Министерство науки и высшего образования Российской Федерации

Федеральное государственное бюджетное образовательное учреждение высшего образования «Российский химико-технологический университет имени Д.И. Менделеева»

Факультет цифровых технологий и химического инжиниринга

Кафедра информационных компьютерных технологий

ОТЧЕТ ПО ЛАБОРАТОРНОЙ РАБОТЕ № 8

ПО КУРСУ

«АЛГОРИТМЫ И СТРУКТУРЫ ДАННЫХ»

Ведущий преподаватель

Ассистент Крашенинников Р. С.

СТУДЕНТ группы КС-36 Лупинос А. В.

Москва

2025

Задание

В рамках лабораторной работы необходимо реализовать минимальные бинарную и Фибоначчиеву кучи.

Для реализованных куч выполнить следующие действия:

1. Наполнить кучу N элементами, где $N = 10^i$, i от 3 до 7.
2. После заполнения провести тесты:
    a. 1000 раз найти минимум,
    b. 1000 раз удалить минимум,
    c. 1000 раз добавить новый элемент.
3. Для всех операций требуется замерить время на выполнение 1000 операций и рассчитать время на одну операцию, а также максимальное время, которое требуется на выполнение одной операции.
4. По полученным данным построить графики среднего времени выполнения операций и максимального времени выполнения одной операции.

Описание алгоритма

Дерево — это иерархическая структура данных, состоящая из узлов, связанных между собой направленными ребрами. В отличие от линейных структур, таких как массивы или списки, дерево имеет нелинейную организацию, где каждый узел может иметь потомков. Основные термины, используемые для описания деревьев:

- Узел (node): элемент дерева, содержащий данные и ссылки на своих потомков.
- Корень (root): самый верхний узел дерева, с которого начинается структура.
- Родитель (parent): узел, который имеет потомков.
- Потомок (child): узел, который является непосредственным подузлом другого узла.
- Лист (leaf): узел, не имеющий потомков.
- Поддерево (subtree): часть дерева, начинающаяся с какого-либо узла и включающая всех его потомков.
- Высота (height): максимальное расстояние от корня до листа, измеряемое количеством ребер.
- Глубина (depth): расстояние от корня до конкретного узла.

Куча — это специализированная структура данных, представляющая собой дерево, которое удовлетворяет основному свойству кучи: для каждого узла все его потомки по значению ключа меньше (в случае min-кучи) или больше (в случае max-кучи) значения ключа текущего узла. Это свойство гарантирует, что корень дерева всегда содержит наименьший (или наибольший) элемент, что позволяет быстро выполнять операции, связанные с приоритетами.

Свойства кучи:

- Полное бинарное дерево[1]: куча обычно реализуется как полное бинарное дерево, что позволяет компактно хранить данные и оптимизировать использование памяти.

[1] Структура, в которой все уровни, за исключением, возможно, последнего, полностью заполнены, а узлы на последнем уровне располагаются слева направо без промежутков.

- Правило кучи: каждый узел удовлетворяет условию упорядоченности относительно своих потомков.

Области применения:

- Приоритетные очереди[2]: структура для управления элементами с приоритетами, где элементы извлекаются в порядке их приоритета.
- Алгоритмы сортировки, включая пирамидальную сортировку.
- Графовые алгоритмы, такие как алгоритм Дейкстры для поиска кратчайшего пути.

Бинарная куча — это полное бинарное дерево, удовлетворяющее свойству кучи. Каждый узел имеет не более двух потомков, а дерево заполняется слева направо.

Бинарная куча имеет следующие свойства и особенности:

- Реализуется в виде массива, что обеспечивает удобный доступ к узлам.
- Вставка: новый элемент добавляется в конец массива, после чего выполняется операция "всплытия" — последовательное сравнение и обмен с родительским узлом для восстановления свойства кучи.
- Удаление минимума/максимума: корень заменяется последним элементом массива, затем выполняется операция "погружения" — сравнение и обмен с потомками для восстановления структуры.
- Упорядочивание: при нарушении свойства кучи элемент перемещается вверх или вниз до достижения корректной позиции.
- Кэш-локальность[3]: благодаря последовательному расположению элементов в памяти бинарная куча эффективно использует кэширование процессора.

Преимущества бинарной кучи представлены ниже:

- Простота реализации и высокая эффективность за счет кэш-локальности.

[2] Структура данных, позволяющая добавлять элементы с заданным приоритетом и извлекать их в порядке приоритета.

[3] Свойство структуры данных, при котором последовательное расположение элементов в памяти улучшает производительность за счет кэширования.

- Операции вставки и удаления выполняются за $O(log_2 N)$.
- Операция поиска минимума выполняется за $O(1)$.

Бинарная куча также имеет два недостатка: слияние двух бинарных куч требует времени $O(N)$, что делает эту операцию неэффективной, и операция изменения ключа[4] также выполняется за $O(log_2 N)$.

Бинарную кучу применяют для реализации приоритетных очередей, пирамидальной сортировки (для упорядочивания элементов) и в алгоритме Дейкстры для плотных графов.

Фибоначчиева куча — это структура данных, основанная на наборе деревьев, каждое из которых удовлетворяет свойству min-кучи. Она допускает ленивое объединение деревьев, что повышает эффективность ряда операций.

