COMP251: DATA STRUCTURES & ALGORITHMS

Instructor: Maryam Siahbani

Computer Information System
University of Fraser Valley

* Some slides from “Algorithms and Data Structures” by Douglas Wilhelm Harder

AVL Trees

Outline

- Background
- Define height balancing
- Maintaining balance within a tree
  - AVL trees
  - Difference of heights
  - Maintaining balance after insertions and erases

Background

From previous lectures:

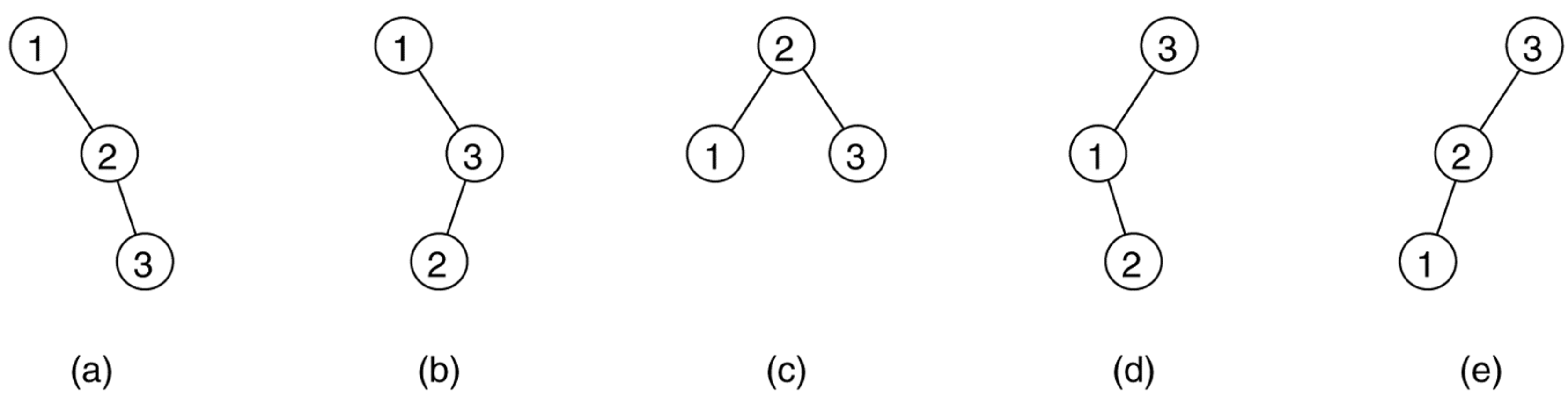
- Binary search trees store linearly ordered data
- Best case height: $\Theta(\ln(n))$
- Worst case height: $\mathbf{O}(n)$

Requirement:

- Define and maintain a ***balance*** to ensure $\Theta(\ln(n))$ height

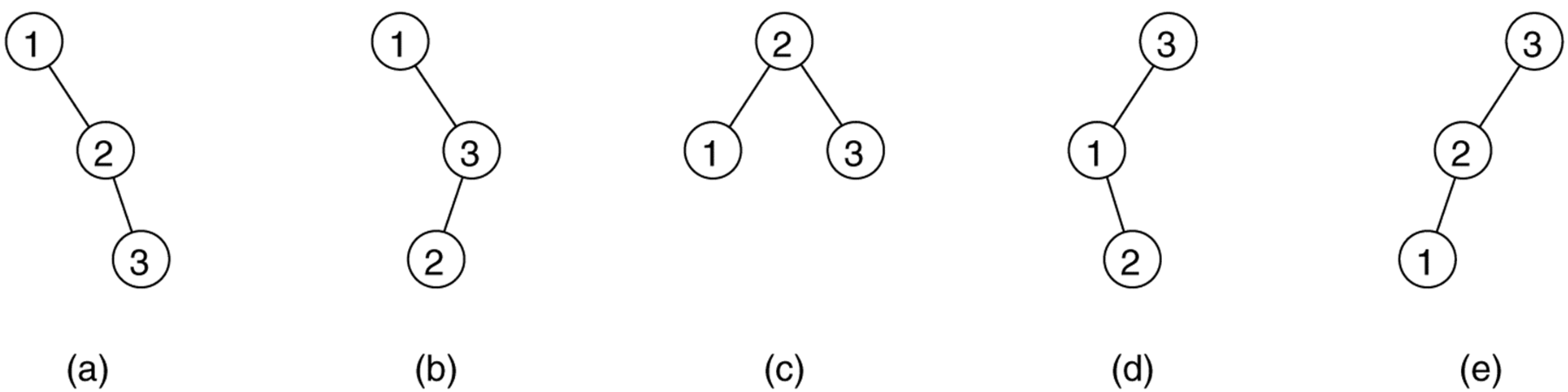
Prototypical Examples

- Binary search trees that can result from inserting a permutation 1, 2, and 3:
  - tree in part (c) is twice as likely as any other trees (2, 1, 3 and 2,3, 1)



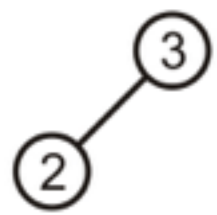
Prototypical Examples

- Trees (a) and (e) are similar (right-right and left-left imbalance)
- Trees (b)and (d) are similar (right-left and left-right imbalance)
- We consider two general cases two correct the imbalance



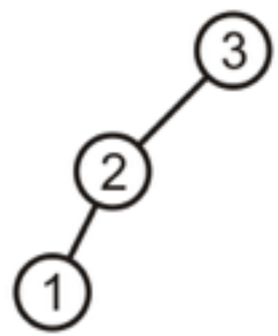
Prototypical Examples

Starting with this tree, add 1:



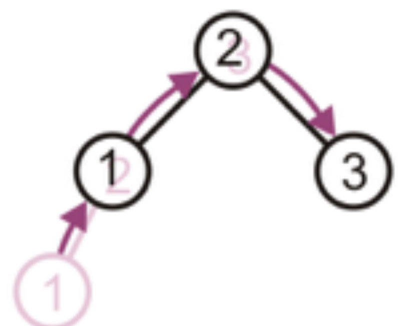
Prototypical Examples

This is more like a linked list; however, we can fix this…



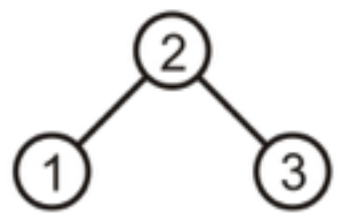
Prototypical Examples

Promote 2 to the root, demote 3 to be 2's right child, and 1 remains the left child of 2



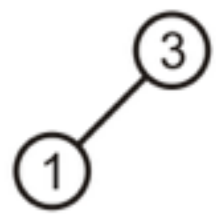
Prototypical Examples

The result is a perfect, though trivial tree



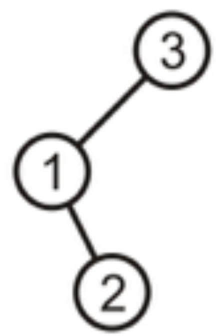
Prototypical Examples

Alternatively, given this tree, insert 2



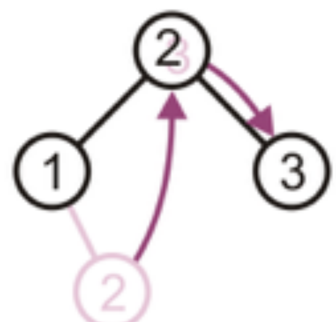
Prototypical Examples

Again, the product is a linked list; however, we can fix this, too



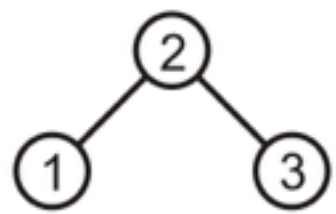
Prototypical Examples

Promote 2 to the root, and assign 1 and 3 to be its children



Prototypical Examples

The result is, again, a perfect tree



These examples may seem trivial, but they are the basis for the corrections in the next data structure we will see: AVL trees

AVL Trees

Balance on binary search trees is defined by comparing the height of the two sub-trees

Recall:

- An empty tree has height -1
- A tree with a single node has height 0

AVL Trees

A binary search tree is said to be AVL balanced if:

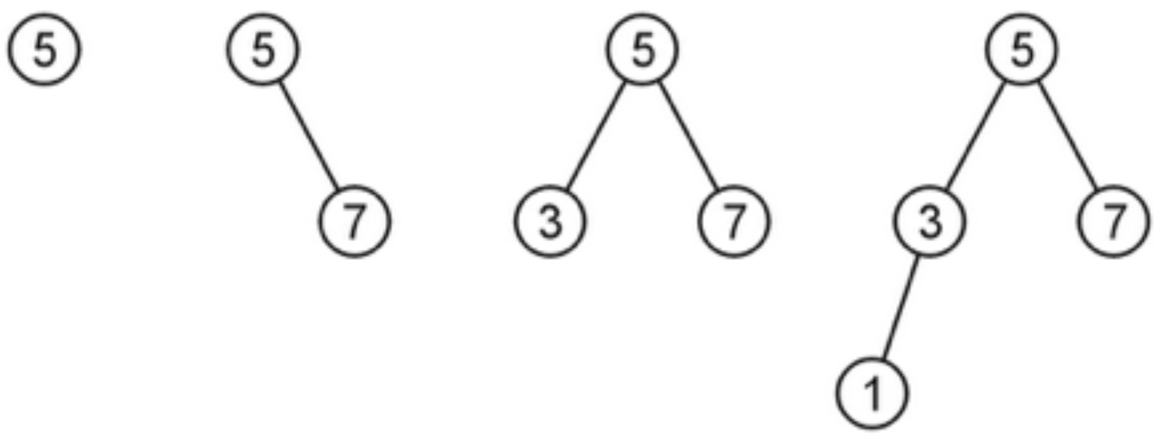
- The difference in the heights between the left and right sub-trees is at most 1, and
- Both sub-trees are themselves AVL trees

AVL trees

- Named after Adelson-Velskii and Landis

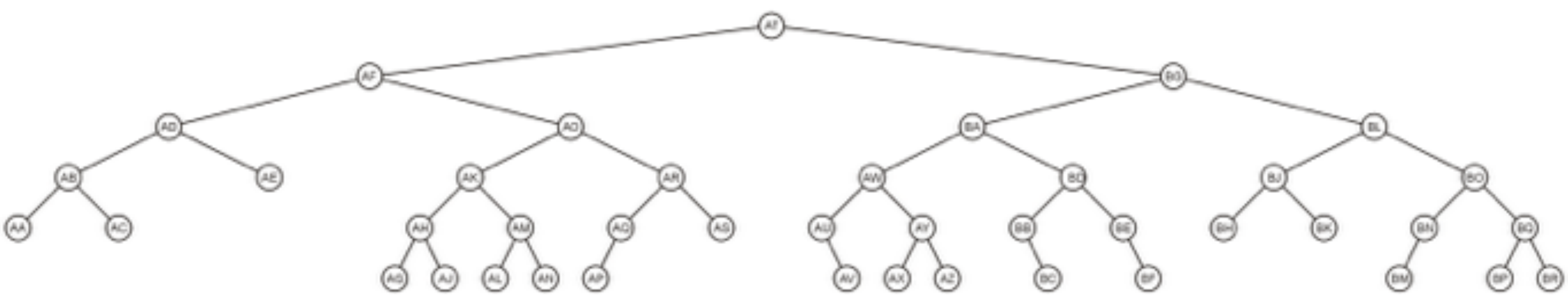
AVL Trees

AVL trees with 1, 2, 3, and 4 nodes:



AVL Trees

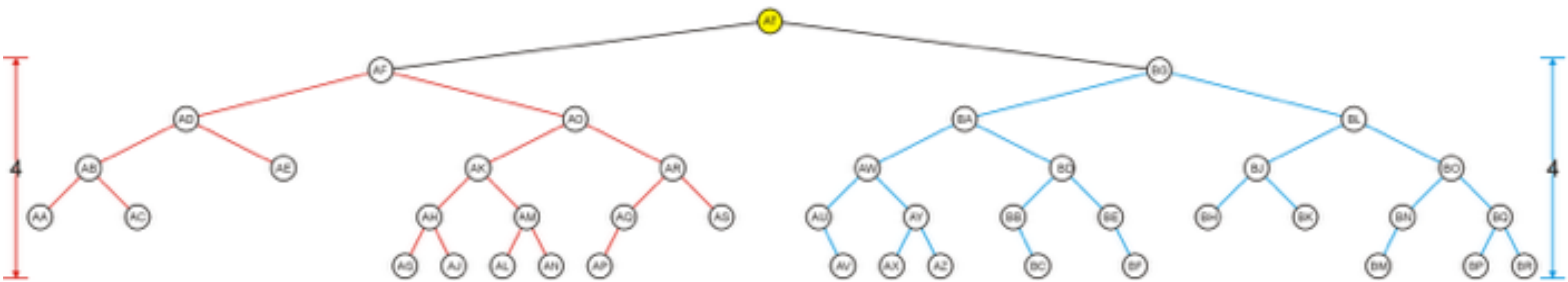
Here is a larger AVL tree (42 nodes):



AVL Trees

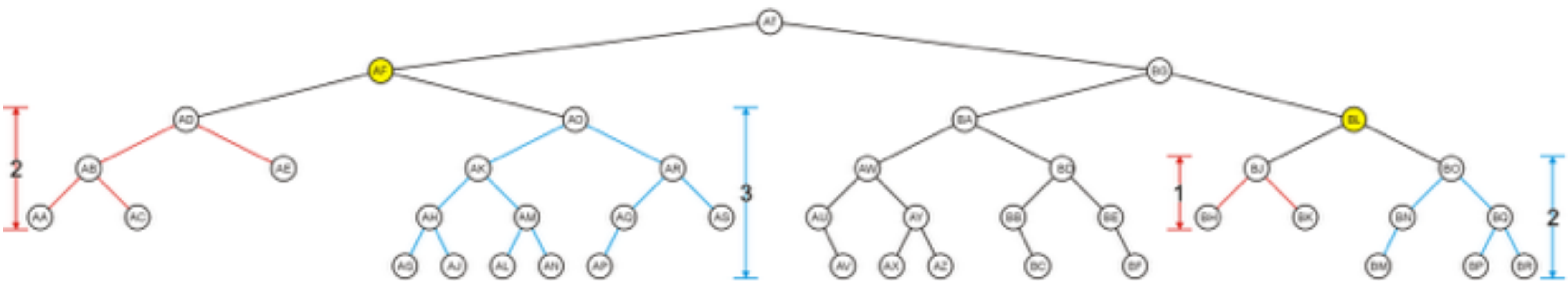
The root node is AVL-balanced:

– Both sub-trees are of height 4:



AVL Trees

- All other nodes (***e.g.***, AF and BL) are AVL balanced
  - The sub-trees differ in height by at most one



Height of an AVL Tree

By the definition of complete trees, any complete binary search tree is an AVL tree

Thus an upper bound on the number of nodes in an AVL tree of height h is a perfect binary tree with $2^{h+1} - 1$ nodes

- What is a lower bound?

Height of an AVL Tree

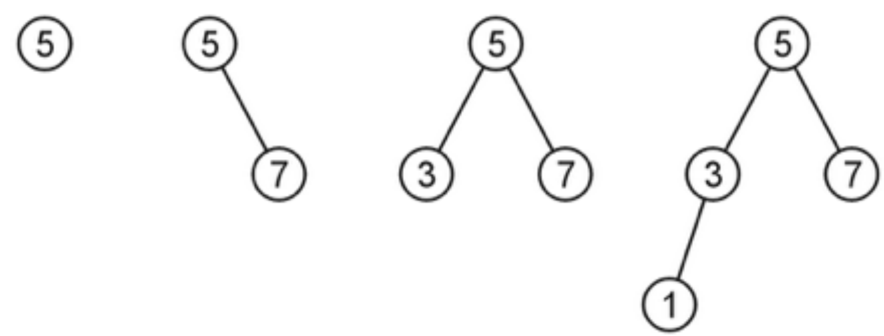
Let $F(h)$ be the fewest number of nodes in a tree of height h

From a previous slide:

$F(0) = 1$

$F(1) = 2$

$F(2) = 4$



Can we find $F(h)$?

Height of an AVL Tree

The worst-case AVL tree of height h would have:

- A worst-case AVL tree of height $h - 1$ on one side,
- A worst-case AVL tree of height $h - 2$ on the other, and
- The root node

We get: $F(h) = F(h-1) + 1 + F(h-2)$

Height of an AVL Tree

This is a recursion relation:

$$\mathrm{F}(h) = \begin{cases} 1 & h = 0 \\ 2 & h = 1 \\ \mathrm{F}(h-1) + \mathrm{F}(h-2) + 1 & h > 1 \end{cases}$$

The solution?

- Note that $F(h) + 1 = (F(h-1) + 1) + (F(h-2) + 1)$
- Therefore, $F(h) + 1$ is a Fibonacci number:

$$\begin{array}{lcl} F(0) + 1 = 2 & \to & F(0) = 1 \\ F(1) + 1 = 3 & \to & F(1) = 2 \\ F(2) + 1 = 5 & \to & F(2) = 4 \\ F(3) + 1 = 8 & \to & F(3) = 7 \\ F(4) + 1 = 13 & \to & F(4) = 12 \\ F(5) + 1 = 21 & \to & F(5) = 20 \\ F(6) + 1 = 34 & \to & F(6) = 33 \end{array}$$

Height of an AVL Tree

Theorem:

Worst case height for an AVL Tree with n nodes is $\Theta(\ln(n))$

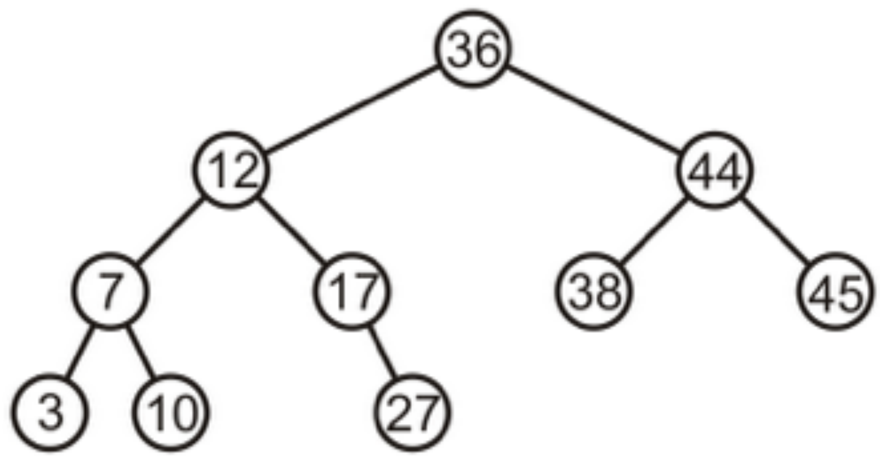
Maintaining Balance

To maintain AVL balance, observe that:

- Inserting a node can increase the height of a tree by at most 1
- Removing a node can decrease the height of a tree by at most 1

Maintaining Balance

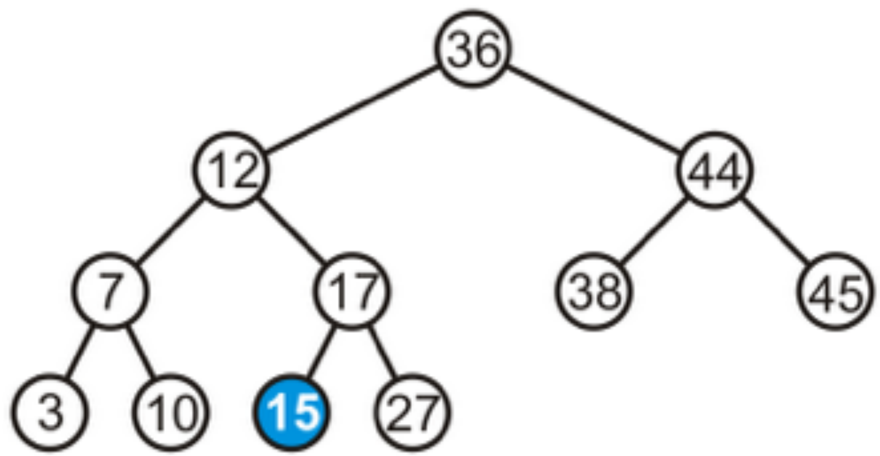
Consider this AVL tree



Maintaining Balance

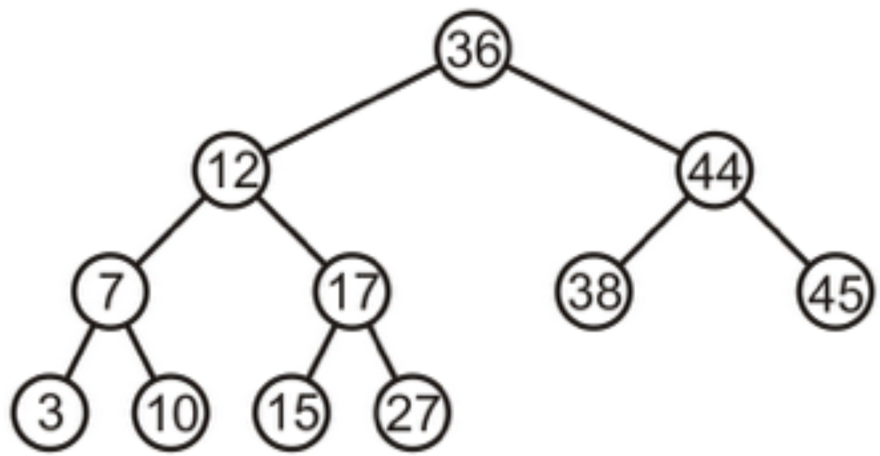
Consider inserting 15 into this tree

- In this case, the heights of none of the trees change



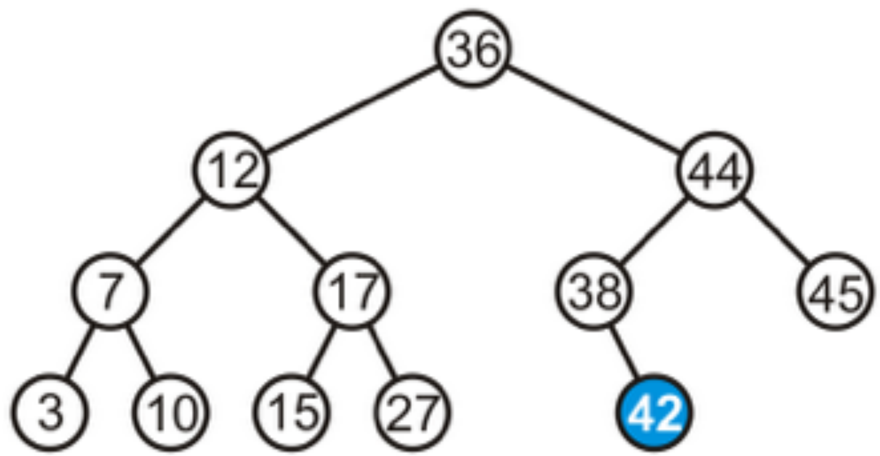
Maintaining Balance

The tree remains balanced



Maintaining Balance

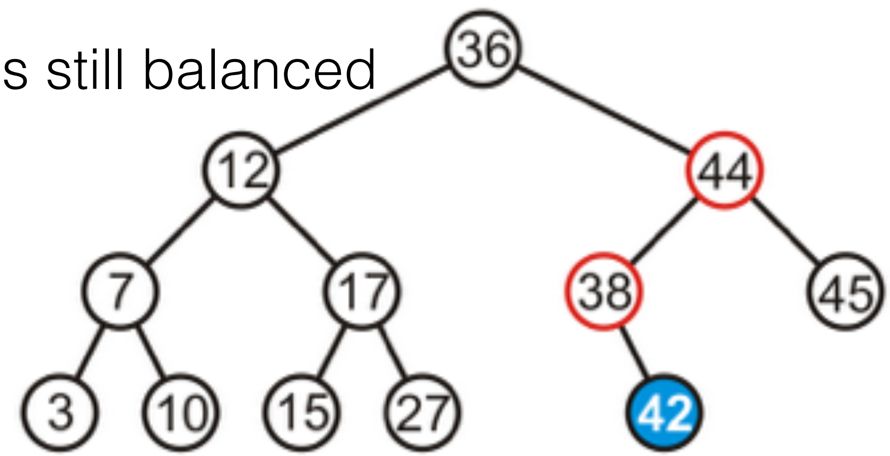
Consider inserting 42 into this tree



Maintaining Balance

Consider inserting 42 into this tree

- Now we see the heights of two sub-trees have increased by one
- The tree is still balanced



Maintaining Balance

To calculate changes in height, the method must run in $\Theta(1)$ time

- Our implementation of height is $\Theta(n)$:

```
public static <AnyType> int height(BinaryNode<AnyType> t){
    if( t == null ) return -1;
    return 1 + Math.max(height(t.left),height(t.right));
}
```

Maintaining Balance

Introduce a field into the node class

`int height;`

The height method is now:

```
public static <AnyType> int height(BinaryNode<AnyType> t){
    if( t == null ) return -1;
    return t.height;
}
```

Maintaining Balance

We need to define a new class AVLTree and therefore new class AVLNode

```
class AVLNode {
        // Constructors
    AVLNode(Comparable theElement ) {
        this( theElement, null, null );
    }

    AVLNode(Comparable theElement, AVLNode lt, AVLNode rt ) {
        element  = theElement;
        left     = lt;
        right    = rt;
        height   = 0;
    }
       // Friendly data; accessible by other package routines
    Comparable element;      // The data in the node
    AVLNode    left;         // Left child
    AVLNode    right;        // Right child
    int        height;       // Height
}
```

Maintaining Balance

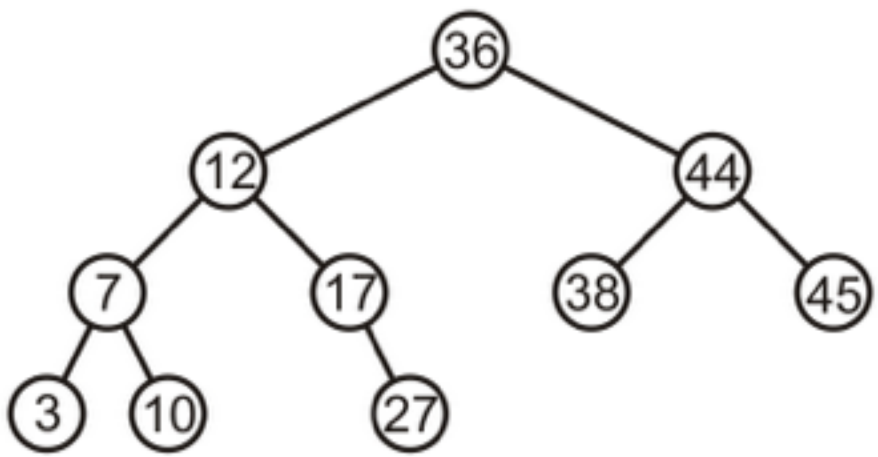
Only insert and erase may change the height

- This is the only place we need to update the height
- These algorithms are already recursive

Maintaining Balance

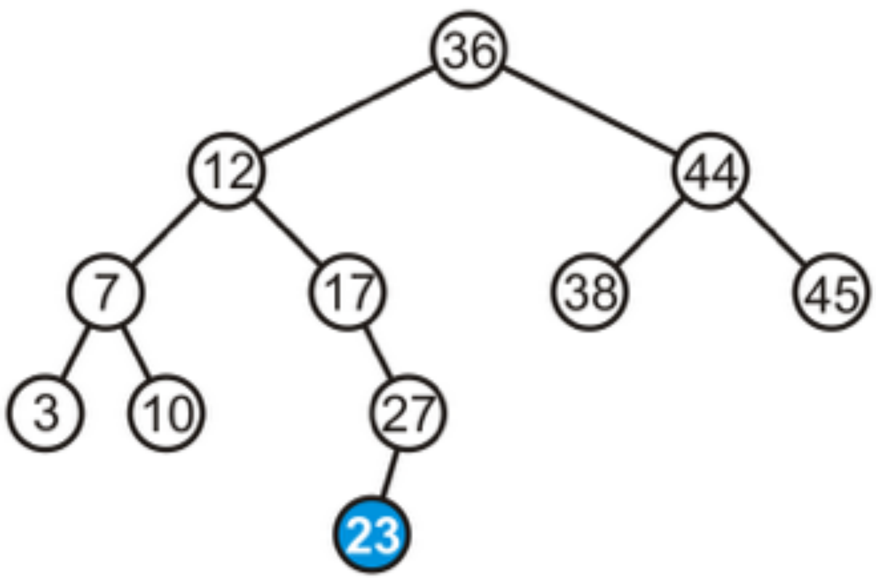
If a tree is AVL balanced, for an insertion to cause an imbalance:

