Discussion Week 6

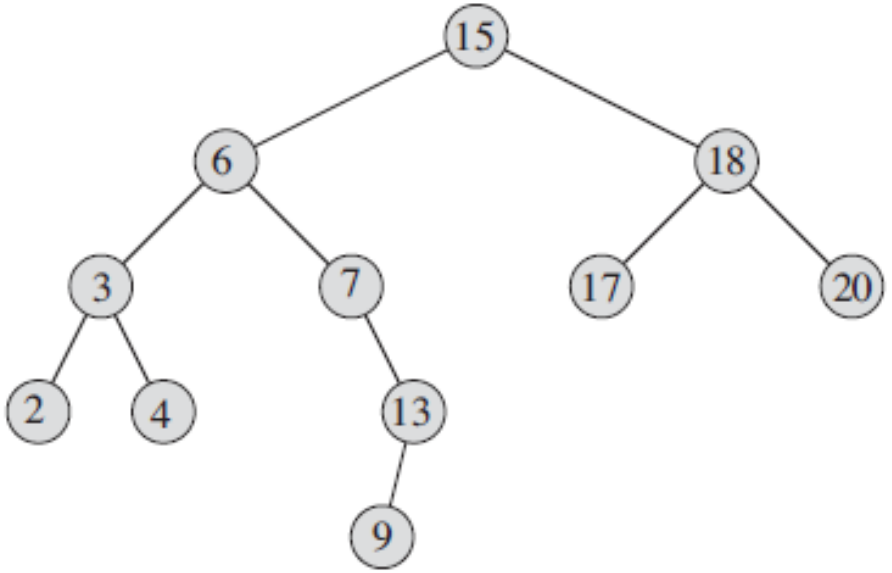
Binary Search Tree

What is the purpose of Binary Search Trees?

- If we implement an Abstract Sorted List using an **array** or a **linked list**, we will have operations which are **O**(*n*).
- Binary Search Trees have operations which are **O**(h), where h is the height of the tree.

Binary Search Trees

- All objects in the left sub-tree to be less than the object stored in the root node.
- All objects in the right sub-tree to be greater than the object in the root object.
- The two sub-trees are themselves binary search trees.

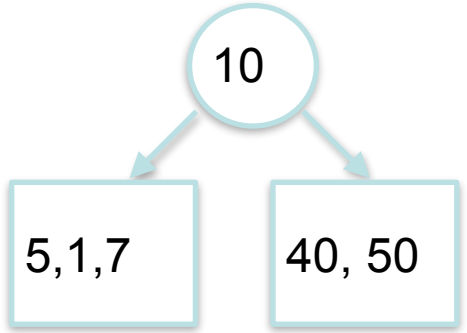


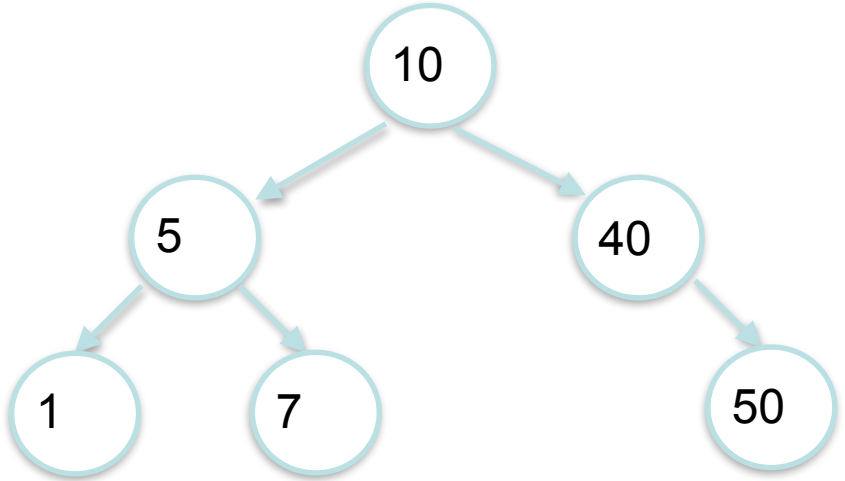
Properties of BST

- Given a pre-order(left to right) sequence of the BST, we can uniquely certain the structure of it.

Question:
The preorder traversal of a binary search tree is {10, 5, 1, 7, 40, 50}, construct the BST.

Method:
- The first element of preorder traversal is always root.
- Let the index of first element which is greater than root is “i”.
- The value between root and “i” belongs to the left subtree.
- The value from “i” to n-1 belongs to the right subtree.

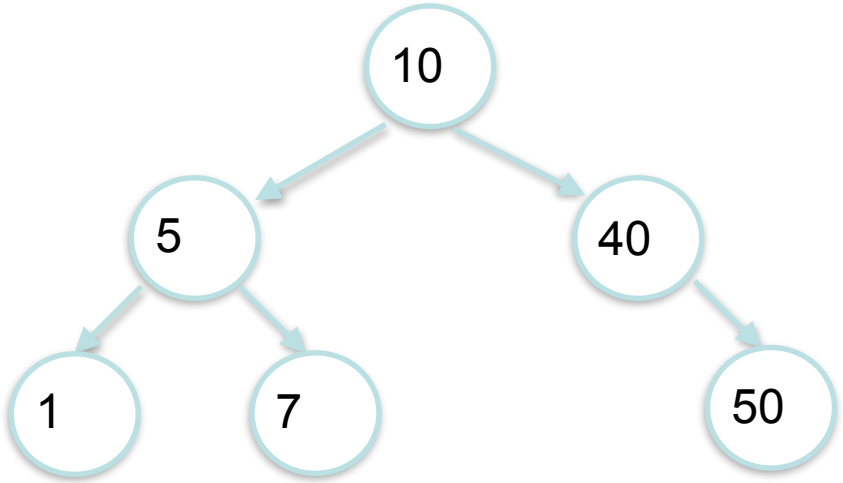




Properties of BST

- The in-order sequence of BST are ascendant.

Example:



In-order sequence: 1, 5, 7, 10, 40, 50

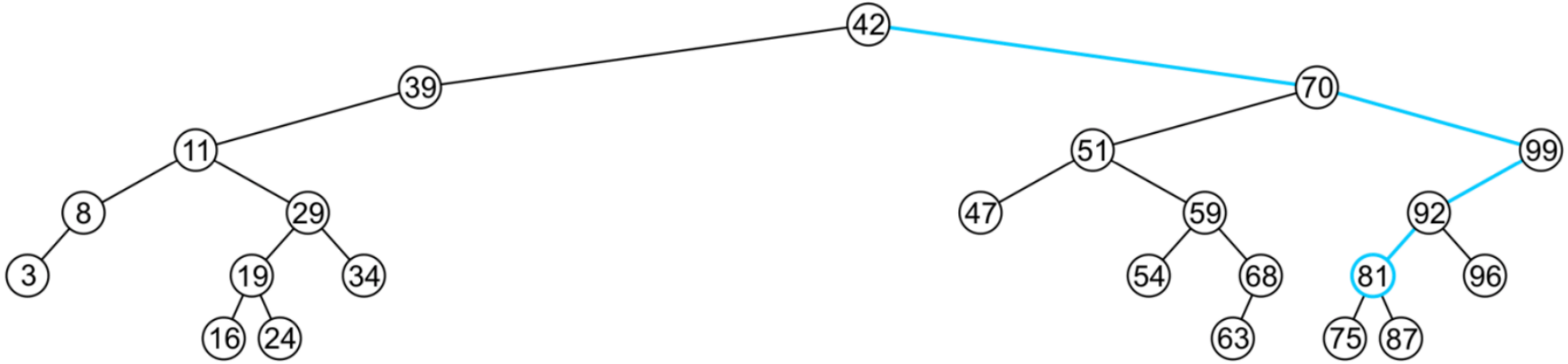
Question:
(T or F) For any two BST with same keys, they always have the same in-order sequence.

Answer: T

Operations in BST

- Search
- Insert
- Erase
- Next & Previous

Search in a BST



Recursive:

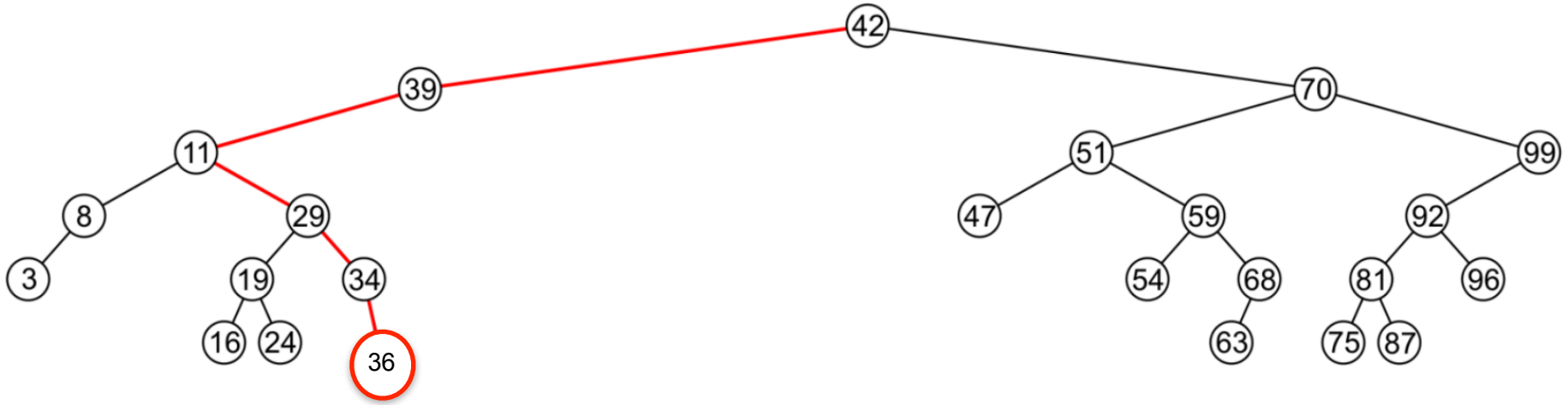
```
TREE-SEARCH(x, k)
1  if x == NIL or k == x.key
2        return x
3  if k < x.key
4        return TREE-SEARCH(x.left, k)
5  else return TREE-SEARCH(x.right, k)
```

Iterative:

```
ITERATIVE-TREE-SEARCH(x, k)
1  while x ≠ NIL and k ≠ x.key
2        if k < x.key
3              x = x.left
4        else x = x.right
5  return x
```

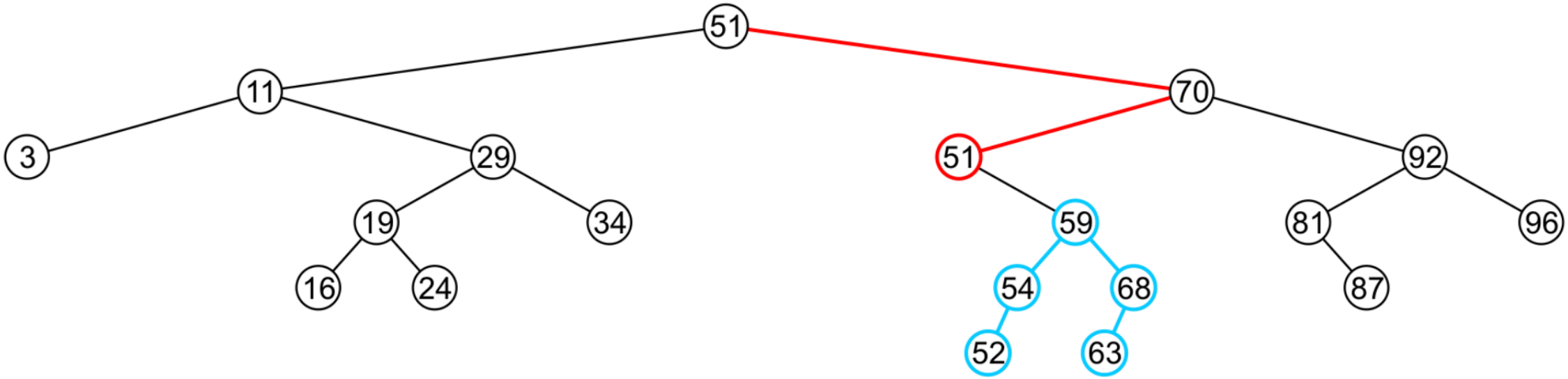
Insertion

- Search for “L” ends at a null link.
- Create new node.
- Reset links (and increment counts on the way up).



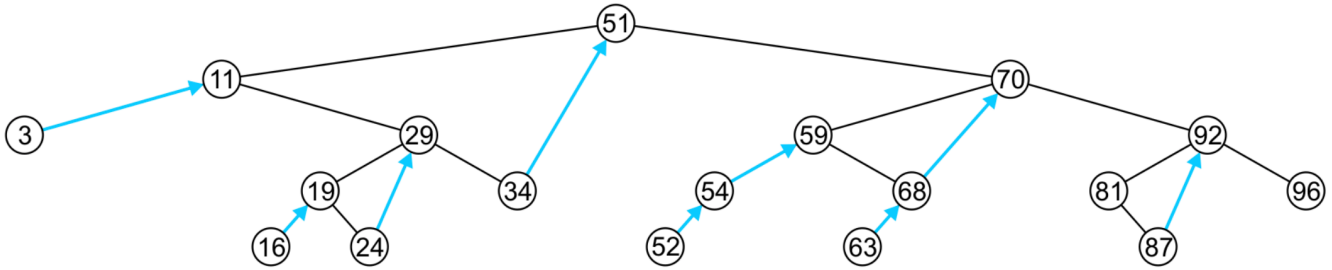
Erase

- For 0 children node, erase it directly.
- For 1 children node, promote the sub-tree associated with the child
- For 2 children node:
  - Replace it with the minimum object in the right sub-tree.
  - Erase that object from the right sub-tree.



Next

- If the node has the right sub-tree, find the min in the right sub-tree.
- If it does not have the right sub-tree, find the first larger object (if any) that exists in the path from the node to the root

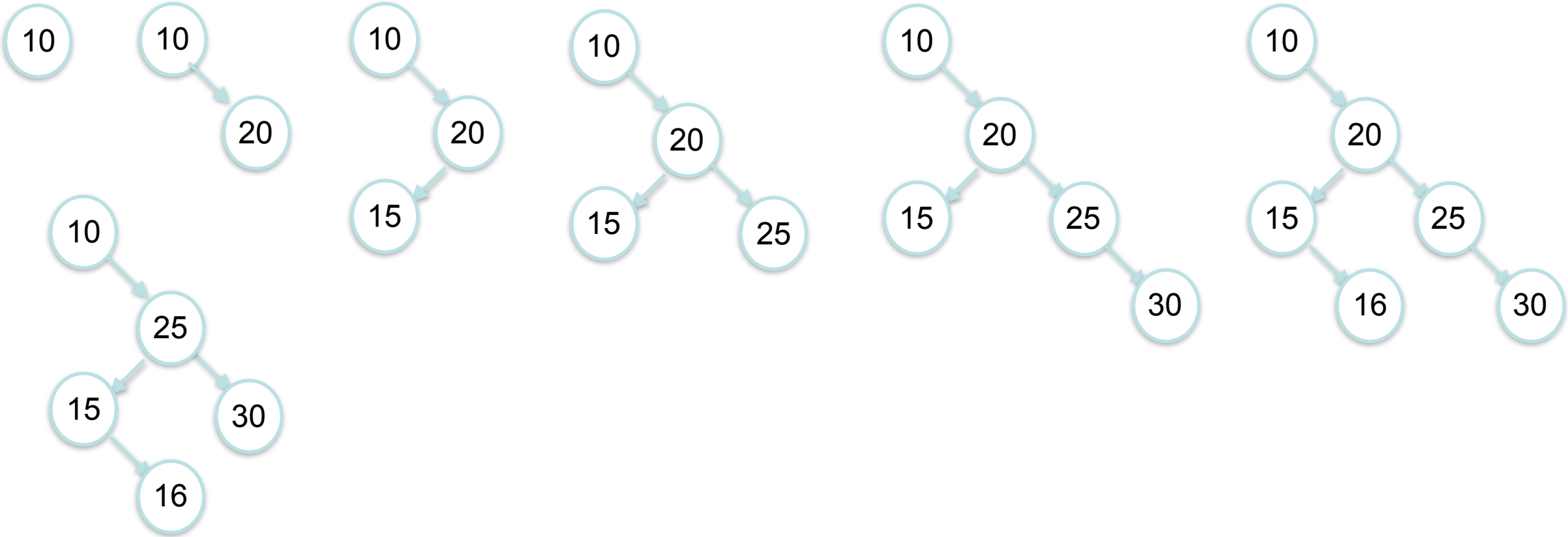



```
TREE-SUCCESSOR(x)
1  if x.right ≠ NIL
2        return TREE-MINIMUM(x.right)
3  y = x.p
4  while y ≠ NIL and x == y.right
5        x = y
6        y = y.p
7  return y
```

The "**previous**" operation is similar.

