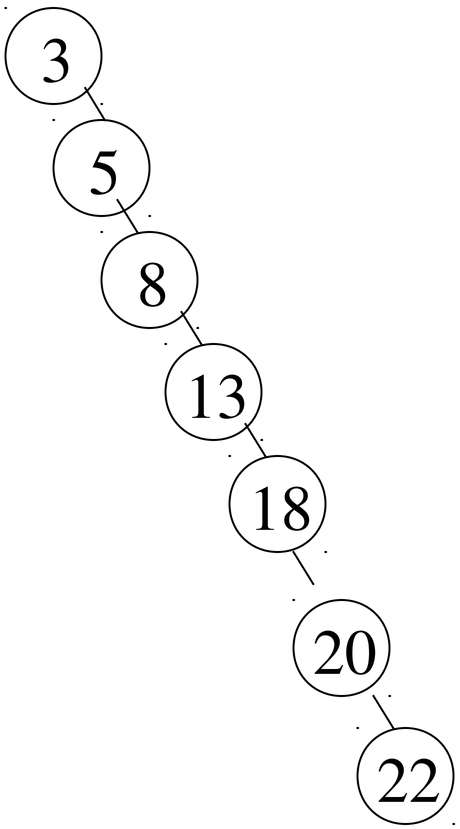
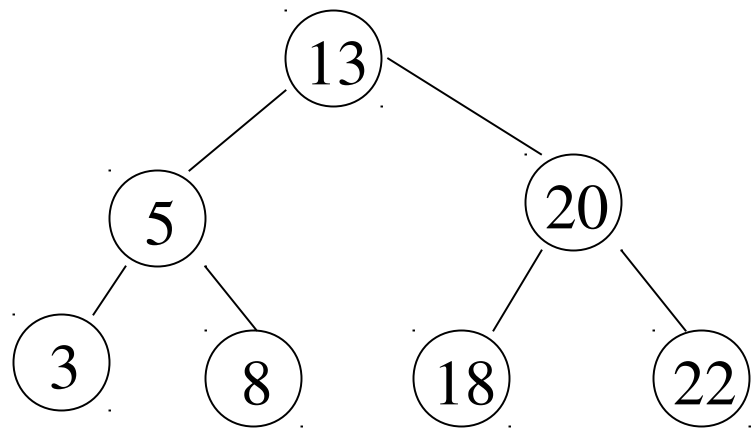
Trees 4: AVL Trees

- Section 4.4

Motivation

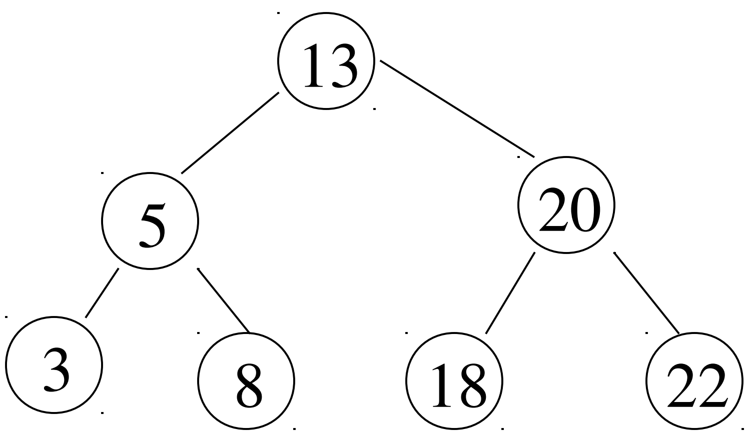
- When building a binary search tree, what type of trees would we like? Example: 3, 5, 8, 20, 18, 13, 22

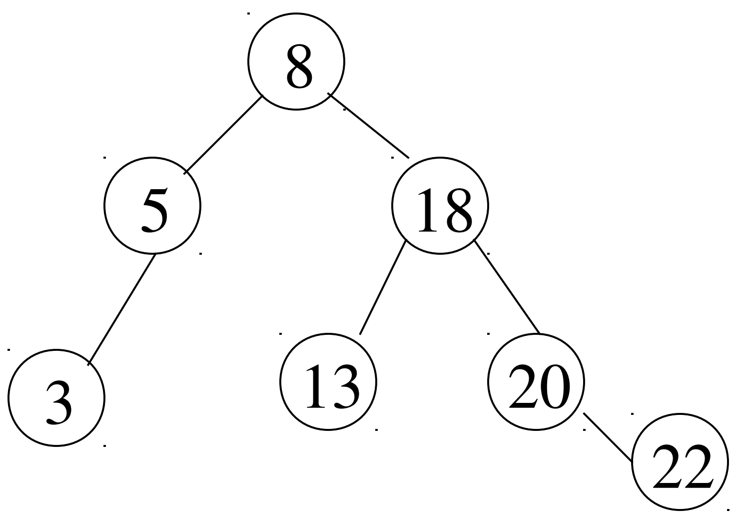




Motivation

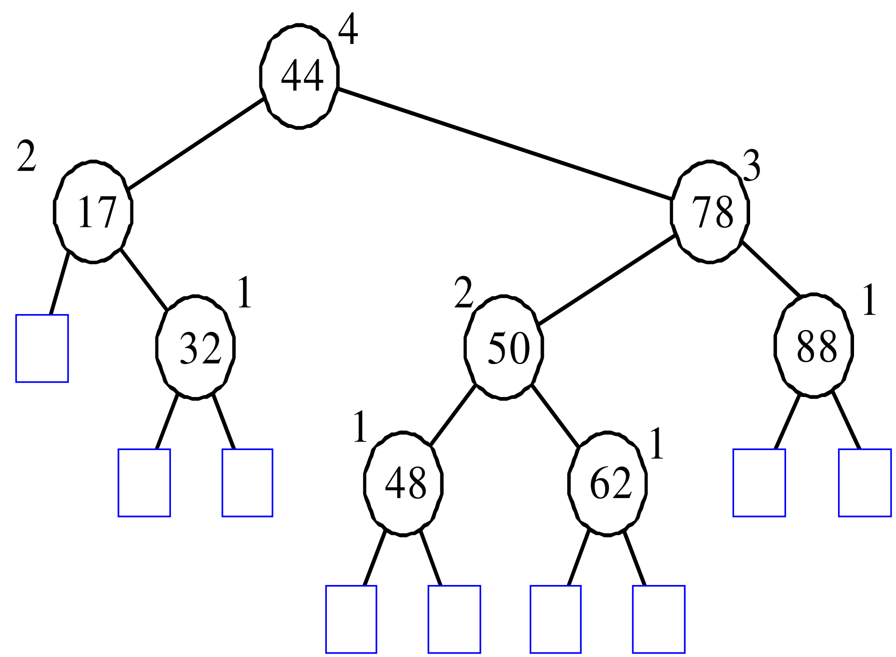
- Complete binary tree is hard to build when we allow dynamic insert and remove.
  - We want a tree that has the following properties
    - Tree height = O(log(N))
    - allows dynamic insert and remove with O(log(N)) time complexity.
  - The AVL tree is one of this kind of trees.





