

Data Structures and Algorithms

(CS210A)

Semester I – 2014-15

Lecture 16:

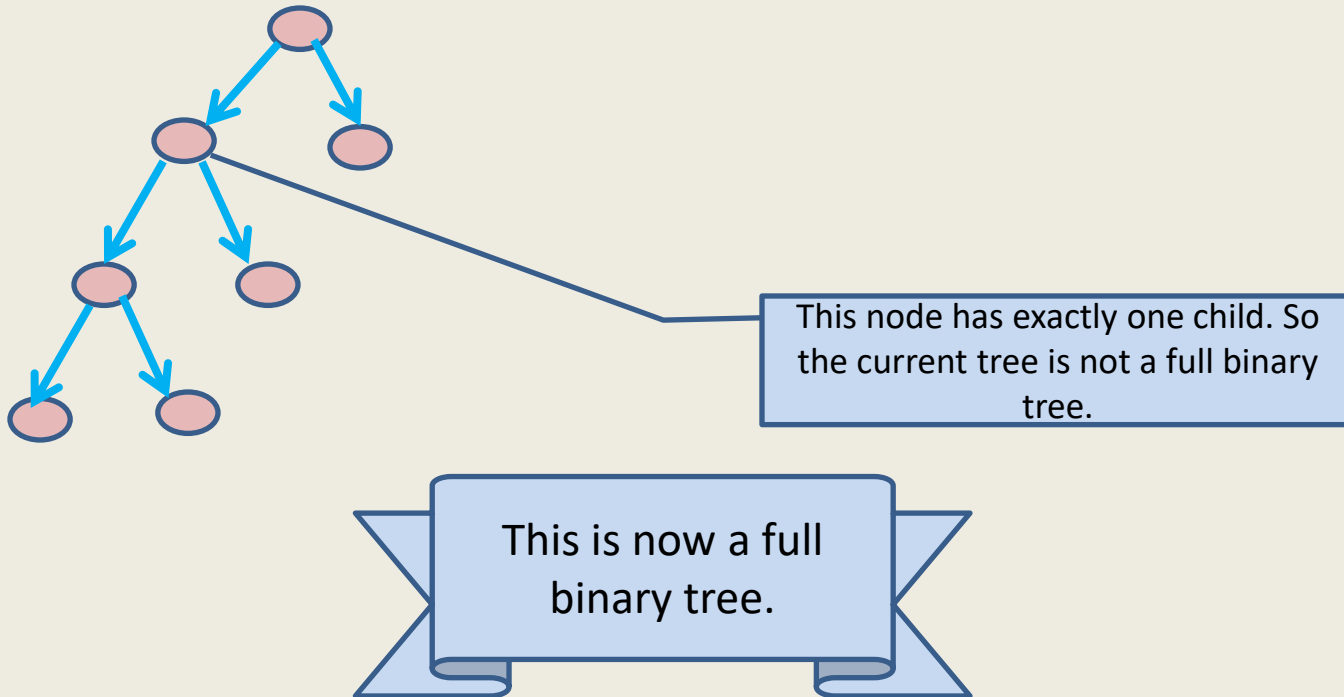
Height balanced BST

- Red-black trees

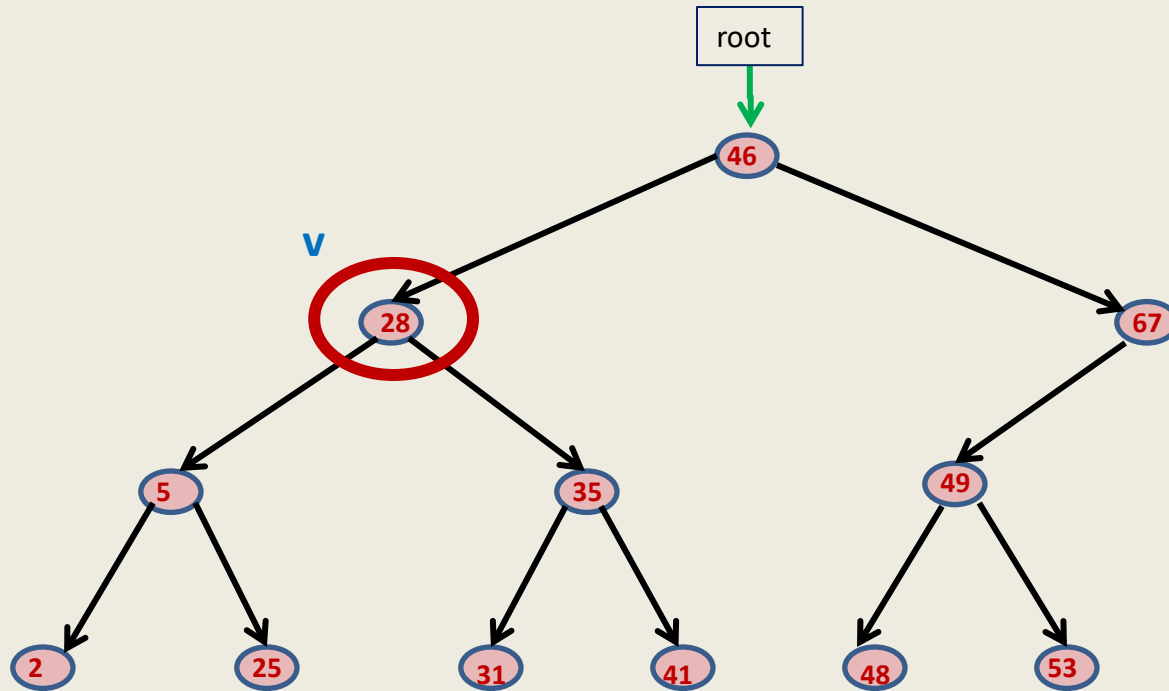
Terminologies

Full binary tree:

A binary tree where every internal node has exactly two children.



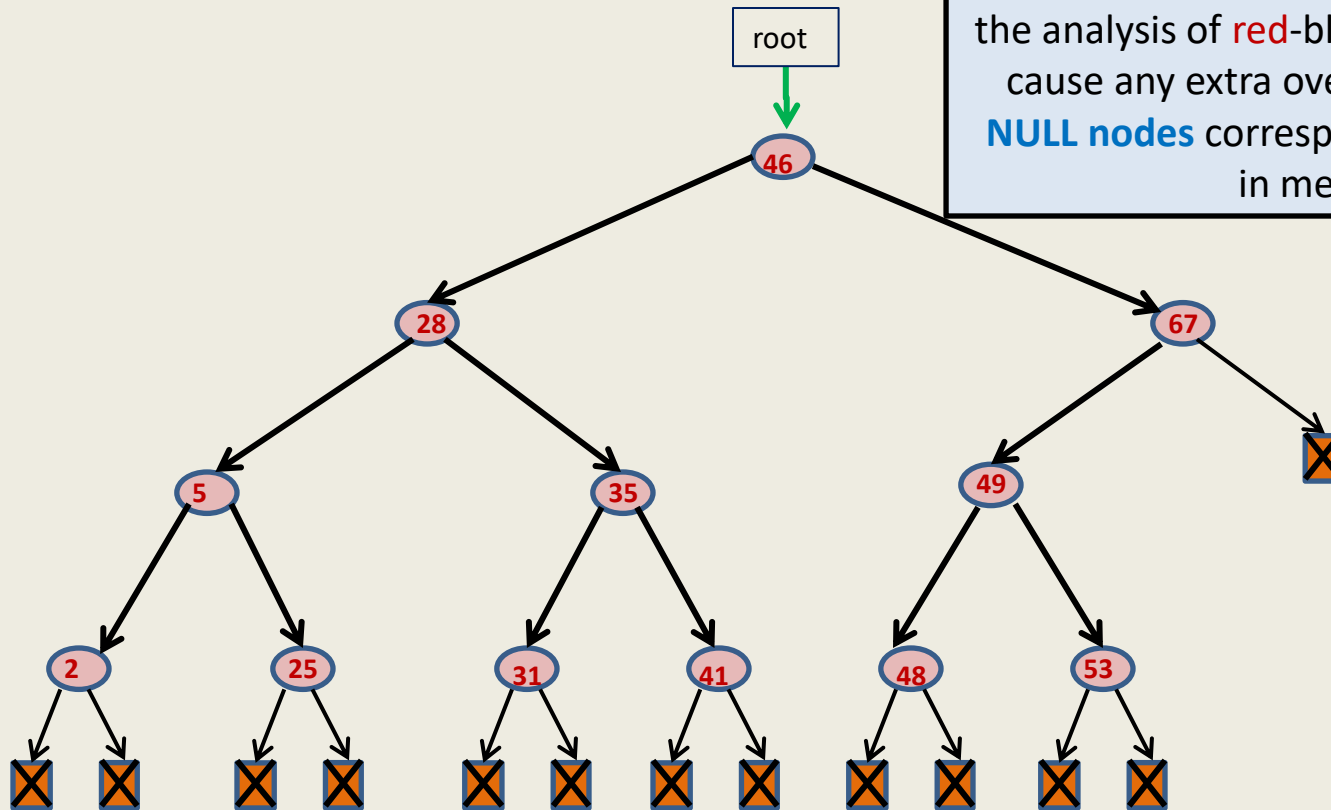
Binary Search Tree



Definition: A Binary Tree **T** storing values is said to be Binary Search Tree if for each node **v** in T

- If **left(v)** \neq NULL, then **value(v)** > **value** of every node in **subtree(left(v))**.
- If **right(v)** \neq NULL, then **value(v)** < **value** of every node in **subtree(right(v))**.

Binary Search Tree: a slight change



This transformation is merely to help us in the analysis of **red-black** trees. It does not cause any extra overhead of space. All **NULL nodes** correspond to a single node in memory.

Henceforth, for each **NULL child link** of a node in a **BST**, we create a **NULL node**

- ➔ 1. Each **leaf node** in a BST will be a **NULL node**.
- 2. the BST will always be a **full binary tree**.

