

Data Structures and Algorithms

(ESO207)

Lecture 11:

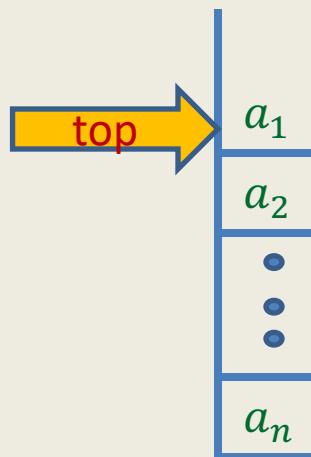
- **Arithmetic expression evaluation: Complete algorithm using stack**
- **Two interesting problems**

Quick Recap of last lecture

Stack: a new data structure

A special kind of list

where all operations (insertion, deletion, query) take place at one end only, called the **top**.



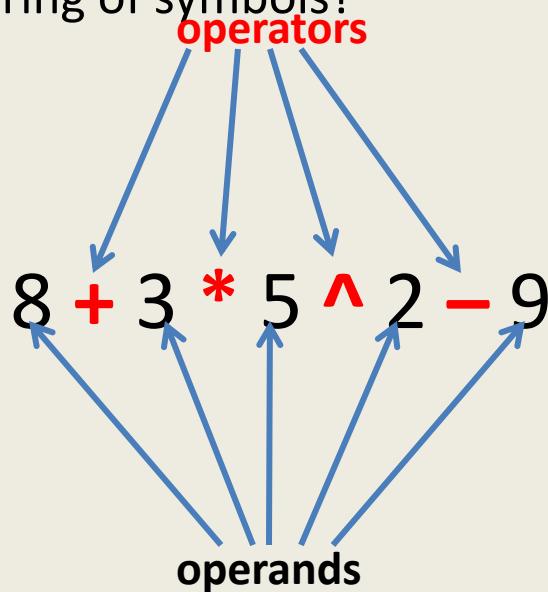
Evaluation of an arithmetic expression

Question: How does a computer/calculator evaluate an arithmetic expression given in the form of a string of symbols ?

$$8 + 3 * 5 ^ 2 - 9$$

Evaluation of an arithmetic expression

Question: How does a computer/calculator evaluate an arithmetic expression given in the form of a string of symbols?



- What about expressions involving **parentheses**: $3+4*(5-6/(8+9^2)+33)$?
- What about **associativity** of the operators ?

Overview of our solution

- 1. Focusing on a simpler version of the problem:**
 1. Expressions without parentheses
 2. Every operator is left associative
- 2. Solving the simpler version**
- 3. Transforming the solution of simpler version to generic**

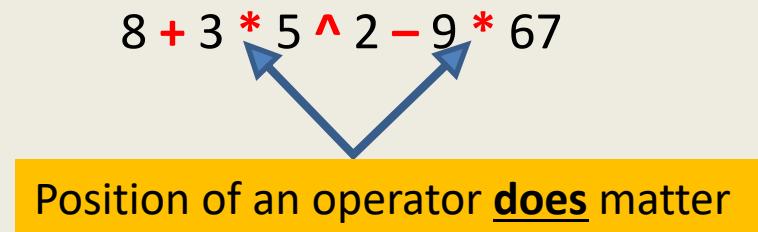
Incorporating precedence of operators through priority number

Operator	Priority
+ , -	1
* , /	2
^	3

Insight into the problem

Let o_i : the operator at position i in the expression.

Aim: To determine an order in which to execute the operators.



Question: Under what conditions can we execute operator o_i immediately?

Answer: if

- $\text{priority}(o_i) > \text{priority}(o_{i-1})$
- $\text{priority}(o_i) \geq \text{priority}(o_{i+1})$

Expression: $n_1 o_1 n_2 o_2 n_3 o_3 n_4 o_4 n_5 o_5 n_6 o_6 \dots$



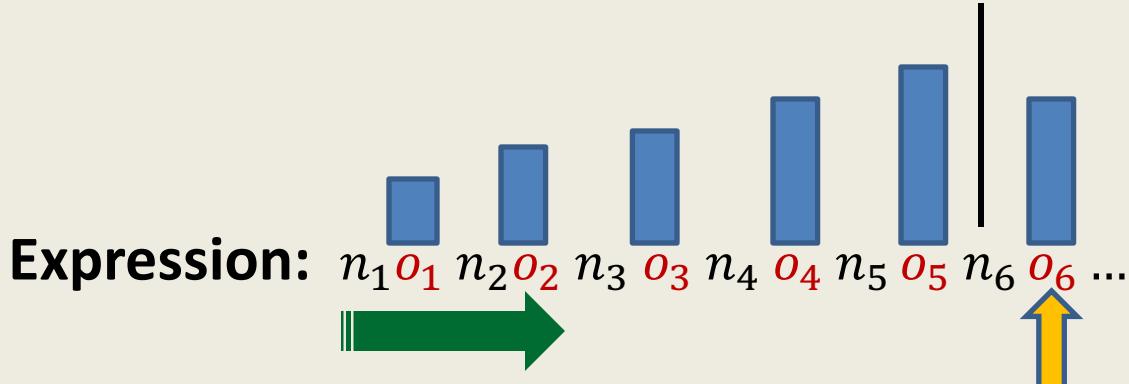
We keep two stacks:



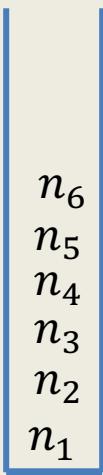
N-stack
for operands



O-stack
for operators



We keep two stacks:



N-stack
for operands

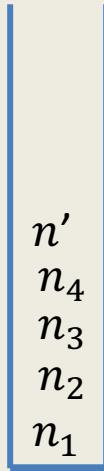


O-stack
for operators

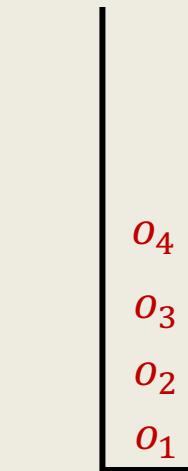
Expression: $n_1 o_1 n_2 o_2 n_3 o_3 n_4 o_4 \dots$



We keep two stacks:

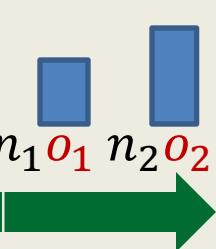


N-stack
for **operands**

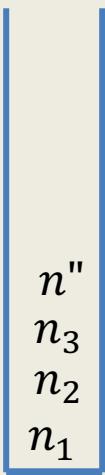


O-stack
for **operators**

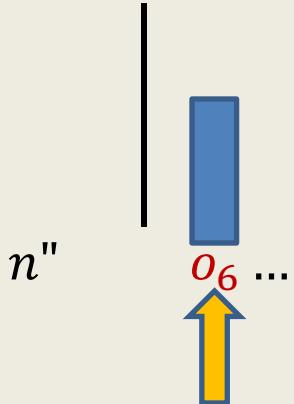
Expression: $n_1 o_1 n_2 o_2 n_3 o_3$



We keep two stacks:



N-stack
for operands



O-stack
for operators

A simple algorithm

```
push($,O-stack);
```

Priority of \$:

Least

```
While ( ? ) do
```

```
{   x ← next_token();
```

Two cases:

x is number : push(x,N-stack);

x is operator :

```
while( PRIORITY(TOP(O-stack)) >= PRIORITY(x) )
```

```
{   o ← POP(O-stack);  
    Execute(o);  
}
```

- POP two numbers from N-stack
- apply operator o on them
- place the result back into N-stack

```
push(x,O-stack);
```

```
}
```

Next step

**Transforming the solution to Solve
the most general case**

How to handle parentheses ?

$$3+4*(5 - 6/2)$$

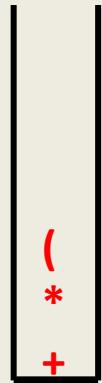
What should we do whenever
we encounter (in the expression ?

Answer:

Evaluate the expression enclosed by this parenthesis
before any other operator currently present in the O-stack.

→ So we must push (into the O-stack.

Observation 1: While (is the **current operator** encountered in the expression,
it must have higher priority than every other operator in the stack.



O-stack

How to handle parentheses ?

$$3+4*(5 - 6/2)$$

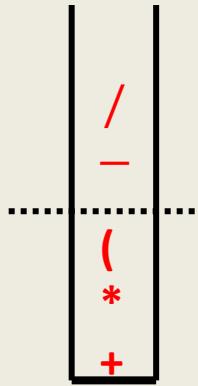
What needs to be done when
(is at the top of the O-stack ?

Answer:



The (should act as an *artificial bottom* of the O-stack .

→ every other operator that follows (should be allowed to sit on the top of (in the stack .



Observation 2 : while (is inside the stack,

it must have less priority than every other operator that follows.

A CONTRADICTION !!

Observation 1: While (is the current operator encountered in the expression,

it must have higher priority than every other operator in the stack

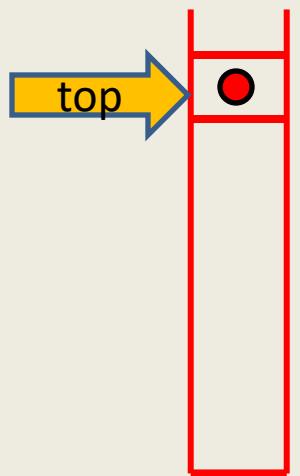
Take a pause for a few minutes to realize **surprisingly that
the **contradicting** requirements for the priority of (**
in fact hints at a **suitable solution for handling (.**

How to handle parentheses ?

Using two **types** of priorities of each operator ●.

InsideStack priority

The priority of an operator ●
when it is **inside** the stack.



O-stack

OutsideStack priority

The priority of an operator ●
when it is **encountered** in the expression.

$n_1 o_1 n_2 o_2 n_3 o_3 n_4 \bullet n_5 o_5 n_6 o_6$



O-stack

How to handle parentheses ?

Using two **types** of priorities of each operator ●.

Operator	<u>InsideStackPriority</u>	<u>OutsideStackPriority</u>
+ , -	1	1
* , /	2	2
^	3	3
(0	4

Does it take care of nested parentheses ? Check it yourself.

How to handle parentheses ?

$$\boxed{3+4^*(5 - 6/2)}$$

Question: What needs to be done whenever we encounter) in the expression ?

Answer: Keep **popping O-stack** and evaluating the operators until we get its matching (.

The algorithm generalized to handle parentheses

```
push($,O-stack);
```

```
While ( ? ) do
```

```
    x ← next_token();
```

Cases:

```
    x is number : push(x,N-stack);
```

```
    x is ) : while( TOP(O-stack) <> ( )
```

```
        { o ← Pop(O-stack);
```

```
        Execute(o);
```

```
    }
```

```
    Pop(O-stack); //popping the matching (
```

```
otherwise : while( InsideStackPriority(TOP(O-stack)) >= OutsideStackPriority(x) )
```

```
    { o ← Pop(O-stack);
```

```
    Execute(o);
```

```
    }
```

```
Push(x,O-stack);
```

Practice exercise

Execute the algorithm on $3+4*((5+6*(3+4)))^2$ and convince yourself through proper reasoning that the algorithm handles parentheses suitably.

How to handle associativity of operators ?

Associativity of arithmetic operators

Left associative operators : + , - , * , /

- $a+b+c = (a+b)+c$
- $a-b-c = (a-b)-c$
- $a*b*c = (a*b)*c$
- $a/b/c = (a/b)/c$

We have already handled left associativity in our algorithm.

Right associative operators: ^

- $2^3^2 = 2^3(3^2) = 512.$

How to handle right associativity ?

What we need is the following:

If $\textcolor{green}{\wedge}$ is **current operator** of the expression, and $\textcolor{orange}{\wedge}$ is on **top of stack**,
then $\textcolor{green}{\wedge}$ should be evaluated before $\textcolor{orange}{\wedge}$.

How to incorporate it ? Play with the **priorities** 😊

How to handle associativity of operators ?

Using two **types** of priorities of each **right associative** operator.

Operator	<u>InsideStackPriority</u>	<u>Outside-stack priority</u>
+ , -	1	1
* , /	2	2
^	3	4
(0	5

The **general** Algorithm

It is the same as the algorithm to handle parentheses :-)

While (?) do

 x \leftarrow next_token();

Cases:

 x is **number** : **push(x,N-stack);**

 x is) : **while(TOP(O-stack) <> ()**

 { o \leftarrow **Pop(O-stack);**

Execute(o);

 }

Pop(O-stack); //popping the matching (

 otherwise : **while(InsideStackPriority(TOP(O-stack)) >= OutsideStackPriority(x))**

 { o \leftarrow **Pop(O-stack);**

Execute(o);

 }

Push(x,O-stack);

Homeworks

1. Execute the general algorithm on $3+4*((4+6)^2)/2$ and convince yourself through proper reasoning that the algorithm handles nested parentheses suitably.
2. Execute the general algorithm on $3+4^2^2*3$ and convince yourself through proper reasoning that the algorithm takes into account the right associativity of operator $^$.
3. What should be the priorities of $\$$?
4. How to take care of the end of the expression ?
Hint: Introduce a new operator symbol $\#$ at the end of the expression so that upon seeing $\#$, we do very much like what we do on seeing $)$. What should be the priorities of $\#$?

Homeworks

- How is recursion implemented during program execution ?

Using stack

```
int Recur(int i)
{
    int j, k, val;
    ...
    ...
    val = Recur(t);
    ...
    ...
}
```

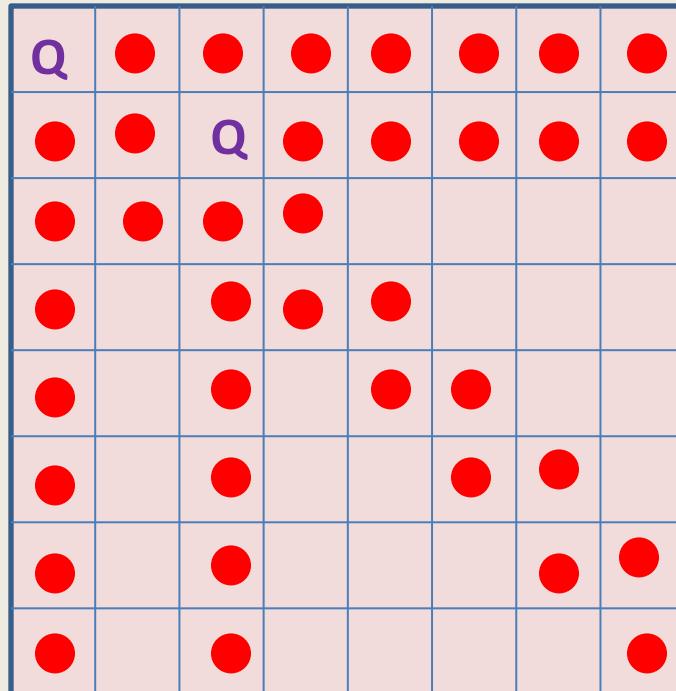
Learn about it from [wikipedia](#) ...

Two interesting problems

**Applications of simple data
structures**

8 queen problem

Place 8 queens on a chess board so that no two of them attack each other.

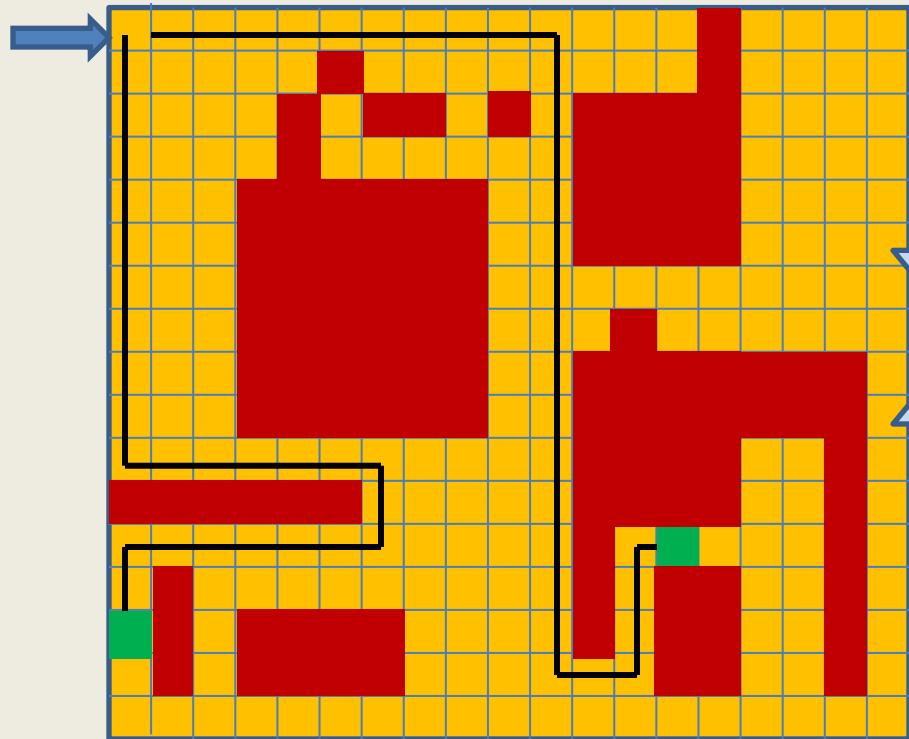


With this sketch/hint,
try to design the
complete algorithm
using stack or
otherwise.

Shortest route in a grid

From a cell in the grid, we can move to any of its neighboring cell in one step.

From top left corner, **find shortest route** to each green cell avoiding obstacles.



Ponder over this
beautiful problem
😊

Data Structures and Algorithms

(ESO207)

Lecture 12:

- **Queue** : a new data Structure :
- Finding shortest route in a grid in presence of obstacles

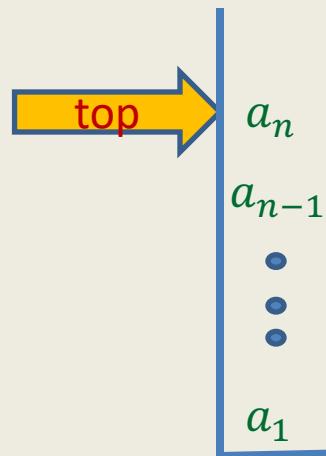
Queue: a new data structure

Data Structure Queue:

- **Mathematical Modeling of Queue**
- **Implementation of Queue using arrays**

Stack

A special kind of list where all operations (insertion, deletion, query) take place at one end only, called the **top**.



Behavior of Stack:

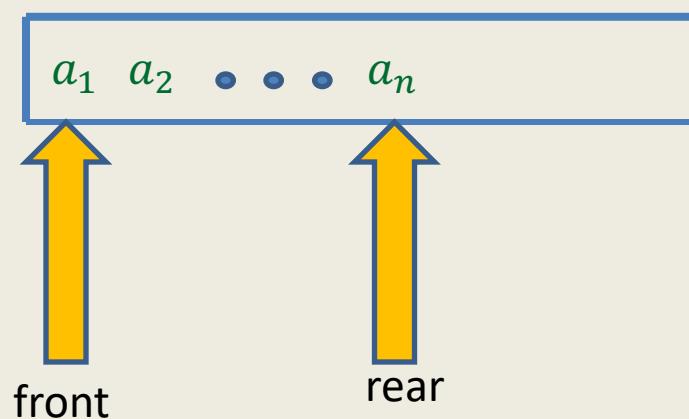
Last in (LIFO)

First out

Queue: a new data structure

A special kind of list based on **(FIFO)**

First in First Out



Operations on a Queue

Query Operations

- **IsEmpty(Q)**: determine if **Q** is an empty queue.
- **Front(Q)**: returns the element at the **front** position of the queue.

Example: If **Q** is a_1, a_2, \dots, a_n , then **Front(Q)** returns a_1 .

Update Operations

- **CreateEmptyQueue(Q)**: Create an empty queue
- **Enqueue(x,Q)**: insert **x** at the **end** of the queue **Q**

Example: If **Q** is a_1, a_2, \dots, a_n , then after **Enqueue(x,Q)**, queue **Q** becomes

a_1, a_2, \dots, a_n, x

- **Dequeue(Q)**: return element from the **front** of the queue **Q** and delete it

Example: If **Q** is a_1, a_2, \dots, a_n , then after **Dequeue(Q)**, queue **Q** becomes

a_2, \dots, a_n

How to access i th element from the front ?

$a_1 \bullet \bullet \bullet a_{i-1} a_i \bullet \bullet \bullet a_n$

- To access i th element, we **must** perform **dequeue** (hence delete) the first $i - 1$ elements from the queue.

An Important point you must remember for every data structure

You can define any **new** operation only in terms of the primitive operations of the data structures defined during its modeling.

Implementation of Queue using array

Assumption: At any moment of time, the number of elements in queue is n .

Keep an array of Q size n , and two variables `front` and `rear`.

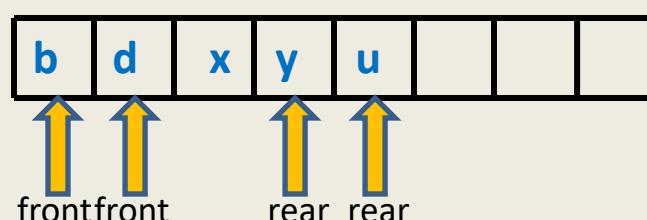
- `front`: the position of the **first** element of the queue in the array.
- `rear`: the position of the **last** element of the queue in the array.

Enqueue(x, Q)

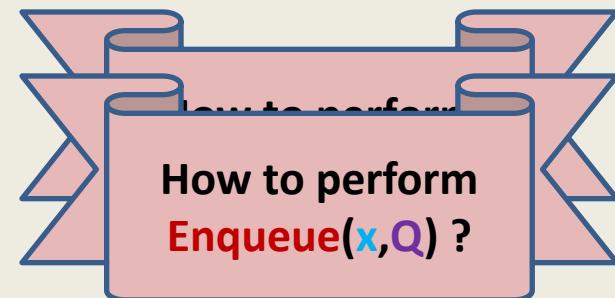
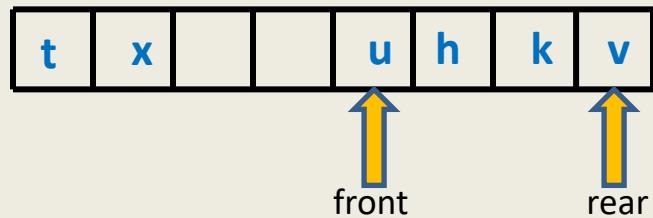
```
{    rear ← rear+1;  
    Q[rear]←x  
}
```

Dequeue(Q)

```
{    x← Q[front];  
    front← front+1;  
    return x;}
```



Implementation of Queue using array



Implementation of Queue using array

Enqueue(x,Q)

```
{    rear ← (rear+1) mod n ;  
    Q[rear]←x  
}
```

Dequeue(Q)

```
{      x← Q[front];  
    front← (front+1) mod n ;  
    return x;  
}
```

IsEmpty(Q)

```
{   Do it as an exercise }
```

Shortest route in a grid with obstacles

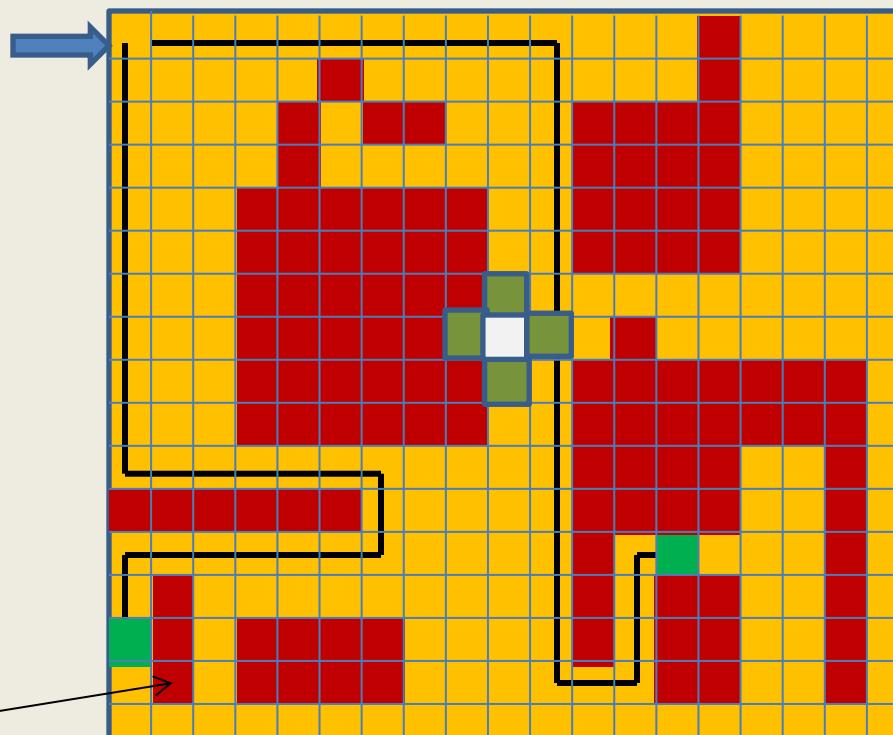
Shortest route in a grid

From a cell in the grid, we can move to any of its neighboring cell in one step.

Problem: From top left corner, find shortest route to each cell avoiding **obstacles**.

Input : a Boolean matrix G representing the grid such that

$G[i, j] = 0$ if (i, j) is an **obstacle**, and 1 otherwise.

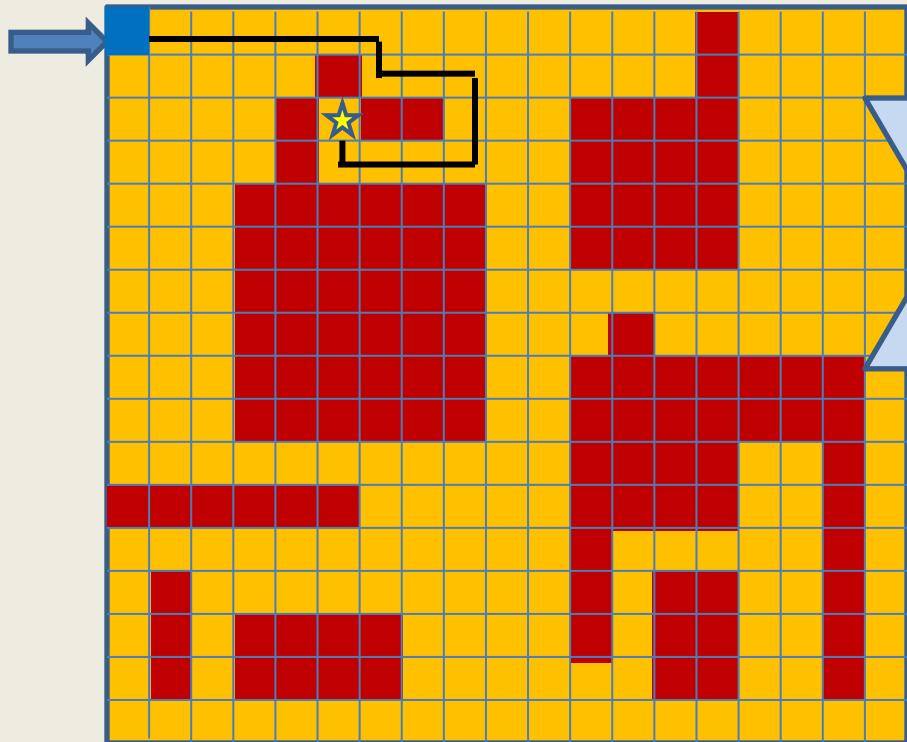


Step 1:

**Realizing
the nontriviality of the problem**

Shortest route in a grid

nontriviality of the problem



Don't proceed to the next slide until you are convinced about the non-triviality and beauty of this problem 😊

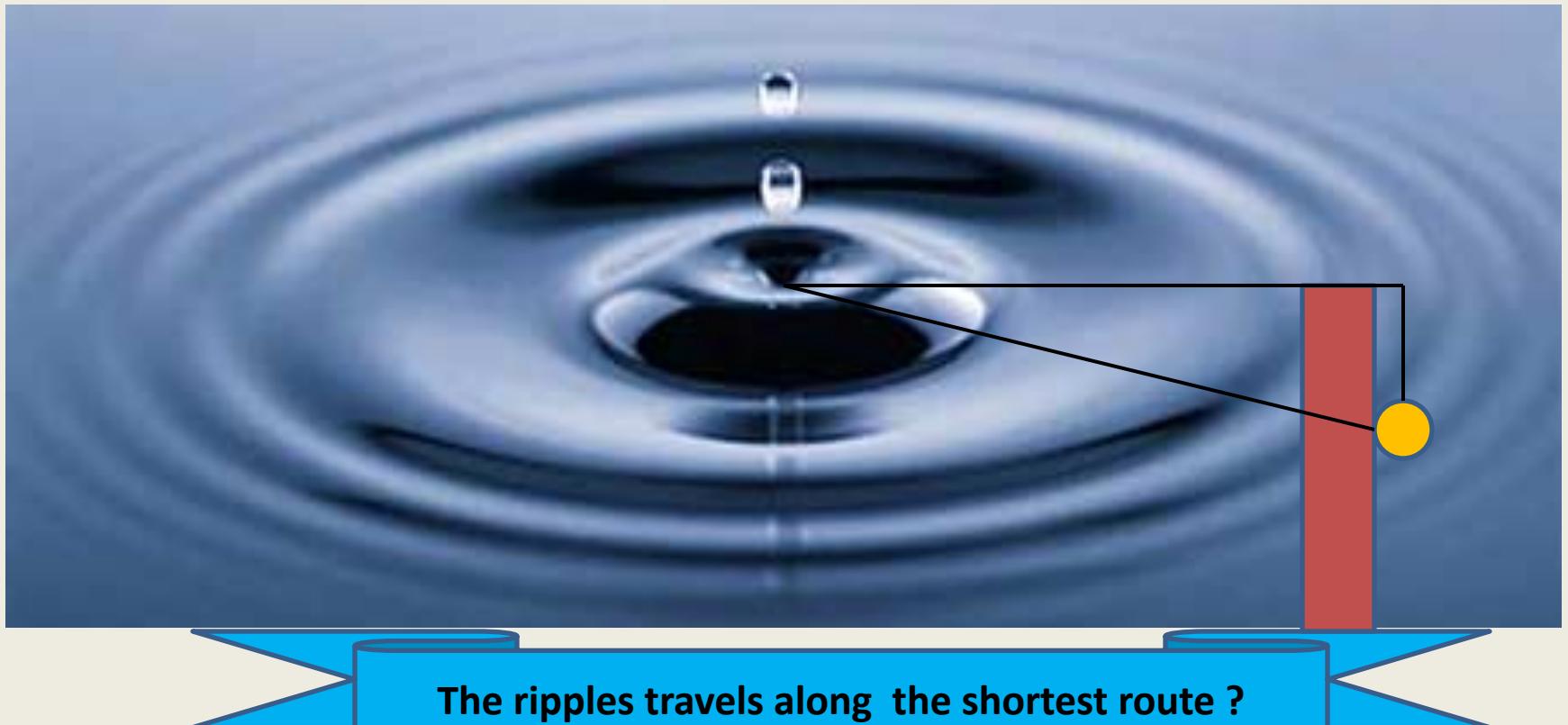
Definition: Distance of a cell c from another cell c'

is the length (number of steps) of the shortest route between c and c' .

We shall design algorithm for computing distance of each cell from the start-cell.

As an exercise, you should extend it to a data structure for retrieving shortest route.

Get **inspiration** from nature

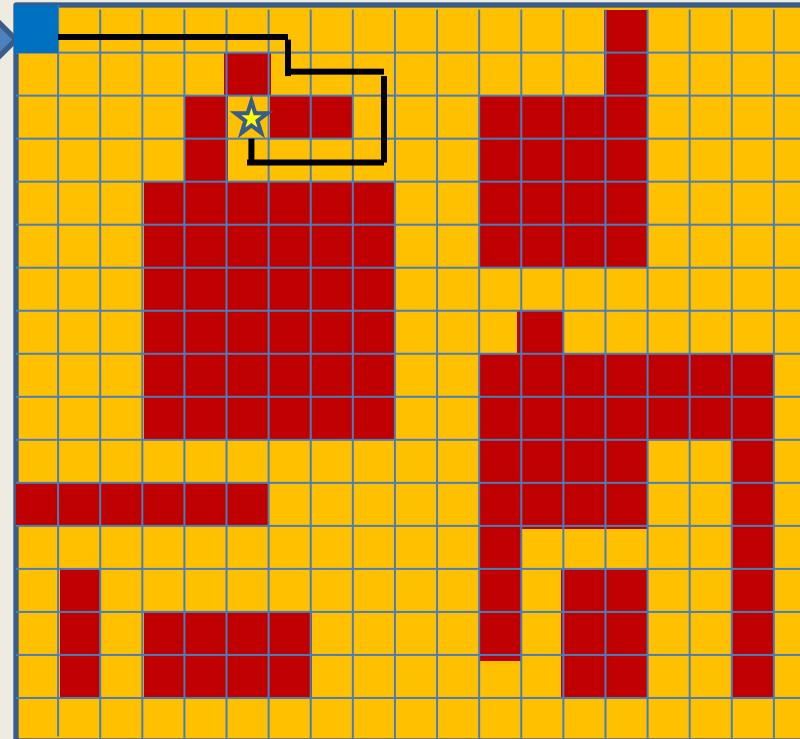


The ripples travels along the shortest route ?

Shortest route in a grid

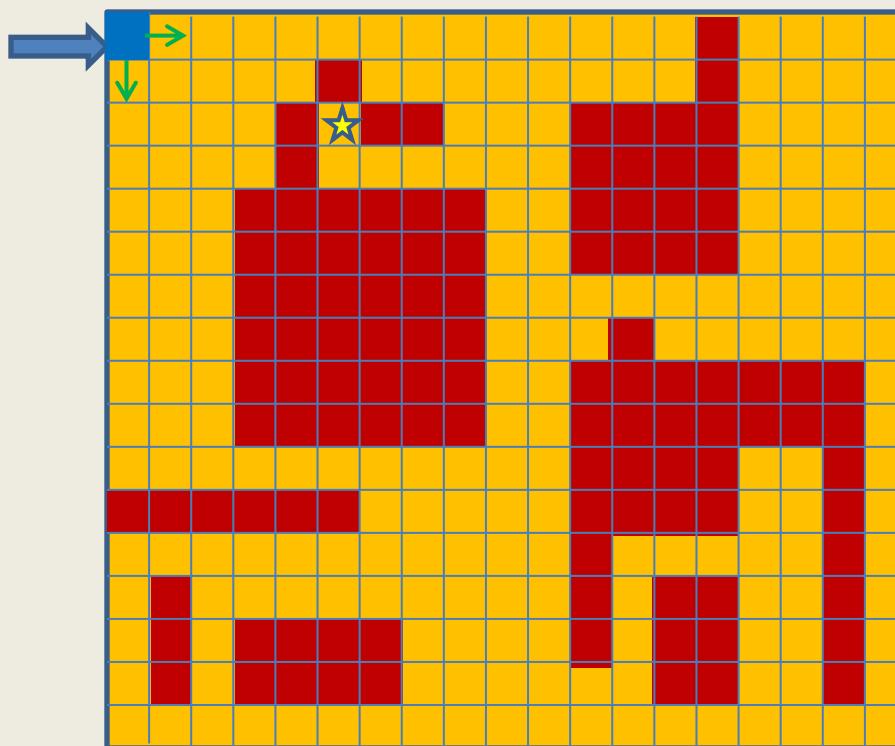
nontriviality of the problem

How to find the shortest route to  in the grid ?

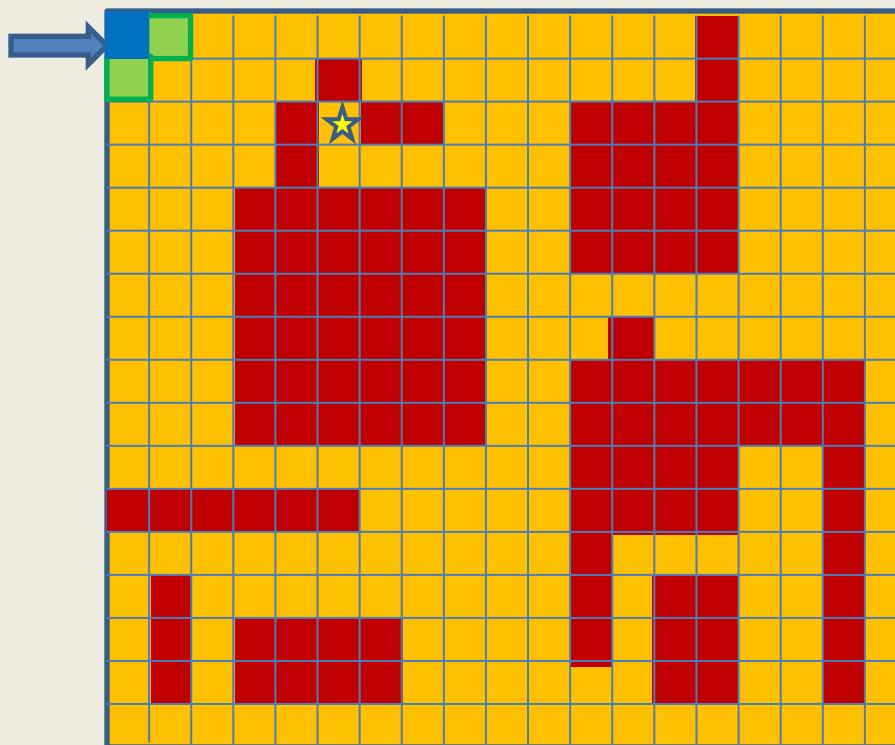


Create a ripple at the start cell and trace
the path it takes to 

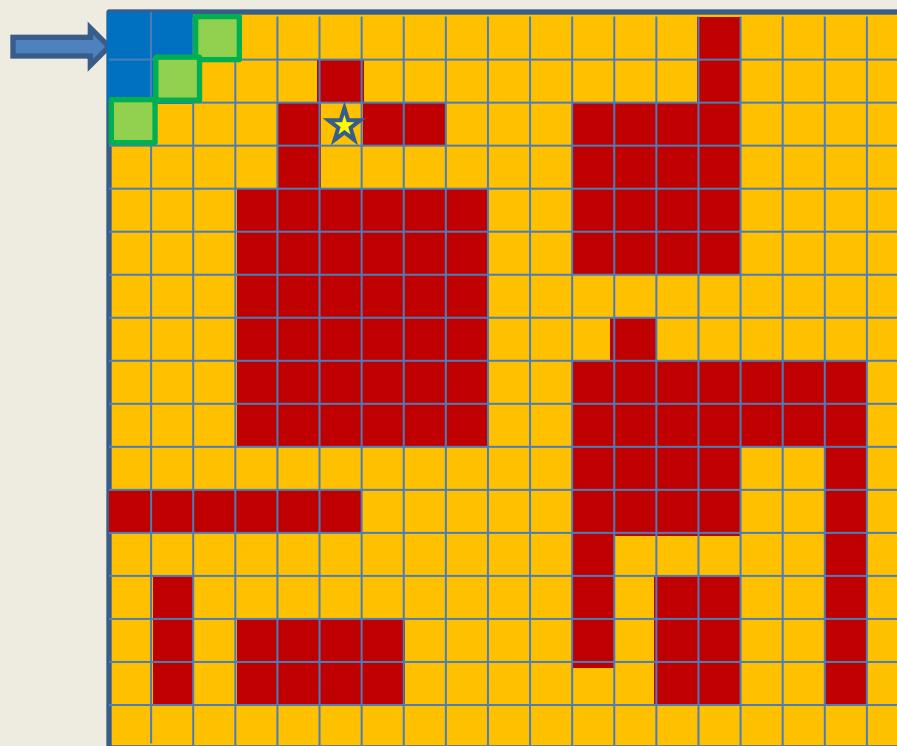
propagation of a ripple from **the start cell**



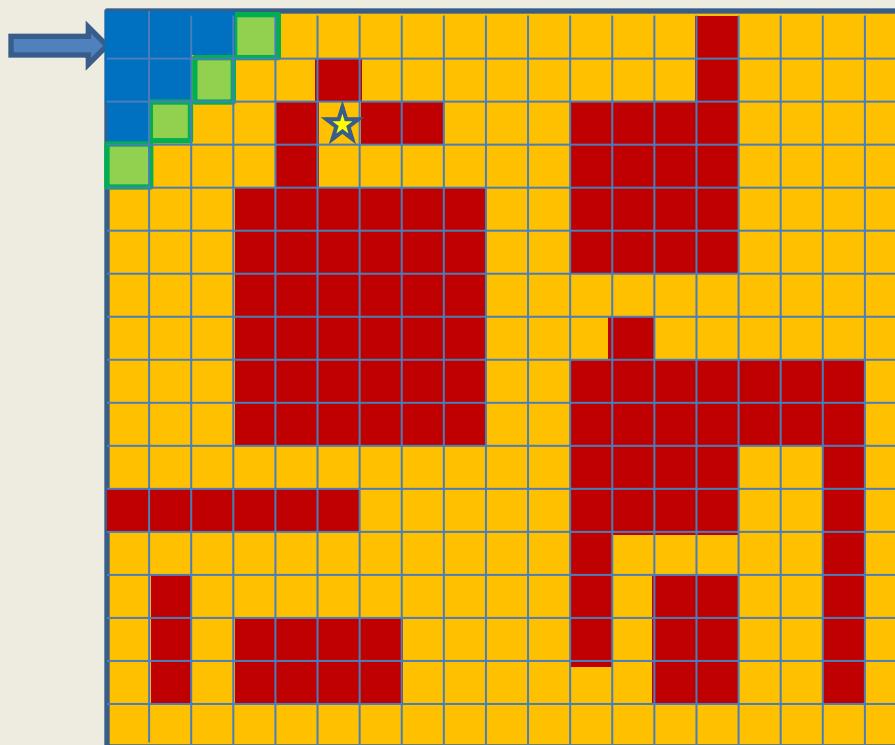
ripple reaches cells at distance 1 in step 1



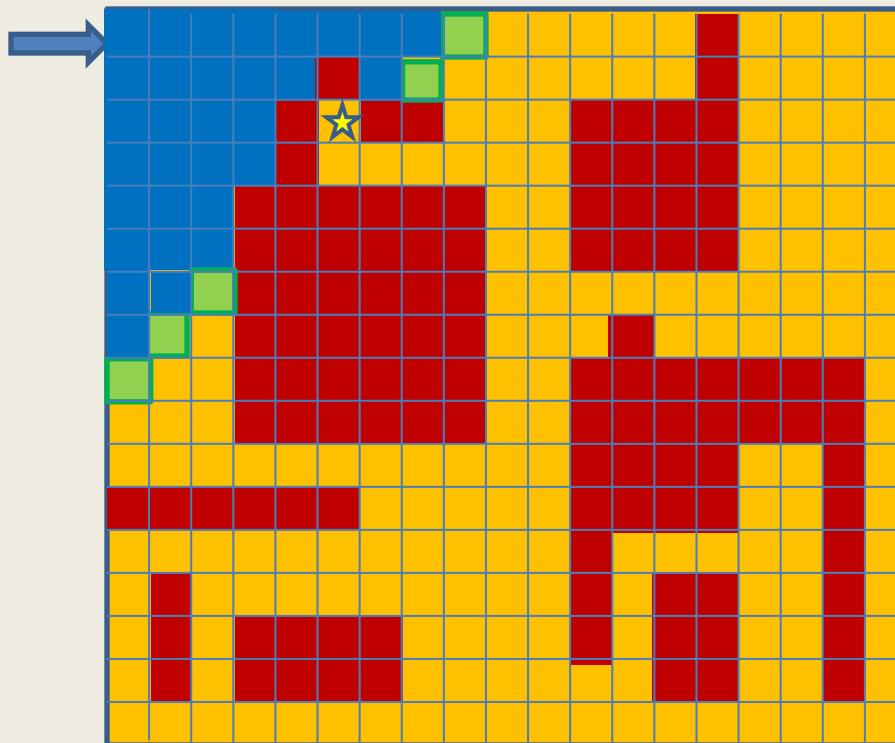
ripple reaches cells at distance 2 in step 2



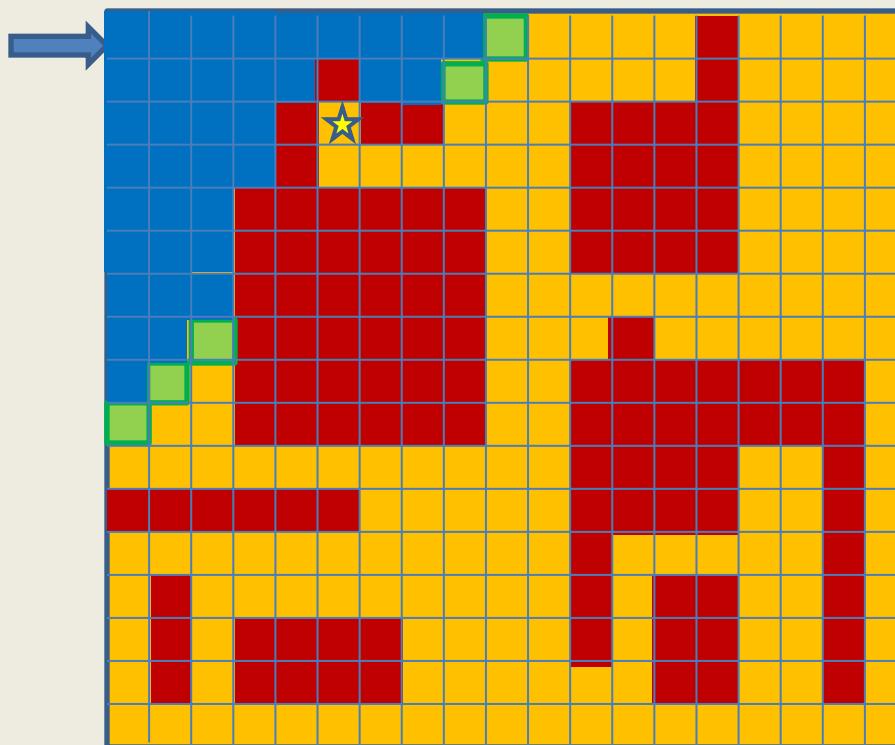
ripple reaches cells at distance 3 in step 3



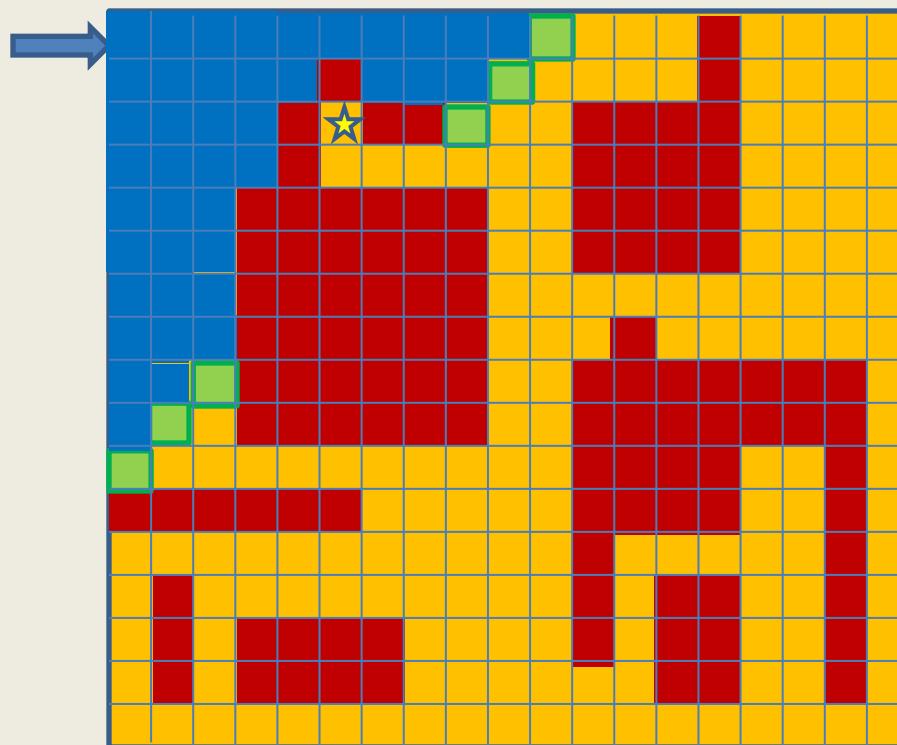
ripple reaches cells at distance 8 in step 8



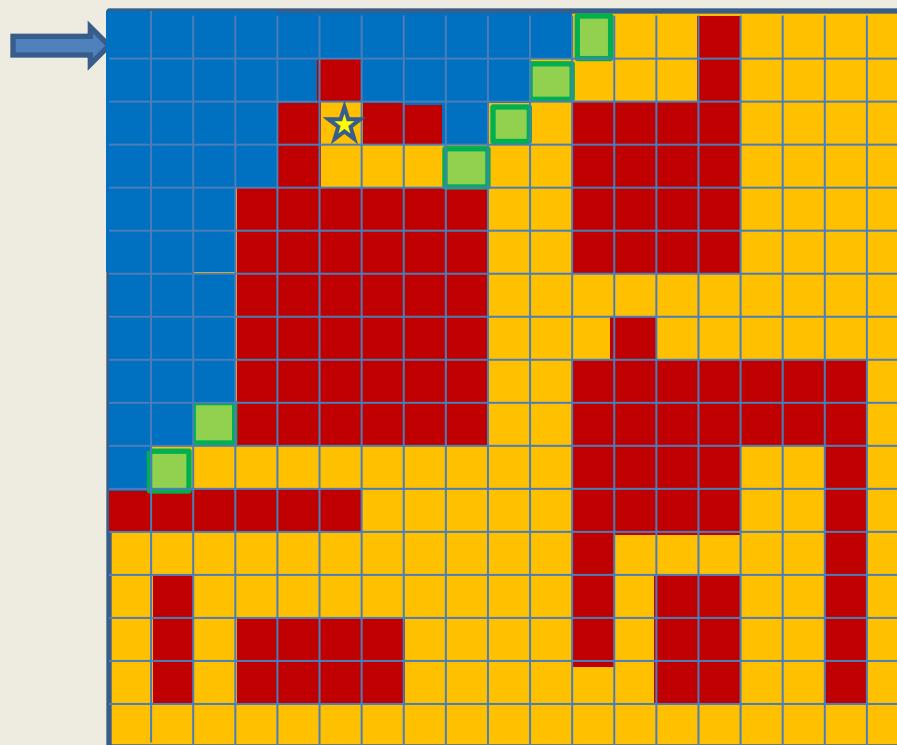
ripple reaches cells at distance 9 in step 9



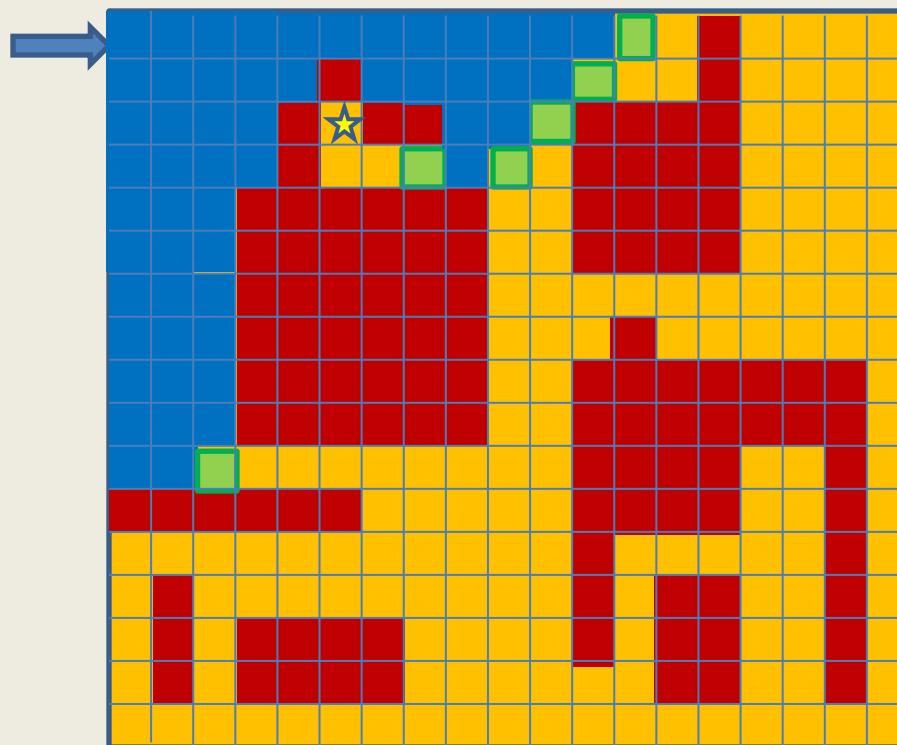
ripple reaches cells at **distance 10** in **step 10**



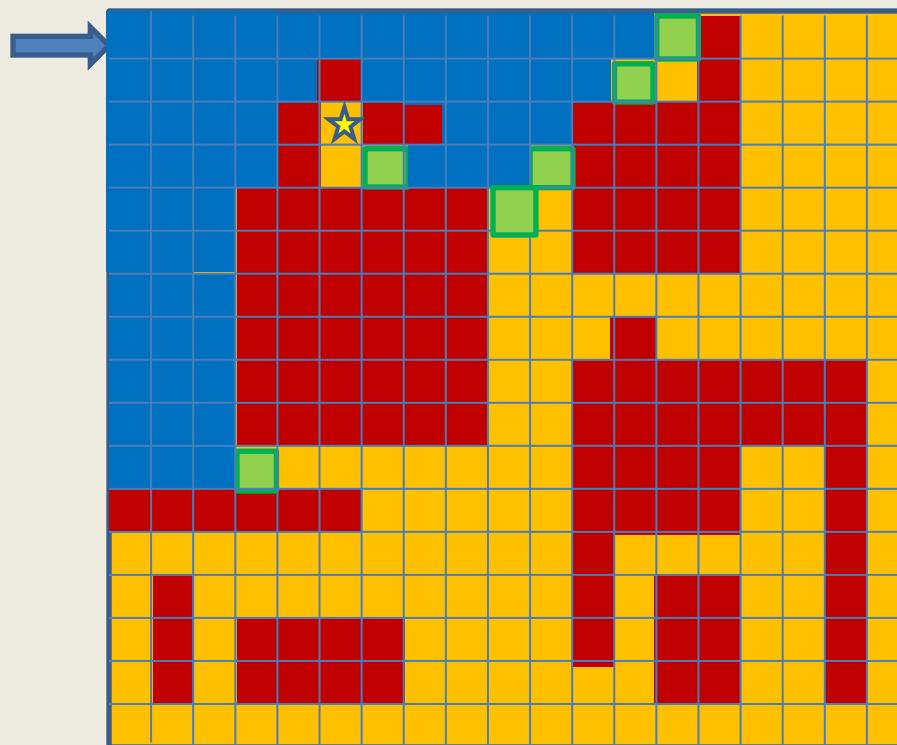
ripple reaches cells at **distance 11** in **step 11**



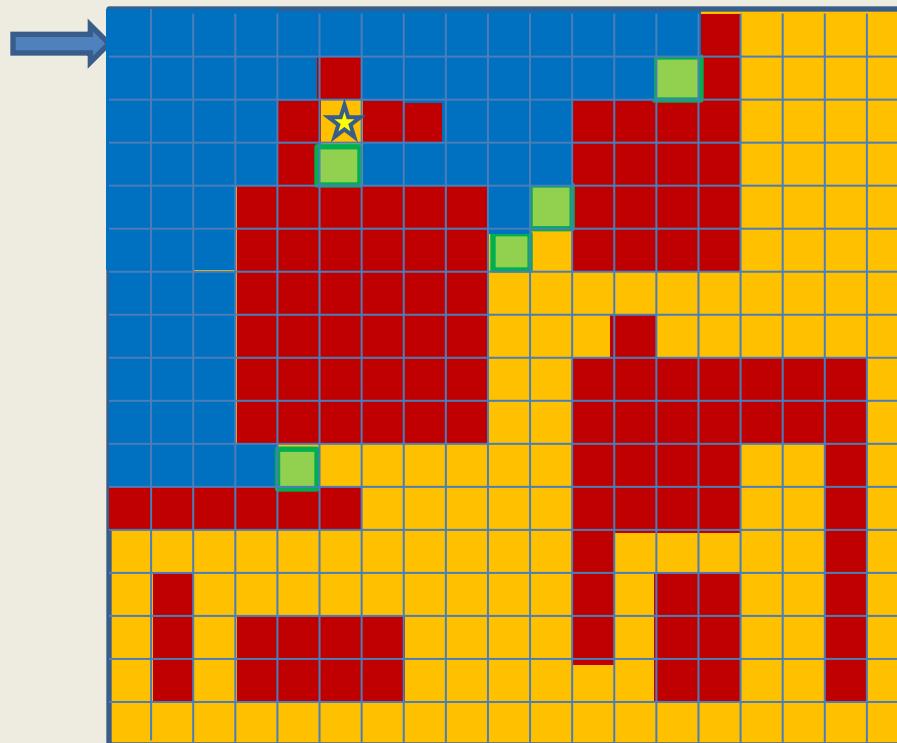
ripple reaches cells at **distance 12** in **step 12**



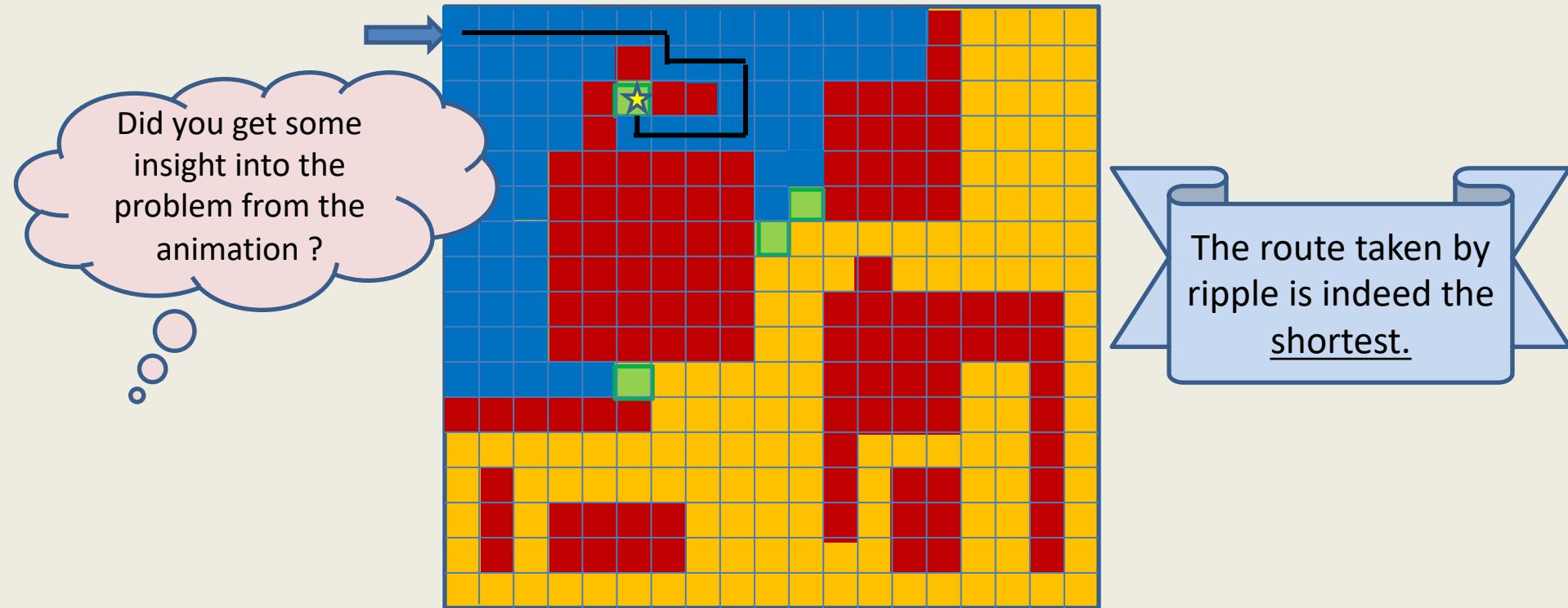
ripple reaches cells at **distance 13** in **step 13**



ripple reaches cells at **distance 14** in **step 14**



ripple reaches cells at **distance 15** in step 15



Think for a few more minutes with a free mind ☺.

