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-36 ............................................. Тернолуцкий Виктор Александрович

Ссылка на репозиторий: .... ... <https://github.com/MUCTR-IKT-CPP/VATernolutski_36>

Приняли: Крашенинников Роман Сергеевич

Дата сдачи: 19.04.2025

Оглавление

[Описание задачи. 2](#_Toc63548272)

[Описание метода/модели. 2](#_Toc63548273)

[Выполнение задачи. 4](#_Toc63548274)

[Заключение. 17](#_Toc63548275)

# Описание задачи.

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

Фибоначчиеву 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, 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.

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).

Двоичная куча представляет собой полное бинарное дерево, для которого выполняется *основное свойство кучи*: приоритет каждой вершины больше приоритетов её потомков. В простейшем случае приоритет каждой вершины можно считать равным её значению. В таком случае структура называется *max-heap*, поскольку корень поддерева является максимумом из значений элементов поддерева. В этой статье для простоты используется именно такое представление. Называется *полным бинарным*, если у каждой вершины есть не более двух потомков, а заполнение уровней вершин идет сверху вниз (в пределах одного уровня – слева направо).

Двоичную кучу удобно хранить в виде одномерного массива, причем левый потомок вершины с индексом **i** имеет индекс **2\*i+1**, а правый **2\*i+2**. Корень дерева – элемент с индексом 0. Высота двоичной кучи равна высоте дерева, то есть log2 N, где **N** – количество элементов массива.

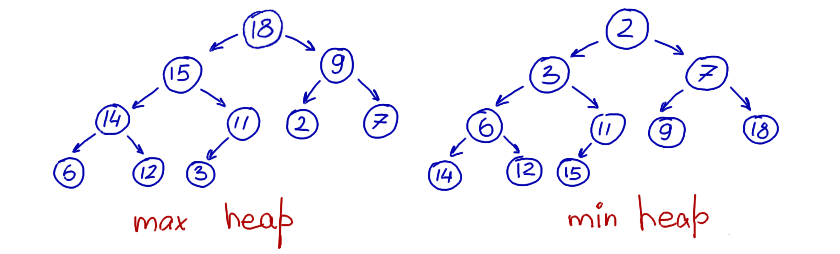


Рисунок 1 – бинарная куча.

Фибоначчиева куча (англ. Fibonacci heap) — структура данных, представляющая собой набор деревьев, упорядоченных в соответствии со свойством неубывающей пирамиды. Фибоначчиевы кучи были введены Майклом Фредманом и Робертом Тарьяном в 1984 году.

Структура является реализацией абстрактного типа данных «Очередь с приоритетом», и замечательна тем, что операции, в которых не требуется удаление, имеют амортизированное время работы, равное (для двоичной кучи и биномиальной кучи амортизационное время работы равно). Кроме стандартных операций INSERT, MIN, DECREASE-KEY, фибоначчиева куча позволяет за время выполнять операцию UNION слияния двух куч.

Фибоначчиева куча — набор из подвешенных деревьев удовлетворяющих свойству: каждый предок не больше своих детей(если дерево на минимум). Это означает, что минимум всей кучи это один из корней этих деревьев. Одно из главных преимуществ Фибоначчиевой кучи — гибкость её структуры из-за того, что на деревья не наложены никакие ограничения по форме. Например, Фибоначчиева куча может состоять хоть из деревьев в каждом из которых по одному элементу. Такая гибкость позволяет выполнять некоторые операции лениво, оставляя работу более поздним операциям. Далее будут даны некоторые определения, которые понадобятся в дальнейшем.

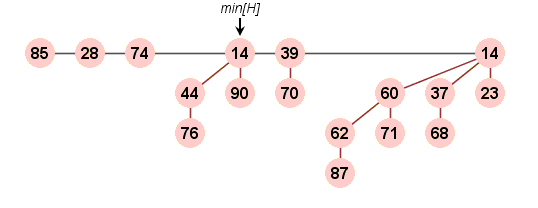


Рисунок 2 – Фибоначчиева куча

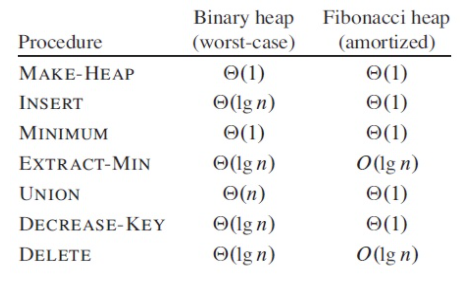


Рисунок 3 – сравнение бинарной кучи и фибоначчиевой.

