

Data Structures and Algorithms

(ESO207)

Lecture 1:

- An **overview** and **motivation** for the course
- some **concrete** examples.

Acknowledgment

Thanks to Prof Surender Baswana for allowing me to use and modify his lecture slides.

The website of the course

moodle.cse.iitk.ac.in



ESO207: Data Structures and Algorithms

Prerequisite of this course

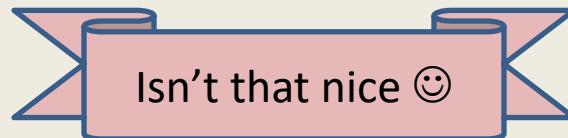
- A good command on Programming in C
 - Programs involving arrays
 - Recursion
 - Linked lists (**preferred**)
- **Fascination for solving Puzzles**

Salient features of the course

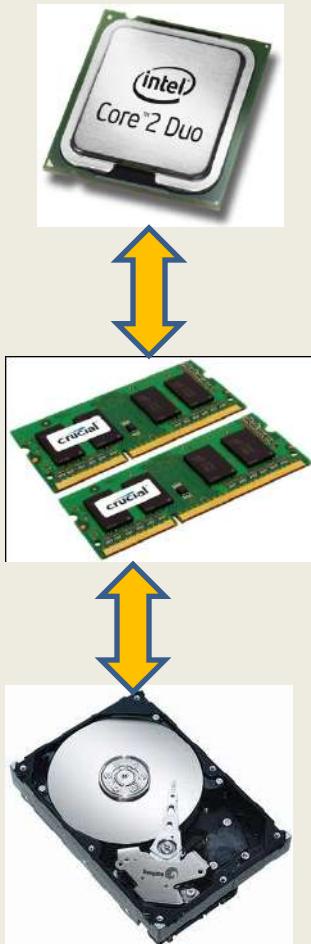
- **Every concept** **We** shall re-invent in the class itself.
- **Solving each problem** Through discussion in the class.

solution will emerge naturally if we ask
right set of questions
and then try to find their **answers**.

... so that finally it is a concept/solution derived by you
and not a concept from some scientist/book/teacher.



Let us open a desktop/laptop



A processor (CPU)

speed = few GHz

(a few **nanoseconds** to execute an instruction)

Internal memory (RAM)

size = a few GB (Stores a billion bytes/words)

speed = a few GHz(a few **nanoseconds** to read a byte/word)

External Memory (Hard Disk Drive)

size = a few tera bytes

speed : seek time = **milliseconds**

transfer rate= around **billion** bits per second

A simplifying assumption (for the rest of the lecture)

It takes around a few **nanoseconds** to execute an instruction.

(This assumption is well supported by the modern day computers)

EFFICIENT ALGORITHMS

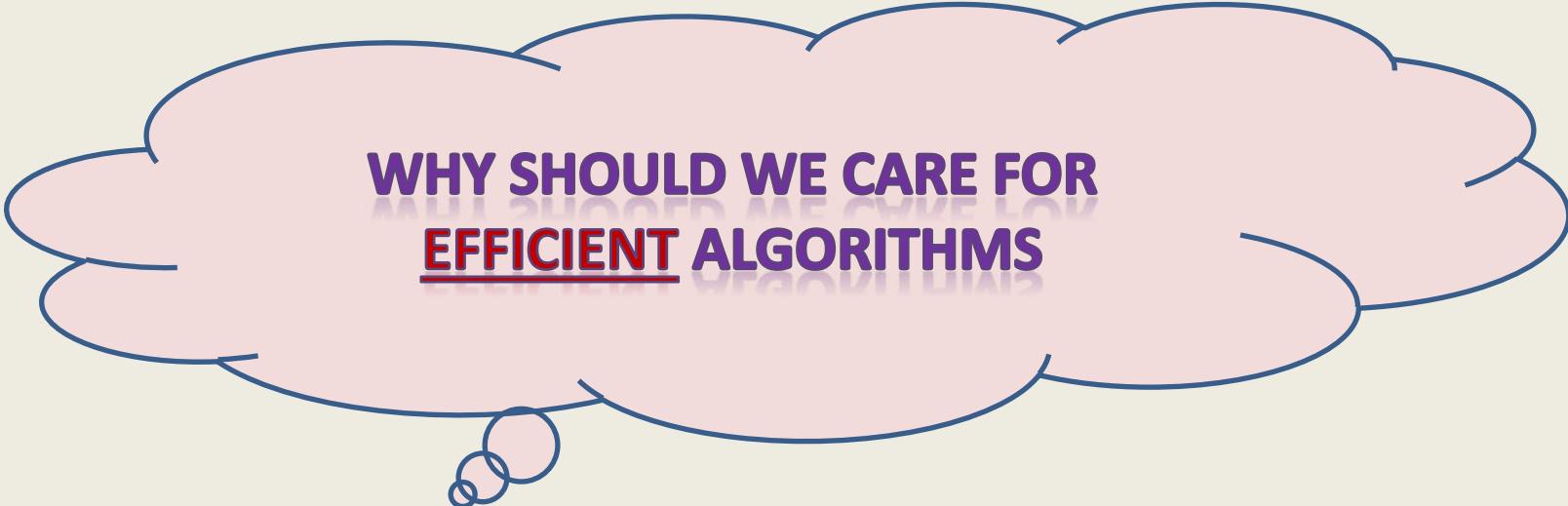
What is an algorithm ?

Definition:

A **finite sequence of well defined instructions**
required to solve a given computational problem.

A prime objective of the course:

Design of **efficient** algorithms



**WHY SHOULD WE CARE FOR
EFFICIENT ALGORITHMS**

WE HAVE PROCESSORS RUNNING AT GIGAHERTZ?

Revisiting problems from ESC101

Problem 1:

Fibonacci numbers

Fibonacci numbers

$$F(0) = 0;$$

$$F(1) = 1;$$

$$F(n) = F(n - 1) + F(n - 2) \text{ for all } n > 1;$$

$$F(n) \approx a \cdot b^n$$

An easy exercise : Using induction or otherwise, show that

$$F(n) > 2^{\frac{n-2}{2}}$$

Algorithms you must have implemented for computing $F(n)$:

- **Iterative**
- **recursive**

Iterative Algorithm for $F(n)$

IFib(n)

if $n=0$ **return** 0;

else if $n=1$ **return** 1;

else {

$a \leftarrow 0$; $b \leftarrow 1$;

For($i=2$ to n) **do**

 { $\text{temp} \leftarrow b$;

$b \leftarrow a+b$;

$a \leftarrow \text{temp}$;

 }

}

return b ;

Recursive algorithm for $F(n)$

Rfib(n)

```
{  if n=0 return 0;  
  else if n=1 return 1;  
    else return(Rfib(n-1) + Rfib(n-2))  
}
```

Homework 1

(compulsory)

Write a **C** program for the following problem:

Input: a number ***n***

n : **long long int** (64 bit integer).

Output: **F(*n*) mod 2014**

Time Taken	Largest <i>n</i> for Rfib	Largest <i>n</i> for IFib
1 minute		
10 minutes		
60 minutes		

Problem 2: Subset-sum problem

Input: An array **A** storing n numbers, and a number s

A	12	3	46	34	19	101	208	120	219	115	220
---	----	---	----	----	----	-----	-----	-----	-----	-----	-----

Output: Determine if there is a subset of numbers from **A** whose sum is s .

The fastest existing algorithm till date : $2^{n/2}$ instructions

- Time for $n = 100$ At least **an year**
- Time for $n = 120$ At least **1000 years**
on the fastest existing computer.

Problem 3:

Sorting

Input: An array **A** storing **n** numbers.

Output: Sorted **A**

A fact:

A significant fraction of the code of all the software is for sorting or searching only.

To sort **10 million** numbers on the present day computers

- **Selection sort** will take at least a few hours.
- **Merge sort** will take only a few seconds.
- **Quick sort** will take ??? .

How to design efficient algorithm for a problem ?

Design of **algorithms** and **data structures** is also
an Art



Requires:

- Creativity
- Hard work
- Practice
- Perseverance (most important)

Summary of Algorithms

- There are many practically relevant problems for which there does not exist any efficient algorithm till date ☺. (How to deal with them ?)
- Efficient algorithms are important for theoretical as well as practical purposes.
- Algorithm design is an art which demands a lot of creativity, intuition, and perseverance.
- More and more applications in real life require efficient algorithms
 - Search engines like **Google** exploits many clever algorithms.

THE DATA STRUCTURES

An Example

Given: a telephone directory storing telephone no. of **hundred million** persons.

Aim: to answer a sequence of **queries** of the form

“what is the phone number of a given person ?”.

Solution 1 :

Keep the directory in an array.

do **sequential search** for each query.

Time per query: around **1/10th** of a **second**

Solution 2:

Keep the directory in an array, and **sort it** according to names,

do **binary search** for each query.

Time per query: less than **100 nanoseconds**

Aim of a data structure ?

To store/organize a given data in the memory of computer so that each subsequent operation (query/update) can be performed quickly ?

Range-Minima Problem

A Motivating example
to realize the importance of data structures

Range-Minima Problem

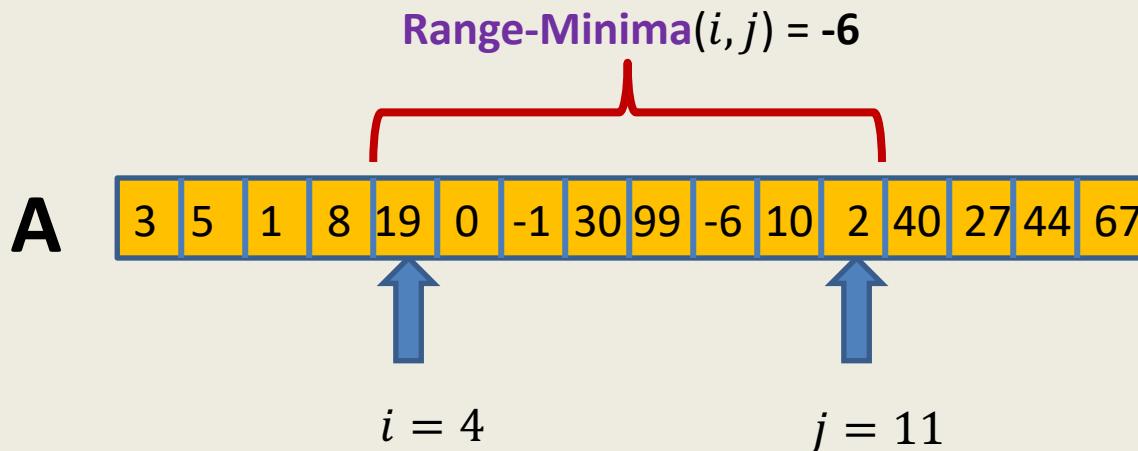
Given: an array **A** storing n numbers,

Aim: a data structure to answer a sequence of queries of the following type

Range-minima(i, j) : report the smallest element from $\mathbf{A}[i], \dots, \mathbf{A}[j]$

Let n = one million.

No. of queries = 10 millions



Range-Minima Problem

Applications:

- Computational geometry
- String matching
- As an efficient subroutine in a variety of algorithms

(we shall discuss these problems sometime in this course or the next level course CS345)

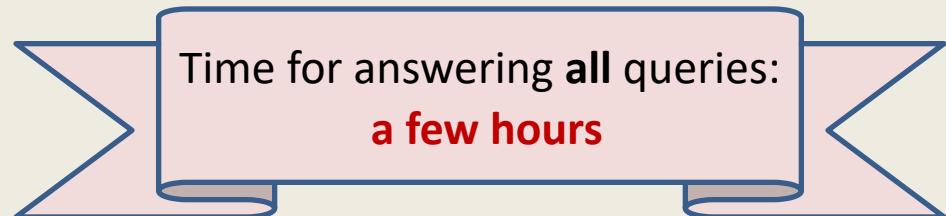
Range-Minima Problem

Solution 1:

Answer each query in a brute force manner using **A** itself.

Range-minima-trivial(*i,j*)

```
{   temp <- i+1;  
    min <- A[i];  
    While(temp <= j)  
    {   if (min > A[temp])  
        min <- A[temp];  
        temp <- temp+1;  
    }  
    return min  
}
```

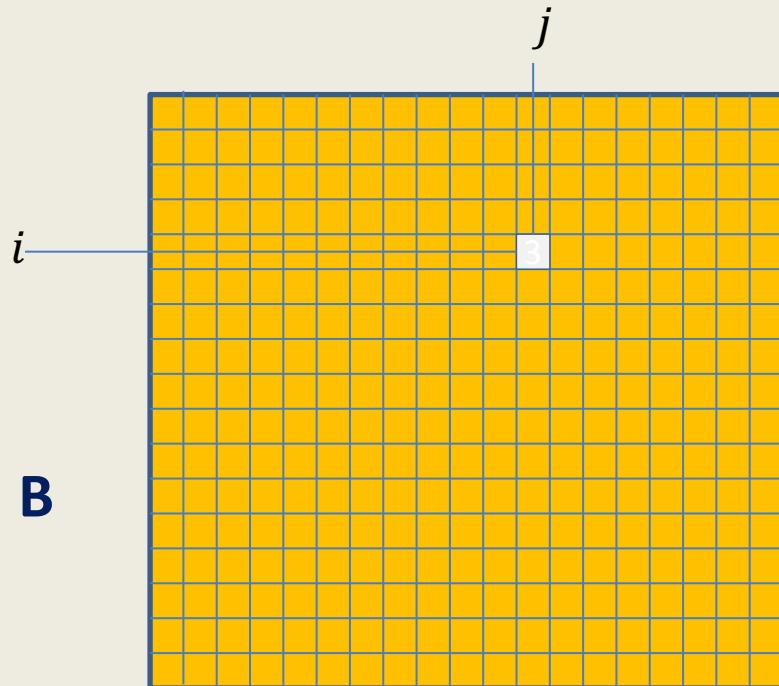


Time taken to answer a query: **few milliseconds**

Range-Minima Problem

Solution 2:

Compute and store answer for each possible query in a $n \times n$ matrix \mathbf{B} .



$\mathbf{B}[i][j]$ stores the smallest element from $\mathbf{A}[i], \dots, \mathbf{A}[j]$

Space : roughly n^2 words.

**Solution 2 is
Theoretically efficient but
practically impossible**

Size of \mathbf{B} is too large to be kept in RAM. So we shall have to keep most of it in the Hard disk drive. Hence it will take a few milliseconds per query.

Range-Minima Problem

Question: Does there exist a data structure for Range-minima which is

- **Compact**
(nearly the same size as the input array A)
- **Can answer each query efficiently ?**
(a few **nanoseconds** per query)

Homework 2: Ponder over the above question.

(we shall solve it soon)

Data structures to be covered in this course

Elementary Data Structures

- Array
- List
- Stack
- Queue

Hierarchical Data Structures

- Binary Heap
- Binary Search Trees

Augmented Data Structures

Most fascinating and powerful data structures

- Look forward to working with all of you to make this course enjoyable.
- This course will be light in contents (no formulas)
But it will be very demanding too.
- In case of any difficulty during the course,
just drop me an email without any delay.
I shall be happy to help ☺

Data Structures and Algorithms

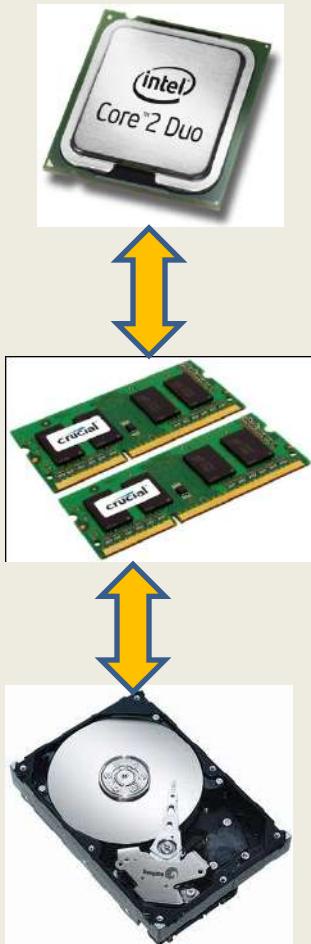
(ESO207)

Lecture 2:

- Model of computation
- Efficient algorithm for $F(n) \bmod m$.

RECAP OF THE 1ST LECTURE

Current-state-of-the-art computer



A processor (CPU)

speed = few GHz

(a few **nanoseconds** to execute an instruction)

Internal memory (RAM)

size = a few GB (Stores few million bytes/words)

speed = a few GHz(a few **nanoseconds** to read a byte/word)

External Memory (Hard Disk Drive)

size = a few tera bytes

speed : seek time = **milliseconds**

transfer rate= a **billion** bytes per second

Motivation

- for Efficient algorithms
 - Subset sum problem
 - Sorting
- for Efficient Data Structures
 - Telephone Directory
 - Range Minima

Homework 1

(compulsory)

Write a **C** program for the following problem:

Input: a number ***n***

n : **long long int** (64 bit integer).

Output: **F(*n*) mod 2014**

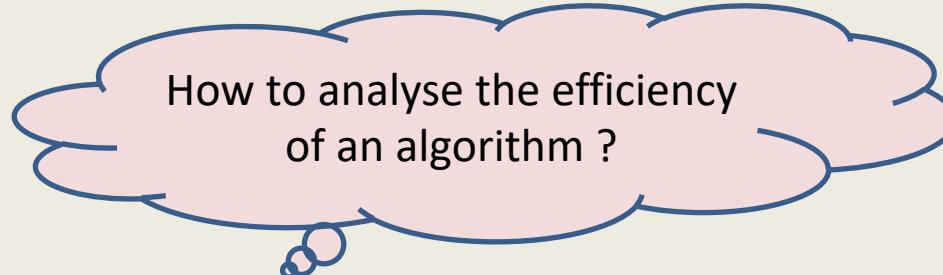
Time Taken	Largest <i>n</i> for Rfib	Largest <i>n</i> for IFib
1 minute	48	4.5×10^9
10 minutes	53	4.5×10^{10}
60 minutes	58	2.6×10^{11}

Processor: 2.7 GHz

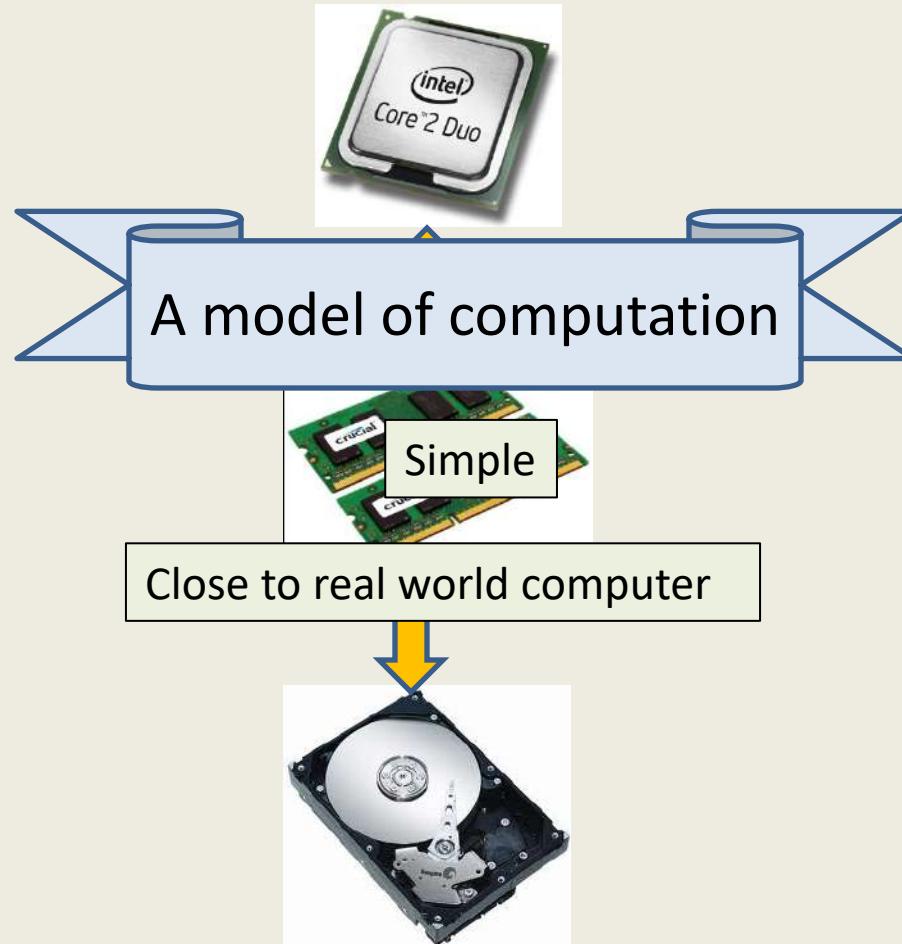
Here are the values obtained by executing the program on a typical computer.

Inferences from the Homework

- Inference for **RFib** algorithm
 - Too slow 😞
- Inference for **Ifib** algorithm :
 - Faster than **Rfib**
 - But not a solution for the problem 😞
- Efficiency of an algorithm does matter
- Time taken to solve a problem may vary depending upon the algorithm used



Current-state-of-the-art Computer

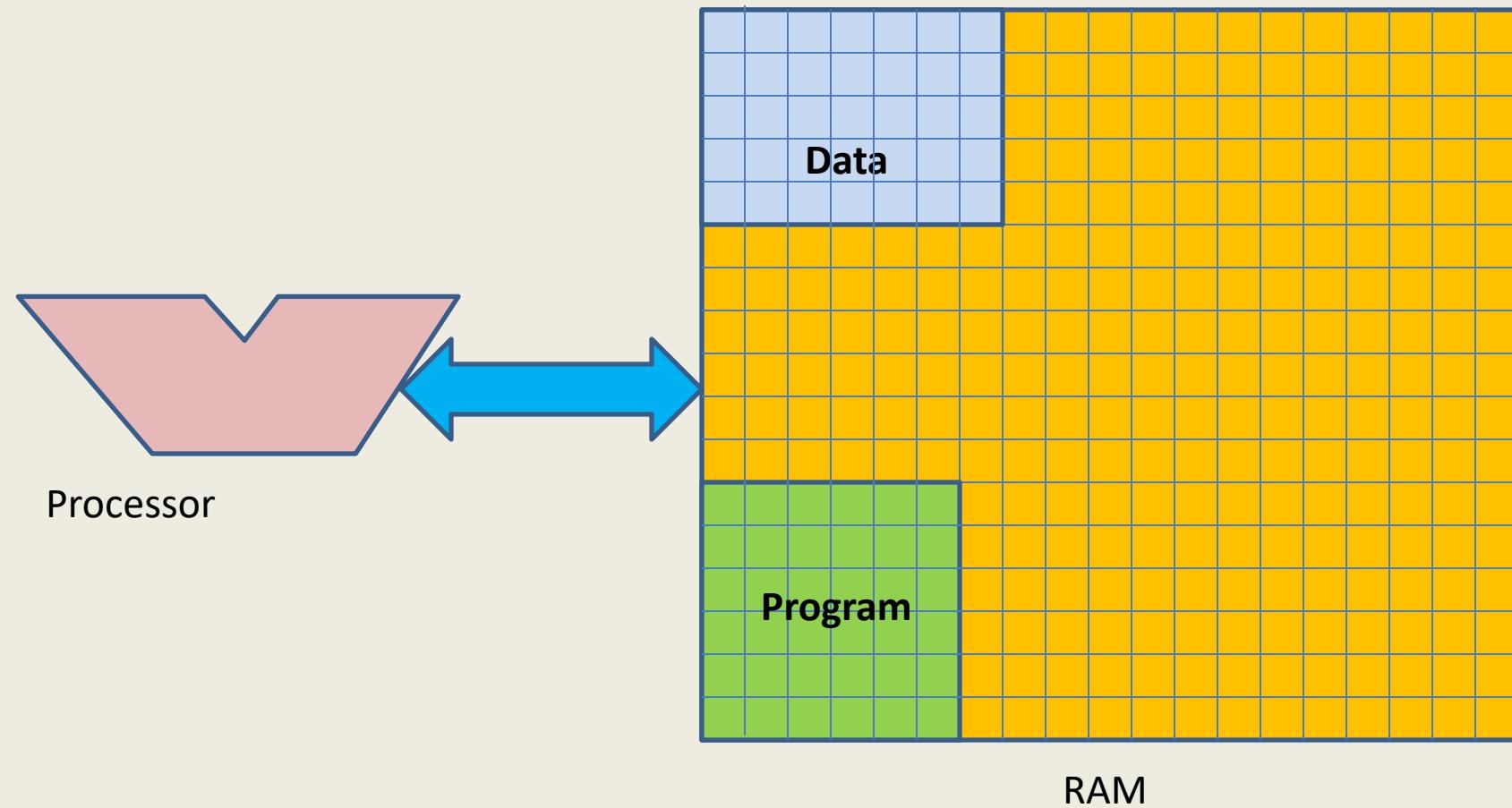


word RAM :

a model of computation

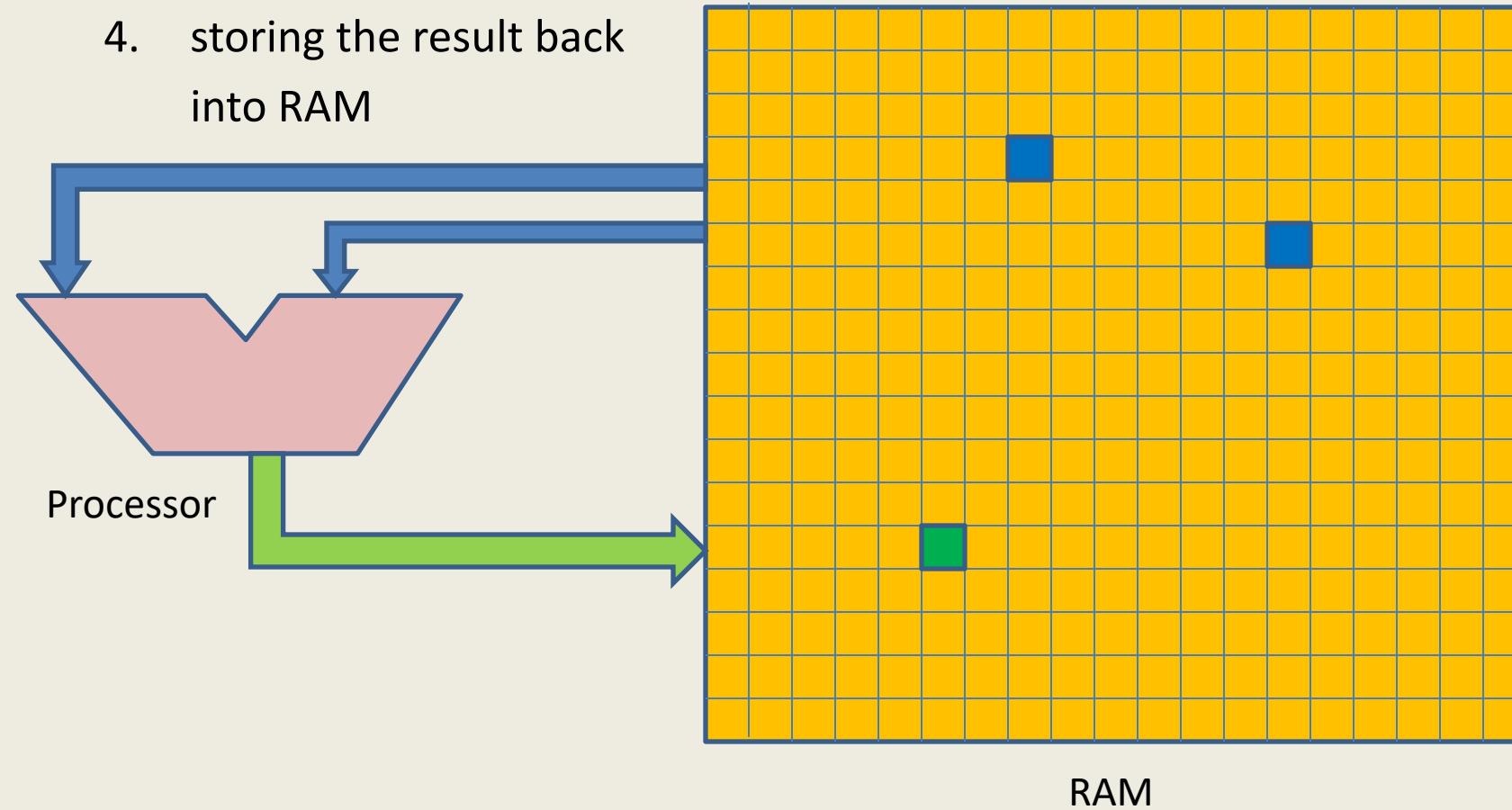
word RAM :

a model of computation



How is an instruction executed?

1. Decoding instruction
2. fetching the operands
3. performing arithmetic/logical operation
4. storing the result back
into RAM



→ Each instruction takes a few cycles (click ticks) to get executed.

word RAM model of computation: Characteristics

- Word is the basic storage unit of RAM. Word is a collection of few bytes.
- 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.
- Data as well as Program reside fully in RAM.
- Each arithmetic or logical operation (+,-,* ,/,or, xor,...) involving a constant number of words takes a constant number of cycles (steps) by the CPU.

Efficiency of an algorithm

Question: How to measure time taken by an algorithm ?

- Number of instructions taken in **word RAM** model.

What about the influence of so many other factors?

We shall judge the influence, if any, of these parameters through experiments.

Variation in the time of various instructions

Architecture : 32 versus 64

Due to Compiler

Operating system

Who knows, these factors might have little or negligible impact on most of algorithms. ☺

Homework 1 from Lecture 1

Computing $F(n) \bmod m$

Iterative Algorithm for $F(n) \bmod m$

IFib(n, m)

if $n = 0$ return 0;

else if $n = 1$ return 1;

else { $a \leftarrow 0; b \leftarrow 1;$

 For($i = 2$ to n) do

 { $\text{temp} \leftarrow b;$

$b \leftarrow a + b \bmod m;$

$a \leftarrow \text{temp};$

 }

}

return b ;

Let us calculate the number of instructions executed by **IFib(n, m)**

Total number of instructions=

$$4+3(n-1)+1 \approx 3n$$

} 4 instructions

n-1 iterations

} 3 instructions per iteration

the final instruction

Recursive algorithm for $F(n) \bmod m$

RFib(n,m)

```
{   if  $n = 0$  return 0;  
    else if  $n = 1$  return 1;  
    else return((RFib( $n - 1,m$ ) + RFib( $n - 2,m$ ) ) mod  $m$ )  
}
```

Let $G(n)$ denote the number of instructions executed by $\text{RFib}(n,m)$

- $G(0) = 1;$
- $G(1) = 2;$
- For $n > 1$

$$G(n) = G(n - 1) + G(n - 2) + 4$$

Observation 1: $G(n) > F(n)$ for all n ;

$$\rightarrow G(n) > 2^{(n-2)/2} !!!$$

Algorithms for $F(n) \bmod m$

- # instructions by **Recursive** algorithm **RFib(n)**: $> 2^{\frac{n-2}{2}}$
(exponential in n)
- # instructions by **Iterative** algorithm **IFib(n)**: $3n$
(linear in n)



None of them works for entire range of **long long int n** and **int m**

Question: Can we compute $F(n) \bmod m$ quickly ?

How to compute $F(n) \bmod m$ quickly ?

... need some better insight ...

A warm-up example

How good are your programming skills ?

Compute $x^n \bmod m$

Problem: Given three integers x , n , and m , compute $x^n \bmod m$.

$$x^n = \begin{cases} 1 & \text{if } n = 0 \\ x \times x^{n-1} & \text{otherwise} \end{cases}$$

Power(x, n, m)

```
If ( $n = 0$ ) return 1;  
else {  
    temp  $\leftarrow$  Power( $x, n - 1, m$ );  
    temp  $\leftarrow$  ( $temp \times x$ ) mod  $m$  ;  
    return temp;  
}
```

4 instructions
excluding the
Recursive call

Compute $x^n \bmod m$

Problem: Given three integers x , n , and m , compute $x^n \bmod m$.

$$x^n = \begin{cases} 1 & \text{if } n = 0 \\ x \times x^{n-1} & \text{otherwise} \end{cases}$$

Power(x, n, m)



Power($x, n - 1, m$);



Power($x, n - 2, m$);



Power($x, 0, m$)

No. of instructions executed by **Power**(x, n, m) = $4n$

Compute $x^n \bmod m$

Problem: Given three integers x , n , and m , compute $x^n \bmod m$.

$$x^n = \begin{cases} 1 & \text{if } n = 0 \\ x^{n/2} \times x^{n/2} & \text{if } n \text{ is even} \\ x^{n/2} \times x^{n/2} \times x & \text{if } n \text{ is odd} \end{cases}$$

Power(x, n, m)

If ($n = 0$) return 1;

else {

$temp \leftarrow \text{Power}(x, n/2, m);$

$temp \leftarrow (temp \times temp) \bmod m;$

if ($n \bmod 2 = 1$) $temp \leftarrow (temp \times x) \bmod m;$

return $temp$;

}

5 instructions
excluding the
Recursive call

Compute $x^n \bmod m$

Problem: Given three integers x , n , and m , compute $x^n \bmod m$.

$$x^n = \begin{cases} 1 & \text{if } n = 0 \\ x^{n/2} \times x^{n/2} & \text{if } n \text{ is even} \\ x^{n/2} \times x^{n/2} \times x & \text{if } n \text{ is odd} \end{cases}$$

$\text{Power}(x, n, m)$



$\text{Power}(x, n/2, m)$



$\text{Power}(x, n/4, m)$



$\text{Power}(x, 0, m)$

No. of instructions executed by $\text{Power}(x, n, m) = 5 \log_2 n$

Efficient Algorithm for $F(n) \bmod m$

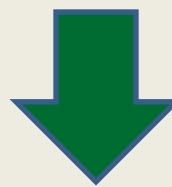
Idea 1

Question: Can we express $F(n)$ as a^n for some constant a ?

