Data Structures and Algorithms

(CS210A)

Semester I – **2014-15**

Lecture 17:

Height balanced BST

- **Red-black trees - II**

Red **Black** Tree

Red Black tree:

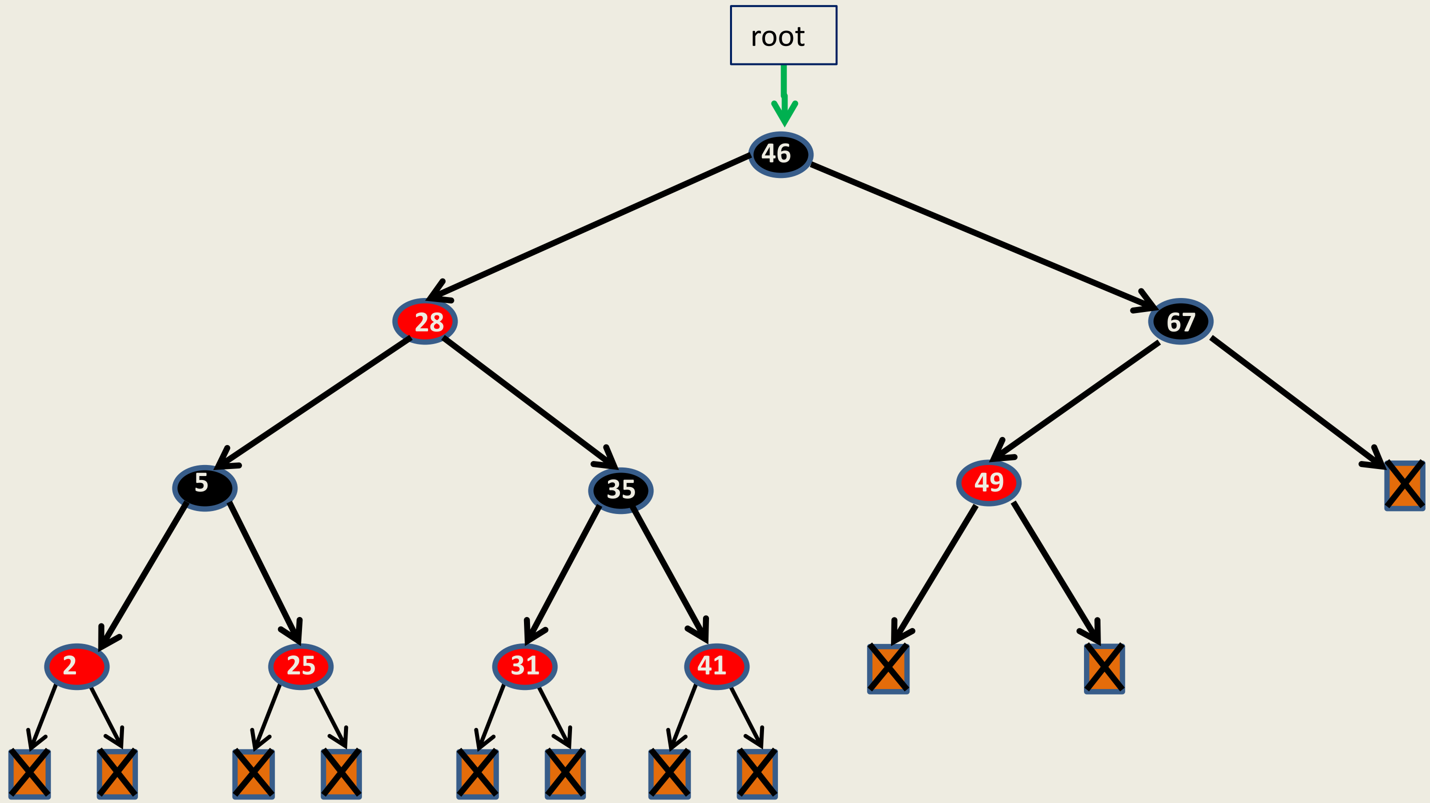
a **full** binary search tree with each leaf as a **null** node

and satisfying the following properties.

- Each node is colored **red** or **black.**
- Each leaf is colored **black** and so is the root.
- Every **red** node will have both its children **black.**
- No. of **black** nodes on a path from root to each leaf node is same.

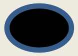
black height

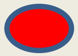
A red-black tree

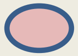


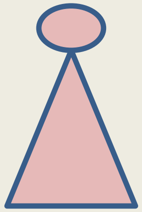
Handling Deletion in a Red Black Tree

Notations to be used

 a **black** node

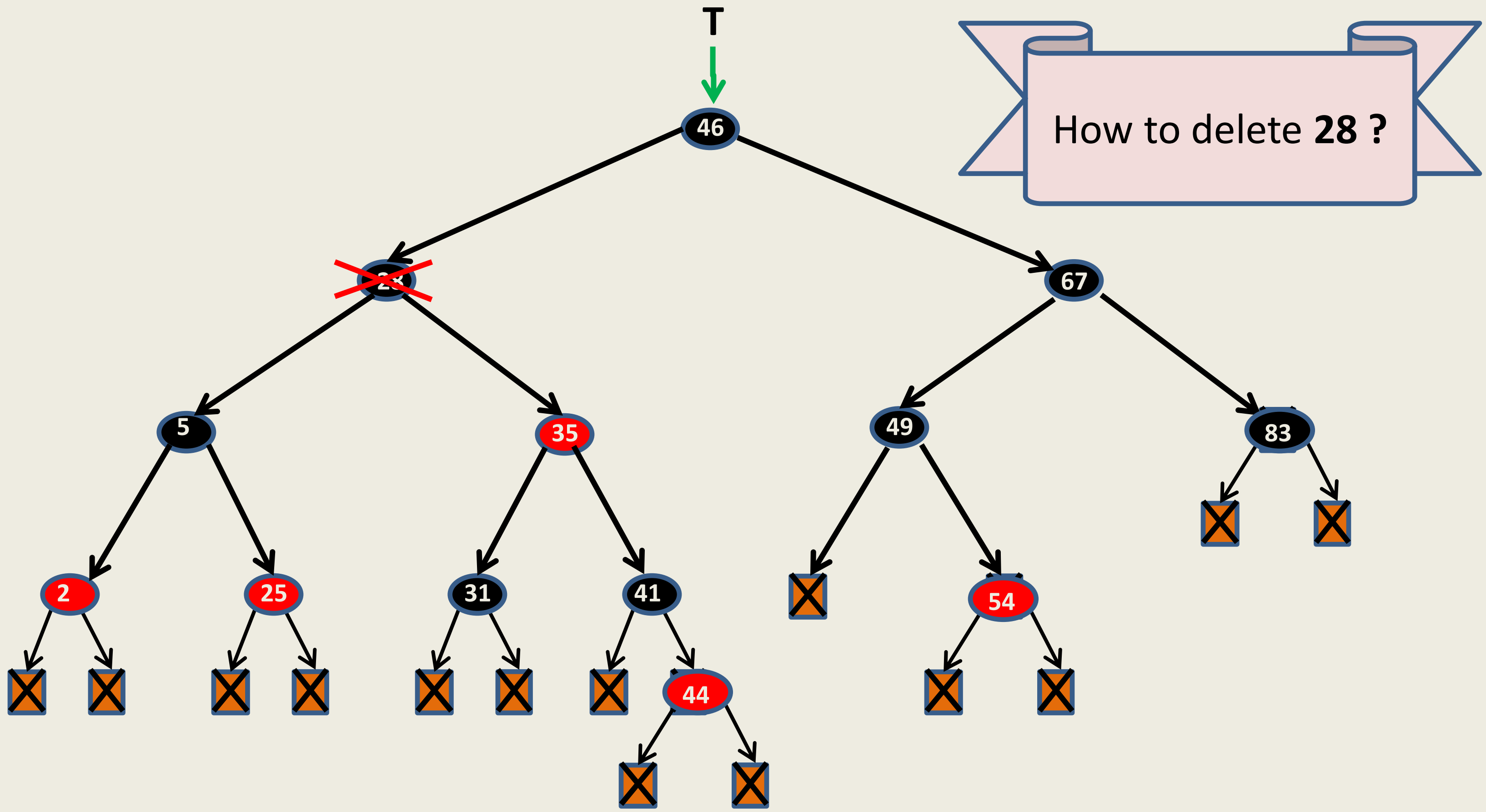
 a **red** node

 a node whose color is not specified

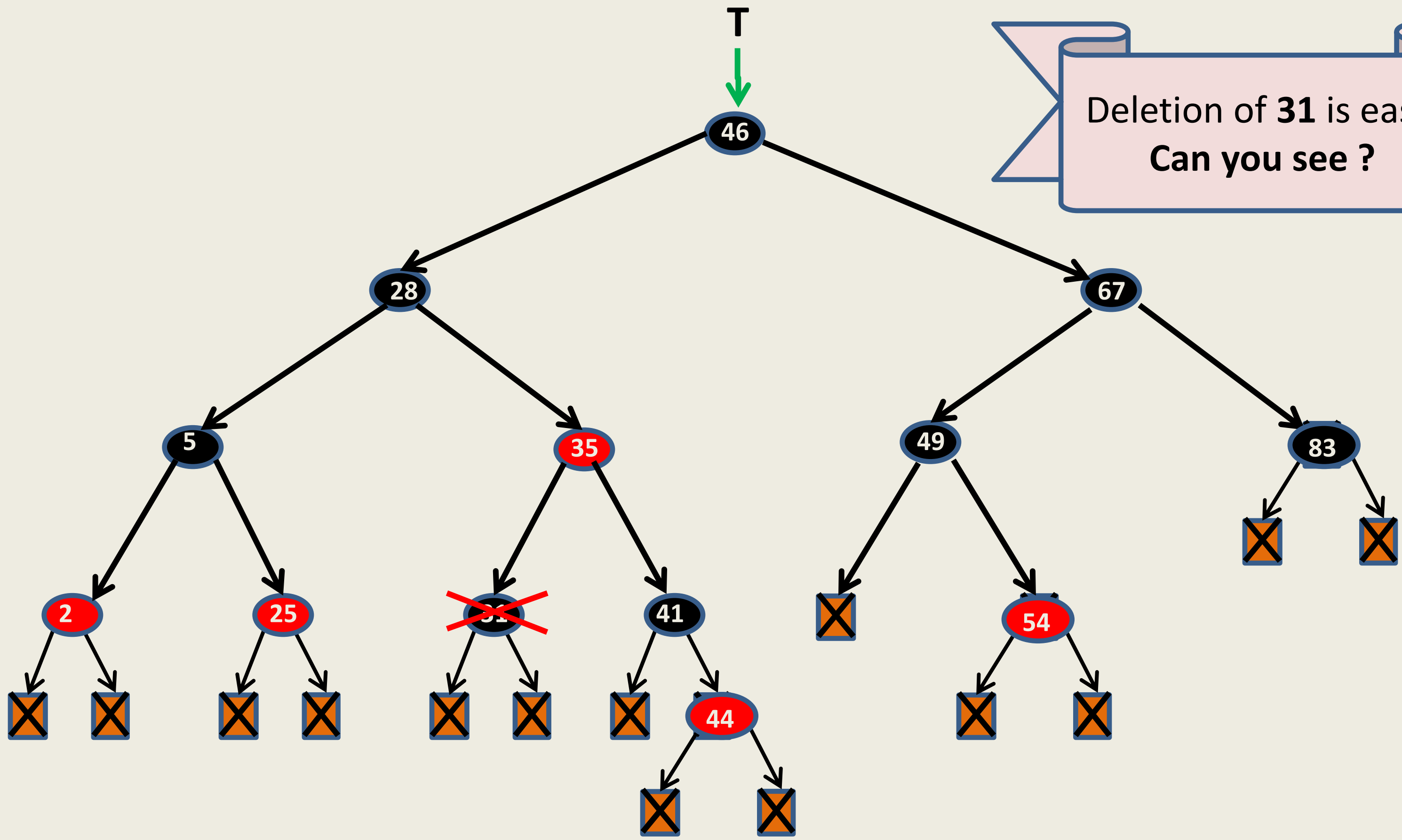
 a BST  Could potentially be 

Deletion in a BST is **slightly harder than Insertion**

(even if we ignore the **height** factor)