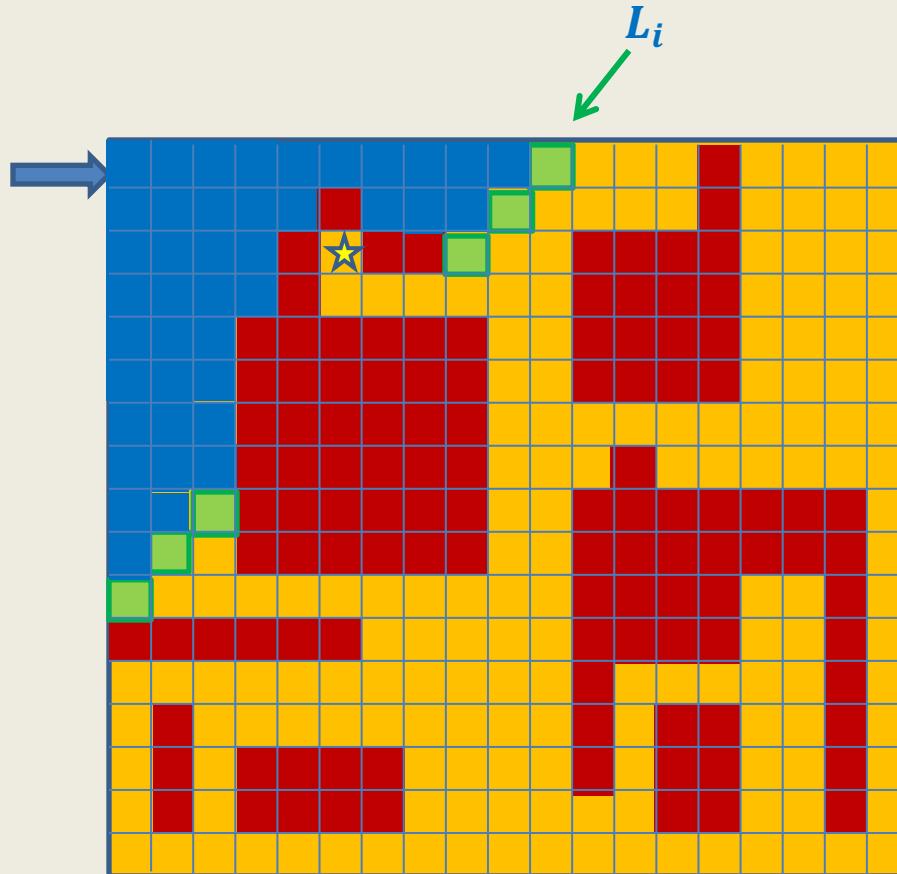
Step 2:

Designing algorithm for distances in grid

(using an insight into propagation of ripple)

A snapshot of ripple after i steps

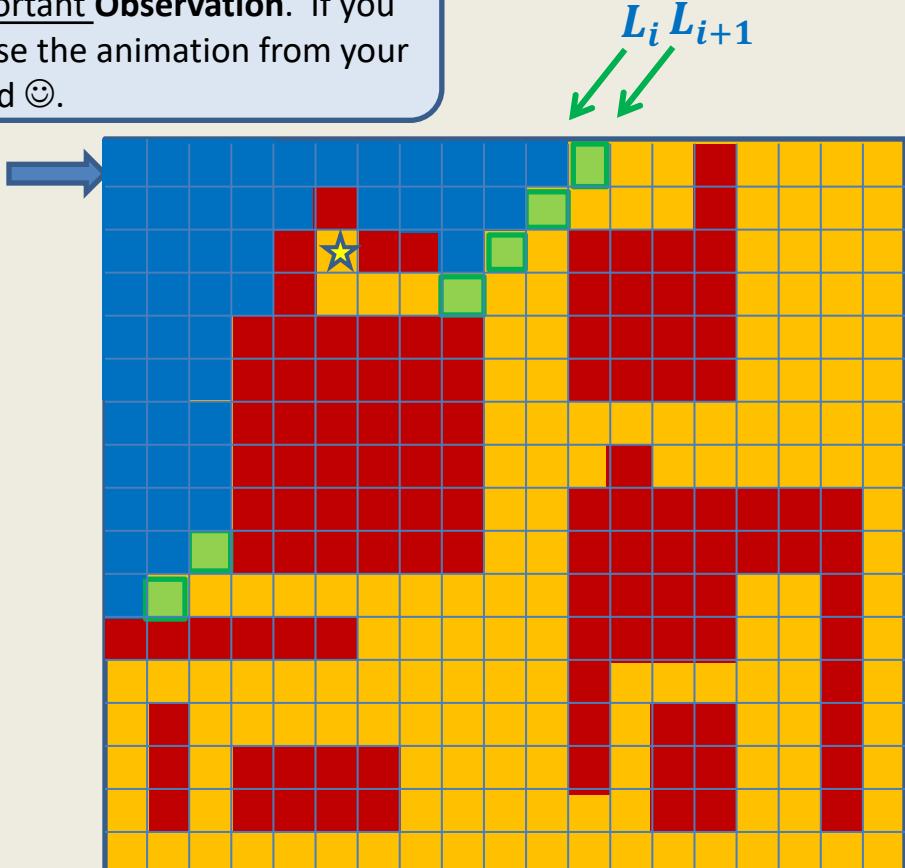
A snapshot of ripple after i steps



L_i : the cells of the grid at distance i from the starting cell.

A snapshot of the ripple after $i + 1$ steps

All the hardwork on the animation was done just to make you realize this important Observation. If you have got it, feel free to erase the animation from your mind ☺.



Observation: Each cell of L_{i+1} is a neighbor of a cell in L_i .

Distance from the start cell

It is worth spending some time on this matrix.
Does the matrix give some idea to answer the question ?

0	1	2	3	4	5	6	7	8	9	10	11	12	13	27	28	29	30		
1	2	3	4	5	6	7	8	9	10	11	12	13	14	26	27	28	29		
2	3	4	5	15	7	8	9	10	11	12				25	26	27	28		
3	4	5	6	14	13	12	11	12	13					24	25	26	27		
4	5	6				13	14							23	24	25	26		
5	6	7				14	15							22	23	24	25		
6	7	8				15	16	17	18	19	20			21	22	23	24		
7	8	9				16	17	18		20	21	22	23	24	25				
8	9	10				17	18								26				
9	10	11				18	19								27				
10	11	12	13	14	15	16	17	18	19	20				35	36		28		
						17	18	19	20	21				34	35		29		
24	23	22	21	20	19	18	19	20	21	22		30	31	32	33	34	30		
25		23	22	21	20	19	20	21	22	23		29		34	35		31		
26		24				21	22	23	24			28		33	34		32		
27		25				22	23	24	25	26	27			32	33		33		
28		27	26	27	26	25	24	23	24	25	26	27	28	29	30	31	32	33	34

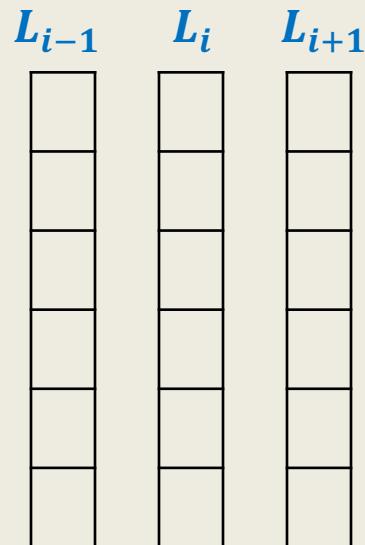
Observation: Each cell of L_{i+1} is a neighbor of a cell in L_i .

But every neighbor of L_i may be a cell of L_{i-1} or L_{i+1} .

How can we generate L_{i+1} from L_i ?

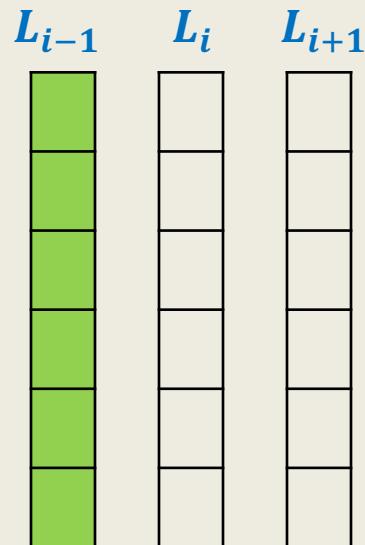
How can we generate L_{i+1} from L_i ?

How can we generate L_{i+1} from L_i ?



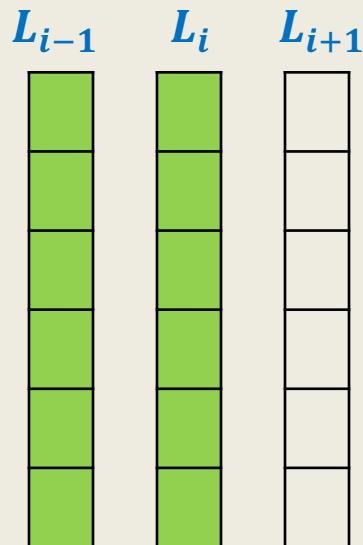
How can we generate L_{i+1} from L_i ?

Suppose all cells of L_{i-1} get visited first.



How can we generate L_{i+1} from L_i ?

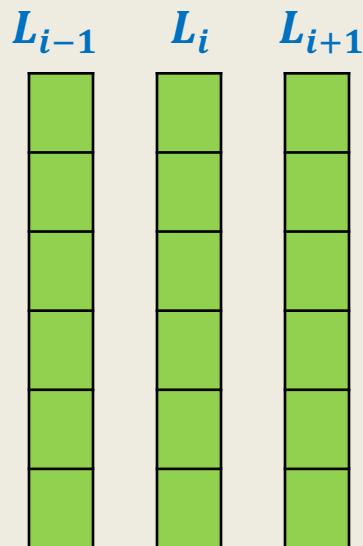
Suppose all cells of L_{i-1} get visited first.
Then all cells of L_i are visited, and



How can we generate L_{i+1} from L_i ?

Suppose all cells of L_{i-1} get visited first.

Then all cells of L_i are visited, and
then all cells of L_{i+1} are visited.



So by the time all cells of L_i are visited, if a cell neighboring to a cell of L_i is unvisited, it must be a cell of L_{i+1} .



How can we generate L_{i+1} from L_i ?

So the algorithm should be:

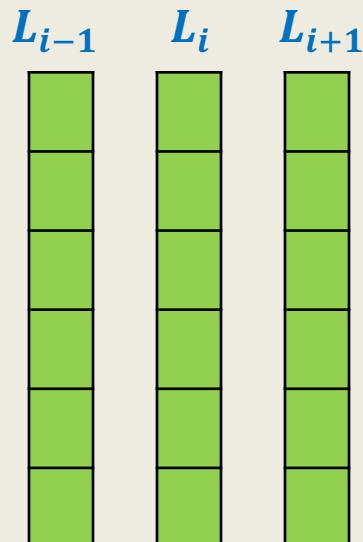
Initialize the distance of all cells except start cell as ∞

First compute L_1 .

Then using L_1 compute L_2

Then using L_2 compute L_3

...



Algorithm to compute L_{i+1} if we know L_i

Compute-next-layer(G, L_i)

{

CreateEmptyList(L_{i+1});

For each cell c in L_i

 For each neighbor b of c which is not an obstacle

 { if ($\text{Distance}[b] = \infty$)

 { Insert(b, L_{i+1});

$\text{Distance}[b] \leftarrow i + 1$;

 }

}

return L_{i+1} ;

}

The first (not so elegant) algorithm (to compute distance to all cells in the grid)

```
Distance-to-all-cells( $G$ ,  $c_0$ )
{
     $L_0 \leftarrow \{c_0\}$ ;
    For( $i = 0$  to ??)
         $L_{i+1} \leftarrow \text{Compute-next-layer}(G, L_i)$ ;
}
```

It can be as high as
 $O(n^2)$

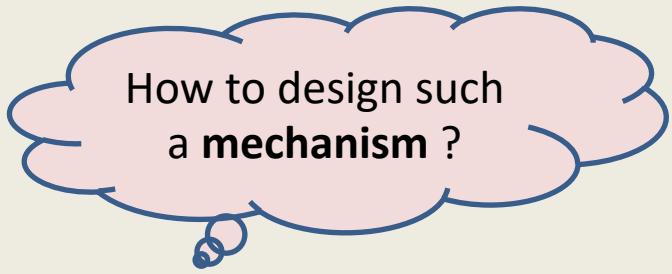
The algorithm is not elegant because of

- So many temporary lists that get created.

Towards an **elegant** algorithm ...

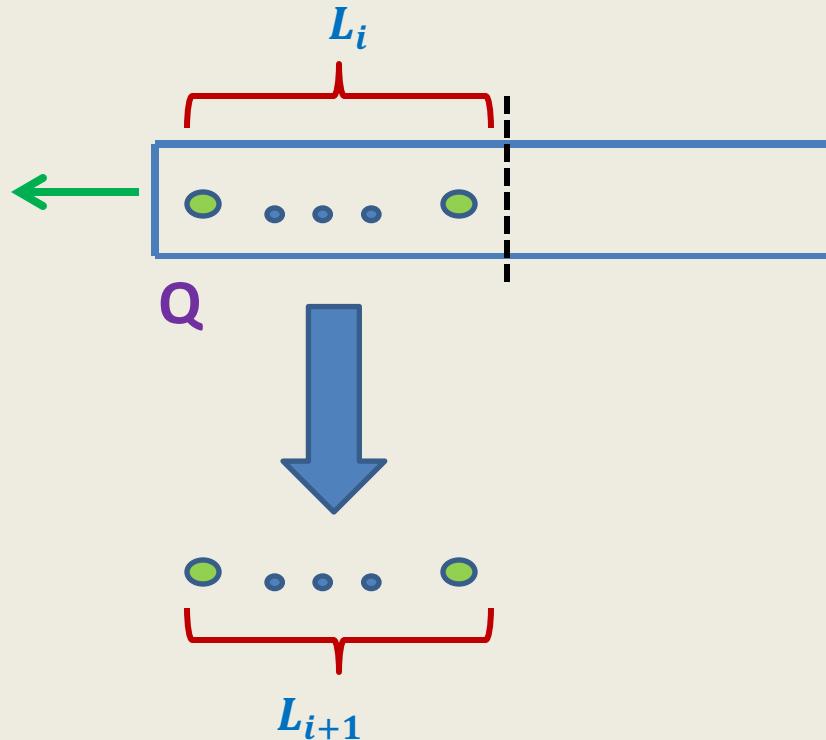
Key points we observed:

- We can compute cells at distance $i + 1$ if we know all cells up to distance i .
- Therefore, we need a mechanism to enumerate the cells in **non-decreasing** order of distances from the start cell.



How to design such
a **mechanism** ?

Keep a queue Q



Spend some time to see how seamlessly the queue ensured the requirement of visiting cells of the grid in non-decreasing order of distance.

An elegant algorithm (to compute distance to all cells in the grid)

Distance-to-all-cells(G, c_0)

CreateEmptyQueue(Q);

Distance(c_0) $\leftarrow 0$;

Enqueue(c_0, Q);

While(Not IsEmptyQueue(Q))

{ $c \leftarrow \text{Dequeue}(Q)$;

For each neighbor b of c which is not an obstacle

{ if ($\text{Distance}(b) = \infty$)

{ $\text{Distance}(b) \leftarrow \text{Distance}(c) + 1$;

 Enqueue(b, Q); ;

}

}

}

Proof of correctness of algorithm

Question: What is to be proved ?

Answer: At the end of the algorithm,

Distance[c]= the distance of cell **c** from the starting cell in the grid.

Question: How to prove ?

Answer: By the principle of mathematical induction on

the distance from the starting cell.

Inductive assertion:

P(*i*):

The algorithm correctly computes distance to all cells at distance **i** from the starting cell.

As an exercise, try to prove **P(*i*)** by induction on **i**.

Data Structures and Algorithms

(ESO207)

Lecture 13:

- Majority element : an efficient and practical algorithm
- word RAM model of computation: further refinements.

Majority element

Definition: Given a multiset S of n elements,

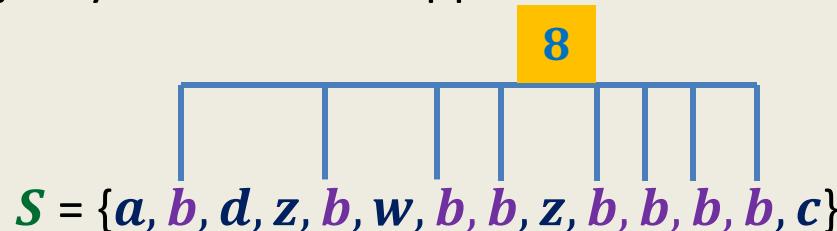
$x \in S$ is said to be majority element if it appears more than $n/2$ times in S .

$$S = \{a, b, d, z, b, w, b, b, z, b, b, b, b, c\}$$

Majority element

Definition: Given a multiset S of n elements,

$x \in S$ is said to be majority element if it appears more than $n/2$ times in S .



Problem: Given a multiset S of n elements, find the majority element, if any, in S .

Majority element

Trivial algorithms:

Algorithm 1:

1. Count occurrence of each element
2. If there is any element with count $> \frac{n}{2}$, report it.

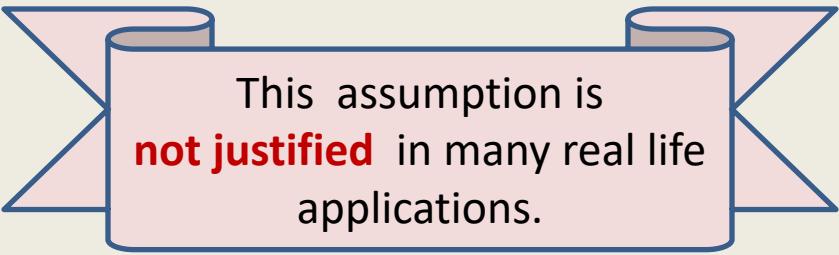
Running time: $\text{O}(n^2)$ time

Majority element

Trivial algorithms:

Algorithm 2:

1. Sort the set S to find its median
2. Let x be the median
3. Count the occurrence of x , and
4. return x if its count is more than $\frac{n}{2}$



This assumption is
not justified in many real life
applications.

Running time: $O(n \log n)$ time

Critical assumption underlying Algorithm 2 :

elements of set S can be compared under some total order ($=, <, >$)

A real life application

Slots for inserting any two cards

Card-matching machine

Problem:
Given n credit cards, determine if they are identical or not using minimum no. of operations on each card.

This machine takes two cards and determines whether they are identical or not.

Some observations

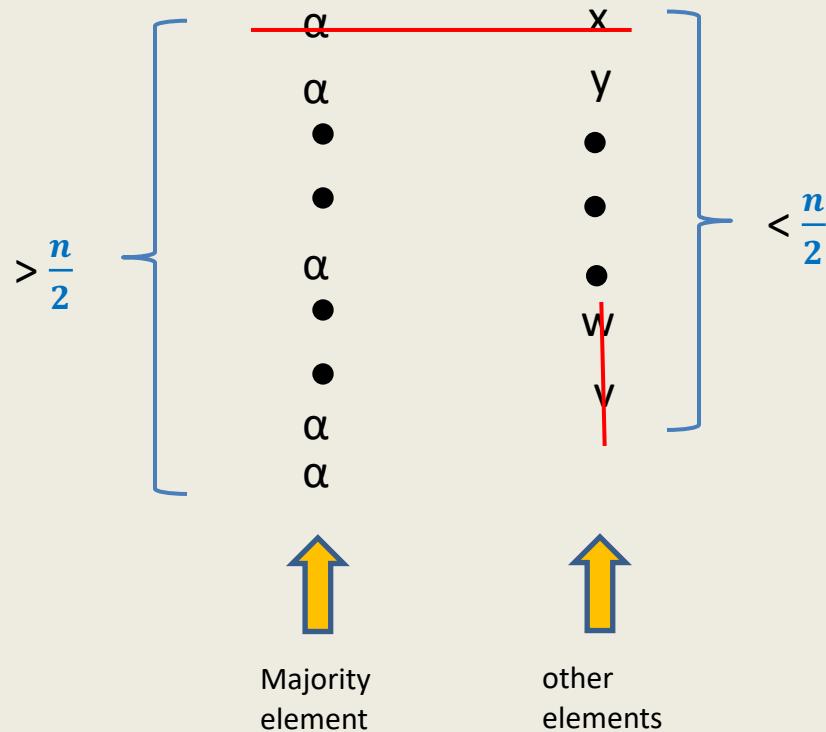
Problem: Given a multiset S of n elements,
where the only relation between any two elements is \neq or $=$,
find the majority element, if any, in S .

Question: How much time does it take to determine if an element $x \in S$ is majority ?

Answer: $O(n)$ time

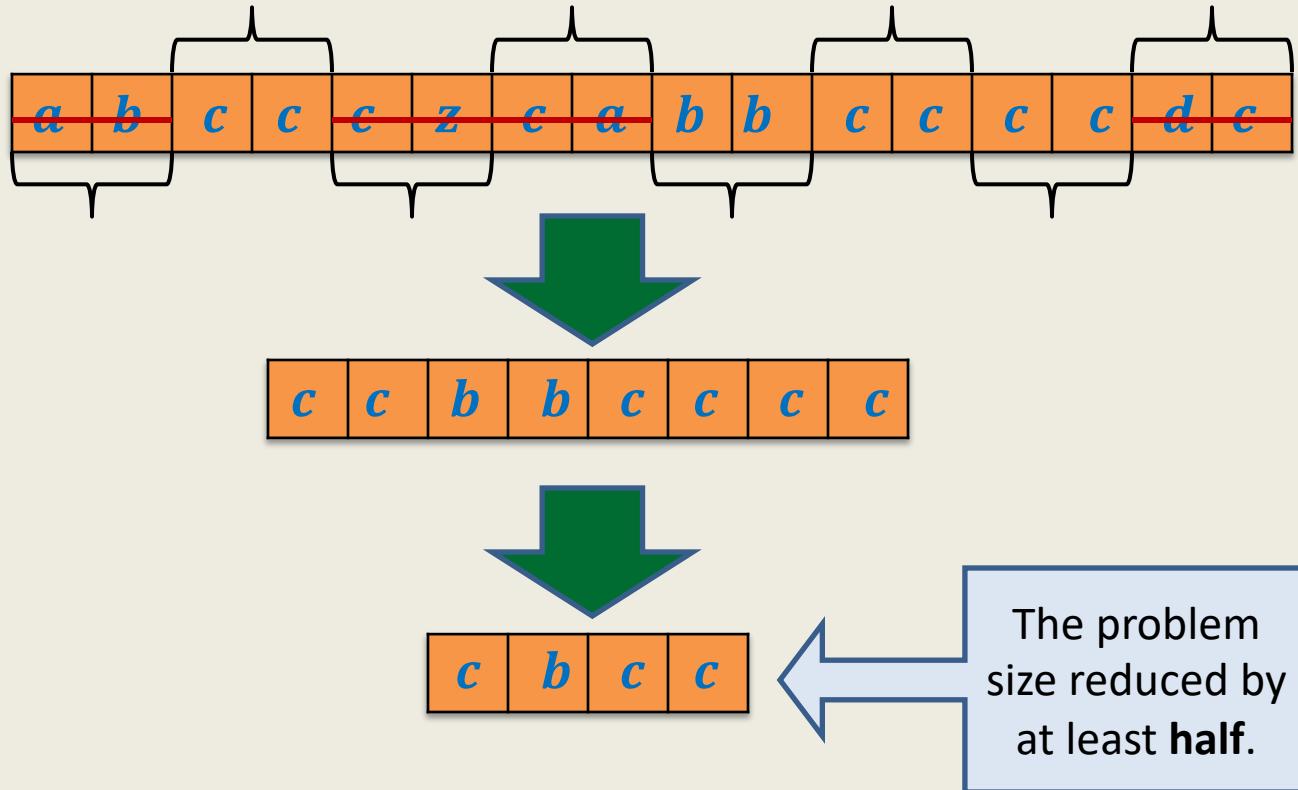
Observation 1: It is easy to verify whether an element is a majority

Some observations



Observation 2: whenever we cancel a pair of distinct elements from the array, the majority element of the array remains preserved.

Some observations



Observation 3: If there are m pairs of **identical elements**, then majority element is preserved even if we keep **one element per pair**.

Algorithm for 2-majority element

Repeat

1. Pair up the elements; Take care if the no. of elements is odd
2. **Eliminate** all pairs of distinct elements;
3. **Keep one element** per pair of identical elements.

Until only one element is left.

Verify if the last element is a **majority** element.

Time complexity:

$$T(n) = c n + c \frac{n}{2} + c \frac{n}{4} + \dots$$

O(n) time

Extra/working space requirement (assuming input is “**read only**”)

O(n)

Further restrictions on the problem

Restrictions:

- We are allowed to make single scan.
- We have very limited extra space.



Our current algorithm doesn't work
for this real life example.

Real life example:

There are 10^{12} numbers stored on hard disk.

RAM can't provide $\mathcal{O}(n)$ extra (working) space in this case.

ALGORITHM FOR 2-MAJORITY ELEMENT

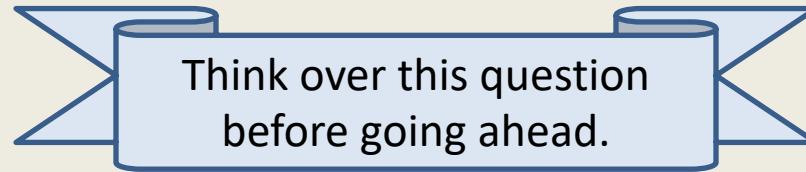
- Single scan and
- $O(1)$ extra space

Designing algorithm for 2-majority element single scan and using $O(1)$ extra space

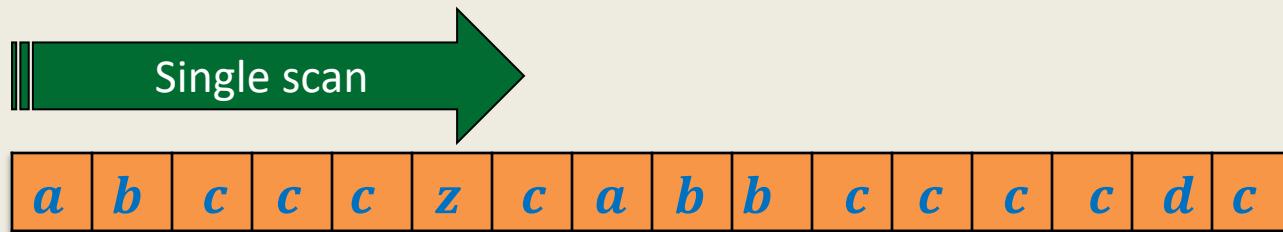
Question: Should we design algorithm from scratch to meet these constraints ?

Answer: No! We should try to adapt our current algorithm to meet these constraints.

Question: How crucial is pairing of elements in our current algorithm ?



Designing algorithm for 2-majority element single scan and using $O(1)$ extra space



Insightful questions:

- Do we really need to keep more than one element ?

No. Just cancel suitably whenever encounter two *distinct* elements.

- Do we really need to keep multiple copies of an element **explicitly** ?

No. Just keeping its count will suffice.

Ponder over these insights and make an attempt to design the algorithm
before moving ahead ☺

Algorithm for 2-majority element

single scan and using $O(1)$ extra space

Algo-2-majority(A)

```
{   count < 0;  
    for( $i = 0$  to  $n - 1$ )  
    {      if ( count = 0 ){ x < A[i];  
            count < 1;  
        }  
        else if( $x <> A[i]$ ) count <= count - 1 ;  
        else count <= count + 1 ;  
    }  
}
```

Count the occurrences of x in A , and if it is more than $n/2$, then
print(x is 2-majority element) else **print(there is no majority element in A)**

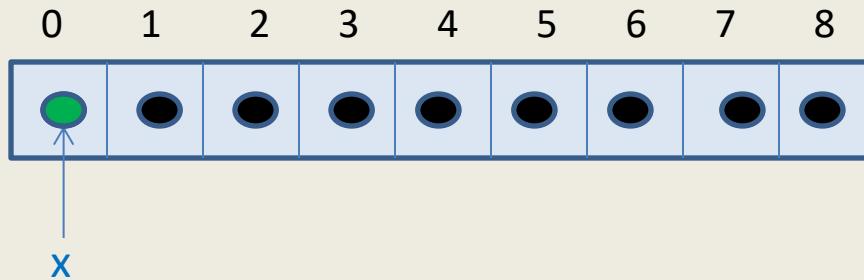
```
}
```

Algorithm for **2-majority** element single scan and using **O(1)** extra space

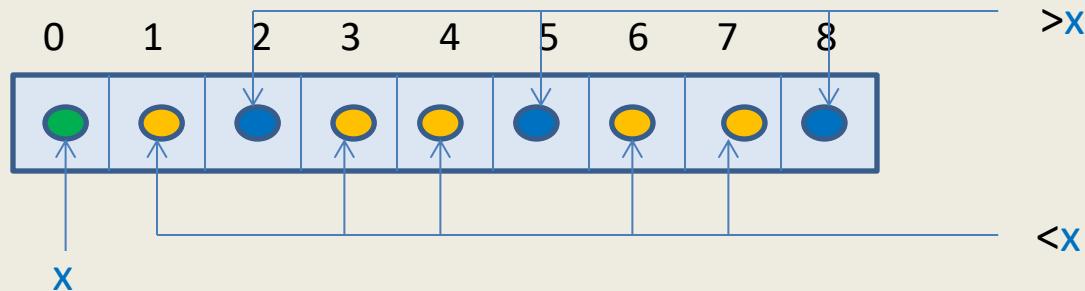
Theorem: There is an algorithm that makes just a **single scan** and uses **O(1)** **extra space** to compute majority element for a given multi-set.

Homework: Algorithm for **3-majority** element

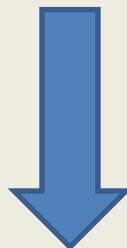
A nice programming exercise ?



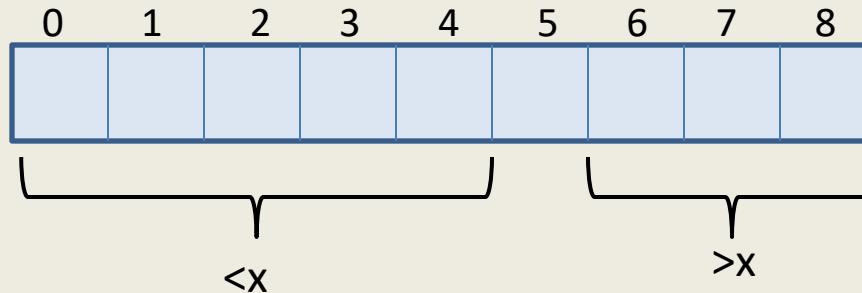
A nice programming exercise ?



A nice programming exercise ?



Implement **Partition()**
in $O(n)$ time
using $O(1)$ space?



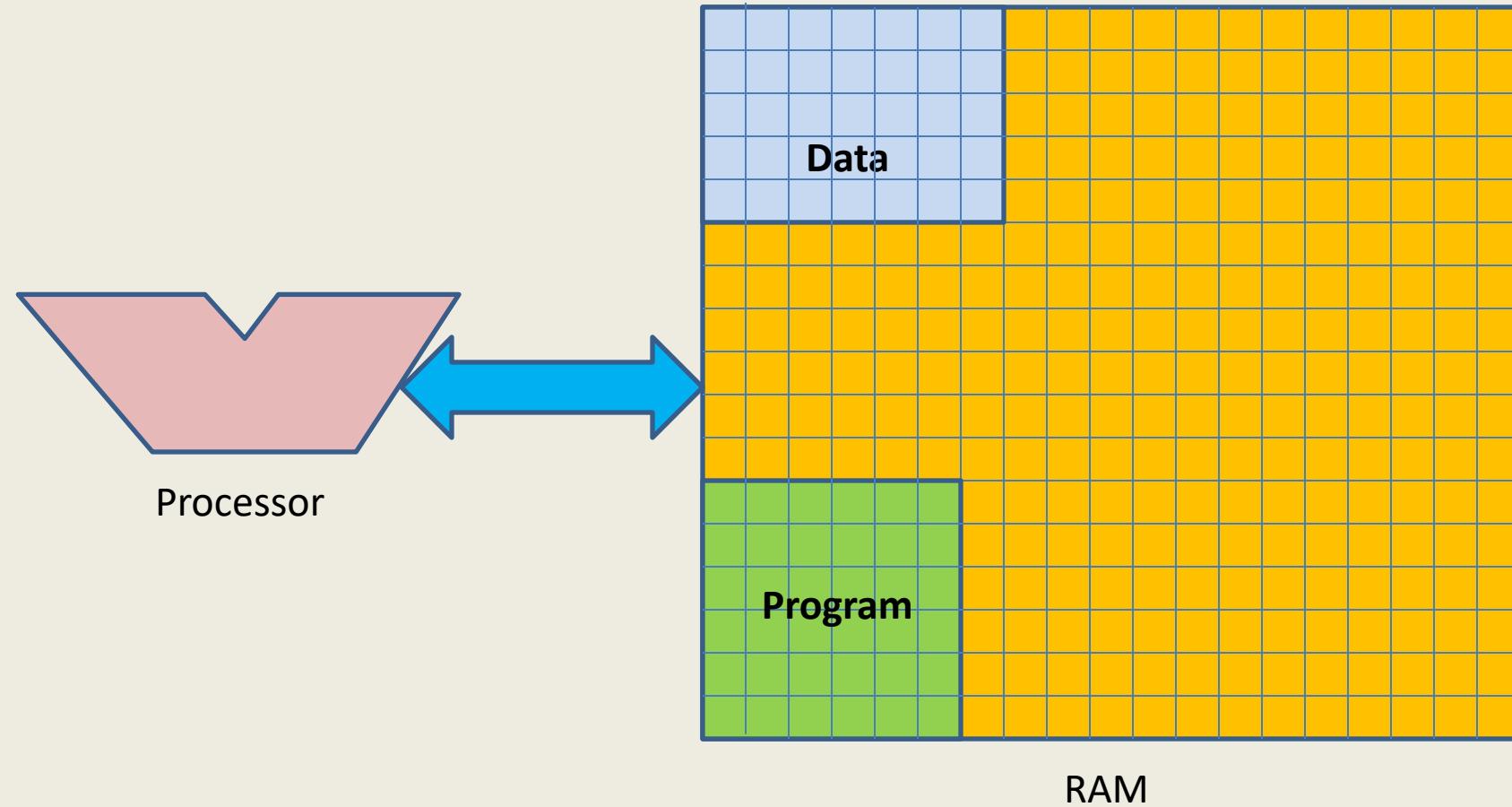
This procedure is called **Partition**.

It **rearranges** the elements so that all elements less than x appear to the left of x and all elements greater than x appear to the right of x .

Word RAM model of computation

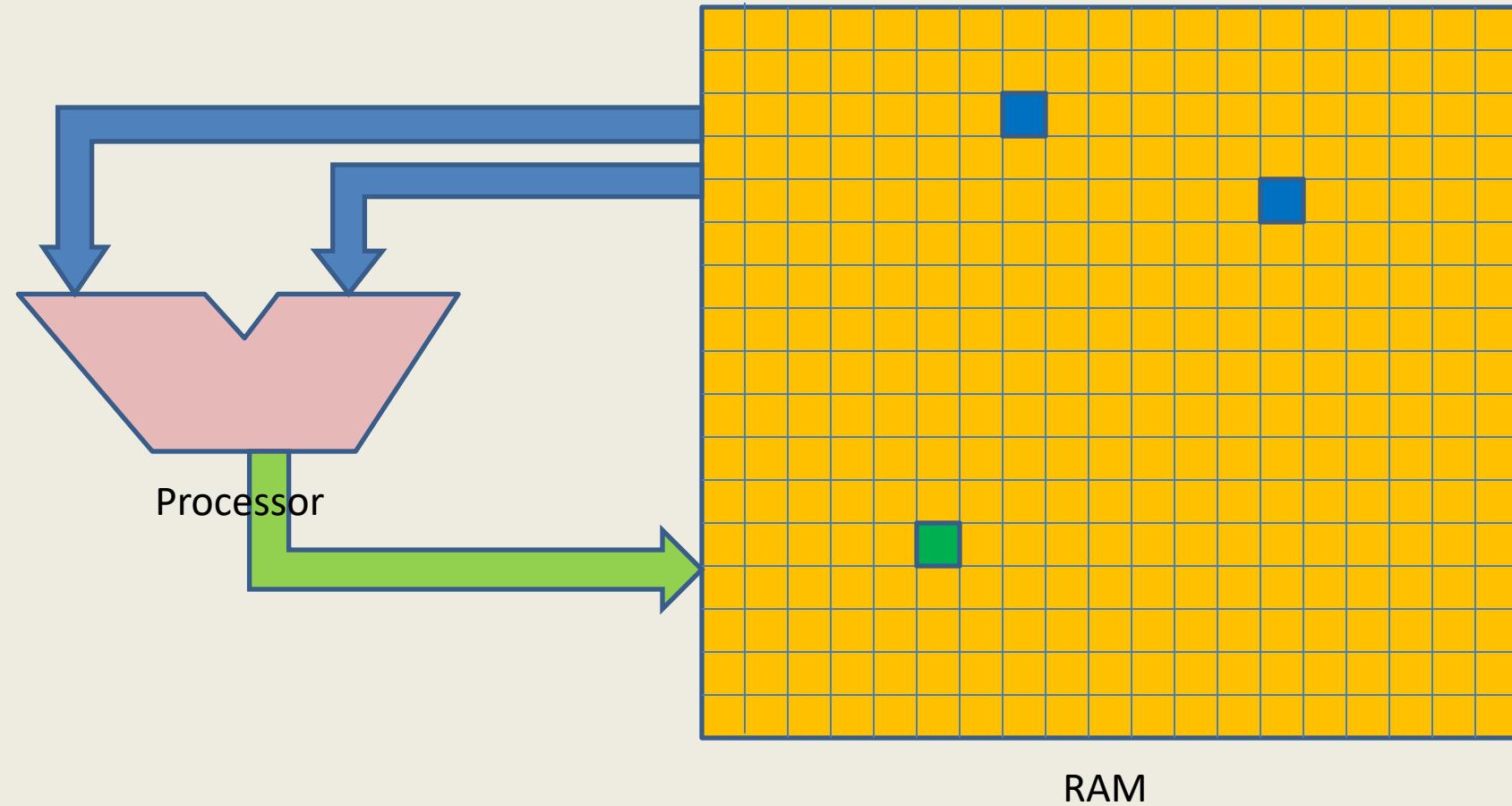
Further refinements

word RAM : a model of computation



Execution of a instruction

(fetching the operands, arithmetic/logical operation, storing the result back into RAM)



A more realistic RAM

n : input size

Input resides completely in RAM.

Question: How many bits are needed to access an input item from RAM ?

Answer: At least $\log n$.

(k bits can be used to create at most 2^k different addresses)

Current-state-of-the-art computers:

- RAM of size **4GB**

Hence 32 bits to address any item in RAM.

- Support for **64-bit arithmetic**

Ability to perform arithmetic/logical operations on any two 64-bit numbers.

word RAM model of computation: Characteristics

- Word is the basic storage unit of RAM. Word is a collection of few bytes.
- Data as well as Program reside fully in RAM.
- Each input item (number, name) is stored in binary format.
- RAM can be viewed as a huge array of words. Any arbitrary location of RAM can be accessed in the same time irrespective of the location.
- Each arithmetic or logical operation (+, -, *, /, or, xor,...) involving $O(\log n)$ bits takes a constant number of steps by the CPU, where n is the number of bits of input instance.

Data Structures and Algorithms

(ESO207)

Lecture 14:

- **Algorithm paradigms**
- **Algorithm paradigm of Divide and Conquer**

Algorithm Paradigms

Algorithm Paradigm

Motivation:

- Many problems whose algorithms are based on a common approach.
- A need of a systematic study of such widely used approaches.

Algorithm Paradigms:

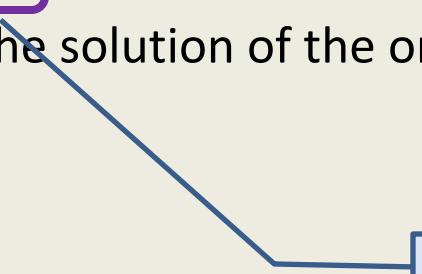
- Divide and Conquer
- Greedy Strategy
- Dynamic Programming
- Local Search

Divide and Conquer paradigm for Algorithm Design

Divide and Conquer paradigm

An Overview

1. **Divide** the problem instance into two or more instances of the same problem
2. Solve each smaller instances recursively (base case suitably defined).
3. **Combine** the solutions of the smaller instances
to get the solution of the original instance.



This is usually the main **nontrivial** step in the design of an algorithm using divide and conquer strategy

Example 1

Sorting

A familiar problem

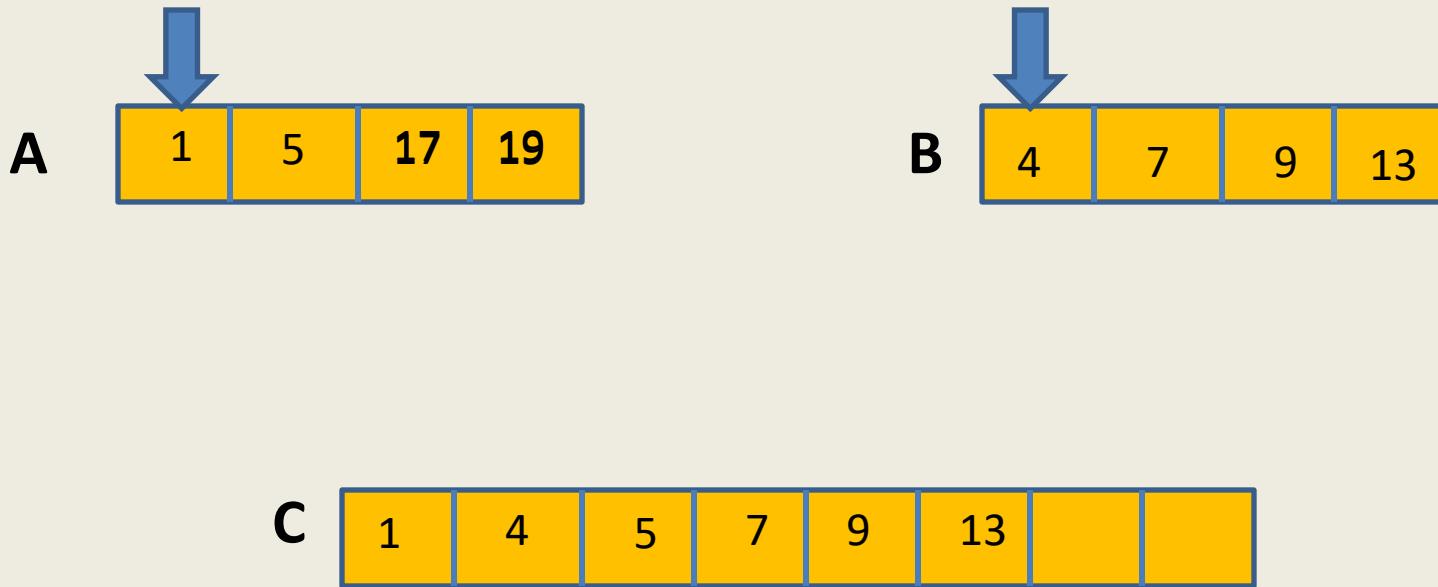
Merging two sorted arrays:

Given two sorted arrays **A** and **B** storing n elements each, Design an $O(n)$ time algorithm to output a sorted array **C** containing all elements of **A** and **B**.

Example: If **A**= $\{1,5,17,19\}$ **B**= $\{4,7,9,13\}$, then output is

C= $\{1,4,5,7,9,13,17,19\}$.

Merging two sorted arrays A and B

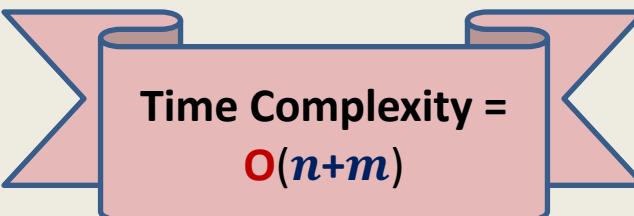


Pesudo-code for Merging two sorted arrays

Merge(A[0..n-1],B[0..m-1], C) // Merging two sorted arrays **A** and **B** into array **C**.

```
{ i<- 0; j<- 0;  
k<- 0;  
While(i<n and j<m)  
{   If(A[i]< B[j]) {   C[k] <- A[i]; k++; i++ }  
    Else          {   C[k] <- B[j]; k++; j++ }  
}  
While(i<n) { C[k] <- A[i]; k++; i++ }  
While(j<m) { C[k] <- B[j]; k++; j++ }  
return C;  
}
```

Correctness : homework exercise



Time Complexity =
O(n+m)

Divide and Conquer based sorting algorithm

MSort(A,*i,j*) // Sorting the subarray A[*i..j*].

```
{ If ( i < j )  
{ mid  $\leftarrow$  (i+j)/2;  
  MSort(A,i,mid); } } Divide step  
MSort(A,mid+1,j); } Create temporarily C[0..j - i]  
Merge(A[i..mid], A[mid+1..j], C); } } Combine/conquer step  
Copy C[0..j - i] to A[i..j] } }  
}
```

This is **Merge Sort**
algorithm

Divide and Conquer based sorting algorithm

```
MSort(A,i,j) // Sorting the subarray A[i..j].  
{  If ( i < j )  
  {    mid  $\leftarrow (\iota + j)/2$ ;  
    MSort(A,i,mid);   ← T(n/2)  
    MSort(A,mid+1,j); ← T(n/2)  
    Create temporarily C[0..j - i]  
    Merge(A[i..mid], A[mid+1..j], C); } }  
    Copy C[0..j - i] to A[i..j] } }  
}
```

Time complexity:

If $n = 1$,

$$T(n) = c \text{ for some constant } c$$

If $n > 1$,

$$T(n) = c n + 2 T(n/2)$$

$$= c n + c n + 2^2 T(n/2^2)$$

$$= c n + c n + c n + 2^3 T(n/2^3)$$

$$= c n + \dots (\log n \text{ terms}) \dots + c n$$

$$= O(n \log n)$$

Proof of correctness of Merge-Sort

MSort(A,*i,j*) // Sorting the subarray A[*i..j*].

```
{ If ( i < j )  
{   mid  $\leftarrow$  (i+j)/2;  
    MSort(A,i,mid);  
    MSort(A,mid+1,j);  
    Create temporarily C[0..j - i]  
    Merge(A[i..mid], A[mid+1..j], C);  
    Copy C[0..j - i] to A[i..j]  
}
```

Question: What is to be proved ?

Answer: **MSort(A,*i,j*)** sorts the subarray A[*i..j*]

Question: How to prove ?

Answer:

- By induction on the length (*j – i + 1*) of the subarray.
- Use correctness of the algorithm **Merge**.

Example 2

**Faster algorithm for
multiplying two integers**

Addition is faster than multiplication

Given: any two n -bit numbers X and Y

Question: how many **bit-operations** are required to compute $X+Y$?

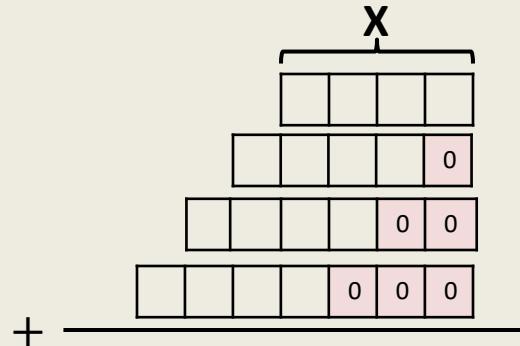
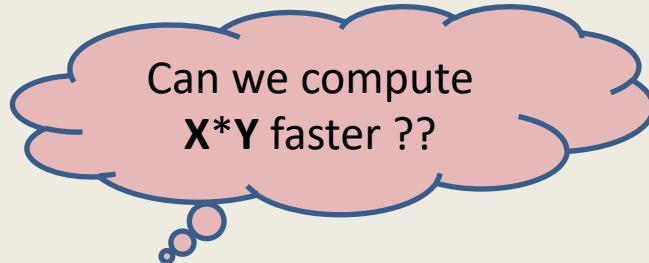
Answer: $O(n)$

Question: how many **bit-operations** are required to compute $X * 2^n$?