Question:

- Staring from an empty binary search tree, insertion of the following integers in sequence {10, 20, 15, 25, 30, 16}. What does the BST look like?
- Deleting 20, what does the BST look like?



AVL Trees

What is the purpose of AVL Trees?

- AVL trees are binary search trees that are guaranteed to be balanced. They guarantee $h = O(\log n)$ on the tree.

AVL Trees' Definition

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

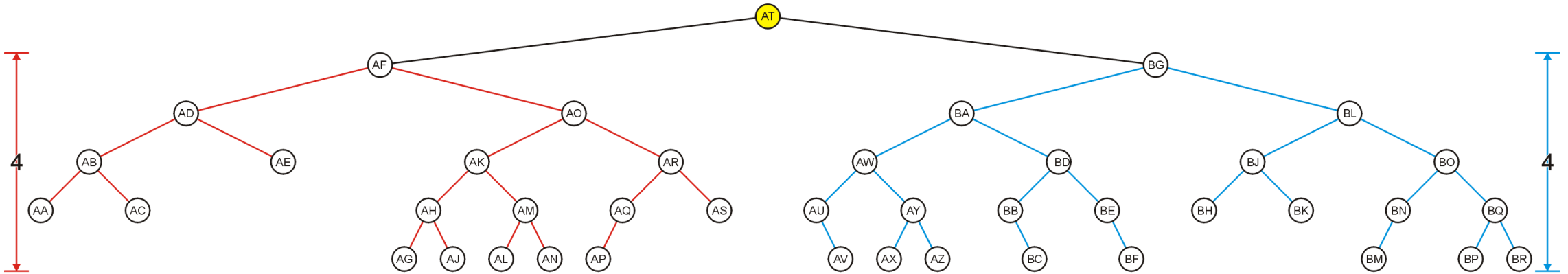
Recall:

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

AVL Trees

The root node is AVL-balanced:

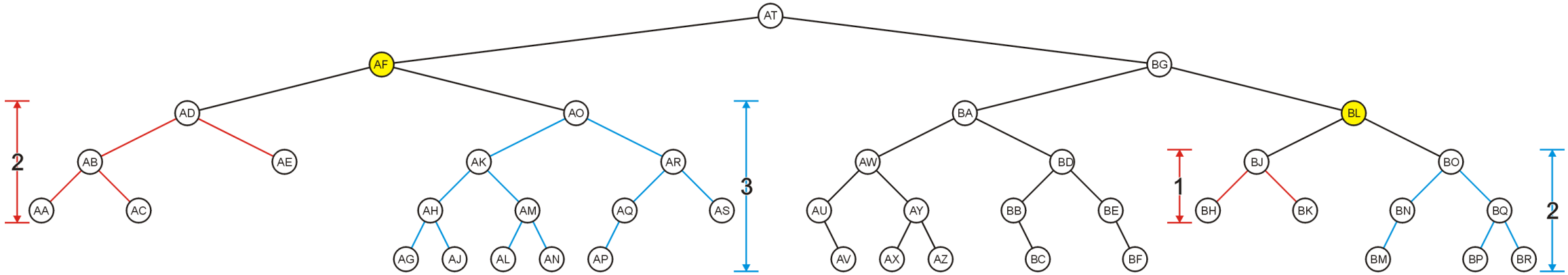
- Both sub-trees are of height 4:



AVL Trees

All other nodes (*e.g.*, AF and BL) are AVL balanced

- The sub-trees differ in height by at most one



AVL tree’s height

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

What is the lower bound?

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:

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

The solution?

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

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

Height of an AVL Tree

This is approximately:

$$F(h) \approx 1.8944\varphi^h - 1$$

Where $\varphi \approx 1.6180$ is the golden ratio

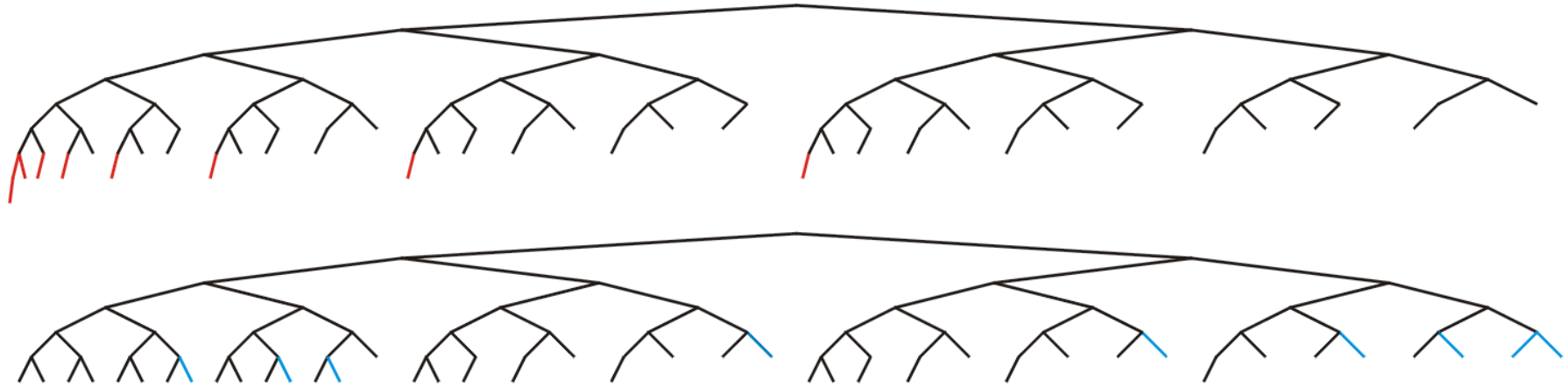
- Thant is, $F(h) = \Theta(\varphi^h)$

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

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

Height of an AVL Tree

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



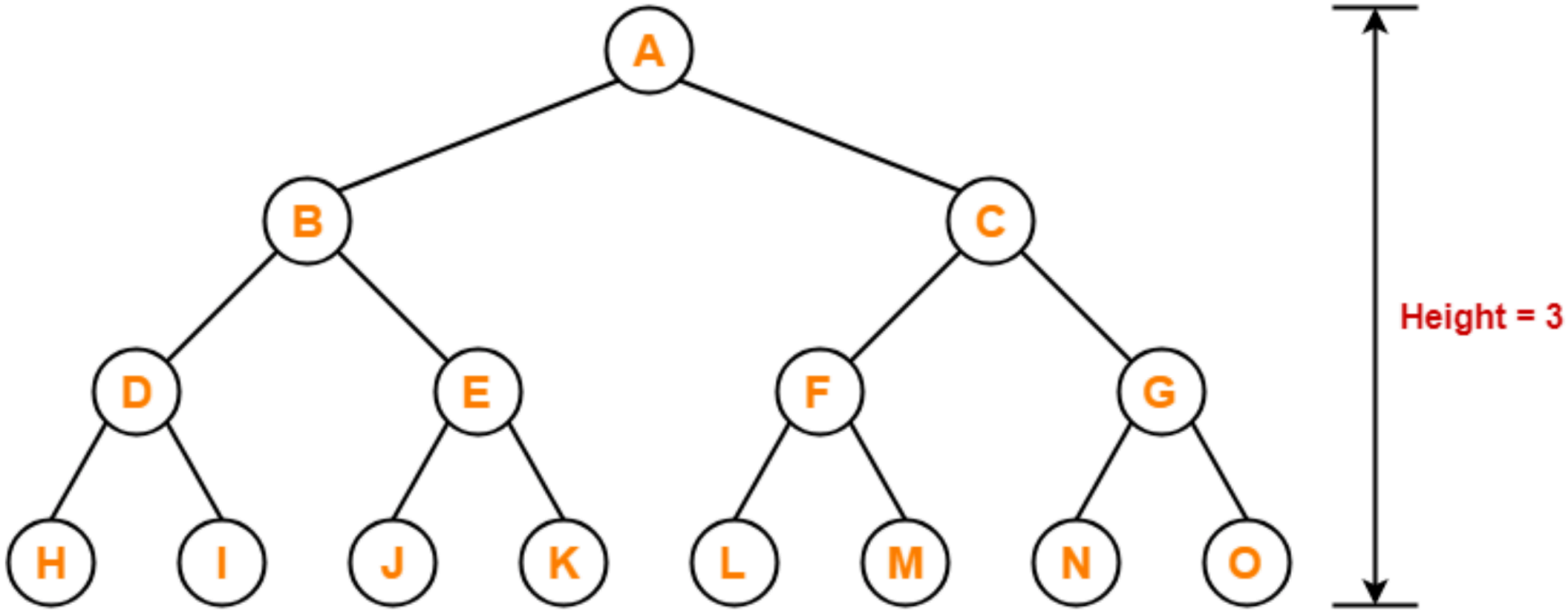
AVL Tree Properties

- Maximum possible number of nodes in AVL tree of height h is $2^{h+1} - 1$

Example

Maximum possible number of nodes in AVL tree of height-3

$= 2^{3+1} - 1$
$= 16 - 1$
$= 15$



AVL Tree Properties

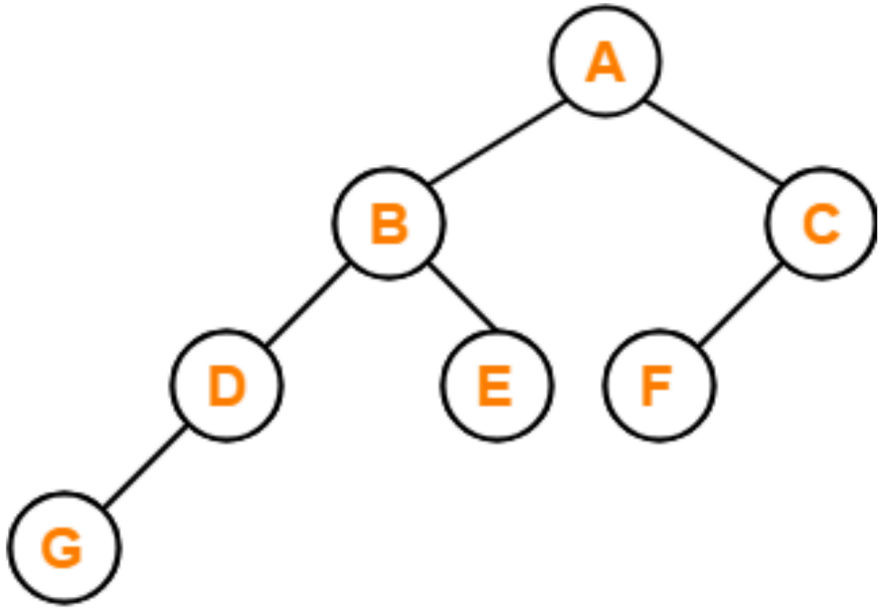
- Minimum number of nodes in AVL Tree of height h is given by a recursive relation

$$N(h) = N(h-1) + N(h-2) + 1$$

Example

Minimum possible number of nodes in AVL tree of height-3

= N(2)+N(1)+1
= (N(1) + N(0) + 1) + N(1) + 1
= 2 + 1 + 1 + 2 + 1
= 7



AVL Tree Properties

- Minimum possible height of AVL Tree using n nodes is $\lfloor log_2 n \rfloor$

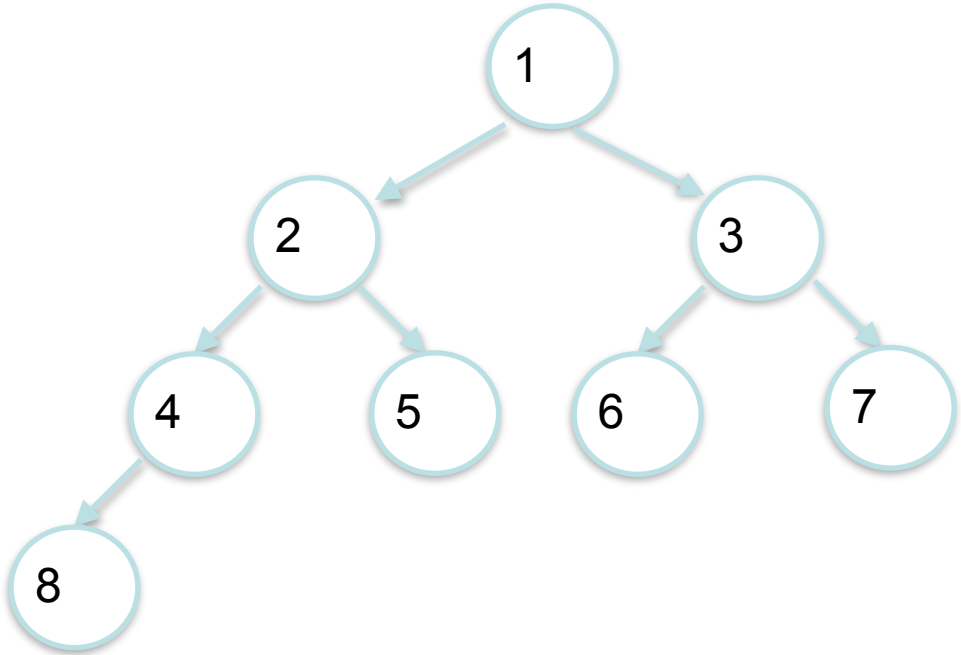
Example

Minimum possible height of AVL Tree using 8 nodes

$= \lfloor log_2 2^3 \rfloor$

$= \lfloor log_2 8 \rfloor$

$= 3$



AVL Tree Properties

- Maximum height of AVL Tree using n nodes is calculated using recursive relation

$$N(h) = N(h-1) + N(h-2) + 1$$

Recall:

$$N(h) = log_{\phi}(n+1) - 1.3277 = 1.4404 \cdot ln(n+1) - 1.3277$$

Question

- What is the maximum height of any AVL tree with 15 nodes?

Answer:

We use the recursive relation: $H(h) = H(h-1) + H(h-2) + 1$
Base case: H(0) = 1, H(1) = 2
By recursive we know,
H(2) = H(1) + H(0) + 1 = 4
H(3) = H(2) + H(1) + 1 = 7
H(4) = H(3) + H(2) + 1 = 12
H(5) = H(4) + H(3) + 1 = 20
Because H(4)< 15 < H(5), so the maximum height is 4.

