AVL Trees

Background

From previous lectures:

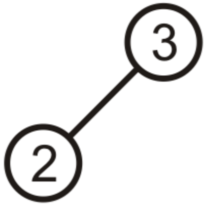
- Binary search trees store linearly ordered data
- Best case height: $\Theta(\ln(n))$
- Worst case height: $\mathbf{O}(n)$

Requirement:

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

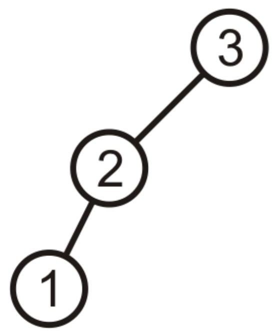
Prototypical Examples

These two examples demonstrate how we can correct for imbalances: starting with this tree, add 1:



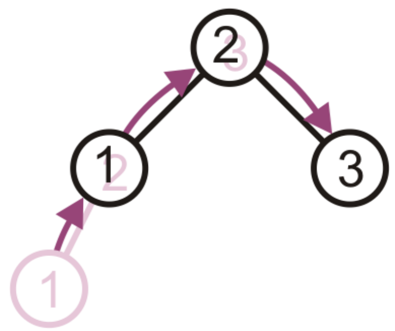
Prototypical Examples

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



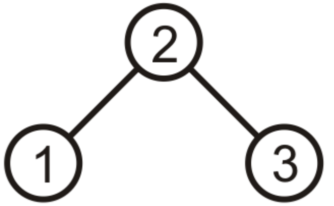
Prototypical Examples

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



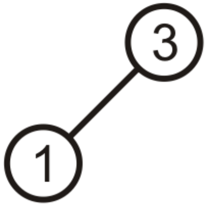
Prototypical Examples

The result is a perfect, though trivial tree



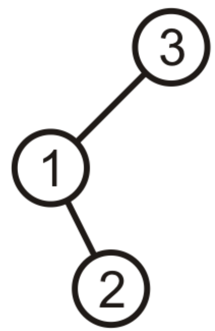
Prototypical Examples

Alternatively, given this tree, insert 2



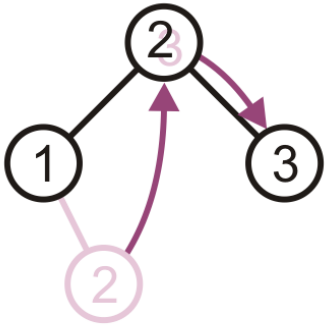
Prototypical Examples

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



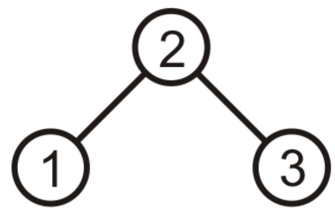
Prototypical Examples

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

Named after Adelson-Velskii and Landis

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

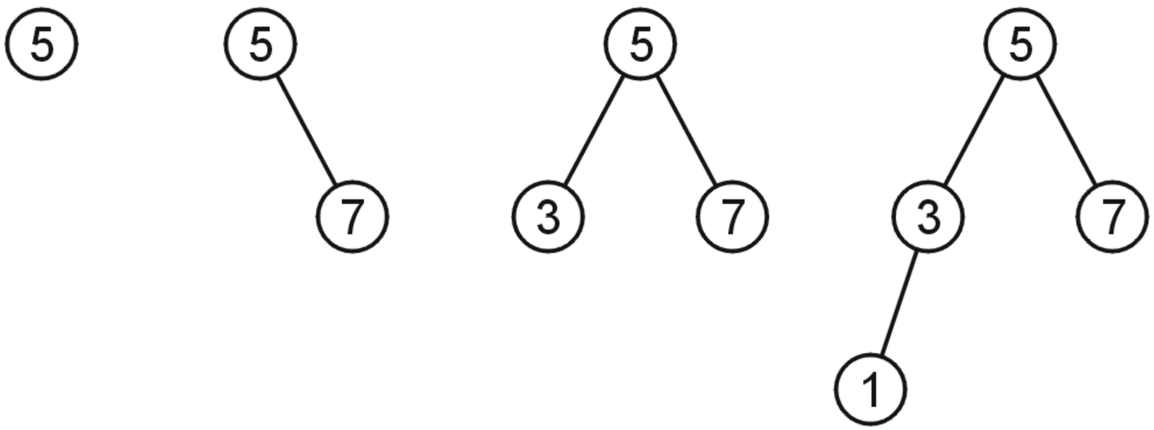
- The difference in the heights between the left and right sub-trees is at most 1, and
- Both sub-trees are themselves AVL trees

Recall:

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

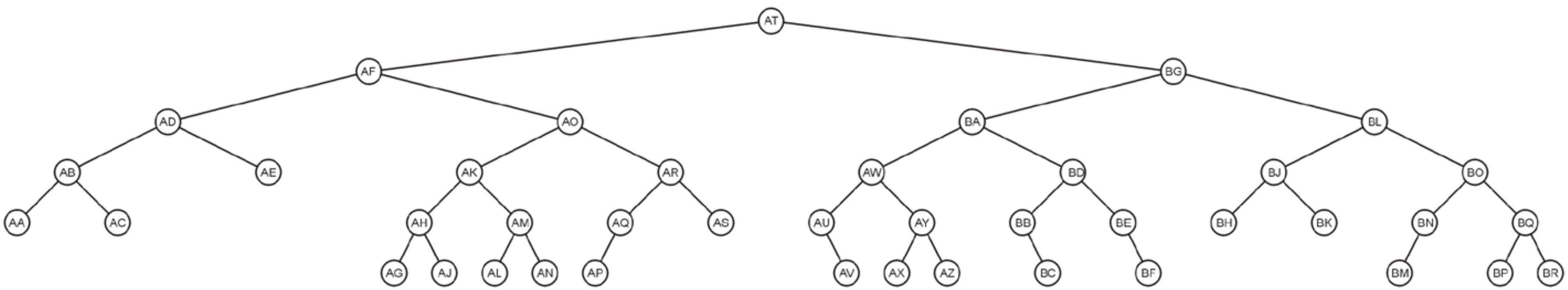
AVL Trees

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



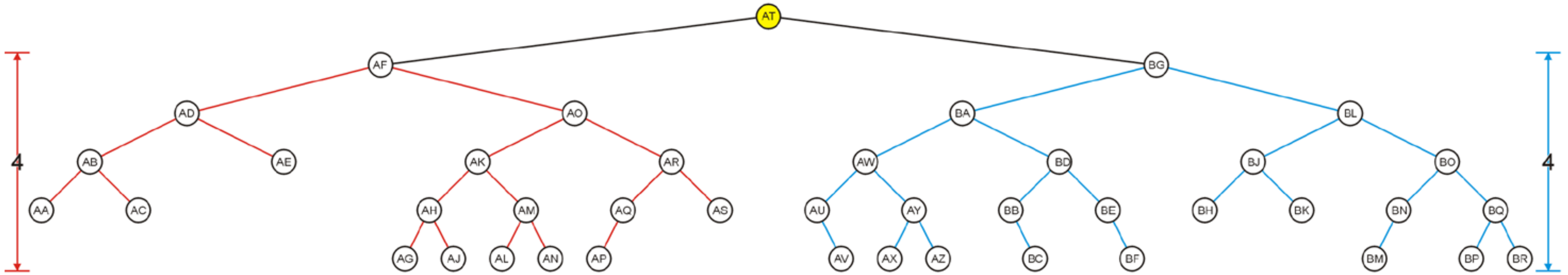
AVL Trees

Here is a larger AVL tree (42 nodes):



AVL Trees

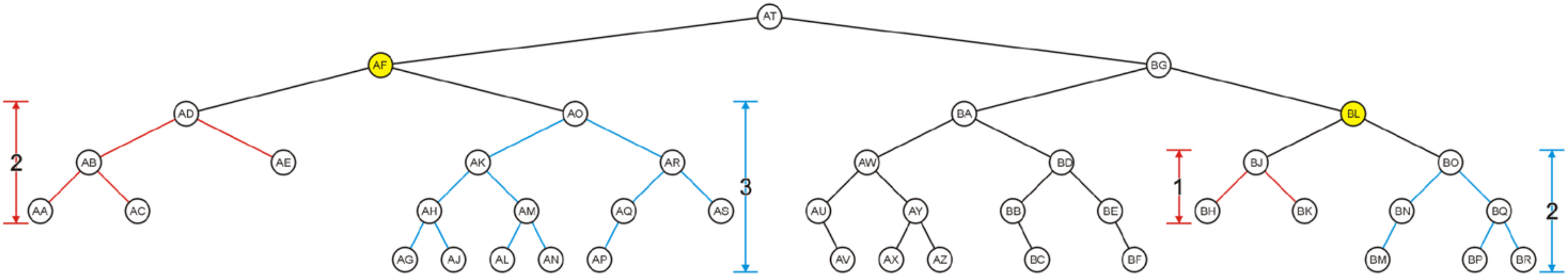
The root node is AVL-balanced:
- Both sub-trees are of height 4:



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

What is the lower bound?

Height of an AVL Tree

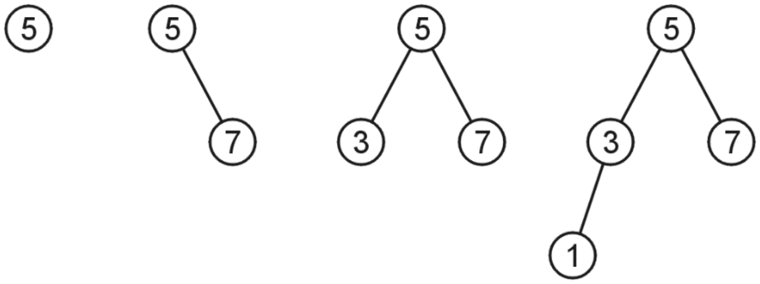
Let $\mathrm{F}(h)$ be the fewest number of nodes in a tree of height h

From a previous slide:

$\mathrm{F}(0) = 1$

$\mathrm{F}(1) = 2$

$\mathrm{F}(2) = 4$



Can we find $\mathrm{F}(h)$?

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 recurrence 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:

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

Height of an AVL Tree

This is approximately

$$\mathrm{F}(h) \approx 1.8944\ \phi^h - 1$$

where $\phi \approx 1.6180$ is the golden ratio

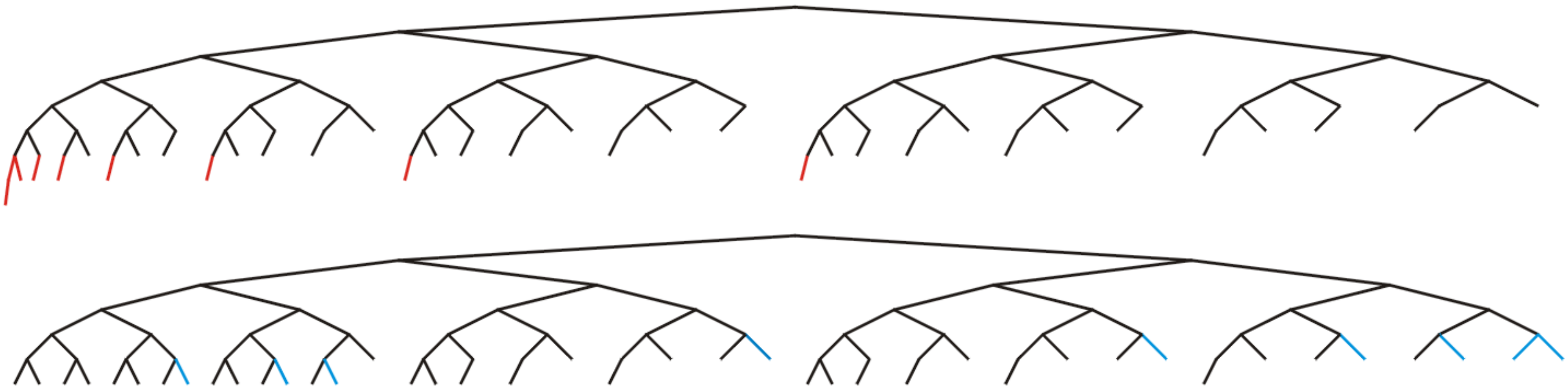
– That is, $\mathrm{F}(h) = \Theta(\phi^h)$

Thus, we may find the maximum value of h for a given n:

$$\log_\phi\left(\frac{n+1}{1.8944}\right) = \log_\phi(n+1) - 1.3277 = 1.4404 \cdot \lg(n+1) - 1.3277$$

Height of an AVL Tree

In this example, $n = 88$, the worst- and best-case scenarios differ in height by only 2



Height of an AVL Tree

If $n = 10^6$, the bounds on h are:

- The minimum height: $\log_2(10^6) - 1 \approx 19$
- the maximum height : $\log_\phi(10^6 / 1.8944) < 28$

Maintaining 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

This may cause some nodes to be unbalanced. We may need to rebalance the tree after insertion or removal.

Maintaining Balance

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

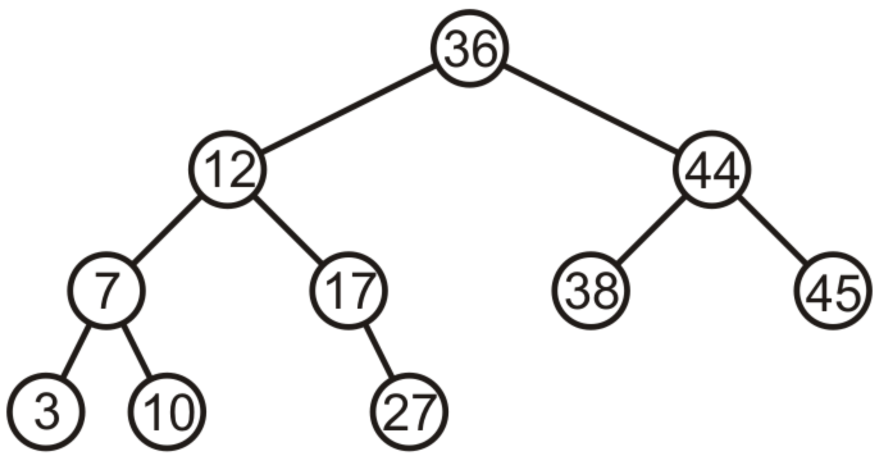
- Our implementation of height is $\Theta(n)$
- Introduce a member variable:

```
int tree_height;
```

- This variable is updated during inserting and erasing

Maintaining Balance

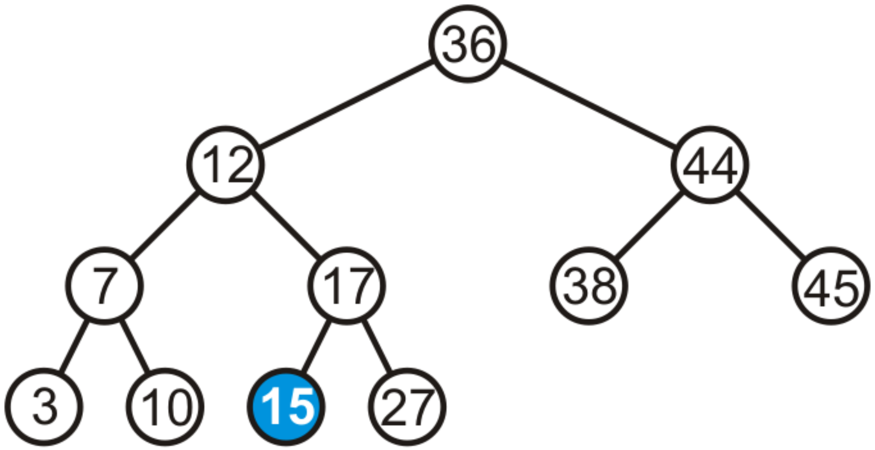
Consider this AVL tree



Maintaining Balance

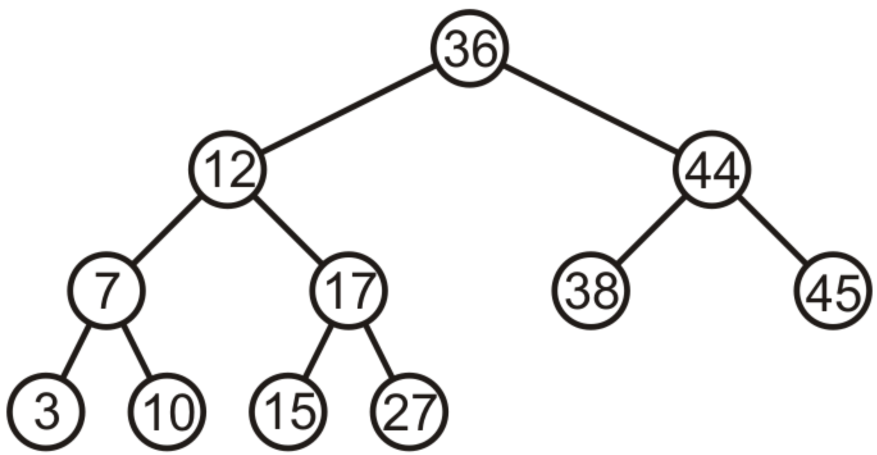
Consider inserting 15 into this tree

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



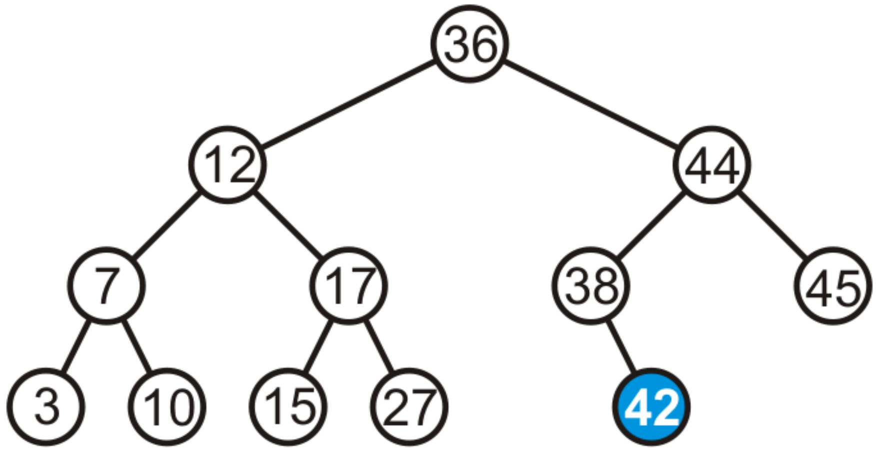
Maintaining Balance

The tree remains balanced



Maintaining Balance

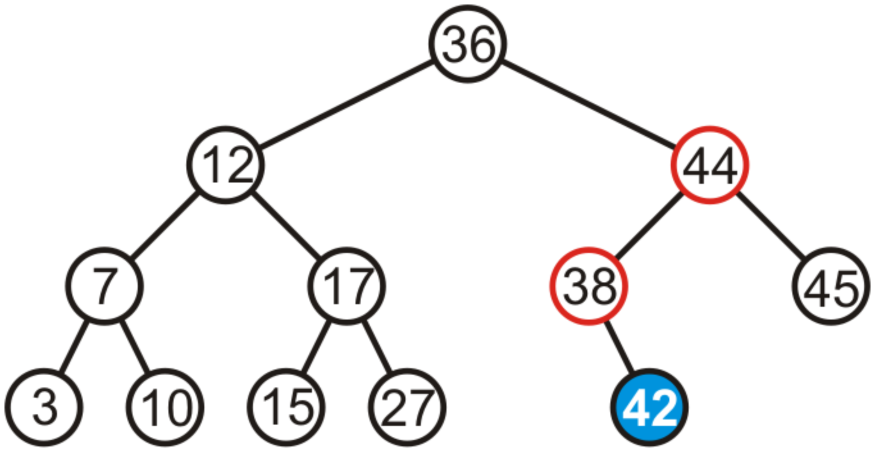
Consider inserting 42 into this tree



Maintaining Balance

Consider inserting 42 into this tree

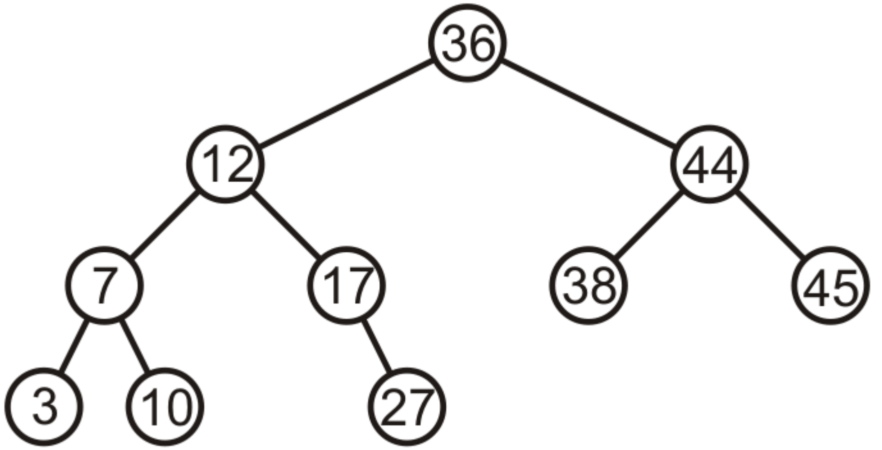
- Now we see the heights of two sub-trees have increased by one
- The tree is still balanced



Maintaining Balance

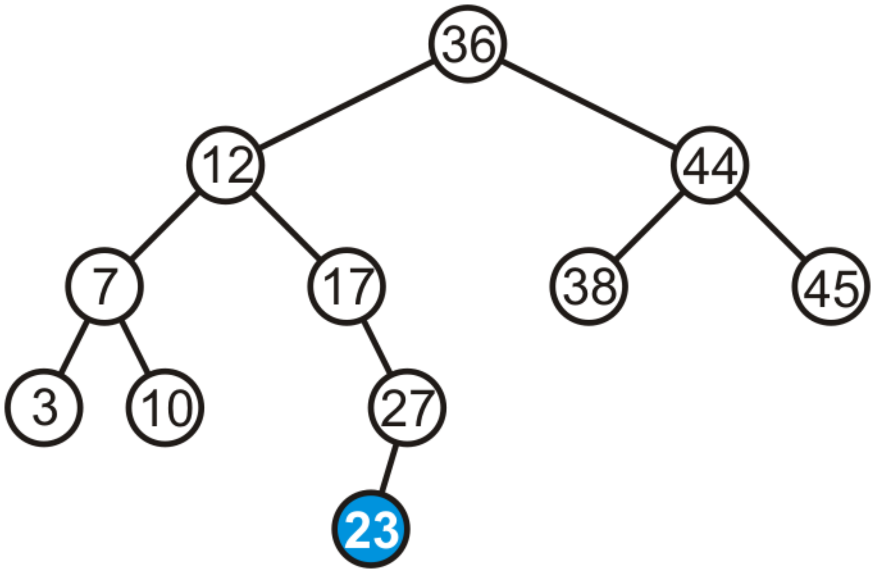
If a tree is AVL balanced, for an insertion to cause an imbalance:

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



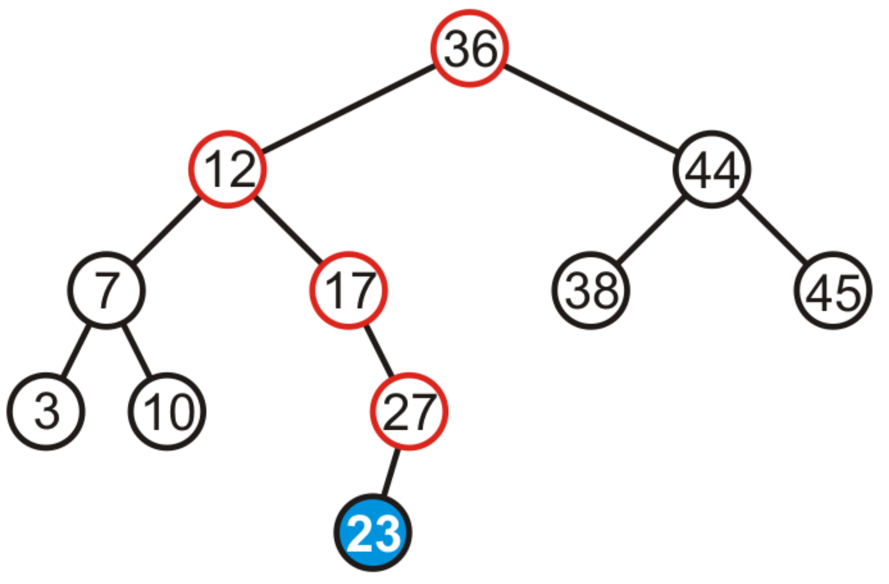
Maintaining Balance

Suppose we insert 23 into our initial tree



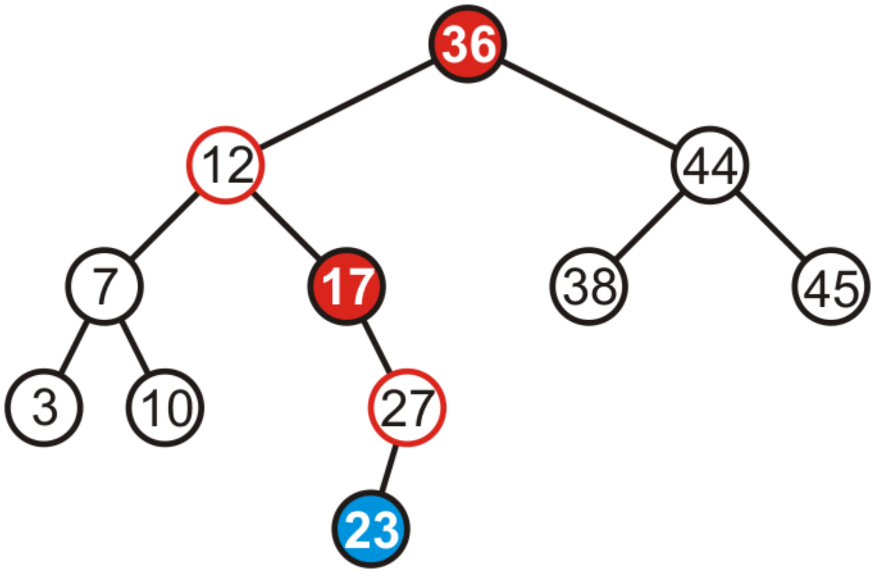
Maintaining Balance

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



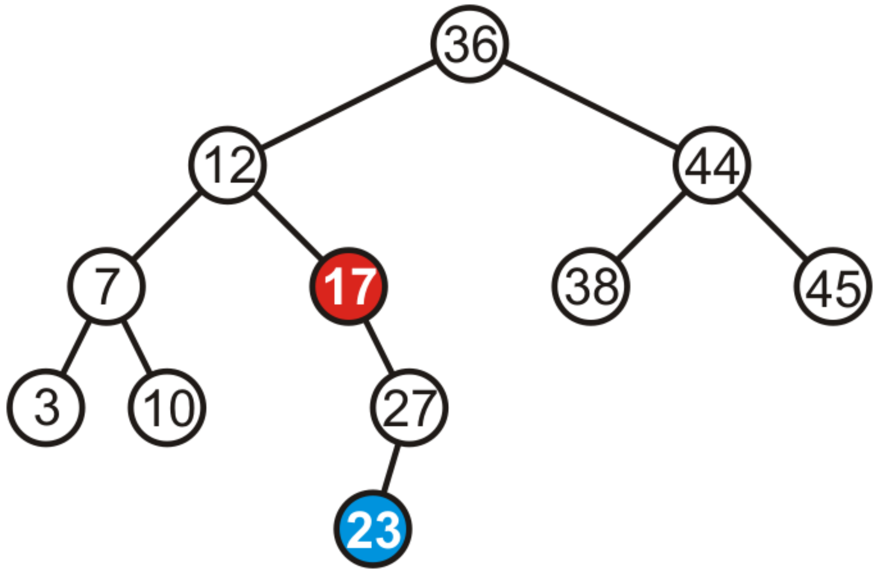
Maintaining Balance

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



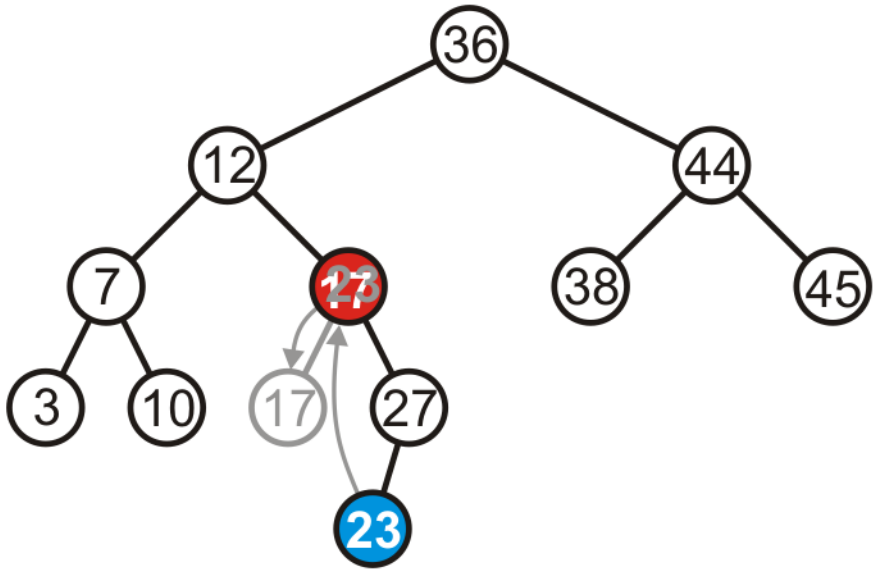
Maintaining Balance

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



Maintaining Balance

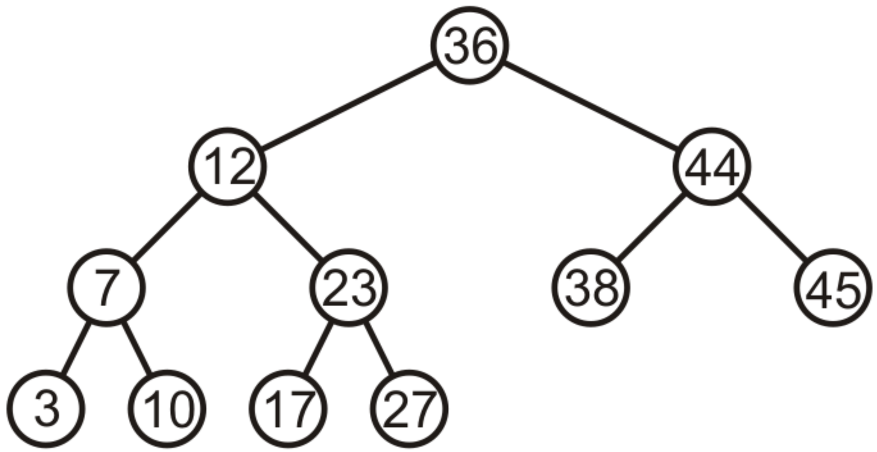
We can promote 23 to where 17 is, and make 17 the left child of 23



Maintaining Balance

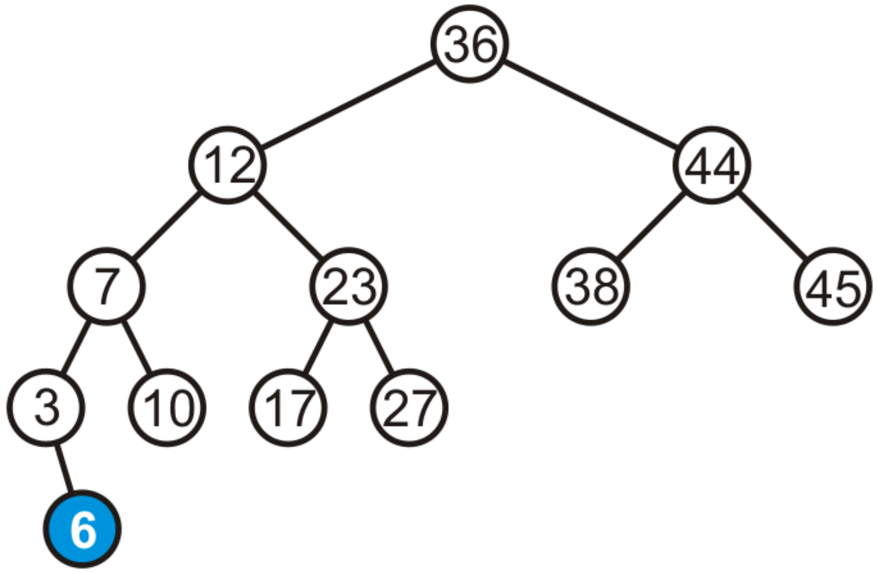
Thus, that node is no longer unbalanced

– Incidentally, the root is now also balanced!



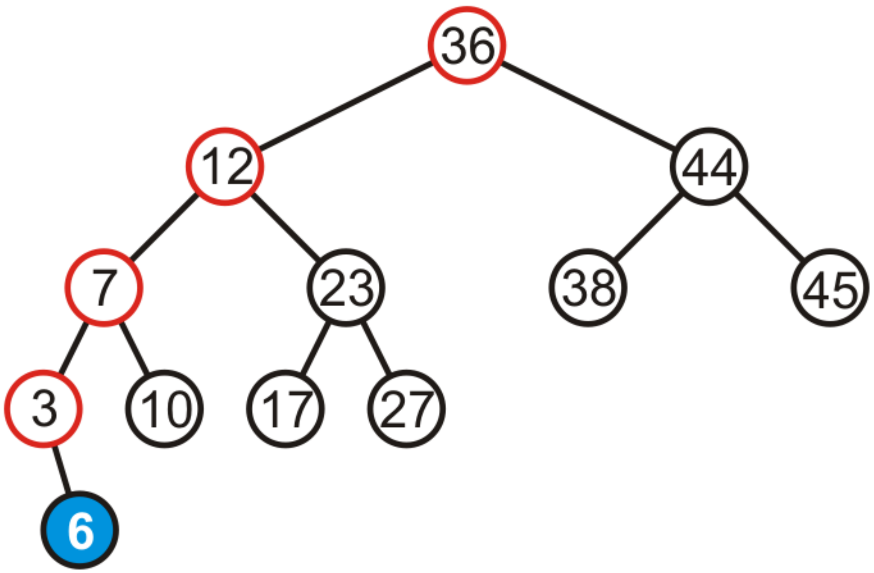
Maintaining Balance

Consider adding 6:



Maintaining Balance

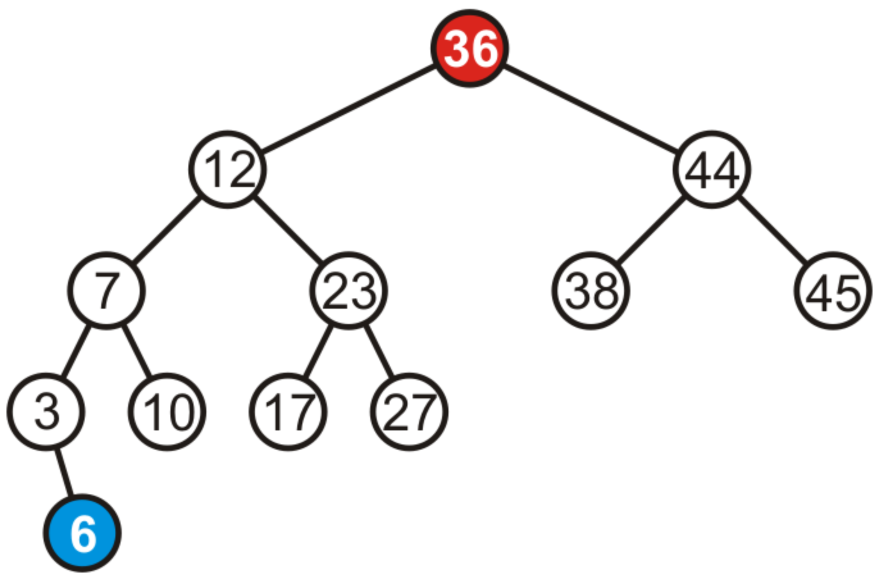
The height of each of the trees in the path back to the root are increased by one



Maintaining Balance

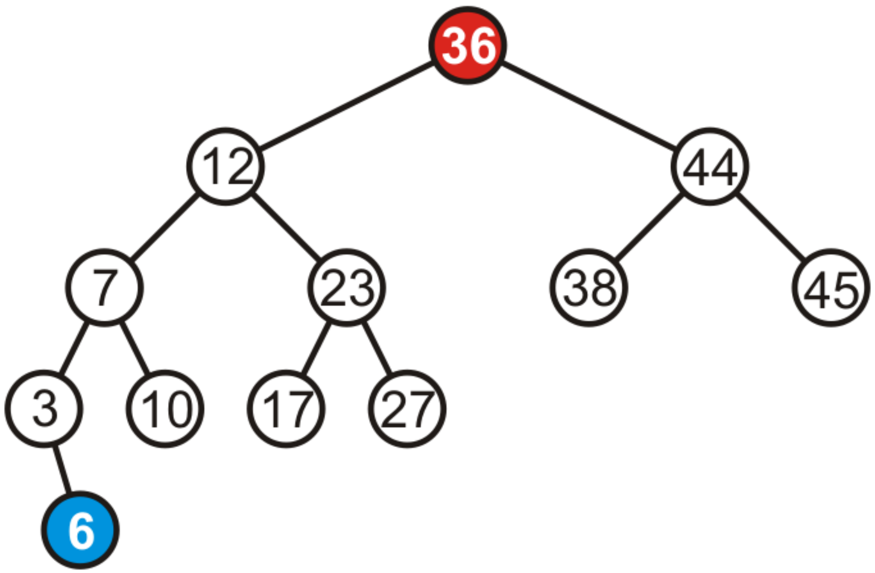
The height of each of the trees in the path back to the root are increased by one

– However, only the root node is now unbalanced



Maintaining Balance

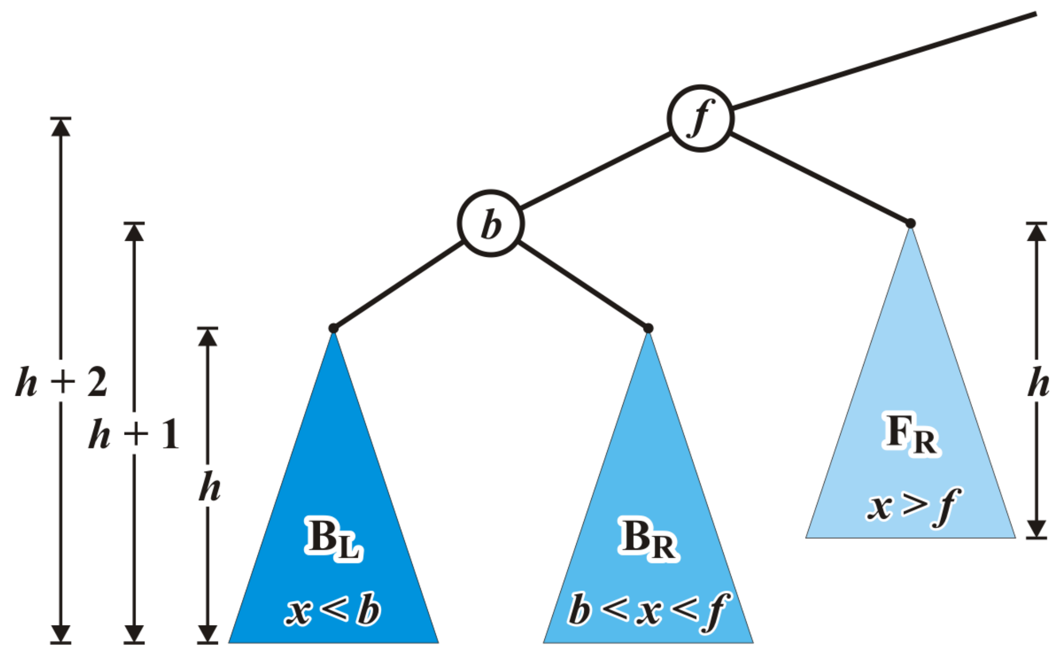
To fix this, we will look at the general case…



Maintaining Balance: Case 1

Consider the following setup

– Each blue triangle represents a tree of height h

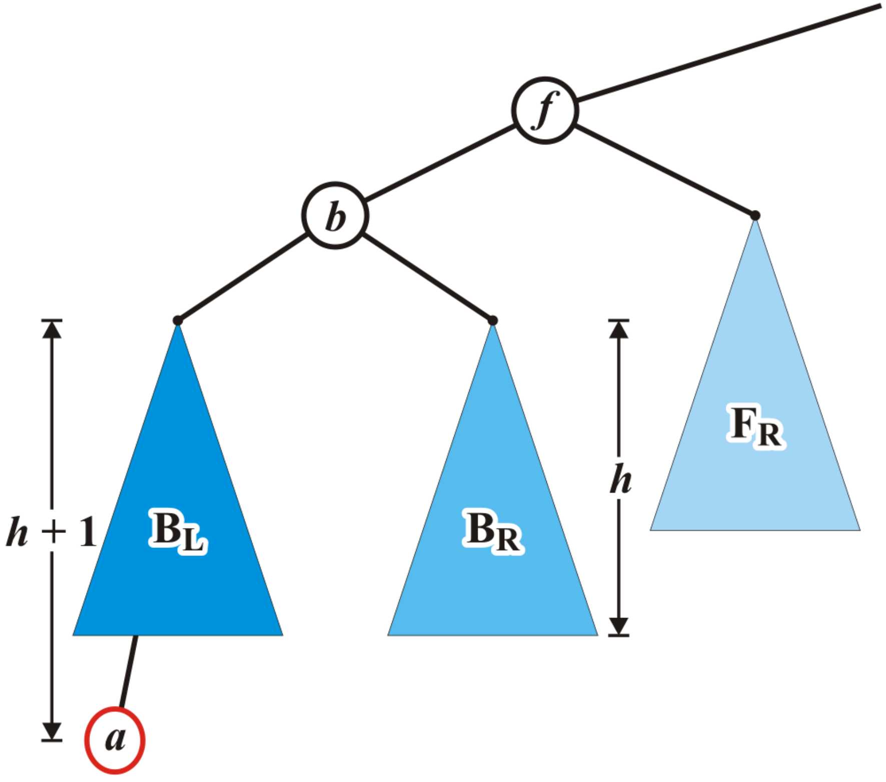


Maintaining Balance: Case 1

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

left-left

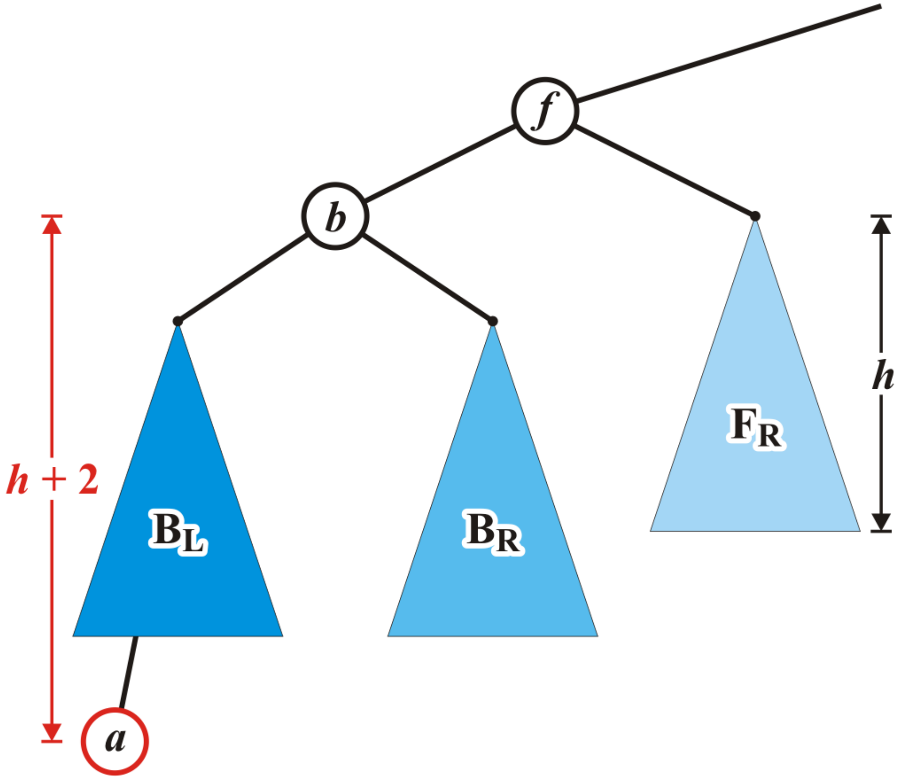
- Assume $\mathbf{B_L}$ remains balanced
- The tree rooted at ***b*** is also balanced



Maintaining Balance: Case 1

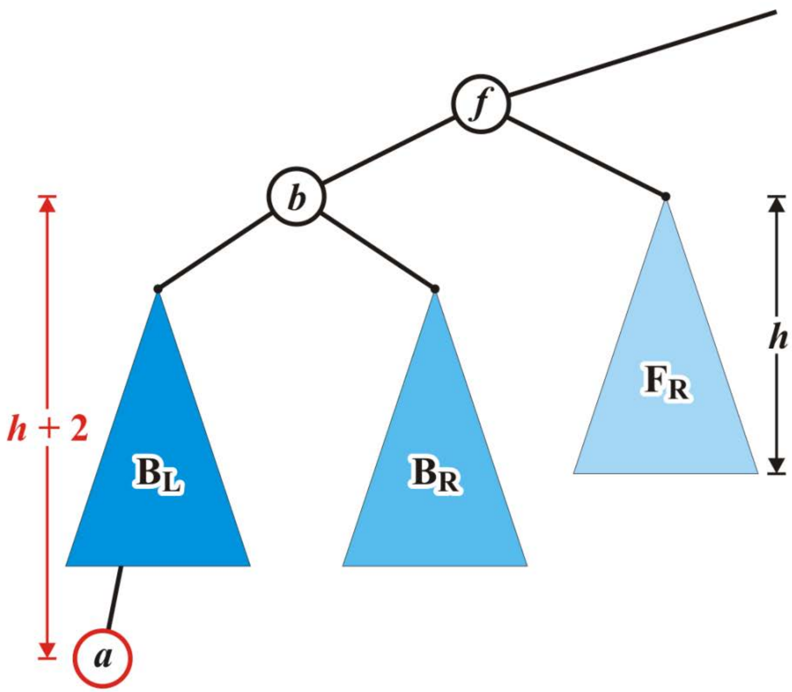
The tree rooted at node ***f*** is now unbalanced

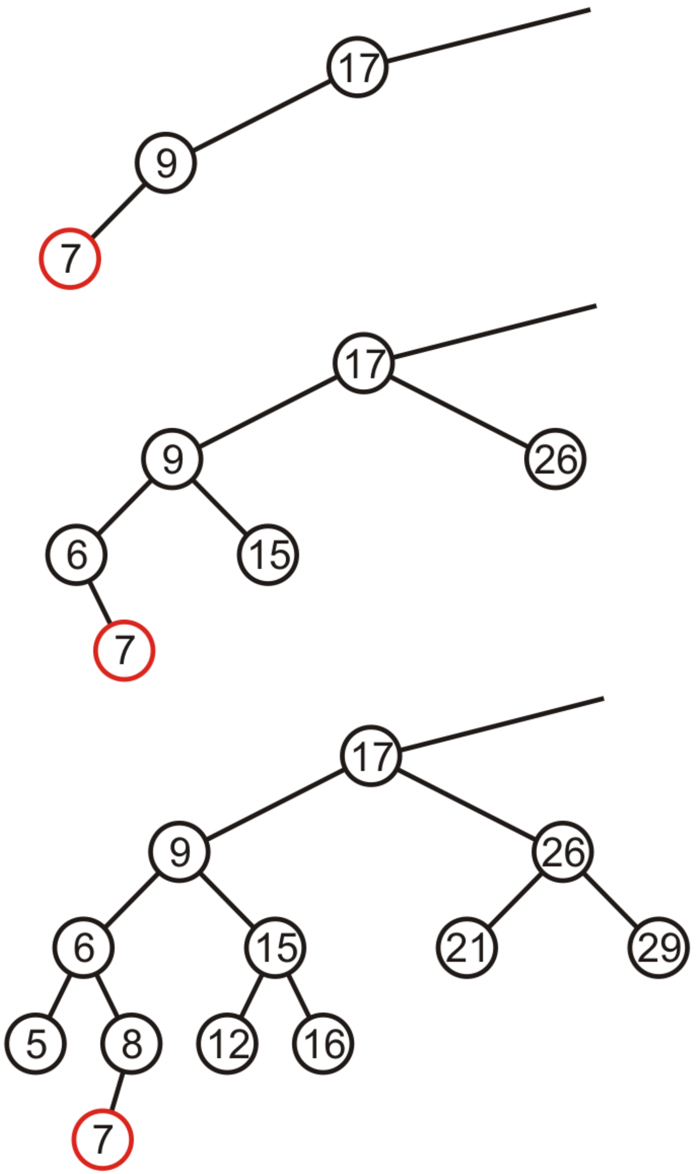
- We will correct the imbalance at this node



Maintaining Balance: Case 1

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



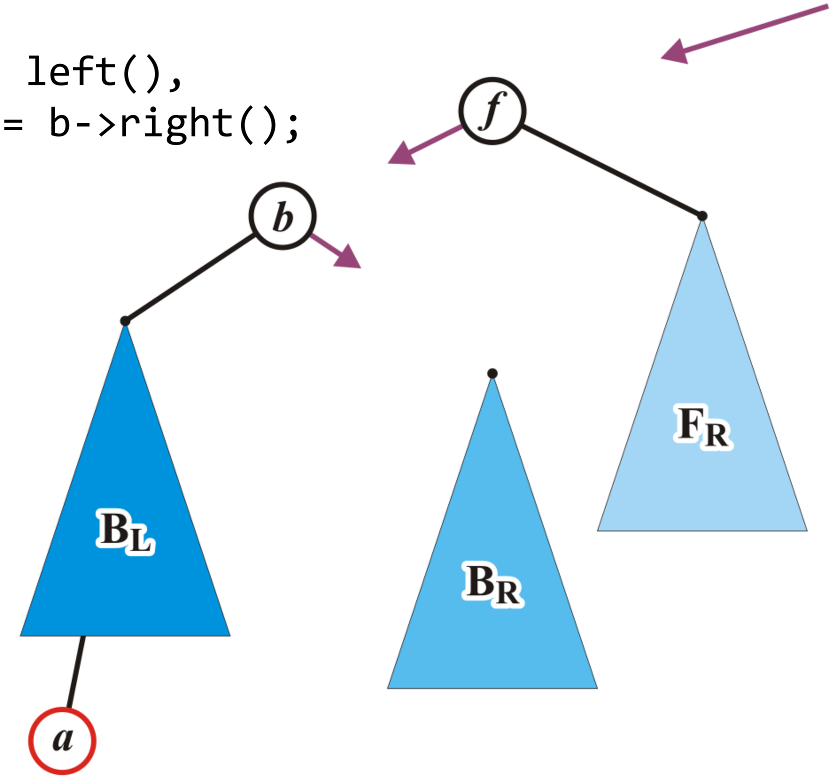


Maintaining Balance: Case 1

We will modify these three pointers

- At this point, `this` references the unbalanced root node ***f***

```
AVL_node<Type> *b = left(),
               *BR = b->right();
```

