Data Structures and Algorithms

(CS210A)

Semester I – **2014-15**

Lecture 8:

Inventing a new Data Structure with

- **Flexibility of lists for updates**
- **Efficiency of arrays for search**

Important Notice

There are basically two ways of introducing a new/innovative solution of a problem. One way is to just explain it without giving any clue as to how the person who invented the concept came up with this solution. Another way is to start from scratch and take a journey of the route which the inventor might have followed to arrive at the solution. This journey goes through various hurdles and questions, each hinting towards a better insight into the problem if we have patience and open mind. Which of these two ways is better ?

I believe that the second way is better and more effective. The current lecture is based on this way. The data structure we shall invent is called a Binary Search Tree. This is the most fundamental and versatile data structure. We shall realize this fact many times during the course …

Doubly Linked List based implementation versus array based implementation of "List"

Operation	Time Complexity per operation for array based implementation	Time Complexity per operation for doubly linked list based implementation
IsEmpty(L)	O(1)	O(1)
Search(x,L)	O(n)	O(n)
Successor(p,L)	O(1)	O(1)
Predecessor(p,L)	O(1)	O(1)
CreateEmptyList(L)	O(1)	O(1)
Insert(x,p,L)	O(n)	O(1)
Delete(p,L)	O(n)	O(1)
MakeListEmpty(L)	O(1)	O(1)

Arrays are very **rigid**

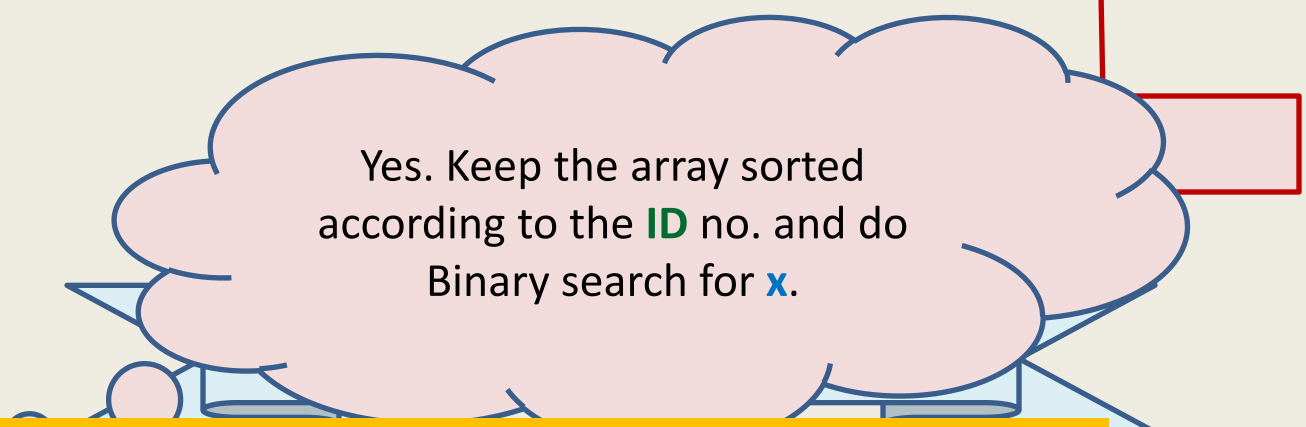
Problem

Maintain a telephone directory

Operations:

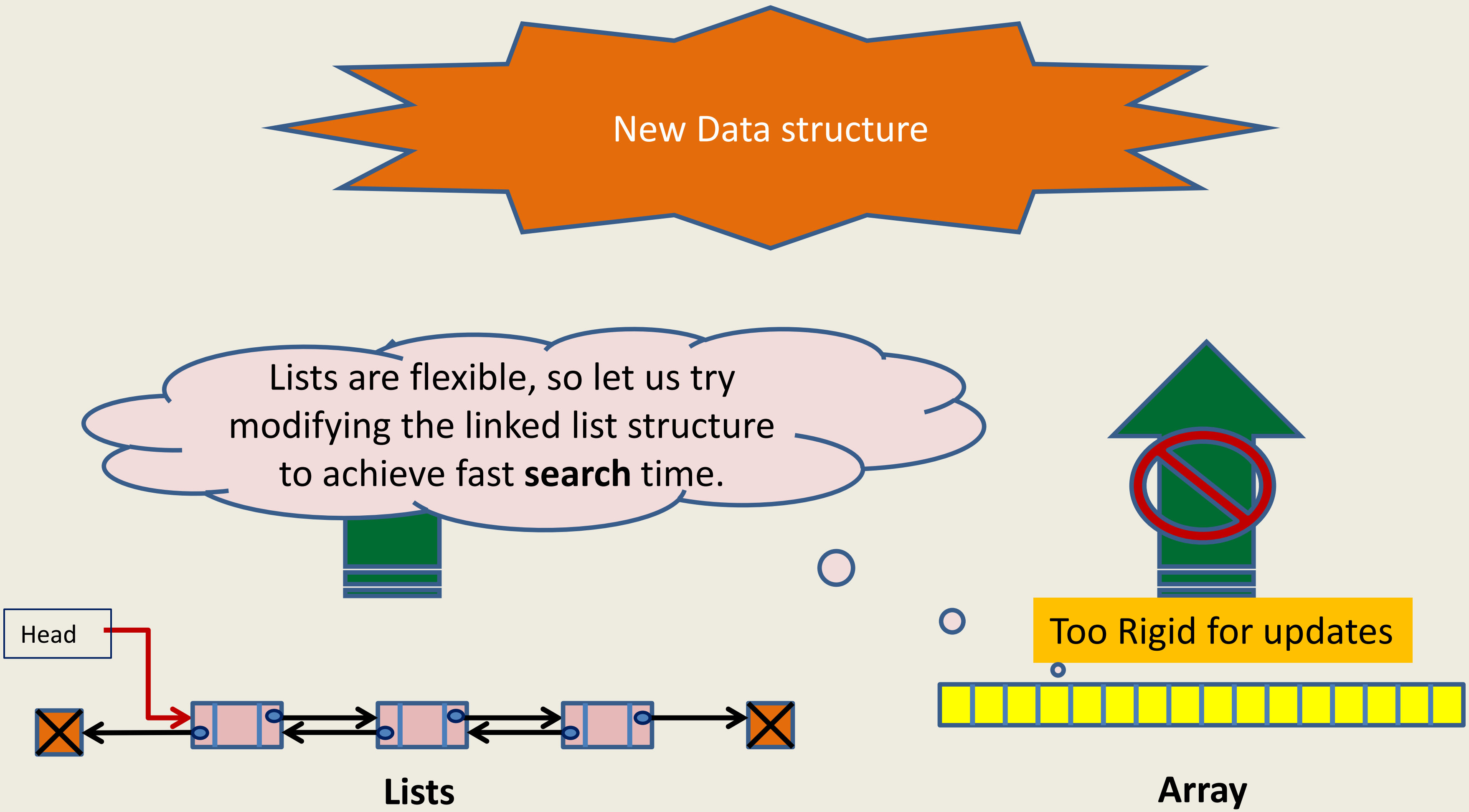
- Search the phone # of a person with ID no. x
- Insert a new record (ID no., phone #,…)

Array based solution	Linked list based solution
Log n	O(n)
O(n)	Log n

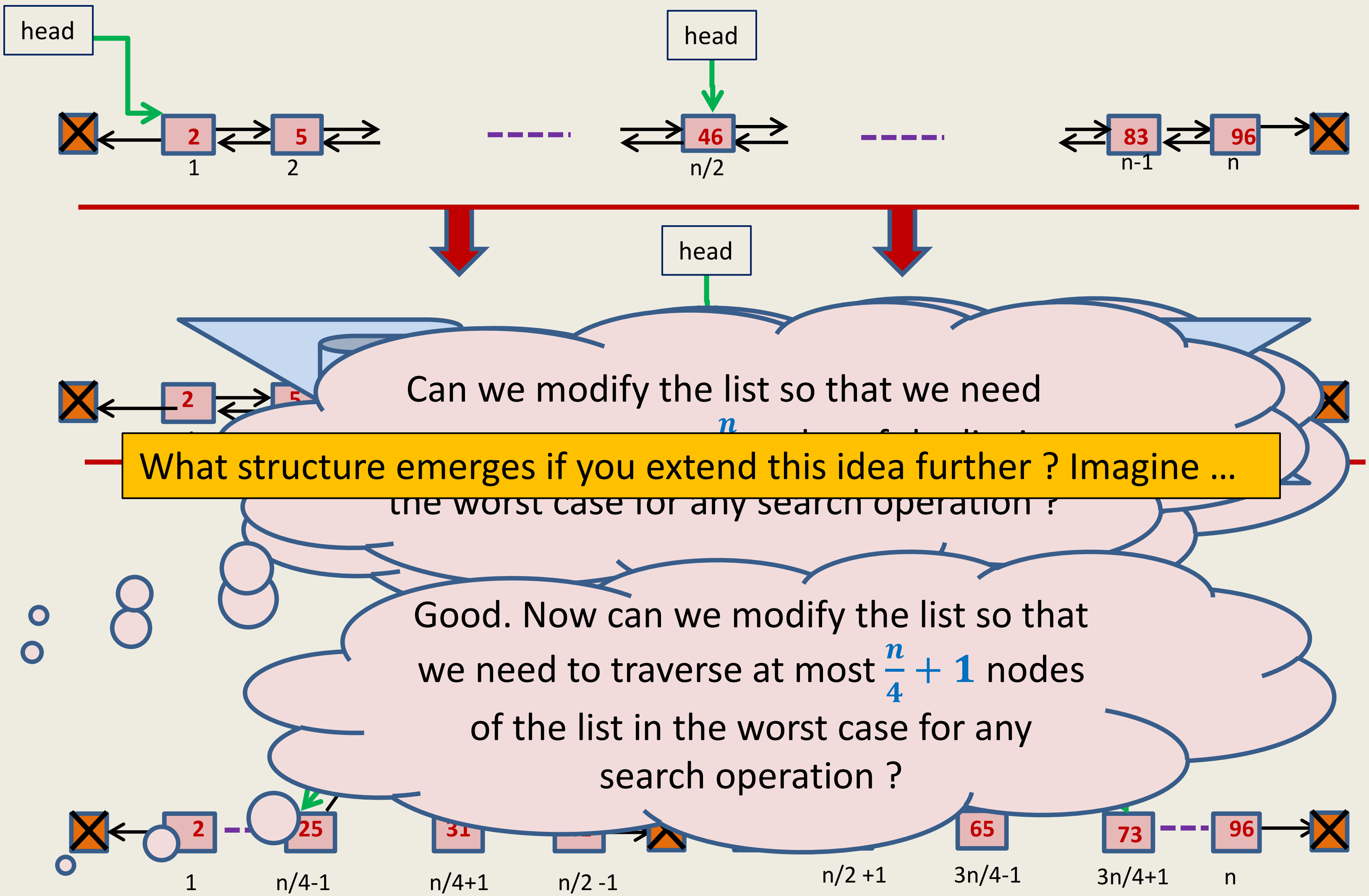


We shall together invent such **a novel data structure** today

Inventing a new data structure



Restructuring doubly linked list



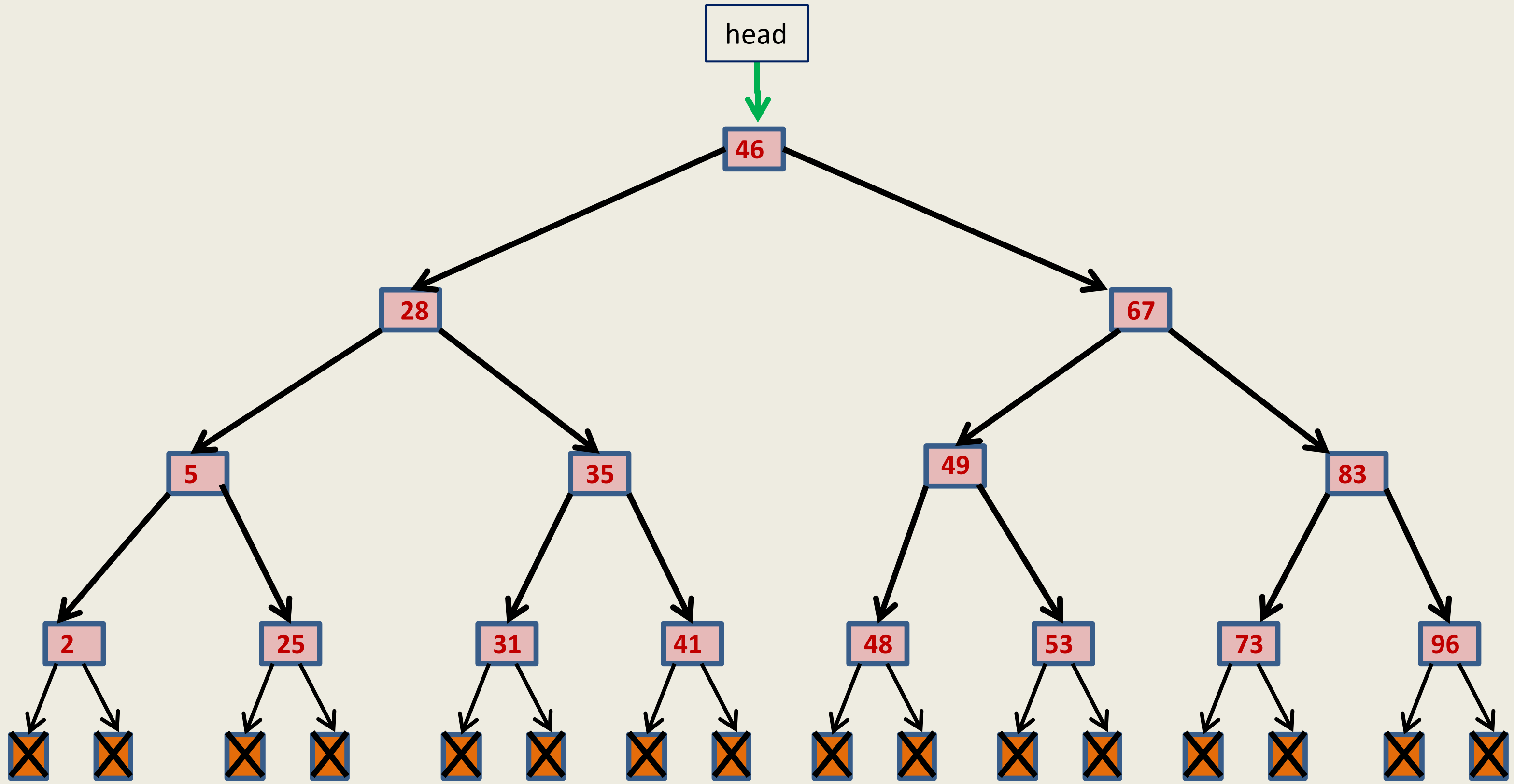
Can we modify the list so that we need

the worst case for any search operation ?

What structure emerges if you extend this idea further ? Imagine …

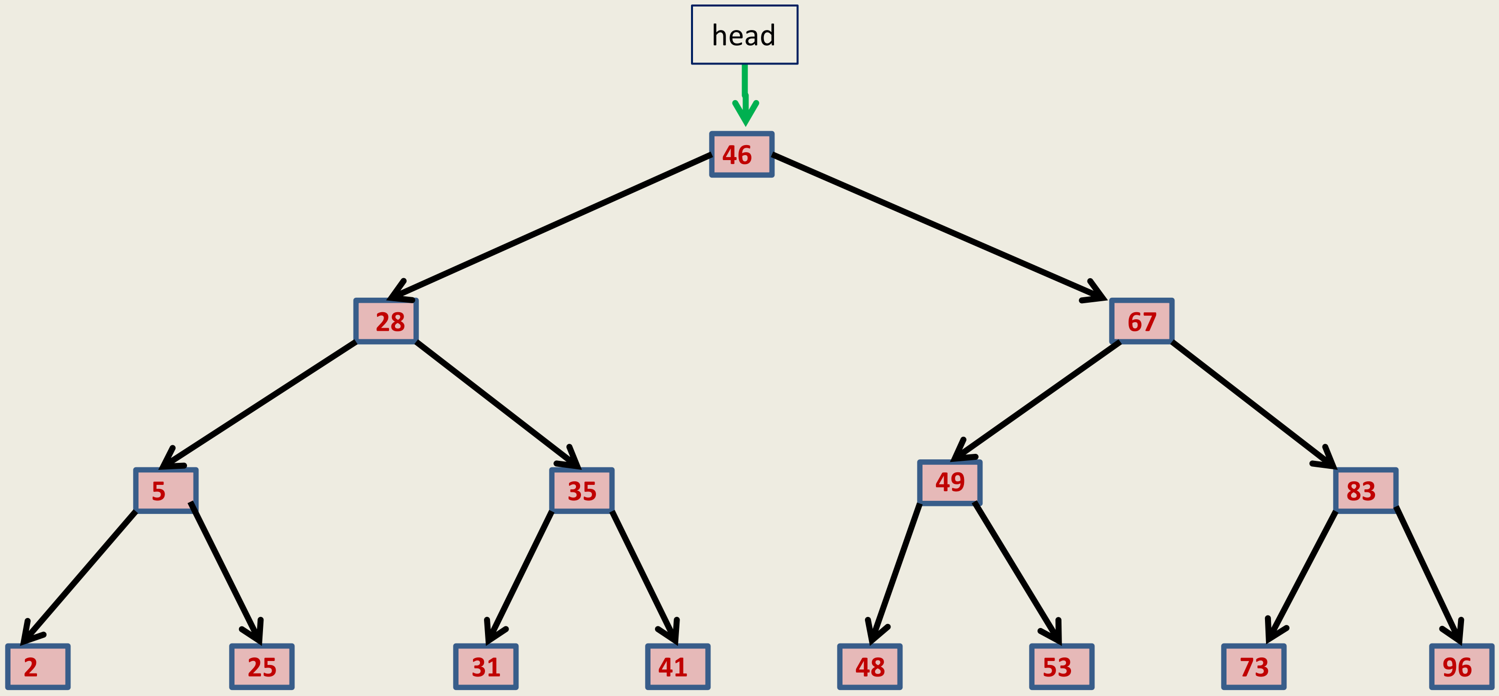
Good. Now can we modify the list so that we need to traverse at most $\frac{n}{4}+1$ nodes of the list in the worst case for any search operation ?

A new data structure emerges

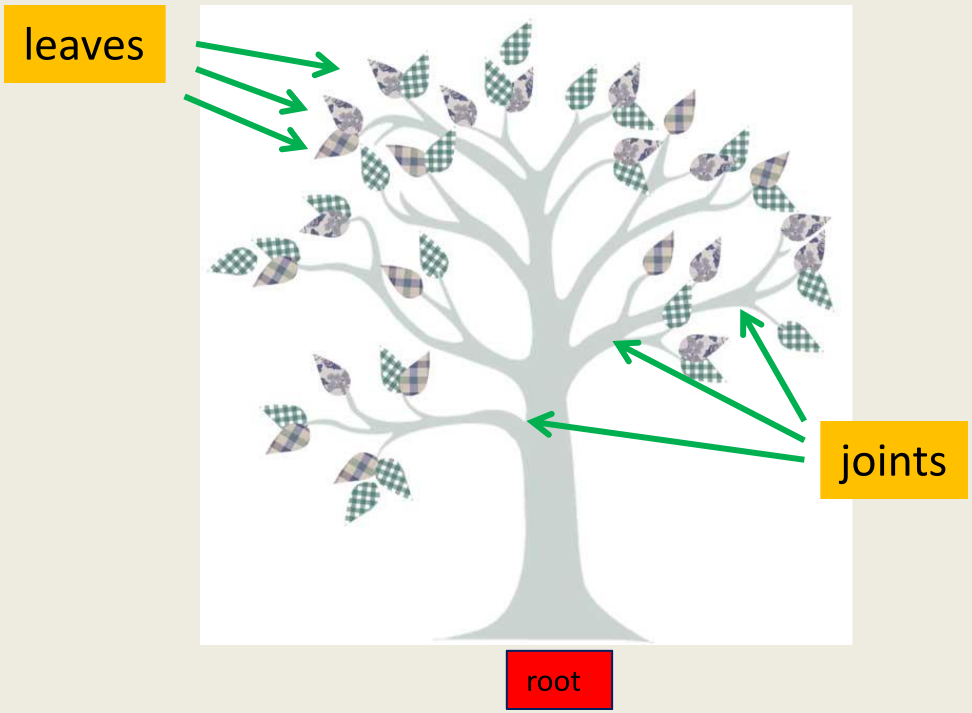


A new data structure emerges

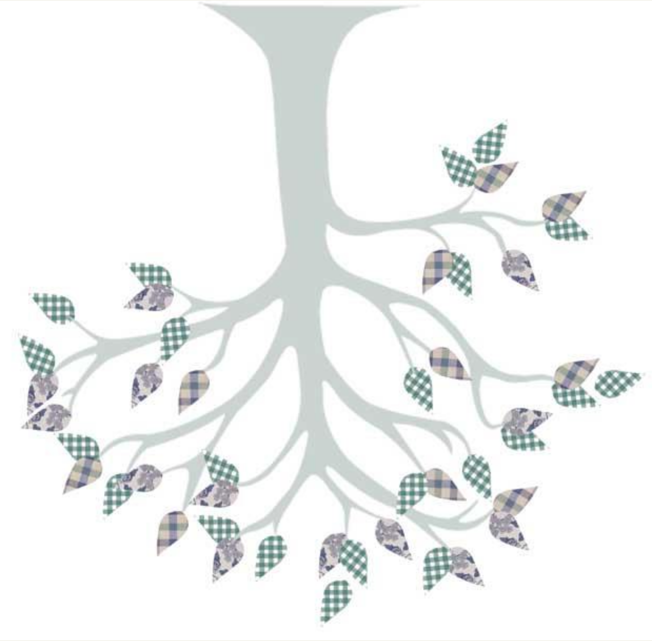
To analyze it mathematically, remove irrlevant details