# Выполнение задачи.

import random

import time

import csv

import os

import matplotlib.pyplot as plt

*# Binary Heap Implementation*

**class** BinaryHeap:

**def** \_\_init\_\_(self):

        self.data = []

**def** insert(self, val):

        self.data.append(val)

        self.\_up\_heap(len(self.data) - 1)

**def** find\_min(self):

        if not self.data:

            raise ValueError("Heap is empty")

        return self.data[0]

**def** delete\_min(self):

        if not self.data:

            raise ValueError("Heap is empty")

        min\_val = self.data[0]

        last = self.data.pop()

        if self.data:

            self.data[0] = last

            self.\_down\_heap(0)

        return min\_val

**def** \_up\_heap(self, idx):

        while idx > 0:

            parent = (idx - 1) // 2

            if self.data[idx] >= self.data[parent]:

                break

            self.data[idx], self.data[parent] = self.data[parent], self.data[idx]

            idx = parent

**def** \_down\_heap(self, idx):

        size = len(self.data)

        while True:

            left = 2 \* idx + 1

            right = 2 \* idx + 2

            smallest = idx

            if left < size and self.data[left] < self.data[smallest]:

                smallest = left

            if right < size and self.data[right] < self.data[smallest]:

                smallest = right

            if smallest == idx:

                break

            self.data[idx], self.data[smallest] = self.data[smallest], self.data[idx]

            idx = smallest

**def** size(self):

        return len(self.data)

*# Fibonacci Heap Implementation*

**class** FibonacciNode:

**def** \_\_init\_\_(self, val):

        self.val = val

        self.parent = None

        self.child = None

        self.left = self

        self.right = self

        self.degree = 0

        self.mark = False

**class** FibonacciHeap:

**def** \_\_init\_\_(self):

        self.min\_node = None

        self.total\_nodes = 0

**def** insert(self, val):

        new\_node = FibonacciNode(val)

        if self.min\_node is None:

            self.min\_node = new\_node

        else:

            self.\_merge\_with\_root\_list(new\_node)

            if new\_node.val < self.min\_node.val:

                self.min\_node = new\_node

        self.total\_nodes += 1

**def** find\_min(self):

        if self.min\_node is None:

            raise ValueError("Heap is empty")

        return self.min\_node.val

**def** delete\_min(self):

        if self.min\_node is None:

            raise ValueError("Heap is empty")

        min\_val = self.min\_node.val

        if self.min\_node.child is not None:

            children = self.\_remove\_from\_list(self.min\_node.child)

            for child in children:

                self.\_merge\_with\_root\_list(child)

                child.parent = None

        self.\_remove\_from\_list(self.min\_node)

        if self.min\_node == self.min\_node.right:

            self.min\_node = None

        else:

            self.min\_node = self.min\_node.right

            self.\_consolidate()

        self.total\_nodes -= 1

        return min\_val

**def** \_merge\_with\_root\_list(self, node):

        if self.min\_node is None:

            self.min\_node = node

        else:

            node.right = self.min\_node.right

            node.left = self.min\_node

            self.min\_node.right.left = node

            self.min\_node.right = node

**def** \_remove\_from\_list(self, node):

        if node.right == node:

            return [node]

        left = node.left

        right = node.right

        left.right = right

        right.left = left

        node.left = node

        node.right = node

        return [node]

**def** \_consolidate(self):

        aux = [None] \* (self.total\_nodes + 1)

        nodes = self.\_get\_root\_list()

        for w in nodes:

            x = w

            d = x.degree

            while aux[d] is not None:

                y = aux[d]

                if x.val > y.val:

                    x, y = y, x

                self.\_link(y, x)

                aux[d] = None

                d += 1

            aux[d] = x

        self.min\_node = None

        for i in range(len(aux)):

            if aux[i] is not None:

                if self.min\_node is None:

                    self.min\_node = aux[i]

                else:

                    self.\_merge\_with\_root\_list(aux[i])

                    if aux[i].val < self.min\_node.val:

                        self.min\_node = aux[i]

