AVL Trees

COL 106

Amit Kumar

Shweta Agrawal

Slide Courtesy : Douglas Wilhelm Harder, MMath, UWaterloo

dwharder@alumni.uwaterloo.ca

Background

So far …

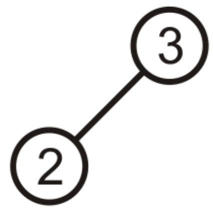
- Binary search trees store linearly ordered data
- Best case height: $\mathbf{\Theta}(\ln(n))$
- Worst case height: $\mathbf{O}(n)$

Requirement:

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

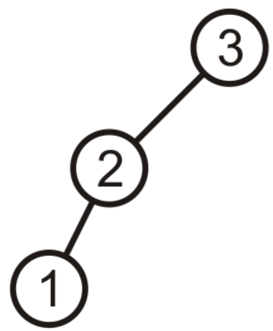
Prototypical Examples

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



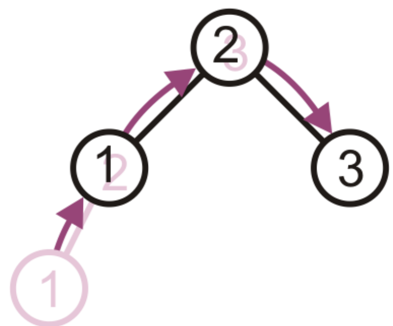
Prototypical Examples

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



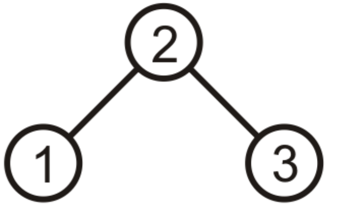
Prototypical Examples

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



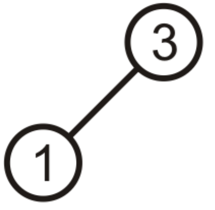
Prototypical Examples

The result is a perfect tree



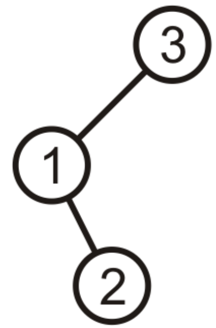
Prototypical Examples

Alternatively, given this tree, insert 2



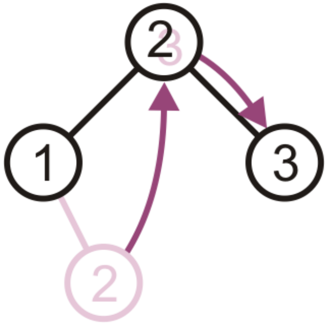
Prototypical Examples

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



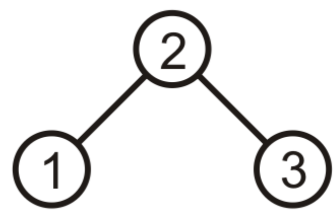
Prototypical Examples

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

Notion of balance in AVL trees?

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

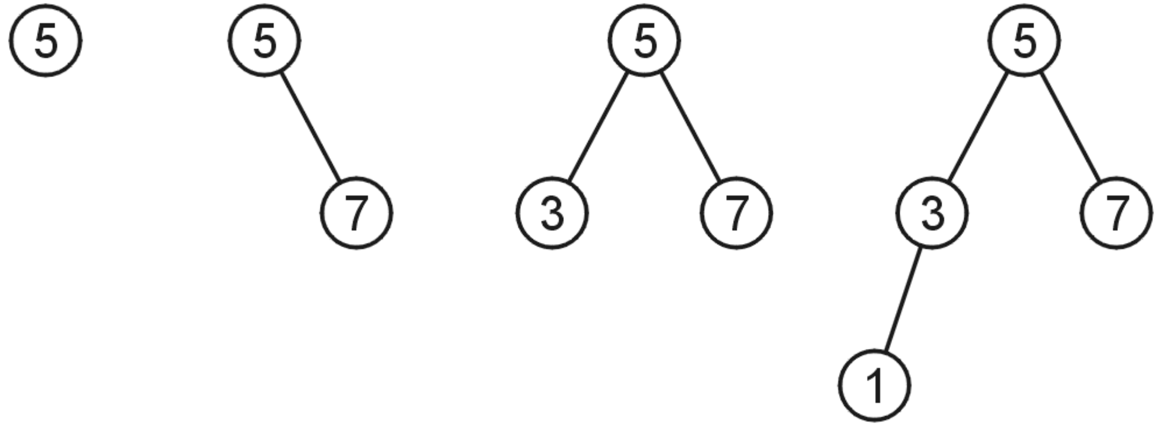
AVL Trees

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

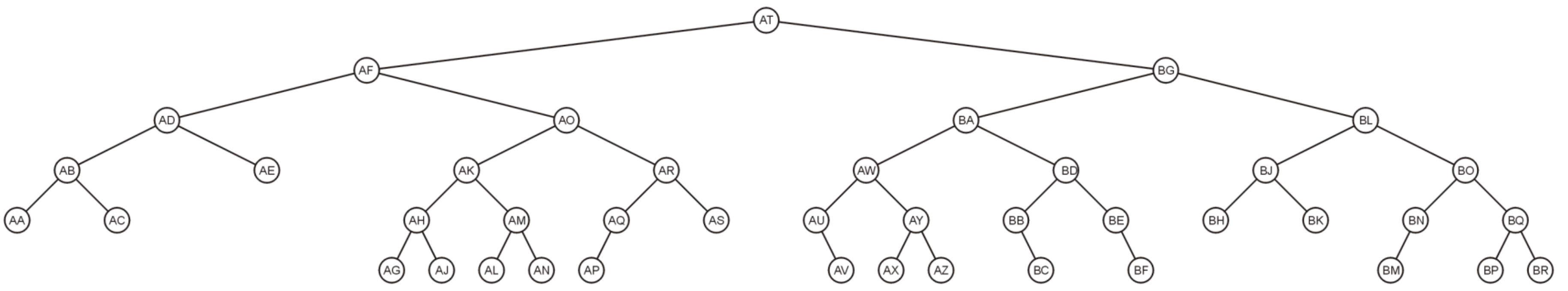
AVL Trees

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



AVL Trees

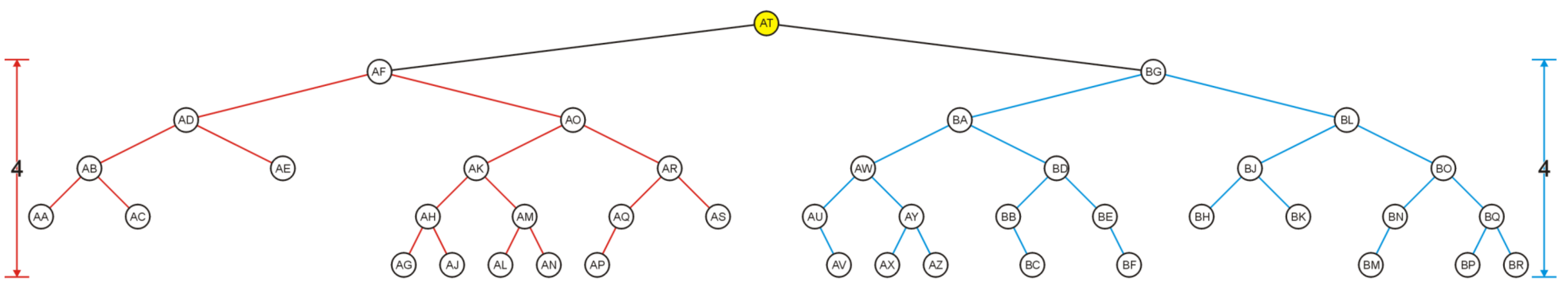
Here is a larger AVL tree (42 nodes):



AVL Trees

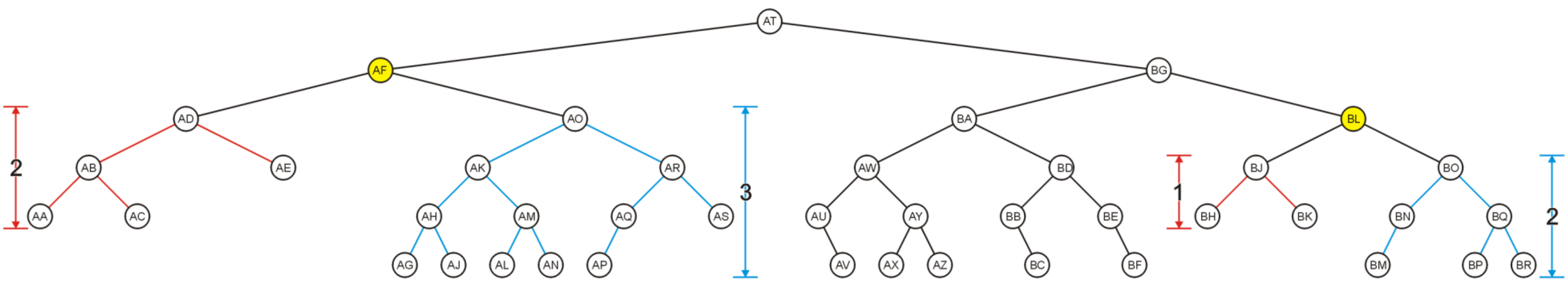
The root node is AVL-balanced:

– Both sub-trees are of height 4:



AVL Trees

- All other nodes 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

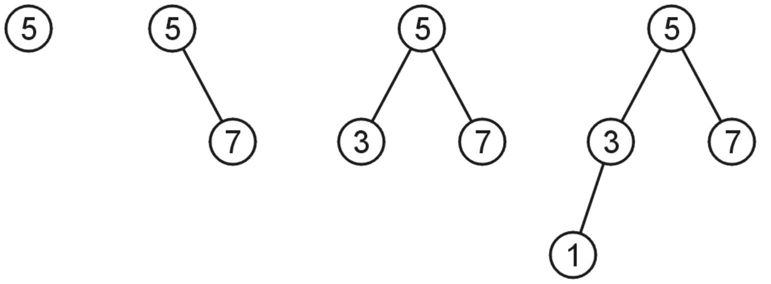
Thus an upper bound on the number of nodes in an AVL tree of height h

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

– What is a 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: $\mathrm{F}(h) = \mathrm{F}(h - 1) + 1 + \mathrm{F}(h - 2)$

Height of an AVL Tree

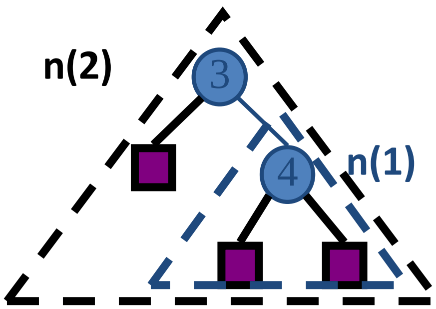
This is a recurrence relation:

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

The solution?

Height of an AVL Tree

- **Fact**: The ***height*** of an AVL tree storing n keys is O(log n).
- **Proof**: Let us bound **n(h):** the minimum number of internal nodes of an AVL tree of height h.
- We easily see that n(1) = 1 and n(2) = 2
- For n > 2, an AVL tree of height h contains the root node, one AVL subtree of height h-1 and another of height h-2.
- That is, n(h) = 1 + n(h-1) + n(h-2)
- Knowing n(h-1) > n(h-2), we get n(h) > 2n(h-2). So
  - n(h) > 2n(h-2), n(h) > 4n(h-4), n(h) > 8n(n-6), … (by induction),
  - n(h) > 2^in(h-2i)
- Solving the base case we get: n(h) > $2^{h/2-1}$
- Taking logarithms: h < 2log n(h) +2
- Thus the height of an AVL tree is O(log n)



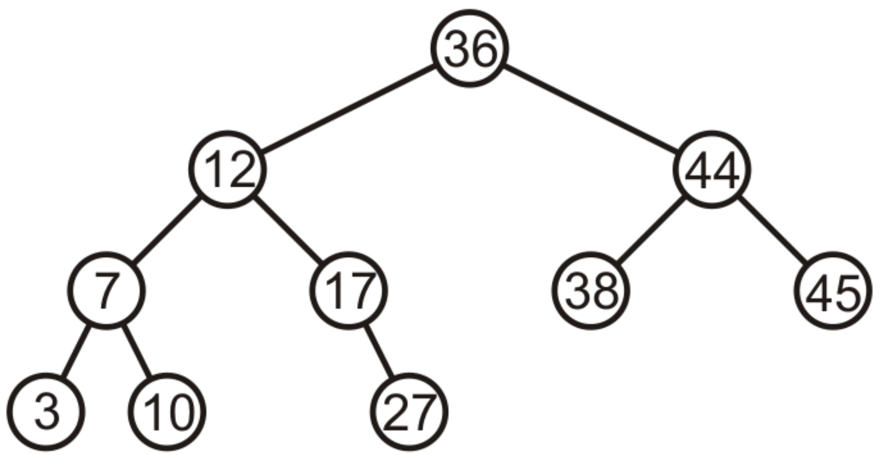
Maintaining Balance

To maintain AVL balance, observe that:

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

Maintaining Balance

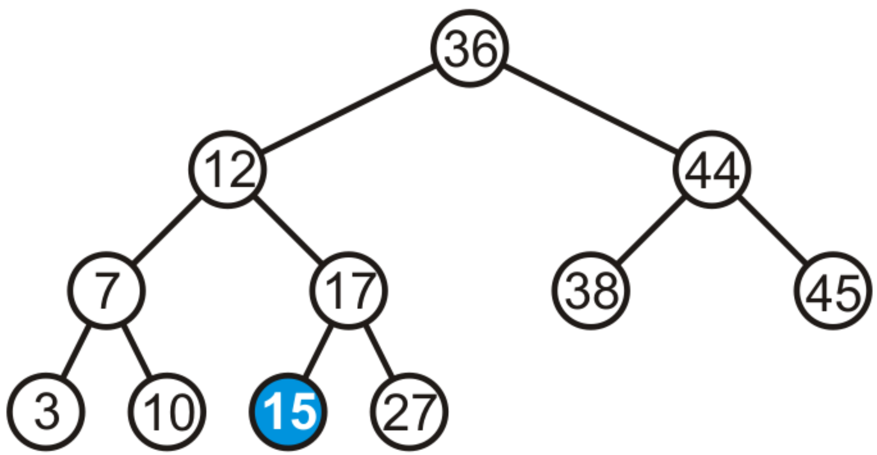
Consider this AVL tree



Maintaining Balance

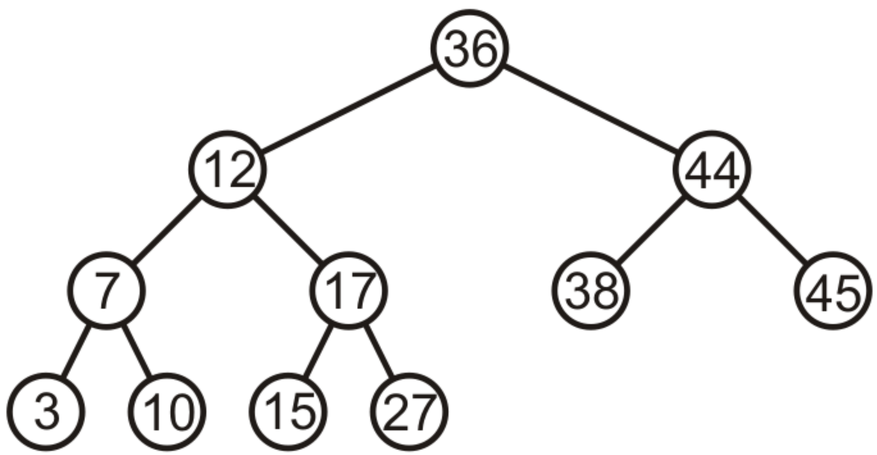
Consider inserting 15 into this tree

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



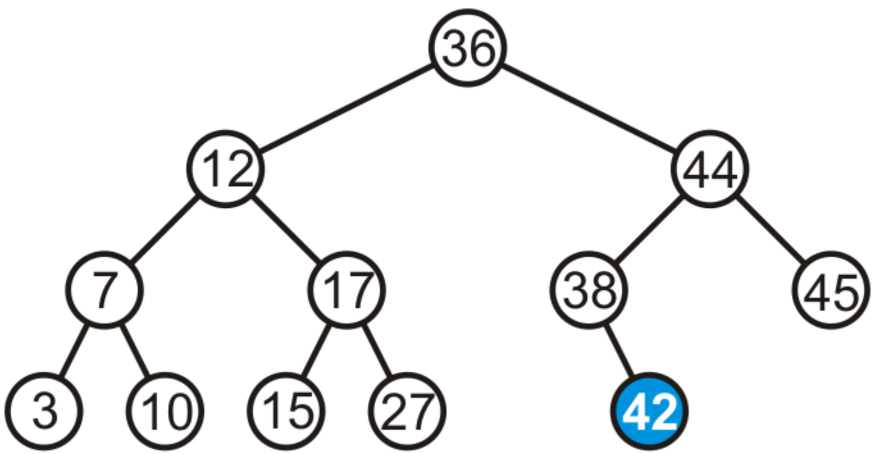
Maintaining Balance

The tree remains balanced



Maintaining Balance

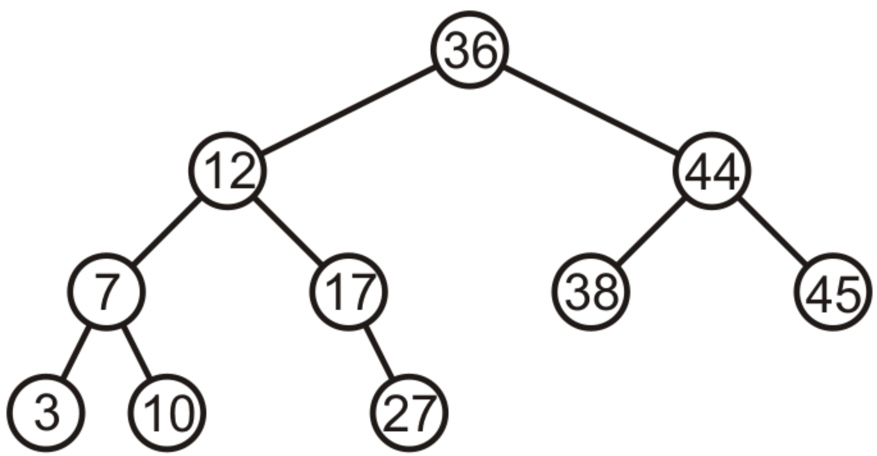
- Consider inserting 42 into this tree
  - In this case, the heights of none of the trees change



Maintaining Balance

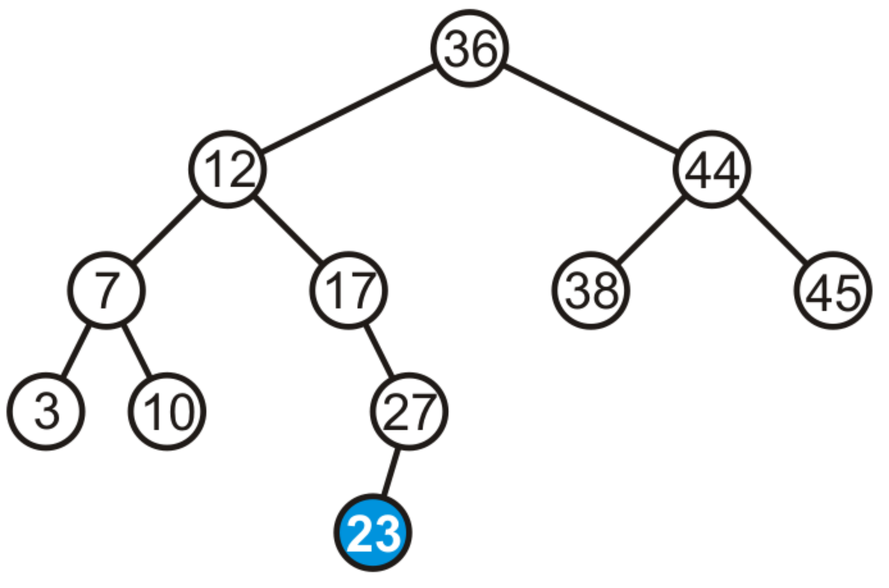
If a tree is AVL balanced, for an insertion to cause an imbalance:

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



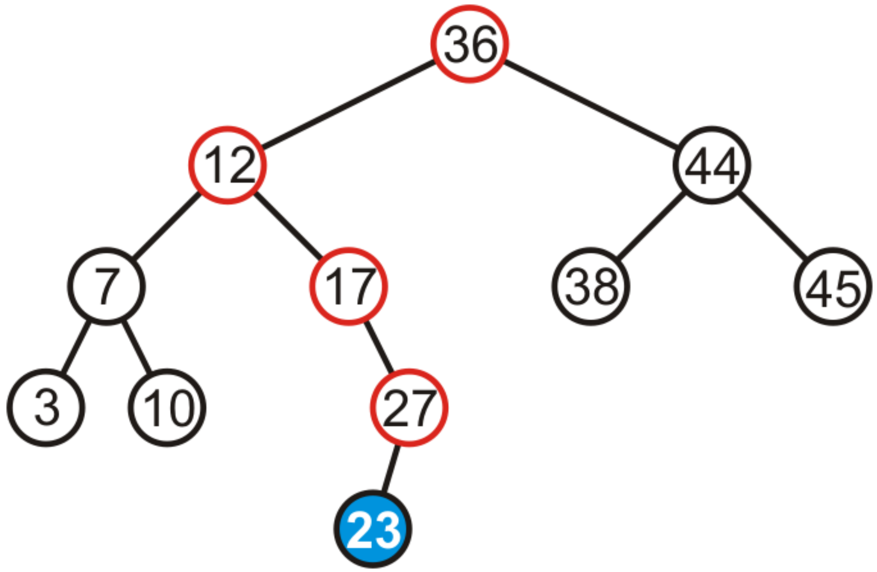
Maintaining Balance

Suppose we insert 23 into our initial tree



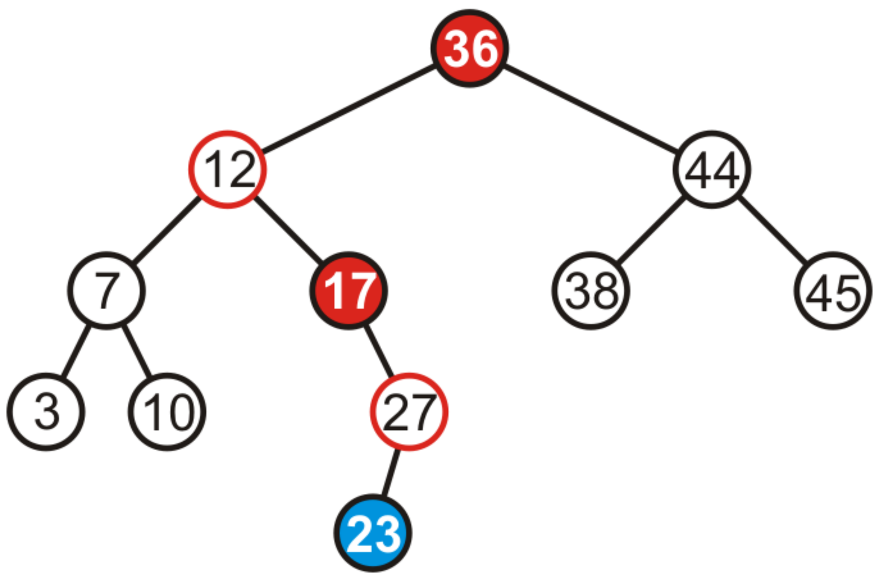
Maintaining Balance

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



Maintaining Balance

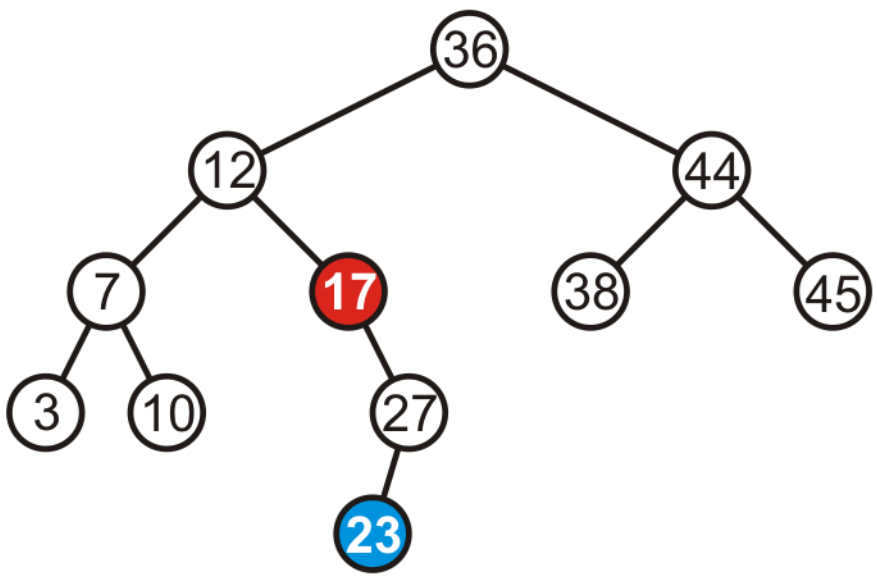
However, only two of the nodes are unbalanced: 17 and 36



Maintaining Balance

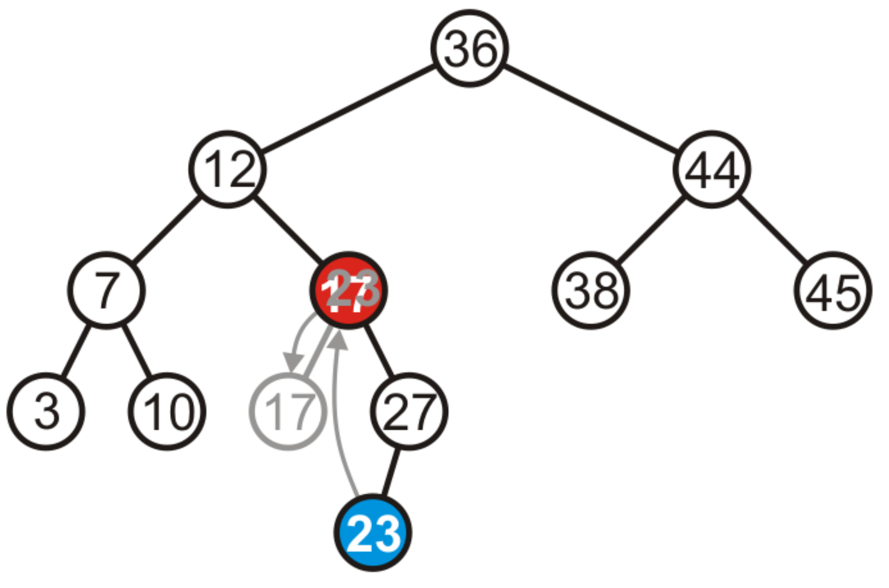
However, only two of the nodes are unbalanced: 17 and 36

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



Maintaining Balance

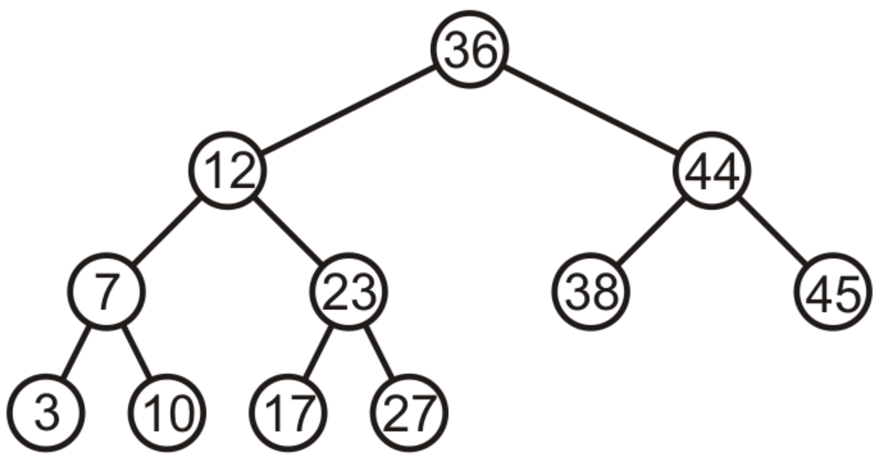
We can promote 23 to where 17 is, and make 17 the left child of 23



Maintaining Balance

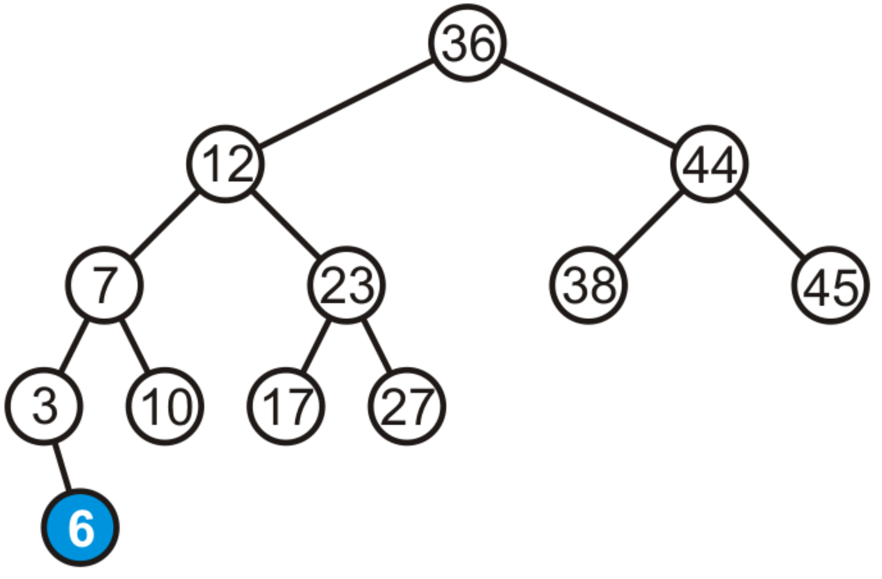
Thus, that node is no longer unbalanced

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



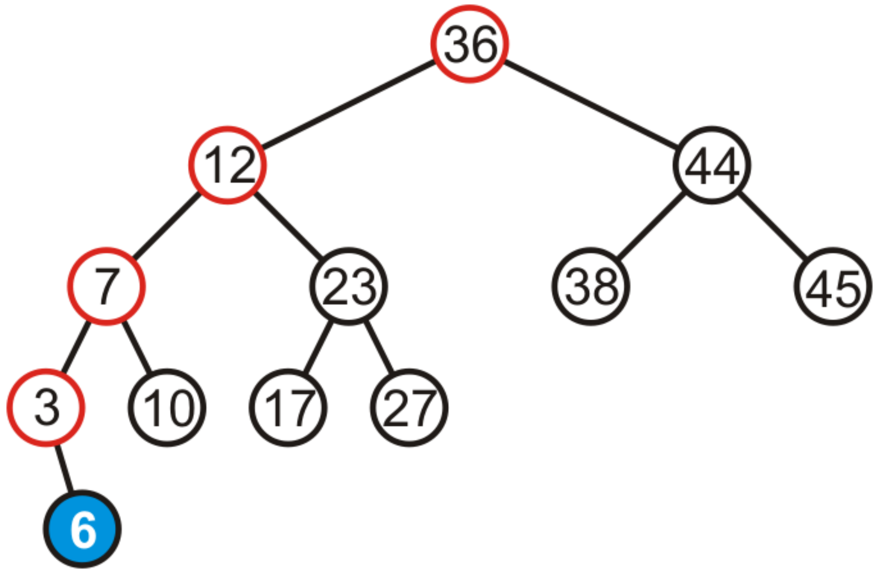
Maintaining Balance

Consider adding 6:



Maintaining Balance

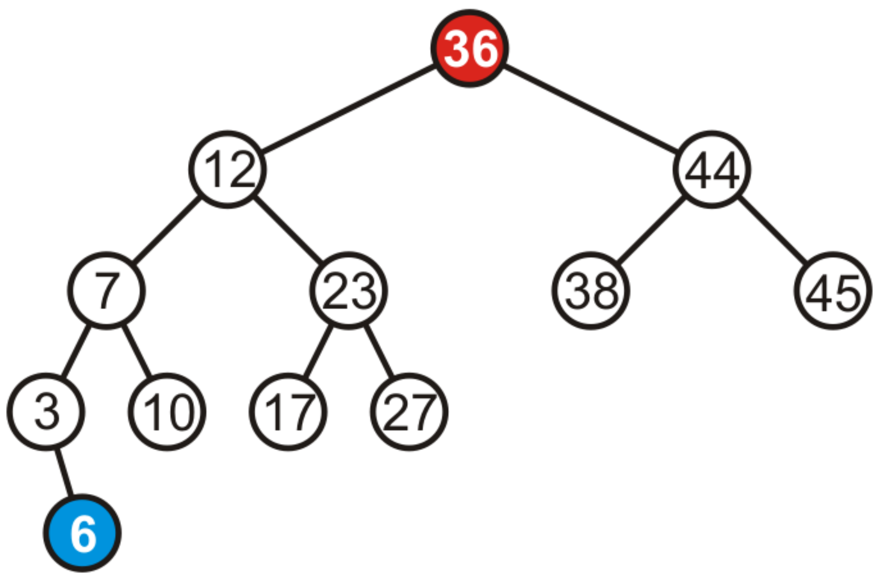
The height of each of the trees in the path back to the root are increased by one



Maintaining Balance

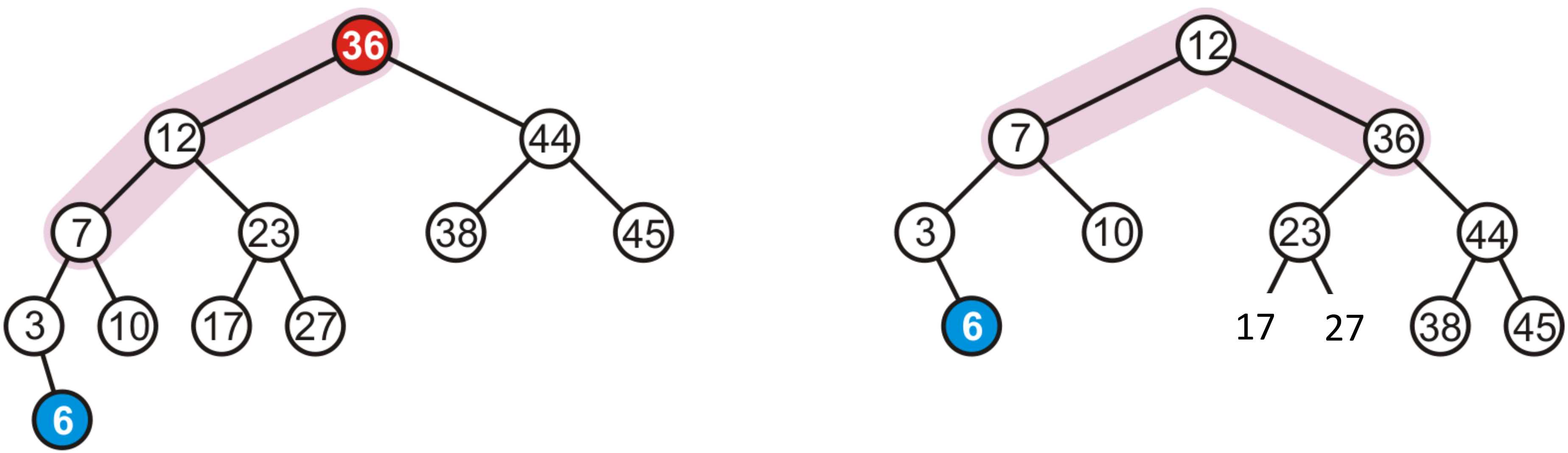
The height of each of the trees in the path back to the root are increased by one

– However, only the root node is now unbalanced



Maintaining Balance

We may fix this by rotating the root to the right

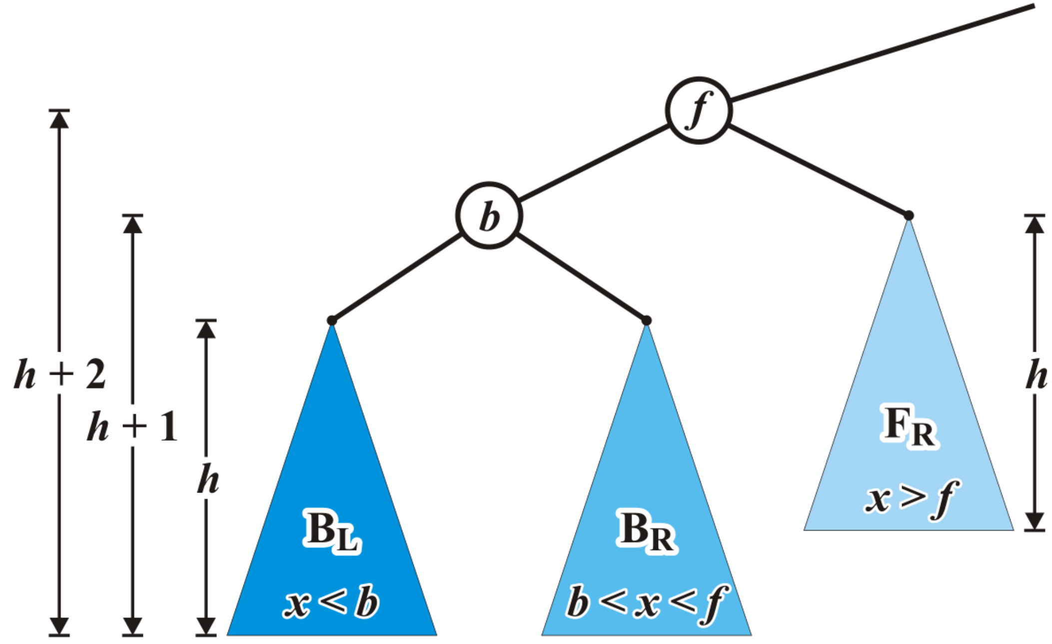


Note: the right subtree of 12 became the left subtree of 36

Case 1 setup

Consider the following setup

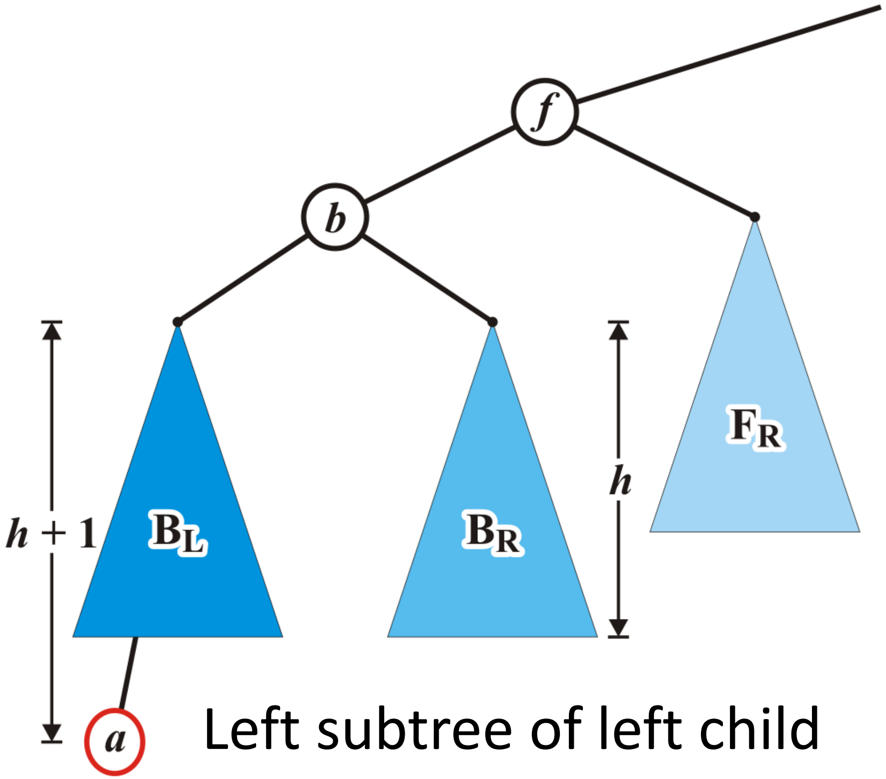
– Each blue triangle represents a tree of height h



