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 6

Выполнил:
Данилова А.В.
К3141 (номер группы)

Проверила:
Артамонова В.Е.

Санкт-Петербург
2024 г.

Содержание отчета

Задачи по варианту

1 задача. Куча ли?

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 $1 \le i \le n$ выполняются условия:

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

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

Листинг кода:

```
from lab_5.utils.utils import work
from pathlib import Path


def left_child_idx(i):
    return 2  i + 1


def right_child_idx(i):
    return 2  i + 2


def is_heap(n, arr):
    for i in range(len(arr) // 2):
        if arr[i] > arr[left_child_idx(i)]:
            return "NO"
        if right_child_idx(i) < len(arr) and arr[i] >
arr[right_child_idx(i)]:
            return "NO"
    return "YES"


if __name__ == "__main__":
    work(Path(__file__).parts[-4], Path(__file__).stem, is_heap)
```

left_child_idx и right_child_idx: Эти функции вычисляют индексы левого и правого потомков узла с индексом i в бинарной куче, представленной в виде массива.

is_heap: Эта функция является основной и проверяет свойство кучи. Она итерируется по всем родительским узлам массива (len(arr) // 2). Для каждого родительского узла она проверяет два условия:

Проверяет, что значение родительского узла меньше или равно значению его левого потомка. Если это условие не выполняется, то массив не является кучей, и функция возвращает "NO".

Проверяет то же самое для правого потомка, но только если правый потомок существует (т.е. right_child_idx(i) находится в пределах массива).

Результат работы кода на примерах из текста задачи:

```
LAB NUMBER: 5
            TASK NUMBER: 1
                    INPUT DATA: (5, [1, 3, 2, 5, 4])
                    OUTPUT DATA: YES
```

Результат работы кода на максимальных и минимальных значениях:

	Время выполнения	Память
Нижняя граница диапазона значений входных данных из текста задачи	1.800013706088066e-06 секунд	30.9609375 МБ
Верхняя граница диапазона значений входных данных из текста задачи	0.1685128000099212 секунд	31.14453125 МБ

Вывод по задаче:
Задача демонстрирует важность понимания свойств структур данных, таких как куча, и способности проверить их выполнение. Правильная реализация алгоритма гарантирует точный ответ на вопрос о том, является ли данный массив неубывающей пирамидой.

7 задача. Снова сортировка

Напишите программу пирамидальной сортировки на Python для последовательности в убывающем порядке.

Листинг кода:

```
from lab_5.utils.utils import work
from pathlib import Path
```

```
def left(i):
    return 2  i + 1


def right(i):
    return 2  i + 2


def max_heapify(arr, n, i):
    _l = left(i)
    r = right(i)

    if _l < n and arr[_l] > arr[i]:
        largest = _l
    else:
        largest = i

    if r < n and arr[r] > arr[largest]:
        largest = r

    if largest != i:
        arr[i], arr[largest] = arr[largest], arr[i]
        max_heapify(arr, n, largest)


def build_max_heap(arr, n):
    for i in range((len(arr) - 1) // 2, -1, -1):
        max_heapify(arr, n, i)


def heapsort(n, arr):
    build_max_heap(arr, n)
    for i in range(n - 1, -1, -1):
        arr[i], arr[0] = arr[0], arr[i]
        max_heapify(arr, i, 0)

    return arr


if __name__ == "__main__":
    work(Path(__file__).parts[-4], Path(__file__).stem, heapsort)
```

left и right: Функции возвращают индексы левого и правого потомков узла с индексом i в бинарной куче, представленной массивом.

max_heapify: Эта рекурсивная функция восстанавливает свойство max-кучи в поддереве с корнем в индексе i. Она находит наибольший элемент среди узла i и его потомков, затем меняет местами наибольший элемент с элементом i и рекурсивно вызывает max_heapify для поддерева, в котором произошла перестановка.

build_max_heap: Эта функция строит max-кучу из массива arr. Она проходит по массиву снизу вверх, начиная с последнего родительского узла ((len(arr) - 1) // 2), и для каждого родительского узла вызывает max_heapify, чтобы восстановить свойство max-кучи в его поддереве.

heapsort: Это главная функция, реализующая алгоритм heapsort. Сначала она строит max-кучу из массива, используя build_max_heap. Затем, она повторяет следующие шаги:

1. Меняет местами наибольший элемент (корень кучи) с последним элементом массива.
2. Уменьшает размер кучи на 1 (игнорируя последний элемент, который уже отсортирован).
3. Вызывает max_heapify для корня кучи, чтобы восстановить свойство max-кучи. Эти шаги повторяются до тех пор, пока размер кучи не станет равным 0.

Результат работы кода на примерах из текста задачи:

```
LAB NUMBER: 5
            TASK NUMBER: 7
                    INPUT DATA: (6, [1, 2, 3, 4, 5, 6])
                    OUTPUT DATA: [1, 2, 3, 4, 5, 6]
```

Результат работы кода на максимальных и минимальных значениях:

	Время выполнения	Память
Нижняя граница диапазона значений входных данных из текста задачи	0.002114700007950887 секунд	32.921875 МБ
Верхняя граница диапазона значений входных данных из текста задачи	0.32775679999031126 секунд	35.69921875 МБ

Вывод по задаче:
На практике пирамидальная сортировка демонстрирует хорошую производительность, особенно в сравнении с другими алгоритмами сортировки с гарантированной сложностью O(n log n), такими как merge sort.

Дополнительные задачи

3 задача. Обработка сетевых пакетов

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.

