ГУАП

КАФЕДРА № 41

ОТЧЕТ
ЗАЩИЩЕН С ОЦЕНКОЙ

ПРЕПОДАВАТЕЛЬ

старший преподаватель		Н. А. Соловьева
должность, уч. степень, звание	подпись, дата	инициалы, фамилия

ОТЧЕТ О ЛАБОРАТОРНОЙ РАБОТЕ

СТРУКТУРЫ ДАННЫХ: МАССИВ, СПИСОК, ХЕШ-ТАБЛИЦА, МНОЖЕСТВО, СТЕК

по курсу: АЛГОРИТМЫ И СТРУКТУРЫ ДАННЫХ

РАБОТУ ВЫПОЛНИЛ

СТУДЕНТ ГР. №	4314		Д. М. Развеев
		подпись, дата	инициалы, фамилия

Санкт-Петербург 2024

1. **Цель работы** научиться реализовывать такие базовые структуры данных, как: очередь, дерево и куча.

2. **Индивидуальное задание (задания 7, 11)**
   a. Реализуйте структуру данных «АВЛ-дерево», элементами которой выступают экземпляры класса Student (минимум 10 элементов), содержащие следующие поля (ФИО, номер группы, курс, возраст, средняя оценка за время обучения), где в качестве ключевого элемента при добавлении будет выступать средняя оценка. 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.
   b. Реализуйте структуру данных «Максимальная куча» на основе двусвязного списка, элементами которой выступают экземпляры класса Student (минимум 10 элементов), содержащие следующие поля (ФИО, номер группы, курс, возраст, средняя оценка за время обучения), где в качестве ключевого элемента при добавлении будет выступать средняя оценка. 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.

