Height-balanced BST

Recall:

- Binary Search Tree can be used to perform dictionary operations (search, insert and delete) in O(h) time.

- h = height of tree is usually much smaller than the size n of the tree (h << n), but in the worst-case, h = n – 1.

- When the tree is built with a random sequence of keys, height h = O(log n). (Note: best case h = O(log n).
- With AVL tree and red-black tree, h is guaranteed to be O(log n) in the worst-case.

Randomly generated binary search tree

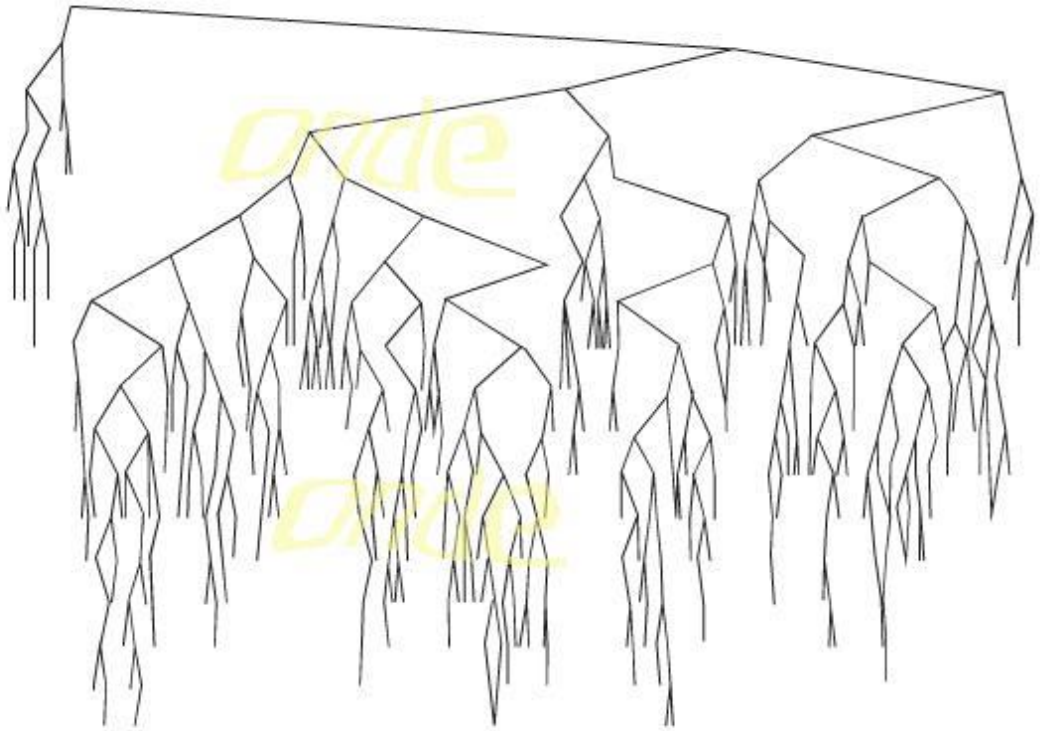


Figure 4.29 A randomly generated binary search tree

Balanced Binary Search Tree

- Worst case height of binary search tree: N-1
  - Insertion, deletion can be O(N) in the worst case
- We want a tree with small height
- Height of a binary tree with N node is at least O(log N)
  - Complete binary tree has height log N.
- Goal: keep the height of a binary search tree O(log N)
- Balanced binary search trees
  - Examples: AVL tree, red-black tree

AVL tree

for each node, the height of the left and right subtrees can differ by at most d = 1.

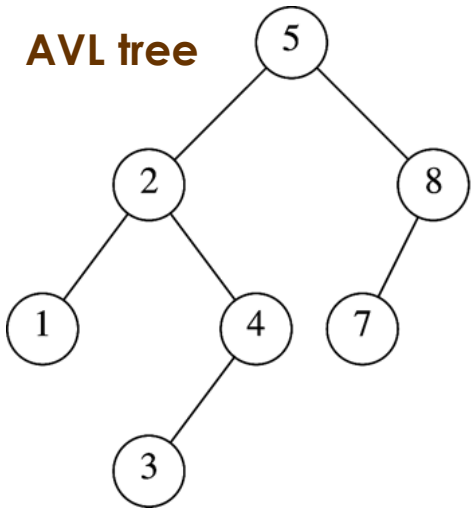
Maintaining a stricter condition (e.g above condition with d = 0) is difficult.

