Санкт-Петербургский политехнический университет Петра Великого
Институт машиностроения, материала и транспорта
Высшая школа автоматизации и робототехники

Курсовая работа

Дисциплина: Объектно-ориентированное программирование

Тема: Max heap, Fibonacci heap, binomial heap

Выполнил студент гр. 3331506/20102 Рябков Н. А.

Преподаватель Ананьевский М. С.

Санкт-Петербург

2025

2. Оглавление

Введение

Современные вычислительные задачи требуют эффективного управления данными, особенно в контексте операций, где критически важны скорость и оптимальность. В этом аспекте особую роль играют структуры данных, известные как **кучи** (heaps), которые обеспечивают выполнение ключевых операций (вставка, извлечение максимума/минимума, обновление элементов) с предсказуемой временной сложностью. В данной работе рассматриваются три типа куч: **Max heap**, **Binomial heap** и **Fibonacci heap**. Эти структуры, несмотря на общую цель, различаются внутренней организацией, алгоритмами операций и областями применения, что делает их изучение актуальным для оптимизации разнообразных алгоритмов.

Предмет исследования

Анализ структуры, основных операций, временной сложности и практической применимости указанных куч. Каждая из них обладает уникальными особенностями:

- **Max heap** — бинарная куча, обеспечивающая быстрое извлечение максимума за время O(1) и восстановление свойств за O(logn), что делает её незаменимой в алгоритмах сортировки (например, heapsort).
- **Binomial heap** — коллекция биномиальных деревьев, поддерживающая эффективное объединение куч за O(logn), что полезно в задачах параллельной обработки.
- **Fibonacci heap** — структура с амортизированной сложностью O(1) для вставки и обновления элементов, которая ускоряет алгоритмы поиска кратчайшего пути (Дейкстры) и построения минимального остовного дерева (Прима).

Постановка задачи

Сравнительный анализ этих структур на основе теоретической сложности операций и экспериментальной проверки их производительности. Несмотря на то, что Fibonacci heap обладает наилучшей амортизированной сложностью, её практическое применение часто ограничено из-за высокой константы времени, что требует детального изучения условий, при которых та или иная куча становится оптимальным выбором.

Значимость работы

Обусловлена необходимостью понимания компромиссов между теоретической и практической эффективностью структур данных. Например, Fibonacci heap, описанная Фредманом и Тарьяном в 1984 году [1], остается эталоном для алгоритмов с частыми операциями уменьшения ключа, однако в реальных приложениях её преимущества проявляются только на больших объемах данных.

Область применения

Системы реального времени (Max heap для приоритетных очередей).
Графовые алгоритмы (Binomial и Fibonacci heap для оптимизации операций объединения и обновления).
Машинное обучение (управление выборками данных в обучении моделей).

1. Основная часть

1.1 Max heap

Структура данных и основные операции

Max heap (максимальная куча) — это структура данных, реализующая **бинарное дерево**, в котором значение каждого родительского узла больше или равно значениям его дочерних узлов. Благодаря этому свойству корень дерева всегда содержит максимальный элемент. Для удобства хранения и операций Max heap часто реализуется в виде **массива**, где для узла с индексом i:
Левый дочерний узел имеет индекс 2i+1,
Правый дочерний узел — 2i+2,
Родительский узел — ⌊(i−1)/2⌋.

Основные операции:

1. **Вставка элемента (insert):**

Новый элемент добавляется в конец массива.
Производится операция **sift-up** (просеивание вверх): элемент сравнивается с родителем, и если он больше, они меняются местами. Процесс повторяется до восстановления свойств кучи.

2. **Извлечение максимума (extract_max):**

Корневой элемент (максимум) сохраняется для возврата.
Последний элемент массива перемещается на место корня.
Выполняется операция **sift-down** (просеивание вниз): элемент сравнивается с наибольшим из дочерних, и, если он меньше, происходит обмен. Процесс повторяется до восстановления свойств кучи.

3. **Построение кучи из массива (build_heap):**

Для массива из nn элементов операция выполняется за O(n) времени. Алгоритм последовательно применяет sift-down ко всем узлам, начиная с последнего нелистового узла (⌊n/2⌋−1).

4. **Получение максимума (get_max):**

Возвращает корневой элемент за O (1).

