CS 225

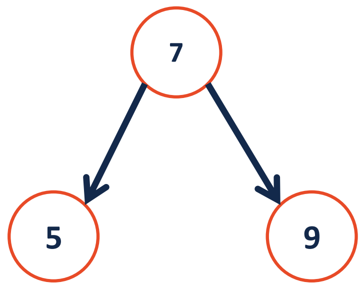
Data Structures

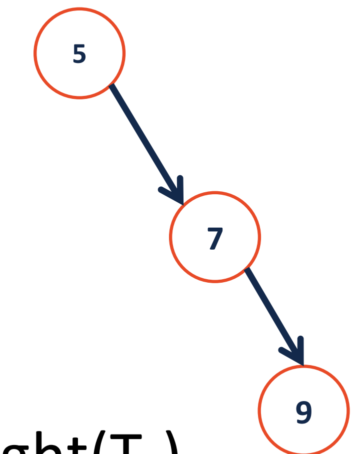
September 23 – BST Balance

G Carl Evans

Height-Balanced Tree

What tree makes you happier?





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

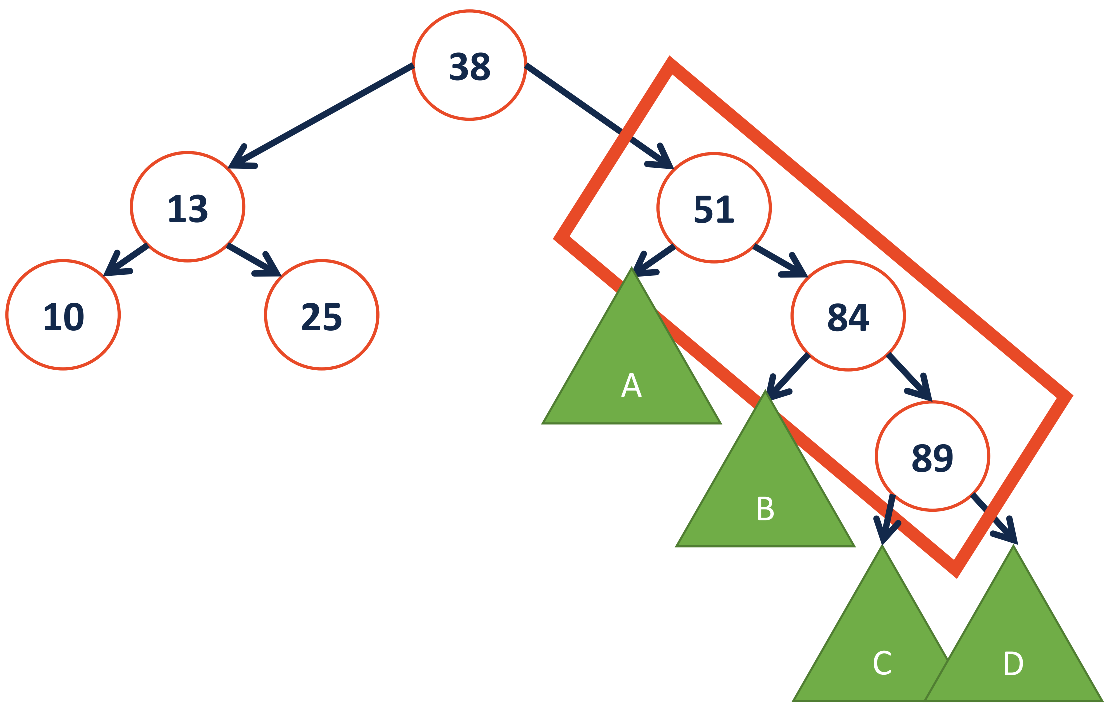
A tree is height balanced if:

