Министерство науки и высшего образования Российской Федерации

ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ

«НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ ИТМО»

(Университет ИТМО)

Факультет **Прикладная информатика**

Образовательная программа **Мобильные и сетевые технологии**

Направление подготовки(специальность) **09.03.03 Прикладная информатика**

Лабораторные работы №0-7

По дисциплине «Алгоритмы и Структуры Данных»

Выполнил Каприн Семён Евгеньевич к3140.
Проверил Ромакина Оксана Михайловна.
Дата 29.06.2025

Санкт-Петербург 2025

Полное решение всех Лабораторных Лежит на GitHub

https://github.com/SUPER-PAU/AlgorithmsFirstSemesterITMO

Лабораторная 0

задание 1

ввод вывод с помощью input() и print()
работа с файлом с помощью open и w, r - default
list(map(int, f.read().split())) - для компактной конвертации строки через split в список, а затем преобразование каждого элемента в int через map
задание 2
Эффективный алгоритм поиска числа фибоначчи через цикл for за O(n)

```
def calc_fib(n):
    if n <= 1:
        return n
    a, b = 0, 1
    for _ in range(2, n + 1):
        a, b = b, a + b
    return b
```

задание 3

последняя цифра числа фибоначчи
Последняя цифра любого числа вычисляется через (int % 10)
последние цифры Фибоначчи по модулю 10 повторяются каждые 60 чисел (Период Пизано для 10) Алгоритм:

```
def calc_fib(n):
    n %= 60
    if n <= 1:
        return n
    prev, curr = 0, 1
    for _ in range(2, n + 1):
        prev, curr = curr, (prev + curr) % 10
    return curr
```

задание 4

декоратор test_performance для проверки времени и памяти сделал в файле tests.py
проверка производилась с помощью time.perf_counter() в начале и в конце выполнения функции
проверка на память производилась с помощью tracemalloc.start() в начале и tracemalloc.get_traced_memory()[1] в после функции

Проверил 2 и 3 задания на время выполнения с помощью файлов с максимальным возможным значением по условию.

Лабораторная 1 - Сортировка вставками

принцип работы

- проходимся индексом j по списку, начиная с 2-го элемента списка и обзначаем этот элемент как key
- Берем предыдущий элемент как i = j - 1
- Идем индексом i до начала, если i-й элемент больше key
- меняем передний элемент i + 1 на текущий i и двигаемся назад i--
- под конец меняем элемент i + 1 на key.

Таким образом мы за элементом j всегда получаем отсортированный массив, и если встречается элемент меньше предыдущего, то "продавливаем" его назад, пока он не встанет на свое место.

время работы алгоритма:
min O(n)
mid O(n^2)
max O(n^2)
Затраты по памяти - O(1)
Алгоритм:

```
def insertion_sort(lst):
    for j in range(1, len(lst)):
        key = lst[j]
        i = j - 1
        while i >= 0 and lst[i] > key:
            lst[i + 1] = lst[i]
            i -= 1
        lst[i + 1] = key
```

задание 1 - отсортировать массив по insertion sort

используем функцию insertion_sort

Задание 3 - Сортировка по убыванию

необходимо переписать алгоритм, чтобы он сортировал по невозрастанию. необходимо использовать swap. Можно ли переписать алгоритм с использованием рекурсии?

чтобы сортировать список в обратном порядке достаточно поменять условие в условном операторе с ">" на "<", когда идем назад i while i > 0 and lst[i - 1] < lst[i]:. Таким образом алгоритм будет менять элементы, в порядке невозрастания.

также, чтобы не возвращать ключ в конце while в список, можно сразу его поставить на место используя swap. в моем случае использовал swap в стиле питона. lst[i], lst[i - 1] = lst[i - 1], lst[i]

Алгоритм:

```
def insertion_sort_rev(lst):
    for j in range(1, len(lst)):
        i = j
        while i > 0 and lst[i - 1] < lst[i]:  # меняем условный оператор с ">" на "<"
            lst[i], lst[i - 1] = lst[i - 1], lst[i] # swap на python
            i -= 1
```

можно ли использовать рекурсию? Да.
Сортируем первые n-1 элементов рекурсивно, вместо прохода while
Вставляем n-й элемент на своё место среди уже отсортированных

Задание 6 - Пузырьковая сортировка

Напишите код на Python и докажите корректность пузырьковой сортиров- ки.

