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 № 7

Выполнил студент группы КС-36 Кошкарев Иван Михайлович

Ссылка на репозиторий: https://github.com/MUCTR-IKT-CPP/KoshkarevIM\_36\_ALG/tree/main/Lab7

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 Декартового дерева поиска и бинарного сбалансированного дерева поиска. Провести ряд тестов, чтобы сравнить их эффективность

# Описание метода/модели.

Декартово дерево, это двоичное дерево поиска которое является достаточно популярной и простой реализацией самобалансирующегося варианта дерева. Декартово дерево в каждом узле помимо ключа, хранит так же приоритет узла, который отражает позицию элемента в такой структуре данных как куча. О куче мы будем подробно говорить в следующих лекциях, пока же, укажем, что куча это древовидная структура у которой родитель дерева больше всех его потомков(или меньше). По этой причине декартово дерево часто называют treap = tree + heap. Такое дерево называется декартовым, по той причине, что его узлы можно уложить на координатной плоскости где х это ключ, а y это приоритет.

АВЛ дерево является обычным двоичным деревом поиска, следовательно его правое поддерево всегда меньше значения корня, а правое поддерево всегда больше. При это, при построении дерева мы руководствуемся правилом балансировки или перебалансировки: для любого узла дерева высота его правого поддерева отличается от высоты левого поддерева не более чем на единицу. Является доказанным, что при соблюдении этого правила высота дерева логарифмически зависит от количества элементов добавляемых в дерево, т.е. h = O(log(n)).

Узел AVL дерева, содержит:

Ключ

Высоту – от лепестков

Левый потомок

Правый потомок

По поводу высоты, можно учесть важный момент, классически атрибут высоты содержит не высоту, а разницу, которая может быть -1,0,1, так как в остальных случаях вызывается перебалансировка, но из за логарифмической природы высоты, можно смело хранить само значение, но тогда придется всегда рассчитывать фактор перебалансировки, который будет разницей между высотами правого и левого поддерева.

Операция вставки и удаления вызываются практически так же, как у обычного двоичного дерева. Разница только в том, что для каждой вставки и каждого удаления требуется последним действием вызывать операцию перебалансировки, каждый раз проверяя от низа к верху необходимость ребалансировать дерево в текущем узле.

Операция балансировки вызывается тогда, когда фактор балансировки становится равным 2 или -2, т.е. тогда, когда разница между правым и левым поддеревьями является больше чем заложено в правиле. В этом случае требуется выполнить операцию поворота дерева, которая переориентирует узлы так, что они изменяют свое положение решая проблему дисбаланса.

# Выполнение задачи.

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

import time  
import random  
import matplotlib.pyplot as plt  
from statistics import mean  
import sys  
  
  
class Node:  
 *"""Узел дерева, используется как в Treap, так и в AVL"""* def \_\_init\_\_(self, key, priority=None):  
 self.key = key # Ключ узла  
 self.priority = priority if priority is not None else random.random() # Приоритет для Treap  
 self.left = None # Левый потомок  
 self.right = None # Правый потомок  
 self.height = 1 # Высота поддерева (для AVL)  
  
  
class Treap:  
 *"""Реализация структуры данных Treap (декартово дерево)"""* def \_\_init\_\_(self):  
 self.root = None # Корень дерева  
  
 def insert(self, key, priority=None):  
 *"""Вставка ключа в Treap"""* self.root = self.\_insert(self.root, key, priority)  
  
 def \_insert(self, node, key, priority=None):  
 *"""Рекурсивная вставка ключа в поддерево"""* if not node:  
 return Node(key, priority) # Создаем новый узел, если достигли пустого места  
  
 if key < node.key:  
 node.left = self.\_insert(node.left, key, priority)  
 # Поддерживаем свойство кучи: если приоритет левого потомка больше  
 if node.left.priority > node.priority:  
 node = self.\_rotate\_right(node) # Правое вращение  
 else:  
 node.right = self.\_insert(node.right, key, priority)  
 # Поддерживаем свойство кучи: если приоритет правого потомка больше  
 if node.right.priority > node.priority:  
 node = self.\_rotate\_left(node) # Левое вращение  
 return node  
  
 def \_rotate\_right(self, y):  
 *"""Правое вращение вокруг узла y"""* x = y.left  
 y.left = x.right  
 x.right = y  
 return x  
  
 def \_rotate\_left(self, x):  
 *"""Левое вращение вокруг узла x"""* y = x.right  
 x.right = y.left  
 y.left = x  
 return y  
  
 def search(self, key):  
 *"""Поиск ключа в дереве"""* return self.\_search(self.root, key)  
  
 def \_search(self, node, key):  
 *"""Рекурсивный поиск ключа в поддереве"""* if not node:  
 return False  
 if key == node.key:  
 return True  
 elif key < node.key:  
 return self.\_search(node.left, key)  
 else:  
 return self.\_search(node.right, key)  
  
 def delete(self, key):  
 *"""Удаление ключа из дерева"""* self.root = self.\_delete(self.root, key)  
  
 def \_delete(self, node, key):  
 *"""Рекурсивное удаление ключа из поддерева"""* if not node:  
 return node  
  
 if key < node.key:  
 node.left = self.\_delete(node.left, key)  
 elif key > node.key:  
 node.right = self.\_delete(node.right, key)  
 else:  
 # Узел найден - выполняем удаление  
 if not node.left:  
 return node.right  
 if not node.right:  
 return node.left  
  
 # Выбираем вращение в зависимости от приоритетов  
 if node.left.priority > node.right.priority:  
 node = self.\_rotate\_right(node)  
 node.right = self.\_delete(node.right, key)  
 else:  
 node = self.\_rotate\_left(node)  
 node.left = self.\_delete(node.left, key)  
 return node  
  
 def max\_depth(self):  
 *"""Вычисление максимальной глубины дерева"""* return self.\_max\_depth(self.root)  
  
 def \_max\_depth(self, node):  
 *"""Рекурсивное вычисление максимальной глубины поддерева"""* if not node:  
 return 0  
 return 1 + max(self.\_max\_depth(node.left), self.\_max\_depth(node.right))  
  
 def get\_all\_depths(self):  
 *"""Получение глубин всех листьев дерева"""* depths = []  
 self.\_get\_all\_depths(self.root, 1, depths)  
 return depths  
  
 def \_get\_all\_depths(self, node, current\_depth, depths):  
 *"""Рекурсивный сбор глубин листьев"""* if not node:  
 return  
 if not node.left and not node.right: # Если это лист  
 depths.append(current\_depth)  
 self.\_get\_all\_depths(node.left, current\_depth + 1, depths)  
 self.\_get\_all\_depths(node.right, current\_depth + 1, depths)  
  
  
class AVL:  
 *"""Реализация структуры данных AVL-дерево"""* def \_\_init\_\_(self):  
 self.root = None # Корень дерева  
  
 def insert(self, key):  
 *"""Вставка ключа в AVL-дерево"""* self.root = self.\_insert(self.root, key)  
  
 def \_insert(self, node, key):  
 *"""Рекурсивная вставка ключа в поддерево"""* if not node:  
 return Node(key)  
  
 if key < node.key:  
 node.left = self.\_insert(node.left, key)  
 else:  
 node.right = self.\_insert(node.right, key)  
  
 # Обновляем высоту текущего узла  
 node.height = 1 + max(self.\_get\_height(node.left),  
 self.\_get\_height(node.right))  
  
 # Проверяем баланс и выполняем балансировку  
 balance = self.\_get\_balance(node)  
  
 # Левое поддерево перевешивает (LL-случай)  
 if balance > 1 and key < node.left.key:  
 return self.\_right\_rotate(node)  
  
 # Правое поддерево перевешивает (RR-случай)  
 if balance < -1 and key > node.right.key:  
 return self.\_left\_rotate(node)  
  
 # Левое-правое поддерево (LR-случай)  
 if balance > 1 and key > node.left.key:  
 node.left = self.\_left\_rotate(node.left)  
 return self.\_right\_rotate(node)  
  
 # Правое-левое поддерево (RL-случай)  
 if balance < -1 and key < node.right.key:  
 node.right = self.\_right\_rotate(node.right)  
 return self.\_left\_rotate(node)  
  
 return node  
  
 def delete(self, key):  
 *"""Удаление ключа из AVL-дерева"""* self.root = self.\_delete(self.root, key)  
  
 def \_delete(self, node, key):  
 *"""Рекурсивное удаление ключа из поддерева"""* if not node:  
 return node  
  
 if key < node.key:  
 node.left = self.\_delete(node.left, key)  
 elif key > node.key:  
 node.right = self.\_delete(node.right, key)  
 else:  
 # Узел найден - выполняем удаление  
 if node.left is None:  
 return node.right  
 elif node.right is None:  
 return node.left  
  
 # Находим минимальный узел в правом поддереве  
 temp = self.\_min\_value\_node(node.right)  
 node.key = temp.key  
 node.right = self.\_delete(node.right, temp.key)  
  
 if node is None:  
 return node  
  
 # Обновляем высоту текущего узла  
 node.height = 1 + max(self.\_get\_height(node.left),  
 self.\_get\_height(node.right))  
  
 # Проверяем баланс и выполняем балансировку  
 balance = self.\_get\_balance(node)  
  
 # Балансировка после удаления  
 if balance > 1 and self.\_get\_balance(node.left) >= 0:  
 return self.\_right\_rotate(node)  
  
 if balance < -1 and self.\_get\_balance(node.right) <= 0:  
 return self.\_left\_rotate(node)  
  
 if balance > 1 and self.\_get\_balance(node.left) < 0:  
 node.left = self.\_left\_rotate(node.left)  
 return self.\_right\_rotate(node)  
  
 if balance < -1 and self.\_get\_balance(node.right) > 0:  
 node.right = self.\_right\_rotate(node.right)  
 return self.\_left\_rotate(node)  
  
 return node  
  
 def \_left\_rotate(self, z):  
 *"""Левое вращение вокруг узла z"""* if z is None or z.right is None:  
 return z  
  
 y = z.right  
 T2 = y.left  
  
 y.left = z  
 z.right = T2  
  
 z.height = 1 + max(self.\_get\_height(z.left),  
 self.\_get\_height(z.right))  
 y.height = 1 + max(self.\_get\_height(y.left),  
 self.\_get\_height(y.right))  
  
 return y  
  
 def \_right\_rotate(self, z):  
 *"""Правое вращение вокруг узла z"""* if z is None or z.left is None:  
 return z  
  
 y = z.left  
 T3 = y.right  
  
 y.right = z  
 z.left = T3  
  
 z.height = 1 + max(self.\_get\_height(z.left),  
 self.\_get\_height(z.right))  
 y.height = 1 + max(self.\_get\_height(y.left),  
 self.\_get\_height(y.right))  
  
