

Министерство науки и высшего образования Российской Федерации
Федеральное государственное бюджетное образовательное учреждение
высшего образования «Российский химико-технологический университет
имени Д.И. Менделеева»

Факультет цифровых технологий и химического инжиниринга

Кафедра информационных компьютерных технологий

**ОТЧЕТ ПО ЛАБОРАТОРНОЙ РАБОТЕ № 8
ПО КУРСУ**

«АЛГОРИТМЫ И СТРУКТУРЫ ДАННЫХ»:

СТУДЕНТ группы КС-33

Костяева К.С.

**Москва
2024**

ОГЛАВЛЕНИЕ

ТЕОРИЯ.....	3
Биноминальное Дерево	3
Биноминальная куча	3
Вставка	3
Объединение куч.....	4
Изъятие минимума.....	4
Устройство	4
ЗАДАНИЕ.....	5
ПРАКТИЧЕСКАЯ ЧАСТЬ	6
Код:	6
Результат работы программы:.....	11
Графики:	13
ВЫВОД.....	15
Анализ результатов	15
Общий вывод.....	15

ТЕОРИЯ

Биномиальное Дерево

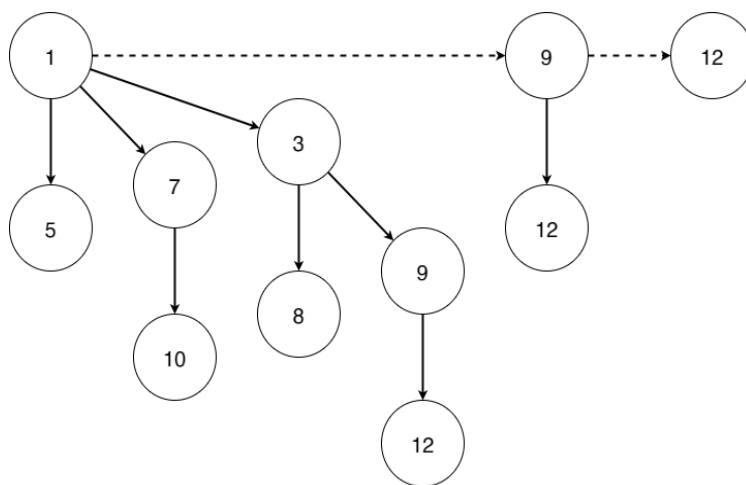
Биномиальное дерево — это такая структура данных, которая задает саму себя рекурсивно через свой предыдущий шаг.

- 1) 2^k вершин
- 2) Высота дерева k
- 3) C_i^k вершин глубины i (вот почему они называются биномиальными: C_i^k биномиальный коэффициент).
- 4) Дети корня — это B, B_{k-2}, \dots, B_0 — именно в этом порядке.
- 5) Максимальная высота вершины в биномиальном дереве $O(\log N)$

Биномиальная куча

Биномиальная куча — это множество биномиальных деревьев, со следующими ограничениями:

- 1) В каждом из биномиальных деревьев сохраняется свойство кучи.
- 2) Нет двух деревьев одинакового размера
- 3) Деревья упорядочены по размеру.



Биномиальная куча обладает двумя свойствами:

- ключ каждой вершины не меньше ключа её родителя;
- все биномиальные деревья имеют разный размер.

