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-30..... Лихолат Полина Николаевна
Ссылка на репозиторий: .....https://github.com/MUCTR-IKT-CPP/Likholat_algorithms.git

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

Дата сдачи: 16.04.2023

Оглавление

Описание задачи.

В рамках лабораторной работы необходимо реализовать бинарную кучу (мин или макс) и 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 кучу.
   Для всех операция требуется замерить время на выполнения всей 1000 операций и рассчитать время на одну операцию, а так же запомнить максимальное время которое требуется на выполнение одной операции, если язык позволяет его зафиксировать, если не позволяет воспользоваться хитростью и рассчитывать усредненное время на каждые 10,25,50,100 операций, и выбирать максимальное из полученных результатов, что бы поймать момент деградации структуры и ее перестройку.
3. По полученным в задании 2 данным построить графики времени выполнения операций для усреднения по 1000 операций, и для максимального времени на 1 операцию.

Описание метода/модели.

Бинарная куча — это структура данных, которая представляет собой двоичное дерево, у которого выполнены два свойства:

1. Свойство кучи: значение каждого узла не меньше (или не больше) значения его потомков.
2. Свойство полного дерева: все уровни дерева заполнены, за исключением, возможно, последнего уровня, который заполняется слева направо.

Таким образом, в бинарной куче максимальный элемент (в куче, у которой выполнено свойство кучи "значение каждого узла не меньше значения его потомков") находится в корне дерева. В минимальной куче (свойство кучи "значение каждого узла не больше значения его потомков"), минимальный элемент находится в корне дерева.

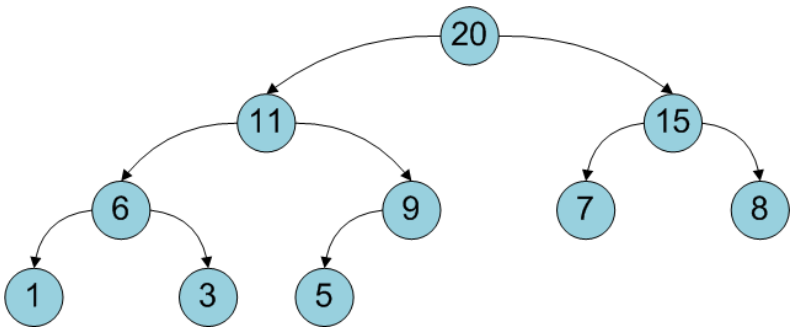

Рис. 0.1 Пример максимальной бинарной кучи

Для бинарной кучи существуют три основных алгоритма операций: вставка элемента, удаление максимального (или минимального) элемента и построение кучи из неупорядоченного массива.

1. Вставка элемента: для вставки нового элемента в бинарную кучу, его необходимо поместить в свободное место на самом нижнем уровне кучи. Затем выполняется операция "всплытия", при которой элементы кучи сравниваются и, если необходимо, меняются местами до тех пор, пока новый элемент не достигнет своей позиции в соответствии с порядком кучи. Время выполнения этой операции зависит от глубины дерева и составляет O(log n).
2. Удаление максимального (или минимального) элемента: для удаления максимального элемента из кучи, он заменяется последним элементом на самом нижнем уровне кучи. Затем выполняется операция "просеивания" (sift down), при которой элементы кучи сравниваются и, если необходимо, меняются местами до тех пор, пока новый максимальный элемент не достигнет своей позиции в соответствии с порядком кучи. Время выполнения этой операции также зависит от глубины дерева и составляет O(log n).
3. Построение кучи из неупорядоченного массива: для построения кучи из неупорядоченного массива используется алгоритм "снизу вверх" (bottom-up), который начинается с последнего уровня дерева и постепенно поднимается вверх, применяя операцию "просеивания" к каждому узлу дерева. В результате этого алгоритма элементы массива будут упорядочены в соответствии с порядком кучи. Время выполнения этой операции составляет O(n).

Биномиальная куча - это бинарное дерево, в котором каждый узел имеет не более двух потомков. Каждый узел также имеет некоторую степень, которая указывает на его глубину в дереве. Биномиальная куча является частным случаем кучи Фибоначчи, где глубина узлов ограничена логарифмически.