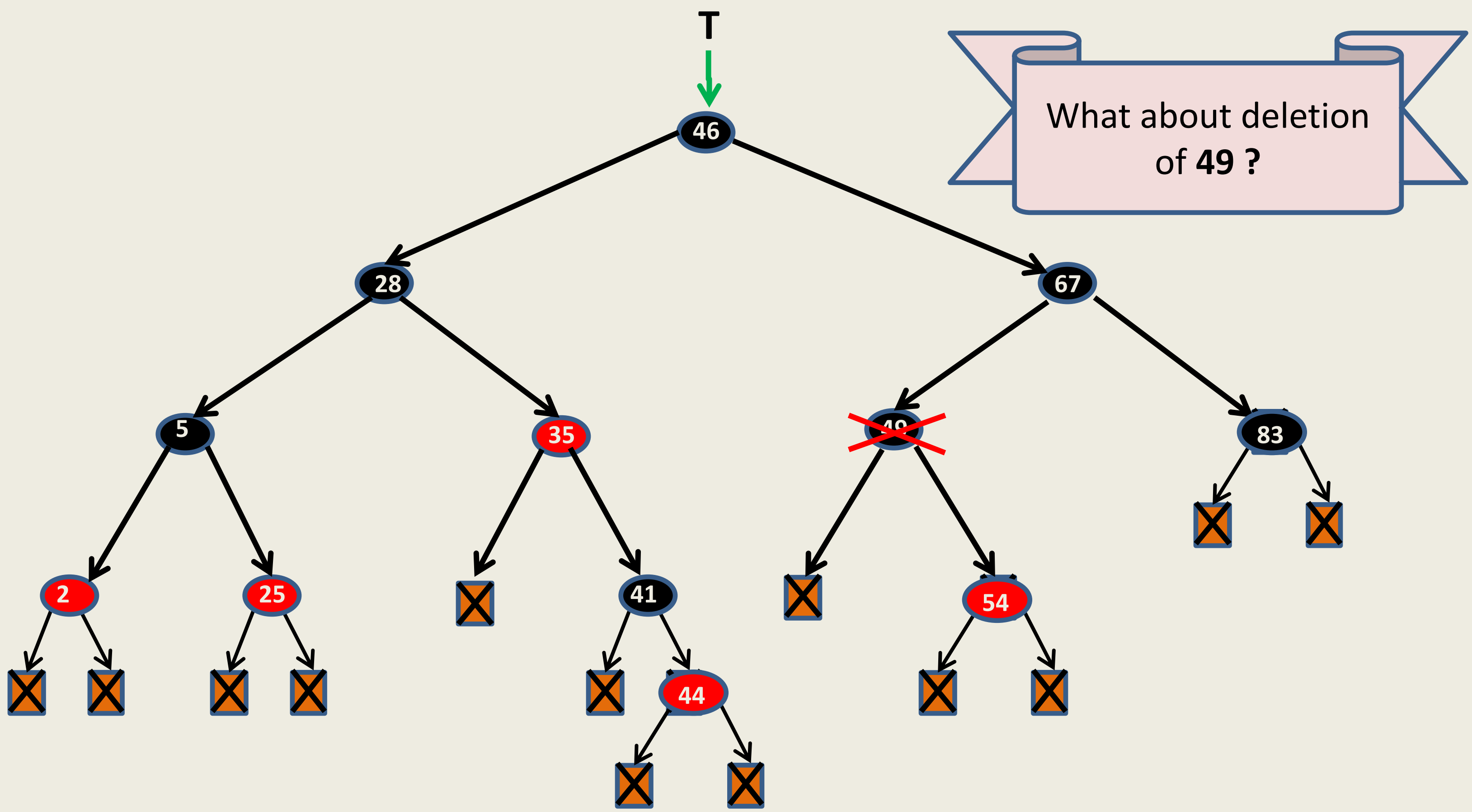


Is deletion of a node **easier** for some cases ?

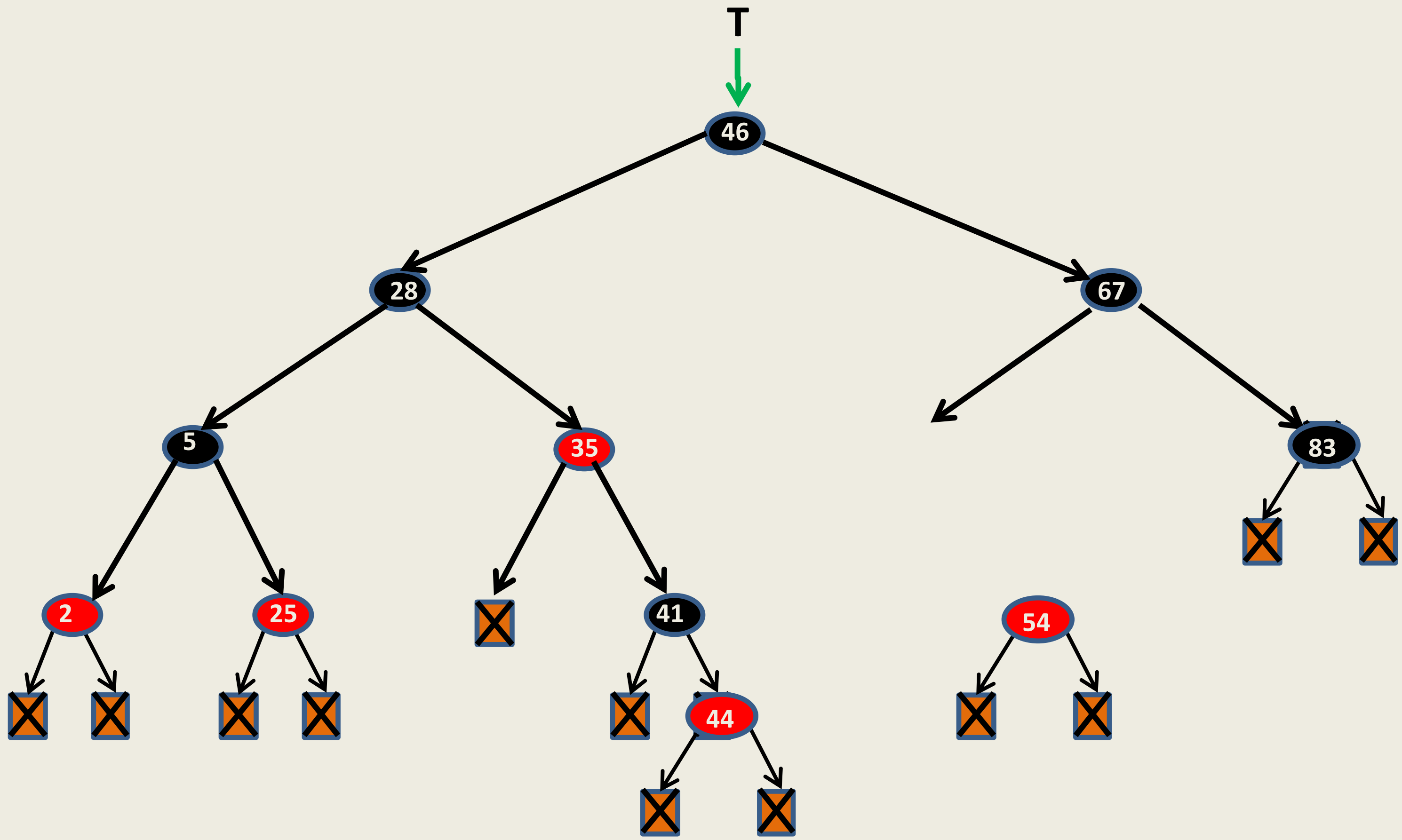


Deletion of **31** is easy.
Can you see ?

Is deletion of a node easier for some cases ?



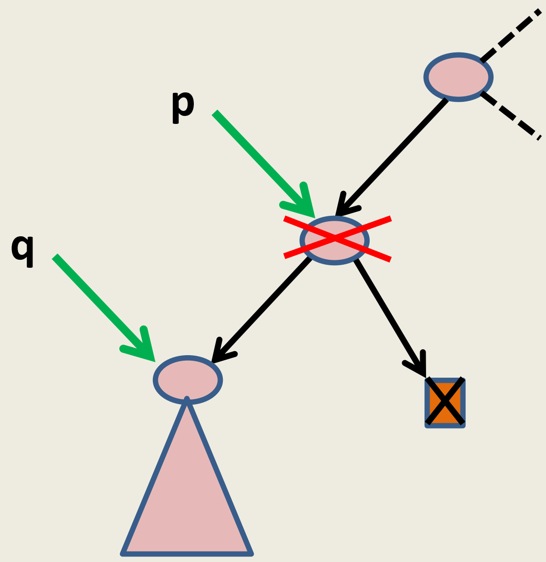
Is deletion of a node easier for some cases ?



An insight

It is easier to maintain a BST under deletion if

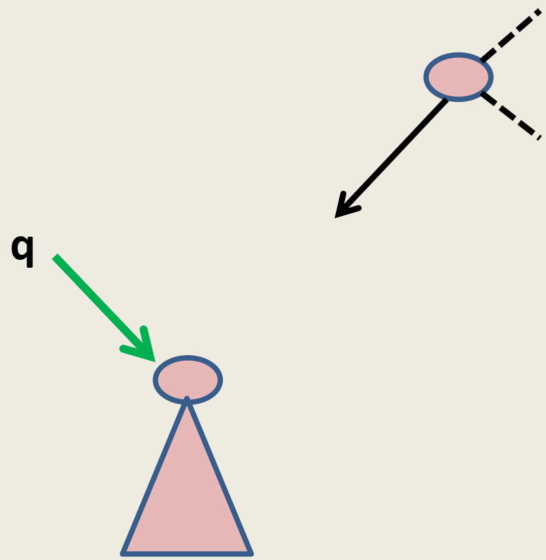
the node to be deleted has **at most** one child which is **non-leaf**.



An insight

It is easier to maintain a BST under deletion if

the node to be deleted has **at most** one child which is **non-leaf**.

