AVL Trees

Outline

Background

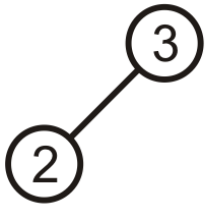
Define height balancing

Maintaining balance within a tree

- AVL trees
- Difference of heights
- Maintaining balance after insertions and Deletes
- Can we store AVL trees as arrays?

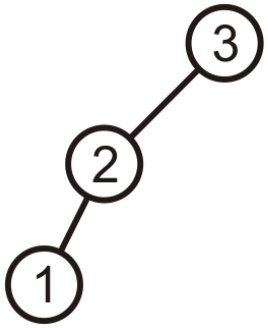
Prototypical Examples

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



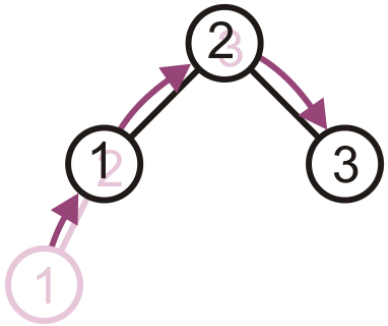
Prototypical Examples

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



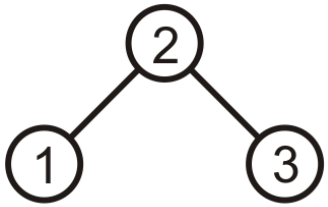
Prototypical Examples

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



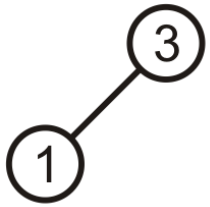
Prototypical Examples

The result is a perfect, though trivial tree



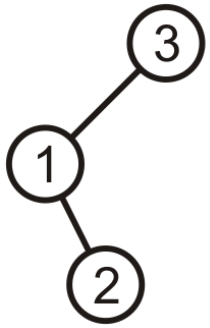
Prototypical Examples

Alternatively, given this tree, insert 2



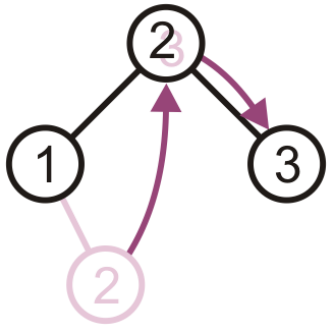
Prototypical Examples

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



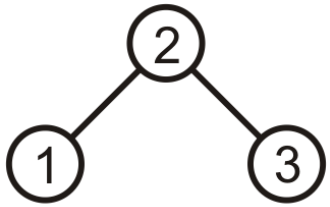
Prototypical Examples

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

We will focus on the first strategy: AVL trees

- Named after Adelson-Velskii and Landis

Balance 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

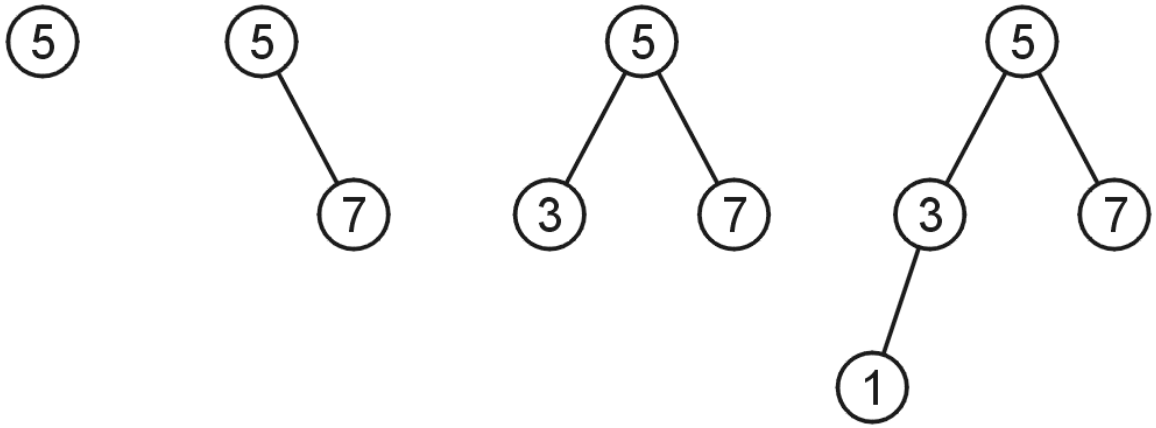
A binary search tree is said to be AVL balanced if:

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

- **Balance Factor:**
- **A node's *balance factor* is:**
  - **the left subtree height minus the right subtree height, which is 1, 0, or -1 in an AVL tree.**

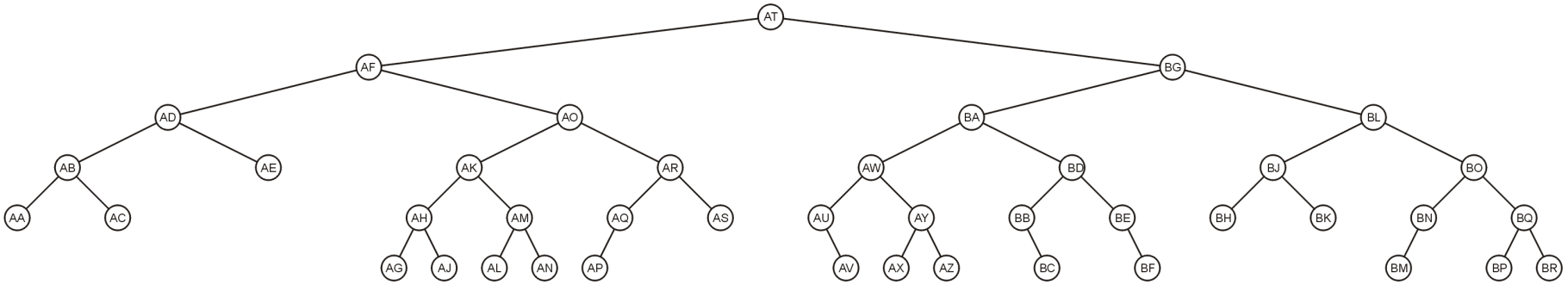
AVL Trees

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



AVL Trees

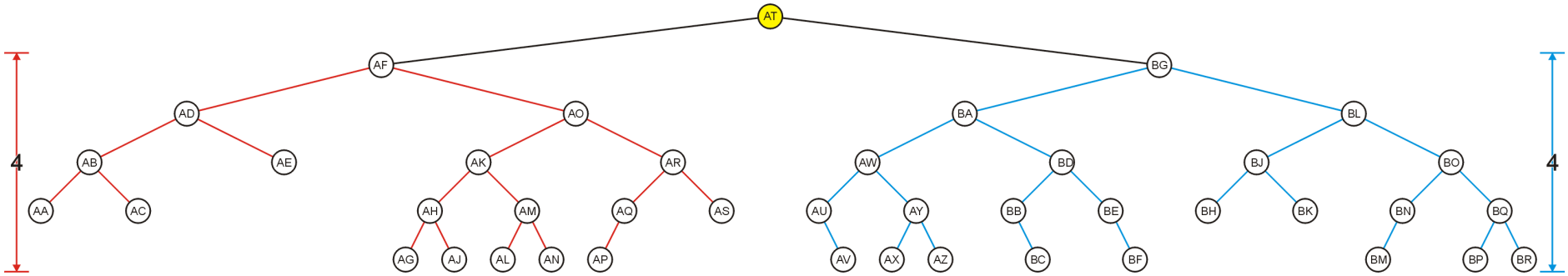
Here is a larger AVL tree (42 nodes):



AVL Trees

The root node is AVL-balanced:

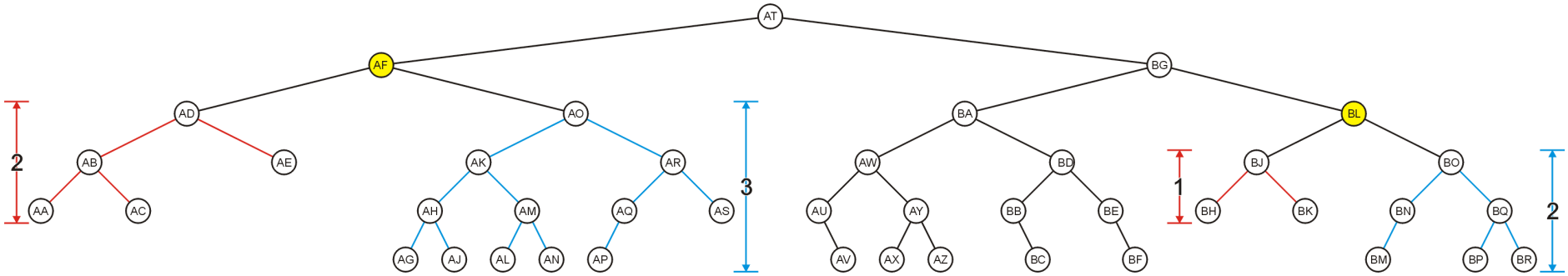
- Both sub-trees are of height 4:



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 a perfect binary tree with $2^{h+1} - 1$ nodes**

- What is an lower bound?

Height of an AVL Tree

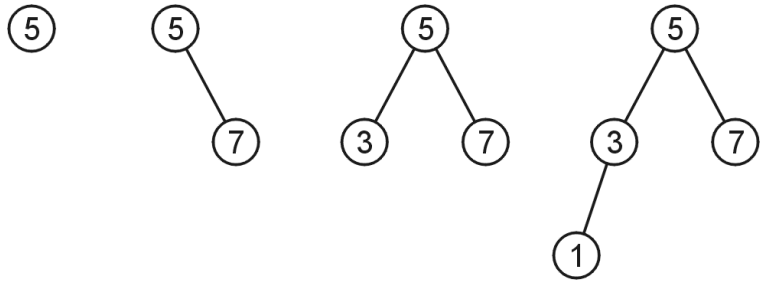
Let $\mathrm{F}(h)$ be the fewest number of nodes in a tree of height h

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

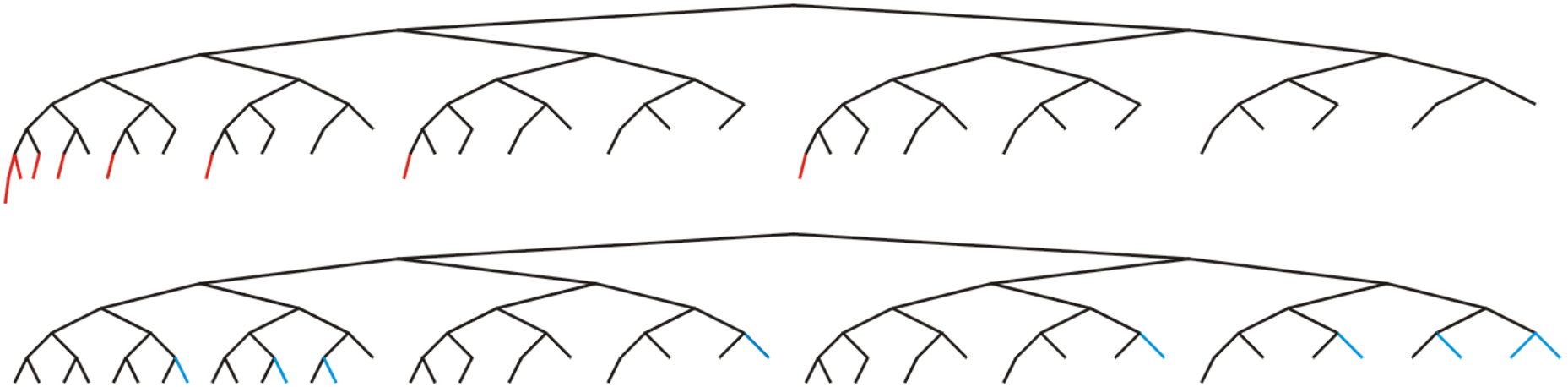
Height of an AVL Tree

So the height is always less than 1.440logn

That is pretty good for an AVL tree!

Height of an AVL Tree

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



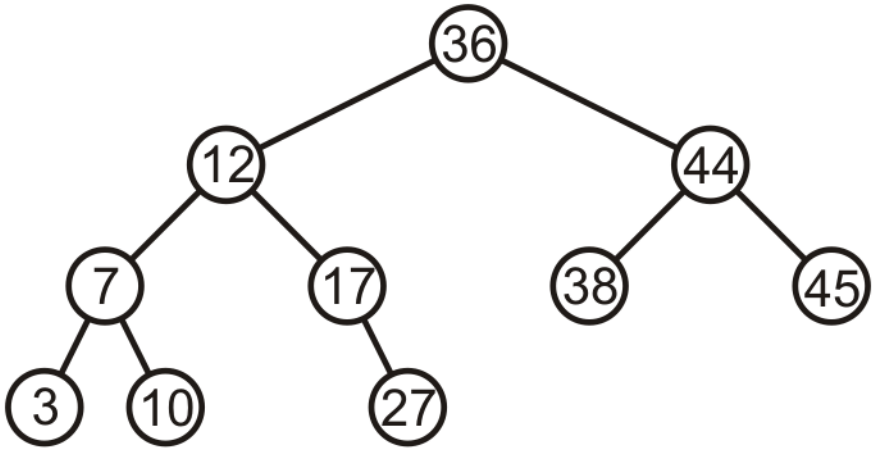
Maintaining Balance

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

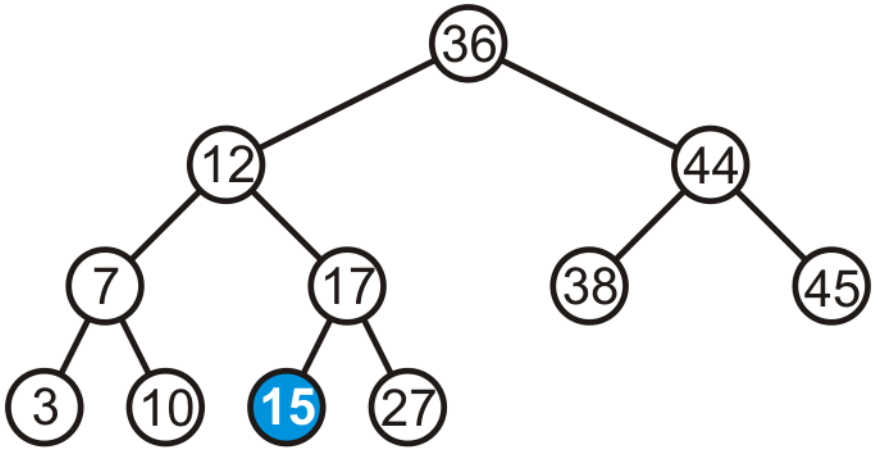
Consider this AVL tree



Maintaining Balance

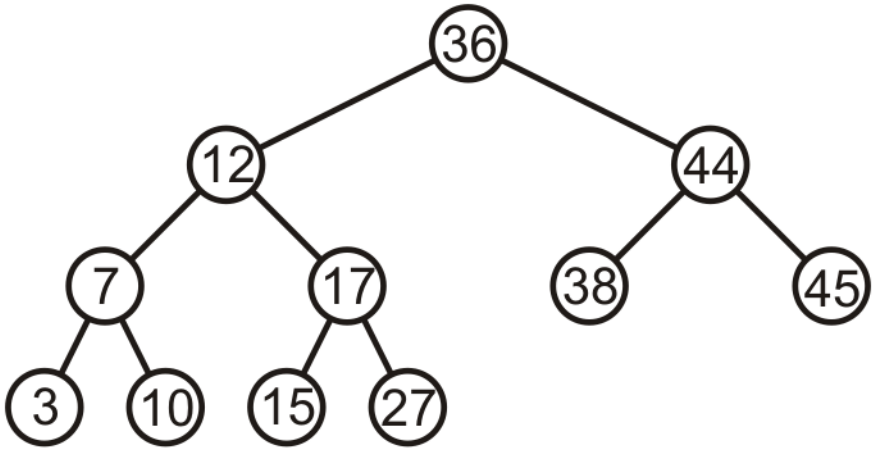
Consider inserting 15 into this tree

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



Maintaining Balance

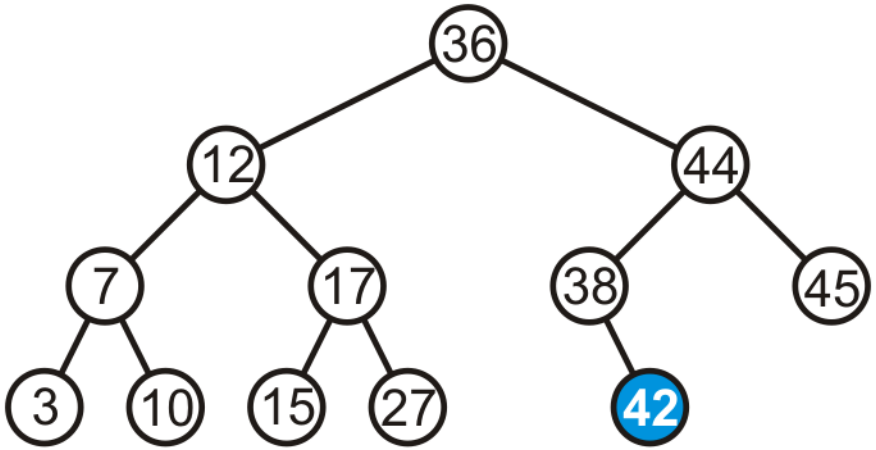
The tree remains balanced



Maintaining Balance

Consider inserting 42 into this tree

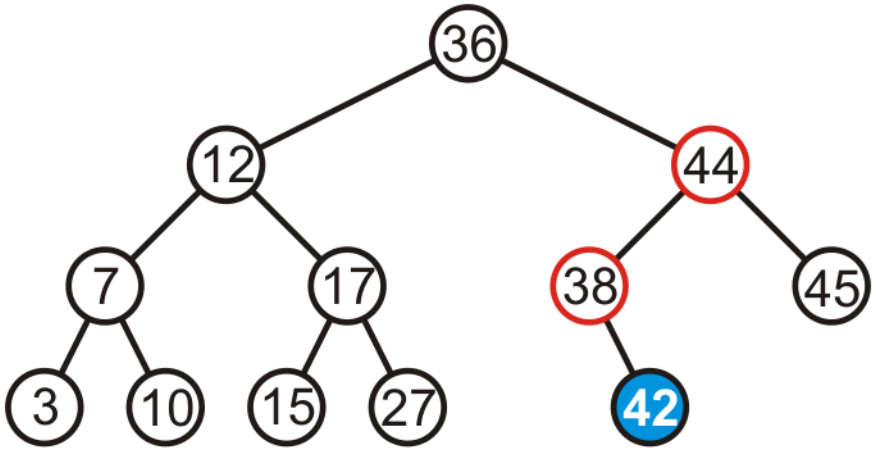
- In this case, the heights of none of the trees change



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

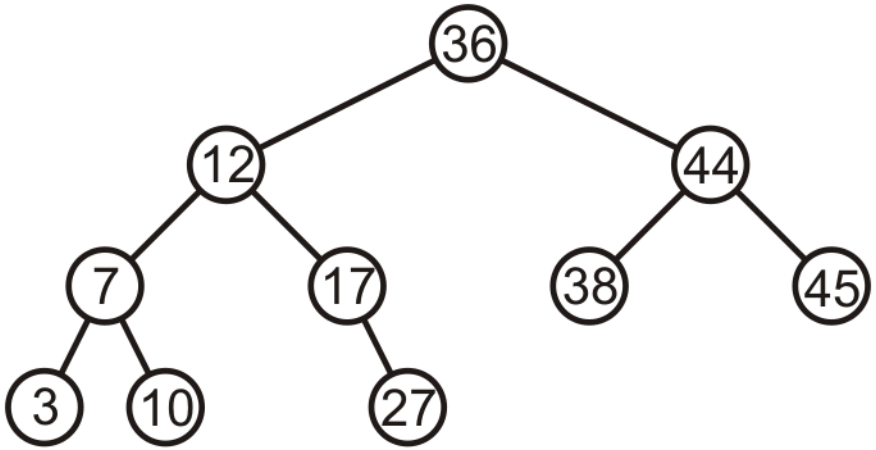
Only **insert and delete** may change the height

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

Maintaining Balance

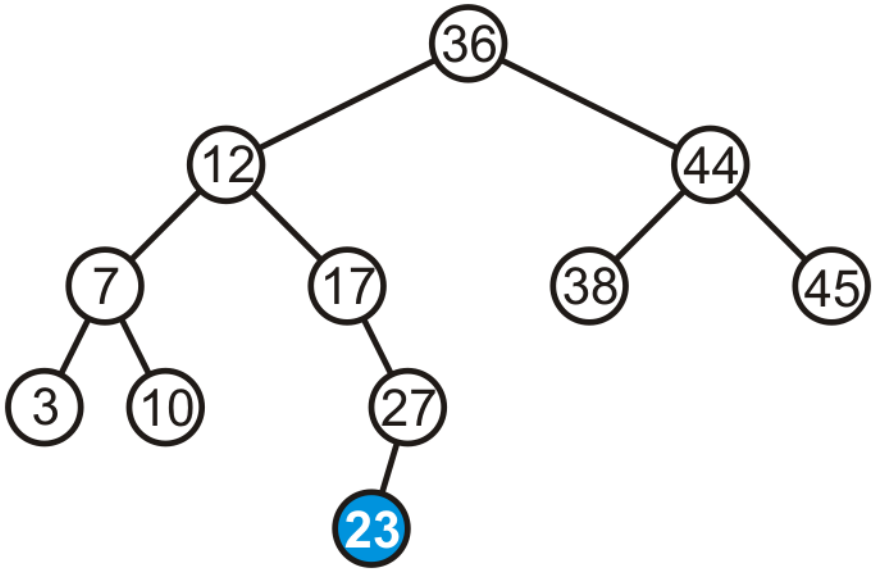
If a tree is AVL balanced, for an insertion to cause an imbalance:

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



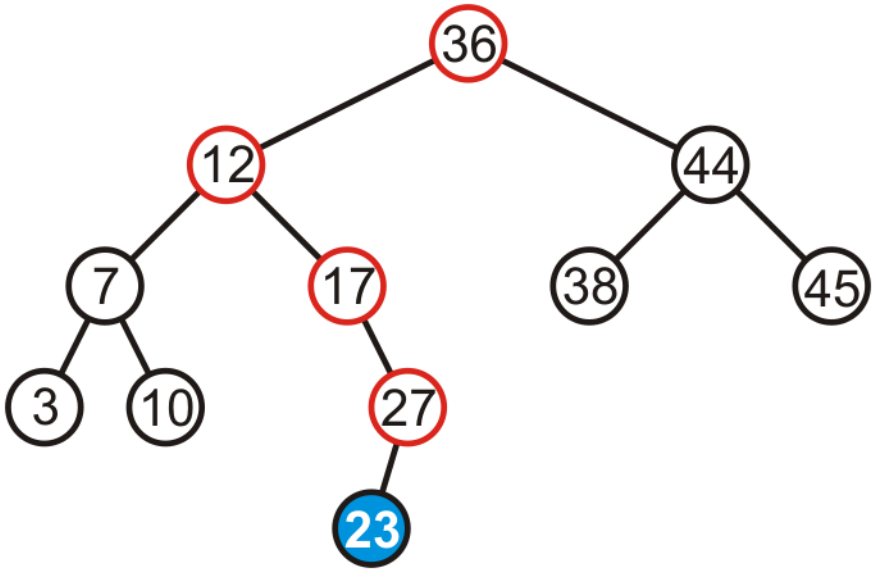
Maintaining Balance

Suppose we insert 23 into our initial tree



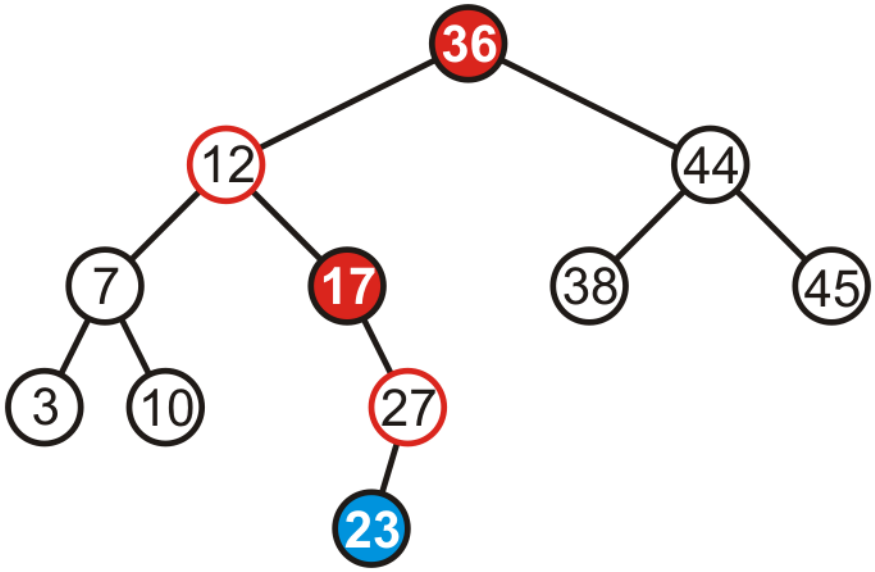
Maintaining Balance

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



Maintaining Balance

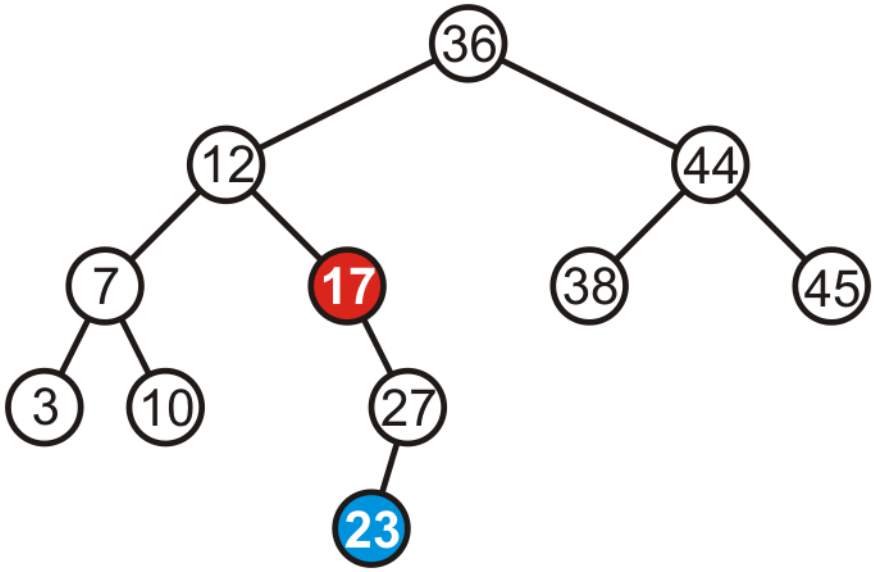
However, only two of the nodes are unbalanced: 17 and 36