Unfortunately **no**.

Idea 2

$$\begin{pmatrix} F(n) \\ F(n-1) \end{pmatrix} = \begin{pmatrix} 1 & ? \\ 1 & 0 \end{pmatrix} \times \begin{pmatrix} F(n-1) \\ F(n-2) \end{pmatrix}$$



Unfolding the RHS of
this equation, we get ...

$$\begin{pmatrix} F(n) \\ F(n-1) \end{pmatrix} = \begin{pmatrix} 1 & n-1 \\ 1 & 0 \end{pmatrix} \times \begin{pmatrix} 1 \\ 0 \end{pmatrix}$$

A clever algorithm for $F(n) \bmod m$

Clever-algo-Fib(n, m)

```
{       $A \leftarrow \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix};$            ← 4 instructions
       $B \leftarrow A^{n-1} \bmod m;$ 
       $C \leftarrow B \times \begin{pmatrix} 1 \\ 0 \end{pmatrix};$        ← 6 instructions
      return  $C[1];$  // the first element of vector  $C$  stores  $F(n) \bmod m$ 
}
```

Question: How to compute $\begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}^{n-1}$ efficiently ?

Answer :

Inspiration from Algorithm for $x^n \bmod m$

A clever algorithm for $F(n) \bmod m$

Let A be a 2×2 matrix.

- If n is even,
$$A^n = A^{n/2} \times A^{n/2}$$
- If n is odd,
$$A^n = A^{n/2} \times A^{n/2} \times A$$

Question: How many instructions are required to multiply two 2×2 matrices ?

Answer: 12

Question: Number of instructions for computing $\begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}^{n-1} \bmod m$?

Answer : $35 \log_2 (n - 1)$

Question: Number of instructions in **New-algo-Fib**(n, m)

Answer: $35 \log_2 (n - 1) + 11$

Three algorithms

Algorithm for $F(n) \bmod m$	No. of Instructions
$\text{RFib}(n, m)$	$> 2^{(n-2)/2}$
$\text{IterFib}(n, m)$	$3n$
$\text{Clever_Algo_Fib}(n, m)$	$35 \log_2 (n - 1) + 11$

A red circle highlights the ' $>$ ' symbol in the first row's value, and a red line with a question mark extends from it to the right.

- Which algorithm is the best ?
- What is the exact base of the exponent in the running time of RFib ?
- Are we justified in ignoring the influence of so many other parameters ?
(Variation in the time of instructions/Architecture/Code optimization/...)
- How close to reality is the RAM model of computation ?

Find out yourself !

Data Structures and Algorithms

(ESO207)

Lecture 3:

- Time complexity, Big “O” notation
- Designing Efficient Algorithm
 - Maximum sum subarray Problem

Three algorithms

Algorithm for $F(n) \bmod m$	No. Instructions in RAM model
$\text{RFib}(n, m)$	$> 2^{(n-2)/2}$
$\text{IterFib}(n, m)$	$3n$
$\text{Clever_Algo_Fib}(n, m)$	$35 \log_2 (n - 1) + 11$



- Which algorithm turned out to be the best experimentally ?

Lesson 1

from Assignment 1 ?

Time taken by algorithm
in real life

No. of instructions executed by
algorithm in **RAM** model

Proportional to

May be different for different input?

Dependence on input

Time complexity of an algorithm

Definition:

the **worst case** number of instructions executed

as a **function** of the **input size** (or a parameter defining the input size)

The number of **bits/bytes/words**

Examples to illustrate **input size**:

Problem	Input size
Computing $F(n) \bmod m$ for <u>any positive integers</u> n and m	$\log_2 n + \log_2 m$ bits
Whether an array storing j numbers (<u>each stored in a word</u>) is sorted ?	j words
Whether a $n \times m$ matrix of numbers (<u>each stored in a word</u>) contains “14” ?	nm words

Homework: What is the time complexity of **Rfib**, **IterFib**, **Clever-Algo-Fib** ?

Example:

Whether an array A storing j numbers (each stored in a word) is sorted ?

IsSorted(A)

```
{   i<-1;  
    flag<- true;  
    while(i < j and flag==true)  
    {  
        If (A[i]< A[i - 1])  flag<-false ;  
        i <- i + 1;  
    }  
    Return flag ;  
}
```

1 time

1 time

$j - 1$ times in the worst case

1 time

Time complexity = $2j + 1$

Example:

Time complexity of matrix multiplication

Matrix-mult($C[n, n], D[n, n]$)

```
{  for  $i = 0$  to  $n - 1$            ←  $n$  times
    {    for  $j = 0$  to  $n - 1$        ←  $n$  times
        {       $M[i, j] \leftarrow 0;$ 
            for  $k = 0$  to  $n - 1$ 
            {               $M[i, j] \leftarrow M[i, j] + C[i, k] * D[k, j];$    } $n + 1$  instructions
            }
        }
    }
Return  $M$                                 ← 1 time
}
```

Each element of the matrices occupies one **word** of RAM.

Time complexity = $n^3 + n^2 + 1$

Lesson 2 learnt from Assignment 1 ?

Algorithm for $F(n) \bmod m$	No. of Instructions
RFib(n, m)	$> 2^{(n-2)/2}$
IterFib(n, m)	$3n$
Clever_Algo_Fib(n, m)	$100 \log_2 (n - 1) + 1000$

Question: What would have been the outcome if

$$\text{No. of instructions of Clever_Algo_Fib}(n, m) = 100 \log_2 (n - 1) + 1000$$

Answer: Clever_Algo_Fib would still be the fastest algorithm for large value of n .

COMPARING EFFICIENCY OF ALGORITHMS

Comparing efficiency of two algorithms

Let **A** and **B** be two algorithms to solve a given problem.

Algorithm **A** has time complexity : $2 n^2 + 125$

Algorithm **B** has time complexity : $5 n^2 + 67 n + 400$

Question: Which algorithm is more efficient ?

Obviously **A** is more efficient than **B**

Comparing **efficiency** of two algorithms

Let **A** and **B** be two algorithms to solve a given problem.

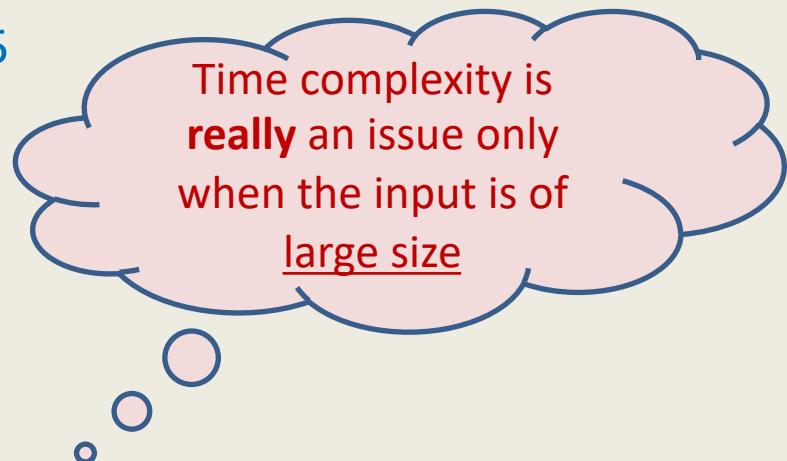
Algorithm **A** has time complexity : $2 n^2 + 125$

Algorithm **B** has time complexity : $50 n + 125$

Question: Which one would you prefer based on the efficiency criteria ?

Answer : **A** is more efficient than **B** for $n < 25$

B is more efficient than **A** for $n > 25$



Time complexity is
really an issue only
when the input is of
large size

Rule 1

Compare the **time complexities** of two algorithms for
asymptotically large value of input size only

Comparing efficiency of two algorithms

Algorithm **B** with time complexity $50n + 125$

is certainly more efficient than

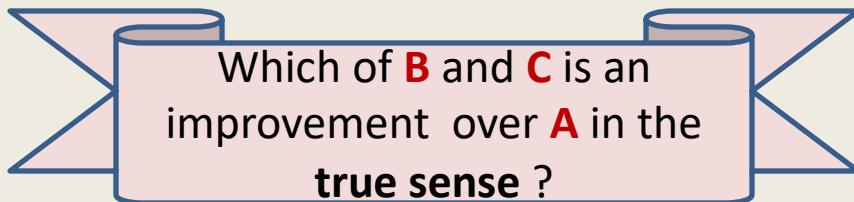
Algorithm **A** with time complexity : $2n^2 + 125$

A judgment question for you !

Algorithm **A** has time complexity $f(n) = 5n^2 + n + 1250$

Researchers have designed two new algorithms **B** and **C**

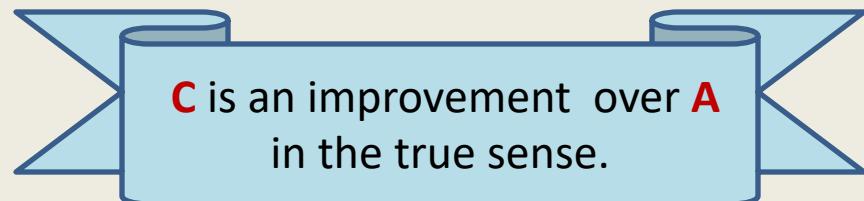
- Algorithm **B** has time complexity $g(n) = n^2 + 10$
- Algorithm **C** has time complexity $h(n) = 10n^{1.5} + 20n + 2000$



Which of **B** and **C** is an improvement over **A** in the true sense ?

$$\lim_{n \rightarrow \infty} \frac{g(n)}{f(n)} = 1/5$$

$$\lim_{n \rightarrow \infty} \frac{h(n)}{f(n)} = 0$$



C is an improvement over **A** in the true sense.

Rule 2

An algorithm **X** is superior to another algorithm **Y** if
the **ratio** of time complexity of **X** and time complexity of **Y**
approaches 0 for asymptotically large input size.

Some Observations

Algorithm **A** has time complexity $f(n) = 5n^2 + n + 1250$

Researchers have designed two new algorithms **B** and **C**

- Algorithm **B** has time complexity $g(n) = n^2 + 10$
- Algorithm **C** has time complexity $h(n) = 10n^{1.5} + 20n + 2000$

Algorithm **C** is the most efficient of all.

Observation 1:

multiplicative or additive **Constants** do not play any role.

Observation 2:

The highest order term governs the time complexity asymptotically.

ORDER NOTATIONS

A **mathematical way**
to capture the **intuitions** developed till now.
(reflect upon it yourself)

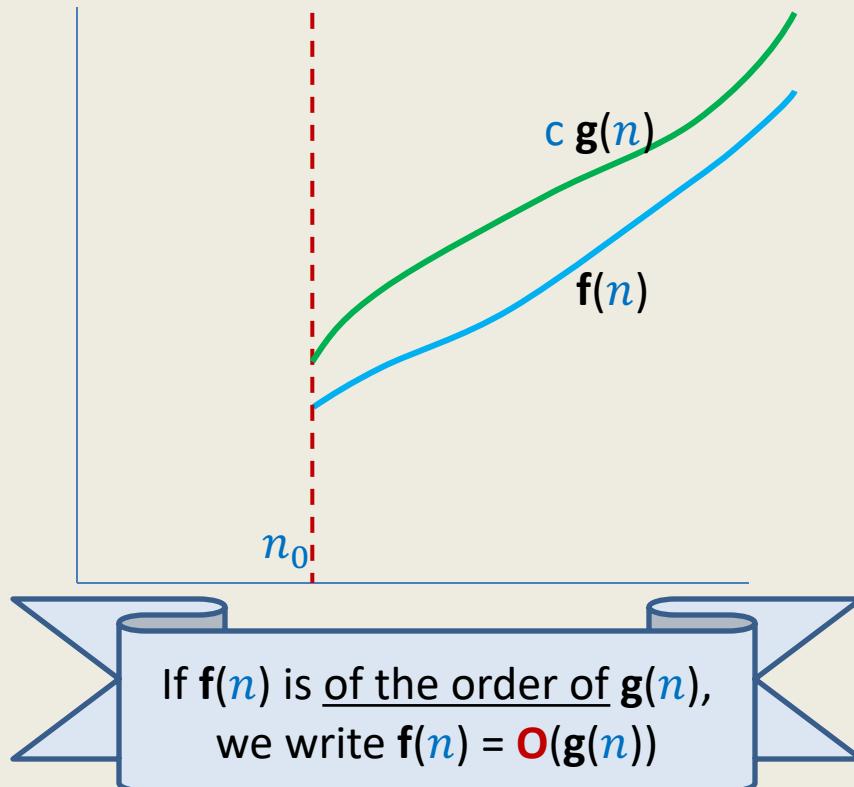
Order notation

Definition: Let $f(n)$ and $g(n)$ be any two increasing functions of n .

$f(n)$ is said to be of the order of $g(n)$

if there exist constants c and n_0 such that

$$f(n) \leq c g(n) \quad \text{for all } n > n_0$$



Order notation : Examples

- $20 n^2 = \mathbf{O}(n^2)$
- ~~$100 n + 60 = \mathbf{O}(n^2)$~~ Loose
- $100 n + 60 = \mathbf{O}(n)$
- ~~$10 n^2 = \mathbf{O}(n^{2.5})$~~ Loose
- $2000 = \mathbf{O}(1)$

$$c = 20, n_0 = 1$$

$$c = 1, n_0 = 160$$

$$c = 160, n_0 = 1$$

While analyzing time complexity of an algorithm accurately, our aim should be to choose the $g(n)$ which is not **loose**. Later in the course, we shall refine & extend this notion suitably.

Simple observations:

- If $f(n) = \mathbf{O}(g(n))$ and $g(n) = \mathbf{O}(h(n))$, then

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

- If $f(n) = \mathbf{O}(h(n))$ and $g(n) = \mathbf{O}(h(n))$, then $f(n) + g(n) = \mathbf{O}(h(n))$

These observations can be helpful for simplifying time complexity.

Prove these observation as **Homeworks**

A neat description of time complexity

- Algorithm **B** has time complexity $g(n) = n^2 + 10$
Hence $g(n) = O(n^2)$
- Algorithm **C** has time complexity $h(n) = 10n^{1.5} + 20n + 2000$
Hence $h(n) = O(n^{1.5})$
- Algorithm for multiplying two $n \times n$ matrices has time complexity
 $n^3 + n^2 + 1 = O(n^3)$

Homeworks:

- $g(n) = 2^n$, $f(n) = 3^n$. Is $f(n) = O(g(n))$? Give proof.
- What is the time complexity of **selection sort** on an array storing n elements?
- What is the time complexity of **Binary search** in a sorted array of n elements?

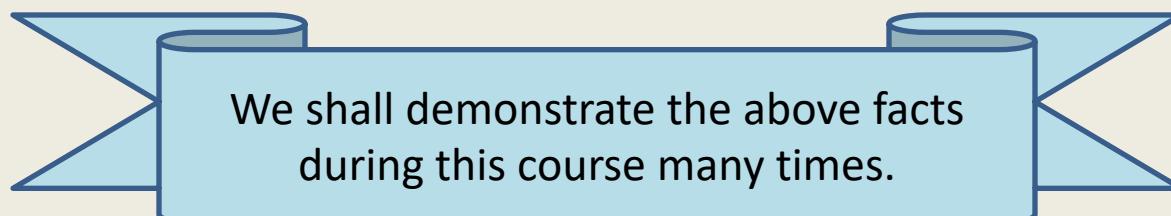
HOW TO DESIGN EFFICIENT ALGORITHM ?

(This sentence captures precisely the goal of theoretical computer science)

Designing an efficient algorithm

Facts from the world of algorithms:

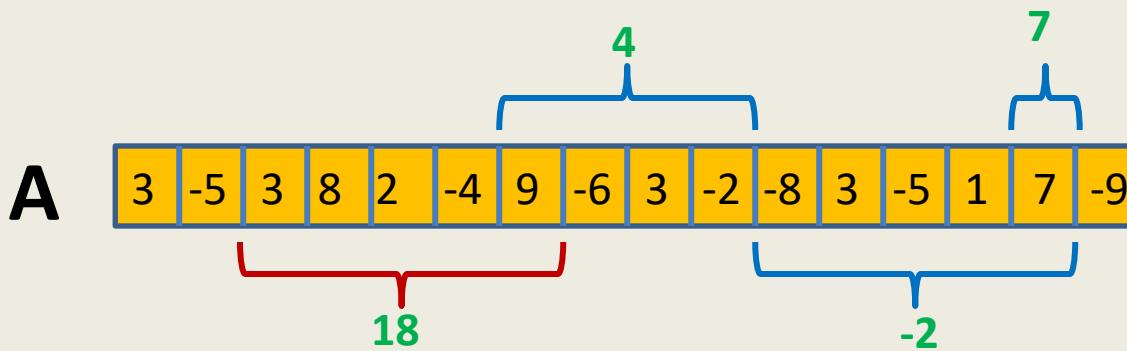
1. **No formula** for designing efficient algorithms.
2. Every new problem demands a **fresh** approach.
3. Designing an efficient algorithm or data structure requires
 1. Ability to make **key observations**.
 2. Ability to ask **right kind of questions**.
 3. A **positive attitude** and ...
 4. a lot of **perseverance**.



We shall demonstrate the above facts
during this course many times.

Max-sum subarray problem

Given an array **A** storing n numbers,
find its **subarray** the sum of whose elements is maximum.



Max-sum subarray problem: A trivial algorithm

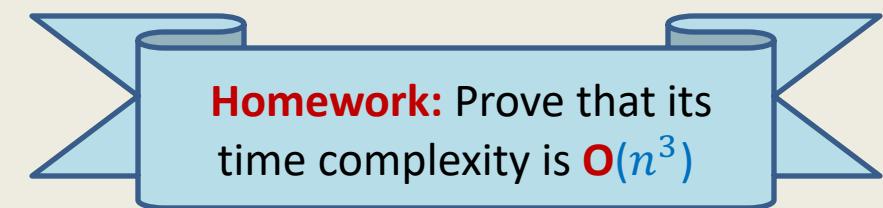
```
A_trivial_algo(A)
```

```
{ max ← A[0];  
  For i=0 to n-1  
    For j=i to n-1  
      {   temp ← compute_sum(A,i,j);  
          if max < temp then max ← temp;  
      }  
  return max;
```

```
}
```



```
compute_sum(A, i,j)  
{ sum ← A[i];  
  For k=i+1 to j    sum ← sum+A[k];  
  return sum;  
}
```



Max-sum subarray problem:

Question: Can we design $O(n)$ time algorithm for Max-sum subarray problem ?

Answer: Yes.

Think over it with a fresh mind

We shall design it together in the next class...😊

Data Structures and Algorithms

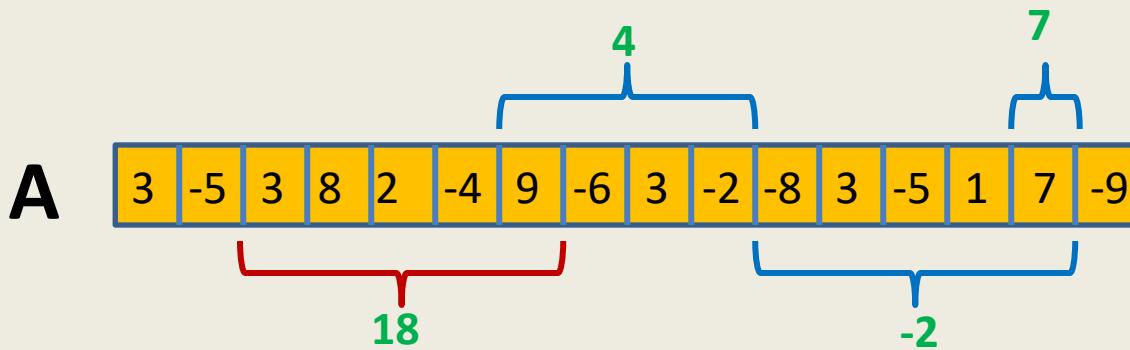
(ESO207)

Lecture 4:

- Design of $O(n)$ time algorithm for Maximum sum subarray
- Proof of correctness of an algorithm
- A new problem : Local Minima in a grid

Max-sum subarray problem

Given an array **A** storing n numbers,
find its **subarray** the sum of whose elements is maximum.



Max-sum subarray problem: A trivial algorithm

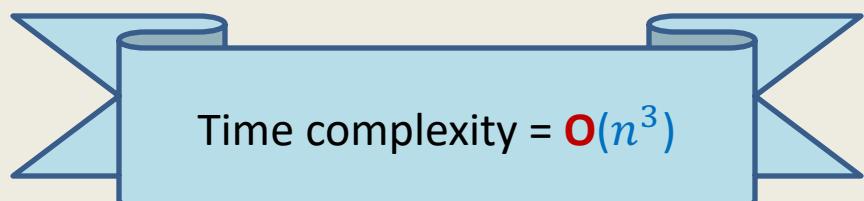
```
A_trivial_algo(A)
```

```
{ max ← A[0];  
  For i=0 to n-1  
    For j=i to n-1  
      {   temp ← compute_sum(A,i,j);  
          if max < temp then max ← temp;  
      }  
  return max;
```

```
}
```



```
compute_sum(A, i,j)  
{ sum ← A[i];  
  For k=i+1 to j    sum ← sum+A[k];  
  return sum;  
}
```

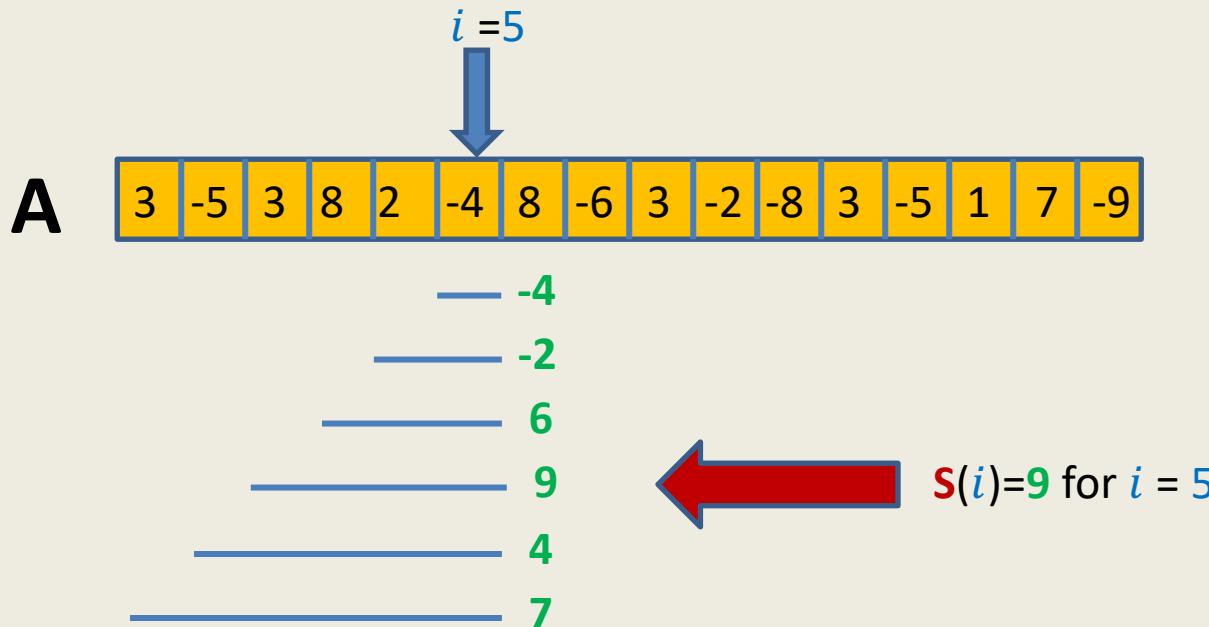


Time complexity = $O(n^3)$

DESIGNING AN $O(n)$ TIME ALGORITHM

Focusing on any particular index i

Let $S(i)$: the sum of the maximum-sum subarray ending at index i .



Observation:

In order to solve the problem, it suffices to compute $S(i)$ for each $0 \leq i < n$.

Focusing on any particular index i

Observation:

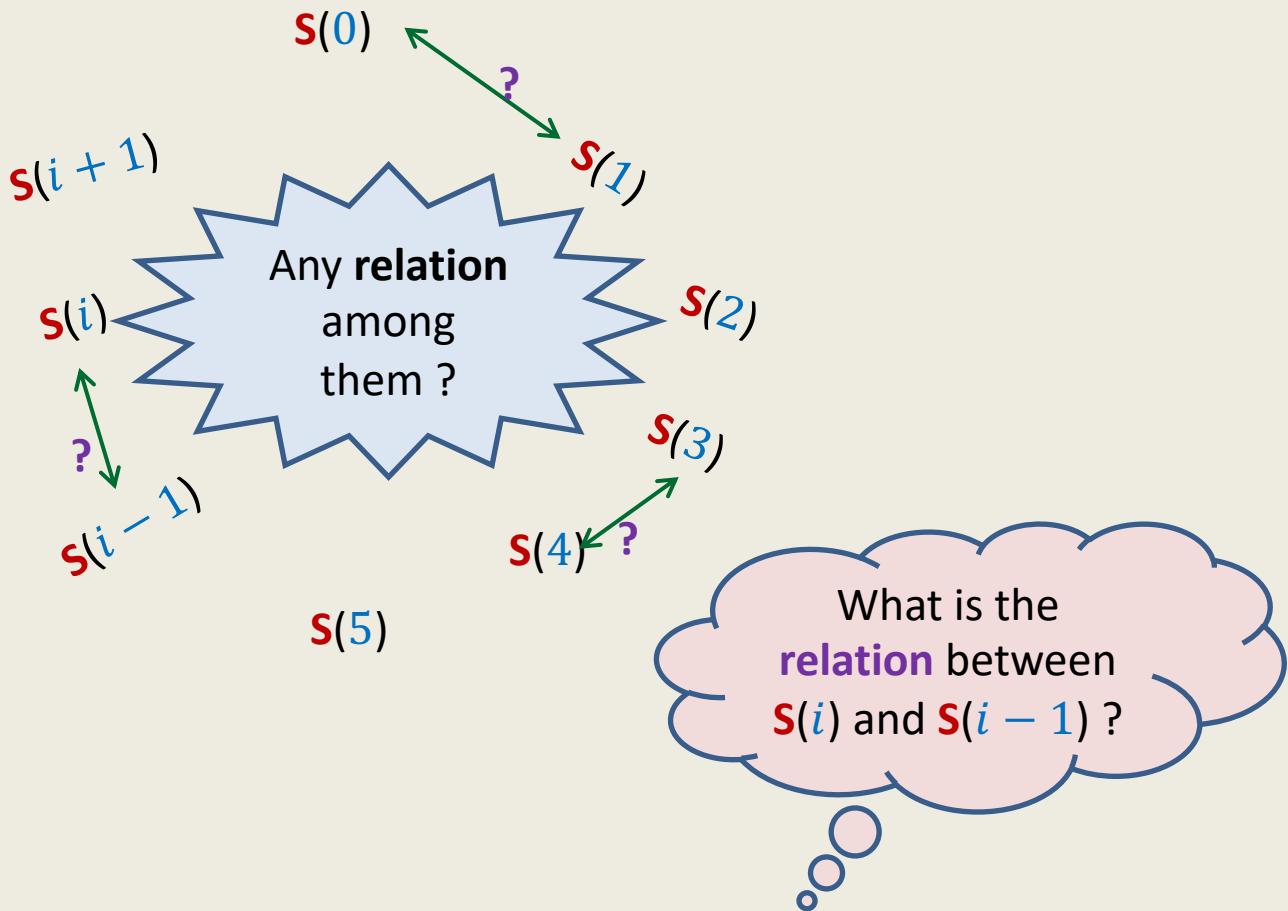
In order to solve the problem, it suffices to compute $S(i)$ for each $0 \leq i < n$.



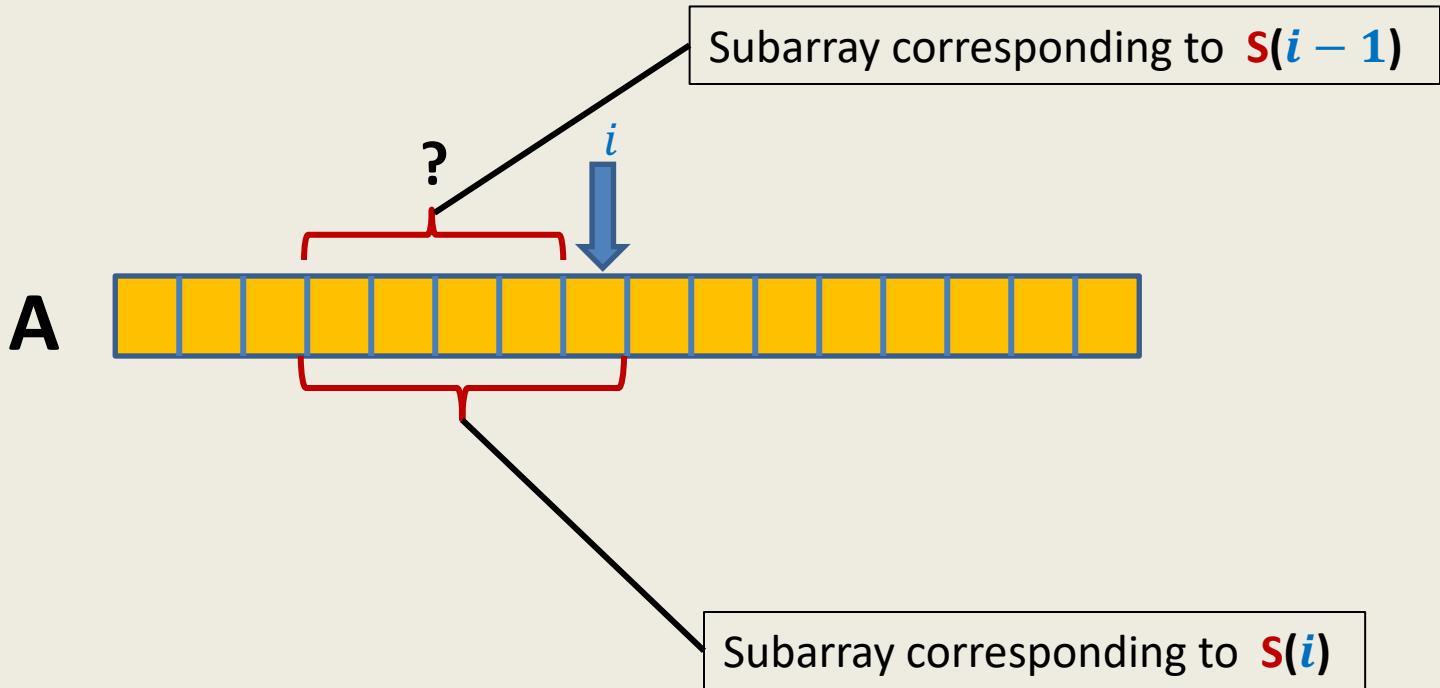
Question: If we wish to achieve $O(n)$ time to solve the problem, how quickly should we be able to compute $S(i)$ for a given index i ?

Answer: $O(1)$ time.

How to compute $S(i)$ in $O(1)$ time ?



Relation between $S(i)$ and $S(i - 1)$



Theorem 1:

If $S(i - 1) > 0$ then $S(i) = S(i - 1) + A[i]$
else $S(i) = A[i]$

An $O(n)$ time Algorithm for Max-sum subarray

Max-sum-subarray-algo($A[0 \dots n - 1]$)

```
{    $S[0] \leftarrow A[0];$             $\overbrace{\hspace{10em}}$   $O(1)$  time  
    for  $i = 1$  to  $n - 1$        $\overbrace{\hspace{10em}}$   $n - 1$  repetitions  
    {      If  $S[i - 1] > 0$  then  $S[i] \leftarrow S[i - 1] + A[i]$  }            $\overbrace{\hspace{10em}}$   $O(1)$  time  
      else  $S[i] \leftarrow A[i]$  }  
  }  
  “Scan  $S$  to return the maximum entry”  $\overbrace{\hspace{10em}}$   $O(n)$  time  
}
```

Time complexity of the algorithm = $O(n)$

Homework:

- Refine the algorithm so that it uses only $O(1)$ extra space.

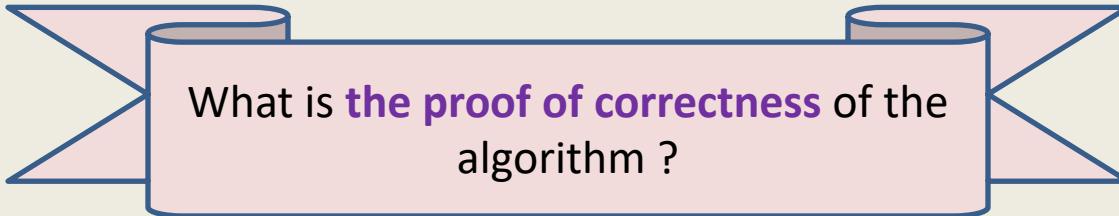
An $O(n)$ time Algorithm for Max-sum subarray

Max-sum-subarray-algo($A[0 \dots n - 1]$)

```
{    $S[0] \leftarrow A[0]$ 
    for  $i = 1$  to  $n - 1$ 
    {      If  $S[i - 1] > 0$  then  $S[i] \leftarrow S[i - 1] + A[i]$ 
          else  $S[i] \leftarrow A[i]$ 
    }
}
```

“Scan S to return the maximum entry”

```
}
```



What is the proof of correctness of the algorithm ?

