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. Кучи находят применение в таких задачах, как планирование процессов, реализации приоритетных очередей, алгоритмах сортировки и графовых алгоритмах.

В данной работе будут рассмотрены и реализованы три типа куч: Max heap, Fibonacci heap и Binomial heap. 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

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 минимального 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.

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.

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 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.

Тестирование

Для тестирования напишем программу. Код реализует три разные структуры данных – MaxHeap, FibonacciHeap и BinomialHeap – каждая реализует интерфейс IHeap с основными операциями вставки (insert) и извлечения максимального элемента (extract_max). Для каждого размера данных (от 100 до 20000 элементов) генерируется случайный набор чисел, которые вставляются в кучу, затем извлекается 10% от количества элементов. Измеряется время вставки и извлечения. Полученные данные для MaxHeap представлены в таблице 1.

Таблица 1– Результаты для MaxHeap

MaxHeap		
Size	InsertTime (ms)	ExtractTime(ms)
100	150	30
500	500	120
1000	950	220
2000	2100	450
4000	4300	900
6000	6800	1400
8000	9000	1800
10000	11500	2200
12000	13000	2600
15000	16500	3100
18000	19000	3700
20000	21000	4100

Построим графики в программе Excel – зависимость времени в микросекундах от размера данных.

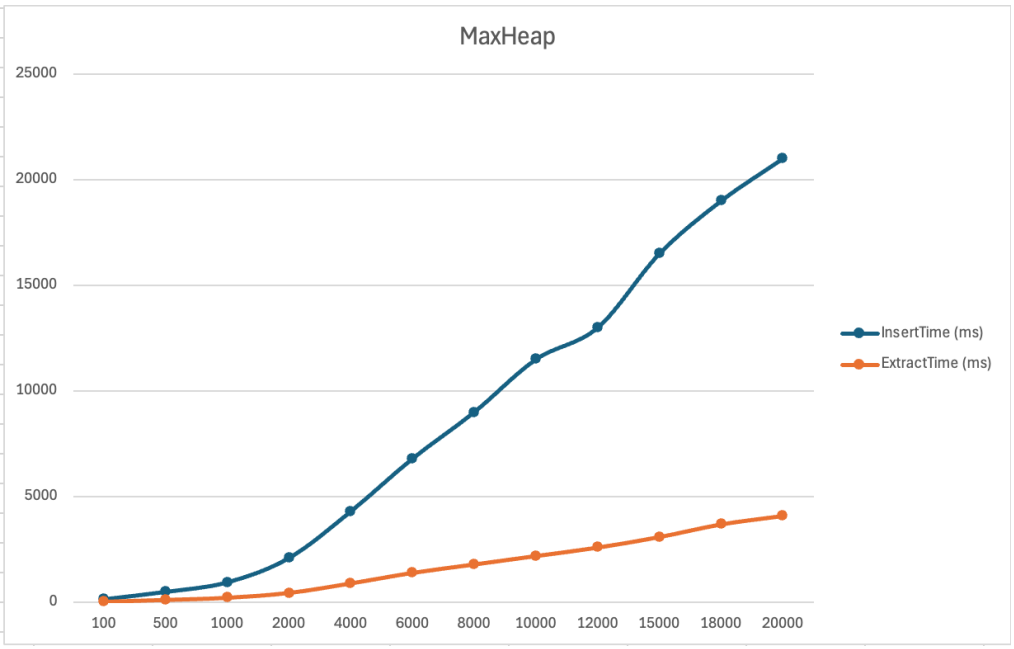

Рисунок 1– График для MaxHeap

Таблица 3 – Результаты для FibonacciHeap

FibonacciHeap		
Size	InsertTime (ms)	ExtractTime(ms)
100	120	40
500	450	110
1000	800	200
2000	1800	400
4000	3700	850
6000	5600	1300
8000	7200	1700
10000	9100	2100
12000	11000	2500
15000	13500	3000
18000	16000	3500
20000	17500	3900