Maintaining Balance

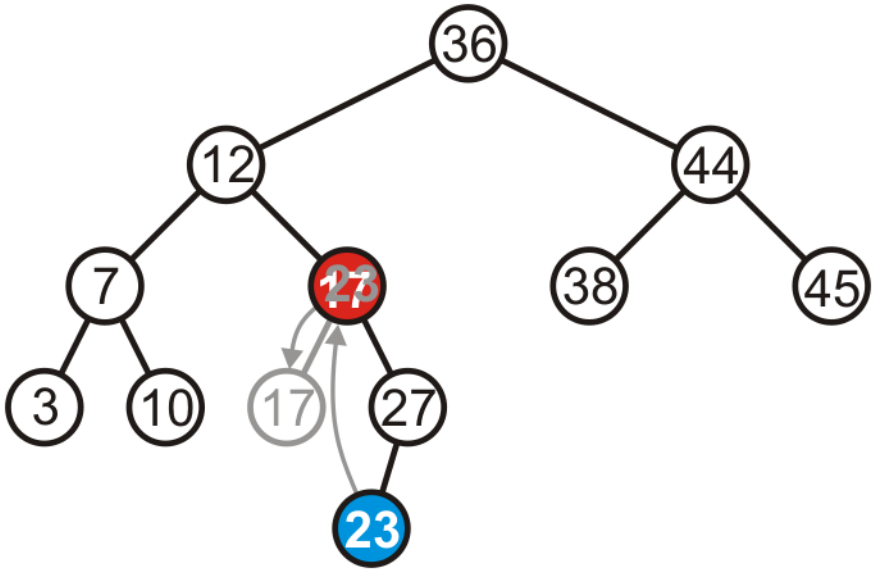
However, only two of the nodes are unbalanced: 17 and 36

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



Maintaining Balance

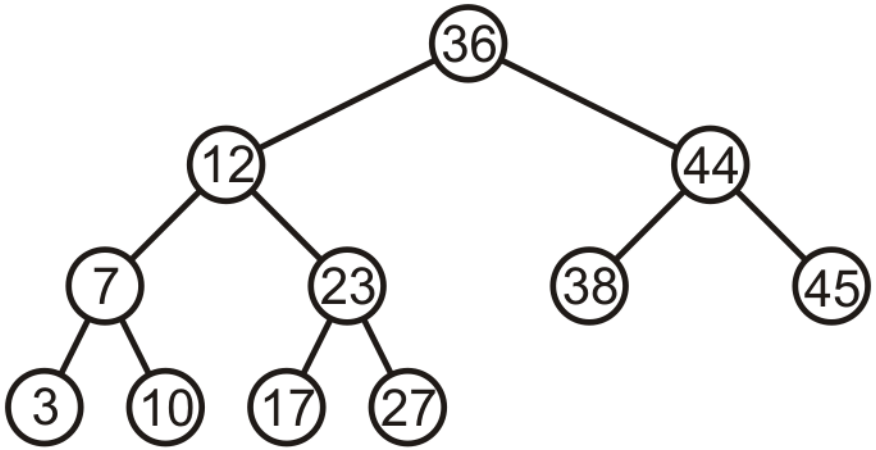
We can promote 23 to where 17 is, and make 17 the left child of 23



Maintaining Balance

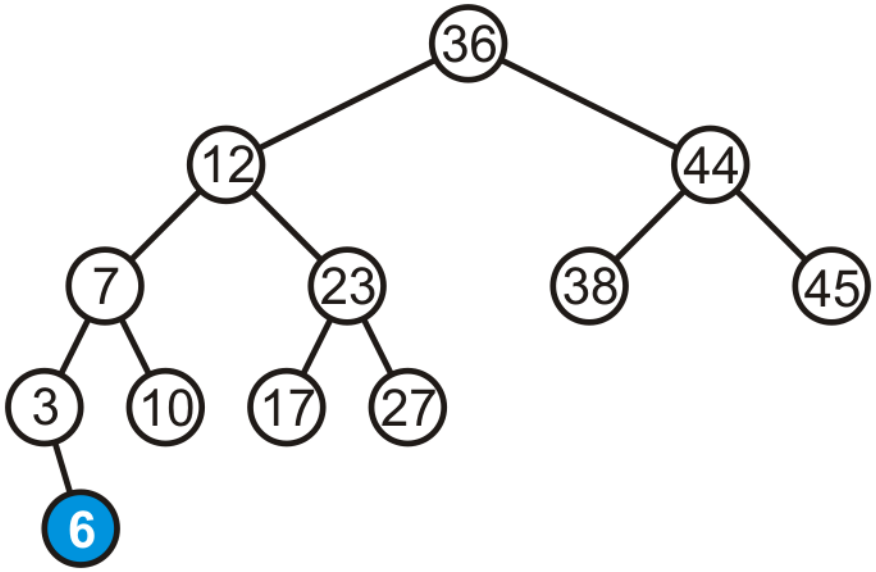
Thus, that node is no longer unbalanced

- Incidentally, neither is the root now balanced again, too



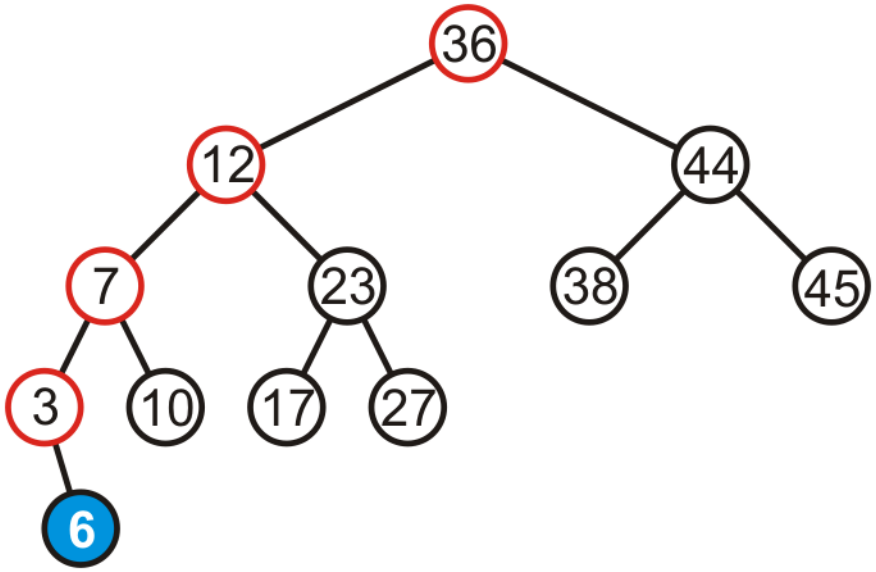
Maintaining Balance

Consider adding 6:



Maintaining Balance

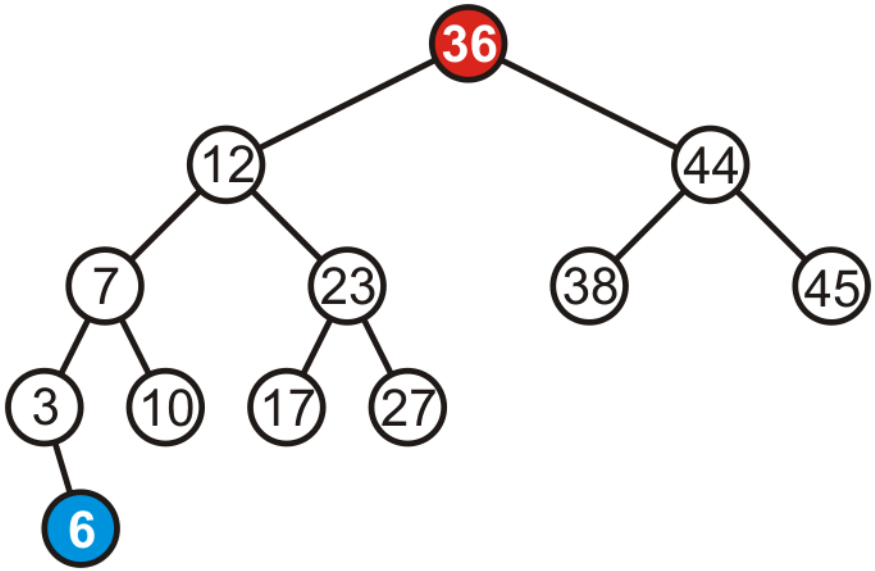
The height of each of the trees in the path back to the root are increased by one



Maintaining Balance

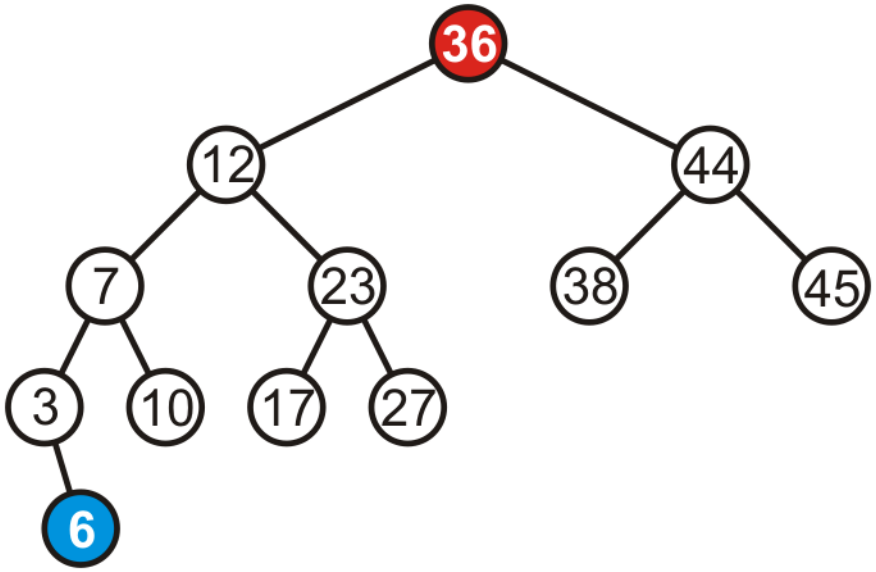
The height of each of the trees in the path back to the root are increased by one

- However, only the root node is now unbalanced



Maintaining Balance

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

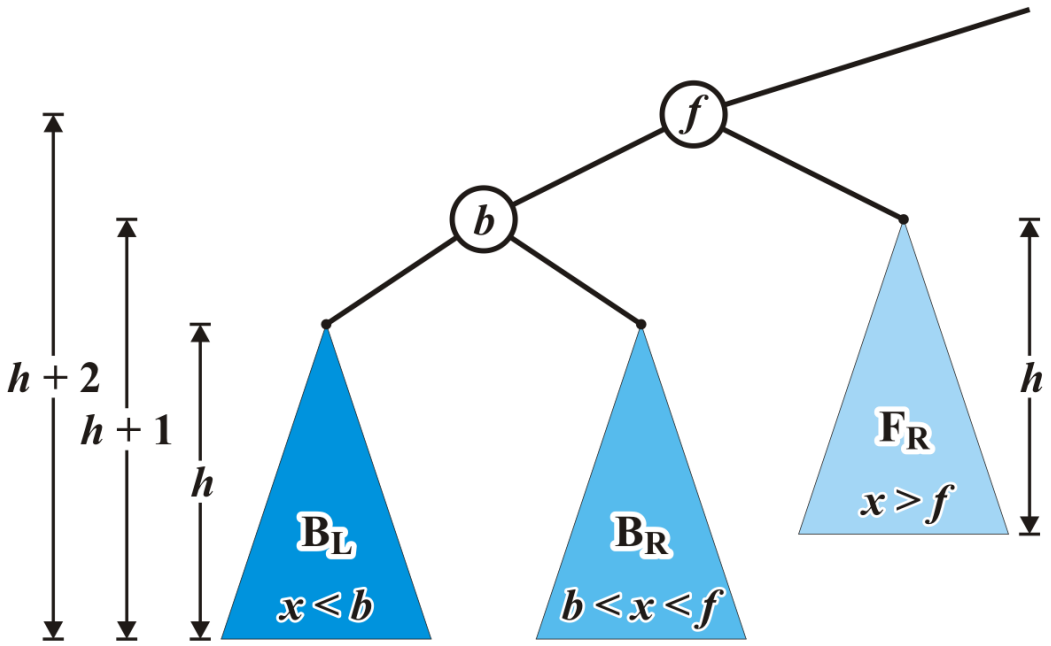


AVL TREES
CASE 1

Maintaining Balance: Case 1

Consider the following setup

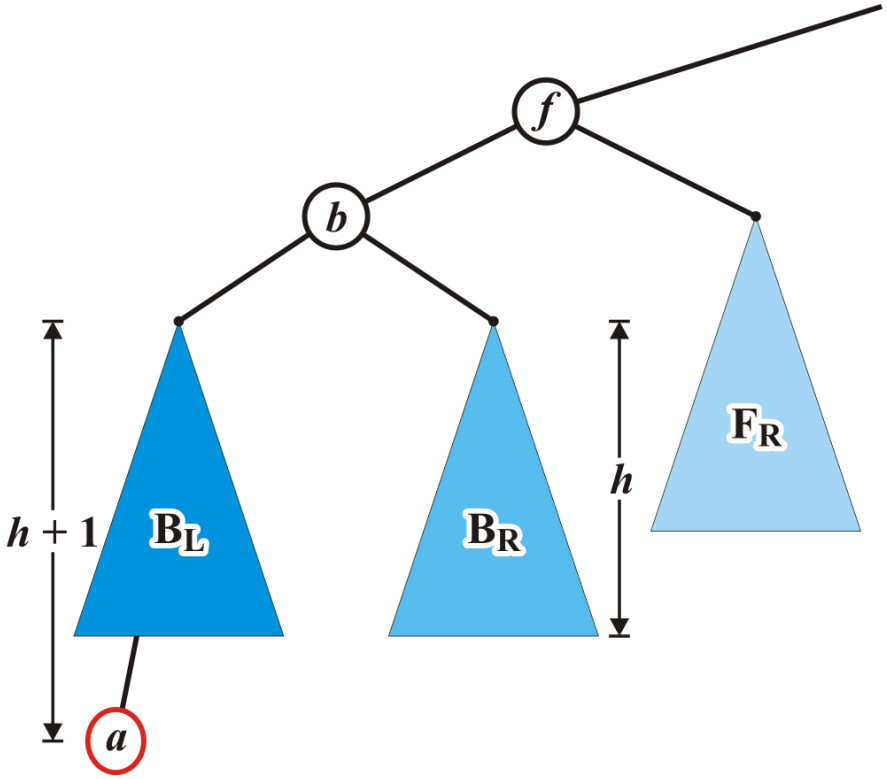
– Each blue triangle represents a tree of height h



Maintaining Balance: Case 1

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

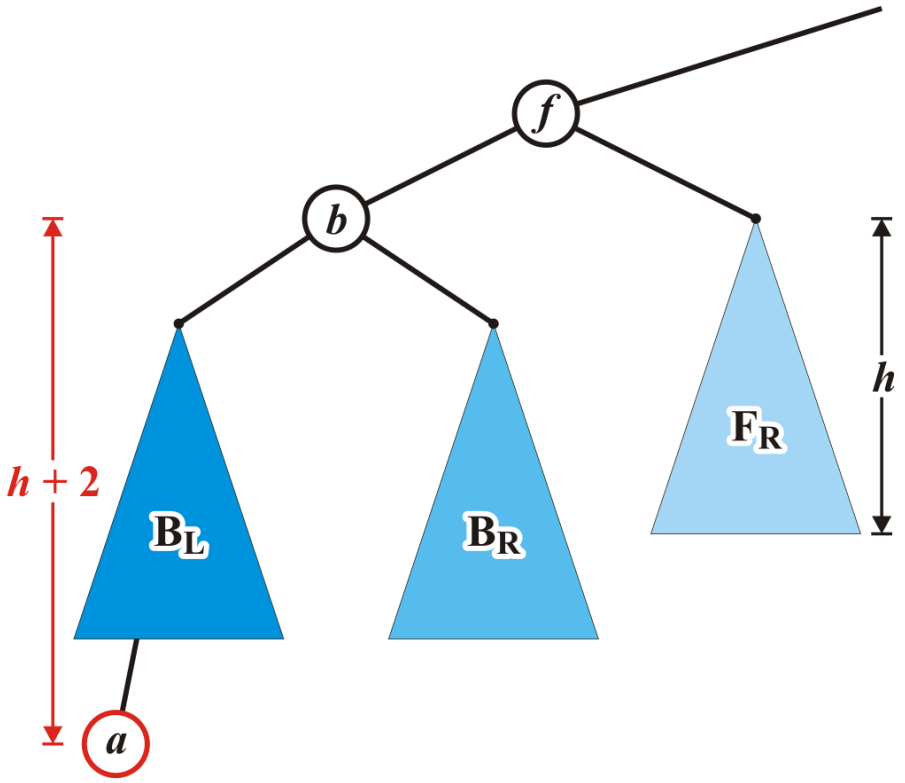
- Assume $\mathbf{B_L}$ remains balanced
- Thus, the tree rooted at ***b*** is also balanced



Maintaining Balance: Case 1

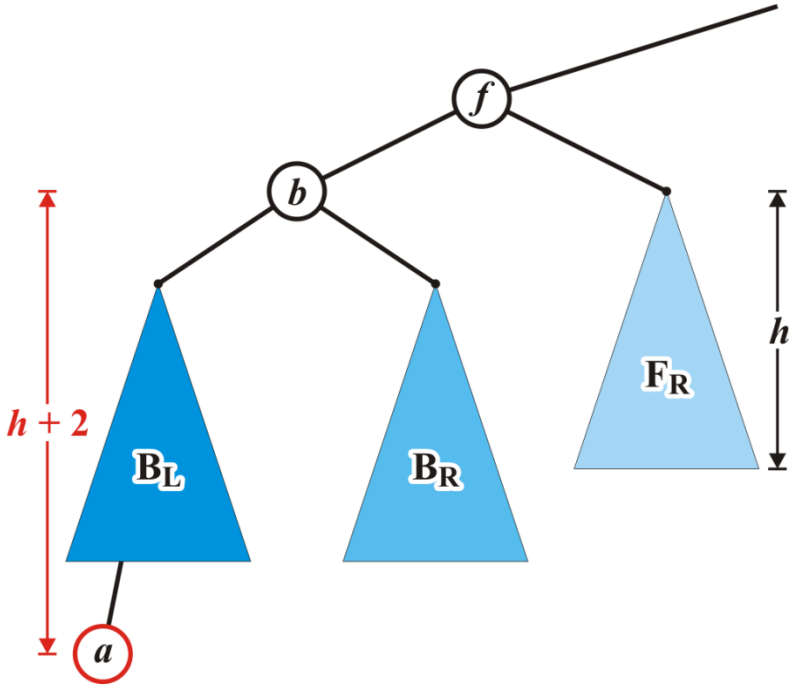
The tree rooted at node f is now unbalanced

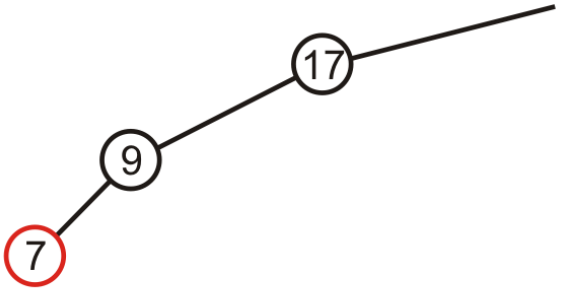
– We will correct the imbalance at this node

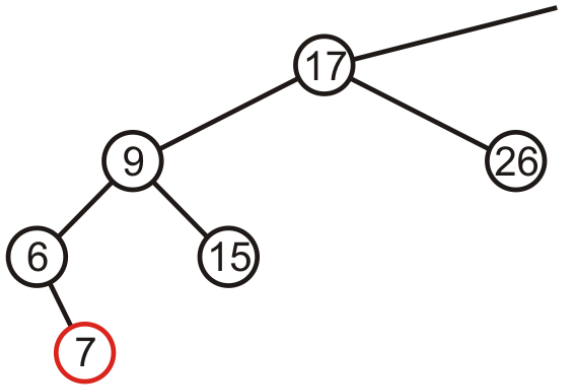


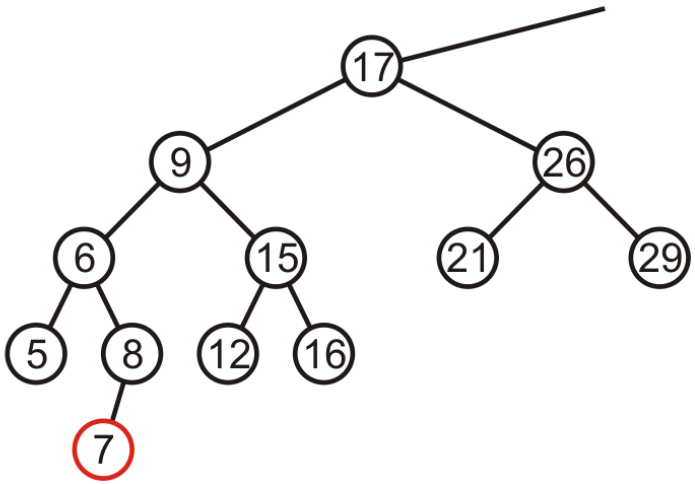
Maintaining Balance: Case 1

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





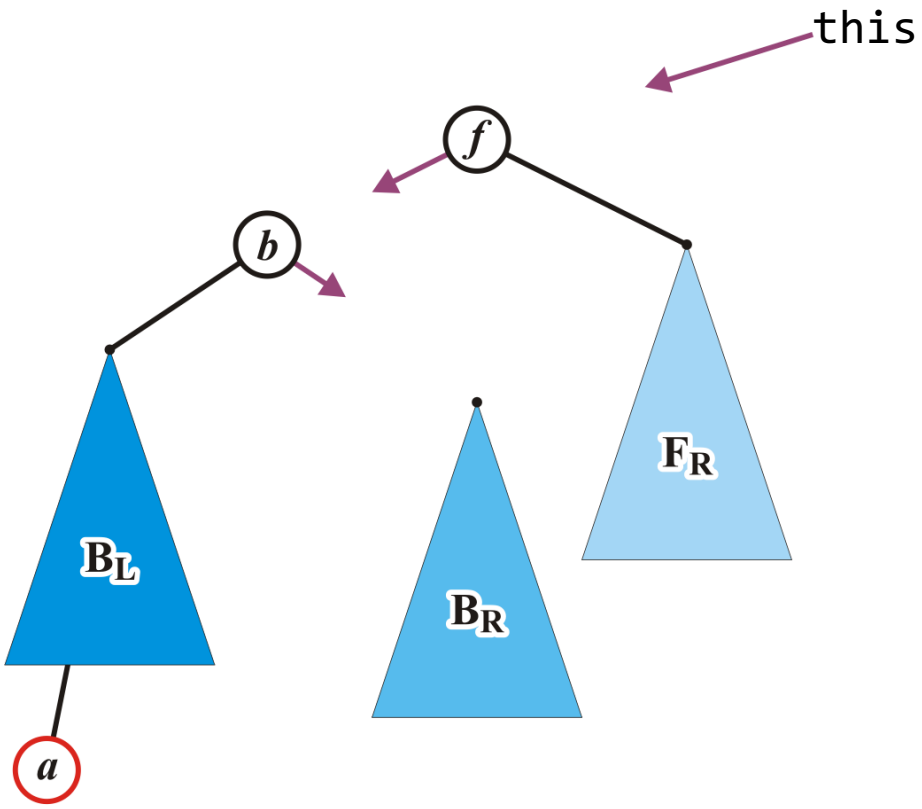




Maintaining Balance: Case 1

We will modify these three pointers

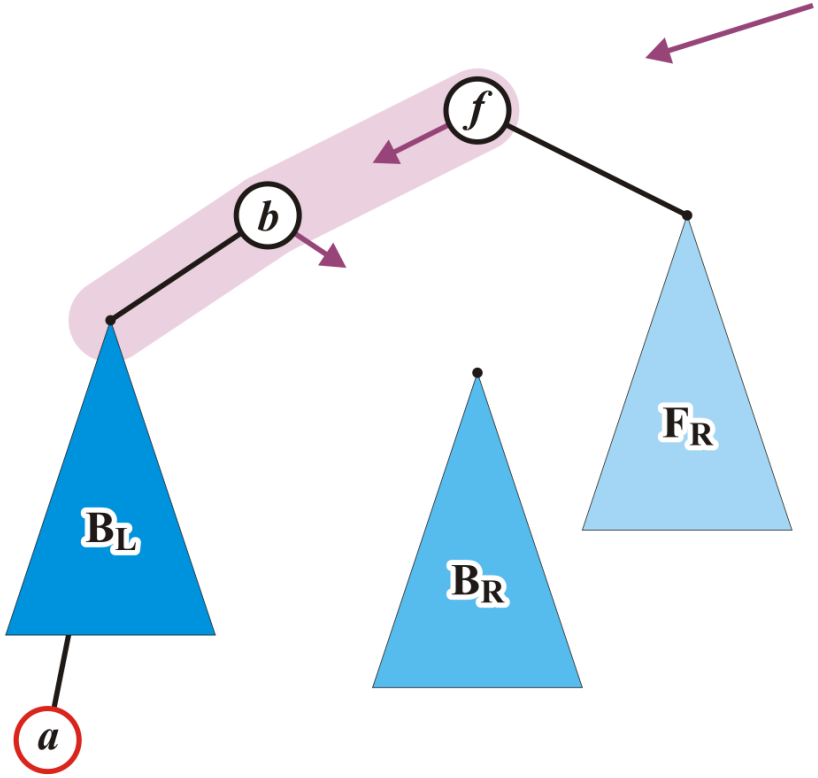
– At this point, `this` references the unbalanced root node *f*



