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 № 6

Выполнил студент группы КС-36 Кошкарев Иван Михайлович

Ссылка на репозиторий: https://github.com/MUCTR-IKT-CPP/KoshkarevIM\_36\_ALG/tree/main/Lab1

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

# Описание метода/модели.

Двоичное дерево – это иерархическая структура, в которой каждый узел содержит не более чем двух потомков.

Для каждого узла, тот узел который стоит выше по иерархии для него называют родительским узлом, а те узлы, что стоят ниже, для которых этот узел является родительским, называются правым и левым наследниками.

Двоичное дерево поиска – это двоичное дерево придерживающееся 2х правил, согласно которым левые потомки всегда меньше или равны текущего элемента, а правые больше или равны текущего элемента.

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  
import sys  
from statistics import mean  
  
"""  
Класс Node представляет узел дерева с ключом и указателями на левого/правого потомка.  
Для AVL-дерева также хранится высота поддерева.  
"""  
  
  
class Node:  
 def \_\_init\_\_(self, key):  
 self.key = key # Значение узла  
 self.left = None # Левый потомок  
 self.right = None # Правый потомок  
 self.height = 1 # Высота поддерева (для AVL)  
  
  
"""  
Класс BST реализует бинарное дерево поиска с основными операциями:  
- вставка (insert)  
- поиск (search)   
- удаление (delete)  
- обход (in\_order)  
"""  
  
  
class BST:  
 def \_\_init\_\_(self):  
 self.root = None # Корень дерева  
  
 # Вставка нового ключа в дерево (итеративная реализация)  
 # Вставка нового ключа в дерево (итеративная реализация)  
 def insert(self, key):  
 if not self.root:  
 self.root = Node(key)  
 return  
  
 current = self.root  
 while True:  
 if key < current.key: # Идем в левое поддерево  
 if not current.left:  
 current.left = Node(key)  
 break  
 else:  
 current = current.left  
 else: # Идем в правое поддерево  
 if not current.right:  
 current.right = Node(key)  
 break  
 else:  
 current = current.right  
  
 # Рекурсивный поиск  
 def search(self, key):  
 return self.\_search\_recursive(self.root, key)  
  
 def \_search\_recursive(self, node, key):  
 if node is None:  
 return False  
  
 if key == node.key:  
 return True  
 elif key < node.key:  
 return self.\_search\_recursive(node.left, key)  
 else:  
 return self.\_search\_recursive(node.right, key)  
  
 # Рекурсивный обход (in-order)  
 def in\_order(self):  
 elements = []  
 self.\_in\_order\_recursive(self.root, elements)  
 return elements  
  
 def \_in\_order\_recursive(self, node, elements):  
 if node:  
 self.\_in\_order\_recursive(node.left, elements)  
 elements.append(node.key)  
 self.\_in\_order\_recursive(node.right, elements)  
  
 # Удаление ключа из дерева  
 def delete(self, key):  
 self.root = self.\_delete(self.root, key)  
  
 # Вспомогательная рекурсивная функция для удаления  
 def \_delete(self, node, key):  
 if node is None:  
 return node  
  
 if key < node.key: # Ищем в левом поддереве  
 node.left = self.\_delete(node.left, key)  
 elif key > node.key: # Ищем в правом поддереве  
 node.right = self.\_delete(node.right, key)  
 else: # Нашли узел для удаления  
 # Узел с одним потомком или без потомков  
 if node.left is None:  
 return node.right  
 elif node.right is None:  
 return node.left  
  
 # Узел с двумя потомками - находим минимальный в правом поддереве  
 temp = self.\_min\_value\_node(node.right)  
 node.key = temp.key # Копируем значение  
 node.right = self.\_delete(node.right, temp.key) # Удаляем дубликат  
  
 return node  
  
 # Поиск узла с минимальным значением в поддереве  
 def \_min\_value\_node(self, node):  
 current = node  
 while current.left:  
 current = current.left  
 return current  
  
  
  
"""  
Класс AVL расширяет BST, добавляя балансировку для поддержания   
оптимальной высоты дерева после операций вставки и удаления.  
"""  
  
  
class AVL(BST):  
 # Переопределяем вставку с балансировкой  
 def insert(self, key):  
 self.root = self.\_insert(self.root, key)  
  