Note that our goal is to perform all the operations search, insert and delete in O(log N) time, including the operations involved in adjusting the tree to maintain the above balance condition.

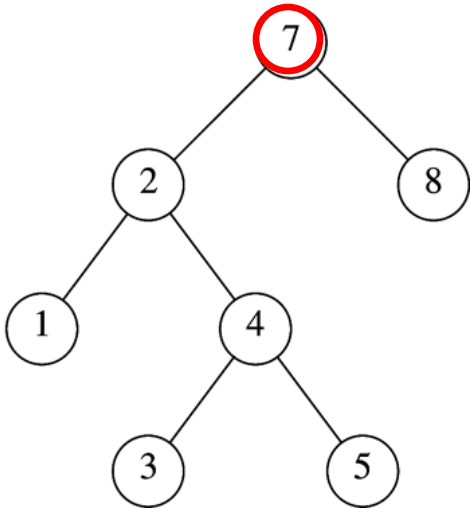
AVL Tree

- An AVL tree is a binary search tree in which
  - for *every* node in the tree, the height of the left and right subtrees differ by at most 1.
- Height of subtree: Max # of edges to a leaf
- Height of an empty subtree: -1
  - Height of one node: 0

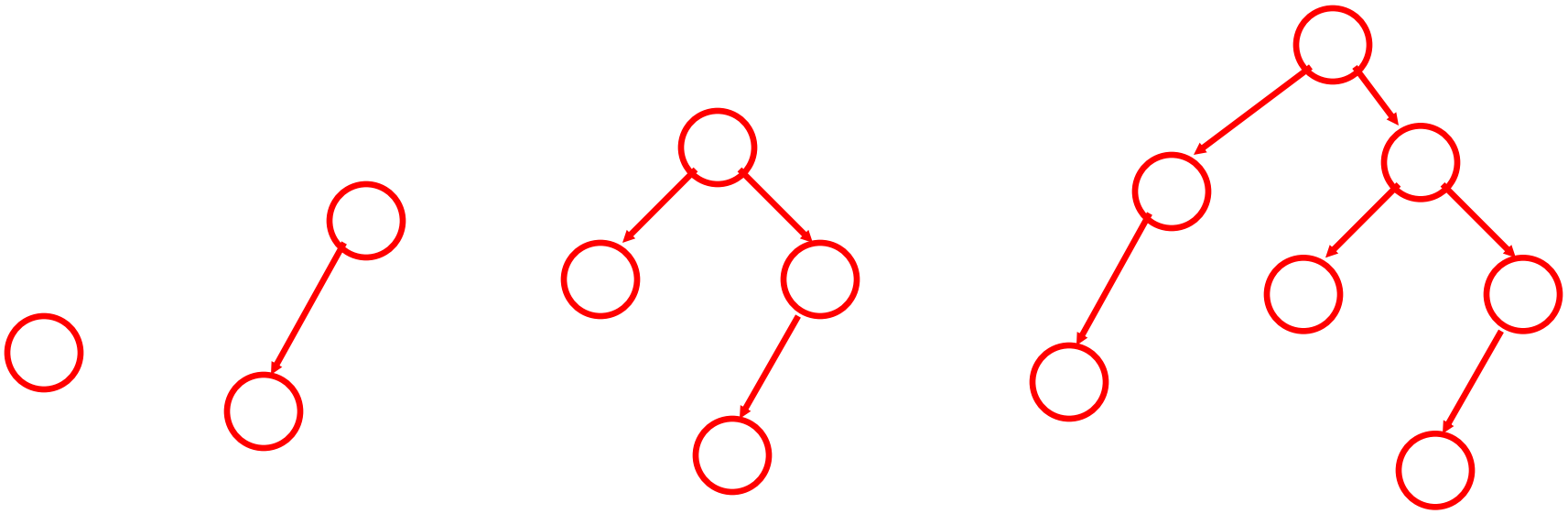
