Министерство науки и высшего образования Российской Федерации
Федеральное государственное бюджетное образовательное учреждение высшего образования
«Московский государственный технический университет имени Н.Э. Баумана
(национальный исследовательский университет)»
(МГТУ им. Н.Э. Баумана)

ФАКУЛЬТЕТ «Информатика и системы управления» (ИУ)

КАФЕДРА «Информационная безопасность» (ИУ8)

Отчёт

по лабораторной работе № 8
по дисциплине «Технологии и методы программирования»

Тема: «Практическая оценка сложности алгоритмов»

Вариант 4

Выполнил:
А. В. Куликова,
студент группы ИУ8-21М
Проверил: А. Ю. Быков

Москва, 2024

1. Постановка задачи

Задание на ЛР8. Практическая оценка сложности алгоритмов

Провести вычислительные эксперименты по оценки времени выполнения некоторых операций с контейнерами, в контейнерах хранятся целые числа. Необходимо измерять время выполнения заданных операций для пары контейнеров в соответствии со своим вариантом. Для измерения времени можно использовать класс MyTimer (Java), или свои разработки.

Для двух контейнеров протестировать операции добавления в контейнер и операции поиска элемента по ключу, представить две пары графиков:

- на одной координатной плоскости два графика - зависимости времени добавления элементов в контейнер от числа добавляемых элементов для двух заданных контейнеров (в начале контейнеры пустые), провести измерения не менее, чем в пяти точках;
- на другой координатной плоскости два графика - зависимости времени поиска элементов в контейнере от числа элементов контейнера (контейнеры уже заполнены в предыдущем тесте) для двух заданных контейнеров, провести измерения не менее, чем в пяти точках, при всех измерениях во всех случаях поиск проводить одинаковое число раз, например, 10000, 100000, 1000000, … значение определить экспериментально, с учет быстродействия компьютера, чтобы время посика было приемлемым.

Число элементов в контейнерах при измерениях определить экспериментально (одинаковые значения для двух контейнеров), чтобы время работы алгоритмов было приемлемым с учетом быстродействия компьютера. Контейнер заполнять с помощью генератора ПСЧ, ключи для поиска элементов получать с помощью генератора ПСЧ.

В отчете, кроме графиков, должны быть представлены теоретические оценки сложности выполняемых операций и выводы.

Варианты заданий:

1. Контейнер на основе двоичного дерева поиска и контейнер на основе списка, в списке добавление в начало.
2. Контейнер на основе двоичного дерева поиска и контейнер на основе списка, в списке добавление в конец.
3. Контейнер на основе двоичного дерева поиска и контейнер на основе массива, в массиве добавление в конец.
4. Контейнер на основе двоичного дерева поиска и контейнер на основе хэш-таблицы.
5. Контейнер на основе хэш-таблицы и контейнер на основе списка, в списке добавление в начало.
6. Контейнер на хэш-таблицы и контейнер на основе списка, в списке добавление в конец.
7. Контейнер на основе хэш-таблицы и контейнер на основе массива, в массиве добавление в конец.
8. Контейнер на основе двоичного дерева поиска и контейнер на основе массива, в массиве добавление в начало.
9. Контейнер на основе хэш-таблицы и контейнер на основе массива, в массиве добавление в начало.

2. Ход работы

Листинг 1 – Код программы main.java

```
import java.awt.BorderLayout;
import java.security.MessageDigest;
import java.security.NoSuchAlgorithmException;
import java.util.HashMap;
import java.util.Map;

import javax.swing.JFrame;
import javax.swing.SwingUtilities;

import org.jfree.chart.ChartFactory;
import org.jfree.chart.ChartPanel;
import org.jfree.chart.JFreeChart;
import org.jfree.chart.plot.PlotOrientation;
import org.jfree.data.xy.XYSeries;
import org.jfree.data.xy.XYSeriesCollection;

// Контейнер на основе двоичного дерева поиска
class BinarySearchTreeContainer {
    private Node root;

    private class Node {
        int key;
        Node left, right;

        public Node(int key) {
            this.key = key;
            left = null;
            right = null;
        }
    }

    public BinarySearchTreeContainer() {
        root = null;
    }

    public void insert(int key) {
        root = insertRec(root, key);
    }

    private Node insertRec(Node root, int key) {
        if (root == null) {
            root = new Node(key);
            return root;
        }
        if (key < root.key) {
            root.left = insertRec(root.left, key);
```

```
        } else if (key > root.key) {
            root.right = insertRec(root.right, key);
        }
        return root;
    }

    public boolean search(int key) {
        return searchRec(root, key);
    }

    private boolean searchRec(Node root, int key) {
        if (root == null) {
            return false;
        }
        if (root.key == key) {
            return true;
        }
        if (key < root.key) {
            return searchRec(root.left, key);
        } else {
            return searchRec(root.right, key);
        }
    }
}

// Контейнер на основе хэш-таблицы
public class HashMapContainer {
    private Map<Integer, String> map;

    public HashMapContainer() {
        map = new HashMap<>();
    }

    public void put(int key, String value) {
        map.put(key, value);
    }

    public String get(int key) {
        return map.get(key);
    }

    public void printAllEntries() {
        // System.out.println("All entries in the HashMap:");
        for (Map.Entry<Integer, String> entry : map.entrySet()) {
            System.out.println(
                    "Ключ: " + entry.getKey() + ", код хэша: " +
entry.getValue());
        }
    }
}
```

