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..... Прилепский Артем Сергеевич
Ссылка на репозиторий: .....github.com/news1d/Algorithms_and_structures

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

Дата сдачи: 21.04.2023

Оглавление

Описание задачи.

В рамках лабораторной работы необходимо реализовать бинарную кучу(мин или макс), а так же 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:
    1. 1000 раз найти минимум/максимум
    2. 1000 раз удалить минимум/максимум
    3. 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, 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 метода/модели.

Двоичная куча – 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 дерево – это такая структура данных, которая задает саму себя рекурсивно через свой предыдущий шаг. 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, чтобы 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.

Для реализации программы был выбран язык программирования Python.

1) Класс двоичной кучи:

```
class BinaryHeapMax():
    # Инициализируем пустую кучу с единичным корнем
    def __init__(self):
        self.heap = [0]
        self.size = 0

    # Функция для вставки элемента в кучу
    def insert(self, val):
        self.heap.append(val)
        self.size += 1
        self._perc_up(self.size)

    # Вспомогательная функция, которая поднимает элемент, если он больше своего
родителя
    def _perc_up(self, i):
        # Пока родительский элемент существует, продолжаем перестраивать кучу
        while i // 2 > 0:
            # Если элемент больше своего родителя, меняем их местами
            if self.heap[i] > self.heap[i // 2]:
                tmp = self.heap[i // 2]
                self.heap[i // 2] = self.heap[i]
                self.heap[i] = tmp
            i = i // 2

    # Функция, которая удаляем максимальный элемент
    def delete_max(self):
        # Если куча существует
        if self.heap and self.size > 0:
            # Сохраняем значение максимального элемента
            retval = self.heap[1]
            # Помещаем последний элемент на вершину кучи
            self.heap[1] = self.heap[self.size]
            # Уменьшаем размер кучи
            self.size -= 1
            # Удаляем последний элемент
            self.heap.pop()
            # Опускаем новый корень кучи на нужное место
            self._perc_down(1)
            return retval
        else:
            return 0

    # Вспомогательная функция, которая опускает элемент вниз, если он меньше
своих потомков
    def _perc_down(self, i):
        # Пока у элемента i есть потомки, продолжаем перестраивать кучу
        while i * 2 <= self.size:
            # Находим наибольшего потомка элемента i
            mc = self._max_child(i * 2)
            # Если значение наибольшего потомка больше значения элемента i,
меняем их местами
            if self.heap[i] < self.heap[mc]:
                tmp = self.heap[i]
                self.heap[i] = self.heap[mc]
                self.heap[mc] = tmp
            i = mc

    # Вспомогательная функция, которая возвращает индекс наибольшего потомка
элемента с заданным индексом
```

```
    def _max_child(self, i):
        # Если у элемента i только один потомок, возвращаем его
        if i + 1 > self.size:
            return i
        else:
            # Иначе сравниваем значения двух потомков и возвращаем индекс
наибольшего
            if self.heap[i] > self.heap[i + 1]:
                return i
            else:
                return i + 1

    # Функция, которая возвращает максимальный элемент кучи
    def get_max(self):
        if self.heap and self.size > 0:
            return self.heap[1]
        return 0
```

2) Класс узла биномиальной кучи:

```
class Node:
    def __init__(self, key):
        self.key = key          # Ключ узла
        self.degree = 0         # Степень узла
        self.parent = None      # Родитель узла
        self.child = None       # Потомок узла
        self.sibling = None     # "Брат" узла
```

