Министерство науки и высшего образования Российской Федерации
Федеральное государственное автономное 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
КАФЕДРА СИСТЕМЫ ОБРАБОТКИ ИНФОРМАЦИИ И УПРАВЛЕНИЯ

ОТЧЕТ
ПО ЛАБОРАТОРНОЙ РАБОТЕ

НА ТЕМУ:

Исследование структур данных и алгоритмов: сбалансированные деревья, алгоритмы сортировки и численные методы

Студент ИУ5-14Б (группа) _____ (подпись, дата) Г.И. Корнеев (И.О. Фамилия)

Руководитель курсовой работы _____ (подпись, дата) М.И. Колосов (И.О. Фамилия)

2025 г.

Целью лабораторной работы является изучение и реализация алгоритмов обработки данных и структур данных, а именно: построение AVL-дерева и 2–3-дерева для списка из n различных целых чисел, реализация алгоритмов сортировки слиянием, быстрой и пирамидальной сортировки, а также исследование их производительности на массивах различного размера и структуры входных данных. Дополнительно целью работы является освоение представления пирамиды в виде массива при пирамидальной сортировке и анализ вычислительной сложности стадии обратной подстановки метода исключения Гаусса.

Постановка задачи:

1. Построение AVL-дерева;
2. Построение 2-3-дерева;
3. Реализация сортировок (слияние, быстрая, пирамидальная);
4. Исследование производительности для заданных размеров и типов входных данных;
5. Сортировка заданных списков пирамидальной сортировкой с представлением пирамиды в виде массива;
6. Анализ стадии обратной подстановки метода Гаусса и её асимптотики $\Theta(n^2)$.

4. Теоретическая часть

1. AVL-дерево: идея балансировки;
2. 2-3-дерево: структура узлов;
3. Merge / Quick / Heap sort — по 3–5 предложений;
4. Обратная подстановка Гаусса — идея алгоритма и формула сложности.

Практическая часть (реализация)

5.1 Структура проекта

Программная реализация лабораторной работы выполнена на языке программирования C++ с использованием системы сборки CMake. Проект имеет модульную структуру, обеспечивающую удобство сборки и сопровождения кода.

Исходные файлы разделены на заголовочные файлы и файлы реализации. Для визуализации результатов экспериментальных исследований используется отдельный скрипт на языке Python. Результаты измерений времени работы алгоритмов сортировки сохраняются в CSV-файл и используются для построения графиков.

Фактическая структура каталогов проекта представлена ниже:

```
lab11-part2/
├── build/
├── include/
```

```
    ├── avl_tree.h
    ├── gauss_back_substitution.h
    ├── sort_algorithms.h
    └── two_three_tree.h
├── src/
    ├── avl_tree.cpp
    ├── gauss_back_substitution.cpp
    ├── main.cpp
    ├── sort_algorithms.cpp
    └── two_three_tree.cpp
├── scripts/
    └── visualize.py
├── plots/
    ├── plot_random.png
    ├── plot_ascending.png
    └── plot_descending.png
├── results.csv
├── CMakeLists.txt
├── README.md
└── .gitignore
```

5.2 AVL-дерево представляет собой самобалансирующееся бинарное дерево поиска, в котором для каждого узла разность высот левого и правого поддеревьев не превышает единицы.

В данной работе AVL-дерево используется для построения структуры по списку из *n* различных целых чисел. В программе реализована структура узла, содержащая ключ, указатели на потомков и высоту узла. Вставка элементов выполняется по правилам бинарного дерева поиска. После каждой вставки производится пересчёт высот узлов и вычисление коэффициента баланса, что позволяет своевременно обнаруживать нарушение балансировки.

Для восстановления баланса дерева реализованы стандартные повороты AVL-дерева: левый, правый и двойные повороты (LR и RL), применяемые в зависимости от типа дисбаланса. Реализация поворотов AVL-дерева и процедуры вставки с балансировкой приведена в приложении А (листинги А.1–А.3).