Maintain balance

Maintain Balance

Observe that:

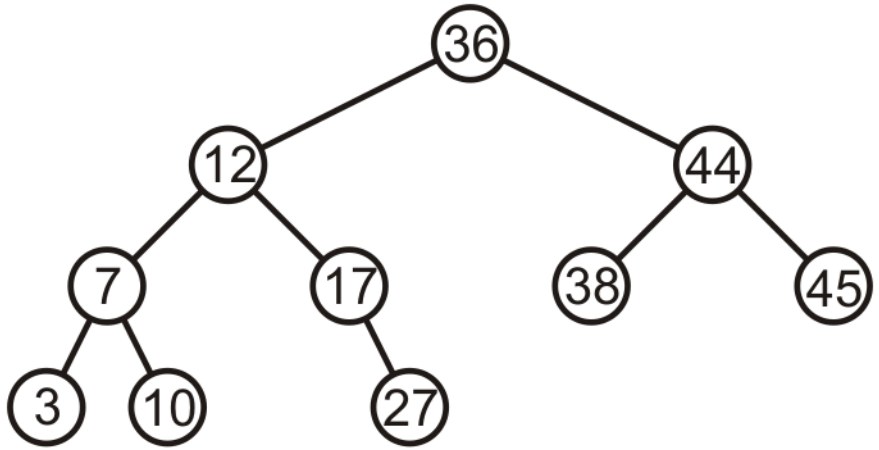
- Inserting a node can increase the height of a tree by at most 1
- Removing a node can decrease the height of a tree by at most 1

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

Maintain Balance

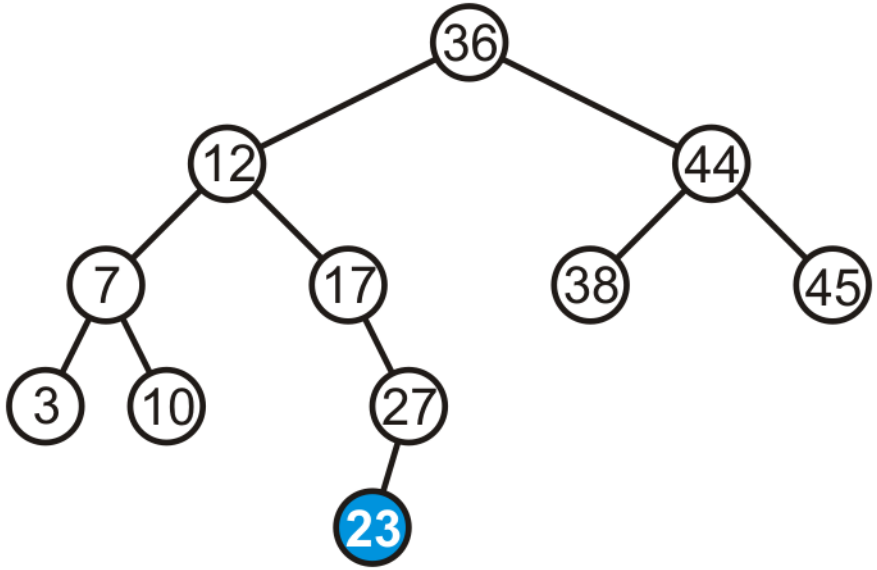
If a tree is AVL balanced, for an insertion to cause an imbalance:

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



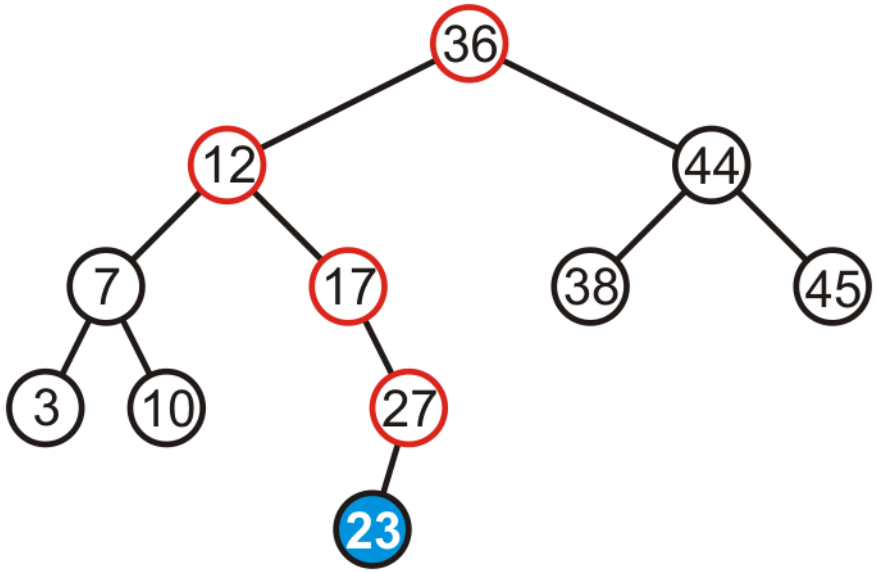
Maintain Balance

Suppose we insert 23 into our initial tree



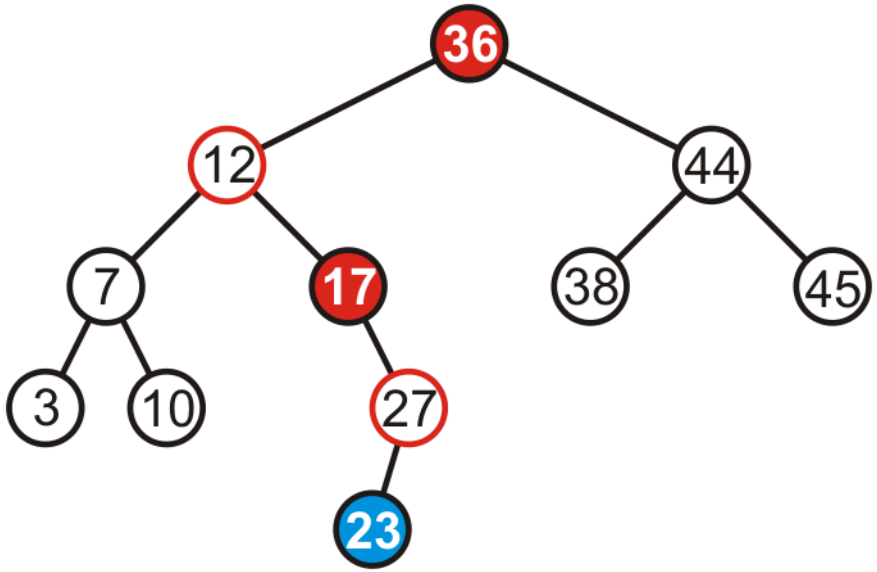
Maintain Balance

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



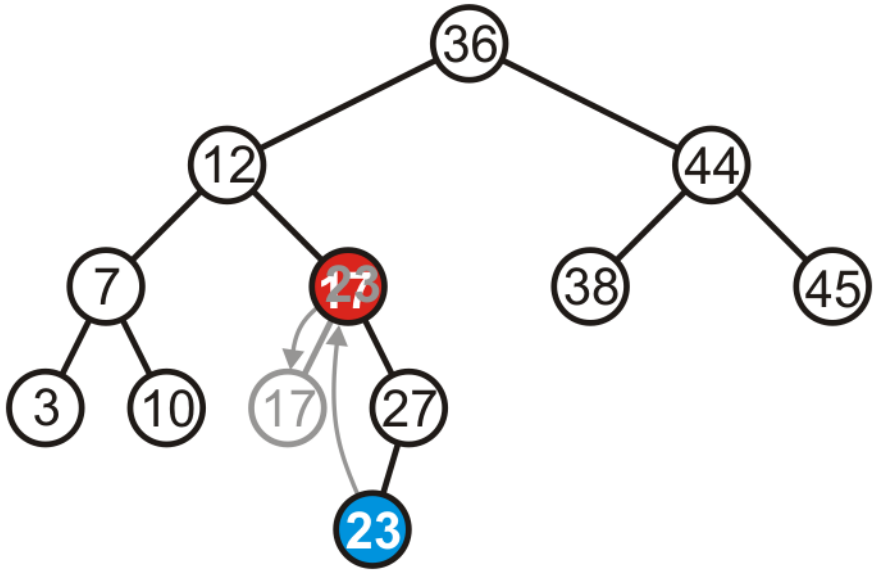
Maintain Balance

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



Maintain Balance

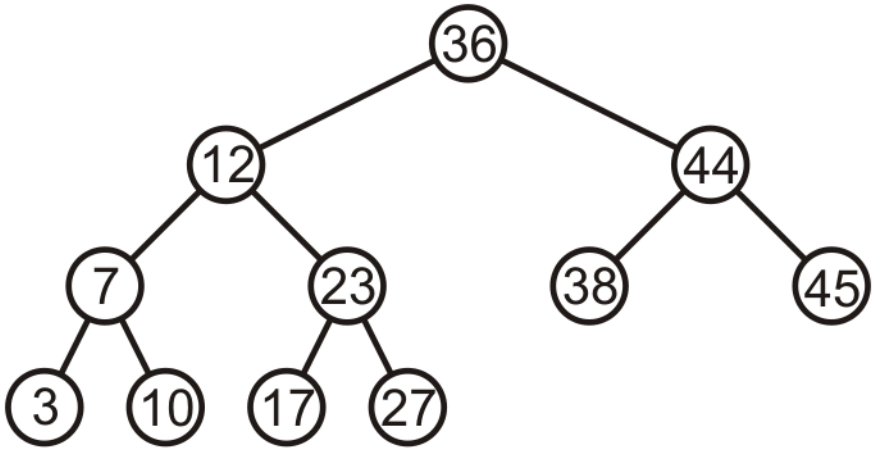
We can promote 23 to where 17 is, and make 17 the left child of 23



Maintain Balance

Thus, that node is no longer unbalanced

- Incidentally, the root is now also balanced!

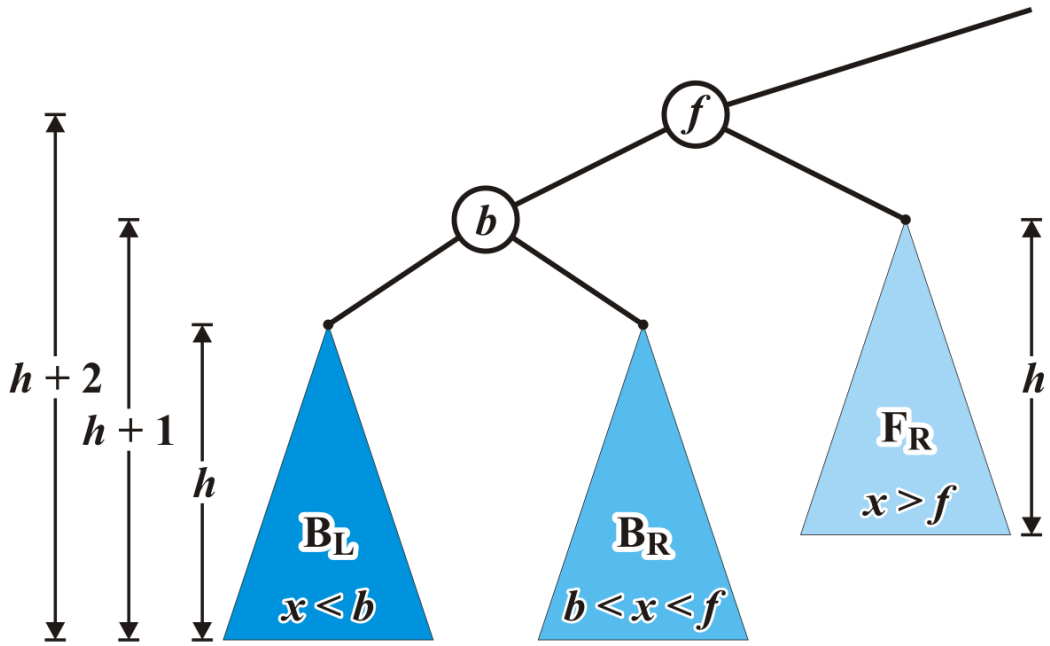


LL/RR Rotation

Maintain Balance: Case 1

Consider the following setup

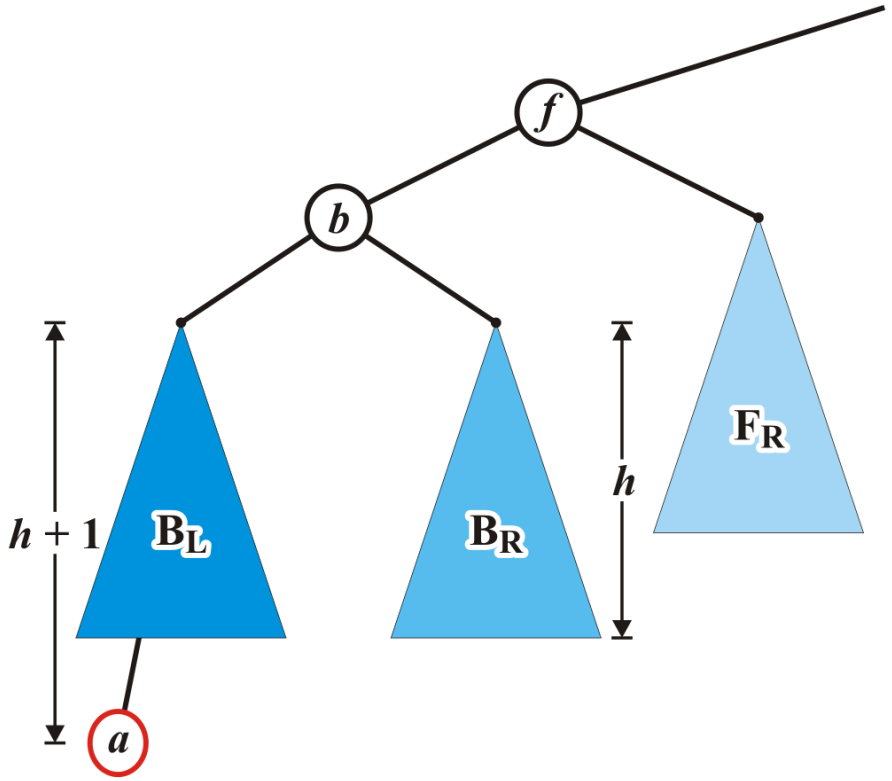
- Each blue triangle represents a tree of height *h*



Maintain Balance: Case 1

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

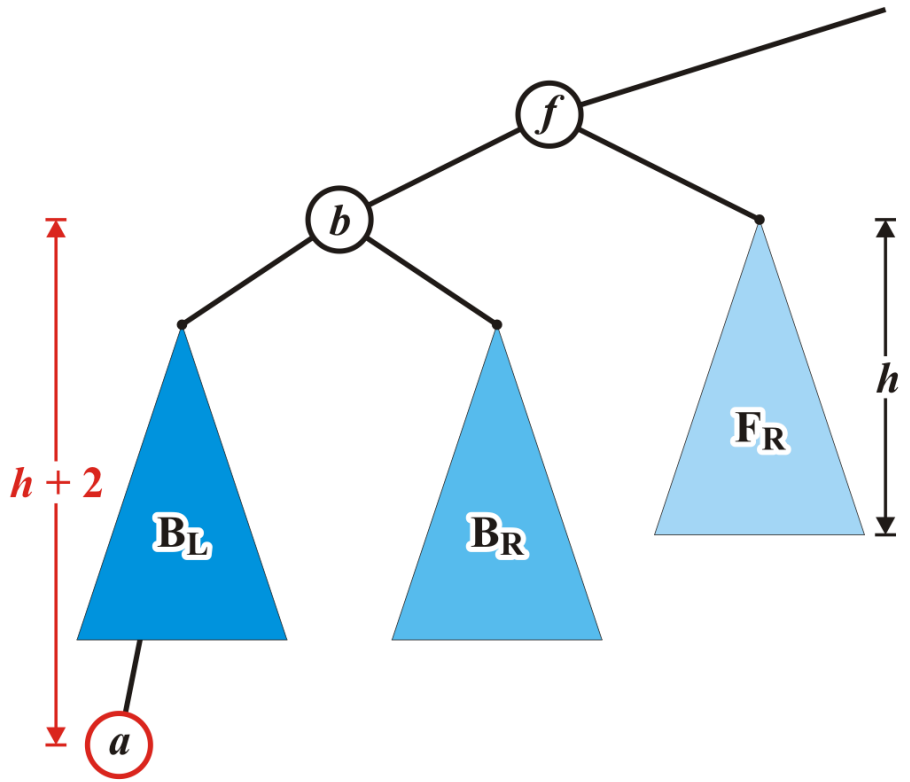
- Assume $\mathbf{B_L}$ remains balanced
- The tree rooted at ***b*** is also balanced



