Data Structures and Algorithms

(CS210A)

Lecture 20

Red Black tree (Final lecture)

- **9 types of operations**

  each executed in **O(log n)** time !

Red Black tree

(Height Balanced BST)

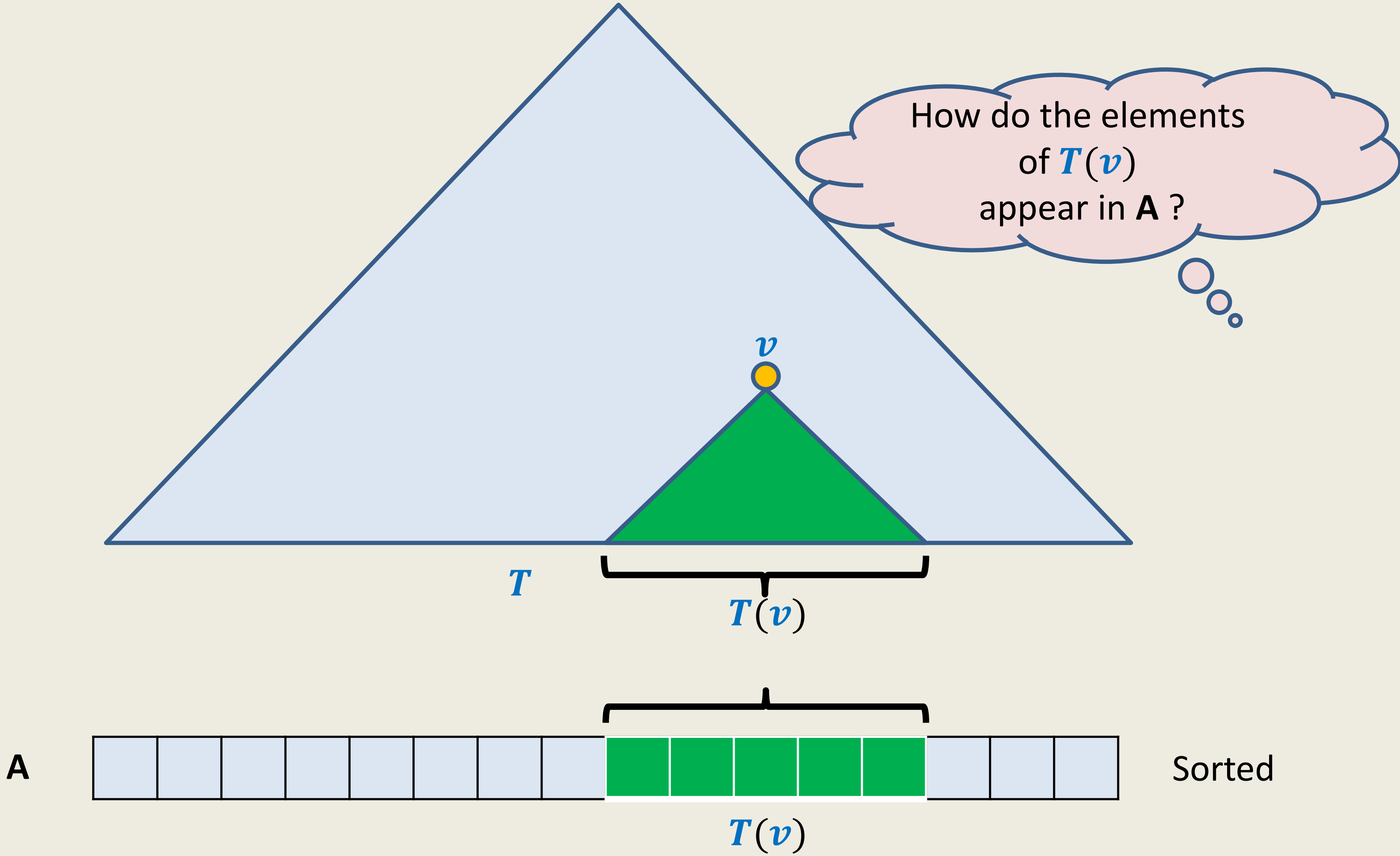
Operations you already know

1. Search(T,x)
2. Insert(T,x)
3. Delete(T,x)
4. Min(T)
5. Max(T)

Every operation in O(log n) time.