AVL (Adelson-Velskii and Landis) Trees

- An AVL Tree is a ***binary search tree*** such that for every internal node v of T, the *heights of the children of v can differ by at most 1.*

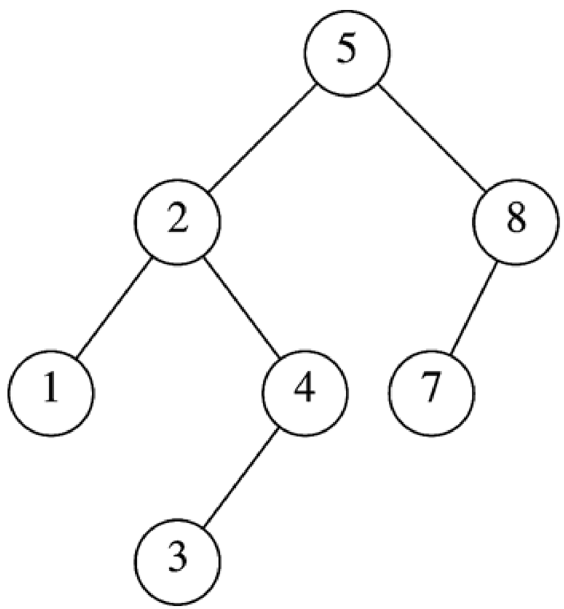


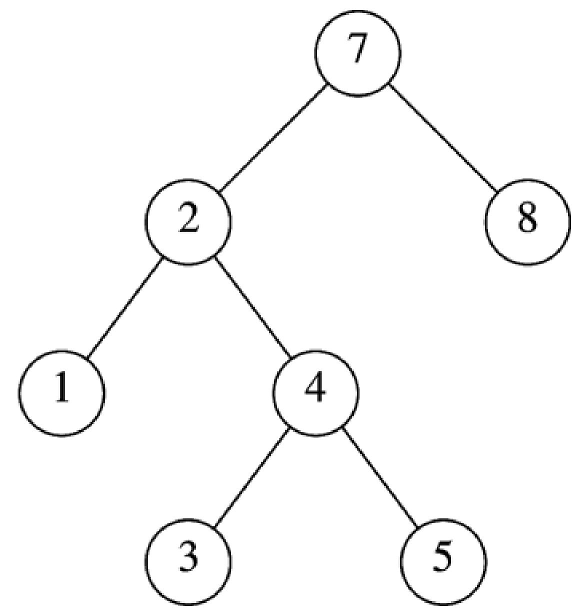
An example of an AVL tree where the heights are shown next to the nodes:

AVL (Adelson-Velskii and Landis) Trees

- AVL tree is a binary search tree with balance condition
  - To ensure depth of the tree is O(log(N))
  - And consequently, search/insert/remove complexity bound O(log(N))
- Balance condition
  - For every node in the tree, height of left and right subtree can differ by at most 1

Which is an AVL Tree?





Height of an AVL tree

- Theorem: The ***height*** of an AVL tree storing n keys is O(log n).
- **Proof**:
  - Let us bound **n(h),** the minimum number of internal nodes of an AVL tree of height h.
  - We easily see that n(0) = 1 and n(1) = 2
  - For h > 2, an AVL tree of height h contains the root node, one AVL subtree of height h-1 and another of height h-2 (at worst).
  - That is, n(h) >= 1 + n(h-1) + n(h-2)
  - Knowing n(h-1) > n(h-2), we get n(h) > 2n(h-2). So

    n(h) > 2n(h-2), n(h) > 4n(h-4), n(h) > 8n(n-6), … (by induction),

    n(h) > 2^in(h-2i)
  - Solving the base case we get: n(h) > $2^{h/2-1}$
  - Taking logarithms: h < 2log n(h) +2
  - Since n>=n(h), h < 2log(n)+2 and the height of an AVL tree is O(log n)

AVL Tree Insert and Remove

- Do binary search tree insert and remove
- The balance condition can be violated sometimes
  - Do something to fix it : rotations
  - After rotations, the balance of the whole tree is maintained

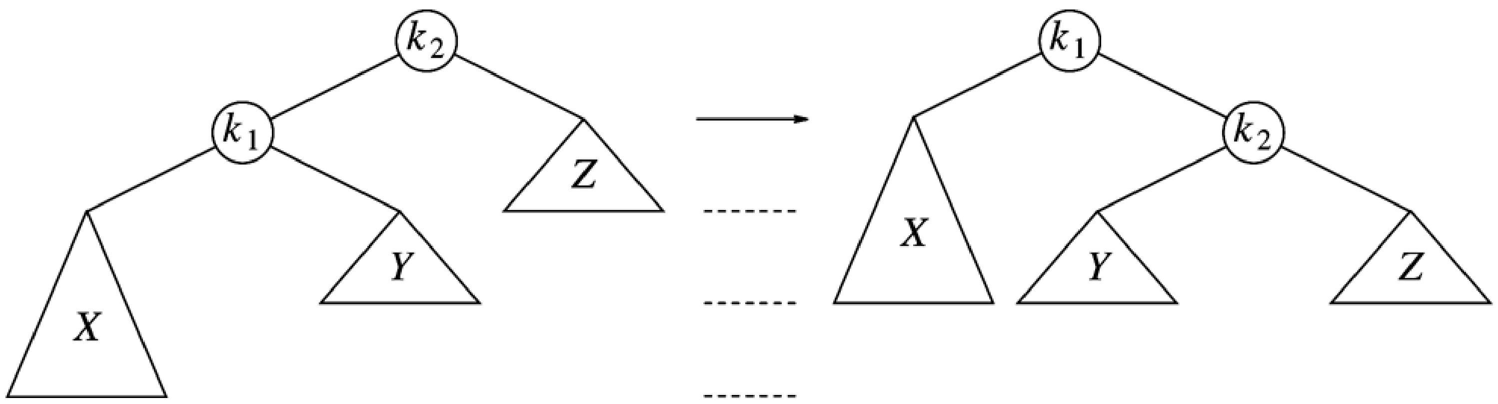
Balance Condition Violation

- If condition violated after a node insertion
  - Which nodes do we need to rotate?
  - Only nodes on path from insertion point to root may have their balance altered
- Rebalance the tree through rotation at the deepest node with balance violated
  - The entire tree will be rebalanced
- Violation cases at node k (deepest node)
  1. An insertion into left subtree of left child of k
  2. An insertion into right subtree of left child of k
  3. An insertion into left subtree of right child of k
  4. An insertion into right subtree of right child of k
  - Cases 1 and 4 equivalent
    - Single rotation to rebalance
  - Cases 2 and 3 equivalent
    - Double rotation to rebalance

AVL Trees Complexity

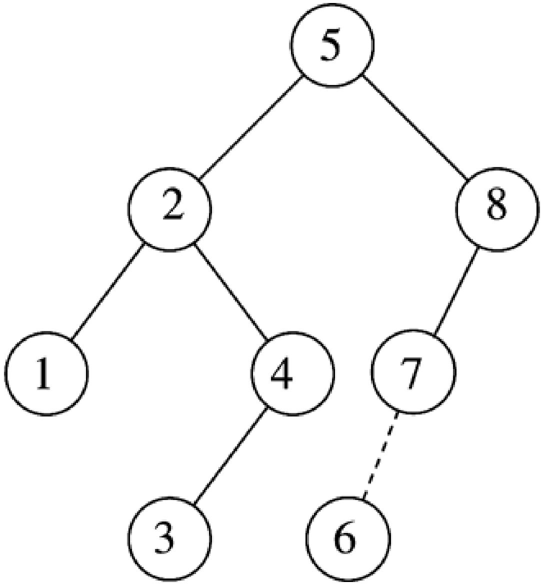
- Overhead
  - Extra space for maintaining height information at each node
  - Insertion and deletion become more complicated, but still O(log N)
- Advantage
  - Worst case O(log(N)) for insert, delete, and search

