Министерство науки и высшего образования Российской Федерации
Российский химико-технологический университет имени Д. И. Менделеева

Факультет информационных технологий и управления
Кафедра информационных компьютерных технологий

ОТЧЕТ ПО ЛАБОРАТОРНОЙ РАБОТЕ №8

«Кучи»

Выполнил студент группы КС-30 Колесников Артем Максимович

Ссылка на репозиторий: https://github.com/MUCTR-IKT-CPP/AMKolesnikov_30_ALG

Приняли: аспирант кафедры ИКТ Пысин Максим Дмитриевич
аспирант кафедры ИКТ Краснов Дмитрий Олегович

Дата сдачи: 23.05.2022

Москва
2022

Содержание

Описание задачи

В рамках лабораторной работы необходимо реализовать бинарную кучу (мин или макс), а также биномиальную 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:
   - 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 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 структуры

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.

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 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, что бы оно состояло из суммы 2^l(i). Важной особенностью биноминальной кучи является то, что она не должна содержать в себе деревьев одного порядка.

Выполнение задачи

Я выполнил данную лабораторную работу на языке C++. Моя программа состоит из 2 классов: «BinaryHeap», «BinomialHeap» и функции «main».

Класс «BinaryHeap» ~ занимается созданием бинарной кучи. В классе есть следующие методы: добавление нового элемента «insertKey», удаление минимального элемента «extractMin», поиск минимального элемента «getMin».

```
class BinaryHeap
{
    int* harr; // pointer to array of elements in heap
    int capacity; // maximum possible size of min heap
    int heap_size; // Current number of elements in min heap
public:
    // Constructor
    BinaryHeap(int capacity);

    // to heapify a subtree with the root at given index
    void BinaryHeapify(int);

    int parent(int i) { return (i - 1) / 2; }

    // to get index of left child of node at index i
    int left(int i) { return (2 * i + 1); }

    // to get index of right child of node at index i
    int right(int i) { return (2 * i + 2); }

    // to extract the root which is the minimum element
    int extractMin();

    // Returns the minimum key (key at root) from min heap
    int getMin() { return harr[0]; }

    // Inserts a new key 'k'
    void insertKey(int k);
};

// Constructor: Builds a heap from a given array a[] of given size
BinaryHeap::BinaryHeap(int cap)
{
    heap_size = 0;
    capacity = cap;
    harr = new int[cap];
}

// Inserts a new key 'k'
void BinaryHeap::insertKey(int k)
{
    if (heap_size == capacity)
    {
        cout << "\nOverflow: Could not insertKey\n";
        return;
```

```
    }

    // First insert the new key at the end
    heap_size++;
    int i = heap_size - 1;
    harr[i] = k;

    // Fix the min heap property if it is violated
    while (i != 0 && harr[parent(i)] > harr[i])
    {
        swap(&harr[i], &harr[parent(i)]);
        i = parent(i);
    }
}


// Method to remove minimum element (or root) from min heap
int BinaryHeap::extractMin()
{
    if (heap_size <= 0)
        return INT_MAX;
    if (heap_size == 1)
    {
        heap_size--;
        return harr[0];
    }

    // Store the minimum value, and remove it from heap
    int root = harr[0];
    harr[0] = harr[heap_size - 1];
    heap_size--;
    BinaryHeapify(0);

    return root;
}


// A recursive method to heapify a subtree with the root at given index
// This method assumes that the subtrees are already heapified
void BinaryHeap::BinaryHeapify(int i)
{
    int l = left(i);
    int r = right(i);
    int smallest = i;
    if (l < heap_size && harr[l] < harr[i])
        smallest = l;
    if (r < heap_size && harr[r] < harr[smallest])
        smallest = r;
    if (smallest != i)
    {
        swap(&harr[i], &harr[smallest]);
        BinaryHeapify(smallest);
    }
}
```

