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 № 8

Выполнил студент группы КС-36 Сары Кристина Ивановна

Ссылка на репозиторий: https://github.com/MUCTR-IKT-CPP/SaryKI\_36\_ALG.git

Приняли: Пысин Максим Дмитриевич

Краснов Дмитрий Олегович

Лобанов Алексей Владимирович

Крашенинников Роман Сергеевич

Дата сдачи: 24.02.2025

Оглавление

[Описание задачи. 3](#_Toc63548272)

[Описание метода/модели. 3](#_Toc63548273)

[Выполнение задачи. 4](#_Toc63548274)

[Заключение. 8](#_Toc63548275)

# Описание задачи.

**Задание:**

В рамках лабораторной работы необходимо реализовать бинарную кучу(мин), а так же

1. Фибоначиеву кучу

Для реализованных куч выполнить следующие действия:

1. Наполнить кучу N кол-ва элементов (где N = 10 ^ i, i от 3 до 7).
2. После заполнения кучи необходимо провести следующие тесты:
   1. 1000 раз найти минимум/максимум
   2. 1000 раз удалить минимум/максимум
   3. 1000 раз добавить новый элемент в кучу  
      Для всех операция требуется замерить время на выполнения всей 1000 операций и рассчитать время на одну операцию, а так же запомнить максимальное время которое требуется на выполнение одной операции если язык позволяет его зафиксировать, если не позволяет воспользоваться хитростью и расчитывать усредненное время на каждые 10,25,50,100 операций, и выбирать максимальное из полученных результатов, что бы поймать момент деградации структуры и ее перестройку.
3. По полученным в задании 2 данным построить графики времени выполнения операций для усреднения по 1000 операций, и для максимального времени на 1 операцию.

# Описание метода/модели.

**Теоретическая часть**

Куча (англ. heap) — это специализированная структура данных, представляющая собой двоичное дерево, для которого выполнено основное свойство кучи (условие упорядоченности). В данной работе рассматриваются две разновидности куч: бинарная куча (минимальная) и фибоначчиева куча.

**Бинарная куча (минимальная)**

**Определение и свойства**

Бинарная куча — это полное двоичное дерево (все уровни заполнены, кроме, возможно, последнего, который заполняется слева направо), в котором выполняется основное свойство кучи: ключ в каждом узле меньше или равен ключам в его потомках (для минимальной кучи).

**Основные операции и их сложность**

1. **Вставка элемента (insert)**:
   * Новый элемент добавляется в конец кучи (последняя позиция в массиве)
   * Затем выполняется операция "просеивания вверх" (sift-up) для восстановления свойства кучи
   * Сложность: O(log n) в худшем случае
2. **Извлечение минимума (extract-min)**:
   * Минимальный элемент находится в корне (первый элемент массива)
   * Последний элемент перемещается на место корня
   * Выполняется операция "просеивания вниз" (sift-down)
   * Сложность: O(log n)
3. **Поиск минимума (find-min)**:
   * Просто возвращается корневой элемент
   * Сложность: O(1)
4. **Уменьшение ключа (decrease-key)**:
   * Изменяется значение элемента
   * Выполняется просеивание вверх
   * Сложность: O(log n)

**Реализация**

Бинарная куча обычно реализуется с помощью массива, где:

* Индекс родителя для элемента i: ((i-1)/2)
* Индекс левого потомка: 2i + 1
* Индекс правого потомка: 2i + 2

**Преимущества и недостатки**

**Преимущества**:

* Простота реализации
* Эффективное использование памяти (хранится в массиве)
* Хорошая производительность для небольших объемов данных

**Недостатки**:

* Слияние двух куч (merge) требует O(n) операций
* Некоторые операции (например, decrease-key) менее эффективны, чем в других структурах

**Фибоначчиева куча**

**Определение и свойства**

Фибоначчиева куча — это набор деревьев, удовлетворяющих свойству мин-кучи (ключ в каждом узле меньше или равен ключам его потомков). В отличие от бинарной кучи, фибоначчиева куча имеет более сложную структуру, но обеспечивает лучшие амортизированные временные границы для некоторых операций.