Корректность построения AVL-дерева проверяется с помощью симметричного (inorder) обхода. Для бинарного дерева поиска данный обход выводит элементы в возрастающем порядке, что подтверждает правильность реализации алгоритма.

5.3 Реализация 2–3-дерева

2–3-дерево является сбалансированным деревом поиска, в котором каждый узел может содержать один или два ключа. В работе реализовано построение 2–3-дерева по списку из *n* различных целых чисел с последовательной вставкой элементов.

При переполнении узлов выполняется их разделение с подъёмом среднего ключа, что обеспечивает сохранение балансировки дерева. Корректность реализации проверяется с помощью inorder-обхода, результат которого представляет собой упорядоченную по возрастанию последовательность элементов. Реализация алгоритма вставки в 2–3-дерево с разделением переполненных узлов приведена в приложении Б (листинги Б.1–Б.2).

5.4 Реализация алгоритмов сортировки

В лабораторной работе реализованы три алгоритма сортировки: сортировка слиянием, быстрая сортировка и пирамидальная сортировка. Алгоритмы применяются к массивам целых чисел и используются для исследования производительности.

Измерение времени выполнения производится на массивах размером

$$n = 10^2, 10^3, 10^4, 10^5, 10^6$$

для случайных, возрастающих и убывающих входных данных. Реализация алгоритмов сортировки слиянием, быстрой и пирамидальной сортировки приведена в приложении В (листинги В.1–В.3).

5.5 Пирамидальная сортировка с представлением пирамиды в виде массива

Для пирамидальной сортировки реализован пошаговый вывод состояния массива, представляющего пирамиду. В процессе работы алгоритма отображается исходный массив, массив после построения пирамиды и промежуточные состояния после каждого извлечения максимального элемента.

Данный вывод позволяет наглядно проследить процесс работы пирамидальной сортировки. Реализация пирамидальной сортировки с пошаговым выводом промежуточных состояний массива приведена в приложении Г (листинги Г.1–Г.2).

6. Экспериментальные исследования

Проведено исследование производительности сортировки слиянием, быстрой и пирамидальной сортировки на массивах размером

$$n = 10^2 \dots 10^6$$

для случайных, возрастающих и убывающих входных данных.

По полученным данным построены графики зависимости времени выполнения от размера массива (рисунки 1–3). Результаты экспериментов подтверждают теоретические оценки временной сложности алгоритмов сортировки. Скрипт построения графиков по результатам измерений приведён в приложении Е (листинги Е.1–Е.3).

