1 –

In a Stack data structure, we have that the order to remove each element is the opposite order that they got insert. This is also known as FILO (first in, last out).

In a Queue data structure, we have that the order to remove each element is the same order that they got inserted. Also known as FIFO (first in, first out).

To get a data structure that behaves as Queue, by using two stacks is pretty simple. Considering that the Stack is as FILO inserting/removing data structure, what we have to do is insert all the elements in one stack and after it, remove one by one and inserting them in the second Stack, this way the first element getting in the first stack will be the last element getting in the second stack, therefore the second stack would work as a Queue for the initial input.

T1(n) = inserting all elements in stack one -> O(n)

T2(n) = removing from stack one and inserting to second stack -> O(n)

T3(n) for the whole function is O(n).

2 –

As said before, the Stack uses a FIFO method, meaning the data structure has the same and only “entrance and exit”. This way, we can look at a list with 2 possible entrance and exit, the beginning (index 0) and the end (index size of list -1), so we can actually use them each as an exclusive entrance and exit for each Stack.

The first stack would be from index 0 until index size/2, while the second stack would be from index size/2 to index n.

The first stack would do push and pop at index 0, while the second stack would do push and pop at index n.

3 –

Not necessarily. In a BST we have that this behavior (left child smaller than root and right child bigger than root) keeps constant to every sub-tree of a node.

Ex: We have a node with value 5, and his right child has a value of 10. For now this might be a BST, but if the node with value 10, has a left child with value 4 (or any value less than 5), it will go against the BST structure but still be a binary tree.

4 –

1, 2, 3 – It’s not possible to rebuild an unique tree with one traversal only because two different tree might have the same traversal. But it’s still possible to build a new binary tree with any given traversal.

4 – Yes it’s possible. In an in-order we can obtain the leftmost and rightmost nodes, and in a pre-order we can obtain the root, this is the only way to build an unique binary tree. Since we have in the pre-order that the root is the first element, and in the in-order we have all the element at root’s left sub-tree locating from index 0 until i-1 (i is the index the root is in the in-order traversal) and root’s right sub-tree at index > i.

5 –

Since we have that the root is the first element in a preorder traversal, knowing the root we can look for the first element that is bigger than the root (let’s say at index i). After finding it, we have that every element from the root to index i is located in the left sub-tree, and from index i (including) until the root, we have all the elements in the right sub-tree. We again have that the first element of each sub-array is the root for the sub-tree, so it’s possible to do it recursively at every sub-tree and placing the node corresponding.

6 –

The successor function might have a O(h), with h being the height of a tree. But if we want to go through all the elements of a tree, the only possible way to do this is to actually go to every and each node, meaning the time complexity for the role process will be O(n), with n being the number of nodes of a tree. O(O(logn)\*n) = O(n).

7 –

1 – Teta(nlog), with logn being the height of the tree (the height in worst and best case for AVL is logN) \* n times.

2 – O(n), we do n times insert.

8 –

def sorted\_to\_avl(lst):

m = len(lst)//2

root = lst[m]

while m in range(0, len(lst)):

root.left = sorted\_to\_avl(lst[:m])

root.right = sorted\_to\_avl(lst[m+1:])

return root

9 –

An option would be to take both in-order traversals, merge them into one sorted array, and build an AVL tree from it using sorted\_to\_avl function.

10 –

Create a new and empty array with size n.

i from 0 to n:

we check the first element of each array and pick the smallest and append it to the output array and remove it from the original array

12 –

We can build a binary search tree from the array, and check only the sub-trees that the root has at least one children and that the root’s value is not bigger than 11, checking if any combination is equal to 11. (root + root.l = 11, root + root.r = 11, root.l + root.r = 11)

13 –

Yes, an ordered array will always represent a Heap, since the Heap D.S. behaves as a preference data structure, in a sorted array we always have the first and last element as two possible preferences for a MinHeap or a MaxHeap. (smallest or biggest element of the array/heap)

Not necessarily. The Heap has always a preference, and it’s either the biggest or the smallest element that is always located at the head of the heap (or the first element of the array), but the heap structure doesn’t keep a sorted structure for all the elements as in a sorted array.

14 –

Class S:

Def \_\_init\_\_(self, lst):

Self.lst = sort(lst)[::-1]

Self.max = self.lst[0]

Self.size = len(self.lst)

Def insert(self, other):

For n in self.lst:

If other > n:

Indx = self.lst.index(n)

Self.lst.insert(indx, other)

Break

Def delete(self, other):

for i in range(len(self.lst)-1):

if self.lst[i] == other:

self.lst.pop(i)

break

def get\_size(self):

return self.size

Def get\_max(self):

Return self.max

Def get\_min(self):

Return self.lst[self.size - 1]

15 –

Yes it’s possible. The class S works as a Heap Data Structure that has both access to the minimum value and the maximum value of the heap, meaning we can have both MaxHeap and MinHeap at the same structure.

16 –

It’s possible to use a type of comparison sorting like insertion sort or bubble sort, that search in the array k times for the smallest element. This would take a time complexity of O(k\*n). For the worst case k = n so would be O(n^2) but average case is O(n).

17 –

One option is to use an algorithm that works as a counting sort, but instead of returning a sorted array, we check in the index\_array the element equal to 5, than return the index (the index will be the element of the original list).

First – find the max value of the arr - > O(n)

Second – use the counting sort technique to count the quantity of each different element in the array, but while doing the counting for every element, we always check the value of the new update, if its equal to five we then return the number. -> O(n)

18 –

Finding the average value of the array -> O(n)

After finding the avg, we do a linear search to find index i of the first element bigger than the avg, with the index of this element we can return all the elements from i - (k//2) to index i + (k//2) not including the element if it’s equal to the average and respecting the index limits of the array. -> O(n)

T(n) for the whole function would be O(n).

1 – ג(x) - ב is the correct answer -> the avg is O(1) but the worst case is O(n) but won’t happen enough times to be always O(n)

2 – א correct

3 – ב correct

4 – ד correct

5 – ב, ד, ה, ו Any immutable object can be stored as a key in a hash table

6 – א, ב, ג, ד, ה, ו Any object can be stored as a value in as hash table

7 – א correct

8 – ב , ג, ד(x) -> ד is not right -> not right to use Sets and not Lists to store values in a hash table

9 – ד(x) – ג is the correct answer -> for a hash table there’s the hash function that might take time and be less useful than a small List

10 – ד correct

11 – א(x) – ב is the correct answer -> What differs Queue and Stack are their interface (methods and functions)

12 – ב, ד, (x) – also ה is the correct answer -> Pop, Push and Peck

13 – א, ג correct -> Enqueue, Dequeue