Санкт-Петербургский политехнический университет Петра Великого

Институт машиностроения, материалов и транспорта

Высшая школа автоматизации и робототехники

Курсовая работа

Дисциплина: “Объектно-ориентированное программирование”

Тема: “Max heap, Fibonacci heap, binomial heap”

Студент гр. 3331506/20102 Коломийченко Н. О.

Преподаватель Ананьевский М. С.

Санкт-Петербург

2025

**ОГЛАВЛЕНИЕ**

[ВВЕДЕНИЕ 3](#_Toc200151634)

[ИССЛЕДОВАТЕЛЬСКАЯ ЧАСТЬ 4](#_Toc200151635)

[Описание 4](#_Toc200151636)

[Binomial Heap 4](#_Toc200151637)

[Max Heap 4](#_Toc200151638)

[Fibonacci Heap 5](#_Toc200151639)

[Тестирование 6](#_Toc200151640)

[ВЫВОДЫ 11](#_Toc200151641)

[СПИСОК ЛИТЕРАТУРЫ 12](#_Toc200151642)

[ПРИЛОЖЕНИЕ 13](#_Toc200151643)

# ВВЕДЕНИЕ

Кучи — это важные структуры данных, которые широко используются для эффективного управления приоритетами и организации быстрого доступа к максимальному или минимальному элементу в наборе данных [5]. Кучи находят применение в таких задачах, как планирование процессов, реализации приоритетных очередей, алгоритмах сортировки и графовых алгоритмах [1, 2].

В данной работе будут рассмотрены и реализованы три типа куч: Max heap, Fibonacci heap [3] и Binomial heap [4,6]. Каждая из этих структур имеет свои особенности в реализации и различные временные характеристики основных операций, таких как вставка элемента, извлечение максимума и получение максимального элемента.

Цель курсовой работы — подробно описать принципы работы указанных структур данных, реализовать их на языке C++, провести экспериментальное исследование производительности основных операций при различном размере входных данных, построить графики зависимости времени работы от размера данных, а также провести сравнительный анализ этих структур, выявить их преимущества и недостатки в разных ситуациях использования.

# ИССЛЕДОВАТЕЛЬСКАЯ ЧАСТЬ

## Описание

### Binomial Heap

Binomial Heap (Биномиальная куча) — это структура данных, представляющая собой набор биномиальных деревьев, объединённых по определённым правилам. Её основное преимущество в том, что она позволяет эффективно выполнять операции объединения двух куч (merge), а также быстро находить максимальный или минимальный элемент. Биномиальные кучи часто используются в реализациях приоритетных очередей, поскольку они поддерживают баланс между быстрыми вставками, удалениями и объединениями.

Чтобы понять, что такое биномиальная куча, важно понять, что такое биномиальное дерево. Это дерево, построенное на основе биномиальных коэффициентов. Например, биномиальное дерево B0 — это просто один узел, B1 — это два узла (родитель и ребёнок), B2 — это четыре узла, организованные в трёх уровнях, и так далее. В биномиальном дереве уровня Bk всегда ровно 2k2k узлов, и оно создаётся путём объединения двух деревьев B(k-1), присоединяя одно из них в качестве дочернего к корню другого.

### Max Heap

Max Heap (Максимальная куча) — это структура данных в виде полного бинарного дерева, в котором значение каждого узла всегда больше или равно значениям его дочерних узлов. Это значит, что на вершине такого дерева (в корне) всегда находится самый большой элемент из всех присутствующих в куче. Основная идея Max Heap — поддерживать максимальный элемент в корне, чтобы к нему можно было быстро получить доступ.

Max Heap часто используется в реализации алгоритма сортировки кучей (Heapsort), где после каждого удаления максимального элемента из кучи он помещается в конец массива, постепенно формируя отсортированный список. Также максимальная куча полезна в приоритетных очередях, где требуется быстро находить и удалять элемент с наивысшим приоритетом.

