

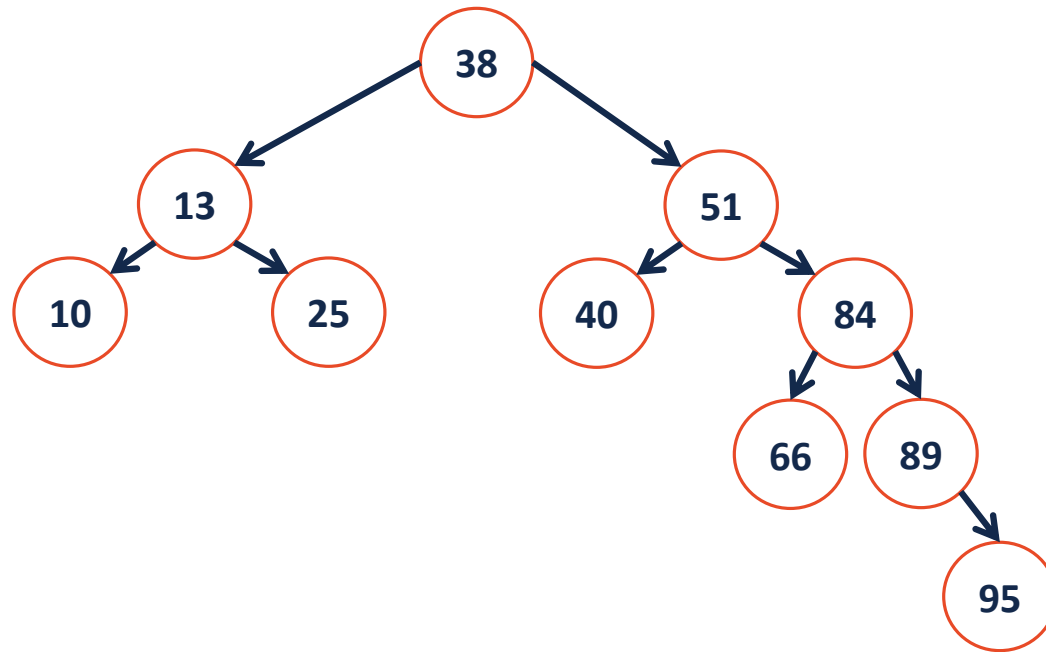
CS 225

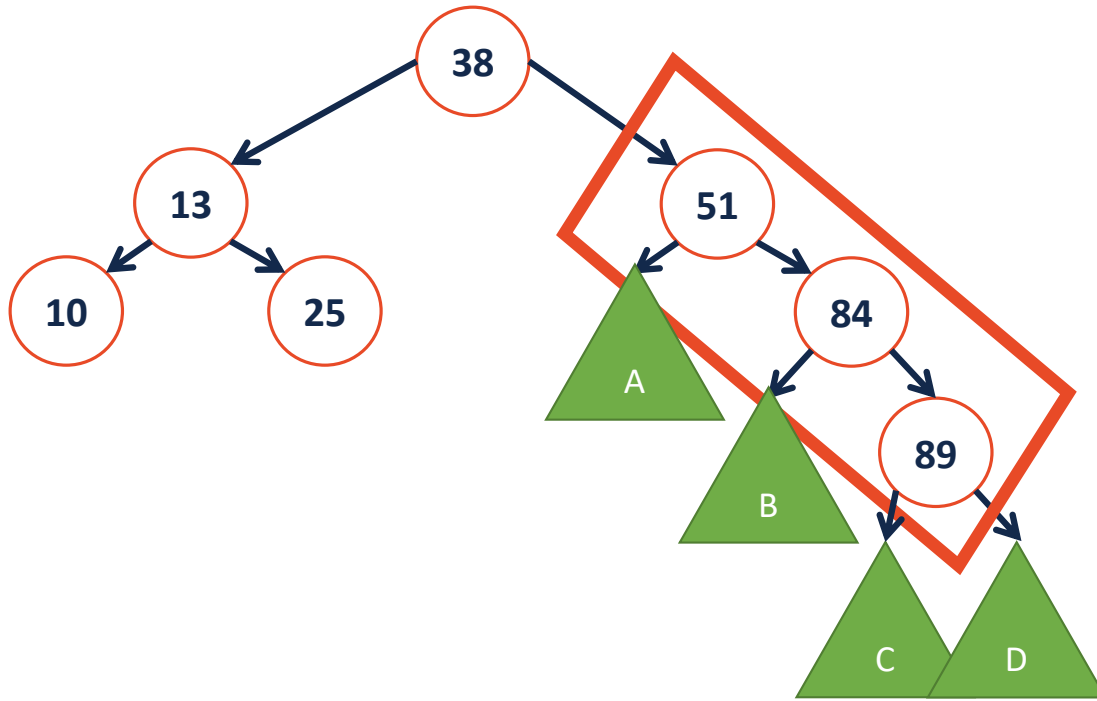
Data Structures

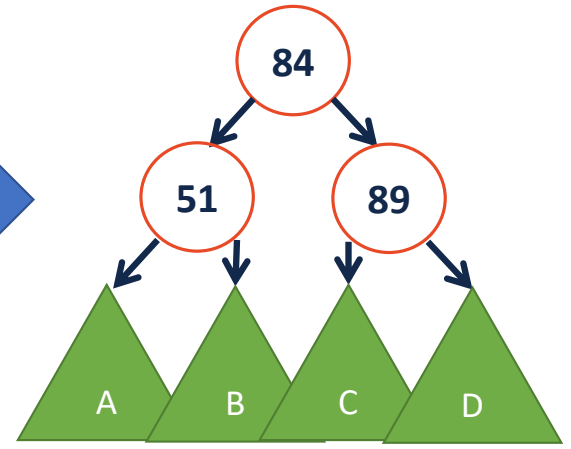
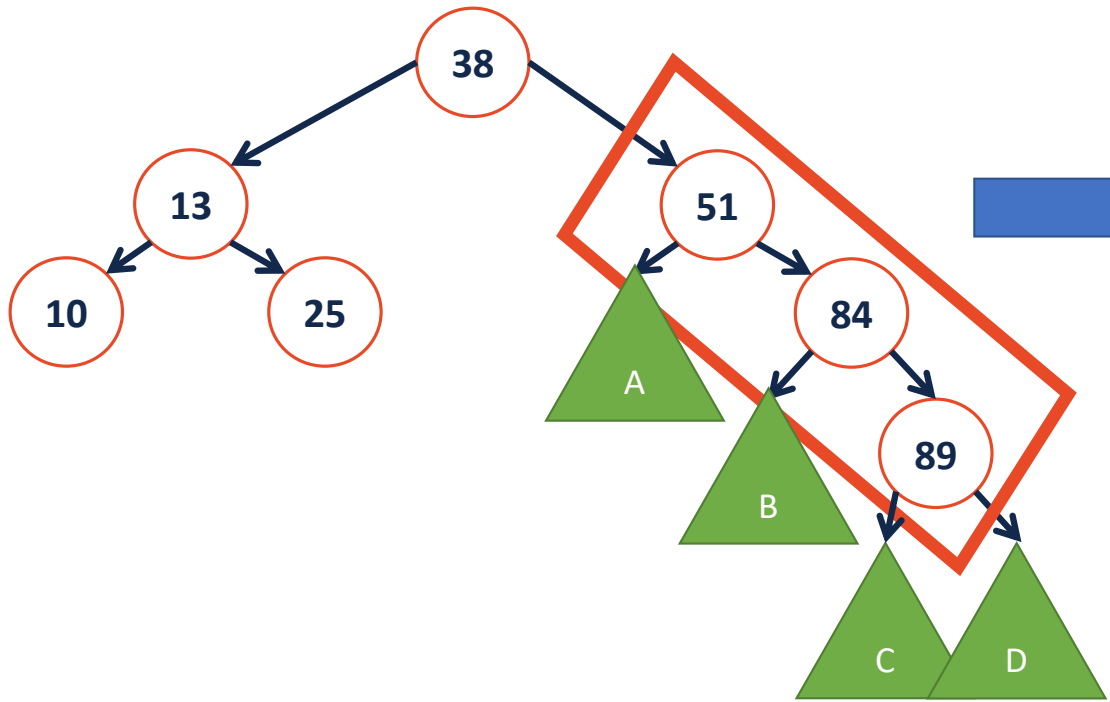
October 12– AVL Trees