What does correctness of an algorithm mean ?

For every possible **valid input**, the algorithm must output **correct** answer.

An $O(n)$ time Algorithm for Max-sum subarray

Max-sum-subarray-algo($A[0 \dots n - 1]$)

```
{    $S[0] \leftarrow A[0]$ 
    for  $i = 1$  to  $n - 1$ 
    {      If  $S[i - 1] > 0$  then  $S[i] \leftarrow S[i - 1] + A[i]$ 
          else  $S[i] \leftarrow A[i]$ 
    }
}
```

“Scan S to return the maximum entry”

```
}
```

Question:

What needs to be proved in order to establish the correctness of this algorithm ?

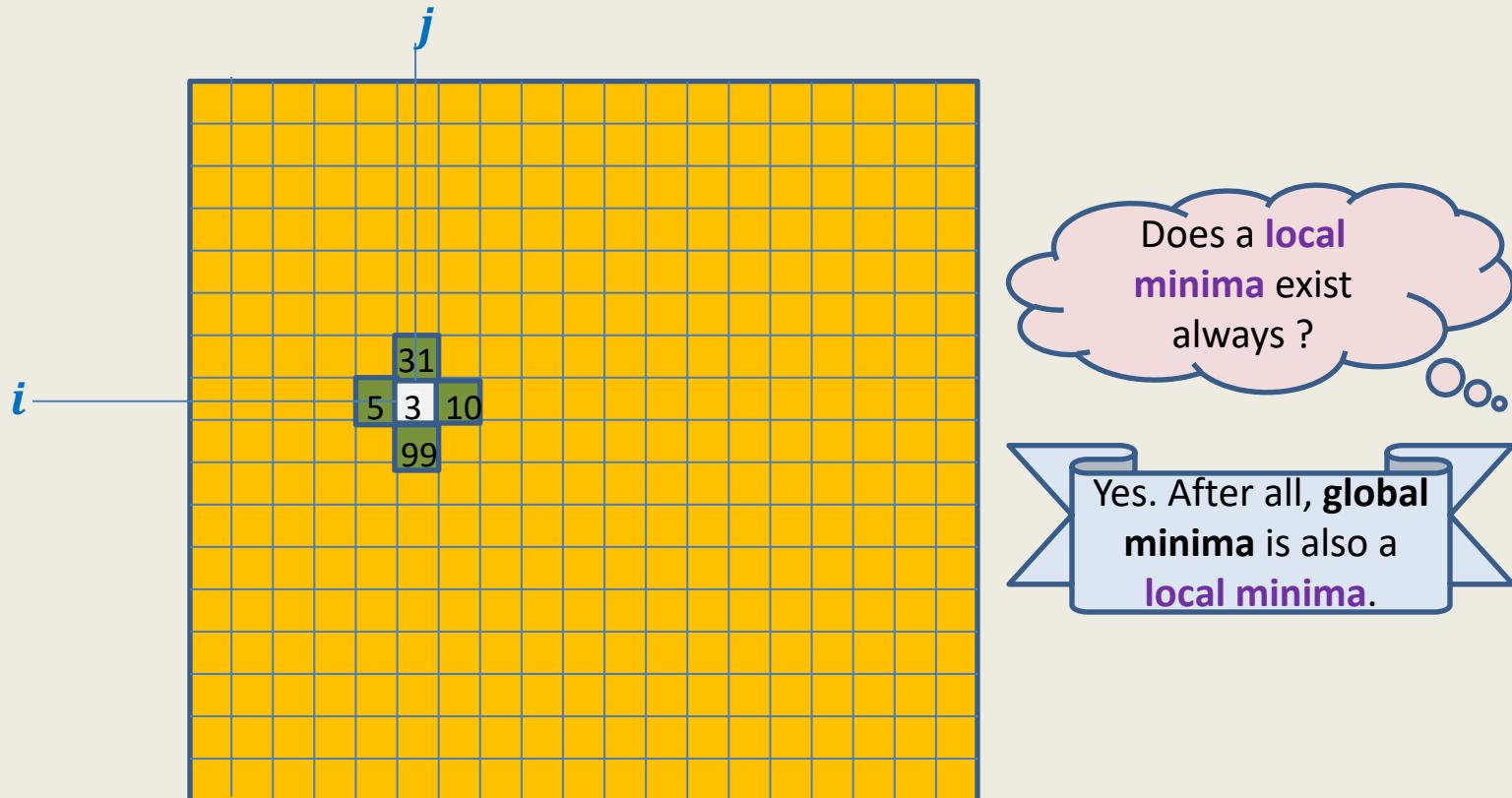
Ponder over this question before coming to the next class...

NEW PROBLEM:

LOCAL MINIMA IN A GRID

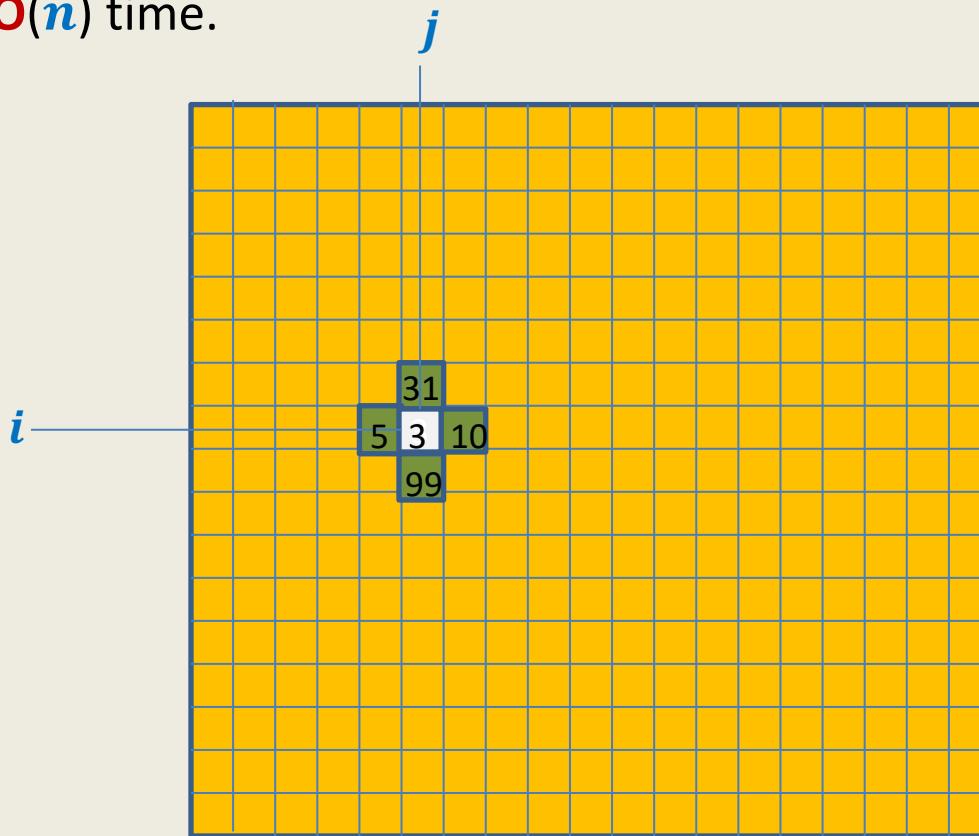
Local minima in a grid

Definition: Given a $n \times n$ grid storing distinct numbers, an entry is local minima if it is smaller than each of its neighbors.



Local minima in a grid

Problem: Given a $n \times n$ grid storing distinct numbers, output any local minima in $O(n)$ time.



Using common sense principles

- There are some simple but very fundamental principles which are not restricted/confined to a specific stream of science/philosophy.
- These principles, which we usually learn as common sense, can be used in so many diverse areas of human life.
- For the current problem of local minima, we shall use two such simple principles.

This should convince you that designing algorithm does not require any thing **magical** 😊!

Two simple principles

1. Respect every new idea even if it does not solve a problem finally.

2. Principle of simplification:

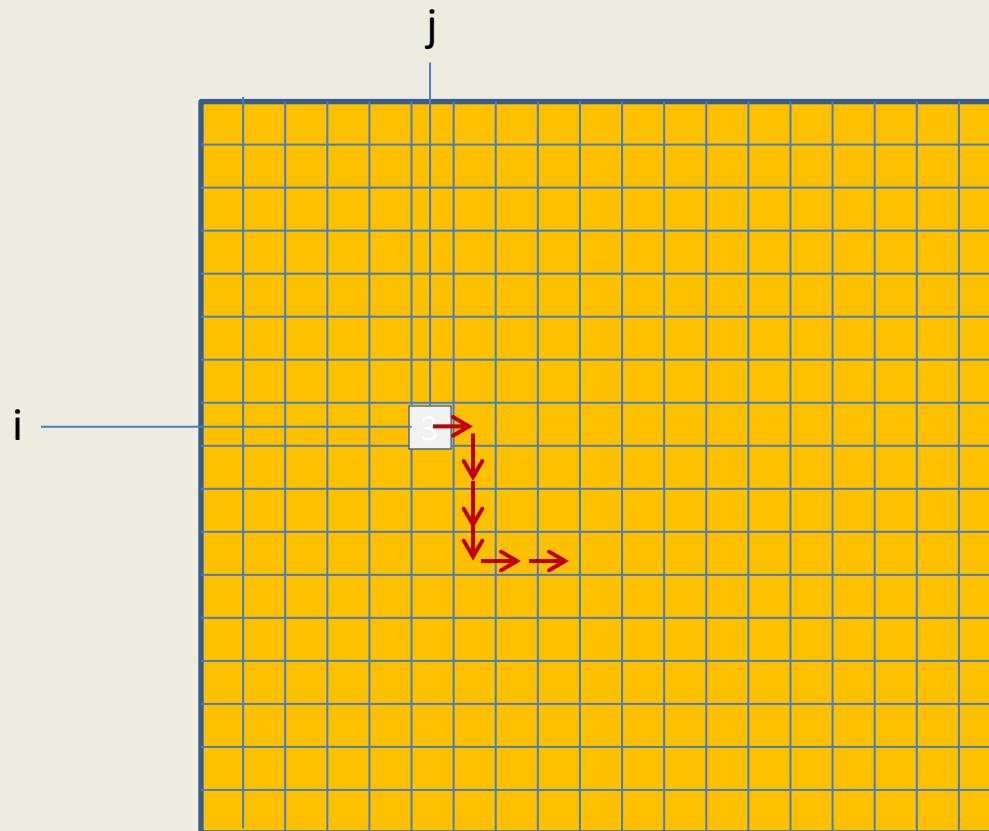
If you find a problem difficult,

→ Try to solve its simpler version, and then ...

→ Try to extend this solution to the original (difficult) version.

A new approach

Repeat : if current entry is not local minima, explore the neighbor storing smaller value.



A new approach

Explore()

```
{   Let c be any entry to start with;  
    While(c is not a local minima)  
    {  
        c ← a neighbor of c storing smaller value  
    }  
    return c;  
}
```

Question: What is the proof of correctness of **Explore** ?

Answer:

- It suffices if we can prove that **While** loop eventually terminates.
- Indeed, the loop terminates since **we never visit a cell twice**.

A new approach

Explore()

```
{   Let c be any entry to start with;  
    While(c is not a local minima)  
    {  
        c ← a neighbor of c storing smaller value  
    }  
    return c;  
}
```

Worst case time complexity : $O(n^2)$



How to apply this principle ?

First principle:
Do not discard **Explore()**

Second principle:
Simplify the problem

Local minima in an array

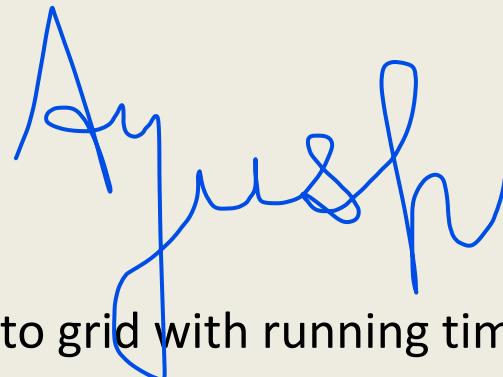
A



Theorem 2: A local minima in an array storing n distinct elements can be found in $O(\log n)$ time.

Homework:

- Design the algorithm stated in **Theorem 2**.
- Spend some time to extend this algorithm to grid with running time= $O(n)$.



Please come prepared in the next class 😊

Data Structures and Algorithms

(ESO207)

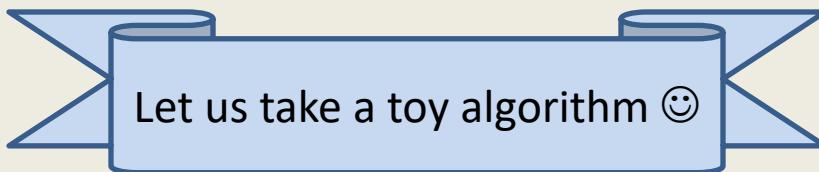
Lecture 5:

- More on **Proof of correctness** of an algorithm
- Design of $O(n)$ time algorithm for **Local Minima in a grid**

PROOF OF CORRECTNESS

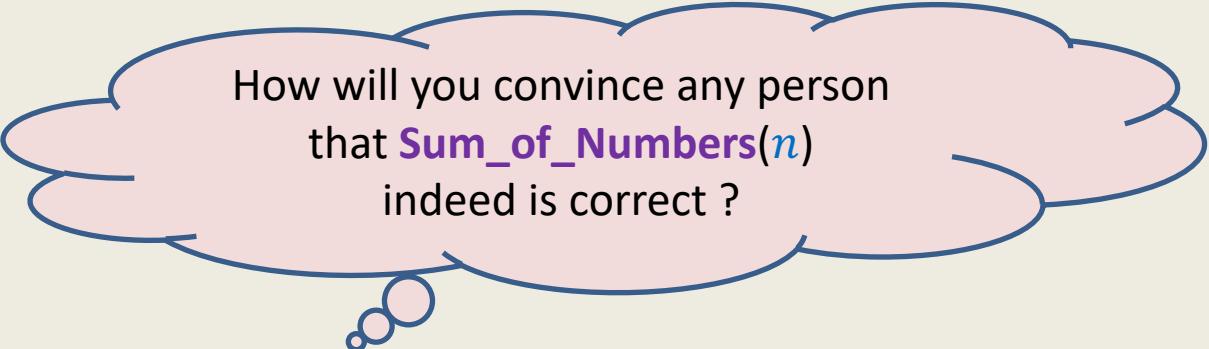
What does correctness of an algorithm mean ?

For every possible **valid input**, the algorithm must output **correct** answer.



Algorithm for computing sum of numbers from 0 to n

```
Sum_of_Numbers( $n$ )
{
    Sum ← 0;
    for  $i = 1$  to  $n$ 
    {
        Sum ← Sum +  $i$ ;
    }
    return Sum;
}
```



How will you convince any person
that **Sum_of_Numbers(n)**
indeed is correct ?

Natural responses:

- It is obvious !
- Compile it and run it for some random values of n .
- Go over first few iterations explaining what happens to **Sum**.

How will you respond

if you have to do it for the following code ?

```
void dij(int n,int v,int cost[10][10],int dist[])
{
    int i,u,count,w,flag[10],min;
    for(i=1;i<=n;i++)
        flag[i]=0,dist[i]=cost[v][i];
    count=2;
    while(count<=n)
    {
        min=99;
        for(w=1;w<=n;w++)
            if(dist[w]<min && !flag[w])
                min=dist[w],u=w;
        flag[u]=1;
        count++;
        for(w=1;w<=n;w++)
            if((dist[u]+cost[u][w]<dist[w]) && !flag[w])
                dist[w]=dist[u]+cost[u][w];
    }
}
```

Think for some time to realize

- the **non-triviality**
- the **Importance** of proof of correctness of an iterative algorithm.

In the following slide, we present an overview of the proof of correctness.

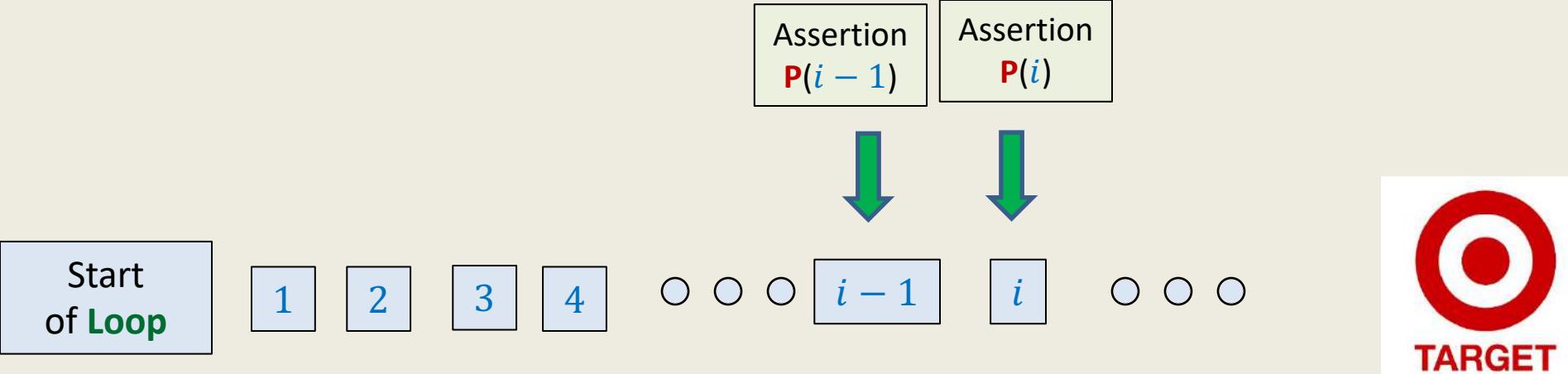
Interestingly, such a proof will be just

Expressing our intuition/insight of the algorithm in a **formal** way ☺.

Proof of correctness

For an **iterative algorithm**

Insight of the algorithm → **Theorem**



Start
of Loop

1

2

3

4

○ ○ ○

$i - 1$

i

○ ○ ○

Prove $P(i)$ by

1. Assuming $P(i - 1)$
2. **Theorem**
3. Body of the **Loop**

Proof by induction

What would you expect
at the end of i th
iteration ?

The most difficult/creative part of proof : To come up with the right assertion $P(i)$

Algorithm for computing sum of numbers from 0 to n

```
{   Sum<-0;  
    for  $i$  = 1 to  $n$   
    {  
        Sum<- Sum +  $i$ ;  
    }  
    return Sum;  
}
```

Assertion $P(i)$: At the end of i th iteration **Sum** stores the sum of numbers from 0 to i .

Base case: $P(0)$ holds.

Assuming $P(i - 1)$, assertion $P(i)$ also holds.

$P(n)$ holds.

An $O(n)$ time Algorithm for Max-sum subarray

Let $S(i)$: the sum of the maximum-sum subarray ending at index i .

Theorem 1 : If $S(i - 1) > 0$ then $S(i) = S(i - 1) + A[i]$
else $S(i) = A[i]$

Max-sum-subarray-algo($A[0 \dots n - 1]$)

```
{    $S[0] \leftarrow A[0]$ 
    for  $i = 1$  to  $n - 1$ 
    {      If  $S[i - 1] > 0$  then  $S[i] \leftarrow S[i - 1] + A[i]$ 
          else  $S[i] \leftarrow A[i]$ 
    }
```

“Scan S to return the maximum entry”

```
}
```

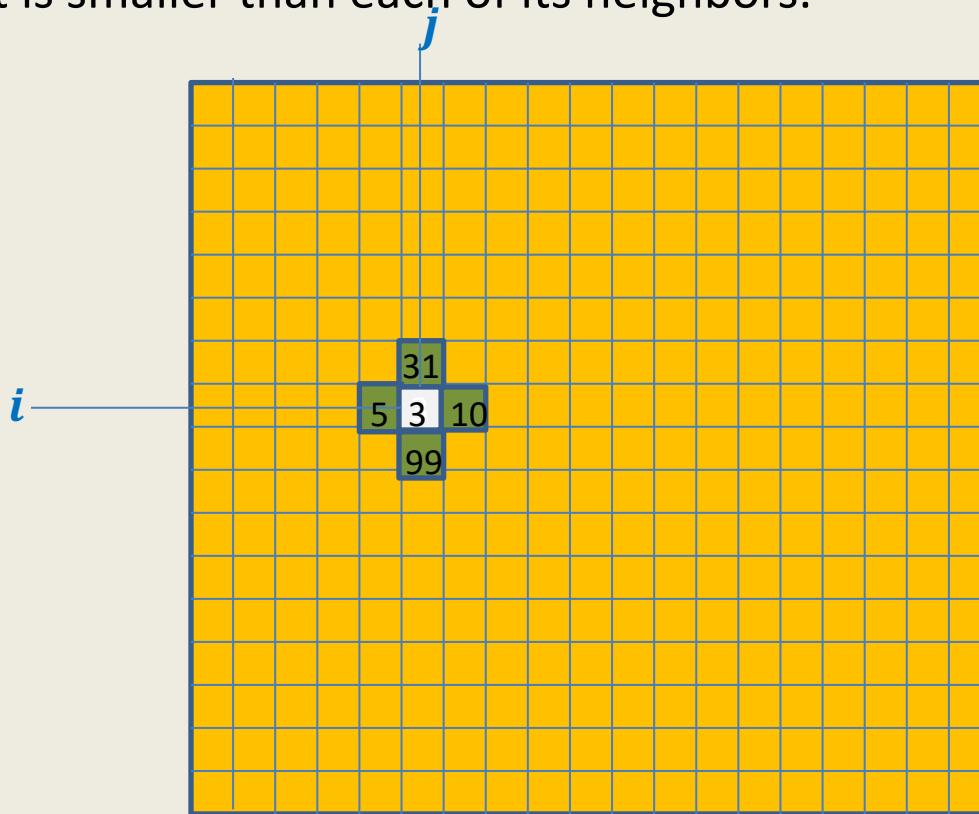
Assertion $P(i)$: $S[i]$ stores the sum of maximum sum subarray ending at $A[i]$.

Homework: Prove that $P(i)$ holds for all $i \leq n - 1$

LOCAL MINIMA IN A GRID

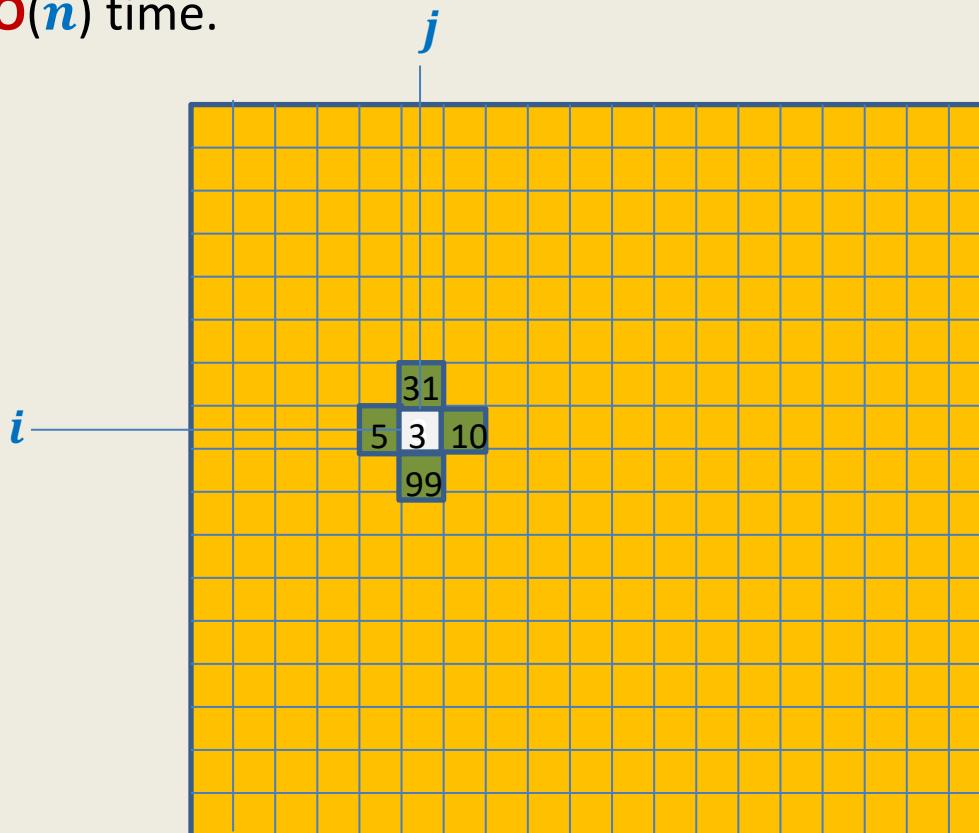
Local minima in a grid

Definition: Given a $n \times n$ grid storing distinct numbers, an entry is local minima if it is smaller than each of its neighbors.



Local minima in a grid

Problem: Given a $n \times n$ grid storing distinct numbers, output any local minima in $O(n)$ time.



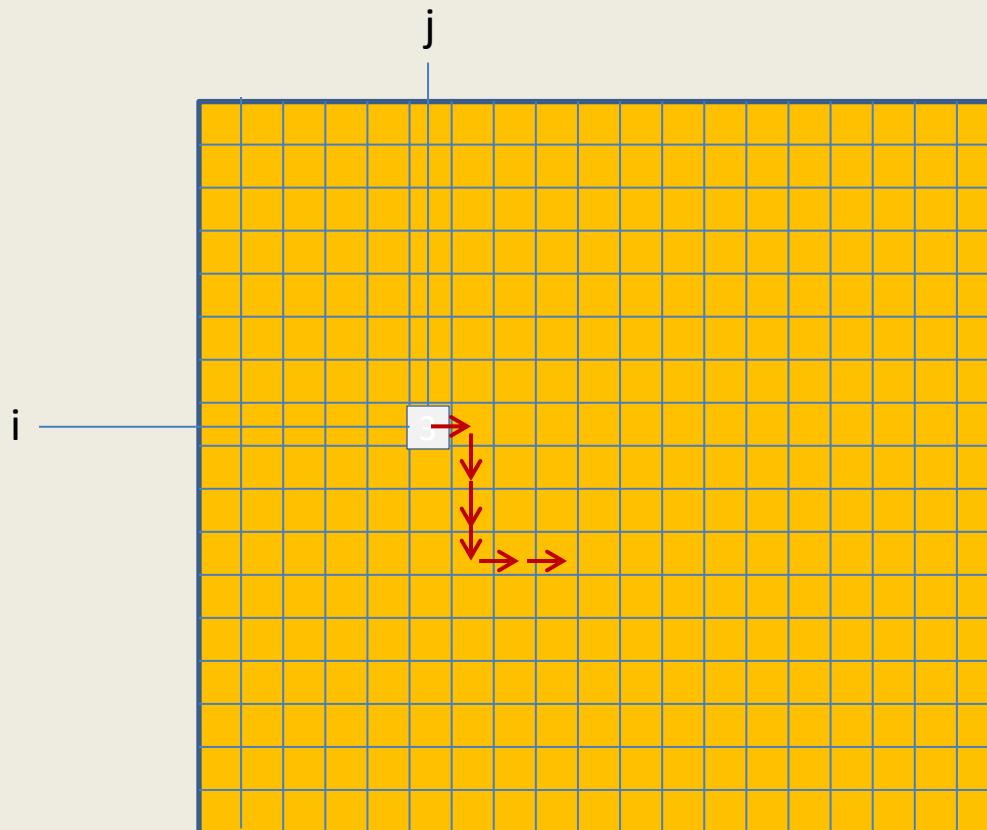
Two simple principles

1. Respect every new idea which solves a problem even partially.

2. Principle of simplification:
If you find a problem difficult,
→ try to solve its simpler version, and then
→ extend this solution to the original (difficult) version.

A new approach

Repeat : if current entry is not local minima, explore the neighbor storing smaller value.



A new approach

Explore()

```
{  Let c be any entry to start with;  
  While(c is not a local minima)  
  {  
    c ← a neighbor of c storing smaller value  
  }  
  return c;  
}
```

A new approach

Explore()

```
{   Let c be any entry to start with;  
    While(c is not a local minima)  
    {  
        c ← a neighbor of c storing smaller value  
    }  
    return c;  
}
```

Worst case time complexity : $O(n^2)$

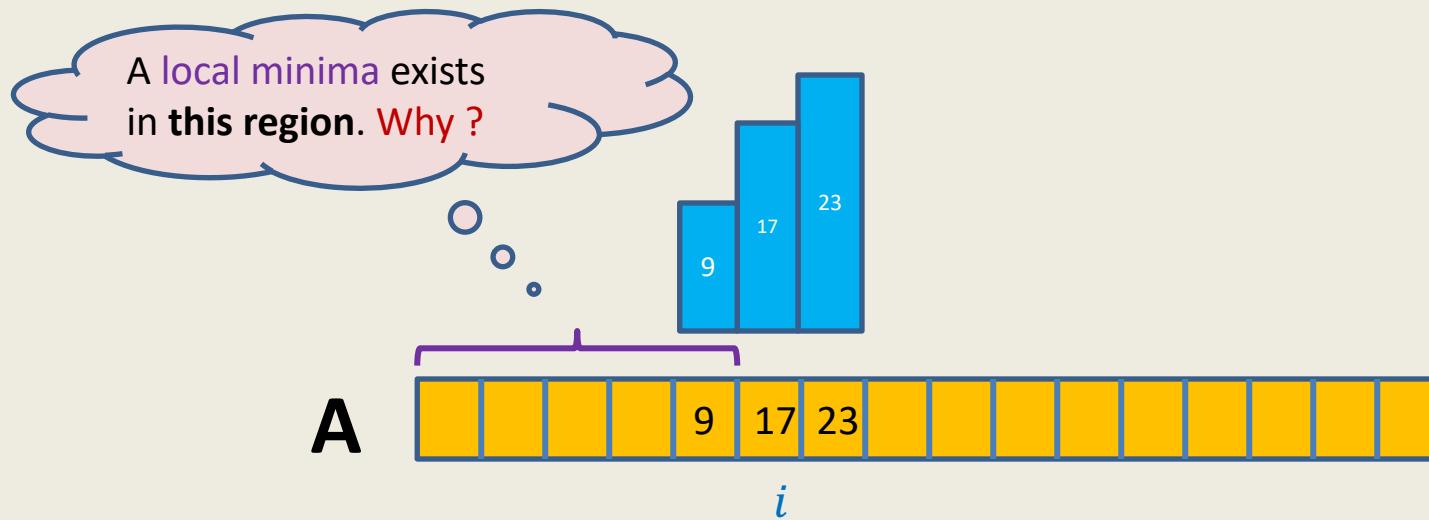
How to apply this principle ?



First principle:
Do not discard **Explore()**

Second principle:
Simplify the problem

Local minima in an array



Theorem: There is a local minima in $A[0, \dots, i - 1]$.

Proof: Suppose we execute **Explore()** from $A[i - 1]$.

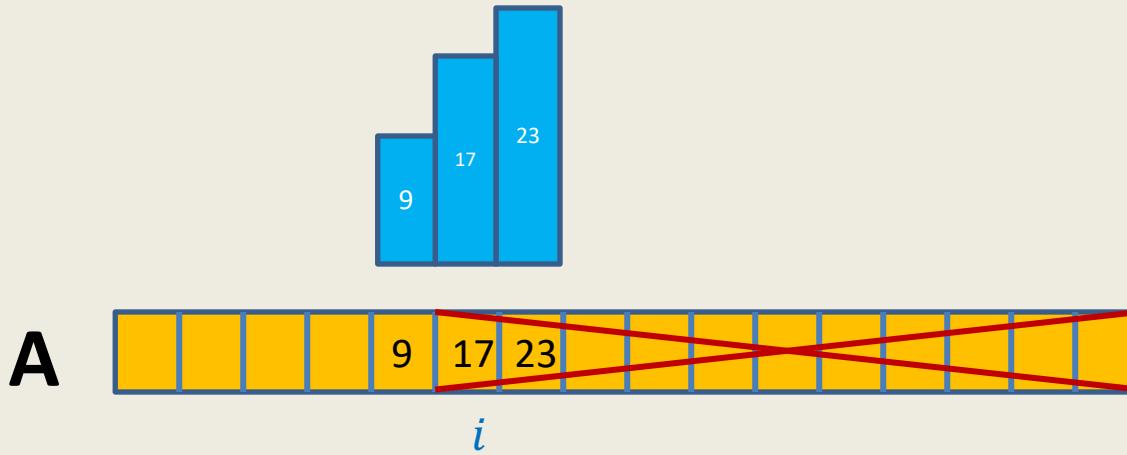
Explore(), if terminates, will return local minima.

It will terminate without ever entering $A[i, \dots, n - 1]$.

Hence there is a local minima in $A[0, \dots, i - 1]$.

Algorithmic proof

Local minima in an array



Theorem: There is a local minima in $A[0, \dots, i - 1]$.

- We can confine our search for local minima to only $A[0, \dots, i - 1]$.
- Our problem size has reduced.



Question: Which *i* should we select so as to reduce problem size significantly ?

Answer: *middle* point of array A.

