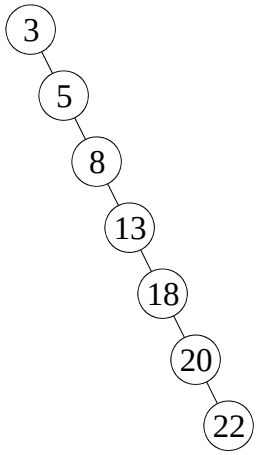
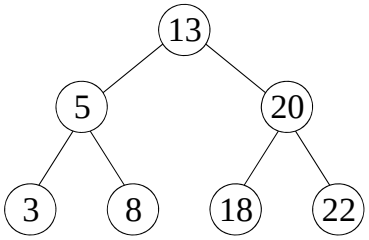
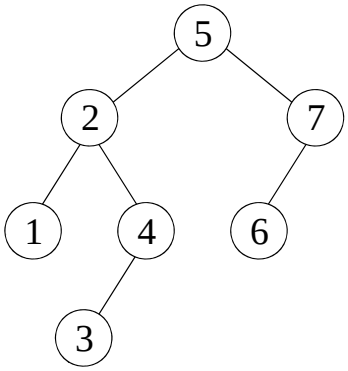
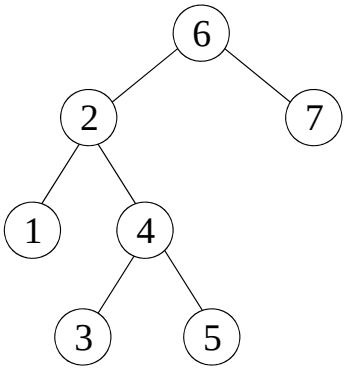
- Motivating Circumstance
  (**Note**: For our purpose, height is defined such that *height of a **1-node tree*** is ***0*** and *height of an **empty tree*** is ***-1***.)
  (**Caveat**: Others may define height such that *height of a **1-node tree*** is ***1*** and *height of an **empty tree*** is ***0***.)
  - For a BST with ***n*** nodes and height ***h***, search, insert and remove is ***O(h)***.
  - Height ***h*** is ***O(n)*** in the worst case and ***O(log n)*** in the best case.
    - Seven-node ***(n = 7)*** example involving 3, 5, 8, 13, 18, 20 and 22:

Worst Case ***(h = 6)***	Best Case ***(h = 2)***
	

  - This means search, insert and remove is ***O(n)*** in the worst case.
  - It is ∴ useful to have a good way to ensure ***h*** is always ***O(log n)*** so that search, insert and remove is ***O(log n)***.
  - To ensure ***h*** is always ***O(log n)***, tree must be *height-balanced*.
    - AVL trees are BSTs that are height-balanced, *i.e.*, also meet specified *balance condition* or *AVL property*. (More formally, AVL tree maintains a *balance invariant* <u>in addition to</u> the *ordering invariant* of BST.)
      - More modern *self-rebalancing* BSTs exist nowadays (red-black tree, for *e.g.*), but AVL tree is first such kind proposed and many of its pioneering concepts/ideas are components of the more modern variants.

- Defining AVL
  - An AVL tree is a BST that is either empty or whose (root node's) sub-trees (LST and RST) have heights that ***differ at most by1*** <u>and</u> (recursively), the LST and RST themselves are also AVL trees.
    - Only *height* balancedness is required → *node counts* of LST and RST *can differ by more than 1*.
      - Terms used by some to describe this: not weight balanced, not perfectly balanced, sub-optimal, …
      - **Quiz**: Every *complete* binary tree is height-balanced, true or false?
    - Example involving 1, 2, 3, 4, 5, 6 and 7:

BST and AVL	BST but Not AVL
	

    - AVL is derived from the names of two Russians that proposed the idea: **A**delson-**V**elskii and **L**andis.

- Facilitating AVL
  - A simple way to help determine if tree rooted at a node is height balanced is to augment the regular binary tree node structure with an additional attribute.
  - Most directly, the additional attribute keeps the height of the tree.
    (Arguably less "optimized" but simpler in concept and more direct in maintenance/application.)
    - Example:

Regular Binary Tree Structure

```
struct btNode
{  int data;
   btNode* left;
   btNode* right;
};
```

Augmented Binary Tree Structure

```
struct bthNode
{  int data;
   btNode* left;
   btNode* right;
   int height;
};
```

    - Signed difference (between heights of LST and RST) gives height-related information about tree.
  - Alternatively (more commonly, it appears), the additional attribute keeps a *balance factor (bf)* we'll define as:

bf = height of RST – height of LST

    (Arguably more "optimized" but more abstract in concept and less direct in maintenance/application.)
    - Example:

Regular Binary Tree Structure

```
struct btNode
{  int data;
   btNode* left;
   btNode* right;
};
```

Augmented Binary Tree Structure

```
struct btbNode
{  int data;
   btNode* left;
   btNode* right;
   int bf; // height(RST) - height(LST)
};
```

    - Sign and magnitude (of the value of the factor itself) give height-related information about tree.
      - Basic samples:

balance factor of a node	***about tree rooted @ the node***
`== 0`	heights of LST and RST are equal
`<= -1`	left heavy (has LST that is taller than RST)
`>= 1`	right heavy (has RST that is taller than LST)
`2` or `-2`	AVL property (balance condition) violation

      - More complex samples (just for getting a good feel → need to be further along in study to fully get them):
        (Assumption: *remove* done in usual standard way, where node physically removed is always a leaf node)
        (In these samples, backtrack point is at parent of node physically removed.)

balance factor @ backtrack point	***about tree rooted @ backtrack point***
becomes `0`	height of LST or RST has decreased by 1
becomes `1` or `-1`	height of LST or RST has not changed
becomes `2` or `-2`	tree rooted at backtrack point is no longer AVL-balanced

    (**Note**: Some define *bf* as *height of LST – height of RST*, with corresponding reversal in the sign's significance.)
- Using AVL
  - Overview of how an AVL tree is put to use (as a storage structure), allowing *dynamic insert*/*remove*:
    (In essence, ensure tree *is AVL when an operation begins* and *remains AVL when the operation completes*.)
    (Focus is on insert and remove → search is identical to BST and would not incur any side effect on the tree.)
    - As usual, usage begins with an empty tree, which is AVL.
      - (The usual empty-tree initial state thus nicely kicks things off, *i.e.*, tree is AVL to begin with.)
    - Inserting/removing an item goes exactly as for (ordinary) BST, augmented with *AVL-maintain* thereafter:
      - Check if AVL-property still holds after BST-insert or BST-remove.
      - If AVL-property no longer holds (balance condition violated), do something (*rotate*, will see) to fix it.

■ Managing AVL-insert

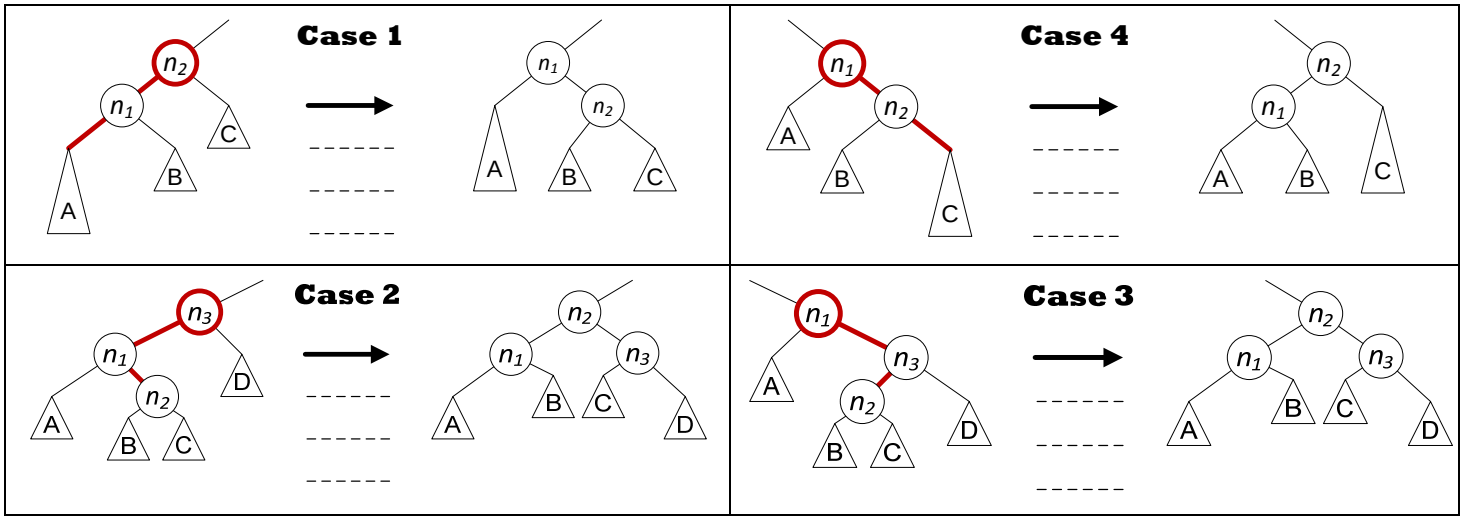
- Following points on balance condition violation/restoration associated with *insert* can be shown/reasoned:
  - After an insert, *only nodes on path between root and parent of inserted node* may have heights altered.
    - Thus, only trees rooted at the said nodes need to be inspected for balance condition violation.
  - Rebalancing tree at *deepest "balance-condition-violated" node* will restore balance of entire (overall) tree. (Key: *height of local tree after rebalancing* ***same as*** *height of said tree before node insertion*)
    - Considering this and the preceding point gives a scheme for *AVL-maintain* after BST-insert is done:

```
Loop (backtrack) through nodes on path from parent of inserted node to root:
   // local tree is tree rooted at node located at current backtrack point
   Update height or balance factor of local tree
   If (updated height or balance factor indicates balance condition violation)
      Perform appropriate rotation to rebalance local tree
      Break
   Else if (height or balance factor remains unchanged during update)
      Break
```

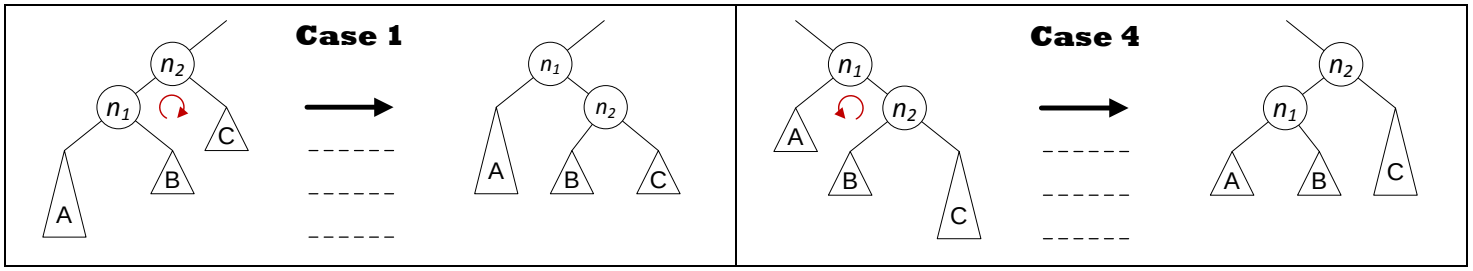
  - Violation cases for deepest "balance-condition-violated" node *v*:
    (**Tip:** Can spot it using the two links *immediately* after node *v* leading to culprit sub-tree that is overly tall.)
    - **1**: Newly inserted node is into *LST of left child* of node *v* ("left-left" or "LL" case)
    - **2**: Newly inserted node is into *RST of left child* of node *v* ("right-left" or "RL" case)
    - **3**: Newly inserted node is into *LST of right child* of node *v* ("left-right" or "LR" case)
    - **4**: Newly inserted node is into *RST of right child* of node *v* ("right-right" or "RR" case)

