Частное учреждение профессионального образования
«Высшая школа предпринимательства» (ЧУПО
«ВШП»)

ОТЧЕТ ПО ПРАКТИКЕ

по основной образовательной программе
среднего профессионального образования по специальности
09.02.07 «Информационные системы и программирование»

Вид практики (учебная, производственная, преддипломная):

Установленный по КУГ срок прохождения практики: с _____.___.20__г. по 20__г.

Место прохождения практики (наименование организации):

_____

_____

_____

Выполнил студент _-го курса

_____ *(подпись)*

_____ *(фамилия, имя, отчество)*

Руководитель от образовательной организации

_____ *(подпись)*

_____ *(ученая степень, фамилия, имя, отчество)*

_____ *(должность)*

Руководитель от предприятия

_____ *(подпись)*

_____ *(ученая степень, фамилия, имя, отчество)*

_____ *(должность)*

Оценка

_____ *(прописью)*

Дата сдачи отчета: __.__. 20__г.

Оглавление

1 - Массивы и связные списки

1.1 Статический массив..... 5

1.2 Динамический массив..... 6

1.3 Односвязный список..... 7

1.4 Двусвязный список..... 8

2 - Стек и очередь

2.5 Стек..... 9

2.6 Очередь..... 10

2.7 Задача “Калькулятор”..... 12

3 - Map, HashMap, хэш-функции и коллизии

3.8 Своя хэш-таблица..... 14

3.9 Частотный словарь..... 16

3.10 Trie + HashMap: автодополнение..... 18

4 - Деревья и графы

4.11 Бинарное дерево поиска (BST)..... 20

4.12 Trie (углубление)..... 22

4.13 Графы..... 24

4.14 Задача “Острова”..... 26

5 - Куча и приоритетные очереди

5.15 Куча..... 27

5.16. Приоритетная очередь..... 29

Введение:

В рамках данной практики были выполнены 16 заданий, охватывающих ключевые темы:

- линейные структуры (массивы, связные списки),
- абстрактные типы данных (стек, очередь),
- ассоциативные структуры (хэш-таблицы, Trie),
- иерархические структуры (деревья, кучи),
- а также графы и алгоритмы их обработки.

Особое внимание уделено не только корректной реализации операций (вставка, удаление, поиск, обход), но и оценке их трудоёмкости, сравнительному анализу различных подходов (например, статический и динамический массив, список и массив, очередь на массиве и очередь на стеках), а также применению структур данных для решения практических задач — таких как проверка скобочных последовательностей, вычисление арифметических выражений методом ОПН, поиск компонент связности («острова»), автодополнение по префиксу, планирование задач и другие.

Все структуры данных реализованы «с нуля» на языке Python, что позволило глубже понять их внутреннее устройство, механизмы управления памятью и алгоритмическую суть.

Данный отчёт содержит описание выполненных заданий, приведен исходный код, представлены результаты тестирования и замеров времени, а также сделаны выводы по эффективности и применимости каждой структуры данных в зависимости от контекста задачи.

Цель практики:

Целью учебной практики «Структуры данных» является формирование глубокого теоретического понимания и практических навыков работы с фундаментальными структурами данных, лежащими в основе современного программирования и разработки эффективных алгоритмов.

В ходе выполнения практики нужно освоить принципы организации, хранения, обработки и управления данными с использованием таких структур, как массивы (статические и динамические), связные списки (одно- и двусвязные), стеки, очереди, хэш-таблицы, деревья (включая бинарные деревья поиска и префиксные деревья Trie), графы, а также кучи и приоритетные очереди. Особое внимание уделяется анализу временной и пространственной сложности операций, сравнению эффективности различных подходов в зависимости от решаемой задачи, а также реализации алгоритмов «с нуля» без использования высокоуровневых встроенных средств языка программирования.

Практика направлена на развитие алгоритмического мышления, способности выбирать оптимальную структуру данных для конкретной проблемы, а также на приобретение опыта решения практических задач, таких как разбор арифметических выражений с использованием обратной польской нотации, реализация систем автодополнения, поиск компонент связности в графах (задача «острова»), планирование задач на основе приоритетов и построение частотных словарей.

Таким образом, практика служит фундаментом для дальнейшего изучения курсов по алгоритмам, базам данных, операционным системам, компиляторам и разработке высокопроизводительных программных систем.

1 - Массивы и связные списки

Задача - 1.1 Статический массив

Реализовать функции:

● pushBack, pushFront,

● insert(index, value),

● remove(index),

● find(value).

Оценить трудоемкость каждой операции (в комментариях).

input:	**output**:
pushBack(10), pushBack(20): [10, 20]	2, 10
pushFront(5): [5, 10, 20]	3, 10
insert(2, 15)	4, 10
find(15)	2
remove(0): [10, 15, 20]	3, 10

```
    def remove(self, index):  # O(n)
         self._check_index(index)
         # Сдвиг влево
         self.data[index : self.size - 1] = self.data[index + 1 :
         self.size]
         self.data[self.size - 1] = None
         self.size -= 1
```

Цель: реализовать базовые операции на фиксированном по размеру массиве.

Подход:

Использован список Python ([None] * capacity) с ограничением size ≤ capacity.

pushBack - O(1); pushFront, insert, remove - требуют сдвига - O(n).

find - линейный поиск - O(n).

Анализ: прост в реализации, но не гибкий. Операции в начале дорогие.

Задача - 1.2 Динамический массив

1. Реализовать динамический массив с автоматическим расширением (стратегия увеличения ×2).
2. Сравнить время вставки 100000 элементов в статический массив vs динамический (опционально — замеры времени).

Тест:	Размер: 100000 элементов
Динамический массив	0.0321 сек
Статический массив	Переполнен на 1000 элементе Время до ошибки: 0.0007 сек

```
class StaticArray:
    def __init__(self, capacity=1000):
        self.capacity = capacity
        self.size = 0
        self.data = [None] * capacity
    def pushBack(self, value):
        if self.size >= self.capacity:
            raise OverflowError("Массив заполнен")
        self.data[self.size] = value
        self.size += 1
```

Цель: автоматически расширять массив при переполнении.

Подход:

При заполнении - создается новый массив в 2 раза больше, копируются элементы.

pushBack - амортизировано O(1).

Сравнение с массивом: статический не растёт, динамический - гибкий, но с накладными расходами на копирование.

Практика: вставка 100 000 элементов работает без ошибок, в отличие от статического.

Задача - 1.3 Односвязный список

Реализовать:

Вставку в начало/конец, удаление по значению, поиск по значению, разворот списка in-place.

Сравнить операции вставки/удаления с массивом.

Сравнение:	Список:	Массив:	Результат:
Вставка в начало	0.0019 сек	0.0133 сек	Список в 7 раз
Вставка в конец	0.9535 сек	0.0005 сек	Массив в 2035 раз
Удаление	0.4574 сек	0.0787 сек	Список в 0.2 раз

```
class SinglyLinkedList:
    def __init__(self):
        self.head = None
    def pushFront(self, value):
        new_node = ListNode(value, self.head)
        self.head = new_node
```

Цель: реализовать основные операции списка.

Подход:

pushFront - O(1); pushBack - O(n) (т.к. идём до конца).

Удаление и поиск - O(n).

reverse() - in-place, без дополнительной памяти, O(n).

Сравнение с массивом:

Вставка в начало быстрее - O(1) против O(n).

Нет сдвига при удалении - экономия времени.

Но нет произвольного доступа.

Задача - 1.4 Двусвязный список

1. Реализовать: Вставку после произвольного узла, удаление узла без поиска “сначала”.

2. Реализовать: Итератор по двусвязному списку.

Исходный список:	**Вставка**:	**Удаление**:	**Итератор**:
10, 20, 30	10, 20, 25, 30	10, 25, 30	10, 25, 30

```
    def insert_after(self, node, value):
        #Вставка нового узла после заданного узла
        if node is None:
            raise ValueError("Узел не может быть None")
        new_node = DListNode(value, prev=node, next=node.next)
        if node.next:
            node.next.prev = new_node
        node.next = new_node
        if node == self.tail:
            self.tail = new_node
        if self.head == node and self.tail is None:
            self.tail = node
```

Цель: эффективное удаление и вставка произвольного узла.

Подход:

Каждый узел содержит ссылки prev и next.

insert_after(node, value), remove_node(node) - O(1), если узел известен.

