SWINBURNE UNIVERSITY OF TECHNOLOGY

Tree: Design pattern example and AVL tree

Recall the categories of pattern

- Creational design patterns
  - These design patterns are all about class instantiation. This pattern can be further divided into class-creation patterns and object- creational patterns.
- Structural design patterns
  - These design patterns are all about Class and Object composition. Structural class-creation patterns use inheritance to compose interfaces.
- Behavioural design patterns
  - These design patterns are all about Class's objects communication. Behavioural patterns are those patterns that are most specifically concerned with communication between objects.

SWINBURNE UNIVERSITY OF TECHNOLOGY

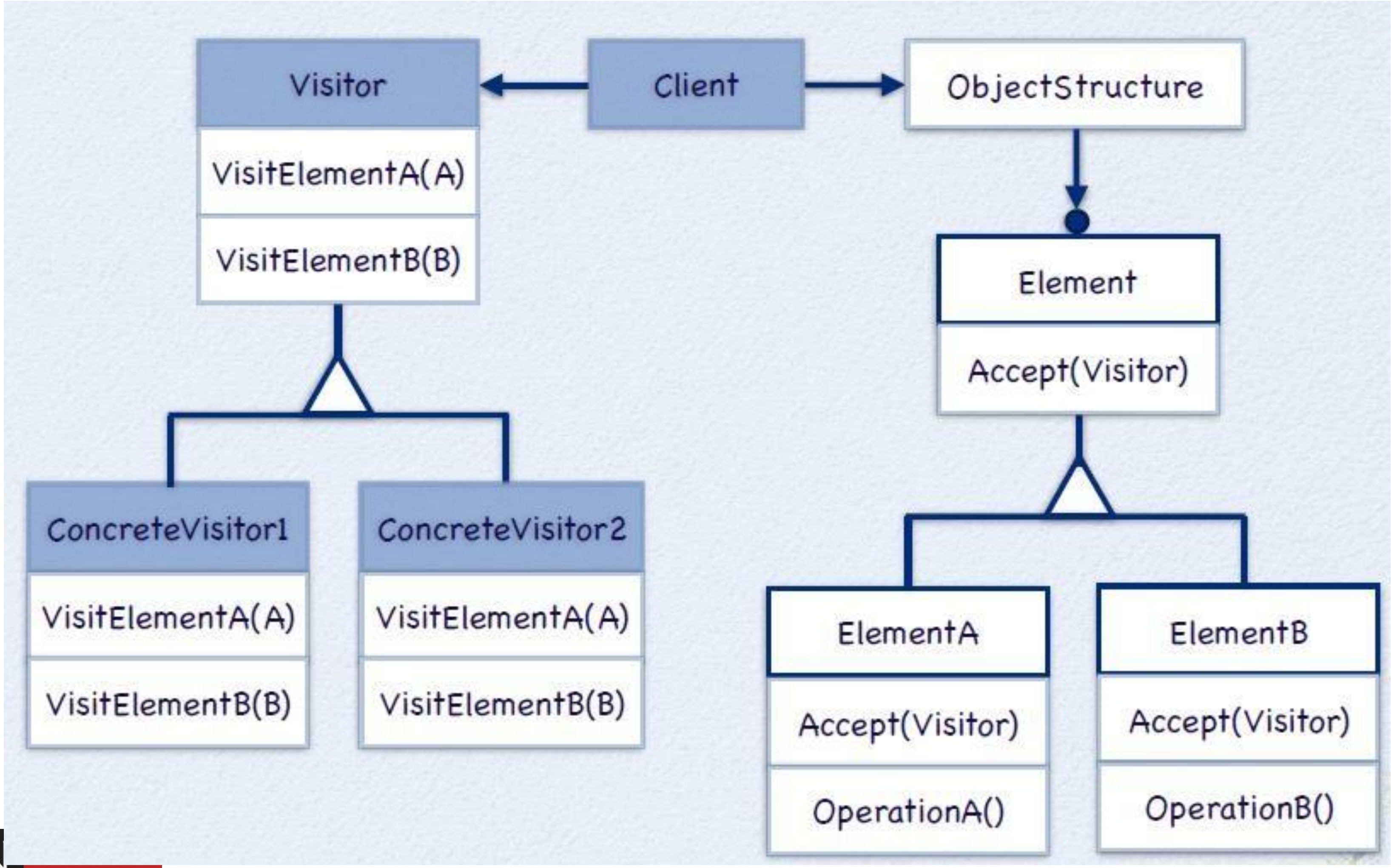
The Visitor Pattern

- The Visitor Pattern falls under the category of behavioral design patterns
- **Intent**
  - Represent an operation to be performed on the elements of an object structure. Visitor lets you define a new operation without changing the classes of the elements on which it operates.
  - The classic technique for recovering lost type information.
  - Do the right thing based on the type of two objects.
    - **Double Dispatch** is used to invoke an overloaded method where the parameters vary among an inheritance hierarchy

The Visitor Pattern

- Collaborations:
- A client that uses the Visitor pattern must create a ConcreteVisitor object and then traverse the object structure, visiting each element with the visitor.
- When an element is visited, it calls the Visitor operation that corresponds to its class. The element supplies itself as an argument to this operation to let the visitor access its state, if necessary.

Structure of Visitor



A Tree Visitor

TreeVisitor.h

```cpp
#include <iostream>

template<class T>
class TreeVisitor
{
public:
    virtual ~TreeVisitor() {}   // virtual default destructor

    // default behavior
    virtual void preVisit( const T& aKey ) const {}
    virtual void postVisit( const T& aKey ) const {}
    virtual void inVisit( const T& aKey ) const {}

    virtual void visit( const T& aKey ) const
    {
        std::cout << aKey << " ";
    }
};
```

Line: 2 Column: 23 C++ Tab Size: 4 TREEVISITOR_H_

Pre-Order,
Post-Order
In-Order
from DFS

SWINBURNE UNIVERSITY OF TECHNOLOGY

PreOrderVisitor

TreeVisitor.h

```cpp
template<class T>
class PreOrderVisitor : public TreeVisitor<T>
{
public:

    // override pre-order behavior
    virtual void preVisit( const T& aKey ) const
    {
        visit( aKey );  // invoke default behavior
    }
};
```

Line: 17 Column: 13 C++ Tab Size: 4 visit

PostOrderVisitor

TreeVisitor.h

```
template<class T>
class PostOrderVisitor : public TreeVisitor<T>
{
public:

    // override post-order behavior
    virtual void postVisit( const T& aKey ) const
    {
        visit( aKey );  // invoke default behavior
    }
};

```

Line: 17 Column: 13 C++ Tab Size: 4 visit

InOrderVisitor

TreeVisitor.h

```
template<class T>
class InOrderVisitor : public TreeVisitor<T>
{
public:

    // override in-order behavior
    virtual void inVisit( const T& aKey ) const
    {
        visit( aKey );  // invoke default behavior
    }
};
```

Line: 17 Column: 13 C++ Tab Size: 4 visit

Depth-first Traversal for BTree

BTreeImpl.h

```cpp

void traverseDepthFirst( const TreeVisitor<T>& aVisitor ) const
{
  if ( !isEmpty() )
  {
    aVisitor.preVisit( key() );         // Show if PreOrder
    left().traverseDepthFirst( aVisitor );
    aVisitor.inVisit( key() );          // Show if InOrder
    right().traverseDepthFirst( aVisitor );
    aVisitor.postVisit( key() );        // Show if PostOrder
  }
}

```

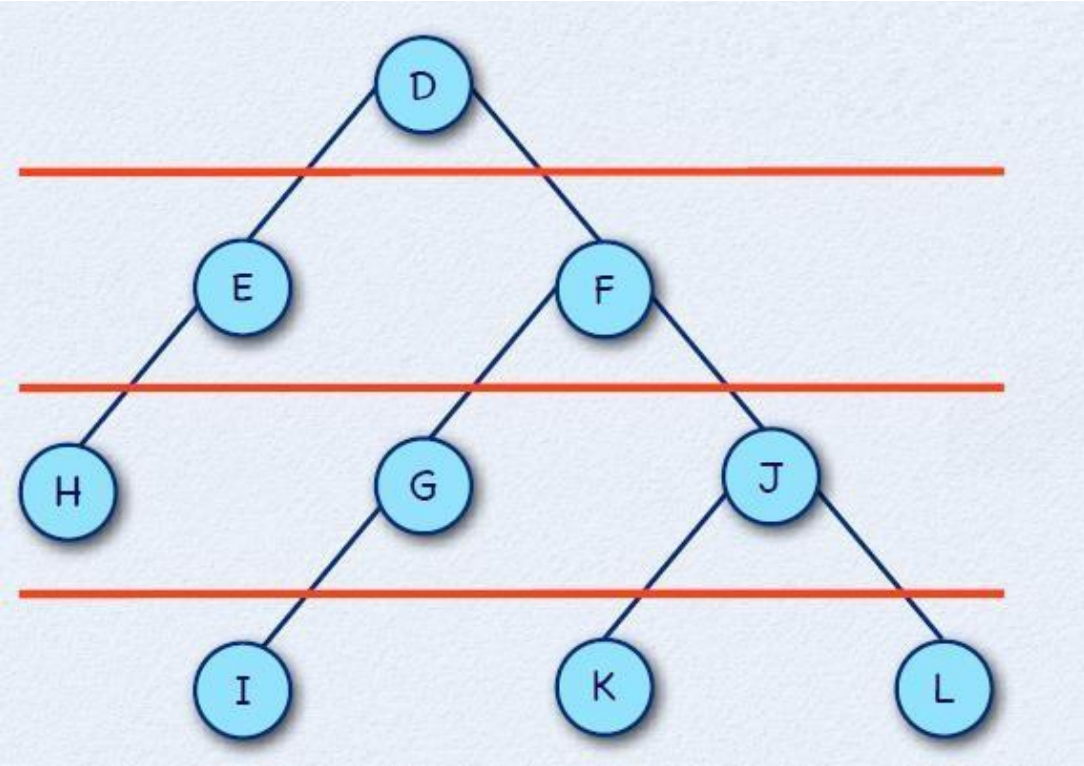
BTreeTest.cpp

```cpp
int main()
{
    string s1( "Hello World!" );
    string s2( "A" );
    string s3( "B" );
    string s4( "C" );

    BTree<string> A2Tree( s1 );

    BTree<string> STree1( s2 );
    BTree<string> STree2( s3 );
    BTree<string> STree3( s4 );

    A2Tree.attachLeft( &STree1 );
    A2Tree.attachRight( &STree2 );
    A2Tree.left().attachLeft( &STree3 );

    cout << "Key: " << A2Tree.key() << endl;
    cout << "Key: " << A2Tree.left().left().key() << endl;

    A2Tree.traverseDepthFirst( PreOrderVisitor<string>() );
    cout << endl;

    A2Tree.left().detachLeft();
    A2Tree.detachLeft();
    A2Tree.detachRight();

    return 0;
}
```

Terminal

```
Sela:HIT3303 Markus$ ./BTreeTest
Key: Hello World!
Key: C
Hello World! A C B
Sela:HIT3303 Markus$ _
```

