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-33 ..... Тернолуцкий Виктор Александрович
Ссылка на репозиторий: ..... https://github.com/MUCTR-IKT-CPP/VATernolutski_36

Приняли:

..... Крашенинников Роман Сергеевич

Дата сдачи: 02.04.2025

Оглавление

Описание задачи.

В рамках лабораторной работы необходимо изучить и реализовать бинарное дерево поиска и его самобалансирующийся вариант в лице AVL дерева.

Для проверки анализа работы структуры данных требуется провести 10 серий тестов.

- В каждой серии тестов требуется выполнять 20 циклов генерации и операций. При этом первые 10 работают с массивом заполненным случайным образом, во второй половине случаев, массив заполняется в порядке возрастания значений индекса, т.е. является отсортированным по умолчанию.
- Требуется создать массив состоящий из 2^(10 + i) элементов, где i это номер серии.
- Массив должен быть помещен в оба варианта двоичных деревьев. При этому замеряется время затраченное на всю операцию вставки всего массива.
- После заполнения массива, требуется выполнить 1000 операций поиска по обоим вариантам дерева, случайного числа в диапазоне генерируемых значений, замерев время на все 1000 попыток и вычислив время 1 операции поиска.
- Провести 1000 операций поиска по массиву, замерить требуемое время на все 1000 операций и найти время на 1 операцию.
- После, требуется выполнить 1000 операций удаления значений из двоичных деревьев, и замерить время затраченное на все операции, после чего вычислить время на 1 операцию.
- После выполнения всех серий тестов, требуется построить графики зависимости времени затрачиваемого на операции вставки, поиска, удаления от количества элементов. При этом требуется разделить графики для отсортированного набора данных и заполненных со случайным распределением. Так же, для операции поиска, требуется так же нанести для сравнения график времени поиска для обычного массива.

Описание метода/модели.

Бинарное дерево — это иерархическая структура данных, в которой каждый узел имеет значение (оно же является в данном случае и ключом) и ссылки на левого и правого потомка. Узел, находящийся на самом верхнем уровне (не являющийся чьим либо потомком) называется корнем. Узлы, не имеющие потомков (оба потомка которых равны NULL) называются листьями.

Бинарное дерево поиска — это бинарное дерево, обладающее дополнительными свойствами: значение левого потомка меньше значения родителя, а значение правого потомка больше значения родителя для каждого узла дерева. То есть, данные в бинарном дереве поиска хранятся в отсортированном виде. При каждой операции вставки нового или удаления существующего узла отсортированный порядок дерева сохраняется. При поиске элемента сравнивается искомое значение с корнем. Если искомое больше корня, то поиск продолжается в правом потомке корня, если меньше, то в левом, если равно, то значение найдено и поиск прекращается.

