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

Вариант 2

Выполнила студентка группы .....КС-33..... Георгиевская Анастасия Игоревна
Ссылка на репозиторий: .....
https://github.com/MUCTR-IKT-CPP/GeorgievskayaAA_33_alg/tree/main/lab%208

Приняли: .....Пысин Максим Дмитриевич
.....Лобанов Алексей Владимирович

Дата сдачи: 02.04.2025

Оглавление

Описание задачи.

В рамках лабораторной работы необходимо реализовать бинарную кучу (мин или макс), а также биноминальную кучу.

Для реализованных куч выполнить следующие действия:

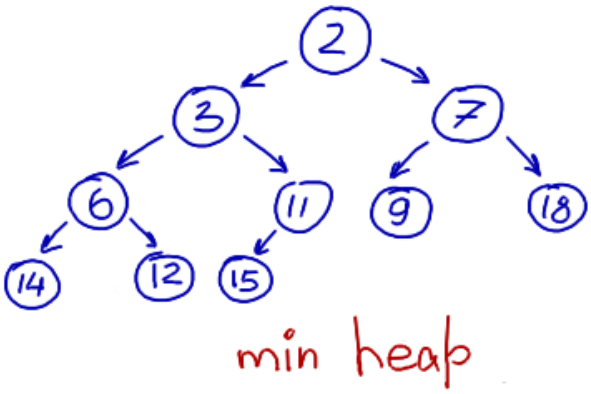
1. Наполнить кучу N кол-ва элементов (где N = 10 ^ i, i от 3 до 7).
2. После заполнения кучи необходимо провести следующие тесты:
    a. 1000 раз найти минимум/максимум
    b. 1000 раз удалить минимум/максимум
    c. 1000 раз добавить новый элемент в кучу

Для всех операция требуется замерить время на выполнения всей 1000 операций и рассчитать время на одну операцию, а также 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, чтобы 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 бинарной и биноминальной куч.

Бинарная куча — это двоичное дерево, которое является полным, то есть все уровни дерева (кроме последнего) полностью заполняются, а последний уровень заполняется слева направо.



Свойства бинарной кучи:

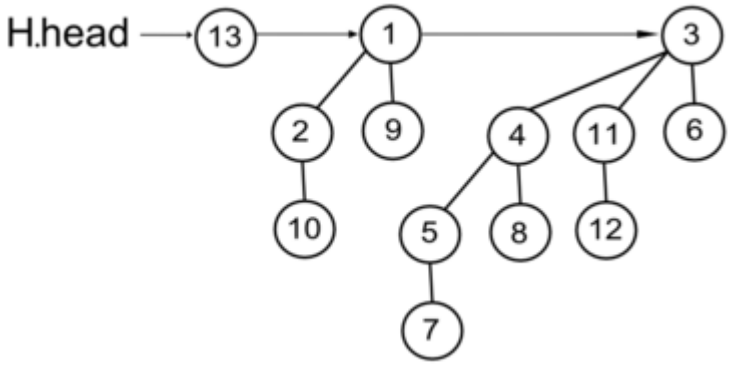
- **Мини-куча**: для каждого узла значение этого узла меньше или равно значениям его детей.
- Макси-куча: для каждого узла значение этого узла больше или равно значениям его детей.
- Это двоичное дерево, где каждый узел имеет не более двух детей.
- Время операций:
    - Вставка: O(log n)
    - Извлечение минимального (или максимального) элемента: O(log n)
    - Построение кучи из массива: O(n)

Основные операции:

1. **Вставка**: Добавление элемента в кучу и корректировка её свойств с помощью "просеивания вверх" (heapify-up).
2. **Извлечение корня** (минимум или максимум): Удаление корня кучи и "просеивание вниз" (heapify-down).
3. **Изменение приоритетов**: При изменении значения элемента выполняется корректировка с помощью просеивания вверх или вниз.

Бинарная куча используется в алгоритмах сортировки (например, в сортировке кучей) и в реализации приоритетных очередей.

Биномиальная куча — это более сложная структура данных, которая состоит из множества биномиальных деревьев, где каждое дерево является биномиальным деревом определенной степени.



Свойства биномиальной кучи:

- **Биномиальные деревья** — это деревья, которые следуют определенному паттерну: деревья степени k имеют 2^k узлов, и их структура напоминает двоичное дерево, но с определенными правилами для соединения узлов.
- Биномиальная куча состоит из нескольких таких деревьев, где каждая степень дерева встречается не более одного раза.