**Уплотнение** — это процесс объединения деревьев одинаковой степени после удаления минимального элемента для поддержания эффективной структуры кучи. Это ключевая операция, обеспечивающая амортизированную сложность O(log n) для extract-min.

**Как работает уплотнение:**

1. Создается таблица (массив) для отслеживания деревьев по их степеням
2. Все корневые узлы обрабатываются последовательно
3. Если встречается два дерева одинаковой степени, они объединяются в одно
4. Процесс повторяется, пока все деревья в корневом списке не будут иметь уникальные степени

**Основные операции и их сложность**

1. **Вставка элемента (insert)**:
   * Новый элемент добавляется как отдельное дерево в список корней
   * Сложность: O(1) амортизированная
2. **Извлечение минимума (extract-min)**:
   * Удаляется минимальный элемент (корень одного из деревьев)
   * Его дети добавляются в список корней
   * Выполняется операция "уплотнения" (consolidate) для объединения деревьев одинакового ранга
   * Сложность: O(log n) амортизированная
3. **Поиск минимума (find-min)**:
   * Минимум хранится в указателе (поддерживается при всех операциях)
   * Сложность: O(1)
4. **Уменьшение ключа (decrease-key)**:
   * Если свойство кучи нарушается, узел отрезается от родителя и добавляется в список корней
   * Может вызвать каскадное разрезание (cascading cut)
   * Сложность: O(1) амортизированная
5. **Слияние куч (merge)**:
   * Просто объединяются списки корней двух куч
   * Сложность: O(1)

**Особенности реализации**

Фибоначчиева куча состоит из нескольких деревьев, которые удовлетворяют свойству мин-кучи. Каждый узел содержит:

* Указатели на родителя, одного из детей и на братьев (обычно реализуется как двусвязный циклический список)
* Степень (количество детей)
* Флаг mark (используется при операции decrease-key)

**Преимущества и недостатки**

**Преимущества**:

* Лучшие амортизированные временные границы для многих операций
* Эффективные операции merge и decrease-key
* Теоретически оптимальна для алгоритмов, требующих частых операций decrease-key (например, алгоритм Дейкстры)

**Недостатки**:

* Сложность реализации
* Большие константные множители в оценке сложности
* На практике может работать медленнее бинарной кучи для небольших объемов данных

**Сравнительный анализ**

|  |  |  |
| --- | --- | --- |
| **Операция** | **Бинарная куча** | **Фибоначчиева куча** |
| insert | O(log n) | O(1) |
| extract-min | O(log n) | O(log n) |
| find-min | O(1) | O(1) |
| decrease-key | O(log n) | O(1) |
| merge | O(n) | O(1) |

# Выполнение задачи.

Программа реализована на языке C++.

