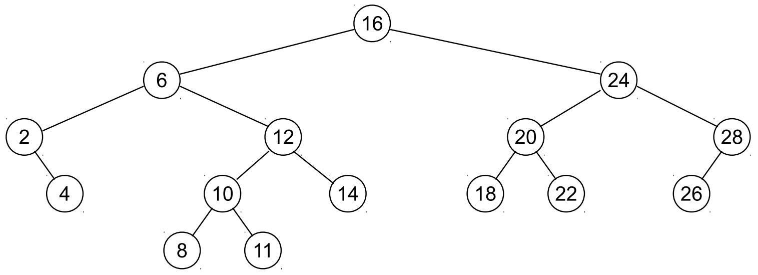
ECE-250 – Algorithms and Data Structures (Winter 2012)

Tutorial 5 (2012-02-16) – Suggested solutions

1 - Given the AVL tree in the following figure:



(a) Remove the smallest element twice, then remove the largest element twice, then remove the root node twice.

(b) After completing all removals from part (a), insert the following values in the given order: 17, 1, 15, 36, 41.

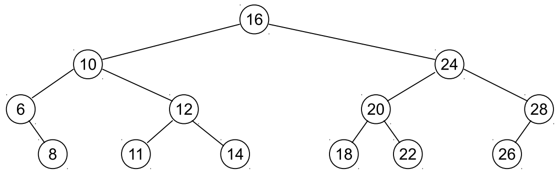
Solution:

(a) Removing 2 promotes 4, which causes an imbalance at node 6 (left sub-tree has height 0, right sub-tree has height 2). The imbalance is the "zig-zag" type (right-left), so we do a double rotation. 10 ends up at the root of that sub-tree, with 6 on its left and 4 as 6's left; 8 ends up at 6's right, and 11 ends up at 12's left.

No imbalance is caused at the root note (which is the next one in ascending sequence towards the root), so we're done — the tree is balanced.

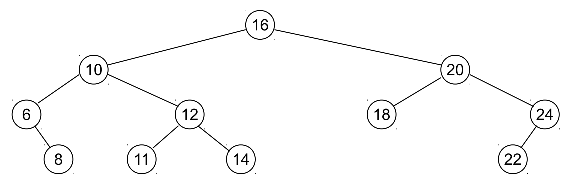
Now the smallest element is 4; removing it causes no imbalance.

So far, the tree is as shown in the figure:



Now we remove the largest element (twice). Removing 28 promotes 26; no imbalance is caused. Then, we remove 26, which causes an imbalance at node 24 — interestingly, we can see this imbalance as a left-left (if we consider the node 18), or as a left-right (if we consider 22). The easier one is the "aligned" case, so we rotate towards the right (detaching 22 first). We end up with 20 at the root of that sub-tree, 24 at its right, and 22 at 24's left. No further imbalance is caused.

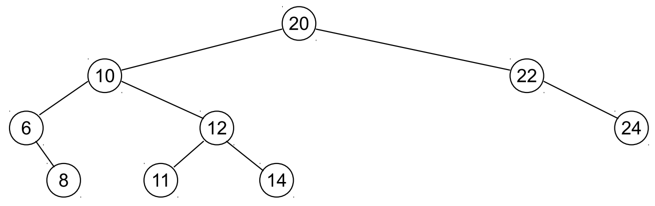
At this point, the tree is as shown in the figure:



We now remove the root node twice. 16 is replaced by 18; since 18 is a leaf node, it can be removed trouble-free (to be promoted to the root's tree). However, despite the removal itself being trouble-free, it does cause an imbalance at node 20, which is fixed by a double rotation, with 22 ending up at the root of that sub-tree, 20 at its left, 24 at its right. No further imbalance is created.

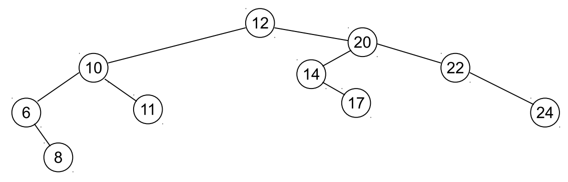
Then, 18 is removed and replaced by 20 (a leaf node), causing non imbalance.