Алгоритм сортировки: проход по каждому элементу массива индексом i и вложенным циклом проход индексом j. Сравниваем каждый элемент j с элементом i и меняем их. Таким образом время работы алгоритма будет всегда O(n^2), что не есть хорошо.

Алгоритм:

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

провека на длину и условие:

```
print("Проверка на длину: " + str(len(lst_start) == len(lst)))
    flag = True
    for i in range(1, len(lst)):
        if lst[i] < lst[i - 1]:
            flag = False
            break
    print("Проверка на условие: " + str(flag))
```

Таким образом алгоритм работает также как и insertion, но в лучшем случае сортировка вставками будет быстрее, так как не будет заходить в цикл while и ее скорость будет O(n)

Лабораторная 2 - Сортировка слиянием

принцип работы

- делим список на 2 части, пока не останется по одному элементу в каждой.
- Собираем список воедино из частей, путем их заполнения по порядку который нам нужен, проходясь по обоим спискам. Заполняем массив оставшимися элементами

Таким образом

время работы алгоритма:
min O(n * log(n))
mid O(n * log(n))
max O(n * log(n))
log(n) - деление
n - рекурсивный подъем
Затраты по памяти - O(n) - на хранение поделенных списков
Алгоритм

```
def merge_sort(arr):
    if len(arr) <= 1:
        return arr

    mid = len(arr) // 2
    left = merge_sort(arr[:mid])
    right = merge_sort(arr[mid:])

    return merge(left, right)

def merge(left, right):
    result = []
    i = j = 0

    while i < len(left) and j < len(right):
        if left[i] <= right[j]:
            result.append(left[i])
            i += 1
        else:
            result.append(right[j])
            j += 1

    while i < len(left):
      result.append(left[i])
      i += 1
    while j < len(right):
      result.append(right[j])
      j += 1
    return result
```

Задача 1

1) Необходимо реализовать сортировку слиянием.
2) Проверить наихудший/наилучший случай
3) Переписать структуру без сигнала остановки.

Сортировка слиянием реализована.
проверяем наихудший и наилучший случаи

- генерируем файлы с помощью алгоритма с циклом for длины 10^4
- сравниваем работу программы:

худший случай: 0.1345 секунд
лучший случай: 0.1335 секунд

- итог: сортировка слиянием имеет "одинаковою" временную сложность O(n * log(n))

Переписываем структуру без использования сигналов для остановки
в реализации презентации использовались сигнальные значения ∞, чтобы эти элементы не копировались в результирующий массив

- я переписал без использования сигнальных значений:

Проверка окончания массива идёт по длине: i < len(left) и j < len(right)

Задача 3 - Число инверсий

Инверсия - ситуация, когда i < j, а A_i > A_j - т.е. left[i] > right[j] и i < len(left).
Соответственно существует len(left) - i инверсий

заводим глобальный счетчик и добавляем InversionCount += len(left) - i при left[i] > right[j]

Задача 5 - Представитель большинства

Необходимо усзнать, существует ли элемент, который встречается больше, чем n/2 раз и вывести его если он есть. Необходимо использовать метод разделяй и властвуй сортировки merge

- используем деление на подмассивы left и right длина будет > 1

Далее рекурсивный подъем:

- подсчитываем элемент, который будет чаще всего встречаться проходя по массиву

реализовал функцию count_occurrences, которая заменяет merge

- если элемент подходит по условию count > n/2, то возвращаем его если нет, то None

Итого получаем нужный элемент или None

временная скорость: log n на деление * n на подсчет наибольшего элемента в подмассивах
O(log n * n)

Лабораторная 3 - Сортировка Quick и Count

Задание 1 - Улучшение Quick sort

Сгенерирую 6 тестов для сравнения сортировок - случаи

1) лучший (sorted) - массив размера 10000 отсортирован
2) средний (random) - массив размера 10000 отсортирован случайным образом
3) худший (reversed) - массив размера 10000 перевернутый отсортированный массив
4) 10^3 - массив размера 10^3, с 5-ю разными числами отсортирован случайным образом
5) 10^4 - массив размера 10^4, с 5-ю разными числами отсортирован случайным образом
6) 10^5 - массив размера 10^5, с 5-ю разными числами отсортирован случайным образом

Qsort

Алгоритм:

Выбирается опорный элемент (pivot), чаще всего крайний элемент массива.
Массив делится на две части: элементы меньше pivot и больше pivot.
Сортировка вызывается рекурсивно для этих двух частей.
Время выполнения:

лучший (sorted) - (Очень долгоооо.....)
средний (random) - 0.1639 секунд
худший (reversed) - (Очень долгоооо.....)
10^3 - 0.1087 секунд
10^4 - 71.7161 секунд
10^5 - (Очень долгоооо.....)

randomized Qsort

Алгоритм:

Вместо фиксированного pivot выбирается случайный элемент в диапазоне [l, r], тем самым избегая худшего случая для QSort
Время выполнения:

лучший (sorted) - 0.1681 секунд
средний (random) - 0.1923 секунд
худший (reversed) - 0.1726 секунд
10^3 - 0.1146 секунд
10^4 - 68.0449 секунд
10^5 - (Очень долгоооо.....)

Partition3

Алгоритм:

Массив делится на три части, а не на две:
Элементы меньше pivot
Элементы равные pivot
Элементы больше pivot
Тем самым эффективнее предыдущих на массивах с малым кол-вом уникальных чисел, т.к избегает лишней рекурсии для одинаковых значений (равных pivot)
Время выполнения:

лучший (sorted) - 0.1967 секунд
средний (random) - 0.1919 секунд
худший (reversed) - 0.1946 секунд
10^3 - 0.0073 секунд
10^4 - 0.0672 секунд
10^5 - 0.7386 секунд

MergeSort

Алгоритм:

Делим массив на подмассивы
собираем их, попутно сортируя, на рекурсивном подъеме.
Время выполнения:

лучший (sorted) - 0.1342 секунд
средний (random) - 0.137 секунд
худший (reversed) - 0.1269 секунд
10^3 - 0.0089 секунд
10^4 - 0.1363 секунд
10^5 - 1.7814 секунд

Итого:

- StandardQuickSort — нормальная на случайном массиве, но проваливается на отсортированных/обратных.
- RandomizedQuickSort — стабилен на всех наборах малого размера, особенно хорошо на отсортированных/обратных, но на больших значительно теряет эффективность.
- Partition3 - медленней на лучшем и худшем случае в сравнении с предыдущими, но лучше всех справился с массивами мало различающихся - чисел
- MergeSort - Лучше всех на первых 3-х тестах и чуть хуже Partition3 в массивах с мало различающимися числами

Задание 6 - Сортировка целых чисел

В этой задаче нужно будет отсортировать много неотрицательных целых чисел. В виде A_i * B_j двух массивов A и B. Вывести сумму каждых десятых чисел получившегося массива C.

Для этого создадим список C = [a * b for a in A for b in B]

Так как элементов в списке C может быть очень много - 6000 * 6000 = 36 000 000 необходимо использовать устойчивую и быструю по времени сортировку - merge sort (Описал алгоритм работы в LAB_2)

C = merge_sort(C)

Результат нужно вывести в виде суммы каждого 10-го элемента C - проходимся циклом по каждому 10-му эл-ту, начиная с 0-го result = sum(C[i] for i in range(0, len(C), 10))

Задание 8 - K ближайших точек

Чтобы найти K ближайших точек к началу координат (0, 0), использую модификацию QSort - QuickSelect.

Главная идея алгоритма заключается в том, что нам нет нужды сортировать полностью массив, его можно разделить в нужной позиции.

Выбираем случайный опорный элемент pivot
Разделяем Массив partition():

- меньше Pivot
- больше pivot

Итого pivot остается на своей позиции в отсортированном массиве.

- Если pivot_position == K, то мы нашли нужный разрез return.
- если >, то ищем слева, < - ищем справа.

Таким образом мы отбрасываем ненужную часть массива
Делаем это до момента пока не pivot_position == K.
Далее, чтобы вывести K точек в отсортированном порядке, можно применить любую эффективную сортировку массива points[:K]. Я в этой задаче применил встроенную sorted, по функции calc_dist.
Полный Алгоритм

```
import random

def calc_dist(point):
    x, y = point
    return (x ** 2) + (y ** 2)

def partition(points, left, right, pivot_index):
    pivot_distance = calc_dist(points[pivot_index])
    points[pivot_index], points[right] = points[right], points[pivot_index]
    store_index = left
    for i in range(left, right):
        if calc_dist(points[i]) < pivot_distance:
            points[store_index], points[i] = points[i], points[store_index]
            store_index += 1
    points[right], points[store_index] = points[store_index], points[right]
```

```
    return store_index

def quickselect(points, left, right, K):
    if left < right:
        pivot_index = random.randint(left, right)
        pivot_index = partition(points, left, right, pivot_index)
        if pivot_index == K:
            return
        elif pivot_index < K:
            quickselect(points, pivot_index + 1, right, K)
        else:
            quickselect(points, left, pivot_index - 1, K)

def k_closest(points, K):
    quickselect(points, 0, len(points) - 1, K)

    result = sorted(points[:K], key=calc_dist)
    return result
```

