

Homework 7

30 октября 2022 г.

Task 1

Насколько я понял в данном задании ключи = значения. Опишем функцию которая позволяет обойти двоичное дерево таким образом, чтобы вывести все ключи в порядке возрастания за $O(n)$.

def Printer(*node: int*):

```
    if node.left  $\neq$  Nill then  
    |     Printer(node.left)  
  
    end  
  
    print(node.value)  
  
    if node.right  $\neq$  Nill then  
    |     Printer(node.right)  
  
    end
```

И запустим эту функцию из корня.

Доказательство корректности.

Докажем по индукции. Для дерева высотой 1 корректность очевидна.

Пусть функция корректно работает для дерева высотой n , докажем что она корректно работает для дерева высотой $n + 1$. Сначала мы распечатает все значения по возрастанию находящиеся в левом поддереве корня, из свойств двоичного дерева поиска все эти значения будут меньше корня.

Затем мы распечатаем сам корень дерева.

И после этого распечатаем все значения правого поддереве корня, из свойств двоичного дерева поиска все эти значения будут больше корня.

Таким образом мы выведем все значения в порядке возрастания.

Так как на каждом вызове мы печатаем одну вершину, и все вершины мы печатаем ровно 1 раз, то сложность выполнения этой функции $O(n)$.

Task 2

Семейство H будет универсальным если $\forall x \neq y : \mathbb{P}(\hat{f}(x) = \hat{f}(y) \mid \hat{f} \in H, x, y) = \frac{1}{m} = \frac{1}{3}$.

$$\begin{aligned}\mathbb{P}(\hat{f}(x) = \hat{f}(y) \mid \hat{f} \in H, x, y) &= \mathbb{P}(\hat{f} = f)\mathbb{P}(f(x) = f(y)) \\ &\quad + \mathbb{P}(\hat{f} = g)\mathbb{P}(g(x) = g(y)) \\ &\quad + \mathbb{P}(\hat{f} = h)\mathbb{P}(h(x) = h(y)) \\ &= \mathbb{P}(\hat{f} = f)\mathbb{1}(f(x) = f(y)) \\ &\quad + \mathbb{P}(\hat{f} = g)\mathbb{1}(g(x) = g(y)) \\ &\quad + \mathbb{P}(\hat{f} = h)\mathbb{1}(h(x) = h(y))\end{aligned}$$

Возьмем $x = 0, y = 3$, тогда

$$\begin{aligned}\mathbb{P}(\hat{f}(x) = \hat{f}(y) \mid \hat{f} \in H, x, y) &= \mathbb{P}(\hat{f} = f)\mathbb{P}(f(x) = f(y)) \\ &\quad + \mathbb{P}(\hat{f} = g)\mathbb{P}(g(x) = g(y)) \\ &\quad + \mathbb{P}(\hat{f} = h)\mathbb{P}(h(x) = h(y)) \\ &= \frac{1}{3}\mathbb{1}(f(0) = f(3)) \\ &\quad + \frac{1}{3}\mathbb{1}(g(0) = g(3)) \\ &\quad + \frac{1}{3}\mathbb{1}(h(0) = h(3)) \\ &= \frac{1}{3} * 0 + \frac{1}{3} * 1 + \frac{1}{3} * 1 \neq \frac{1}{3}\end{aligned}$$

Следовательно семейство H не является универсальным.

Task 3

Уточнение: когда $b \geq \lceil \frac{n}{2} \rceil$ то устанавливаем значение $b = 0$ и переносим всю осмысленную часть массива в начало.

Доказательство амортизационной сложности.

Заведем для удобства еще один указатель c - конец нашей очереди. Будем считать, что добавление стоит условные 4 монеты.

Добавление в массив

Пусть 1 монету стоит добавление в текущий массив если в нем есть место.

Еще 1 монета нужна для копирования самого элемента в новый массив вдвое большего размера, если место закончилось.

Еще 1 монета нужна для копирования элемента под номером $i - \frac{n}{2}$ в массив вдвое большего размера, если место закончилось

Еще 1 монету отложим для переноса. Корректность.

Докажем по индукции.

База индукции очевидна.

Пусть в массива размера n закончилось место, тогда у правой половины массива есть 3 свободные монеты, то есть $3\frac{n}{2}$ монет. Создадим новый массив вдвое большего размера и скопируем туда текущий это будет стоить n монет, следовательно у монеты в правой заполненной части останется по свободной монете.

При добавлении новых элементов у каждого из них будет 3 свободные монеты и когда место снова закончится, у правой половины снова будет 3 свободные монеты которые смогут оплатить переезд.