Maintain Balance: Case 1

The tree rooted at node *f* is now unbalanced

- We will correct the imbalance at this node

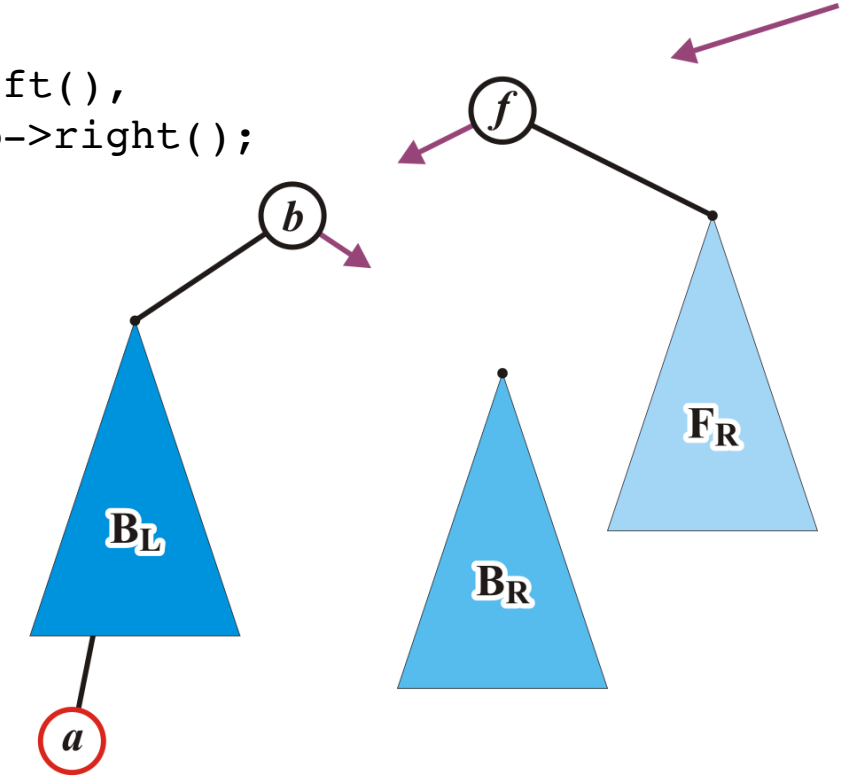


Maintain Balance: Case 1

We will modify these three pointers

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

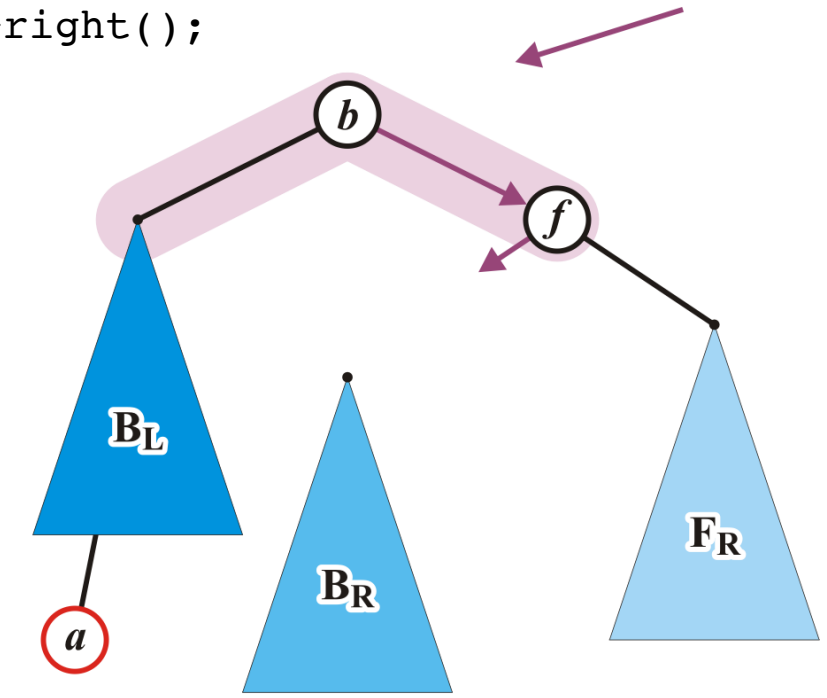
```
AVL_node<Type> *b = left(),
               *BR = b->right();
```



Maintain 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;
```



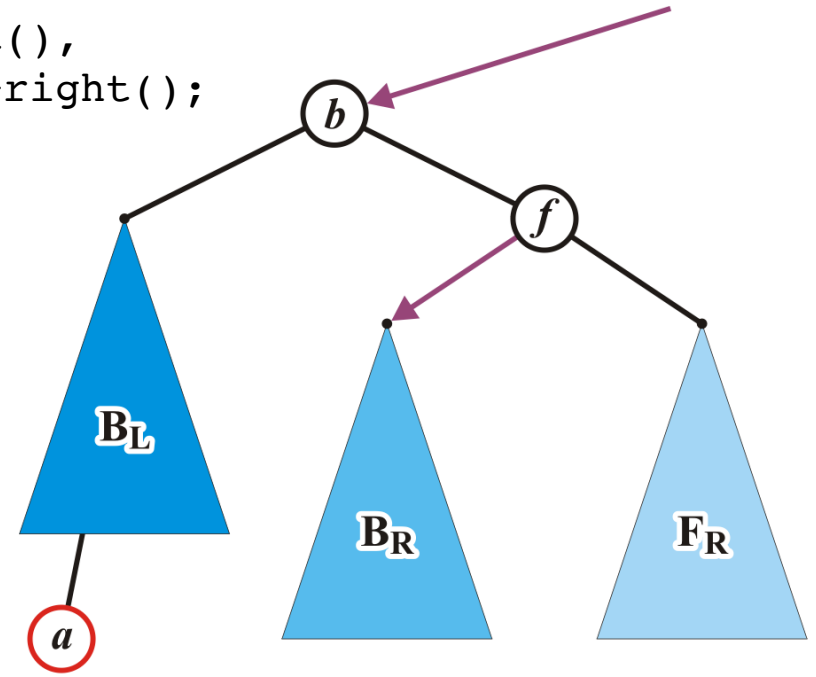
Maintain Balance: Case 1

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***

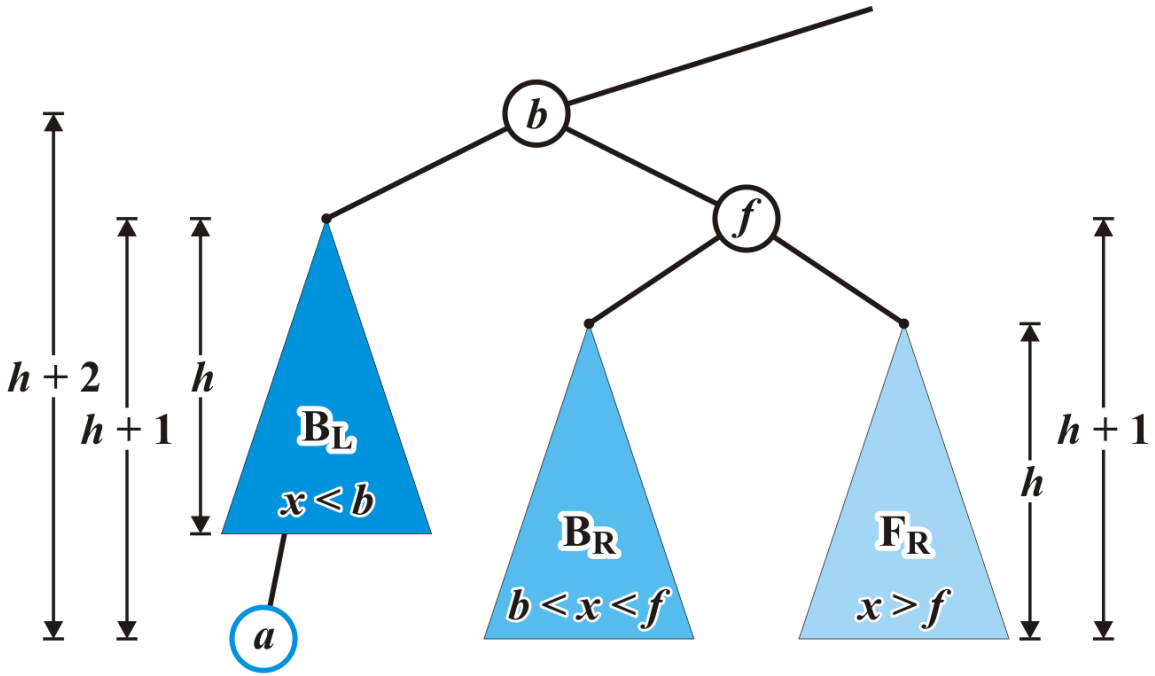
```
AVL_node<Type> *b = left(),
               *BR = b->right();
b->right_tree = this;