 # Рекурсивная вставка с обновлением высот и балансировкой  
 def \_insert(self, node, key):  
 if not node:  
 return Node(key) # Базовый случай рекурсии  
  
 # Обычная вставка как в BST  
 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 повороты при необходимости  
 balance = self.\_get\_balance(node)  
  
 # Левое-левое нарушение  
 if balance > 1 and key < node.left.key:  
 return self.\_right\_rotate(node)  
  
 # Правое-правое нарушение  
 if balance < -1 and key > node.right.key:  
 return self.\_left\_rotate(node)  
  
 # Левое-правое нарушение  
 if balance > 1 and key > node.left.key:  
 node.left = self.\_left\_rotate(node.left)  
 return self.\_right\_rotate(node)  
  
 # Правое-левое нарушение  
 if balance < -1 and key < node.right.key:  
 node.right = self.\_right\_rotate(node.right)  
 return self.\_left\_rotate(node)  
  
 return node  
  
 # Переопределяем удаление с балансировкой  
 def delete(self, key):  
 self.root = self.\_delete(self.root, key)  
  
 # Расширенное удаление с балансировкой  
 def \_delete(self, node, key):  
 # Сначала выполняем стандартное удаление BST  
 node = super().\_delete(node, 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 повороты  
 balance = self.\_get\_balance(node)  
  
 # Левое-левое  
 if balance > 1 and self.\_get\_balance(node.left) >= 0:  
 return self.\_right\_rotate(node)  
  
 # Правое-правое  
 if balance < -1 and self.\_get\_balance(node.right) <= 0:  
 return self.\_left\_rotate(node)  
  
 # Левое-правое  
 if balance > 1 and self.\_get\_balance(node.left) < 0:  
 node.left = self.\_left\_rotate(node.left)  
 return self.\_right\_rotate(node)  
  
 # Правое-левое  
 if balance < -1 and self.\_get\_balance(node.right) > 0:  
 node.right = self.\_right\_rotate(node.right)  
 return self.\_left\_rotate(node)  
  
 return node  
  
 # Левый поворот для балансировки  
 def \_left\_rotate(self, z):  
 if z is None or z.right is None:  
 return z  
  
 y = z.right  
 T2 = y.left  
  
 # Выполняем поворот  
 y.left = z  
 z.right = T2  
  
 # Обновляем высоты  
 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 # Новый корень поддерева  
  
 # Правый поворот для балансировки  
 def \_right\_rotate(self, z):  
 if z is None or z.left is None:  
 return z  
  
 y = z.left  
 T3 = y.right  
  
 # Выполняем поворот  
 y.right = z  
 z.left = T3  
  
 # Обновляем высоты  
 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 # Новый корень поддерева  
  
 # Получение высоты узла  
 def \_get\_height(self, node):  
 if not node:  
 return 0  
 return node.height  
  
 # Расчет баланс-фактора (разница высот поддеревьев)  
 def \_get\_balance(self, node):  
 if not node:  
 return 0  
 return self.\_get\_height(node.left) - self.\_get\_height(node.right)  
  
  
"""  
Функции для генерации тестовых данных:  
- generate\_random\_array - создает массив случайных чисел  
- generate\_sorted\_array - создает отсортированный массив  
"""  
  
  
def generate\_random\_array(size):  
 return [random.randint(0, 100000) for \_ in range(size)]  
  
  
def generate\_sorted\_array(size):  
 return [i for i in range(size)]  
  
  
"""  
Функция тестирования производительности дерева:  
1. Вставка всех элементов массива  
2. 1000 операций поиска случайных элементов  
3. 1000 операций удаления/вставки (чтобы размер дерева не менялся)  
Возвращает среднее время для каждой операции.  
"""  
  
  
def test\_tree\_performance(tree\_class, array):  
 # Тест вставки  
 insert\_time = time.time()  
 tree = tree\_class()  
 for key in array:  
 tree.insert(key)  
 insert\_time = time.time() - insert\_time  
  
 # Тест поиска (1000 операций)  
 search\_time = time.time()  
 for \_ in range(1000):  
 key = random.choice(array) # Выбираем случайный ключ из массива  
 tree.search(key)  
 search\_time = (time.time() - search\_time) / 1000 # Среднее время поиска  
  