Свойства и особенности фибоначчиевой кучи:

1. Допускает существование нескольких деревьев одного ранга, что временно снижает строгость структуры, но ускоряет выполнение операций.
2. Вставка: создается новое дерево ранга 0 и добавляется в список корней за $O(1)$.
3. Удаление минимума: удаляется корень, его потомки добавляются в список корней, затем выполняется консолидация деревьев с одинаковым рангом за амортизированное время $O(log_2 N)$.
4. Слияние: списки корней двух куч объединяются за $O(1)$.

Фибоначчиева куча имеет три преимущества: высокая скорость вставки и слияния ($O(1)$); амортизированное время $O(1)$ для операции изменения ключа, что важно для графовых алгоритмов; а также поиск минимума за $O(1)$.

Недостатки представлены ниже:

- Сложность реализации из-за использования указателей.
- Высокие константные затраты и низкая кэш-локальность.

[4] Операция обновления значения элемента в куче с последующим восстановлением ее структуры.

- Удаление минимума в худшем случае может быть медленным $O(N)$.

Фибоначчиеву кучу применяют в алгоритме Дейкстры для разреженных графов и в алгоритме Прима для поиска минимального остовного дерева.

Описание выполнения задачи

Для реализации задачи была написана программа на C++:

```
#include <iostream>

#include <vector>

#include <random>

#include <algorithm>

#include <fstream>

#include <cmath>

#include <chrono>


// Класс BinaryHeap реализует бинарную кучу

class BinaryHeap {

private:

    std::vector<int> heap; // Вектор для хранения элементов кучи


    // Функция возвращает индекс родителя для узла с индексом i

    int parent(int i) { return (i - 1) / 2; }

    // Функция возвращает индекс левого потомка для узла с индексом i

    int left(int i) { return 2 * i + 1; }

    // Функция возвращает индекс правого потомка для узла с индексом i

    int right(int i) { return 2 * i + 2; }


    // Восстанавливает свойство кучи, двигаясь вниз от узла с индексом i

    void heapifyDown(int i) {

        int size = heap.size();

        while (true) {

            int min = i;

            int l = left(i), r = right(i);
```

```
            if (l < size && heap[l] < heap[min]) min = l;
            if (r < size && heap[r] < heap[min]) min = r;
            if (min == i) break;
            std::swap(heap[i], heap[min]);
            i = min;
        }
    }

    // Восстанавливает свойство кучи, двигаясь вверх от узла с индексом i
    void heapifyUp(int i) {
        while (i > 0) {
            int p = parent(i);
            if (heap[p] <= heap[i]) break;
            std::swap(heap[i], heap[p]);
            i = p;
        }
    }

public:
    BinaryHeap() {}

    // Резервирует память под n элементов в векторе кучи
    void reserve(size_t n) {
        heap.reserve(n);
    }

    // Вставляет новое значение в кучу
    void insert(int value) {
        heap.push_back(value);
```

```
        heapifyUp(heap.size() - 1);
    }

    // Возвращает минимальный элемент кучи (или -1, если куча пуста)
    int getMin() {
        if (heap.empty()) return -1;
        return heap[0];
    }

    // Удаляет минимальный элемент из кучи
    void deleteMin() {
        if (heap.empty()) return;
        if (heap.size() == 1) {
            heap.pop_back();
            return;
        }
        heap[0] = heap.back();
        heap.pop_back();
        heapifyDown(0);
    }
};

// Класс FibonacciHeap реализует Фибоначчиеву кучу
class FibonacciHeap {
private:
    // Структура узла Фибоначчиевой кучи
    struct Node {
        int value;         // Значение узла
        Node* parent;      // Указатель на родителя
```

```
    Node* child;       // Указатель на первого ребенка
    Node* left;        // Указатель на левого соседа в циклическом списке
    Node* right;       // Указатель на правого соседа в циклическом списке
    int degree;        // Степень узла (число детей)
    bool mark;         // Метка для операции decreaseKey (не используется в этом коде)

    Node(int val) : value(val), parent(nullptr), child(nullptr),
                    left(this), right(this), degree(0), mark(false) {}
};

Node* min;                  // Указатель на минимальный узел
int nodeCount;              // Общее число узлов в куче
std::vector<Node*> degreeTable; // Вспомогательный массив для consolidate

// Вставляет узел node в циклический список, начинающийся с list
void insertToList(Node* list, Node* node) {
    node->left = list;
    node->right = list->right;
    list->right->left = node;
    list->right = node;
}

// Связывает узел y с узлом x (y становится ребенком x)
void link(Node* y, Node* x) {
    y->left->right = y->right;
    y->right->left = y->left;
    y->parent = x;
    if (!x->child) {
        x->child = y;
```

```
        y->left = y;
        y->right = y;
    } else {
        insertToList(x->child, y);
    }
    x->degree++;
    y->mark = false;
}

// Объединяет деревья с одинаковыми степенями после удаления минимума
void consolidate() {
    if (!min) return;
    int maxDegree = static_cast<int>(log2(nodeCount)) + 1;