Итератор: реализован через __iter__ - можно использовать в for-циклах.

Плюс: идеален для LRU cache, где важно быстро удалять несвязанные узлы.

2 - Стек и очередь

Задача - 2.5 Стек

1. Реализовать стек на: массиве, связном списке.

2. Используя стек, проверить корректность скобочной последовательности.

Проверка скобочных последовательностей:

'()' - True

'()[]{}' - True

'([{}])' - True

'([)]' - False

'(((' - False

')))' - False

'' - True

'{[()]}' - True

```
class Stack_Array: #Массив
    def __init__(self):
        self.data = []
    def push(self, x):
        self.data.append(x)
    def pop(self):
        if self.is_empty():
            raise IndexError("Стек пуст")
        return self.data.pop()
```

Цель: реализовать LIFO-структуру двумя способами.

Подход:

На массиве: используем list.append() и list.pop() - O(1).

На списке: вставка/удаление в голову - O(1).

Применение: проверка скобок (()[]{}).

Открывающие - в стек; закрывающие - сверяем с вершиной.

Время - O(n), память - O(n).

Вывод: стек - идеален для разбора вложенных структур.

Задача - 2.6 Очередь

Реализовать очередь на циклическом массиве.

Реализовать очередь на двух стеках.

```
class QueueTwoStacks:
    def __init__(self):
        self.in_stack = Stack()
        self.out_stack = Stack()
    def enqueue(self, x):
        self.in_stack.push(x)
    def dequeue(self):
        if self.out_stack.is_empty():
            if self.in_stack.is_empty():
                raise IndexError("Очередь пуста")
            while not self.in_stack.is_empty():
                self.out_stack.push(self.in_stack.pop())
        return self.out_stack.pop()
```

Вывод через терминал:

1. Очередь на циклическом массиве (ёмкость = 5):

enqueue(1) → очередь: [1, None, None, None, None], head=0, tail=1, size=1

enqueue(2) → очередь: [1, 2, None, None, None], head=0, tail=2, size=2

enqueue(3) → очередь: [1, 2, 3, None, None], head=0, tail=3, size=3

enqueue(4) → очередь: [1, 2, 3, 4, None], head=0, tail=4, size=4

enqueue(5) → очередь: [1, 2, 3, 4, 5], head=0, tail=0, size=5

Извлечение элементов:

dequeue() → 1, size=4

dequeue() → 2, size=3

dequeue() → 3, size=2

dequeue() → 4, size=1

dequeue() → 5, size=0

2. Очередь на двух стеках:

enqueue(10)

enqueue(11)

enqueue(12)

enqueue(13)

enqueue(14)

enqueue(15)

Извлечение элементов:

dequeue() → 10

dequeue() → 11

dequeue() → 12

dequeue() → 13

dequeue() → 14

dequeue() → 15

Цель: реализовать FIFO двумя способами.

На циклическом массиве:

Используем head, tail, size - enqueue/dequeue - O(1).

Экономит память, не сдвигает элементы.

На двух стеках:

in_stack - для вставки, out_stack - для извлечения.

Амортизировано O(1) на операцию.

Сравнение: циклический массив - быстрее и предсказуемее.

Задача - 2.7 Задача “Калькулятор”

Считать выражение в инфиксной форме.

Используя стек, преобразовать в обратную польскую нотацию (ОПН).

Вычислить результат.

```
def eval_postfix(postfix):
    stack = []
    for token in postfix:
        if isinstance(token, int):
           stack.append(token)
        else:
            if len(stack) < 2:
                raise ValueError("Недостаточно операндов")
            b = stack.pop()
            a = stack.pop()
            if token == '+':
                stack.append(a + b)
            elif token == '-':
                stack.append(a - b)
            elif token == '*':
                stack.append(a * b)
            elif token == '/':
                if b == 0:
                    raise ZeroDivisionError("Деление на ноль")
                stack.append(int(a / b))#усечение к нулю
    if len(stack) != 1:
        raise ValueError("Некорректное выражение")
    return stack[0]
```

Вывод результата в терминал:

3 + 4 * 2 = 11

(3 + 4) * 2 = 14

15 / 3 - 2 = 3

10 - 2 - 3 = 5

2 * (5 + 3) / 4 = 4

100 / 10 / 2 = 5

((2 + 3) * (5 - 1)) / 2 = 10

Цель: вычислить арифметическое выражение.

Этапы:

Токенизация: разбор строки на числа и операторы.

Унарный минус: замена - на u-, если стоит в начале или после (.

Алгоритм Дейкстры (сортировочная станция):

Операторы кладутся в стек с учетом приоритета и ассоциативности.

Скобки управляют порядком.

Вычисление ОПН: стек операндов; при операторе - извлекаем 2 (или 1 для u-), вычисляем.

Особенности: поддержка отрицательных чисел, деление с усечением к нулю (int(a/b)).

Сложность: O(n) по времени и памяти.

Значение: основа всех компиляторов и интерпретаторов.

03 - Map, HashMap, хэш-функции и коллизии

Задача - 3.8 Своя хэш-таблица

1. Реализовать: хэш-функцию для строк, метод разрешения коллизий (цепочки или открытая адресация), функции put(key, value), get(key), remove(key).

2. Визуализировать состояние таблицы после серии вставок.

```
    def _hash(self, key: str) -> int:
        if not isinstance(key, str):
            raise TypeError("Ключ должен быть строкой")
        hash_value = 0
        base = 31
        for ch in key:
            hash_value = (hash_value * base + ord(ch))% self.size
        return hash_value
```

Вывод результата в терминал:

Вставка элементов в хэш-таблицу

put('яблоко', 10)

put('груша', 5)

put('апельсин', 8)

put('манго', 3)

put('киви', 7)

put('банан', 12)

put('ананас', 4)

Состояние хэш-таблицы

Bucket 0: [пусто]

Bucket 1: [('манго', 3)]

Bucket 2: [пусто]

Bucket 3: [('яблоко', 10), ('груша', 5), ('банан', 12)]

Bucket 4: [('киви', 7), ('ананас', 4)]

Bucket 5: [('апельсин', 8)]

Bucket 6: [пусто]

Значение для 'банан': 12

Удален ключ 'груша'

Состояние хэш-таблицы

```
Bucket 0: [пусто]
Bucket 1: [('манго', 3)]
Bucket 2: [пусто]
Bucket 3: [('яблоко', 10), ('банан', 12)]
Bucket 4: [('киви', 7), ('ананас', 4)]
Bucket 5: [('апельсин', 8)]
Bucket 6: [пусто]
```

Цель: реализовать хэш-таблицу с разрешением коллизий.

Подход:

Хэш-функция: h = (h * 31 + ord(c)) % size - хорошее распределение.

Коллизии - метод цепочек (списки в бакетах).

Операции put/get/remove - O(1) в среднем, O(n) в худшем.

Визуализация: вывод бакетов с парами ключ-значение - показывает, как распределяются данные.

Задача - 3.9 Частотный словарь

1. Построить HashMap частот встречаемости слов в тексте.

2. Вывести топ-10 самых частых слов.

3. Сравнить время построения частотного словаря при: плохой хэш-функции (например, всегда 1), хорошей хэш-функции.

```
    def __init__(self, size=1009, hash_func=None):
        self.size = size
        self.buckets = [[] for _ in range(size)]
        self.hash_func = hash_func if hash_func else
        self._good_hash
```

Вывод результата в терминал:

Обработано 13 слов.

Замер времени с хорошей хэш-функцией

Время: 0.000106 сек

Замер времени с плохой хэш-функцией

Время: 0.000142 сек

Сравнение:

Хорошая хэш-функция: 0.000106 сек

Плохая хэш-функция: 0.000142 сек

Разница: в 1.3 раз медленнее

10 самых частых слов:

1. два -3
2. один -2
3. дин -1
4. пять -1
5. шесть -1
6. восемь -1
7. семь -1
8. девять -1
9. три -1
10. четыре - 1

Цель: проанализировать влияние качества хэш-функции.

Подход:

Хорошая функция - равномерное распределение - O(1) на операцию.

Плохая (всегда 1) - все слова в одном бакете - O(n) на каждую операцию - квадратичное время.

Топ-10: сортировка по частоте - вывод самых популярных слов.

Вывод: качественная хэш-функция критична для производительности.