**def** \_link(self, y, x):

        self.\_remove\_from\_list(y)

        y.parent = x

        if x.child is None:

            x.child = y

            y.left = y

            y.right = y

        else:

            self.\_merge\_with\_root\_list(x.child, y)

        x.degree += 1

        y.mark = False

**def** \_get\_root\_list(self):

        if self.min\_node is None:

            return []

        nodes = []

        start = self.min\_node

        current = start

        while True:

            nodes.append(current)

            current = current.right

            if current == start:

                break

        return nodes

**def** size(self):

        return self.total\_nodes

*# Benchmarking Function*

**def** benchmark\_heap(name, heap, n):

    results = [name, str(n)]

    random.seed(time.time())

    for \_ in range(n):

        heap.insert(random.randint(0, n \* 10))

**def** measure(op, rounds):

        max\_dur = 0

        total = 0

        batch\_sizes = [10, 25, 50, 100]

        for \_ in range(rounds):

            start = time.time()

            op()

            duration = time.time() - start

            total += duration

            if duration > max\_dur:

                max\_dur = duration

        for batch in batch\_sizes:

            for i in range(0, rounds, batch):

                batch\_start = time.time()

                for j in range(min(batch, rounds - i)):

                    op()

                batch\_dur = (time.time() - batch\_start) / batch if batch > 0 else 0

                if batch\_dur > max\_dur:

                    max\_dur = batch\_dur

        return total / rounds, max\_dur

**def** find\_min\_op():

        heap.find\_min()

    avg, max\_dur = measure(find\_min\_op, 1000)

    results.extend([str(avg), str(max\_dur)])

**def** delete\_min\_op():

        try:

            heap.delete\_min()

        except ValueError:

            heap.insert(random.randint(0, n \* 10))

    avg, max\_dur = measure(delete\_min\_op, 1000)

    results.extend([str(avg), str(max\_dur)])

**def** insert\_op():

        heap.insert(random.randint(0, n \* 10))

    avg, max\_dur = measure(insert\_op, 1000)

    results.extend([str(avg), str(max\_dur)])

    return results

**def** benchmark():

    sizes = [10\*\*i for i in range(3, 8)]

    headers = ["Heap", "N", "FindMin\_Avg", "FindMin\_Max", "DeleteMin\_Avg", "DeleteMin\_Max", "Insert\_Avg", "Insert\_Max"]

    binary\_results = []

    fib\_results = []

    for n in sizes:

        print(**f**"Testing BinaryHeap with N = {n}")

        binary\_results.append(benchmark\_heap("BinaryHeap", BinaryHeap(), n))

        print(**f**"Testing FibonacciHeap with N = {n}")

        fib\_results.append(benchmark\_heap("FibonacciHeap", FibonacciHeap(), n))

    save\_csv("binary\_heap\_results.csv", headers, binary\_results)

    save\_csv("fibonacci\_heap\_results.csv", headers, fib\_results)

**def** save\_csv(filename, headers, data):

    with open(filename, 'w', newline='') as file:

        writer = csv.writer(file)

        writer.writerow(headers)

        writer.writerows(data)

*# Graph Generation*

**def** read\_csv\_results(path):

    with open(path, 'r') as file:

        reader = csv.reader(file)

        next(reader)

        return [{

            "N": int(row[1]),

            "FindMin\_Avg": float(row[2]),

            "FindMin\_Max": float(row[3]),

            "DeleteMin\_Avg": float(row[4]),

            "DeleteMin\_Max": float(row[5]),

            "Insert\_Avg": float(row[6]),

            "Insert\_Max": float(row[7])

        } for row in reader]

**def** plot\_graph(filename, title, ylabel, data, value\_key):

    plt.figure()

    plt.title(title)

    plt.xlabel("N")

    plt.ylabel(ylabel)

    x = [d["N"] for d in data]

    y = [d[value\_key] for d in data]

    plt.plot(x, y, marker='o')

    plt.savefig(filename)

    plt.close()

**def** generate\_plots():

    for csv\_path, prefix in [("binary\_heap\_results.csv", "binary"), ("fibonacci\_heap\_results.csv", "fibonacci")]:

        data = read\_csv\_results(csv\_path)

        os.makedirs("charts", exist\_ok=True)

        plot\_graph(**f**"charts/{prefix}\_findmin\_avg.png", **f**"{prefix} FindMin Avg", "Time (s)", data, "FindMin\_Avg")

        plot\_graph(**f**"charts/{prefix}\_findmin\_max.png", **f**"{prefix} FindMin Max", "Time (s)", data, "FindMin\_Max")

        plot\_graph(**f**"charts/{prefix}\_deletemin\_avg.png", **f**"{prefix} DeleteMin Avg", "Time (s)", data, "DeleteMin\_Avg")

        plot\_graph(**f**"charts/{prefix}\_deletemin\_max.png", **f**"{prefix} DeleteMin Max", "Time (s)", data, "DeleteMin\_Max")

        plot\_graph(**f**"charts/{prefix}\_insert\_avg.png", **f**"{prefix} Insert Avg", "Time (s)", data, "Insert\_Avg")

        plot\_graph(**f**"charts/{prefix}\_insert\_max.png", **f**"{prefix} Insert Max", "Time (s)", data, "Insert\_Max")

        print(**f**"Graphs for {prefix} saved in 'charts' folder")

if \_\_name\_\_ == "\_\_main\_\_":

    benchmark()

    generate\_plots()

Результаты:  
Heap,N,FindMin\_Avg,FindMin\_Max,DeleteMin\_Avg,DeleteMin\_Max,Insert\_Avg,Insert\_Max

BinaryHeap,1000,0.0,4.999637603759765e-06,1.999616622924805e-06,0.0005018711090087891,1.0001659393310548e-06,0.0005044937133789062

BinaryHeap,10000,0.0,4.999637603759765e-06,3.998756408691406e-06,0.0005004405975341797,1.4998912811279298e-06,0.0005013942718505859

BinaryHeap,100000,5.004405975341797e-07,0.0005004405975341797,5.971193313598633e-06,0.0005304813385009766,1.0290145874023437e-06,0.0005288124084472656

BinaryHeap,1000000,0.0,2.00653076171875e-05,9.496212005615234e-06,0.0005285739898681641,1.4989376068115234e-06,0.0005002021789550781

BinaryHeap,10000000,5.01394271850586e-07,0.0005013942718505859,1.0499477386474609e-05,0.0005309581756591797,9.696483612060547e-07,0.0005042552947998047

Heap,N,FindMin\_Avg,FindMin\_Max,DeleteMin\_Avg,DeleteMin\_Max,Insert\_Avg,Insert\_Max

FibonacciHeap,1000,0.0,0,9.6893310546875e-07,0.0004987716674804688,2.0160675048828124e-06,0.0005183219909667969

FibonacciHeap,10000,0.0,5.035400390625e-06,9.95159149169922e-07,0.0005121231079101562,9.658336639404297e-07,0.0004982948303222656

FibonacciHeap,100000,4.980564117431641e-07,0.0004980564117431641,9.698867797851562e-07,0.0005004405975341797,9.992122650146484e-07,0.0004999637603759766

FibonacciHeap,1000000,0.0,4.718303680419922e-05,1.4998912811279298e-06,0.0005283355712890625,1.4996528625488281e-06,0.0005004405975341797

FibonacciHeap,10000000,0.0,2.0885467529296874e-05,4.99725341796875e-07,0.000499725341796875,9.97304916381836e-07,0.0004990100860595703

