TUTORIAL SESSION 4

BINARY TREE

Question 1.

a. Given a list of integers as follows, insert those integers into an empty BST one-by-one. Suppose that the numbers will be added to the tree in the same order as that of the list.

{59, 17, 4, 13, 72, 91, 87, 21, 33, 60, 71, 1, 19}

b. Randomly re-order the list in Question 1a as follows. Please generate the BST again (from an empty BST). (The purpose is to observe the input-order-sensitive of the BST.)

{71, 1, 4, 13, 87, 91, 72, 33, 21, 60, 59, 17, 19}

c. Redraw the BST after deleting the node 60 from the BST in Question 1a.

d. Redraw the BST after deleting the node 71 from the BST in Question 1b.

Question 2.

a. Given a list of integers as follows, generate the corresponding AVL tree by inserting numbers in the list one-by-one from an empty AVL.

{71, 1, 4, 13, 87, 91, 72, 33, 19, 60, 59, 21, 17}

b. Re-do question 2a with the list as follows.

{1, 4, 71, 13, 87, 33, 72, 91, 19, 60, 59, 17, 21}

Question 3. Given the following algorithm of balancing the tree as presented in Listing 1.

```
algorithm recursive_Insert (ref subroot <pointer>,  val DataIn <DataType>, ref taller <boolean>)
Inserts a new node into an AVL tree.

Pre  subroot points to the root of a tree/ subtree.
     DataIn contains data to be inserted into the subtree.

Post  If the key of DataIn already belongs to the subtree, duplicate_error is returned.
       Otherwise, DataIn is inserted into the subtree in such a way that the properties of an AVL
       tree are preserved.
       If the subtree is increased in height, the parameter taller is set to TRUE; otherwise it is
       set to FALSE.

Return duplicate_error or success.
Uses  recursive_Insert, left_balance, right_balance functions.

1.  result = success

2.  if (subroot is NULL)
       1.  Allocate subroot
       2.  subroot ->data = DataIn
       3.  taller = TRUE

3.  else if (DataIn = subroot ->data)
```
1. result = duplicate_error
2. taller = FALSE

4. else if (DataIn < subroot->data) // Insert in left subtree
   1. result = recursive_Insert(subroot->left, DataIn, taller_temp )
   2. if (taller_temp = TRUE)
      1. if (balance of subroot = left_higher)
         1. left_balance (subroot)
         2. taller = FALSE
      // Rebalancing always shortens the tree.
      1. else if (balance of subroot = equal_height)
         1. subroot->balance = left_higher
         2. taller = TRUE
      1. else if (balance of subroot = right_higher)
         1. subroot->balance = equal_height
         2. taller = FALSE
   5. else // (DataIn > subroot->data) Insert in right subtree
      1. result = recursive_Insert(subroot->right, DataIn, taller_temp)
      2. if (taller_temp = TRUE)
         1. if (balance of subroot = left_higher)
            1. subroot->balance = equal_height
            2. taller = FALSE
         1. else if (balance of subroot = equal_height)
            1. subroot->balance = right_higher
            2. taller = TRUE
         1. else if (balance of subroot = right_higher)
            1. right_balance (subroot)
            2. taller = FALSE
      // Rebalancing always shortens the tree.
      1. return result

end recursive_Insert

Listing 1 – recursive_Insert algorithm

Fill the appropriate value to the following Table 1

<table>
<thead>
<tr>
<th>old value of subroot-&gt;balance</th>
<th>subtree to be inserted</th>
<th>returned taller_temp</th>
<th>new value of subroot-&gt;balance</th>
<th>returned value of taller</th>
</tr>
</thead>
<tbody>
<tr>
<td>equal_height</td>
<td>left</td>
<td>true</td>
<td>left_higher</td>
<td>true</td>
</tr>
<tr>
<td>equal_height</td>
<td>left</td>
<td>false</td>
<td>?</td>
<td>?</td>
</tr>
<tr>
<td>equal_height</td>
<td>right</td>
<td>true</td>
<td>?</td>
<td>?</td>
</tr>
<tr>
<td>equal_height</td>
<td>right</td>
<td>false</td>
<td>?</td>
<td>?</td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Explanation: In the second row, the initial value of subroot->balance is equal_height. The new data will be inserted into the left subtree. The returned taller_temp is true,
meaning that the height of the left subtree will be increased after the insertion. Thus, based on the algorithm, the new value of `subroot->balance` should be `left_higher` and the returned value of `taller` should be `true`.

**Question 4.**
Write an auxiliary recursive function of a Binary Tree ADT in pseudocode to validate a BST given as a subroot pointer. Also write a method in pseudocode to validate the BST.

```plaintext
algorithm checkBST_recur (val subroot <BinaryNode>,
               ref min <DataType>, ref max <DataType>)
    This algorithm check if the input subroot is a BST recursively
    Pre  subroot points to a root of the subtree
    Post  min and max are the smallest and largest value in the subtree
    Return true if the subtree is a BST, false otherwise
end checkBST_recur

algorithm isBST ()
    This algorithm check if the tree is a BST
    Pre
    Post
    Return true if the tree is a BST, false otherwise
end isBST

**Question 5.**
Write an auxiliary recursive function of a Binary Tree ADT in pseudocode to validate an AVL given as a subroot pointer. Also write a method in pseudocode to validate the AVL.

```plaintext
algorithm checkAVL_recur (val subroot <BinaryNode>,
               ref min <DataType>, ref max <DataType>,
               ref height <int>)
    This algorithm check if the input subroot is an AVL recursively
    Pre  subroot points to a root of the subtree
    Post  min and max are the smallest and largest value in the subtree
          and height is the height of the subtree
    Return true if the subtree is an AVL, false otherwise
end checkAVL_recur

algorithm isAVL ()
    This algorithm check if the tree is an AVL
    Pre
    Post
    Return true if the tree is an AVL, false otherwise
end isAVL

**Question 6.**
Write a global function in pseudocode to generate a BST from an input list by insert elements in the list into an initial empty BST. Refer to question 1 for an example.
**Algorithm** generateBSTfromList (val list <List>)

This algorithm generate a BST from the input list

**Pre**

**Post** the BST is built by inserting elements in the list into an initial empty tree one-by-one from the beginning of the list.

**Return** the BST

**end** generateBSTfromList

**Question 7.**

Write an auxiliary recursive function of a Binary Tree ADT in pseudocode to generate an AVL from an input ordered sub-list such as:
- For a sub-list from position `idx_1` to `idx_2`, the element at position `mid = (idx_1 + idx_2) / 2` will be the root of the sub-tree.
- The sub-tree on the left is build recursively from the sub-list from position `idx_1` to `(mid-1)`.
- The sub-tree on the right is build recursively from the sub-list from position `(mid+1)` to `idx_2`.

Also write a method in pseudocode to generate the AVL from an input ordered list.

**Algorithm** buildAVLfromList_recur (val list <List>, val idx1 <int>, val idx2 <int>)

This algorithm build an AVL from the input ordered list by taking the middle element in the list as the root and recursively build the left and right subtree from the left part and right part of the list

**Pre** list is an ordered list, `idx1` and `idx2` initialized 0 and `list.size-1`

**Post** the AVL is built

**Return** the root of the AVL

**end** buildAVLfromList_recur

**Algorithm** buildAVLfromList (val list <List>)

This algorithm build an AVL from the input ordered list by calling the function `buildAVLfromList`

**Pre**

**Post** the AVL is built

**end** buildAVLfromList