 return y  
  
 def \_get\_height(self, node):  
 *"""Получение высоты поддерева"""* if not node:  
 return 0  
 return node.height  
  
 def \_get\_balance(self, node):  
 *"""Вычисление баланса узла (разница высот поддеревьев)"""* if not node:  
 return 0  
 return self.\_get\_height(node.left) - self.\_get\_height(node.right)  
  
 def \_min\_value\_node(self, node):  
 *"""Поиск узла с минимальным ключом в поддереве"""* current = node  
 while current.left:  
 current = current.left  
 return current  
  
 def search(self, key):  
 *"""Поиск ключа в дереве"""* return self.\_search(self.root, key)  
  
 def \_search(self, node, key):  
 *"""Рекурсивный поиск ключа в поддереве"""* if not node:  
 return False  
 if key == node.key:  
 return True  
 elif key < node.key:  
 return self.\_search(node.left, key)  
 else:  
 return self.\_search(node.right, key)  
  
 def max\_depth(self):  
 *"""Вычисление максимальной глубины дерева"""* return self.\_max\_depth(self.root)  
  
 def \_max\_depth(self, node):  
 *"""Рекурсивное вычисление максимальной глубины поддерева"""* if not node:  
 return 0  
 return 1 + max(self.\_max\_depth(node.left), self.\_max\_depth(node.right))  
  
 def get\_all\_depths(self):  
 *"""Получение глубин всех листьев дерева"""* depths = []  
 self.\_get\_all\_depths(self.root, 1, depths)  
 return depths  
  
 def \_get\_all\_depths(self, node, current\_depth, depths):  
 *"""Рекурсивный сбор глубин листьев"""* if not node:  
 return  
 if not node.left and not node.right: # Если это лист  
 depths.append(current\_depth)  
 self.\_get\_all\_depths(node.left, current\_depth + 1, depths)  
 self.\_get\_all\_depths(node.right, current\_depth + 1, depths)  
  
  
def generate\_random\_array(size):  
 *"""Генерация массива случайных чисел"""* return [random.randint(0, 10 \* size) for \_ in range(size)]  
  
  
def run\_comparison\_tests():  
 *"""Запуск тестов для сравнения Treap и AVL деревьев"""* sizes = [2 \*\* i for i in range(10, 16)] # Размеры массивов от 2^10 до 2^15  
 num\_repeats = 50 # Количество повторений для каждого размера  
  
 # Словарь для хранения результатов  
 results = {  
 'Treap': {  
 'insert\_time': [],  
 'delete\_time': [],  
 'search\_time': [],  
 'max\_height': [],  
 'avg\_depth': [], # Добавляем среднюю глубину веток  
 'all\_depths': []  
 },  
 'AVL': {  
 'insert\_time': [],  
 'delete\_time': [],  
 'search\_time': [],  
 'max\_height': [],  
 'avg\_depth': [], # Добавляем среднюю глубину веток  
 'all\_depths': []  
 }  
 }  
  
 for n in sizes:  
 print(f"\nТестирование размера: {n}")  
 treap\_insert\_times = []  
 treap\_delete\_times = []  
 treap\_search\_times = []  
 treap\_max\_heights = []  
 treap\_avg\_depths = [] # Средние глубины для каждого теста Treap  
 treap\_all\_depths = []  
  
 avl\_insert\_times = []  
 avl\_delete\_times = []  
 avl\_search\_times = []  
 avl\_max\_heights = []  
 avl\_avg\_depths = [] # Средние глубины для каждого теста AVL  
 avl\_all\_depths = []  
  
 for repeat in range(num\_repeats):  
 print(f" Повтор {repeat + 1}/{num\_repeats}", end="\r")  
 data = generate\_random\_array(n)  
  
 # Тестирование Treap  
 start = time.time()  
 treap = Treap()  
 for key in data:  
 treap.insert(key)  
 treap\_insert\_times.append(time.time() - start)  
  
 max\_h = treap.max\_depth()  
 treap\_max\_heights.append(max\_h)  
  
 depths = treap.get\_all\_depths()  
 treap\_all\_depths.extend(depths)  
 avg\_depth = mean(depths) if depths else 0  
 treap\_avg\_depths.append(avg\_depth)  
  
 search\_data = random.choices(data, k=100)  
 start = time.time()  
 for key in search\_data:  
 treap.search(key)  
 treap\_search\_times.append((time.time() - start) / 100)  
  
 delete\_data = random.choices(data, k=100)  
 start = time.time()  
 for key in delete\_data:  
 treap.delete(key)  
 treap\_delete\_times.append((time.time() - start) / 100)  
  
 # Тестирование AVL  
 start = time.time()  
 avl = AVL()  
 for key in data:  
 avl.insert(key)  
 avl\_insert\_times.append(time.time() - start)  
  
 max\_h = avl.max\_depth()  
 avl\_max\_heights.append(max\_h)  
  
 depths = avl.get\_all\_depths()  
 avl\_all\_depths.extend(depths)  
 avg\_depth = mean(depths) if depths else 0  
 avl\_avg\_depths.append(avg\_depth)  
  
 start = time.time()  
 for key in search\_data:  
 avl.search(key)  
 avl\_search\_times.append((time.time() - start) / 100)  
  
 start = time.time()  
 for key in delete\_data:  
 avl.delete(key)  
 avl\_delete\_times.append((time.time() - start) / 100)  
  