Local minima in an array

(Similar to binary search)

```
int Local-minima-in-array(A) {  
    L ← 0;  
    R ← n - 1;  
    found ← FALSE;  
    while( not found ) {  
        mid ← (L + R)/2;  
        If (mid is a local minima)  
            found ← TRUE;  
        else if(A[mid + 1] < A[mid])  
            L ← mid + 1  
        else R ← mid - 1  
    }  
    return mid; }
```

How many iterations ?

$O(\log n)$

$O(1)$ time
in one iteration

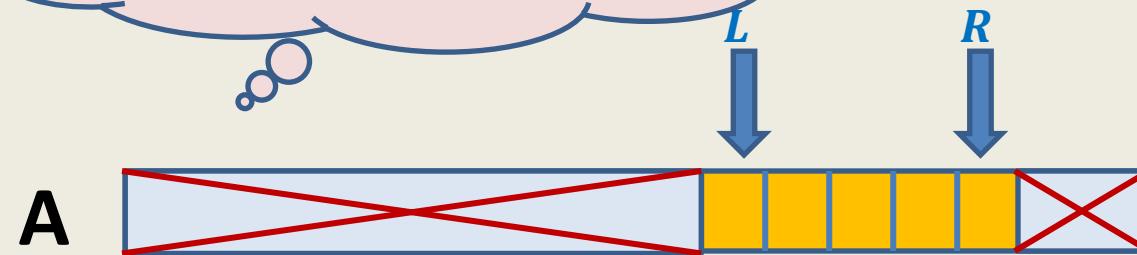
Proof of correctness ?

→ Running time of the algorithm = $O(\log n)$

Local minima in an array

(Proof of correctness)

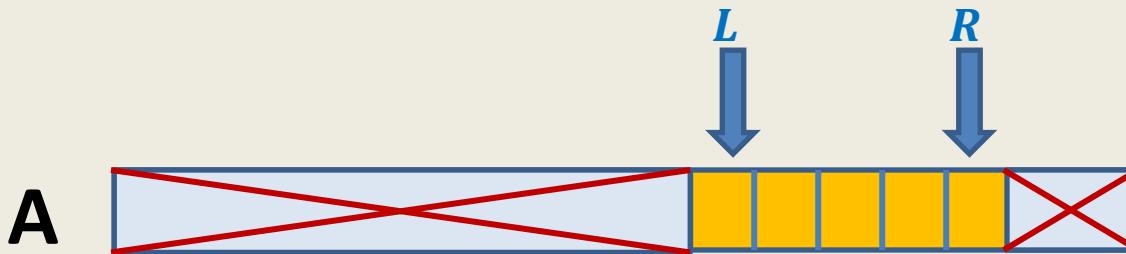
What can you say about the algorithm at the end of i th iteration.



P(i) : At the end of i th iteration,
“A local minima of array **A** exists in $\mathbf{A}[\mathbf{L}, \dots, \mathbf{R}]$.”

Local minima in an array

(Proof of correctness)



$\mathbf{P}(i)$: At the end of i th iteration,
“A local minima of array \mathbf{A} exists in $\mathbf{A}[L, \dots, R]$.”

=

“ $\mathbf{A}[L] < \mathbf{A}[L - 1]$ ” and “ $\mathbf{A}[R] < \mathbf{A}[R + 1]$ ”.

Homework:

- Make sincere attempts to prove the assertion $\mathbf{P}(i)$.
- How will you use it to prove that **Local-minima-in-array(A)** outputs a local minima ?

Local minima in an array

Theorem: A local minima in an array storing n distinct elements can be found in $O(\log n)$ time.

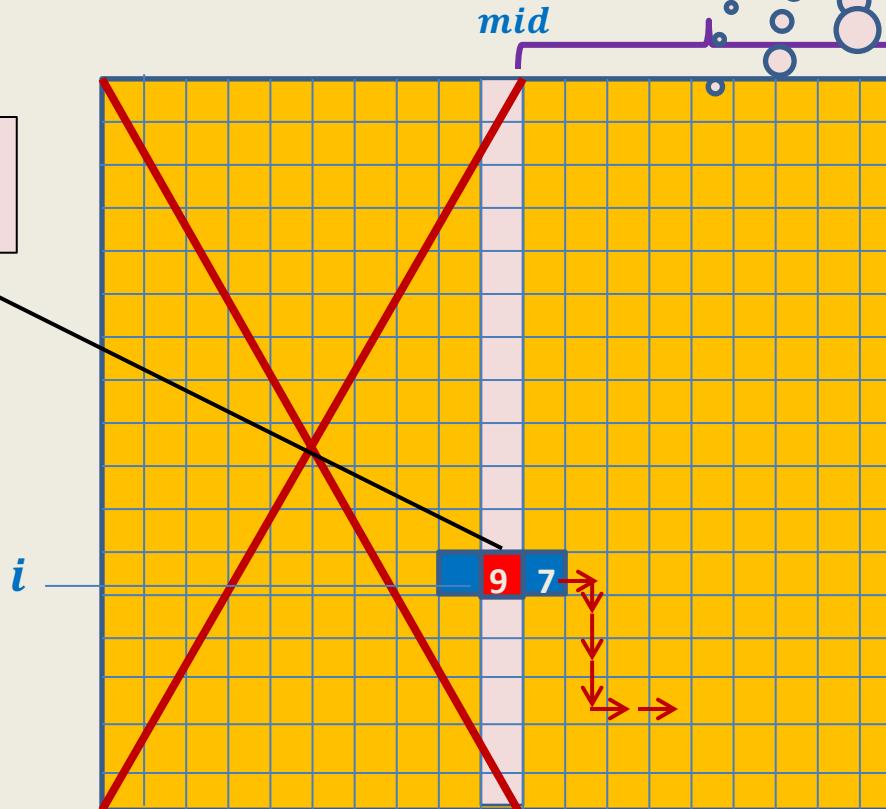
Local minima in a grid

(extending the solution from 1-D to 2-D)

Search for a local minima in the column $M[* , mid]$

Under what circumstances even this smallest element is not a local minima ?

Smallest element of the column



Execute **Explore()** from $M[i, mid + 1]$

Homework:

Use this idea to design an $O(n \log n)$ time algorithm for this problem.

... and do not forget to prove its correctness ☺.

Make sincere attempts to
answer all questions raised in this lecture.

Data Structures and Algorithms

(ESO207)

Lecture 6:

- Design of $O(n)$ time algorithm for Local Minima in a grid
- Data structure gem: Range minima Problem

Local minima in an array

Theorem: A local minima in an array storing n distinct elements can be found in $O(\log n)$ time.

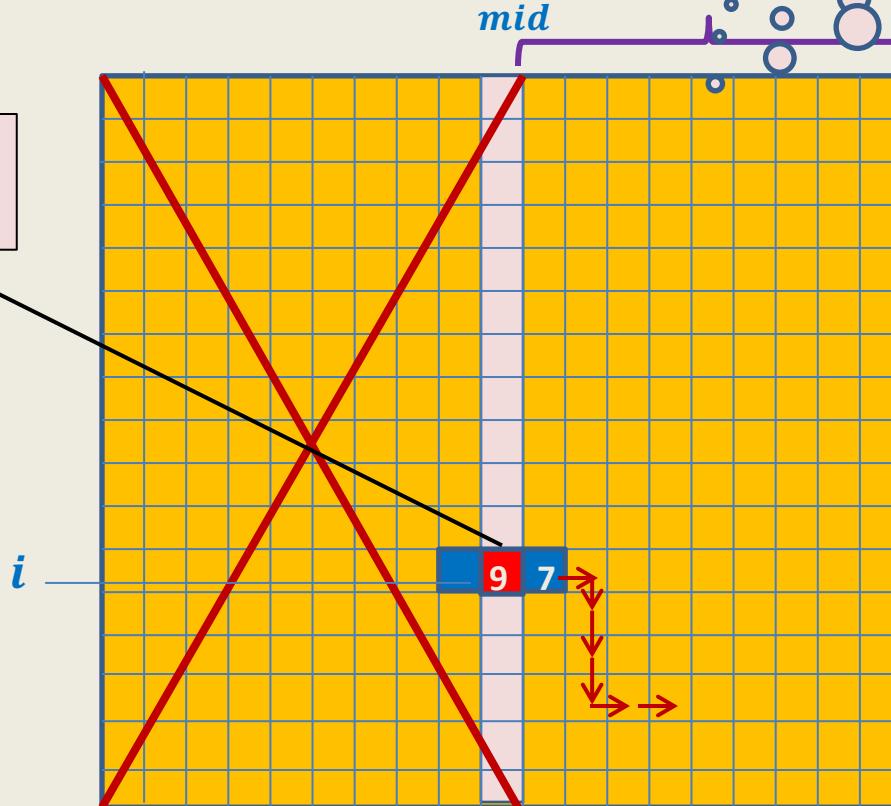
Local minima in a grid

(extending the solution from 1-D to 2-D)

Search for a local minima in the column $M[*, mid]$

Under what circumstances even this smallest element is not a local minima ?

Smallest element of the column



Execute **Explore()** from $M[i, mid + 1]$

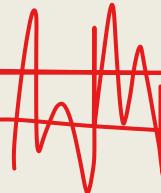
Homework:

Use this idea to design an $O(n \log n)$ time algorithm for this problem.

... and do not forget to prove its correctness ☺.



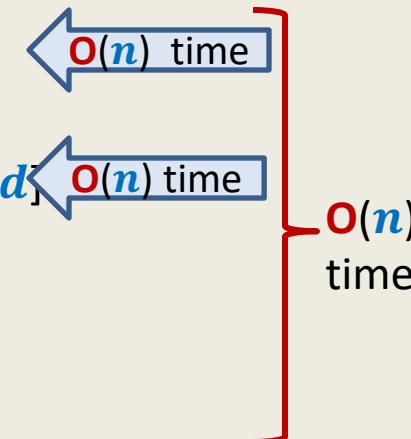
Local minima in a grid



Int Local-minima-in-grid(M) // returns the column containing a local minima

```
{      L  $\leftarrow 0$ ;  
      R  $\leftarrow n - 1$ ;  
      found  $\leftarrow \text{FALSE}$ ;  
      while(not found)  
      {      mid  $\leftarrow (L + R)/2$ ;  
          If ( $M[*]$ , mid) has a local minima) found  $\leftarrow \text{TRUE}$ ;  
          else {  
              let  $M[k, mid]$  be the smallest element in  $M[*]$ , mid  
              if( $M[k, mid + 1] < M[k, mid]$ ) L  $\leftarrow mid + 1$  ;  
              else R  $\leftarrow mid - 1$   
          }  
      }  
      return mid;  
}
```

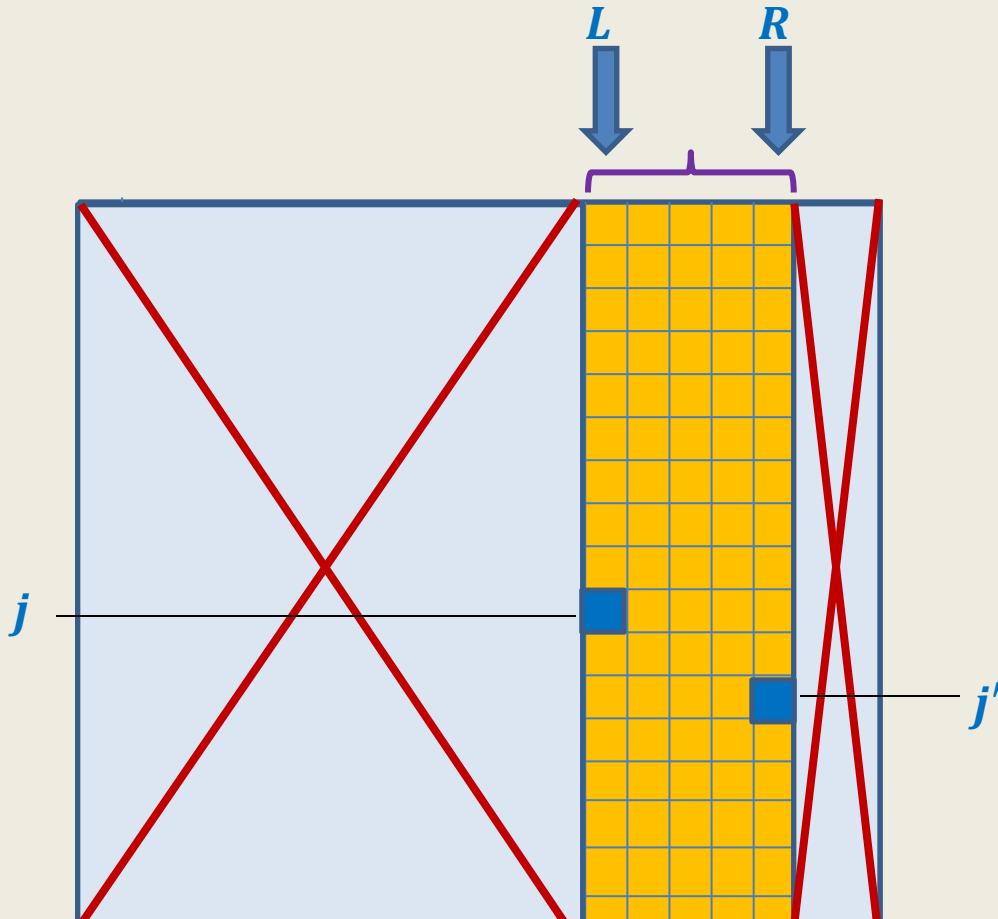
→ Running time of the algorithm = $O(n \log n)$



Proof of correctness ?

Local minima in a grid

(Proof of Correctness)



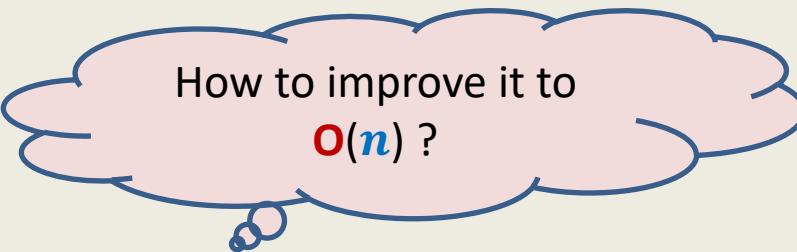
$\mathbf{P}(i) :$

"A local minima of grid \mathbf{M} exists in $\mathbf{M}[L, \dots, R]$."

$\exists j$ such that $\mathbf{M}[j, L] < \mathbf{M}[\ast, L - 1]$ " and $\exists j'$ such that $\mathbf{M}[j', R] < \mathbf{M}[\ast, R + 1]$ "

Local minima in a grid

Theorem: A local minima in an $n \times n$ grid storing distinct elements can be found in $O(n \log n)$ time.



How to improve it to
 $O(n)$?

Local minima in a grid in $O(n)$ time

Let us carefully look at the calculations of the running time of the current algo.

$$cn + cn + cn + \dots \quad (\log n \text{ terms}) \quad \dots + cn = O(n \log n)$$

What about the following series

$$c\frac{n}{2} + c\frac{n}{4} + c\frac{n}{8} + \dots \quad (\log n \text{ terms}) \quad \dots + cn = ?$$

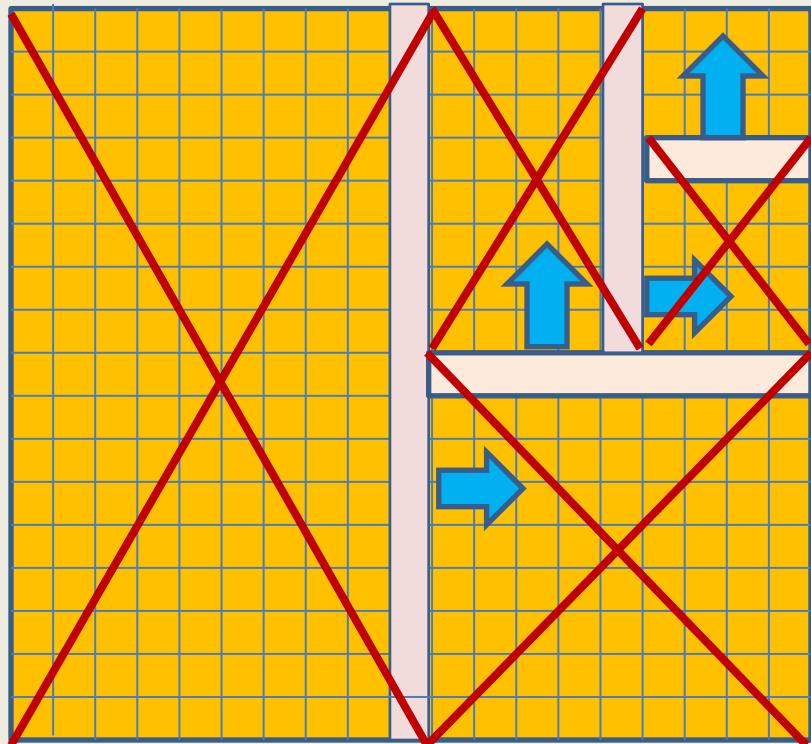


It is $2cn = O(n)$.

Get an !DEA from this series to modify our current algorithm

Local minima in a grid in $O(n)$ time

Bisect alternatively along rows and column



INCORRECT

Exercise: Think of a counterexample!

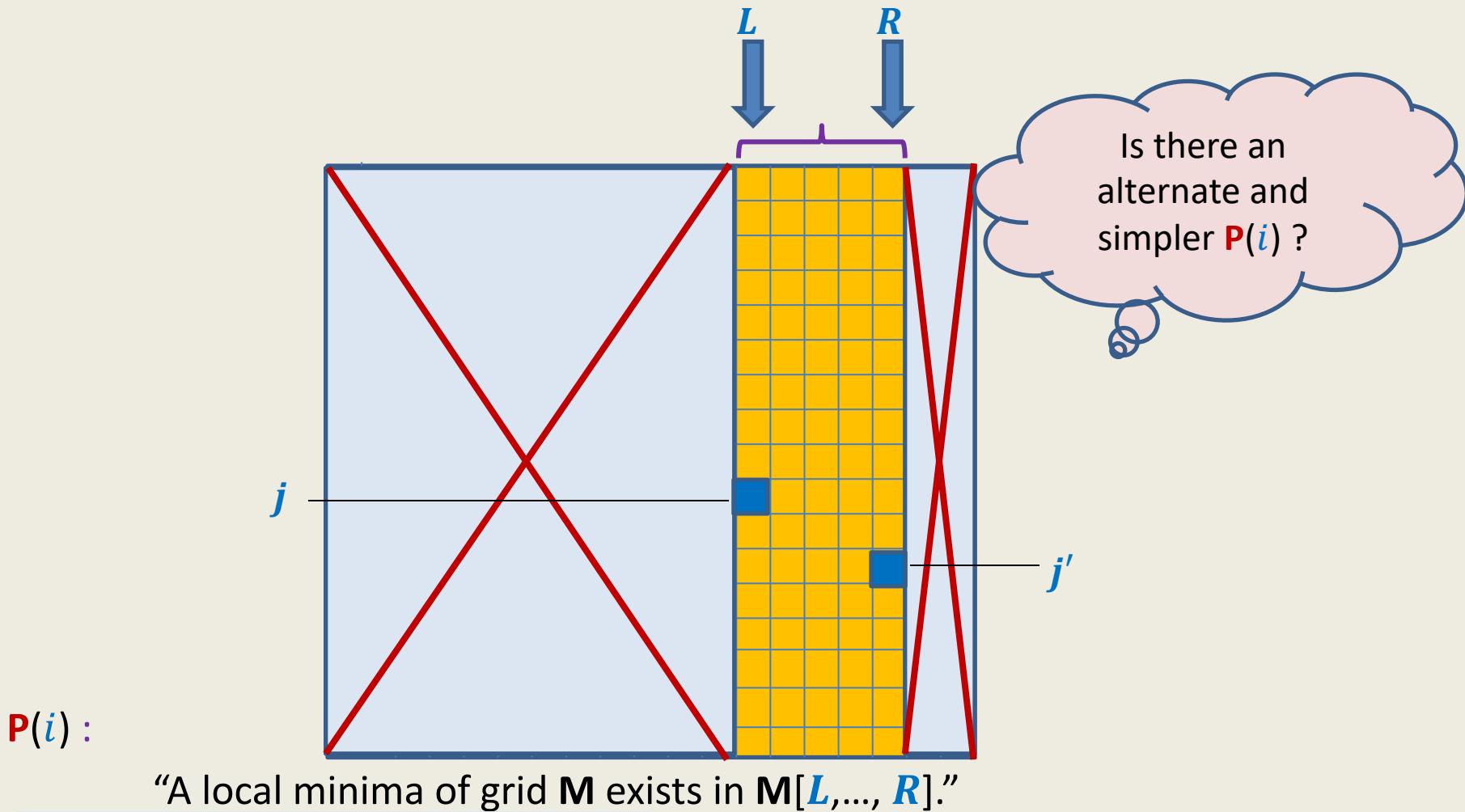
Lessons learnt

- No hand-waving works for iterative algorithms ☺
- We must be sure about
 - What is $P(i)$
 - Proof of $P(i)$.

Let us revisit the ($n \log n$) algorithm

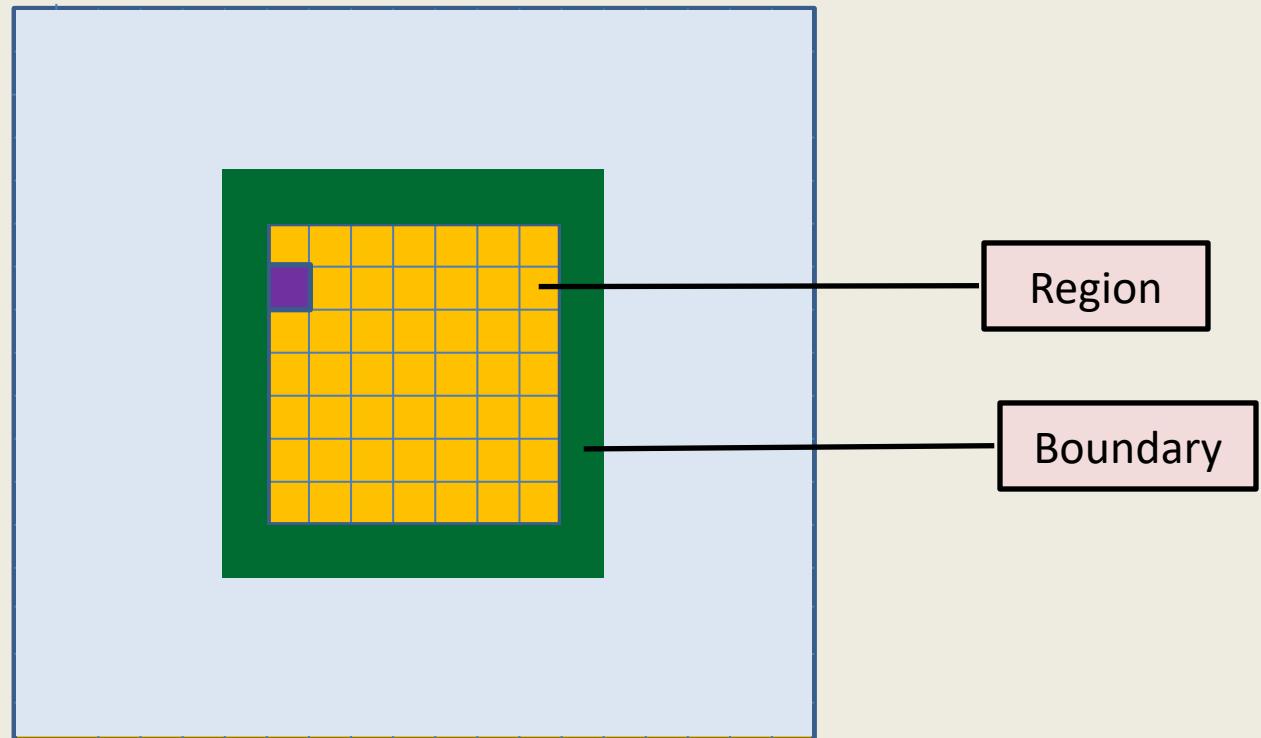
Local minima in a grid

(Proof of Correctness)



$\exists j$ such that $\mathbf{M}[j, L] < \mathbf{M}[\ast, L - 1]$ " and $\exists j'$ such that $\mathbf{M}[j', R] < \mathbf{M}[\ast, R + 1]$ "

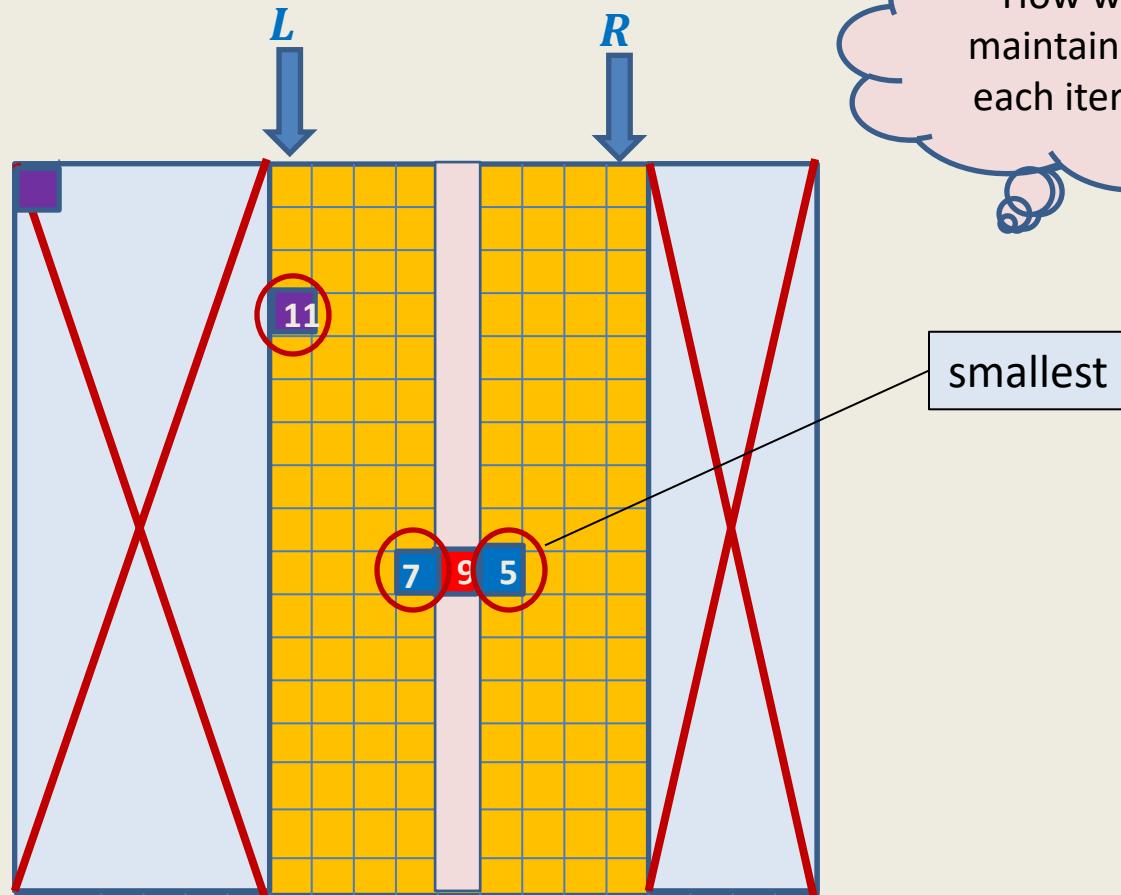
At any stage, what guarantees the existence of local minima in the region ?



- At each stage, our algorithm may maintain a cell in the region whose value is smaller than all elements lying at the **boundary** of the **region** ?
(Note the boundary lies outside the region)

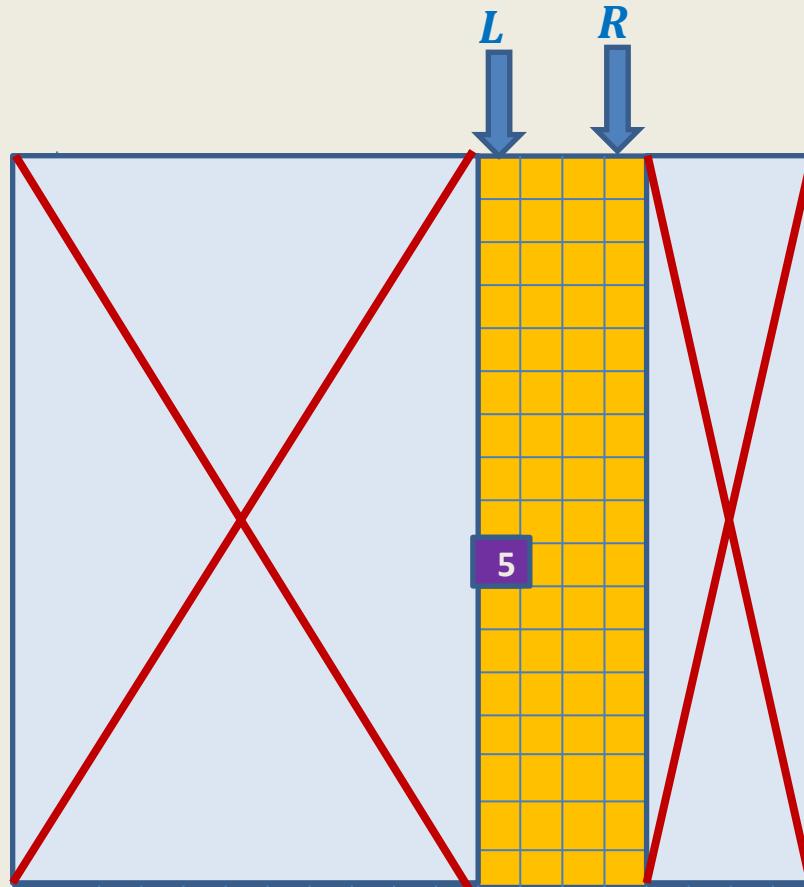
Local minima in a grid

Alternate $O(n \log n)$ algorithm)



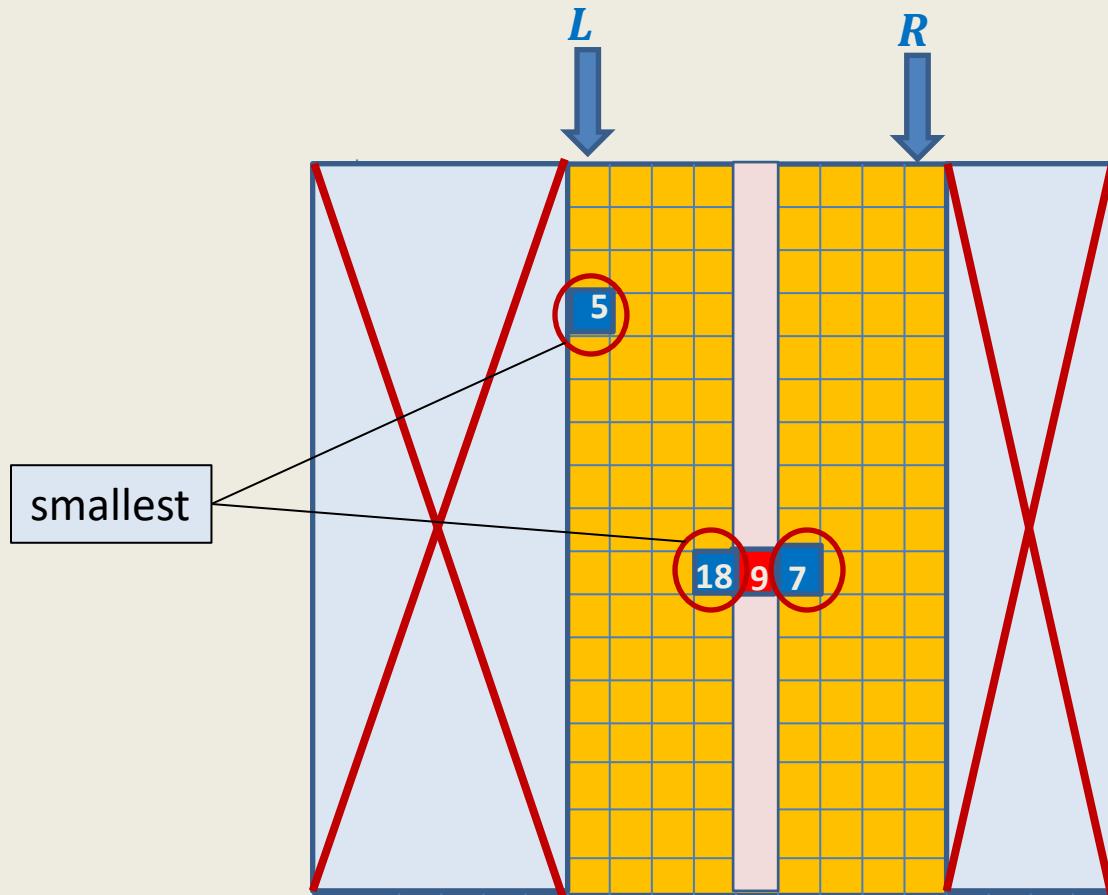
Local minima in a grid

Alternate $O(n \log n)$ algorithm



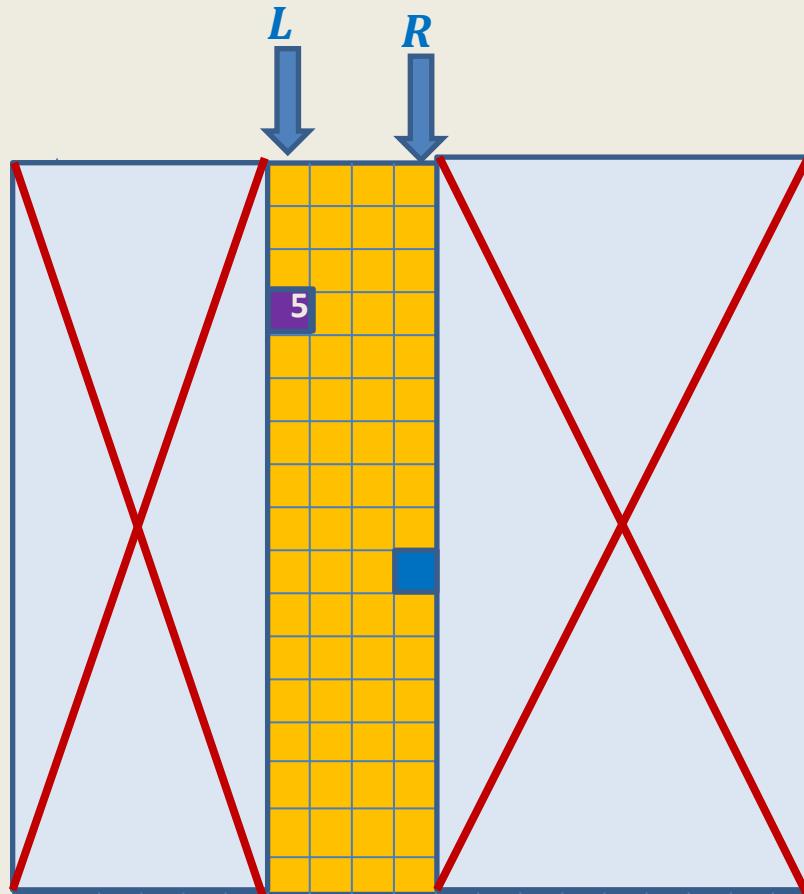
Local minima in a grid

Alternate $O(n \log n)$ algorithm



Local minima in a grid

Alternate $O(n \log n)$ algorithm



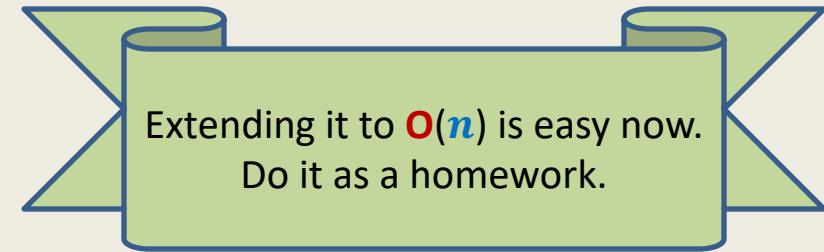
- A neat pseudocode of this algorithm is given on the following slide.