Binary Search Tree

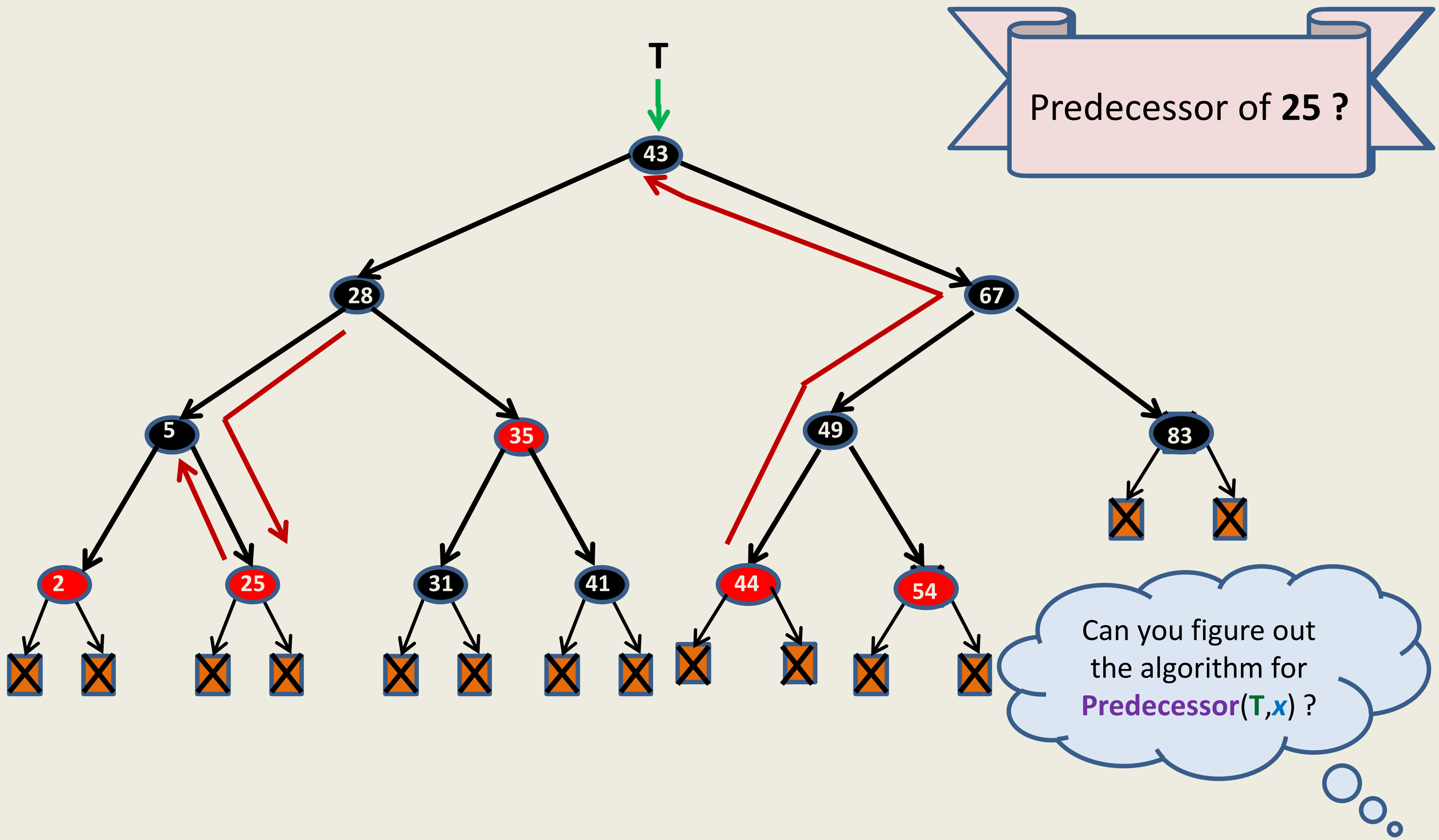
How well have you understood ?


How do the elements of $T(v)$ appear in **A** ?
v
T
$T(v)$
A
Sorted
$T(v)$

Predecessor(T,x)

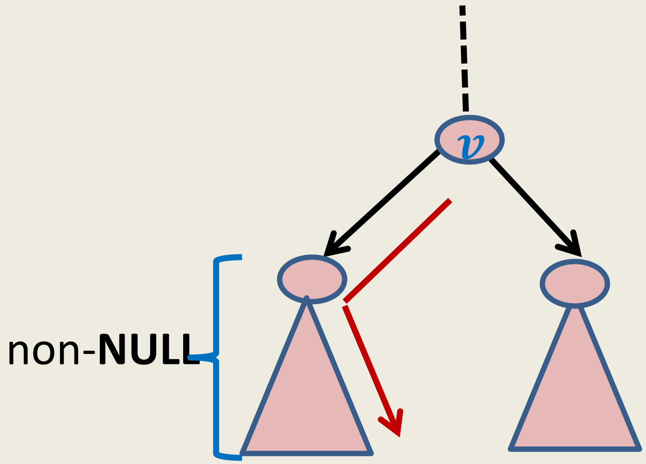
The **largest** element in **T** which is **smaller than x**

Predecessor(T,x)



Predecessor(T,x)

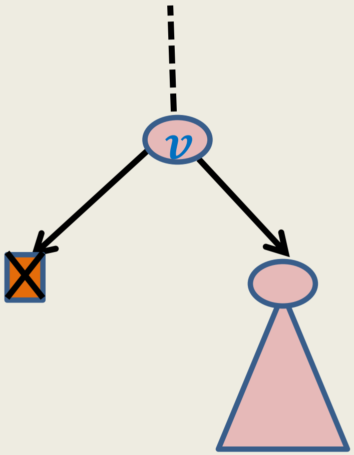
Let v be the **node** of **T** storing value x.



Case 1: **left**(v) <> **NULL** , then **Predecessor(T,x)** is **Max(left(v))**

Predecessor(T,x)

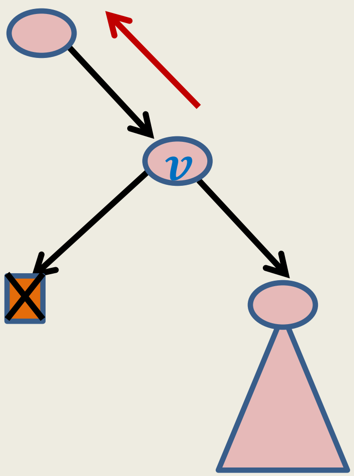
Let v be the **node** of **T** storing value x.



Case 2: **left**(v) == **NULL** , then **Predecessor**(**T**,x) is ?

Predecessor(T,x)

Let v be the **node** of **T** storing value x.

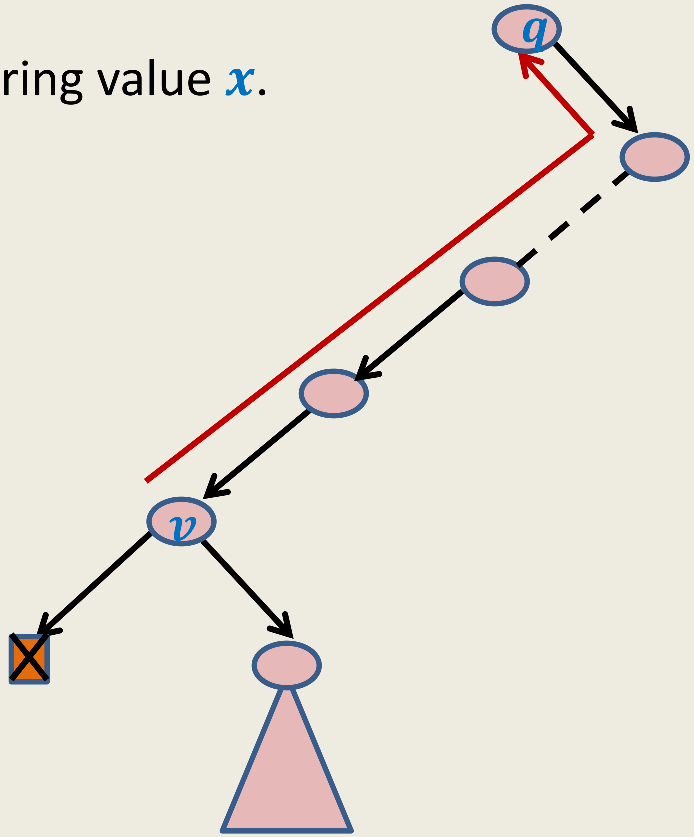


Case 2: **left**(v) == **NULL** , and v is **right child** of its **parent**

then **Predecessor(T,x)** is **parent**(v)

Predecessor(T,x)

Let v be the **node** of **T** storing value x.