AVL tree



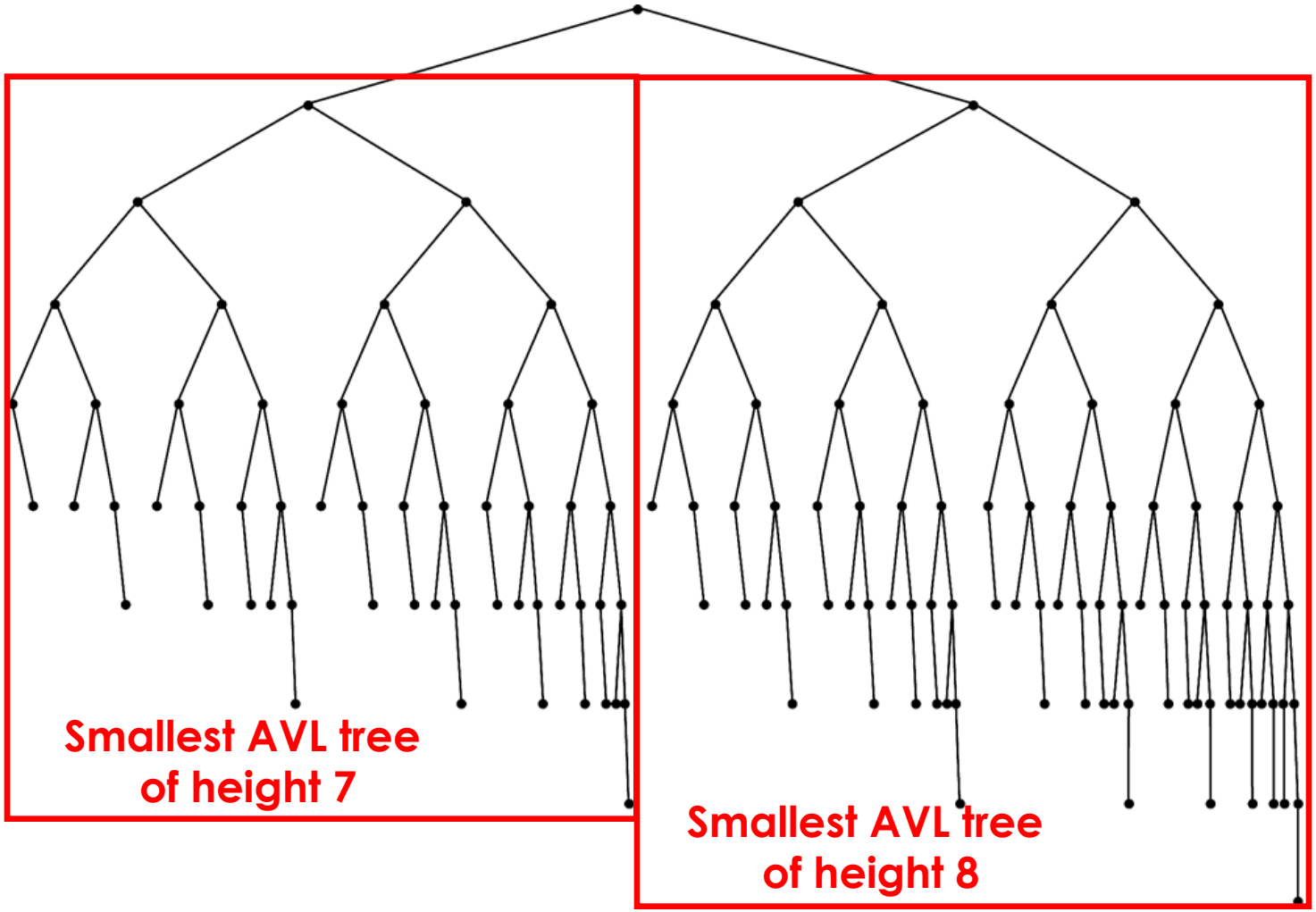
AVL tree property violated here



AVL Tree with Minimum Number of Nodes



$N_0 = 1$ $N_1 = 2$ $N_2 = 4$ $N_3 = N_1 + N_2 + 1 = 7$


Smallest AVL tree
of height 7
Smallest AVL tree
of height 8

Height of AVL Tree with N nodes

- Denote N_h the minimum number of nodes in an AVL tree of height h

- $N_0 = 1$, $N_1 = 2$ (base)
  $N_h = N_{h-1} + N_{h-2} + 1$ (recursive relation)

- $N > N_h = N_{h-1} + N_{h-2} + 1$
  $> 2\ N_{h-2} > 4\ N_{h-4} > \ldots > 2^i\ N_{h-2i}$