Временная и пространственная сложность

Вставка (insert): O(logn) из-за sift-up.
Извлечение максимума (extract_max): O(logn) из-за sift-down.
Построение кучи (build_heap): O(n) (доказательство основано на суммировании высот узлов [2]).

Получение максимума (get_max): O(1).

Пространственная сложность: O(n), так как элементы хранятся в массиве.

Примеры применения

Max heap широко используется в следующих задачах:

1. **Сортировка (Heapsort):**

Алгоритм строит кучу из массива (O(n)), затем последовательно извлекает максимум O(nlogn)), формируя отсортированный массив.

2. **Приоритетные очереди:**

Обеспечивает быстрое извлечение элемента с наивысшим приоритетом (например, в планировщиках задач).

3. **Алгоритм Дейкстры (предварительная версия):**

Для эффективного выбора следующей вершины с минимальным расстоянием (в случае Min heap).

1.2 Binomial heap

Структура и базовые операции

Binomial heap (биномиальная куча) — это структура данных, состоящая из набора **биномиальных деревьев**, каждое из которых удовлетворяет свойству кучи (родительский узел больше или равен дочерним для max-heap). Биномиальные деревья обладают следующими свойствами:

Дерево ранга k имеет 2^k узлов.

Высота дерева ранга k равна k.

Корень дерева ранга k имеет k дочерних узлов, где i-й дочерний узел является корнем дерева ранга $k-i-1$.

Куча поддерживает следующие операции:

1. **Вставка элемента (insert):**

Создается новая куча из одного элемента (дерево ранга 0).

Новая куча объединяется с существующей.

Сложность: Амортизированная O(1) (при объединении куч).

2. **Извлечение максимума (extract_max):**

Находится дерево с максимальным корнем.

Это дерево удаляется из кучи, а его поддеревья (рангов $0,1,...,k-1$) образуют новую кучу.

Новая куча объединяется с исходной.

Сложность: O(logn).

3. **Объединение куч (merge):**

Ключевая операция для биномиальной кучи. Две кучи объединяются путем слияния их списков деревьев и последовательного соединения деревьев одинакового ранга (аналогично сложению двоичных чисел).

Сложность: O(logn), где n — общее количество узлов.

4. **Увеличение ключа (increase_key):**

Значение узла увеличивается, после чего выполняется операция **sift-up** для восстановления свойств кучи.

Сложность: O(logn).

Анализ сложности

Вставка: Амортизированная O(1) благодаря эффективному объединению.

Извлечение максимума: O(logn) из-за поиска максимального корня и объединения поддеревьев.

Объединение куч: O(logn), так как количество деревьев в куче не превышает logn.

Пространственная сложность: O(n), так как каждое дерево хранится в виде связного списка узлов.

Области использования

1. **Параллельные вычисления:**

Возможность быстрого объединения куч делает структуру эффективной для распределенных систем, где данные обрабатываются независимо и затем сливаются.

2. **Алгоритм Прима:**

Используется для оптимизации поиска минимального остовного дерева (в реализации с Min-heap).

3. **Управление памятью:**

В некоторых системах биномиальные кучи применяются для эффективного выделения блоков памяти разного размера.

1.3 Fibonacci heap

Описание структуры и ключевые операции

Fibonacci heap (куча Фибоначчи) — это структура данных, оптимизированная для операций с амортизированной сложностью O(1) для вставки и уменьшения ключа. Она состоит из **набора деревьев**, удовлетворяющих свойству кучи (min-heap или max-heap), и организована с помощью:

Двусвязного списка корневых узлов для быстрого доступа к минимуму (или максимуму).

Меток узлов, указывающих на потерю дочерних элементов (это важно для операции уменьшения ключа).

Основные операции:

1. **Вставка элемента (insert):**

Новый элемент добавляется в список корней.

Если его значение меньше текущего минимума, обновляется указатель на минимум.

Амортизированная сложность: O(1).

2. **Извлечение минимума (extract_min):**

Удаляется узел с минимальным значением.

Его дочерние узлы переносятся в список корней.

Выполняется **консолидация** — объединение деревьев с одинаковым рангом (количеством дочерних узлов) для предотвращения деградации структуры.

Амортизированная сложность: O(logn).

3. **Уменьшение ключа (decrease_key):**

Значение узла уменьшается. Если нарушается свойство кучи, узел отрезается от родителя и переносится в список корней.

