Data Structures and Algorithms

(ESO207)

Lecture 17:

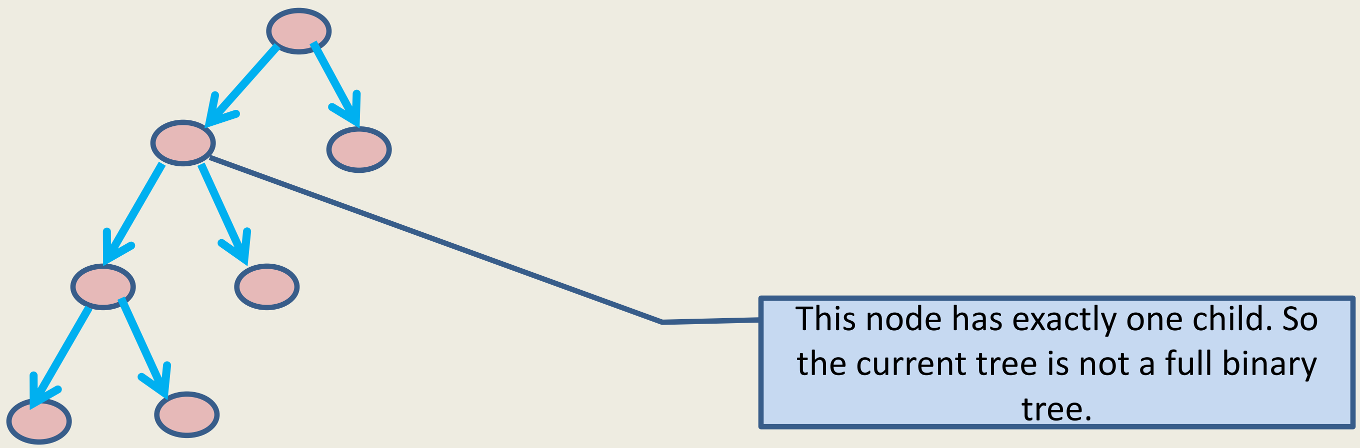
Height balanced BST

- Red-black trees

Terminologies

Full binary tree:

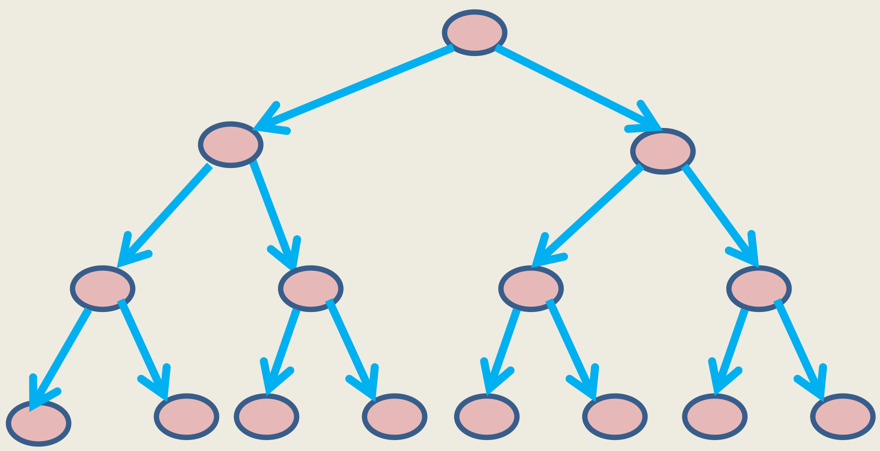
A binary tree where every internal node has <u>exactly two children.</u>



Terminologies

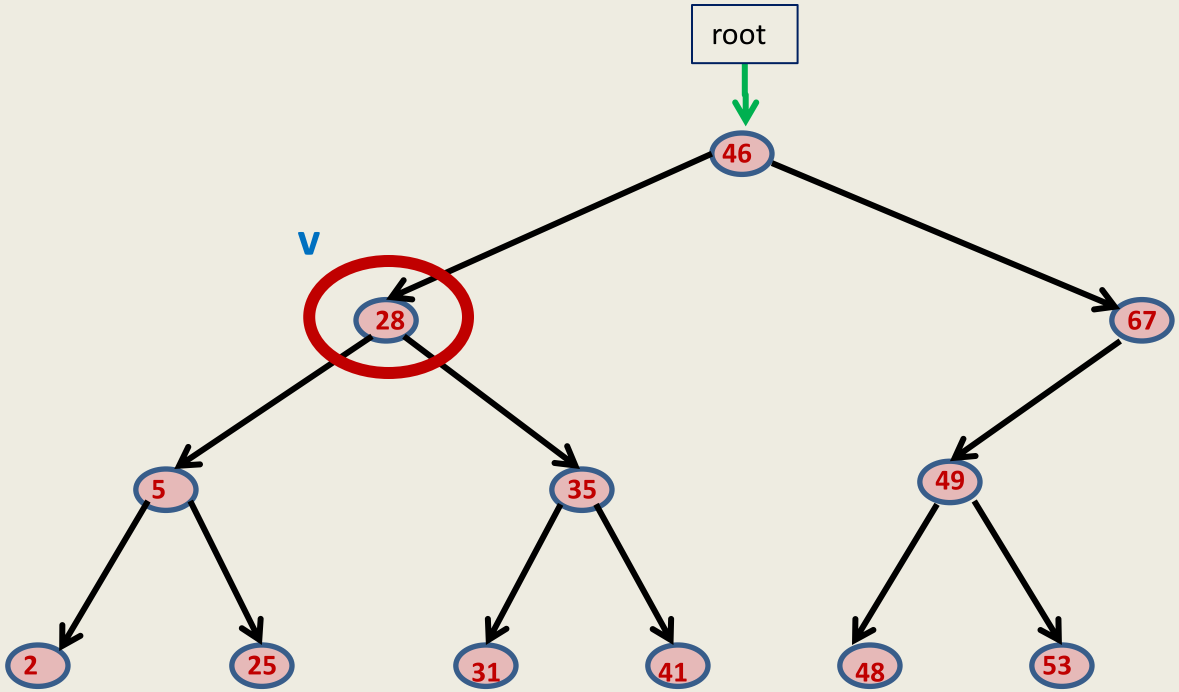
Complete binary tree:

A full binary tree where every leaf node is at the **same level**.



We shall later extend this definition when we discuss "**Binary heap**".