 # Сохранение средних результатов  
 results['Treap']['insert\_time'].append(mean(treap\_insert\_times))  
 results['Treap']['delete\_time'].append(mean(treap\_delete\_times))  
 results['Treap']['search\_time'].append(mean(treap\_search\_times))  
 results['Treap']['max\_height'].append(mean(treap\_max\_heights))  
 results['Treap']['avg\_depth'].append(mean(treap\_avg\_depths))  
 results['Treap']['all\_depths'].append(treap\_all\_depths)  
  
 results['AVL']['insert\_time'].append(mean(avl\_insert\_times))  
 results['AVL']['delete\_time'].append(mean(avl\_delete\_times))  
 results['AVL']['search\_time'].append(mean(avl\_search\_times))  
 results['AVL']['max\_height'].append(mean(avl\_max\_heights))  
 results['AVL']['avg\_depth'].append(mean(avl\_avg\_depths))  
 results['AVL']['all\_depths'].append(avl\_all\_depths)  
  
 # Вывод статистики для текущего размера  
 print("\nРезультаты для размера", n)  
 print("Treap:")  
 print(f" Средняя максимальная глубина: {mean(treap\_max\_heights):.2f}")  
 print(f" Среднее время вставки: {mean(treap\_insert\_times):.6f} сек")  
 print(f" Среднее время удаления: {mean(treap\_delete\_times):.6f} сек")  
 print(f" Среднее время поиска: {mean(treap\_search\_times):.6f} сек")  
 print(f" Средняя глубина веток: {mean(treap\_avg\_depths):.2f}")  
  
 print("\nAVL:")  
 print(f" Средняя максимальная глубина: {mean(avl\_max\_heights):.2f}")  
 print(f" Среднее время вставки: {mean(avl\_insert\_times):.6f} сек")  
 print(f" Среднее время удаления: {mean(avl\_delete\_times):.6f} сек")  
 print(f" Среднее время поиска: {mean(avl\_search\_times):.6f} сек")  
 print(f" Средняя глубина веток: {mean(avl\_avg\_depths):.2f}")  
 print("----------------------------------------")  
  
 return sizes, results  
  
  
def plot\_results(sizes, results):  
 *"""Построение графиков с результатами сравнения"""* # 1. График времени вставки  
 plt.figure(figsize=(12, 6))  
 plt.plot(sizes, results['Treap']['insert\_time'], 'o-', label='Treap')  
 plt.plot(sizes, results['AVL']['insert\_time'], 'o-', label='AVL')  
 plt.xscale('log', base=2)  
 plt.yscale('log')  
 plt.xlabel('Количество элементов')  
 plt.ylabel('Время (секунды)')  
 plt.title('Сравнение времени вставки')  
 plt.legend()  
 plt.grid(True)  
 plt.show()  
  
 # 2. График времени удаления  
 plt.figure(figsize=(12, 6))  
 plt.plot(sizes, results['Treap']['delete\_time'], 'o-', label='Treap')  
 plt.plot(sizes, results['AVL']['delete\_time'], 'o-', label='AVL')  
 plt.xscale('log', base=2)  
 plt.yscale('log')  
 plt.xlabel('Количество элементов')  
 plt.ylabel('Время (секунды)')  
 plt.title('Сравнение времени удаления')  
 plt.legend()  
 plt.grid(True)  
 plt.show()  
  
 # 3. График времени поиска  
 plt.figure(figsize=(12, 6))  
 plt.plot(sizes, results['Treap']['search\_time'], 'o-', label='Treap')  
 plt.plot(sizes, results['AVL']['search\_time'], 'o-', label='AVL')  
 plt.xscale('log', base=2)  
 plt.yscale('log')  
 plt.xlabel('Количество элементов')  
 plt.ylabel('Время (секунды)')  
 plt.title('Сравнение времени поиска')  
 plt.legend()  
 plt.grid(True)  
 plt.show()  
  
 # 4. График максимальной высоты  
 plt.figure(figsize=(12, 6))  
 plt.plot(sizes, results['Treap']['max\_height'], 'o-', label='Treap')  
 plt.plot(sizes, results['AVL']['max\_height'], 'o-', label='AVL')  
 plt.xscale('log', base=2)  
 plt.xlabel('Количество элементов')  
 plt.ylabel('Максимальная высота')  
 plt.title('Сравнение высоты деревьев')  
 plt.legend()  
 plt.grid(True)  
 plt.show()  
  
 # 5. Гистограмма распределения высот (N=2^15)  
 plt.figure(figsize=(12, 6))  
 plt.hist(results['Treap']['all\_depths'][-1], bins=50, alpha=0.5, label='Treap')  
 plt.hist(results['AVL']['all\_depths'][-1], bins=50, alpha=0.5, label='AVL')  
 plt.xlabel('Глубина')  
 plt.ylabel('Частота')  
 plt.title(f'Распределение глубин (N={sizes[-1]})')  
 plt.legend()  
 plt.grid(True)  
 plt.show()  
  
 # 6. Гистограмма средних максимальных высот  
 plt.figure(figsize=(12, 6))  
 plt.hist(results['Treap']['max\_height'], bins=20, alpha=0.5, label='Treap')  
 plt.hist(results['AVL']['max\_height'], bins=20, alpha=0.5, label='AVL')  
 plt.xlabel('Максимальная высота')  
 plt.ylabel('Частота')  
 plt.title('Распределение максимальных высот')  
 plt.legend()  
 plt.grid(True)  
 plt.show()  
  
  
if \_\_name\_\_ == "\_\_main\_\_":  
 sys.setrecursionlimit(1000000) # Увеличиваем лимит рекурсии для больших деревьев  
 print("Начало тестирования...")  
 sizes, results = run\_comparison\_tests()  
 print("\nТестирование завершено. Построение графиков...")  
 plot\_results(sizes, results)  
 print("Готово!")