Задача - 3.10 Trie + HashMap: автодополнение

1. Реализовать Trie для хранения слов.

2. Реализовать поиск по префиксу: метод autocomplete(prefix).

3. Используя HashMap + Trie: хранить слова + их частоты, предлагать подсказки в порядке убывания частоты.

```
class Trie:
    def __init__(self):
        self.root = TrieNode()
    def insert(self, word: str):
        node = self.root
        for ch in word:
            if ch not in node.children:
                node.children[ch] = TrieNode()
            node = node.children[ch]
        node.is_end = True
```

Вывод через терминал:

Добавление слов

+ 'apple'

+ 'application'

+ 'apply'

+ 'app'

+ 'banana'

+ 'band'

+ 'bandana'

Частоты (HashMap):

app: 1

apple: 1

application: 1

apply: 1

banana: 1

band: 1

bandana: 1

Автодополнение по префиксам:

Префикс: 'app'

```
  1. app (частота: 1)
  2. apple (частота: 1)
  3. application (частота: 1)
  4. apply (частота: 1)
Префикс: 'ban'
  1. banana (частота: 1)
  2. band (частота: 1)
  3. bandana (частота: 1)
Префикс: 'a'
  1. app (частота: 1)
  2. apple (частота: 1)
  3. application (частота: 1)
  4. apply (частота: 1)
Префикс: 'x'
Нет подсказок
```

Цель: поиск по префиксу с учётом частоты.

Подход:

Trie - дерево префиксов: каждый узел - буква.

HashMap не используется напрямую: частота хранится в узле is_end и freq.

autocomplete(prefix) - DFS от узла префикса - сбор всех слов - сортировка по убыванию частоты.

04 - Деревья и графы

Задача - 4.11 Бинарное дерево поиска (BST)

1. Реализовать: Вставку, поиск, удаление, обходы: in-order, pre-order, post-order.

2. Проверить, является ли дерево сбалансированным.

```
    def inorder(self):
        return self._traverse(self.root, "in")
    def preorder(self):
        return self._traverse(self.root, "pre")
    def postorder(self):
        return self._traverse(self.root, "post")
```

Вывод в терминал:

Вставка значений: [10, 5, 15, 3, 7]

Поиск:

find(7) = True

find(20) = False

Обходы:

in-order : [3, 5, 7, 10, 15]

pre-order : [10, 5, 3, 7, 15]

post-order : [3, 7, 5, 15, 10]

Удаление значения 10

in-order после удаления: [3, 5, 7, 15]

Сбалансировано ли дерево?

Вырожденное дерево [1,2,3,4,5] True

После операций: False

Вырожденное дерево [1,2,3,4,5] True

После операций: False

Вырожденное дерево [1,2,3,4,5] False

После операций: False

Вырожденное дерево [1,2,3,4,5] False

После операций: False

Вырожденное дерево [1,2,3,4,5] False

После операций: False

Цель: реализовать дерево с упорядоченным доступом.

Операции:

Вставка/поиск - O(log n) в среднем, O(n) в худшем (вырожденное дерево).

Удаление - три случая: лист, один потомок, два потомка (замена на минимум в правом поддереве).

Обходы: in-order - отсортированная последовательность.

Балансировка: проверка через height() - если на любом узле (left - right) > 1 - несбалансированно.

Задача - 4.12 Trie (углубление)

1. Добавить в Trie возможность:

Хранения слов целиком,

подсчета количества вариантов по префиксу,

удаления слова.

```
    def delete(self, word: str) -> bool:
        def _delete(node: TrieNode, word: str, depth: int) bool:
            if depth == len(word):
                if not node.is_end:
                    return False
                node.is_end = False
                return len(node.children) == 0
            ch = word[depth]
            if ch not in node.children:
                return False
            should_delete_child = _delete(node.children[ch],word,
            depth + 1)
            if should_delete_child:
                del node.children[ch]
                return len(node.children) == 0 and not
                node.is_end
            return False
        return _delete(self.root, word, 0)
```

Вывод в терминал:

Вставка слов: ['apple', 'app', 'application', 'apply', 'appreciate']

Подсчёт слов по префиксу:

Префикс 'app': 5 слов

Префикс 'appl': 3 слов

Префикс 'appr': 1 слов

Префикс 'xyz': 0 слов

Удаление слова 'app'...

Успешно удалено: False

Слово 'app' существует после удаления: False

```
Слово 'apple' все еще существует: True
После удаления 'app':
Префикс 'app': 4 слов Префикс 'appl': 3 слов

Удаление несуществующего слова 'banana': False
```

Удаление: рекурсивно, с проверкой — можно ли удалить узел (нет потомков и не конец слова).

Подсчет вариантов: count_prefix(prefix) → возвращает размер поддерева.

Эффективность: поиск по префиксу — O(m), где m — длина префикса.

Задача - 4.13 Графы

1. Хранение графов: Матрица смежности, список смежности.

2. Реализовать алгоритмы: BFS, DFS, поиск кратчайшего пути в невзвешенном графе (BFS).

```
    def bfs(self, start: int) -> List[int]:
        visited = [False] * self.n
        queue = deque([start])
        visited[start] = True
        order = []
        while queue:
            u = queue.popleft()
            order.append(u)
            for v in self._get_neighbors(u):
                if not visited[v]:
                    visited[v] = True
                    queue.append(v)
        return order
```

Вывод в терминал:

1. Список смежности:

BFS из вершины 0: [0, 1, 2, 3, 4]

DFS из вершины 0: [0, 1, 3, 2, 4]

Кратчайший путь 0 -> 4: 3

2. Матрица смежности:

BFS из вершины 0: [0, 1, 2, 3, 4]

DFS из вершины 0: [0, 1, 3, 2, 4]

Кратчайший путь 0 -> 4: 3

3. Проверка недостижимости (вершина 0 -> 5 в графе из 5 вершин):

Результат: -1

Цель: реализовать два способа хранения + обходы.

Способы хранения:

Матрица смежности - $O(V^2)$ памяти, но O(1) проверка ребра.

Список смежности - O(V + E), гибкий для разреженных графов.

Алгоритмы:

BFS - очередь - находит кратчайший путь в невзвешенном графе.

DFS - рекурсия/стек - полный обход.

Кратчайший путь: BFS от стартовой вершины - массив dist[].

Задача - 4.14 Задача “Острова”

Дан двумерный массив 0/1. Найти количество “островов” (компонент связности). Использовать DFS или BFS.

```
    def dfs(r, c):
        if (r < 0 or r >= rows or c < 0 or c >= cols or
            visited[r][c] or grid[r][c] == '0'):
            return
        visited[r][c] = True
        for dr, dc in [(1,0), (-1,0), (0,1), (0,-1)]:
            dfs(r + dr, c + dc)
    for i in range(rows):
        for j in range(cols):
            if grid[i][j] == '1' and not visited[i][j]:
                dfs(i, j)
                count += 1
    return count
```

Вывод в терминал:

Количество островов

3

Задача: количество компонент связности в матрице 0/1.

Подход:

Обход в глубину (DFS) или ширину (BFS) от каждой 1.

После обхода — все связанные 1 помечаются как посещённые.

Счётчик увеличивается на 1 за каждый новый остров.

Сложность: O(M×N), так как каждый элемент посещается один раз.

Применение: обработка карт, сегментация изображений.

05 - Куча и приоритетные очереди

Задача - 5.15 Куча

1. Реализовать бинарную мин-кучу: Вставку, извлечение минимума, построение кучи из массива.

Проверить корректность свойств кучи после каждой операции.

```
class MinHeap:
    def __init__(self):
        self.heap = []
    def _parent(self, i: int) -> int:
        return (i - 1) // 2
    def _left(self, i: int) -> int:
        return 2 * i + 1
    def _right(self, i: int) -> int:
        return 2 * i + 2
    def _swap(self, i: int, j: int):
        self.heap[i], self.heap[j] = self.heap[j], self.heap[i]
    def _heapify_up(self, i: int):
        while i > 0 and self.heap[self._parent(i)]> self.heap[i]:
            self._swap(i, self._parent(i))
            i = self._parent(i)
```

Вывод в терминал:

1. Вставка и извлечение:

 push(10) → [10] (корректность: True)

 push(5) → [5, 10] (корректность: True)

 push(15) → [5, 10, 15] (корректность: True)

 push(3) → [3, 5, 15, 10] (корректность: True)

 push(7) → [3, 5, 15, 10, 7] (корректность: True)

 Извлечение минимума:

  pop() → 3, остаток: [5, 7, 15, 10] (корректность: True)

  pop() → 5, остаток: [7, 10, 15] (корректность: True)

  pop() → 7, остаток: [10, 15] (корректность: True)

  pop() → 10, остаток: [15] (корректность: True)

  pop() → 15, остаток: [] (корректность: True)

2. Построение кучи из массива:

 Исходный массив: [4, 10, 3, 5, 1]

```
Построенная куча: [1, 4, 3, 5, 10]
Корректность: True
3. Граничные случаи:
Корректность пустой кучи: True
Корректность кучи из одного элемента: True
```

Цель: реализовать бинарную кучу.

