Data Structures and Algorithms

(CS210A)

Semester I – **2014-15**

Lecture 18:

Analysis of

- **Red Black trees**
- **Nearly Balanced BST**

A **Red** **Black** Tree is height balanced

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

Red **Black** Tree

Red Black tree:

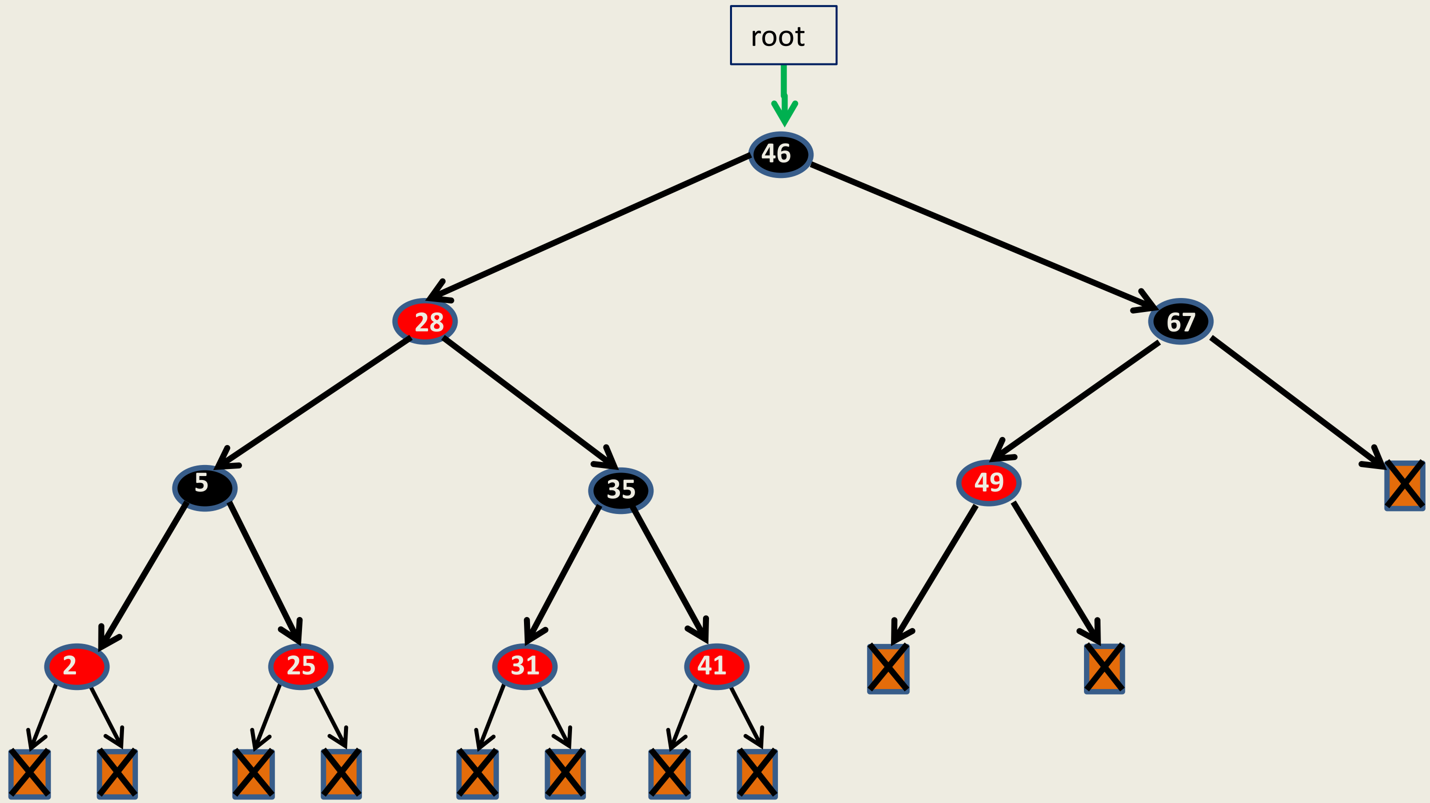
a **full** binary search tree with each leaf as a **null** node