***Результаты:***

Начало тестирования...

Тестирование размера: 1024

Повтор 50/50

Результаты для размера 1024

Treap:

Средняя максимальная глубина: 22.02

Среднее время вставки: 0.001079 сек

Среднее время удаления: 0.000000 сек

Среднее время поиска: 0.000000 сек

Средняя глубина веток: 13.40

AVL:

Средняя максимальная глубина: 12.00

Среднее время вставки: 0.003393 сек

Среднее время удаления: 0.000012 сек

Среднее время поиска: 0.000001 сек

Средняя глубина веток: 10.34

----------------------------------------

Тестирование размера: 2048

Повтор 50/50

Результаты для размера 2048

Treap:

Средняя максимальная глубина: 25.16

Среднее время вставки: 0.003354 сек

Среднее время удаления: 0.000000 сек

Среднее время поиска: 0.000000 сек

Средняя глубина веток: 14.88

AVL:

Средняя максимальная глубина: 13.04

Среднее время вставки: 0.009124 сек

Среднее время удаления: 0.000004 сек

Среднее время поиска: 0.000000 сек

Средняя глубина веток: 11.36

----------------------------------------

Тестирование размера: 4096

Повтор 50/50

Результаты для размера 4096

Treap:

Средняя максимальная глубина: 27.82

Среднее время вставки: 0.007067 сек

Среднее время удаления: 0.000000 сек

Среднее время поиска: 0.000000 сек

Средняя глубина веток: 16.03

AVL:

Средняя максимальная глубина: 14.40

Среднее время вставки: 0.019137 сек

Среднее время удаления: 0.000011 сек

Среднее время поиска: 0.000000 сек

Средняя глубина веток: 12.38

----------------------------------------

Тестирование размера: 8192

Повтор 50/50

Результаты для размера 8192

Treap:

Средняя максимальная глубина: 30.58

Среднее время вставки: 0.016435 сек

Среднее время удаления: 0.000001 сек

Среднее время поиска: 0.000001 сек

Средняя глубина веток: 17.61

AVL:

Средняя максимальная глубина: 15.76

Среднее время вставки: 0.043613 сек

Среднее время удаления: 0.000005 сек

Среднее время поиска: 0.000001 сек

Средняя глубина веток: 13.39

----------------------------------------

Тестирование размера: 16384

Повтор 50/50

Результаты для размера 16384

Treap:

Средняя максимальная глубина: 32.92

Среднее время вставки: 0.036823 сек

Среднее время удаления: 0.000002 сек

Среднее время поиска: 0.000003 сек

Средняя глубина веток: 18.94

AVL:

Средняя максимальная глубина: 16.98

Среднее время вставки: 0.093847 сек

Среднее время удаления: 0.000005 сек

Среднее время поиска: 0.000001 сек

Средняя глубина веток: 14.41

----------------------------------------

Тестирование размера: 32768

Результаты для размера 32768

Treap:

Средняя максимальная глубина: 35.76

Среднее время вставки: 0.089160 сек

Среднее время удаления: 0.000002 сек

Среднее время поиска: 0.000002 сек

Средняя глубина веток: 20.13

AVL:

Средняя максимальная глубина: 18.00

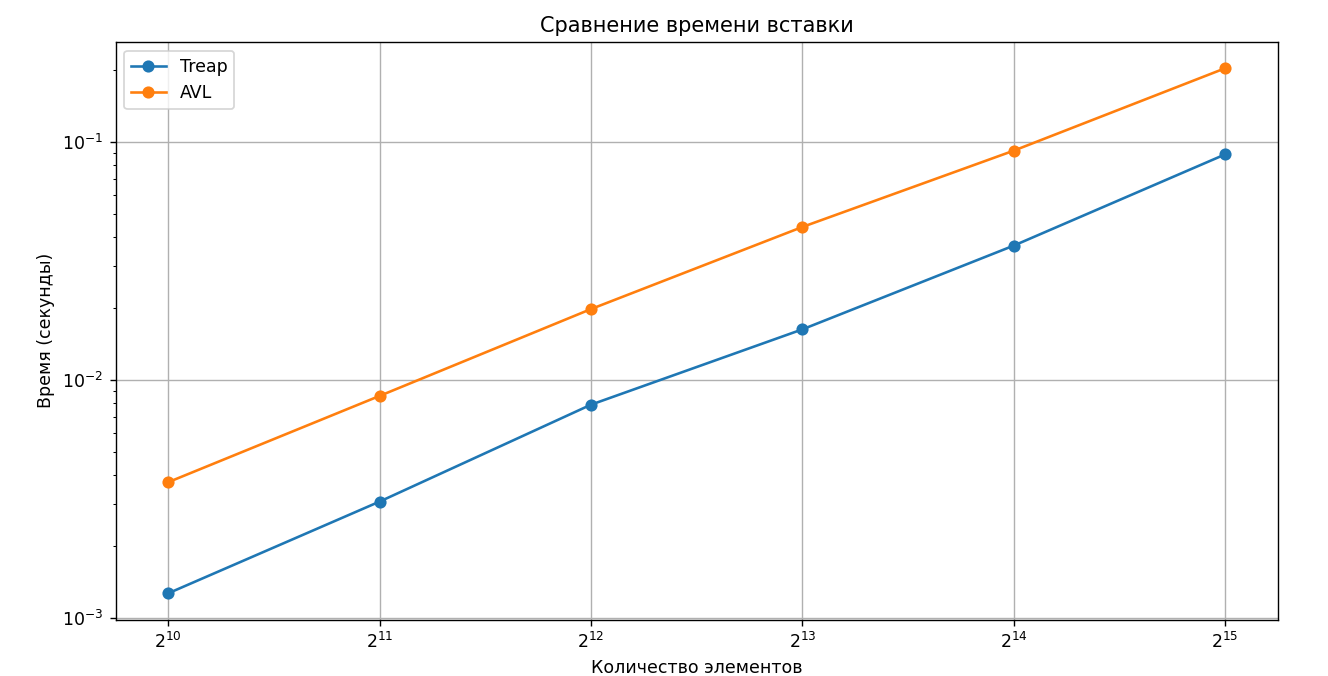
Среднее время вставки: 0.210860 сек

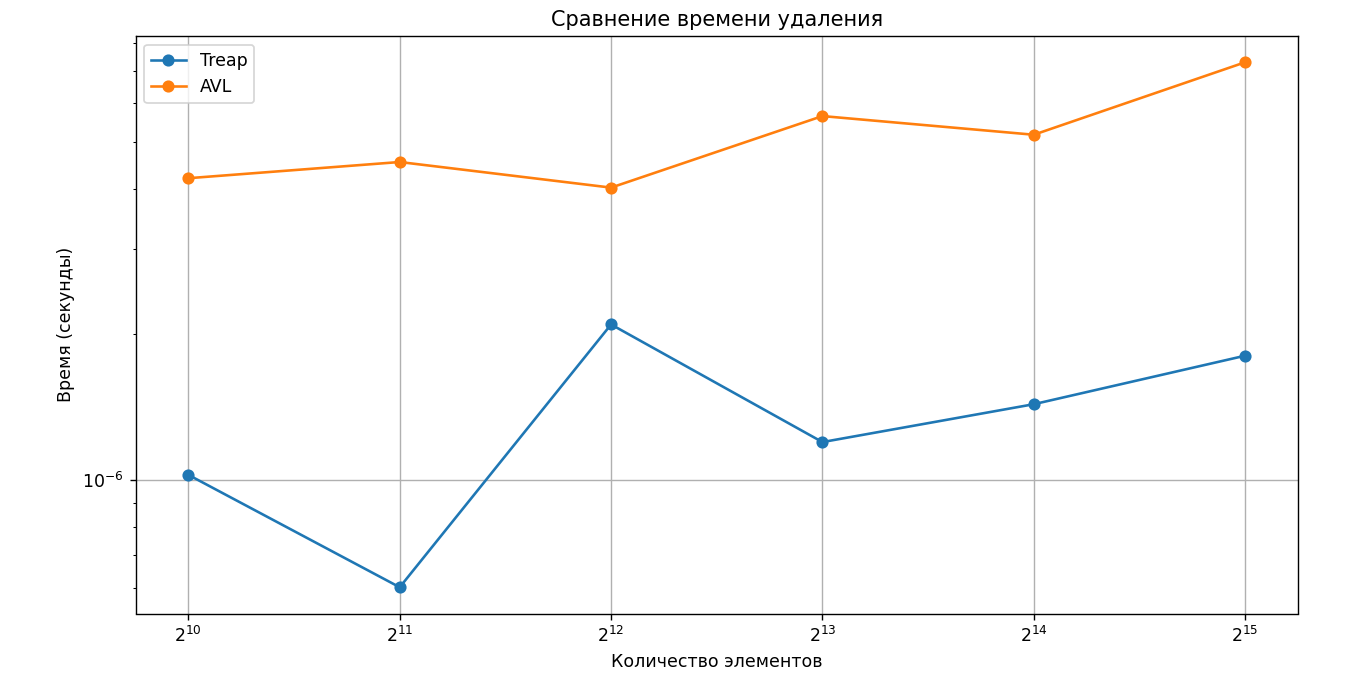
Среднее время удаления: 0.000008 сек

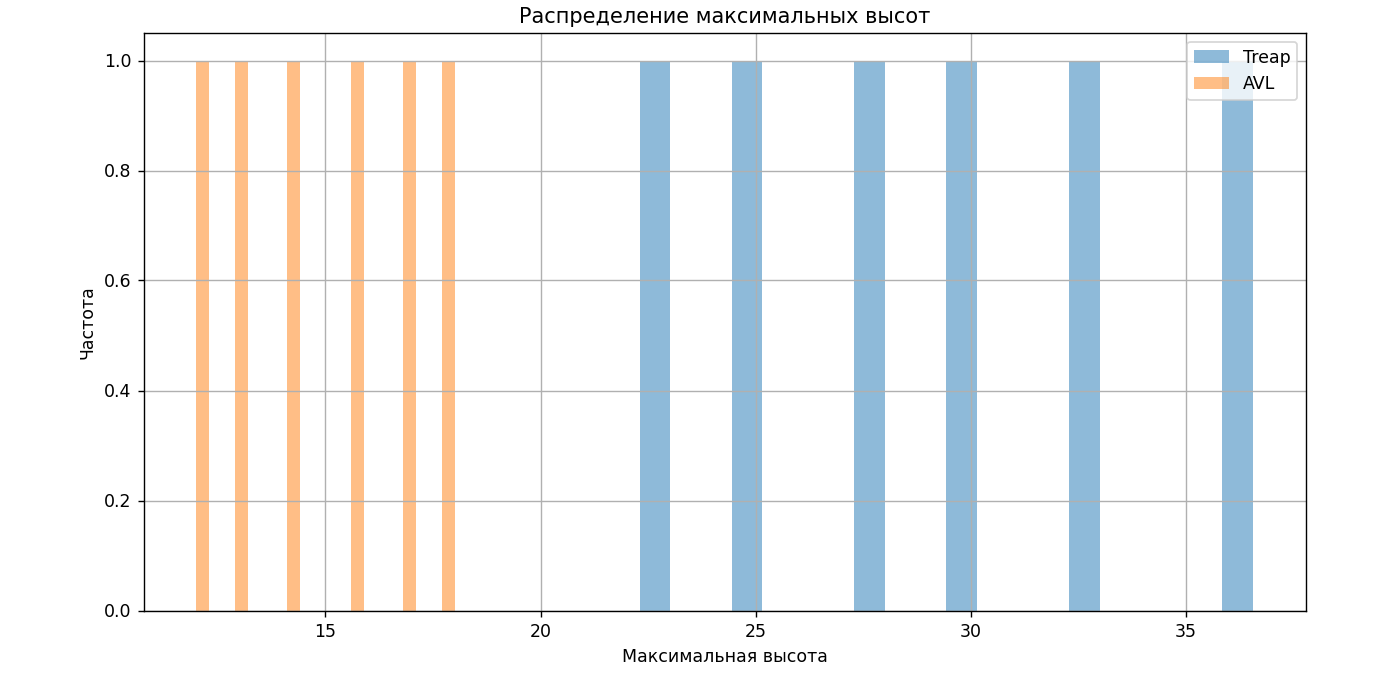
