МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ

РОССИЙСКОЙ ФЕДЕРАЦИИ

федеральное государственное бюджетное образовательное учреждение

высшего образования

«УЛЬЯНОВСКИЙ ГОСУДАРСТВЕННЫЙ ТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ»

**Лабораторная работа № 1**

*по дисциплине «Алгоритмы и структуры данных»*

Выполнил студент

группы ПИбд-12

Нгуен Т. М.

Проверил доцент кафедры

«Информационные системы»

Кулешов А.В.

Ульяновск, 2025

Задание:

Реализовать АТД **Queue** на основе структуры данных – **двусвязный список**.  
Реализовать 2 сортировки – **вставками** и **быстрая**.

Описание решения №1  
1. Queue на двусвязном списке (Doubly Linked List)

Queue – **FIFO** - Первый элемент зашел, первый элемент выходит. Представляет собой АТД, в котором элементы добавляются в начало и удаляются в конце.

Узел (Node) – Объект с полями данных, а также ссылками на предыдущий и следующий объект.

Так как у меня сфера области задания связано с планировщиком дел на неделю, методы будут называться не push, out и т.д., иначе будут

Основные методы (Двусвязного списка):

* public void MoreImportant(String data)

Добавляет элемент в начало (Как более важный)

* public void LessImportant(String data)

Добавляет элемент в конец (Как менее важный)

* public void RemoveMoreImportant()

Удаляет элемент в начале (Как более важный)

* public void RemoveLessImportant()

Удаляет элемент в конце (Как менее важный)

Основные методы (Очередь):

* public void In(int data)

Добавить элемент в начало

* public void Out()

Удалить элемент в конце

Двусвязный список

class LinkedList{

    private Node head;

    private Node tail;

    private static class Node{

        String data;

        Node next;

        Node prev;

        Node(String data){

            this.data = data; // Принимает переменную, переданную в основном файле

        }

    }

    public void MoreImportant(String data){ // Добавляет более важное дело в начало

        Node newNode = new Node(data); // Создается новая узел с ссылкой для хранения переменной

        if (head == null){ // Если список пустой, то первый элемент списка присваивает статус \*Начало\* и \*Конца\*

            head = tail = newNode;

        } else{ // Если список будет состоять как минимум 1 элемент до добавления нового, то ниже будут уже махинации для удаления двойного статуса у первого элемента

            newNode.next = head; // Перекидывает старый элемент (Начало) на +1 индекс

            head.prev = newNode; // Возвращается на -1 индекс и присваивает новый элемент

            head = newNode; // Ссылка на начало присваивает новый элемент, чтобы в дальнейшем его перекидывать на +1 индекс, если будут добавляться другие элементы в начало

        }

    }

    public void LessImportant(String data){

        Node newNode = new Node(data); // Аналогично работает с концом

        if (tail == null) {

            head = tail = newNode;

        } else{

            tail.next = newNode; // Новый элемент в конце

            newNode.prev = tail; // Предыдущий берет старый элемент

            tail = newNode; //Ссылка на конец забирает новый элемент для дальнейшей работы

        }

    }

    public void RemoveMoreImportant() { // Удаляет элемент в начале

        if (head == null) return; // Работает, если список пустой

        if (head == tail) { // Если список состоит из 1 элемента, то удаляет его

            head = tail = null;

        } else {

            head = head.next; // Ссылка на начало переходит на +1 индекс

            head.prev = null; // Самая начальный элемент удаляется

        }

    }

    public void RemoveLessImportant() {

        if (tail == null) return; // Аналогично работает и удалением в конце

        if (head == tail) {

            head = tail = null;

        } else {

            tail = tail.prev;

            tail.next = null;

        }

    }

    public void printList() { // Вывод списка для проверки

        Node current = head;

        while (current != null) {

            System.out.print(current.data + " <-> ");

            current = current.next;

        }

        System.out.println("null");

    }

**Очередь**

class Queue {

    private Node head;

    private Node tail;

    private static class Node{

        int data;

        Node next;

        Node prev;

        Node(int data){

            this.data = data;

        }

    }

    public void In(int data){

        Node newNode = new Node(data); // Аналогично работает как MoreImportant, просто взят из двусвязного списка

        if (head == null){

            head = tail = newNode;

        } else{

            newNode.next = head;

            head.prev = newNode;

            head = newNode;

        }

    }

    public void Out() {

        if (tail == null) return; // Работает аналогично как RemoveLessImportant, удаление в конце

        if (head == tail) {

            head = tail = null;

        } else {

            tail = tail.prev;

            tail.next = null;

        }

    }

    public void printList() {

        Node current = head;

        while (current != null) {

            System.out.print(current.data + " <-> ");

            current = current.next;

        }

        System.out.println("null");

    }

}

1. Сортировка вставкой (Insertion Sort)

Сортировка вставкой – это простой алгоритм сортировки, который эффективен на маленьких или почти отсортированных массивах.

Основные шаги:

* Начинаем со второго элемента массива, считая, что предыдущий отсортирован.
* Вставляем текущий элемент в нужное место среди отсортированных.
* Для этого сдвигаем элементы, которые больше текущего, вправо.
* Повторяем для всех элементов до конца массива.

