САНКТ-ПЕТЕРБУРГСКИЙ НАЦИОНАЛЬНЫЙ

ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ ИНФОРМАЦИОННЫХ

ТЕХНОЛОГИЙ, МЕХАНИКИ И ОПТИКИ

ФАКУЛЬТЕТ ИНФОКОММУНИКАЦИОННЫХ ТЕХНОЛОГИЙ

Отчет по лабораторной работе № 5 по курсу «Алгоритмы и структуры данных»

**Деревья. Пирамида, пирамидальная сортировка. Очередь с приоритетами**

Вариант 23

Выполнил:

Хабиби Ясер

Группа К3140   
  
Проверил:   
Афанасьев А. В

Санкт-Петербург 2024 г.

# Введение

Лабораторная работа посвящена разбору следующих структур данных: деревья, пирамида или двоичная куча, очередь с приоритетами, а также еще одному виду сортировки за n log n: пирамидальной (heapsort).

Задачи по варианту: 2 и 7, также были взяты и решены задания 1, 4,

Задача 2: Высота дерева

В данной задаче требовалось реализовать структуру данных для произвольного дерева и вычислить его высоту. Высота дерева определяется как максимальная глубина узла, то есть максимальное расстояние от листа до корня. Для решения задачи использовался алгоритм обхода дерева (например, с помощью рекурсии), который позволяет вычислить высоту путем поиска самого глубокого узла. Задача дает возможность познакомиться с особенностями работы с деревьями и углубить понимание структуры данных)

# Код программы

def calculate\_tree\_height(n, parents):  
 *"""Вычисляет высоту дерева."""* children = [[] for \_ in range(n)]  
 root = -1  
  
 for child\_index, parent\_index in enumerate(parents):  
 if parent\_index == -1:  
 root = child\_index  
 else:  
 children[parent\_index].append(child\_index)  
  
 def compute\_height(node):  
 if not children[node]:  
 return 1  
 return 1 + max(compute\_height(child) for child in children[node])  
  
 return compute\_height(root)  
  
def process\_tree\_height():  
 *"""Функция для обработки данных дерева."""* data = file\_read()  
 n = int(data[0][0]) # Первое значение - количество узлов  
 parents = list(map(int, data[1])) # Вторая строка - массив родителей  
 height = calculate\_tree\_height(n, parents)  
 file\_write([height

# Тесты и анализ

Алгоритм был протестирован на различных сетах данных . Все результаты были проведены быстро и успешно.

def test\_simple\_tree(self):  
 # given  
 n = 5  
 parents = [-1, 0, 0, 1, 1]  
 expected = 3  
  
 # when  
 result, elapsed\_time, peak\_memory\_megabytes = measure\_performance(calculate\_tree\_height, n, parents)  
  
 # then  
 self.assertEqual(result, expected)  
 self.assertLessEqual(elapsed\_time, 2, "Execution time exceeded 2 seconds")  
 self.assertLessEqual(peak\_memory\_megabytes, 256, "Memory usage exceeded 256 MB")  
  
def test\_single\_node(self):  
 # given  
 n = 1  
 parents = [-1]  
 expected = 1  
  
 # when  
 result, elapsed\_time, peak\_memory\_megabytes = measure\_performance(calculate\_tree\_height, n, parents)  
  
 # then  
 self.assertEqual(result, expected)  
 self.assertLessEqual(elapsed\_time, 2, "Execution time exceeded 2 seconds")  
 self.assertLessEqual(peak\_memory\_megabytes, 256, "Memory usage exceeded 256 MB")  
  
def test\_linear\_tree(self):  
 # given  
 n = 4  
 parents = [-1, 0, 1, 2]  
 expected = 4  
  
 # when  
 result, elapsed\_time, peak\_memory\_megabytes = measure\_performance(calculate\_tree\_height, n, parents)  
  
 # then  
 self.assertEqual(result, expected)  
 self.assertLessEqual(elapsed\_time, 2, "Execution time exceeded 2 seconds")  
 self.assertLessEqual(peak\_memory\_megabytes, 256, "Memory usage exceeded 256 MB")  
  
def test\_balanced\_tree(self):  
 # given  
 n = 7  
 parents = [-1, 0, 0, 1, 1, 2, 2]  
 expected = 3  
  
 # when  
 result, elapsed\_time, peak\_memory\_megabytes = measure\_performance(calculate\_tree\_height, n, parents)  
  
 # then  
 self.assertEqual(result, expected)  
 self.assertLessEqual(elapsed\_time, 2, "Execution time exceeded 2 seconds")  
 self.assertLessEqual(peak\_memory\_megabytes, 256, "Memory usage exceeded 256 MB")

Задача 7: Снова Сортировка

В данной задаче требовалось реализовать алгоритм пирамидальной сортировки (heap sort) для сортировки массива в убывающем порядке. Алгоритм пирамидальной сортировки основан на структуре данных "куча", где на каждом шаге извлекается максимальный элемент и восстанавливается свойство кучи. Для проверки работы программы необходимо было создать несколько случайных массивов, удовлетворяющих указанным параметрам, и проверить корректность сортировки.

