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
Finding the Rotation

If an insertion occurred in subtrees $t_3$ or $t_4$ and a subtree was detected at $t$: 
Finding the Rotation
Finding the Rotation
Finding the Rotation

t

\[ t \]

\( t_L \)

\( t_A \)

\( t_B \)
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 $t$-$\text{right}$ is ______.
Example:
Finding the Rotation

\[ t \]

\[ \begin{align*}
& t_1 \\
& \quad \downarrow \\
& t_2 \quad t_3 \\
& \quad \downarrow \quad \downarrow \\
& t_4 \quad t_4
\end{align*} \]
Finding the Rotation

Theorem:
If an insertion occurred in subtrees \( t_2 \) or \( t_3 \) and a subtree was detected at \( t \):
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 $t$-$right$ is ______.
Insertion into an AVL Tree

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

_insert(6.5)
Insertion into an AVL Tree

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;
};

_insert(6.5)
```
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);
}
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: