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-38.....(Нергарян Геворг Гарегинович)
Ссылка на репозиторий: .....(https://github.com/MUCTR-IKT-CPP/Nergaryan_KC-38_Algos)

Приняли: .....Пысин Максим Дмитриевич
.....Краснов Дмитрий Олегович
.....Лобанов Алексей Владимирович
.....Крашенинников Роман Сергеевич

Дата сдачи: 23.05.2023

Оглавление

Описание задачи.

В рамках лабораторной работы необходимо реализовать бинарную кучу(мин или макс), а так же Фибоначиеву кучу

Для реализованных куч выполнить следующие действия:

- Наполнить кучу N кол-ва элементов (где N = 10 ^ i, i от 3 до 7).
- После заполнения кучи необходимо провести следующие тесты:
- 1000 раз найти минимум/максимум
- 1000 раз удалить минимум/максимум
- 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 расчитывать усредненное время на каждые 10,25,50,100 операций, и выбирать максимальное из полученных результатов, что бы поймать момент деградации структуры и ее перестройку.

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.

Бинарная куча, также известная как двоичная куча, является структурой данных, которая обеспечивает эффективное хранение и выполнение операций на наборе элементов с определенным порядком. Бинарная куча может быть реализована как мин-куча или макс-куча в зависимости от задачи.

Мин-куча представляет собой бинарное дерево, в котором для каждого узла выполняется условие, что значение в узле меньше или равно значениям его дочерних узлов. Это означает, что наименьший элемент находится в корне кучи, а остальные элементы располагаются в порядке возрастания. Макс-куча, напротив, удовлетворяет условию, что значение в узле больше или равно значениям его дочерних узлов. Таким образом, наибольший элемент находится в корне кучи, а остальные элементы располагаются в порядке убывания.

Основные операции, выполняемые на бинарной куче, включают вставку, удаление минимального (максимального) элемента и поиск минимального (максимального) элемента. При вставке нового элемента в кучу, он сравнивается со значениями существующих элементов и соответствующим образом вставляется на свободное место, сохраняя свойство мин-кучи или макс-кучи. При удалении минимального (максимального) элемента, он заменяется последним элементом в куче, после чего выполняется перестройка кучи для восстановления ее свойств. Поиск минимального (максимального) элемента осуществляется просто доступом к корневому узлу.

Фибоначчиева куча - это особый вид кучи, который является более эффективным по сравнению с бинарной кучей. Она основана на числах Фибоначчи и представляет собой набор деревьев, в которых каждый узел имеет связи с другими узлами на том же уровне. В отличие от бинарной кучи, Фибоначчиева куча позволяет более эффективно выполнять операции вставки, удаления и изменения значений элементов.

Фибоначчиева куча поддерживает следующие операции:

- Вставка: новый элемент добавляется в корень одного из деревьев и сравнивается с минимальным (максимальным) элементом.
- Удаление минимального (максимального) элемента: минимальный (максимальный) элемент удаляется, а его дочерние узлы объединяются с другими деревьями.
- Обновление значения элемента: значение элемента изменяется, и при необходимости выполняется перестройка дерева.

Основное преимущество Фибоначчиевой кучи заключается в том, что она обеспечивает амортизированную сложность O(1) для операций вставки и удаления, что делает ее эффективной для динамических изменений в структуре данных. Однако, она может потреблять больше памяти по сравнению с бинарной кучей и иметь более высокую константу времени выполнения операций.

В целом, Фибоначчиева куча представляет собой интересную и эффективную структуру данных, которая может быть применена в различных задачах, требующих операций вставки, удаления и обновления элементов с лучшей амортизированной сложностью. Однако, выбор между бинарной кучей и Фибоначчиевой кучей зависит от конкретных требований и особенностей задачи, а также от компромиссов между временем выполнения операций и использованием памяти.

Выполнение задачи.

Код на Языке C++

```
#include <iostream>
#include <vector>
#include <cmath>
#include <set>
#include <random>
#include <fstream>
#include <chrono>

using namespace std;

class BinaryHeap {

    vector<int> array;

public:

    //минимум
    int getMin() {
        return array[0];
    }

    //вставка элемента в двоичную кучу
    void insert(int data) {
        array.push_back(data); // Добавляем элемент в конец массива
        int i = array.size() - 1;  // Получаем индекс добавленного элемента
        int k = (i - 1) / 2;// Вычисляем индекс родительского элемента
        while (i > 0 && array[k] > array[i]) {  // Пока добавленный элемент не станет корнем кучи и его значение меньше значения родителя
            int temp = array[i]; // Меняем местами добавленный элемент и его родителя
            array[i] = array[k];
            array[k] = temp;
            i = k;// Обновляем индексы добавленного элемента и его родителя
            k = (i - 1) / 2;
```

```
        }
    }


    //удаляем мин элемент из двоичной кучи и возвращаем значение
    int remove() {
        int min = array[0]; // Сохраняем значение минимального элемента
        array[0] = array[array.size() - 1];// Заменяем минимальный элемент на последний элемент массива
        int i = 0;// Обновляем индекс текущего элемента
        while (2 * i + 1 < array.size()) { // Пока у текущего элемента есть хотя бы один потомок
            int left = 2 * i + 1; // Вычисляем индексы левого и правого потомков
            int right = 2 * i + 2;
            int ch; // Выбираем потомка с меньшим значением
            if (right < array.size() && array[right] < array[left]) {
                ch = right;
            }
            else {
                ch = left;
            }
            if (array[i] <= array[ch]) { // Если значение текущего элемента меньше или равно значению
выбранного потомка, то прерываем цикл
                break;
            }
            else {
                int temp = array[i];
                array[i] = array[ch];
                array[ch] = temp;
                i = ch;
            }
        }
        return min;
    }
};

class FibHeap {

    class Node {
```

```
    public:
        int data;     // ключ
        Node* parent;  // указатель на родительский узел
        Node* child;   // указатель на один из дочерних узлов
        Node* left;   // указатель на левый узел того же предка
        Node* right;   // указатель на правый узел того же предка
        int degree;    // степень вершины


        Node(int data) {
            this->data = data;
            this->parent = nullptr;
            this->child = nullptr;
            this->left = nullptr;
            this->right = nullptr;
            this->degree = 0;//степень
        }

        Node() {

        }
    };

    int size;
    Node* min;

public:
    FibHeap() {
        this->size = 0;
        this->min = nullptr;
    }

    int getMin() {
        return min->data;
    }

    void insert(int data) {
```

```
        Node* node = new Node(data); // создаем новый узел
        if (size == 0) { // если это первый элемент в списке
            min = node;// он становится минимальным и указывает на самого себя
            node->left = node; // указываем, что левый и правый элементы - это сам новый элемент
            node->right = node;
        }
        else {
            Node* min_right = min->right;// если уже есть другие элементы в списке запоминаем правый элемент от минимального
            min->right = node;// минимальный указывает на новый элемент
            node->left = min; // новый элемент указывает на минимальный
            node->right = min_right; // новый элемент указывает на правый элемент от минимального
            min_right->left = node; //правый элемент от минимального указывает на новый элемент

        }
        if (node->data < min->data) {// если новый элемент меньше текущего минимального
            min = node; // он становится минимальным

        }
        size++;
    }

    //два узла в один список
    void Unite(Node* left, Node* right) {
        if (left == nullptr) {// если левый узел не существует
            min = right;// то правый становится минимальным элементом
            return;
        }
        if (right == nullptr) {// если правый узел не существует
            min = left;
            return;
        }
        Node* l = left->left;// запоминаем левый элемент от левого узла
        Node* r = right->right;// запоминаем правый элемент от правого узла
        right->right = left;// правый узел указывает на левый узел
        left->left = right;// левый узел указывает на правый узел
        l->right = r;// левый элемент от левого узла указывает на правый элемент от правого узла
```

```
        r->left = l;// правый элемент от правого узла указывает на левый
    }

    void Consolidate() {
        vector<Node*> array(size, nullptr);// создаем вектор, хранящий узлы кучи