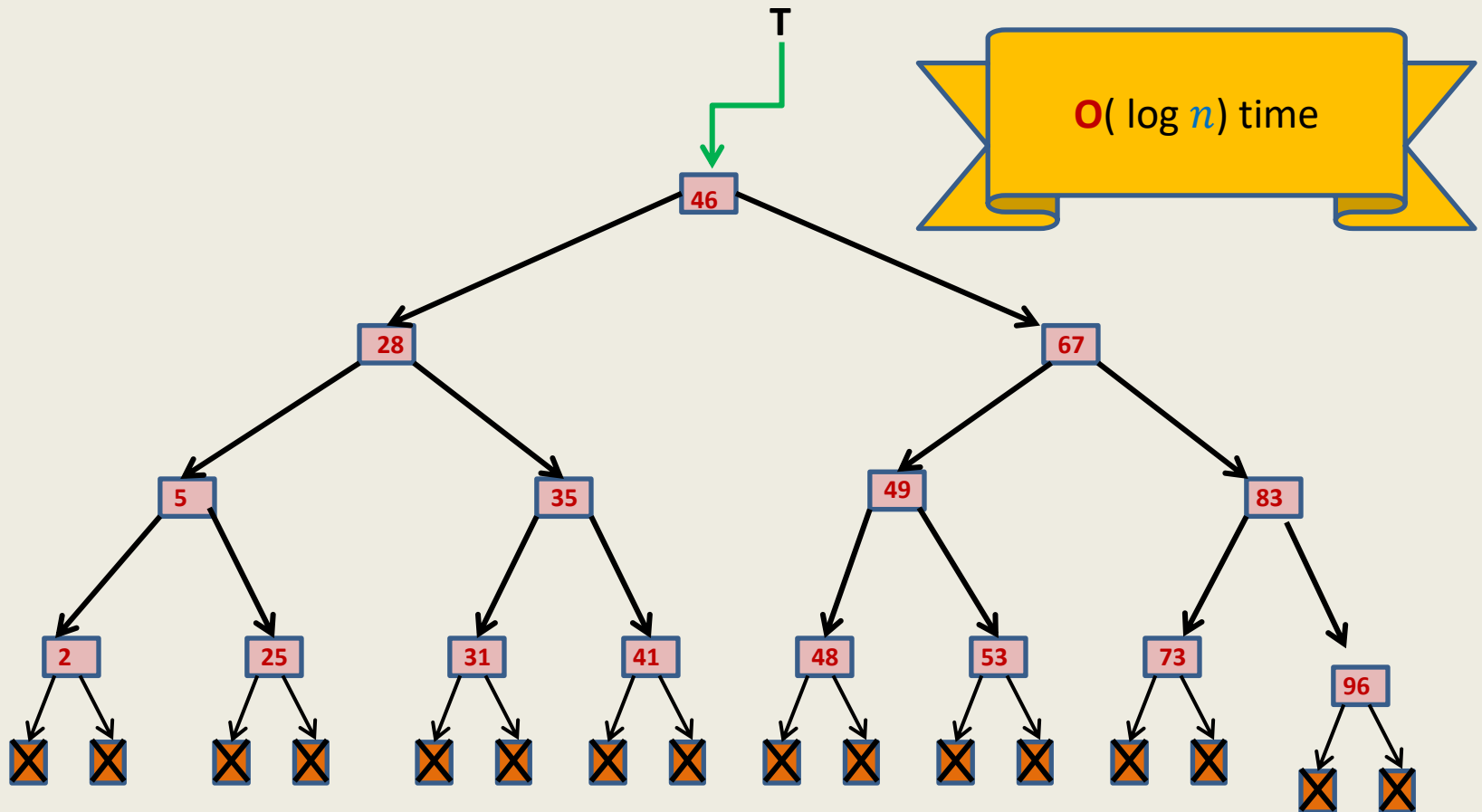
A fact we noticed in our previous discussion on BSTs (Lecture 8)

Time complexity of $\text{Search}(T, x)$ and $\text{Insert}(T, x)$ in a Binary Search Tree $T = O(\text{Height}(T))$

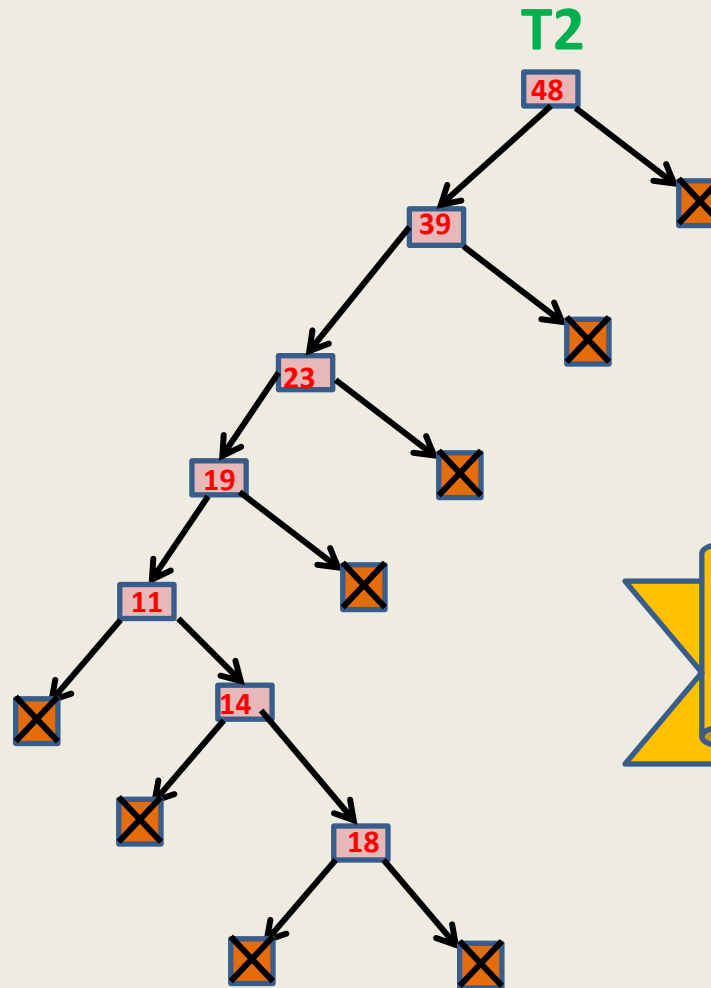
Height(T):

The maximum number of nodes on any path from root to a leaf node.

Searching and inserting in a **perfectly balanced BST**



Searching and inserting in a **skewed** BST on n nodes



$O(n)$ time !!

Nearly balanced Binary Search Tree

Terminology:

size of a binary tree is the number of nodes present in it.

Definition: A binary search tree **T** is said to be nearly balanced at node **v**, if

$$\text{size}(\text{left}(\mathbf{v})) \leq \frac{3}{4} \text{size}(\mathbf{v})$$

and

$$\text{size}(\text{right}(\mathbf{v})) \leq \frac{3}{4} \text{size}(\mathbf{v})$$

Definition: A binary search tree **T** is said to be **nearly balanced** if it is nearly balanced at each node.

Nearly balanced Binary Search Tree

- **Search**(**T**,**x**) operation is the same.
- Modify **Insert**(**T**,**x**) operation as follows:
 - Carry out normal insert and update the **size** fields of nodes traversed.
 - If **BST T** ceases to be **nearly imbalanced** at any node **v**,
transform **subtree(v)** into **perfectly balanced BST**.

→ $O(\log n)$ time for **search**

→ $O(n \log n)$ time for **n insertions**

Disadvantages:

- How to handle **deletions** ?
- Some insertions may take $O(n)$ time 😞

This fact will be proved soon.
However, you would have
verified it experimentally
through **Assignment 2**

Can we achieve $O(\log n)$ time for
search/insert/delete ?

- AVL Trees [1962]

- Red Black Trees [1978] 

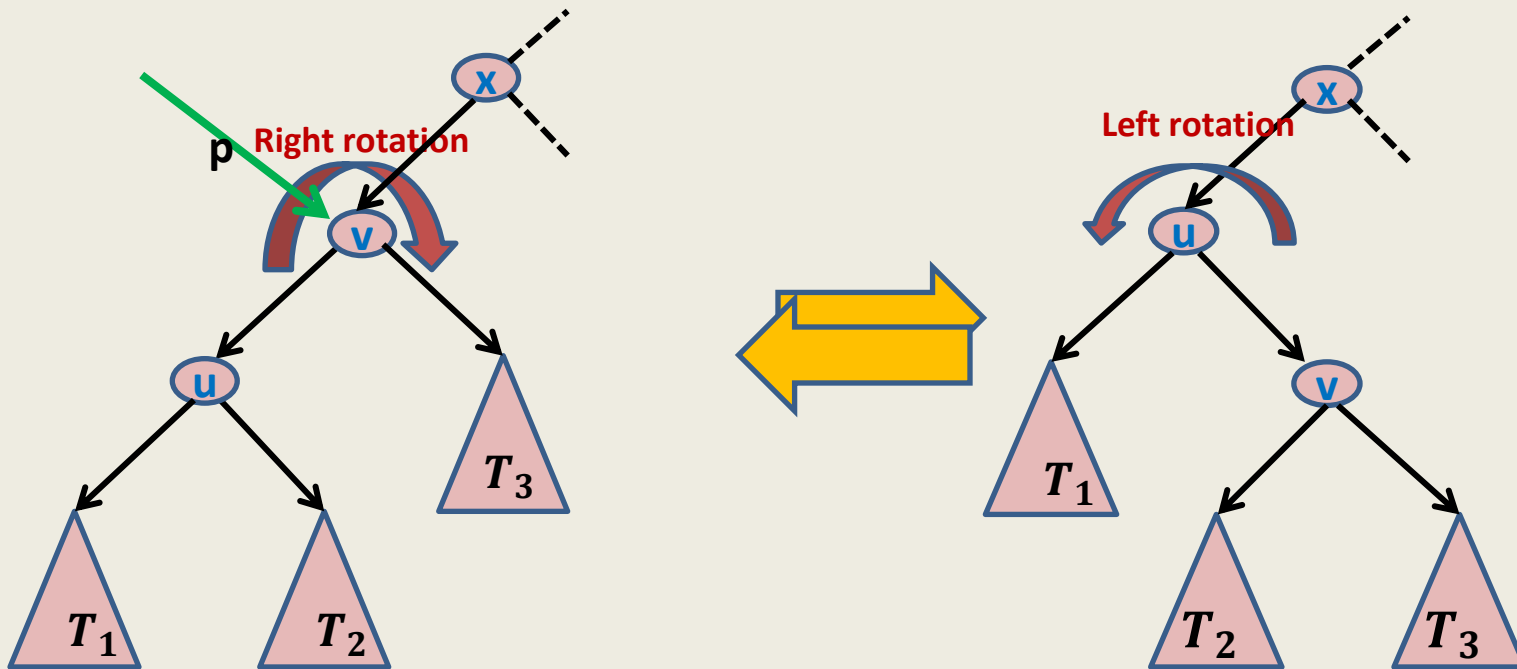
Rotation around a node

An important tool for balancing trees

Each height balanced **BST** employs this tool which is derived from the **flexibility** which is hidden in the structure of a **BST**.