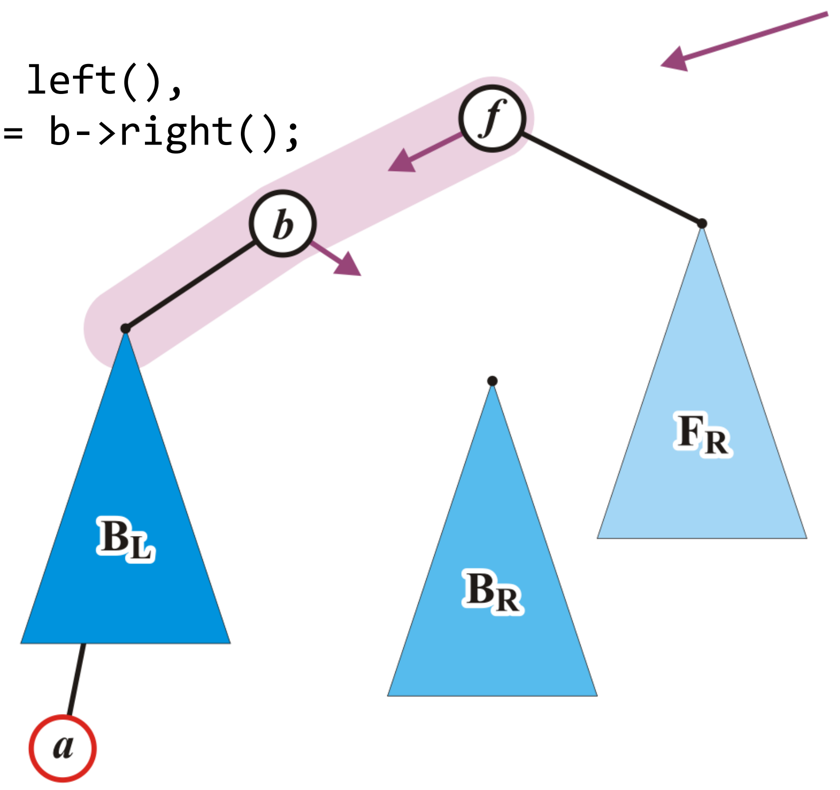


Maintaining Balance: Case 1

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

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

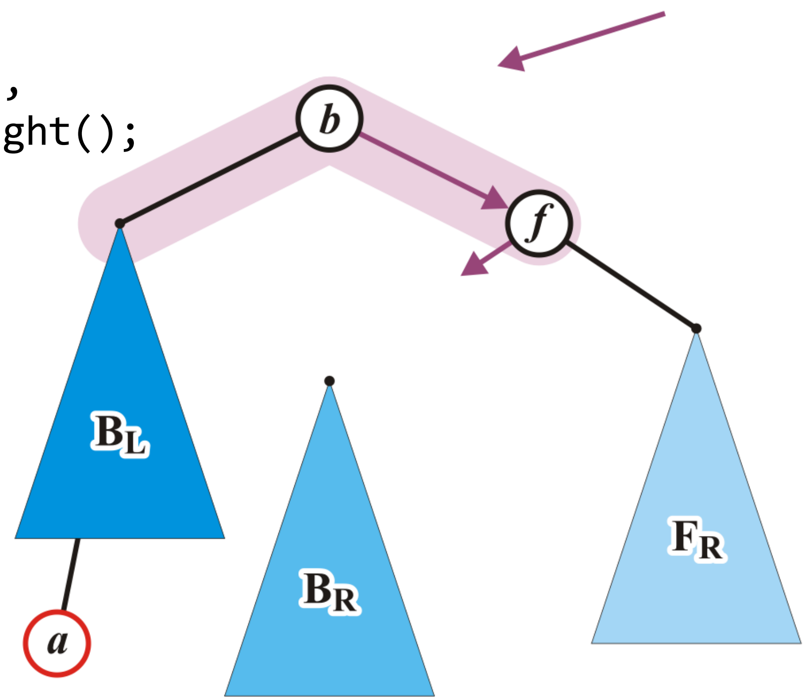
```
AVL_node<Type> *b = left(),
               *BR = b->right();
```



Maintaining Balance: Case 1

This requires the address of node ***f*** to be assigned to the `right_tree` member variable of node ***b***

```
AVL_node<Type> *b = left(),
               *BR = b->right();
b->right_tree = this;
```

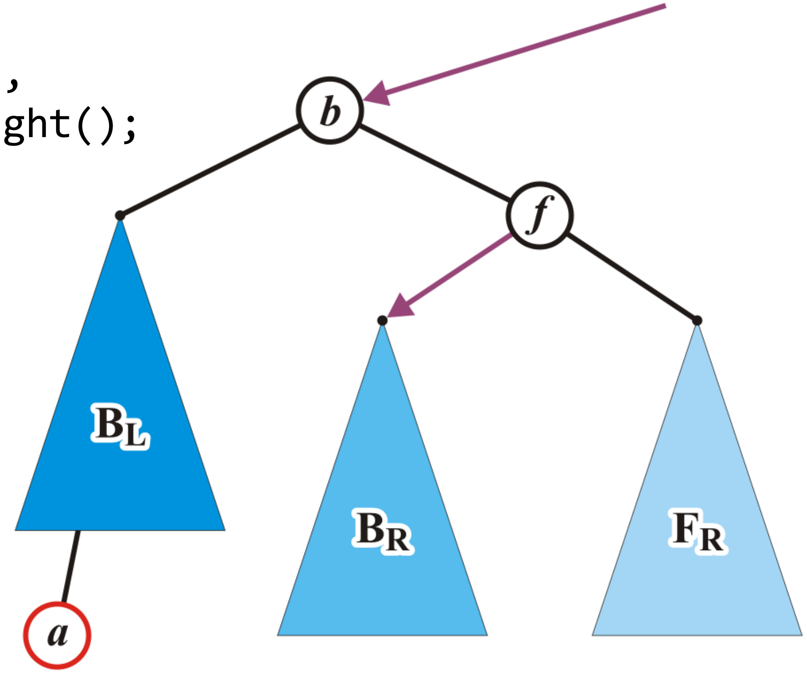


Maintaining Balance: Case 1

Assign any former parent of node ***f*** to the address of node ***b***

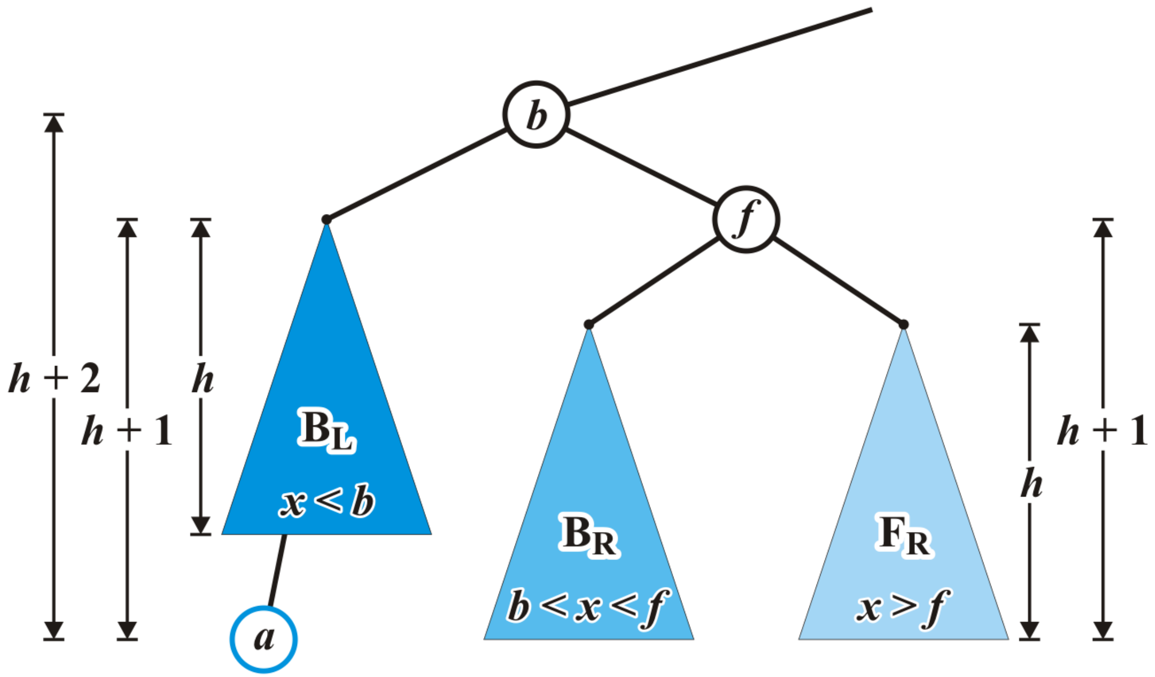
Assign the address of the tree $\mathbf{B_R}$ to `left_tree` of node ***f***

```
AVL_node<Type> *b = left(),
               *BR = b->right();
b->right_tree = this;
ptr_to_this   = b;
left_tree     = BR;
```



Maintaining Balance: Case 1

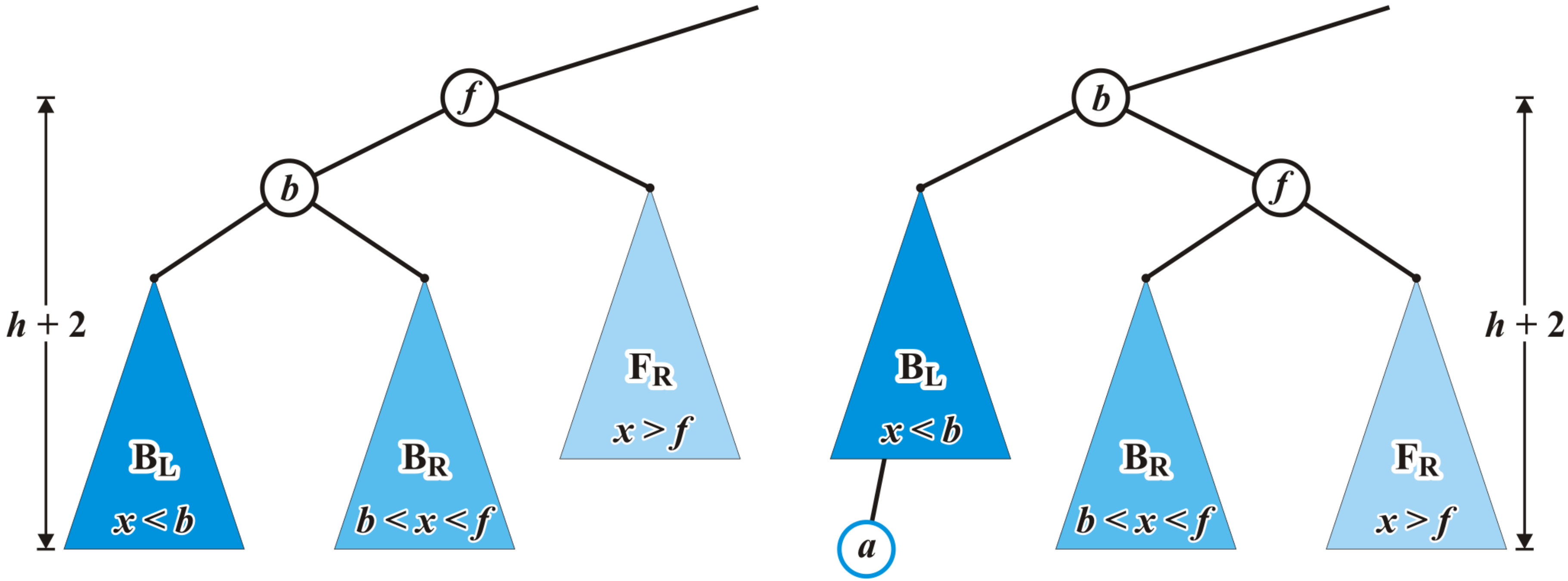
The nodes ***b*** and ***f*** are now balanced and all remaining nodes of the subtrees are in their correct positions



Maintaining Balance: Case 1

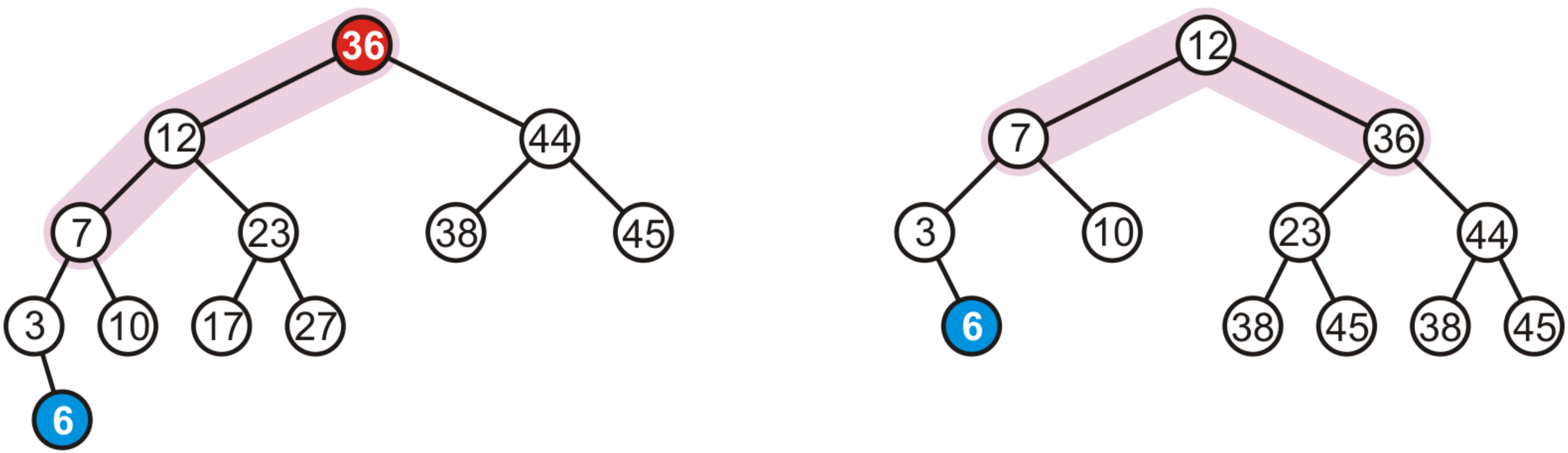
Q: will this insertion affect the balance of any ancestors all the way back to the root?

No, because height of the tree rooted at ***b*** equals the original height of the tree rooted at ***f***



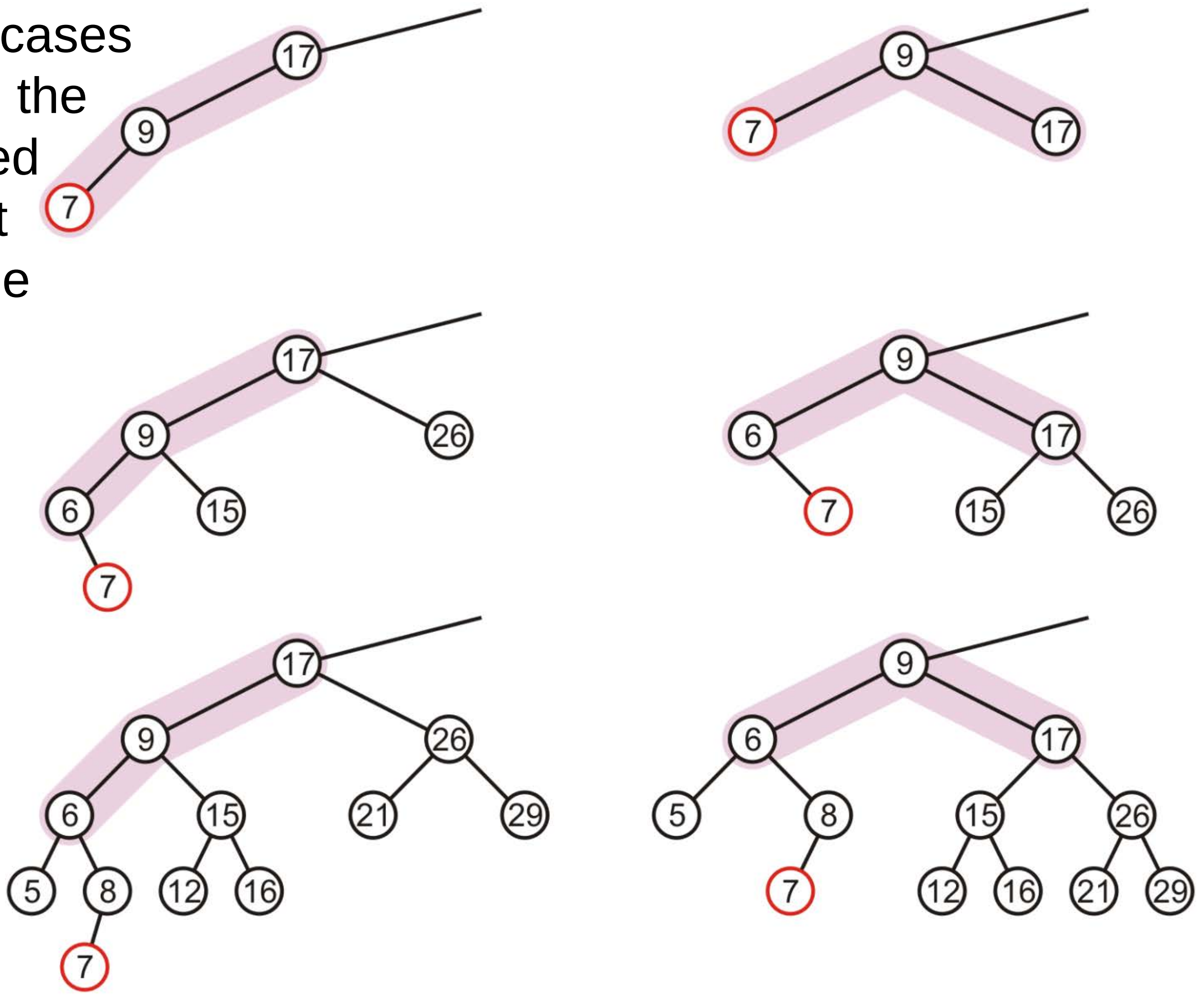
Maintaining Balance: Case 1

In our example case, the correction



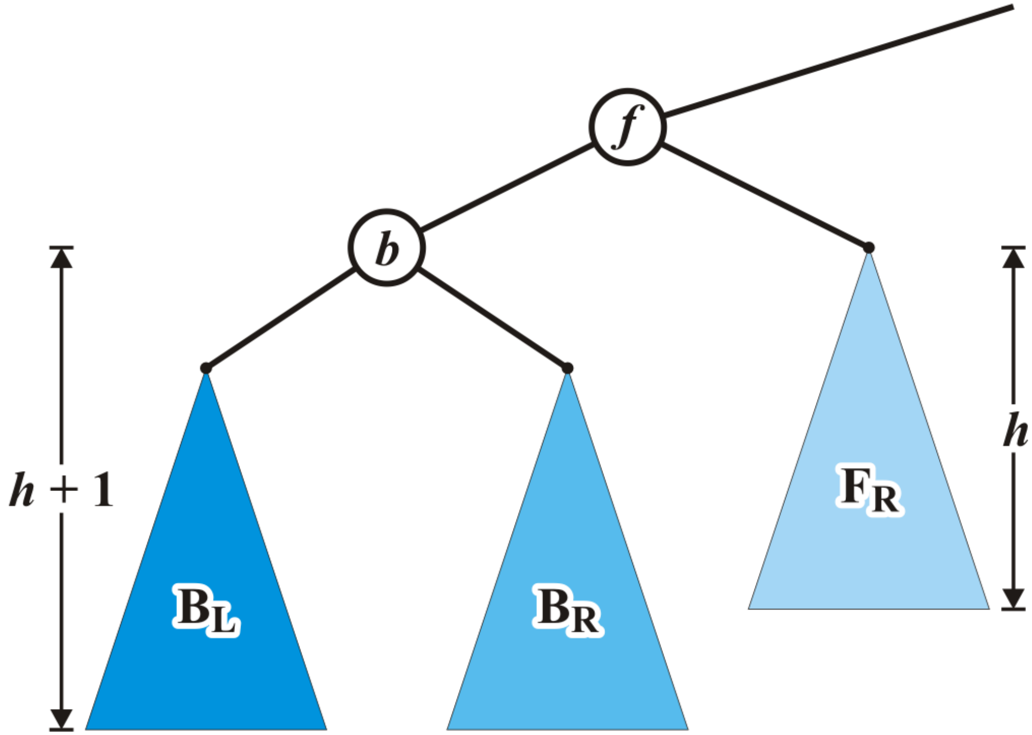
Maintaining Balance: Case 1

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 c where $b < c < f$ into our original tree