 # Тест удаления (1000 операций)  
 delete\_time = time.time()  
 for \_ in range(1000):  
 key = random.choice(array)  
 tree.delete(key)  
 tree.insert(key) # Восстанавливаем размер дерева  
 delete\_time = (time.time() - delete\_time) / 1000 # Среднее время удаления  
  
 return insert\_time, search\_time, delete\_time  
  
  
"""  
Функция тестирования поиска в массиве:  
Выполняет 1000 операций поиска и возвращает среднее время.  
"""  
  
  
def test\_array\_performance(array):  
 search\_time = time.time()  
 for \_ in range(1000):  
 key = random.choice(array)  
 key in array # Оператор in для поиска в списке  
 search\_time = (time.time() - search\_time) / 1000  
 return search\_time  
  
  
"""  
Основная функция тестирования:  
1. Выполняет серии тестов для разных размеров массивов  
2. Для каждого размера проводит 10 тестов со случайными данными   
 и 10 тестов с отсортированными данными  
3. Сохраняет средние значения времени для каждой операции  
"""  
  
  
def run\_test\_series():  
 # Структура для хранения результатов  
 results = {  
 'bst\_random': {'insert': [], 'search': [], 'delete': []},  
 'avl\_random': {'insert': [], 'search': [], 'delete': []},  
 'bst\_sorted': {'insert': [], 'search': [], 'delete': []},  
 'avl\_sorted': {'insert': [], 'search': [], 'delete': []},  
 'array\_random': {'search': []},  
 'array\_sorted': {'search': []}  
 }  
  
 # 5 серий тестов для размеров 2^11 до 2^15  
 for i in range(1, 6):  
 size = 2 \*\* (10 + i)  
 print(f"\nСерия {i}: Размер массива = {size}")  
  
 # Списки для хранения временных результатов  
 bst\_random\_insert, bst\_random\_search, bst\_random\_delete = [], [], []  
 avl\_random\_insert, avl\_random\_search, avl\_random\_delete = [], [], []  
 array\_random\_search = []  
  
 bst\_sorted\_insert, bst\_sorted\_search, bst\_sorted\_delete = [], [], []  
 avl\_sorted\_insert, avl\_sorted\_search, avl\_sorted\_delete = [], [], []  
 array\_sorted\_search = []  
  
 # 10 тестов со случайными данными  
 print(" Тестируем случайные данные...")  
 for \_ in range(10):  
 random\_array = generate\_random\_array(size)  
  
 # Тестируем BST  
 ins, srch, dlt = test\_tree\_performance(BST, random\_array)  
 bst\_random\_insert.append(ins)  
 bst\_random\_search.append(srch)  
 bst\_random\_delete.append(dlt)  
  
 # Тестируем AVL  
 ins, srch, dlt = test\_tree\_performance(AVL, random\_array)  
 avl\_random\_insert.append(ins)  
 avl\_random\_search.append(srch)  
 avl\_random\_delete.append(dlt)  
  
 # Тестируем массив  
 array\_random\_search.append(test\_array\_performance(random\_array))  
  
 # 10 тестов с отсортированными данными  
 print(" Тестируем отсортированные данные...")  
 for \_ in range(10):  
 sorted\_array = generate\_sorted\_array(size)  
  
 # Тестируем BST  
 ins, srch, dlt = test\_tree\_performance(BST, sorted\_array)  
 bst\_sorted\_insert.append(ins)  
 bst\_sorted\_search.append(srch)  
 bst\_sorted\_delete.append(dlt)  
  
 # Тестируем AVL  
 ins, srch, dlt = test\_tree\_performance(AVL, sorted\_array)  
 avl\_sorted\_insert.append(ins)  
 avl\_sorted\_search.append(srch)  
 avl\_sorted\_delete.append(dlt)  
  
 # Тестируем массив  
 array\_sorted\_search.append(test\_array\_performance(sorted\_array))  
  
 # Вычисляем средние значения и сохраняем результаты  
 def avg(lst):  
 return sum(lst) / len(lst)  
  
 # Сохраняем результаты для случайных данных  
 results['bst\_random']['insert'].append(avg(bst\_random\_insert))  
 results['bst\_random']['search'].append(avg(bst\_random\_search))  
 results['bst\_random']['delete'].append(avg(bst\_random\_delete))  
  
 results['avl\_random']['insert'].append(avg(avl\_random\_insert))  
 results['avl\_random']['search'].append(avg(avl\_random\_search))  
 results['avl\_random']['delete'].append(avg(avl\_random\_delete))  
  
 results['array\_random']['search'].append(avg(array\_random\_search))  
  