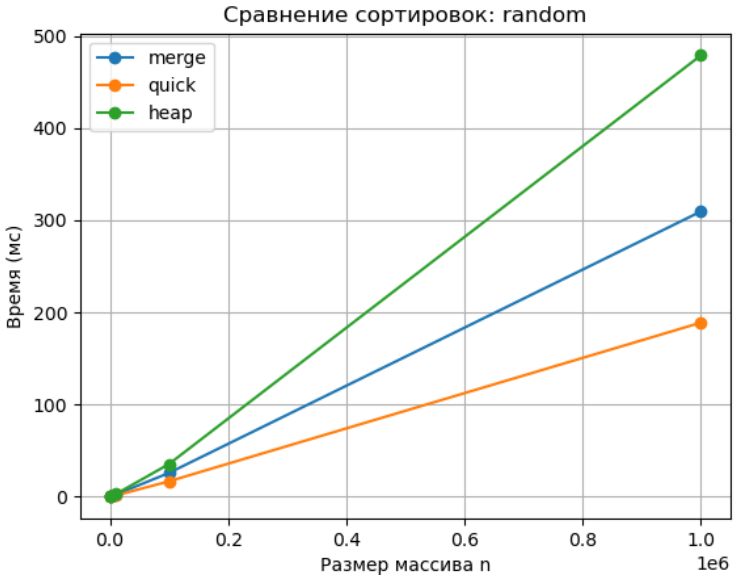

Рисунок 1 - Сравнение времени работы сортировок на случайных данных

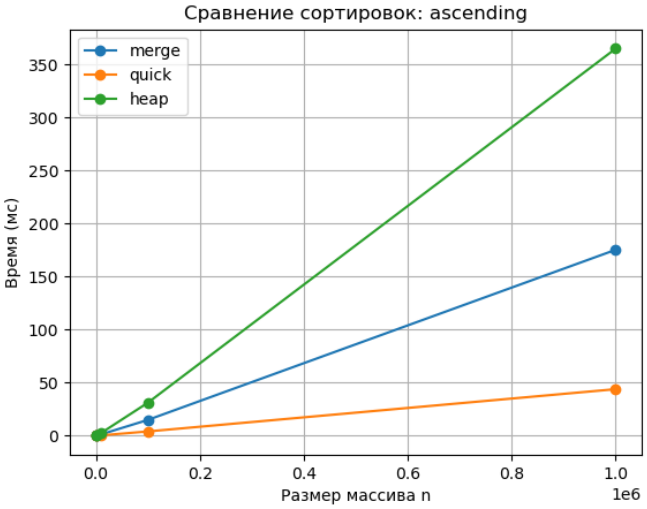

Рисунок 2 — Сравнение времени работы сортировок на возрастающих данных

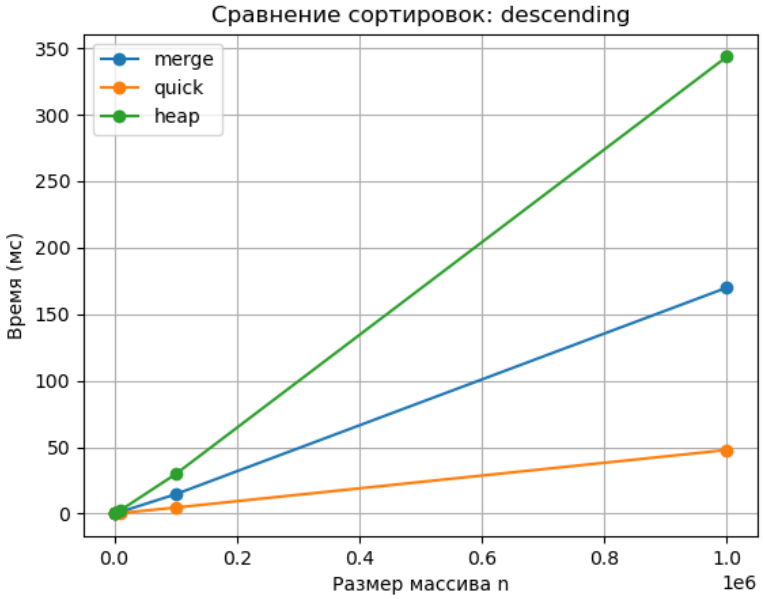

Рисунок 3 — Сравнение времени работы сортировок на убывающих данных

Вывод

В ходе выполнения лабораторной работы были реализованы структуры данных AVL-дерево и 2–3-дерево, а также алгоритмы сортировки слиянием, быстрой и пирамидальной сортировки. Проведено экспериментальное исследование производительности алгоритмов сортировки на массивах различного размера и структуры входных данных. Полученные результаты подтвердили теоретические оценки временной сложности алгоритмов. Также была рассмотрена стадия обратной подстановки метода исключения Гаусса и показано, что её вычислительная сложность равна $\Theta(n^2)$.

ПРИЛОЖЕНИЯ

Ниже приведены сокращённые фрагменты (по ключевым строкам) из исходного кода. Полный код находится в электронном приложении (архив проекта).

