САНКТ-ПЕТЕРБУРГСКИЙ НАЦИОНАЛЬНЫЙ
ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ
ИНФОРМАЦИОННЫХ ТЕХНОЛОГИЙ, МЕХАНИКИ И ОПТИКИ
ФАКУЛЬТЕТ ИНФОКОММУНИКАЦИОННЫХ ТЕХНОЛОГИЙ

Отчет по лабораторной работе №5
по курсу «Алгоритмы и структуры данных»
Тема: Деревья. Пирамида, пирамидальная сортировка. Очередь с приоритетами.
Вариант 3

Выполнил:
Бай М.О.
К3141

Проверил:
Афанасьев А.В.

Санкт-Петербург
2024 г.

Содержание отчета

Оглавление

Задачи по варианту

Задача №1. Куча ли?

Текст задачи:

Структуру данных «куча», или, более конкретно, «неубывающая пирамида», можно реализовать на основе массива.

Для этого должно выполнятся основное свойство неубывающей пирамиды, которое заключается в том, что для каждого $1 \leq i \leq n$ выполняются условия:

1. если $2i \leq n$, то $a_i \leq a_{2i}$,
2. если $2i + 1 \leq n$, то $a_i \leq a_{2i+1}$.

Дан массив целых чисел. Определите, является ли он неубывающей пирамидой.

Код программы:

```
import sys
import os

sys.path.append(os.path.abspath(os.path.join(os.path.dirname(__file__), '../..')))
from Lab5.utils import read_input, write_output, decorate


def Heap(l: int, ln: int):
    n = ln
    for i in range(1, n//2 + 1):
        one = 2 * i
        two = 2 * i + 1

        if one <= n:
            if l[i - 1] > l[one - 1]:
                return "NO"
        if two <= n:
            if l[i - 1] > l[two - 1]:
                return "NO"
    return "YES"


def main():
    lf = read_input(task=1)
    l = lf[1].split()
    ln = int(lf[0])
    out = Heap(l, ln)
    write_output(1, out)
    print(out)
    print()

if __name__ == '__main__':
    decorate(task=1,task_name='Heap')
```

Описание работы программы:

Основные Компоненты:

Импорты:

Код импортирует модули sys, os, а также функции read_input, write_output и decorate из Lab5.utils.
Функция Heap(l: int, ln: int):
Принимает два аргумента: l — список элементов и ln — количество элементов в списке.
Проходит по каждому элементу массива (начиная с первого) и сравнивает его со своими дочерними элементами (если они существуют).
Если родительский элемент больше любого из своих дочерних, функция возвращает "NO", указывая, что массив не является кучей. Если все проверки проходят, возвращается "YES".
Функция main():
Считывает входные данные, где первый элемент — это количество элементов в массиве, а второй элемент — сам массив.
Вызывает функцию Heap для проверки кучи и записывает результат в выходной файл. Также выводит результат на экран.
Точка Входа:
Код проверяет, запускается ли он как основной модуль, и вызывает функцию decorate, которая может использоваться для логирования или управления задачами.

Тесты:

```
import unittest
from Lab5.task1.src.Heap import Heap


class TestHoap(unittest.TestCase):

    def test_should_non_decreasing_heap(self):
        self.assertEqual(Heap([1, 2, 3, 4, 5], 5), "YES")
        self.assertEqual(Heap([1, 2, 1, 3, 2], 5), "YES")
        self.assertEqual(Heap([1, 1, 1, 1], 4), "YES")

    def test_should_not_a_heap(self):
        self.assertEqual(Heap([1, 3, 2, 4, 5], 5), "YES")
        self.assertEqual(Heap([5, 3, 2, 4, 1], 5), "NO")
        self.assertEqual(Heap([3, 2, 5, 1], 4), "NO")

    def test_should_single_element(self):
        self.assertEqual(Heap([1], 1), "YES")
        self.assertEqual(Heap([100], 1), "YES")

if __name__ == '__main__':
    unittest.main()
```

Вывод по задаче:
Код предоставляет простой и эффективный способ проверки, является ли массив представлением минимальной кучи

Задача №2. Высота дерева.

Текст задачи:

В этой задаче ваша цель - привыкнуть к деревьям. Вам нужно будет прочитать описание дерева из входных данных, реализовать структуру данных, сохранить дерево и вычислить его высоту.