This flexibility (**pointer manipulation**) was inherited from linked list 😊.

Rotation around a node



Note that the tree T continues to remain a BST even after rotation around any node.

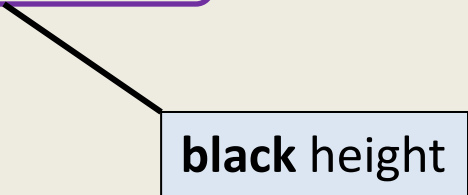
Red Black Tree

A height balanced BST

Red Black Tree

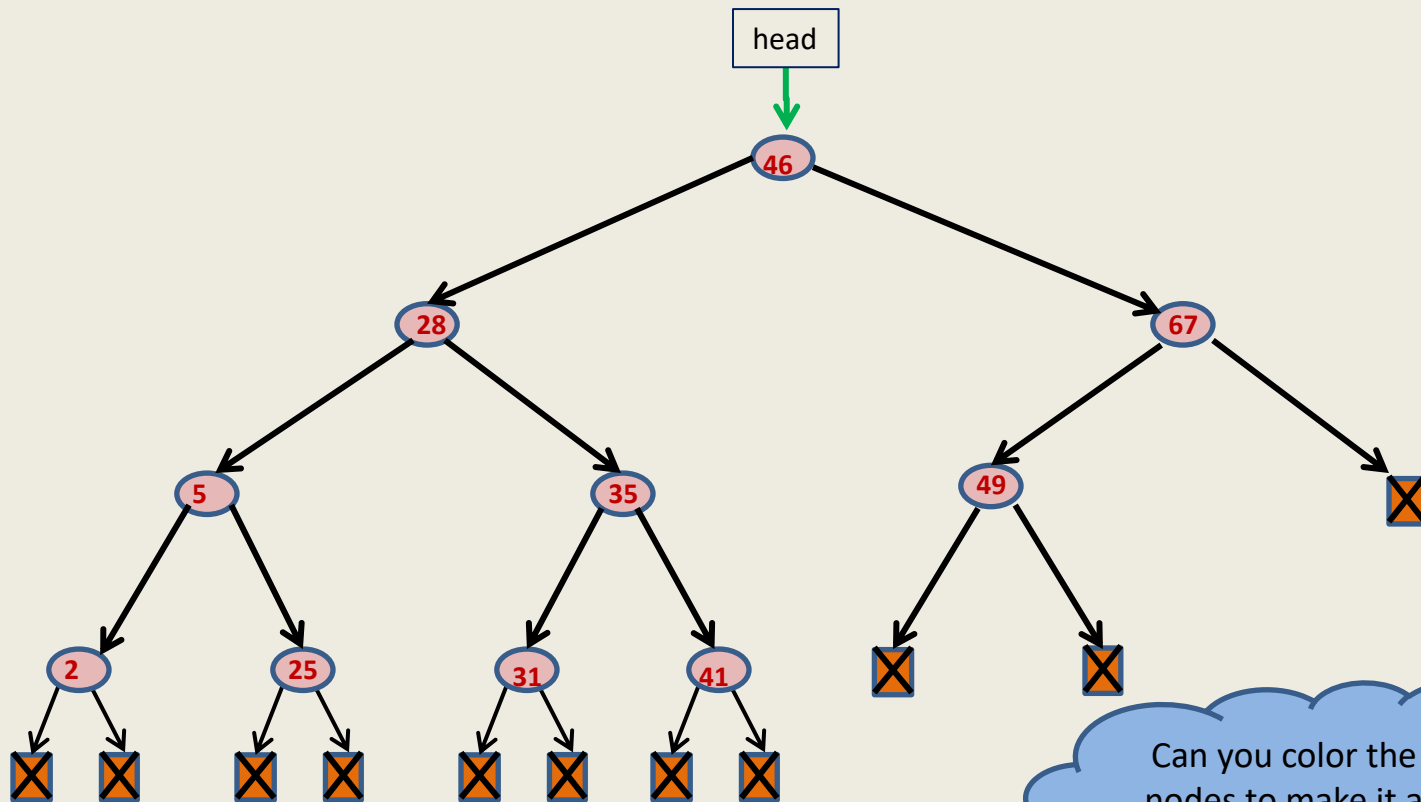
Red-Black tree is a binary search tree satisfying the following properties:

- 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.



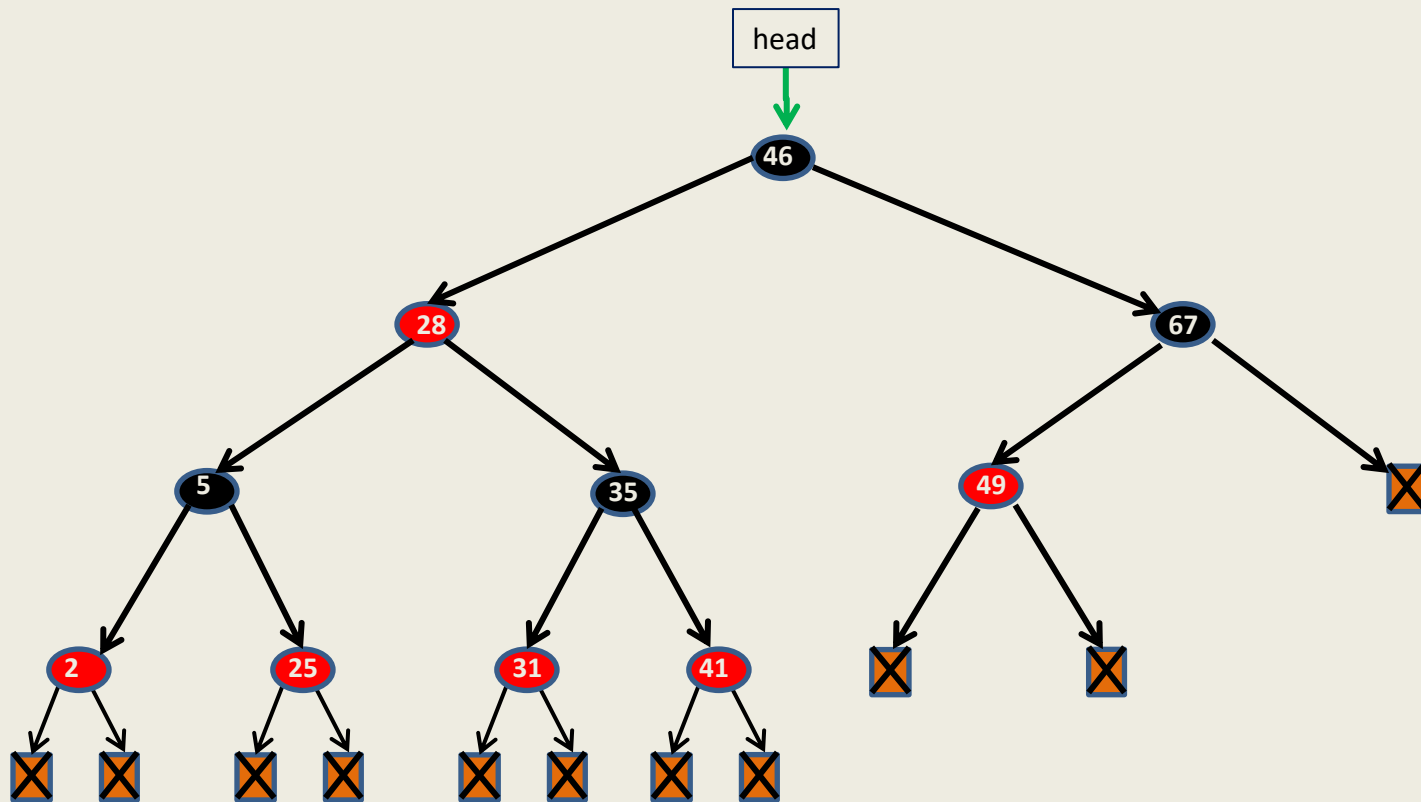
black height

A binary search tree



Can you color the nodes to make it a red-black tree ?