ptr_to_this   = b;
left_tree     = BR;
```



Maintain Balance: Case 1

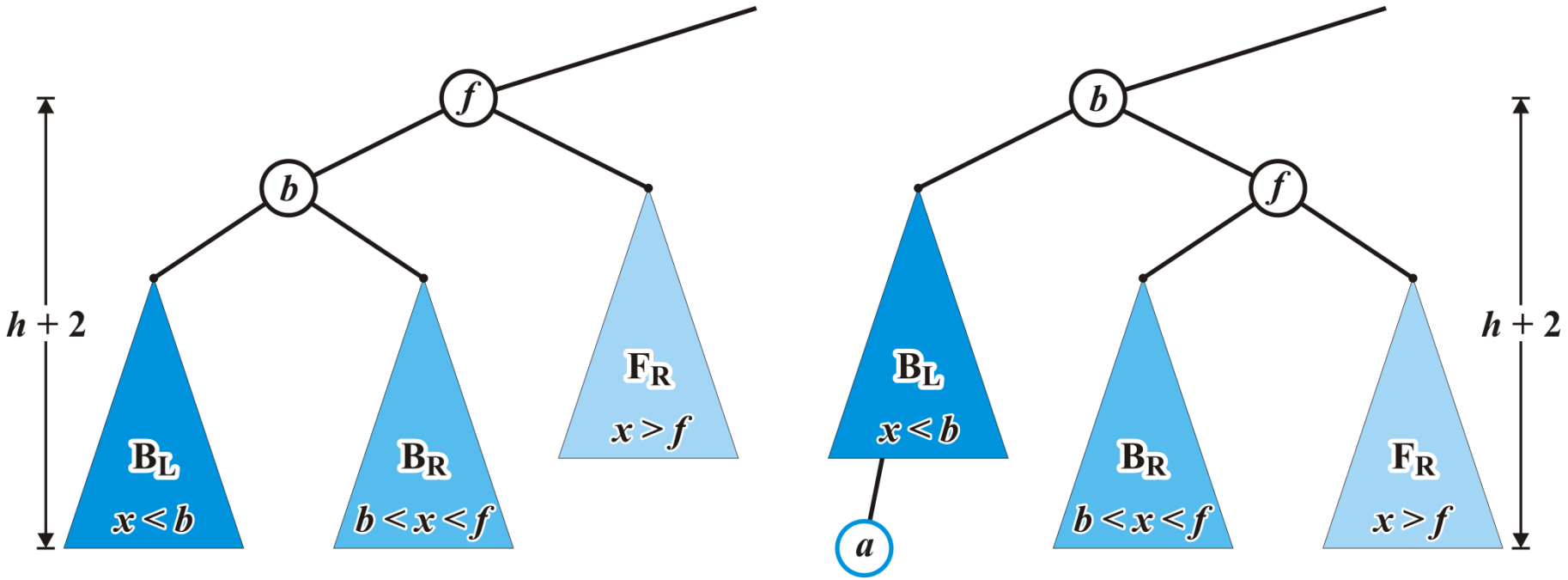
The nodes ***b*** and ***f*** are now balanced and all remaining nodes of the subtrees are in their correct positions



Maintain Balance: Case 1

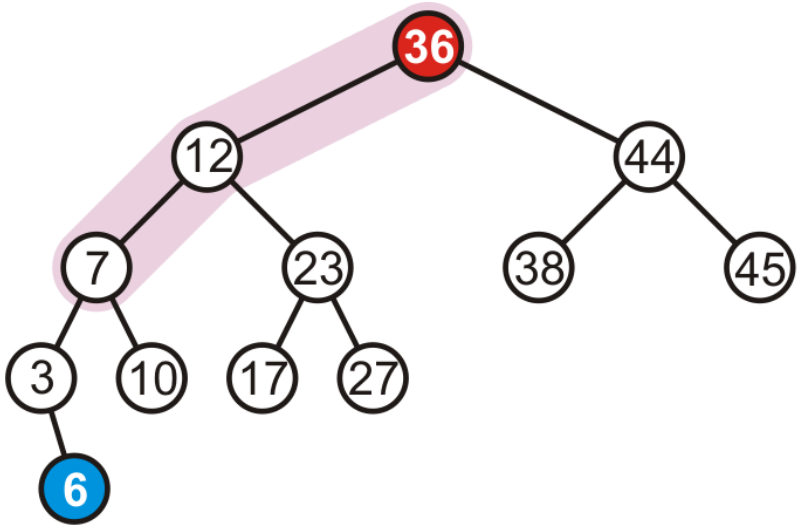
Q: will this insertion affect the balance of any ancestors all the way back to the root?

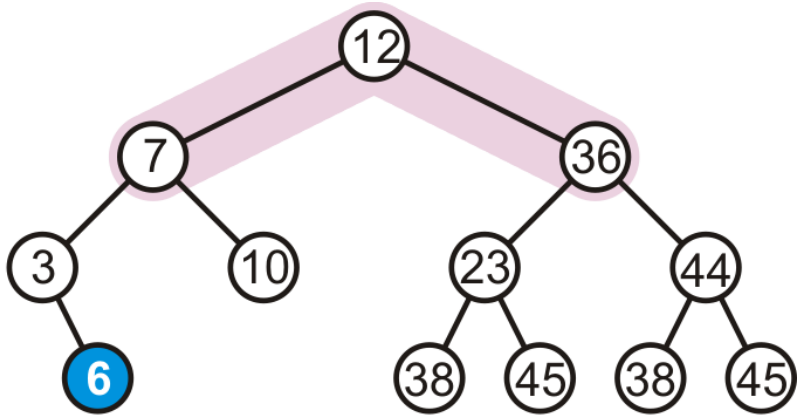
No, because height of the tree rooted at ***b*** equals the original height of the tree rooted at ***f***



Maintain Balance: Case 1

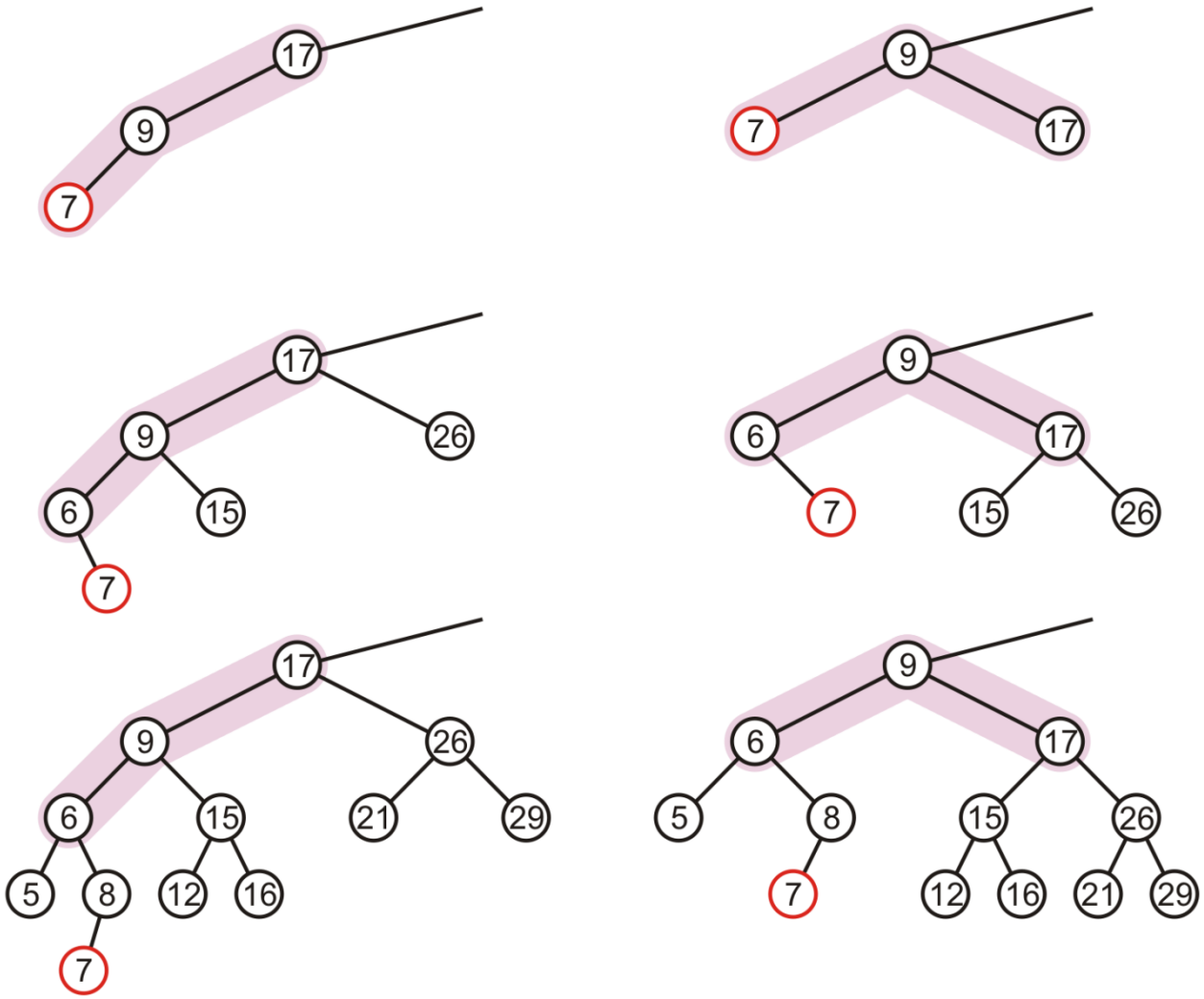
In our example case, the correction





Maintain Balance: Case 1

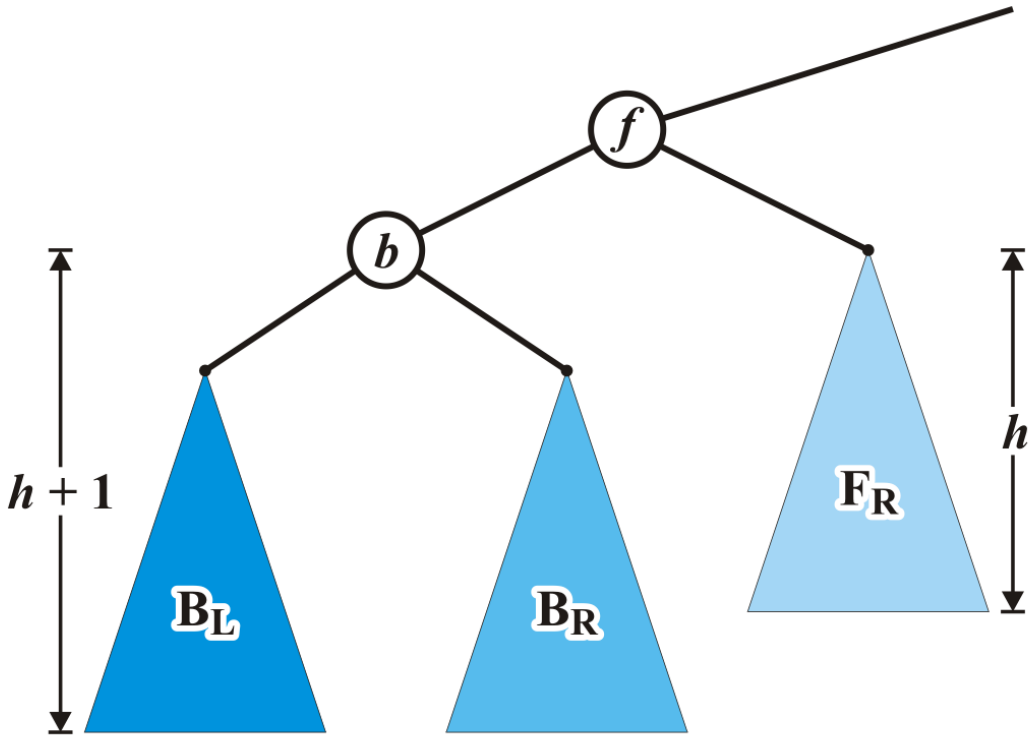
In our three sample cases with $h = -1$, 0, and 1, the node is now balanced and the same height as the tree before the insertion



LR/RL Rotation

Maintain Balance: Case 2

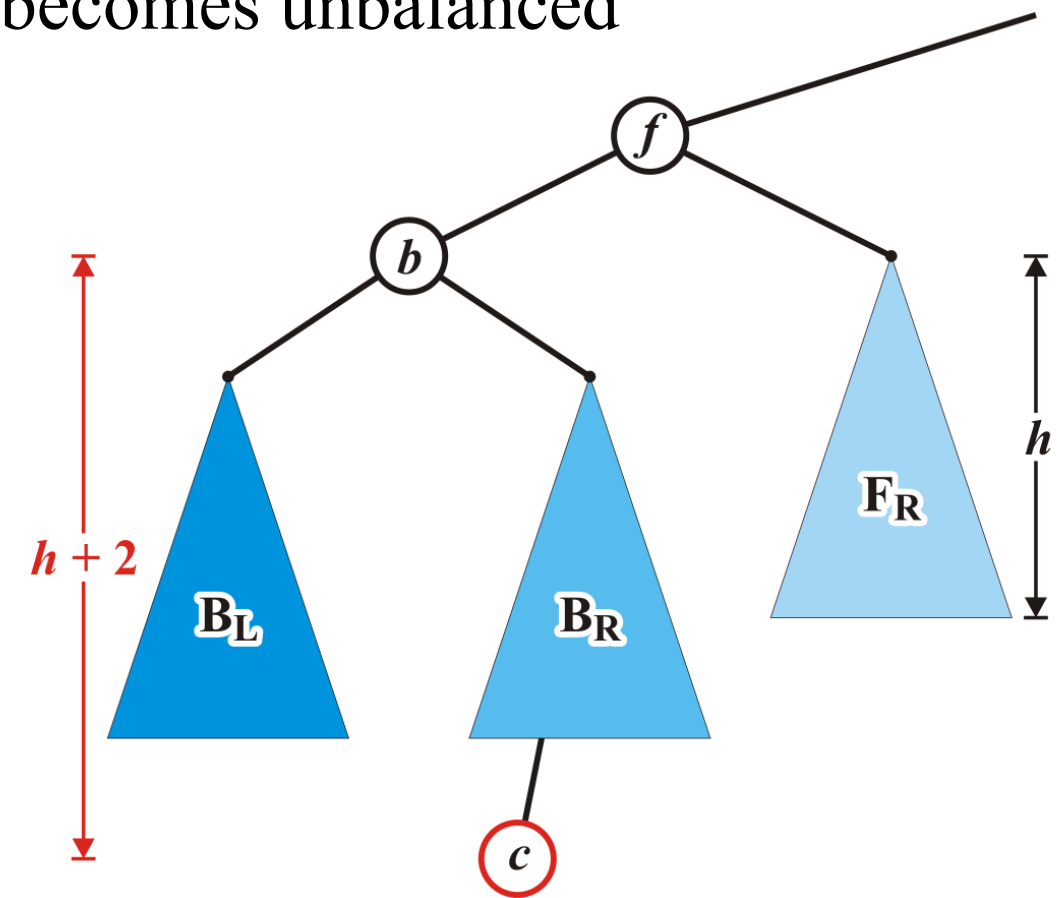
Alternatively, consider the insertion of ***c*** where ***b*** < ***c*** < ***f*** into our original tree



Maintain Balance: Case 2

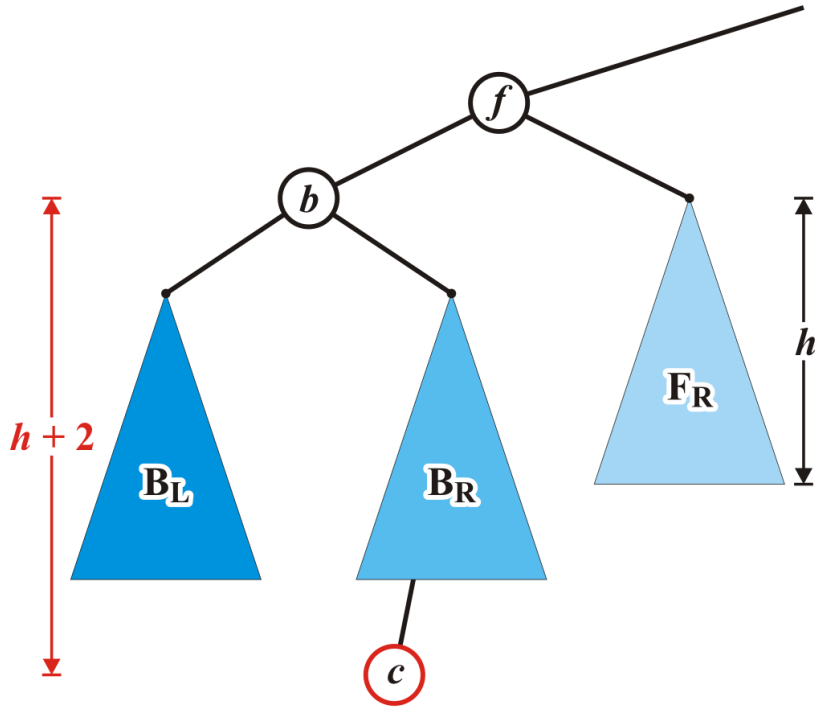
Assume that the insertion of ***c*** increases the height of $\mathbf{B_R}$

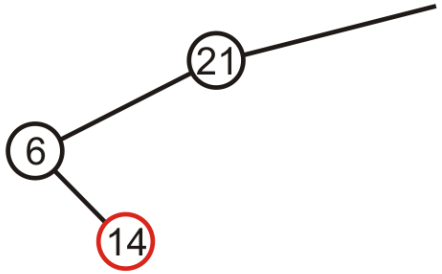
- Once again, ***f*** becomes unbalanced