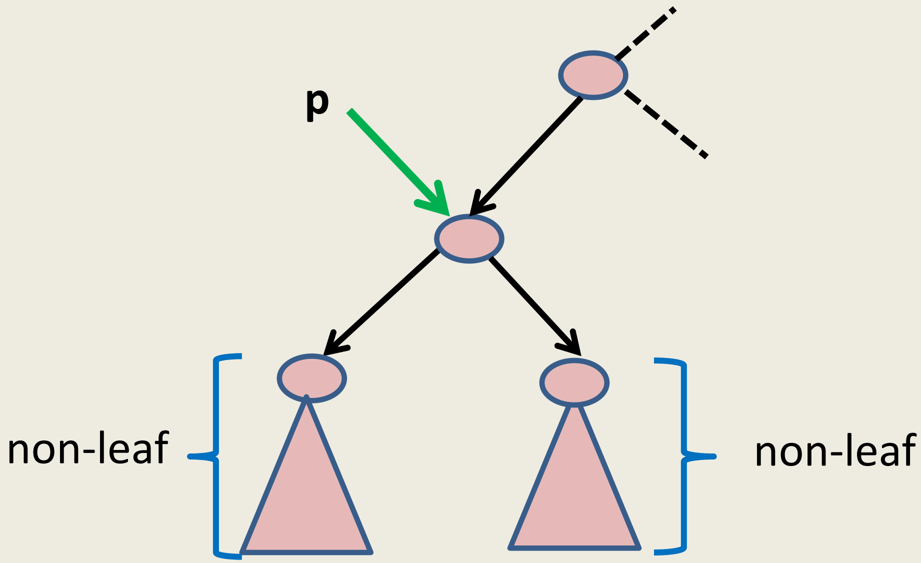


An important question

It is easier to maintain a BST under deletion if

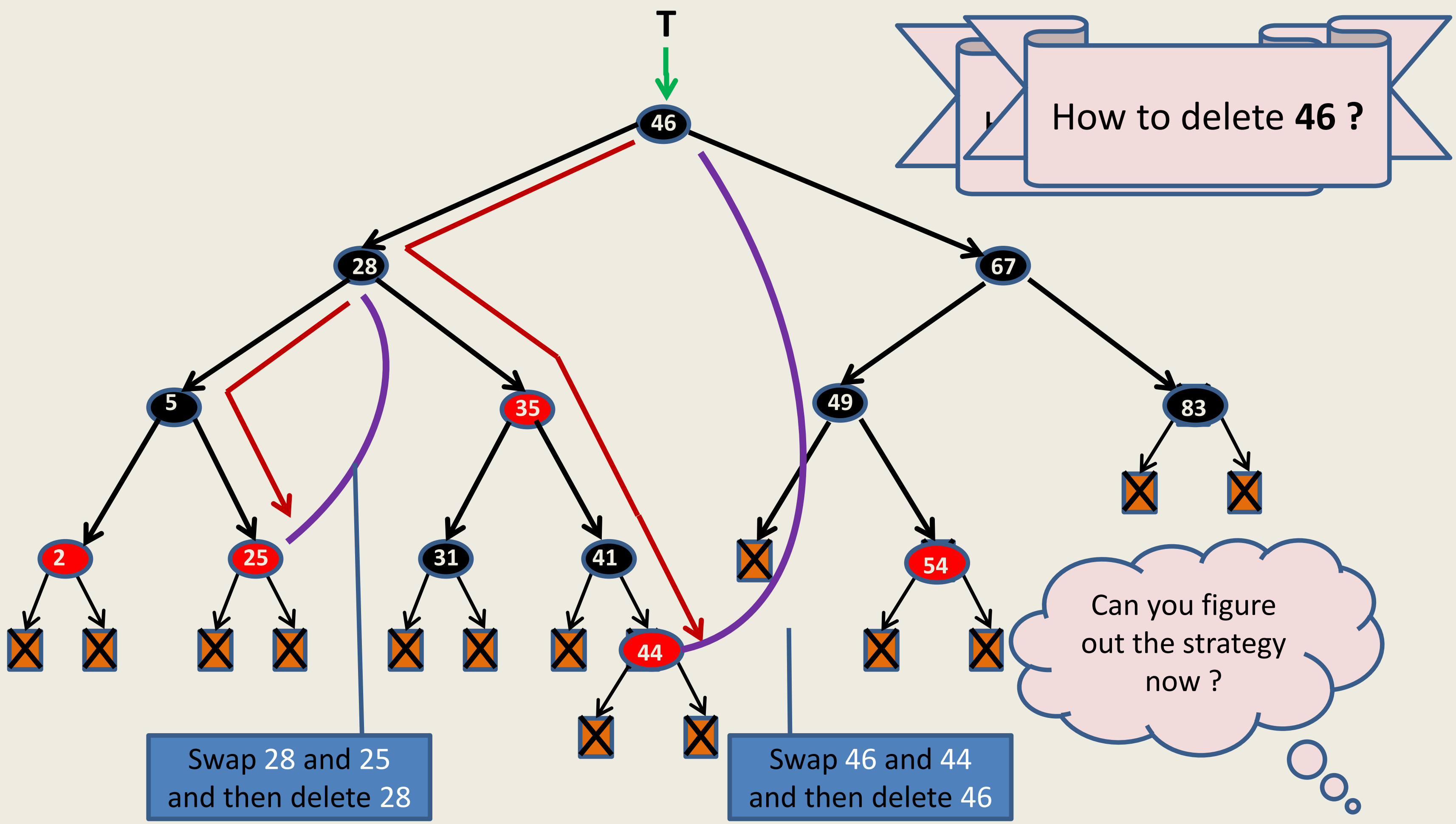
the node to be deleted has **at most** one child which is **non-leaf**.

Question: Can we transform every other case to the above case ?



Answer: ??

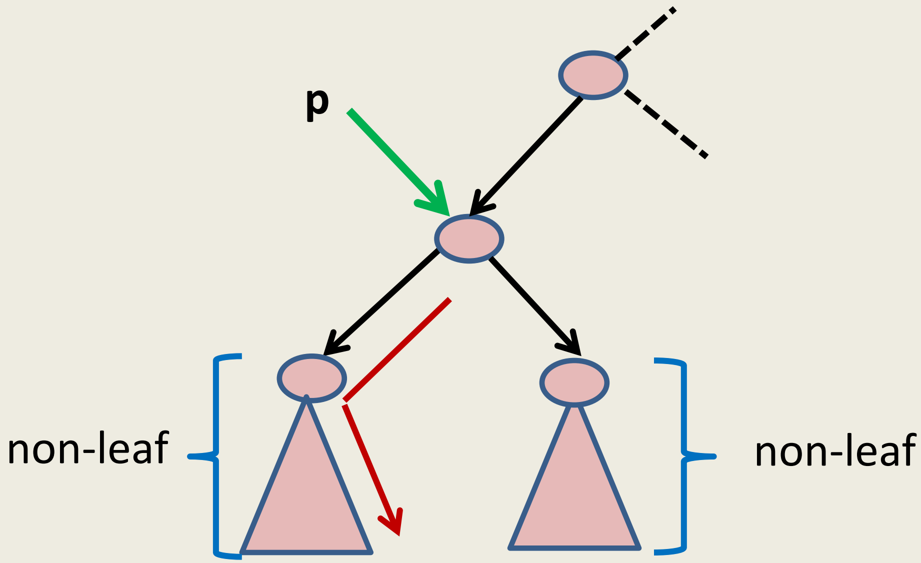
How to delete a node whose both children are non-leaves?



An important observation

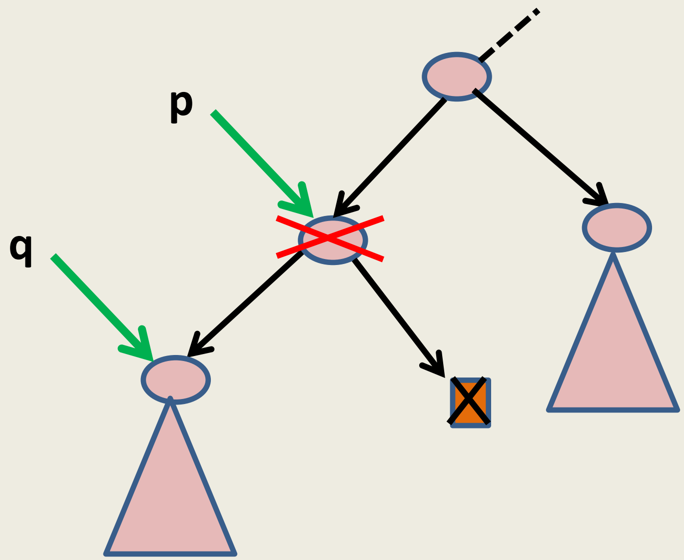
It is <u>easier</u> to maintain a BST under deletion if the node to be deleted has **at most** one child which is **non-leaf**.

Question: Can we transform every other case to the above case ?



Answer: by swapping value(**p**) with its predecessor.

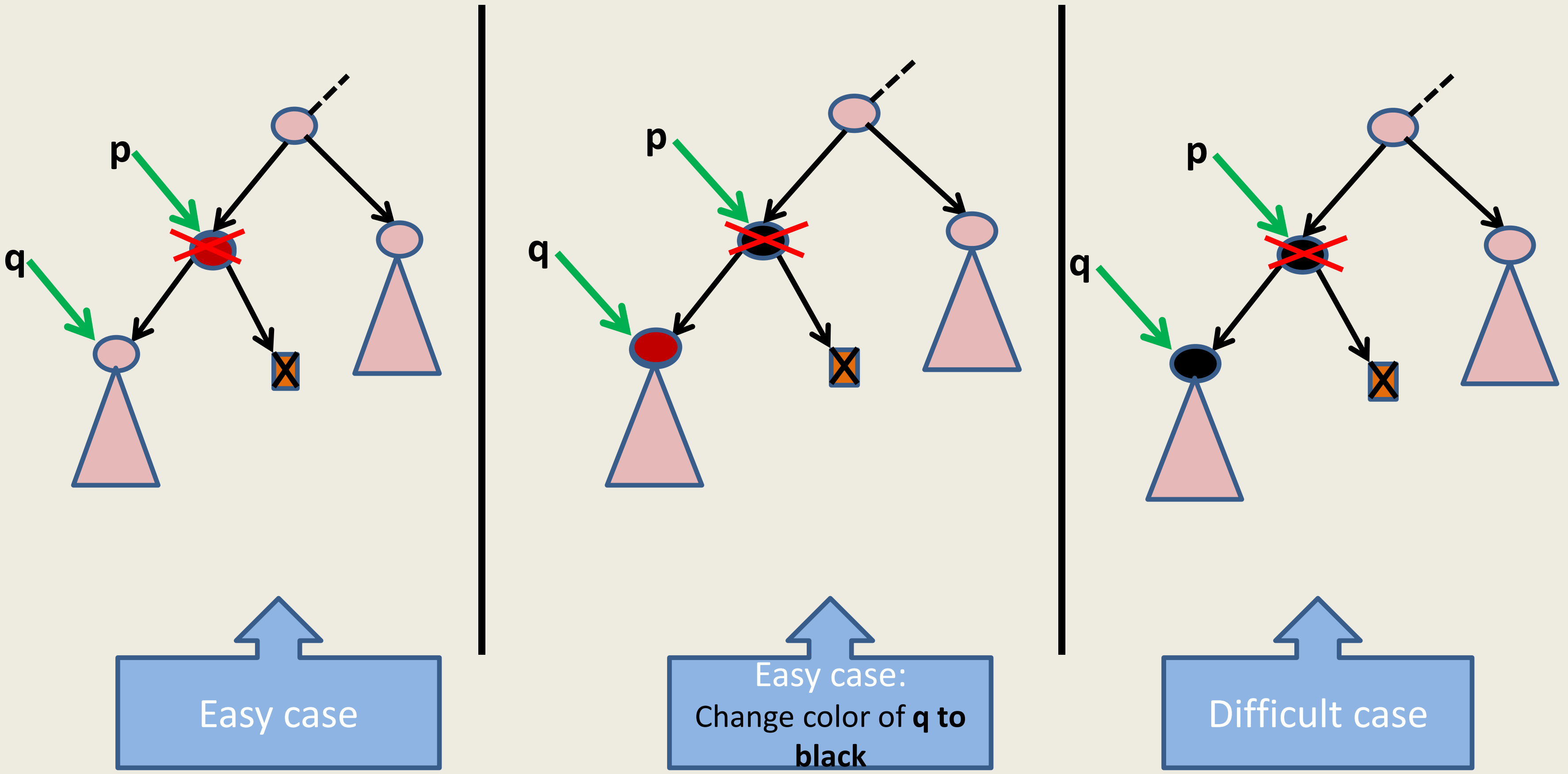
We need to handle deletion only for the following case



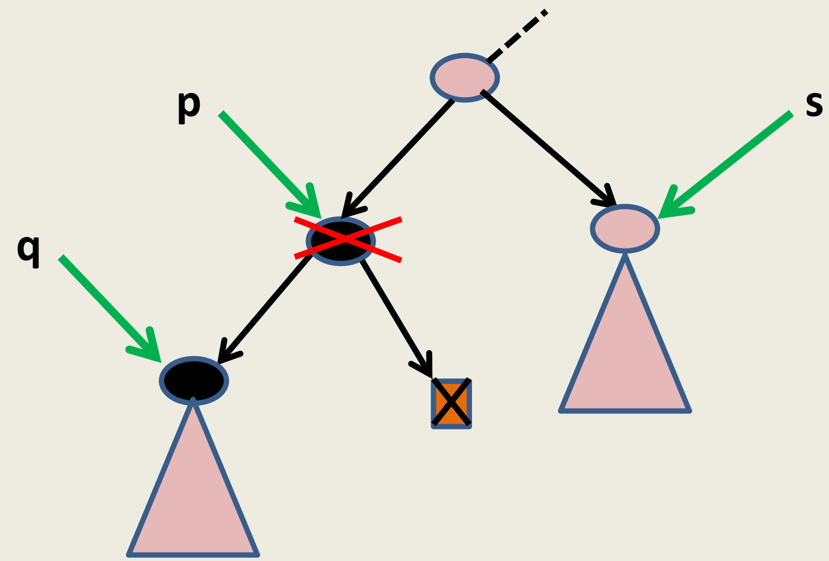
How to maintain a red-black tree under deletion ?