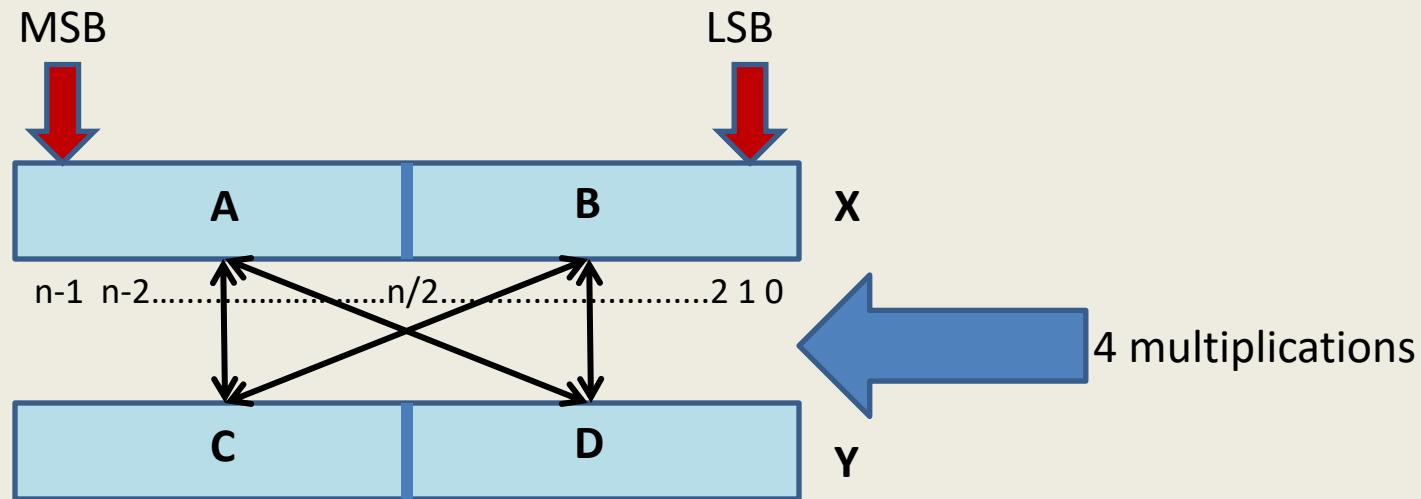
Answer: $O(n)$ [left shift the number X by n places, (do it carefully)]

Question: how many **bit-operations** are required to compute $X * Y$?

Answer: $O(n^2)$



Pursuing Divide and Conquer approach



Question: how to express $X * Y$ in terms of **multiplication/addition** of $\{A,B,C,D\}$?

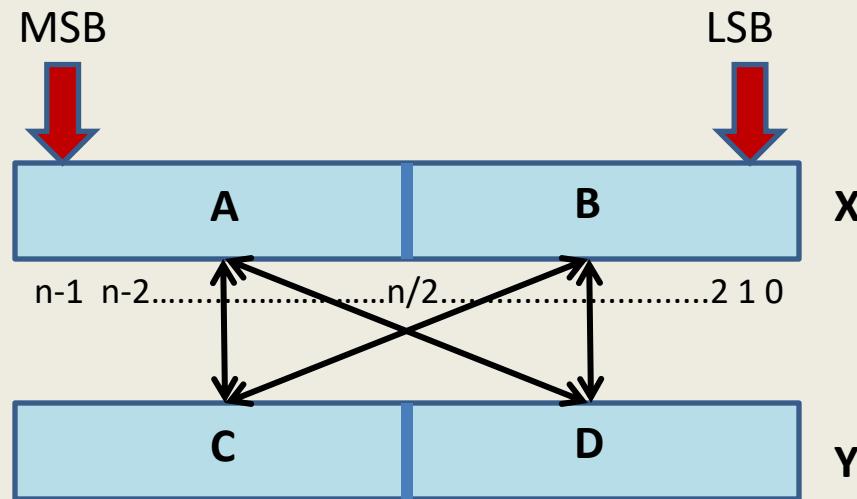
Hint: First Express X and Y in terms of $\{A,B,C,D\}$.

$$X = A * 2^{n/2} + B \quad \text{and} \quad Y = C * 2^{n/2} + D .$$

Hence ...

$$X * Y = (A * C) * 2^n + (A * D + B * C) * 2^{n/2} + B * D$$

Pursuing Divide and Conquer approach



$$X * Y = (A * C) * \boxed{2^n} + (A * D + B * C) * \boxed{2^{n/2}} + B * D$$

Let $T(n)$: time complexity of multiplying X and Y using the above equation.

$$T(n) = c n + 4 T(n/2) \text{ for some constant } c$$

$$= c n + 2c n + 4^2 T(n/2^2)$$

$$= c n + 2c n + 4c n + 4^3 T(n/2^3)$$

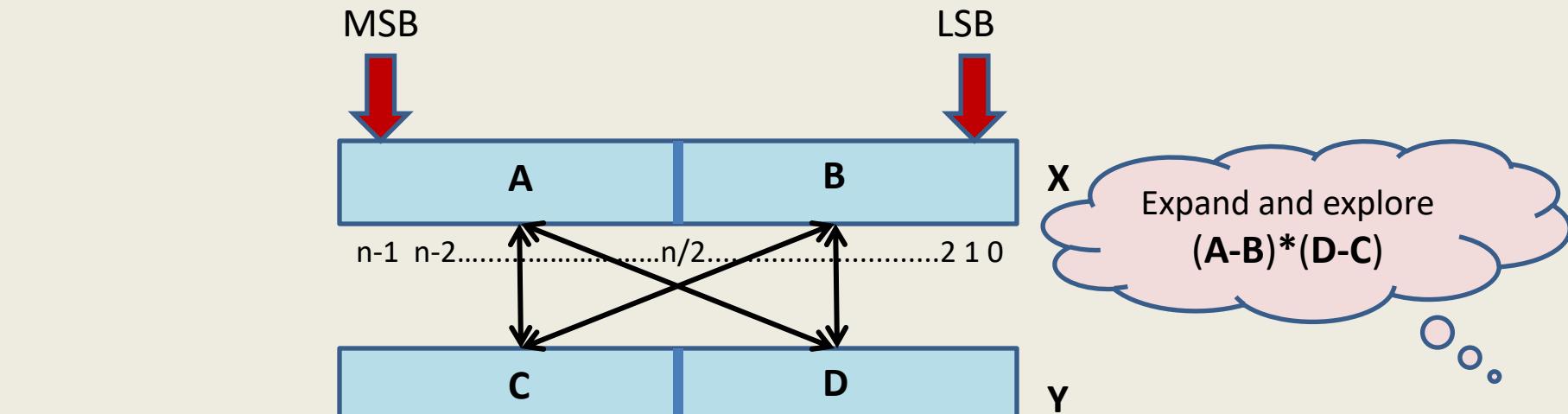
$$= c n + 2c n + 4c n + 8c n + \dots + 4^{\log_2 n} T(1)$$

$$= c n + 2c n + 4c n + 8c n + \dots + c n^2$$



$O(n^2)$ time algo

Pursuing Divide and Conquer approach



$$X*Y = (A*C)*2^n + (A*D + B*C)*2^{n/2} + B*D$$

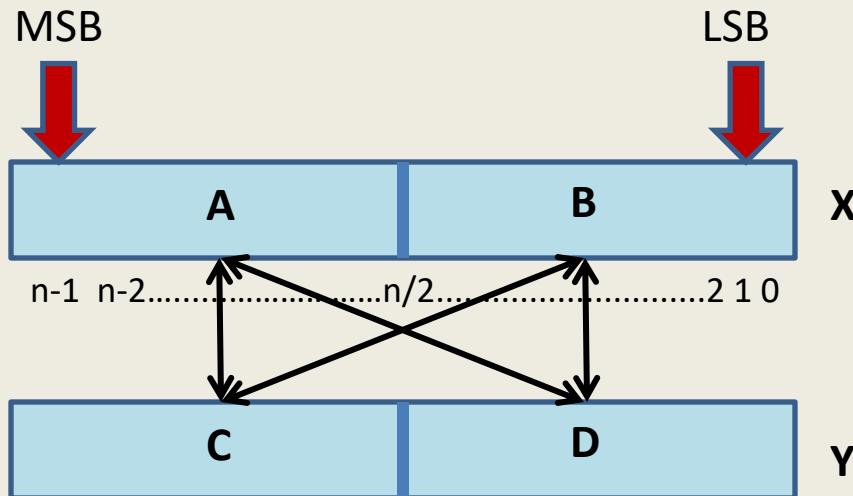
Observation: $A*D + B*C = (A-B)*(D-C) + A*C + B*D$

Question: How many multiplications do we need now to compute $X*Y$?

Answer: 3 multiplications :

- $A*C$
- $B*D$
- $(A-B)*(D-C)$.

Pursuing Divide and Conquer approach



$$X * Y = (A * C) * 2^n + ((A - B) * (D - C) + A * C + B * D) 2^{n/2} + B * D$$

Let $T(n)$: time complexity of the new algo for multiplying two n -bit numbers

$$T(n) = c n + 3 T(n/2) \text{ for some constant } c$$

$$= c n + 3 c \frac{n}{2} + 3^2 T(n/2^2)$$

$$= c n + 3c \frac{n}{2} + 9c \frac{n}{4} + \dots + 3^{\log_2 n} T(1)$$

$$= O(n^{\log_2 3}) = O(n^{1.58})$$

Conclusion

Theorem: There is a **divide and conquer** based algorithm for multiplying any two n -bit numbers in $O(n^{1.58})$ time (**bit operations**).

Note:

The fastest algorithm for this problem runs in almost $O(n \log n)$ time.

Example 3

Counting the number of
“*inversions*” in an array

Counting Inversions in an array

Problem description

Definition (Inversion): Given an array A of size n ,
a pair (i,j) , $0 \leq i < j < n$ is called an inversion if $A[i] > A[j]$.

Example:

	0	1	2	3	4	5	6	7
A	3	15	8	19	9	67	11	27

Inversions are :

$(1,2), (1,4), (1,6),$

$(3,4), (3,6),$

$(5,6), (5,7)$

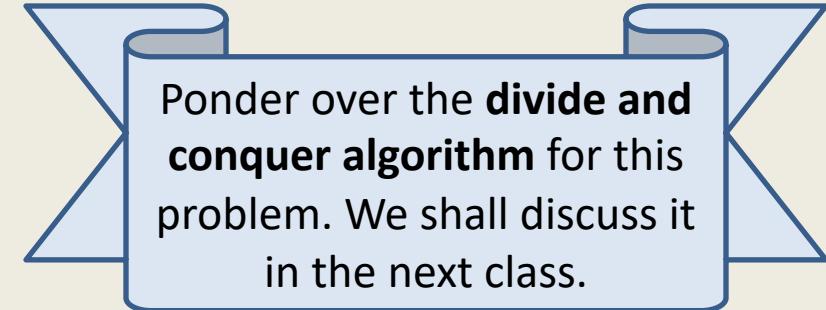
AIM: An efficient algorithm to count the number of inversions in an array A .

Counting Inversions in an array

Problem familiarization

Trivial-algo($A[0..n-1]$)

```
{ count ← 0;  
  For(j=1 to n-1) do  
  {    For( i=0 to j-1 )  
    {      If (A[i]>A[j]) count ← count + 1;  
    }  
  }  
}
```



Time complexity: $O(n^2)$

Question: What can be the max. no. of inversions in an array A ?

Answer: $\binom{n}{2}$, which is $O(n^2)$.

Question: Is the algorithm given above optimal ?

Answer: No, our aim is not to report all inversions but to report the count.

Data Structures and Algorithms

(ESO207)

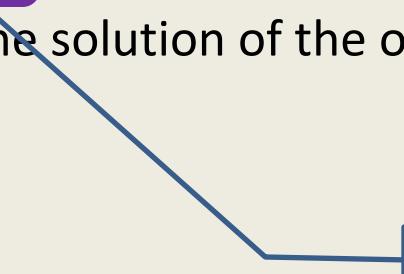
Lecture 15:

- **Algorithm paradigm of Divide and Conquer :**
Counting the number of Inversions
- **Another sorting algorithm based on Divide and Conquer : Quick Sort**

Divide and Conquer paradigm

An Overview

1. **Divide** the problem instance into two or more instances of the same problem
2. Solve each smaller instances recursively (base case suitably defined).
3. **Combine** the solutions of the smaller instances
to get the solution of the original instance.



This is usually the main **nontrivial** step
in the design of an algorithm using
divide and conquer strategy

2 IMPORTANT LESSONS

THAT WE WILL LEARN TODAY...

- 1. Role of Data structures in algorithms**
- 2. Learn from the past ...**

Role of Data Structures in designing efficient algorithms

Definition: A collection of data elements *arranged and connected* in a way which can facilitate efficient executions of a (possibly long) sequence of operations.

Parameters:

- Query/Update time
- Space
- Preprocessing time

Role of Data Structures in designing efficient algorithms

Definition: A collection of data elements *arranged and connected* in a way which can facilitate efficient executions of a (possibly long) sequence of operations.

Consider an Algorithm **A**.

Suppose **A** performs many operations of **same type** on some data.

Improving time complexity
of these operations



Improving the time complexity of **A**.

So, it is worth designing
a suitable **data structure**.

Counting Inversions in an array

Problem description

Definition (Inversion): Given an array A of size n ,
a pair (i,j) , $0 \leq i < j < n$ is called an inversion if $A[i] > A[j]$.

Example:

	0	1	2	3	4	5	6	7
A	3	15	8	19	9	67	11	27

Inversions are :

$(1,2), (1,4), (1,6),$

$(3,4), (3,6),$

$(5,6), (5,7)$

AIM: An efficient algorithm to count the number of inversions in an array A .

Counting Inversions in an array

Problem familiarization

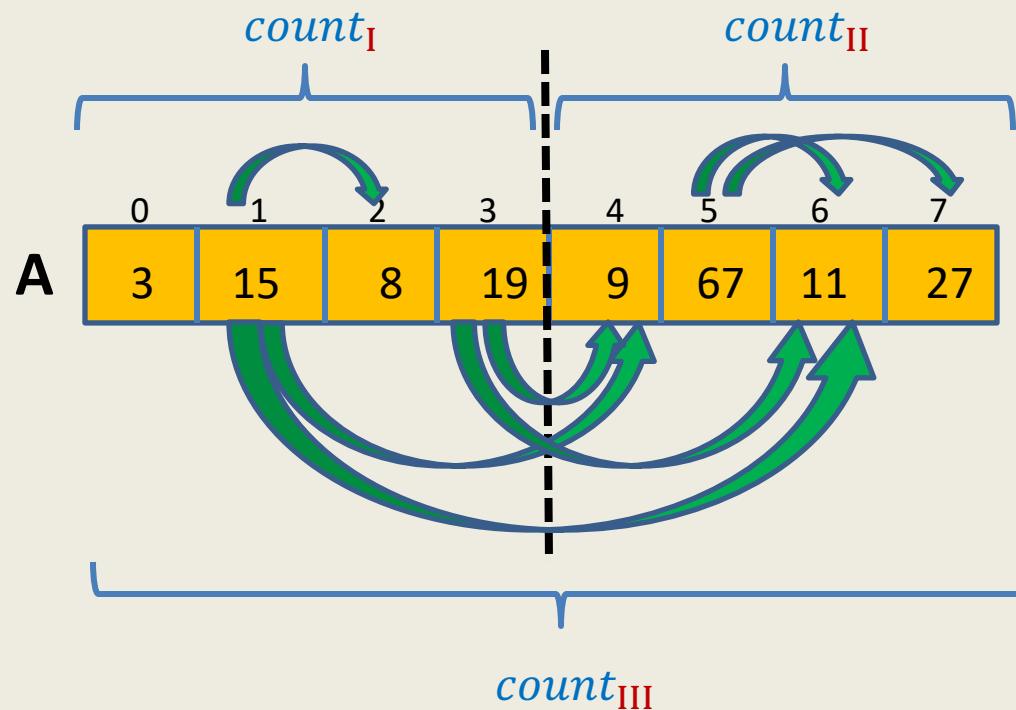
Trivial-algo($A[0..n - 1]$)

```
{ count ← 0;  
  For( $j=1$  to  $n - 1$ ) do  
  {    For(  $i=0$  to  $j - 1$  )  
      {      If (  $A[i] > A[j]$  ) count ← count + 1;  
      }  
    }  return count;  
}
```

Time complexity: $O(n^2)$

Let us try to design a
Divide and Conquer based algorithm

How do we approach using divide & conquer



Counting Inversions

Divide and Conquer based algorithm

```
CountInversion( A,i,k) // Counting no. of inversions in A[i..k]
```

```
If (i = k) return 0;
```

```
Else{ mid ← (i + k)/2;
```

```
    countI ← CountInversion(A,i,mid);
```

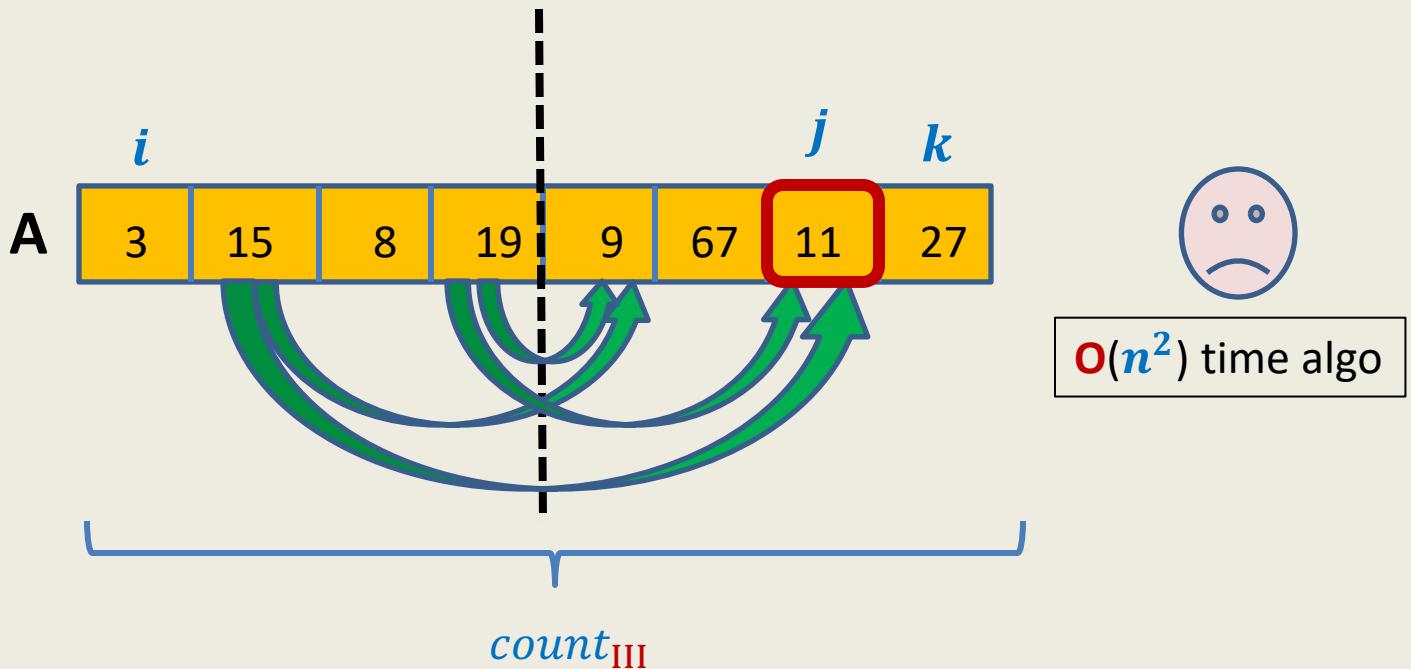
```
    countII ← CountInversion(A,mid + 1,k);
```

.... Code for *count*_{III}

```
    return countI + countII + countIII ;
```

```
}
```

How to efficiently compute $count_{III}$ (Inversions of type III) ?



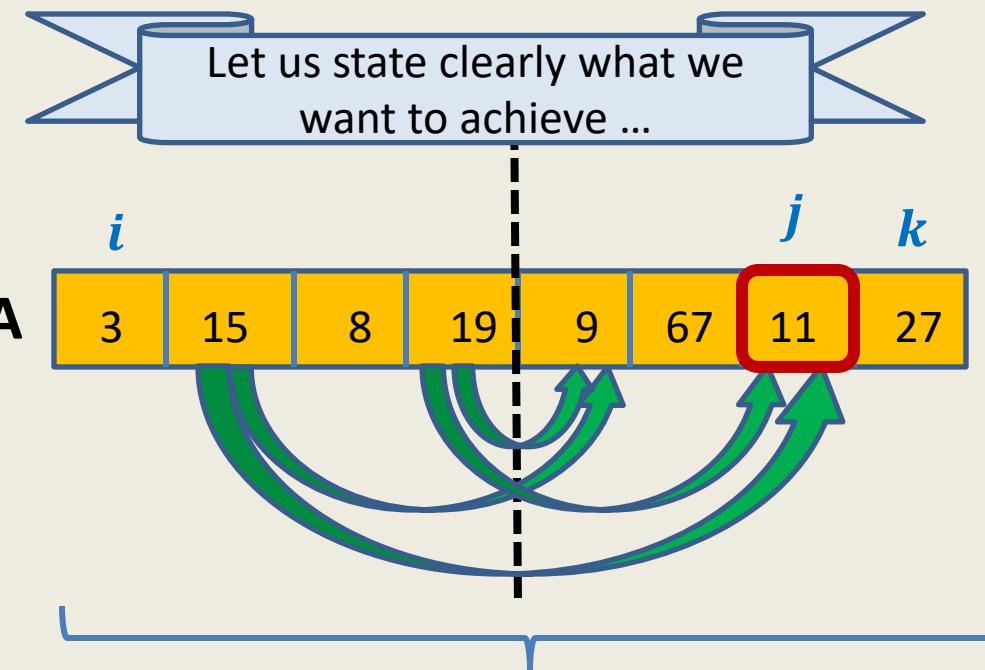
Aim: For each $mid < j \leq k$, count the elements in $A[i..mid]$ that are **greater** than $A[j]$.

Trivial way: $O(\text{size of the subarray } A[i..mid])$ time for a given j .

→ $O(n)$ time for a given j in the first call of the algorithm.

→ $O(n^2)$ time for computing $count_{III}$ since there are $n/2$ possible values of j .

How to efficiently compute $count_{III}$ (Inversions of type III) ?



count the elements in $A[i..mid]$ that are **greater** than $A[j]$.

What should be
the **data structure** ?

Time to apply Lesson 1

Sorted subarray $A[i..mid]$.

Counting Inversions

First algorithm based on divide & conquer

CountInversion(A, i , k)

If ($i = k$) return 0;

Else{ $mid \leftarrow (i + k)/2$;

$count_I \leftarrow \text{CountInversion}(A, i, mid)$;

$count_{II} \leftarrow \text{CountInversion}(A, mid + 1, k)$;

Sort(A, i , mid);

For each $mid < j \leq k$

do **binary search** for $A[j]$ in $A[i..mid]$ to compute
the *number* of elements greater than $A[j]$.

Add this *number* to $count_{III}$;

return $count_I + count_{II} + count_{III}$;

}

$2 T(n/2)$

$c n \log n$

Counting Inversions

First algorithm based on divide & conquer

Time complexity analysis:

If $n = 1$,

$$T(n) = c \text{ for some constant } c$$

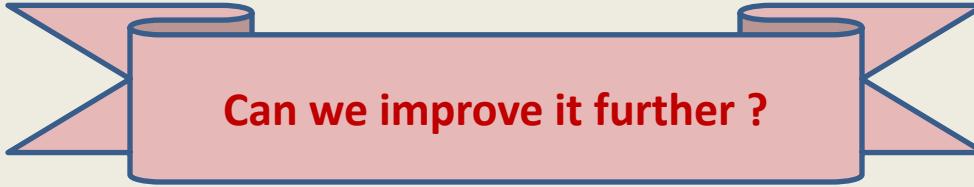
If $n > 1$,

$$T(n) = c n \log n + 2 T(n/2)$$

$$= c n \log n + c n ((\log n) - 1) + 2^2 T(n/2^2)$$

$$= c n \log n + c n ((\log n) - 1) + c n ((\log n) - 2) + 2^3 T(n/2^3)$$

$$= O(n \log^2 n)$$



Can we improve it further ?

Counting Inversions

First algorithm based on divide & conquer

CountInversion(A, i , k)

If ($i = k$) return 0;

Else{ $mid \leftarrow (i + k)/2$;

$count_I \leftarrow \text{CountInversion}(A, i, mid)$;

$count_{II} \leftarrow \text{CountInversion}(A, mid + 1, k)$;

Sort(A, i , mid);

For each $mid < j \leq k$

do **binary search** for $A[j]$ in $A[i..mid]$ to compute
the *number* of elements greater than $A[j]$.

Add this *number* to $count_{III}$;

$\} \quad 2 T(n/2)$

$c n \log n$

return $count_I + count_{II} + count_{III}$;

}

Sequence of observations

To achieve better running time

- The extra $\log n$ factor arises because for the “**combine**” step, we are spending $O(n \log n)$ time instead of $O(n)$.
- The reason for $O(n \log n)$ time for the “**combine**” step:
 - Sorting $A[0..n/2]$ takes $O(n \log n)$ time.
 - Doing **Binary Search** for $n/2$ elements from $A[n/2..n-1]$
- Each of the above tasks have optimal running time.
- So the only way to improve the running time of “**combine**” step is some new idea

Revisiting MergeSort algorithm

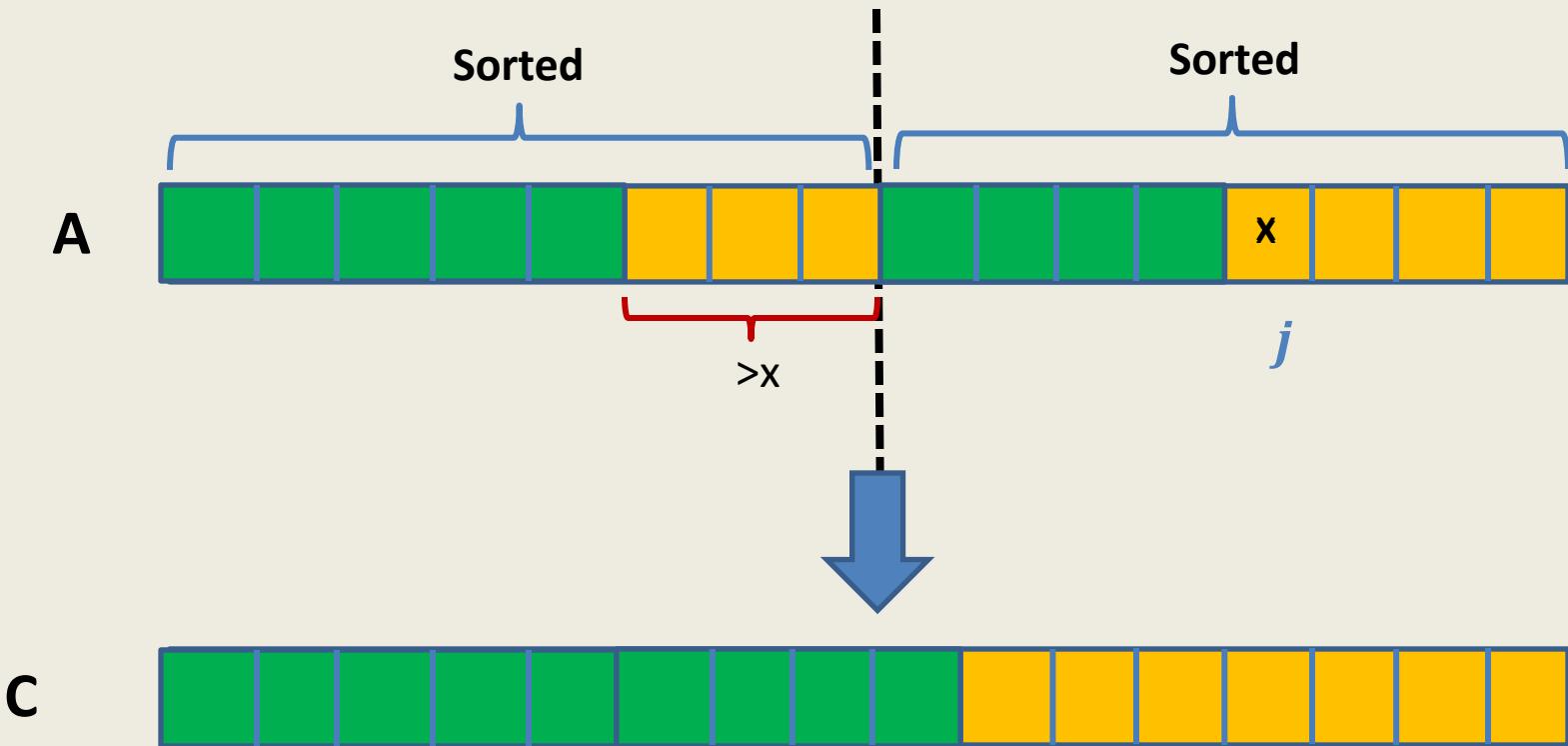
MSort(A, i , k)// Sorting A[$i..k$]

```
{  If ( $i < k$ )
{    mid  $\leftarrow (i + k)/2;$ 
    MSort(A,  $i$ , mid);
    MSort(A,  $mid + 1$ ,  $k$ );
    Create a temporary array C[0.. $k - i$ ]
    Merge(A,  $i$ , mid,  $k$ , C);
    Copy C[0.. $k - i$ ] to A[ $i..k$ ]
}
```

We shall carefully look at the **Merge()** procedure to find an efficient way to count the number of elements from A[$i..mid$] which are smaller than A[j] for any given $mid < j \leq k$

Relook

Merging $A[i..mid]$ and $A[mid + 1..k]$



Pesudo-code for Merging two sorted arrays

Merge(A,*i*,*mid*,*k*,C)

p \leftarrow *i*; *j* \leftarrow *mid* + 1; *r* \leftarrow 0;

While(*p* \leq *mid* and *j* \leq *k*)

{ **If**(A[*p*] < A[*j*]) { C[*r*] \leftarrow A[*p*]; *r*++; *p*++ }

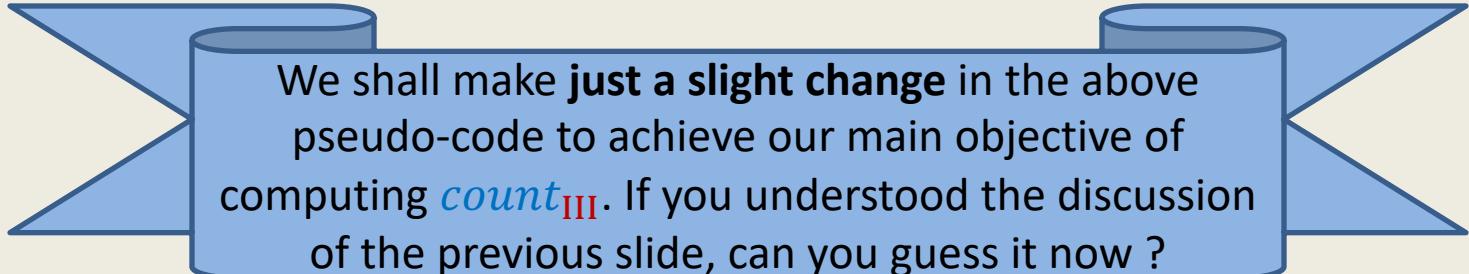
Else { C[*r*] \leftarrow A[*j*]; *r*++; *j*++ }

}

While(*p* \leq *mid*) { C[*k*] \leftarrow A[*i*]; *k*++; *i*++ }

While(*j* \leq *k*) { C[*k*] \leftarrow A[*j*]; *k*++; *j*++ }

return C ;



We shall make **just a slight change** in the above pseudo-code to achieve our main objective of computing *count*_{III}. If you understood the discussion of the previous slide, can you guess it now ?

Pesudo-code for Merging and counting inversions

Merge_and_CountInversion(A,*i*,*mid*,*k*,C)

p \leftarrow *i*; *j* \leftarrow *mid* + 1; *r* \leftarrow 0;
*count*_{III} \leftarrow 0;
While(*p* \leq *mid* and *j* \leq *k*)
{ **If**(A[*p*] < A[*j*]) { *C[r]* \leftarrow A[*p*]; *r*++; *p*++ }

Else { *C[r]* \leftarrow A[*j*]; *r*++; *j*++

*count*_{III} \leftarrow *count*_{III} + (*mid* - *p* + 1);

 }

}

While(*p* \leq *mid*) { *C[k]* \leftarrow A[*i*]; *k*++; *i*++ }

While(*j* \leq *k*) { *C[k]* \leftarrow A[*j*]; *k*++; *j*++ }

return *count*_{III};



Nothing extra is
needed here.

Counting Inversions

Final algorithm based on divide & conquer

Sort_and_CountInversion(A, i , k)

```
{  If ( $i = k$ ) return 0;  
  else  
  {    mid  $\leftarrow (i + k)/2$ ;  
    countI  $\leftarrow$  Sort_and_CountInversion (A, $i$ , mid);  
    countII  $\leftarrow$  Sort_and_CountInversion (A,mid + 1,  $k$ );  
    Create a temporary array C[ 0..  $k - i$ ]  
    countIII  $\leftarrow$  Merge_and_CountInversion(A, $i$ , mid,  $k$ ,C);  
    Copy C[0..  $k - i$ ] to A[ $i..k$ ];  
    return countI + countII + countIII ;  
  }  
}
```

$2 T(n/2)$
 $O(n)$

Counting Inversions

Final algorithm based on divide & conquer

Time complexity analysis:

If $n = 1$,

$$T(n) = c \text{ for some constant } c$$

If $n > 1$,

$$T(n) = c n + 2 T(n/2)$$

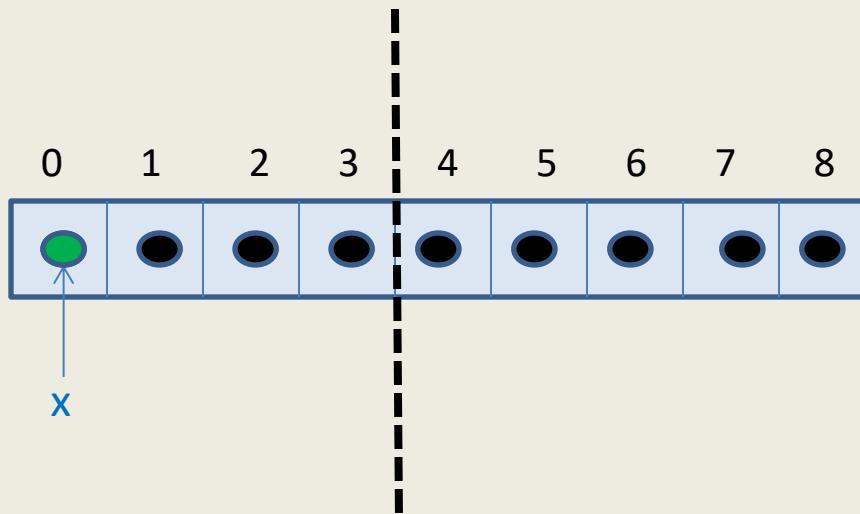
$$= O(n \log n)$$

Theorem: There is a **divide and conquer** based algorithm for computing the number of inversions in an array of size n .
The running time of the algorithm is $O(n \log n)$.

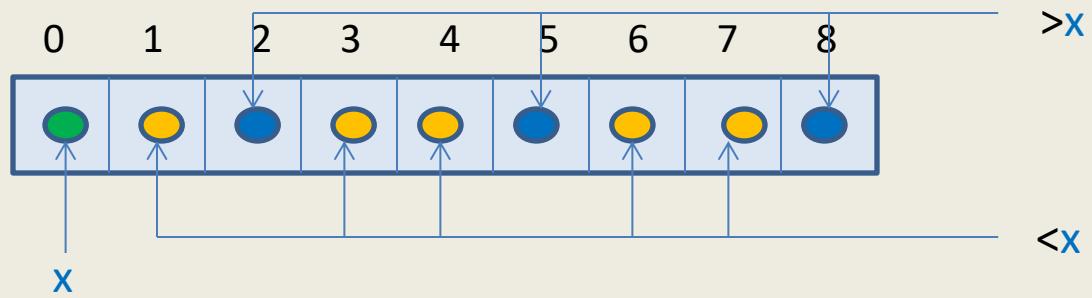
**Another sorting algorithm based on
divide and conquer**

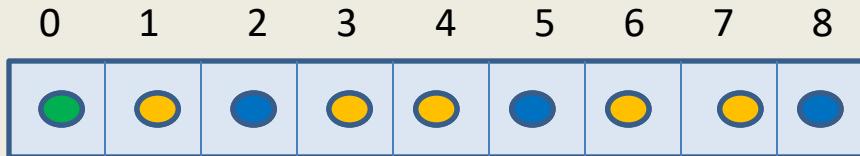
QuickSort

Is there any alternate way to divide ?

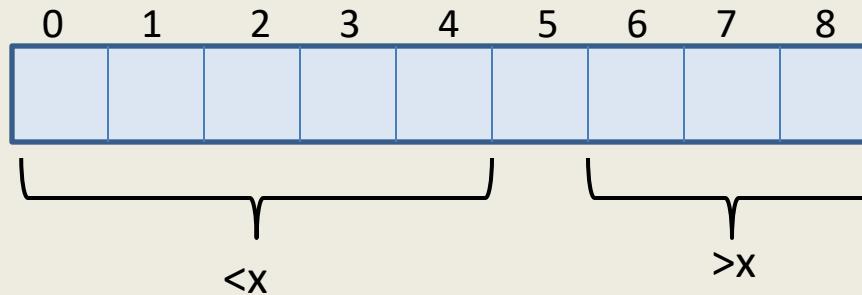
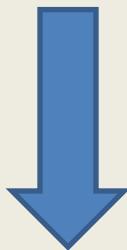


In MergeSort, we divide the input instance in an obvious manner.





Can you now guess a divide and conquer algorithm for sorting based on **Partition()** ?



This procedure is called **Partition**.

It **rearranges** the elements so that all elements less than x appear to the left of x and all elements greater than x appear to the right of x .

Pseudocode for QuickSort(S)

QuickSort(S)

{ **If** ($|S| > 1$)

Pick and remove an element x from S ;

$(S_{<x}, S_{>x}) \leftarrow \text{Partition}(S, x);$

return(Concatenate(QuickSort($S_{<x}$), x , QuickSort($S_{>x}$))

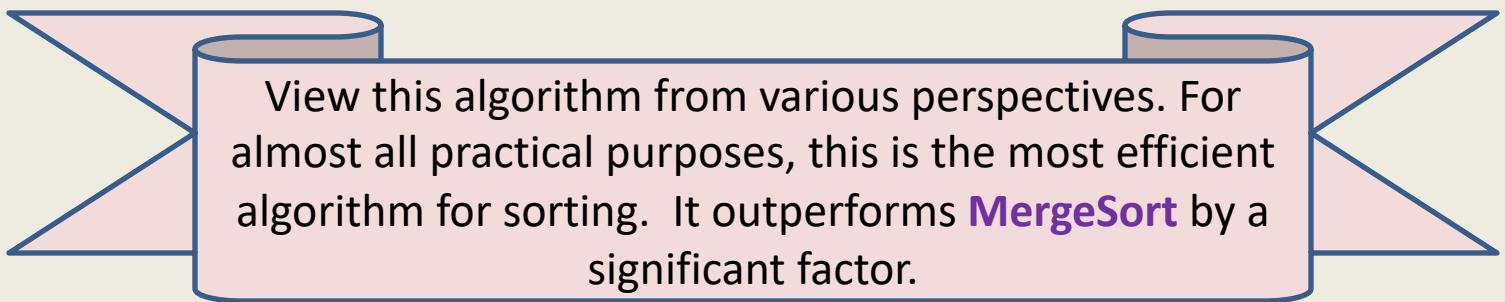
}

Pseudocode for QuickSort(S)

When the input S is stored in an array

QuickSort(A, l, r)

```
{   If ( $l < r$ )
     $i \leftarrow \text{Partition}(A, l, r); // i$  is index where element  $A[l]$  is finally placed
    QuickSort( $A, l, i - 1$ );
    QuickSort( $A, i + 1, r$ )
}
```



View this algorithm from various perspectives. For almost all practical purposes, this is the most efficient algorithm for sorting. It outperforms **MergeSort** by a significant factor.

QuickSort

Homework:

- The running time of Quick Sort depends upon the element we choose for partition in each recursive call.
- What can be the worst case running time of Quick Sort ?
- What can be the best case running time of Quick Sort ?
- Give an implementation of **Partition** that takes $O(r - l)$ time and using $O(1)$ extra space only. (Given as homework earlier)

*Sometime later in the course, we shall revisit **QuickSort** and analyze it **theoretically (average time complexity)** and **experimentally**.*

*The outcome will be **surprising** and **counterintuitive**. ☺*

Data Structures and Algorithms

(ESO207)

Lecture 16:

- Solving recurrences
that occur frequently in the analysis of algorithms.

Commonly occurring recurrences

$$T(n) = \log_{\frac{5}{2}}(n) + T(\frac{n}{5})$$

Methods for solving Recurrences

**commonly occurring in
algorithm analysis**

Methods for solving common Recurrences

- **Unfolding** the recurrence.
- **Guessing** the solution and then proving by induction.
- **A General solution** for a large class of recurrences (**Master theorem**)

Solving a recurrence by unfolding

Let $T(1) = 1$,

$T(n) = cn + 4 T(n/2)$ for $n > 1$, where c is some positive constant

Solving the recurrence for $T(n)$ by **unfolding** (expanding)

$$\begin{aligned}T(n) &= cn + 4 T(n/2) \\&= cn + 2cn + 4^2 T(n/2^2) \\&= cn + 2cn + 4cn + 4^3 T(n/2^3) \\&= cn + 2cn + 4cn + 8cn + \dots + 4^{\log_2 n} \\&= \underbrace{cn + 2cn + 4cn + 8cn + \dots}_{\text{A geometric increasing series with } \log n \text{ terms and common ratio 2}} + n^2\end{aligned}$$

A geometric increasing series with $\log n$ terms and common ratio 2

$$= \mathbf{O}(n^2)$$

Solving a recurrence by guessing and then proving by induction

$$T(1) = c_1$$

$$T(n) = 2T(n/2) + c_2 n$$

Guess: $T(n) \leq a n \log n + b$ for some con.

It looks similar/identical to the recurrence of merge sort. So we guess $T(n) = O(n \log n)$

Proof by induction:

Base case: holds true if $b \geq c_1$

Induction hypothesis: $T(k) \leq a k \log k + b$ for all $k < n$

To prove: $T(n) \leq a n \log n + b$

Proof: $T(n) = 2T(n/2) + c_2 n$

$$\leq 2(a \frac{n}{2} \log \frac{n}{2} + b) + c_2 n \quad // \text{by induction hypothesis}$$

$$= a n \log n - a n + 2 b + c_2 n$$

$$= a n \log n + b + (b + c_2 n - a n)$$

$$\leq a n \log n + b \quad \text{if } a \geq b + c_2$$

These inequalities can be satisfied simultaneously by selecting $b = c_1$ and $a = c_1 + c_2$

Hence $T(n) \leq (c_1 + c_2) n \log n + c_1$ for all value of n .
So $T(n) = O(n \log n)$

Solving a recurrence by **guessing** and then proving by **induction**

Key points:

- You have to make a right guess (past experience may help)
- What if your guess is too loose ?
- Be careful in the **induction step**.

Solving a recurrence by guessing and then proving by induction

Exercise: Find error in the following reasoning.

For the recurrence $T(1) = c_1$, and $T(n) = 2T(n/2) + c_2 n$,

one guesses $T(n) = O(n)$

Proposed (wrong) proof by induction:

Induction hypothesis: $T(k) \leq ak$ for all $k < n$

$$\begin{aligned} T(n) &= 2T(n/2) + c_2 n \\ &\leq 2(a \frac{n}{2}) + c_2 n \quad // \text{by induction hypothesis} \\ &= an + c_2 n \\ &= O(n) \end{aligned}$$

A General Method for solving a large class of Recurrences

Solving a large class of recurrences

$$T(1) = 1,$$

$$T(n) = f(n) + a T(n/b)$$

Where

- a and b are constants and $b > 1$
- $f(n)$ is a *multiplicative* function:

$$f(xy) = f(x)f(y)$$

AIM : To solve $T(n)$ for $n = b^k$

Warm-up

$\mathbf{f}(n)$ is a *multiplicative* function:

$$\mathbf{f}(xy) = \mathbf{f}(x)\mathbf{f}(y)$$

$$\mathbf{f}(1) = 1$$

$$\mathbf{f}(a^i) = \mathbf{f}(a)^i$$

$$\mathbf{f}(n^{-1}) = 1/\mathbf{f}(n)$$

Example of a *multiplicative* function : $\mathbf{f}(n) = n^\alpha$

Question: Can you express $a^{\log_b c}$ as power of c ?

Answer: $c^{\log_b a}$

Solving a slightly general class of recurrences

$$\begin{aligned} T(n) &= f(n) + a T(n/b) \\ &= f(n) + a f(n/b) + a^2 T(n/b^2) \\ &= f(n) + a f(n/b) + a^2 f(n/b^2) + a^3 T(n/b^3) \\ &= \dots \\ &= f(n) + a f(n/b) + \dots + a^i f(n/b^i) + \dots + a^{k-1} f(n/b^{k-1}) + a^k T(1) \\ &\quad \underbrace{\qquad\qquad\qquad}_{\sum_{i=0}^{k-1} a^i f(n/b^i)} \\ &= \left(\sum_{i=0}^{k-1} a^i f(n/b^i) \right) + a^k \\ &\quad \dots \text{ after rearranging } \dots \\ &= a^k + \sum_{i=0}^{k-1} a^i f(n/b^i) \\ &\quad \dots \text{ continued to the next page } \dots \end{aligned}$$

$$T(n) = a^k + \sum_{i=0}^{k-1} a^i f(n/b^i)$$

(since f is multiplicative)

$$= a^k + \sum_{i=0}^{k-1} a^i f(n)/f(b^i)$$

$$= a^k + \sum_{i=0}^{k-1} a^i f(n)/(f(b))^i$$

$$= a^k + f(n) \sum_{i=0}^{k-1} a^i / (f(b))^i$$

$$= a^k + f(n) \underbrace{\sum_{i=0}^{k-1} (a/f(b))^i}_{\text{A geometric series}}$$

$$= a^k + (f(b))^k \sum_{i=0}^{k-1} (a/f(b))^i$$

Case 1: $a = f(b)$, $T(n) = a^k(k+1) = O(a^{\log_b n} \log_b n) = O(n^{\log_b a} \log_b n)$

$$T(n) = a^k + \sum_{i=0}^{k-1} a^i f(n/b^i)$$

(since f is multiplicative)

$$= a^k + \sum_{i=0}^{k-1} a^i f(n)/f(b^i)$$

$$= a^k + \sum_{i=0}^{k-1} a^i f(n)/(f(b))^i$$

$$= a^k + f(n) \sum_{i=0}^{k-1} a^i / (f(b))^i$$

$$= a^k + f(n) \sum_{i=0}^{k-1} (a/f(b))^i$$

$$= a^k + (f(b))^k \boxed{\sum_{i=0}^{k-1} (a/f(b))^i}$$

For $a < f(b)$, the sum of this series is bounded by

$$\frac{1}{1 - \frac{a}{f(b)}} = O(1)$$

Case 2: $a < f(b)$, $T(n) =$

$$a^k + (f(b))^k \cdot O(1)$$

$$= O((f(b))^k)$$

$$= O(f(n))$$

$$T(n) = a^k + \sum_{i=0}^{k-1} a^i f(n/b^i)$$

(since f is multiplicative)

$$= a^k + \sum_{i=0}^{k-1} a^i f(n)/f(b^i)$$

$$= a^k + \sum_{i=0}^{k-1} a^i f(n)/(f(b))^i$$

$$= a^k + f(n) \sum_{i=0}^{k-1} a^i / (f(b))^i$$

$$= a^k + f(n) \sum_{i=0}^{k-1} (a/f(b))^i$$

$$= a^k + (f(b))^k \boxed{\sum_{i=0}^{k-1} (a/f(b))^i}$$

For $a > f(b)$, the sum of this series is equal to

$$\frac{\left(\frac{a}{f(b)}\right)^k - 1}{\frac{a}{f(b)} - 1}$$

Case 3: $a > f(b)$, $T(n) = \boxed{a^k + O(a^k)}$ $= O(n^{\log_b a})$

Three cases

$$T(n) = a^k + (f(b))^k \sum_{i=0}^{k-1} (a/f(b))^i$$

Case 1: $a = f(b)$,

$$T(n) = O(n^{\log_b a} \log_b n)$$

Case 2: $a < f(b)$,

$$T(n) = O(f(n))$$

Case 3: $a > f(b)$,

$$T(n) = O(n^{\log_b a})$$

Master theorem

$T(1) = 1,$

$T(n) = f(n) + a T(n/b)$ where f is multiplicative.

There are the following solutions

Case 1: $a = f(b)$, $T(n) = n^{\log_b a} \log_b n$

Case 2: $a < f(b)$, $T(n) = O(f(n))$

Case 3: $a > f(b)$, $T(n) = O(n^{\log_b a})$

Examples

Master theorem

If $T(1) = 1$, and $T(n) = f(n) + a T(n/b)$ where f is multiplicative, then there are the following solutions

Case 1: $a = f(b)$, $T(n) = n^{\log_b a} \log_b n$

Case 2: $a < f(b)$, $T(n) = O(f(n))$

Case 3: $a > f(b)$, $T(n) = O(n^{\log_b a})$

Example 1: $T(n) = n + 4 T(n/2)$

Solution: $T(n) =$ $O(n^2)$



This is case 3

Master theorem

If $T(1) = 1$, and $T(n) = f(n) + a T(n/b)$ where f is multiplicative, then there are the following solutions

Case 1: $a = f(b)$, $T(n) = n^{\log_b a} \log_b n$

Case 2: $a < f(b)$, $T(n) = O(f(n))$

Case 3: $a > f(b)$, $T(n) = O(n^{\log_b a})$

Example 2: $T(n) = n^2 + 4 T(n/2)$

Solution: $T(n) = O(n^2 \log_2 n)$



This is case 1

Master theorem

If $T(1) = 1$, and $T(n) = f(n) + a T(n/b)$ where f is multiplicative, then there are the following solutions

Case 1: $a = f(b)$, $T(n) = n^{\log_b a} \log_b n$

Case 2: $a < f(b)$, $T(n) = O(f(n))$

Case 3: $a > f(b)$, $T(n) = O(n^{\log_b a})$

Example 3: $T(n) = n^3 + 4 T(n/2)$

Solution: $T(n) = O(n^3)$



This is case 2

Master theorem

If $T(1) = 1$, and $T(n) = f(n) + a T(n/b)$ where f is multiplicative, then there are the following solutions

Case 1: $a = f(b)$, $T(n) = n^{\log_b a} \log_b n$

Case 2: $a < f(b)$, $T(n) = O(f(n))$

Case 3: $a > f(b)$, $T(n) = O(n^{\log_b a})$

Example 4: $T(n) = 2 n^{1.5} + 3 T(n/2)$

Solution: $T(n) =$

We can not apply master theorem directly since $f(n) = 2 n^{1.5}$ is not multiplicative.

Master theorem

If $T(1) = 1$, and $T(n) = f(n) + a T(n/b)$ where f is multiplicative, then there are the following solutions

Case 1: $a = f(b)$, $T(n) = n^{\log_b a} \log_b n$

Case 2: $a < f(b)$, $T(n) = O(f(n))$

Case 3: $a > f(b)$, $T(n) = O(n^{\log_b a})$

Example 4: $T(n) = 2 n^{1.5} + 3 T(n/2)$

Solution: $G(n) = T(n)/2$

$$\rightarrow G(n) = n^{1.5} + 3 G(n/2)$$

$$\rightarrow G(n) = O(n^{\log_2 3}) = O(n^{1.58})$$

$$\rightarrow T(n) = O(n^{1.58}).$$



This is case 3

Master theorem

If $T(1) = 1$, and $T(n) = f(n) + a T(n/b)$ where f is multiplicative, then there are the following solutions

Case 1: $a = f(b)$, $T(n) = n^{\log_b a} \log_b n$

Case 2: $a < f(b)$, $T(n) = O(f(n))$

Case 3: $a > f(b)$, $T(n) = O(n^{\log_b a})$

Example 6: $T(n) = T(\sqrt{n}) + c n$

Solution: $T(n) =$

We can not apply master theorem directly since $T(\sqrt{n}) \neq T(n/b)$ for any constant b .

Solving $T(n) = T(\sqrt{n}) + c n$ using the method of unfolding

$$\begin{aligned}T(n) &= c n + T(\sqrt{n}) \\&= c n + c\sqrt{n} + T(\sqrt[4]{n}) \\&= c n + c\sqrt{n} + c \sqrt[4]{n} + T(\sqrt[8]{n}) \\&= c n + c\sqrt{n} + \dots c \sqrt[i]{n} + \dots + T(1)\end{aligned}$$

A series which is decreasing at a rate faster than any geometric series

$$= \mathbf{O}(n)$$

Can you guess the number of terms in this series ?



Master theorem

If $T(1) = 1$, and $T(n) = f(n) + a T(n/b)$ where f is multiplicative, then there are the following solutions

Case 1: $a = f(b)$, $T(n) = n^{\log_b a} \log_b n$

Case 2: $a < f(b)$, $T(n) = O(f(n))$

Case 3: $a > f(b)$, $T(n) = O(n^{\log_b a})$

Example 5: $T(n) = n (\log n)^2 + 2 T(n/2)$

Solution: $T(n) =$

Using the method of “unfolding”,
it can be shown that $T(n) = O(n (\log n)^3)$.

Homework

Solve the following recurrences systematically (if possible by various methods). Assume that $T(1) = 1$ for all these recurrences.

