Tutorium #2

04/10/2022

An **ugly number** is a positive integer whose prime factors are limited to `2`, `3`, and `5`.

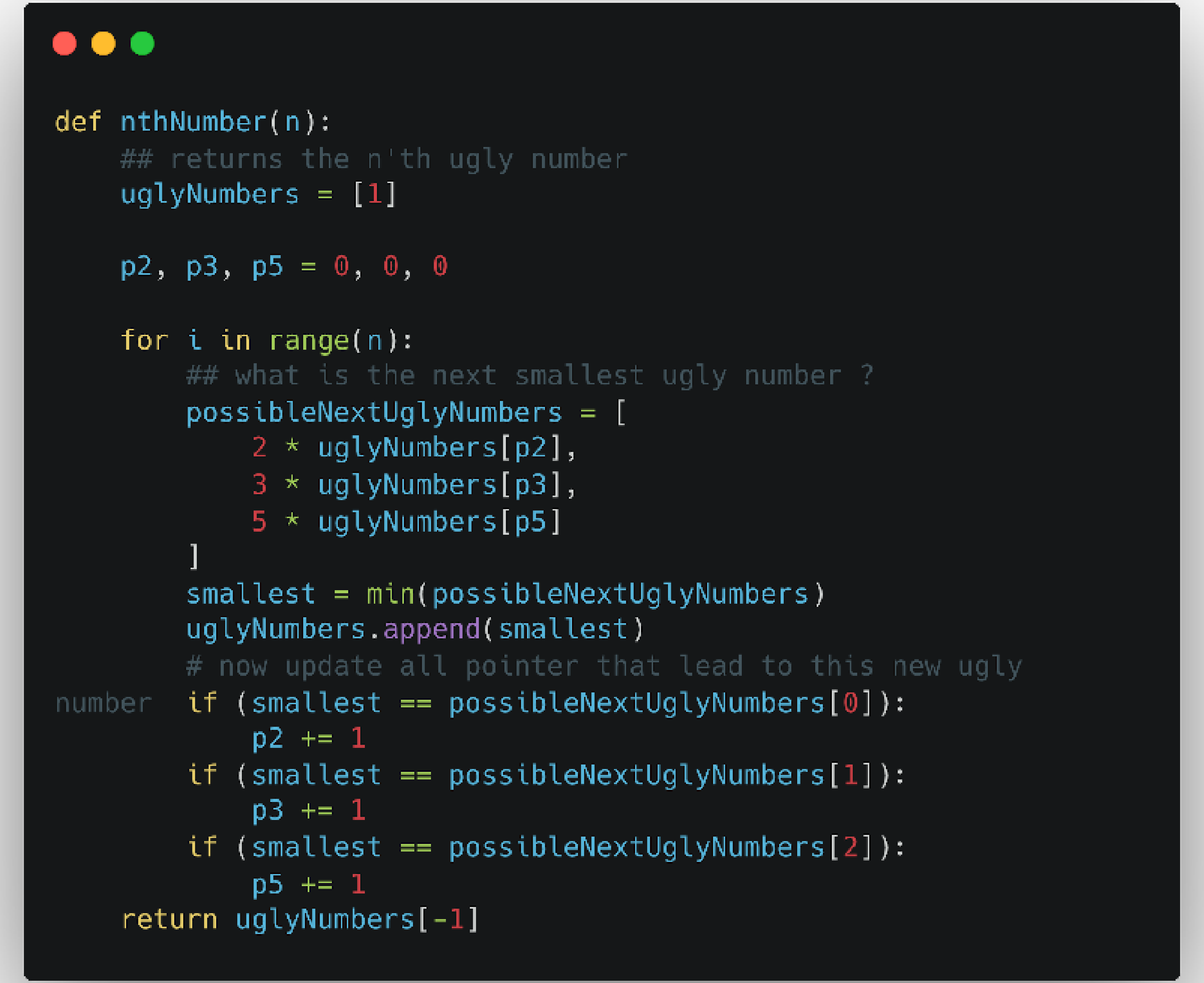
Given an integer `n`, return *the* n^{th} ***ugly number***.