Height of AVL Tree with N nodes

- If h is even, let i=h/2-1. The equation becomes

$$N > 2^{h/2-1}N_2 \Rightarrow N > 2^{h/2-1}x4 \Rightarrow h=O(\log N)$$

- If h is odd, let i=(h-1)/2. The equation becomes

$$N > 2^{(h-1)/2}N_1 \Rightarrow N > 2^{(h-1)/2}x2 \Rightarrow h=O(\log N)$$

- Recall the cost of operations search, insert and delete is O(h).

- cost of searching = O(log N). (Just use the search algorithm for the binary search tree.)

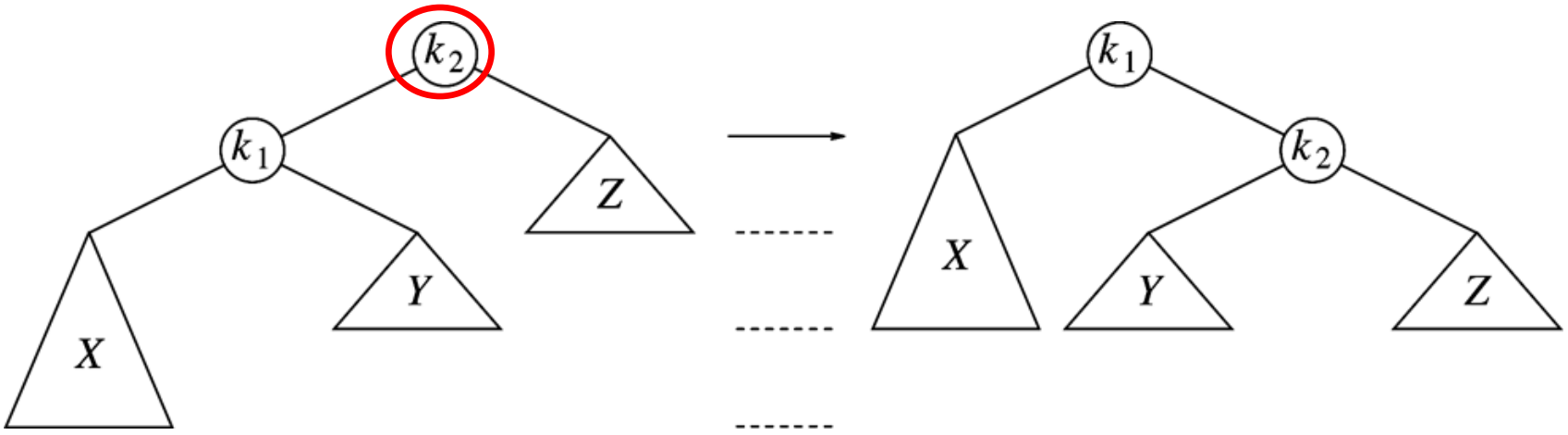
- But insert and delete are more complicated.

