	0.0016226999869104475
10000	0.019884699999238364
100000	0.1284691000037128

Таблица 4.2 – Удаление в Двоичном дереве поиска

5 Код программы для задания 10

Путь файла с Односвязным списком – structuredata/simplelinkedlists.py

```
from typing import Generic, Optional, TypeVar, Iterator
from dataclasses import dataclass


T = TypeVar("T")

# класс Узла для ОдносвязногоСписка,
# в котором хранится элемент и ссылка на None или на следующий Узел
@dataclass
class Node(Generic[T]):
    data: T
    next: Optional['Node[T]'] = None


class SimpleLinkedList(Generic[T]):
    # две основные переменные:
    # голова (начальный Узел) и хвост (конечный Узел)
    def __init__(self) -> None:
        self.head: Optional[Node[T]] = None
        self.tail: Optional[Node[T]] = None

    # возвращает длинну Односвязного списка
    def get_length(self) -> int:
        length: int = 0
        node = self.head
        # если пусто то 0
        if node is None:
```

```
            return 0
        # засчитываем каждый Узел
        while node is not None:
            node = node.next
            length += 1
        return length

    # возвращает True если индекс есть в Списке, False - Нету
    def check_range(self, index: int) -> bool:
        length = self.get_length()
        # Если 0, то пусто
        if length == 0:
            return False
        # Если индекс меньше длинны и больше или равно нулю, то True,
иначе - False
        return 0 <= index < length

    # вставка в начало
    def push_head(self, data: T) -> None:
        # создаем новый Узел с ссылкой на прошлый начальный Узел
        self.head = Node(data, self.head)
        # Если список пуст, то к хвосту присваивается тот же новый
Узел
        if self.tail is None:
            self.tail = self.head

    # вставка в конец
    def push_tail(self, data: T) -> None:
        # Если список не пуст, то добавляем к хвосту ссылку и
назначаем новых хвост на новый Узел
        if self.head is not None:
            self.tail.next = Node(data, None)
            self.tail= self.tail.next
        else:
            # иначе если пуст, то начало и конец равны
            self.head = Node(data, None)
            self.tail = self.head

    # вставка по индексу, задаётся индекс туда, там и будет стоять
новый Узел
    def insert(self, index: int, data: T) -> None:
        # если в начало, то воспользуемся методом push_head
        if index == 0:
            self.push_head(data)
            return
        elif index == self.get_length():
            # если в конец, то воспользуемся методом push_tail
            self.push_tail(data)
            return
        elif not self.check_range(index):
            # выдаёт ошибку, если индекс вне диапазона списка
```

```
            raise IndexError("Index is outside the range of the
SingleLinkedList")

        node = self.head
        for _ in range(index - 1):
            node = node.next
        # добавляем новый Узел
        node.next = Node(data, node.next)

    # получить Узел по индексу
    def get_node(self, index: int) -> Optional[Node[T]]:
        if not self.check_range(index):
            return None
        # поиск значения
        node = self.head
        for _ in range(index):
            node = node.next
        return node

    # получить значение по индексу
    def get(self, index: int) -> Optional[T]:
        ok = self.get_node(index)
        if ok is not None:
            return ok.data


    # удаление Узла по индексу
    def remove(self, index: int) -> None:
        # проверка индекса
        if not self.check_range(index):
            raise IndexError("Index is outside the range of the
SingleLinkedList")
        elif index == 0:
            # Если в начало, то сохраняем ссылку головы в саму голову
            self.head = self.head.next
            # Если же список теперь пуст, то в хвосте тоже должно быть
пусто
            if self.head == None:
                self.tail = None
            return

        # удаление Узла по индексу
        node = self.head
        for _ in range(index - 1):
            node = node.next

        # Если удаляется конечный Узел (хвост), то нужно перезаписать
хвост на предпоследний Узел
        if index == self.get_length() - 1:
            self.tail = node
        # Присваиваем к Узлу по index - 1 ссылку на Узел по index + 1
```

```
        node.next = node.next.next

    # очистка (пустой список)
    def clear(self) -> None:
        self.head = None
        self.tail = None

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

    # магический метод вывода класса односвязного списка 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.

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

Класс **Node** – это Узел для Односвязного списка.

Класс принимает два значения – data (любые данные) и next (ссылка на следующий Узел).

Класс **SimpleLinkedList** – это Односвязный список, хранящий такие методы как:

- **__init__()** – имеет две основные переменные – head (начальный Узел) и tail (конечный Узел);
- **get_length()** – возвращает длину Односвязного списка;
- **check_range(index)** – возвращает True если индекс есть в Списке, False – Нету;
- **push_head(data)** – вставка в начало;
- **push_tail(data)** – вставка в конец;
- **insert(index, data)** – вставка по индексу, задаётся индекс там, где и будет стоять новый Узел;
- **get_node(index)** – получить Узел по индексу;
- **get(index)** – получить значение по индексу;
- **remove(index)** – удаление Узла по индексу;
- **clear()** – очистка (пустой список);
- **__iter__()** – магический метод итерирования - возвращает Узел;
- **__str__()** – магический метод вывода класса односвязного списка в формате [знач1, знач2, ... знач N];
- **__repr__()** – магический метод предназначен для машинно-ориентированного вывода;

Путь файла Минимальной кучи – structuredata/minheap.py