- Вам дается корневое дерево. Ваша задача - вычислить и вывести его высоту. Напомним, что высота (корневого) дерева - это максимальная глубина узла или максимальное расстояние от листа до корня. Вам дано произвольное дерево, не обязательно бинарное дерево.
- **Формат ввода или входного файла (input.txt).** Первая строка содержит число узлов n ($1 \leq n \leq 10^5$). Вторая строка содержит n целых чисел от -1 до $n-1$ – указание на родительский узел. Если i-ое значение равно -1, значит, что узел i - корневой, иначе это число является обозначением индекса родительского узла этого i-го узла ($0 \leq i \leq n-1$). **Индексы считать с 0.** Гарантируется, что дан только один корневой узел, и что входные данные представляют дерево.

Код программы:

```
import sys
import os
from Lab5.utils import read_input, write_output, decorate

sys.path.append(os.path.abspath(os.path.join(os.path.dirname(__file__), '../..')))


class TreeHeightCalculator:
    def __init__(self):
        input_file = read_input(task=2)
        self.n = int(input_file[0])
        self.parents = [int(i) for i in input_file[1].split(' ')]
        self.tree = [[] for _ in range(self.n)]
        self.root = -1

    def height(self, node):
        if not self.tree[node]:
            return 1
        max_h = 0
        for child in self.tree[node]:
            max_h = max(max_h, self.height(child))
        return max_h + 1

    def tree_height(self, n, parents):
        """
        Вычисляет высоту дерева.

        :param n: Количество узлов.
        :param parents: Список родителей для каждого узла (индекс = узел, значение = родитель).
        :return: Высота дерева.
        """
        self.tree = [[] for _ in range(n)]
        self.root = -1

        for i in range(n):
```

```
            if parents[i] == -1:
                self.root = i
            else:
                self.tree[parents[i]].append(i)
        return self.height(self.root)


def main():
    tree = TreeHeightCalculator()
    res = tree.tree_height(tree.n, tree.parents)
    write_output(2, res)
    print(res)
    print()


if __name__ == '__main__':
    decorate(task=2, task_name="tree_height")
```

Описание работы программы:

Основные Компоненты:

Импорты:

Код импортирует модули sys, os, а также функции read_input, write_output и decorate из Lab5.utils.

Класс TreeHeightCalculator:

Инициализация:

В конструкторе __init__ происходит считывание входных данных, где первый элемент — это количество узлов n, а второй элемент — список родителей. Создается пустой список self.tree для хранения дочерних узлов и переменная self.root для хранения индекса корня дерева.

Метод height(node):

Рекурсивный метод, который вычисляет высоту дерева, начиная с узла node. Если узел не имеет дочерних узлов (листья), высота для него равна 1. В противном случае метод рекурсивно вычисляет высоту для каждого дочернего узла и возвращает максимальную высоту плюс 1.

Метод tree_height(n, parents):

Метод для построения дерева на основе массива родителей и вычисления его высоты. Он создает структуру дерева, определяет корень и вызывает метод height для вычисления высоты.

Функция main():

Создает экземпляр класса TreeHeightCalculator, вызывает метод tree_height для вычисления высоты дерева и записывает результат в выходной файл. Также выводит результат на экран.

Точка Входа:

Код проверяет, запускается ли он как основной модуль, и вызывает функцию decorate, которая может использоваться для логирования или управления задачами.

Тесты:

```
import unittest
from Lab5.task2.src.tree_height import TreeHeightCalculator

class TestTreeHeightCalculator(unittest.TestCase):

    def setUp(self):
        # Подготовка тестовых данных
        self.calculator = TreeHeightCalculator()

    def test_tree_height_single_node(self):
        # Тест для дерева с одним узлом
        self.calculator.n = 1
        self.calculator.parents = [-1]
        expected_height = 1
        result = self.calculator.tree_height(self.calculator.n, self.calculator.parents)
        self.assertEqual(result, expected_height)

    def test_tree_height_two_nodes(self):
        # Тест для дерева с двумя узлами
        self.calculator.n = 2
        self.calculator.parents = [-1, 0]  # Узел 0 - корень, узел 1 - дочерний
        expected_height = 2
        result = self.calculator.tree_height(self.calculator.n, self.calculator.parents)
        self.assertEqual(result, expected_height)

    def test_tree_height_complex_tree(self):
        # Тест для более сложного дерева
        self.calculator.n = 5
        self.calculator.parents = [-1, 0, 0, 1, 1]  # Корень 0, узлы 1 и 2 - дочерние к 0, 3 и 4 - дочерние к 1
        expected_height = 3
        result = self.calculator.tree_height(self.calculator.n, self.calculator.parents)
        self.assertEqual(result, expected_height)


if __name__ == '__main__':
    unittest.main()
```

Вывод по задаче:

Реализация вычисления высоты дерева в данном коде является хорошей основой для работы с деревьями.