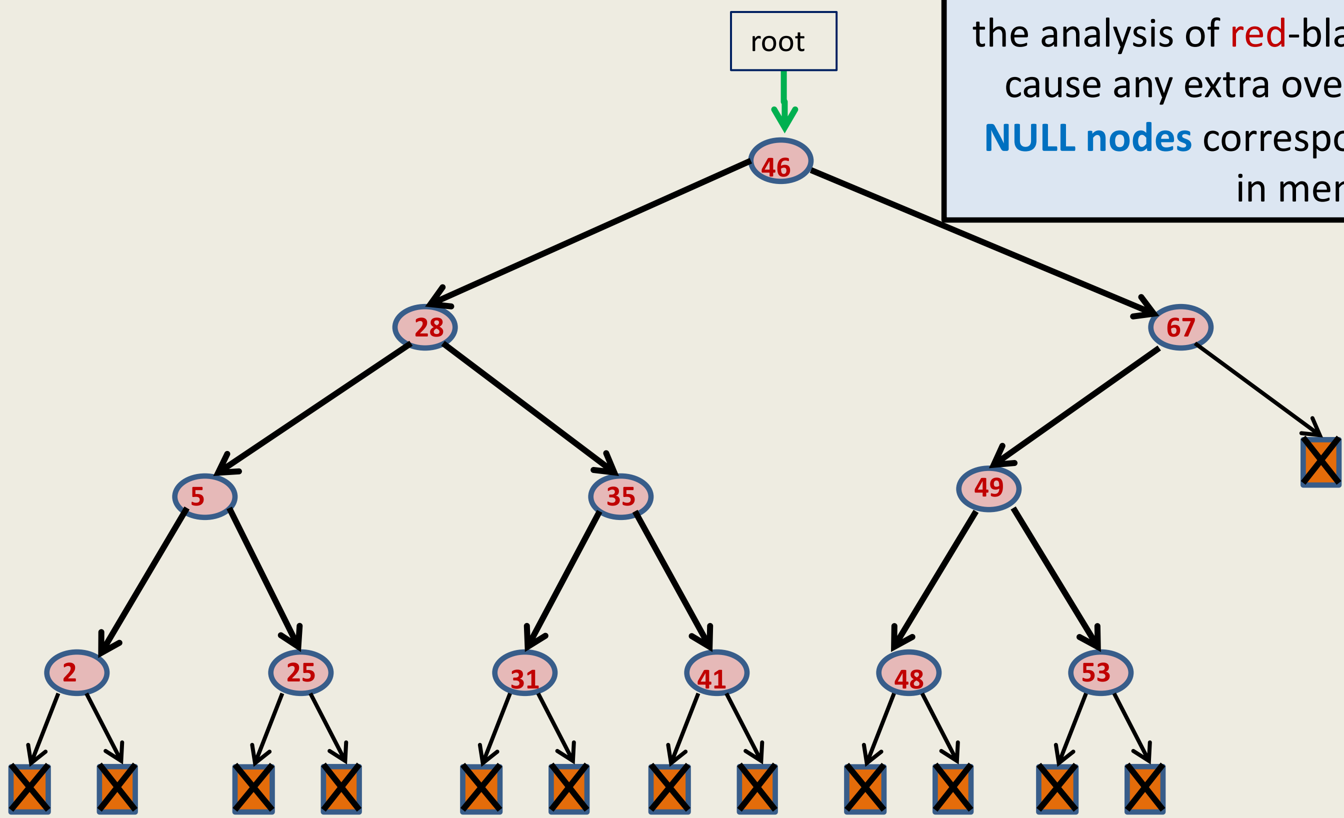
Binary Search Tree



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

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

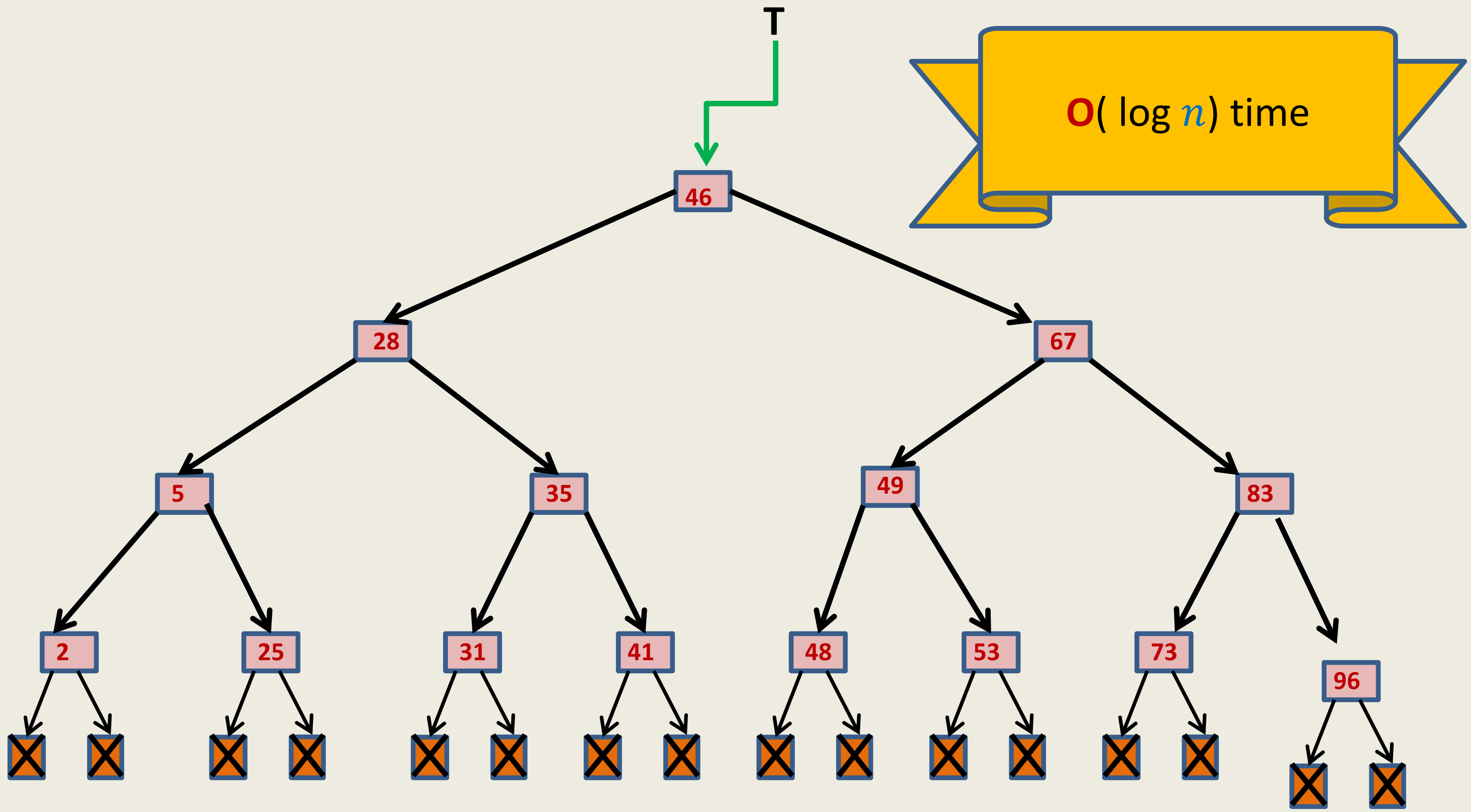
A fact we noticed in our previous discussion on BSTs (Lecture 9)

Time complexity of Search(T,x) and Insert(T,x) in a Binary Search Tree T = **O(Height(T))**

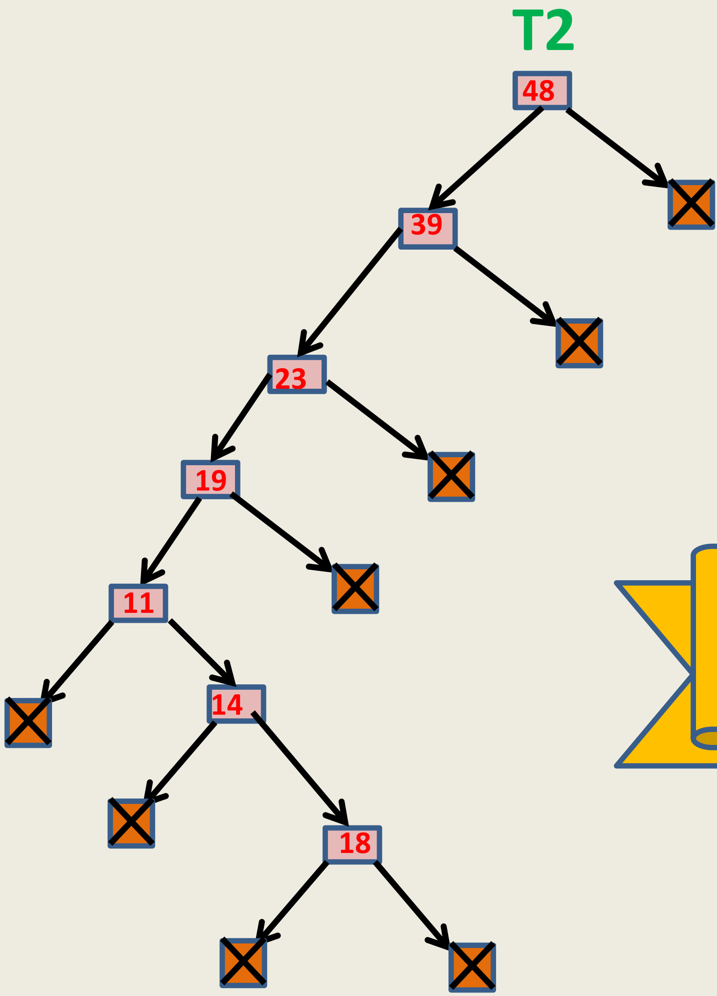
Height(*T*):

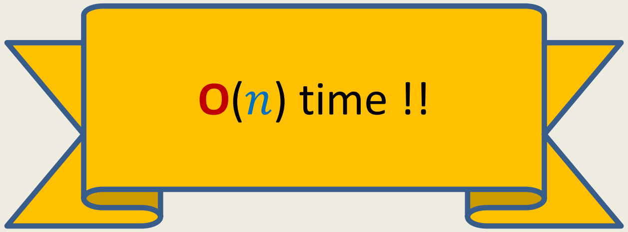
The maximum number of nodes on any path from root to a leaf node.

Searching and inserting in a perfectly balanced BST



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





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

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

and

size(**right**(**v**)) $\leq \frac{3}{4}$ **size**(**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 is 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**

This fact will be proved soon in a subsequent lecture.

Disadvantages:

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

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

- AVL Trees [1962]
- Red Black Trees [1978]