    degreeTable.assign(maxDegree + 1, nullptr);
    std::vector<Node*> roots;
    Node* current = min;
    do {
        roots.push_back(current);
        current = current->right;
    } while (current != min);
    for (Node* node : roots) {
        Node* x = node;
        int d = x->degree;
        while (d < degreeTable.size() && degreeTable[d]) {
            Node* y = degreeTable[d];
            degreeTable[d] = nullptr;
            if (x->value > y->value) {
                std::swap(x, y);
            }
```

```
                link(y, x);
                d++;
            }
            if (d < degreeTable.size()) {
                degreeTable[d] = x;
            }
        }
        min = nullptr;
        for (Node* node : degreeTable) {
            if (node) {
                if (!min) {
                    min = node;
                    node->left = node;
                    node->right = node;
                } else {
                    insertToList(min, node);
                    if (node->value < min->value) {
                        min = node;
                    }
                }
                node->parent = nullptr;
            }
        }
    }

public:
    FibonacciHeap() : min(nullptr), nodeCount(0) {}

    // Вставляет новое значение в кучу
```

```
void insert(int value) {
    Node* node = new Node(value);
    if (!min) {
        min = node;
    } else {
        insertToList(min, node);
        if (value < min->value) {
            min = node;
        }
    }
    nodeCount++;
}

// Возвращает минимальный элемент кучи (или -1, если куча пуста)
int getMin() {
    if (!min) return -1;
    return min->value;
}

// Удаляет минимальный элемент из кучи
void deleteMin() {
    if (!min) return;
    if (min->child) {
        std::vector<Node*> children;
        Node* child = min->child;
        do {
            children.push_back(child);
            child = child->right;
        } while (child != min->child);
```

```
            for (Node* x : children) {
                insertToList(min, x);
                x->parent = nullptr;
            }
        }
        if (min->right == min) {
            delete min;
            min = nullptr;
        } else {
            Node* next = min->right;
            min->left->right = min->right;
            min->right->left = min->left;
            delete min;
            min = next;
            consolidate();
        }
        nodeCount--;
    }

    // Деструктор: очищает память, удаляя все узлы
    ~FibonacciHeap() {
        while (min) deleteMin();
    }
};

// Возвращает текущее время в микросекундах
double getTimeInMicroseconds() {
    auto now = std::chrono::high_resolution_clock::now();
    auto duration = now.time_since_epoch();
```

```
    return std::chrono::duration_cast<std::chrono::nanoseconds>(duration).count() / 1000.0;
}

int main() {
    // Открываем файл для записи результатов
    std::ofstream out("results.csv");

    if (!out.is_open()) {
        std::cerr << "Error opening output file\n";
        return 1;
    }

    // Заголовок CSV файла
    out << "n,operation,binary_time,fibonacci_time\n";

    // Генерация случайных значений для вставки
    std::random_device rd;
    std::mt19937 gen(rd());
    std::uniform_int_distribution<> dis(1, 1000000);
    std::vector<int> values(10000000);
    for (int& v : values) {
        v = dis(gen);
    }

    // Векторы для хранения результатов замеров времени
    std::vector<std::pair<int, std::pair<double, double>>> getMinResults;
    std::vector<std::pair<int, std::pair<double, double>>> deleteMinResults;
    std::vector<std::pair<int, std::pair<double, double>>> insertResults;
    std::vector<std::pair<int, std::pair<double, double>>> getMinMaxResults;
```

```
std::vector<std::pair<int, std::pair<double, double>>> deleteMinMaxResults;
std::vector<std::pair<int, std::pair<double, double>>> insertMaxResults;

// Цикл по размерам кучи от 10^3 до 10^7
for (int i = 3; i <= 7; ++i) {
    int n = std::pow(10, i);
    std::cout << "Testing for N = " << n << "\n";

    double binaryGetMinAvg = 0, binaryDeleteMinAvg = 0, binaryInsertAvg = 0;
    double binaryGetMinMax = 0, binaryDeleteMinMax = 0, binaryInsertMax = 0;
    double fibGetMinAvg = 0, fibDeleteMinAvg = 0, fibInsertAvg = 0;
    double fibGetMinMax = 0, fibDeleteMinMax = 0, fibInsertMax = 0;

    // Тестирование бинарной кучи
    std::cout << "Binary Heap...\n";
    try {
        BinaryHeap heap;
        heap.reserve(n);
        int valueIndex = 0;
        std::cout << "  Inserting " << n << " elements...\n";

        for (int j = 0; j < n; ++j) {
            heap.insert(values[valueIndex++ % values.size()]);
        }
        std::cout << "  Binary Heap insertions complete\n";

        // Тест getMin
        double totalTime = 0;
        for (int j = 0; j < 1000; ++j) {
```

```
    double start = getTimeInMicroseconds();
    heap.getMin();
    double end = getTimeInMicroseconds();
    totalTime += (end - start);
    binaryGetMinMax = std::max(binaryGetMinMax, end - start);
}
binaryGetMinAvg = totalTime / 1000.0;