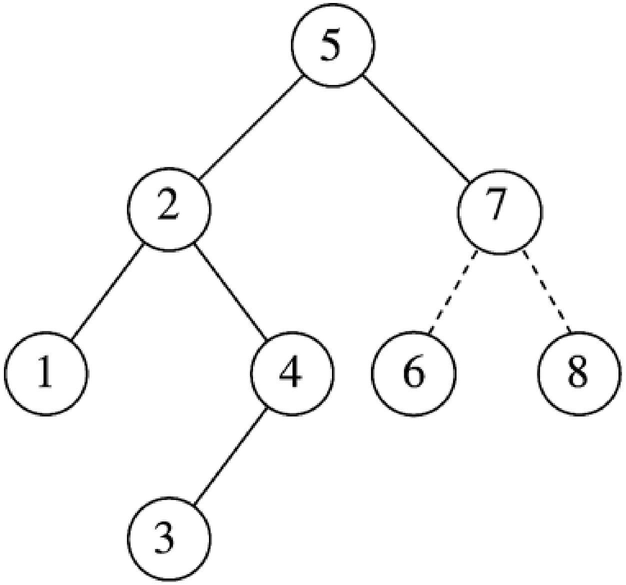
Single Rotation (Case 1)



- Replace node k_2 by node k_1
- Set node k_2 to be right child of node k_1
- Set subtree Y to be left child of node k_2
- Case 4 is similar

Example

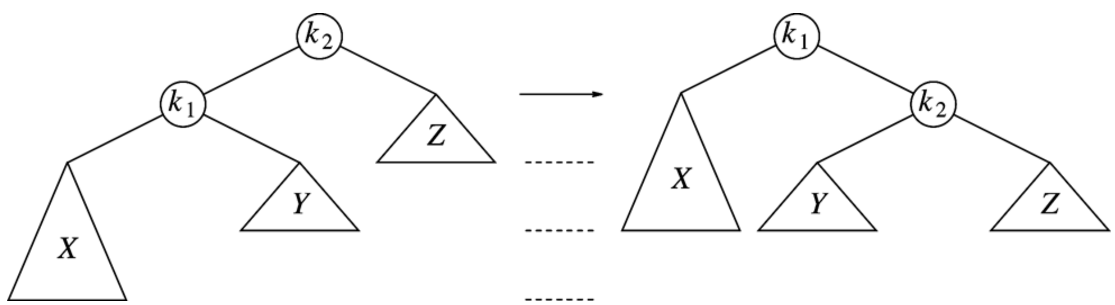




- After inserting 6
  - Balance condition at node 8 is violated

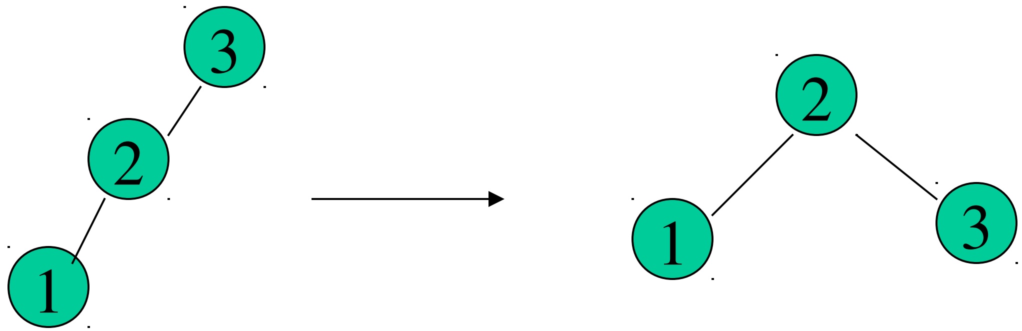
Single Rotation (Case 1)

```
/**
 * 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;
}
```



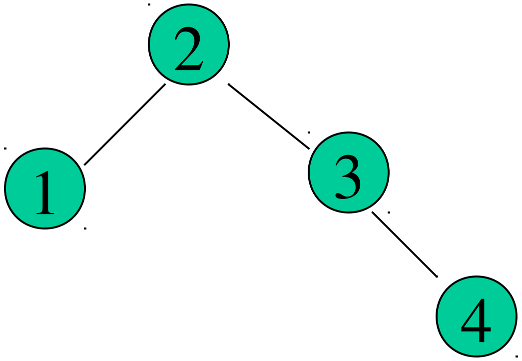
Example

- Inserting 3, 2, 1, and then 4 to 7 sequentially into empty AVL tree

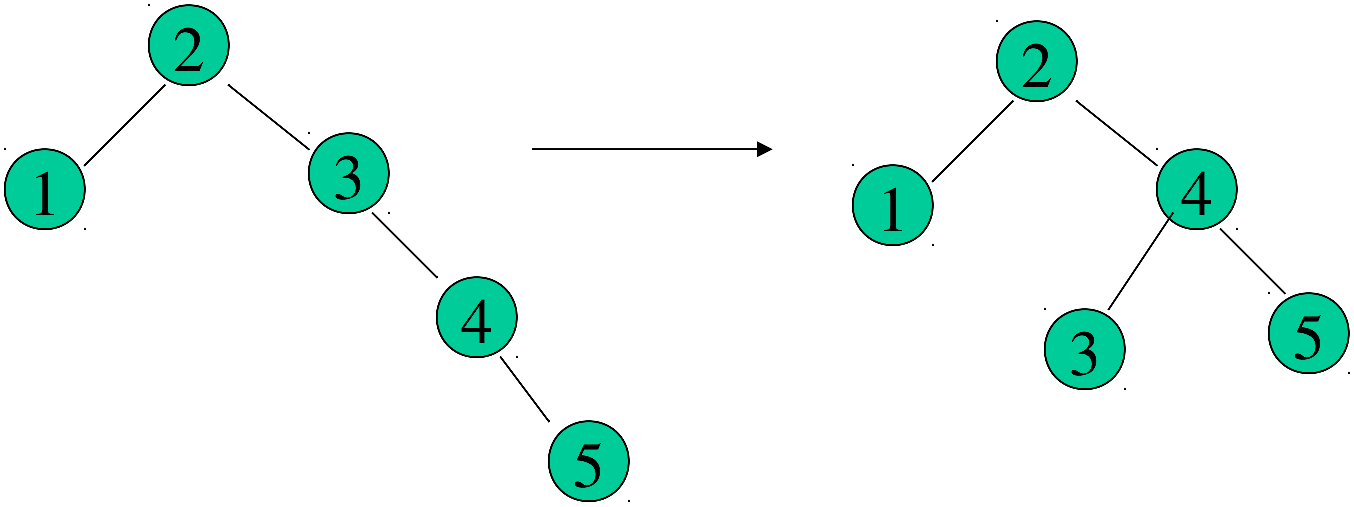


Example (Cont'd)