Case 3: **left**(v) == **NULL** , and v is **left child** of its **parent**

then Predecessor(T,x) is ?

Predecessor(T,x)

```
Predecessor(T,x)
{   Let v be the node of T storing value x.
    If (left(v) <> NULL)  then return  Max(left(v))
    else
        if (v = right (parent(v)) return  parent(v)
        else
        {
            while(v = left (parent(v))
                v ← parent(v);
            return parent(v);
        }
}
```

Predecessor(T,x)

```
Predecessor(T,x)
{   Let v be the node of T storing value x.
    If (left(v) <> NULL)  then return  Max(left(v))
    else
    {       while(v = left (parent(v))
                v ← parent(v);
            return parent(v);
    }
}
```

Homework 1: Modify the code so that it runs even when x is minimum element.

Homework 2: Modify the code so that it runs even when $x \notin$ T .

Successor(T,x)

The **smallest** element in **T** which is **bigger than x**

Red Black tree

(Height Balanced BST)

Operations you already know

1. **Search**(T,x)
2. **Insert**(T,x)
3. **Delete**(T,x)
4. **Min**(T)
5. **Max**(T)
6. **Predecessor**(T,x)
7. **Successor**(T,x)

A NOTATION

T < T' :

every element of T is <u>smaller</u> than every element of T'.

New operations

8. **SpecialUnion**(T, T'):

Given T and T' such that T **<** T',

compute T*=T**U**T'.

NOTE: T and T' don't exist after the union.

9. **Split**(T, x):

Split T into T' and T'' such that T' < **x** < T''.

Every operation in O(log n) time.

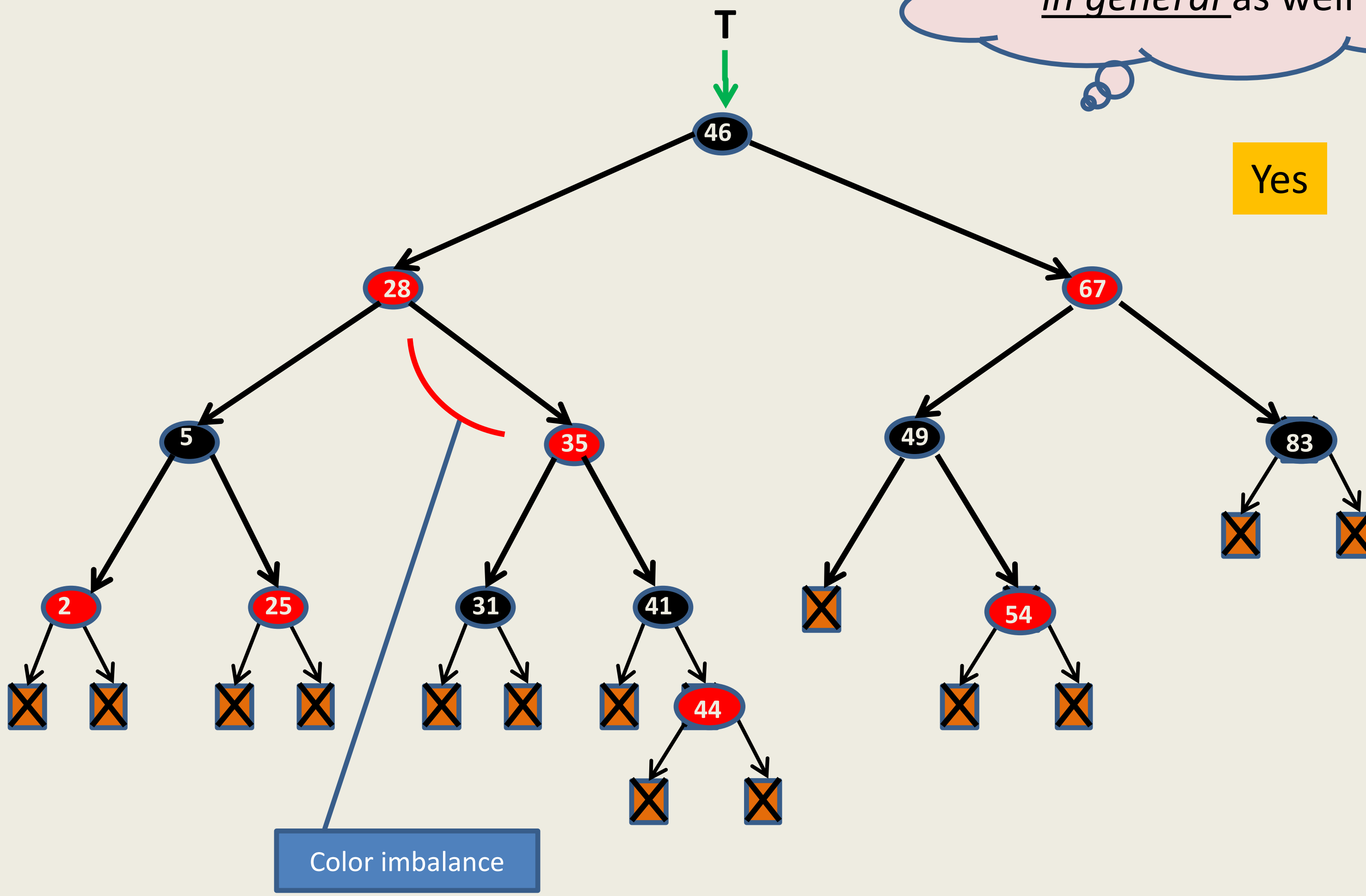
Red-Black Tree

How well have you understood ?

