Министерство науки и высшего образования Российской Федерации

Федеральное государственное бюджетное образовательное учреждение высшего образования

«Российский химико-технологический университет имени Д.И. Менделеева»

Кафедра информационных компьютерных технологий

ОТЧЕТ ПО ЛАБОРАТОРНОЙ РАБОТЕ № 2

Выполнил студент группы КС-36 Полковникова Д.Д.

Ссылка на репозиторий: https://github.com/MUCTR-IKT-CPP/Polkovnikova\_CS-36

Приняли: Пысин Максим Дмитриевич

Краснов Дмитрий Олегович

Лобанов Алексей Владимирович

Крашенинников Роман Сергеевич

Дата сдачи: 02.03.2025

Оглавление

[Описание задачи. 2](#_Toc191747995)

[Описание метода/модели. 3](#_Toc191747996)

[Выполнение задачи. 4](#_Toc191747997)

[Заключение. 11](#_Toc191747998)

# Описание задачи.

* Реализовать проведения тестирования алгоритма сериями расчетов для измерения параметров времени. За один расчет выполняется следующие операции:
  1. Генерируется массив случайных значений
  2. Запоминается время начала расчета алгоритма сортировки
  3. Выполняется алгоритм сортировки
  4. Вычисляется время, затраченное на сортировку: текущее время - время начала
  5. Сохраняется время для одной попытки. После этого расчет повторяется до окончания серии.
  6. Алгоритм вычисляется 8 сериями по 20 раз за серию.
  7. Алгоритм в каждой серии вычисляется для массива размером M. (1000, 2000, 4000, 8000, 16000, 32000, 64000, 128000)
  8. Массив заполняется значения числами с плавающей запятой в интервале от -1 до 1.
  9. Для серии запоминаются, все времена которые были замерены
* При работе сортировки подсчитать:
  1. общее количество вызовов функции построения кучи
  2. количество вызовов внутренней функции построения кучи(вызов внутри функции формирования кучи)
  3. времени исполнения сортировки
* По полученным данным времени построить графики зависимости времени от числа элементов в массиве:
  1. Совмещённый график наихудшего времени выполнения сортировки и сложности алгоритма указанной в нотации O большое.

Для построения графика вычисляется O большое для каждого размера массива. При этом при вычислении функции O(c \* g(N)) подбирается такая константа с, что бы при значении > 1000 график O(N) был выше графика наихудшего случая, но второй график на его фоне не превращался в прямую линию

* 1. Совмещенный график среднего, наилучшего и наихудшего времени исполнения.
  2. Совмещенный график среднего, наилучшего и наихудшего глубины рекурсии.
  3. Совмещенный график среднего по серии количетсва вызовов функции построения кучи и количетсва вызовов внутренней функции.
  4. График среднего процентного соотношения вызовово внутренней функции к общему вызову функции.
* По результатом расчетов оформляется отчет по предоставленной форме, в отчете:
  1. Приводится описание алгоритма.
  2. Приводится описания выполнения задачи (Описание кода и спецефических элементов реализации)
  3. Приводятся выводы (Графики и их анализ). Требуется ответить на вопрос о поведении алгоритма изученного в процессе выполнения лаборатной работы и зафиксировать его особенности.

# Описание метода/модели.

**Алгоритм пирамидальной сортировки** (также называют «сортировка кучей») **основан на построении и изменении бинарного дерева таким образом, чтобы в его корне всегда находилось максимальное значение**.

**Основные шаги алгоритма**:

1. **Получение бинарного дерева из несортированного массива**. Начиная с нулевого элемента последовательно заполняют уровни дерева слева направо.
2. **Преобразование дерева** таким образом, чтобы в корне дерева находился максимальный элемент, а значение в каждом узле было больше, чем значения в его потомках. Для этого смотрят на предпоследний уровень дерева и для каждого узла проверяют, выполняется условие: каждый потомок должен быть меньше своего родителя. Если нет, то находят максимальный элемент среди потомков и меняют его местом с родителем. Так проходят по каждому уровню дерева снизу вверх и в результате получают сортирующее дерево, в корне которого находится максимальное число.
3. **Поменяют местами нулевой и последний элемент массива и «отрежут» ветку с максимальным элементом**.
4. **Продолжают приводить дерево к такому виду**, чтобы выполнялось условие: каждый потомок должен быть меньше своего родителя. Действия похожи, только теперь идут сверху вниз и выполняют перестановки там, где условие нарушается.