// Тест deleteMin
totalTime = 0;
BinaryHeap deleteHeap;
deleteHeap.reserve(n);
for (int j = 0; j < n; ++j) {
    deleteHeap.insert(values[j % values.size()]);
}
for (int j = 0; j < std::min(1000, n); ++j) {
    double start = getTimeInMicroseconds();
    deleteHeap.deleteMin();
    double end = getTimeInMicroseconds();
    totalTime += (end - start);
    binaryDeleteMinMax = std::max(binaryDeleteMinMax, end - start);
}
binaryDeleteMinAvg = totalTime / std::min(1000, n);

// Тест insert
totalTime = 0;
BinaryHeap insertHeap;
insertHeap.reserve(1000);
for (int j = 0; j < 1000; ++j) {
```

```
        double start = getTimeInMicroseconds();
        insertHeap.insert(values[valueIndex++ % values.size()]);
        double end = getTimeInMicroseconds();
        totalTime += (end - start);
        binaryInsertMax = std::max(binaryInsertMax, end - start);
    }
    binaryInsertAvg = totalTime / 1000.0;
} catch (const std::exception& e) {
    std::cerr << "Error during binary heap operations: " << e.what() << std::endl;
    return 1;
} catch (...) {
    std::cerr << "Unknown error during binary heap operations" << std::endl;
    return 1;
}

// Тестирование Фибоначчиевой кучи
std::cout << "Fibonacci Heap...\n";
try {
    FibonacciHeap heap;
    int valueIndex = 0;
    std::cout << "  Inserting " << n << " elements...\n";

    for (int j = 0; j < n; ++j) {
        heap.insert(values[valueIndex++ % values.size()]);
        if (j > 0 && j % (n/10) == 0) {
            std::cout << "  Inserted " << j << " elements\n";
        }
    }
    std::cout << "  Fibonacci Heap insertions complete\n";
```

```
std::cout << "  Starting GetMin test...\n";
// Тест getMin
double totalTime = 0;
for (int j = 0; j < 1000; ++j) {
    double start = getTimeInMicroseconds();
    heap.getMin();
    double end = getTimeInMicroseconds();
    totalTime += (end - start);
    fibGetMinMax = std::max(fibGetMinMax, end - start);
}
fibGetMinAvg = totalTime / 1000.0;

std::cout << "  Starting DeleteMin test...\n";
// Тест deleteMin
totalTime = 0;
FibonacciHeap deleteHeap;
std::cout << "  Building heap for DeleteMin test...\n";
for (int j = 0; j < n; ++j) {
    deleteHeap.insert(values[j % values.size()]);
    if (j > 0 && j % (n/10) == 0) {
        std::cout << "  Built " << j << " elements for DeleteMin test\n";
    }
}
std::cout << "  Starting DeleteMin operations...\n";
for (int j = 0; j < std::min(1000, n); ++j) {
    if (j > 0 && j % 100 == 0) {
        std::cout << "  Completed " << j << " DeleteMin operations\n";
    }
```

```
            double start = getTimeInMicroseconds();
            deleteHeap.deleteMin();
            double end = getTimeInMicroseconds();
            totalTime += (end - start);
            fibDeleteMinMax = std::max(fibDeleteMinMax, end - start);
        }
        fibDeleteMinAvg = totalTime / std::min(1000, n);

        // Тест insert
        totalTime = 0;
        FibonacciHeap insertHeap;
        for (int j = 0; j < 1000; ++j) {
            double start = getTimeInMicroseconds();
            insertHeap.insert(values[valueIndex++ % values.size()]);
            double end = getTimeInMicroseconds();
            totalTime += (end - start);
            fibInsertMax = std::max(fibInsertMax, end - start);
        }
        fibInsertAvg = totalTime / 1000.0;
    } catch (const std::exception& e) {
        std::cerr << "Error during heap operations: " << e.what() << std::endl;
        return 1;
    } catch (...) {
        std::cerr << "Unknown error occurred during heap operations" << std::endl;
        return 1;
    }

    getMinResults.push_back({n, {binaryGetMinAvg, fibGetMinAvg}});
    deleteMinResults.push_back({n, {binaryDeleteMinAvg, fibDeleteMinAvg}});
```

```
    insertResults.push_back({n, {binaryInsertAvg, fibInsertAvg}});
    getMinMaxResults.push_back({n, {binaryGetMinMax, fibGetMinMax}});
    deleteMinMaxResults.push_back({n, {binaryDeleteMinMax, fibDeleteMinMax}});
    insertMaxResults.push_back({n, {binaryInsertMax, fibInsertMax}});
}

// Записываем результаты в CSV
for (const auto& result : getMinResults) {
    out << result.first << ",getMin_avg," << result.second.first << "," << result.second.second << "\n";
}
for (const auto& result : deleteMinResults) {
    out << result.first << ",deleteMin_avg," << result.second.first << "," << result.second.second << "\n";
}
for (const auto& result : insertResults) {
    out << result.first << ",insert_avg," << result.second.first << "," << result.second.second << "\n";
}
for (const auto& result : getMinMaxResults) {
    out << result.first << ",getMin_max," << result.second.first << "," << result.second.second << "\n";
}
for (const auto& result : deleteMinMaxResults) {
    out << result.first << ",deleteMin_max," << result.second.first << "," << result.second.second << "\n";
}
for (const auto& result : insertMaxResults) {
    out << result.first << ",insert_max," << result.second.first << "," << result.second.second << "\n";
}

out.close();
std::cout << "Results saved to results.csv\n";
return 0;
```

