1a-) 0 mod 19 = 0
19 mod 19 = 0

The key 0 and 19 had the same hash, meaning they would collide.

b-) h(19) = 0 h(82) = 6
h(2) = 2 h(71) = 14
h(5) = 5 h(55) = 17
h(17) = 17 h(213) = 4
h(107) = 12 h(123) = 9
h(13) = 13

Will occur one collision at keys 17 and 55.

c-) Will occur one collision at keys 17 and 55.

d-) i: We first delele the key 123 from the index 9 and also the key 55 from index 18, than we inse the key 37, to the index 18 using the hash function (h = key mod 19).

ii: We delete the key 123 from index 9, after we do the hash function to insert the key 37, h = 37 mod 19 = 18, since the index 18 is already taken, we use the linear probing method to find an empty index for the key, getting the index 19 and inserting the key 37 in it, afte it we delete 55 from index 18.

e-) $h_0(K) = (0 + K) \bmod T$
$h_1(K) = (h_0 + K) \bmod T$
$* \; h_i(K) = (h_{i-1}(K) + K) \bmod T$

/ /

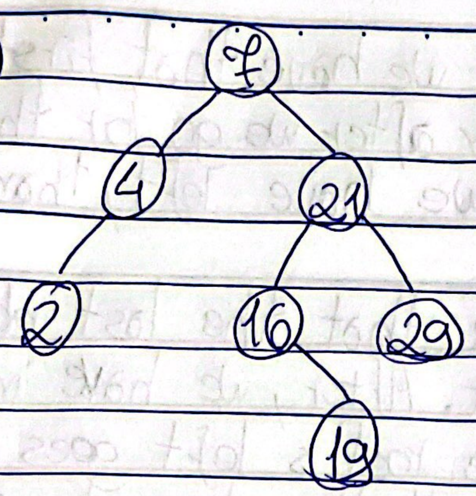
```
2-) def check_permutation(A, B):
        n = len(A)
        arr = [None] * n

        for Key in A:
            if arr[Key % n] == None:
                arr[Key % n] = Key
O(n)        else:
                i = 1
                while arr[(Key%n)+i] == None:        Linear
                    i += 1                           Probing
                arr[(Key%n)+i] = Key
        for Key in B:
            if arr[Key % n] == Key:
                head = arr[Key % n]
                head.next = Key
O(n)        else:
                i = 1
                while arr[(Key%n)+i] != Key:
                    i += 1
                [illegible]
                head = arr[(Key%n)+i]
                head.next = Key
        for node in arr:
            if node.head != node.next:
O(n)            return False
            else:
                return True
```

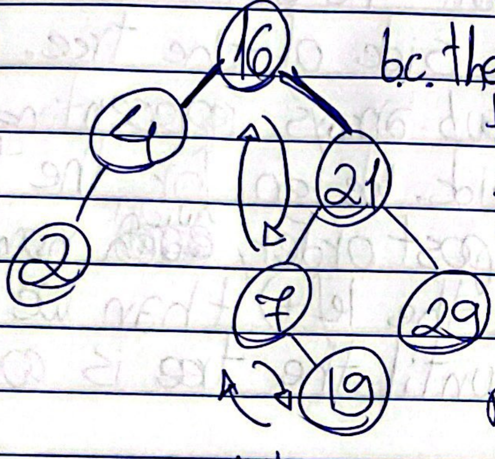
$T(n) = O(n) + O(n) + O(n) = O(3n) = O(n)$

3 a-)