 # Сохраняем результаты для отсортированных данных  
 results['bst\_sorted']['insert'].append(avg(bst\_sorted\_insert))  
 results['bst\_sorted']['search'].append(avg(bst\_sorted\_search))  
 results['bst\_sorted']['delete'].append(avg(bst\_sorted\_delete))  
  
 results['avl\_sorted']['insert'].append(avg(avl\_sorted\_insert))  
 results['avl\_sorted']['search'].append(avg(avl\_sorted\_search))  
 results['avl\_sorted']['delete'].append(avg(avl\_sorted\_delete))  
  
 results['array\_sorted']['search'].append(avg(array\_sorted\_search))  
  
 # Выводим результаты текущей серии  
 print(  
 f" BST случайные: вставка={avg(bst\_random\_insert):.6f}, поиск={avg(bst\_random\_search):.6f}, удаление={avg(bst\_random\_delete):.6f}")  
 print(  
 f" AVL случайные: вставка={avg(avl\_random\_insert):.6f}, поиск={avg(avl\_random\_search):.6f}, удаление={avg(avl\_random\_delete):.6f}")  
 print(f" Массив случайные: поиск={avg(array\_random\_search):.6f}")  
  
 print(  
 f" BST отсорт.: вставка={avg(bst\_sorted\_insert):.6f}, поиск={avg(bst\_sorted\_search):.6f}, удаление={avg(bst\_sorted\_delete):.6f}")  
 print(  
 f" AVL отсорт.: вставка={avg(avl\_sorted\_insert):.6f}, поиск={avg(avl\_sorted\_search):.6f}, удаление={avg(avl\_sorted\_delete):.6f}")  
 print(f" Массив отсорт.: поиск={avg(array\_sorted\_search):.6f}")  
  
 return results  
  
  
"""  
Функция построения графиков:  
Создает 6 графиков для визуализации результатов тестирования:  
1. Вставка (случайные данные)  
2. Поиск (случайные данные) с сравнением с массивом  
3. Удаление (случайные данные)  
4. Вставка (отсортированные данные)  
5. Поиск (отсортированные данные) с сравнением с массивом  
6. Удаление (отсортированные данные)  
"""  
  
  
def plot\_results(results):  
 x = [2 \*\* (10 + i) for i in range(1, 6)] # Размеры массивов  
  
 plt.figure(figsize=(18, 12)) # Создаем большое окно для графиков  
  
 # График 1: Вставка (случайные данные)  
 plt.subplot(2, 3, 1)  
 plt.plot(x, results['bst\_random']['insert'], 'o-', label='BST')  
 plt.plot(x, results['avl\_random']['insert'], 'o-', label='AVL')  
 plt.xlabel('Размер массива')  
 plt.ylabel('Время (сек)')  
 plt.title('Вставка (случайные данные)')  
 plt.legend()  
 plt.grid(True)  
  
 # График 2: Поиск (случайные данные)  
 plt.subplot(2, 3, 2)  
 plt.plot(x, results['bst\_random']['search'], 'o-', label='BST')  
 plt.plot(x, results['avl\_random']['search'], 'o-', label='AVL')  
 plt.plot(x, results['array\_random']['search'], 'o-', label='Массив')  
 plt.xlabel('Размер массива')  
 plt.ylabel('Время (сек)')  
 plt.title('Поиск (случайные данные)')  
 plt.legend()  
 plt.grid(True)  
  
 # График 3: Удаление (случайные данные)  
 plt.subplot(2, 3, 3)  
 plt.plot(x, results['bst\_random']['delete'], 'o-', label='BST')  
 plt.plot(x, results['avl\_random']['delete'], 'o-', label='AVL')  
 plt.xlabel('Размер массива')  
 plt.ylabel('Время (сек)')  
 plt.title('Удаление (случайные данные)')  
 plt.legend()  
 plt.grid(True)  
  