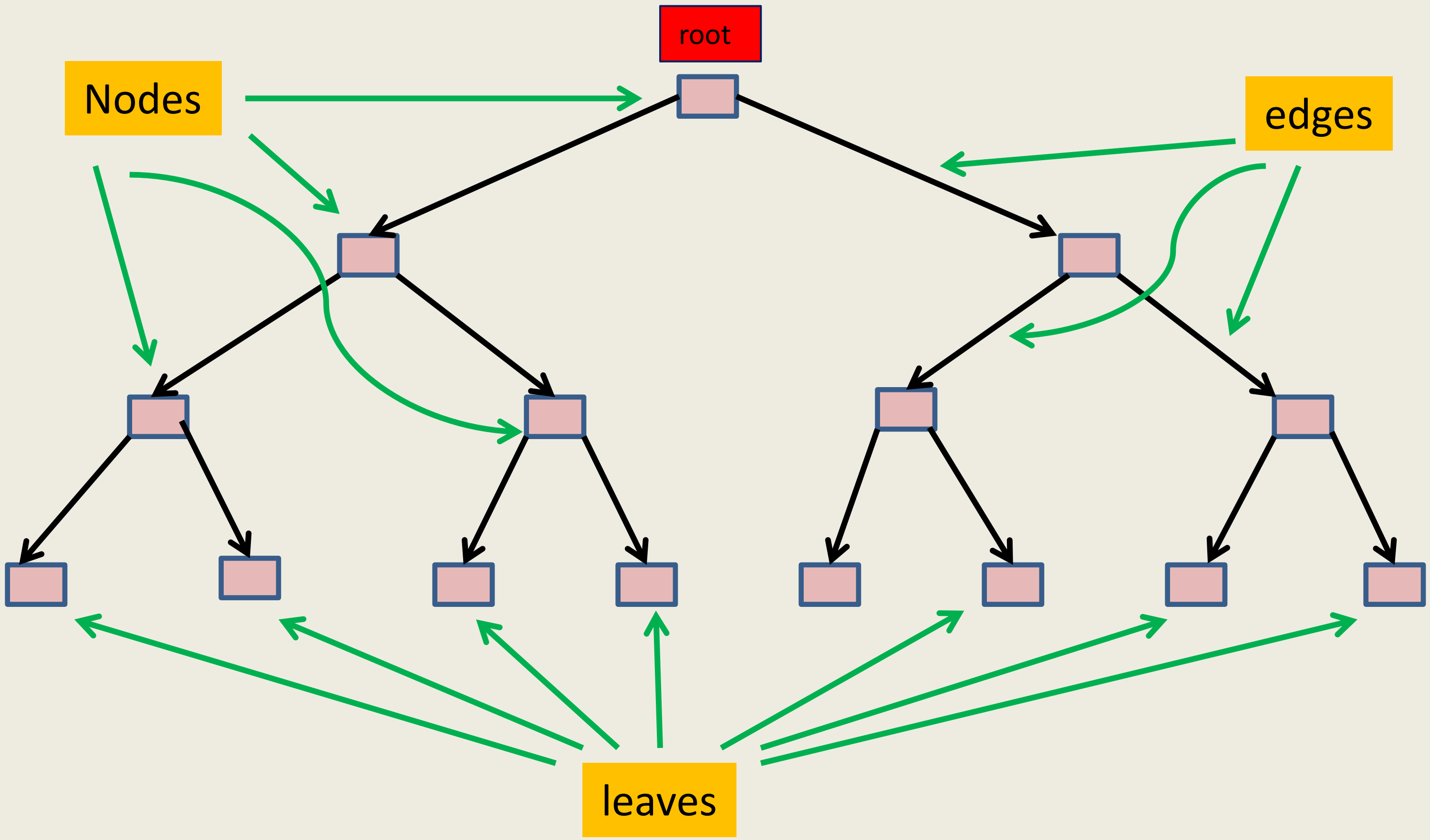
Nature is a great source of inspiration



Nature is a great source of inspiration



Nature is a great source of inspiration



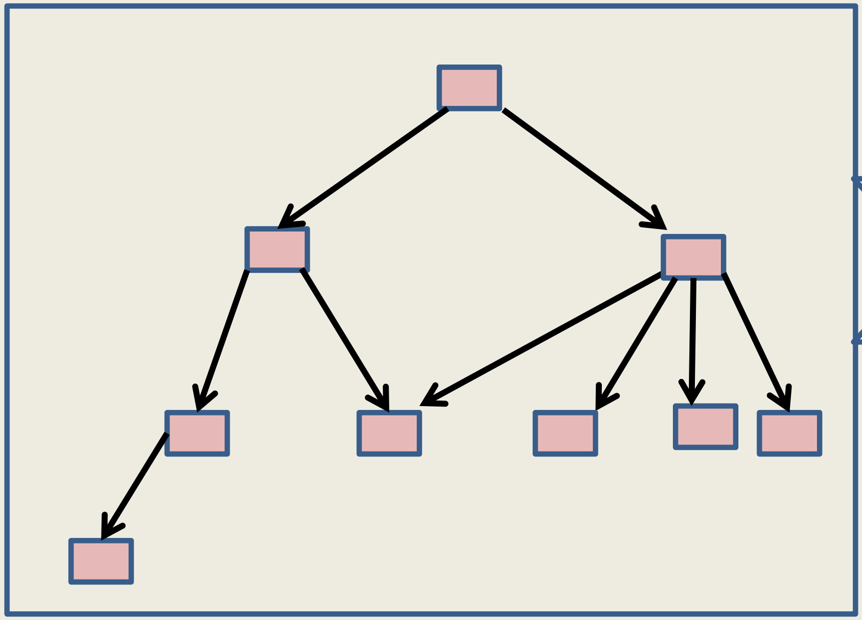
Binary Tree: A mathematical model

Definition: A collection of nodes is said to form a binary tree if

1. There is exactly one node with no incoming edge.

   This node is called the **root** of the tree.

2. Every node other than root node has **exactly one incoming edge**.
3. Each node has **at most** **two outgoing edges**.



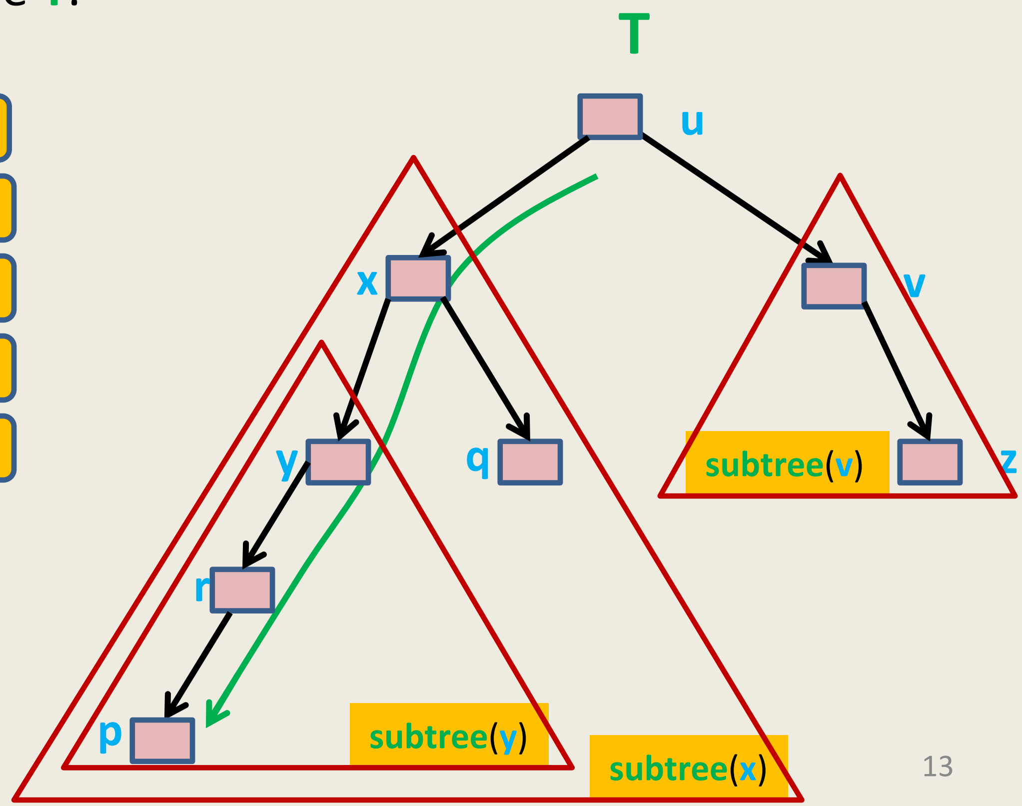
Which of these are **not** binary trees ?

Binary Tree: some terminologies

- If there is an edge from node u to node v,

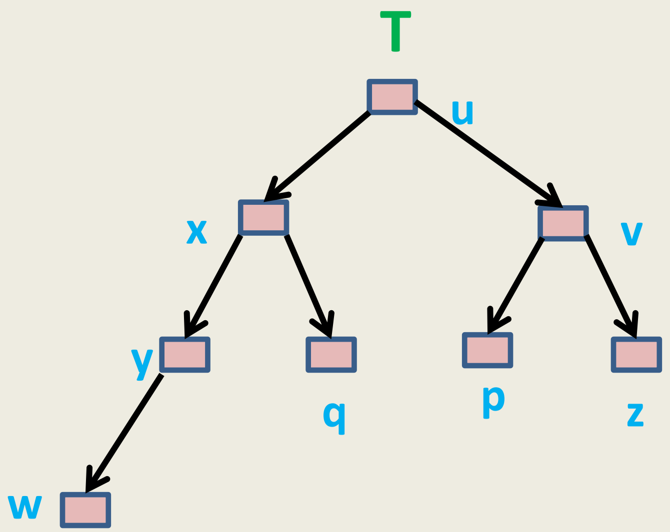
  then u is called **parent** of v ,and v is called **child** of u.
- The **Height** of a Binary tree **T** is the <u>maximum</u> number of edges from the root to any leaf node in the tree **T**.

parent(y)	=	**x**
parent(v)	=	**u**
children(y)	=	**{r}**
children(x)	=	**{y,q}**
height(T)	=	**4**



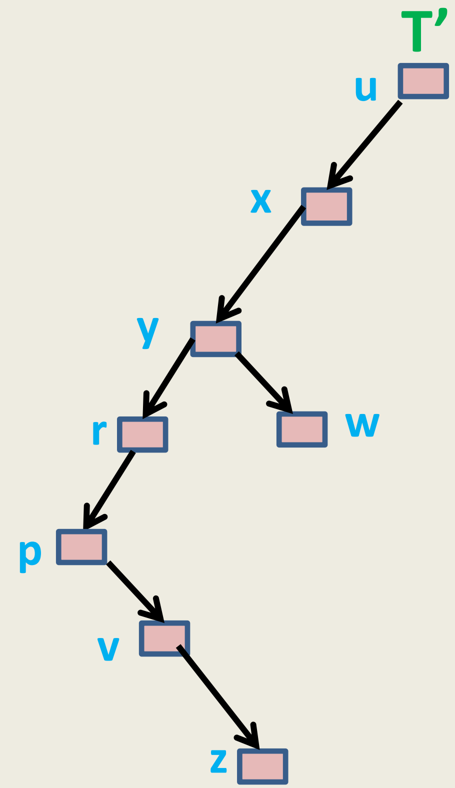
Varieties of Binary trees

We call it **Perfectly balanced**

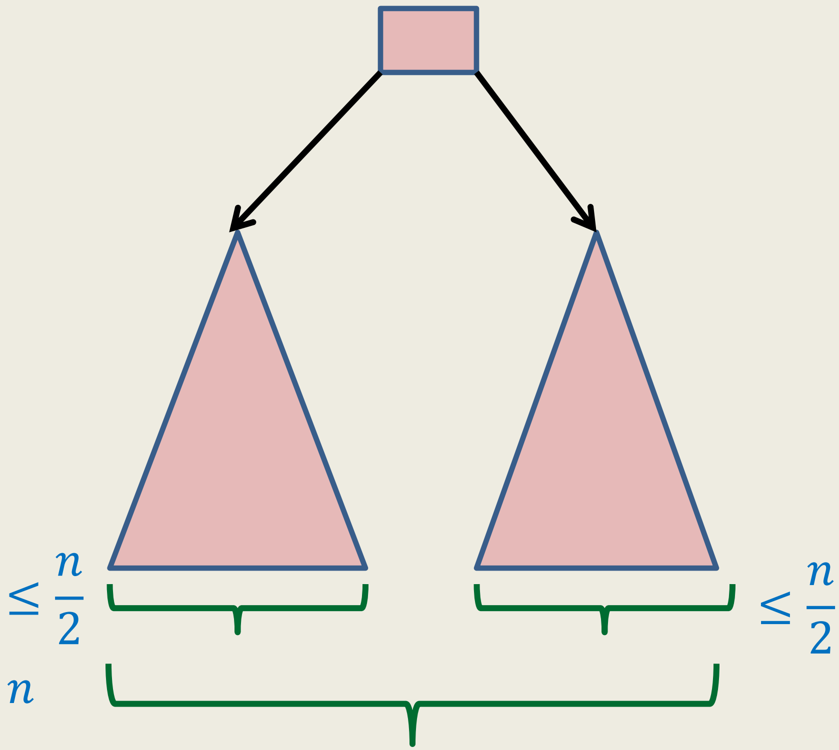


For every node, the number of nodes in the **subtrees of its two children** differ at **atmost** by 1.