**Код:**

**//лаб8**

**#include <iostream>**

**#include <vector>**

**#include <queue>**

**#include <chrono>**

**#include <random>**

**#include <algorithm>**

**#include <fstream>**

**#include <iomanip>**

**// бинарная мин-кучу, где минимальный элемент всегда находится в корне**

**class BinaryMinHeap {**

**private:**

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

**// Функция возвращает индекс родителя для заданного узла с индексом i**

**int parent(int i) { return (i - 1) / 2; } // Родитель узла i находится по формуле (i-1)/2**

**int left(int i) { return 2 \* i + 1; } // Левый потомок узла i находится по формуле 2\*i + 1**

**int right(int i) { return 2 \* i + 2; } // Правый потомок узла i находится по формуле 2\*i + 2**

**void siftUp(int i) { //поднимает элемент с индексом i к правильной позиции в куче**

**while (i > 0 && heap[parent(i)] > heap[i]) { // // Продолжаем, пока не достигли корня (i > 0)**

**// и пока родитель больше текущего элемента**

**// Меняем местами текущий элемент и его родителя**

**std::swap(heap[i], heap[parent(i)]);**

**// Переходим к родителю**

**i = parent(i);**

**}**

**}**

**void siftDown(int i) { // опускает элемент с индексом i к правильной позиции в куче**

**int minIndex = i; // Изначально предполагаем, что минимальный элемент - это текущий узел**

**int l = left(i), r = right(i); // Получаем индексы левого и правого потомков**

**// Проверяем, есть ли левый потомок и меньше ли он текущего минимального элемента**

**if (l < heap.size() && heap[l] < heap[minIndex])**

**minIndex = l; // Если да, обновляем индекс минимального элемента**

**// Проверяем, есть ли правый потомок и меньше ли он текущего минимального элемента**

**if (r < heap.size() && heap[r] < heap[minIndex])**

**minIndex = r;**

**// Если минимальный элемент не текущий узел, меняем их местами и продолжаем просеивание вниз**

**if (i != minIndex) {**

**std::swap(heap[i], heap[minIndex]);**

**siftDown(minIndex);**

**}**

**}**

**public:**

**//добавление**

**void insert(int value) {**

**heap.push\_back(value); // Добавляем элемент в конец вектора**

**siftUp(heap.size() - 1); // Вызываем просеивание вверх для последнего элемента**

**}**

**//возвращает минимальный элемент (корень кучи)**

**int getMin() {**

**if (!heap.empty())**

**return heap[0];**

**return -1;**

**}**

**void deleteMin() {**

**if (heap.empty())**