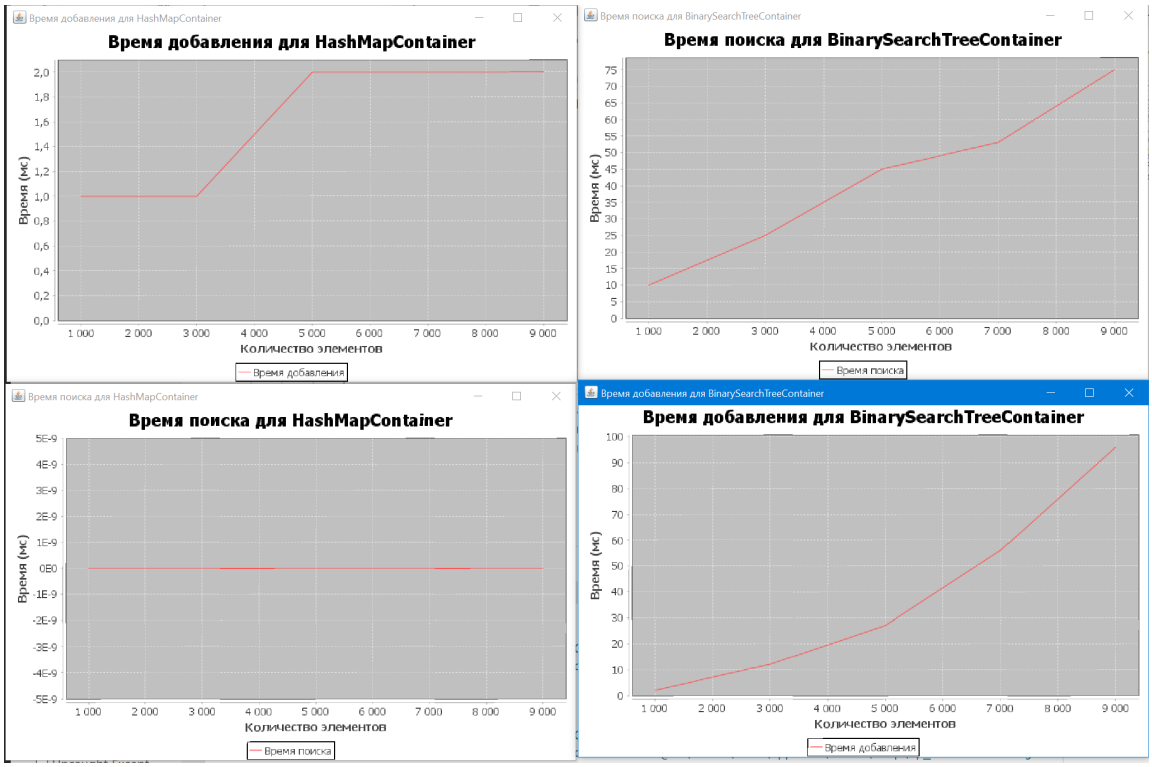
```
class Main {
    public static class MyTimer {
        private long startTime;

        public void start() {
            startTime = System.currentTimeMillis();
        }

        public long stop() {
            long endTime = System.currentTimeMillis();
            return endTime - startTime;
        }

        public MyTimer() {
            startTime = System.currentTimeMillis();
        }

        public void reset() {
            startTime = System.currentTimeMillis();
        }

        public long getTimeElapsed() {
            return System.currentTimeMillis() - startTime;
        }
    }

    private static void testAdditionAndSearch(Object container, String
containerType) {

        MyTimer timer = new MyTimer();
        XYSeries addSeries = new XYSeries("Время добавления");
        XYSeries searchSeries = new XYSeries("Время поиска");

        for (int i = 1000; i <= 10000; i += 2000) {
            timer.start();
            for (int j = 0; j < i; j++) {
                if (container instanceof BinarySearchTreeContainer) {
                    ((BinarySearchTreeContainer) container).insert(j);
                } else if (container instanceof HashMapContainer) {
                    ((HashMapContainer) container).put(j, "значение" + j);

                } else {
                    System.out.println("Неподдерживаемый тип контейнера");
                    return;
                }

            }

            long addTime = timer.stop();
            addSeries.add(i, addTime);
```

```
            // System.out.println("addTime: " + addTime + " ms");

            timer.start();
            for (int j = 0; j < 10000; j++) {
                int keyToSearch = j % i;
                if (container instanceof BinarySearchTreeContainer) {
                    ((BinarySearchTreeContainer) container).search(keyToSearch);
                } else if (container instanceof HashMapContainer) {
                    ((HashMapContainer) container).get(keyToSearch);
                }
            }
            long searchTime = timer.stop();
            searchSeries.add(i, searchTime);
        }

        createChart(addSeries, "Время добавления для " + containerType);
        createChart(searchSeries, "Время поиска для " + containerType);
    }

    private static void createChart(XYSeries series, String title) {
        XYSeriesCollection dataset = new XYSeriesCollection(series);
        JFreeChart chart = ChartFactory.createXYLineChart(title, "Количество
элементов",
                "Время (мс)", dataset,
                PlotOrientation.VERTICAL, true, true, false);

        SwingUtilities.invokeLater(() -> {
            JFrame frame = new JFrame(title);
            frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
            frame.setLayout(new BorderLayout());
            ChartPanel chartPanel = new ChartPanel(chart);
            frame.add(chartPanel, BorderLayout.CENTER);
            frame.pack();
            frame.setLocationRelativeTo(null);
            frame.setVisible(true);
        });
    }

    public static void main(String[] args) {
        BinarySearchTreeContainer bstContainer = new BinarySearchTreeContainer();
        testAdditionAndSearch(bstContainer, "BinarySearchTreeContainer");

        HashMapContainer hashMapContainer = new HashMapContainer();
        testAdditionAndSearch(hashMapContainer, "HashMapContainer");

    }
}
```