- $T(n) = 1 + 2 T(n/2)$
- $T(n) = n^3 + 2 T(n/2)$
- $T(n) = n^2 + 7 T(n/3)$
- $T(n) = n / \log n + 2T(n/2)$
- $T(n) = 1 + T(n/5)$
- $T(n) = \sqrt{n} + 2 T(n/4)$
- $T(n) = 1 + T(\sqrt{n})$
- $T(n) = n + T(9n/10)$
- $T(n) = \log n + T(n/4)$

Data Structures and Algorithms

(ESO207)

Lecture 17:

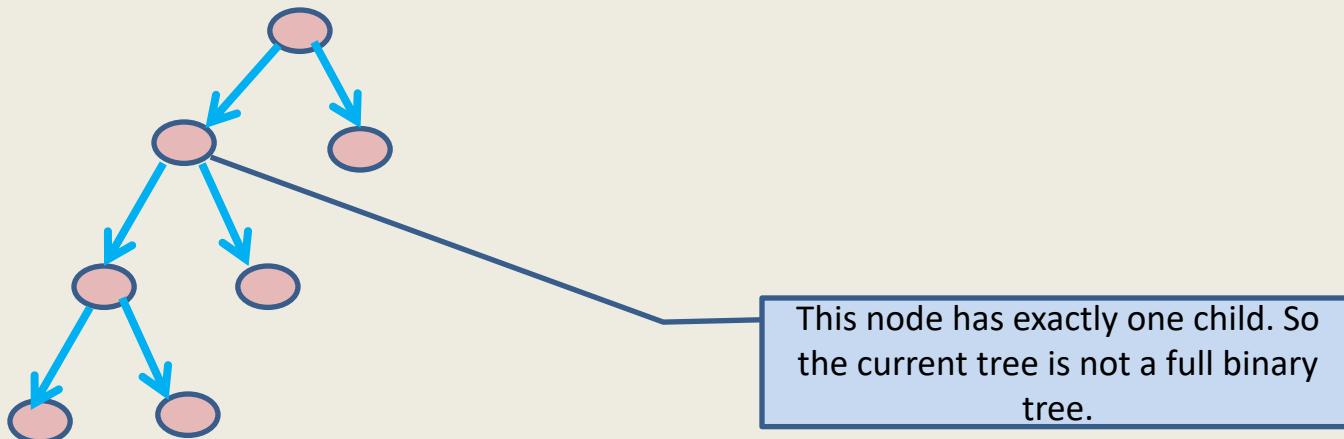
Height balanced BST

- Red-black trees

Terminologies

Full binary tree:

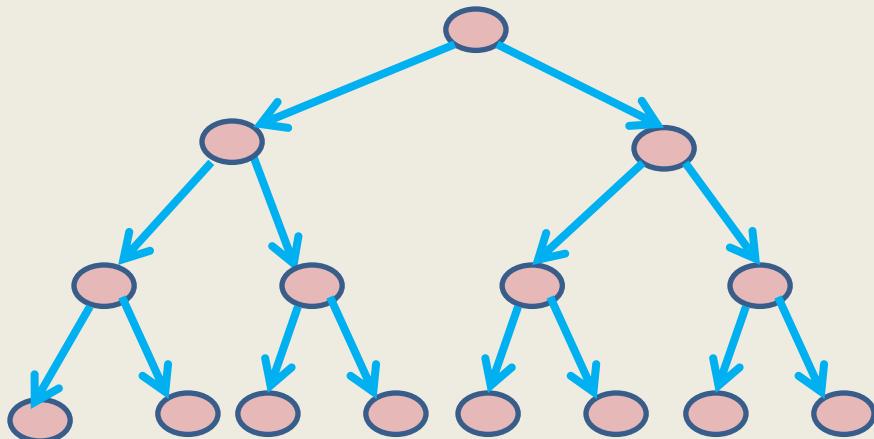
A binary tree where every internal node has exactly two children.



Terminologies

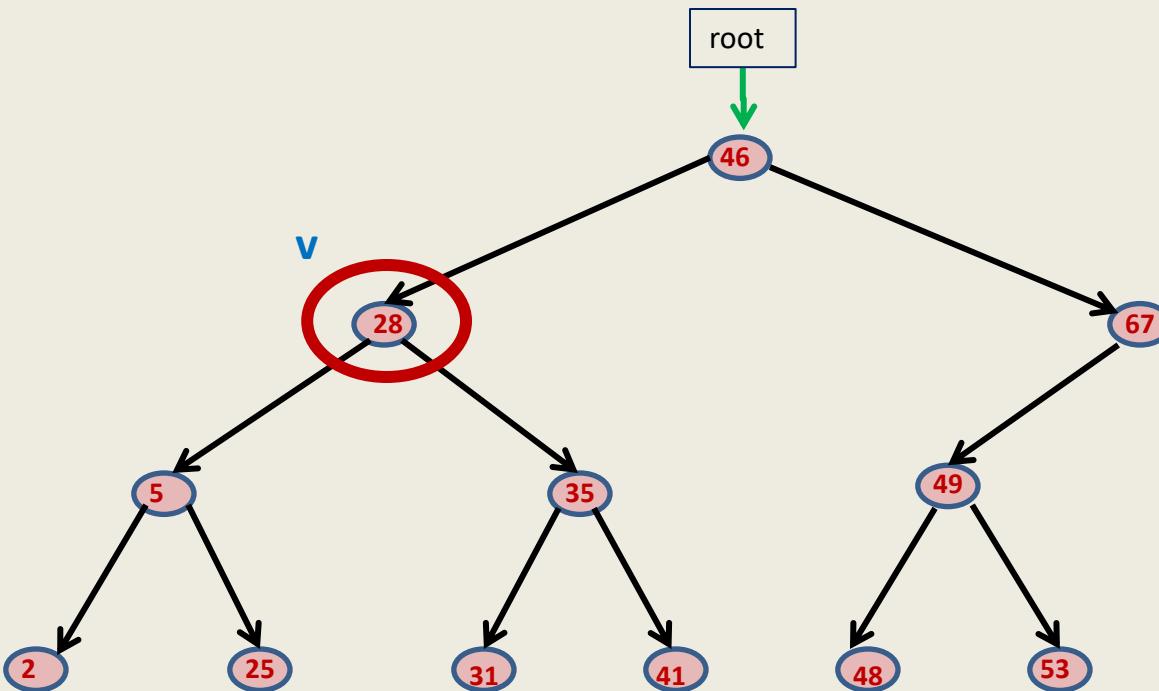
Complete binary tree:

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



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

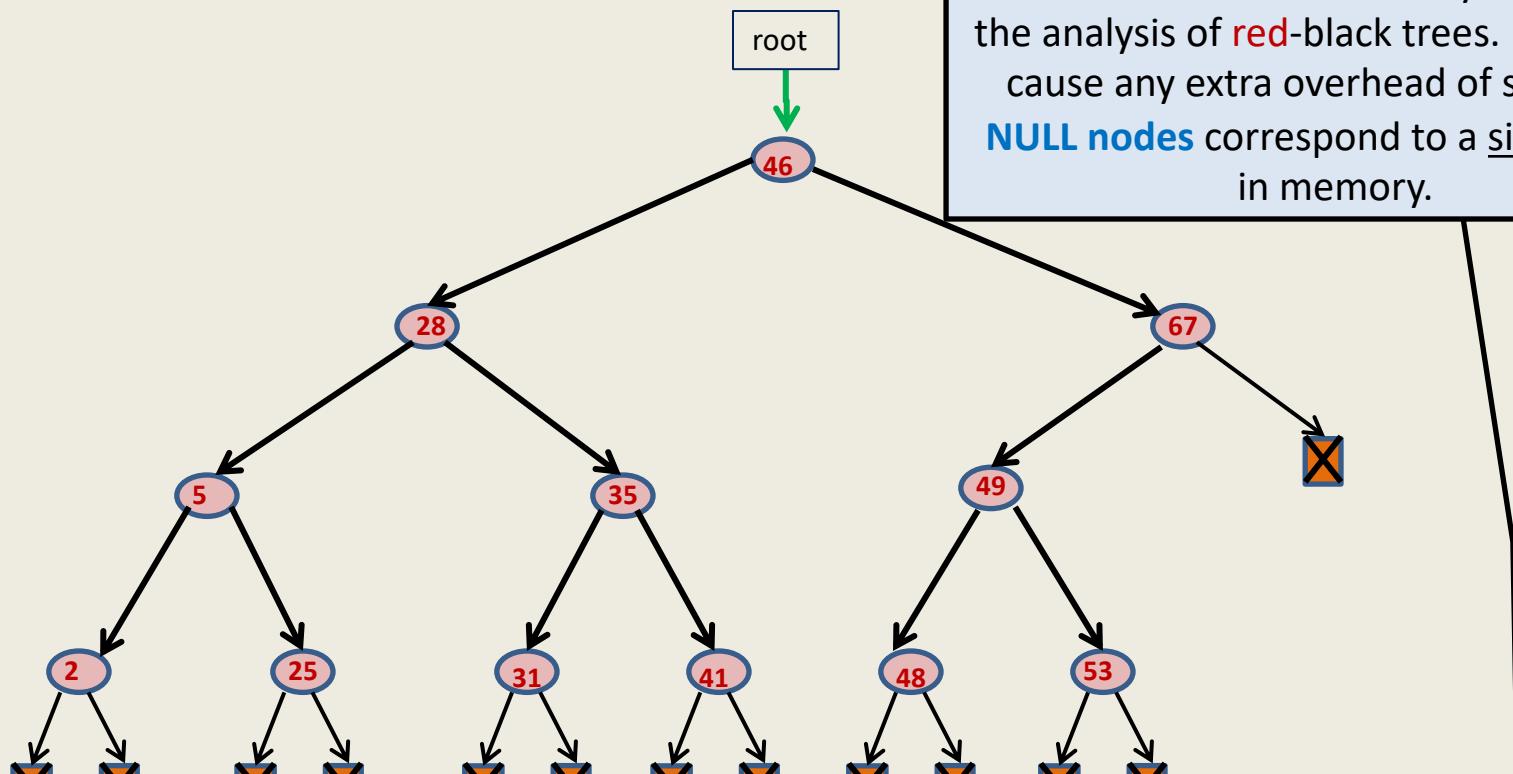
Binary Search Tree



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

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

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**.

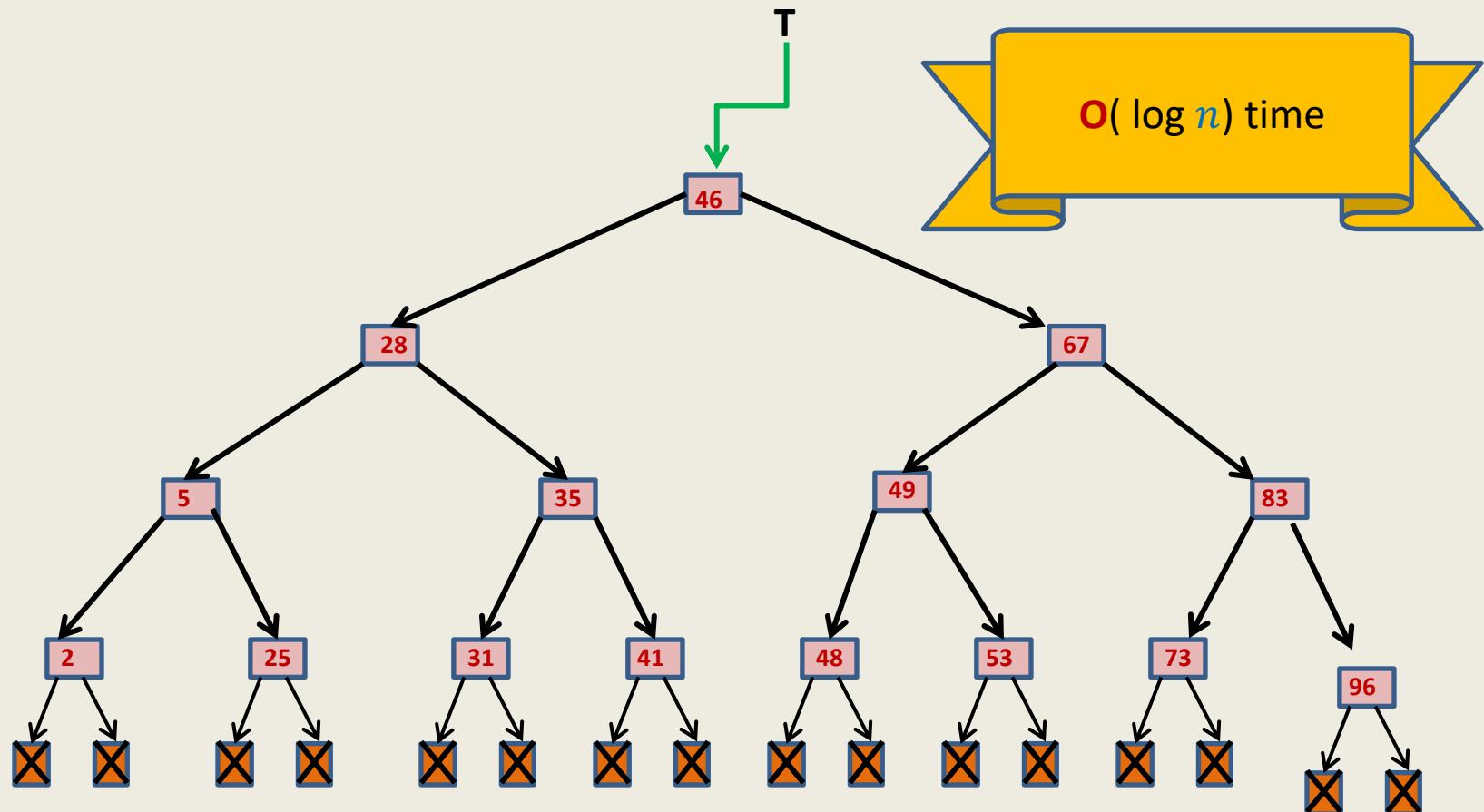
A fact we noticed in our previous discussion on BSTs (Lecture 9)

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

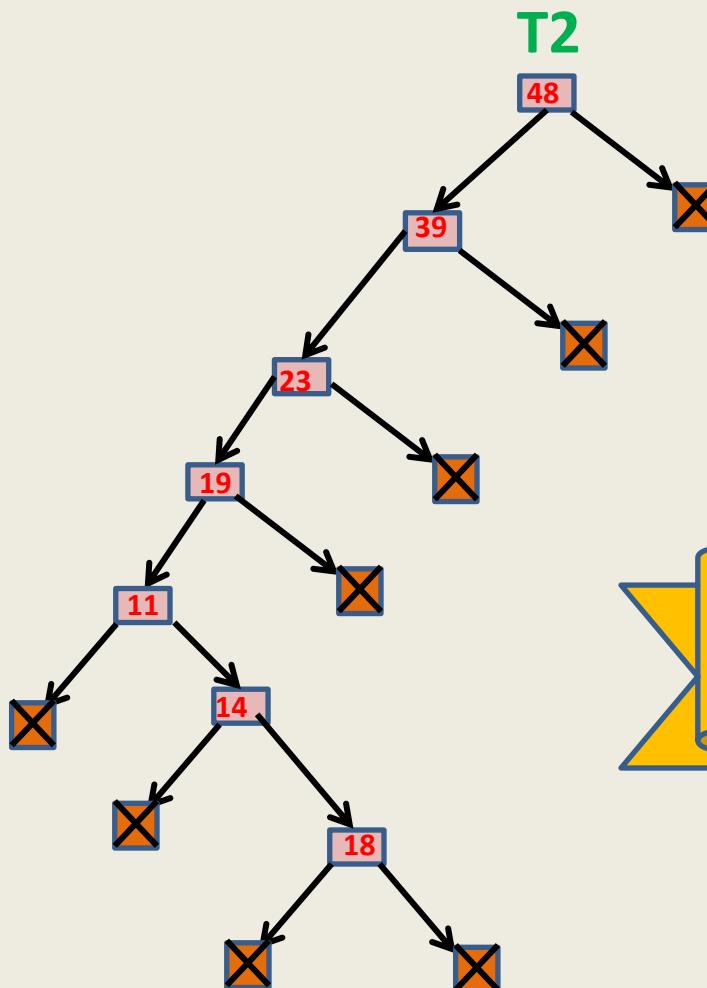
Height(T):

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

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

and

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

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

- $\text{Search}(T, x)$ operation is the same.
- Modify $\text{Insert}(T, x)$ operation as follows:
 - Carry out normal insert and update the **size** fields of nodes traversed.
 - If **BST** T ceases to be **nearly balanced** 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 in
a subsequent lecture.

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

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

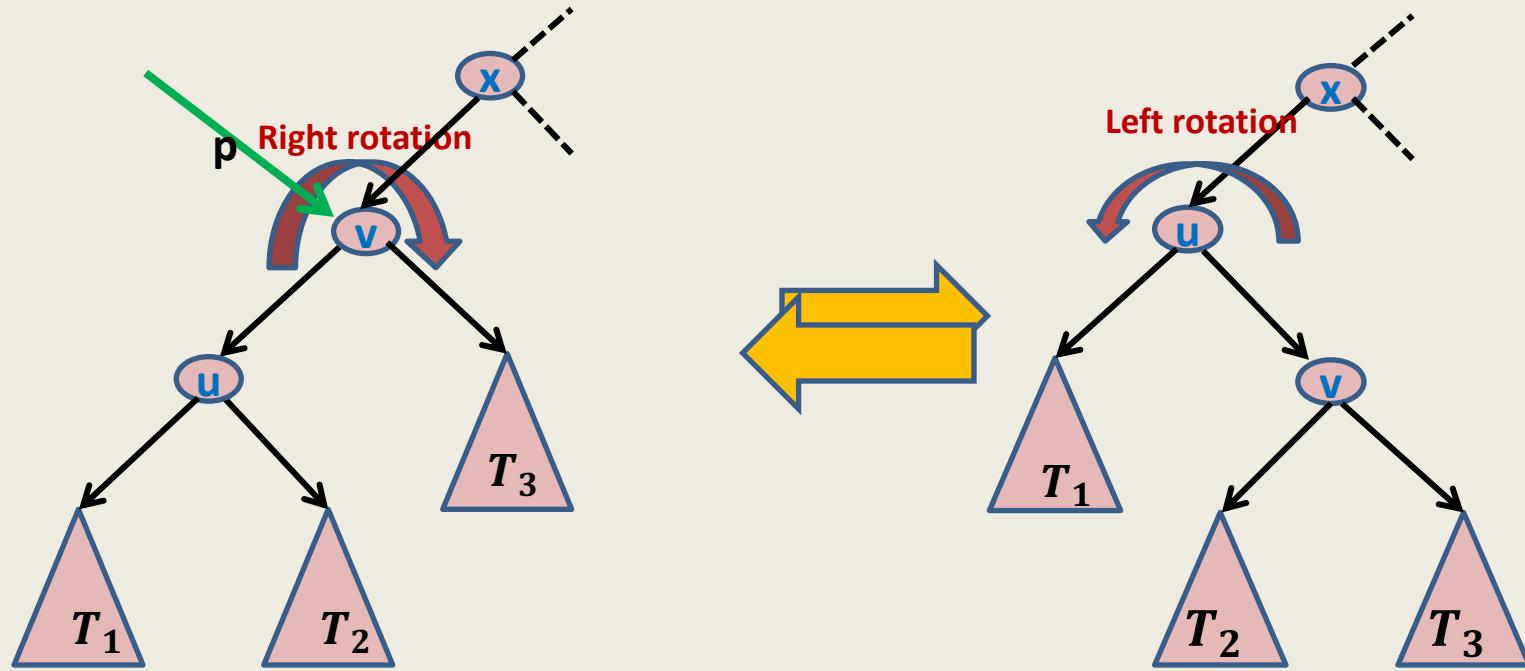
Rotation around a node

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

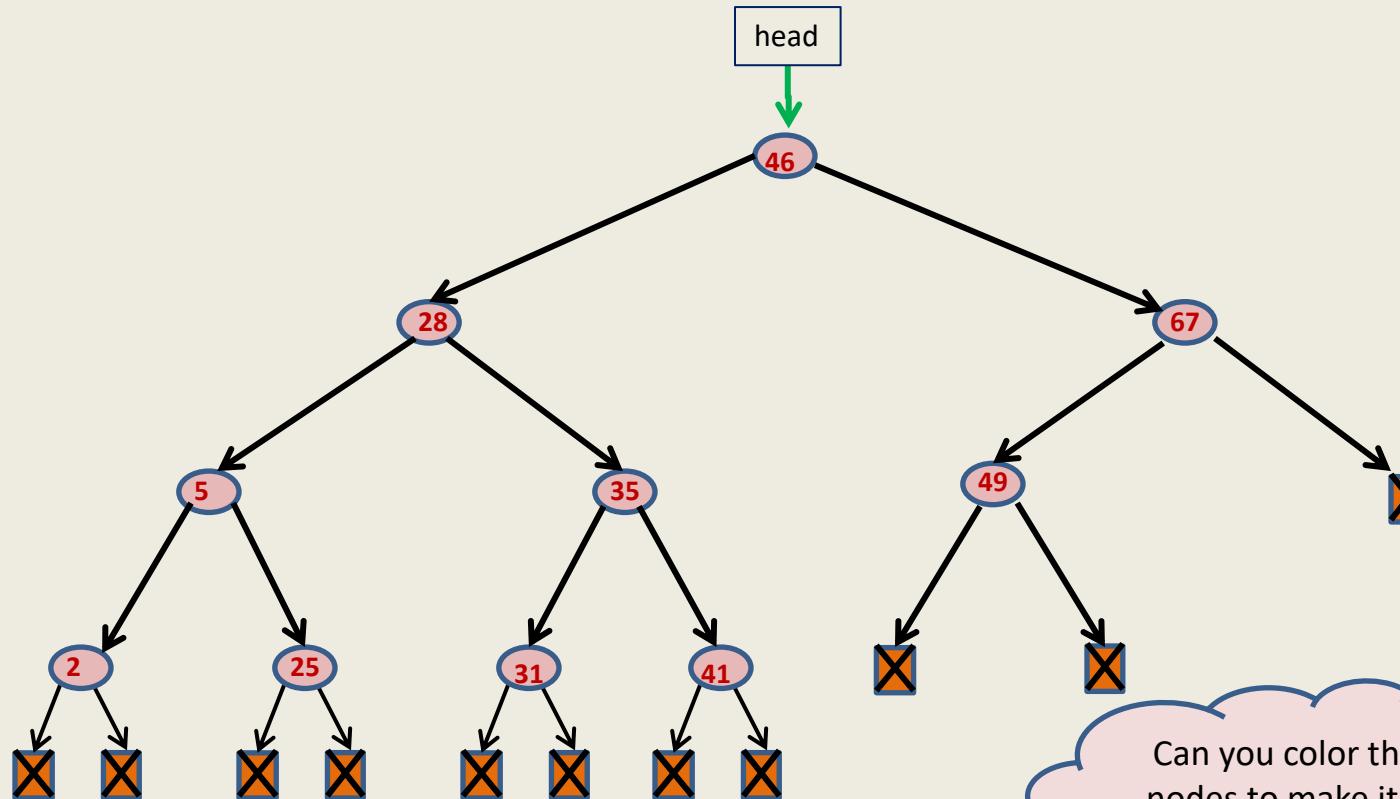
Red Black Tree

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.

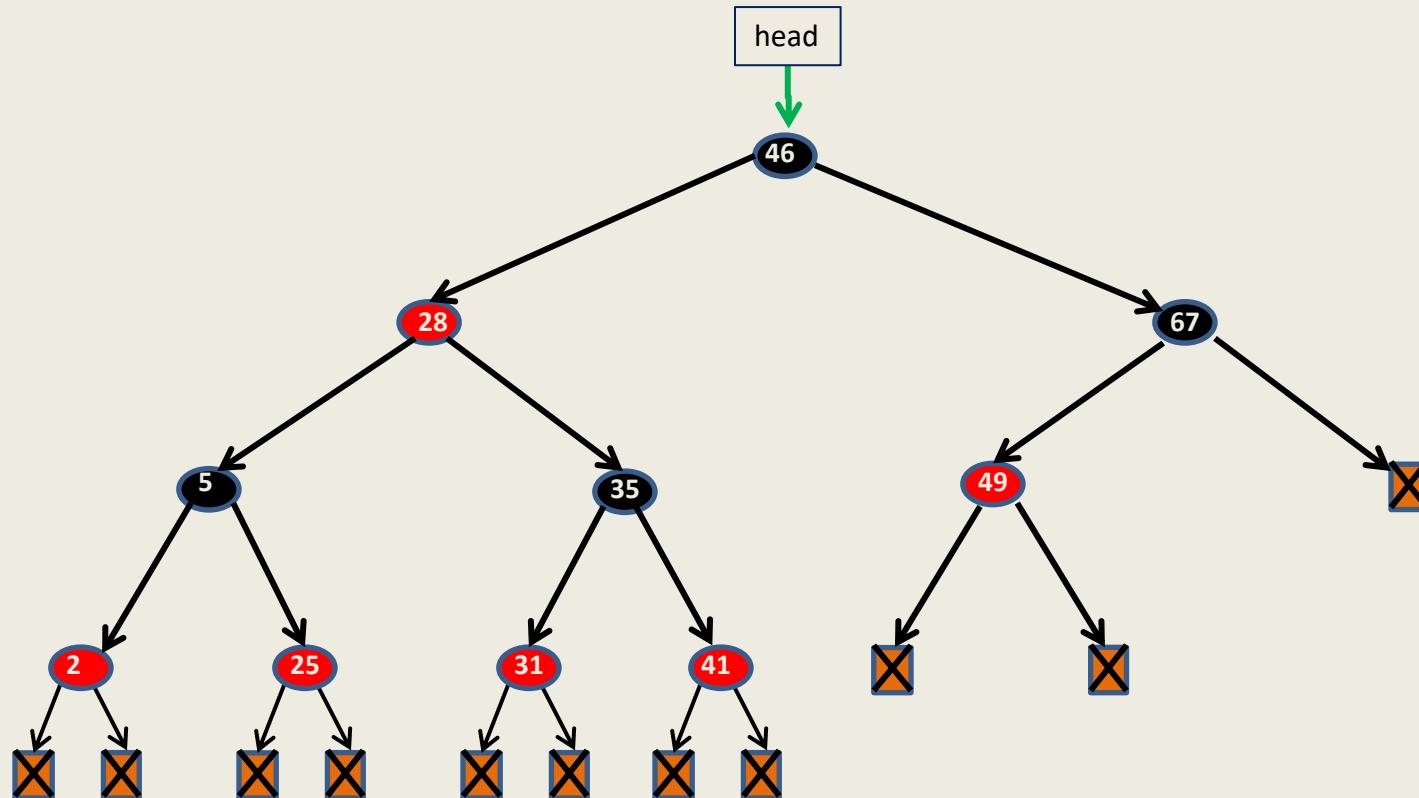
black height

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 ?

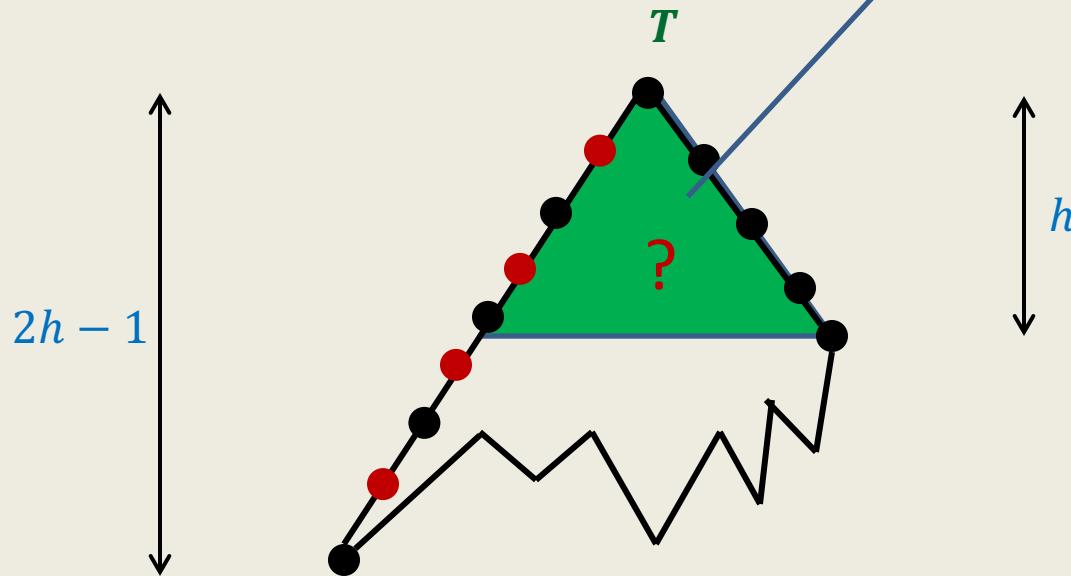
T : a red black tree

h : black height of T .

Question: What can be height of T ?

Answer: $\leq 2h - 1$

What is this “green structure” ?



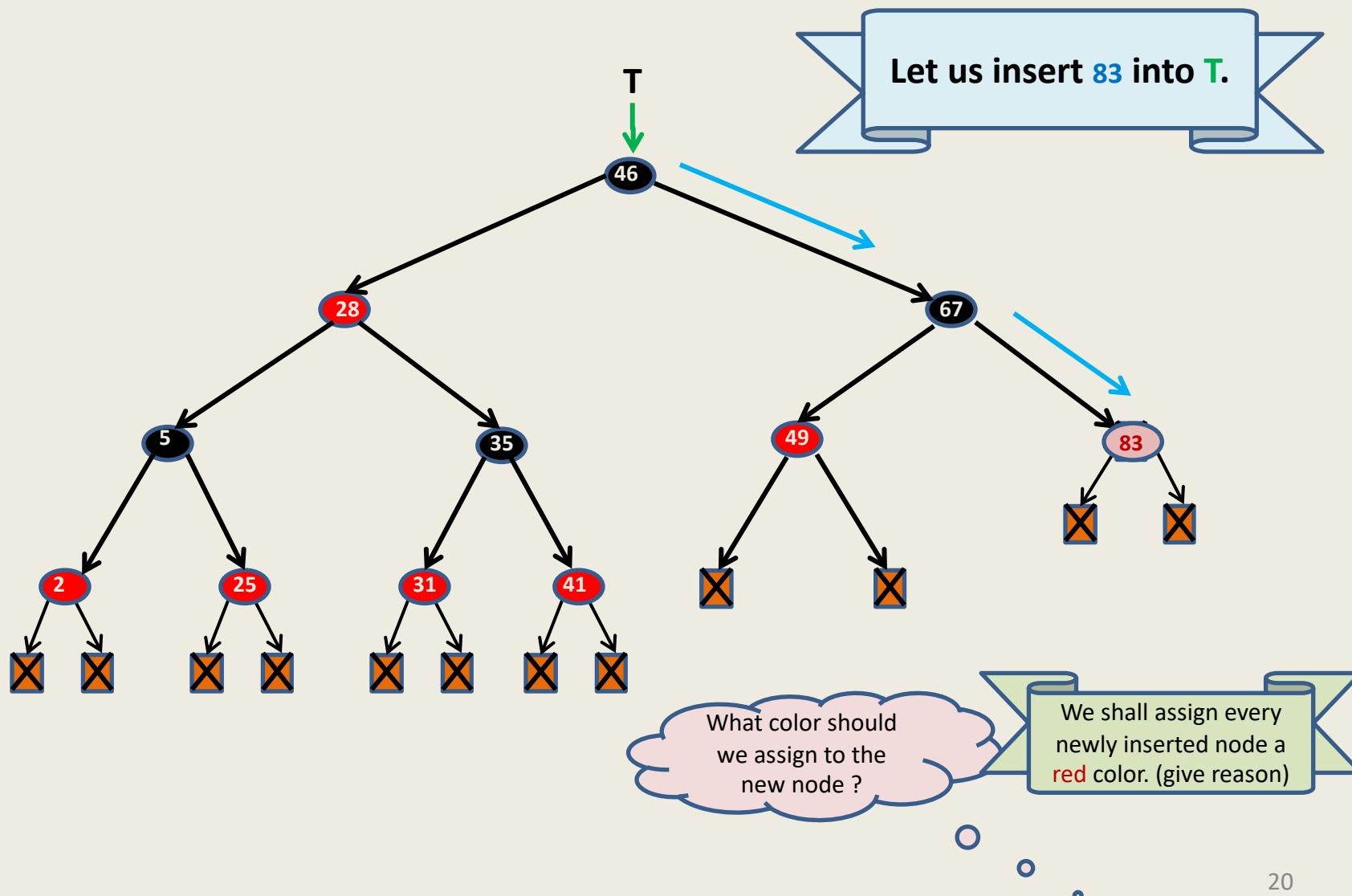
Homework: Ponder over the above hint to prove that T has $\geq 2^h - 1$ elements.¹⁸

Insertion in a Red Black tree

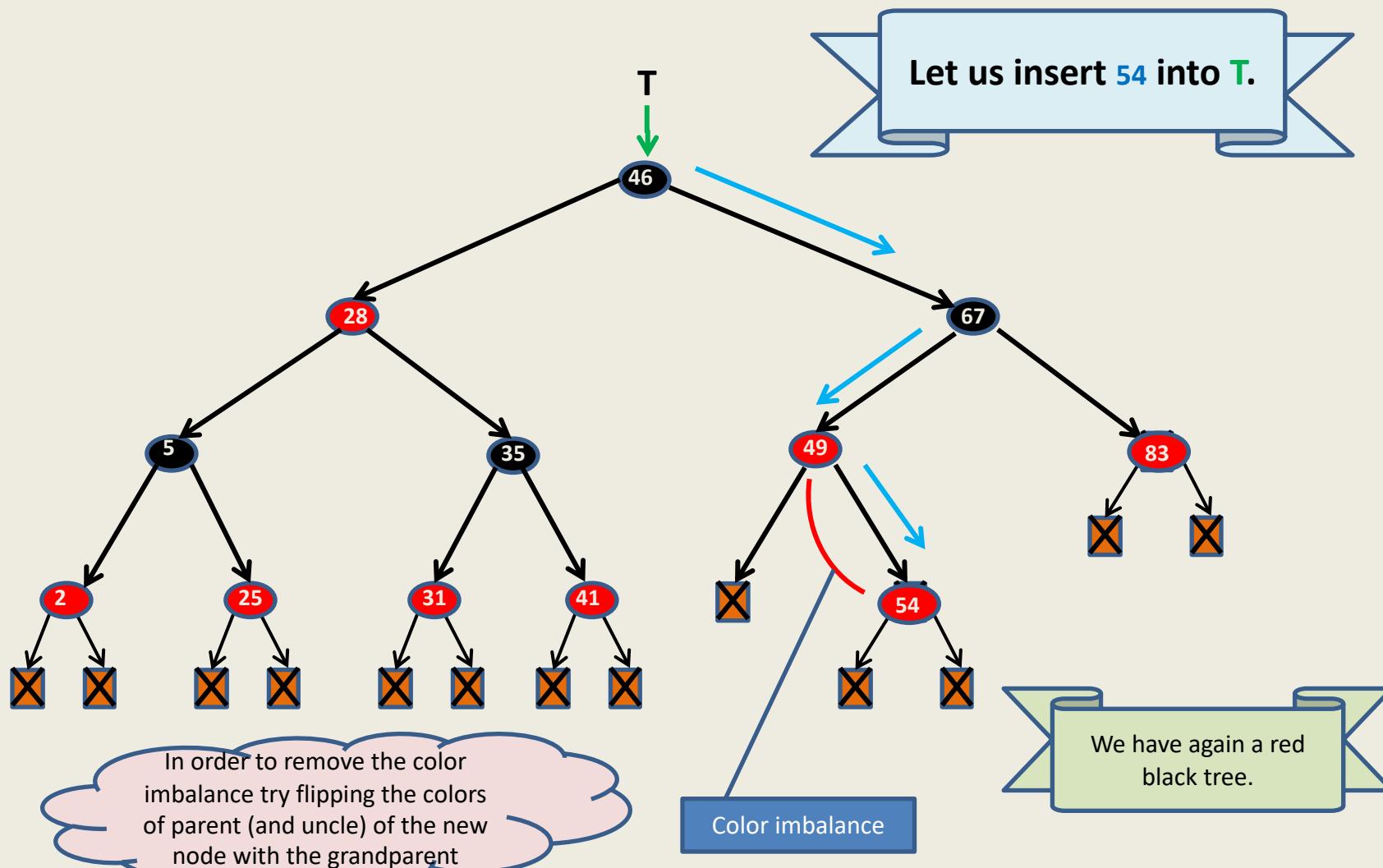
All it involves is

- playing with **colors** 😊
- and **rotations** 😊

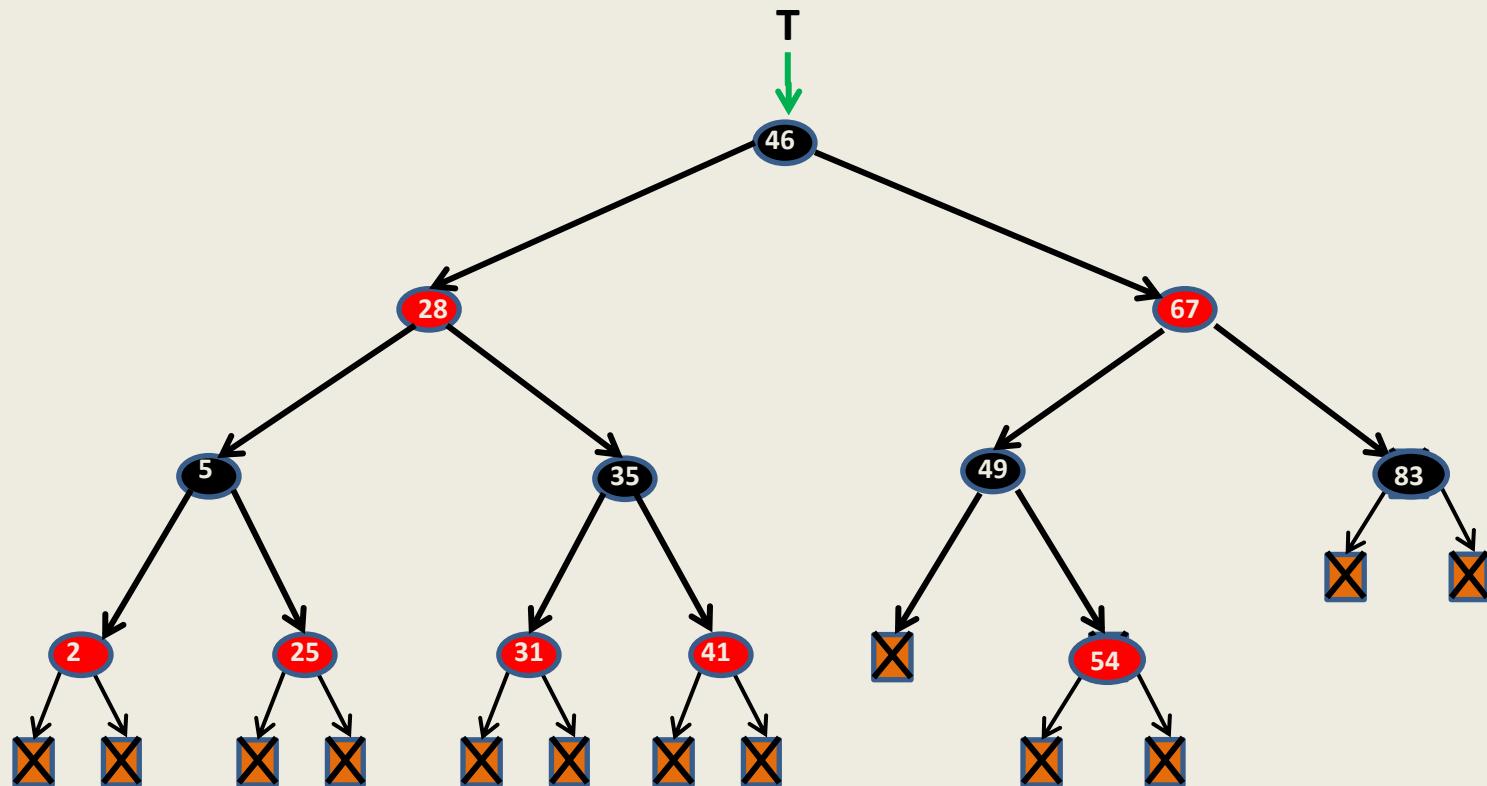
Insertion in a red-black tree



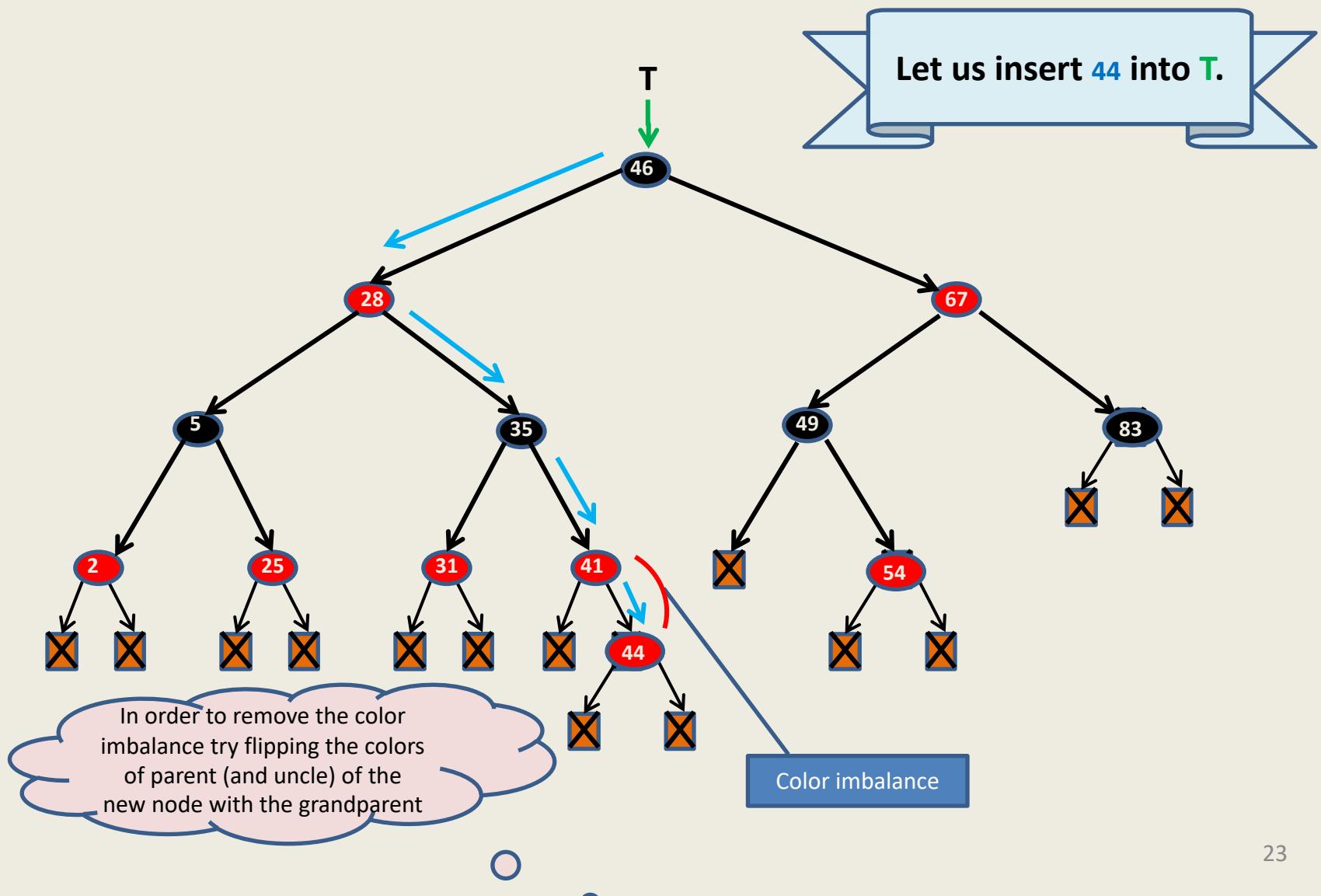
Insertion in a red-black tree



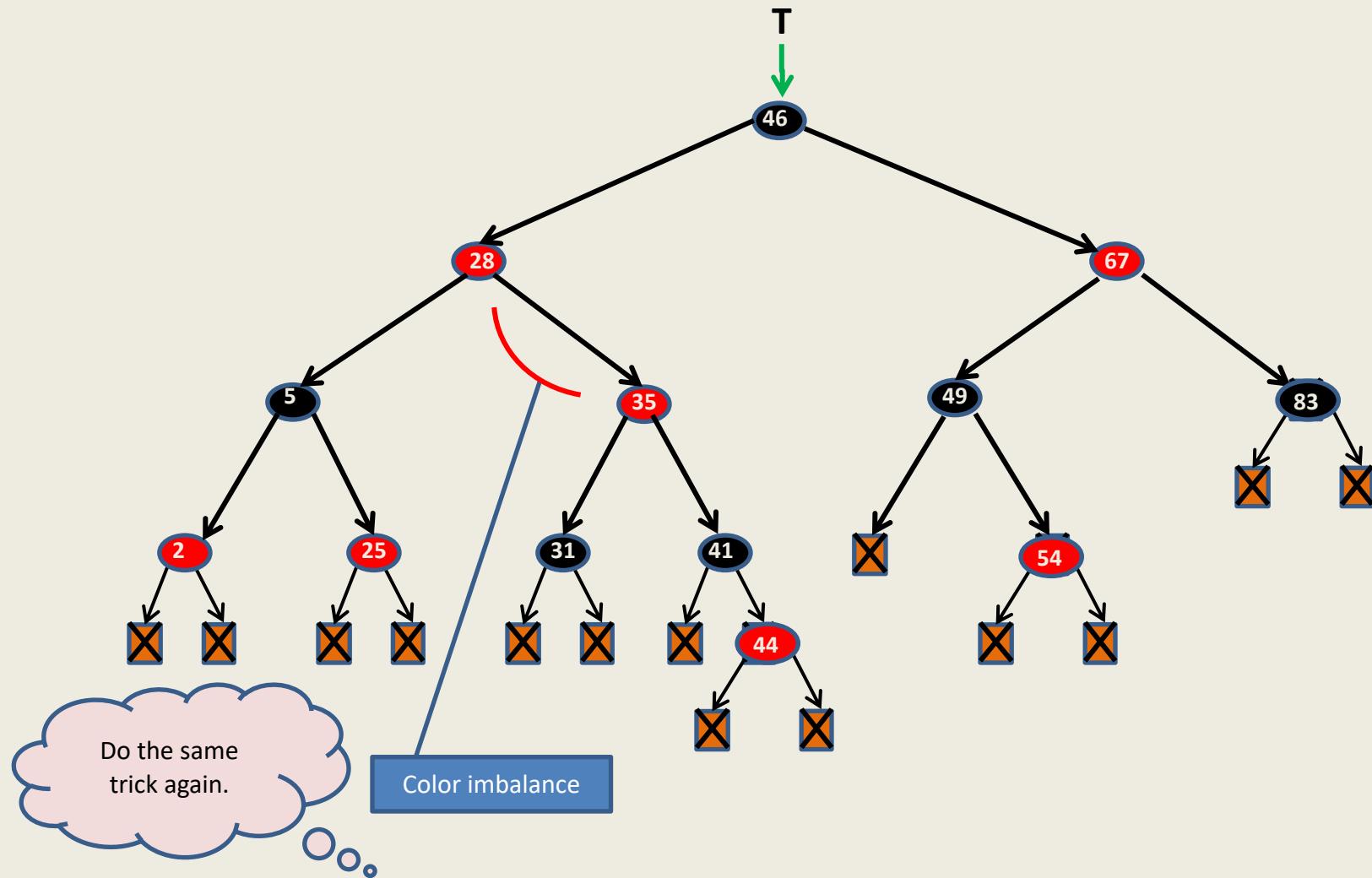
Insertion in a red-black tree



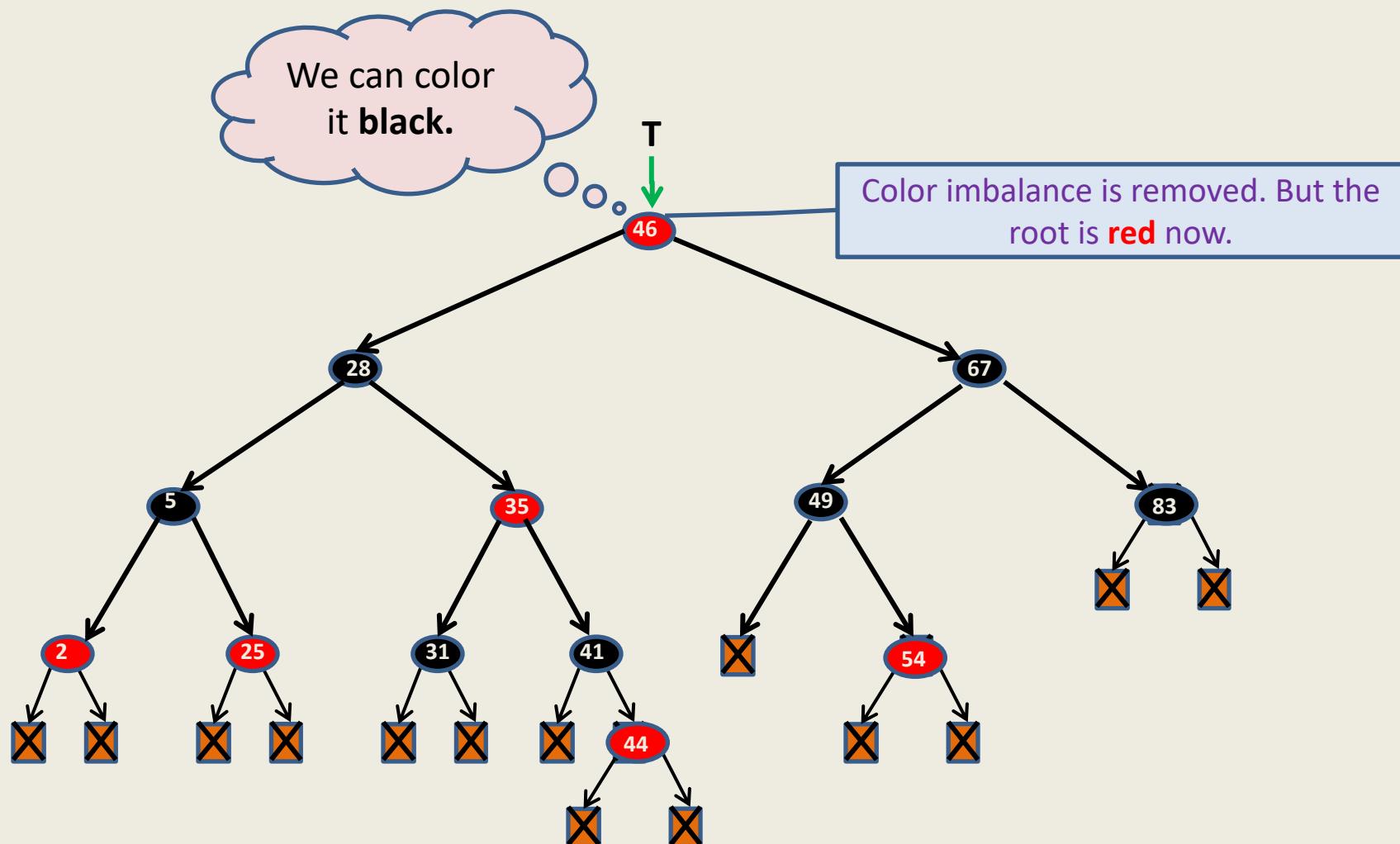
Insertion in a red-black tree



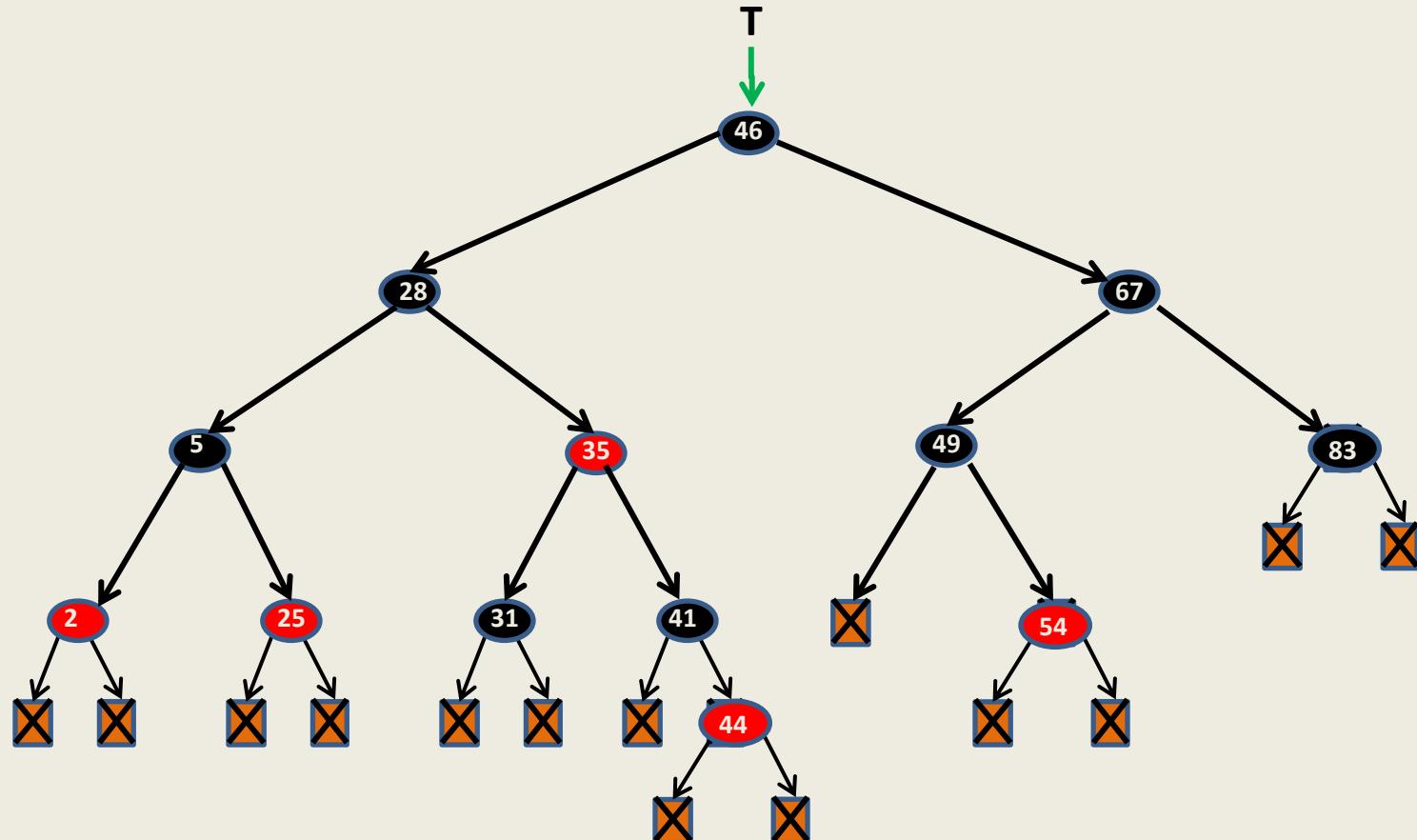
Insertion in a red-black tree



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 **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

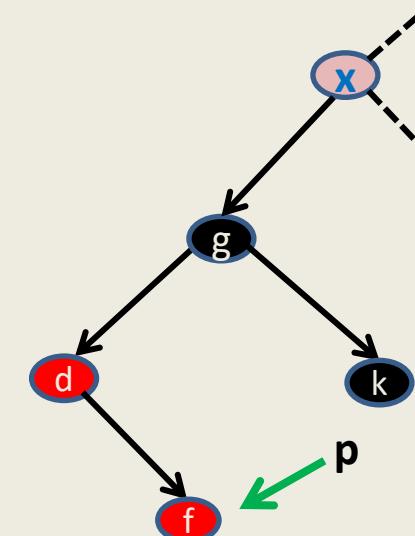
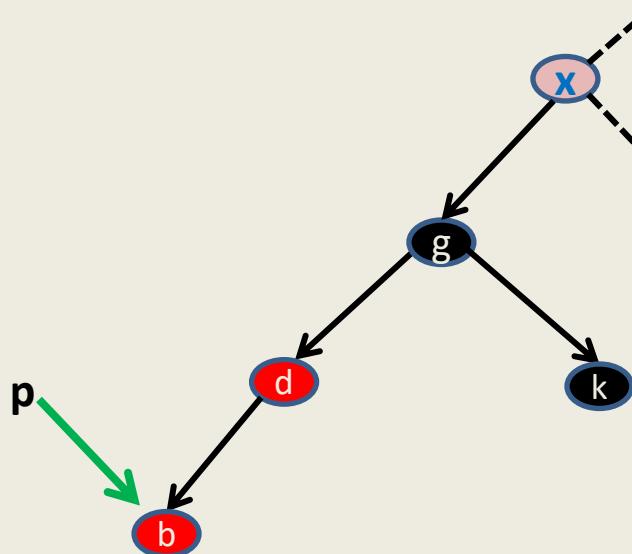
- p is a **red** colored node.
- $\text{parent}(p)$ is also **red**.
- $\text{uncle}(p)$ is **black**.