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

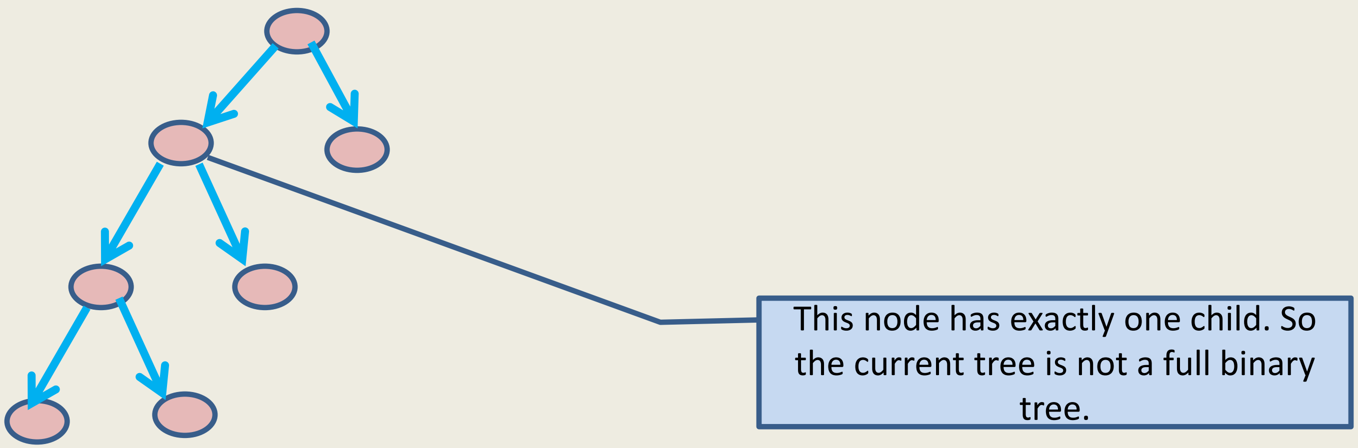
A red-black tree



Terminologies

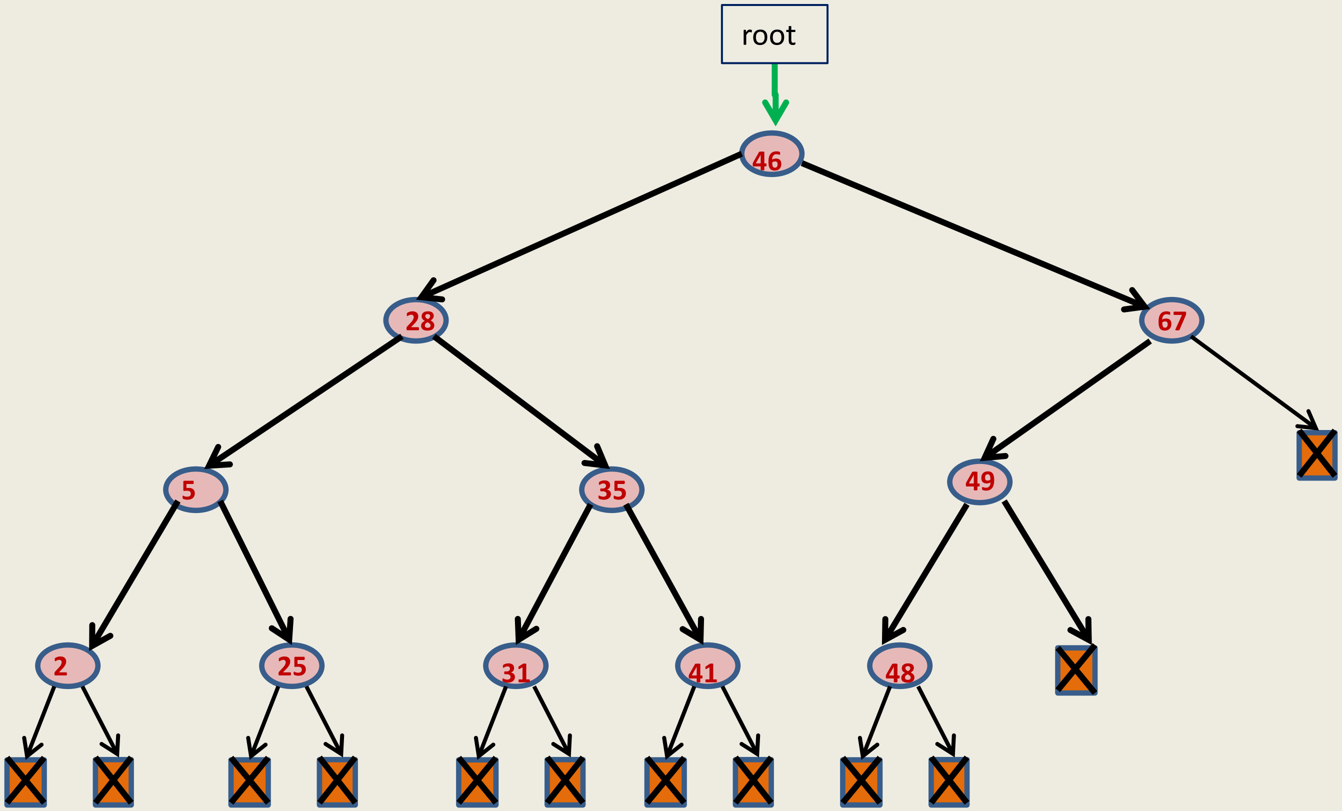
Full binary tree:

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



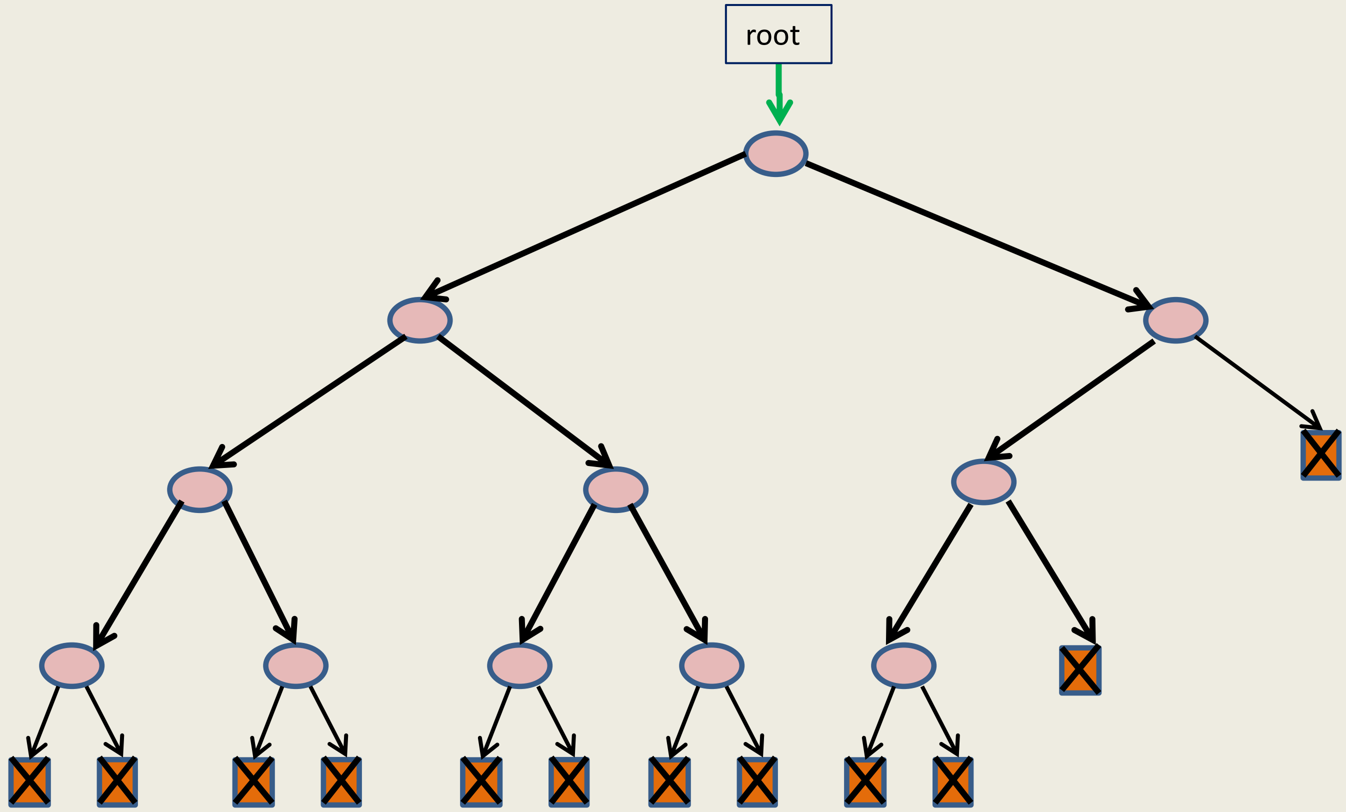
This is now a full binary tree.