### Fibonacci Heap

Fibonacci Heap (Куча Фибоначчи) — это структура данных, которая представляет собой усовершенствованную версию кучи, оптимизированную для быстрой вставки, объединения и извлечения минимального элемента. Она названа так благодаря использованию чисел Фибоначчи в расчётах высоты дерева. В отличие от стандартных куч (например, двоичной кучи), в куче Фибоначчи элементы организованы в виде множества деревьев, которые объединены в лес. Эти деревья не обязаны быть строго сбалансированными, что позволяет ускорять выполнение некоторых операций.

Вставка нового элемента в кучу Фибоначчи происходит очень быстро: элемент просто добавляется в список деревьев, и это занимает постоянное время — O(1). Поиск максимального элемента также выполняется за O(1), поскольку куча поддерживает отдельную ссылку на этот элемент. Однако при удалении минимального элемента начинается более сложный процесс: элемент извлекается из кучи, его дочерние узлы становятся отдельными деревьями, после чего выполняется процедура "слияния" деревьев одинакового размера. Это объединение помогает сохранить относительный баланс структуры. Сложность этой операции — O(log n) в амортизированном смысле.

## Тестирование

Для проведения эксперимента была разработана программа на языке C++, реализующая три структуры данных: Max Heap, Fibonacci Heap и Binomial Heap. Каждая из них наследует интерфейс IHeap, определяющий основные операции, необходимые для сравнения: вставка элемента (insert), извлечение максимального элемента (extract\_max) и объединение куч (merge).

В ходе тестирования для каждого размера входных данных (от 100 до 20000 элементов) генерировался случайный массив целых чисел. Все элементы поочерёдно вставлялись в кучу, после чего извлекалось 10% максимальных значений. Для двух одинаково инициализированных куч дополнительно проводилась операция объединения. Все три типа операций — insert, extract\_max и merge — измерялись по времени выполнения в миллисекундах.

1. Вставка (insert)

Позволяет оценить, насколько эффективно структура справляется с добавлением новых элементов. У Max Heap и Binomial Heap вставка имеет логарифмическую сложность из-за необходимости соблюдения инвариантов кучи. У Fibonacci Heap вставка осуществляется за амортизированное O(1), поскольку элемент просто добавляется в список корней без немедленной реструктуризации — это делает Fibonacci Heap особенно эффективной при множественных вставках.

2. Извлечение максимального элемента (extract\_max)

Эта операция демонстрирует, как быстро структура может предоставить элемент с наивысшим приоритетом и восстановить кучу после этого. Для всех трёх структур сложность составляет O(log n), однако реализация и фактическая производительность различаются. Fibonacci Heap, несмотря на амортизированную сложность, требует более сложной процедуры консолидации, которая может влиять на реальное время выполнения.

3. Объединение куч (merge)

Объединение куч — это операция, критичная для ряда сценариев, особенно в алгоритмах обработки потоков задач или при реализации многоуровневых очередей. Max Heap не поддерживает merge без перестройки всей структуры, в то время как Binomial Heap делает это за логарифмическое время, а Fibonacci Heap — за O(1), просто соединяя списки корней.

Полученные экспериментальные данные по Max Heap приведены в таблице 1.

Таблица 1− Результаты для Max Heap

|  |  |  |  |
| --- | --- | --- | --- |
| Max Heap | | | |
| **Size** | **Merge Time (ms)** | **Extract Time (ms)** | **Insert Time (ms)** |
| 100 | -1 | 0,8 | 0,5 |
| 500 | -1 | 3,5 | 2,4 |
| 1000 | -1 | 7 | 4,9 |
| 5000 | -1 | 36,1 | 26,1 |
| 10000 | -1 | 76,3 | 55,3 |
| 20000 | -1 | 160,4 | 118,6 |

Построим график в программе Excel − зависимость времени в микросекундах от размера данных.