Основные операции:

1. **Вставка**: Вставка элемента в биномиальную кучу может быть выполнена слиянием двух куч за O(log n) с помощью операции слияния.
2. **Извлечение минимального элемента**: Эта операция требует поиска минимального элемента среди корней всех биномиальных деревьев и удаления его, что также занимает O(log n).
3. **Слияние куч**: Основное преимущество биномиальной кучи — эффективная операция слияния двух куч, которая выполняется за O(log n).

Преимущества биномиальной кучи:

- Более эффективное слияние двух куч по сравнению с бинарной кучей, так как операция слияния работает за O(log n), а не за O(n), как в бинарной куче.

Биномиальная куча используется в алгоритмах, где важна операция слияния куч, например, в алгоритмах для нахождения кратчайших путей (например, в алгоритме Дейкстры).

Сравнение бинарной и биномиальной кучи

Вставка: В бинарной куче операция вставки выполняется за O(log n), так же как и в биномиальной куче.

Извлечение минимального элемента: Операция извлечения минимального элемента также выполняется за O(log n) для обеих куч.

Слияние куч: Это основное различие. В бинарной куче слияние двух куч может потребовать O(n) времени, тогда как в биномиальной куче оно выполняется за O(log n).

Использование памяти: Биномиальная куча может использовать больше памяти, так как хранит несколько деревьев.

Выполнение задачи.

Для выполнения задачи лабораторной был использован язык C++. Реализация AVL-дерева была взята из предыдущей лабораторной работы. Поэтому здесь будет приведена реализация декартова дерева и основной функции

```
#include <iostream>
#include <vector>
#include <cmath>
#include <cstdlib>
#include <ctime>
#include <chrono>
#include <fstream>

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

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

    void heapifyDown(int index) {
        int size = heap.size();
        while (2 * index + 1 < size) {
            int leftChild = 2 * index + 1;
            int rightChild = 2 * index + 2;
            int smallest = index;

            if (leftChild < size && heap[leftChild] < heap[smallest]) {
                smallest = leftChild;
            }
            if (rightChild < size && heap[rightChild] < heap[smallest]) {
                smallest = rightChild;
            }

            if (smallest == index) {
                break;
            }

            std::swap(heap[index], heap[smallest]);
            index = smallest;
```

```
        }
    }

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

    int extractMin() {
        if (heap.empty()) {
            throw std::out_of_range("Heap is empty");
        }

        int minValue = heap[0];
        heap[0] = heap.back();
        heap.pop_back();
        heapifyDown(0);

        return minValue;
    }

    int getMin() {
        if (heap.empty()) {
            throw std::out_of_range("Heap is empty");
        }

        return heap[0];
    }

    bool isEmpty() {
        return heap.empty();
    }
};
```

- Класс BinaryHeap создает бинарную кучу
    - Функция heapifyUp
        - Вход: индекс элемента в куче, который нужно переместить вверх.
        - Работа: Эта функция проверяет, нужно ли переместить элемент вверх по куче. Она сравнивает значение элемента с его родительским элементом и меняет их местами, если элемент меньше родителя. Повторяет процесс до тех пор, пока элемент не окажется в правильном месте.
        - Выход: Нет. Измененное состояние кучи
    - Функция heapifyDown
        - Вход: индекс элемента в куче, который нужно переместить вниз.

- Работа: Функция проверяет, нужно ли переместить элемент вниз. Для этого она сравнивает элемент с его потомками и, если необходимо, меняет их местами, продолжая процесс, пока элемент не окажется на своем месте.
- Выход: Нет. Измененное состояние кучи

