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 Полковникова Д.Д.

Ссылка на репозиторий: https://github.com/MUCTR-IKT-CPP/Polkovnikova\_CS-36

Приняли: Пысин Максим Дмитриевич

Краснов Дмитрий Олегович

Лобанов Алексей Владимирович

Крашенинников Роман Сергеевич

Дата сдачи: 21.04.2025

Оглавление

[Описание задачи. 2](#_Toc195526846)

[Описание метода/модели 3](#_Toc195526847)

[Выполнение задачи. 6](#_Toc195526848)

[Заключение. 16](#_Toc195526849)

# Описание задачи.

В рамках лабораторной работы необходимо реализовать бинарную кучу (мин или макс), а также 1 из ниже приведенных структур куч:

1. Фибоначчиеву кучу
2. Биноминальную 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 рассчитывать 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

**Куча –** это особая структура, которая является деревом удовлетворяющее основному правилу кучи: все узлы потомки текущего узла по значению ключа меньше, чем значение ключа текущего узла.

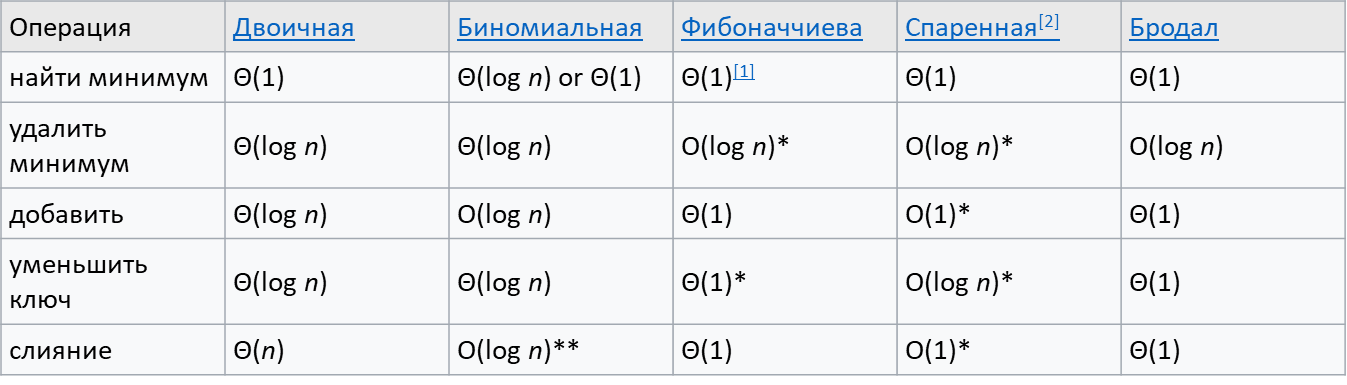
**Над кучами обычно проводятся следующие операции:**

* найти максимум или найти минимум: найти максимальный элемент в max-куче или минимальный элемент в min-куче, соответственно
* удалить максимум или удалить минимум: удалить корневой узел в max- или min-куче, соответственно
* увеличить ключ или уменьшить ключ: обновить ключ в max- или min-куче, соответственно
* добавить: добавление нового ключа в кучу.
* слияние: соединение двух куч с целью создания новой кучи, содержащей все элементы обеих исходных.

**Виды куч:**

* 2-3 куча
* Двуродительская куча
* Двоичная куча
* Биномиальная куча
* Очередь Бродала
* Куча с D потомками
* Фибоначчиева куча
* Куча с приоритетом самого левого
* Спаренная куча
* Асимметричная куча
* Мягкая куча
* Тернарная куча
* Декартово дерево