Класс «BinomialHeap» ~ занимается созданием биномиальной кучи на основе структуры «BinNode». В классе есть следующие методы: добавление нового элемента «insert», удаление минимального элемента «extractMin», поиск минимального элемента «first», слияние двух куч «merge», проверка на пустоту «isEmpty».

```
class BinomialHeap
{
private:
```

```
    struct BinNode
    {
        int key;
        int degree;
        BinNode* f, * p, * c;

        BinNode()
        {
            this->key = 0;
            this->degree = 0;
            this->f = this->p = this->c = NULL;
        }

        BinNode(int key)
        {
            this->key = key;
            this->degree = 0;
            this->f = this->p = this->c = NULL;
        }
    };

    BinNode* roots;
    BinNode* min;
    void linkTrees(BinNode*, BinNode*);
    BinNode* mergeRoots(BinomialHeap*, BinomialHeap*);

public:
    BinomialHeap();
    BinomialHeap(BinNode*);
    bool isEmpty();
    void insert(int);
    void merge(BinomialHeap*);
    BinNode* first();
    int getMin() { return first()->key; }
    BinNode* extractMin();
};

BinomialHeap::BinomialHeap()
{
    this->roots = NULL;
}

BinomialHeap::BinomialHeap(BinNode* x)
{
    this->roots = x;
}

bool BinomialHeap::isEmpty()
{
    return (this->roots == NULL);
}

void BinomialHeap::insert(int x)
{
    this->merge(new BinomialHeap(new BinNode(x)));
}

void BinomialHeap::linkTrees(BinNode* y, BinNode* z)
{
    // Precondition: y -> key >= z -> key
    y->p = z;
    y->f = z->c;
    z->c = y;
    z->degree = z->degree + 1;
}
```

```
BinNode* BinomialHeap::mergeRoots(BinomialHeap* x, BinomialHeap* y)
{
    BinNode* ret = new BinNode();
    BinNode* end = ret;

    BinNode* L = x->roots;
    BinNode* R = y->roots;
    if (L == NULL) return R;
    if (R == NULL) return L;
    while (L != NULL || R != NULL)
    {
        if (L == NULL)
        {
            end->f = R;
            end = end->f;
            R = R->f;
        }
        else if (R == NULL)
        {
            end->f = L;
            end = end->f;
            L = L->f;
        }
        else
        {
            if (L->degree < R->degree)
            {
                end->f = L;
                end = end->f;
                L = L->f;
            }
            else
            {
                end->f = R;
                end = end->f;
                R = R->f;
            }
        }
    }
    return (ret->f);
}

void BinomialHeap::merge(BinomialHeap* bh)
{
    BinomialHeap* H = new BinomialHeap();
    H->roots = mergeRoots(this, bh);

    if (H->roots == NULL)
    {
        this->roots = NULL;
        this->min = NULL;
        return;
    }

    BinNode* prevX = NULL;
    BinNode* x = H->roots;
    BinNode* nextX = x->f;
    while (nextX != NULL)
    {
        if (x->degree != nextX->degree || (nextX->f != NULL && nextX->f->degree == x-
>degree))
        {
            prevX = x;
            x = nextX;
```

```
        }
        else if (x->key <= nextX->key)
        {
            x->f = nextX->f;
            linkTrees(nextX, x);
        }
        else
        {
            if (prevX == NULL) H->roots = nextX;
            else prevX->f = nextX;
            linkTrees(x, nextX);
            x = nextX;
        }
        nextX = x->f;
    }

    this->roots = H->roots;
    this->min = H->roots;
    BinNode* cur = this->roots;
    while (cur != NULL)
    {
        if (cur->key < this->min->key) this->min = cur;
        cur = cur->f;
    }
}

BinNode* BinomialHeap::first()
{
    return this->min;
}



BinNode* BinomialHeap::extractMin()
{
    BinNode* ret = this->first();

    // delete ret from the list of roots
    BinNode* prevX = NULL;
    BinNode* x = this->roots;
    while (x != ret)
    {
        prevX = x;
        x = x->f;
    }
    if (prevX == NULL) this->roots = x->f;
    else prevX->f = x->f;

    // reverse the list of ret's children
    BinNode* revChd = NULL;
    BinNode* cur = ret->c;
    while (cur != NULL)
    {
        BinNode* next = cur->f;
        cur->f = revChd;
        revChd = cur;
        cur = next;
    }

    // merge the two lists
    BinomialHeap* H = new BinomialHeap();
    H->roots = revChd;
    this->merge(H);

    return ret;
```

```
}
```

Функция «main» ~ состоит из цикла, в котором происходит генерация бинарной и биномиальной кучи, состоящих из 10^(3 + i) элементов, где i это номер серии теста. В каждой серии проводится 1000 операций поиска минимума, удаления минимуму и вставки элементов и замеряется время каждой операции, как суммарное, так и на 1 операцию. Рассчитывается максимальное время на одну операцию. Полученные значения выводятся в файл «Time.txt»

```
int main()
{

    srand(time(0));
    setlocale(LC_ALL, "Rus");

    const int NUM_TEST = 5;
    const int NUM_OPERATIONS = 1000;

    vector<vector<double>> time_search_binary(NUM_TEST, vector<double>(NUM_OPERATIONS));
    vector<vector<double>> time_search_binomial(NUM_TEST, vector<double>(NUM_OPERATIONS));
    vector<double> sum_time_search_binary(NUM_TEST);
    vector<double> sum_time_search_binomial(NUM_TEST);

    vector<vector<double>> time_insert_binary(NUM_TEST, vector<double>(NUM_OPERATIONS));
    vector<vector<double>> time_insert_binomial(NUM_TEST, vector<double>(NUM_OPERATIONS));
    vector<double> sum_time_insert_binary(NUM_TEST);
    vector<double> sum_time_insert_binomial(NUM_TEST);

    vector<vector<double>> time_remove_binary(NUM_TEST, vector<double>(NUM_OPERATIONS));
    vector<vector<double>> time_remove_binomial(NUM_TEST, vector<double>(NUM_OPERATIONS));
    vector<double> sum_time_remove_binary(NUM_TEST);
    vector<double> sum_time_remove_binomial(NUM_TEST);

    ofstream tout("Time1.txt");

    if (tout.is_open()) {
        for (int i = 0; i < NUM_TEST; i++)
        {
            int num_el = pow(10, 3 + i);

            BinomialHeap binomial;
            BinaryHeap binary(num_el);

            for (int i = 0; i < num_el; i++)
            {
                int num = numGenerator(-num_el, num_el);
                binary.insertKey(num);
                binomial.insert(num);
            }

            //ПОИСК МИНИМАЛЬНОГО
            for (int j = 0; j < NUM_OPERATIONS; j++)
            {
                chrono::high_resolution_clock::time_point start_search_binary =
chrono::high_resolution_clock::now();
                binary.getMin();
                chrono::high_resolution_clock::time_point end_search_binary =
chrono::high_resolution_clock::now();
                chrono::duration<double, milli> milli_diff_search_binary =
end_search_binary - start_search_binary;
                time_search_binary[i][j] = milli_diff_search_binary.count();
                sum_time_search_binary[i] += milli_diff_search_binary.count();
```

```
                chrono::high_resolution_clock::time_point start_search_binomial =
chrono::high_resolution_clock::now();
                binomial.getMin();
                chrono::high_resolution_clock::time_point end_search_binomial =
chrono::high_resolution_clock::now();
                chrono::duration<double, milli> milli_diff_search_binomial =
end_search_binomial - start_search_binomial;
                time_search_binomial[i][j] = milli_diff_search_binomial.count();
                sum_time_search_binomial[i] += milli_diff_search_binomial.count();
            }

            //УДАЛЕНИЕ МИНИМАЛЬНОГО
            for (int j = 0; j < NUM_OPERATIONS; j++)
            {
                chrono::high_resolution_clock::time_point start_remove_binary =
chrono::high_resolution_clock::now();
                binary.extractMin();
                chrono::high_resolution_clock::time_point end_remove_binary =
chrono::high_resolution_clock::now();
                chrono::duration<double, milli> milli_diff_remove_binary =
end_remove_binary - start_remove_binary;
                time_remove_binary[i][j] = milli_diff_remove_binary.count();
                sum_time_remove_binary[i] += milli_diff_remove_binary.count();

                chrono::high_resolution_clock::time_point start_remove_binomial =
chrono::high_resolution_clock::now();
                binomial.extractMin();
                chrono::high_resolution_clock::time_point end_remove_binomial =
chrono::high_resolution_clock::now();
                chrono::duration<double, milli> milli_diff_remove_binomial =
end_remove_binomial - start_remove_binomial;
                time_remove_binomial[i][j] = milli_diff_remove_binomial.count();
                sum_time_remove_binomial[i] += milli_diff_remove_binomial.count();
            }

            //ВСТАВКА
            for (int j = 0; j < NUM_OPERATIONS; j++)
            {
                int num = numGenerator(-num_el, num_el);

                chrono::high_resolution_clock::time_point start_insert_binary =
chrono::high_resolution_clock::now();
                binary.insertKey(num);