- Функция insert
  - Вход: значение, которое нужно добавить в кучу.
  - • Работа: Добавляет элемент в конец кучи и вызывает heapifyUp, чтобы переместить его на правильное место в куче.
  - Выход: Нет.
- Функция extractMin
  - Вход: Нет.
  - Работа: Удаляет минимальный элемент (корень кучи), заменяет его последним элементом в куче и вызывает heapifyDown, чтобы восстановить структуру кучи.
  - Выход: Возвращает минимальный элемент (корень кучи).
- Функция getMin
  - Вход: Нет.
  - Работа: Возвращает минимальный элемент (корень кучи), не изменяя кучу.
  - Выход: Минимальное значение в куче.
- Функция isEmpty
  - Вход: Нет.
  - Работа: Проверяет, пуста ли куча.
  - Выход: Возвращает true, если куча пуста, иначе false.

```
class BinomialHeap {
private:
    struct Node {
        int value;
        int degree;
        Node* parent;
        Node* child;
        Node* sibling;

        Node(int value) : value(value), degree(0), parent(nullptr), child(nullptr), sibling(nullptr) {}
    };

    Node* mergeTrees(Node* tree1, Node* tree2) {
        if (tree1->value > tree2->value) {
            std::swap(tree1, tree2);
        }
```

```
        tree2->parent = tree1;
        tree2->sibling = tree1->child;
        tree1->child = tree2;
        tree1->degree++;
        return tree1;
    }

    Node* unionHeaps(Node* h1, Node* h2) {
        if (!h1) return h2;
        if (!h2) return h1;

        Node* newHeap = nullptr;
        Node** tail = &newHeap;

        while (h1 && h2) {
            if (h1->degree < h2->degree) {
                *tail = h1;
                h1 = h1->sibling;
            }
            else if (h1->degree > h2->degree) {
                *tail = h2;
                h2 = h2->sibling;
            }
            else {
                Node* merged = mergeTrees(h1, h2);
                *tail = merged;
                h1 = h1->sibling;
                h2 = h2->sibling;
            }
            tail = &(*tail)->sibling;
        }

        if (h1) *tail = h1;
        if (h2) *tail = h2;

        return newHeap;
    }

    Node* extractMinNode(Node* head, Node*& minNode) {
        if (!head) return nullptr;
        if (!minNode || head->value < minNode->value) {
            minNode = head;
        }
        return extractMinNode(head->sibling, minNode);
    }

public:
    Node* head;

    BinomialHeap() : head(nullptr) {}

    void insert(int value) {
```

```
        Node* newNode = new Node(value);
        head = unionHeaps(head, newNode);
    }

    int extractMin() {
        if (!head) throw std::out_of_range("Heap is empty");

        Node* minNode = nullptr;
        head = extractMinNode(head, minNode);

        Node* newHead = nullptr;
        if (minNode->child) {
            Node* child = minNode->child;
            while (child) {
                Node* nextChild = child->sibling;
                child->sibling = newHead;
                child->parent = nullptr;
                newHead = child;
                child = nextChild;
            }
        }

        head = unionHeaps(head, newHead);
        return minNode->value;
    }

    int getMin() {
        if (!head) throw std::out_of_range("Heap is empty");

        Node* minNode = nullptr;
        extractMinNode(head, minNode);

        return minNode->value;
    }

    bool isEmpty() {
        return head == nullptr;
    }
};
```

- Класс BinomialHeap: Создание биномиальной кучи
    - Функция mergeTrees
        - Вход: tree1, tree2 — две деревья биномной кучи.
        - Работа: Объединяет два дерева биномной кучи, чтобы сформировать новое дерево с меньшим корнем.
        - Выход: Возвращает корень объединенного дерева.
    - Функция unionHeaps
        - Вход: h1, h2 — два состояния биномной кучи.

- Работа: Объединяет две кучи, сливая их по степеням деревьев и вызывает mergeTrees для деревьев одинаковой степени.
- Выход: Возвращает новый список деревьев (слиянную кучу).

- Функция extractMinNode
  - Вход: head — список деревьев кучи, minNode — минимальный элемент, который нужно обновить.
  - Работа: Обходит все деревья кучи и находит минимальный элемент среди них.
  - Выход: Возвращает обновленный список деревьев и минимальный элемент кучи.
- Функция insert
  - Вход: значение, которое нужно добавить в кучу.
  - Работа: Создает новое дерево для элемента и сливает его с текущей кучей с помощью unionHeaps.
  - Выход: Нет.
- Функция extractMin
  - Вход: Нет.
  - Работа: Извлекает минимальный элемент из кучи, удаляет его и сливает оставшиеся деревья, чтобы восстановить структуру кучи.
  - Выход: Возвращает минимальный элемент.
- Функция getMin
  - Вход: Нет.
  - Работа: Ищет минимальный элемент в куче, не изменяя саму кучу.
  - Выход: Минимальный элемент кучи.
- Функция isEmpty
  - Вход: Нет.
  - Работа: Проверяет, пуста ли куча.
  - Выход: Возвращает true, если куча пуста, иначе false.

```
   long long measureAvgTime(std::function<void()> operation, int count) {
       long long totalTime = 0;
       for (int i = 0; i < count; ++i) {
           auto start = std::chrono::high_resolution_clock::now();
           operation();  // Выполнение операции
           auto end = std::chrono::high_resolution_clock::now();
           long                long                duration                =
std::chrono::duration_cast<std::chrono::nanoseconds>(end - start).count();
           totalTime += duration;
       }
       return totalTime / count;
```

```
}
```

- Функция measureAvgTime измеряет среднее время выполнения 1000 операций
    - Вход: функция, которая будет выполняться несколько раз; целое число, указывающее, сколько раз нужно выполнить операцию operation.
    - Работа: Вычисляется продолжительность выполнения операции в наносекундах. Сравнивается полученное время с текущим максимальным временем и обновляются переменные.
    - Выход: Возвращается среднее время (в наносекундах) для запусков операции.

```
long long measureMaxTime(std::function<void()> operation, int count) {
    long long maxTime = 0;
    for (int i = 0; i < count; ++i) {
        auto start = std::chrono::high_resolution_clock::now();
        operation();  // Выполнение операции
        auto end = std::chrono::high_resolution_clock::now();
        long long duration = std::chrono::duration_cast<std::chrono::nanoseconds>(end - start).count();
        maxTime = std::max(maxTime, duration);
    }
    return maxTime;
}
```

- Функция measureMaxTime измеряет максимальное время выполнения переданной операции
    - Вход: функция, которая будет выполняться несколько раз; целое число, указывающее, сколько раз нужно выполнить операцию operation.
    - Работа: Вычисляется продолжительность выполнения операции в наносекундах. Сравнивается полученное время с текущим максимальным временем и обновляется переменная maxTime, если время выполнения текущей операции больше предыдущего максимума.
    - Выход: Возвращается максимальное время (в наносекундах) из всех запусков операции.

```
void measurePerformance(BinaryHeap& binaryHeap, BinomialHeap& binomialHeap, int N, std::ofstream& outFile) {
    // Случайные числа для тестов
    std::vector<int> randomNumbers(N);
    for (int i = 0; i < N; ++i) {
        randomNumbers[i] = rand() % 1000;
```

```
    }

    // Тест 1: 1000 раз найти минимум
    outFile << "Измерение для поиска минимума (1000 операций):\n";
    outFile << "Среднее время для поиска минимума для бинарной кучи: "
            << measureAvgTime([&]() {
                if (!binaryHeap.isEmpty()) {
                    binaryHeap.getMin();
                }
            }, 1000) << " наносекунд\n";
    outFile << "Среднее время для поиска минимума для биномиальной кучи: "
            << measureAvgTime([&]() {
                if (!binomialHeap.isEmpty()) {
                    binomialHeap.getMin();
                }
            }, 1000) << " наносекунд\n";

    // Тест 2: 1000 раз удалить минимум
    outFile << "Измерение для удаления минимума (1000 операций):\n";
    outFile << "Среднее время для удаления минимума для бинарной кучи: "
            << measureAvgTime([&]() {
                if (!binaryHeap.isEmpty()) {
                    binaryHeap.extractMin();
                }
            }, 1000) << " наносекунд\n";
    outFile << "Среднее время для удаления минимума для биномиальной кучи: "
            << measureAvgTime([&]() {
                if (!binomialHeap.isEmpty()) {
                    binomialHeap.extractMin();
                }
            }, 1000) << " наносекунд\n";

    // Тест 3: 1000 раз добавить новый элемент
    outFile << "Измерение для добавления элемента (1000 операций):\n";
    outFile << "Среднее время для добавления элемента в бинарную кучу: "
            << measureAvgTime([&]() {
                binaryHeap.insert(randomNumbers[rand() % N]);
            }, 1000) << " наносекунд\n";
    outFile << "Среднее время для добавления элемента в биномиальную кучу: "
            << measureAvgTime([&]() {
                binomialHeap.insert(randomNumbers[rand() % N]);
            }, 1000) << " наносекунд\n";

    // Максимальное время для 10, 25, 50 и 100 операций
    std::vector<int> operations = {10, 25, 50, 100};

    // Максимальное время для поиска минимума
    for (int opCount : operations) {
        outFile << "Максимальное время для " << opCount << " операций поиска
минимума для бинарной кучи: "
                << measureMaxTime([&]() {
                    if (!binaryHeap.isEmpty()) {
```

```
                        binaryHeap.getMin();
                    }
                }, opCount) << " наносекунд\n";
            outFile << "Максимальное время для " << opCount << " операций поиска
минимума для биномиальной кучи: "
                << measureMaxTime([&]() {
                    if (!binomialHeap.isEmpty()) {
                        binomialHeap.getMin();
                    }
                }, opCount) << " наносекунд\n";
        }

        // Максимальное время для удаления минимума
        for (int opCount : operations) {
            outFile << "Максимальное время для " << opCount << " операций удаления
минимума для бинарной кучи: "
                << measureMaxTime([&]() {
                    if (!binaryHeap.isEmpty()) {
                        binaryHeap.extractMin();
                    }
                }, opCount) << " наносекунд\n";
            outFile << "Максимальное время для " << opCount << " операций удаления
минимума для биномиальной кучи: "
                << measureMaxTime([&]() {
                    if (!binomialHeap.isEmpty()) {
                        binomialHeap.extractMin();
                    }
                }, opCount) << " наносекунд\n";
        }

        // Максимальное время для добавления элемента
        for (int opCount : operations) {
            outFile << "Максимальное время для " << opCount << " операций
добавления элемента в бинарную кучу: "
                << measureMaxTime([&]() {
                    binaryHeap.insert(randomNumbers[rand() % N]);
                }, opCount) << " наносекунд\n";
            outFile << "Максимальное время для " << opCount << " операций
добавления элемента в биномиальную кучу: "
                << measureMaxTime([&]() {
                    binomialHeap.insert(randomNumbers[rand() % N]);
                }, opCount) << " наносекунд\n";
        }
    }
```

- Функция measurePerformance
    - Вход:
        - binaryHeap — объект бинарной кучи.
        - binomialHeap — объект биномной кучи.
        - N — количество операций для тестирования.

        - outFile — файл для записи результатов.
    - Работа: Производит тестирование производительности для разных операций (поиск минимума, удаление минимума, вставка нового элемента) для обеих куч и записывает время выполнения операций в файл.
    - Выход: Нет. Функция записывает результаты в файл.

```
int main() {
    srand(time(0));
    // Открываем файл для записи результатов
    std::ofstream outFile("heap_performance_results.txt");
    if (!outFile) {
        std::cerr << "Не удалось открыть файл для записи!" << std::endl;
        return 1;
    }
    for (int i = 3; i <= 7; ++i) {
        int N = pow(10, i);  // N = 10^i
        BinaryHeap binaryHeap;
        BinomialHeap binomialHeap;
        // Заполнение кучами
        for (int j = 0; j < N; ++j) {
            int randomValue = rand() % 1000;  // Случайное значение от 0 до
999
            binaryHeap.insert(randomValue);
            binomialHeap.insert(randomValue);
        }
       // Измерение производительности и запись в файл
        outFile << "Тест для N = " << N << ":\n";
        measurePerformance(binaryHeap, binomialHeap, N, outFile);
        outFile << std::endl;
    }
    outFile.close();  // Закрываем файл
    std::cout << "Результаты записаны в файл heap_performance_results.txt\n";
    return 0;
}
```

- Функция main
    - Вход: Нет.
    - Работа: Инициализирует генератор случайных чисел, открывает файл для записи, выполняет тестирование производительности для различных размеров куч и записывает результаты в файл.
    - Выход: Возвращает 0, если выполнение прошло успешно, и 1, если возникла ошибка при открытии файла.

Заключение.

Результаты работы были представлены в виде таблицы значений, а затем интересующие данные были представлены в виде графиков

	мкс					
N	Bin Search	Binom Search	Bin Delete	Binom Delete	Bin Insert	Binom Insert
1000	63,732	49,124	333,983	45,434	124,492	98,287
10000	76,863	65,495	449,698	43,441	102,257	113,918
100000	67,198	49,672	601,623	43,886	95,721	104,873
1000000	66,49	50,721	776,375	40,528	114,729	119,327
10000000	72,576	52,548	885,945	42,168	234,864	213,648

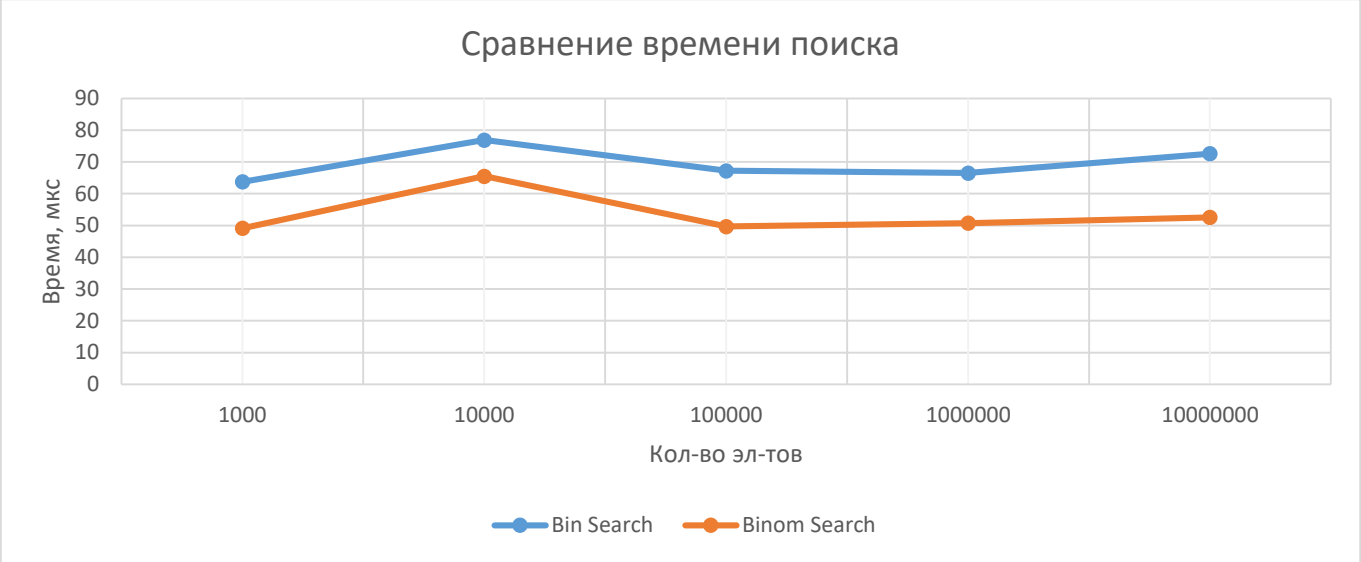
Таблица 1. Значения времени для 1000 операций

