Question 1.

(a)

(b)

(c)

(d)
Question 2a.

71, 1, 4:

72, 33, 19:

60:

59:

21:

17:
Question 2b.

```
1,4,71:  1        4       71
         "     "     "
72,91,19,60,59:
  13       33      87
     "   "     "
13,87,33:
10     71      13
    "     "     "
```

Question 3.

<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>equal_height</td>
<td>false</td>
</tr>
<tr>
<td>equal_height</td>
<td>right</td>
<td>true</td>
<td>right_higher</td>
<td>true</td>
</tr>
<tr>
<td>equal_height</td>
<td>right</td>
<td>false</td>
<td>equal_height</td>
<td>false</td>
</tr>
<tr>
<td>left_higher</td>
<td>left</td>
<td>true</td>
<td>equal_height</td>
<td>false</td>
</tr>
<tr>
<td>left_higher</td>
<td>left</td>
<td>false</td>
<td>left_higher</td>
<td>false</td>
</tr>
<tr>
<td>left_higher</td>
<td>right</td>
<td>true</td>
<td>equal_height</td>
<td>false</td>
</tr>
<tr>
<td>left_higher</td>
<td>right</td>
<td>false</td>
<td>left_higher</td>
<td>false</td>
</tr>
<tr>
<td>right_higher</td>
<td>left</td>
<td>true</td>
<td>equal_height</td>
<td>false</td>
</tr>
<tr>
<td>right_higher</td>
<td>left</td>
<td>false</td>
<td>right_higher</td>
<td>false</td>
</tr>
<tr>
<td>right_higher</td>
<td>right</td>
<td>true</td>
<td>equal_height</td>
<td>false</td>
</tr>
<tr>
<td>right_higher</td>
<td>right</td>
<td>false</td>
<td>right_higher</td>
<td>false</td>
</tr>
</tbody>
</table>
Question 4.

**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

1. isBST = true
2. **if** (subroot)
   1. min = max = subroot->data
   2. **if** (subroot->left)
      Check the left subtree and retrieve the min and the max of the left subtree. The min of the left subtree is also the min of this tree.
      1. isBST = checkBST_recur(subroot->left, min, lmax)
      2. isBST = isBST and lmax < subroot->data
   3. **end if**
3. **end if**
4. **if** (subroot->right)
   Check the right subtree and retrieve the min and the max of the right subtree. The max of the right subtree is also the max of this tree.
   1. isBST = isBST and checkBST_recur(subroot->right, rmin, max)
   2. isBST = isBST and rmin > subroot->data
5. **end if**
3. **end if**
4. **return** isBST
**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

1. **return** checkBST_recur(root, min, max)
**end** isBST

Question 5.

**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

1. isAVL = true
2. if (subroot)
   1. min = max = subroot->data
   2. lh = rh = 0
   3. if (subroot->left)
      1. isAVL = checkAVL_recur(subroot->left, min, lmax, lh)
      2. isAVL = isAVL and lmax < subroot->data
   4. end if
   5. if (subroot->right)
      1. isAVL = isAVL and checkAVL_recur(subroot->right, rmin, max, rh)
      2. isAVL = isAVL and rmin > subroot->data
   6. end if
   7. isAVL = isAVL and ((lh == rh) or (lh == rh + 1) or (lh + 1 == rh))
   8. height = (lh > rh) ? lh + 1 : rh + 1
3. end if
4. return isAVL

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

1. return checkAVL_recur(root, min, max, height)
end isAVL

Question 6.

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

1. aBST = create a Binary tree
2. idx = 0
3. loop (idx < list.size())
   1. list.retrieve (idx, x)
   2. aBST.insert (x)
   3. idx++
4. end loop
5. return aBST
end generateBSTfromList

Question 7.

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 if an ordered list, idx1 and idx2 initialized 0 and list.size-1
Post the AVL is built
Return the root of the AVL

1. subroot = null
2. if (idx1 <= idx2)
   1. Allocate the subroot
   2. mid = (idx1 + idx2)/2
   3. list.retrieve (mid, subroot->data)
   4. subroot->left = buildAVLfromList_recur(list, idx1, mid-1)
   5. subroot->right = buildAVLfromList_recur(list, mid+1, idx2)
3. end if
4. return subroot
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

1. root = buildAVLfromList_recur (list, 0, list.size()-1)
end buildAVLfromList