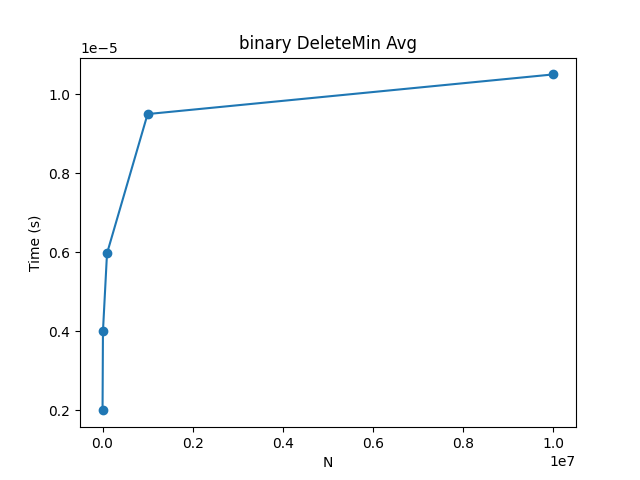
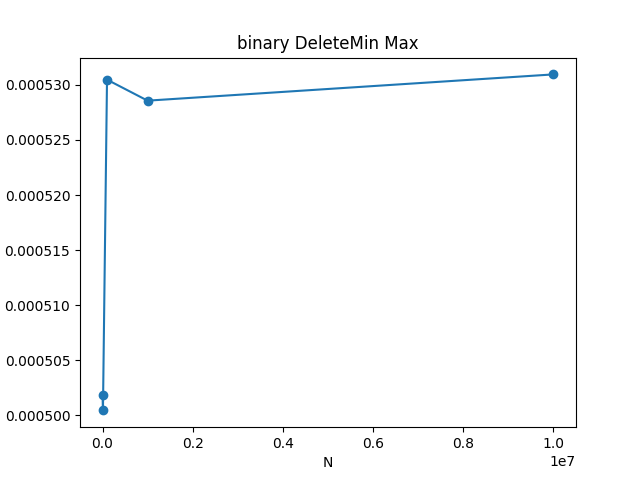


Рисунок 4 – среднее время удаления в бинарной куче

  
Рисунок 5– Максимальное время удаления в бинарной куче

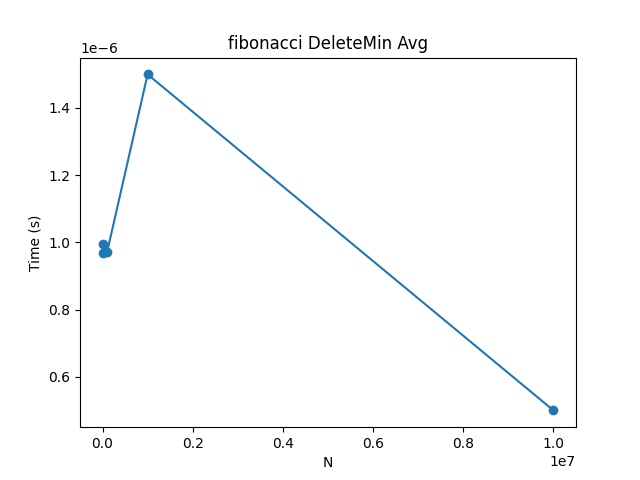
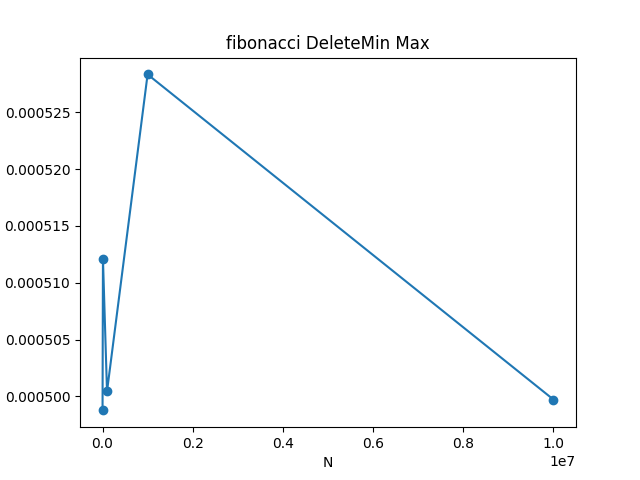


Рисунок 6 – среднее время удаление в Фибоначчиевой куче.

  
Рисунок 7 – Максимальное время удаление в Фибоначчиевой куче.

