Lecture 6
AVL Tree, Huffman Tree

陆清怡

2022.11.18

991《数据结构与算法》考纲

5、树

（1）树的概念和性质。

（2）二叉树的概念、性质和实现。

（3）二叉树的顺序存储结构和链式存储结构。

（4）遍历二叉树。

（5）树和森林的存储结构、遍历。

（6）堆与优先队列。

（6）二叉排序树。

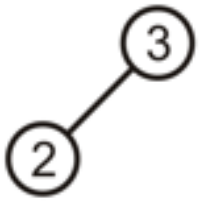
（7）平衡二叉树。

（8）哈夫曼(Huffman)树和哈夫曼编码。

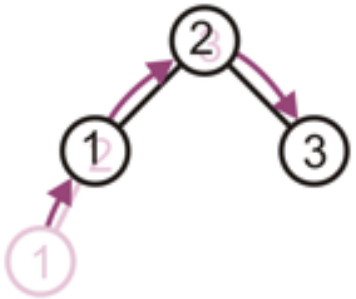
AVL Tree

Prototypical Examples

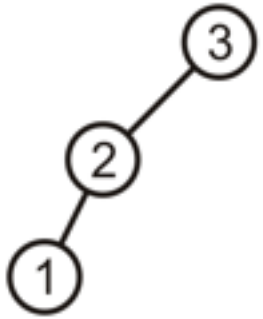
These two examples demonstrate how we can correct for imbalances: starting with this tree, add 1:



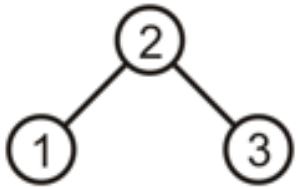
Promote 2 to the root, demote 3 to be 2's right child, and 1 remains the left child of 2



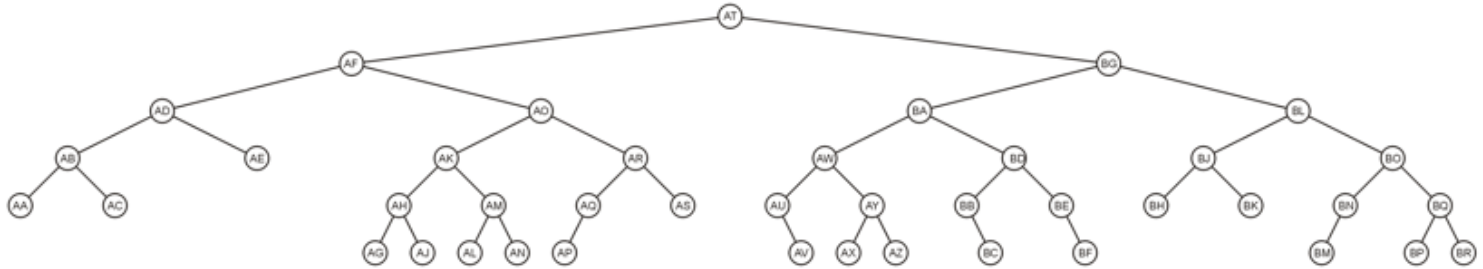
This is more like a linked list; however, we can fix this…



The result is a perfect, though trivial tree



AVL Trees



Named after Adelson-Velskii and Landis

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

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

Recall:

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

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:

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

The solution?

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

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

Height of an AVL Tree

This is approximately

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

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

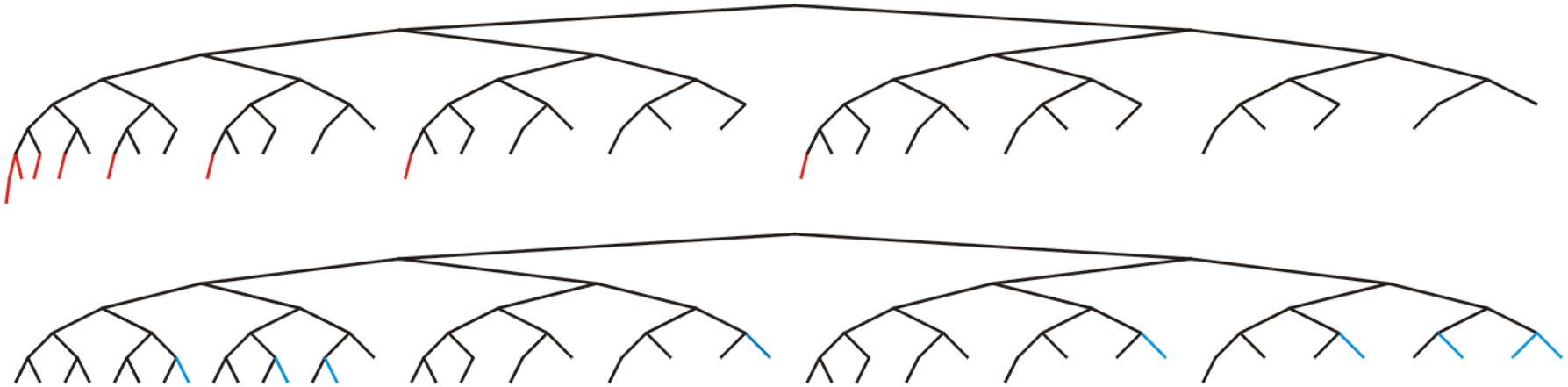
- That is, $F(h) = \Theta(\phi^h)$

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

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

Height of an AVL Tree

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



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

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

Maintaining Balance

Observe that:

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

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

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

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

```
int tree_height;
```

- This variable is updated during inserting and erasing

Only insert and erase may change the height

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

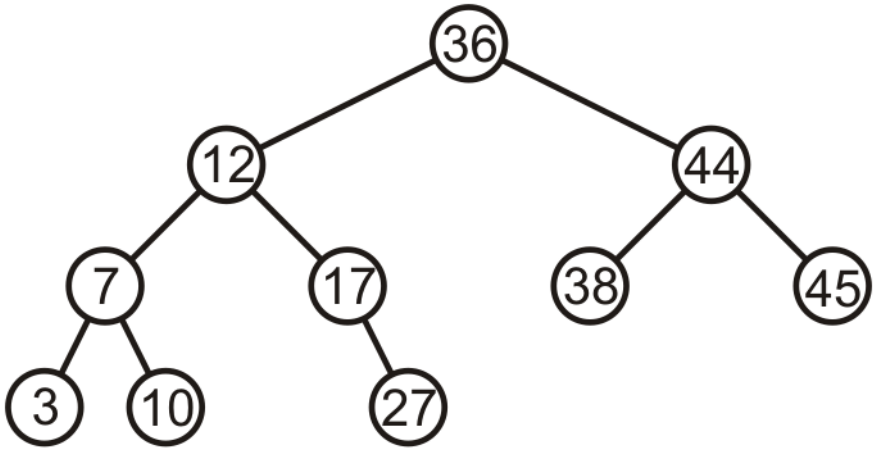
Insert

```
template <typename Type>
bool AVL_node<Type>::insert( const Type & obj, AVL_node<Type> *&ptr_to_this ) {
    if ( empty() ) {
        ptr_to_this = new AVL_node( obj );
        return true;
    } else if ( obj < element ) {
        if ( left()->insert( obj, left_tree ) ) {
            tree_height = max( height(), 1 + left()->height() );
            return true;
        } else {
            return false;
        }
    } else if ( obj > element ) {
        // ...
    } else {
        return false;
    }
}
```

Maintaining Balance

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

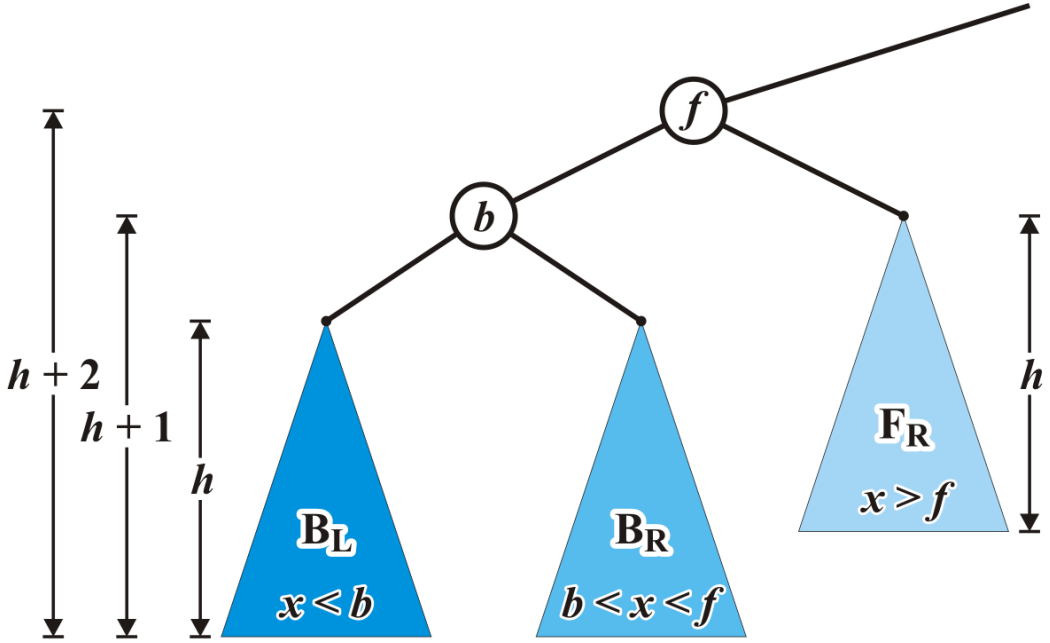
- The heights of the sub-trees must differ by 1
- The insertion must increase the height of the deeper sub-tree by 1



Maintaining Balance: Case 1

Consider the following setup

- Each blue triangle represents a tree of height h

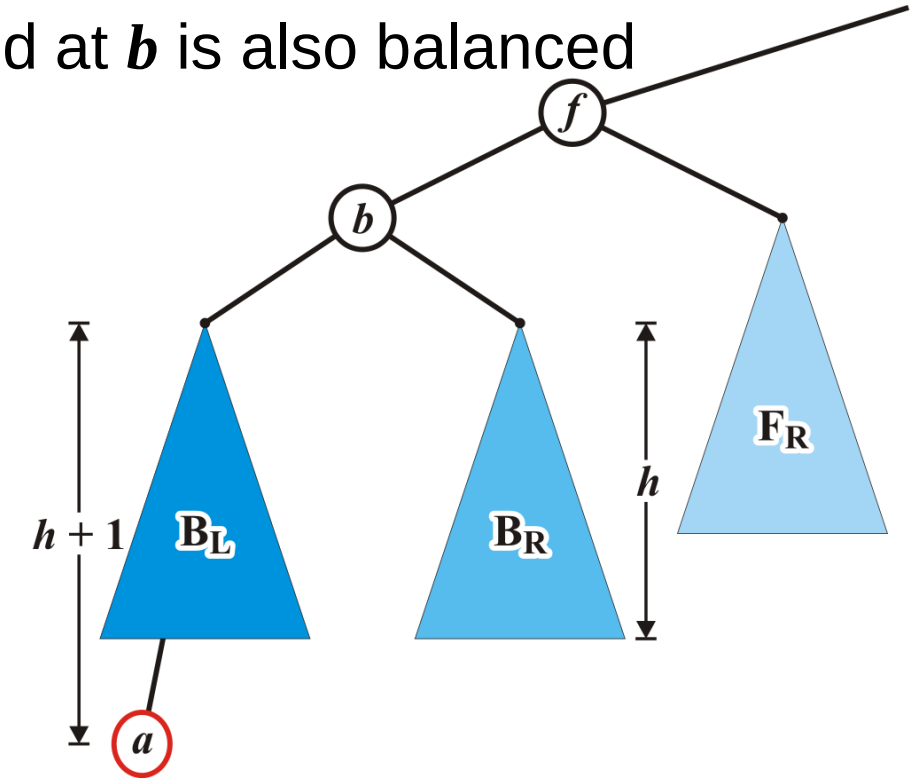


Maintaining Balance: Case 1

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

left-left

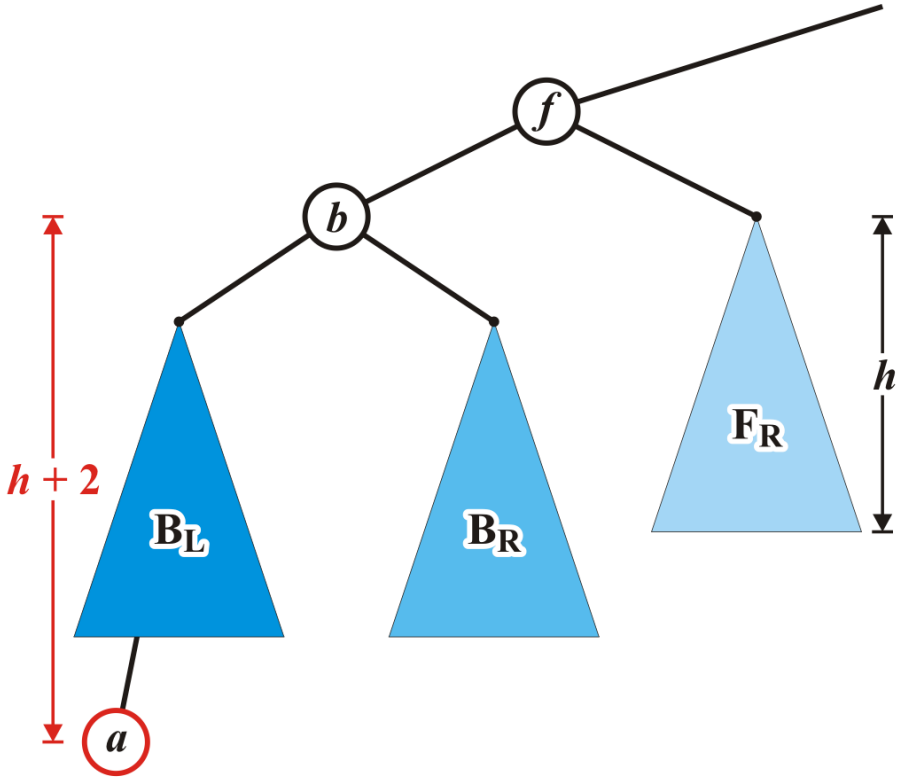
- Assume $\mathbf{B_L}$ remains balanced
- The tree rooted at ***b*** is also balanced