We shall first perform deletion like in an ordinary BST and then restore all properties of red-black tree.

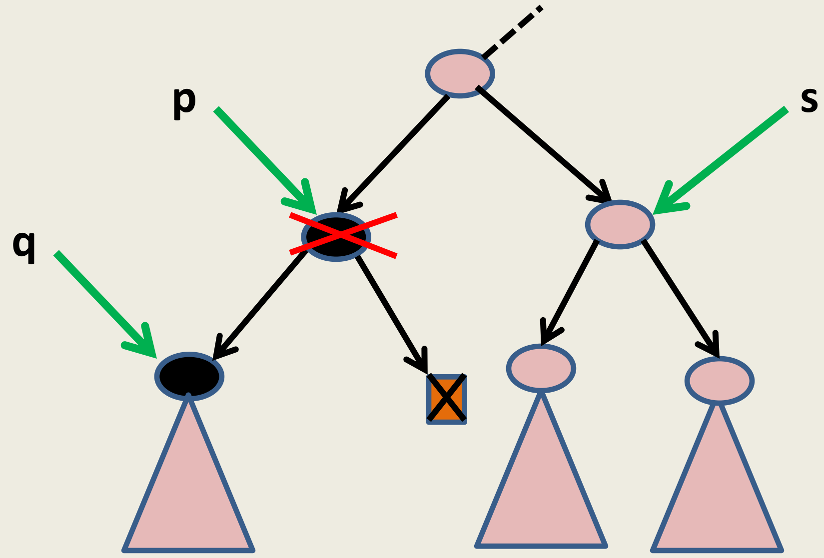
Easy cases and difficult case



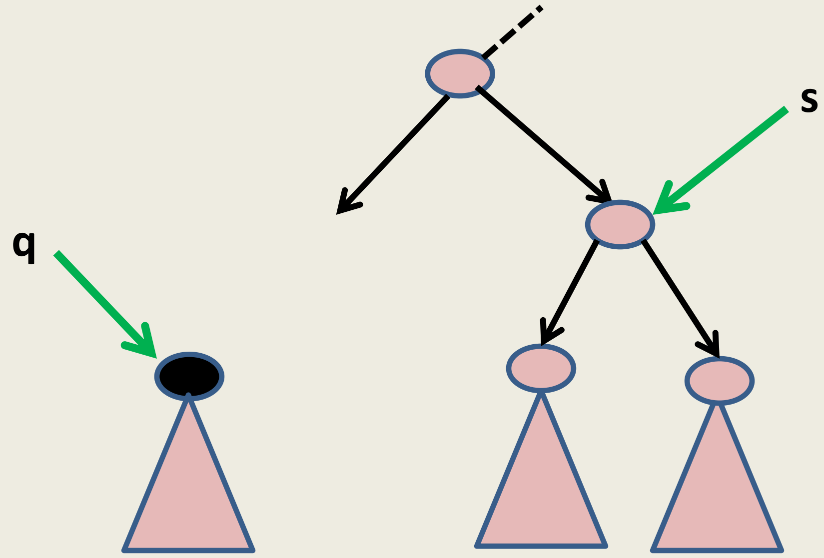
Handling the difficult case



Handling the difficult case

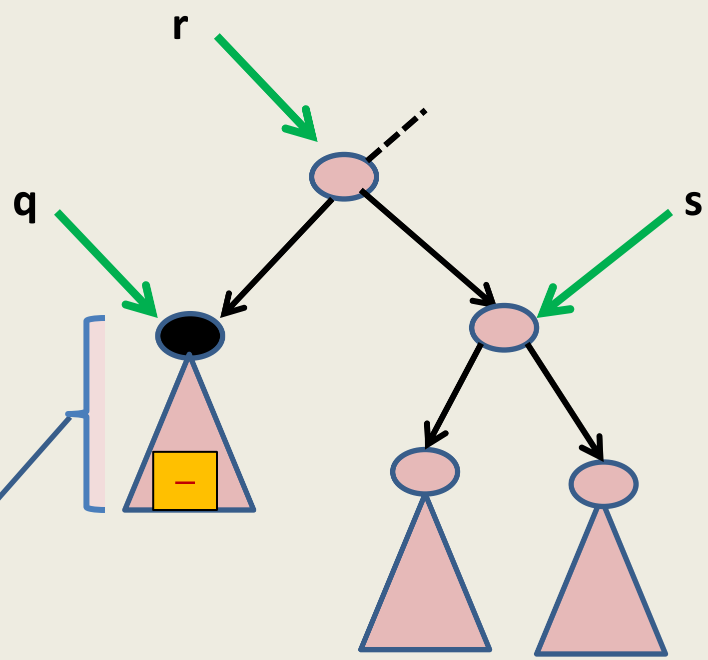


Handling the difficult case



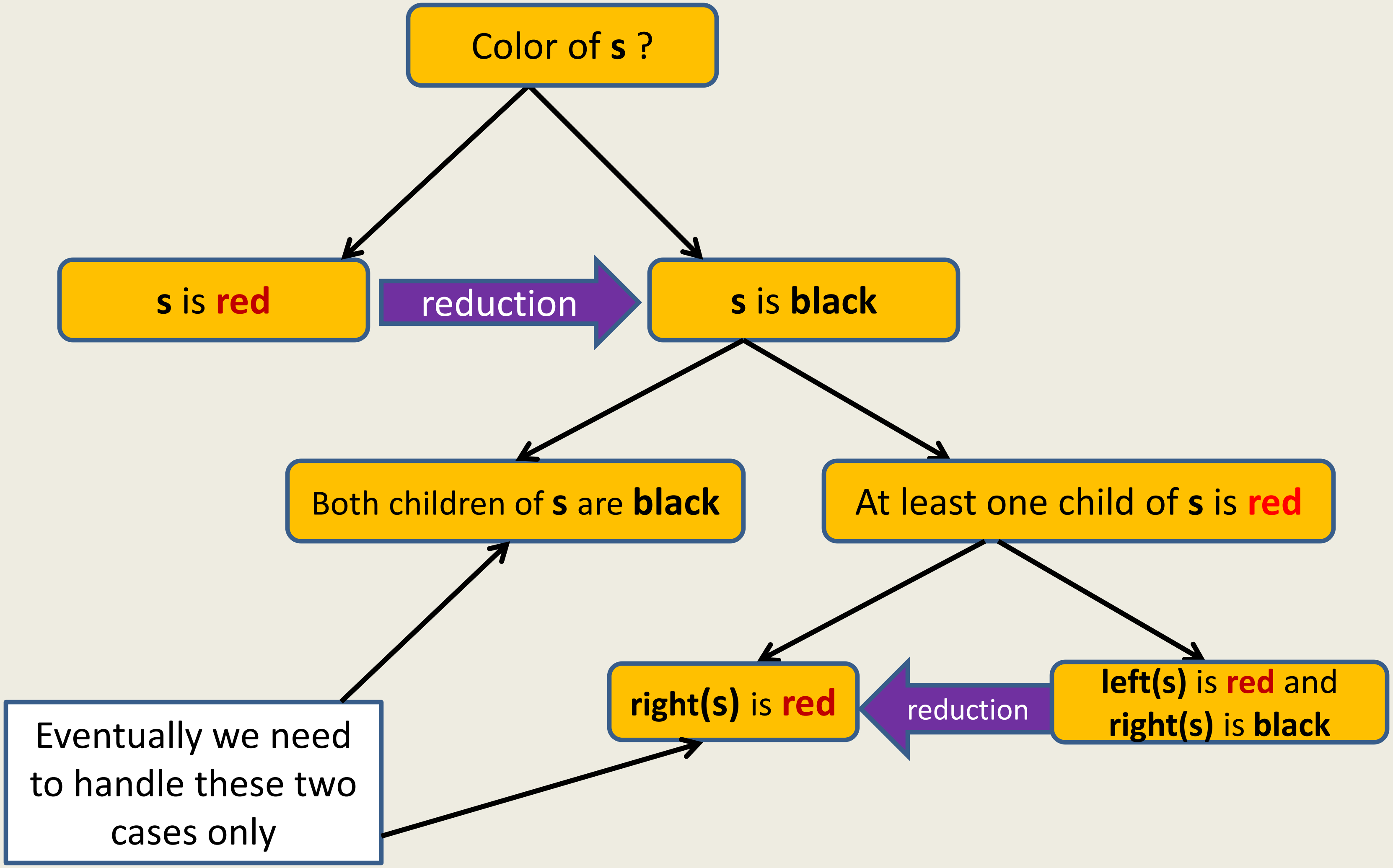
Handling the difficult case

Notice that the number of black nodes to each leaf node in subtree(**q**) has become **one** less than leaf nodes in other trees. We need an algorithm to remove this **black-height imbalance**.



As some students had noticed after the class that the subtree(**q**) will actually be just a leaf node in the beginning. But we are not showing it explicitly here. This is because we are depicting the most general case. During the algorithm, we might shift the height imbalance upwards and in that case the subtree(**q**) might not be a leaf node.

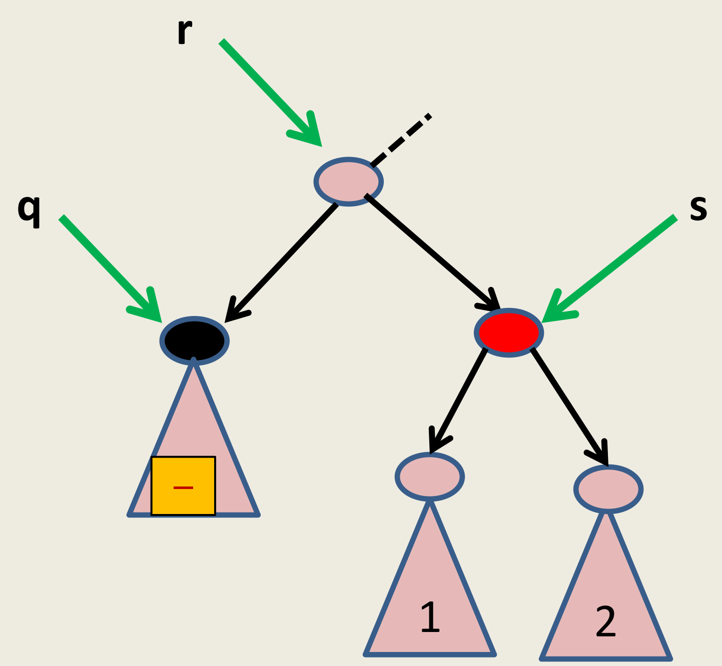
Handling the difficult case: An overview