Задача №3. Обработка сетевых пакетов

Текст задачи:

В этой задаче вы реализуете программу для моделирования обработки сетевых пакетов.

- Вам дается серия входящих сетевых пакетов, и ваша задача - смоделировать их обработку. Пакеты приходят в определенном порядке. Для каждого номера пакета i вы знаете время, когда пакет прибыл A_i и время, необходимое процессору для его обработки P_i (в миллисекундах). Есть только один процессор, и он обрабатывает входящие пакеты в порядке их поступления. Если процессор начал обрабатывать какой-либо пакет, он не прерывается и не останавливается, пока не завершит обработку этого пакета, а обработка пакета i занимает ровно P_i миллисекунд.

  Компьютер, обрабатывающий пакеты, имеет сетевой буфер фиксированного размера S. Когда пакеты приходят, они сохраняются в буфере перед обработкой. Однако, если буфер заполнен, когда приходит пакет (есть S пакетов, которые прибыли до этого пакета, и компьютер не завершил обработку ни одного из них), он отбрасывается и не обрабатывается вообще. Если несколько пакетов поступают одновременно, они сначала все сохраняются в буфере (из-за этого некоторые из них могут быть отброшены - те, которые описаны позже во входных данных). Компьютер обрабатывает пакеты в порядке их поступления и начинает обработку следующего доступного пакета из буфера, как только заканчивает обработку предыдущего. Если в какой-то момент компьютер не занят и в буфере нет пакетов, компьютер просто ожидает прибытия следующего пакета. Обратите внимание, что пакет покидает буфер и освобождает пространство в буфере, как только компьютер заканчивает его обработку.

Код программы:

```
import sys
import os
from Lab5.utils import read_input, write_output, decorate

sys.path.append(os.path.abspath(os.path.join(os.path.dirname(__file__), '../..')))


class NetworkPacketsProcessor:
    def __init__(self):
        input_file = read_input(task=3)
        first_line = input_file[0].split()
        self.buffer_size = int(first_line[0])
        self.n = int(first_line[1])
        self.packets = [list(map(int, i.split())) for i in input_file[1:] if len(i.split()) == 2]
        self.buffer = []
        self.start_times = []

    def network_packets(self, buffer_size, packets, n):
        for packet in packets:
            if len(packet) != 2:
                print(f"Неверный пакет: {packet}")
                continue

            arrival_time, processing_time = packet

            while self.buffer and self.buffer[0] <= arrival_time:
```

```
            self.buffer.pop(0)

        if len(self.buffer) >= self.buffer_size:
            self.start_times.append('-1')
        else:
            start_time = arrival_time if not self.buffer else self.buffer[-1]
            self.start_times.append(str(start_time))
            self.buffer.append(start_time + processing_time)

    return self.start_times


def main():
    npp = NetworkPacketsProcessor()
    res = npp.network_packets(npp.buffer_size, npp.packets, npp.n)
    write_output(3, *res)
    [print(i) for i in res]
    print()


if __name__ == "__main__":
    decorate(task=3, task_name='network_packet')
```

Описание работы программы:

Описание Кода

Данный код реализует обработку сетевых пакетов с учетом размера буфера. Он принимает входные данные, содержащие размеры буфера и информацию о пакетах, и вычисляет время начала обработки каждого пакета, учитывая, что пакеты могут приходить в разное время и занимать разное время на обработку.

Основные Компоненты:

Импорты:

Код импортирует модули sys, os, а также функции read_input, write_output и decorate из Lab5.utils.

Класс NetworkPacketsProcessor:

Инициализация:

В конструкторе __init__ происходит считывание входных данных, где первая строка содержит размер буфера и количество пакетов. Затем считываются пакеты, каждый из которых представлен парой значений: время прихода и время обработки.

Создаются пустые списки self.buffer для хранения времени завершения обработки пакетов и self.start_times для хранения времени начала обработки каждого пакета.

Метод network_packets(buffer_size, packets, n):

Основной метод для обработки пакетов. Для каждого пакета:

Проверяется, корректно ли он задан (должен содержать два значения).