Maintaining Balance: Case 1

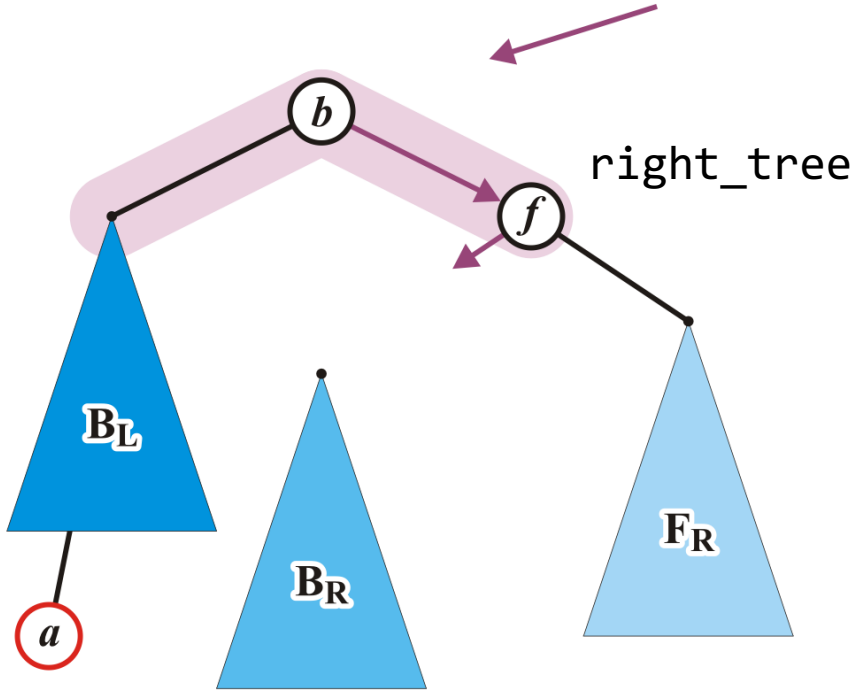
Specifically, we will rotate these two nodes around the root:

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



Maintaining Balance: Case 1

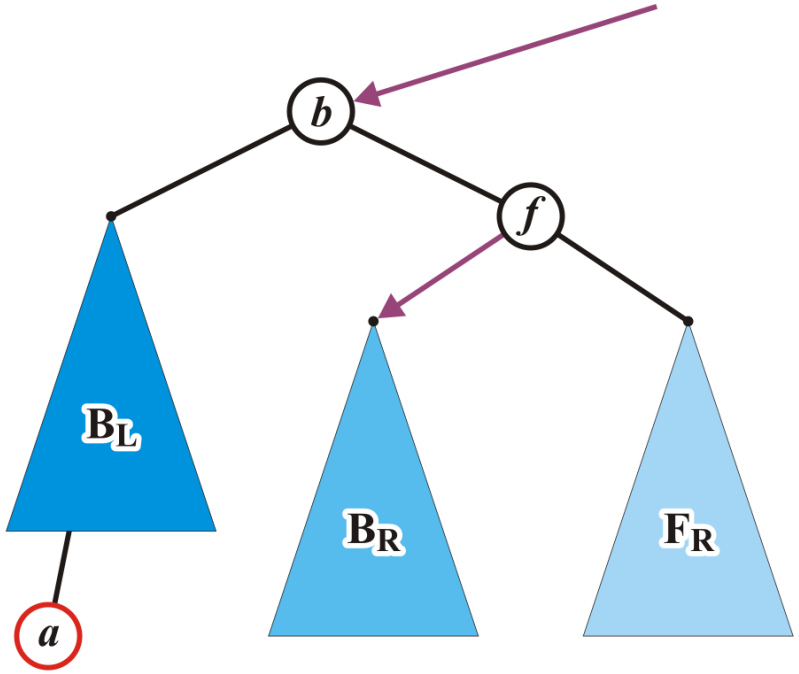
This requires the address of node *f* to be assigned to the `right_tree` member variable of node *b*



Maintaining Balance: Case 1

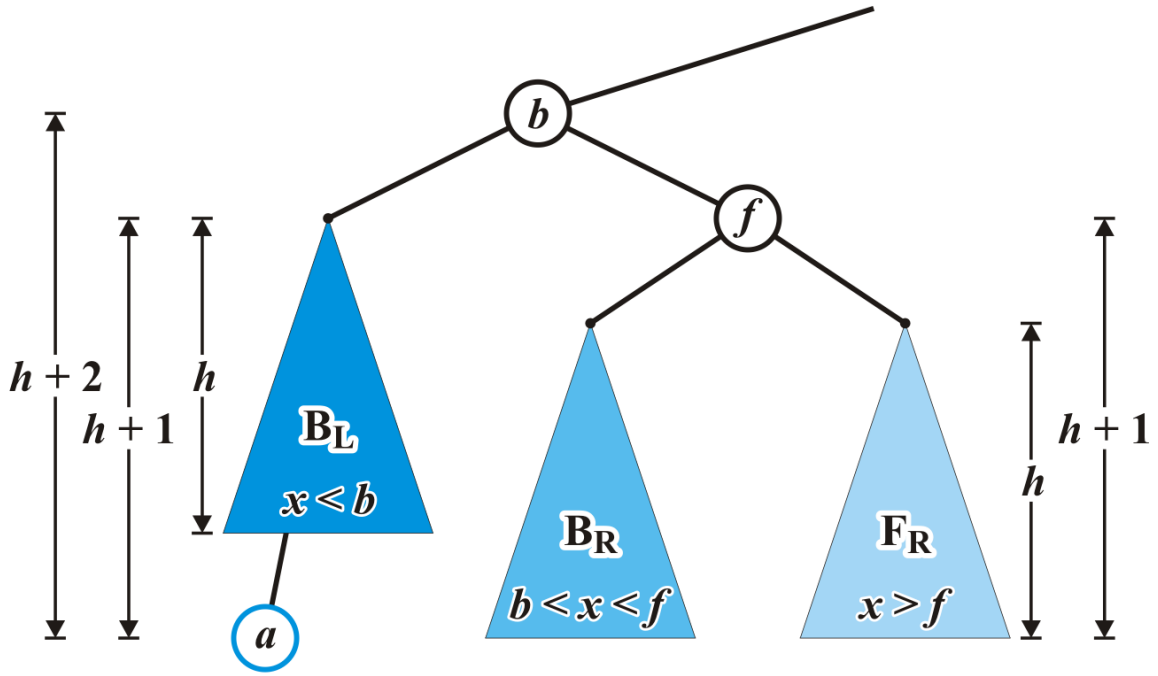
Assign any former parent of node f to the address of node b

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



Maintaining Balance: Case 1

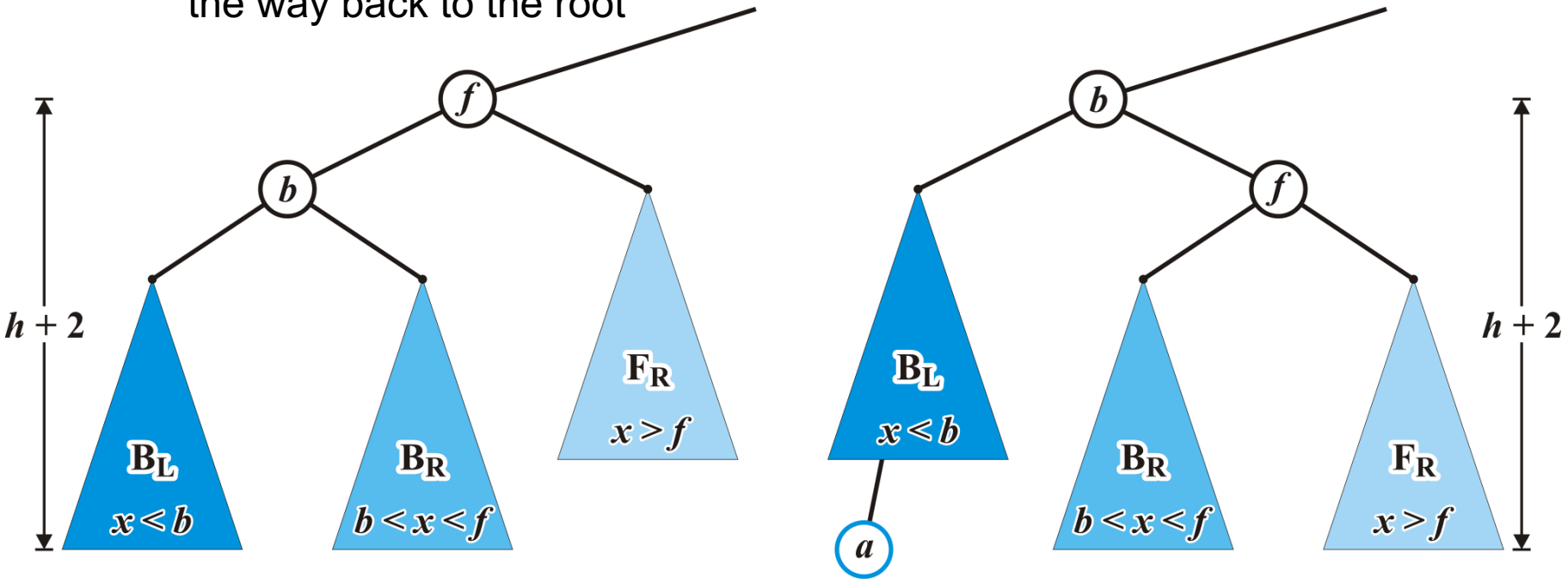
The nodes *b* and *f* are now balanced and all remaining nodes of the subtrees are in their correct positions



Maintaining Balance: Case 1

Additionally, height of the tree rooted at b equals the original height of the tree rooted at f

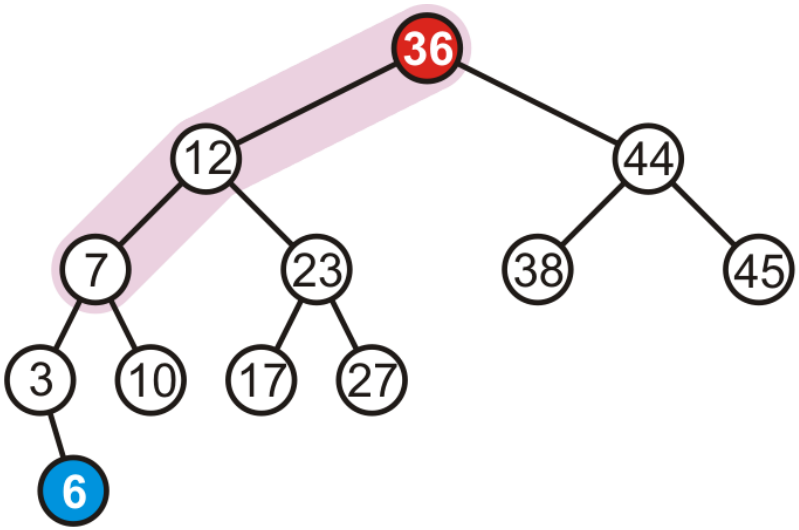
– Thus, this insertion will no longer affect the balance of any ancestors all the way back to the root

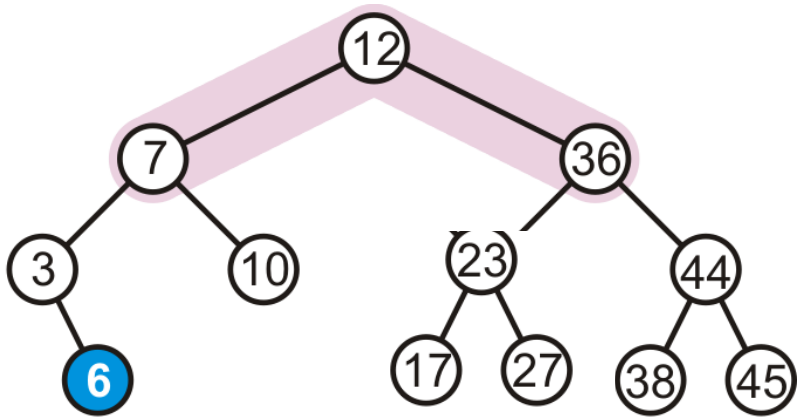


Maintaining Balance: Case 1

In our example case, the correction

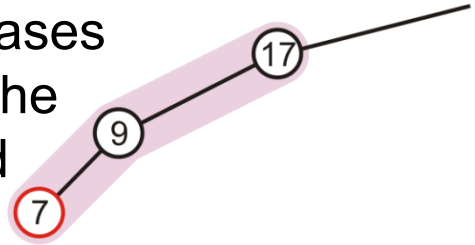
*Compute out the **balance factor for 36**.*

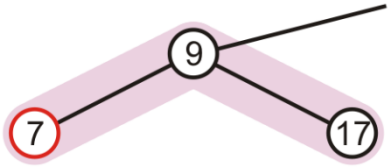


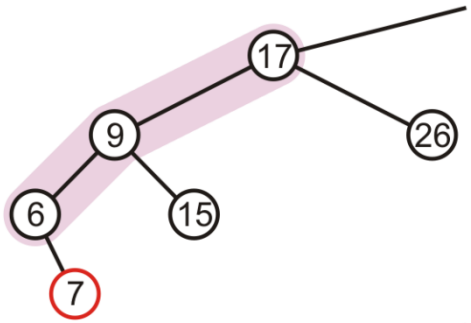


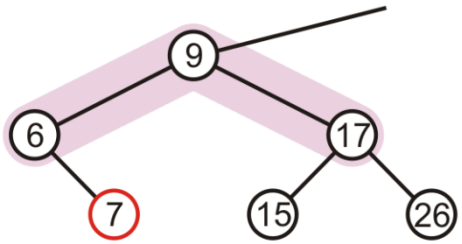
Maintaining Balance: Case 1

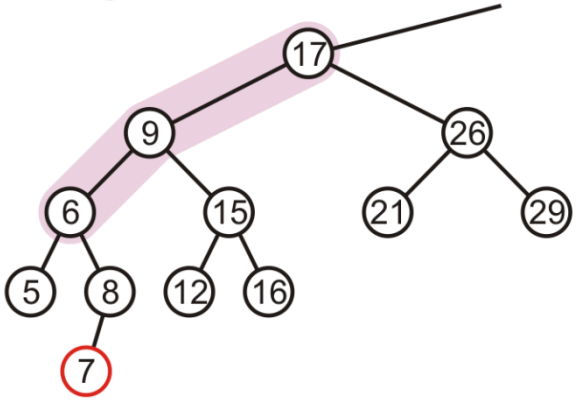
In our three sample cases with $h = -1$, 0, and 1, the node is now balanced and the same height as the tree before the insertion

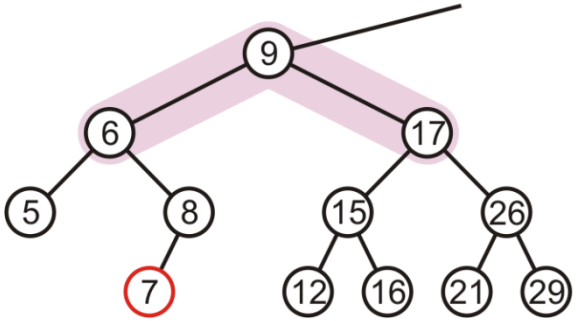








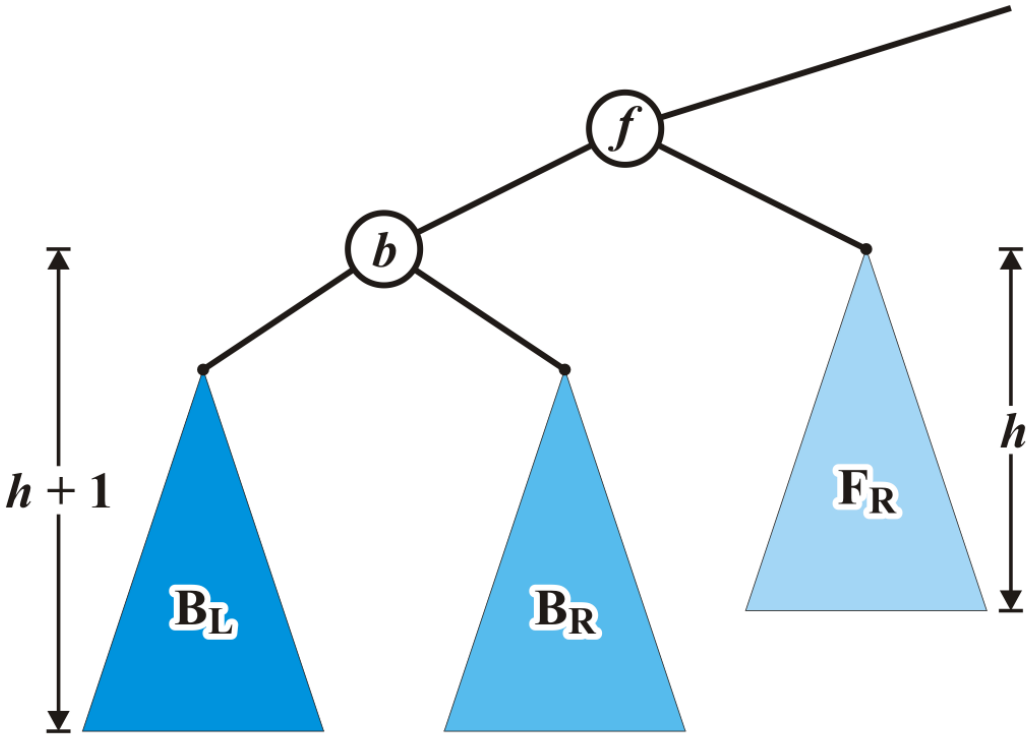




AVL TREES
CASE 2

Maintaining Balance: Case 2

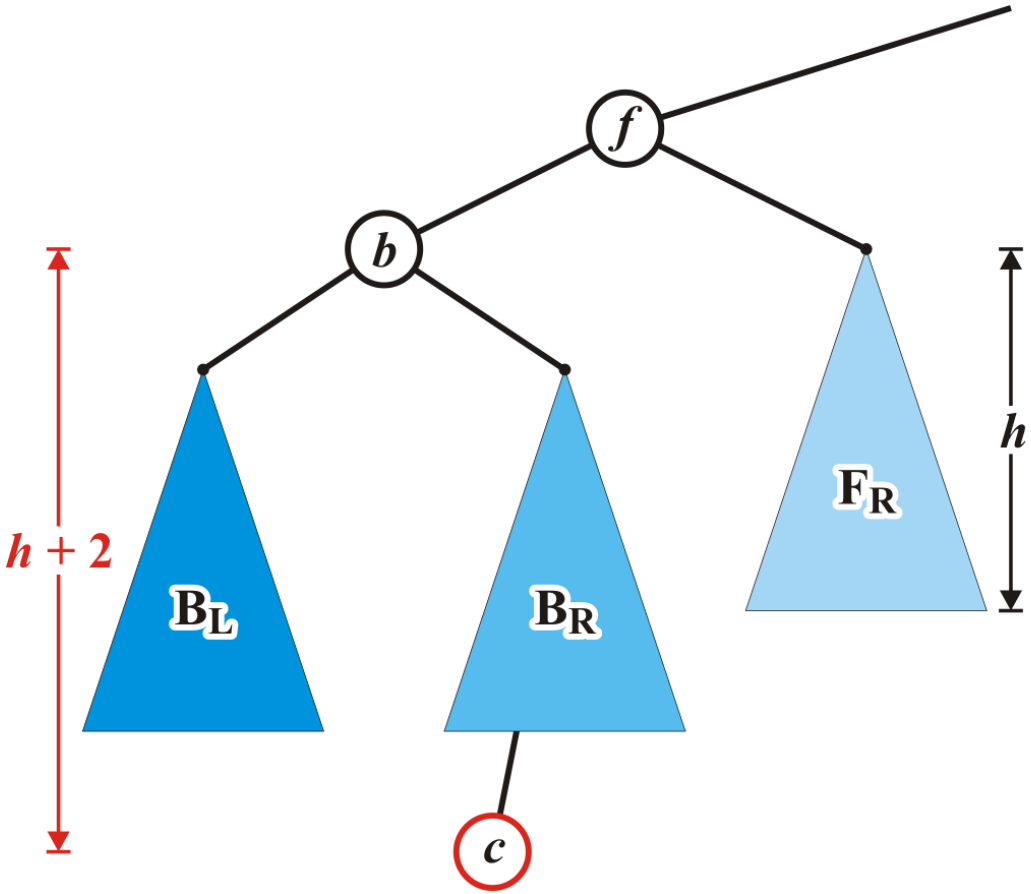
Alternatively, consider the insertion of c where $b < c < f$ into our original tree



Maintaining Balance: Case 2

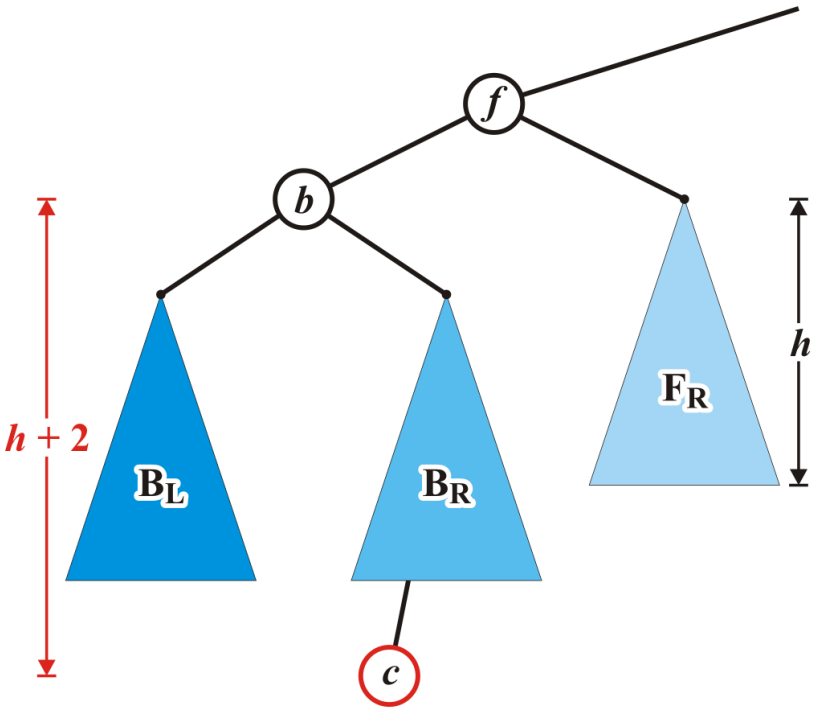
Assume that the insertion of c increases the height of $\mathbf{B_R}$