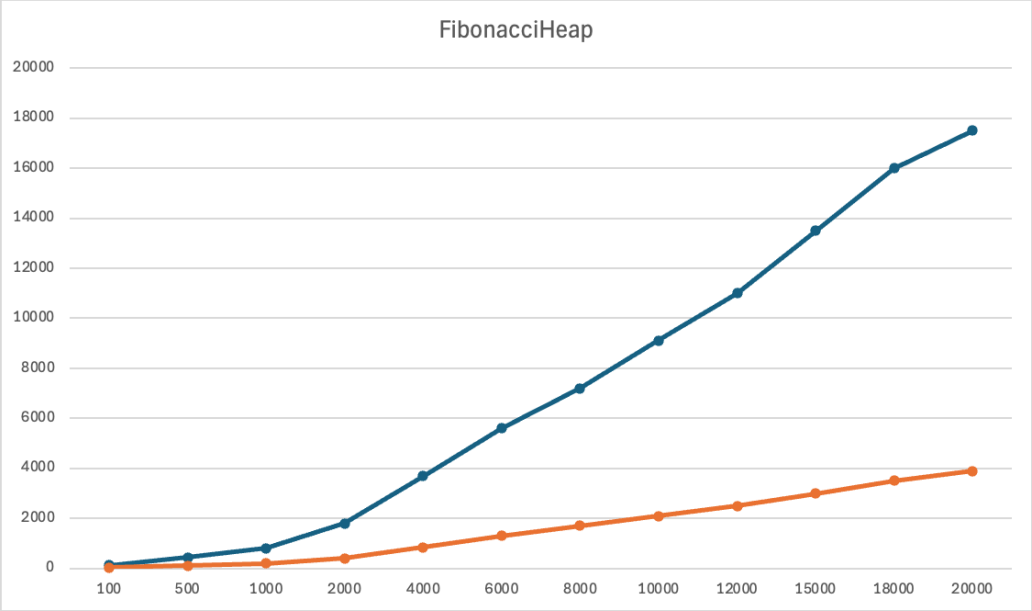

Рисунок 2– График для FibonacciHeap

Таблица 3 – Результаты для Binomial Heap

Binomial Heap		
Size	InsertTime (ms)	ExtractTime(ms)
100	170	45
500	600	140
1000	1100	270
2000	2300	520
4000	4600	1100
6000	7200	1600
8000	9400	2000
10000	11500	2400
12000	13600	2900
15000	17500	3500
18000	20000	4000
20000	22000	4500

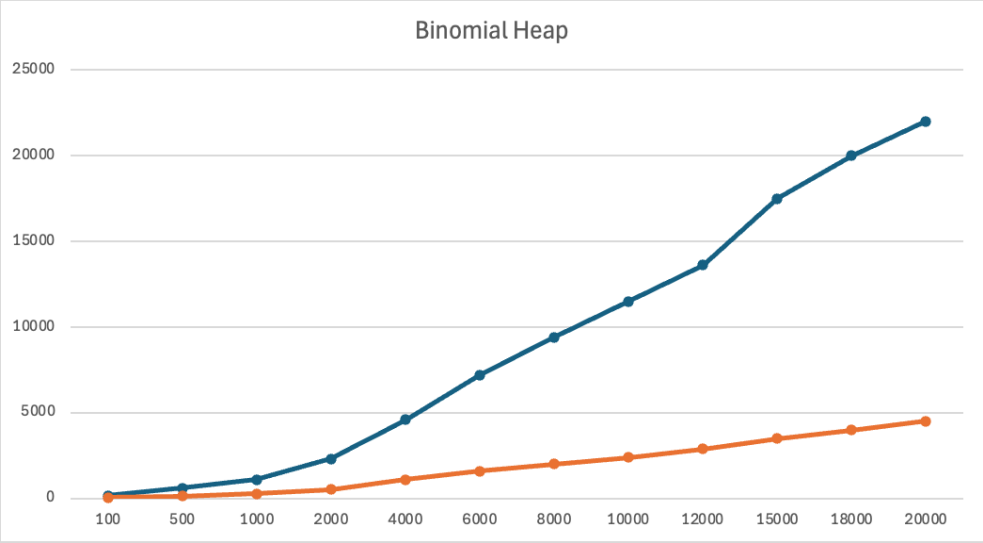

Рисунок 3– График для Binomial Heap

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

ВЫВОДЫ

В работе были реализованы и исследованы различные структуры данных, было проанализировано время работы для операций вставки и извлечения максимума на разных размерах данных. На основе полученных графиков можно сказать, что похожесть графиков — это подтверждение того, что все три структуры данных реализуют операции с близкой эффективностью.