A binary search tree



A Red Black Tree

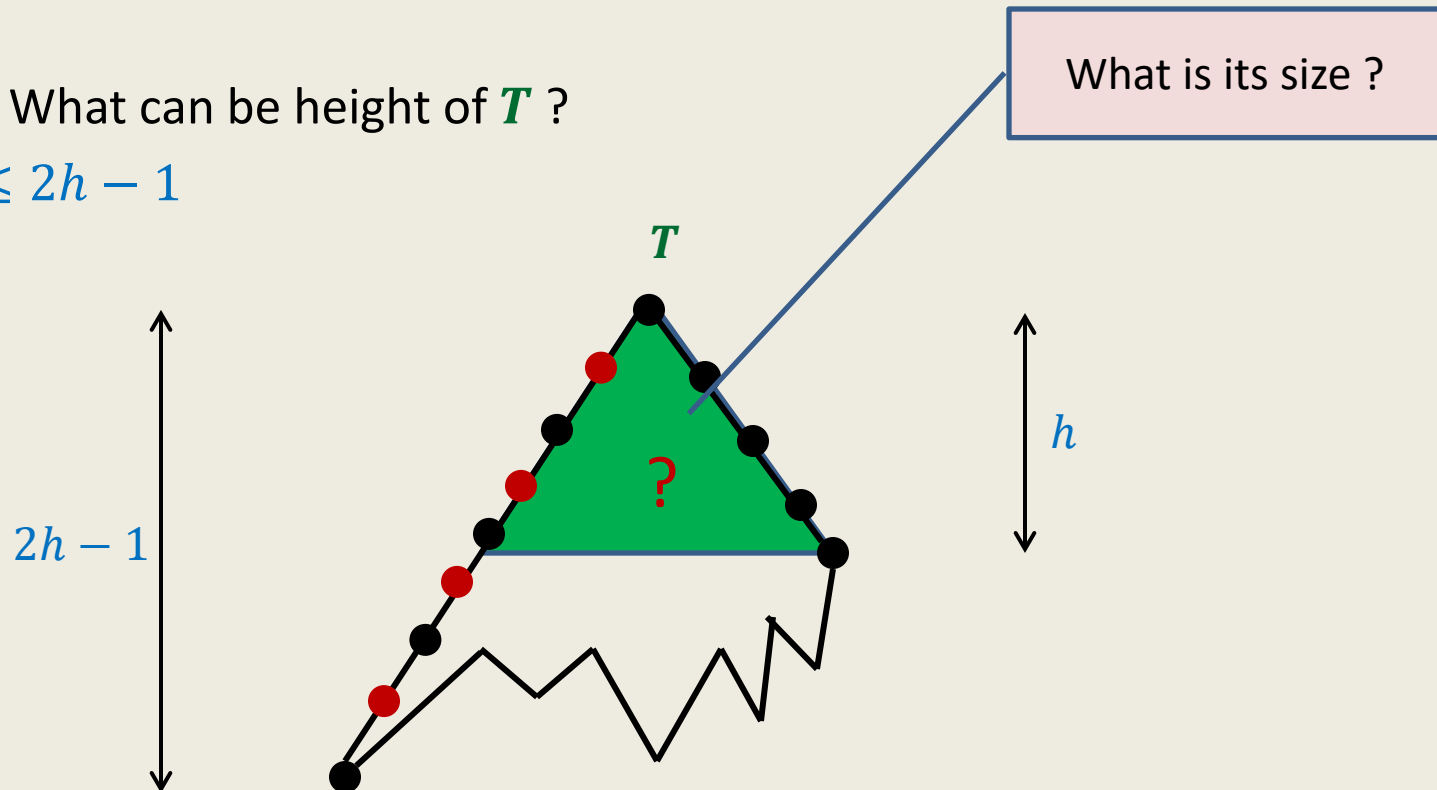
Why is a **red** black tree height balanced ?

T : a **red** black tree

h : **black** height of T .

Question: What can be height of T ?

Answer: $\leq 2h - 1$



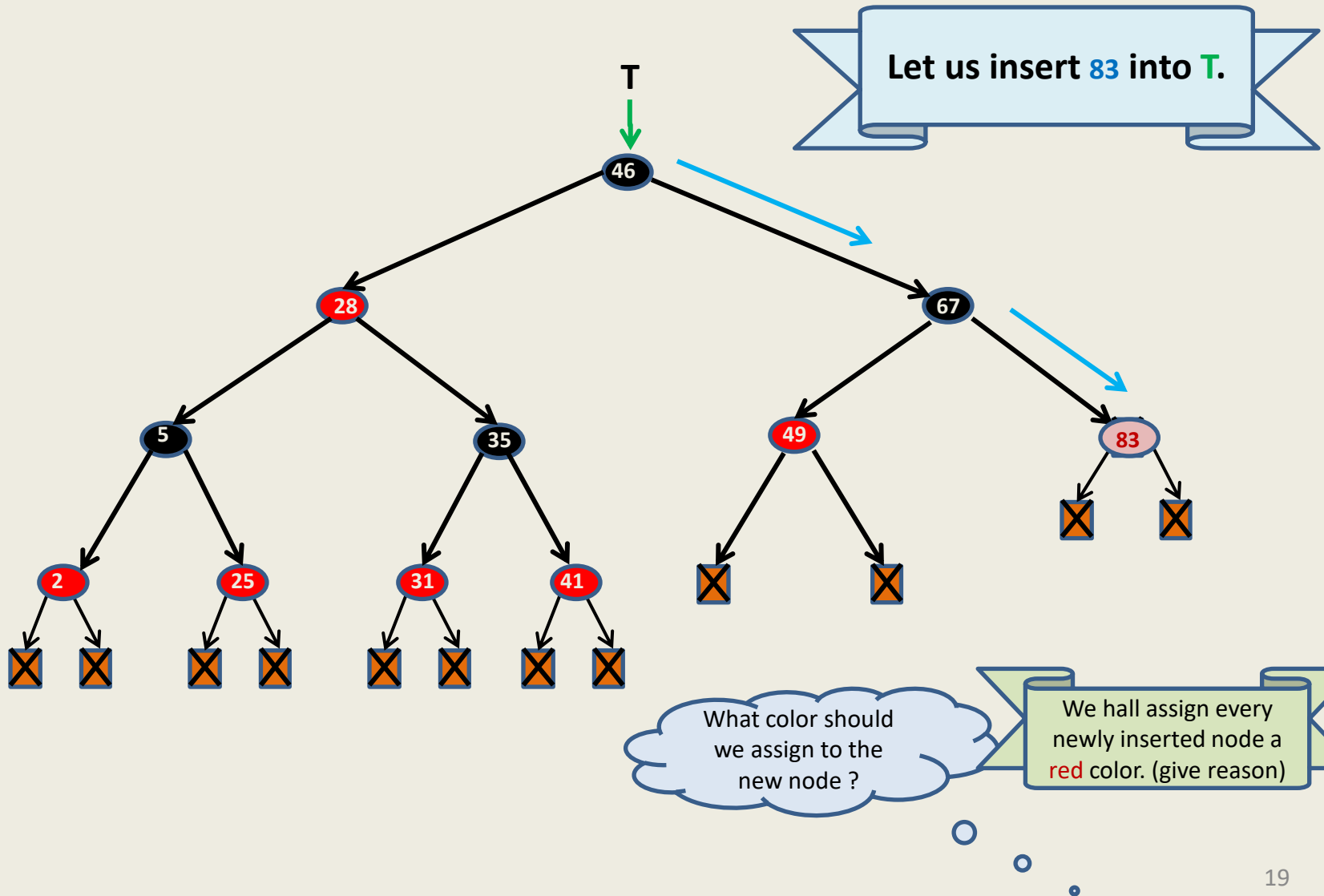
Homework: Ponder over the above hint to prove that T has $\geq 2^h$ elements.