Example 1:

```
Input: n = 10
Output: 12
Explanation: [1, 2, 3, 4, 5, 6, 8, 9, 10, 12] is the sequence of the first
10 ugly numbers.
```

Example 2:

```
Input: n = 1
Output: 1
Explanation: 1 has no prime factors, therefore all of its prime factors are
limited to 2, 3, and 5.
```



```
def nthNumber(n):
    ## returns the n'th ugly number
    uglyNumbers = [1]

    p2, p3, p5 = 0, 0, 0

    for i in range(n):
        ## what is the next smallest ugly number ?
        possibleNextUglyNumbers = [
            2 * uglyNumbers[p2],
            3 * uglyNumbers[p3],
            5 * uglyNumbers[p5]
        ]
        smallest = min(possibleNextUglyNumbers)
        uglyNumbers.append(smallest)
        # now update all pointer that lead to this new ugly
number  if (smallest == possibleNextUglyNumbers[0]):
            p2 += 1
        if (smallest == possibleNextUglyNumbers[1]):
            p3 += 1
        if (smallest == possibleNextUglyNumbers[2]):
            p5 += 1
    return uglyNumbers[-1]
```

Problem 1 (2 points)

1. Give the important differences between a normal *priority queue* and an addressable *priority queue*.
2. Compare the running time of a `merge` operation for *pairing heaps* and *binary heaps*.

1. in contrast to a normal *priority queue*, the addressable *priority queue* allows direct access to arbitrary elements via a *handle h*. This is returned by `insert` operation as a return value. Moreover, it enables additional operations: `remove(h)`, `decreaseKey(h,k)`. The `merge` operation can in principle also be provided by a normal *Priority Queues*, however, it can only do it less efficiently.

2. In a pairing heap *pairing heap* a set of trees is maintained. A **`merge`** operation is hence possible in constant time, by concatenating the corresponding lists and updating the **`minPtr`** to the minimum of both forests. In a *binary heap* this does not work. In the most widely used array implementation there are multiple possibilities. If the number of elements is given by n_1 and n_2, $n_1 < n_2$, then we get the following running times:

   a) inserting the smaller set of elements: $O(n_1 \cdot \log(n_1 + n_2))$

   b) complete rebuild: $O(n_1 + n_2)$

   Depending on the distribution of the elements, one or the other one can be reasonable.

Problem 2 (4 points)

1. Prove the general lower bound for addressable *priority queues* of $\Omega(\log n)$ for `deleteMin` under the condition that `insert` runs in constant time.
2. Why does this bound not have to hold if `insert` is allowed to consume more time?

1. With a running time of $O(f(n))$ of `deleteMin` you can do comparison-based sorting in time $O(nf(n)) + n \cdot O(1)$, by first inserting all elements and then calling `deleteMin` n times. For `deleteMin` in sublogarithmic time, that would be a contradiction to the known lower bound for comparison-based sorting $\Omega(n \log n)$.
2. `insert` could after each call store a sorted list. With this list you can answer `min`- and `deleteMin`-operations in constant time.

1. The ingredient that should be shown on the display is the one with the least remaining supply. Additionally, we should be able to update the supply of all ingredients while the system is running. The ingredients only get reduced over time. The classic data structure here is a addressable *heap*.

