AVLNode class

```
class AvlNode<AnyType>{
1       AvlNode( AnyType theElement ){
2               this( theElement, null, null );
        }

        AvlNode( AnyType theElement, AvlNode<AnyType> lt, AvlNode<AnyType> rt ){
1               element = theElement;
2               left = lt;
3               right = rt;
4               height = 0;
        }

1       AnyType element;            // The data in the node
2       AvlNode<AnyType> left;      // Left child
3       AvlNode<AnyType> right;     // Right child
4       int height;                 // Height
}
```

AVLTree class

```
class AVLTree{
1       private static int ALLOWED_IMBALANCE = 1;

2       AvlNode<Integer> root;

3       static int debug = 0;
}
```

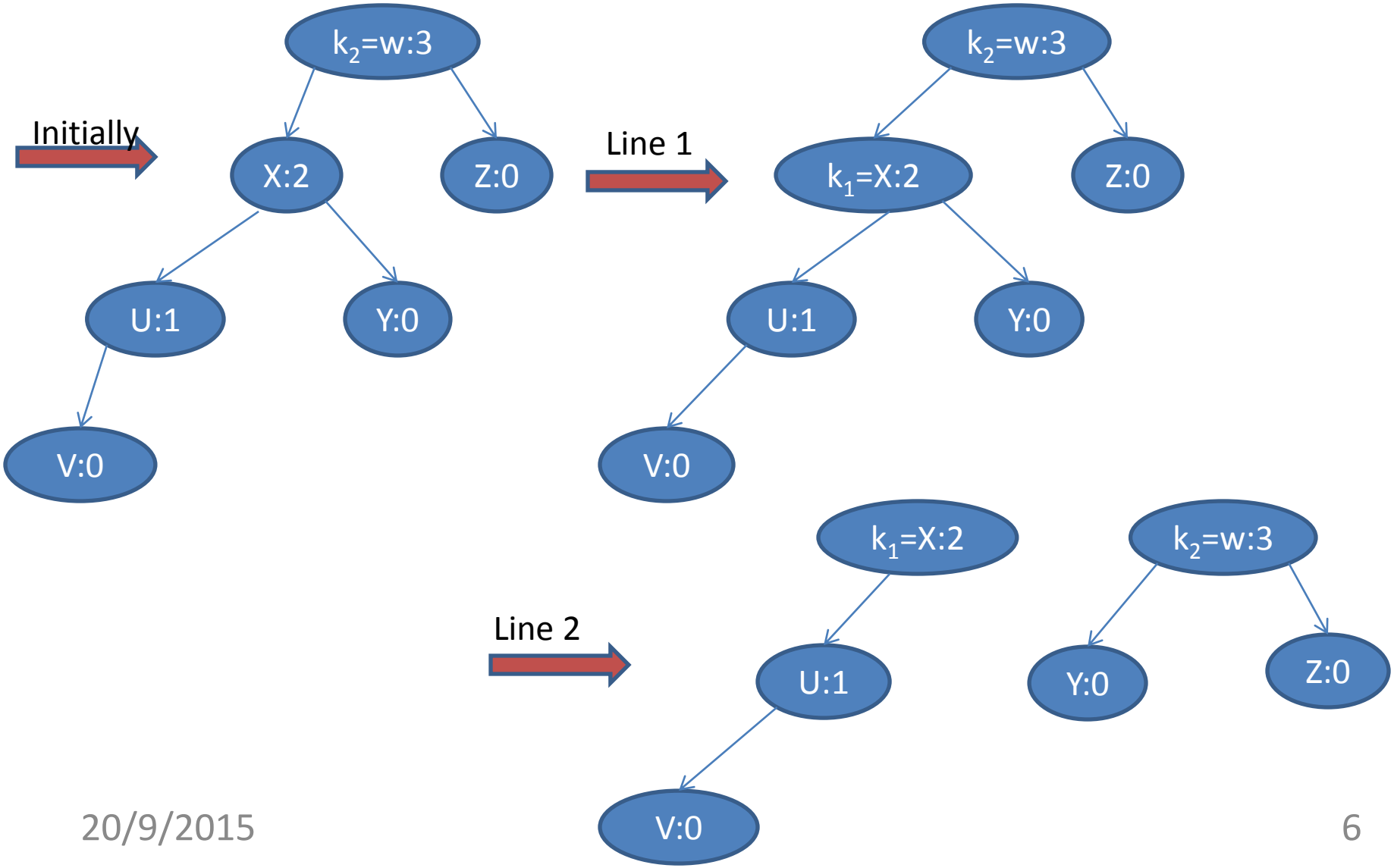
height(AvlNode<Integer> t)

```
/*
        Given an AvlNode it returns height of that node
*/
private int height( AvlNode<Integer> t ){
1       return t == null ? -1 : t.height;
}
```