Процесс построения кучи повторяют до тех пор, пока все вершины дерева не будут удалены.

Асимптотическая сложность пирамидальной сортировки — **O(n log n)**.

**Преимущества Heap Sort**

1. Гарантированная сложность O(n log n)
   * В отличие от быстрой сортировки, которая может деградировать до O(n2)O(n^2)O(n2) в худшем случае, пирамидальная сортировка всегда работает за O(nlog⁡n)O(n \log n)O(nlogn).
2. Не требует дополнительной памяти *(в отличие от Merge Sort, который требует O(n)O(n)O(n) памяти)*
   * Heap Sort выполняется на месте (in-place) и требует только O(1) дополнительной памяти, так как вся работа ведётся внутри массива.
3. Подходит для потоковой обработки (онлайн-сортировка)
   * Кучу можно использовать для обработки данных на лету (например, в приоритетных очередях).
4. Хорош для работы с большими данными
   * Его производительность не сильно зависит от начального распределения элементов (в отличие от Quick Sort, который может замедляться на уже отсортированных данных).

**Недостатки Heap Sort**

1. Хуже кэш-локальность по сравнению с Quick Sort
   * Куча работает через разреженный доступ к памяти, из-за чего процессорный кеш используется менее эффективно, чем в быстрой сортировке.
   * Это приводит к медленной работе на современных CPU, где последовательный доступ предпочтительнее.
2. Не является стабильной сортировкой
   * Heap Sort не сохраняет порядок равных элементов, что может быть критично для некоторых задач.
3. Практически медленнее, чем Quick Sort
   * Несмотря на O(nlogn) сложность в худшем случае, быстрая сортировка на практике работает быстрее за счёт кэш-оптимизации и меньшего числа перестановок.
4. Неэффективен для небольших массивов
   * Для маленьких массивов (до 1000 элементов) простые алгоритмы вроде Insertion Sort работают быстрее из-за меньших накладных расходов.

# Выполнение задачи.

Для расчёта промежутков времени для выполнения сортировки был использован язык программирования С++, а для построения графиков - Excel.

**Код:**

#include <iostream>

#include <vector>

#include <random>

#include <chrono>

#include <fstream>

using namespace std;

using namespace chrono;

// Глобальные счётчики вызовов heapify

uint64\_t totalHeapifyCalls = 0;       // Всего вызовов heapify

uint64\_t internalHeapifyCalls = 0;    // Внутренние (рекурсивные) вызовы heapify

// Функция генерации случайных чисел в диапазоне [-1, 1]

vector<double> generateNumbers(int N) {

    random\_device rd;

    mt19937 gen(rd());

    uniform\_real\_distribution<double> dist(-1.0, 1.0);

    vector<double> numbers(N);

    for (double &num : numbers) {

        num = dist(gen);

    }

    return numbers;

}

// Вспомогательная функция для просеивания вниз в пирамидальной сортировке

void heapify(vector<double>& arr, int n, int i, bool isInternal = false) {

    totalHeapifyCalls++; // Каждый вызов heapify

    if (isInternal) internalHeapifyCalls++; // Если рекурсивный вызов

    int largest = i;

    int left = 2 \* i + 1;

    int right = 2 \* i + 2;

    if (left < n && arr[left] > arr[largest])

        largest = left;

    if (right < n && arr[right] > arr[largest])

        largest = right;

    if (largest != i) {

        swap(arr[i], arr[largest]);

        heapify(arr, n, largest, true); // Передаём true, так как это внутренний вызов

    }

}

// Функция пирамидальной сортировки (Heap Sort)

void heapSort(vector<double>& arr) {

    int n = arr.size();