Структура: полное двоичное дерево в массиве.

Индексы: parent = (i-1)//2, left = 2i+1, right = 2i+2.

Операции:

push - heapify_up - O(log n).

pop - замена корня на последний, heapify_down - O(log n).

build_heap - за O(n) (специальный алгоритм от последнего родителя).

Проверка: функция is_min_heap() - каждый родитель ≤ потомков.

Задача - 5.16 Приоритетная очередь

1. Используя heap, реализовать: push(value, priority), pop() - всегда возвращает элемент с минимальным приоритетом.

2. Применить к задаче: планирование задач (task scheduling), поиск k минимальных элементов массива.

```
def task_scheduling_demo():
    print("1.Планирование задач:")
    pq = PriorityQueue()
    tasks = [("C", 3),("A", 1),("B", 2),("D", 4)]
    print("Добавлены задачи:")
    for task, prio in tasks:
        pq.push(task, prio)
        print(f"    '{task}' (приоритет: {prio})")
    print("\nПорядок выполнения:")
    i = 1
    while not pq.is_empty():
        task = pq.pop()
        print(f"    {i}. {task}")
        i += 1
```

Вывод в терминал:

1. Планирование задач:

Добавлены задачи:

'C' (приоритет: 3)

'A' (приоритет: 1)

'B' (приоритет: 2)

'D' (приоритет: 4)

Порядок выполнения:

1. A
2. B
3. C
4. D

2. Поиск k минимальных элементов:

Массив: [15, 3, 9, 1, 12, 7, 5]

k = 4

4 минимальных: [1, 3, 5, 7]

Цель: очередь, где приоритет определяет порядок извлечения.

Реализация: обёртка над мин-кучей, хранящей кортежи (priority, value).

Применения:

Планирование задач: задачи с наименьшим приоритетом выполняются первыми.

k минимальных элементов: вставить все в кучу - извлечь k раз - O(n + k log n).

Альтернатива: можно использовать heapq в Python, но реализация «с нуля» даёт понимание.

Заключение:

В результате выполнения учебной практики «Структуры данных» были успешно реализованы все 16 заданий, охватывающие ключевые разделы теории структур данных. Были разработаны и протестированы собственные реализации базовых и продвинутых структур, включая односвязные и двусвязные списки, стеки и очереди (на массивах, связных списках и стеках), хэш-таблицы с цепочками, бинарные деревья поиска, префиксные деревья Trie, графы (в представлении списка и матрицы смежности), а также бинарные кучи и приоритетные очереди.

Для каждой реализованной структуры проведён анализ трудоёмкости основных операций (вставка, удаление, поиск, обход), выполнено сравнение с альтернативными подходами (например, статический - динамический массив, список - массив, очередь на массиве - очередь на двух стеках). Практическая значимость изучаемых структур подтверждена решением прикладных задач: проверкой корректности скобочных последовательностей, вычислением арифметических выражений методом ОПН, подсчётом островов в бинарной матрице, построением частотных словарей и реализацией автодополнения по префиксу.

Все реализации выполнены на языке Python «с нуля», без использования встроенных аналогов (list, dict, heapq и др.), что позволило глубоко понять внутреннее устройство, механизмы управления памятью и алгоритмическую суть каждой структуры данных. Результаты тестирования и замеров времени подтверждают корректность реализаций и соответствие теоретическим оценкам сложности.

Практика полностью достигла поставленной цели: сформированы прочные навыки проектирования, анализа и применения структур данных для решения разнообразных вычислительных задач. Полученные знания и умения составляют важнейшую основу для дальнейшего профессионального роста в области разработки программного обеспечения.

1.1 Статический массив

```
class StaticArray:
    def __init__(self, capacity=10):
        self.capacity = capacity
        self.data = [None] * capacity
        self.size = 0
    def _check_overflow(self):
        if self.size >= self.capacity:
            raise OverflowError("Массив заполнен")
    def _check_index(self, idx, allow_end=False):
         if idx < 0 or idx > self.size if allow_end else idx >=
self.size:
            raise IndexError("Неверный индекс")
    def pushBack(self, value):  # O(1)
        self._check_overflow()
        self.data[self.size] = value
        self.size += 1
    def pushFront(self, value):  # O(n)
        self.insert(0, value)
    def insert(self, index, value):  # O(n)
        self._check_index(index, allow_end=True)
        self._check_overflow()
        # Сдвиг вправо
         self.data[index + 1 : self.size + 1] = self.data[index :
self.size]
        self.data[index] = value
        self.size += 1
    def remove(self, index):  # O(n)
        self._check_index(index)
        # Сдвиг влево
         self.data[index : self.size - 1] = self.data[index + 1 :
self.size]
        self.data[self.size - 1] = None
        self.size -= 1
    def find(self, value):  # O(n)
        try:
            return self.data[:self.size].index(value)
        except ValueError:
            return -1
    def __str__(self):
         return f"[{', '.join(map(str, self.data[:self.size]))}]
(размер: {self.size}, ёмкость: {self.capacity})"
```

1.2 Динамический массив

```
import time
class DynamicArray:
    def __init__(self):
        self.capacity = 1
        self.size = 0
```

```
        self.data = [None] * self.capacity
    def _resize(self):
        self.capacity *= 2
        new_data = [None] * self.capacity
        for i in range(self.size):
            new_data[i] = self.data[i]
        self.data = new_data
    def pushBack(self, value):
        if self.size == self.capacity:
            self._resize()
        self.data[self.size] = value
        self.size += 1
class StaticArray:
    def __init__(self, capacity=1000):
        self.capacity = capacity
        self.size = 0
        self.data = [None] * capacity
    def pushBack(self, value):
        if self.size >= self.capacity:
            raise OverflowError("Массив заполнен")
        self.data[self.size] = value
        self.size += 1
if __name__ == "__main__":
    print("Замер времени вставки 100 000 элементов")
    start = time.time()
    dyn_arr = DynamicArray()
    for i in range(100_000):
        dyn_arr.pushBack(i)
    dyn_time = time.time() - start
    print(f"Динамический массив: {dyn_time:.4f} сек")
    print(f"Размер: {dyn_arr.size}, Емкость: {dyn_arr.capacity}")
    print("\nСтатический массив (ёмкость = 1000)")
    start = time.time()
    try:
        stat_arr = StaticArray(capacity=1000)
        for i in range(100_000):
            stat_arr.pushBack(i)
    except OverflowError:
        stat_time = time.time() - start
        print(f" Переполнение на {i}-м элементе")
        print(f"   Время до ошибки: {stat_time:.4f} сек")
```