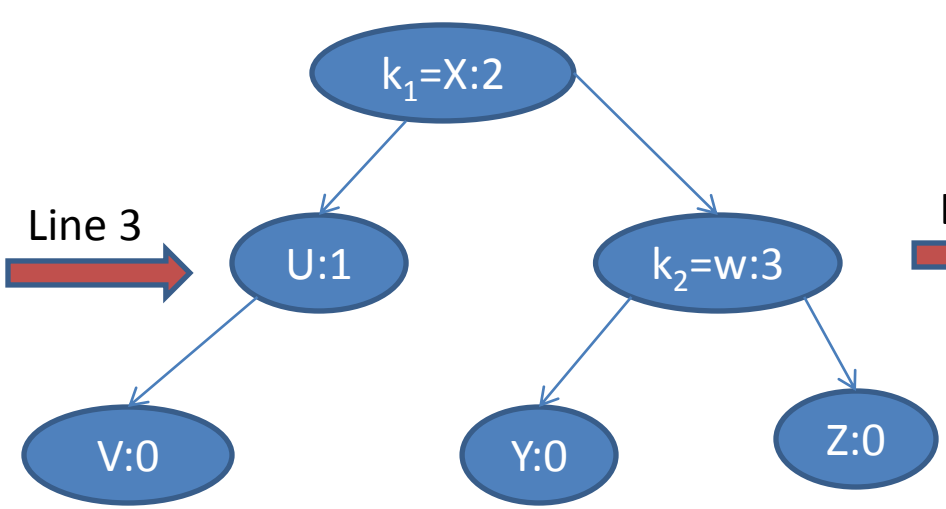
rotateWithLeftChild(AvlNode<Integer> k2)

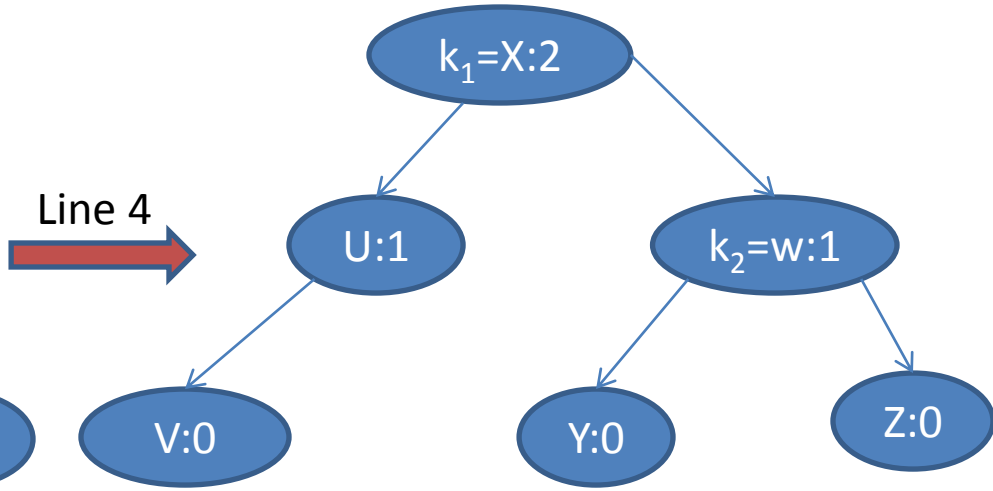
```
/**
*       Rotate binary tree node with left child.
*       For AVL trees, this is a single rotation for case 1.
*       Update heights, then return new root.
*/
private AvlNode<Integer> rotateWithLeftChild( AvlNode<Integer> k2 ){
1           AvlNode<Integer> k1 = k2.left;
2           k2.left = k1.right;
3           k1.right = k2;
4           k2.height = Math.max( height( k2.left ), height( k2.right ) ) + 1;
5           k1.height = Math.max( height( k1.left ), k2.height ) + 1;
6           return k1;
}
```