- The heights of the sub-trees must differ by 1
- The insertion must increase the height of the deeper sub-tree by 1



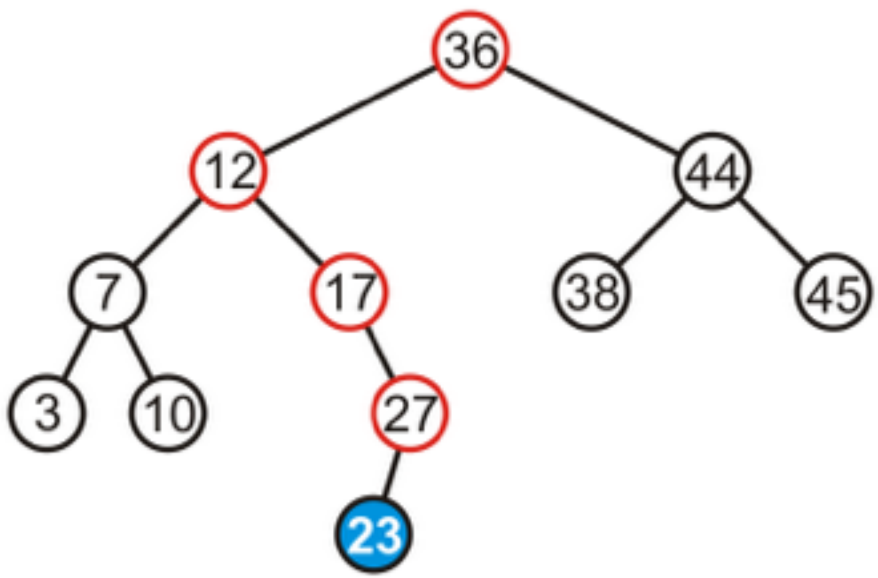
Maintaining Balance

Suppose we insert 23 into our initial tree



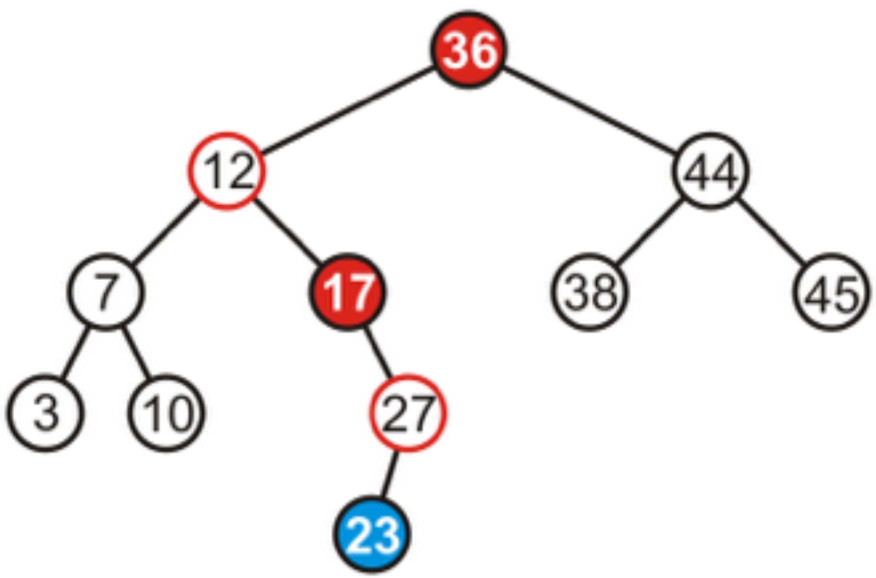
Maintaining Balance

The heights of each of the sub-trees from here to the root are increased by one



Maintaining Balance

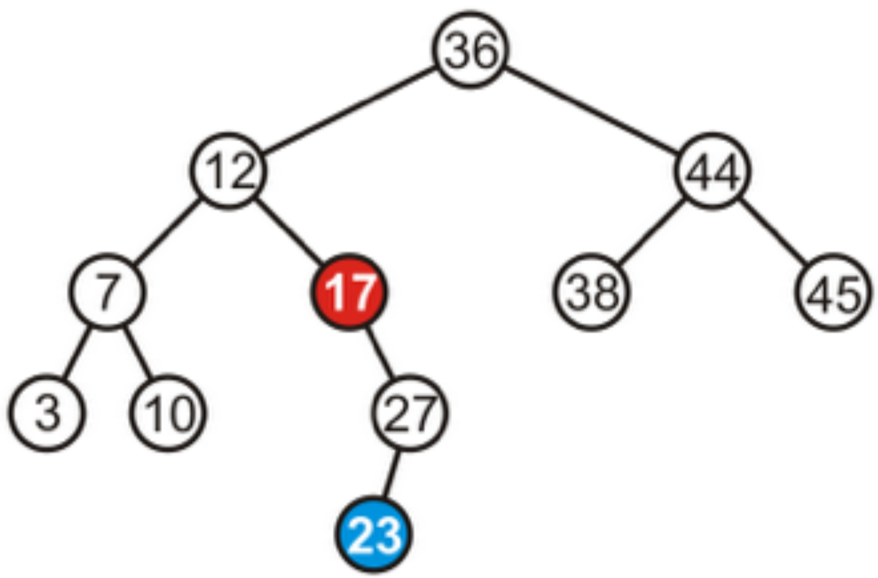
However, only two of the nodes are unbalanced: 17 and 36



Maintaining Balance

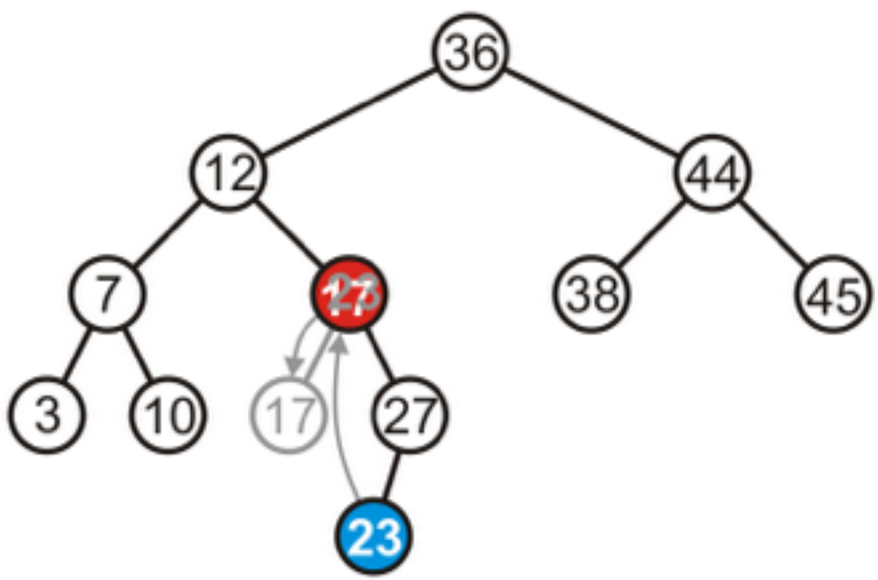
However, only two of the nodes are unbalanced: 17 and 36

- We only have to fix the imbalance at the lowest node



Maintaining Balance

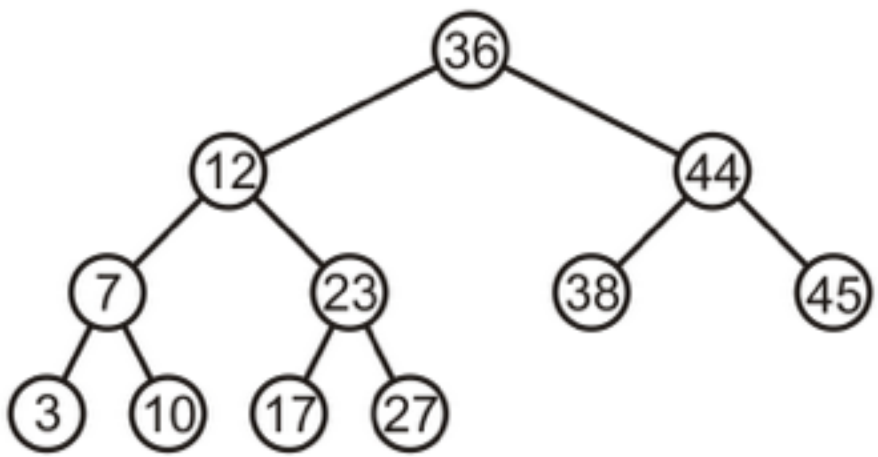
We can promote 23 to where 17 is, and make 17 the left child of 23



Maintaining Balance

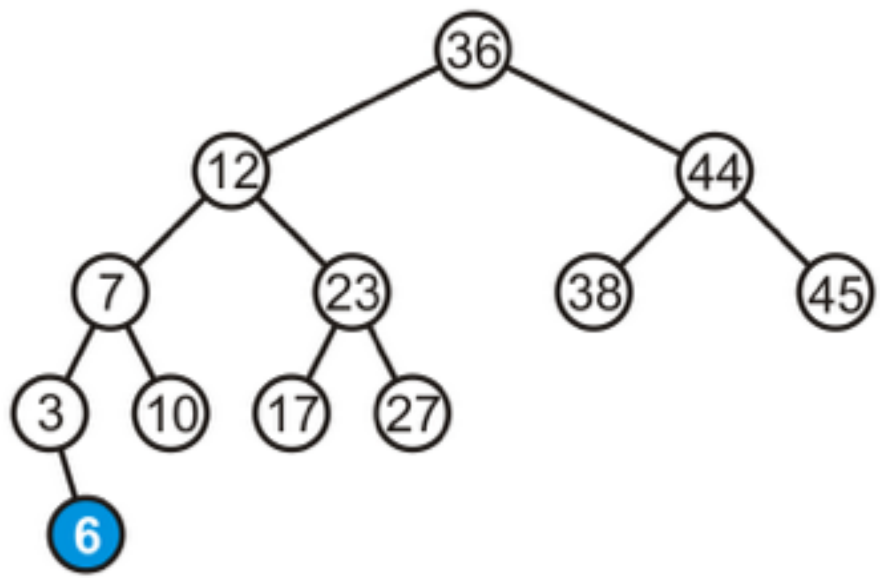
Thus, that node is no longer unbalanced

- Incidentally, the root now is balanced again



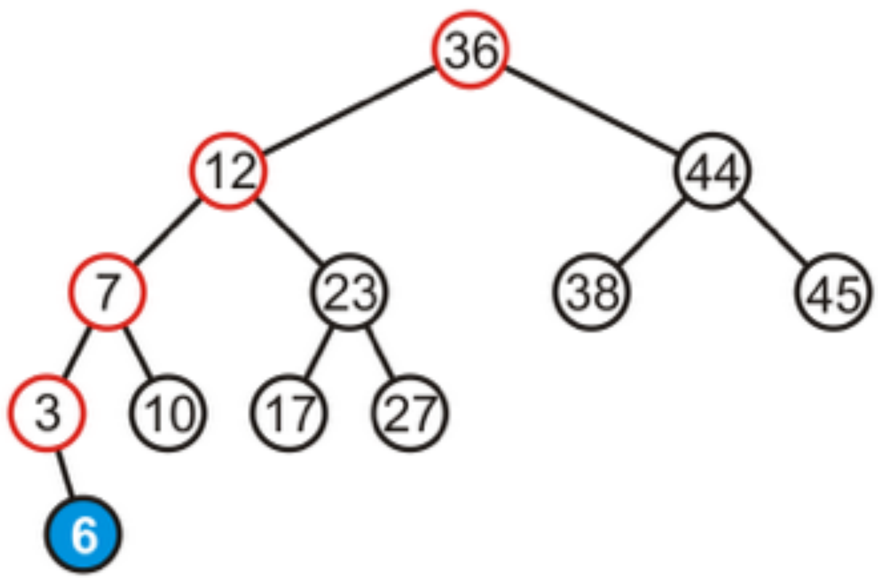
Maintaining Balance

Consider adding 6:



Maintaining Balance

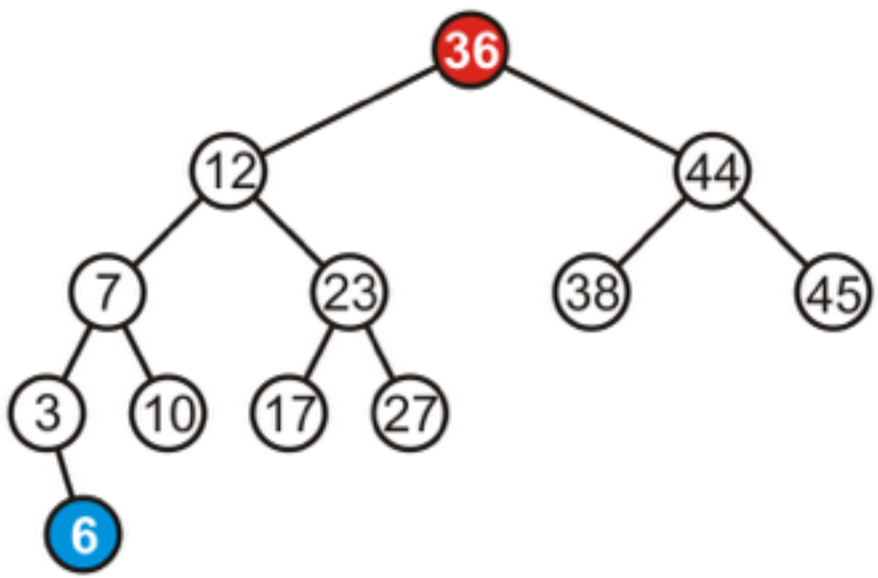
The height of each of the trees in the path back to the root are increased by one



Maintaining Balance

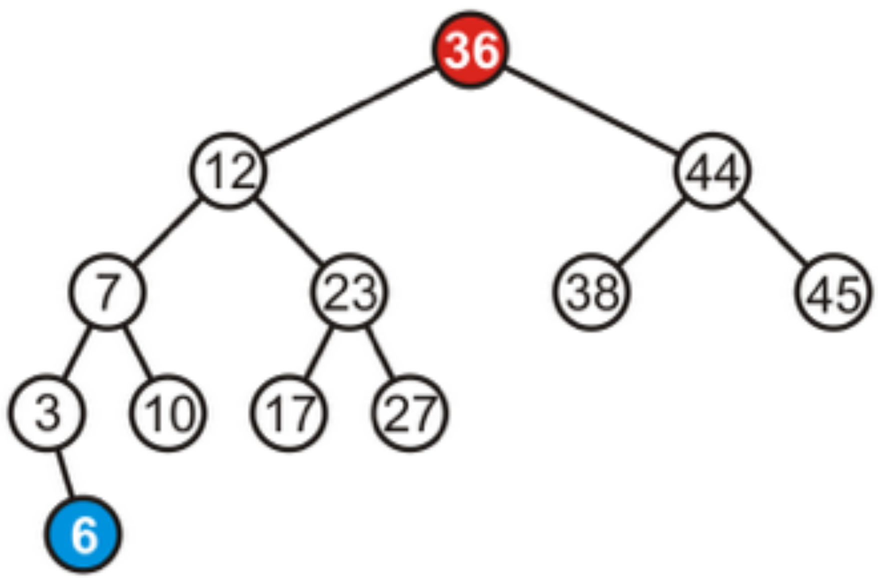
The height of each of the trees in the path back to the root are increased by one

- However, only the root node is now unbalanced



Maintaining Balance

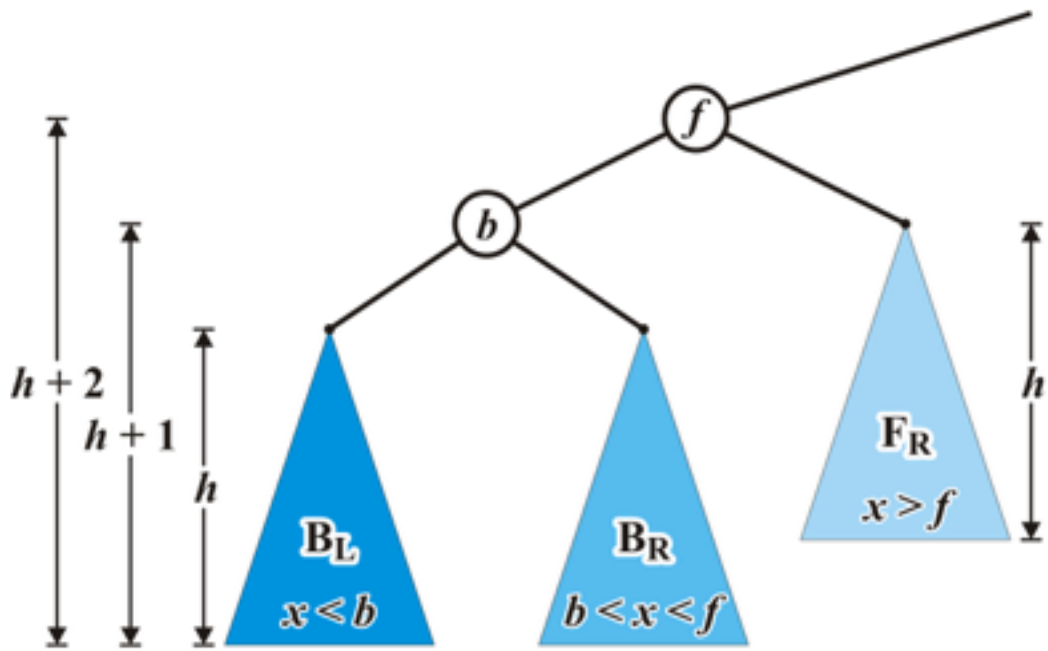
To fix this, we will look at the general case…



Maintaining Balance: Case 1

Consider the following setup

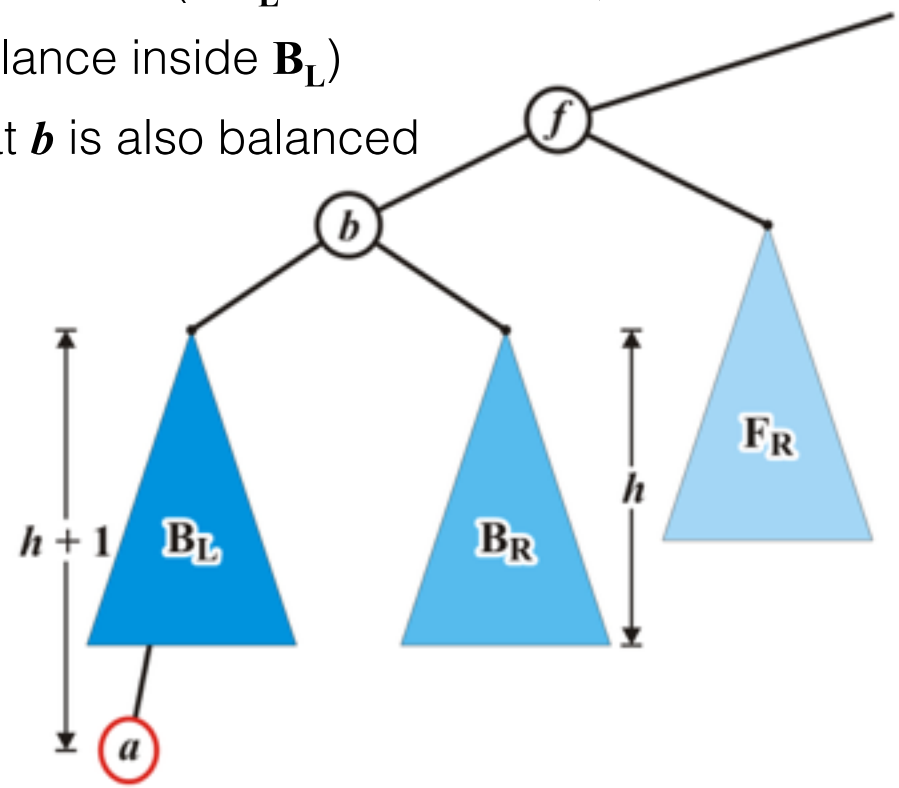
- Each blue triangle represents a tree of height h



Maintaining Balance: Case 1

Insert $\boldsymbol{a}$ into this tree: it falls into the left subtree $\mathbf{B_L}$ of $\boldsymbol{b}$

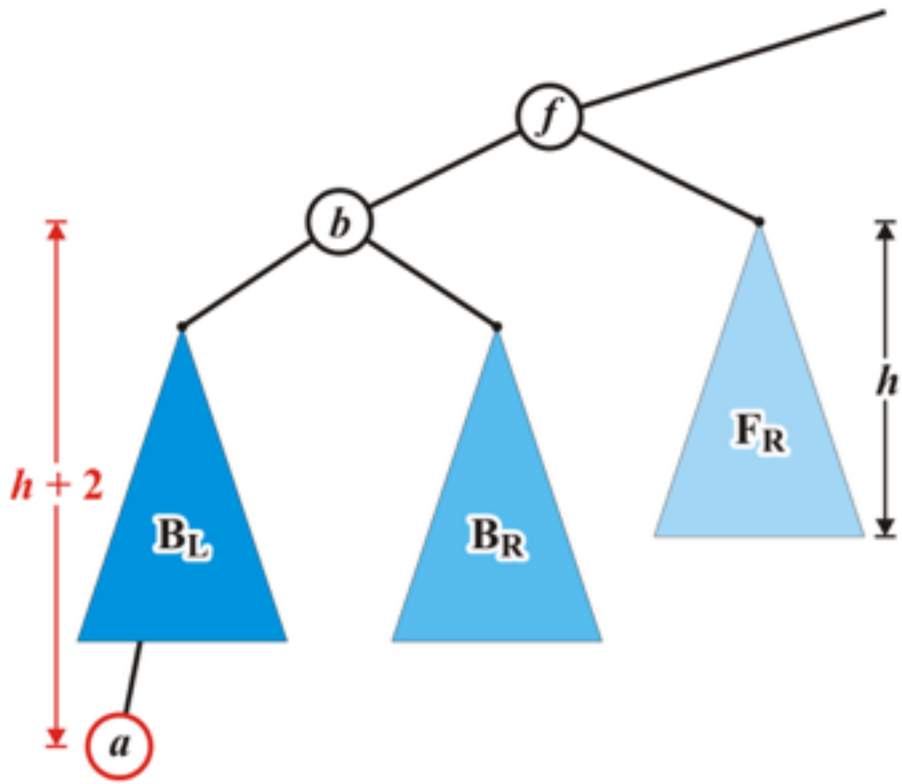
- Assume $\mathbf{B_L}$ remains balanced (if $\mathbf{B_L}$ is imbalanced, we should fix the imbalance inside $\mathbf{B_L}$)
- Thus, the tree rooted at $\boldsymbol{b}$ is also balanced



Maintaining Balance: Case 1

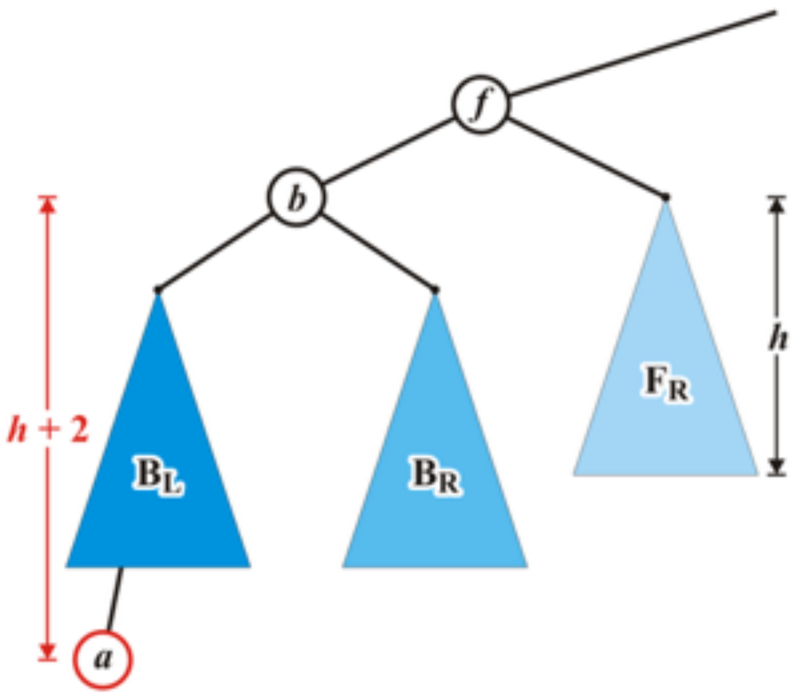
The tree rooted at node ***f*** is now unbalanced

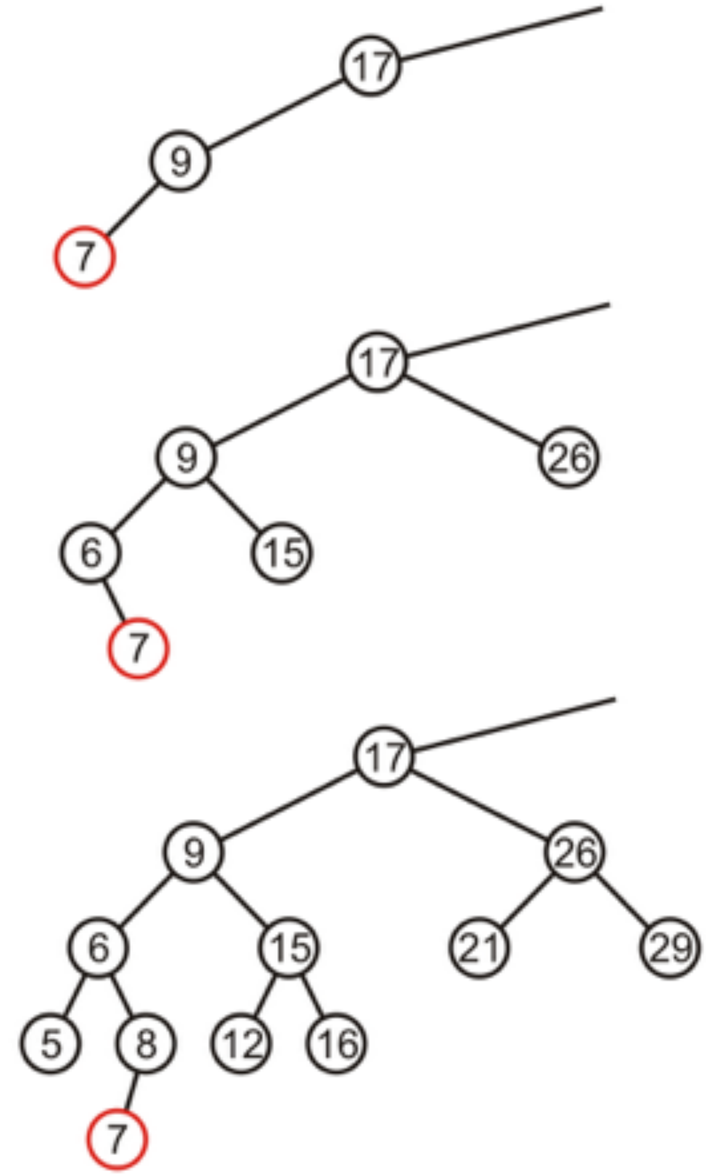
- We will correct the imbalance at this node



Maintaining Balance: Case 1

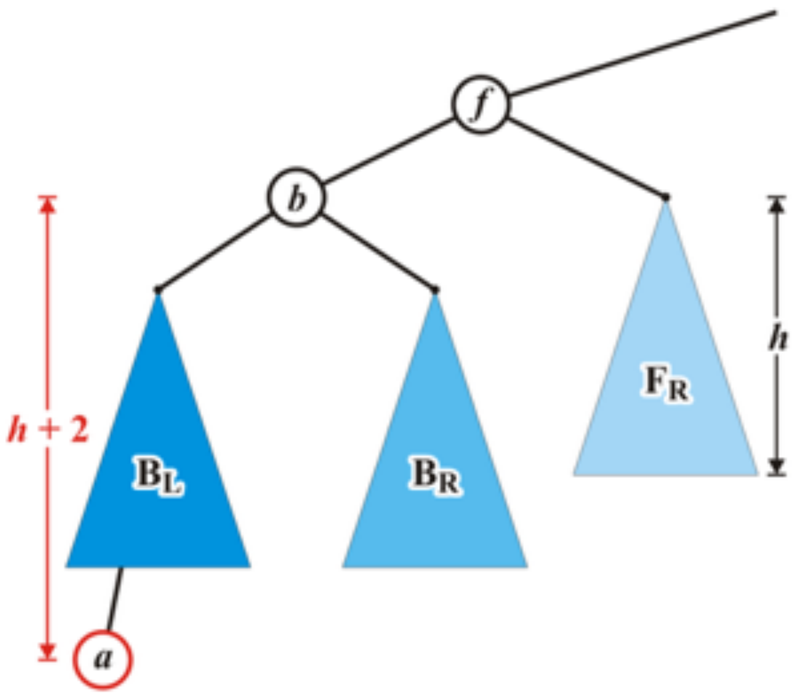
Here are examples of when the insertion of 7 may cause this situation when $h = -1$, 0, and 1

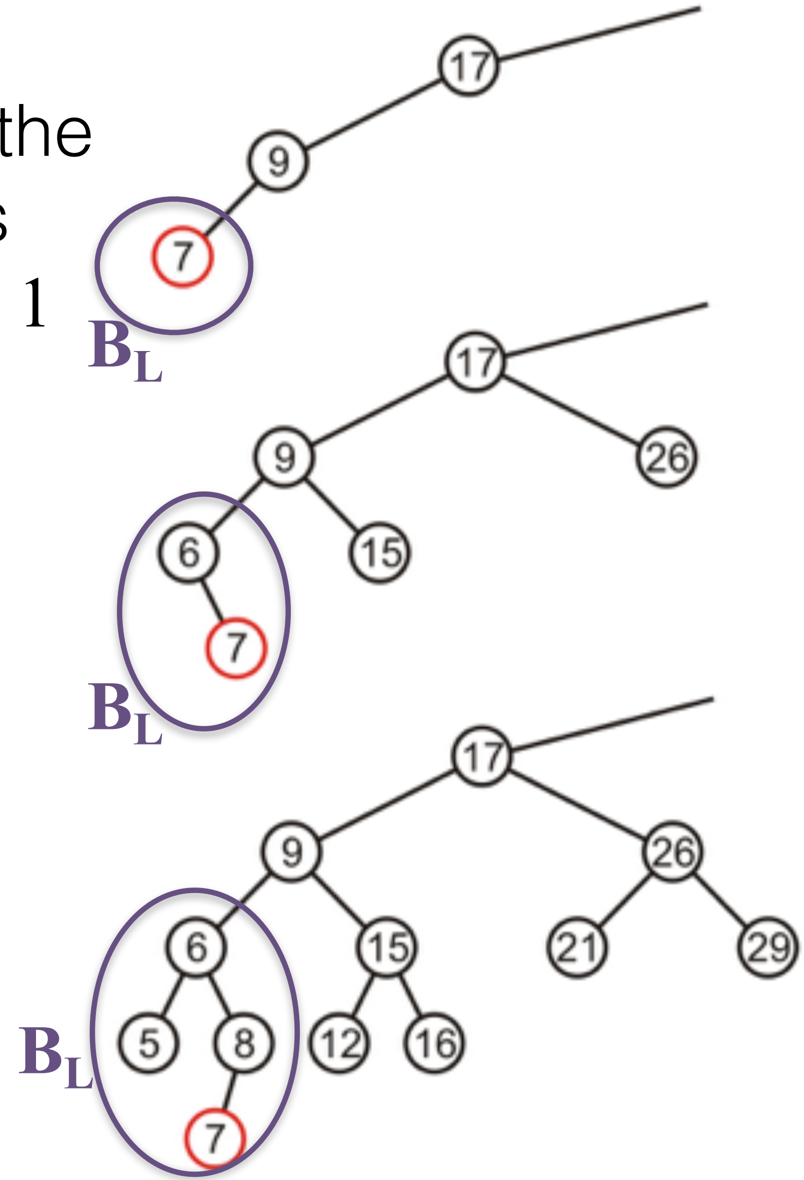




Maintaining Balance: Case 1

Here are examples of when the insertion of 7 may cause this situation when $h = -1$, 0, and 1



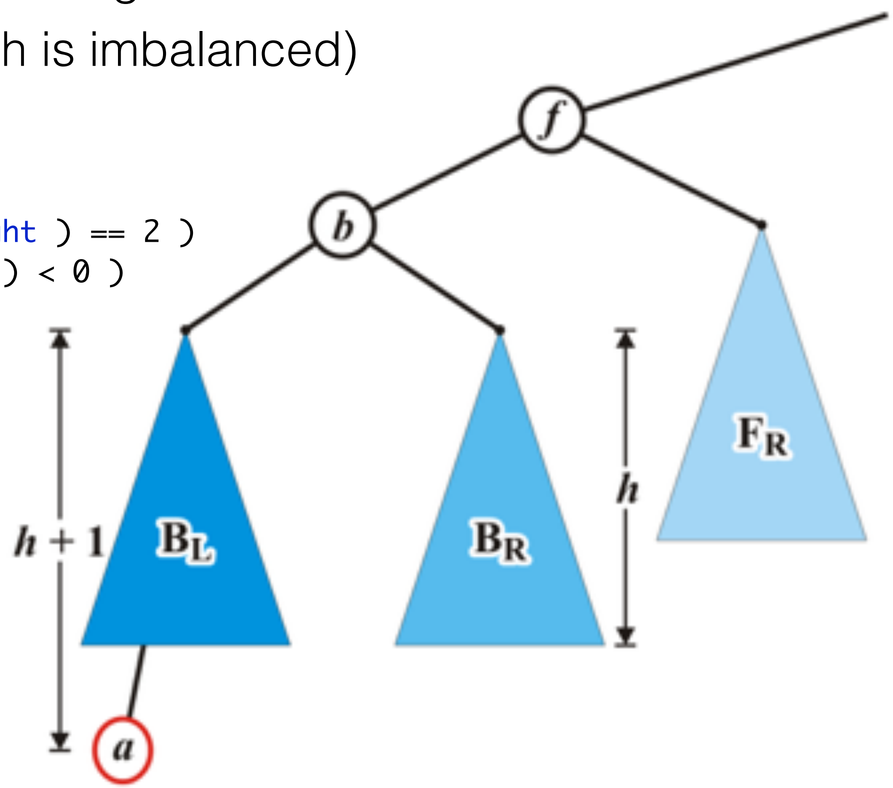


Maintaining Balance: Case 1

Insert ***a*** into this tree: it falls into the left subtree $\mathbf{B_L}$ of ***b***

– how to find it? check the height after insertion:
(here `t` is node f which is imbalanced)

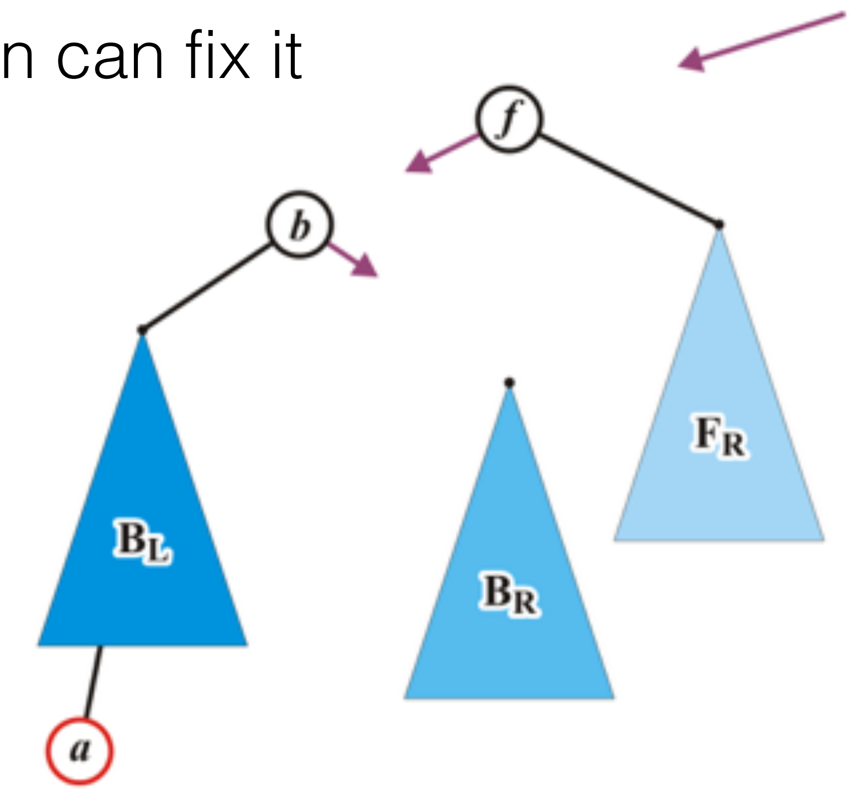
```
if( x.compareTo( t.element ) < 0 ) {
    t.left = insert( x, t.left );
    if( height( t.left ) - height( t.right ) == 2 )
        if( x.compareTo( t.left.element ) < 0 )
            ...
```



Maintaining Balance: Case 1

We need to rearrange these sub-trees ($\mathbf{B_L}$, $\mathbf{B_R}$, $\mathbf{F_R}$) to balance the tree again.

- Just one rotation can fix it

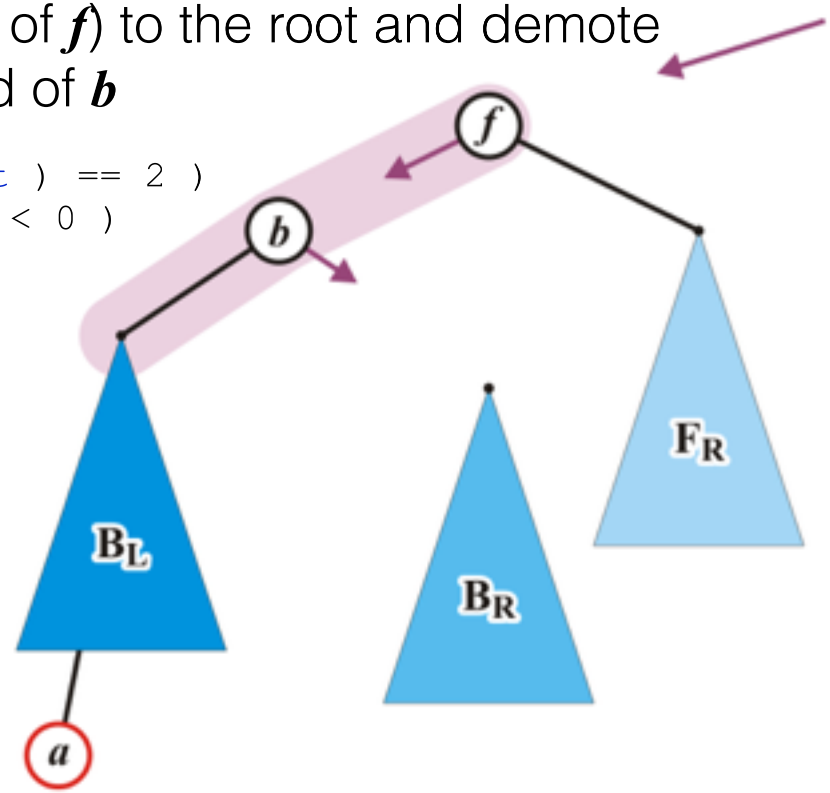


Maintaining Balance: Case 1

Specifically, we will rotate these two nodes around the root:

- Recall the first prototypical example (left-left imbalance)
- Promote node ***b*** (left child of ***f***) to the root and demote node ***f*** to be the right child of ***b***

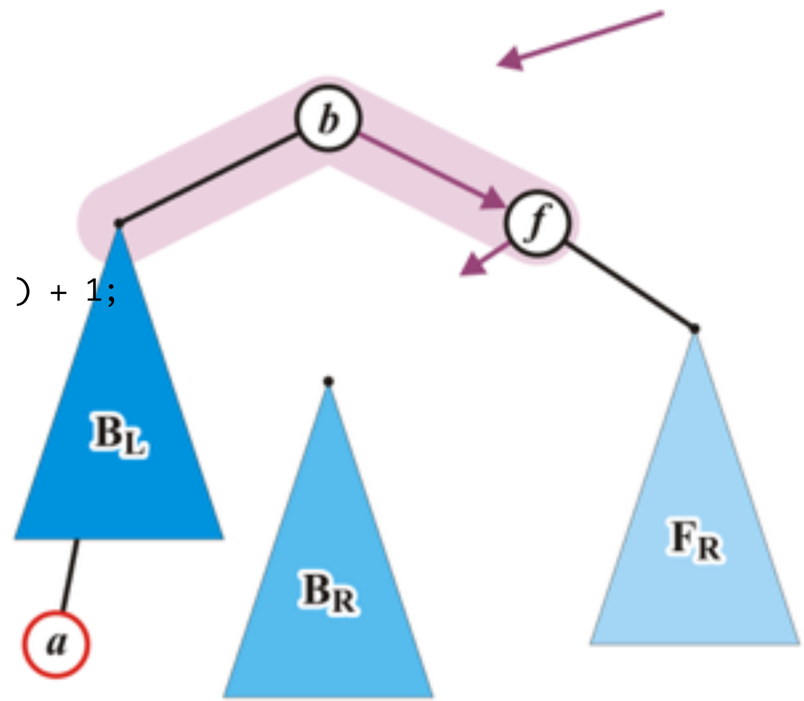
```
if( height( t.left ) - height( t.right ) == 2 )
    if( x.compareTo( t.left.element ) < 0 )
        t = rotateWithLeftChild( t );
    ...
```



Maintaining Balance: Case 1

This requires the `right` child of ***b*** (sub-tree $\mathbf{B_R}$)to be assigned as `left` child of ***f*** and node ***f*** to be assigned as the `right` child of ***b***

```
private static AVLNode rotateWithLeftChild( AVLNode f )
{
    AVLNode b = f.left;
    f.left = b.right;  //B_R
    b.right = f;
    f.height = max( height( f.left ), height( f.right ) ) + 1;
    b.height = max( height( b.left ), f.height ) + 1;
    return b;
}
```

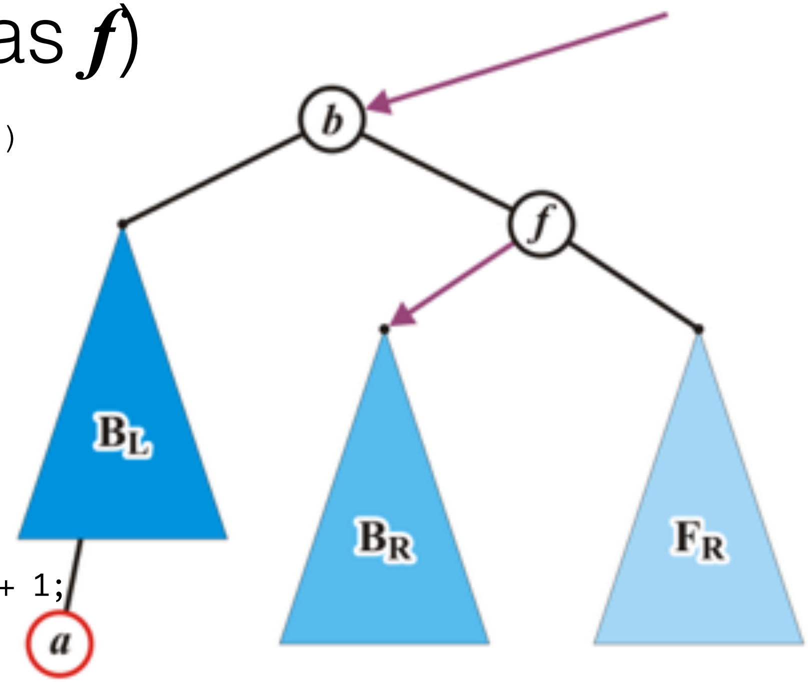


Maintaining Balance: Case 1

- This is the result
- We just need to return ***b*** as the root node of the tree (previously it was ***f***)

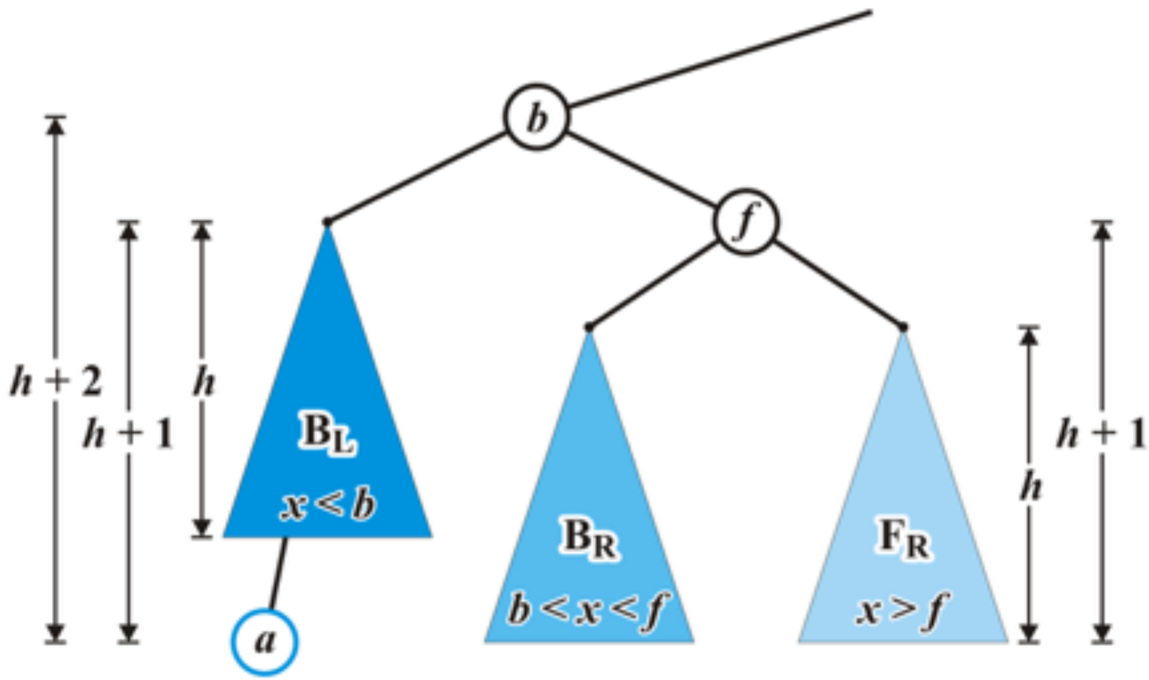
```
if( height( t.left ) - height( t.right ) == 2 )
    if( x.compareTo( t.left.element ) < 0 )
        t = rotateWithLeftChild( t );
    ...
```

```
private static AVLNode rotateWithLeftChild( AVLNode f )
{
    AVLNode b = f.left;
    f.left = b.right;  //B_R
    b.right = f;
    f.height = max( height( f.left ), height( f.right ) ) + 1;
    b.height = max( height( b.left ), f.height ) + 1;
    return b;
}
```



Maintaining Balance: Case 1

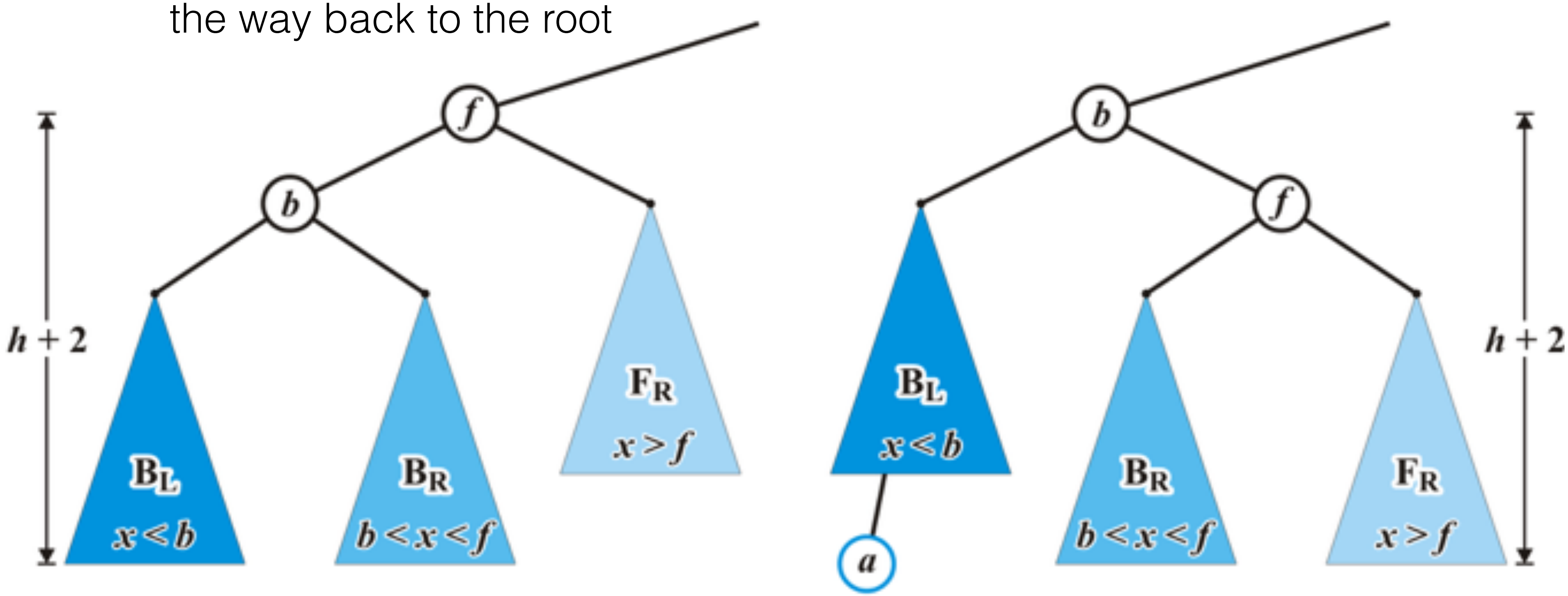
The nodes ***b*** and ***f*** are now balanced and all remaining nodes of the subtrees are in their correct positions



Maintaining Balance: Case 1

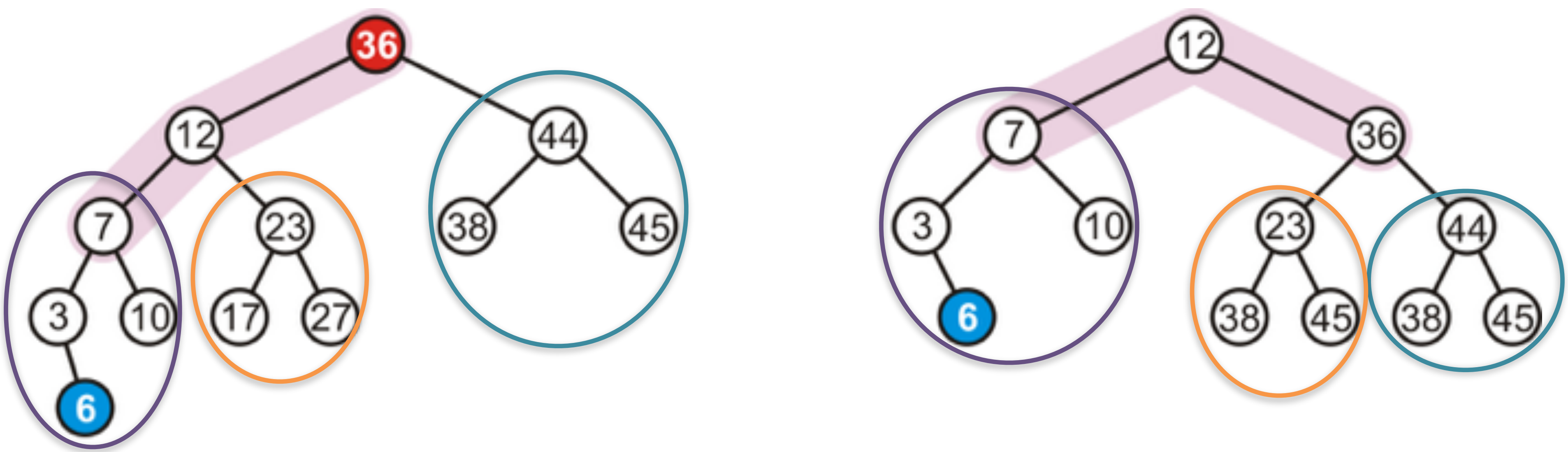
Now, height of the tree rooted at ***b*** equals the original height of the tree rooted at ***f*** (before the insertion)

- Thus, this insertion will no longer affect the balance of any ancestors all the way back to the root