Local minima in a grid

Alternate $O(n \log n)$ algorithm

Int Local-minima-in-grid(M) // returns the column containing a local minima

```
{   L ← 0;  
    R ←  $n - 1$ ;  
    found ← FALSE;  
    C ← (0,0);  
    while(not found)  
    {      mid ← (L + R)/2;  
        If ( $M[* , mid]$  has a local minima)   found ← TRUE;  
        else {  
            let  $M[k, mid]$  be the smallest element in  $M[* , mid]$   
            C' ← ( $k, mid - 1$ );  
            C'' ← ( $k, mid + 1$ );  
            Cmin ← the cell containing the minimum value among {C, C', C''};  
            If (column of Cmin is present in (mid + 1, R))   L ← mid + 1;  
            else R ← mid - 1;  
            C ← Cmin ;  
        }  
    }  
    return mid;  
}
```



Homework: Prove the correctness of this algorithm.

Theorem:

Given an $n \times n$ grid storing n^2 distinct elements, a local minima can be found in $O(n)$ time.

Question:

On which algorithm paradigm, was this algorithm based on ?

- Greedy
- Divide and Conquer
- Dynamic Programming

Proof of correctness of algorithms

Worked out examples:

- **GCD**
- **Binary Search**

GCD

```
GCD(a,b)    // a ≥ b
{
    while (b <> 0)
    {
        t ← b;
        b ← a mod b ;
        a ← t
    }
    return a;
}
```

Lemma (Euclid):

If $n \geq m > 0$, then

$$\gcd(n,m) = \gcd(m, n \bmod m)$$

Proof of correctness of GCD(a,b) :

Let a_i : the value of variable a after i th iteration.

b_i : the value of variable b after i th iteration.

Assertion $P(i)$: $\gcd(a_i, b_i) = \gcd(a, b)$

Theorem : $P(i)$ holds for each iteration $i \geq 0$.

Proof: (By induction on i).

Base case: ($i = 0$) hold trivially.

Induction step:

(Assume $P(j)$ holds, show that $P(j + 1)$ holds too)

$P(j) \rightarrow$

$$\gcd(a_j, b_j) = \gcd(a, b). \quad \dots \quad (1)$$

$(j + 1)$ iteration \rightarrow

$$a_{j+1} = b_j \text{ and } b_{j+1} = a_j \bmod b_j \quad \dots \quad (2)$$

Using Euclid's Lemma and (2),

$$\gcd(a_j, b_j) = \gcd(a_{j+1}, b_{j+1}) \quad \dots \quad (3).$$

Using (1) and (3), assertion $P(j + 1)$ holds too.

Binary Search

```
Binary-Search(A[0...n - 1], x)
```

```
L < 0;
```

```
R < n - 1;
```

```
Found < false;
```

```
While ( L ≤ R and Found = false )
```

```
{   mid < (L+R)/2;
```

```
  If (A[mid] = x) Found < true;
```

```
  else if (A[mid] < x) L < mid + 1 ;
```

```
  else R < mid - 1
```

```
}
```

```
if Found return true;
```

```
else return false;
```

Observation: If the code returns **true**, then indeed **output** is correct.

So all we need to prove is that whenever code returns **false**, then indeed **x** is not present in **A[]**.

This is because **Found** is set to **true** only when **x** is indeed found.

Binary Search

```
Binary-Search(A[0...n - 1], x)
```

```
L  $\leftarrow$  0;
```

```
R  $\leftarrow$  n - 1;
```

```
Found  $\leftarrow$  false;
```

```
While ( L  $\leq$  R and Found = false )
```

```
{   mid  $\leftarrow$  (L+R)/2;
```

```
  If (A[mid] = x) Found  $\leftarrow$  true;
```

```
  else if (A[mid] < x) L  $\leftarrow$  mid + 1 ;
```

```
  else R  $\leftarrow$  mid - 1
```

```
}
```

```
if Found return true;
```

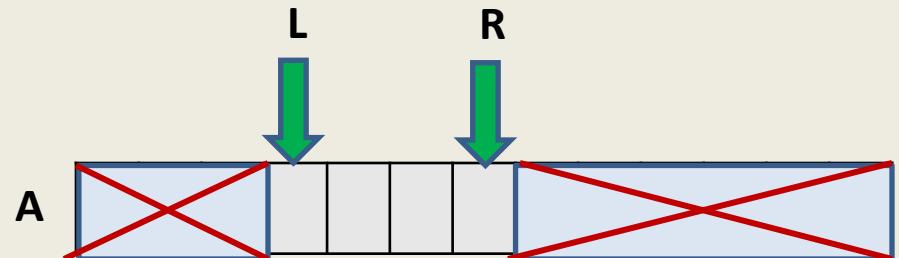
```
else return false;
```

Assertion $P(i)$:

$x \notin \{ A[0], \dots, A[L-1] \}$

and

$x \notin \{ A[R+1], \dots, A[n-1] \}$



Range-Minima Problem

A Motivating example
to realize the importance of data structures

Range-Minima Problem

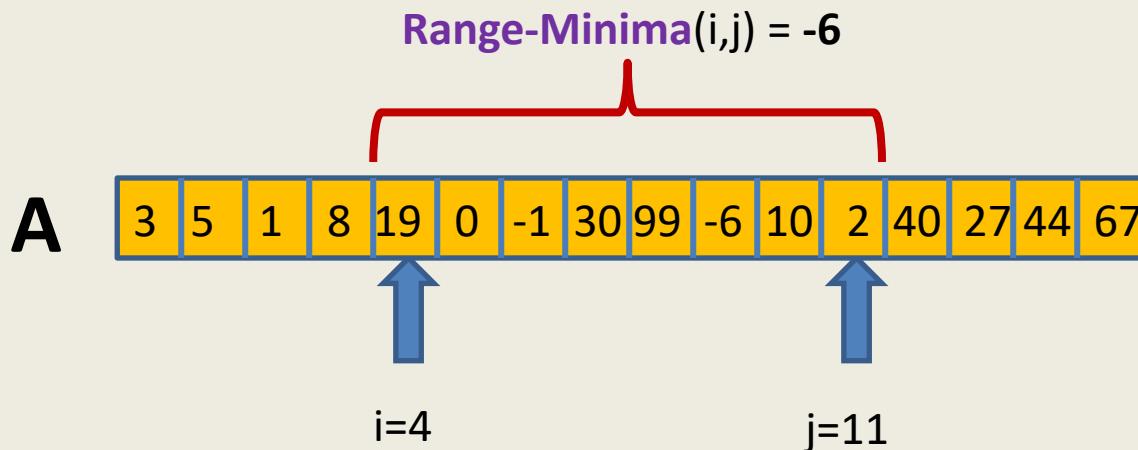
Given: an array **A** storing n numbers,

Aim: a data structure to answer a sequence of queries of the following type

Range-minima(i,j) : report the smallest element from $A[i], \dots, A[j]$

Let **A** store one **million** numbers

Let the number of queries be **10 millions**



Range-Minima Problem

Question: Does there exist a data structure which is

- **Compact**
($O(n \log n)$ size)
- **Can answer each query efficiently ?**
($O(1)$ time)

Homework : Ponder over the above question.

(we shall solve it in the next class)

Data Structures and Algorithms

(ESO207)

Lecture 7:

- **Data structure for Range-minima problem**
Compact and fast

Data structures

AIM:

To organize a data in the memory
so that any query can be answered efficiently.

Example:

Data: A set S of n numbers

Query: “Is a number x present in S ?”

A trivial solution: sequential search

$O(n)$ time per query

A Data structure solution:

- Sort S

$O(n \log n)$ time to build sorted array.

- Use **binary search** for answering query

$O(\log n)$ time per query

Data structures

AIM:

To organize a data in the memory
so that any query can be answered efficiently.

Important assumption:

No. of queries to be answered will be many.

Parameters of Efficiency

- Query time
- Space
- Preprocessing time

RANGE-MINIMA Problem

Range-Minima Problem

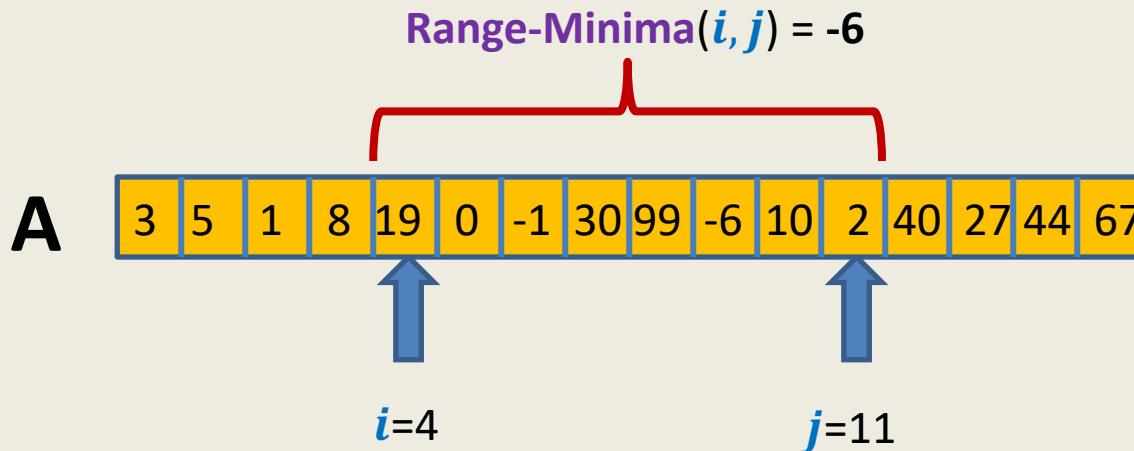
Given: an array \mathbf{A} storing n numbers,

Aim: a data structure to answer a sequence of queries of the following type

Range-minima(i, j) : report the smallest element from $\mathbf{A}[i], \dots, \mathbf{A}[j]$

Let \mathbf{A} store one **million** numbers

Let the number of queries be **10 millions**



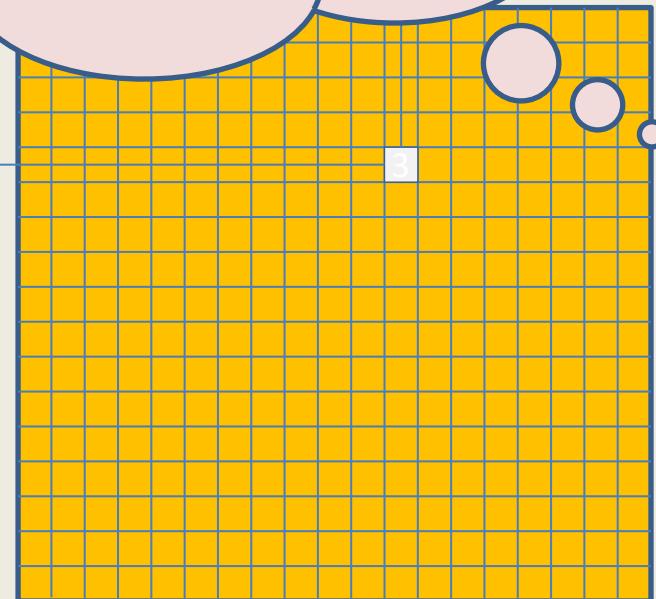
Range-Minima Problem

Solution 1 (brute force)

Range-minima-trivial(i, j)

```
{  temp <-  $i + 1$ ;  
  min <- A[ $i$ ];  
  While(temp <=  $j$ )  
  {    if (min > A[temp])  
        min <- A[temp];  
        temp <- temp+1;  
  }  
  return min  
}
```

Size of **B** is **too large** to be kept in RAM.
So we shall have to keep most of it in
the **Hard disk drive**.
Hence it will take a few **milliseconds per query**.



Time complexity for one query: $O(n)$
(a few **hours** for 10 million queries)



Space : $O(n^2)$

Impractical

Range-Minima Problem

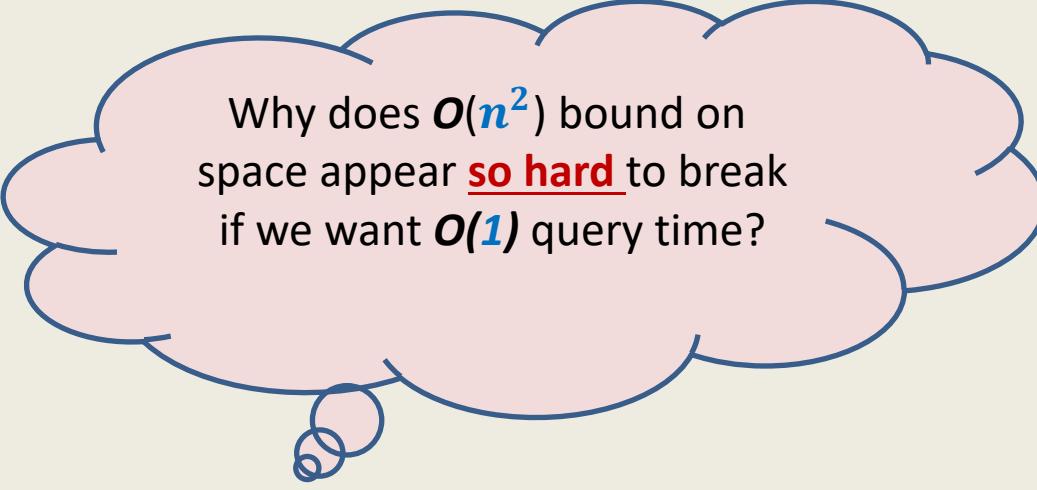
Query:

Report_min(A, i,j) : report smallest element from $\{A[i], \dots, A[j]\}$



Aim :

- **compact** data structure
- **O(1)** **Query time** for any $1 \leq i < j \leq n$.



Why does $O(n^2)$ bound on space appear so hard to break if we want $O(1)$ query time?

... Because of artificial hurdles

Artificial hurdle

If we want to answer each query in $O(1)$ time,

→ we must store its answer explicitly.

→ Since there are around $O(n^2)$ queries,
so $O(n^2)$ space is needed.

Spend some time to find the origin of this hurdle....

Artificial hurdle



... If we fix the first parameter i for all queries, we need $O(n)$ space.

True Fact



for all i , we need $O(n^2)$ space.

A wrong inference

because it assumes that data structure for an index i will work in total isolation of others.

Collaboration (team effort) works in real life



Why not try
collaboration for the
given problem ?

Range-minima problem: Breaking the $O(n^2)$ barrier using collaboration

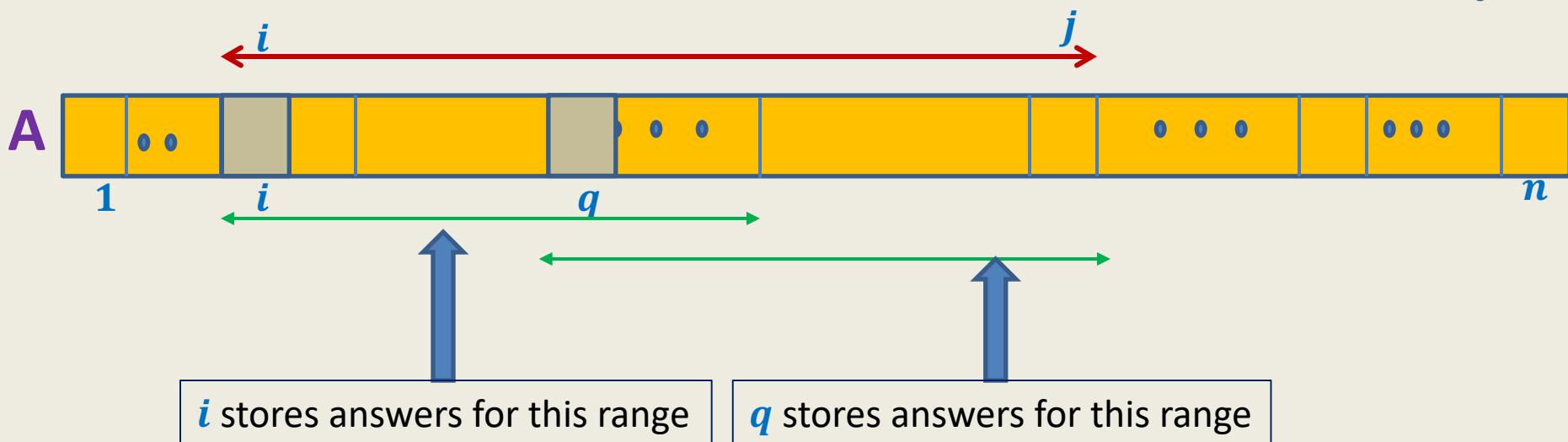
An Overview:

- Keep n tiny data structures:
Each index i stores minimum only for a few $j > i$.
- For a query **Range-minima(i, j)**,
if the answer is not stored in the tiny data structure of i ,
look up tiny data structure of some index q (chosen carefully).

**HOW DOES COLLABORATION WORK
IN THIS PROBLEM ?**

Range-minima problem: Breaking the $O(n^2)$ barrier using collaboration

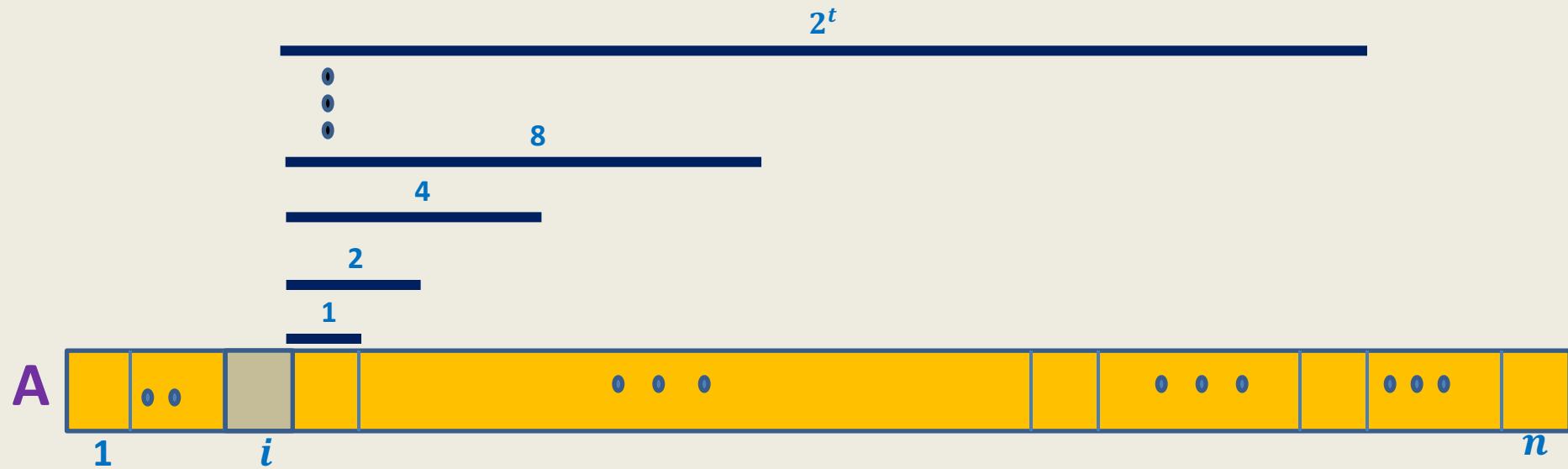
We may use the tiny data structure of index q to answer Range-Minima(i, j)



DETAILS OF TINY DATA STRUCTURES

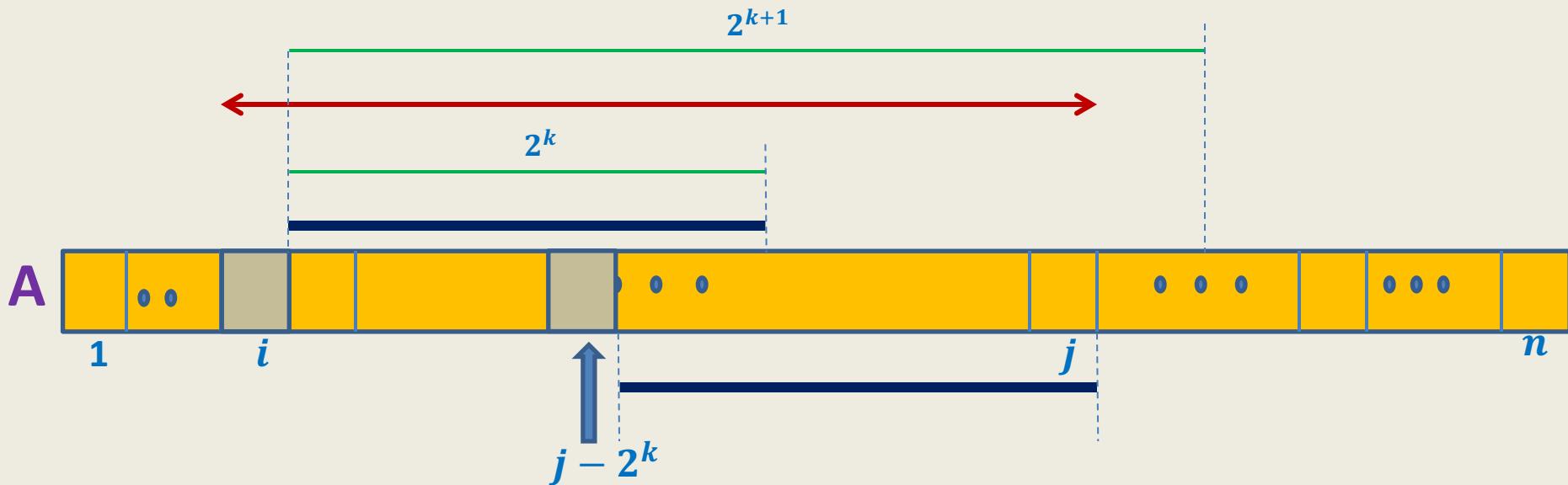
Range-minima problem :

Details of tiny data structure stored at each i



Tiny data structure of Index i stores minimum element for $\{A[i], \dots, A[i + 2^k]\}$ for each $k \leq \log_2 n$

Answering Range-minima query for index i : Collaboration works



We shall use two additional arrays

Definition :

Power-of-2[m] : the greatest number of the form 2^k such that $2^k \leq m$.

Examples: Power-of-2[5] = 4,

Power-of-2[19]= 16,

Power-of-2[32]=32.

Definition :

Log[m] : the greatest integer k such that $2^k \leq m$.

Examples: Log[5] = 2,

Log[19]= 4,

Log[32]=5.

Homework: Design $O(n)$ time algorithm to compute arrays **Power-of-2[]** and **Log[]** of size n .

FINAL SOLUTION FOR RANGE MINIMA PROBLEM

Range-Minima Problem:

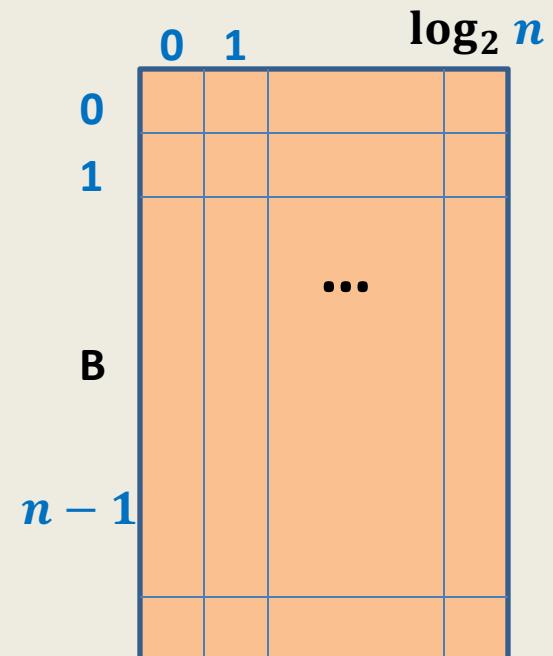
Data structure with $O(n \log n)$ space and $O(1)$ query time

Data Structure:

- $n \times \log n$ matrix \mathbf{B} where $\mathbf{B}[i][k]$ stores minimum of $\{\mathbf{A}[i], \mathbf{A}[i+1], \dots, \mathbf{A}[i+2^k]\}$
- Array **Power-of-2[]**
- Array **Log[]**

Range-minima-(i, j)

```
{   L ← j - i;  
    t ← Power-of-2[L];  
    k ← Log[L];  
    If (t = L) return B[i][k];  
    else      return min( B[i][k] , B[j - t][k] );  
}
```



Theorem:

There is a data structure for range-minima problem that takes
 $O(n \log n)$ space and **$O(1)$ query time.**

Preprocessing time:

$O(n^2 \log n)$: Trivial

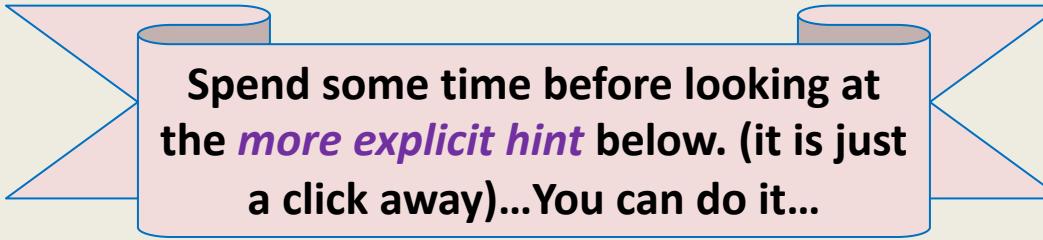
$O(n \log n)$: Doable with **little hints**

Homework:

Design an $O(n \log n)$ time algorithm

to build the $n \times \log n$ matrix \mathbf{B} used in data structure of Range-Minima problem.

Hint: (Inspiration from iterative algorithm for Fibonacci numbers).



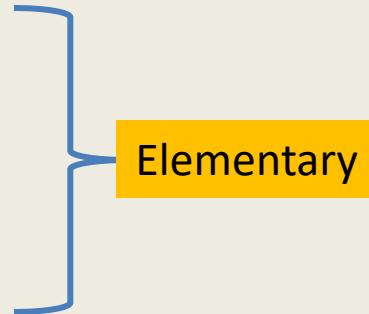
Spend some time before looking at the *more explicit hint* below. (it is just a click away)... You can do it...

To compute $\mathbf{B}[i][k]$, you need to know only two entries from column **.

Data structures

(To be discussed in the course)

- Arrays
- Linked Lists
- Stacks
- Queues



Tree Data Structures:

- Binary heap
- Binary Search Trees
- Augmented Data structures

Data Structures for integers:

- Hash Tables
- Searching in $O(\log \log n)$ time (if time permits)

Data Structures and Algorithms

(ESO207)

Lecture 8:

Data structures:

- **Modeling** versus **Implementation**
- Abstract data type “**List**” and its implementation

Data Structure

Definition:

A collection of data elements *arranged* and *connected* in a way
that can facilitate efficient executions
of a (potentially long) sequence of operations.

Two steps process for designing a Data Structure

Step 1: Mathematical Modeling

A **Formal** description of the possible operations of a data structure.

Operations can be classified into two categories:

Query Operations: Retrieving some information from the data structure

Update operations: Making a change in the data structure

Outcome of Mathematical Modeling: an **Abstract Data Type**

Step 2: Implementation

Explore the ways of organizing the data that facilitates performing each operation efficiently using the exist-

Since we don't specify here the way how each operation of the data structure will be implemented

MODELING OF LIST

OUTCOME WILL BE:
ABSTRACT DATA TYPE “LIST”

Mathematical Modeling of a List

- List of Roll numbers passing a course.
- List of Criminal cases pending in High Court.
- List of Rooms reserved in a hotel.
- List of Students getting award in IITK convocation 2018.

What is common in all these examples ?

Inference: List is a sequence of elements.

$L: a_1, a_2, \dots, a_{i-1}, a_i, a_{i+1}, \dots, a_n$



*i*th element of list L

Query Operations on a List

- **IsEmpty(L)**: determine if L is an empty list.
- **Search(x,L)**: determine if x appears in list L.
- **Successor(p,L)**:

The type of this parameter
will depend on the implementation

return the element of list L which succeeds/follows the element at location p.

Example:

If L is $a_1, a_2, \dots, a_{i-1}, a_i, a_{i+1}, \dots, a_n$ and p is location of element a_i ,
then Successor(p,L) returns a_{i+1} .

- **Predecessor(p,L)**:
Return the element of list L which precedes (appears before) the element at
location p.

Other possible operations:

- **First(L)**: return the first element of list L.
- **Enumerate(L)**: Enumerate/print all elements of list L in the order they appear.

Update Operations on a List

- **CreateEmptyList(L)**: Create an empty list.
- **Insert(x,p,L)**: Insert x at a given location p in list L .

Example: If L is $a_1, \dots, a_{i-1}, a_i, a_{i+1}, \dots, a_n$ and p is location of element a_i , then after **Insert(x,p,L)**, L becomes

$$a_1, \dots, a_{i-1}, x, a_i, a_{i+1}, \dots, a_n$$

- **Delete(p,L)**: Delete element at location p in L

Example: If L is $a_1, \dots, a_{i-1}, a_i, a_{i+1}, \dots, a_n$ and p is location of element a_i , then after **Delete(p,L)**, L becomes

$$a_1, \dots, a_{i-1}, a_{i+1}, \dots, a_n$$

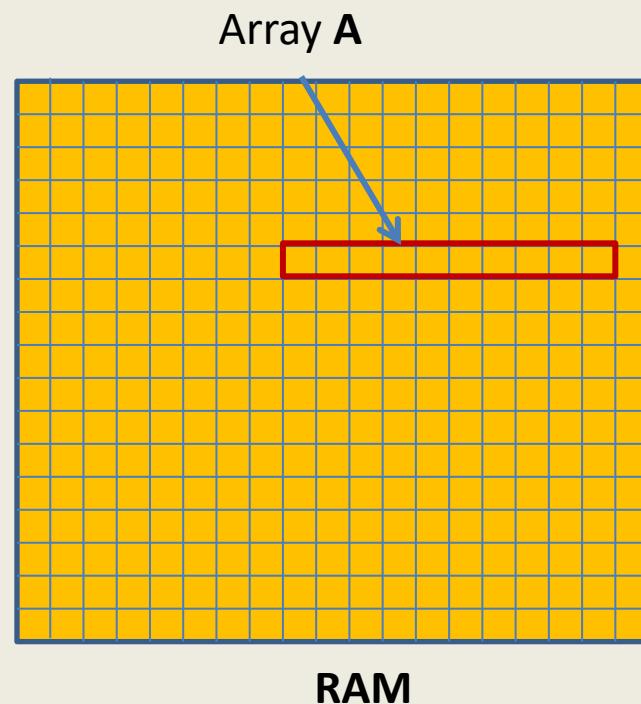
- **MakeListEmpty(L)**: Make the List L empty.