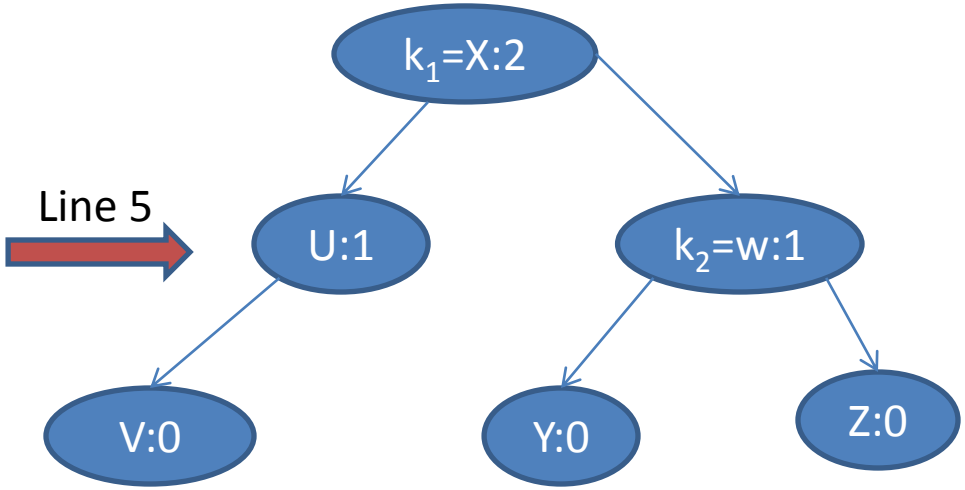
rotateWithLeftChild(k2 = w)



rotateWithLeftChild(k2 = w)