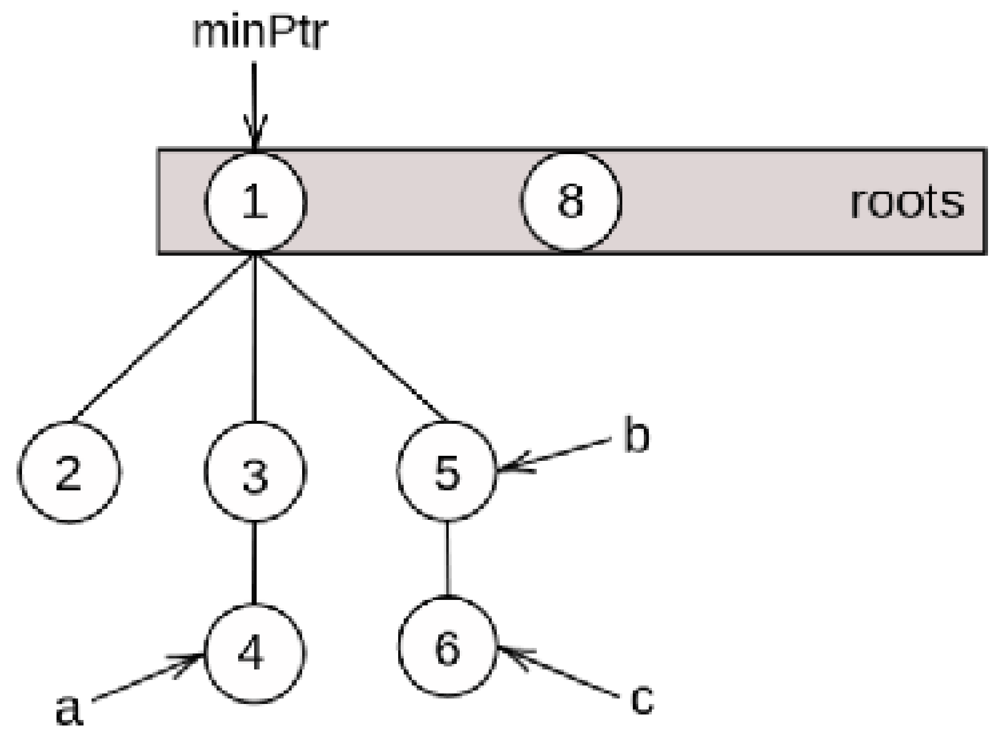
2. The function `MixDrink`, only has the task to keep the remaining ingredients up to date.

```
function MixDrink($r$ : Recipe, $q$ : Queue, $d$ : Display)
    for ingredient $i \in r$ do
        $q$.decreaseKey($i$, amount($i$))
    end for
    $d$.show($q$.min(), amount($q$.min()))
end function
```

3. When exchanging a storage box, we have to remove the element from the heap and insert it with full storage box value afterwards. It is important that not necessarily we exchange the minimal element. Hence, a sequence of operations should be done. First, the key of the ingredient should be set to $-\infty$. Afterwards, we can do a `deleteMin` and an `insert` afterwards.

Problem 4 (4 points)

The definition of a *pairing heap* does not state which elements are neighbors. For the solution of the next tasks we assume a sorted list of roots (by insertion order) and neighborhood induced by this. Given is a *Pairing Heap* in the following state.

