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 Кошкарев Иван Михайлович

Ссылка на репозиторий: https://github.com/MUCTR-IKT-CPP/KoshkarevIM\_36\_ALG/tree/main/Lab8

Приняли: Крашенинников Роман Сергеевич

Дата сдачи: 17.03

Оглавление

[Описание задачи. 2](#_Toc63548272)

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

[Выполнение задачи. 2](#_Toc63548274)

[Заключение. 2](#_Toc63548275)

# Описание задачи.

Реализовать модель бинарной и фибоначчиевой кучи. Провести ряд тестов, чтобы сравнить их эффективность

# Описание метода/модели.

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.

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.

Фибоначчиева куча, это так же биноминальная куча, у которой прощаются все операции кроме операции удаления. За счет сознательного ухудшения состояния кучи внутри нее. Мы предполагаем, что наша куча может иметь в себе множество деревьев одинакового ранга. Т.е. у нас может быть хоть 20 деревьев первого нулевого ранга.

В такой ситуации мы можем позволить себе при добавлении в дерево нового элемента просто добавлять в кучу дерево ранга 0 содержащего наше новое значение, перепроверяя только минимум/максимум в куче(в зависимости от того в какую сторону мы взращиваем кучу).

Объединение двух куч так же позволяет нам совершать эту операцию просто выполняя объединение двух списков деревьев из которых состоят кучи, в конце лишь выбирая один из двух имеющихся минимумов.

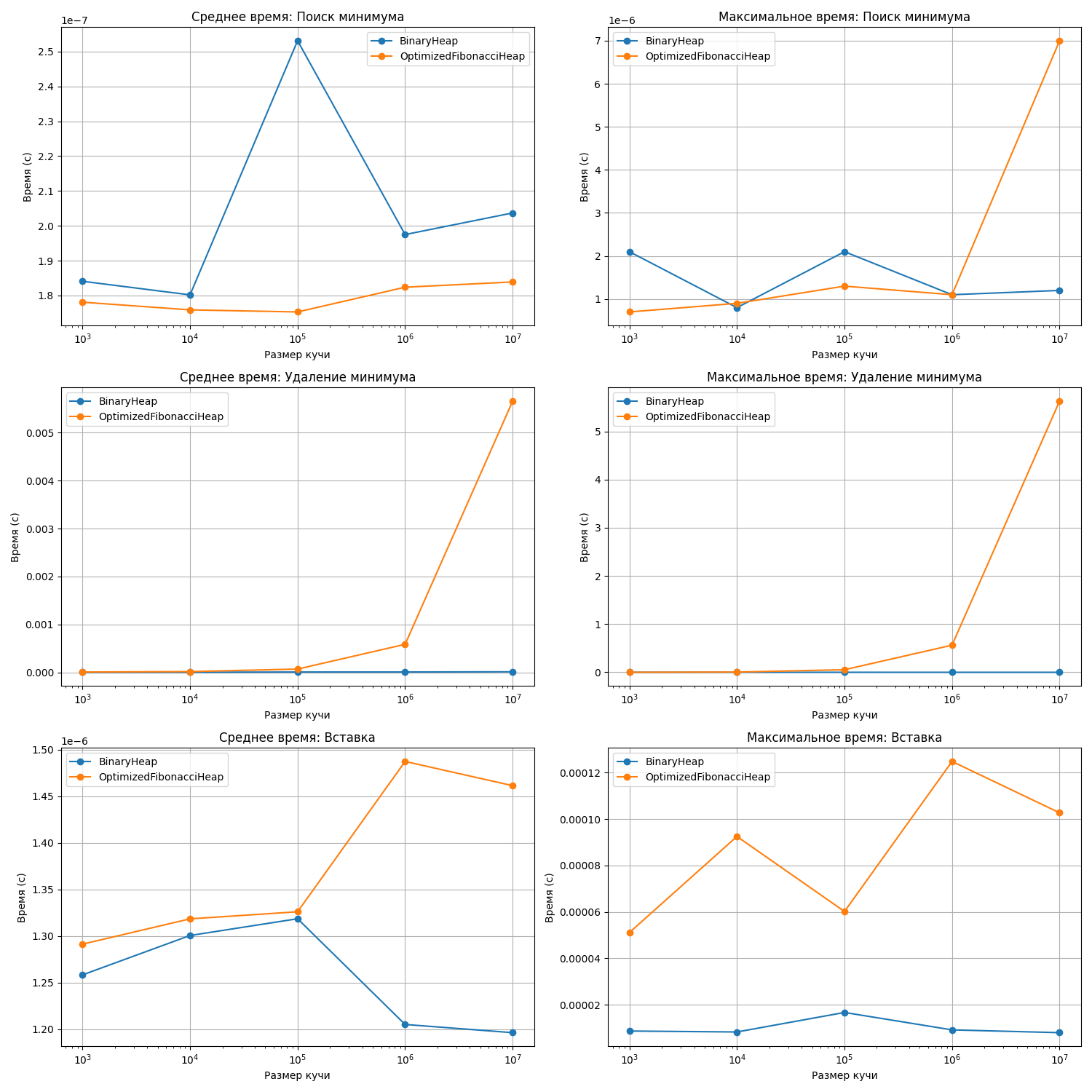
Данные действия сознательно ухудшат структуру и значит, что за это придется платить, основным взимателем этой платы является операция удаления минимума, которая проводит нормализацию дерева объединяя все созданные дубликаты деревьев.

# Выполнение задачи.

Для реализации модели и тестов я использовал язык Python из-за его простой архитектуры.

import time  
import math  
import matplotlib.pyplot as plt  
import random  
from collections import deque  
  
  
class BinaryHeap:  
 def \_\_init\_\_(self):  
 self.heap = []  
  
 def insert(self, key):  
 self.heap.append(key)  
 self.\_heapify\_up(len(self.heap) - 1)  
  
 def \_heapify\_up(self, index):  
 while index > 0:  
 parent = (index - 1) // 2  
 if self.heap[parent] <= self.heap[index]:  
 break  
 self.heap[parent], self.heap[index] = self.heap[index], self.heap[parent]  
 index = parent  
  
 def get\_min(self):  
 return self.heap[0] if self.heap else None  
  
 def delete\_min(self):  
 if not self.heap:  
 return None  
  
 self.heap[0], self.heap[-1] = self.heap[-1], self.heap[0]  
 min\_val = self.heap.pop()  
 if self.heap:  
 self.\_heapify\_down(0)  
 return min\_val  
  
 def \_heapify\_down(self, index):  
 n = len(self.heap)  
 while True:  
 left = 2 \* index + 1  
 right = 2 \* index + 2  
 smallest = index  
  
 if left < n and self.heap[left] < self.heap[smallest]:  
 smallest = left  
 if right < n and self.heap[right] < self.heap[smallest]:  
 smallest = right  
  
 if smallest == index:  
 break  
  
 self.heap[index], self.heap[smallest] = self.heap[smallest], self.heap[index]  
 index = smallest  
  
  
class OptimizedFibonacciHeap:  
 def \_\_init\_\_(self):  
 self.min\_node = None  
 self.num\_nodes = 0  
 self.\_iter\_cache = []  
  
 def insert(self, key):  
 node = FibonacciHeapNode(key)  
 self.\_add\_to\_root\_list(node)  
 if self.min\_node is None or node.key < self.min\_node.key:  
 self.min\_node = node  
 self.num\_nodes += 1  
 return node  
  
 def \_add\_to\_root\_list(self, node):  
 if self.min\_node is None:  
 self.min\_node = node  
 node.left = node.right = 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 minimum(self):  
 return self.min\_node.key if self.min\_node else None  
  
 def extract\_min(self):  
 if not self.min\_node:  
 return None  
  
 min\_node = self.min\_node  
  
 if min\_node.child:  
 self.\_iter\_cache.clear()  
 current = min\_node.child  
 while True:  
 self.\_iter\_cache.append(current)  
 current = current.right  
 if current == min\_node.child:  
 break  
  
 for child in self.\_iter\_cache:  
 self.\_add\_to\_root\_list(child)  
 child.parent = None  
  
 self.\_remove\_from\_root\_list(min\_node)  
  
 if min\_node == min\_node.right:  
 self.min\_node = None  
 else:  
 self.min\_node = min\_node.right  
 self.\_consolidate()  
  
 self.num\_nodes -= 1  
 return min\_node  
  
 def \_remove\_from\_root\_list(self, node):  
 if node.right == node:  
 self.min\_node = None  
 else:  
 node.left.right = node.right  
 node.right.left = node.left  
 if node == self.min\_node:  
 self.min\_node = node.right  
  
 def \_consolidate(self):  
 if not self.min\_node:  
 return  
  
 max\_degree = math.floor(math.log2(self.num\_nodes)) + 2  
 degree\_table = [None] \* max\_degree  
  
 nodes = []  
 current = self.min\_node  
 while True:  
 nodes.append(current)  
 current = current.right  
 if current == self.min\_node:  
 break  
  
 for node in nodes:  
 degree = node.degree  
 while degree\_table[degree] is not None:  
 other = degree\_table[degree]  
 if node.key > other.key:  
 node, other = other, node  
 self.\_link(node, other)  
 degree\_table[degree] = None  
 degree += 1  
 degree\_table[degree] = node  
  
 self.min\_node = None  
 for node in degree\_table:  
 if node is not None:  
 if self.min\_node is None:  
 self.min\_node = node  
 node.left = node.right = node  
 else:  
 self.\_add\_to\_root\_list(node)  
 if node.key < self.min\_node.key:  
 self.min\_node = node  
  
 def \_link(self, parent, child):  
 self.\_remove\_from\_root\_list(child)  
 child.parent = parent  
 child.mark = False  
  