1.3 Односвязный список

```
import time
import random
class ListNode:
    def __init__(self, val=0, next=None):
        self.val = val
        self.next = next
class SinglyLinkedList:
```

```
    def __init__(self):
        self.head = None
    def pushFront(self, value):
        new_node = ListNode(value, self.head)
        self.head = new_node
    def pushBack(self, value):
        new_node = ListNode(value)
        if not self.head:
            self.head = new_node
            return
        cur = self.head
        while cur.next:
            cur = cur.next
        cur.next = new_node
    def remove(self, value):
        if not self.head:
            return
        if self.head.val == value:
            self.head = self.head.next
            return
        cur = self.head
        while cur.next and cur.next.val != value:
            cur = cur.next
        if cur.next:
            cur.next = cur.next.next
    def find(self, value):
        cur = self.head
        while cur:
            if cur.val == value:
                return True
            cur = cur.next
        return False
def array_push_front(arr, value):
    arr.insert(0, value)  # O(n)

def array_remove_value(arr, value):
    try:
        arr.remove(value)  # O(n)
    except ValueError:
        pass
def array_find(arr, value):
    return value in arr  # O(n)
def array_push_front(arr, val):
    arr.insert(0, val)  # O(n)
def array_push_back(arr, val):
    arr.append(val)     # O(1)
def array_remove(arr, val):
    try:
        arr.remove(val)  # O(n)
    except ValueError:
        pass
def compare_operations():
```

```
    N = 10000
    print(f"равнение операций на {N} элементах\n")
    start = time.perf_counter()
    ll = SinglyLinkedList()
    for i in range(N):
        ll.pushFront(i)
    list_time_front = time.perf_counter() - start
    start = time.perf_counter()
    arr = []
    for i in range(N):
        array_push_front(arr, i)
    array_time_front = time.perf_counter() - start
    print(f"[Вставка в начало]")
    print(f"Список: {list_time_front:.4f} сек")
    print(f"Массив: {array_time_front:.4f} сек")
            print(f"Список   быстрее   в   {array_time_front   /
list_time_front:.0f} раз\n")
    start = time.perf_counter()
    ll = SinglyLinkedList()
    for i in range(N):
        ll.pushBack(i)
    list_time_back = time.perf_counter() - start
    start = time.perf_counter()
    arr = []
    for i in range(N):
        array_push_back(arr, i)
    array_time_back = time.perf_counter() - start
    print(f"[Вставка в конец]")
    print(f"Список: {list_time_back:.4f} сек")
    print(f"Массив: {array_time_back:.4f} сек")
             print(f"Массив   быстрее   в   {list_time_back   /
array_time_back:.0f} раз\n")
    ll = SinglyLinkedList()
    for i in range(N):
        ll.pushBack(i)
    arr = list(range(N))
    start = time.perf_counter()
    for i in range(0, N, 2):
        ll.remove(i)
    list_time_del = time.perf_counter() - start
    start = time.perf_counter()
    for i in range(0, N, 2):
        array_remove(arr, i)
    array_time_del = time.perf_counter() - start
    print(f"[Удаление по значению ({N//2} элементов)]")
    print(f"  Список: {list_time_del:.4f} сек")
    print(f"  Массив: {array_time_del:.4f} сек")
    print(f"Список быстрее в {array_time_del / list_time_del:.1f}
раз\n")
if __name__ == "__main__":
    compare_operations()
```

1.4 Двусвязный список

```
class DListNode:
    def __init__(self, val=0, prev=None, next=None):
        self.val = val
        self.prev = prev
        self.next = next
class DoublyLinkedList:
    def __init__(self):
        self.head = None
        self.tail = None
    def insert_after(self, node, value):
        if node is None:
            raise ValueError("Узел не может быть None")
        new_node = DListNode(value, prev=node, next=node.next)
        if node.next:
            node.next.prev = new_node
        node.next = new_node
        if node == self.tail:
            self.tail = new_node
        if self.head == node and self.tail is None:
            self.tail = node
    def remove_node(self, node):
        if node is None:
            return
        if node.prev:
            node.prev.next = node.next
        else:
            self.head = node.next
        if node.next:
            node.next.prev = node.prev
        else:
            self.tail = node.prev
    def push_back(self, value):
        new_node = DListNode(value)
        if not self.head:
            self.head = self.tail = new_node
        else:
            new_node.prev = self.tail
            self.tail.next = new_node
            self.tail = new_node
    def __iter__(self):
        current = self.head
        while current:
            yield current.val
            current = current.next
    def to_list(self):
        return list(self)
if __name__ == "__main__":
    dll = DoublyLinkedList()
    for val in [10, 20, 30]:
        dll.push_back(val)
```

```
    print("Исходный список:", dll.to_list())
    current = dll.head
    while current and current.val != 20:
        current = current.next
    node_20 = current
    dll.insert_after(node_20, 25)
    print("После insert_after(20, 25):", dll.to_list())
    dll.remove_node(node_20)
    print("После remove_node(20):", dll.to_list())
    print("Обход через итератор:")
    for val in dll:
        print("  ", val)
```

2.5 Стек

```
class ListNode:
    def __init__(self, val=0, next=None):
        self.val = val
        self.next = next
class StackArray: #Массив
    def __init__(self):
        self.data = []
    def push(self, x):
        """O(1)"""
        self.data.append(x)
    def pop(self):
        """O(1)"""
        if self.is_empty():
            raise IndexError("Стек пуст")
        return self.data.pop()
    def is_empty(self):
        return len(self.data) == 0
    def top(self):
        if self.is_empty():
            raise IndexError("Стек пуст")
        return self.data[-1]
class StackList: #Список
    def __init__(self):
        self.head = None
    def push(self, x):
        """O(1) — вставка в голову"""
        new_node = ListNode(x, self.head)
        self.head = new_node
    def pop(self):
        """O(1)"""
        if self.is_empty():
            raise IndexError("Стек пуст")
        val = self.head.val
        self.head = self.head.next
        return val
    def is_empty(self):
```

```
        return self.head is None
    def top(self):
        if self.is_empty():
            raise IndexError("Стек пуст")
        return self.head.val
def is_valid_parentheses(s: str) -> bool:
    stack = StackArray()
    pairs = {')': '(', ']': '[', '}': '{'}
    for char in s:
        if char in '([{':
            stack.push(char)
        elif char in pairs:
            if stack.is_empty():
                return False
            if stack.pop() != pairs[char]:
                return False
    return stack.is_empty()
if __name__ == "__main__":
    print("1. Стек на массиве:")
    s1 = StackArray()
    s1.push(10)
    s1.push(20)
    print("  push(10), push(20)")
    print("  pop() =", s1.pop())
    print("  pop() =", s1.pop())
    print("  пуст?", s1.is_empty())
    print("\n2. Стек на связном списке:")
    s2 = StackList()
    s2.push("A")
    s2.push("B")
    print("  push('A'), push('B')")
    print("  pop() =", s2.pop())
    print("  pop() =", s2.pop())
    print("  пуст?", s2.is_empty())
    print("\n3. Проверка скобочных последовательностей:")
    test_cases = [
        "()",
        "()[]{}",
        "([{}])",
        "([)]",
        "(((",
        ")))",
        "",
        "{[()]}",
    ]
    for expr in test_cases:
        result = is_valid_parentheses(expr)
        print(f"  '{expr}' -> {result}")
```

2.6 Очередь

```
class CircularQueue:
    def __init__(self, capacity=5):
        self.capacity = capacity
        self.queue = [None] * capacity
        self.head = 0
        self.tail = 0
        self.size = 0
    def enqueue(self, x):
        if self.size == self.capacity:
            raise OverflowError("Очередь заполнена")
        self.queue[self.tail] = x
        self.tail = (self.tail + 1) % self.capacity
        self.size += 1
    def dequeue(self):
        if self.size == 0:
            raise IndexError("Очередь пуста")
        val = self.queue[self.head]
        self.head = (self.head + 1) % self.capacity
        self.size -= 1
        return val
    def is_empty(self):
        return self.size == 0
class Stack:
    def __init__(self):
        self.data = []
    def push(self, x):
        self.data.append(x)
    def pop(self):
        return self.data.pop()
    def is_empty(self):
        return len(self.data) == 0
class QueueTwoStacks:
    def __init__(self):
        self.in_stack = Stack()
        self.out_stack = Stack()
    def enqueue(self, x):
        self.in_stack.push(x)
    def dequeue(self):
        if self.out_stack.is_empty():
            if self.in_stack.is_empty():
                raise IndexError("Очередь пуста")
            while not self.in_stack.is_empty():
                self.out_stack.push(self.in_stack.pop())
        return self.out_stack.pop()
    def is_empty(self):
        return self.in_stack.is_empty() and
self.out_stack.is_empty()
if __name__ == "__main__":
    print("1. Очередь на циклическом массиве (ёмкость = 5):")
    cq = CircularQueue(capacity=5)
```

```
    for i in range(1, 6):
        cq.enqueue(i)
        print(f"  enqueue({i}) → очередь: [{', '.join(str(x) for
x in cq.queue)}], head={cq.head}, tail={cq.tail},
size={cq.size}")
    print("  Извлечение элементов:")
    while not cq.is_empty():
        val = cq.dequeue()
        print(f"    dequeue() → {val}, size={cq.size}")
    print("\n2. Очередь на двух стеках:")
    qs = QueueTwoStacks()
    for i in range(10, 16):
        qs.enqueue(i)
        print(f"  enqueue({i})")
    print("  Извлечение элементов:")
    while not qs.is_empty():
        val = qs.dequeue()
        print(f"    dequeue() → {val}")
```