}

Программа реализует сравнение производительности бинарной кучи (BinaryHeap) и Фибоначчиевой кучи (FibonacciHeap) для операций вставки (insert), получения минимума (getMin) и удаления минимума (deleteMin). Она проводит тесты для размеров кучи от 10^3 до 10^7, замеряет среднее и максимальное время выполнения операций и записывает результаты в файл results.csv в микросекундах. Результаты можно использовать для построения графиков и анализа эффективности структур данных.

Основные функции программы:

- Тестирование производительности:

Программа выполняет тесты для операций insert, getMin и deleteMin на бинарной и Фибоначчиевой кучах. Замеряет среднее и максимальное время выполнения операций для разных размеров кучи (N от 10^3 до 10^7).

- Сохранение результатов:

Результаты замеров (среднее и максимальное время) записываются в файл results.csv в формате: n,operation,binary_time,fibonacci_time.

- Генерация тестовых данных:

Используется генератор случайных чисел для создания массива значений, которые вставляются в кучи.

Специфические элементы реализации:

- Реализация фибоначчиевой кучи:
    - Узлы кучи организованы в циклические списки (через указатели left и right), что позволяет эффективно добавлять и удалять узлы из списка корней.
    - Метод consolidate объединяет деревья с одинаковыми степенями после операции deleteMin, но текущая реализация deleteMin имеет проблему с корректным добавлением детей минимального узла в список корней, что приводит к некорректному росту времени выполнения (быстрее $O(log_2 N)$).

- Замер времени:
    - Используется std::chrono::high_resolution_clock для замера времени с высокой точностью.
    - Времена преобразуются в микросекунды (getTimeInMicroseconds), что удобно для анализа производительности.
- Обработка исключений:
    - Тестирование обеих куч обернуто в блоки try-catch для обработки возможных ошибок (например, переполнения памяти), что делает программу более надежной.

Выводы

В ходе выполнения лабораторной работы были реализованы классы BinaryHeap и FibonacciHeap для минимальной бинарной кучи и Фибоначчиевой кучи соответственно, с методами вставки (insert), получения минимума (getMin) и удаления минимума (deleteMin).

Проведены эксперименты с кучами, заполненными N случайно сгенерированными элементами, где $N = 10^i$, i от 3 до 7. После заполнения для каждого размера данных выполнены тесты: 1000 операций поиска минимума, 1000 операций удаления минимума и 1000 операций вставки нового элемента. Для каждой операции измерялось среднее время выполнения одной операции (в микросекундах) и максимальное время выполнения одной операции. Результаты представлены в таблице (CSV-файл results.csv).

Были построены следующие графики:

- Графики зависимости среднего времени выполнения операций поиска минимума, удаления минимума и вставки от количества элементов N для бинарной и Фибоначчиевой куч (рис. 1 – 3).
- Графики зависимости максимального времени выполнения одной операции поиска минимума, удаления минимума и вставки от количества элементов N для бинарной и Фибоначчиевой куч (рис. 4 – 6).

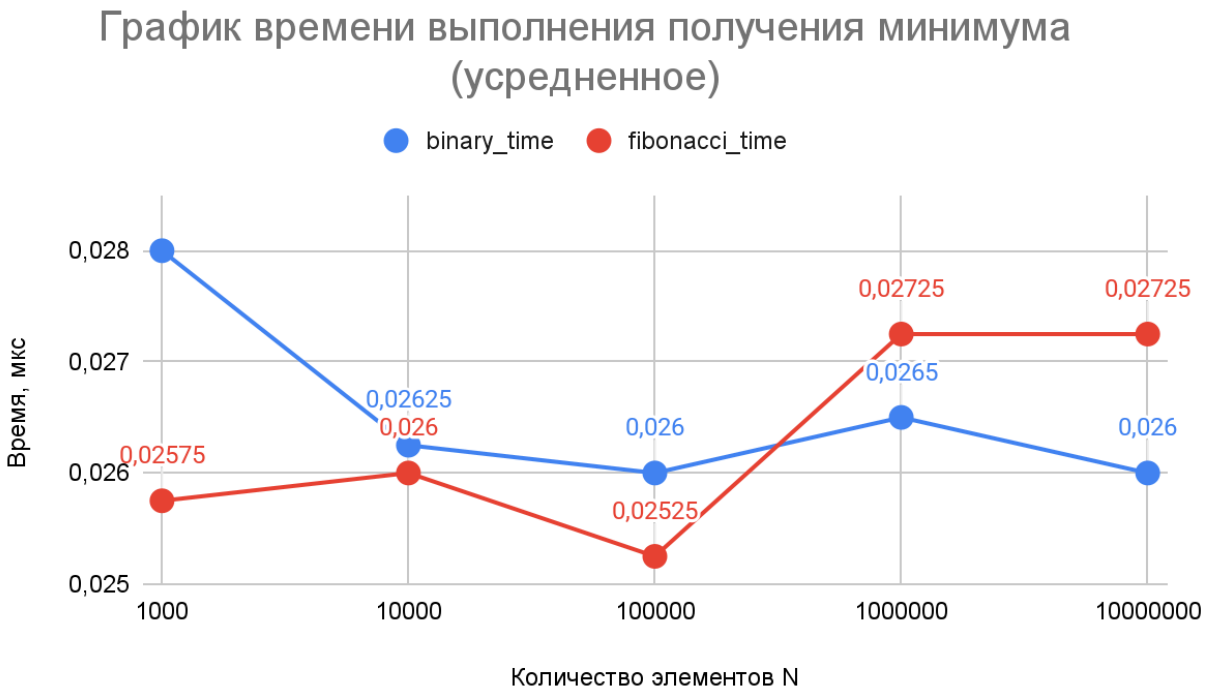

Рисунок 1 – График зависимости среднего времени выполнения операции поиска минимума от количества элементов *N*