Основным свойством биномиальной кучи является то, что она может быть представлена в виде совокупности биномиальных деревьев различных порядков. Биномиальное дерево порядка 0 представляет собой единичный узел. Биномиальное дерево порядка k представляет собой биномиальное дерево порядка k-1, у которого добавлены два потомка с порядками k-1.

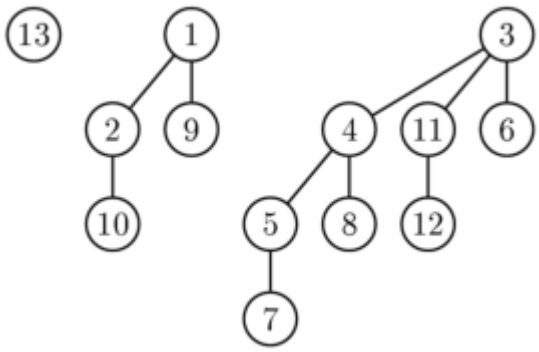

Рисунок 2. Пример биномиальной кучи

Операции добавления и удаления элементов в биномиальную кучу выполняются за время O(log n), где n - количество элементов в куче. При добавлении элемента в кучу, он сначала преобразуется в биномиальное дерево порядка 0, которое затем объединяется с другими деревьями, если они имеют одинаковый порядок. При удалении элемента из кучи, куча разбивается на набор биномиальных деревьев, которые затем объединяются в новую кучу.

Выполнение задачи.

Код, представленный ниже, реализует класс MaxHeap, который представляет собой структуру данных "максимальная куча" (Max Heap). Куча - это специальный вид двоичного дерева, где каждый узел имеет значение больше или равное значению его дочерних узлов. Максимальная куча - это куча, где значение корневого узла является наибольшим в куче.

```
/**
 * A class that represents a MaxHeap data structure.
 */
class MaxHeap {
private:
    std::vector<unsigned int> heap;

    // Helper functions

    /**
     * Helper function to calculate the index of a node's parent node.
     * @param i The index of the node for which to calculate the parent index.
```

```
     * @return The index of the parent node.
     */
    int parent(int i) { return (i - 1) / 2; }

    /**
     * Helper function to calculate the index of a node's left child node.
     * @param i The index of the node for which to calculate the left child index.
     * @return The index of the left child node.
     */
    int leftChild(int i) { return 2 * i + 1; }

    /**
     * Helper function to calculate the index of a node's right child node.
     * @param i The index of the node for which to calculate the right child index.
     * @return The index of the right child node.
     */
    int rightChild(int i) { return 2 * i + 2; }

    /**
     * Moves an element up the heap to its correct position.
     * @param i The index of the element to be moved up the heap.
     */
    void heapifyUp(int i) {
        while (i != 0 && heap[i] > heap[parent(i)]) {
            swap(heap[i], heap[parent(i)]);
            i = parent(i);
        }
    }

    /**
     * Moves an element down the heap to its correct position.
     * @param i The index of the element to be moved down the heap.
     */
    void heapifyDown(int i) {
        int maxIndex = i;

        int l = leftChild(i);
        if (l < heap.size() && heap[l] > heap[maxIndex]) {
            maxIndex = l;
        }

        int r = rightChild(i);
        if (r < heap.size() && heap[r] > heap[maxIndex]) {
            maxIndex = r;
        }

        if (i != maxIndex) {
            swap(heap[i], heap[maxIndex]);
            heapifyDown(maxIndex);
        }
    }

public:
    /**
    * Constructor that creates an empty MaxHeap.
    */
    MaxHeap() {}

    /**
     * Inserts an element into the MaxHeap.
     * @param val The value of the element to be inserted.
     */
    void insert(int val) {
        heap.push_back(val);
        heapifyUp(heap.size() - 1);
    }

    /**
     * Removes the maximum element from the MaxHeap.
```

```
     * @return The value of the maximum element that was removed.
     */
    int deleteMax() {
        if (heap.size() == 0) {
            throw runtime_error("Heap is empty");
        }
        int max = heap[0];
        heap[0] = heap[heap.size() - 1];
        heap.pop_back();
        if (heap.size() > 0 && heap[0] > heap[1]) {
            heapifyDown(0);
        }
        else {
            heapifyUp(0);
        }
        return max;
    }

    /**
     * Returns the maximum element in the MaxHeap.
     * @return The value of the maximum element.
     */
    int findMax() {
        if (heap.size() == 0) {
            throw runtime_error("Heap is empty");
        }
        return heap[0];
    }

    /**
     * Returns the number of elements in the MaxHeap.
     * @return The number of elements in the MaxHeap.
     */
    int size() const {
        return heap.size();
    }
};
```