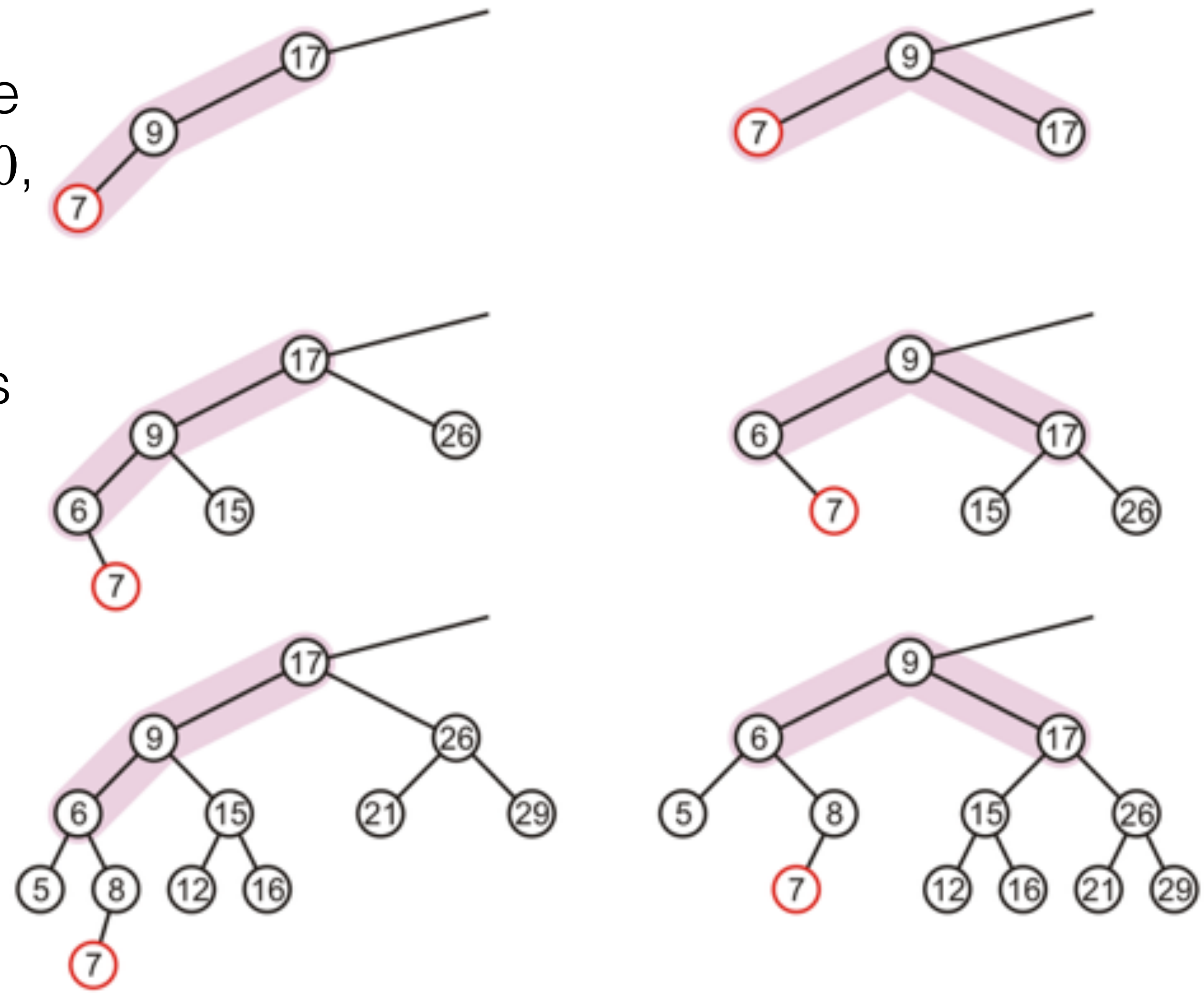
Maintaining Balance: Case 1

In our example case, the correction



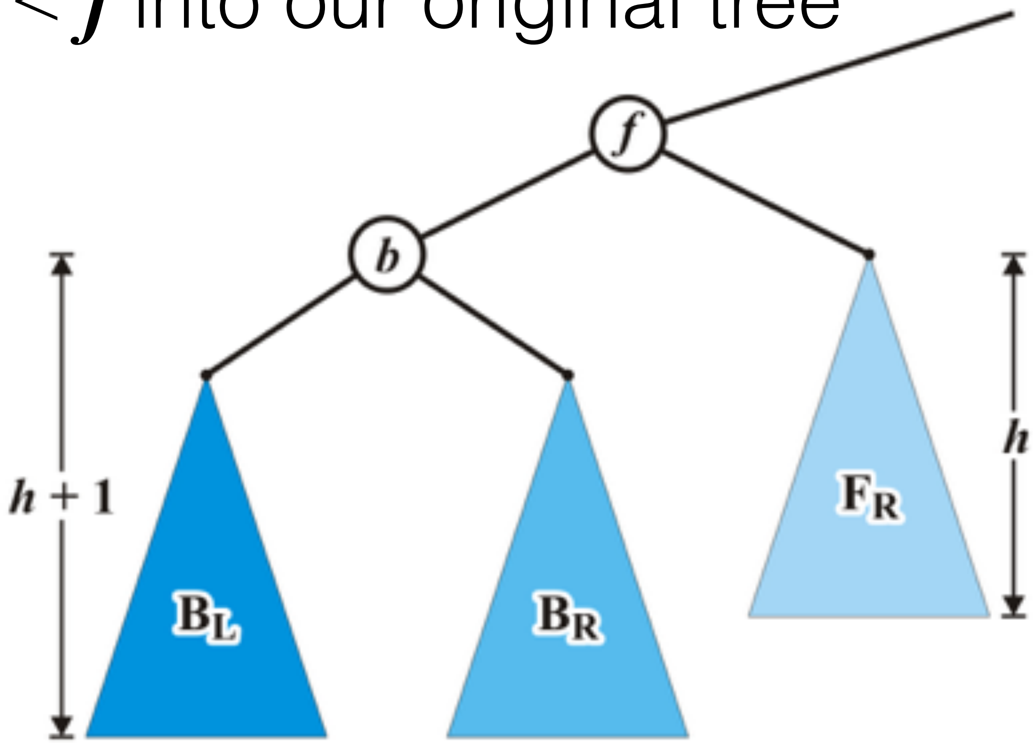
Maintaining Balance: Case 1

In our three sample cases with $h = -1$, 0, and 1, the node is now balanced and the same height as the tree before the insertion



Maintaining Balance: Case 2

Alternatively, consider the insertion of $\boldsymbol{c}$ where $\boldsymbol{b} < \boldsymbol{c} < \boldsymbol{f}$ into our original tree

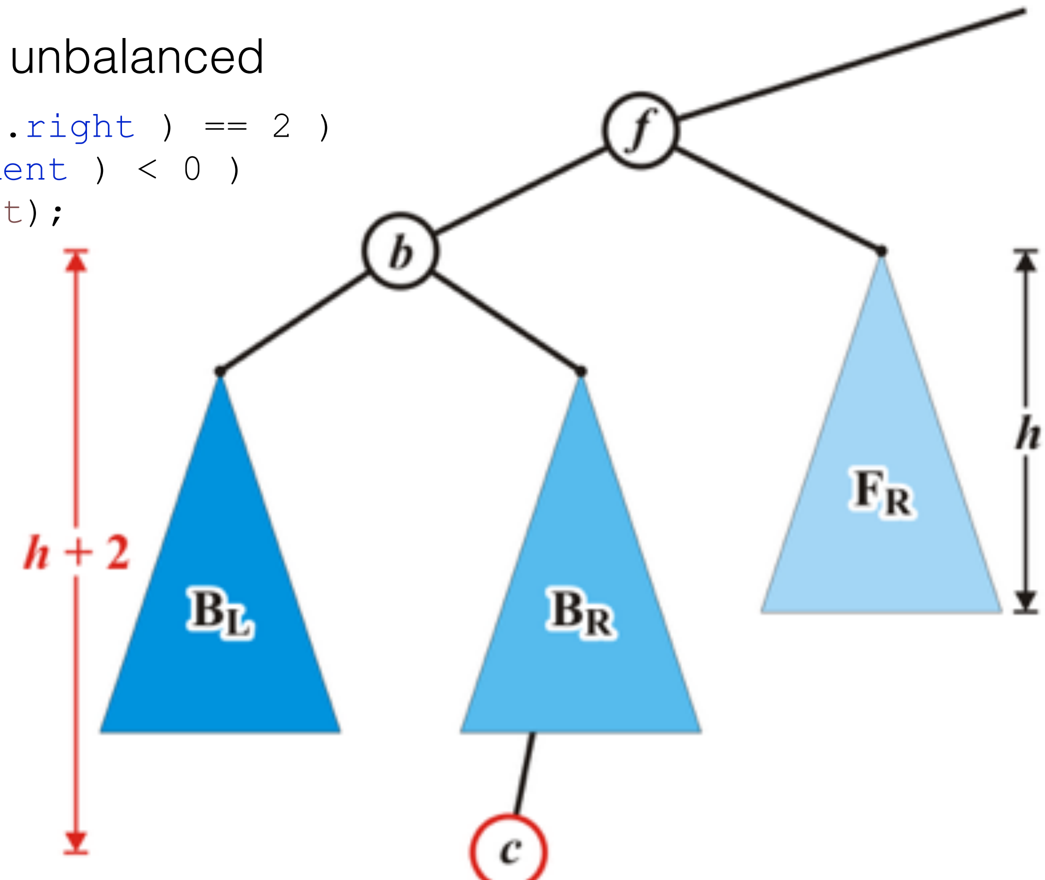


Maintaining Balance: Case 2

Assume that the insertion of c increases the height of $\mathbf{B_R}$

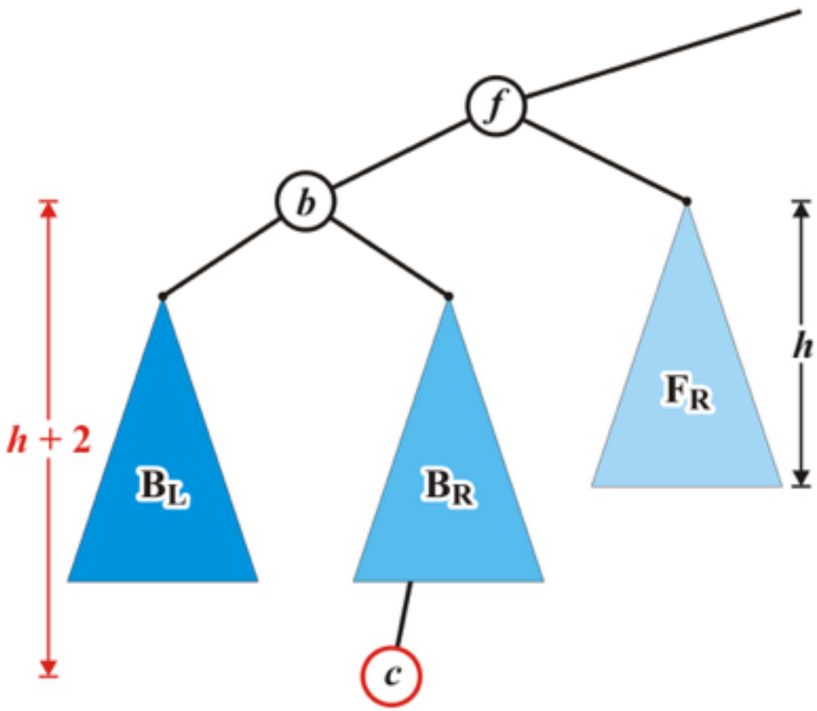
– Once again, f becomes unbalanced

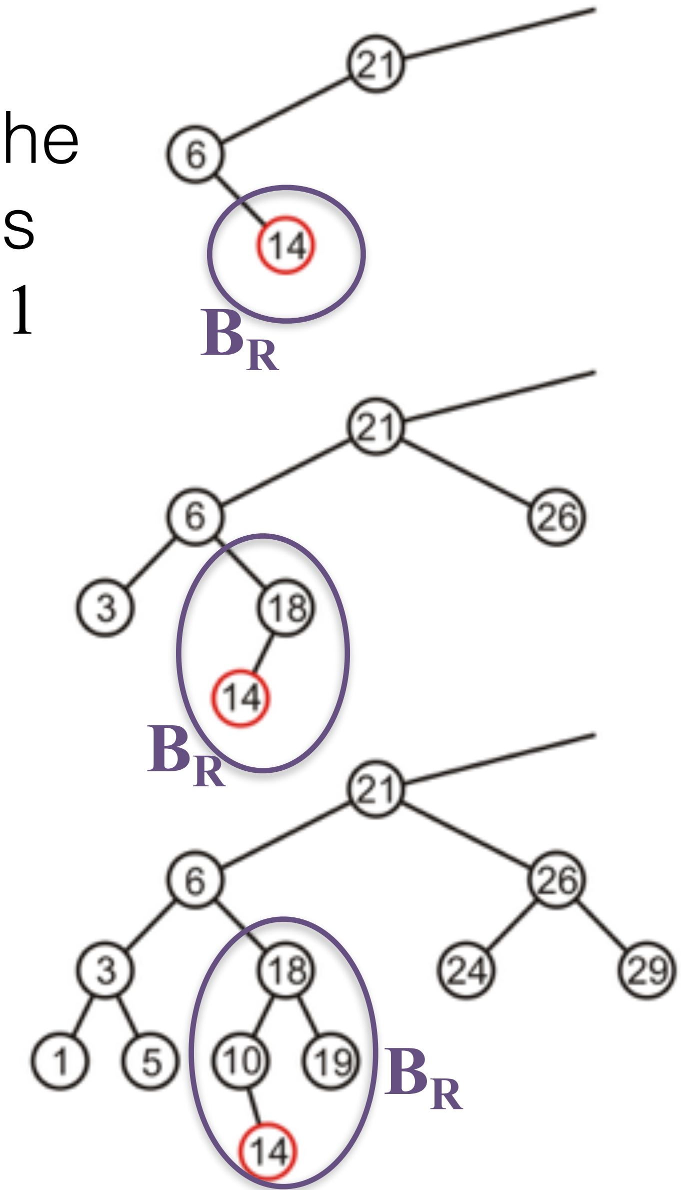
```
if( height( t.left ) - height( t.right ) == 2 )
    if( x.compareTo( t.left.element ) < 0 )
        t = rotateWithLeftChild(t);
    else
        ??
```



Maintaining Balance: Case 2

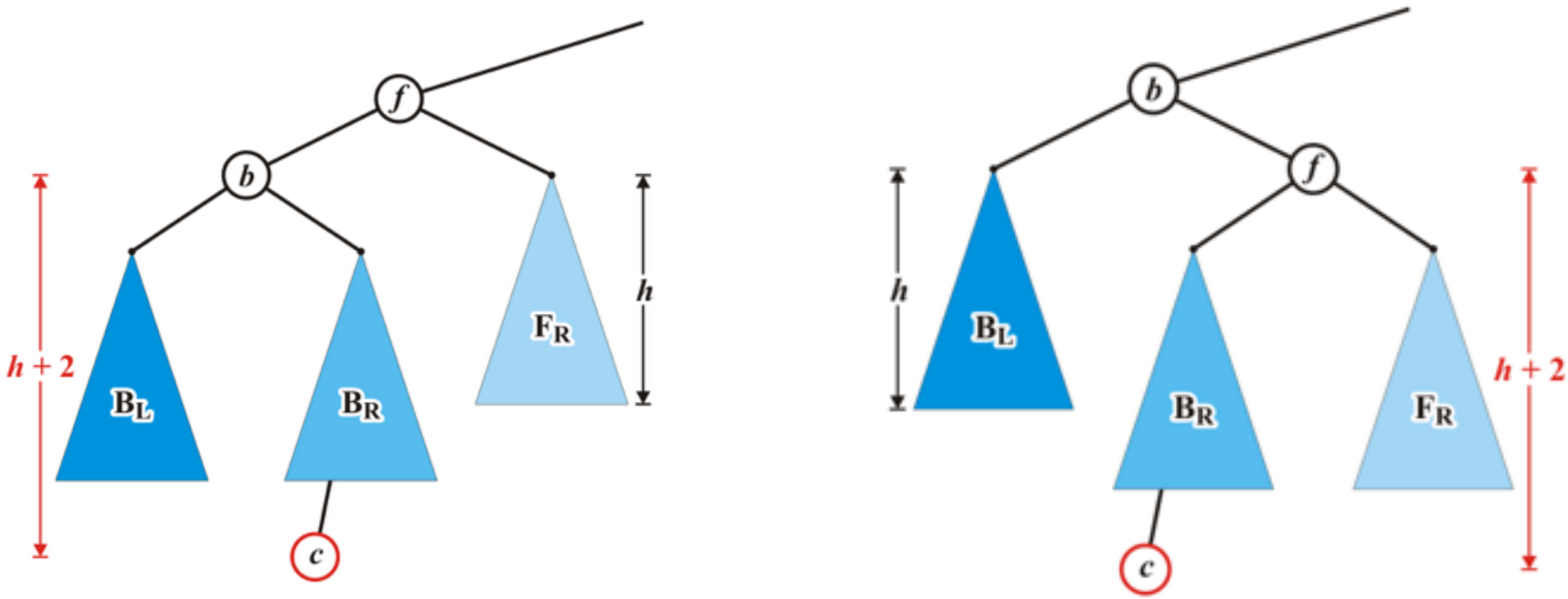
Here are examples of when the insertion of 14 may cause this situation when $h = -1$, 0, and 1





Maintaining Balance: Case 2

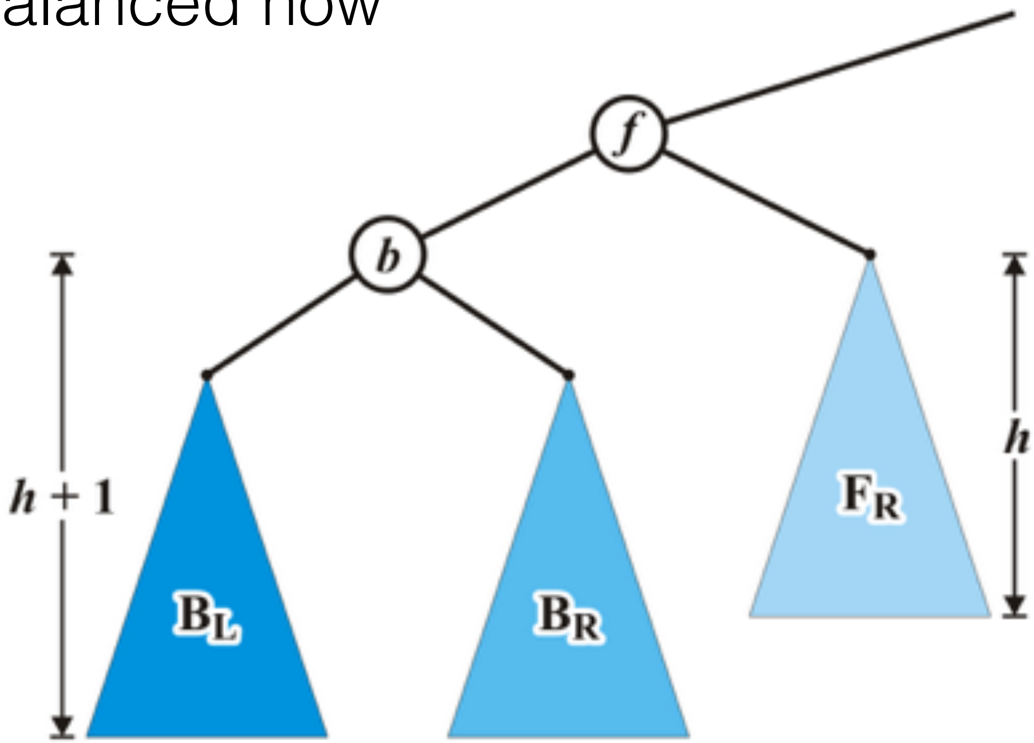
Unfortunately, the previous correction does not fix the imbalance at the root of this sub-tree: the new root, ***b***, remains unbalanced



Maintaining Balance: Case 2

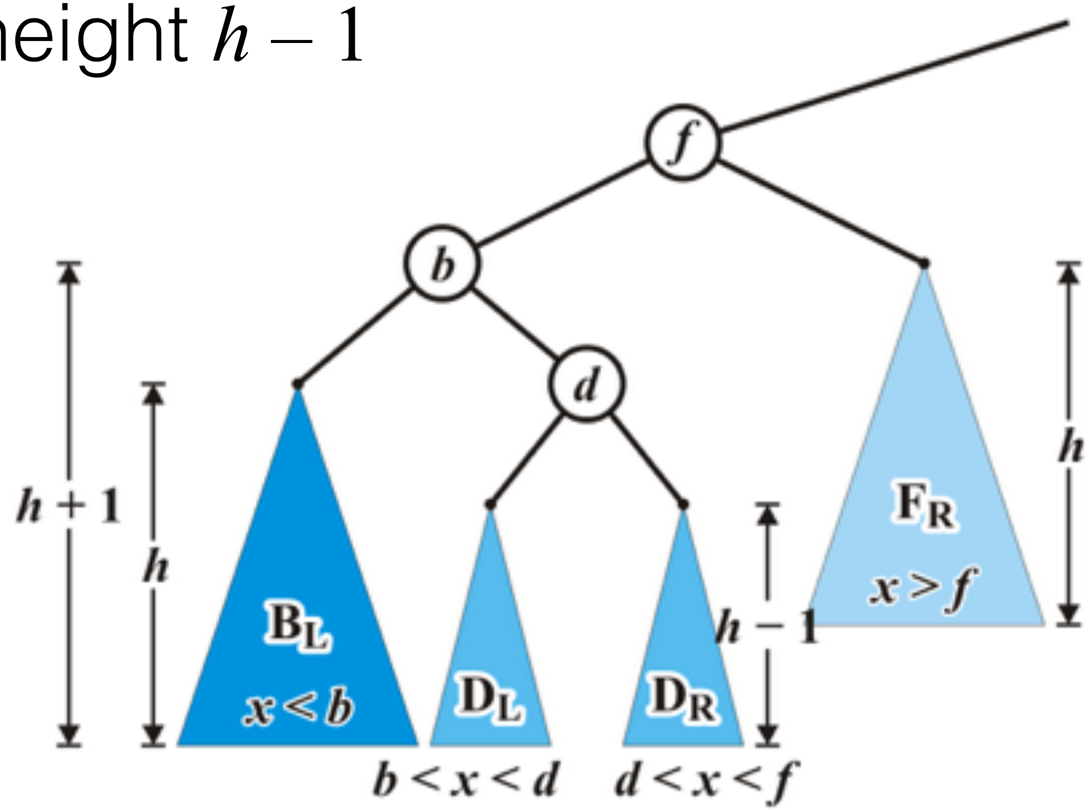
This is the sub-tree before insertion

- ***f*** is balanced now



Maintaining Balance: Case 2

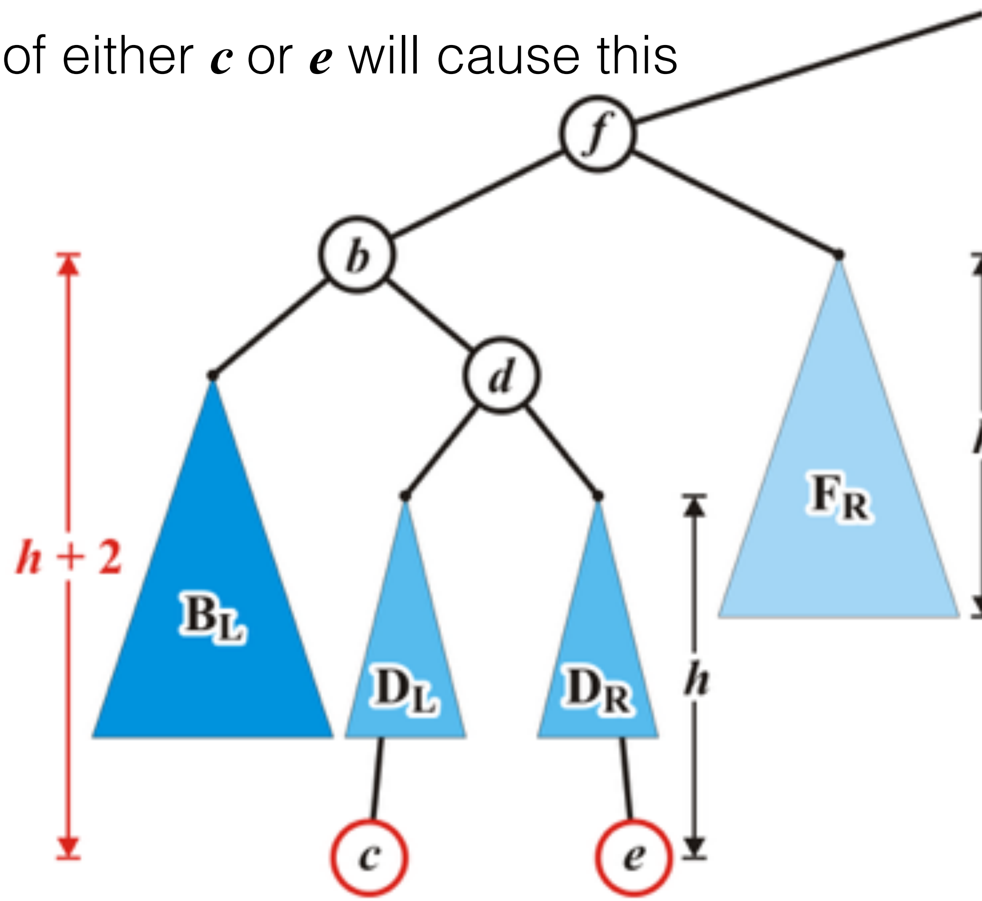
Re-label the tree by dividing the left subtree of $\boldsymbol{b}$ into a tree rooted at $\boldsymbol{d}$ with two subtrees of height $h - 1$



Maintaining Balance: Case 2

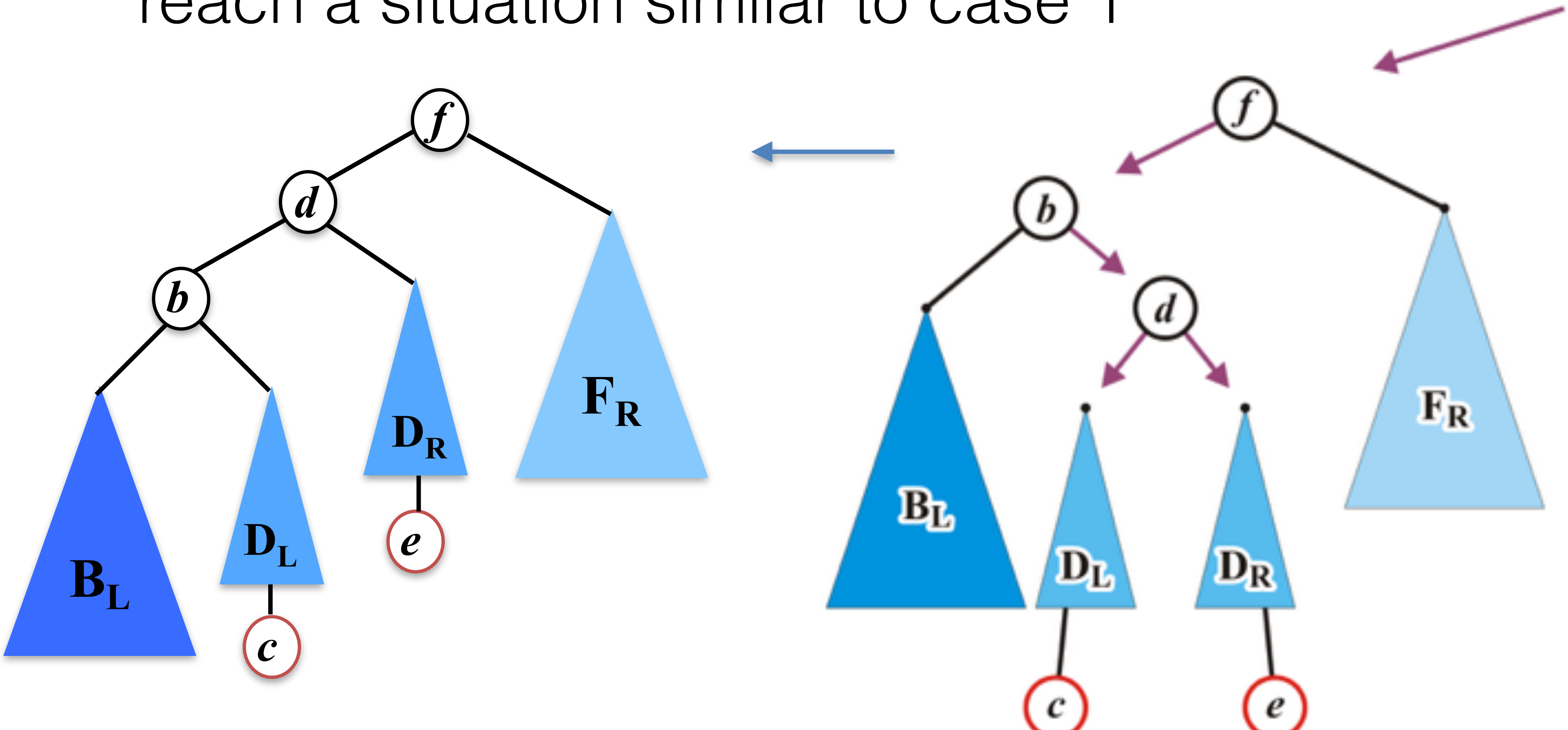
We can see an insertion causes an imbalance at ***f***

- The insertion of either ***c*** or ***e*** will cause this



Maintaining Balance: Case 2

If we first rotate b and its right child d, we can reach a situation similar to case 1