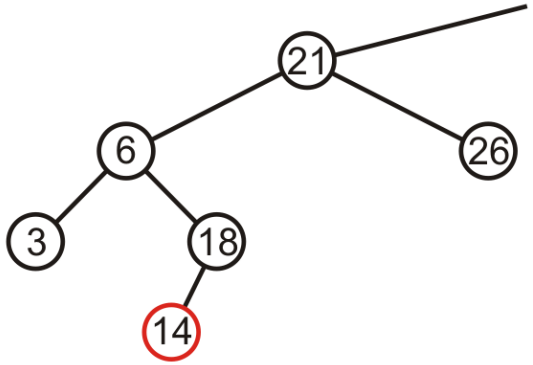


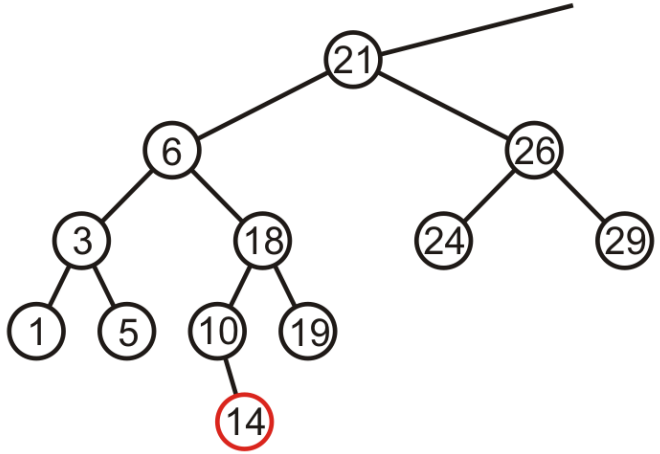
Maintain Balance: Case 2

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



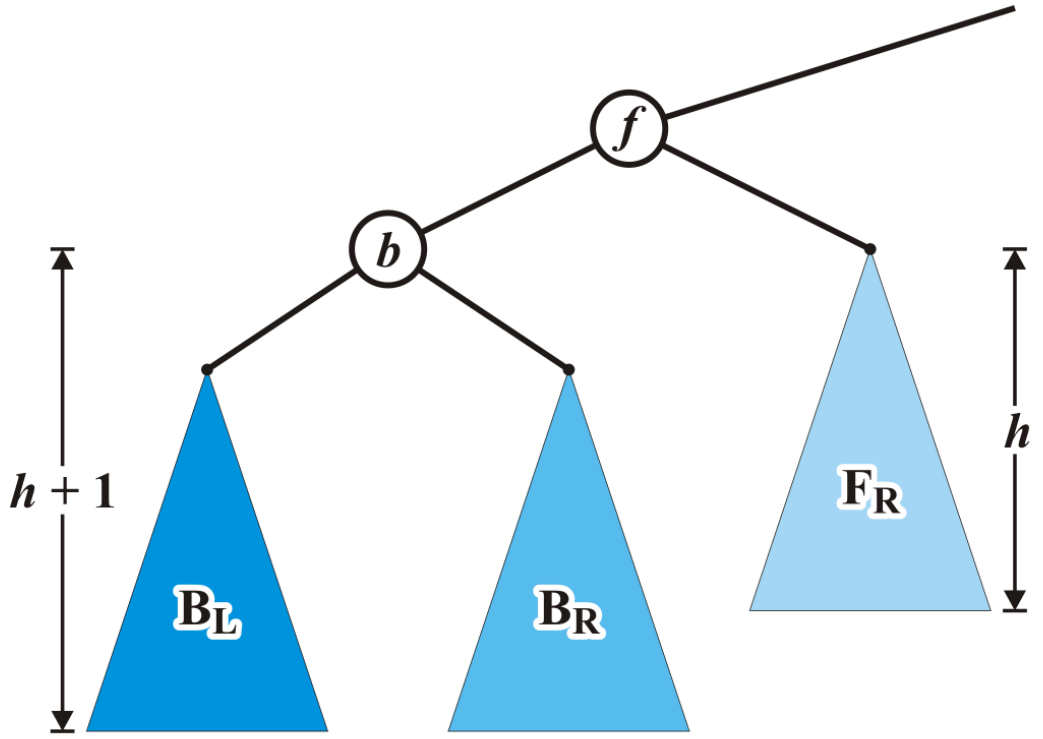






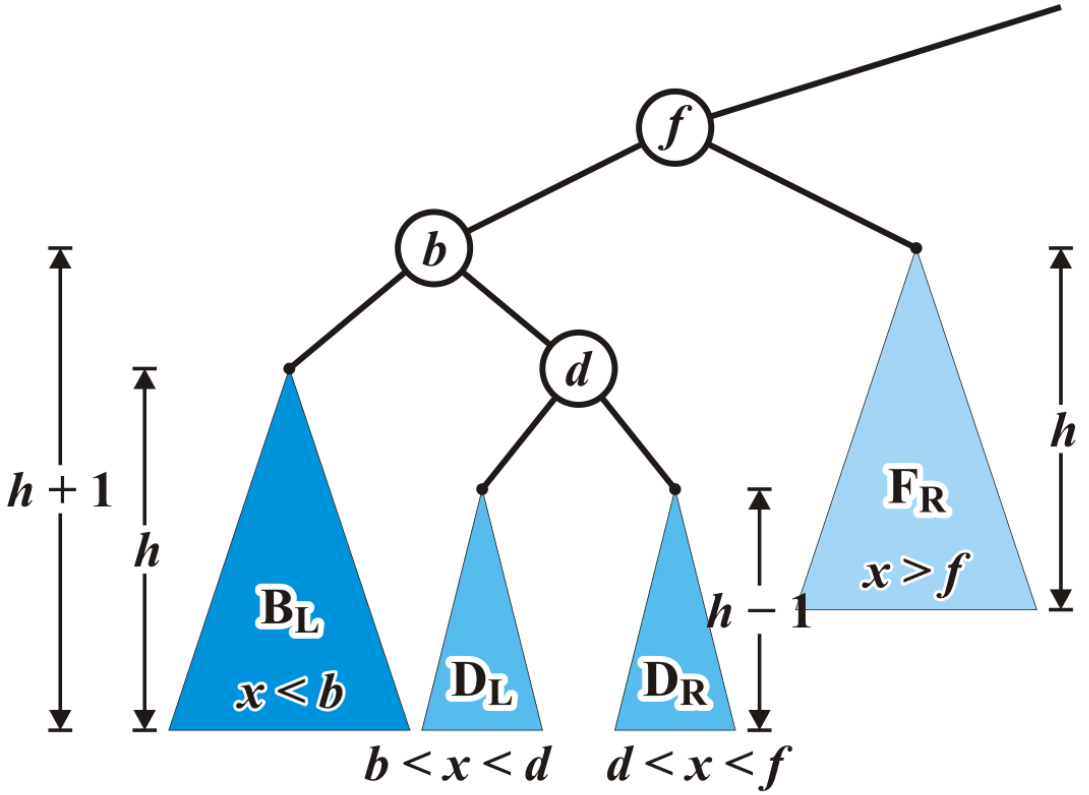
Maintain Balance: Case 2

We need to look into B_R



Maintain Balance: Case 2

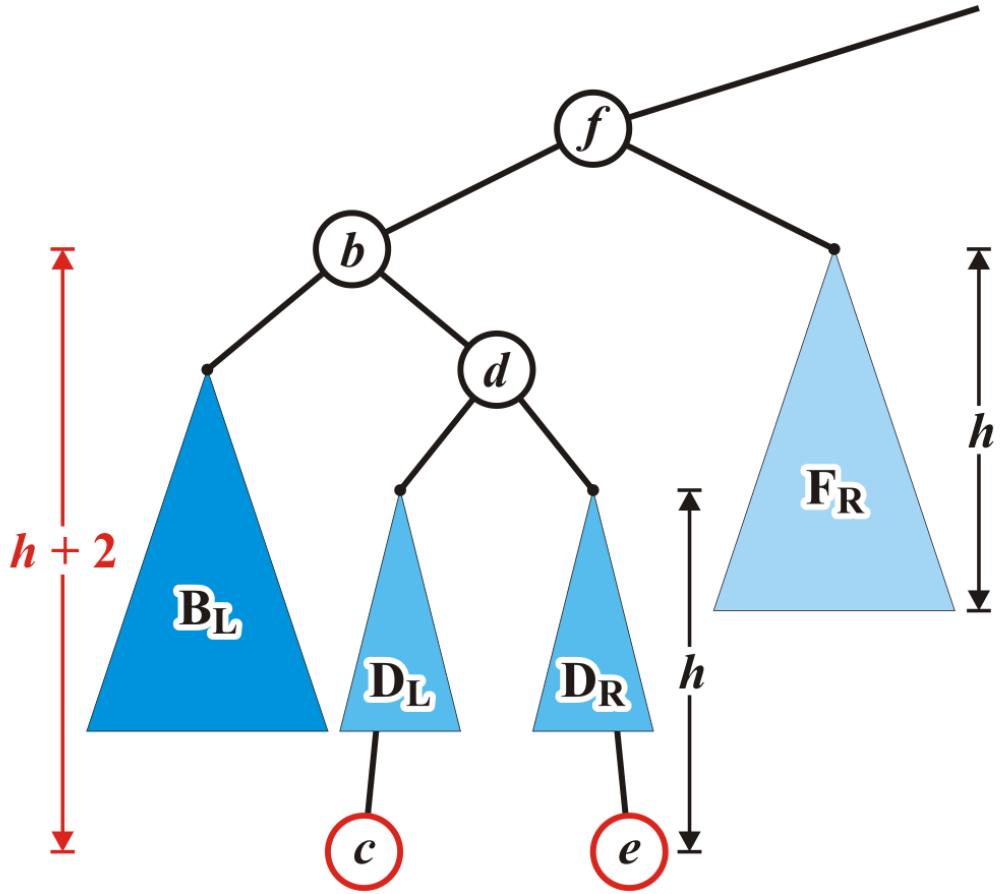
Re-label B_R as a tree rooted at *d* with two subtrees of height *h* – 1



Maintain Balance: Case 2

Now an insertion causes an imbalance at *f*

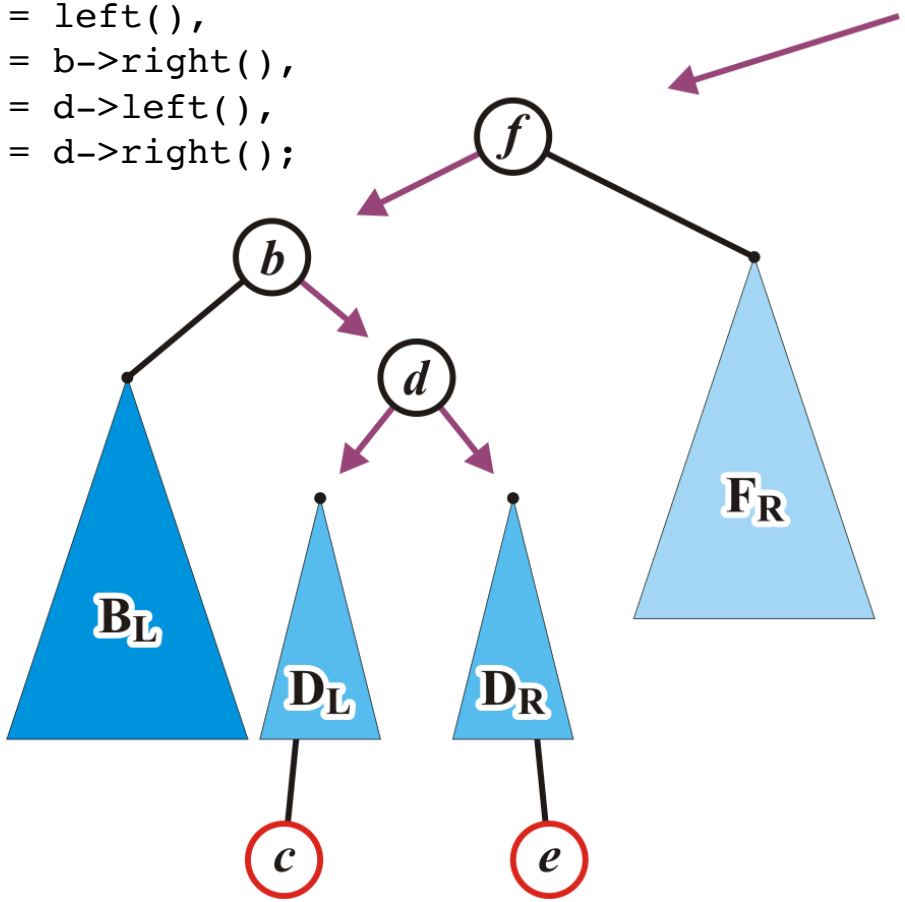
- The addition of either ***c*** or ***e*** will cause this



Maintain Balance: Case 2

We will reassign the following pointers

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

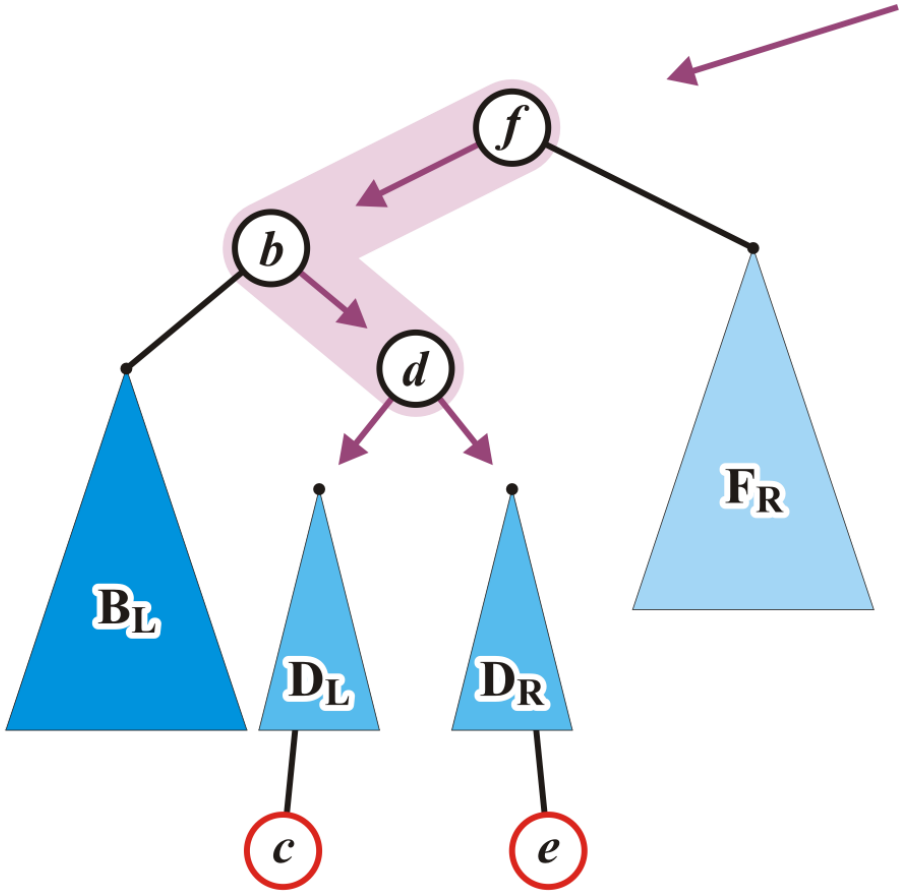


Maintain Balance: Case 2

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

- Recall the second prototypical example

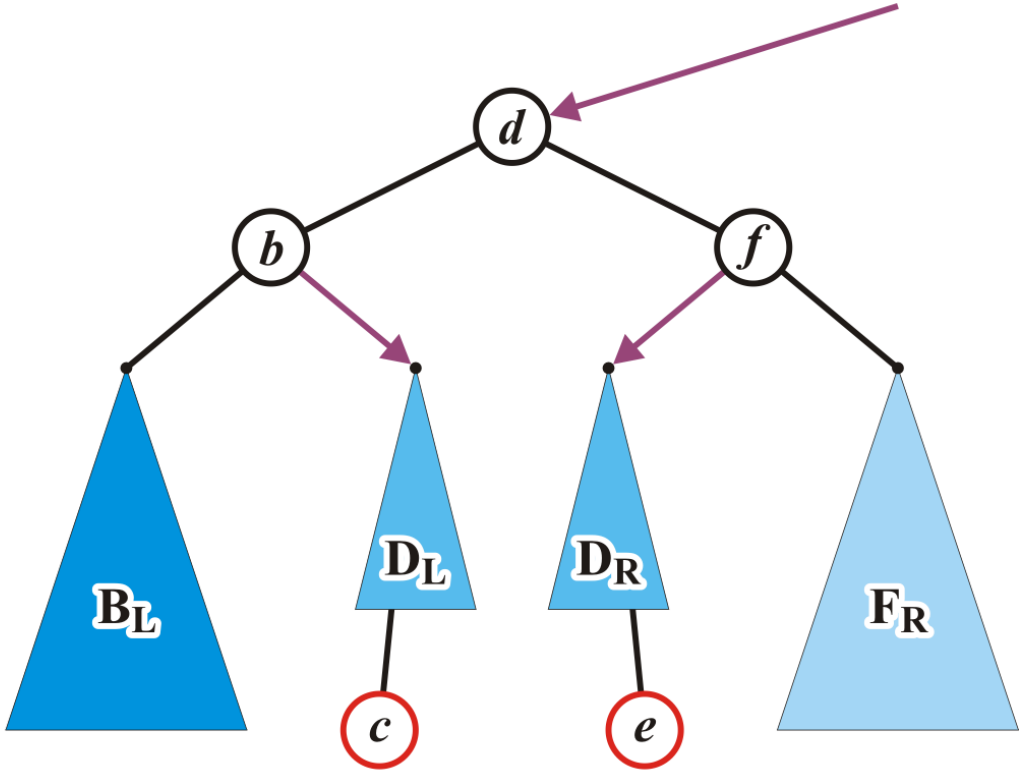
```
AVL_node<Type> *b  = left(),
               *d  = b->right(),
               *DL = d->left(),
               *DR = d->right();
```



Maintain Balance: Case 2

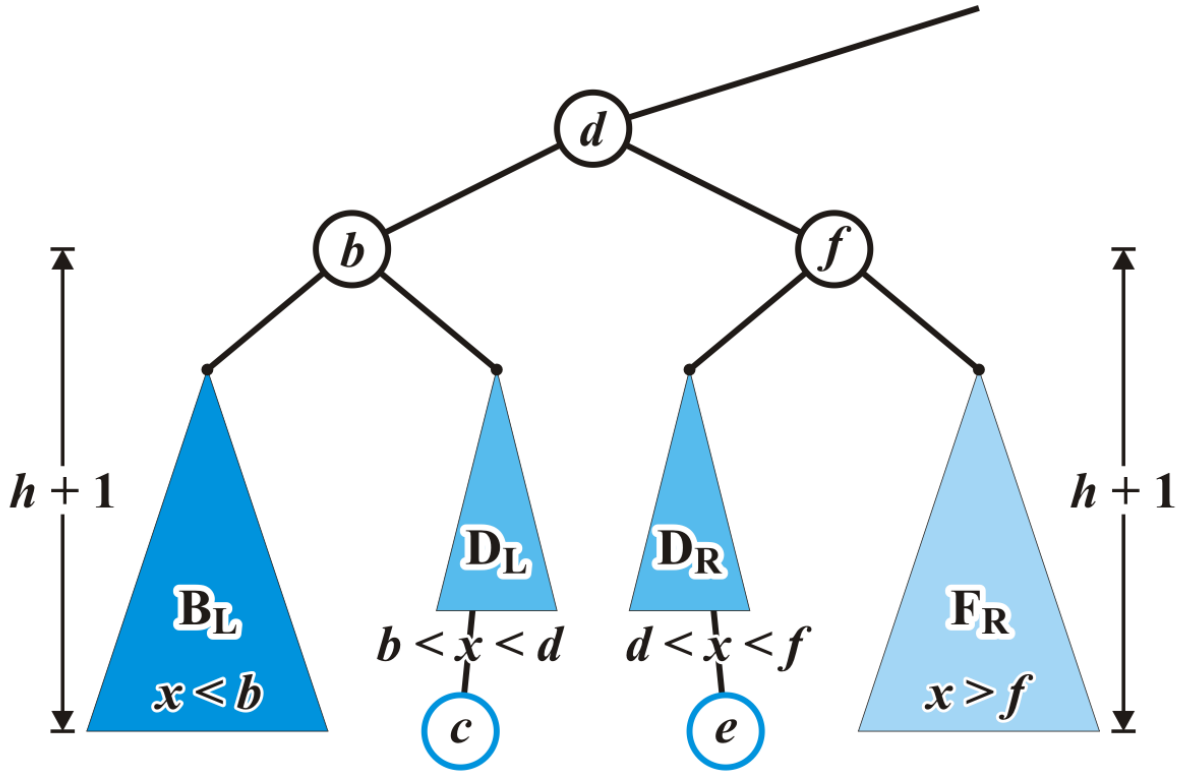
We also have to connect the two subtrees and original parent of ***f***