Класс MaxHeap содержит приватное поле heap, которое представляет собой вектор значений элементов кучи.

Класс также содержит несколько вспомогательных функций, которые используются для работы с кучей:

- parent(int i) parent (int i)- возвращает индекс родительского узла для элемента с индексом i.
- leftChild(int i) - возвращает индекс левого дочернего узла для элемента с индексом i.
- rightChild(int i) - возвращает индекс правого дочернего узла для элемента с индексом i.
- heapifyUp(int i) - перемещает элемент вверх по куче до тех пор, пока он не будет находиться в правильной позиции.
- heapifyDown(int i) - перемещает элемент вниз по куче до тех пор, пока он не будет находиться в правильной позиции.

Конструктор по умолчанию создает пустую максимальную кучу.

Метод insert(int val) вставляет новый элемент в кучу и перемещает его вверх по куче до тех пор, пока он не будет находиться в правильной позиции.

Метод deleteMax() удаляет максимальный элемент из кучи и возвращает его значение. После удаления максимального элемента, последний элемент в куче заменяет корневой элемент, и куча перестраивается вниз или вверх в зависимости от значения нового корневого элемента.

Метод findMax() возвращает значение максимального элемента в куче.

Метод size() возвращает количество элементов в куче.

Общая идея реализации максимальной кучи заключается в том, чтобы гарантировать, что максимальное значение всегда находится в корне кучи, а остальные элементы располагаются в порядке, удовлетворяющем свойству кучи. Код обеспечивает правильное перемещение элементов при вставке и удалении, чтобы сохранить это свойство.