3) Класс биномиальной кучи:

```
class BinomialHeap:
    def __init__(self):
        self.head = None

    # Функция для поиска минимального ключа в куче
    def get_min(self):
        min_node = None
        current = self.head
        min_val = inf       # Инициализируем мин. значение как бесконечность
        # Проходим по всем узлам кучи
        while current:
            # Если значение текущего узла меньше минимального, обновляем
минимальное значение
            if current.key < min_val:
                min_val = current.key
                min_node = current
            current = current.sibling
        return min_node.key

    # Вспомогательная функция, которая добавляет новый узел в список дочерних
узлов корневого узла
    def _tree_link(self, new_child, root):
        # Если ключ корня больше ключа нового потомка
        if root.key > new_child.key:
            return 0
        # Если новый потомок или корень равны отсутствуют
        if new_child is None or root is None:
            return 0
        # Устанавливаем корень в качестве родителя для нового потомка
        new_child.parent = root
        # Устанавливаем нового потомка в качестве первого дочернего элемента
корня
        new_child.sibling = root.child
        # Обновляем ссылку на первый дочерний элемент корня на нового потомок
        root.child = new_child
        # Увеличиваем степень корня на 1
```

```
        root.degree += 1

    # Вспомогательная функция, которая добавляет биномиальные деревья в порядке степеней
    def _heap_merge(self, other_heap):
        # Устанавливаем ссылки на корни каждой кучи
        x_node = self.head
        y_node = other_heap.head
        # Переменная для хранения выбранного узла на каждой итерации
        selected = None
        # Переменная для хранения корневого узла объединенной кучи
        head_node = None
        # Переменная для хранения предыдущего узла на каждой итерации
        prev_node = None

        # Пока есть хотя бы один узел в одной из куч
        while x_node or y_node:
            # Если узел x_node отсутствует, то выбираем y_node
            if x_node is None:
                selected = y_node
                y_node = y_node.sibling
            # Если узел y_node отсутствует, то выбираем x_node
            elif y_node is None:
                selected = x_node
                x_node = x_node.sibling
            # Если степень узла x node меньше или равна степени узла y node, то выбираем x_node
            elif x_node.degree <= y_node.degree:
                selected = x_node
                x_node = x_node.sibling
            # Если степень узла y_node меньше степени узла x_node, то выбираем y_node
            else:
                selected = y_node
                y node = y node.sibling

            # Если еще не был выбран корневой узел, то выбираем его (только для первой итерации)
            if head_node is None:
                head_node = selected
            # Иначе, устанавливаем выбранный узел как следующий узел для предыдущего узла
            else:
                prev_node.sibling = selected

            # Обновляем предыдущий узел
            prev_node = selected

        # Возвращаем корневой узел объединенной кучи
        return head_node

    # Вспомогательная функция, которая объединяет две кучи
    def _union(self, other_heap):
        # Создаем новую кучу
        unite = BinomialHeap()

        # Объединяем две кучи и устанавливаем корневой узел объединенной кучи
        unite.head = self._heap_merge(other_heap)
        # Если объединенная куча пуста, то возвращаем ее
        if unite.head is None:
            return unite

        # Инициализируем переменные для хранения текущего, предыдущего и следующего узла
        prev = None
        current = unite.head
        next = current.sibling
```

```
        # Проходим по всем узлам новой кучи
        while next:
            # Если текущий узел и следующий узел имеют разные степени или
            # степень текущего узла равна степени узла после следующего,
            # перемещаемся к следующему узлу
            if (current.degree != next.degree or
                    next.sibling is not None and next.sibling.degree ==
current.degree):
                prev = current
                current = next
                # Если степень текущего узла меньше степени следующего узла,
выбираем текущий узел
            else:
                # Если значение текущего узла меньше или равно значению
следующего, то связываем их между собой
                if current.key <= next.key:
                    current.sibling = next.sibling
                    self._tree_link(next, current)
                else:
                    # Если предыдущий узел отсутсвует, обновляем корень новой
кучи на следующий узел
                    if prev is None:
                        unite.head = next
                    else:
                        prev.sibling = next

                    # Связываем текущий узел и следующий между собой
                    self._tree_link(current, next)
                    current = next

            # Обновляем следующий узел
            next = current.sibling

        # Возвращаем объединенную кучу
        return unite

    # Функция для вставки узла в биномиальную кучу
    def insert(self, key):
        # Создаем новый узел и сохраняем его в переменной node
        node = Node(key)
        # Создаем новую биномиальную кучу с одним элементом
        single = BinomialHeap()
        # Кореню новой кучи присваиваем узел node
        single.head = node
        # Объединяем текущую кучу с single кучей и присваиваем ее корень текущей
куче
        self.head = self._union(single).head

    # Функция для удаления  узла с минимальным значением
    def delete_min(self):
        # Если биномиальная куча отсутствует
        if self.head is None:
            return 0

        # Инициализируем предыдущий узел, узел с минимальным значением и левого
потомка
        prev = None
        min_node = None
        left_sibling = None
        # Устанавливаем корневой узел в качестве текущего
        current = self.head
        # Инициализируем мин. значение как бесконечность
        min_val = inf

        # Проходимся по всем узлам
        while current:
```

```
            # Проверяем, является ли текущий узел минимальным
            if current.key < min_val:
                # Обновляем переменные
                min_val = current.key
                min_node = current
                prev = left_sibling
            left_sibling = current
            current = current.sibling

        # Удаляем минимальный узел из кучи
        if prev is None:
            self.head = min_node.sibling
        else:
            prev.sibling = min_node.sibling

        # Создаем новую кучу
        heap = BinomialHeap()
        # Присваем переменной node значения потомка удаленного узла
        node = min_node.child

        # Проходим по всем потомкам удаленного узла и добавляем их в новую кучу
        while node:
            # Сохраняем ссылку на следующий узел перед изменением ссылок текущего
узла
            next = node.sibling
            # Отсоединяем узел от его родителя
            node.parent = None
            # Добавляем узел в кучу, сделав его корневым элементом
            node.sibling = heap.head
            heap.head = node
            # Переходим к следующему дочернему узлу
            node = next

        # Объединяем новую кучу с оставшимися кучами.
        self.head = self. union(heap).head
        return min_node
```