Maintaining Balance: Case 1

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

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

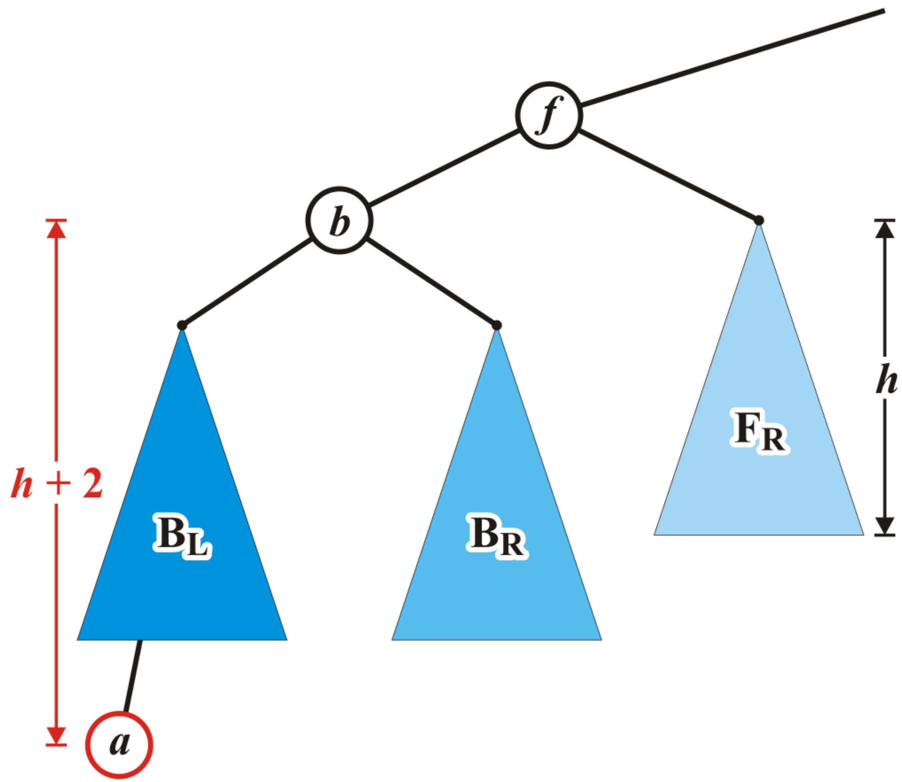


Left subtree of left child

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

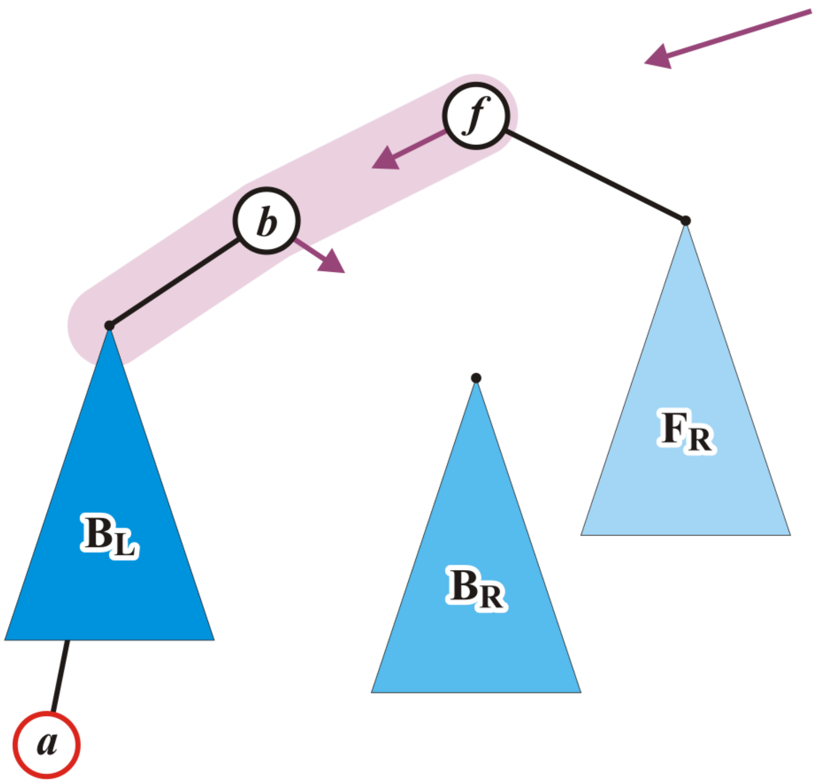
We will modify three pointers:

Maintaining Balance: Case 1

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

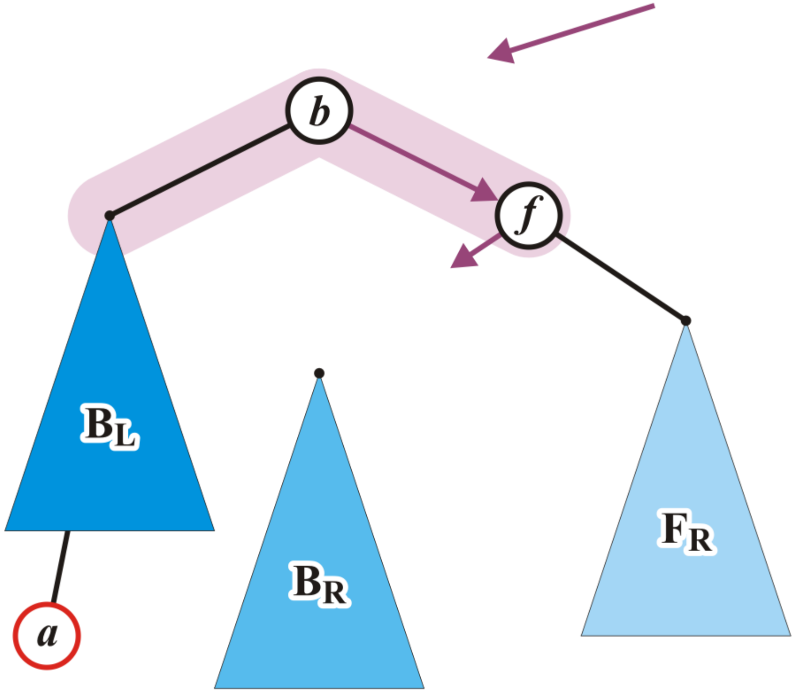
– Recall the first prototypical example

– Promote node ***b*** to the root and demote node ***f*** to be the right child of ***b***



Maintaining Balance: Case 1

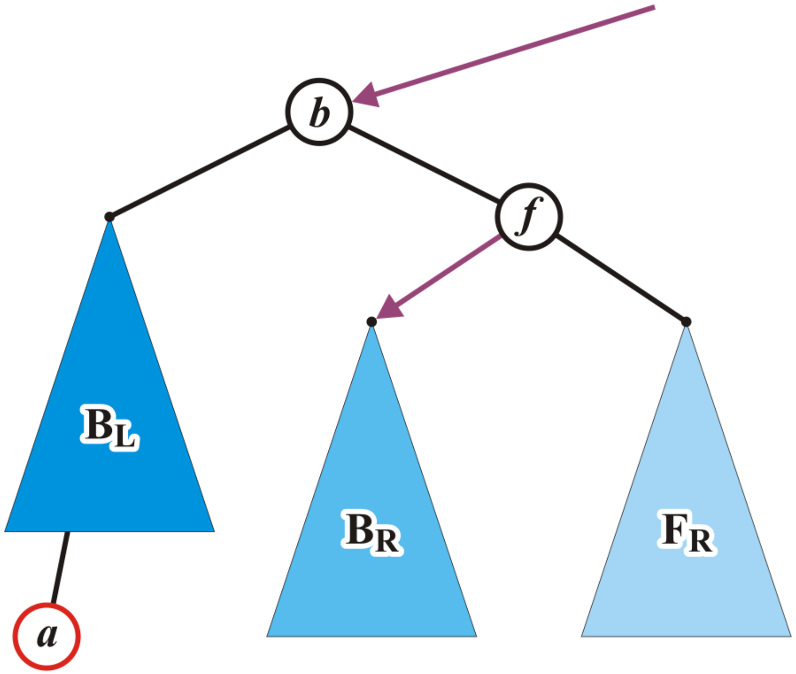
Make ***f*** the right child of ***b***



Maintaining Balance: Case 1

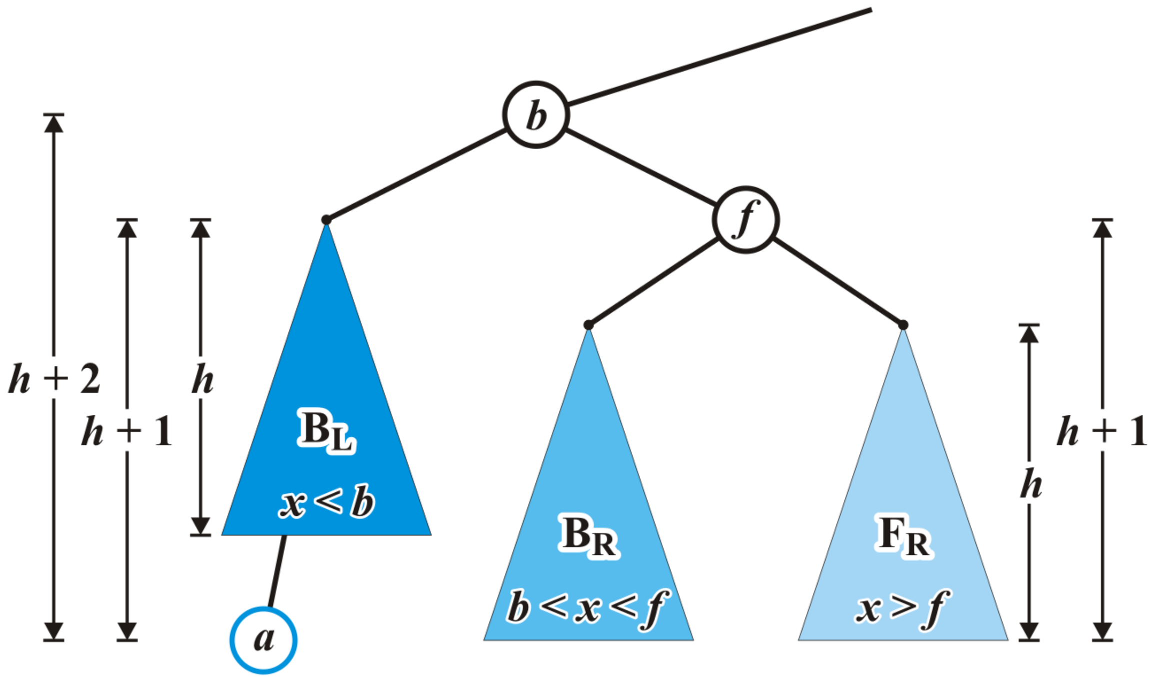
Assign former parent of node ***f*** to point to node ***b***