Rotations

- Rebalance of AVL tree are done with simple modification to tree, known as rotation
- Insertion occurs on the "outside" (i.e., left-left or right-right) is fixed by single rotation of the tree
- Insertion occurs on the "inside" (i.e., left-right or right-left) is fixed by double rotation of the tree

Single Rotation to Fix Case 1 (LL-rotation)

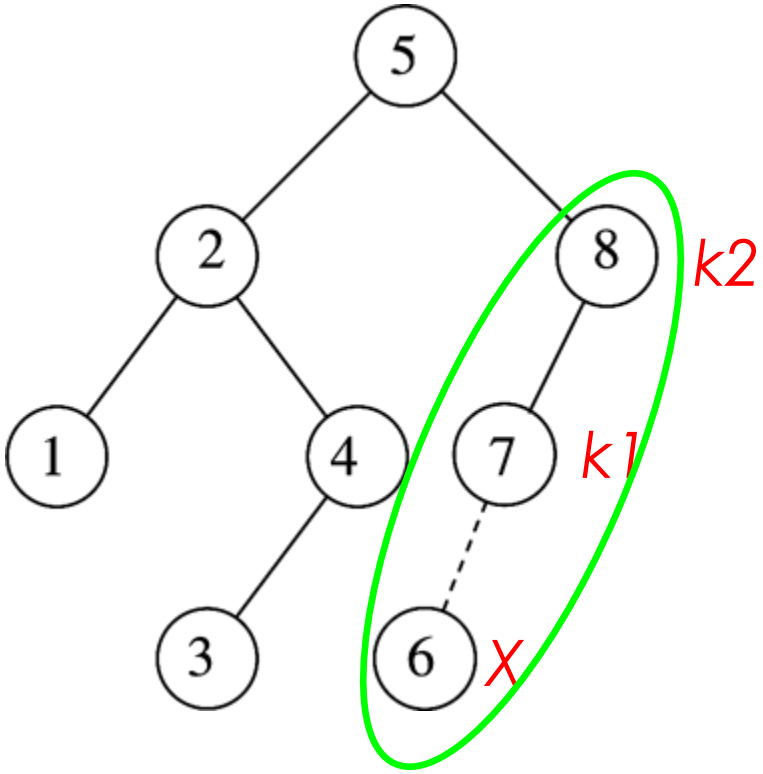
k2 is the pivot node

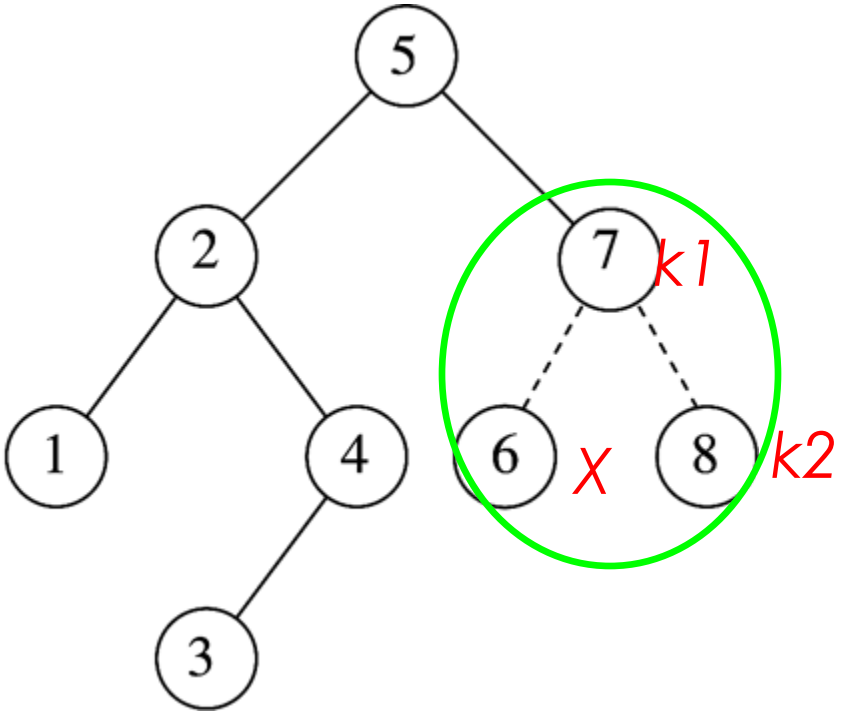


Questions:

- Can Y have the same height as the new X?
- Can Y have the same height as Z?