2.7 Задача “Калькулятор”

```
def tokenize(expr: str):
    tokens = []
    i = 0
    while i < len(expr):
        if expr[i].isspace():
            i += 1
            continue
        if expr[i].isdigit():
            num = ''
            while i < len(expr) and expr[i].isdigit():
                num += expr[i]
                i += 1
            tokens.append(int(num))
            continue
        if expr[i] in '+-*/()':
            tokens.append(expr[i])
            i += 1
        else:
            raise ValueError(f"Недопустимый символ: '{expr[i]}'")
    return tokens
def infix_to_postfix(tokens):
    output = []
    stack = []
    precedence = {'+': 1, '-': 1, '*': 2, '/': 2}
    for token in tokens:
        if isinstance(token, int):
            output.append(token)
        elif token == '(':
            stack.append(token)
        elif token == ')':
```

```
            while stack and stack[-1] != '(':
                output.append(stack.pop())
            if not stack:
                raise ValueError("Несбалансированные скобки")
            stack.pop()
        elif token in precedence:
            while (
                stack and
                stack[-1] != '(' and
                stack[-1] in precedence and
                precedence[stack[-1]] >= precedence[token]
            ):
                output.append(stack.pop())
            stack.append(token)
        else:
            raise ValueError(f"Неизвестный токен: {token}")
    while stack:
        if stack[-1] in '()':
            raise ValueError("Несбалансированные скобки")
        output.append(stack.pop())
    return output
def eval_postfix(postfix):
    stack = []
    for token in postfix:
        if isinstance(token, int):
            stack.append(token)
        else:
            if len(stack) < 2:
                raise ValueError("Недостаточно операндов")
            b = stack.pop()
            a = stack.pop()
            if token == '+':
                stack.append(a + b)
            elif token == '-':
                stack.append(a - b)
            elif token == '*':
                stack.append(a * b)
            elif token == '/':
                if b == 0:
                    raise ZeroDivisionError("Деление на ноль")
                stack.append(int(a / b))
    if len(stack) != 1:
        raise ValueError("Некорректное выражение")
    return stack[0]
def calculate(expression: str) -> int:
    tokens = tokenize(expression)
    postfix = infix_to_postfix(tokens)
    result = eval_postfix(postfix)
    return result
if __name__ == "__main__":
    test_cases = [
        "3 + 4 * 2",
```

```
        "(3 + 4) * 2",
        "15 / 3 - 2",
        "10 - 2 - 3",
        "2 * (5 + 3) / 4",
        "100 / 10 / 2",
        "((2 + 3) * (5 - 1)) / 2"
    ]
    for expr in test_cases:
            res = calculate(expr)
            print(f"  {expr} = {res}")
```

3.8 Своя хэш-таблица

```
class HashTable:
    def __init__(self, size=10):
        self.size = size
        self.buckets = [[] for _ in range(size)]
    def _hash(self, key: str) -> int:
        if not isinstance(key, str):
            raise TypeError("Ключ должен быть строкой")
        hash_value = 0
        base = 31
        for ch in key:
            hash_value = (hash_value * base + ord(ch)) %
self.size
        return hash_value
    def put(self, key: str, value):
        index = self._hash(key)
        bucket = self.buckets[index]
        for i, (k, v) in enumerate(bucket):
            if k == key:
                bucket[i] = (key, value)
                return
        bucket.append((key, value))
    def get(self, key: str):
        index = self._hash(key)
        for k, v in self.buckets[index]:
            if k == key:
                return v
        raise KeyError(f"Ключ '{key}' не найден")
    def remove(self, key: str):
        index = self._hash(key)
        bucket = self.buckets[index]
        for i, (k, v) in enumerate(bucket):
            if k == key:
                del bucket[i]
                return
        raise KeyError(f"Ключ '{key}' не найден")
    def visualize(self):
        print("\nСостояние хэш-таблицы")
        for i, bucket in enumerate(self.buckets):
```

```
            if bucket:
                items = ', '.join([f"('{k}', {v})" for k, v in
bucket])
                print(f"Bucket {i}: [{items}]")
            else:
                print(f"Bucket {i}: [пусто]")
if __name__ == "__main__":
    ht = HashTable(size=7)
    words = [
        ("яблоко", 10),("груша", 5),("апельсин", 8),
        ("манго", 3),
        ("киви", 7),
        ("банан", 12),
        ("ананас", 4)
    ]
    print("Вставка элементов в хэш-таблицу")
    for key, value in words:
        ht.put(key, value)
        print(f"  put('{key}', {value})")
    ht.visualize()
    print(f"Значение для 'банан': {ht.get('банан')}")
    ht.remove("груша")
    print("Удалён ключ 'груша'")
    ht.visualize()
```

3.9 Частотный словарь

```
import time
import re
from typing import List, Tuple
class HashTable:
    def __init__(self, size=1009, hash_func=None):
        self.size = size
        self.buckets = [[] for _ in range(size)]
        self.hash_func = hash_func if hash_func else
self._good_hash
    def _good_hash(self, key: str) -> int:
        h = 0
        base = 31
        for ch in key:
            h = (h * base + ord(ch)) % self.size
        return h
    def _bad_hash(self, key: str) -> int:
        return 1
    def put(self, key: str, value: int):
        index = self.hash_func(key)
        bucket = self.buckets[index]
        for i, (k, v) in enumerate(bucket):
            if k == key:
                bucket[i] = (key, value)
                return
```

```
        bucket.append((key, value))
    def get(self, key: str) -> int:
        index = self.hash_func(key)
        for k, v in self.buckets[index]:
            if k == key:
                return v
        return 0  # если не найдено
    def get_all_items(self) -> List[Tuple[str, int]]:
        items = []
        for bucket in self.buckets:
            items.extend(bucket)
        return items
def preprocess_text(text: str) -> List[str]:
    text = text.lower()
    words = re.findall(r'\b[a-яёa-z]+\b', text)
    return words
def build_frequency_table(words: List[str], use_bad_hash: bool =
False) -> HashTable:
    if use_bad_hash:
        ht = HashTable(hash_func=lambda key: 1)
    else:
        ht = HashTable()
    for word in words:
        freq = ht.get(word)
        ht.put(word, freq + 1)
    return ht
def get_top10(ht: HashTable) -> List[Tuple[str, int]]:
    items = ht.get_all_items()
    items.sort(key=lambda x: x[1], reverse=True)
    return items[:10]
SAMPLE_TEXT = """
дин два три четыре пять шесть семь восемь девять один один два
два
"""
if __name__ == "__main__":

    words = preprocess_text(SAMPLE_TEXT)
    print(f"Обработано {len(words)} слов.\n")

    print("Замер времени с хорошей хэш-функцией")
    start = time.perf_counter()
    ht_good = build_frequency_table(words, use_bad_hash=False)
    time_good = time.perf_counter() - start
    print(f"Время: {time_good:.6f} сек")
    print("\nЗамер времени с плохой хэш-функцией")
    start = time.perf_counter()
    ht_bad = build_frequency_table(words, use_bad_hash=True)
    time_bad = time.perf_counter() - start
    print(f"Время: {time_bad:.6f} сек")
    print(f"\nСравнение:")
    print(f"Хорошая хэш-функция: {time_good:.6f} сек")
    print(f"Плохая хэш-функция:  {time_bad:.6f} сек")
```

```
    if time_good > 0:
        print(f"Разница: в {time_bad / time_good:.1f} раз
медленнее")
    print("\n10 самых частых слов:")
    top10 = get_top10(ht_good)
    for i, (word, freq) in enumerate(top10, 1):
        print(f"{i:2}. {word:<12} - {freq}")
```