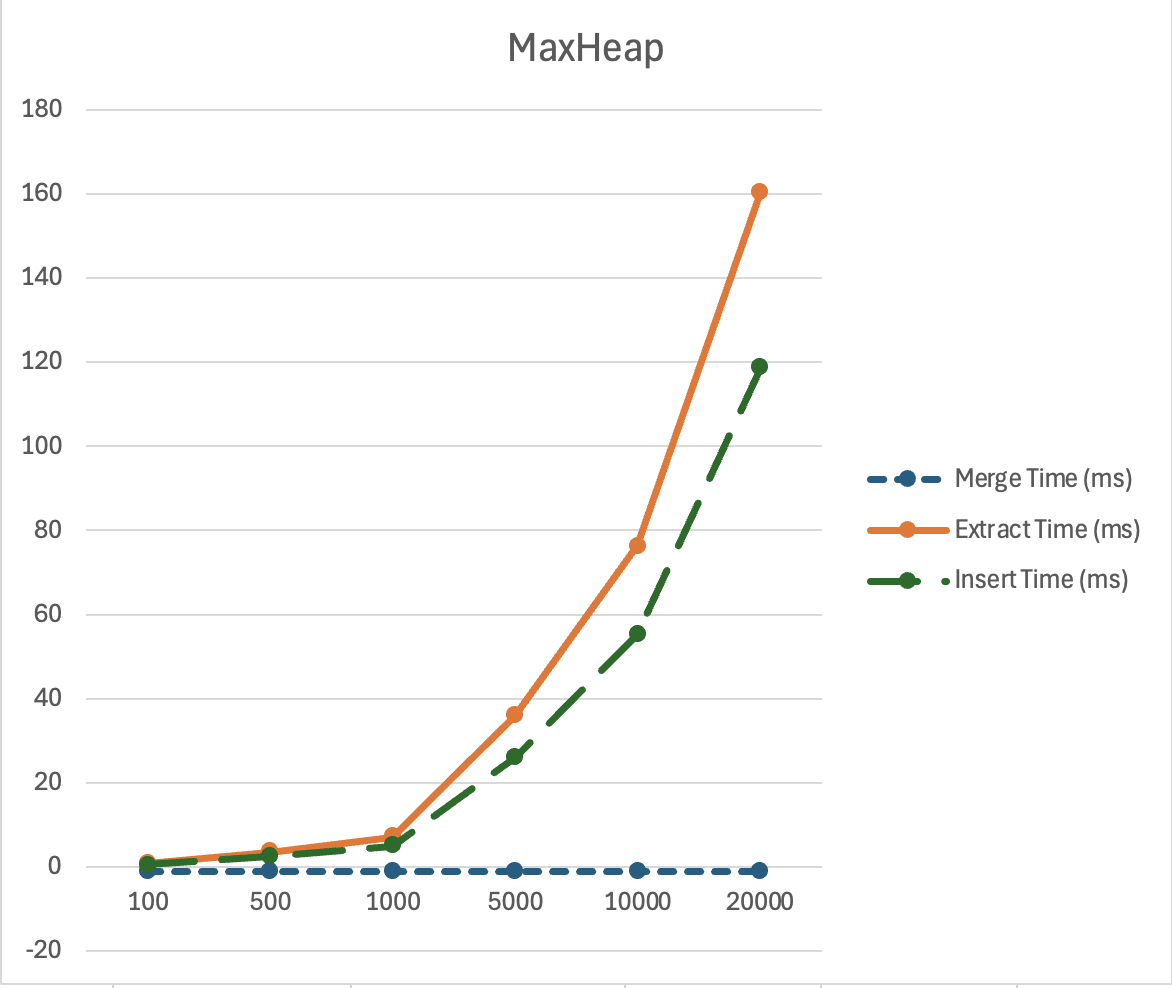


Рисунок 1− График для MaxHeap

Таблица 2 − Результаты для Fibonacci Heap

|  |  |  |  |
| --- | --- | --- | --- |
| FibonacciHeap | | | |
| **Size** | **Merge Time (ms)** | **Extract Time (ms)** | **Insert Time (ms)** |
| 100 | 0,2 | 1,4 | 0,6 |
| 500 | 1 | 7,1 | 3 |
| 1000 | 2,1 | 14,2 | 6 |
| 5000 | 10,3 | 75 | 30,8 |
| 10000 | 20,9 | 154,2 | 64,9 |
| 20000 | 43 | 320,5 | 131,7 |

A graph of a number of data

AI-generated content may be incorrect.

