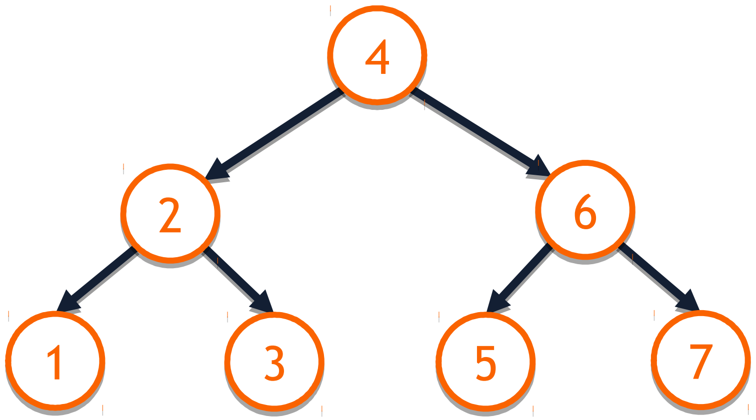
AVL Analysis

Prof. Wade Fagen-Ulmschneider

ILLINOIS

Balanced BSTs that are kept in balance through tree rotations on insert and remove are called **AVL** trees, named after **Adelson-Velsky** and **Landis**.



AVL Tree

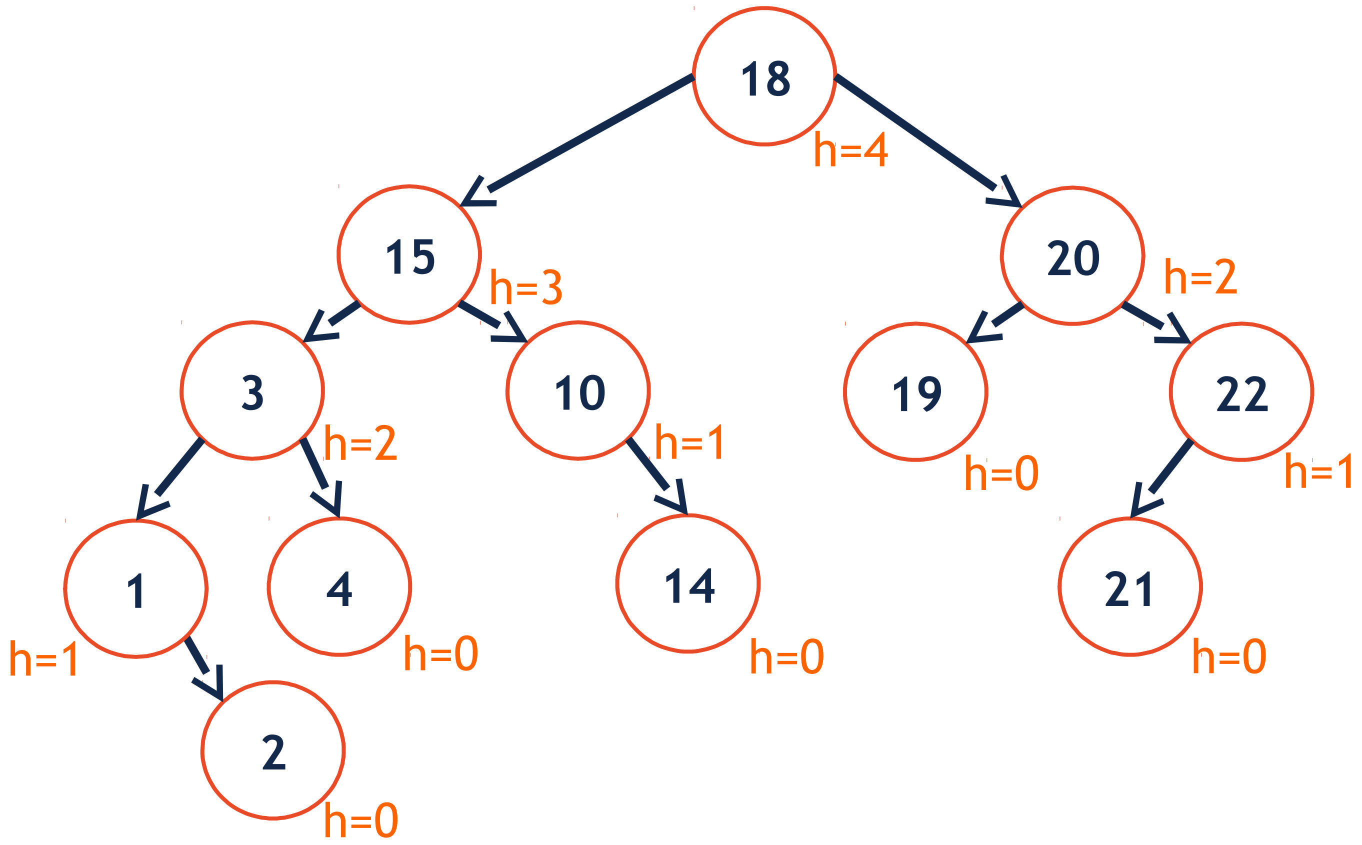
Everything about finding, inserting, and removing from a BST remains true.