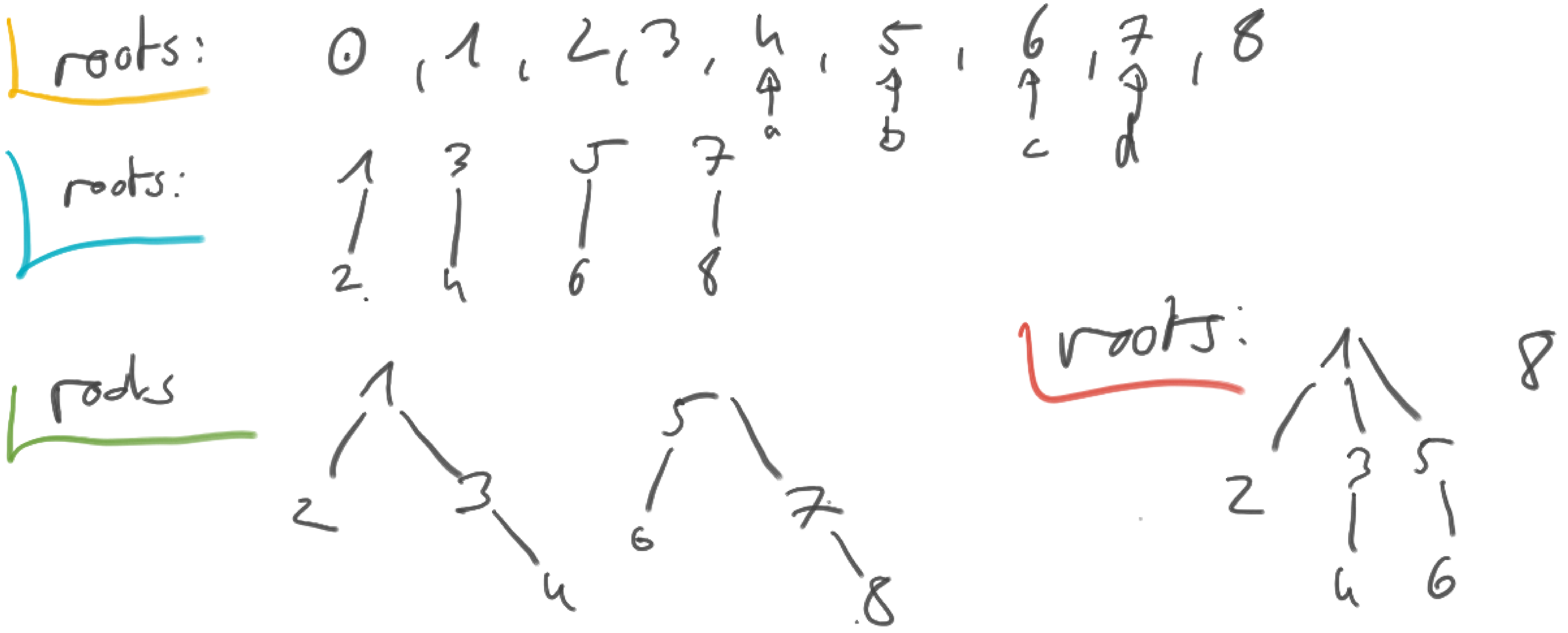


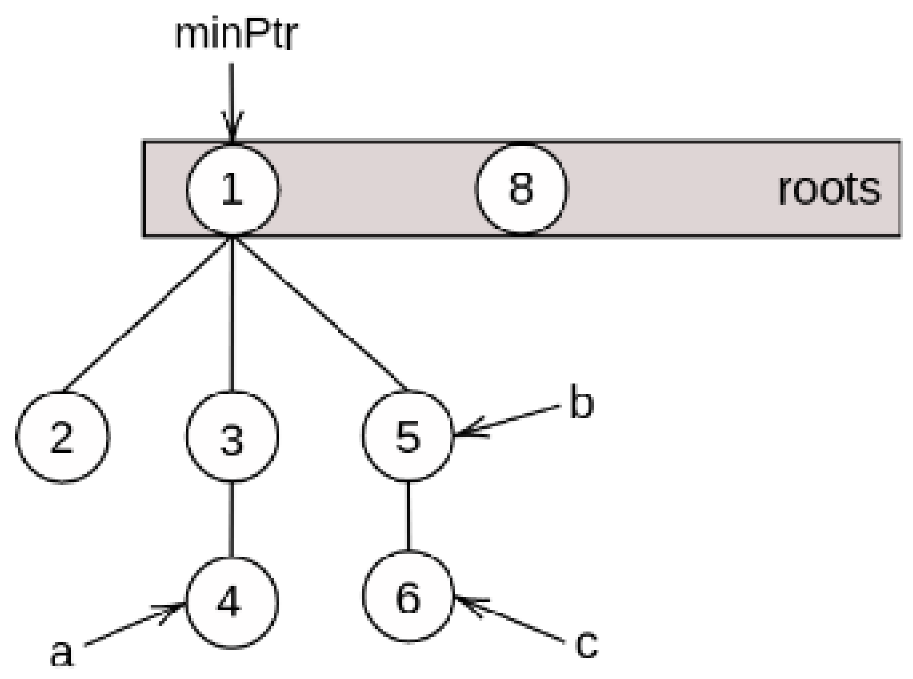
1. Give a very short sequence of operations that creates this state.
2. Execute the following operations step by step on the given heap and draw the intermediate state of the heap after each operation:
   - `deleteMin()`
   - `insert(9)`
   - `decreaseKey(a,1)`
   - `remove(b)`