Insertion in a red-black tree

Can you handle it
in general as well ?

Yes

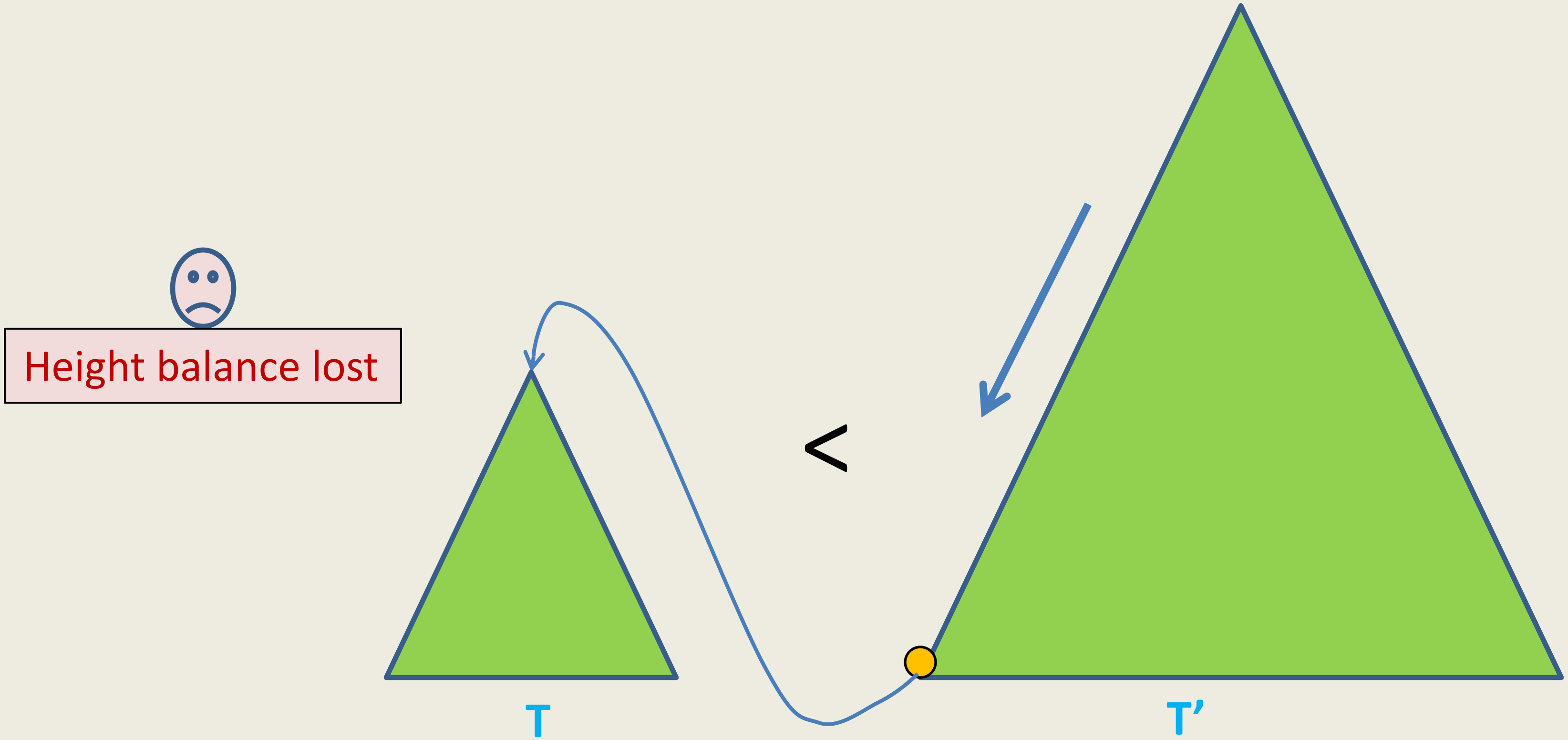


SpecialUnion(T,T')

Remember:

every element of T is smaller than every element of T'

A trivial algorithm that does not work



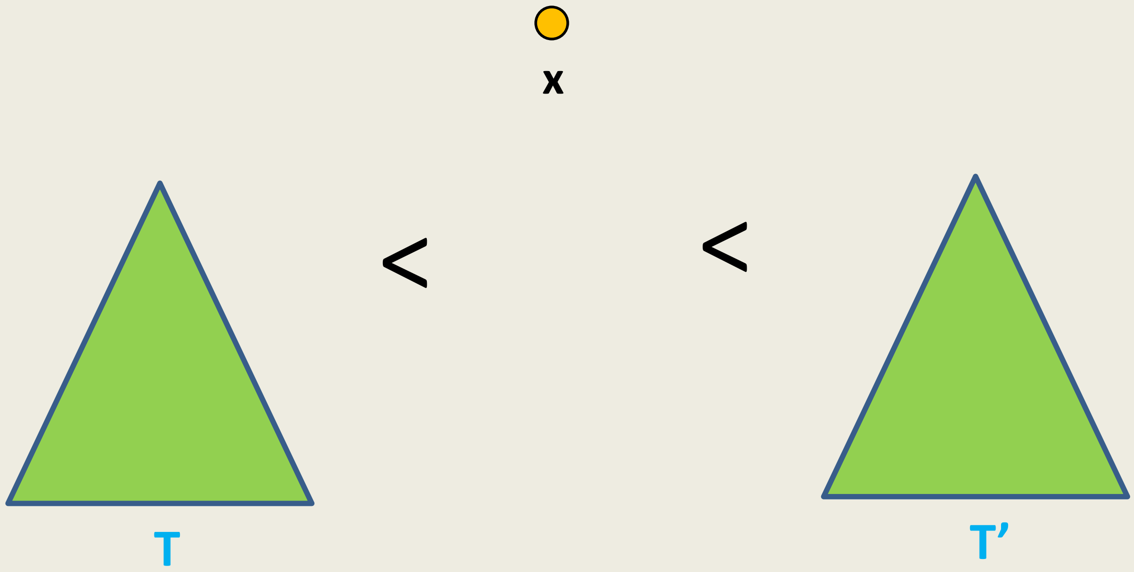
Time complexity: O(log n)

Towards an O(log n) time for SpecialUnion(T,T') …

Can we solve some simple cases easily ?