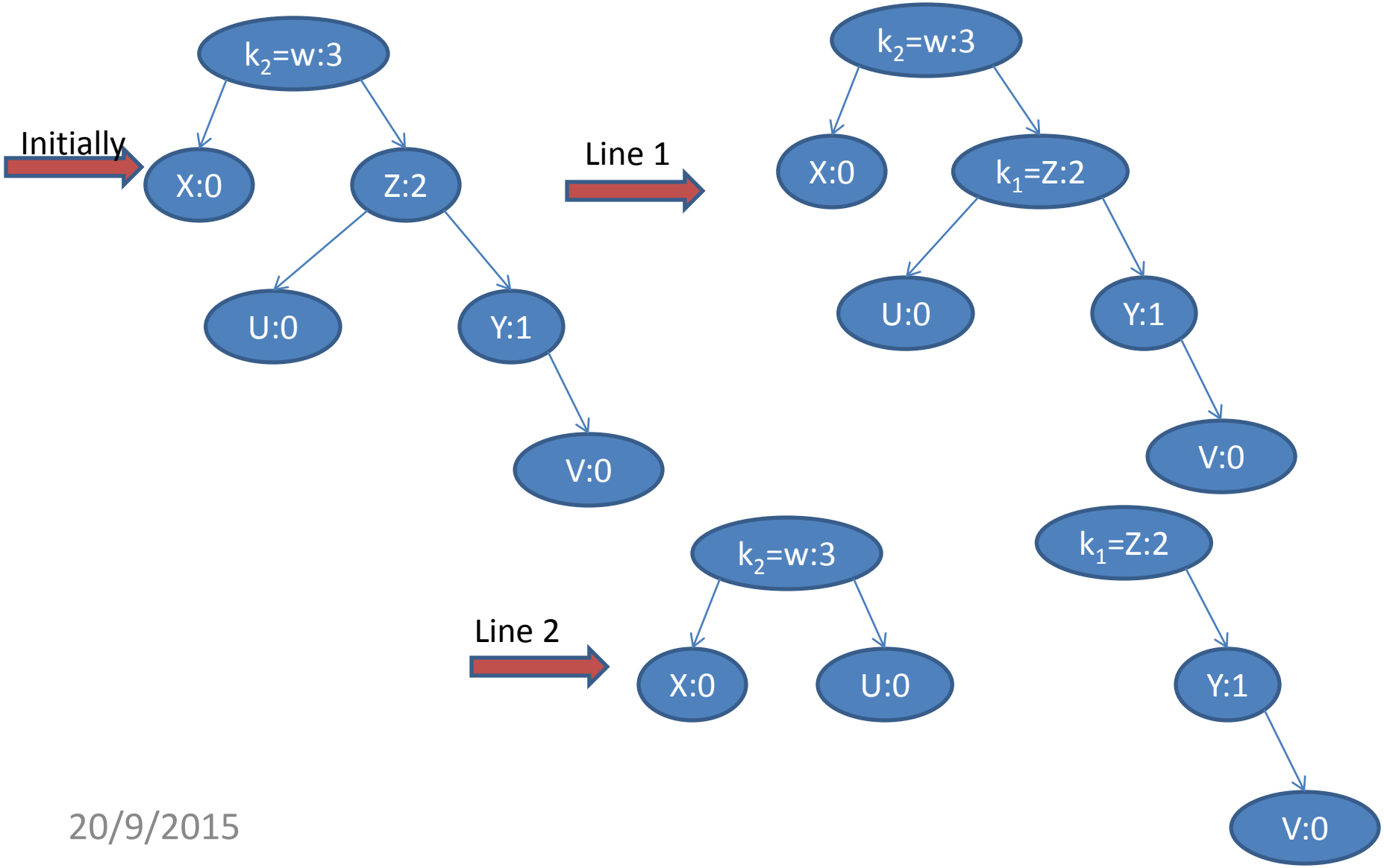




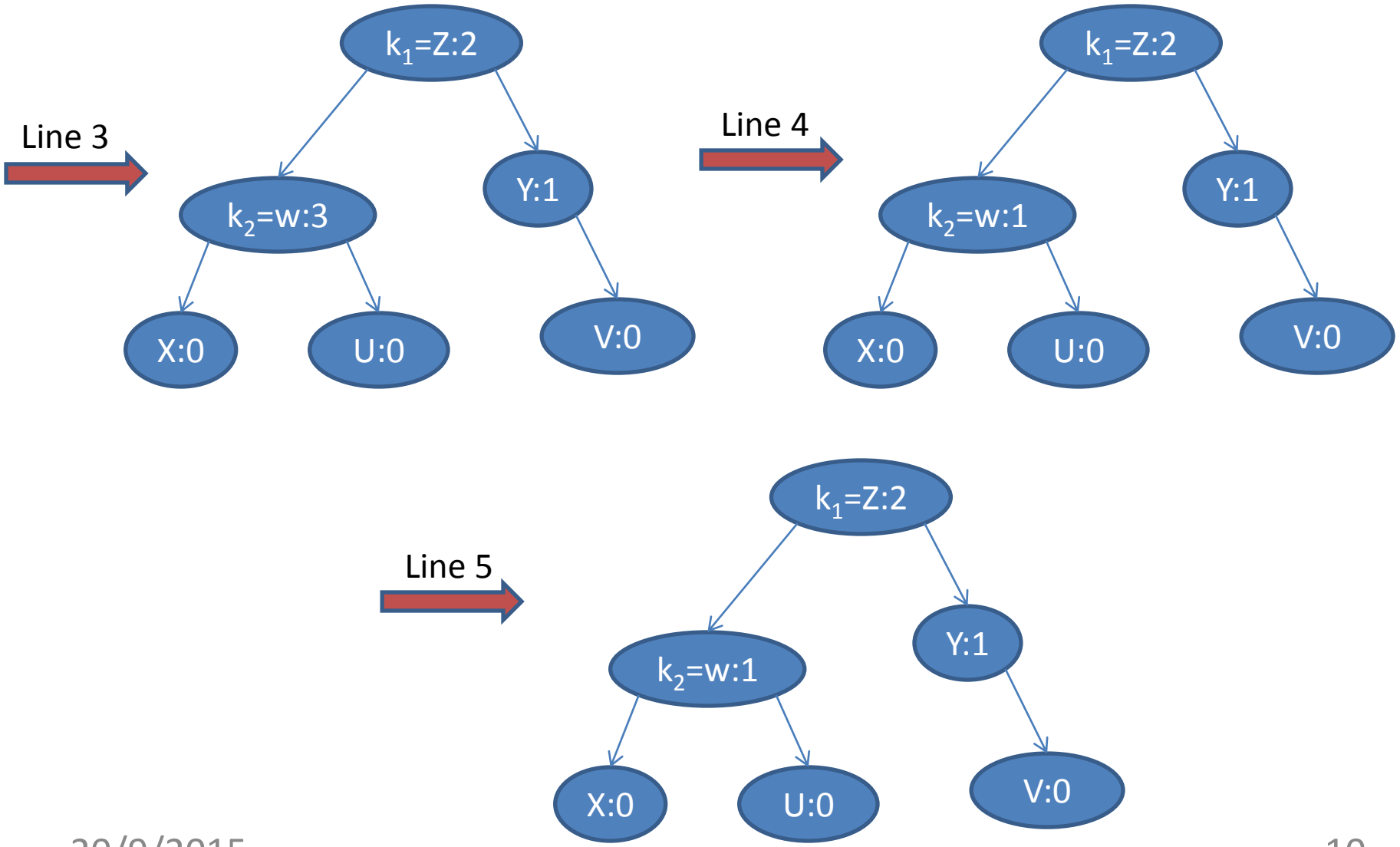
rotateWithRightChild(AvlNode<Integer> k2)

```
/**
*         Rotate binary tree node with right child.
*         For AVL trees, this is a single rotation for case 4.
*         Update heights, then return new root.
*/
private AvlNode<Integer> rotateWithRightChild( AvlNode<Integer> k2 ){
        AvlNode<Integer> k1 = k2.right;
        k2.right = k1.left;
        k1.left = k2;
        k2.height = Math.max( height( k2.right ), height( k2.left ) ) + 1;
        k1.height = Math.max( height( k1.right ), k2.height ) + 1;
        return k1;
}
```