    // Обнуление счетчиков перед сортировкой

    totalHeapifyCalls = 0;

    internalHeapifyCalls = 0;

    // Построение кучи

    for (int i = n / 2 - 1; i >= 0; i--) {

        heapify(arr, n, i);

    }

    // Извлечение элементов из кучи

    for (int i = n - 1; i > 0; i--) {

        swap(arr[0], arr[i]);

        heapify(arr, i, 0);

    }

}

int main() {

    vector<int> sizes = {1000, 2000, 4000, 8000, 16000, 32000, 64000, 128000};

    // Открываем CSV-файл

    ofstream file("results.csv");

    if (!file.is\_open()) {

        cerr << "Ошибка: не удалось открыть файл для записи.\n";

        return 1;

    }

    // Заголовки столбцов в CSV

    file << "Серия,Размер массива,Попытка,Время (сек),Всего вызовов heapify,Внутренних вызовов heapify\n";

    for (size\_t series = 0; series < sizes.size(); series++) {

        int size = sizes[series];

        for (int attempt = 0; attempt < 20; attempt++) {

            vector<double> numbers = generateNumbers(size);

            auto start = high\_resolution\_clock::now();

            heapSort(numbers);

            auto end = high\_resolution\_clock::now();

            double elapsedTime = duration<double>(end - start).count();

            // Записываем результаты в CSV

            file << (series + 1) << "," << size << "," << (attempt + 1) << ","

                 << elapsedTime << "," << totalHeapifyCalls << "," << internalHeapifyCalls << "\n";

        }

    }

    file.close();

    cout << "Результаты сохранены в файл results.csv\n";

    return 0;

}

Этот код реализует пирамидальную (кучную) сортировку (Heap Sort) и анализирует её производительность.  
Он выполняет следующие **ключевые задачи**:

**1. Генерация случайных чисел**

* Генерирует массив случайных вещественных чисел в диапазоне [-1, 1].
* Используется равномерное распределение (uniform distribution).

**2. Функция heapify (просеивание вниз)**

* Поддерживает свойства максимальной кучи.
* Перемещает наибольший элемент вверх, сравнивая с потомками.
* Подсчитывает количество всех вызовов и внутренних рекурсивных вызовов.

**3. Основная программа Main():**

1. Открывает CSV-файл results.csv для сохранения результатов.
2. Проходит по разным размерам массива (от 1000 до 128000 элементов).
3. Для каждого размера выполняет 20 попыток сортировки.
4. Измеряет время выполнения сортировки (high\_resolution\_clock).
5. Сохраняет данные в CSV-файл:
   * Номер серии
   * Размер массива
   * Попытка
   * Время выполнения
   * Общее число вызовов heapify
   * Число рекурсивных вызовов heapify

Сортирует массив пирамидальной сортировкой (Heap Sort)

Подсчитывает количество вызовов heapify (всех и внутренних)

Измеряет время работы

Сохраняет результаты в CSV-файл

Этот код полезен для анализирования производительности Heap Sort и создания графиков для разных размеров массива.

Из данных получаются следующие **графики**:

Рисунок 1 – Зависимость наихудшего, наилучшего и среднего значений времени от размера массива.

Рисунок 2 – Зависимость наихудшего, наилучшего и среднего значений глубины рекурсии от размера массива.

Рисунок 3 – Наихудший случай и сложность алгоритма.

Рисунок 4 – Средние значения количества вызовов функци построения кучи и количества вызовов внутренних функций.

Рисунок 5 – Среднее процентное соотношение вызовов внутренней функции к общему вызову функции.

# Заключение.

Heap Sort — это надёжный и гарантированно эффективный алгоритм сортировки, работающий за O(n log n)) во всех случаях. Он основан на двоичной куче (Heap) и выполняется на месте, не требуя дополнительной памяти, как Merge Sort. Идеальное применение: большие массивы, где нельзя рисковать деградацией Quick Sort до O(n2).