Breadth-first Traversal Implementation



- Traversal : D-E-F-H-G-J-I-K-L

Breadth-first Traversal for BTree

BTreeImpl.h

```cpp
    void traverseBreadthFirst( const TreeVisitor<T>& aVisitor ) const
    {
      Queue< BTree<T> > lQueue;

      if ( !isEmpty() ) lQueue.enqueue( *this );                       // start with root node

      while ( !lQueue.isEmpty() )
      {
        const BTree<T>& head = lQueue.dequeue();

        if ( !head.isEmpty() ) aVisitor.visit( head.key() );           // output
        if ( !head.left().isEmpty() ) lQueue.enqueue( head.left() );   // enqueue left
        if ( !head.right().isEmpty() ) lQueue.enqueue( head.right() ); // enqueue right
      }
    }
```

BTreeTest.cpp

```cpp
int main()
{
    string s1( "Hello World!" );
    string s2( "A" );
    string s3( "B" );
    string s4( "C" );

    BTree<string> A2Tree( s1 );

    BTree<string> STree1( s2 );
    BTree<string> STree2( s3 );
    BTree<string> STree3( s4 );

    A2Tree.attachLeft( &STree1 );
    A2Tree.attachRight( &STree2 );
    A2Tree.left().attachLeft( &STree3 );

    cout << "Key: " << A2Tree.key() << endl;
    cout << "Key: " << A2Tree.left().left().key() << endl;

    A2Tree.traverseBreadthFirst( PreOrderVisitor<string>() );
    cout << endl;

    A2Tree.left().detachLeft();
    A2Tree.detachLeft();
    A2Tree.detachRight();

    return 0;
}
```

Terminal

```
Sela:HIT3303 Markus$ ./BTreeTest
Key: Hello World!
Key: C
Hello World! A B C
Sela:HIT3303 Markus$ _
```

Visitor Design Pattern (Java)

URL: https://www.youtube.com/watch?v=pL4mOUDi54o

AVL Tree

- Named after 2 Russian mathematicians
- Georgii Adelson-Velsky (1922 - ?)
- Evgenii Mikhailovich Landis (1921-1997)

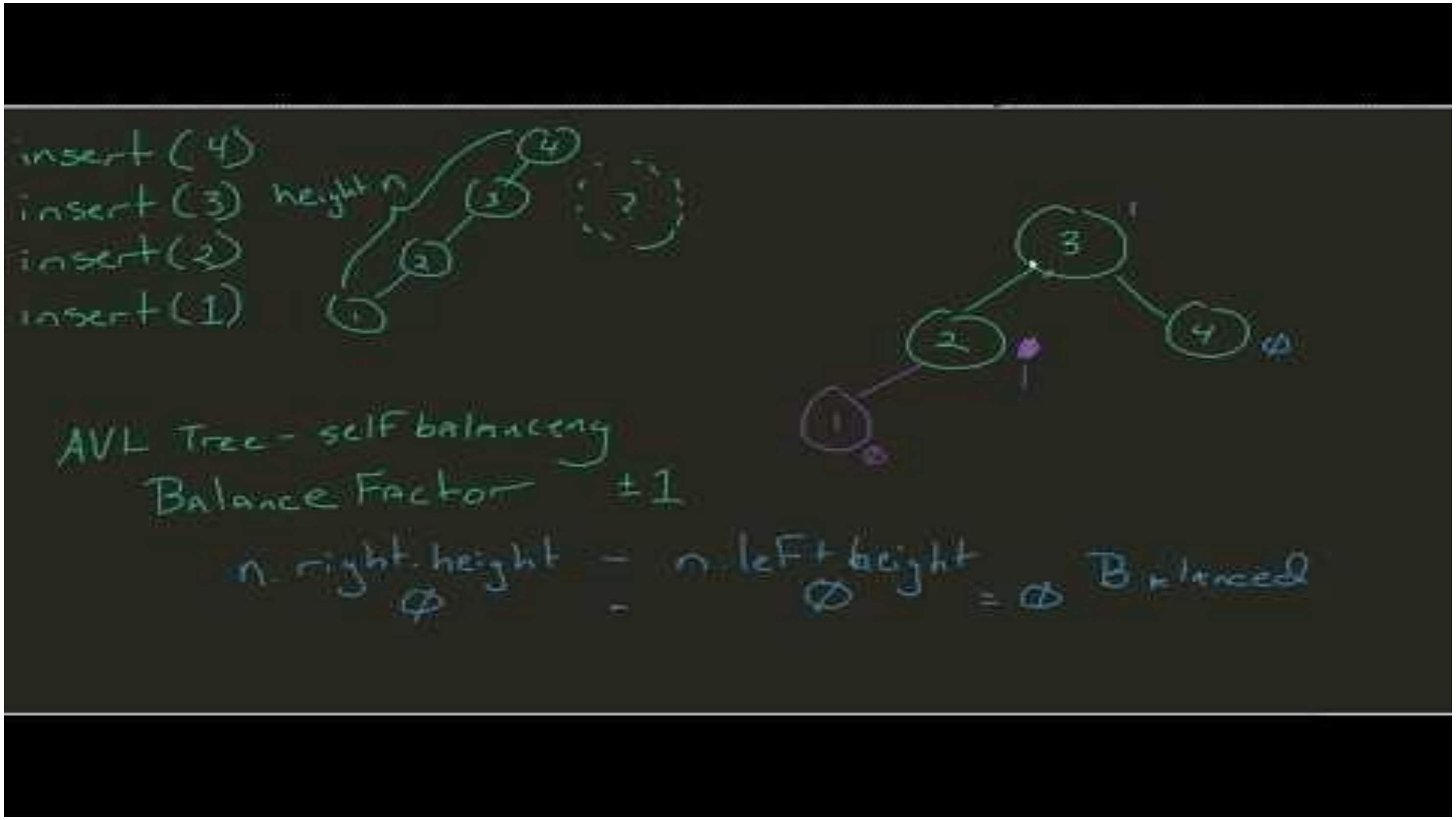
AVL Tree

- AVL trees are height-balanced **binary** search trees
  - A tree where no leaf is much farther away from the root than any other leaf.
  - 1) Left subtree of Tree is balanced
    2) Right subtree of Tree is balanced
    3) The difference between heights of left subtree and right subtree is
    not more than 1.
- Balance factor of a node
  - height(left subtree) - height(right subtree)

AVL Tree

- An AVL tree has balance factor calculated at every node
- For every node, heights of left and right subtree can differ by no more than 1
- Store current heights in each node

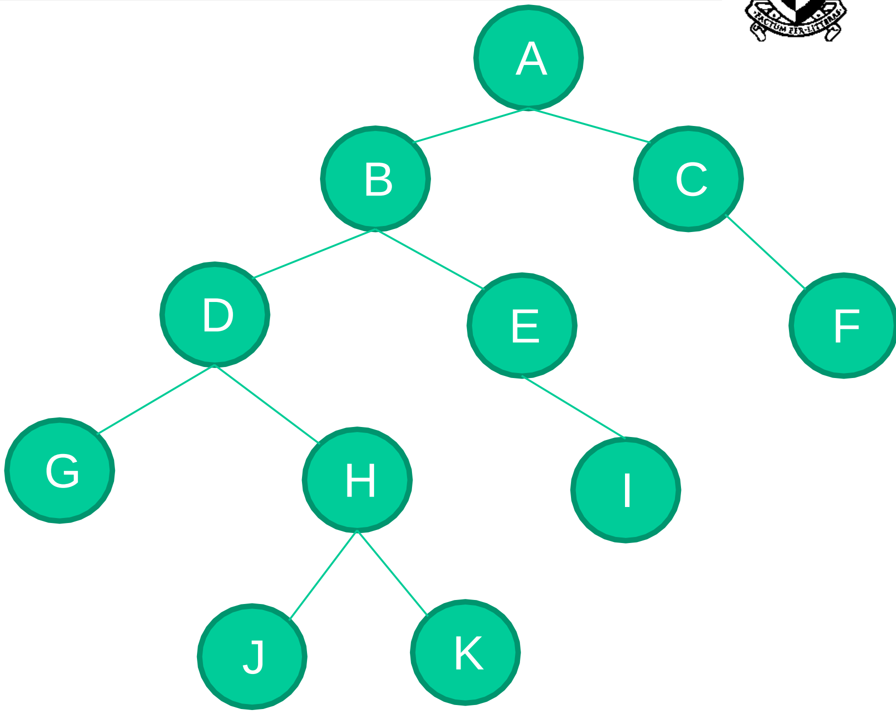
AVL Trees



URL: https://www.youtube.com/watch?v=IGd7yIcxL6U

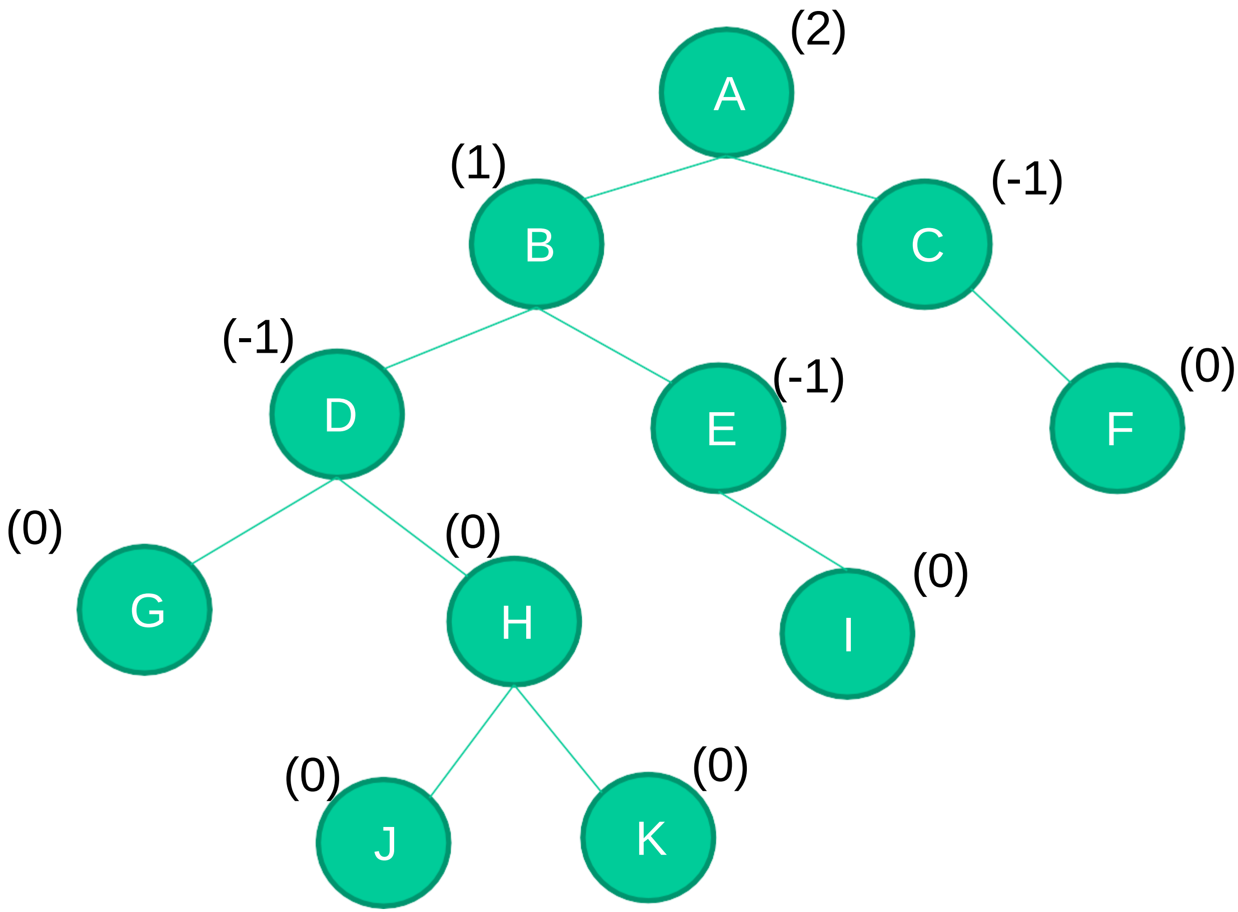
Is this a AVL tree?

- The height of the tree is 4, meaning the length of the longest path from the root to a leaf node.
- The height of the left subtree of the root is 3, meaning that the length of the longest path from the node B to one of the leaf nodes (G, J K or I).
- For finding the balancing factor of the **root** we subtract the height of the right subtree and the left subtree : 3-1 = 2.
- The balancing factor of the node with the key I is very easy to determine. We notice that the node has no children so the balancing factor is 0



- For finding the balancing factor of the node with key D we substract the height of the right subtree from the height of the left subtree: 0-1= -1.

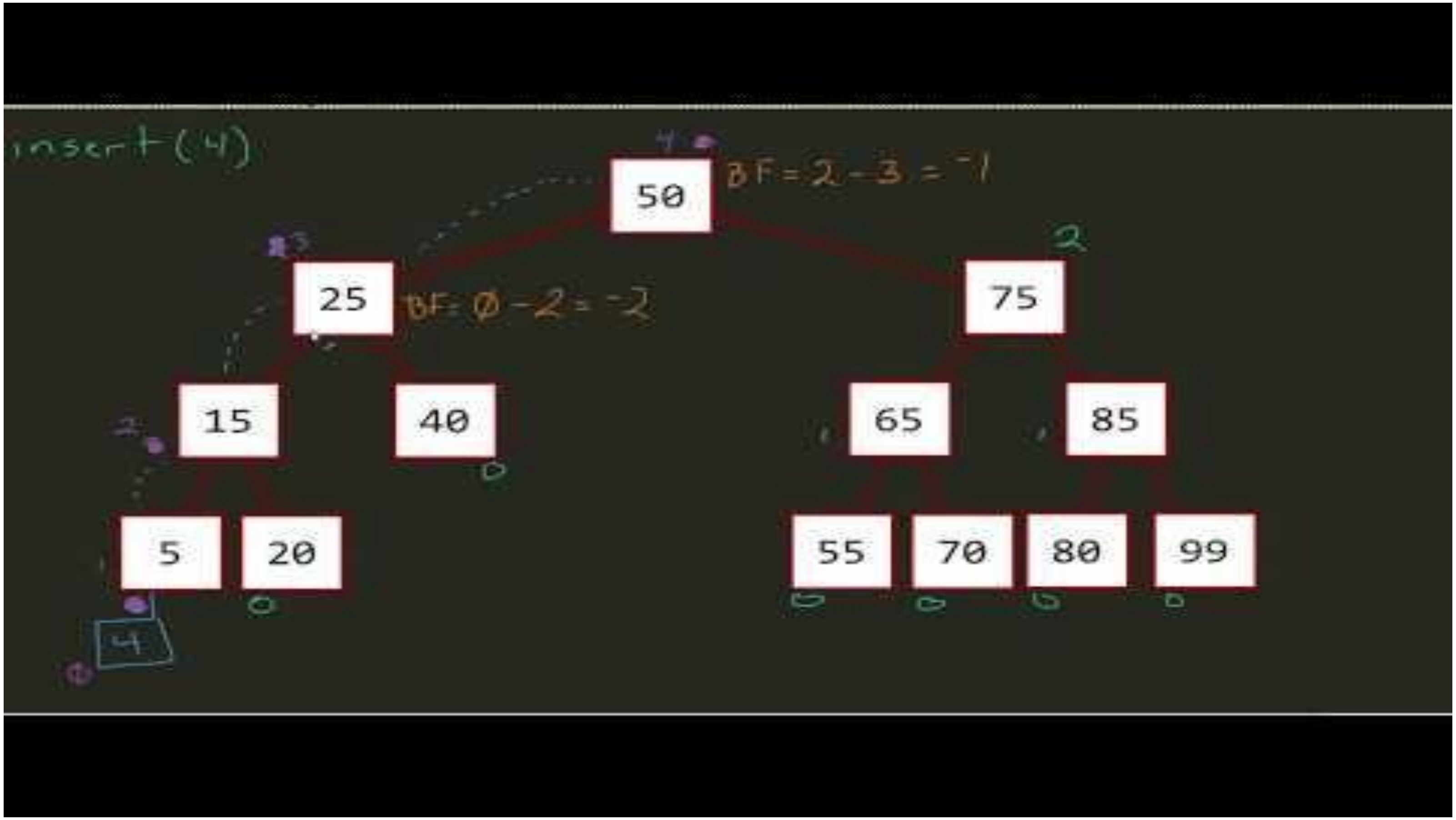
Balance factor



The binary tree is **balanced** when all the balancing factors of all the nodes are -1,0,+1.

This binary tree is not balanced at the **Root**.

AVL Tree – Calculating Balance Factor

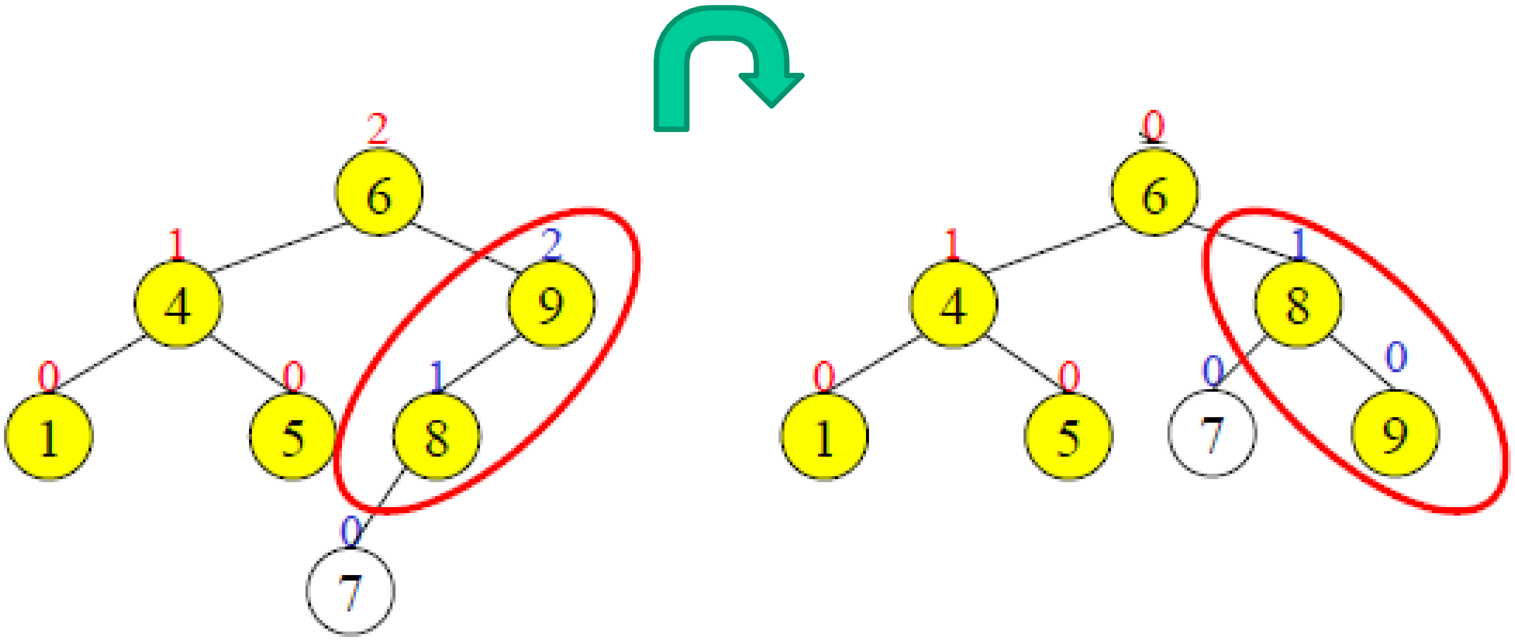


URL: https://www.youtube.com/watch?v=ke4DeoG1bUA

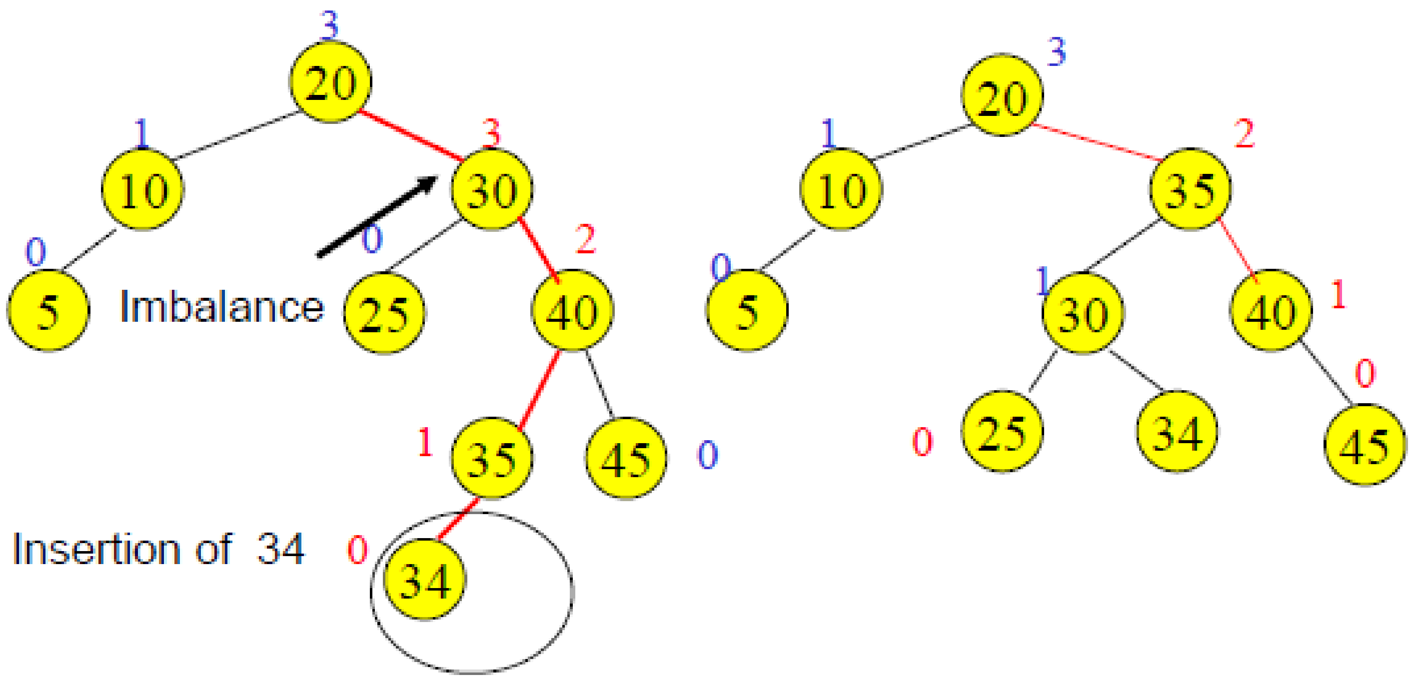
Insert and Rotation in AVL Trees

- Insert operation may cause balance factor to become 2 or –2 for some node
- only nodes on the path from insertion point to root node have possibly changed in height
- So after the Insert, go back up to the root node by node, updating heights
- If a new balance factor (the difference h_{left}-h_{right}) is 2 or –2, adjust tree by *rotation* around the node

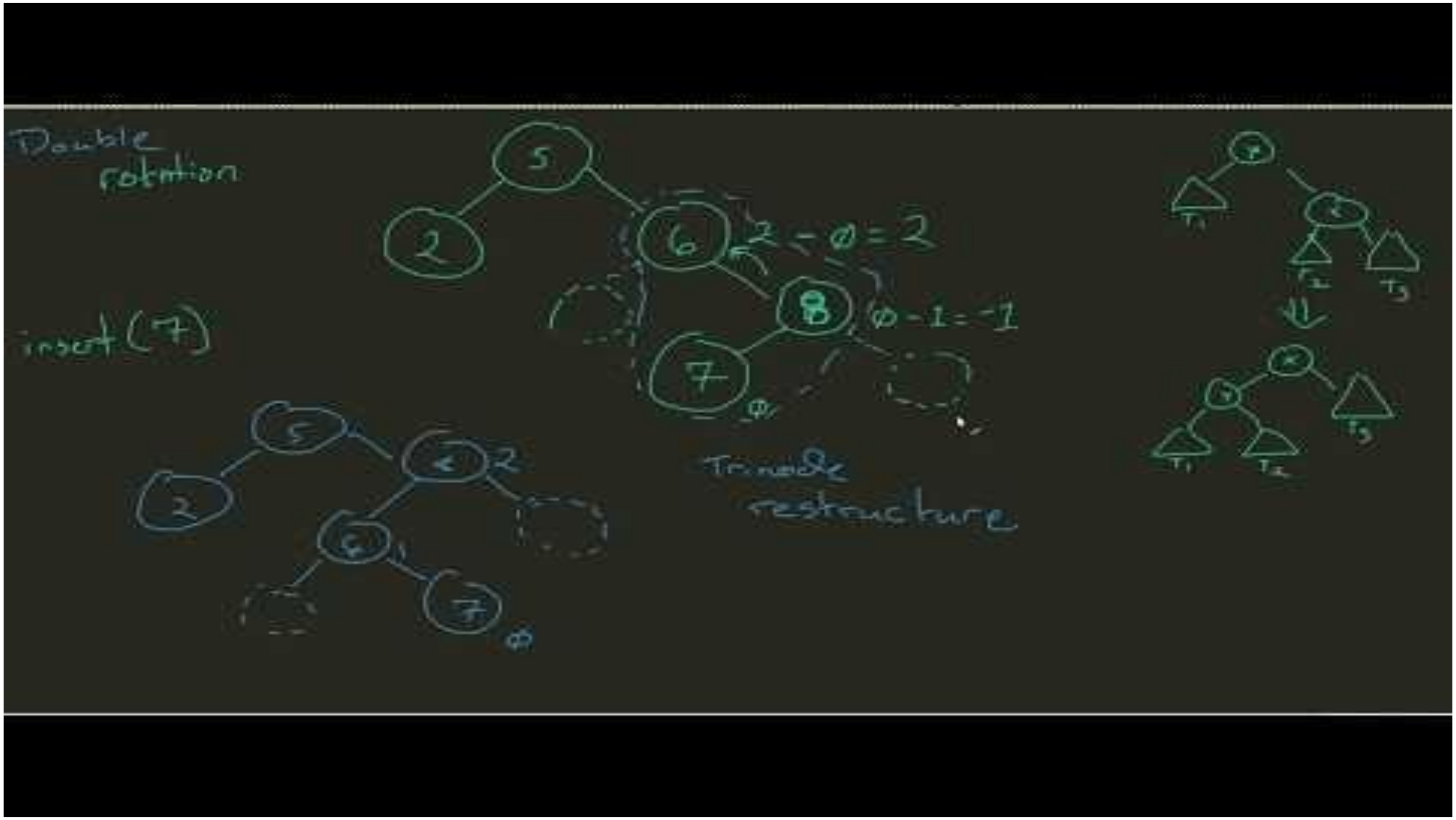
Single Rotation in an AVL Tree



Double Rotation

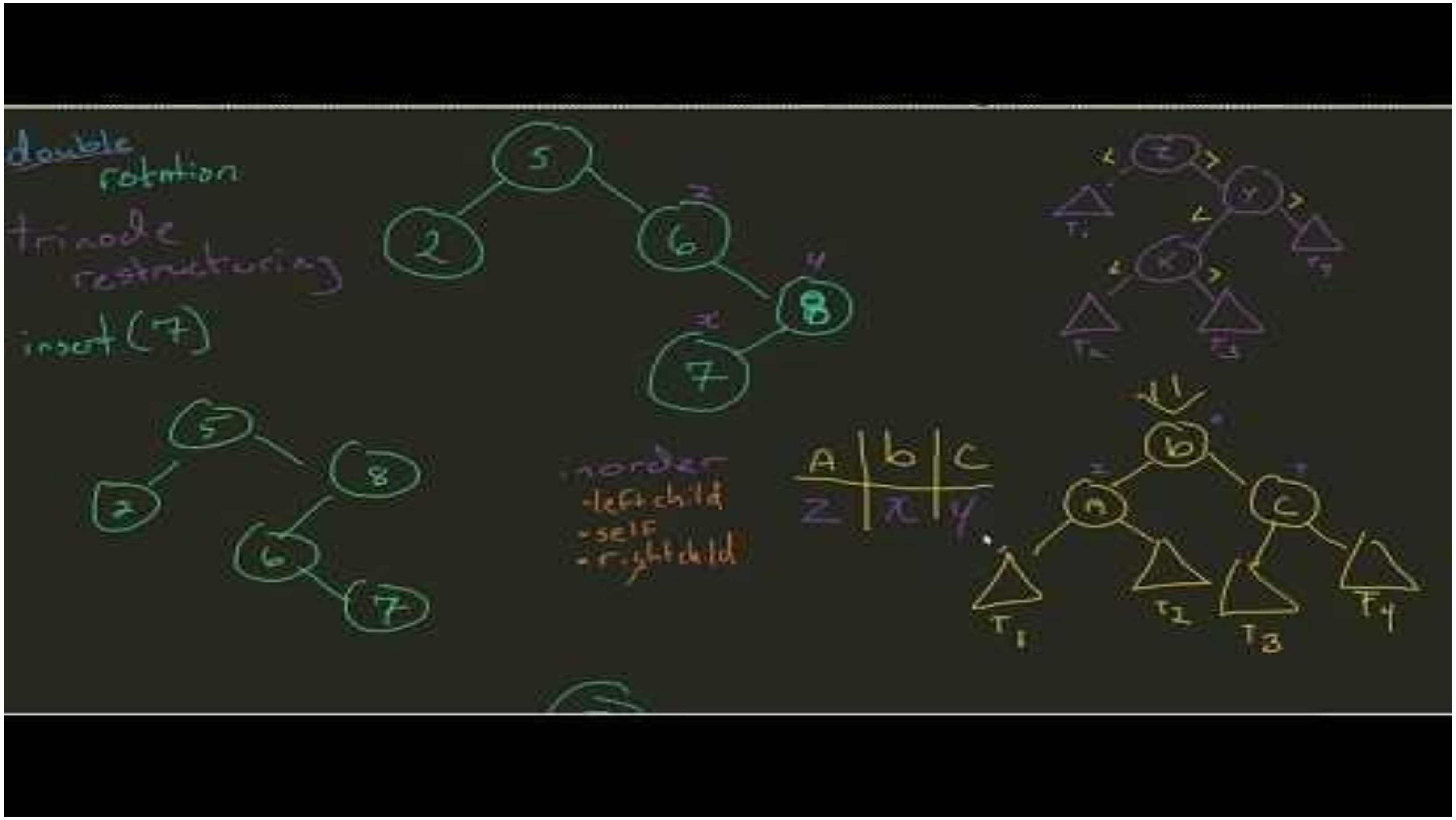


AVL Tree – Understanding Double Rotation



URL: https://www.youtube.com/watch?v=5m5fPidj6Mo

AVL Tree – Double Rotation

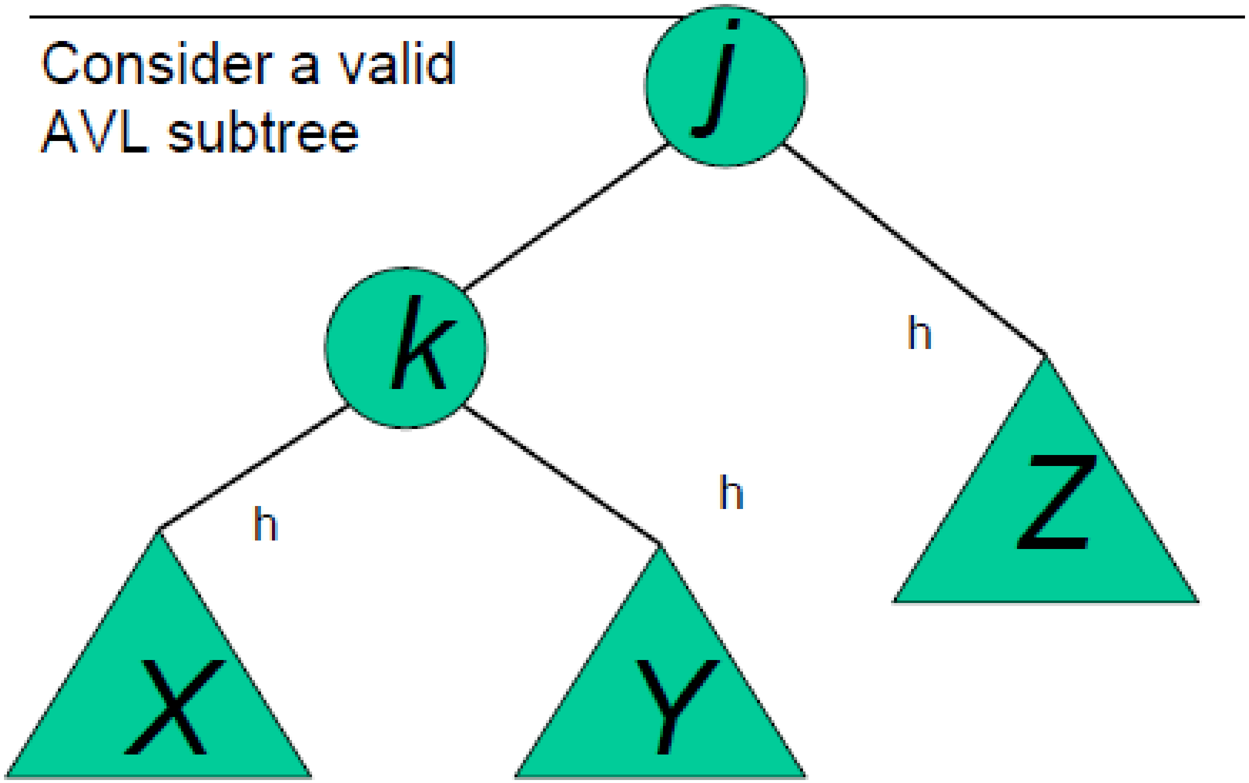


URL: https://www.youtube.com/watch?v=M_yhWNNCnGU

Insertions in AVL Trees

- Let the node that needs rebalancing be α.
- There are 4 cases:
- Outside Cases (require single rotation) :
  1. Insertion into left subtree of left child of α.
  2. Insertion into right subtree of right child of α.
- Inside Cases (require double rotation) :
  3. Insertion into right subtree of left child of α.
  4. Insertion into left subtree of right child of α.
- The rebalancing is performed through four separate rotation algorithms, left-left rotation, right-right rotation, right-left rotation and left-right rotation.

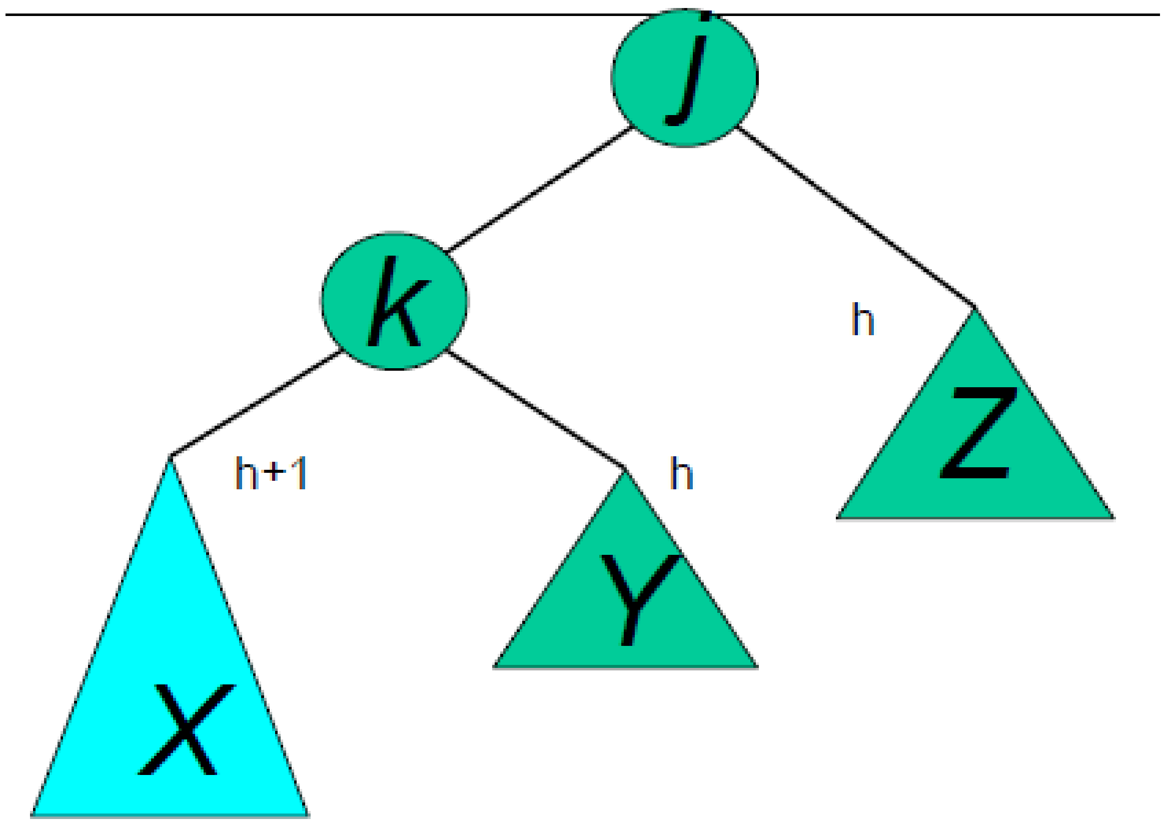
AVL Insertion: Outside Case



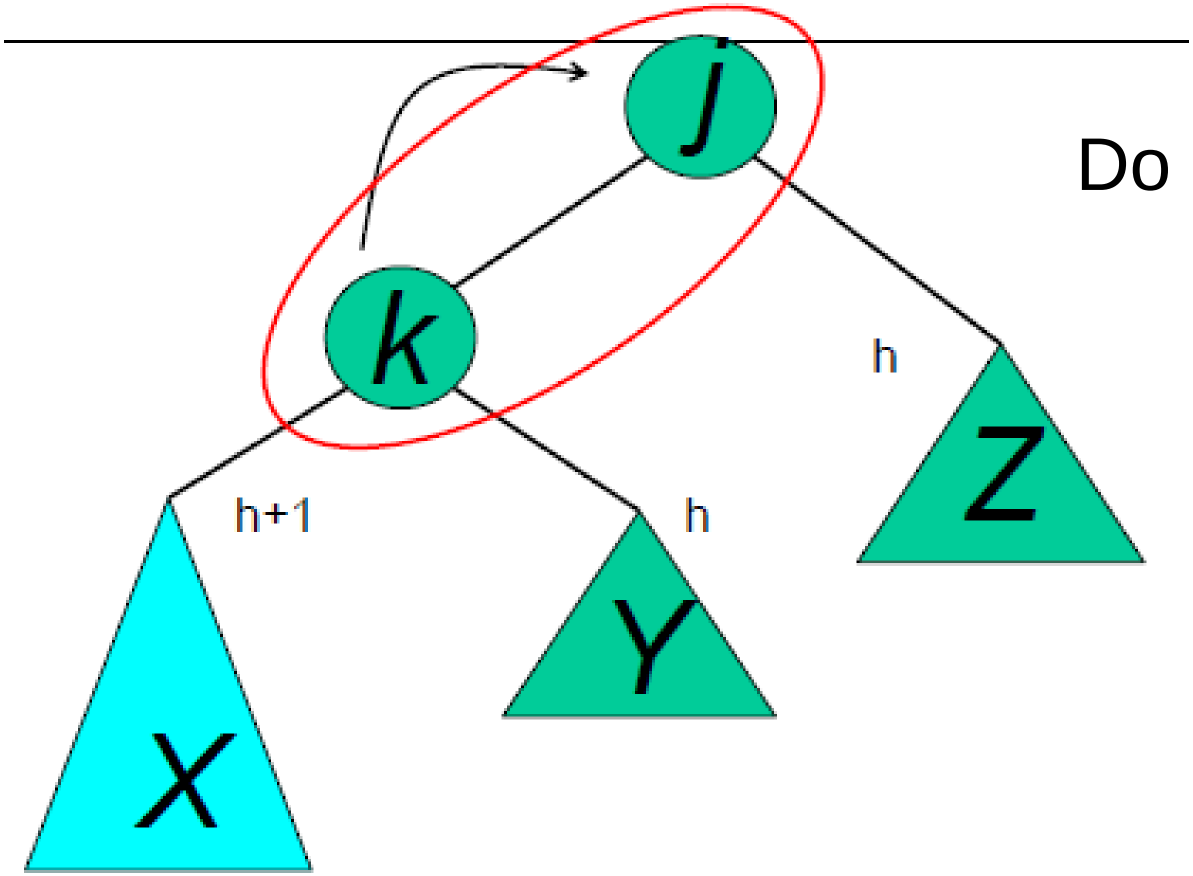
SWINBURNE UNIVERSITY OF TECHNOLOGY

AVL Insertion: Outside Case

- Inserting into X destroys the AVL property at node j

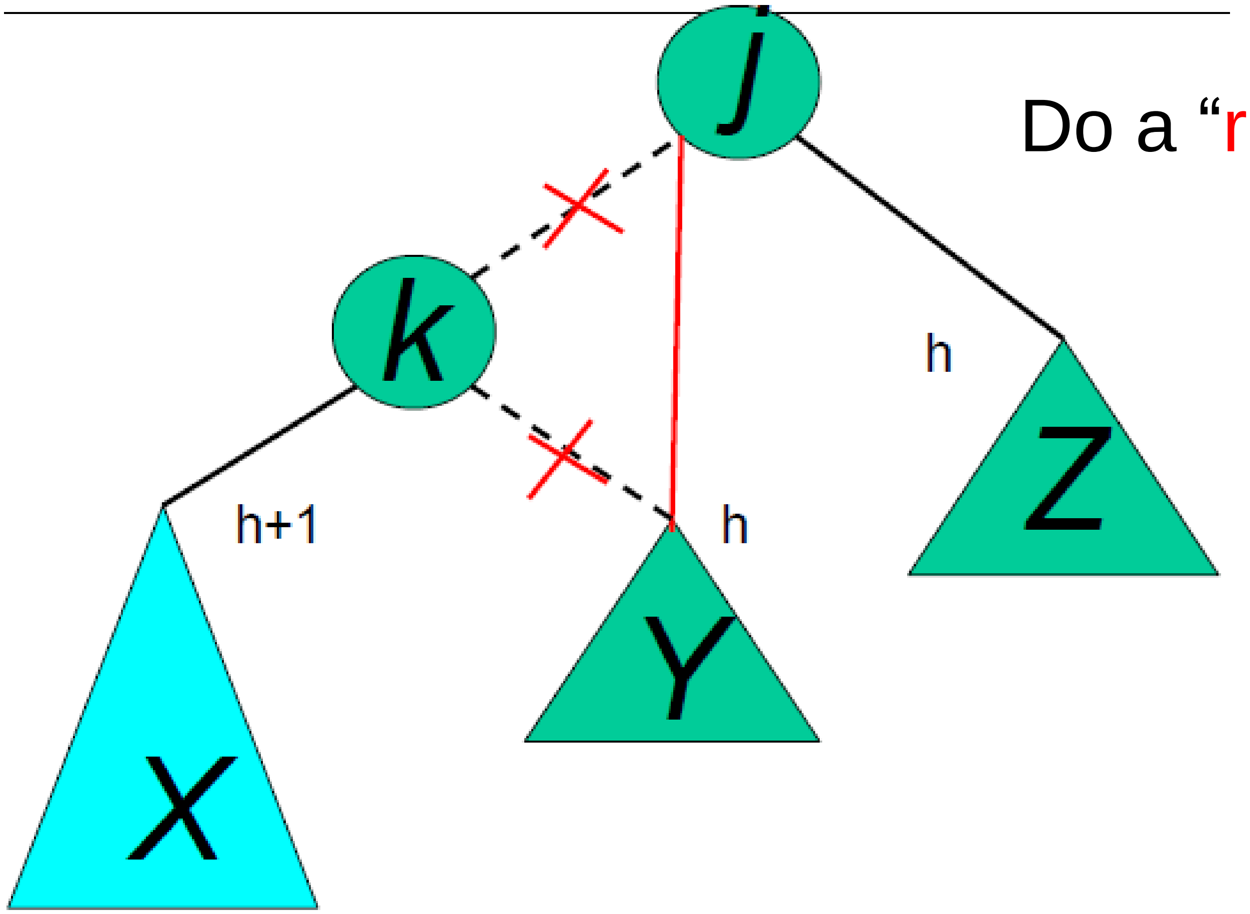


AVL Insertion: Outside Case



Do a “right rotation”

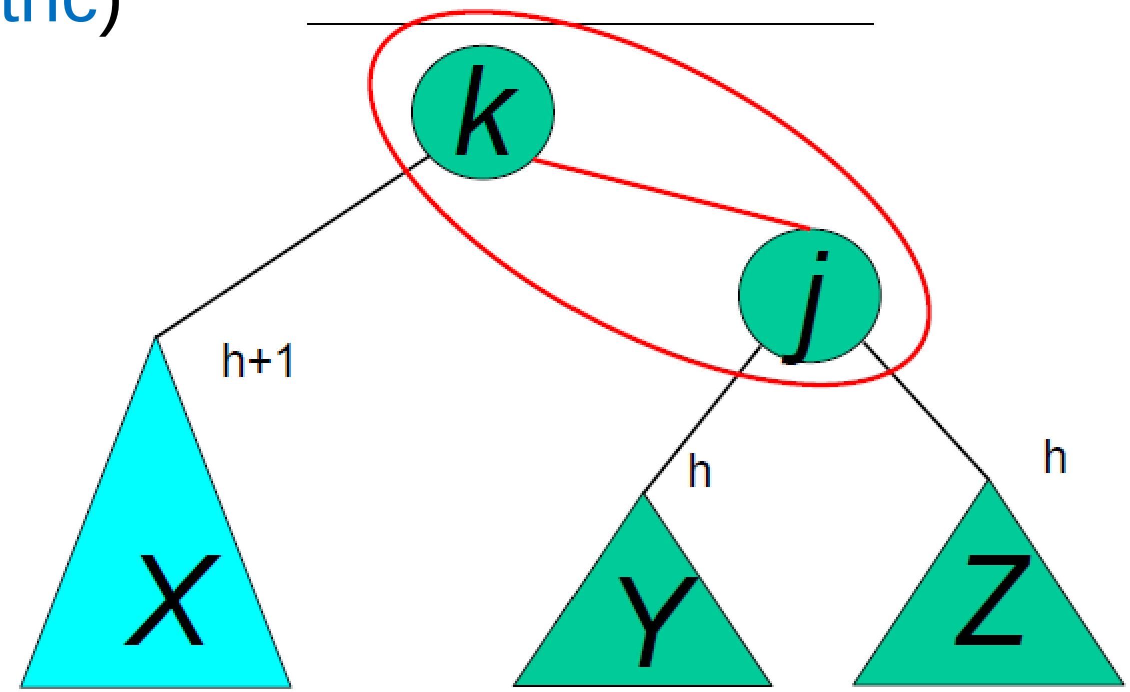
Single Right Rotation



Do a "right rotation"

Outside Case Completed

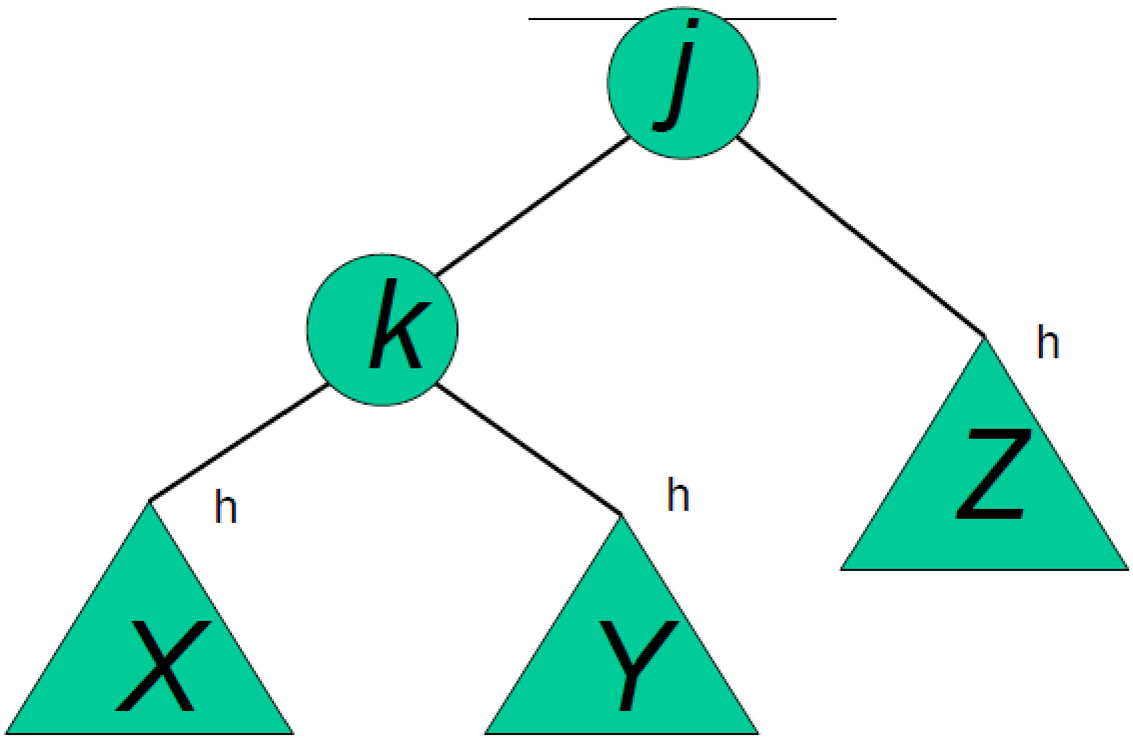
- "Right rotation" done! ("Left rotation" is mirror symmetric)



- AVL property has been restored!

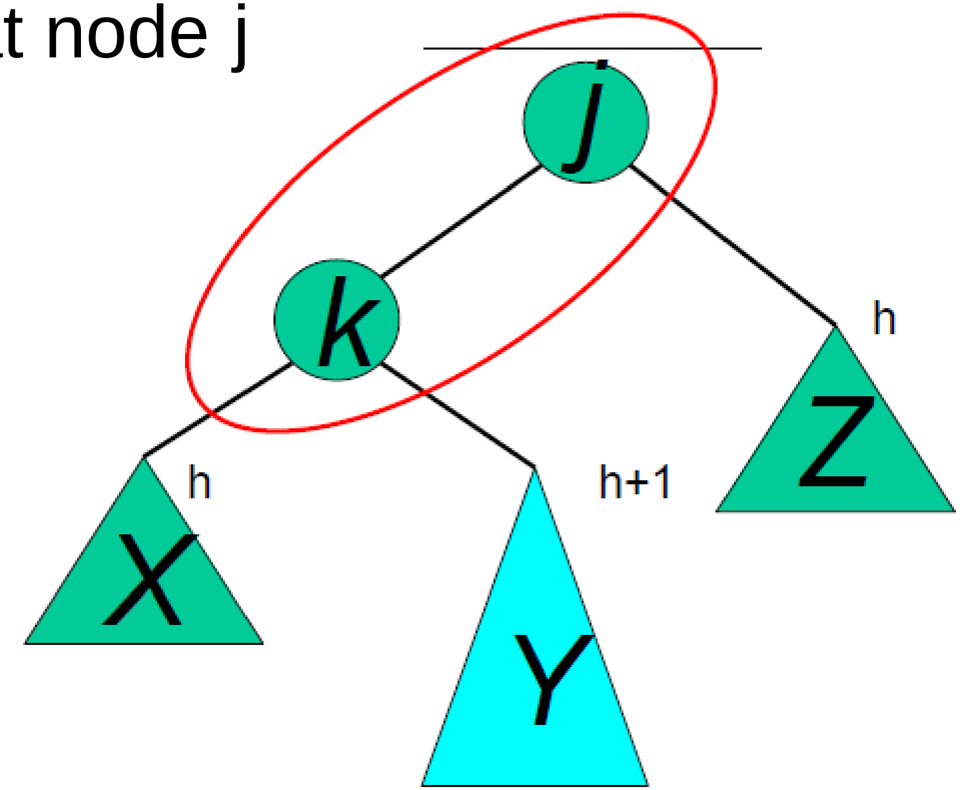
AVL Insertion: Inside Case

- Consider a valid AVL subtree



AVL Insertion: Inside Case

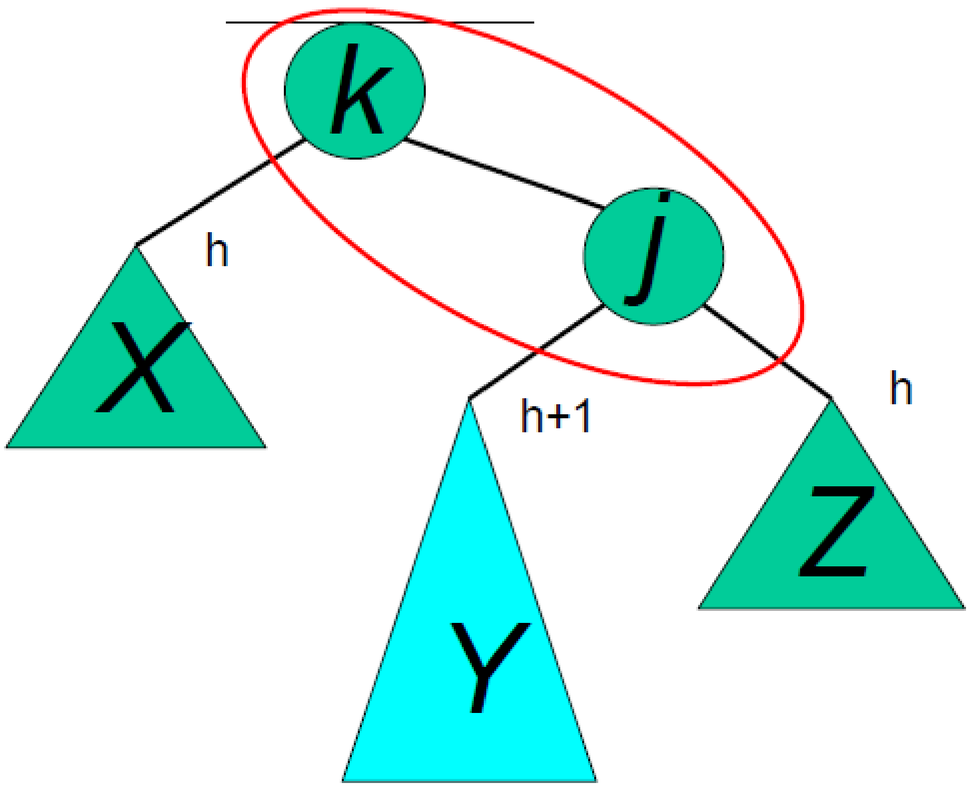
- Inserting into Y destroys the AVL property at node j



- Does “right rotation” restore balance?

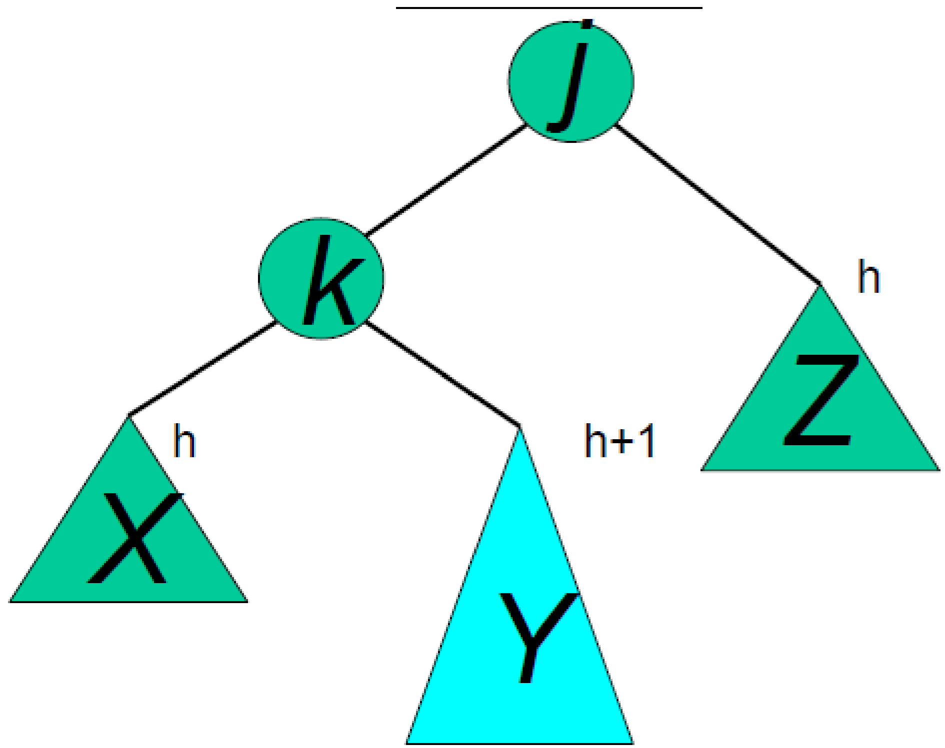
AVL Insertion: Inside Case

- “Right rotation” does not restore balance… now k is out of balance



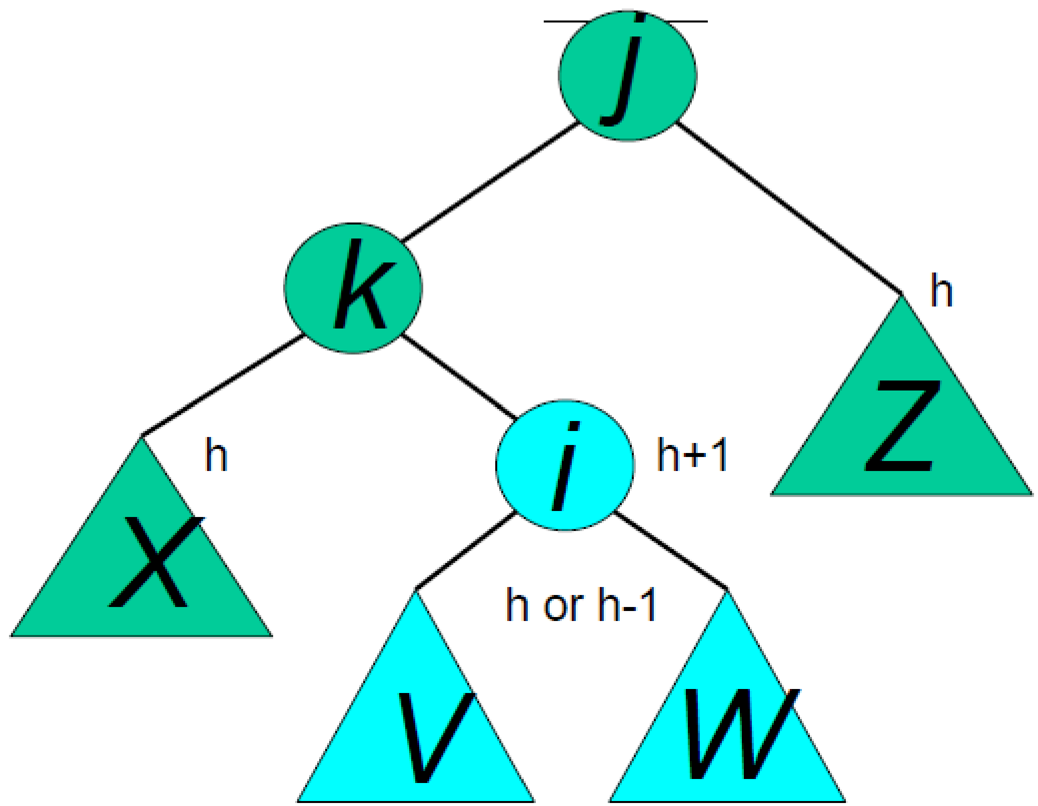
AVL Insertion: Inside Case

- Consider the structure of subtree Y…



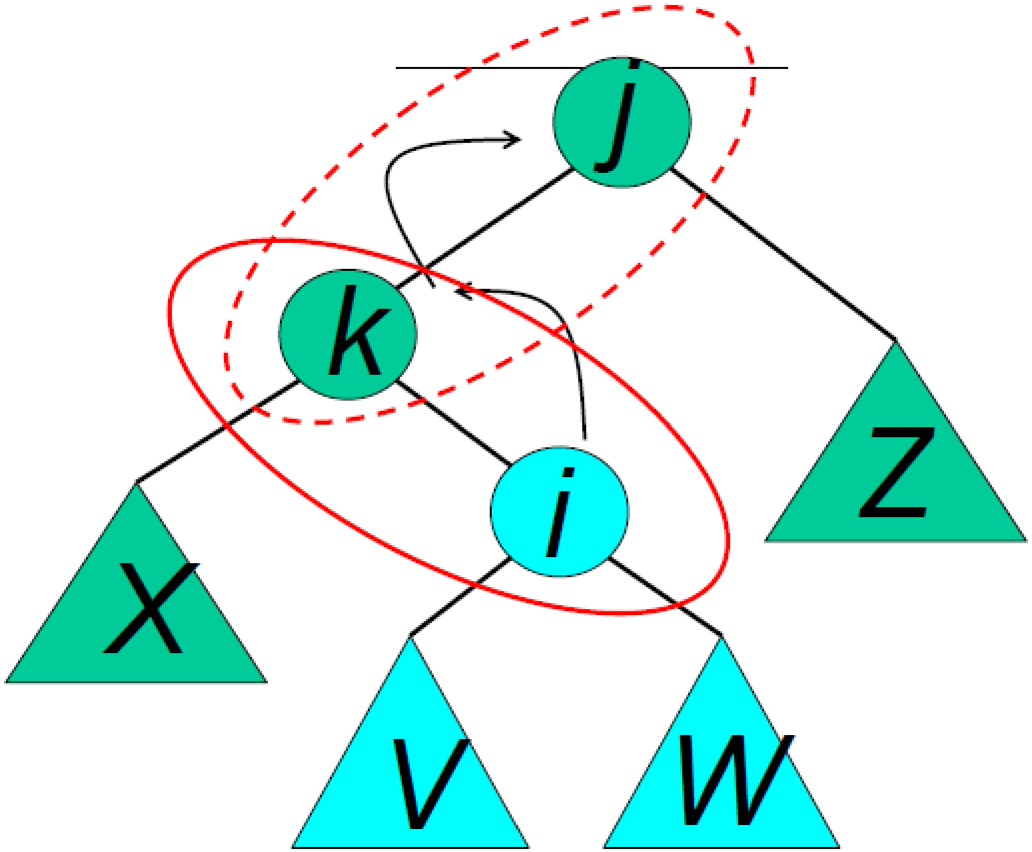
AVL Insertion: Inside Case

- Y = node i and subtrees V and W



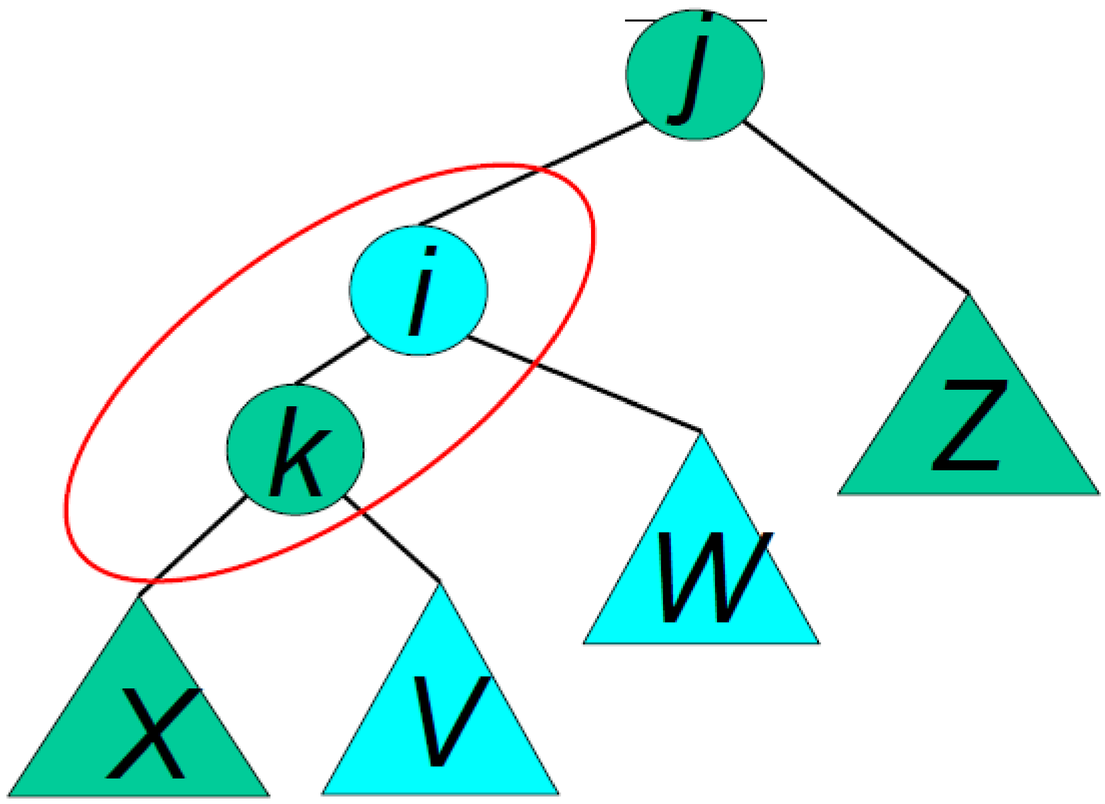
AVL Insertion: Inside Case

- We will do a left-right “double rotation” . . .



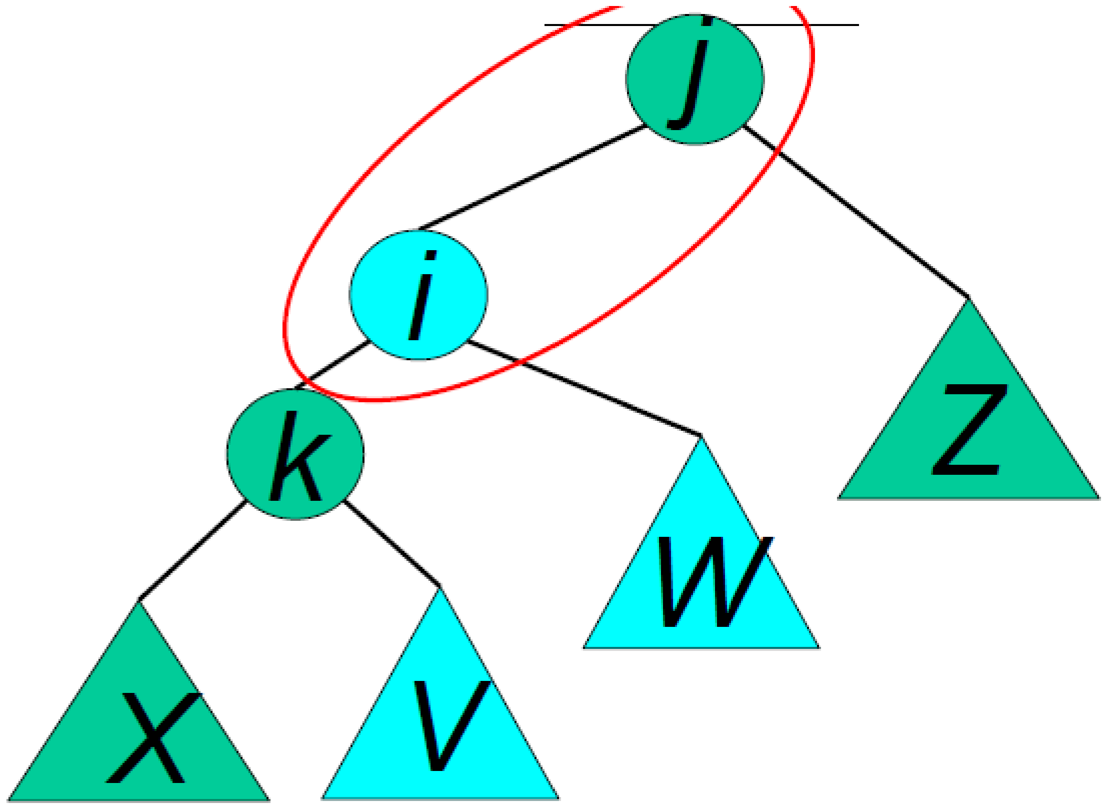
Double rotation : first rotation

- left rotation complete

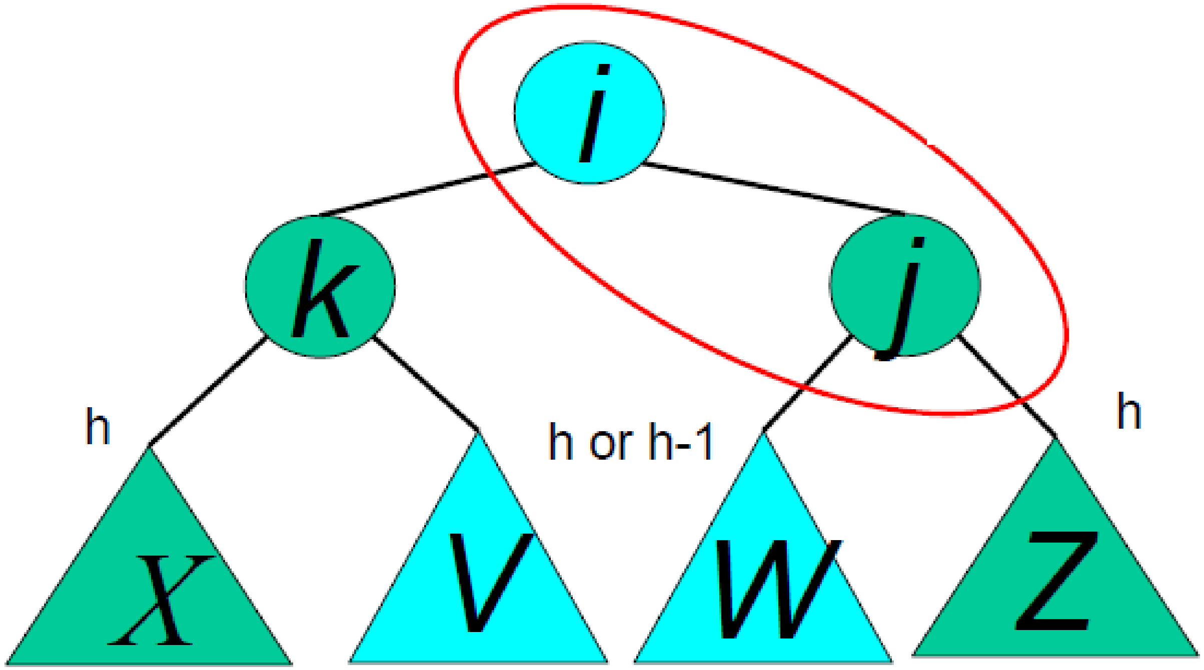


Double rotation : second rotation

- Then do a right rotation



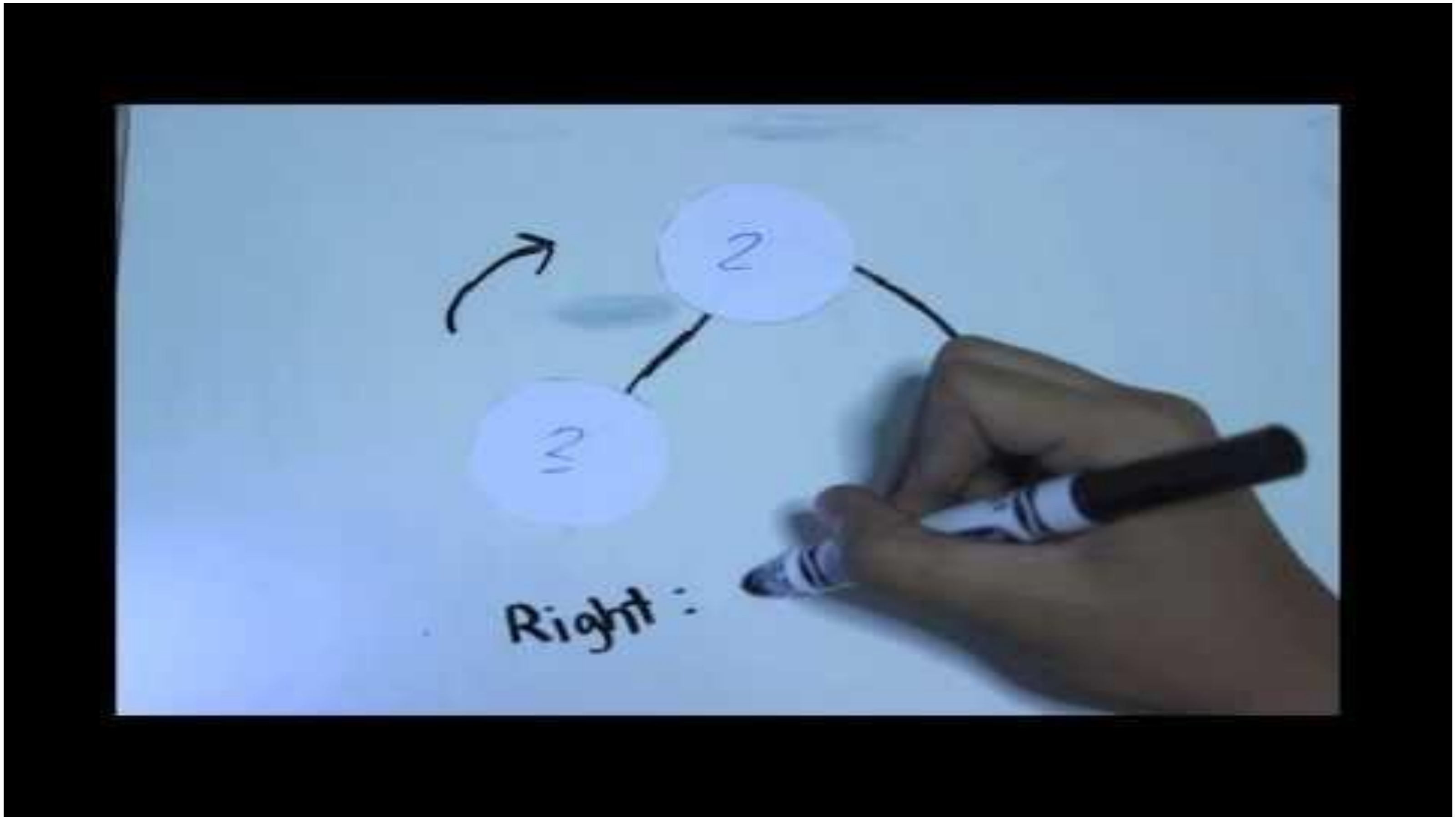
Double rotation : second rotation



- right rotation complete: Balance has been restored

AVL Tree – Insertion and Rotation

Some mistakes in the video: At 7:36, It is a Left Rotation, not Right Rotation.
At 8:40, the 5 node was accidentally placed on the right side. It should be on the left.



URL: https://www.youtube.com/watch?v=C2wec04RcZg

End of Tree (Part 2)
Design pattern example and AVL tree

You can check these links out for extra reading:

- Binary Trees

http://math.hws.edu/eck/cs225/s03/binary_trees/

- AVL Trees and Where to Rotate Them

https://medium.com/@sarahzhao25/avl-trees-where-to-find-rotate-them-7b062e0a30f8