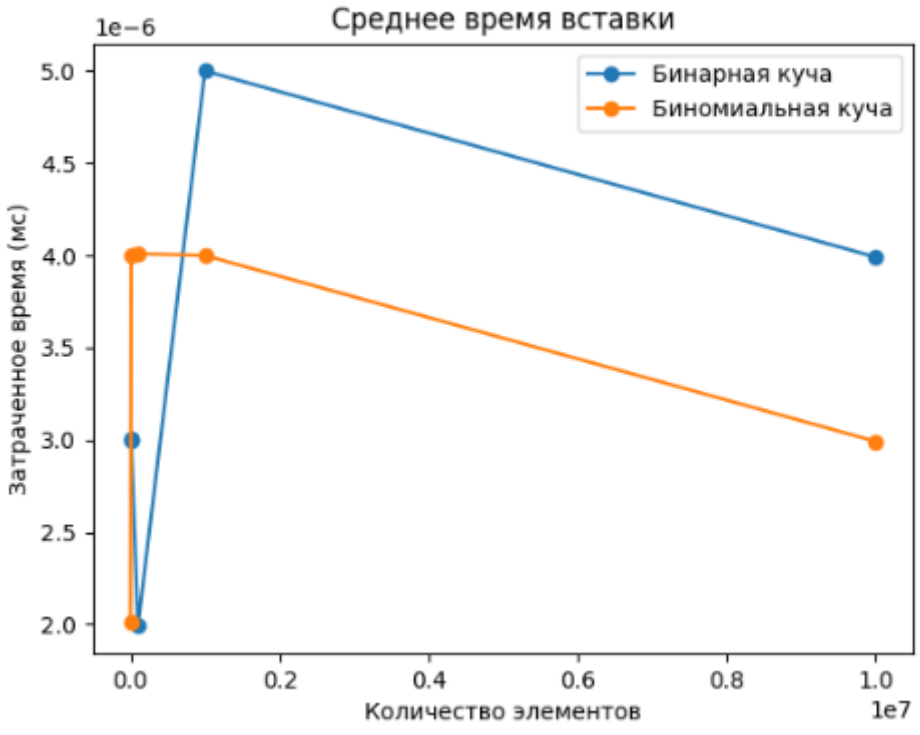

График зависимости среднего времени вставки от количества элементов

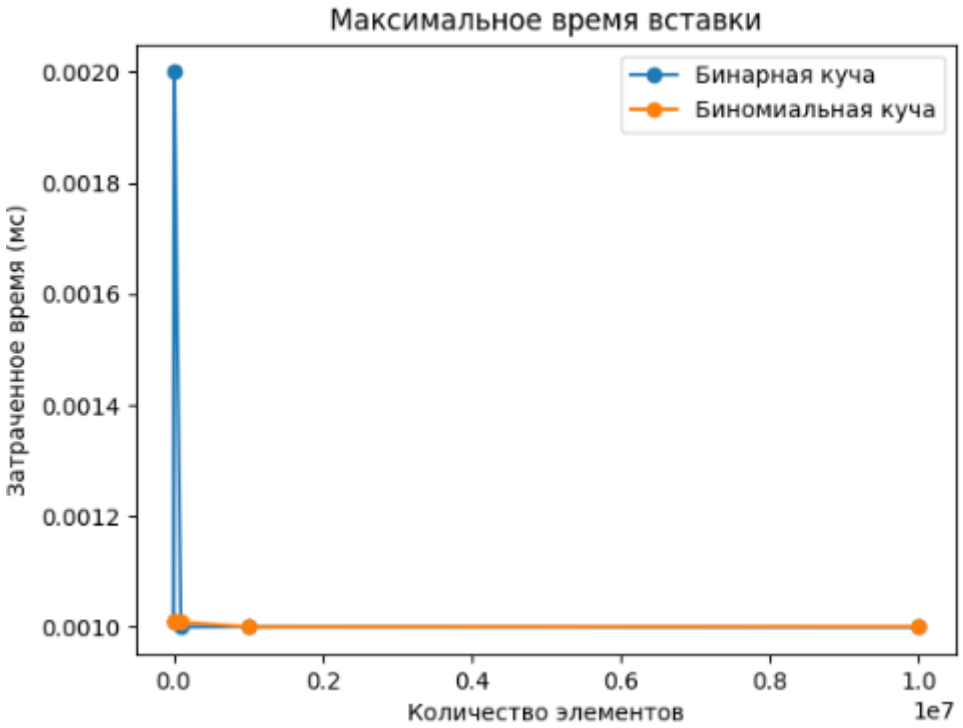

График зависимости максимального времени вставки от количества элементов

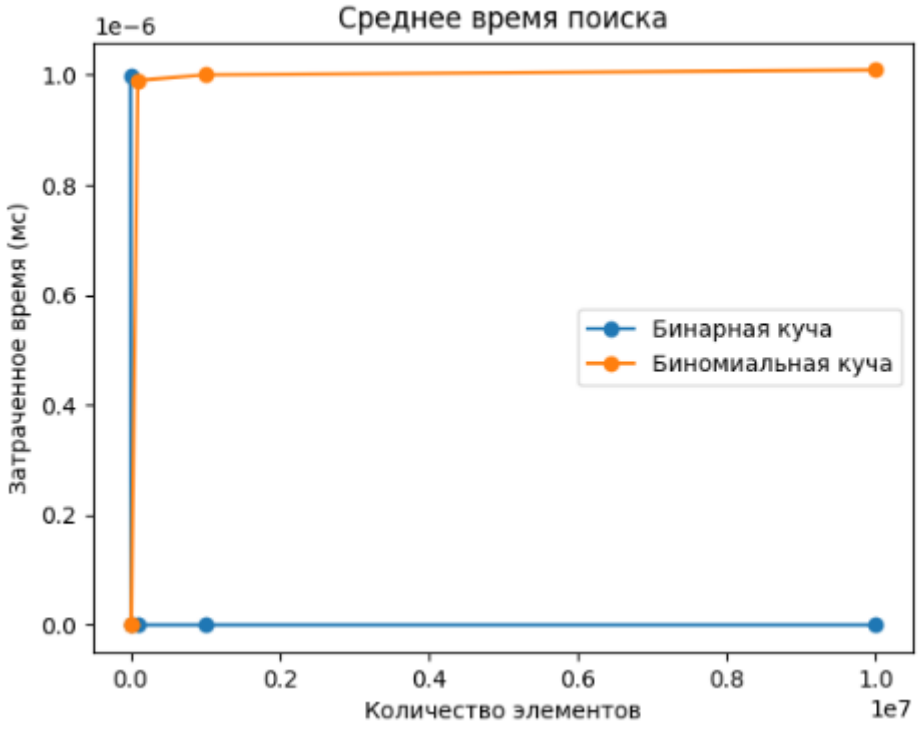

График зависимости среднего времени поиска от количества элементов

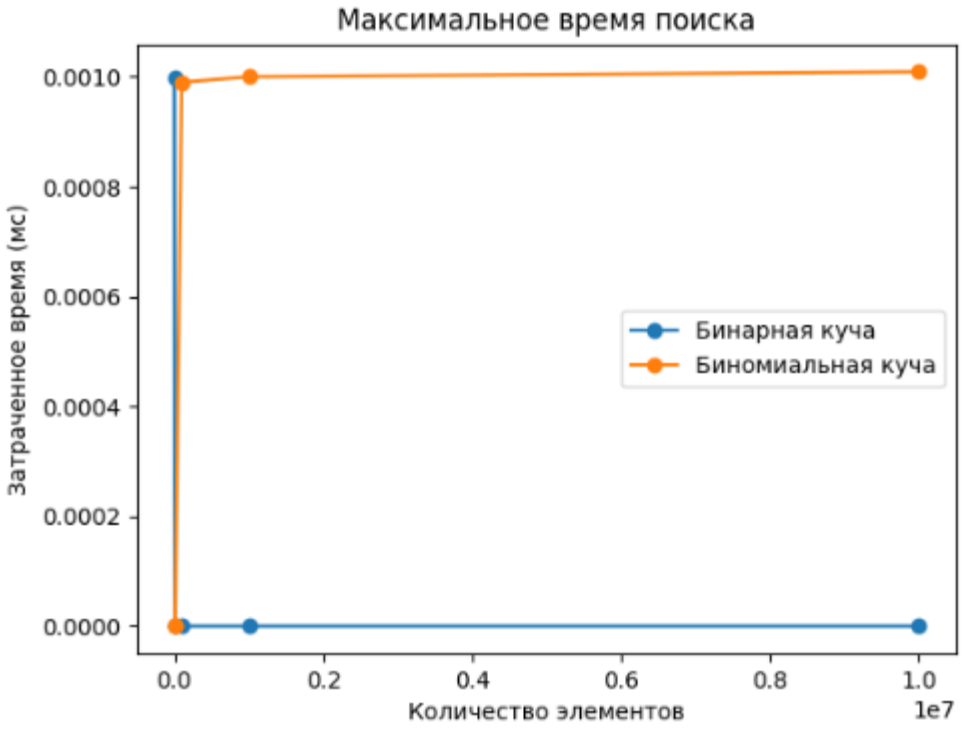