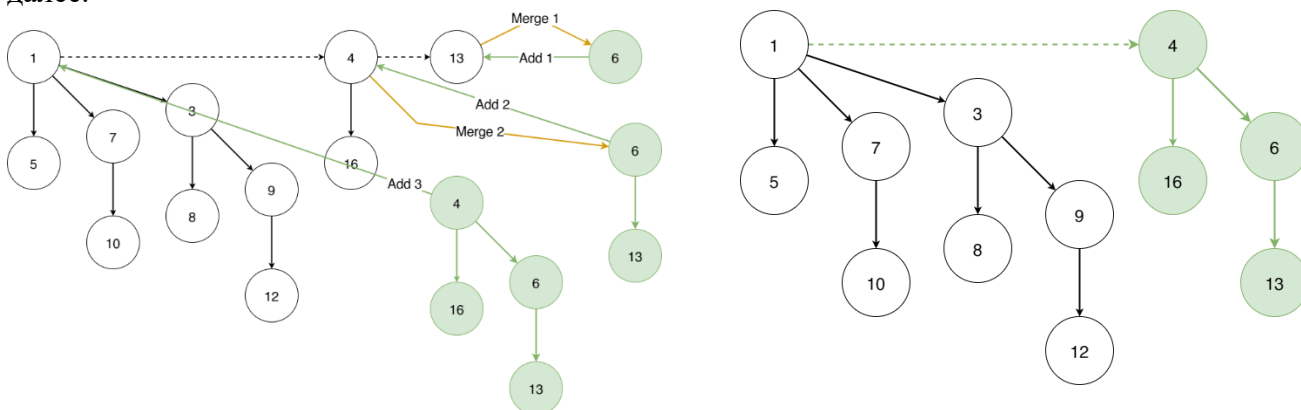
Вставка

При вставке в такую кучу нам потребуется сделать следующее:

Представить вставляемый элемент как дерево B_0 .

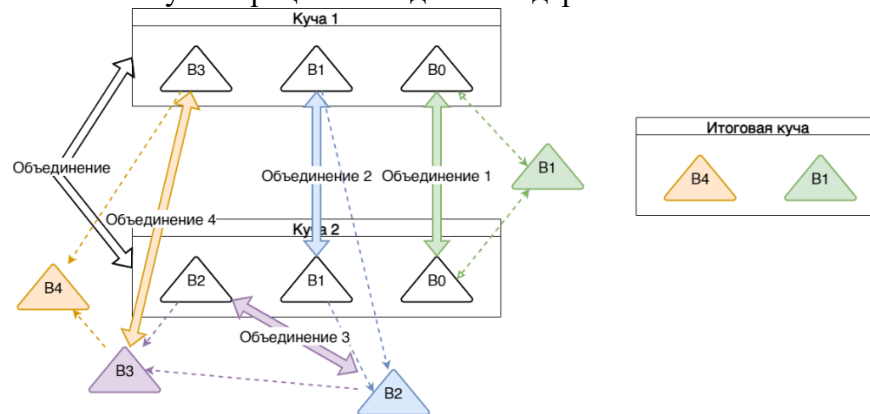
При попытке вставки в кучу, в которой уже есть дерево порядка B_0 провести операцию объединения деревьев.

Если после объединения все еще будут существовать деревья одного порядка, объединять их далее.



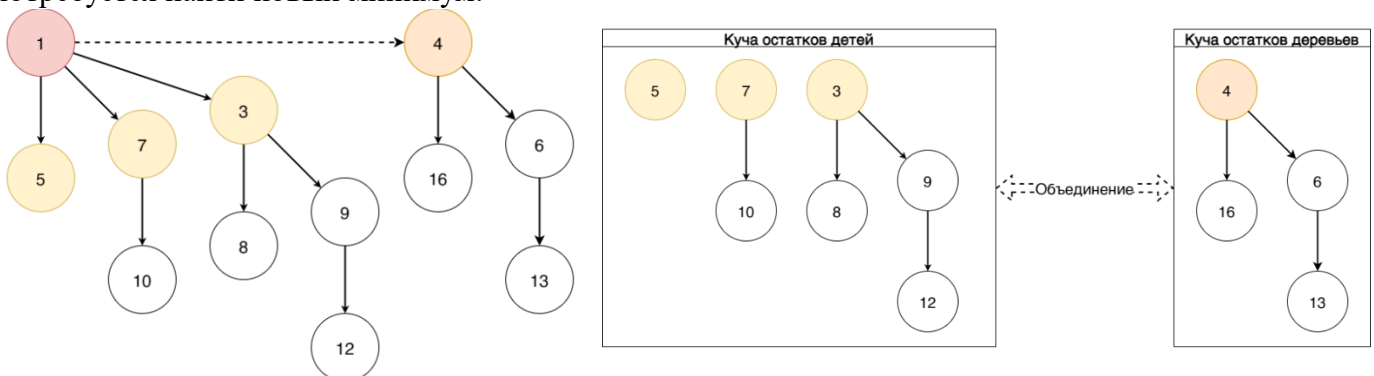
Объединение куч

Для объединения биномиальных куч, мы выполняем операцию аналогичную добавление одного элемента, постоянно используя операцию объединения деревьев.