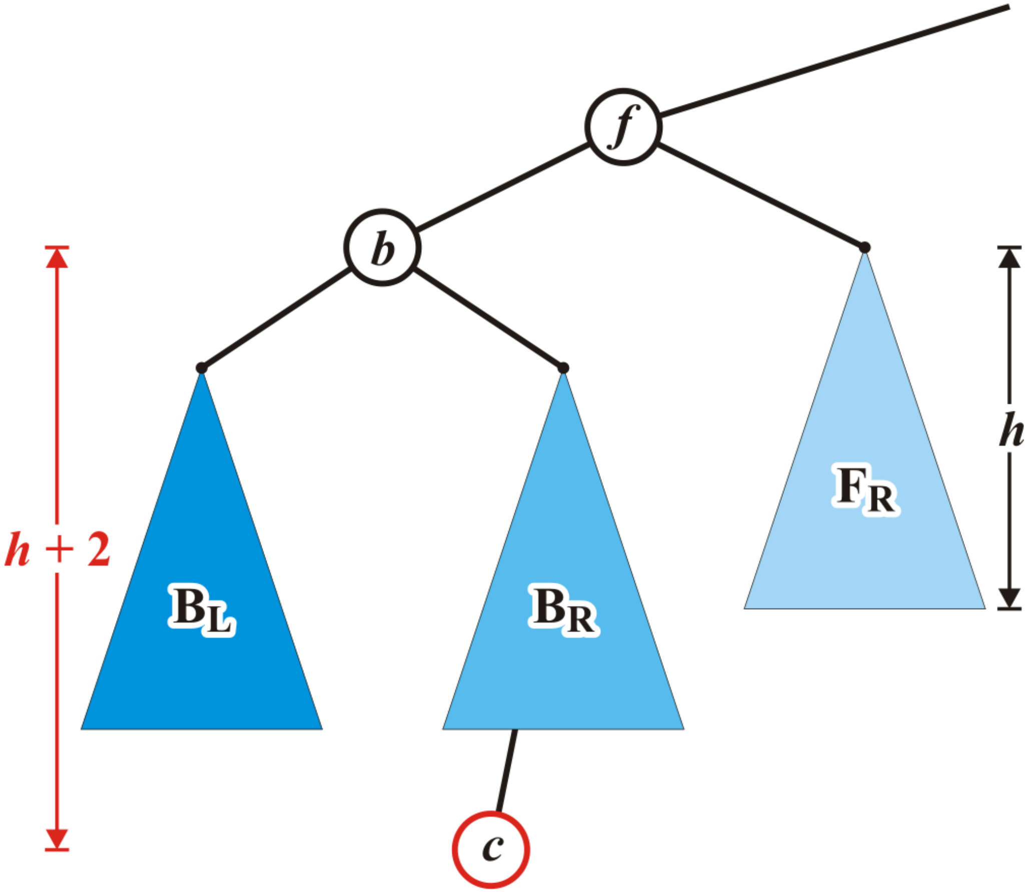


Maintaining Balance: Case 2

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

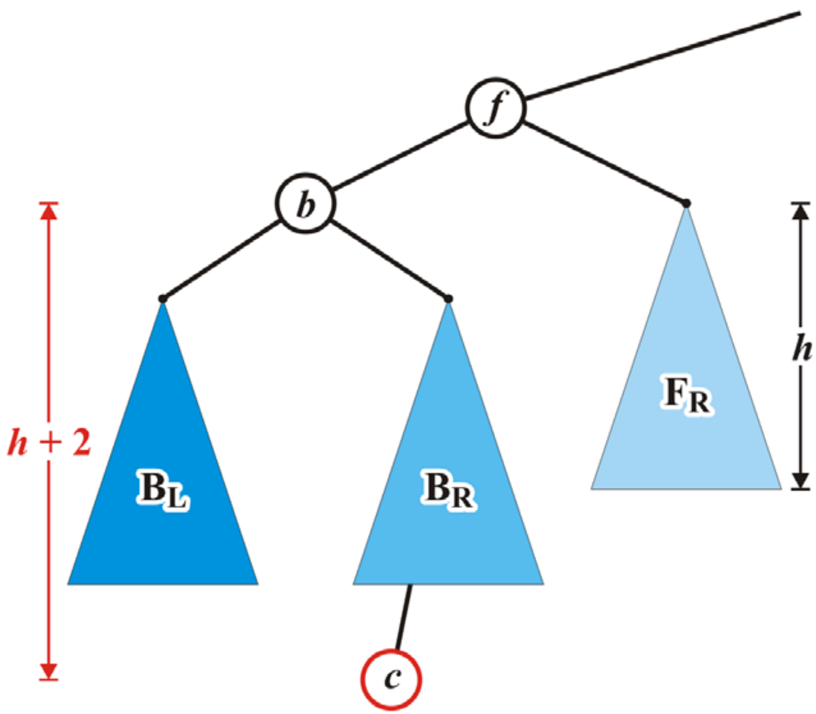
- Once again, f becomes unbalanced

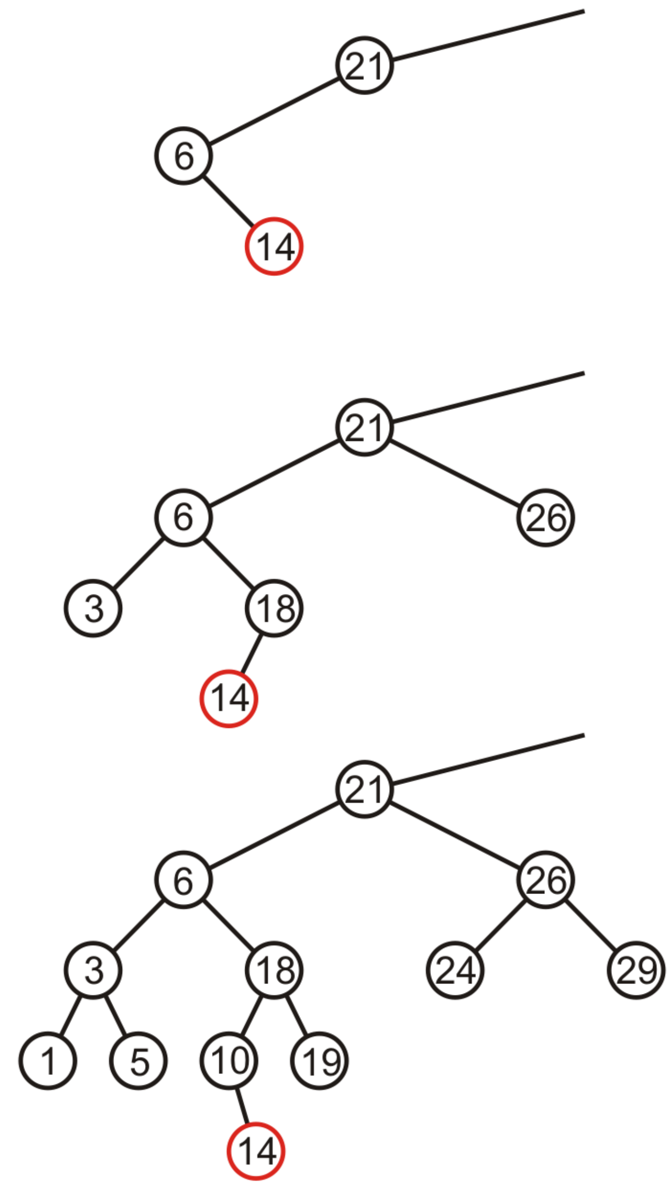
left-right



Maintaining Balance: Case 2

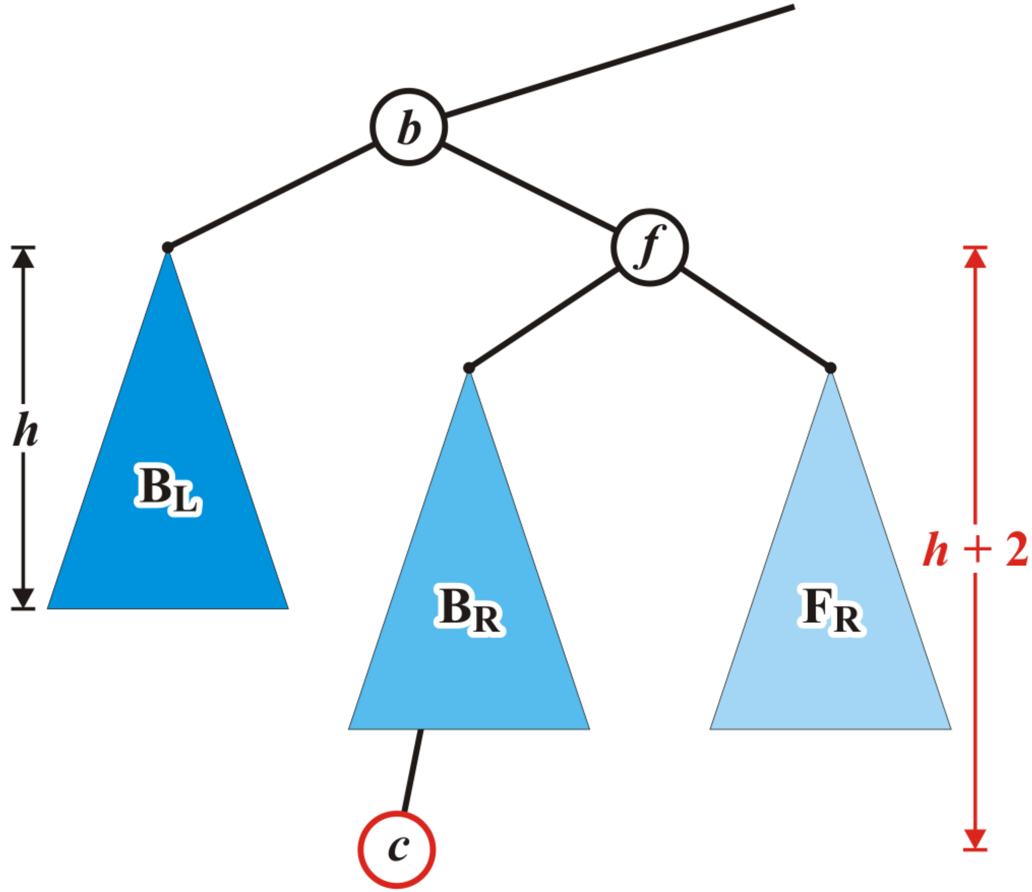
Here are examples of when the insertion of 14 may cause this situation when $h = -1$, 0, and 1





Maintaining Balance: Case 2

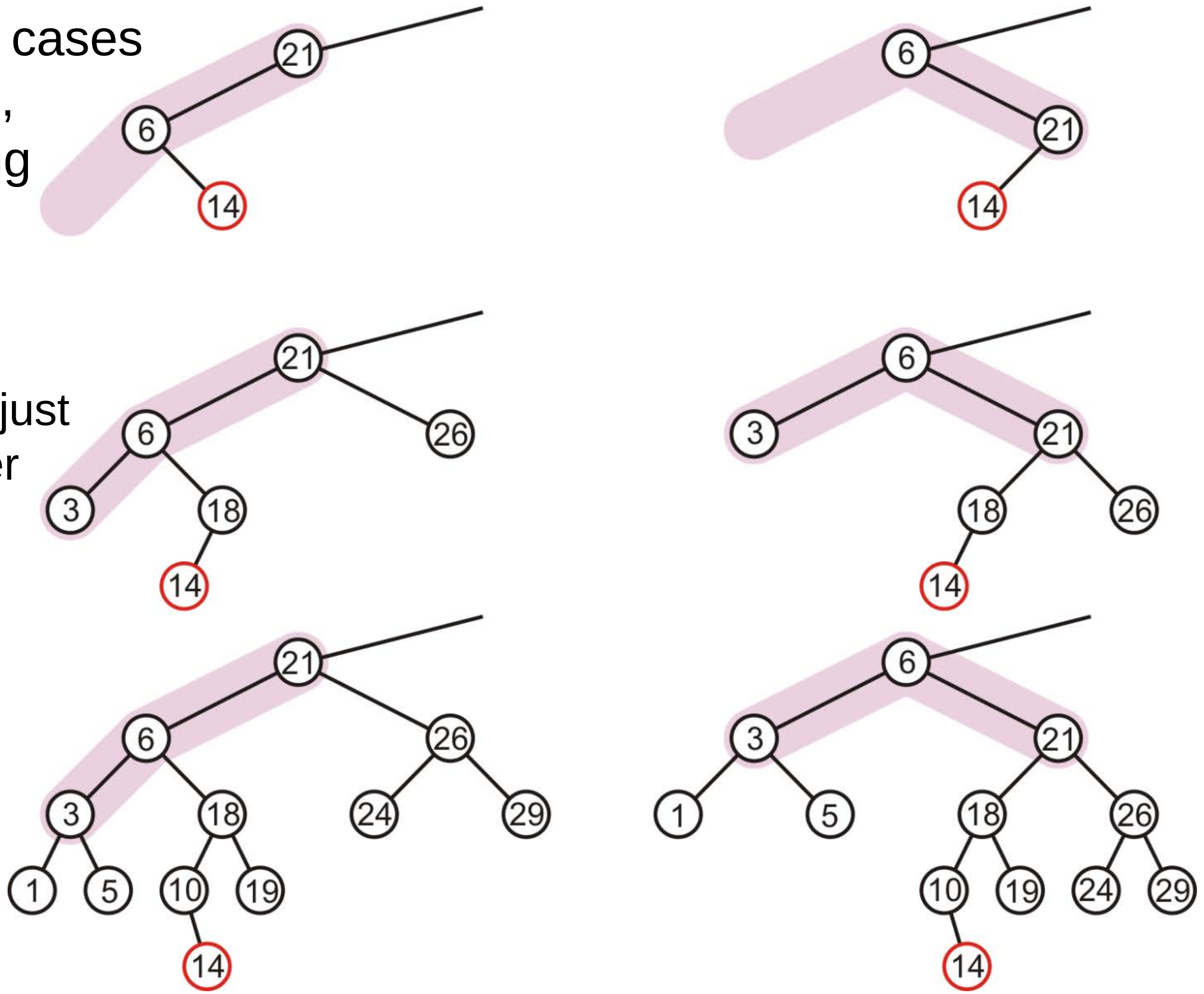
Unfortunately, the previous correction does not fix the imbalance at the root of this sub-tree: the new root, ***b***, remains unbalanced



Maintaining Balance: Case 2

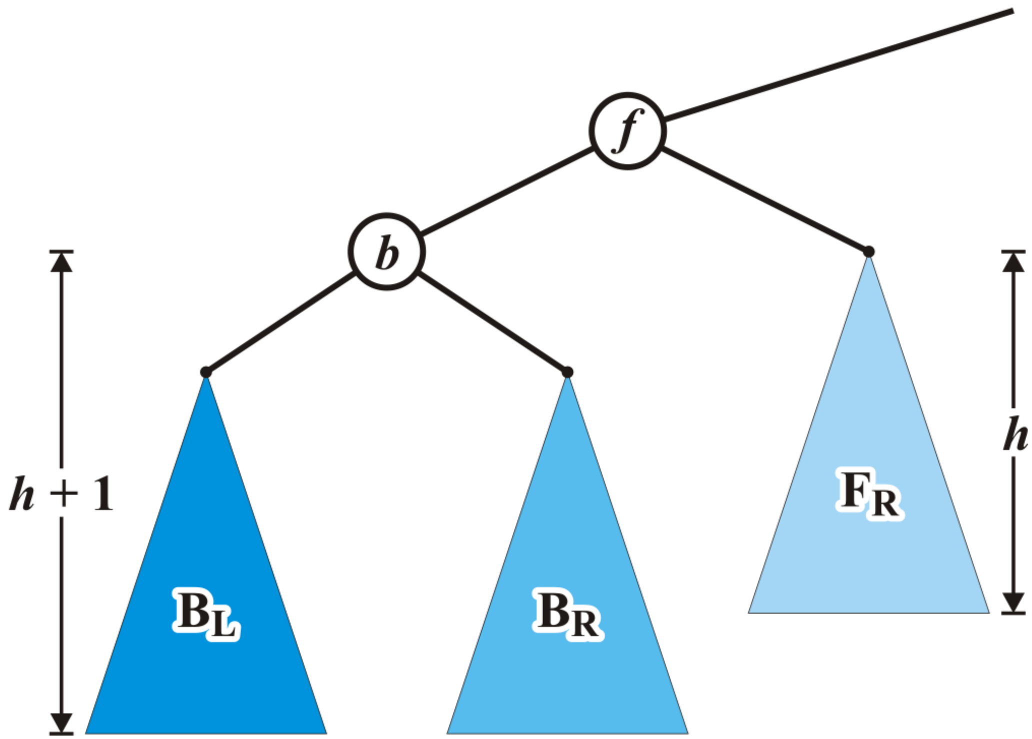
In our three sample cases with $h = -1$, 0, and 1, doing the same thing as before results in a tree that is still unbalanced…

- The imbalance is just shifted to the other side



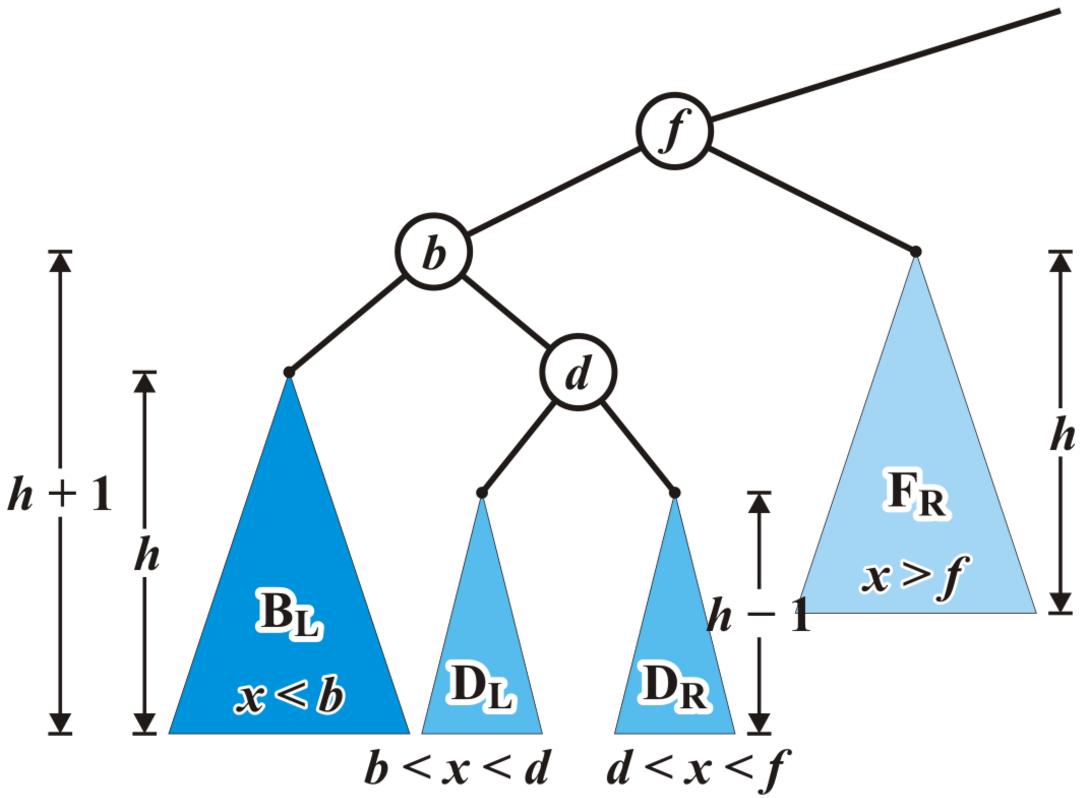
Maintaining Balance: Case 2

We need to look into B_R



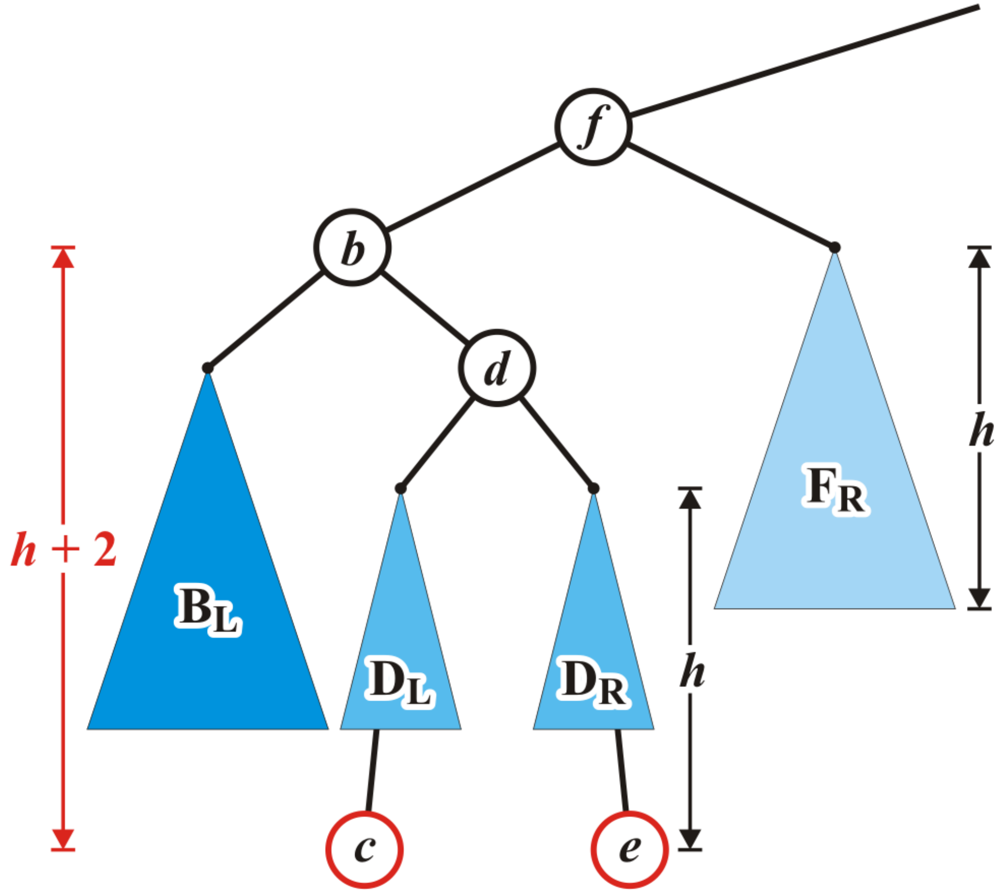
Maintaining Balance: Case 2

Re-label B_R as a tree rooted at d with two subtrees of height $h-1$



Maintaining Balance: Case 2

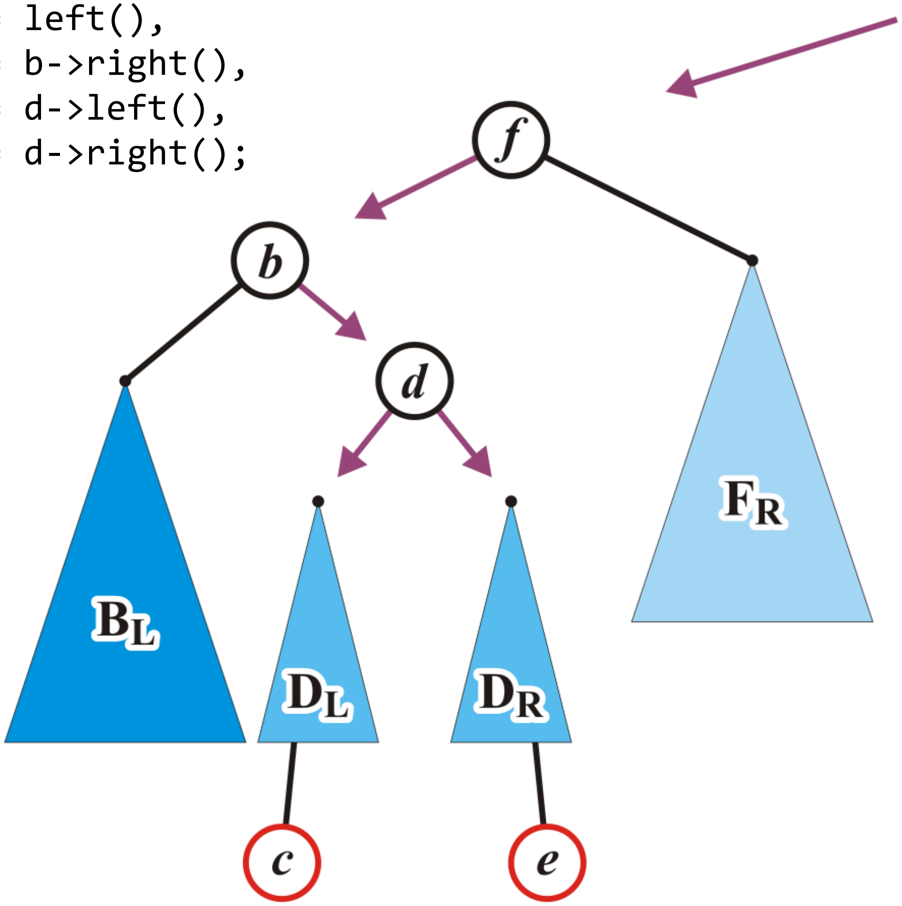
- Now an insertion causes an imbalance at ***f***
  - The addition of either ***c*** or ***e*** will cause this



Maintaining Balance: Case 2

We will reassign the following pointers

```
AVL_node<Type> *b  = left(),
               *d  = b->right(),
               *DL = d->left(),
               *DR = d->right();
```

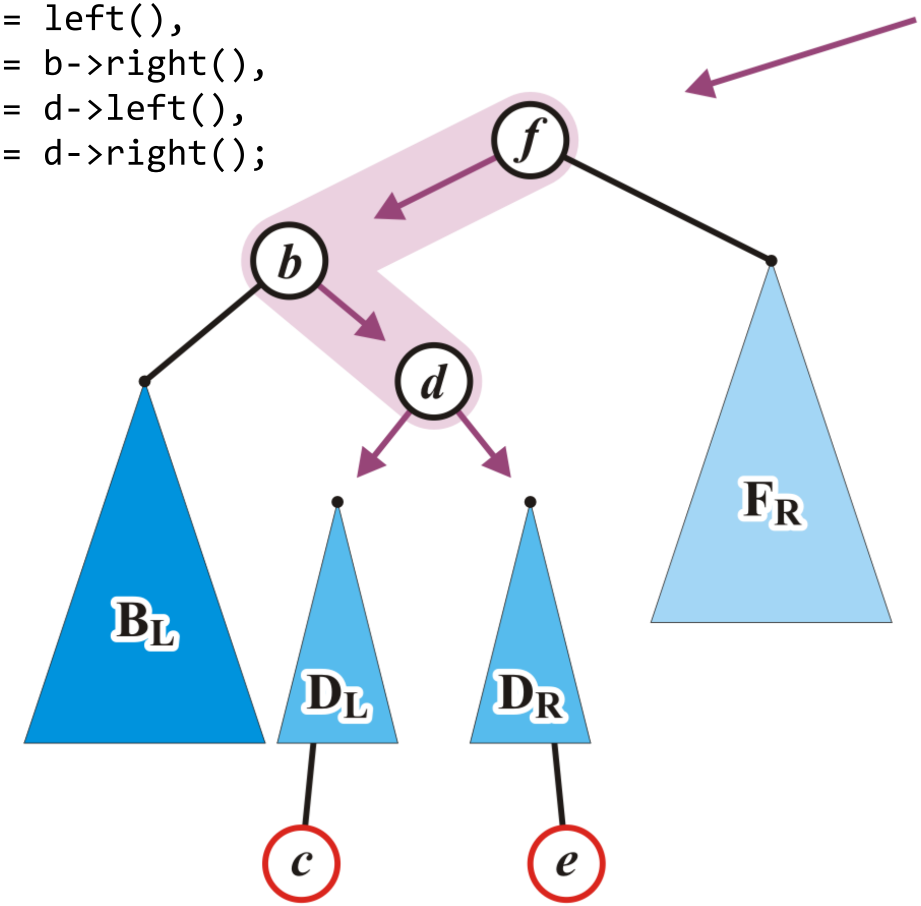


Maintaining Balance: Case 2

Specifically, we will order these three nodes as a perfect tree

- Recall the second prototypical example

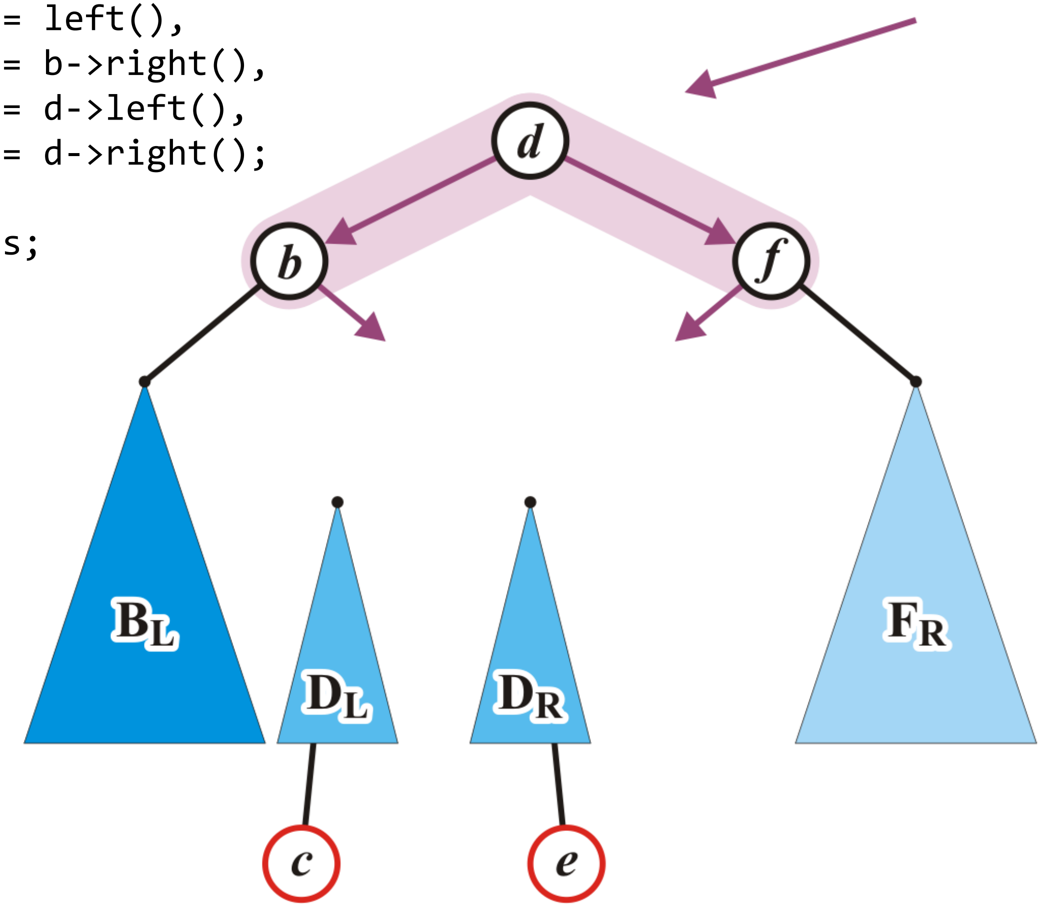
```
AVL_node<Type> *b  = left(),
               *d  = b->right(),
               *DL = d->left(),
               *DR = d->right();
```



Maintaining Balance: Case 2

To achieve this, ***b*** and ***f*** will be assigned as children of the new root ***d***