Код программы

def max\_heapify(arr, n, i):  
 *"""Итеративный Max-Heapify для обеспечения свойств кучи."""* largest = i  
 left = 2 \* i + 1  
 right = 2 \* i + 2  
  
 while True:  
 if left < n and arr[left] > arr[largest]:  
 largest = left  
 if right < n and arr[right] > arr[largest]:  
 largest = right  
 if largest != i:  
 arr[i], arr[largest] = arr[largest], arr[i]  
 i = largest  
 left = 2 \* i + 1  
 right = 2 \* i + 2  
 else:  
 break  
  
  
def heap\_sort(arr):  
 *"""Реализация пирамидальной сортировки."""* n = len(arr)  
  
 # Построение max-кучи  
 for i in range(n // 2 - 1, -1, -1):  
 max\_heapify(arr, n, i)  
  
 # Извлечение элементов из кучи  
 for i in range(n - 1, 0, -1):  
 arr[0], arr[i] = arr[i], arr[0]  
 max\_heapify(arr, i, 0)  
  
 # Возвращаем массив в порядке убывания  
 return arr[::-1]  
  
  
def process\_heap\_sort():  
 *"""Читает данные, сортирует массив и записывает результат."""* data = file\_read()  
 n = int(data[0][0])  
 arr = list(map(int, data[1]))  
 sorted\_arr = heap\_sort(arr)  
 file\_write([" ".join(map(str, sorted\_arr))])

# Тесты и анализ

Алгоритм был протестирован на различных сетах данных . Все результаты были проведены быстро и успешно.

def test\_sorted\_array(self):  
 # given  
 array = [5, 3, 8, 4, 2]  
 expected = [8, 5, 4, 3, 2]  
  
 # when  
 result, elapsed\_time, peak\_memory\_megabytes = measure\_performance(heap\_sort, array)  
  
 # then  
 self.assertEqual(result, expected)  
 self.assertLessEqual(elapsed\_time, 2, "Execution time exceeded 2 seconds")  
 self.assertLessEqual(peak\_memory\_megabytes, 256, "Memory usage exceeded 256 MB")  
  
def test\_large\_array(self):  
 # given  
 array = [i for i in range(1000, 0, -1)]  
 expected = sorted(array, reverse=True)  
  
 # when  
 result, elapsed\_time, peak\_memory\_megabytes = measure\_performance(heap\_sort, array)  
  
 # then  
 self.assertEqual(result, expected)  
 self.assertLessEqual(elapsed\_time, 2, "Execution time exceeded 2 seconds")  
 self.assertLessEqual(peak\_memory\_megabytes, 256, "Memory usage exceeded 256 MB")  
  
def test\_small\_array(self):  
 # given  
 array = [3, 1, 2]  
 expected = [3, 2, 1]  
  
 # when  
 result, elapsed\_time, peak\_memory\_megabytes = measure\_performance(heap\_sort, array)  
  
 # then  
 self.assertEqual(result, expected)  
 self.assertLessEqual(elapsed\_time, 2, "Execution time exceeded 2 seconds")  
 self.assertLessEqual(peak\_memory\_megabytes, 256, "Memory usage exceeded 256 MB")

Задача 1 Куча Ли ? :

Задача проверяет, является ли заданный массив корректной кучей (heap). В куче для каждого узла значение должно быть меньше либо равно значения его потомков (для мин-кучи) или больше либо равно (для макс-кучи).

# Код программы

def is\_heap(arr):  
 *"""Проверяет, является ли массив неубывающей пирамидой."""* n = len(arr)  
 for i in range(n // 2):  
 left = 2 \* i + 1  
 right = 2 \* i + 2  
 if left < n and arr[i] > arr[left]:  
 return "NO"  
 if right < n and arr[i] > arr[right]:  
 return "NO"  
 return "YES"  
  
  
def process\_is\_heap():  
 *"""Читает данные, проверяет массив и записывает результат."""* data = file\_read()  
 n = int(data[0][0])  
 arr = list(map(int, data[1]))  
 result = is\_heap(arr)  
 file\_write([result])

# Тесты и анализ

Алгоритм был протестирован на различных сетах данных . Во всех случаях алгоритм отработал быстро и безошибочно, что доказывает, что тот был написан верно.

def test\_heap\_valid(self):  
 # given  
 array = [1, 2, 3, 4, 5]  
 expected = "YES"  
  
 # when  
 result, elapsed\_time, peak\_memory\_megabytes = measure\_performance(is\_heap, array)  
  
 # then  
 self.assertEqual(result, expected)  
 self.assertLessEqual(elapsed\_time, 2, "Execution time exceeded 2 seconds")  
 self.assertLessEqual(peak\_memory\_megabytes, 256, "Memory usage exceeded 256 MB")  
  
def test\_heap\_invalid(self):  
 # given  
 array = [5, 1, 0, 1, 2, 0]  
 expected = "NO"  
  