```
from temp.car import element, Car
import json
from typing import Generic, Optional, TypeVar
from .simplelinkedlists import SimpleLinkedList, Node

T = TypeVar("T")


# Минимальная куча с реализацией на Односвязном списке
class MinHeap(Generic[T]):
    # Хранит Односвязный список
    def __init__(self) -> None:
        self.list: SimpleLinkedList[Optional[Node[T]]] =
SimpleLinkedList()

    # Возвращает количество узлов в Односвязном списке
    def get_size(self) -> int:
        return self.list.get_length()
```

```
    # Возвращает индекс(позицию) родителя по индексу
    @staticmethod
    def get_index_parent(index: int) -> int:
        # index = 2i + 2 (справа) иначе indez = 2i + 1 (слева)
        if index % 2 == 0:
            return (index - 2) // 2
        return (index - 1) // 2

    # Возвращает значение узла по индексу
    def get(self, index: int) -> Optional[T]:
        return self.list.get(index)

    # Меняет местами узлы по индексам (меняет их ссылки - next)
    def swap(self, index1: int, index2: int) -> None:
        # индексы родителей
        preind1 = index1 - 1
        preind2 = index2 - 1

        # если индексы равны или индекс отрицательный или индекс вышел
из диапазона то ничего не меняется
        if index1 == index2 or index1 < 0 or index2 < 0 or max(index1,
index2) > self.get_size() - 1:
            return

        # Узлы по индексу
        node_1: Optional[Node[T]] = self.list.get_node(index1)
        node_2: Optional[Node[T]] = self.list.get_node(index2)

        # Если первый индекс это голова (начальный узел) - то заменяем
голову на другой узел
        if preind1 < 0:
            pre_node_2: Optional[Node[T]] =
self.list.get_node(preind2)
            pre_node_2.next = node_1
            node_1.next, node_2.next = node_2.next, node_1.next
            self.list.head = node_2
        elif preind2 < 0:
            # Если второй индекс это голова (начальный узел) - то
заменяем голову на другой узел
            pre_node_1: Optional[Node[T]] =
self.list.get_node(preind1)
            pre_node_1.next = node_2
            node_1.next, node_2.next = node_2.next, node_1.next
            self.list.head = node_1
        else:
            pre_node_1: Optional[Node[T]] =
self.list.get_node(preind1)
            pre_node_2: Optional[Node[T]] =
self.list.get_node(preind2)
```

```
            pre_node_1.next = node_2
            pre_node_2.next = node_1
            node_1.next, node_2.next = node_2.next, node_1.next

        # Если индекс это хвост (конец) - то меняем на другой узел
        if self.get_size() - 1 == index2:
            self.list.tail = node_1
        elif self.get_size() - 1 == index1:
            self.list.tail = node_2
        return

    def heapify(self) -> None:
        # Количество всех родительских веток (ветки, которые имеют
одного или двух детей)
        step = self.get_index_parent(self.get_size() - 1) + 1
        save = 0
        if step - 1 < 0:
            return
        while save != step:
            save = 0
            for i in range(step):
                # индексы (позиции) левого и правого
                left = 2 * i + 1
                right = 2 * i + 2
                if self.get(left) is None:
                    min_ind_data = right
                elif self.get(right) is None:
                    min_ind_data = left
                else:
                    min_ind_data = min([left, right], key=self.get)

                if self.get(min_ind_data) < self.get(i):
                    self.swap(min_ind_data, i)
                else:
                    save += 1

    # Вставка в конец и применяется heapify для упорядочивании в
соответствии со свойствами кучи
    def insert(self, data: T) -> None:
        self.list.push_tail(data)
        self.heapify()

    # Удаляет минимальное значение (корень кучи) и применяется heapify
для упорядочивании в соответствии со свойствами кучи
    def remove(self) -> Optional[T]:
        if self.get_size() == 0:
            return None
        delnode = self.list.head.data
        self.swap(0, self.get_size() - 1)
        self.list.remove(self.get_size() - 1)
        self.heapify()
```