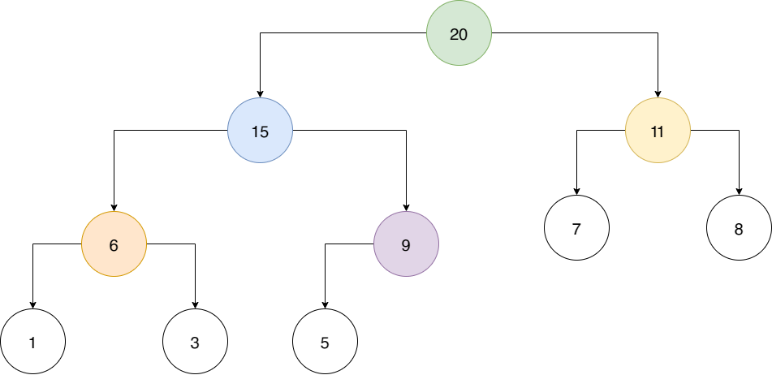
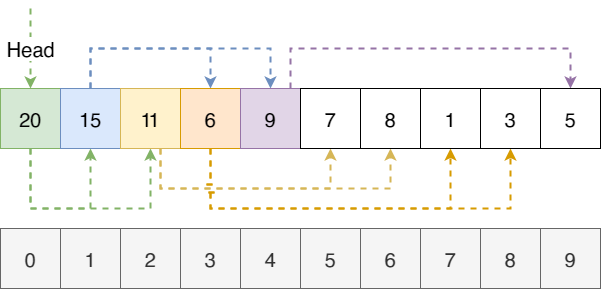
**Сложность операций**



Двоичная куча (binary heap) – просто реализуемая структура данных, позволяющая быстро (за логарифмическое время) добавлять элементы и извлекать элемент с максимальным приоритетом (например, максимальный по значению).

Основной особенностью двоичной кучи является то, что каждый из узлов кучи не может иметь более чем двух потомков.

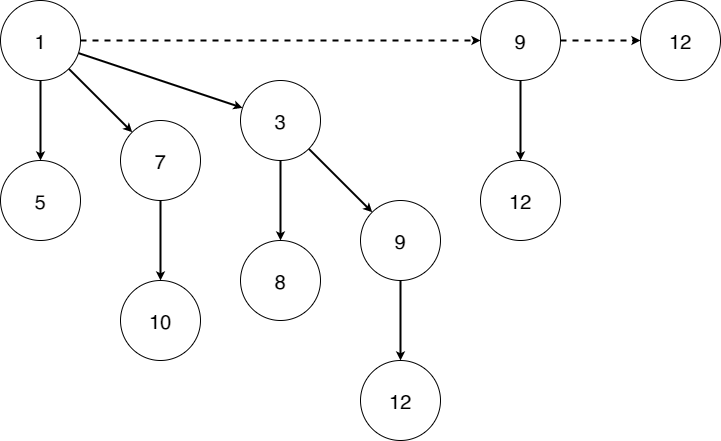
Двоичная куча отлично представляется в виде одномерного массива, при этом: нулевой элемент массива всегда является вершиной кучи, а первый и второй потомок вершины с индексом i получают свои положения на основании формул: 2 \* i + 1 левый, 2 \* i + 2 правый.



**Биноминальное дерево** – это такая структура данных, которая задает саму себя рекурсивно через свой предыдущий шаг.

**Биноминальное куча** – это биноминальное дерево, которое подчиняется правилу кучи, т.е. любой родитель всегда больше любого его ребенка.

При этом биноминальное дерево всегда содержит в себе 2^k вершин для дерева BK. Т.е. если у нас имеется элементов меньше или больше, чем некое 2^k, мы не можем использовать одно биноминальное дерево и нам нужно разложить количество вершин так, чтобы оно состояло из суммы 2^l(i). Важной особенностью биноминальной кучи является то, что она не должна содержать в себе деревьев одного порядка.



**Фибоначчиева куча**

**Фибоначчиева куча –** это так же биноминальная куча, у которой прощаются все операции, кроме операции удаления. За счет сознательного ухудшения состояния кучи внутри нее. Мы предполагаем, что наша куча может иметь в себе множество деревьев одинакового ранга. Т.е. у нас может быть хоть 20 деревьев первого нулевого ранга.