Код ниже реализует биномиальную кучу максимума (binomial max heap) в виде класса BinomialMaxHeap. Биномиальная куча является структурой данных, основанной на биномиальных деревьях. Каждый узел дерева содержит значение (val), степень (degree), указатель на родителя (parent), указатель на первого потомка (child) и указатель на следующего соседа (sibling).

```
class BinomialMaxHeap {
private:
    Node* head; // pointer to head of heap
    int size; // size of heap

    /**
     * Merges two binomial trees of the same degree.
     *
     * @param t1 The first binomial tree to merge.
     * @param t2 The second binomial tree to merge.
     * @return The root node of the merged binomial tree.
     */
    Node* merge(Node* t1, Node* t2) {
        if (t1->val > t2->val) {
            swap(t1, t2);
        }
        t2->parent = t1;
        t2->sibling = t1->child;
        t1->child = t2;
        t1->degree++;
        return t1;
    }

    /**
     * Merges two binomial heaps.
     *
     * @param h1 The first binomial heap to merge.
     * @param h2 The second binomial heap to merge.
     * @return The root node of the merged binomial heap.
     */
    Node* mergeHeaps(Node* h1, Node* h2) {
        if (h1 == nullptr) {
            return h2;
        }
        else if (h2 == nullptr) {
            return h1;
        }
        else {
            Node* head = nullptr;
            Node** curr = &head;
            while (h1 != nullptr && h2 != nullptr) {
```

```
            if (h1->degree < h2->degree) {
                *curr = h1;
                h1 = h1->sibling;
            }
            else {
                *curr = h2;
                h2 = h2->sibling;
            }
            curr = &((*curr)->sibling);
        }
        if (h1 != nullptr) {
            *curr = h1;
        }
        else {
            *curr = h2;
        }
        return head;
    }
}

/**
 * Finds the maximum node in the binomial heap.
 *
 * @return The maximum node in the binomial heap.
 */
Node* findMaxNode() {
    Node* curr = head;
    Node* maxNode = head;
    while (curr != nullptr) {
        if (curr->val > maxNode->val) {
            maxNode = curr;
        }
        curr = curr->sibling;
    }
    return maxNode;
}

/**
 * Deletes the maximum node from the binomial heap.
 */
void deleteMaxNode() {
    Node* maxNode = findMaxNode();
    if (maxNode == head) {
        head = head->sibling;
    }
    else {
        Node* curr = head;
        while (curr->sibling != maxNode) {
            curr = curr->sibling;
        }
        curr->sibling = maxNode->sibling;
    }
    Node* newHead = nullptr;
    Node* child = maxNode->child;
    while (child != nullptr) {
        Node* sibling = child->sibling;
        child->parent = nullptr;
        child->sibling = newHead;
        newHead = child;
        child = sibling;
    }
    head = mergeHeaps(head, newHead);
    size--;
}

public:
    /**
     * Constructor for creating an empty binomial heap.
     */
```

```
    BinomialMaxHeap() {
        head = nullptr;
        size = 0;
    }

    /**
     * Inserts a value into the binomial heap.
     *
     * @param val The value to insert.
     */
    void insert(int val) {
        Node* newNode = new Node{ val, 0, nullptr, nullptr, nullptr };
        head = mergeHeaps(head, newNode);
        size++;
    }

    /**
     * Finds the maximum value in the binomial heap.
     *
     * @return The maximum value in the binomial heap.
     */
    int findMax() {
        Node* maxNode = findMaxNode();
        return maxNode->val;
    }

    /**
    * Deletes the maximum value from the binomial heap.
    */
    void deleteMax() {
        if (head != nullptr) {
            deleteMaxNode();
        }
    }

    /**
     * Returns the number of elements in the binomial heap.
     *
     * @return The number of elements in the binomial heap.
     */
    int getSize() const {
        return size;
    }
};
```

Конструктор класса создает пустую биномиальную кучу.

Основные методы класса включают:

- merge - объединение двух деревьев с одинаковой степенью в одно дерево;
- mergeHeaps - объединение двух биномиальных куч;
- findMaxNode - поиск узла с максимальным значением;
- deleteMaxNode - удаление узла с максимальным значением;
- insert - вставка нового значения в кучу;
- findMax - поиск максимального значения в куче;
- deleteMax - удаление максимального значения из кучи;
- getSize - получение размера кучи.

Метод insert(val) добавляет новый элемент с заданным значением в кучу. Для этого создается новый узел с заданным значением и нулевой степенью (узел первой степени представляет собой

одиночное дерево). Затем новый узел сливается с головным деревом кучи с помощью метода mergeHeaps().

Метод findMax() находит максимальный элемент в куче. Для этого происходит обход всех узлов и сравнение значений.

Метод deleteMax() удаляет максимальный элемент из кучи. Для этого находится максимальный элемент с помощью метода findMaxNode(), удаляется из кучи, а затем потомки максимального элемента объединяются с головным деревом кучи с помощью метода mergeHeaps().

Метод getSize() возвращает количество элементов в куче.