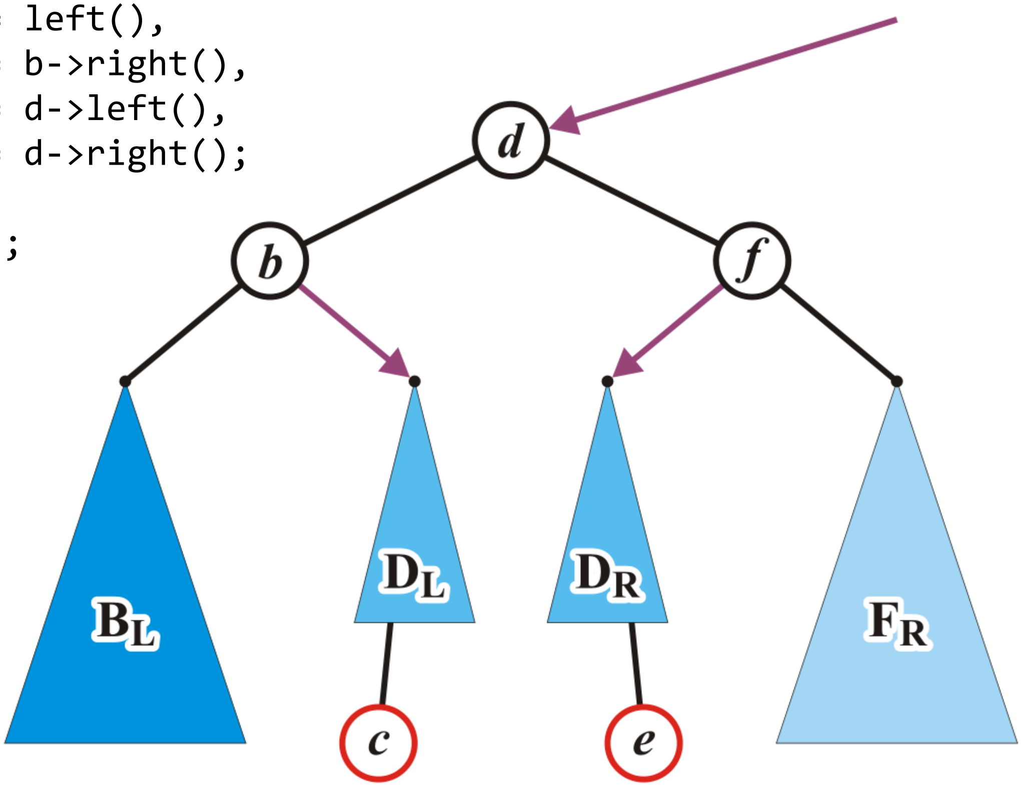
```
AVL_node<Type> *b  = left(),
               *d  = b->right(),
               *DL = d->left(),
               *DR = d->right();
d->left_tree  = b;
d->right_tree = this;
```



Maintaining Balance: Case 2

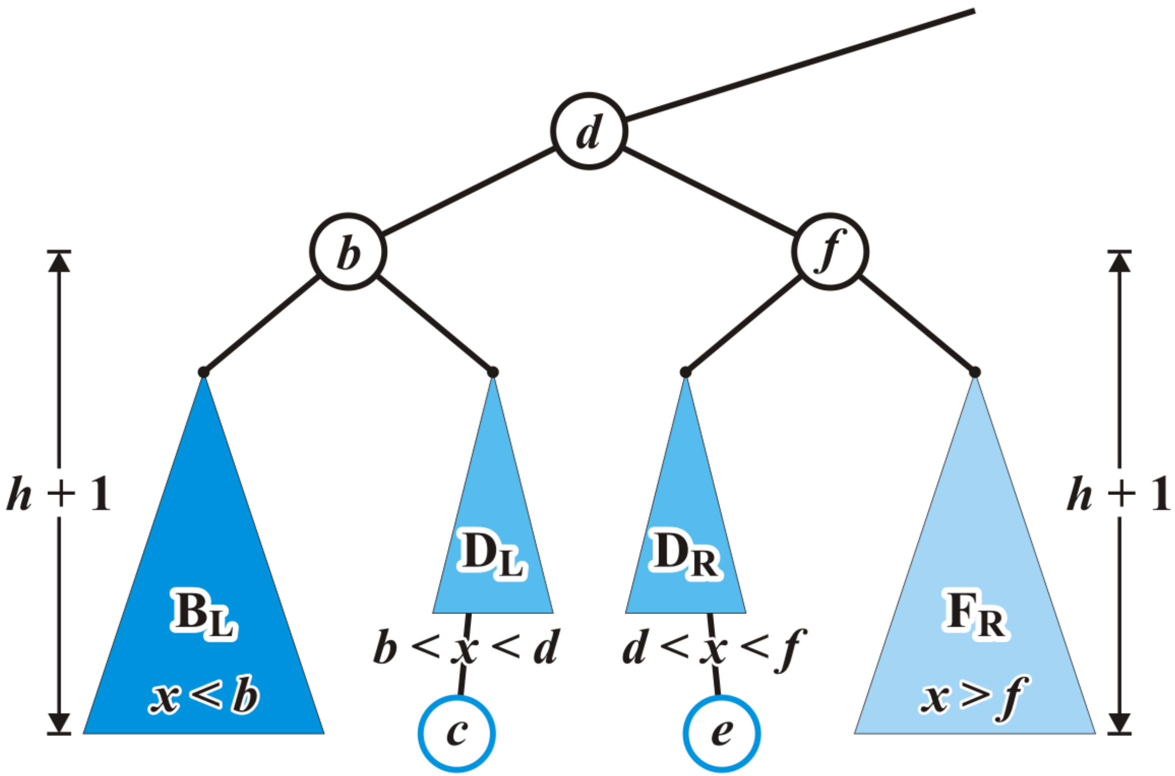
We also have to connect the two subtrees and original parent of ***f***

```
AVL_node<Type> *b  = left(),
               *d  = b->right(),
               *DL = d->left(),
               *DR = d->right();
d->left_tree  = b;
d->right_tree = this;
ptr_to_this   = d;
b->right_tree = DL;
left_tree     = DR;
```



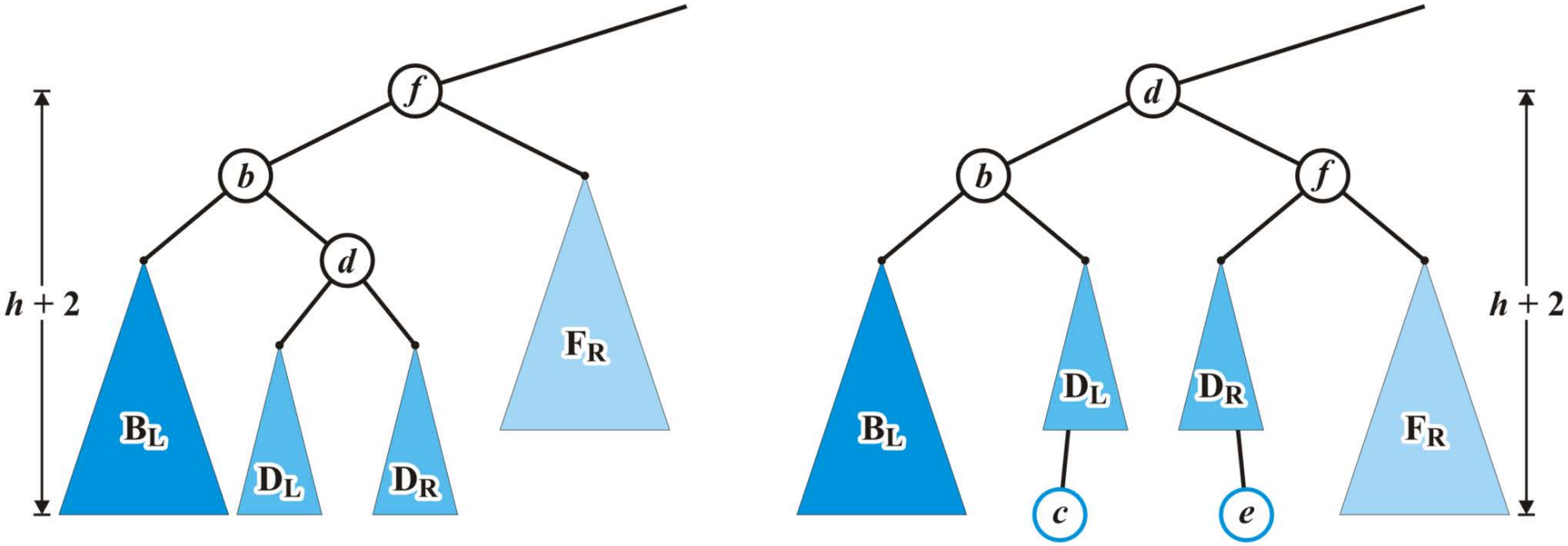
Maintaining Balance: Case 2

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



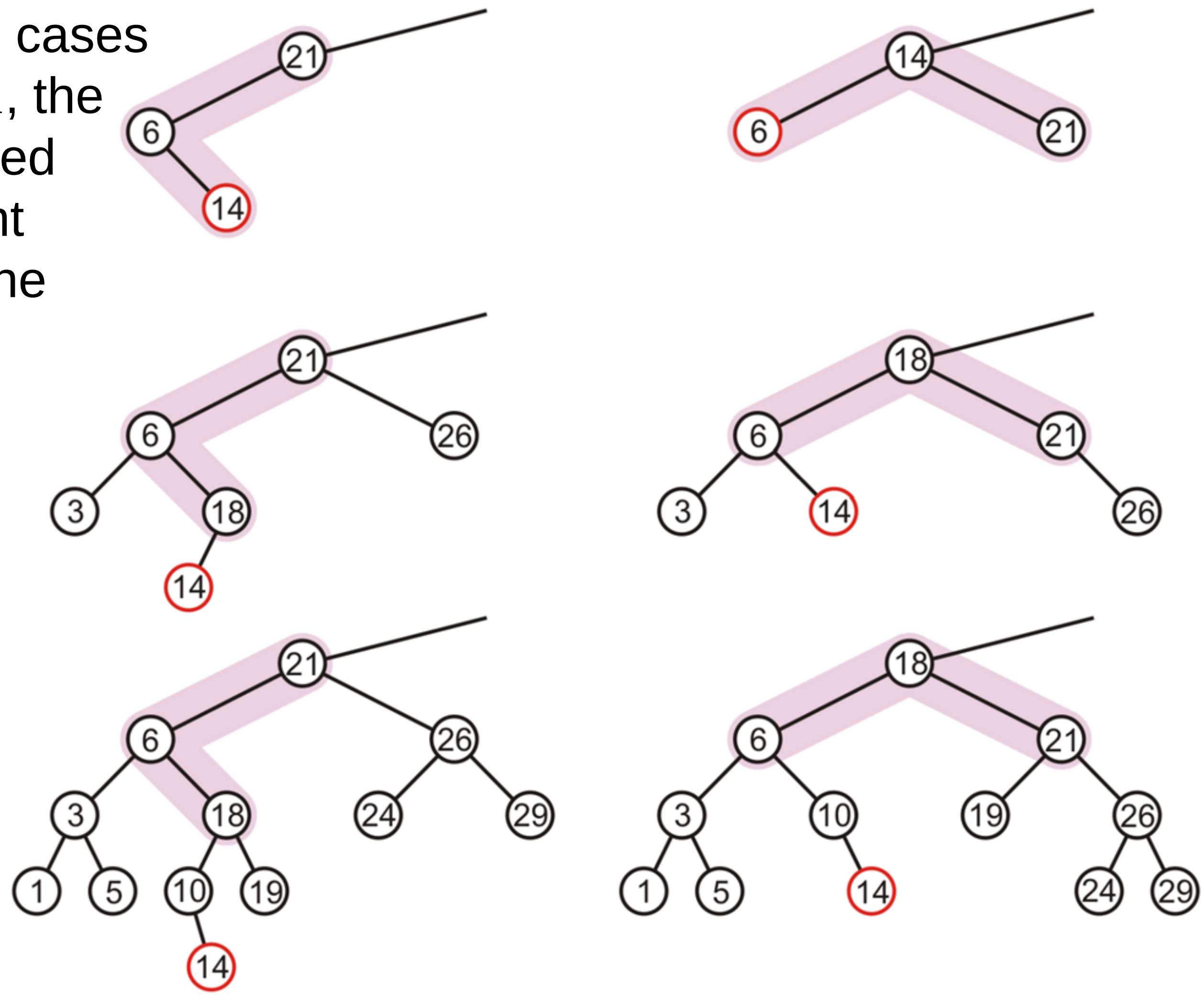
Maintaining Balance: Case 2

Again, the height of the root did not change



Maintaining Balance: Case 2

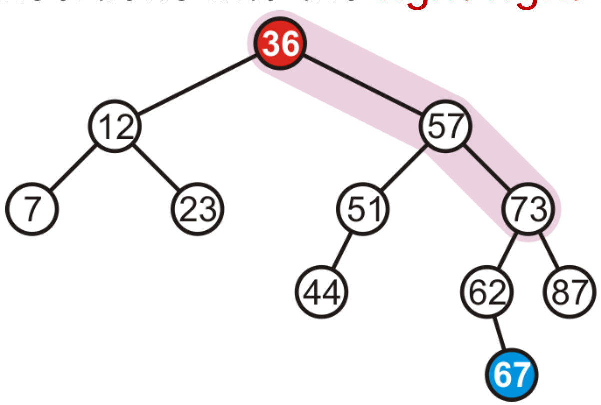
In our three sample cases with $h = -1$, 0, and 1, the node is now balanced and the same height as the tree before the insertion

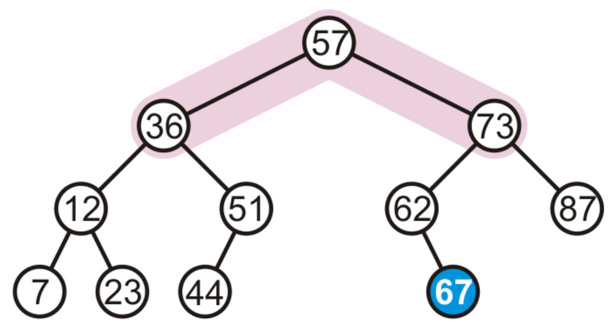


Maintaining balance: symmetric cases

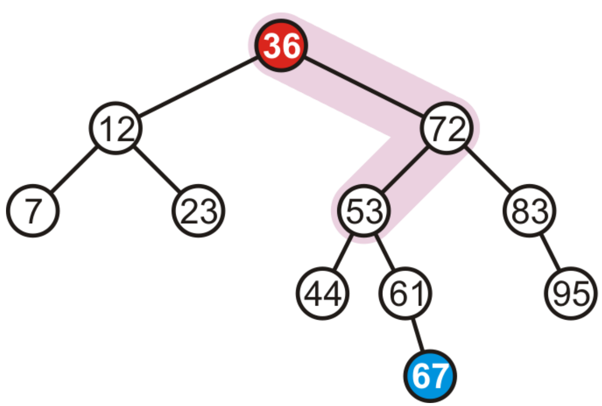
There are two symmetric cases to those we have examined:

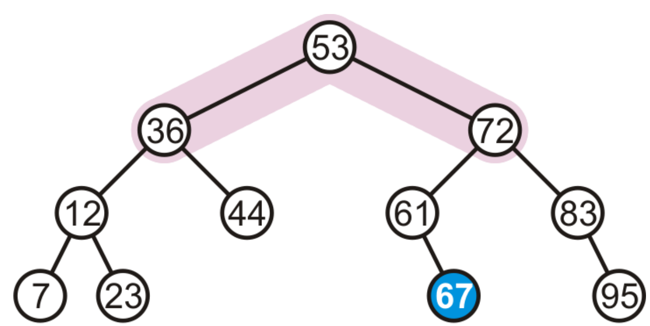
– Insertions into the right-right sub-tree





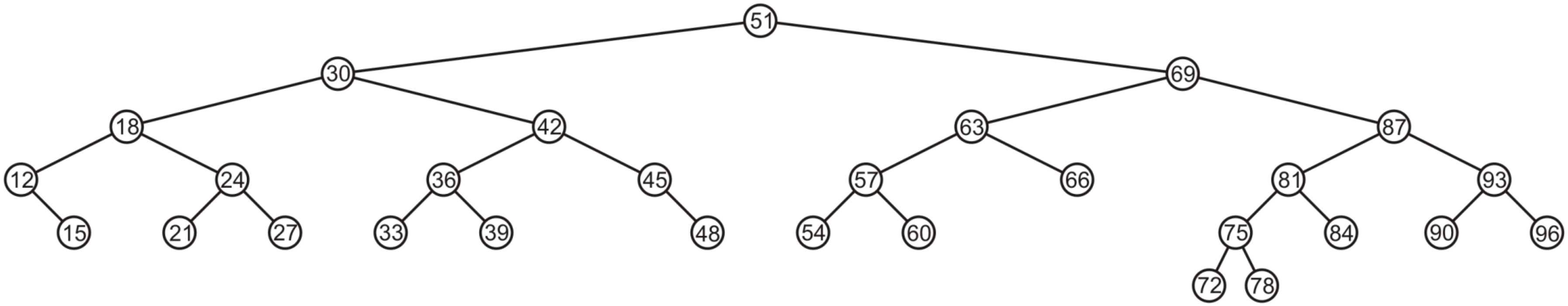
– Insertions into either the right-left sub-tree