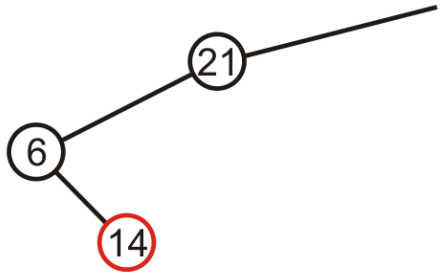
– Once again, f becomes unbalanced

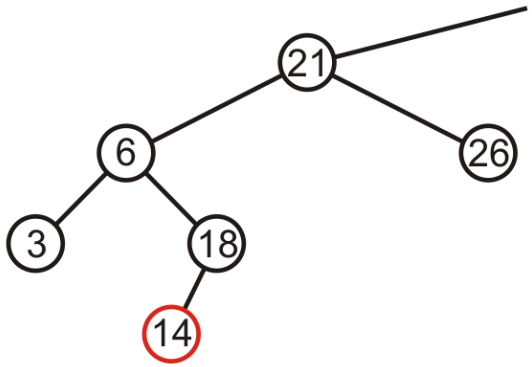


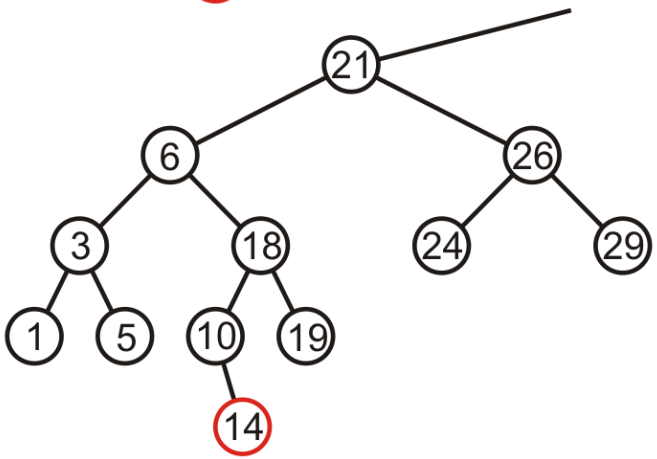
Maintaining Balance: Case 2

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



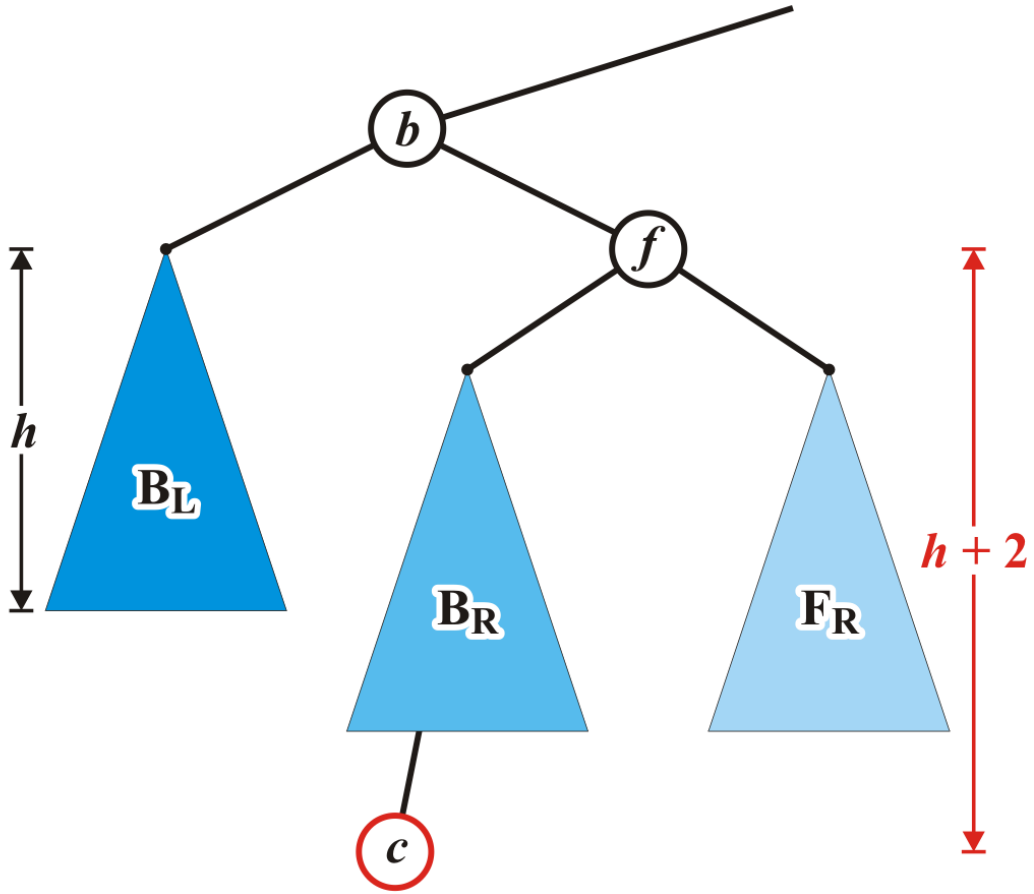






Maintaining Balance: Case 2

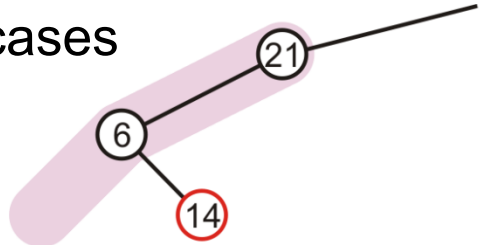
Unfortunately, the previous correction does not fix the imbalance at the root of this sub-tree: the new root, ***b***, remains unbalanced

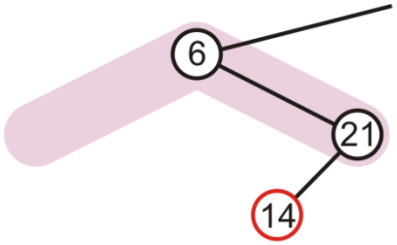


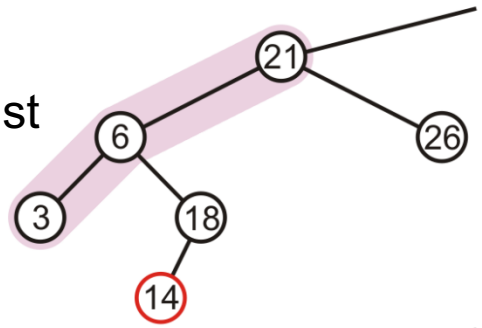
Maintaining Balance: Case 2

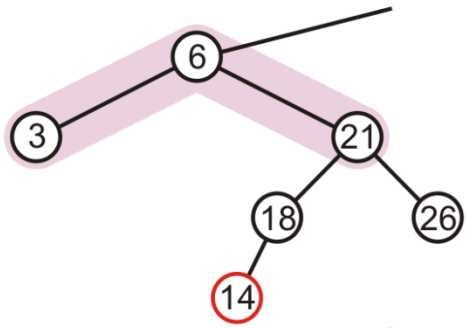
In our three sample cases with $h = -1$, 0, and 1, doing the same thing as before results in a tree that is still unbalanced…

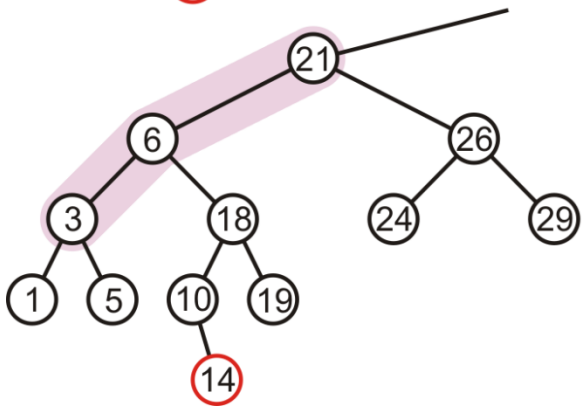
- The imbalance is just shifted to the other side

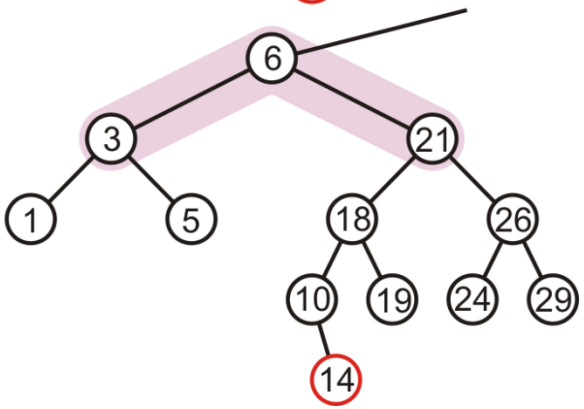






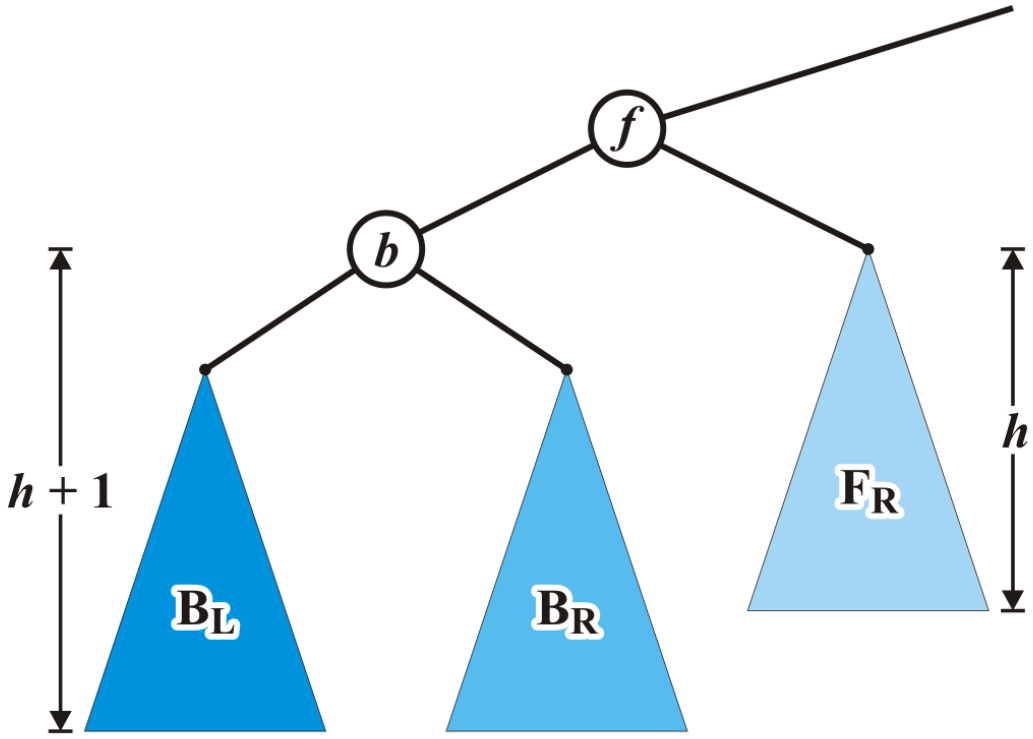






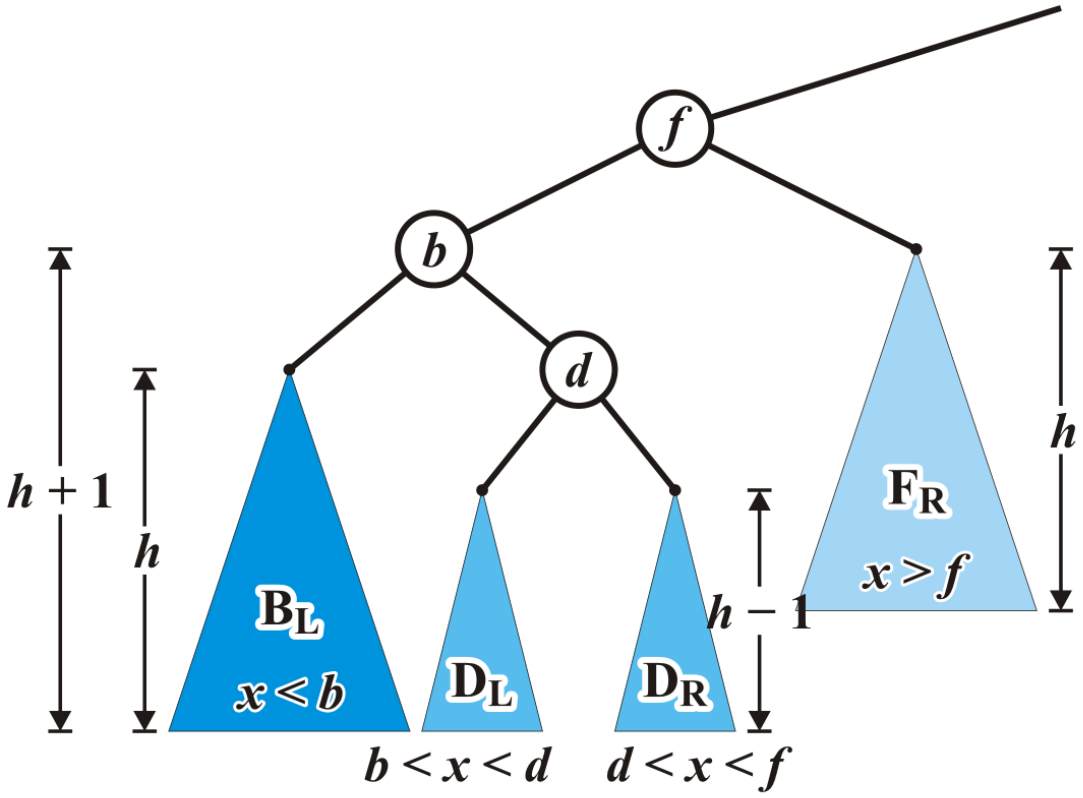
Maintaining Balance: Case 2

Unfortunately, this does not fix the imbalance at the root of this subtree



Maintaining Balance: Case 2

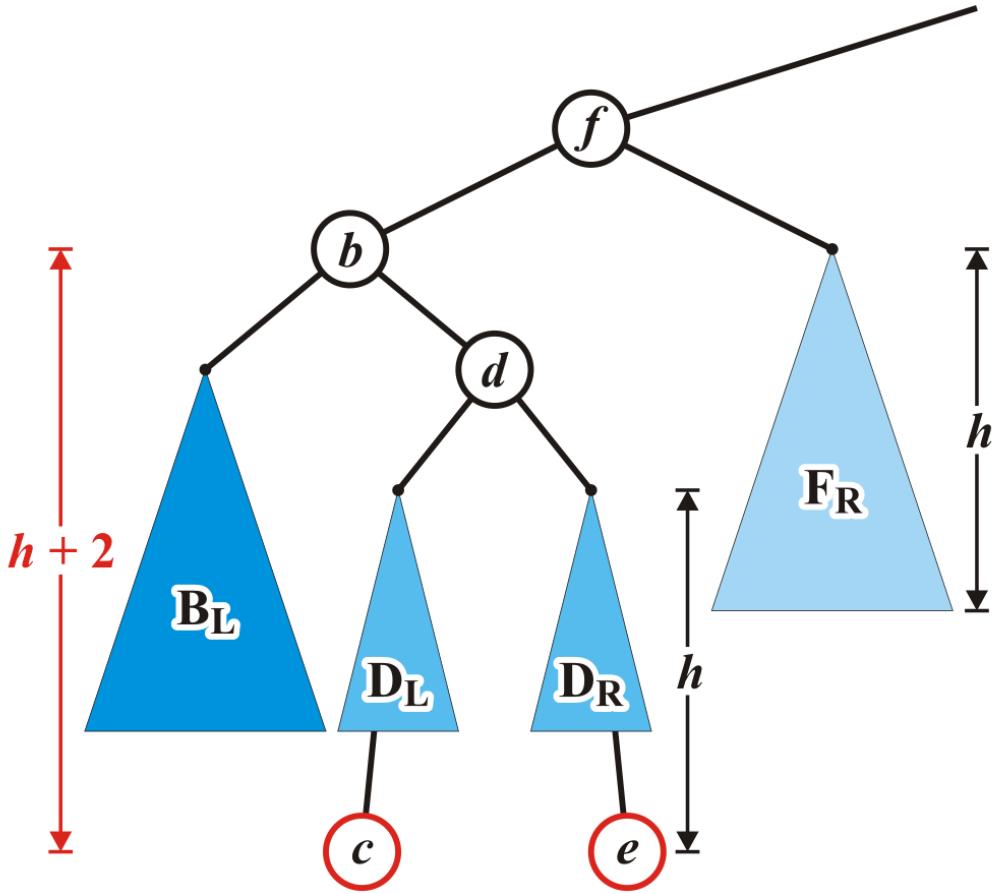
Re-label the tree by dividing the left subtree of f into a tree rooted at d with two subtrees of height $h - 1$



Maintaining Balance: Case 2

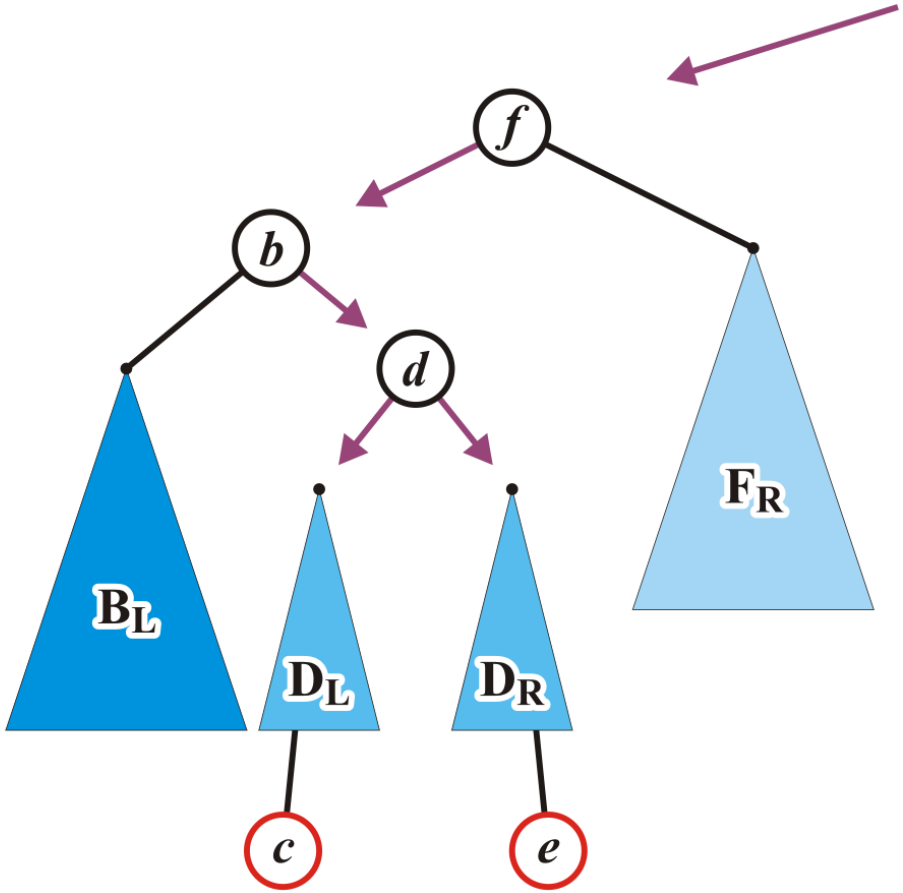
Now an insertion causes an imbalance at f

– The addition of either c or e will cause this



Maintaining Balance: Case 2

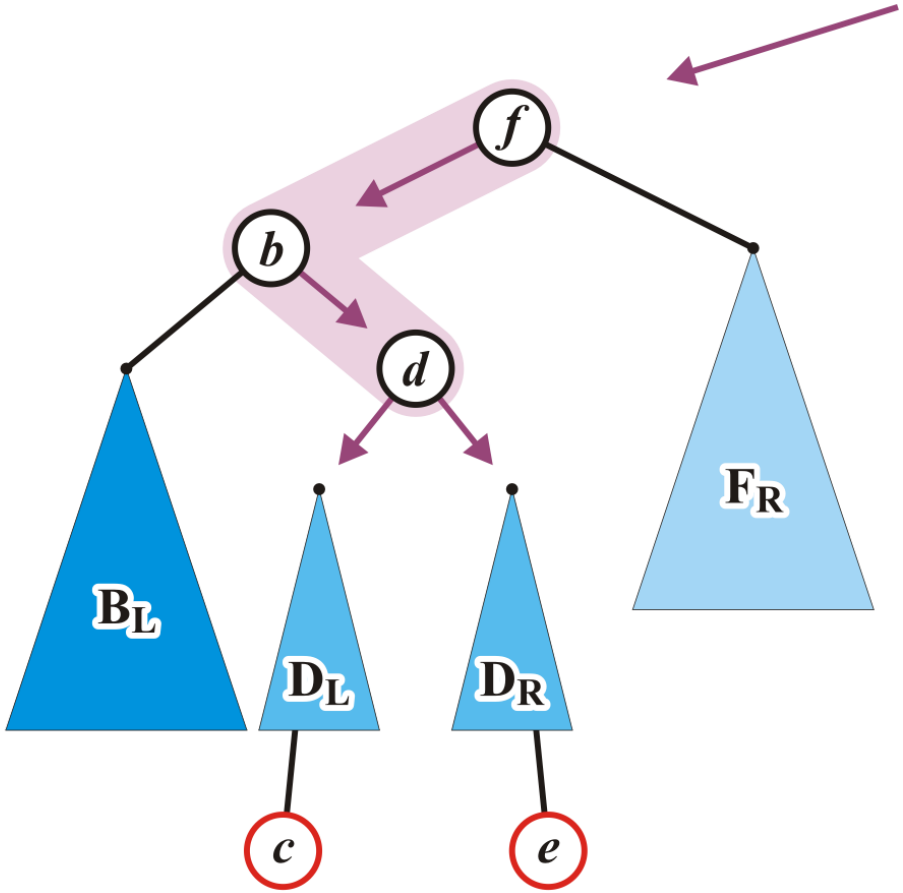
We will reassign the following pointers



Maintaining Balance: Case 2

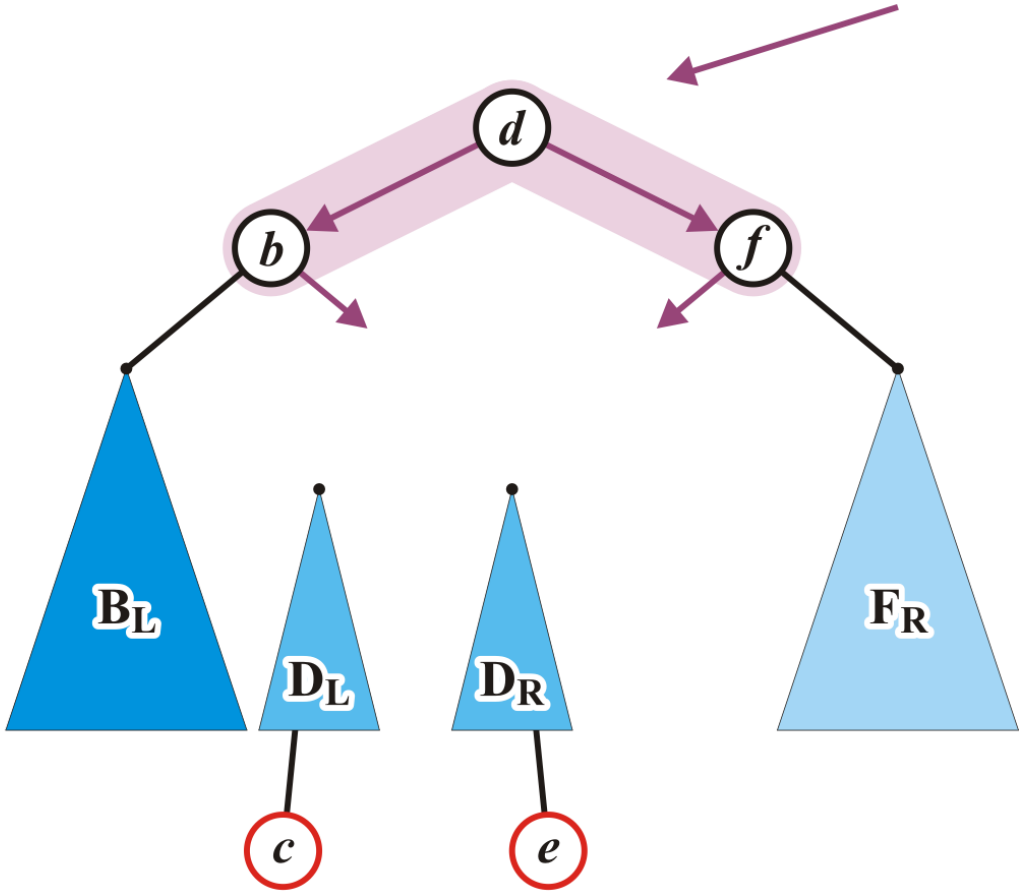
Specifically, we will order these three nodes as a perfect tree