4. **Ход работы**

Начнем с выполнения задания 7 Реализация:

```
import time
from typing import Optional, List

class Student:
    def __init__(self, full_name: str, group_number: str, course: int, age: int,
average_grade: float) -> None:
        self.full_name = full_name
        self.group_number = group_number
        self.course = course
        self.age = age
        self.average_grade = average_grade

    def __repr__(self) -> str:
        return f"Student({self.full_name}, {self.group_number}, {self.course},
{self.age}, {self.average_grade})"

class AVLNode:
    def __init__(self, student: Student) -> None:
        self.student = student
        self.left: Optional[AVLNode] = None
        self.right: Optional[AVLNode] = None
        self.height: int = 1

class AVLTree:
    def __init__(self) -> None:
        self.root: Optional[AVLNode] = None
```

```
    def _height(self, node: Optional[AVLNode]) -> int:
        return node.height if node else 0

    def _balance_factor(self, node: Optional[AVLNode]) -> int:
        return self._height(node.left) - self._height(node.right)

    def _rotate_right(self, y: AVLNode) -> AVLNode:
        x = y.left
        if x is None:
            return y  # No rotation possible
        T2 = x.right
        x.right = y
        y.left = T2
        y.height = 1 + max(self._height(y.left), self._height(y.right))
        x.height = 1 + max(self._height(x.left), self._height(x.right))
        return x

    def _rotate_left(self, x: AVLNode) -> AVLNode:
        y = x.right
        if y is None:
            return x  # No rotation possible
        T2 = y.left
        y.left = x
        x.right = T2
        x.height = 1 + max(self._height(x.left), self._height(x.right))
        y.height = 1 + max(self._height(y.left), self._height(y.right))
        return y

    def _insert(self, node: Optional[AVLNode], student: Student) -> AVLNode:
        if not node:
            return AVLNode(student)

        if student.average_grade < node.student.average_grade:
            node.left = self._insert(node.left, student)
        else:
            node.right = self._insert(node.right, student)

        node.height = 1 + max(self._height(node.left), self._height(node.right))
        balance = self._balance_factor(node)

        if balance > 1 and student.average_grade <
node.left.student.average_grade:
            return self._rotate_right(node)

        if balance < -1 and student.average_grade >
node.right.student.average_grade:
            return self._rotate_left(node)

        if balance > 1 and student.average_grade >
node.left.student.average_grade:
            node.left = self._rotate_left(node.left)
```

```
            return self._rotate_right(node)

        if balance < -1 and student.average_grade <
node.right.student.average_grade:
            node.right = self._rotate_right(node.right)
            return self._rotate_left(node)

        return node

    def insert(self, student: Student) -> None:
        self.root = self._insert(self.root, student)

    def _find_min(self, node: AVLNode) -> AVLNode:
        if node.left is None:
            return node
        return self._find_min(node.left)

    def _delete(self, node: Optional[AVLNode], average_grade: float) ->
Optional[AVLNode]:
        if not node:
            return node

        if average_grade < node.student.average_grade:
            node.left = self._delete(node.left, average_grade)
        elif average_grade > node.student.average_grade:
            node.right = self._delete(node.right, average_grade)
        else:
            if not node.left:
                return node.right
            elif not node.right:
                return node.left
            temp = self._find_min(node.right)
            node.student = temp.student
            node.right = self._delete(node.right, temp.student.average_grade)

        node.height = 1 + max(self._height(node.left), self._height(node.right))
        balance = self._balance_factor(node)

        if balance > 1 and self._balance_factor(node.left) >= 0:
            return self._rotate_right(node)

        if balance < -1 and self._balance_factor(node.right) <= 0:
            return self._rotate_left(node)

        if balance > 1 and self._balance_factor(node.left) < 0:
            node.left = self._rotate_left(node.left)
            return self._rotate_right(node)

        if balance < -1 and self._balance_factor(node.right) > 0:
            node.right = self._rotate_right(node.right)
            return self._rotate_left(node)
```

```
        return node

    def delete(self, average_grade: float) -> None:
        self.root = self._delete(self.root, average_grade)

    def save_to_file(self, filename: str) -> None:
        with open(filename, 'w') as f:
            self._save_recursive(self.root, f)

    def _save_recursive(self, node: Optional[AVLNode], file) -> None:
        if node:
            file.write(f"{node.student.full_name},{node.student.group_number},{no
de.student.course},"
                       f"{node.student.age},{node.student.average_grade}\n")
            self._save_recursive(node.left, file)
            self._save_recursive(node.right, file)

    def load_from_file(self, filename: str) -> None:
        with open(filename, 'r') as f:
            for line in f:
                full_name, group_number, course, age, average_grade =
line.strip().split(',')
                student = Student(full_name, group_number, int(course), int(age),
float(average_grade))
                self.insert(student)

    def search(self, average_grade: float) -> bool:
        return self._search(self.root, average_grade)

    def _search(self, node: Optional[AVLNode], average_grade: float) -> bool:
        if not node:
            return False
        if node.student.average_grade == average_grade:
            return True
        elif average_grade < node.student.average_grade:
            return self._search(node.left, average_grade)
        else:
            return self._search(node.right, average_grade)

    def search_via_inorder(self, average_grade: float) -> bool:
        return self._search_via_inorder(self.root, average_grade)

    def _search_via_inorder(self, node: Optional[AVLNode], average_grade: float)
-> bool:
        if node:
            found = self._search_via_inorder(node.left, average_grade)
            if found:
                return True
            if node.student.average_grade == average_grade:
                return True
```

```
            return self._search_via_inorder(node.right, average_grade)
        return False

def benchmark() -> None:
    q = AVLTree()
    start_time = time.time()

    # Вставка 1000 студентов
    for i in range(1000):
        q.insert(Student(f"Student {i}", str(i), 1, 18, 4.0 + (i % 10) * 0.1))
    print(f"Time to push 1000 students: {time.time() - start_time:.6f} seconds")

    # Удаление 1000 студентов
    start_time = time.time()
    for i in range(1000):
        q.delete(4.0 + (i % 10) * 0.1)  # Удаляем студентов с этими средними
оценками
    print(f"Time to delete 1000 students: {time.time() - start_time:.6f}
seconds")

def run_tests() -> None:
    tree = AVLTree()

    # Тест 1: Вставка студентов
    student1 = Student("Иванов Иван", "Группа 1", 1, 18, 4.5)
    student2 = Student("Петров Петр", "Группа 2", 2, 19, 3.5)
    student3 = Student("Сидоров Сидор", "Группа 3", 3, 20, 4.0)

    tree.insert(student1)
    tree.insert(student2)
    tree.insert(student3)

    assert tree.search(4.5) == True
    assert tree.search(3.5) == True
    assert tree.search(4.0) == True
    assert tree.search(5.0) == False  # Студент с этой оценкой не должен быть
найден

    # Тест 2: Удаление студента
    tree.delete(3.5)
    assert tree.search(3.5) == False  # Удалили студента с оценкой 3.5

    # Тест 3: Сохранение и загрузка
    tree.save_to_file('test_students.txt')
    new_tree = AVLTree()
    new_tree.load_from_file('test_students.txt')
    assert new_tree.search(4.5) == True
    assert new_tree.search(4.0) == True
    assert new_tree.search(3.5) == False  # После удаления не должен быть найден

    # Тест 4: Проверка балансировки
```

```
    for i in range(100):
        tree.insert(Student(f"Student {i}", str(i), 1, 18, 4.0 + (i % 10) * 0.1))
    assert tree.root is not None  # Дерево не должно быть пустым

    # Тест 5: Проверка на несуществующую оценку
    assert tree.search(10.0) == False  # Оценка 10.0 не должна существовать

if __name__ == "__main__":
    benchmark()
    run_tests()
```