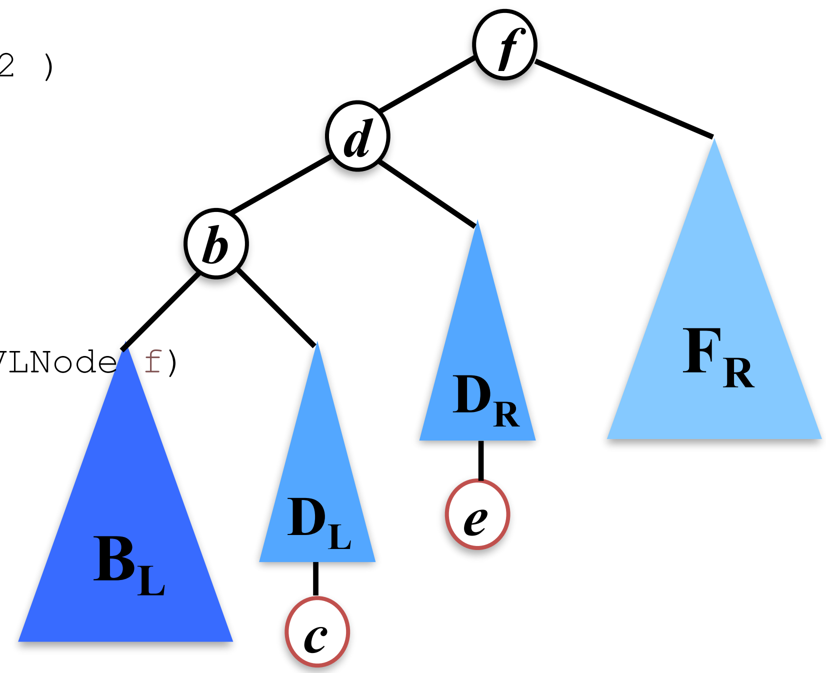


Maintaining Balance: Case 2

- But ***f*** is still imbalance, but the situation is just like case 1,
    - So we need to rotate ***f*** and ***d*** (the new left child of ***f***)

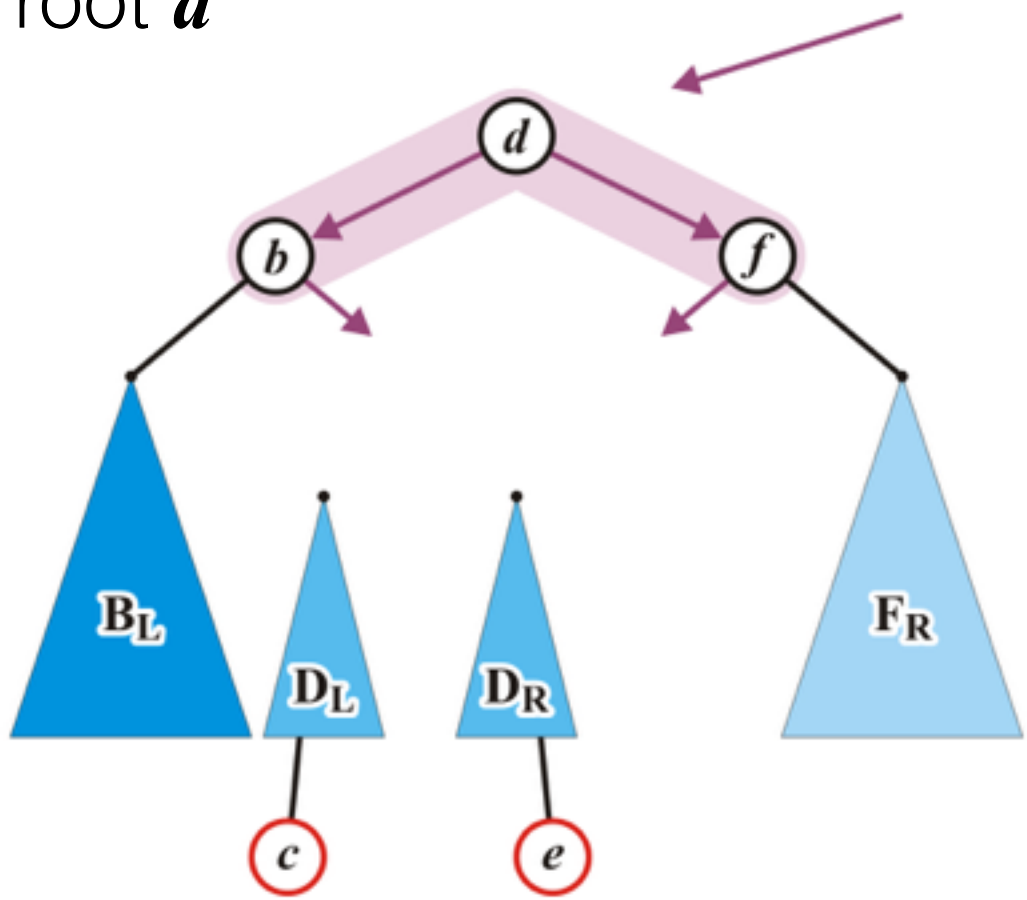
```
if( height( t.left ) - height( t.right ) == 2 )
    if( x.compareTo( t.left.element ) < 0 )
        t = rotateWithLeftChild(t);
    else
        t = doubleWithLeftChild( t );


private static AVLNode doubleWithLeftChild(AVLNode f)
{
    f.left = rotateWithRightChild(f.left);
    return rotateWithLeftChild(f);
}
```

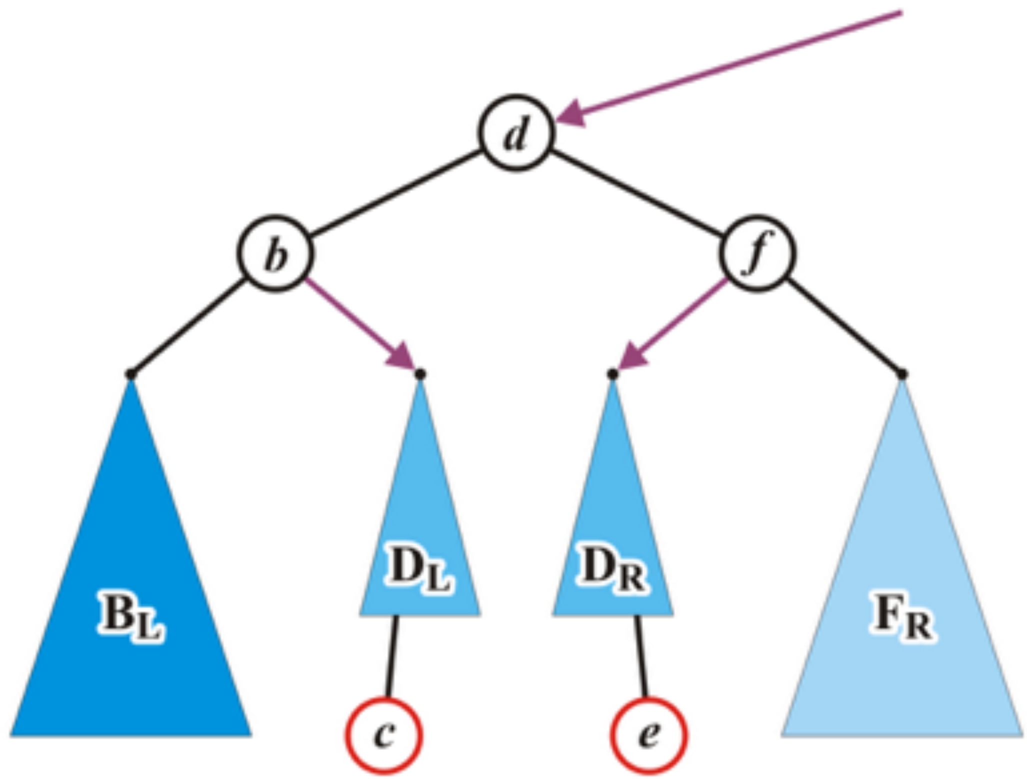


Maintaining Balance: Case 2

In fact, ***b*** and ***f*** will be assigned as children of the new root ***d***

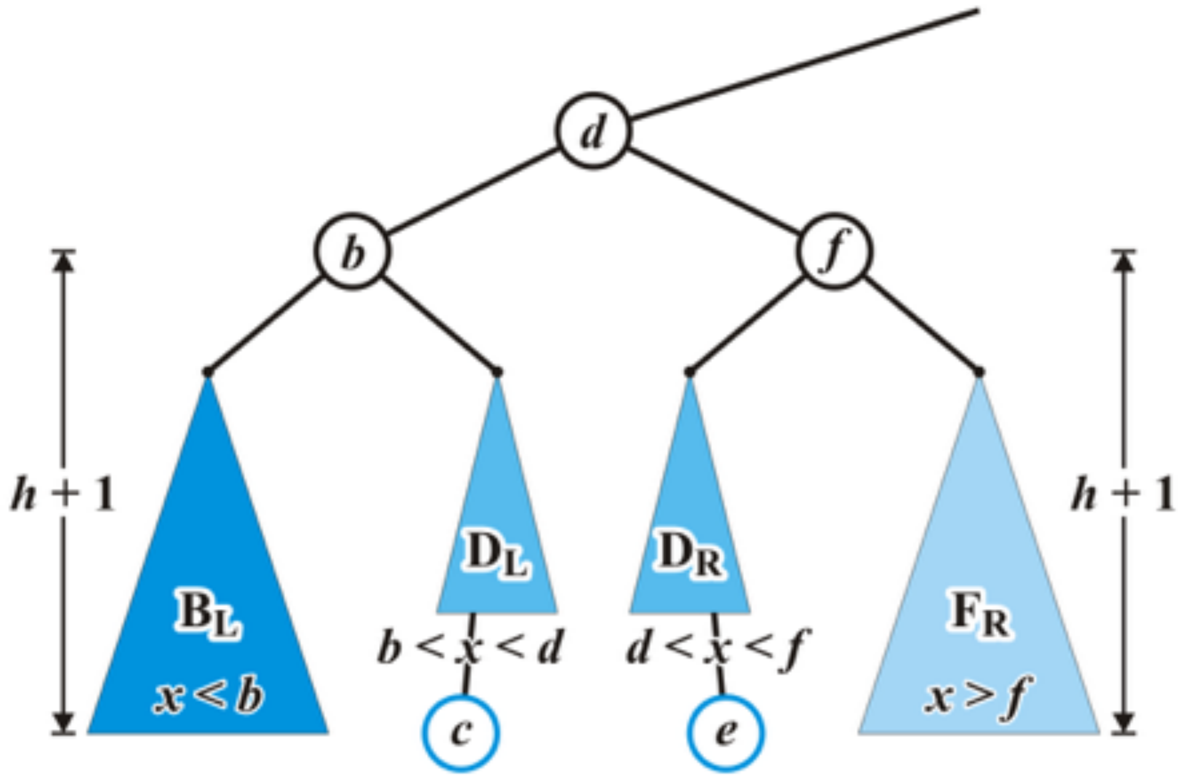


Maintaining Balance: Case 2



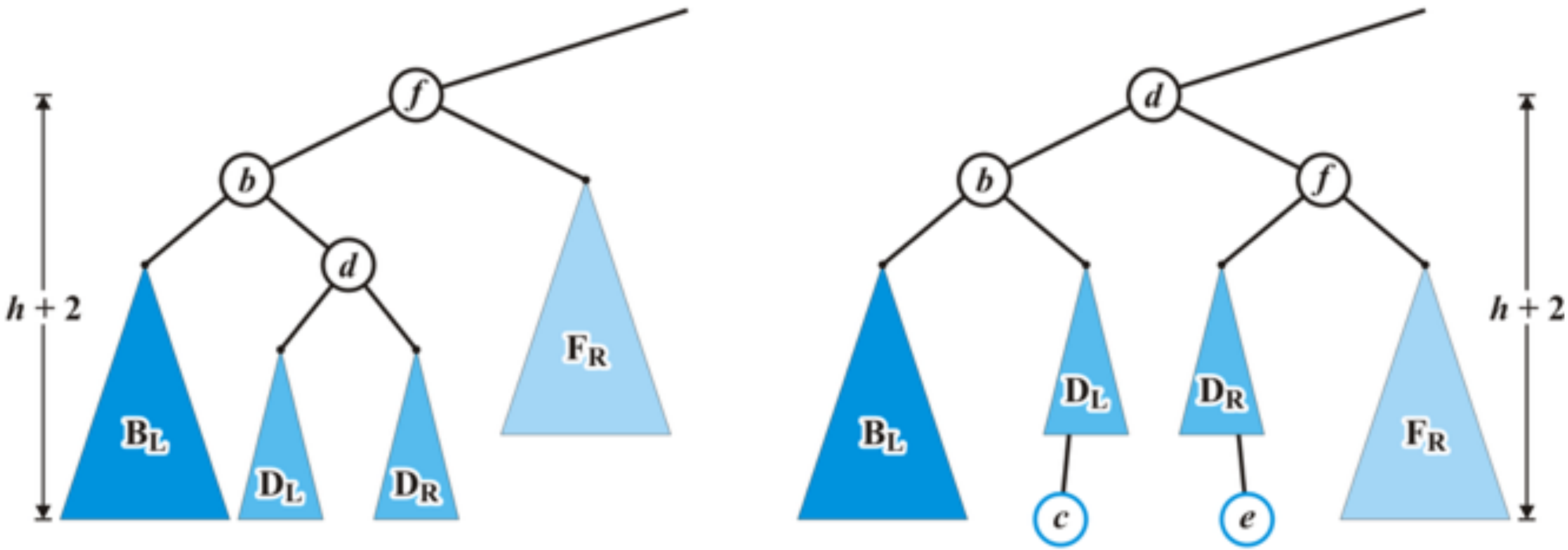
Maintaining Balance: Case 2

Now the tree rooted at ***d*** is balanced



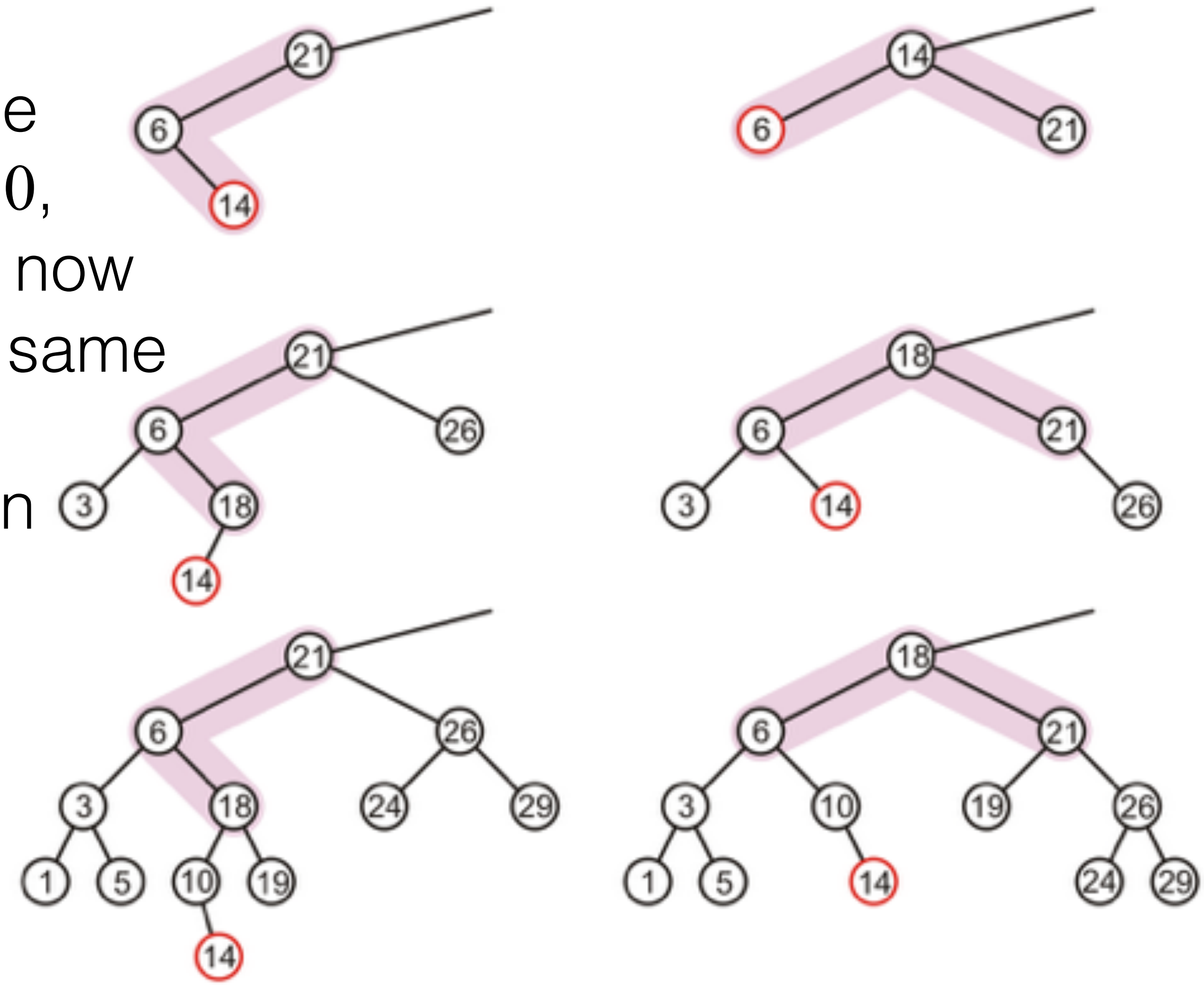
Maintaining Balance: Case 2

Again, the height of the root did not change



Maintaining Balance: Case 2

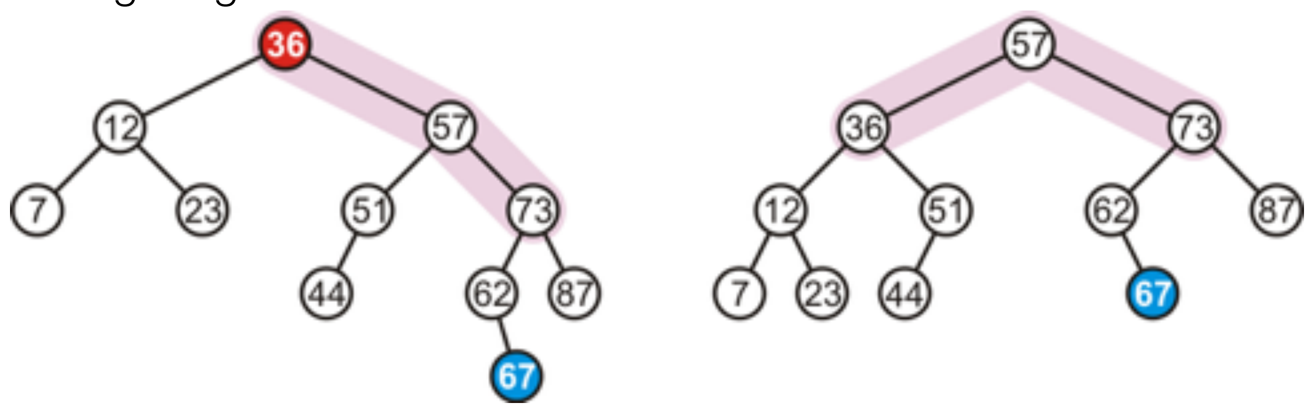
In our three sample cases with $h = -1$, 0, and 1, the node is now balanced and the same height as the tree before the insertion



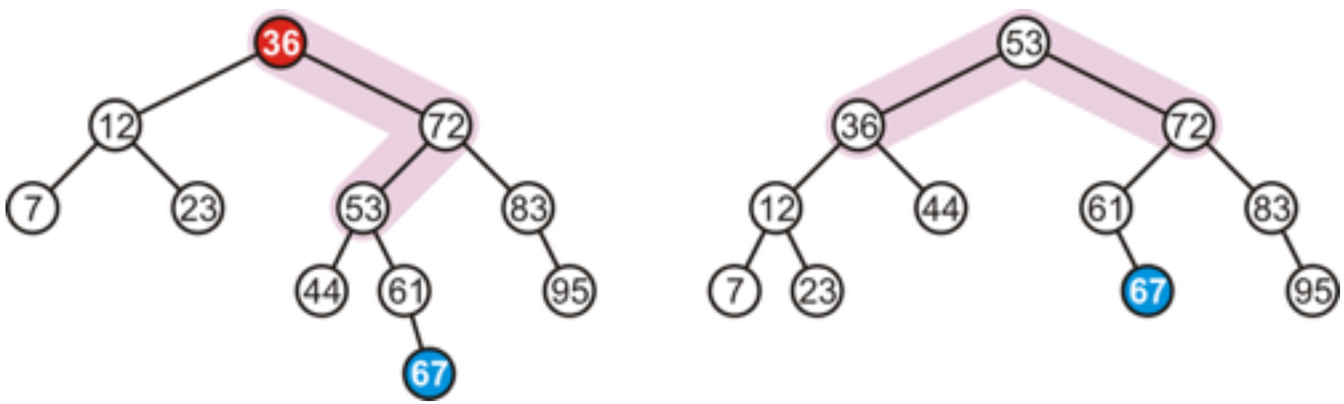
Maintaining balance: Summary

There are two symmetric cases to those we have examined:

- Insertions into the right-right sub-tree



- Insertions into either the right-left sub-tree



Insert

```
public void insert( Comparable x ) {
    root = insert( x, root );
}

private AVLNode insert( Comparable x, AVLNode t ) {
    if( t == null )
       t = new AVLNode( x, null, null );
    else if( x.compareTo( t.element ) < 0 ) {
       t.left = insert( x, t.left );
       if( height( t.left ) - height( t.right ) == 2 )
           if( x.compareTo( t.left.element ) < 0 )
              t = rotateWithLeftChild( t );
           else
              t = doubleWithLeftChild( t );
    }
    else if( x.compareTo( t.element ) > 0 ) {
       t.right = insert( x, t.right );
       if( height( t.right ) - height( t.left ) == 2 )
           if( x.compareTo( t.right.element ) > 0 )
               t = rotateWithRightChild( t );
           else
               t = doubleWithRightChild( t );
    }
    else
       ;  // Duplicate; do nothing or you can throw an exception
    t.height = max( height( t.left ), height( t.right ) ) + 1;
    return t;
}
```

Insert

```
private static AVLNode 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;
    return k1;
}

/**
 * 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 static AVLNode rotateWithRightChild( AVLNode k1 )
{
    AVLNode k2 = k1.right;
    k1.right = k2.left;
    k2.left = k1;
    k1.height = max( height( k1.left ), height( k1.right ) ) + 1;
    k2.height = max( height( k2.right ), k1.height ) + 1;
    return k2;
}
```

Insert

```
 /**
 * 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 static AVLNode doubleWithLeftChild( AVLNode k3 )
{
    k3.left = rotateWithRightChild( k3.left );
    return rotateWithLeftChild( k3 );
}

/**
 * Double rotate binary tree node: first right child
 * with its left child; then node k1 with new right child.
 * For AVL trees, this is a double rotation for case 3.
 * Update heights, then return new root.
 */
private static AVLNode doubleWithRightChild( AVLNode k1 )
{
    k1.right = rotateWithLeftChild( k1.right );
    return rotateWithRightChild( k1 );
}
```

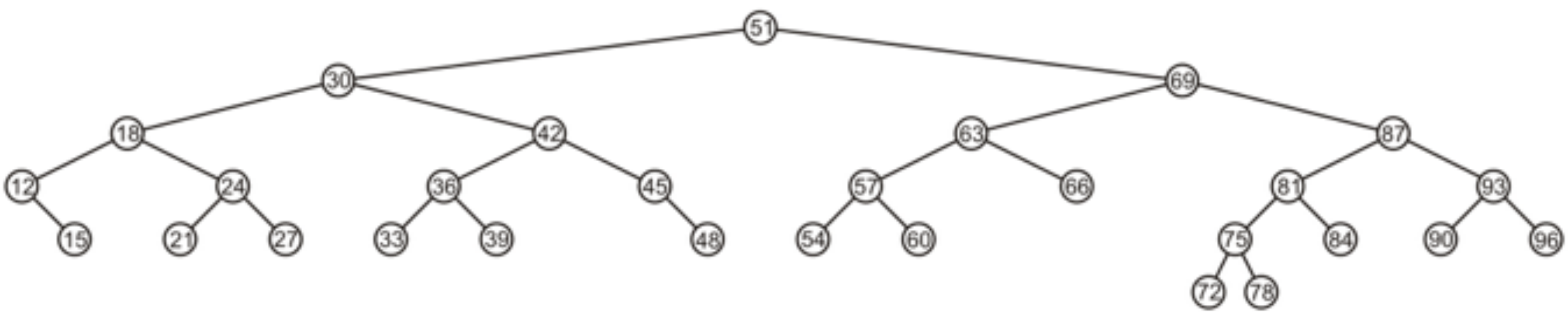
Insertion (Implementation)

Comments:

- Both balances are $\Theta(1)$
- All insertions are still $\Theta(\ln(n))$
- It is possible to ***tighten*** the previous code