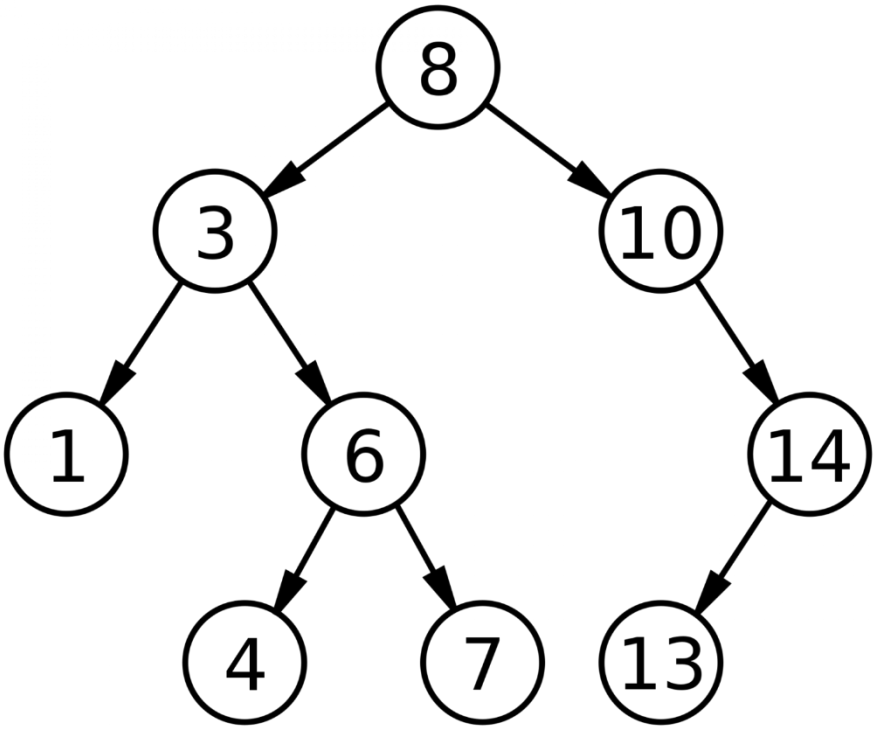

Рисунок 1 – Бинарное дерево поиска

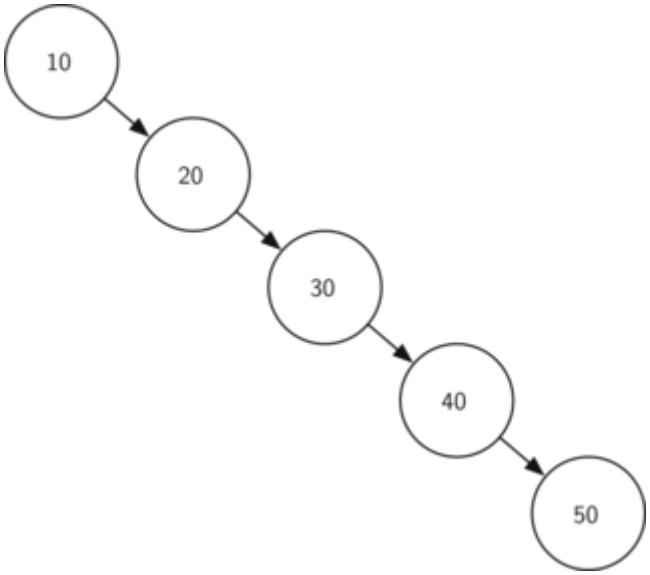

Рисунок 2 - Экстремально несбалансированное бинарное дерево поиска

АВЛ-дерево — это прежде всего двоичное дерево поиска, ключи которого удовлетворяют стандартному свойству: ключ любого узла дерева не меньше любого ключа в левом поддереве данного узла и не больше любого ключа в правом поддереве этого узла. Это значит, что для поиска нужного ключа в АВЛ-дереве можно использовать стандартный алгоритм. Для простоты дальнейшего изложения будем считать, что все ключи в дереве целочисленны и не повторяются.

Особенностью АВЛ-дерева является то, что оно является сбалансированным в следующем смысле: *для любого узла дерева высота его правого поддерева отличается от высоты левого поддерева не более чем на единицу*.