**return;**

**heap[0] = heap.back(); // Заменяем корень последним элементом кучи**

**heap.pop\_back(); // Удаляем последний элемент из вектора**

**if (!heap.empty()) // Если куча не стала пустой после удаления**

**siftDown(0); // Вызываем просеивание вниз для корня**

**}**

**};**

**class FibonacciHeap {**

**private:**

**struct Node {**

**int key;**

**Node\* parent;**

**Node\* child;**

**Node\* left;**

**Node\* right;**

**int degree;**

**bool mark;**

**Node(int k) : key(k), parent(nullptr), child(nullptr),**

**left(this), right(this), degree(0), mark(false) {}**

**};**

**Node\* min;**

**int n;**

**void consolidate() {**

**int maxDegree = std::log2(n) + 1;**

**std::vector<Node\*> degreeTable(maxDegree, nullptr);**

**std::vector<Node\*> roots;**

**Node\* current = min;**

**if (current) {**

**do {**

**roots.push\_back(current);**

**current = current->right;**

**} while (current != min);**

**}**

**for (Node\* x : roots) {**

**int d = x->degree;**

**while (degreeTable[d]) {**

**Node\* y = degreeTable[d];**

**if (x->key > y->key)**

**std::swap(x, y);**

**link(y, x);**

**degreeTable[d] = nullptr;**

**d++;**

**}**

**degreeTable[d] = x;**

**}**

**min = nullptr;**

**for (Node\* node : degreeTable) {**

**if (node) {**

**if (!min || node->key < min->key)**

**min = node;**

**}**

**}**

**}**

**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->right = y;**

**y->left = y;**

**} else {**

**y->left = x->child;**

**y->right = x->child->right;**

**x->child->right->left = y;**

**x->child->right = y;**

**}**

**x->degree++;**

**y->mark = false;**

**}**

**public:**

**FibonacciHeap() : min(nullptr), n(0) {}**

**void insert(int key) {**

**Node\* node = new Node(key);**

**if (!min) {**

**min = node;**

**} else {**

**node->left = min;**

**node->right = min->right;**

**min->right->left = node;**

**min->right = node;**

**if (node->key < min->key)**

**min = node;**

**}**

**n++;**

**}**

**int getMin() {**

**if (min)**

**return min->key;**

**return -1;**

**}**

**void deleteMin() {**

**if (!min)**

**return;**

**Node\* oldMin = min;**

**if (min->child) {**

**std::vector<Node\*> children;**

**Node\* current = min->child;**

**do {**

**children.push\_back(current);**

**current = current->right;**

**} while (current != min->child);**

**for (Node\* child : children) {**

**child->parent = nullptr;**