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



Case 3.1:

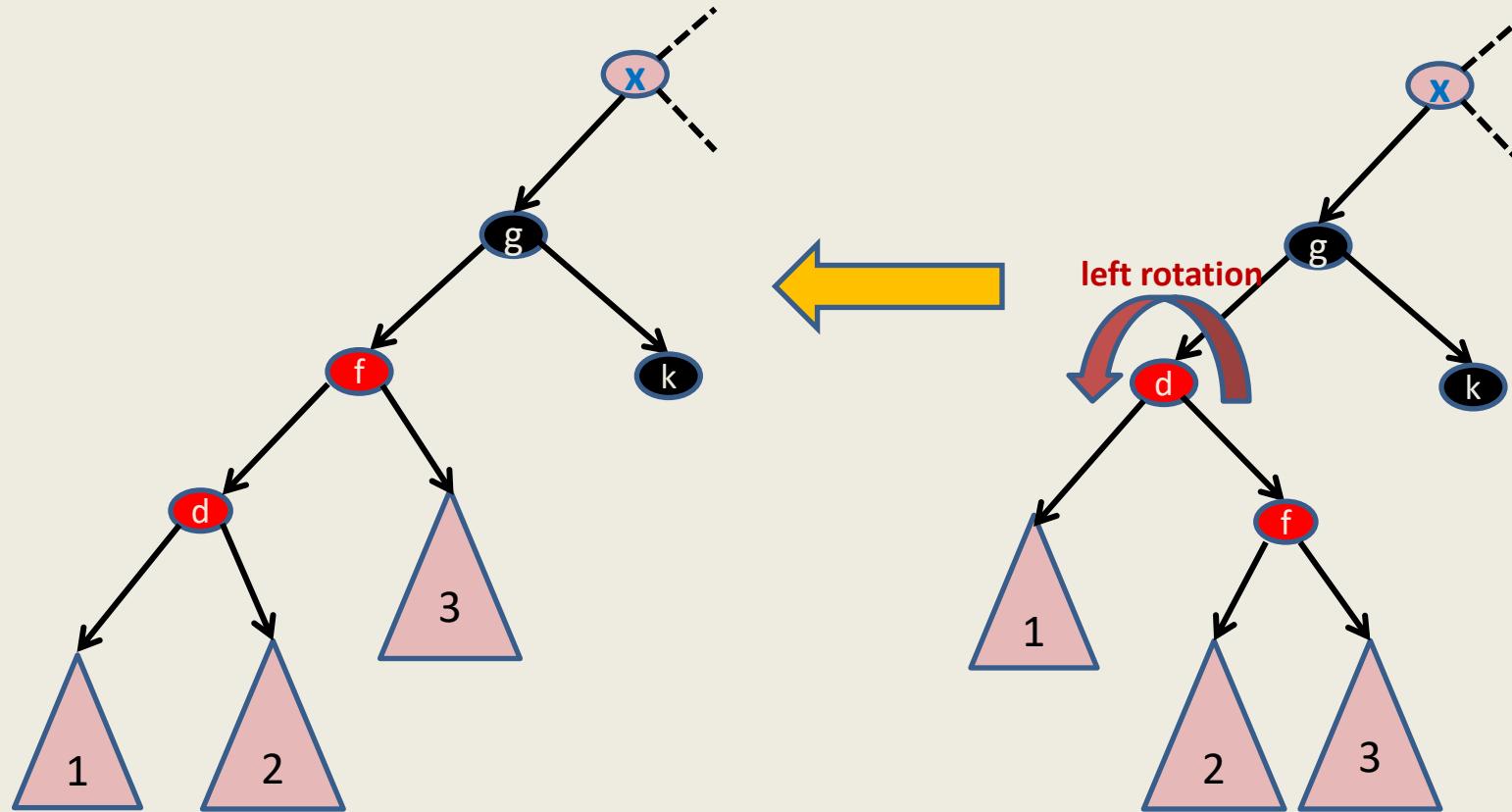
p is **left child** of its parent

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

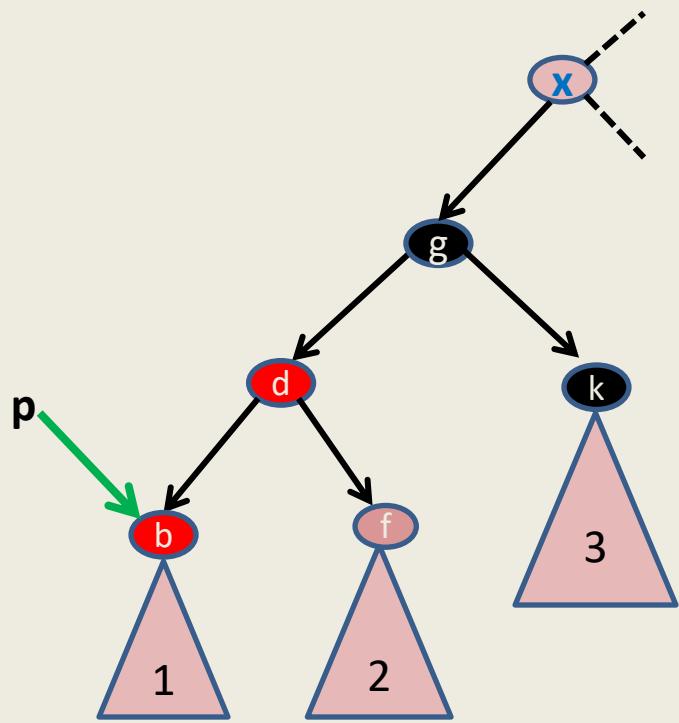


Vow!
This is exactly **Case 3.1**

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

We need to handle only case 3.1

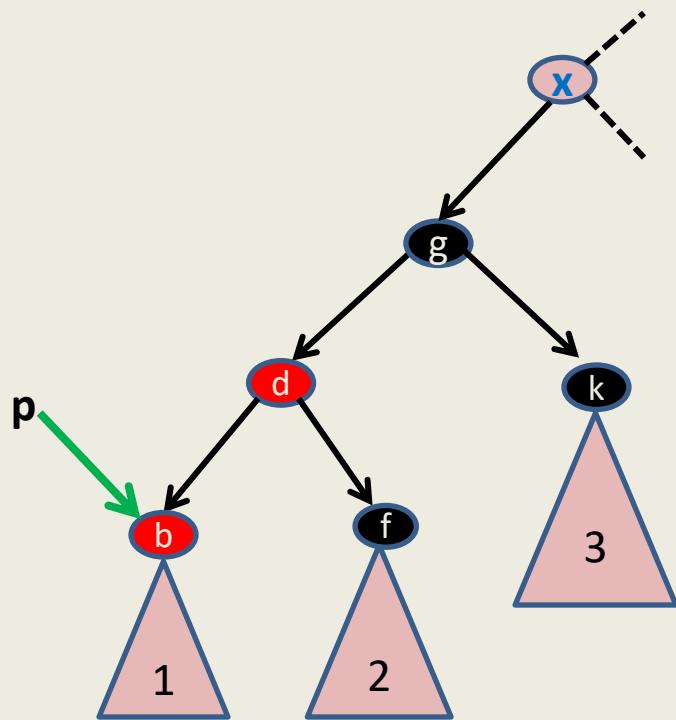
Handling the case 3.1



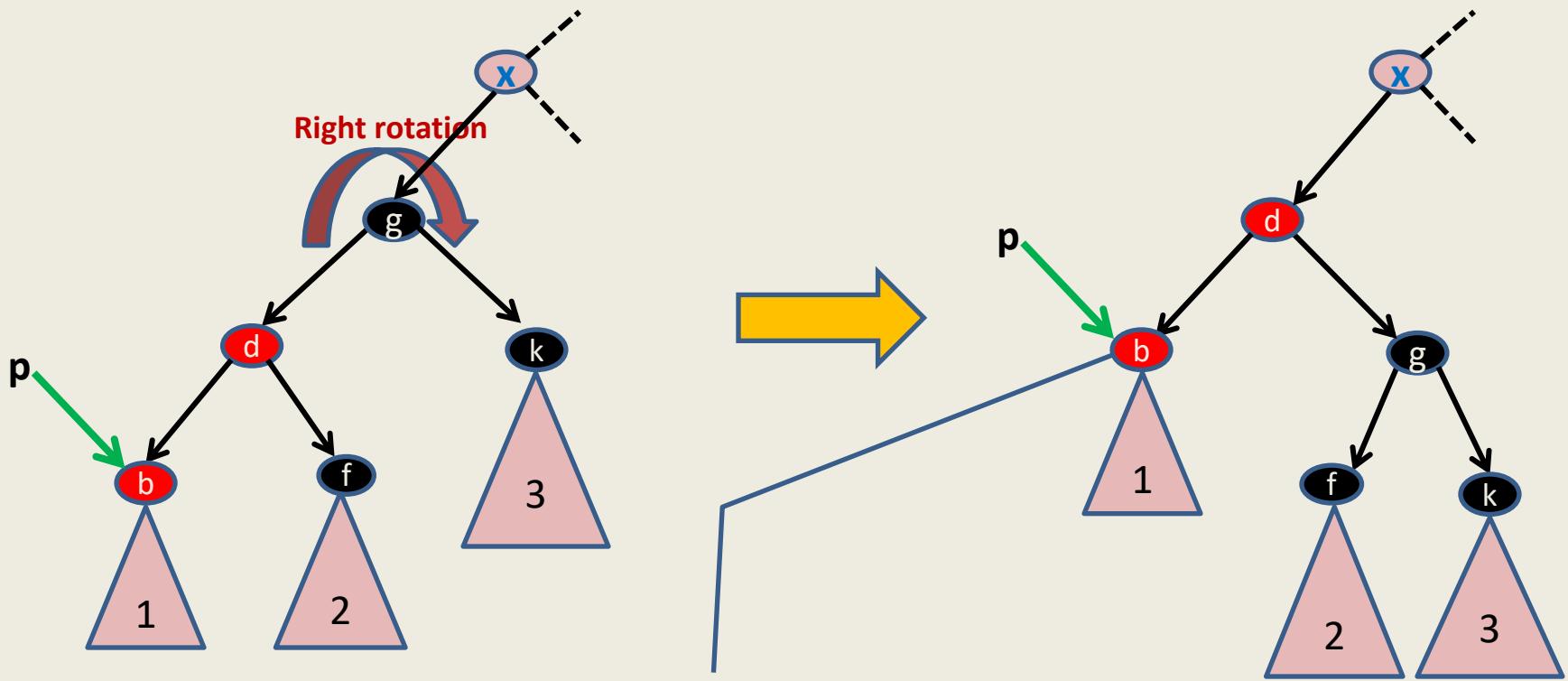
Can we say
anything about
color of node f ?

black

Handling the case 3.1



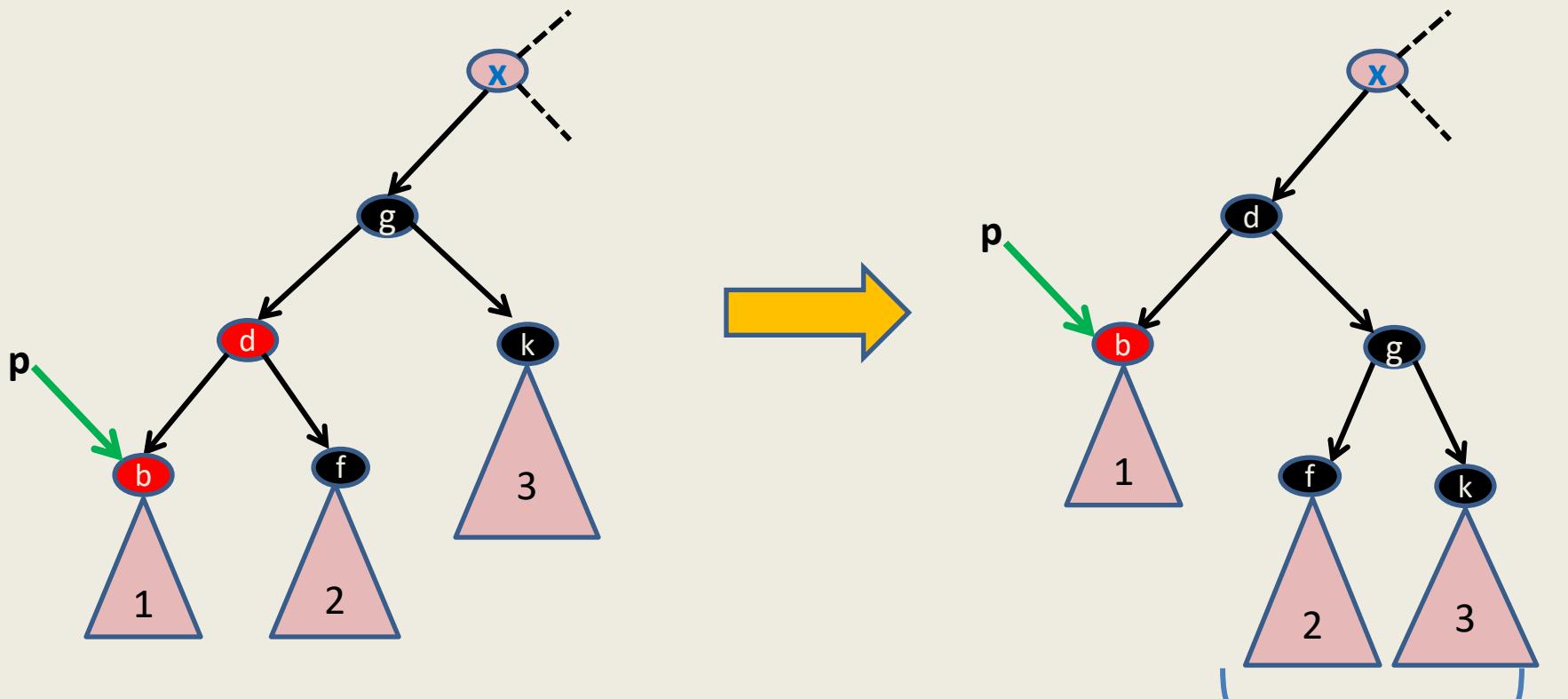
Handling the case 3.1



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**

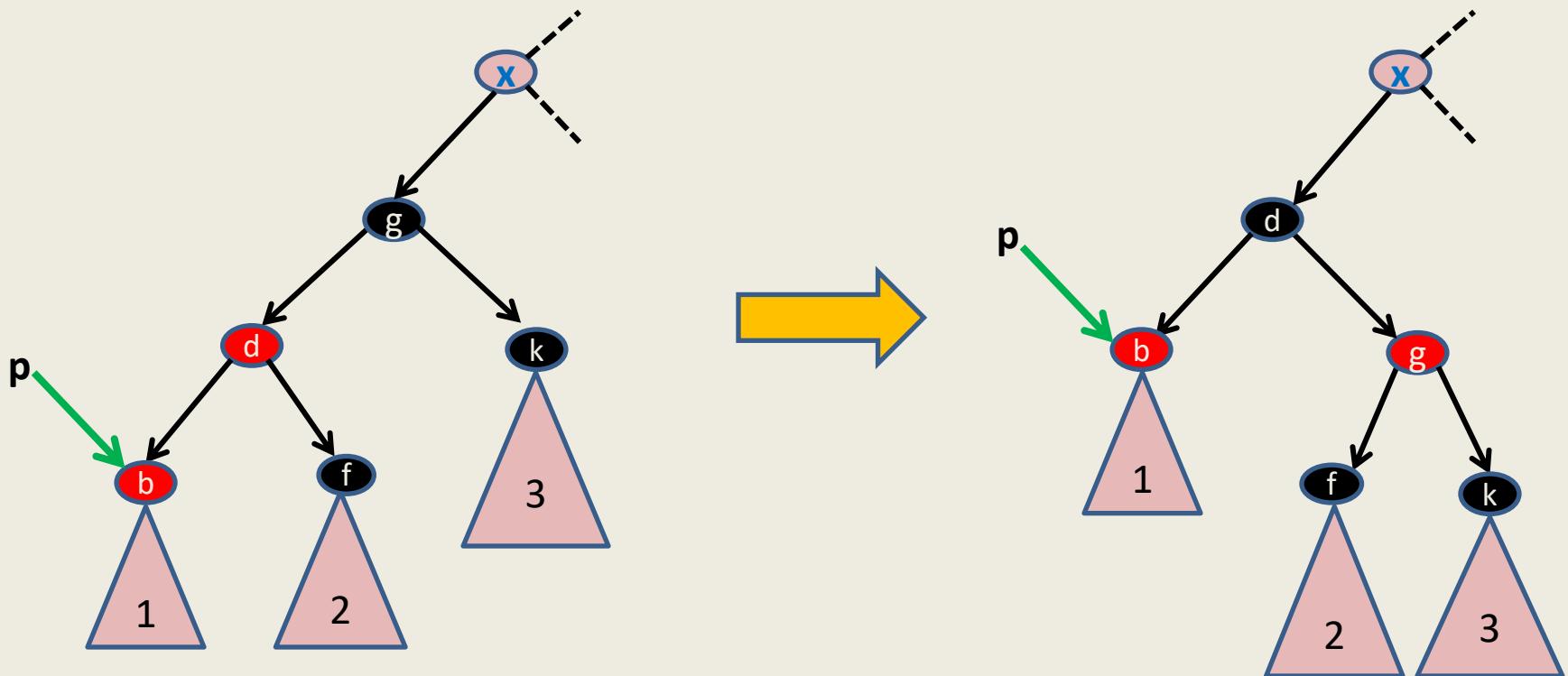
Handling the case 3.1



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 $\mathbf{O}(\log n)$ 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.

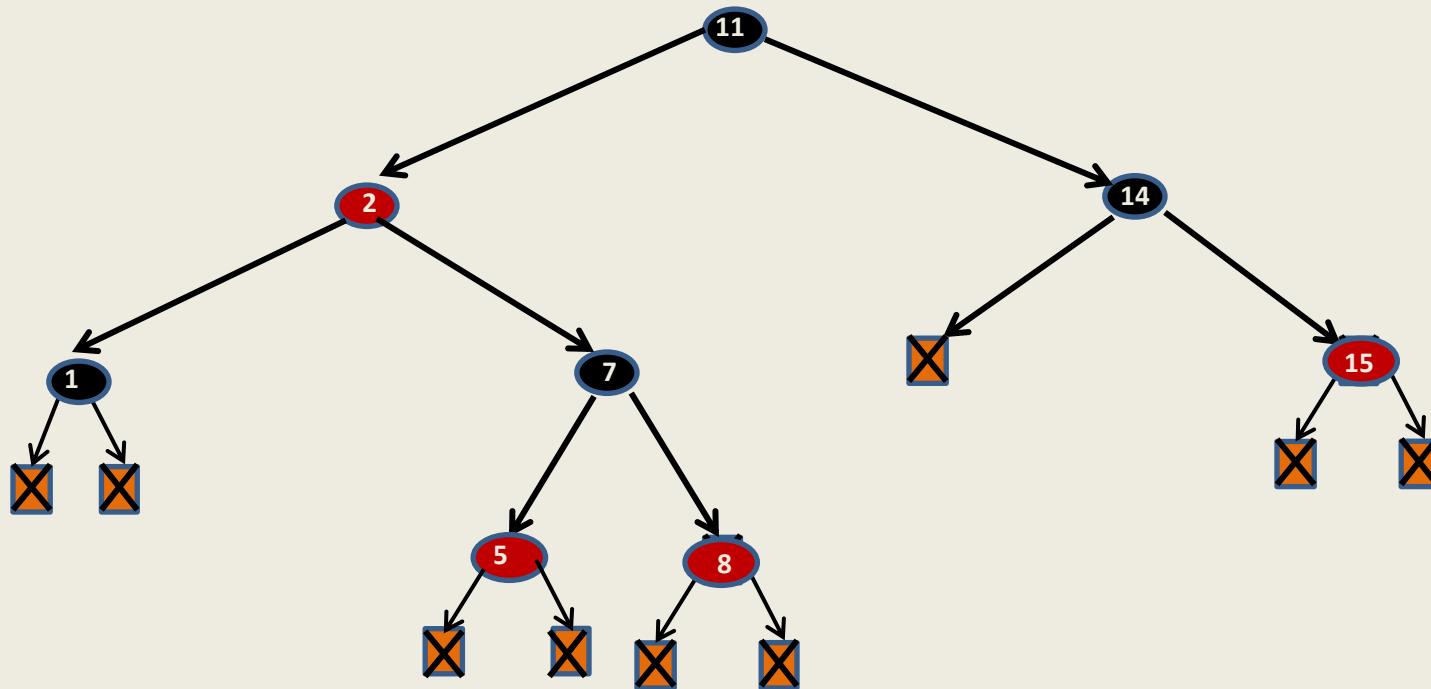
- Why we are handling insertions in “this *particular way*” ?
- Are there *alternative and simpler* ways to handle insertions ?

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 ?

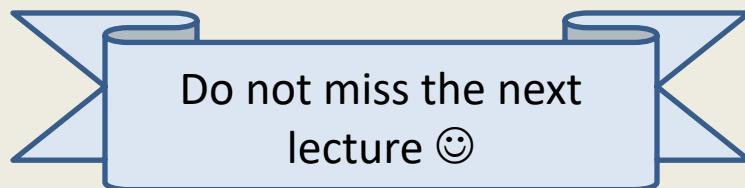


How do will we handle deletion ?

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 ☺



Data Structures and Algorithms

(ESO207)

Lecture 18:

Height balanced BST

- Red-black trees - II

Red Black Tree

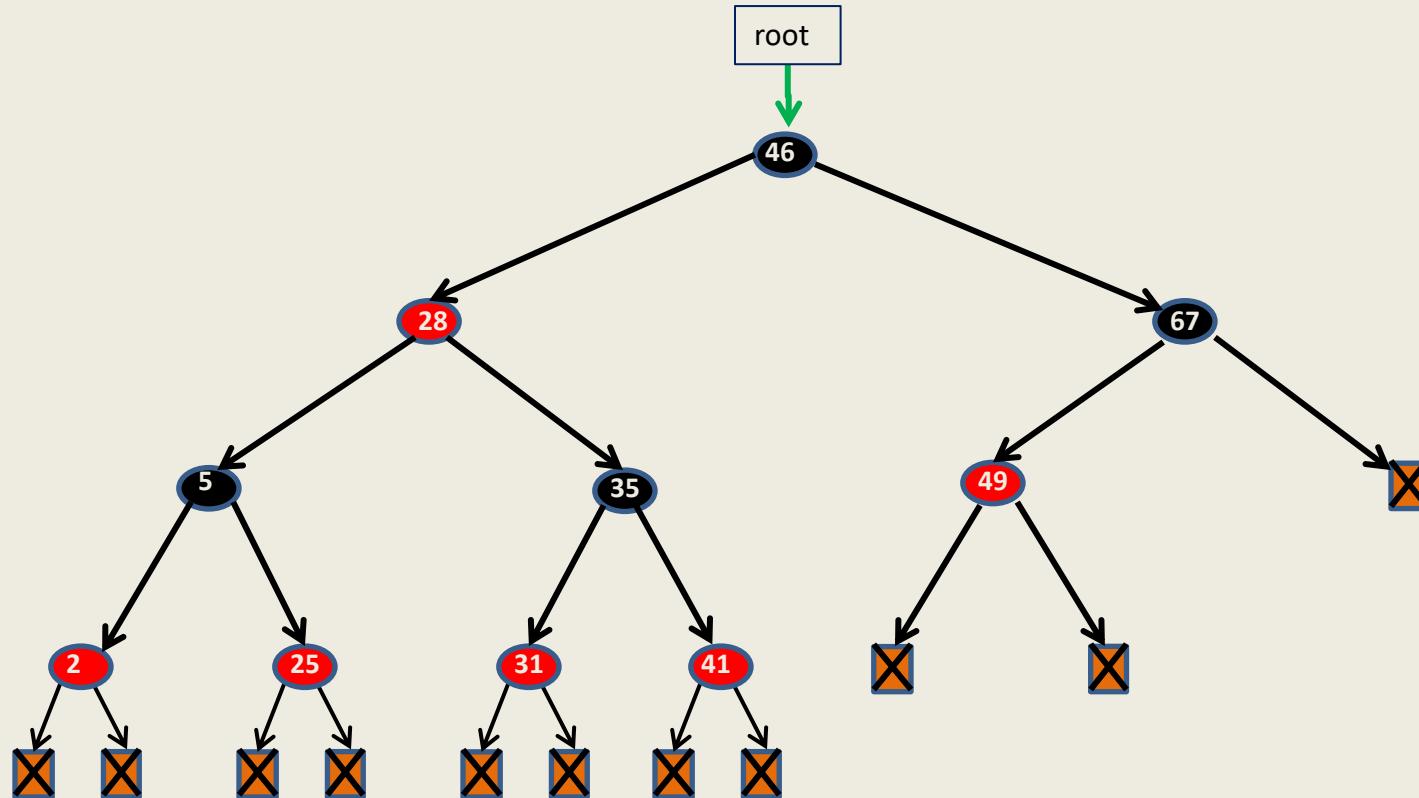
Red Black tree:

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



Handling Deletion in a Red Black Tree

Notations to be used



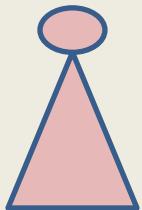
a **black** node



a **red** node



a node whose color is not specified



a BST

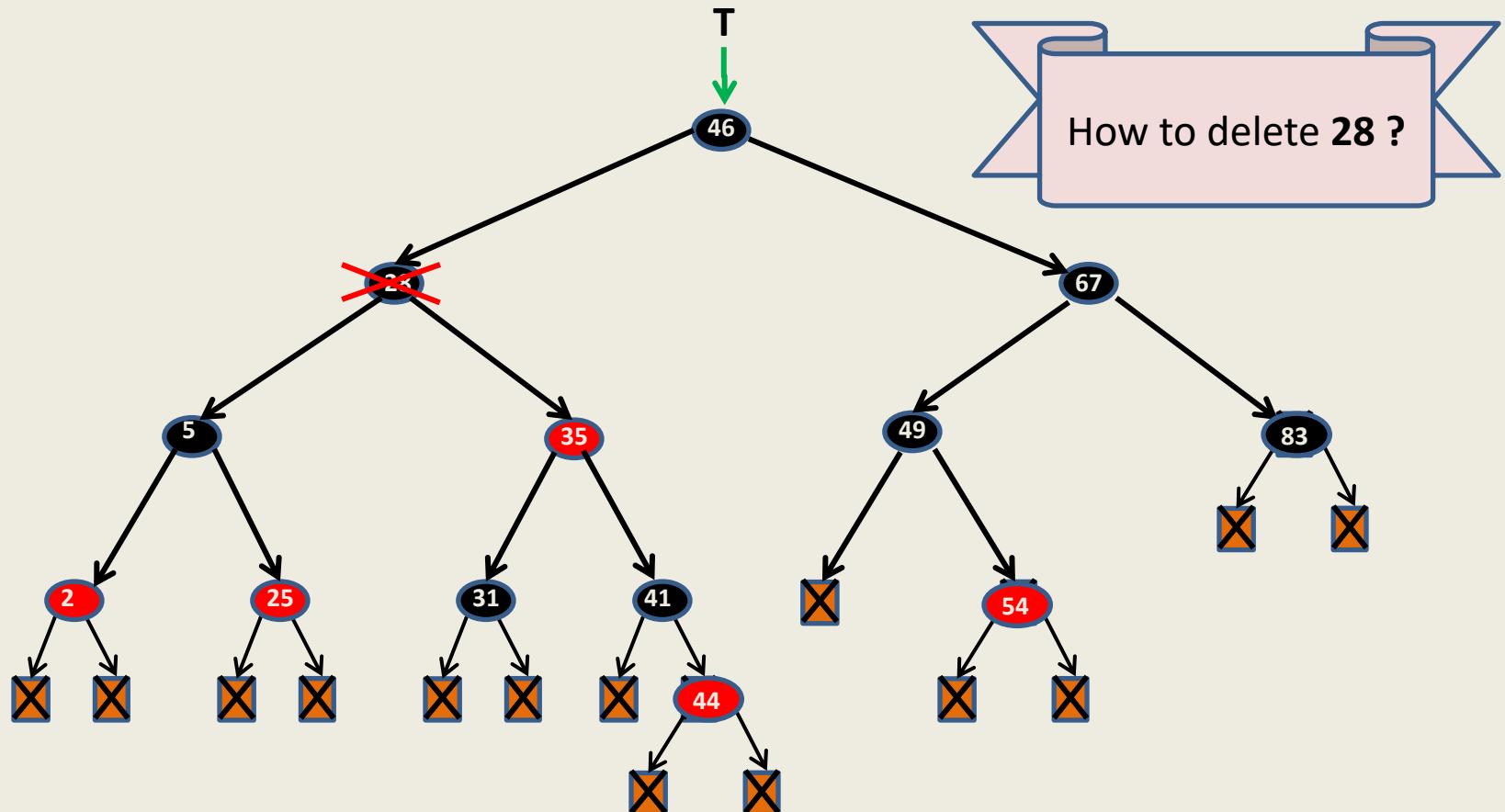


Could potentially be

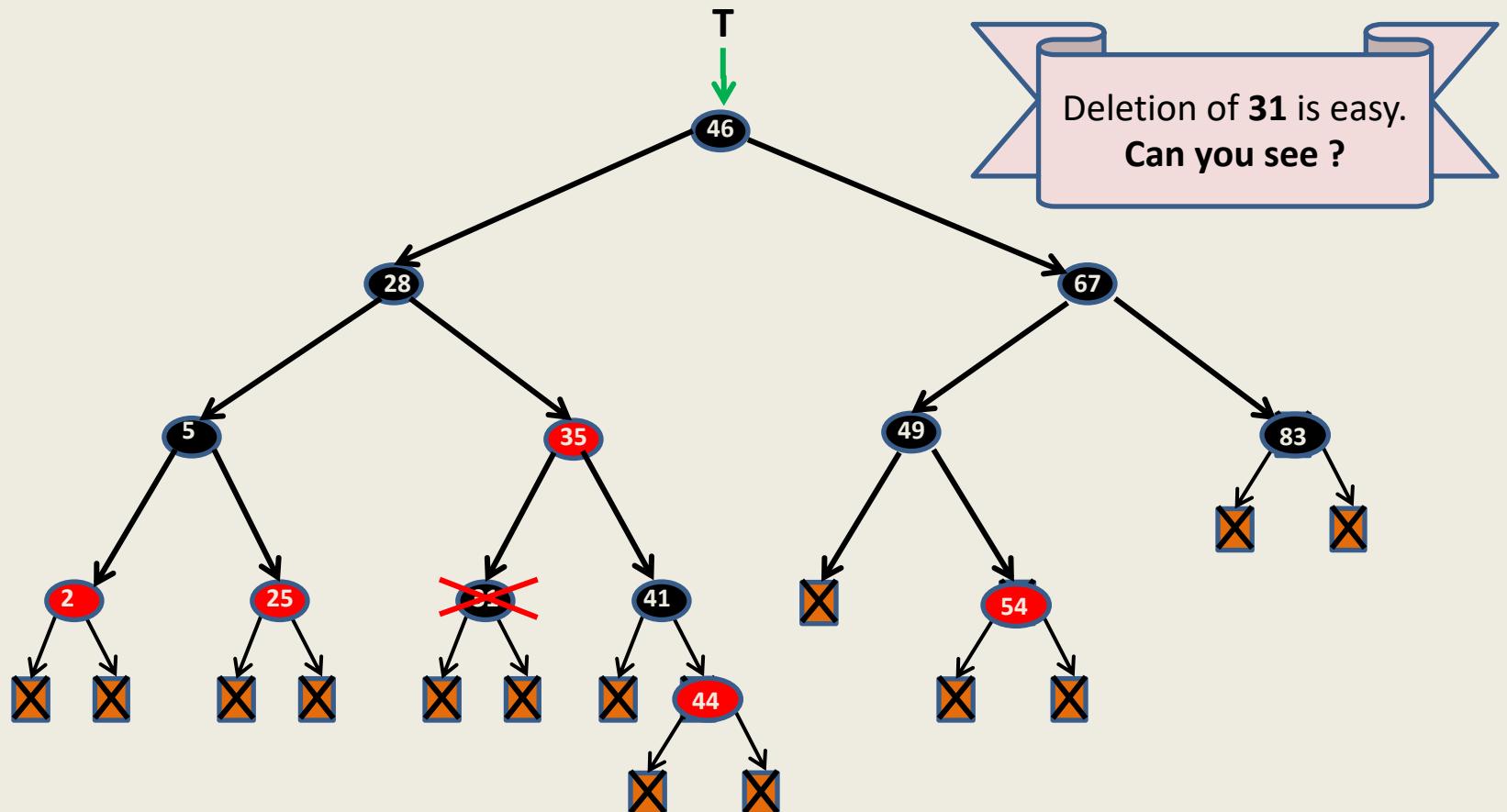


Deletion in a BST is **slightly harder than Insertion**

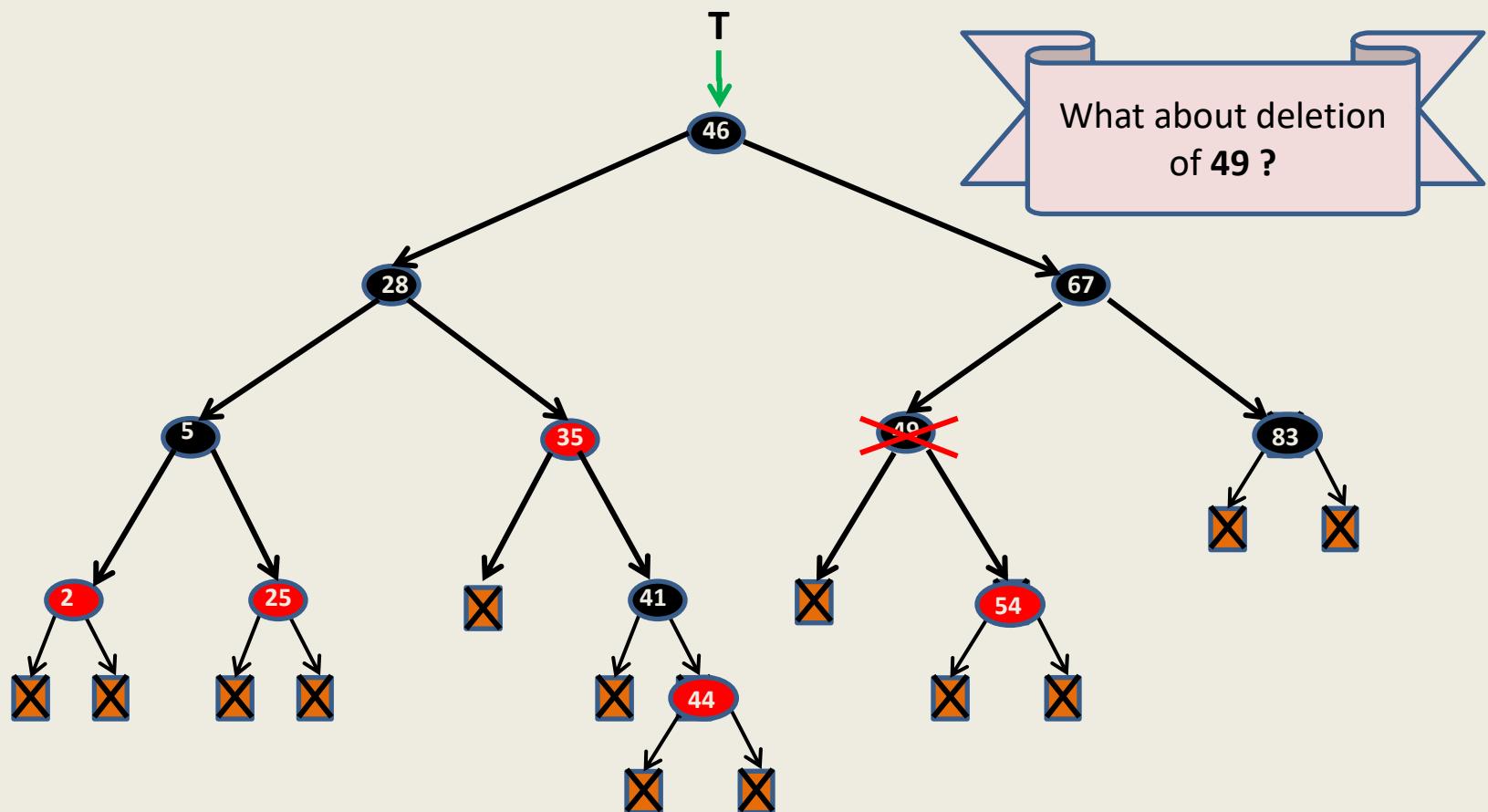
(even if we ignore the **height** factor)



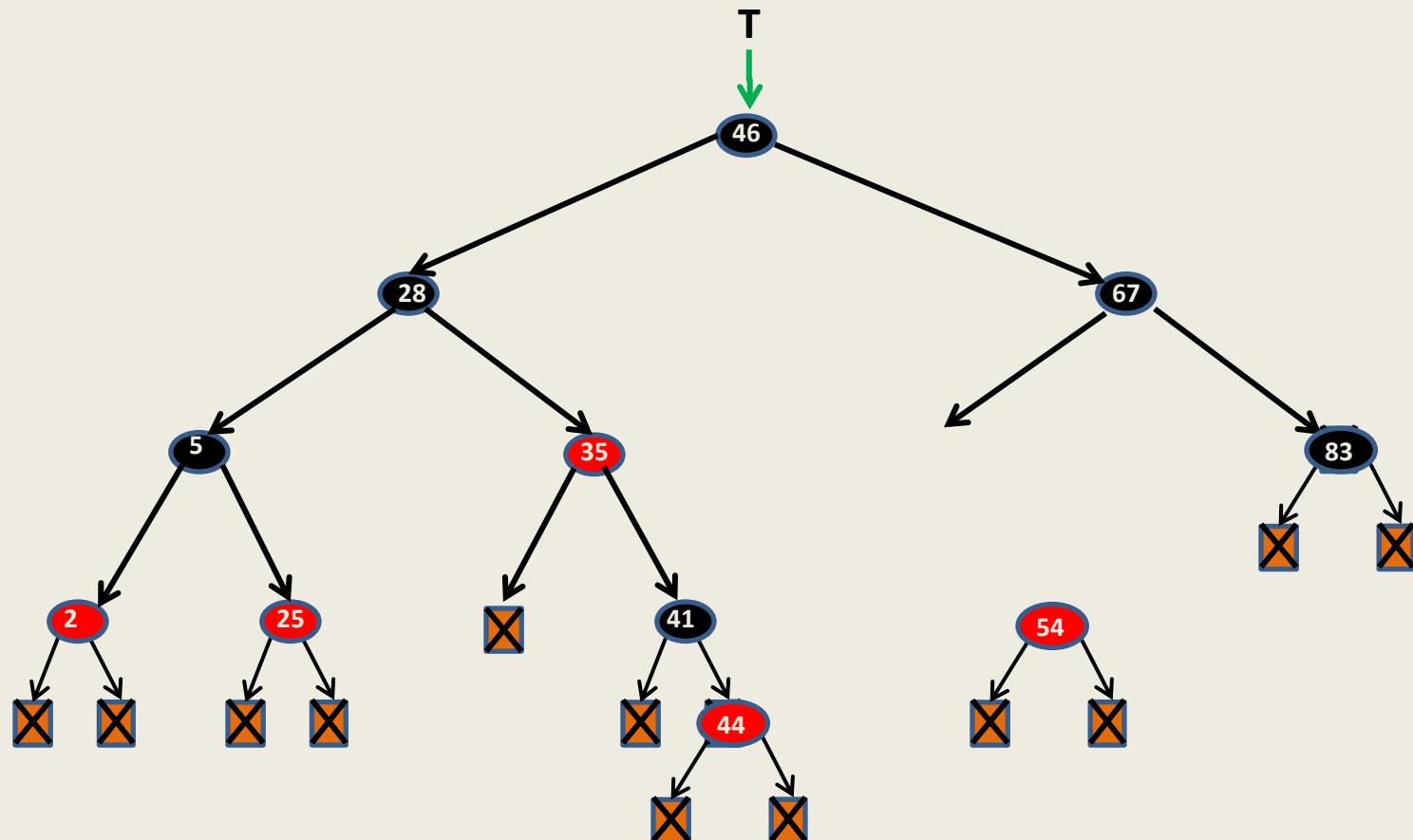
Is deletion of a node easier for some cases ?



Is deletion of a node easier for some cases ?

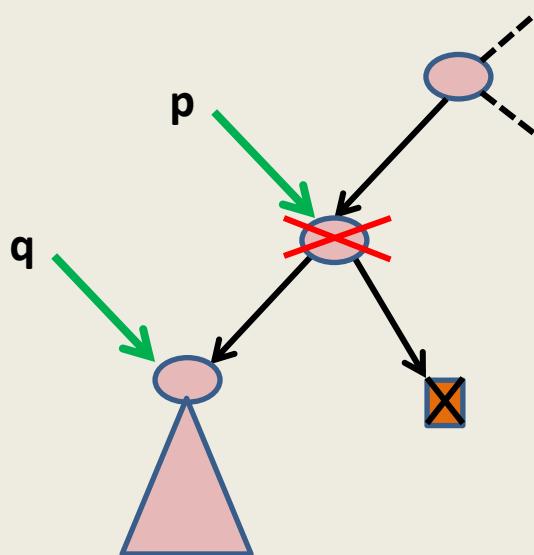


Is deletion of a node easier for some cases ?



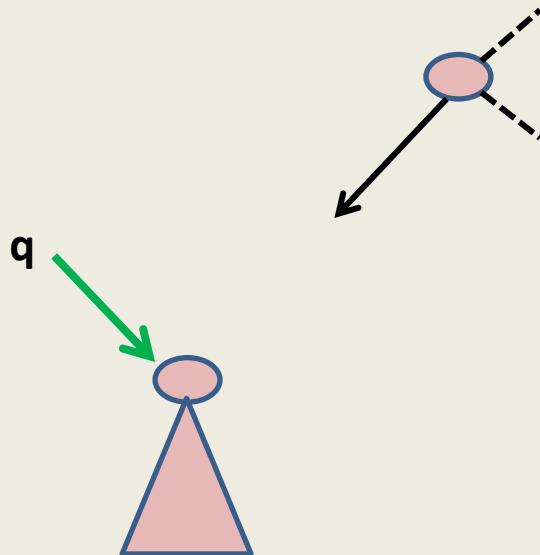
An insight

It is easier to maintain a BST under deletion if the node to be deleted has at most one child which is non-leaf.



An insight

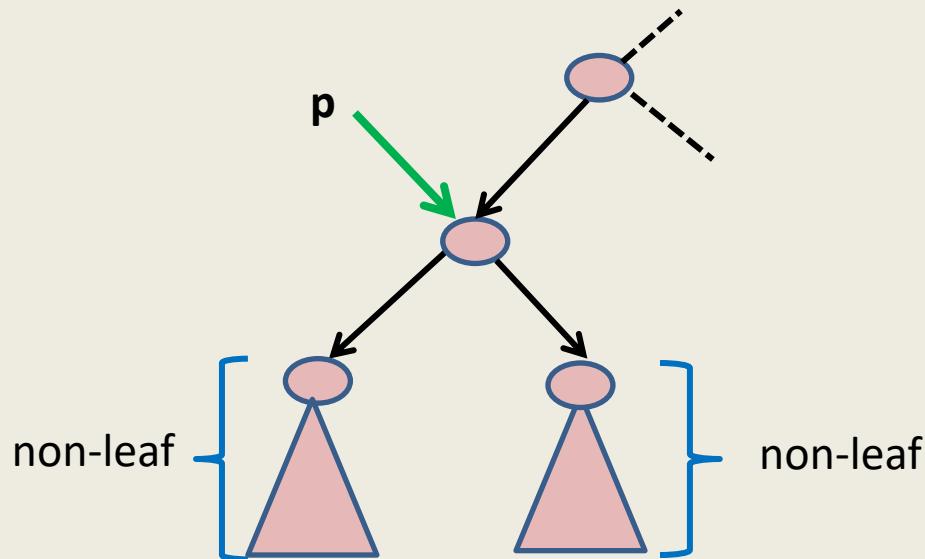
It is easier to maintain a BST under deletion if the node to be deleted has at most one child which is non-leaf.



An important question

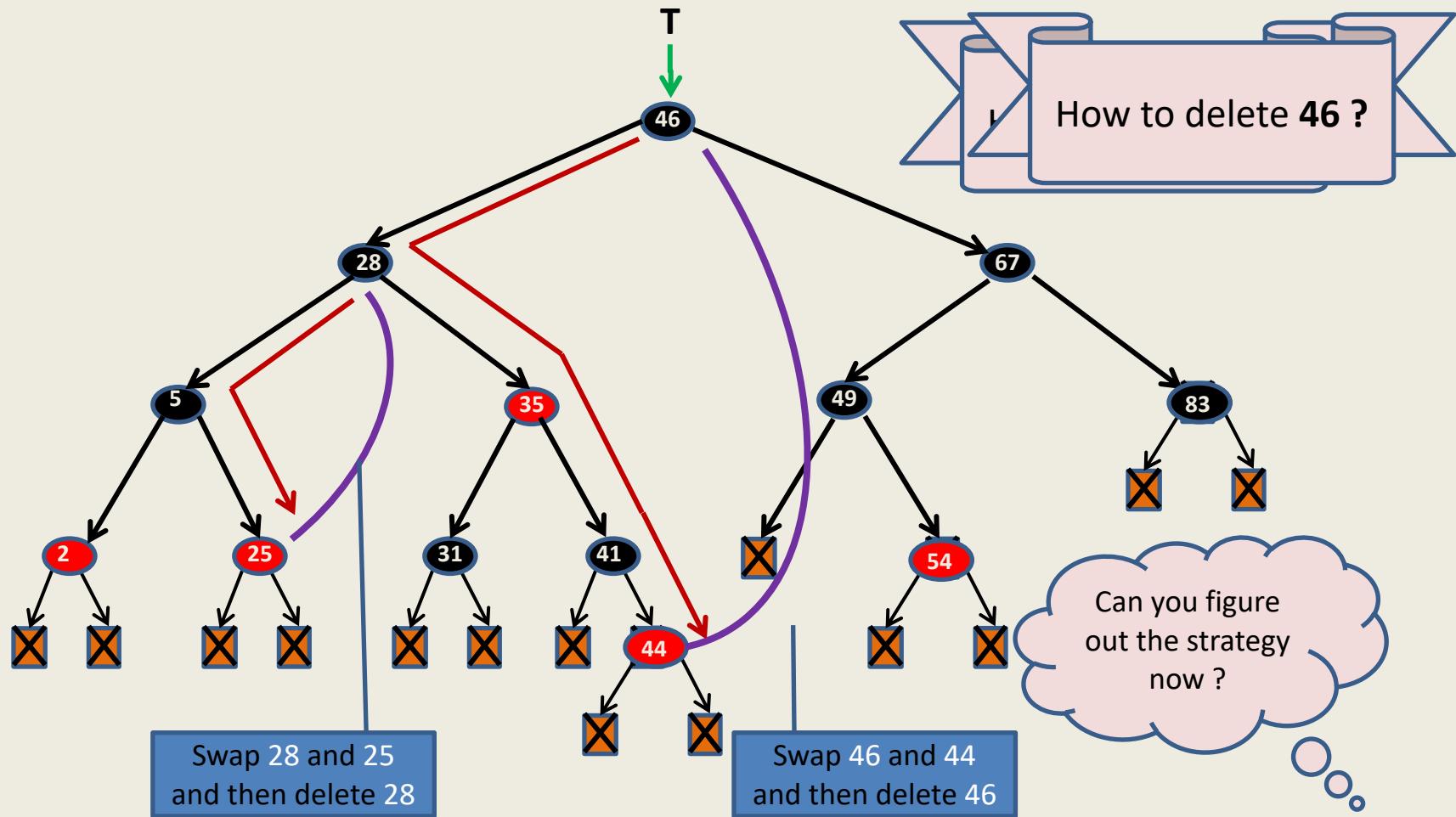
It is easier to maintain a BST under deletion if the node to be deleted has **at most** one child which is **non-leaf**.

Question: Can we transform every other case to the above case ?



Answer: ??

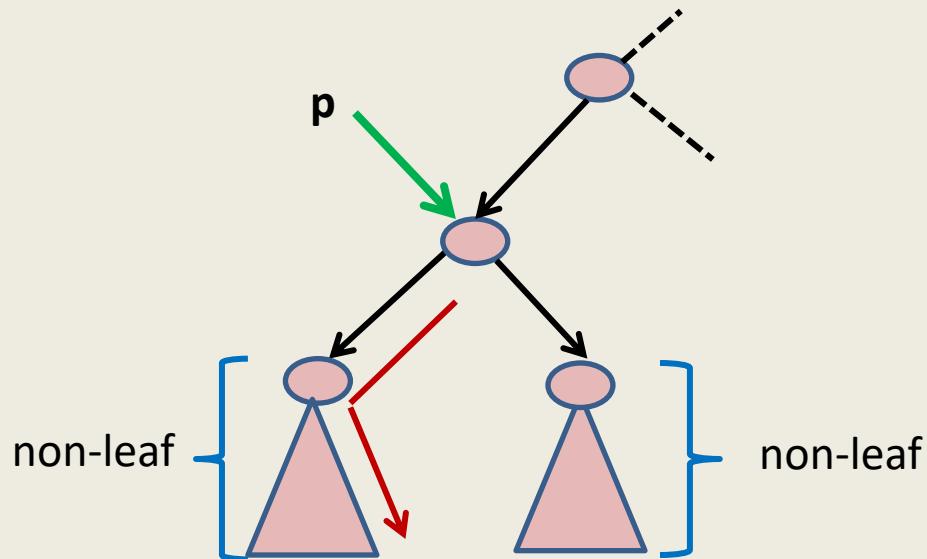
How to delete a node whose both children are non-leaves?



An important observation

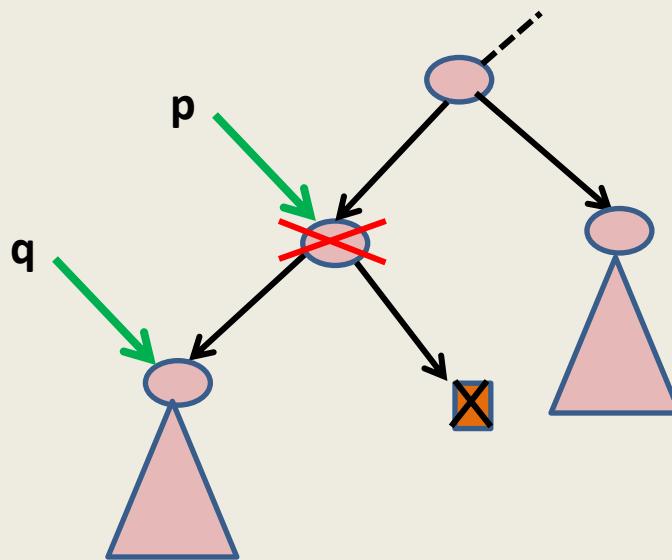
It is easier to maintain a BST under deletion if the node to be deleted has **at most** one child which is **non-leaf**.

Question: Can we transform every other case to the above case ?



Answer: by swapping **value(p)** with its predecessor,
and then deleting the predecessor node.

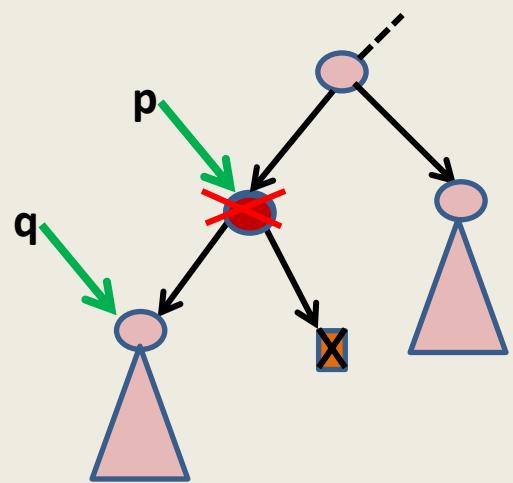
We need to handle deletion only for the following case



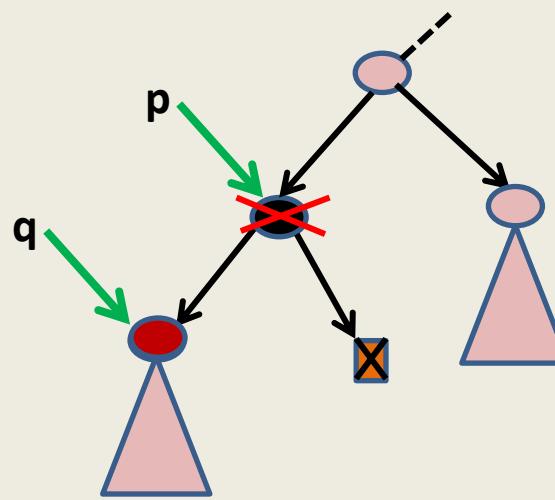
How to maintain a **red-black** tree under deletion ?