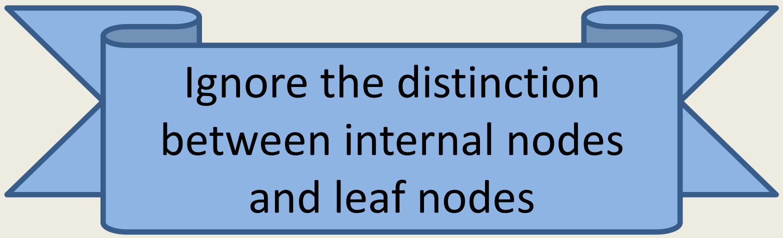
Red-black tree: as a Full Binary Tree



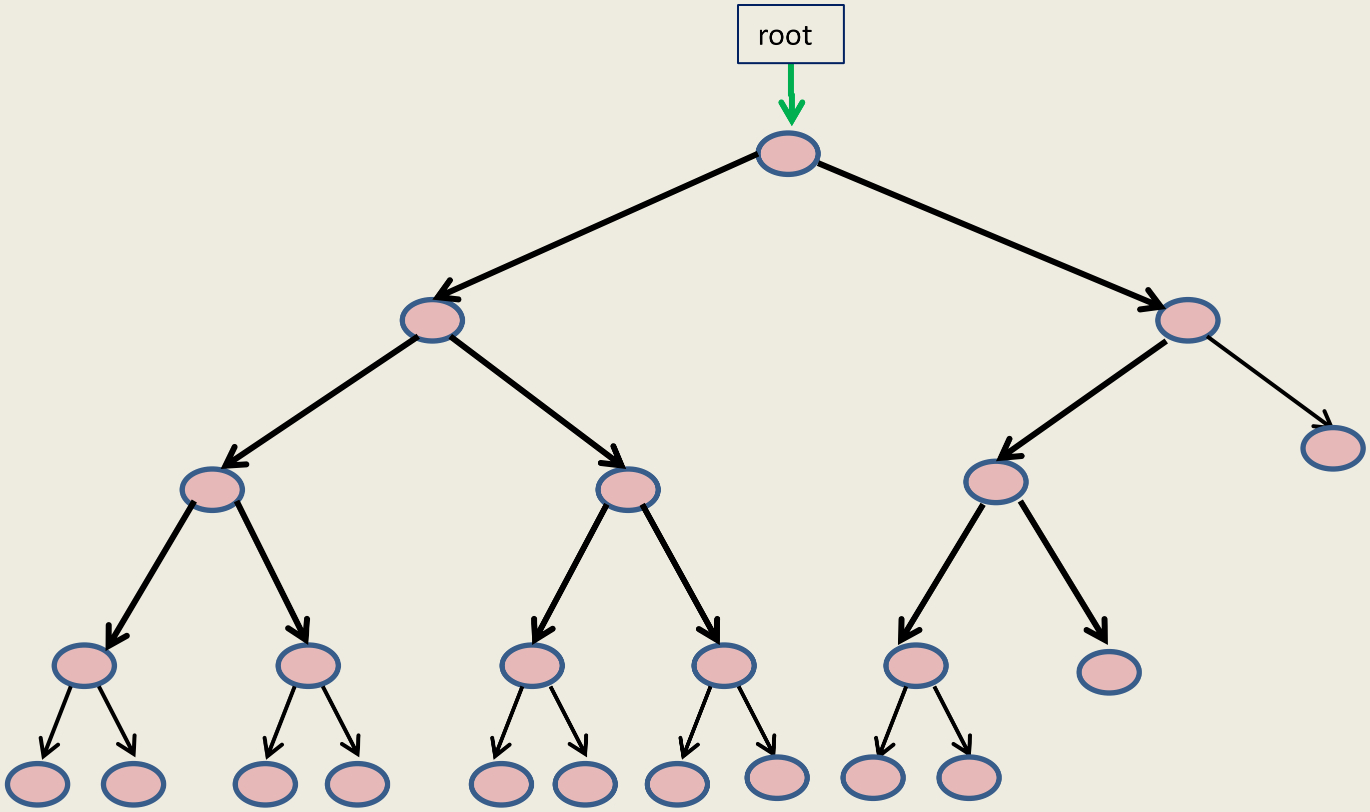
Ignore the values

Red-black tree: as a Full Binary Tree





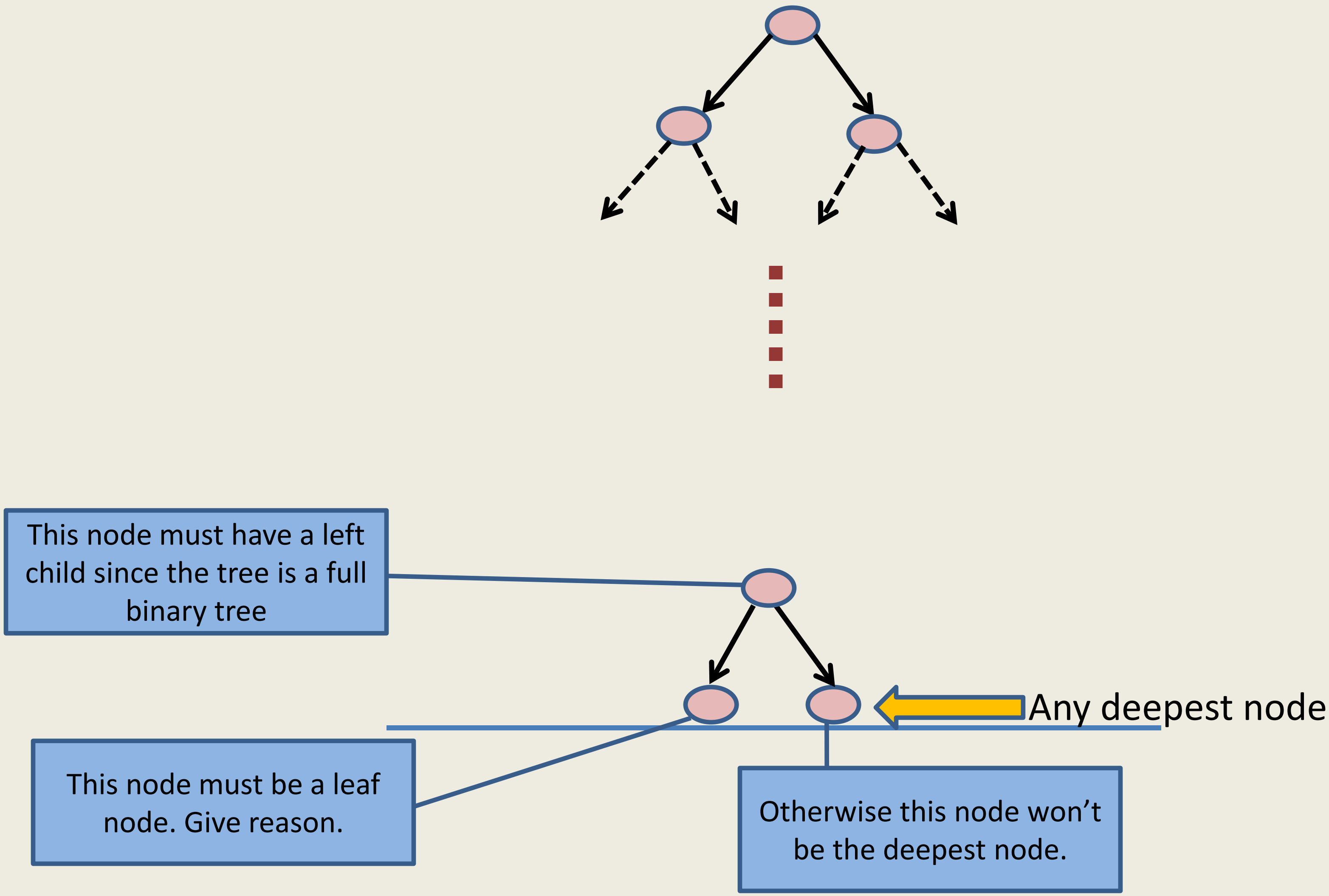
Red-black tree: as a Full Binary Tree



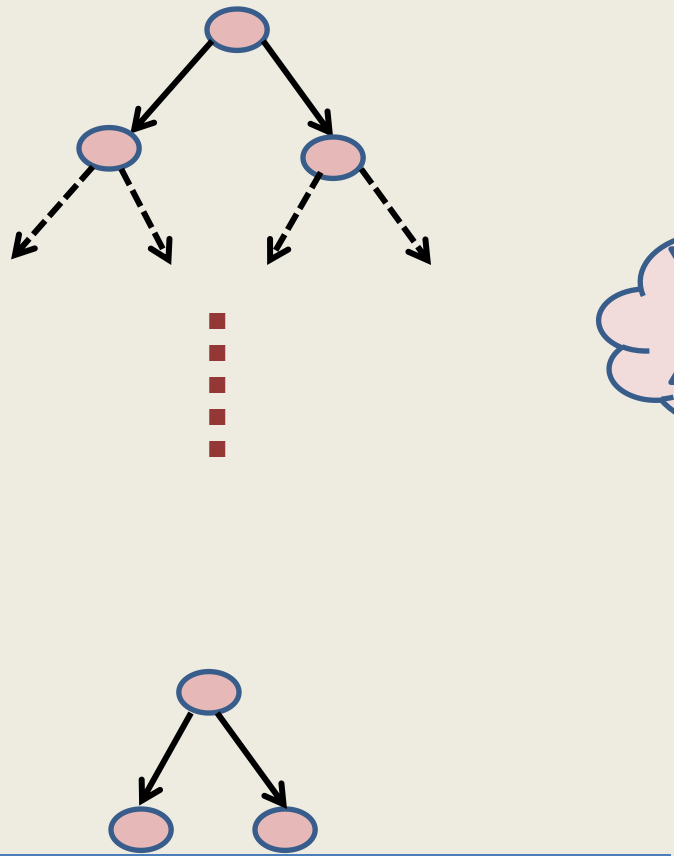
Properties of
a Red-Black Tree viewed as a full binary tree

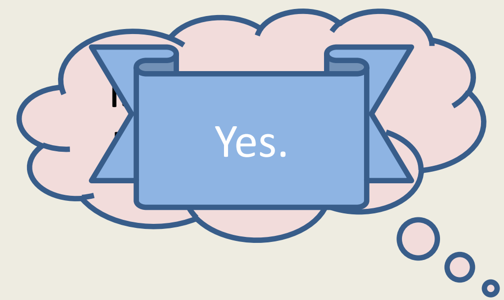
Relationship between
Number of leaf nodes and Number of internal nodes

A full binary tree



A full binary tree





What happened to the number of leaf nodes ? **Reduced by one**

What happened to the number of internal nodes ? **Reduced by one**

A full binary tree

Analyze the process:

Repeat

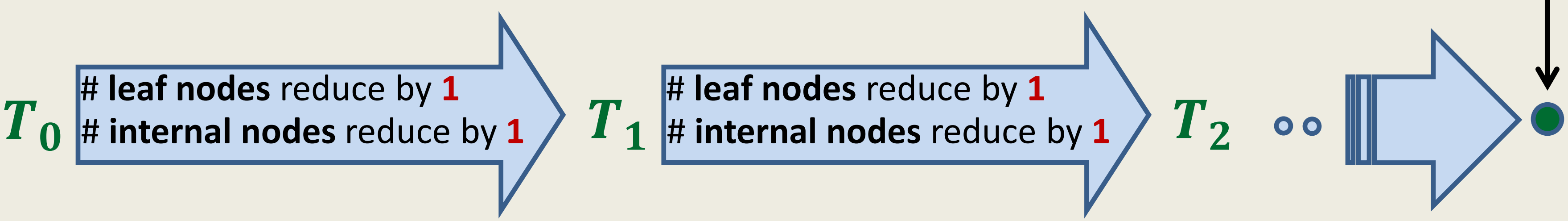
{ Delete the **deepest node** and its **sibling** }

until only **root** remains

Finally only root node remains

Let T_0 be the full binary tree before the process starts.