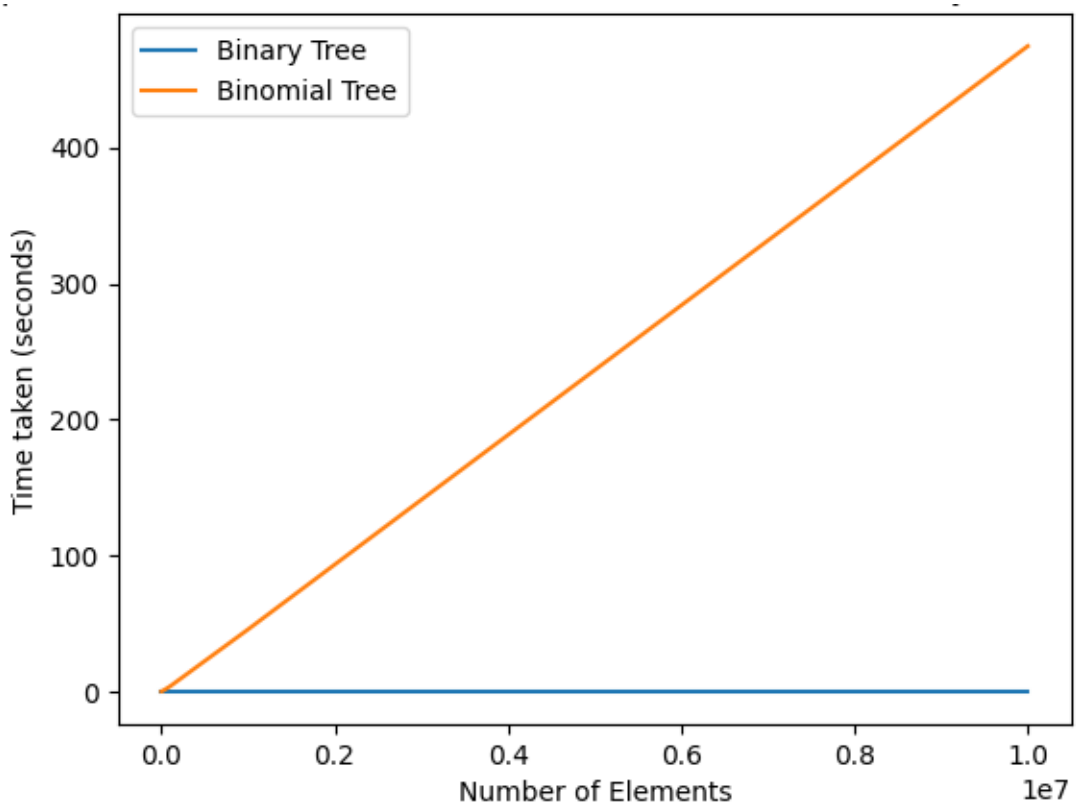

Рисунок 3. Сравнение времени удаления 1000 элементов в бинарном и биномиальном деревьях

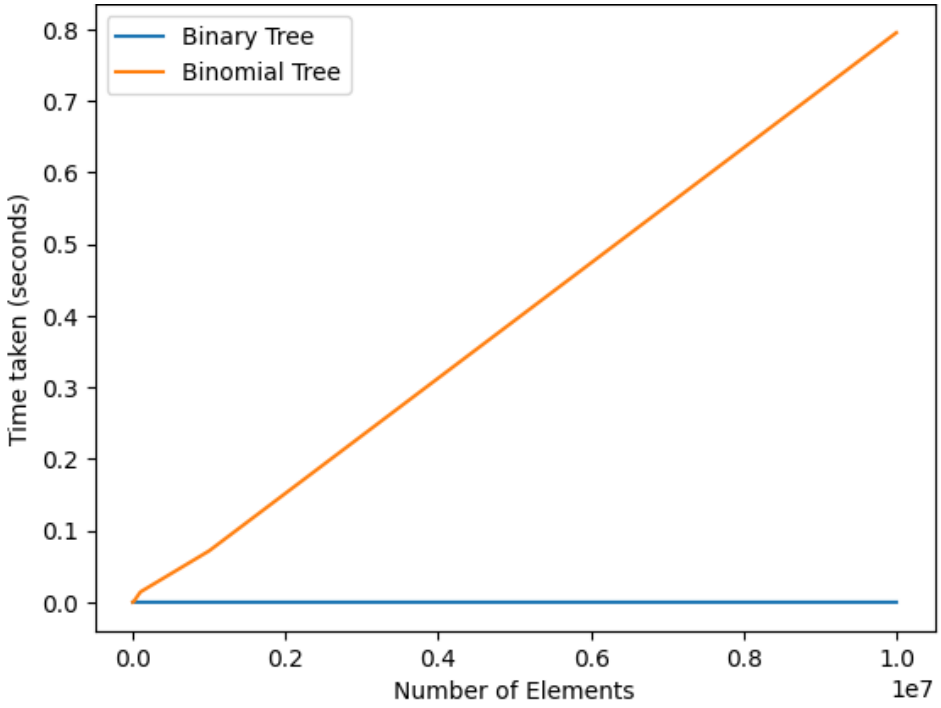

Рисунок 4. Сравнение худшего времени удаления одного элемента в бинарном и биномиальном деревьях