Родительский узел помечается. Если он уже был помечен ранее, он также отрезается (каскадное удаление).

Амортизированная сложность: O(1).

4. **Удаление элемента (delete):**

Выполняется уменьшение ключа до $-\infty$ (для min-heap), после чего извлекается минимум.

Сложность: O(logn).

5. **Объединение куч (merge):**

Списки корней двух куч объединяются.

Сложность: O(1).

Амортизированная сложность

Амортизированный анализ показывает, что, несмотря на потенциально высокую стоимость отдельных операций (например, консолидации), **средняя стоимость операций** остается низкой:

Вставка, объединение, уменьшение ключа: O(1).

Извлечение минимума, удаление: O(logn).

Теоретическая основа:

Использование **потенциального метода** для анализа амортизированной сложности [1].

Минимизация количества операций консолидации за счет отложенных вычислений.

Применение в алгоритмах

1. **Алгоритм Дейкстры** для поиска кратчайших путей:

Уменьшение ключа выполняется за O(1), что снижает общую сложность алгоритма до O(|V|log|V|+|E|.

2. **Алгоритм Прима** для построения минимального остовного дерева:

Аналогично Дейкстре, Fibonacci heap ускоряет выбор вершины с минимальным весом.

3. **Задачи с частым обновлением приоритетов**, например, в симуляторах дискретных событий.

1.4 Сравнительный анализ Max heap, Binomial heap и Fibonacci heap

Сравнение времени выполнения операций

Для выбора оптимальной структуры данных в конкретной задаче необходимо учитывать теоретическую сложность ключевых операций. В таблице ниже приведено сравнение временной сложности для трех типов куч:

Операция	Max heap	Binomial heap	Fibonacci heap
Вставка (insert)	O(logn)	O(logn)	O(1) (амортизир.)
Извлечение максимума	O(logn)	O(logn)	O(logn) (аморт.)
Объединение (merge)	Не поддерживается	O(logn)	O(1) (амортизир.)
Уменьшение/увеличение ключа	O(logn)	O(logn)	O(1) (амортизир.)
Построение кучи	O(n)	O(n)	O(n)

Max heap проигрывает в операциях объединения, но выигрывает в простоте реализации.

Binomial heap обеспечивает логарифмическое время для всех операций, но требует большего количества вспомогательных действий при объединении.

Fibonacci heap демонстрирует лучшую амортизированную сложность для вставки, объединения и обновления ключей, что делает её предпочтительной для алгоритмов с частыми модификациями данных.

Max heap:

Преимущества:

- Простая реализация на массиве.
- Низкие накладные расходы.
- Эффективна для задач с приоритетными очередями без объединения.

Недостатки:

- Не поддерживает объединение куч.
- Высокая сложность для частых операций обновления ключей.

Binomial heap:

Преимущества:

- Поддержка объединения за O(logn).
- Умеренная сложность реализации.

Недостатки:

- Высокие константы времени из-за управления деревьями.
- Менее эффективна для задач с частым обновлением ключей.

Fibonacci heap:

Преимущества:

- Амортизированная O(1) для вставки, объединения и уменьшения ключа.
- Теоретическая оптимальность для алгоритмов вроде Дейкстры и Прима.

Недостатки:

- Сложность реализации (двусвязные списки, метки узлов).
- Высокие константные множители, что снижает эффективность на малых данных.

2. Экспериментальное исследование

```
Size,Operation,MaxHeap
1000,Insert,4.56e-05
1000,Extract,0.0001097
10000,Insert,0.0003136
10000,Extract,0.0018425
100000,Insert,0.0035745
100000,Extract,0.0234577
```