Maintaining Balance: Case 1

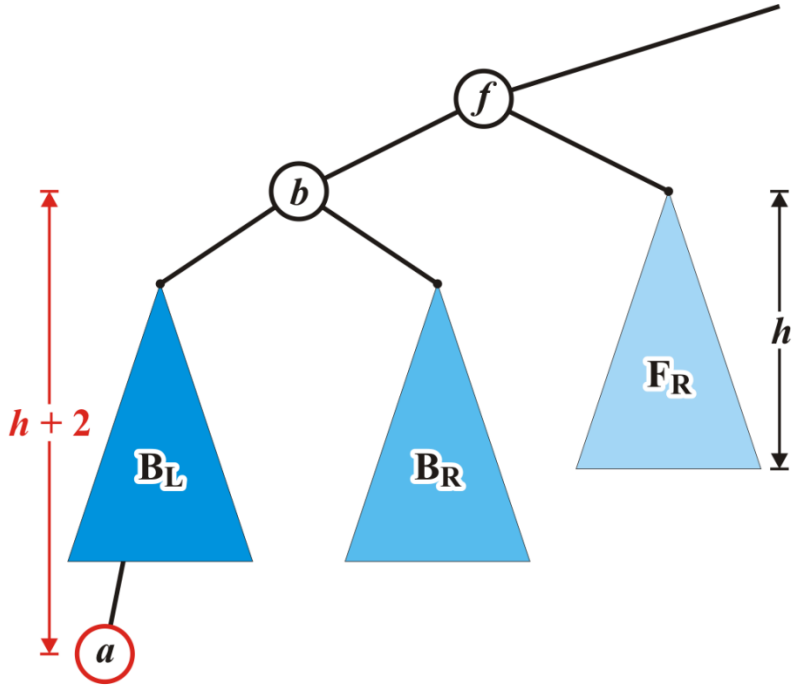
The tree rooted at node f is now unbalanced

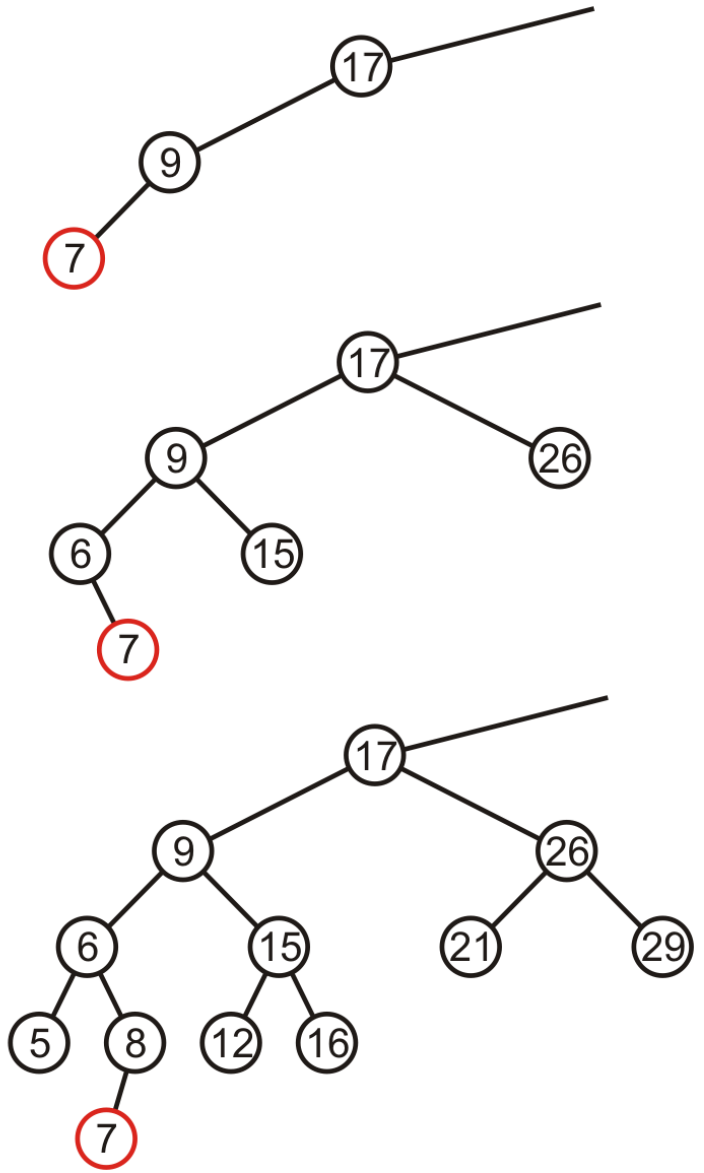
- We will correct the imbalance at this node



Maintaining Balance: Case 1

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



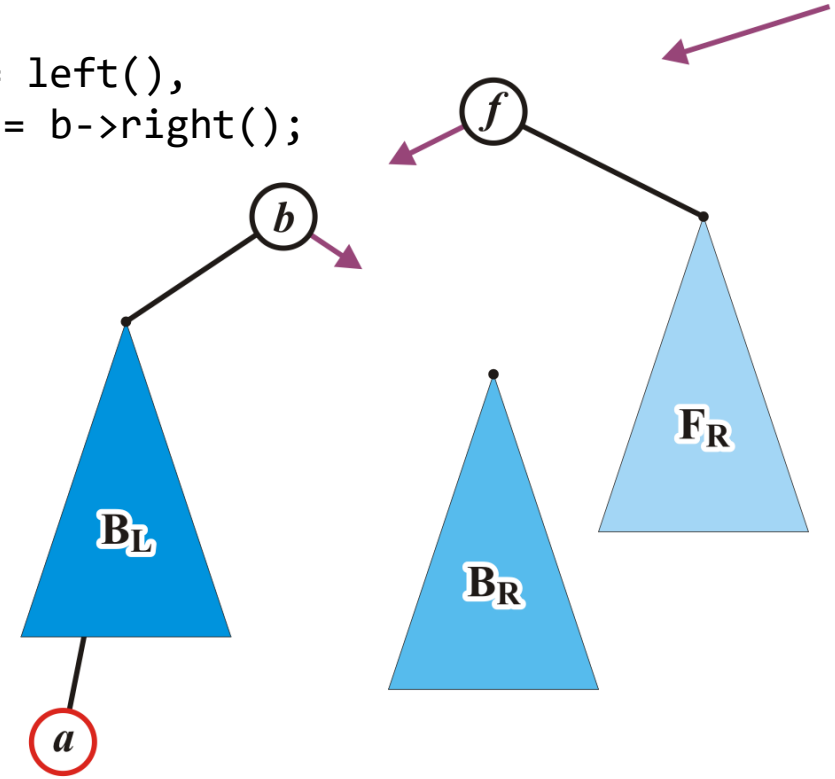


Maintaining Balance: Case 1

We will modify these three pointers

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

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

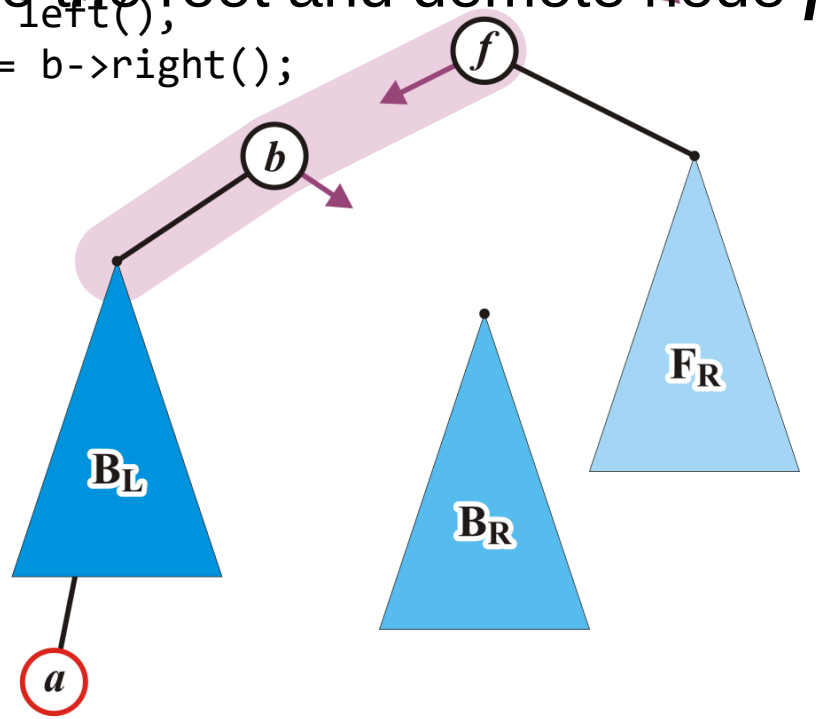


Maintaining Balance: Case 1

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

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

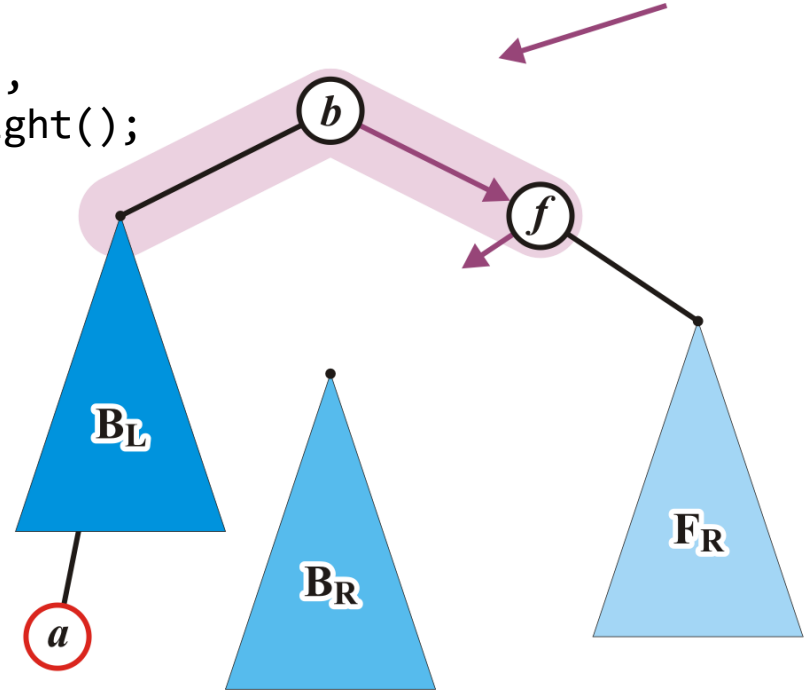
```
AVL_node<Type> *b = left();
               *BR = b->right();
```



Maintaining Balance: Case 1

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

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

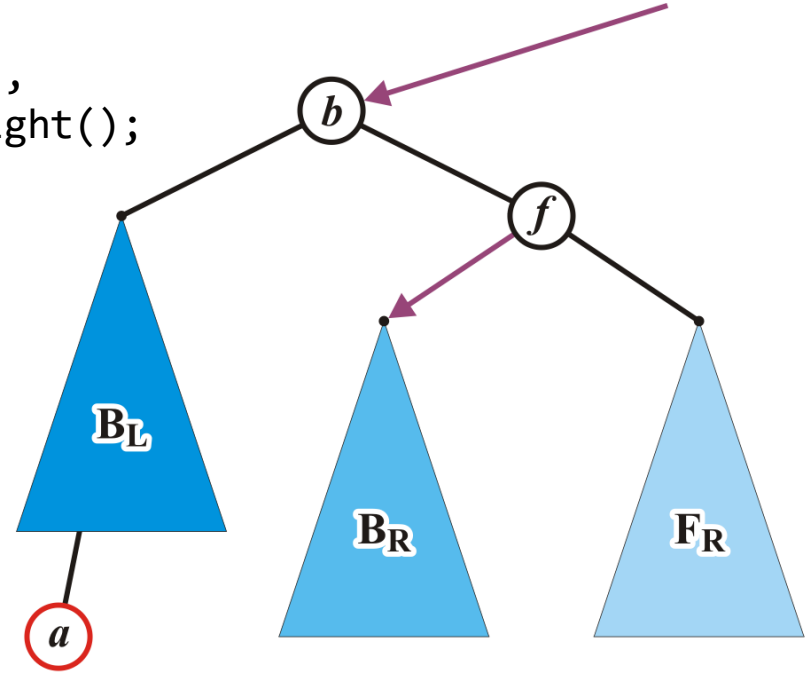


Maintaining Balance: Case 1

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

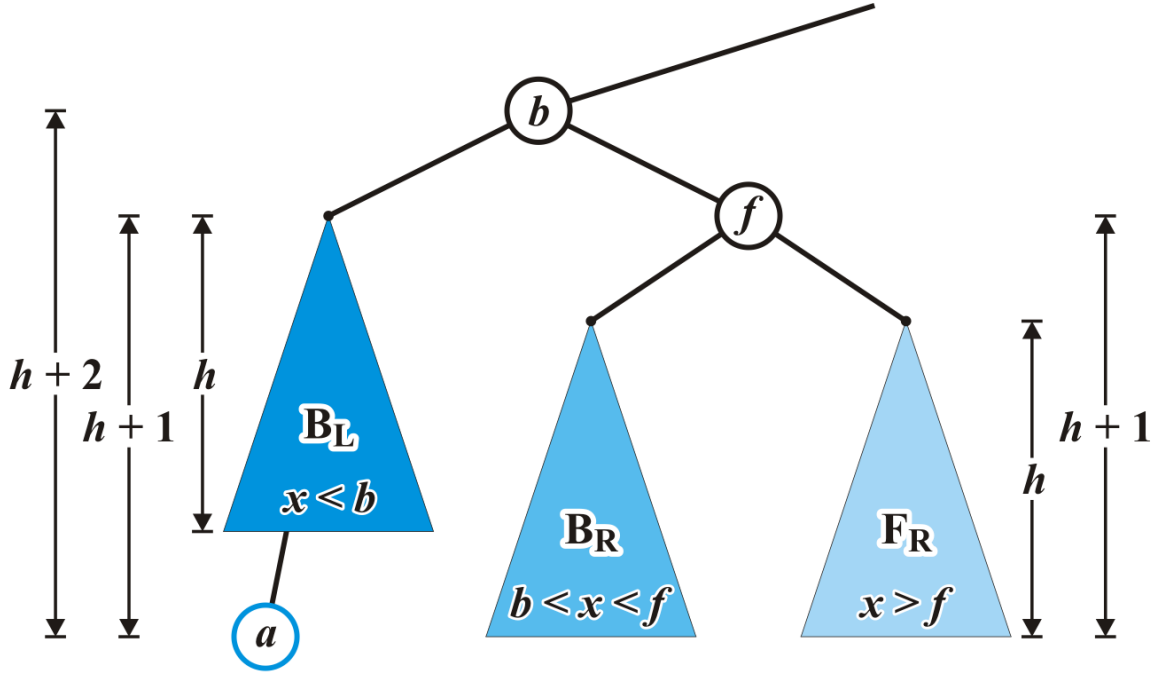
Assign the address of the tree $\mathbf{B_R}$ to `left_tree` of node ***f***

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



Maintaining Balance: Case 1

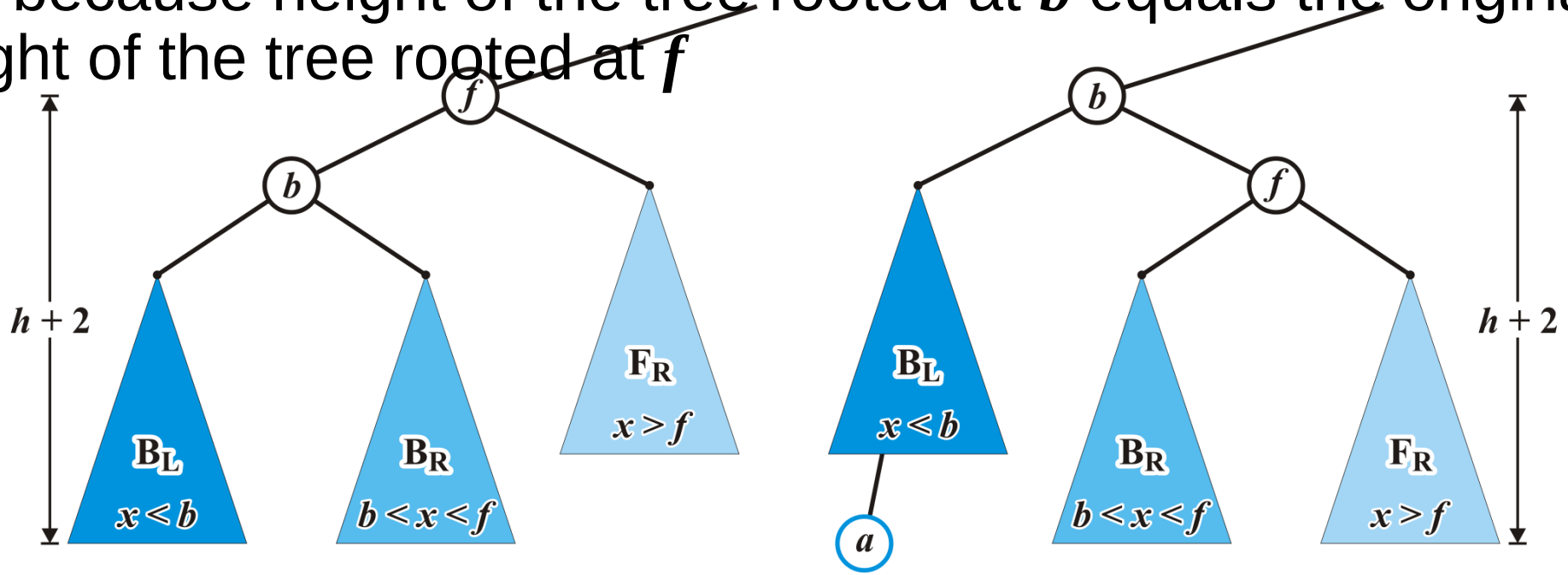
The nodes ***b*** and ***f*** are now balanced and all remaining nodes of the subtrees are in their correct positions



Maintaining Balance: Case 1

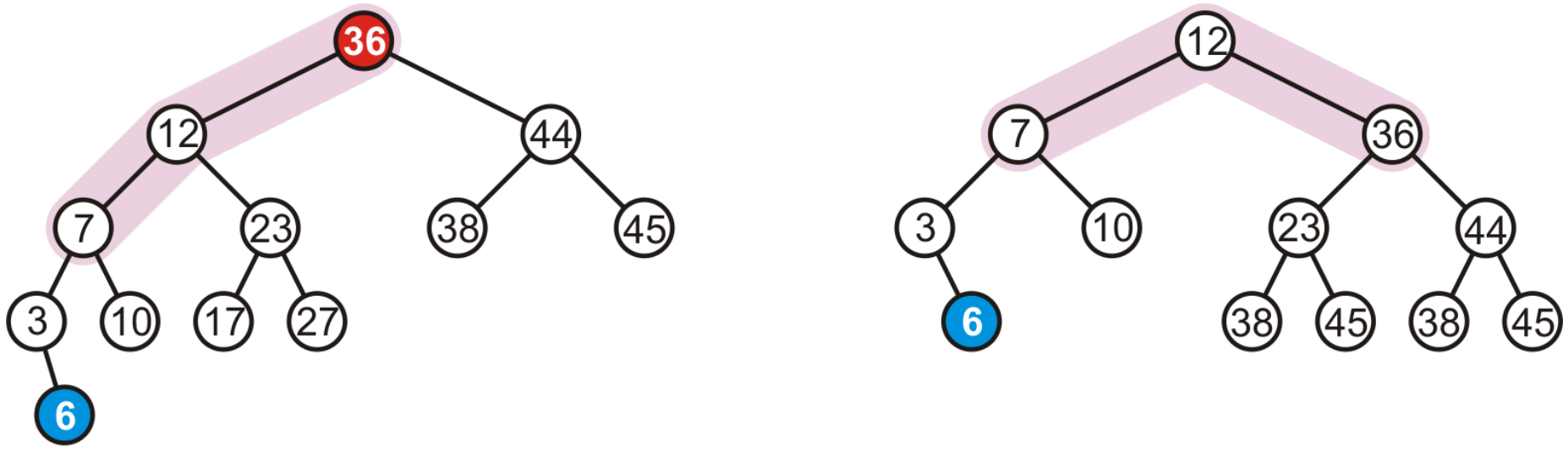
Q: will this insertion affect the balance of any ancestors all the way back to the root?

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



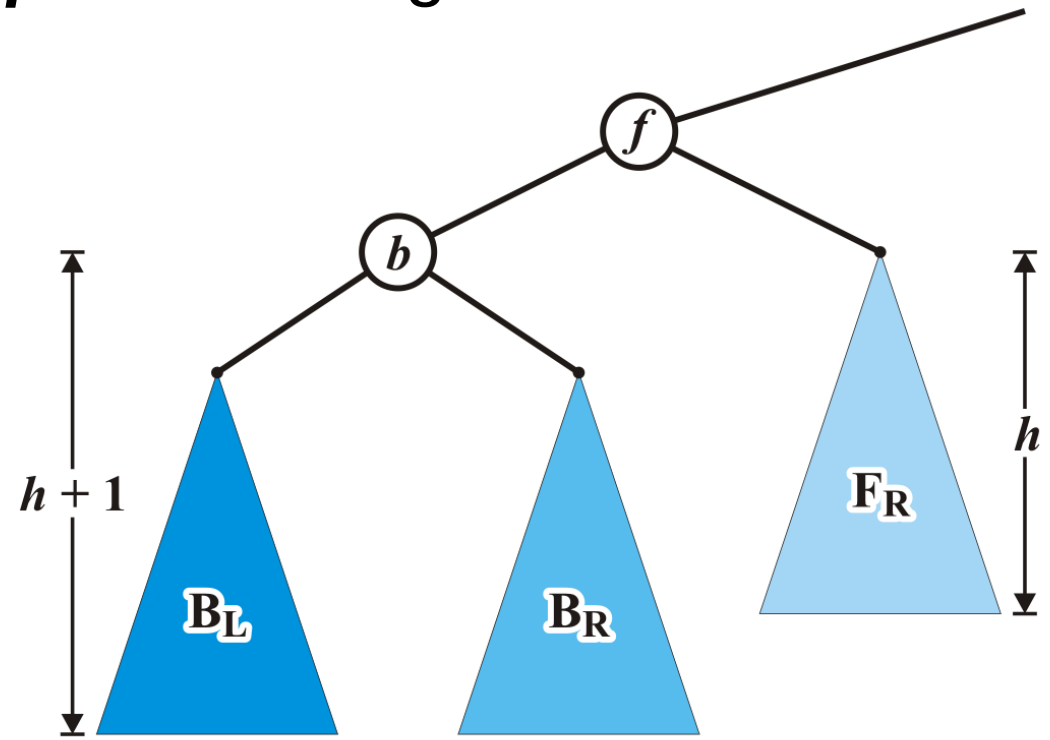
Maintaining Balance: Case 1

In our example case, the correction



Maintaining Balance: Case 2

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

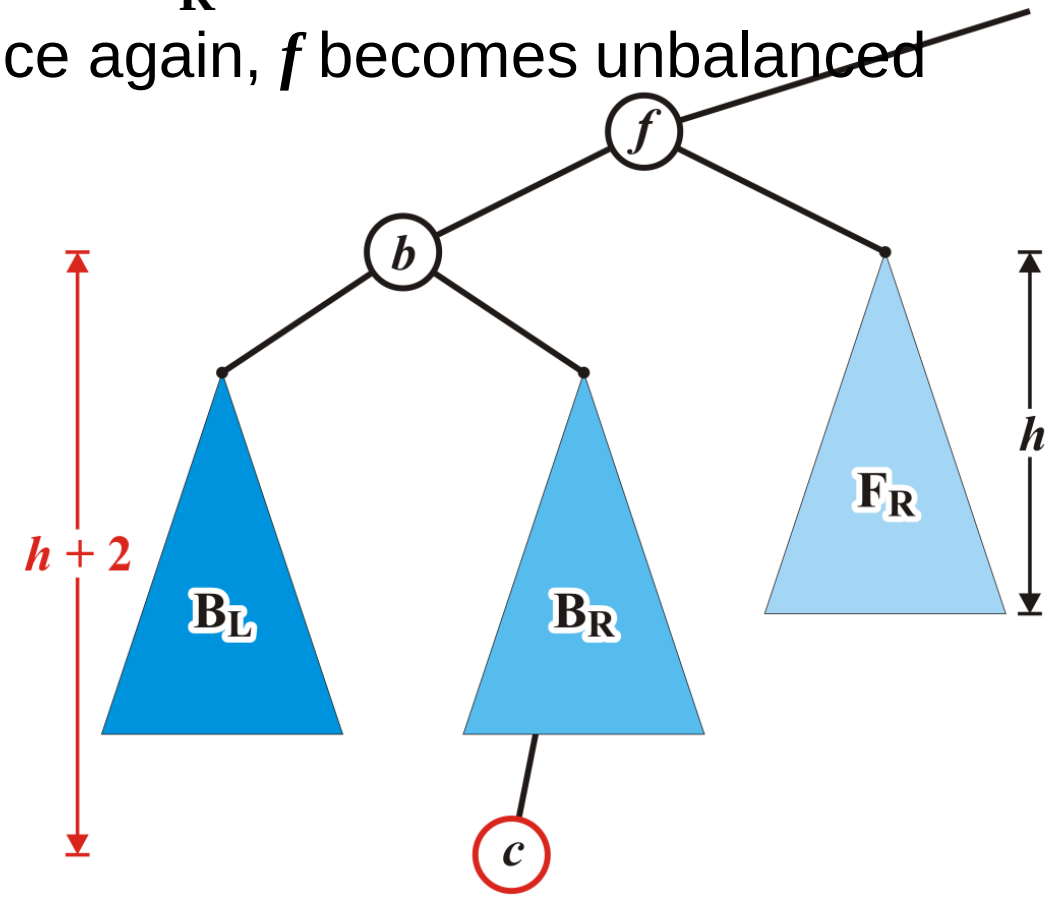


Maintaining Balance: Case 2

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

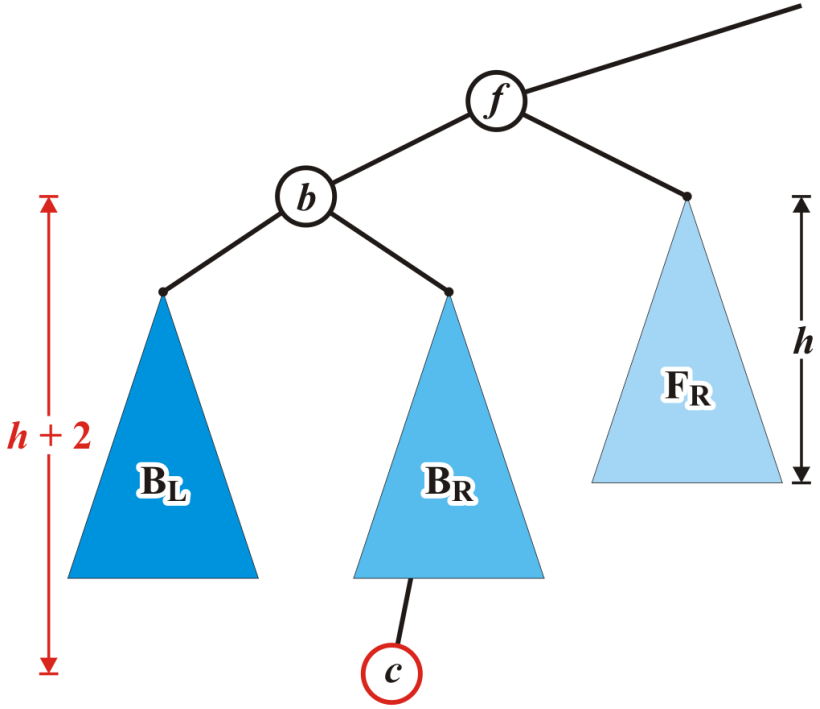
left-right

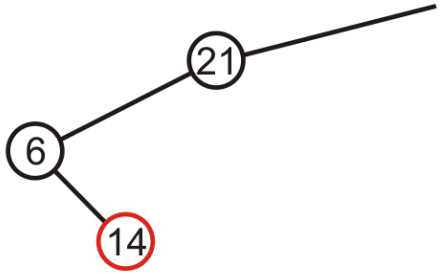
- Once again, f becomes unbalanced

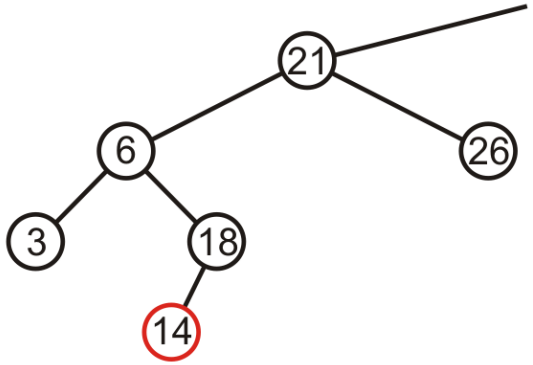


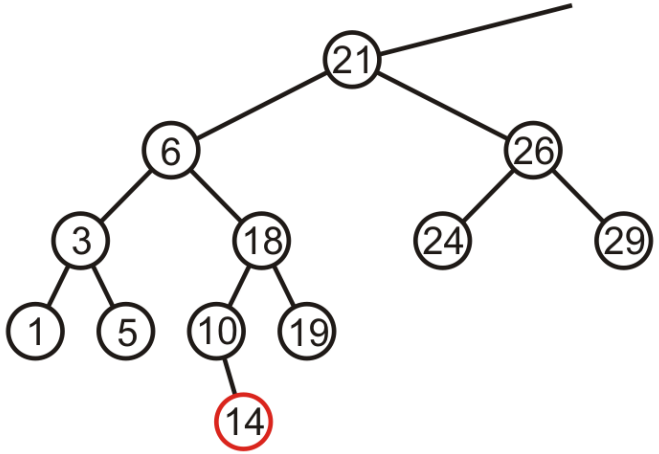
Maintaining Balance: Case 2

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



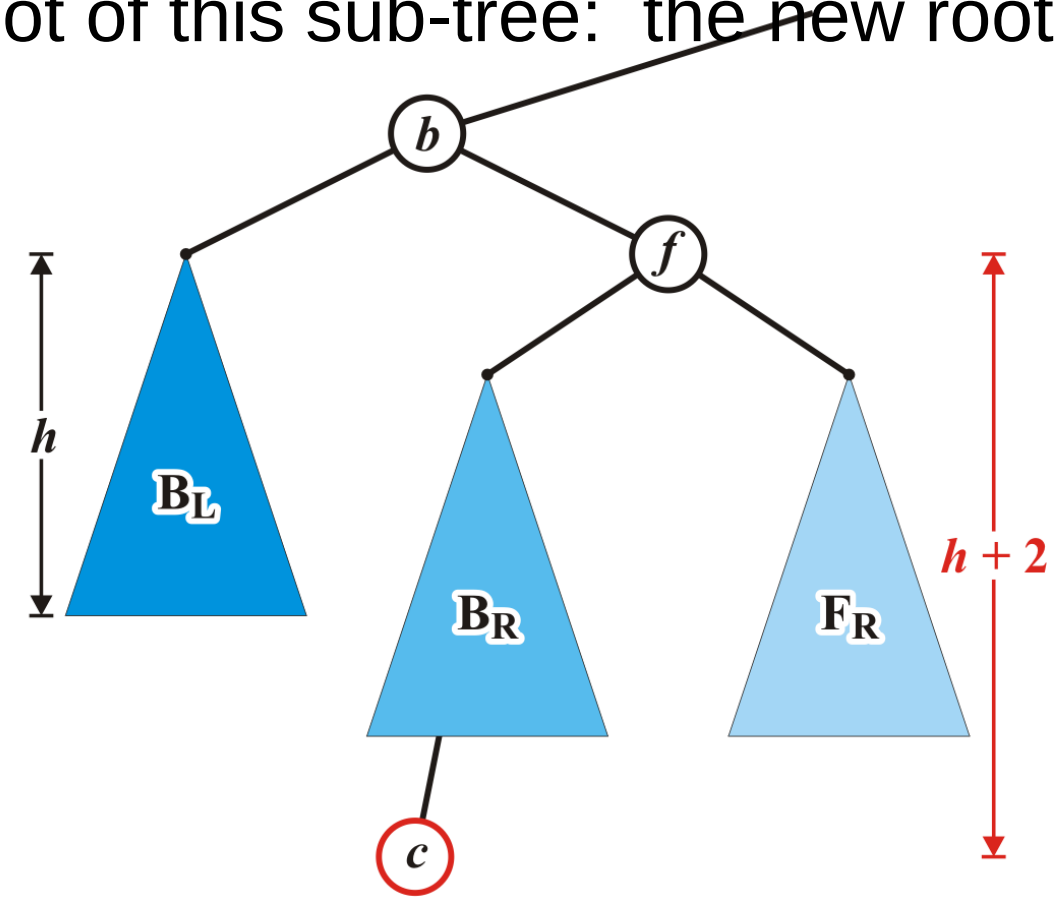






Maintaining Balance: Case 2

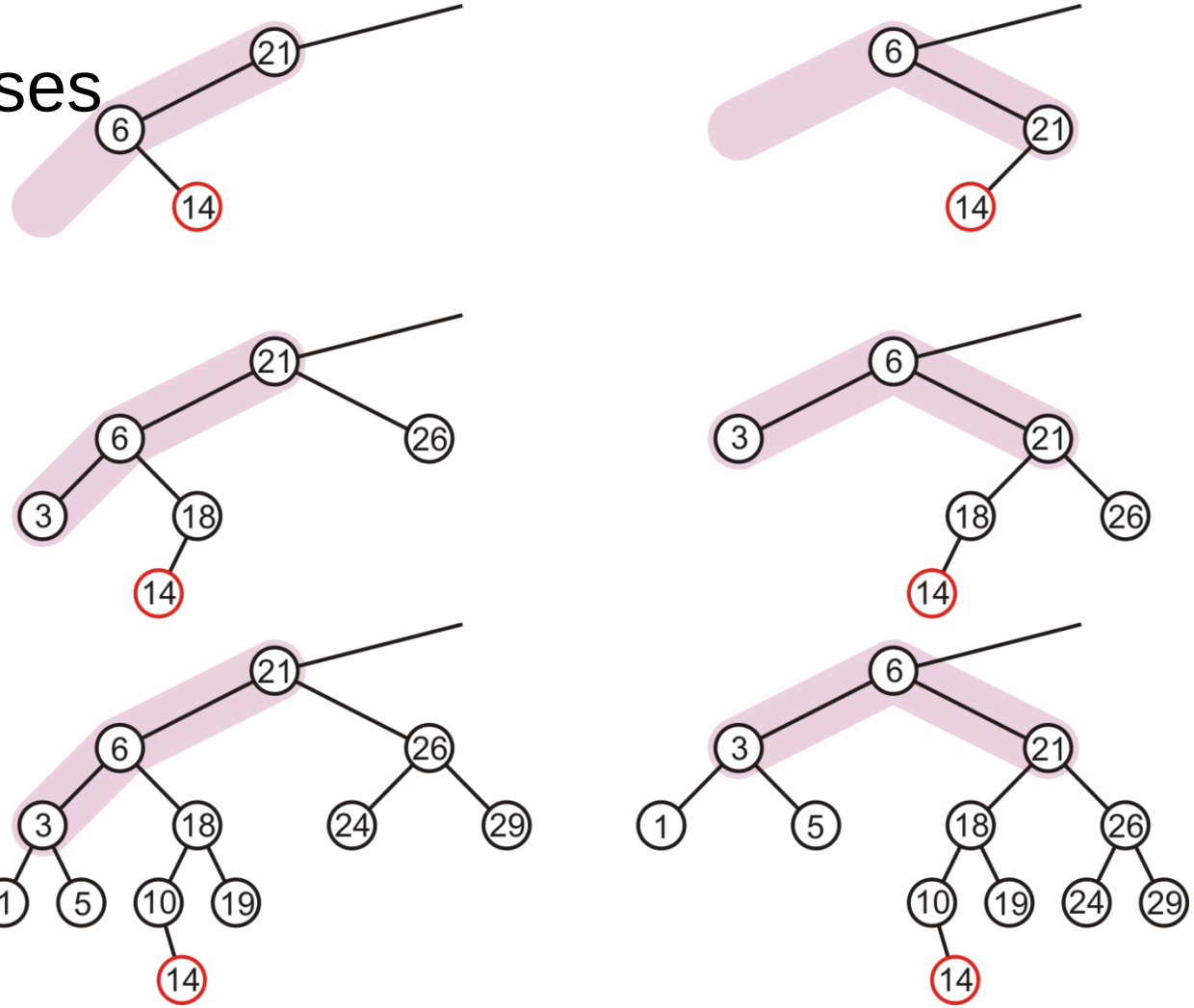
Unfortunately, the previous correction does not fix the imbalance at the root of this sub-tree: the new root, ***b***, remains unbalanced



Maintaining Balance: Case 2

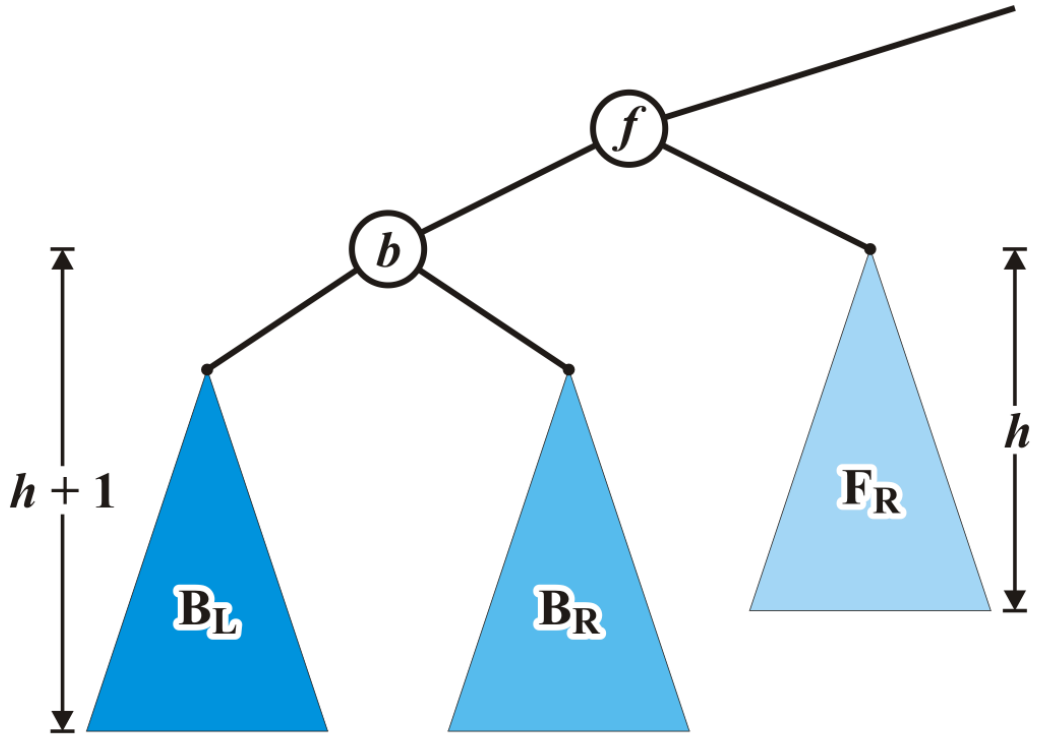
In our three sample cases with $h = -1, 0$, and 1, doing the same thing as before results in a tree that is still unbalanced…

- The imbalance is just shifted to the other side



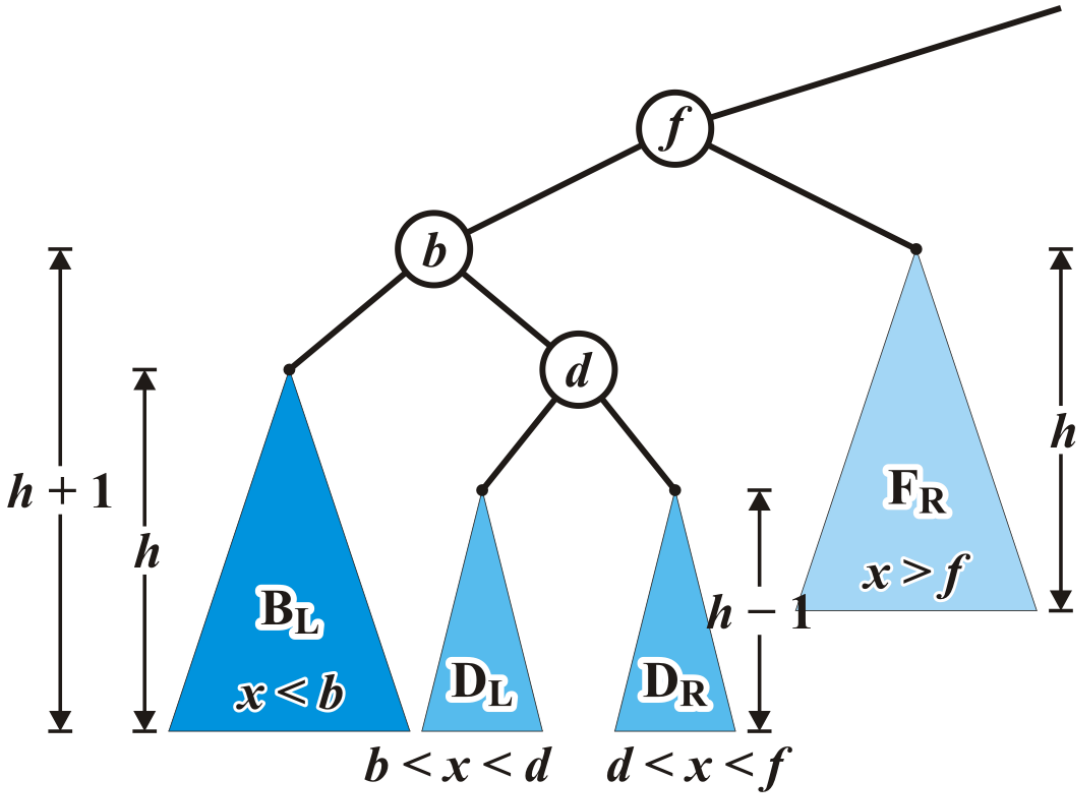
Maintaining Balance: Case 2

We need to look into B_R



Maintaining Balance: Case 2

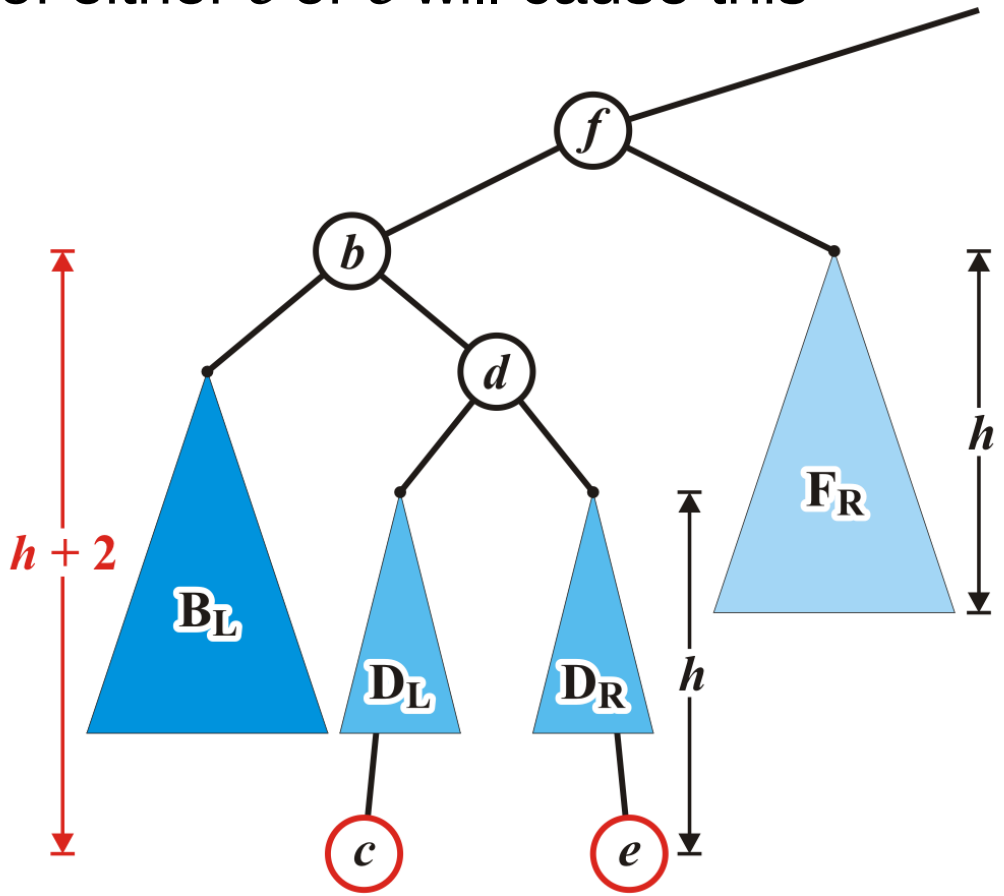
Re-label B_R as a tree rooted at d with two subtrees of height $h - 1$



Maintaining Balance: Case 2

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

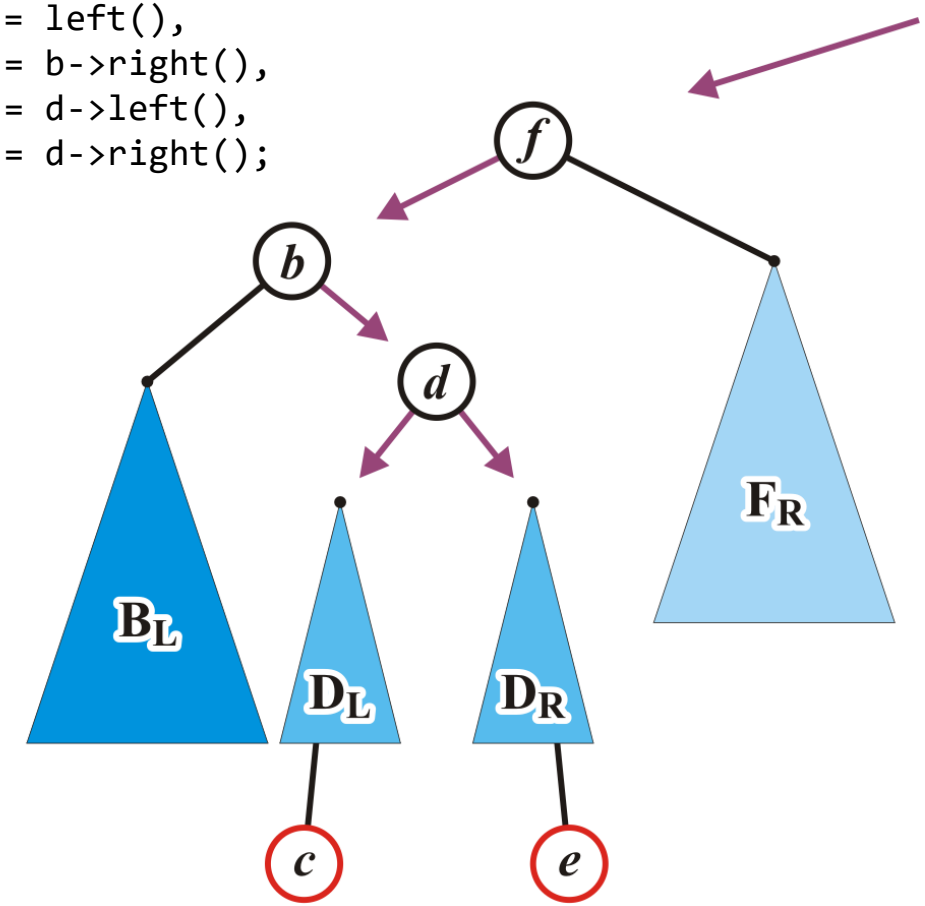
- The addition of either ***c*** or ***e*** will cause this