                chrono::high_resolution_clock::time_point end_insert_binary =
chrono::high_resolution_clock::now();
                chrono::duration<double, milli> milli_diff_insert_binary =
end_insert_binary - start_insert_binary;
                time_insert_binary[i][j] = milli_diff_insert_binary.count();
                sum_time_insert_binary[i] += milli_diff_insert_binary.count();

                chrono::high_resolution_clock::time_point start_insert_binomial =
chrono::high_resolution_clock::now();
                binomial.insert(num);
                chrono::high_resolution_clock::time_point end_insert_binomial =
chrono::high_resolution_clock::now();
                chrono::duration<double, milli> milli_diff_insert_binomial =
end_insert_binomial - start_insert_binomial;
                time_insert_binomial[i][j] = milli_diff_insert_binomial.count();
                sum_time_insert_binomial[i] += milli_diff_insert_binomial.count();
            }


            double max_time_search_binary = time_search_binary[i][0];
            double max_time_search_binomial = time_search_binomial[i][0];
```

```
            double max_time_remove_binary = time_remove_binary[i][0];
            double max_time_remove_binomial = time_remove_binomial[i][0];
            double max_time_insert_binary = time_insert_binary[i][0];
            double max_time_insert_binomial = time_insert_binomial[i][0];

            for (int j = 0; j < NUM_OPERATIONS; j++)
            {
                if (max_time_search_binary < time_search_binary[i][j])
                    max_time_search_binary = time_search_binary[i][j];
                if (max_time_search_binomial < time_search_binomial[i][j])
                    max_time_search_binomial = time_search_binomial[i][j];
                if (max_time_remove_binary < time_remove_binary[i][j])
                    max_time_remove_binary = time_remove_binary[i][j];
                if (max_time_remove_binomial < time_remove_binomial[i][j])
                    max_time_remove_binomial = time_remove_binomial[i][j];
                if (max_time_insert_binary < time_insert_binary[i][j])
                    max_time_insert_binary = time_insert_binary[i][j];
                if (max_time_insert_binomial < time_insert_binomial[i][j])
                    max_time_insert_binomial = time_insert_binomial[i][j];
            }

            tout << "Тест " << i + 1 << endl;
            tout << "Число элементов: " << num_el << endl;
            tout << "\nВремя 1000 операций (мс):" << endl;
            tout << "Поиск минимума:" << endl;
            tout << "Бинарная: " << sum_time_search_binary[i] << endl;
            tout << "Биномиальная: " << sum_time_search_binomial[i] << endl;
            tout << "Удаление минимума:" << endl;
            tout << "Бинарная: " << sum_time_remove_binary[i] << endl;
            tout << "Биномиальная: " << sum_time_remove_binomial[i] << endl;
            tout << "Вставка:" << endl;
            tout << "Бинарная: " << sum_time_insert_binary[i] << endl;
            tout << "Биномиальная: " << sum_time_insert_binomial[i] << endl;

            tout << "\nСреднее время 1 операции (мс):" << endl;
            tout << "Поиск минимума:" << endl;
            tout << "Бинарная: " << sum_time_search_binary[i] / NUM_OPERATIONS << endl;
            tout << "Биномиальная: " << sum_time_search_binomial[i] / NUM_OPERATIONS <<
endl;
            tout << "Удаление минимума:" << endl;
            tout << "Бинарная: " << sum_time_remove_binary[i] / NUM_OPERATIONS << endl;
            tout << "Биномиальная: " << sum_time_remove_binomial[i] / NUM_OPERATIONS <<
endl;
            tout << "Вставка:" << endl;
            tout << "Бинарная: " << sum_time_insert_binary[i] / NUM_OPERATIONS << endl;
            tout << "Биномиальная: " << sum_time_insert_binomial[i] / NUM_OPERATIONS <<
endl;

            tout << "\nМаксимальное время 1 операции (мс):" << endl;
            tout << "Поиск минимума:" << endl;
            tout << "Бинарная: " << max_time_search_binary << endl;
            tout << "Биномиальная: " << max_time_search_binomial << endl;
            tout << "Удаление минимума:" << endl;
            tout << "Бинарная: " << max_time_remove_binary << endl;
            tout << "Биномиальная: " << max_time_remove_binomial << endl;
            tout << "Вставка:" << endl;
            tout << "Бинарная: " << max_time_insert_binary << endl;
            tout << "Биномиальная: " << max_time_insert_binomial << endl << endl;
            tout << "-------------------------------------------------------------------
----" << endl << endl;

        }

        tout.close();
    }
```

