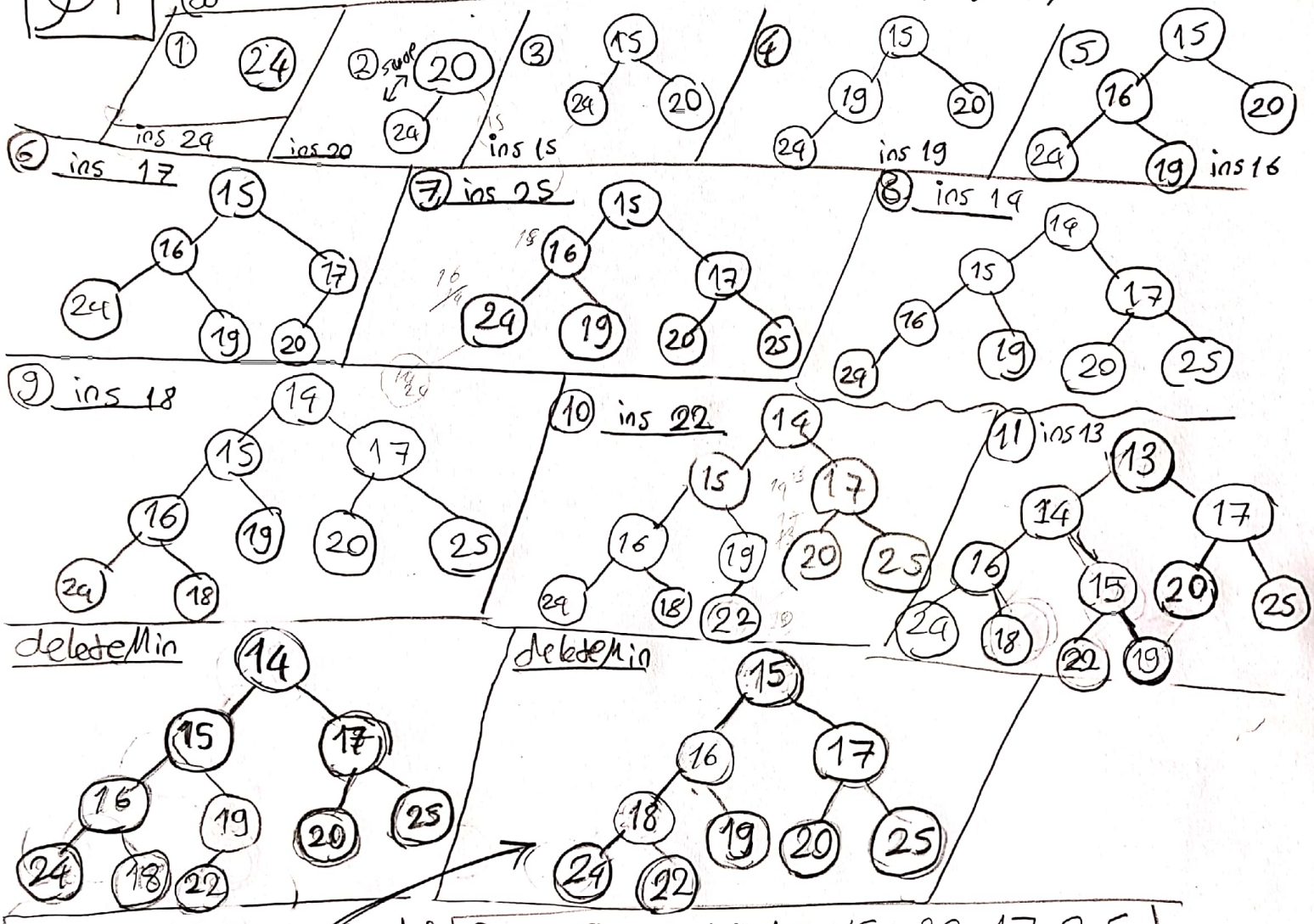
N. Onur Vurol - 21902330 - CS202 HW3

Q1

a) Min heap insert: 24, 20, 15, 19, 16, 17, 25, 14, 18, 22, 13 / delete twice



b) Inorder traversel: 24 18 22 16 19 15 20 17 25

It can be observed that the traversel does not give sorted outputs. This is mostly because that as opposed to a BST there is not a direct relation between left & right children's item. In BST, left child is always smaller than right however in a min-heap the insertion order matters and therefore a left child can be bigger than right or vice-versa. What matters is that both children's values have to be larger than Parent's value.

Preorder traversel: 15 16 18 24 22 19 17 20 25

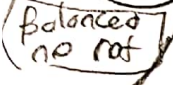
It is not sorted again due to heap Property (left & right children do not have a direct relation with each other, insertion order matters, only they have to include data larger than their parent, that's all!

Postorder traversel: 24 22 18 19 16 20 25 17 15

Same here!

Q1 / C) insert 29, 22, 19, 23, 18, 20, 27, 16, 15, 17 to AVL respectively!

1) insert 29



Balanced no rot

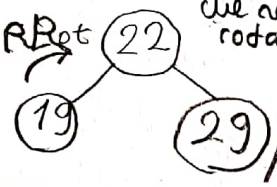
2) insert 22

Balanced no rot

3) insert 19

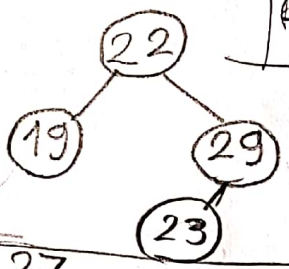
To balance we need rotation!