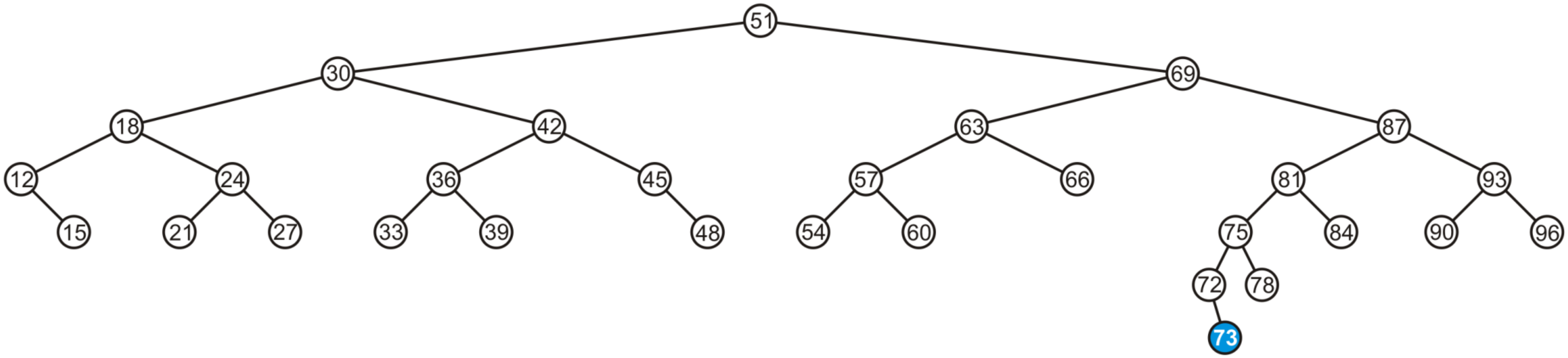
Insertion

Consider this AVL tree



Insertion

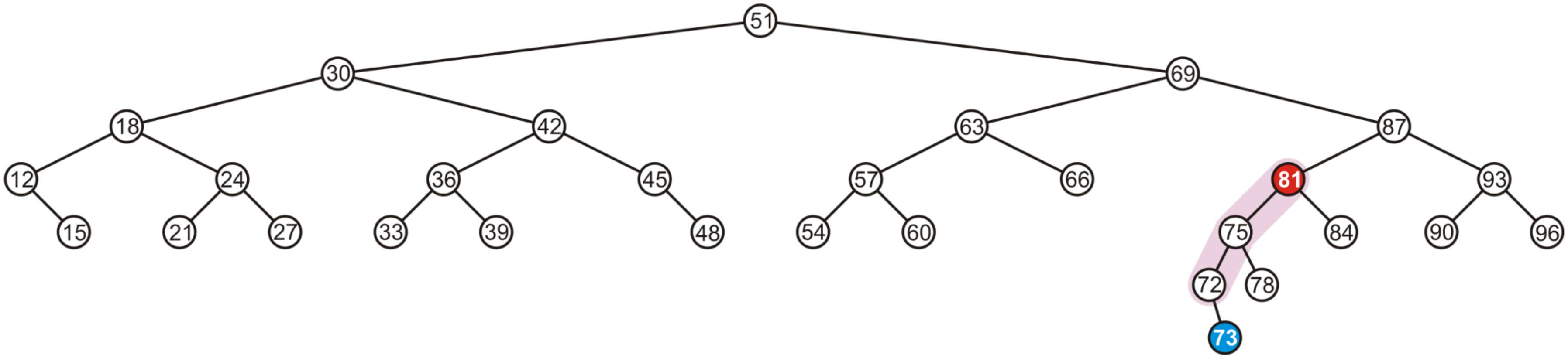
Insert 73



Insertion

The node 81 is unbalanced

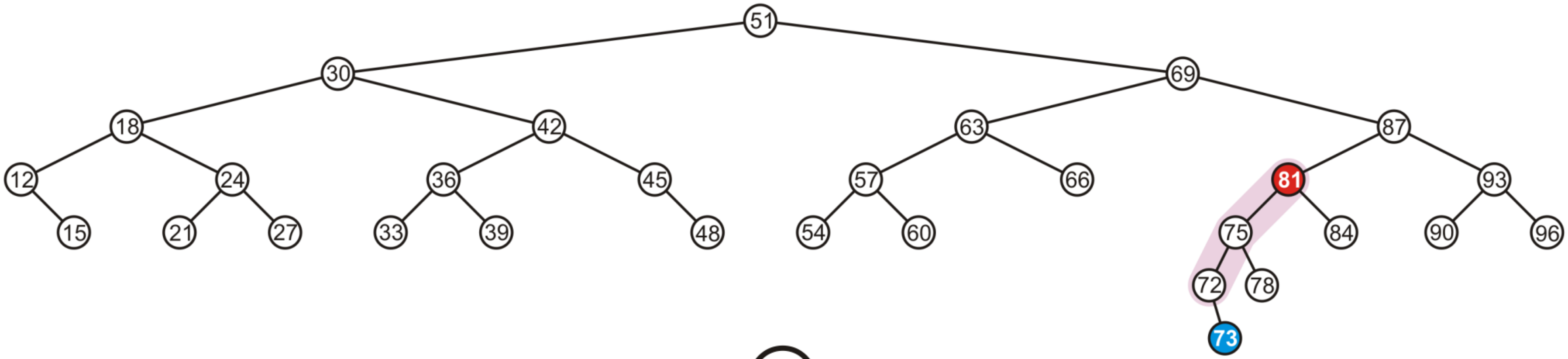
- A left-left imbalance

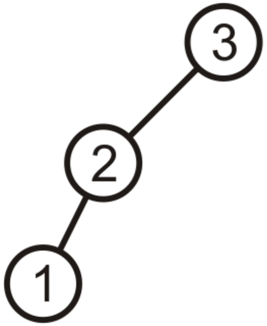


Insertion

The node 81 is unbalanced

- A left-left imbalance

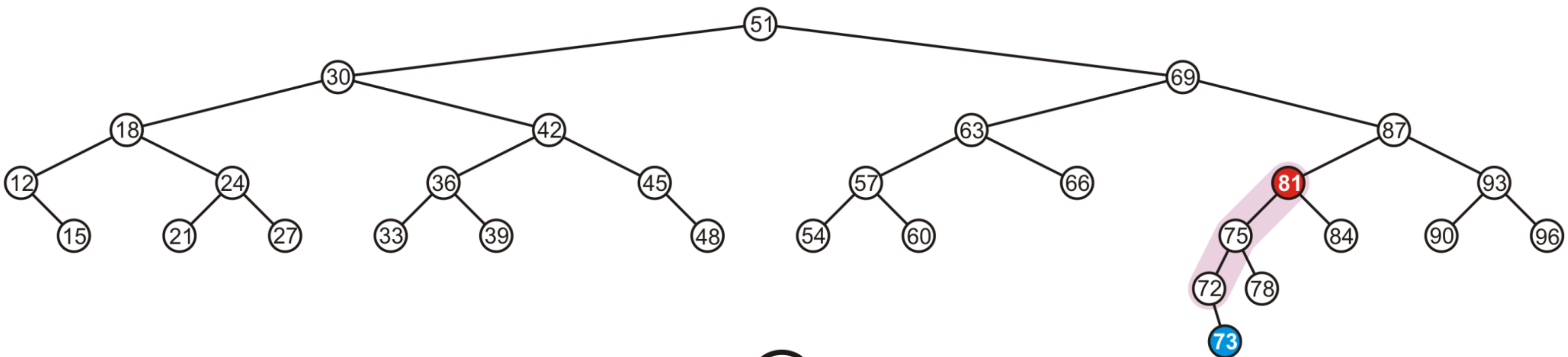


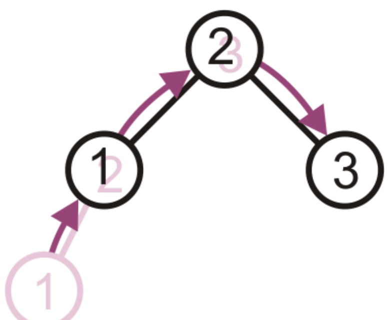


Insertion

The node 81 is unbalanced

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

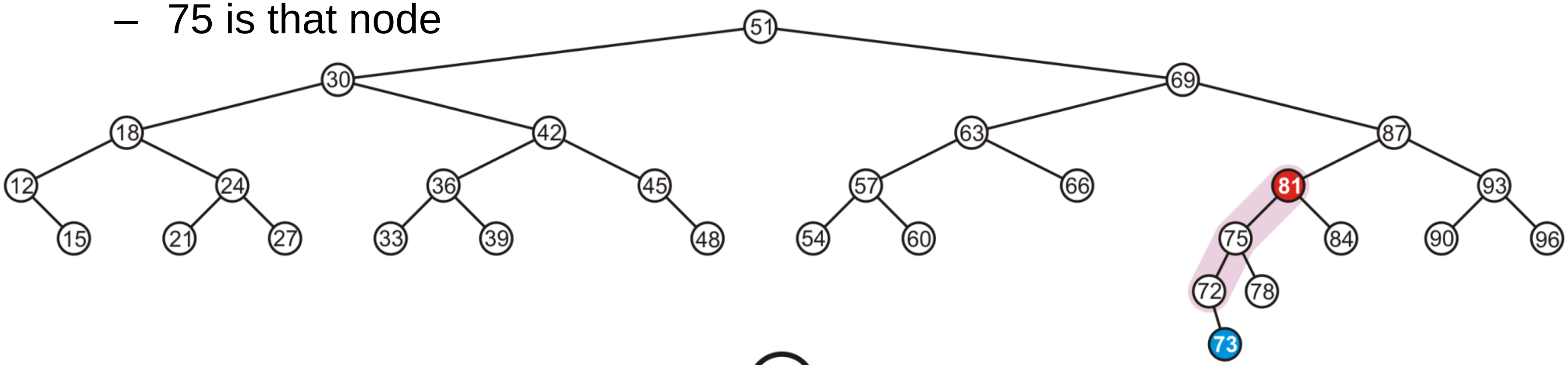


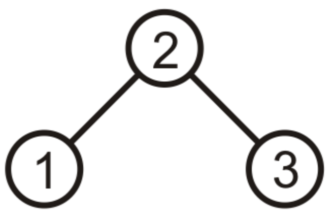


Insertion

The node 81 is unbalanced

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

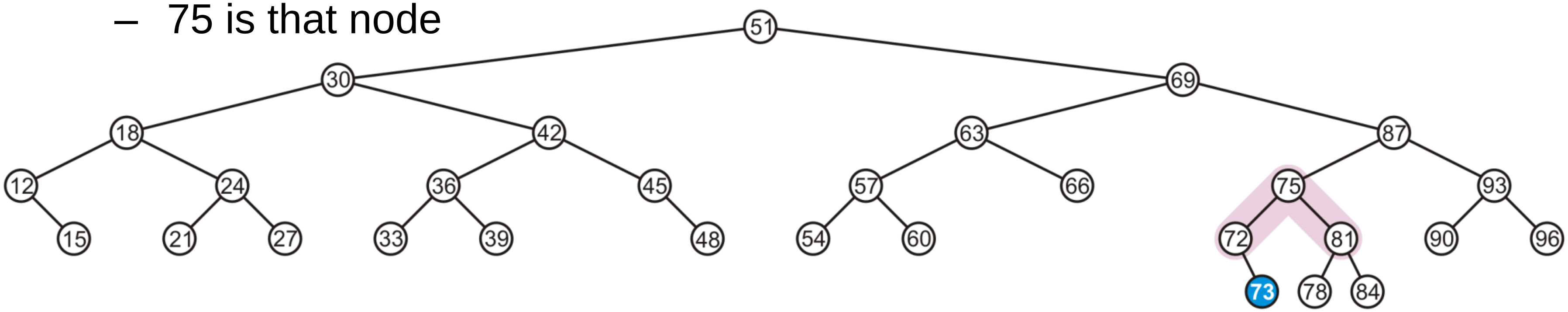




Insertion

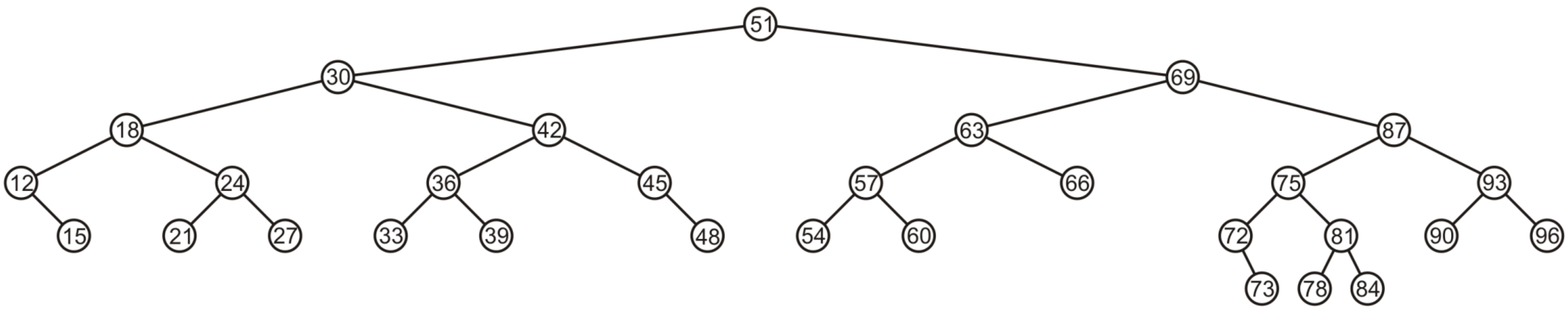
The node 81 is unbalanced

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



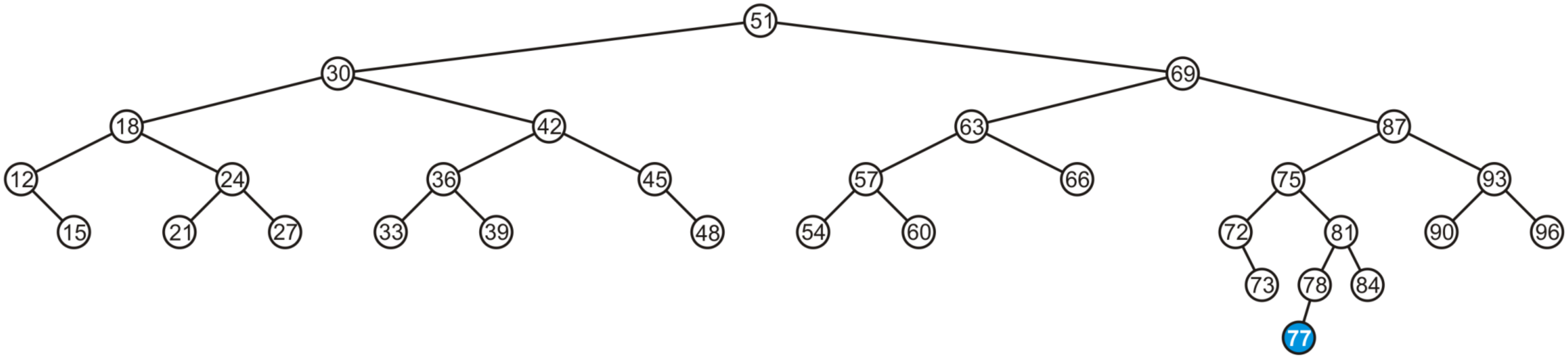
Insertion

The tree is AVL balanced



Insertion

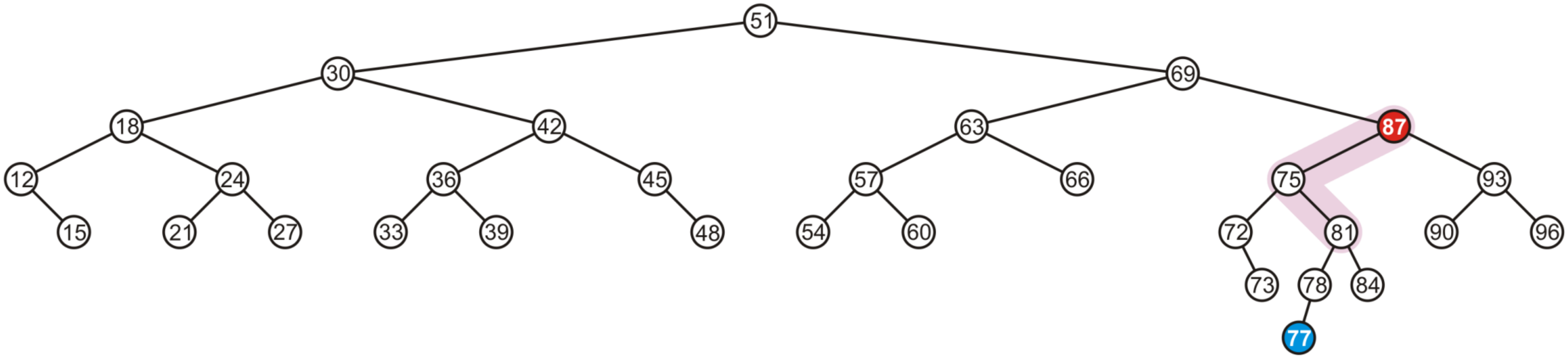
Insert 77



Insertion

The node 87 is unbalanced

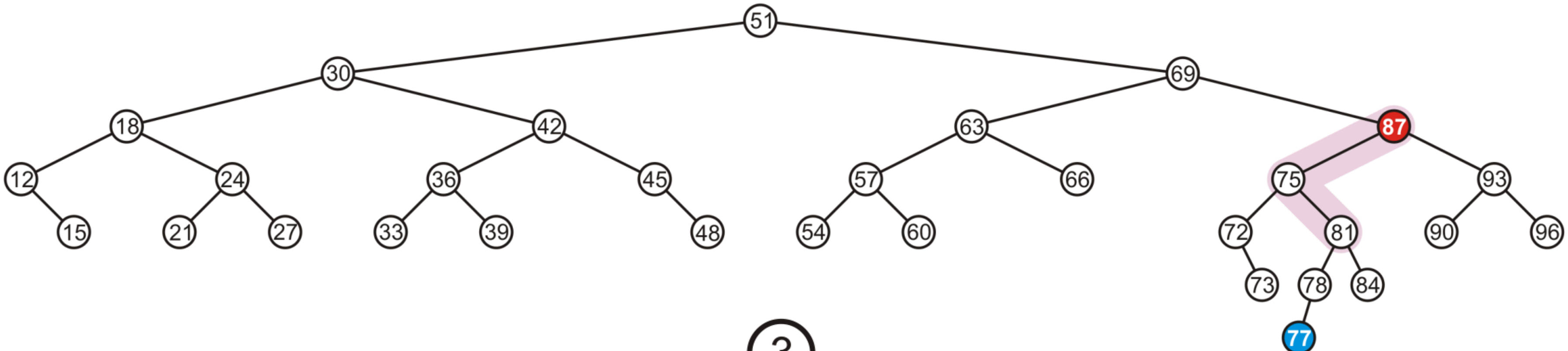
- A left-right imbalance

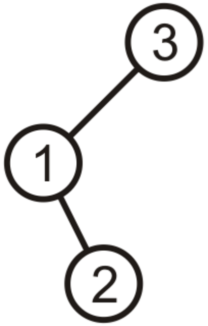


Insertion

The node 87 is unbalanced

- A left-right imbalance

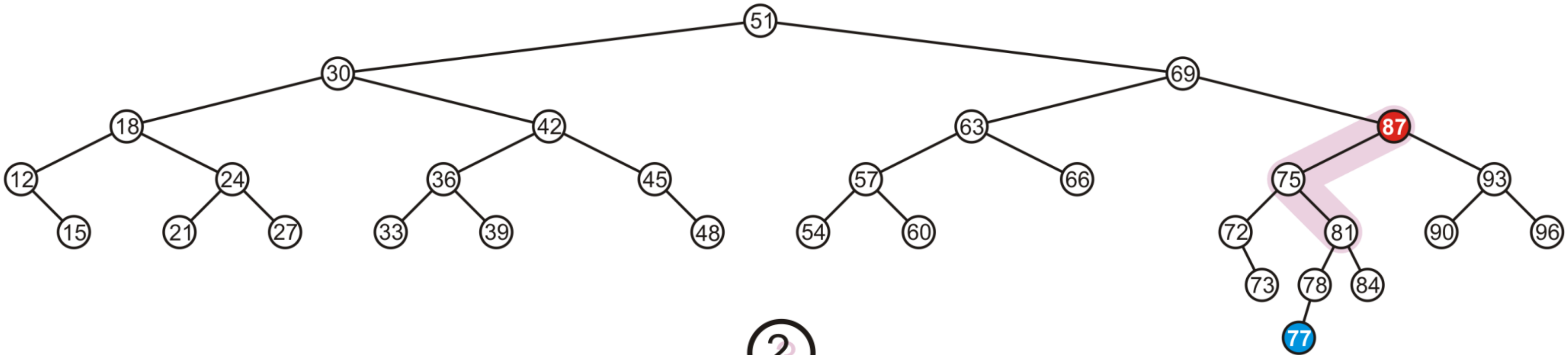


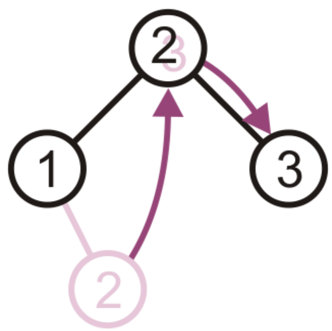


Insertion

The node 87 is unbalanced

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

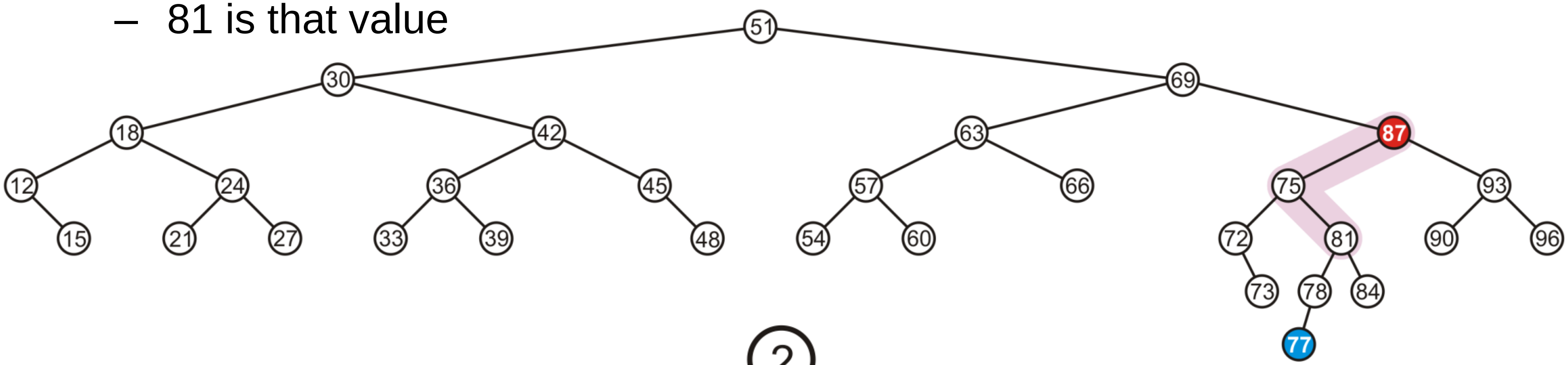


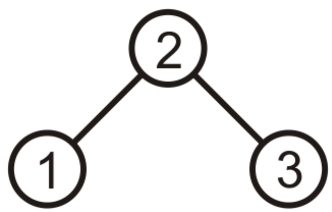


Insertion

The node 87 is unbalanced

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

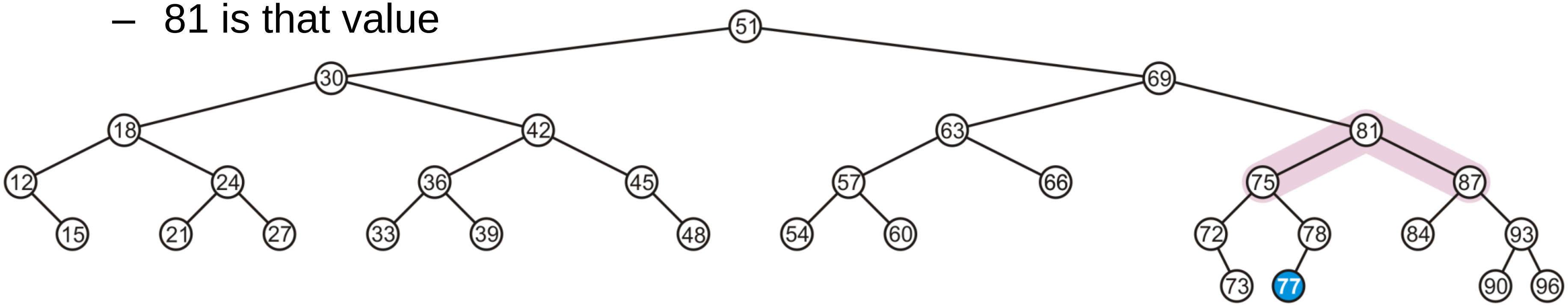




Insertion

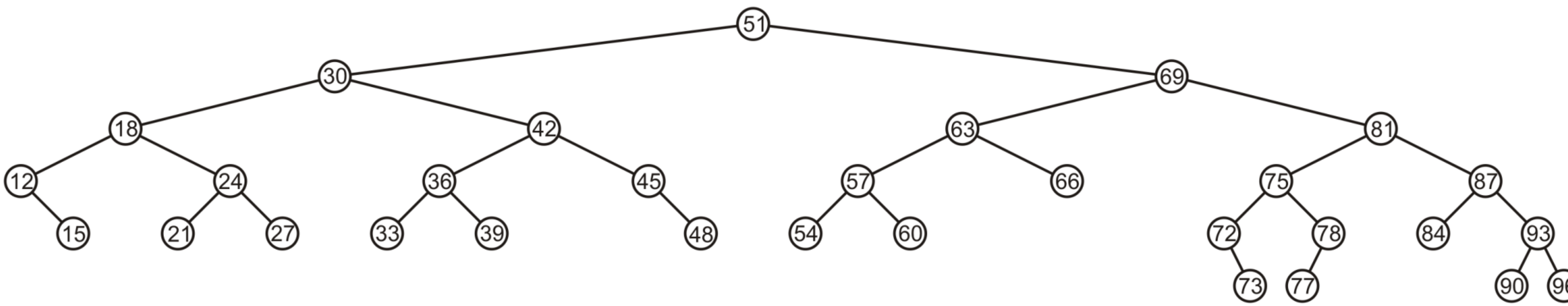
The node 87 is unbalanced

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



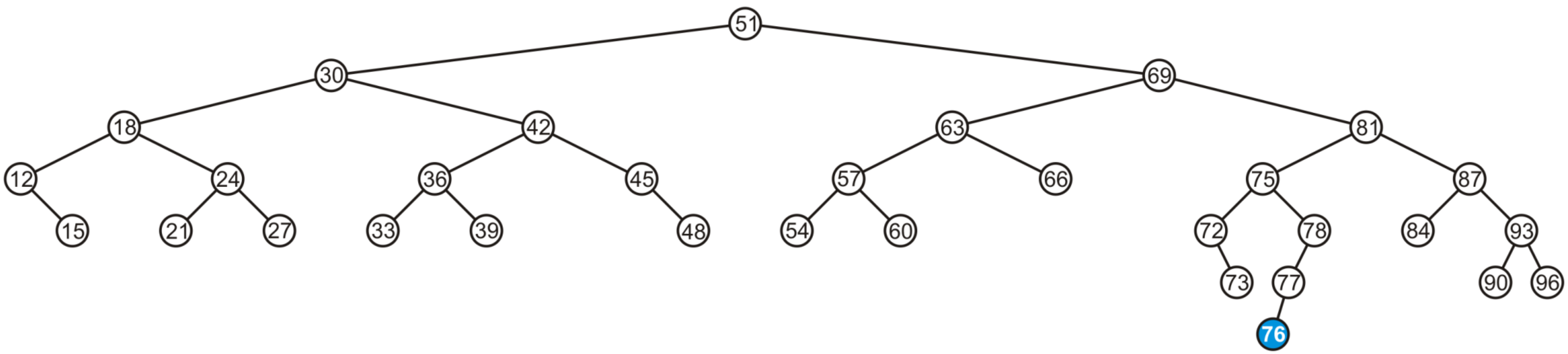
Insertion

The tree is balanced



Insertion

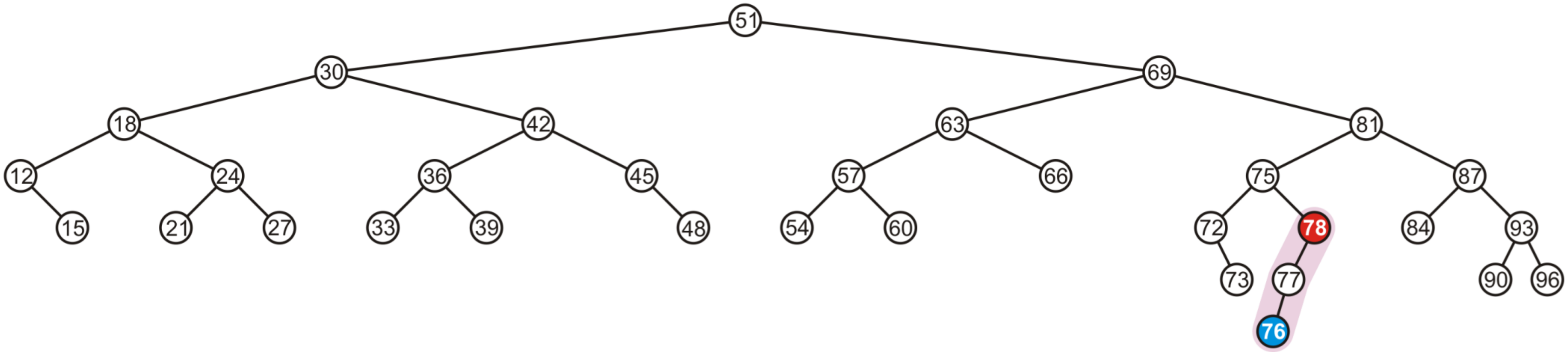
Insert 76



Insertion

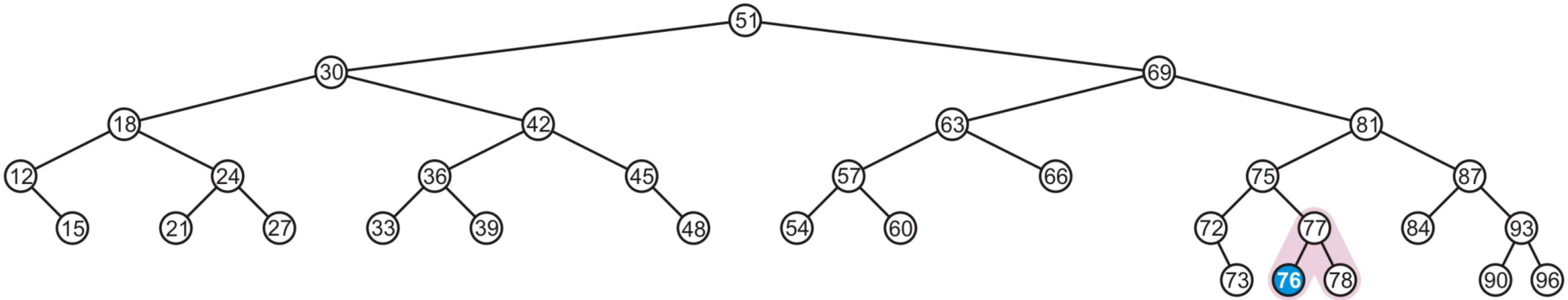
The node 78 is unbalanced

- A left-left imbalance



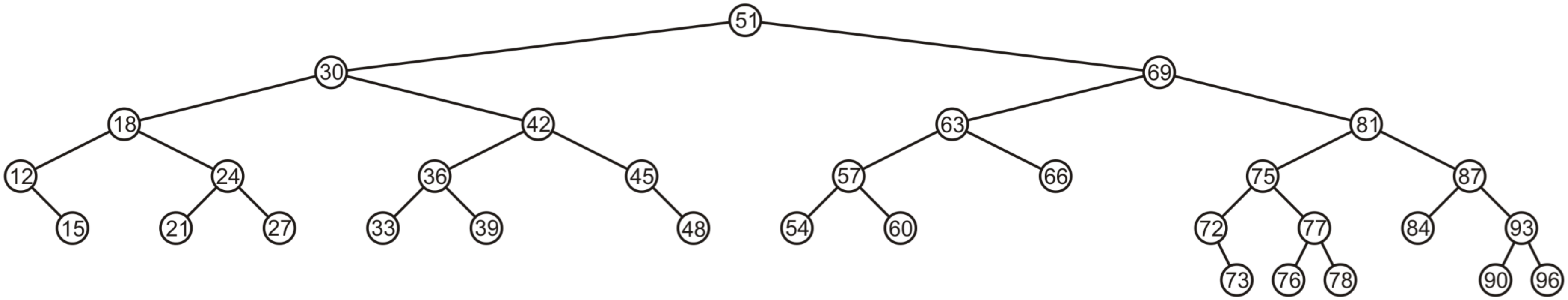
Insertion

- The node 78 is unbalanced
  - Promote 77



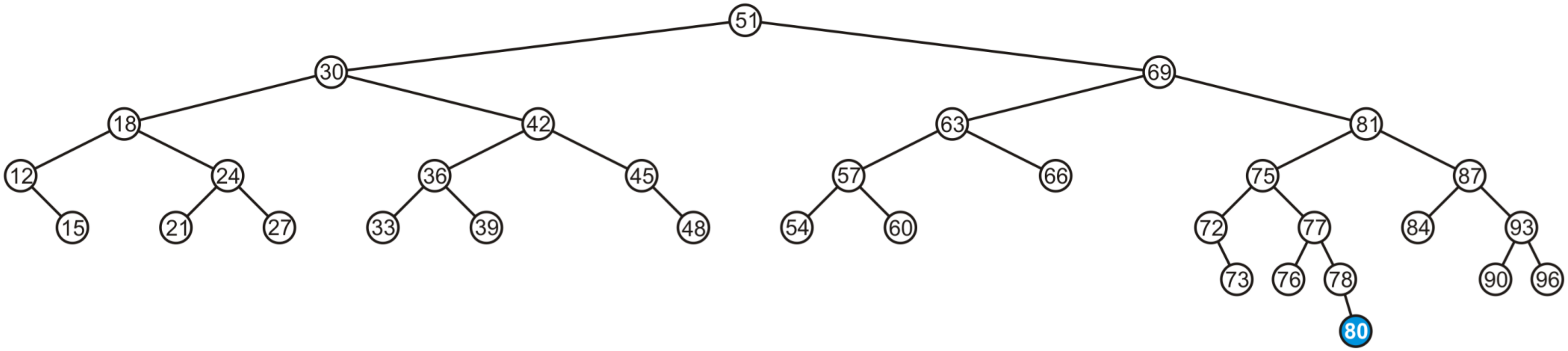
Insertion

Again, balanced



Insertion

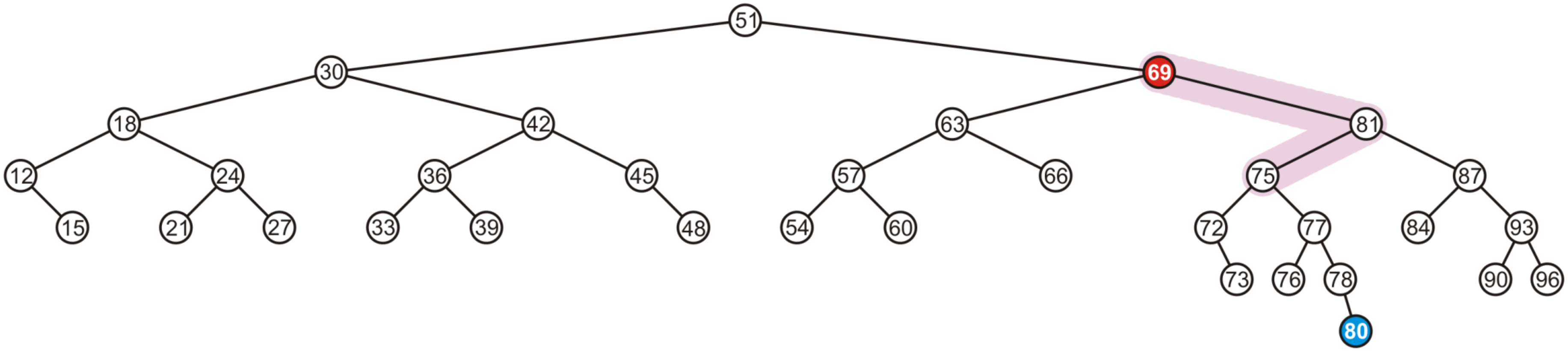
Insert 80



Insertion

The node 69 is unbalanced

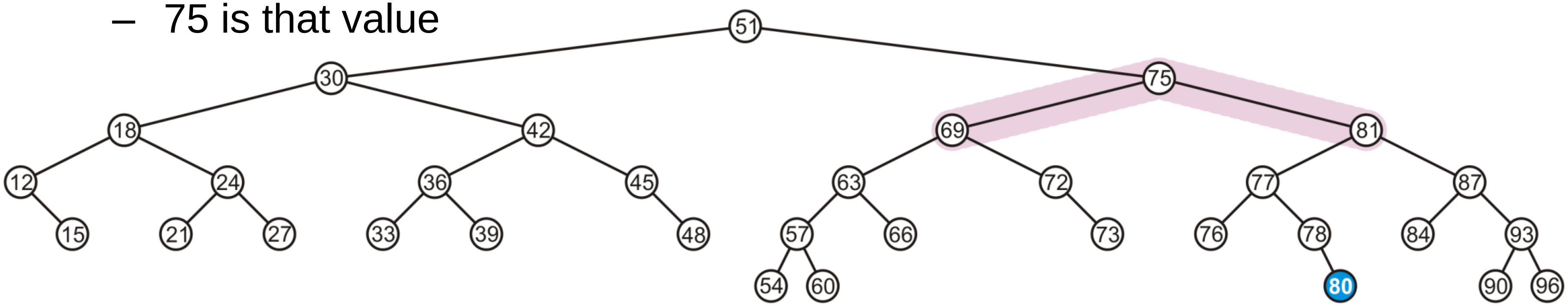
- A right-left imbalance
- Promote the intermediate node to the imbalanced node