- Inserting 4

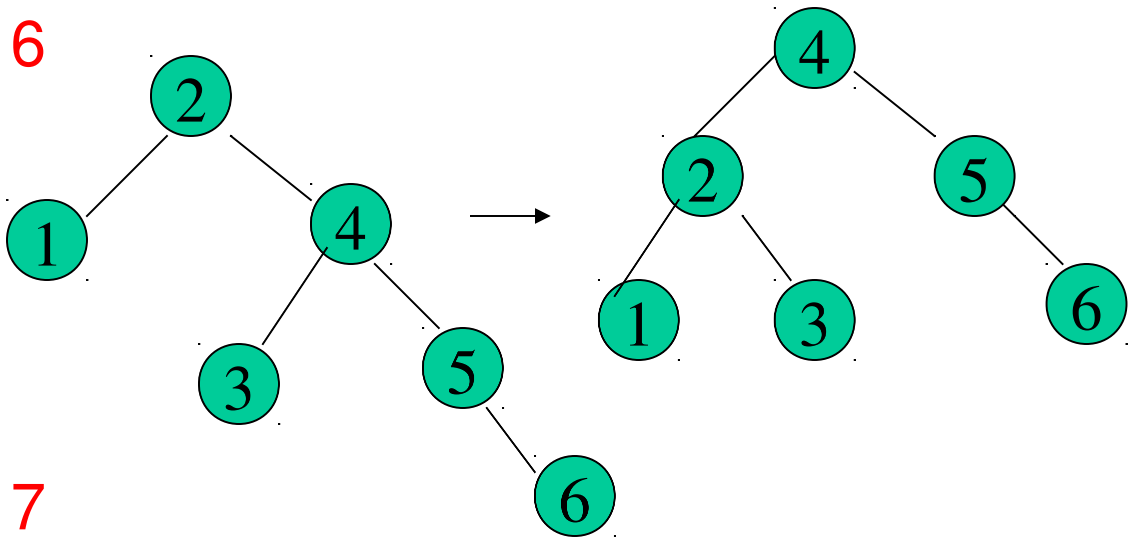


- Inserting 5

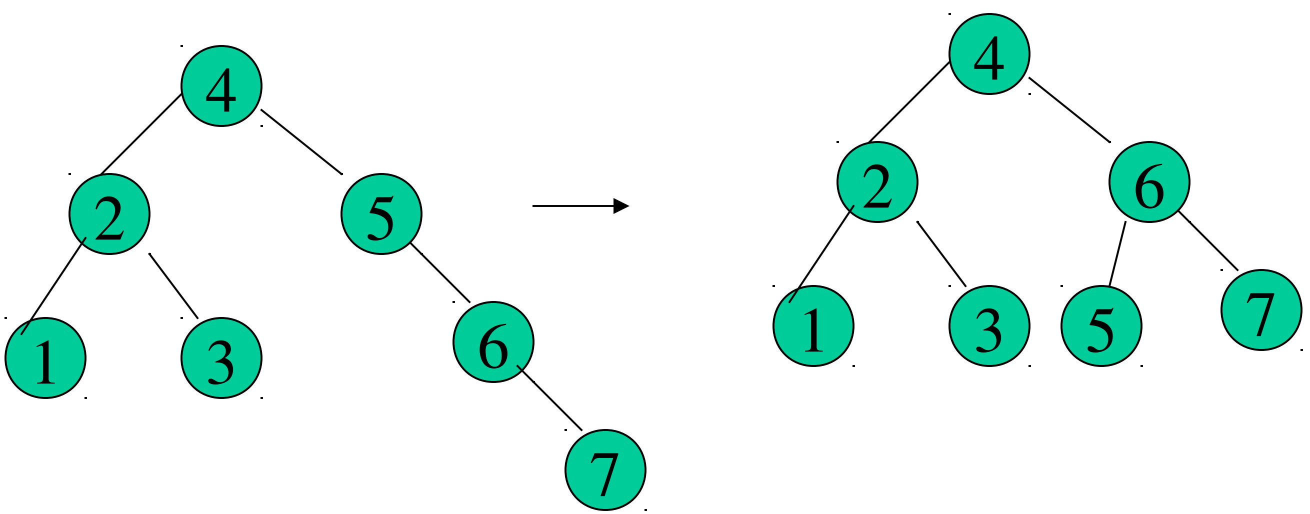


Example (Cont'd)

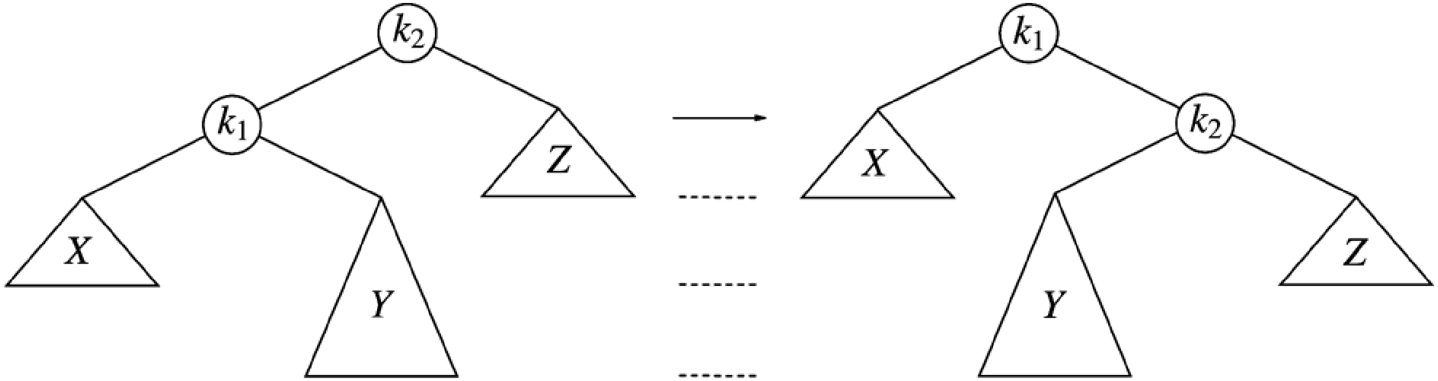
- Inserting 6



- Inserting 7

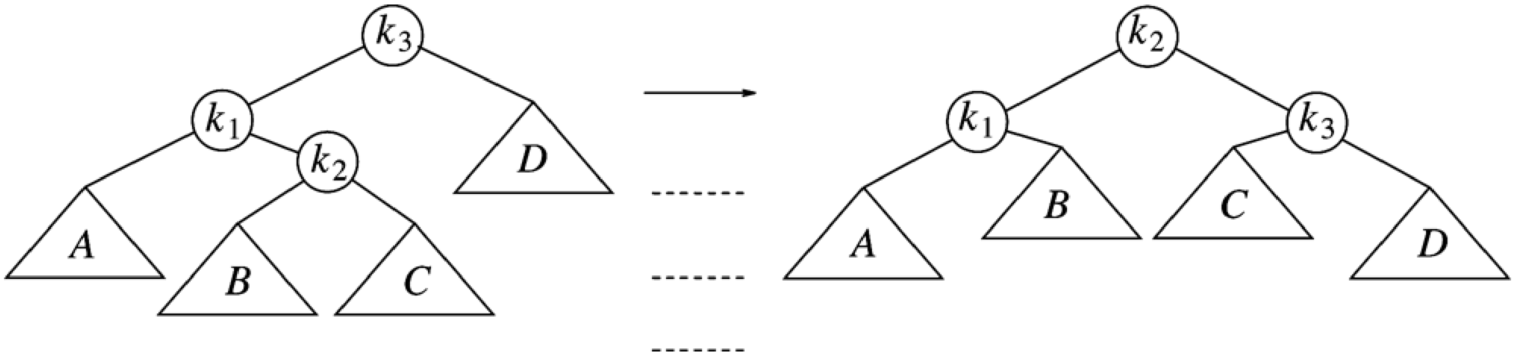


Single Rotation Will Not Work for the Other Case



- For case 2
- After single rotation, k_1 still not balanced
- Double rotations needed for case 2 and case 3