- **Simplifying the problem**
- **Solving the simpler version efficiently**
- **Extending the solution to generic version**

Simplifying the problem



Simplified problem:

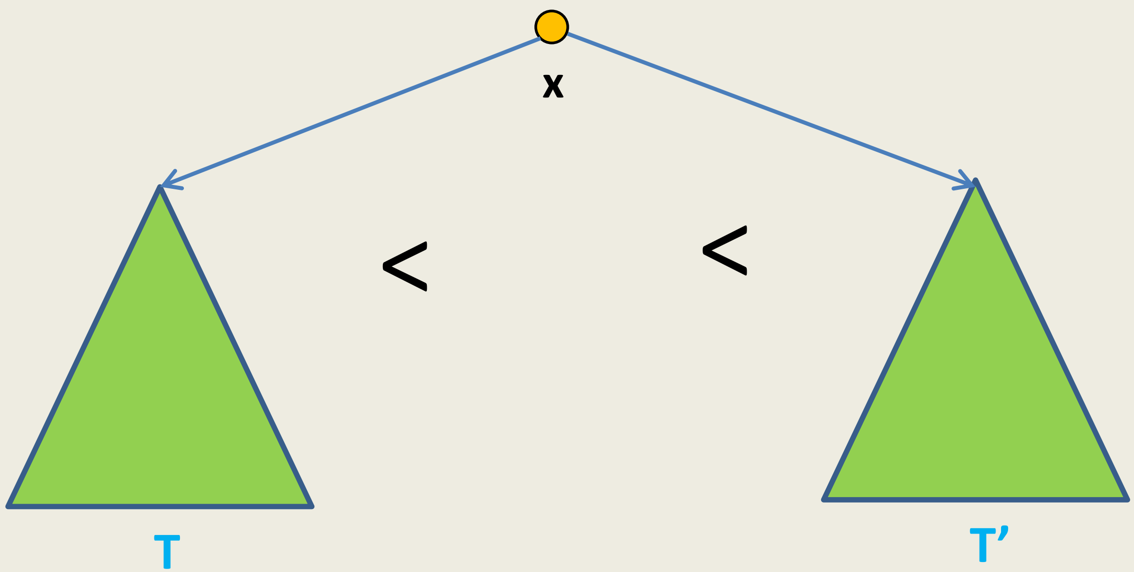
Given two trees T, T’ of <u>same</u> **black height**

and a key **x**, such that T<**x**<T’,

transform them into a tree T*=TU{**x**}UT’

Solving the simplified problem

O (1) time



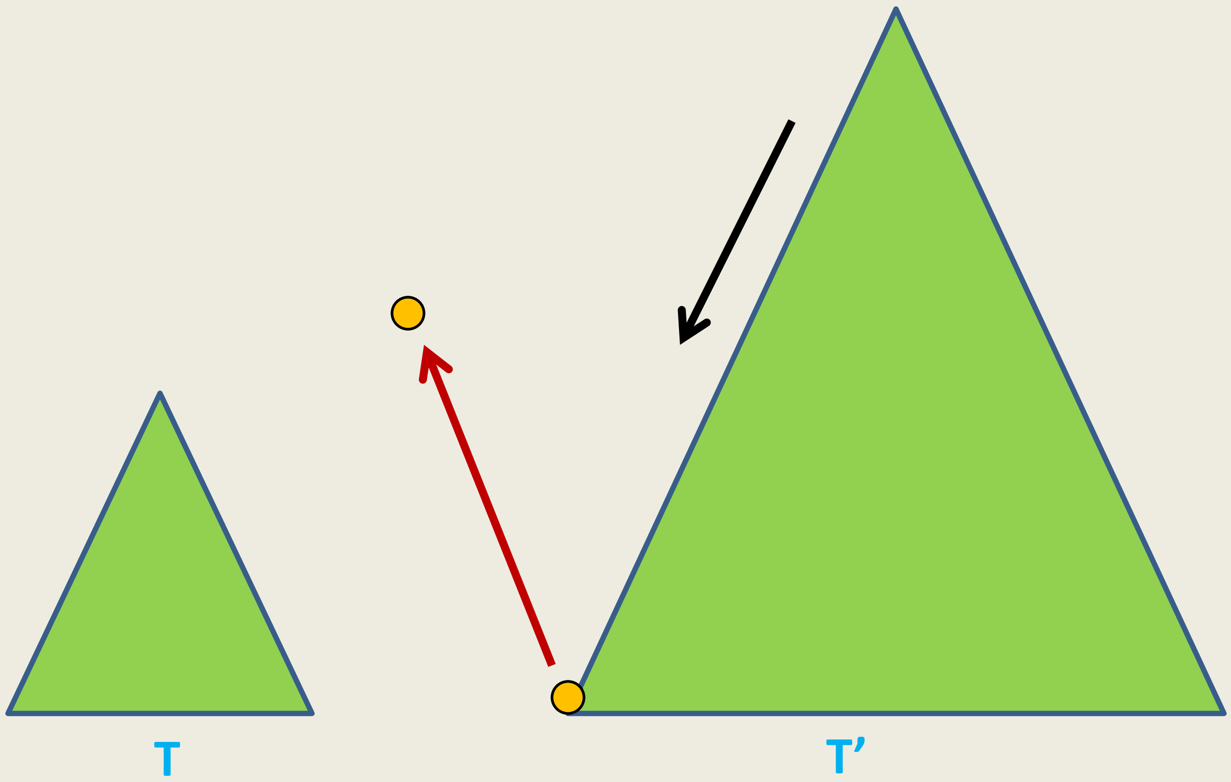
Simplified problem:

Given two trees **T**, **T’** of <u>same</u> **black height**

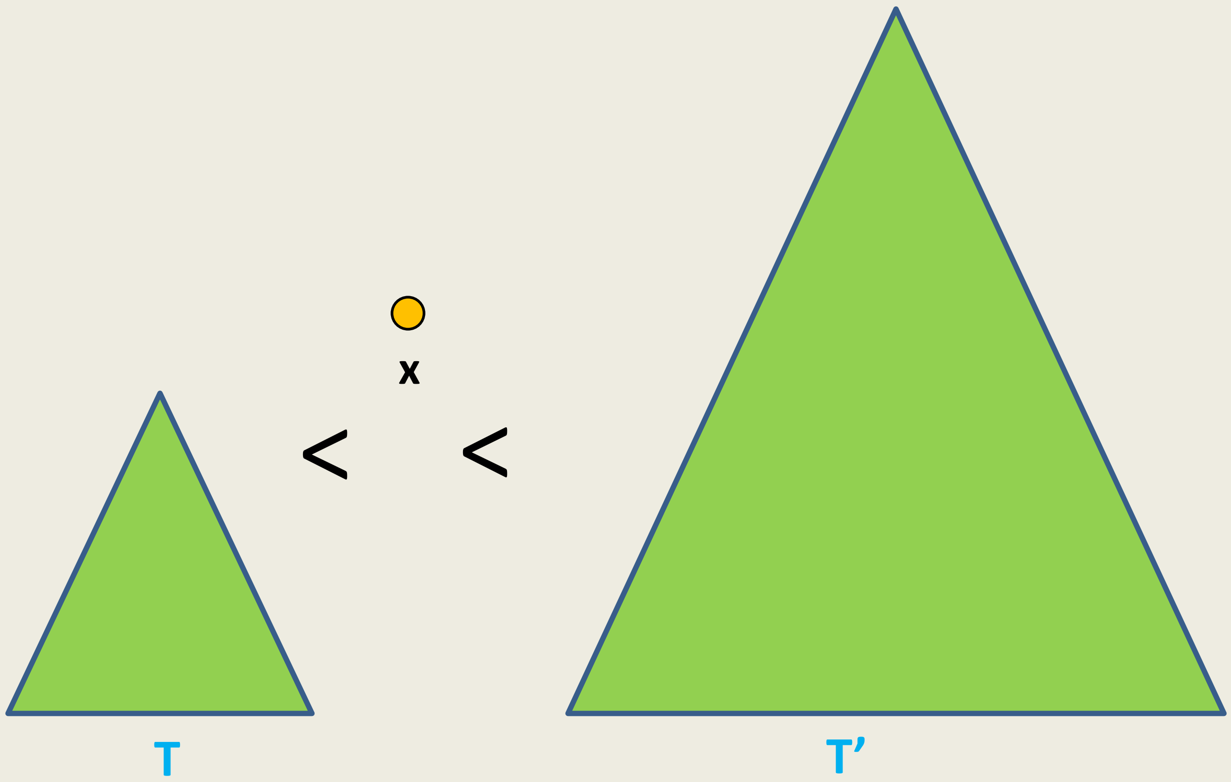
and a key **x**, such that **T**<**x**<**T’**,

transform them into a tree **T***=**T**U{**x**}U**T’**

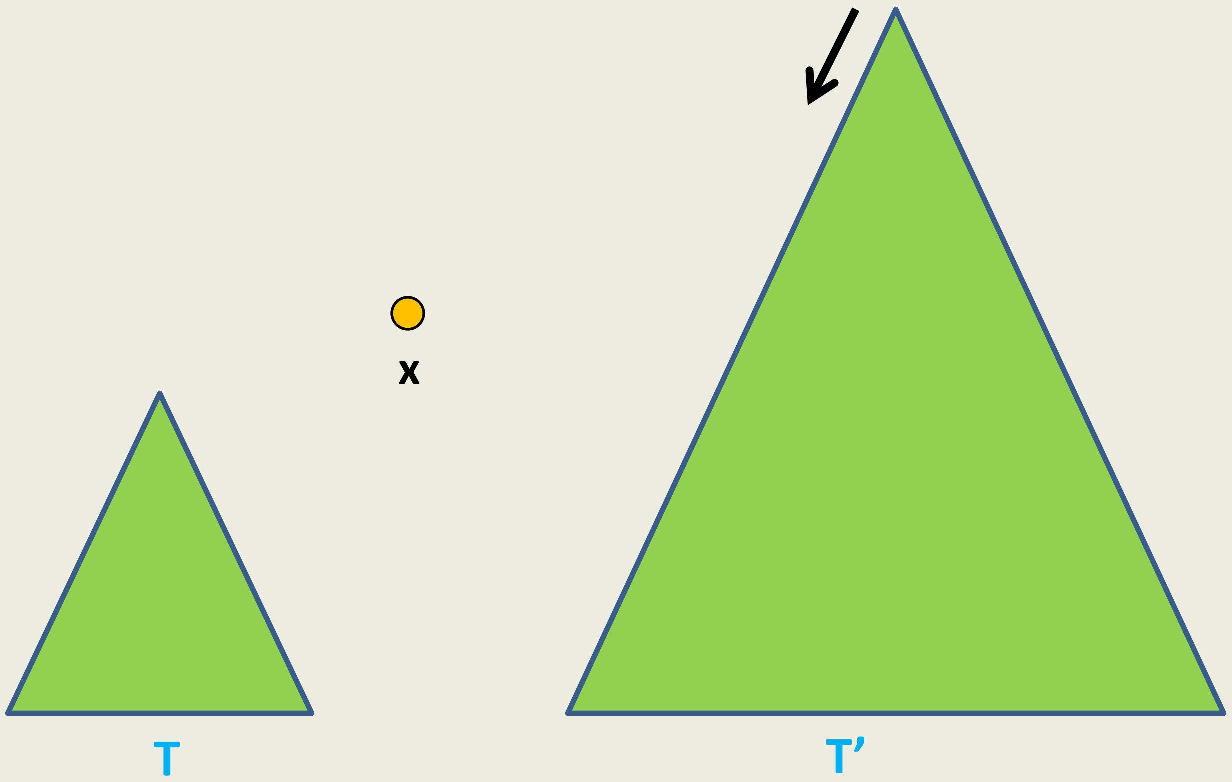
Extending the algorithm to the generic problem