График зависимости максимального времени поиска от количества элементов

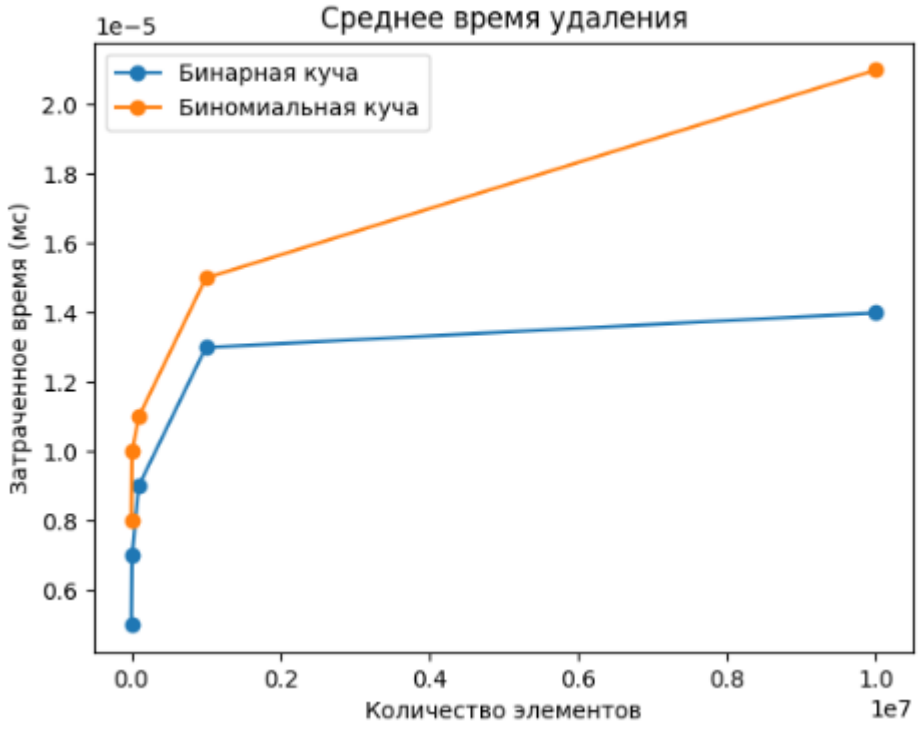

График зависимости среднего времени удаления от количества элементов

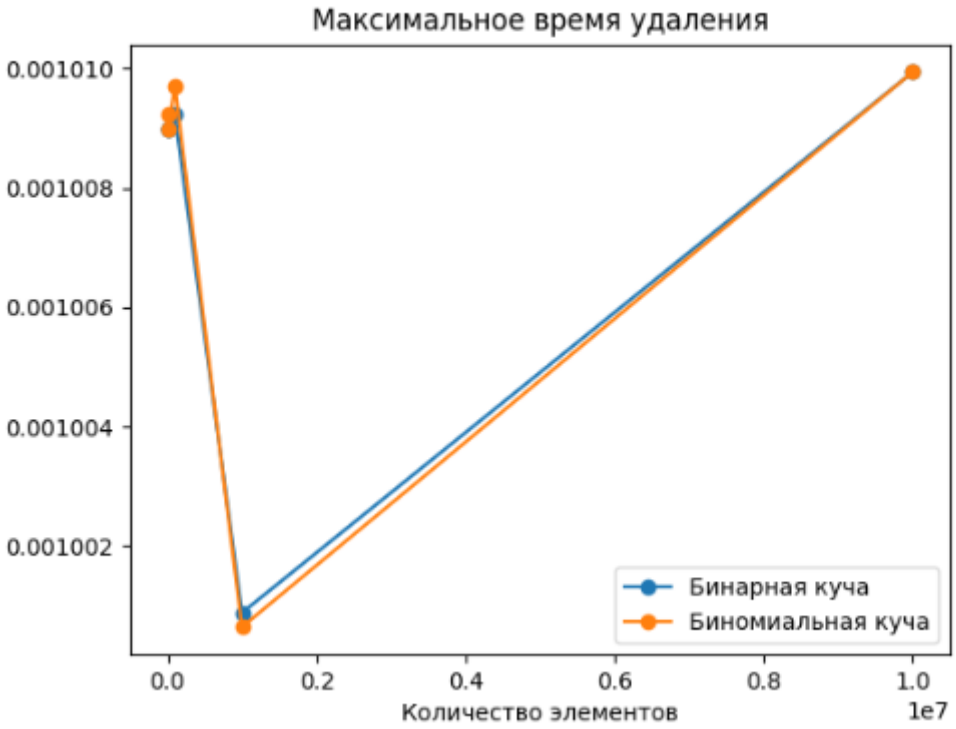

График зависимости максимального времени удаления от количества элементов

Заключение.

Подведем итоги, обе кучи имеют одинаковую временную сложность для вставки и удаления минимального элемента. Однако, биномиальная куча имеет лучшую асимптотическую временную сложность для поиска минимального элемента. В бинарной куче поиск минимального элемента выполняется за O(1), что делает ее более подходящей для приложений, где часто требуется получать минимальный элемент.

В заключении можно сказать, что обе кучи являются эффективными структурами данных для хранения и обработки элементов в порядке возрастания или убывания, но выбор между ними зависит от конкретных требований и характеристик приложения. Если приложение требует быстрого поиска минимального элемента, то бинарная куча может быть лучшим выбором, а если требуется быстрое добавление и удаление элементов, то биномиальная куча может быть предпочтительнее.