Let $T_1, T_2, \ldots$ be the full binary trees after 1st, 2nd, … iterations of the process.



Question: What might be the relation between leaf nodes and internal nodes in T_0 ?

Answer: No. of **leaf nodes** in T_0 = No. of **internal nodes** in T_0 + 1.

A full binary tree

Question: If $\boldsymbol{i}$ is the number of internal nodes in a full binary tree ***T***, what is the size (number of nodes) of the tree ?

Answer: $\boldsymbol{2i + 1}$

Question: What is the size of a **Red** **Black** tree storing $\boldsymbol{n}$ keys ?

Answer: $\boldsymbol{2n + 1}$

A complete binary tree of height h and its Properties

A **complete** binary tree of height h

Definition:

A full binary tree of height h is said to be a **complete** binary tree of height h if every leaf node is at depth h.

> There is no reason to believe **at this stage** that there is a unique complete binary tree of height h.

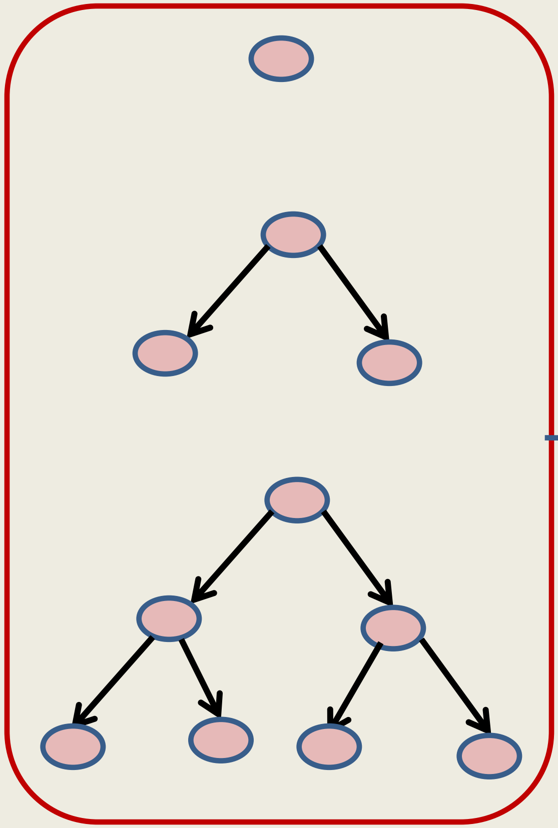
Question: How will any complete binary tree of height h look like ?