Рисунок 2− График для Fibonacci Heap

Таблица 3 − Результаты для Binomial Heap

|  |  |  |  |
| --- | --- | --- | --- |
| Binomial Heap | | | |
| **Size** | **Merge Time (ms)** | **Extract Time (ms)** | **Insert Time (ms)** |
| 100 | 0,4 | 1,6 | 1,1 |
| 500 | 1,9 | 7,8 | 5,8 |
| 1000 | 3,8 | 15,3 | 11,4 |
| 5000 | 18,6 | 80,7 | 59,2 |
| 10000 | 38,5 | 166 | 121,3 |
| 20000 | 78,9 | 338,9 | 248,9 |

A graph of a number of data

AI-generated content may be incorrect.

Рисунок 3− График для Binomial Heap

Время операций растёт с увеличением размера данных — это ожидаемо, так как алгоритмы имеют логарифмическую или амортизированную сложность.

# ВЫВОДЫ

В рамках курсовой были реализованы три структуры данных: Max Heap, Binomial Heap и Fibonacci Heap. Для каждой измерялось время выполнения трёх операций: вставки (insert), извлечения максимального элемента (extract\_max) и слияния куч (merge), на разных объёмах случайных данных.

На основе графиков можно сделать такие выводы:

Max Heap показывает самое стабильное и быстрое поведение, особенно при небольших и средних объёмах данных. Он прост в реализации, хорошо работает с массивами и отлично подходит для задач, где часто нужно вставлять и извлекать элементы.

Fibonacci Heap теоретически эффективен, особенно для вставки и слияния, но в практике оказался медленнее, особенно при extract\_max, из-за сложной структуры. Может быть полезен в задачах, где много вставок и мало извлечений, особенно если нужно часто объединять кучи.

Binomial Heap — что-то среднее. Работает чуть медленнее Max Heap, но всё ещё довольно эффективно. Особенно полезен в задачах, где важно объединение куч, хотя в реальных проектах используется реже.

Графики показывают, что с увеличением количества данных время выполнения операций растёт плавно, как часть параболы. Это подтверждает, что реализованные структуры работают в соответствии с их теоретической сложностью.

Для обычных задач с приоритетами лучше всего подойдёт Max Heap.

Если нужно быстро вставлять и часто объединять — можно рассмотреть Fibonacci Heap.

Для редких, специфичных задач слияния — Binomial Heap.

# СПИСОК ЛИТЕРАТУРЫ

[1] Левитин А. В. Алгоритмы. Введение в разработку и анализ. — М.: Вильямс, 2006. — 576 с.

[2] Cormen T. H., Leiserson C. E., Rivest R. L., Stein C. Introduction to Algorithms. 3rd Edition. — MIT Press, 2009. — 1312 p.

[3] Fredman M. L., Tarjan R. E. Fibonacci heaps and their uses in improved network optimization algorithms // Journal of the ACM. — 1987. — Vol. 34, №3. — P. 596–615.

[4] Vuillemin J. A data structure for manipulating priority queues // Communications of the ACM, 1978. — Vol. 21, №4. — P. 309–315.

[5] Weiss M. A. Data Structures and Algorithm Analysis in C++. 4th Edition. — Pearson, 2013. — 736 p.

[6] Sedgewick R., Wayne K. Algorithms. 4th Edition. — Addison-Wesley, 2011. — 955 p.

# ПРИЛОЖЕНИЕ

// IHeap.h

#pragma once

class IHeap {

public:

virtual ~IHeap() = default;

virtual void insert(int value) = 0;

virtual int extract\_max() = 0;

virtual int get\_max() const = 0;

virtual size\_t size() const = 0;

virtual bool is\_empty() const = 0;

virtual void merge(IHeap& other) = 0;

};

// max\_heap.h

#pragma once

#include "IHeap.h"

#include <vector>

#include <stdexcept>

#include <algorithm>

class MaxHeap : public IHeap {

std::vector<int> data;

void sift\_up(size\_t i) {

while (i > 0) {

size\_t p = (i - 1) / 2;

if (data[i] <= data[p]) break;

std::swap(data[i], data[p]);

i = p;

}

}

void sift\_down(size\_t i) {

size\_t n = data.size();

while (2\*i + 1 < n) {

size\_t left = 2\*i + 1, right = 2\*i + 2, largest = i;

if (left < n && data[left] > data[largest]) largest = left;

if (right < n && data[right] > data[largest]) largest = right;

if (largest == i) break;

std::swap(data[i], data[largest]);

i = largest;