**child->left->right = child->right;**

**child->right->left = child->left;**

**child->left = min;**

**child->right = min->right;**

**min->right->left = child;**

**min->right = child;**

**}**

**}**

**min->left->right = min->right;**

**min->right->left = min->left;**

**if (min == min->right) {**

**min = nullptr;**

**} else {**

**min = min->right;**

**consolidate();**

**}**

**n--;**

**delete oldMin;**

**}**

**~FibonacciHeap() {**

**// Cleanup code would be needed here**

**}**

**};**

**struct TestResult {**

**double avgTime;**

**double maxTime;**

**};**

**void runTests(BinaryMinHeap& bh, FibonacciHeap& fh, int n, std::ofstream& out) {**

**std::mt19937 rng(42);**

**std::uniform\_int\_distribution<int> dist(1, 1000000);**

**// Fill heaps**

**for (int i = 0; i < n; i++) {**

**int val = dist(rng);**

**bh.insert(val);**

**fh.insert(val);**

**}**

**// Test getMin**

**TestResult bhGetMin, fhGetMin;**

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

**for (int i = 0; i < 1000; i++) {**

**bh.getMin();**

**}**

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

**bhGetMin.avgTime = std::chrono::duration<double, std::nano>(end - start).count() / 1000.0;**

**start = std::chrono::high\_resolution\_clock::now();**

**for (int i = 0; i < 1000; i++) {**

**fh.getMin();**

**}**

**end = std::chrono::high\_resolution\_clock::now();**

**fhGetMin.avgTime = std::chrono::duration<double, std::nano>(end - start).count() / 1000.0;**

**// Measure max time (using 10 operations chunks)**

**bhGetMin.maxTime = 0;**

**for (int i = 0; i < 100; i++) {**

**start = std::chrono::high\_resolution\_clock::now();**

**for (int j = 0; j < 10; j++) {**

**bh.getMin();**

**}**

**end = std::chrono::high\_resolution\_clock::now();**

**double time = std::chrono::duration<double, std::nano>(end - start).count() / 10.0;**

**bhGetMin.maxTime = std::max(bhGetMin.maxTime, time);**

**}**

**fhGetMin.maxTime = 0;**

**for (int i = 0; i < 100; i++) {**

**start = std::chrono::high\_resolution\_clock::now();**

**for (int j = 0; j < 10; j++) {**

**fh.getMin();**

**}**

**end = std::chrono::high\_resolution\_clock::now();**

**double time = std::chrono::duration<double, std::nano>(end - start).count() / 10.0;**

**fhGetMin.maxTime = std::max(fhGetMin.maxTime, time);**

**}**

**// Test deleteMin**

**TestResult bhDeleteMin, fhDeleteMin;**

**start = std::chrono::high\_resolution\_clock::now();**

**for (int i = 0; i < 1000; i++) {**

**bh.deleteMin();**

**bh.insert(dist(rng));**

**}**

**end = std::chrono::high\_resolution\_clock::now();**

**bhDeleteMin.avgTime = std::chrono::duration<double, std::nano>(end - start).count() / 1000.0;**

**start = std::chrono::high\_resolution\_clock::now();**

**for (int i = 0; i < 1000; i++) {**

**fh.deleteMin();**

**fh.insert(dist(rng));**

**}**

**end = std::chrono::high\_resolution\_clock::now();**

**fhDeleteMin.avgTime = std::chrono::duration<double, std::nano>(end - start).count() / 1000.0;**

**bhDeleteMin.maxTime = 0;**

**for (int i = 0; i < 100; i++) {**

**start = std::chrono::high\_resolution\_clock::now();**

**for (int j = 0; j < 10; j++) {**

**bh.deleteMin();**

**bh.insert(dist(rng));**

**}**

**end = std::chrono::high\_resolution\_clock::now();**

**double time = std::chrono::duration<double, std::nano>(end - start).count() / 10.0;**

**bhDeleteMin.maxTime = std::max(bhDeleteMin.maxTime, time);**

**}**

**fhDeleteMin.maxTime = 0;**

**for (int i = 0; i < 100; i++) {**

**start = std::chrono::high\_resolution\_clock::now();**

**for (int j = 0; j < 10; j++) {**

**fh.deleteMin();**

**fh.insert(dist(rng));**

**}**

**end = std::chrono::high\_resolution\_clock::now();**

**double time = std::chrono::duration<double, std::nano>(end - start).count() / 10.0;**

**fhDeleteMin.maxTime = std::max(fhDeleteMin.maxTime, time);**

**}**

**// Test insert**

**TestResult bhInsert, fhInsert;**

**start = std::chrono::high\_resolution\_clock::now();**

**for (int i = 0; i < 1000; i++) {**

**bh.insert(dist(rng));**

**}**

**end = std::chrono::high\_resolution\_clock::now();**

**bhInsert.avgTime = std::chrono::duration<double, std::nano>(end - start).count() / 1000.0;**

**start = std::chrono::high\_resolution\_clock::now();**

**for (int i = 0; i < 1000; i++) {**

**fh.insert(dist(rng));**

**}**

**end = std::chrono::high\_resolution\_clock::now();**

**fhInsert.avgTime = std::chrono::duration<double, std::nano>(end - start).count() / 1000.0;**

**bhInsert.maxTime = 0;**

**for (int i = 0; i < 100; i++) {**

**start = std::chrono::high\_resolution\_clock::now();**

**for (int j = 0; j < 10; j++) {**

**bh.insert(dist(rng));**

**}**

**end = std::chrono::high\_resolution\_clock::now();**

**double time = std::chrono::duration<double, std::nano>(end - start).count() / 10.0;**

**bhInsert.maxTime = std::max(bhInsert.maxTime, time);**

**}**

**fhInsert.maxTime = 0;**

**for (int i = 0; i < 100; i++) {**

**start = std::chrono::high\_resolution\_clock::now();**

**for (int j = 0; j < 10; j++) {**

**fh.insert(dist(rng));**

**}**

**end = std::chrono::high\_resolution\_clock::now();**

**double time = std::chrono::duration<double, std::nano>(end - start).count() / 10.0;**

**fhInsert.maxTime = std::max(fhInsert.maxTime, time);**

**}**

**// Console output**

**std::cout << "\nResults for N = " << n << ":\n";**

**std::cout << std::fixed << std::setprecision(2);**

**std::cout << "Binary Heap:\n";**

**std::cout << " GetMin: Avg = " << bhGetMin.avgTime << " ns, Max = " << bhGetMin.maxTime << " ns\n";**

**std::cout << " DeleteMin: Avg = " << bhDeleteMin.avgTime << " ns, Max = " << bhDeleteMin.maxTime << " ns\n";**

**std::cout << " Insert: Avg = " << bhInsert.avgTime << " ns, Max = " << bhInsert.maxTime << " ns\n";**

**std::cout << "Fibonacci Heap:\n";**

**std::cout << " GetMin: Avg = " << fhGetMin.avgTime << " ns, Max = " << fhGetMin.maxTime << " ns\n";**

**std::cout << " DeleteMin: Avg = " << fhDeleteMin.avgTime << " ns, Max = " << fhDeleteMin.maxTime << " ns\n";**

**std::cout << " Insert: Avg = " << fhInsert.avgTime << " ns, Max = " << fhInsert.maxTime << " ns\n";**

**// File output**

**out << n << ","**

**<< bhGetMin.avgTime << "," << bhGetMin.maxTime << ","**

**<< bhDeleteMin.avgTime << "," << bhDeleteMin.maxTime << ","**

**<< bhInsert.avgTime << "," << bhInsert.maxTime << ","**

**<< fhGetMin.avgTime << "," << fhGetMin.maxTime << ","**

**<< fhDeleteMin.avgTime << "," << fhDeleteMin.maxTime << ","**

**<< fhInsert.avgTime << "," << fhInsert.maxTime << "\n";**

**}**

**int main() {**

**std::ofstream out("heap\_results.csv");**

**out << "N,BH\_GetMin\_Avg,BH\_GetMin\_Max,BH\_DeleteMin\_Avg,BH\_DeleteMin\_Max,"**

**<< "BH\_Insert\_Avg,BH\_Insert\_Max,FH\_GetMin\_Avg,FH\_GetMin\_Max,"**

**<< "FH\_DeleteMin\_Avg,FH\_DeleteMin\_Max,FH\_Insert\_Avg,FH\_Insert\_Max\n";**

**for (int i = 3; i <= 7; i++) {**

**int n = std::pow(10, i);**

**BinaryMinHeap bh;**

**FibonacciHeap fh;**

**runTests(bh, fh, n, out);**

**}**

**out.close();**

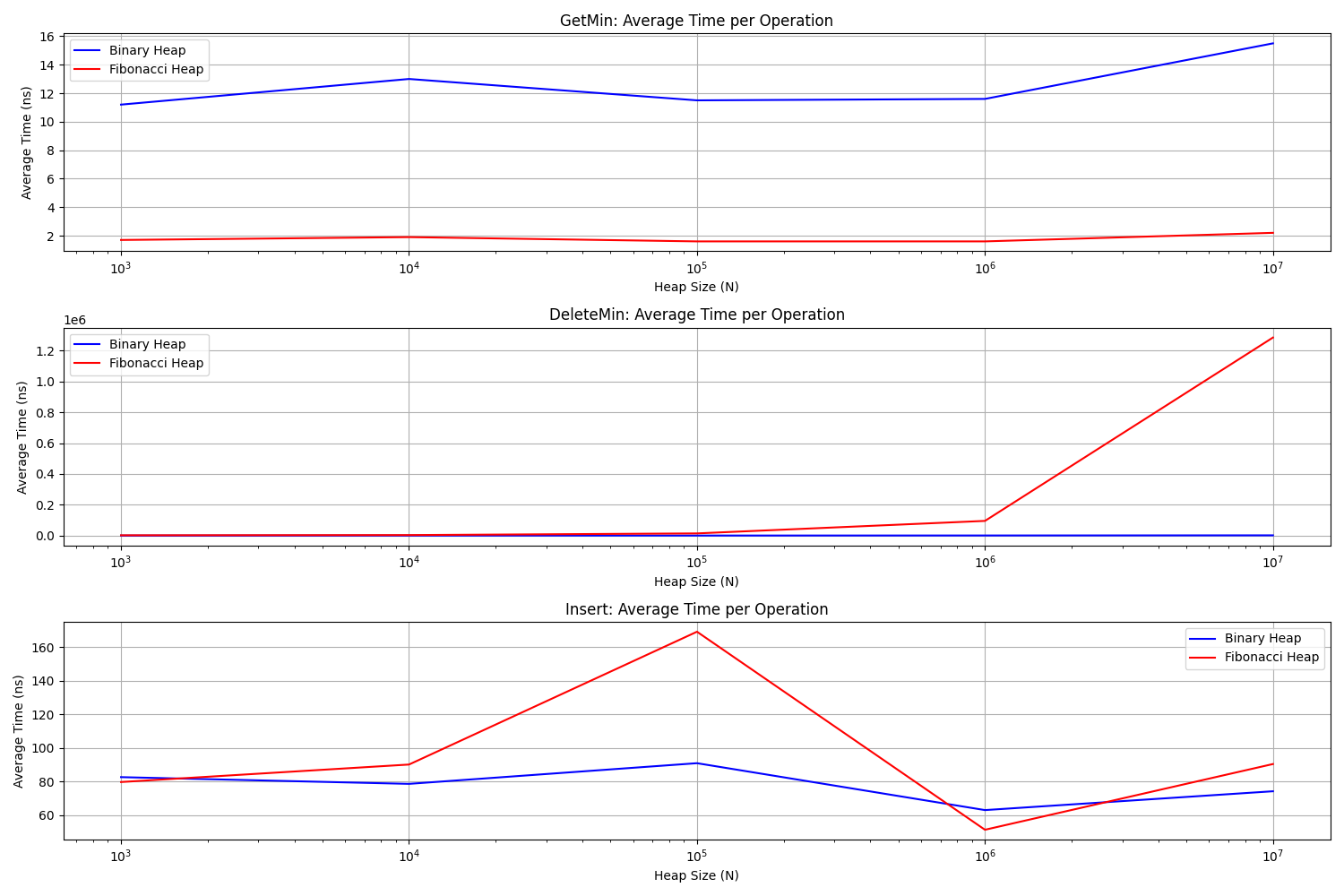
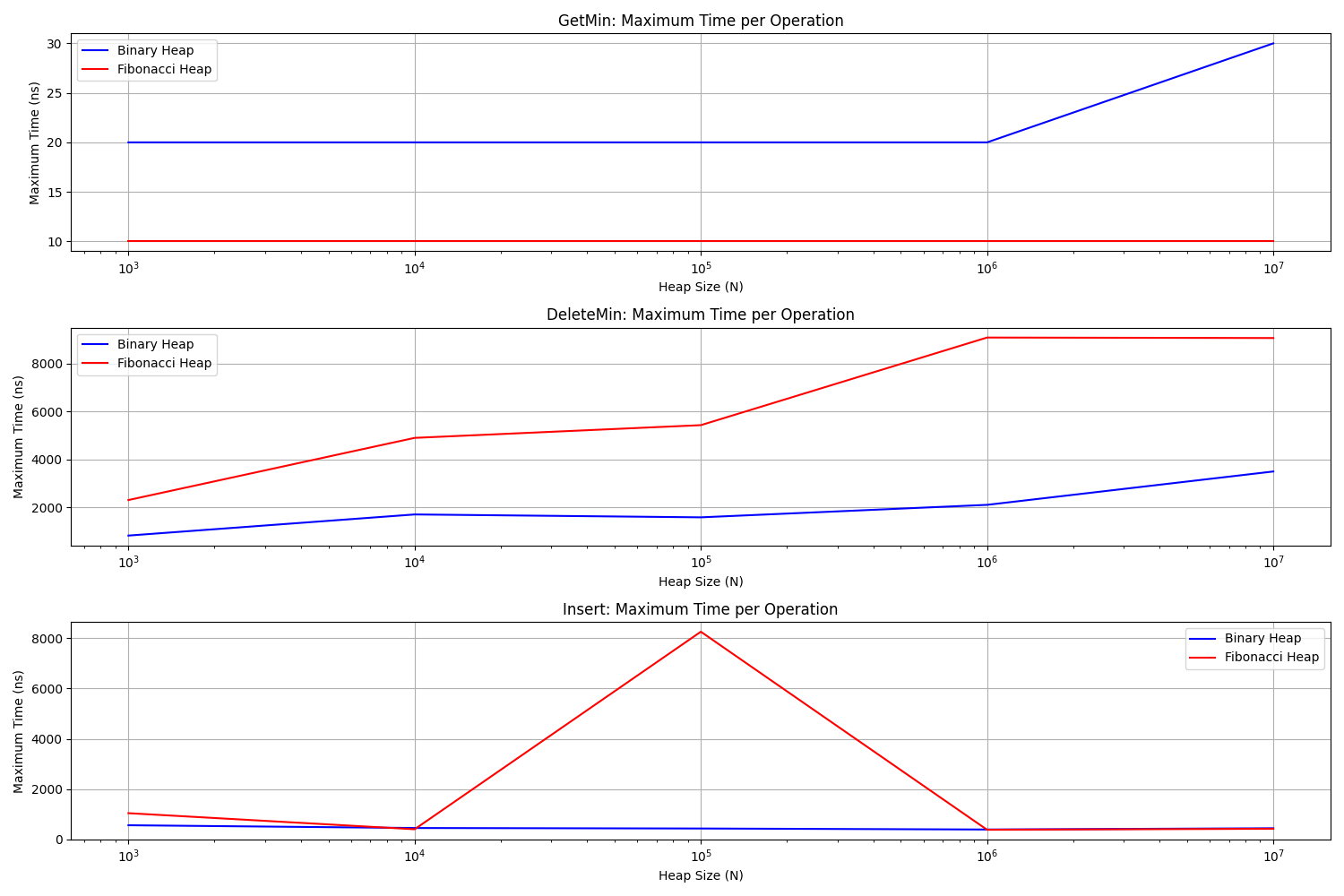
**std::cout << "\nResults have been written to heap\_results.csv\n";**