A **complete** binary tree of height h

Complete binary tree of height **1** **?**

Complete binary tree of height **2** **?**

Complete binary tree of height **3** **?**

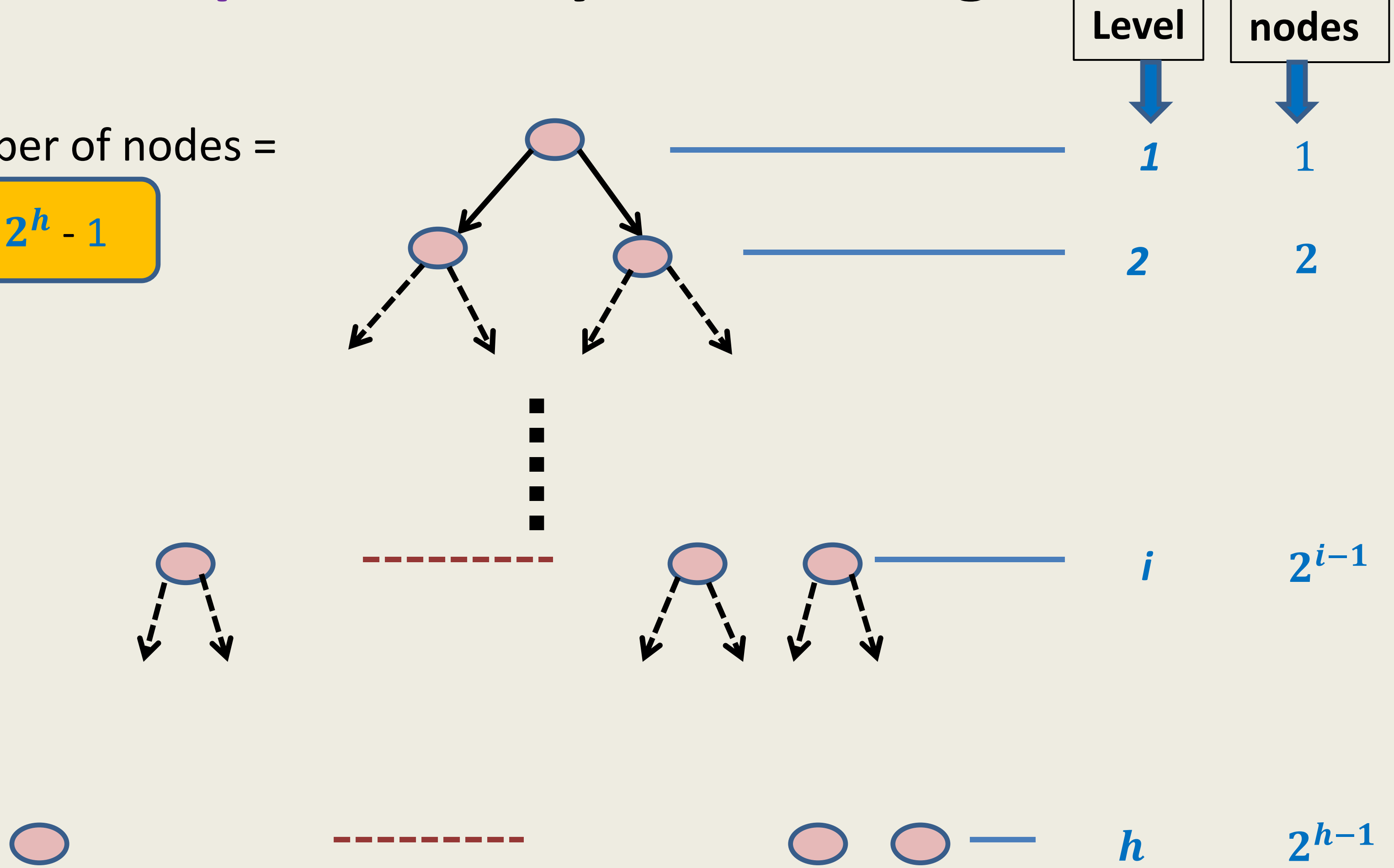


Try to generalize the type of tree shown here to tree of height h.

A **complete** binary tree of height h

Total number of nodes =

$2^h - 1$



Certainly this tree is a complete binary tree of height h
We shall now show that this is **the only possible** complete binary tree of height h.

Uniqueness
of a complete binary tree of height h

Let T^* be the complete binary tree of height h shown in previous slide.

Notice that this is **densest** possible tree of height h.

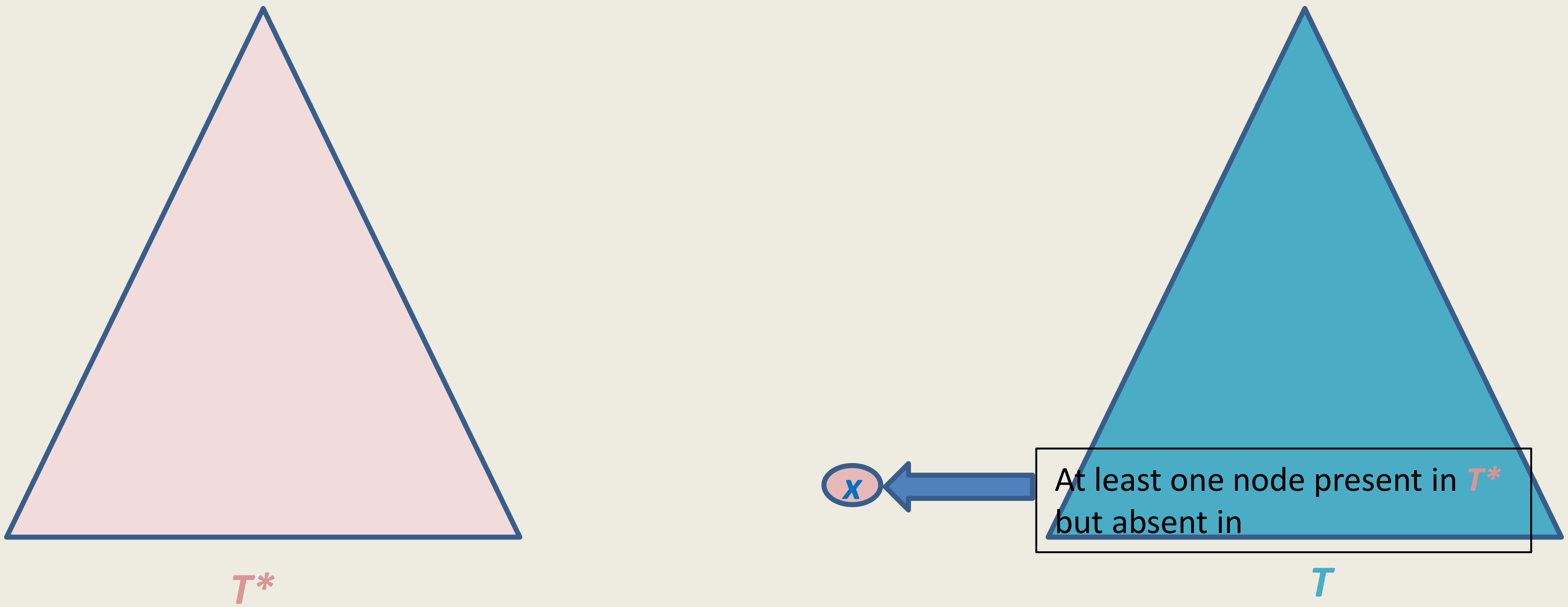
Let T be any other complete binary tree of height h **different** from T^*.