SRRot



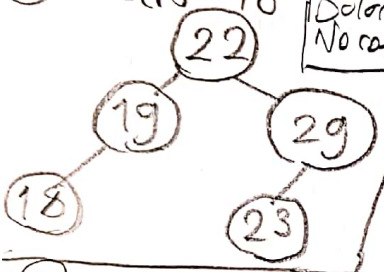
4) insert 23

Balanced No rot



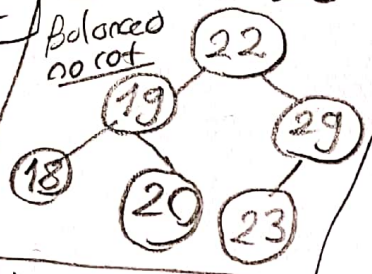
5) insert 18

Balanced No rot

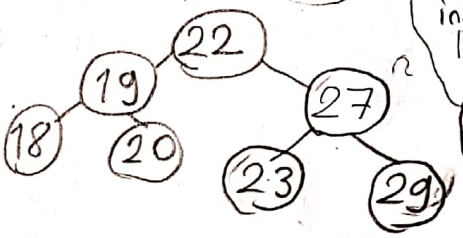


6) insert 20

Balanced no rot



7) insert 27



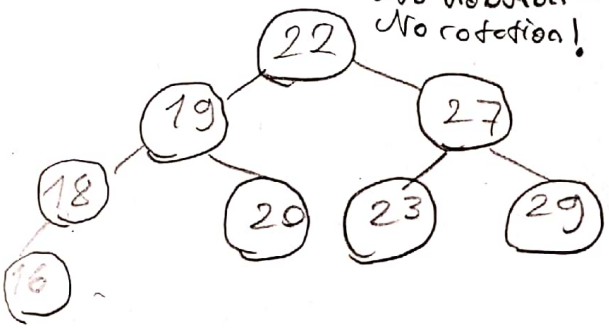
inserting n's left child's right subtree

DLR Rot

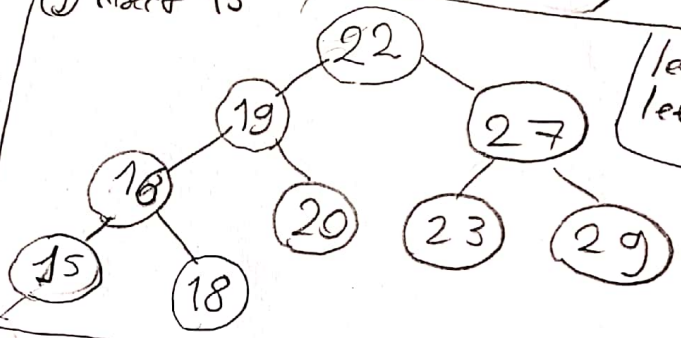
1. around 23
2. around 29

8) insert 16

No violation - No rotation!

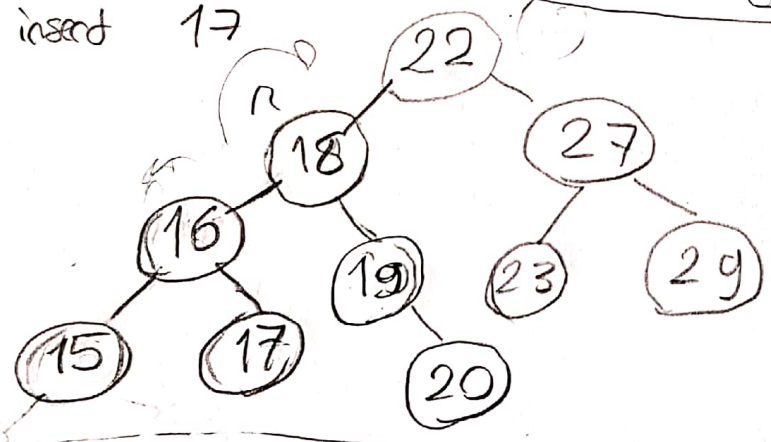


9) insert 15



left child's left subtree SRRot

10) insert 17



n's left child's right subtree DLRRot

- inserting n's left child's left subtree: Single Right Rotation (SRRot)
- inserting n's right child's right subtree: Single Left Rotation (SLRot)
- inserting n's right child's left subtree: Double RightLeft Rotation (DRLRot)
- inserting n's left child's right subtree: Double LeftRight Rotation (DLRRot)

N.Onur Vural 21902330 - CS202 HW3

Question 3

In the implementation of Question 3, in order to find the proper minimum printer number, the printer number started from one and increased until the printers were able to sustain the given average time condition. This may be acceptable for small inputs, but when it comes to handle large print requests by checking the proper printer number one by one, the process turns out to be not much efficient. In a very large library with many potential printers (N), instead of increasing the printer number until $K \leq N$, a special search algorithm may turn out to be much efficient. As in parallel to the binary search algorithm, after finding the average time result with one printer, the middle value in the range of N can be computed, given that N value is known, and afterwards the algorithm may move on checking the middle value in a recursive fashion instead of iteratively checking one by one. This will result in a decrease in the search time from O(n) to O(logn) and therefore decrease the total time in search part to find the optimum number of printers remarkably. But at this point it has to be mentioned that if the N value is unknown, which is most likely the case, the algorithm cannot do binary search between 1 and N, but it can check the printer number by using the powers of two (as 1,2,4,8,16,32,64,128…). If two consecutive printer numbers produce a situation where the upper range gives an acceptable average time and lower range does not, than the binary search algorithm can be applied in this range. For example if printer number 128 gives an acceptable average time (less than average time) value and 64 gives a greater average time than desired, algorithm will apply binary search between 64, 128. This will also contribute to a decrease from O(n) search time to O(logn) and together with simulator's time, the total time to find optimum printer number will be lower.