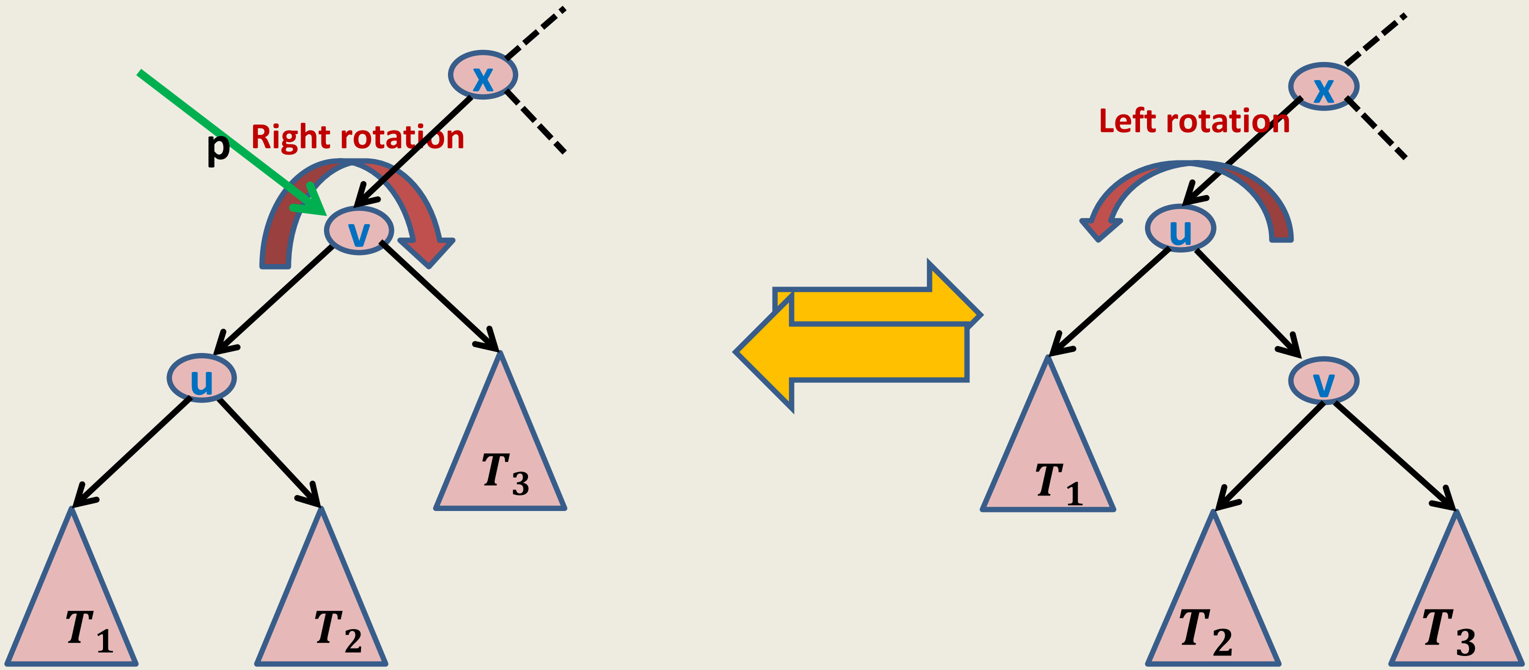
Rotation around a node

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

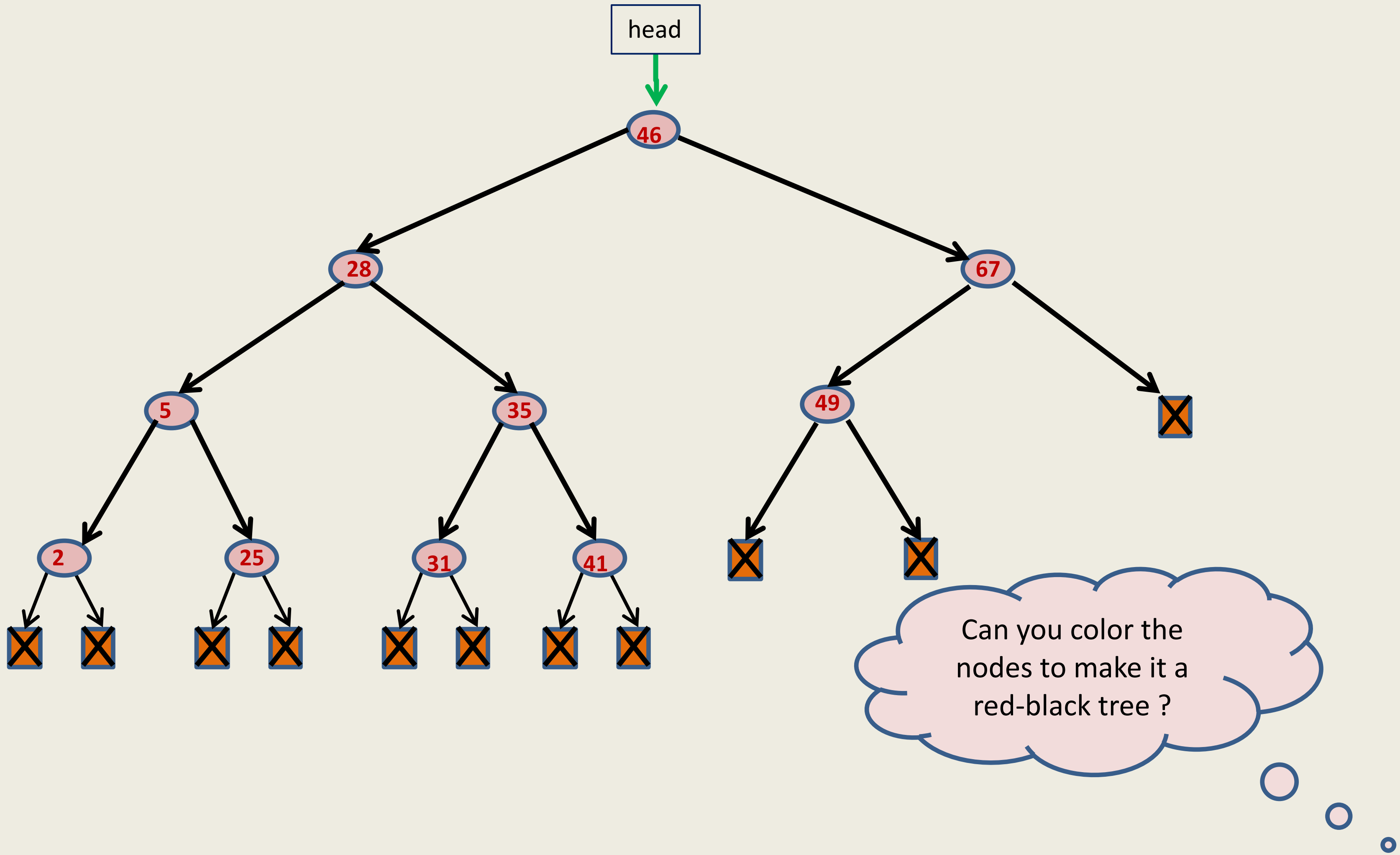
Red **Black** Tree

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

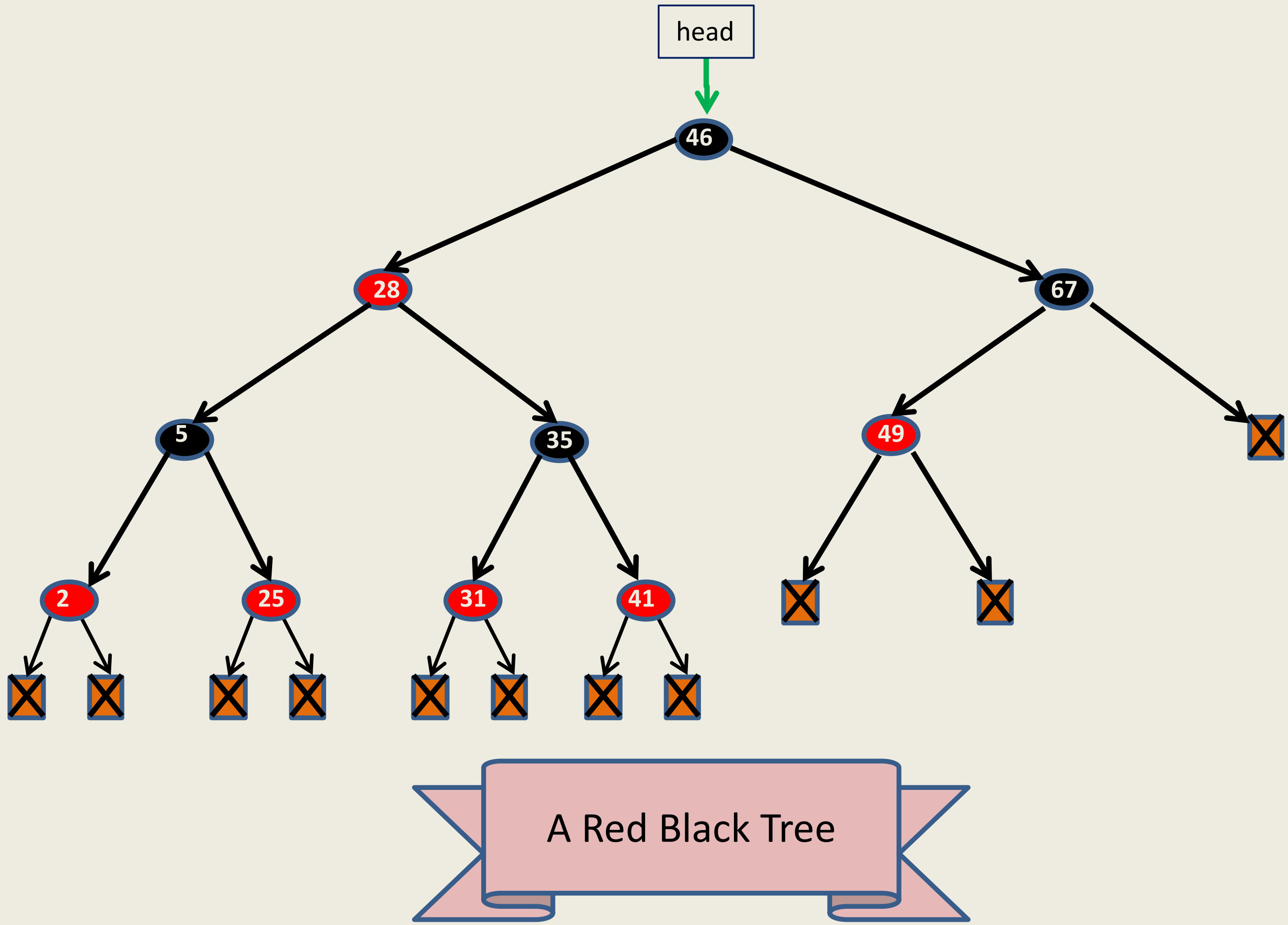
- Each node is colored **red** or **black.**
- Each leaf is colored **black** and so is the root.
- Every **red** node will have both its children **black.**
- No. of **black** nodes on a path from root to each leaf node is same.

black height

A binary search tree



A binary search tree



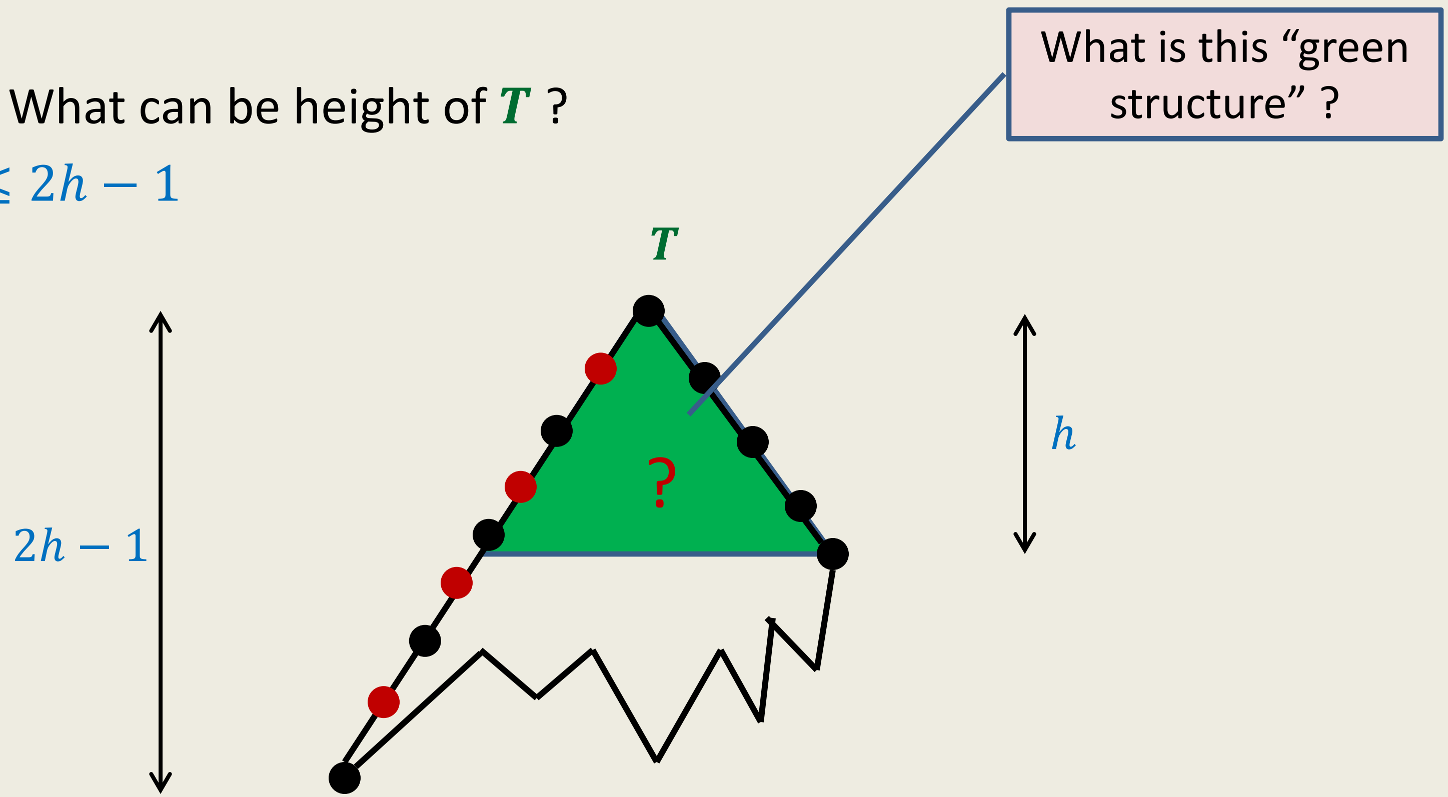
Why is a red black tree height balanced ?