Question: How to show that T can not exist ?

Think over this question carefully.

Watch the following slide carefully.

Uniqueness
of a complete binary tree of height h



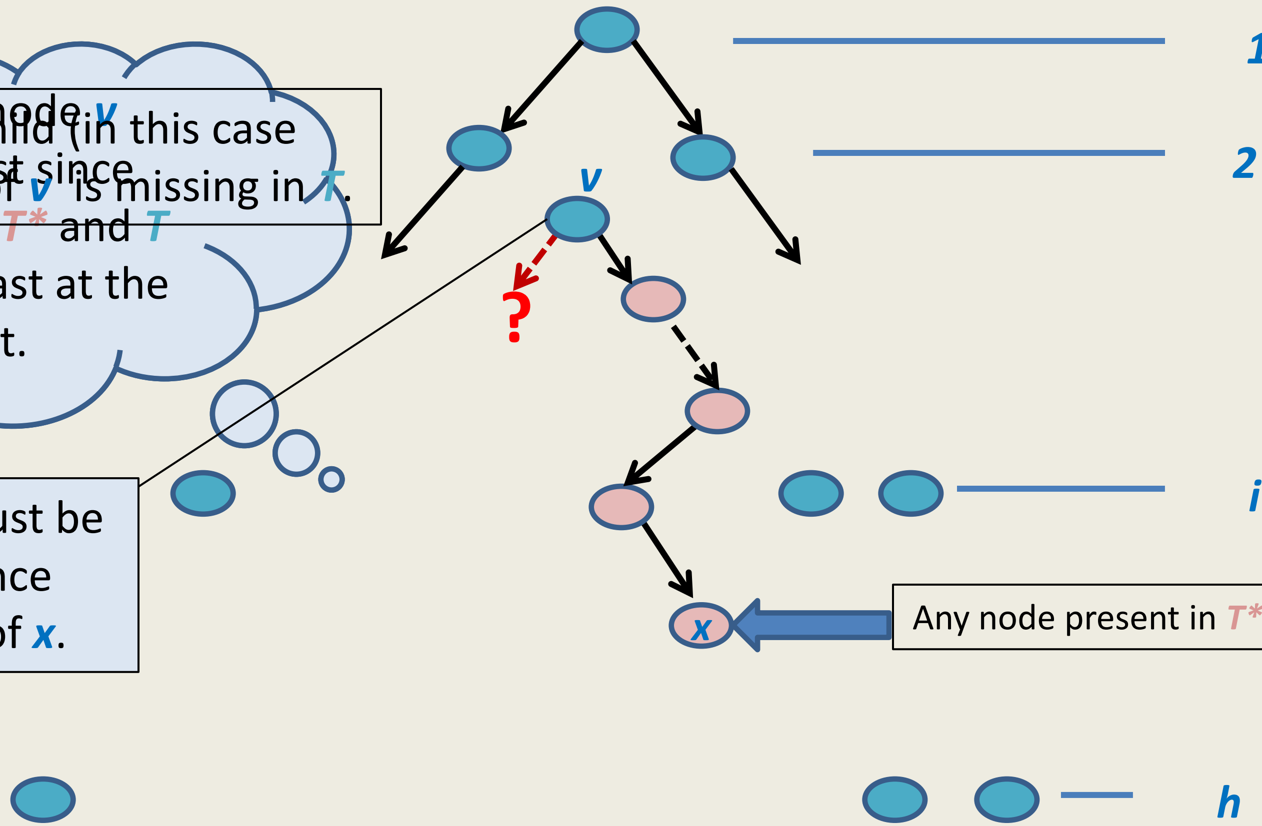
Just *place* *T* symmetrically above *T**.
If *T* is *different* from *T**, *then what will you see ?*

Uniqueness
of a complete binary tree of height h

Note that one child (in this case the right child) of v is missing in T.

Such a node v must exist since both both T^* and T *meet* at least at the root.

Depth of v must be less than h since v is ancestor of x.



Any node present in T^* but absent in T

Since T is a full binary tree and **right child** of v is missing,

➔ v can not be an internal node in T.

➔ v **must be** leaf node.

Hence v is a leaf node of T at depth < h
Hence
T is not a compete binary tree of height h

Hence there is no **complete** binary tree of height h different from *T**.

➔ **There exists a unique** a **complete** binary tree of height h.

Theorem:

A complete binary tree of height h has exactly 2^h - 1 nodes.

A **Red** **Black** Tree is height balanced

The final proof

T : a red black tree storing n keys.

Total number of nodes = $2n + 1$

h : the black **height**

Every leaf node is at depth ≥ h

Question: How does T look like if we remove all nodes at depth > h ?