Примерный результат работы программы:

```
Тест 1
Число элементов: 1000

Время 1000 операций (мс):
Поиск минимума:
Бинарная: 0.2001
Биномиальная: 0.2257
Удаление минимума:
Бинарная: 1.1319
Биномиальная: 1.7334
Вставка:
Бинарная: 0.3876
Биномиальная: 1.4483

Среднее время 1 операции (мс):
Поиск минимума:
Бинарная: 0.0002001
Биномиальная: 0.0002257
Удаление минимума:
Бинарная: 0.0011319
Биномиальная: 0.0017334
Вставка:
Бинарная: 0.0003876
Биномиальная: 0.0014483

Максимальное время 1 операции (мс):
Поиск минимума:
Бинарная: 0.001
Биномиальная: 0.0026
Удаление минимума:
Бинарная: 0.0106
Биномиальная: 0.0357
Вставка:
Бинарная: 0.0017
Биномиальная: 0.0526
```

Рисунок 1 - Результат из файла "Time1.txt"

	Время 1000 операций (мс)					
	Поиск минимума		Удаление минимума		Вставка	
Число элементов	Бинарная	Биномиальная	Бинарная	Биномиальная	Бинарная	Биномиальная
1000	0,2001	0,2257	1,1319	1,7334	0,3876	1,4483
10000	0,2275	0,2569	1,6848	2,3624	0,4224	1,6683
100000	0,1982	0,2214	2,1962	2,6988	0,4132	1,707
1000000	0,1981	0,2211	3,3889	3,8304	0,4411	1,9184

Таблица 1 – Время (мс) выполнения 1000 операций

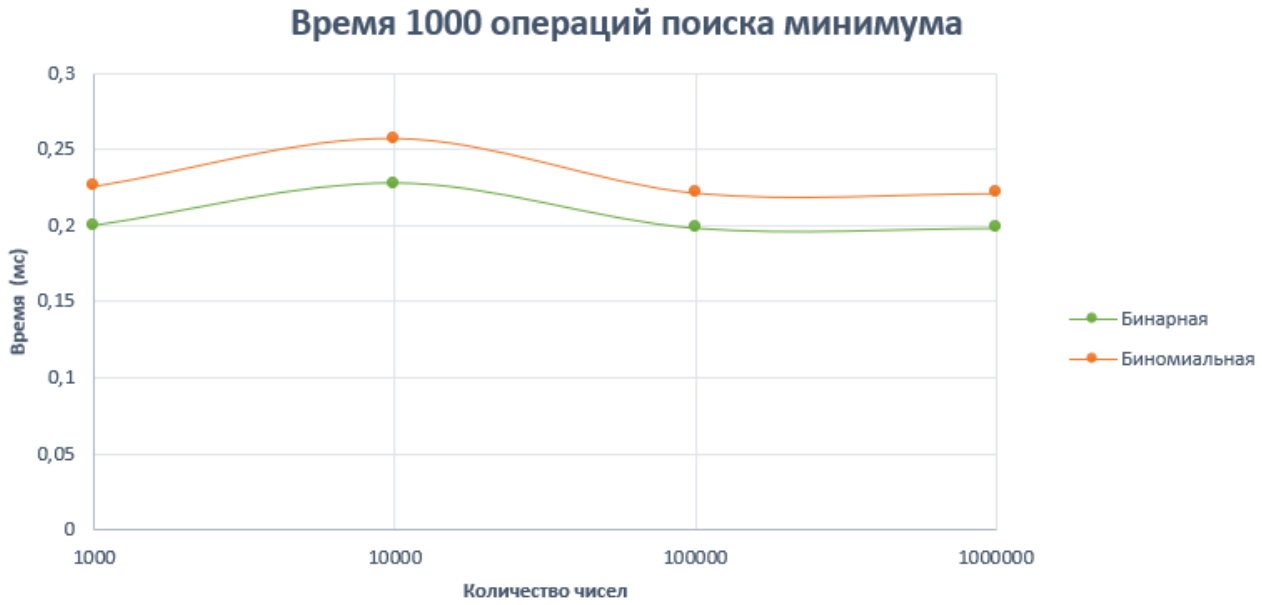

Рисунок 2 – График суммарного времени выполнения 1000 операций поиска минимума в зависимости от количества элементов кучи.

Рисунок 3 – График суммарного времени выполнения 1000 операций удаления минимума в зависимости от количества элементов кучи.

Рисунок 4 – График суммарного времени выполнения 1000 операций вставки в зависимости от количества элементов кучи.

	Среднее время 1 операции (мс)					
	Поиск минимума		Удаление минимума		Вставка	
Число элементов	Бинарная	Биномиальная	Бинарная	Биномиальная	Бинарная	Биномиальная
1000	0,0002001	0,0002257	0,0011319	0,0017334	0,0003876	0,0014483
10000	0,0002275	0,0002569	0,0016848	0,0023624	0,0004224	0,0016683
100000	0,0001982	0,0002214	0,0021962	0,0026988	0,0004132	0,001707
1000000	0,0001981	0,0002211	0,0033889	0,0038304	0,0004411	0,0019184

Таблица 2 – Среднее время (мс) выполнения 1 операции.