Insertion

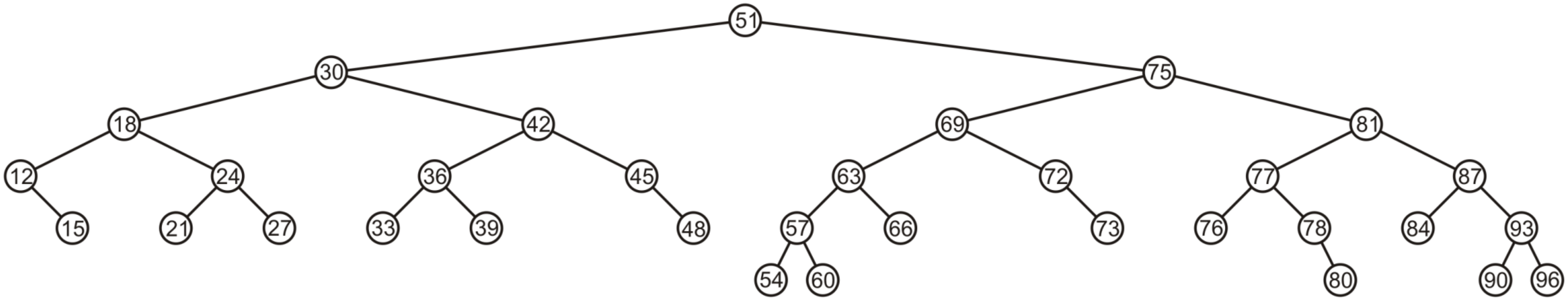
The node 69 is unbalanced

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



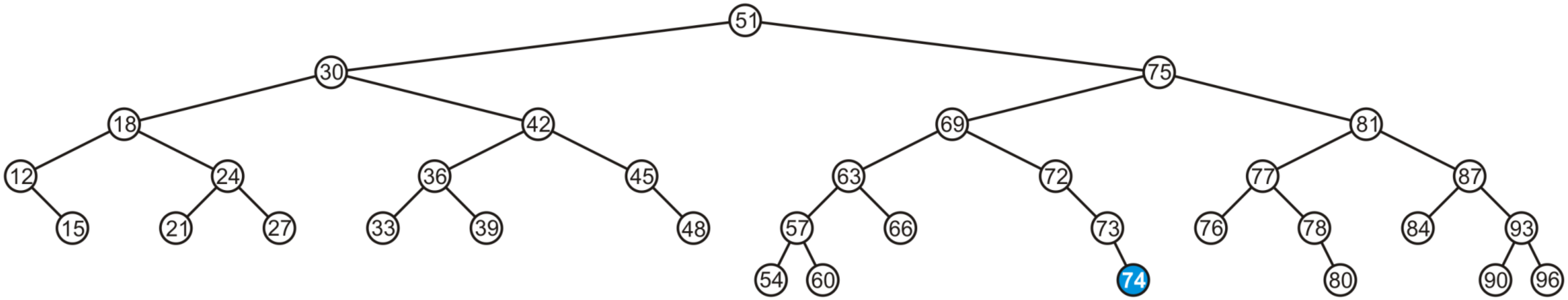
Insertion

Again, balanced



Insertion

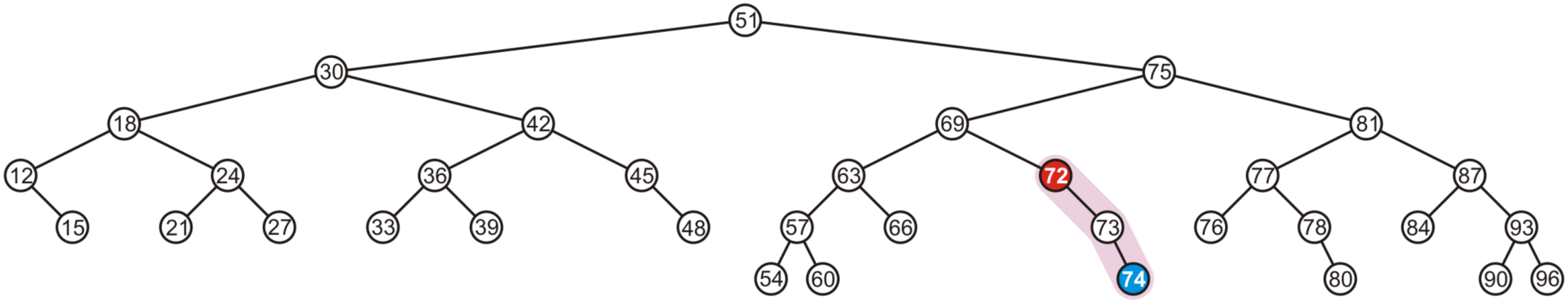
Insert 74



Insertion

The node 72 is unbalanced

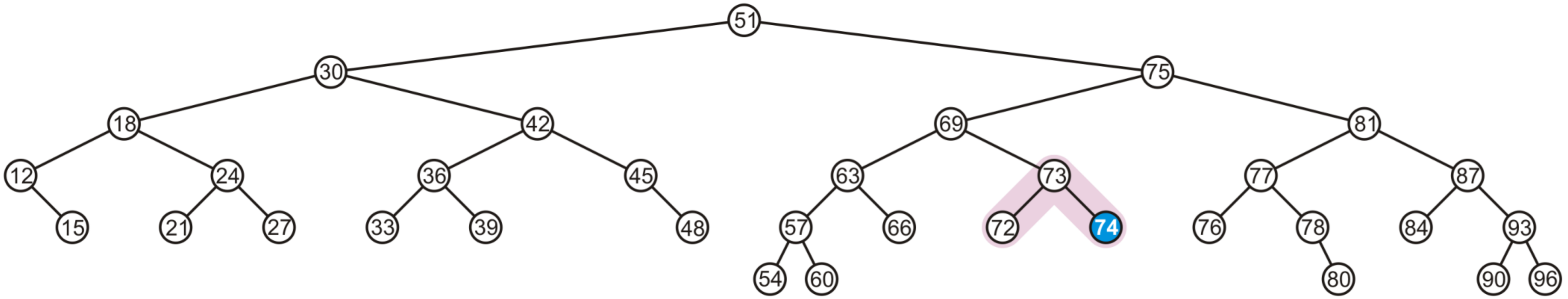
- A right-right imbalance
- Promote the intermediate node to the imbalanced node



Insertion

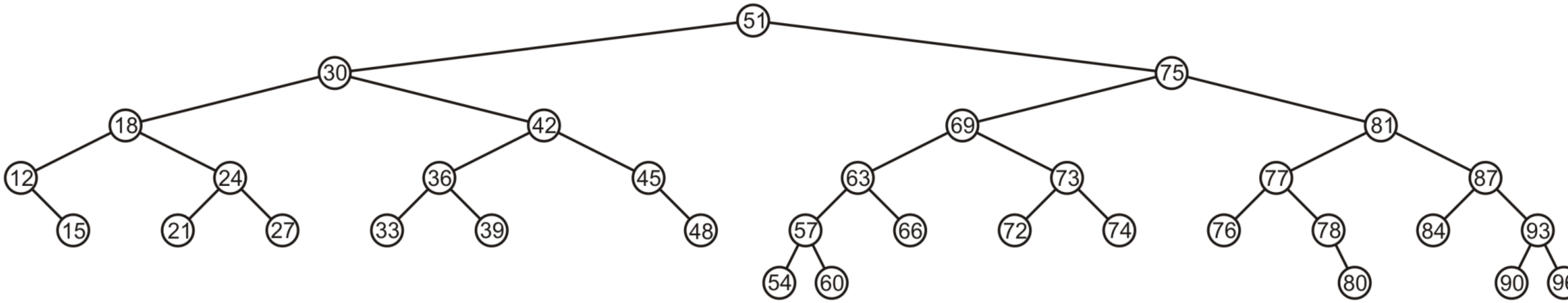
The node 72 is unbalanced

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



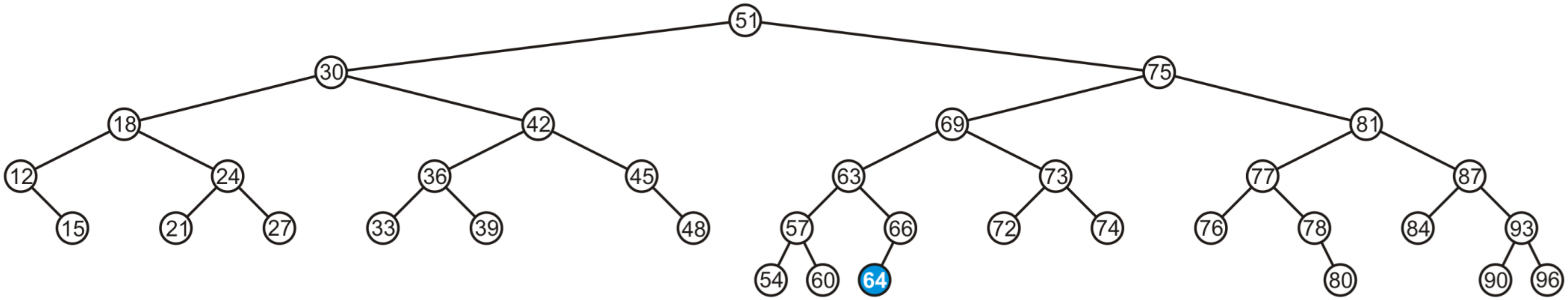
Insertion

Again, balanced



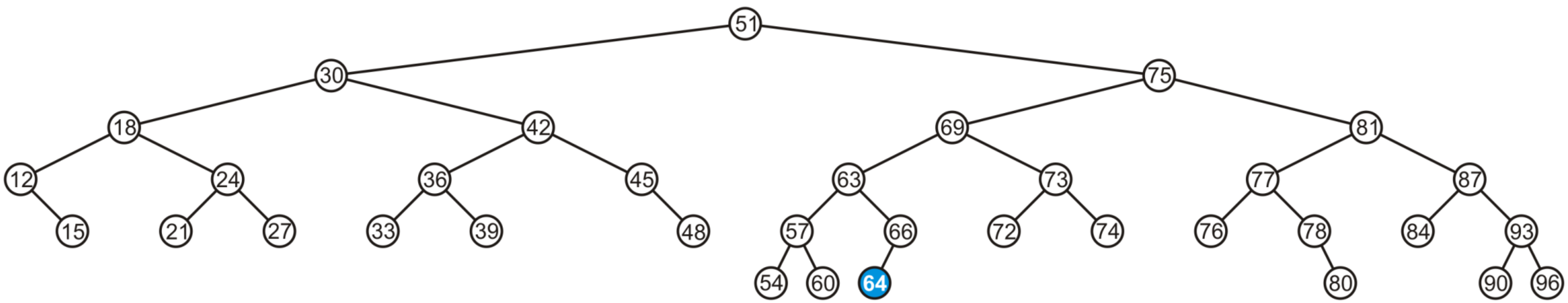
Insertion

Insert 64



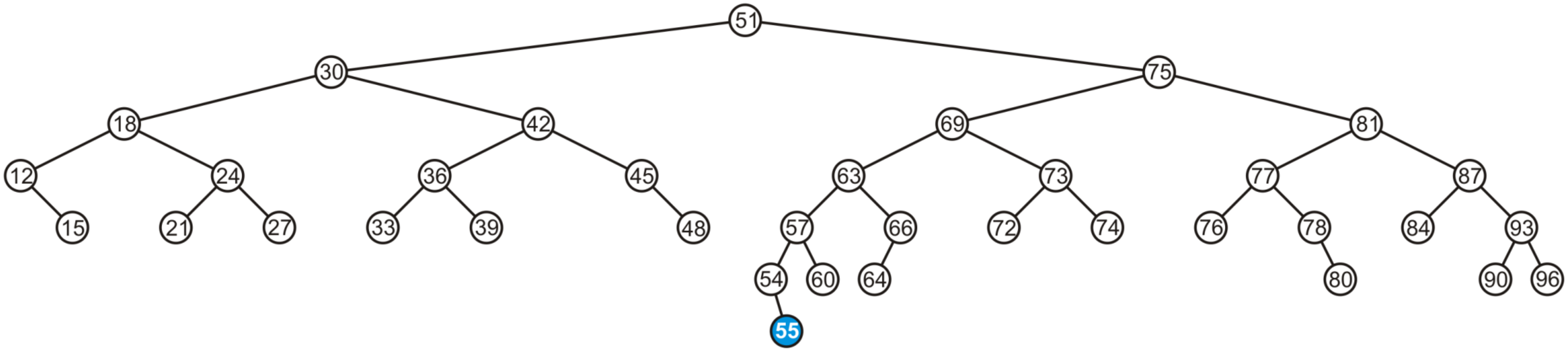
Insertion

This causes no imbalances



Insertion

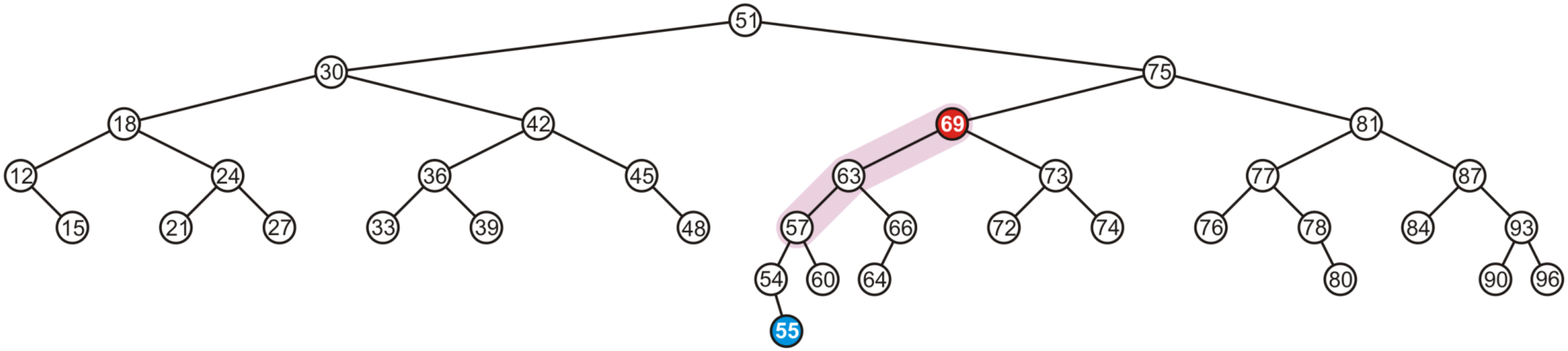
Insert 55



Insertion

The node 69 is imbalanced

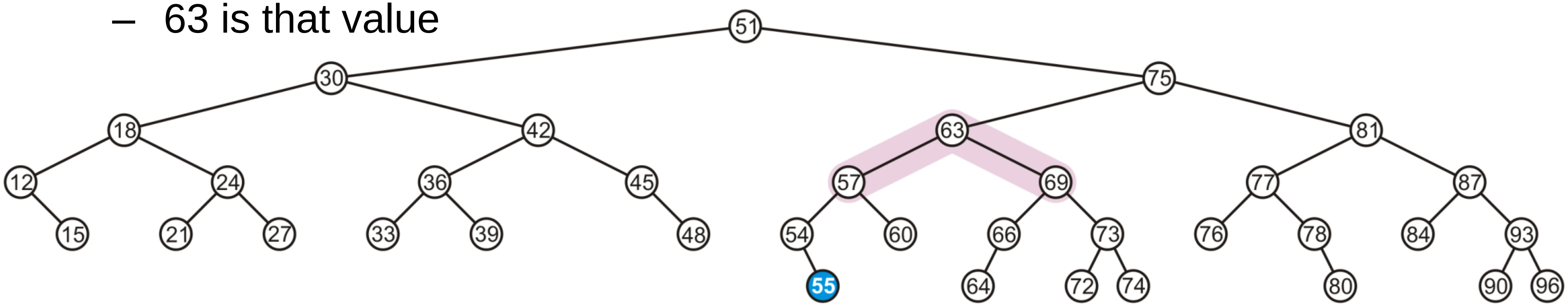
- A left-left imbalance
- Promote the intermediate node to the imbalanced node



Insertion

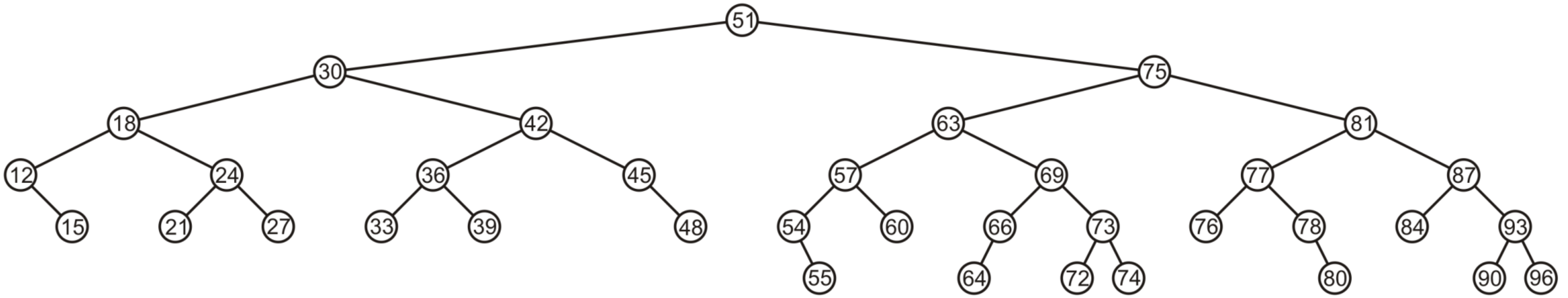
The node 69 is imbalanced

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



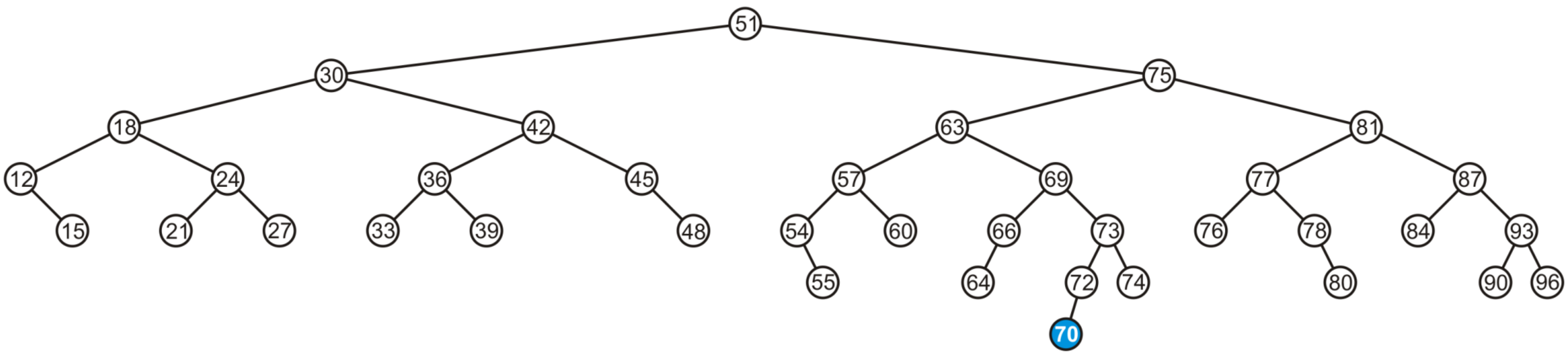
Insertion

The tree is now balanced



Insertion

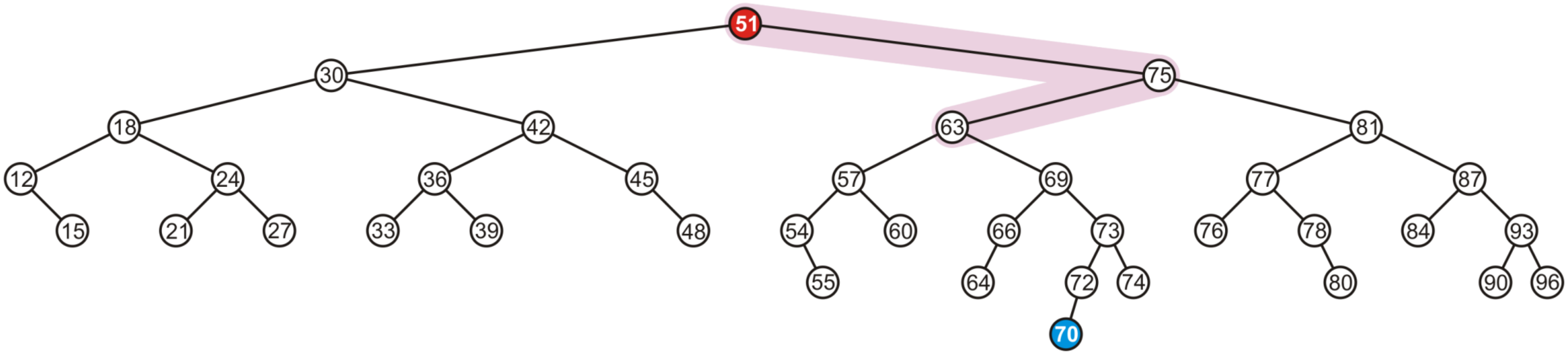
Insert 70



Insertion

The root node is now imbalanced

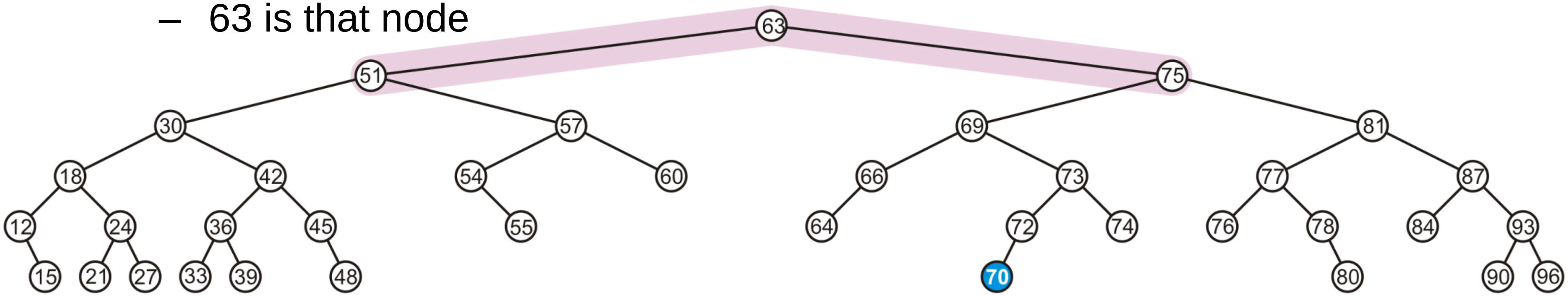
- A right-left imbalance
- Promote the intermediate node to the root