Make $\mathbf{B_R}$ left child of node ***f***



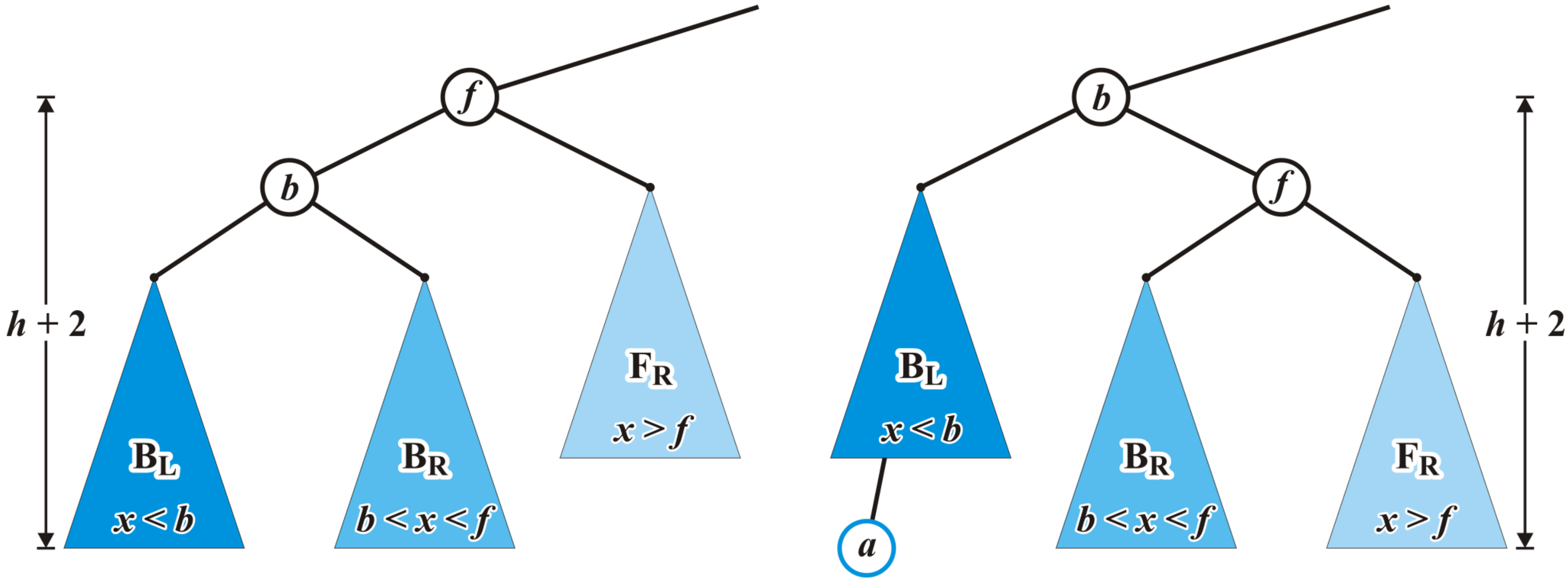
Maintaining Balance: Case 1

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

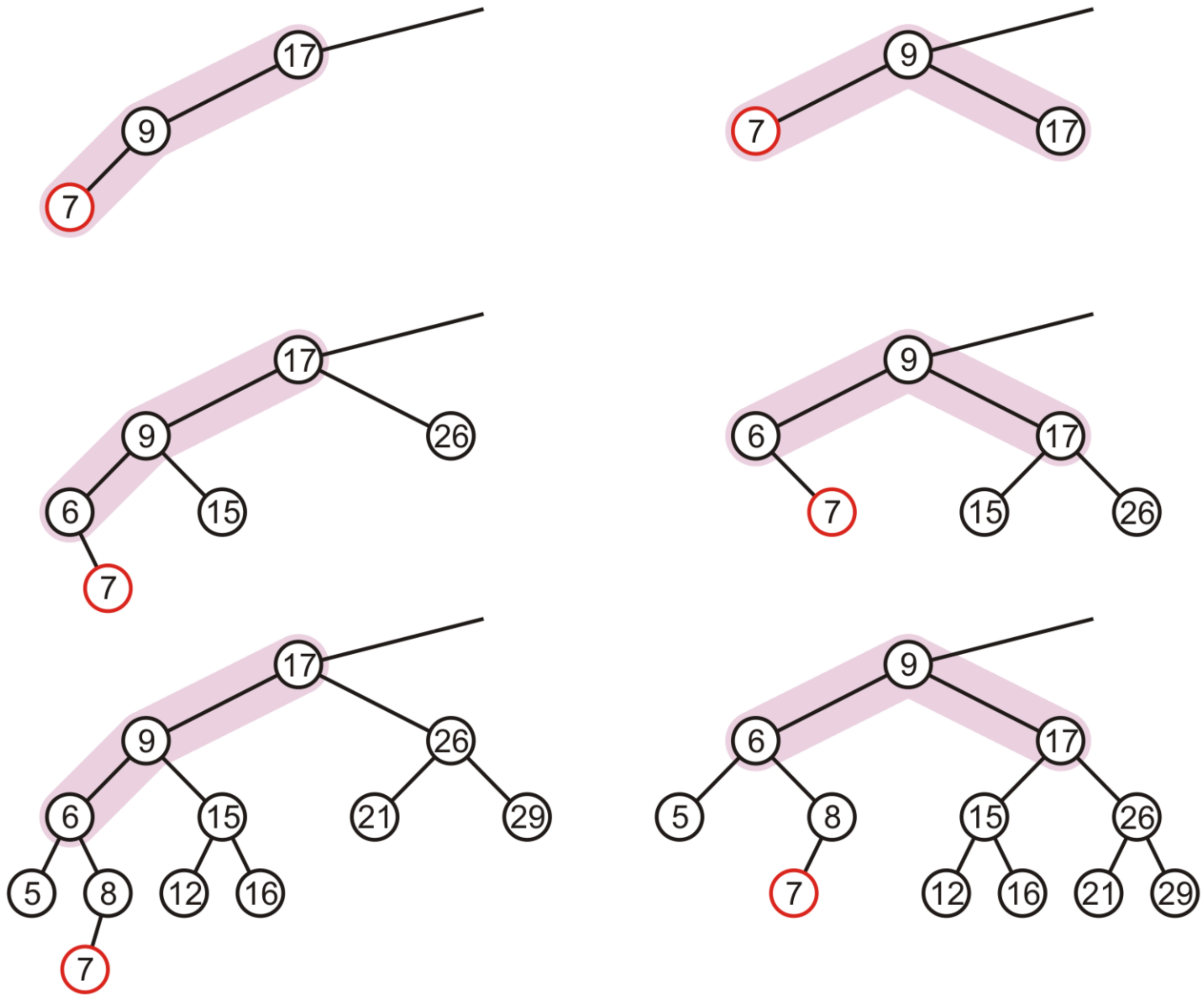


Maintaining Balance: Case 1

- 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

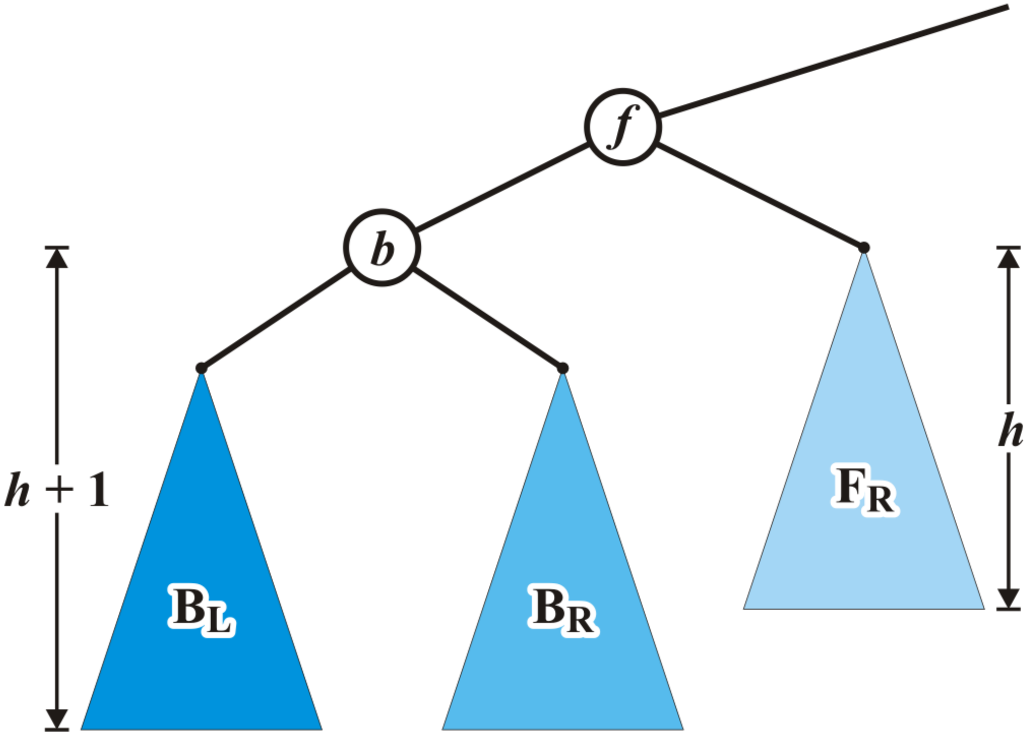


More Examples



Maintaining Balance: Case 2

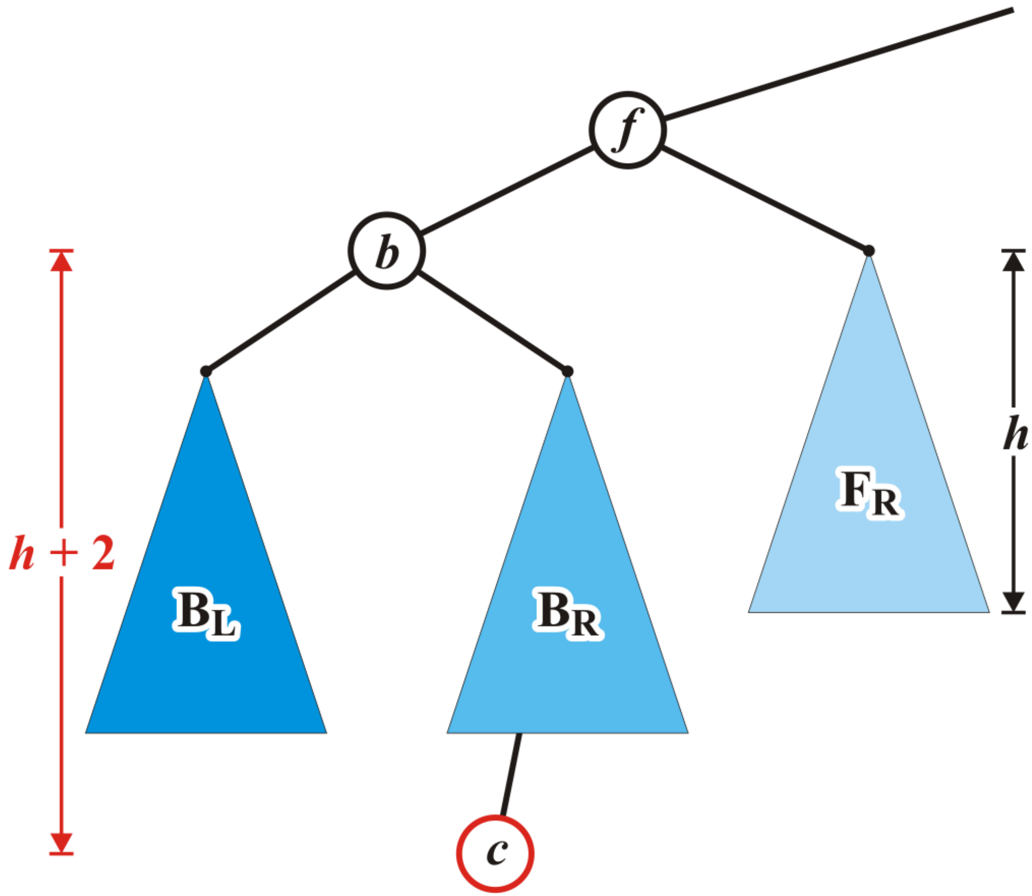
Alternatively, consider the insertion of c where $b < c < f$ into our original tree



Maintaining Balance: Case 2

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

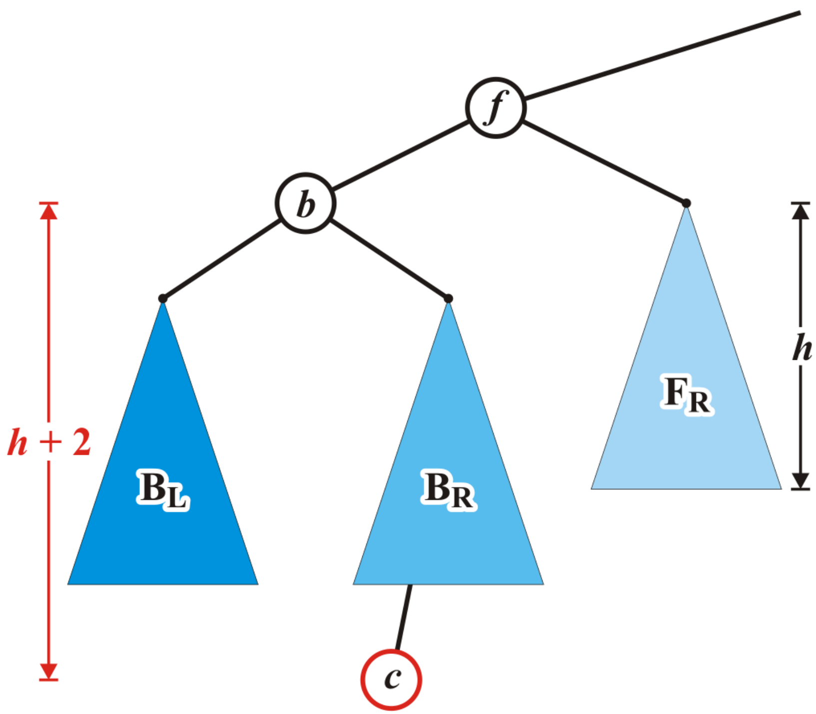
– Once again, f becomes unbalanced

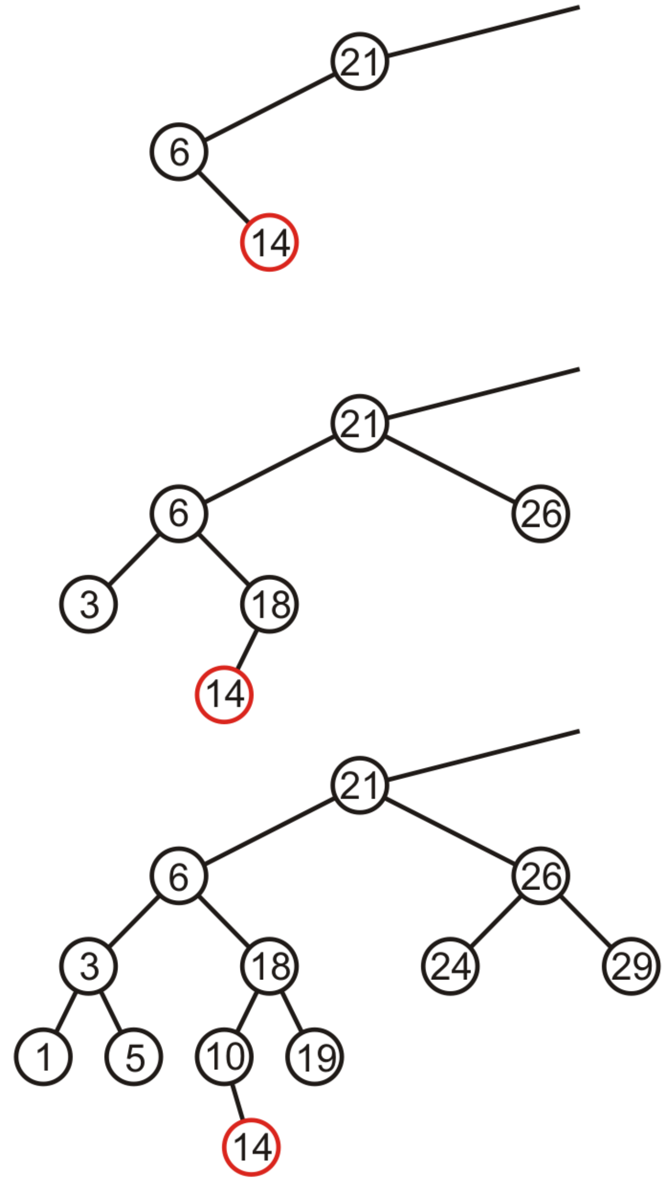


Right subtree of left child

Maintaining Balance: Case 2

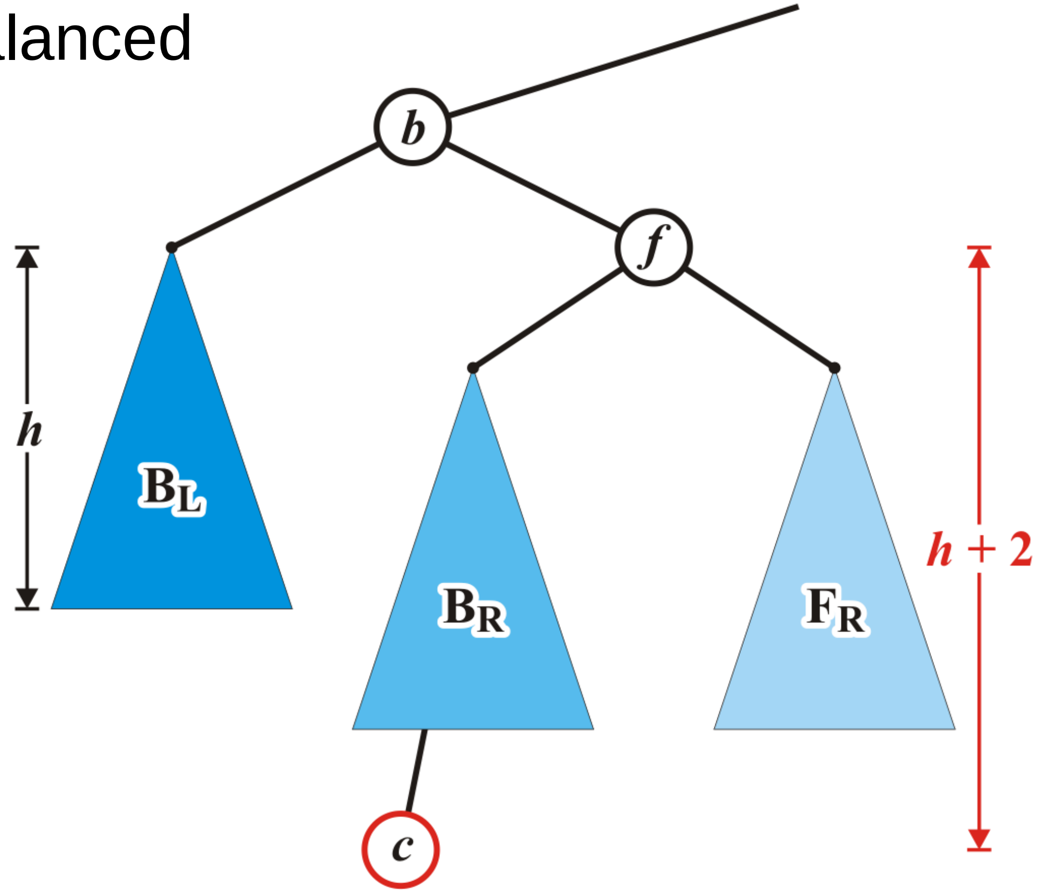
Here are examples of when the insertion of 14 may cause this situation when $h = -1$, 0, and 1





Maintaining Balance: Case 2

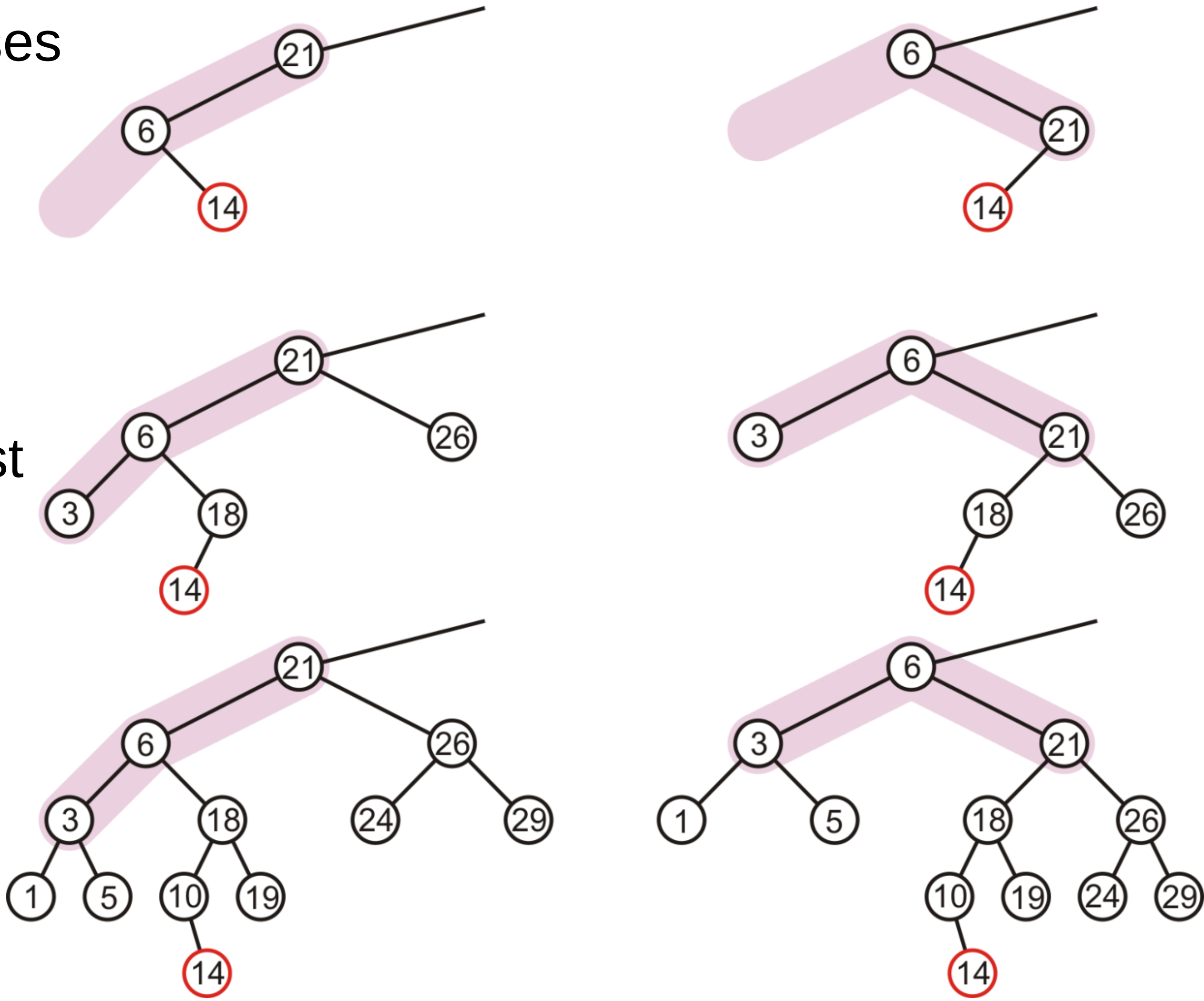
Unfortunately, the previous correction does not fix the imbalance at the root of this sub-tree: the new root, ***b***, remains unbalanced



Maintaining Balance: Case 2

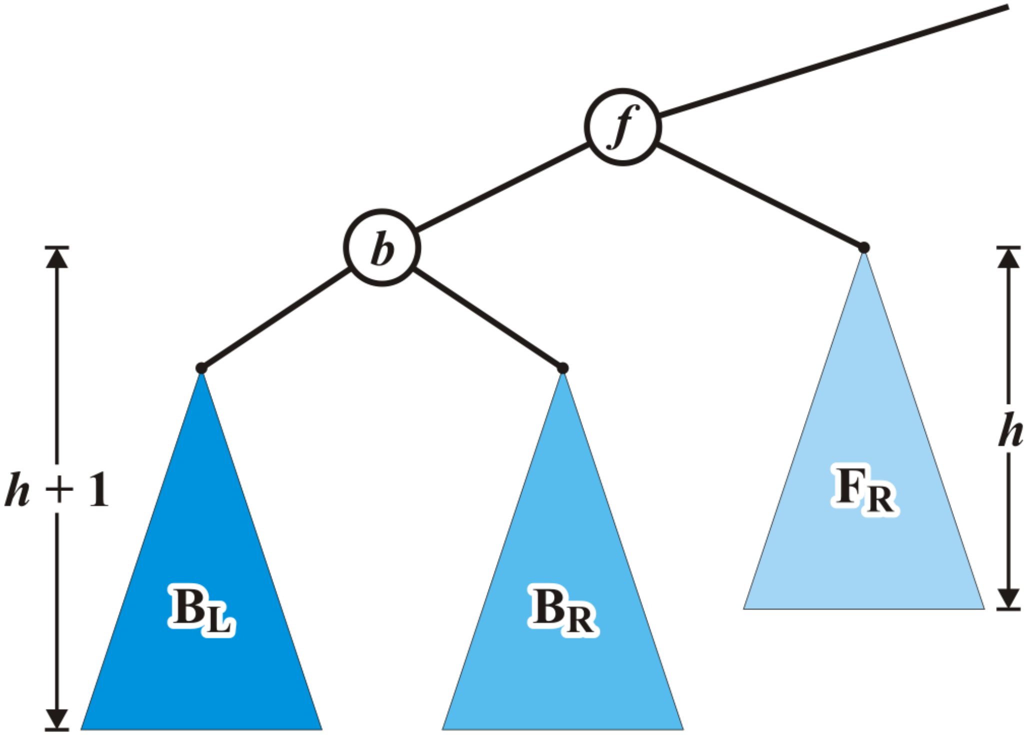
In our three sample cases with $h = -1$, 0, and 1, doing the same thing as before results in a tree that is still unbalanced…