IMPLEMENTATION OF ABSTRACT DATA TYPE “LIST”

Array based Implementation

Taught in a Computer Hardware course.

- RAM allows $O(1)$ time to access any memory location.
- Array is a contiguous chunk of memory kept in RAM.
- For an array $A[]$ storing n words, the address of element $A[i] =$ “start address of array A ” + i



Array based Implementation

- Store the elements of List in array **A** such that $A[i]$ denotes $(i + 1)$ th element of the list at each stage (since index starts from 0).
(Assumption: The maximum size of list is known in advance.)
- Keep an integer variable **Length** to denote the number of elements in the list at each stage.

Example: If at any moment of time List is **3,5,1,8,0,2,40,27,44,67**, then the array **A** looks like:



Question: How to describe location of an element of the list ?

Answer: by the corresponding array index. Location of 5th element of List is 4₁₀

Time Complexity of each List operation using Array based implementation



Arrays are very **rigid**

Operation	Time Complexity per operation
IsEmpty(L)	O(1)
Search(x,L)	O(n)
Successor(<i>i</i> ,L)	O(1)
Predecessor(<i>i</i> ,L)	O(1)
CreateEmptyList(L)	O(1)
Insert(x, <i>i</i> ,L)	O(n)
Delete(<i>i</i> ,L)	O(n)
MakeListEmpty(L)	O(1)

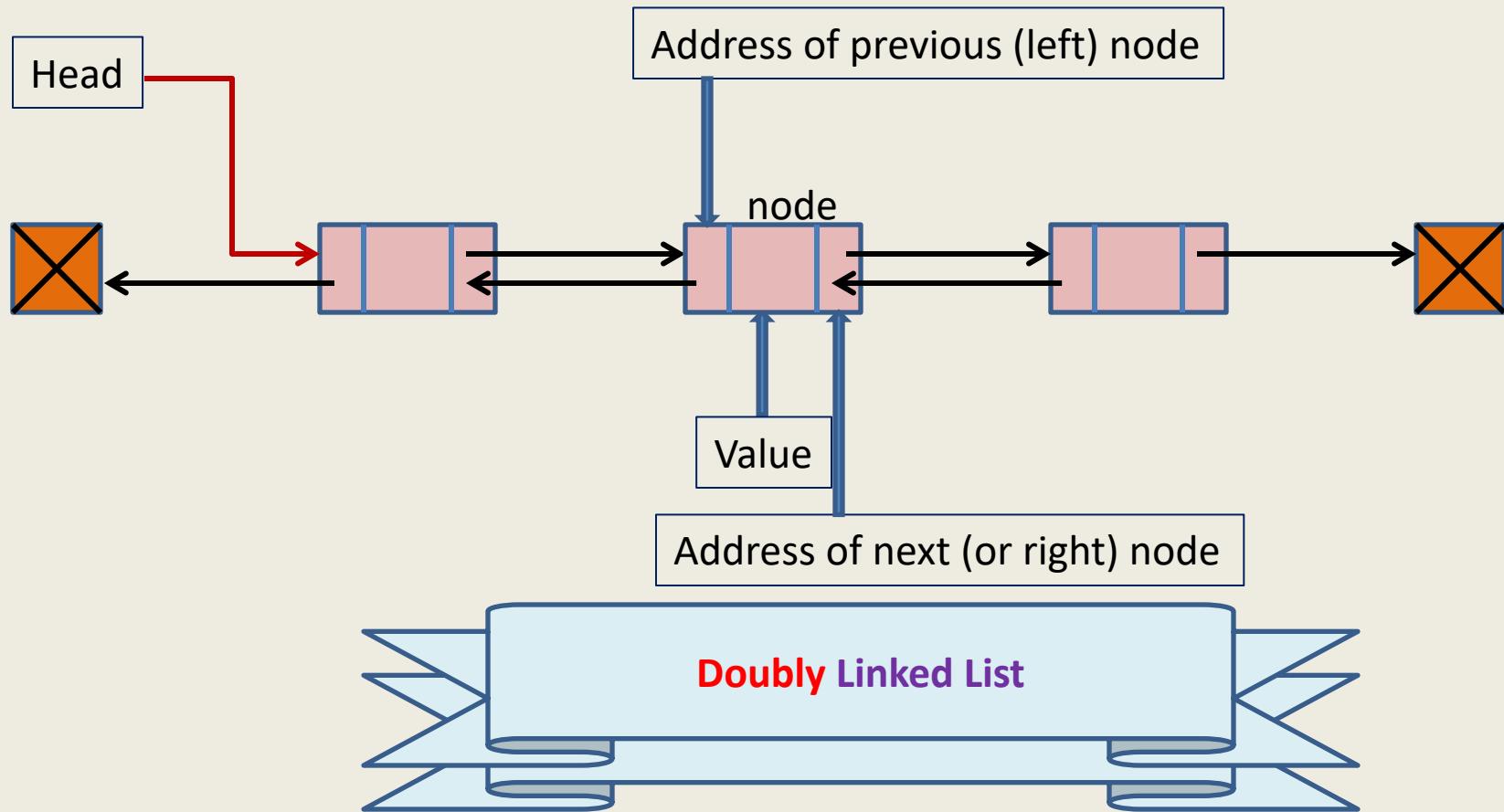
n: number of elements in list at present

All elements from A[*i*] to A[*n* - 1] have to be shifted to the **right** by one place.

All elements from A[*i* + 1] to A[*n* - 1] have to be shifted to the **left** by one place.

operation with matching complexity.

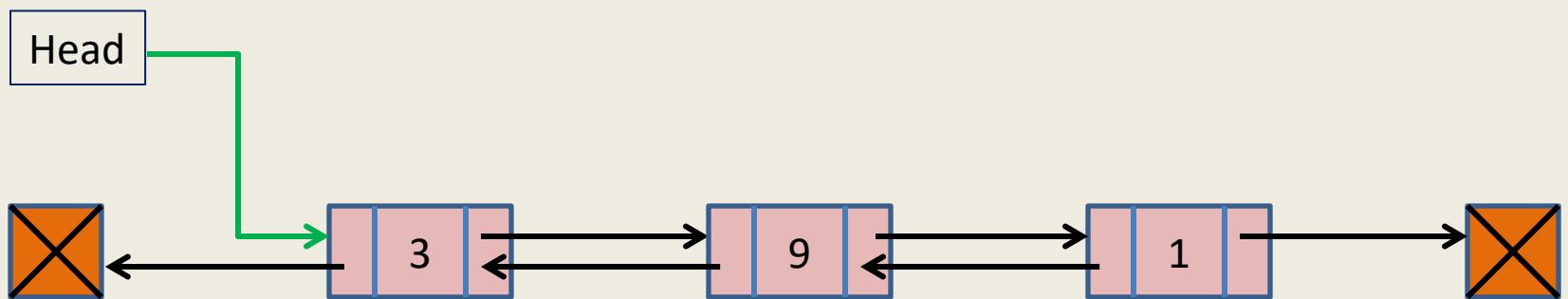
Link based Implementation:



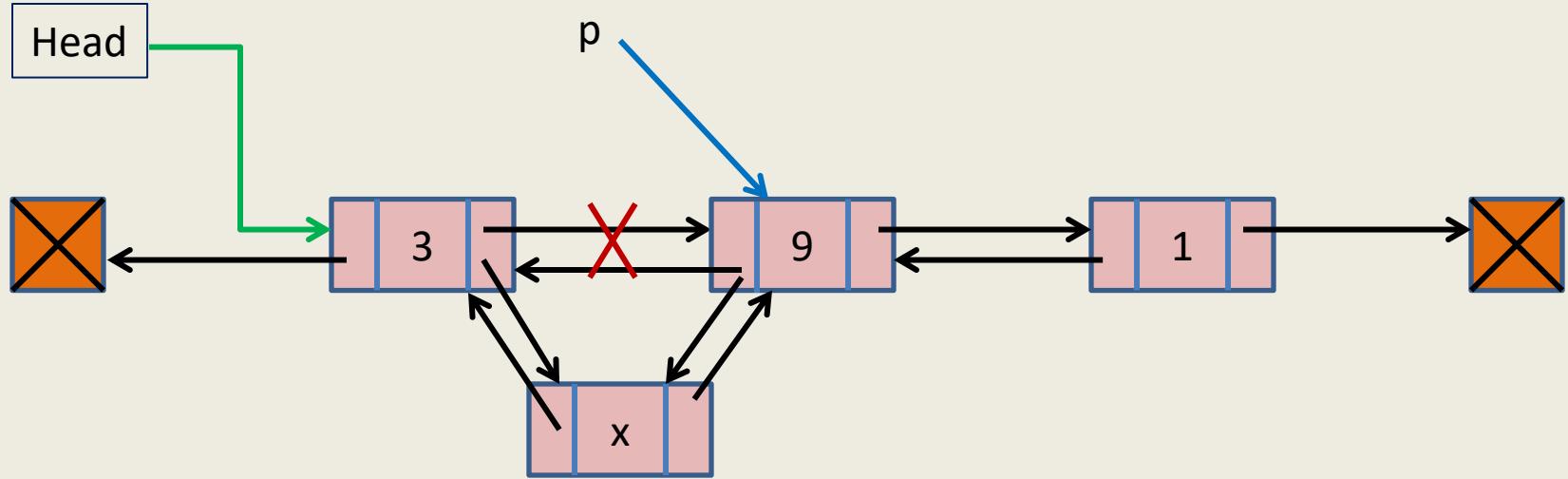
Doubly Linked List based Implementation

- Keep a doubly linked list where elements appear in the order we follow while traversing the list.
- The location of an element : the address of the node containing it.

Example: List **3,9,1** appears as

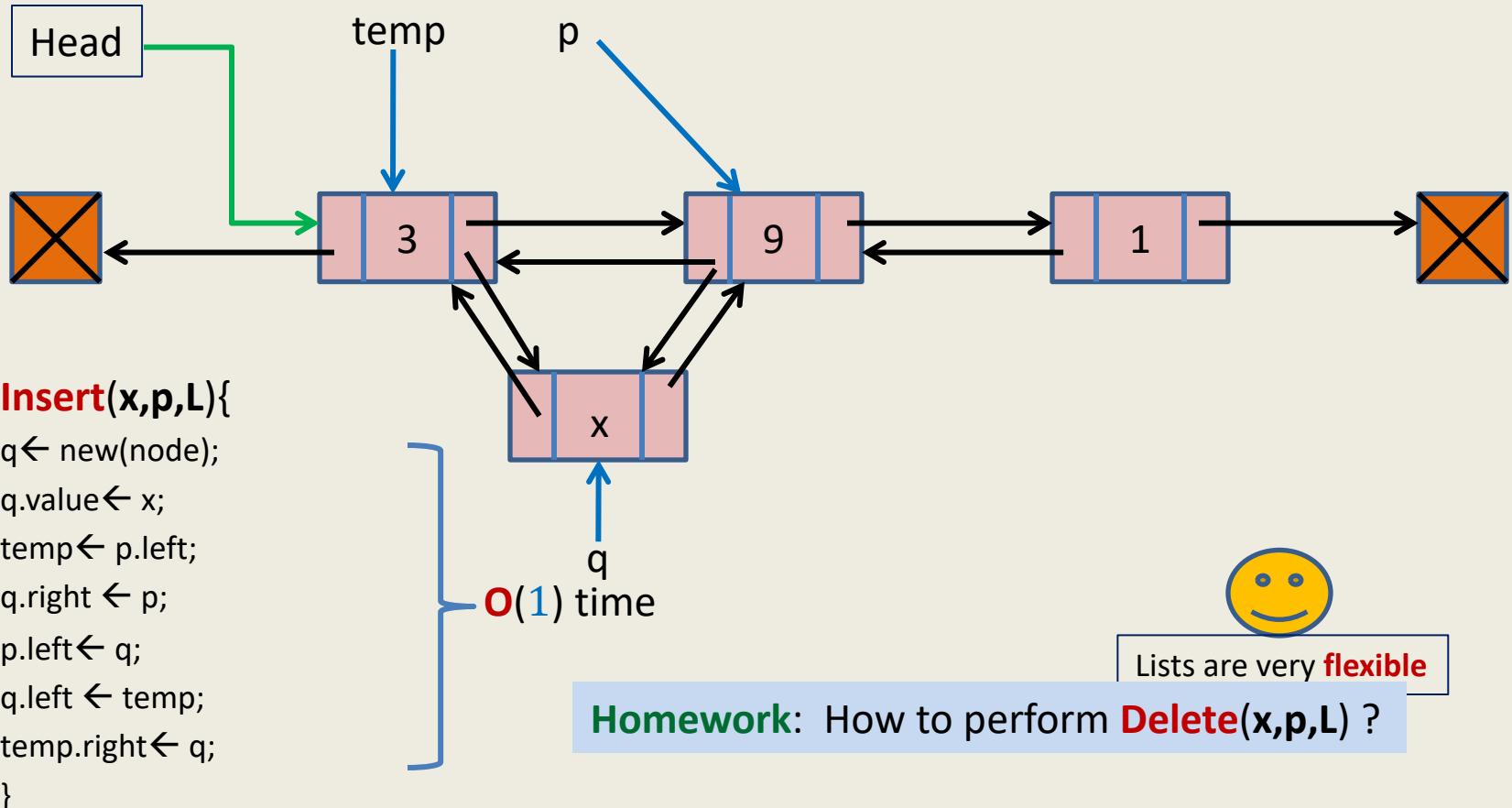


How to perform **Insert(x,p,L)** ?



How is it done actually ?

How to perform **Insert(x,p,L)** ?



A Common Mistake

Often students interpret the parameter p in $\text{Insert}(x,p,L)$ with the integer signifying the order (1^{st} , 2^{nd} , ...) at which element x is to be inserted in list L .

Based on this interpretation, they think that $\text{Insert}(x,p,L)$ will require scanning the list and hence will take time of the order of p and not $O(1)$.

Here is my advice:

Please refer to the **modeling** of the list for exact interpretation of p .

The implementation in the lecture is for that modeling and indeed takes $O(1)$ time.

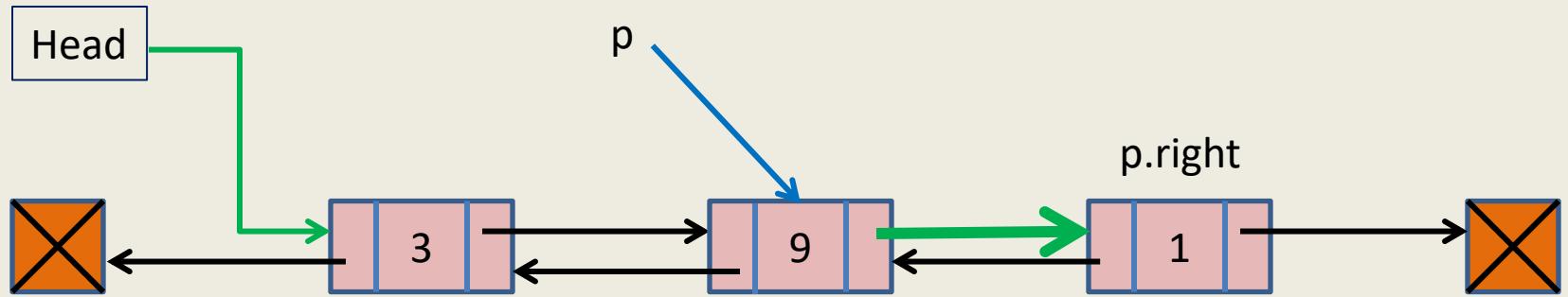
However, if you wish to model list *differently* such that the parameter p is an integer parameter as defined above, then you are right.

Lesson learnt:

Implementation of a data structure

depends upon its mathematical modeling (interpretation of various operations).

How to perform **successor(p,L)** ?



Successor(p,L){

```
q← p.right;  
return q.value;  
}
```

Time Complexity of each List operation using Doubly Linked List based implementation

Operation	Time Complexity per operation
IsEmpty(L)	$O(1)$
Search(x,L)	$O(n)$
Successor(p,L)	$O(1)$
Predecessor(p,L)	$O(1)$
CreateEmptyList(L)	$O(1)$
Insert(x,p,L)	$O(1)$
Delete(p,L)	$O(1)$
MakeListEmpty(L)	$O(1)$

It takes $O(1)$ time if we implement it by setting the **head** pointer of list to NULL. However, if one has to **free** the memory used by the list, then it will require traversal of the entire list and hence $O(n)$ time. You might learn more about it in Operating System course.

Homework: Write C Function for each operation with matching complexity.

Doubly Linked List based implementation versus array based implementation of “List”

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

A CONCRETE PROBLEM

Problem

Maintain a telephone directory

Operations:

- Search the phone # of a person with name x
- Insert a new record (name, phone #,...)

Array based solution	Linked list based solution
$\text{Log } n$	$O(n)$
$O(n)$	$O(1)$

Yes. Keep the array sorted according to the **names** and do Binary search for x .

Can we achieve **the best of the two data structure simultaneously ?**

We shall together invent such a **novel data structure** in the next class

Data Structures and Algorithms

(ESO207)

Lecture 9:
Inventing a new Data Structure with

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

Important Notice

There are basically two ways of introducing a new/innovative solution of a problem.

1. One way is to just explain it without giving any clue as to how the person who invented the concept came up with this solution.
2. Another way is to start from scratch and take a journey of the route which the inventor might have followed to arrive at the solution.

This journey goes through various hurdles and questions, each hinting towards a better insight into the problem if we have patience and open mind.

Which of these two ways is better ?

I believe that the second way is better and more effective.

The current lecture is based on this way. The data structure we shall **invent** is called **a Binary Search Tree**. This is the most fundamental and versatile data structure. We shall realize this fact many times during the course ...

Doubly Linked List based implementation versus array based implementation of “List”

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



Arrays are very **rigid**

Problem

Maintain a telephone directory

Operations:

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

Array based solution	Linked list based solution
$\text{Log } n$	$O(n)$
$O(n)$	$\text{Log } n$



Can we achieve **the best of** the two data structure simultaneously ?

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

Inventing a new data structure

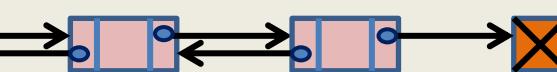


Lists are flexible, so let us try modifying the linked list structure to achieve fast **search** time.

Head



Lists

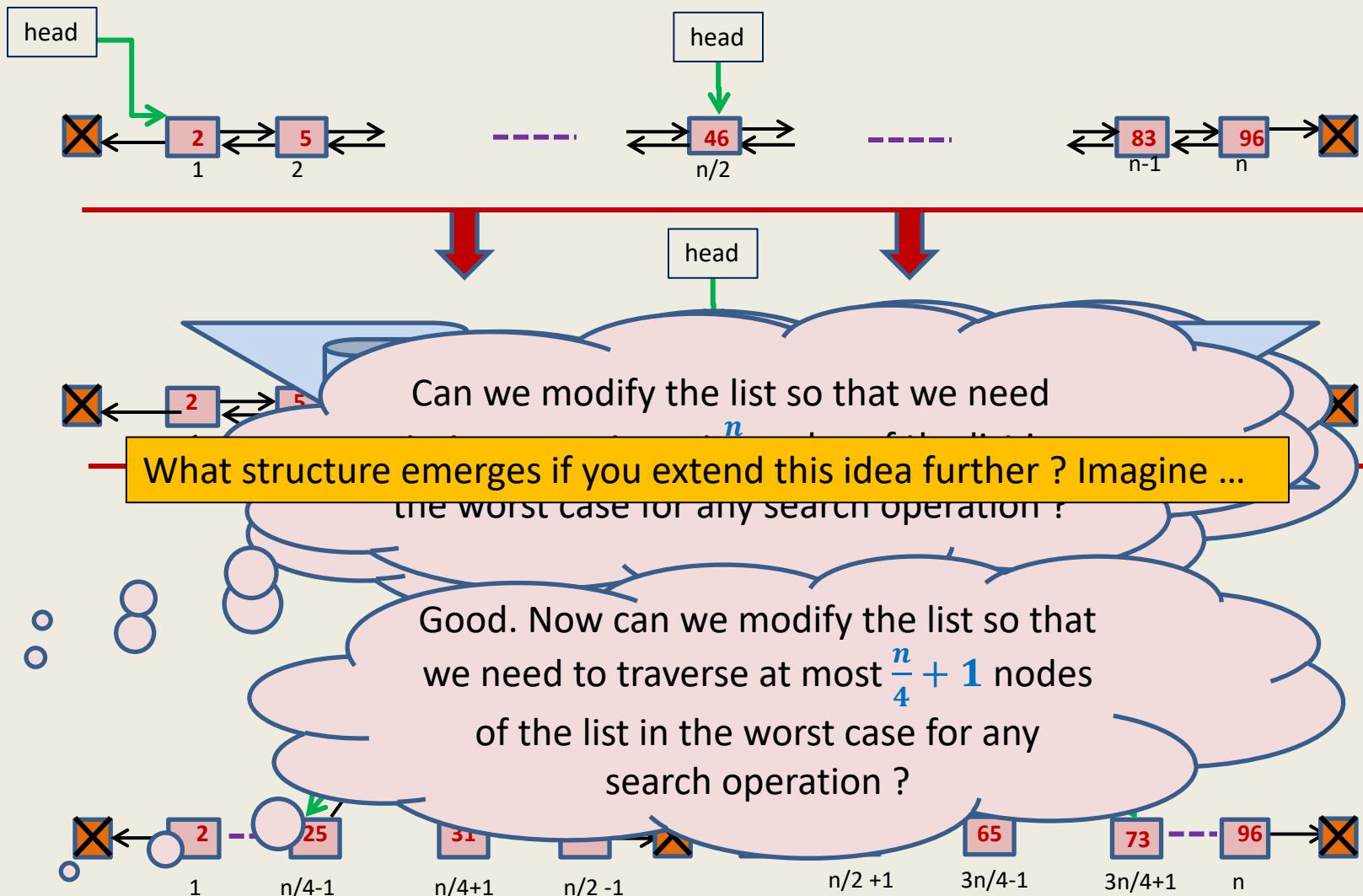


Too Rigid for updates

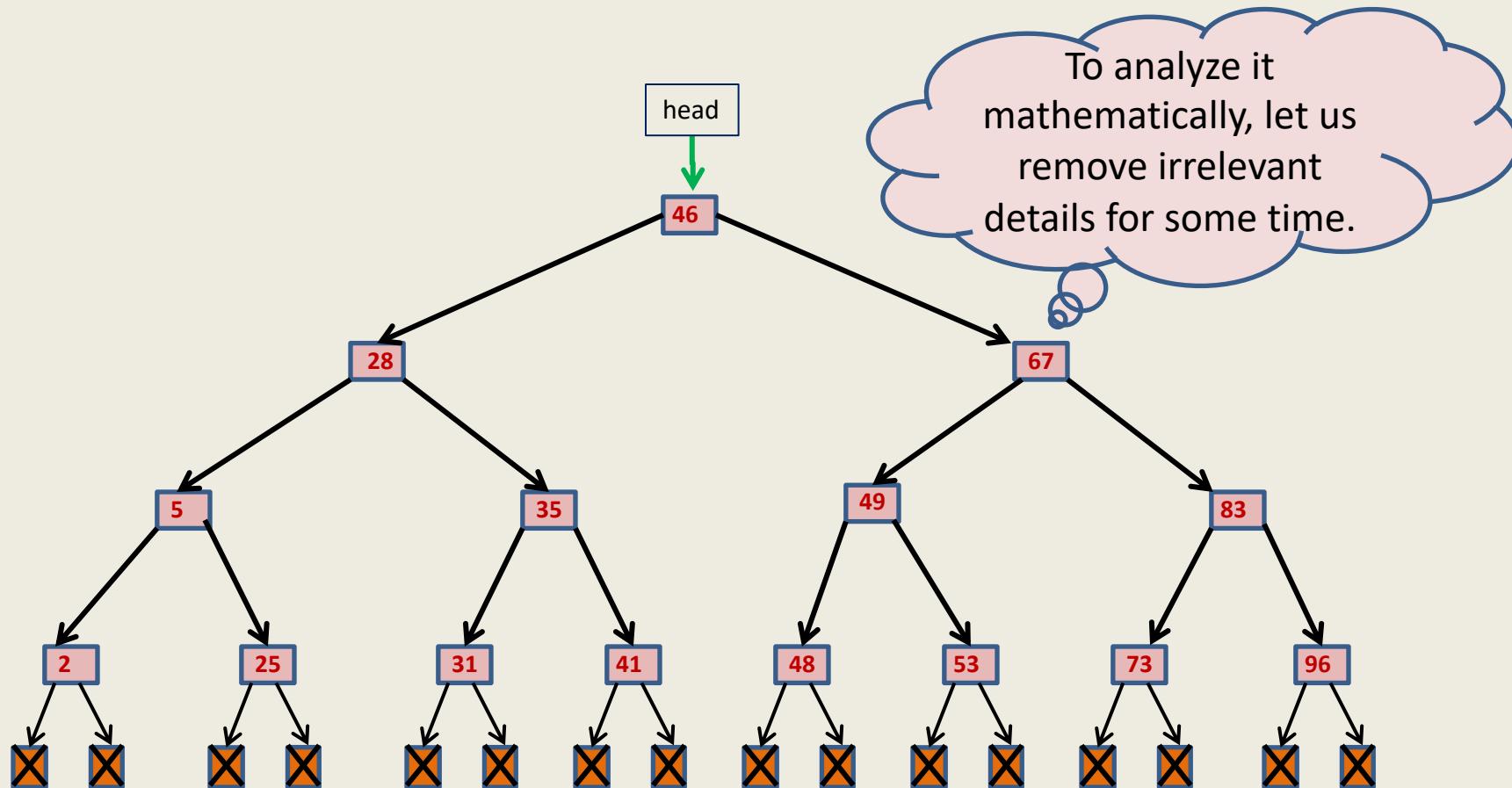


Array

Restructuring doubly linked list

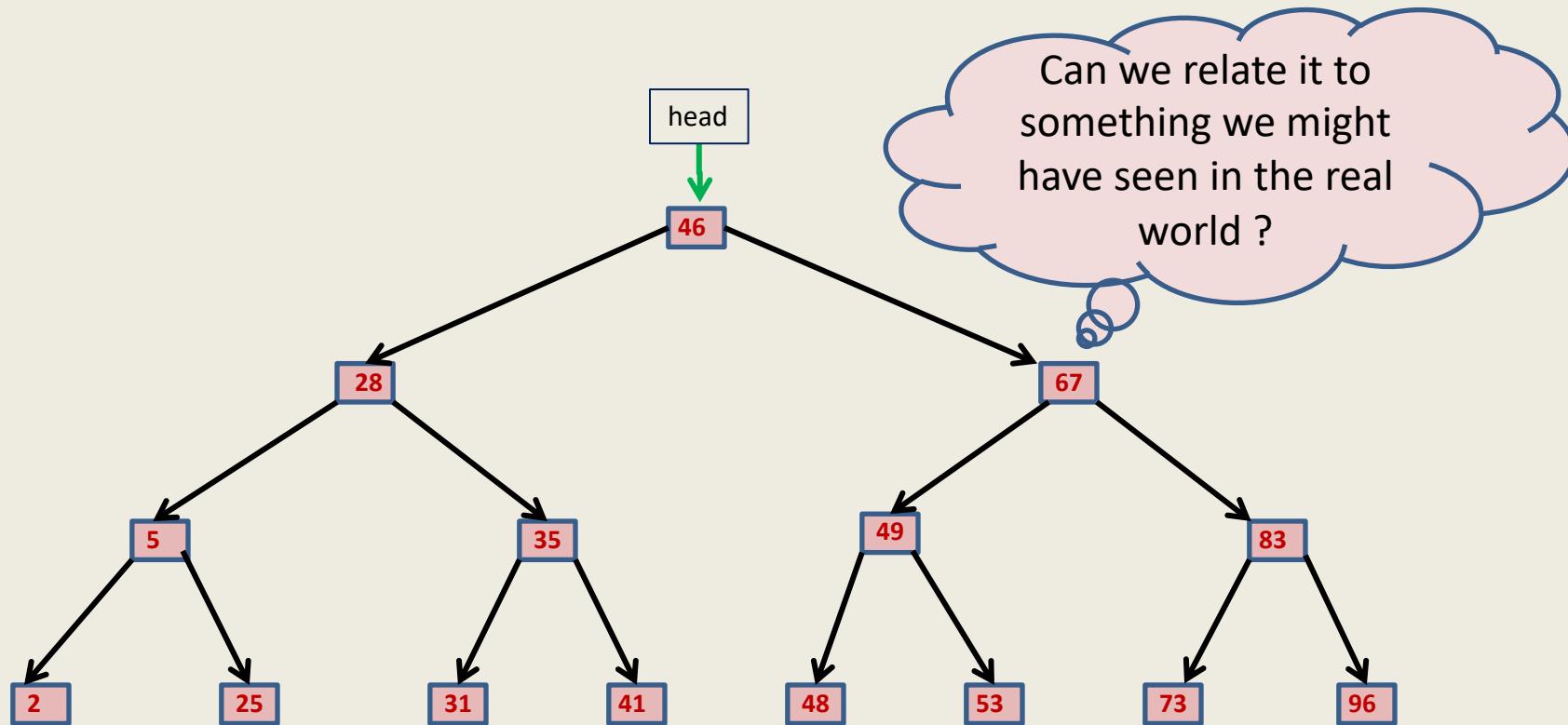


A new data structure emerges



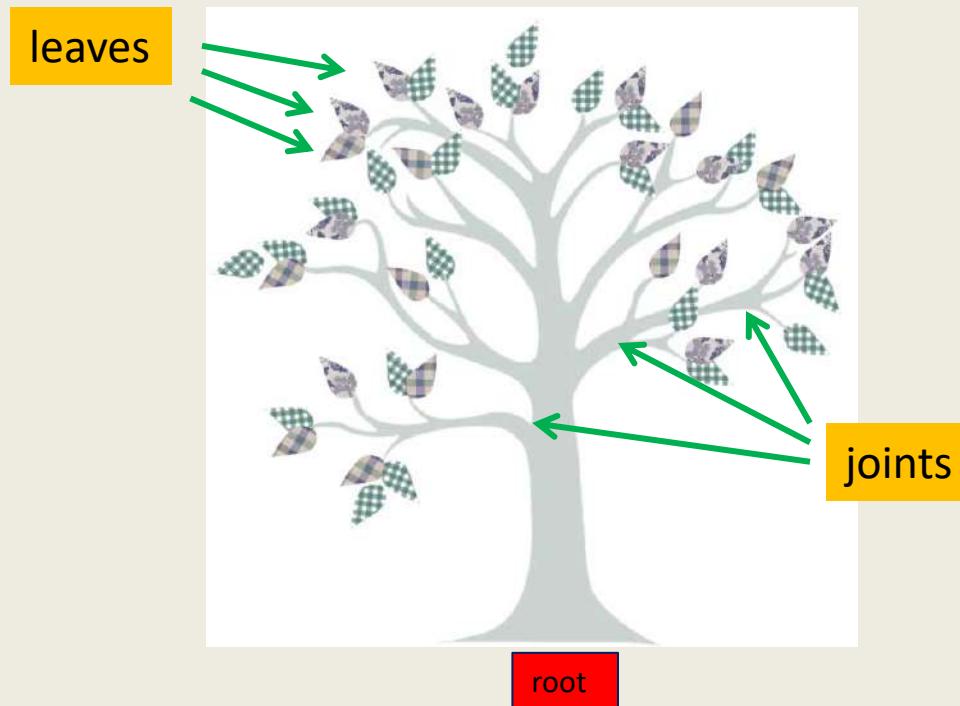
To analyze it
mathematically, let us
remove irrelevant
details for some time.

A new data structure emerges



Can we relate it to something we might have seen in the real world ?