In an AVL tree, we add steps to ensure we maintain balance.

- Extra work on insert/remove
- To quickly compute the balance factor, AVL trees store the height of every node as part of the node itself.

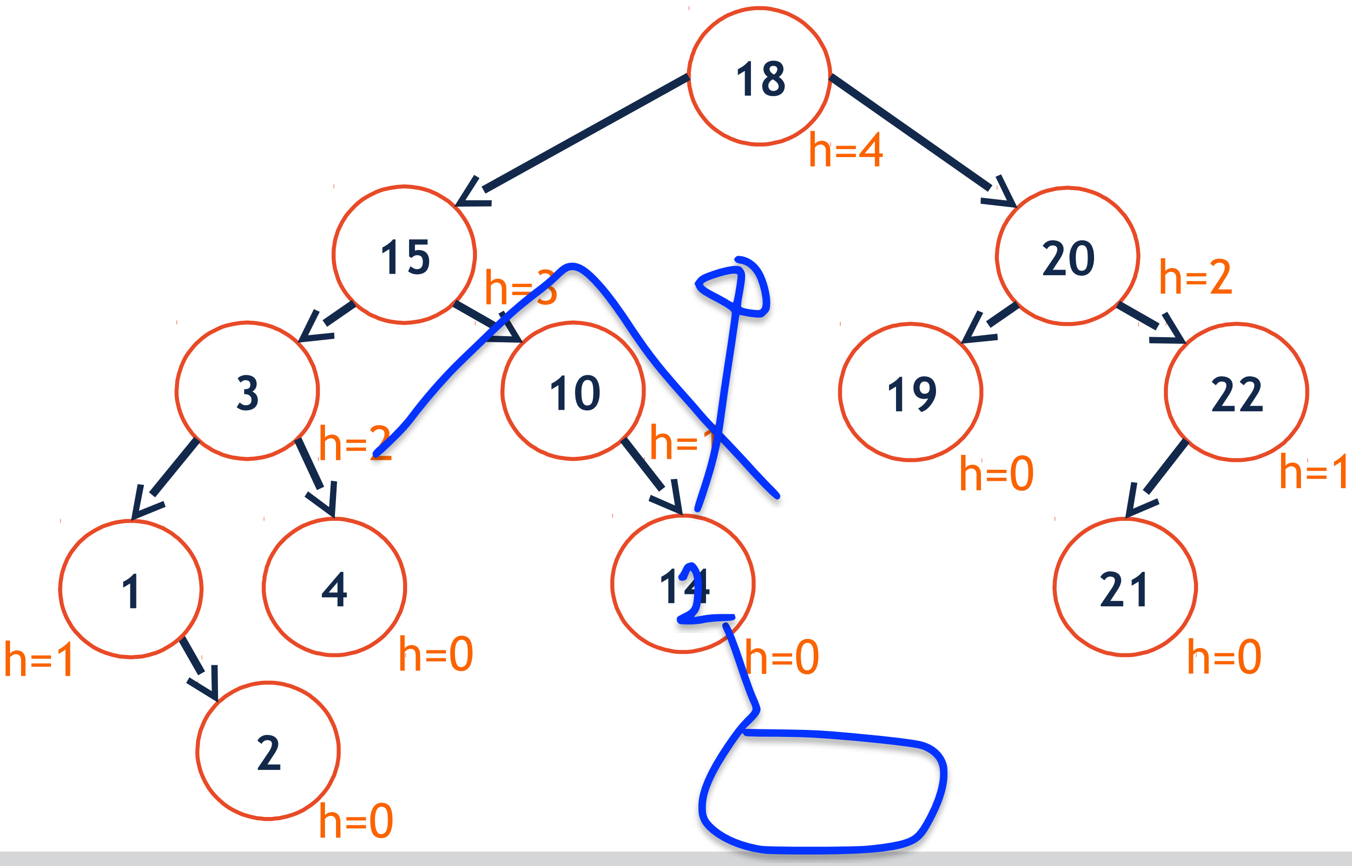
AVL Trees

insert(11)



AVL Insert

insert(11)



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

여기에 하나를 더 추가함으로서 inbalance 되기때문에 balancing 작업을 해줘야함

avl/AVL.cpp

```cpp
template <typename K, typename D>
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 ) { _rotateRight( cur ); }
    else                   { _rotateLeftRight( cur ); }
  } else if ( balance == 2 ) {
    int r_balance = height(cur->right->right) - height(cur->right->left);
    if( r_balance == 1 ) { _rotateLeft( cur ); }
    else                 { _rotateRightLeft( cur ); }
  }

  _updateHeight(cur);
}
```