Single Rotation Case 1 Example





Informal proof/argument for the LL-case

We need to show the following:

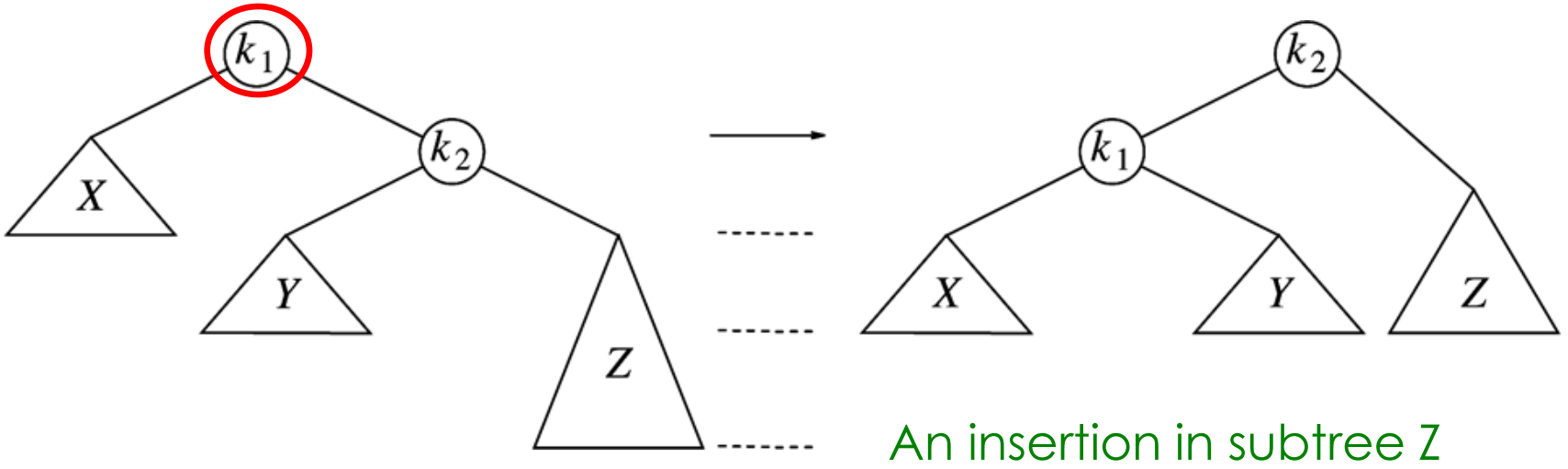
- After the rotation, the balance condition at the pivot node is restored and at all the nodes in the subtree rooted at the pivot.

- After the rotation, the balance condition is restored at all the ancestors of the pivot.

- We will prove both these informally.

Single Rotation to Fix Case 4 (RR-rotation)

k1 violates AVL tree balance condition



An insertion in subtree Z

- Case 4 is a symmetric case to case 1
- Insertion takes O(h) time (h = height of the tree), single rotation takes O(1) time. (including locating the pivot node). Details are not obvious, but see the code …