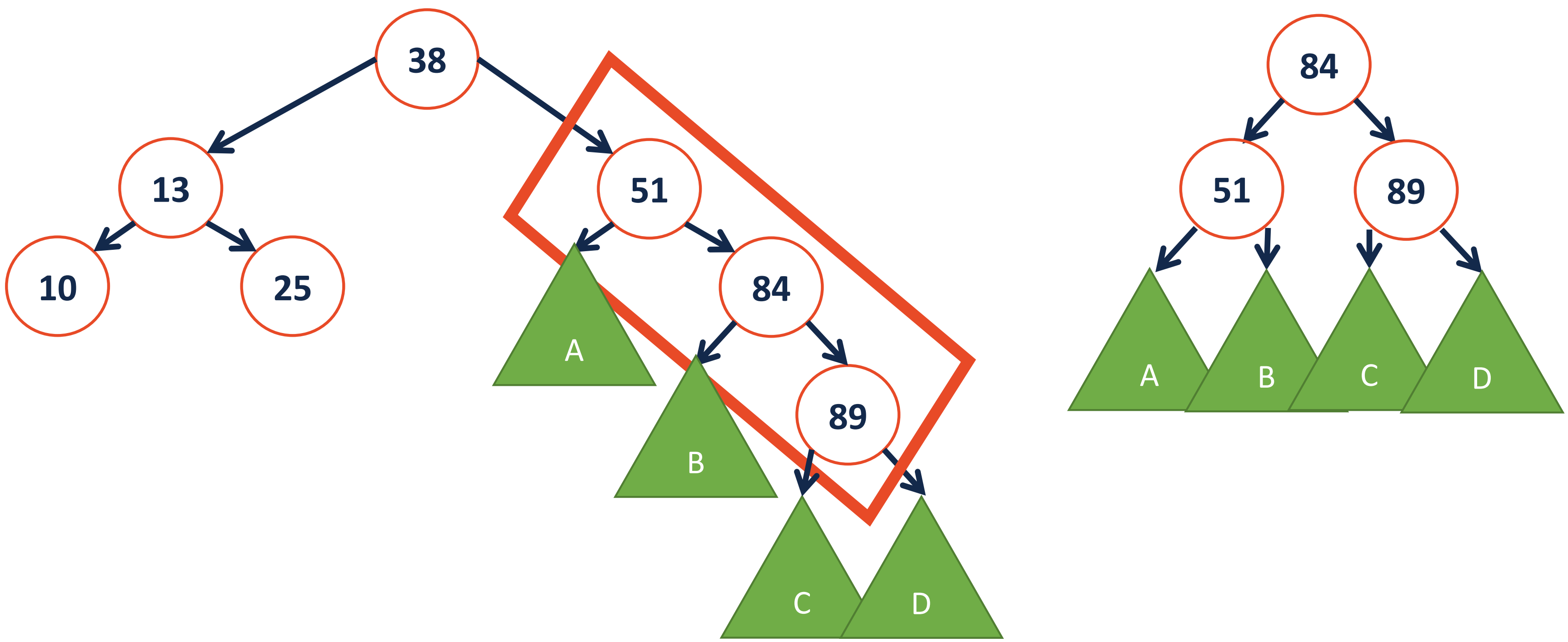
BST Rotation

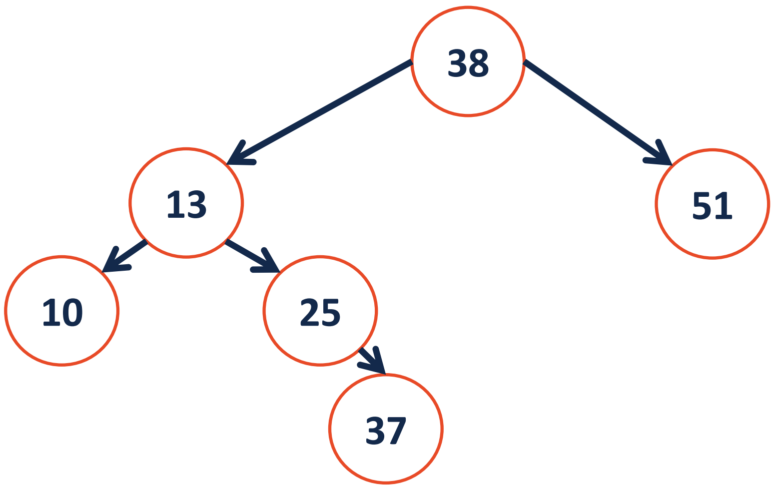
We will perform a rotation that maintains two properties:

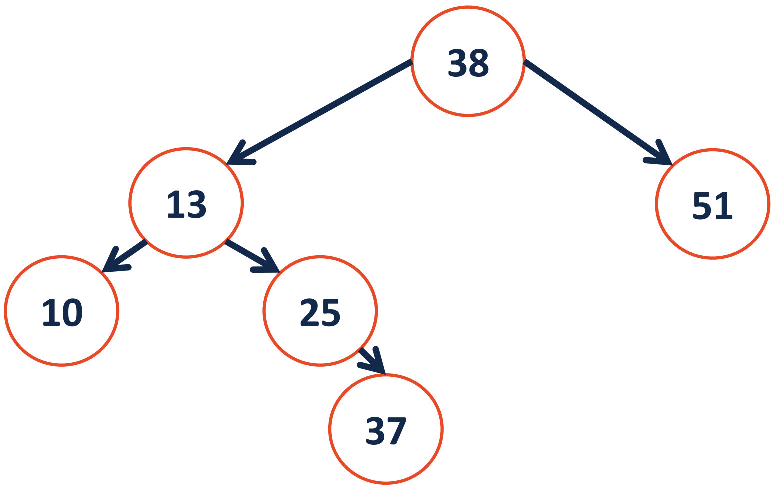
1.

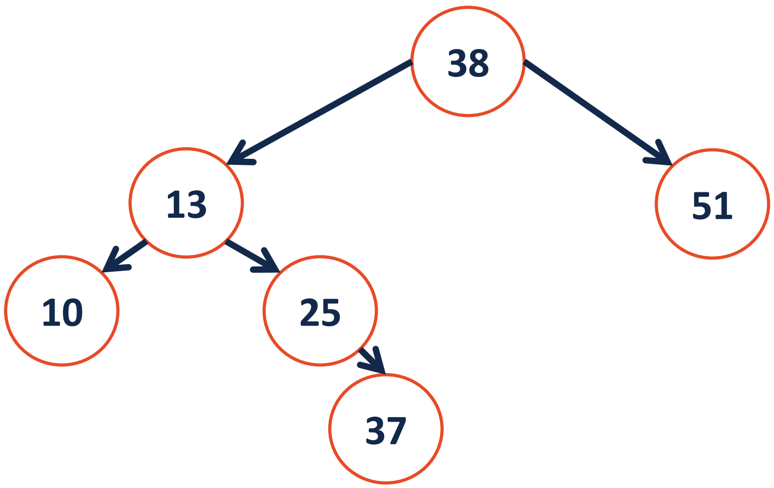
2.