skewed



Height of a perfectly balanced Binary tree

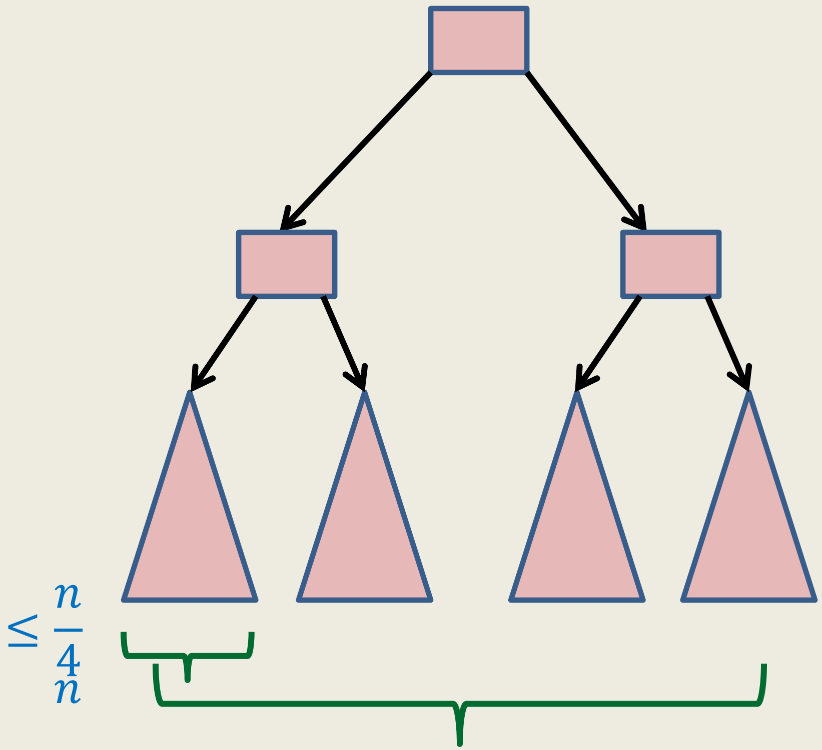


$\boldsymbol{H}(n)$: Height of a perfectly balanced binary tree on n nodes.

$\boldsymbol{H}(1) = 0$

$\boldsymbol{H}(n) \leq \mathbf{1} + \boldsymbol{H}\left(\frac{n}{2}\right)$

Height of a perfectly balanced Binary tree



$\boldsymbol{H}(n)$: Height of a perfectly balanced binary tree on n nodes.

$\boldsymbol{H}(1) = 0$

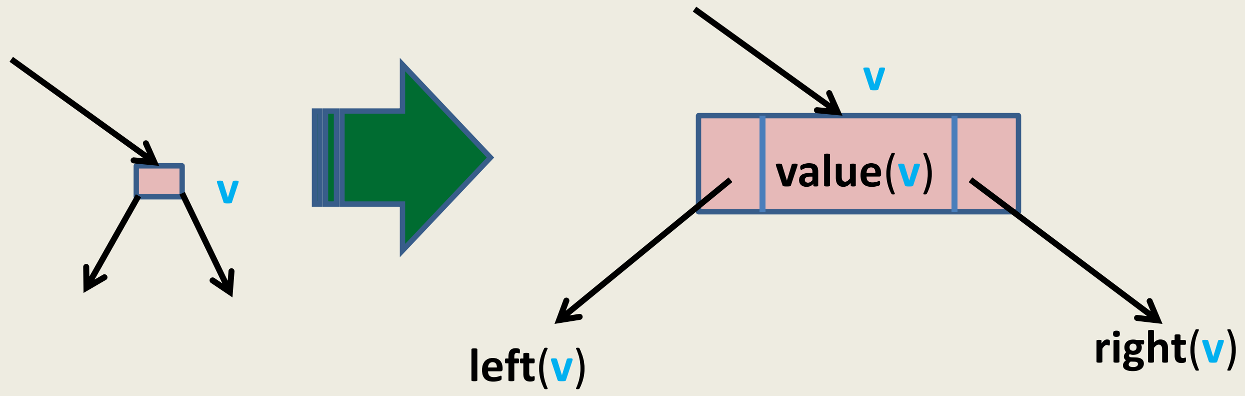
$$\boldsymbol{H}(n) \leq \mathbf{1} + \boldsymbol{H}\left(\frac{n}{2}\right)$$

$$\leq \mathbf{1} + \mathbf{1} + \boldsymbol{H}\left(\frac{n}{4}\right)$$

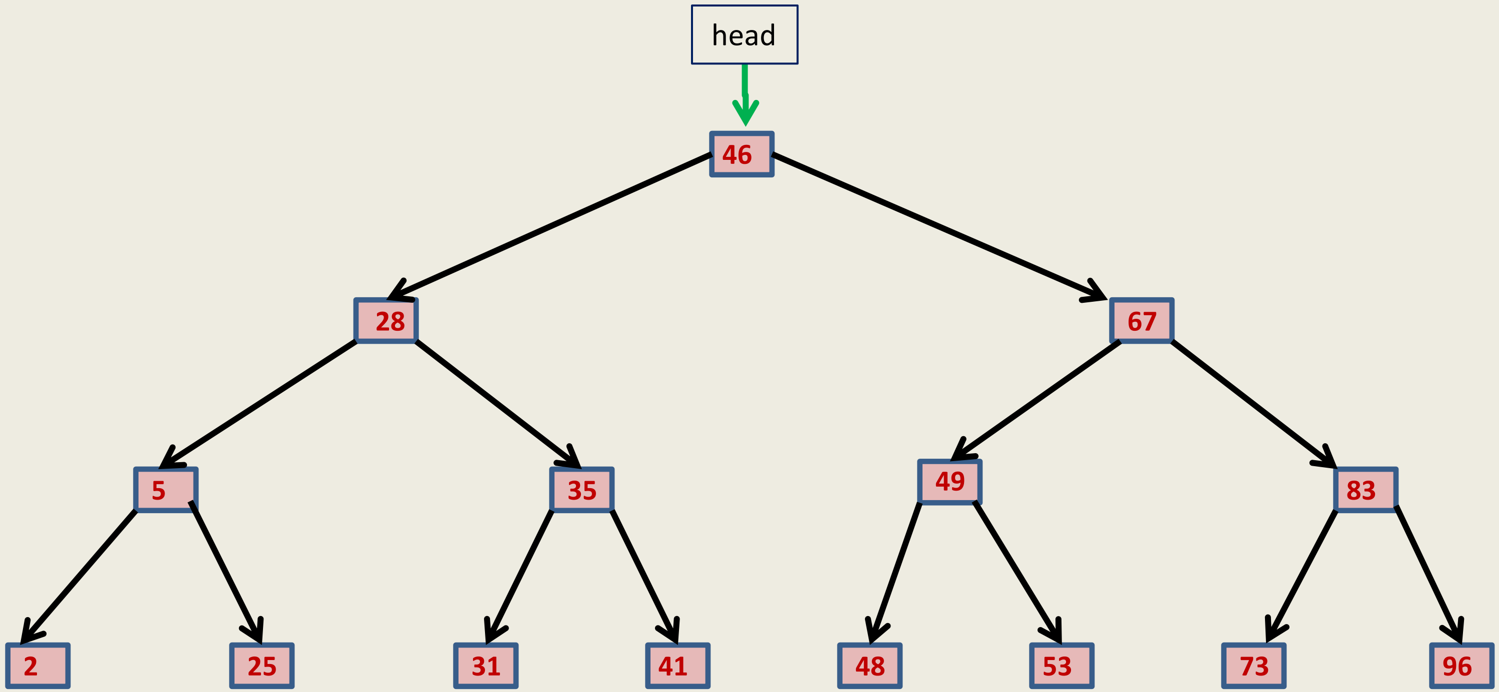
$$\leq \underbrace{\mathbf{1} + \mathbf{1} + \cdots +}_{i} \boldsymbol{H}\left(\frac{n}{2^i}\right)$$