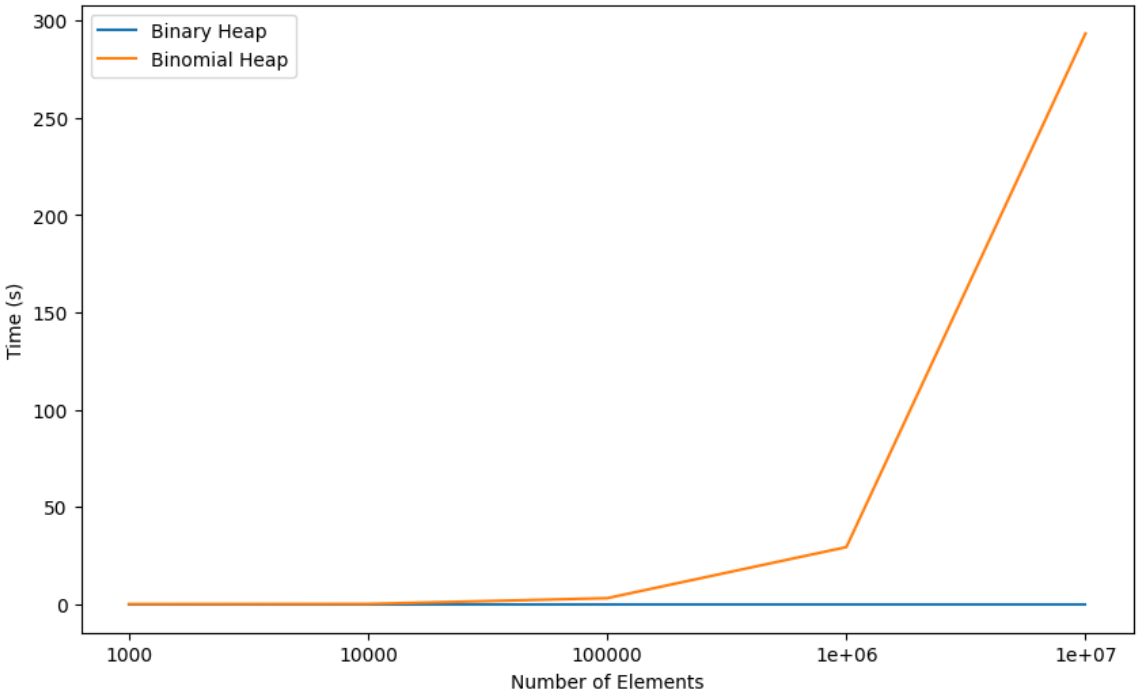

Рисунок 5. Сравнение времени поиска максимального элемента в бинарном и биномиальном деревьях