        array[min->degree] = min;// находим минимальный элемент и помещаем его в соответствующую ячейку вектора

        Node* current = min->right;// начинаем смотреть узлы справа от минимального элемента
        while (array[current->degree] != nullptr) {// пока нашли узел с такой же степенью, как у текущего узла
            auto conflict = array[current->degree]; // запоминаем этот узел
            Node* addTo, * adding;

            if (conflict->data < current->data) {// если значение в конфликтующем узле меньше, чем в текущем
                addTo = conflict; // узел с меньшим значением становится родительски
                adding = current;// узел с большим значением становится дочерним
            }
            else {
                addTo = current;
                adding = conflict;
            }

            Unite(addTo->child, adding); // объединяем дочерние узлы родительского и дочернего узла
            adding->parent = addTo;// устанавливаем родительский узел для дочернего узла
            addTo->degree++; // увеличиваем степень родительского узла
            current = addTo;

            array[current->degree] = nullptr; //обнуляем ячейку массива, соответствующую степени родительского узла
        }

        // находим узел с минимальным значением
        min = current;

        for (int i = 0; i < size; i++) {
```

```
            if (array[i] != nullptr && array[i]->data < min->data)
                min = array[i];
        }
        // обновляем минимальный
        if (min->left->data < min->data)
            min = min->left;

    }


    int Remove() {
        if (this->min == nullptr)
            throw runtime_error("error");

        Node* to_delete = this->min;// сохраняем удаляемый узел
        Unite(this->min, this->min->child);// объединяем дочерние узлы удаляемого узла с корнем кучи

        Node* left = min->left;// сохраняем левого соседа минимального узла
        Node* right = min->right;// сохраняем правого соседа минимального узла
        left->right = right;// связываем левого и правого соседей минимального узла
        right->left = left;

        if (to_delete->right == to_delete) {//если у нас один узел
            this->min = nullptr;// обнуляем корень кучи
            this->size--;// уменьшаем размер кучи
            return to_delete->data;// возвращаем значение удаленного узла
        }

        this->min = min->right;// новый корень кучи - правый сосед минимального узла
        Consolidate();// пересчитываем степени узла
        this->size--;//уменьшаем размер кучи
        return to_delete->data;
    }
};


//Возвращает рандомный int между start и end
```

```
int GetRandomInt(int start, int end) {
    std::random_device rd;
    std::mt19937 engine(rd());
    std::uniform_int_distribution<int> gen(start, end);
    return gen(engine);
}

int main() {

    srand(time(NULL));

    ofstream insert;
    insert.open(R"(C:\Users\Arai\Desktop\КС-28\Алгосы\Laba8\Insert.txt)");

    ofstream search;
    search.open(R"(C:\Users\Arai\Desktop\КС-28\Алгосы\Laba8\Search.txt)");

    std::ofstream remove;
    remove.open(R"(C:\Users\Arai\Desktop\КС-28\Алгосы\Laba8\Delete.txt)");


    for (int i = 3; i <= 7; i++) {
        vector<int> array(pow(10, i), 0);
        set<int> s;
        for (int k = 0; k < array.size(); k++) {
            int x;
            do {
                x = GetRandomInt(0, array.size() * 3);
            } while (s.count(x)); // проверяем, есть ли это число в наборе
            s.insert(x); // добавляем число в набор
            array[k] = x; // помещаем число в массив
        }
        BinaryHeap binaryHeap;
        FibHeap fibHeap;

        for (int k = 0; k < array.size(); k++) {
            binaryHeap.insert(array[k]);
```

```
            fibHeap.insert(array[k]);
        }

        chrono::duration<double, std::milli> binary_time_max = chrono::duration<double, std::milli>::min();
//максимальное время
        chrono::high_resolution_clock::time_point startTime = chrono::high_resolution_clock::now();
        for (int k = 0; k < 1000; k++) {
            chrono::high_resolution_clock::time_point start = chrono::high_resolution_clock::now();

            binaryHeap.getMin();

            chrono::high_resolution_clock::time_point end = chrono::high_resolution_clock::now();

            if (binary_time_max < end - start) {
                binary_time_max = end - start;
            }
        }
        chrono::high_resolution_clock::time_point endTime = chrono::high_resolution_clock::now();
        chrono::duration<double, std::milli> binary_time = endTime - startTime;

        chrono::duration<double, std::milli> fib_time_max = chrono::duration<double, std::milli>::min();
//максимальное время
        startTime = chrono::high_resolution_clock::now();
        for (int k = 0; k < 1000; k++) {
            chrono::high_resolution_clock::time_point start = chrono::high_resolution_clock::now();

            fibHeap.getMin();

            chrono::high_resolution_clock::time_point end = chrono::high_resolution_clock::now();