Приложение А — AVL-дерево

Листинг А.1 — Поворот вправо (файл: src/avl_tree.cpp)

```
AVLNode* rotateRight(AVLNode* y) {
    AVLNode* x = y->left;
    AVLNode* T2 = x->right;

    // Perform rotation
    x->right = y;
    y->left = T2;

    // Update heights
    updateHeight(y);
    updateHeight(x);

    // Return new root
    return x;
}
```

Листинг А.2 — Поворот влево (файл: src/avl_tree.cpp)

```
AVLNode* rotateLeft(AVLNode* x) {
    AVLNode* y = x->right;
    AVLNode* T2 = y->left;

    // Perform rotation
    y->left = x;
    x->right = T2;

    // Update heights
    updateHeight(x);
    updateHeight(y);

    // Return new root
    return y;
}
```

Листинг А.3 — Вставка и балансировка (файл: src/avl_tree.cpp)

```
AVLNode* avlInsert(AVLNode* node, int key) {
    // Perform normal BST insertion
    if (node == nullptr)
        return new AVLNode(key);

    if (key < node->key)
        node->left = avlInsert(node->left, key);
    else if (key > node->key)
        node->right = avlInsert(node->right, key);
    else
        return node; // Duplicate keys are not inserted

    // Update this node's height
    updateHeight(node);

    // Check the balance factor to see if this node became unbalanced
    int balance = getBalance(node);

    // Left Left Case
    if (balance > 1 && key < node->left->key)
        return rotateRight(node);
```

```
    // ... (фрагмент сокращён) ...

    // Right Left Case
    if (balance < -1 && key < node->right->key) {
        node->right = rotateRight(node->right);
        return rotateLeft(node);
    }

    // Return unchanged node pointer
    return node;
}
```

Приложение Б — 2–3-дерево

Листинг Б.1 — Вставка в поддерево и split (файл: src/two_three_tree.cpp)

```
static InsertResult insertInternal(Node23* node, int key) {
    InsertResult res;
    // If this node is a leaf, insert the key here
    if (node->isLeaf()) {
        // Insert the key in sorted order
        node->keys.insert(std::upper_bound(node->keys.begin(), node-
>keys.end(), key), key);
        // If the node now has three keys, we need to split
        if (node->keys.size() == 3) {
            // keys are sorted; median is at index 1
            int median = node->keys[1];
            Node23* right = new Node23();
            // right node gets the largest key
            right->keys.push_back(node->keys[2]);
            // left node (current) keeps the smallest key
            node->keys.erase(node->keys.begin() + 1, node->keys.end());
            res.hasPromoted = true;
            res.promotedKey = median;
            res.rightChild = right;
        }
        return res;
    }

    // Non-leaf: determine which child to descend into
    size_t i = 0;
    if (node->keys.size() == 1) {
        if (key < node->keys[0]) i = 0;
        else i = 1;
    } else { // size == 2
    // ... (фрагмент сокращён) ...
            node->keys.erase(node->keys.begin() + 1, node->keys.end());
            node->children.erase(node->children.begin() + 2, node-
>children.end());
            // Prepare result for parent
            res.hasPromoted = true;
            res.promotedKey = median;
            res.rightChild = right;
            return res;
        }
    }
    // No split at this level
    return res;
}
```

Листинг Б.2 — Вставка в дерево (файл: src/two_three_tree.cpp)

```
void insert23(Node23*& root, int key) {
    if (!root) {
        root = new Node23();
        root->keys.push_back(key);
        return;
    }
    InsertResult res = insertInternal(root, key);
    // If the root split, create a new root
    if (res.hasPromoted) {
        Node23* newRoot = new Node23();
        newRoot->keys.push_back(res.promotedKey);
        newRoot->children.push_back(root);
        newRoot->children.push_back(res.rightChild);
        root = newRoot;
    }
}
```

Приложение B — Алгоритмы сортировки

Листинг B.1 — Merge sort (файл: src/sort_algorithms.cpp)

```
void mergeSort(std::vector<int>& arr, std::size_t l, std::size_t r) {
    if (arr.empty() || l >= r) return;
    std::vector<int> temp(arr.size());
    mergeSortRec(arr, temp, l, r);
}
```

Листинг B.2 — Quick sort (файл: src/sort_algorithms.cpp)

```
void quickSort(std::vector<int>& arr, std::size_t l, std::size_t r) {
    if (arr.empty() || l >= r) return;
    quickSortInternal(arr, static_cast<long long>(l), static_cast<long
long>(r));
}
```