Answer: a complete binary tree of height h

Hence $2n + 1 \geq 2^h - 1$

➔ $2^h \leq 2n + 2$

➔ $h \leq 1 + \log_2(n+1)$

So Height of $T \leq 2h - 1 \leq 2\log_2(n+1) + 1$

Analysis

NEARLY BALANCED BST

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

and

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

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

Theorem: Height of a **nearly balanced** BST on n nodes is **O**($\log_{4/3} n$)

Nearly balanced Binary Search Tree

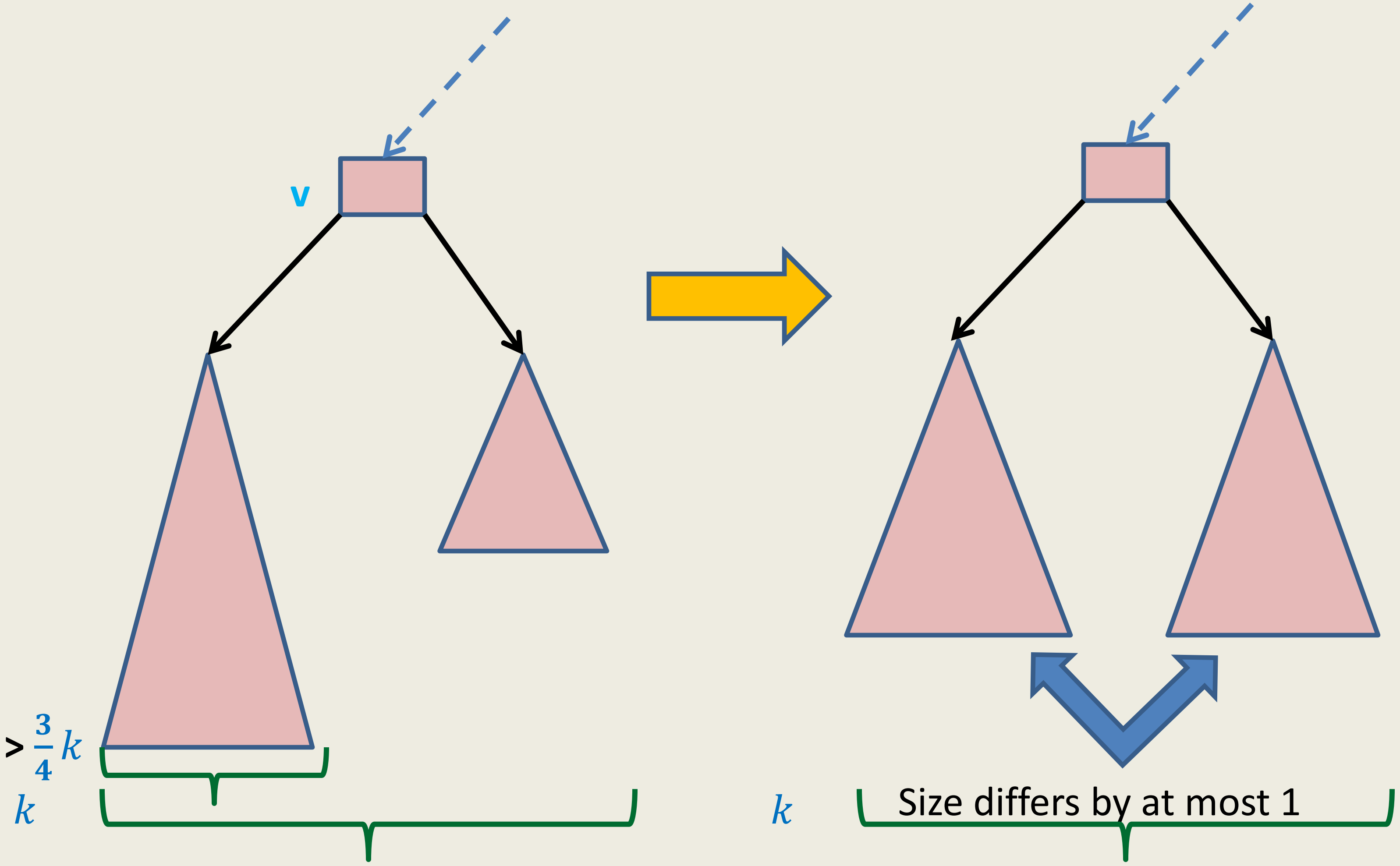
Maintaining under Insertion

Each node **v** in **T** maintains additional field **size**(**v**) which is the number of nodes in the **subtree**(v).

- Keep **Search**(**T**,x) operation unchanged.

- 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**.

"**Perfectly Balancing**" subtree at a node **v**



Nearly balanced Binary Search Tree

Observation :

It takes **O(** k**)** time to transform an imbalanced tree of size k into a perfectly balanced BST.

Observation: Worst case search time in **nearly balanced BST** is **O(log** n**)**

Theorem:

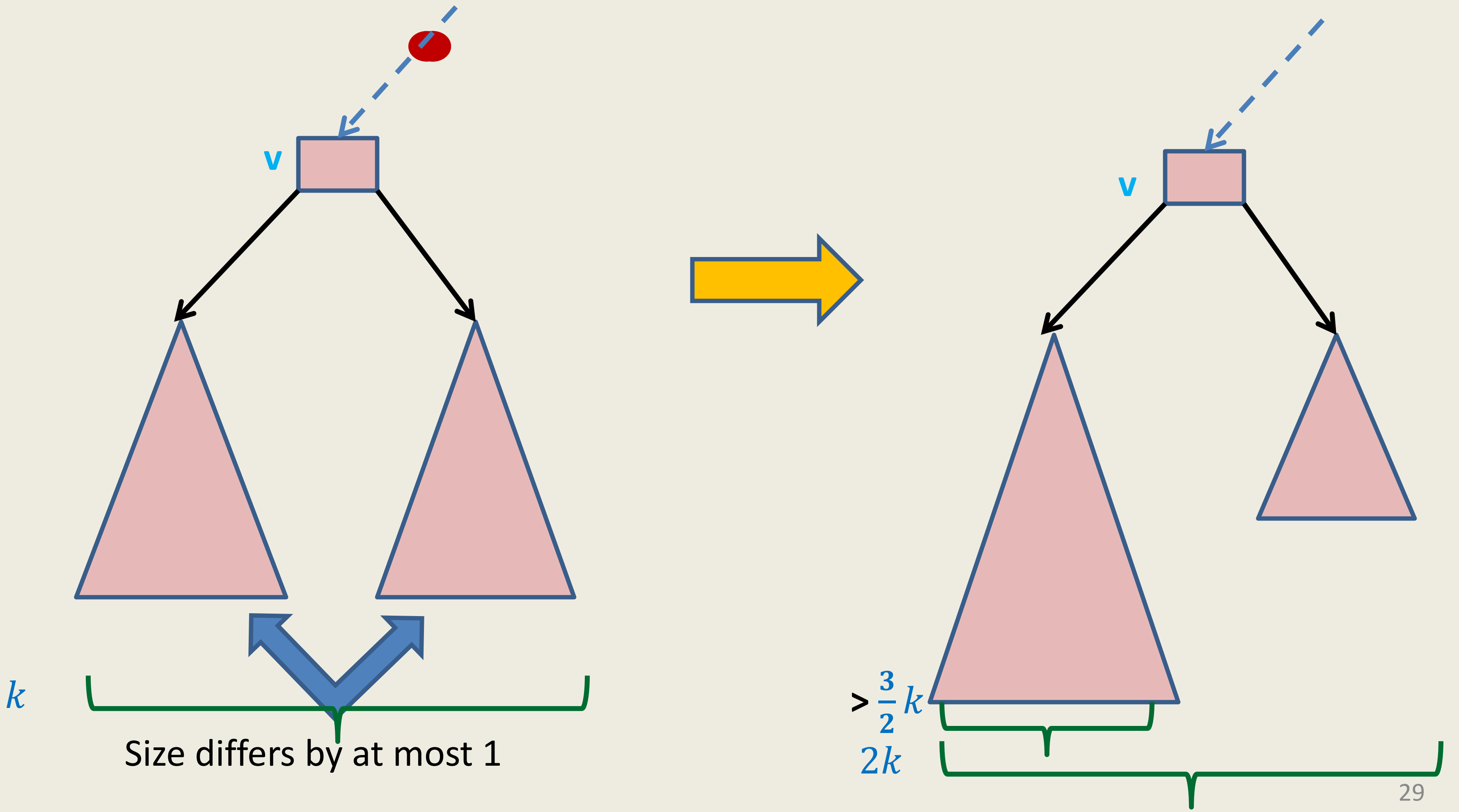
For any arbitrary sequence of n **operations**, total time will be **O(**n **log** n**)**.