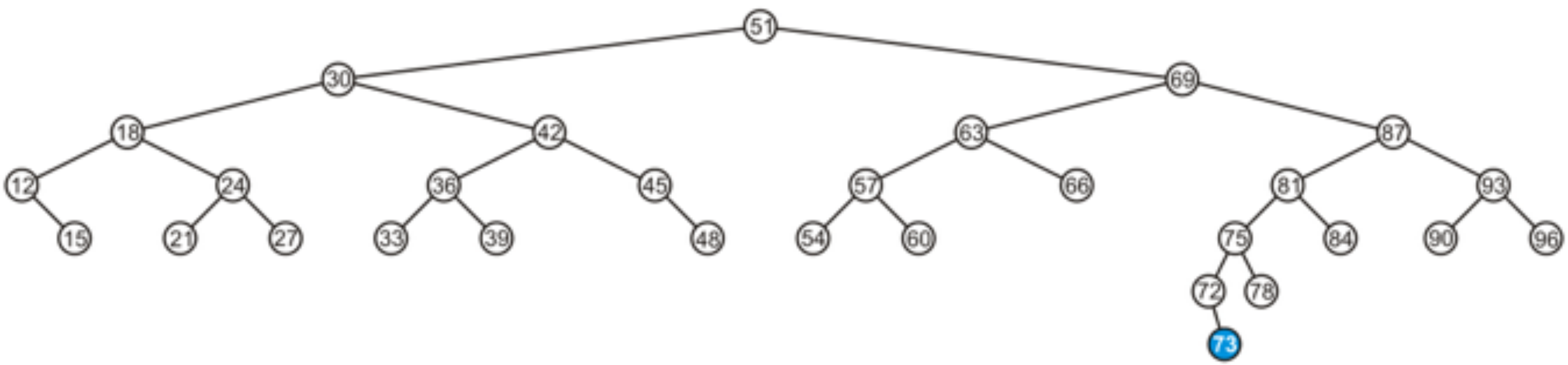
Insertion

Consider this AVL tree



Insertion

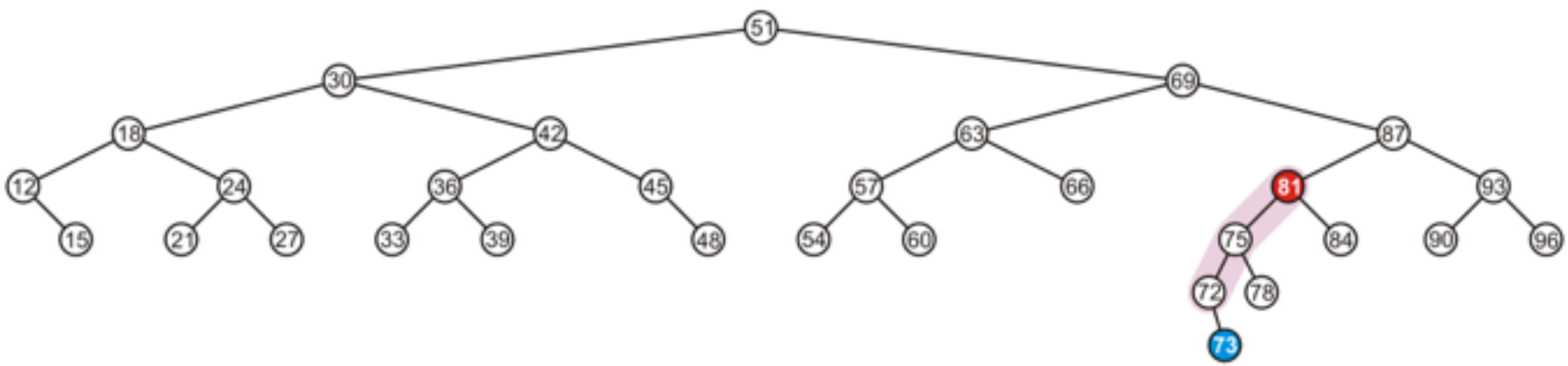
Insert 73



Insertion

The node 81 is unbalanced

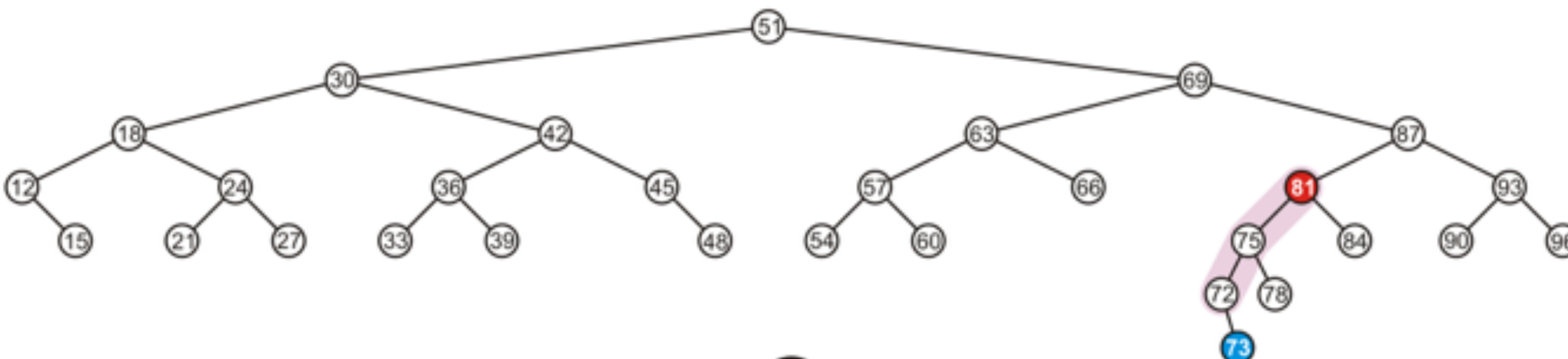
- A left-left imbalance

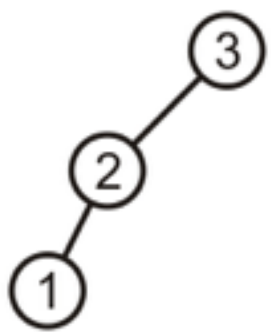


Insertion

The node 81 is unbalanced

- A left-left imbalance

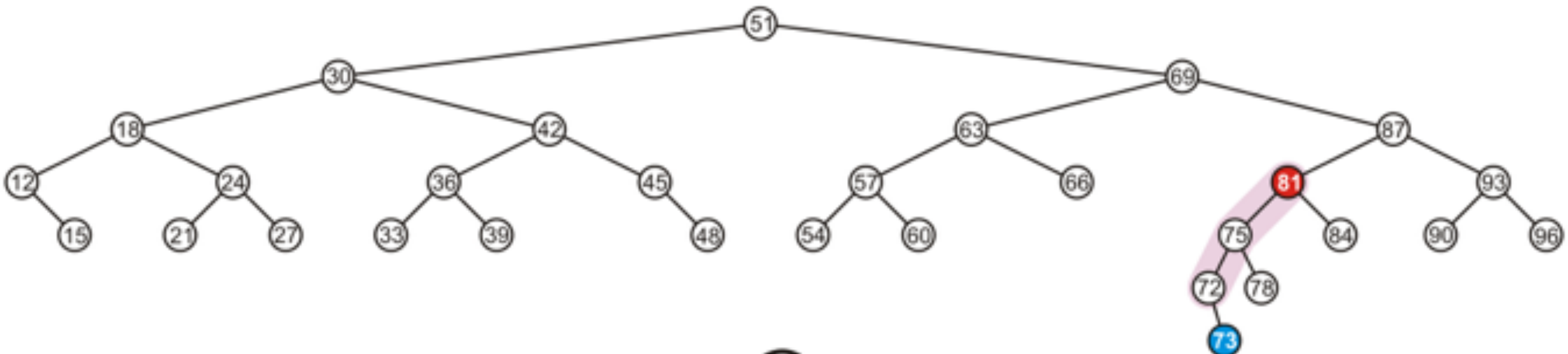


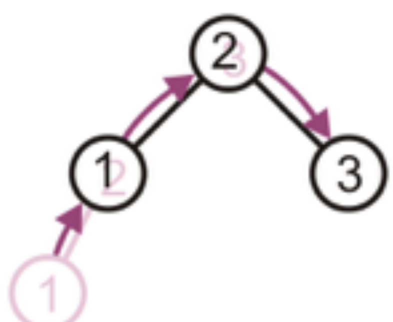


Insertion

The node 81 is unbalanced

- A left-left imbalance
- Promote the intermediate node to the imbalanced node

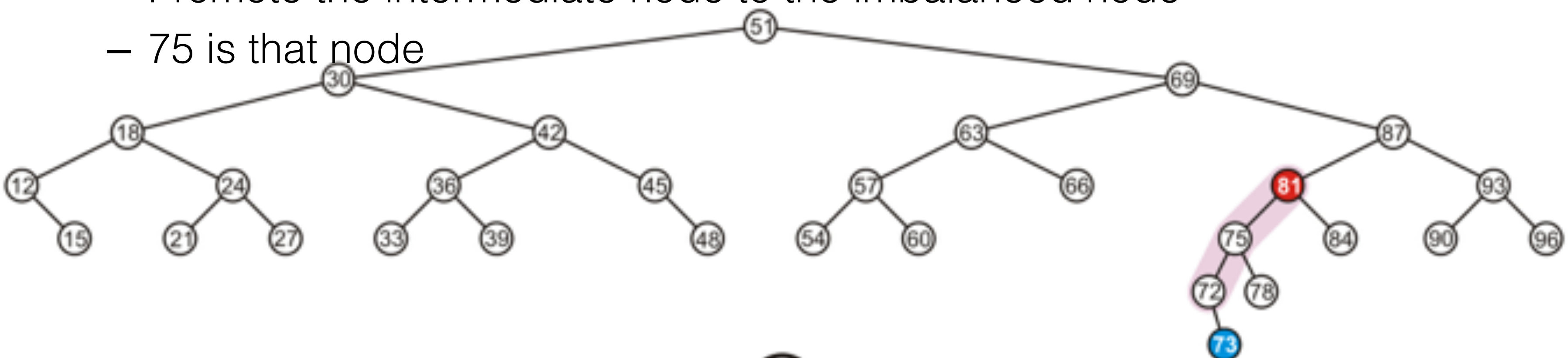


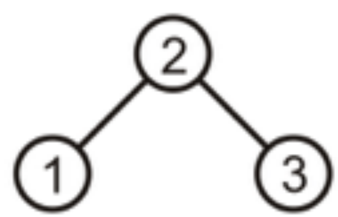


Insertion

The node 81 is unbalanced

- A left-left imbalance
- Promote the intermediate node to the imbalanced node
- 75 is that node

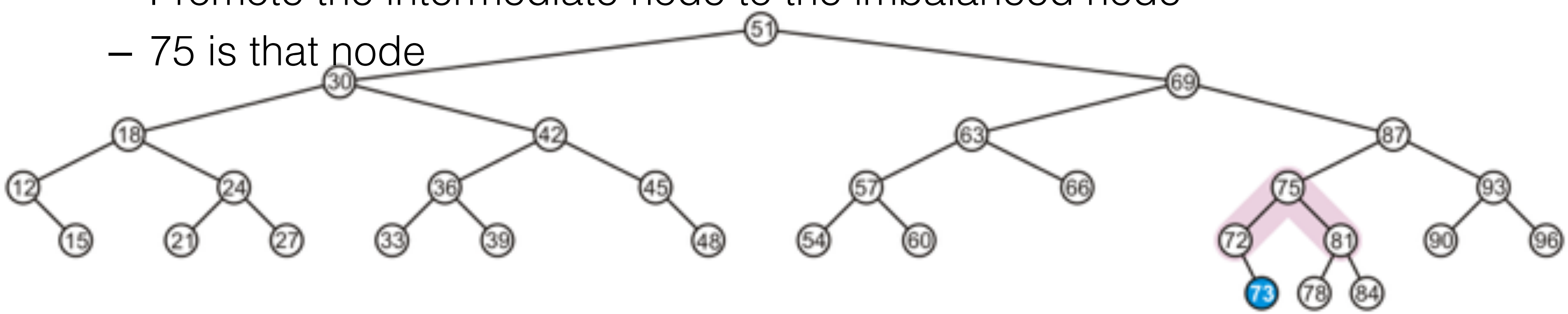




Insertion

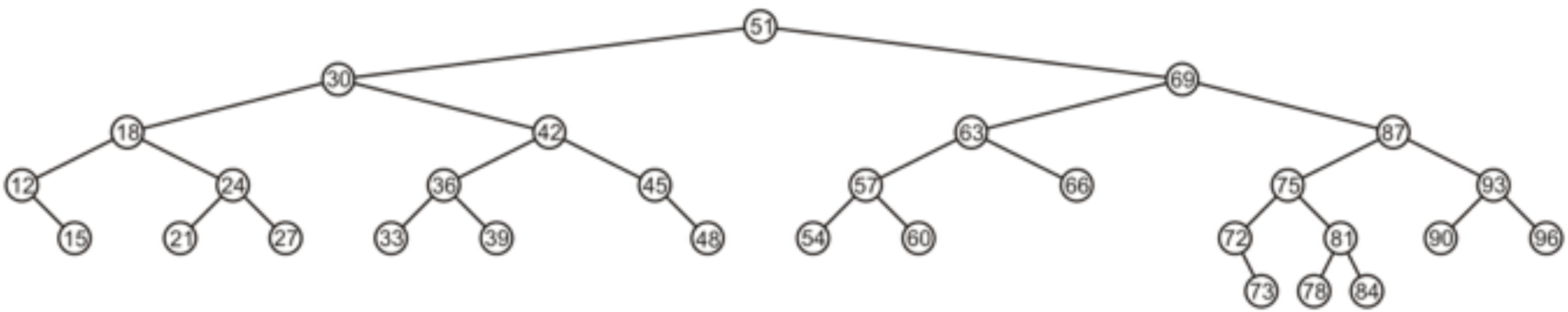
The node 81 is unbalanced

- A left-left imbalance
- Promote the intermediate node to the imbalanced node
- 75 is that node



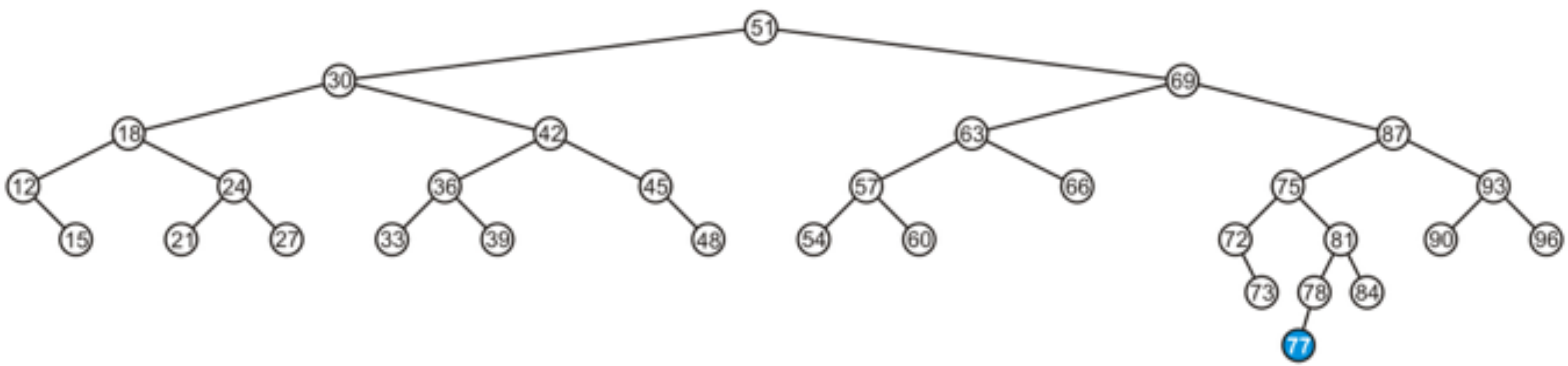
Insertion

The tree is AVL balanced



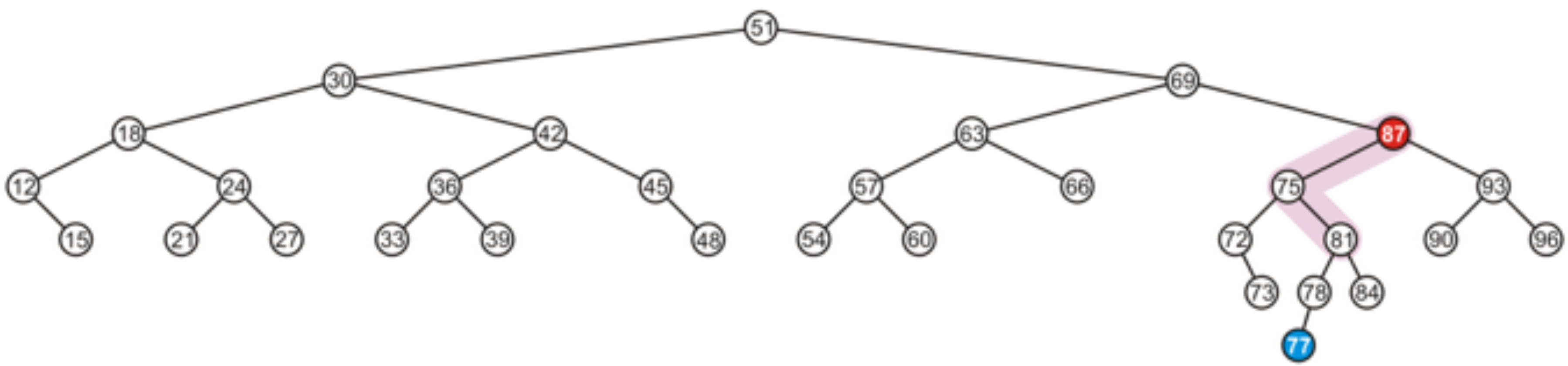
Insertion

Insert 77



Insertion

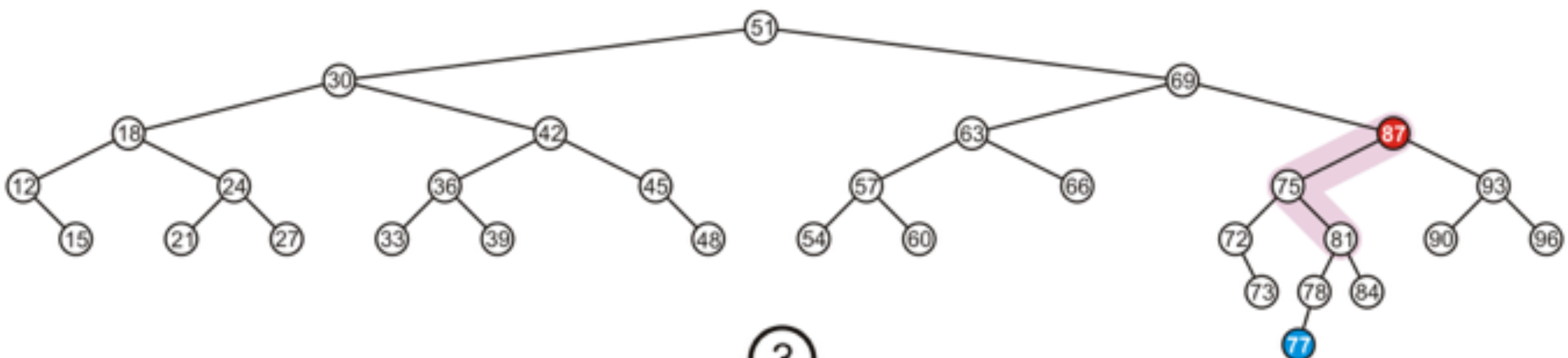
- The node 87 is unbalanced
  - A left-right imbalance

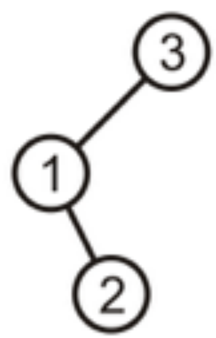


Insertion

The node 87 is unbalanced

- A left-right imbalance

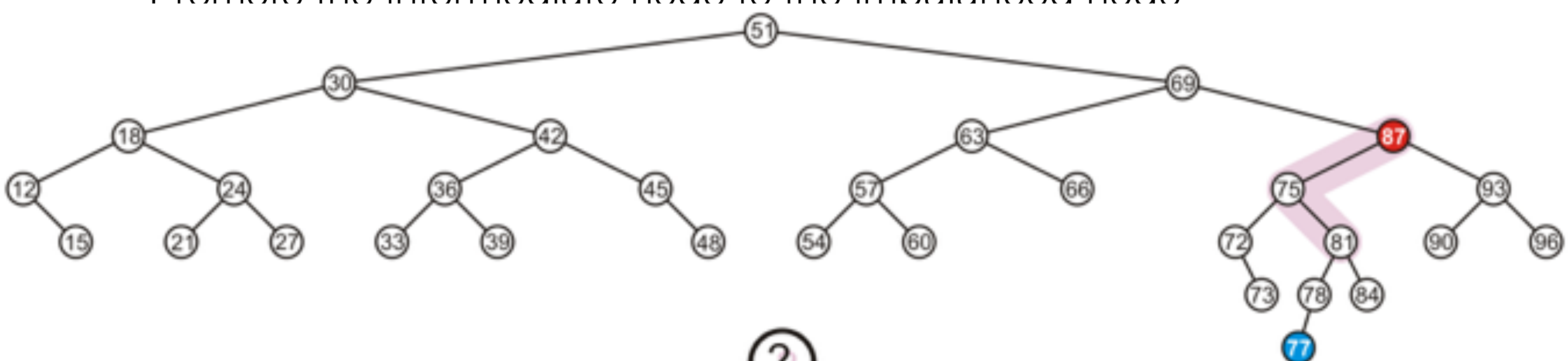


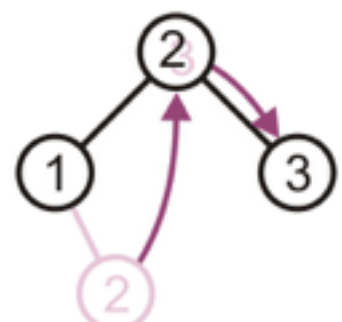


Insertion

The node 87 is unbalanced

- A left-right imbalance
- Promote the intermediate node to the imbalanced node

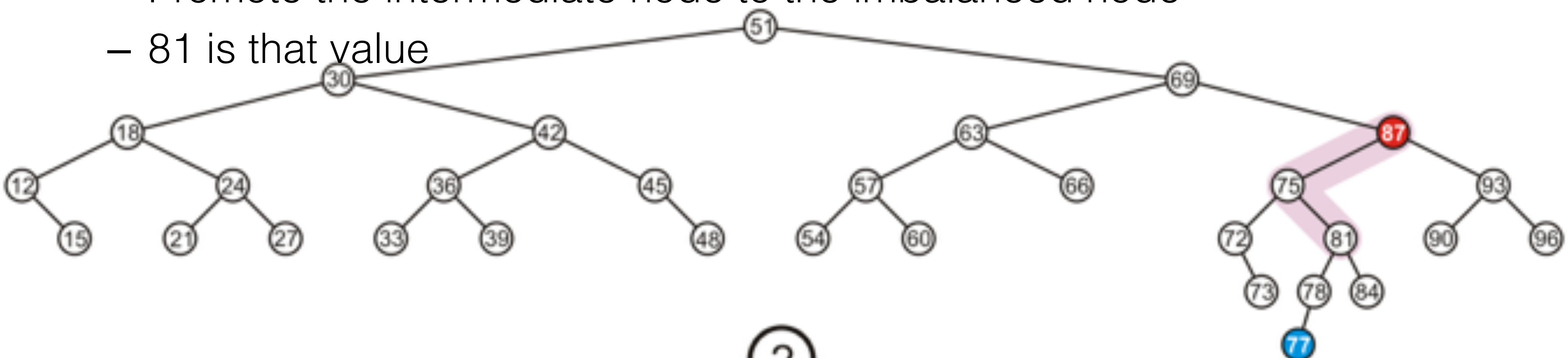


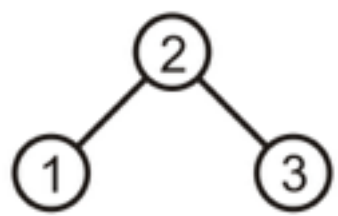


Insertion

The node 87 is unbalanced

- A left-right imbalance
- Promote the intermediate node to the imbalanced node
- 81 is that value

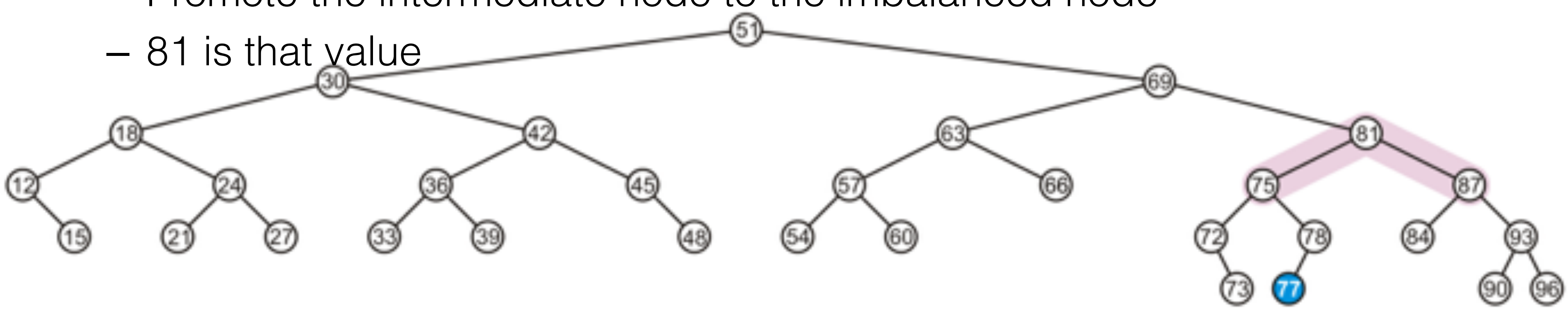




Insertion

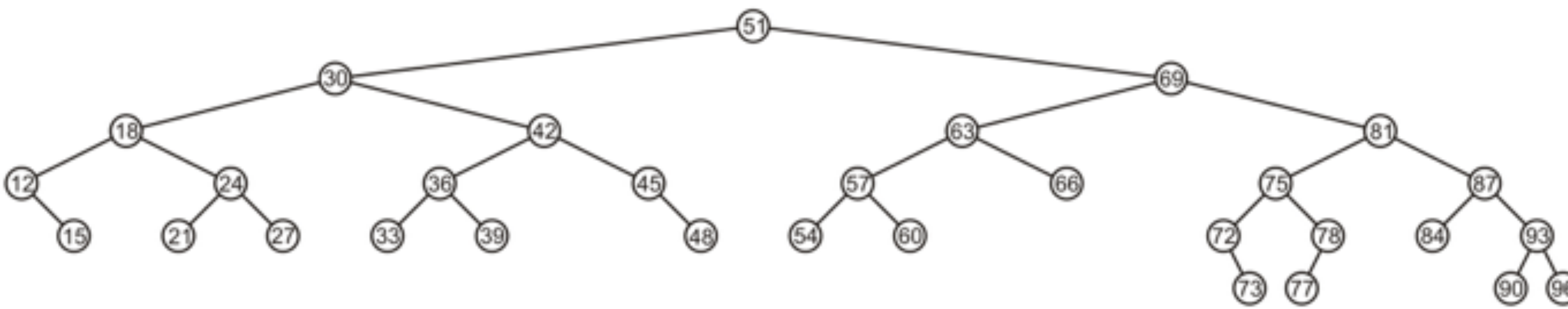
The node 87 is unbalanced

- A left-right imbalance
- Promote the intermediate node to the imbalanced node
- 81 is that value