Insertion in a Red Black tree

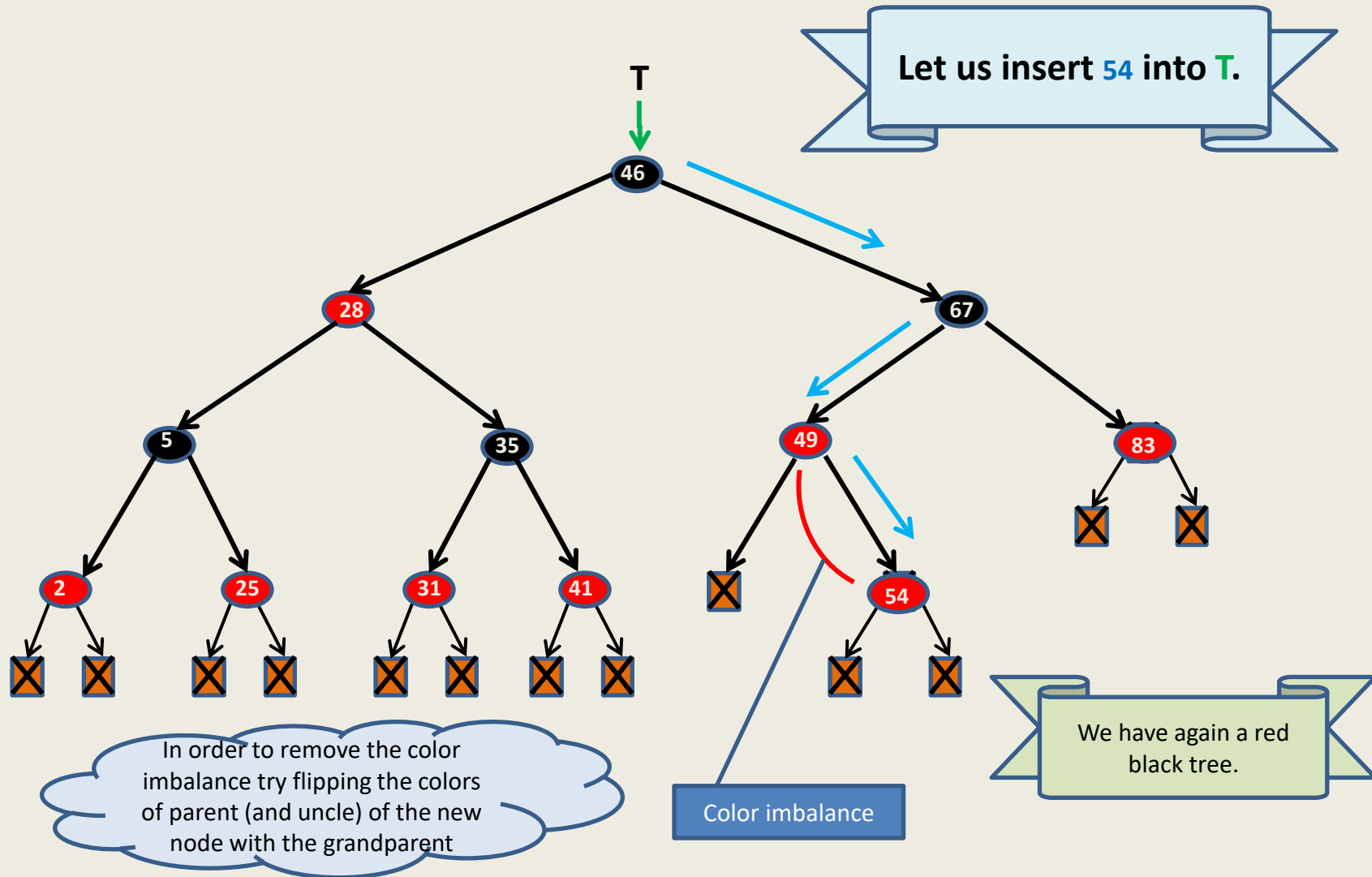
All it involves is

- playing with colors 😊
- and rotations 😊

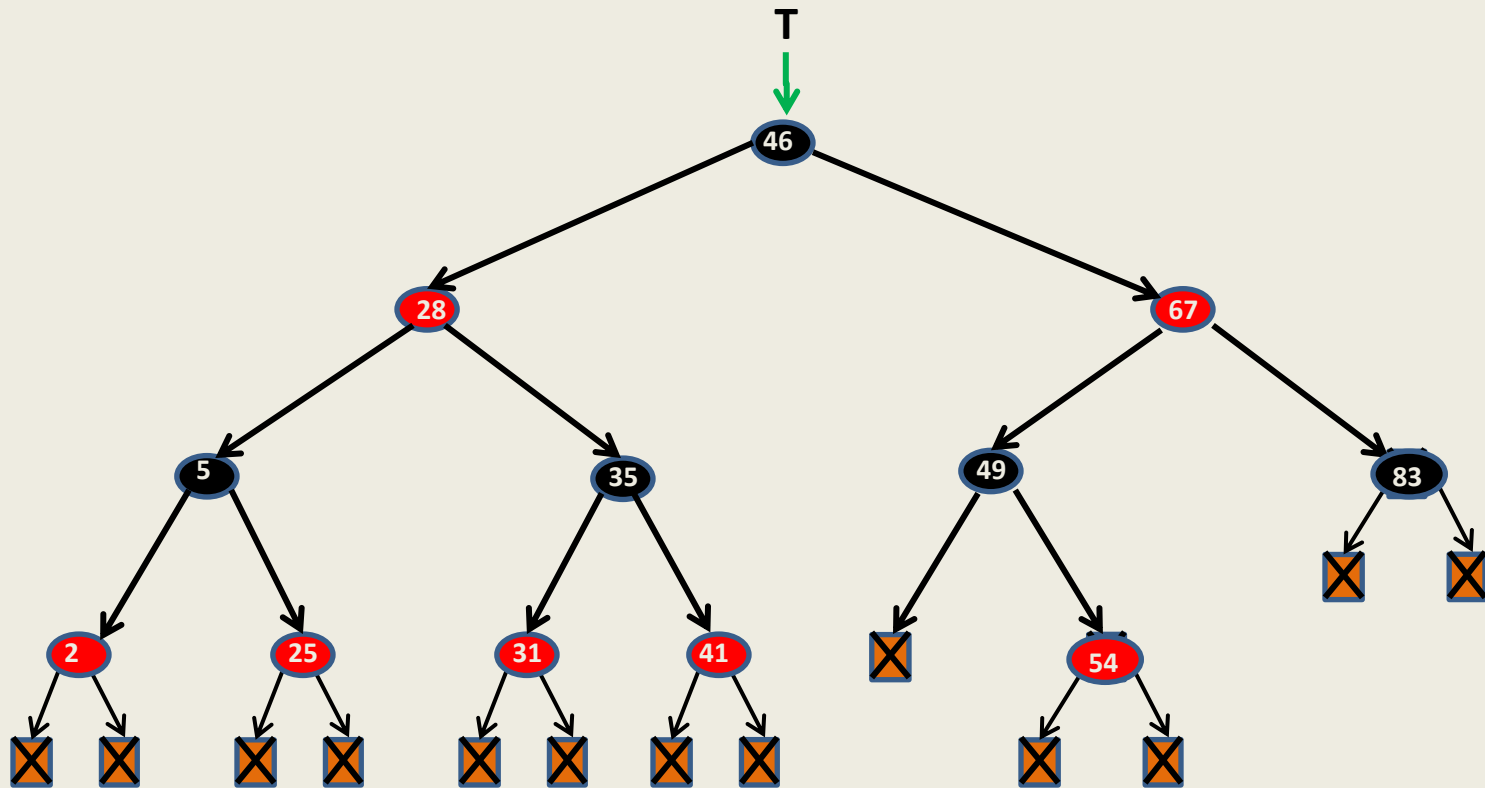
Insertion in a red-black tree



Insertion in a red-black tree

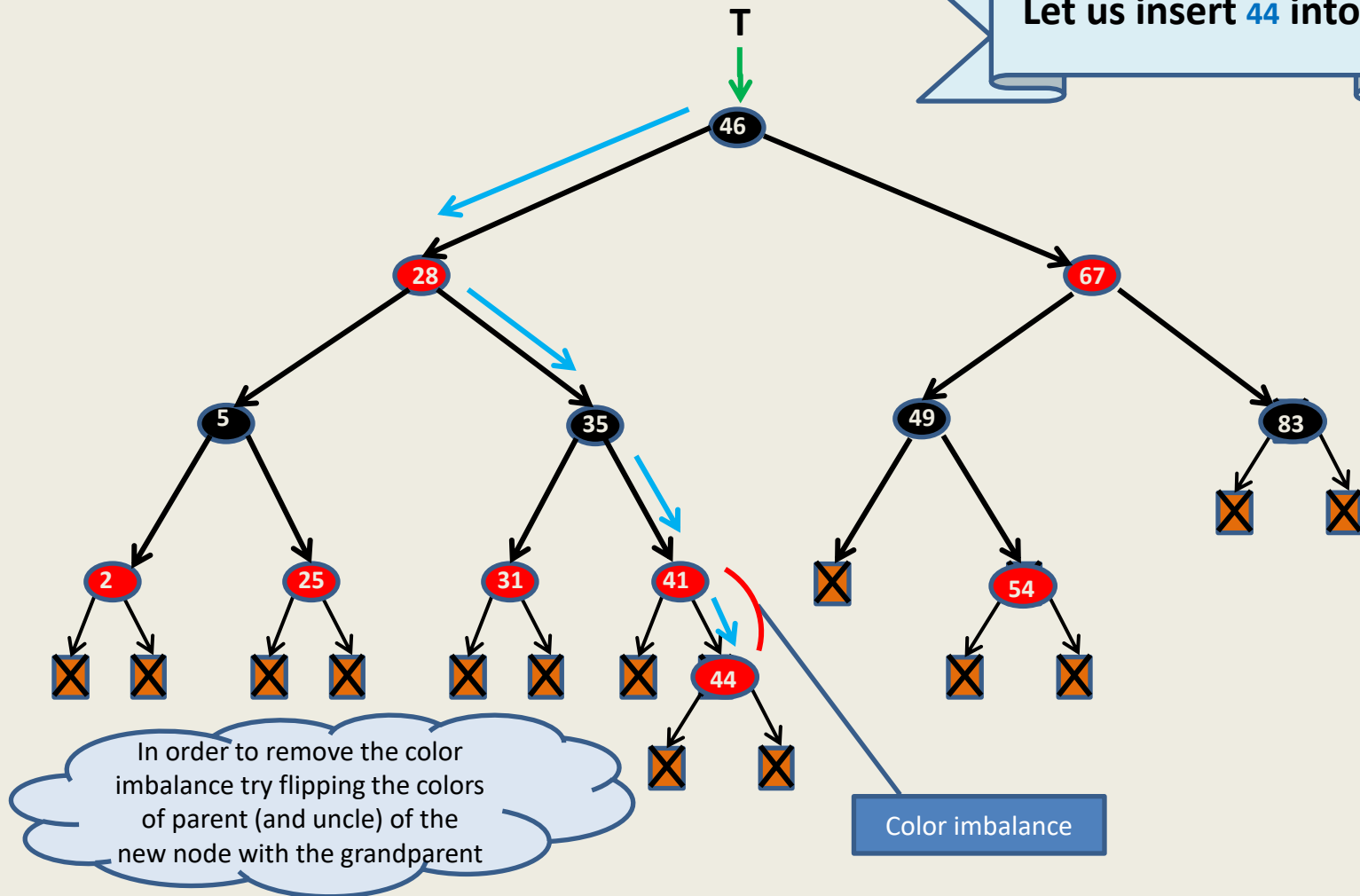


Insertion in a red-black tree

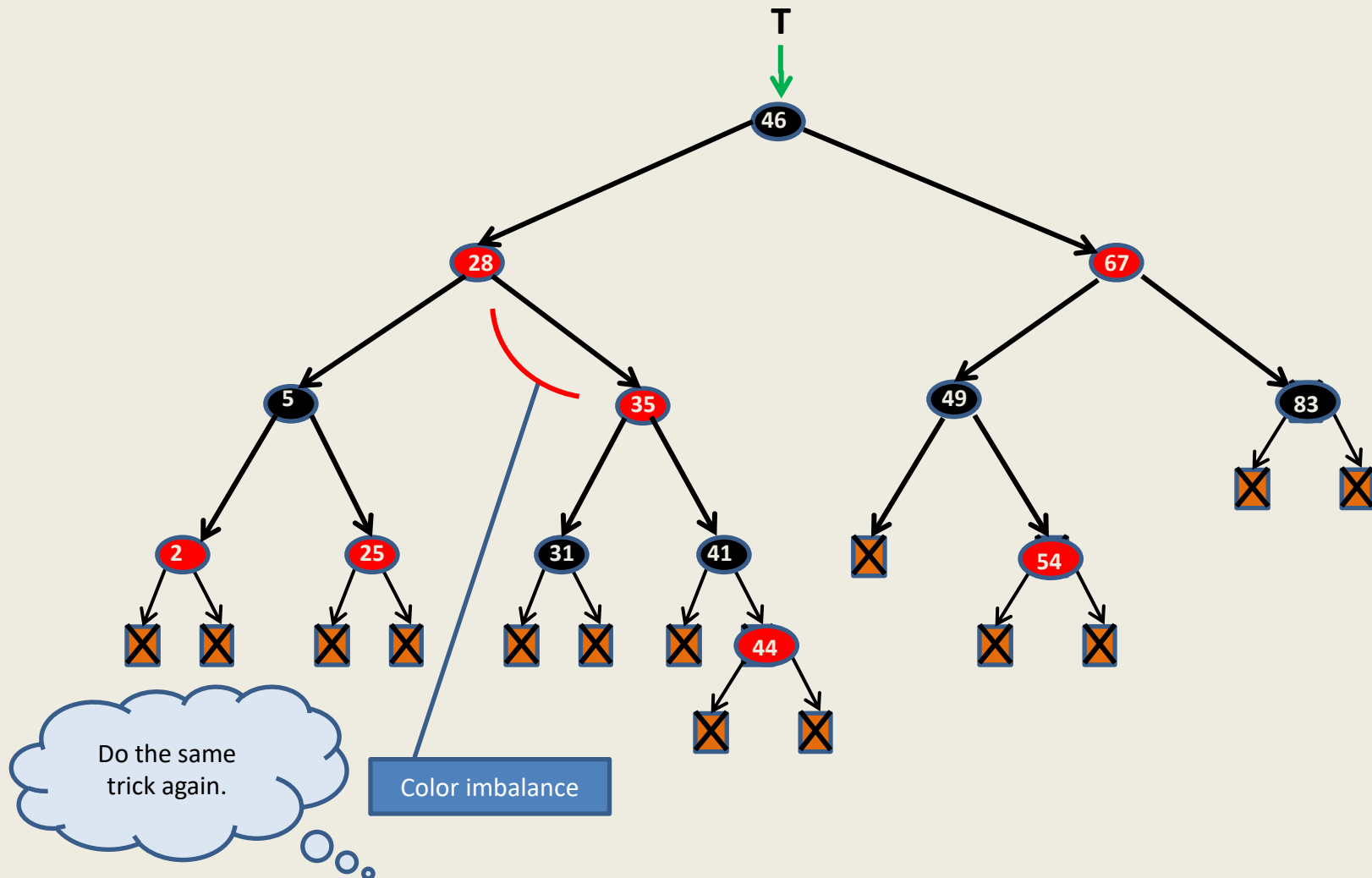


Insertion in a red-black tree

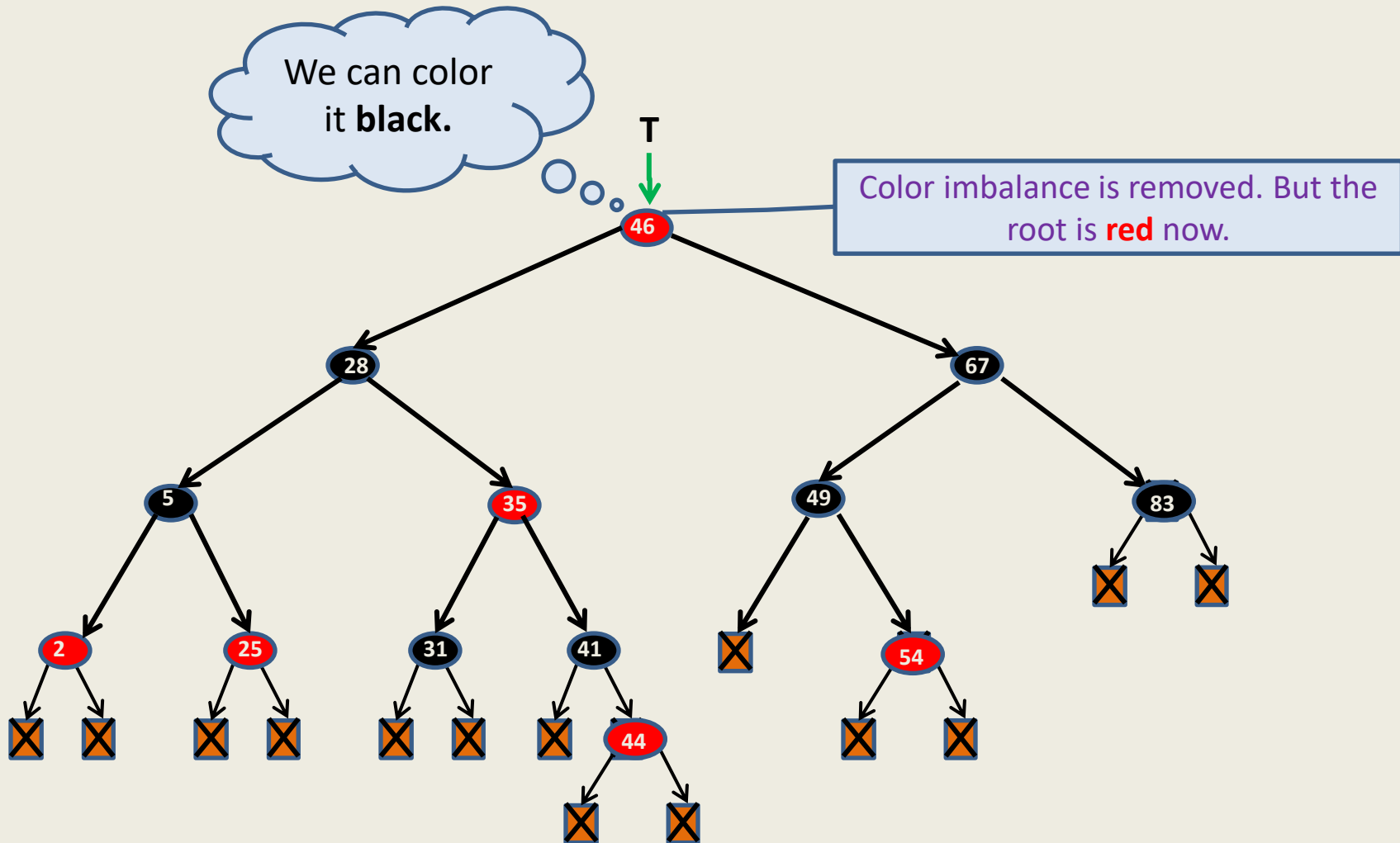
Let us insert 44 into T.



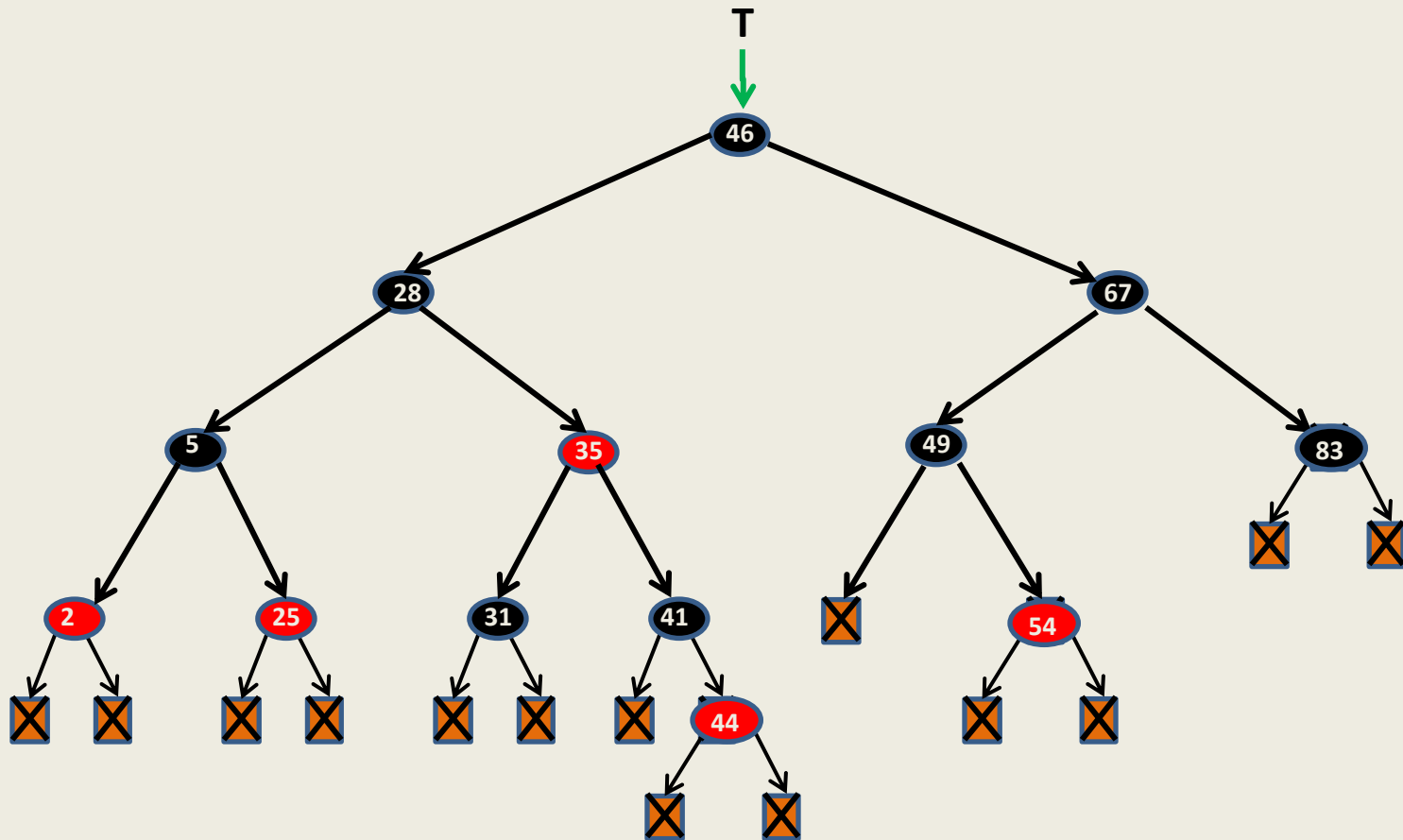
Insertion in a red-black tree



Insertion in a red-black tree



Insertion in a red-black tree



Insertion in a red-black tree

summary till now ...

Let p be the newly inserted node. Assign **red** color to p .

Case 1: $\text{parent}(p)$ is **black**

nothing needs to be done.

Case 2: $\text{parent}(p)$ is **red** and $\text{uncle}(p)$ is **red**,

Swap colors of parent (and uncle) with $\text{grandparent}(p)$.

This balances the color at p but may lead to imbalance of color at grandparent of p . So $p \leftarrow \text{grandparent}(p)$, and proceed upwards similarly.

If in this manner p becomes **root**, then we color it **black**.

Case 3: $\text{parent}(p)$ is **red** and $\text{uncle}(p)$ is **black**.