```
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;
```



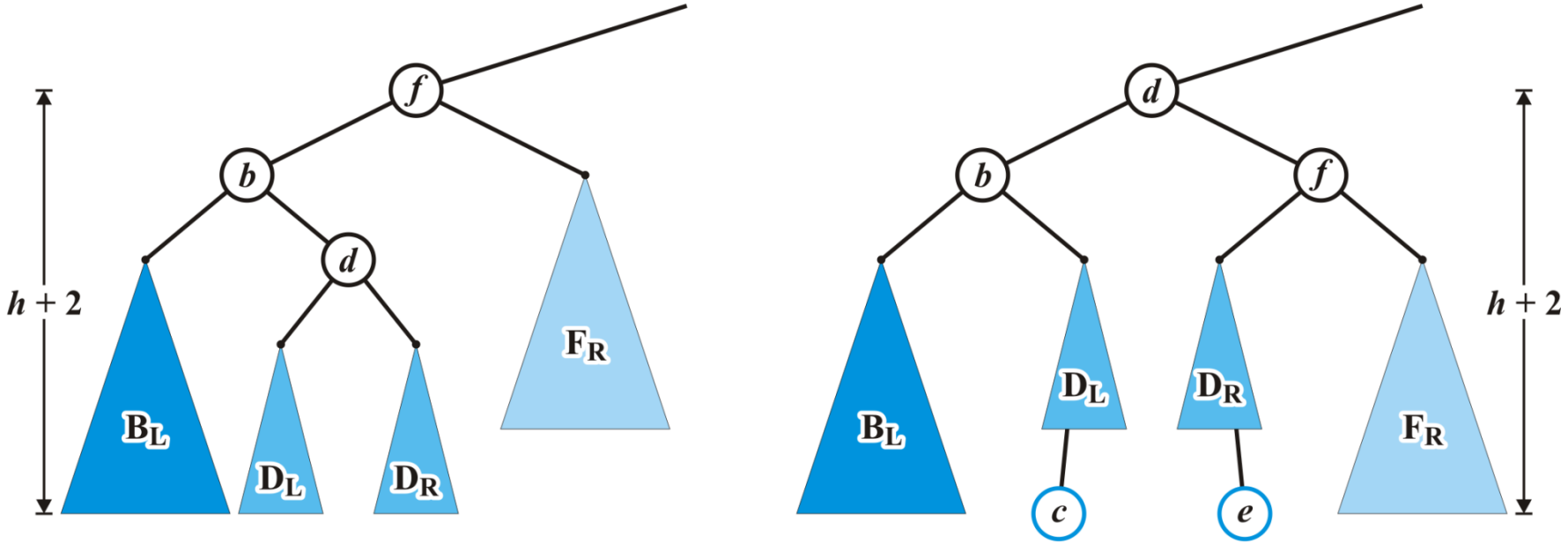
Maintain Balance: Case 2

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



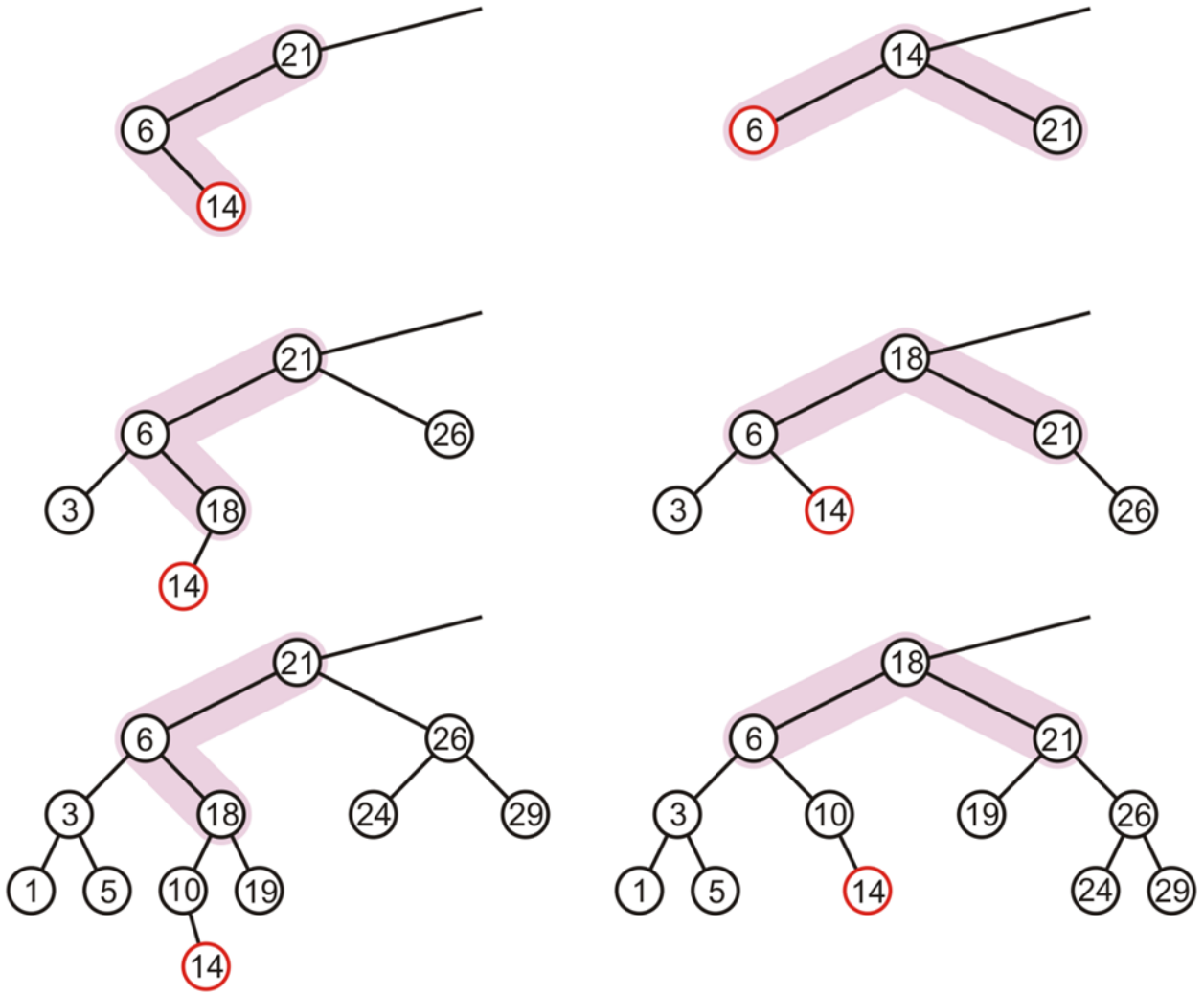
Maintain Balance: Case 2

Again, the height of the root did not change



Maintain Balance: Case 2

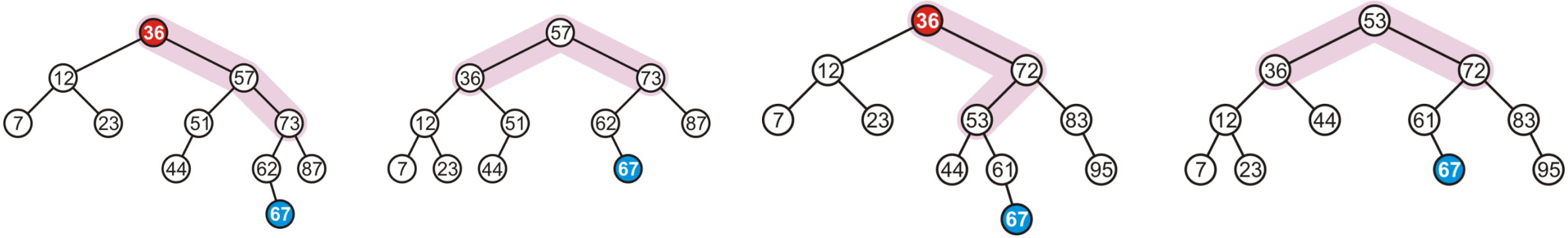
In our three sample cases with $h = -1, 0$, and 1, the node is now balanced and the same height as the tree before the insertion



Maintain Balance: Case 2

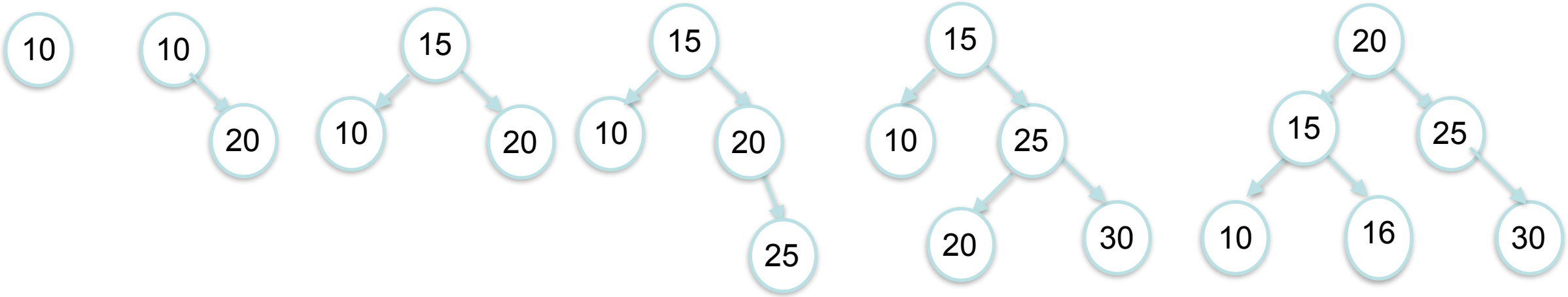
There are two symmetric cases to those we have examined:

- Insertions into the right-right sub-tree
- Insertions into either the right-left sub-tree



Question

- Insert the sequence {10, 20, 15, 25, 30, 16} in a AVL Tree



Deletion

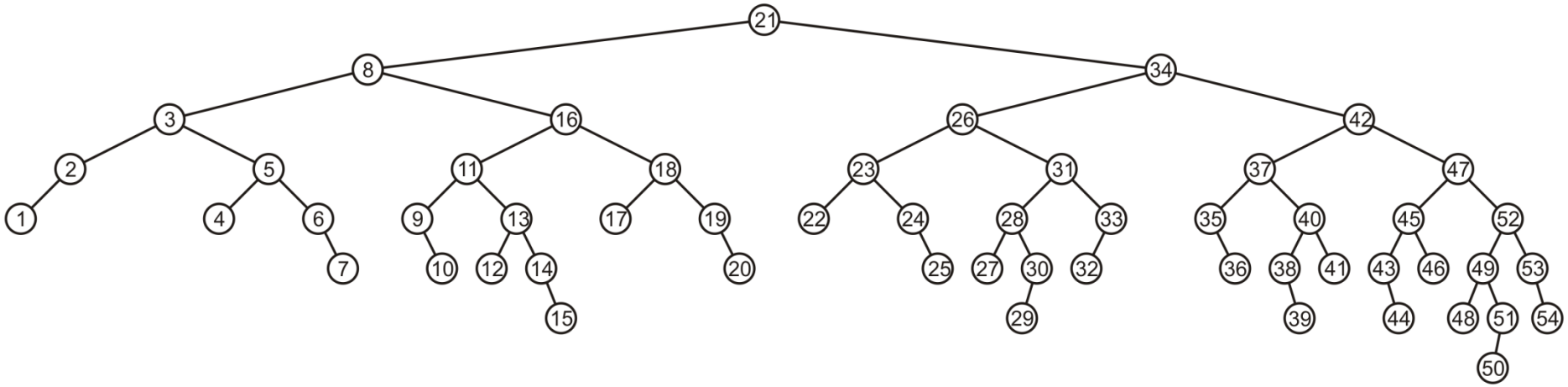
Erase

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 O(h) imbalances that must be corrected
  - Insertions will only cause one imbalance that must be fixed

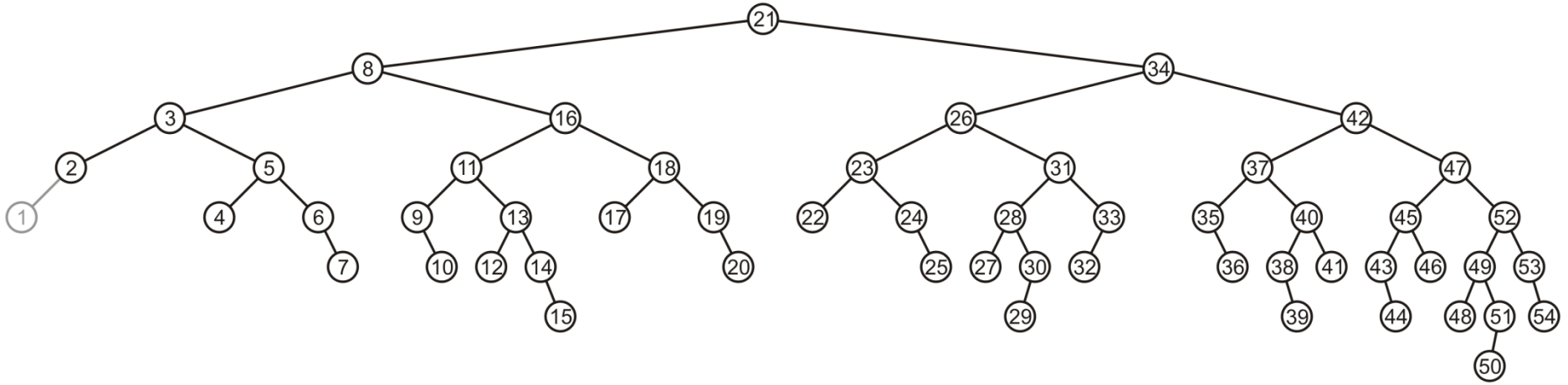
Erase

Consider the following AVL tree



Erase

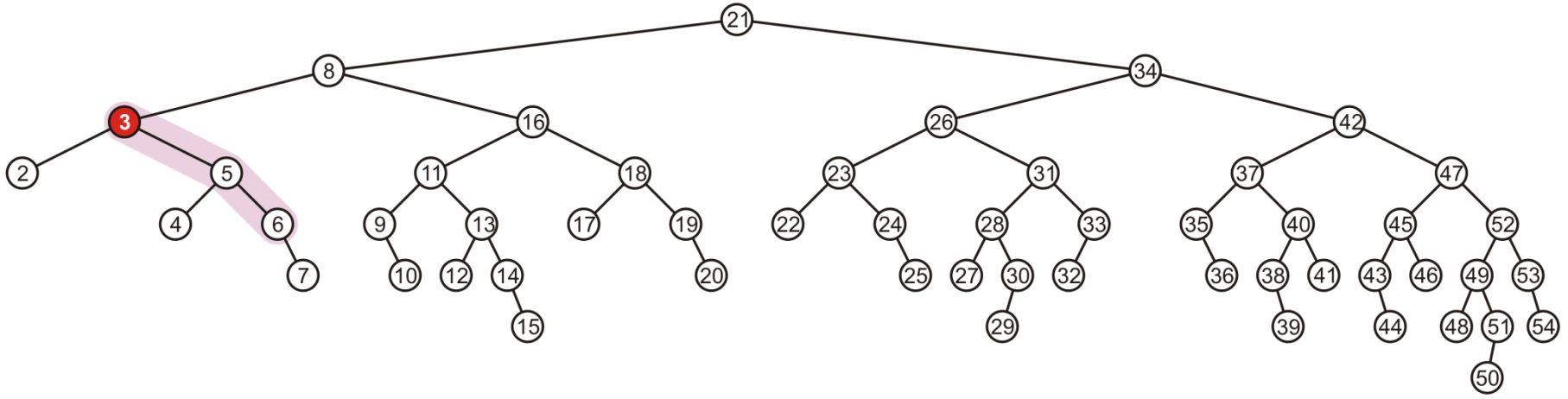
Suppose we erase the front node: 1



Erase

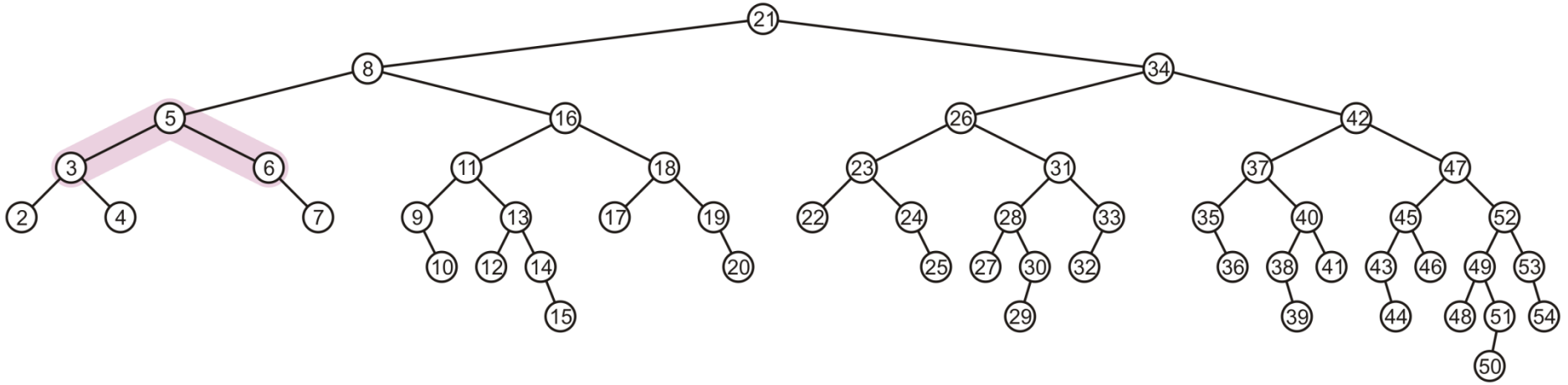
While its previous parent, 2, is not unbalanced, its grandparent 3 is