N	bin search	binom sear	bin delete	binom del	bin insert	binom insert
1000	124	98	580	156	240	2577
10000	231	137	626	162	323	1455
100000	8489	245	927	209	549	3877
1000000	316	4063	1142	352	587	1765
10000000	220	153	1620	187	564	1838

Таблица 2. Значения макс. времени для 1 операции

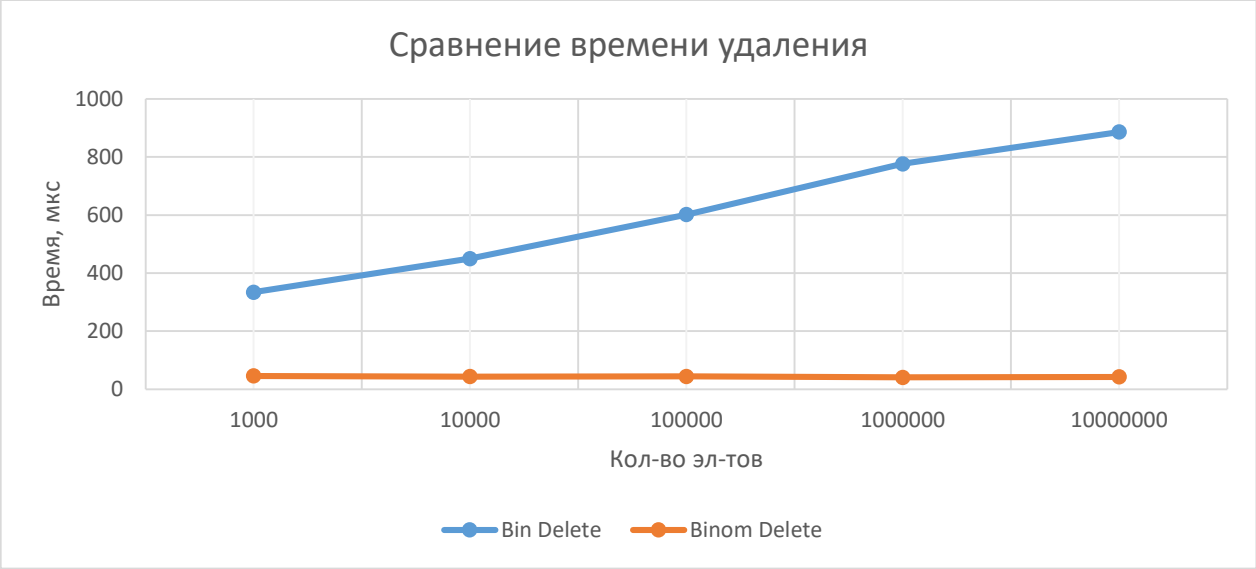
Поиск

- С увеличением N, время работы обеих структур увеличивается, темп роста в обеих структурах близок.