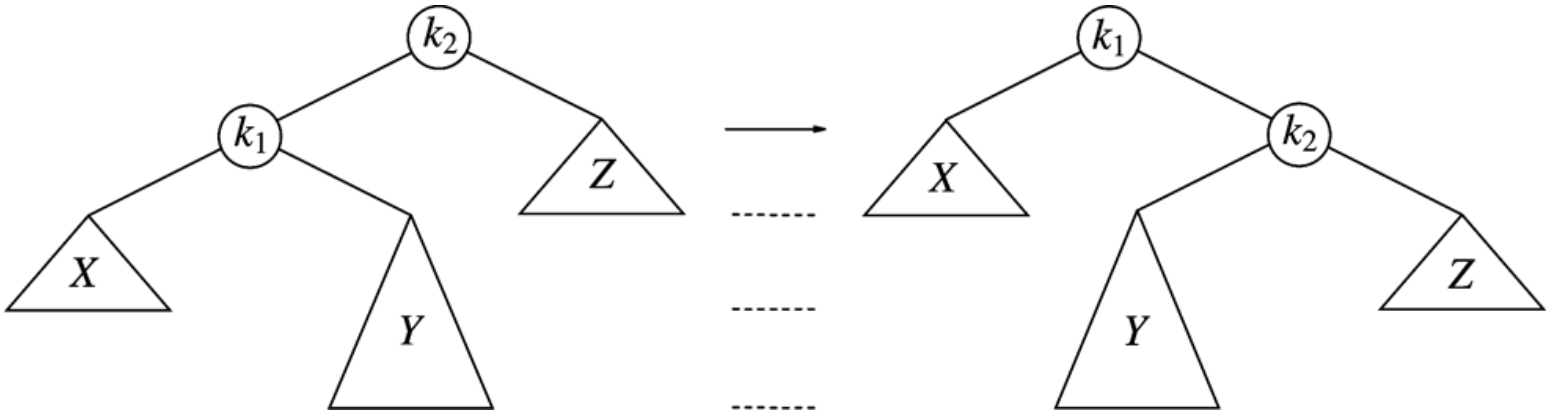
Code for left (single) rotation

```
  /* Rotate binary tree node with left child.
   * For AVL trees, this is a single rotation for case 1.
   * Update heights, then set new root.
   */
void rotateWithLeftChild( AvlNode * & k2 ) {
 AvlNode *k1 = k2->left;
 k2->left = k1->right;
 k1->right = k2;
 k2->height = max( height( k2->left ), height( k2->right ) ) + 1;
 k1->height = max( height( k1->left ), k2->height ) + 1;
 k2 = k1; }
```

- Height function

```
1  /**
2   * Return the height of node t or -1 if nullptr.
3   */
4  int height( AvlNode *t ) const
5  {
6      return t == nullptr ? -1 : t->height;
7  }
```

Single Rotation Fails to fix Case 2&3



Case 2: violation in k2 because of insertion in subtree Y

Single rotation result

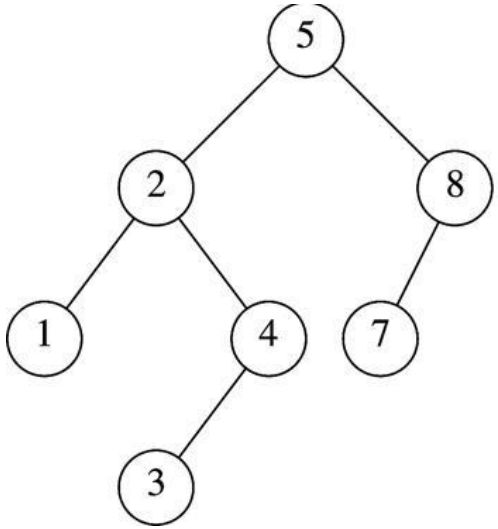
- Single rotation fails to fix case 2&3
- Take case 2 as an example (case 3 is a symmetry to it)
  - The problem is subtree Y is too deep
  - Single rotation doesn't make it any less deep

Code for double rotation

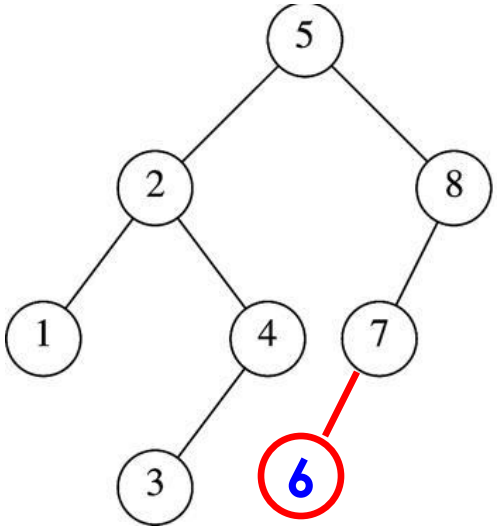
```
void doubleWithLeftChild( AvlNode * & k3 )
{
 rotateWithRightChild( k3->left );
 rotateWithLeftChild( k3 );
}
```