Лабораторная 4

Задание 1 - Стэк

Стэк - Это структура данных, элементы которого можно добавлять в конец массива и извелекать также только с конца. Хранить Данные будем в списке.

Работа со Стэком

```
def pushStack(arr, val):
    arr.append(val)

def popStack(arr):
    return arr.pop()
```

Интерфейс для задачи:

```
with open("input.txt") as file:
    lines = file.readlines()

stack = []

for line in lines:
    line.strip()
    if line.startswith("+"):
        command, value = line.split()
```

```
        pushStack(stack, int(value))
    elif line.startswith("-"):
        print(popStack(stack))
```

Задание 3 - Скобочная последовательность

Идея задачи: для каждой (или [должна быть своя) или]

реализовать эту проверку можно с помощью стека:

- будем добавлять символы ([в стек, и удалять их из стека, если встретится)]

таким образом стек останется заполненным при неправильной последовательности

- также необходимо учитывать момент, когда есть]), но стэк пуст

функция проверки скобок

```
def checkBrackets(string):
    stack = []
    pairs = {')': '(', ']': '['}

    for char in string:
        if char in "([":
            stack.append(char)
        elif char in ")]":
            if not stack or stack[-1] != pairs[char]:
                return False
            stack.pop()

    return not stack
```

работа с файлом

```
with open("input.txt") as file:
    _ = file.readline()
    lines = file.readlines()

for line in lines:
    line.strip()
    if (checkBrackets(line)):
        print("Yes")
    else:
        print("No")
```

Задание 6 - Очередь с минимумом

необходимо реализовать очередь, которая будет овечать на запрос о своем минимальном элементе

Это можно сделать с помощью Очереди на 2-ух стэках

- есть 2 очереди in_queue и out_queue, которые состаят из кортежей (значиние, мин число до этого элемента)
- в in_queue поступают новые элементы, из out_queue удаляются элементы.
- при удалении проверяем out_queue, если он пуст, то переносим все элементы из in_queue
- при этом пересчитываем минимум, добавляя элементы методом _push
- Вставляем элементы в обратном порядке, тем самым сохраняя инвариант очереди.
- in_queue очищается.

Поиск минимума происходит в обоих стэках на последнем элементе. Выводится Минимальный из них.
Реализация очереди

```
class MinQueue:
    def __init__(self):
        self.in_stack = []
        self.out_stack = []

    def _push(self, stack, value):
        curr_min = value
        if stack:
            curr_min = min(value, stack[-1][1])
        stack.append((value, curr_min))

    def addQueue(self, value):
        self._push(self.in_stack, value)

    def deleteQueue(self):
        if not self.out_stack:
            while self.in_stack:
                val = self.in_stack.pop()[0]
                self._push(self.out_stack, val)
        self.out_stack.pop()

    def get_min(self):
        mins = []
        if self.in_stack:
            mins.append(self.in_stack[-1][1])
        if self.out_stack:
            mins.append(self.out_stack[-1][1])
        return min(mins)
```

работа с файлами

```
def main():
    with open("input.txt") as file:
        _ = file.readline()
        lines = file.readlines()
```

```
    queue = MinQueue()

    for line in lines:
        line = line.strip()
        if line.startswith('+'):
            _, num = line.split()
            queue.addQueue(int(num))
        elif line == '-':
            queue.deleteQueue()
        elif line == '?':
            print((queue.get_min()))
```

Задание 9 - Поликлиника

необходимо реализовать очередь, с возможностью добавления в середину. Это можно сделать с помощью "Двойной очереди" левой и правой. В конце каждого действия они будут балансироваться.

