Министерство образования и науки Российской Федерации

Федеральное государственное автономное образовательное учреждение высшего образования «Санкт-Петербургский национальный исследовательский университет информационных технологий, механики и оптики»

Мегафакультет трансляционных информационных технологий

Факультет инфокоммуникационных технологий

Дисциплина: Алгоритмы и структуры данных

**Отчет по Лабораторной работе №5**

Выполнила: Микулина Алиса Романовна

Группа: K3143, 1 курс

Преподаватель: Харьковская Татьяна Александровна

Санкт-Петербург

09.12.2021

**Описание задания**

**Задание 1.**

Необходимо определить, является содержимое массива кучей (ну или возрастающей пирамидой, на языке умных). Вывести YES или NO в зависимости от полученного результата.

**Задание 2.**

Нужно соорудить структуру данных, то есть класс, в который мы сможем считывать входные данные так, чтобы они выстраивались в дерево. Дерево не бинарное, то есть “детей” может быть больше двух. Задача в том, чтобы можно было посчитать высоту такого дерева. Высота = максимальное расстояние от листа до корня.

**Задание 4.**

Необходимо построить убывающую пирамиду, используя min\_heap. При каждой перестановке выводим индексы элементов, чтобы убедиться, что эти перестановки происходят. Количество перестановок не должно превышать 4n, где n – количество элементов в куче.

**Задание 7.**

Реализовать пирамидальную сортировку, в результате которой выводится массив, отсортированный по убыванию.

**Описание решения и исходный код**

**Задача 1.**

Как и сказано в задаче, кучу можно реализовать на основе массива (ура, не надо писать класс!!!). Задачка элементарная, так как у каждого предка есть возможность определить индексы детей, ведь детей максимум двое. В начало массива пихаем -1, чтобы хотя бы начало было точно меньше всего остального. У каждого предка смотрим детей. Если оба ребенка меньше предка: идем дальше. Если меньше, то сразу возвращаем ошибку. Если ни одной ошибки не найдено, возвращаем YES. Радуемся, улыбаемся и машем, запивая чаем грусть от того, что следующая задачка неубиваемо-сложная.

(И, да, я переселилась из PyCharmа в VS Code, поэтому теперь все черненькое, а буковки красивенькие, обожаю этот стиль.)

def check\_level(*i*, *heap*):

    left = 2 \* *i*

    right = 2 \* *i* + 1

*if* left < len(*heap*):

*if* not *heap*[left] >= *heap*[*i*]:

*return* False

*if* right < len(*heap*):

*if* not *heap*[right] >= *heap*[*i*]:

*return* False

*return* True

heap = input().split(' ')

*for* i *in* range(len(heap)):

    heap[i] = int(heap[i])

heap.insert(0, -1)

ans = "YES"

*for* i *in* range(len(heap)):

*if* check\_level(i, heap):

*continue*

*else*:

        ans = "NO"

*break*

print(ans)

**Задача 2.**

Пришлось полностью перерешивать эту чудо-задачку утром перед защитой, ибо на тестах из чата оно великолепным образом не работало… И грустно, и скучно, и некому в душу поныть.

*# n = 10 \*\* 1*

*# heap = [i + 1 for i in range(n) + 1]*

*# heap[-1] = -1*

*# heap = [4, -1, 4, 1, 1]*

*# n = 5*

*# heap = [-1, 0, 4, 0, 3]*

*# n = 5*

heap = [9, 7, 5, 5, 2, 9, 9, 9, 2, -1]

n = 10

heap\_len = [0 *for* i *in* range(n)]

max\_depth = 0

*if* len(heap) == 1:

    print(max\_depth)

*else*:

*for* i *in* range(len(heap)):

        now = i

        temporary = []

*while* True:

*if* heap[now] == -1:

                heap\_len[now] = 1

            temporary.append(now)

*if* heap\_len[now] == 0:

                now = heap[now]

*else*:

*break*

*for* j *in* range(len(temporary) - 2, -1, -1):

            heap\_len[temporary[j]] = heap\_len[temporary[j + 1]] + 1

*if* heap\_len[temporary[j]] > max\_depth:

                max\_depth = heap\_len[temporary[j]]

            i

    print(max\_depth)