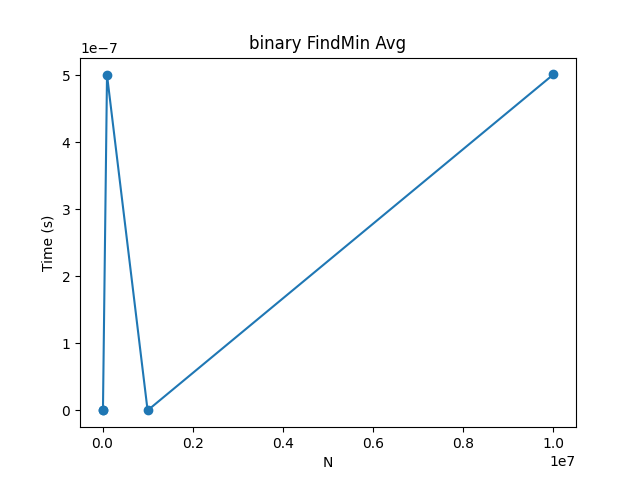
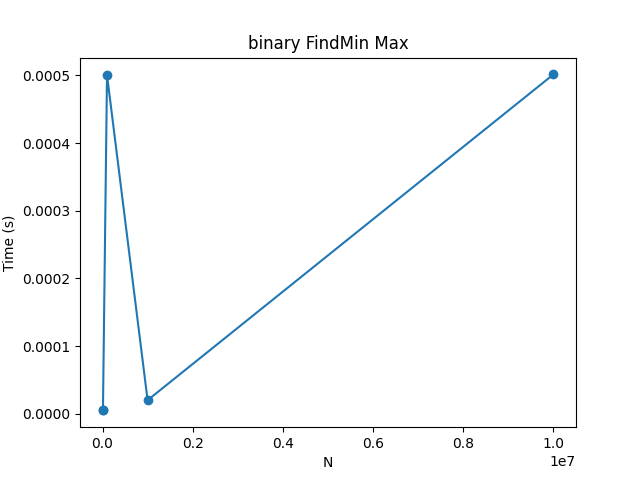


Рисунок 8 – среднее время поиска минимального элемента в бинарной куче  
  
  
Рисунок 8 – Максимальное время поиска минимального элемента в бинарной куче

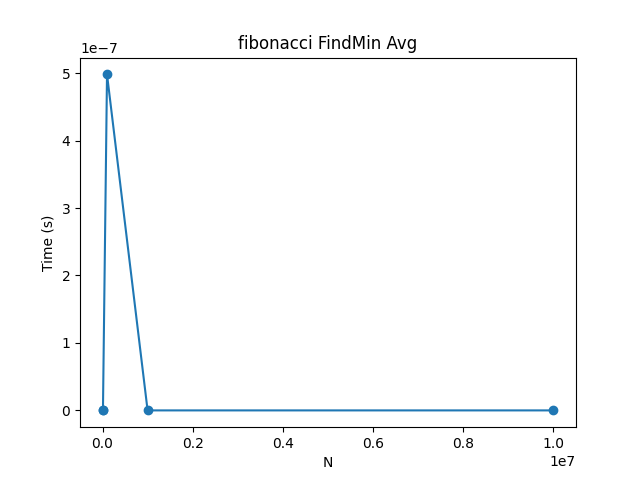


Рисунок 9 – среднее время поиска минимального элемента в Фибоначчиевой куче.

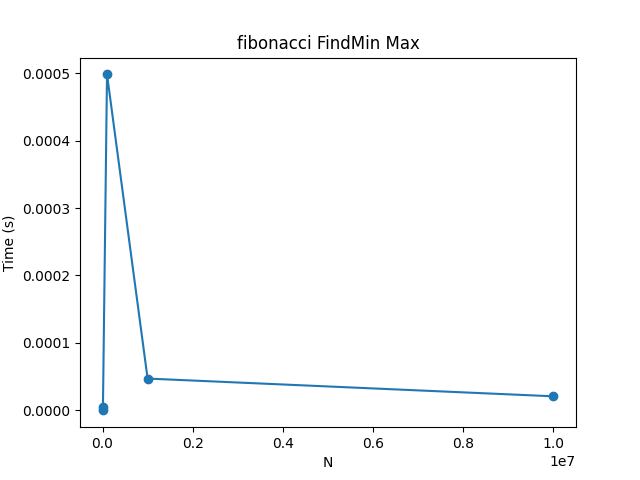


Рисунок 10 – Максимальное время поиска минимального элемента в Фибоначчиевой куче.

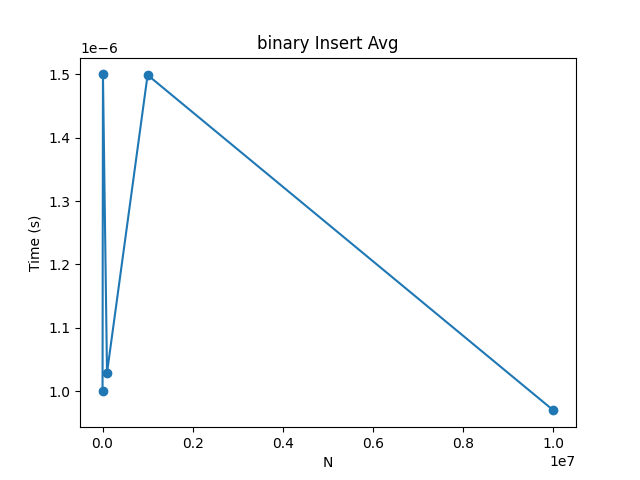


Рисунок 10 – среднее время вставки в бинарной куче.