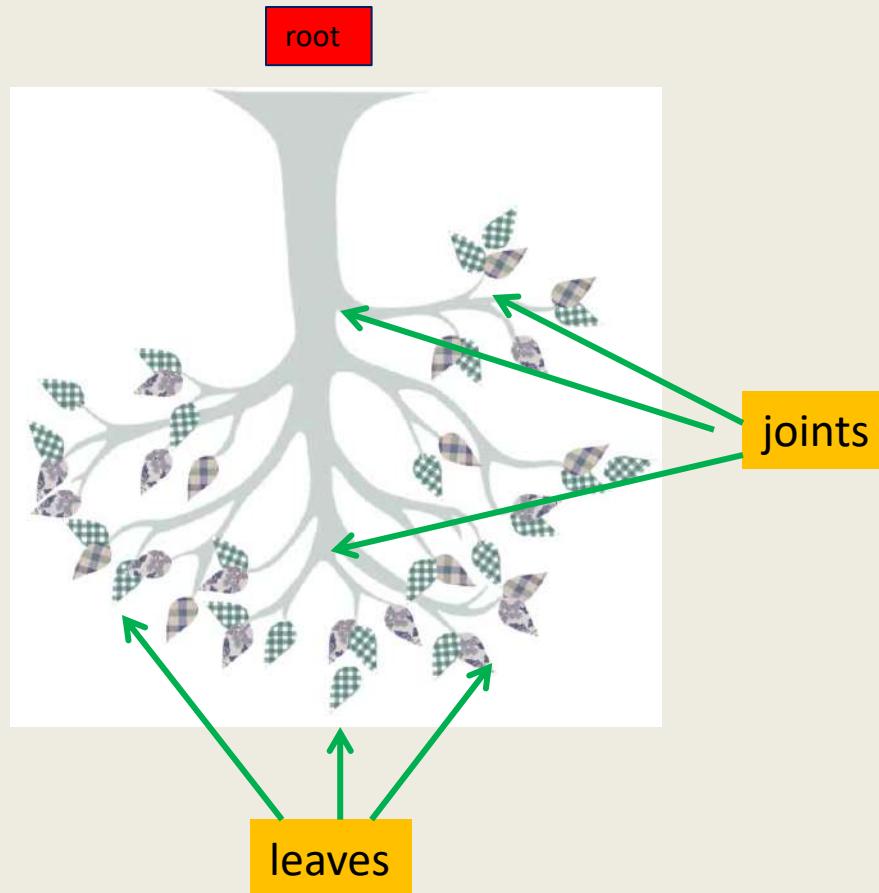
Nature : a great source of **inspiration**



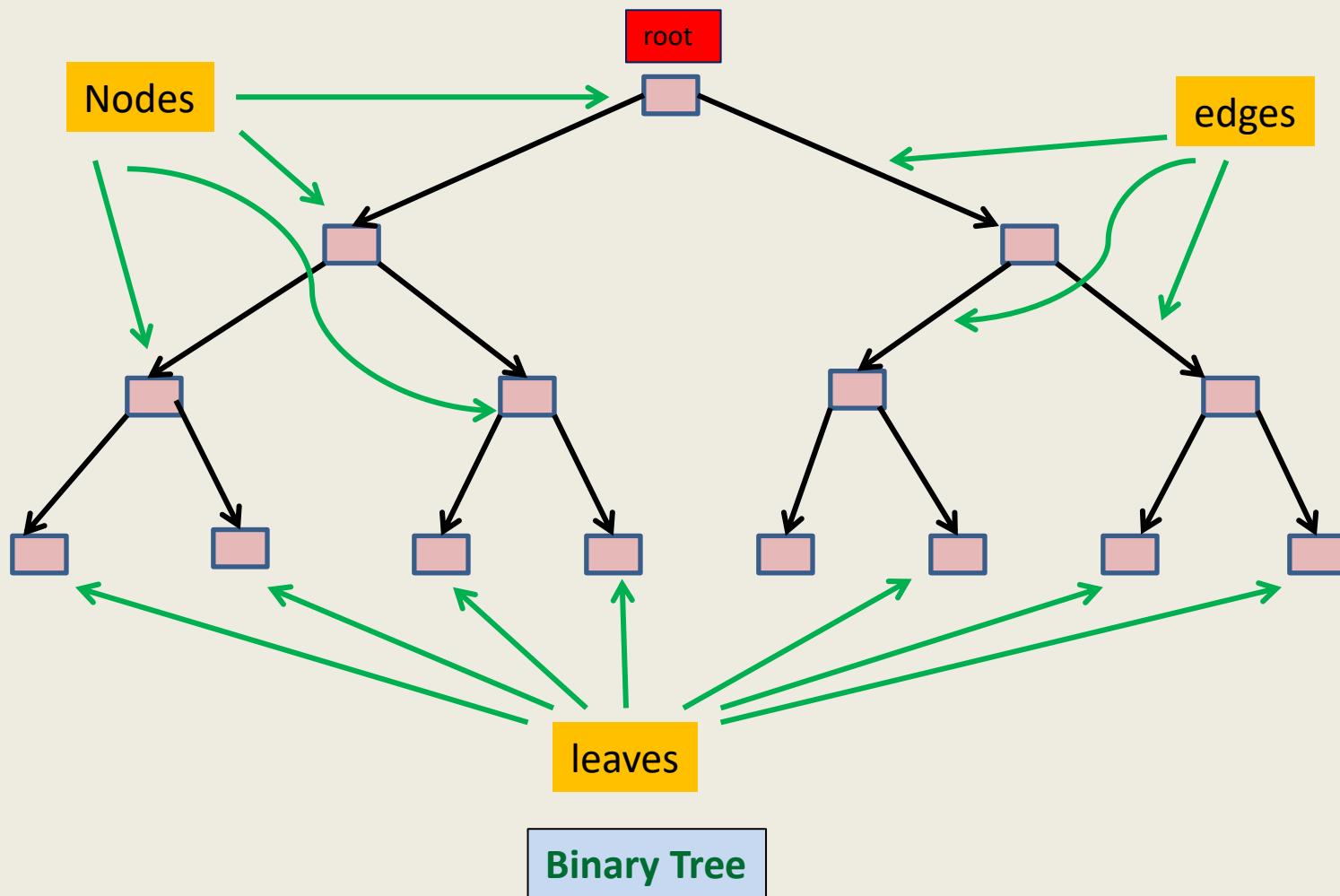
Nature : a great source of **inspiration**



Nature : a great source of **inspiration**



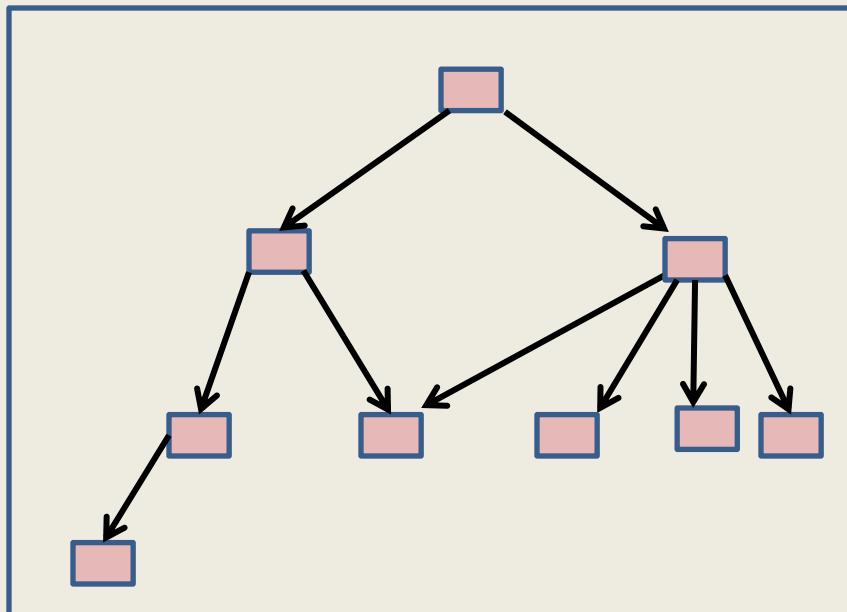
Nature : a great source of inspiration



Binary Tree: A mathematical model

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

1. There is exactly one node with no incoming edge.
This node is called the **root** of the tree.
2. Every node other than root node has **exactly one** incoming edge.
3. Each node has **at most two** outgoing edges.

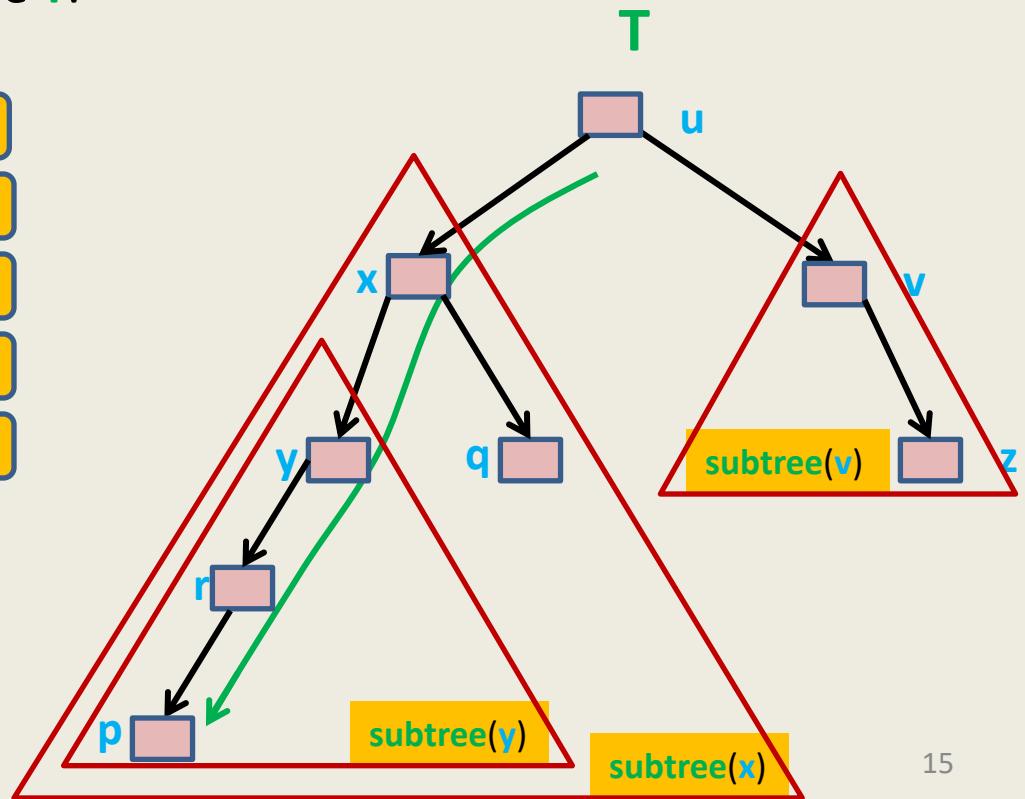


Which of these
are **not** **binary**
trees ?

Binary Tree: some terminologies

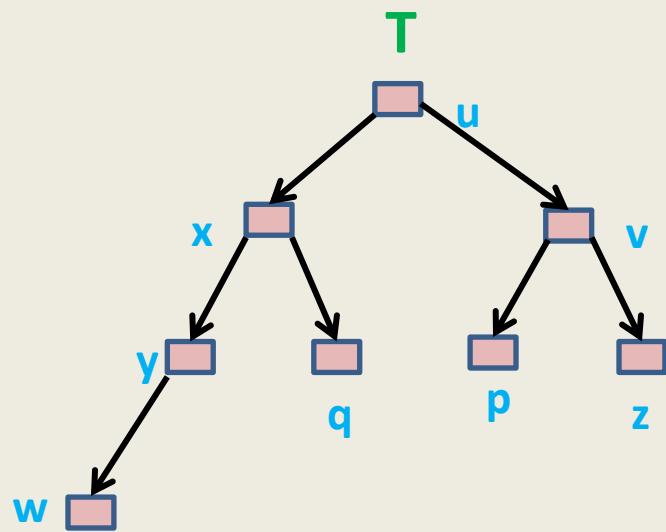
- If there is an edge from node u to node v ,
then u is called **parent** of v , and v is called **child** of u .
- The **Height** of a Binary tree T is the maximum number of edges from the root to any leaf node in the tree T .

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



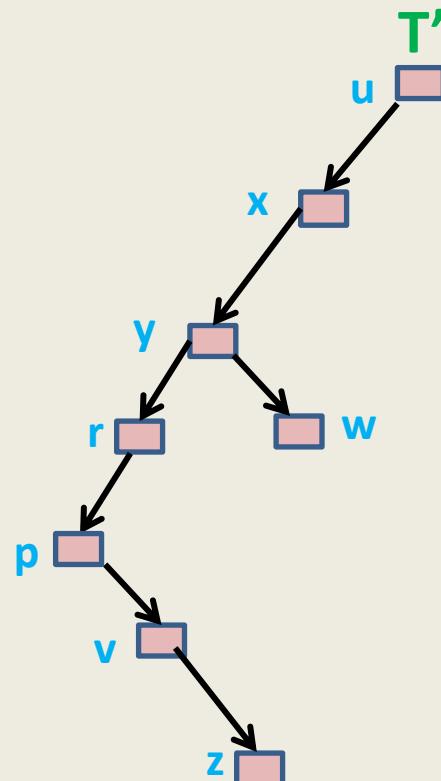
Varieties of Binary trees

We call it **Perfectly balanced**

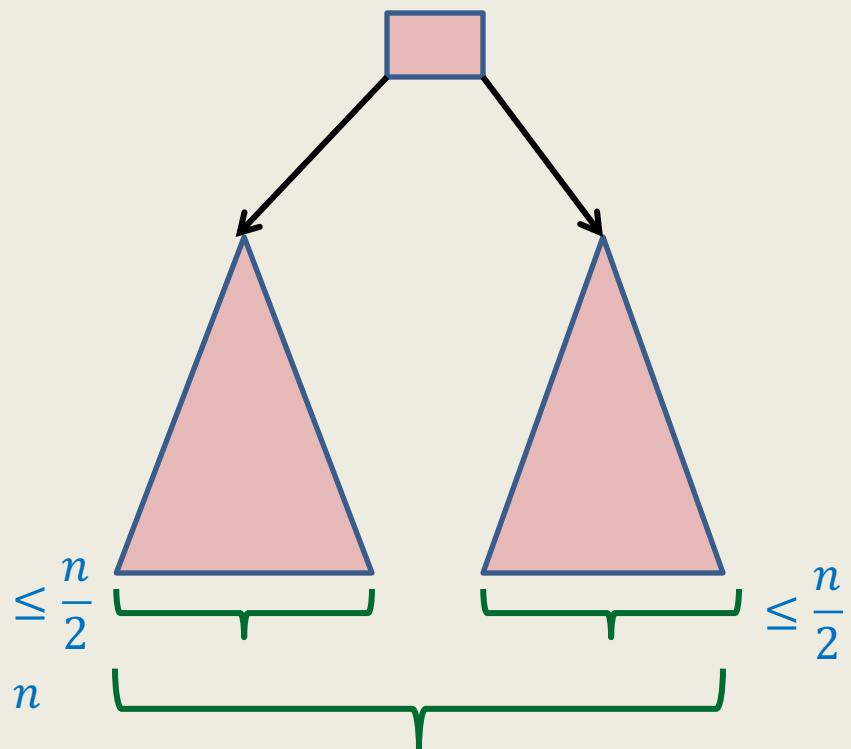


For every node, the number of nodes in the **subtrees of its two children** differ at **atmost** by 1.

skewed



Height of a perfectly balanced Binary tree

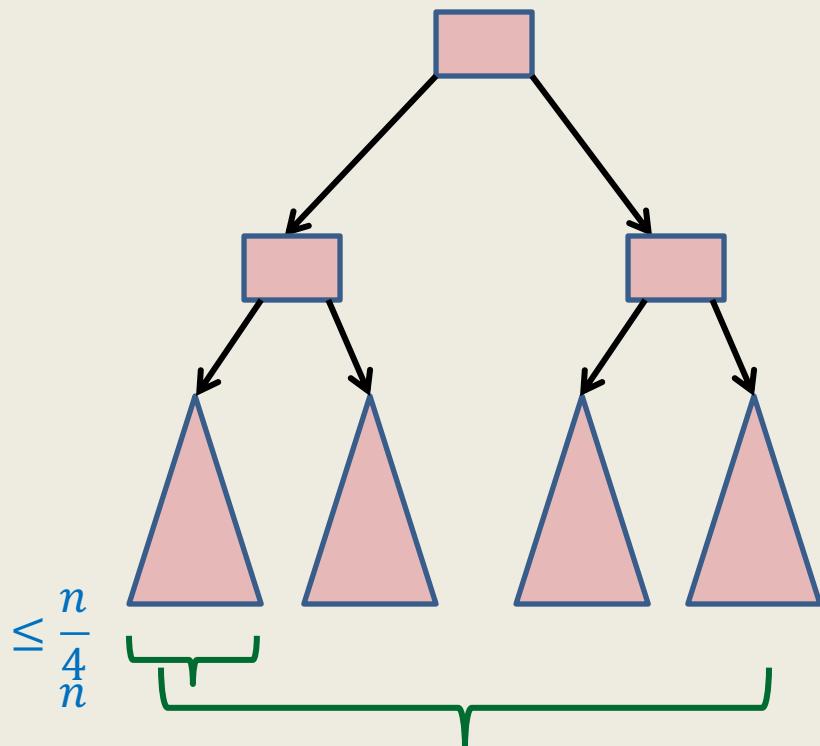


$H(n)$: Height of a perfectly balanced binary tree on n nodes.

$$H(1) = 0$$

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

Height of a perfectly balanced Binary tree

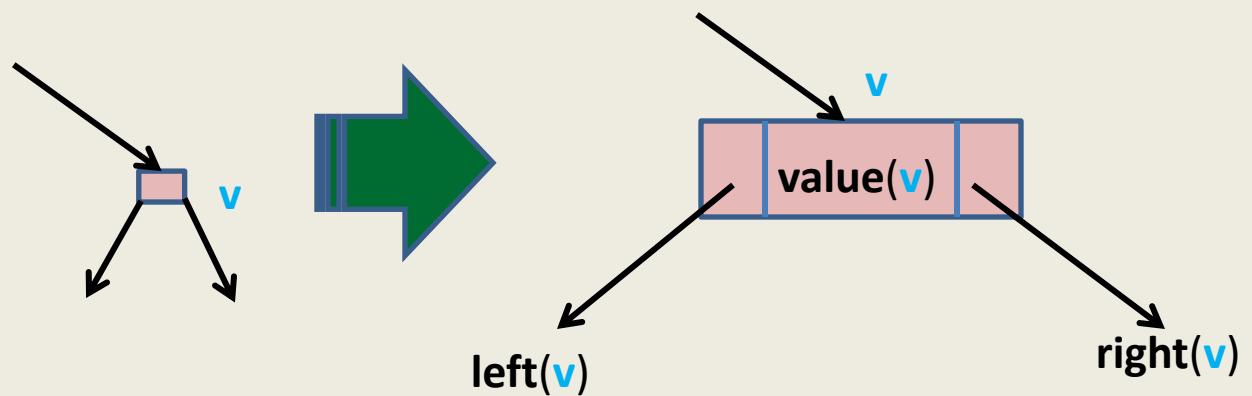


$H(n)$: Height of a perfectly balanced binary tree on n nodes.

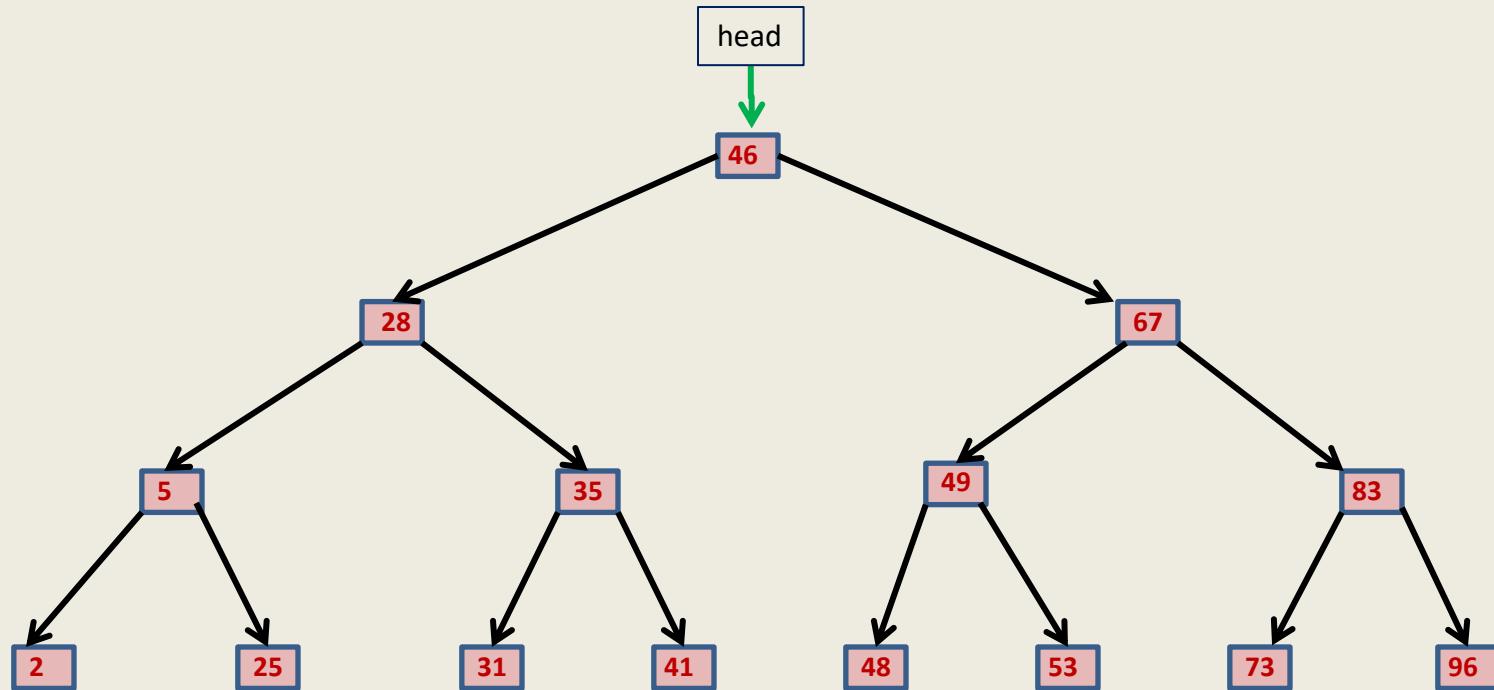
$$H(1) = 0$$

$$\begin{aligned} H(n) &\leq 1 + H\left(\frac{n}{2}\right) \\ &\leq 1 + 1 + H\left(\frac{n}{4}\right) \\ &\leq 1 + 1 + \cdots + H\left(\frac{n}{2^i}\right) \\ &\leq \log_2 n \end{aligned}$$

Implementing a Binary tree



Binary Search Tree (BST)



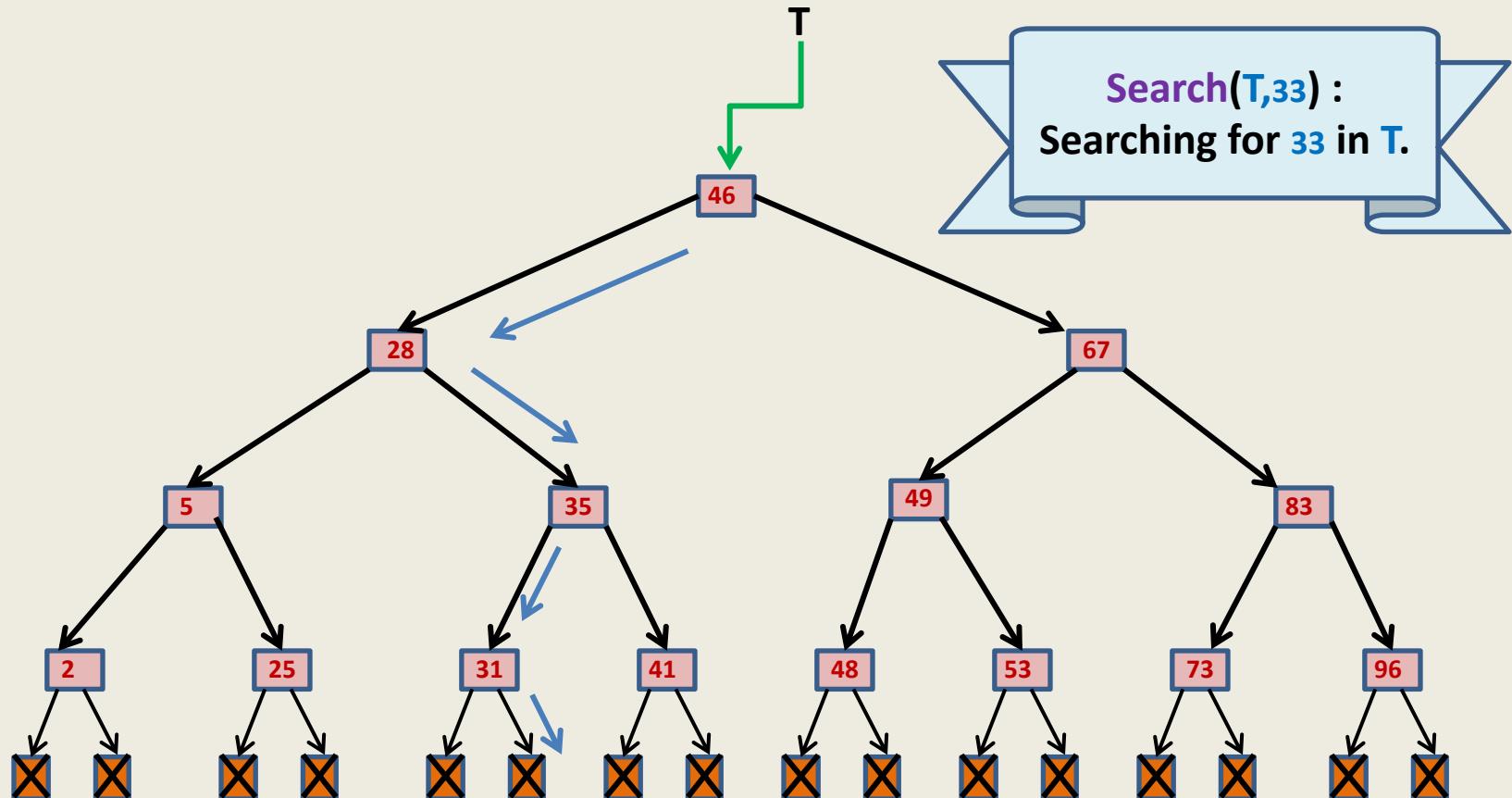
Definition: A Binary Tree T storing values is said to be **Binary Search Tree**

if for each node v in T

- If $\text{left}(v) \neq \text{NULL}$, 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 $\text{subtree}(\text{right}(v))$.

Search(T, x)

Searching in a Binary Search Tree



Search(T, x)

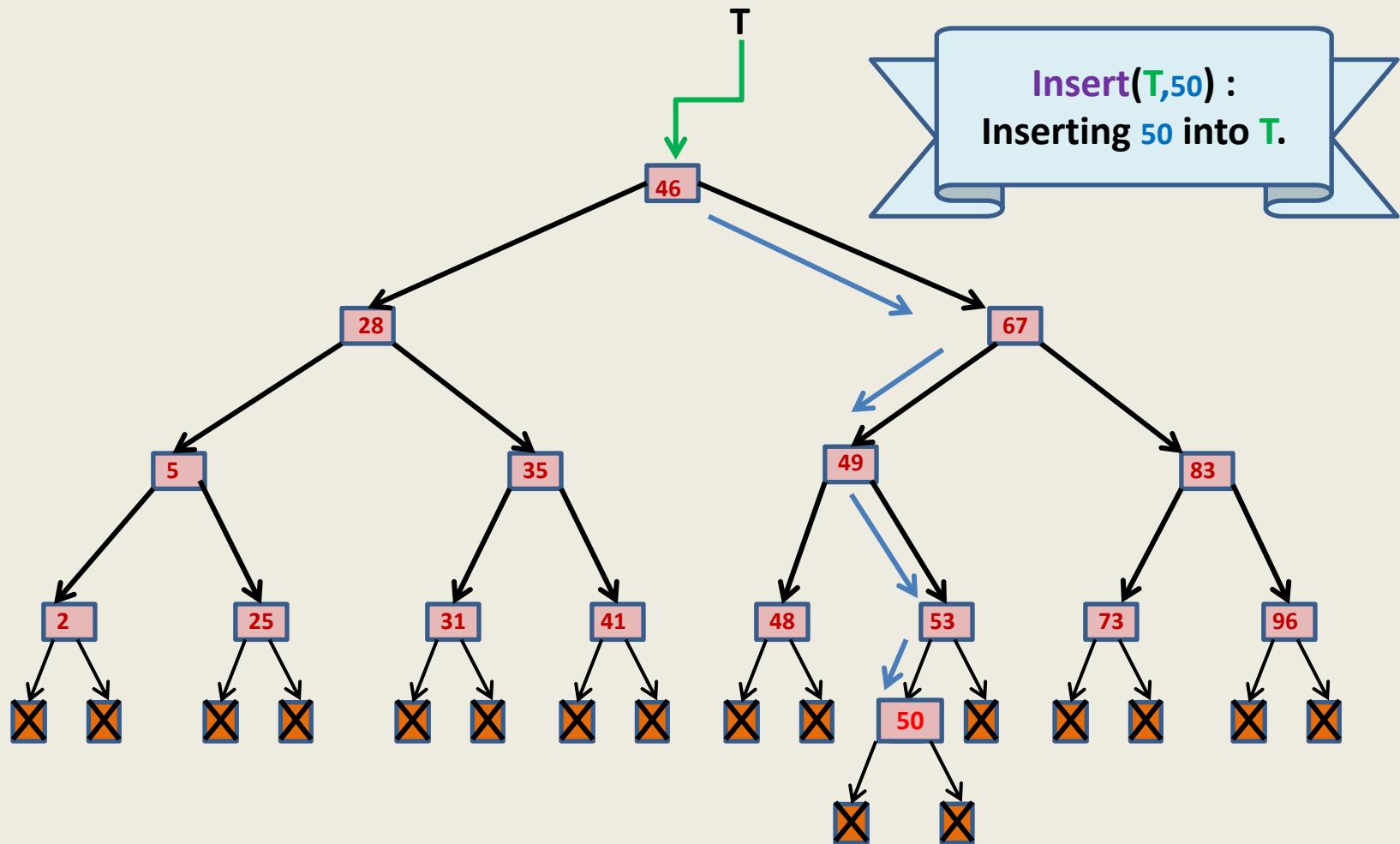
Searching in a Binary Search Tree

Search(T, x)

```
{    p  $\leftarrow T$ ;  
    Found  $\leftarrow \text{FALSE}$ ;  
    while( Found = FALSE & p  $\neq \text{NULL}$  )  
    {        if( value(p) = x ) Found  $\leftarrow \text{TRUE}$  ;  
            else if ( value(p) < x ) p  $\leftarrow \text{right}(p)$  ;  
            else p  $\leftarrow \text{left}(p)$  ;  
    }  
    return p;  
}
```

Insert(T, x)

Insertion in a Binary Search Tree



A question

Time complexity of

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

Homeworks

1. Write pseudocode for $\text{Insert}(T, x)$ operation similar to the pseudocode we wrote for $\text{Search}(T, x)$.
2. Design an algorithm for the following problem:

Given a sorted array A storing n elements,
build a “perfectly balanced” BST storing all elements of A
in $O(n)$ time.

Homework 3

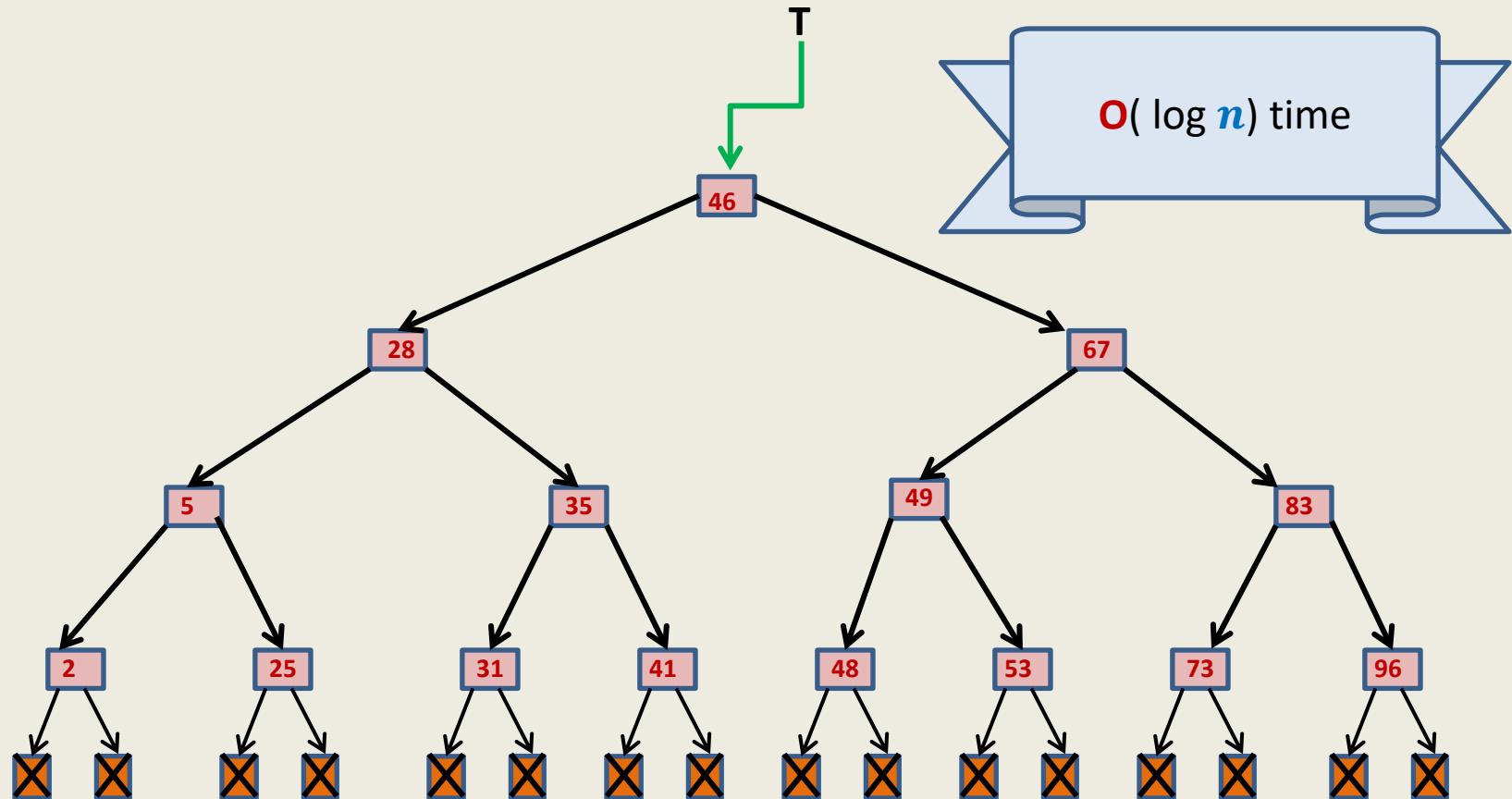
What does the following algorithm accomplish ?