We shall first perform deletion like in an ordinary BST and then restore all properties of red-black tree.

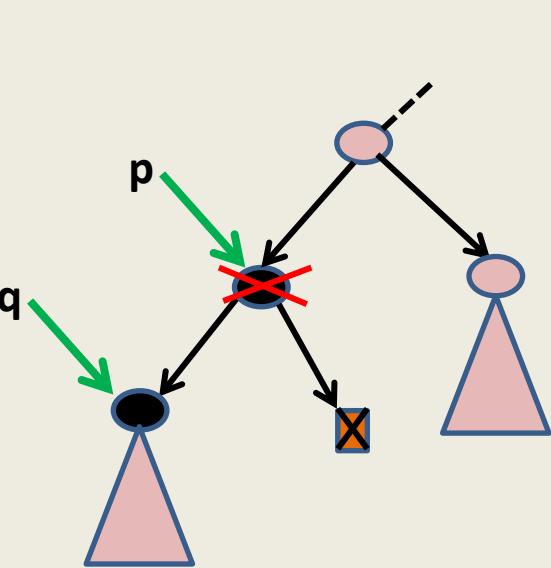
Easy cases and difficult case



Easy case

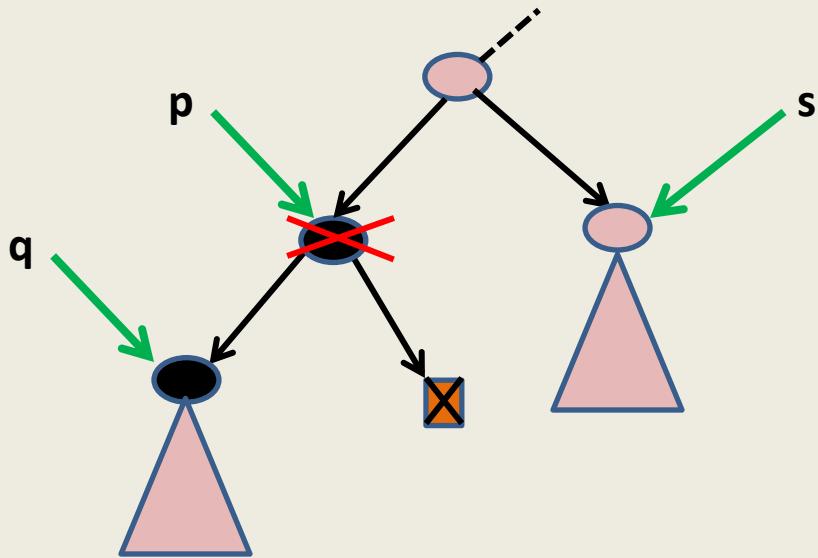


Easy case:
Change color of q to
black

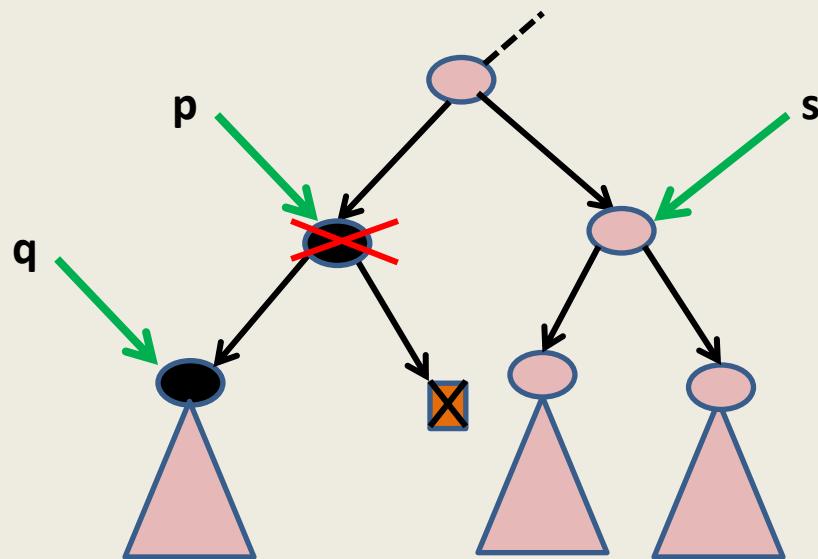


Difficult case

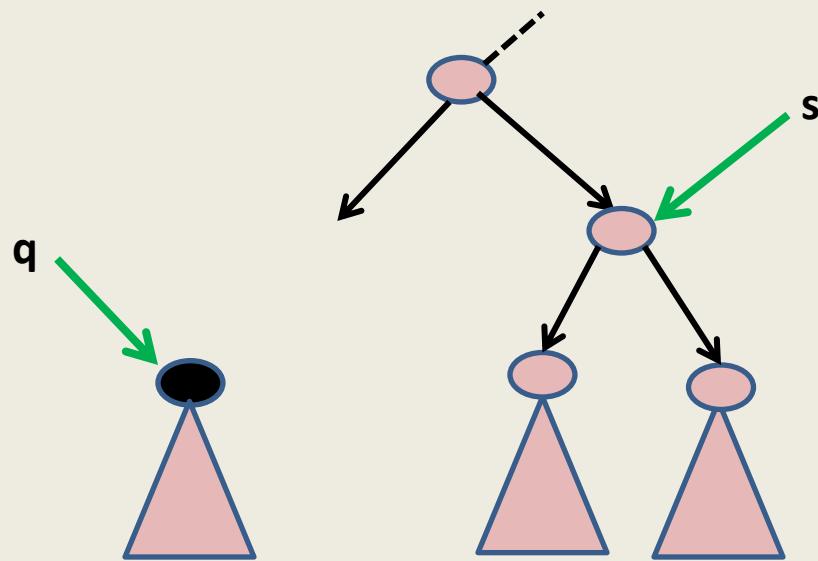
Handling the difficult case



Handling the difficult case

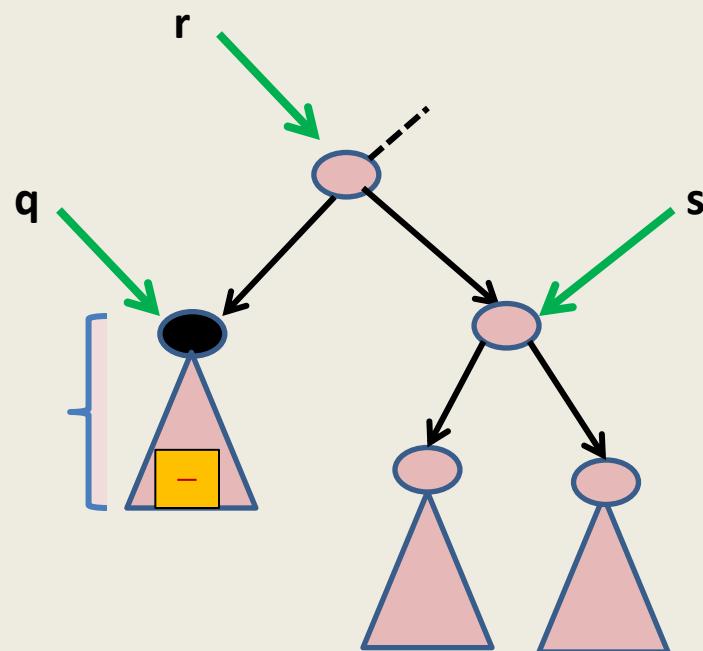


Handling the difficult case

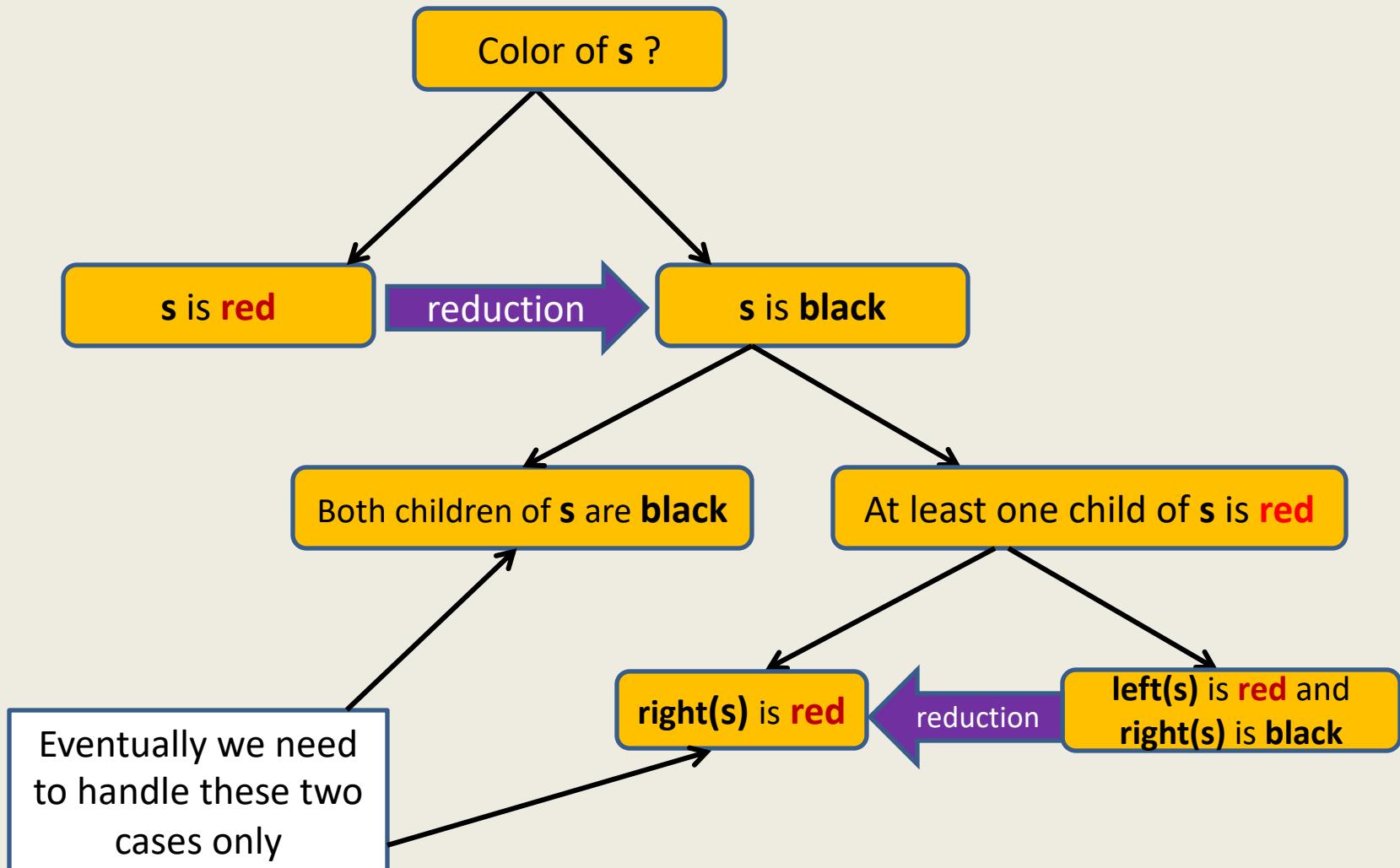


Handling the difficult case

Notice that the number of black nodes to each leaf node in subtree(q) has become **one** less than leaf nodes in other trees. We need an algorithm to remove this **black-height imbalance**.



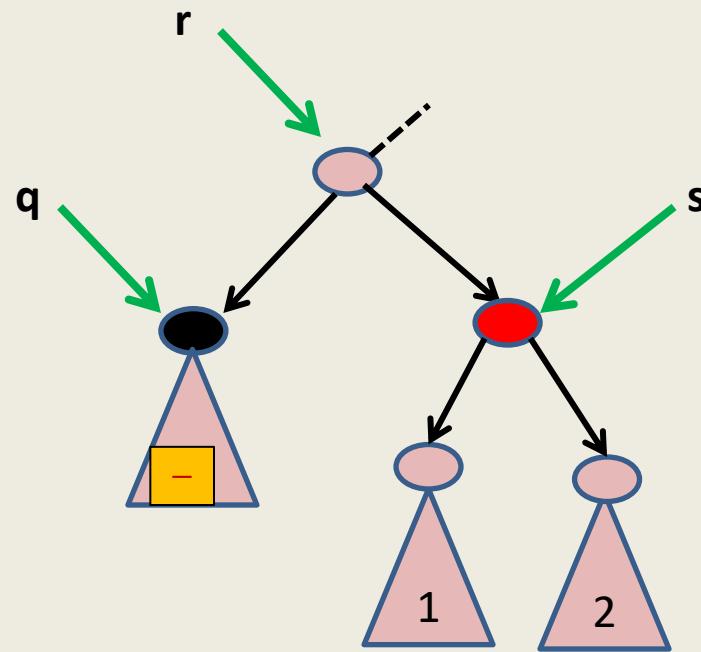
Handling the difficult case: An overview



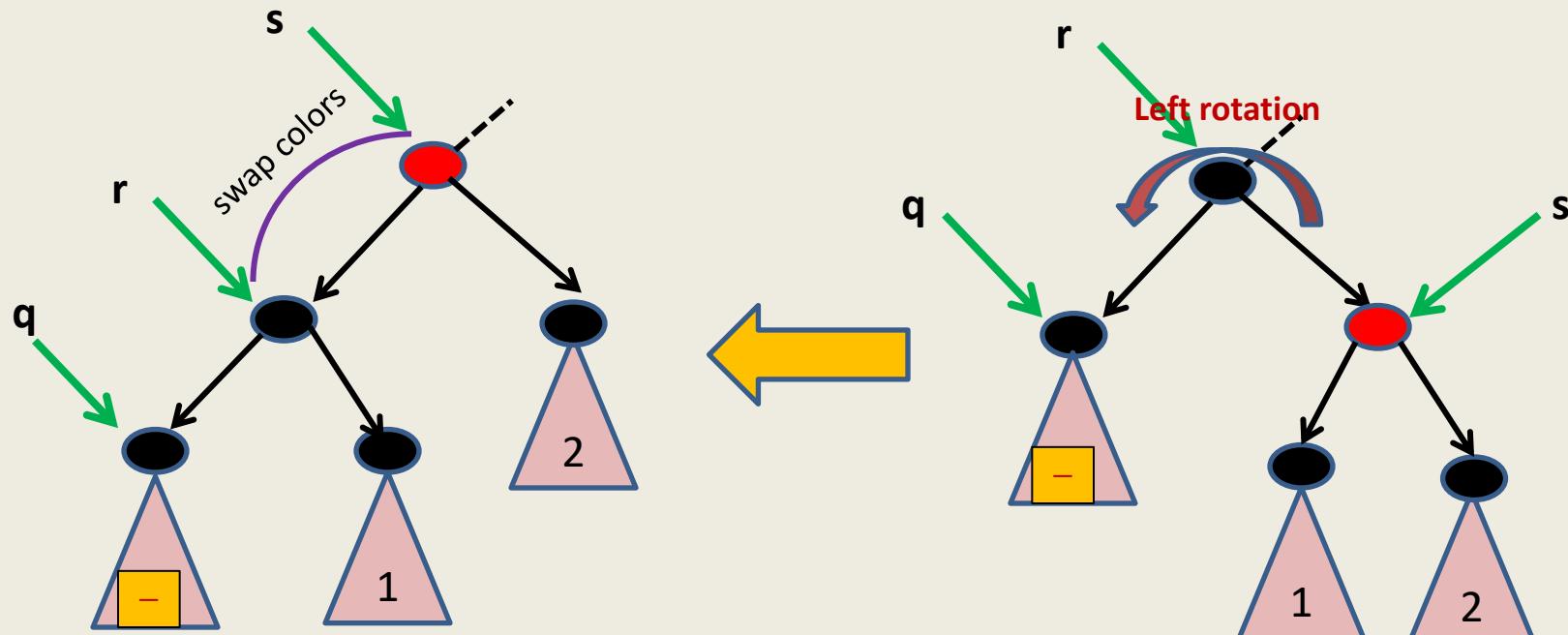
“s is red”  reduction “s is black”

“s is red” → reduction “s is black”

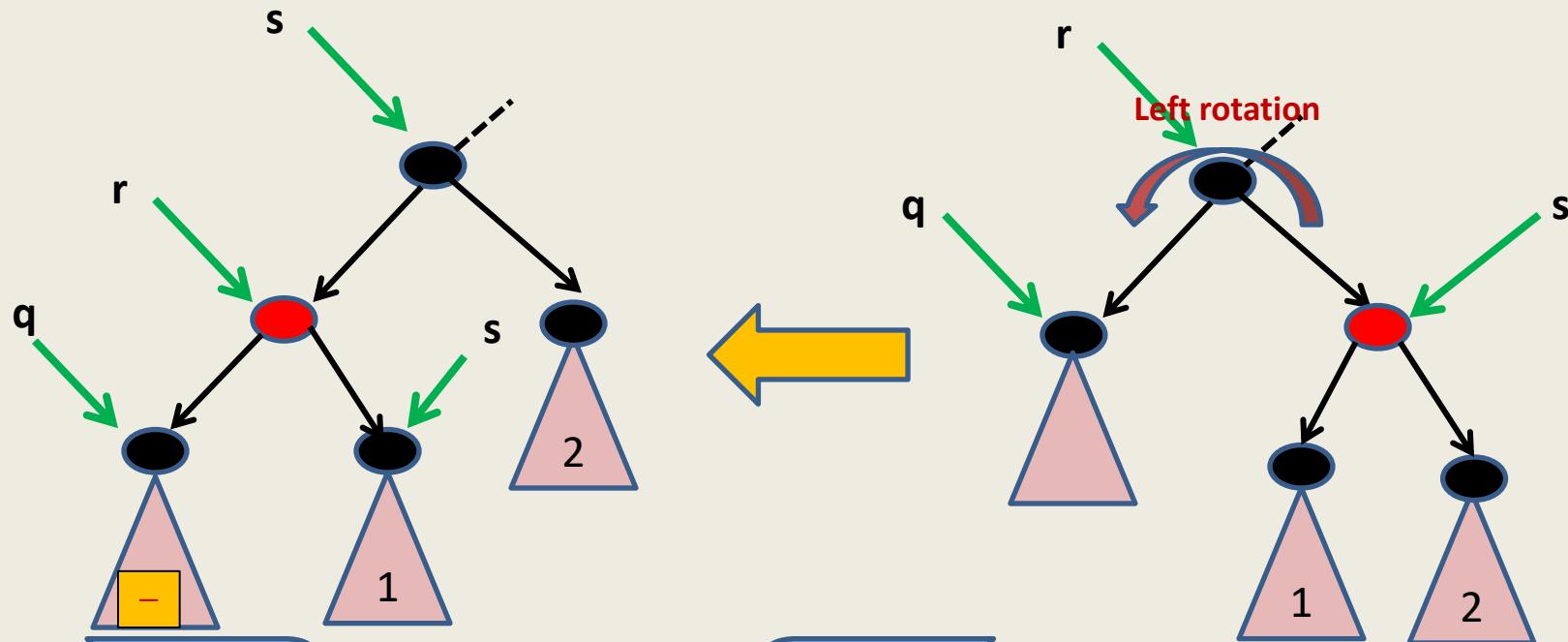
What can we say
about **parent** and
children of s ?



“s is red” → reduction → “s is black”



“s is red” reduction “s is black”



Convince yourself that the number of black nodes to any leaf of subtree(q) or subtrees 1 and 2 is now the same as before the rotation. And now the sibling of q is black. So we are done.

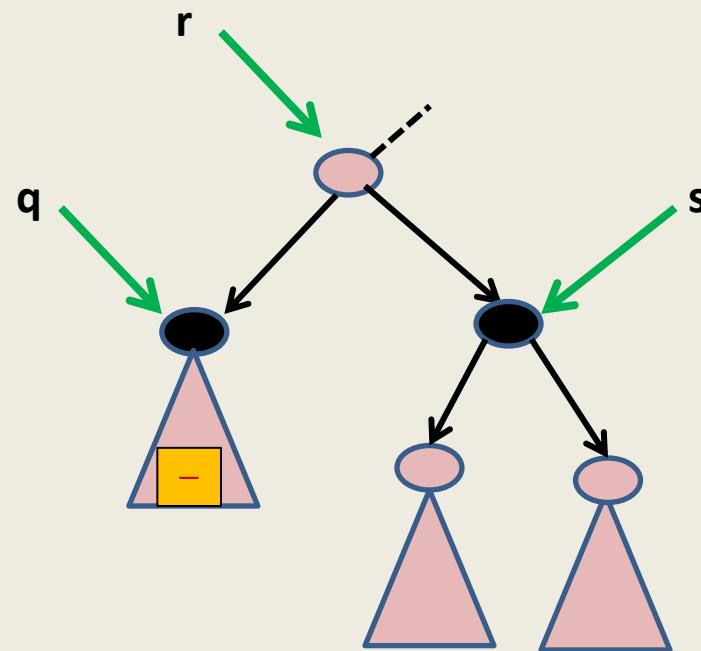
We just need to handle the case

“s is black”

Handling the case: s is black

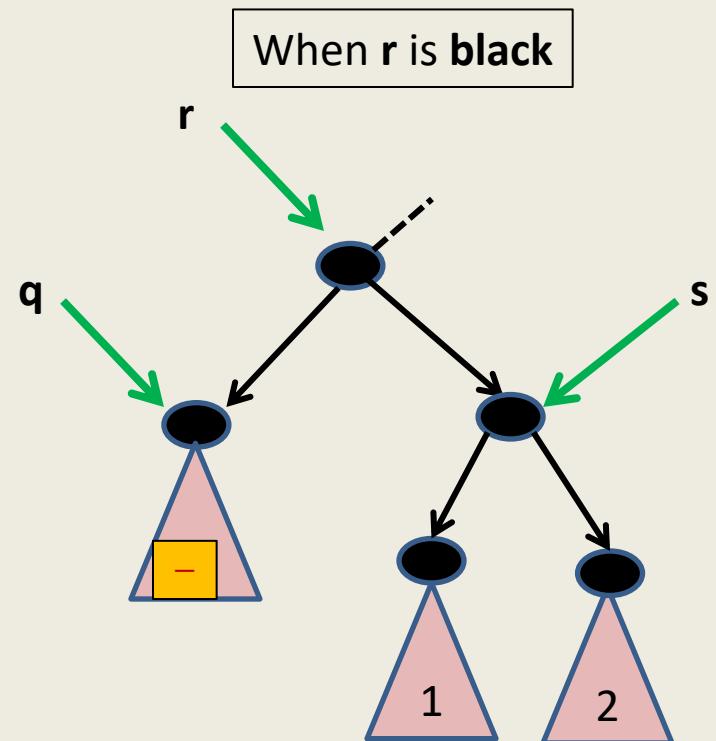
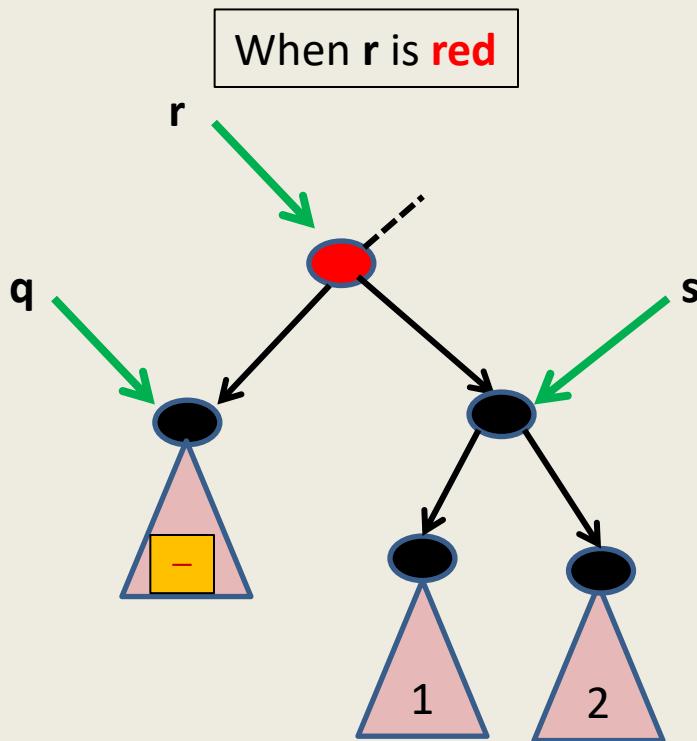
Case 1: both children of s are **black**

Case 2: at least one child of s is **red**



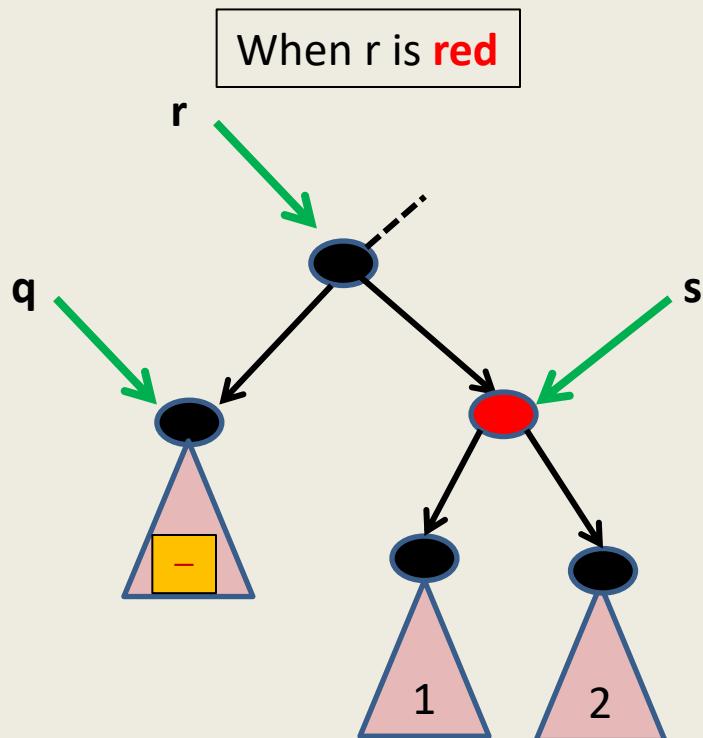
Handling the case:
s is black and both children of s are black

Handling the case: s is black and both children of s are black

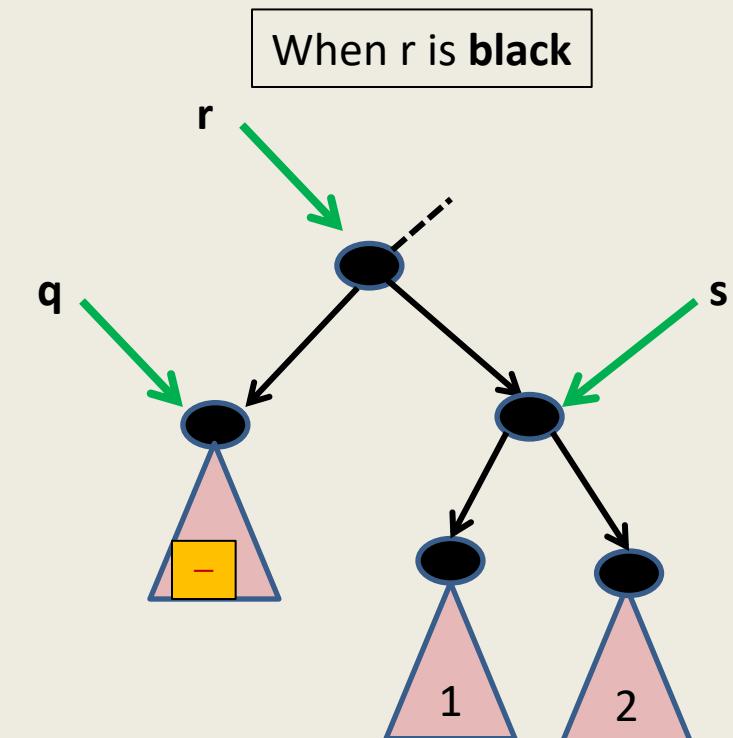


How to handle this case ?

Handling the case: s is black and both children of s are black

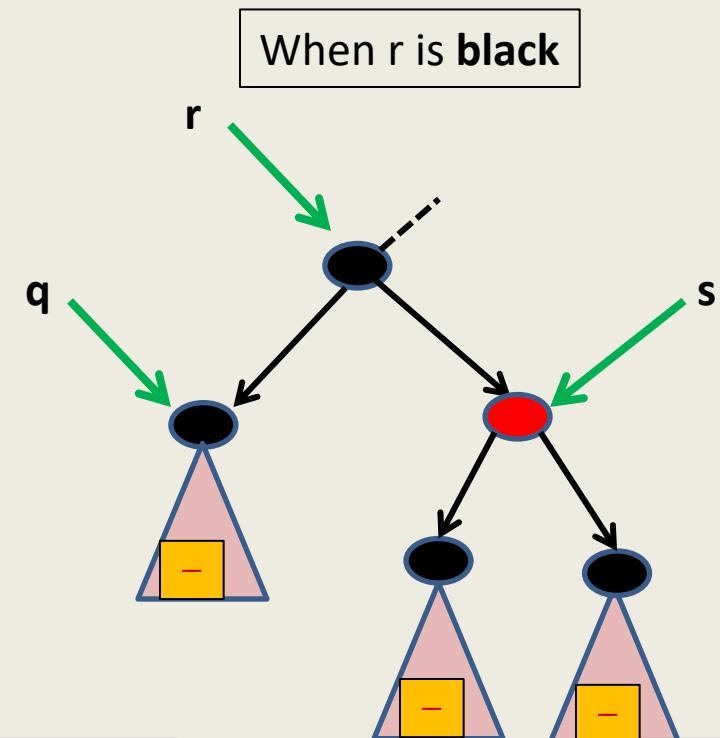
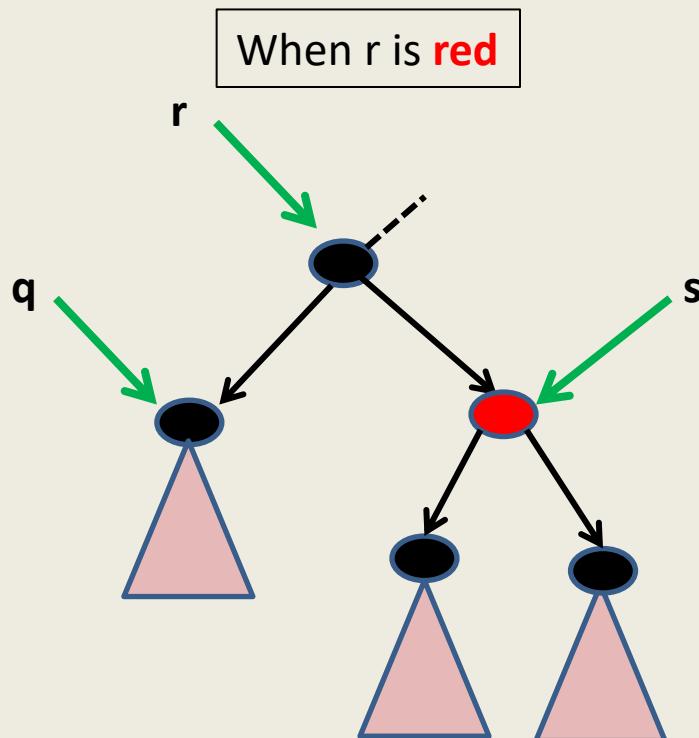


YES.
As a result of swapping the colors, the number of black nodes to the leaves of trees 1 and 2 unchanged. Interestingly, the deficiency of one black node on the path to the leaves of subtree(q) is also compensated. So we are done😊



How to handle this case ?

Handling the case: s is black and both children of s are black

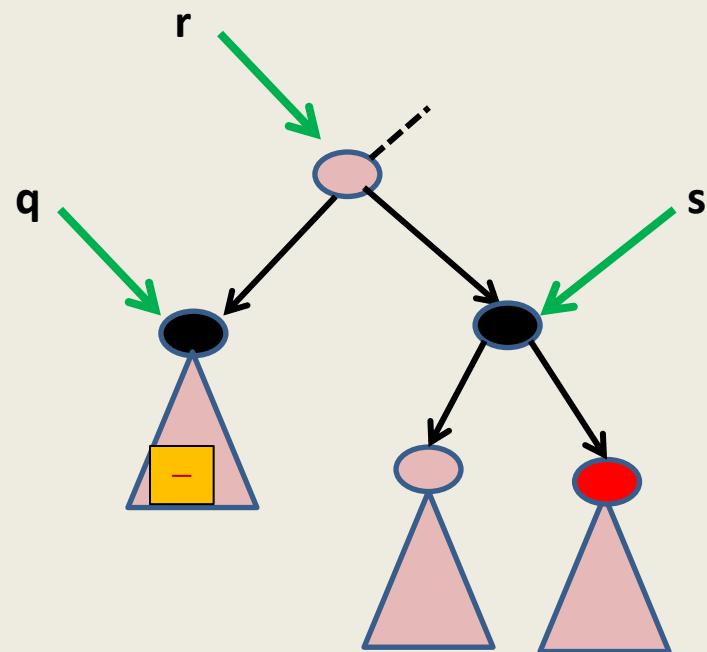


Changing color of s to **red** has reduced the number of black nodes on the path to the root of subtree(s) by one. As a result the imbalance of black height has *propagated* upward. So we process the new **q**.

Handling the case:
s is **black** and one of its children is **red**

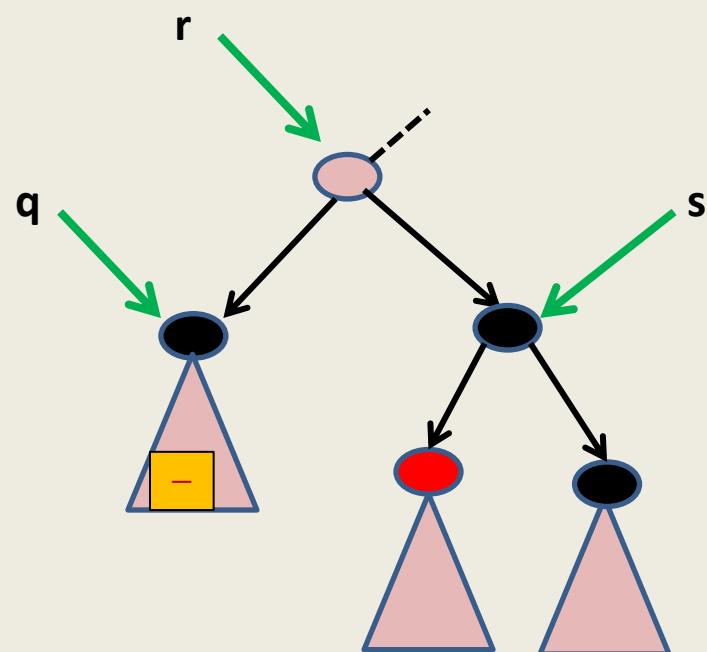
There are two cases

When **right(s)** is **red**



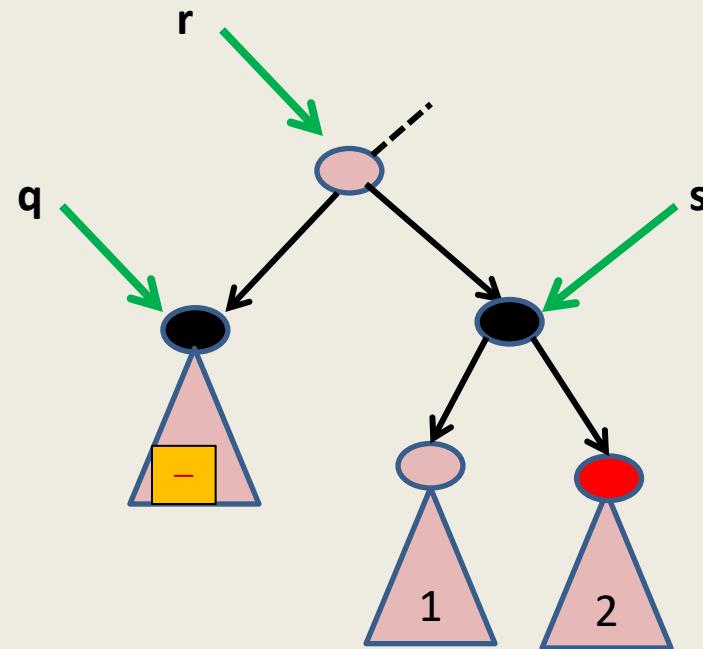
reduction

When **left(s)** is **red** and **right(s)** is **black**



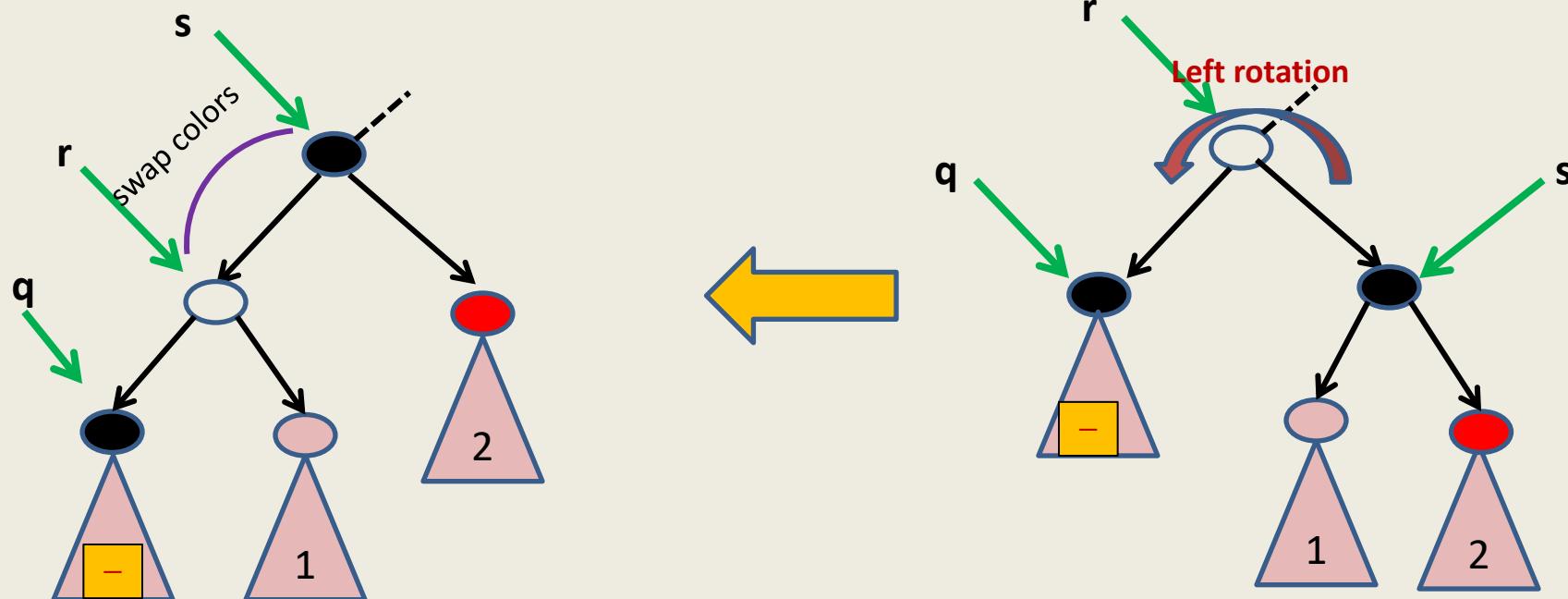
Handling the case: right(s) is red

Handling the case: right(s) is red



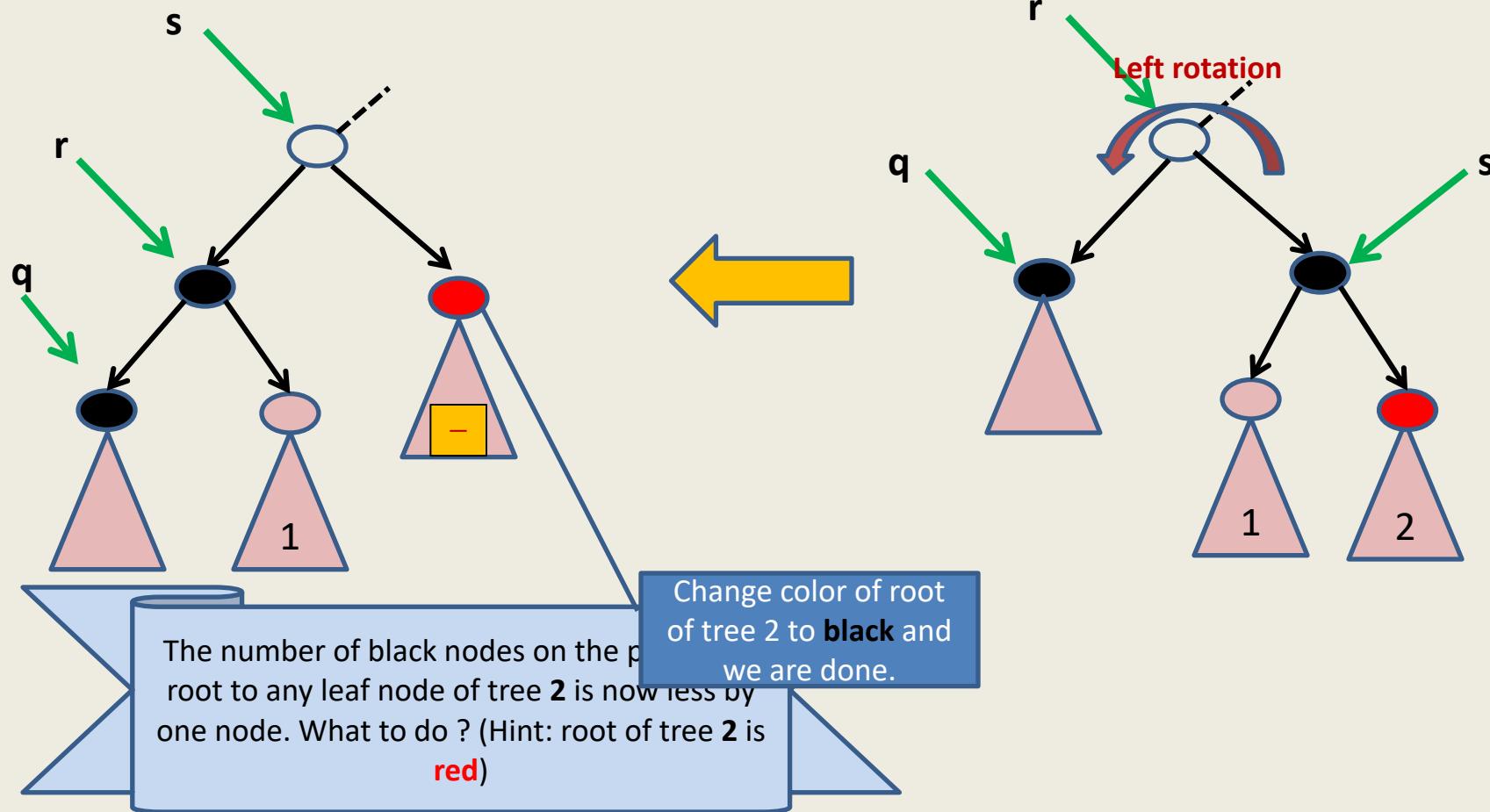
Let $\text{color}(r)$ be c

Handling the case: right(s) is red

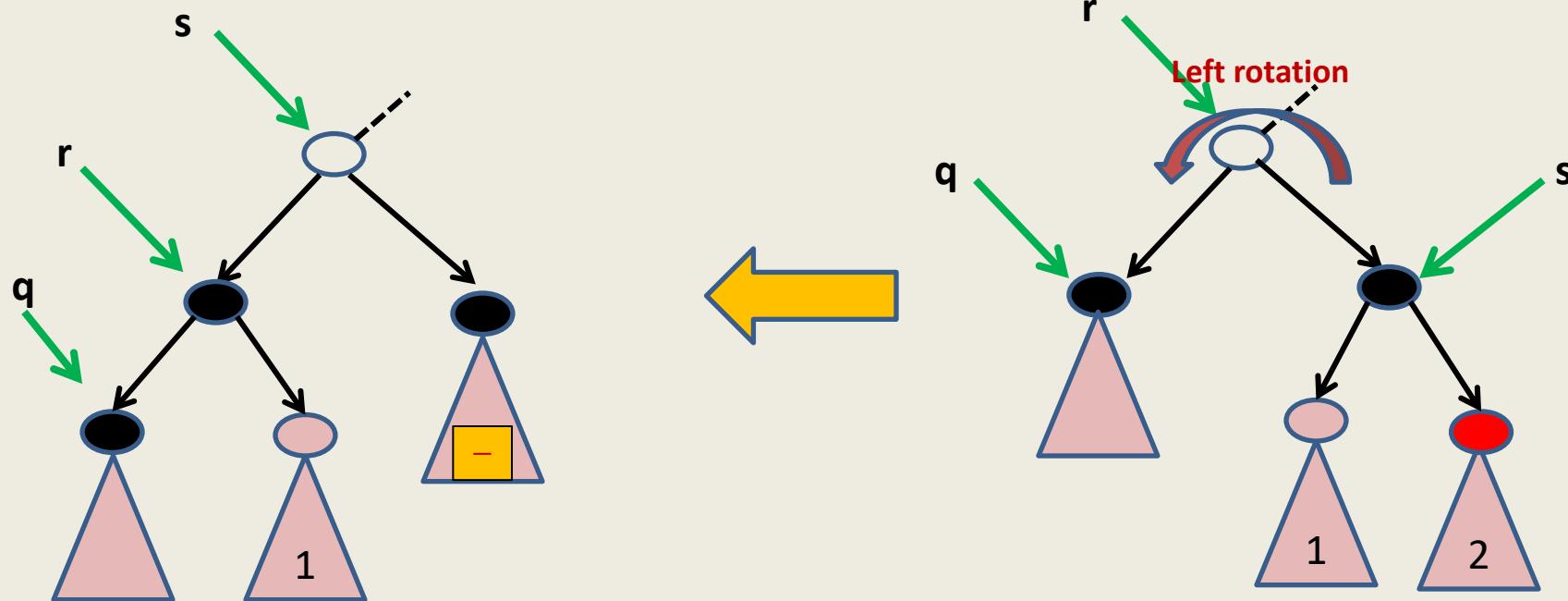


The number of black nodes on the path from root to any leaf node of subtree(q) has increased by one (this is good!), has remained unchanged for leaves of tree 1, and is uncertain for leaves of tree 2(depends upon c). How to get rid of this uncertainty ?

Handling the case: right(s) is red



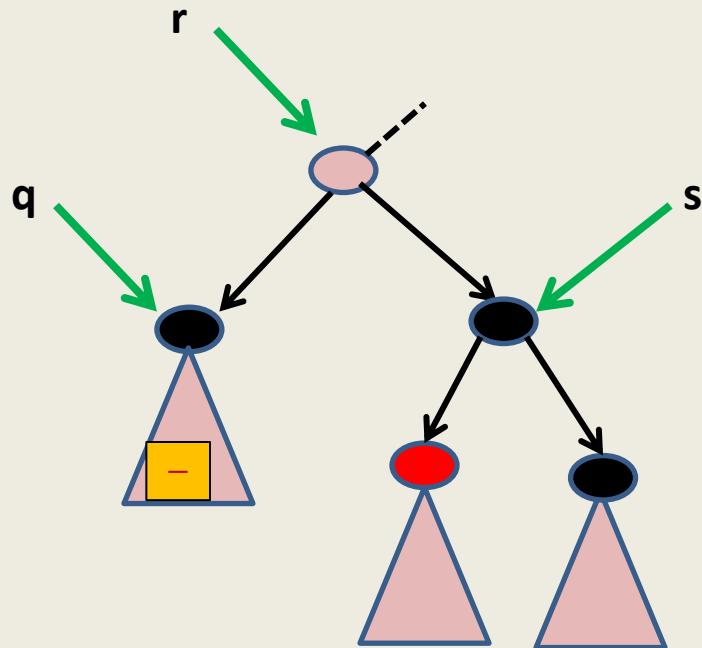
Handling the case: right(s) is red



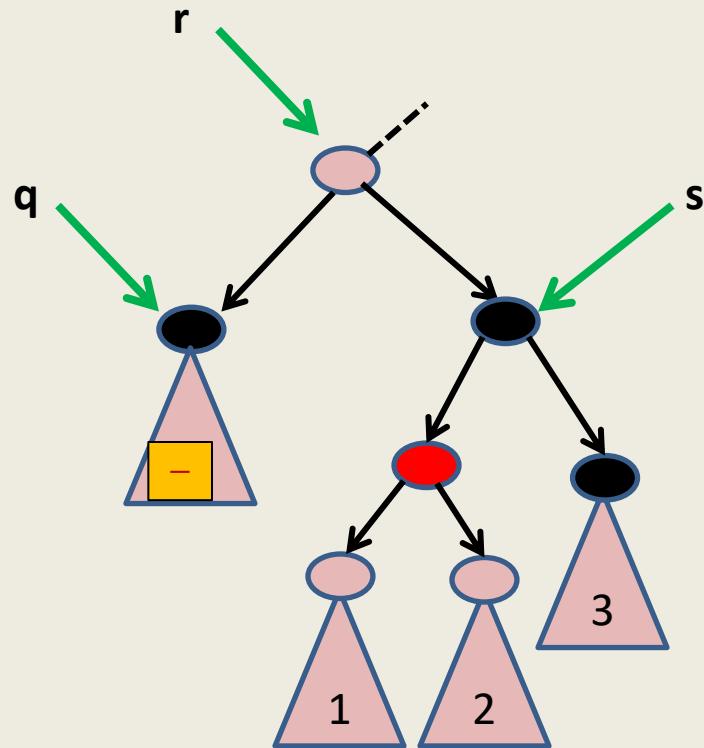
Convince yourself that left rotation at r , followed by color swap of s and r , followed by change of color of root of tree 2 removes the imbalance of black height for all leaf nodes of the subtrees shown.

Handling the case
“left(s) is **red** and right(s) is **black**”

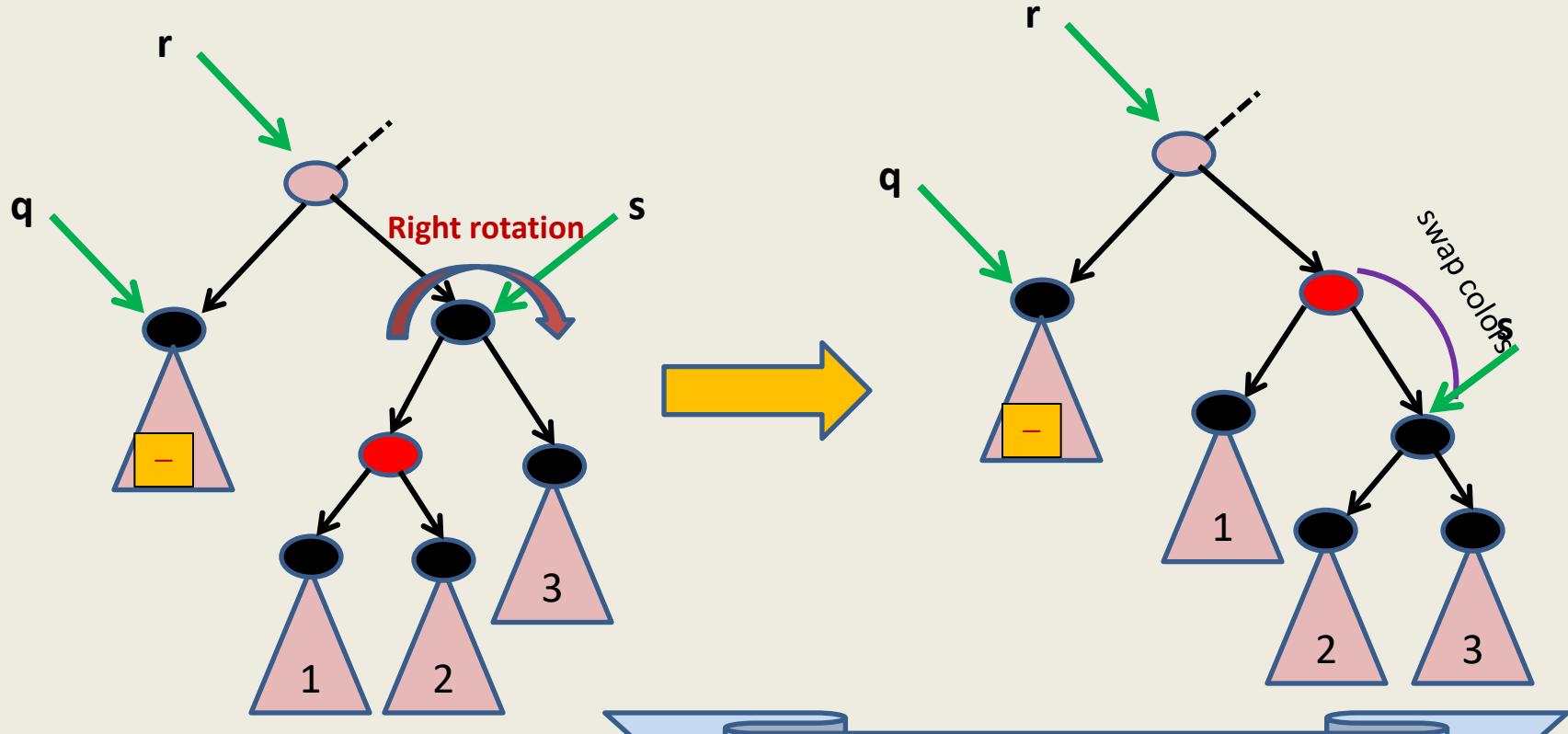
Handling the case: left(s) is red and right(s) is black



Handling the case: left(s) is **red** and right(s) is **black**

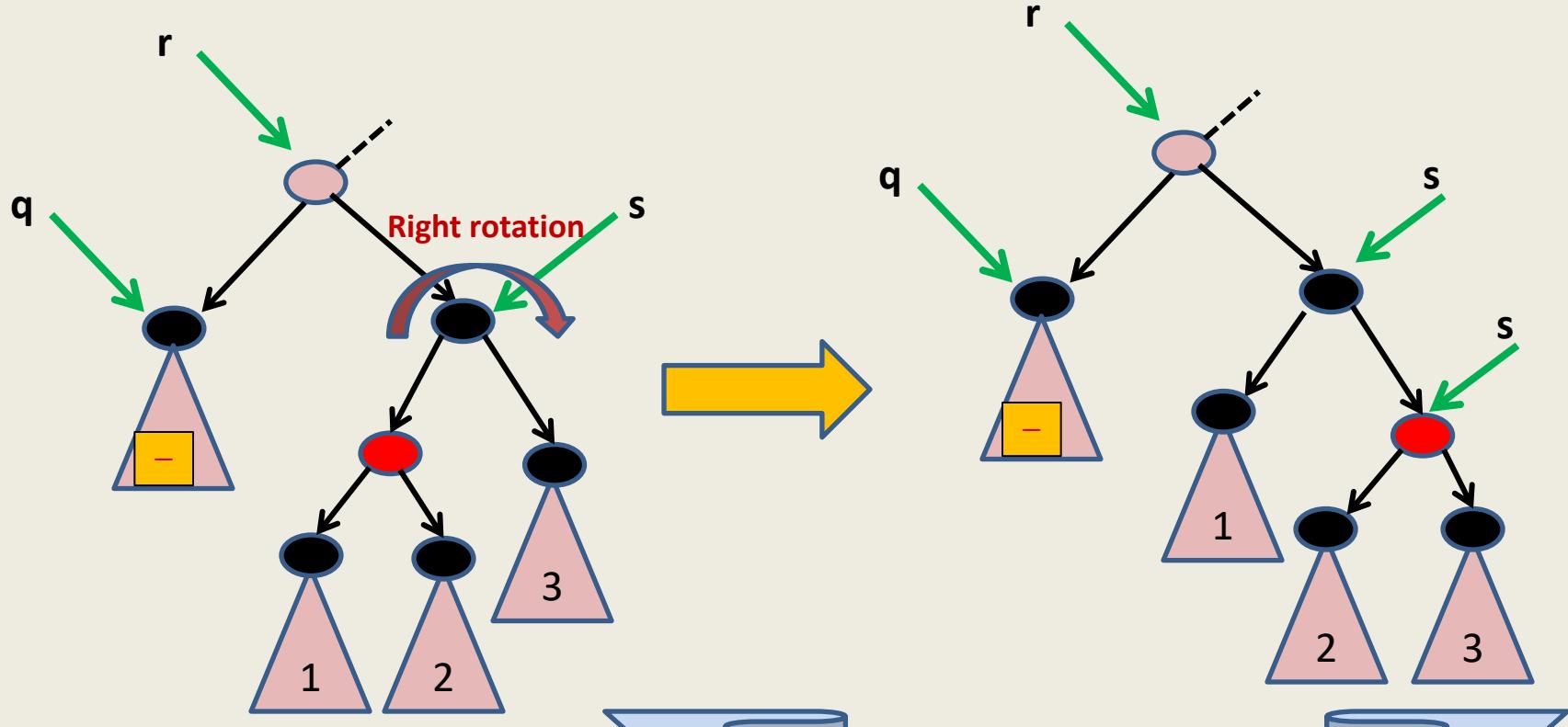


Handling the case: left(s) is **red** and right(s) is **black**



The number of black nodes on the path from root to any leaf node in tree 1 has now reduced by one although it is the same for trees 2 and 3.
What should we do ?