avl/AVL.cpp

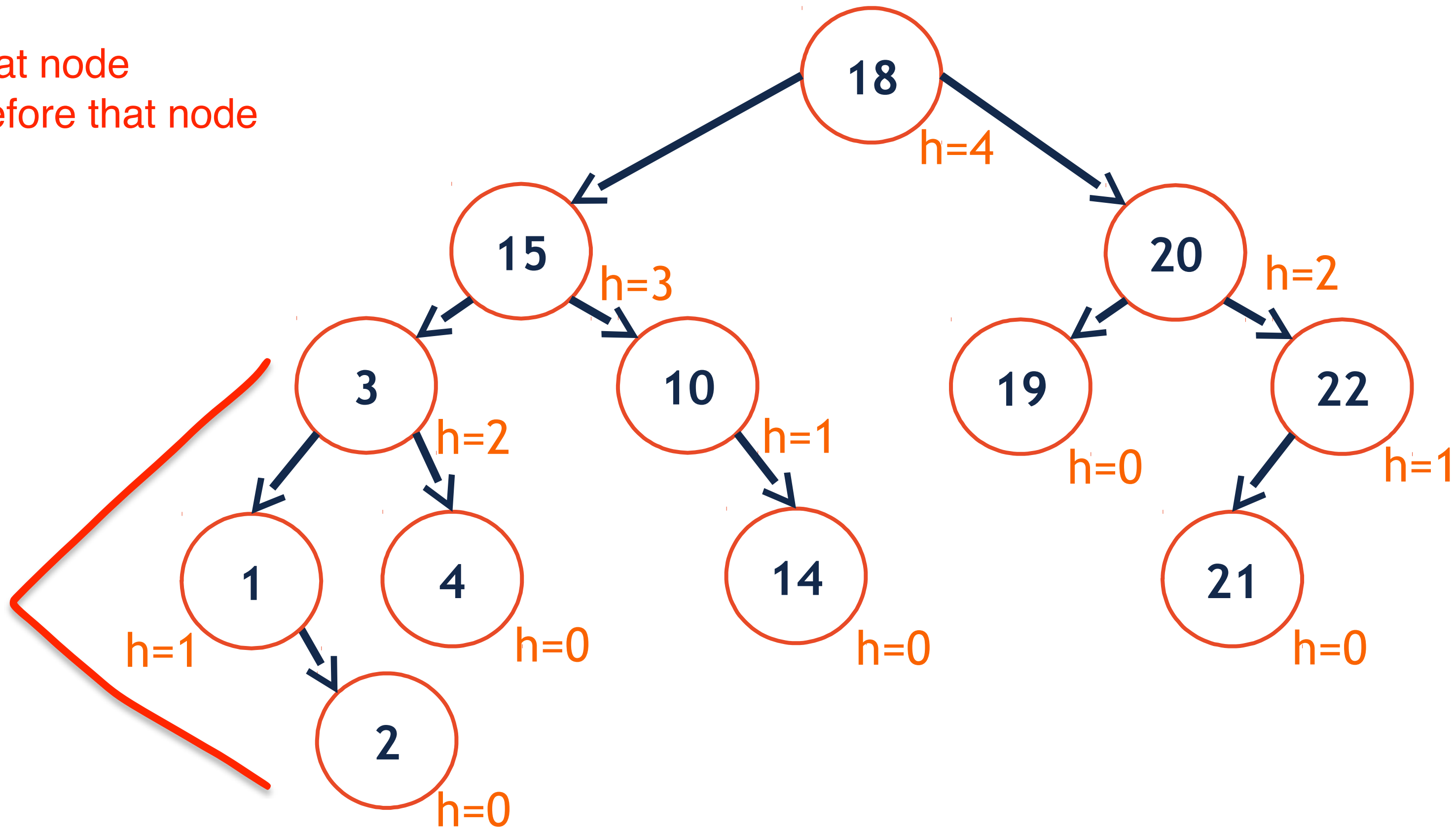
```cpp
template <typename K, typename D>
void AVL<K, D>::_updateHeight(TreeNode *& cur) {
  cur->height = 1 + max(height(cur->left), height(cur->right));
}
```

```cpp
template <typename K, typename D>
void AVL<K, D>::_rotateLeft(TreeNode *& cur) {
  TreeNode *x = cur;
  TreeNode *y = cur->right;

  x->right = y->left;
  y->left = x;
  cur = y;

  _updateHeight(x);
  _updateHeight(y);
}
```

AVL::remove

remove(15)

Remember : in order to remove,
we need to find the in-order predecessor of that node
-> we have to find the node that comes just before that node
or rightmost node in the left sub tree.



avl/AVL.cpp

```cpp
/**
* Recursive IoP remove.
*/
template <typename K, typename D>
const D & AVL<K, D>::_iopRemove(TreeNode *& node, TreeNode *& iop) {
  if (iop->right != nullptr) {
    // IoP not found, keep going deeper:
    const D & d = _iopRemove(node, iop->right);
    if (iop) { _ensureBalance(iop); }
    return d;
  } else {
    // Found IoP, swap the location:
    _swap( node, iop );
    std::swap( node, iop );

    // Remove the swapped node (at iop's position):
    return _remove(iop);
  }
}
```

AVL Trees

- An AVL Tree is an implementation of a **balanced** Binary Search Tree (BST).

- An implementation of an AVL tree starts with a BST implementation and adds two key ideas:
  - Maintains the height at each node
  - Maintains the balance factor after insert and remove