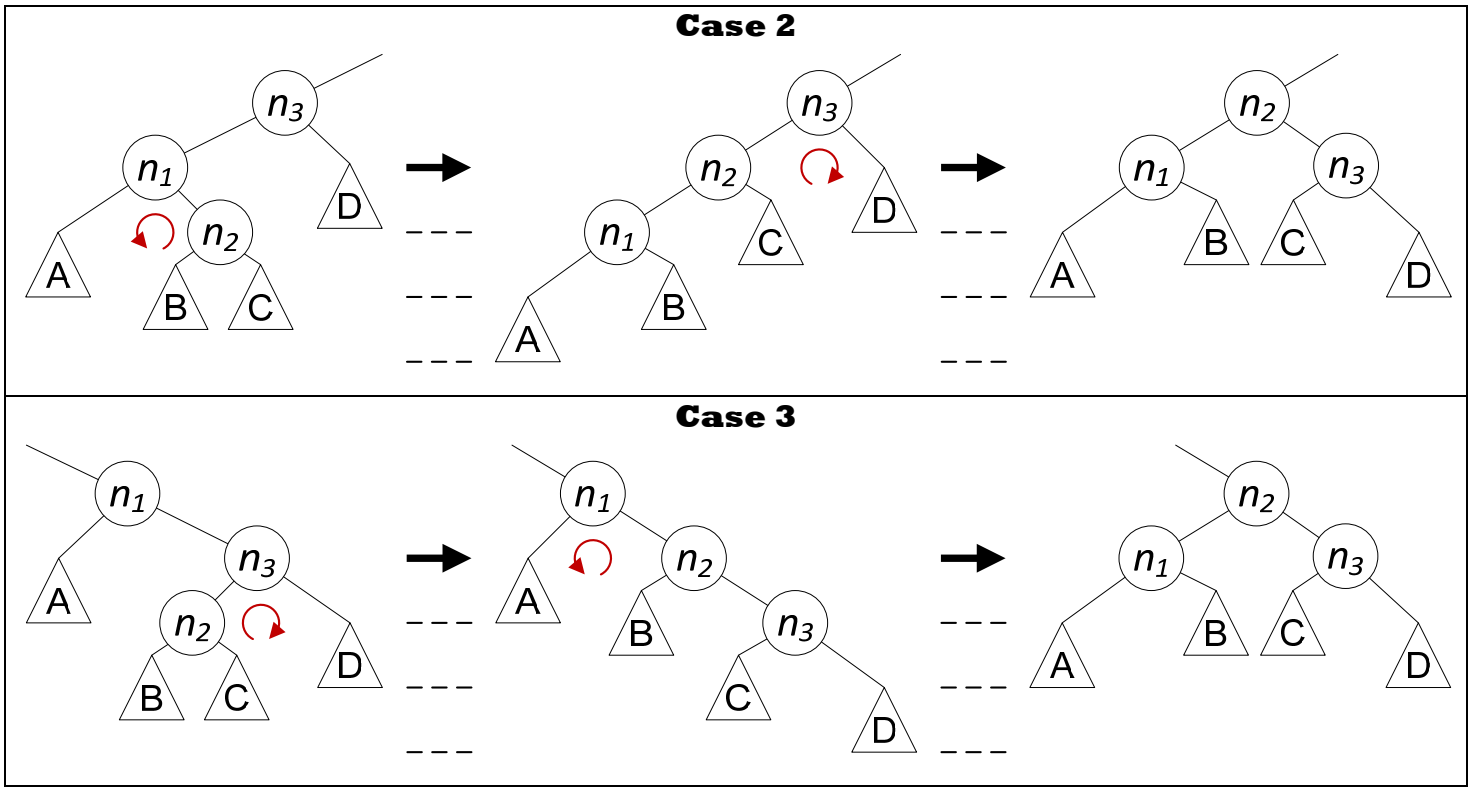
    (Case 4 is *mirror image* of Case 1, and Case 3 is *mirror image* of Case 2.)
    (Also, to rebalance, Cases 1 and 4 need only *single rotation* whereas Cases 2 and 3 need *double rotation*.)