Листинг B.3 — Heap sort (файл: src/sort_algorithms.cpp)

```
void heapSort(std::vector<int>& arr) {
    std::size_t n = arr.size();
    if (n <= 1) return;
    // Build max heap
    for (long long i = static_cast<long long>(n) / 2 - 1; i >= 0; --i) {
        heapify(arr, n, static_cast<std::size_t>(i));
    }
    // One by one extract elements
    for (long long i = static_cast<long long>(n) - 1; i > 0; --i) {
        std::swap(arr[0], arr[static_cast<std::size_t>(i)]);
        heapify(arr, static_cast<std::size_t>(i), 0);
    }
}
```

Приложение Г — Пирамидальная сортировка (пошагово)

Листинг Г.1 — Heap sort с шагами (int) (файл: src/sort_algorithms.cpp)

```
void heapSortWithSteps(std::vector<int>& arr) {
    std::size_t n = arr.size();
    if (n <= 1) return;

    std::cout << "Исходный массив: ";
    printIntArray(arr);

    // 1) Построение max-heap
    for (long long i = static_cast<long long>(n) / 2 - 1; i >= 0; --i) {
        heapify(arr, n, static_cast<std::size_t>(i));
    }
    std::cout << "После построения пирамиды (max-heap): ";
    printIntArray(arr);

    // 2) Извлечение элементов
    for (long long i = static_cast<long long>(n) - 1; i > 0; --i) {
        std::swap(arr[0], arr[static_cast<std::size_t>(i)]);
        heapify(arr, static_cast<std::size_t>(i), 0);

        std::cout << "Шаг: вынесли максимум на позицию " << i << ", массив:
";
        printIntArray(arr);
    }

    std::cout << "Отсортировано: ";
    printIntArray(arr);
}
```

Листинг Г.2 — Heap sort с шагами (char) (файл: src/sort_algorithms.cpp)

```
void heapSortCharWithSteps(std::vector<char>& arr) {
    std::size_t n = arr.size();
    if (n <= 1) return;

    std::cout << "Исходный массив: ";
    printCharArray(arr);

    for (long long i = static_cast<long long>(n) / 2 - 1; i >= 0; --i) {
        heapifyChar(arr, n, static_cast<std::size_t>(i));
    }
    std::cout << "После построения пирамиды (max-heap): ";
    printCharArray(arr);

    for (long long i = static_cast<long long>(n) - 1; i > 0; --i) {
        std::swap(arr[0], arr[static_cast<std::size_t>(i)]);
        heapifyChar(arr, static_cast<std::size_t>(i), 0);

        std::cout << "Шаг: вынесли максимум на позицию " << i << ", массив:
";
        printCharArray(arr);
    }

    std::cout << "Отсортировано: ";
    printCharArray(arr);
}
```

Приложение Д — Обратная подстановка Гаусса

Листинг Д.1 — backSubstitution(U, b) (файл: src/gauss_back_substitution.cpp)

```
std::vector<double> backSubstitution(const std::vector<std::vector<double>>&
U,
                                     const std::vector<double>& b) {
    std::size_t n = U.size();
    if (n == 0) return {};
    if (b.size() != n) {
        throw std::runtime_error("Размеры U и b не совпадают.");
    }
    for (std::size_t i = 0; i < n; ++i) {
        if (U[i].size() != n) {
            throw std::runtime_error("Матрица U должна быть квадратной
n×n.");
        }
    }

    std::vector<double> x(n, 0.0);

    // Идём снизу вверх: i = n-1, n-2, ..., 0
    for (long long i = static_cast<long long>(n) - 1; i >= 0; --i) {
        double sum = 0.0;
        for (std::size_t j = static_cast<std::size_t>(i) + 1; j < n; ++j) {
            sum += U[static_cast<std::size_t>(i)][j] * x[j];
        }
        double diag =
U[static_cast<std::size_t>(i)][static_cast<std::size_t>(i)];
        if (diag == 0.0) {
            throw std::runtime_error("Нулевой элемент на диагонали — деление
невозможно.");
        }
        x[static_cast<std::size_t>(i)] = (b[static_cast<std::size_t>(i)] -
sum) / diag;
    }

    return x;
}
```