Handling the case: left(s) is red and right(s) is black



Notice that now the new sibling of q has its right child red. So we have effectively reduced the current case to the case which we know how to handle.

Theorem: We can maintain red-black trees in $O(\log n)$ time per insert/delete/search operation.

where n is the number of the nodes in the tree.

A **Red** Black Tree is height balanced

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

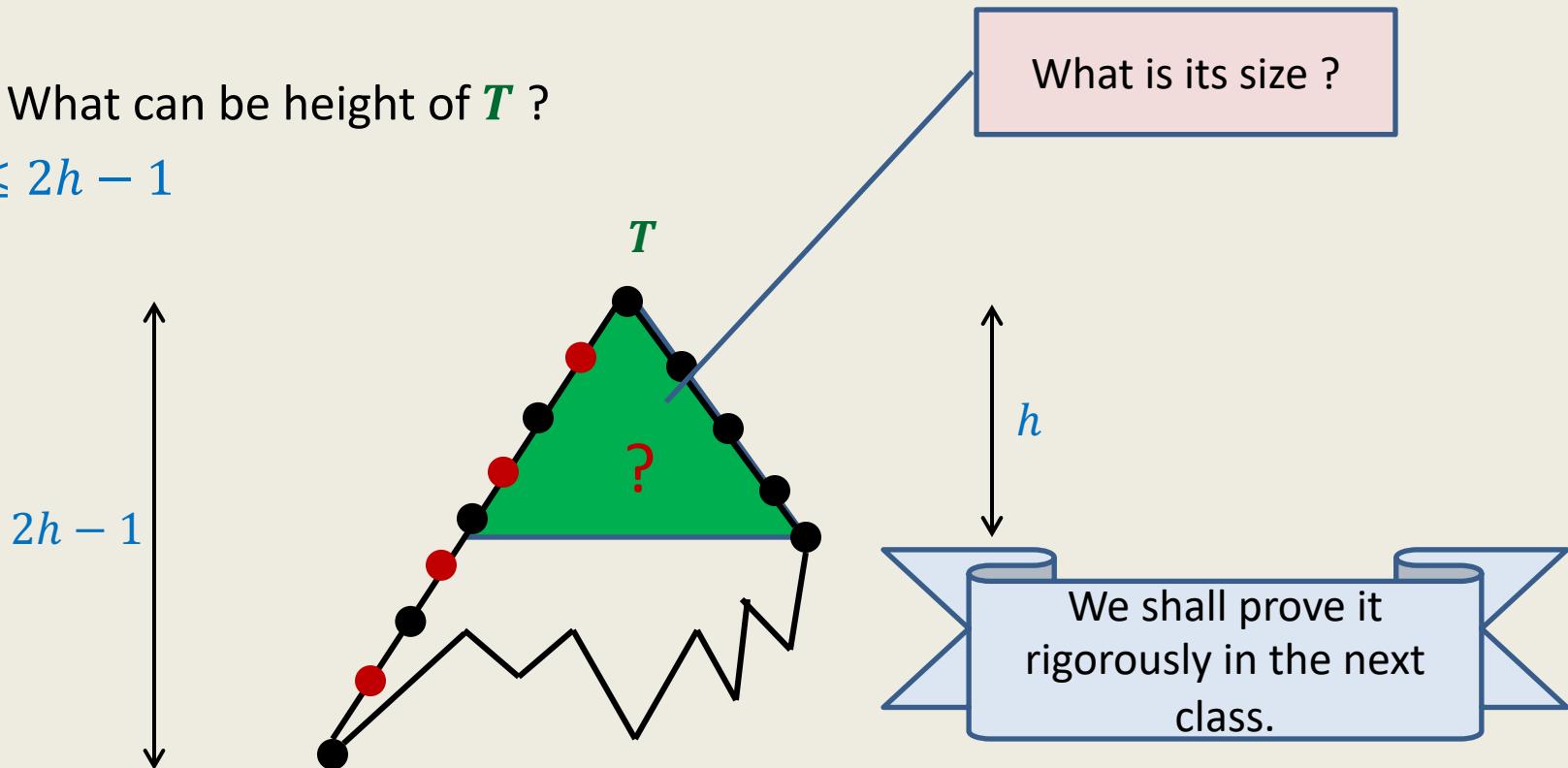
Why is a red black tree height balanced ?

T : a red black tree

h : black height of T .

Question: What can be height of T ?

Answer: $\leq 2h - 1$

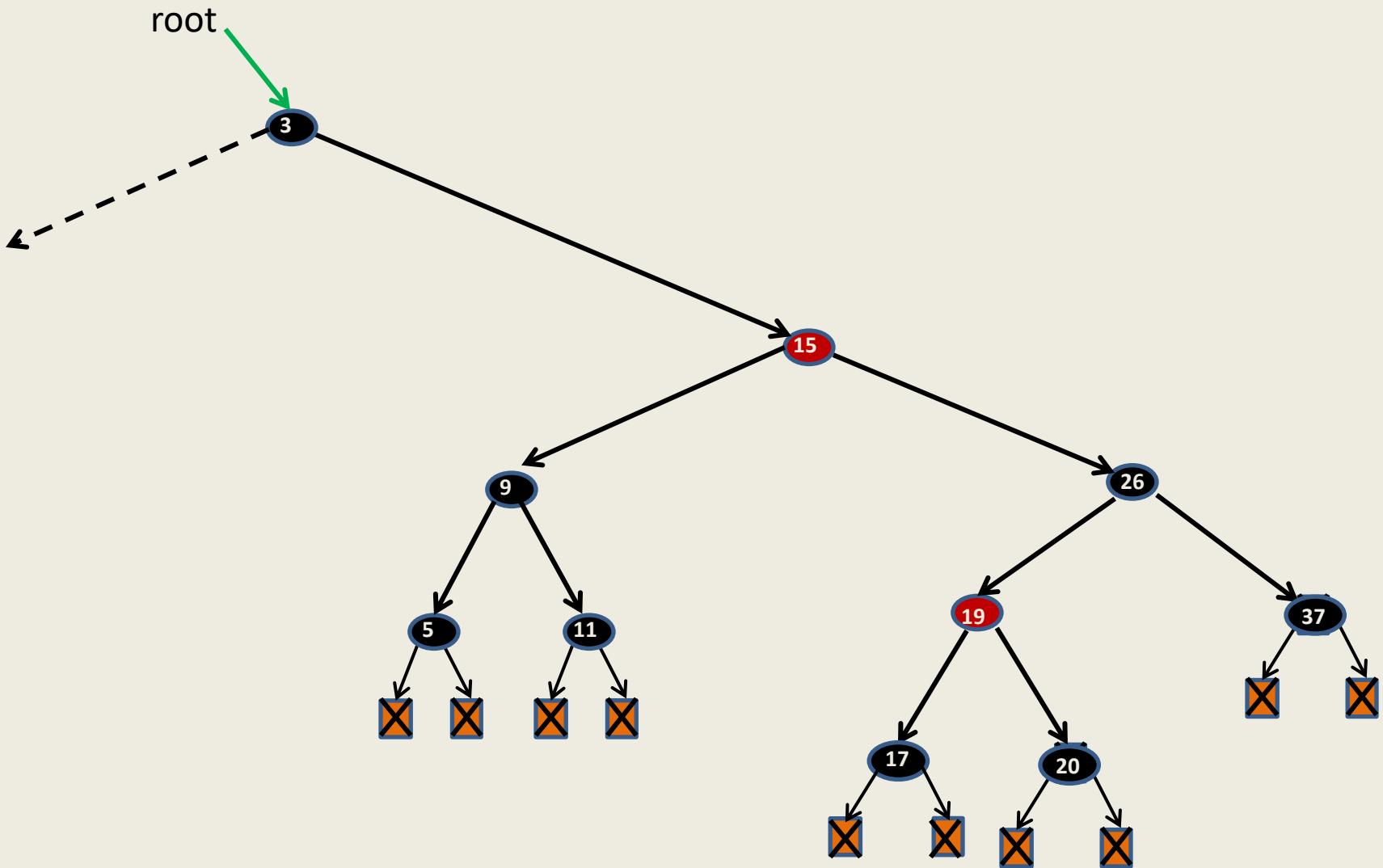


Theorem: The shaded green tree is a complete binary tree & so has $\geq 2^h$ elements.⁴⁷

A practice problem

On deletion in
red-black trees

How to delete 9 ?



Data Structures and Algorithms

(ESO207)

Lecture 19

Analysis of

- Red Black trees
- Nearly Balanced BST

A **Red** Black Tree is height balanced

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

Red Black Tree

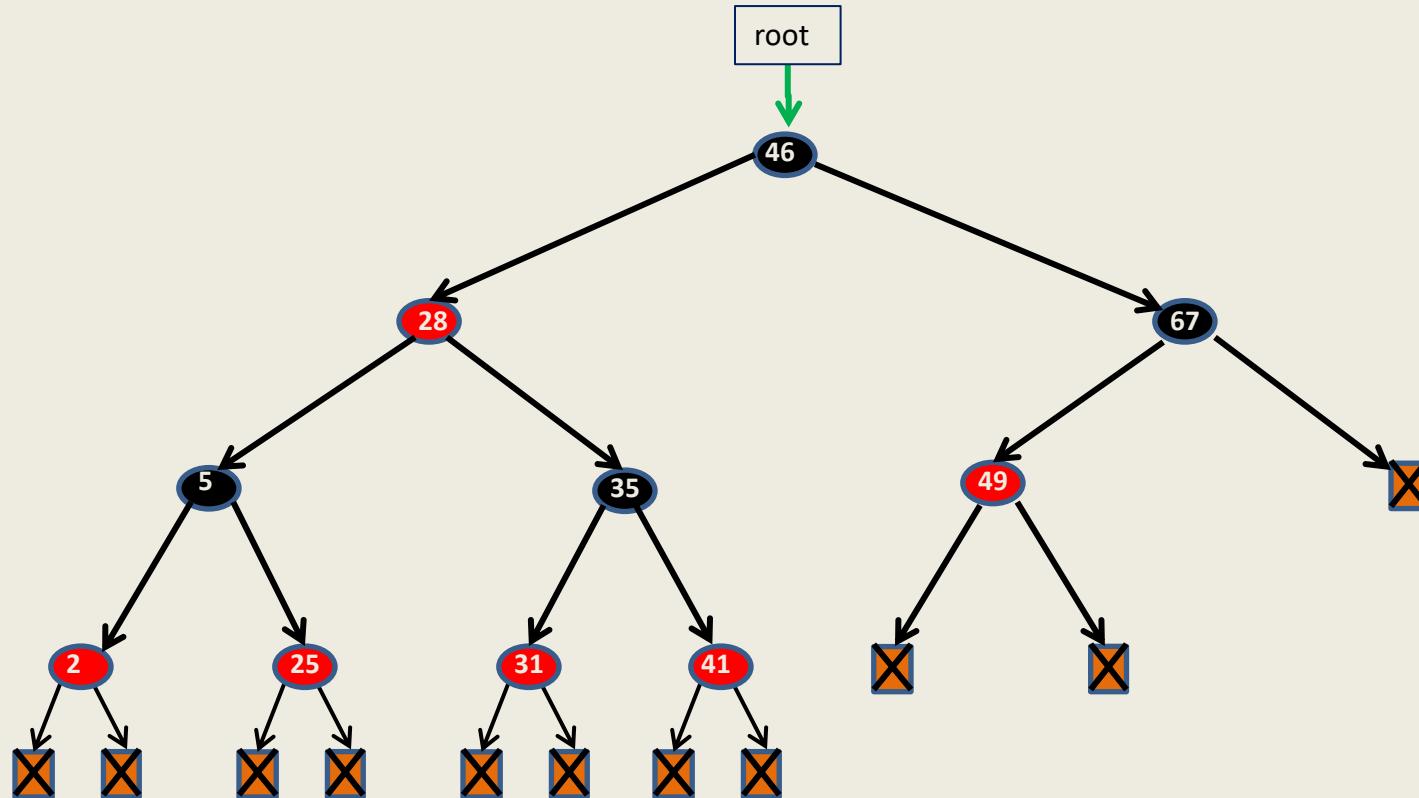
Red Black tree:

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

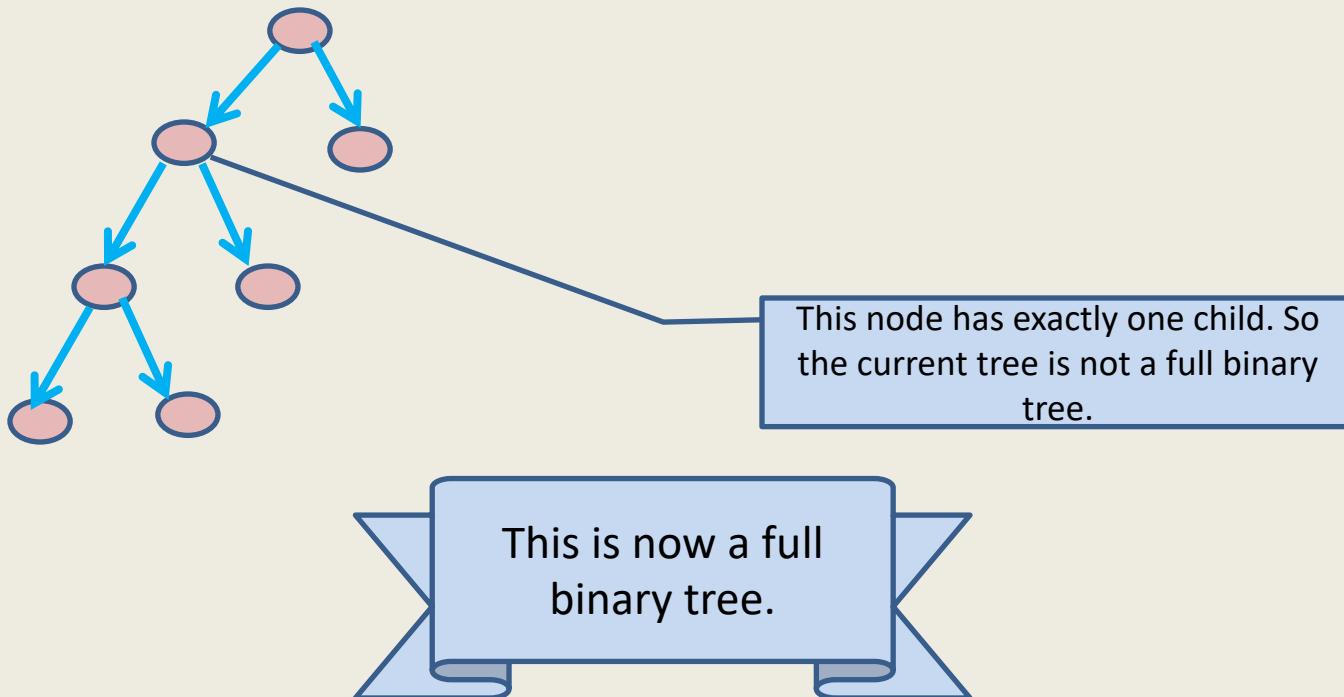
A red-black tree



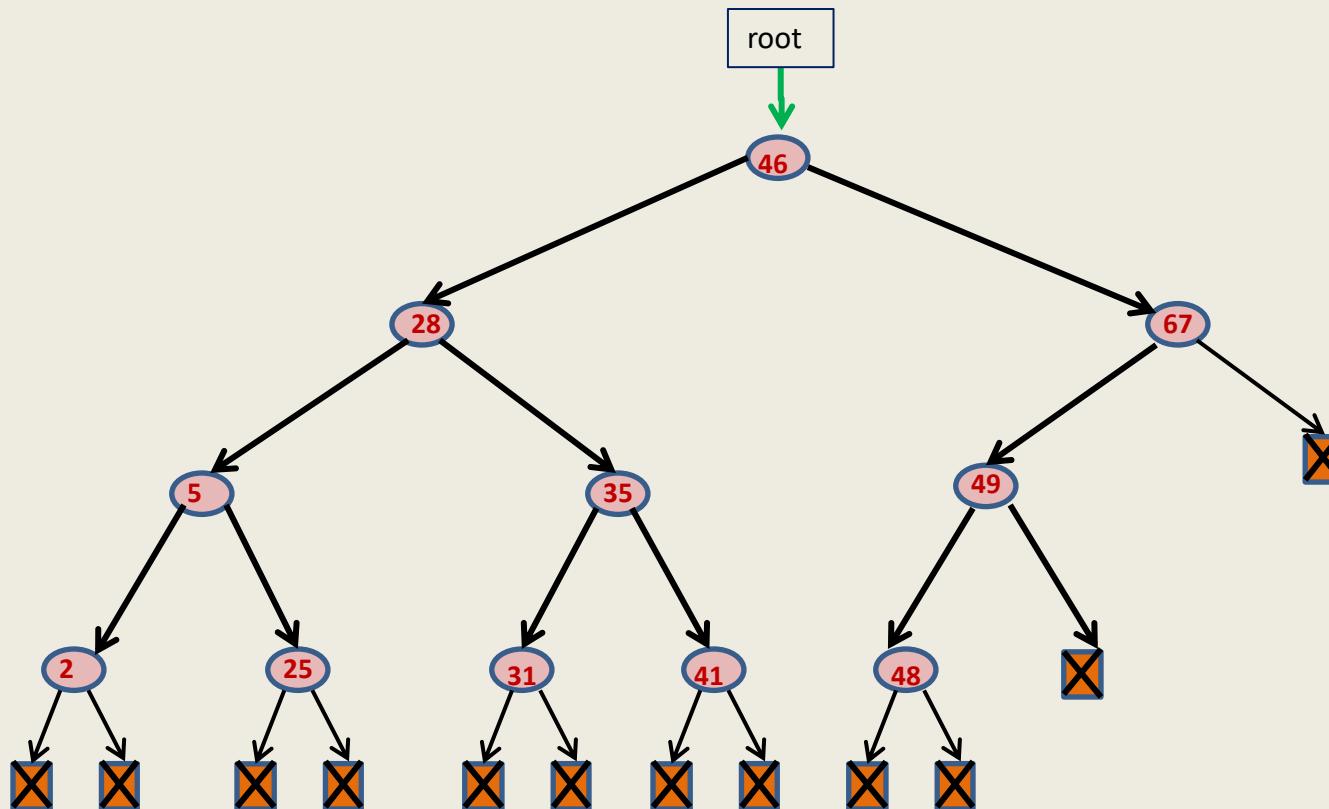
Terminologies

Full binary tree:

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

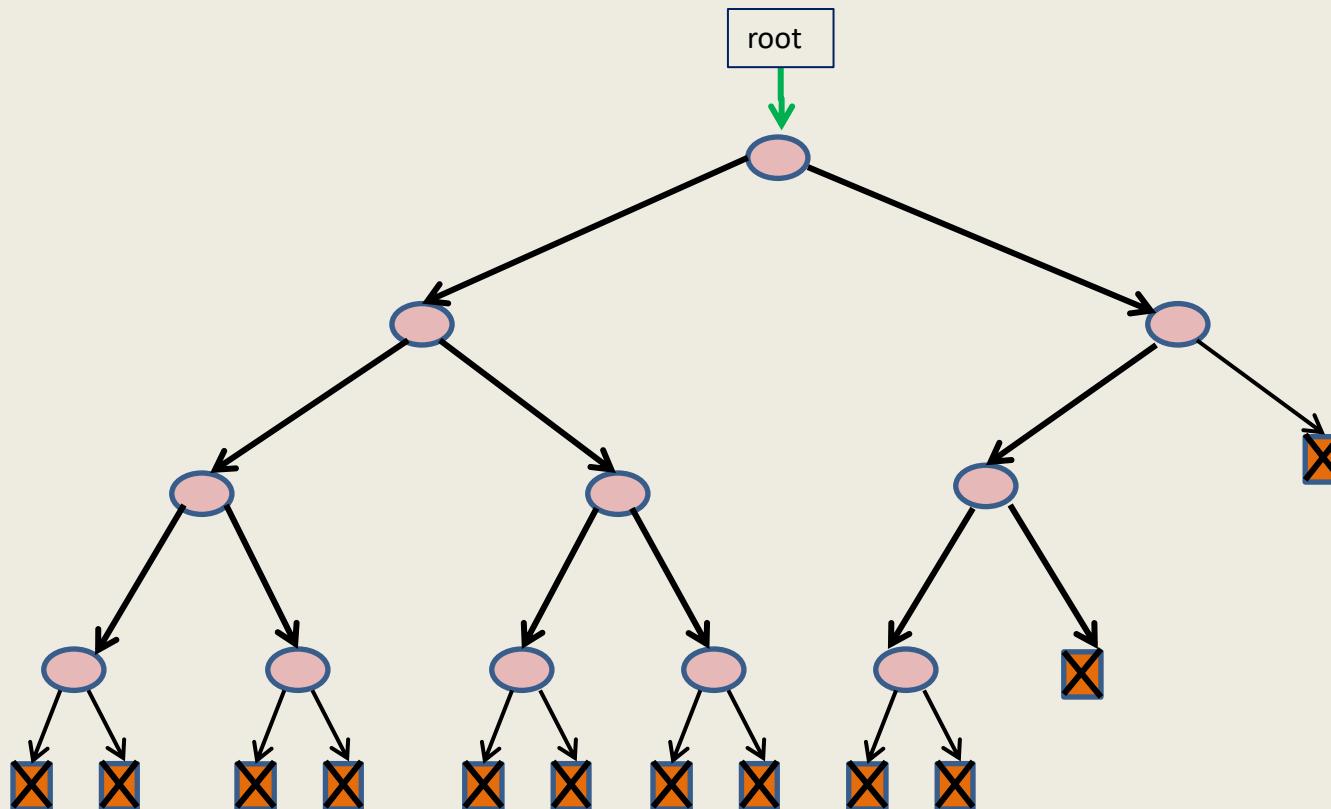


Red-black tree: as a Full Binary Tree



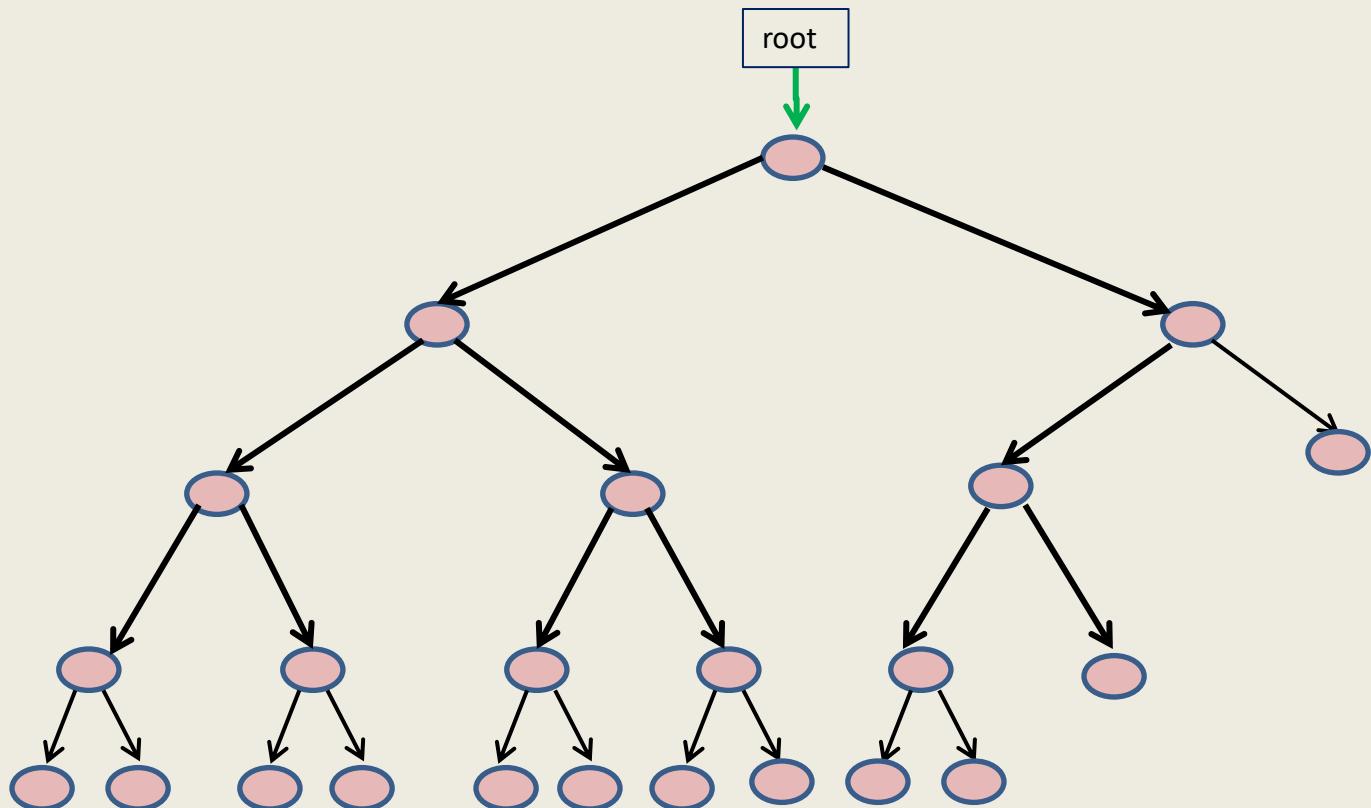
Ignore the values

Red-black tree: as a Full Binary Tree



Ignore the distinction
between internal nodes
and leaf nodes

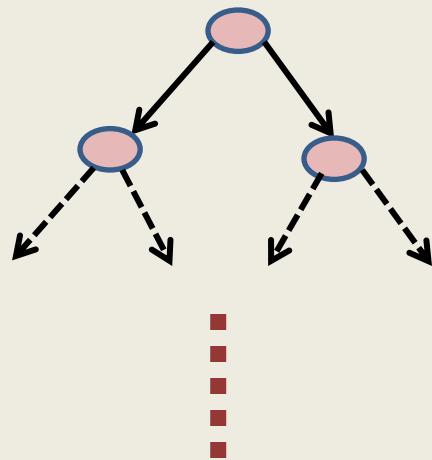
Red-black tree: as a Full Binary Tree



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

**Relationship between
Number of leaf nodes and
Number of internal nodes**

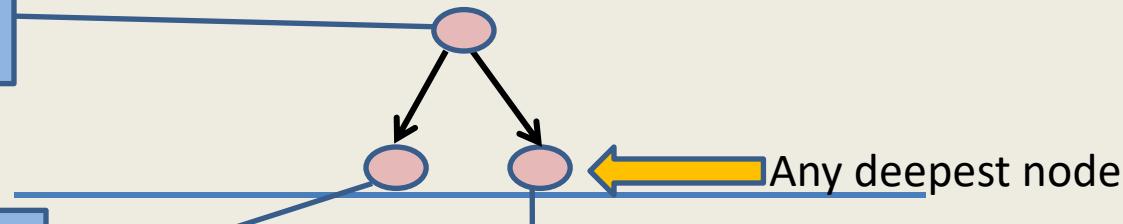
A full binary tree



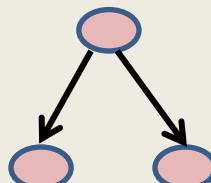
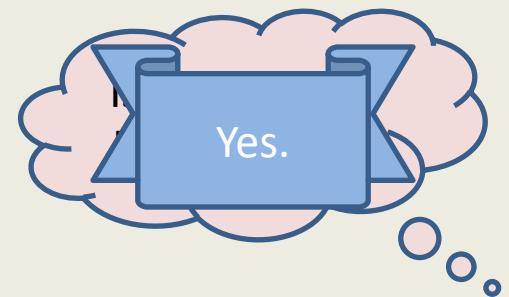
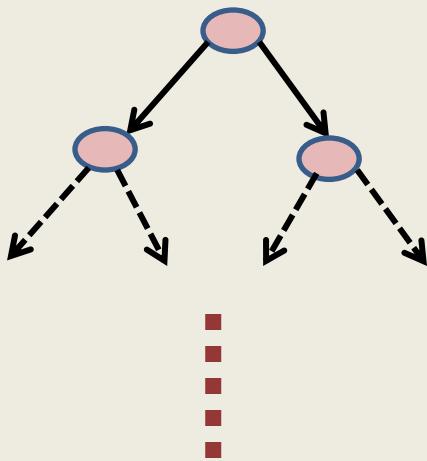
This node must have a left child since the tree is a full binary tree

This node must be a leaf node. Give reason.

Otherwise this node won't be the deepest node.



A full binary tree



What happened to the number of internal nodes ?

Reduced by one

What happened to the number of leaf nodes ?

Reduced by one

A full binary tree

Analyze the process:

Repeat

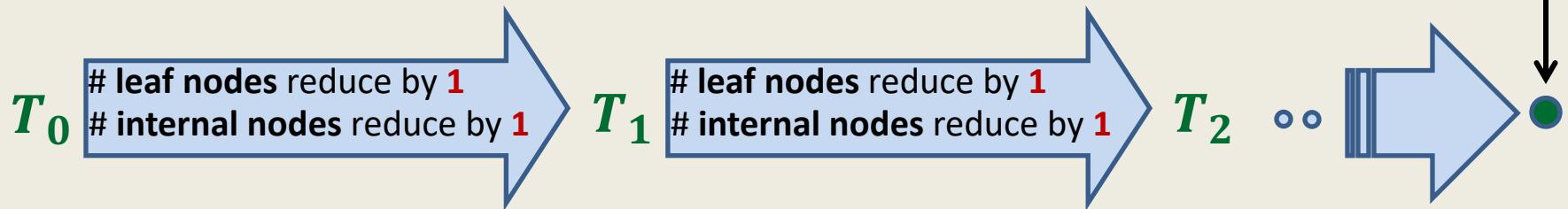
{ Delete the deepest node and its sibling }

until only **root** remains

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

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

Finally only
root node remains



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 i is the number of internal nodes in a full binary tree T , what is the size (number of nodes) of the tree ?

Answer: $2i + 1$

Question: What is the size of a Red Black tree storing n keys ?

Answer: $2n + 1$

A **complete** binary tree of height ***h*** and its Properties

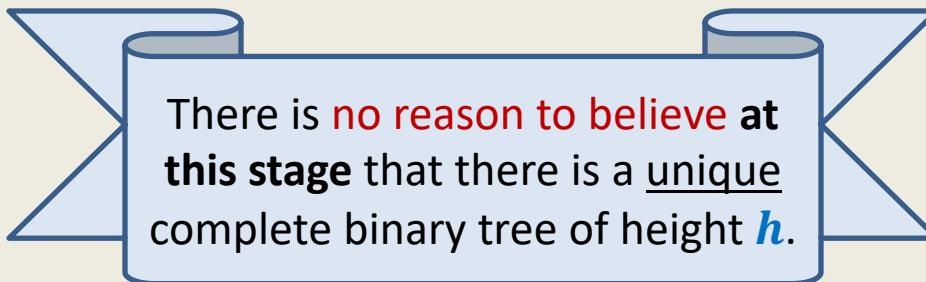
A **complete** binary tree of height h

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 .



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

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

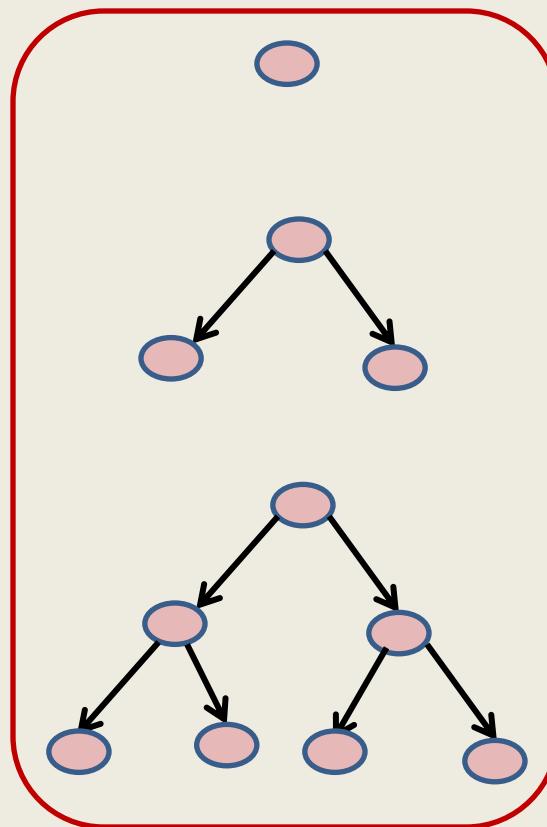
if every leaf node is at depth h .

A complete binary tree of height h

Complete binary tree of height 1 ?

Complete binary tree of height 2 ?

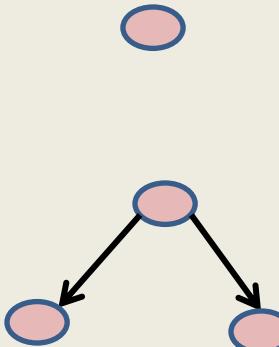
Complete binary tree of height 3 ?



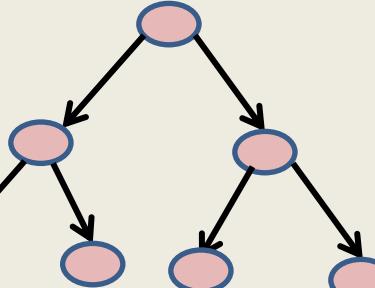
Try to generalize
the type of tree
shown here to
tree of height h .

A complete binary tree of height h

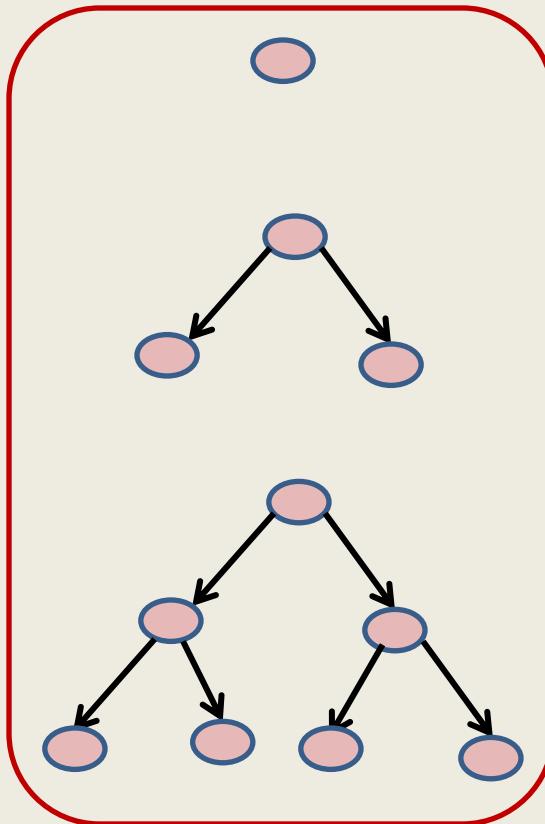
Complete binary tree of height 1 ?



Complete binary tree of height 2 ?



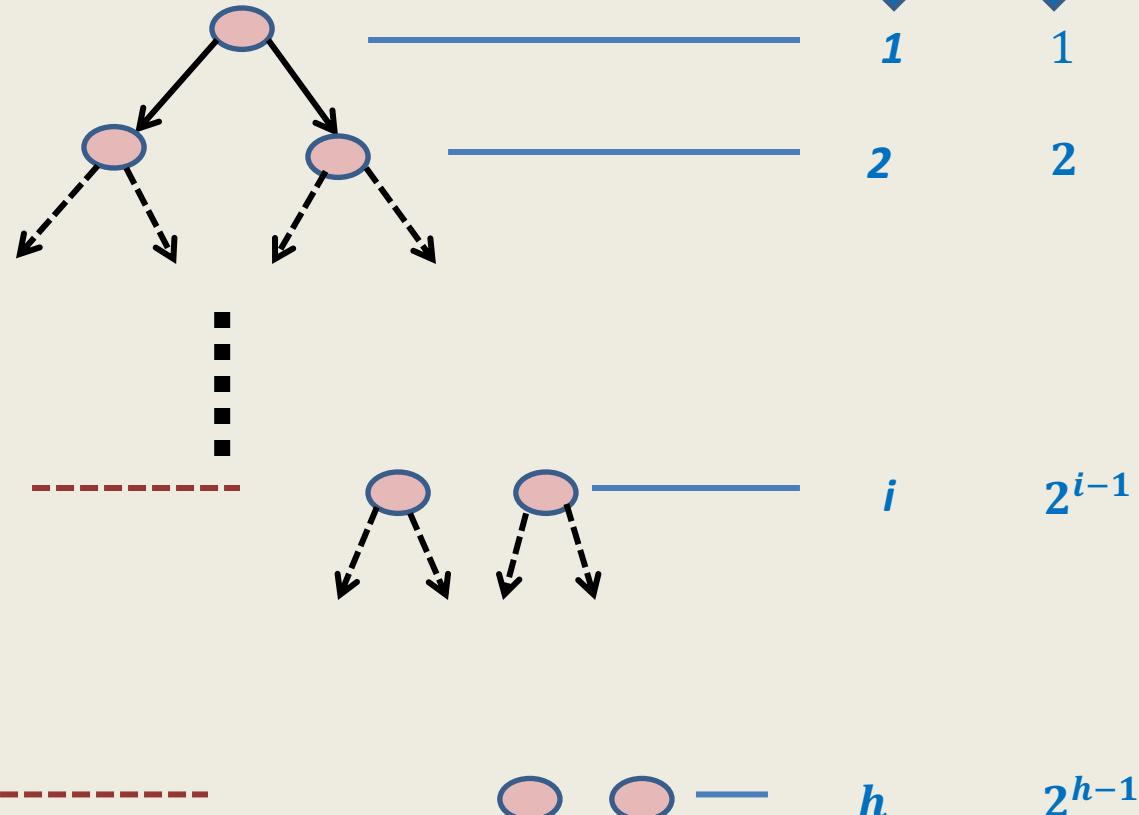
Complete binary tree of height 3 ?



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 .

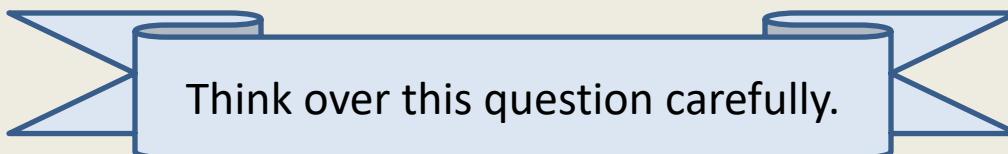
Uniqueness of a **complete** binary tree of height h

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

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

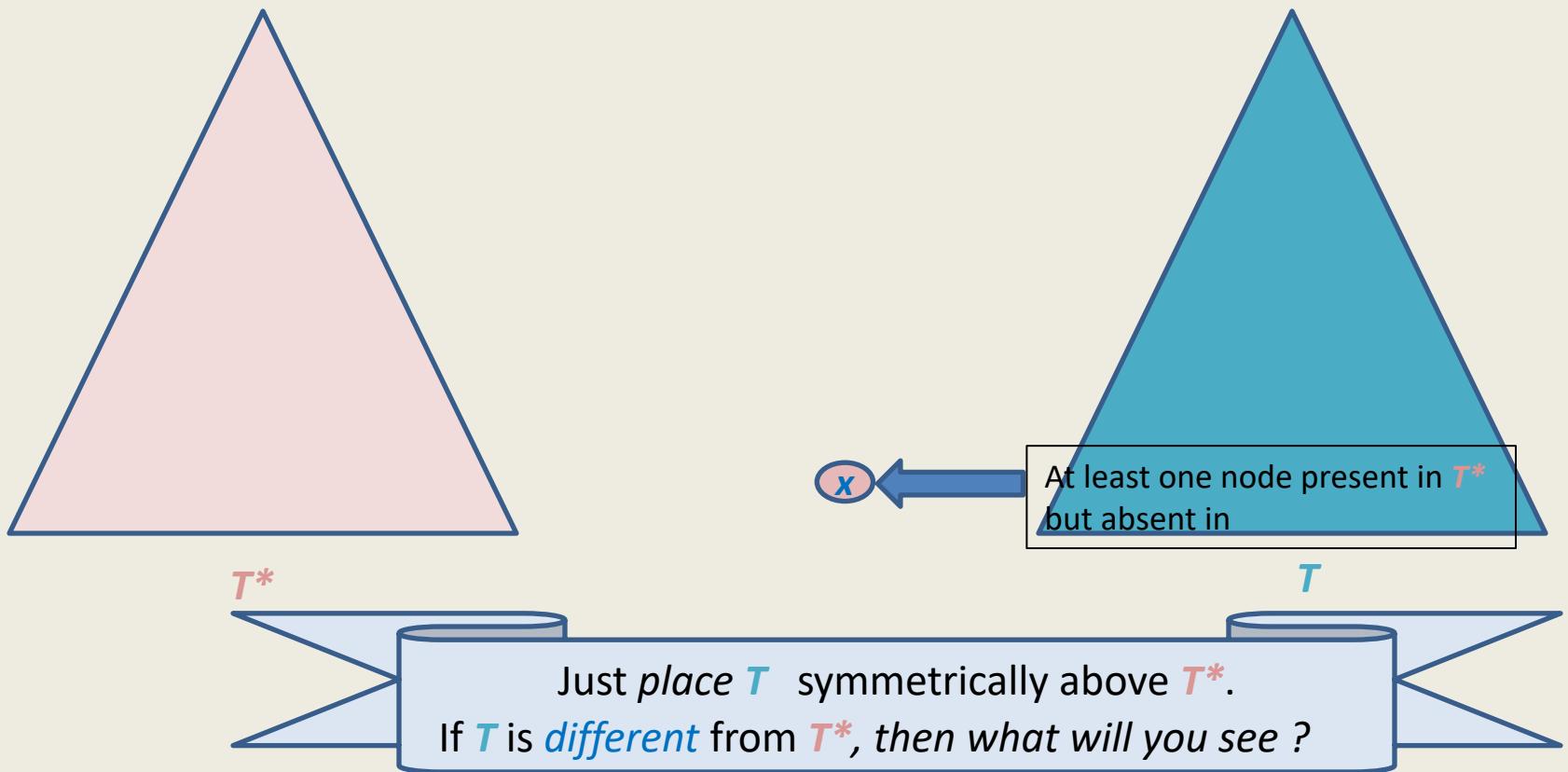
Let T be any other **complete** binary tree of height h different from T^* .

Question: How to show that T can not exist ?

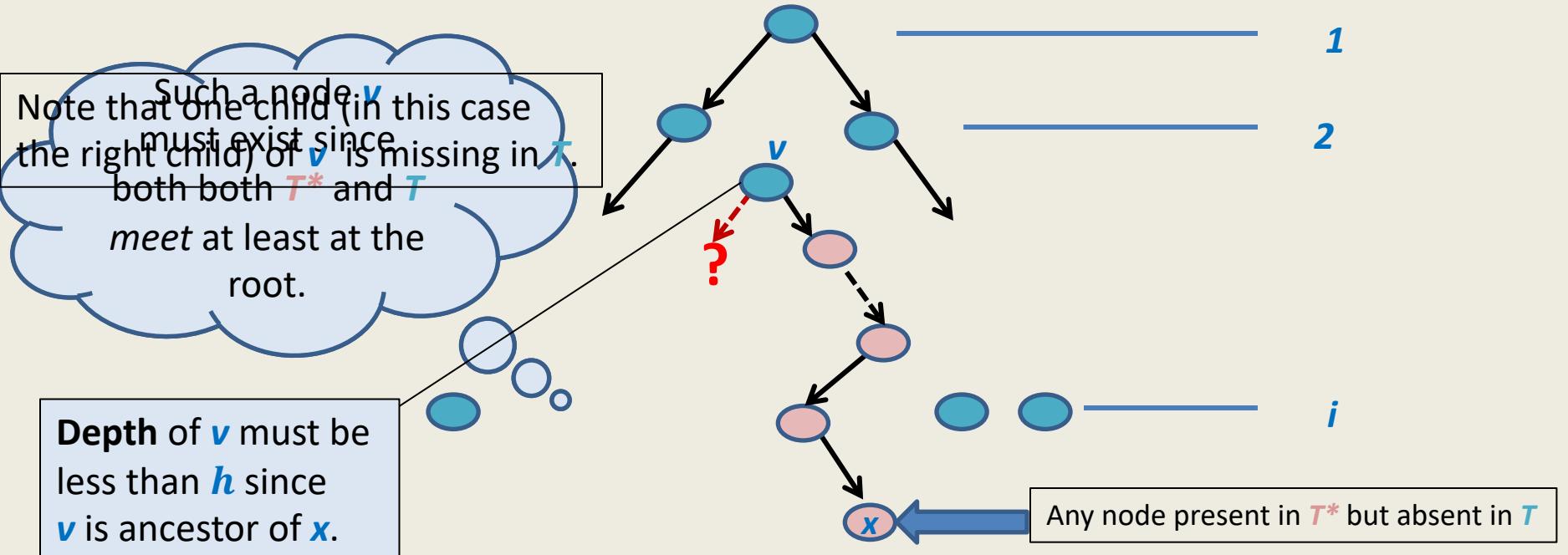


Watch the following slide carefully.

Uniqueness of a complete binary tree of height h



Uniqueness of a complete binary tree of height h



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 complete binary tree of height h

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

→ There exists a unique **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 $\geq h$

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

$$\rightarrow 2^h \leq 2n + 2$$

$$\rightarrow h \leq 1 + \log_2(n + 1)$$

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

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

a complete binary tree of height h

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

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

and

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

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 $O(\log_{4/3} n)$

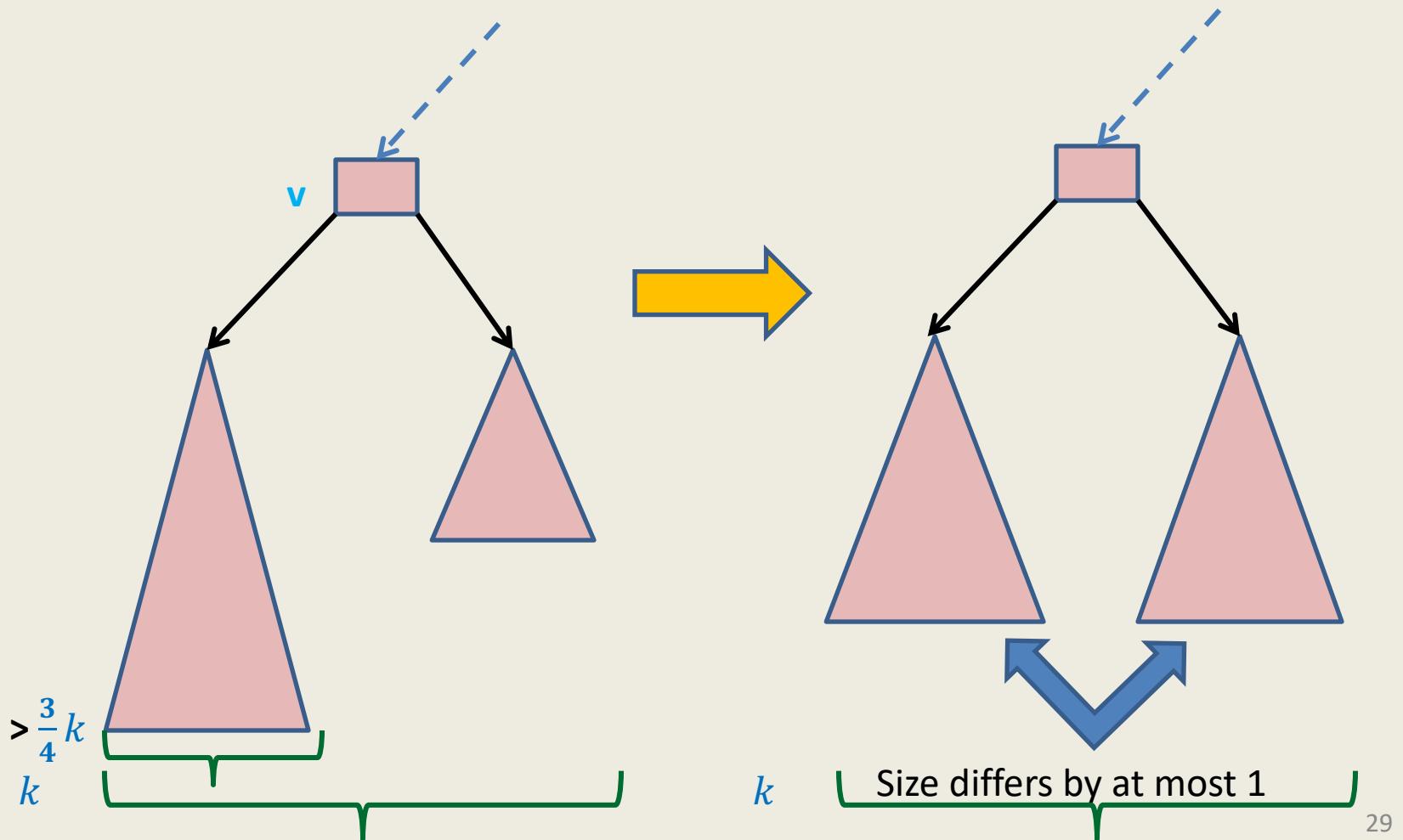
Nearly balanced Binary Search Tree

Maintaining under Insertion

Each node v in T maintains additional field $\text{size}(v)$ which is the number of nodes in the $\text{subtree}(v)$.

- Keep $\text{Search}(T, x)$ operation unchanged.
- Modify $\text{Insert}(T, x)$ operation as follows:
 - Carry out normal insert and update the size fields of nodes traversed.
 - If BST T ceases to be **nearly balanced** at any node v , transform $\text{subtree}(v)$ 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. (It was given as a Homework.)

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 $O(n \log n)$.