T : a **red black tree**

h : **black** height of T.

Question: What can be height of T ?

Answer: $\leq 2h-1$



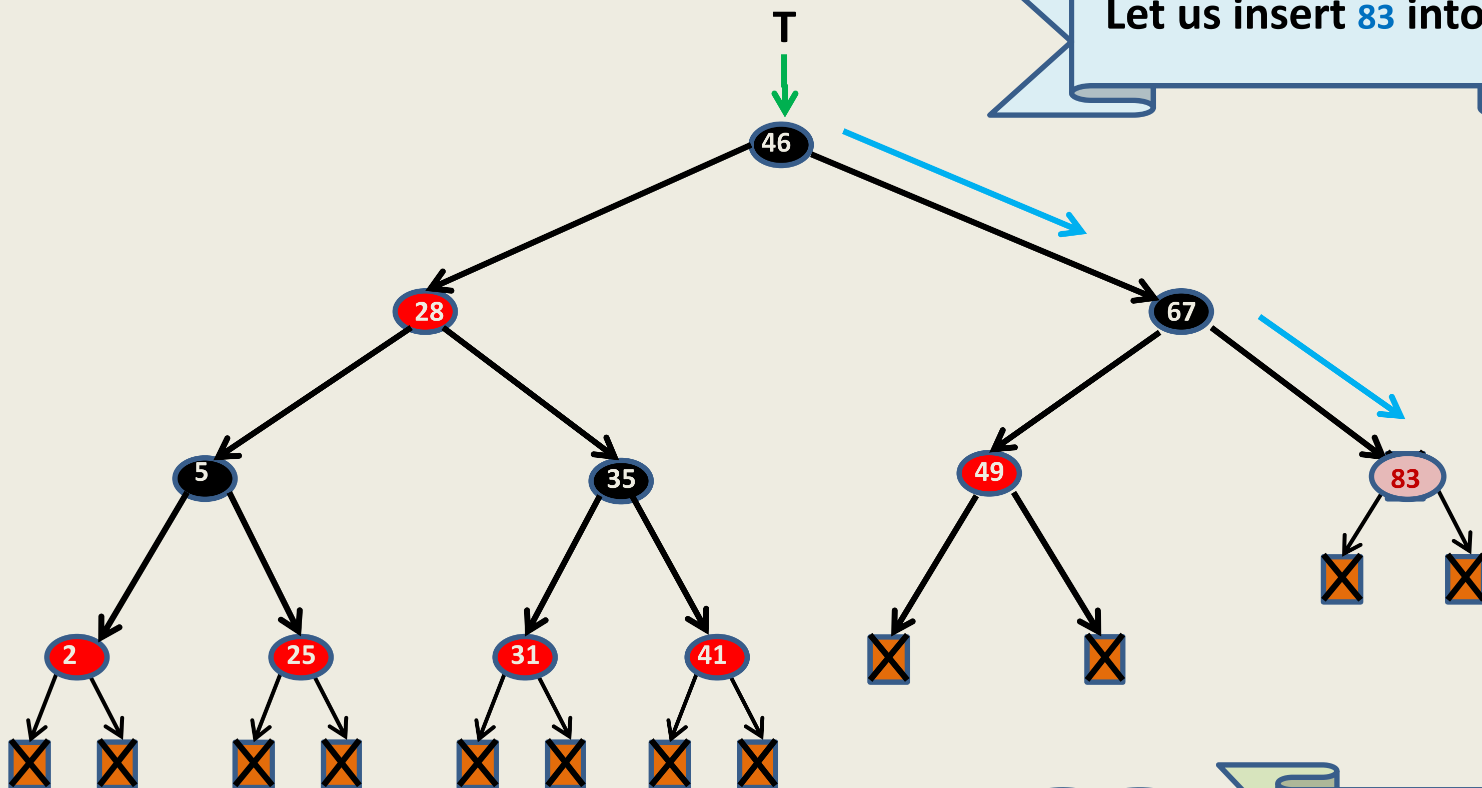
Homework: Ponder over the above hint to prove that T has $\geq 2^h - 1$ elements.

Insertion in a Red Black tree

All it involves is

- playing with **colors** ☺
- and **rotations** ☺

Insertion in a red-black tree

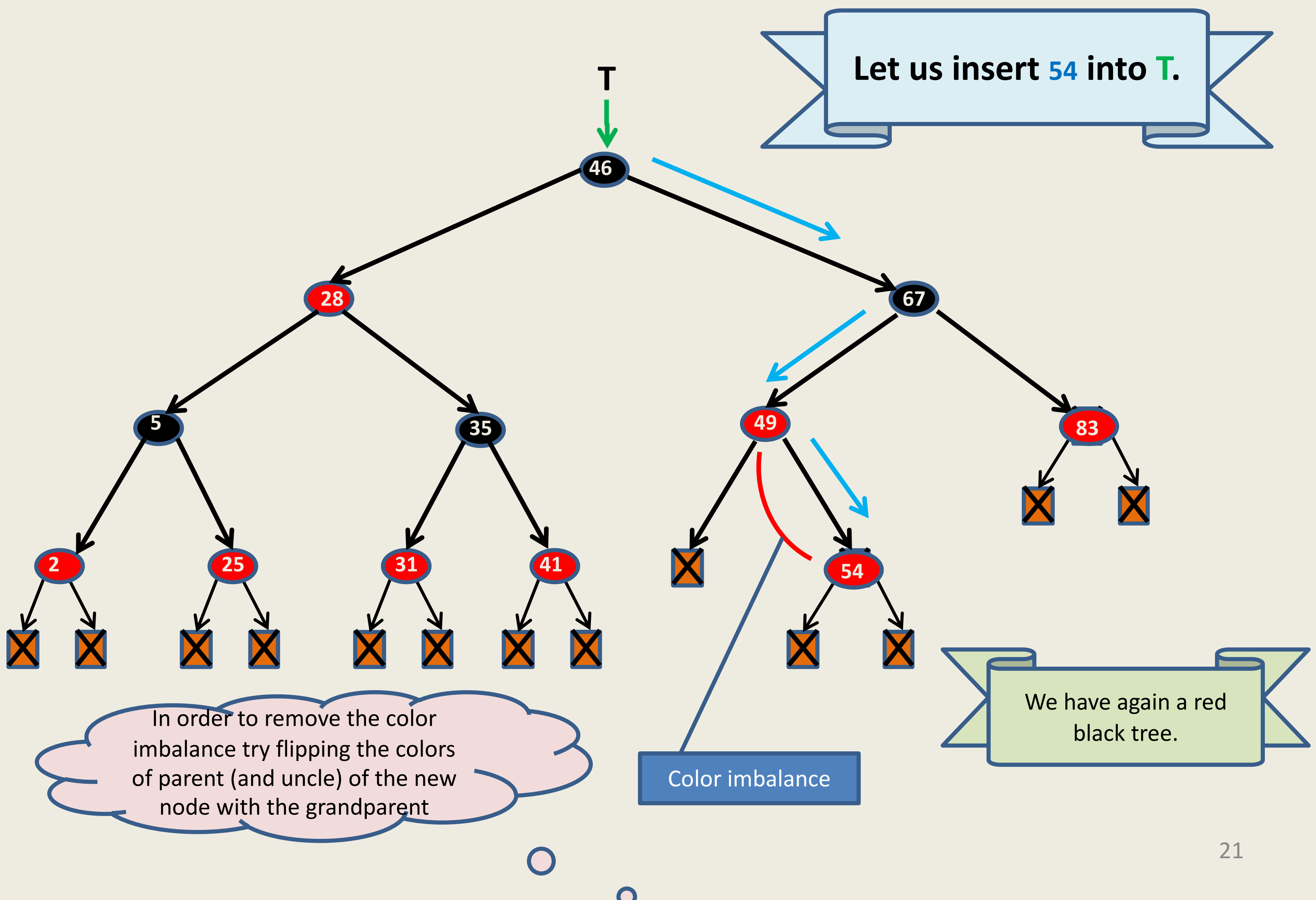


Let us insert 83 into T.

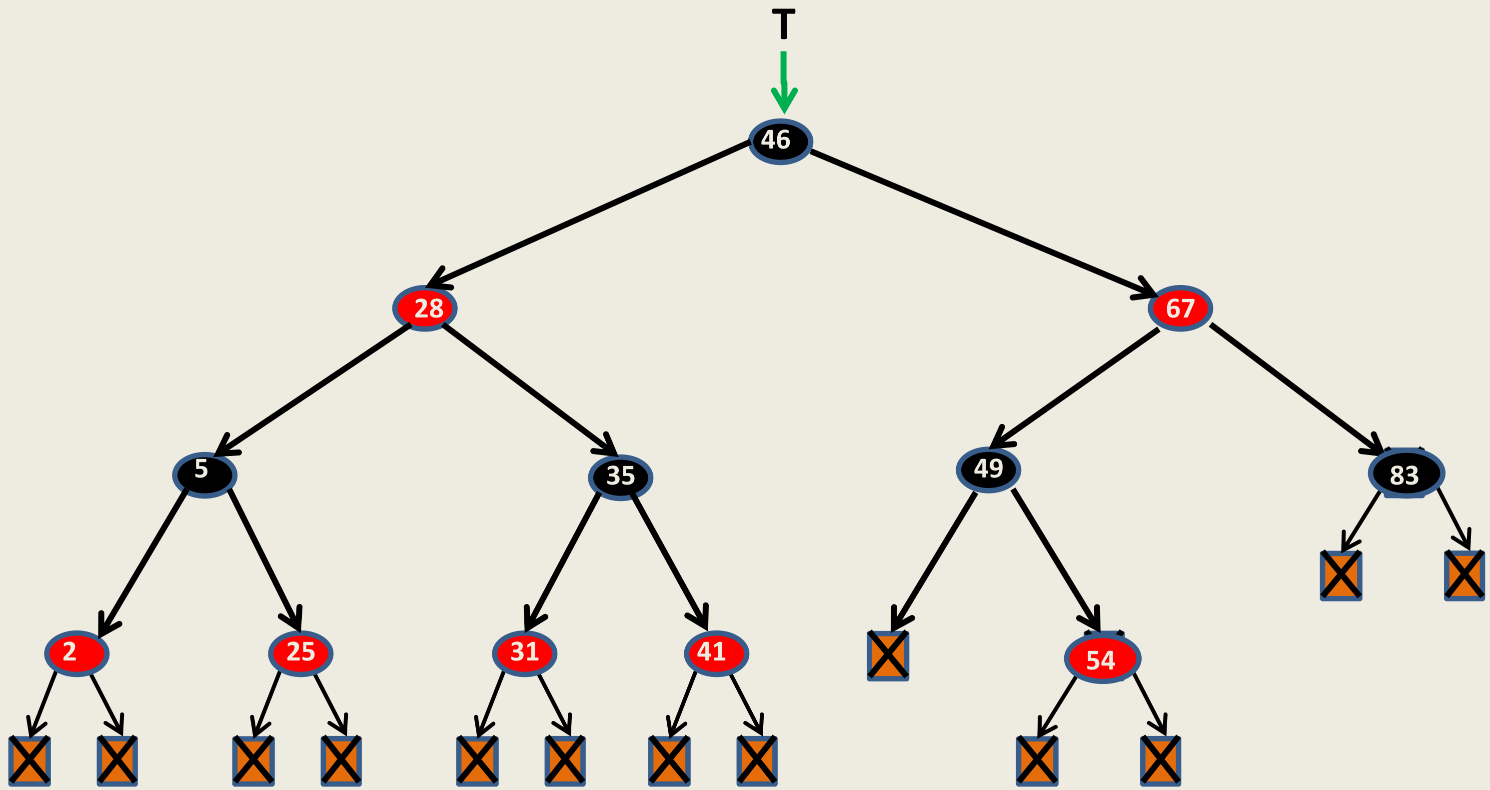
What color should we assign to the new node ?