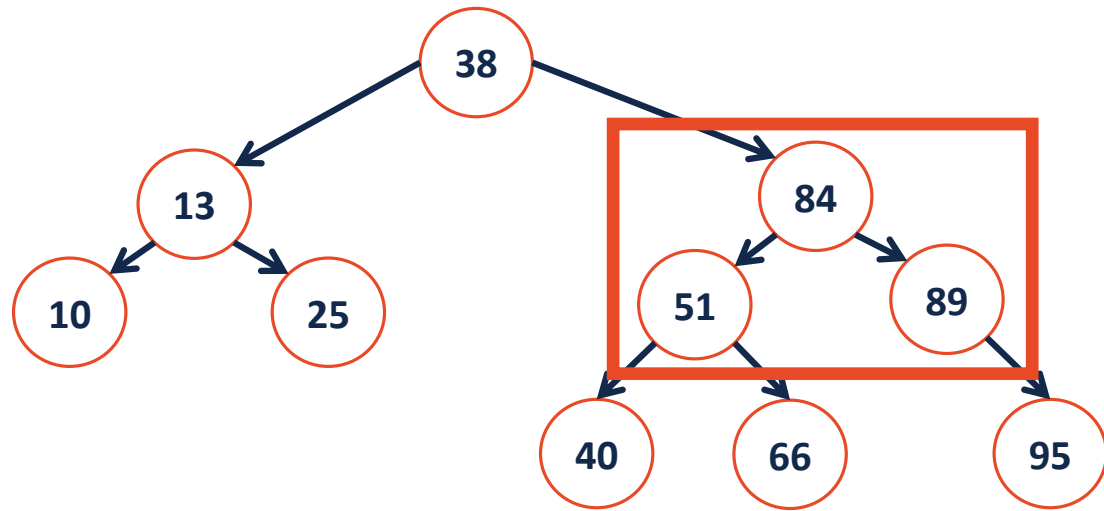
G Carl Evans

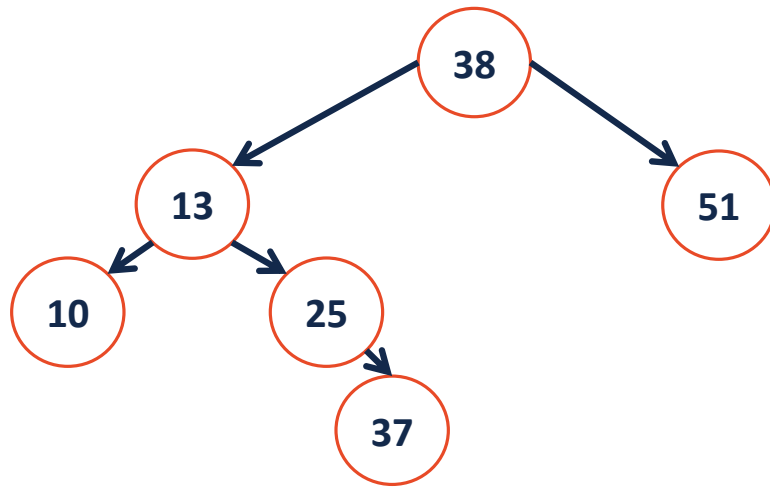
Left Rotation

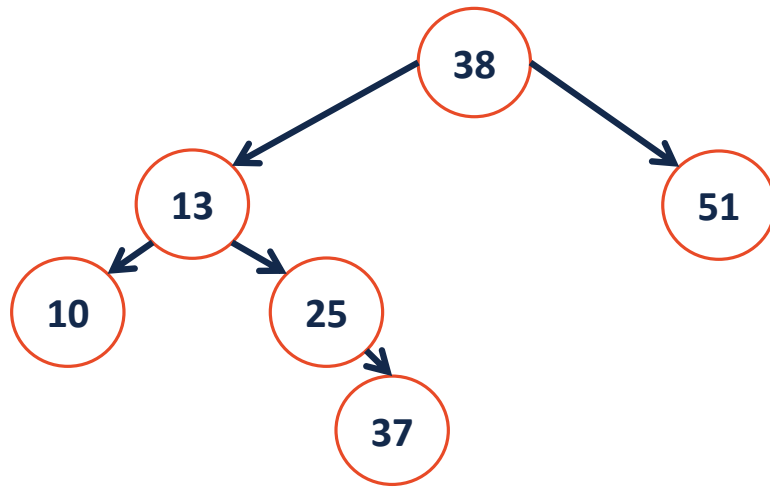














BST Rotation Summary

- Four kinds of rotations (L, R, LR, RL)
- All rotations are local (subtrees are not impacted)
- All rotations are constant time: $O(1)$
- BST property maintained

GOAL:

We call these trees:



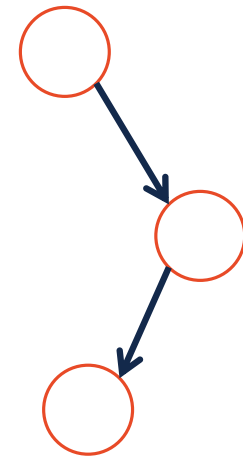
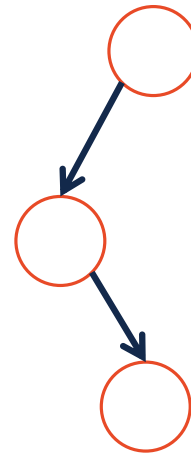
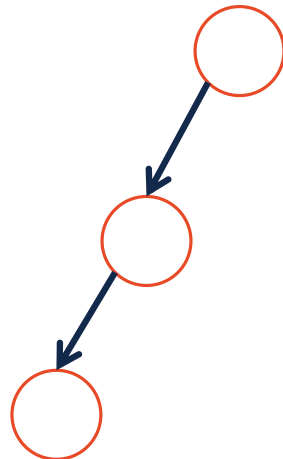
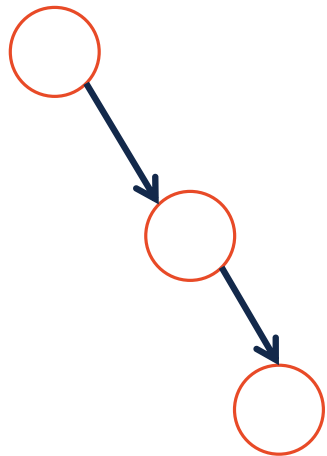
AVL Trees

Three issues for consideration:

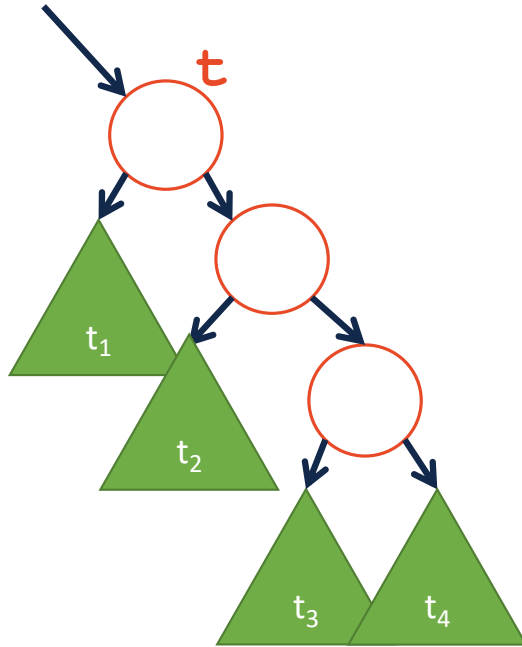
- Rotations
- Maintaining Height
- Detecting Imbalance

AVL Tree Rotations

Four templates for rotations:



Finding the Rotation on Insert

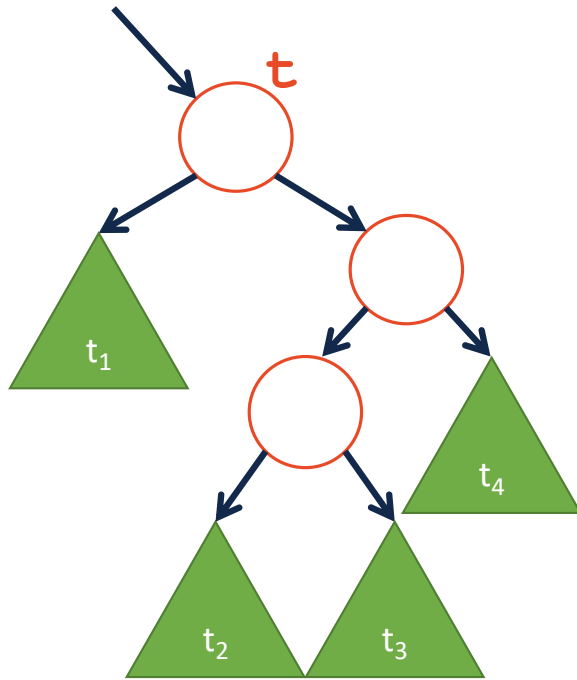


Theorem:

If an insertion occurred in subtrees t_3 or t_4 and a subtree was detected at t , then a _____ rotation about t restores the balance of the tree.

We gauge this by noting the balance factor of $t \rightarrow \text{right}$ is _____.