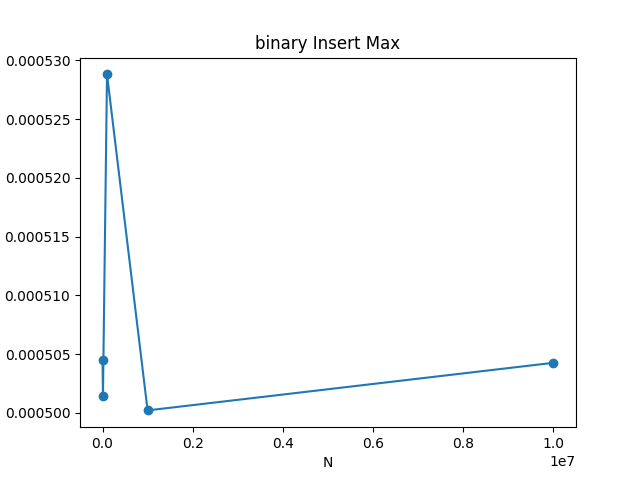


Рисунок 10 – Максимальное время вставки в бинарной куче.

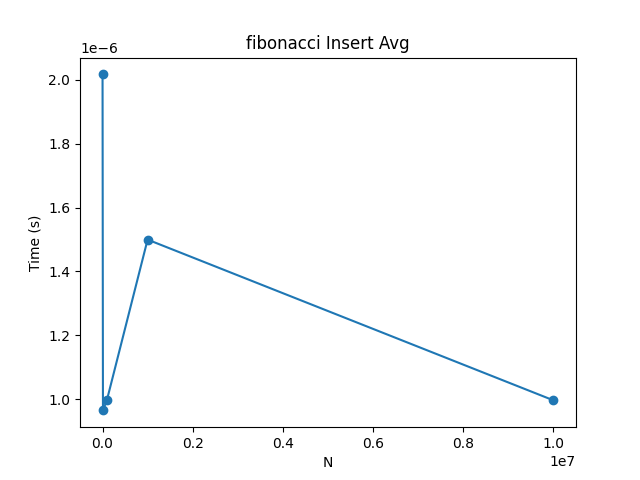


Рисунок 11 – среднее время вставки в Фибоначчиевой куче.

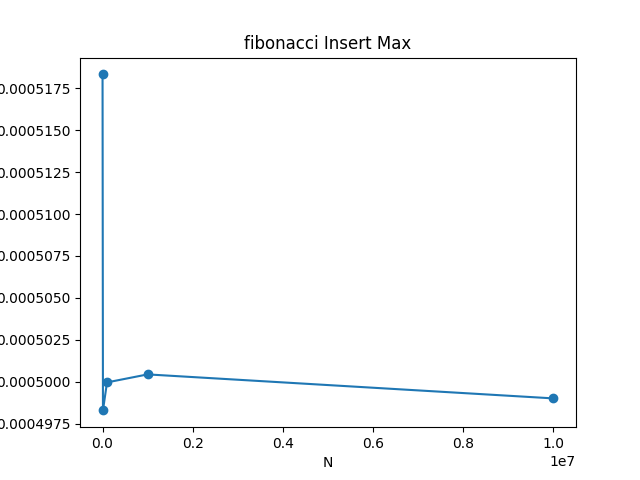


Рисунок 11 – максимальное время вставки в Фибоначчиевой куче.

# Заключение.

(BinaryHeap) подойдет для задач, где критичны стабильность и предсказуемость времени выполнения операций, особенно если операции удаления минимального элемента происходят часто.

(FibonacciHeap) может быть более выгодным выбором для сценариев с множеством операций вставки, где нужно минимизировать затраты на эту операцию, но при этом следует учитывать возможное значительное ухудшение производительности при удалении минимальных элементов для больших N.