rotateWithRightChild(k2 = w)



rotateWithRightChild(k2 = w)

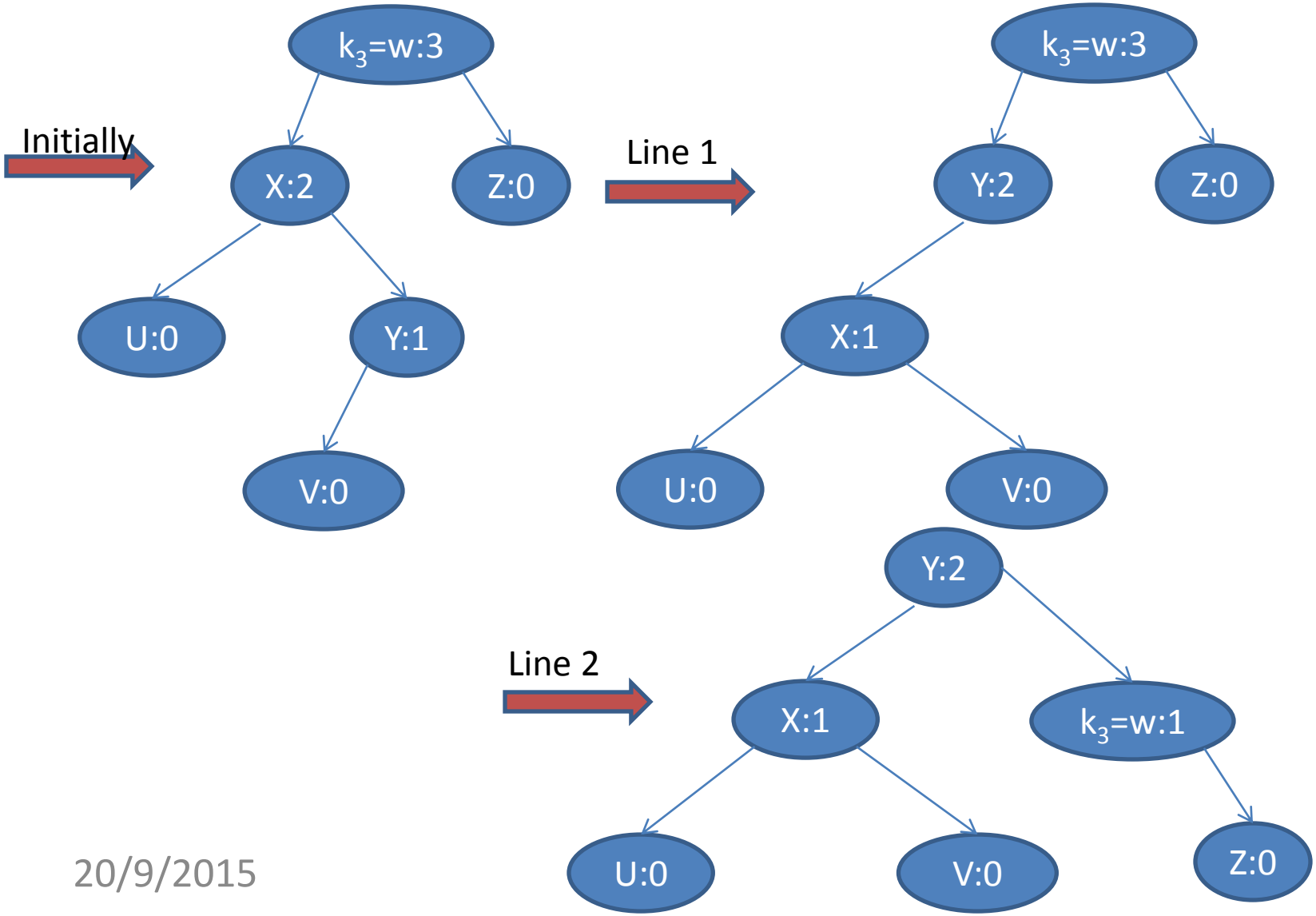


doubleWithLeftChild(AvlNode<Integer> k3)

```
/**
*          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 return new root.
*/

private AvlNode<Integer> doubleWithLeftChild( AvlNode<Integer> k3 ){
1          k3.left = rotateWithRightChild( k3.left );
2          return rotateWithLeftChild( k3 );
}
```

doubleWithLeftChild(k3 = w)