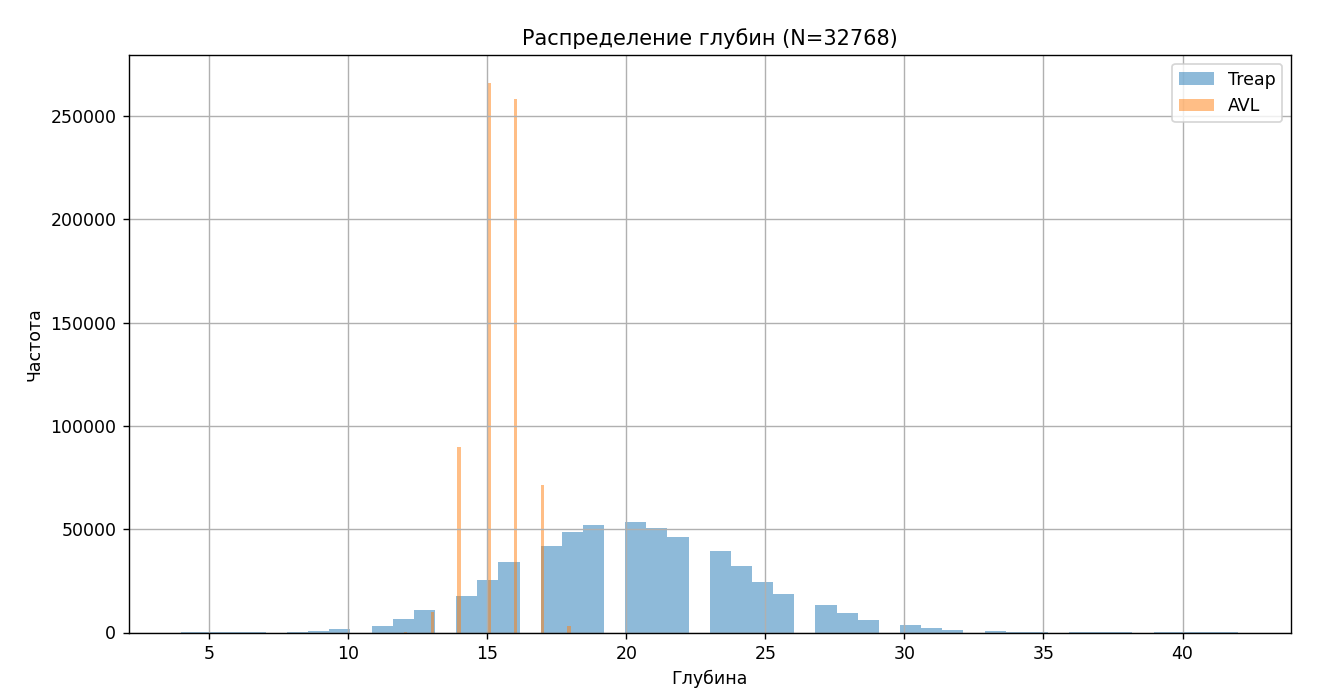
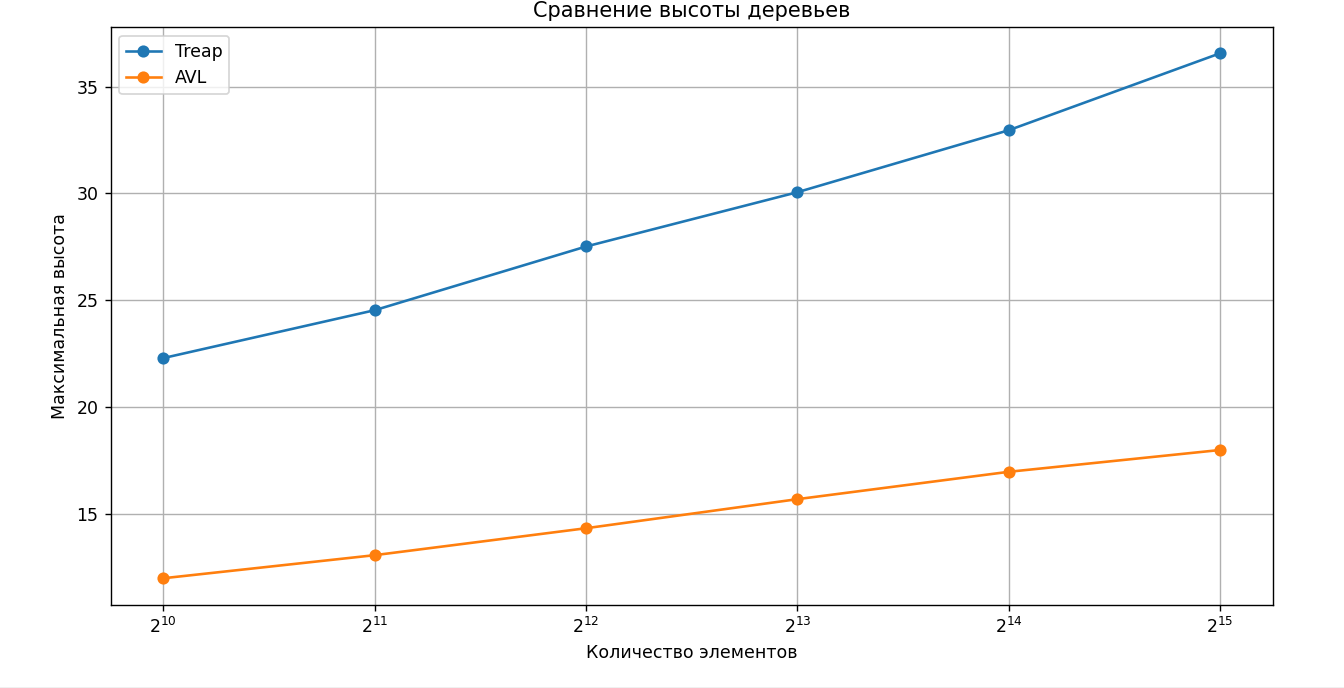
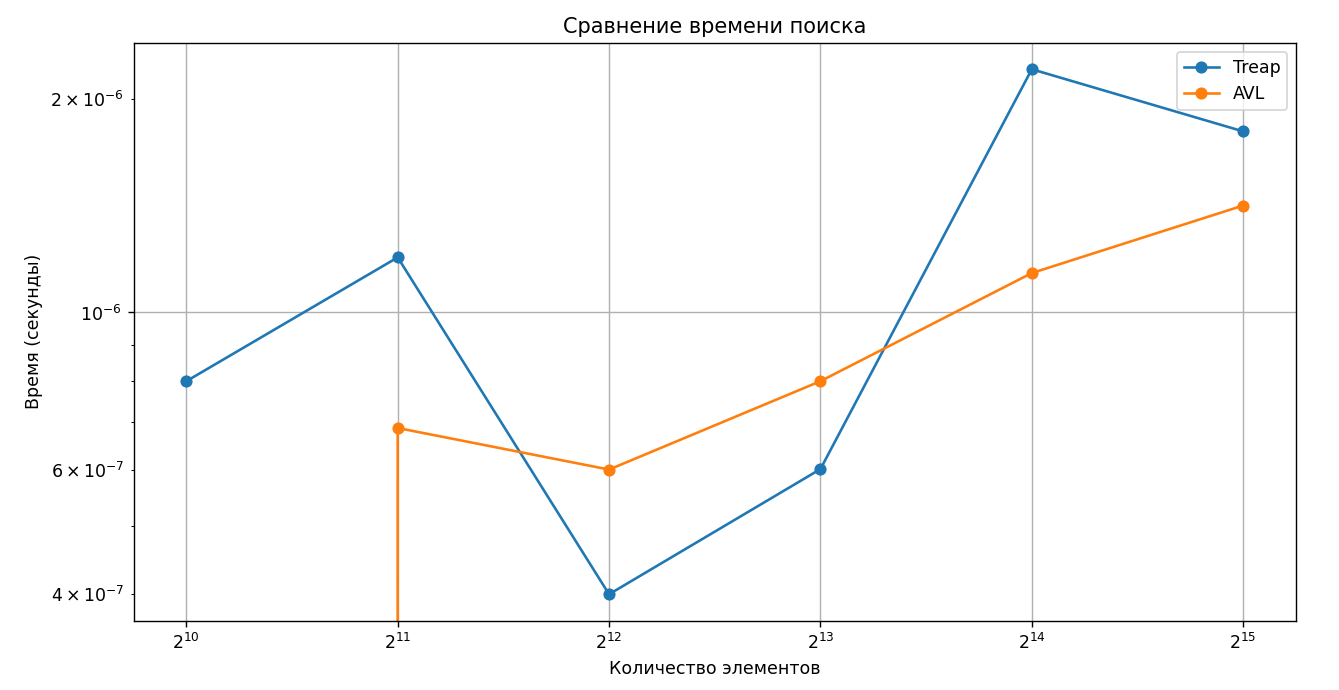
Среднее время поиска: 0.000001 сек

Средняя глубина веток: 15.42

----------------------------------------

Тестирование завершено.





# Заключение.

Бинарные поисковые деревья – полезный инструмент для поиска. В зависимости от ситуации можно использовать как самобалансируемое бинарное поисковое дерево, так и Декартово.