**Средний и худший случаи:** O(n2) (если массив изначально отсортирован в обратном порядке).

**Лучший случай:** O(n) (если массив уже отсортирован).

* Сортировка вставкой работает **in place** (не требует дополнительной памяти) и **стабильна** (не меняет порядок равных элементов).

public class InsertSorting {

    public static int[] Insert(int[] array){

        for(int i = 0; i < array.length; i++){

            int element = array[i], j = i;

            while (j > 0 && element < array[j-1]){

                array[j] = array[j-1];

                j--;

            }

            array[j] = element;

        }

        return array;

    }

}

1. Быстрая сортировка (Quick Sort)

Быстрая сортировка – это алгоритм сортировки, который использует подход "разделяй и властвуй".

Основные шаги:

* Выбор опорного элемента (pivot): выбирается элемент, вокруг которого будет происходить разделение (обычно это первый, последний или средний элемент).
* Разделение: массив разделяется на две части, так что все элементы меньшие опорного оказываются слева, а большие или равные – справа.
* Рекурсивная сортировка: к каждой части применяется быстрая сортировка.
* Базисом рекурсии служит массив, состоящий из одного элемента (вырожденный случай — если массив состоит из одного элемента, значит, он уже отсортирован).

Быстрая сортировка работает эффективно на больших массивах и имеет среднюю временную сложность O(n log n).

В худшем случае быстрая сортировка может иметь временную сложность O(n2).

Алгоритм работает **in place,** но использует дополнительную память на стеке для рекурсивных вызовов.

public class FastSort {

    public static void quickSort(int[] arr, int low , int high){

        if (low < high) {

            int pivotIndex = partition(arr, low, high);

            quickSort(arr, low, pivotIndex -1);

            quickSort(arr, pivotIndex+1, high);

        }

    }

    private static int partition(int[] arr, int low, int high){

        int pivot = arr[high];

        int i = low - 1;

        for (int j = low; j <= high; j++){

            if (arr[j] < pivot) {

                i++;

                swap(arr, i , j);

            }

        }

        swap(arr, i+1 , high);

        return i+1;

    }

    private static void swap(int[] arr, int i, int j){

        int temp = arr[i];

        arr[i] = arr[j];

        arr[j] = temp;

    }

}

Для сравнения результатов и скорости работы массивов, сделал простой скрипт для создании картины на основе данных

import numpy as np

import matplotlib.pyplot as plt

insertSort = np.array([1.2527, 1.688, 1.9221, 3.369, 5.0867, 4.7298, 6.1659, 7.4381, 7.7073, 7.2387])

quickSort = np.array([1.1173, 1.3193, 1.3528, 1.6011, 1.8268, 1.6981, 1.8213, 1.9218, 1.8247, 2.4908])

numsOfTest = np.array([100, 200, 300, 400, 500 ,600, 700 ,800, 900, 1000])

plt.plot(numsOfTest, insertSort, label = "Сортировка вставками", marker = "o")

plt.plot(numsOfTest, quickSort, label = "Быстрая сортировка", marker = "o")

plt.title("Сранение времени работы сортировок")

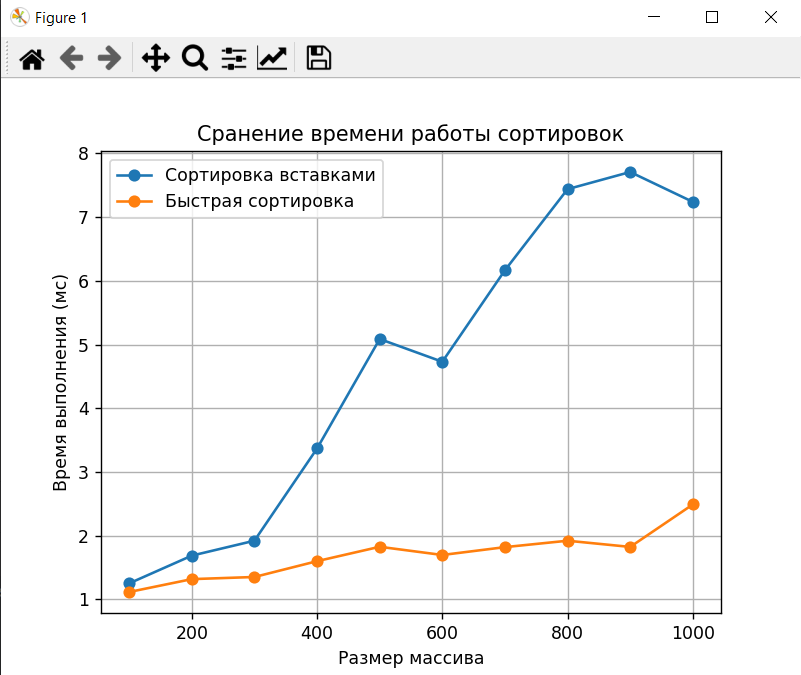
plt.xlabel("Размер массива")

plt.ylabel("Время выполнения (мс)")

plt.legend()

plt.grid()

plt.show()



### ****Вывод****

Если нужно отсортировать **большой массив**, особенно случайный, лучше использовать **быструю сортировку**. Если же массив **маленький** или **почти отсортирован**, сортировка **вставкой** может быть более эффективной. В реальных реализациях сортировки нередко комбинируют: например, в стандартных библиотеках сначала применяется **Quicksort**, а для небольших подмассивов переключается на **Insertion Sort** для оптимизации