DATA STRUCTURS, Ex06

Aviad Hahami

302188347

Q1.

*note – This algorithm is correct assuming 2 nodes with equal value will be treated as “bigger than” (i.e. if we have two “8”, then one will be the sub-tree root, and the other will be the right child)

```
main(Pointer p,BST t){

	// Create two pointers
	Pointer q,parent;

	// Check if the given pointer points to tree root, if so
	// we return null as no parent exists
	if (p == t.root){
		return null;
	}

	// Instansiate q to the root, and parent to the root too.
	// parent is instanciated to the root because we return it
	// and we don't want to return un-initialized variable
	q = t.root;
	parent = t.root;

	// We terminate loop if we hit an empty node.
	// This should'nt occour as p is adressed as a legitimate pointer
	while (q != null) do{

		// If p == q we return parent as it points to
		// q's & p's parent
		if (p == q){
			return parent;
		}

		// If we got here, it means we still need to dig further
		parent = q;

		// Regular BST dive
		if (p.value >= q.value){
			q = q.right;
		}else{
			q = q.left;
		}
	}
	return parent;
}
```

Q2.

```
ReverseInOrder(TreeNode<T> n){
    if( n == null ){
        return;
    }
    ReverseInOrder(n.getRight());
    print(n.key);
    ReverseInOrder(n.getLeft());
}
```

Q3.

True.

It is known that in *post-order traversal* the current sub-tree's root will be printed last, hence we can say that if we have the numbers printed in increasing order, the root of all sub-trees will be printed last, and this implies the whole tree itself.

Q4.

False.

We will contradict using the formula to calculate the amount of nodes in a tree:

$$General\ formula\ for\ height\ h\ \rightarrow\ n(h) = n(h-1) + n(h-2)\ +\ 1$$

$$n(4) = n(3) + n(2) + 1$$

$$I)\ n(3) = n(2) + n(1) + 1$$

$$II) n(2) = n(1) + n(0) + 1$$

$$n(0) = 1, \mathrm{n}(1)\ =\ 2, \mathrm{n}(2)\ = 4$$

$$hence, we\ got \rightarrow n(4) = 7 + 4 + 1 = 12$$

And we can say that there is no AVL tree, where h=4 and n=11

Q5.

<u>Tree A</u> – This tree is not a BST since we have “2” on the tree’s RHS, where the root is 6. Since the tree is not a BST, it is not an AVL (AVL is a BST special case) hence no rotation will fix it and it’s a lost case. RIP.

<u>Tree B</u> – This tree is a valid AVL hence a valid BST. Legendary.

<u>Tree C</u> – This tree is a valid BST yet not valid AVL.

> We can see that “6” BF (balance factor) is 2, while he has no right child but has a grand-child on the left.
>
> We can perform a rotation over nodes “6” and “5” in order to restore the height property.

<u>Tree D</u> – This tree is a valid BST yet not valid AVL.

> Like Tree C, we differ in BF by more than one, hence we violet the AVL property.
>
> In order to achieve that we can perform two rotations on the tree’s LHS (4-1-3) and one rotation on the RHS (9-8-7). Performing these three actions will yield valid AVL tree.

Q6.

```

//IMPORTANT:
// This is psuedo-code.
// Expections are not handeled (i.e index out of range)
GenerateTreeFromArray(Array A){
    if (A.length == 0){
        return;
    }

    // Find the array's center
    localRoot = a[(a.length -1)/2];

    // Divid the array into two chunks
    leftA = a from 0 to localRoot;
    rightA = a from localRoot+1 to length-1;

    // Deploy the sentinels to the left
    leftChild = GenerateTreeFromArray(leftA);

    // Deploy the sentinels the the right
    rightChild - GenerateTreeFromArray(rightA);
}
```

We “touch” each array element with O(1) and we perform that action n times, hence we performed the BST construction using O(n) runtime complexity.

Q7.

A.

<u>Let person X be as follows</u>:

$$X = \{\, name, birth, death, expertise \,\}$$

<u>Description</u>:

We will use an augmented interval tree as described by *Cormen*.

The tree will be constructed from special nodes.

Each node will contain a person X, the person’s lifespan as the interval and the maximum high value among the tree. Below there’s an example of a node in the tree:

X (Person data)

Life span = [X.death, X.birth]

Max high value = Maximum value of birth in sub-tree

Construct tree	--	O(n log n)
Insert a person X into the data structure	1. Create a new tree node as formatted above. 2. Insert to tree according to regular interval tree properties.	O(log n)
Remove a person X from the data structure.	1. Find person X in tree 2. Remove from tree according to regular interval tree properties	O(log n)
Given a new person, X, find at least one name of a person Y that lived in the same period.	1. Search the tree for a corresponding interval according to regular interval tree properties.	O(log n)
Given a person, X, return another person, Y, that was	Search the tree for a corresponding interval according	O(log n)

born in the same month and year as X, if one exits.	to regular interval tree properties.	
Give the name of the person that was born first	Since the tree is sorted by lowest key (according to *Cormen*'s specification) we should return the name of the left most child.	O(log *n*)
Give the name of the person that passed away last	1. Return name of the person in the most-right node.	O(log *n*)

B.

In order to insert into the tree, we will do the following:

Add a field to each node in the following format:

(4 digits for year, 2 for day and month)

Y	Y	Y	Y	M	M	D	D
2	0	1	5	0	5	2	6

Insert the nodes into temporary array, and sort them via *Radix* (O(n) + O(8n) ~ O(n))

Now, since we know we are dealing with a tree graph, we can use post-order traversal (DFS) in order to insert the nodes to the tree.

The DFS is blocked by O(m) where m is the edges in the graph.

Edges in tree graph are known to be n-1, hence we have

$$O(m) \sim O(n-1) \sim O(n)$$

So we've concluded this in worst-case time complexity of O(n).