3.10 Trie + HashMap: автодополнение

```
class TrieNode:
    def __init__(self):
        self.children = {}
        self.is_end = False
class Trie:
    def __init__(self):
        self.root = TrieNode()
    def insert(self, word: str):
        node = self.root
        for ch in word:
            if ch not in node.children:
                node.children[ch] = TrieNode()
            node = node.children[ch]
        node.is_end = True
    def _dfs(self, node: TrieNode, prefix: str, results: list):
        if node.is_end:
            results.append(prefix)
        for ch, child in node.children.items():
            self._dfs(child, prefix + ch, results)
    def get_all_words_with_prefix(self, prefix: str) -> list:
        node = self.root
        for ch in prefix:
            if ch not in node.children:
                return []
            node = node.children[ch]
        results = []
        self._dfs(node, prefix, results)
        return results
class AutocompleteSystem:
    def __init__(self):
        self.trie = Trie()
        self.freq_map = {}
    def add_word(self, word: str):
        word = word.lower()
        self.trie.insert(word)
        self.freq_map[word] = self.freq_map.get(word, 0) + 1
    def autocomplete(self, prefix: str) -> list:
        prefix = prefix.lower()
        candidates = self.trie.get_all_words_with_prefix(prefix)
        candidates.sort(key=lambda w: (-self.freq_map[w], w))
        return candidates
```

```
if __name__ == "__main__":
    ac = AutocompleteSystem()
    words = [
        "apple", "application", "apply", "app",
        "banana", "band", "bandana"
    ]
    print("Добавление слов")
    for w in words:
        ac.add_word(w)
        print(f"  + '{w}'")
    print(f"\nЧастоты (HashMap):")
    for word, freq in sorted(ac.freq_map.items()):
        print(f"  {word}: {freq}")
    test_prefixes = ["app", "ban", "a", "x"]
    print("\nАвтодополнение по префиксам:")
    for p in test_prefixes:
        suggestions = ac.autocomplete(p)
        print(f"\nПрефикс: '{p}'")
        if suggestions:
            for i, word in enumerate(suggestions[:5], 1):
                print(f"  {i}. {word} (частота:
{ac.freq_map[word]})")
        else:
            print("Нет подсказок")
```

4.11 Бинарное дерево поиска (BST)

```
class BSTNode:
    def __init__(self, val):
        self.val = val
        self.left = None
        self.right = None
class BST:
    def __init__(self):
        self.root = None
    def insert(self, val):
        self.root = self._insert(self.root, val)
    def _insert(self, node, val):
        if not node:
            return BSTNode(val)
        if val < node.val:
            node.left = self._insert(node.left, val)
        else:
            node.right = self._insert(node.right, val)
        return node
    def find(self, val):
        return self._find(self.root, val)
    def _find(self, node, val):
        if not node:
            return False
        if val == node.val:
```

```
            return True
        elif val < node.val:
            return self._find(node.left, val)
        else:
            return self._find(node.right, val)
    def delete(self, val):
        self.root = self._delete(self.root, val)
    def _delete(self, node, val):
        if not node:
            return node
        if val < node.val:
            node.left = self._delete(node.left, val)
        elif val > node.val:
            node.right = self._delete(node.right, val)
        else:
            if not node.left:
                return node.right
            if not node.right:
                return node.left
            min_val = self._min_value(node.right)
            node.val = min_val
            node.right = self._delete(node.right, min_val)
        return node
    def _min_value(self, node):
        while node.left:
            node = node.left
        return node.val
    def inorder(self):
        return self._traverse(self.root, "in")
    def preorder(self):
        return self._traverse(self.root, "pre")
    def postorder(self):
        return self._traverse(self.root, "post")
    def _traverse(self, node, order):
        if not node:
            return []
        res = []
        if order == "pre":
            res.append(node.val)
        res += self._traverse(node.left, order)
        if order == "in":
            res.append(node.val)
        res += self._traverse(node.right, order)
        if order == "post":
            res.append(node.val)
        return res
    def is_balanced(self):
        def height(node):
            if not node:
                return 0
            left_h = height(node.left)
            right_h = height(node.right)
```

```
            if left_h == -1 or right_h == -1 or abs(left_h -
right_h) > 1:
                return -1
            return max(left_h, right_h) + 1
        return height(self.root) != -1
if __name__ == "__main__":
    bst = BST()
    values = [10, 5, 15, 3, 7]
    print("Вставка значений:", values)
    for v in values:
        bst.insert(v)
    print("\nПоиск:")
    for v in [7, 20]:
        print(f"  find({v}) = {bst.find(v)}")
    print("\nОбходы:")
    print("  in-order   :", bst.inorder())
    print("  pre-order  :", bst.preorder())
    print("  post-order :", bst.postorder())
    print("\nУдаление значения 10")
    bst.delete(10)
    print("  in-order после удаления:", bst.inorder())
    print("\nПроверка баланса")
    bst_degenerate = BST()
    for val in [1, 2, 3, 4, 5]:
        bst_degenerate.insert(val)
        print(" Вырожденное дерево [1,2,3,4,5]",
bst_degenerate.is_balanced())
        print("  После операций:", bst.is_balanced())
```

4.12 Trie (углубление)

```
class TrieNode:
    def __init__(self):
        self.children = {}
        self.is_end = False
class Trie:
    def __init__(self):
        self.root = TrieNode()
    def insert(self, word: str):
        node = self.root
        for ch in word:
            if ch not in node.children:
                node.children[ch] = TrieNode()
            node = node.children[ch]
        node.is_end = True
    def count_words_with_prefix(self, prefix: str) -> int:
        node = self.root
        for ch in prefix:
            if ch not in node.children:
                return 0
            node = node.children[ch]
```

```
        return self._count_words_from(node)
    def _count_words_from(self, node: TrieNode) -> int:
        count = 1 if node.is_end else 0
        for child in node.children.values():
            count += self._count_words_from(child)
        return count
    def delete(self, word: str) -> bool:
        def _delete(node: TrieNode, word: str, depth: int) ->
bool:
            if depth == len(word):
                if not node.is_end:
                    return False
                node.is_end = False
                return len(node.children) == 0
            ch = word[depth]
            if ch not in node.children:
                return False
            should_delete_child = _delete(node.children[ch],
word, depth + 1)
            if should_delete_child:
                del node.children[ch]
                return len(node.children) == 0 and not
node.is_end
            return False
        return _delete(self.root, word, 0)
    def search(self, word: str) -> bool:
        node = self.root
        for ch in word:
            if ch not in node.children:
                return False
            node = node.children[ch]
        return node.is_end
if __name__ == "__main__":
    trie = Trie()
    words = ["apple", "app", "application", "apply",
"appreciate"]
    print("Вставка слов:", words)
    for w in words:
        trie.insert(w)
    test_prefixes = ["app", "appl", "appr", "xyz"]
    print("\nПодсчёт слов по префиксу:")
    for p in test_prefixes:
        cnt = trie.count_words_with_prefix(p)
        print(f"  Префикс '{p}': {cnt} слов")
    print("\nУдаление слова 'app'...")
    deleted = trie.delete("app")
    print(f"  Успешно удалено: {deleted}")
    print(f"  Слово 'app' существует после удаления:
{trie.search('app')}")
    print(f"  Слово 'apple' всё ещё существует:
{trie.search('apple')}")
    print("\nПосле удаления 'app':")
```

```
    for p in ["app", "appl"]:
        cnt = trie.count_words_with_prefix(p)
        print(f"  Префикс '{p}': {cnt} слов")
    print(f"\nУдаление несуществующего слова 'banana':
{trie.delete('banana')}")
```