- добавление обычного пациента происходит в правую очередь, а привелигерованного - в левую
- извлечение происходит из левого.
- балансировка происходит путем извлечения крайних элементов первого списка и вставку во второй
- балансировка происходит, если длины не равны (левый больше при нечетной длине)

Балансировка

```
def balance(left, right):
    while len(left) < len(right):
        left.append(right.pop(0))
    while len(left) > len(right) + 1:
        right.insert(0, left.pop())
```

работа с интерфейсом

```
with open("input2.txt") as file:
    _ = file.readline()
    lines = file.readlines()

left = []
right = []

for line in lines:
    line = line.strip()
    if line.startswith('+'):
        _, val = line.split()
        right.append(val)
    elif line.startswith('*'):
        _, val = line.split()
        left.append(val)
```

```
    elif line == '-':
        print(left.pop(0))

    balance(left, right)
```

Лабораторная 5 - деревья и куча

Задание 2 - высота дерева

Дерево будем хранить в списке, где каждый элемент списка - список индексов детей вершины. Для удобного построения дерева можно сразу создать вложенный список длины n

```
with open("input.txt") as f:
    n = int(f.readline())
    parent_indexes = list(map(int, f.readline().split()))

tree = [[] for _ in range(n)]
root = None
```

на вход поступили n индексов родителей ix детей. -1 индекс родителя - вершина = корень дерева

заполнение дерева

```
for child in range(n):
    parent = parent_indexes[child]
    if parent == -1:
        root = child
    else:
        tree[parent].append(child)
```

Далее необходимо узнать высоту построенного дерева. сделать это можно с помощью рекурсивного обхода в высоту.

Мы начинаем с корня дерева и запускаем рекурсию от всех детей.

- Если натыкаемся на вершину без детей, то просто возвращаем 1 - это лист.
- Если вершина с детьми, то выбираем макс результат. Итого получаем высоту дерева

```
def dfs(node):
    if not tree[node]:
        return 1
    return 1 + max(dfs(child) for child in tree[node])
```

Задача 5 - потоки

Чтобы решить эту задачу, использую приоритетную очередь (мин. кучу).

Логика мин кучи

- дети вершины всегда меньше ее значения
- при добавлении элемент вставляется в конец и просеивается вверх siftUP
- siftUp просеивает элемент рекурсивно или итеративно (у меня цикл while) в начало списка, пока он не будет удовлетворять условию мин кучи
- при удалении первый элемент удаляется, а последний переносится в начало и просеивается вниз siftDOWN
- siftDOWN просеивает элемент в конец списка, выбирая минимальный из детей, пока не выполнится условие кучи.

храним значения в списке.

- Доступ к детям вершины по формуле left = 2 * i + 1 и right = 2 * i + 2
- Доступ к родителю parent = (i - 1) // 2

Реализация Кучи:

```
class Heap:
    def __init__(self):
        self.data = []

    def push(self, elem):
        self.data.append(elem)
        self.siftUp(len(self.data) - 1)

    def pop(self):
        self.data[0], self.data[len(self.data) - 1] = self.data[len(self.data) - 1], self.data[0]
        elem = self.data.pop()
        self.siftDown(0)
        return elem

    def siftUp(self, i):
        parent = (i - 1) // 2
        while i > 0 and self.data[i] < self.data[parent]:
            self.data[i], self.data[parent] = self.data[parent], self.data[i]
            i = parent
            parent = (i - 1) // 2

    def siftDown(self, i):
        n = len(self.data)
        while True:
            left = 2 * i + 1
            right = 2 * i + 2
            smallest = i
```

```
            if left < n and self.data[left] < self.data[smallest]:
                smallest = left
            if right < n and self.data[right] < self.data[smallest]:
                smallest = right
            if smallest == i:
                break
            self.data[i], self.data[smallest] = self.data[smallest], self.data[i]
            i = smallest
```

в этой куче будут храниться кортежи (Время начала, Номер Потока)

- Изначально время начала будет 0

Проходим по списку задач (их время выполнения)

- извлекаем поток с наименьшим временем, выводим его - в это время он начал новую задачу
- Добавляем этот поток обратно в кучу, увеличивая его значение времени на время выполнения.