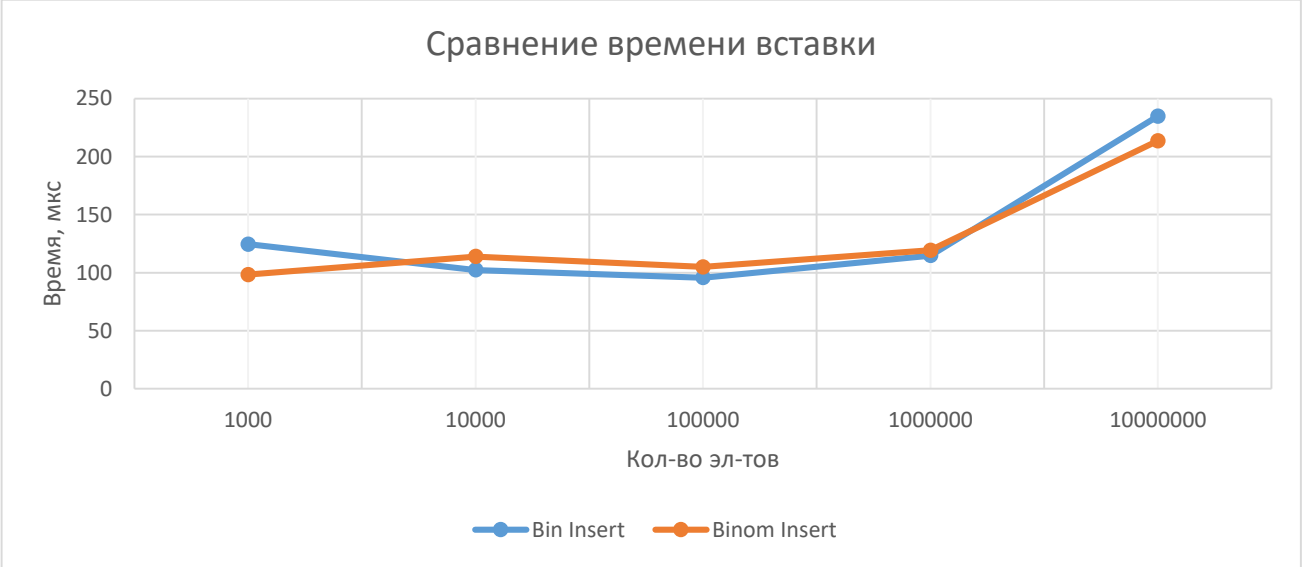
Удаление

- Операция удаления для бинарной кучи требует больше времени из-за необходимости "просеивать" элементы в одноуровневой структуре, что менее эффективно, чем в биномиальной куче. Разница во времени для удаления становится менее выраженной по мере увеличения N, но биномиальная куча остается быстрее на протяжении всех значений N.



Вставка

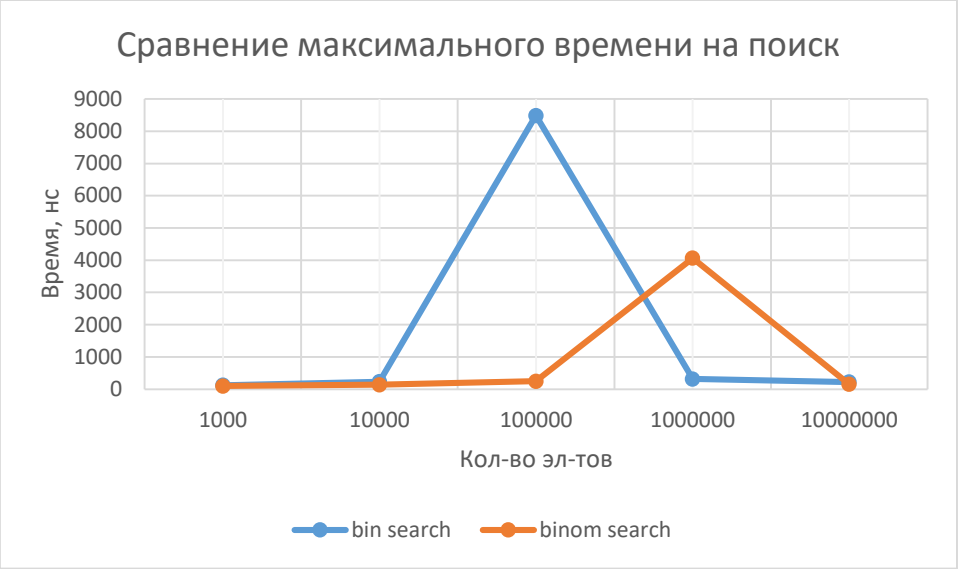
- Вставка в бинарной куче требует перераспределения элементов в куче, что чуть более затратно, чем в биномиальной куче, где операции слияния деревьев более эффективны.