1. The sequence

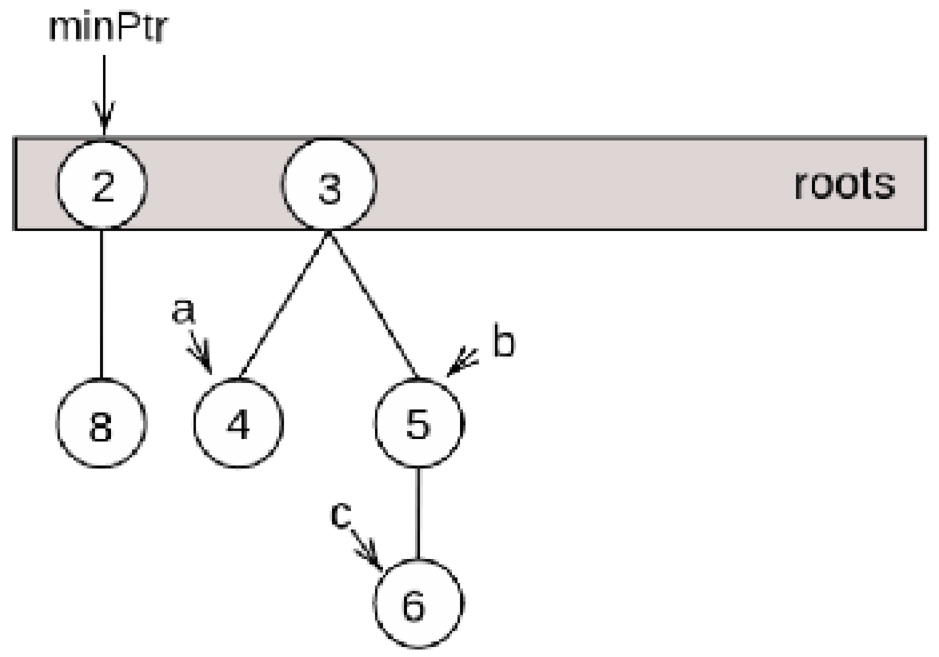
`insert(0), insert(1), insert(2), insert(3), a:=insert(4), b:=insert(5), c:=insert(6), d:=insert(7), insert(8), deleteMin(), insert(0),deleteMin(), decreaseKey(d,0), deleteMin()`

creates the given state.