            if (fib_time_max < end - start) {
                fib_time_max = end - start;
            }
        }
        endTime = chrono::high_resolution_clock::now();
        chrono::duration<double, std::milli> fib_time = endTime - startTime;
```

```
        cout << "Search time to binary: " << binary_time.count() << ' ' << binary_time_max.count() << ' ' <<
"Size: " << array.size() << endl;
        cout << "Search time to Fib : " << fib_time.count() << ' ' << fib_time_max.count() << " Size: " <<
array.size()
             << endl;

        if (search.is_open()) {
            search << array.size() << ' ' << binary_time.count() << ' ' << binary_time.count() / 1000 << ' ' <<
binary_time_max.count() << ' '
                << fib_time.count() << ' ' << fib_time.count() / 1000 << ' ' << fib_time_max.count() << endl;
        }



        startTime = chrono::high_resolution_clock::now();
        binary_time_max = chrono::duration<double, std::milli>::min();
        for (int k = 0; k < 1000; k++) {
            chrono::high_resolution_clock::time_point start = chrono::high_resolution_clock::now();

            binaryHeap.remove();

            chrono::high_resolution_clock::time_point end = chrono::high_resolution_clock::now();

            if (binary_time_max < end - start) {
                binary_time_max = end - start;
            }
        }
        endTime = chrono::high_resolution_clock::now();
        binary_time = endTime - startTime;

        fib_time_max = chrono::duration<double, std::milli>::min();
        startTime = chrono::high_resolution_clock::now();
        for (int k = 0; k < 1000; k++) {
            chrono::high_resolution_clock::time_point start = chrono::high_resolution_clock::now();

            fibHeap.Remove();

            chrono::high_resolution_clock::time_point end = chrono::high_resolution_clock::now();
```

```
        if (fib_time_max < end - start) {
            fib_time_max = end - start;
        }
    }
    endTime = chrono::high_resolution_clock::now();
    fib_time = endTime - startTime;

    cout << "Delete time to binary: " << binary_time.count() << ' ' << binary_time.count() / 1000 << ' ' <<
binary_time_max.count() << " Size: "
        << array.size() << endl;
    cout << "Delete time to Fib : " << fib_time.count() << ' ' << fib_time_max.count() << ' ' << "Size: " <<
array.size()
        << endl;

    if (remove.is_open()) {
        remove << array.size() << ' ' << binary_time.count() << ' ' << binary_time.count() / 1000 << ' ' <<
binary_time_max.count() << ' '
            << fib_time.count() << ' ' << fib_time.count() / 1000 << ' ' << fib_time_max.count() << endl;
    }



    vector<int> mas(1000);
    set<int> set;
    for (int k = 0; k < mas.size(); k++) {
        int x;
        do {
            x = GetRandomInt(0, 10000000);
        } while (set.count(x)); // проверяем, есть ли это число в наборе
        set.insert(x); // добавляем число в набор
        mas[k] = x; // помещаем число в массив
    }

    startTime = chrono::high_resolution_clock::now();
    binary_time_max = chrono::duration<double, std::milli>::min();
    for (int k = 0; k < mas.size(); k++) {
        chrono::high_resolution_clock::time_point start = chrono::high_resolution_clock::now();
```

```
            binaryHeap.insert(mas[k]);

            chrono::high_resolution_clock::time_point end = chrono::high_resolution_clock::now();

            if (binary_time_max < end - start) {
                binary_time_max = end - start;
            }
        }
        endTime = chrono::high_resolution_clock::now();
        binary_time = endTime - startTime;

        fib_time_max = chrono::duration<double, std::milli>::min();
        startTime = chrono::high_resolution_clock::now();
        for (int k = 0; k < mas.size(); k++) {
            chrono::high_resolution_clock::time_point start = chrono::high_resolution_clock::now();

            fibHeap.insert(mas[k]);

            chrono::high_resolution_clock::time_point end = chrono::high_resolution_clock::now();

            if (fib_time_max < end - start) {
                fib_time_max = end - start;
            }
        }
        endTime = chrono::high_resolution_clock::now();
        fib_time = endTime - startTime;

        cout << "Insert time to binary: " << binary_time.count() << ' ' << binary_time.count() / 1000 << ' ' <<
binary_time_max.count() << " Size: "
            << array.size() << endl;
        cout << "Insert time to Fib : " << fib_time.count() << ' ' << fib_time.count() / 1000 << ' ' <<
fib_time_max.count() << ' ' << "Size: "
            << array.size()
            << endl;

        if (insert.is_open()) {
```

```
            insert << array.size() << ' ' << binary_time.count() << ' ' << binary_time.count() / 1000 << ' ' <<
binary_time_max.count() << ' '
                << fib_time.count() << ' ' << fib_time.count() / 1000 << ' ' << fib_time_max.count() << endl;
        }
    }
    return 0;
}
```