4.13 Графы

```
from collections import deque
from typing import List, Optional
class Graph:
    def __init__(self, n: int, edges: Optional[List[tuple]] =
None, use_matrix: bool = False):
        self.n = n
        self.use_matrix = use_matrix
        if use_matrix:
            self.matrix = [[0] * n for _ in range(n)]
            self.adj_list = None
        else:
            self.adj_list = [[] for _ in range(n)]
            self.matrix = None
        if edges:
            for u, v in edges:
                self.add_edge(u, v)
    def add_edge(self, u: int, v: int):
        if self.use_matrix:
            self.matrix[u][v] = 1
            self.matrix[v][u] = 1
        else:
            self.adj_list[u].append(v)
            self.adj_list[v].append(u)
    def _get_neighbors(self, u: int) -> List[int]:
        if self.use_matrix:
            return [v for v in range(self.n) if self.matrix[u][v]
== 1]
        else:
            return self.adj_list[u]
    def bfs(self, start: int) -> List[int]:
        visited = [False] * self.n
        queue = deque([start])
        visited[start] = True
        order = []
        while queue:
            u = queue.popleft()
            order.append(u)
            for v in self._get_neighbors(u):
                if not visited[v]:
                    visited[v] = True
                    queue.append(v)
        return order
    def dfs(self, start: int) -> List[int]:
```

```
        visited = [False] * self.n
        order = []
        def _dfs(u):
            visited[u] = True
            order.append(u)
            for v in self._get_neighbors(u):
                if not visited[v]:
                    _dfs(v)
        _dfs(start)
        return order
    def shortest_path(self, start: int, end: int) -> int:
        if start == end:
            return 0
        visited = [False] * self.n
        queue = deque([(start, 0)])
        visited[start] = True
        while queue:
            u, dist = queue.popleft()
            for v in self._get_neighbors(u):
                if v == end:
                    return dist + 1
                if not visited[v]:
                    visited[v] = True
                    queue.append((v, dist + 1))
        return -1
if __name__ == "__main__":
    edges = [(0, 1), (0, 2), (1, 3), (2, 3), (3, 4)]
    n = 5
    print("1. Список смежности:")
    g_list = Graph(n, edges, use_matrix=False)
    print("  BFS из вершины 0:", g_list.bfs(0))
    print("  DFS из вершины 0:", g_list.dfs(0))
    print("  Кратчайший путь 0 -> 4:", g_list.shortest_path(0,
4))
    print("\n2. Матрица смежности:")
    g_matrix = Graph(n, edges, use_matrix=True)
    print("  BFS из вершины 0:", g_matrix.bfs(0))
    print("  DFS из вершины 0:", g_matrix.dfs(0))
    print("  Кратчайший путь 0 -> 4:", g_matrix.shortest_path(0,
4))
    print("\n3. Проверка недостижимости (вершина 0 -> 5 в графе
из 5 вершин):")
    print("  Результат:", g_list.shortest_path(0, 5))
```

4.14 Задача “Острова”

```
def num_islands(grid):
    if not grid:
        return 0
    rows, cols = len(grid), len(grid[0])
    visited = [[False] * cols for _ in range(rows)]
```

```
    count = 0
    def dfs(r, c):
        if (r < 0 or r >= rows or c < 0 or c >= cols or
            visited[r][c] or grid[r][c] == '0'):
            return
        visited[r][c] = True
        for dr, dc in [(1,0), (-1,0), (0,1), (0,-1)]:
            dfs(r + dr, c + dc)
    for i in range(rows):
        for j in range(cols):
            if grid[i][j] == '1' and not visited[i][j]:
                dfs(i, j)
                count += 1
    return count
grid = [
    ["1","1","0","0","0"],
    ["1","1","0","0","0"],
    ["0","0","1","0","0"],
    ["0","0","0","1","1"]
]
print("Количество островов")
print(num_islands(grid))
```

5.15 Куча

```
class MinHeap:
    def __init__(self):
        self.heap = []
    def _parent(self, i: int) -> int:
        return (i - 1) // 2
    def _left(self, i: int) -> int:
        return 2 * i + 1
    def _right(self, i: int) -> int:
        return 2 * i + 2
    def _swap(self, i: int, j: int):
        self.heap[i], self.heap[j] = self.heap[j], self.heap[i]
    def _heapify_up(self, i: int):
        while i > 0 and self.heap[self._parent(i)] >
self.heap[i]:
            self._swap(i, self._parent(i))
            i = self._parent(i)
    def _heapify_down(self, i: int):
        while True:
            smallest = i
            l, r = self._left(i), self._right(i)
            if l < len(self.heap) and self.heap[l] <
self.heap[smallest]:
                smallest = l
            if r < len(self.heap) and self.heap[r] <
self.heap[smallest]:
                smallest = r
```

```
            if smallest == i:
                break
            self._swap(i, smallest)
            i = smallest
    def push(self, val: int):
        self.heap.append(val)
        self._heapify_up(len(self.heap) - 1)
    def pop(self) -> int:
        if not self.heap:
            raise IndexError("Куча пуста")
        if len(self.heap) == 1:
            return self.heap.pop()
        root = self.heap[0]
        self.heap[0] = self.heap.pop()
        self._heapify_down(0)
        return root
    def build_heap(self, arr: list):
        self.heap = arr[:]
        for i in range(len(self.heap) // 2 - 1, -1, -1):
            self._heapify_down(i)
    def is_min_heap(self) -> bool:
        n = len(self.heap)
        for i in range(n):
            l, r = self._left(i), self._right(i)
            if l < n and self.heap[i] > self.heap[l]:
                return False
            if r < n and self.heap[i] > self.heap[r]:
                return False
        return True
    def __str__(self):
        return str(self.heap)
if __name__ == "__main__":
    print("1. Вставка и извлечение:")
    h = MinHeap()
    for val in [10, 5, 15, 3, 7]:
        h.push(val)
        is_ok = h.is_min_heap()
        print(f"  push({val}) → {h} (корректность: {is_ok})")
    print("  Извлечение минимума:")
    while h.heap:
        val = h.pop()
        is_ok = h.is_min_heap()
        print(f"    pop() → {val}, остаток: {h} (корректность:
{is_ok})")
    print("\n2. Построение кучи из массива:")
    arr = [4, 10, 3, 5, 1]
    h2 = MinHeap()
    h2.build_heap(arr)
    print(f"  Исходный массив: {arr}")
    print(f"  Построенная куча: {h2}")
    print(f"  Корректность: {h2.is_min_heap()}")
    print("\n3. Граничные случаи:")
```

```
    h3 = MinHeap()
    h3.build_heap([])
    print(f"Корректность пустой кучи: {h3.is_min_heap()}")
    h4 = MinHeap()
    h4.build_heap([42])
    print(f"Корректность кучи из одного элемента:
{h4.is_min_heap()}")
```

5.16. Приоритетная очередь

```
class MinHeap:
    def __init__(self):
        self.heap = []
    def _parent(self, i): return (i - 1) // 2
    def _left(self, i): return 2 * i + 1
    def _right(self, i): return 2 * i + 2
    def _swap(self, i, j):
        self.heap[i], self.heap[j] = self.heap[j], self.heap[i]
    def _heapify_up(self, i):
        while i > 0 and self.heap[self._parent(i)][0] >
self.heap[i][0]:
            self._swap(i, self._parent(i))
            i = self._parent(i)
    def _heapify_down(self, i):
        n = len(self.heap)
        while True:
            smallest = i
            l, r = self._left(i), self._right(i)
            if l < n and self.heap[l][0] <
self.heap[smallest][0]:
                smallest = l
            if r < n and self.heap[r][0] <
self.heap[smallest][0]:
                smallest = r
            if smallest == i:
                break
            self._swap(i, smallest)
            i = smallest
    def push(self, priority, value):
        self.heap.append((priority, value))
        self._heapify_up(len(self.heap) - 1)
    def pop(self):
        if not self.heap:
            raise IndexError("Очередь пуста")
        if len(self.heap) == 1:
            return self.heap.pop()[1]
        top_val = self.heap[0][1]
        self.heap[0] = self.heap.pop()
        self._heapify_down(0)
        return top_val
    def is_empty(self):
```

```
        return len(self.heap) == 0
class PriorityQueue:
    def __init__(self):
        self.heap = MinHeap()
    def push(self, value, priority):
        self.heap.push(priority, value)
    def pop(self):
        return self.heap.pop()
    def is_empty(self):
        return self.heap.is_empty()
def task_scheduling_demo():
    print("1. Планирование задач:")
    pq = PriorityQueue()
    tasks =[("C", 3),("A", 1),("B", 2),("D", 4)]
    print("  Добавлены задачи:")
    for task, prio in tasks:
        pq.push(task, prio)
        print(f"    '{task}' (приоритет: {prio})")
    print("\n  Порядок выполнения:")
    i = 1
    while not pq.is_empty():
        task = pq.pop()
        print(f"    {i}. {task}")
        i += 1
def k_smallest_elements(arr, k):
    if k <= 0:
        return []
    pq = PriorityQueue()
    for x in arr:
        pq.push(x, x)
    return [pq.pop() for _ in range(min(k, len(arr)))]
def k_smallest_demo():
    print("\n2. Поиск k минимальных элементов:")
    arr = [15, 3, 9, 1, 12, 7, 5]
    k = 4
    print(f"  Массив: {arr}")
    print(f"  k = {k}")
    result = k_smallest_elements(arr, k)
    print(f"  {k} минимальных: {result}")
if __name__ == "__main__":
    task_scheduling_demo()
    k_smallest_demo()
```