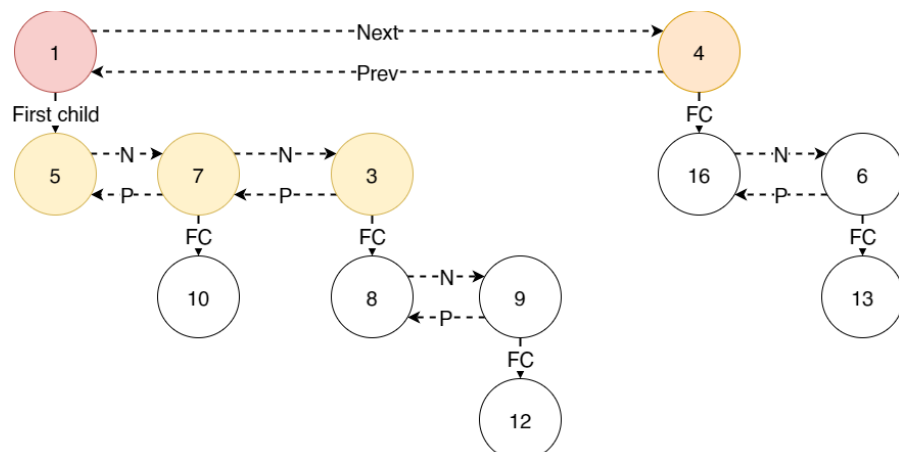
Изъятие минимума

Минимумом(максимумом) в биномиальном дереве всегда является корень. При удалении минимума из кучи появляются дети этого минимума, так как оно является корнем, таким образом, при удалении минимума нам потребуется выполнить объединение двух куч, при этом считая всех потомков минимума одной общей кучей, которая объединяется с остатком основной кучи, исключая дерево, на которое проводилось воздействие. После завершения объединения, потребуется найти новый минимум.



Устройство

Биномиальная куча хранится и организуется в виде знаний у каждого узла о своем предыдущем и следующем брате, а также крайнем левом ребенке, что позволит легко двигаться по куче в любом направлении.



ЗАДАНИЕ

вариант 2

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

1. 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:
 - i. 1000 раз найти минимум/максимум
 - ii. 1000 раз удалить минимум/максимум
 - iii. 1000 раз добавить новый элемент в кучуДля 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.

ПРАКТИЧЕСКАЯ ЧАСТЬ

Код:

язык программирования C++

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

using namespace std;
using namespace std::chrono;

// Бинарная куча (min-heap)
class BinaryHeap {
private:
    vector<int> heap;

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

    void heapifyDown(int index) {
        int left, right, smallest;
        while (true) {
            left = 2 * index + 1;
            right = 2 * index + 2;
            smallest = index;

            if (left < heap.size() && heap[left] < heap[smallest]) {
                smallest = left;
            }
            if (right < heap.size() && heap[right] < heap[smallest]) {
                smallest = right;
            }

            if (smallest != index) {
                swap(heap[index], heap[smallest]);
                index = smallest;
            }
            else {
                break;
            }
        }
    }

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

    int getMin() {
        if (heap.empty()) return INT_MAX;
    }
};
```

```

        return heap[0];
    }

    int extractMin() {
        if (heap.empty()) return INT_MAX;
        int min = heap[0];
        heap[0] = heap.back();
        heap.pop_back();
        heapifyDown(0);
        return min;
    }

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

    size_t size() {
        return heap.size();
    }
};

// Узел биномиальной кучи
struct BinomialNode {
    int key;
    int degree;
    BinomialNode* child;
    BinomialNode* sibling;
    BinomialNode* parent;

    BinomialNode(int k) : key(k), degree(0), child(nullptr), sibling(nullptr),
parent(nullptr) {}
};