 # График 4: Вставка (отсортированные данные)  
 plt.subplot(2, 3, 4)  
 plt.plot(x, results['bst\_sorted']['insert'], 'o-', label='BST')  
 plt.plot(x, results['avl\_sorted']['insert'], 'o-', label='AVL')  
 plt.xlabel('Размер массива')  
 plt.ylabel('Время (сек)')  
 plt.title('Вставка (отсортированные данные)')  
 plt.legend()  
 plt.grid(True)  
  
 # График 5: Поиск (отсортированные данные)  
 plt.subplot(2, 3, 5)  
 plt.plot(x, results['bst\_sorted']['search'], 'o-', label='BST')  
 plt.plot(x, results['avl\_sorted']['search'], 'o-', label='AVL')  
 plt.plot(x, results['array\_sorted']['search'], 'o-', label='Массив')  
 plt.xlabel('Размер массива')  
 plt.ylabel('Время (сек)')  
 plt.title('Поиск (отсортированные данные)')  
 plt.legend()  
 plt.grid(True)  
  
 # График 6: Удаление (отсортированные данные)  
 plt.subplot(2, 3, 6)  
 plt.plot(x, results['bst\_sorted']['delete'], 'o-', label='BST')  
 plt.plot(x, results['avl\_sorted']['delete'], 'o-', label='AVL')  
 plt.xlabel('Размер массива')  
 plt.ylabel('Время (сек)')  
 plt.title('Удаление (отсортированные данные)')  
 plt.legend()  
 plt.grid(True)  
  
 plt.tight\_layout() # Автоматическая настройка отступов  
 plt.show() # Показываем графики  
  
  
"""  
1. Устанавливаем увеличенный лимит рекурсии для работы с большими деревьями  
2. Запускаем тестирование  
3. Строим графики результатов  
"""  
  
sys.setrecursionlimit(1000000) # Увеличиваем лимит рекурсии для больших деревьев  
print("Начало тестирования...")  
results = run\_test\_series() # Запускаем тесты  
plot\_results(results) # Строим графики  
print("Тестирование завершено успешно!")

***Результаты:***

Начало тестирования...

Серия 1: Размер массива = 2048

Тестируем случайные данные...

Тестируем отсортированные данные...

BST случайные: вставка=0.002801, поиск=0.000002, удаление=0.000003

AVL случайные: вставка=0.015611, поиск=0.000002, удаление=0.000017

Массив случайные: поиск=0.000010

BST отсорт.: вставка=0.128356, поиск=0.000215, удаление=0.000244

AVL отсорт.: вставка=0.015391, поиск=0.000002, удаление=0.000017

Массив отсорт.: поиск=0.000010

Серия 2: Размер массива = 4096

Тестируем случайные данные...

Тестируем отсортированные данные...

BST случайные: вставка=0.005113, поиск=0.000002, удаление=0.000004

AVL случайные: вставка=0.033471, поиск=0.000002, удаление=0.000019

Массив случайные: поиск=0.000019

BST отсорт.: вставка=0.502899, поиск=0.000446, удаление=0.000552

AVL отсорт.: вставка=0.032583, поиск=0.000002, удаление=0.000018

Массив отсорт.: поиск=0.000019

Серия 3: Размер массива = 8192

Тестируем случайные данные...

Тестируем отсортированные данные...

BST случайные: вставка=0.012018, поиск=0.000002, удаление=0.000004

AVL случайные: вставка=0.070932, поиск=0.000002, удаление=0.000020

Массив случайные: поиск=0.000037

BST отсорт.: вставка=2.061859, поиск=0.000898, удаление=0.001199

AVL отсорт.: вставка=0.071654, поиск=0.000002, удаление=0.000020

Массив отсорт.: поиск=0.000039

Серия 4: Размер массива = 16384

Тестируем случайные данные...

Тестируем отсортированные данные...

BST случайные: вставка=0.026505, поиск=0.000003, удаление=0.000004

AVL случайные: вставка=0.154957, поиск=0.000003, удаление=0.000022

Массив случайные: поиск=0.000073

BST отсорт.: вставка=8.405071, поиск=0.001826, удаление=0.002483

AVL отсорт.: вставка=0.153566, поиск=0.000002, удаление=0.000020

Массив отсорт.: поиск=0.000076

Серия 5: Размер массива = 32768

Тестируем случайные данные...

Тестируем отсортированные данные...