Finding the Rotation on Insert



Theorem:

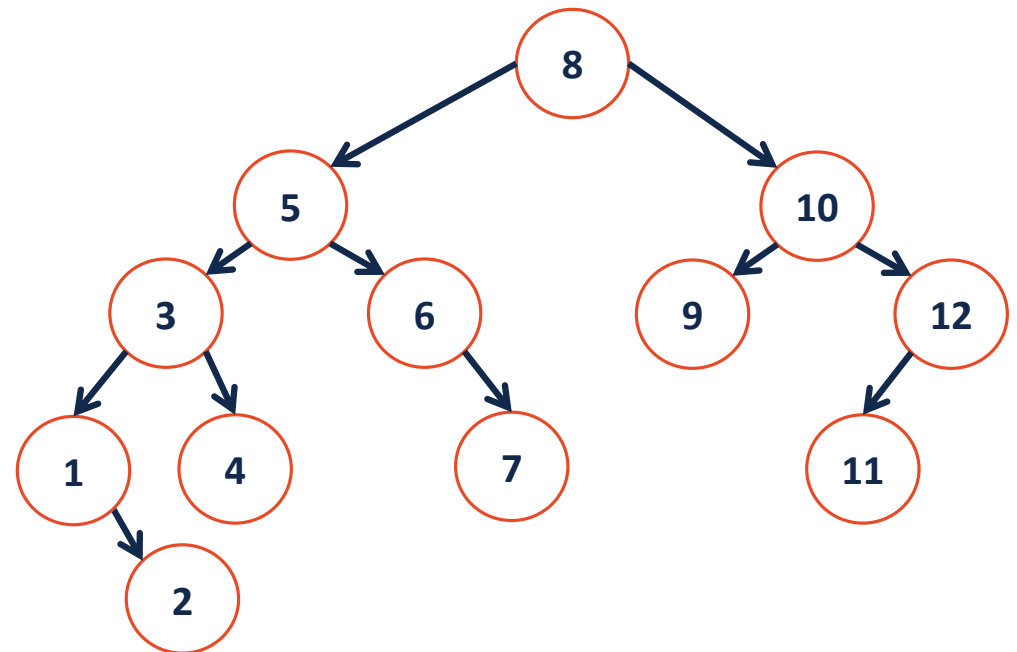
If an insertion occurred in subtrees t_2 or t_3 and a subtree was detected at t , then a _____ rotation about t restores the balance of the tree.

We gauge this by noting the balance factor of $t \rightarrow \text{right}$ is _____.

Insertion into an AVL Tree

`_insert(6.5)`

```
1 struct TreeNode {  
2     T key;  
3     unsigned height;  
4     TreeNode *left;  
5     TreeNode *right;  
6 };
```

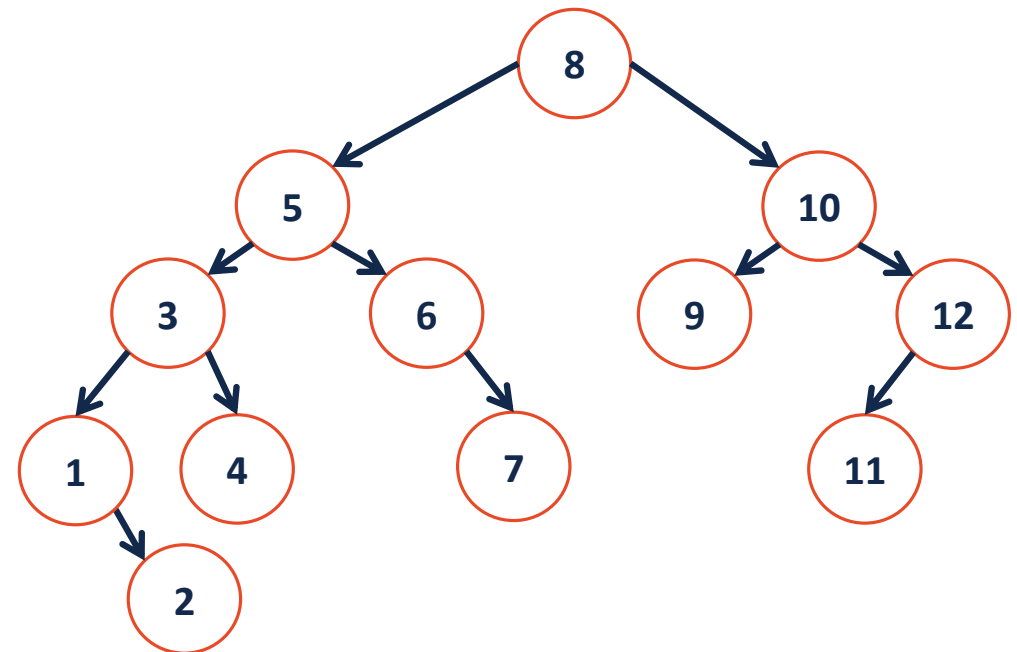


Insertion into an AVL Tree

`_insert(6.5)`

Insert (pseudo code):