Листинг кода:

```
from lab_5.utils.utils import work
from pathlib import Path


def processing_net_packets(s: int, n: int, packages):
    if n == 0:
        return None
    deque = []
    result = []
    head = 0
```

```
    buffer_time = packages[0][0]
    for p in packages:
        start_time, p_i = p
        head = max([head] + [index for index in range(len(deque)) if
deque[index] <= start_time])
        if len(deque[head+1:]) < s:
            result += [max(buffer_time, start_time)]
            buffer_time += p_i
            deque.append(buffer_time)
        else:
            result += [-1]
    return result


if __name__ == "__main__":
    work(Path(__file__), processing_net_packets)
```

Входные данные:

s: размер буфера (максимальное количество пакетов, которые могут храниться одновременно).

n: количество пакетов.

packages: кортеж, содержащий информацию о пакетах. Каждый элемент кортежа — это кортеж (start_time, p_i), где start_time — время прибытия пакета, а p_i — время обработки пакета.

deque: Список, представляющий буфер. Он хранит время завершения обработки пакетов, находящихся в буфере.

result: Список, в который записываются результаты обработки пакетов. Если пакет обработан, записывается время начала его обработки; если буфер полон, записывается -1.

head: Индекс в deque, указывающий на первый пакет в буфере, который может быть обработан (т.е. время его обработки меньше или равно времени прибытия текущего пакета).

buffer_time: Время, когда освободится буфер после обработки пакета.

Цикл for p in packages:

head = max([head] + [index for index in range(len(deque)) if deque[index] <= start_time]):

Находит индекс head — первого пакета в буфере, время окончания обработки которого меньше или равно времени прибытия текущего пакета. Это эффективно очищает буфер от завершенных пакетов.

if len(deque[head+1:]) < s:

Проверяет, есть ли место в буфере. Если места достаточно, пакет добавляется в буфер (deque.append(buffer_time)), а время начала обработки и время завершения обработки записываются в result. Если места нет (else), в result записывается -1.

Результат работы кода на примерах из текста задачи:

```
LAB NUMBER: 5
            TASK NUMBER: 3
                    INPUT DATA: (2, 3, [0, 1], [3, 1], [10, 1])
                    OUTPUT DATA: [0, 3, 10]
```

Результат работы кода на максимальных и минимальных значениях:

	Время выполнения	Память
Нижняя граница диапазона значений входных данных из текста задачи	1.1099997209385037e-05 секунд	30.86328125 МБ
Верхняя граница диапазона значений входных данных из текста задачи	3.0664576000126544 секунд	47.6484375 МБ

Вывод по задаче:
Эта задача иллюстрирует важность управления ресурсами (буфер) в системах обработки данных. Алгоритм, использующий очередь, обеспечивает эффективное моделирование обработки пакетов с учетом ограничений буфера.

4 задача. Построение пирамиды

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 ai и aj 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.

Листинг кода:

```
from lab_5.utils.utils import work
from pathlib import Path


class MinHeap:
    def __init__(self, n, array):
        self.heap = array
        self.high = n
        self.swaps = []

    def heap_sort(self):
        for i in range(self.high // 2 - 1, -1, -1):
            self.swap(i)
        return len(self.swaps), self.swaps

    def swap(self, index):
        smallest = index
        left = 2  index + 1
        right = 2  index + 2

        if left < self.high and self.heap[left] < self.heap[smallest]:
            smallest = left

        if right < self.high and self.heap[right] < self.heap[smallest]:
            smallest = right

        if smallest != index:
            self.swaps += [(index, smallest)]
            self.heap[index], self.heap[smallest] = self.heap[smallest],
self.heap[index]
            self.swap(smallest)


def main(n, arr):
    return MinHeap(n, arr).heap_sort()
```

Класс MinHeap:

__init__(self, n, array):

Инициализирует объект MinHeap с массивом array размера n. high хранит размер кучи, а swaps — список кортежей, представляющих перестановки элементов во время построения кучи.

heap_sort(self):

Строит min-кучу и возвращает количество перестановок и список самих перестановок.

swap(self, index):

Рекурсивно восстанавливает свойство min-кучи, начиная с узла с индексом index. Находит наименьший элемент среди текущего узла и его детей, меняет его местами с текущим и рекурсивно вызывает swap для поддерева.

Функция main(n, arr):

Создает объект MinHeap и вызывает метод heap_sort(), возвращая результат.

Результат работы кода на примерах из текста задачи:

```
LAB NUMBER: 5
            TASK NUMBER: 4
                    INPUT DATA: (5, [1, 2, 3, 5, 4])
                    OUTPUT DATA: (3, [(1, 4), (0, 1), (1, 3)])
```

Результат работы кода на максимальных и минимальных значениях:

	Время выполнения	Память
Нижняя граница диапазона значений входных данных из текста задачи	5.600013537332416e-06 секунд	30.7109375 МБ
Верхняя граница диапазона значений входных данных из текста задачи	0.060863699996843934 секунд	43.62890625 МБ

Вывод по задаче:

Благодаря кучам можно реализовать интересную модифицированную сортировку с выводом перестановок.

Вывод

Куча - очень простая структура данных для выделения памяти под другие структуры данных. Стек совершенно не подходит, в т.ч. для хранения информации о свободных и занятых ячейках, по причине того, что доступ к данным кучи не должен быть ограничен последним добавленным

элементом. Грубо: куча нужна для динамического выделения в ней участков памяти нужного размера.