– Recall the second prototypical example



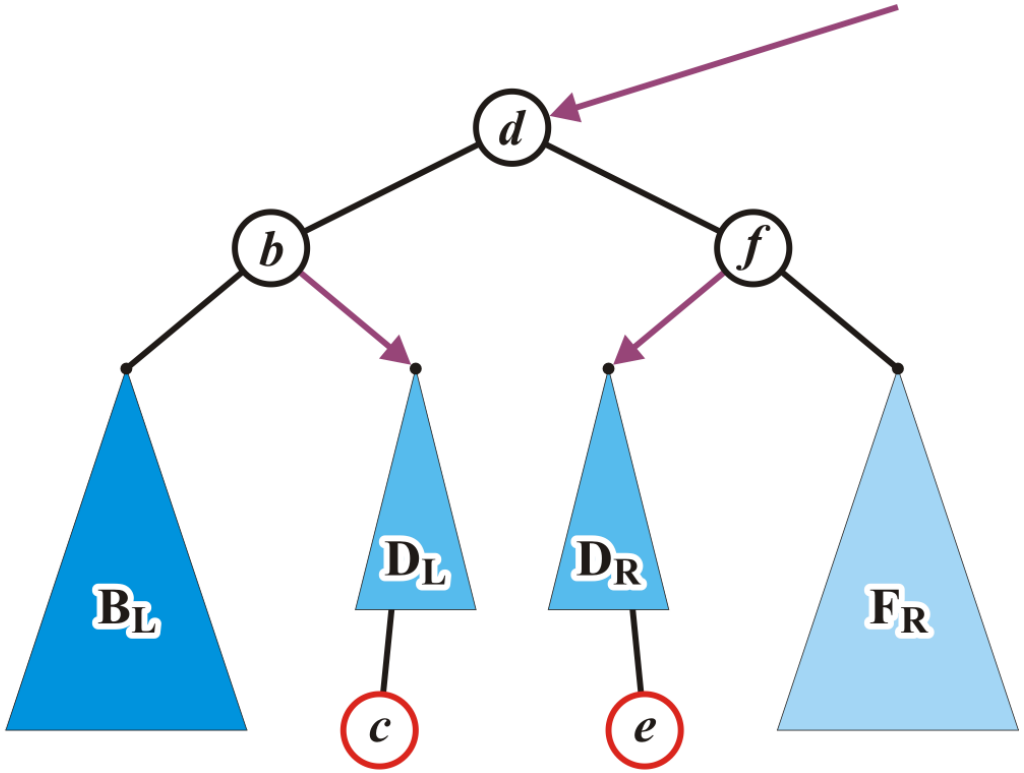
Maintaining Balance: Case 2

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



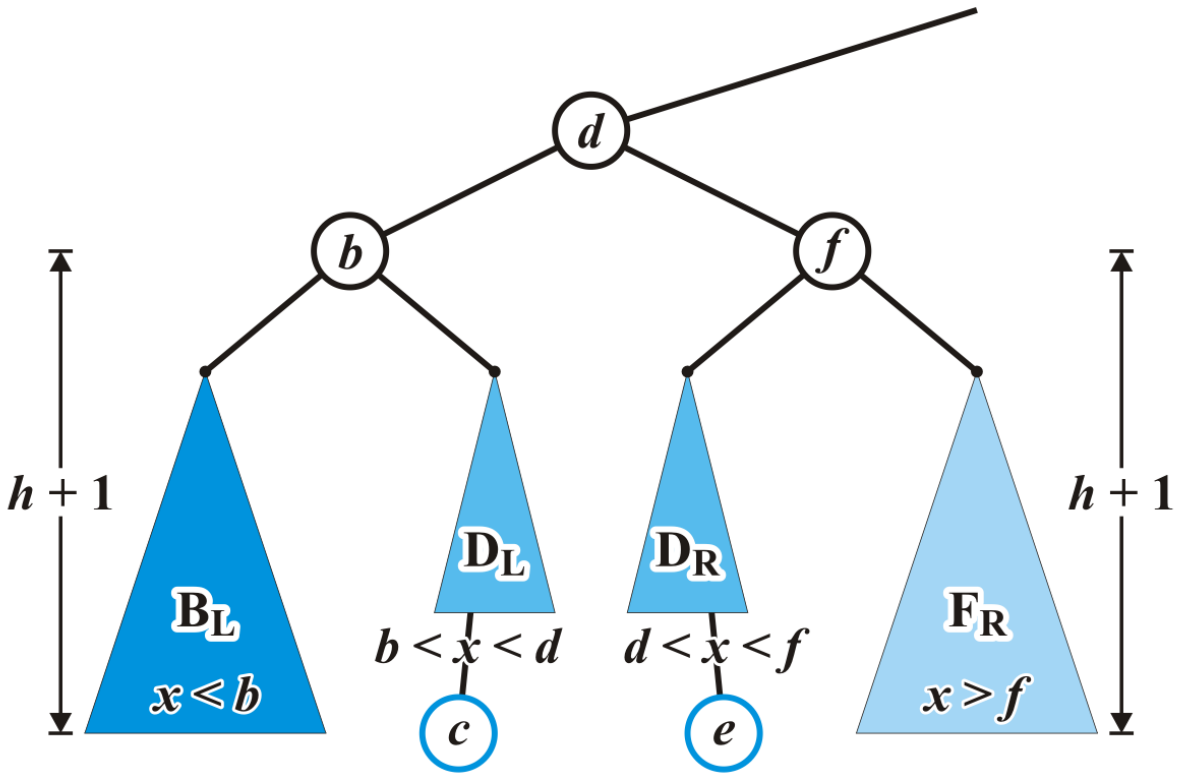
Maintaining Balance: Case 2

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



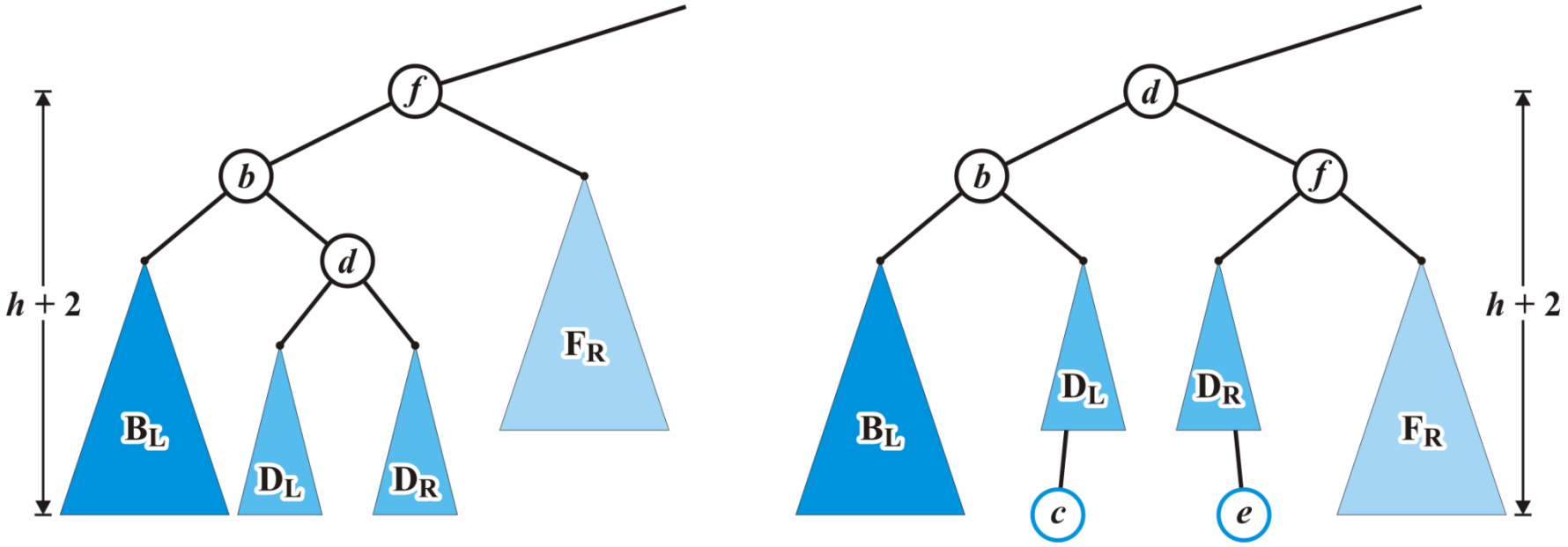
Maintaining Balance: Case 2

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



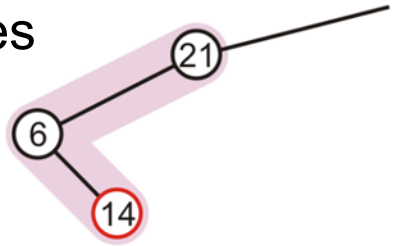
Maintaining Balance: Case 2

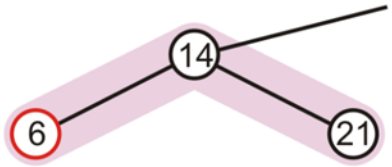
Again, the height of the root did not change

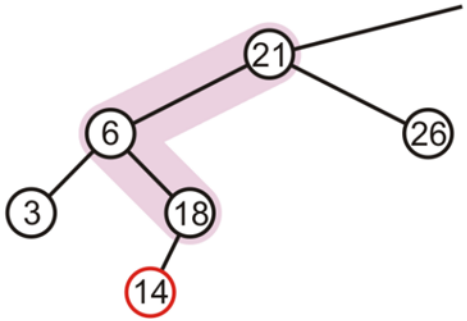


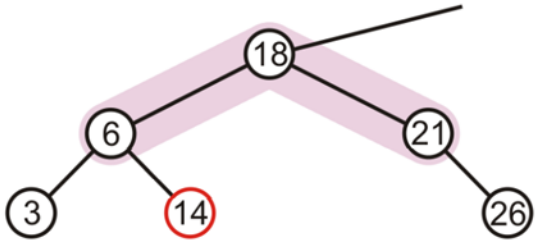
Maintaining Balance: Case 2

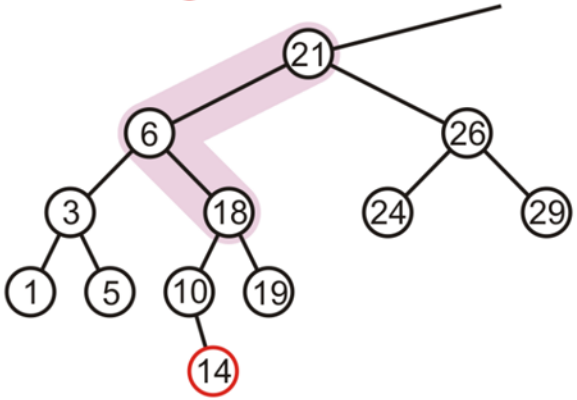
In our three sample cases with $h = -1$, 0, and 1, the node is now balanced and the same height as the tree before the insertion

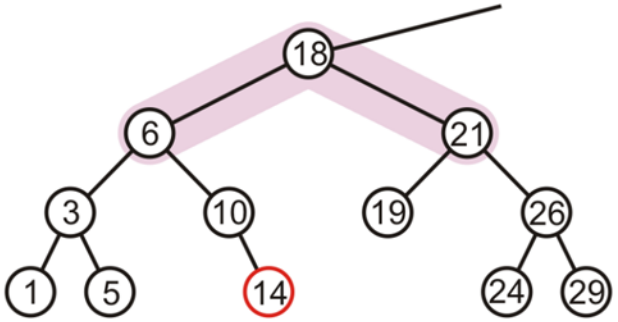








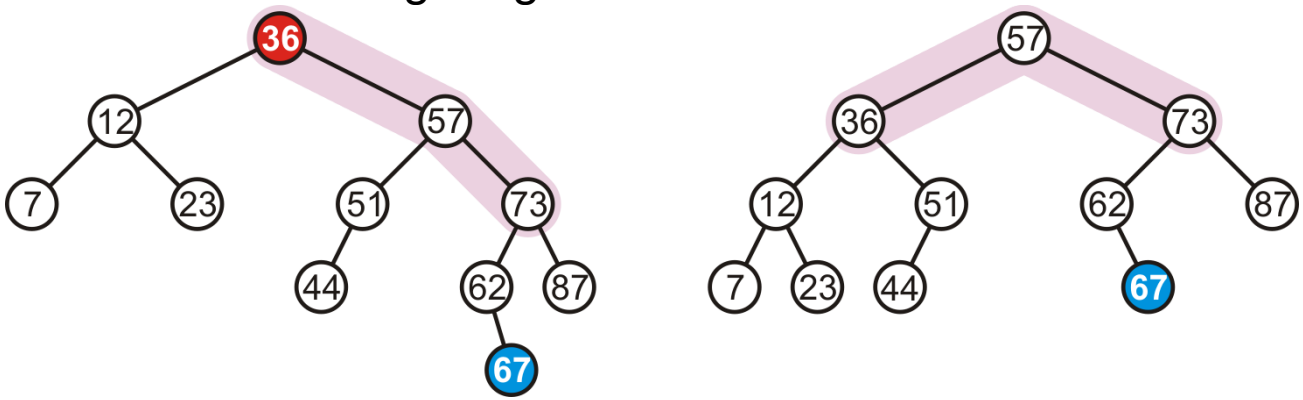




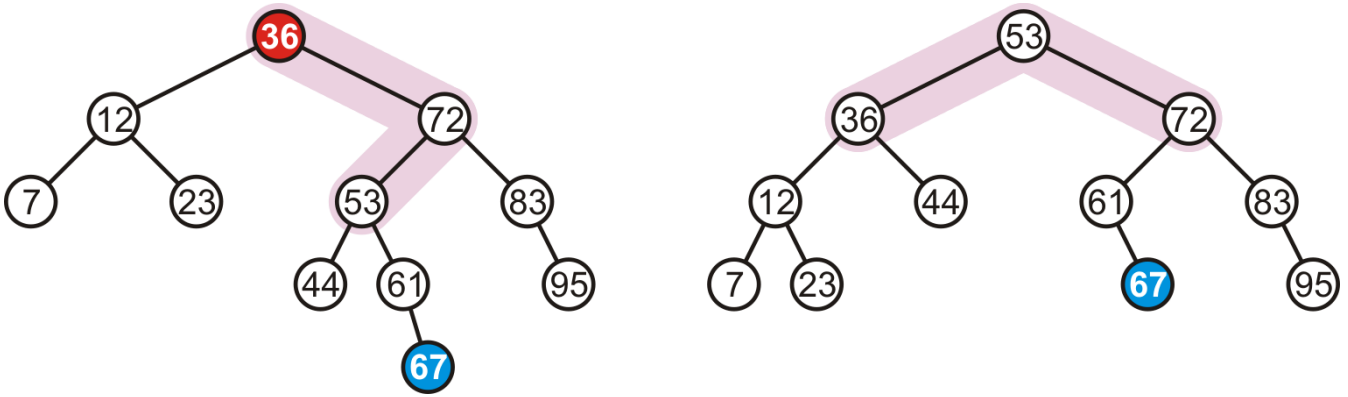
Maintaining balance: Summary

There are two symmetric cases to those we have examined:

- Insertions into the right-right sub-tree

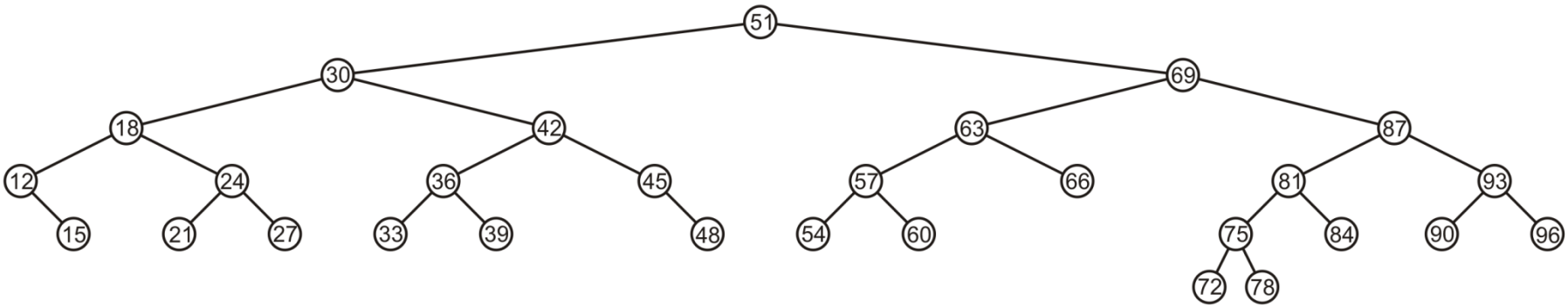


- Insertions into either the right-left sub-tree



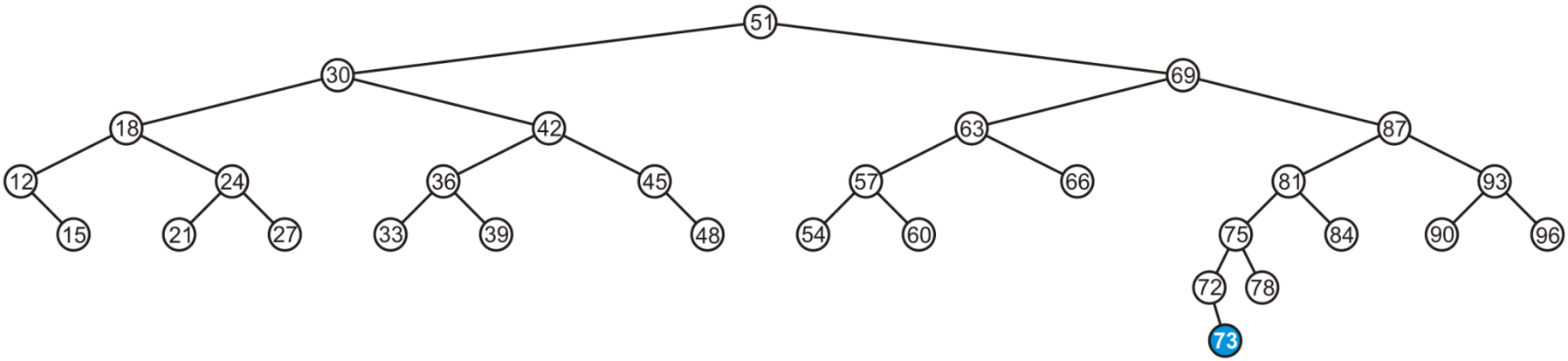
Insertion

Consider this AVL tree



Insertion

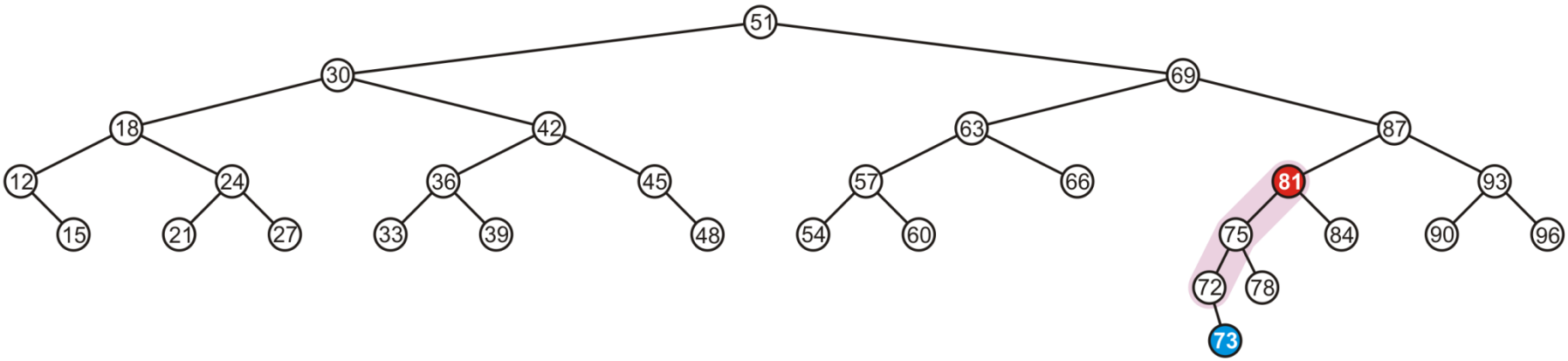
Insert 73



Insertion

The node 81 is unbalanced

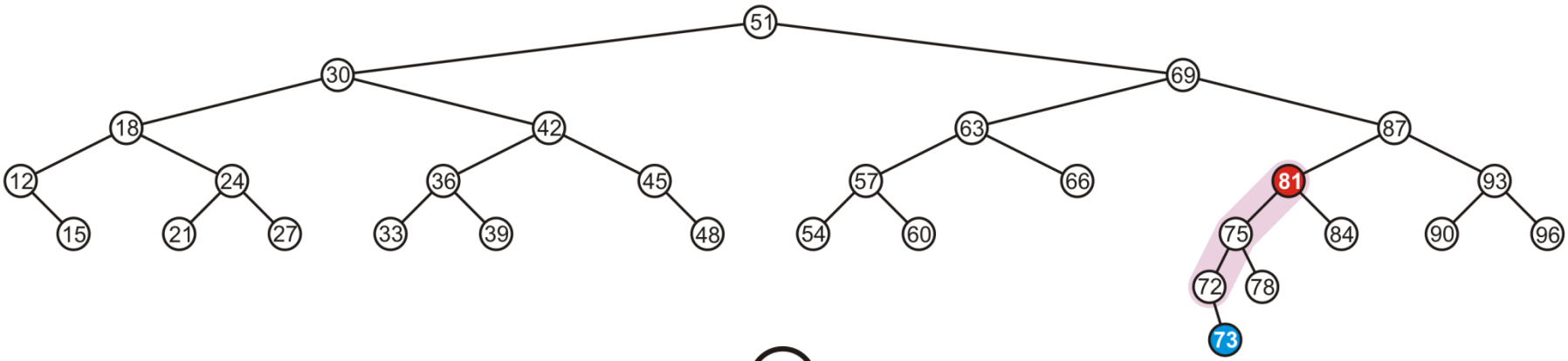
- A left-left imbalance

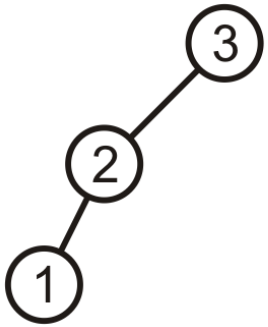


Insertion

The node 81 is unbalanced

- A left-left imbalance

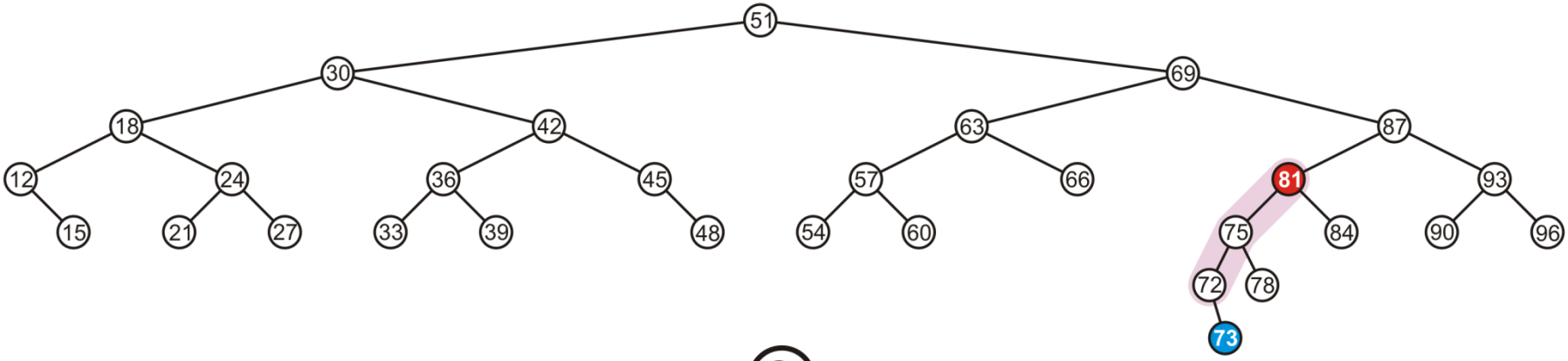


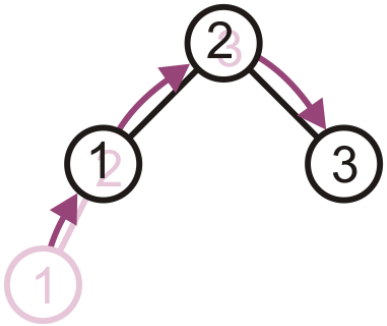


Insertion

The node 81 is unbalanced

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

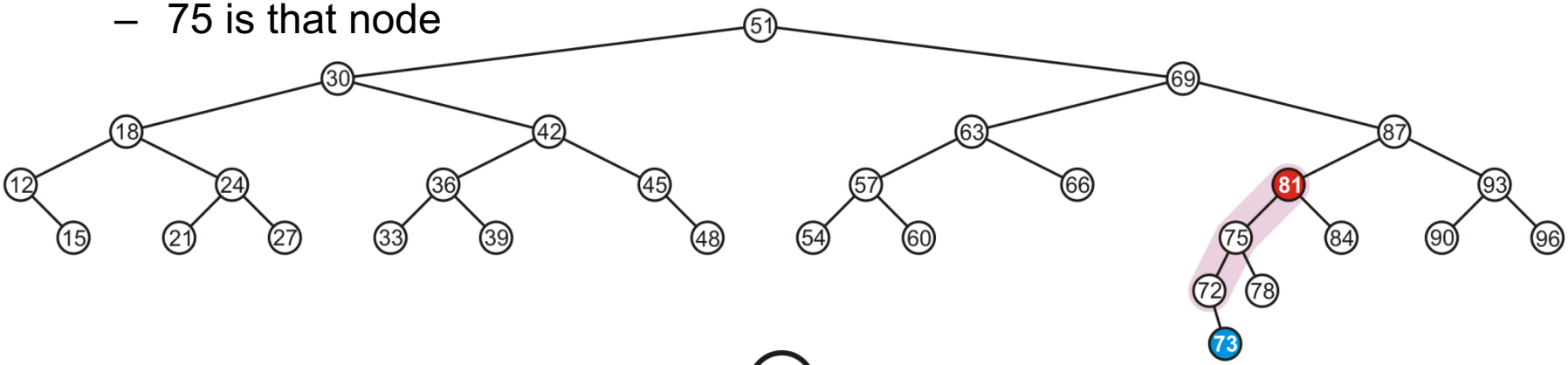


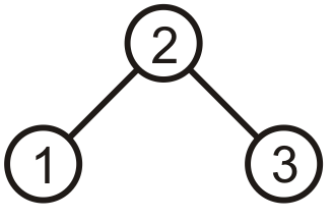


Insertion

The node 81 is unbalanced

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

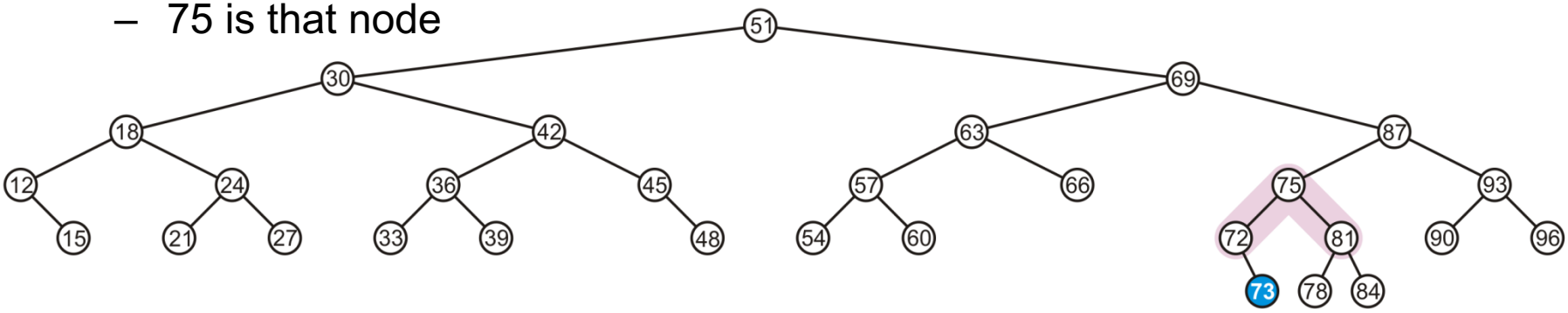




Insertion

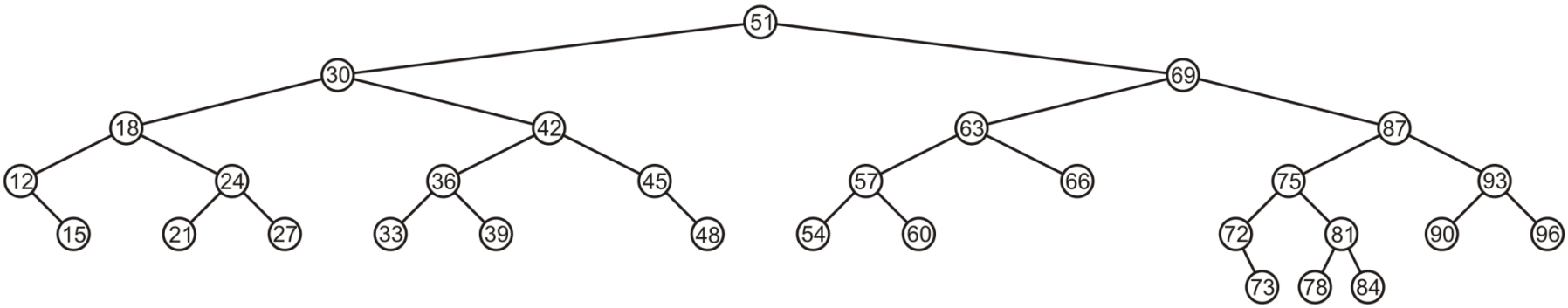
The node 81 is unbalanced

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



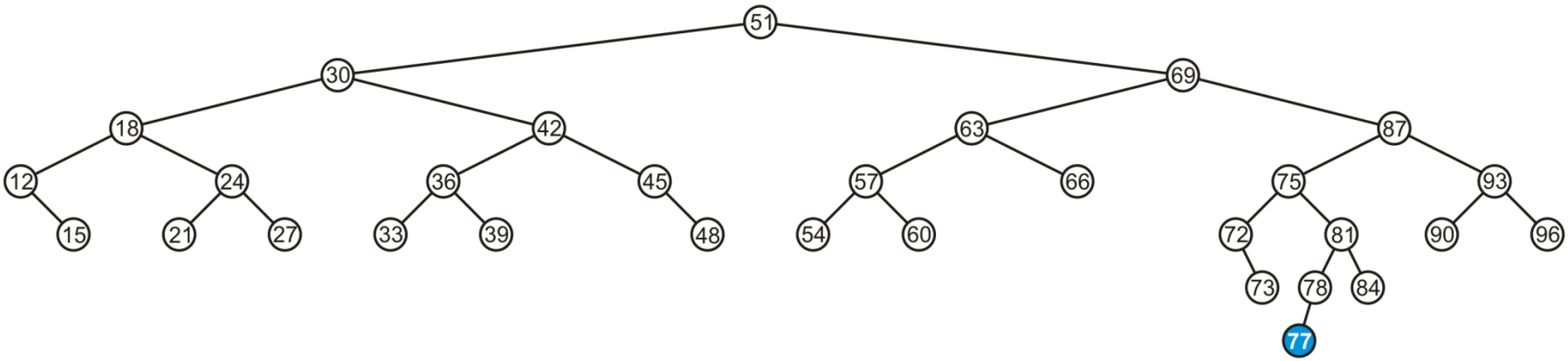
Insertion

The tree is AVL balanced



Insertion

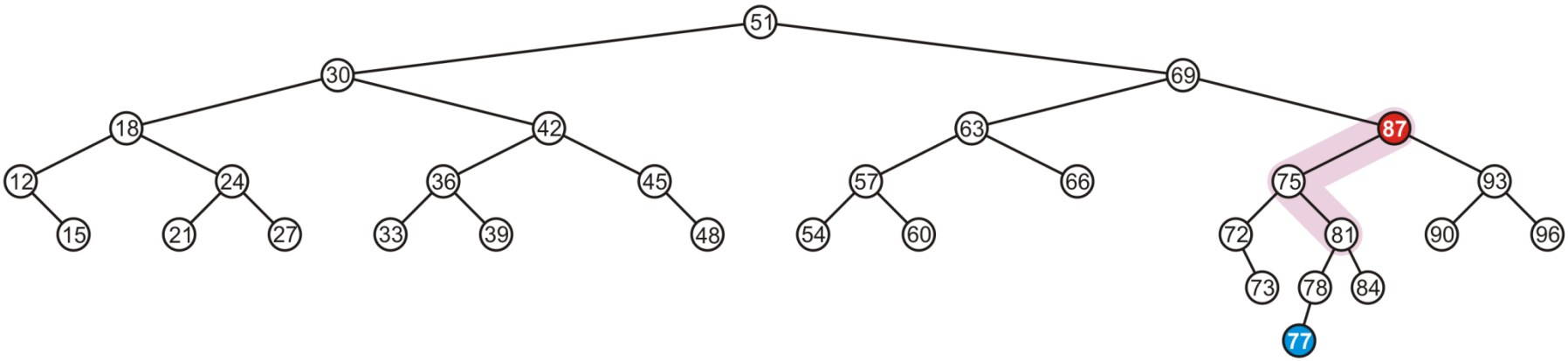
Insert 77



Insertion

The node 87 is unbalanced

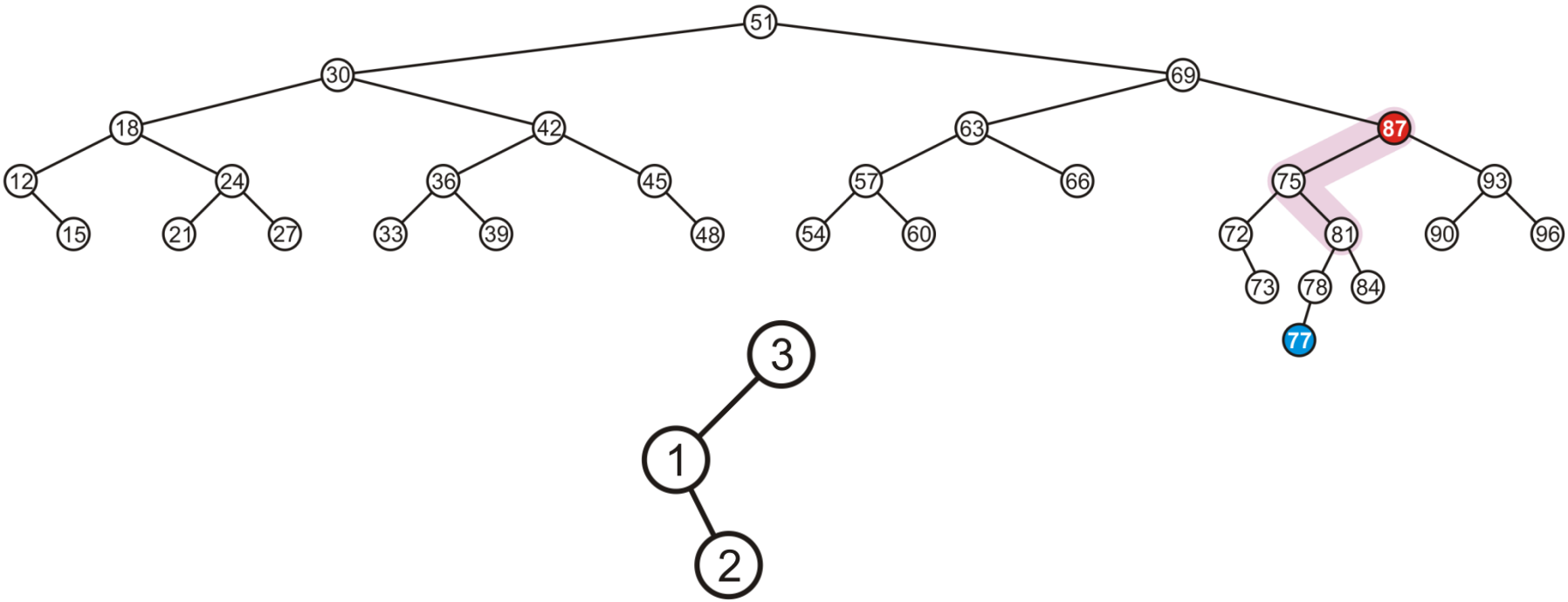
– A left-right imbalance



Insertion

The node 87 is unbalanced

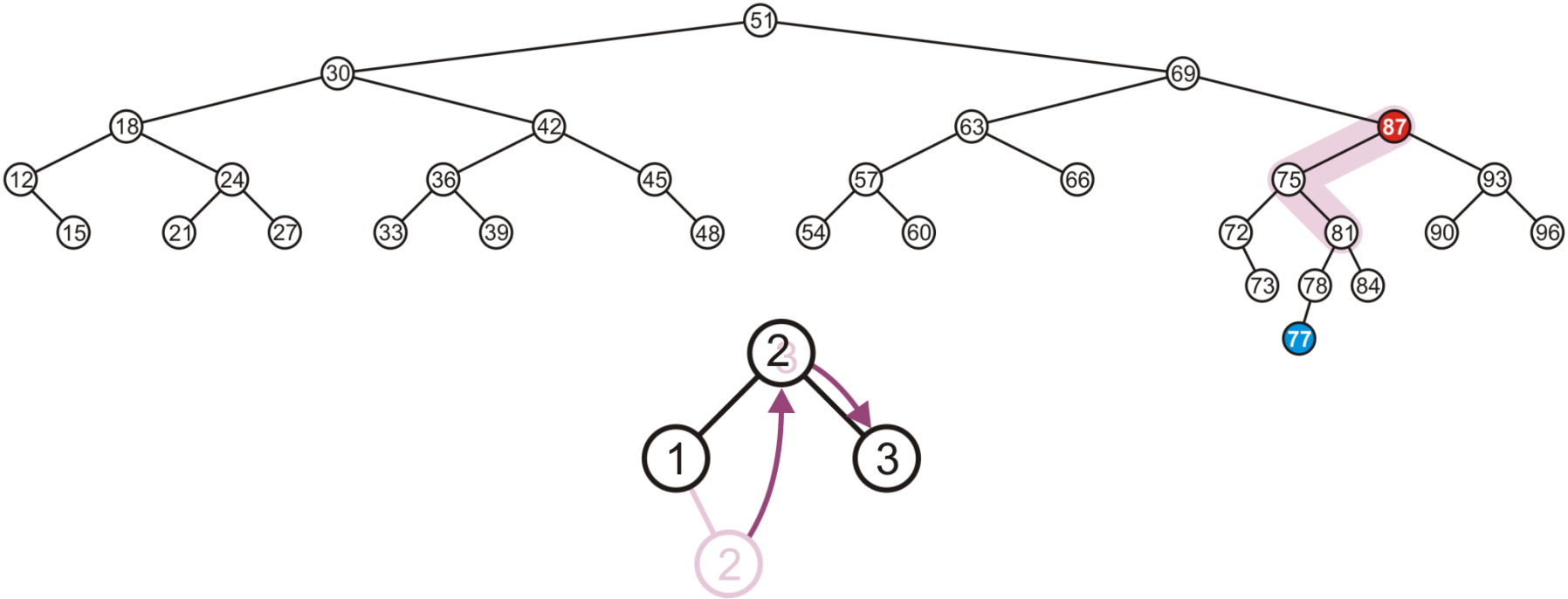
– A left-right imbalance



Insertion

The node 87 is unbalanced

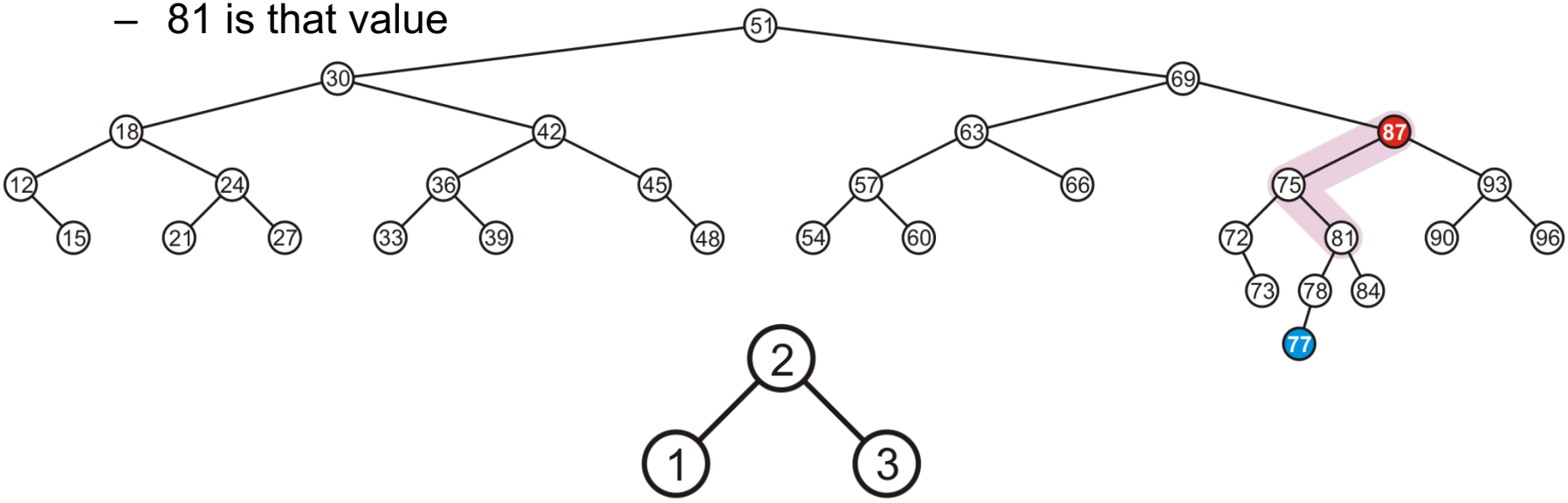
- A left-right imbalance
- Promote the intermediate node to the imbalanced node



Insertion

The node 87 is unbalanced

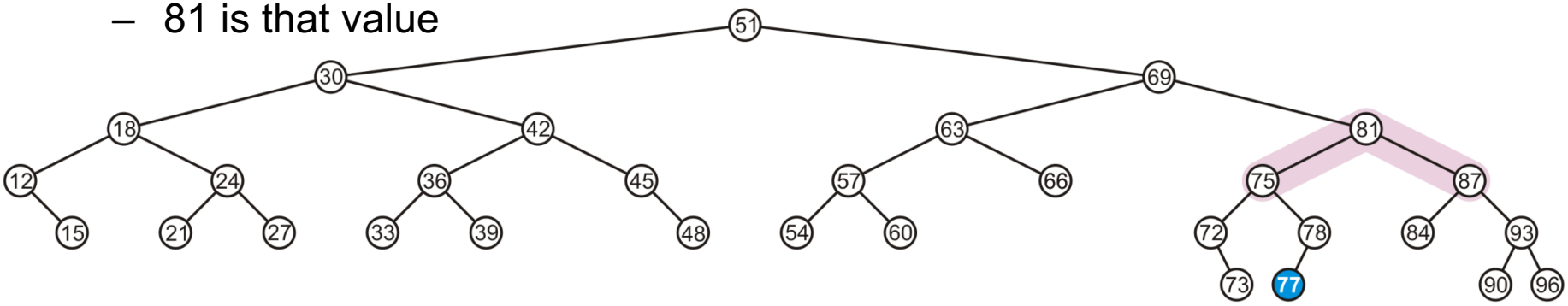
- A left-right imbalance
- Promote the intermediate node to the imbalanced node
- 81 is that value



Insertion

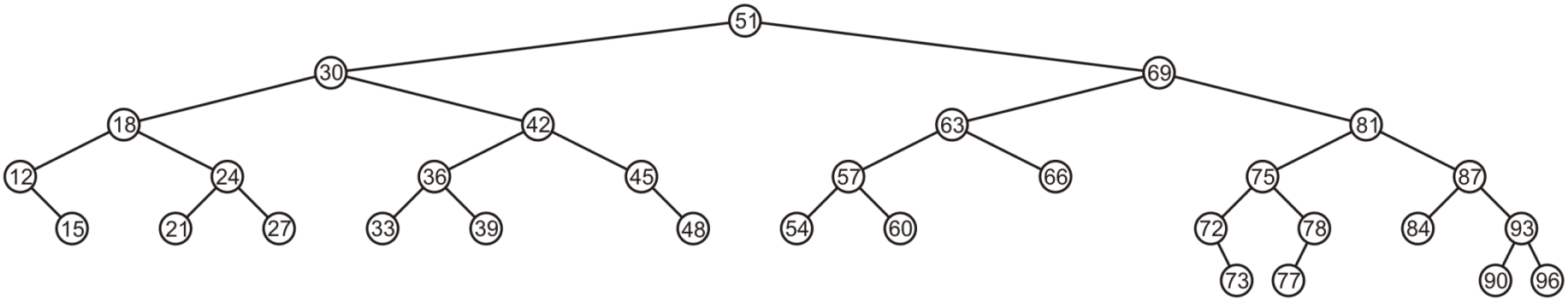
The node 87 is unbalanced

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



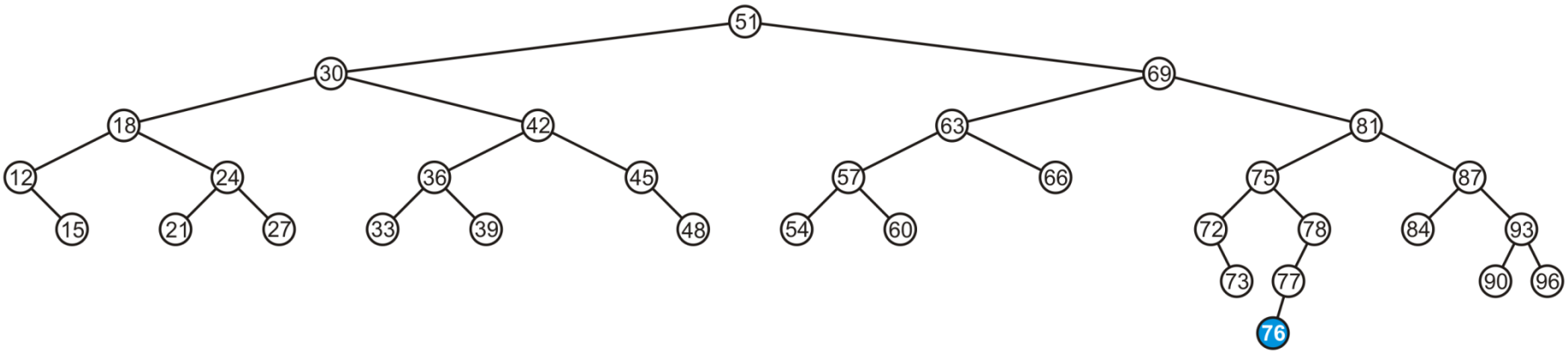
Insertion

The tree is balanced



Insertion

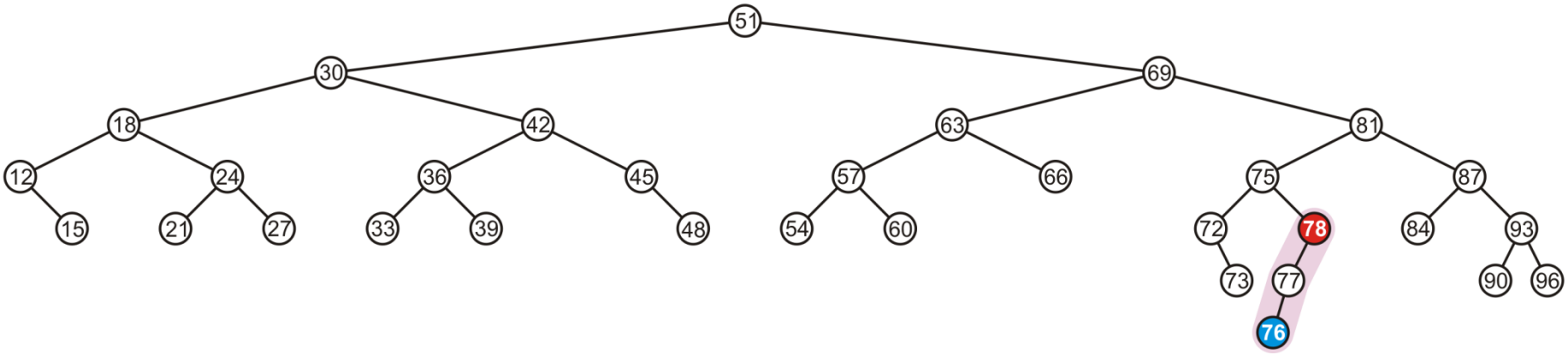
Insert 76



Insertion

The node 78 is unbalanced

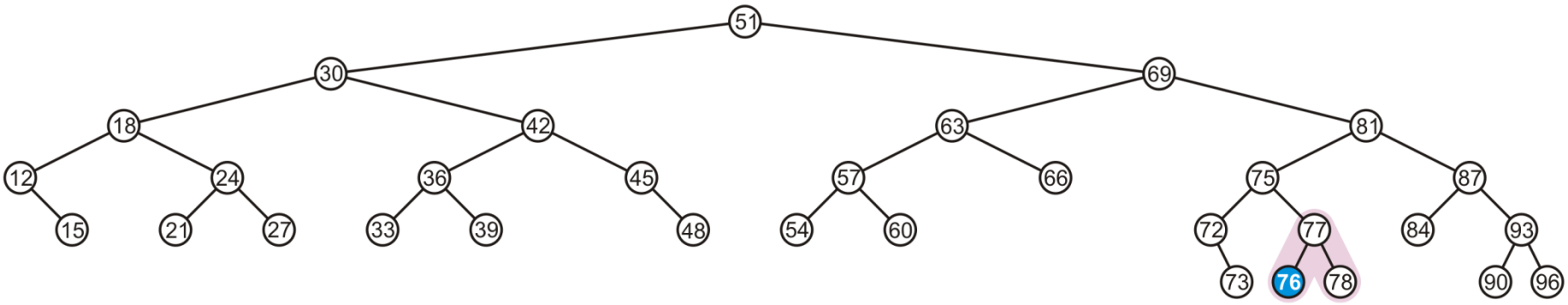
- A left-left imbalance



Insertion

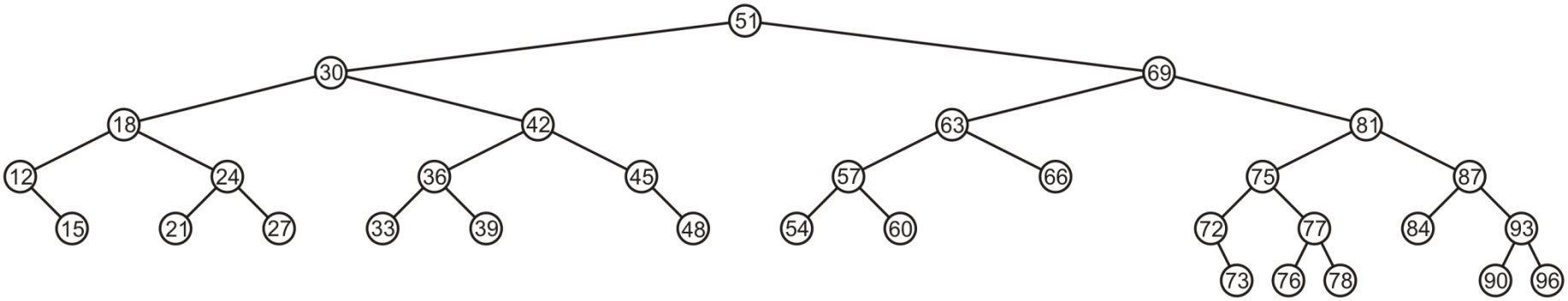
The node 78 is unbalanced

- Promote 77



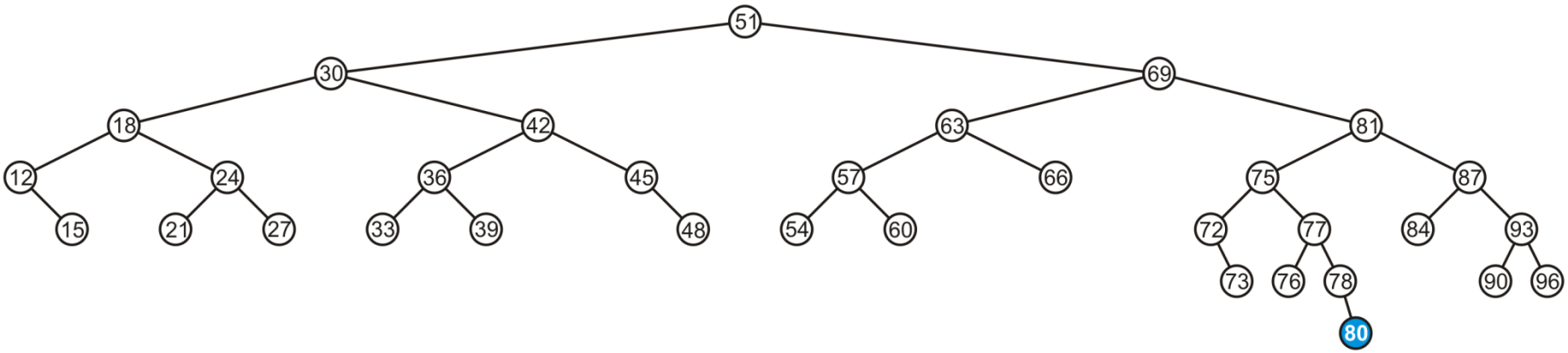
Insertion

Again, balanced



Insertion

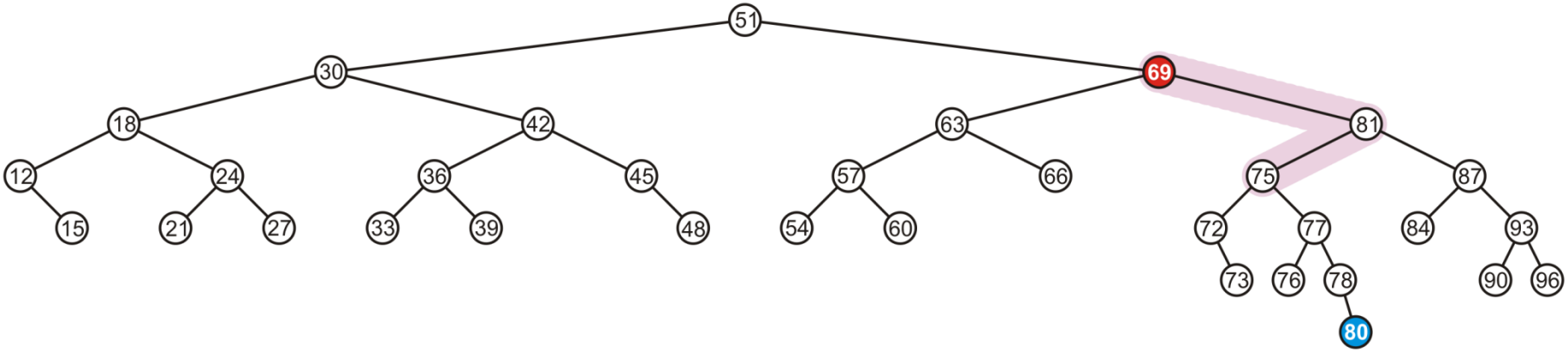
Insert 80



Insertion

The node 69 is unbalanced

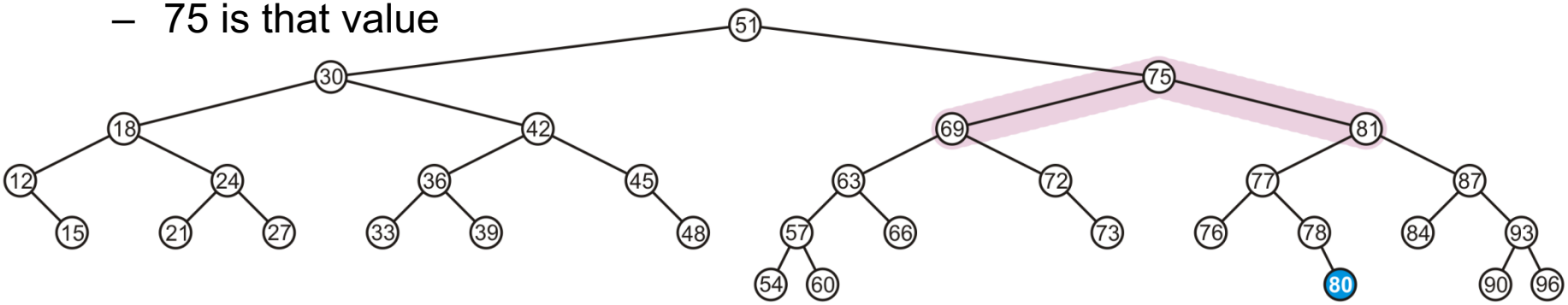
- A right-left imbalance
- Promote the intermediate node to the imbalanced node