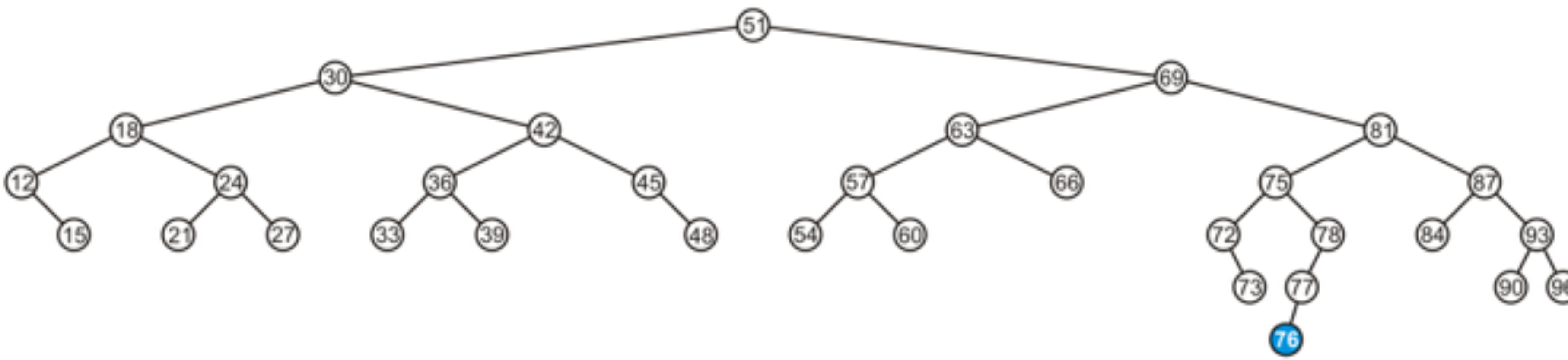
Insertion

The tree is balanced



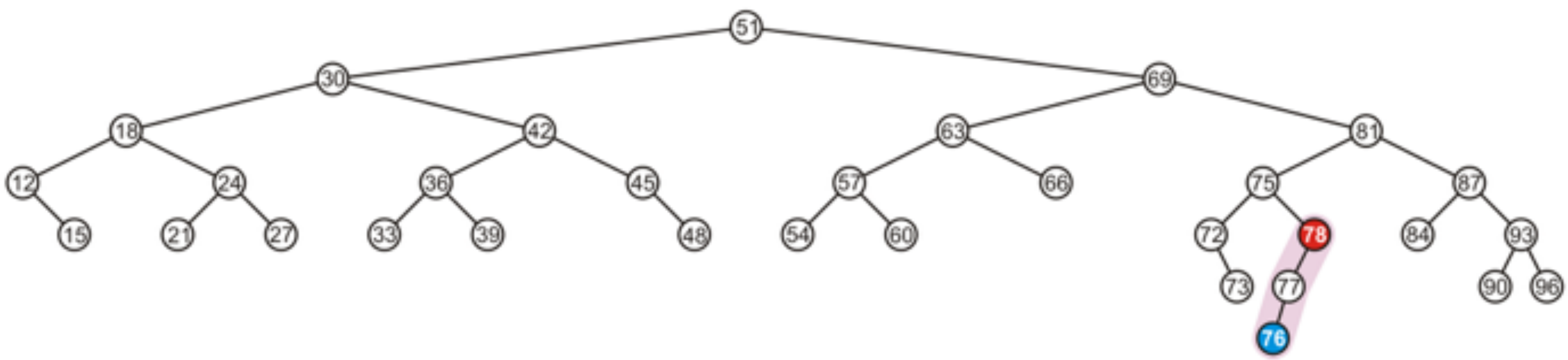
Insertion

Insert 76



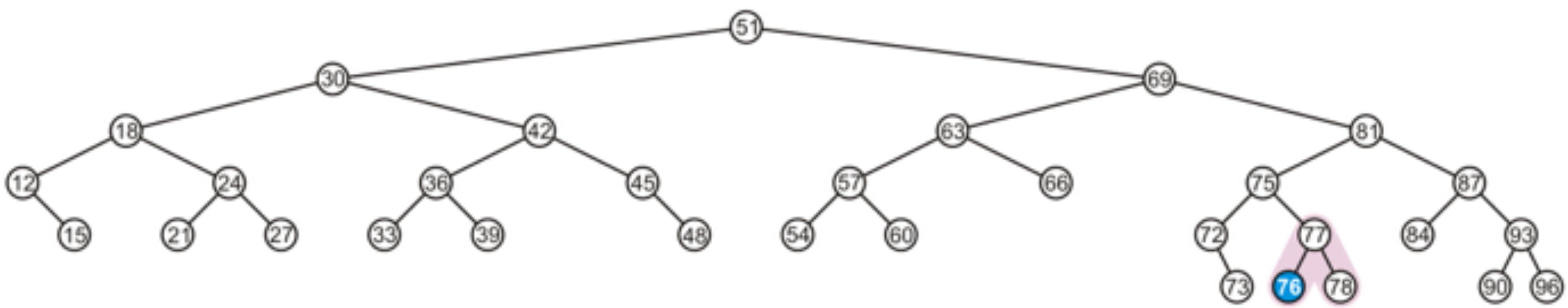
Insertion

- The node 78 is unbalanced
  - A left-left imbalance



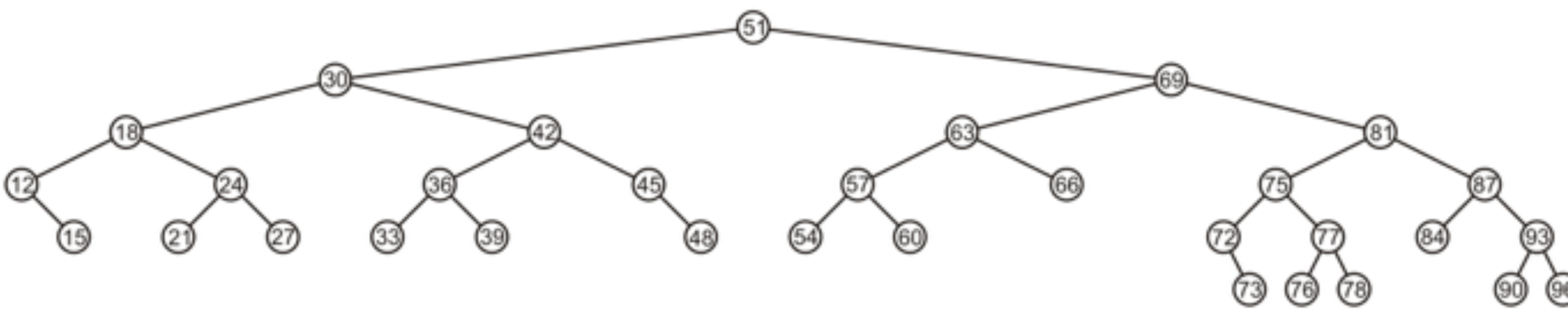
Insertion

- The node 78 is unbalanced
  - Promote 77



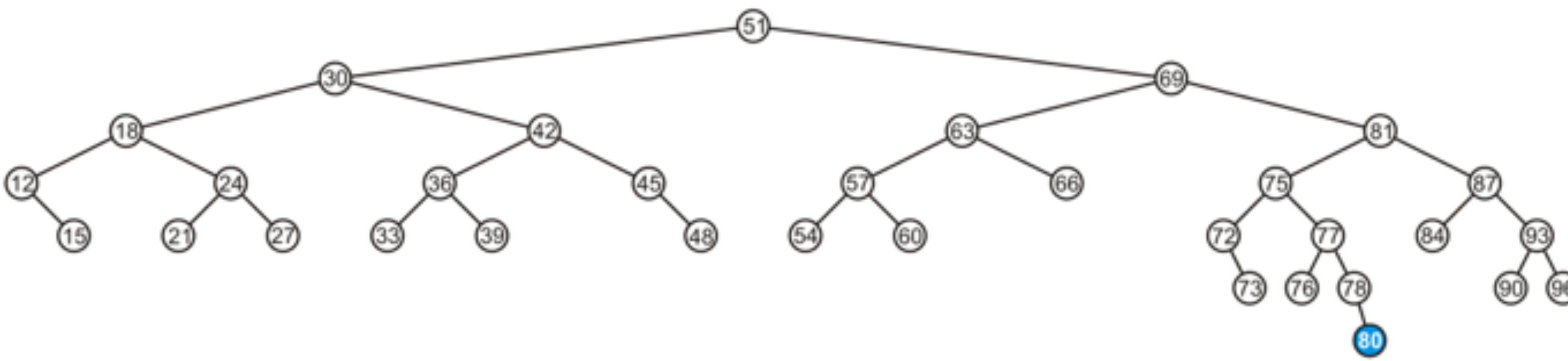
Insertion

Again, balanced



Insertion

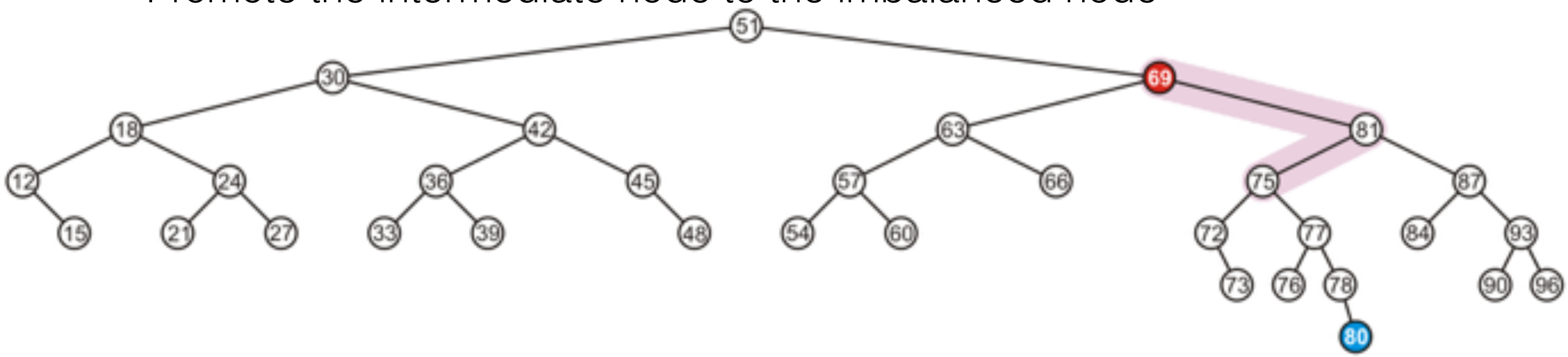
Insert 80



Insertion

The node 69 is unbalanced

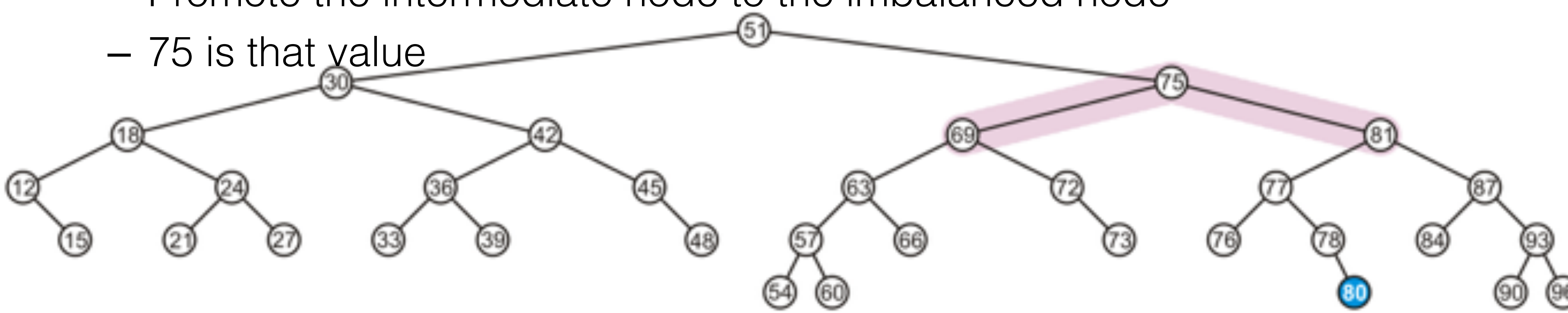
- A right-left imbalance
- Promote the intermediate node to the imbalanced node



Insertion

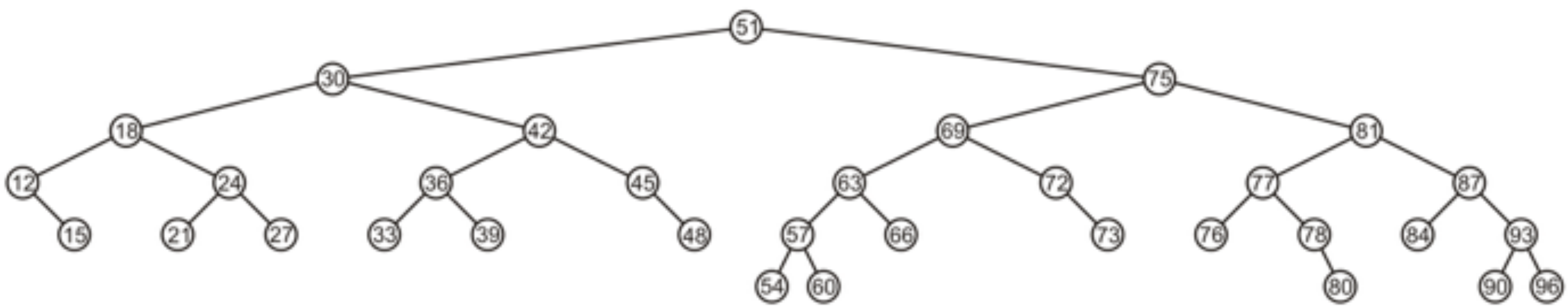
The node 69 is unbalanced

- A left-right imbalance
- Promote the intermediate node to the imbalanced node
- 75 is that value



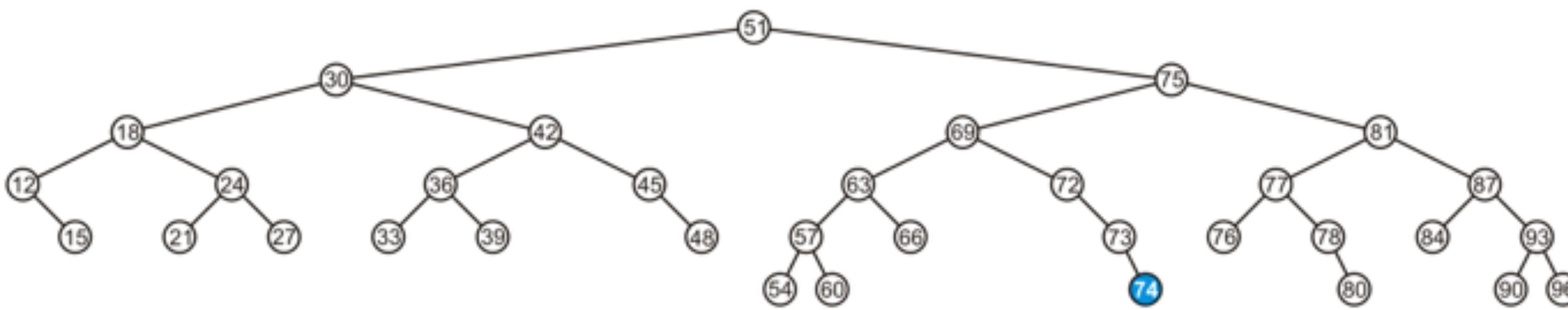
Insertion

Again, balanced



Insertion

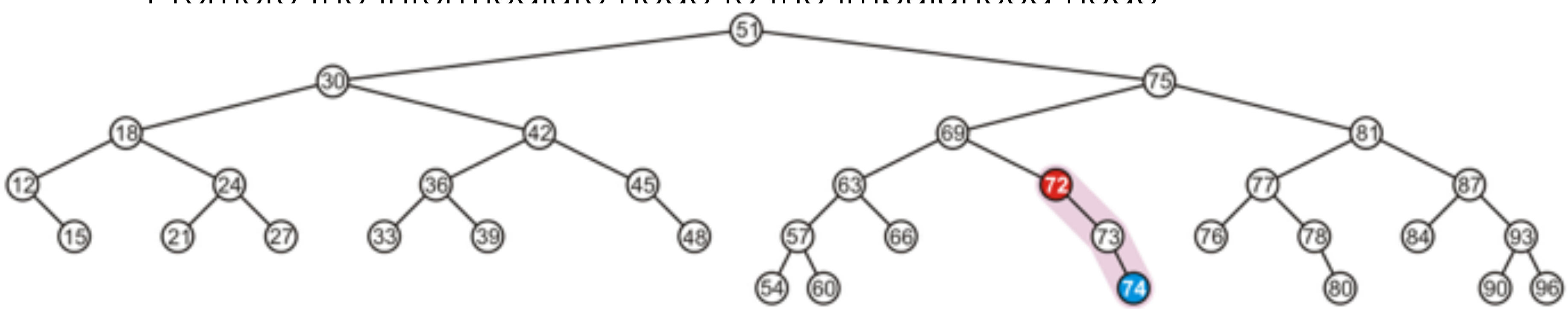
Insert 74



Insertion

The node 72 is unbalanced

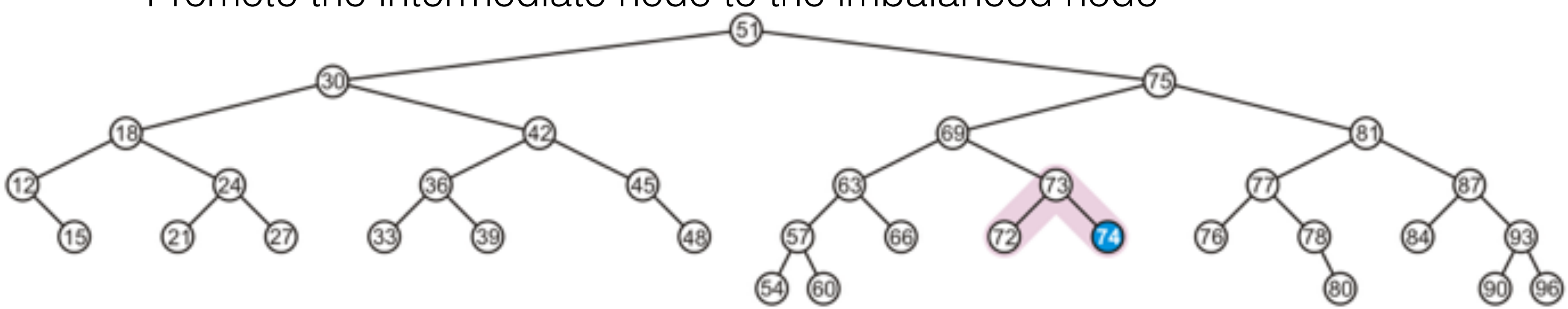
- A right-right imbalance
- Promote the intermediate node to the imbalanced node



Insertion

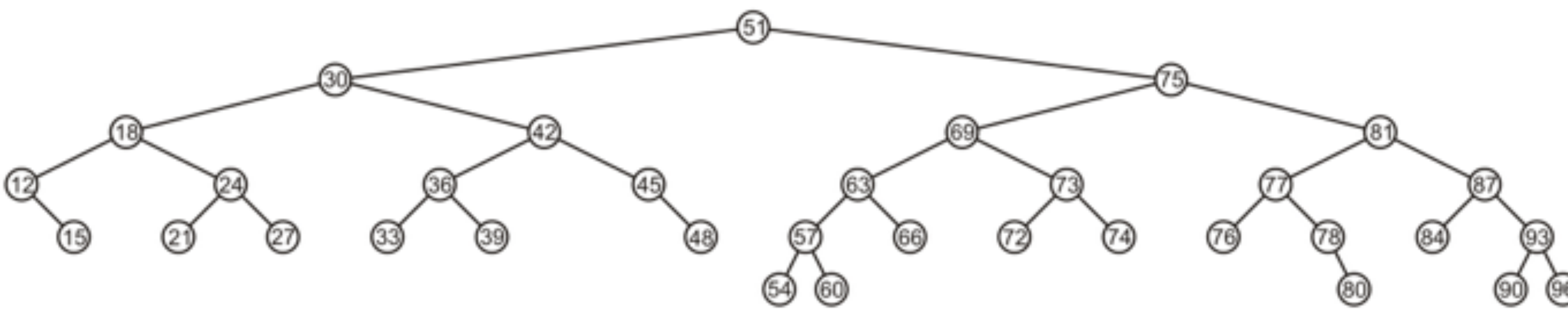
The node 72 is unbalanced

- A right-right imbalance
- Promote the intermediate node to the imbalanced node



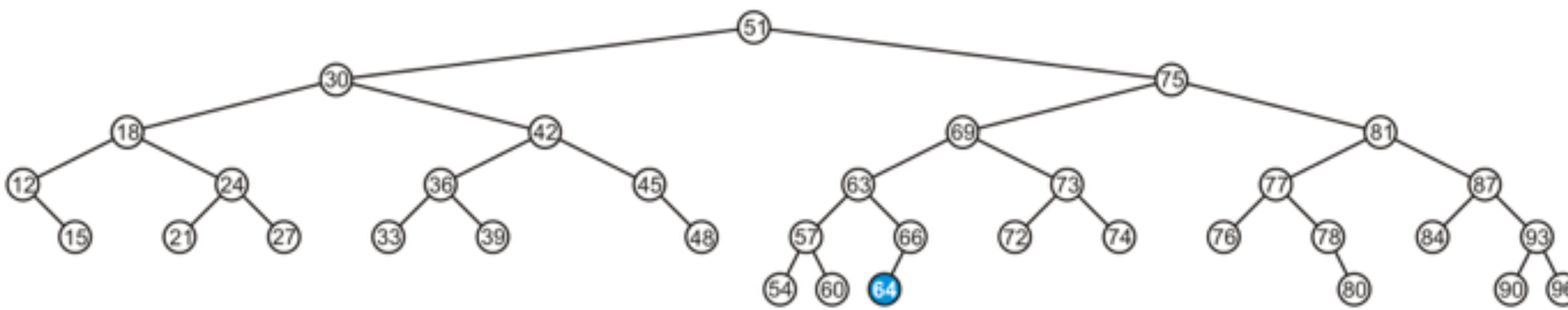
Insertion

Again, balanced



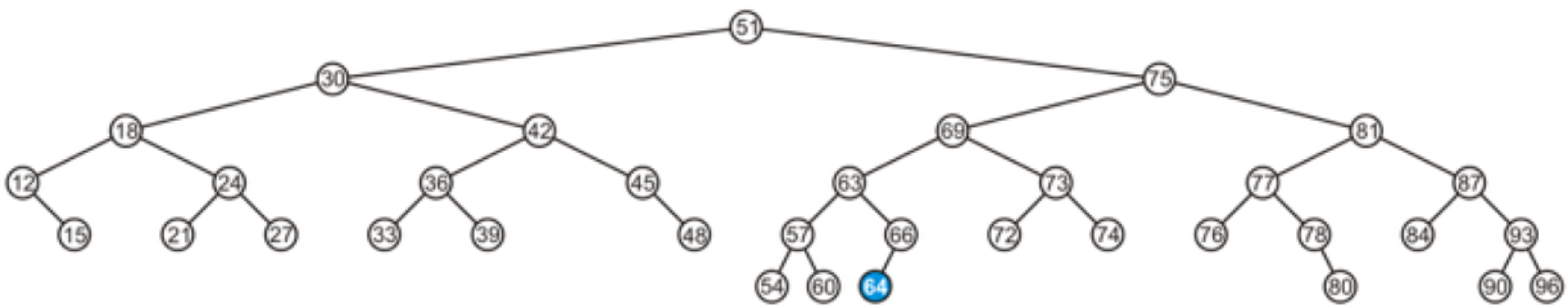
Insertion

Insert 64



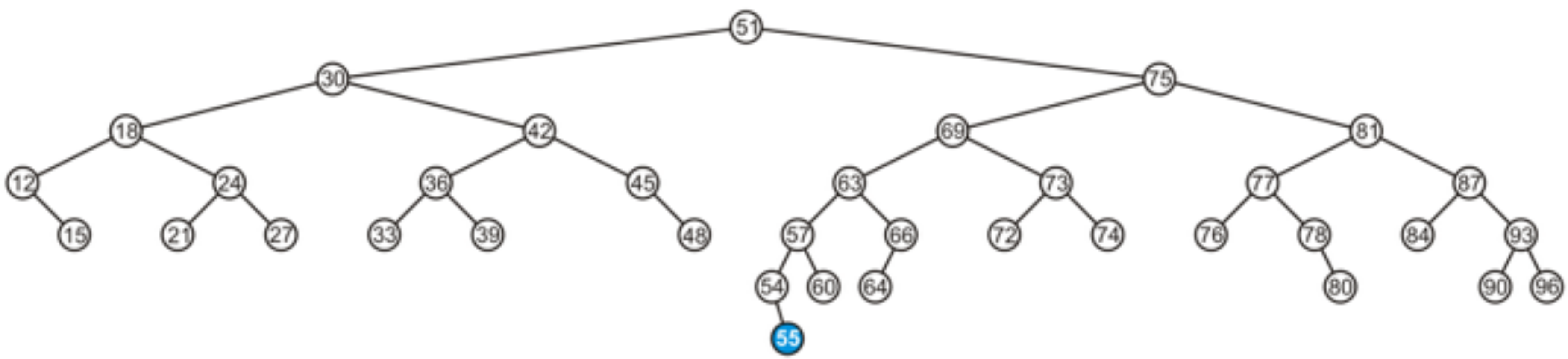
Insertion

This causes no imbalances



Insertion

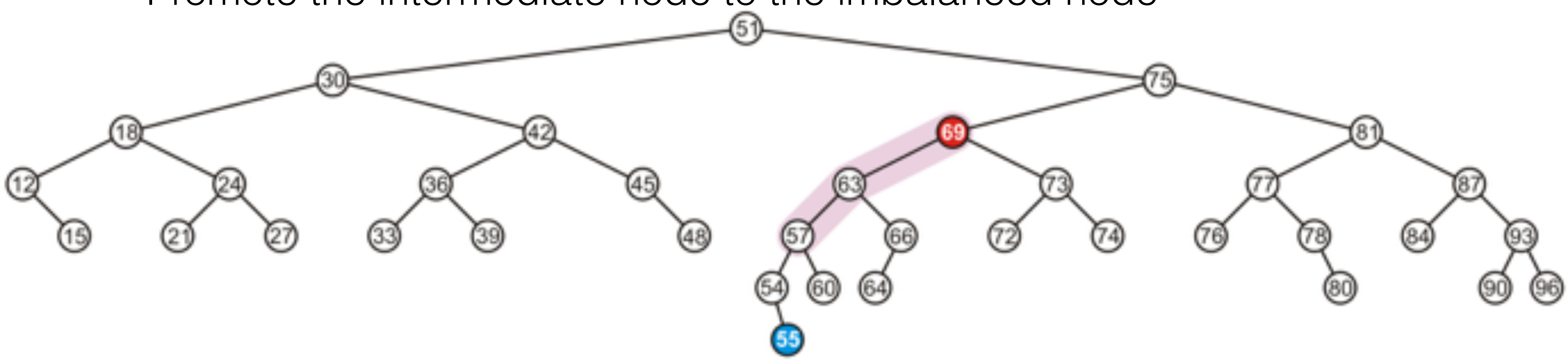
Insert 55



Insertion

The node 69 is imbalanced

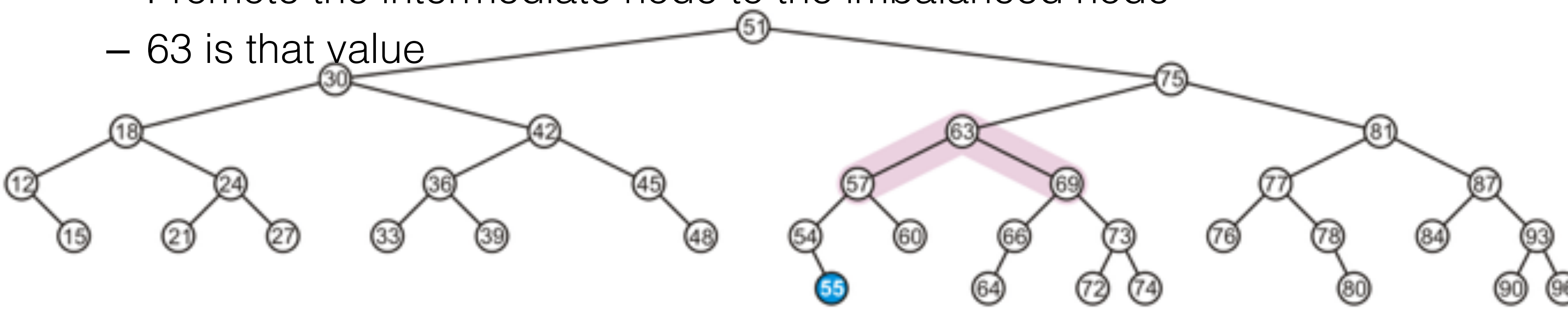
- A left-left imbalance
- Promote the intermediate node to the imbalanced node



Insertion

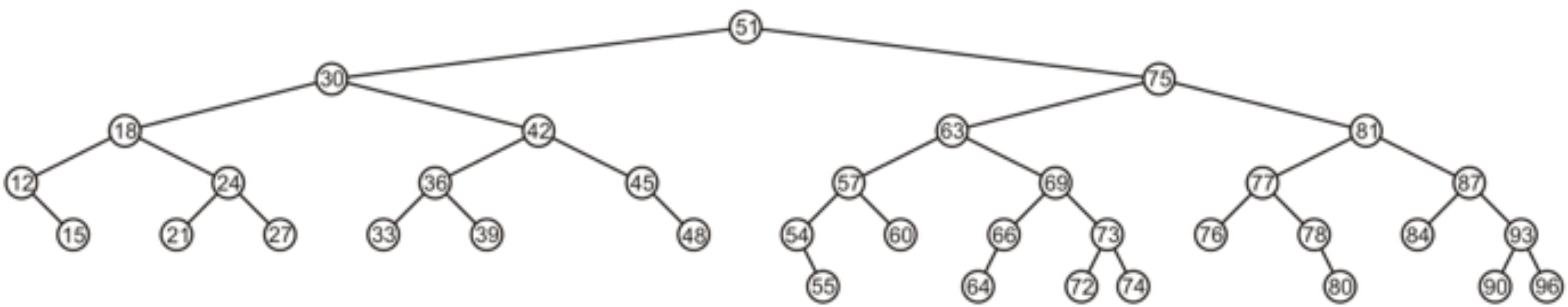
The node 69 is imbalanced

- A left-left imbalance
- Promote the intermediate node to the imbalanced node
- 63 is that value