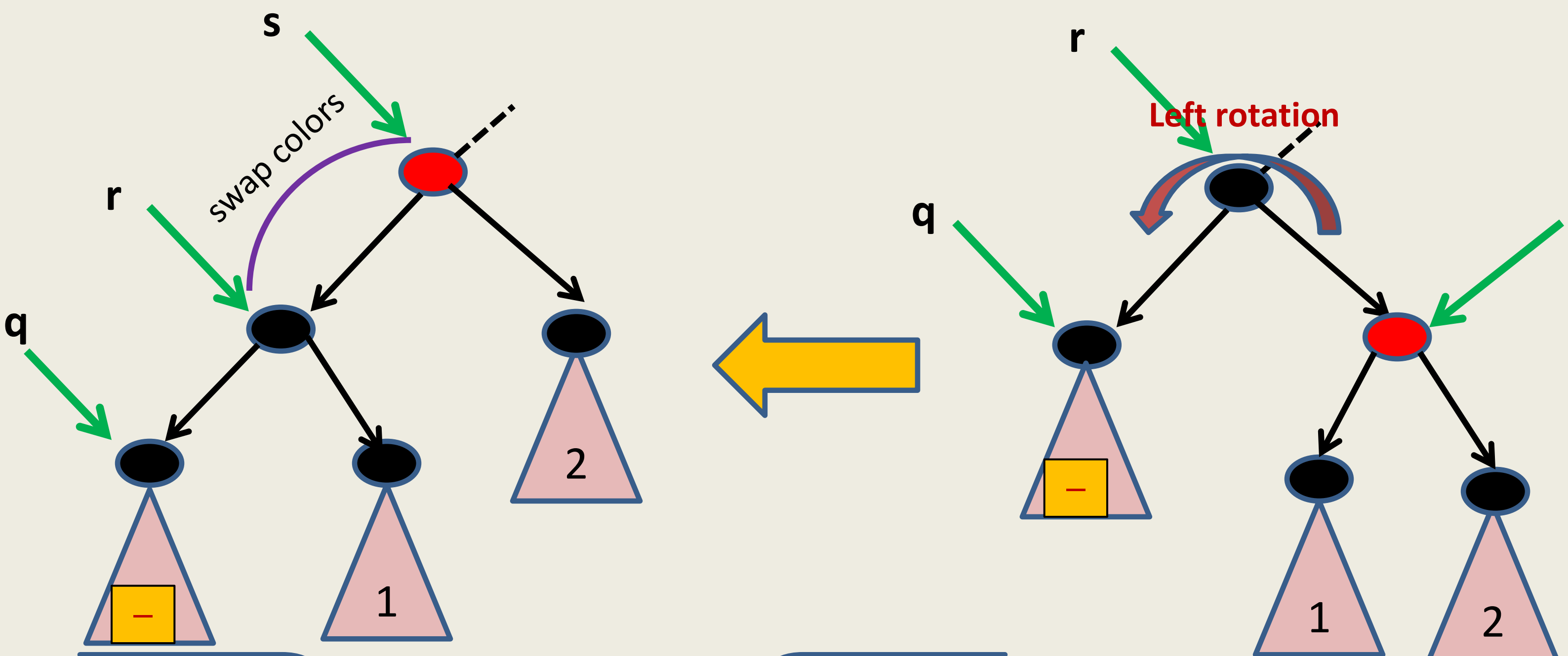
“s is red” → reduction → **“s is black”**

“s is **red**” → reduction → “s is black”

What can we say about **parent** and **children** of **s** ?

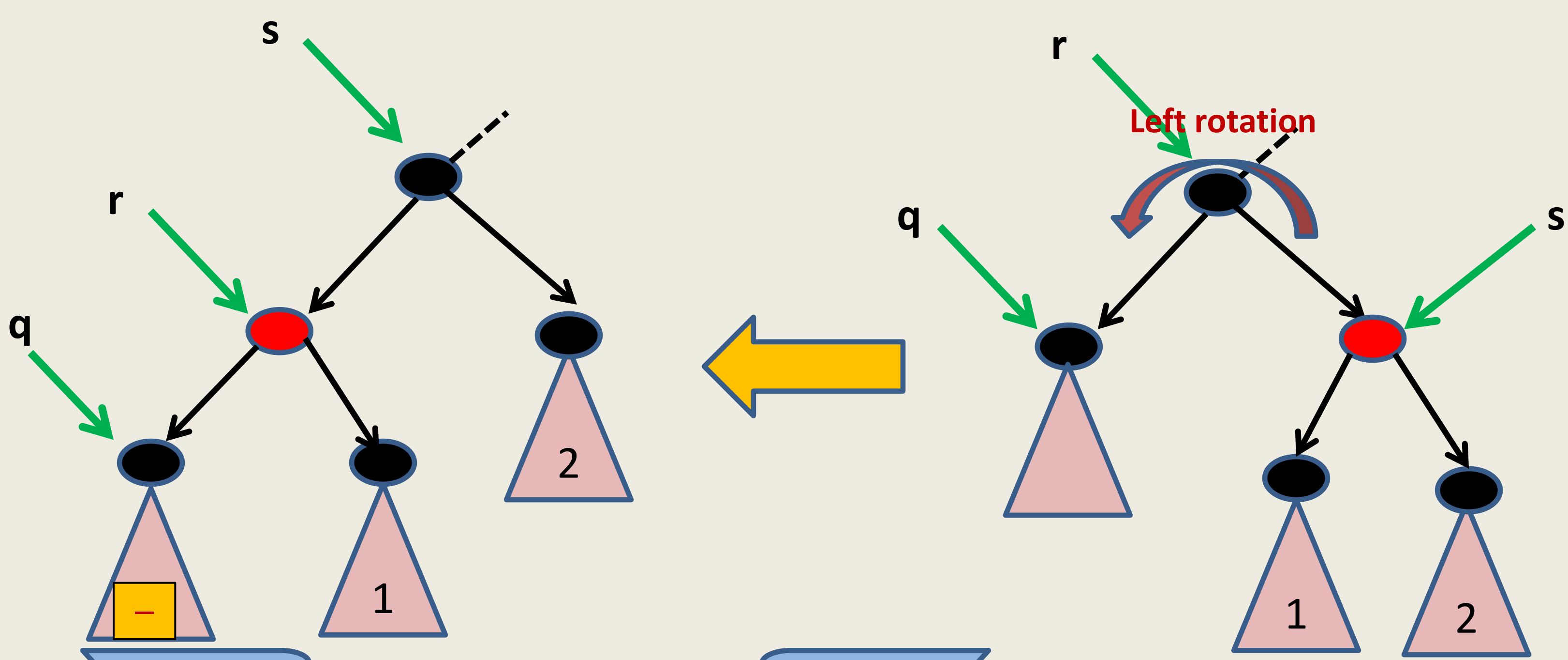


"s is red" → reduction → "s is black"



The new sibling of **q** is now a black node. But the number of black nodes to leaves of tree 2 have reduced by one. What to do ?

“s is red” reduction “s is black”



Convince yourself that the number of black nodes to any leaf of subtree(**q**) or subtrees 1 and 2 is now the same as before the rotation. And now the sibling of **q** is black. So we are done.

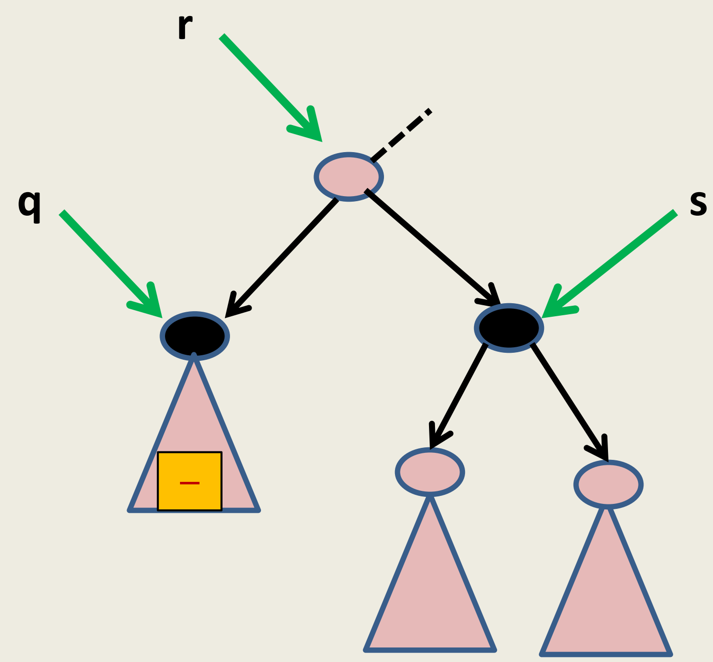
We just need to handle the case

“**s** is **black**”

Handling the case: s is black

Case 1: both children of **s** are **black**

Case 2: at least one child of **s** is **red**



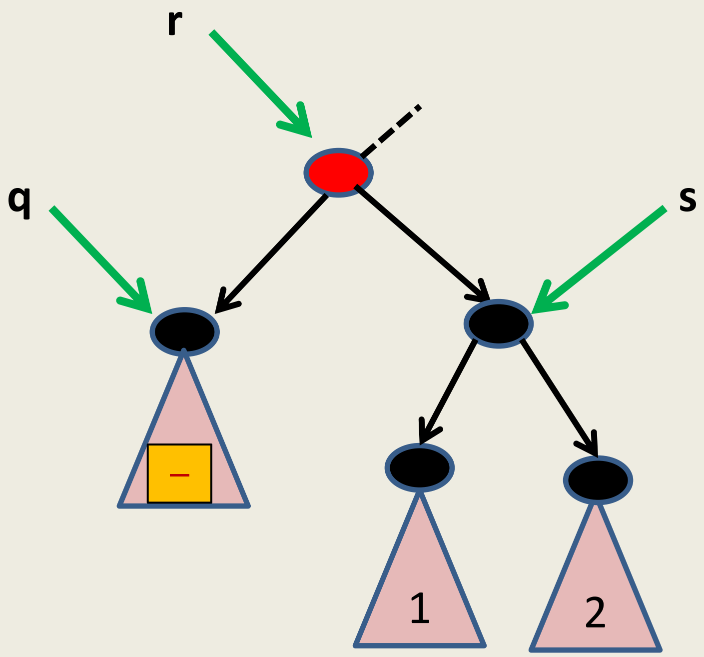
Handling the case:

s is **black** and both children of **s** are **black**

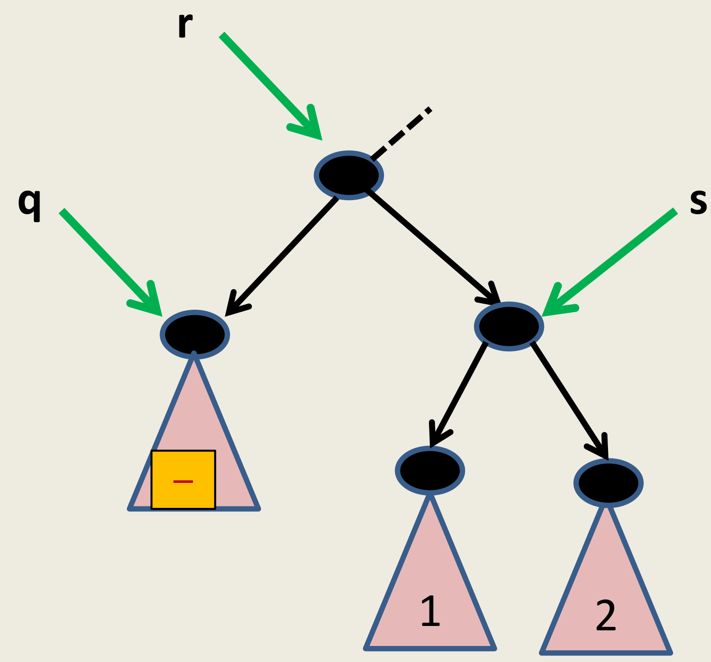
Handling the case:

s is **black** and both children of **s** are **black**

When **r** is **red**



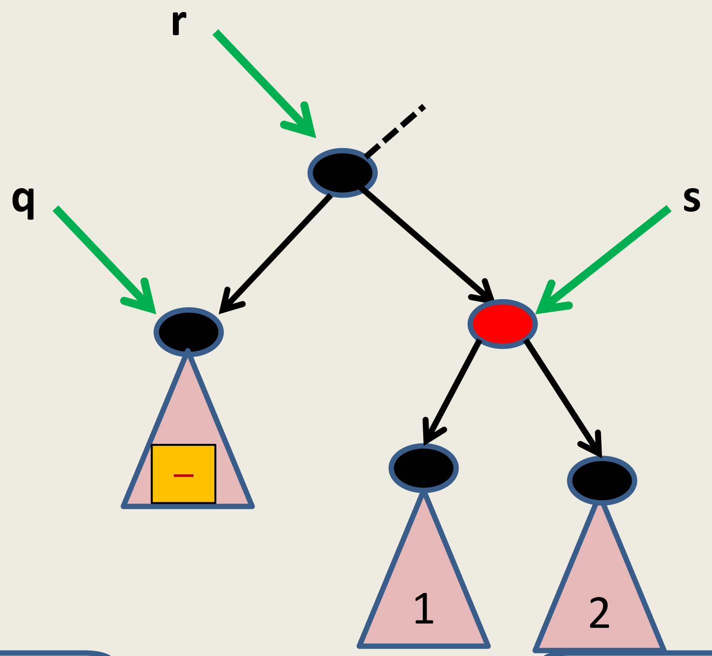
When **r** is **black**



How to handle this case ?