Double Rotation (Case 2)

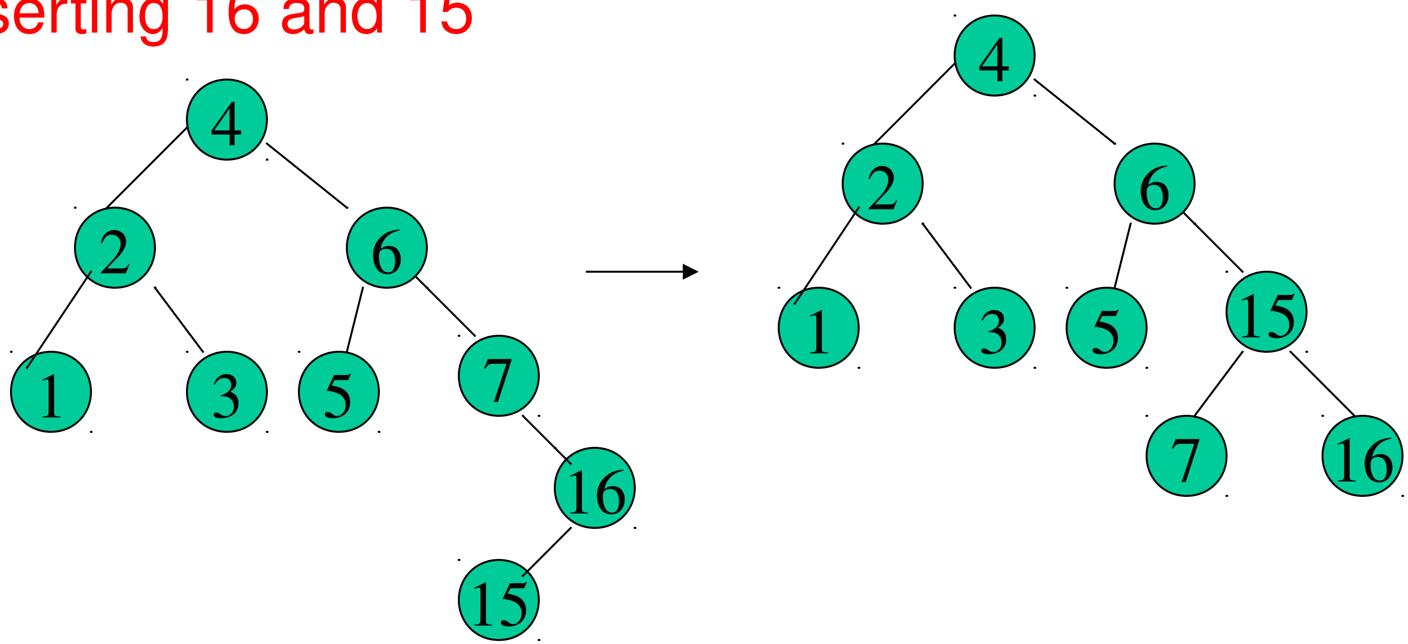


- Left-right double rotation to fix case 2
- First rotate between $\mathbf{k_1}$ and $\mathbf{k_2}$
- Then rotate between $\mathbf{k_2}$ and $\mathbf{k_3}$
- Case 3 is similar

Example

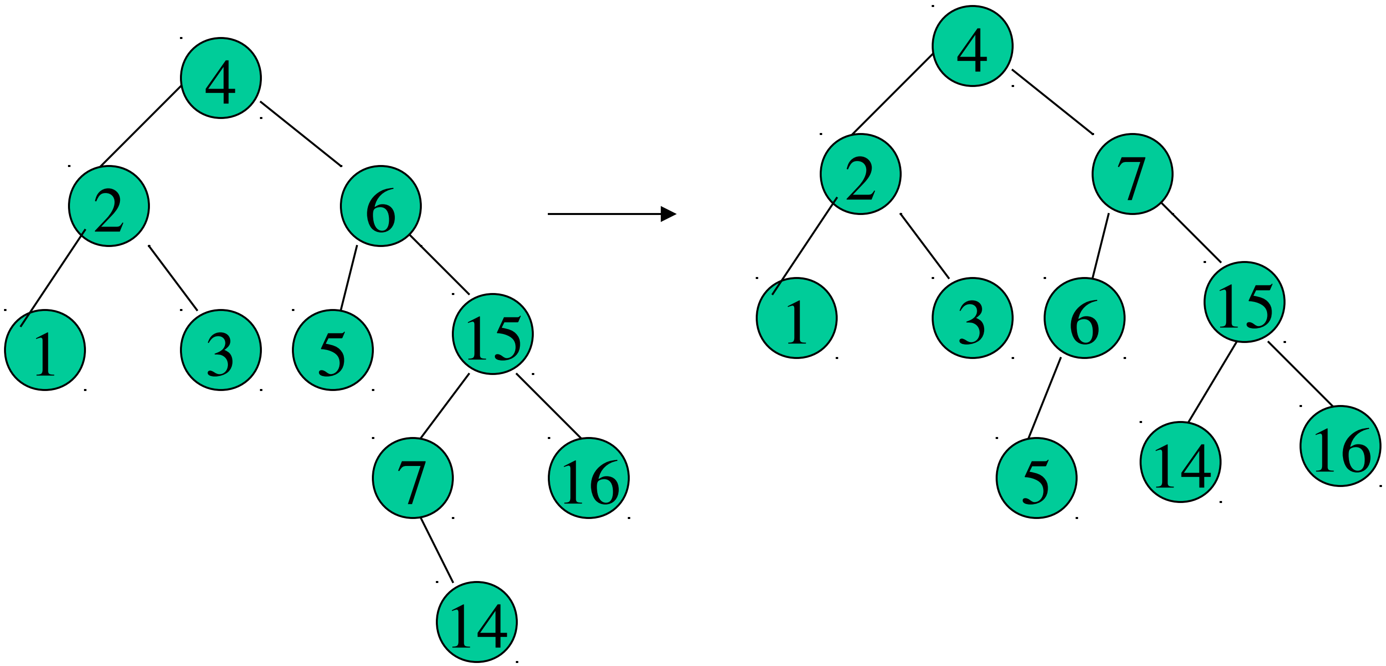
- Continuing the previous example by inserting
  - 16 down to 10, and then 8 and 9