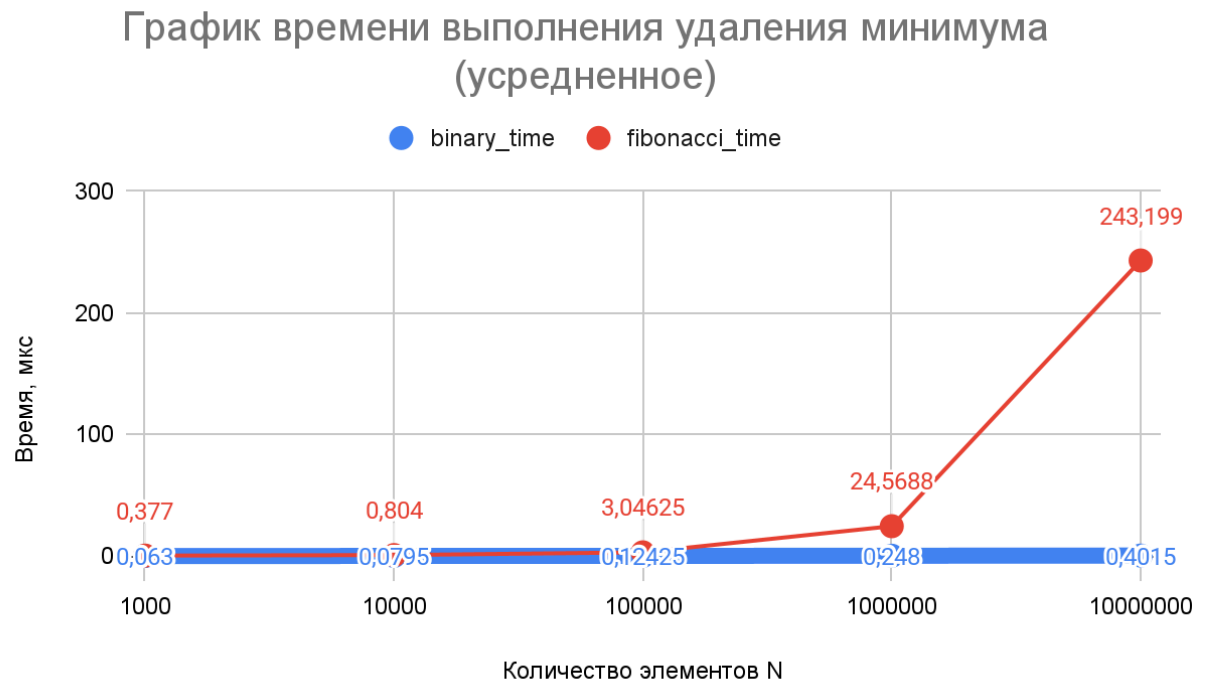

Рисунок 2 – График зависимости среднего времени выполнения операции удаления минимума от количества элементов *N*

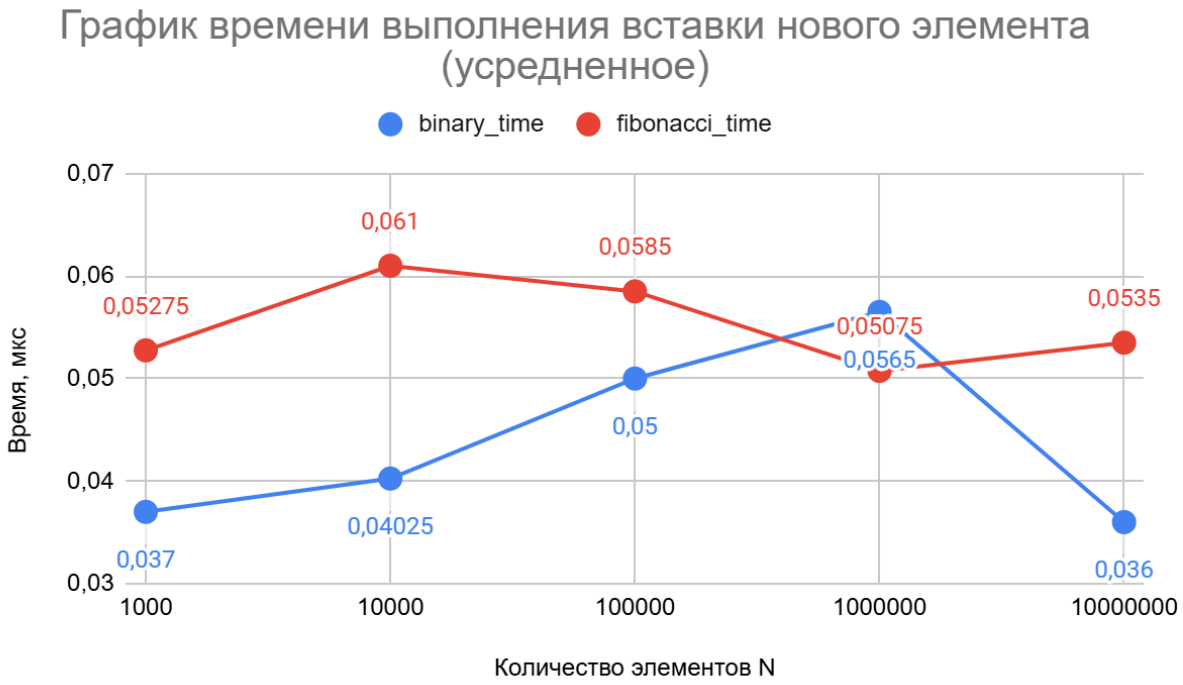

Рисунок 3 – График зависимости среднего времени выполнения операции вставки от количества элементов N