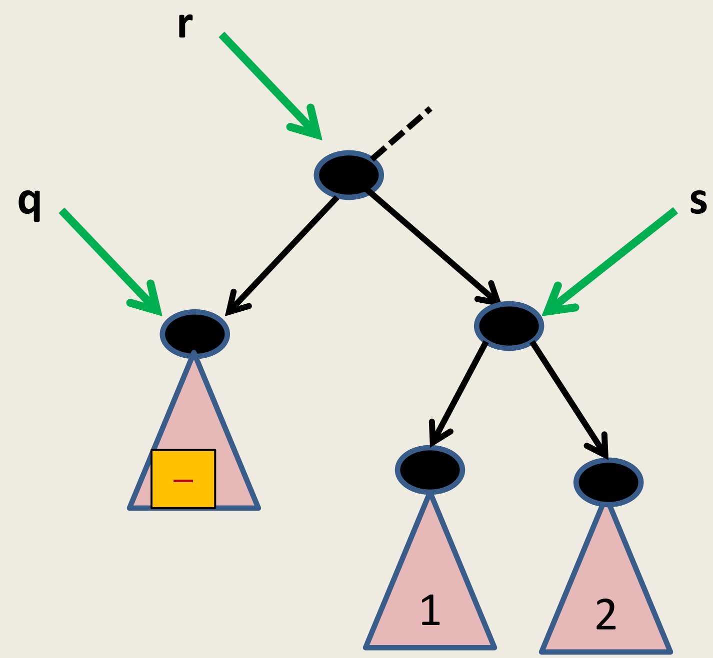
Handling the case:
s is black and both children of s are black

When r is **red**



YES.
As a result of swapping the colors, the number of black nodes to the leaves of trees 1 and 2 unchanged. Interestingly, the deficiency of one black node on the path to the leaves of subtree(**q**) is also compensated. So we are done☺

When r is **black**

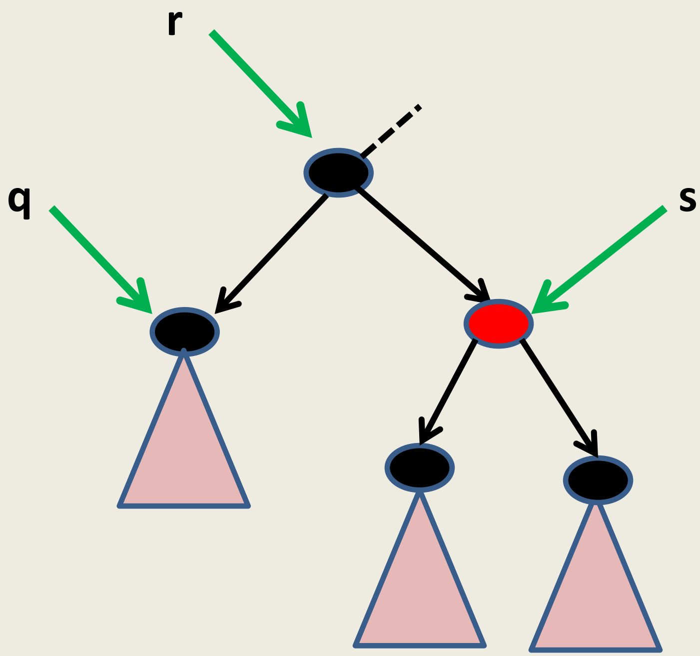


How to handle this case ?

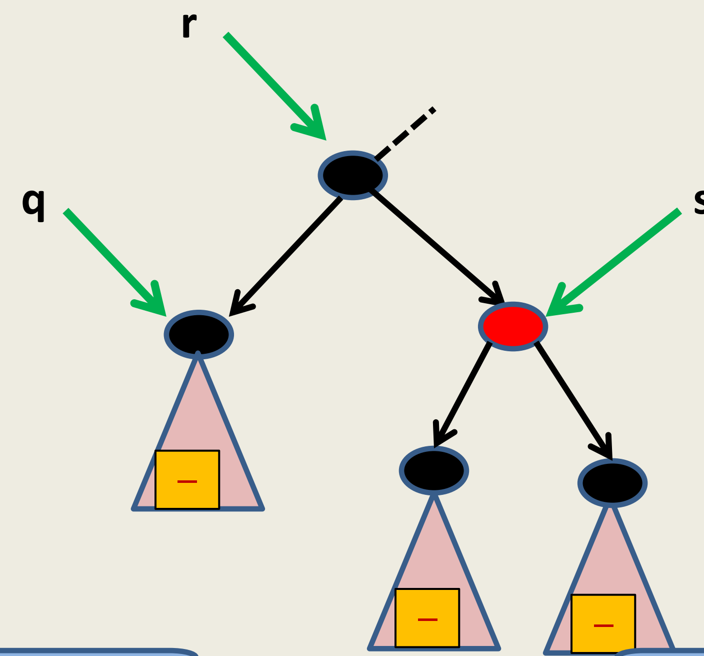
Handling the case:

s is **black** and both children of **s** are **black**

When r is **red**



When r is **black**



Changing color of **s** to **red** has reduced the number of black nodes on the path to the root of subtree(**s**) by one. As a result the imbalance of black height has *propagated* upward. So we process the new **q**.

Handling the case:

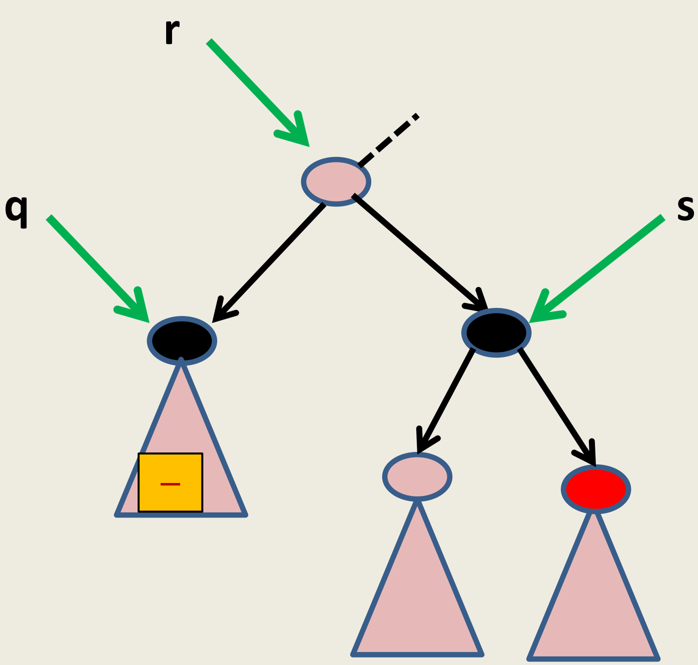
s is **black** and <u>one</u> of its children is **red**

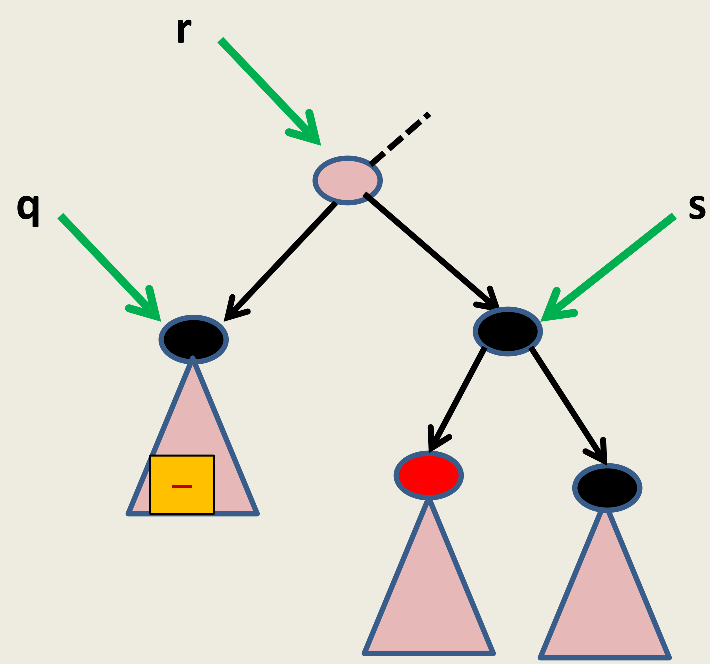
There are two cases

When **right**(s) is **red**

reduction

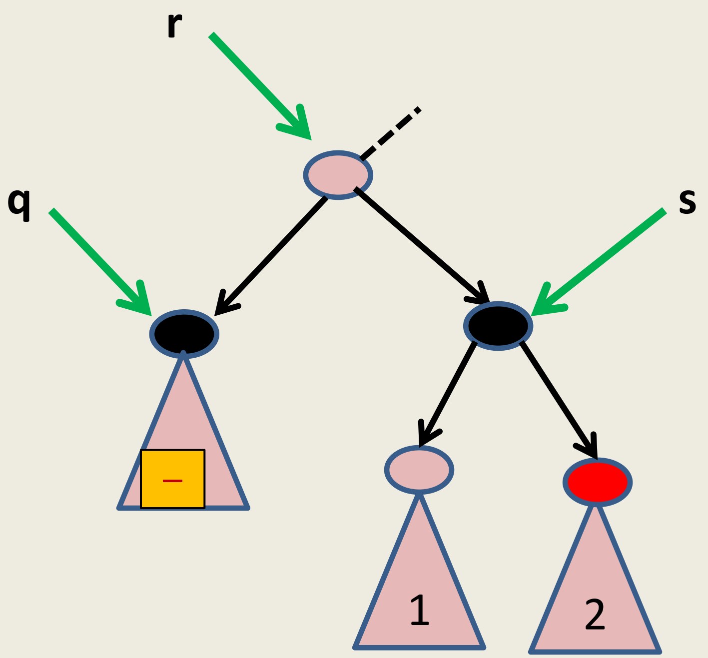
When **left**(s) is **red** and **right(**s**)** is **black**