Результаты бенчмаркинга:

```
Time to push 1000 students: 0.021003 seconds
Time to delete 1000 students: 0.002002 seconds
```

Описание кода

1. **Классы**:
   - **Student**: Представляет студента с полями для полного имени, номера группы, курса, возраста и средней оценки. Содержит метод для представления информации о студенте.
   - **AVLNode**: Узел AVL-дерева, который содержит экземпляр `Student`, ссылки на левого и правого потомков, а также высоту узла.
   - **AVLTree**: Основной класс, представляющий AVL-дерево. Содержит методы для вставки, удаления и поиска студентов, а также методы для балансировки дерева.
2. **Методы AVLTree**:
   - `_insert`: Рекурсивно добавляет студента в дерево и выполняет балансировку при необходимости.
   - `_delete`: Удаляет студента с указанной средней оценкой, также поддерживая балансировку.
   - `search`: Проверяет наличие студента с указанной средней оценкой.
   - `save_to_file`: Сохраняет данные студентов в текстовый файл в формате CSV.
   - `load_from_file`: Загружает данные студентов из текстового файла и добавляет их в дерево.
3. **Функции для тестирования**:
   - **benchmark**: Измеряет время вставки и удаления студентов в/из дерева.
   - **run_tests**: Выполняет набор тестов, проверяющих основные функции дерева, такие как вставка, удаление, сериализация и десериализация.

Принцип работы

Код создает структуру данных, которая позволяет эффективно управлять информацией о студентах, обеспечивая быструю вставку, удаление и поиск по средней оценке. AVL-дерево автоматически поддерживает балансировку, что гарантирует логарифмическое время выполнения операций. Сохранение и загрузка данных из текстовых файлов позволяет сохранять состояние структуры между сессиями.