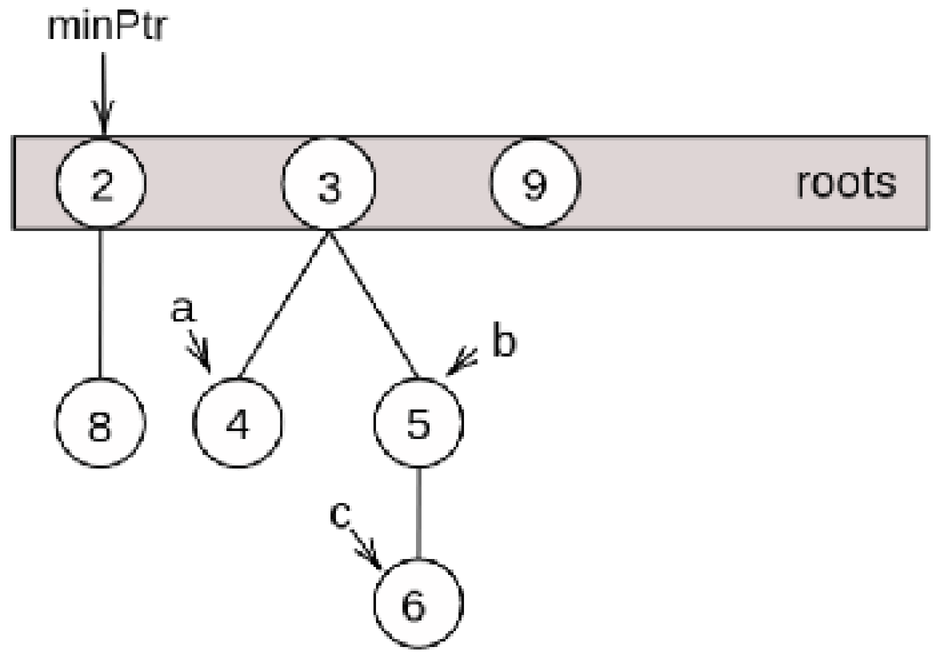




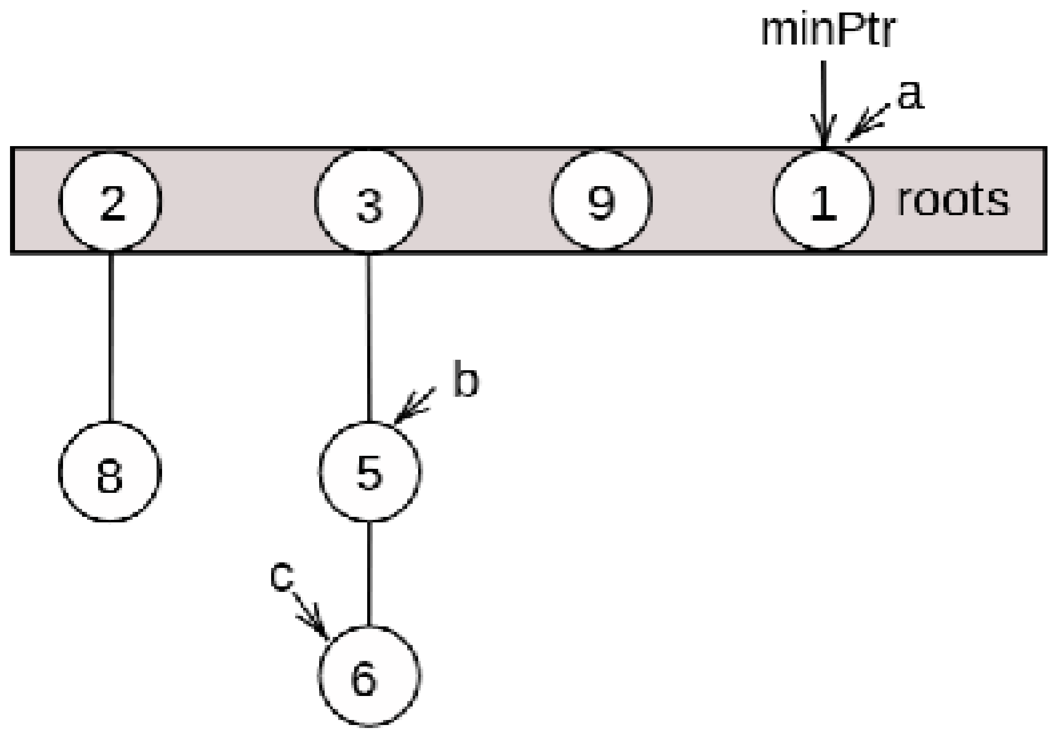
`deleteMin()`:



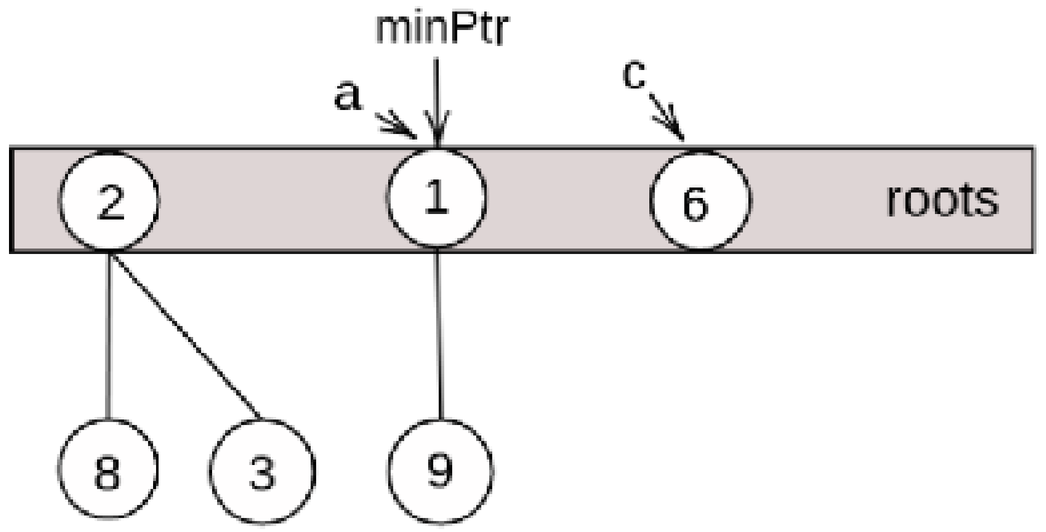
`insert(9)`:



`decreaseKey(a,1):`



`remove(b):(decreaseKey(b,0) + deleteMin())`


minPtr
a
c
2
1
6
roots
8
3
9