Анализ времени выполнения для 1 операции

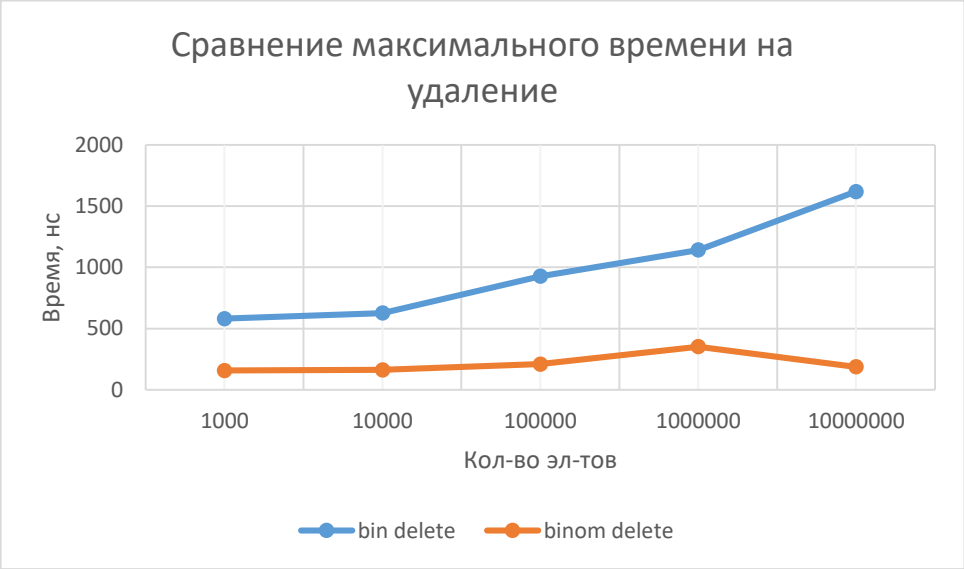
Поиск

Бинарная куча работает немного медленнее для поиска, особенно при больших NNN, так как её структура менее сбалансирована по сравнению с биномиальной кучей.



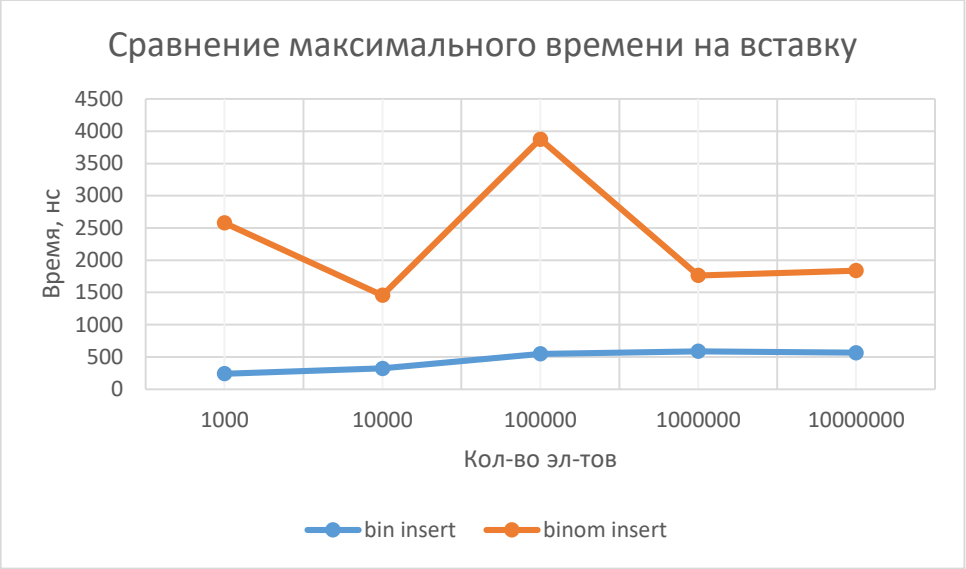
Удаление

Биномиальная куча остаётся значительно быстрее при удалении элемента, что подтверждает её эффективность в таких операциях.



Вставка

Вставка в бинарной куче более оптимизирована, так как она использует простую стратегию "просеивания" вверх, в отличие от биномиальной кучи, где происходит слияние деревьев, что может потребовать больше времени на маленьких N.



Выводы:

1. Бинарная куча (Bin) выигрывает по времени поиска (Bin Search) на малых значениях N, но с ростом N разница становится меньше.
2. Биномиальная куча (Binom) значительно быстрее в операциях удаления (Bin Delete) и вставки (Bin Insert), особенно при больших значениях N.
3. По мере увеличения N разница между бинарной и биномиальной кучей становится более выраженной, что делает биномиальную кучу предпочтительнее в случаях, когда операции вставки и удаления выполняются часто и N велико.

В итоге биномиальная куча может быть предпочтительнее для приложений, где важно эффективно сливать кучи и часто выполнять операции вставки и удаления, а бинарная куча — для задач, где ключевыми являются быстрые поиски элементов.