Insertion

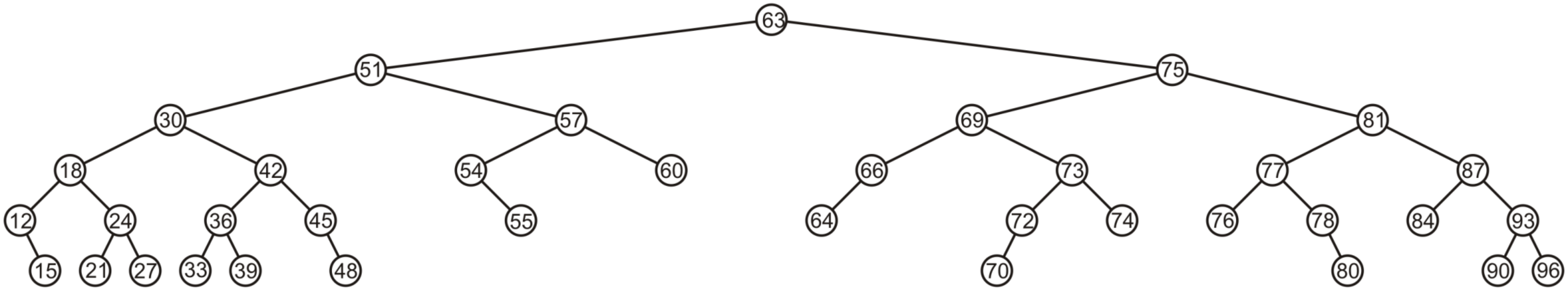
The root node is imbalanced

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



Insertion

The result is AVL balanced



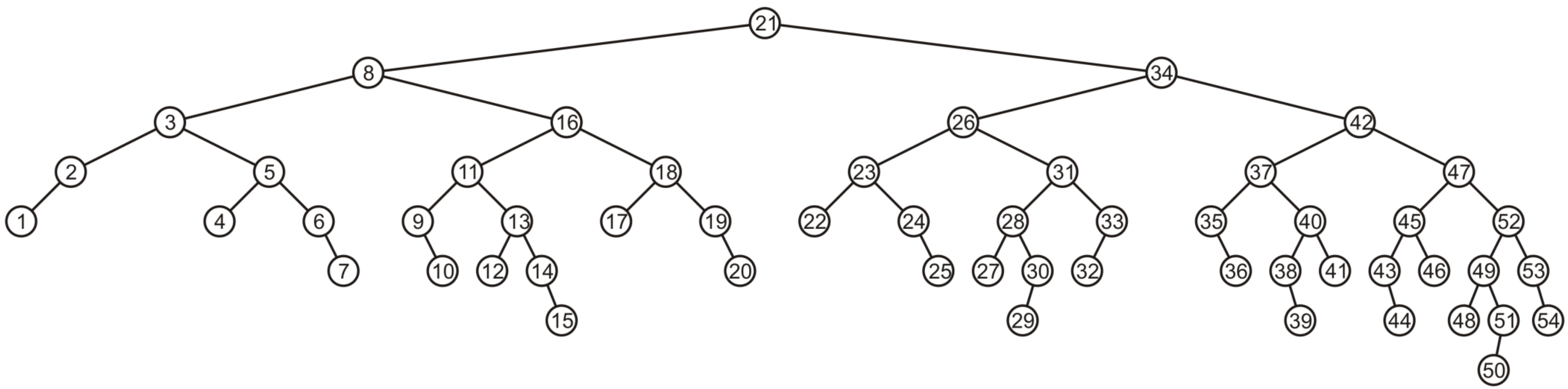
Erase

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

- Like insert, erase must check 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

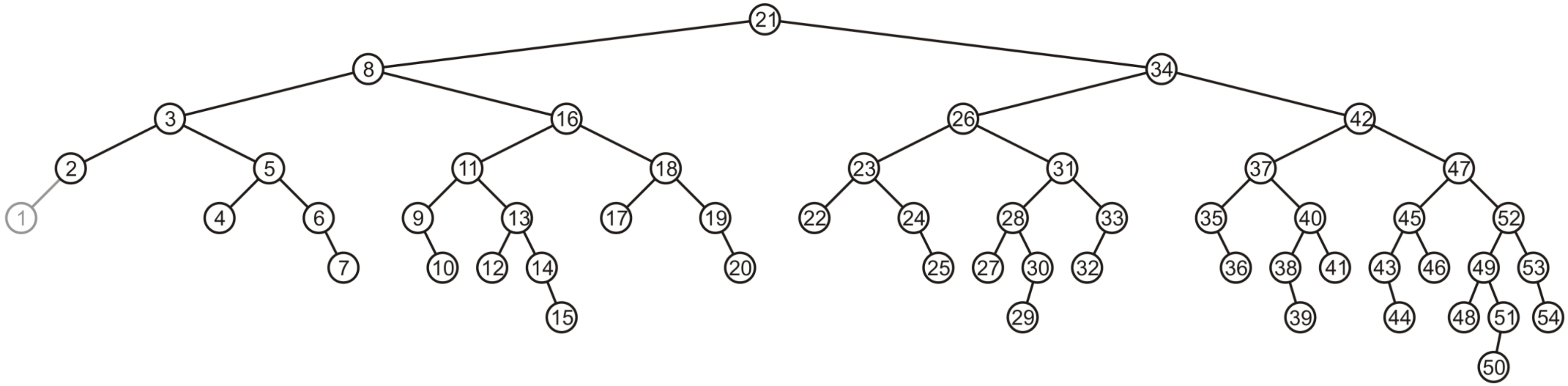
Erase

Consider the following AVL tree



Erase

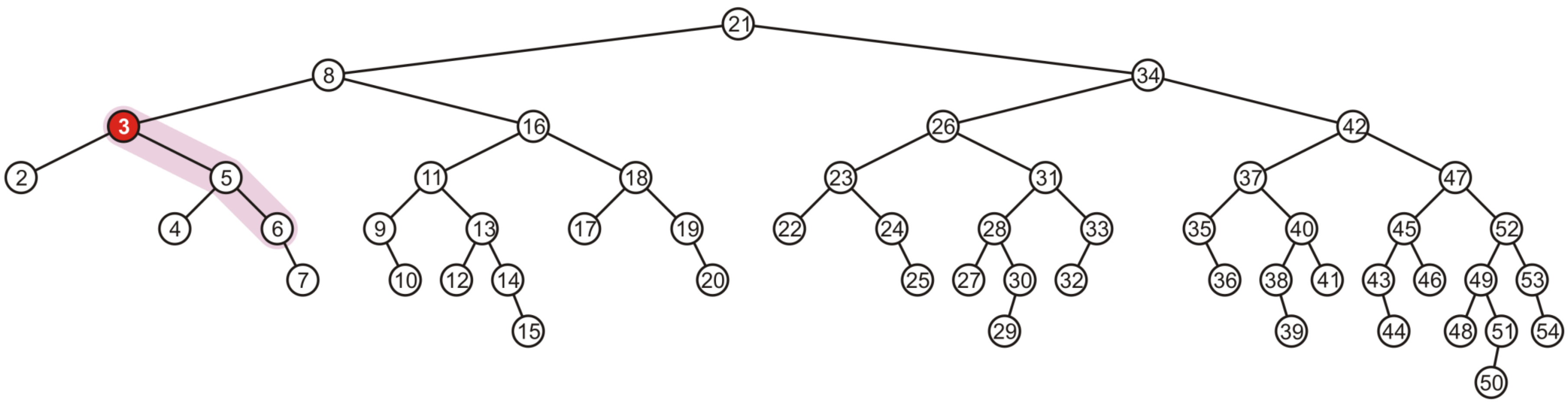
Suppose we erase the front node: 1



Erase

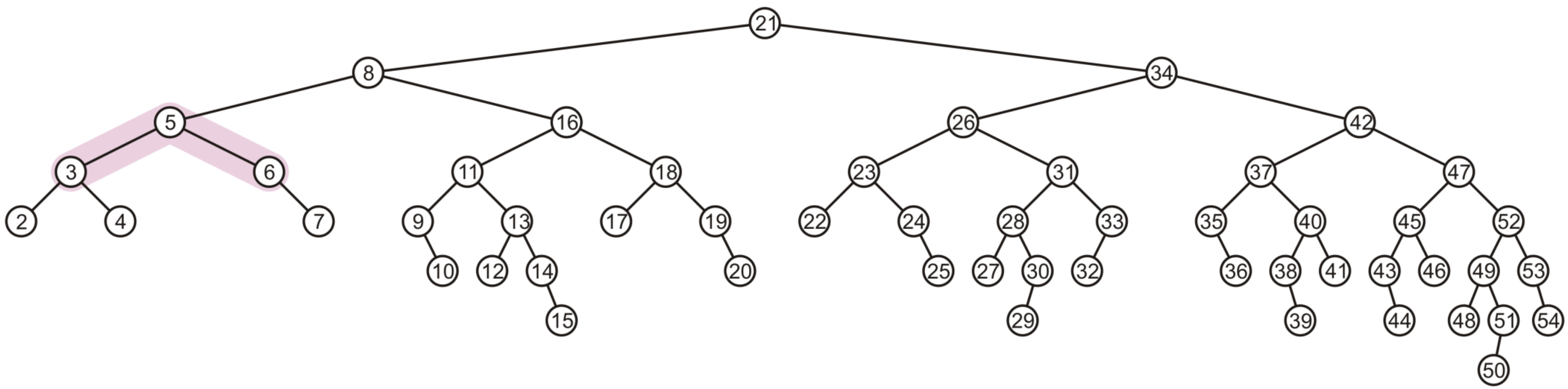
While its previous parent, 2, is not unbalanced, its grandparent 3 is

– The imbalance is in the right-right subtree



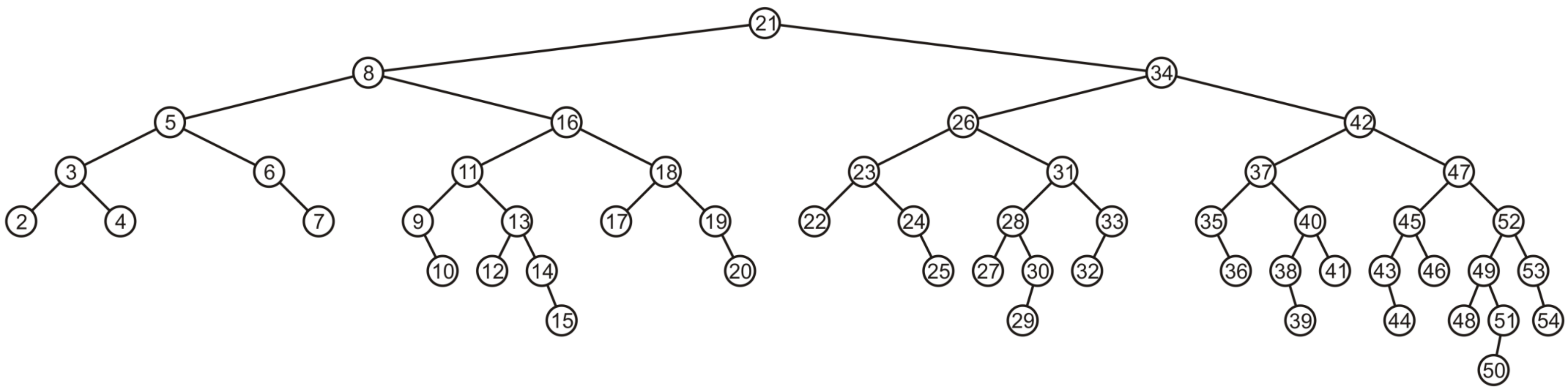
Erase

We can correct this with a simple balance



Erase

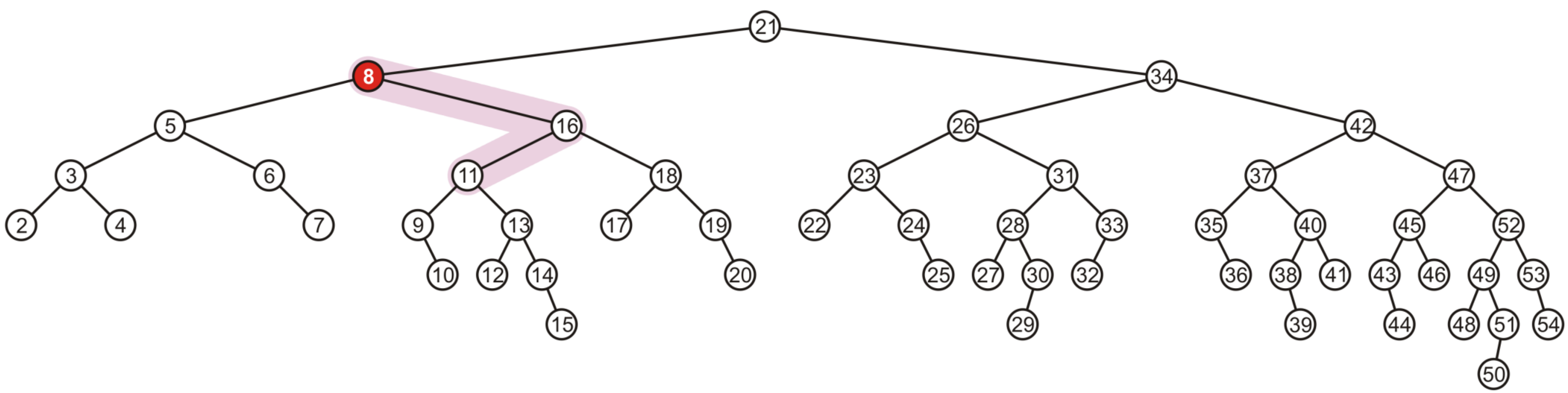
The node of that subtree, 5, is now balanced



Erase

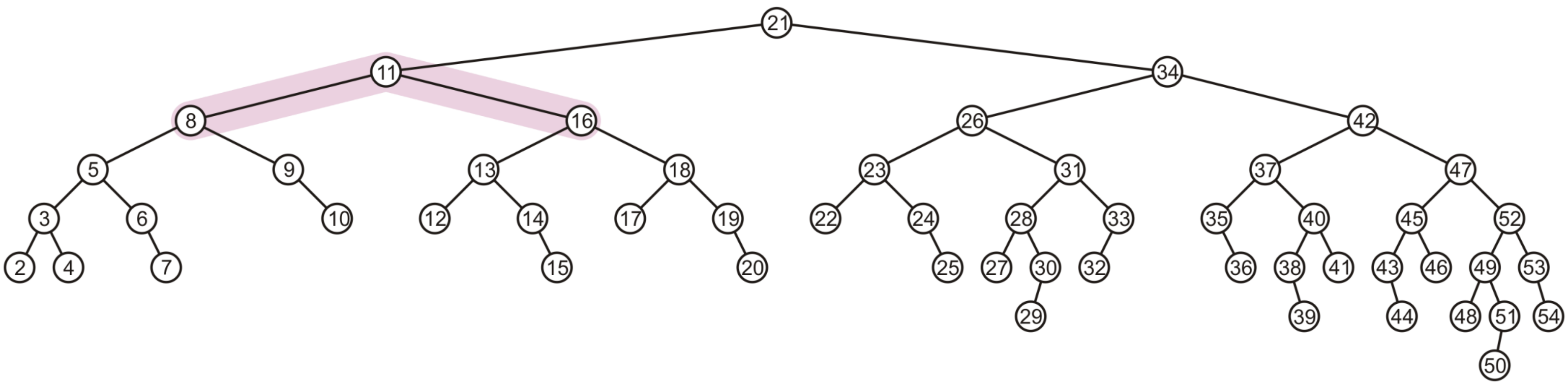
Recursing to the root, however, 8 is also unbalanced

- This is a right-left imbalance



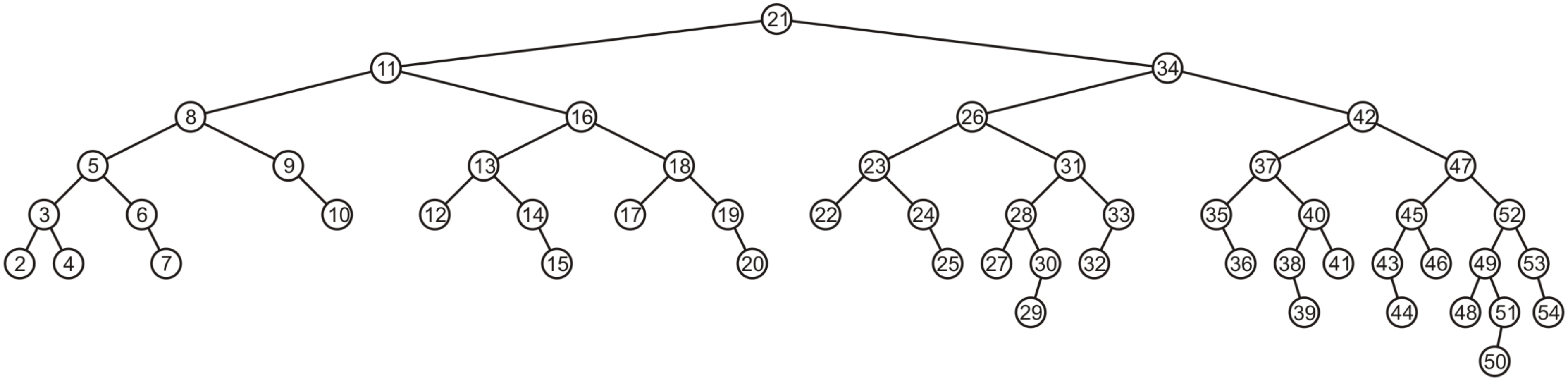
Erase

Promoting 11 to the root corrects the imbalance



Erase

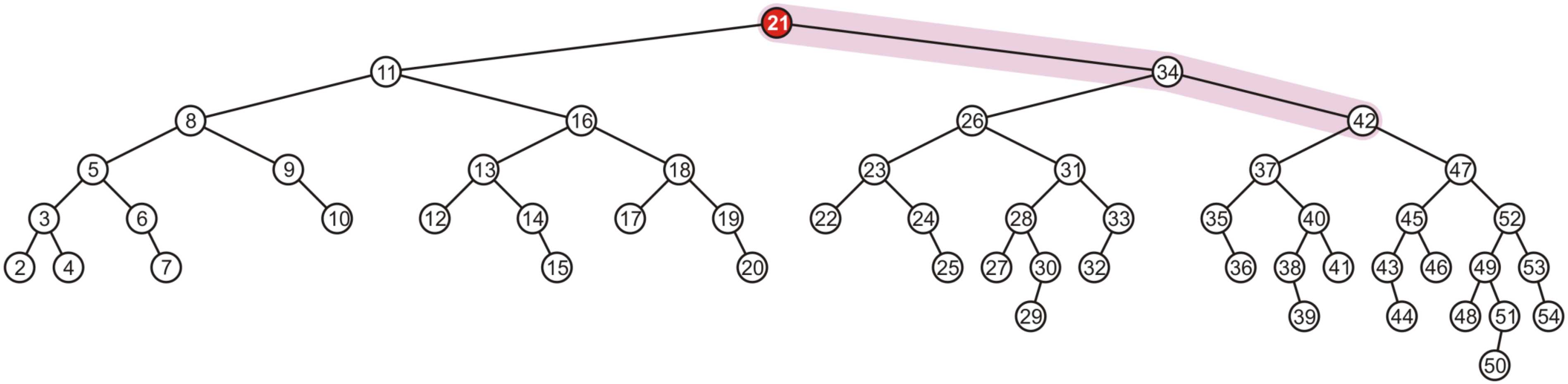
At this point, the node 11 is balanced



Erase

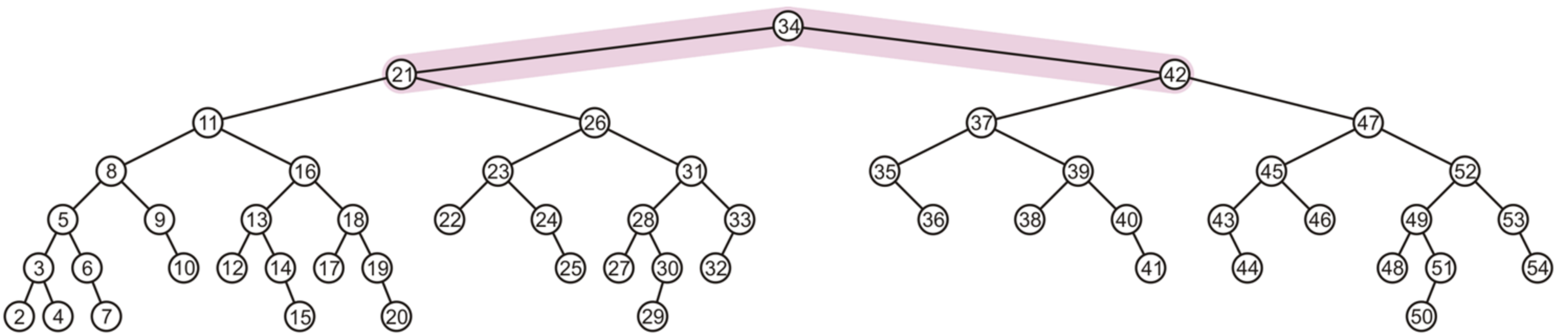
Still, the root node is unbalanced

– This is a right-right imbalance



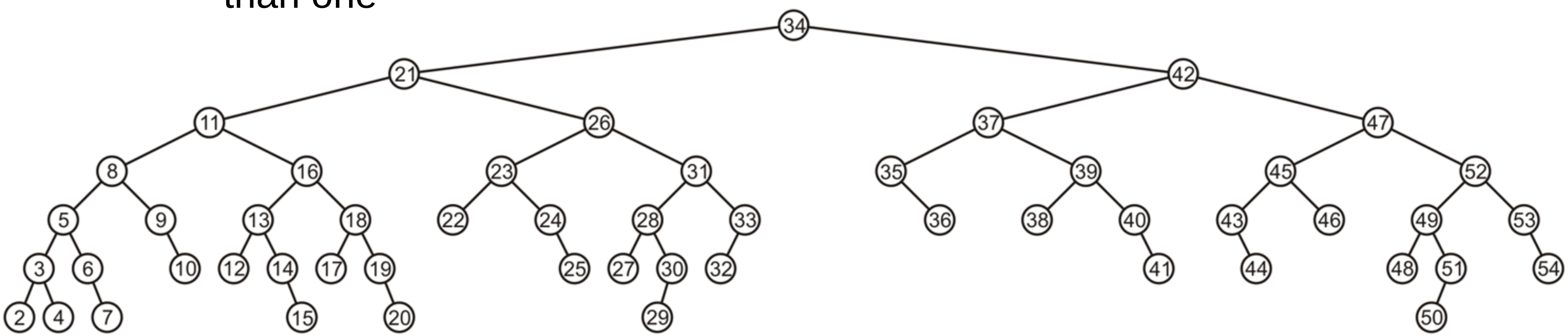
Erase

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



Time complexity

Insertion

- May require one correction to maintain balance
- Each correction require $\Theta(1)$ time

Erasion

- May require $\mathrm{O}(h)$ corrections to maintain balance
- Each correction require $\Theta(1)$ time
- Depth h is $\boldsymbol{\Theta}(\ \ln(n)\ )$
- So the time complexity is $\mathbf{O}(\ \ln(n)\ )$

AVL Trees as Arrays?

We previously saw that:

- Complete tree can be stored using an array using $\Theta(n)$ memory
- An arbitrary tree of n nodes requires $\mathbf{O}(2^n)$ memory

Is it possible to store an AVL tree as an array and not require exponentially more memory?

AVL Trees as Arrays?

Recall that in the worst case, an AVL tree of n nodes has a height at most

$$\log_\phi(n) - 1.3277$$

Such a tree requires an array of size

$$2^{\log_\phi(n) - 1.3277 + 1} - 1$$

We can rewrite this as

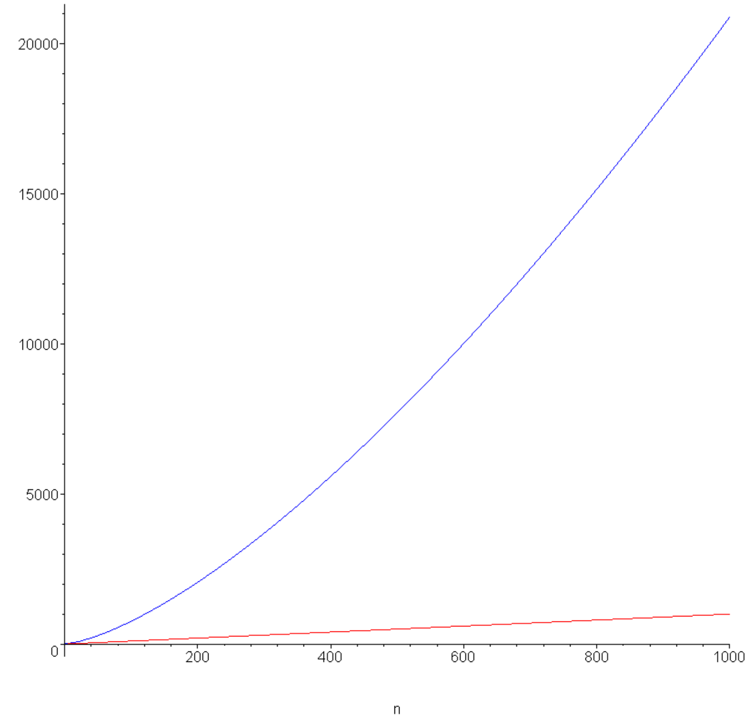
$$2^{-0.3277}\, n^{\log_\phi(2)} \approx 0.7968\, n^{1.44}$$

Thus, we would require $\mathbf{O}(n^{1.44})$ memory

AVL Trees as Arrays?

While the polynomial behaviour of $n^{1.44}$ is not as bad as exponential behaviour, it is still reasonably sub-optimal when compared to the linear growth associated with link-allocated trees

Here we see n and $n^{1.44}$ on $[0, 1000]$



Summary

AVL trees

- Balance: difference in subtree heights is 0 or 1
- Insertion and erasion may require the tree to be rebalanced by tree rotations