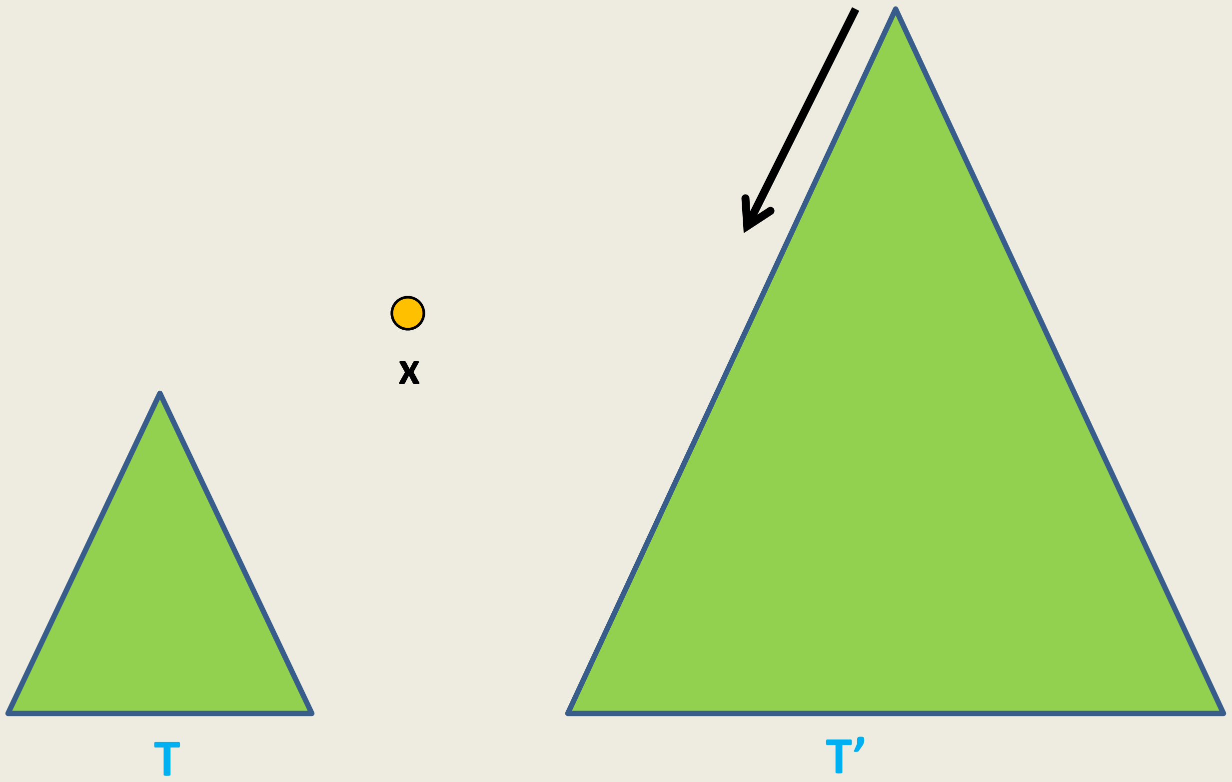
Extending the algorithm to the generic problem



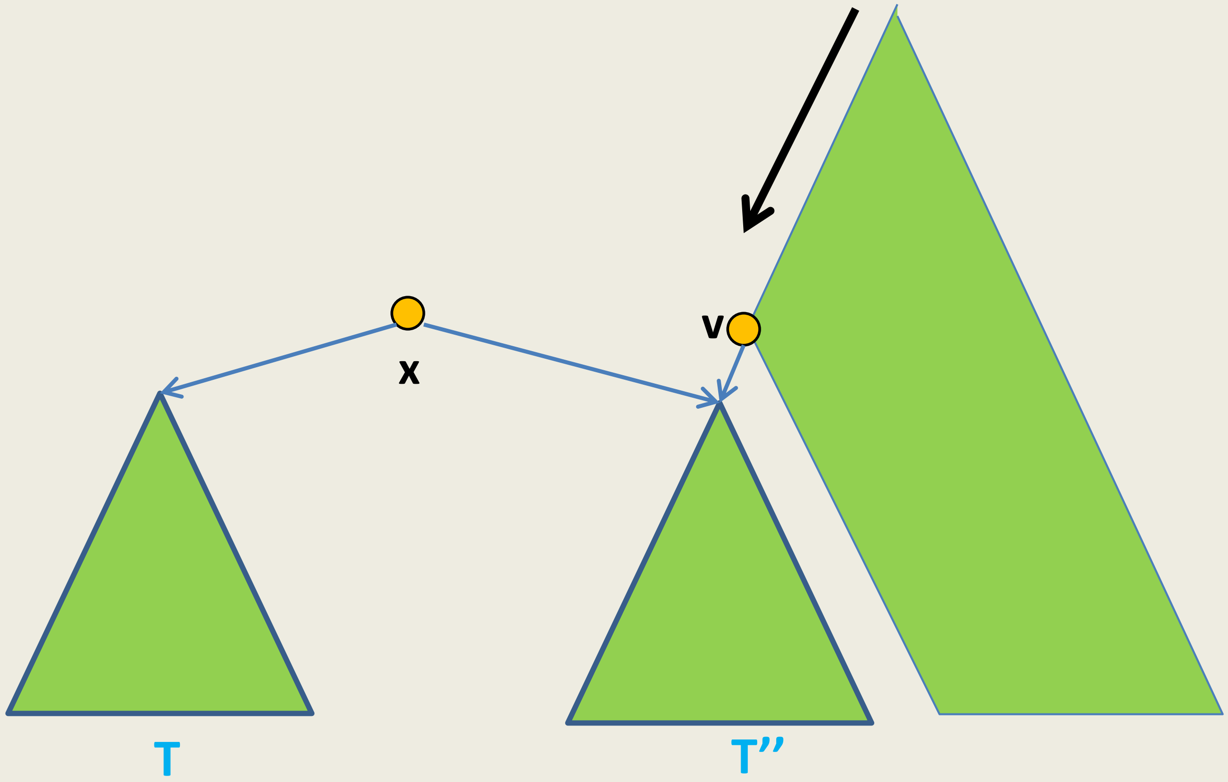
Extending the algorithm to the generic problem