$$\leq \mathbf{log}_2\, n$$

Implementing a Binary trees



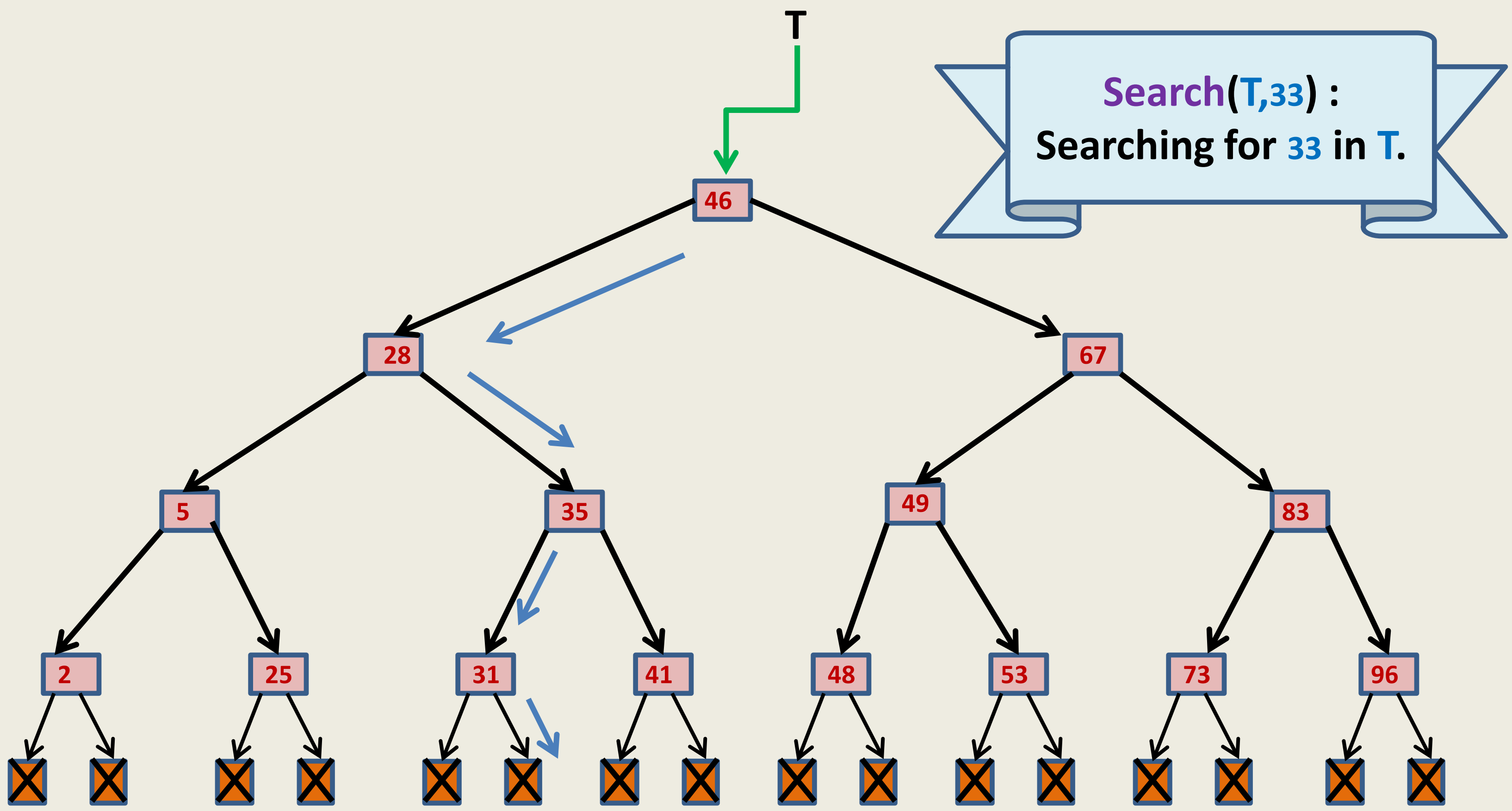
Binary Search Tree (BST)



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**)).

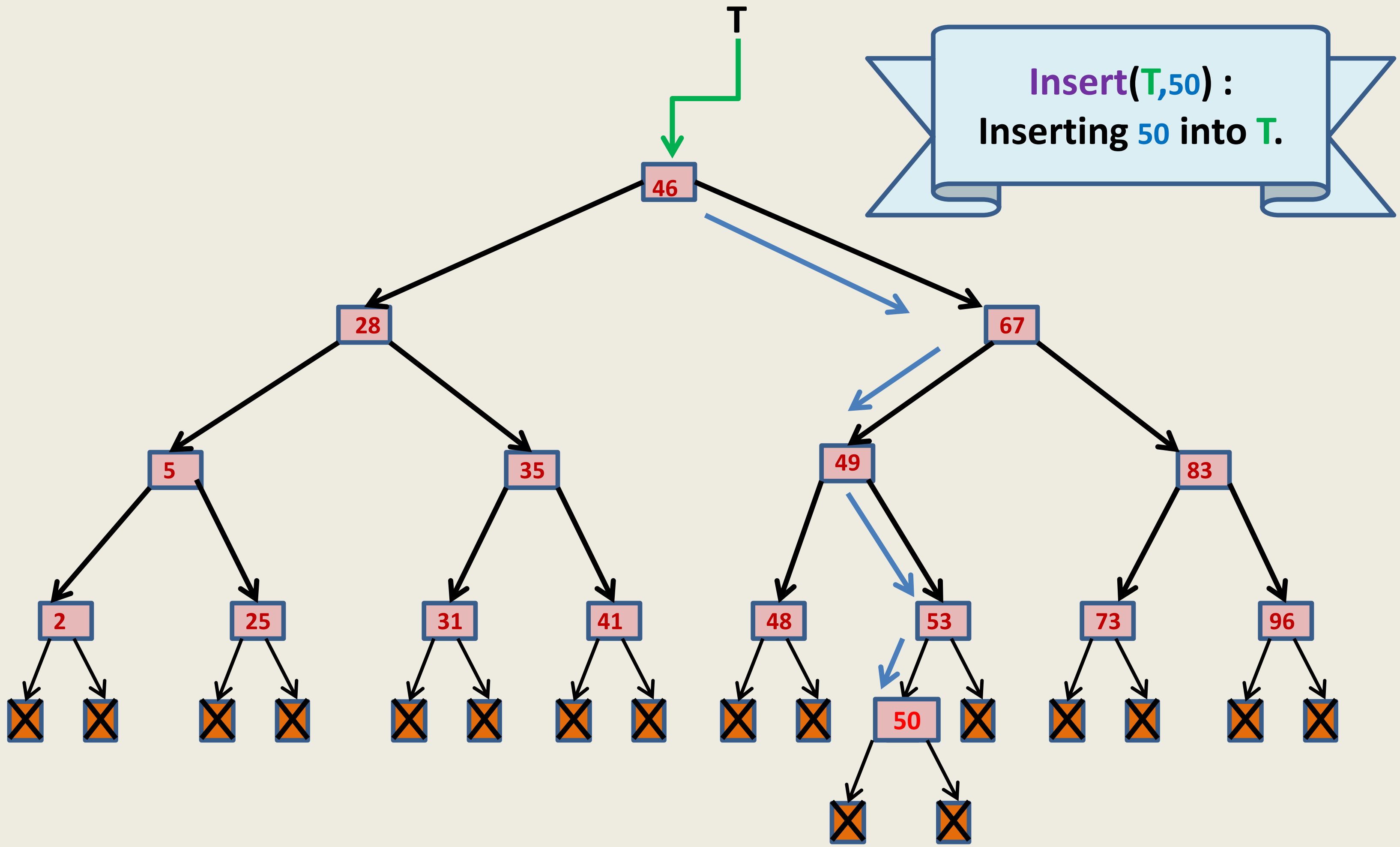
Search(T,x)
Searching in a Binary Search Tree



Search(T,x)
Searching in a Binary Search Tree

```
Search(T,x)
{     p← T;
      Found← FALSE ;
      while( Found= FALSE & p<> NULL )
      {       if(value(p) = x)  Found← TRUE ;
              else if (value(p) < x)  p← right(p) ;
                      else  p← left(p) ;
      }
      return p;
}
```

Insert(T,x)
Insertion in a Binary Search Tree



A question

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

Homeworks

- Write pseudocode for **Insert**(**T**,x) operation similar to the pseudocode we wrote for **Search**(**T**,x).

- Design an algorithm for the following problem:

Given a sorted array **A** storing n elements,

build a "perfectly balanced" BST storing all elements of **A**

in **O**(n) time.