```
        return delnode

    # Возвращает корень кучи (минимальное значение)
    def peek(self) -> Optional[T]:
        if self.get_size() == 0:
            return None
        return self.list.head.data

    # Проверяет есть ли такой элемент (возвращает True если да, иначе
False)
    def find(self, data: T) -> bool:
        if self.get_size() == 0:
            return False
            # Возвращает индекс по значению узла
        for it in self.list:
            if it.data == data:
                return True
        return False

    # Сохранение состояния Двоичного дерева поиска в json формате
    def save(self, path: str) -> None:
        with open(path, "w", encoding="UTF-8") as file_save:
            json.dump(self.list.head, file_save, ensure_ascii=False,
default=str)

    # Загрузка состояния Двоичного дерева поиска в json формате

    def load(self, path: str) -> None:
        with open(path, "r", encoding="UTF-8") as file_load:
            self.list.clear()
            self.list.head = eval(json.load(file_load))
            self.list.tail = self.list.get_node(self.get_size() - 1)

    # Магический метод вывода кучи
    def __str__(self) -> str:
        return self.list.__str__()
```

Импортируем класс Односвязного списка в Минимальной куче, для её использования.

Минимальная куча хранит в себе односвязный список.

Методы кучи:

- **get_size()** - возвращает количество узлов в Односвязном списке;
- **статический get_index_parent(index)** - возвращает индекс(позицию) родителя по индексу;
- **get(index)** - возвращает значение узла по индексу;
- **swap(index1, index2)** - меняет местами узлы по индексам (меняет их ссылки - next);

- **heapify()** - количество всех родительских веток (ветки, которые имеют одного или двух детей);
- **insert(data)** - вставка в конец и применяется heapify для упорядочивании в соответствии со свойствами кучи;
- **remove()** - удаляет минимальное значение (корень кучи) и применяется heapify для упорядочивании в соответствии со свойствами кучи;
- **peek()** - возвращает корень кучи (минимальное значение);
- **find(data)** - проверяет есть ли такой элемент (возвращает True если да, иначе False);
- **save(path)** - сохранение состояния Двоичного дерева поиска в json формате;
- **load(path)** - загрузка состояния Двоичного дерева поиска в json формате;
- **__str__()** - магический метод вывода кучи.

5.1 Тесты

Код теста:

```
from structuredata.minheap import MinHeap
from temp.car import Car, element
import os.path

ex_car1 = Car("Haval 0.2v", "LGUFF4A59HF507891", 7, 3_890_000, 88)
same_cost = Car("Haval 0.3v", "LHHFF4G67HF777001", 5, 3_890_000, 180)
heap = MinHeap()
lheaply = [
    1800500, 2044000, 2000000, 2104000, 2111000,
    3003000, 4003900.5, 3060000, 2500000, 3000000
    ]

# вставка
def test_insert():
    for i in element:
        heap.insert(i)
    for it in element:
        assert heap.find(it)

# вставка с таким же ключём и проверка размера
def test_insert_same_and_size():
    heap.insert(ex_car1)
    heap.insert(same_cost)
    assert heap.get_size() == len(element) + 2

# удаление
def test_remove():
    m = element + [ex_car1] + [same_cost]
    for it in m:
```

```
        assert heap.remove() in m
    assert heap.get_size() == 0

# сохранение состояния в файл (проверка существует ли файл)
def test_save():
    for i in element:
        heap.insert(i)
    path = "test_json2.json"
    heap.save(path)
    assert os.path.exists(path)

# загрузка файла в пустое дерево
def test_load():
    path = "test_json2.json"
    orig = MinHeap()
    orig.list = heap.list
    heap.list.clear()
    heap.load(path)
    assert orig.list == heap.list

# проверка нахождения ключа
def test_find():
    assert heap.find(element[1])
    assert not(heap.find(ex_car1))

# Проверка на максимальное значение и минимальное
def test_heapify():
    for i in range(len(element)):
        assert heap.get(i).cost == lheaply[i]
```

Результат пройденных тестов (рисунок 5.1).

```
PS D:\Saves\GUAP\SEM3\AuSD\laba3> pytest test_10.py
====================================== test session starts ======================================
platform win32 -- Python 3.12.7, pytest-8.3.3, pluggy-1.5.0
rootdir: D:\Saves\GUAP\SEM3\AuSD\laba3
collected 7 items

test_10.py .......                                                                      [100%]

======================================= 7 passed in 0.02s =======================================
```

Рисунок 5.1 – результат проверки теста 10 задания

Все 7 тестов были пройдены успешно.

5.2 Бенчмарк

Был произведён бенчмарк на вместимость и количество значений.

Код программы:

```
from structuredata.minheap import MinHeap
```

```
from temp.car import Car, element
from timeit import timeit
from random import choice, randint

times1 = ""
times2 = ""

for su in range(1, 4):
    table = MinHeap()
    # колличество элементов
    k1 = 10 ** su
    dit = []
    marks = [i.mark for i in element]
    for i in range(k1):
        dit += [Car(
            mark=choice(marks),
            vin="".join(choice("0123456789ABCDEFGHJKLMNPRSTUVWXYZ")
for i in range(17)),
            engine_capacity=randint(5_00, 40_00)/100,
            cost=randint(900_000 * 100, 6_000_000 * 100)/100,
            average_speed=randint(70_00, 250_00)/100
        )]

    code1 = """for x in dit:
        table.insert(x)
    """
    code2 = """for x in dit:
        table.remove()
    """
    time1 = timeit(code1, number=1, globals={"dit": dit, "table":
table})
    time2 = timeit(code2, number=1, globals={"dit": dit, "table":
table})
    times1 += f"        {k1}            |          {time1}\n"
    times2 += f"        {k1}            |        {time2}\n"

print("Вставка")
print(f"Кол-во елементов    |   Время")
print(times1)
print("Удаление")
print(f"Кол-во елементов    |   Время")
print(times2)
```

Вывод в консоли изображено на рисунке 5.2.

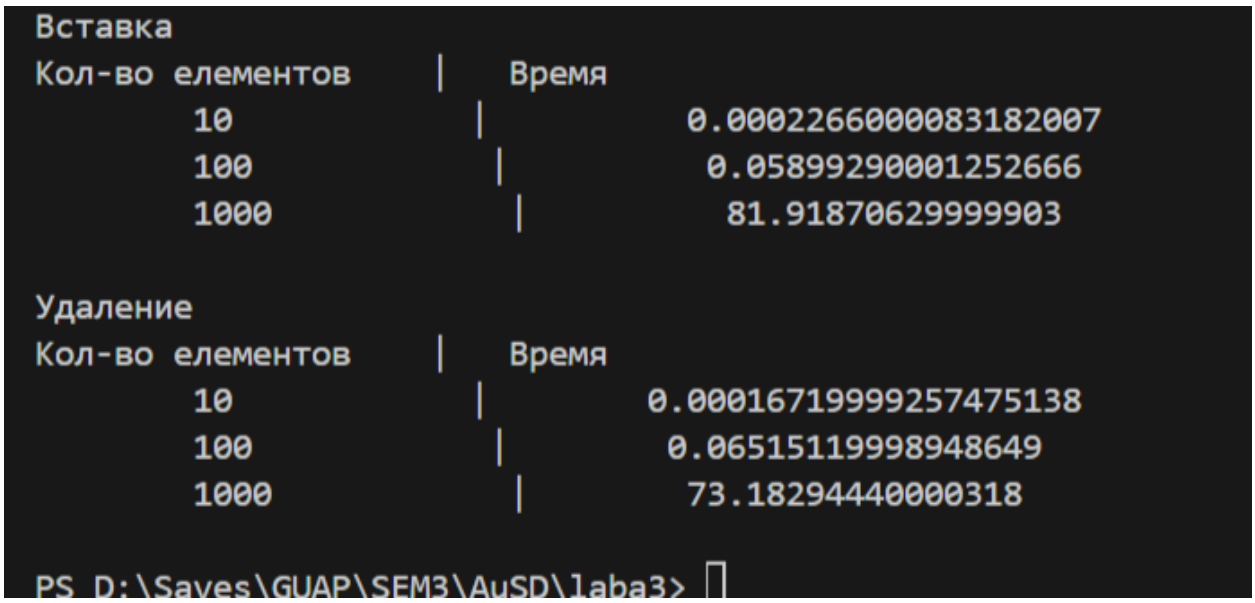

Рисунок 5.2 – вывод в консоли времени в секунды

Можно заметить, что количество элементов сильно влияет на время обработки Минимальной кучи. Причина этому постоянно вызывающий метод heapify(), который упорядочивает кучу бесконечным циклом, поэтому чем больше элементов тот имеет, чем дольше будет обрабатываться куча и добавка с удалением будет работать медленнее.

Кол-во элементов (вместимость)	Время
10	0.0002266000083182007
100	0.05899290001252666
1000	81.91870629999903

Таблица 5.1 – Вставка в Минимальной куче

Кол-во элементов (вместимость)	Время
10	0.00016719999257475138
100	0.06515119998948649
1000	73.18294440000318

Таблица 5.2 – Удаление в Минимальной куче

Выводы

В данной лабораторной работе я создала такие структуры данные как:

- Односвязный список
- Двоичное дерево поиска
- Минимальная куча

Научилась применять аннотацию типов с помощью библиотеки typing. А также провела тест своих структур данных вместе с бенчмарком.