We shall assign every newly inserted node a red color. (give reason)

Insertion in a red-black tree

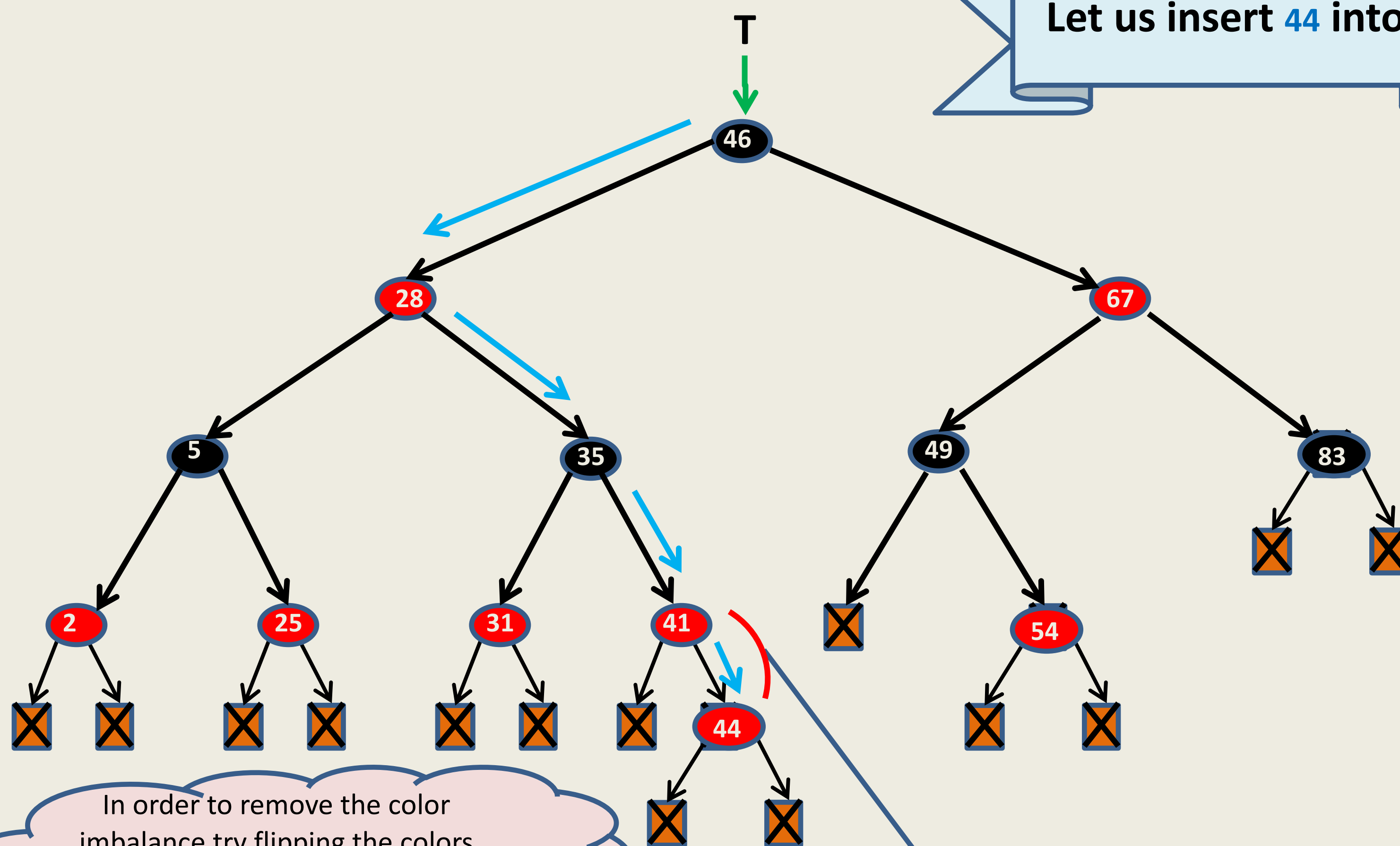


Insertion in a red-black tree



Insertion in a red-black tree

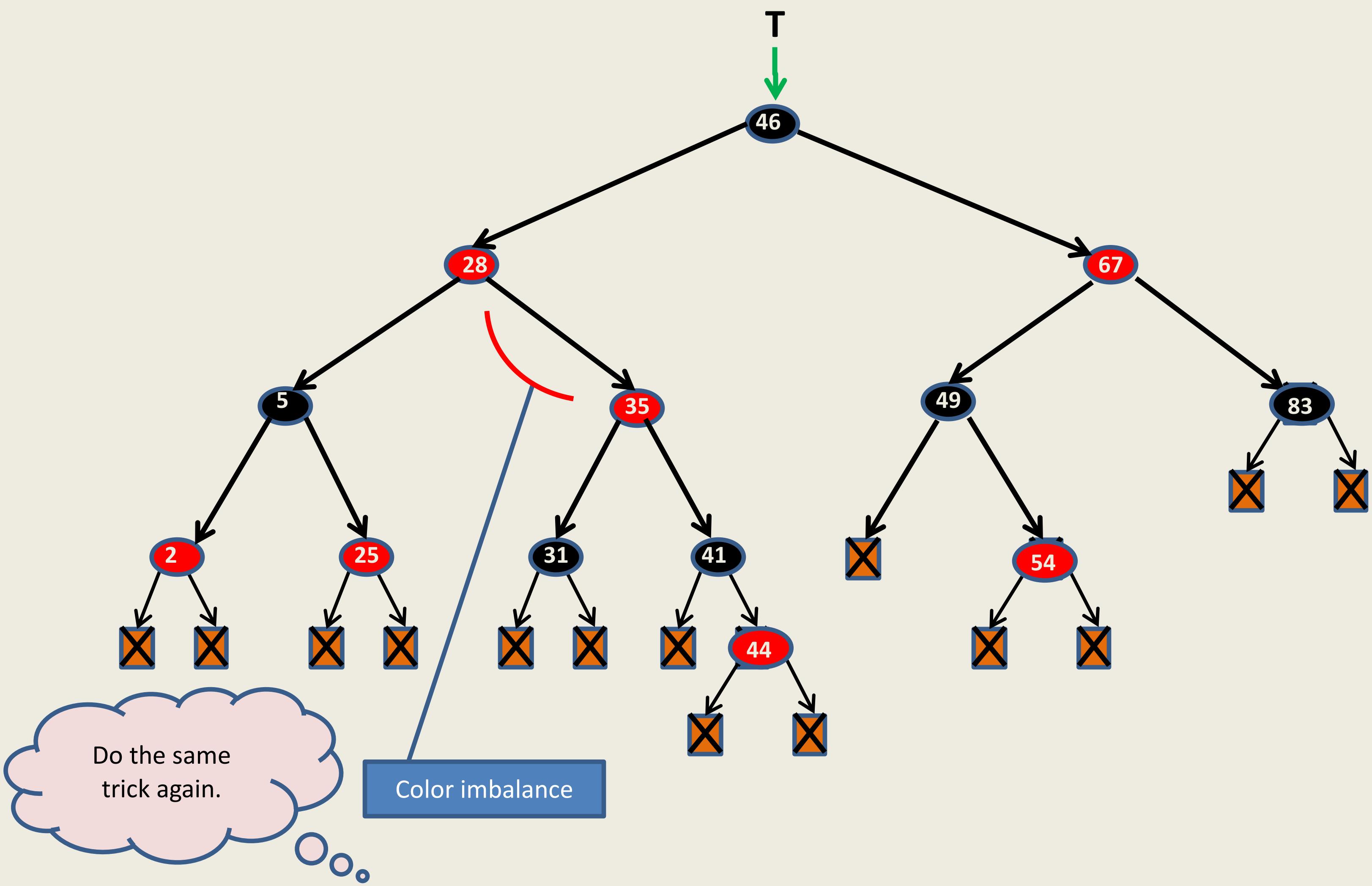
Let us insert 44 into T.