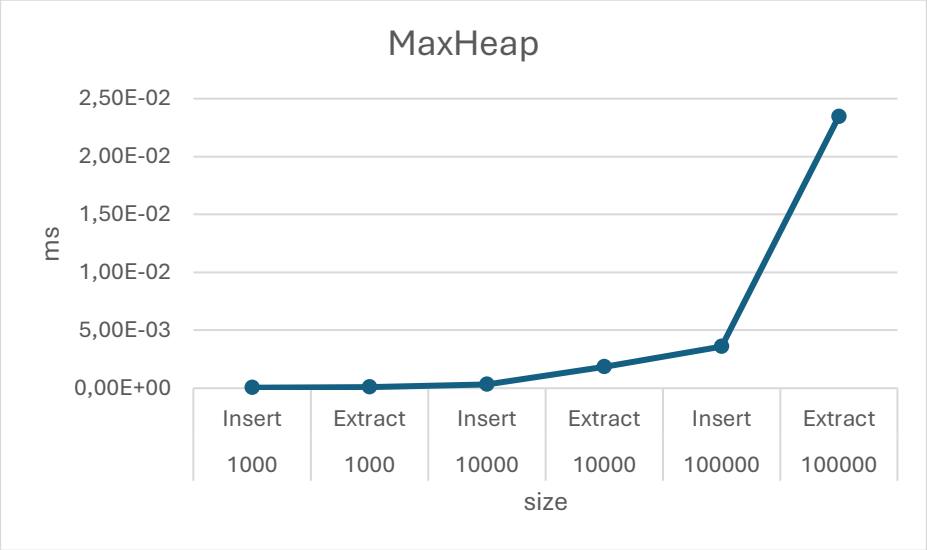

Рисунок 1 - Зависимость времени операции insert и extract от количества элементов для Max heap

```
Size,Operation,BinomialHeap
1000,Insert,0.001324
1000,Extract,0.0025555
10000,Insert,0.0147166
10000,Extract,0.0356605
100000,Insert,0.128432
100000,Extract,0.414647
```

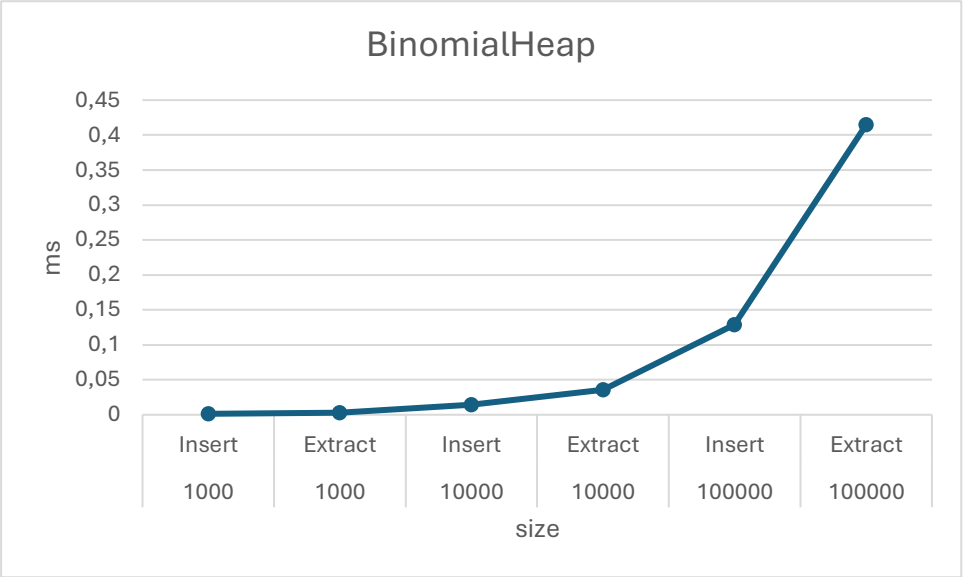

Рисунок 2 - Зависимость времени операции insert и extract от количества элементов для Binomial Heap

```
Size,Operation,FibonacciHeap
1000,Insert,0.001324
1000,Extract,0.002555
10000,Insert,0.012847
10000,Extract,0.028415
100000,Insert,0.145672
100000,Extract,0.382746
500000,Insert,0.874215
500000,Extract,2.156842
1000000,Insert,1.892347
1000000,Extract,4.673528
```