Следовательно добавление при амортизационном анализе работает за $O(1)$.

Удаление

Тогда 1 монету стоит само добавление, а на 1 монету мы *оплатим* перенос элемента в первую половину, если такой состоится. Любой элемент мы переносим в начало только 1 раз (так как n только растёт), если уж элемент оказался в какой то момент в левой половине, то этот индекс уже не сможет попасть в правую половину массива. Следовательно удаление при амортизационном анализе работает за $O(1)$.

И мы использовали не более чем $2n$ памяти.

Task 4

Заведём HashMap ключом которого будет ключ, значением - приоритет.

Тогда добавление нового ключа с приоритетом будет стоить $O(1)$, установка приоритета так же $O(1)$, так как добавление и изменение элементов в HashMap осуществляются за $O(1)$.

Для того чтобы совершить extractmax, необходимо обойти все пары ключ, значение запоминая максимальное значение и соответствующий ему ключ. Тогда мы найдем ключ с максимальным значением за $O(n)$. После этого удаление этого ключа стоит $O(1)$, итоговая сложность extractmax $O(n)$.

Task 5

Заведём HashMap в котором ключом будет ключ, значением - количество повторений этого ключа.

Соответственно нам нужно пройтись по массиву, если ключ уже присутствует в HashMap то увеличить значение на 1, иначе создать со значением 1. Это занимает $O(n)$, так как элементов n , вставка и получение значения по ключу $O(1)$.

После этого создадим новый массив, в котором хранятся пары ключ, значение в произ-

вольном порядке. Это так же занимает $O(n)$ так как нам необходимо обойти HashMap размер которого не больше n .

После этого мы можем отсортировать данных массив с помощью CountingSort в убывающем порядке, сравнивая значения в нашем массиве (то есть количество повторений). Так как количество повторений минимум 1, максимум n , то сложность CountingSort $O(n)$.

Итого мы получили все ключи с количеством их повторений, в порядке убывания количества повторений за $O(n)$.

Task 6

Обозначим данный массив за a . Заведем 3 счетчика left, right, counter и HashSet currentnumbers, будем поддерживать следующий инвариант.

- $\text{left} < \text{right}$.
- $\text{currentnumbers} = \text{HashSet}(a[\text{left} : \text{right}])$
- counter - количество различных подмассивов, в которых все элементы различны, в промежутке $a[: \text{right}]$.
- все элементы $a[\text{left} : \text{right}]$ - различны.
- $\nexists \text{left}' : \text{left}' < \text{left}$, все элементы $a[\text{left}' : \text{right}]$ - различны

Изначально $\text{left} = 1$, $\text{right} = 2$, $\text{counter} = 1$, $\text{currentnumbers} = \text{HashSet}([a[1]])$.

Если элемент right не встречался в currentnumbers, то увеличим right на 1 и доабвим этот элемент к текущему множеству.

Иначе уберем крайний левый элемент из текущих.

Из построения легко видеть, что все перечисленные свойства выполняются (кроме counter).

Заметим, что при увеличении right на 1, количество непрерывных подмассивов с различными числами, увеличивается на $\text{right} - \text{left} = \text{len}(\text{currentnumbers})$. Так как добавляются

подмассивы $a[\text{right} : \text{right} + 1]$, $a[\text{right} - 1 : \text{right} + 1]$, ..., $a[\text{left} : \text{right} + 1]$.

Следовательно для поддержания инварианта $\text{counter} += \text{len}(\text{currentnumbers})$. Таким образом мы посчитали все подмассивы состоящие из различных чисел, которые в какой то момент находились внутри $a[\text{left}, \text{right}]$. Но это и есть все возможные подмассивы с различными числами, так как если предположить противное, то в момент когда right находилась в правом крае подобного массива, а left по предположению была правее левого края подобного массива, то это противоречит последнему инварианту.

Так как на каждой итерации увеличивается left или right , а каждая итерация стоит $O(1)$, то итоговая сложность $O(n)$.

Псевдокод приведен ниже.

```
left = 1
right = 2
counter = 1
currentnumbers = HashSet([a[1]])
while left  $\neq$  n + 1 or right  $\neq$  n + 1 do
    if right  $\neq$  n + 1 and a[right] not in currentnumbers then
        right += 1
        currentnumbers.add(a[right])
        counter += len(currentnumbers)
    else
        left += 1
        currentnumbers.remove(a[left - 1])
    end
end
```

Task 7

1

Будем осуществлять поиск *лесенкой* начиная с левого верхнего угла.