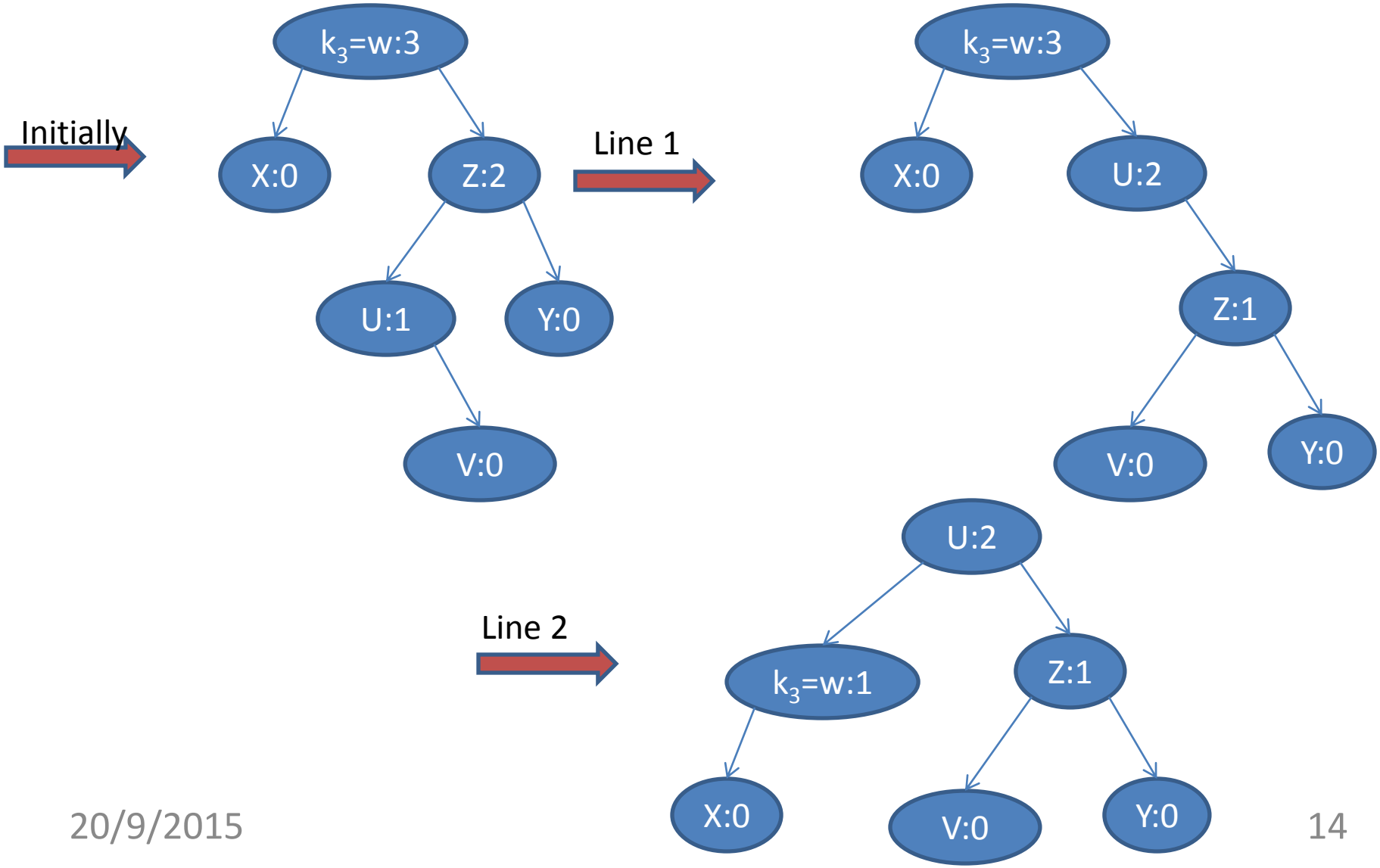


doubleWithRightChild(AvlNode<Integer> k3)

```
/**
*          Double rotate binary tree node: first right child
*          with its left child; then node k3 with new right child.
*          For AVL trees, this is a double rotation for case 2.
*          Update heights, then return new root.
*/

private AvlNode<Integer> doubleWithRightChild( AvlNode<Integer> k3 ){
        k3.right = rotateWithLeftChild( k3.right );
        return rotateWithRightChild( k3 );
}
```

doubleWithRightChild(k3 = w)



balance(AvlNode<Integer> t)

```
private AvlNode<Integer> balance( AvlNode<Integer> t ){
1       if( t == null )
2           return t;

3       if( height( t.left ) - height( t.right ) > ALLOWED_IMBALANCE ){
4           if( height( t.left.left ) >= height( t.left.right ) ){
5               t = rotateWithLeftChild( t );
6           }
7           else{
8               t = doubleWithLeftChild( t );
9           }
10      }
```

balance(AvlNode<Integer> t)

```
11     else{
12         if( height( t.right ) - height( t.left ) > ALLOWED_IMBALANCE ){
13             if( height( t.right.right ) >= height( t.right.left ) ){
14                 t = rotateWithRightChild( t );
15             }
16             else{
17                 t = doubleWithRightChild( t );
18             }
19         }
20     }
21     t.height = Math.max( height( t.left ), height( t.right ) ) + 1;
22     return t;
23 }
```

balance(AvlNode<Integer> t)

Line 4-6 Left Left case
rotateWithLeftChild

Line 7-9 Left Right case
doubleWithLeftChild

Line 13-15 Right Right case
rotateWithRightChild

Line 16-18 Right Left case
doubleWithRightChild

insert(Integer x, AvlNode<Integer> t)

```
private AvlNode<Integer> insert( Integer x, AvlNode<Integer> t ){
1       if( t == null )
2               return new AvlNode<Integer>( x, null, null );

3       int compareResult = x.compareTo( t.element );

4       if( compareResult < 0 ){
5               t.left = insert( x, t.left );
6       }
7       else if( compareResult > 0 ){
8               t.right = insert( x, t.right );
9       }
10      else{
11              ;// Duplicate; do nothing
12      }

13      return balance( t );
}
```

insert(Integer x, AvlNode<Integer> t)