// Биномиальная куча (min-heap)
class BinomialHeap {
private:
    list<BinomialNode*> trees;

    BinomialNode* mergeTrees(BinomialNode* a, BinomialNode* b) {
        if (a->key > b->key) {
            swap(a, b);
        }
        b->parent = a;
        b->sibling = a->child;
        a->child = b;
        a->degree++;
        return a;
    }

    void mergeHeaps(list<BinomialNode*>& h1, list<BinomialNode*>& h2) {
        auto it1 = h1.begin();
        auto it2 = h2.begin();
        list<BinomialNode*> result;

        while (it1 != h1.end() && it2 != h2.end()) {
            if ((*it1)->degree <= (*it2)->degree) {
                result.push_back(*it1);
                ++it1;
            }
            else {
                result.push_back(*it2);
                ++it2;
            }
        }

        while (it1 != h1.end()) {
            result.push_back(*it1);
            ++it1;
        }
    }
};

```

```

    }

    while (it2 != h2.end()) {
        result.push_back(*it2);
        ++it2;
    }

    h1 = result;
}

void adjustHeap() {
    if (trees.empty()) return;
    auto it1 = trees.begin();
    auto it2 = it1;
    auto it3 = it1;

    if (trees.size() == 1) {
        it2 = trees.end();
        it3 = trees.end();
    }
    else {
        it2++;
        it3 = it2;
        it3++;
    }

    while (it1 != trees.end()) {
        if (it2 == trees.end()) {
            it1++;
        }
        else if ((*it1)->degree < (*it2)->degree) {
            it1++;
            it2++;
            if (it3 != trees.end()) it3++;
        }
        else if (it3 != trees.end() &&
            (*it1)->degree == (*it2)->degree &&
            (*it1)->degree == (*it3)->degree) {
            it1++;
            it2++;
            it3++;
        }
        else if ((*it1)->degree == (*it2)->degree) {
            *it1 = mergeTrees(*it1, *it2);
            it2 = trees.erase(it2);
            if (it3 != trees.end()) it3++;
        }
    }
}

BinomialNode* findMinNode() {
    if (trees.empty()) return nullptr;
    BinomialNode* minNode = *trees.begin();
    for (auto tree : trees) {
        if (tree->key < minNode->key) {
            minNode = tree;
        }
    }
    return minNode;
}

public:
    void insert(int key) {
        BinomialHeap tempHeap;
        tempHeap.trees.push_back(new BinomialNode(key));
        mergeHeaps(trees, tempHeap.trees);
        adjustHeap();
    }
}

```



```

int getMin() {
    BinomialNode* minNode = findMinNode();
    if (!minNode) return INT_MAX;
    return minNode->key;
}

int extractMin() {
    BinomialNode* minNode = findMinNode();
    if (!minNode) return INT_MAX;

    trees.remove(minNode);

    BinomialHeap childHeap;
    BinomialNode* child = minNode->child;
    while (child) {
        BinomialNode* next = child->sibling;
        child->sibling = nullptr;
        child->parent = nullptr;
        childHeap.trees.push_front(child);
        child = next;
    }

    mergeHeaps(trees, childHeap.trees);
    adjustHeap();

    int minKey = minNode->key;
    delete minNode;
    return minKey;
}

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

size_t size() {
    size_t count = 0;
    for (auto tree : trees) {
        int degree = tree->degree;
        count += (1 << degree);
    }
    return count;
}
};

// Генератор случайных чисел
int generateRandomNumber(int min, int max) {
    static random_device rd;
    static mt19937 gen(rd());
    uniform_int_distribution<int> dis(min, max);
    return dis(gen);
}

// Тестирование кучи
template<typename Heap>
void testHeap(const string& heapName, int n, ofstream& outFile) {
    Heap heap;
    vector<int> elements;

    // Заполнение кучи
    auto start = high_resolution_clock::now();
    for (int i = 0; i < n; ++i) {
        int num = generateRandomNumber(1, 1000000);
        heap.insert(num);
        elements.push_back(num);
    }
    auto end = high_resolution_clock::now();
    auto fillTime = duration_cast<microseconds>(end - start).count();

```

```

const int batchSize = 10;

// Тест 1: 1000 операций getMin
start = high_resolution_clock::now();
long long maxOpTime = 0;
for (int i = 0; i < 1000; i += batchSize) {
    auto batchStart = high_resolution_clock::now();
    for (int j = 0; j < batchSize; ++j) {
        heap.getMin();
    }
    auto batchEnd = high_resolution_clock::now();
    auto batchTime = duration_cast<nanoseconds>(batchEnd - batchStart).count();
    long long avgBatchTime = batchTime / batchSize;
    if (avgBatchTime > maxOpTime) maxOpTime = avgBatchTime;
}
end = high_resolution_clock::now();
auto test1Time = duration_cast<microseconds>(end - start).count();
double avgTest1Time = test1Time / 1000.0;
long long getMinMaxTime = maxOpTime;

// Тест 2: 1000 операций extractMin
start = high_resolution_clock::now();
maxOpTime = 0;
for (int i = 0; i < 1000; i += batchSize) {
    auto batchStart = high_resolution_clock::now();
    for (int j = 0; j < batchSize; ++j) {
        heap.extractMin();
    }
    auto batchEnd = high_resolution_clock::now();
    auto batchTime = duration_cast<nanoseconds>(batchEnd - batchStart).count();
    long long avgBatchTime = batchTime / batchSize;
    if (avgBatchTime > maxOpTime) maxOpTime = avgBatchTime;
}
end = high_resolution_clock::now();
auto test2Time = duration_cast<microseconds>(end - start).count();
double avgTest2Time = test2Time / 1000.0;
long long extractMinMaxTime = maxOpTime;

// Тест 3: 1000 операций insert
start = high_resolution_clock::now();
maxOpTime = 0;
for (int i = 0; i < 1000; i += batchSize) {
    auto batchStart = high_resolution_clock::now();
    for (int j = 0; j < batchSize; ++j) {
        int num = generateRandomNumber(1, 1000000);
        heap.insert(num);
    }
    auto batchEnd = high_resolution_clock::now();
    auto batchTime = duration_cast<nanoseconds>(batchEnd - batchStart).count();
    long long avgBatchTime = batchTime / batchSize;
    if (avgBatchTime > maxOpTime) maxOpTime = avgBatchTime;
}
end = high_resolution_clock::now();
auto test3Time = duration_cast<microseconds>(end - start).count();
double avgTest3Time = test3Time / 1000.0;
long long insertMaxTime = maxOpTime;

// Запись результатов в CSV
outFile << heapName << "," << n << "," << fillTime << ","
    << test1Time << "," << avgTest1Time << "," << getMinMaxTime << ","
    << test2Time << "," << avgTest2Time << "," << extractMinMaxTime << ","
    << test3Time << "," << avgTest3Time << "," << insertMaxTime << "\n";
}

int main() {
    ofstream outFile("heap_results.csv");
    if (!outFile.is_open()) {

```

```

        cerr << "Failed to open output file." << endl;
        return 1;
    }

    // Заголовок CSV
    outFile << "HeapType,N,FillTime(us), "
        << "GetMinTotalTime(us),GetMinAvgTime(us),GetMinMaxTime(us), "
        << "ExtractMinTotalTime(us),ExtractMinAvgTime(us),ExtractMinMaxTime(us), "
        << "InsertTotalTime(us),InsertAvgTime(us),InsertMaxTime(us)\n";