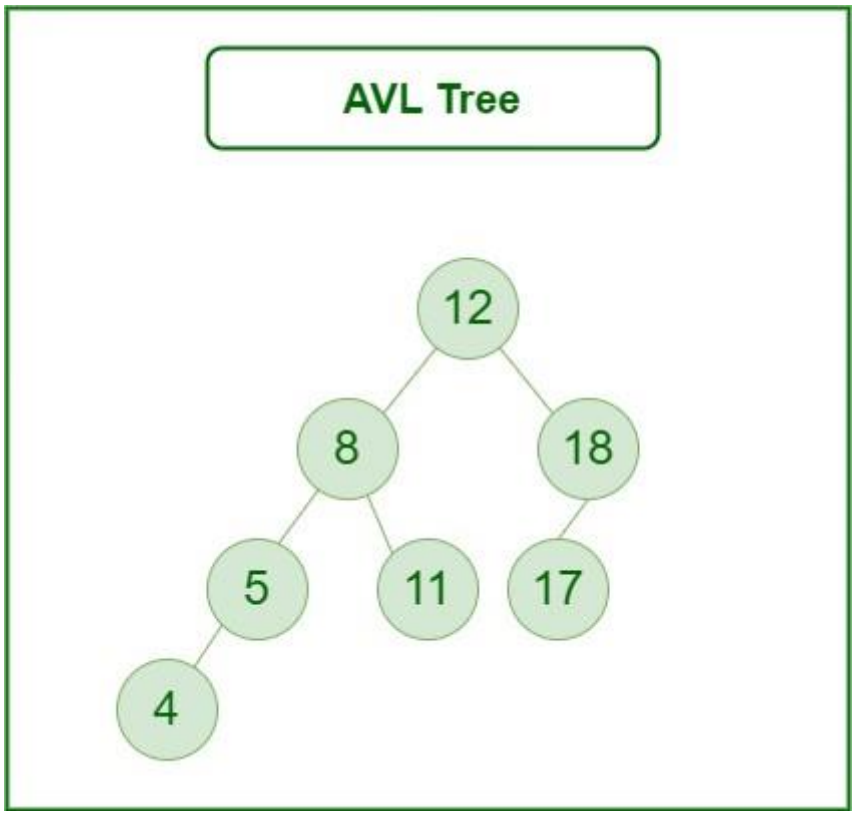

Рисунок 3 – AVL дерево

Выполнение задачи.

```
import random
import time
import matplotlib.pyplot as plt

class Node:
def __init__(self, key):
self.key = key
self.left = None
self.right = None
self.height = 1 # Для AVL дерева

class BST:
def __init__(self):
self.root = None

def insert(self, key):
if self.root is None:
self.root = Node(key)
return
current = self.root
while True:
if key < current.key:
if current.left is None:
current.left = Node(key)
return
current = current.left
elif key > current.key:
if current.right is None:
current.right = Node(key)
return
current = current.right
else:
return # Дубликаты не вставляем

def search(self, key):
current = self.root
while current is not None:
if key == current.key:
return current
elif key < current.key:
current = current.left
else:
current = current.right
return None
```

```
def delete(self, key):
parent = None
current = self.root
# Поиск удаляемого узла
while current is not None and current.key != key:
parent = current
if key < current.key:
current = current.left
else:
current = current.right
if current is None:
return # Узел не найден
# Случай 1: У узла нет детей или только один ребенок
if current.left is None or current.right is None:
new_child = current.left if current.left else current.right
if parent is None:
self.root = new_child
else:
if parent.left == current:
parent.left = new_child
else:
parent.right = new_child
else:
# Случай 2: У узла два ребенка
successor_parent = current
successor = current.right
while successor.left is not None:
successor_parent = successor
successor = successor.left
current.key = successor.key
if successor_parent.left == successor:
successor_parent.left = successor.right
else:
successor_parent.right = successor.right

class AVLTree:
def __init__(self):
self.root = None

def insert(self, key):
self.root = self._insert(self.root, key)

def _insert(self, node, key):
if node is None:
return Node(key)
if key < node.key:
node.left = self._insert(node.left, key)
elif key > node.key:
node.right = self._insert(node.right, key)
else:
return node # Дубликаты не вставляем
```

```
# Обновление высоты
node.height = 1 + max(self._get_height(node.left),
self._get_height(node.right))
# Балансировка
balance = self._get_balance(node)
# Left Left
if balance > 1 and key < node.left.key:
return self._right_rotate(node)
# Right Right
if balance < -1 and key > node.right.key:
return self._left_rotate(node)
# Left Right
if balance > 1 and key > node.left.key:
node.left = self._left_rotate(node.left)
return self._right_rotate(node)
# Right Left
if balance < -1 and key < node.right.key:
node.right = self._right_rotate(node.right)
return self._left_rotate(node)
return node

def search(self, key):
current = self.root
while current is not None:
if key == current.key:
return current
elif key < current.key:
current = current.left
else:
current = current.right
return None

def delete(self, key):
self.root = self._delete(self.root, key)

def _delete(self, node, key):
if node is None:
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:
if node.left is None:
return node.right
elif node.right is None:
return node.left
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
# Обновление высоты
node.height = 1 + max(self._get_height(node.left),
self._get_height(node.right))
# Балансировка
balance = self._get_balance(node)
# Left Left
if balance > 1 and self._get_balance(node.left) >= 0:
return self._right_rotate(node)
# Left Right
if balance > 1 and self._get_balance(node.left) < 0:
node.left = self._left_rotate(node.left)
return self._right_rotate(node)
# Right Right
if balance < -1 and self._get_balance(node.right) <= 0:
return self._left_rotate(node)
# Right Left
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):
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, y):
x = y.left
T2 = x.right
x.right = y
y.left = T2
y.height = 1 + max(self._get_height(y.left),
self._get_height(y.right))
x.height = 1 + max(self._get_height(x.left),
self._get_height(x.right))
return x

def _get_height(self, node):
if node is None:
return 0
return node.height

def _get_balance(self, node):
if node is None:
```

```
return 0
return self._get_height(node.left) - self._get_height(node.right)

def _min_value_node(self, node):
current = node
while current.left is not None:
current = current.left
return current

def run_tests():
repeats = 2
sizes = [2**(8+i) for i in range(12)]
results = {
'insert': {'BST': [], 'AVL': [], 'Array': []},
'search': {'BST': [], 'AVL': [], 'Array': []},
'delete': {'BST': [], 'AVL': [], 'Array': []}
}

for size in sizes:
print(f"\nTesting size: {size}")
# Инициализация результатов для текущего размера
for op in results:
for ds in results[op]:
results[op][ds].append(0)
for r in range(repeats):
print(f" Repeat {r+1}/{repeats}")
# Генерация данных (первые 5 тестов - отсортированные, последние - случайные)
if sizes.index(size) < 3: # Первые 3 размера - отсортированные
data = list(range(size))
else:
data = [random.randint(0, size*10) for _ in range(size)]
random.shuffle(data) # Для BST важно не передавать отсортированные данные
# Тестирование вставки
for ds in ['BST', 'AVL', 'Array']:
start = time.time()
if ds == 'BST':
tree = BST()
for num in data:
tree.insert(num)
elif ds == 'AVL':
tree = AVLTree()
for num in data:
tree.insert(num)
else: # Array
arr = []
for num in data:
arr.append(num)
results['insert'][ds][-1] += (time.time() - start)/repeats
# Подготовка структур для тестов поиска/удаления
bst = BST()
avl = AVLTree()
```

```
arr = []
for num in data:
bst.insert(num)
avl.insert(num)
arr.append(num)
# Тестирование поиска (1000 операций)
search_keys = random.sample(data, min(1000, size))
for ds in ['BST', 'AVL', 'Array']:
start = time.time()
if ds == 'BST':
for key in search_keys:
bst.search(key)
elif ds == 'AVL':
for key in search_keys:
avl.search(key)
else: # Array
for key in search_keys:
key in arr
total_time = time.time() - start
results['search'][ds][-1] += (total_time/len(search_keys))/repeats
# Тестирование удаления (1000 операций)
delete_keys = random.sample(data, min(1000, size))
for ds in ['BST', 'AVL', 'Array']:
start = time.time()
if ds == 'BST':
for key in delete_keys:
bst.delete(key)
elif ds == 'AVL':
for key in delete_keys:
avl.delete(key)
else: # Array
arr_copy = arr.copy()
for key in delete_keys:
if key in arr_copy:
arr_copy.remove(key)
total_time = time.time() - start
results['delete'][ds][-1] += (total_time/len(delete_keys))/repeats

# Построение графиков
operations = ['insert', 'search', 'delete']
for op in operations:
plt.figure(figsize=(10, 6))
for ds in ['BST', 'AVL', 'Array']:
plt.plot(sizes, results[op][ds], label=ds, marker='o')
plt.xscale('log')
plt.yscale('log')
plt.xlabel('Number of elements')
plt.ylabel('Time per operation (s)')
plt.title(f'{op.capitalize()} performance comparison')
plt.legend()
plt.grid(True)
plt.savefig(f'{op}_performance.png')
```

```
plt.close()

print("\nTesting completed!")
print("Graphs saved as: insert_performance.png, search_performance.png, delete_performance.png")

if __name__ == '__main__':
run_tests()
```