■ Summarizing Insert-related "Violation Cases and Associated Rotations" Pictorially
(bold red oval in each case → deepest "balance-condition-violated" node *v*)
(bold red lines in each case → two links immediately after node *v* leading to culprit sub-tree that is overly tall)
(In Cases 2 and 3, B and C are *put amid 2 levels* to mean 1 is at the higher level and the other at the lower level, unless both happen to be empty.)



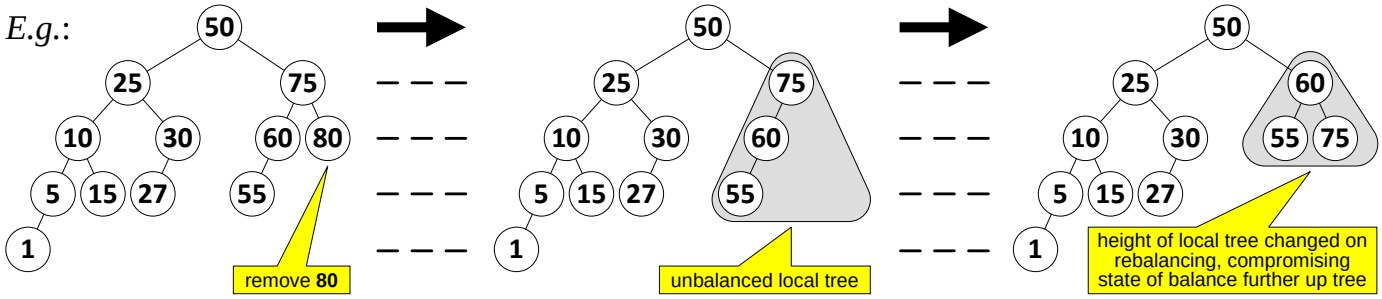
■ Clarifying Terms "*Single* Rotation" and "*Double* Rotation"





- Managing AVL-remove
  - Following points on balance condition violation/restoration associated with *remove* can be shown/reasoned:
    - After a remove, *only nodes on path between root and parent of removed node* may have heights altered. (**Note**: "*removed node*" should be the one *physically unlinked*, not one with just the data overwritten.)
      - Thus, only trees rooted at the said nodes need to be inspected for balance condition violation.
    - Rebalancing tree at a *"balance-condition-violated" node* may compromise state of balance further up tree. (Key: *height of local tree after rebalancing* ***may differ from*** *height of said tree before node removal*)

      E.g.:



      (This has to be the *most striking difference* compared to the situation seen under "Managing AVL-insert".)
      - Considering this and the preceding point gives a scheme for *AVL-maintain* after BST-remove is done:

```
Loop (backtrack) through nodes on path from parent of removed node to root:
   // local tree is tree rooted at node located at current backtrack point
   Update height or balance factor of local tree
   If (updated height or balance factor indicates balance condition violation)
      Perform appropriate rotation to rebalance local tree
      If (height or balance factor of local tree is unchanged during removal)
         Break
   Else if (height or balance factor remains unchanged during update)
      Break
```

- Sizing Up AVL
  - Gain: search, insert and remove take *O(log n)* time for average and worst cases.
  - Cost: extra *O(n)* space for height or balance factor and extra *O(log n)* effort for insert and remove.

(Select reference: ***https://en.wikipedia.org/wiki/AVL_tree***)