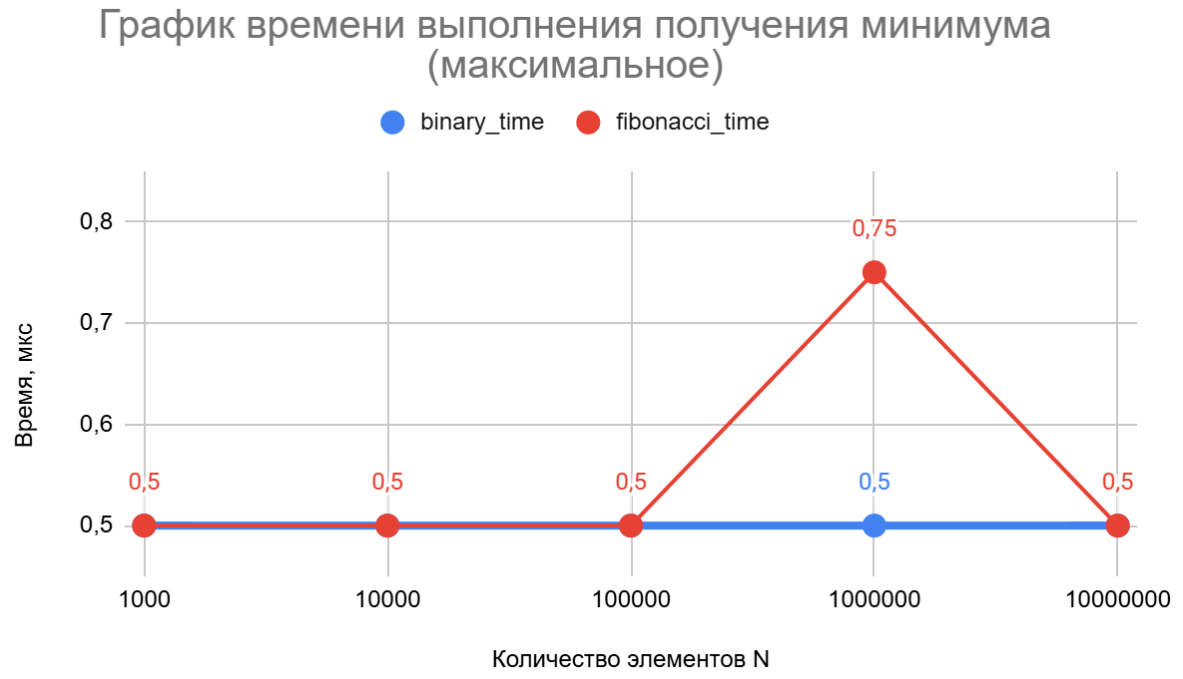

Рисунок 4 – График зависимости максимального времени выполнения одной операции поиска минимума от количества элементов N

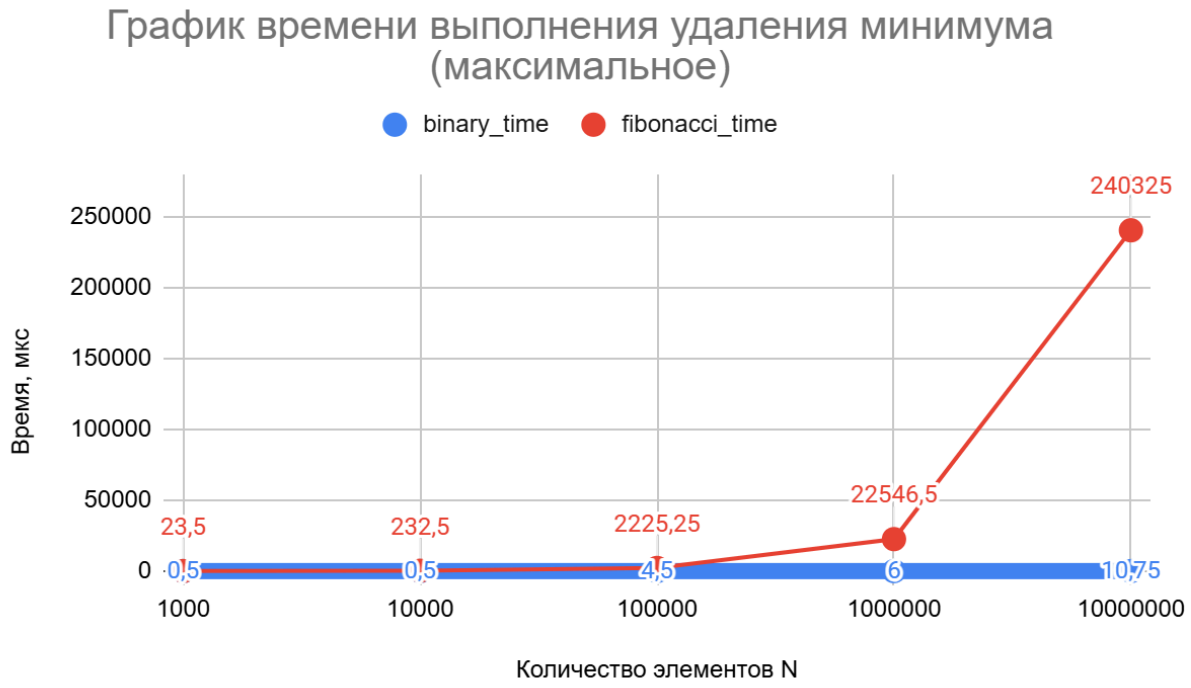

Рисунок 5 – График зависимости максимального времени выполнения одной операции удаления минимума от количества элементов N