Вот это вот чудо вроде как работает, я решила не заморачиваться с классами, ибо уже просто не было времени. Считаем глубину для каждого элемента и вуаля! \* сижу плачу по классам \*

ВСЕ ЧТО ДАЛЕЕ НЕ РАБОТАЕТ ААААААААААААААААААА

На первый взгляд задачка — ну нечего делать. Да. Как бы не так. Я все еще страдаю от написания классов, но уже не так сильно, конечно. Тут больше проблема не в том, чтобы написать класс с функциями, а в том, чтобы сделать нормальную функцию для добавления элементов, которая бы рекурсивно искала, куда этот элемент добавить.

Сначала, казалось бы, все просто. Считываем строку, преобразуем в массив, где уже, казалось бы, все нормально: индексы соответствуют числам, которые мы добавляем… Да. Но это неудобно. Но сначала надо написать класс.

Прописываем ноды, где есть ссылка на само значение элемента (далее “вершинка”), на родителя элемента и на его детей. Детей делаем списком, а не left-right, потому что их может быть много. А может быть только один. Проще хранить в массиве.

Далее пишем само дерево, в нем \_\_init\_\_ и push. Push у нас зависит от самого дерева, элемента, который мы добавляем, и его родителя, чтобы сразу привязаться к нему. Первый элемент (тот самый, с -1) делаем корнем, а далее все сложнее. Нужно сделать так, чтобы остальные элементы привязались к своим родителям. И вот тут загадка:

А что, если пока мы искали -1, мы уже пропустили половину элементов, не добавив их никуда? А если -1 вообще в конце? Их вообще можно куда-то сохранить, чтобы они, не найдя себе родителя, не удалялись в закат? А если по несколько раз итерироваться по одному и тому же массиву, после добавления каждого элемента, это же какие потери времени…

Вот я сидела, думала, добавляла, и поняла, нужна сортировка, причем еще до того, как мы пойдем выращивать дерево. Сортировка, при этом, не должна нарушить иерархию, а наоборот должна ее упростить. То есть, нам нужно что-то, что будет хранить и сами элементы, и их родителей, еще и чтобы они все добавляться потом могли по очереди, и мы не добавляли элемент с родителем 4, если у нас максимальная вершинка в массиве вообще 2, и крепиться не к чему.

Придумала использовать сортировку подсчетом. С ее помощью, я храню детей в элементах с индексами родителей этих элементов, а элемент, у которого находим -1, записываем в самый конец массива: его оттуда брать удобно, а потом будет меньше проблем с итерацией, чем если бы мы его хранили в самом начале.

Теперь, мы сначала добавляем первую вершинку, делая ее корнем, а после этого уже рекурсивно бегаем по дереву, добавляя новых детей к уже 100% существующим родителям. Звучит, как раздача в детдоме, ну да и ладно.

После построения этого чудо-дерева (вот бы оно еще и желания исполняло, я бы попросила автомат по информатике) мы точно таким же рекурсивным макаром пробегаем по детям-родителям, добавляя к счетчику циферки каждый раз, как увеличивается глубина рекурсии. Каждый раз сравниваем с максимальной глубиной из отдельной переменой и обновляем при необходимости.

Ура, мы распределили детей и посчитали глубину семейной кучи!

class Node:

    def \_\_init\_\_(*self*, *data*):

*self*.data = *data*

*self*.parent = None

*self*.children = []

class Tree:

    def \_\_init\_\_(*self*):

*self*.root = None

    def push(*self*, *elem*, *parent*):

*if* *parent* == -1:

*self*.root = Node(*elem*)

*self*.parent = None

*return*

*if* *self*.root:

            new\_child = Node(*elem*)

            new\_child.parent = *parent*

            iteration = *self*.root

            itr\_check = *self*.root

            parent\_adder(iteration, itr\_check, *parent*, new\_child)

*return*

    def to\_array(*self*):

        array = []

        start = *self*.root

        array.append(start.data)