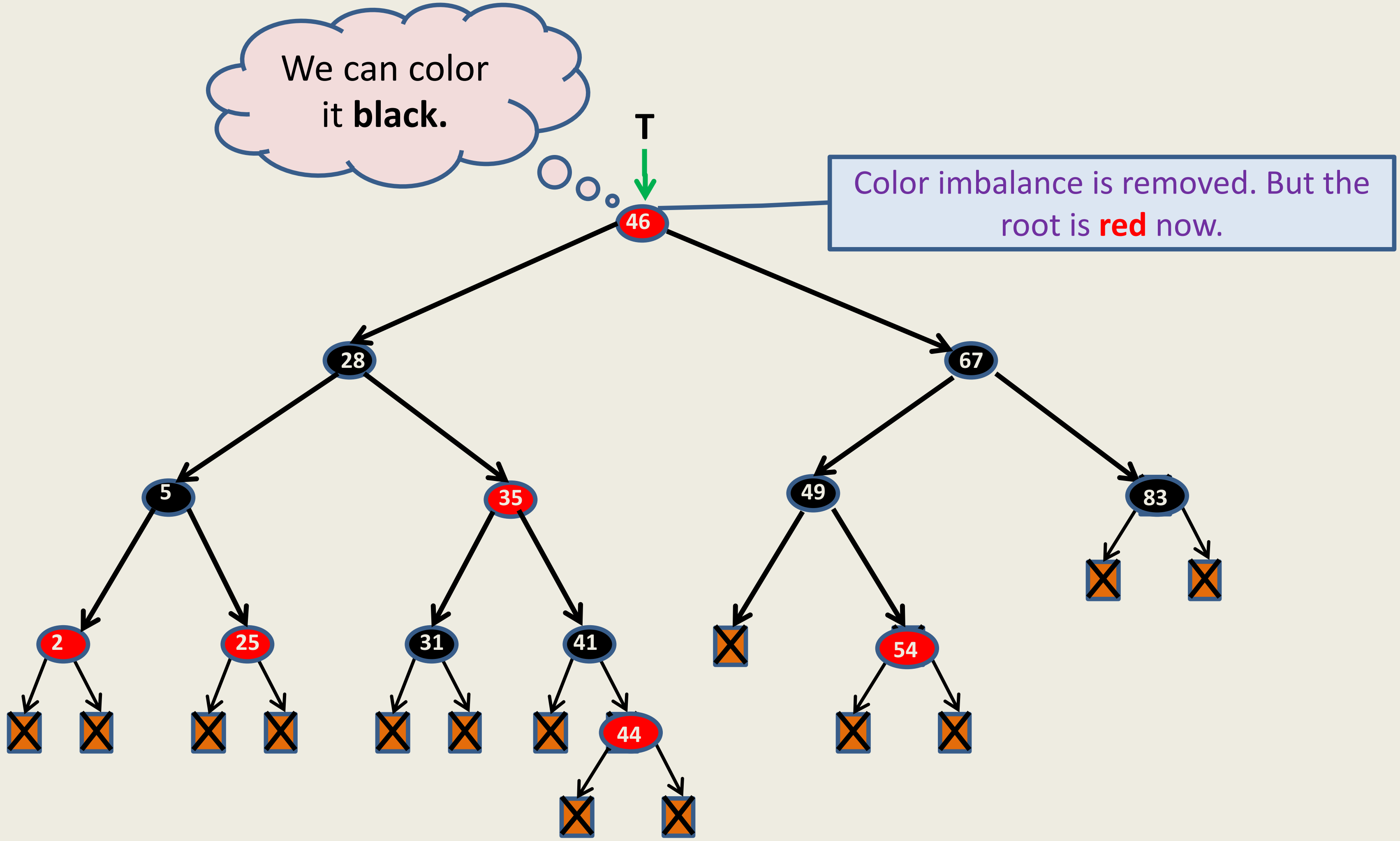
Color imbalance

In order to remove the color imbalance try flipping the colors of parent (and uncle) of the new node with the grandparent

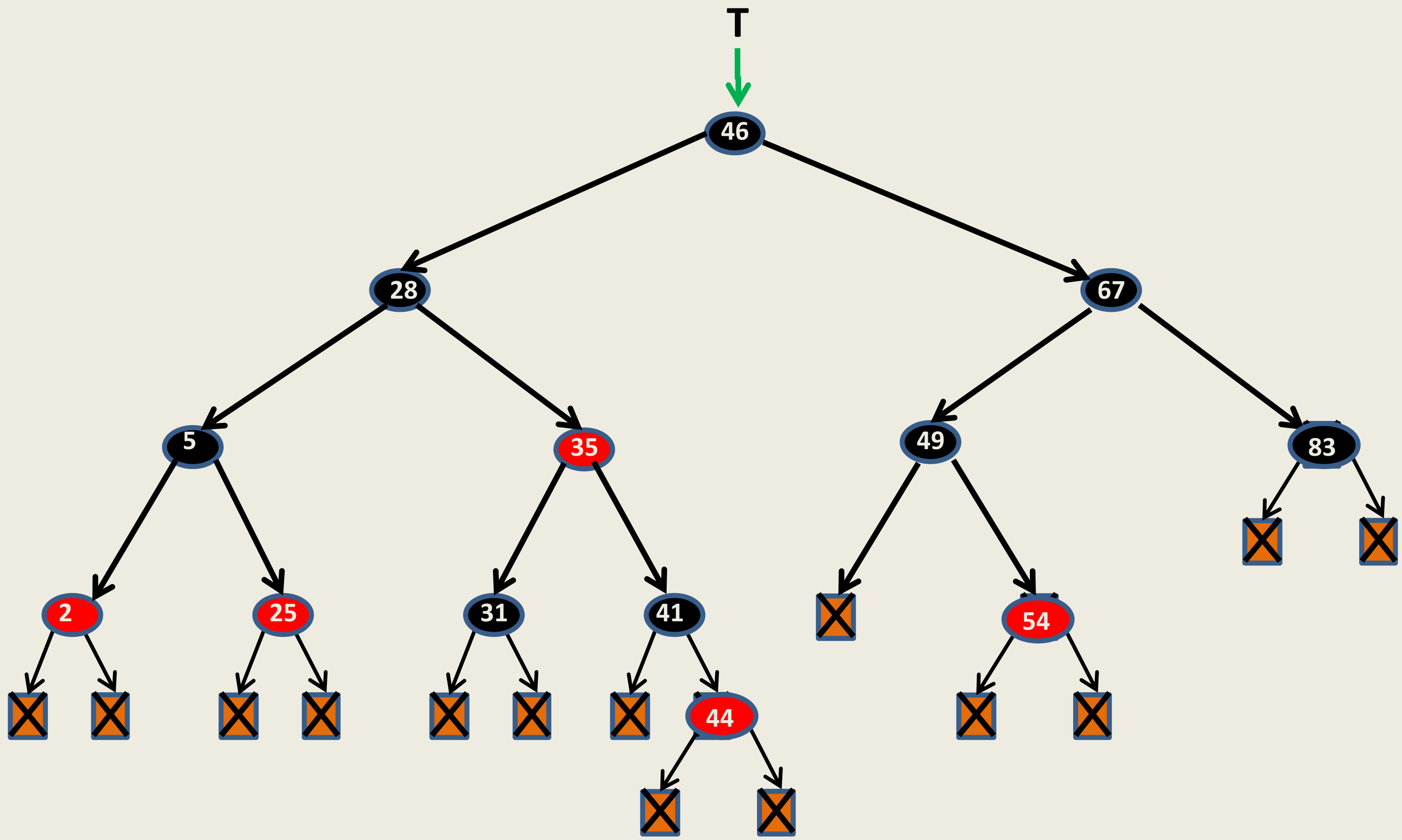
Insertion in a red-black tree



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: ***parent***(p) is **black**

nothing needs to be done.

Case 2: ***parent***(p) is **red** and ***uncle***(p) is **red**,

Swap colors of ***parent*** (and ***uncle***) with ***grandparent***(p).

This balances the color at p but may lead to <u>imbalance</u> of color at grandparent of p. So $p \leftarrow$ ***grandparent***(p), and proceed upwards similarly.

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

Case 3: ***parent***(p) is **red** and ***uncle***(p) is **black**.

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

Handling case 3

Description of Case 3

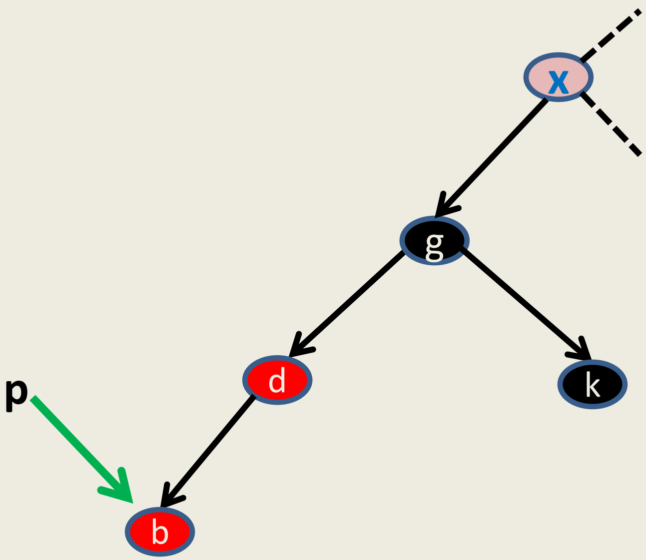
- *p* is a **red** colored node.
- ***parent(p)*** is also **red**.
- ***uncle(p)*** is **black.**

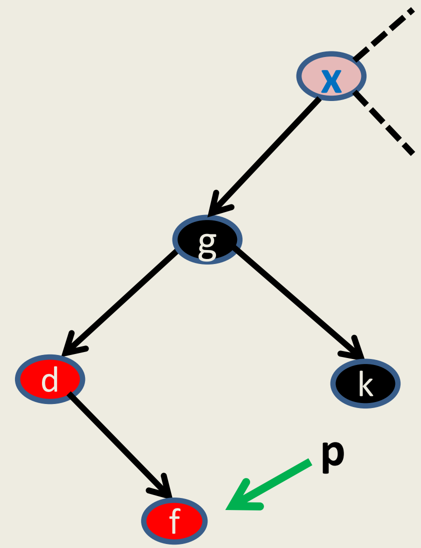
Without loss of generality assume: ***parent(p)*** is ***left child*** of ***grandparent(p).***

(The case when ***parent(p)*** is ***right child*** of ***grandparent(p)*** is handled similarly.)