Листинг Д.2 — Псевдокод и Θ(n^2) (файл: src/gauss_back_substitution.cpp)

```
void printBackSubstitutionPseudocodeAndComplexity() {
    std::cout << "\n==============================\n";
    std::cout << "Пункт 5: Обратная подстановка (метод Гаусса)\n";
    std::cout << "==============================\n\n";

    std::cout << "Псевдокод (стадия обратной подстановки для Ux = b):\n";
    std::cout << "-----------------------------------------------\n";
    std::cout << "for i = n-1 .. 0:\n";
    std::cout << "    sum = 0\n";
    std::cout << "    for j = i+1 .. n-1:\n";
    std::cout << "        sum = sum + U[i][j] * x[j]\n";
    std::cout << "    x[i] = (b[i] - sum) / U[i][i]\n";
    std::cout << "-----------------------------------------------\n\n";

    std::cout << "Почему время работы Θ(n^2):\n";
    std::cout << "Внешний цикл выполняется n раз (i = n-1 .. 0).\n";
    std::cout << "Внутри для каждого i выполняется суммирование по j от i+1
до n-1,\n";
    std::cout << "то есть примерно (n-1) + (n-2) + ... + 1 = n(n-1)/2
операций.\n";
    std::cout << "Это квадратичная функция, поэтому сложность Θ(n^2).\n\n";
}
```

Приложение Е — Визуализация результатов

Листинг Е.1 — load_results (файл: scripts/visualize.py)

```
def load_results(csv_path):
    """Load results from a CSV file into a nested dictionary.

    Returns a mapping of distribution -> algorithm -> list of (n, time_ms).
    """
    data = defaultdict(lambda: defaultdict(list))
    with open(csv_path, newline='', encoding='utf-8') as f:
        reader = csv.DictReader(f)
        for row in reader:
            alg = row['algorithm']
            n = int(row['n'])
            dist = row['distribution']
            time_ms = float(row['time_ms'])
            data[dist][alg].append((n, time_ms))
    # Sort each list by n to ensure lines connect in order
    for dist in data:
        for alg in data[dist]:
            data[dist][alg].sort(key=lambda x: x[0])
    return data
```

Листинг Е.2 — plot_distributions (файл: scripts/visualize.py)

```
def plot_distributions(data, output_dir):
    """Plot running times for each distribution and save PNG files."""
    for dist, alg_data in data.items():
        plt.figure()
        for alg, points in alg_data.items():
            ns = [p[0] for p in points]
            times = [p[1] for p in points]
            plt.plot(ns, times, marker='o', label=alg)
        plt.xlabel('Размер массива n')
        plt.ylabel('Время (мс)')
        plt.title(f'Сравнение сортировок: {dist}')
        plt.legend()
        plt.grid(True)
        # Save using distribution name
        filename = f'plot_{dist}.png'.replace(' ', '_')
        plt.savefig(os.path.join(output_dir, filename))
        plt.close()
        print(f'Plot saved to {filename}')
```

Листинг Е.3 — main (файл: scripts/visualize.py)

```
def main():
    if len(sys.argv) > 2:
        print("Usage: python visualize.py [results.csv]")
        sys.exit(1)
    csv_path = sys.argv[1] if len(sys.argv) == 2 else 'results.csv'
    if not os.path.isfile(csv_path):
        print(f'Error: {csv_path} does not exist')
        sys.exit(1)
    data = load_results(csv_path)
    output_dir = os.path.join(os.path.dirname(os.path.abspath(csv_path)),
'plots')
    os.makedirs(output_dir, exist_ok=True)
    plot_distributions(data, output_dir)
```