Удаляются из буфера все пакеты, которые уже обработаны к моменту прихода текущего пакета.
Если буфер заполнен, в start_times добавляется '-1', что означает, что пакет не может быть обработан. В противном случае вычисляется время начала обработки текущего пакета и добавляется в start_times, а также обновляется буфер с учетом времени обработки.
Функция main():
Создает экземпляр класса NetworkPacketsProcessor, вызывает метод network_packets для обработки пакетов и записывает результат в выходной файл. Также выводит результат на экран.
Точка Входа:
Код проверяет, запускается ли он как основной модуль, и вызывает функцию decorate, которая может использоваться для логирования или управления задачами.
Тесты:

```
from Lab5.task3.src.network_packet import *

import unittest

class TestNetworkPacketsProcessor(unittest.TestCase):

    def setUp(self):
        # Prepare test data for the NetworkPacketsProcessor
        self.processor = NetworkPacketsProcessor()
        # Override the input data for testing purposes
        self.processor.buffer_size = 2
        self.processor.n = 5
        self.processor.packets = [
            [0, 3],  # Packet 1: arrival time 0, processing time 3
            [1, 9],  # Packet 2: arrival time 1, processing time 9
            [2, 6],  # Packet 3: arrival time 2, processing time 6
            [3, 4],  # Packet 4: arrival time 3, processing time 4
            [4, 5],  # Packet 5: arrival time 4, processing time 5
        ]
        self.processor.buffer = []
        self.processor.start_times = []


    def test_buffer_limit(self):
        self.processor.buffer_size = 1  # Set buffer size to 1
        self.processor.packets = [
            [0, 3],  # Packet 1: arrival time 0, processing time 3
            [1, 1],  # Packet 2: arrival time 1, processing time 1
            [2, 1],  # Packet 3: arrival time 2, processing time 1
        ]
        expected_results = ['0', '-1', '-1']  # Only first packet can be processed
        results = self.processor.network_packets(self.processor.buffer_size, self.processor.packets, len(self.pro-
cessor.packets))
        self.assertEqual(results, expected_results)

if __name__ == '__main__':
    unittest.main()
```

Вывод по задаче:

Реализация обработки сетевых пакетов в данном коде является хорошей основой для работы с задачами, связанными с сетевыми потоками.

Задача №3. Обработка сетевых пакетов

Текст задачи:

В этой задаче вы реализуете программу для моделирования обработки сетевых пакетов.

- Вам дается серия входящих сетевых пакетов, и ваша задача - смоделировать их обработку. Пакеты приходят в определенном порядке. Для каждого номера пакета i вы знаете время, когда пакет прибыл A_i и время, необходимое процессору для его обработки P_i (в миллисекундах). Есть только один процессор, и он обрабатывает входящие пакеты в порядке их поступления. Если процессор начал обрабатывать какой-либо пакет, он не прерывается и не останавливается, пока не завершит обработку этого пакета, а обработка пакета i занимает ровно P_i миллисекунд.

  Компьютер, обрабатывающий пакеты, имеет сетевой буфер фиксированного размера S. Когда пакеты приходят, они сохраняются в буфере перед обработкой. Однако, если буфер заполнен, когда приходит пакет (есть S пакетов, которые прибыли до этого пакета, и компьютер не завершил обработку ни одного из них), он отбрасывается и не обрабатывается вообще. Если несколько пакетов поступают одновременно, они сначала все сохраняются в буфере (из-за этого некоторые из них могут быть отброшены - те, которые описаны позже во входных данных). Компьютер обрабатывает пакеты в порядке их поступления и начинает обработку следующего доступного пакета из буфера, как только заканчивает обработку предыдущего. Если в какой-то момент компьютер не занят и в буфере нет пакетов, компьютер просто ожидает прибытия следующего пакета. Обратите внимание, что пакет покидает буфер и освобождает пространство в буфере, как только компьютер заканчивает его обработку.