- The imbalance is in the right-right subtree



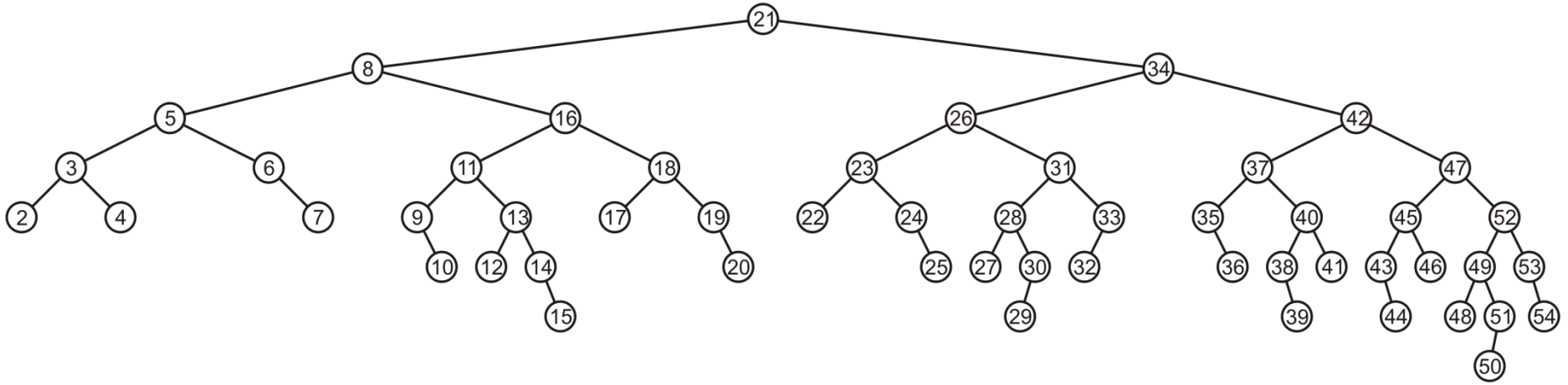
Erase

We can correct this with a simple balance



Erase

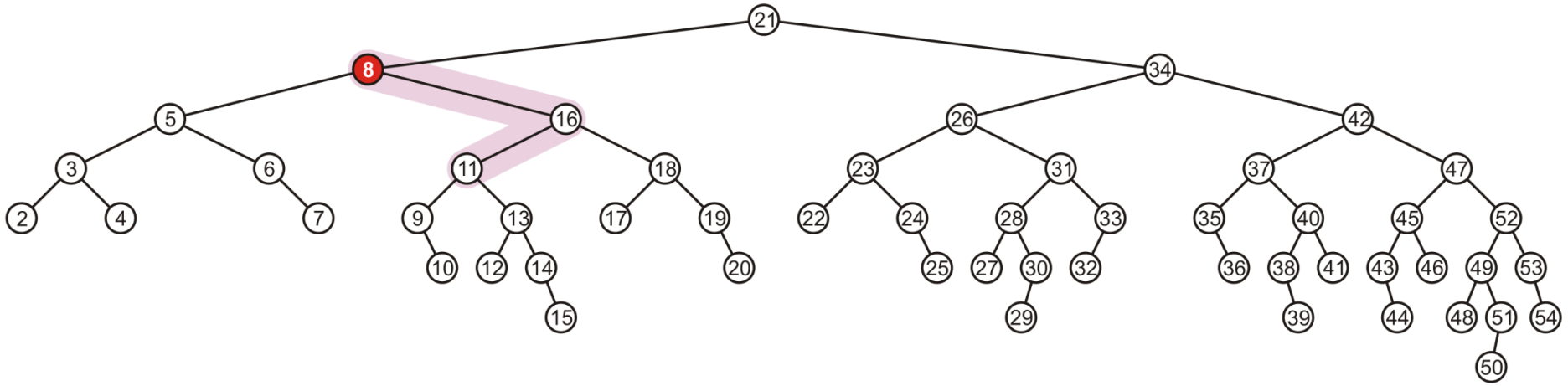
The node of that subtree, 5, is now balanced



Erase

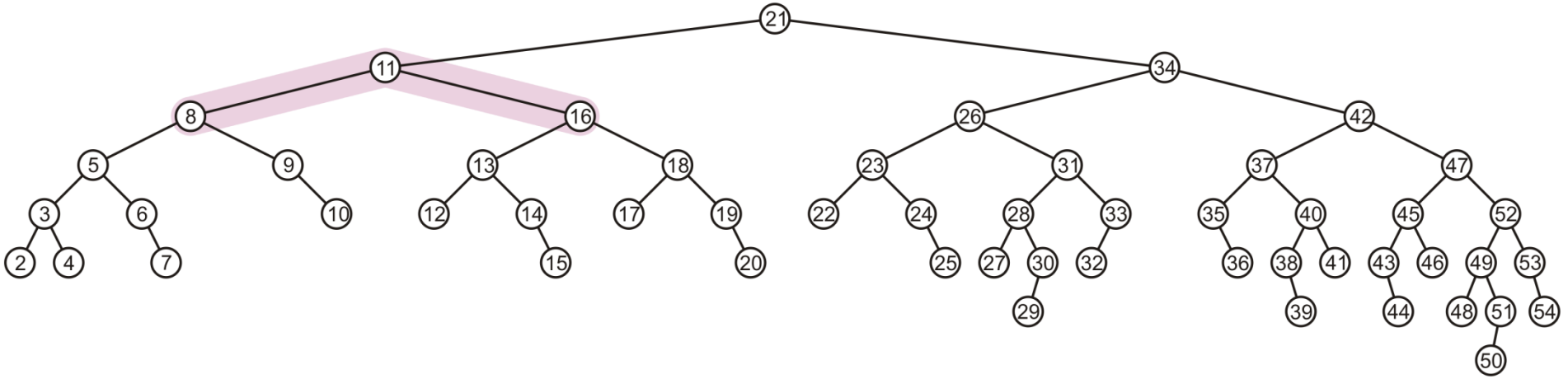
Recursing to the root, however, 8 is also unbalanced

- This is a right-left imbalance



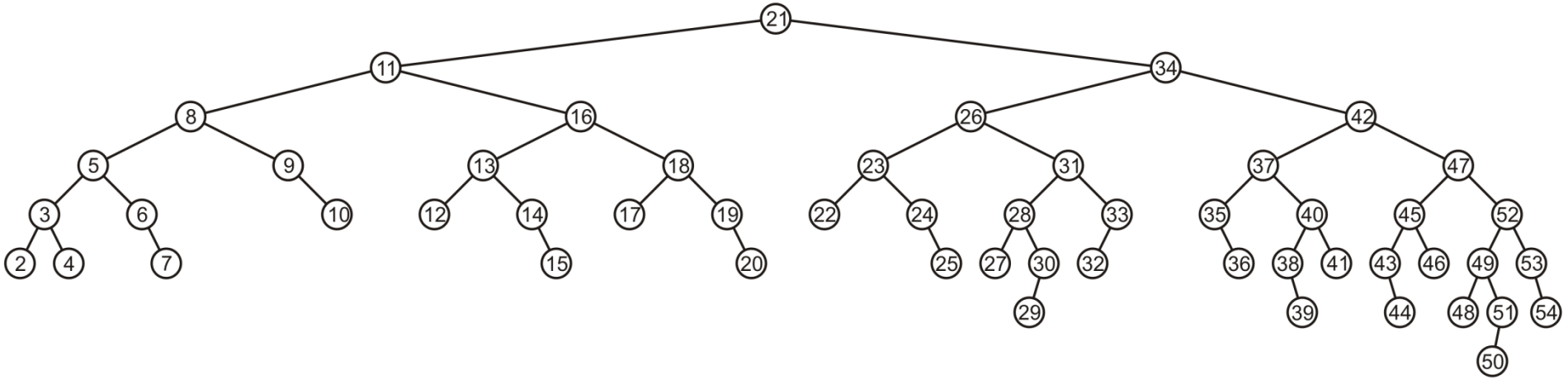
Erase

Promoting 11 to the root corrects the imbalance



Erase

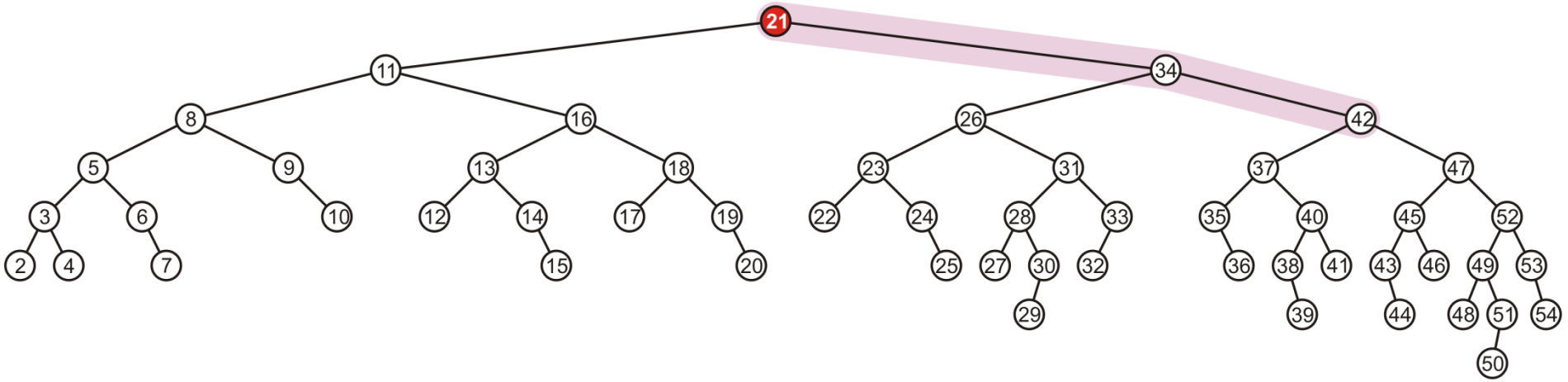
At this point, the node 11 is balanced



Erase

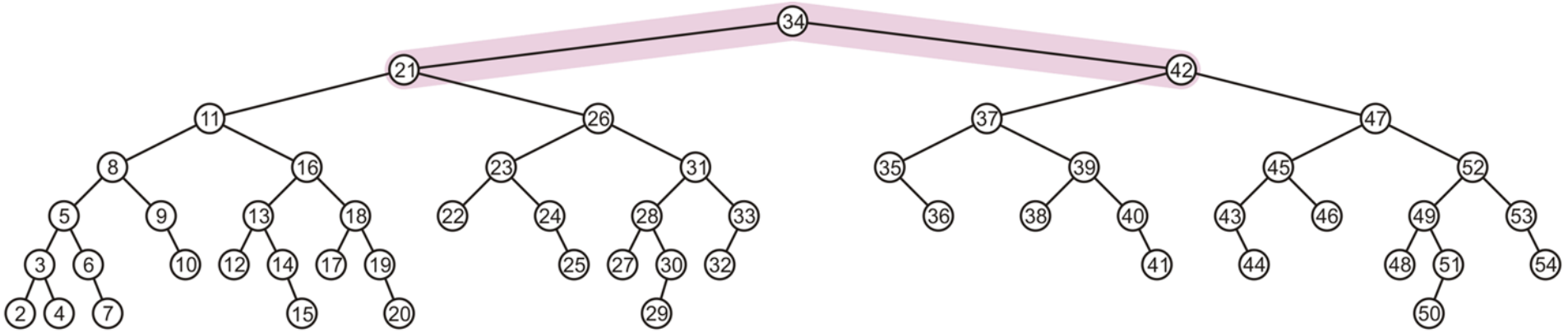
Still, the root node is unbalanced

- This is a right-right imbalance



Erase

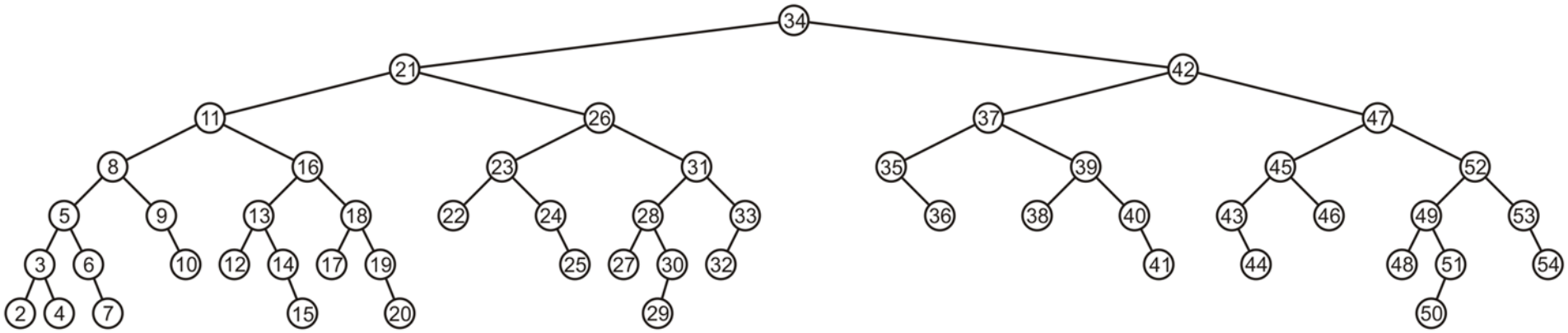
Again, a simple balance fixes the imbalance



Erase

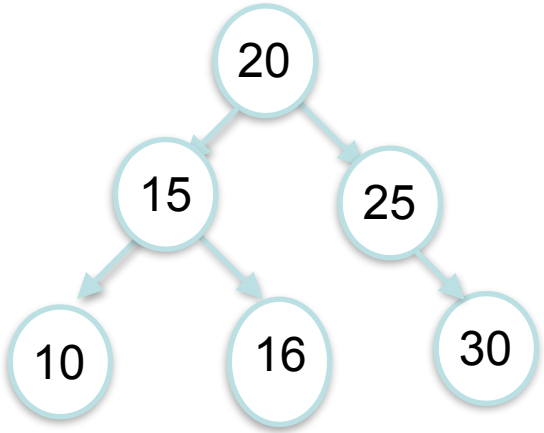
The resulting tree is now AVL balanced

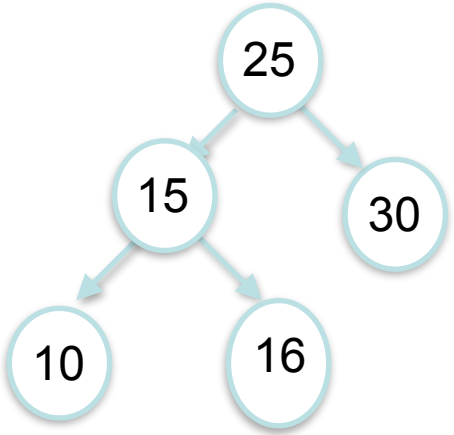
- Note, few erases will require one balance, even fewer will require more than one



Question

- Delete 20 in the given AVL Tree





Time Complexity

Insertion

- May require one correction to maintain balance
- Each correction requires $\Theta(1)$ time
- The total time is $O(h) + O(1) = O(\ln(n))$

Erasion

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

Quiz Time