Выполнение задания 11 Реализация:

```python
import time
from typing import Optional

class Student:
    def __init__(self, full_name: str, group_number: str, course: int, age: int,
average_grade: float) -> None:
        self.full_name = full_name
        self.group_number = group_number
        self.course = course
        self.age = age
        self.average_grade = average_grade

    def __repr__(self) -> str:
        return f"Student({self.full_name}, {self.group_number}, {self.course},
{self.age}, {self.average_grade})"

class Node:
    def __init__(self, student: Student) -> None:
        self.student = student
        self.prev: Optional['Node'] = None
        self.next: Optional['Node'] = None

class MaxHeap:
    def __init__(self) -> None:
        self.head: Optional[Node] = None
        self.tail: Optional[Node] = None
        self.size: int = 0

    def _parent_index(self, index: int) -> int:
        return (index - 1) // 2

    def _left_index(self, index: int) -> int:
        return 2 * index + 1

    def _right_index(self, index: int) -> int:
        return 2 * index + 2

    def _swap(self, node1: Node, node2: Node) -> None:
        node1.student, node2.student = node2.student, node1.student

    def insert(self, student: Student) -> None:
        new_node = Node(student)
        if not self.head:
            self.head = new_node
            self.tail = new_node
        else:
            self.tail.next = new_node
            new_node.prev = self.tail
            self.tail = new_node
            self._heapify_up(self.size)
```

```
        self.size += 1

    def _heapify_up(self, index: int) -> None:
        current_index = index
        while current_index > 0:
            parent_index = self._parent_index(current_index)
            if self._get_student_at_index(current_index).average_grade >
self._get_student_at_index(parent_index).average_grade:
                self._swap(self._get_node_at_index(current_index),
self._get_node_at_index(parent_index))
                current_index = parent_index
            else:
                break

    def extract_max(self) -> Optional[Student]:
        if self.size == 0:
            return None

        max_student = self.head.student
        if self.size == 1:
            self.head = None
            self.tail = None
        else:
            self.head = self.head.next
            if self.head:
                self.head.prev = None

        self.size -= 1
        if self.size > 0:
            self._heapify_down(0)

        return max_student

    def _heapify_down(self, index: int) -> None:
        current_index = index
        while current_index < self.size:
            left_index = self._left_index(current_index)
            right_index = self._right_index(current_index)
            largest_index = current_index

            if left_index < self.size and
self._get_student_at_index(left_index).average_grade >
self._get_student_at_index(largest_index).average_grade:
                largest_index = left_index

            if right_index < self.size and
self._get_student_at_index(right_index).average_grade >
self._get_student_at_index(largest_index).average_grade:
                largest_index = right_index

            if largest_index != current_index:
```

```
                self._swap(self._get_node_at_index(current_index),
self._get_node_at_index(largest_index))
                current_index = largest_index
            else:
                break

    def _get_student_at_index(self, index: int) -> Student:
        current = self.head
        for _ in range(index):
            current = current.next
        return current.student

    def _get_node_at_index(self, index: int) -> Node:
        current = self.head
        for _ in range(index):
            current = current.next
        return current

    def save_to_file(self, filename: str) -> None:
        with open(filename, 'w', encoding='utf-8') as f:
            current = self.head
            while current:
                f.write(f"{current.student.full_name},{current.student.group_numb
er},{current.student.course},"
                        f"{current.student.age},{current.student.average_grade}\
n")
                current = current.next

    def load_from_file(self, filename: str) -> None:
        with open(filename, 'r', encoding='utf-8') as f:
            for line in f:
                full_name, group_number, course, age, average_grade =
line.strip().split(',')
                student = Student(full_name=full_name, group_number=group_number,
                                  course=int(course), age=int(age),
                                  average_grade=float(average_grade))
                self.insert(student)

    def search(self, average_grade: float) -> bool:
        current = self.head
        while current:
            if current.student.average_grade == average_grade:
                return True
            current = current.next
        return False

def run_tests() -> None:
    heap = MaxHeap()

    # Тест 1: Вставка студентов
    heap.insert(Student("Иванов Иван", "Группа 1", 1, 18, 4.5))
```

```
    heap.insert(Student("Петров Петр", "Группа 2", 2, 19, 3.5))
    heap.insert(Student("Сидоров Сидор", "Группа 3", 3, 20, 4.0))

    assert heap.search(4.5) == True
    assert heap.search(3.5) == True
    assert heap.search(4.0) == True
    assert heap.search(5.0) == False  # Не найден студент с оценкой 5.0

    # Тест 2: Извлечение максимального
    max_student = heap.extract_max()
    assert max_student.full_name == "Иванов Иван"  # Максимальная оценка 4.5

    # Тест 3: Извлечение после удаления
    second_max_student = heap.extract_max()
    assert second_max_student.full_name == "Сидоров Сидор"  # Следующий по
максимальной оценке

    # Тест 4: Проверка состояния кучи
    assert heap.size == 1  # Оставшийся студент
    assert heap.search(3.5) == True  # Остался студент с оценкой 3.5

    # Тест 5: Сохранение и загрузка
    heap.save_to_file('test_students2.txt')
    new_heap = MaxHeap()
    new_heap.load_from_file('test_students2.txt')

    # Проверяем, что студенты загружены правильно
    assert new_heap.search(4.5) == False
    assert new_heap.search(4.0) == False
    assert new_heap.search(3.5) == True  # Проверяем, что 3.5 остался
    assert new_heap.search(5.0) == False  # Не найден студент с оценкой 5.0

def benchmark() -> None:
    heap = MaxHeap()
    start_time = time.time()

    # Вставка 1000 студентов
    for i in range(1000):
        heap.insert(Student(f"Student {i}", f"Group {i}", (i % 10) + 1, 18 + (i %
5), 4.0 + (i % 10) * 0.1))
    print(f"Time to push 1000 students: {time.time() - start_time:.6f} seconds")

    # Извлечение 1000 студентов
    start_time = time.time()
    for _ in range(1000):
        heap.extract_max()
    print(f"Time to extract 1000 students: {time.time() - start_time:.6f}
seconds")

if __name__ == "__main__":
    benchmark()
```

```
run_tests()
```