BST случайные: вставка=0.056783, поиск=0.000003, удаление=0.000005

AVL случайные: вставка=0.339702, поиск=0.000002, удаление=0.000023

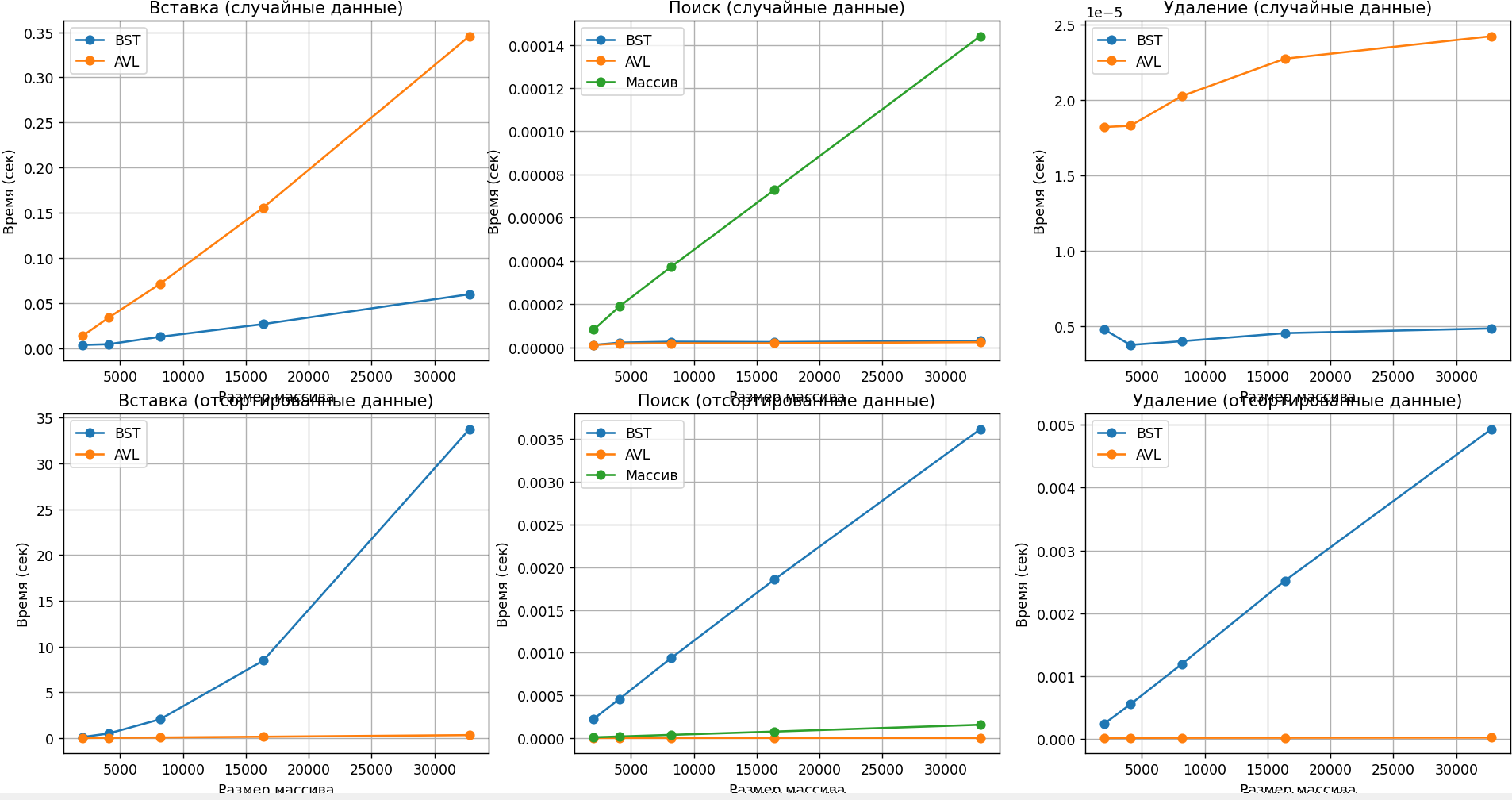
Массив случайные: поиск=0.000137

BST отсорт.: вставка=33.343508, поиск=0.003482, удаление=0.004707

AVL отсорт.: вставка=0.328180, поиск=0.000003, удаление=0.000023

Массив отсорт.: поиск=0.000155

Тестирование завершено успешно!



# Заключение.

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