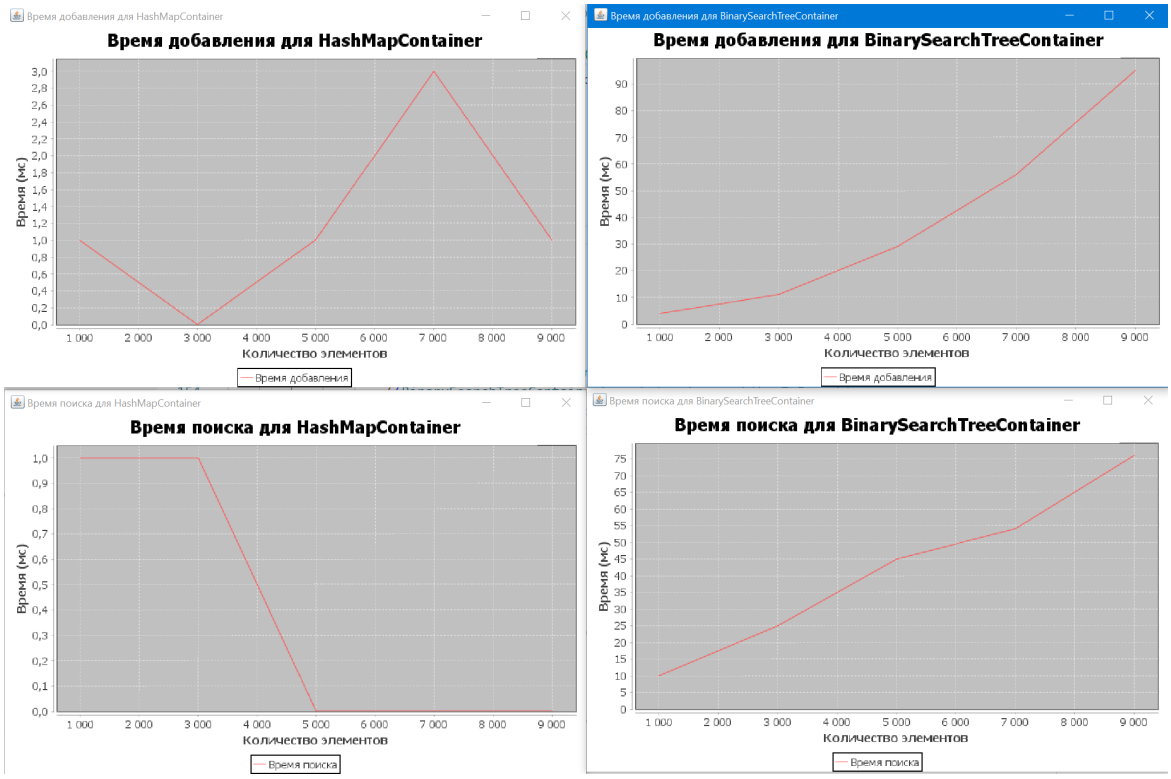
Результат с тестовым количеством [1000; 10000]:

Т.к. происходит генерация символов, то для лучшего сравнения проведем несколько испытаний

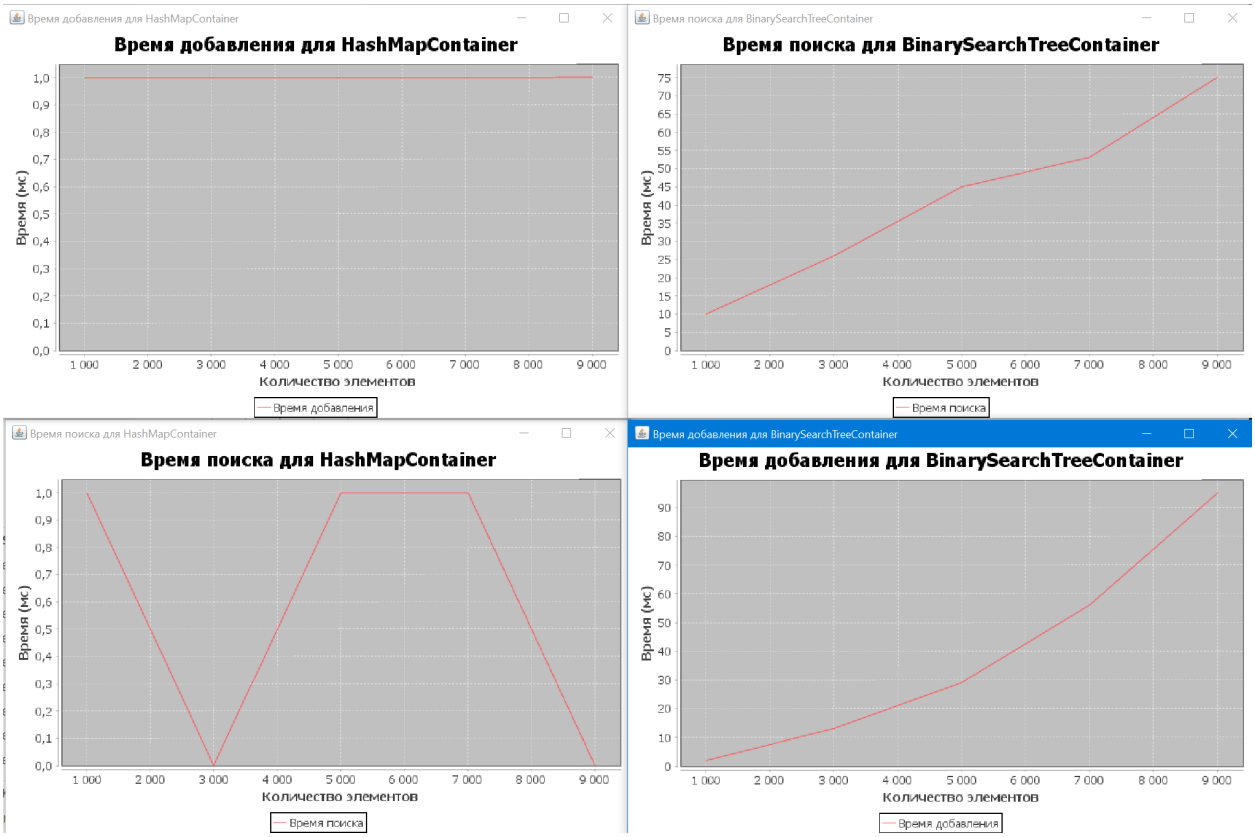
Испытание 1:



Испытание 2:



Испытание 3:



Контейнер на основе двоичного дерева поиска и контейнер на основе хэш-таблицы имеют различные графики времени выполнения операций добавления и поиска элементов из-за особенностей их структур.

1. **Двоичное дерево поиска**

- Добавление элемента. В среднем добавление элемента в двоичное дерево поиска требует времени O(log n), где n - количество элементов в дереве.
- Поиск элемента. Поиск элемента в двоичном дереве поиска также в среднем требует времени O(log n), так как процесс поиска зависит от высоты дерева.

2. **Хэш-таблица**

- Добавление элемента. В среднем добавление элемента в хэш-таблицу требует времени O(1).

- Поиск элемента. При хорошей хэш-функции поиск элемента в хэш-таблице также требует времени O(1).

Из-за различий в алгоритмах добавления и поиска элементов у двоичного дерева поиска и хэш-таблицы, графики времени выполнения операций могут различаться.

Двоичное дерево поиска будет иметь более плавный график с увеличением количества элементов из-за логарифмической сложности, в то время как у хэш-таблицы график будет более стабильным и близким к константе за счет O(1) времени доступа при хорошей хэшировании.