Результаты расчётов(Неотсортированный массив):

Testing size: 256

Repeat 1/2

Repeat 2/2

Testing size: 512

Repeat 1/2

Repeat 2/2

Testing size: 1024

Repeat 1/2

Repeat 2/2

Testing size: 2048

Repeat 1/2

Repeat 2/2

Testing size: 4096

Repeat 1/2

Repeat 2/2

Testing size: 8192

Repeat 1/2

Repeat 2/2

Testing size: 16384

Repeat 1/2

Repeat 2/2

Testing size: 32768

Repeat 1/2

Repeat 2/2

Testing size: 65536

Repeat 1/2

Repeat 2/2

Testing size: 131072

Repeat 1/2

Repeat 2/2

Testing size: 262144

Repeat 1/2

Repeat 2/2

Testing size: 524288

Repeat 1/2

Repeat 2/2

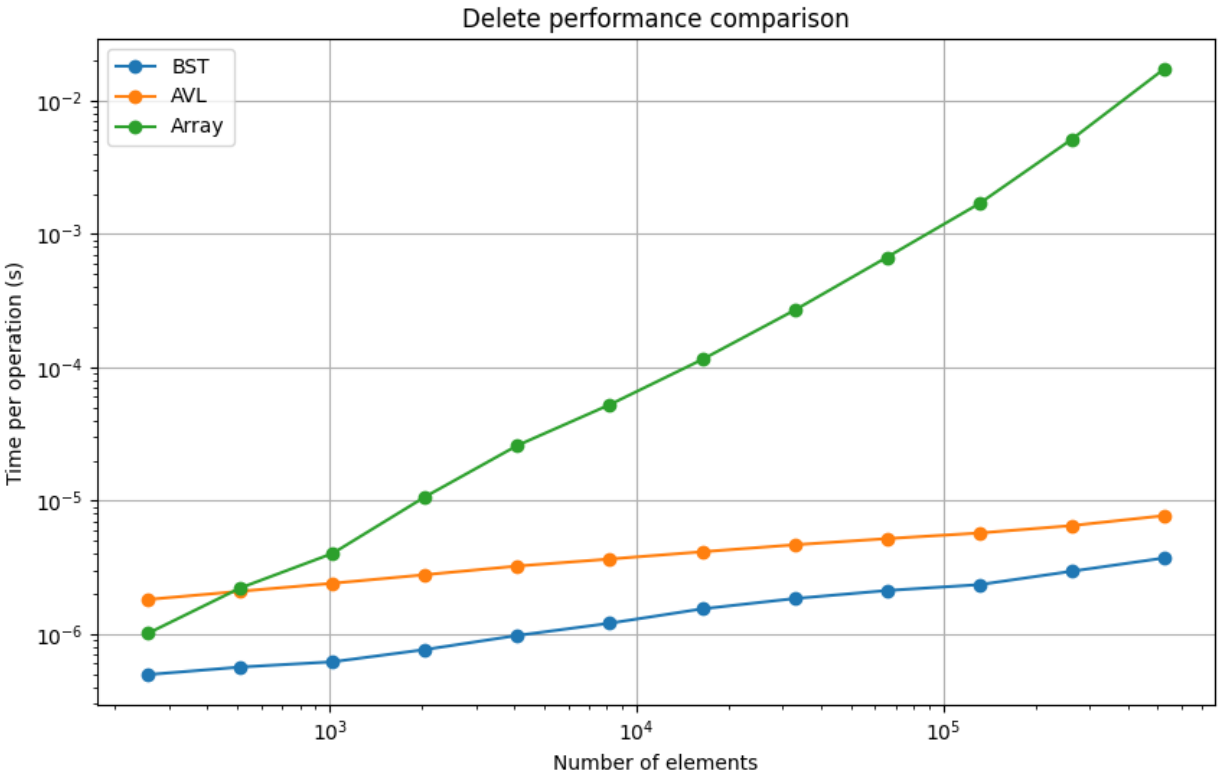

Рисунок 4 – Зависимость времени удаления от количества элементов