Код программы:

```
import sys
import os
from Lab5.utils import read_input, write_output, decorate

sys.path.append(os.path.abspath(os.path.join(os.path.dirname(__file__), '../..')))


class NetworkPacketsProcessor:
    def __init__(self):
        input_file = read_input(task=3)
        first_line = input_file[0].split()
        self.buffer_size = int(first_line[0])
        self.n = int(first_line[1])
        self.packets = [list(map(int, i.split())) for i in input_file[1:] if len(i.split()) == 2]
        self.buffer = []
```

```
        self.start_times = []

    def network_packets(self, buffer_size, packets, n):
        for packet in packets:
            if len(packet) != 2:
                print(f"Неверный пакет: {packet}")
                continue

            arrival_time, processing_time = packet

            while self.buffer and self.buffer[0] <= arrival_time:
                self.buffer.pop(0)

            if len(self.buffer) >= self.buffer_size:
                self.start_times.append('-1')
            else:
                start_time = arrival_time if not self.buffer else self.buffer[-1]
                self.start_times.append(str(start_time))
                self.buffer.append(start_time + processing_time)

        return self.start_times


def main():
    npp = NetworkPacketsProcessor()
    res = npp.network_packets(npp.buffer_size, npp.packets, npp.n)
    write_output(3, *res)
    [print(i) for i in res]
    print()


if __name__ == "__main__":
    decorate(task=3, task_name='network_packet')
```

Описание работы программы:

Основные Компоненты:

Импорты:

Код импортирует модули sys, os, а также функции read_input, write_output и decorate из Lab5.utils.

Класс NetworkPacketsProcessor:

Инициализация:

В конструкторе __init__ происходит считывание входных данных, где первая строка содержит размер буфера и количество пакетов. Затем считываются пакеты, каждый из которых представлен парой значений: время прихода и время обработки.

Создаются пустые списки self.buffer для хранения времени завершения обработки пакетов и self.start_times для хранения времени начала обработки каждого пакета.

Метод network_packets(buffer_size, packets, n):

Основной метод для обработки пакетов. Для каждого пакета:

Проверяется, корректно ли он задан (должен содержать два значения).

Удаляются из буфера все пакеты, которые уже обработаны к моменту прихода текущего пакета.
Если буфер заполнен, в start_times добавляется '-1', что означает, что пакет не может быть обработан. В противном случае вычисляется время начала обработки текущего пакета и добавляется в start_times, а также обновляется буфер с учетом времени обработки.
Функция main():
Создает экземпляр класса NetworkPacketsProcessor, вызывает метод network_packets для обработки пакетов и записывает результат в выходной файл. Также выводит результат на экран.
Точка Входа:
Код проверяет, запускается ли он как основной модуль, и вызывает функцию decorate, которая может использоваться для логирования или управления задачами.

Тесты:

```
from Lab5.task3.src.network_packet import *

import unittest

class TestNetworkPacketsProcessor(unittest.TestCase):

    def setUp(self):
        # Prepare test data for the NetworkPacketsProcessor
        self.processor = NetworkPacketsProcessor()
        # Override the input data for testing purposes
        self.processor.buffer_size = 2
        self.processor.n = 5
        self.processor.packets = [
            [0, 3],  # Packet 1: arrival time 0, processing time 3
            [1, 9],  # Packet 2: arrival time 1, processing time 9
            [2, 6],  # Packet 3: arrival time 2, processing time 6
            [3, 4],  # Packet 4: arrival time 3, processing time 4
            [4, 5],  # Packet 5: arrival time 4, processing time 5
        ]
        self.processor.buffer = []
        self.processor.start_times = []


    def test_buffer_limit(self):
        self.processor.buffer_size = 1  # Set buffer size to 1
        self.processor.packets = [
            [0, 3],  # Packet 1: arrival time 0, processing time 3
            [1, 1],  # Packet 2: arrival time 1, processing time 1
            [2, 1],  # Packet 3: arrival time 2, processing time 1
        ]
        expected_results = ['0', '-1', '-1']  # Only first packet can be processed
        results = self.processor.network_packets(self.processor.buffer_size, self.processor.packets, len(self.pro-
cessor.packets))
        self.assertEqual(results, expected_results)

if __name__ == '__main__':
    unittest.main()
```

Вывод по задаче:

Реализация обработки сетевых пакетов в данном коде является хорошей основой для работы с задачами, связанными с сетевыми потоками.

Задача №4. Построение пирамиды

Текст задачи:

В этой задаче вы преобразуете массив целых чисел в пирамиду. Это важнейший шаг алгоритма сортировки под названием `HeapSort`. Гарантированное время работы в худшем случае составляет $O(n \log n)$, в отличие от ***среднего*** времени работы `QuickSort`, равного $O(n \log n)$. `QuickSort` обычно используется на практике, потому что обычно он быстрее, но HeapSort используется для внешней сортировки, когда вам нужно отсортировать огромные файлы, которые не помещаются в памяти вашего компьютера.