Insertion in AVL Tree

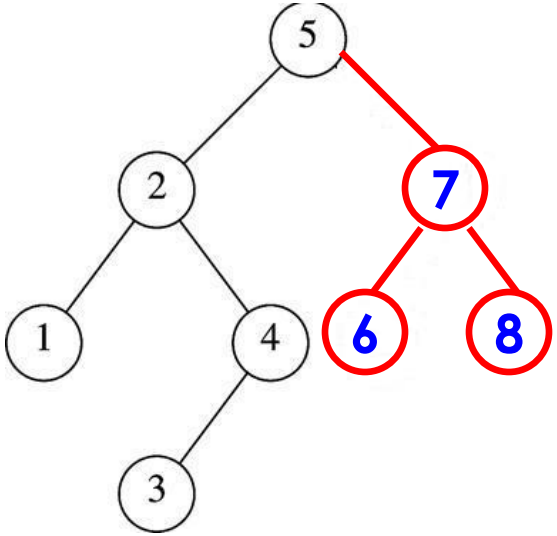
- Basically follows insertion strategy of binary search tree
  - But may cause violation of AVL tree property
- Restore the destroyed balance condition if needed



Original AVL tree



Insert 6
Property violated



Restore AVL property

Balancing a node in AVL tree

```
static const int ALLOWED_IMBALANCE = 1;

// Assume t is balanced or within one of being balanced
void balance( AvlNode * & t )
{
    if( t == nullptr )
        return;

    if( height( t->left ) - height( t->right ) > ALLOWED_IMBALANCE )
        if( height( t->left->left ) >= height( t->left->right ) )
            rotateWithLeftChild( t );
        else
            doubleWithLeftChild( t );
    else
    if( height( t->right ) - height( t->left ) > ALLOWED_IMBALANCE )
        if( height( t->right->right ) >= height( t->right->left ) )
            rotateWithRightChild( t );
        else
            doubleWithRightChild( t );

    t->height = max( height( t->left ), height( t->right ) ) + 1;
}
```

AVL tree insertion

```
/**
 * Internal method to insert into a subtree.
 * x is the item to insert.
 * t is the node that roots the subtree.
 * Set the new root of the subtree.
 */
void insert( const Comparable & x, AvlNode * & t )
{
    if( t == nullptr )
        t = new AvlNode{ x, nullptr, nullptr };
    else if( x < t->element )
        insert( x, t->left );
    else if( t->element < x )
        insert( x, t->right );

    balance( t );
}
```

AVL tree - deletion

```
/**
 * Internal method to remove from a subtree.
 * x is the item to remove.
 * t is the node that roots the subtree.
 * Set the new root of the subtree.
 */
void remove( const Comparable & x, AvlNode * & t )
{
    if( t == nullptr )
        return;   // Item not found; do nothing

    if( x < t->element )
        remove( x, t->left );
    else if( t->element < x )
        remove( x, t->right );
    else if( t->left != nullptr && t->right != nullptr )  // Two children
    {
        t->element = findMin( t->right  )->element;
        remove( t->element, t->right );
    }
    else
    {
        AvlNode *oldNode = t;
        t = ( t->left != nullptr ) ? t->left : t->right;
        delete oldNode;
    }

     balance( t );
}
```

Summary

- AVL trees with n nodes will have height at most $1.5 \log_2 n$

- To implement AVL tree, we use an extra field at each node that stores the height of the subtree rooted at the node.

- To insert a key, at most a constant number of pointer changes need to be performed. This is done by a single rotation or a double rotation at the pivot node.

- All dictionary operations can be performed in O(log n) time in the WORST-CASE using AVL trees.