- Inserting 16 and 15



Example (Cont'd)

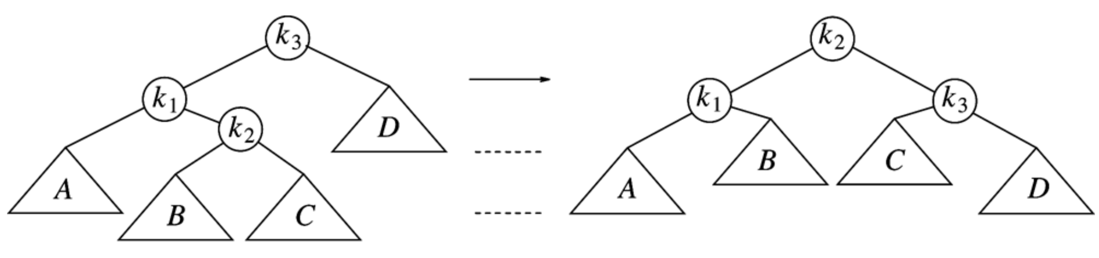
- Inserting 14



- Other cases as exercises

Double Rotation (Case 2)

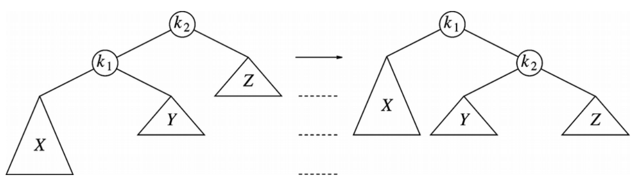
```
/**
 * Double rotate binary tree node: first left child
 * with its right child; then node k3 with new left child.
 * For AVL trees, this is a double rotation for case 2.
 * Update heights, then set new root.
 */
void doubleWithLeftChild( AvlNode * & k3 )
{
    rotateWithRightChild( k3->left );
    rotateWithLeftChild( k3 );
}
```



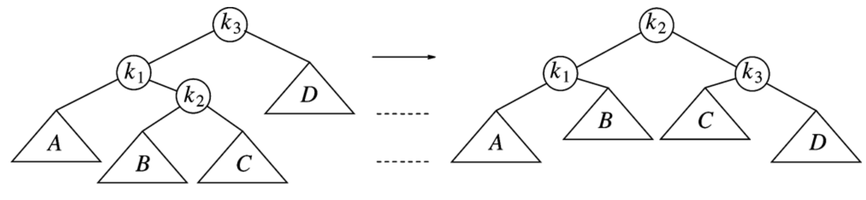
Summary

Violation cases at node k (deepest node)

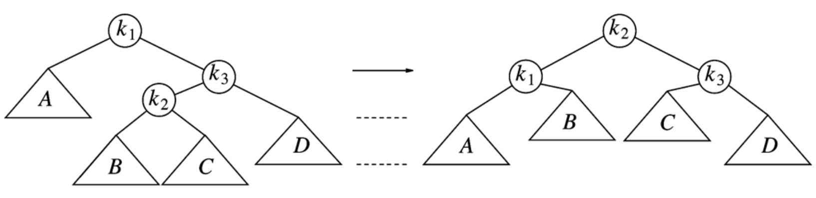
1. An insertion into left subtree of left child of k
2. An insertion into right subtree of left child of k
3. An insertion into left subtree of right child of k
4. An insertion into right subtree of right child of k



Case 1



Case 2



Case 3

Case 4?

Implementation of AVL Tree

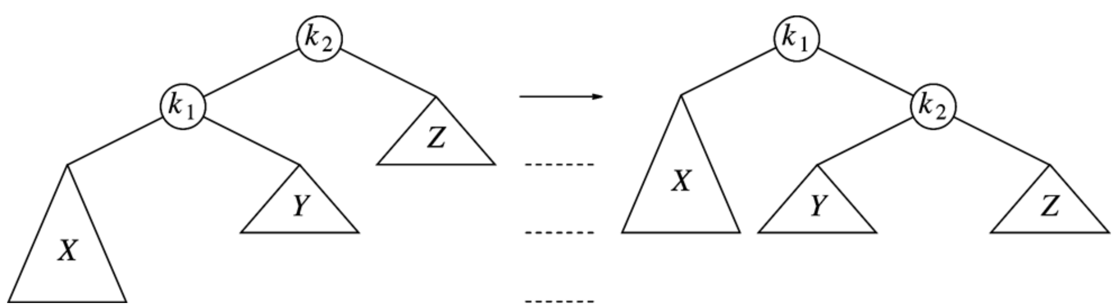
```
struct AvlNode
{
    Comparable element;
    AvlNode   *left;
    AvlNode   *right;
    int       height;

    AvlNode( const Comparable & theElement, AvlNode *lt,
                                            AvlNode *rt, int h = 0 )
      : element( theElement ), left( lt ), right( rt ), height( h )
};
```

```
/**
 * Return the height of node t or -1 if NULL.
 */
int height( AvlNode *t ) const
{
    return t == NULL ? -1 : t->height;
}
```

```
/**
 * 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 == NULL )
        t = new AvlNode( x, NULL, NULL );
    else if( x < t->element )
    {
        insert( x, t->left );
        if( height( t->left ) - height( t->right ) == 2 )
            if( x < t->left->element )
                rotateWithLeftChild( t );      ← Case 1
            else
                doubleWithLeftChild( t );      ← Case 2
    }
    else if( t->element < x )
    {
        insert( x, t->right );
        if( height( t->right ) - height( t->left ) == 2 )
            if( t->right->element < x )
                rotateWithRightChild( t );     ← Case 4
            else
                doubleWithRightChild( t );     ← Case 3
    }
    else
        ;  // Duplicate; do nothing
    t->height = max( height( t->left ), height( t->right ) ) + 1;
}
```

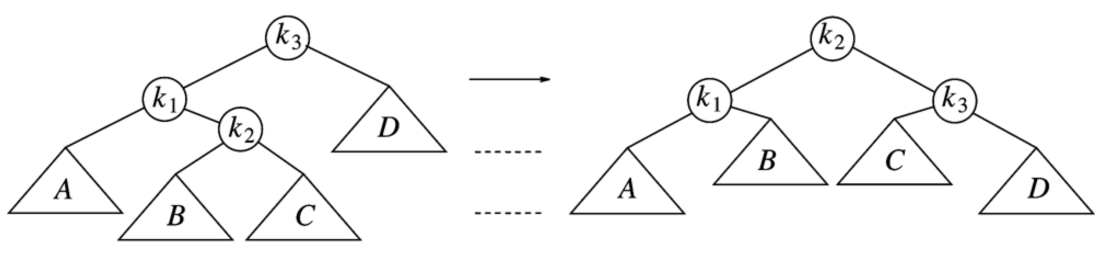
Single Rotation (Case 1)

```
/**
 * 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;
}
```