По полученным данным были построены графики времени выполнения операций для усреднения по 1000 операций:

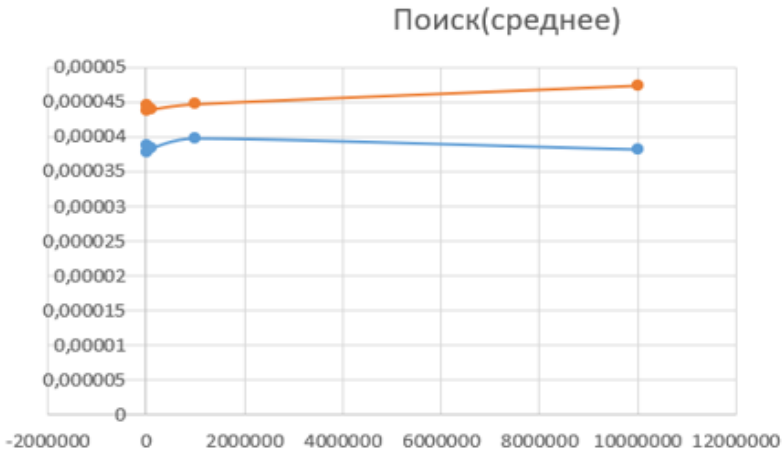

Рис. 1 Поиск. Усреднение 1000 операций

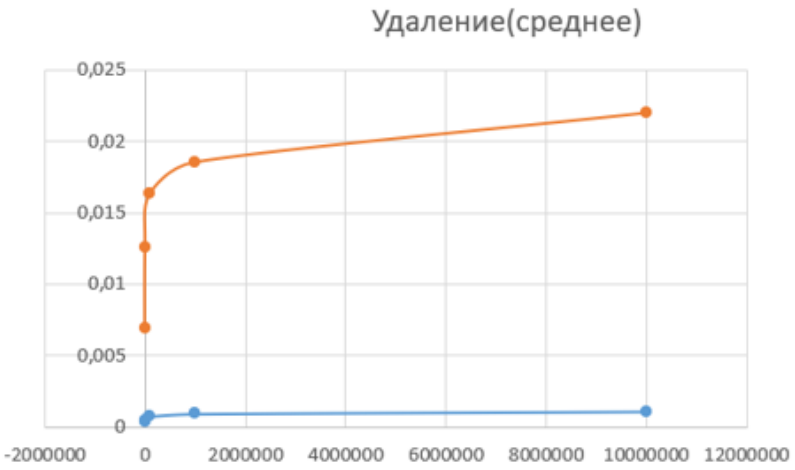

Рис. 2 Удаление. Усреднение 1000 операций

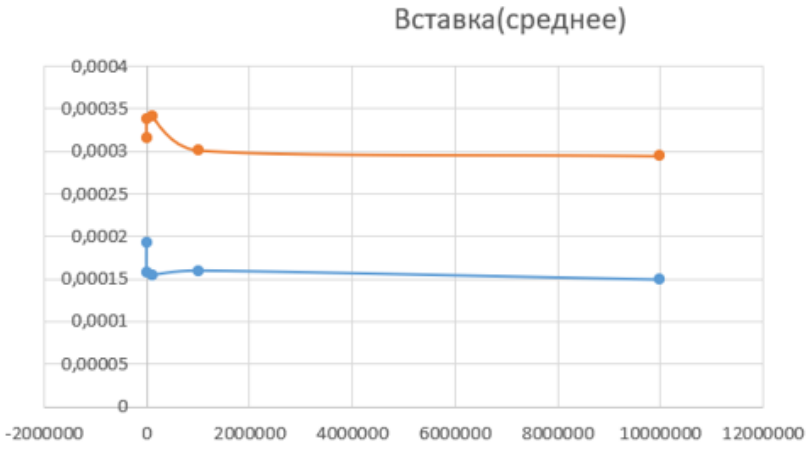

Рис. 3 Вставка. Усреднение 1000 операций

А также для максимального времени на 1 операцию:

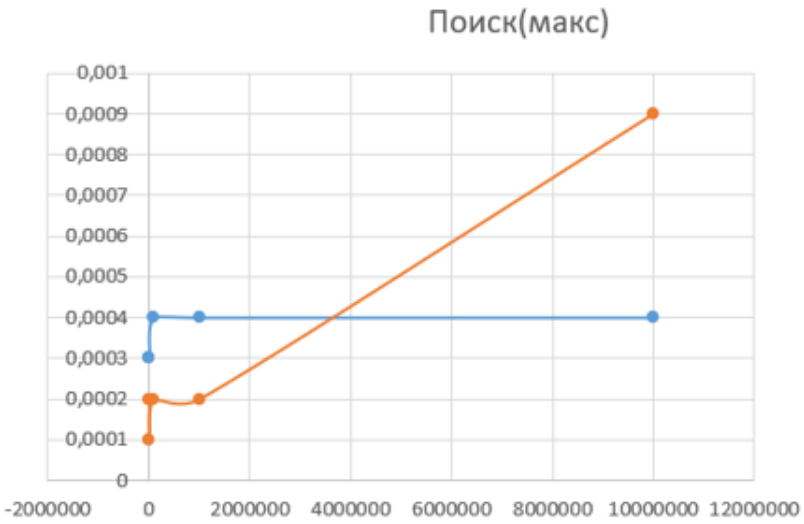

Рис. 4 Поиск. Максимальное время

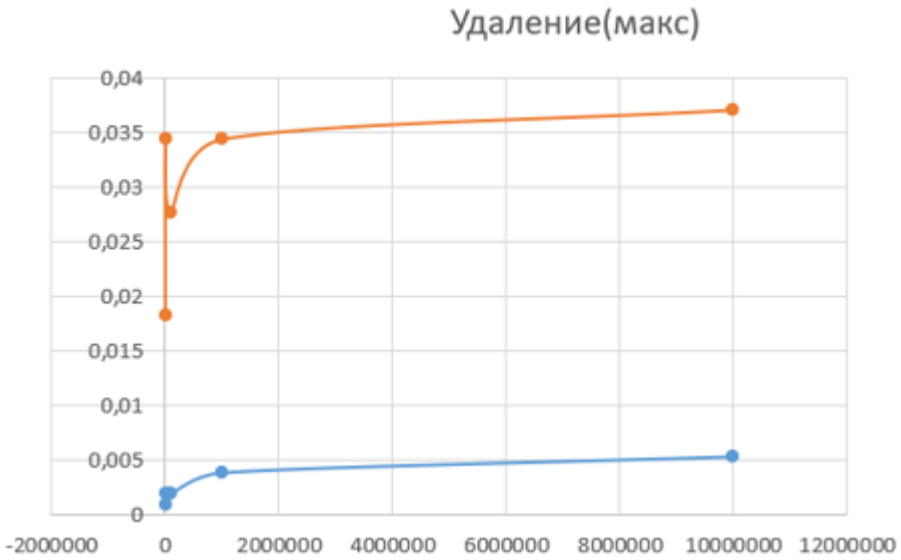

Рис. 5 Удаление. Максимальное время

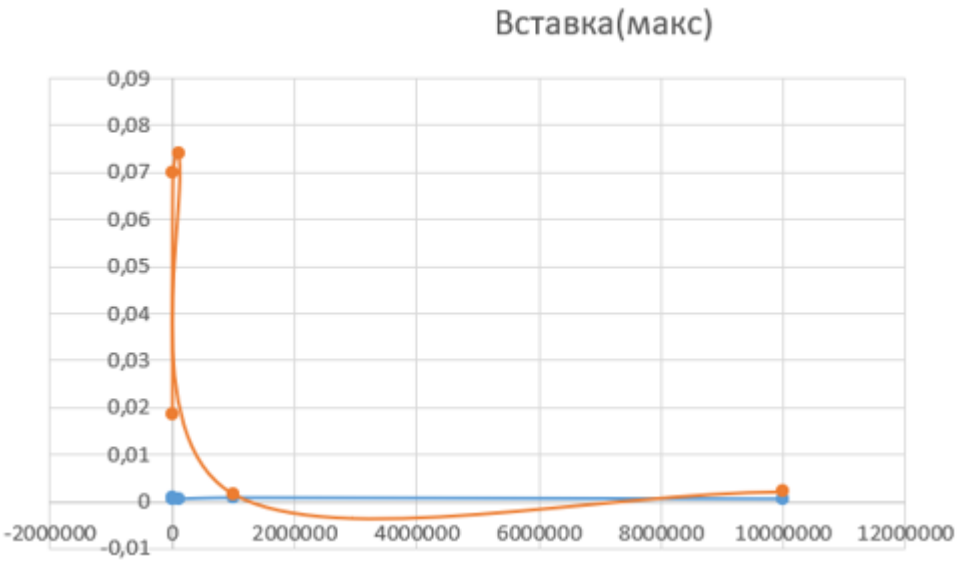

Рис. 6 Вставка. Максимальное время

Заключение.

В результате проведенной работы и сравнения бинарной кучи и Фибоначчиевой кучи были получены следующие выводы. Фибоначчиева куча оказалась более эффективной при выполнении операций вставки новых элементов и удаления минимума/максимума. Это объясняется ее способностью объединять две кучи за амортизированное константное время и выполнять удаление за амортизированное логарифмическое время. Бинарная куча, в свою очередь, требует логарифмическое время для выполнения этих операций. Однако, оба типа куч показали схожую производительность при операции нахождения минимума/максимума, выполняющейся за константное время.

Таким образом, Фибоначчиева куча является предпочтительным выбором, если задача требует частых операций вставки и удаления элементов, так как она обеспечивает лучшую асимптотическую производительность. Однако, при выборе между этими двумя типами куч следует учитывать требования конкретной задачи, особенности данных и ограничения по памяти. В зависимости от конкретной ситуации и ограничений, бинарная куча также может быть эффективным решением.