Homework 3

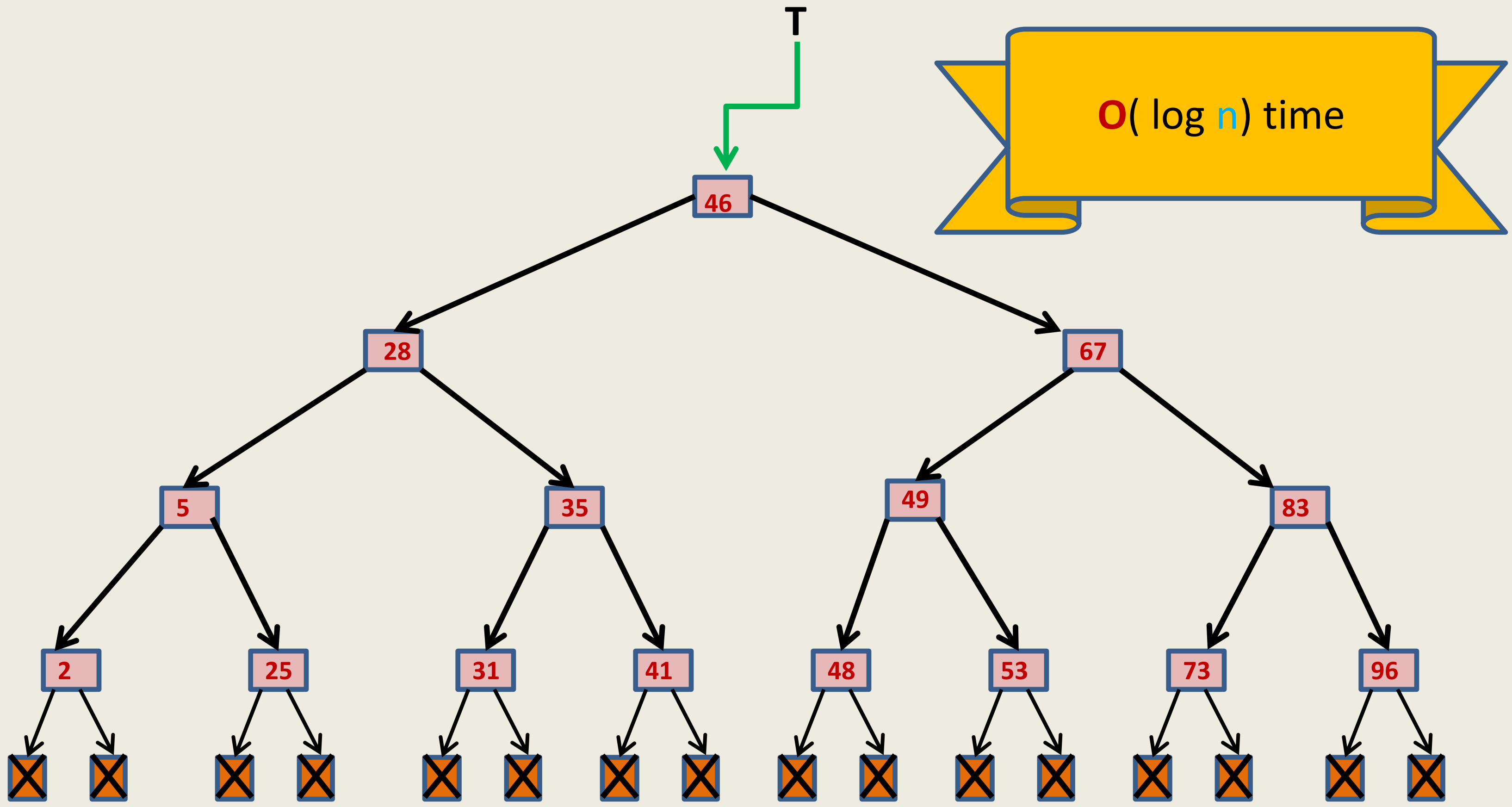
What does the following algorithm accomplish ?

```
Traversal(T)
{     p← T;
      if(p=NULL) return;
      else{     if(left(p) <> NULL)    Traversal(left(p));
                print(value(p));
                if(right(p) <> NULL)    Traversal(right(p));
            }
}
```

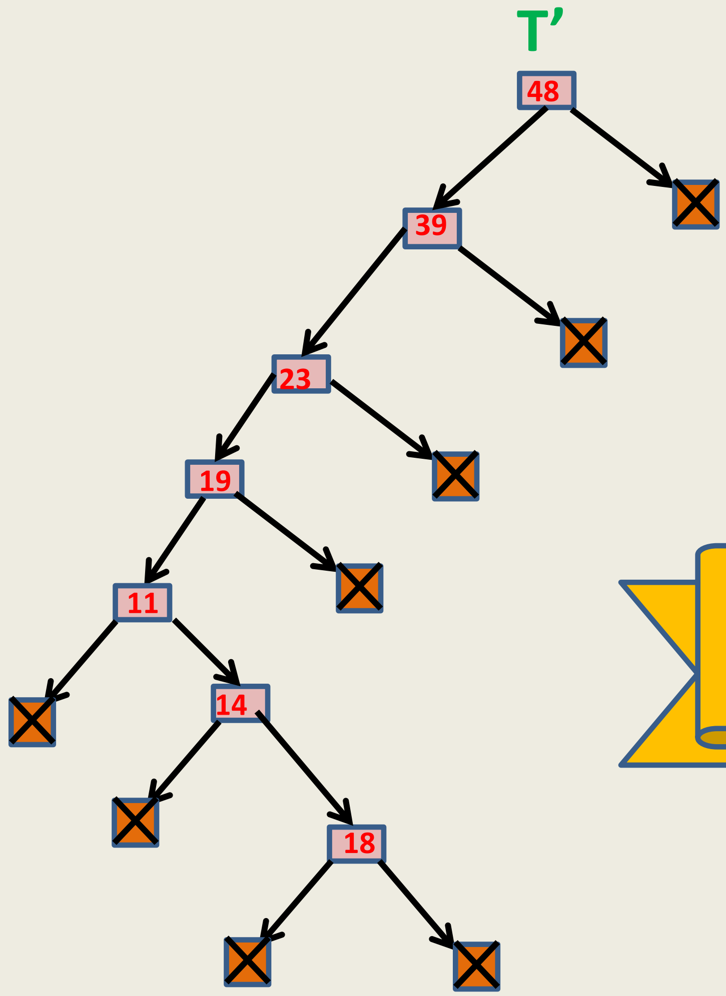
Ponder over this algorithm for a few minutes to know what it is doing. You might like to try it out on some example of BST.

It prints the elements of binary search tree T in increasing order of their values. What is its time complexity ?

Time complexity of any search and any single insertion in a perfectly balanced Binary Search Tree on n nodes



Time complexity of any search and any single insertion in a sqewed Binary Search Tree on n nodes



O(n) time !☹

Our original Problem

Maintain a telephone directory

Operations:

- Search the phone # of a person with **ID** no. **x**
- Insert a new record (**ID** no., phone #,…)

Array based solution	Linked list based solution
Log n	**O(n)**
O(n)	**Log n**

Solution : We may keep **perfectly balanced** BST.

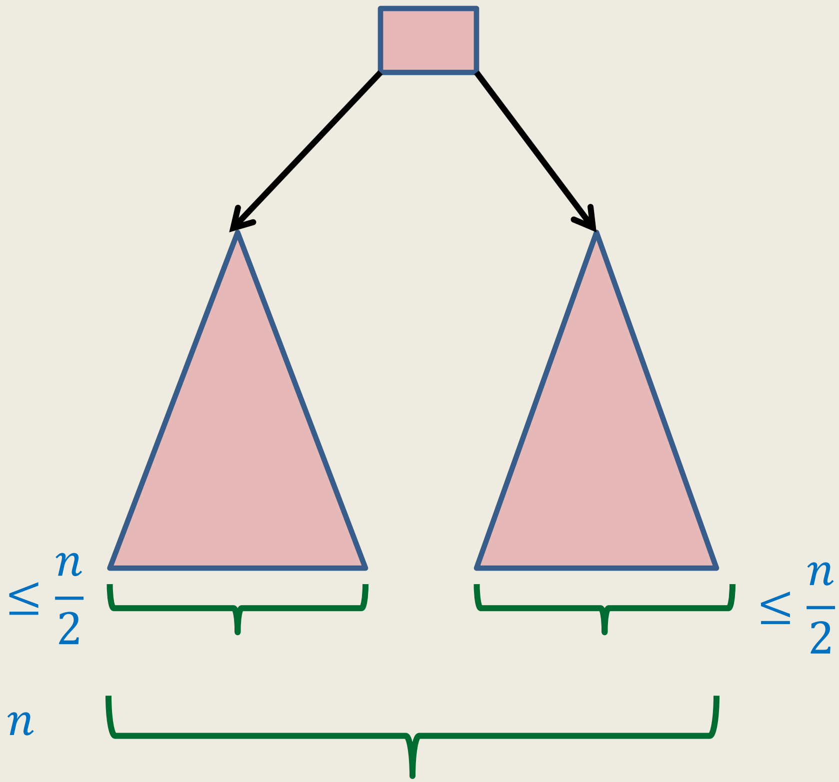
Hurdle: What if we insert records in increasing order of **ID** ?