- The imbalance is just shifted to the other side



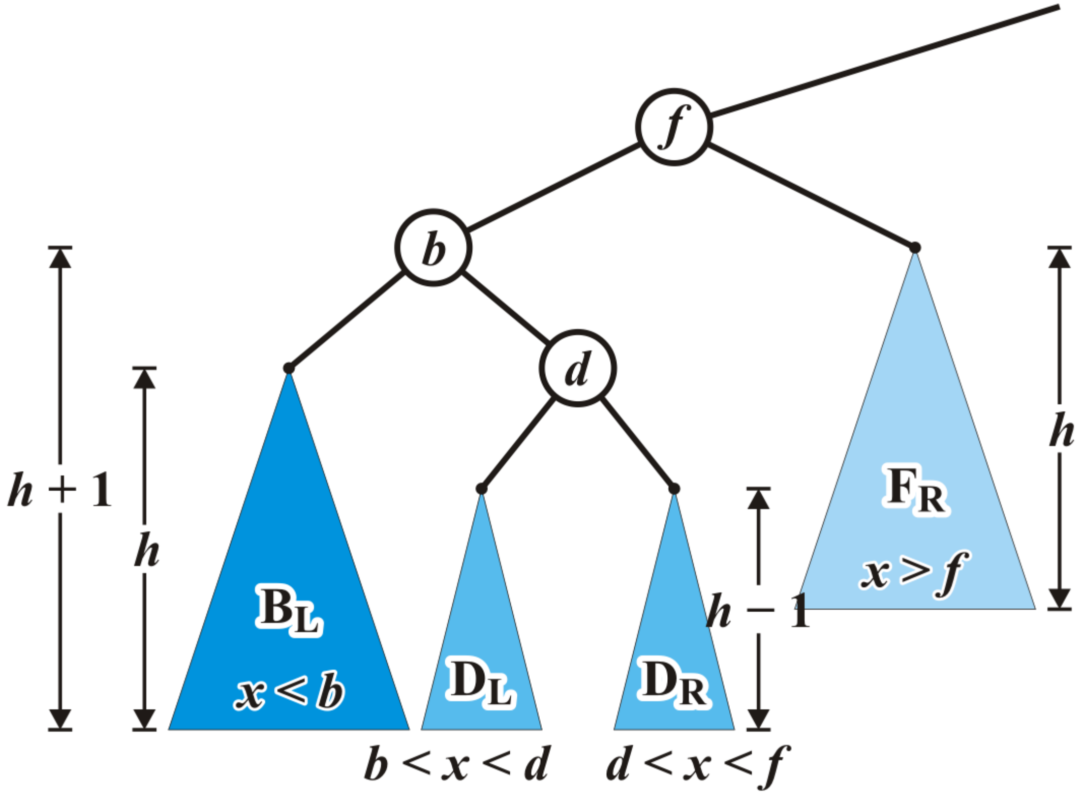
Maintaining Balance: Case 2

Lets start over …



Maintaining Balance: Case 2

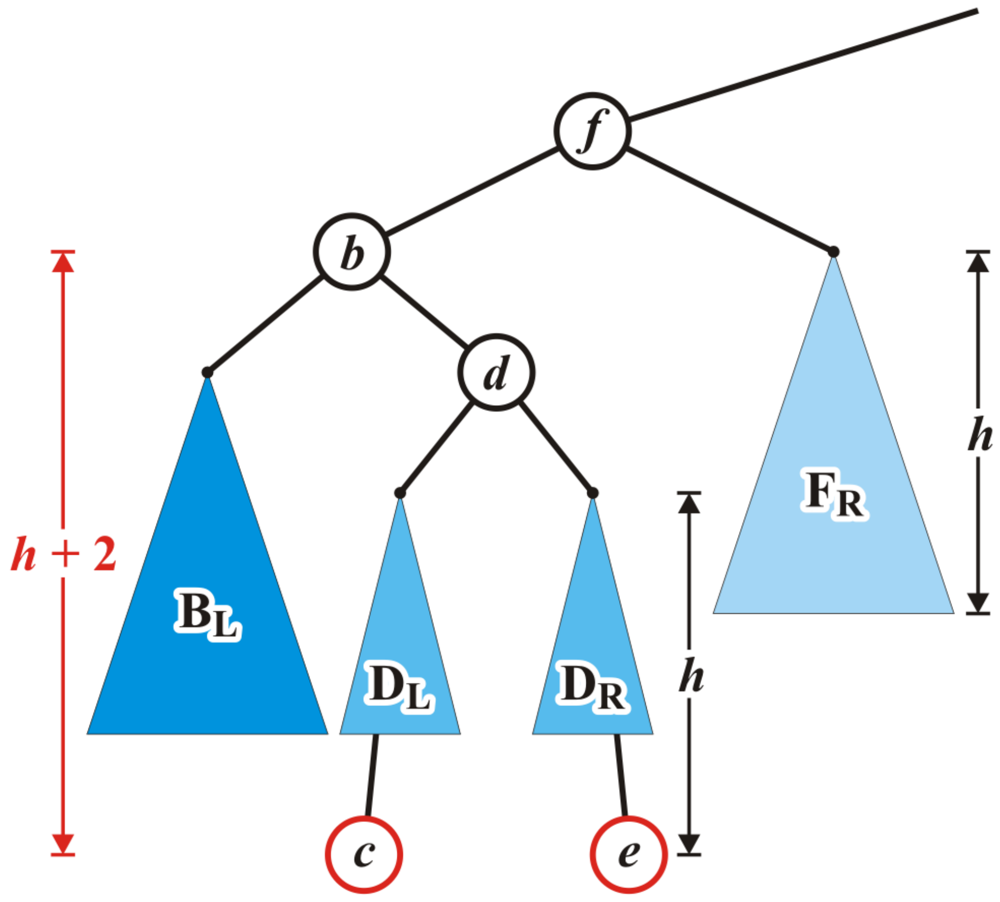
Re-label the tree by dividing the left subtree of f into a tree rooted at d with two subtrees of height $h-1$



Maintaining Balance: Case 2

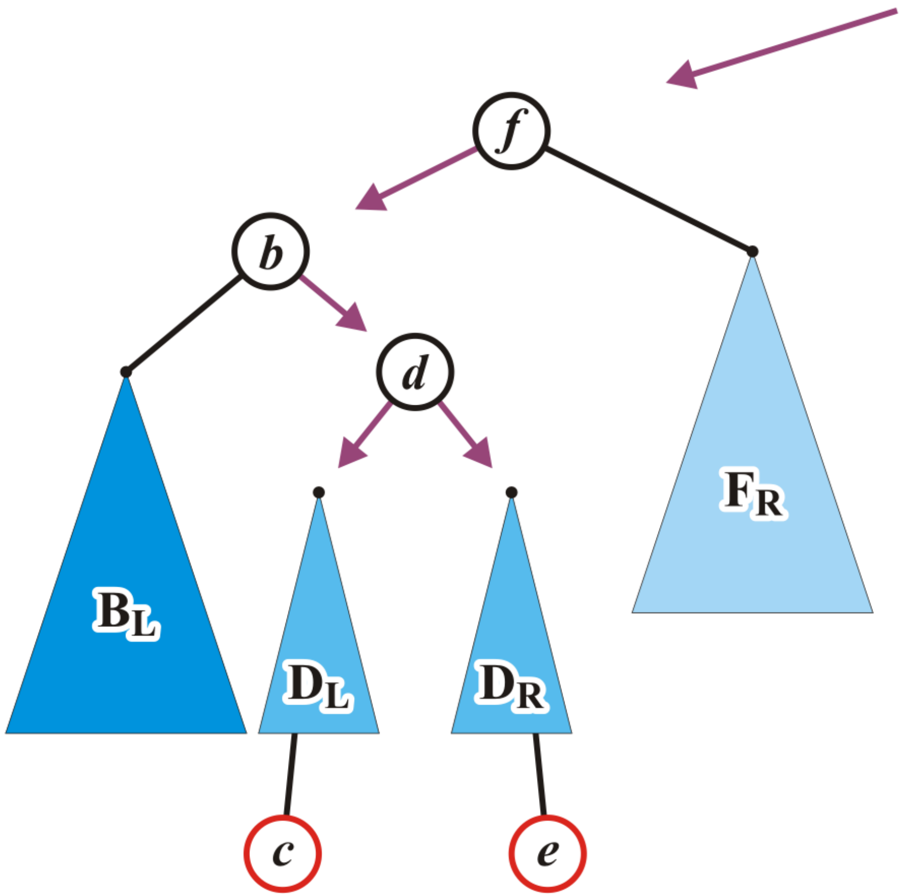
Now an insertion causes an imbalance at ***f***

– The addition of either ***c*** or ***e*** will cause this



Maintaining Balance: Case 2

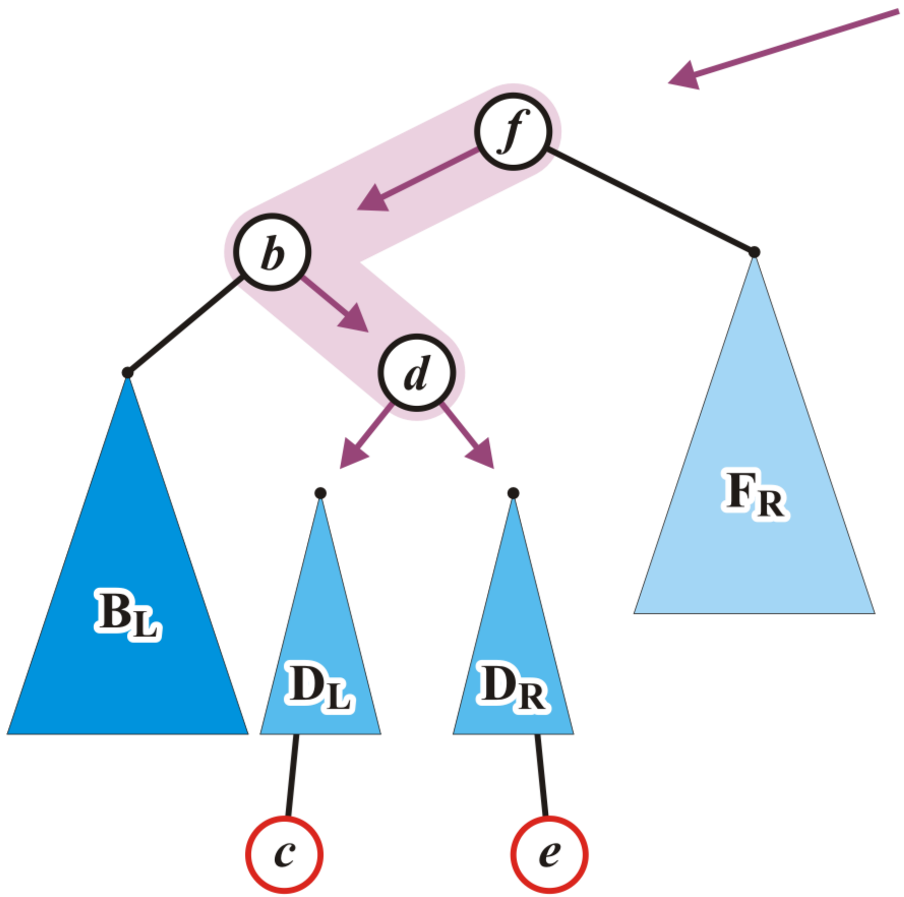
We will reassign the following pointers



Maintaining Balance: Case 2

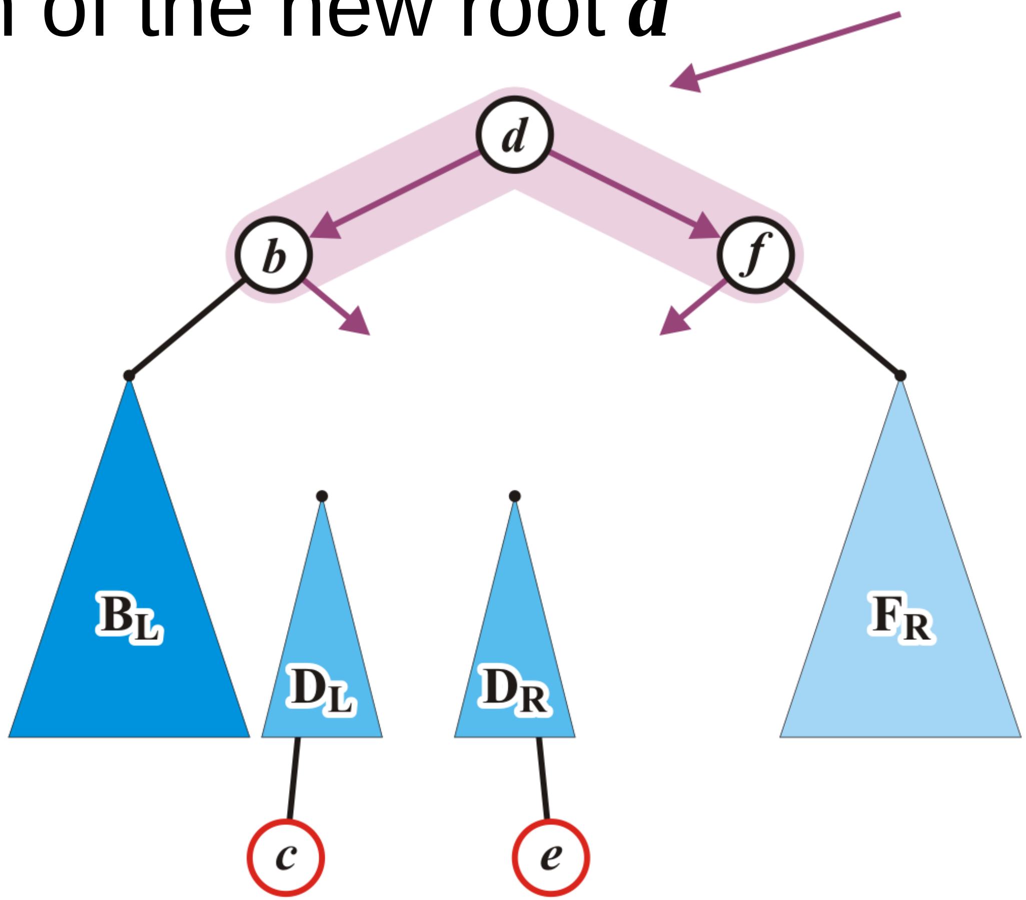
Specifically, we will order these three nodes as a perfect tree