At this point, the tree is as shown in the figure:



(b)

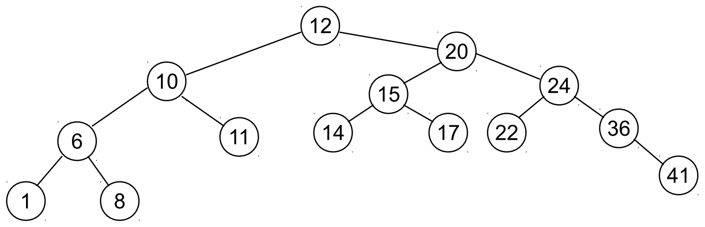
Inserting 17 (at the right of 14) causes an imbalance at the root node only. Since it is the only node, it is the "deepest" one (which is where we have to fix the imbalance). This is a "zig-zag" imbalance (left-right), so we do a double rotation, with 12 ending up as the root, as shown in the figure:



Now, inserting 1 is trouble-free (goes at the left of 6, causing no imbalance anywhere). 15 is inserted at the left of 17, causing imbalance at node 14, fixed with a double rotation, with 15 ending as parent of 14 and 17.

36 is then inserted at the right of 24, causing an imbalance at node 22, easily fixed with a single rotation towards the left. Finally, 41 is inserted at the right of 36, causing no imbalance.

The tree ends up as shown below:



2 - Write a recursive C++ function that determines whether a given binary tree has a configuration consistent with a valid AVL tree. The function declaration or function prototype would be as follows:

```
template <typename Type>
bool is_avl (const Binary_node<Type> * tree);
```

Solution:

We simply follow the recursive definition for AVL trees, which means that we just need to compare heights additionally to check whether it is a BST (which we already did last week's tutorial — in any case, it is in the suggested solutions for last week's tutorial):

```
template <typename Type>
bool is_avl (const Binary_node<Type> * tree)
{
    if (tree == NULL)
    {
        return true;
    }

    return    is_bst (tree)
           && is_avl (tree->left()) && is_avl (tree->right())
           && std::abs(height(tree->left()) - height(tree->right())) <= 1;
}
```

3 - Discuss (and sketch the modified function) how could you make this function more efficient, by avoiding redundant calculations of the height of sub-trees. (Hint: how about the recursion going bottom-to-top, and given the height of the lower sub-trees, we compute the height of the sub-trees corresponding to the parent — maybe the function could return an `int` instead of just `bool`?)

Solution:

The key detail here is that the function is_avl could return an int with the height *if it is* an AVL tree, or -2 if it's not. The recursive function then checks; if any of the sub-trees (any of the recursive calls) return -2, then we return -2 without any further analysis/processing.

Otherwise, we check the two returned heights; if they differ by more than 1, we return -2 (our two sub-trees are AVL trees, but the difference in their heights makes *us* not an AVL tree). If their difference is within ± 1, then we return 1 + the higher of the two heights being returns by the two recursive calls (the two sub-trees).

The code could be something like this:

```
template <typename Type>
int avl_height (const Binary_node<Type> * tree)
{
    if (tree == NULL)
    {
        return -1;
    }

    const int height_left = avl_height (tree->left());
    const int height_right = avl_height (tree->right());

    if (   height_left == -2 || height_right == -2
        || std::abs(height_left - height_right) > 1)
    {
        return -2;
    }

    return 1 + std::max (height_left, height_right);
}

template <typename Type>
bool is_avl (const Binary_node<Type> * tree)
{
    return avl_height(tree) > -2;
}
```

4 - Given a binary search tree, which of the four traversals (breadth-first, pre-order, in-order, and post-order depth-first) will list the entries in such a way as to reconstruct the same tree contents if they are inserted into an initially empty binary search tree in the order in which they're visited with the given traversal? (for the yes cases, explain why, and for the no cases, show a counter-example to support your case)

How would the answer change if we consider an AVL tree instead of simply a binary search tree? That is, if we have an AVL tree and would like to insert the values into an initially empty AVL tree in such an order as to recreate the same AVL tree?

Solution:

Clearly, in-order traversal is out — we know that it will produce a tree that is arranged in linear sequence (like a linked list).

Post-order traversal is also out, since two values a, b, with $a < b$ can be either arranged as parent /

right-child, or left-child / parent, depending on the order that they are inserted. This simple example illustrates this:

We have a tree with root 1, and right-child 2. If we print post-order, we get the sequence 2 - 1, which when inserted into an initially empty BST, produces a tree with 2 at the root and 1 as its left-child.

The important detail is that as long as parent nodes are output before any of its children, the tree will be reconstructed identically — the nodes that restrict where a value can be placed are all the ancestors; if all ancestors have been already inserted in the tree, then each new value will necessarily go in the same position where it was in the original tree.

So, either pre-order depth-first or breadth-first would do.

As for AVL, clearly breadth-first is the only one that would work — since you exhaust all the siblings before going any deeper, no insertions would cause imbalances, so no rotations would occur, which means that there is no difference with respect to the regular BST case.

Any depth-first traversal would end up causing imbalances, and as soon as things are rotated, etc., patterns are lost, so we surely end up with a different tree.