AVL Trees

Slides and figures have been collected from various publicly available Internet sources for preparing the lecture slides of IT2001 course. I acknowledge and thank all the original authors for their contribution to prepare the content.

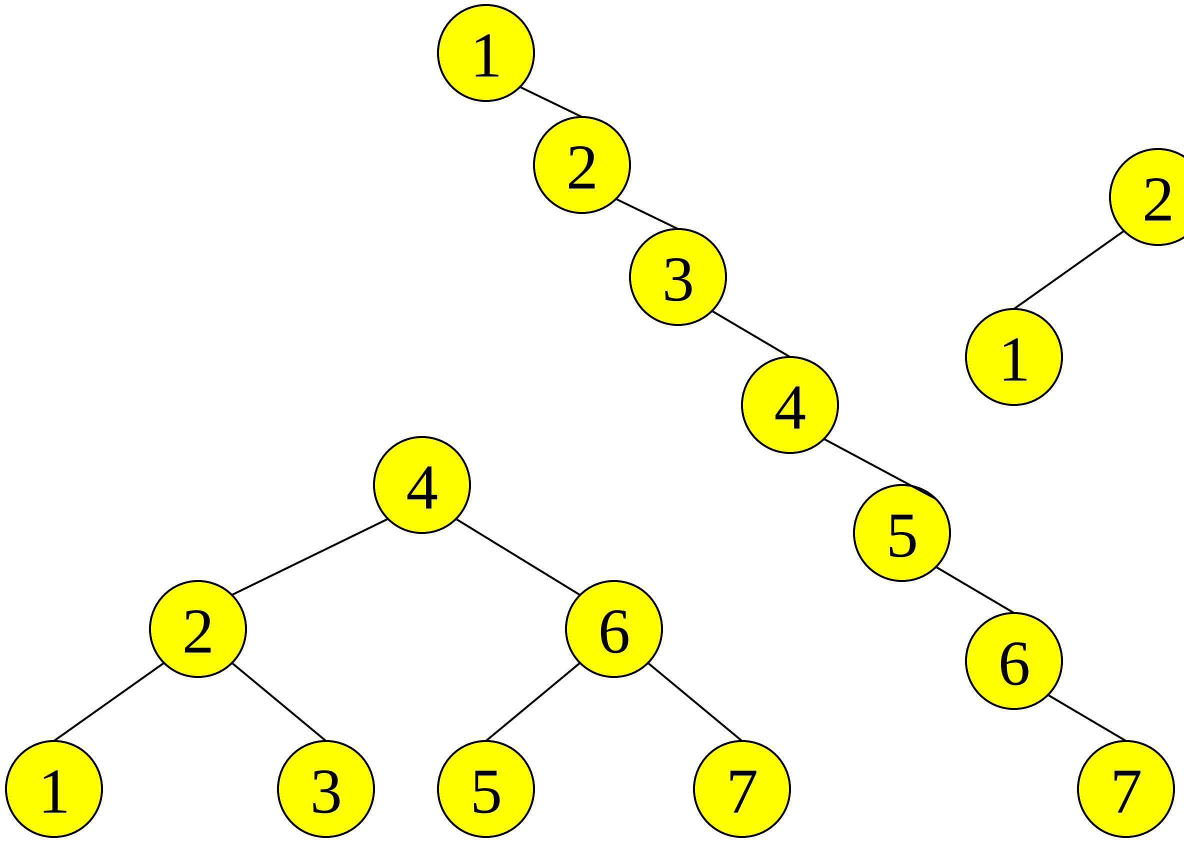
Binary Search Tree - Best Time

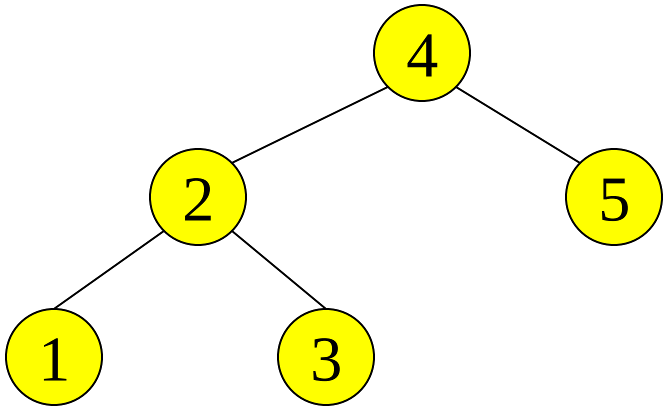
- All BST operations are O(h), where h is the height of the tree.
- Minimum h is $h = \lfloor \lg N \rfloor$ for a binary tree with N nodes
  - › What is the best case tree?
  - › What is the worst case tree?
- So best case running time of BST operations (e.g., insertion, searching, deletion, find min) is O(lg N)

Binary Search Tree - Worst Time

- Worst case running time is O(N)
  - › What happens when you Insert elements in ascending order?
    - Insert: 2, 4, 6, 8, 10, 12 into an empty BST
  - › Problem: Lack of “balance”:
    - Compare depths of left and right subtree
  - › Unbalanced degenerate tree

Balanced and unbalanced BST





Is this “balanced”?

Approaches to balancing trees

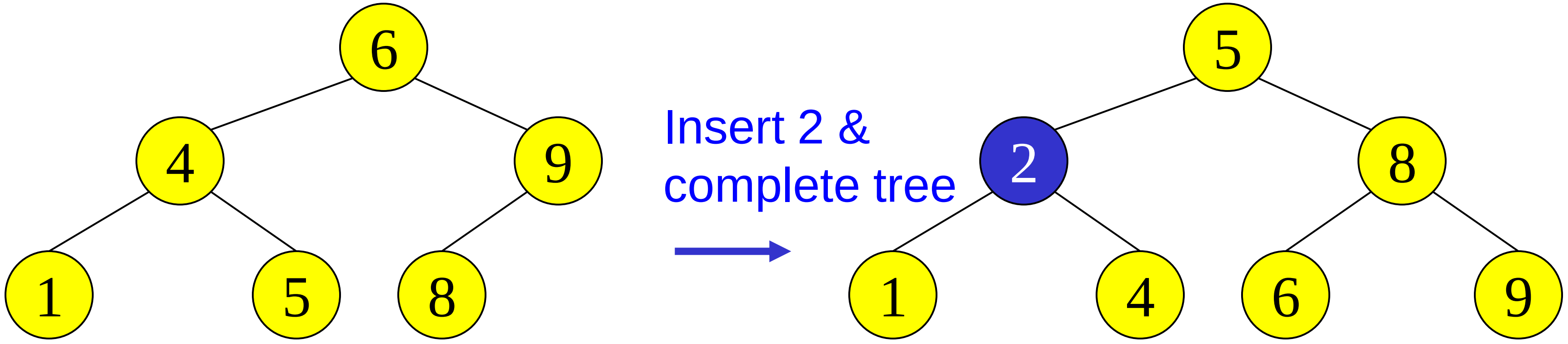
- Don't balance
  - May end up with some nodes very deep
- Strict balance
  - The tree must always be balanced perfectly
- Pretty good balance
  - Only allow a little out of balance
- Adjust on access
  - Self-adjusting

Balanced Search Trees

- Many algorithms exist to keep search trees balanced
  - Adelson-Velskii and Landis (AVL) trees (height-balanced binary search trees).
  - red-black trees
  - Splay trees and other self-adjusting trees
  - B-trees and other multiway search trees

Perfect Balance

- Want a complete tree after every operation
  - tree is full except possibly in the lower right
- This is expensive
  - For example, insert 2 in the tree on the left and then rebuild as a complete tree



AVL - Good but not Perfect Balance

- AVL trees are height-balanced binary search trees
- For every node x, define its balance factor

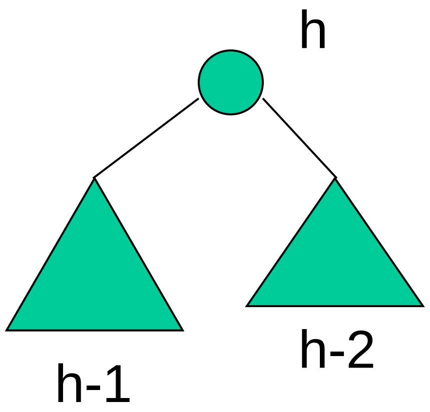
  balance factor of x = height of left subtree of x

  \- height of right subtree of x

- 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
  - Balance factor of every node x is -1, 0, or 1
  - Store current heights in each node

Height of an AVL Tree

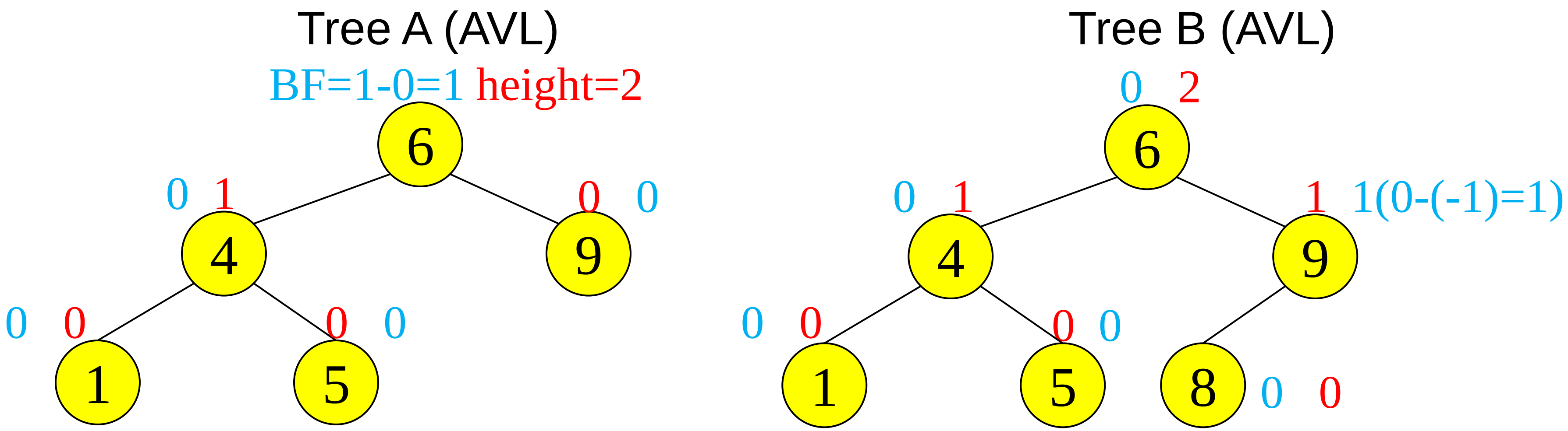
- $N(h)$ = minimum number of nodes in an AVL tree of height h.
- Basis
  - $N(0) = 1$, $N(1) = 2$
- Induction
  - $N(h) = N(h-1) + N(h-2) + 1$
- Solution (Fibonacci analysis)
  - $N(h) \geq \phi^h \quad (\phi \approx 1.62)$



Height of an AVL Tree

- $N(h) \geq \phi^h$ ($\phi \approx 1.62$)
- Suppose we have n nodes in an AVL tree of height h.
  - $n \geq N(h)$ (because N(h) was the minimum)
  - $n \geq \phi^h$ hence $\log_\phi n \geq h$ (relatively well balanced tree!!)
  - $h \leq 1.44 \log_2 n$ (i.e., Find takes O(lgn))

Node Heights and Balance factor

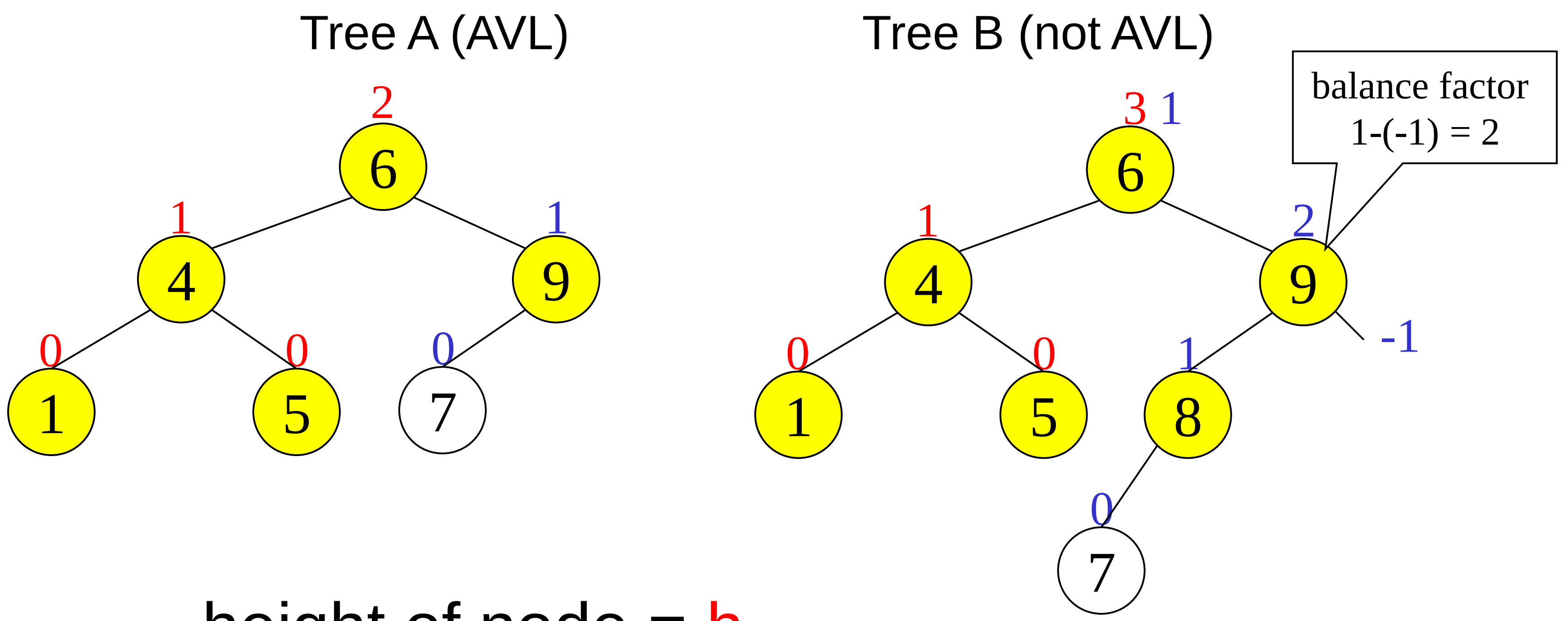


height of node = h

balance factor = h_{left}-h_{right}

Empty node height = -1

Node Heights (after Inserting 7)



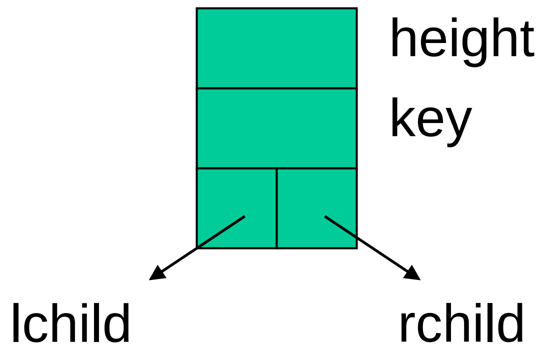
height of node = h

balance factor = h_{left}-h_{right}

empty height = -1

Implementation

```
typedef struct avlNode{
 int data;
   struct avlNode *lchild,*rchild;
   int height;
}avlNode;
```



Height of a AVL Tree

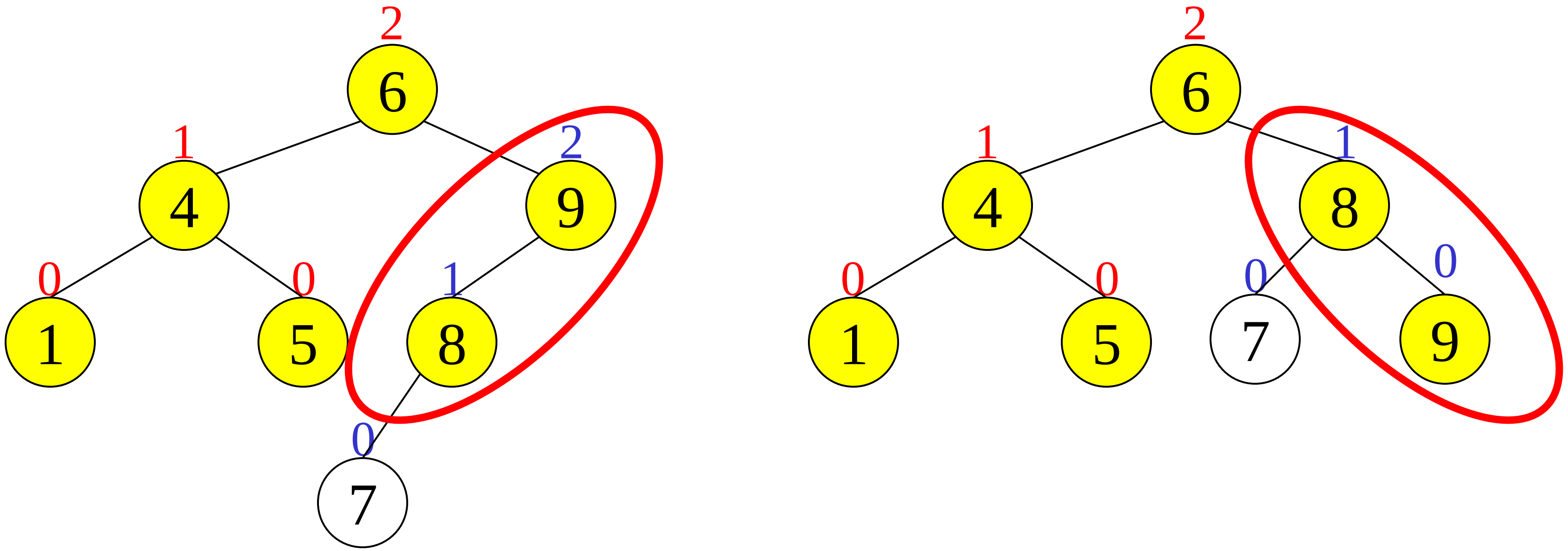
```
int height(avlNode *T)       // to find height of the subtree at node T
{
    int lheight, rheight;        //variables for height of left and right subtrees
    if (T == NULL)
        return (-1);
    if (T->lchild == NULL)
        lheight = 0;
     else
        lheight = 1 + T->lchild->height;
    if (T->rchild == NULL)
        rheight = 0;
     else
        rheight = 1 + T->rchild->height;
    if (lheight > rheight)
        return    lheight;
    else return    rheight;
}
```

```
int BF(avlNode *T)
//to find balance factor of T
{
    //get height of left (lheight) and
right (rheight) subtrees
    return(lheight-rheight);
}
```

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



Insertions in AVL Trees

Let the node that needs rebalancing be V.

There are 4 cases:

Outside Cases (require single rotation) :

1. Insertion into left subtree of left child of V (LL case).
2. Insertion into right subtree of right child of V(RR case).

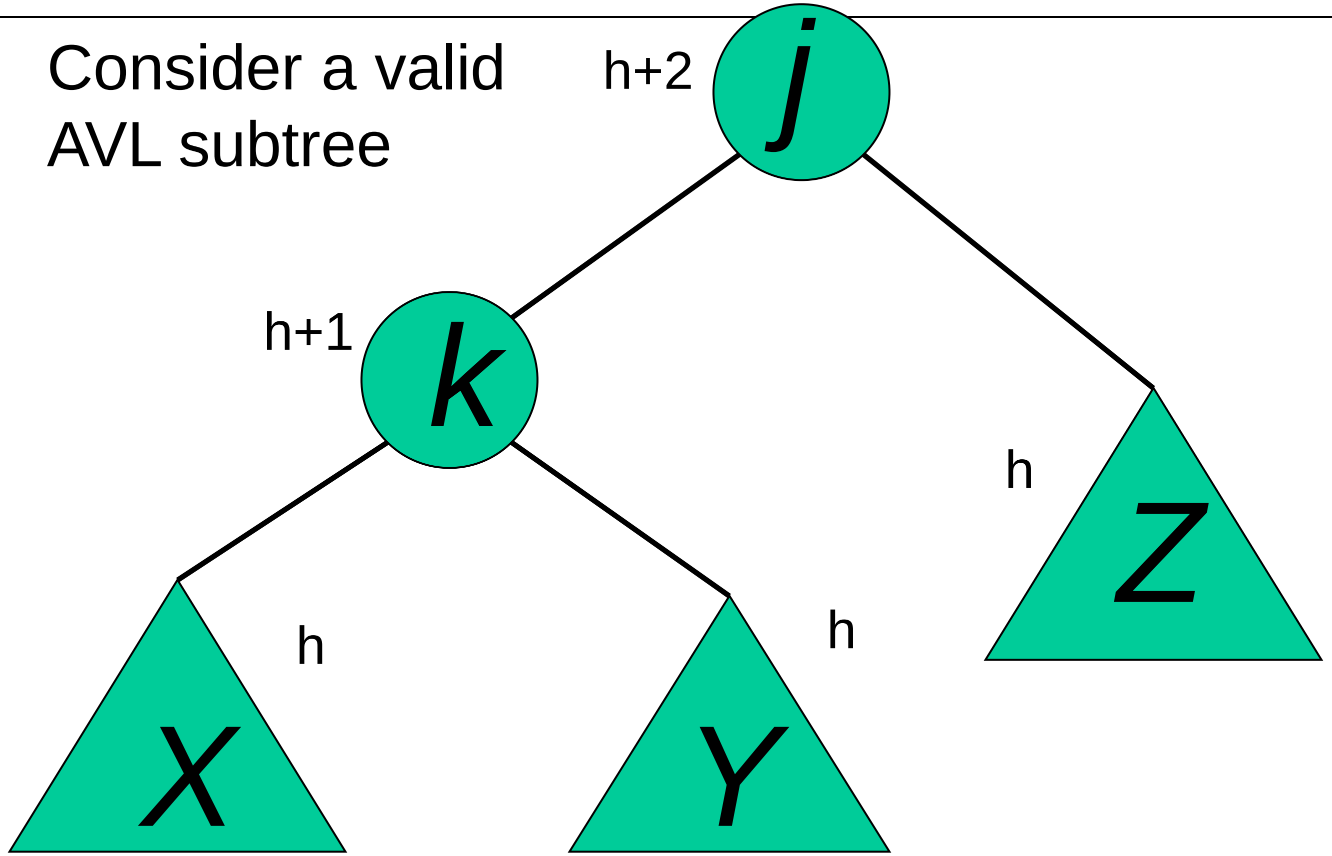
Inside Cases (require double rotation) :

3. Insertion into right subtree of left child of V (RL case).
4. Insertion into left subtree of right child of V(LR case).

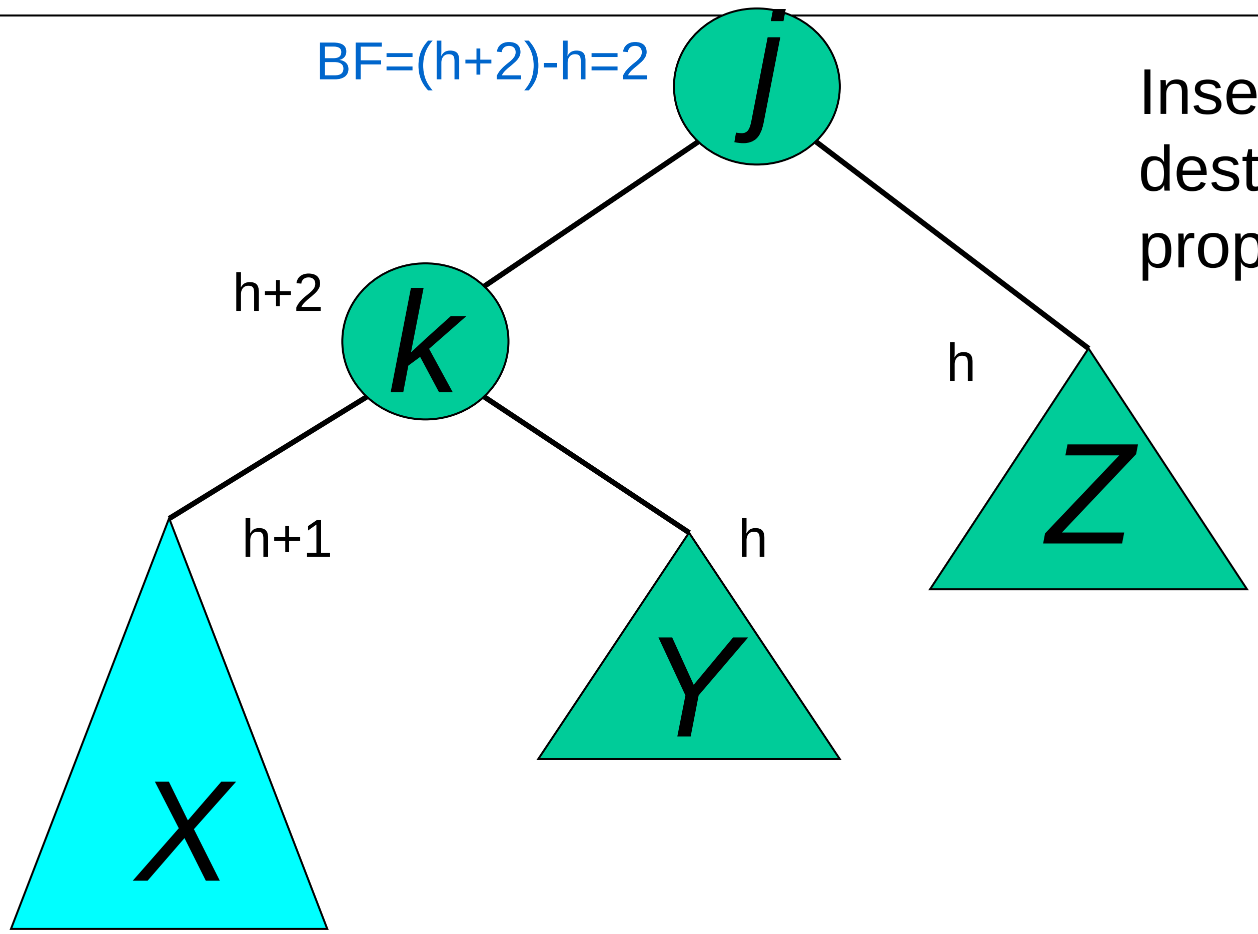
The rebalancing is performed through four separate rotation algorithms.

AVL Insertion: Outside Case

Consider a valid
AVL subtree



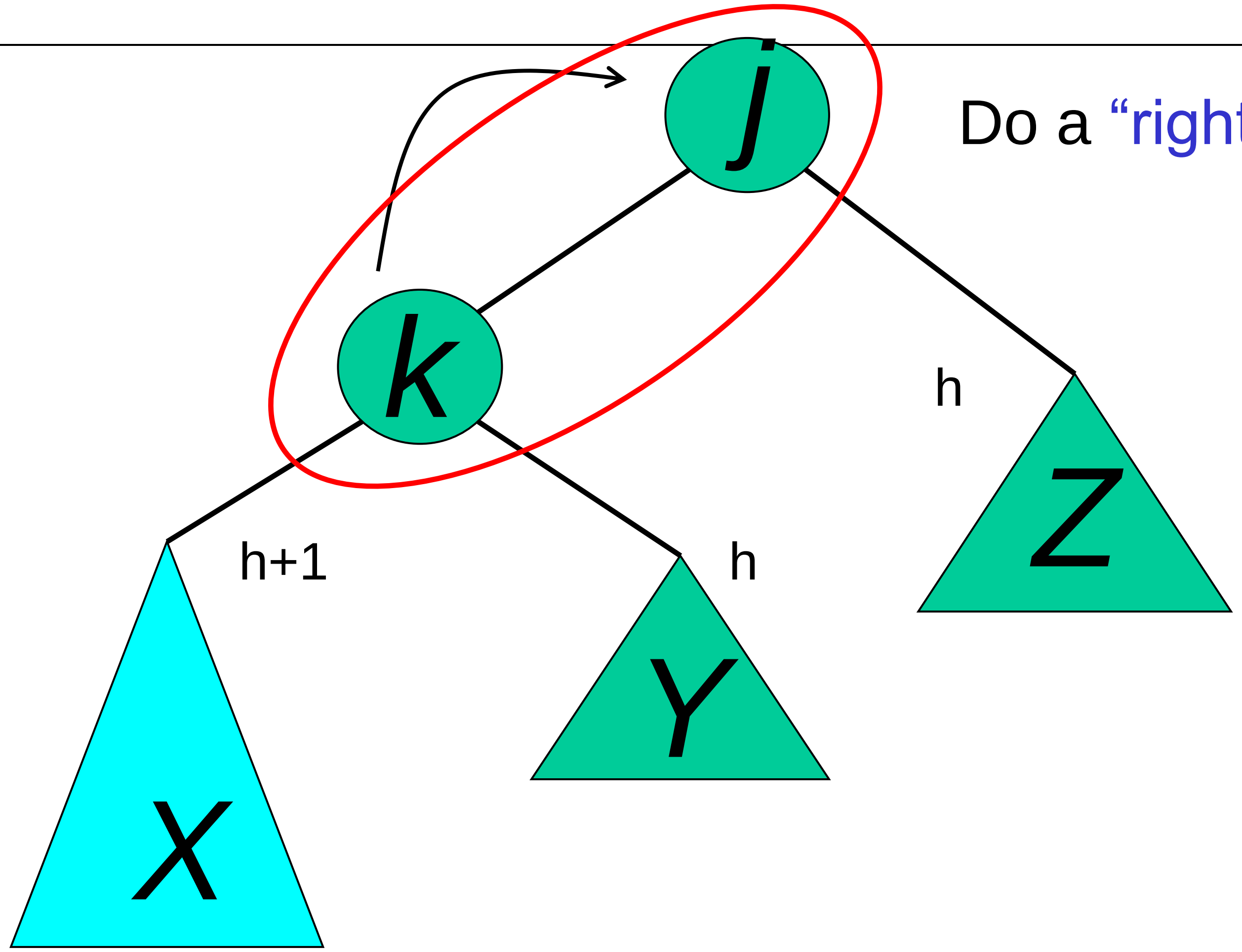
AVL Insertion: Outside Case



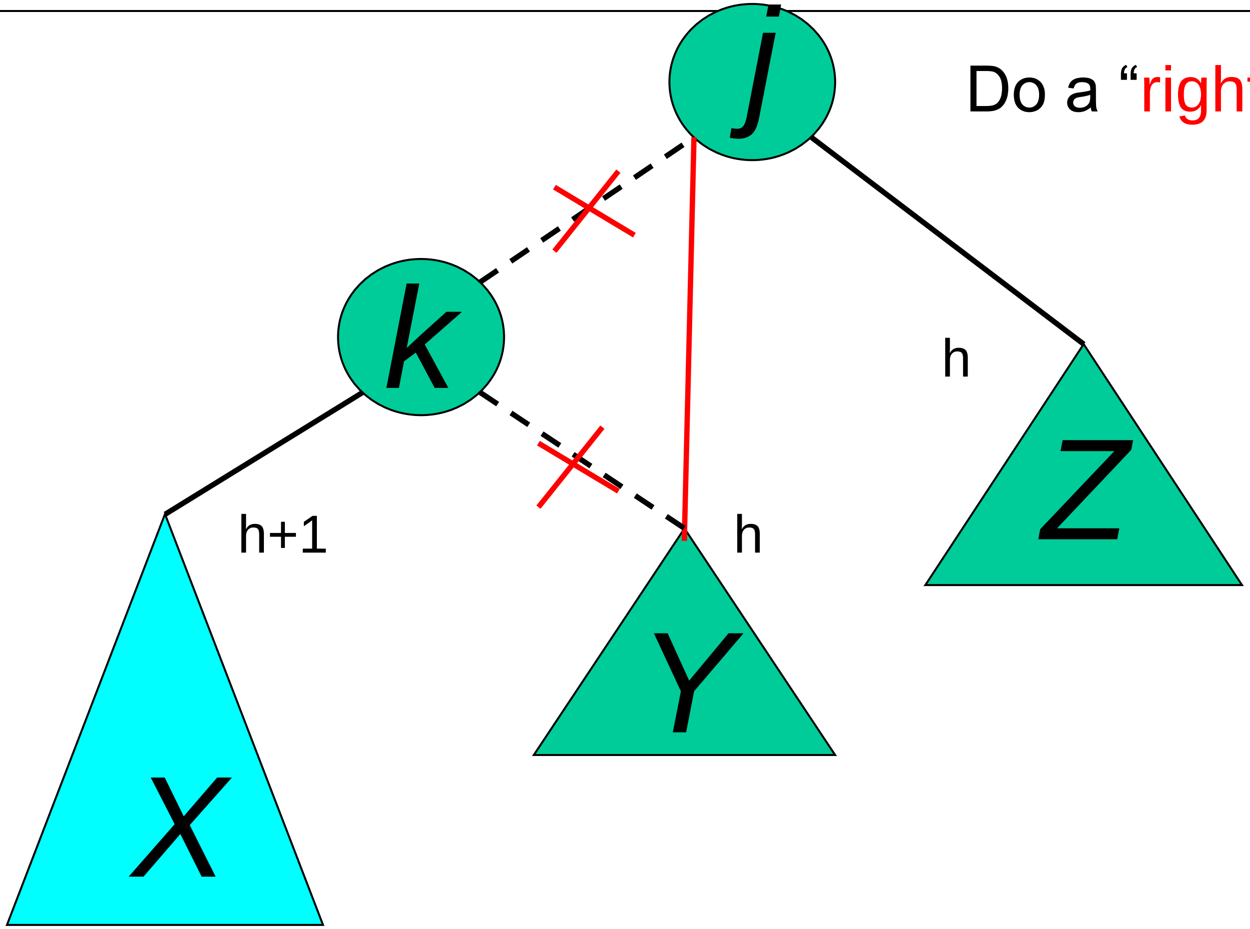
Inserting into X destroys the AVL property at node j

AVL Insertion: Outside Case

Do a “right rotation”

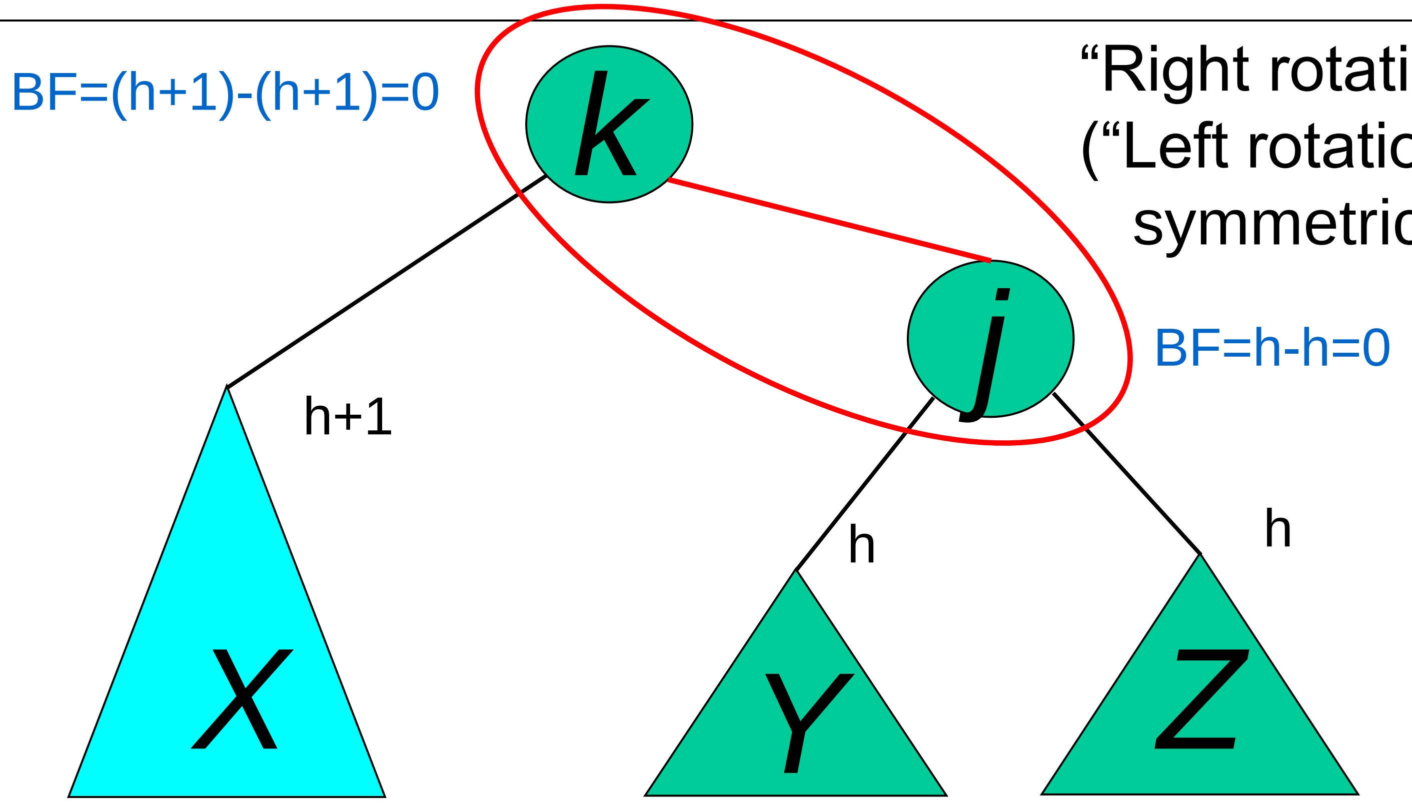


Single right rotation



Do a “right rotation”

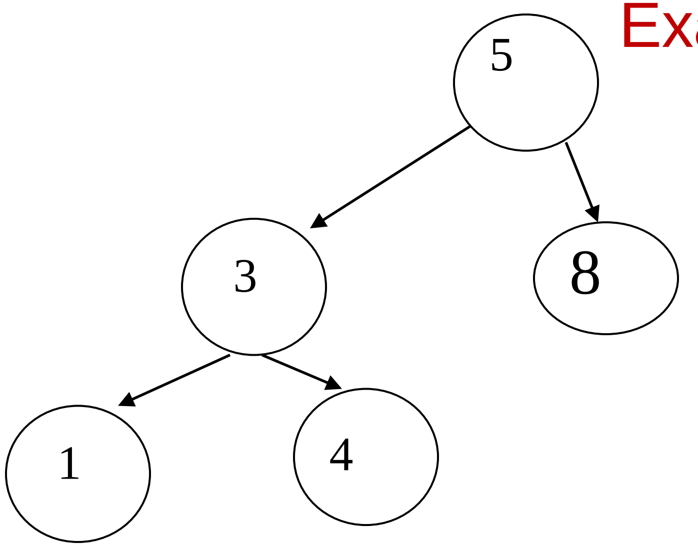
Outside Case Completed



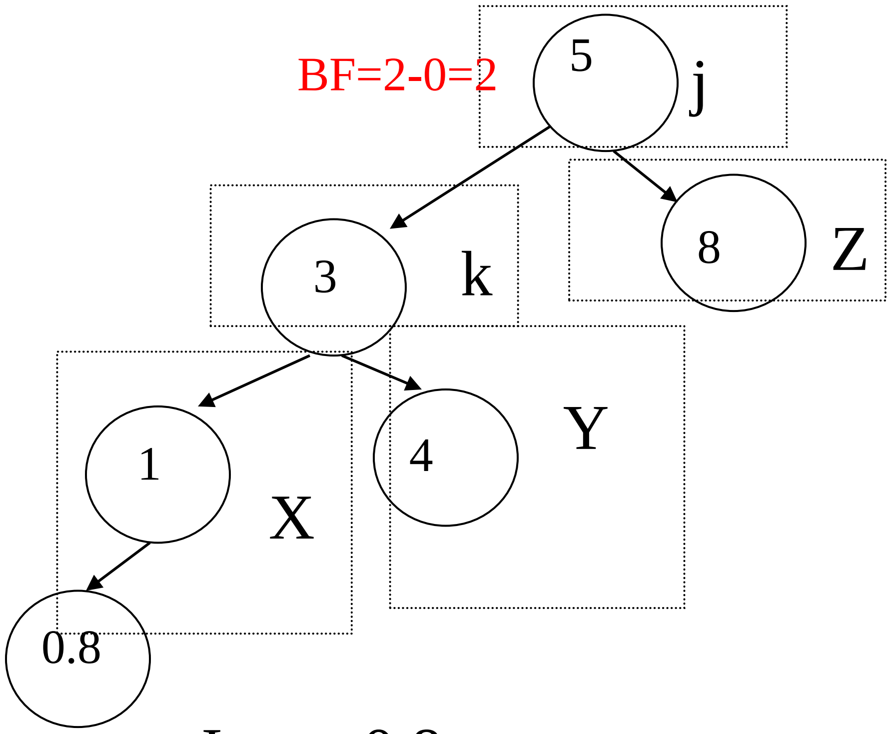
"Right rotation" done!
("Left rotation" is mirror symmetric)

AVL property has been restored!

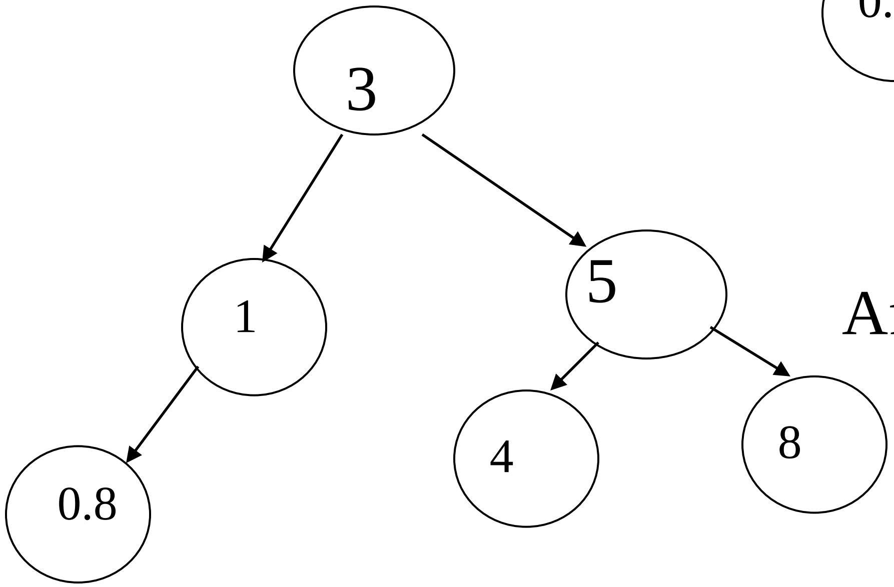
Example: Right Rotation



AVL Tree

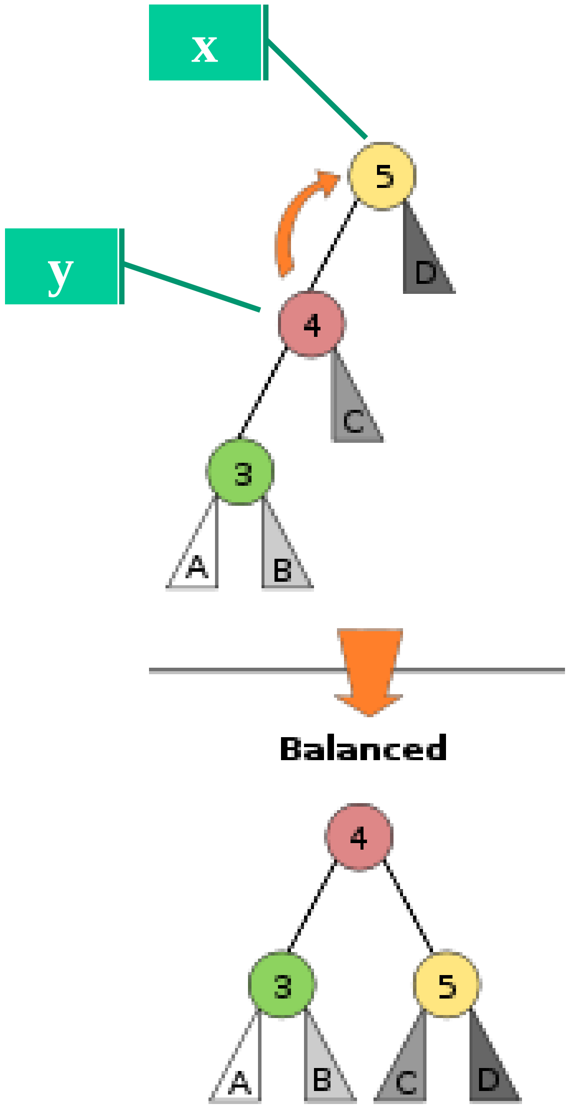


Insert 0.8



After Rotation

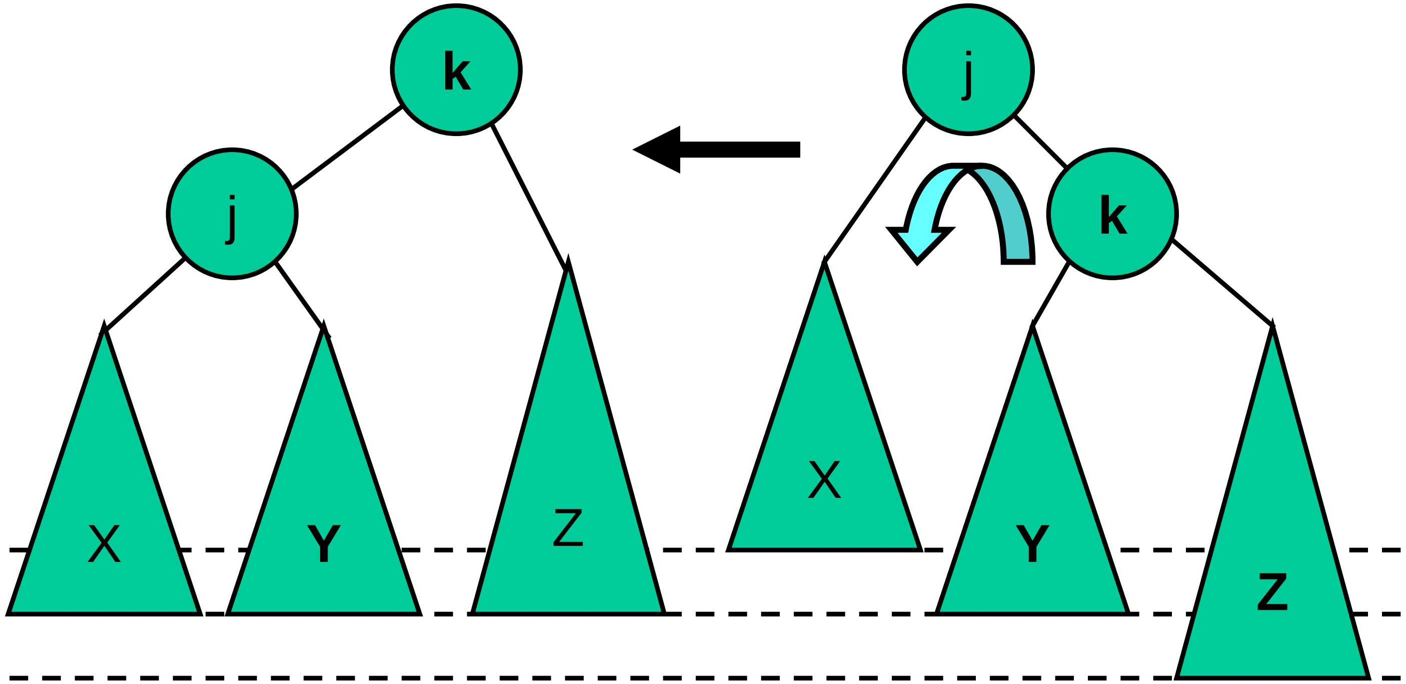
Right Rotation



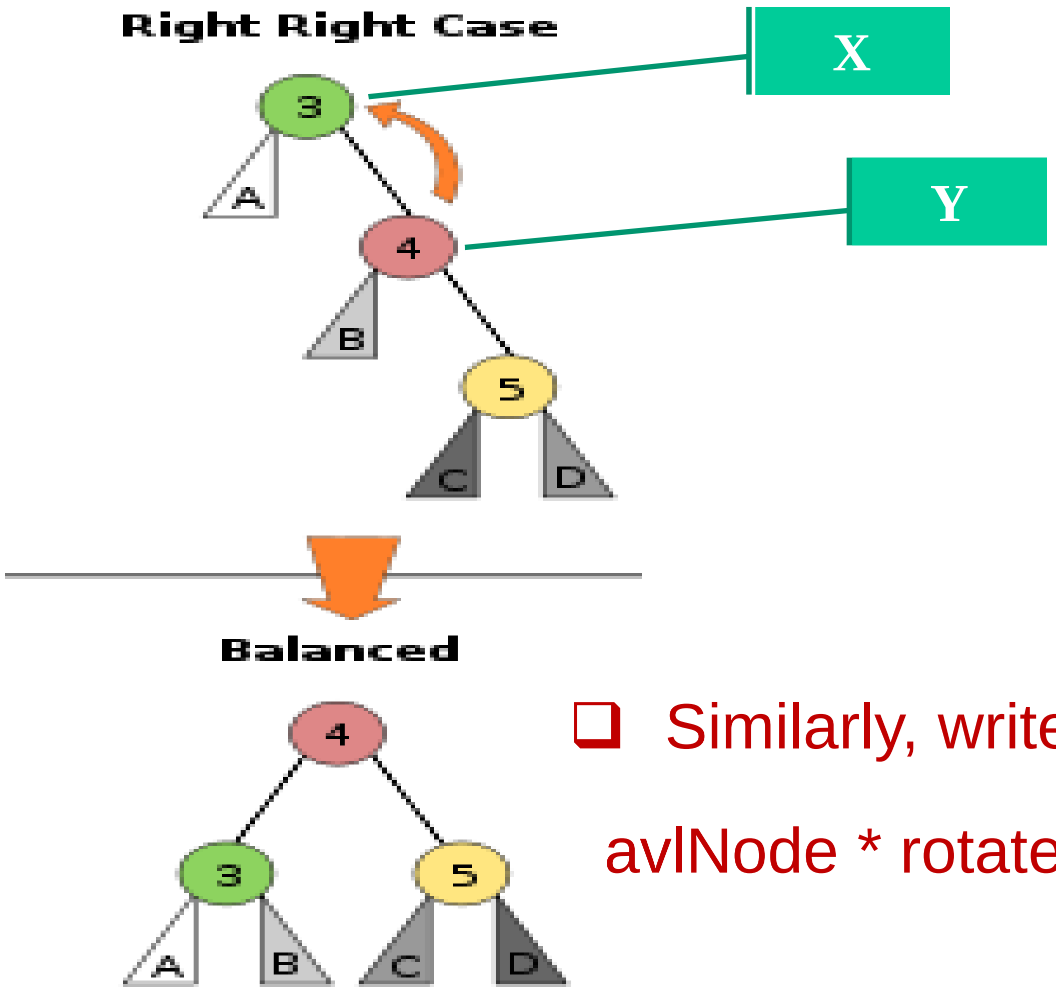

```
avlNode * rotate_right(avlNode *x)
{
    avlNode *y;
    y = x->lchild;
    x-> lchild = y->rchild;
    y-> rchild = x;
    x->height = height(x);
    y->height = height(y);
    return(y);
}
```

Single Rotation

Inserting into Z destroys the AVL property at node j



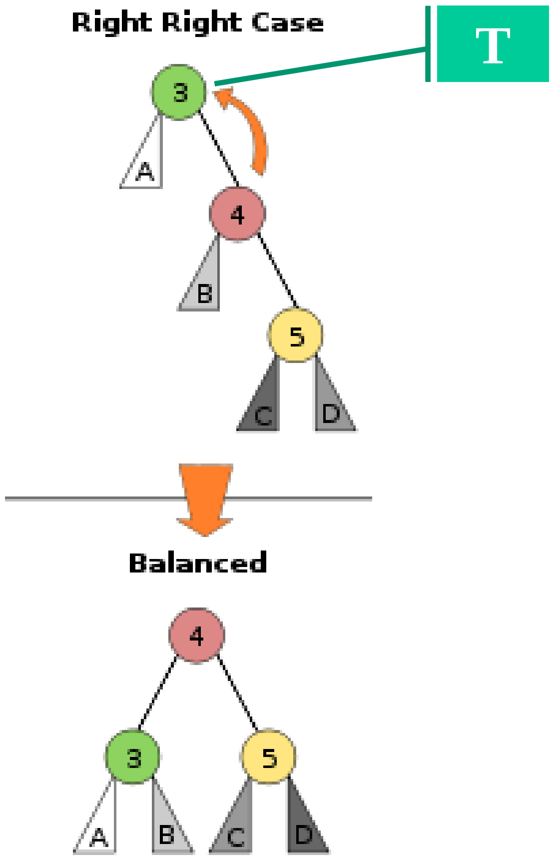
Left Rotations



- Similarly, write code for

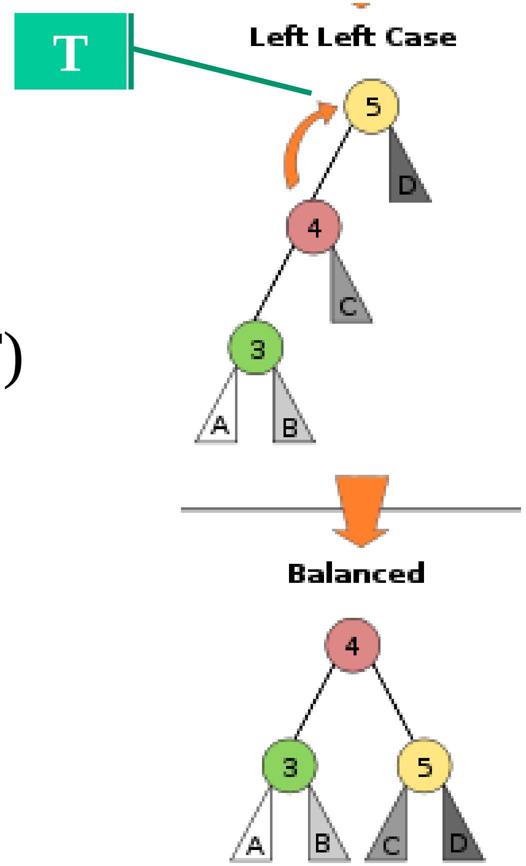
  avlNode * rotate_left(avlNode *x)

RR and LL Rotations



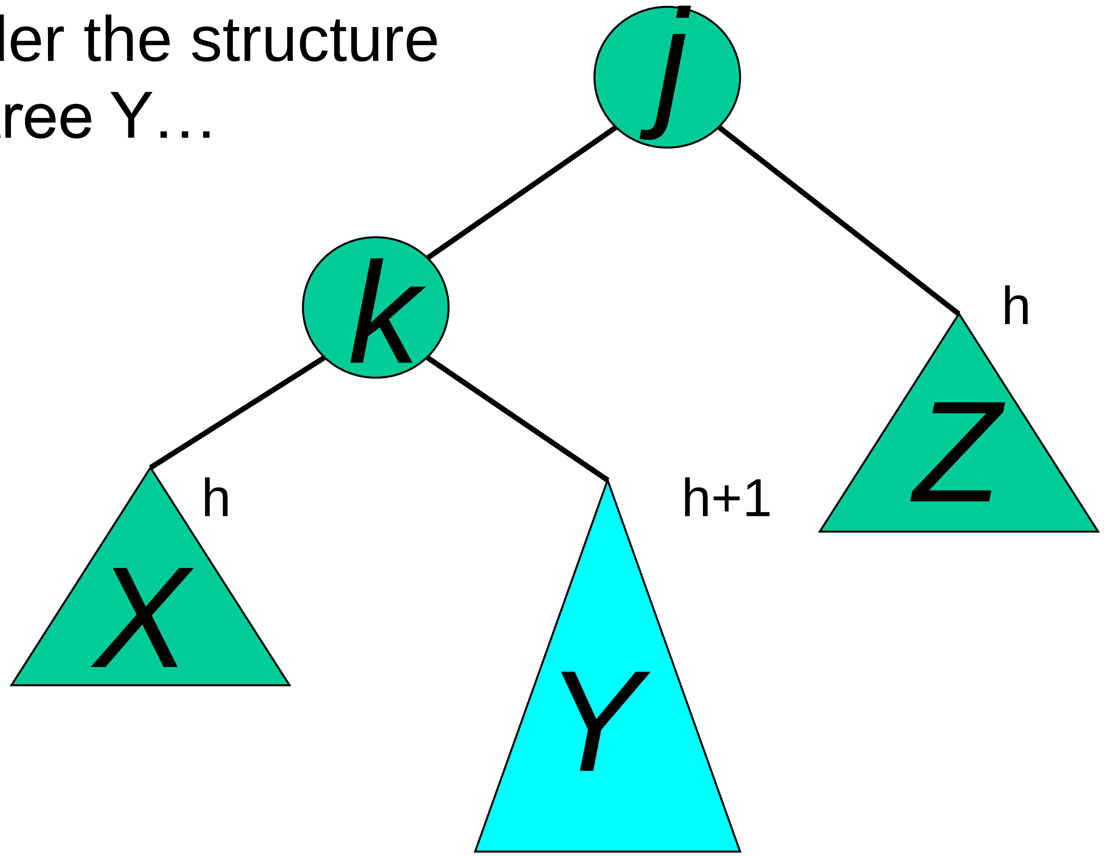

```
avlNode * RR(avlNode *T)
{
    T=rotate_left(T);
    return(T);
}
```

```
avlNode * LL(avlNode *T)
{
    T=rotate_right(T);
    return(T);
}
```



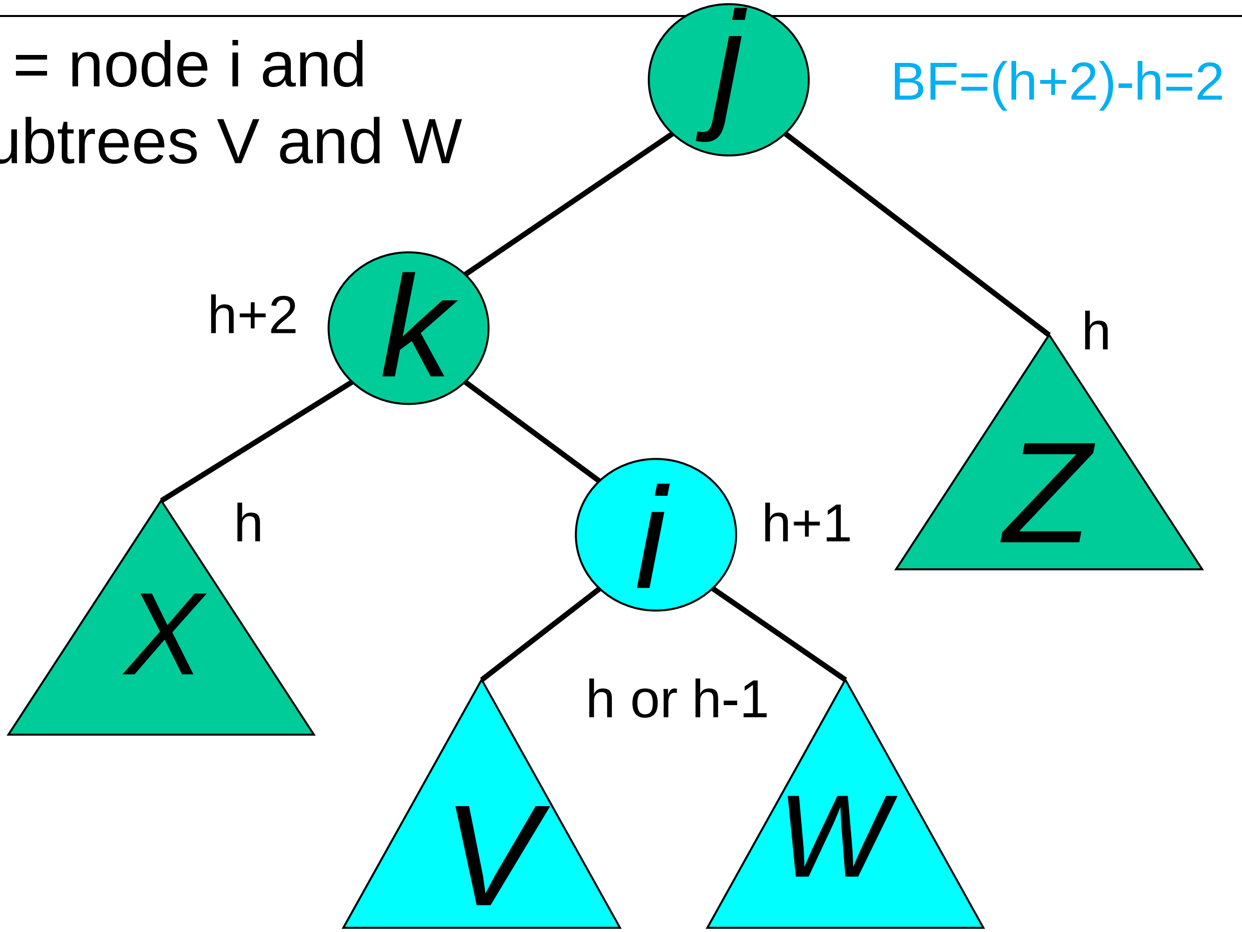
AVL Insertion: Inside Case

Consider the structure of subtree Y…

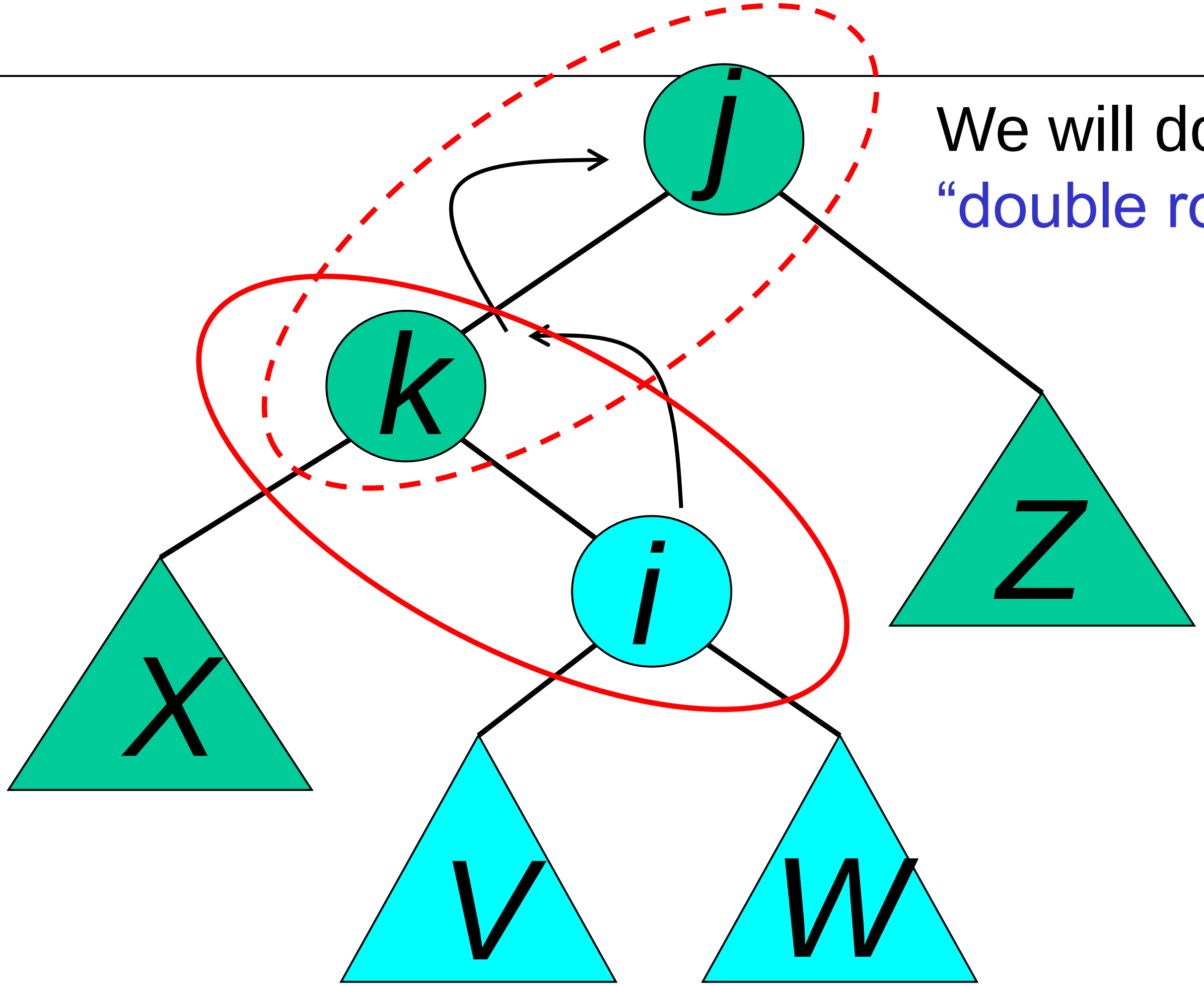


AVL Insertion: Inside Case

Y = node i and
subtrees V and W

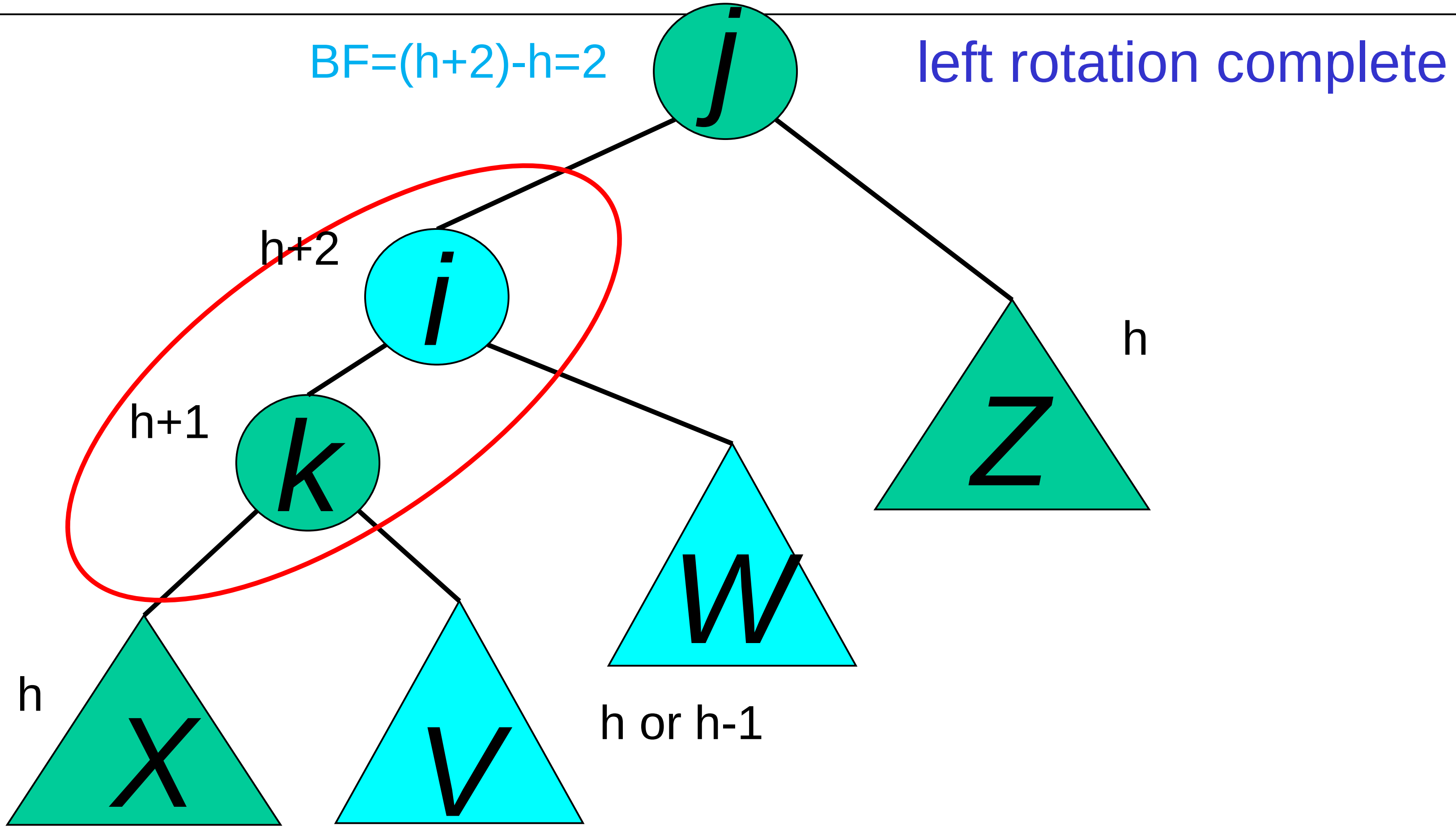


AVL Insertion: Inside Case

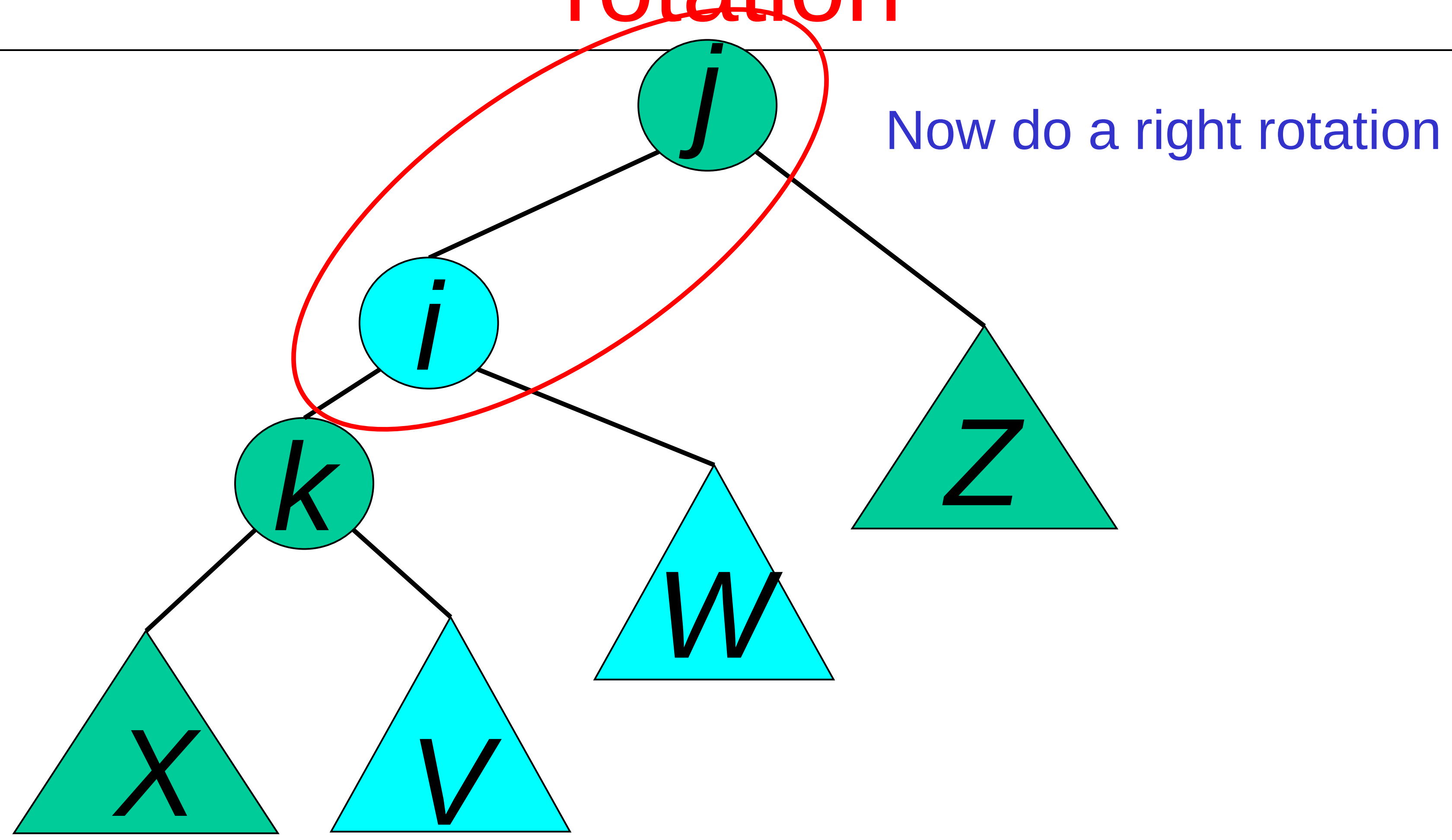


We will do a left-right "double rotation" . . .

Double rotation : first rotation



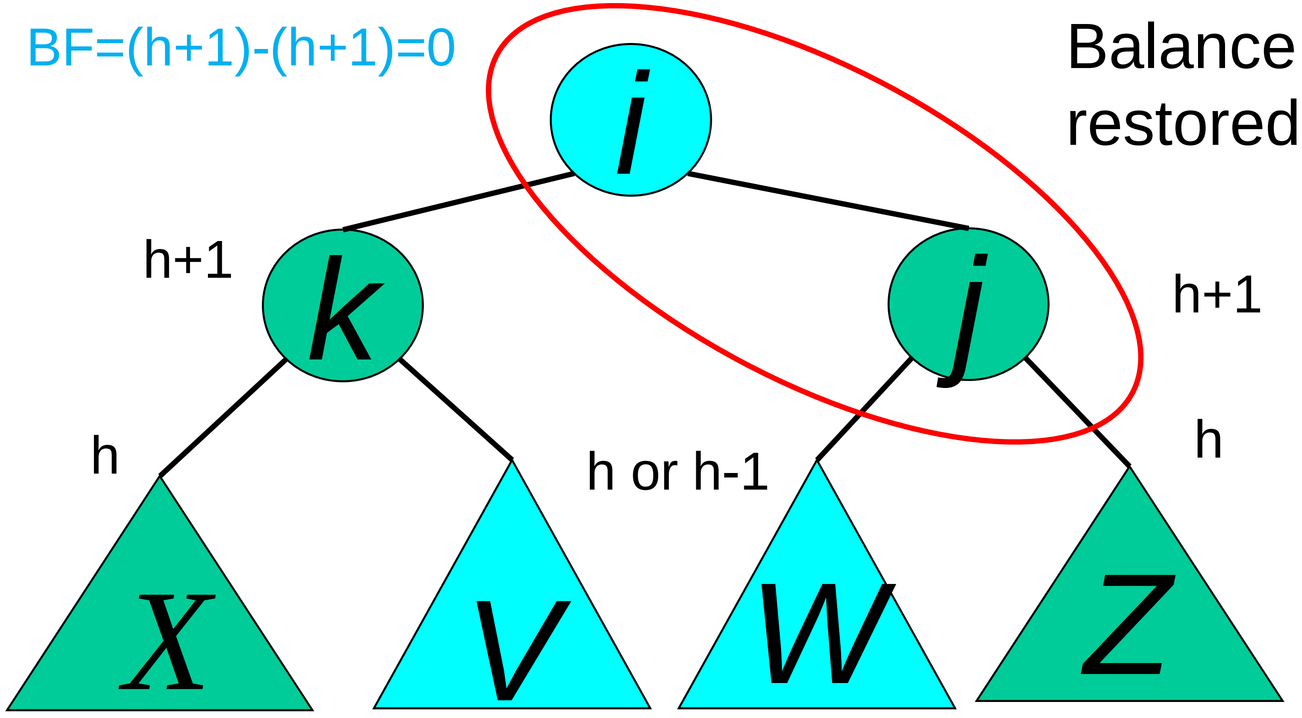
Double rotation : second rotation



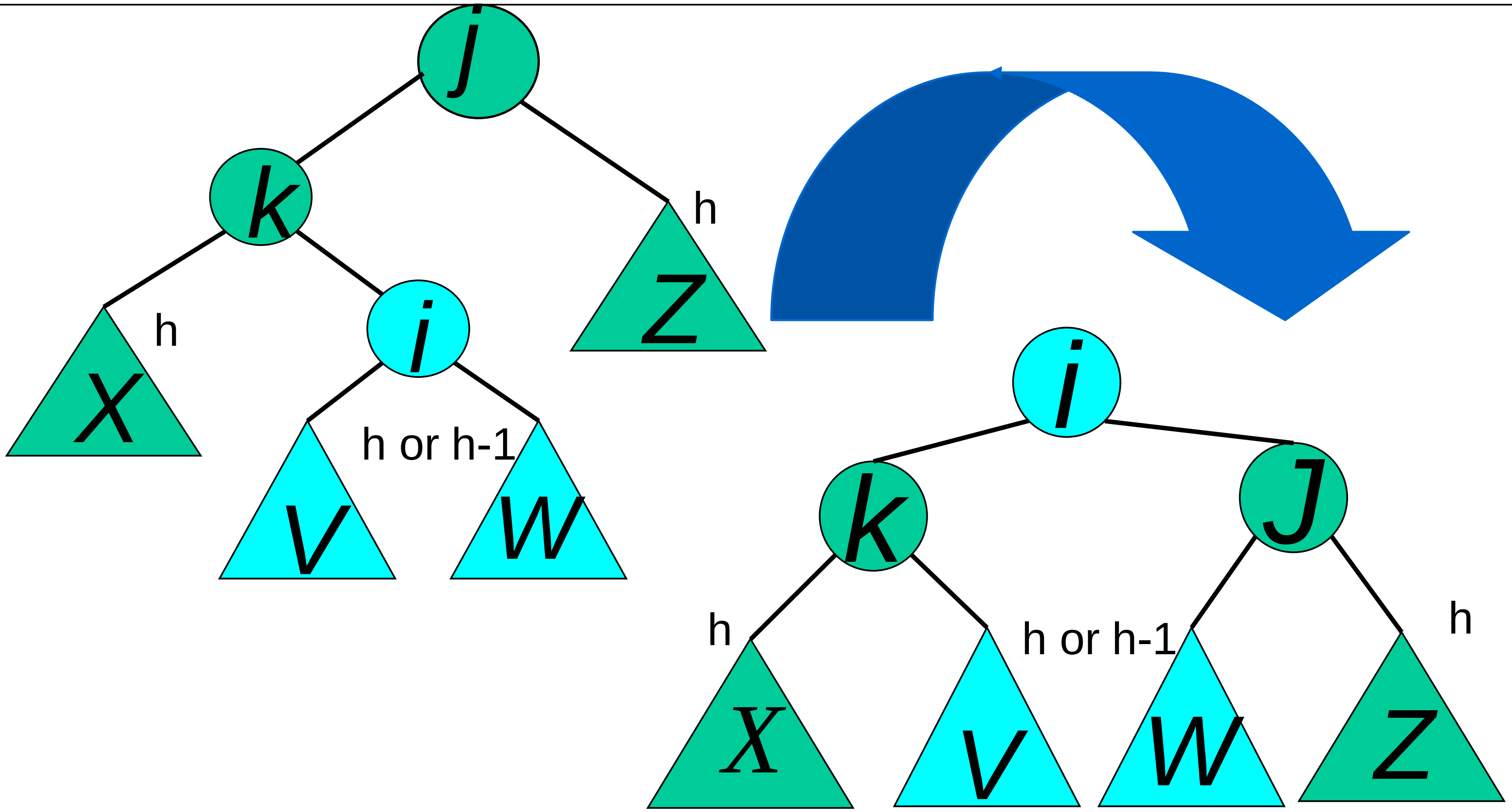
Double rotation : second rotation

right rotation complete

Balance has been restored

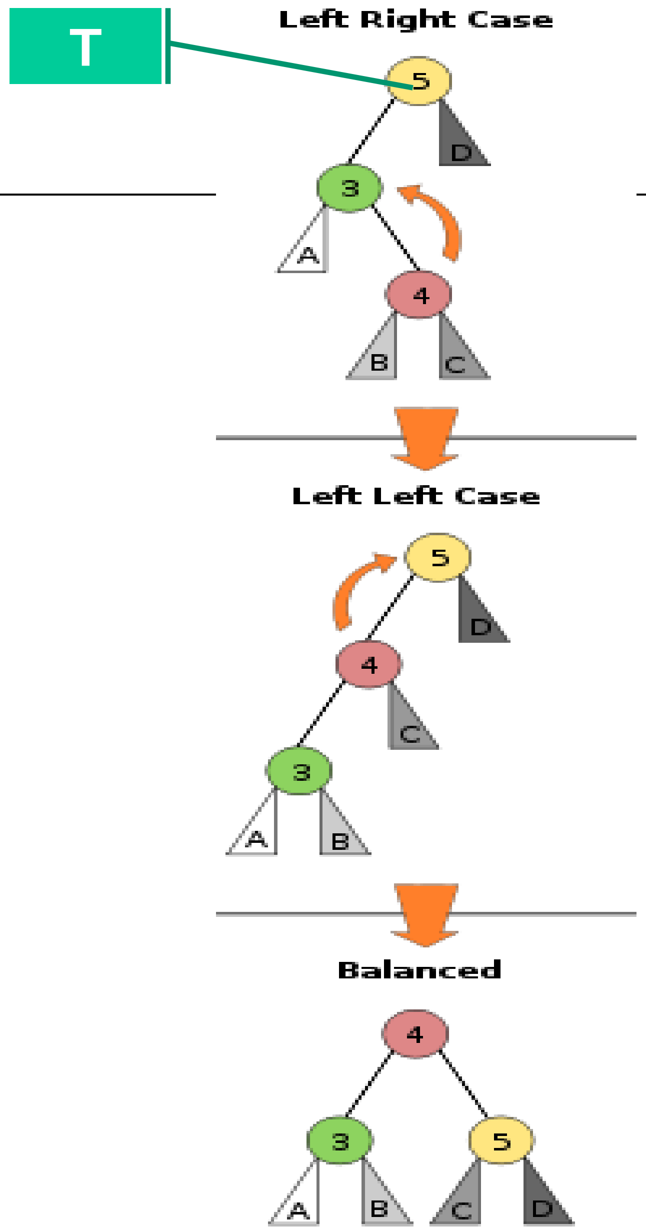


Double rotation

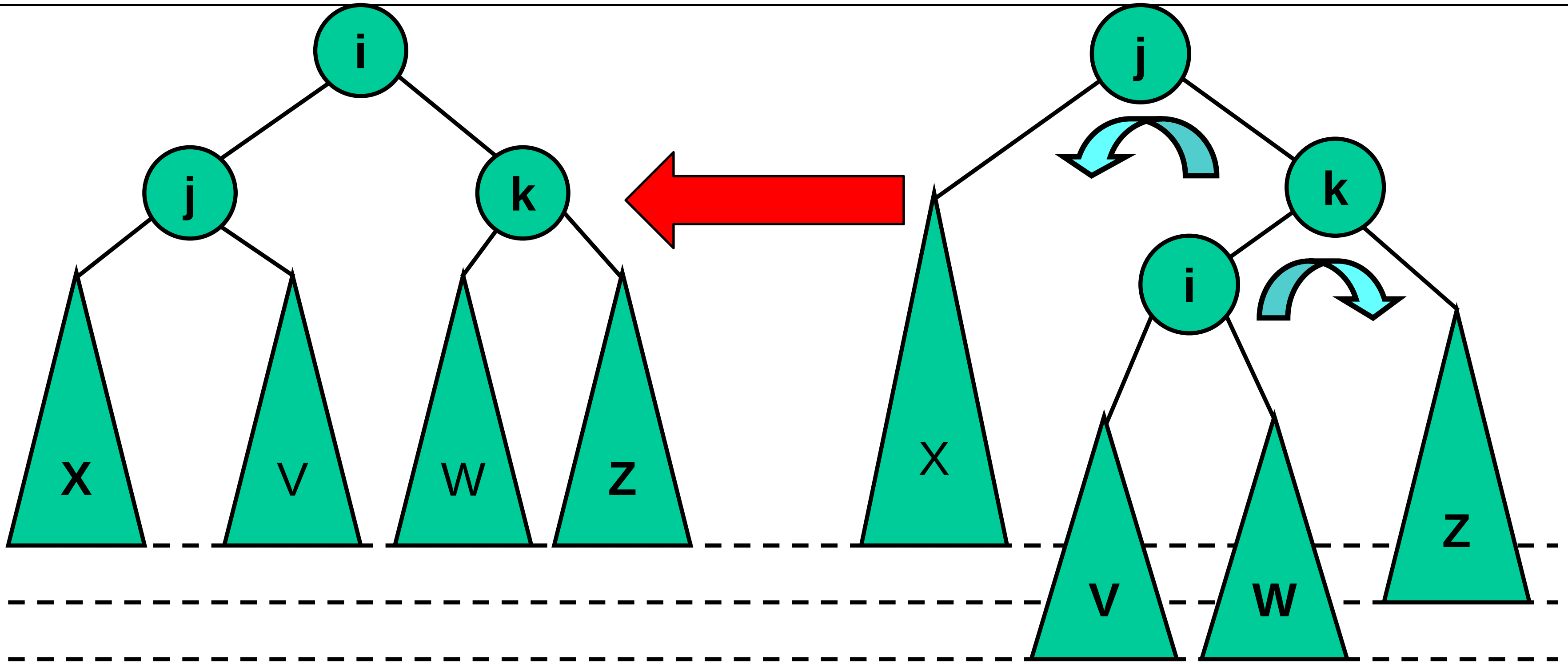


Left Right Rotations

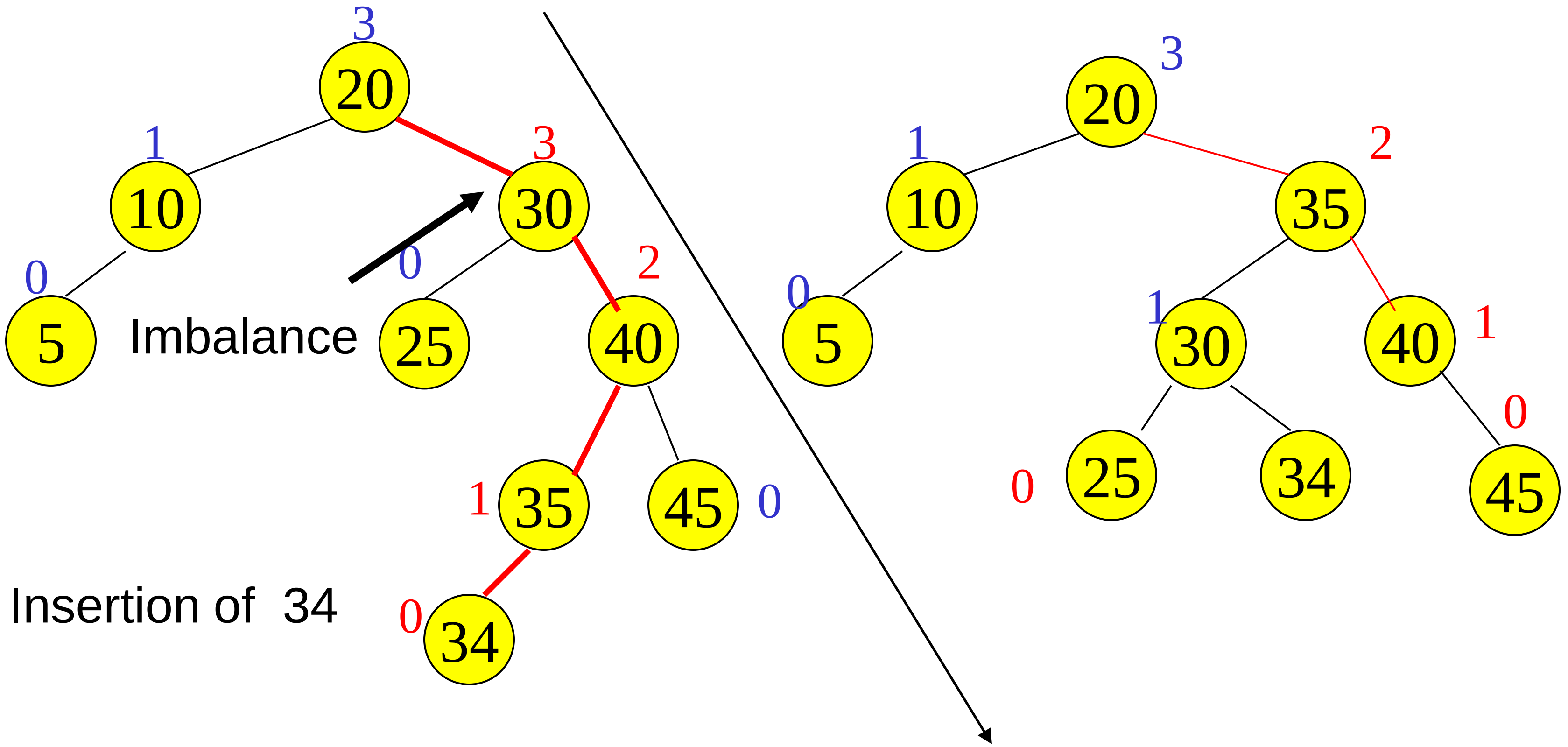
```
avlNode * LR(avlNode *T)
{
    T->lchild=rotate_left(T->lchild);
     T=rotate_right(T);
     return(T);
}
```



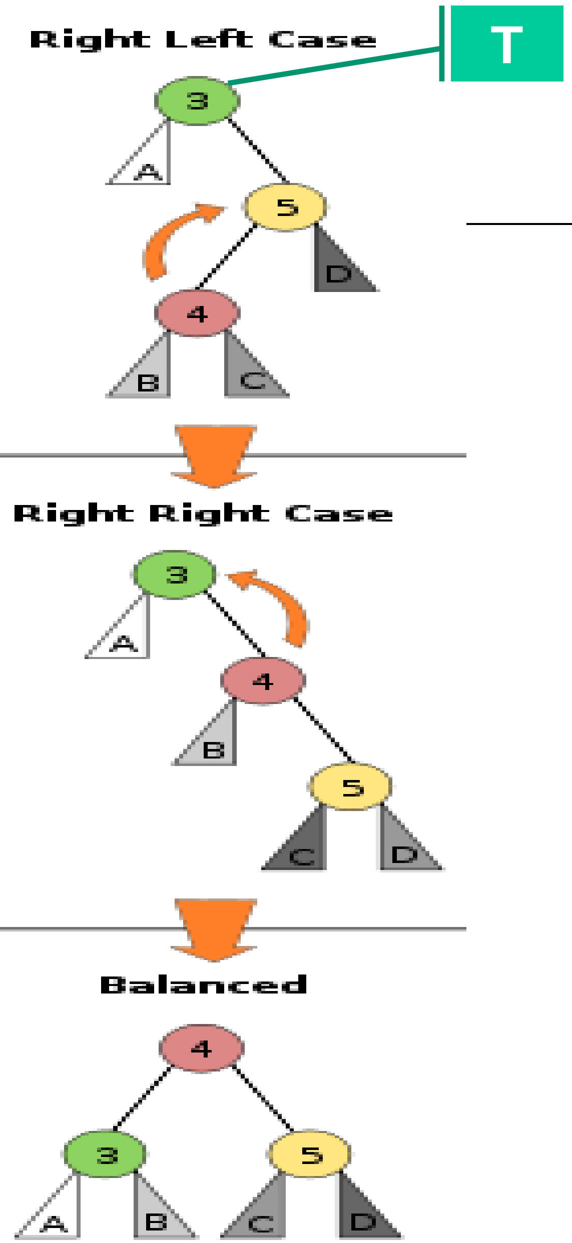
Double Rotation



Example: Double rotation (inside case)



Right Left Rotations

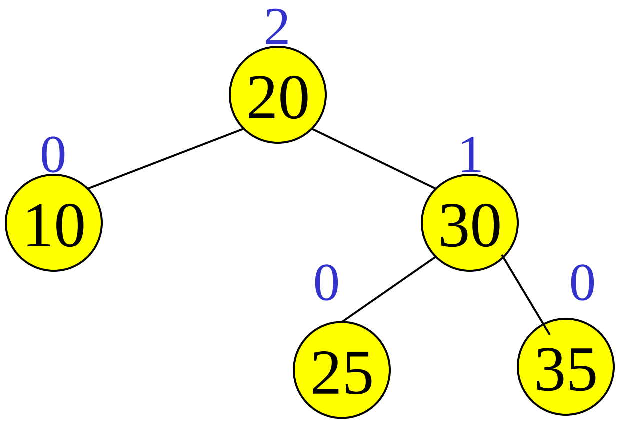



```
avlNode * RL(avlNode *T)
{
    T->rchild=rotate_right(T->rchild);
    T=rotate_left(T);
    return(T);
}
```

Insertion in AVL Trees

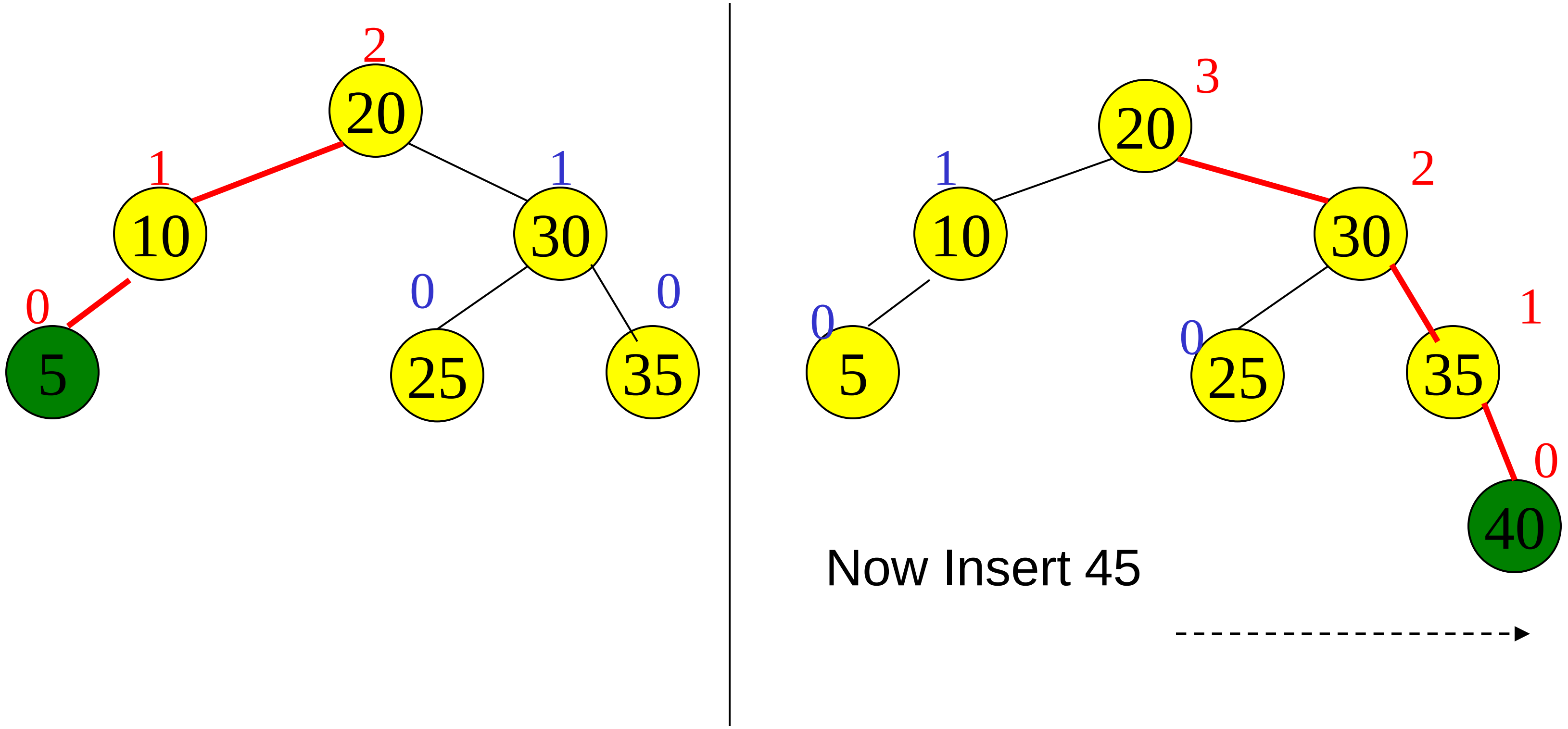
- Insert at the leaf (as for all BST)
  - Only nodes on the path from insertion point to root node have possibly changed in height
  - So after the Insert, go back up to the root node by node, updating heights
  - If a new balance factor (the difference h_{left}-h_{right}) is 2 or –2, adjust tree by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

Example of Insertions in an AVL Tree

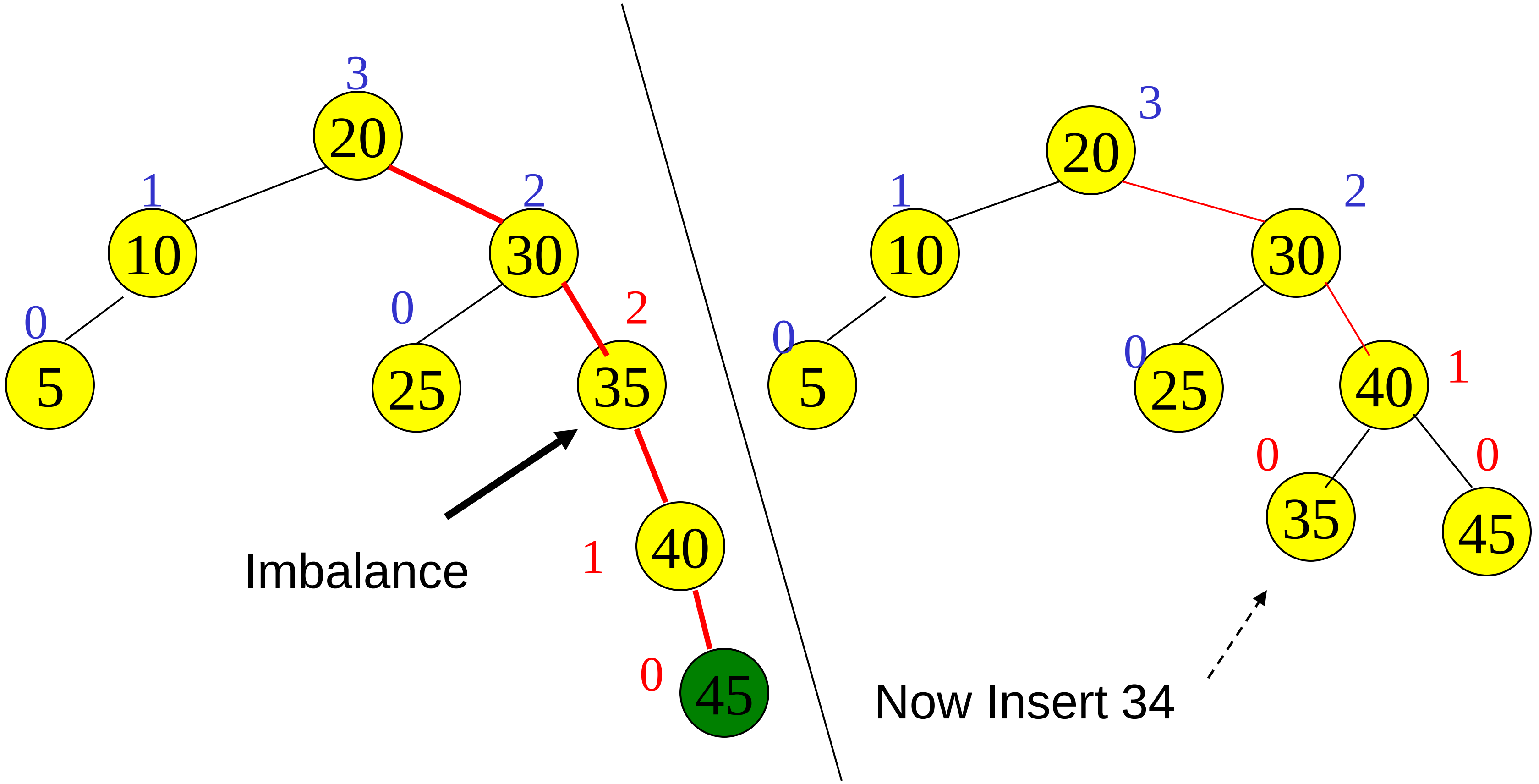


Insert 5, 40

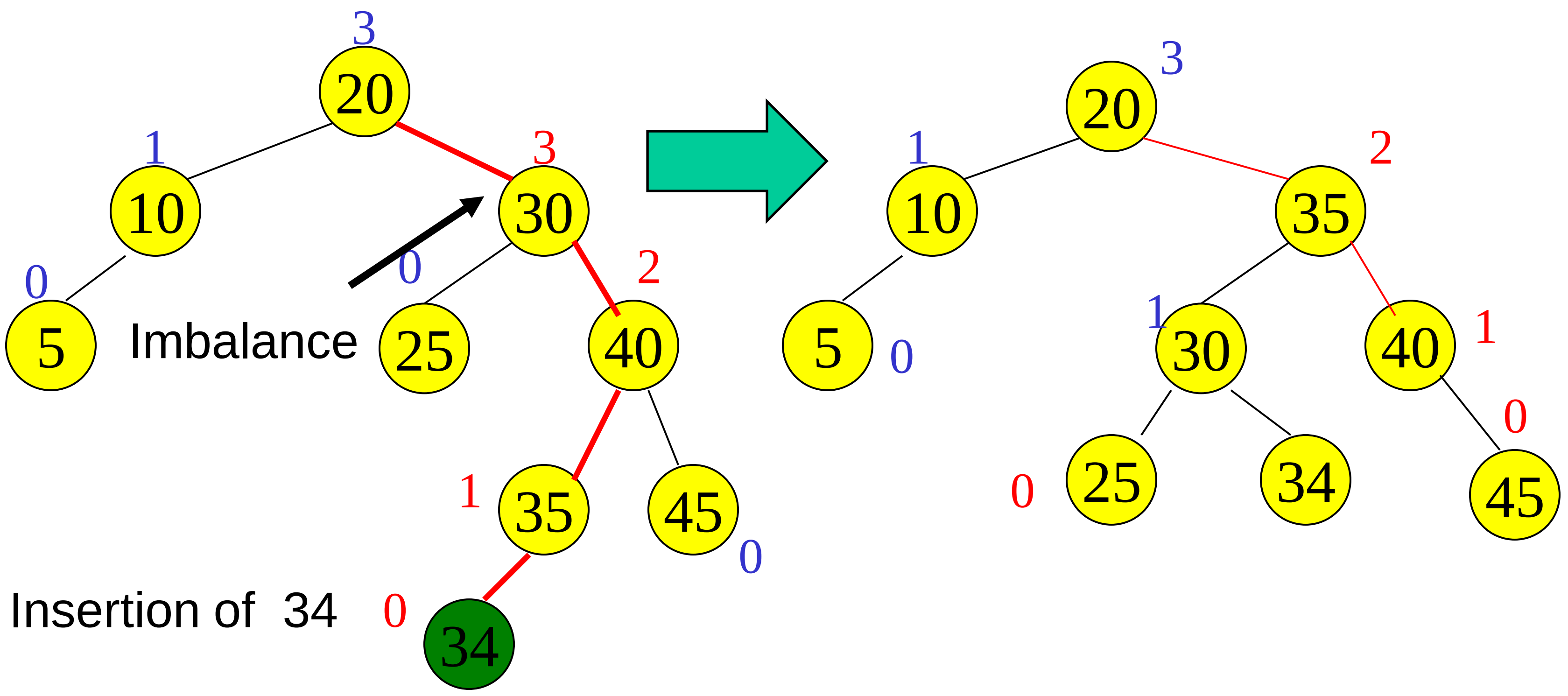
Example of Insertions in an AVL Tree



Single rotation (outside case)



Double rotation (inside case)



BST – Implementation – Insertion

```
ptrnode Insert(ptrnode root, int key){
  if( root == NULL ){
    /* Create and return a one-node tree */
  }
  else if( key < root -> data )
    root->lchild = Insert(root->lchild, key);
  else if( key > node->data)
    root->rchild = Insert(root->rchild,key);
  /* Else key is in the tree already; do nothing */
  return root;  /* Do not forget this line!! */
}
```

```
avlNode * insert(avlNode *T, int y)      //initially pass root and data to be inserted
{
  if (T==NULL)
  {     //get new node T and set its data (to y), lchild (to NULL) and rchild (to NULL) fields;
         height         (height  will be set later  )  }
  else
    if (T->data< y)           // insert in right subtree of T
    {
      //recursively call  insert function for rchild of T
     //check balance factor and if BF(T) = -2 then if (y > T->rchild->data)  perform
          RR rotation  otherwise perform RL rotation
    }
    else
      if (T->data>y)         // insert in left subtree of T
      {
        //recursively call  insert function for lchild of T
          //check balance factor and if BF(T) = 2 then if (y < T->lchild->data) perform
          LL  rotation otherwise perform LR rotation
      }
     //set height of the subtree at node T
      return(T);
}
```

```
avlNode * insert(avlNode *T, int y)      //initially pass root and data to be inserted
{
    if (T==NULL)
    {
        //get new node T and set its data (to y), lchild (to NULL) and rchild (to NULL) fields;
          height (height) will be set later
    }
    else if ( T->data<y)             // insert in right subtree
      {
        T->rchild = insert(T->rchild, y);     //recursively call  insert function for rchild of T
        if (BF(T) == -2)                      //check balance factor and do rotations
          if (y > T->rchild->data)
            T=RR(T);
          else
            T=RL(T);
      }
      else                                  // insert in left subtree
        {      //recursively call  insert function for lchild of T in a similar way
               // check the balance factor (as 2) and call LL(T) and LR(T) as required
        }
     T->height=height(T);  //set height of T
     return(T);
}
```

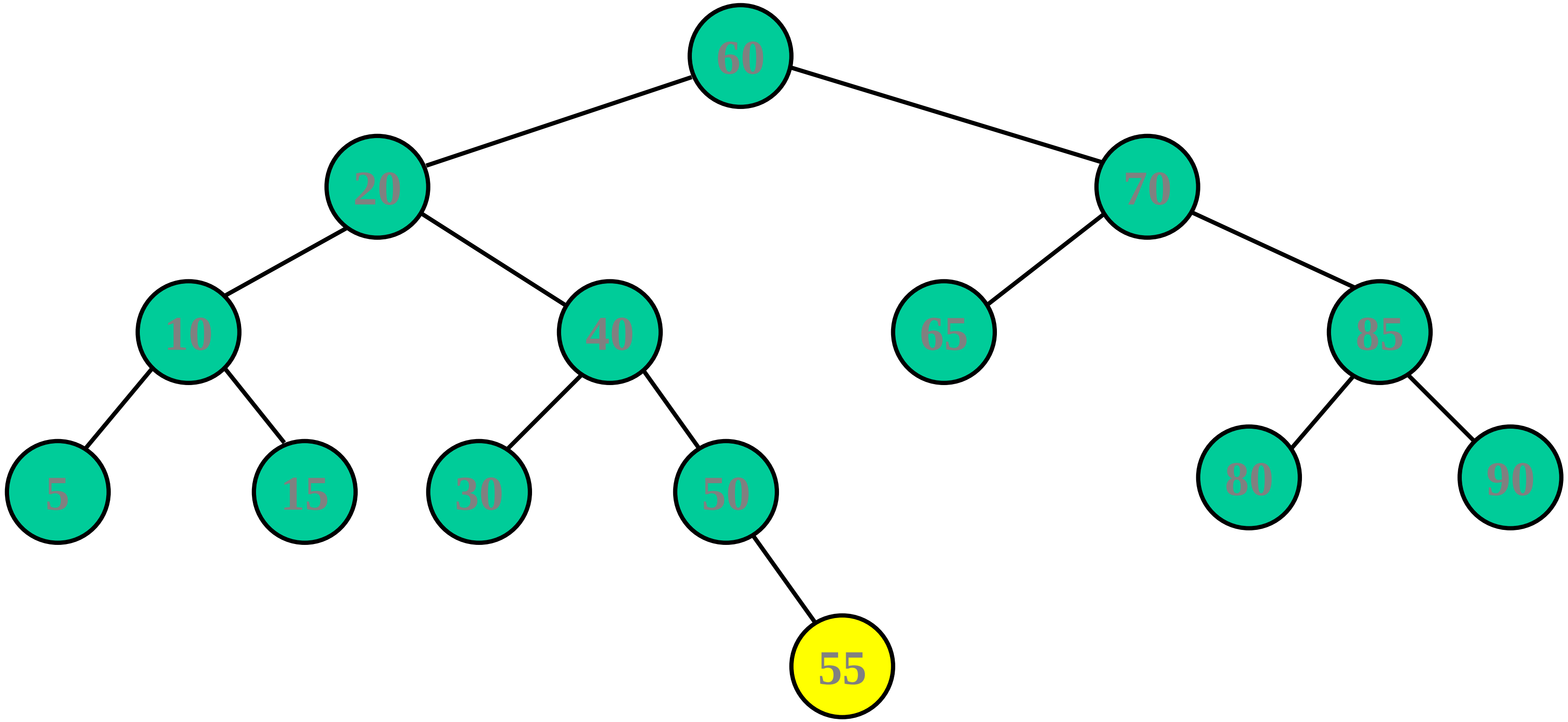
AVL Tree Deletion

- Similar but more complex than insertion
  - Rotations and double rotations needed to rebalance
  - Imbalance may propagate upward so that many rotations may be needed.

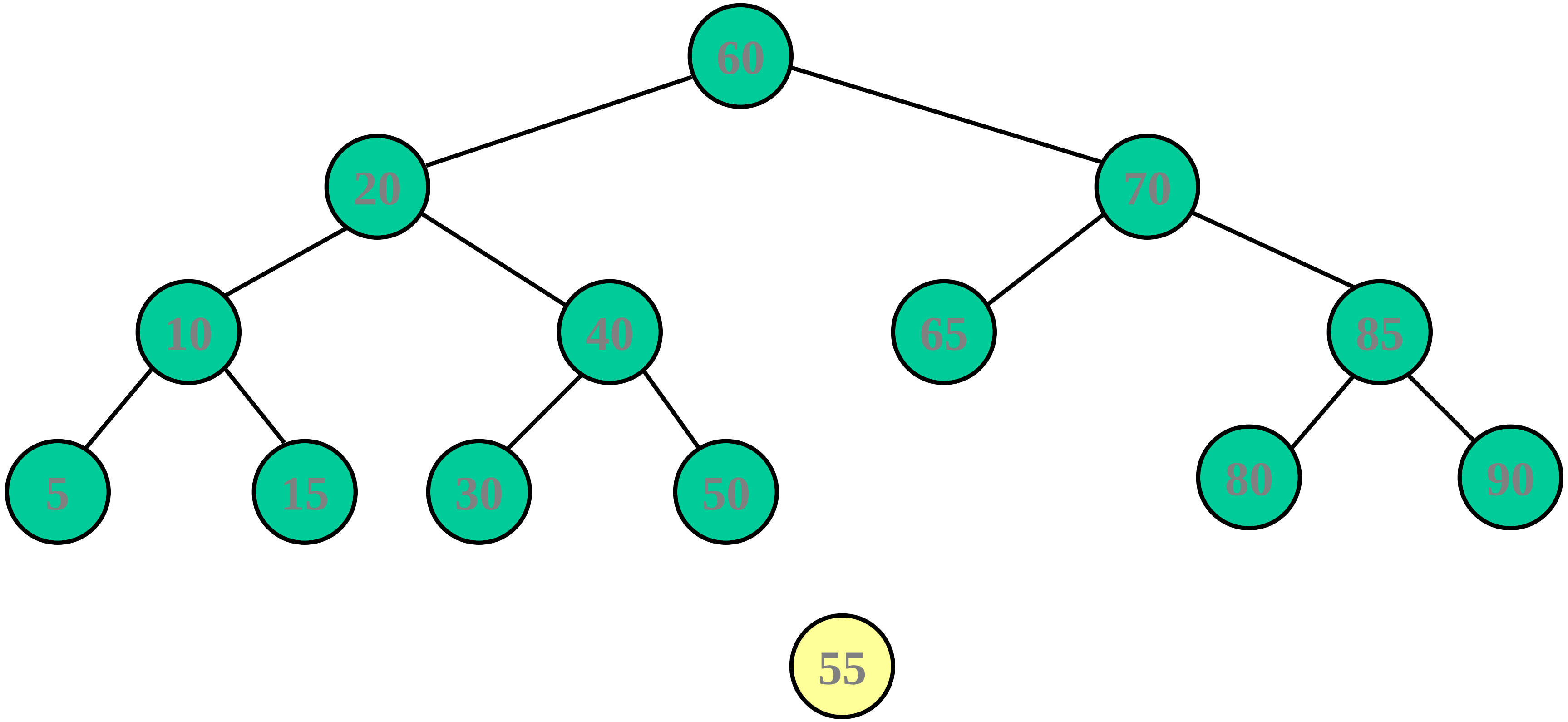
Deletion X in AVL Trees

- Deletion:
  - Case 1: if X is a leaf, delete X
  - Case 2: if X has 1 child, use it to replace X
  - Case 3: if X has 2 children, replace X with its inorder predecessor (and recursively delete it)
- Rebalancing

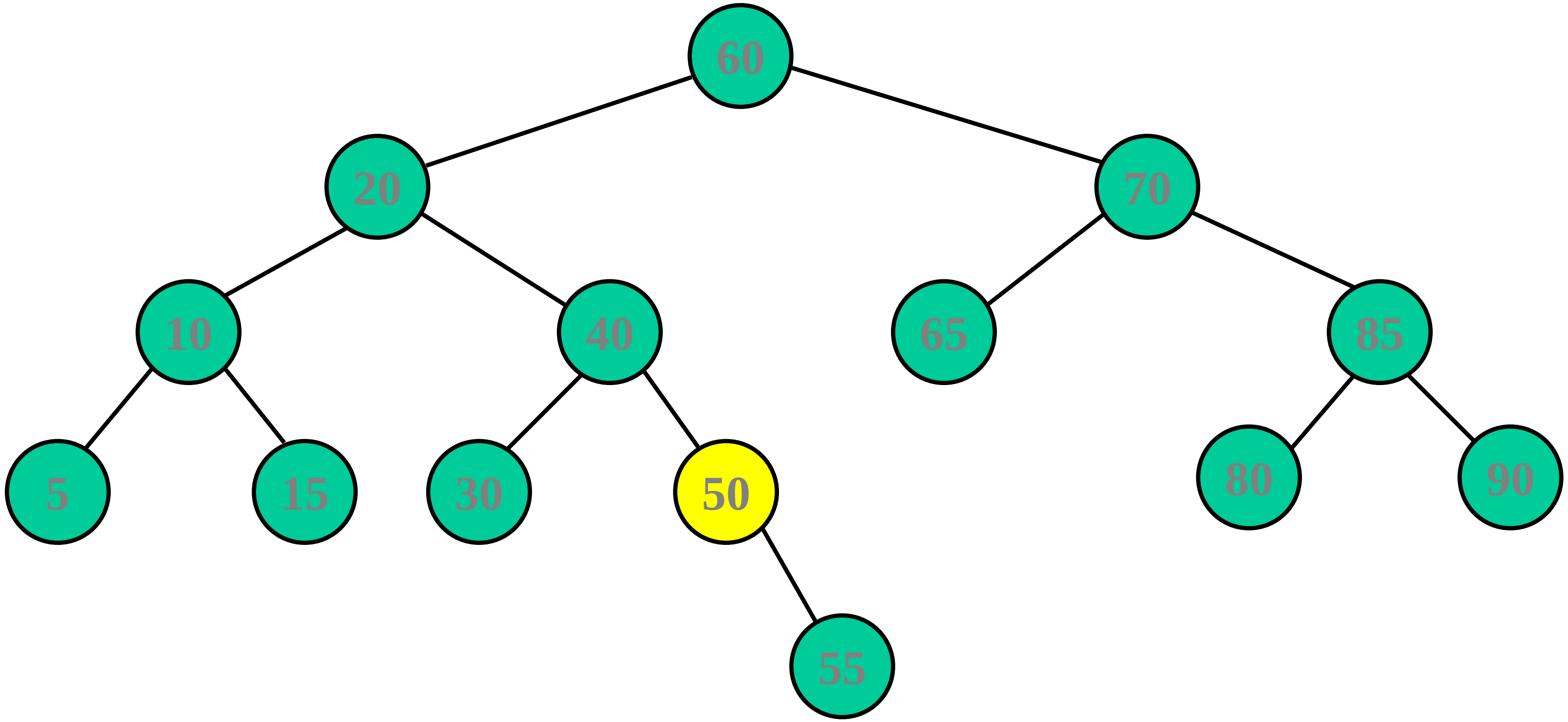
Delete 55 (case 1)



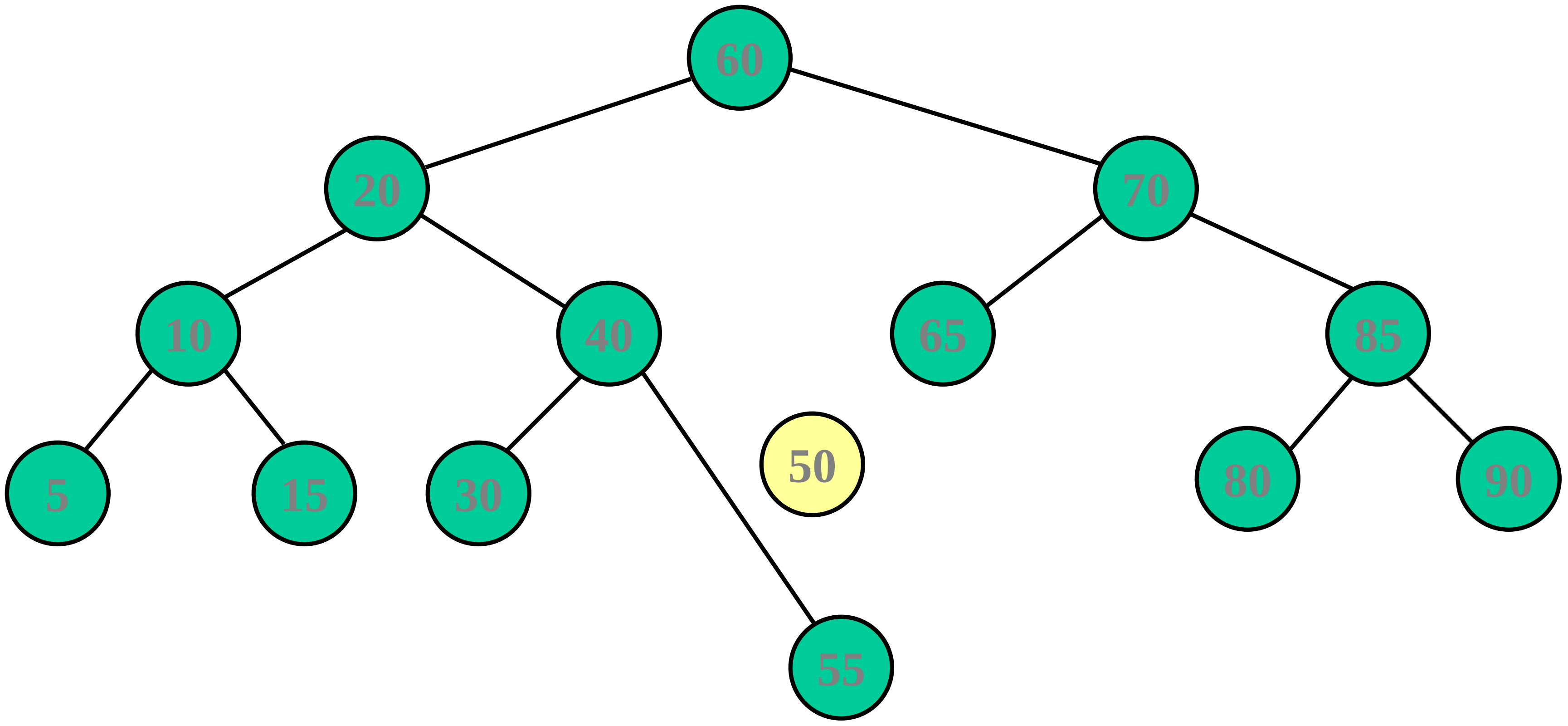
Delete 55 (case 1)



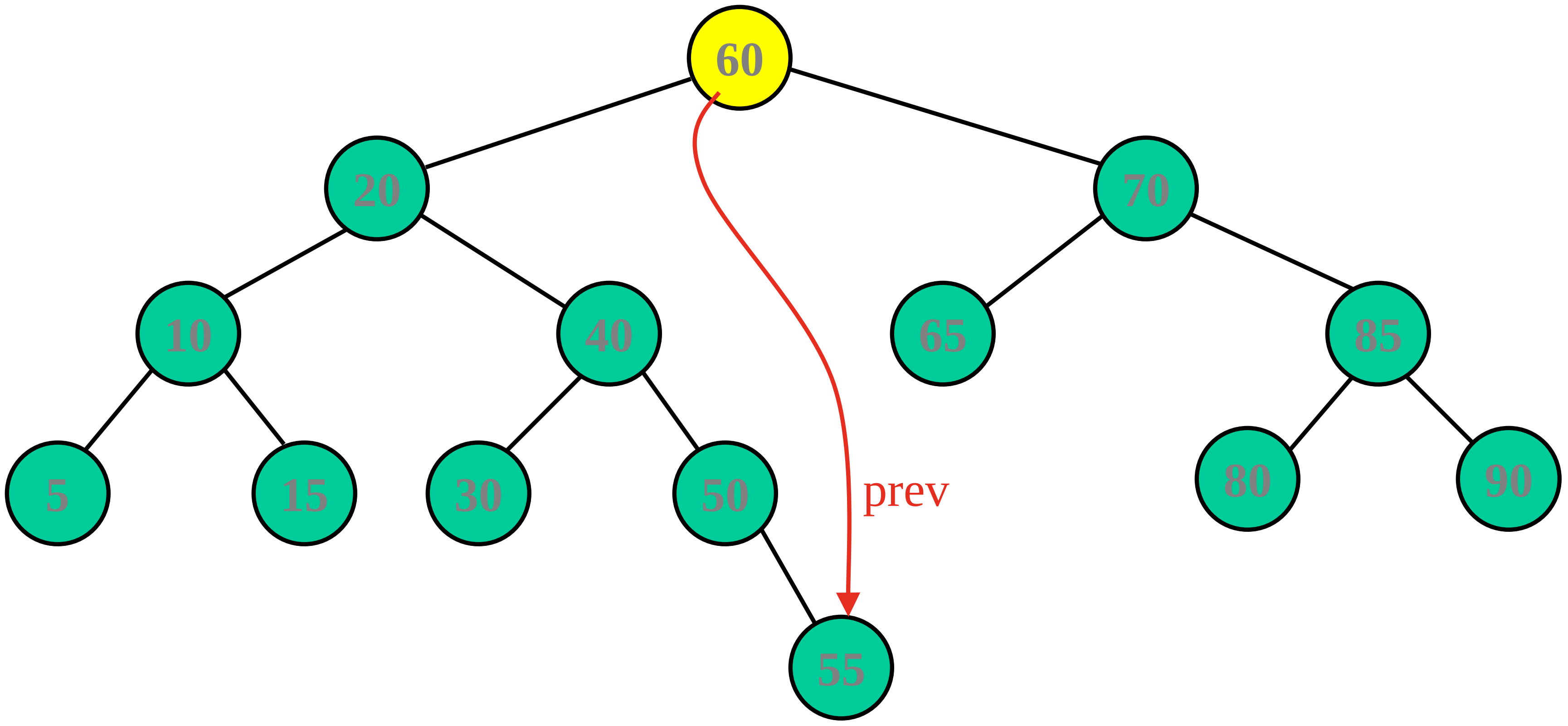
Delete 50 (case 2)



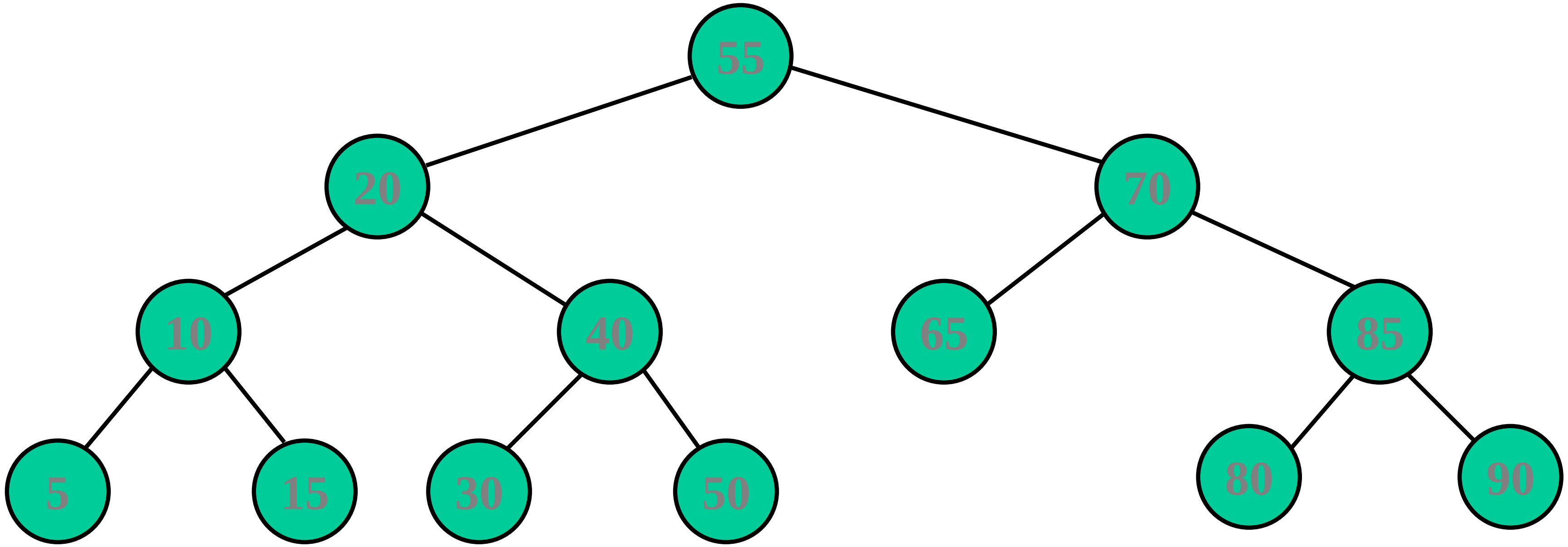
Delete 50 (case 2)



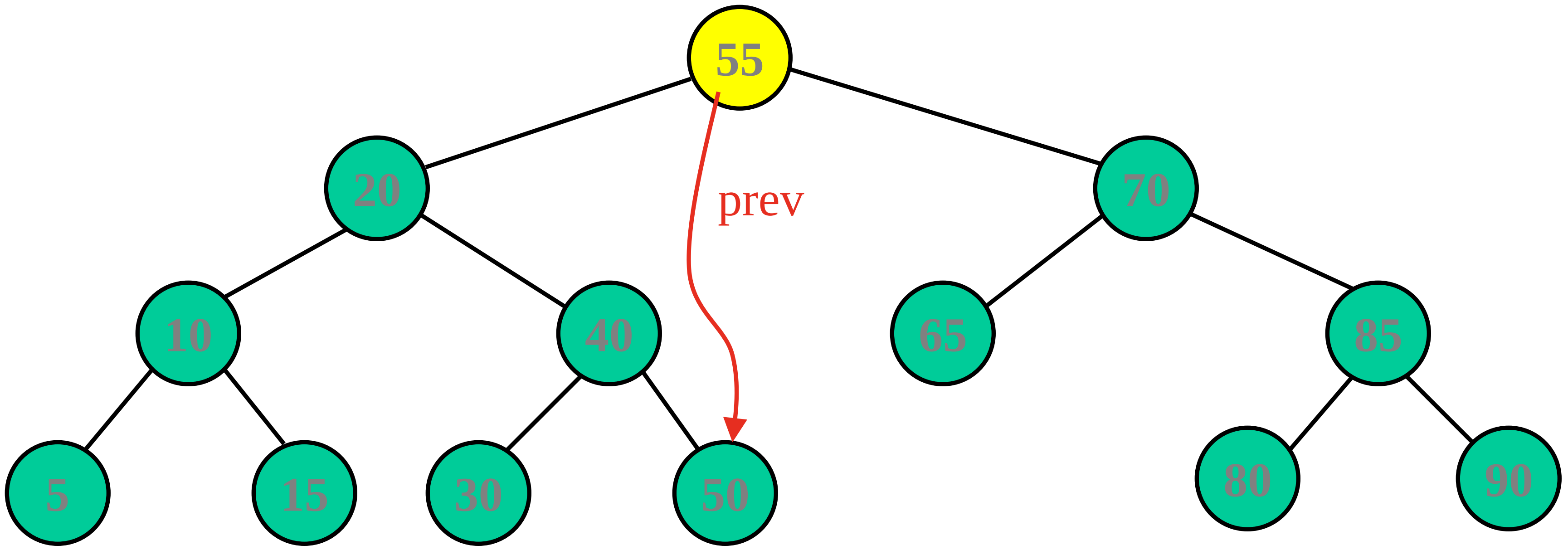
Delete 60 (case 3)



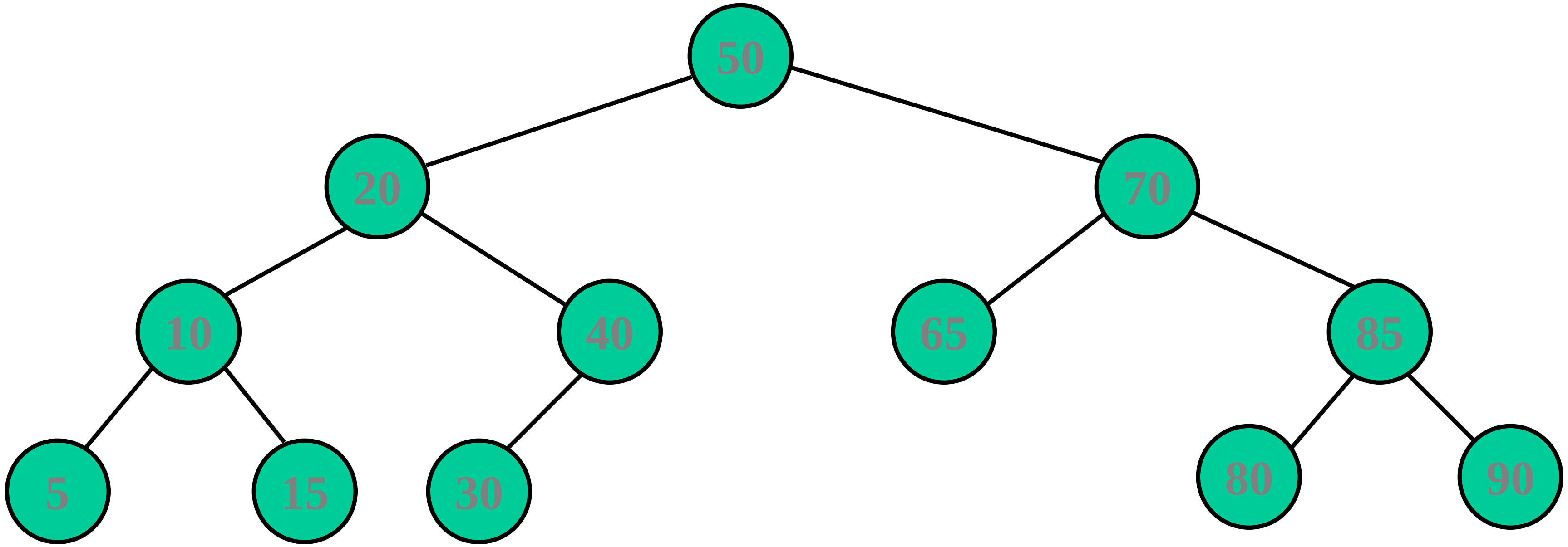
Delete 60 (case 3)



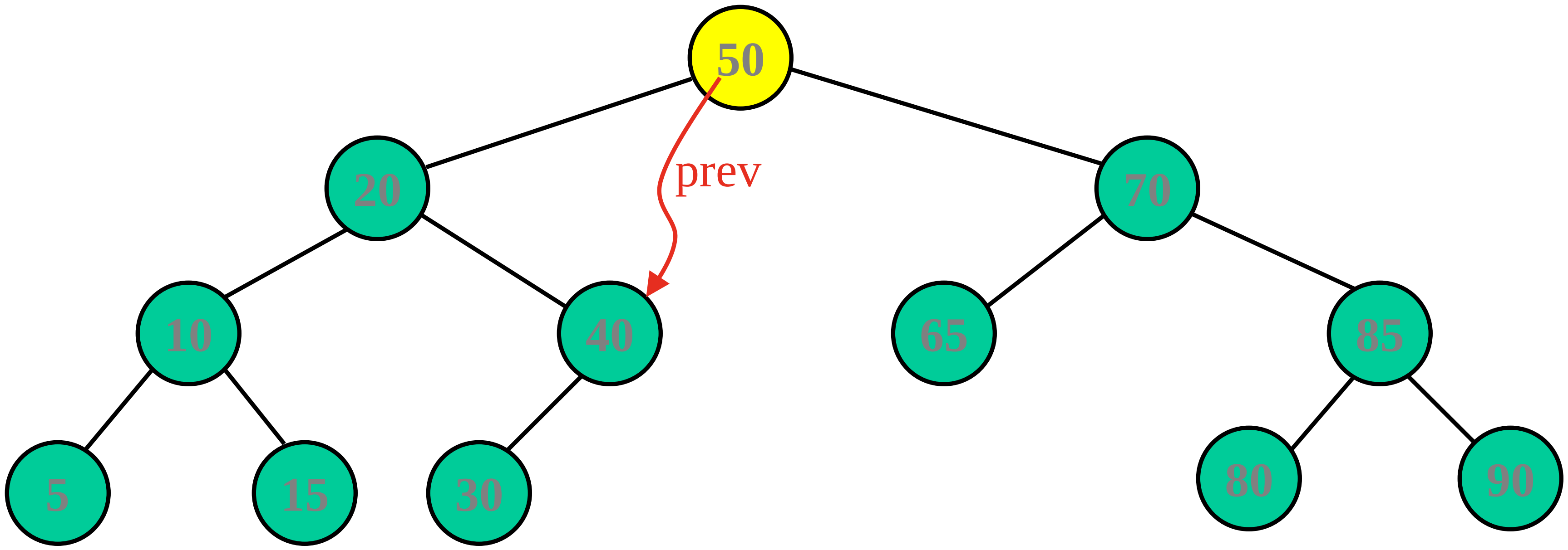
Delete 55 (case 3)



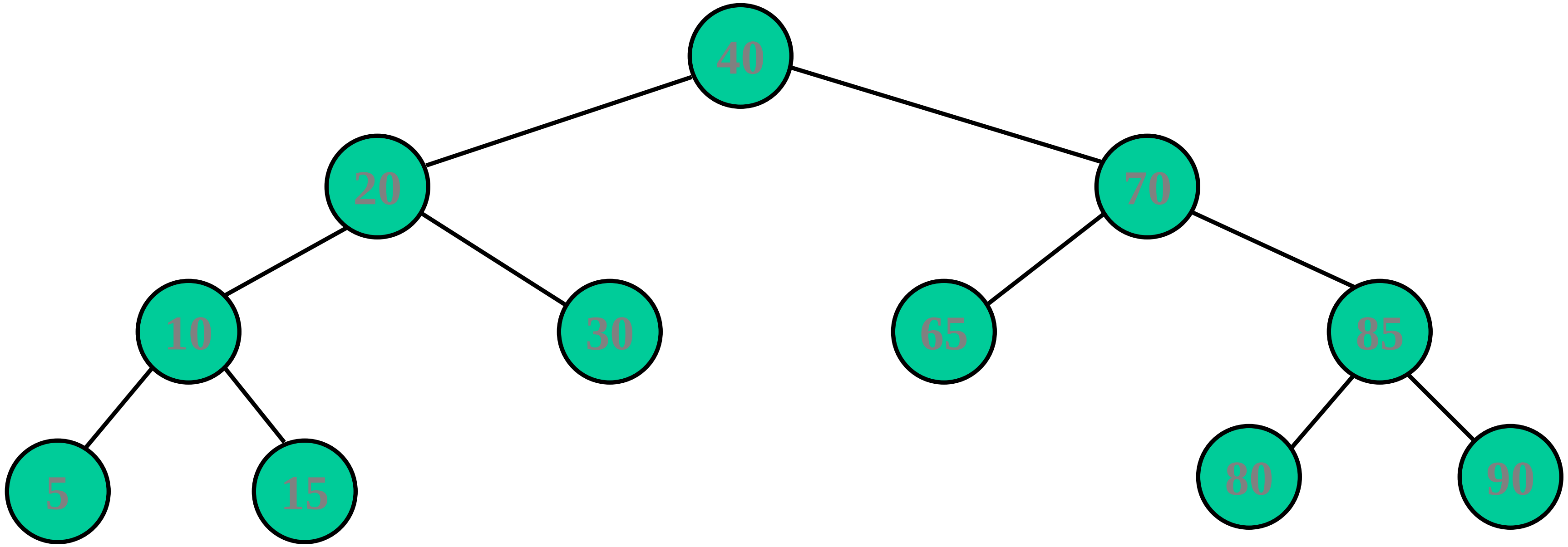
Delete 55 (case 3)



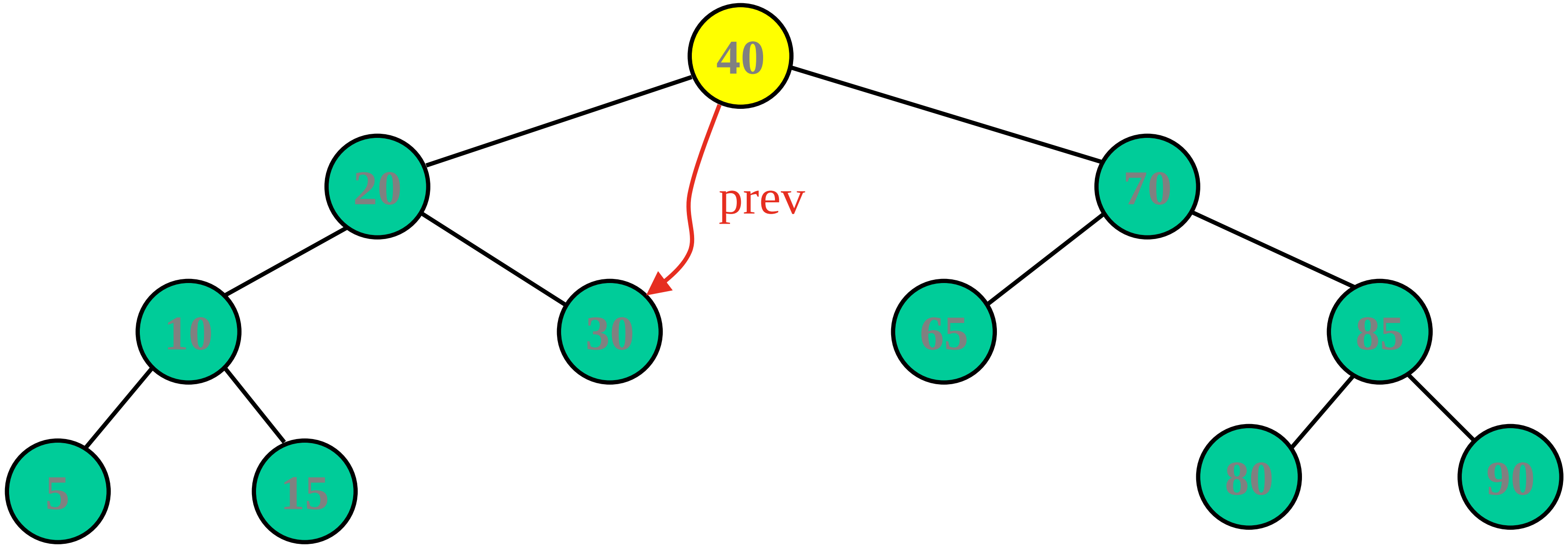
Delete 50 (case 3)



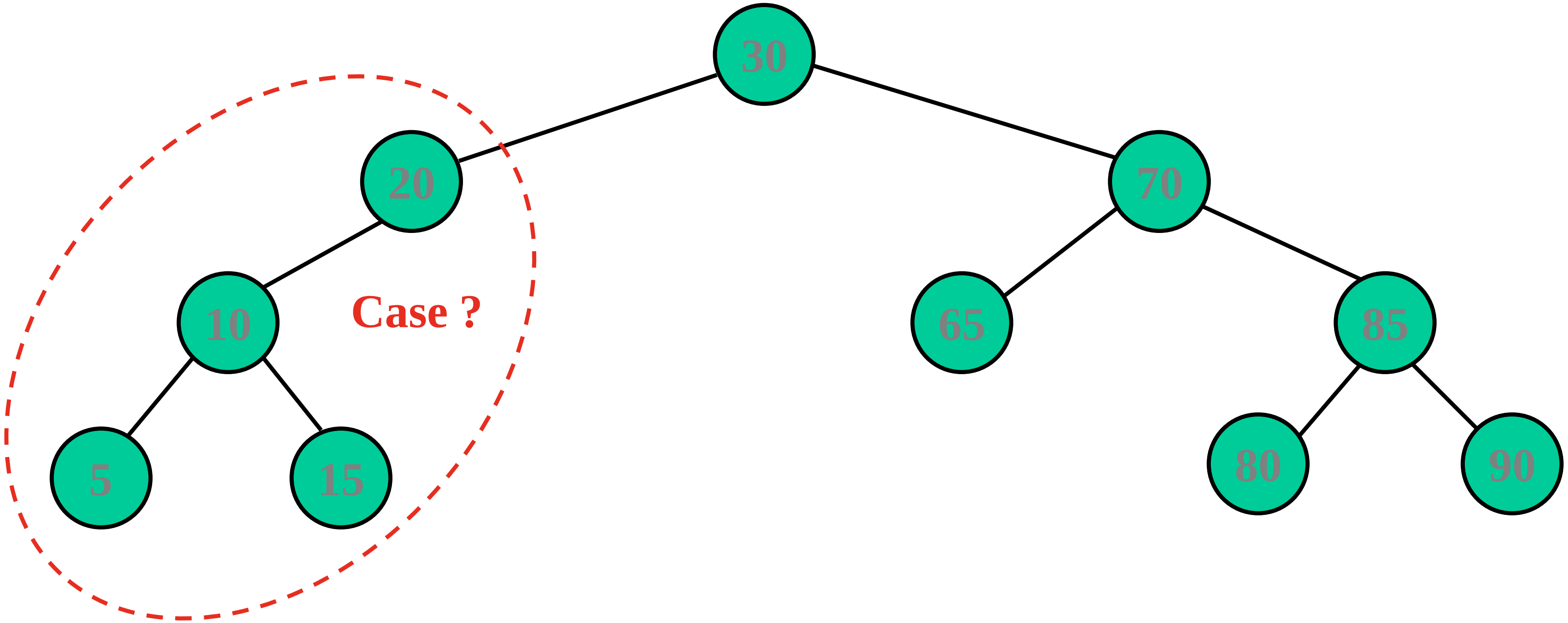
Delete 50 (case 3)



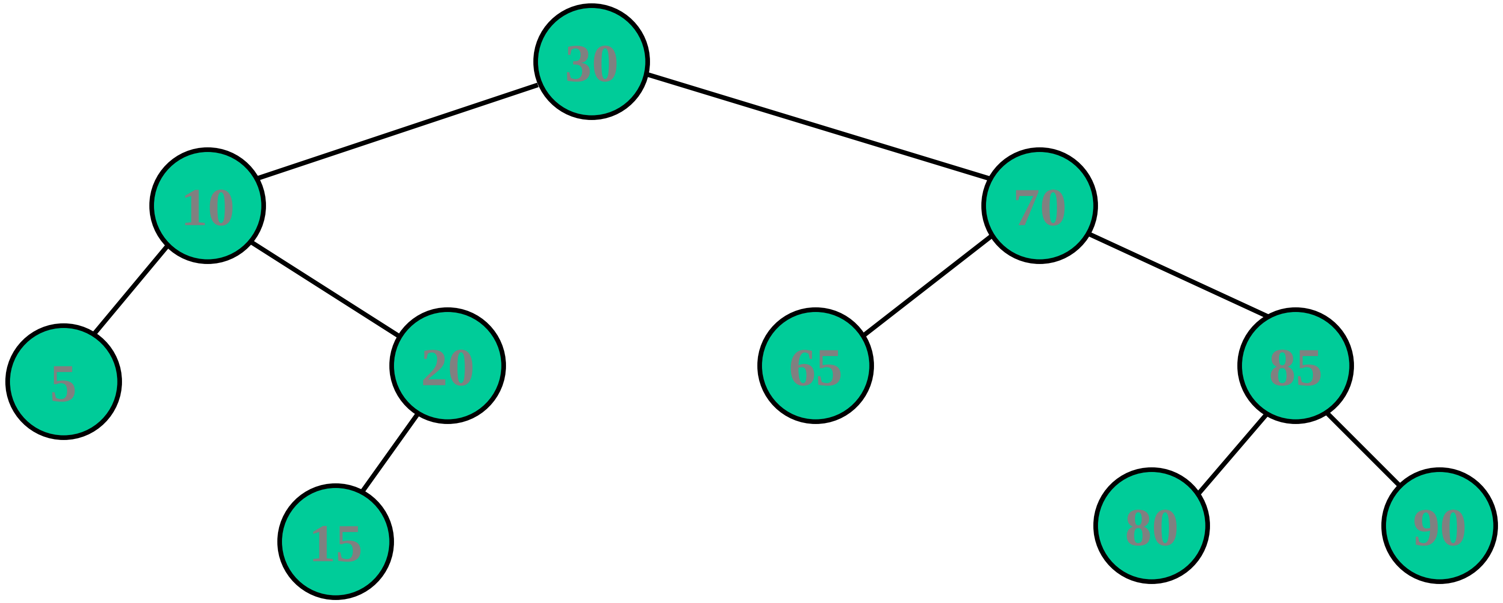
Delete 40 (case 3)



Delete 40 : Rebalancing



Delete 40: after rebalancing



Single rotation is preferred!

AVL Tree: analysis

- The depth of AVL Trees is at most logarithmic.
- So, all of the operations on AVL trees are also logarithmic.
- The worst-case height is at most 44 percent more than the minimum possible for binary trees.

Pros and Cons of AVL Trees

- Arguments for AVL trees:
  1. Search is O(lg N) since AVL trees are always balanced.
  2. Insertion and deletions are also O(lg N)
  3. The height balancing adds no more than a constant factor to the speed of insertion.
- Arguments against using AVL trees:
  1. Difficult to program & debug; more space for balance factor.
  2. Asymptotically faster but rebalancing costs time.
  3. Most large searches are done in database systems on disk and use other structures (e.g. B-trees).
  4. May be OK to have O(N) for a single operation if total run time for many consecutive operations is fast (e.g. Splay trees).

Summary

- Find element, insert element, and remove element operations all have complexity O(lg N) for worst case
- Insert operation: top-down insertion and bottom up balancing