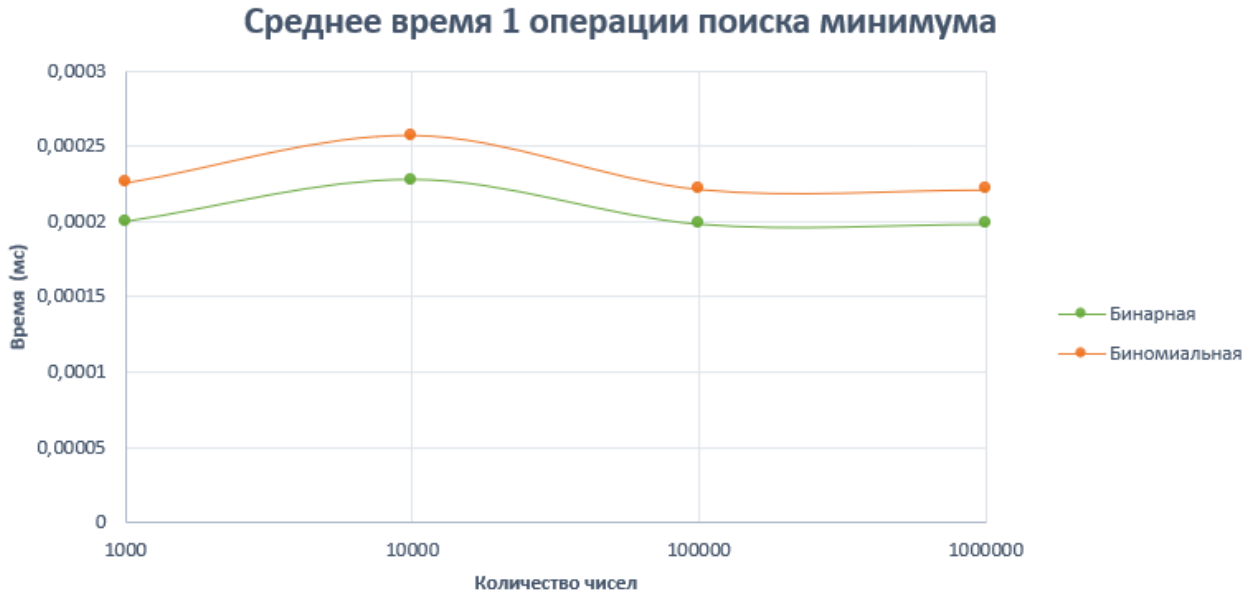

Рисунок 5 – График среднего времени выполнения 1 операции поиска минимума в зависимости от количества элементов кучи.

Рисунок 6 – График среднего времени выполнения 1 операции удаления минимума в зависимости от количества элементов кучи.

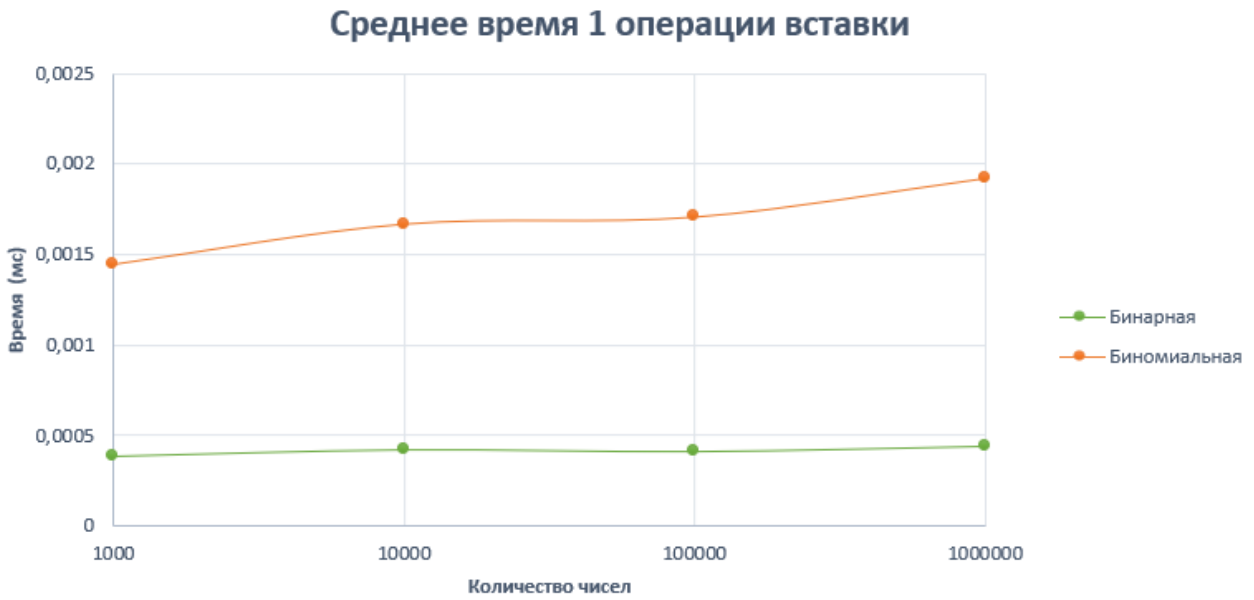

Рисунок 7 – График среднего времени выполнения 1 операции вставки в зависимости от количества элементов кучи.