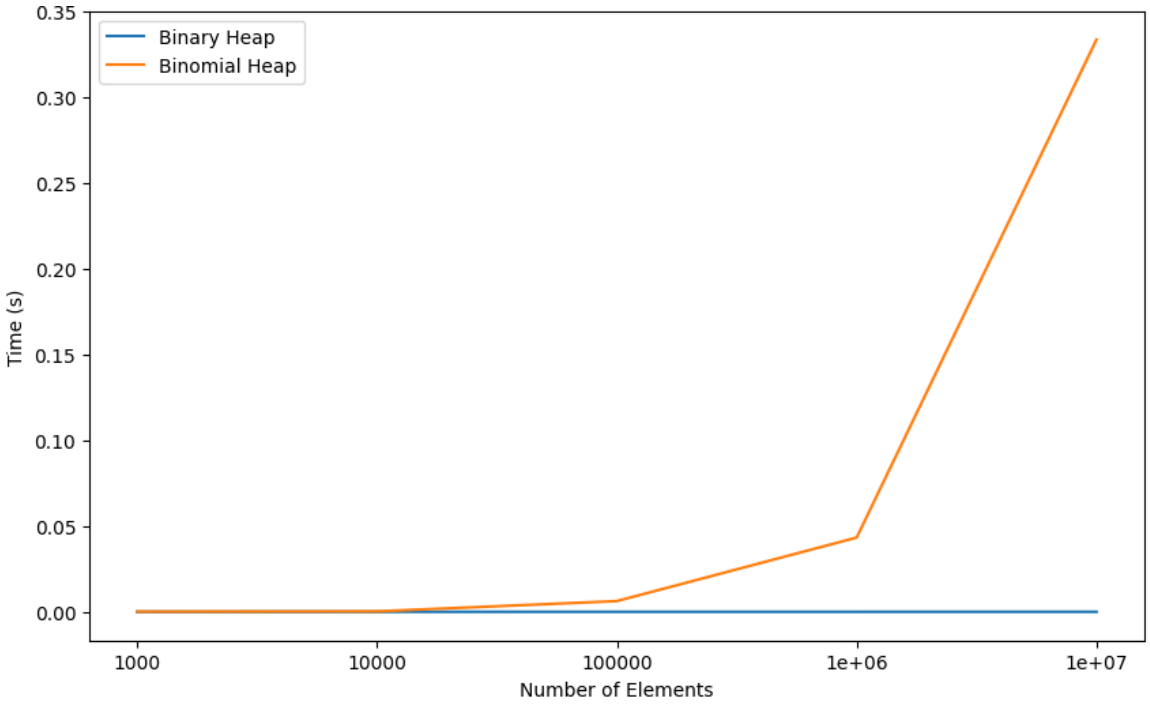

Рисунок 6. Сравнение максимального времени поиска максимального элемента в бинарном и биномиальном деревьях

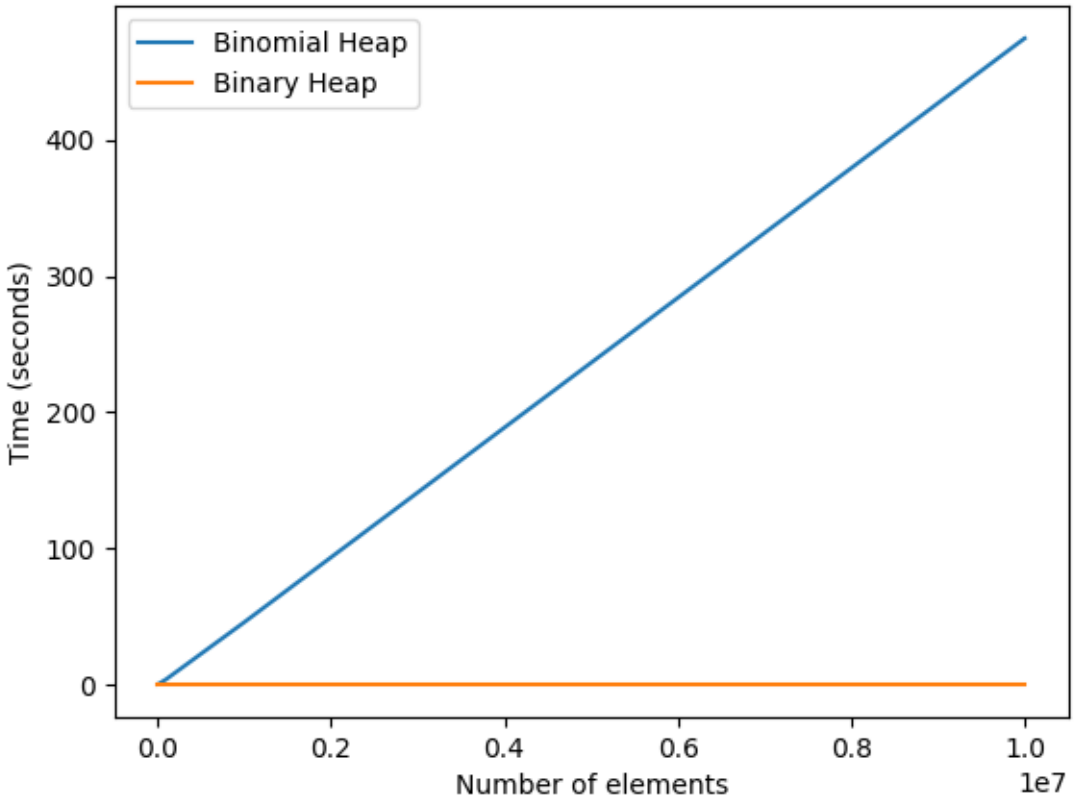

Рисунок 7. Сравнение времени вставки 1000 элементов в бинарное и биномиальное дерево