Line 1-2 **if root is null**

make a new node as root and return

Line 3 **root is not null**

decide whether we should go left or right to insert

Line 4-6 **ok we have decided that we need to go left**

recursive functions ☺ again

it is not necessary that we can insert at this node go recursively until find exact place to insert x

Line 7-9 **ok we have decided that we need to go left**

again go recursively until find exact position to insert x

Line 10-12 Don't do any thing on duplicate x value

Line 13 **And this line assures you that imbalance will be removed on lowest node.**

recall that imbalance can be on multiple nodes but we need to remove it on lowest one

Practice Problem: Dry Run Insert Routine

If you won't dry run you won't get insight

Believe me there are still hidden concepts in insertion and deletion routine, which you can understand only if you dry run yourself

We can't dry run each and every piece of code in class

You need to put effort

findMin(AvlNode<Integer> t)

```
private AvlNode<Integer> findMin( AvlNode<Integer> t ){
        if( t == null )
                return null;
        else if( t.left == null )
                return t;
        return findMin( t.left );
}
```

remove(Integer x, AvlNode<Integer> t)

```
private AvlNode<Integer> remove( Integer x, AvlNode<Integer> t ){
1           if( t == null ){
2                   return t; // Item not found; do nothing
3           }

4           int compareResult = x.compareTo( t.element );

5           if( compareResult < 0 ){
6                   t.left = remove( x, t.left );
7           }
8           else if( compareResult > 0 ){
9                   t.right = remove( x, t.right );
10          }
11          else if( t.left != null && t.right != null ){
12                  t.element = findMin( t.right ).element;
13                  t.right = remove( t.element, t.right );
14          } else{
15                  t = ( t.left != null ) ? t.left : t.right;
16          }
17          return balance( t );
}
```

remove(Integer x, AvlNode<Integer> t)

Line 1-3 if root is null -> x not found
do nothing

Line 4 root is not null
decide whether we should go left or right to insert

Line 5-7 ok we have decided that we need to go left
it is not necessary that we found x at this node go recursively until we find exact position of x to remove

Line 8-10 ok we have decided that we need to go left
again go recursively until we find exact position of x to remove

Line 11-13 We find the node we want to delete
Node has 2 childs and code handle remove correctly. How ? Plz dry run

Line 14-16 We find the node we want to delete
Node has 1 childs and code handle remove correctly. How ? Plz dry run

Practice Problem: Dry Run Delete Routine

If you won't dry run you won't get insight

Believe me there are still hidden concepts in insertion and deletion routine, which you can understand only if you dry run yourself

We can't dry run each and every piece of code in class

You need to put effort

preOrder(AvlNode root)

```
void preOrder(AvlNode root){
1           if(root != null){
2                       System.out.print(root.element + " ");
3                       preOrder(root.left);
4                       preOrder(root.right);
5           }
}
```

main

```
public static void main(String[] arguments) {
1       AVLTree tree = new AVLTree();                   //Make a tree

2       tree.root = tree.insert(10, tree.root);
3       tree.root = tree.insert(20, tree.root);
4       tree.root = tree.insert(30, tree.root);
5       tree.root = tree.insert(40, tree.root);
6       tree.root = tree.insert(50, tree.root);
7       tree.root = tree.insert(25, tree.root);         //Insert many nodes
8       tree.root = tree.insert(60, tree.root);
9       tree.root = tree.insert(70, tree.root);
10      tree.root = tree.insert(80, tree.root);
11      tree.root = tree.insert(90, tree.root);
12      tree.root = tree.insert(65, tree.root);

13      tree.preOrder(tree.root);                       // print tree in preorder

14      tree.root = tree.remove(65, tree.root);         // remove one node

15      System.out.println();
16      tree.preOrder(tree.root);                       //print tree in preorder again
}
```