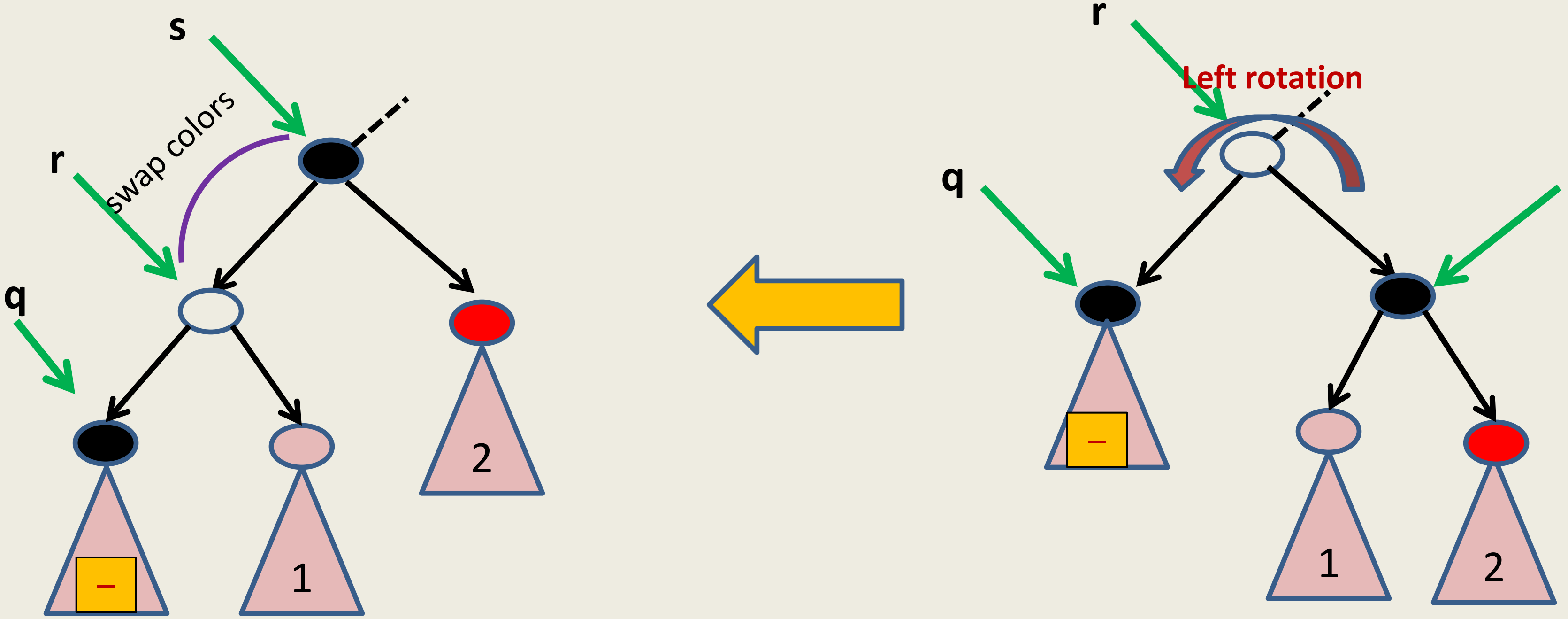
Handling the case: right(s) is red

Handling the case: right(s) is red



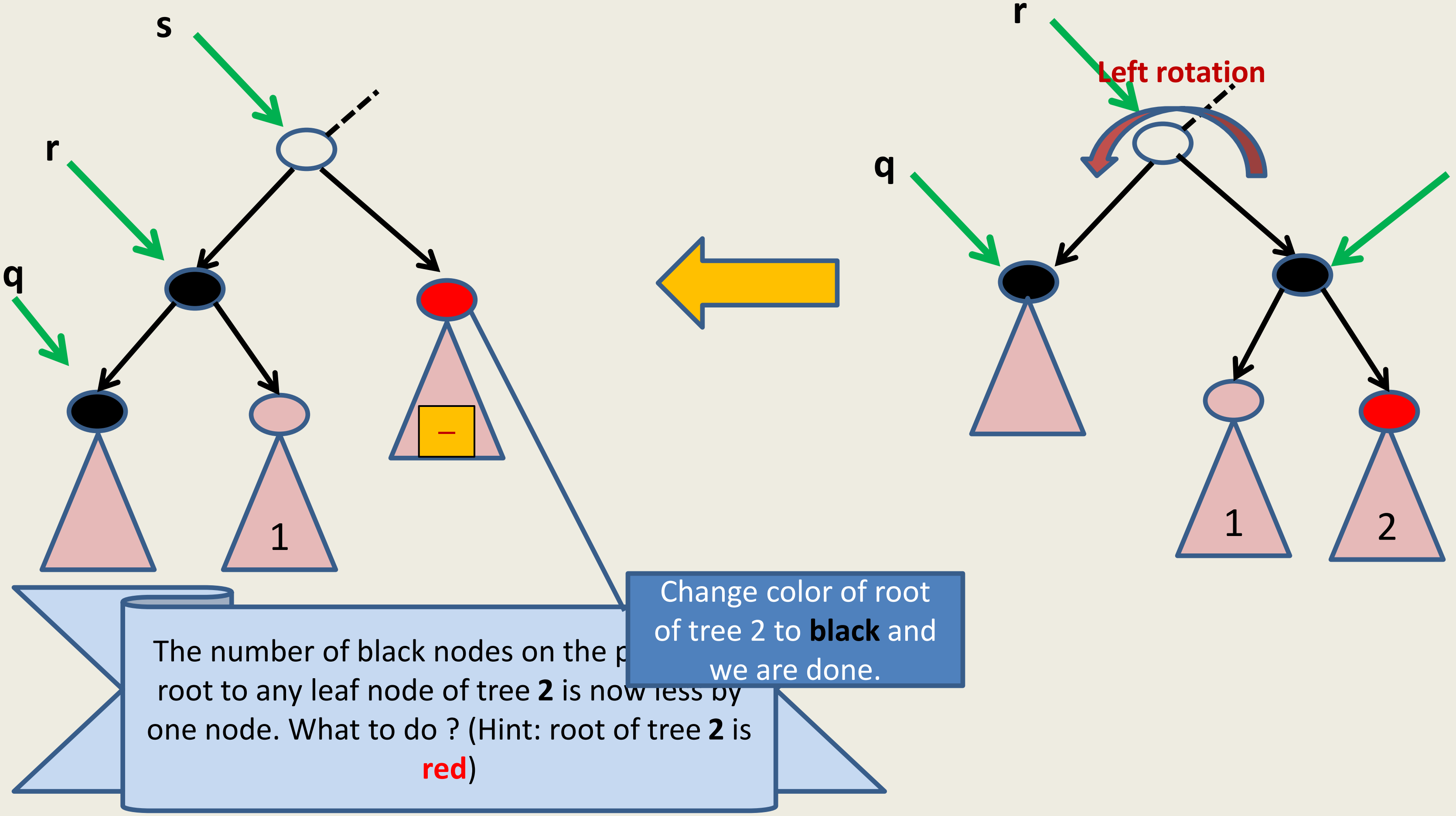
Let **color**(**r**) be c

Handling the case: right(s) is red

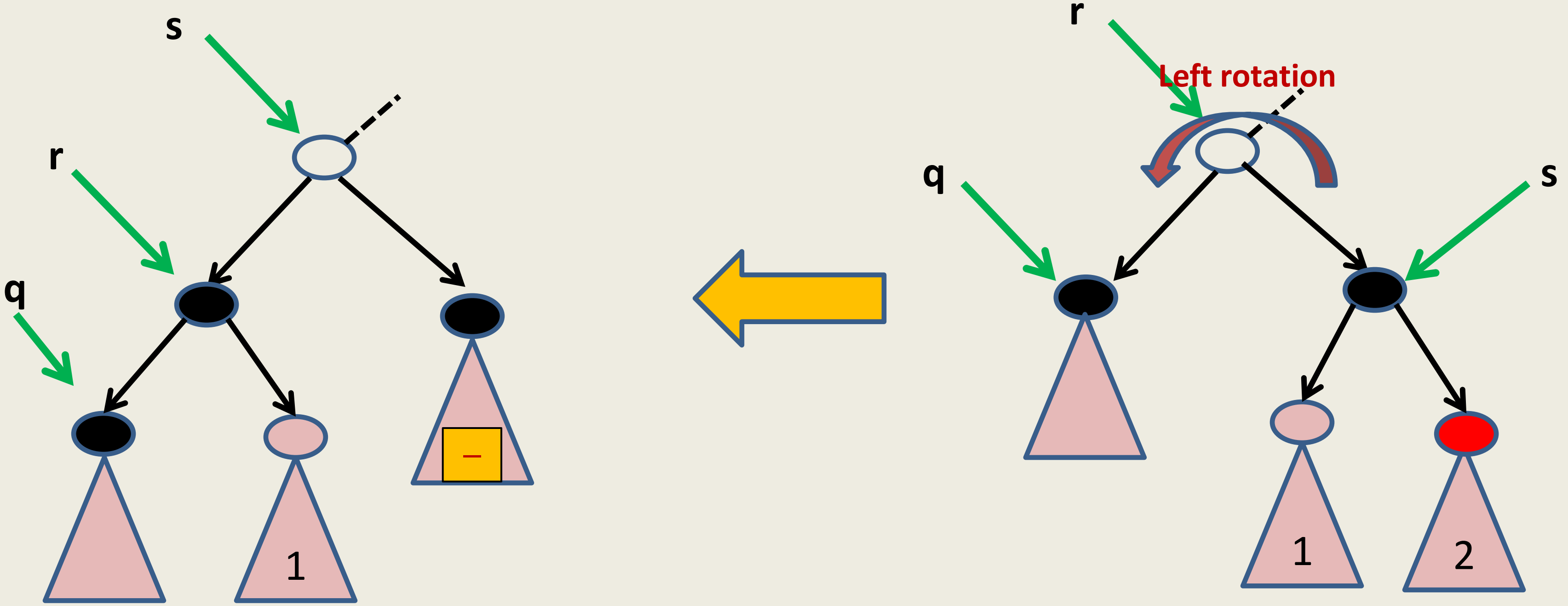


The number of black nodes on the path from root to any leaf node of subtree(**q**) has increased by one (this is good!), has remained unchanged for leaves of tree 1, and is uncertain for leaves of tree 2(depends upon c). How to get rid of this uncertainty ?

Handling the case: right(s) is red



Handling the case: right(s) is red

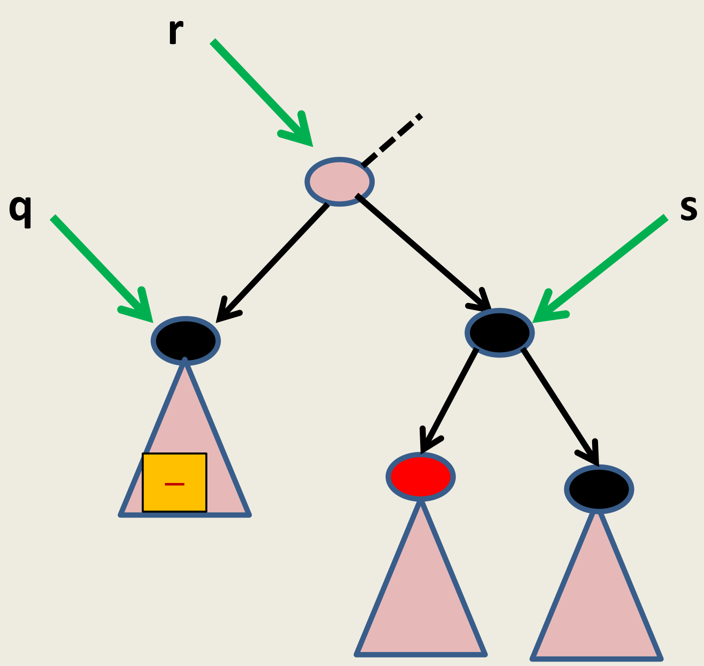


Convince yourself that left rotation at **r**, followed by color swap of **s** and **r**, followed by change of color of root of tree **2** removes the imbalance of black height for all leaf nodes of the subtrees shown.

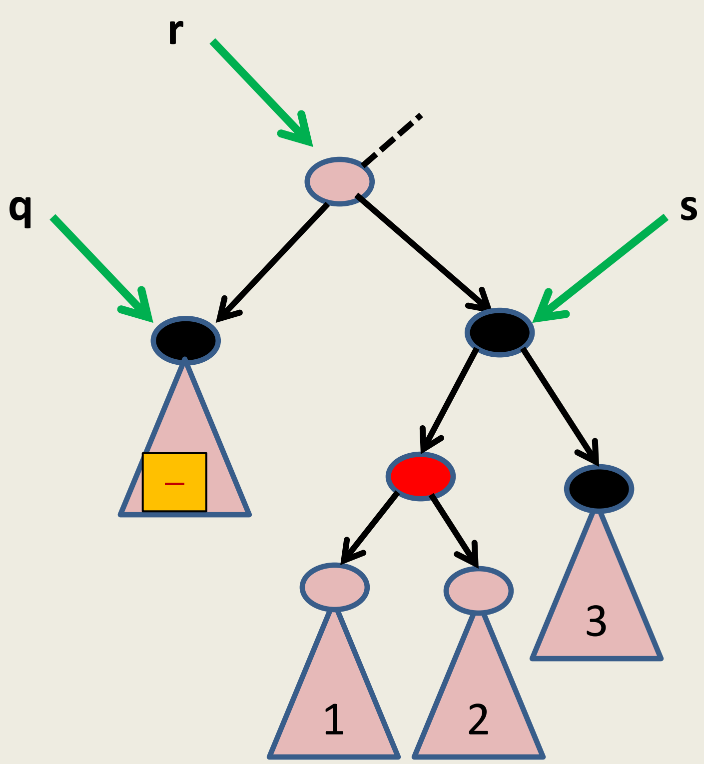
Handling the case

“left(**s**) is **red** and right(**s**) is **black**”