- 1: Insert at proper place
- 2: Check for imbalance
- 3: Rotate, if necessary
- 4: Update height



```
1 struct TreeNode {
2     T key;
3     unsigned height;
4     TreeNode *left;
5     TreeNode *right;
6 };
```

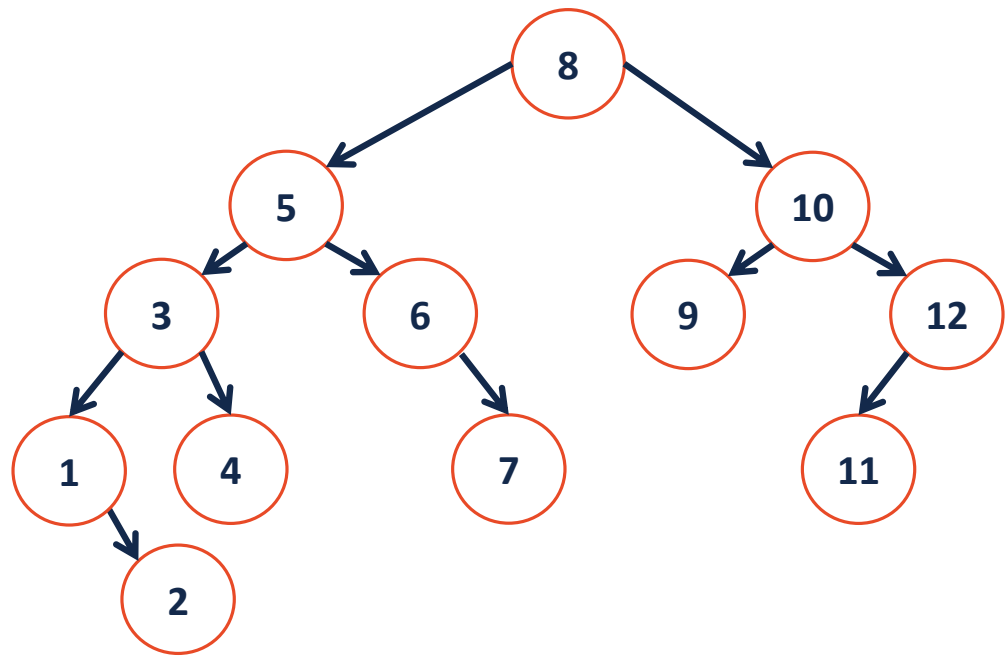
```
151 template <typename K, typename V>
152 void AVL<K, D>::_insert(const K & key, const V & data, TreeNode
    *& cur) {
153     if (cur == NULL)           { cur = new TreeNode(key, data);   }
157     else if (key < cur->key) { _insert( key, data, cur->left ); }
160     else if (key > cur->key) { _insert( key, data, cur->right );}
166     _ensureBalance(cur);
167 }
```

```
119 template <typename K, typename V>
120 void AVL<K, D>::_ensureBalance(TreeNode *& cur) {
121     // Calculate the balance factor:
122     int balance = height(cur->right) - height(cur->left);
123
124     // Check if the node is current not in balance:
125     if ( balance == -2 ) {
126         int l_balance =
127             height(cur->left->right) - height(cur->left->left);
128         if ( l_balance == -1 ) { _____; }
129         else { _____; }
130     } else if ( balance == 2 ) {
131         int r_balance =
132             height(cur->right->right) - height(cur->right->left);
133         if( r_balance == 1 ) { _____; }
134         else { _____; }
135     }
136     _updateHeight( cur );
};
```




Height-Balanced Tree

Height balance: $b = \text{height}(T_R) - \text{height}(T_L)$





AVL Tree Analysis

We know: insert, remove and find runs in: _____.

We will argue that: $h =$ _____.



AVL Tree Analysis

Definition of big-O:

...or, with pictures:

