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

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.

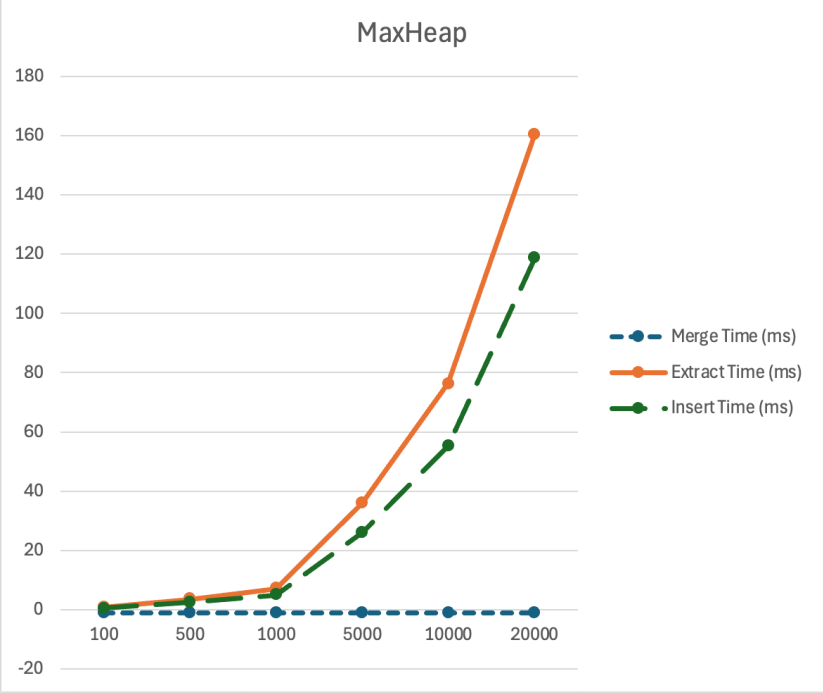

Рисунок 1– График для MaxHeap

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

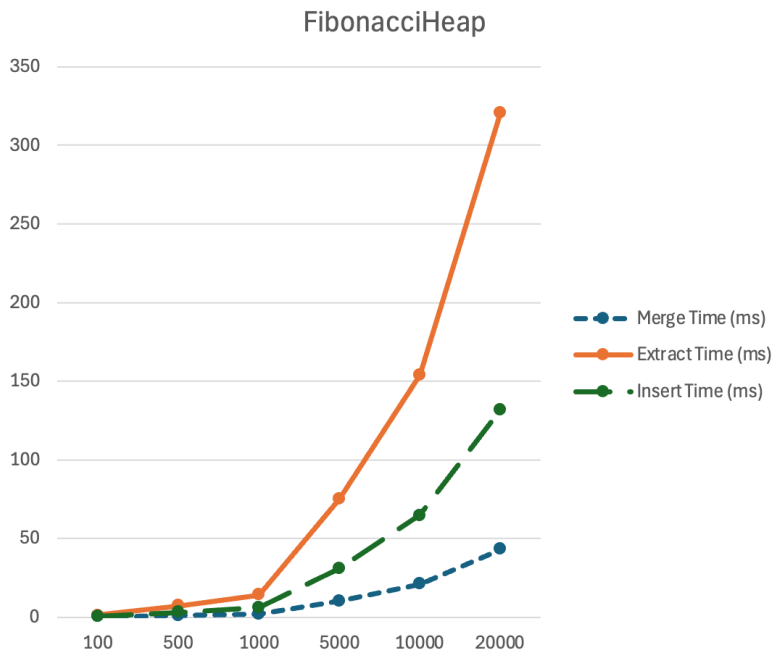

Рисунок 2– График для Fibonacci Heap

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

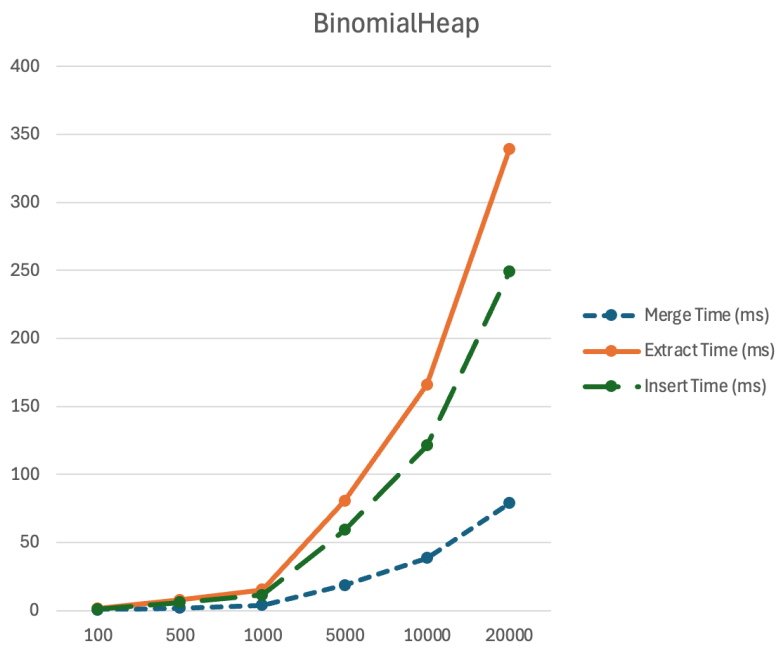

Рисунок 3– График для Binomial Heap

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;
};
```

```cpp
// 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;
}
```