Maintaining Balance: Case 2

We will reassign the following pointers

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

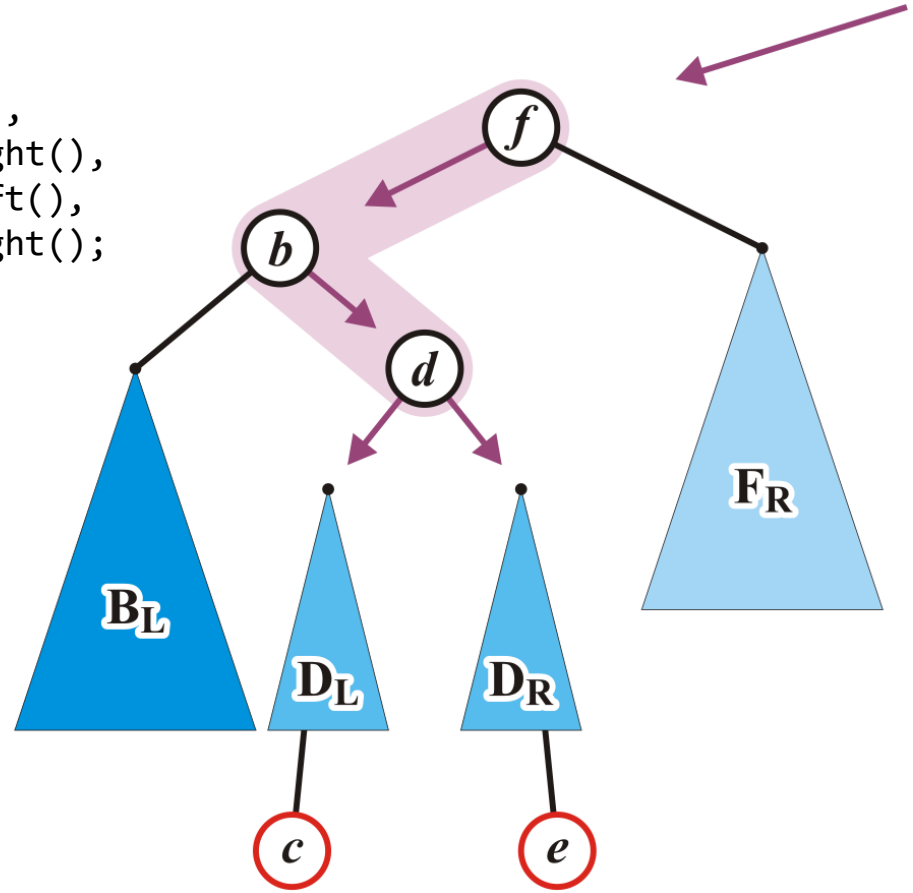


Maintaining Balance: Case 2

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

- Recall the

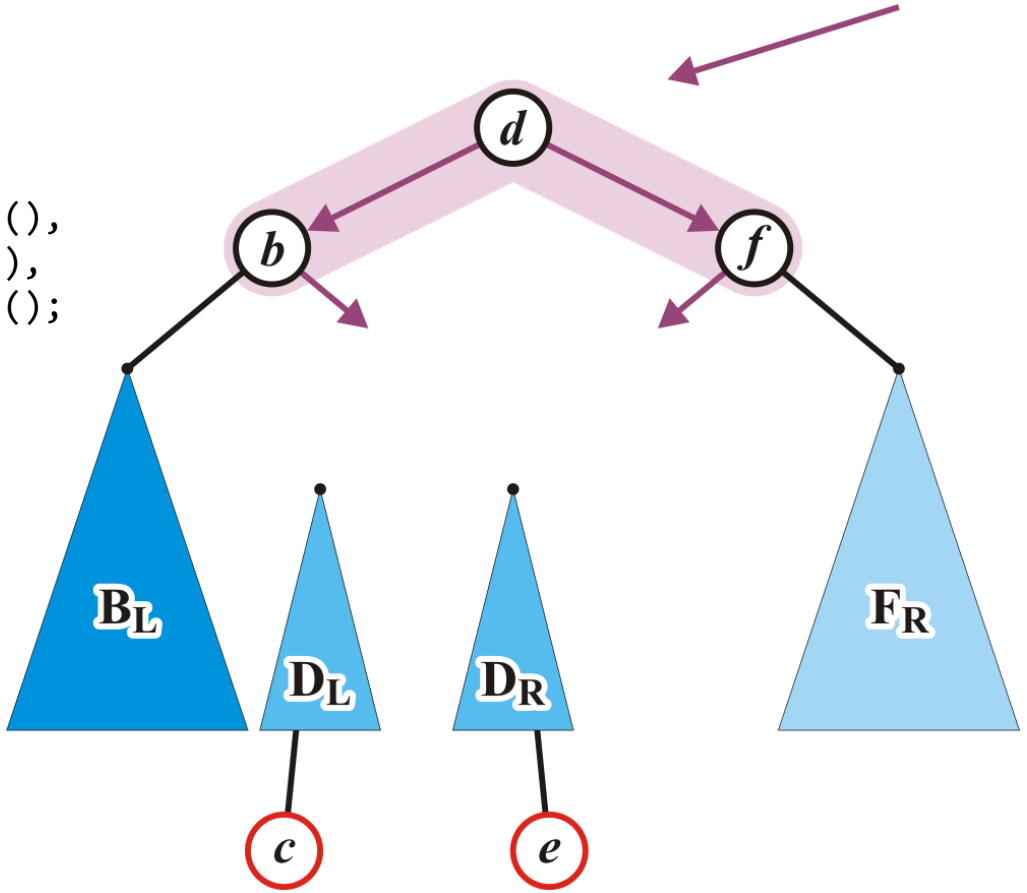
```
AVL_node<Type> *b  = left(),
               *d  = b->right(),
               *DL = d->left(),
               *DR = d->right();
```



Maintaining Balance: Case 2

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

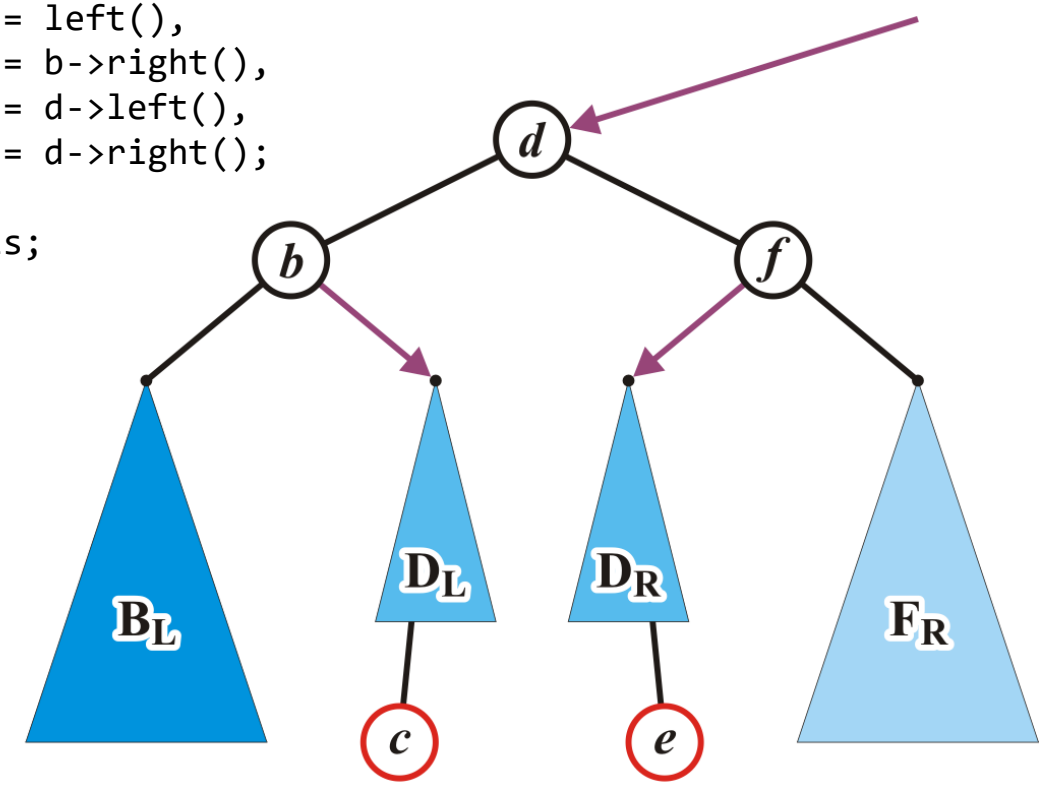
```
AVL_node<Type> *b  = left(),
               *d  = b->right(),
               *DL = d->left(),
               *DR = d->right();

   d->left_tree  = b;
   d->right_tree = this;
```



Maintaining Balance: Case 2

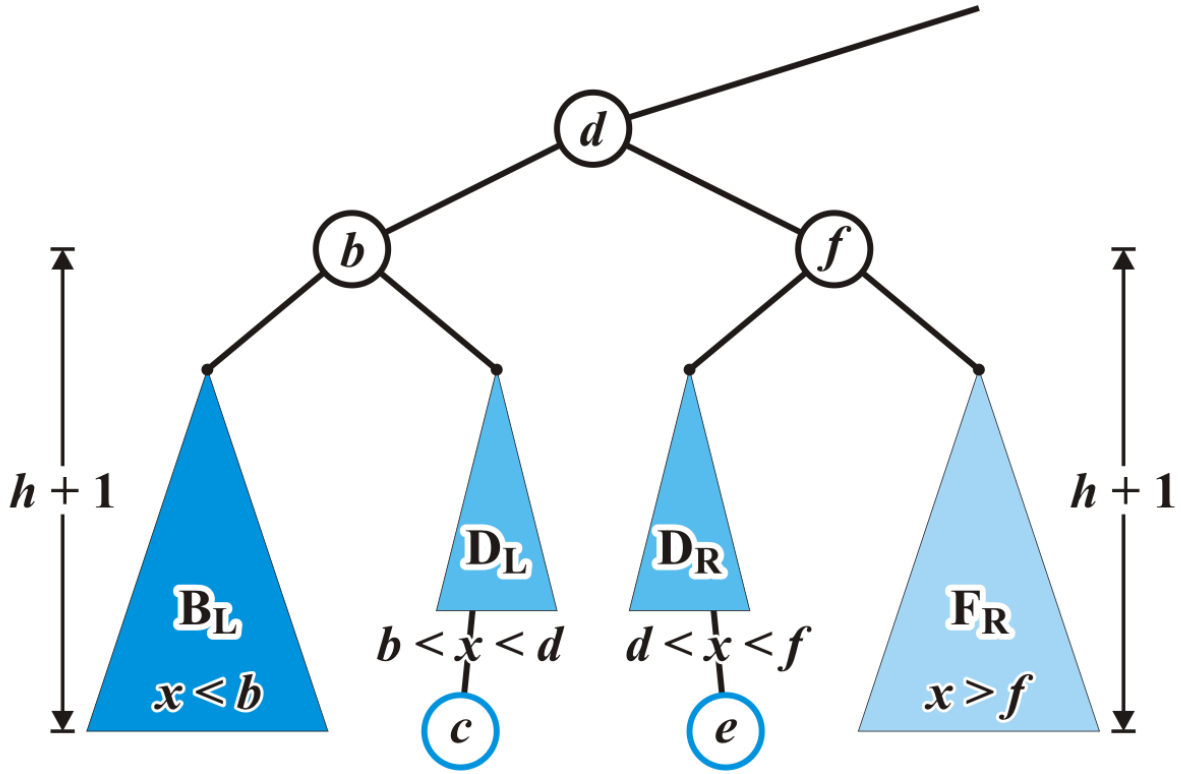
We also have to connect the two subtrees and original parent of ***f***