Handling the case 3

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





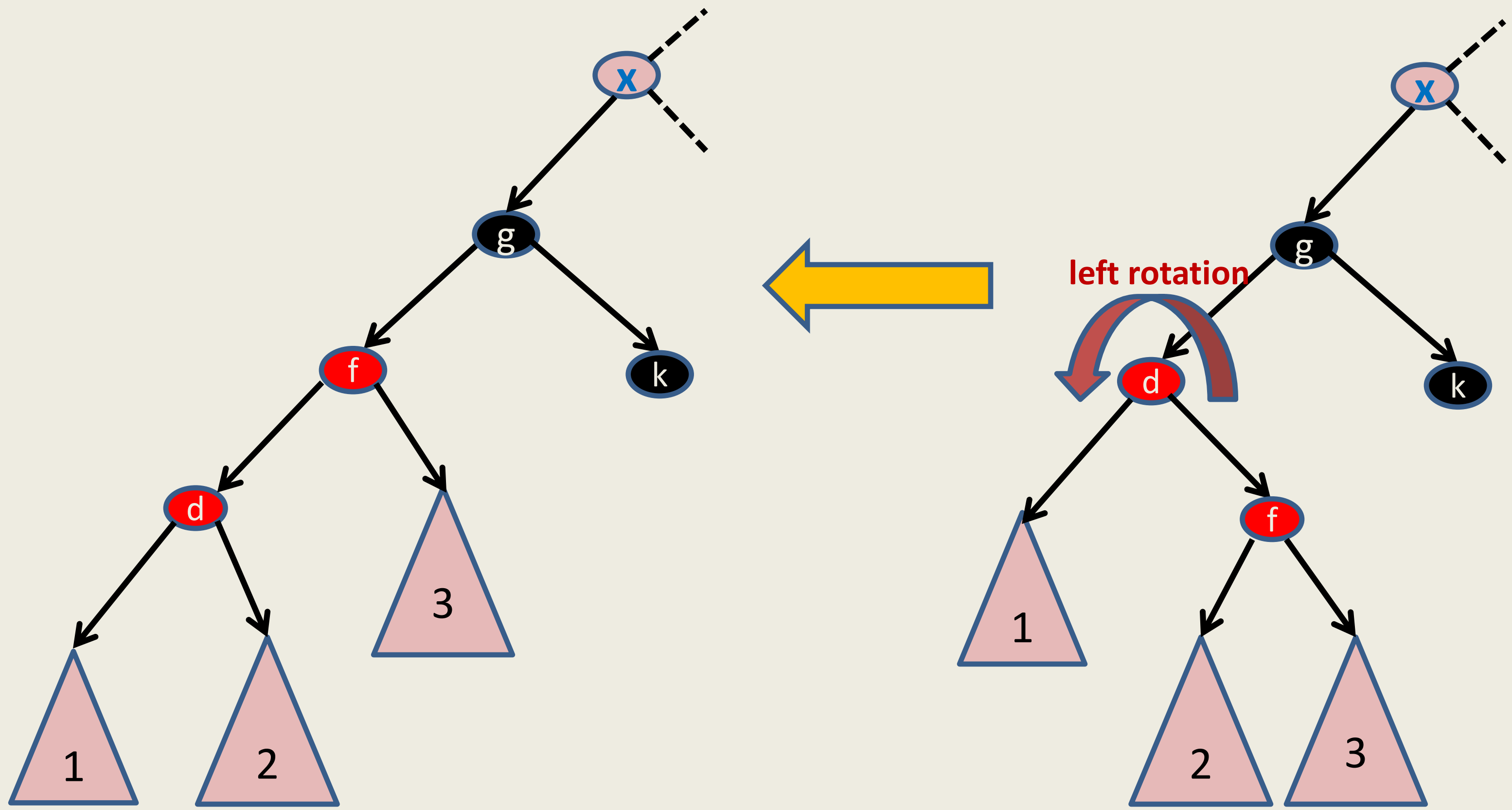
Can you transform **Case 3.2** to **Case 3.1** ?

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

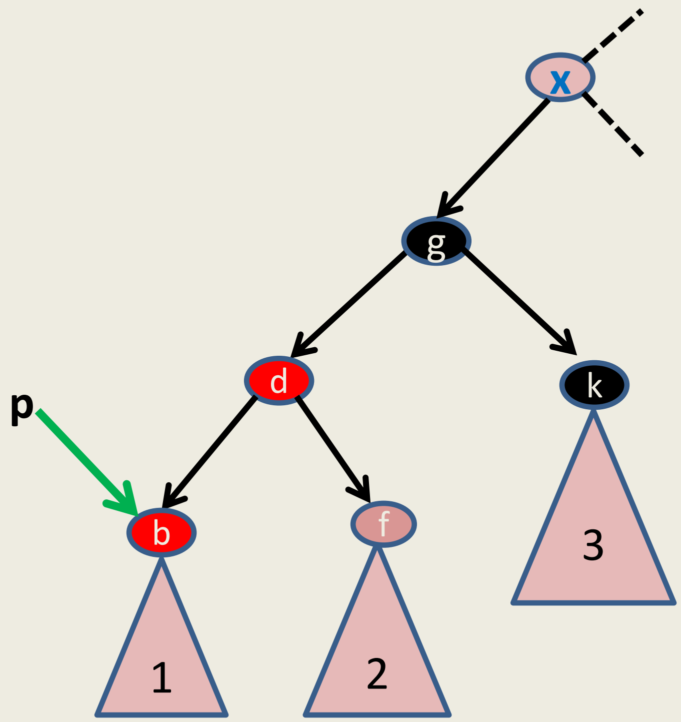


Vow!
This is exactly **Case 3.1**

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

We need to handle only case 3.1

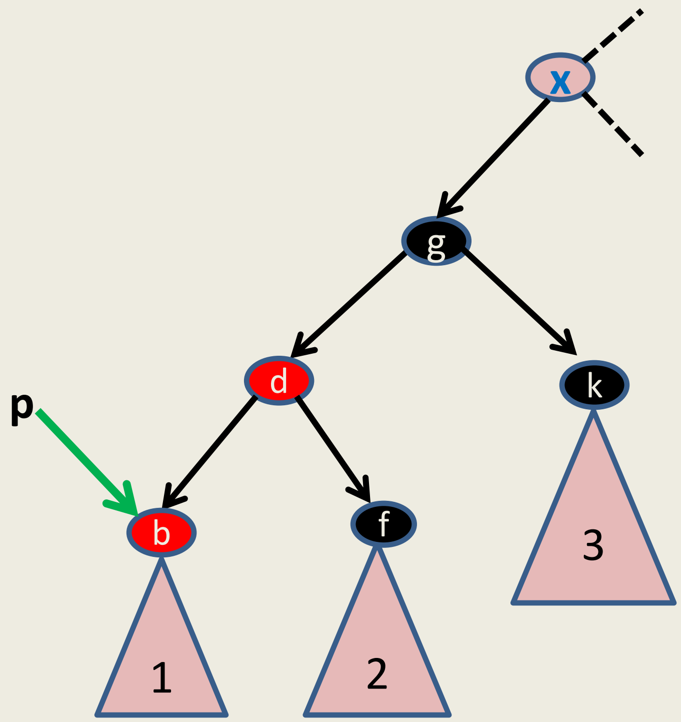
Handling the case 3.1



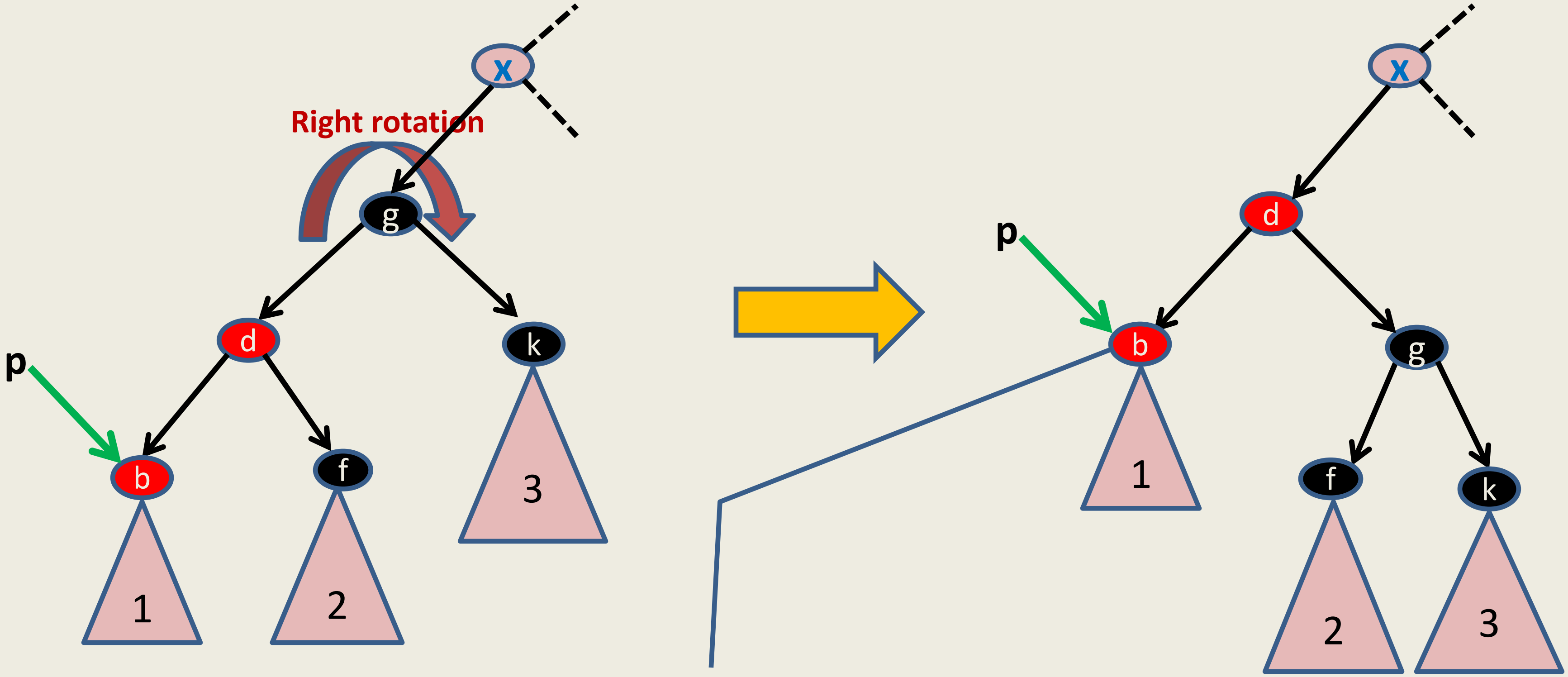
Can we say anything about color of node f ?