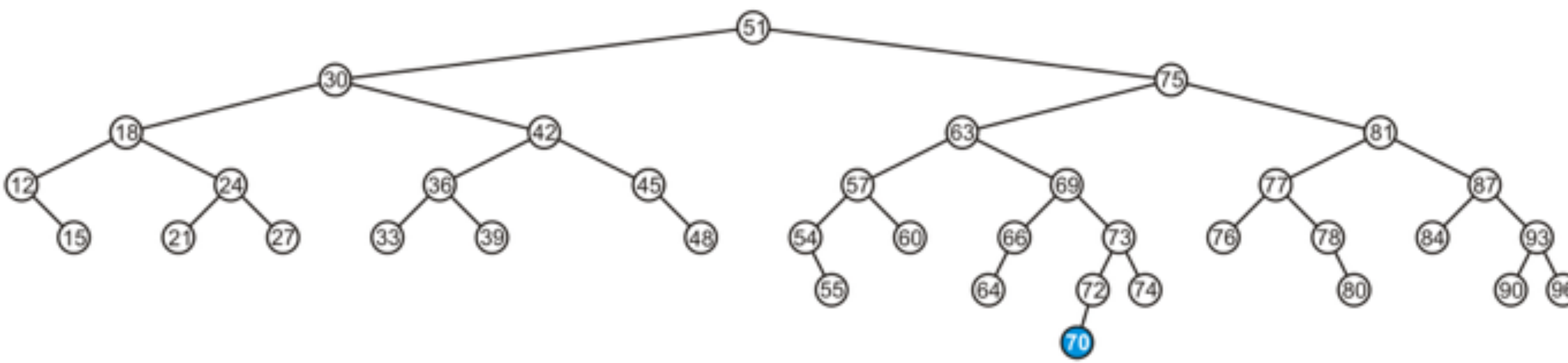
Insertion

The tree is now balanced



Insertion

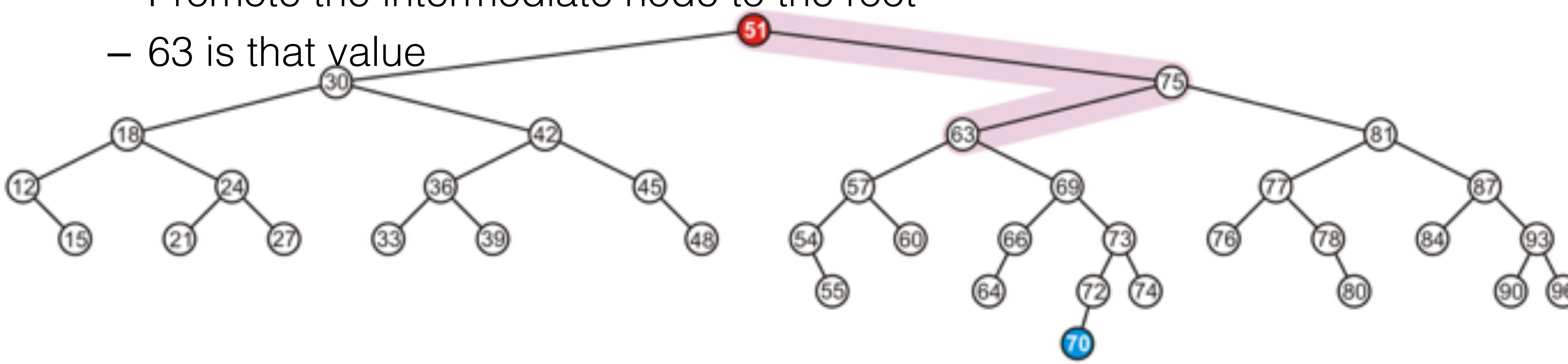
Insert 70



Insertion

The root node is now imbalanced

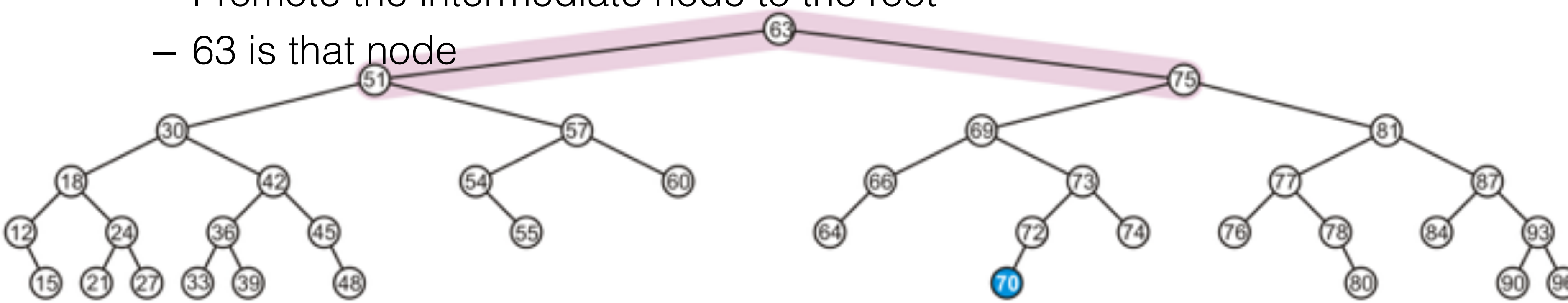
- A right-left imbalance
- Promote the intermediate node to the root
- 63 is that value



Insertion

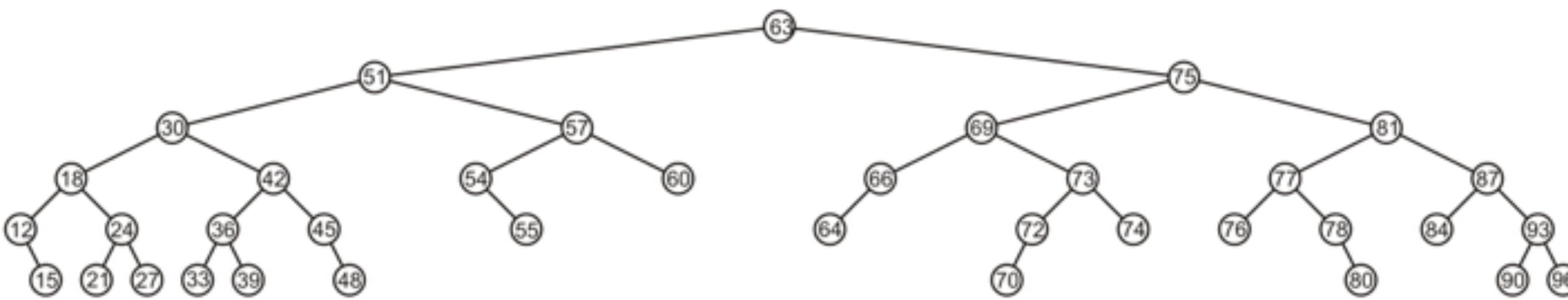
The root node is imbalanced

- A right-left imbalance
- Promote the intermediate node to the root
- 63 is that node



Insertion

The result is AVL balanced



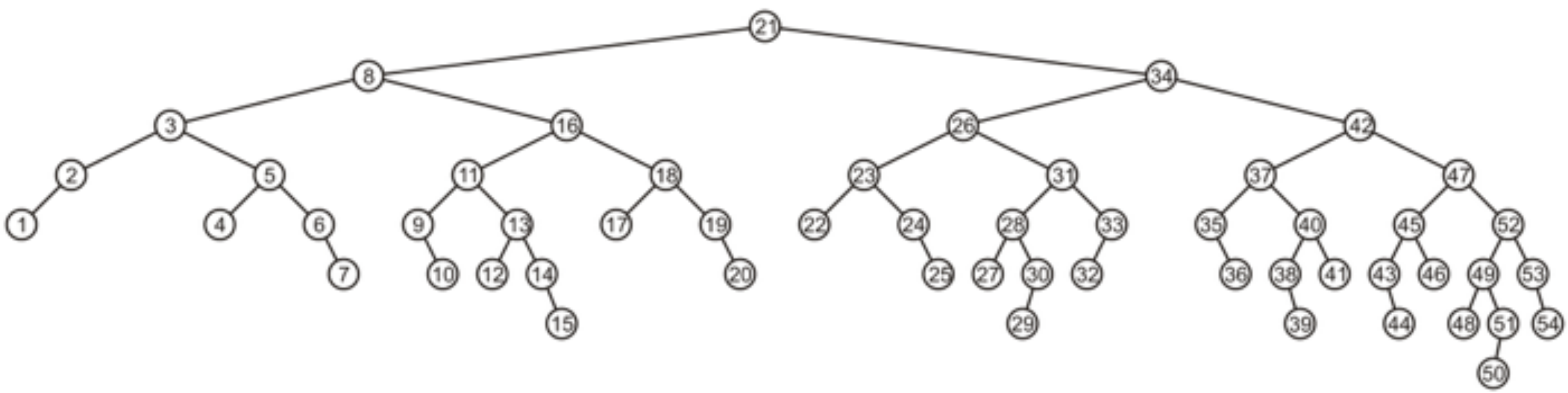
Erase

Removing a node from an AVL tree may cause more than one AVL imbalance

- Like insert, erase must check after it has been successfully called on a child to see if it caused an imbalance
- Unfortunately, it may cause $\mathrm{O}(h)$ imbalances that must be corrected
  - Insertions will only cause one imbalance that must be fixed

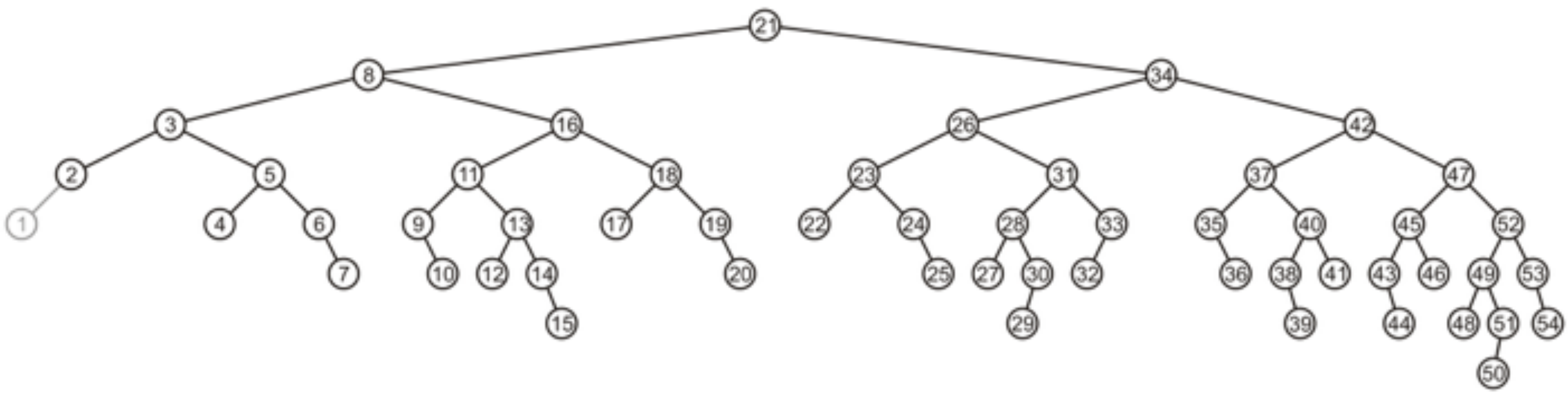
Erase

Consider the following AVL tree



Erase

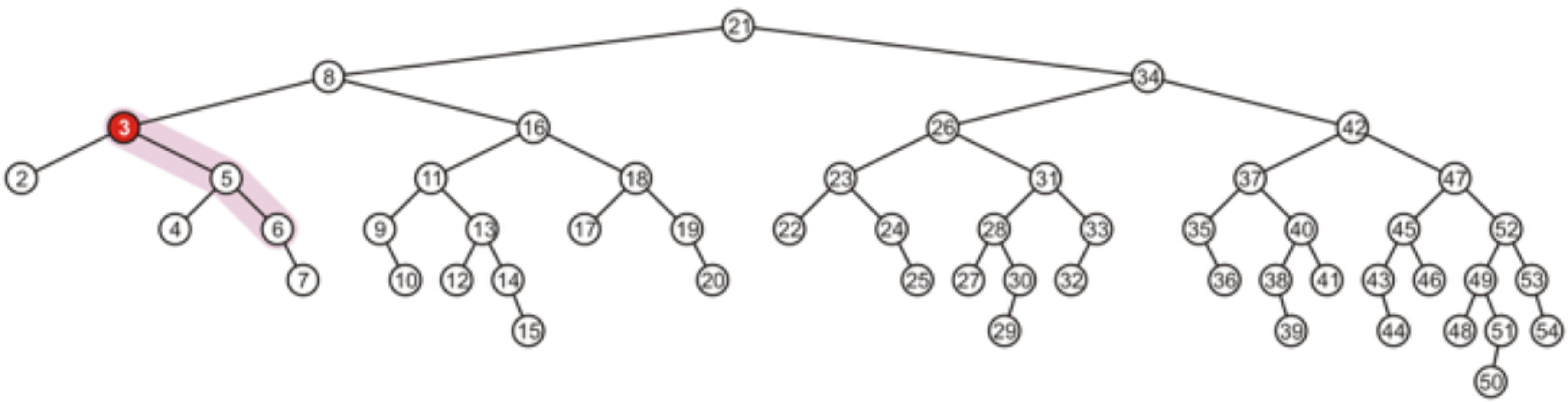
Suppose we erase the front node: 1



Erase

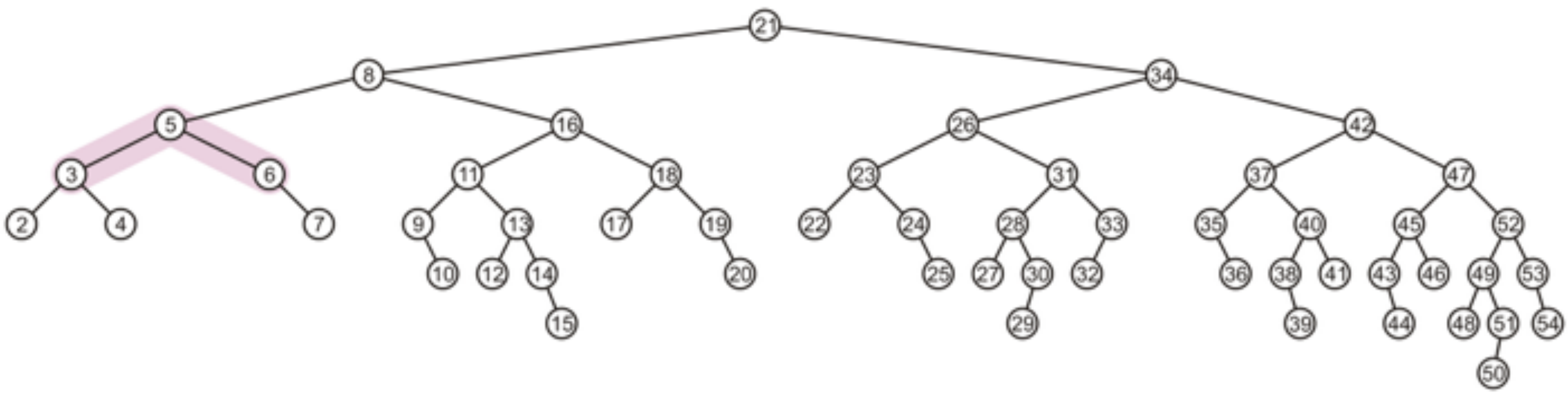
While its previous parent, 2, is not unbalanced, its grandparent 3 is

- The imbalance is in the right-right subtree



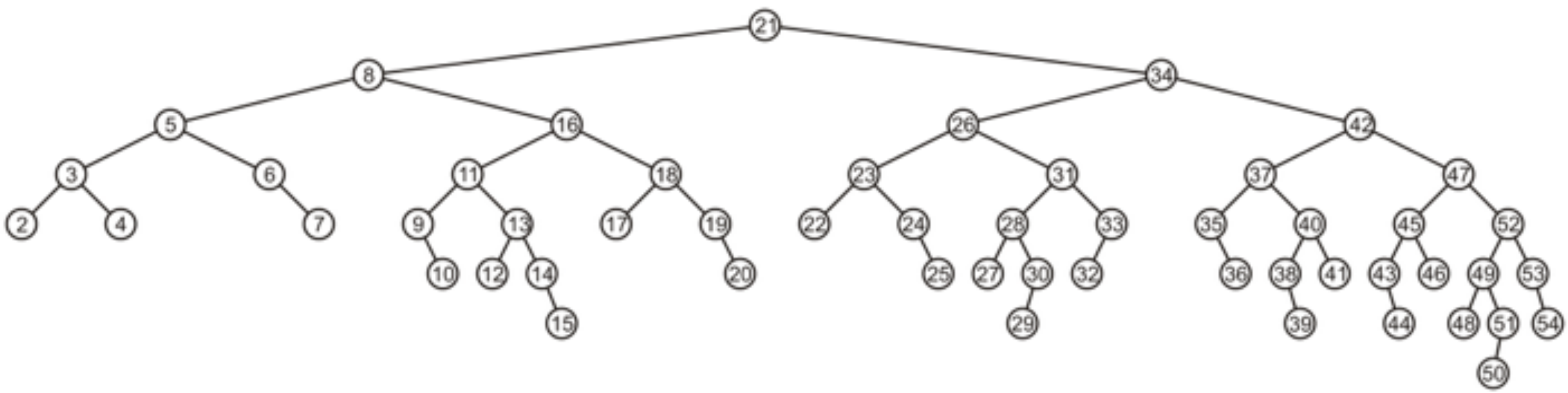
Erase

We can correct this with a simple balance



Erase

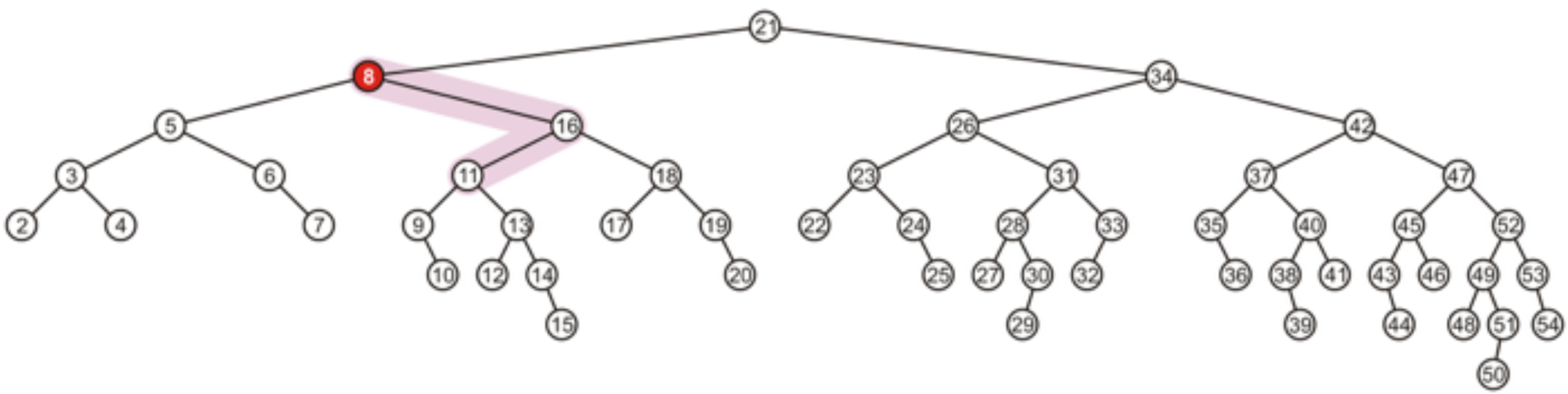
The node of that subtree, 5, is now balanced



Erase

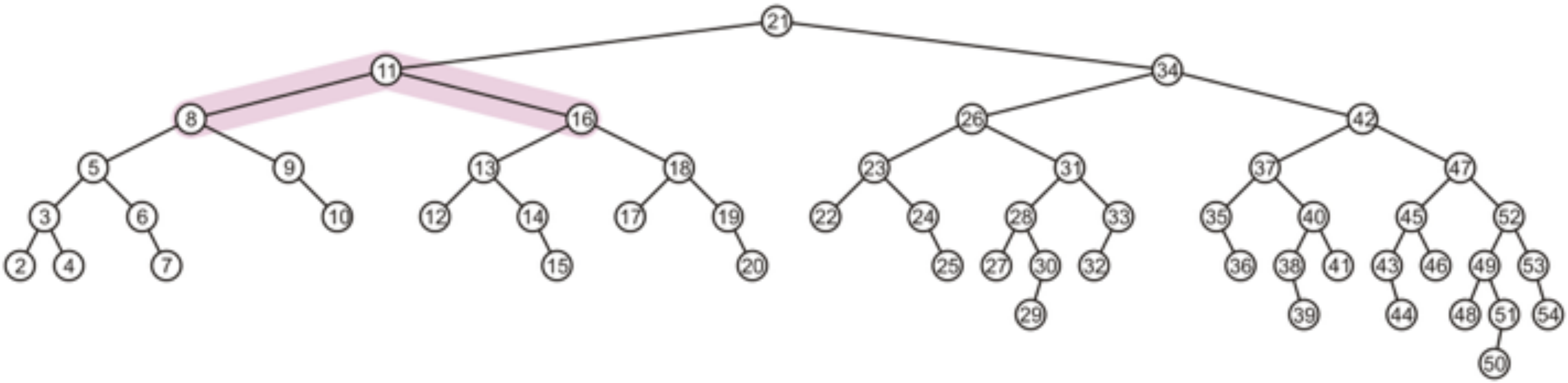
Recursing to the root, however, 8 is also unbalanced

- This is a right-left imbalance



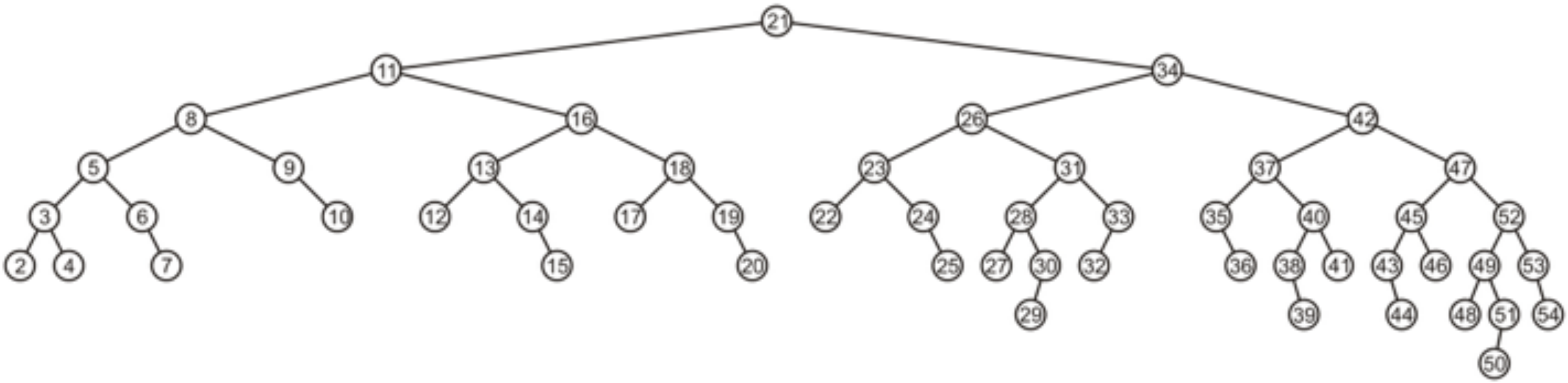
Erase

Promoting 11 to the root corrects the imbalance



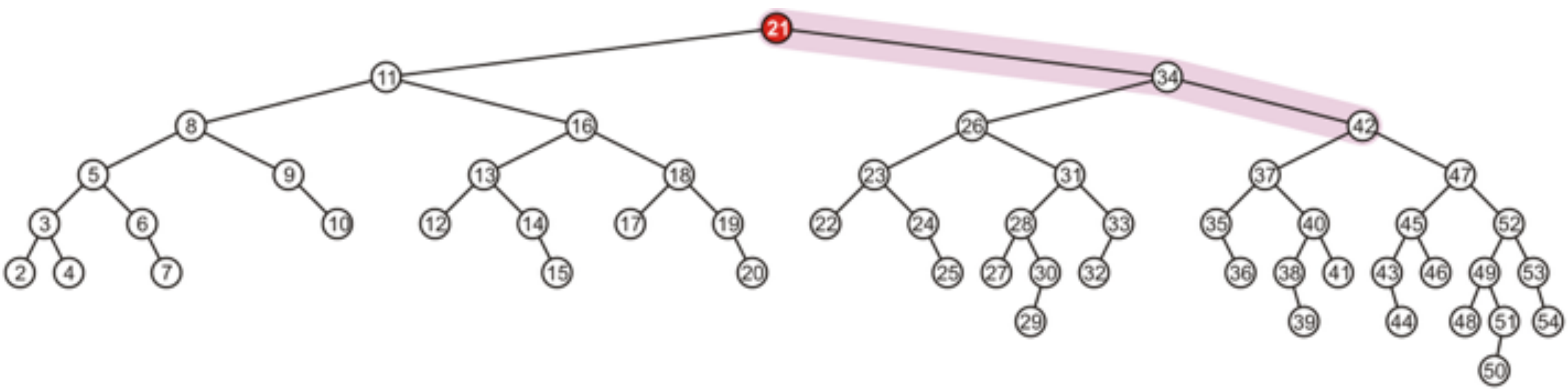
Erase

At this point, the node 11 is balanced



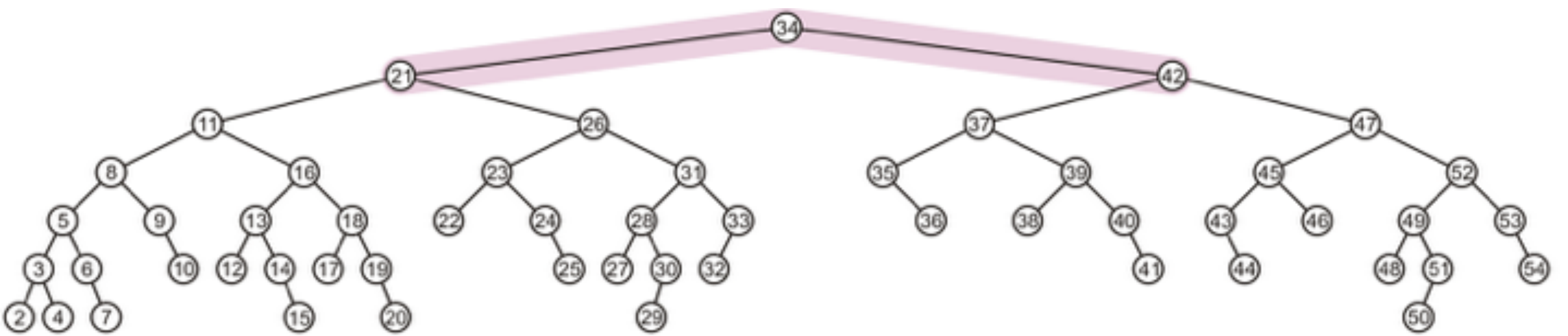
Erase

- Still, the root node is unbalanced
  - This is a right-right imbalance



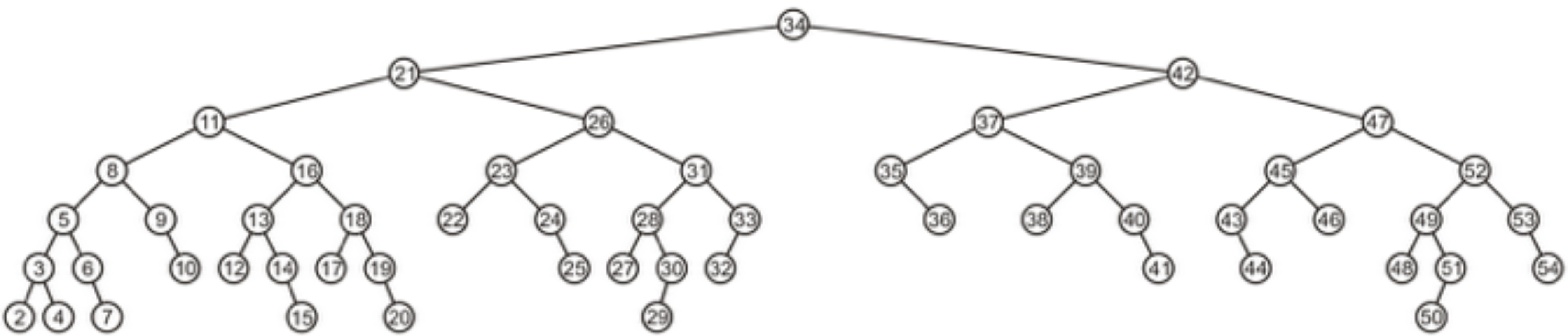
Erase

Again, a simple balance fixes the imbalance



Erase

- The resulting tree is now AVL balanced
  - Note, few erases will require one balance, even fewer will require more than one



Summary

In this topic we have covered:

- AVL balance is defined by ensuring the difference in heights is 0 or 1
- Insertions and erases are like binary search trees
- Each insertion requires at least one correction to maintain AVL balance
- Erases may require $\mathrm{O}(h)$ corrections
- These corrections require $\Theta(1)$ time
- Depth is $\Theta(\ \ln(n)\ )$

  $\therefore$ all $\mathbf{O}(h)$ operations are $\mathbf{O}(\ \ln(n)\ )$