– Recall the second prototypical example



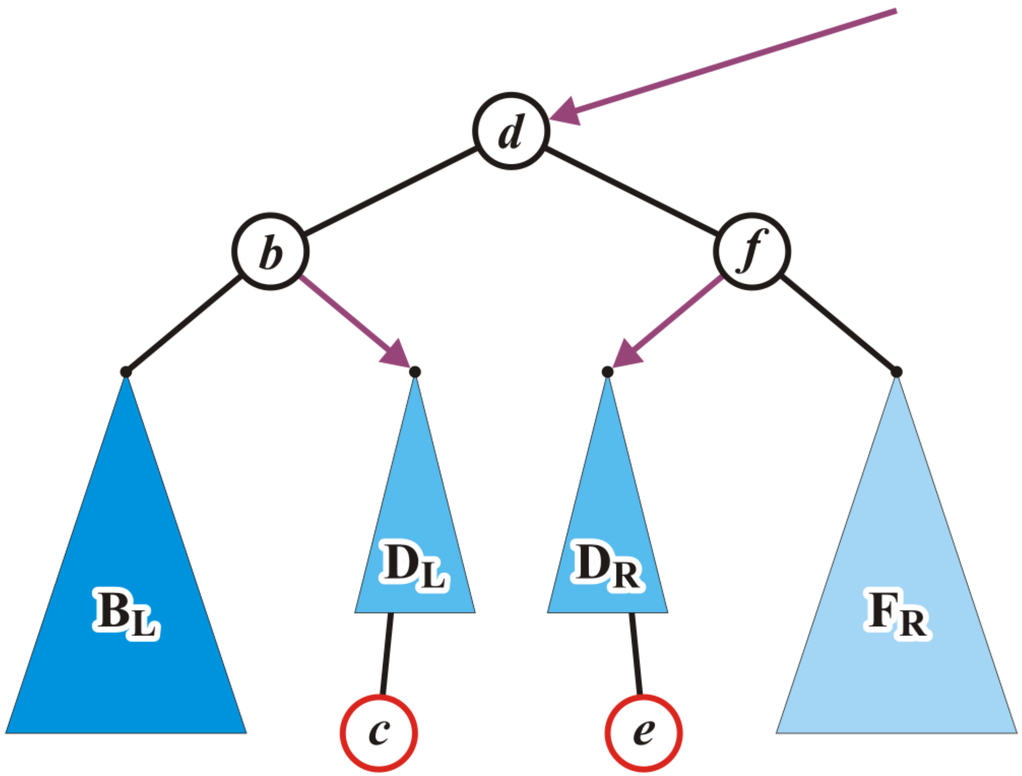
Maintaining Balance: Case 2

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



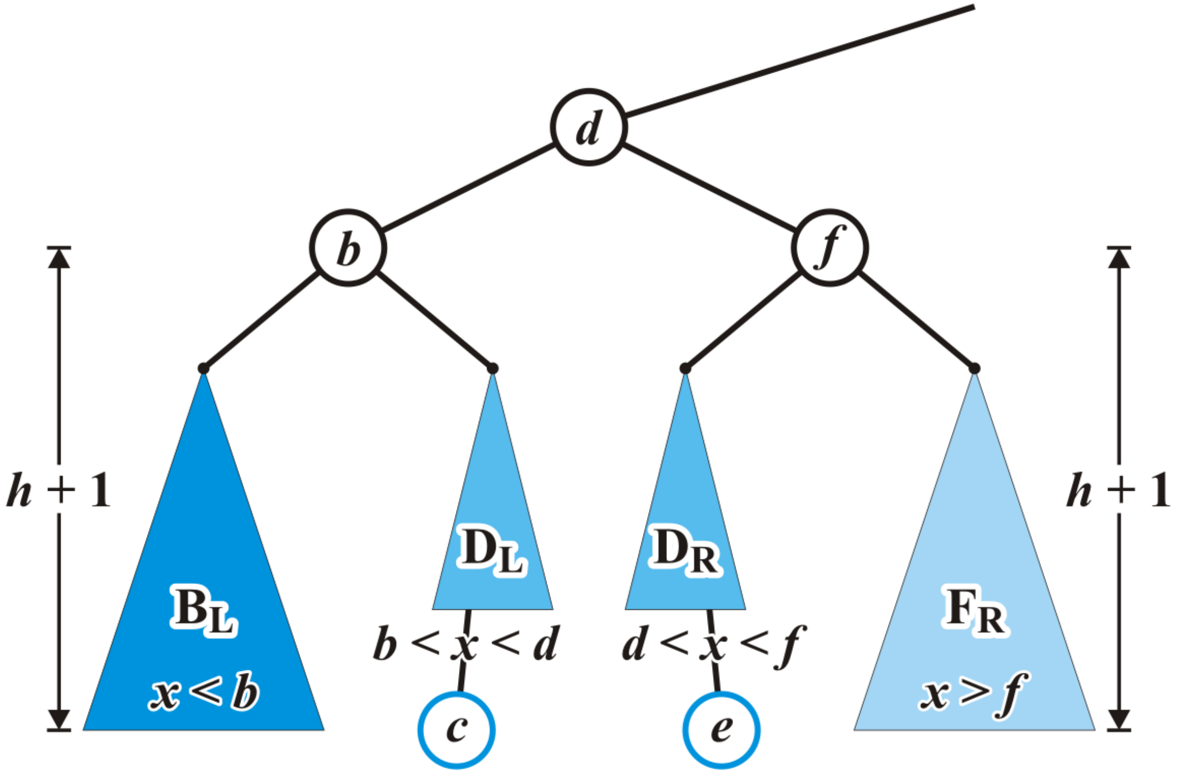
Maintaining Balance: Case 2

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



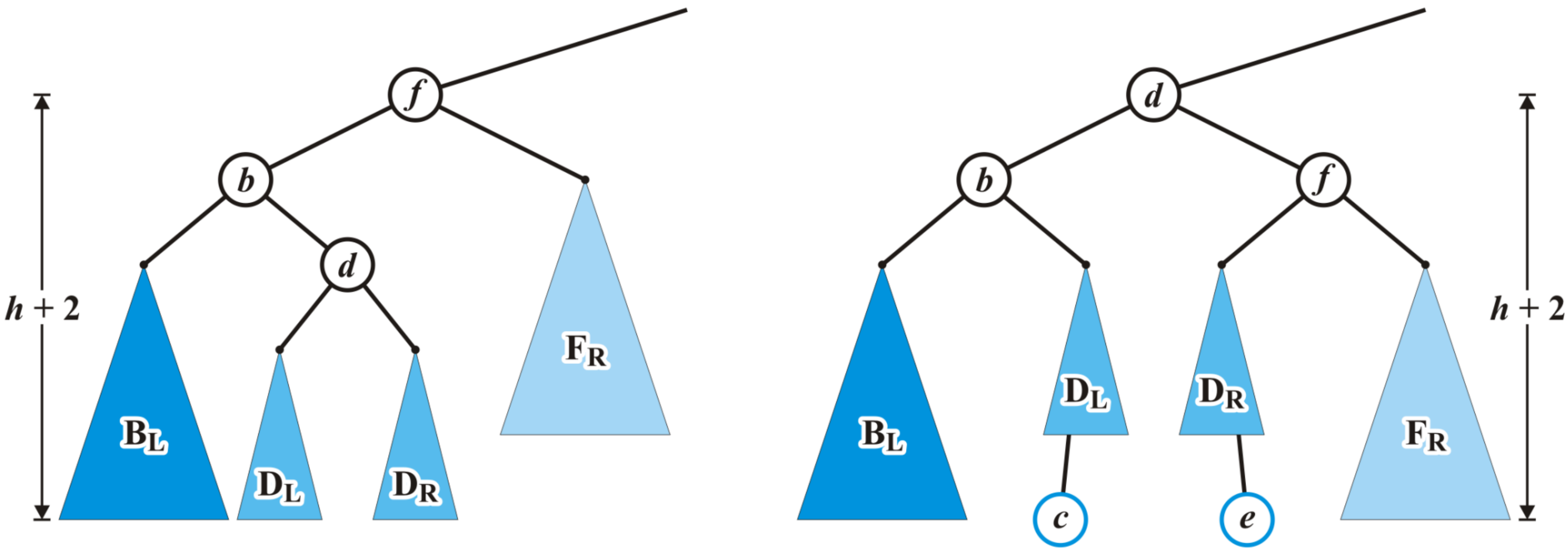
Maintaining Balance: Case 2

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



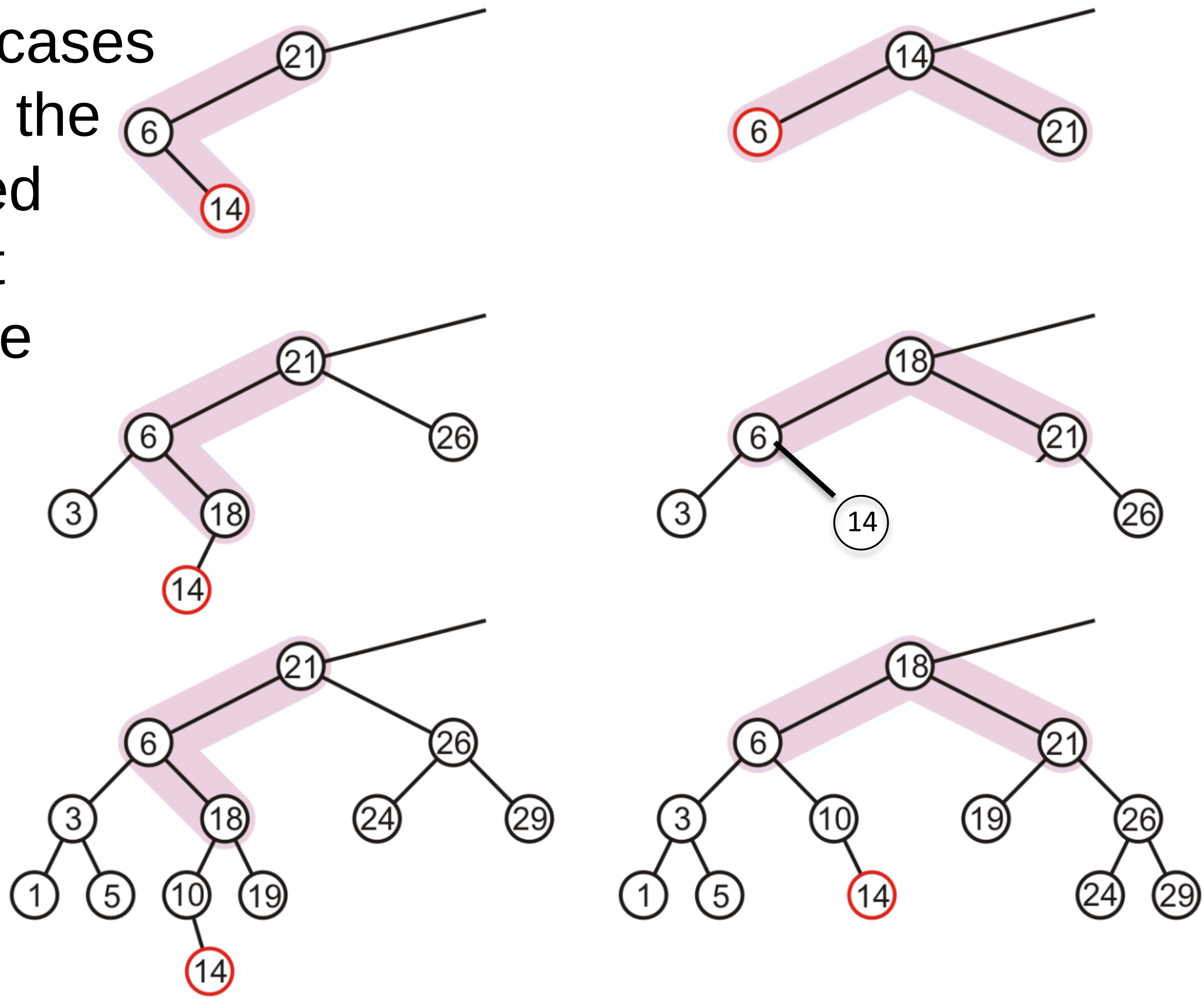
Maintaining Balance: Case 2

Again, the height of the root did not change



Maintaining Balance: Case 2

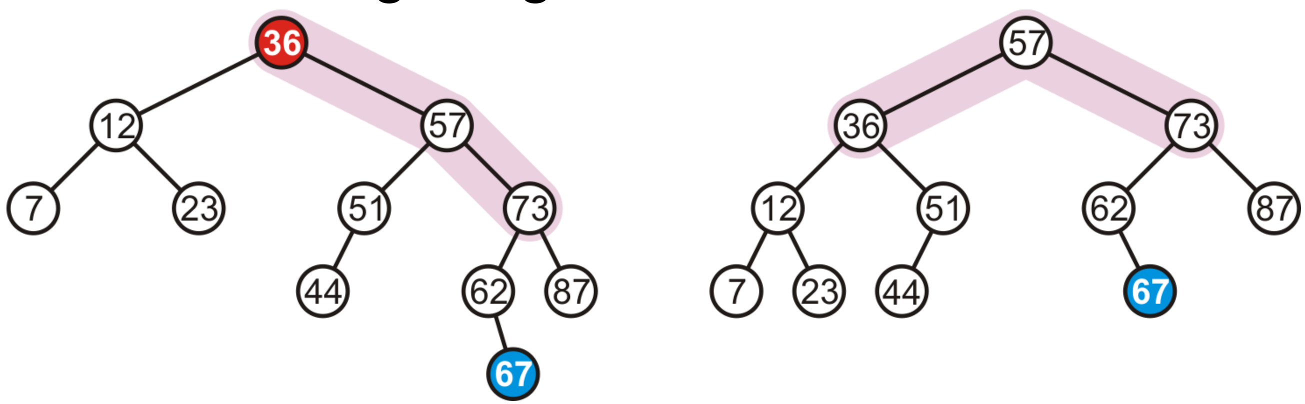
In our three sample cases with $h = -1$, 0, and 1, the node is now balanced and the same height as the tree before the insertion



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