38
13
51
10
25
84
89
A
B
C
D


38
13
51
10
25
84
89
A
B
C
D
84
51
89
A
B
C
D

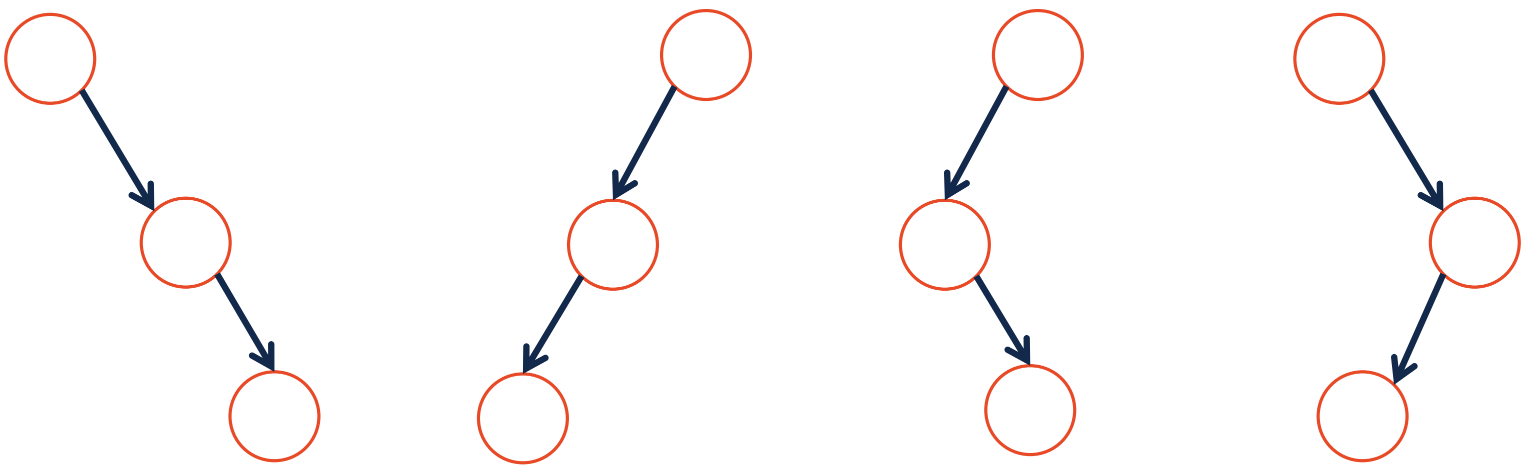

38
13
51
10
25
37


38
13
51
10
25
37


38
13
51
10
25
37

AVL Tree Rotations

Four templates for rotations:



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:

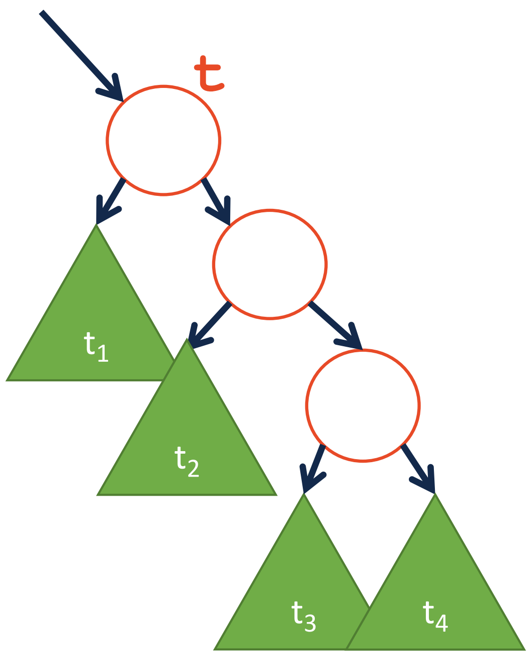
We call these trees:

AVL Trees

Three issues for consideration:

- Rotations
- Maintaining Height
- Detecting Imbalance

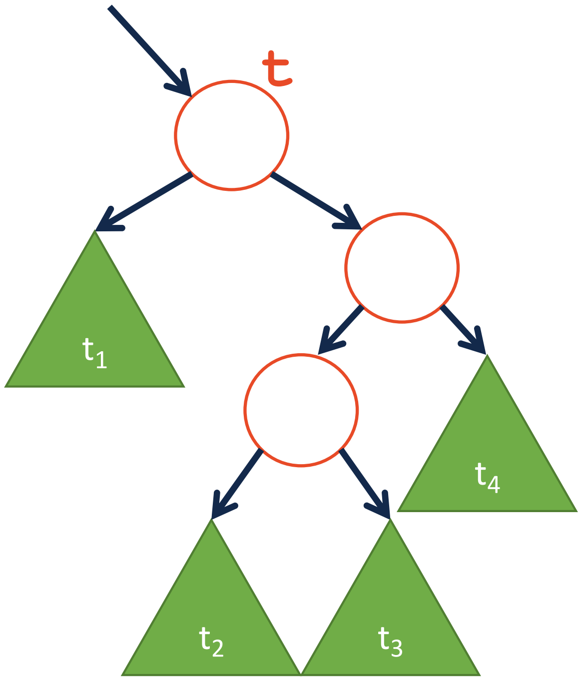
Finding the Rotation on Insert



Theorem:
If an insertion occurred in subtrees $\mathbf{t_3}$ or $\mathbf{t_4}$ 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->right** is ______.