*return* turn(array, start)

    def depth(*self*):

*if* *self*.root == None:

*return* 0

        max\_length = 1

        local\_counter = 1

        start = *self*.root

*return* count\_depth(max\_length, local\_counter, start)

def count\_depth(*max\_length*, *local\_counter*, *start*):

*if* *max\_length* < *local\_counter*:

*max\_length* = *local\_counter*

*for* child *in* *start*.children:

*local\_counter* += 1

        new\_start = child

*if* new\_start.children:

*return* count\_depth(*max\_length*, *local\_counter*, new\_start)

*return* *max\_length*

def turn(*array*: list, *start*: Node):

*for* child *in* *start*.children:

*array*.append(child.data)

        new\_start = child

*if* new\_start.children:

*return* turn(*array*, new\_start)

*return* *array*

def parent\_adder(*iteration*: Node, *itr\_check*: Node, *parent*, *new\_child*: Node):

*if* *parent* == *itr\_check*.data:

*itr\_check*.children.append(*new\_child*)

*return*

*if* not *iteration*.children:

*return*

*for* child *in* *iteration*.children:

*if* child.data == *parent*:

            child.children.append(*new\_child*)

*return*

*if* child.children:

            new\_iteration = child

*return* parent\_adder(new\_iteration, *itr\_check*, *parent*, *new\_child*)

*return*

def count\_sort(*array*):

    counts = [[] *for* i *in* range(len(*array*) + 1)]

    ans = []

*for* i *in* range(len(*array*)):

*if* *array*[i] == -1:

            counts[-1].append(-1)

            counts[-1].append(i)

*continue*

        counts[*array*[i]].append(i)

*return* counts

tree = Tree()

aaa = '4 -1 4 1 1'

heap = aaa.split(' ')

*for* i *in* range(len(heap)):

    heap[i] = int(heap[i])

heap = count\_sort(heap)

tree.push(heap[-1][-1], heap[-1][0])

*for* i *in* range(len(heap) - 1):

*for* j *in* heap[i]:

        tree.push(j, i)

print(tree.to\_array())

print(tree.depth())

**Задача 4.**

Находясь в полной прострации после двух решения предыдущей задачки, решила написать хоть что-то попроще, то есть простое построение пирамиды. Реализуем две функции: min\_heapify и build\_min\_heap. Сравниваем элементы в массиве с их детьми по очереди, при необходимости меняем местами. При каждой смене местами заново проверяем рекурсивно всех нижестоящих детей, чтобы убедиться, что зависимость не нарушена, а если нарушена – восстанавливаем.

После предыдущей задачки так просто, что плакать хочется. Но нужно еще сортировку добить было, а уже второй час ночи… Ммммм… вкусно

def min\_heapify(*i*, *heap*, *ans*):

*if* *i* < 0:

*return*

    left = 2 \* *i* + 1

    right = 2 \* *i* + 2

    smallest = *i*

*if* left < len(*heap*):

*if* *heap*[left] < *heap*[*i*]:

            smallest = left

*else*:

        smallest = *i*

*if* right < len(*heap*):

*if* *heap*[right] < *heap*[smallest]:

            smallest = right

*if* smallest != *i*:

*ans*.append([*heap*[*i*], *heap*[smallest]])

*heap*[*i*], *heap*[smallest] = *heap*[smallest], *heap*[*i*]

*return* min\_heapify(smallest, *heap*, *ans*)

def build\_min\_heap(*to\_heap*):

    ans = []

    size = len(*to\_heap*)

*for* i *in* range(size // 2, -1, -1):

        min\_heapify(i, *to\_heap*, ans)

    print(len(ans))

*for* pair *in* ans:

        print(pair[0], pair[1])

*return* *to\_heap*

line = '5 4 3 2 1'

to\_heap = line.split(' ')

*for* i *in* range(len(to\_heap)):

    to\_heap[i] = int(to\_heap[i])

print(build\_min\_heap(to\_heap))

**Задача 7.**

Так как четвертую задачу я решила, а седьмая, по факту, строится на основе 4ой, во втором часу ночи было решено написать эту сортировку “типа быстро отделаться и все”. Было бы хорошо, если бы не было так грустно, ведь я опять запуталась в индексах… Это моя вечная боль, путать индексацию (а еще аргументы при обращении к функциям, да… эту ошибку я искала пол часа…).