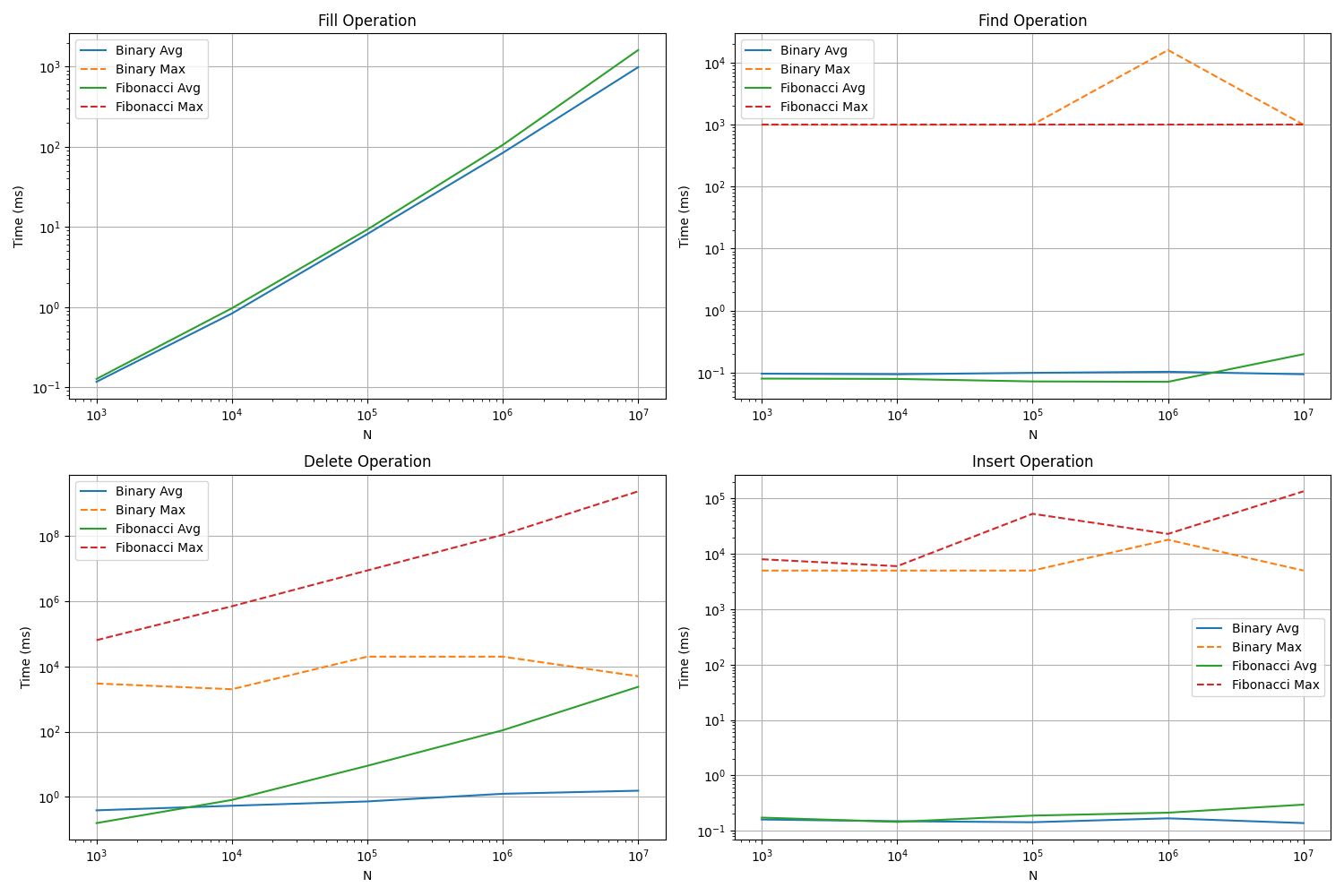
В такой ситуации мы можем позволить себе при добавлении в дерево нового элемента просто добавлять в кучу дерево ранга 0 содержащего наше новое значение, перепроверяя только минимум/максимум в куче (в зависимости от того в какую сторону мы взращиваем кучу).