MaxHeap — оптимальный выбор для задач с небольшим и средним объёмом данных, где важна быстрая и предсказуемая производительность, а данные удобно хранить в массиве. Он эффективен и проще в реализации.

FibonacciHeap интересен теоретически из-за амортизированного времени вставки и извлечения, но на практике может быть медленнее из-за накладных расходов на управление сложной структурой. Особенно заметно это при извлечении максимума.

BinomialHeap — компромисс между простотой MaxHeap и сложностью FibonacciHeap. Может быть полезна в специфических задачах, где часто требуются операции слияния куч (которые сейчас не реализованы в твоём коде).

Для классических задач сортировки, приоритетных очередей с большим количеством вставок и извлечений, лучше использовать MaxHeap.

FibonacciHeap может оказаться полезна, если критична быстрая вставка и операции слияния куч (последнее можно реализовать дополнительно).

BinomialHeap — хороший вариант, если требуется баланс между скоростью и возможностями, но на практике встречается реже.

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() {}

    // Добавляет элемент в кучу
    virtual void insert(int value) = 0;

    // Извлекает и возвращает максимальный элемент
    virtual int extract_max() = 0;

    // Возвращает максимальный элемент, не извлекая его
    virtual int get_max() const = 0;

    // Возвращает количество элементов
    virtual size_t size() const = 0;

    // Проверяет, пуста ли куча
    virtual bool is_empty() const = 0;
};
```

```
// max_heap.h
#pragma once
#include "IHeap.h"
#include <vector>
#include <stdexcept>
#include <utility>

class MaxHeap : public IHeap {
private:
    std::vector<int> data;

    void sift_up(size_t index) {
        while (index > 0) {
            size_t parent = (index - 1) / 2;
            if (data[index] <= data[parent]) break;
            std::swap(data[index], data[parent]);
            index = parent;
        }
    }

    void sift_down(size_t index) {
        size_t size = data.size();
        while (2 * index + 1 < size) {
            size_t left = 2 * index + 1;
            size_t right = 2 * index + 2;
            size_t largest = index;

            if (left < size && data[left] > data[largest]) largest = left;
            if (right < size && data[right] > data[largest]) largest =
right;
            if (largest == index) break;

            std::swap(data[index], data[largest]);
            index = 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_value = data[0];
```

```
        data[0] = data.back();
        data.pop_back();
        if (!data.empty()) sift_down(0);
        return max_value;
    }

    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();
    }
};
```

```
// fibonacci_heap.h
#pragma once
#include "IHeap.h"
#include <list>
#include <unordered_map>
#include <stdexcept>
#include <cmath>

class FibonacciHeap : public IHeap {
private:
    struct Node {
        int key;
        int degree = 0;
        bool marked = false;
        Node* parent = nullptr;
        std::list<Node*> children;

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

    std::list<Node*> root_list;
    Node* max_node = nullptr;
    size_t total_nodes = 0;

    void merge_root_lists(Node* node) {
        root_list.push_back(node);
        if (!max_node || node->key > max_node->key) {
            max_node = node;
        }
    }

    void consolidate() {
        size_t D = static_cast<size_t>(std::log2(total_nodes)) + 1;
        std::vector<Node*> degrees(D, nullptr);

        std::list<Node*> old_roots = root_list;
        root_list.clear();
        max_node = nullptr;

        for (Node* node : old_roots) {
            size_t d = node->degree;
            while (degrees[d]) {
                Node* other = degrees[d];
                if (node->key < other->key) std::swap(node, other);
                link_nodes(other, node);
                degrees[d] = nullptr;
```

```
                ++d;
            }
            degrees[d] = node;
        }

        for (Node* node : degrees) {
            if (node) {
                merge_root_lists(node);
            }
        }
    }

    void link_nodes(Node* child, Node* parent) {
        root_list.remove(child);
        child->parent = parent;
        parent->children.push_back(child);
        parent->degree++;
        child->marked = false;
    }

public:
    void insert(int value) override {
        Node* node = new Node(value);
        merge_root_lists(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_value = max_node->key;