Finding the Rotation on Insert

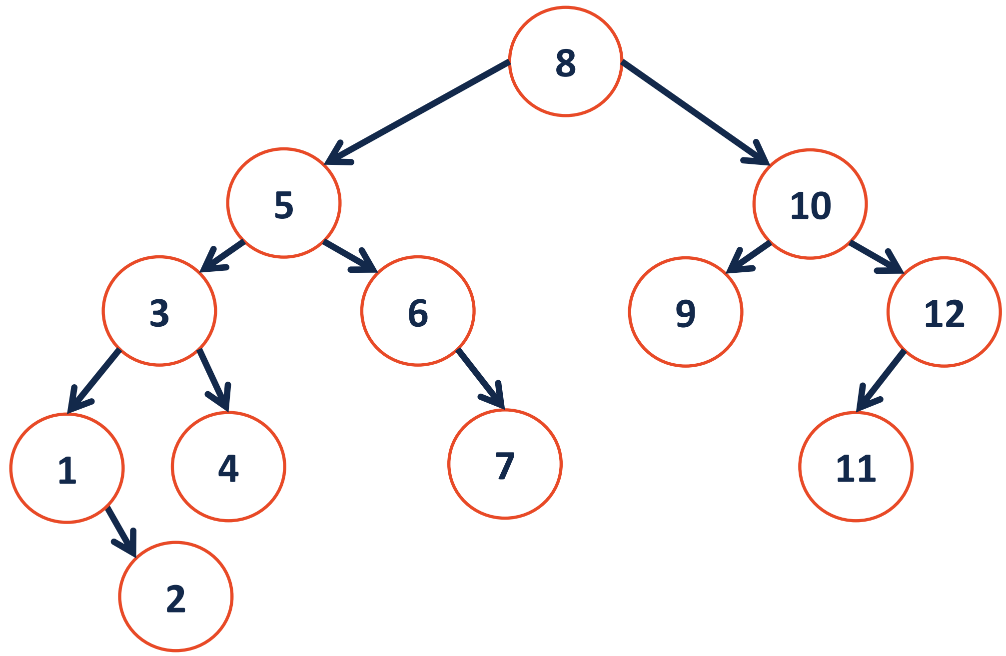


Theorem:
If an insertion occurred in subtrees $\mathbf{t_2}$ or $\mathbf{t_3}$ and a subtree was detected at **t**, then a __________ rotation about **t** restores the balance of the tree.

We gauge this by noting the balance factor of **t->right** is ______.

Insertion into an AVL Tree

`_insert(6.5)`



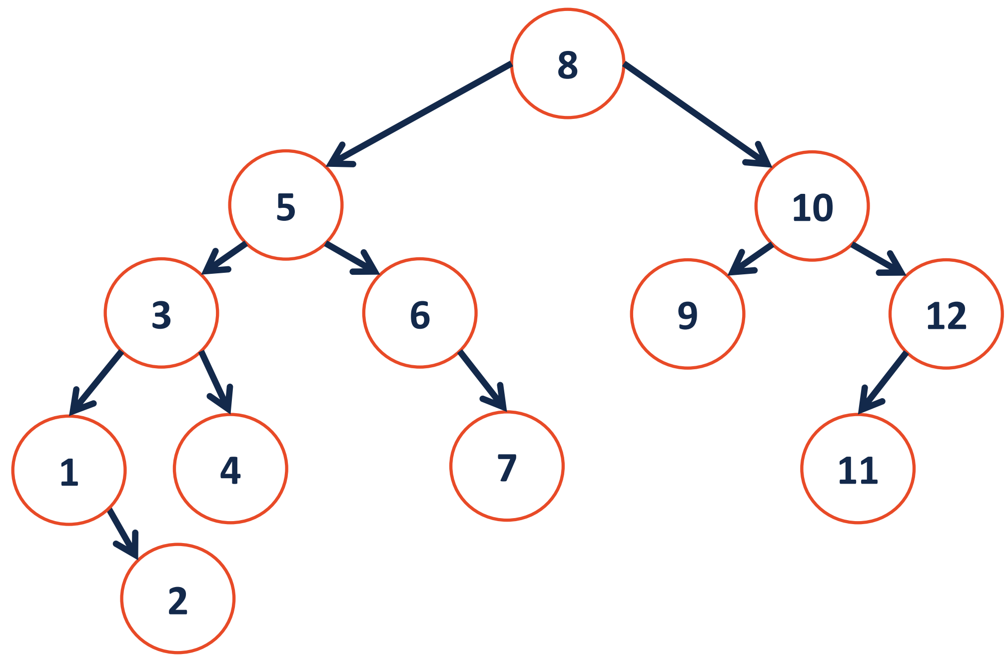

```
struct TreeNode {
  T key;
  unsigned height;
  TreeNode *left;
  TreeNode *right;
};
```