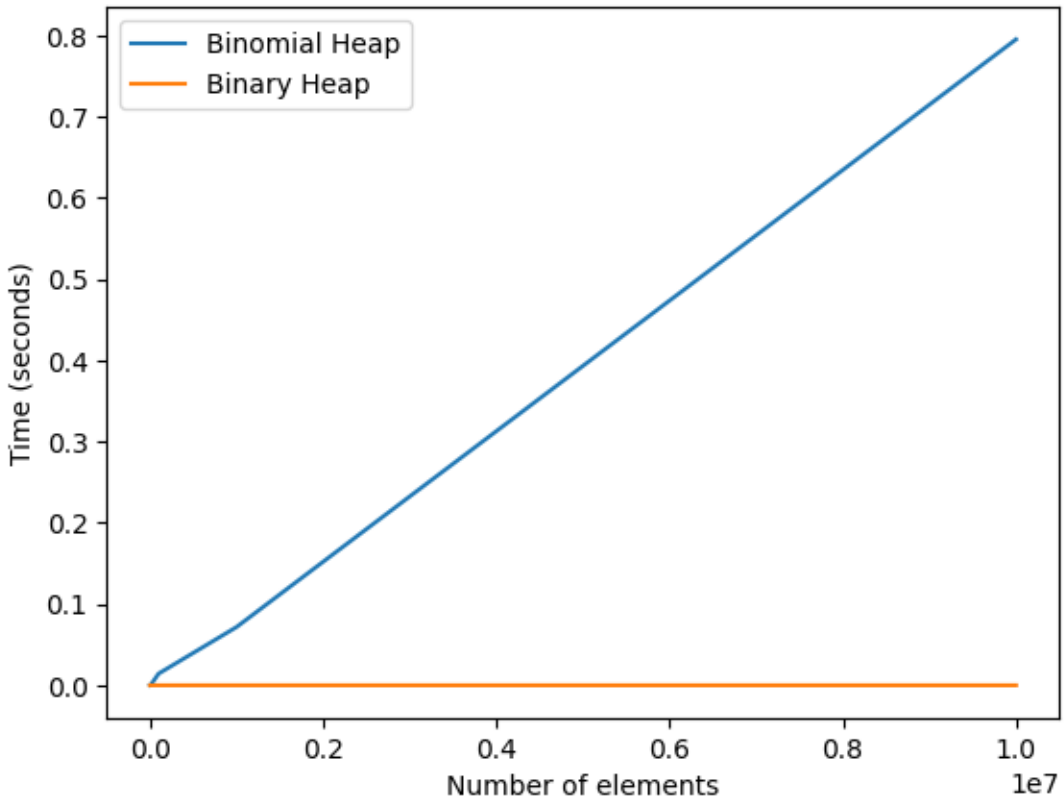

Рисунок 8. Сравнение худшего времени вставки одного элемента в бинарное и биномиальное дерево

Заключение.

Бинарная куча - самая простая реализация кучи, которая обеспечивает эффективное добавление новых элементов, извлечение максимального элемента и поиск максимального элемента, используя операции перехода между родительскими и дочерними узлами в бинарном дереве. Однако бинарная куча может иметь худшую производительность при объединении нескольких куч в одну.

Таким образом, бинарная куча проста и эффективна для операций добавления, извлечения и поиска максимального элемента, но не очень эффективна для объединения нескольких куч. Биномиальная куча, с другой стороны, может быть несколько более сложной для реализации и медленнее для операций добавления и поиска максимального элемента, но обеспечивает эффективное объединение нескольких куч в одну.