➔ BST will be skewed ☹

BST data structure that we invented looks very elegant,
let us try to find a way to overcome the hurdle.

- Let us try to find a way of achieving **Log** n search time.
- Perfectly balanced BST achieve **Log** n search time.
- But the definition of Perfectly balanced BST looks **too restrictive**.
- Let us investigate : How crucial is perfect balance of a BST ?

How crucial is perfect balance of a BST ?

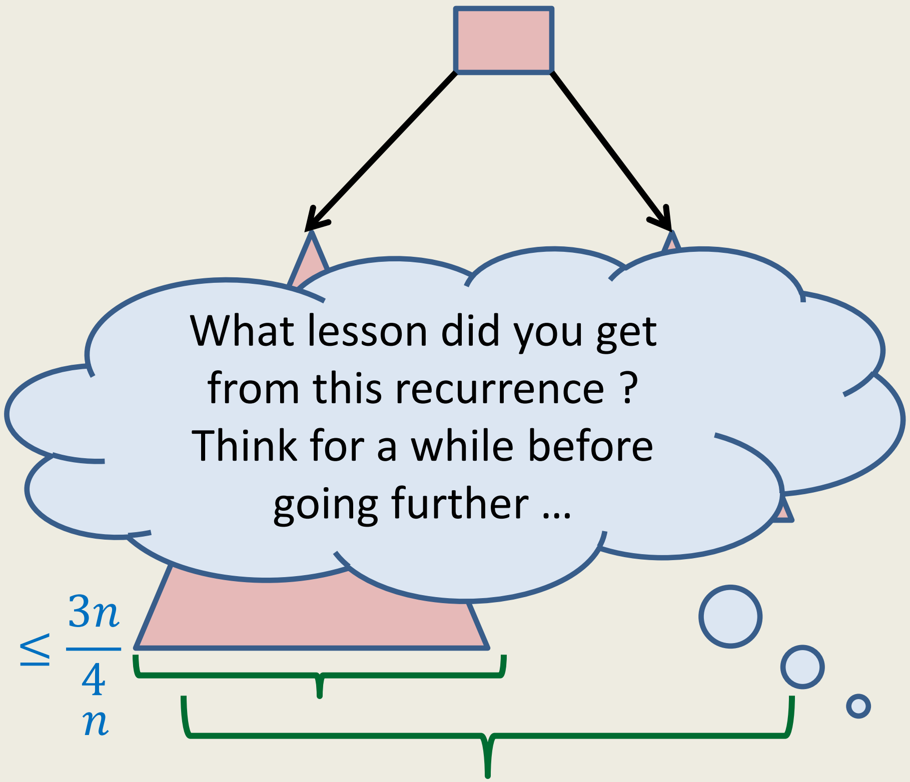


$\boldsymbol{H}(1) = 0$

$\boldsymbol{H}(n) = \leq \mathbf{1} + \boldsymbol{H}\left(\frac{n}{2}\right)$

Let us change this recurrence slightly.

How **crucial** is perfect **balance** of a BST ?



$H(1) = 0$

$$H(n) \leq \mathbf{1} + H\left(\frac{3n}{4}\right)$$

$$\leq \mathbf{1} + \mathbf{1} + H\left(\left(\frac{3}{4}\right)^2 n\right)$$

$$\leq \mathbf{1} + \mathbf{1} + \cdots + H\left(\left(\frac{3}{4}\right)^i n\right)$$

$$\leq \mathbf{log}_{4/3}\, n$$

Lesson learnt :
We may as well work with **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**)) $\leq \frac{3}{4}$ **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

Think of ways of using nearly balanced **BST** for solving our dictionary problem.

You might find the following **observations/tools** helpful :

- If a node **v** is perfectly balanced, it requires **many insertions** till **v** ceases to remain nearly balanced.
- Any arbitrary **BST** of size n can be converted into a perfectly balanced **BST** in **O**(n) time.

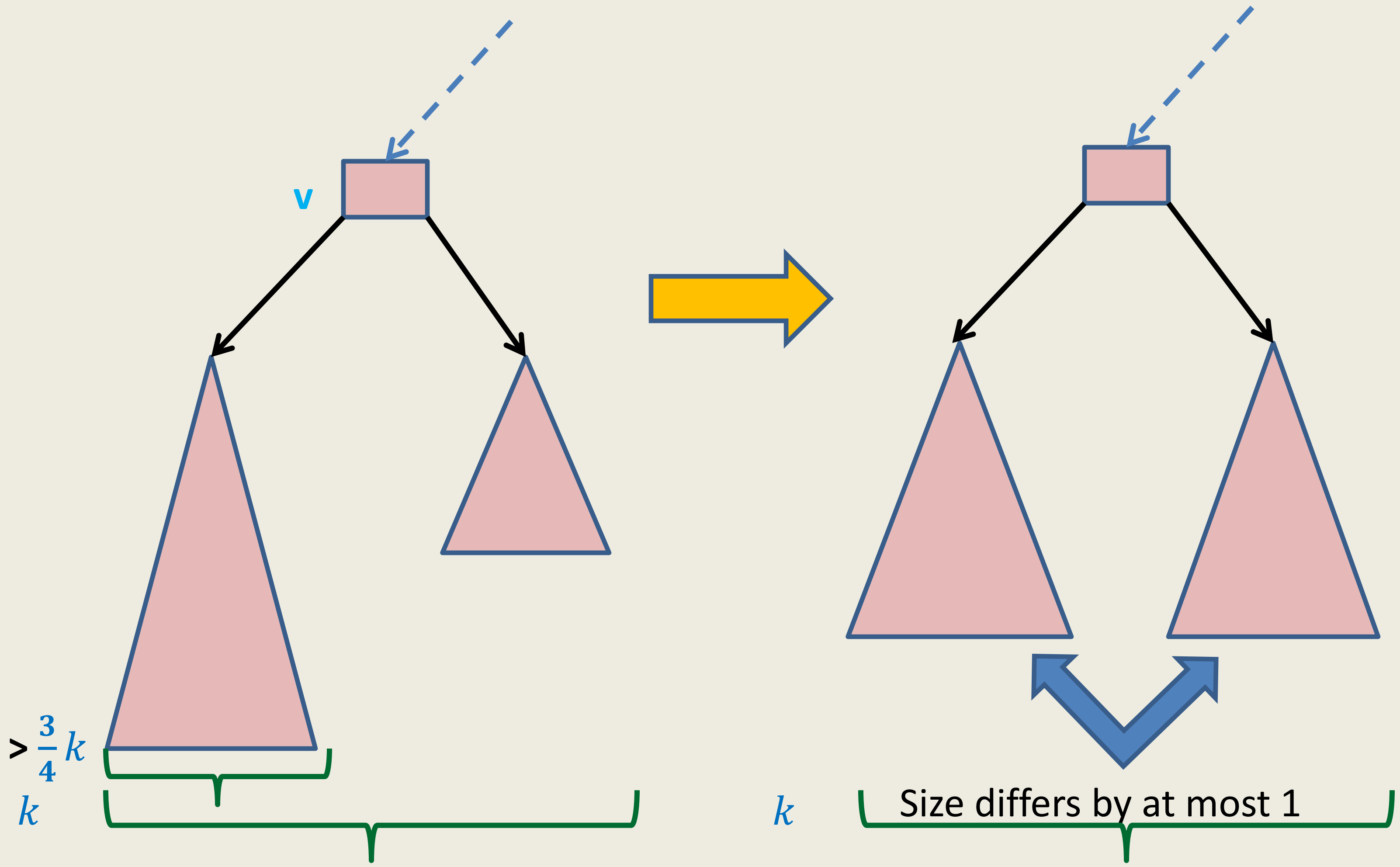
Solving our dictionary problem

Preserving O(log n) height after each operation

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**



What can we say about this data structure ?

It is elegant and reasonably simple to implement.

Yes, there will be huge computation for some insertion operations.

But the number of such operations will be rare.

So, at least intuitively, the data structure appears to be efficient.

Indeed, this data structure achieve the following goals:

- For any arbitrary sequence of n **operations**, total time will be **O(n log n)**.
- Worst case search time: **O(log n)**

You will do programming assignment to verify the validity of the two claims mentioned above experimentally.

What about the theoretical analysis to justify these claims ?

Keep thinking till we do it in a few weeks ☺.