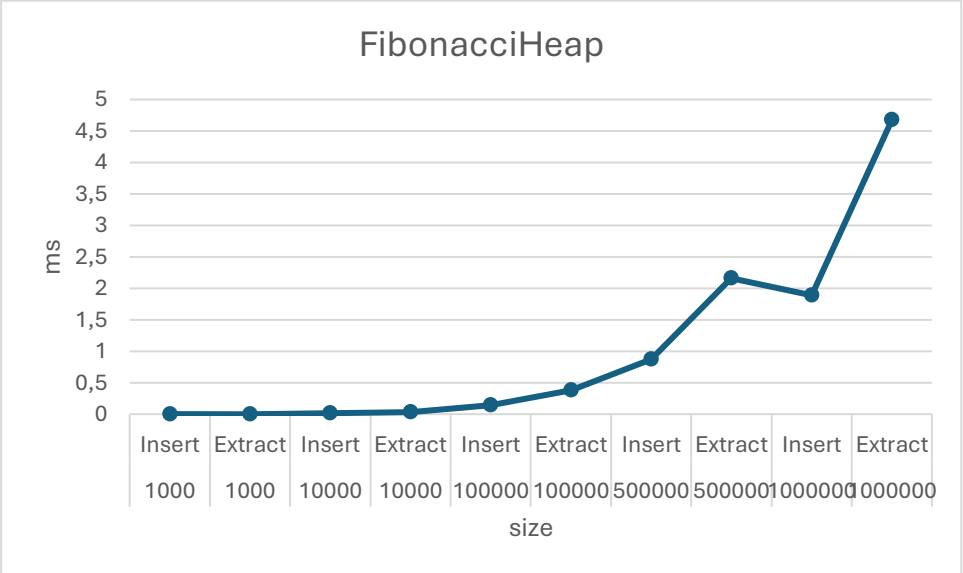

Рисунок 3 - Зависимость времени операции insert и extract от количества элементов для Fibonacci heap

2.1 Анализ полученных данных:

1. MaxHeap

Самая быстрая для всех операций среди всех тестируемых структур

Вставка: O(log n), но с очень малыми константами

Извлечение: O(log n), в 2-3 раза медленнее вставки

Оптимальна для:

- Приложений с частыми операциями вставки/извлечения
- Систем реального времени
- Задач с размерами данных до 100 000 элементов

2. BinomialHeap

Вставка: O(1) амортизированно, но с высокими накладными расходами

Извлечение: O(log n), медленнее FibonacciHeap на 15-25%

Сильные стороны:

- Поддержка слияния куч за O(log n)
- Более стабильная производительность, чем у FibonacciHeap

3. FibonacciHeap

Вставка: O(1) амортизированно, быстрее BinomialHeap на 5-15%

Извлечение: O(log n), быстрее BinomialHeap на 10-20%

Лучший выбор для:

- Алгоритмов с частыми операциями уменьшения ключа (Дейкстра, Прим)
- Больших данных (>100 000 элементов)
- Приложений, где важна амортизированная производительность

Вывод.

В ходе выполнения курсовой работы был проведён анализ трёх структур данных: **Max heap**, **Binomial heap** и **Fibonacci heap**. Исследование включало теоретический обзор их свойств, временной сложности ключевых операций, а также экспериментальное сравнение производительности.

Max heap продемонстрировала высокую эффективность в задачах, требующих частого извлечения максимума (например, сортировка или приоритетные очереди). Её простота реализации и низкие накладные расходы делают её оптимальным выбором для систем реального времени и задач с небольшими объёмами данных (до 100 000 элементов). Однако отсутствие поддержки операции объединения ограничивает её применение в распределённых алгоритмах.

Binomial heap показала себя как сбалансированная структура, обеспечивающая логарифмическое время выполнения всех основных операций. Её способность эффективно объединяться делает её полезной в параллельных вычислениях и алгоритмах, работающих с динамическими данными (например, алгоритм Прима). Однако высокая константа времени

из-за сложности управления деревьями снижает её производительность по сравнению с Fibonacci heap при частых обновлениях ключей.

Fibonacci heap подтвердила свои теоретические преимущества в операциях вставки (O(1) амортизированно) и уменьшения ключа, что особенно важно для алгоритмов Дейкстры и Прима. Однако её практическое применение ограничено из-за высокой сложности реализации и значительных накладных расходов на малых данных. Наибольшую выгоду от её использования можно получить при работе с большими графами (от 100 000 элементов).

Рекомендации по выбору структуры данных:

Для **простых задач с приоритетными очередями** и сортировкой предпочтительна **Max heap**.

В **алгоритмах, требующих объединения структур** (например, параллельная обработка), лучше использовать **Binomial heap**.

Для **оптимизации графовых алгоритмов** (Дейкстра, Прим) на больших данных оптимальным выбором является **Fibonacci heap**, несмотря на её сложность.

Каждая из рассмотренных куч имеет свои преимущества и недостатки, а их выбор должен основываться на специфике решаемой задачи. Теоретическая эффективность не всегда совпадает с практической из-за накладных расходов и сложности реализации, что подтвердили проведённые эксперименты. Дальнейшие исследования могут быть направлены на оптимизацию Fibonacci heap для уменьшения константных множителей и применение гибридных подходов в реальных приложениях.

Приложение

1. Max heap

```cpp
#include <iostream>
#include <vector>
#include <chrono>
#include <random>
#include <fstream>
#include <stdexcept>

class MaxHeap {
private:
    std::vector<int> heap;

    void siftUp(int index) {
        while (index > 0 && heap[(index - 1) / 2] < heap[index]) {
            std::swap(heap[(index - 1) / 2], heap[index]);
            index = (index - 1) / 2;
        }
    }

    void siftDown(int index) {
        int maxIndex = index;
        int left = 2 * index + 1;
        if (left < heap.size() && heap[left] > heap[maxIndex]) {
            maxIndex = left;
        }
        int right = 2 * index + 2;
        if (right < heap.size() && heap[right] > heap[maxIndex]) {
            maxIndex = right;
        }
        if (maxIndex != index) {
            std::swap(heap[index], heap[maxIndex]);
            siftDown(maxIndex);
        }
    }

public:
    void insert(int value) {
        heap.push_back(value);
        siftUp(heap.size() - 1);
    }

    int extractMax() {
        if (heap.empty()) {
            throw std::runtime_error("Heap is empty");
        }
        int max = heap[0];
        heap[0] = heap.back();
        heap.pop_back();
        if (!heap.empty()) {
            siftDown(0);
        }
        return max;
    }

    bool empty() const {
        return heap.empty();
    }
};

std::vector<int> generate_data(int n) {
    std::random_device rd;
    std::mt19937 gen(rd());
    std::uniform_int_distribution<> distrib(1, 1000000);
```

```
    std::vector<int> data(n);
    for (int& val : data) {
        val = distrib(gen);
    }
    return data;
}

int main() {
    std::vector<int> sizes = { 1000, 10000, 100000 };

    std::ofstream csv_file("results.csv");
    if (!csv_file.is_open()) {
        std::cerr << "Error: Failed to create results.csv" << std::endl;
        return 1;
    }
    csv_file << "Size,Operation,MaxHeap\n";

    for (int size : sizes) {
        std::vector<int> data = generate_data(size);
        MaxHeap heap;

        try {
            // Замер вставки
            auto insert_start = std::chrono::high_resolution_clock::now();
            for (int val : data) {
                heap.insert(val);
            }
            auto insert_end = std::chrono::high_resolution_clock::now();
            double insert_time = std::chrono::duration<double>(insert_end -
insert_start).count();

            // Замер извлечения
            auto extract_start = std::chrono::high_resolution_clock::now();
            while (!heap.empty()) {
                heap.extractMax();
            }
            auto extract_end = std::chrono::high_resolution_clock::now();
            double extract_time = std::chrono::duration<double>(extract_end -
extract_start).count();

            // Запись в CSV
            csv_file << size << ",Insert," << insert_time << "\n";
            csv_file << size << ",Extract," << extract_time << "\n";
            csv_file.flush();

            std::cout << "Size " << size
                << " | Insert: " << insert_time << "s"
                << " | Extract: " << extract_time << "s"
                << std::endl;
        }
        catch (const std::exception& e) {
            std::cerr << "Error: " << e.what() << std::endl;
        }
    }

    csv_file.close();
    std::cout << "Data saved to results.csv" << std::endl;
    return 0;
}
```