	Максимальное время 1 операции (мс)					
	Поиск минимума		Удаление минимума		Вставка	
Число элементов	Бинарная	Биномиальная	Бинарная	Биномиальная	Бинарная	Биномиальная
1000	0,001	0,0026	0,0106	0,0357	0,0017	0,0526
10000	0,0031	0,0042	0,0058	0,0677	0,0017	0,1004
100000	0,001	0,0005	0,0055	0,0452	0,006	0,0726
1000000	0,0011	0,0005	0,0145	0,0614	0,0021	0,082

Таблица 3 – Максимальное время (мс) выполнения 1 операции.

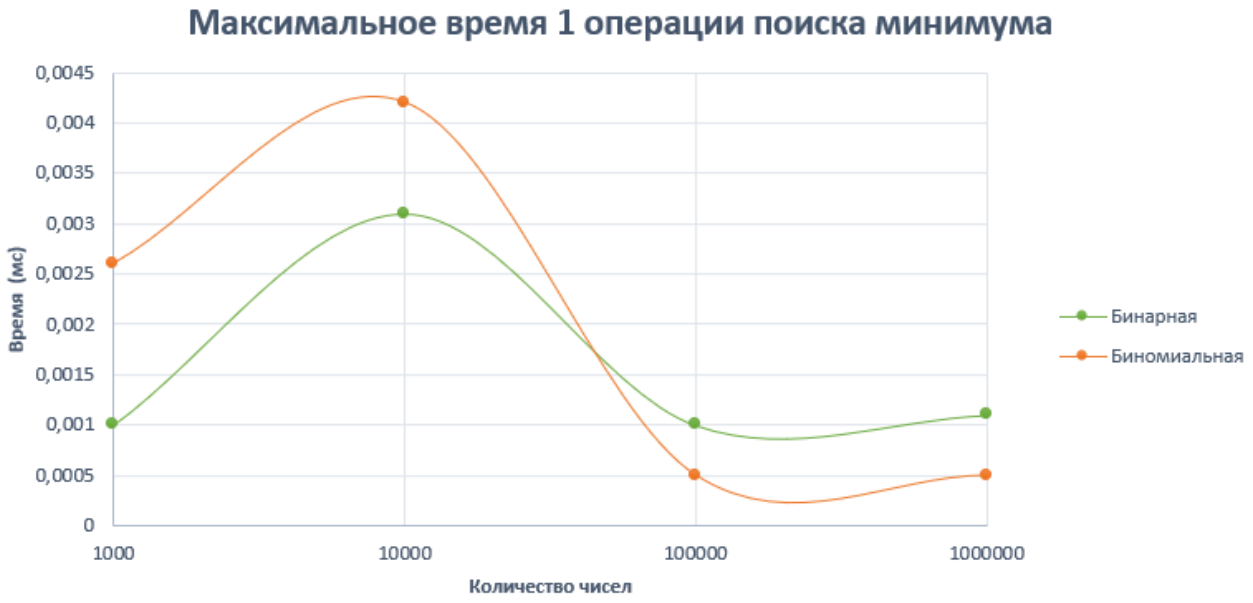

Рисунок 8 – График максимального времени выполнения 1 операции поиска минимума в зависимости от количества элементов кучи.