Handling the case: left(s) is red and right(s) is black

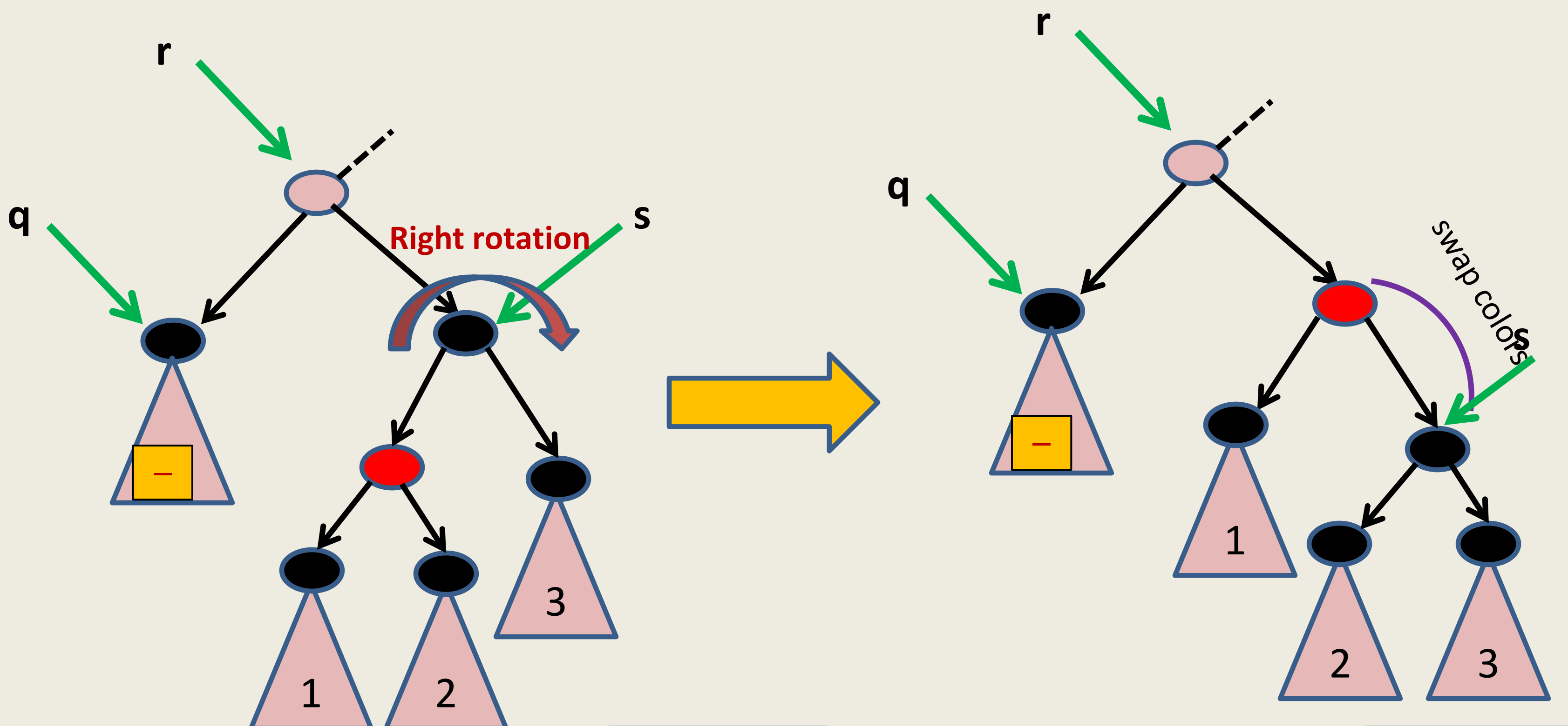


Handling the case:
left(s) is red and right(s) is **black**



Handling the case:

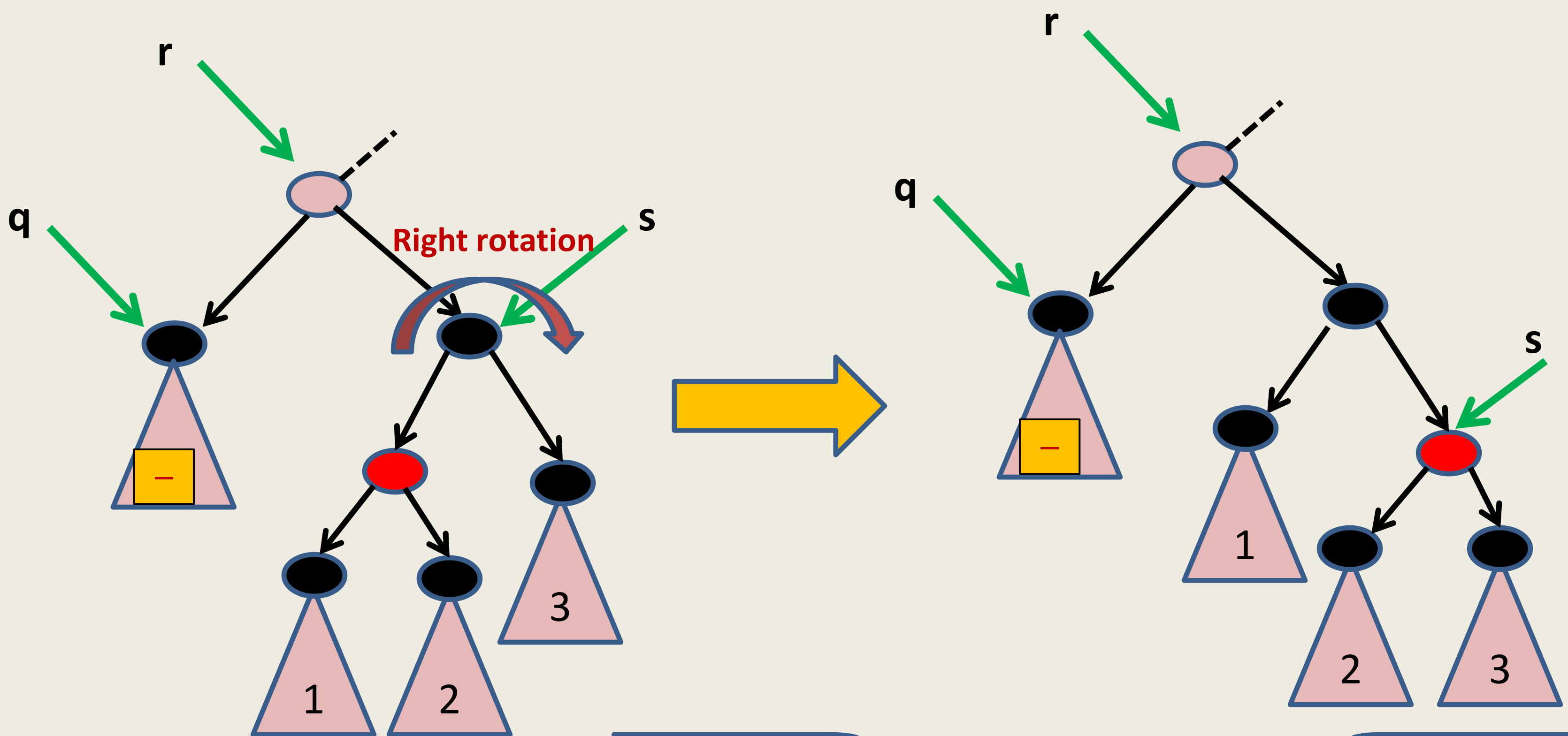
left(s) is red and right(s) is black



The number of black nodes on the path from root to any leaf node in tree 1 has now reduced by one although it is the same for trees 2 and 3. What should we do ?

Handling the case:

left(s) is **red** and right(s) is **black**



Notice that now the new sibling of **q** has its right child **red**. So we have effectively reduced the current case to the case which we know how to handle.

Theorem: We can maintain red-black trees in **O**(log n) time per insert/delete/search operation.

where n is the number of the nodes in the tree.

A **Red** **Black** Tree is <u>height balanced</u>

A **detailed proof** from **scratch**

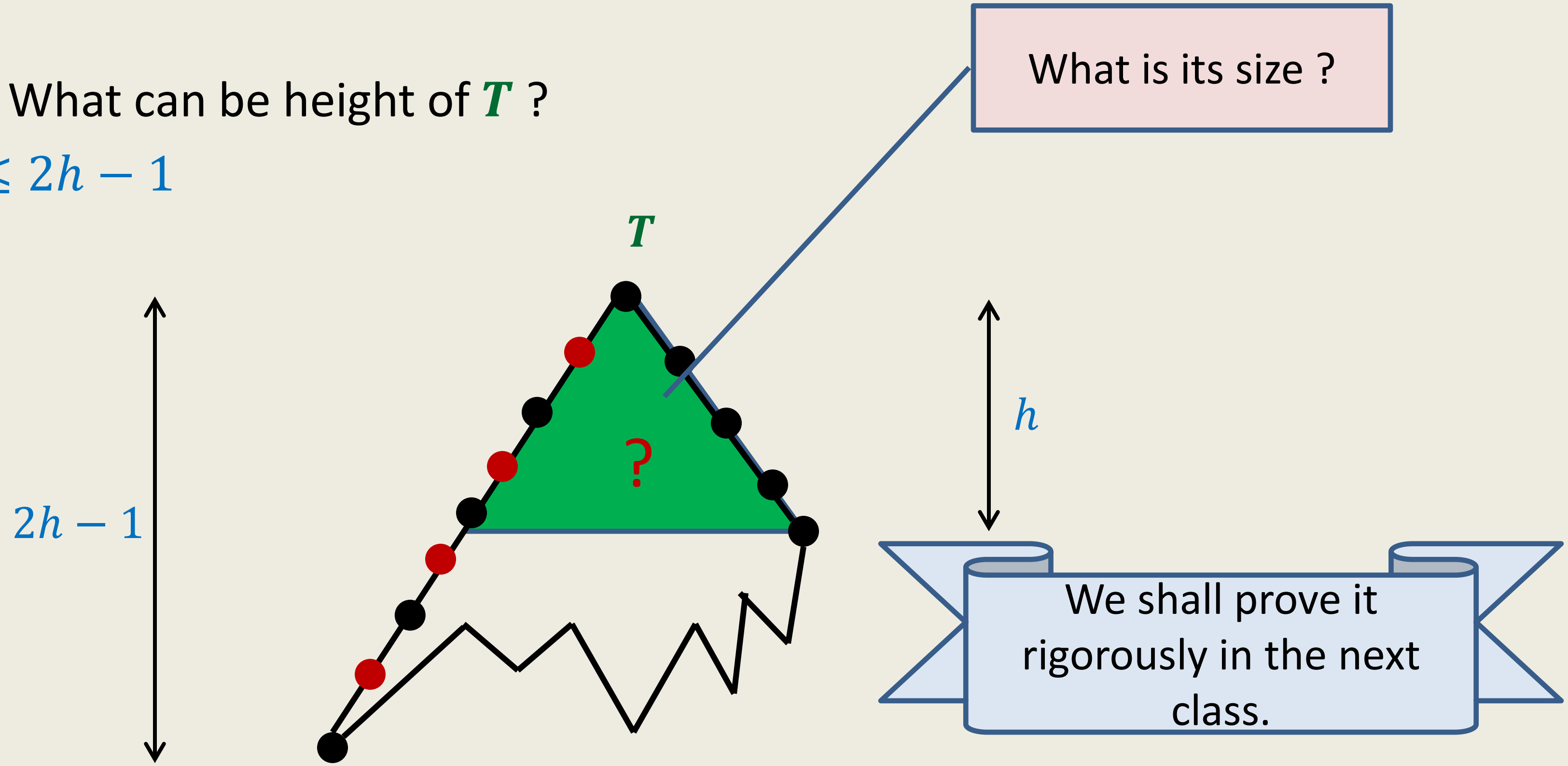
Why is a red black tree height balanced ?

T : a **red black tree**

h : **black** height of T.

Question: What can be height of T ?

Answer: $\leq 2h-1$



Theorem: The shaded green tree is a complete binary tree & so has $\geq 2^h$ elements.