This is a nontrivial case. So we need some more tools

Handling case 3

Description of Case 3

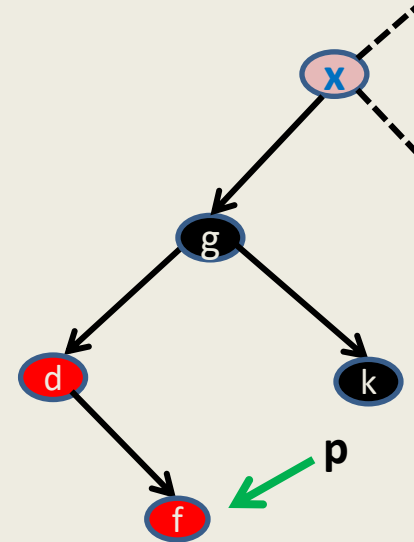
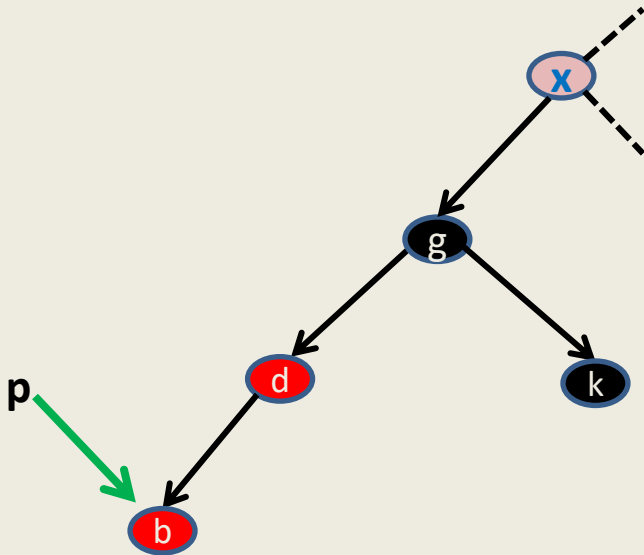
- p is a **red** colored node.
- $\text{parent}(p)$ is also **red**.
- $\text{uncle}(p)$ is **black**.

Without loss of generality assume: $\text{parent}(p)$ is *left child* of $\text{grandparent}(p)$.

(The case when $\text{parent}(p)$ is *right child* of $\text{grandparent}(p)$ is handled similarly.)

Handling the case 3

two cases arise depending upon whether p is left/right child of its parent



Can you transform
Case 3.2 to
Case 3.1 ?

Case 3.1:

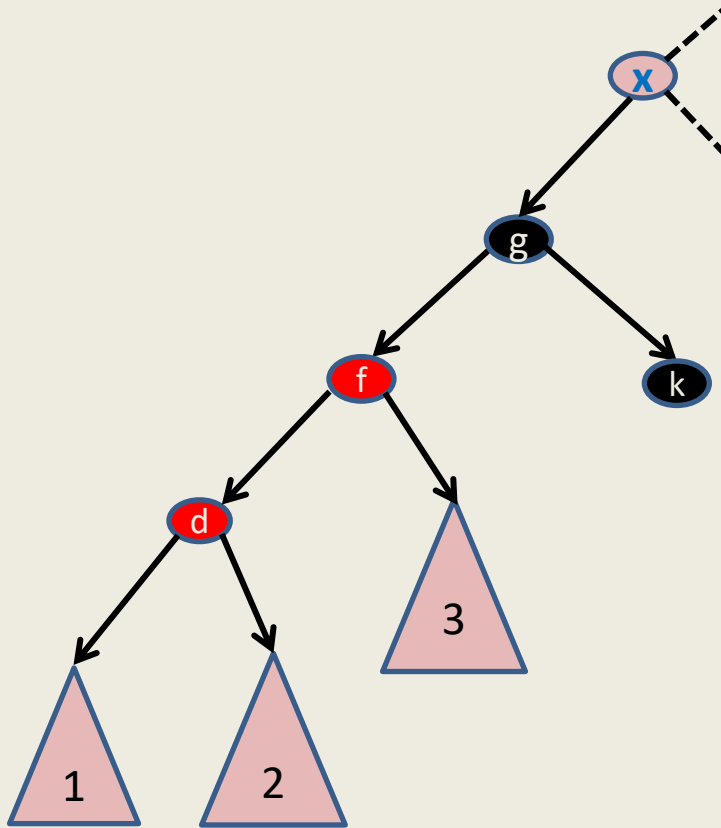
p is **left child** of its **parent**

Case 3.2:

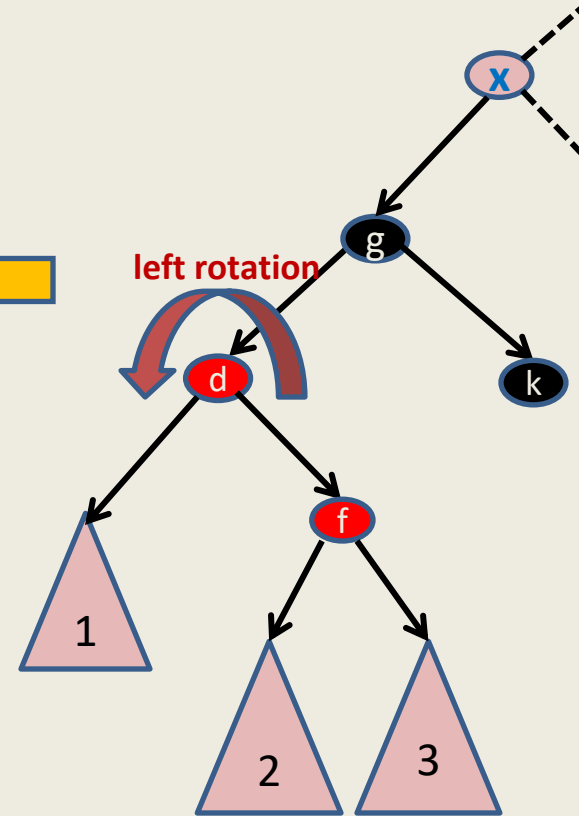
p is **right child** of its **parent**

Handling the case 3

two cases arise depending upon whether p is left/right child of its parent



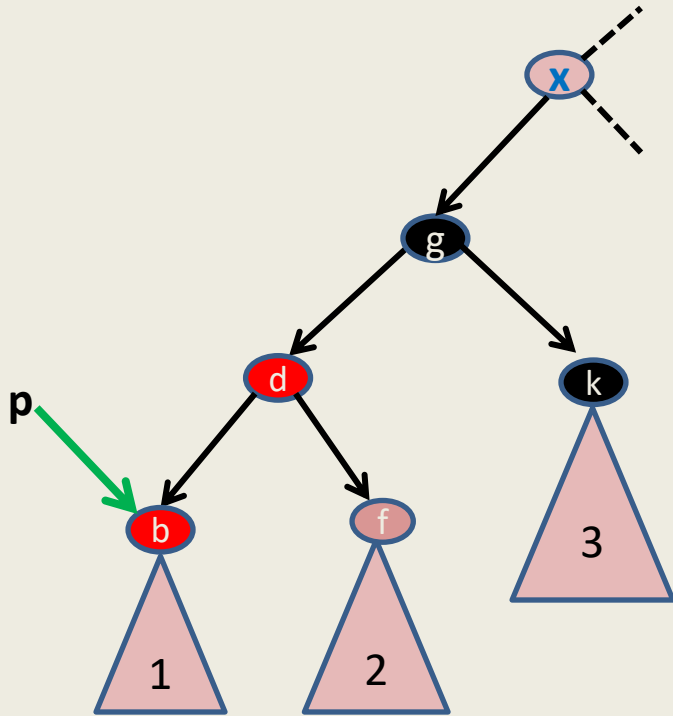
Vow!
This is exactly **Case 3.1**



Case 3.2:
p is **right** child of its parent

We need to handle only case 3.1

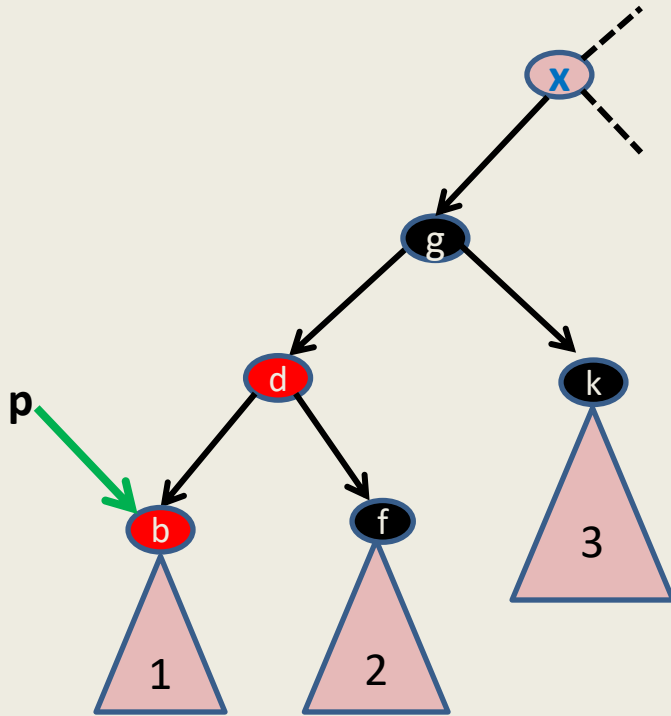
Handling the case 3.1



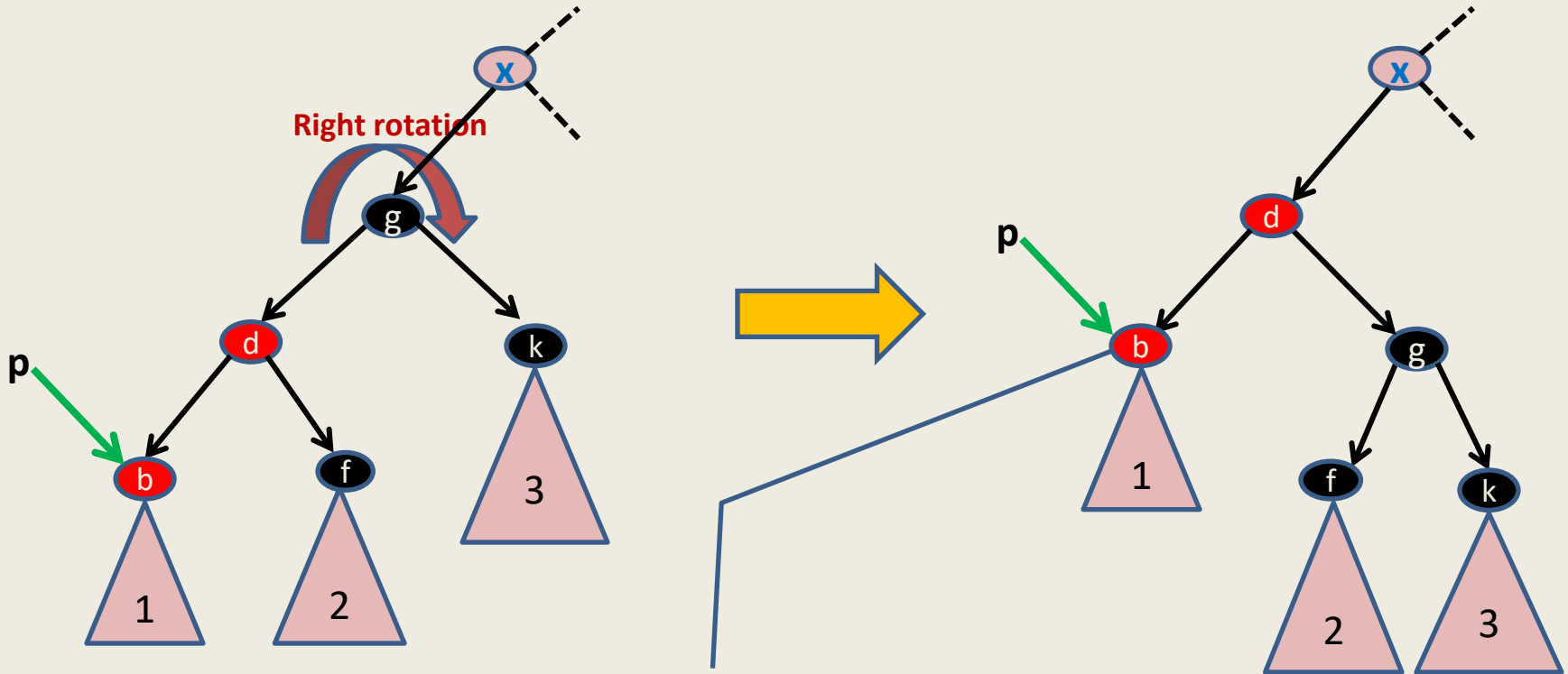
It has to be **black**.

Can we say anything about color of node f ?

Handling the case 3.1



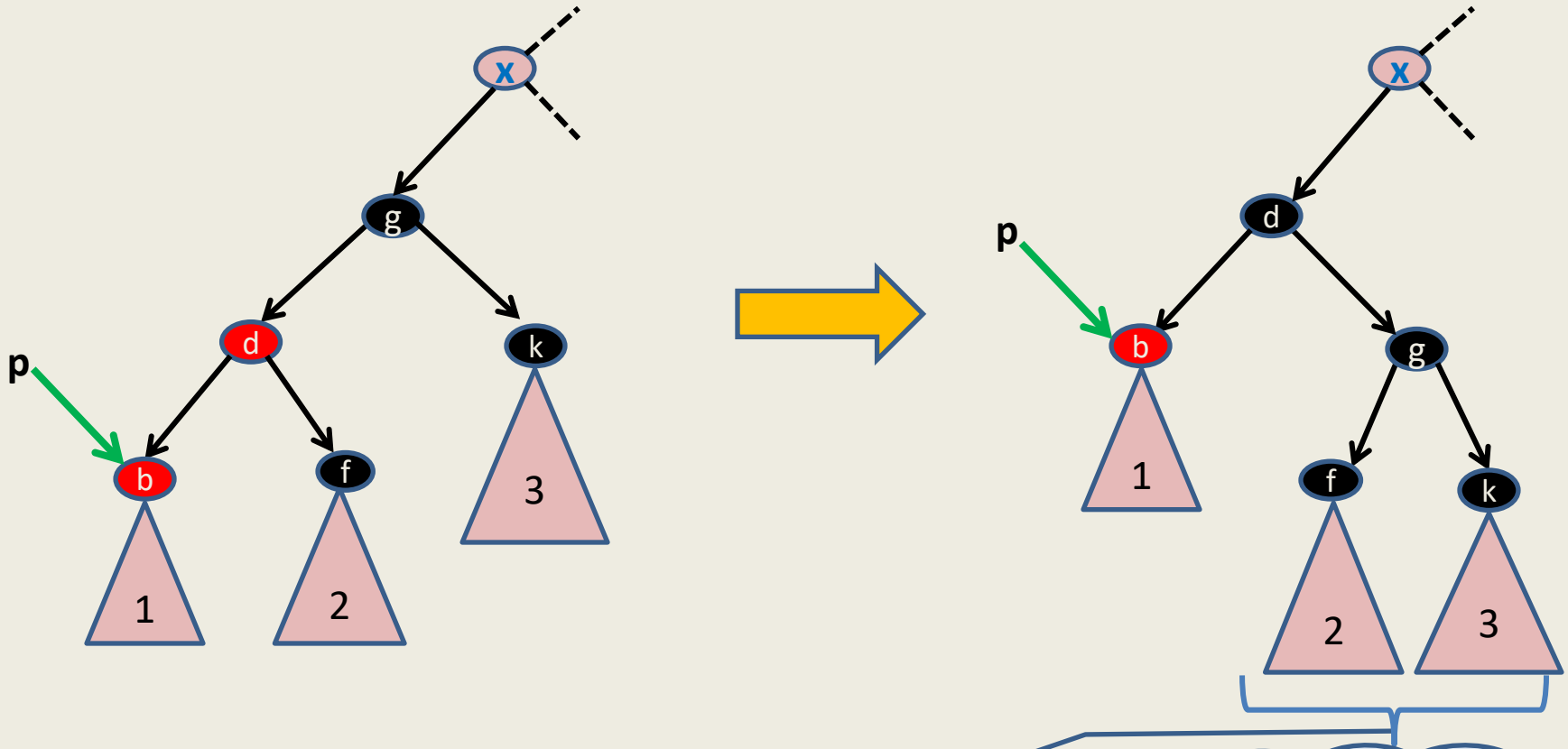
Handling the case 3.1



Now every node in tree 1 has one less **black** node on the path to root !
We must restore it. Moreover, the color imbalance exists even now.
What to do ?

Change color of
node d to **black**

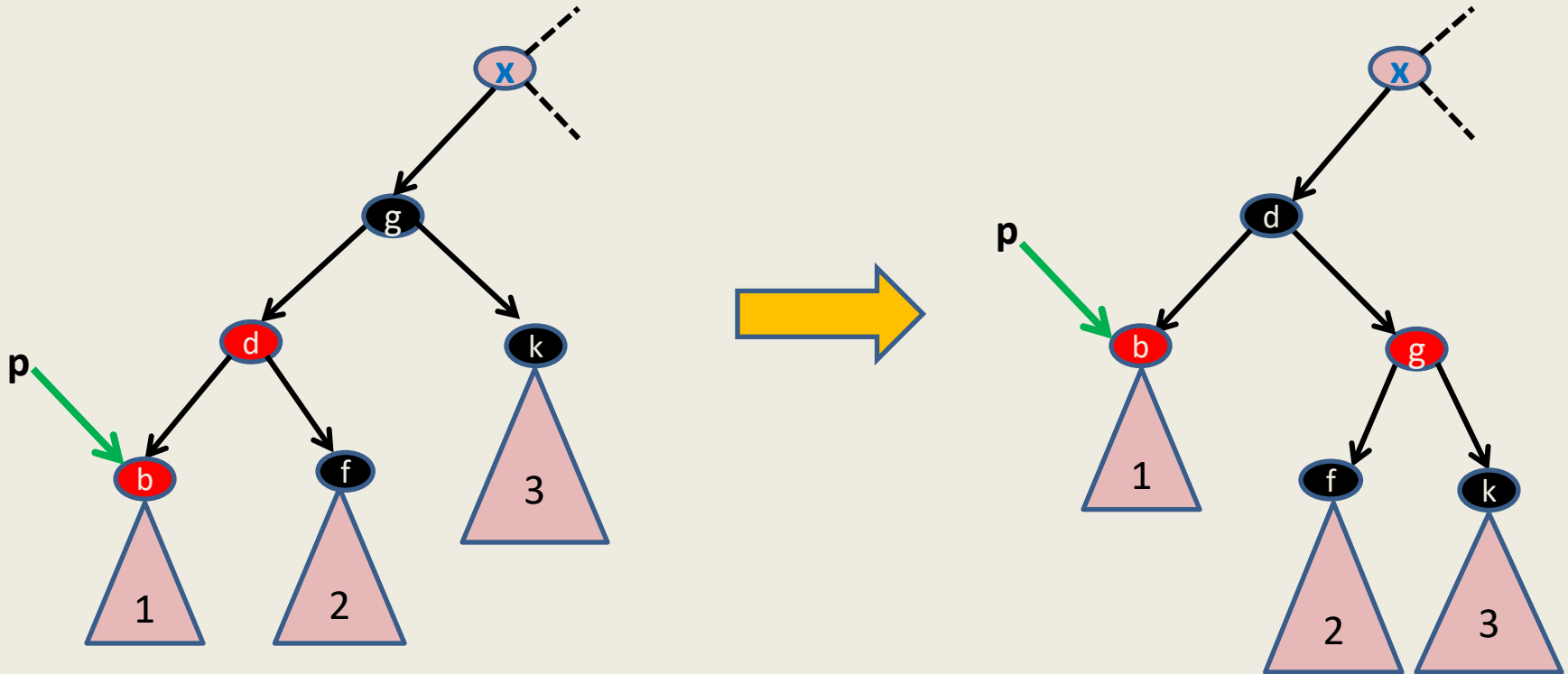
Handling the case 3.1



The number of **black** nodes on the path restored for tree 1. Color imbalance is restored. But the number of **black** nodes on the path increased by one for trees 2 and 3. What to do now ?

Color node g **red**

Handling the case 3.1



The black height is restored for all trees.
This completes **Case 3.1**

Theorem:

We can maintain **red-black** trees under insertion of nodes in $O(\log n)$ time per insert/search operation where n is the number of the nodes in the tree.

I hope you enjoyed the real fun in handling insertion in a **red black** tree.

How do will we handle deletion ?

This is going to be a bit more complex.

So please try on your own first before coming to the next class.

It will still involve playing with colors and rotations 😊