Ну в общем снимаем сверху кучи элемент, меняем его местами с iтым элементом с конца, прогоняем min\_heapify, чтобы сохранить убывающую кучу убывающей и радуемся жизни. Да, я потратила на эти 10 строчек полтора часа своей жизни И ЧТО?!))))

def min\_heapify(*heap*, *heap\_size*, *i*):

*if* *i* < 0:

*return*

    left = 2 \* *i* + 1

    right = 2 \* *i* + 2

    smallest = *i*

*if* left < *heap\_size*:

*if* *heap*[left] < *heap*[*i*]:

            smallest = left

*else*:

        smallest = *i*

*if* right < *heap\_size*:

*if* *heap*[right] < *heap*[smallest]:

            smallest = right

*if* smallest != *i*:

*heap*[*i*], *heap*[smallest] = *heap*[smallest], *heap*[*i*]

*return* min\_heapify(*heap*, *heap\_size*, smallest)

def build\_min\_heap(*to\_heap*):

    heap\_size = len(*to\_heap*)

*for* i *in* range(heap\_size, -1, -1):

        min\_heapify(*to\_heap*, heap\_size, i)

*return* *to\_heap*

def heap\_sort(*array*):

    heap\_size = len(*array*)

*array* = build\_min\_heap(*array*)

*for* i *in* range(heap\_size - 1, 0, -1):

*array*[i], *array*[0] = *array*[0], *array*[i]

        min\_heapify(*array*, i, 0)

*return* *array*

line = '1 2 3 4 5'

array = line.split(' ')

*for* i *in* range(len(array)):

    array[i] = int(array[i])

print(heap\_sort(array))

**Описание проведенных тестов.**

**1**

|  |  |
| --- | --- |
| **Input** | **Output** |
| 1 0 1 2 0 | NO |
| 1 3 2 5 4 | YES |
| 1 2 3 4 5 6 7 8 9 1 | NO |
| 1 2 3 4 5 6 7 8 9 10 | YES |

**2**

|  |  |
| --- | --- |
| **Input** | **Output** |
| 4 -1 4 1 1 | 3 |
| -1 0 4 0 3 | 4 |
| 9, 7, 5, 5, 2, 9, 9, 9, 2, -1 | 4 |
| 10 \*\* 5 | 100000 |

**4**

|  |  |
| --- | --- |
| **Input** | **Output** |
| 5 4 3 2 1 | 3  4 1  5 1  5 2 |
| 1 2 3 4 5 | 0 |
| 18 34 4 5 2 74 9 7 6 8 19 | 5  34 2  34 8  18 2  18 5  18 6 |

**7**

|  |  |
| --- | --- |
| **Input** | **Output** |
| 1 2 3 4 5 | [5, 4, 3, 2, 1] |
| 5 4 3 2 1 | [5, 4, 3, 2, 1] |
| 18 34 4 5 2 74 9 7 6 8 19 | [74, 34, 19, 18, 9, 8, 7, 6, 5, 4, 2] |

**Выводы по проделанной работе.**

И скучно, и грустно, и плакать хочется. Завтра защита, все, вроде, готово, а я устала дико. Мне еще английские слова сейчас учить. Спаситепомогите….

Что касается задач, очень приятно, что их всего 4, потому что все дедлайны и так заставляют задуматься о бренности этой жизни. Вообще кучи интересная штука, а еще мне прям понравилось, как мое дерево строится. Надеюсь на легкую защиту) потому что мозг и так превратился в кашу, а нервы – в клочок сконцентрированной вокруг нервного срыва энергии. Хочется спать, а спать некогда, так еще и лабу Добриборща надо делать. Спасите…

Страшно не закрыться, страшно не успеть, страшно, что не получится все решить. Да, пока что все получалось, пусть даже через дикие усилия, но сейчас кажется, что я уже на пределе… Живем-живем, так сказать))