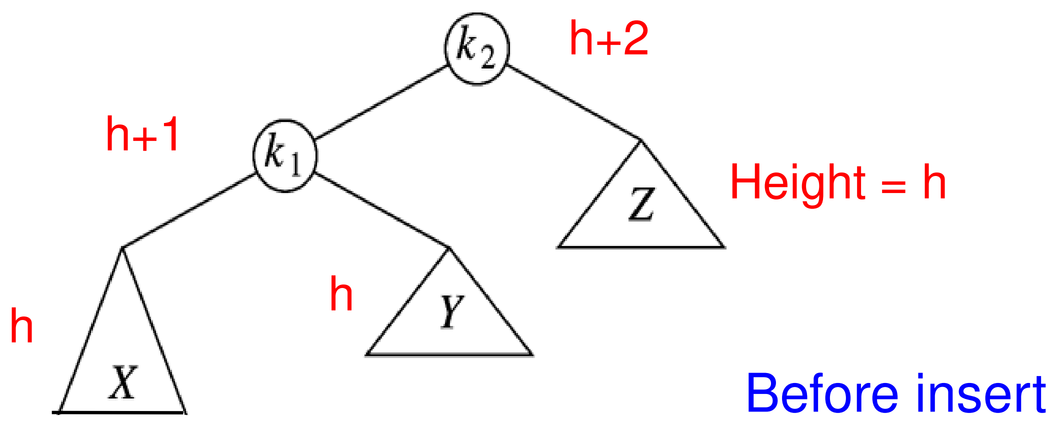


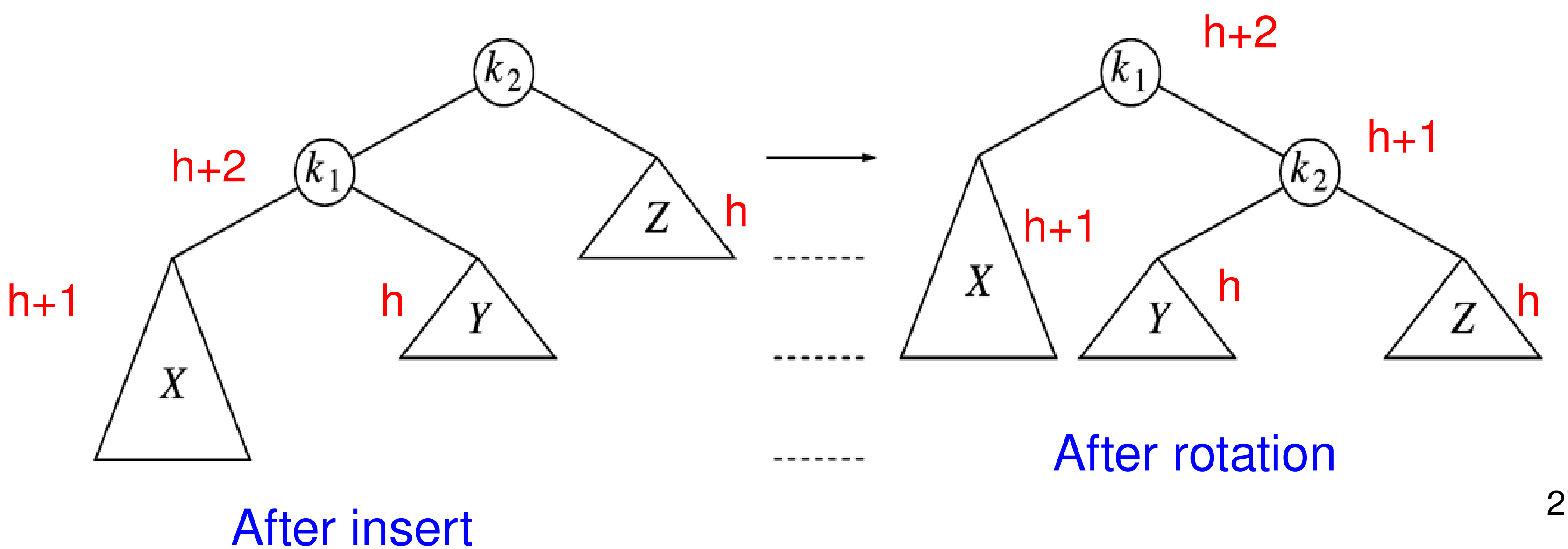
Double Rotation (Case 2)

```
/**
 * Double rotate binary tree node: first left child
 * with its right child; then node k3 with new left child.
 * For AVL trees, this is a double rotation for case 2.
 * Update heights, then set new root.
 */
void doubleWithLeftChild( AvlNode * & k3 )
{
    rotateWithRightChild( k3->left );
    rotateWithLeftChild( k3 );
}
```