 # when  
 result, elapsed\_time, peak\_memory\_megabytes = measure\_performance(is\_heap, array)  
  
 # then  
 self.assertEqual(result, expected)  
 self.assertLessEqual(elapsed\_time, 2, "Execution time exceeded 2 seconds")  
 self.assertLessEqual(peak\_memory\_megabytes, 256, "Memory usage exceeded 256 MB")  
  
def test\_heap\_edge\_case(self):  
 # given  
 array = [10]  
 expected = "YES"  
  
 # when  
 result, elapsed\_time, peak\_memory\_megabytes = measure\_performance(is\_heap, array)  
  
 # then  
 self.assertEqual(result, expected)  
 self.assertLessEqual(elapsed\_time, 2, "Execution time exceeded 2 seconds")  
 self.assertLessEqual(peak\_memory\_megabytes, 256, "Memory usage exceeded 256 MB")

Задача 4 Построение Пирамиды:

Задача посвящена построению пирамиды (heap) из неупорядоченного массива. Это важный шаг для алгоритмов, таких как пирамидальная сортировка (heap sort).

# Код программы

def build\_heap(data):  
 *"""Преобразует массив в min-heap и возвращает список перестановок."""* n = len(data)  
 swaps = []  
  
 def sift\_down(i):  
 min\_index = i  
 left = 2 \* i + 1  
 right = 2 \* i + 2  
  
 if left < n and data[left] < data[min\_index]:  
 min\_index = left  
 if right < n and data[right] < data[min\_index]:  
 min\_index = right  
  
 if i != min\_index:  
 swaps.append((i, min\_index))  
 data[i], data[min\_index] = data[min\_index], data[i]  
 sift\_down(min\_index)  
  
 for i in range(n // 2 - 1, -1, -1):  
 sift\_down(i)  
  
 return swaps  
  
  
def process\_build\_heap():  
 *"""Читает данные, строит пирамиду и записывает результат."""* data = file\_read()  
 n = int(data[0][0])  
 arr = list(map(int, data[1]))  
 swaps = build\_heap(arr)  
 output = [len(swaps)] + [f"{i} {j}" for i, j in swaps]  
 file\_write(output)

# Тесты и анализ

Алгоритм был протестирован на различных сетах данных . Во всех случаях алгоритм отработал быстро и безошибочно, что доказывает, что тот был написан верно.

def test\_example\_1(self):  
 # given  
 array = [5, 4, 3, 2, 1]  
 expected\_swaps = [(1, 4), (0, 1), (1, 3)]  
  
 # when  
 result, elapsed\_time, peak\_memory\_megabytes = measure\_performance(build\_heap, array)  
  
 # then  
 self.assertEqual(result, expected\_swaps)  
 self.assertLessEqual(elapsed\_time, 3, "Execution time exceeded 3 seconds")  
 self.assertLessEqual(peak\_memory\_megabytes, 512, "Memory usage exceeded 512 MB")  
  
def test\_example\_2(self):  
 # given  
 array = [1, 2, 3, 4, 5]  
 expected\_swaps = []  
  
 # when  
 result, elapsed\_time, peak\_memory\_megabytes = measure\_performance(build\_heap, array)  
  
 # then  
 self.assertEqual(result, expected\_swaps)  
 self.assertLessEqual(elapsed\_time, 3, "Execution time exceeded 3 seconds")  
 self.assertLessEqual(peak\_memory\_megabytes, 512, "Memory usage exceeded 512 MB")  
  
def test\_large\_input(self):  
 # given  
 array = list(range(1000, 0, -1))  
 # Проверка только на корректное выполнение, без конкретного результата  
  
 # when  
 result, elapsed\_time, peak\_memory\_megabytes = measure\_performance(build\_heap, array)  
  
 # then  
 self.assertLessEqual(elapsed\_time, 3, "Execution time exceeded 3 seconds")  
 self.assertLessEqual(peak\_memory\_megabytes, 512, "Memory usage exceeded 512 MB")

# Заключение

В процессе выполнения лабораторной работы, посвящённой быстрой сортировке и её различным вариантам, были разработаны и протестированы несколько алгоритмов сортировки и алгоритмов сортировки за линейное время. Каждый из методов, включая различные способы использования алгоритма быстрой сортировки и алгоритмов сортировки за линейное время, был подробно изучен, реализован на Python и проверен с помощью модуля unittest. Были проанализированы принципы их работы, выявлены преимущества и недостатки, а также выполнена оценка их временной сложности.

Выполнение этой лабораторной работы позволило углубить знания об основных алгоритмах сортировки и алгоритмов сортировки за линейное время, улучшить навыки программирования на Python, а также развить умение проводить модульное тестирование программ.

развить умение проводить модульное тестирование программ.