Первым шагом алгоритма `HeapSort` является создание пирамиды (heap) из массива, который вы хотите отсортировать.

Ваша задача - реализовать этот первый шаг и преобразовать заданный массив целых чисел в пирамиду. Вы сделаете это, применив к массиву определенное количество перестановок (swaps). Перестановка - это операция, как вы помните, при которой элементы a_i и a_j массива меняются местами для некоторых i и j. Вам нужно будет преобразовать массив в пирамиду, используя только $O(n)$ перестановок. Обратите внимание, что в этой задаче вам нужно будет использовать `min-heap` вместо `max-heap`.

Задан массив из n целых чисел - $a_1, ..., a_n$ и число $m < n$, нужно найти максимум среди последовательности ("окна") $\{a_i, ..., a_{i+m-1}\}$ для каждого значения $1 \leq i \leq n - m + 1$. Простой алгоритм решения этой задачи за $O(nm)$ сканирует каждое "окно"отдельно.

Ваша цель - алгоритм за $O(n)$.

Код программы:

```
import sys
import os

from Lab5.utils import read_input, write_output, decorate

sys.path.append(os.path.abspath(os.path.join(os.path.dirname(__file__), '../..')))


def min_heapify(array, n, i, swaps):
    smallest = i
    left = 2 * i + 1
    right = 2 * i + 2

    if left < n and array[left] < array[smallest]:
        smallest = left

    if right < n and array[right] < array[smallest]:
```

```
        smallest = right

    if smallest != i:
        array[i], array[smallest] = array[smallest], array[i]
        swaps.append((i, smallest))
        min_heapify(array, n, smallest, swaps)

def build_min_heap(array):
    n = len(array)
    swaps = []

    # Построение пирамиды
    for i in range(n // 2 - 1, -1, -1):
        min_heapify(array, n, i, swaps)

    return swaps

def result(swaps):
    res = [f"{len(swaps)}"]
    for swap in swaps:
        res.append(f"{swap[0]} {swap[1]}")
    return res

def main():
    input_file = read_input(4)
    array = list(map(int, input_file[1].split()))

    swaps = build_min_heap(array)
    res = result(swaps)
    write_output(4, *res)
    [print(i) for i in res]
    print()

if __name__ == "__main__":
    decorate(task=4, task_name='heap_builder')
```

Тесты:

```
import unittest
from Lab4.task12.src.max_in_moving_sequence import *
from Lab4.utils import read_input


class TestStack(unittest.TestCase):
    def setUp(self):
        self.queue = Queue()

    def test_case_0(self):
        # given
        self.queue.n = 5
        self.queue.arr = [1, 3, 5, 7, 9]
        self.queue.m = 3
        expected_result = ['5', '7', '9']

        # when
        result = self.queue.result()

        # then
        self.assertEqual(expected_result, result)
```

```
def test_case_1(self):
    # given
    self.queue.n = 6
    self.queue.arr = [2, 4, 1, 3, 5, 6]
    self.queue.m = 4
    expected_result = ['4', '5', '6']  # Максимумы для каждой подгруппы

    # when
    result = self.queue.result()

    # then
    self.assertEqual(expected_result, result)

def test_case_2(self):
    # given
    self.queue.n = 5
    self.queue.arr = [10, 20, 30, 40, 50]
    self.queue.m = 2
    expected_result = ['20', '30', '40', '50']  # Максимумы для каждой подгруппы

    # when
    result = self.queue.result()

    # then
    self.assertEqual(expected_result, result)

def test_case_3(self):
    # given
    self.queue.n = 4
    self.queue.arr = [5, 3, 8, 1]
    self.queue.m = 3
    expected_result = ['8', '8']  # Максимумы для каждой подгруппы

    # when
    result = self.queue.result()

    # then
    self.assertEqual(expected_result, result)

def test_case_4(self):
    # given
    self.queue.n = 7
    self.queue.arr = [1, 2, 3, 4, 5, 6, 7]
    self.queue.m = 5
    expected_result = ['5', '6', '7']  # Максимумы для каждой подгруппы

    # when
    result = self.queue.result()

    # then
    self.assertEqual(expected_result, result)

def test_case_5(self):
    # given
    self.queue.n = 6
    self.queue.arr = [7, 3, 5, 8, 1, 2]
    self.queue.m = 4
    expected_result = ['8', '8', '8']  # Максимумы для каждой подгруппы

    # when
```

```
        result = self.queue.result()

        # then
        self.assertEqual(expected_result, result)

    def test_case_6(self):
        # given
        self.queue.n = 5
        self.queue.arr = [1, 3, 5, 7, 9]
        self.queue.m = 3
        expected_result = ['5', '7', '9']

        # when
        result = self.queue.result()

        # then
        self.assertEqual(expected_result, result)

if __name__ == '__main__':
    unittest.main()
```