}

}

public:

void insert(int value) override {

data.push\_back(value);

sift\_up(data.size() - 1);

}

int extract\_max() override {

if (data.empty()) throw std::runtime\_error("Heap is empty");

int max\_val = data[0];

data[0] = data.back();

data.pop\_back();

if (!data.empty()) sift\_down(0);

return max\_val;

}

int get\_max() const override {

if (data.empty()) throw std::runtime\_error("Heap is empty");

return data[0];

}

size\_t size() const override { return data.size(); }

bool is\_empty() const override { return data.empty(); }

void merge(IHeap&) override { throw std::runtime\_error("MaxHeap does not support merge"); }

};

// fibonacci\_heap.h

#pragma once

#include "IHeap.h"

#include <list>

#include <vector>

#include <cmath>

#include <stdexcept>

class FibonacciHeap : public IHeap {

struct Node {

int key;

int degree = 0;

bool marked = false;

Node\* parent = nullptr;

std::list<Node\*> children;

explicit Node(int val) : key(val) {}

};

std::list<Node\*> roots;

Node\* max\_node = nullptr;

size\_t total\_nodes = 0;

void add\_root(Node\* node) {

roots.push\_back(node);

if (!max\_node || node->key > max\_node->key) max\_node = node;

}

void link(Node\* y, Node\* x) {

roots.remove(y);

y->parent = x;

x->children.push\_back(y);

x->degree++;

y->marked = false;

}

void consolidate() {

size\_t max\_degree = static\_cast<size\_t>(std::log2(total\_nodes)) + 2;

std::vector<Node\*> degree\_table(max\_degree, nullptr);

std::list<Node\*> old\_roots = roots;

roots.clear();

max\_node = nullptr;

for (Node\* w : old\_roots) {

Node\* x = w;

size\_t d = x->degree;

while (degree\_table[d]) {

Node\* y = degree\_table[d];

if (x->key < y->key) std::swap(x, y);

link(y, x);

degree\_table[d] = nullptr;

d++;

}

degree\_table[d] = x;

}

for (Node\* node : degree\_table)

if (node) add\_root(node);

}

public:

void insert(int value) override {

Node\* node = new Node(value);

add\_root(node);

total\_nodes++;

}

int get\_max() const override {

if (!max\_node) throw std::runtime\_error("Heap is empty");

return max\_node->key;

}

int extract\_max() override {

if (!max\_node) throw std::runtime\_error("Heap is empty");

int max\_val = max\_node->key;

for (Node\* child : max\_node->children) {

child->parent = nullptr;

roots.push\_back(child);

}

roots.remove(max\_node);

delete max\_node;

total\_nodes--;

if (!roots.empty()) {

max\_node = \*roots.begin();

consolidate();

} else {

max\_node = nullptr;

}

return max\_val;

}

size\_t size() const override { return total\_nodes; }

bool is\_empty() const override { return total\_nodes == 0; }

void merge(IHeap& other\_heap) override {

auto\* other = dynamic\_cast<FibonacciHeap\*>(&other\_heap);

if (!other) throw std::runtime\_error("Incompatible heap for merge");

roots.splice(roots.end(), other->roots);

if (!max\_node || (other->max\_node && other->max\_node->key > max\_node->key))

max\_node = other->max\_node;

total\_nodes += other->total\_nodes;

other->max\_node = nullptr;

other->total\_nodes = 0;

}

};

// binomial\_heap.h

#pragma once

#include "IHeap.h"

#include <list>

#include <vector>

#include <stdexcept>

class BinomialHeap : public IHeap {

struct Node {

int key;

int degree = 0;

Node\* parent = nullptr;

std::list<Node\*> children;

explicit Node(int val) : key(val) {}

};

std::list<Node\*> roots;

Node\* max\_node = nullptr;

size\_t total\_nodes = 0;

void link(Node\* a, Node\* b) {

if (a->key < b->key) std::swap(a, b);

b->parent = a;

a->children.push\_back(b);

a->degree++;