Таким образом мин куча помогает распределять потоки по минимальному времени освобождения задачи.

main

```
with open("input.txt") as f:
    n, m = map(int, f.readline().split())
    tasks = list(map(int, f.readline().split()))

heap = Heap()
for i in range(n):
    heap.push((0, i))

for task_time in tasks:
    ready_time, thread_index = heap.pop()
    print(f"{thread_index} {ready_time}")
    heap.push((ready_time + task_time, thread_index))
```

время выполнения будет

O(log n) - каждая операция (siftUp, siftDown)
m - количество потоков
Всего - O(m log n)

Лабораторная 6 - Хэш

(40 * 9 % 50) % 9 = 1

Задача 1 - Множество

Необходимо реализовать множество с операциями «добавление ключа», «удаление ключа», «проверка существования ключа».

Для этого я реализую хэш-таблицу через списки (хеш-таблицу с открытой адресацией)

- Возьмем простое число, чтобы вычислять хэш (модуль) MOD = 10**6 + 5, это также длина нашего списка.
- В каждом элементе нашего списка хранится еще список - цепочка значений с повторяющимся Хэшем

Таким образом мы можем удалять, добавлять и проверять значения не во всем массиве, а только в одной цепочке, что уменьшает затраты по времени.
Программа:

```
MOD = 10**6 + 5

class HashSet:
    def __init__(self):
        self.table = [[] for _ in range(MOD)]

    def hash(self, x):
        return x  % MOD

    def add(self, x):
        h = self._hash(x)
        if x not in self.table[h]:
            self.table[h].append(x)

    def delete(self, x):
        h = self._hash(x)
        if x in self.table[h]:
            self.table[h].remove(x)

    def exists(self, x):
        h = self._hash(x)
        return x in self.table[h]

def main():
    hs = HashSet()

    with open("input.txt") as file:
        n = int(file.readline())
        for _ in range(n):
            cmd, val = file.readline().strip().split()
            x = int(val)

            if cmd == 'A':
```

```
            hs.add(x)
        elif cmd == 'D':
            hs.delete(x)
        elif cmd == '?':
            print('Y' if hs.exists(x) else 'N')
```

Задание 3 - Хеширование с цепочками

необходимо реализовать хэш таблицу с использованием цепочек и полиминальной хэш-функции imgpng S[i] - код ASCII i-го символа строки S, p = 1000000007 и x = 263. m - количество сегментов (входные данные).

Метод вычисления Хэша

```
def _hash(self, s):
    hash_val = 0
    for i in range(len(s)):
        hash_val += ord(s[i]) * (X ** i)
    return (hash_val % P) % self.m
```

Также необходимо поменять вставку из прошлого задания, чтобы слова вставлялись в начало.

```
def add(self, x):
    h = self.hash(x)
    if x not in self.table[h]:
        self.table[h].insert(0, x)
```

Задание 4 - Прошитый ассоциативный массив

Необходимо реализовать словарь, который будет также выполнять функционал двусвязного списка (доступ к элементам предыдущего и следующего ключа).

Для этого реализуем словарь для хранения значений self.map = {}, а также словарь для доступа к текущей ноде ключа self.order = {}.

Также нам необходима информация о head и tail двусвязного списка

Инициализация класса выглядит так:

```
class LinkedMap:
    def __init__(self):
        self.map = {}
        self.order = {}
        self.head = None
        self.tail = None
```

При добавлении добавляем значение в map и создаем соответствующую ноду для ключа и добавляем ее в конец списка. Таким образом у нас сохраняется порядок добавления ключей

```
def put(self, key, value):
    if key in self.map:
        self.map[key] = value
    else:
        self.map[key] = value
        node = Node(key)
        self.order[key] = node
        if self.tail:
            self.tail.next = node
            node.prev = self.tail
            self.tail = node
        else:
            self.head = self.tail = node
```

При удалении также необходимо удалить соответствующую ноду в списке, при этом сохраняя порядок списка.

```
def delete(self, key):
    if key not in self.map:
        return
    del self.map[key]

    node = self.order.pop(key)
    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
```

Таким образом несложно получить next и prev элементы, зная структуру двусвязного списка

```
def prev(self, key):
    if key not in self.order:
        return "<none>"
    prev_node = self.order[key].prev
    return self.map.get(prev_node.key) if prev_node else "<none>"

def next(self, key):
    if key not in self.order:
        return "<none>"
    next_node = self.order[key].next
```

```
    return self.map.get(next_node.key) if next_node else "<none>"
```

Лабораторная 7 - ДП

Задача 6 - Наибольшая возрастающая подпоследовательность

Необходимо найти НВП в последовательности

чтобы это сделать эффективно O(n log n) воспользуемся следующим алгоритмом:

заводим 3 списка
tail - он хранит возможные значения концов нвп длины i + 1
tail_idx - хранит индекс значений tail в исходной последовательности
prev - хранит индекс предыдущего элемента в нвп, для восстановления ответа.
затем находим для каждого элемента место, куда его можно вставить в последовательности tail - pos
если это - конец, то это значение становится новым максимумом длины.
если нет, то вставляем элемент a[i] в это самое место pos, заменяя тем самым больший элемент.
также обновляем предыдущий элемент для текущего prev[i] = tail_idx[pos - 1], если он не начальный.
Затем циклом восстанавливаем ответ, начиная с конца tail_idx и идя по индексам предыдущих элементов в prev.
алгоритм:

```
def nvp(arr):
    n = len(arr)
    tail = []
    tail_idx = []
    prev = [-1] * n

    for i in range(n):
        num = arr[i]
        pos = bisect_left(tail, num)
        if pos == len(tail):
            tail.append(num)
            tail_idx.append(i)
        else:
            tail[pos] = num
            tail_idx[pos] = i
        if pos > 0:
            prev[i] = tail_idx[pos - 1]
```

```
    res = []
    k = tail_idx[-1]
    while k != -1:
        res.append(arr[k])
        k = prev[k]
    res.reverse()
    return len(res), res
```

Задача 7 - шаблоны

Необходимо сравнить шаблон, состоящий из символов ? - случайный символ и * - случайная подстрока любой длины (даже 0) со второй строкой. Если шаблон может быть строкой, то вывести Да, если нет, то Нет.

создаем таблицу dp = [[False] * (m + 1) for _ in range(n + 1)],

- где i - идет по символам шаблона
- j по символам строки
- dp[i][j] = True означает, что первые i символов шаблона равны первым j символам строки
- dp[0][0] делаем True так как пустой шаблон соответствует пустой строке.
- если шаблон состоит только из * то заполняем dp[0-n][0] = True т.к. такой шаблон может быть любой строкой.

```
for i in range(1, n + 1):
    if template[i - 1] == '*':
        dp[i][0] = dp[i - 1][0]
    else:
        break
```

заполняем таблицу dp проходясь по ней

```
for i in range(1, n + 1):
    for j in range(1, m + 1):
        if template[i - 1] == '*':
            dp[i][j] = dp[i - 1][j] or dp[i][j - 1]
        elif template[i - 1] == '?' or template[i - 1] == string[j - 1]:
            dp[i][j] = dp[i - 1][j - 1]
```

- если i - 1 элемент шаблона равен ? или элементы template[i - 1] == string[j - 1] то
    - записываем предыдущий результат в таблицу dp[i][j] = dp[i - 1][j - 1].
    - так она продолжится с True по диагонали.

-- если i - 1 элемент шаблона равен *

- текущий элемент [i][j] будет равен элементу [i-1][j], если он True - строка может быть пустой
- либо элемент [i][j] будет равен dp[i][j - 1], если он True - строка может быть любой длины.

если на каком либо из этапов прохода по i оба условных операторов не выполнились, то остальная таблица будет состоять из false - условие не выполнилось

если все норм, то в конце dp[n][m] будет стоять True

```
def matches(template, string):
    n, m = len(template), len(string)
    dp = [[False] * (m + 1) for _ in range(n + 1)]
    dp[0][0] = True
    for i in range(1, n + 1):
        if template[i - 1] == '*':
            dp[i][0] = dp[i - 1][0]
        else:
            break
    for i in range(1, n + 1):
        for j in range(1, m + 1):
            if template[i - 1] == '*':
                dp[i][j] = dp[i - 1][j] or dp[i][j - 1]
            elif template[i - 1] == '?' or template[i - 1] == string[j - 1]:
                dp[i][j] = dp[i - 1][j - 1]
    return dp[n][m]

def main():
    with open("input.txt") as f:
        template = f.readline().strip()
        string = f.readline().strip()

    if matches(template, string):
        print("YES")
    else:
        print("NO")

if __name__ == '__main__':
    main()
```

пример

template = "k?t*n"
string = "kitten"
тогда таблица dp:

[True, False, False, False, False, False, False]
[False, True, False, False, False, False, False]
[False, False, True, False, False, False, False]
[False, False, False, True, False, False, False]
[False, False, False, True, True, True, True] — *
[False, False, False, False, False, False, True]