We shall now prove this theorem formally.

Watch the **next** slide slowly to get a useful insight.

How many <u>new elements</u> to make **T**(**v**) imbalanced ? $\geq k$

Suppose **T**(**v**) is perfectly balanced at some moment.



The intuition for proving the Theorem

“A perfectly balanced subtree **T**(**v**) will have to have **large number of insertions** before it becomes unbalanced enough to be rebuilt again.”

We shall transform this intuition into a formal proof now.

Notations

$\textbf{size}_k(v)$: no. of nodes in **T**(v) at the end of kth insertion.

For kth insertion,

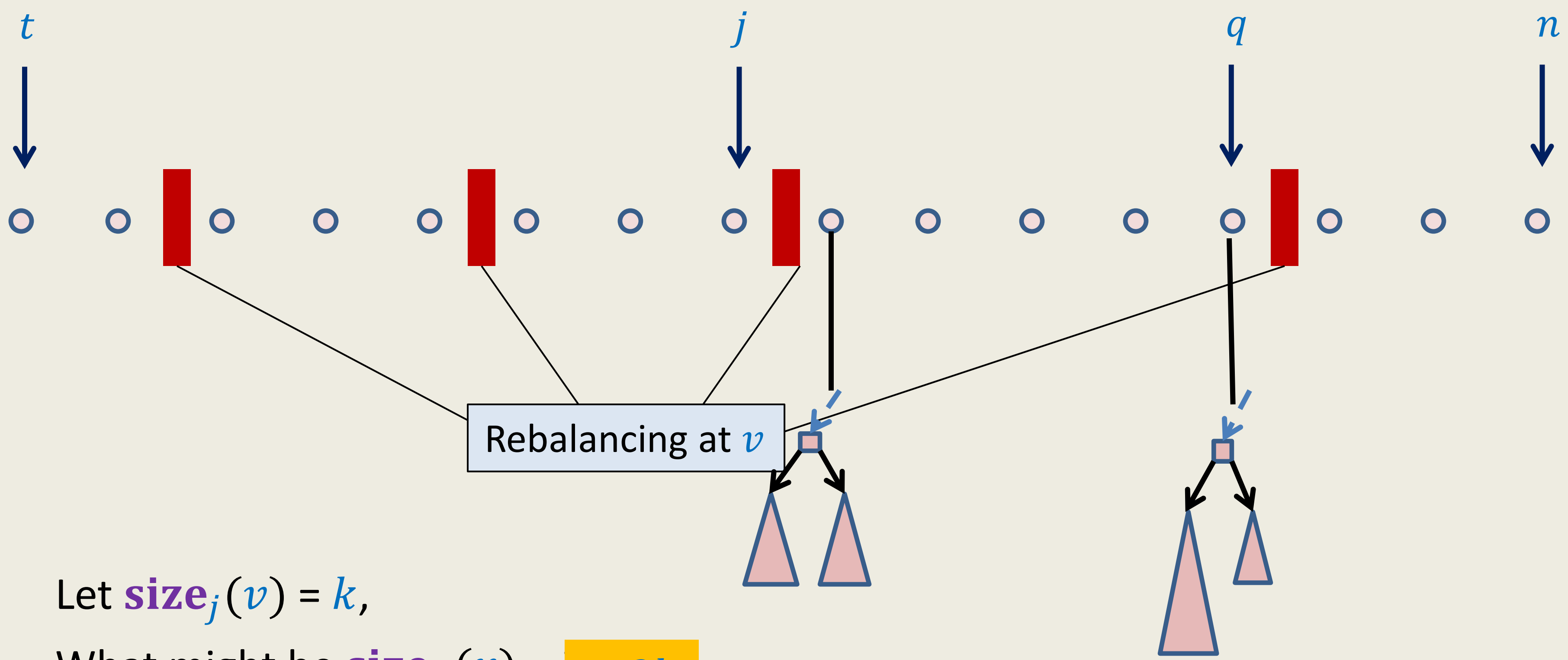
$$I_k(v) = \begin{cases} 1 & \text{if } k\text{th insertion increases } \textbf{size}(v) \\ 0 & \text{Otherwise} \end{cases}$$

Question: For a nearly balanced BST, what is

$$\sum_v I_k(v) = \textbf{O(log } k\textbf{)}$$

$$\sum_{k=1 \text{ to } n} \sum_v I_k(v) = \textbf{O(}n\textbf{ log }n\textbf{)}$$

Journey of an element/node v during n insertions



Let $\mathbf{size}_j(v) = k$,

What might be $\mathbf{size}_q(v)$ = $\geq 2k$

Time complexity of rebalancing $\mathbf{T}(v)$ after qth insertion **= O(k)**

What might be $\sum_{r=j+1}^{q} \boldsymbol{I}_r(v)$ $\geq k$

➔ Time complexity of rebalancing $\mathbf{T}(v)$ after qth insertion = **O(** $\sum_{r=j+1}^{q} \boldsymbol{I}_r(v)$**)**

Time complexity of n insertions

For a vertex v,

Time complexity of **rebalancing T**(v) during n **insertions =** $\sum_{r=t}^{n} I_r(v)$

For all vertices,

the time complexity of **rebalancing** during n **insertions** $= \sum_{v} \sum_{r=t_v}^{n} I_r(v)$

After swapping these two “summations”

$$= \sum_{k=1 \text{ to } n} \sum_{v} I_k(v)$$

= O(n log n)

Theorem:

For any arbitrary sequence of n **insert operations**, total time to maintain nearly balanced BST will be **O(**n **log** n**)**.