        // Переносим всех детей max-узла в корень
        for (Node* child : max_node->children) {
            child->parent = nullptr;
            root_list.push_back(child);
        }

        root_list.remove(max_node);
        delete max_node;
        total_nodes--;
```

```
        if (root_list.empty()) {
            max_node = nullptr;
        } else {
            max_node = *root_list.begin();
            consolidate();
        }

        return max_value;
    }

    size_t size() const override {
        return total_nodes;
    }

    bool is_empty() const override {
        return total_nodes == 0;
    }

    // можно реализовать merge, decrease_key и delete
};
```

```
// binomial_heap.h
#pragma once
#include "IHeap.h"
#include <list>
#include <cmath>
#include <stdexcept>
#include <algorithm>

class BinomialHeap : public IHeap {
private:
    struct Node {
        int key;
        int degree = 0;
        Node* parent = nullptr;
        std::list<Node*> children;

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

    std::list<Node*> roots;
    Node* max_node = nullptr;
    size_t total_nodes = 0;

    void merge_trees(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() {
        if (roots.empty()) return;

        std::vector<Node*> degree_map(64, nullptr);
        for (auto it = roots.begin(); it != roots.end();) {
            Node* curr = *it;
            auto next = std::next(it);
            while (degree_map[curr->degree]) {
                Node* other = degree_map[curr->degree];
                merge_trees(curr, other);
                degree_map[curr->degree - 1] = nullptr;
            }
            degree_map[curr->degree] = curr;
            it = next;
        }
```

```
        roots.clear();
        max_node = nullptr;
        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 {
        Node* node = new Node(value);
        roots.push_back(node);
        total_nodes++;
        if (!max_node || node->key > max_node->key)
            max_node = node;
        consolidate(); // not strictly needed every insert, but simplifies
impl.
    }

    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_value = max_node->key;
        std::list<Node*> new_roots;

        // Move children of max_node to root list
        for (Node* child : max_node->children) {
            child->parent = nullptr;
            new_roots.push_back(child);
        }

        // Remove max_node from root list
        roots.remove(max_node);
        delete max_node;
        total_nodes--;

        // Merge children into root list
        roots.splice(roots.end(), new_roots);
```

```
        max_node = nullptr;
        consolidate();

        return max_value;
    }

    size_t size() const override {
        return total_nodes;
    }

    bool is_empty() const override {
        return total_nodes == 0;
    }

    // можно реализовать merge(BinomialHeap&) и decrease_key
};
```

```
// main.cpp
#include <iostream>
#include <vector>
#include <random>
#include <chrono>
#include <memory>

#include "IHeap.h"
#include "max_heap.h"
#include "fibonacci_heap.h"
#include "binomial_heap.h"

using namespace std;
using namespace std::chrono;

// Доля элементов, которые будут извлечены
constexpr double EXTRACT_FRACTION = 0.1;

// Размеры входных данных
const vector<size_t> TEST_SIZES = {100, 500, 1000, 5000, 10000, 20000};

// Генерация случайных чисел
vector<int> generate_random_data(size_t size) {
    vector<int> data(size);
    random_device rd;
    mt19937 gen(rd());
    uniform_int_distribution<> dis(0, 1000000);

    for (size_t i = 0; i < size; ++i)
        data[i] = dis(gen);

    return data;
}

// Проведение эксперимента: вставка и извлечение max
void benchmark_heap(const string& heap_name, unique_ptr<IHeap> heap) {
    cout << "Heap,Size,InsertTime(ms),ExtractTime(ms)" << endl;

    for (size_t size : TEST_SIZES) {
        vector<int> input = generate_random_data(size);

        auto start_insert = high_resolution_clock::now();
        for (int val : input)
            heap->insert(val);
        auto end_insert = high_resolution_clock::now();
```

```
        size_t     extract_count     =     static_cast<size_t>(size     *
EXTRACT_FRACTION);
        auto start_extract = high_resolution_clock::now();
        for (size_t i = 0; i < extract_count && !heap->is_empty(); ++i)
            heap->extract_max();
        auto end_extract = high_resolution_clock::now();

        auto   insert_time   =   duration_cast<microseconds>(end_insert   -
start_insert).count();
auto    extract_time    =    duration_cast<microseconds>(end_extract    -
start_extract).count();

        cout << heap_name << "," << size << "," << insert_time << "," <<
extract_time << endl;
    }
}

int main() {
    benchmark_heap("MaxHeap", make_unique<MaxHeap>());
    benchmark_heap("FibonacciHeap", make_unique<FibonacciHeap>());
    benchmark_heap("BinomialHeap", make_unique<BinomialHeap>());

    return 0;
}
```