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




```
struct TreeNode {
  T key;
  unsigned height;
  TreeNode *left;
  TreeNode *right;
};
```

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

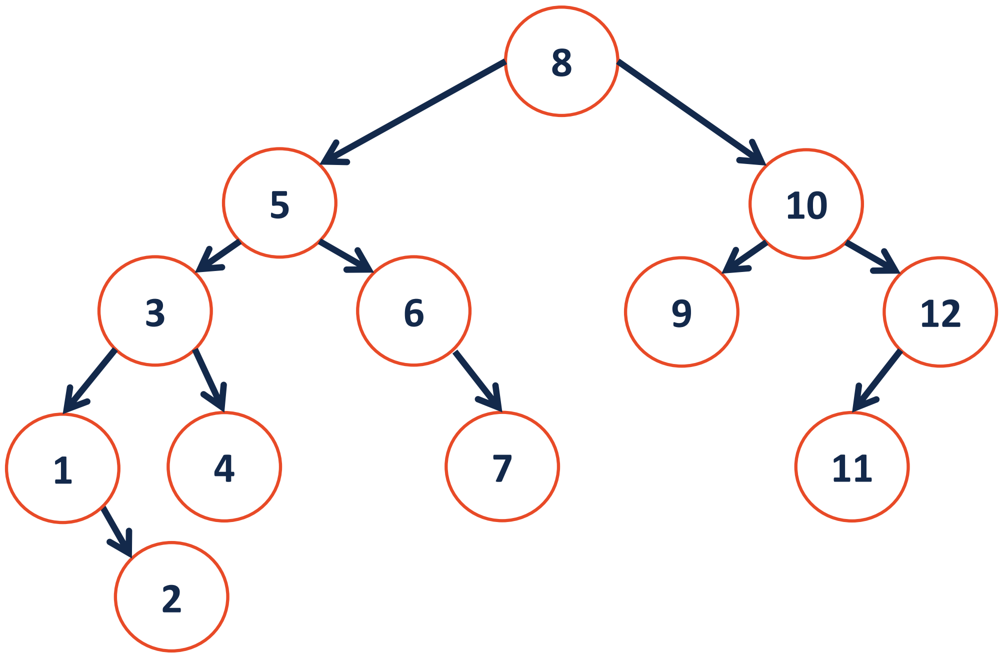
```
template <typename K, typename V>
void AVL<K, D>::_ensureBalance(TreeNode *& cur) {
  // Calculate the balance factor:
  int balance = height(cur->right) - height(cur->left);

  // Check if the node is current not in balance:
  if ( balance == -2 ) {
    int l_balance =
          height(cur->left->right) - height(cur->left->left);
    if ( l_balance == -1 ) { ____________________________; }
    else                   { ____________________________; }
  } else if ( balance == 2 ) {
    int r_balance =
          height(cur->right->right) - height(cur->right->left);
    if( r_balance == 1 ) { _____________________________; }
    else                 { _____________________________; }
  }

  _updateHeight(cur);
};
```

Height-Balanced Tree

Height balance: $b = height(T_R) - height(T_L)$



8
5
10
3
6
9
12
1
4
7
11
2