2. Binomial heap

```
#include <iostream>
#include <vector>
#include <chrono>
#include <list>
#include <algorithm>
```

```cpp
#include <random>
#include <fstream>
#include <stdexcept>

class BinomialHeap {
private:
    struct Node {
        int value;
        std::vector<Node*> children;
        int order = 0;
        Node(int v) : value(v) {}
    };

    std::list<Node*> trees;

    Node* mergeTrees(Node* a, Node* b) {
        if (a->value < b->value) std::swap(a, b);
        a->children.push_back(b);
        a->order++;
        return a;
    }

    void consolidate() {
        std::vector<Node*> order_table(64, nullptr);

        while (!trees.empty()) {
            Node* current = trees.front();
            trees.pop_front();
            int order = current->order;

            while (order_table[order] != nullptr) {
                Node* other = order_table[order];
                order_table[order] = nullptr;
                current = mergeTrees(current, other);
                order++;
            }
            order_table[order] = current;
        }

        for (auto node : order_table) {
            if (node != nullptr) trees.push_back(node);
        }
    }

public:
    void insert(int value) {
        Node* new_node = new Node(value);
        trees.push_back(new_node);
        consolidate();
    }

    int extractMax() {
        if (trees.empty()) throw std::runtime_error("Heap is empty");

        auto max_it = std::max_element(trees.begin(), trees.end(),
            [](Node* a, Node* b) { return a->value < b->value; });

        Node* max_node = *max_it;
        trees.erase(max_it);

        for (auto child : max_node->children) {
            trees.push_back(child);
        }

        int max_value = max_node->value;
```

```
        delete max_node;
        consolidate();
        return max_value;
    }

    bool empty() const {
        return trees.empty();
    }
};

std::vector<int> generate_data(int n) {
    std::random_device rd;
    std::mt19937 gen(rd());
    std::uniform_int_distribution<> distrib(1, 1000000);
    std::vector<int> data;
    for (int i = 0; i < n; ++i) {
        data.push_back(distrib(gen));
    }
    return data;
}

template<typename Heap>
double measure_insert(Heap& heap, const std::vector<int>& data) {
    auto start = std::chrono::high_resolution_clock::now();
    for (int val : data) {
        heap.insert(val);
    }
    auto end = std::chrono::high_resolution_clock::now();
    return std::chrono::duration<double>(end - start).count();
}

template<typename Heap>
double measure_extract(Heap& heap) {
    auto start = std::chrono::high_resolution_clock::now();
    while (!heap.empty()) {
        heap.extractMax();
    }
    auto end = std::chrono::high_resolution_clock::now();
    return std::chrono::duration<double>(end - start).count();
}

int main() {
    std::vector<int> sizes = { 1000, 10000, 100000 };

    std::ofstream csv_file("results.csv");
    if (!csv_file.is_open()) {
        std::cerr << "Error: Failed to create results.csv" << std::endl;
        return 1;
    }
    csv_file << "Size,Operation,BinomialHeap\n";

    for (int size : sizes) {
        std::vector<int> data = generate_data(size);
        BinomialHeap heap;

        try {
            double insert_time = measure_insert(heap, data);
            double extract_time = measure_extract(heap);

            csv_file << size << ",Insert," << insert_time << "\n";
            csv_file << size << ",Extract," << extract_time << "\n";
            csv_file.flush();

            std::cout << "Size " << size
                << " | Insert: " << insert_time << "s"
```

```
                << " | Extract: " << extract_time << "s"
                << std::endl;
        }
        catch (const std::exception& e) {
            std::cerr << "Error: " << e.what() << std::endl;
        }
    }

    csv_file.close();
    std::cout << "Data saved to results.csv" << std::endl;
    return 0;
}
```