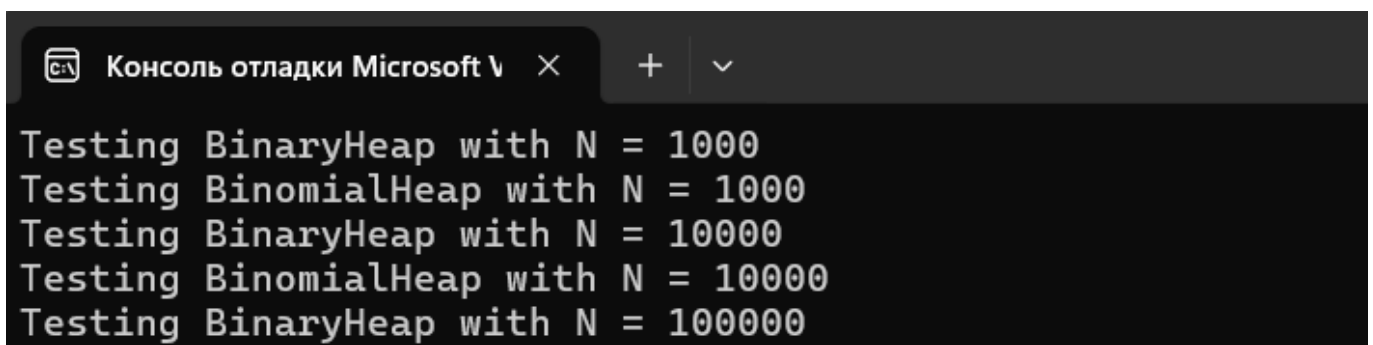
    // Тестирование для разных N
    vector<int> testSizes = { 1000, 10000, 100000, 1000000, 10000000 };
    for (int n : testSizes) {
        cout << "Testing BinaryHeap with N = " << n << endl;
        testHeap<BinaryHeap>("BinaryHeap", n, outFile);

        if (n <= 100000) { // Биномиальная куча слишком медленная для больших N
            cout << "Testing BinomialHeap with N = " << n << endl;
            testHeap<BinomialHeap>("BinomialHeap", n, outFile);
        }
    }

    outFile.close();
    cout << "Testing complete. Results saved to heap_results.csv" << endl;
    return 0;
}

```

Результат работы программы:



```

Консоль отладки Microsoft V  +  v
Testing BinaryHeap with N = 1000
Testing BinomialHeap with N = 1000
Testing BinaryHeap with N = 10000
Testing BinomialHeap with N = 10000
Testing BinaryHeap with N = 100000

```

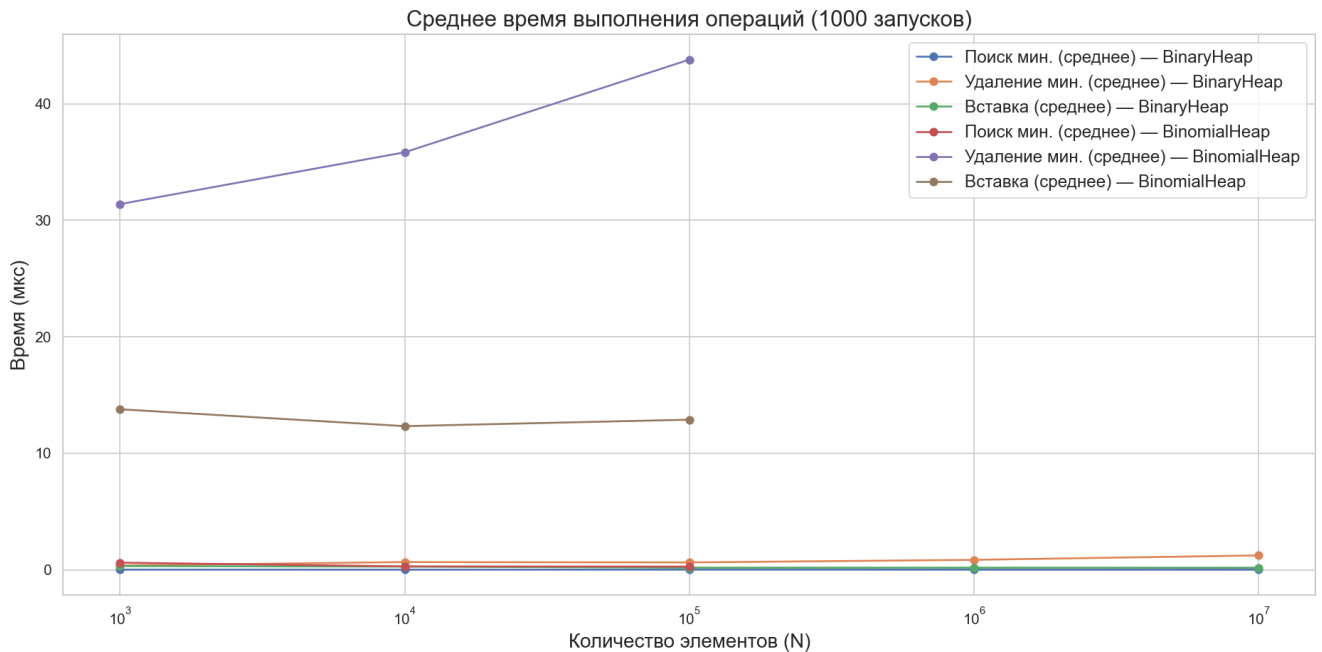
вывод результатов в файл

	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V
1	HeapType,N,FillTime(us),GetMinTotalTime(us),GetMinAvgTime(us),GetMinMaxTime(us),ExtractMinTotalTime(us),ExtractMinAvgTime(us),ExtractMinMaxTime(us),InsertTotalTime(us),InsertAvgTime(us),InsertMaxTime(us)																					
2	BinaryHeap,1000,2487,29,0.029,70,646,0.646,3020,377,0.377,3470																					
3	BinomialHeap,1000,15380,527,0.527,1250,28374,28.374,62960,10426,10.426,15780																					
4	BinaryHeap,10000,2566,17,0.017,20,508,0.508,1080,206,0.206,580																					
5	BinomialHeap,10000,120821,257,0.257,320,35320,35.32,52180,11962,11.962,15750																					
6	BinaryHeap,100000,24356,17,0.017,30,614,0.614,1070,190,0.19,580																					
7	BinomialHeap,100000,1341757,289,0.289,790,43555,43.555,69020,12418,12.418,16100																					
8	BinaryHeap,1000000,240890,18,0.018,40,6574,6.574,571210,188,0.188,570																					
9	BinaryHeap,10000000,2404165,17,0.017,40,1325,1.325,3240,181,0.181,600																					
10																						
11																						
12																						
13																						
14																						
15																						
16																						
17																						
18																						
19																						
20																						
21																						
22																						
23																						
24																						
25																						
26																						

Графики:

язык программирования Python

1. Среднее время выполнения операций (усреднённое по 1000 запусков)



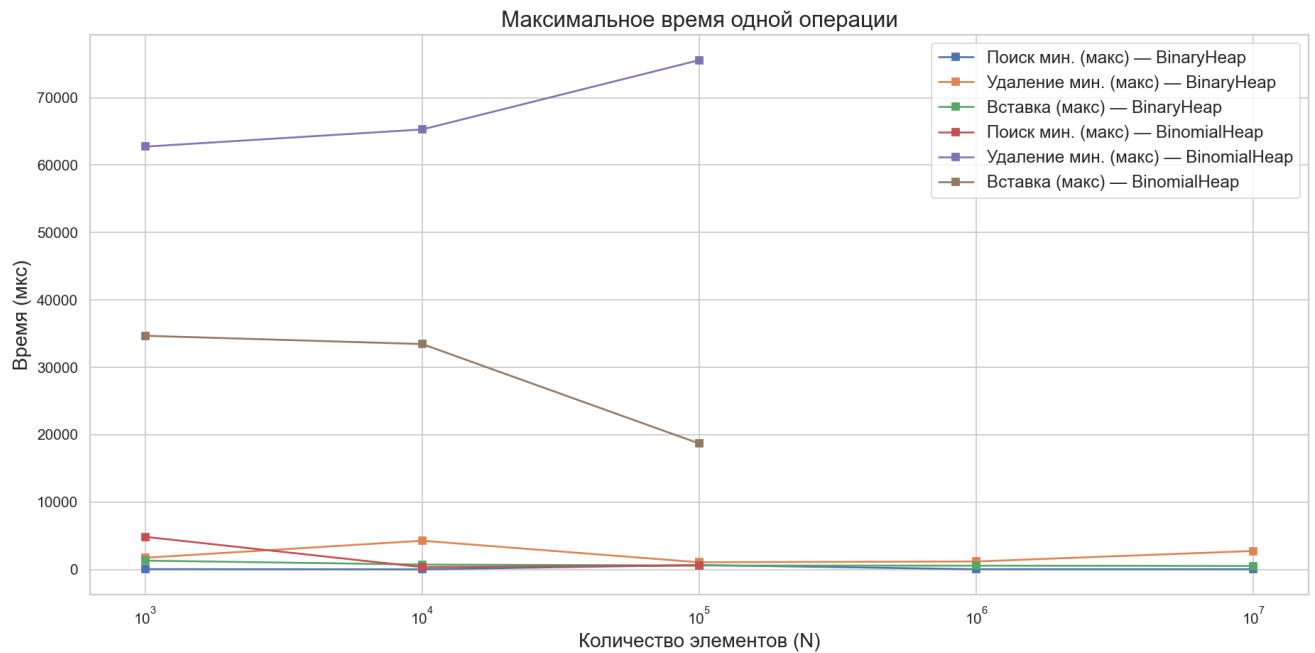
На графике:

- **Ось X** — размер кучи (N), от 10^3 до 10^7
- **Ось Y** — среднее время выполнения операции в микросекундах (us)

Что видно:

- У **Insert(вставка)** биномиальной кучи время растёт заметно медленнее, чем у бинарной.
- **ExtractMin(удаление мин. элемента)** у биномиальной кучи тоже более эффективно при больших N.
- **GetMin(поиск мин. элемента)** — очень быстрая операция у обеих куч, почти горизонтальная линия (почти константа по времени).

2. График со всеми операциями для максимального времени



На этом графике:

- **Ось X** — размер кучи (N)
- **Ось Y** — максимальное время одной операции (наихудший случай за 1000 запусков)

Что видно:

- **Insert(вставка)** в бинарной куче иногда даёт "всплески" времени — возможно, из-за перестроения дерева.
- У биномиальной кучи время наихудшего случая для **ExtractMin(удаление мин. элемента)** более стабильно.
- **GetMin(поиск мин. элемента)**, как и в первом графике, показывает минимальное и стабильное время.

ВЫВОД

В ходе выполнения задания были реализованы и протестированы две структуры данных: **бинарная куча** и **биномиальная куча**. Для каждой из них проводились измерения времени выполнения основных операций:

1. Insert (вставка элемента),
2. GetMin (получение минимального элемента),
3. ExtractMin (удаление минимального элемента).

Тестирование проводилось для различных размеров входных данных (от 10^3 до 10^7) с измерением:

- **суммарного времени** выполнения 1000 операций,
- **усреднённого времени** на одну операцию,
- **максимального времени** среди окон операций.

На основе собранных данных были построены графики, наглядно демонстрирующие зависимость времени от объёма данных.

Анализ результатов

4. GetMin:
 - Эта операция оказалась самой быстрой, особенно для бинарной кучи (в среднем $\sim O(1)$).
 - У биномиальной кучи GetMin имеет сложность $\sim O(\log N)$, однако при больших размерах данных её производительность остаётся приемлемой.
5. Insert:
 - Вставка в биномиальную кучу показала более **плавный рост времени** по сравнению с бинарной.
 - Это связано с тем, что в биномиальной куче вставка сводится к быстрой операции слияния с кучей из одного элемента.
6. ExtractMin:
 - Эта операция является самой затратной и показала наибольший рост времени, особенно для бинарной кучи.
 - У биномиальной кучи рост времени был **более пологим**, что говорит о лучшей масштабируемости структуры.

Общий вывод

- **Биномиальная куча** продемонстрировала **лучшую производительность** на больших объёмах данных, особенно по операциям Insert и ExtractMin.
- **Бинарная куча** остаётся эффективной для небольших объёмов, однако начинает проигрывать при увеличении числа элементов.
- Построенные графики подтвердили теоретические оценки сложности операций и подчеркнули важность выбора структуры данных в зависимости от задачи.

Таким образом, **биномиальные кучи являются более предпочтительными при работе с большими объёмами данных**, особенно когда важна эффективность массовых вставок и удалений.