Результаты бенчмаркинга:

```
Time to push 1000 students: 0.019002 seconds
Time to delete 1000 students: 0.002003 seconds
```

Описание кода

1. **Класс `Student`**:
   - Представляет студента с полями: ФИО, номер группы, курс, возраст и средняя оценка.
   - Реализует метод `__repr__` для удобного отображения информации о студенте.
2. **Класс `Node`**:
   - Узел двусвязного списка, содержащий экземпляр `Student` и ссылки на предыдущий и следующий узлы.
3. **Класс `MaxHeap`**:
   - Реализует структуру данных «максимальная куча» с использованием двусвязного списка.
   - Поддерживает операции:
     - `insert(student)`: Вставляет нового студента в кучу и поддерживает свойство максимальной кучи.
     - `extract_max()`: Извлекает студента с максимальной средней оценкой.
     - `search(average_grade)`: Проверяет наличие студента с указанной средней оценкой.
     - `save_to_file(filename)`: Сохраняет студентов в текстовый файл в формате CSV (значения разделены запятыми).
     - `load_from_file(filename)`: Загружает студентов из текстового файла.
4. **Функция `run_tests()`**:
   - Проводит несколько тестов, включая вставку студентов, извлечение максимального, проверку состояния кучи и сохранение/загрузку из файла.
5. **Функция `benchmark()`**:
   - Измеряет время, необходимое для вставки и извлечения 1000 студентов из кучи.
6. **Главный блок `if __name__ == "__main__":`**:
   - Вызывает функции `benchmark()` и `run_tests()` для тестирования и производительности кучи.

Вывод:
Были реализованы 2 структуры данных: максимальная куча на основе двусвязного списка и АВЛ дерево с возможностью сохранения в файл и выгрузки из него.
Преимущества реализации максимальной кучи на основе двусвязного списка включают простоту вставки и удаления элементов, а недостатки — повышенные затраты памяти на хранение дополнительных указателей и меньшую эффективность по времени в сравнении с массивной реализацией из-за необходимости линейного поиска.
Преимущества данной реализации AVL-дерева включают автоматическую балансировку для поддержания логарифмической сложности операций вставки и удаления, а недостатками являются сложность кода и дополнительные затраты на поддержание высоты узлов, что может привести к увеличению времени выполнения в некоторых случаях.