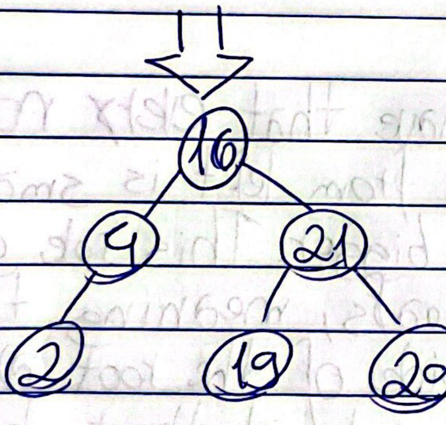
h=3



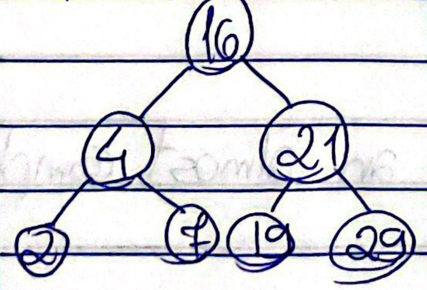
b-)



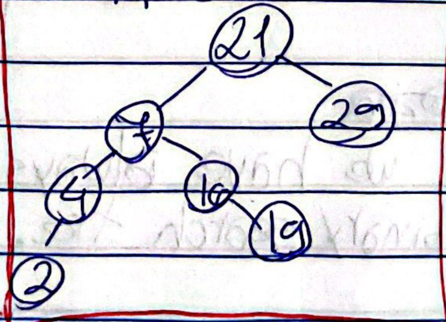
We first check how many childrens it ha
b.c. there're two, we change places between the root and his percursor. than we check if the old root has now childrens, since there's only one we change places and the pointer to 7 is now lost.



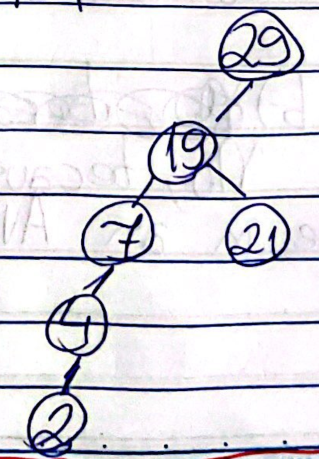
c-) h=2



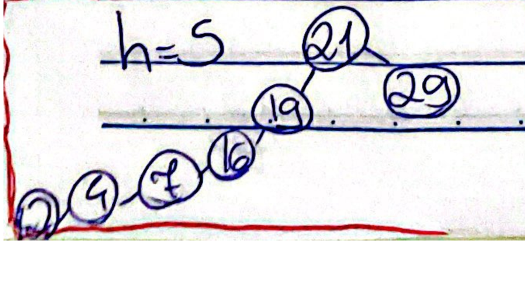
h=3



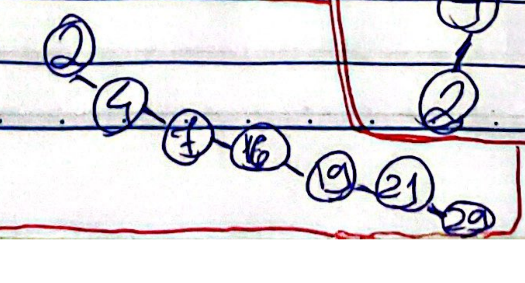
h=4



h=5



h=6



/ /

4.) In a post order we have that first we go left, than right and only after we go for the root.
In an in order we have left, than root and than right.
This way, we have that the last element of post order is the root. After, we have in the inorder that every element at the root's left goes to the left side of the tree, and all the elements at root's right goes to the right side of the tree.
After we have two sub-arrays representing the left side and the right side. We go for the left sub array, we search in the post order, ~~[illegible]~~ wich number shows first (from right to the left) than we put in the root and keep doing until the tree is complete on both sides.

5 a-) Yes, because we have that every node keeps the rule that every children from left is smaller and every children from right is bigger. This rule goes for all elements from root to leafs, meaning that all the elements at the left side of the root are smaller than the root and all elements at the right side of the root are bigger than the root.

b-) ~~No, because in~~
No, because we have always an almost complete tree in an AVL binary search tree.

tilibra