Объединение двух куч так же позволяет нам совершать эту операцию просто выполняя объединение двух списков деревьев, из которых состоят кучи, в конце лишь выбирая один из двух имеющихся минимумов.

Данные действия сознательно ухудшат структуру и значит, что за это придется платить, основным взимателем этой платы является операция удаления минимума, которая проводит нормализацию дерева объединяя все созданные дубликаты деревьев.

# Выполнение задачи.

В данной лабораторной работе получили следующие графики:



**Общая цель кода**

Код реализует две структуры данных — **бинарную мин-кучу** и **Фибоначчиеву кучу** — и проводит тесты их производительности. Он:

1. Заполняет каждую кучу определенным количеством элементов (N = 10³, 10⁴, 10⁵, 10⁶, 10⁷).
2. Выполняет 1000 операций поиска минимума, удаления минимума и вставки новых элементов, замеряя время выполнения.
3. Сохраняет среднее и максимальное время операций в CSV-файл для последующего построения графиков.
4. Позволяет визуализировать результаты с помощью Python-скрипта.

**Основные компоненты кода**

**1. Бинарная мин-куча (BinaryMinHeap)**

Бинарная мин-куча — это структура данных в виде полного бинарного дерева, где каждый узел меньше или равен своим потомкам. В коде она реализована с помощью массива (std::vector<int>), где:

Элемент с индексом i имеет родителя с индексом (i-1)/2, левого ребенка с индексом 2\*i+1 и правого ребенка с индексом 2\*i+2.

**Операции:**

**Вставка (insert)**:

* + Добавляет новый элемент в конец массива.
  + Поднимает элемент вверх по дереву, сравнивая с родителем и меняя местами, если родитель больше, пока элемент не окажется на правильной позиции.

**Поиск минимума (findMin)**:

* + Возвращает первый элемент массива (корень дерева), так как он всегда минимальный.

**Удаление минимума (deleteMin)**:

* + Заменяет корень последним элементом массива.
  + Удаляет последний элемент.
  + "Просеивает" новый корень вниз, сравнивая с детьми и меняя местами с меньшим из них, пока структура кучи не восстановится.

**2. Фибоначчиева куча (FibonacciHeap)**

Фибоначчиева куча — более сложная структура, состоящая из набора деревьев, где каждое дерево удовлетворяет свойству мин-кучи (родитель меньше детей). Узлы связаны в двусвязный циклический список, а каждое дерево может иметь произвольное количество детей. В коде узлы представлены структурой Node, содержащей значение, указатели на родителя, детей, соседей и дополнительные поля (степень и метка).

**Операции:**

**Вставка (insert)**:

* + Создает новый узел с заданным значением.
  + Добавляет его в циклический список корней кучи.
  + Если новый узел меньше текущего минимального, обновляет указатель на минимум.

**Поиск минимума (findMin)**:

* + Возвращает значение узла, на который указывает min (указатель на минимальный элемент).

**Удаление минимума (deleteMin)**:

* + Если есть дети минимального узла, добавляет их в список корней, отключая их от родителя.
  + Удаляет минимальный узел из списка корней.
  + Если список корней не пуст, выбирает новый минимальный узел и выполняет **консолидацию**:
    - Объединяет деревья с одинаковым количеством детей (степенью) в одно, пока каждое количество детей не станет уникальным.
    - Обновляет указатель на минимум, проверяя все корни.
  + Освобождает память удаленного узла.

**Особенности:**

* Фибоначчиева куча оптимизирована для быстрой вставки и поиска минимума, но удаление минимума сложнее из-за консолидации.
* Деструктор освобождает память, рекурсивно удаляя все узлы.