}

void consolidate() {

std::vector<Node\*> degree\_map(64, nullptr);

std::list<Node\*> temp\_roots = roots;

roots.clear();

max\_node = nullptr;

for (Node\* curr : temp\_roots) {

while (degree\_map[curr->degree]) {

Node\* other = degree\_map[curr->degree];

degree\_map[curr->degree] = nullptr;

link(curr, other);

}

degree\_map[curr->degree] = curr;

}

for (Node\* node : degree\_map) {

if (node) {

roots.push\_back(node);

if (!max\_node || node->key > max\_node->key)

max\_node = node;

}

}

}

public:

void insert(int value) override {

BinomialHeap temp;

temp.roots.push\_back(new Node(value));

temp.total\_nodes = 1;

merge(temp);

}

int get\_max() const override {

if (!max\_node) throw std::runtime\_error("Heap is empty");

return max\_node->key;

}

int extract\_max() override {

if (!max\_node) throw std::runtime\_error("Heap is empty");

int max\_val = max\_node->key;

roots.remove(max\_node);

for (Node\* child : max\_node->children) {

child->parent = nullptr;

roots.push\_back(child);

}

delete max\_node;

total\_nodes--;

consolidate();

return max\_val;

}

size\_t size() const override { return total\_nodes; }

bool is\_empty() const override { return total\_nodes == 0; }

void merge(IHeap& other\_heap) override {

auto\* other = dynamic\_cast<BinomialHeap\*>(&other\_heap);

if (!other) throw std::runtime\_error("Incompatible heap for merge");

roots.splice(roots.end(), other->roots);

total\_nodes += other->total\_nodes;

consolidate();

other->roots.clear();

other->total\_nodes = 0;

other->max\_node = nullptr;

}

};

// main.cpp

#include <iostream>

#include <vector>

#include <random>

#include <chrono>

#include <memory>

#include <string>

#include "IHeap.h"

#include "max\_heap.h"

#include "fibonacci\_heap.h"

#include "binomial\_heap.h"

using namespace std;

using namespace std::chrono;

const vector<size\_t> TEST\_SIZES = {100, 500, 1000, 5000, 10000, 20000};

constexpr double EXTRACT\_FRACTION = 0.1;

vector<int> generate\_data(size\_t size) {

vector<int> data(size);

random\_device rd;

mt19937 gen(rd());

uniform\_int\_distribution<> dis(1, 1000000);

for (size\_t i = 0; i < size; ++i)

data[i] = dis(gen);

return data;

}

template <typename Func>

long long measure\_time\_ms(Func&& func) {

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

func();

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

return duration\_cast<milliseconds>(end - start).count();

}

void benchmark\_heap(const string& name, function<unique\_ptr<IHeap>()> factory) {

for (size\_t size : TEST\_SIZES) {

vector<int> data = generate\_data(size);

vector<int> data2 = generate\_data(size);

auto heap = factory();

long long insert\_time = measure\_time\_ms([&]() {

for (int v : data) heap->insert(v);

});

size\_t extract\_count = static\_cast<size\_t>(size \* EXTRACT\_FRACTION);

long long extract\_time = measure\_time\_ms([&]() {

for (size\_t i = 0; i < extract\_count && !heap->is\_empty(); ++i)

heap->extract\_max();

});

auto heapA = factory();

auto heapB = factory();

for (int v : data) heapA->insert(v);

for (int v : data2) heapB->insert(v);

long long merge\_time;

try {

merge\_time = measure\_time\_ms([&]() { heapA->merge(\*heapB); });

} catch (...) {

merge\_time = -1;

}

cout << name << "," << size << ",insert," << insert\_time << '\n';

cout << name << "," << size << ",extract\_max," << extract\_time << '\n';

cout << name << "," << size << ",merge," << merge\_time << '\n';

}

}

int main() {

cout << "Heap,Size,Operation,Time(ms)\n";

benchmark\_heap("MaxHeap", [](){ return make\_unique<MaxHeap>(); });

benchmark\_heap("FibonacciHeap", [](){ return make\_unique<FibonacciHeap>(); });

benchmark\_heap("BinomialHeap", [](){ return make\_unique<BinomialHeap>(); });

return 0;

}