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



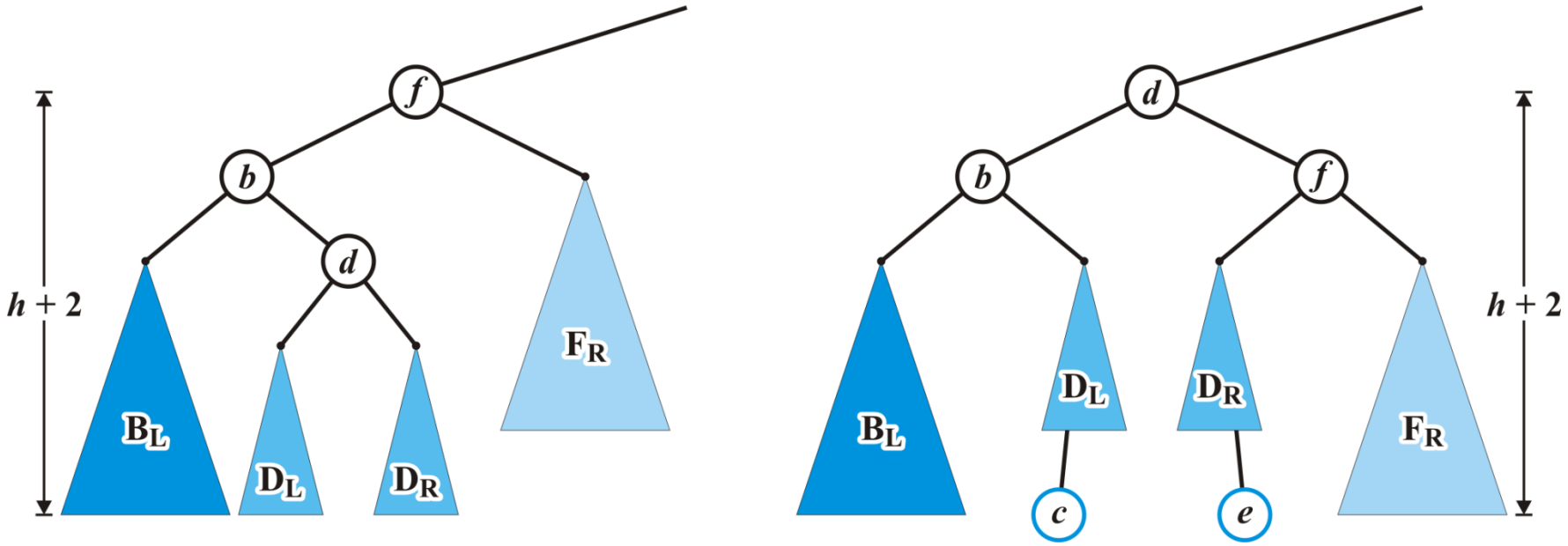
Maintaining Balance: Case 2

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



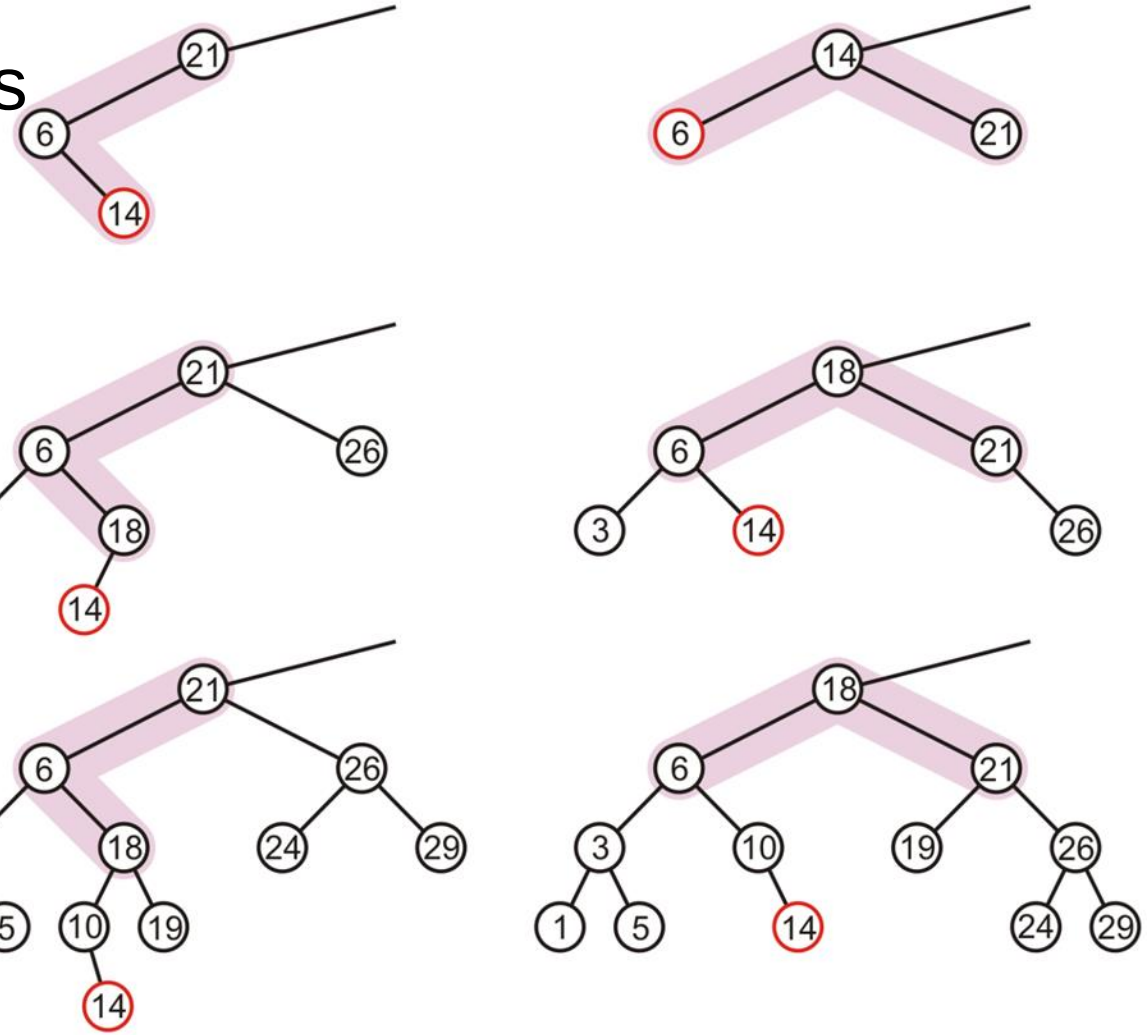
Maintaining Balance: Case 2

Again, the height of the root did not change



Maintaining Balance: Case 2

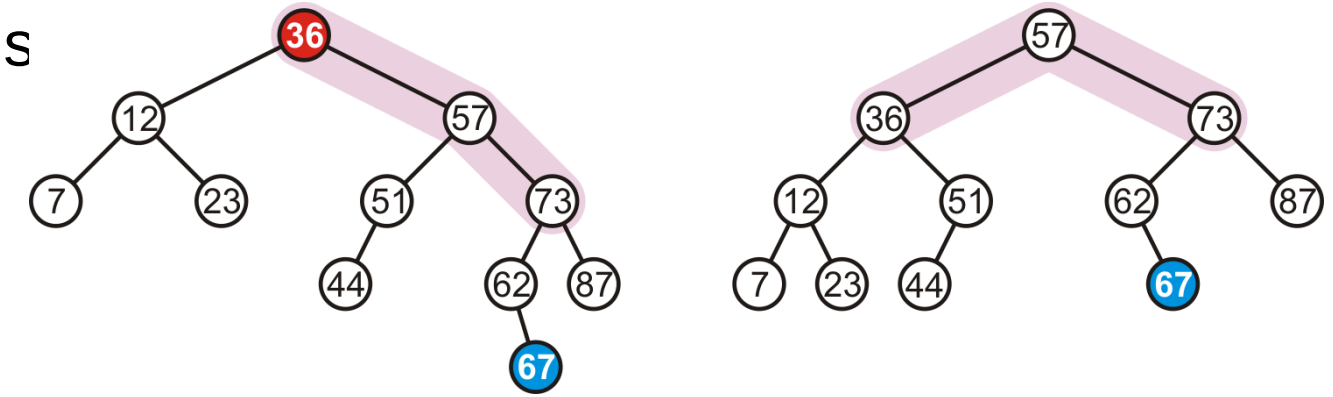
In our three sample cases with $h = -1, 0$, and 1, the node is now balanced and the same height as the tree before the insertion



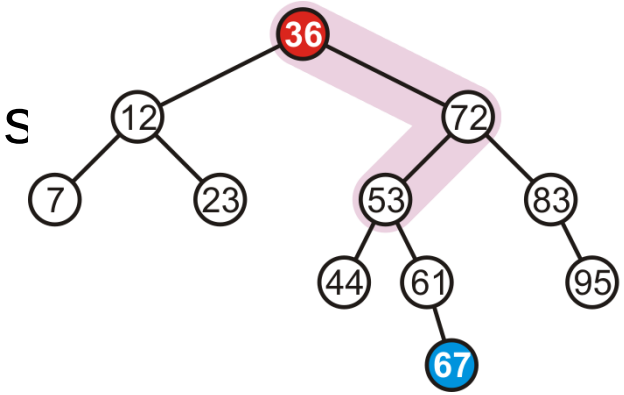
Maintaining balance: symmetric cases

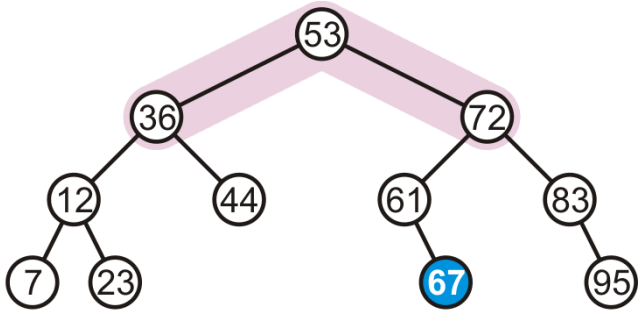
There are two symmetric cases to those we have examined:

- Insertions



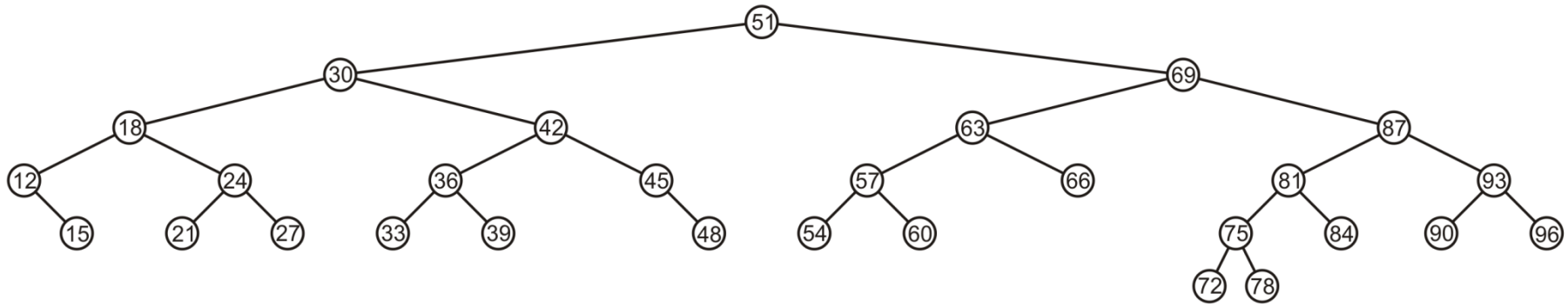
- Insertions





Insertion

Consider this AVL tree



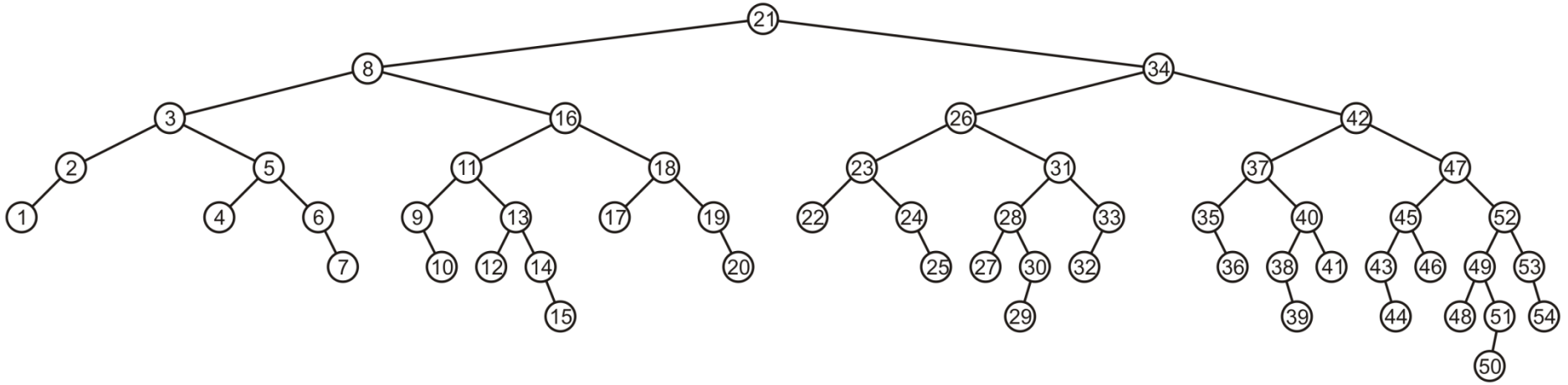
Erase

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



Time complexity

Insertion

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

Erasion

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

AVL Trees as Arrays?

We previously saw that:

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

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

AVL Trees as Arrays?

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

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

Such a tree requires an array of size

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

We can rewrite this as

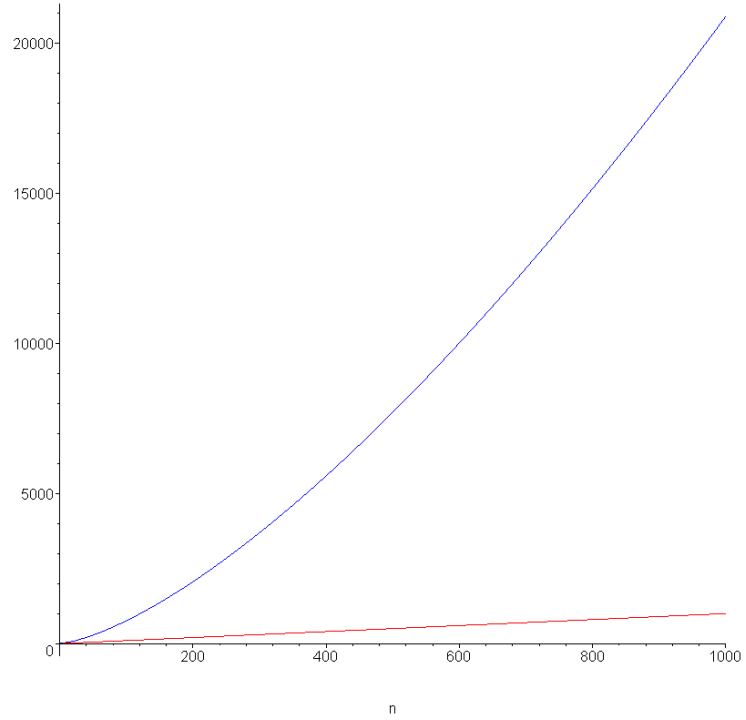
$$2^{-0.3277}\, n^{\log_\phi(2)} \approx 0.7968\, n^{1.44}$$

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

AVL Trees as Arrays?

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

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



Huffman Tree

The Basic Idea

- Not all characters occur with the same frequency!
- Yet all characters are allocated the same amount of space
  - 1 char = 1 byte, be it e or x

- Idea: tailoring codes to frequency of characters
  - Use fewer bits to represent frequent characters
  - Use more bits to represent infrequent characters

Example

Symbol	A	B	C	D
Frequency	12.5%	25%	50%	12.5%
Original Encoding	00	01	10	11
	2 bits	2 bits	2 bits	2 bits
Huffman Encoding	110	10	0	111
	3 bits	2 bits	1 bit	3 bits

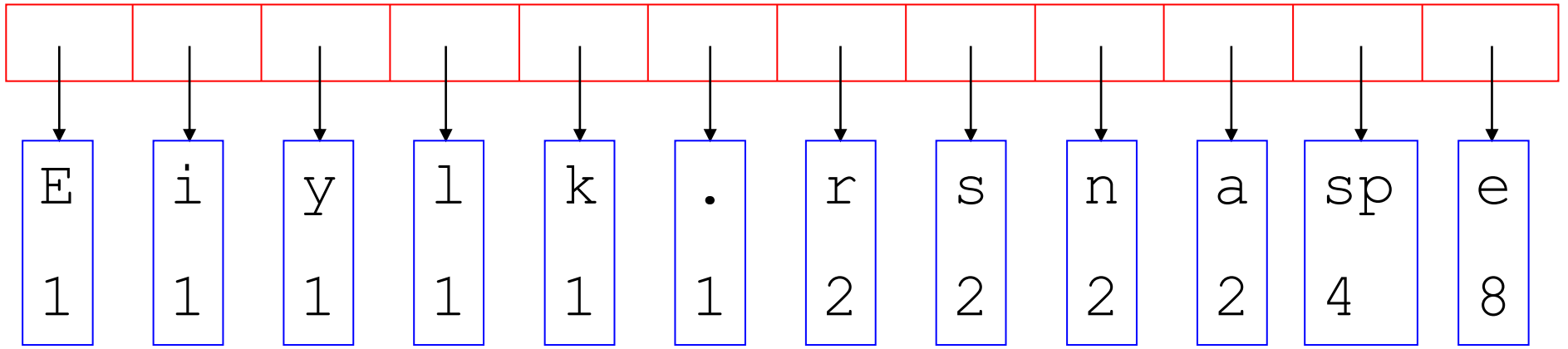
- Expected size
  - Original $\Rightarrow$ 1/8×2 + 1/4×2 + 1/2×2 + 1/8×2 = 2 bits / symbol
  - Huffman $\Rightarrow$ 1/8×3 + 1/4×2 + 1/2×1 + 1/8×3 = 1.75 bits / symbol

Algorithm

1. Scan text to be compressed and count frequencies of all characters.
2. Prioritize characters based on their frequencies in text.
3. Build Huffman code tree based on prioritized list.
4. Perform a traversal of tree to determine all code words.
5. Encode the text using the Huffman codes.