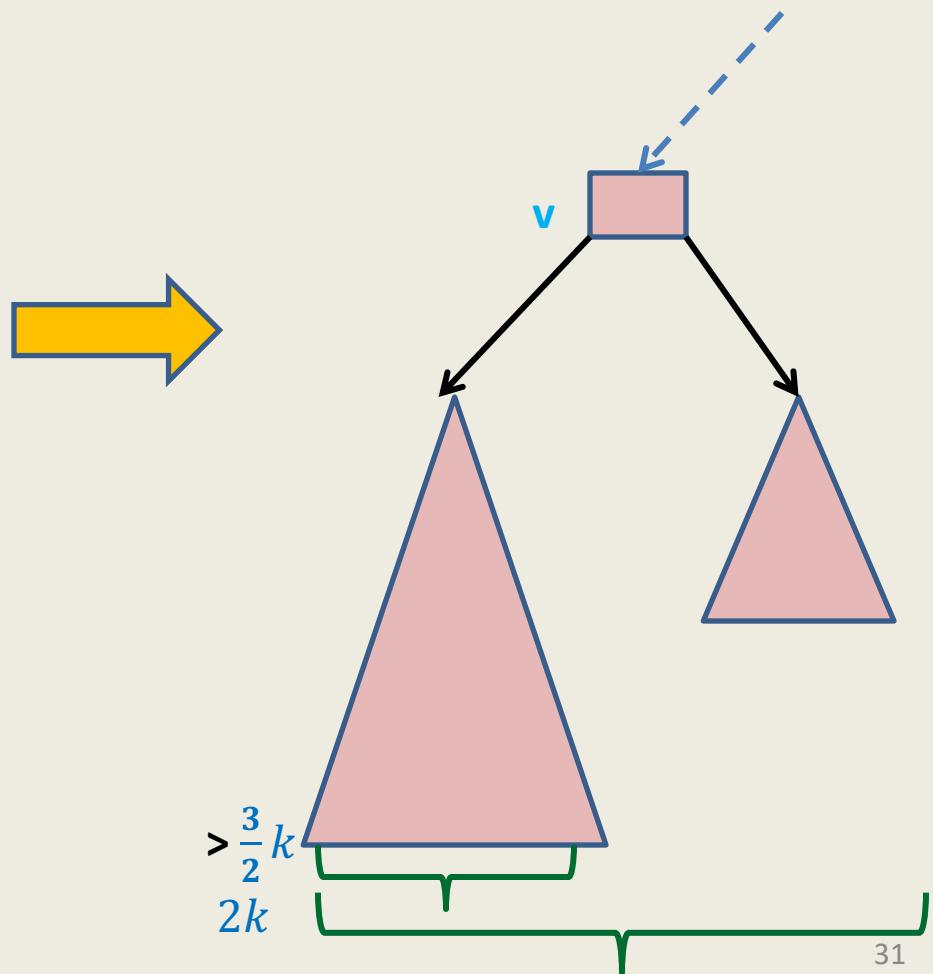
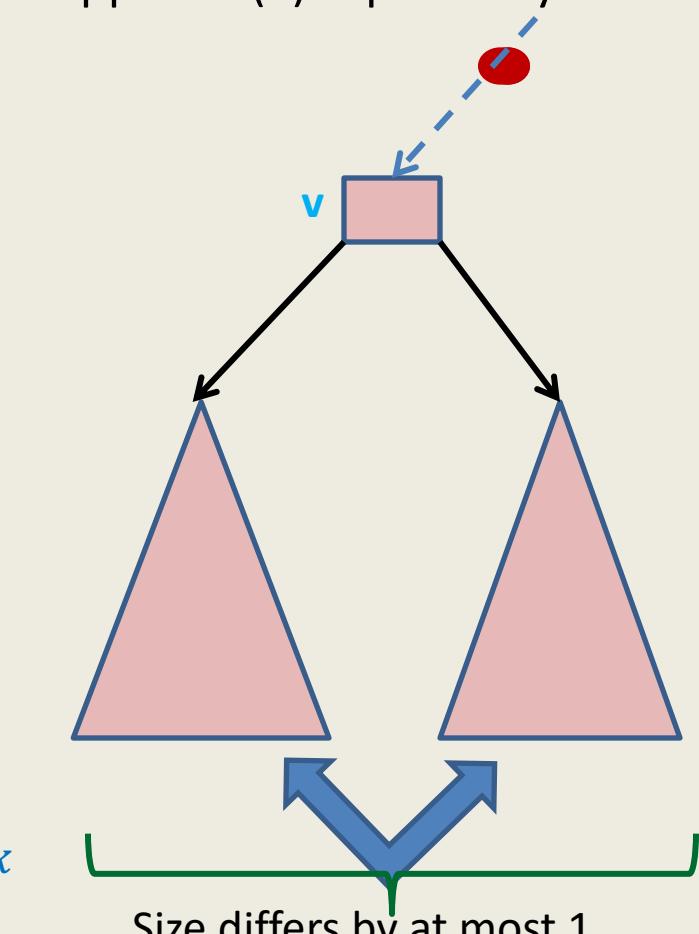
We shall now prove this theorem formally.

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

How many new elements 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

size(v) : no. of nodes in $T(v)$ at any moment.

For k th insertion,

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

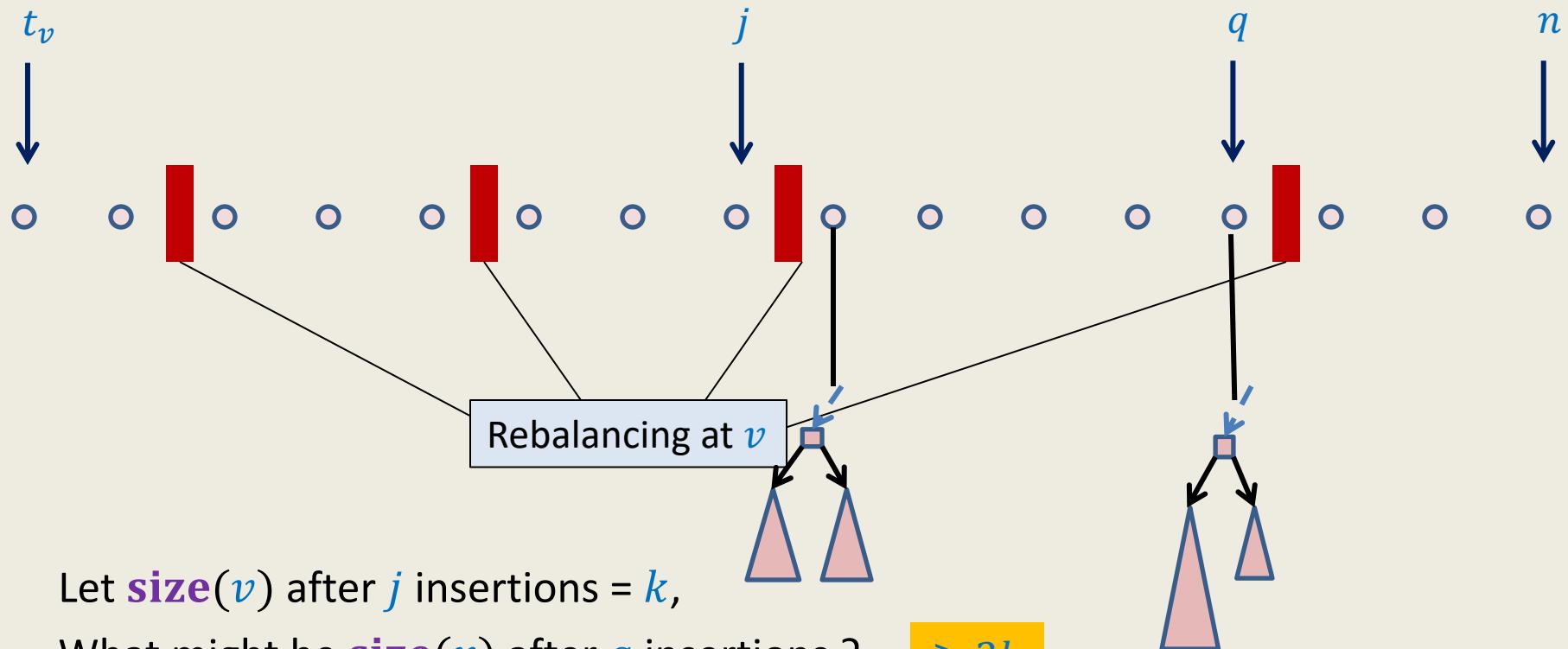
Question: For a nearly balanced BST, what is

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

This is because

- T , being nearly Balanced, has $O(\log k)$ height.
and
- an insertion can increase **size** field for only the nodes lying along a root to leaf path.

Journey of an element/node v during n insertions



Let $\text{size}(v)$ after j insertions = k ,

What might be $\text{size}(v)$ after q insertions ?

$$\geq 2k$$

Time complexity of rebalancing $T(v)$ after q th insertion = $O(k)$

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

→ Time complexity of rebalancing $T(v)$ after q th insertion = $O(\sum_{r=j+1}^q I_r(v))$

Time complexity of n insertions

For a vertex v ,

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

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”

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

Theorem:

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

Data Structures and Algorithms

(ESO207)

Lecture 20

Red Black tree (Final lecture)

- **9 types of operations**

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

Red Black tree

(Height Balanced BST)

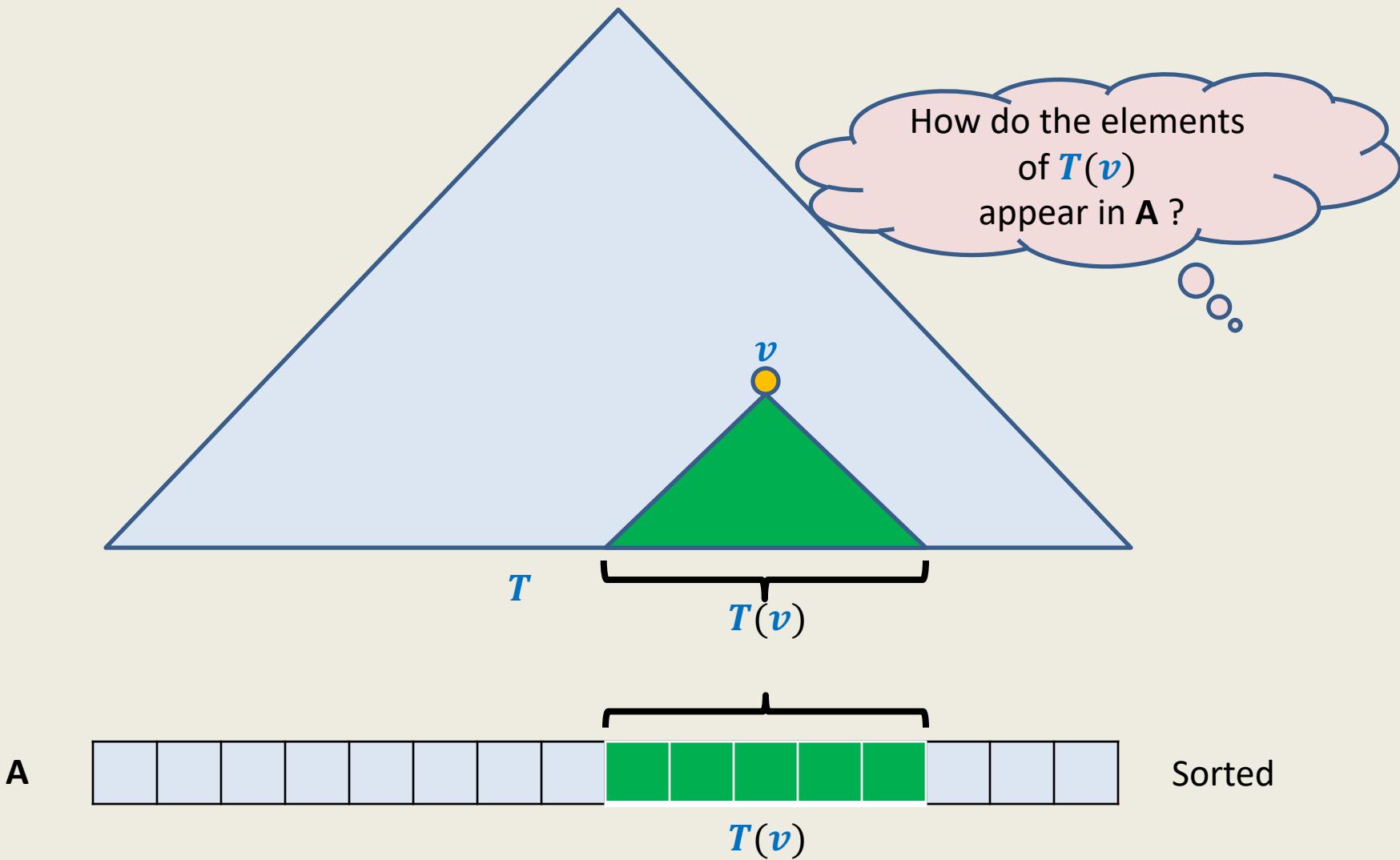
Operations you already know

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

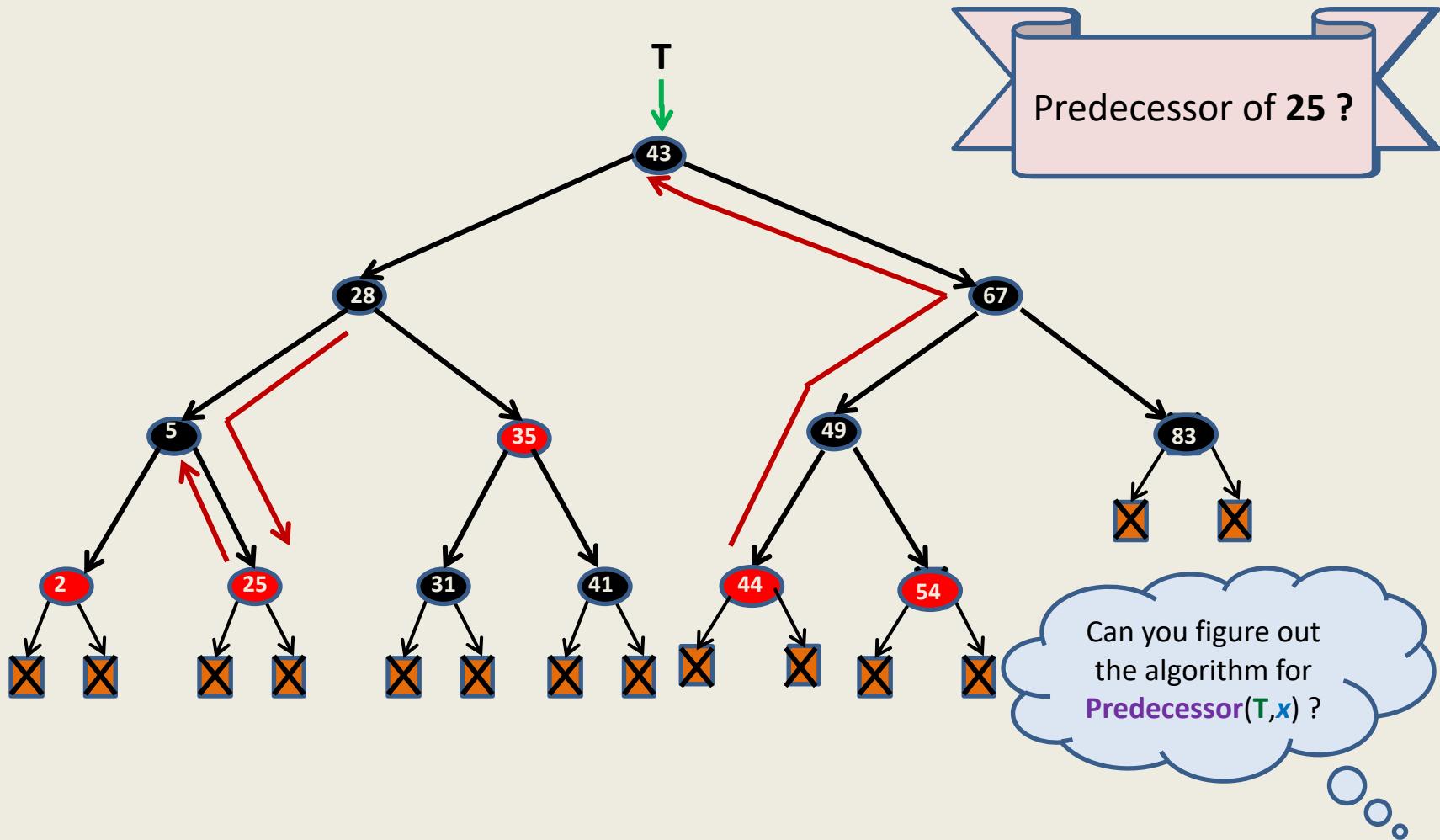
How well have you understood ?



Predecessor(T, x)

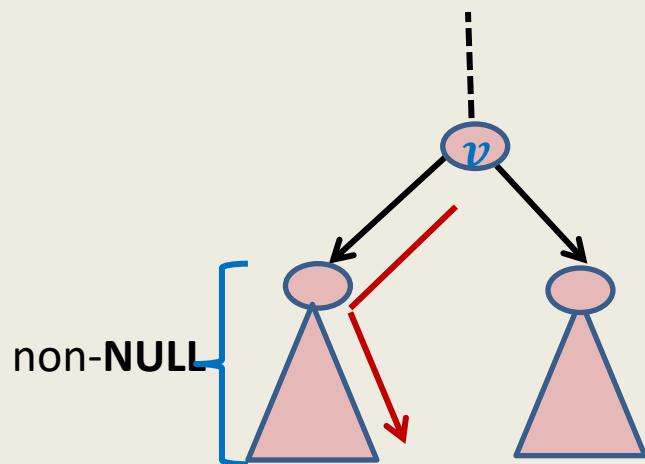
The **largest** element in T which is smaller than x

Predecessor(T, x)



Predecessor(T, x)

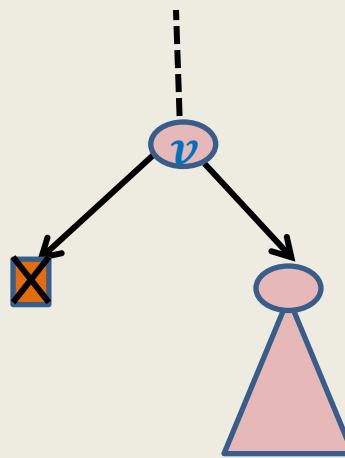
Let v be the **node** of T storing value x .



Case 1: $\text{left}(v) \neq \text{NULL}$, then $\text{Predecessor}(T, x)$ is Max(left(v))

Predecessor(T, x)

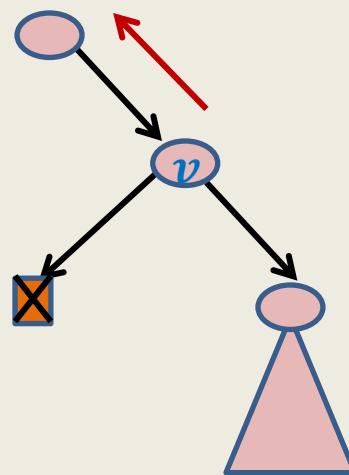
Let v be the **node** of T storing value x .



Case 2: $\text{left}(v) == \text{NULL}$, then $\text{Predecessor}(T, x)$ is ?

Predecessor(T, x)

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

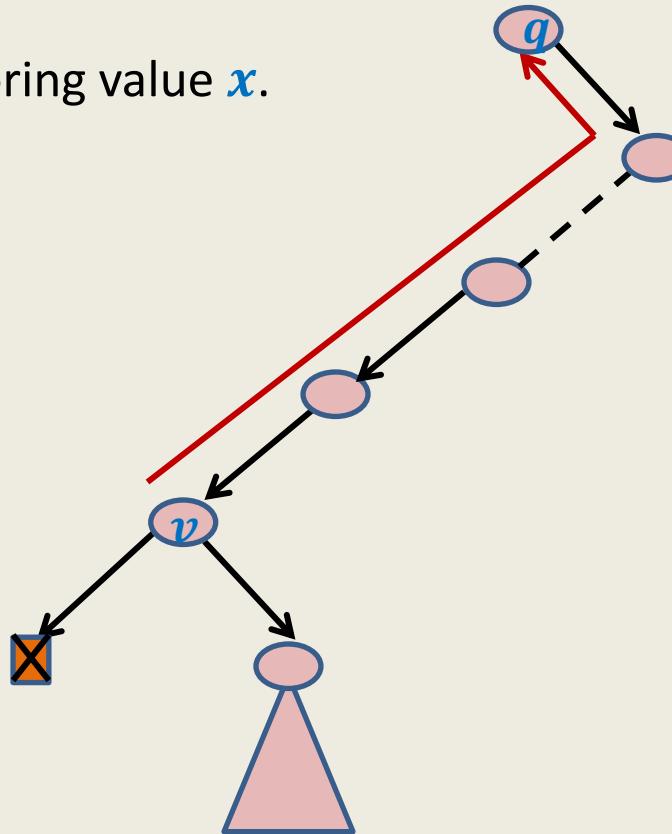


Case 2: $\text{left}(v) == \text{NULL}$, and v is **right child** of its **parent**

then Predecessor(T, x) is $\text{parent}(v)$

Predecessor(T, x)

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



Case 3: $\text{left}(v) == \text{NULL}$, 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 ($\text{left}(v) \neq \text{NULL}$) then return Max(left(v))

else

if ($v = \text{right}(\text{parent}(v))$) return parent(v)

else

{

while($v = \text{left}(\text{parent}(v))$

$v \leftarrow \text{parent}(v);$

return $\text{parent}(v)$;

}

}

Predecessor(T, x)

Predecessor(T, x)

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

If ($\text{left}(v) \neq \text{NULL}$) then return Max(left(v))

else

{ while($v = \text{left}(\text{parent}(v))$

$v \leftarrow \text{parent}(v);$

return $\text{parent}(v)$;

}

}

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)**

New operations

8. **SpecialUnion(T, T'):**
Given T and T' such that $T < T'$,
compute $T^* = T \cup T'$.

NOTE: T and T' don't exist after the union.
9. **Split(T, x):**
Split T into T' and T'' such that $T' < x < T''$.

A NOTATION

$T < T'$:

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

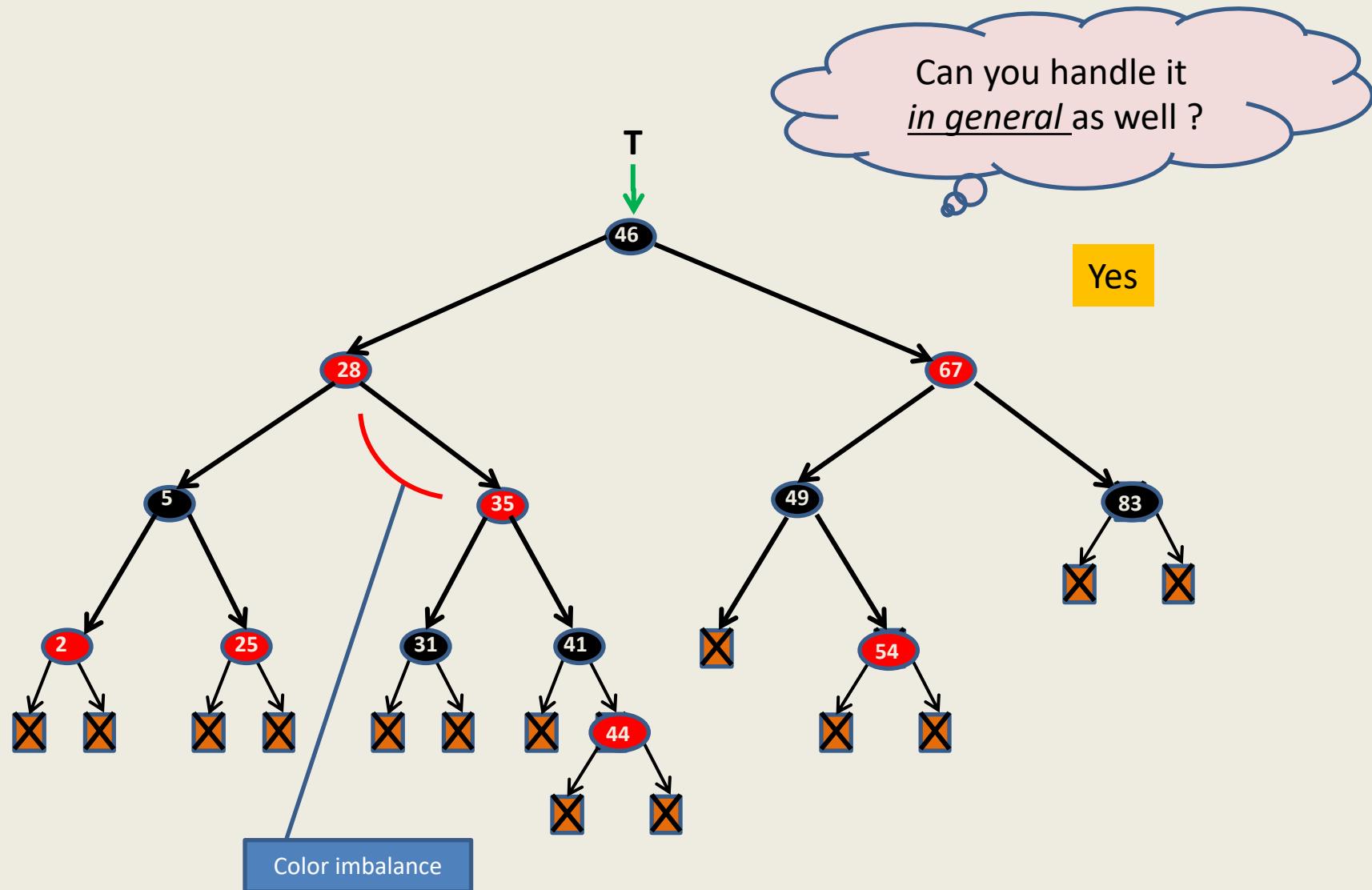


Every operation in $O(\log n)$ time.

Red-Black Tree

How well have you understood ?

Insertion in a red-black tree

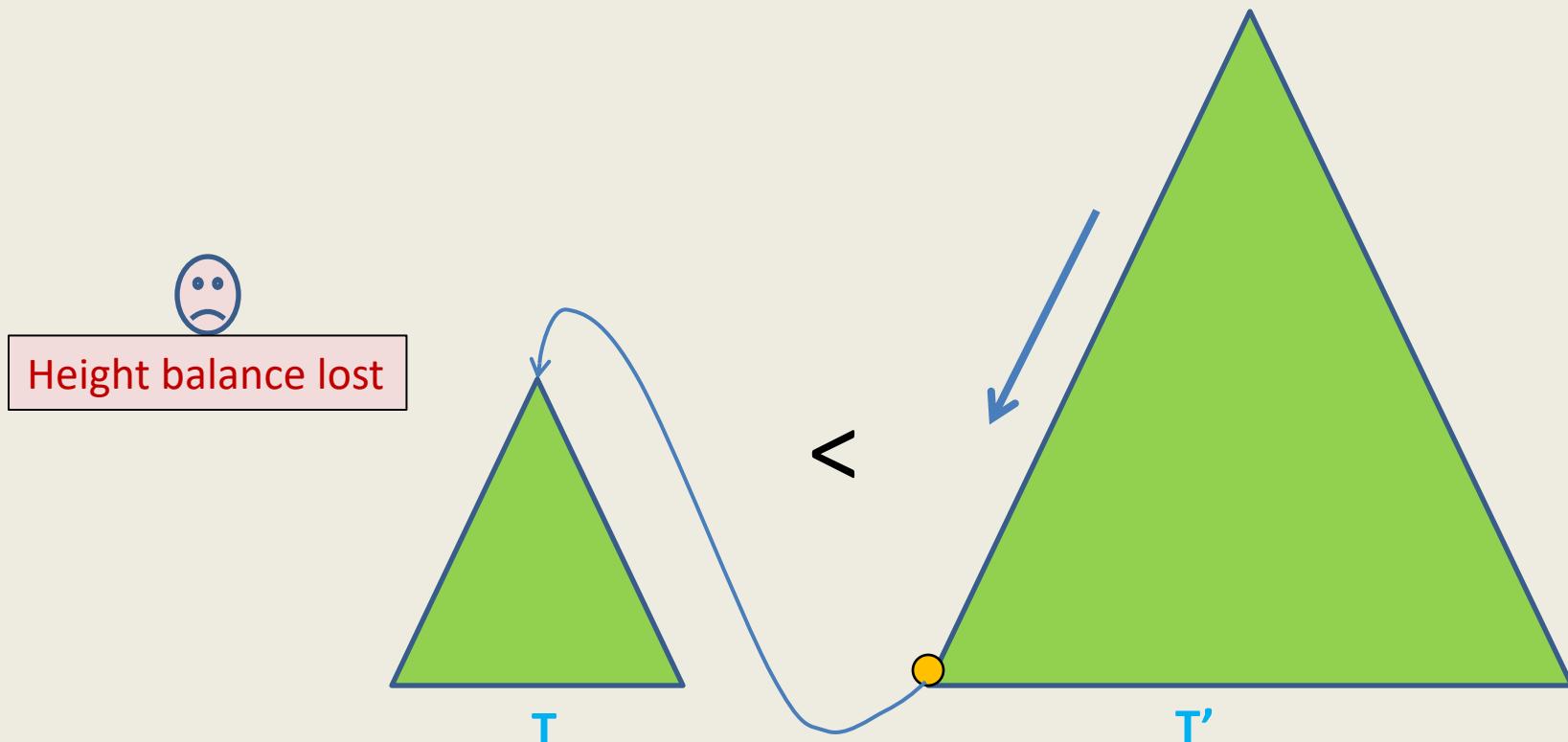


SpecialUnion(T, T')

Remember:

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

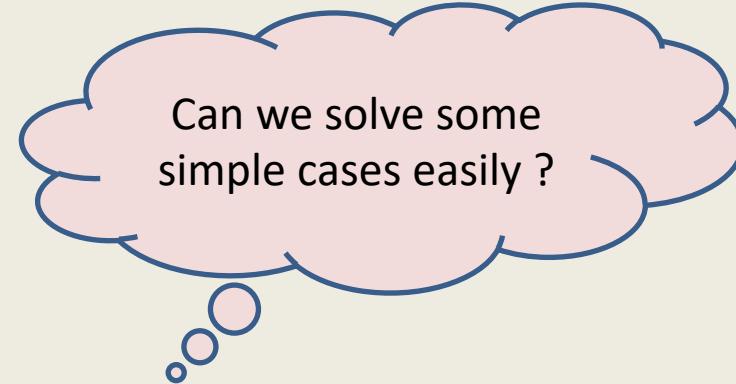
A trivial algorithm that does not work



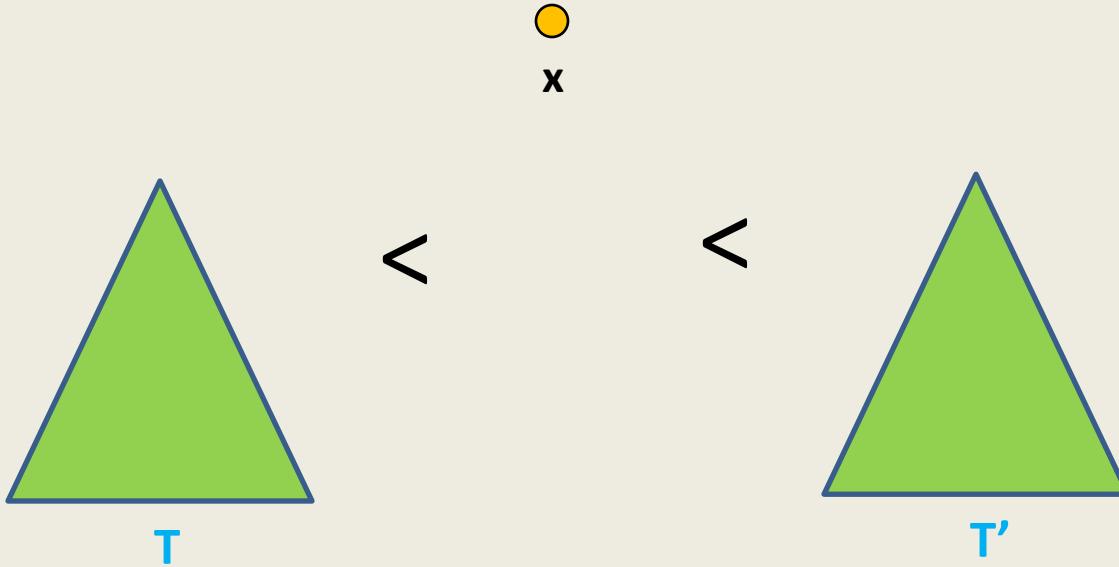
Time complexity: $O(\log n)$

Towards an $O(\log n)$ time for $\text{SpecialUnion}(T, T')$...

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



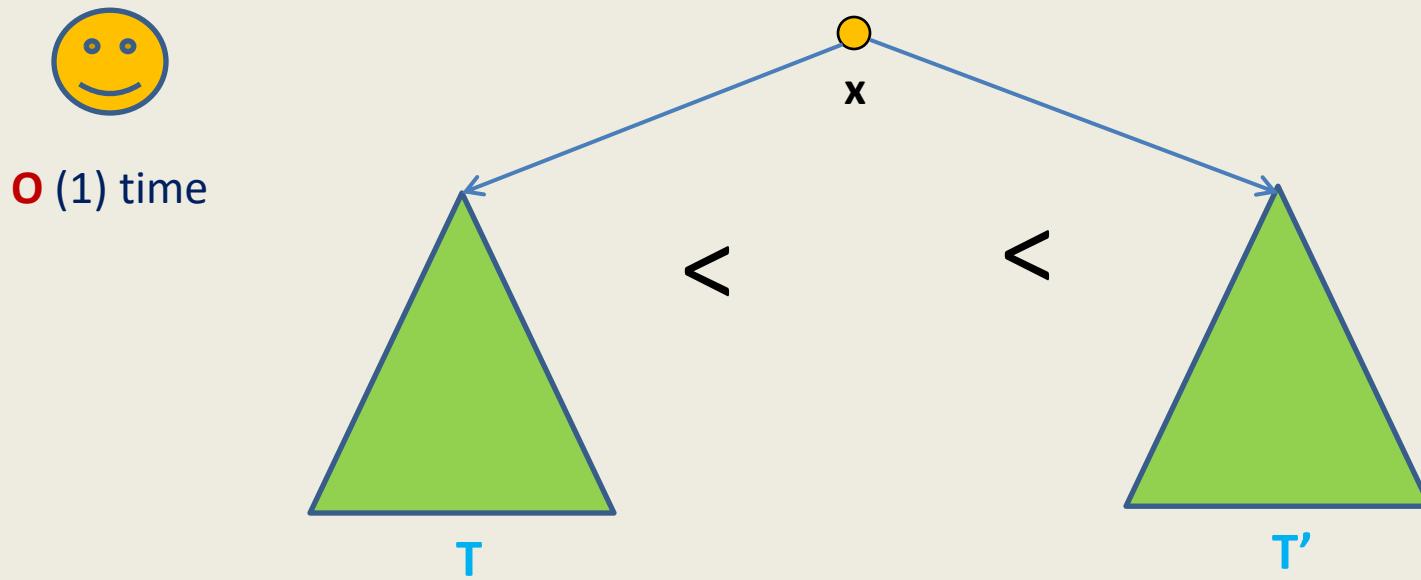
Simplifying the problem



Simplified problem:

Given two trees T, T' of same black height and a key x , such that $T < x < T'$, transform them into a tree $T^* = T \cup \{x\} \cup T'$

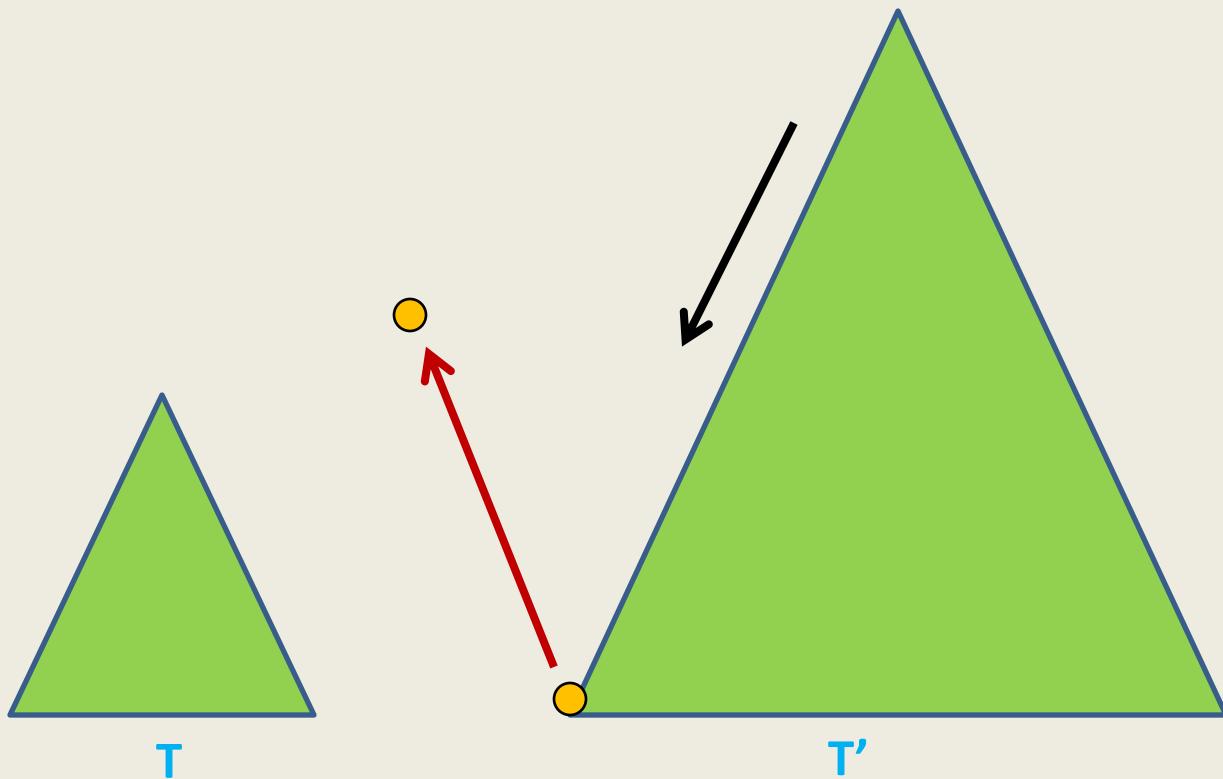
Solving the simplified problem



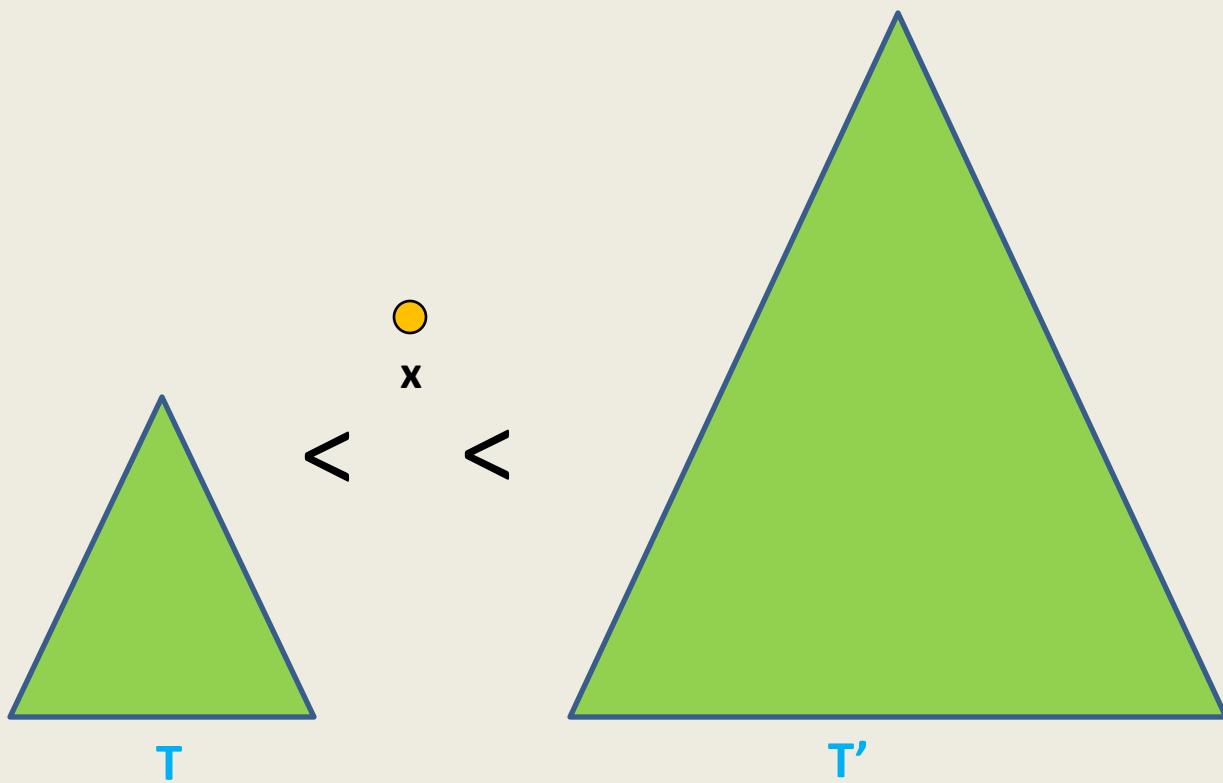
Simplified problem:

Given two trees T, T' of same black height
and a key x , such that $T < x < T'$,
transform them into a tree $T^* = T \cup \{x\} \cup T'$

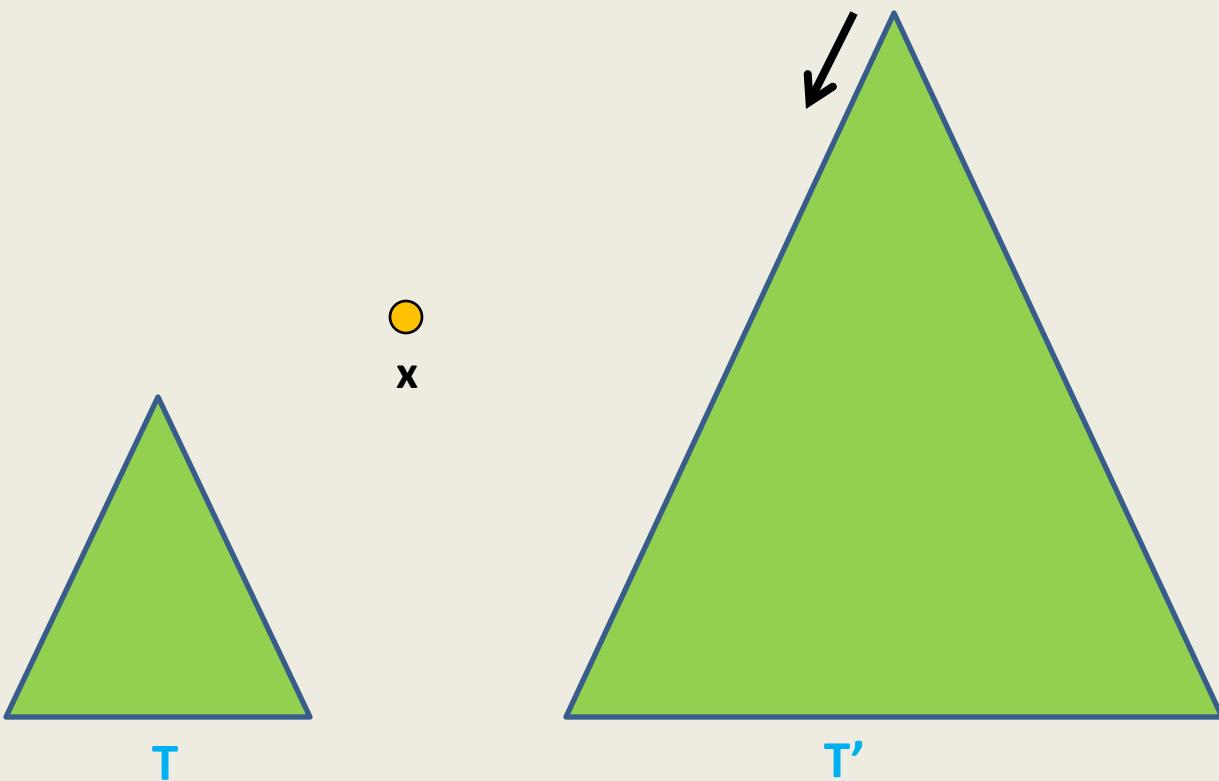
Extending the algorithm to the generic problem



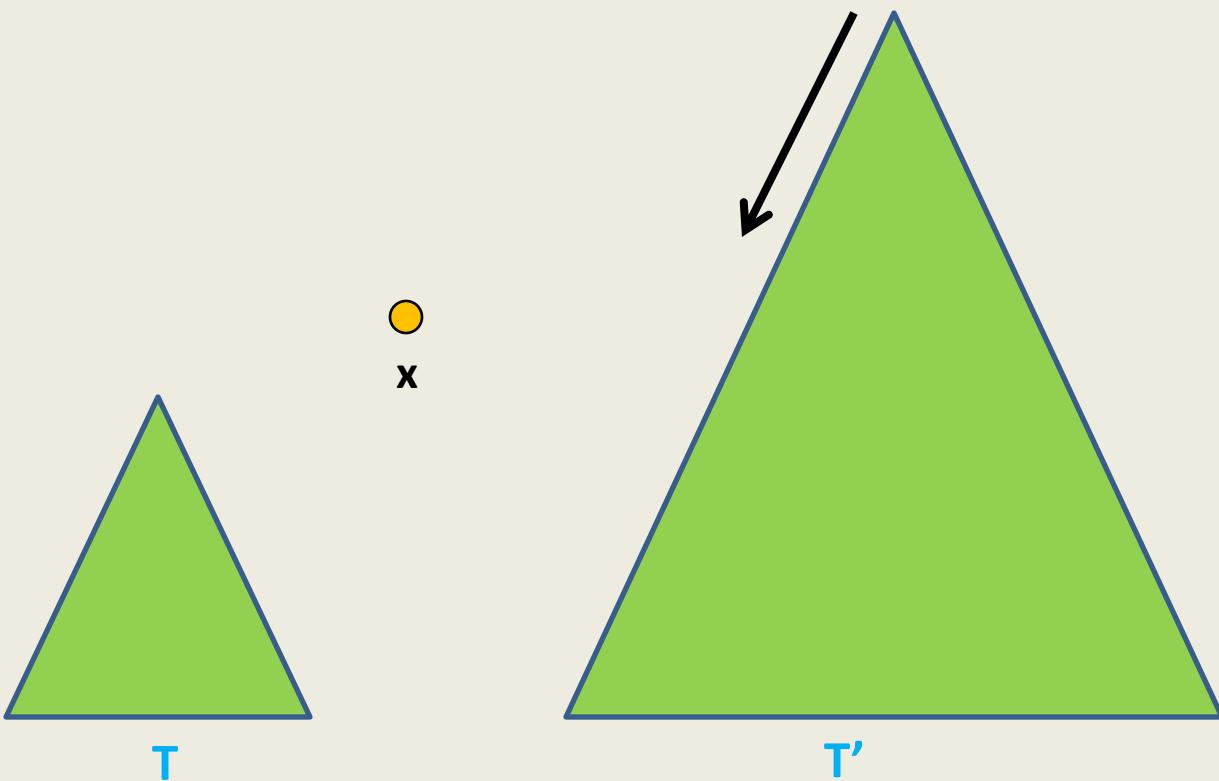
Extending the algorithm to the generic problem



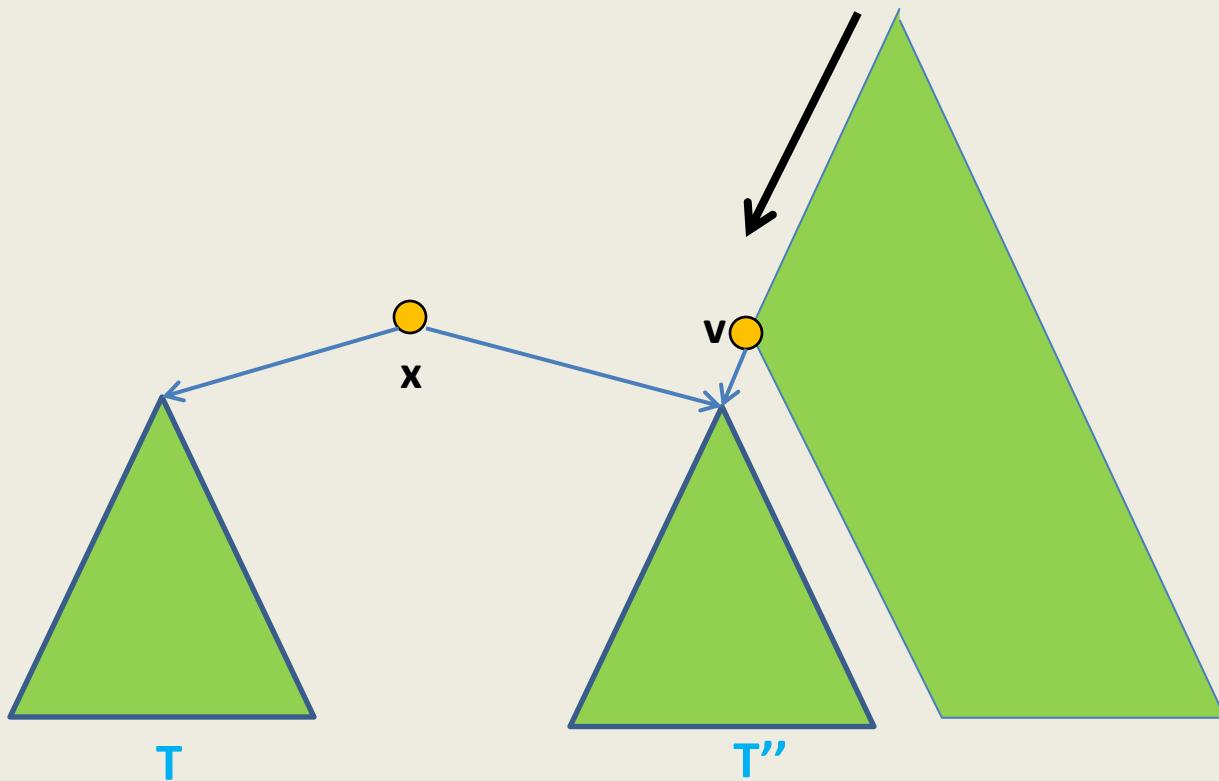
Extending the algorithm to the generic problem



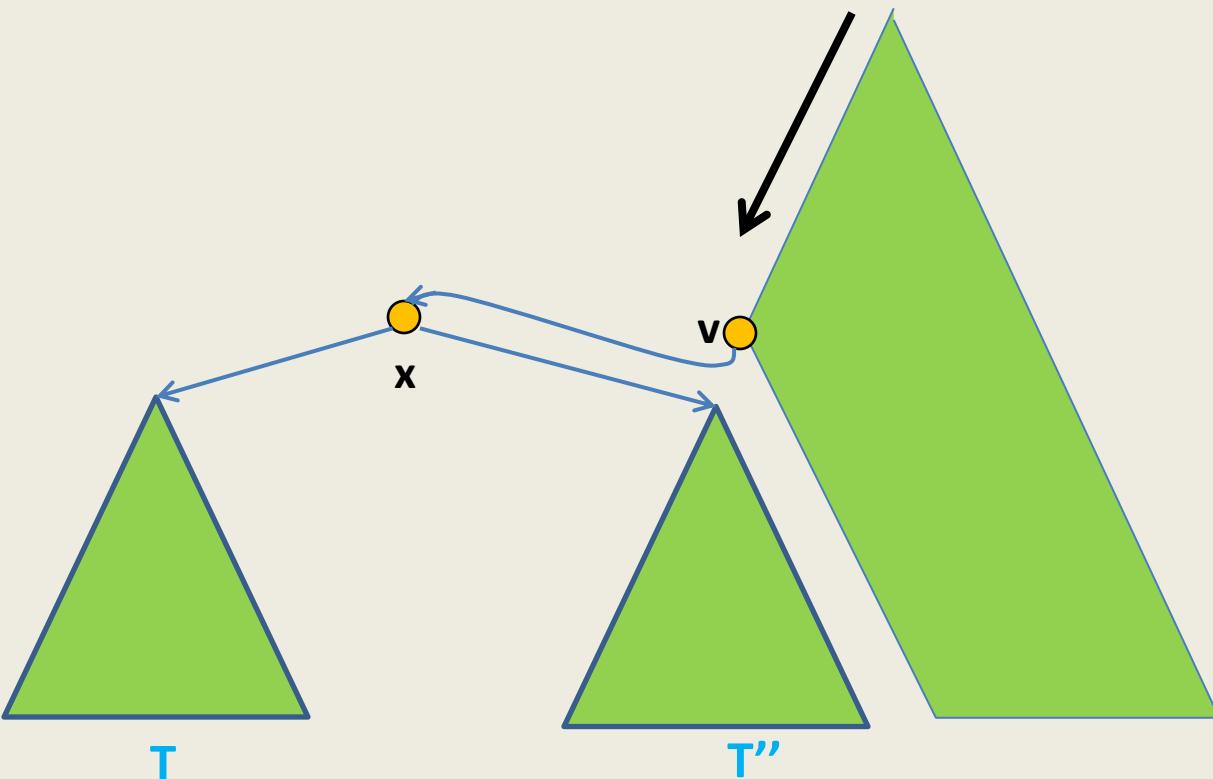
Extending the algorithm to the generic problem



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 $T \leq$ **black height** of T'

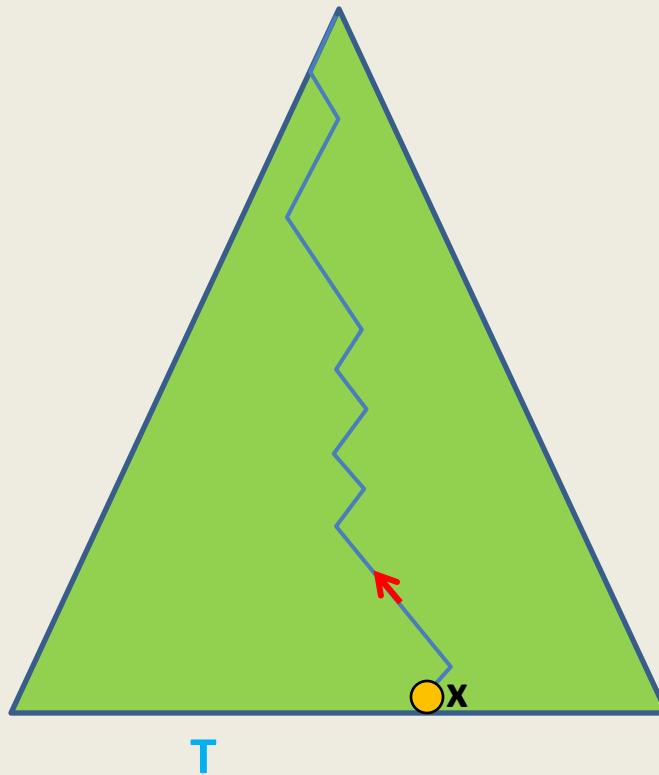
1. Keep following left pointer of T' until we reach a node v such that
 1. $\text{left}(v)$ is black
 2. The subtree T'' rooted at $\text{Left}(v)$ has black height same as that of T
2. $\text{left}(x) \leftarrow T;$
3. $\text{right}(x) \leftarrow T'';$
4. $\text{Color}(x) \leftarrow \text{red};$
5.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

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



Split(T,x)

Achieving $O(\log n)$ time for $\text{Split}(T,x)$



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