black

Handling the case 3.1



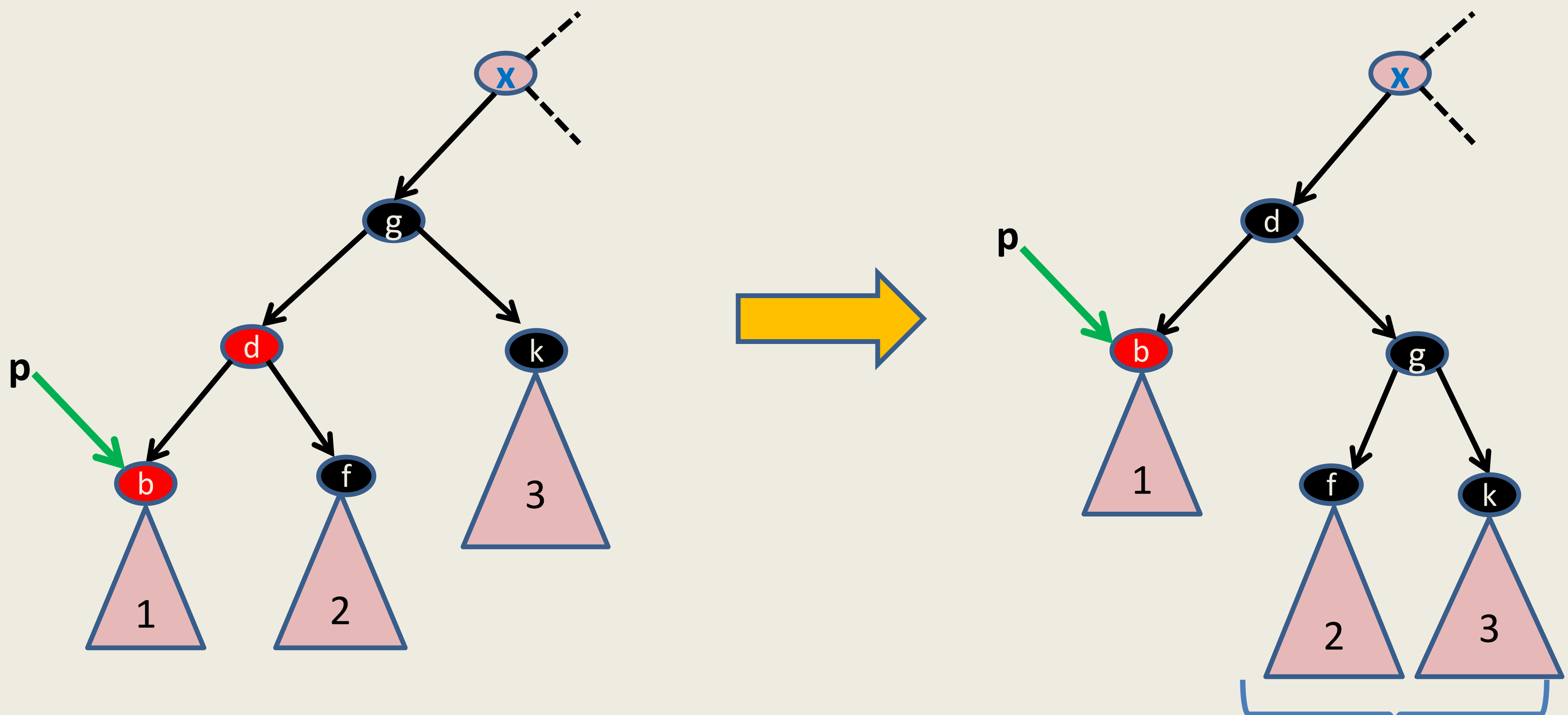
Handling the case 3.1



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**

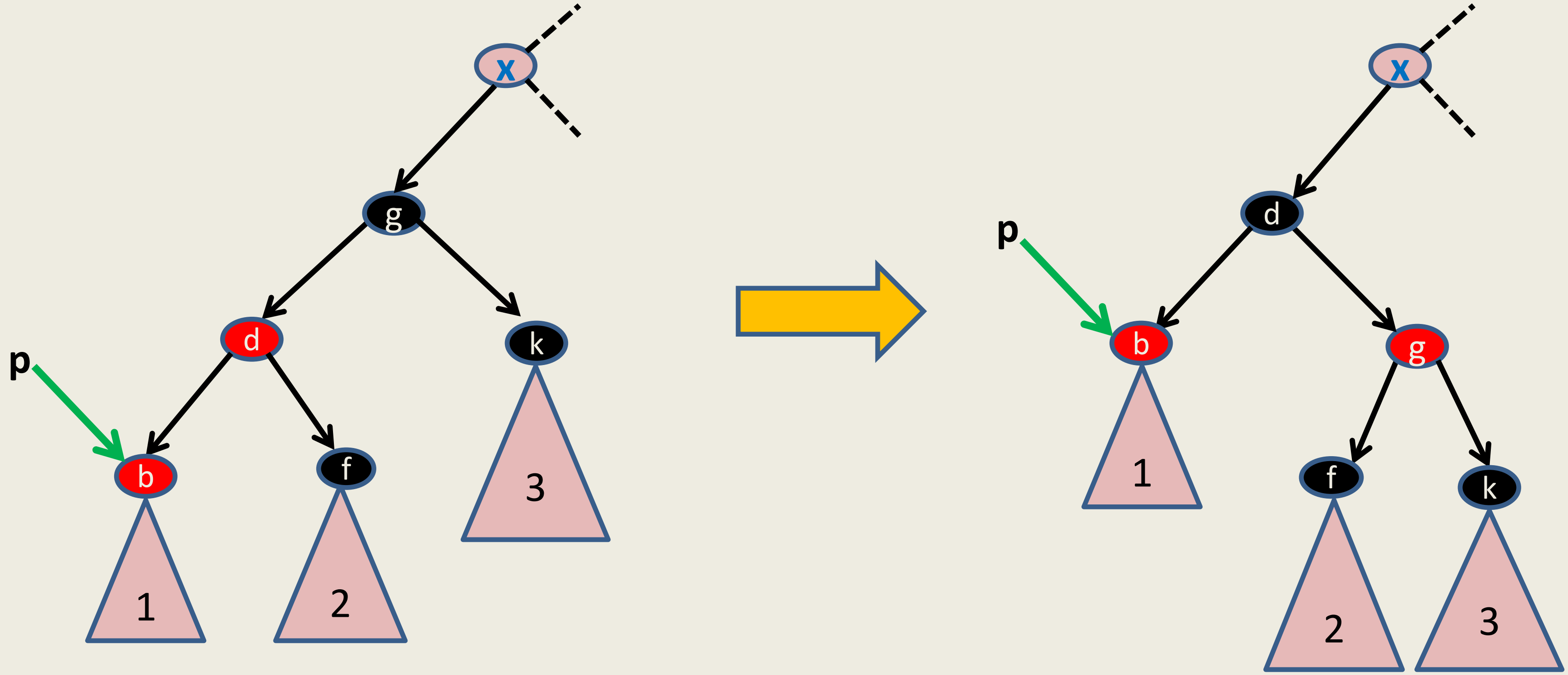
Handling the case 3.1



The number of **black** nodes on the path … restored for tree 1. Color imbalance is …
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.

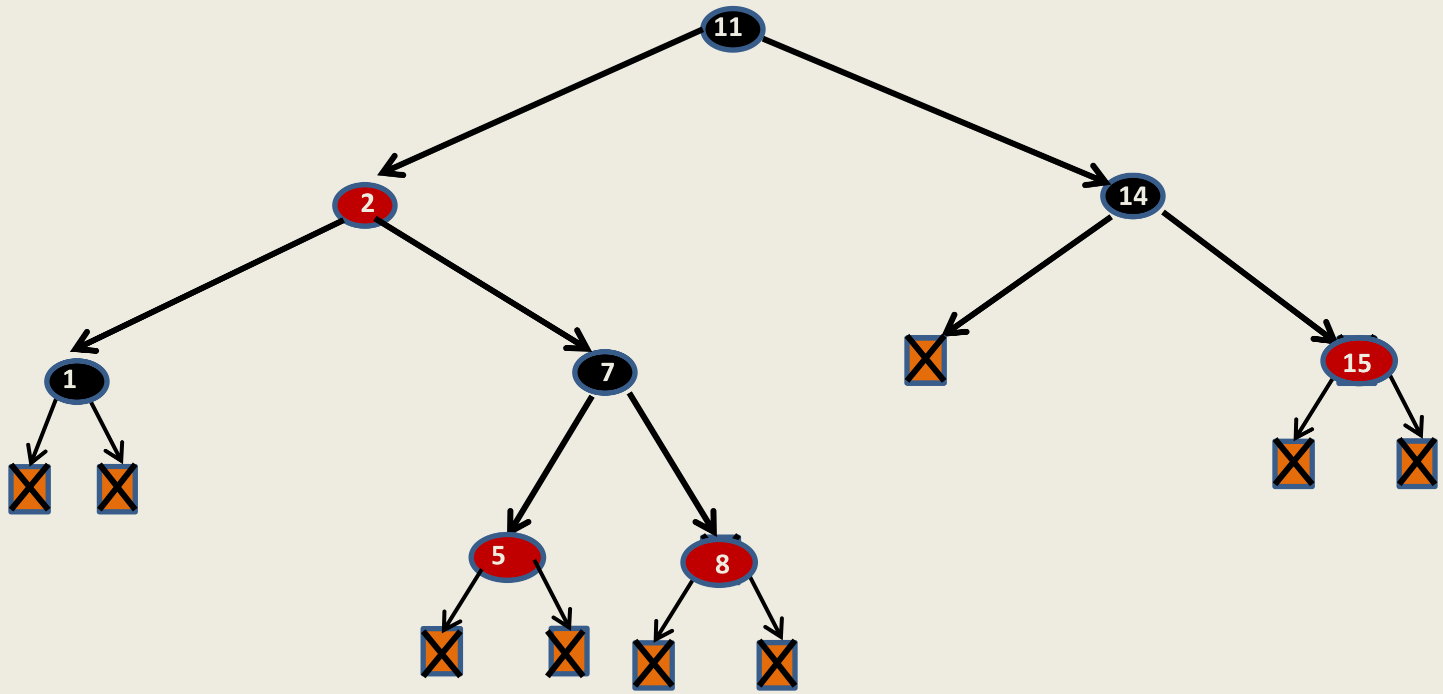
The following are the natural questions to ask.

- Why we are handling insertions in “this *particular* way” ?
- Are there *alternative and simpler* ways to handle insertions ?

You are encouraged to explore the answer to both these questions.
You are welcome to discuss them with me.

- Please solve the problem on the following slide.

How to insert 4 ?



How do will we handle deletion ?

This is going to be a bit more complex.

So please try on your own first before next lecture.

It will still involve playing with colors and rotations ☺

Do not miss the next lecture ☺