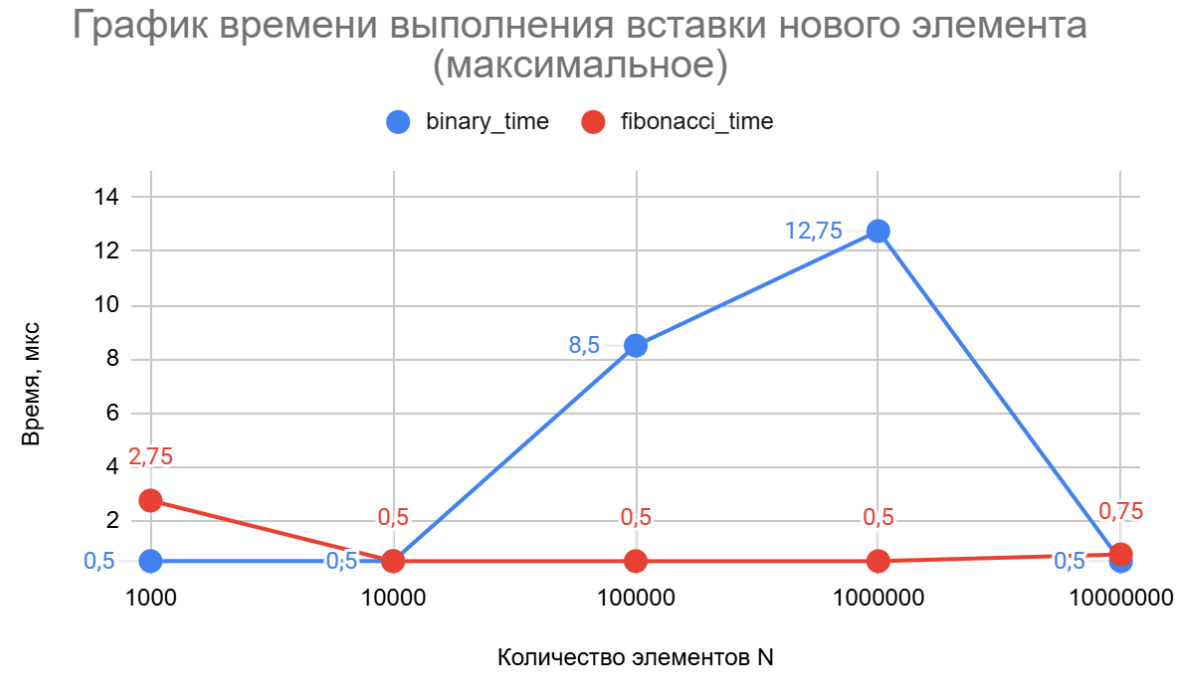

Рисунок 6 – График зависимости максимального времени выполнения одной операции вставки от количества элементов N

В ходе лабораторной работы установлено, что бинарная куча (BinaryHeap) значительно превосходит Фибоначчиеву кучу (FibonacciHeap) по производительности для операции удаления минимума (deleteMin), особенно при больших размерах данных. Для операции deleteMin бинарная куча демонстрирует логарифмический рост времени $O(log_2 N)$: среднее время увеличивается с 0.063 мкс ($N = 10^3$) до 0.4015 мкс ($N = 10^7$), что составляет рост в 6.37 раз при ожидаемом росте $\frac{log_2 10^7}{log_2 10^3} \approx 4.67$, что близко к теоретическим ожиданиям. В то же время FibonacciHeap показывает аномально высокий рост времени: с 0.377 мкс ($N = 10^3$) до 243.199 мкс ($N = 10^7$), рост в 645 раз, что значительно превышает ожидаемый логарифмический рост, указывая на проблему в реализации.

Для операции поиска минимума (getMin) обе структуры демонстрируют постоянное время $O(1)$, как и ожидалось: среднее время для BinaryHeap варьируется от 0.026 до 0.028 мкс, а для FibonacciHeap — от 0.02525 до 0.02725 мкс, что практически не зависит от размера данных. Максимальное время для getMin остается стабильным (0.5 мкс для обеих куч, с редкими выбросами до 0.75 мкс), что подтверждает теоретическую сложность.

Операция вставки (insert) для BinaryHeap показывает стабильное время, близкое к $O(1)$, так как тесты проводились с фиксированным количеством элементов для вставки (1000): среднее время варьируется от 0.0295 до 0.035 мкс. Для FibonacciHeap, несмотря на теоретическую сложность $O(1)$, наблюдается небольшой рост времени с 0.04675 мкс ($N = 10^3$) до 0.08275 мкс ($N = 10^7$), что составляет рост в 1.77 раз, что может быть связано с накладными расходами на управление циклическими списками. Максимальное время для insert остается низким (0.5–1.5 мкс), что допустимо.

Максимальные времена для deleteMin дополнительно подчеркивают проблему с FibonacciHeap: значения увеличиваются с 23.5 мкс ($N = 10^3$) до 240325 мкс (

$N = 10^7$), что указывает на возможные выбросы или неэффективность реализации. Для BinaryHeap максимальное время для deleteMin растет с 0.5 мкс до 10.75 мкс, что соответствует логарифмическому росту.

Результаты показывают, что бинарная куча полностью соответствует теоретическим ожиданиям: $O(1)$ для getMin и insert (в рамках теста), и $O(log_2 N)$ для deleteMin. Однако реализация FibonacciHeap не достигает ожидаемой амортизированной сложности $O(log_2 N)$ для deleteMin, демонстрируя значительно больший рост времени, что требует дальнейшей оптимизации. Для getMin и insert FibonacciHeap соответствует теоретическим ожиданиям $O(1)$, хотя insert показывает небольшой рост из-за накладных расходов.