Описание работы кода:

Основные Компоненты:

Импорты:

Код использует модули sys, os, а также функции read_input, write_output и decorate из Lab5.utils.

Функция min_heapify(array, n, i, swaps):

Основная функция для поддержания свойства минимальной кучи. Она проверяет узел с индексом i и его дочерние узлы:

Если левый дочерний узел меньше текущего узла, обновляет индекс на левый дочерний.

Если правый дочерний узел меньше текущего узла (или обновленного), обновляет индекс на правый дочерний.

Если индекс изменился, выполняет обмен элементов и рекурсивно вызывает min_heapify для обновленного узла.

Функция build_min_heap(array):

Строит минимальную кучу из массива, начиная с первого не-листового узла и вызывая min_heapify для каждого узла. Возвращает список обменов, которые были выполнены.

Функция result(swaps):

Форматирует результат для вывода. Сначала добавляет количество обменов, затем добавляет каждый обмен в формате "индекс1 индекс2".

Функция main():

Считывает входные данные, преобразует их в массив целых чисел, строит минимальную кучу, формирует результат и записывает его в выходной файл. Также выводит результат на экран.

Точка Входа:
Код проверяет, запущен ли он как основной модуль, и вызывает функцию decorate, которая может использоваться для логирования или управления задачами.

Тесты:

```
import unittest
from Lab5.task4.src.heap_builder import *


class TestStack(unittest.TestCase):

    def test_case_input(self):
        # given
        array = [5, 4, 3, 2, 1]
        expected_result = ['3', '1 4', '0 1', '1 3']

        # when
        swaps = build_min_heap(array)
        res = result(swaps)

        # then
        self.assertEqual(expected_result, res)

    def test_case_0(self):
        # given
        array = [5, 4, 3, 2, 1]
        expected_result = ['3', '1 4', '0 1', '1 3']

        # when
        swaps = build_min_heap(array)
        res = result(swaps)

        # then
        self.assertEqual(expected_result, res)


    def test_case_1(self):
        # given
        array = [1, 2, 3, 4, 5]
        expected_result = ['0']  # Никаких обменов не должно быть

        # when
        swaps = build_min_heap(array)
        res = result(swaps)

        # then
        self.assertEqual(expected_result, res)

    def test_case_2(self):
        # given
        array = [3, 1, 4, 1, 5, 9, 2, 6, 5]
        expected_result = ['3', '2 6', '0 1', '1 3']

        # when
```

```
        swaps = build_min_heap(array)
        res = result(swaps)

        # then
        self.assertEqual(expected_result, res)

    def test_case_3(self):
        # given
        array = [10, 9, 8, 7, 6, 5, 4, 3, 2, 1]
        expected_result = ['8', '4 9', '3 8', '2 6', '1 4', '4 9', '0 1', '1 3', '3 7']

        # when
        swaps = build_min_heap(array)
        res = result(swaps)

        # then
        self.assertEqual(expected_result, res)

    def test_case_4(self):
        # given
        array = [1, 3, 2, 4, 5]
        expected_result = ['0']  # Никаких обменов не должно быть

        # when
        swaps = build_min_heap(array)
        res = result(swaps)

        # then
        self.assertEqual(expected_result, res)

    def test_case_5(self):
        # given
        array = [5, 3, 8, 4, 2]
        expected_result = ['3', '1 4', '0 1', '1 4']

        # when
        swaps = build_min_heap(array)
        res = result(swaps)

        # then
        self.assertEqual(expected_result, res)

if __name__ == '__main__':
    unittest.main()
```

Вывод по задаче:

Реализация построения минимальной кучи в данном коде является хорошей основой для работы с задачами, связанными с кучами и приоритетными очередями.

Вывод

В данной работе мы изучили работу с деревьями, пирамидами, пирамидальной сортировкой.