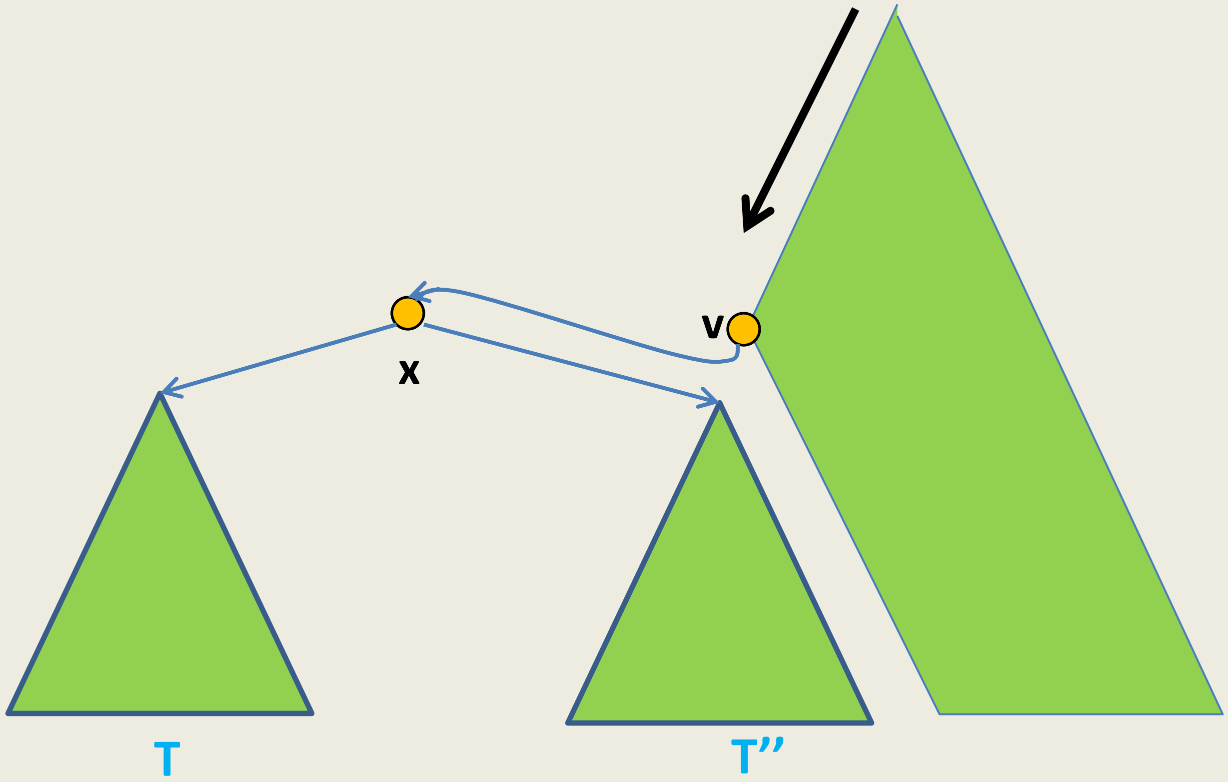
Extending the algorithm to the generic problem



Extending the algorithm to the generic problem



Extending the algorithm to the generic problem



Extending the algorithm to the generic problem

Algorithm for **SpecialUnion**(**T**,**T'**):

1. Let **x** be the node storing smallest element of **T'**.
2. **Delete** the node **x** from **T'.**

Let **black height** of **T** ≤ **black height** of **T'**

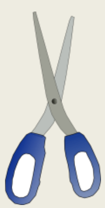
1. Keep following left pointer of **T'** until we reach a node **v** such that
   1. **left**(**v**) is black
   2. The subtree **T"** rooted at **Left**(**v**) has black height same as that of **T**
2. **left(x)← T;**
3. **right(x) ← T";**
4. **Color(x)← red;**
5. **left(v)← x;**
6. **parent(x)← v;**
7. If **color(v)** is **red,** remove the color imbalance

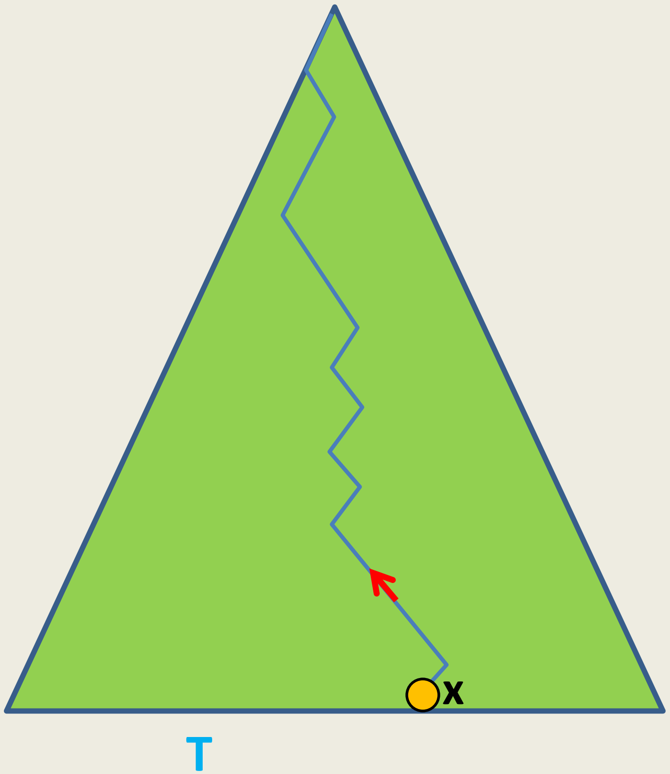
   (like in the usual procedure of insertion in a **red-black** tree)

Total time : O(log n)

Split(T,x)

Achieving O(log n) time for Split(T,x)





- Take a scissor
- cut **T** into trees starting from **x**
- Make use of **SpecialUnion** algorithm.