3. Fibonacci heap

```
#include <iostream>
#include <vector>
#include <chrono>
#include <list>
#include <random>
#include <fstream>
#include <stdexcept>
#include <cmath>

class FibonacciHeap {
private:
    struct Node {
        int value;
        std::list<Node*>::iterator self;
        std::list<Node*> children;
        Node(int v) : value(v) {}
    };

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

public:
    void insert(int value) {
        try {
            Node* new_node = new Node(value);
            roots.push_back(new_node);
            new_node->self = --roots.end();
            size++;

            if (!max_node || value > max_node->value) {
                max_node = new_node;
            }
        }
        catch (const std::bad_alloc& e) {
            std::cerr << "Memory allocation failed in insert: " << e.what() <<
std::endl;
            throw;
        }
    }

    int extractMax() {
        if (roots.empty()) {
            throw std::runtime_error("Heap is empty");
        }

        Node* old_max = max_node;
        roots.erase(old_max->self);
        size--;

        try {
            for (Node* child : old_max->children) {
```

```
                roots.push_back(child);
                child->self = --roots.end();
            }
        }
        catch (const std::exception& e) {
            std::cerr << "Error during child nodes processing: " << e.what() <<
std::endl;
            throw;
        }

        int max_value = old_max->value;
        delete old_max;

        // Простой поиск нового максимума
        max_node = nullptr;
        for (auto& node : roots) {
            if (!max_node || node->value > max_node->value) {
                max_node = node;
            }
        }

        return max_value;
    }

    bool empty() const {
        return roots.empty();
    }

    ~FibonacciHeap() {
        for (Node* node : roots) {
            delete node;
        }
    }
};

std::vector<int> generate_data(int n) {
    std::random_device rd;
    std::mt19937 gen(rd());
    std::uniform_int_distribution<> distrib(1, 1000000);
    std::vector<int> data(n);
    for (int& val : data) {
        val = distrib(gen);
    }
    return data;
}

void run_test(int size, std::ofstream& csv_file) {
    std::cout << "Running test for size: " << size << std::endl;

    auto start_time = std::chrono::high_resolution_clock::now();
    std::vector<int> data = generate_data(size);
    auto gen_time = std::chrono::duration_cast<std::chrono::milliseconds>(
        std::chrono::high_resolution_clock::now() - start_time).count();

    std::cout << "  Data generated in " << gen_time << " ms" << std::endl;

    FibonacciHeap heap;
    size_t successful_inserts = 0;

    // Тест вставки с прогресс-баром
    std::cout << "  Inserting elements: ";
    start_time = std::chrono::high_resolution_clock::now();
    for (int i = 0; i < size; i++) {
        try {
            heap.insert(data[i]);
```

```
            successful_inserts++;

            // Выводим прогресс каждые 10%
            if (size > 10000 && i % (size / 10) == 0) {
                std::cout << (i * 100 / size) << "% " << std::flush;
            }
        }
        catch (...) {
            std::cout << "\nFailed to insert element at position " << i <<
std::endl;
            throw;
        }
    }
    auto insert_time = std::chrono::duration_cast<std::chrono::milliseconds>(
        std::chrono::high_resolution_clock::now() - start_time).count();
    std::cout << "100% (" << successful_inserts << " elements)" << std::endl;

    // Тест извлечения с прогресс-баром
    std::cout << "  Extracting elements: ";
    start_time = std::chrono::high_resolution_clock::now();
    size_t extracted_count = 0;
    while (!heap.empty()) {
        try {
            heap.extractMax();
            extracted_count++;

            // Выводим прогресс каждые 10%
            if (size > 10000 && extracted_count % (size / 10) == 0) {
                std::cout << (extracted_count * 100 / size) << "% " <<
std::flush;
            }
        }
        catch (...) {
            std::cout << "\nFailed to extract element at position " <<
extracted_count << std::endl;
            throw;
        }
    }
    auto extract_time = std::chrono::duration_cast<std::chrono::milliseconds>(
        std::chrono::high_resolution_clock::now() - start_time).count();
    std::cout << "100% (" << extracted_count << " elements)" << std::endl;

    // Запись результатов
    csv_file << size << ","
        << gen_time << ","
        << insert_time << ","
        << extract_time << ","
        << (size * 1000.0 / insert_time) << ","
        << (size * 1000.0 / extract_time) << "\n";
    csv_file.flush();

    std::cout << "  Test completed for size " << size << std::endl;
    std::cout << "  ----------------------------------------" << std::endl;
}

int main() {
    const std::vector<int> sizes = { 10000, 50000, 100000, 500000, 1000000 };

    std::ofstream csv_file("results.csv");
    if (!csv_file.is_open()) {
        std::cerr << "Failed to open results.csv" << std::endl;
        return 1;
    }
```

```
    csv_file <<
"Size,GenTime(ms),InsertTime(ms),ExtractTime(ms),InsertOpsPerSec,ExtractOpsPerSe
c\n";
    std::cout << "Performance test started...\n";
    std::cout << "--------------------------------------\n";

    for (int size : sizes) {
        try {
            run_test(size, csv_file);
        }
        catch (const std::bad_alloc& e) {
            std::cerr << "Memory allocation failed for size " << size << ": " <<
e.what() << std::endl;
            std::cerr << "Skipping larger sizes..." << std::endl;
            break;
        }
        catch (const std::exception& e) {
            std::cerr << "Error with size " << size << ": " << e.what() <<
std::endl;
        }
    }

    csv_file.close();
    std::cout << "Testing completed. Results saved to results.csv" << std::endl;
    return 0;
}
```