**3. Тестирование производительности (testHeap)**

Функция testHeap тестирует производительность кучи (бинарной или Фибоначчиевой) для заданного размера N. Она:

**Заполняет кучу**:

* + Генерирует N случайных чисел (от 1 до 1,000,000) и вставляет их в кучу.
  + Замеряет общее время заполнения.

**Тестирует операции**:

* + Выполняет 1000 операций поиска минимума, удаления минимума и вставки новых случайных элементов.
  + Для каждой операции:
    - Замеряет общее время выполнения 1000 операций.
    - Замеряет время каждой отдельной операции и сохраняет максимальное время (чтобы поймать моменты "деградации" структуры, например, при консолидации в Фибоначчиевой куче).

**Сохраняет результаты**:

* + Записывает в CSV-файл (heap\_results.csv) данные: тип кучи, размер N, тип операции, среднее время одной операции (в микросекундах) и максимальное время одной операции (в наносекундах).

**4. Основная программа (main)**

Создает CSV-файл с заголовком: HeapType,N,Operation,AvgTime,MaxTime.

Перебирает размеры кучи N = 10³, 10⁴, 10⁵, 10⁶, 10⁷.

Для каждого N вызывает testHeap для бинарной и Фибоначчиевой кучи.

Выводит сообщение о завершении и сохранении результатов.

**5. Python-скрипт для графиков (plot\_results.py)**

Читает данные из heap\_results.csv.

Для каждой операции (заполнение, поиск, удаление, вставка) строит график:

* + Ось X: размер кучи N (логарифмическая шкала).
  + Ось Y: время выполнения (среднее или максимальное, логарифмическая шкала).
  + Линии показывают среднее и максимальное время для каждой кучи (бинарной и Фибоначчиевой).

Сохраняет графики в файл heap\_performance.png.

# Заключение.

В ходе изучения бинарных и фибоначчиевых куч были выявлены их ключевые сходства, различия, преимущества и недостатки, а также области применения.

**Сходства:**

1. Обе структуры данных относятся к **приоритетным очередям** и поддерживают операции:
   * insert (вставка),
   * extract-min (извлечение минимума),
   * decrease-key (уменьшение ключа),
   * merge (слияние куч).
2. Основаны на **кучеобразной** (частично упорядоченной) структуре.
3. Используются в алгоритмах, где важна эффективность операций с приоритетами (например, алгоритм Дейкстры, алгоритм Прима).

**Различия:**

| **Критерий** | **Бинарная куча** | **Фибоначчиева куча** |
| --- | --- | --- |
| **Структура** | Полное бинарное дерево | Набор деревьев (возможно, несбалансированных) |
| **Сложность операции** | insert: O(log n) | insert: O(1) (амортизированно) |
|  | extract-min: O(log n) | extract-min:O(log n)  (амортизированно) |
|  | decrease-key: O(log n) | decrease-key: O(1) (амортизированно) |
|  | merge: O(n) | merge: O(1) |
| **Память** | Более простая структура, меньше накладных расходов | Сложнее, требует больше памяти из-за списков детей и пометок |
| **Эффективность** | Хуже при частых decrease-key и merge | Лучше в алгоритмах с большим количеством таких операций |

**Области применения:**

* **Бинарные кучи** используются в стандартных реализациях priority\_queue (например, в C++ std::priority\_queue), алгоритмах сортировки (пирамидальная сортировка), а также в алгоритмах, где не требуется частое изменение ключей.
* **Фибоначчиевы кучи** применяются в алгоритмах, где критически важны операции decrease-key и merge, например:
  + Алгоритм Дейкстры для разреженных графов (уменьшение ключа происходит часто).
  + Алгоритм Прима для минимального остовного дерева.
  + Теоретически оптимальные алгоритмы, где важна амортизированная сложность.

**Вывод:**

Бинарные кучи проще и эффективнее для большинства стандартных задач, тогда как фибоначчиевы кучи имеют теоретическое преимущество в сложности операций, но на практике их применение оправдано только в специфических сценариях (например, при работе с очень большими графами). Выбор между ними зависит от конкретной задачи и требований к производительности.