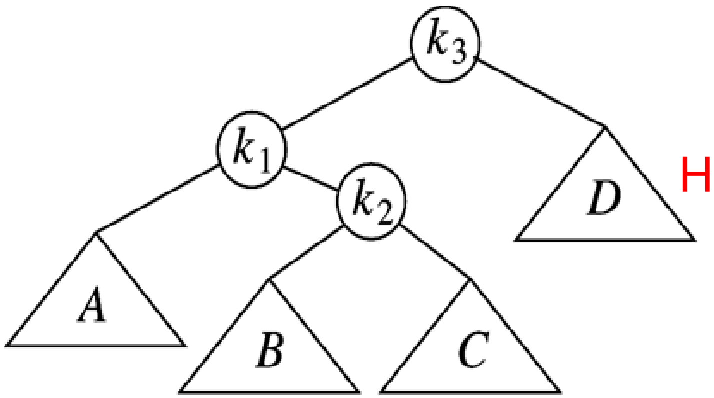


Review Insertion -- Case 1





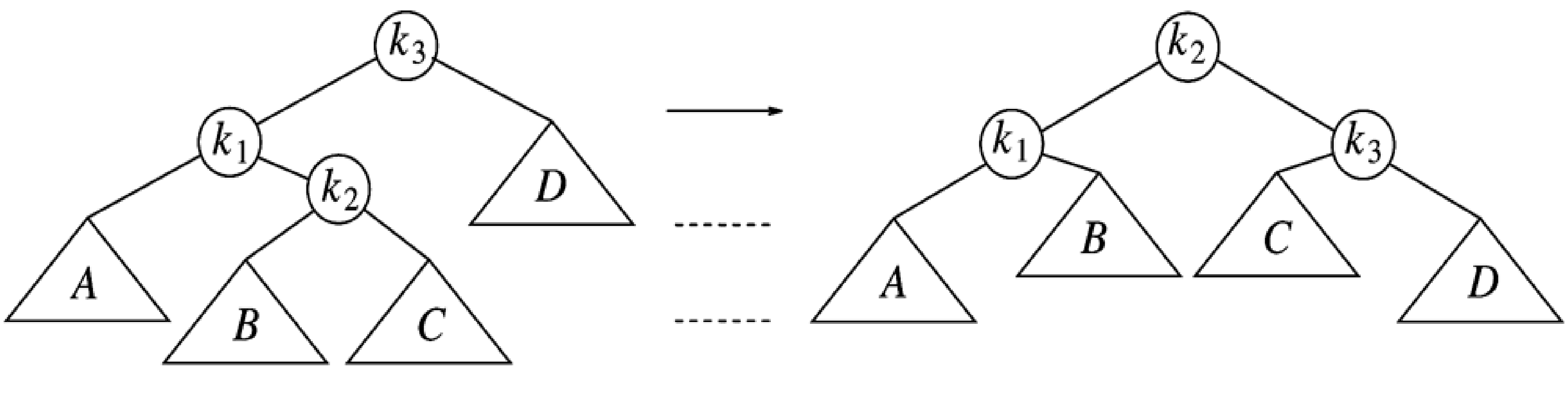
Review Insertion -- Case 2



Height = h

Determine all heights

Before insert

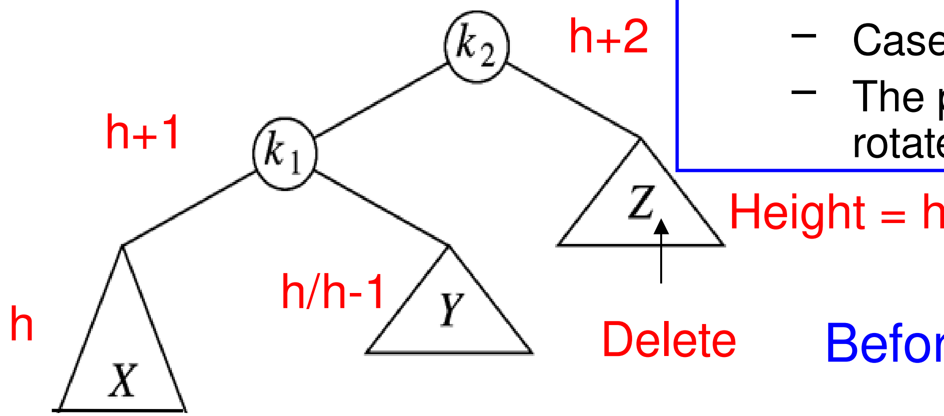


After insert

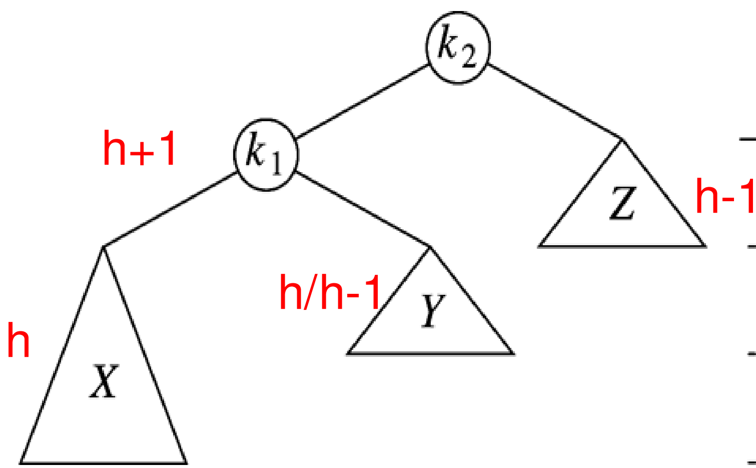
After double rotation

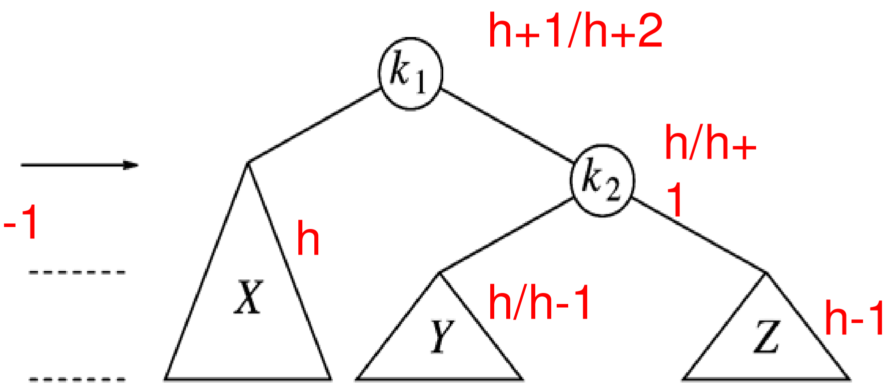
Delete -- Case 1

- Consider deepest unbalanced node
  - Case 1: Left child's left side is too high
  - Case 4: Right child's right side is too high
  - The parents may need to be recursively rotated



Before Deletion



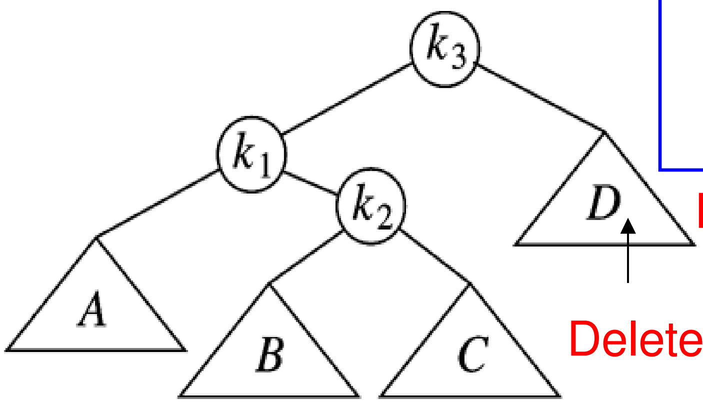


After single rotation

After delete

Delete -- Case 2

- Consider deepest unbalanced node
  - Case 2: Left child's right side is too high
  - Case 3: Right child's left side is too high
  - The parents may need to be recursively rotated

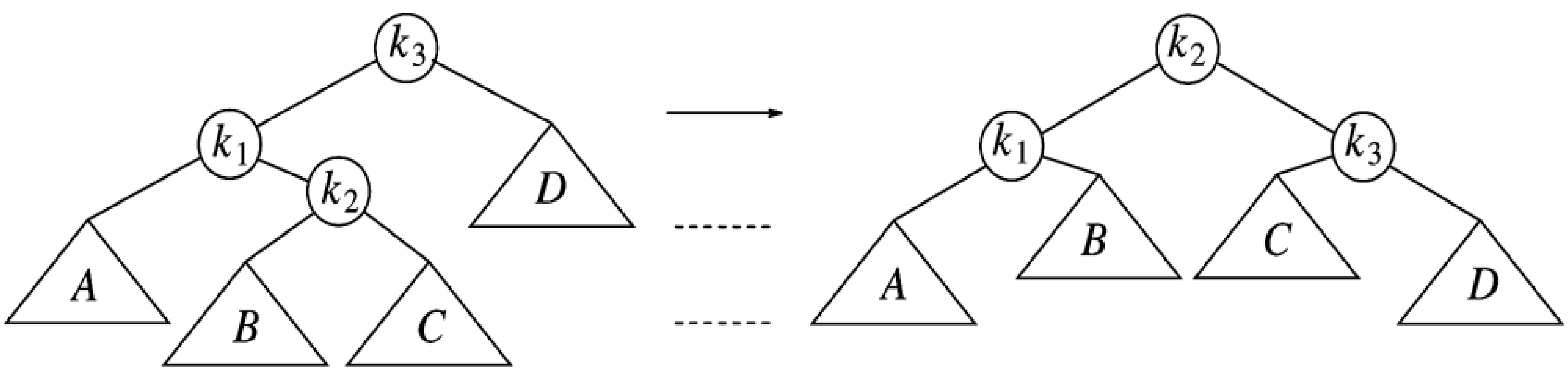


Height = h

Delete

Determine all heights

Before Deletion



After Delete

After double rotation