Insertion

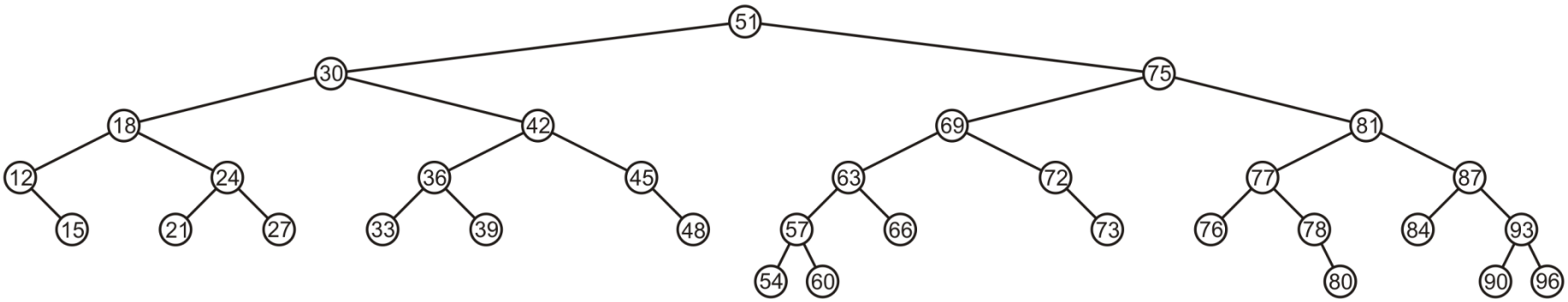
The node 69 is unbalanced

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



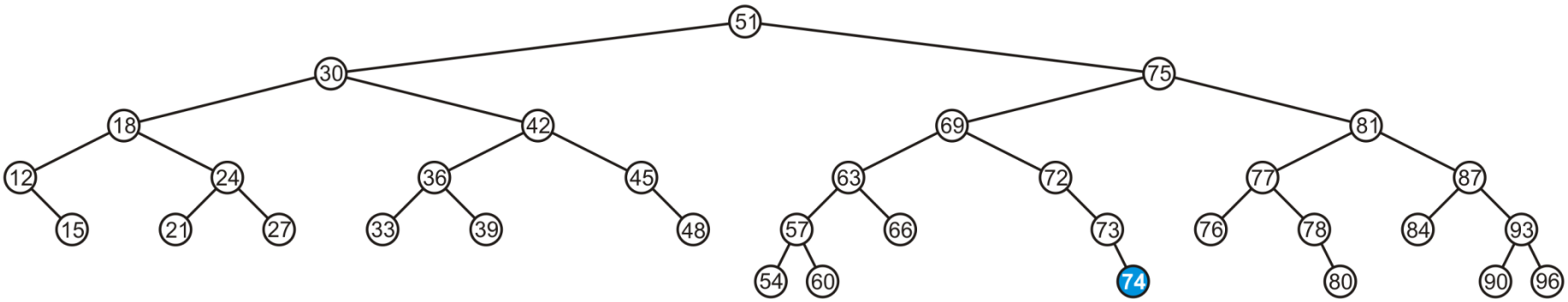
Insertion

Again, balanced



Insertion

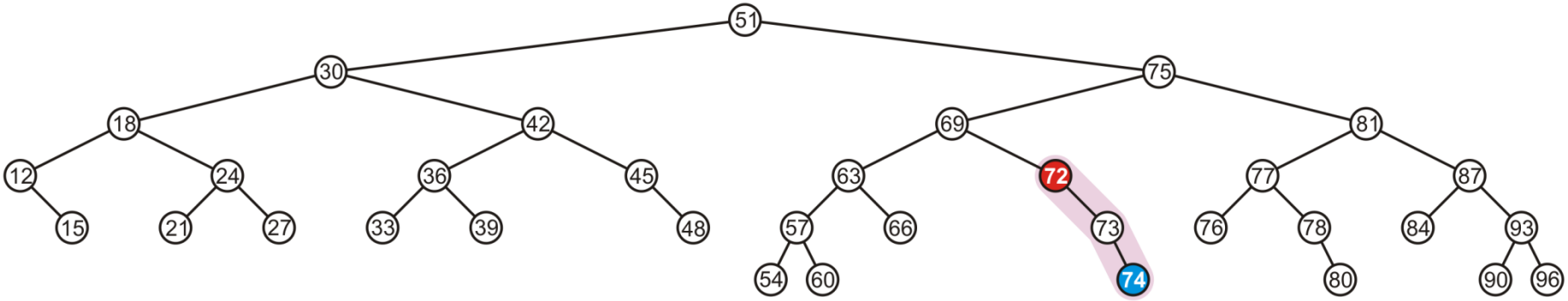
Insert 74



Insertion

The node 72 is unbalanced

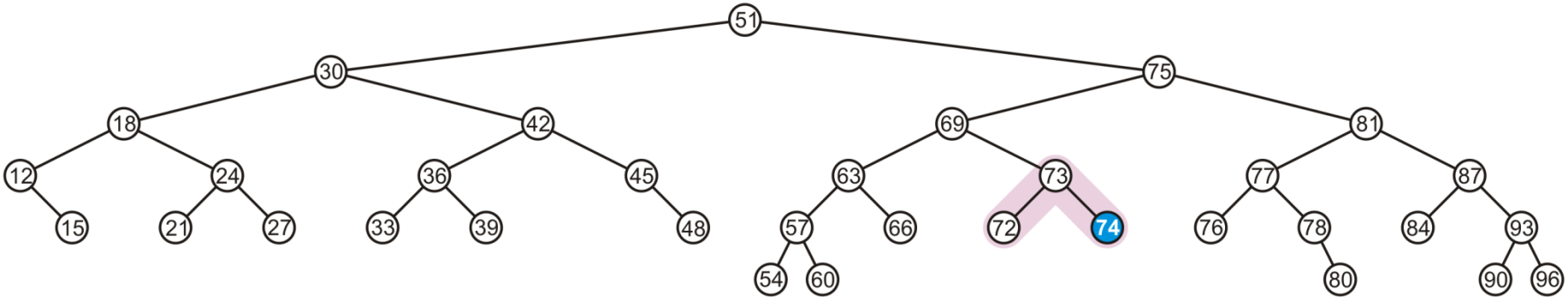
- A right-right imbalance
- Promote the intermediate node to the imbalanced node



Insertion

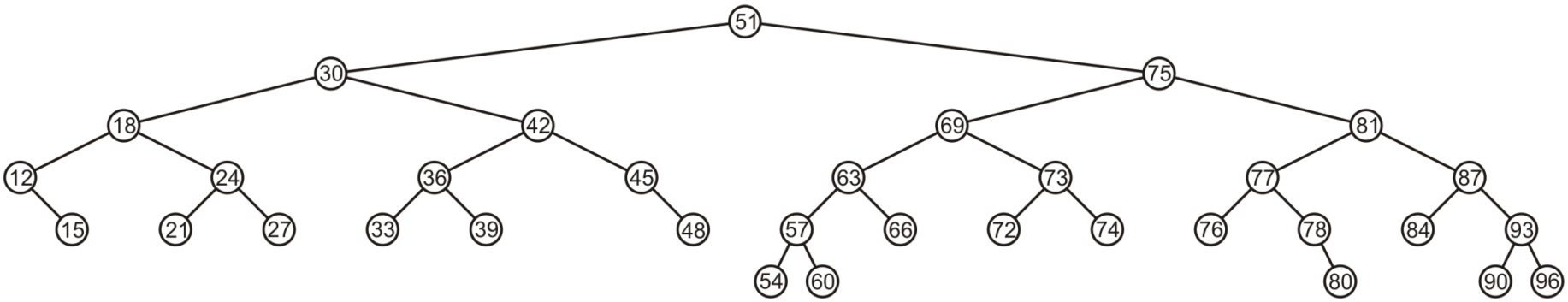
The node 72 is unbalanced

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



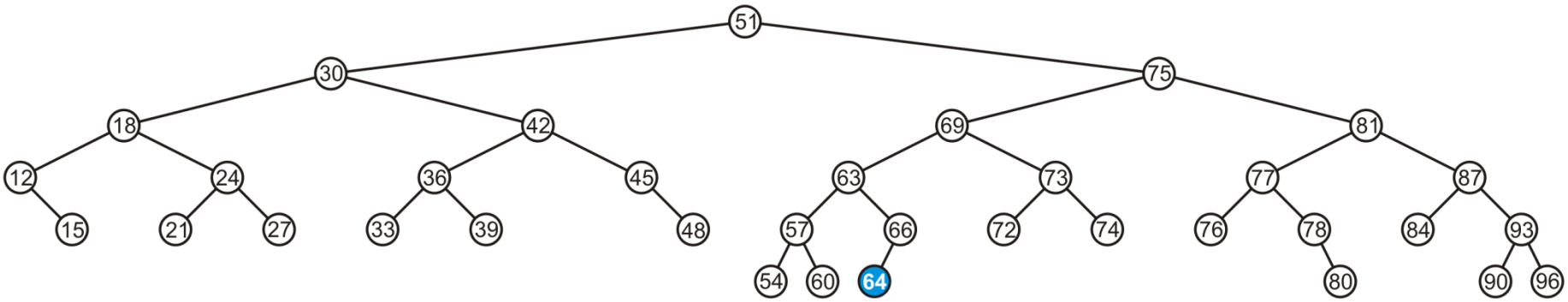
Insertion

Again, balanced



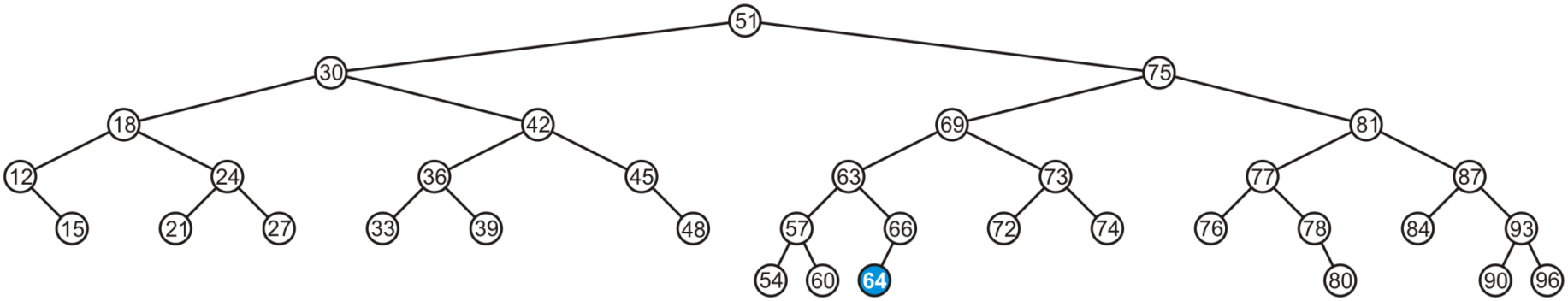
Insertion

Insert 64



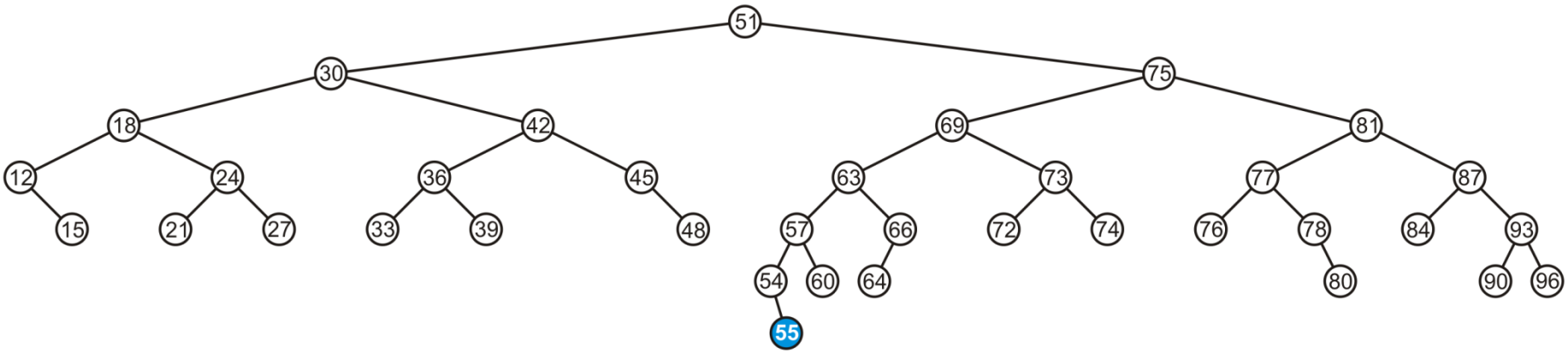
Insertion

This causes no imbalances



Insertion

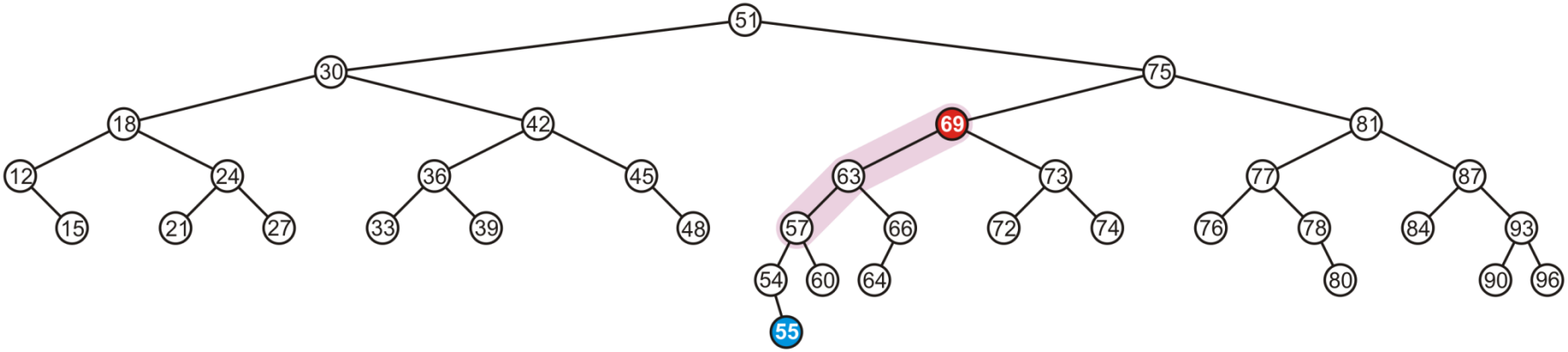
Insert 55



Insertion

The node 69 is imbalanced

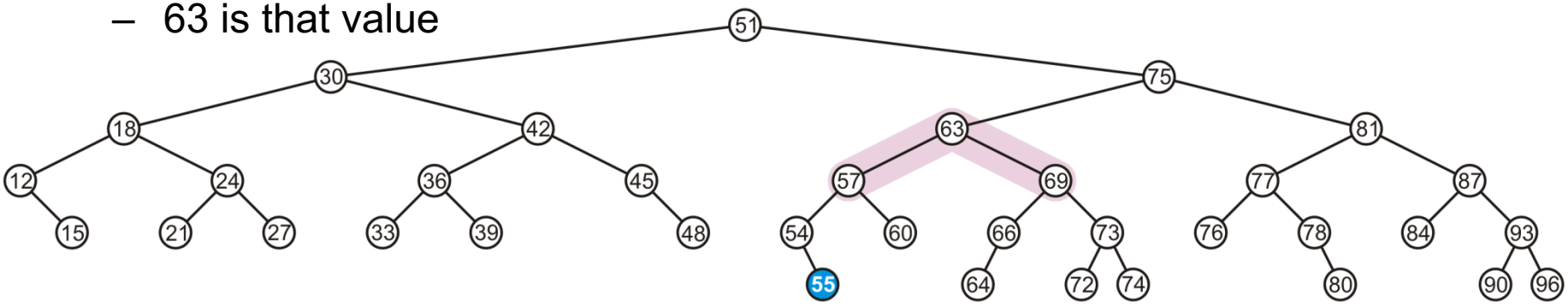
- A left-left imbalance
- Promote the intermediate node to the imbalanced node



Insertion

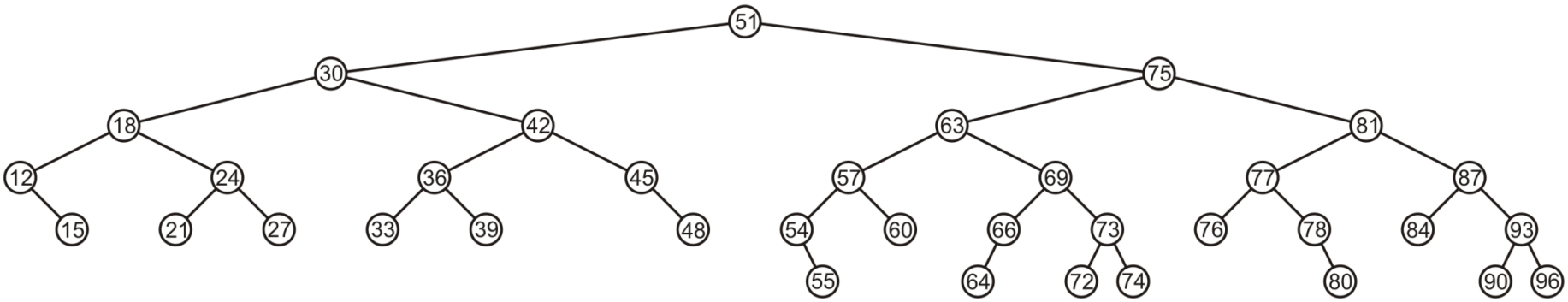
The node 69 is imbalanced

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



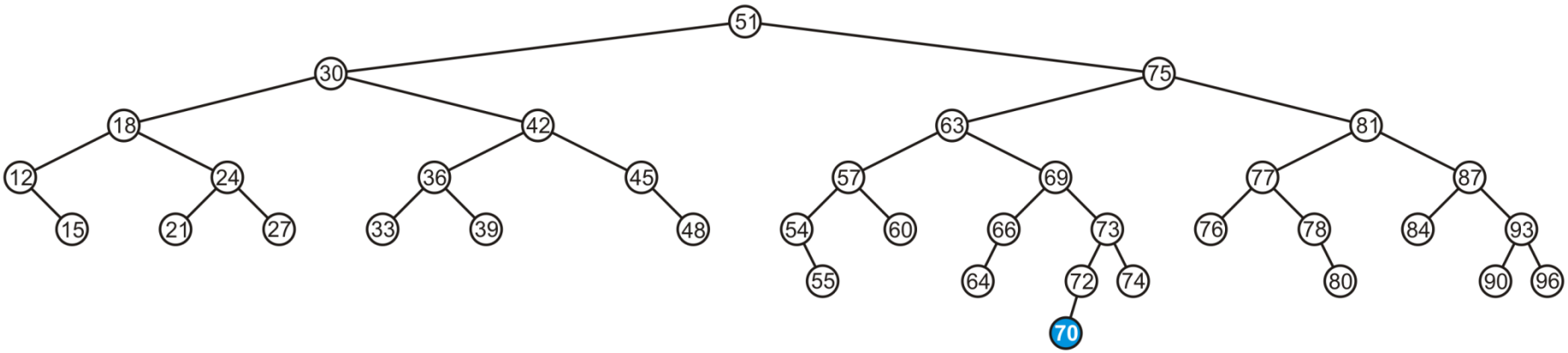
Insertion

The tree is now balanced



Insertion

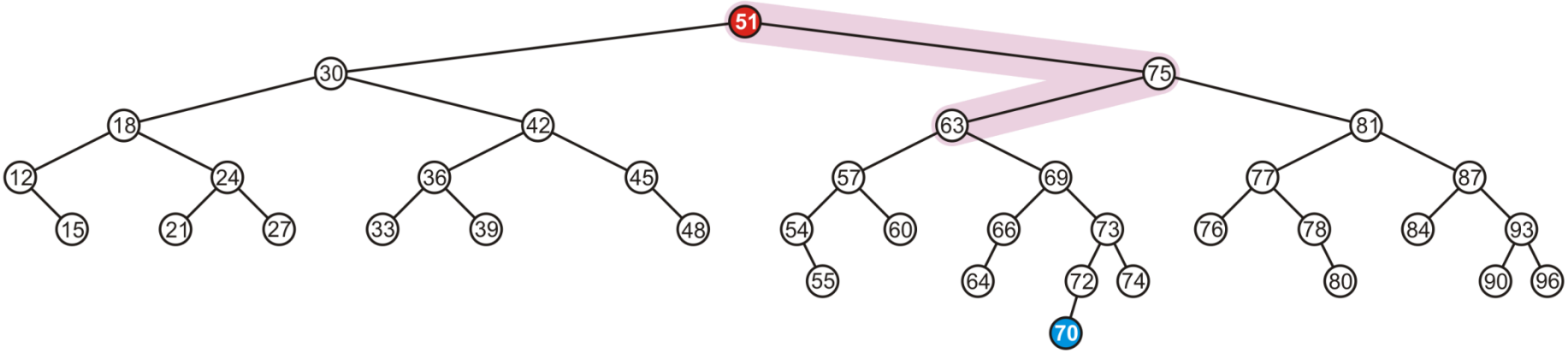
Insert 70



Insertion

The root node is now imbalanced

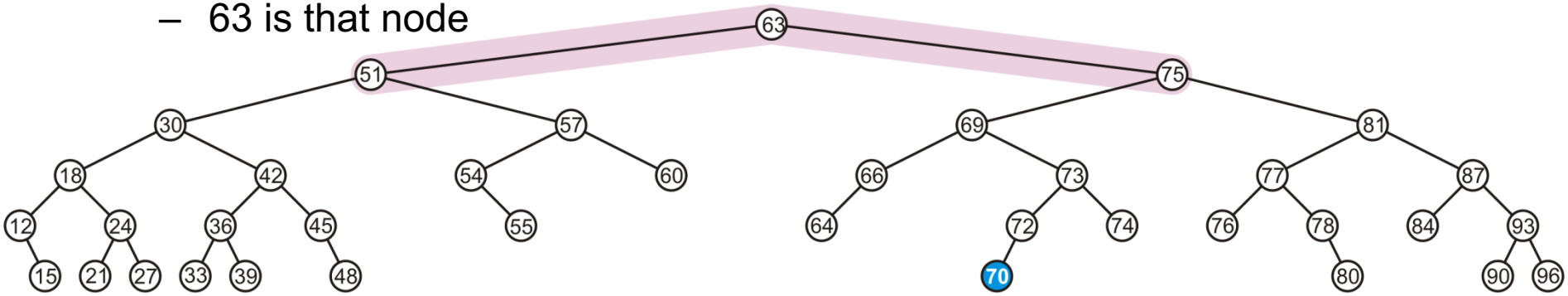
- A right-left imbalance
- Promote the intermediate node to the root