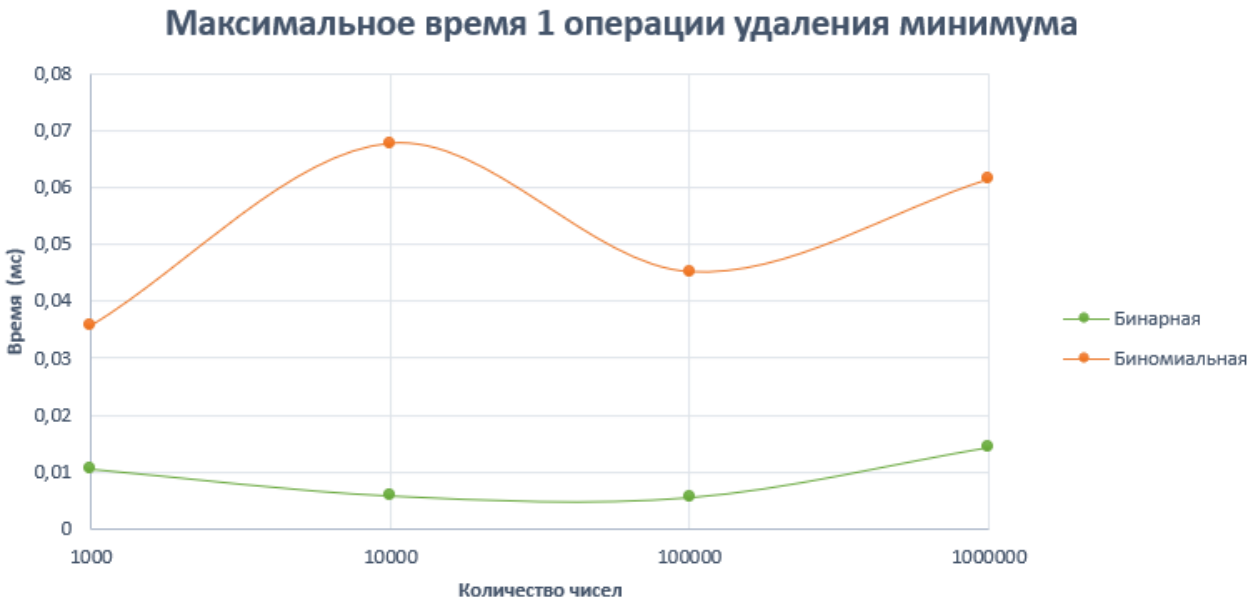

Рисунок 9 – График максимального времени выполнения 1 операции удаления минимума в зависимости от количества элементов кучи.

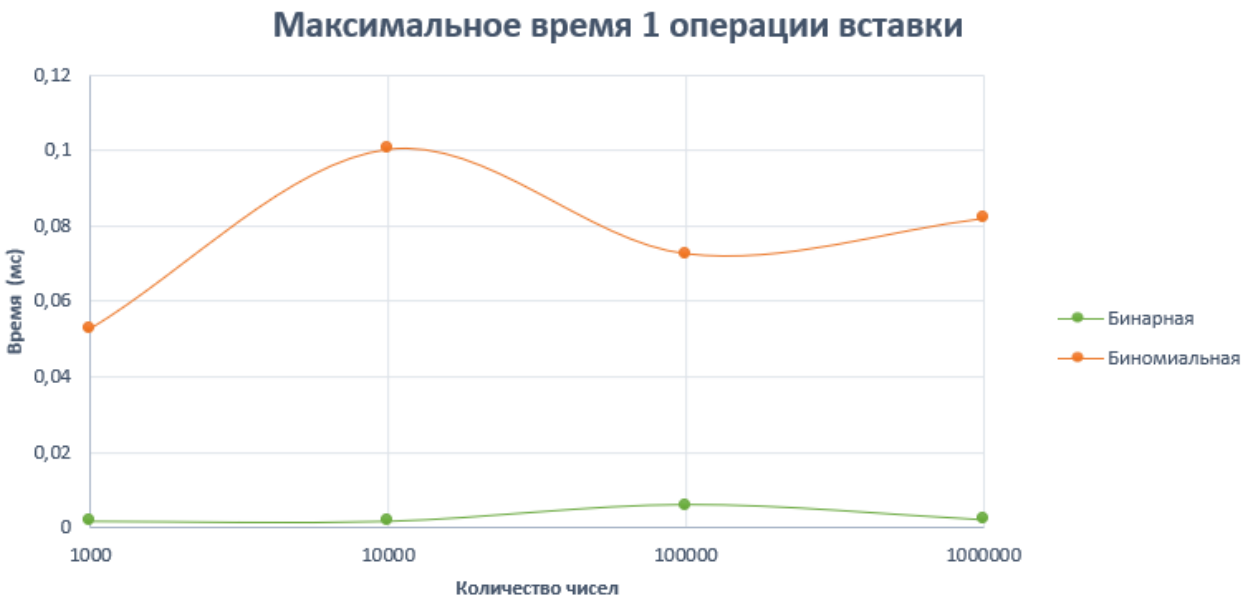

Рисунок 10 – График максимального времени выполнения 1 операции вставки в зависимости от количества элементов кучи.

Заключение

В этой лабораторной работе мне удалось реализовать бинарную и биномиальную кучи. Анализируя полученные графики времени выполнения 1000 операций, видно, что разница во времени поиска минимального элемента несущественна и не зависит от количества элементов кучи, поскольку минимальный элемент кучи всегда находится на вершине кучи. Графики времени удаления минимального элемента имеют похожую форму, однако разница между кучами уже более существенна, удаление минимума из биномиальной кучи занимает больше времени, также заметно, что с увеличением количества элементов куч – увеличивается и время удаления. На графиках времени вставки наблюдается существенная различие по скорости, вставка элемента в бинарной куче занимает примерно в 3 раза меньше времени, чем вставка в биномиальную, также имеется несущественная зависимость во времени от количества элементов кучи. В целом, практически во всех операциях, бинарная куча оказалась быстрее биномиальной.