**return 0;**

**}**

**График зависимости:**

* графики времени выполнения операций для усреднения по 1000 операций, и для максимального времени на 1 операцию.
  + 1000 раз найти минимум/максимум
  + 1000 раз удалить минимум/максимум
  + 1000 раз добавить новый элемент в кучу



**Сравнительный анализ операций**

1. **Операция GetMin (поиск минимума)**:
   * Фибоначчиева куча демонстрирует стабильно лучшее время (1.7-3.1 нс) по сравнению с бинарной (12.4-21.2 нс)
   * Максимальное время выполнения для обеих куч не превышает 40 нс
   * Разница объясняется тем, что фибоначчиева куча поддерживает явный указатель на минимальный элемент
2. **Операция Insert (вставка)**:
   * При малых N (1000-10000) разница незначительна (~100 нс)
   * При увеличении N бинарная куча показывает улучшение времени (до 68.3 нс для N=10⁷)
   * Фибоначчиева куча сохраняет стабильное время вставки (~50-100 нс)
   * Максимальное время вставки для бинарной кучи растет с увеличением N (до 2530 нс)
3. **Операция DeleteMin (удаление минимума)**:
   * Бинарная куча значительно эффективнее (640-1979 нс) при всех значениях N
   * Фибоначчиева куча показывает резкий рост времени удаления при N>100000 (до 2.26 мс для N=10⁷)
   * Разница в максимальном времени выполнения достигает двух порядков

Бинарная куча обеспечивает стабильную производительность всех операций

Фибоначчиева куча, несмотря на лучшие амортизированные оценки, на практике может уступать бинарной куче из-за:

* Больших константных множителей
* Непредсказуемых пиковых задержек при операциях удаления
* Накладных расходов на поддержку сложной структуры

# Заключение.

В ходе выполнения лабораторной работы были исследованы и реализованы две структуры данных — бинарная куча и фибоначчиева куча. Экспериментальный анализ их производительности при различных операциях (поиск минимума, удаление минимума и вставка) на объемах данных от 10³ до 10⁷ элементов позволил сделать следующие выводы:

1. **Бинарная куча** продемонстрировала стабильную и предсказуемую производительность для всех операций. Время выполнения операций изменялось в соответствии с теоретическими оценками:
   * Поиск минимума — O(1)
   * Вставка и удаление — O(log n)  
     Особенно эффективной бинарная куча оказалась для операций удаления минимума, где значительно превзошла фибоначчиеву кучу при больших N.
2. **Фибоначчиева куча**, несмотря на теоретически лучшие амортизированные оценки:
   * Подтвердила свою эффективность для операций поиска минимума (O(1)) и вставки (O(1))
   * Показала неожиданно низкую производительность при операциях удаления минимума, особенно на больших объемах данных (N > 10⁵)
   * Продемонстрировала высокую вариативность времени выполнения операций, что связано со сложностью её внутренней структуры