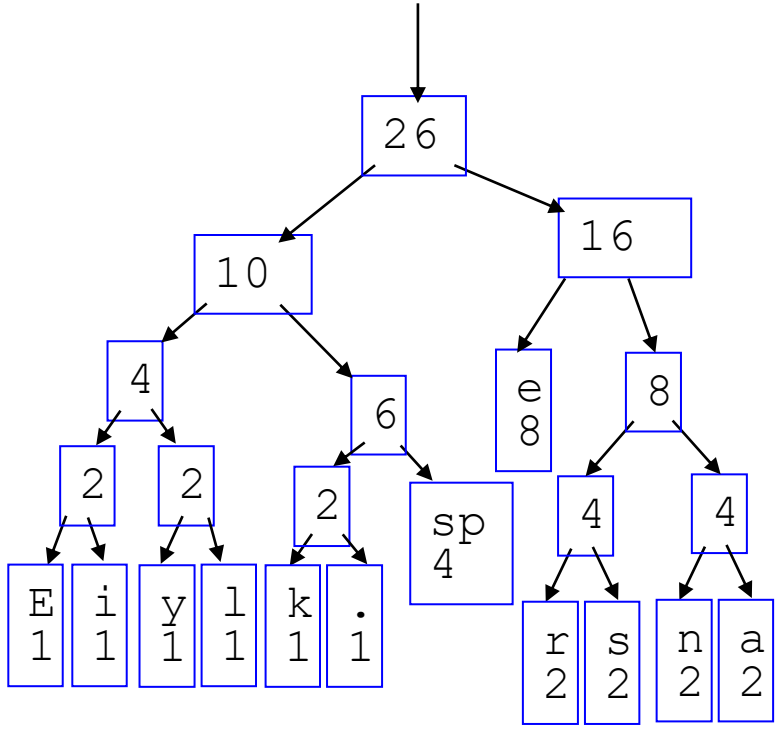
Prioritize characters

- The priority queue after inserting all nodes



Traverse Tree for Codes

Char	Code
E	0000
i	0001
y	0010
l	0011
k	0100
.	0101
space	011
e	10
r	1100
s	1101
n	1110
a	1111



Encoding the File

- Have we made things any better?
  - 73 bits to encode the text
  - ASCII would take 8 * 26 = 208 bits

Eerie eyes seen near lake.

```
00001011000001100111000101011011010011111010111100011111010100100101
```

Char	Code
E	0000
i	0001
y	0010
l	0011
k	0100
.	0101
space	011
e	10
r	1100
s	1101
n	1110
a	1111

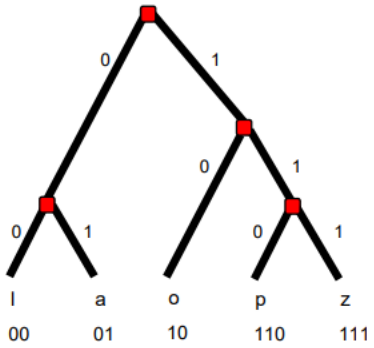
Property

- Lossless
- Prefix-free
- Uniquely decodable
- Optimal

Property

Huffman code is prefix-free

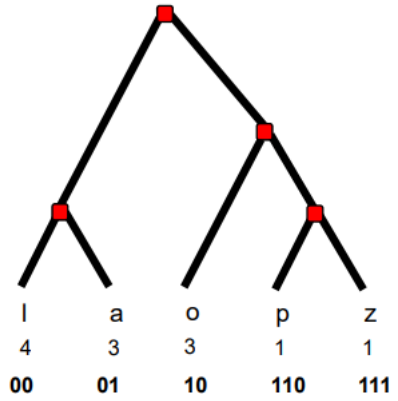
- Any two codewords correspond to paths from the root to leaves.
  - The 2 paths split from each other somewhere.
  - After the split, neither codeword is a prefix of the other.
- Huffman codes are uniquely decodable.



Property

Huffman code is optimal

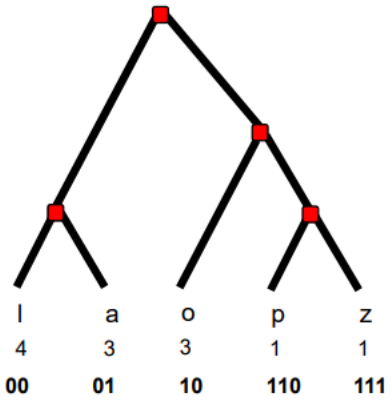
- Huffman encoding gives the shortest uniform encoding of strings.
  - Uniform basically means you can't change your encoding method for different strings.
- Call an encoding in which codewords are derived from paths in trees a tree code.
- Fact There exists tree codes that are optimal.
- Since the Huffman code is a tree code, to prove Huffman is optimal, we just need to prove it's an optimal tree code.



Property

Huffman code is optimal

- Def The cost of a tree code is $cost(T) = \Sigma_{v \in T}\, d(v) \cdot f(v)$, where $d(v)$ denotes the depth of letter v, and $f(v)$ denotes its frequency.
  - $cost(T)$ = number of bits to represent original string.
- Claim 1 In an optimal tree code, every leaf has a sibling.
- Proof Otherwise, replace the lone leaf by its parent to get a tree code with lower cost.

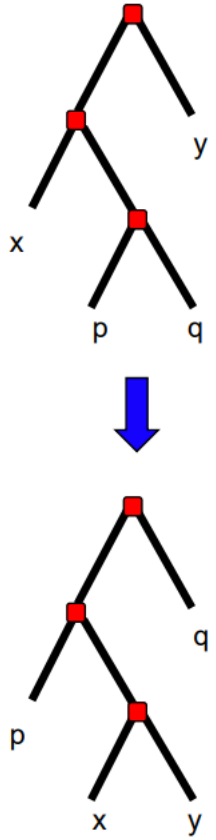


cost(T) = 4*2+3*2+3*2+
3*1+3*1=26

Property

Huffman code is optimal

- Claim 2 Consider the two least frequent letters x and y. In the an optimal tree code T, x and y are siblings of each other at the max depth of the tree.
- Proof Suppose not, and let p be a node at the max depth.
  - p has a sibling q by Claim 1.
  - p and q have higher frequency than x and y, resp.
  - Create a new tree where we swap p and x, and q and y.
  - The new tree has strictly lower cost than T, since p, q have higher frequency than x, y. Contradiction.



Property

Huffman code is optimal

- Thm Huffman code is optimal.
- Proof Use induction on number of letters in the code. Suppose it's true up to $n-1$.
  - Consider a code with n letters. Let x,y be the letters with the lowest frequency.
  - Let T be the Huffman code tree on the n letters.
    - Create a new node z with frequency $f(z) = f(x) + f(y)$.
    - Let S be a tree formed from T by removing x and y, and replacing their parent by z.
  - S is the Huffman code on the $n-1$ letters.
    - Because of the recursive way Huffman encoding works.
  - S is an optimal tree code on the $n-1$ letters, by induction.
  - $cost(T) = cost(S) + f(x) + f(y)$.
    - All the nodes in S and T are the same, except x,y,z.
    - $cost(z) = d(z) \cdot f(z) = d(z) \cdot (f(x) + f(y))$.
    - $d(z) = d(x) - 1 = d(y) - 1$.
    - $cost(x) + cost(y) = cost(z) + f(x) + f(y)$.

Property

Huffman code is optimal

- Proof (continued)
  - Let T' be an optimal tree code on the n letters.
    - By Claim 2, x and y are siblings in T'.
    - Merge them into a node z', with $f(z') = f(x) + f(y)$. Form a tree S' by removing x and y from T', and replacing their parent by z'.
    - S' is a tree code on $n - 1$ letters.
  - $cost(T') = cost(S') + f(x) + f(y) \geq cost(S) + f(x) + f(y) = cost(T)$.
    - First equality because x,y at depth one greater than z.
    - First inequality because S is opt tree code on $n - 1$ letters.
  - So, the tree T produced by Huffman encoding is optimal.

历年真题

2. In a character-coding problem, if a file contains only characters ‘a’, ‘b’, and ‘c’ with frequencies 45, 13, and 12 respectively, which of the following codewords is an optimal character code for the file? （ ）

   在一个字符编码问题中，如果一个文件只由 a, b, c 组成，它们出现的频率分别是 45, 13, 12，以下哪些编码是最优的？（ ）

   A. a:000, b:010, c:101

   B. a:000, b:010, c:100

   C. a:00, b:01, c:10

   D. a:1, b:01, c:00

历年真题

在一个只含有字符 a,b,c,d,e,f 的文本中，a,b,c,d,e,f 的使用频次分别是 23, 24, 25, 26, 27, 28，根据这些统计信息，可以针对该文本生成 Huffman（哈夫曼）编码。请写出一组针对字符串 abcdef 的霍夫曼编码，以及必要的推导过程。

历年真题

5. AVL tree (25 points) 二叉平衡树（25 分）

Given an initially empty AVL tree, insert the sequence of integers 15, 20, 23, 10, 13, 7, 30, 25 from left to right into the AVL tree. If there is an imbalance after you insert an integer, you should restore the balance of AVL tree before you insert the next node.

将一个的整数序列 15, 20, 23, 10, 13, 7, 30, 25 按从左到右的顺序依次插入到一个初始状态为空的二叉平衡树（AVL tree）。在插入一个整数之后，如果破坏了当前状态下树的平衡，在插入下一个节点之前需要将树重新调整至平衡。

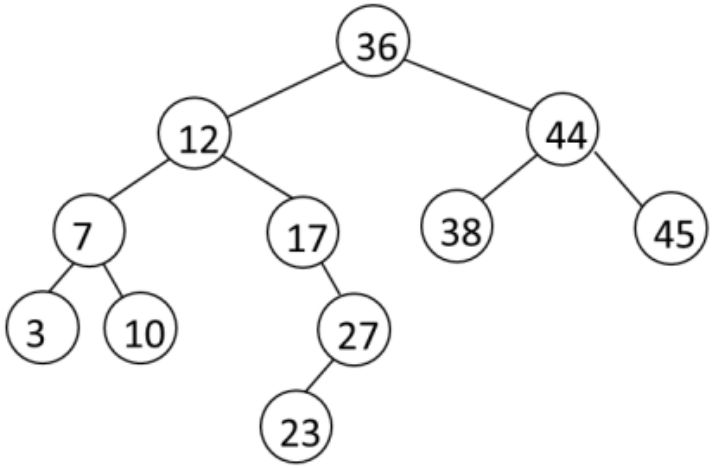
1) Draw the final tree after all these insertions. (10 points)
画出将所有整数插入之后所建立的平衡树。（10 分）

2) Write the sequence of integers resulted from pre-order traversal of the final AVL tree built in 1). (5 points)
写出对题 1）中所建立的平衡树进行先序遍历所得到的整数序列。（5 分）

3) Continue with the final AVL tree in 1), erase the node with integer 23 from the AVL tree. Draw the resulting AVL tree after the deletion. (10 points)
将 1）中得到的平衡树中 23 所在位置的节点删除，画出删除节点后得到的新的平衡树。（10 分）

历年真题

一个二叉搜索树（Binary Search Tree）的初始状态如图-2 所示：



5) 考虑树的 AVL 平衡特性，即任意节点左右两边子树的高度差不大于 1，在初始树中对 23 这个节点导致的非平衡性进行调整，画出调整后的 AVL 树。（2 分）
6) 在问题 5) 获得的树的基础上，同样考虑 AVL 平衡特性，画出插入元素 2 后的树。（4 分）
7) 在 AVL 树中，假设节点数量为 n，那么可以证明整棵树的高度 h 在最坏情况下为 $\lfloor 1.44*\lg(n+1)-1.33 \rfloor <= h <= \lfloor 1.44*\lg(n+1)-1.33 \rfloor +1$，请写出最好情况下树的高度，即节点数量为 n 时 AVL 树的最小树高。（2 分）
8) 根据问题 7)，简要分析 AVL 树是否适合使用数组形式存储（提示：二叉树的数组存储类似二叉堆中在数组中的存储方式，即节点的父子关系可以直接通过数组下标计算得出）（4 分）

Q&A