Insertion

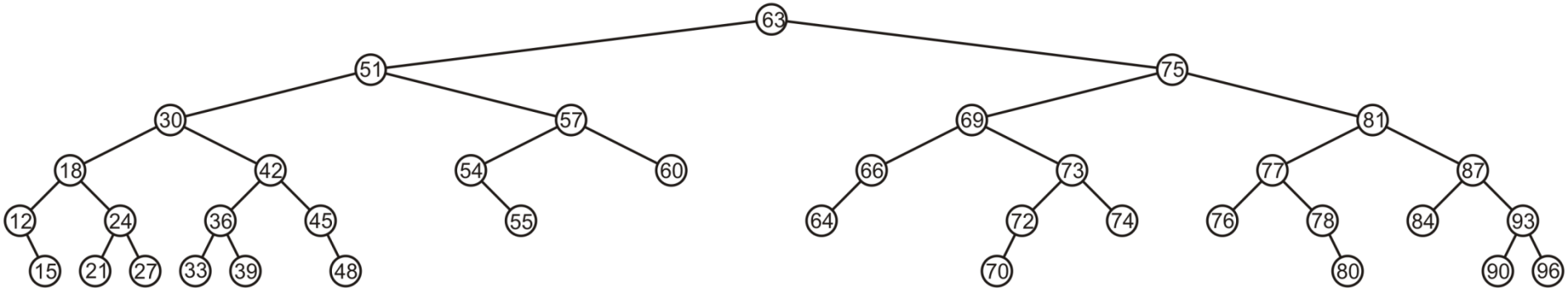
The root node is imbalanced

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



Insertion

The result is AVL balanced



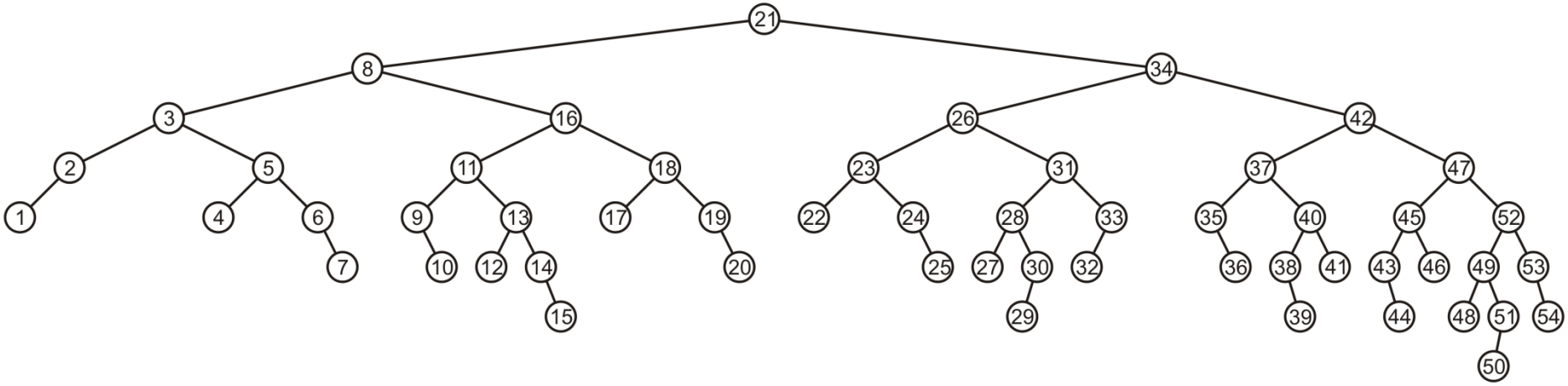
Delete

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

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

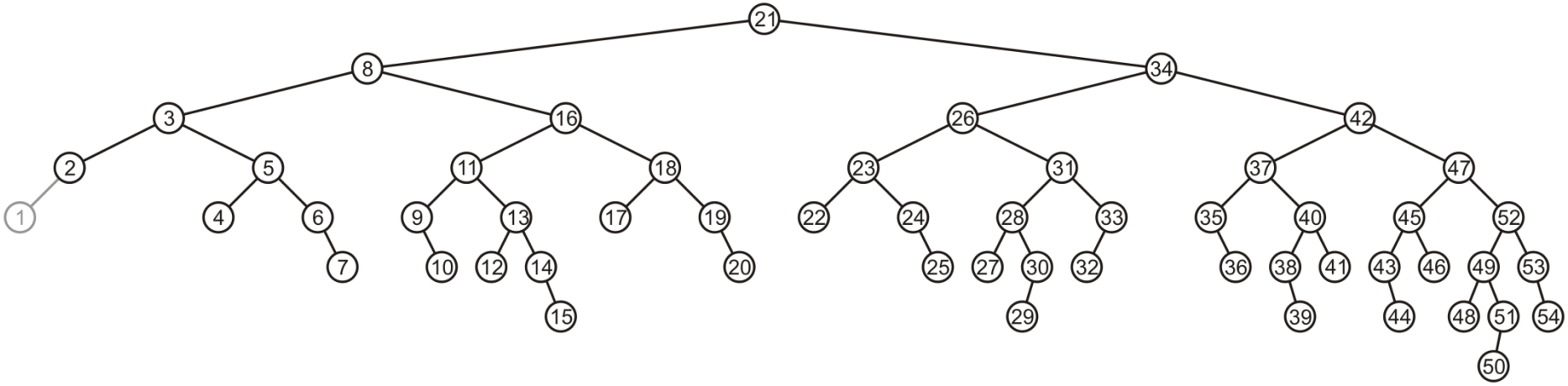
Delete

Consider the following AVL tree



Delete

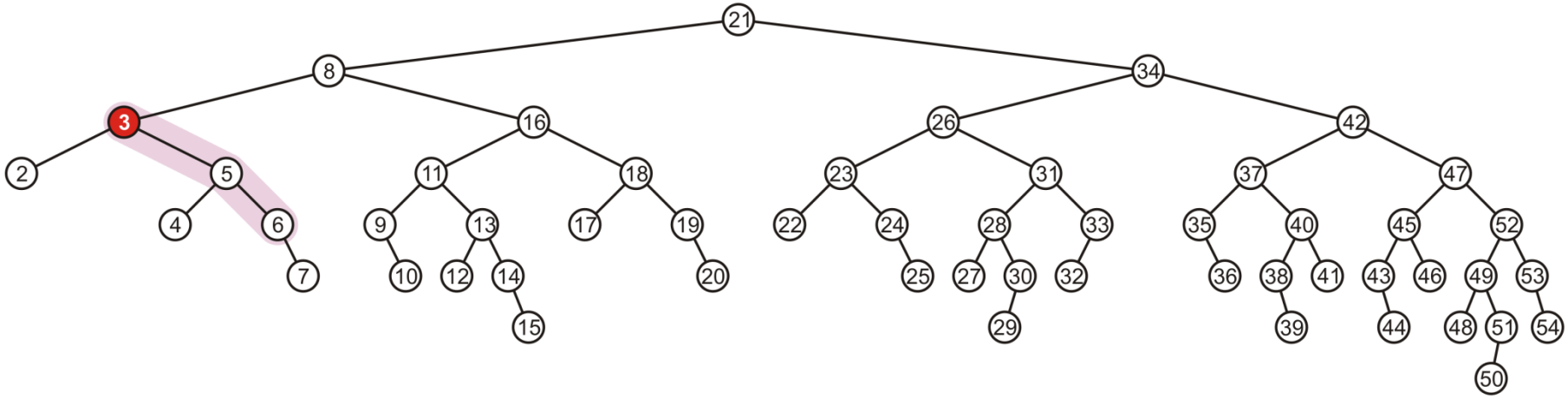
Suppose we Delete the front node: 1



Delete

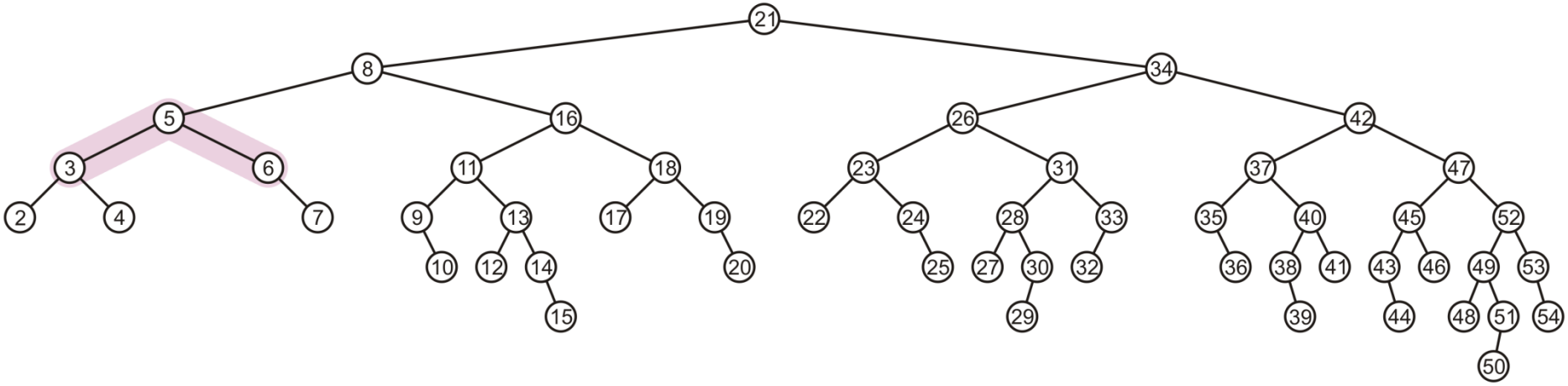
While its previous parent, 2, is not unbalanced, its grandparent 3 is

- The imbalance is in the right-right subtree



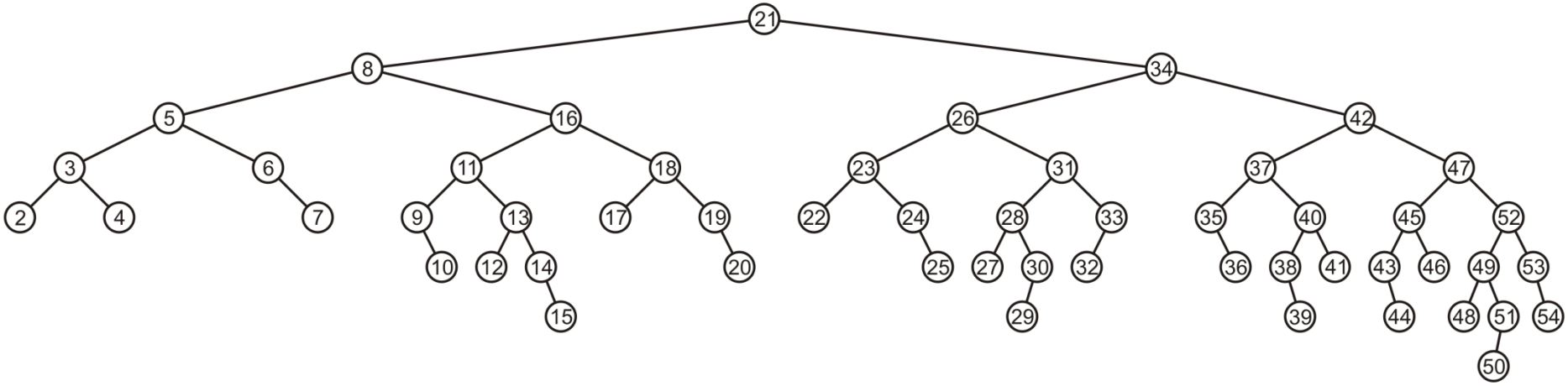
Delete

We can correct this with a simple balance



Delete

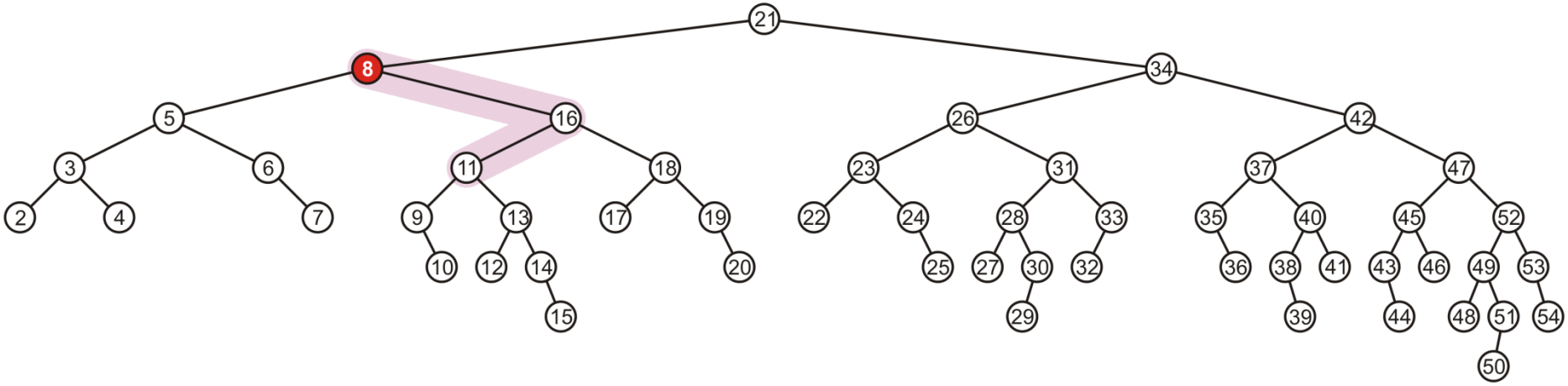
The node of that subtree, 5, is now balanced



Delete

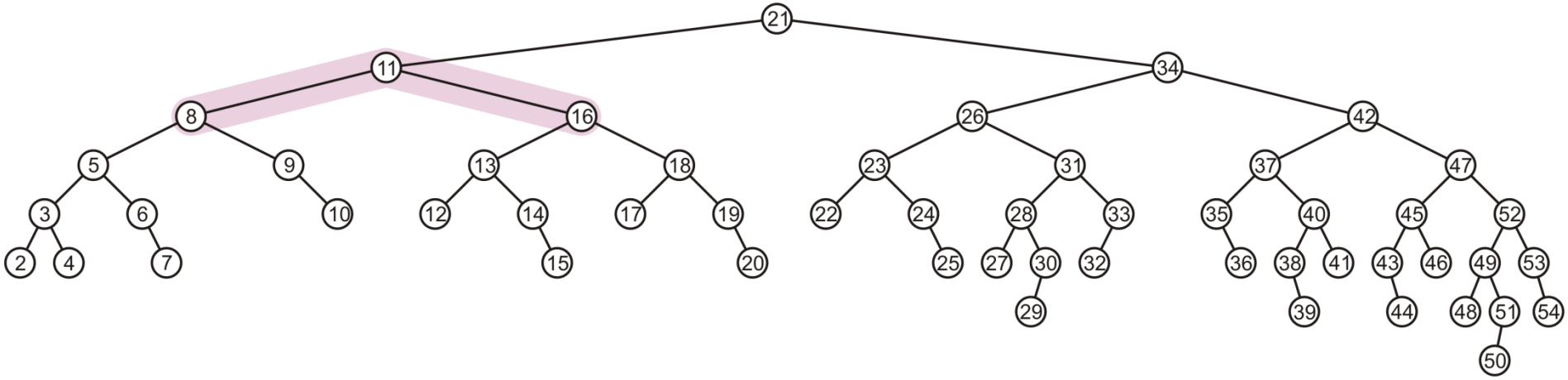
Recursing to the root, however, 8 is also unbalanced

- This is a right-left imbalance



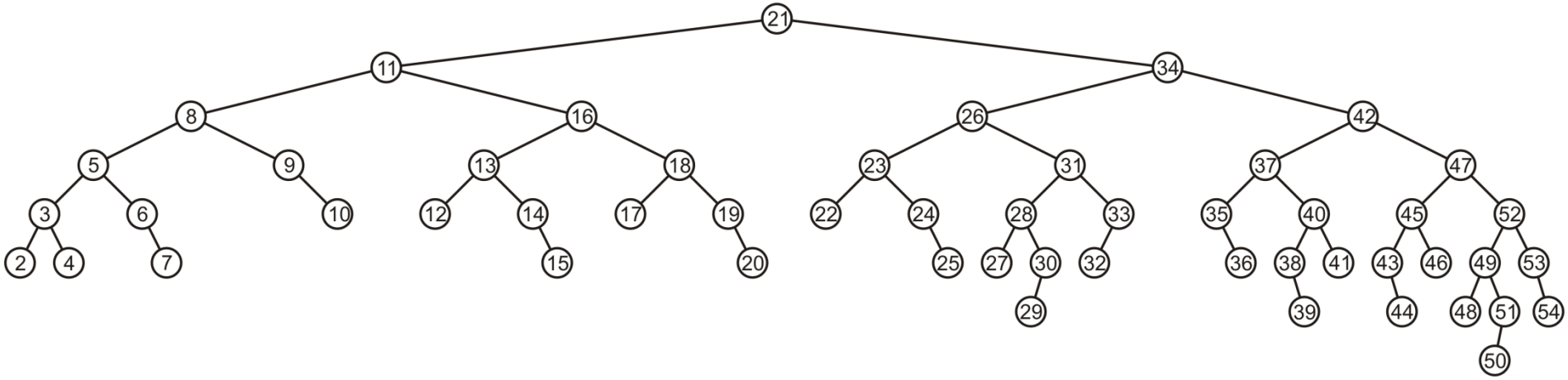
Delete

Promoting 11 to the root corrects the imbalance



Delete

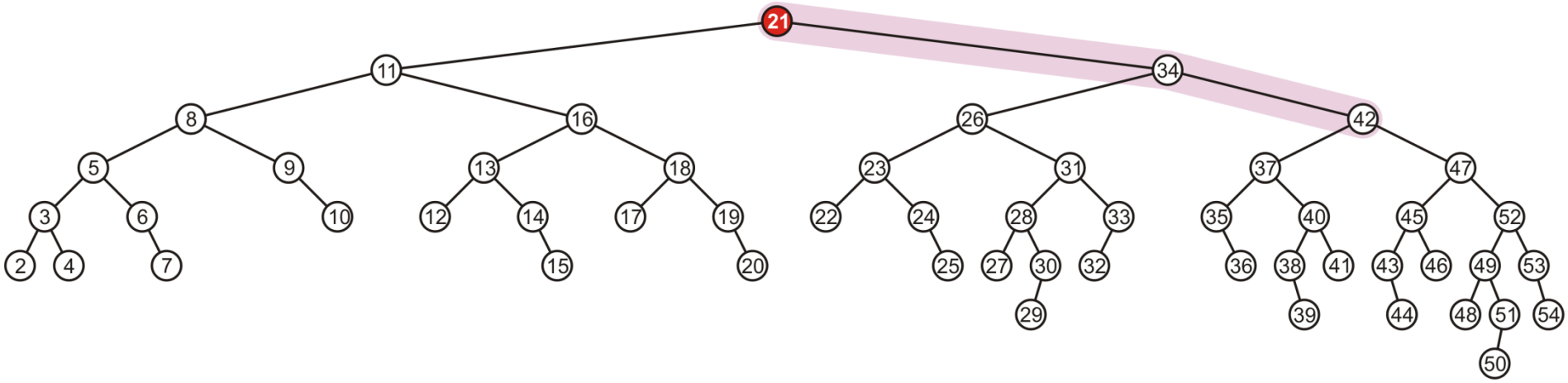
At this point, the node 11 is balanced



Delete

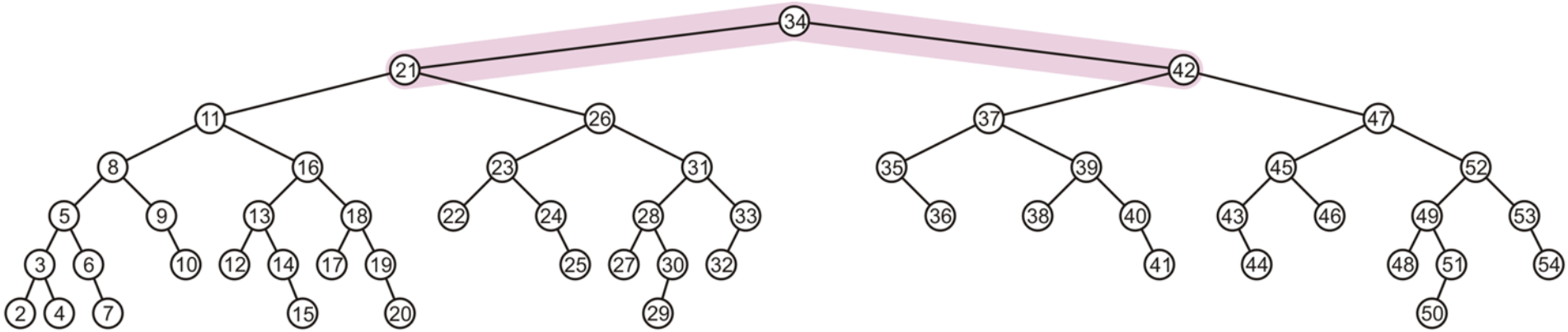
Still, the root node is unbalanced

- This is a right-right imbalance



Delete

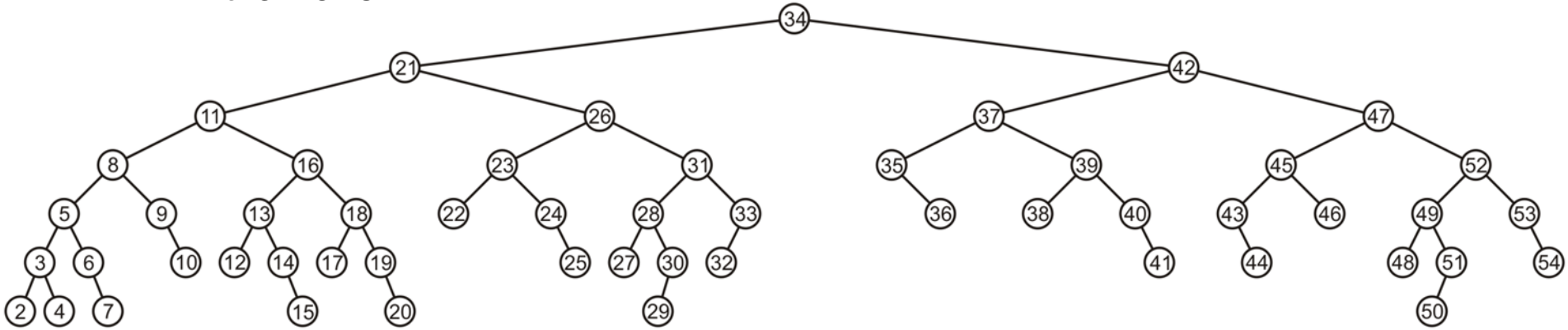
Again, a simple balance fixes the imbalance



Delete

The resulting tree is now AVL balanced

- Note, few Deletes will require one balance, even fewer will require more than one



IMPLEMENTATION OF AVL TREES
Recommended Text Book
Python Code

AVL.py

```
from .binary_search_tree import TreeMap

class AVLTreeMap(TreeMap):
  """Sorted map implementation using an AVL tree."""

  #-------------------------- nested _Node class -------------------------
  class _Node(TreeMap._Node):
    """Node class for AVL maintains height value for balancing.

    We use convention that a "None" child has height 0, thus a leaf has height 1.
    """
    __slots__ = '_height'       # additional data member to store height

    def __init__(self, element, parent=None, left=None, right=None):
      super().__init__(element, parent, left, right)
      self._height = 0          # will be recomputed during balancing

    def left_height(self):
      return self._left._height if self._left is not None else 0

    def right_height(self):
      return self._right._height if self._right is not None else 0
```

AVL.py

```
#----------------------- positional-based utility methods -----------------------
 def _recompute_height(self, p):
   p._node._height = 1 + max(p._node.left_height(), p._node.right_height())

 def _isbalanced(self, p):
   return abs(p._node.left_height() - p._node.right_height()) <= 1

 def _tall_child(self, p, favorleft=False): # parameter controls tiebreaker
   if p._node.left_height() + (1 if favorleft else 0) > p._node.right_height():
     return self.left(p)
   else:
     return self.right(p)

 def _tall_grandchild(self, p):
   child = self._tall_child(p)
   # if child is on left, favor left grandchild; else favor right grandchild
   alignment = (child == self.left(p))
   return self._tall_child(child, alignment)
```

AVL.py

```
def _rebalance(self, p):
  while p is not None:
    old_height = p._node._height                    # trivially 0 if new node
    if not self._isbalanced(p):                     # imbalance detected!
      # perform trinode restructuring, setting p to resulting root,
      # and recompute new local heights after the restructuring
      p = self._restructure(self._tall_grandchild(p))
      self._recompute_height(self.left(p))
      self._recompute_height(self.right(p))
    self._recompute_height(p)                       # adjust for recent changes
    if p._node._height == old_height:               # has height changed?
      p = None                                      # no further changes needed
    else:
      p = self.parent(p)                            # repeat with parent

 #------------------------- override balancing hooks -------------------------
 def _rebalance_insert(self, p):
   self._rebalance(p)

 def _rebalance_delete(self, p):
   self._rebalance(p)
```

Summary

In this topic we have covered:

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

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