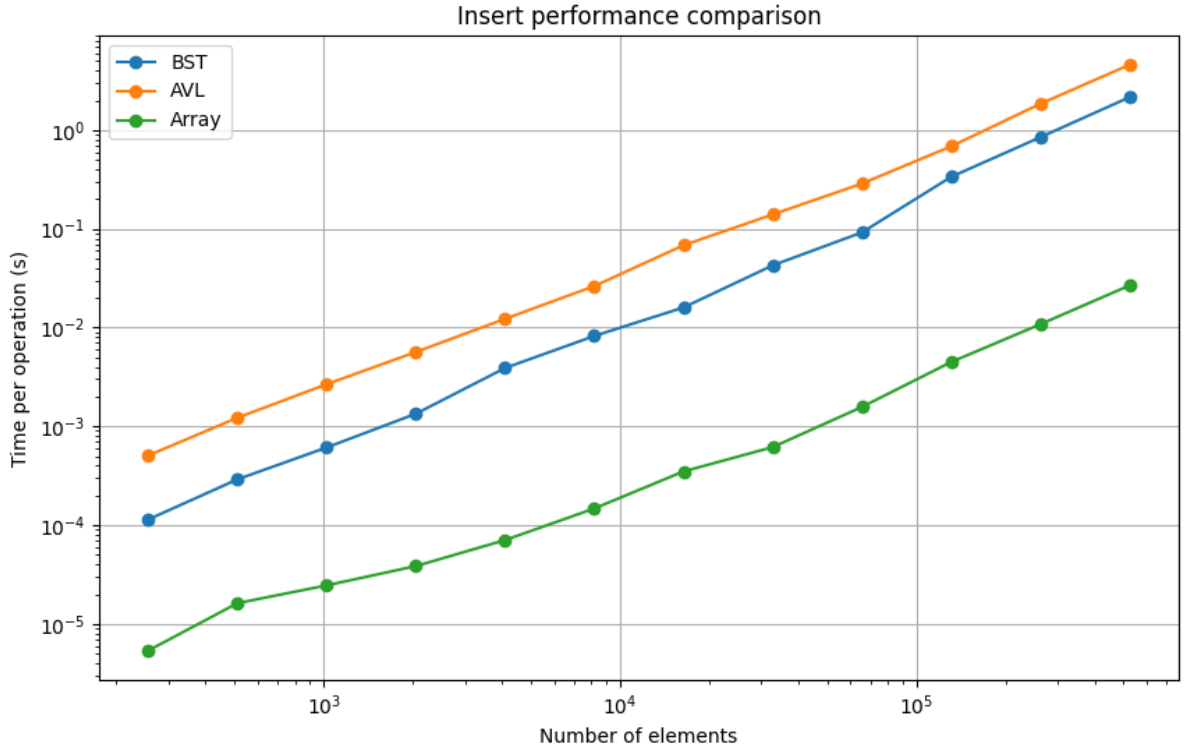

Рисунок 5 – Зависимость времени вставки от количества элементов

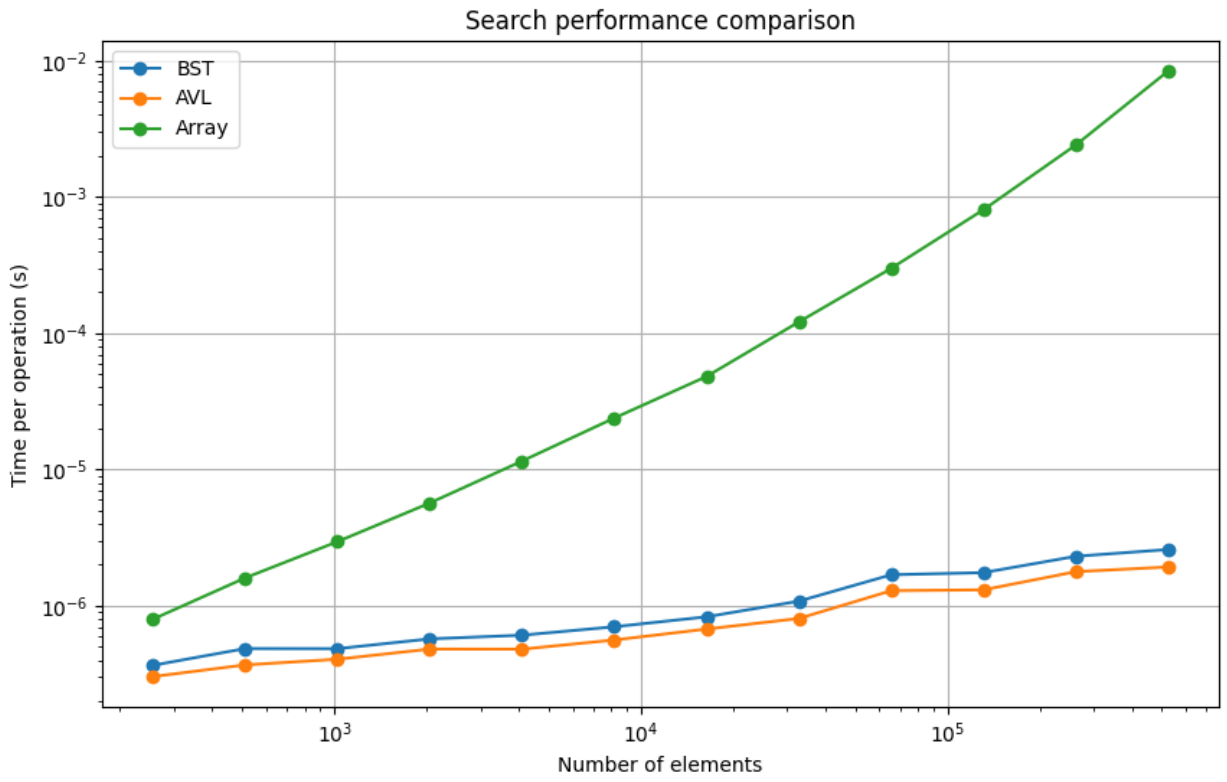

Рисунок 6 – Зависимость времени поиска от количества элементов

Заключение.

Были реализованы и протестированы две структуры данных: бинарное дерево поиска (BST) и самобалансирующееся AVL-дерево.

В сбалансированном состоянии BST обеспечивает логарифмическую сложность операций. Вставка: O(log n), поиск: O(log n), удаление: O(log n)

Однако при вставке отсортированных данных BST вырождается в связный список, ухудшая производительность до O(n).

В таком случае поиск, вставка и удаление работают очень медленно, так как приходится обходить весь путь от корня до листа.

AVL-дерево автоматически балансируется, что гарантирует сложность операций O(log n) в любом случае.

При вставке и удалении AVL-дерево выполняет повороты для поддержания баланса, поэтому оно остается эффективным даже на отсортированных данных.

По сравнению с BST, AVL-дерево немного медленнее при вставке и удалении из-за дополнительных затрат на балансировку, но быстрее при поиске.

Если данные поступают в случайном порядке, BST может работать хорошо, но если данные отсортированы или близки к отсортированным, BST теряет свою эффективность. В таких случаях

использование AVL-дерева оправдано, так как оно поддерживает сбалансированную структуру и обеспечивает быструю работу всех операций.