 if parent.child is None:  
 parent.child = child  
 child.left = child.right = child  
 else:  
 child.right = parent.child.right  
 child.left = parent.child  
 parent.child.right.left = child  
 parent.child.right = child  
  
 parent.degree += 1  
  
  
class FibonacciHeapNode:  
 \_\_slots\_\_ = ('key', 'degree', 'parent', 'child', 'mark', 'left', 'right')  
  
 def \_\_init\_\_(self, key):  
 self.key = key  
 self.degree = 0  
 self.parent = None  
 self.child = None  
 self.mark = False  
 self.left = self  
 self.right = self  
  
  
def run\_operations(heap, op\_type, num\_ops=1000):  
 times = []  
  
 if op\_type == 'find\_min':  
 op = heap.minimum if isinstance(heap, OptimizedFibonacciHeap) else heap.get\_min  
 elif op\_type == 'delete\_min':  
 op = heap.extract\_min if isinstance(heap, OptimizedFibonacciHeap) else heap.delete\_min  
 elif op\_type == 'insert':  
 op = lambda: heap.insert(random.randint(1, 10 \*\* 6))  
  
 for \_ in range(num\_ops):  
 start = time.perf\_counter()  
 result = op()  
 times.append(time.perf\_counter() - start)  
 if op\_type == 'delete\_min' and result is None:  
 break  
  
 avg\_time = sum(times) / len(times) if times else 0  
 max\_time = max(times) if times else 0  
  
 return avg\_time, max\_time  
  
  
def benchmark():  
 sizes = [10 \*\* 3, 10 \*\* 4, 10 \*\* 5, 10 \*\* 6, 10 \*\* 7]  
 heaps = {  
 'BinaryHeap': BinaryHeap,  
 'OptimizedFibonacciHeap': OptimizedFibonacciHeap  
 }  
 results = {name: {} for name in heaps}  
  
 print("Начало тестирования производительности...")  
  
 for size in sizes:  
 print(f"\nРазмер кучи: {size}")  
 data = [random.randint(1, 10 \*\* 6) for \_ in range(size)]  
  
 for name, heap\_class in heaps.items():  
 print(f" Тестируем {name}...", end=' ', flush=True)  
  
 # Создание и заполнение кучи  
 start = time.perf\_counter()  
 heap = heap\_class()  
 for x in data:  
 heap.insert(x)  
 creation\_time = time.perf\_counter() - start  
  
 # Тестирование операций  
 avg\_find, max\_find = run\_operations(heap, 'find\_min')  
 avg\_del, max\_del = run\_operations(heap, 'delete\_min')  
 avg\_insert, max\_insert = run\_operations(heap, 'insert')  
  
 results[name][size] = {  
 'find\_min': {'avg': avg\_find, 'max': max\_find},  
 'delete\_min': {'avg': avg\_del, 'max': max\_del},  
 'insert': {'avg': avg\_insert, 'max': max\_insert}  
 }  
 print("готово")  
  
 # Построение графиков  
 print("\nСоздание графиков...")  
 operations = ['find\_min', 'delete\_min', 'insert']  
 op\_names = ['Поиск минимума', 'Удаление минимума', 'Вставка']  
  
 fig, axes = plt.subplots(3, 2, figsize=(15, 15))  
  
 for i, (op, op\_name) in enumerate(zip(operations, op\_names)):  
 # Графики среднего времени  
 ax\_avg = axes[i, 0]  
 for name in heaps:  
 x = sizes  
 y = [results[name][s][op]['avg'] for s in sizes]  
 ax\_avg.plot(x, y, marker='o', label=name)  
  
 ax\_avg.set\_xscale('log')  
  
 ax\_avg.set\_xlabel('Размер кучи')  
 ax\_avg.set\_ylabel('Время (с)')  
 ax\_avg.set\_title(f'Среднее время: {op\_name}')  
 ax\_avg.legend()  
 ax\_avg.grid(True)  
  
 # Графики максимального времени  
 ax\_max = axes[i, 1]  
 for name in heaps:  
 x = sizes  
 y = [results[name][s][op]['max'] for s in sizes]  
 ax\_max.plot(x, y, marker='o', label=name)  
  
 ax\_max.set\_xscale('log')  
  
 ax\_max.set\_xlabel('Размер кучи')  
 ax\_max.set\_ylabel('Время (с)')  
 ax\_max.set\_title(f'Максимальное время: {op\_name}')  
 ax\_max.legend()  
 ax\_max.grid(True)  
  
 plt.tight\_layout()  
 plt.savefig('сравнение\_производительности\_операций.png')  
 print("Графики сохранены в файл 'сравнение\_производительности\_операций.png'")  
  
  
if \_\_name\_\_ == "\_\_main\_\_":  
 benchmark()

***Результаты:***



# Заключение.

Бинарные кучи – полезная структура данных, которая упрощает задачи поиска. Однако из-за сложности реализации обычно фибоначчиевая куча используется в специализированных задачах.