Ex.2

ALGO:

Initialize self-balancing BST
with intervals of first item x
INSERT $((-\infty, x), (x, +\infty))$
Initialize hashtable for interval_to_parent association
DICT $[(-\infty, x)] = x$
DICT $[(x, +\infty)] = x$

output = []

for z in sequence :
  (l,r) = find_interval (BST, z) O(log N)
  BST. delete ((l,r)) O(log N)
  BST. insert ((l,z)) DICT[(l,z)] = z O(log N)
  BST. insert ((z,r)) DICT [(z,r)] = z O(log N)
  output. append (DICT.pop((l,r))) O(1)

O(log N) } O(log N) } O(N log N)

return output

```
def find_interval (BST, x):
    curr_int = BST.root
    while True:
        if x in curr_int:
            return curr_int
        elif x > curr_int[1]:
            curr_int = curr_int.right
        else:
            curr_int = curr_int.left
```

Proof of correctness:

To give a brief explanation on why the algorithm as described works correctly, we start by recalling that each newly added node is a leaf at the time of insertion. This information is key since it tells us that at every new insertion a previously available leaf spot in the BST is filled with the inserted value, this translates into a new leaf that, while removing one former leaf spot, creates two new potential leaf spots for future insertions. How are these “slots” defined and described? By intervals in which the new element in the BST may belong to. To give an example, let’s start by assuming 6 is the first element to insert in the BST. Since the tree is initially empty, we insert 6 as the root. This leads to the creation of two potential leaf spots in the self-balancing BST corresponding to candidates < 6 in the left child and > 6 in the right child. Now, let us say we want to insert 4 to the BST. Then, what will happen is that since $4 < 6$, 4 will become the left child of the root 6 and will output 6 as its parent node. At the same time, the insertion of 4 created two intervals in which a next value could belong to, namely (-inf, 4) and (4, 6), that will be added to the still available slot as the root’s right-child corresponding to (6, +inf). As we can already notice, each time a node x is inserted, the interval in which it belonged gets split into two in corrispondence of x. Hence, we can keep track of the extremum x every time there is a split due to its insertion, and we know that eventually, when a new element will belong to one of its intervals, it will take the slot corresponding to said interval and have x as its parent node.