Traversal(T)

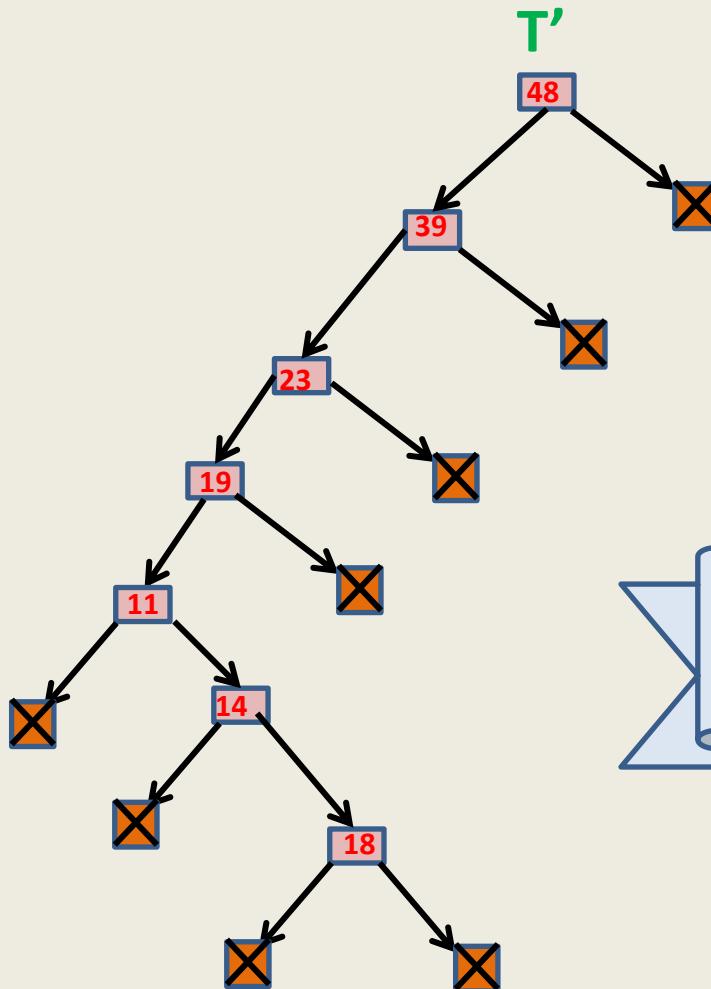
```
{    p ← T;  
    if(p=NULL) return;  
    else{    if(left(p) <> NULL)    Traversal(left(p));  
            print(value(p));  
            if(right(p) <> NULL)    Traversal(right(p));  
    }  
}
```

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

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



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



$O(n)$ time ! \ominus

Our Original Problem

Maintain a telephone directory

Operations:

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

Array based solution	Linked list based solution
$\text{Log } n$	$O(n)$
$O(n)$	$\text{Log } n$

Solution : We may keep perfectly balanced BST.

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

→ BST will be skewed 😞

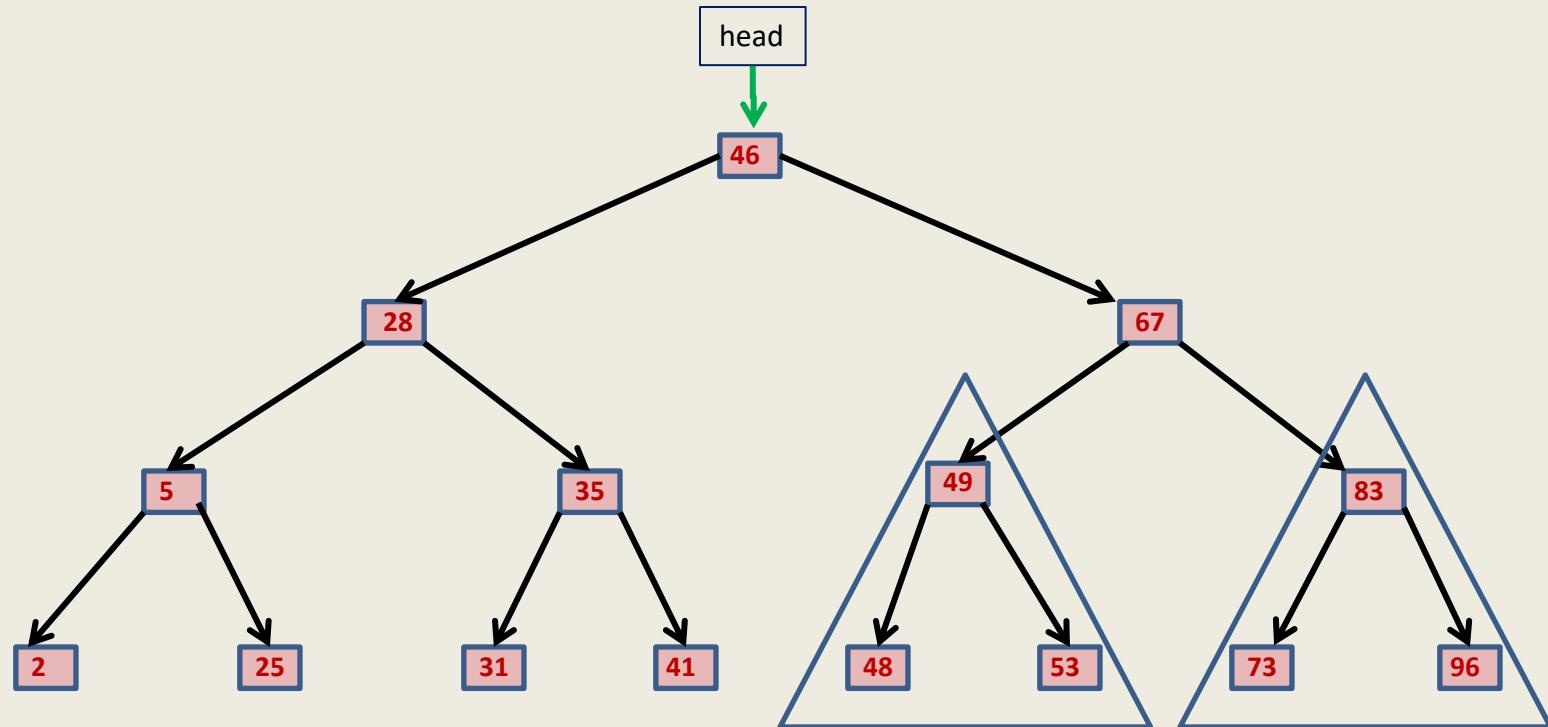
Data Structures and Algorithms

(ESO207)

Lecture 10:

- Exploring nearly balanced BST for the directory problem
- Stack: a new data structure

Binary Search Tree (BST)



Definition: A Binary Tree T storing values is said to be **Binary Search Tree**

if for each node v in T

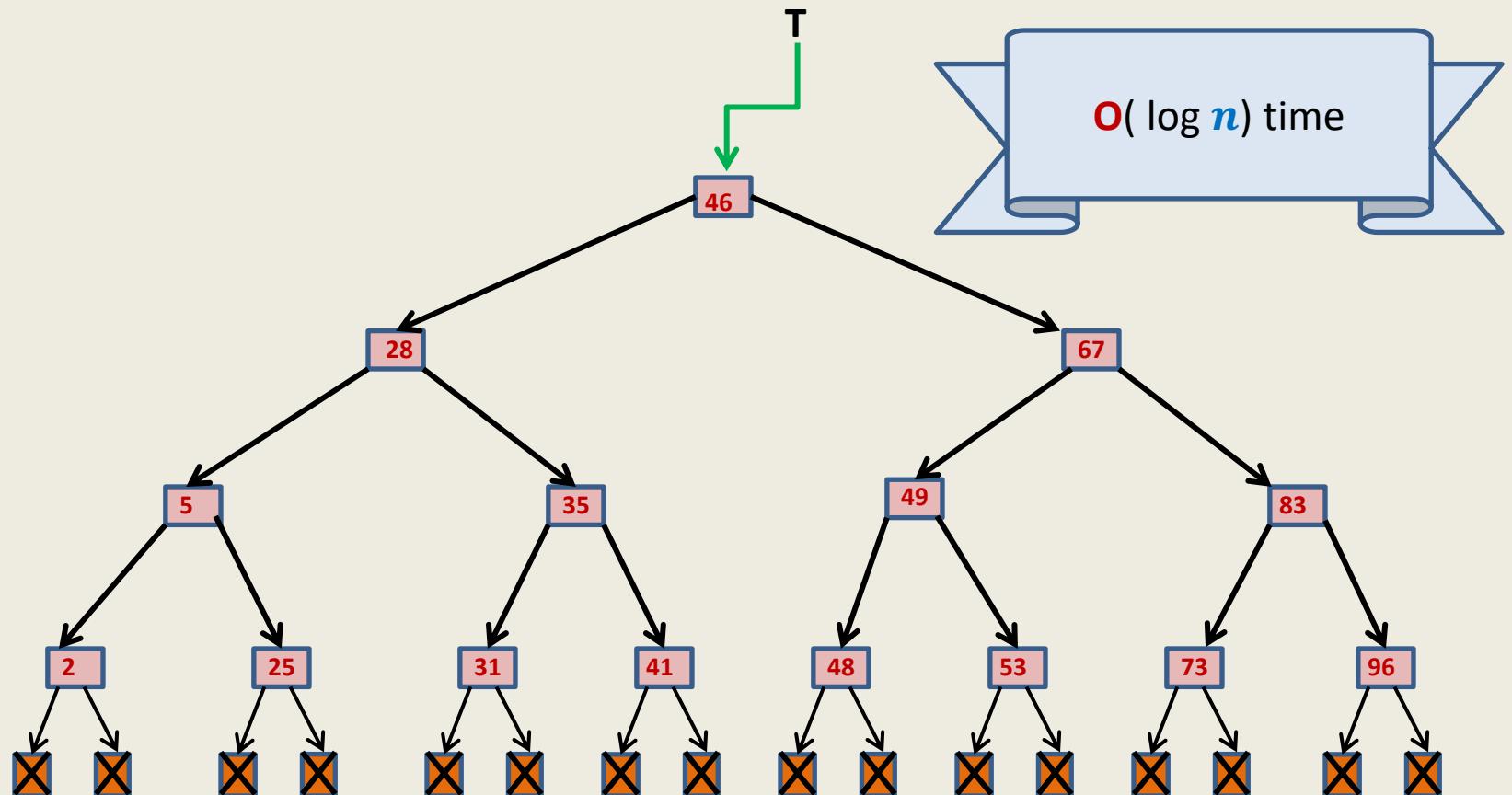
- If $\text{left}(v) \neq \text{NULL}$, then $\text{value}(v) > \text{value}$ of every node in $\text{subtree}(\text{left}(v))$.
- If $\text{right}(v) \neq \text{NULL}$, then $\text{value}(v) < \text{value}$ of every node in $\text{subtree}(\text{right}(v))$.

A question

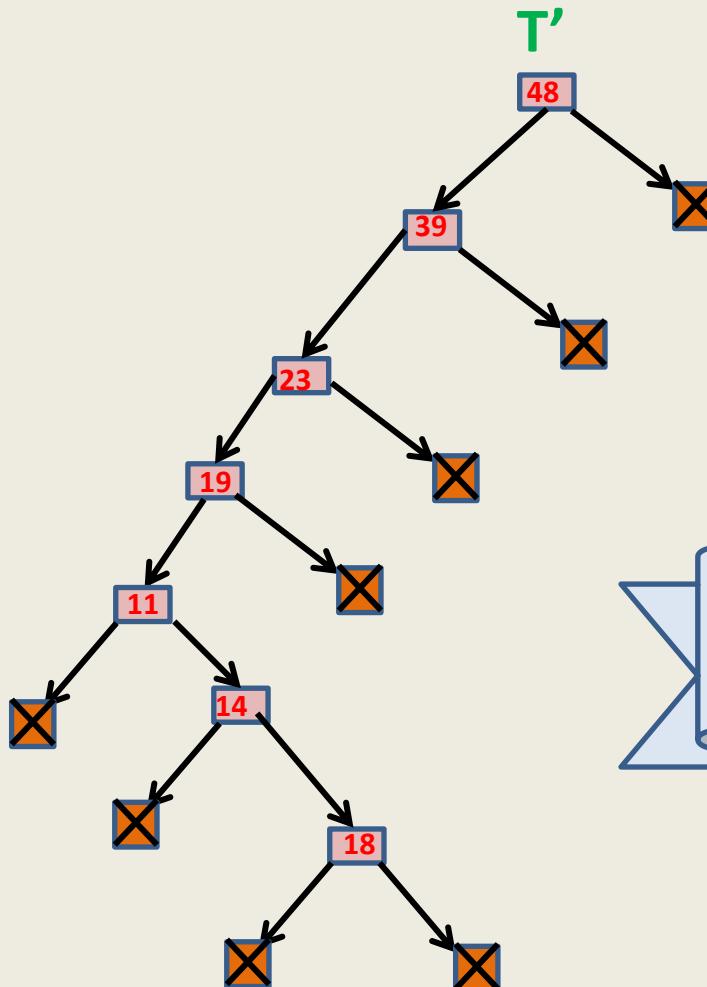
Time complexity of

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

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



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



$O(n)$ time ! \ominus

Our Original Problem

Maintain a telephone directory

Operations:

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

Array based solution	Linked list based solution
$\text{Log } n$	$O(n)$
$O(n)$	$\text{Log } n$

Solution : We may keep perfectly balanced BST.

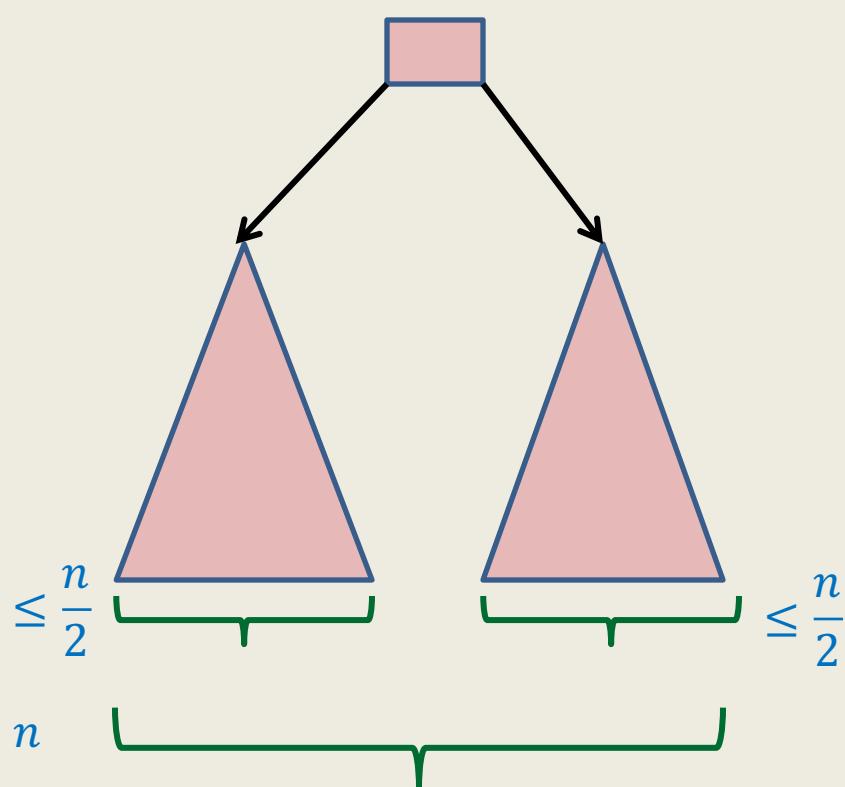
Hurdle: What if we insert records in increasing order of ID ?

→ BST will be skewed 😞

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

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

How crucial is perfect balance of a BST ?

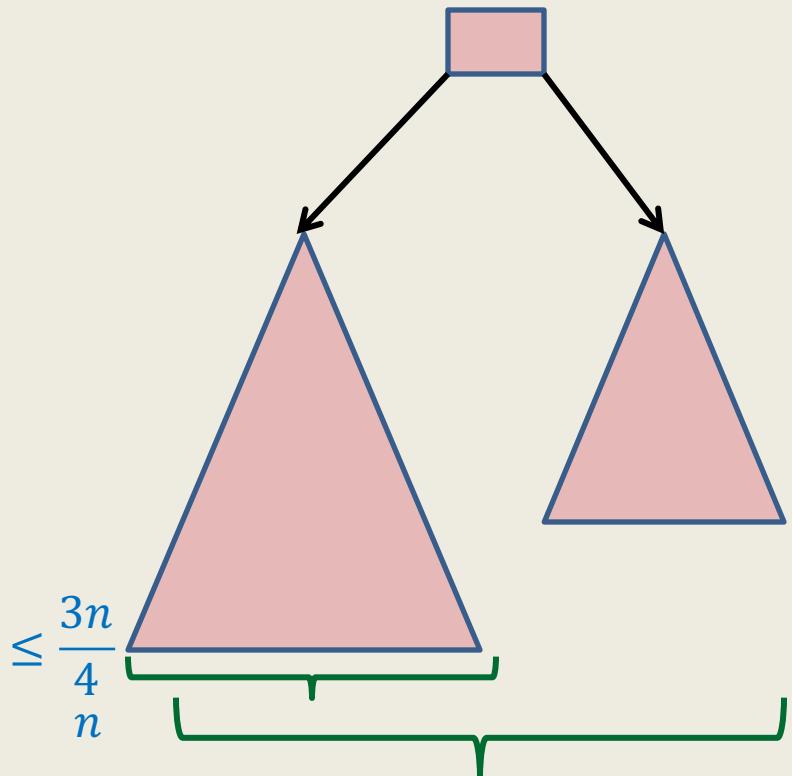


$$H(1) = 0$$

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

Let us change this recurrence slightly.

How crucial is perfect balance of a BST ?



Lesson learnt :
We may as well work with nearly balanced BST

$$H(1) = 0$$

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

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

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

$$\leq \log_{4/3} n$$

What lesson did you get
from this recurrence ?
Think for a while before
going further ...

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

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

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

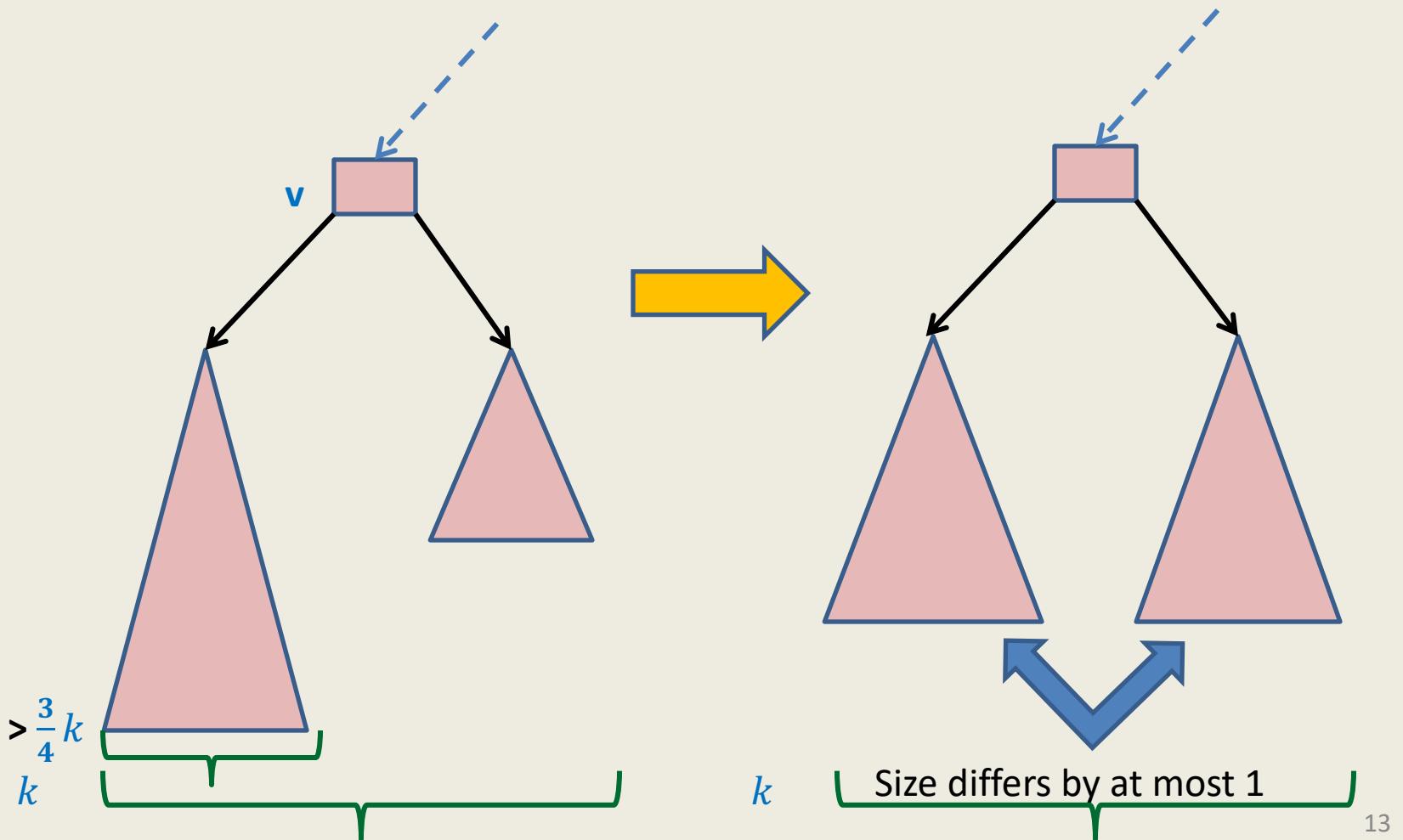
- If a node **v** is **perfectly balanced**, it requires many insertions till **v** ceases to remain **nearly balanced**.
- Any arbitrary **BST** of size **n** can be converted into a **perfectly balanced BST** in **O(n)** time.

Solving our dictionary problem

Preserving $O(\log n)$ height after each operation

- Each node v in T maintains an 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



What can we say about this data structure ?

It is elegant and reasonably simple to implement.

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

But the number of such operations will be rare.

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

Indeed, this data structure achieve the following goals:

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

How can we justify these claims ?

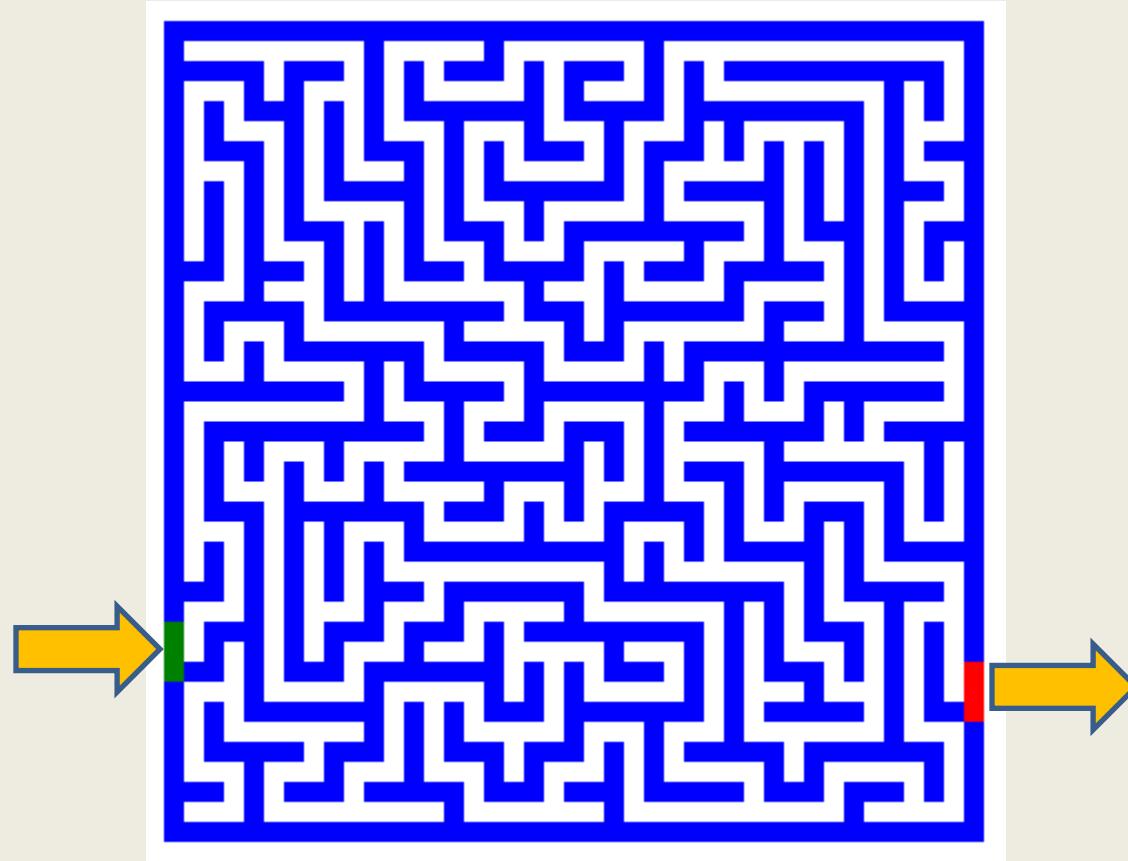
Keep thinking till we do it in a few weeks 😊.

Stack: a data structure

A few **motivating examples**

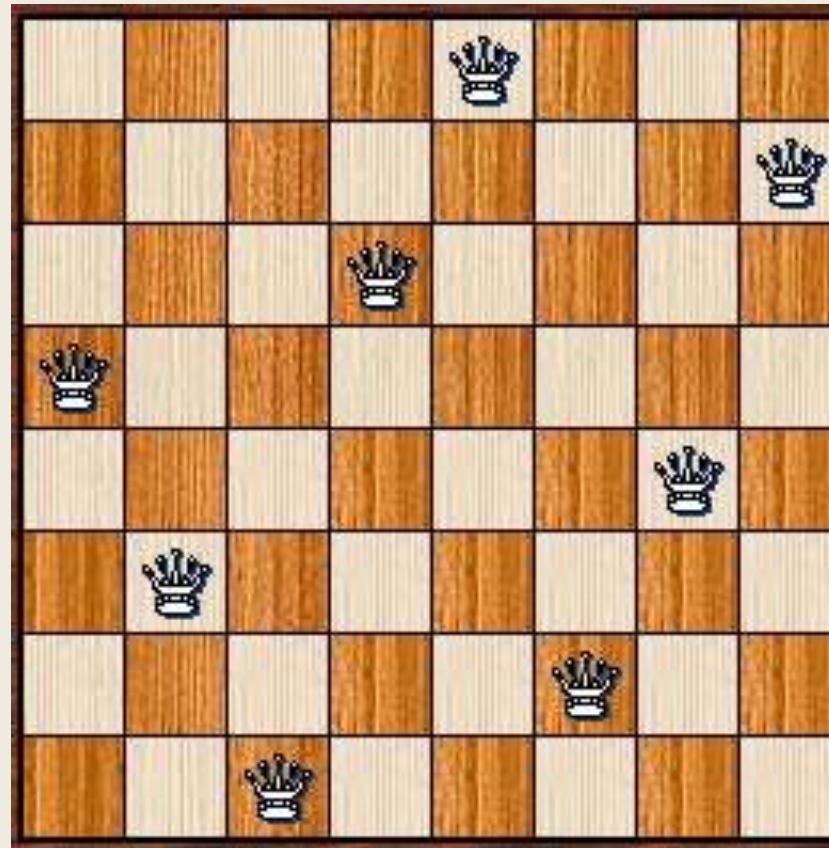
Finding path in a maze

Problem : How to design an algorithm for finding a path in a maze ?



8-Queens Problem

Problem: How to place **8 queens on a chess board**
so that no two of them attack each other ?



Expression Evaluation

- $x = 3 + 4 * (5 - 6 * (8 + 9^2) + 3)$

Problem:

Can you write a program to evaluate any arithmetic expression ?

Stack: a data structure

Stack

Data Structure Stack:

- Mathematical Modeling of Stack
- Implementation of Stack

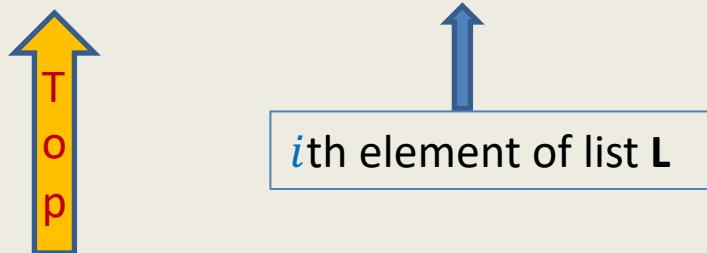
will be left as an exercise

Revisiting List

List is modeled as a sequence of elements.

we can **insert/delete/query** element at any arbitrary position in the list.

$L : a_1, a_2, \dots, a_{i-1}, a_i, a_{i+1}, \dots, a_n$

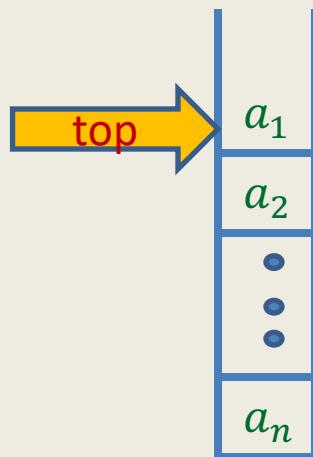


What if we **restrict** all these operations to take place only at one end of the list ?

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**.



Operations on a Stack

Query Operations

- **IsEmpty(S)**: determine if S is an empty stack
- **Top(S)**: returns the element at the top of the stack

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

Update Operations

- **CreateEmptyStack(S)**: Create an empty stack
- **Push(x,S)**: push x at the top of the stack S

Example: If S is a_1, a_2, \dots, a_n , then after **Push(x,S)**, stack S becomes

x, a_1, a_2, \dots, a_n

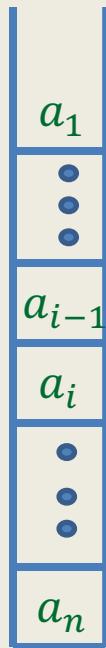
- **Pop(S)**: Delete element from top of the stack S

Example: If S is a_1, a_2, \dots, a_n , then after **Pop(S)**, stack S becomes

a_2, \dots, a_n

An Important point about stack

How to access i th element from the top ?



- To access i th element, we must pop (hence delete) **one by one** the top $i - 1$ elements from the stack.

A puzzling question/confusion

- Why do we restrict the functionality of a list ?
- What will be the use of such restriction ?

How to evaluate an arithmetic expression

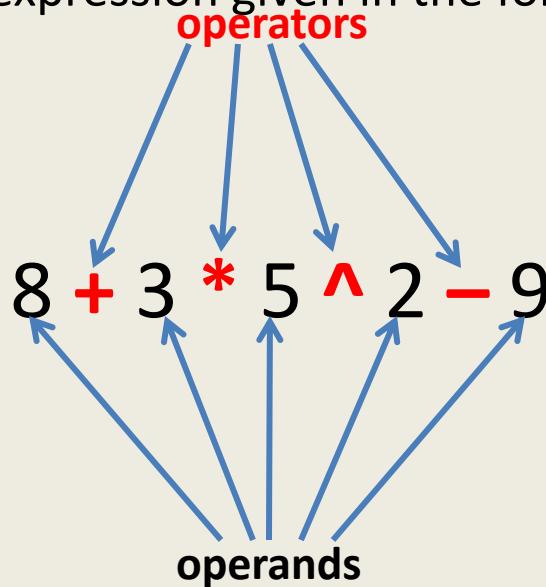
Evaluation of an arithmetic expression

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 ?



First it splits the string into **tokens** which are operators or operands (numbers). This is not difficult. But how does it evaluate it finally ???

Precedence of operators

Precedence: “priority” among different operators

- Operator $+$ has same precedence as $-$.
- Operator $*$ (as well as $/$) has higher precedence than $+$.
- Operator $*$ has same precedence as $/$.
- Operator \wedge has higher precedence than $*$ and $/$.

Associativity of operators

What is 2^3^2 ?

What is $3-4-2$?

What is $4/2/2$?

Associativity:

“How to group operators of same type ?”

$A \bullet B \bullet C = ??$

$$(A \bullet B) \bullet C \quad \text{or} \quad A \bullet (B \bullet C)$$



Left associative



Right associative

A trivial way to evaluate an arithmetic expression

8 + 3 * 5² - 9

- First perform all \wedge operations.
- Then perform all $*$ and $/$ operations.
- Then perform all $+$ and $-$ operations.

Disadvantages:

1. An ugly and case analysis based algorithm
2. Multiple scans of the expression (one for each operator).
3. What about expressions involving parentheses: $3+4*(5-6/(8+9^2)+33)$
4. What about associativity of the operators:
 - $2^3^2 = 512$ and not 64
 - $16/4/2 = 2$ and not 8.

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**

Step 1

Focusing on a simpler version of the problem

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

$$8 + 3 * 5 \wedge 2 - 9 * 67$$

Position of an operator does matter

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

Answer: if

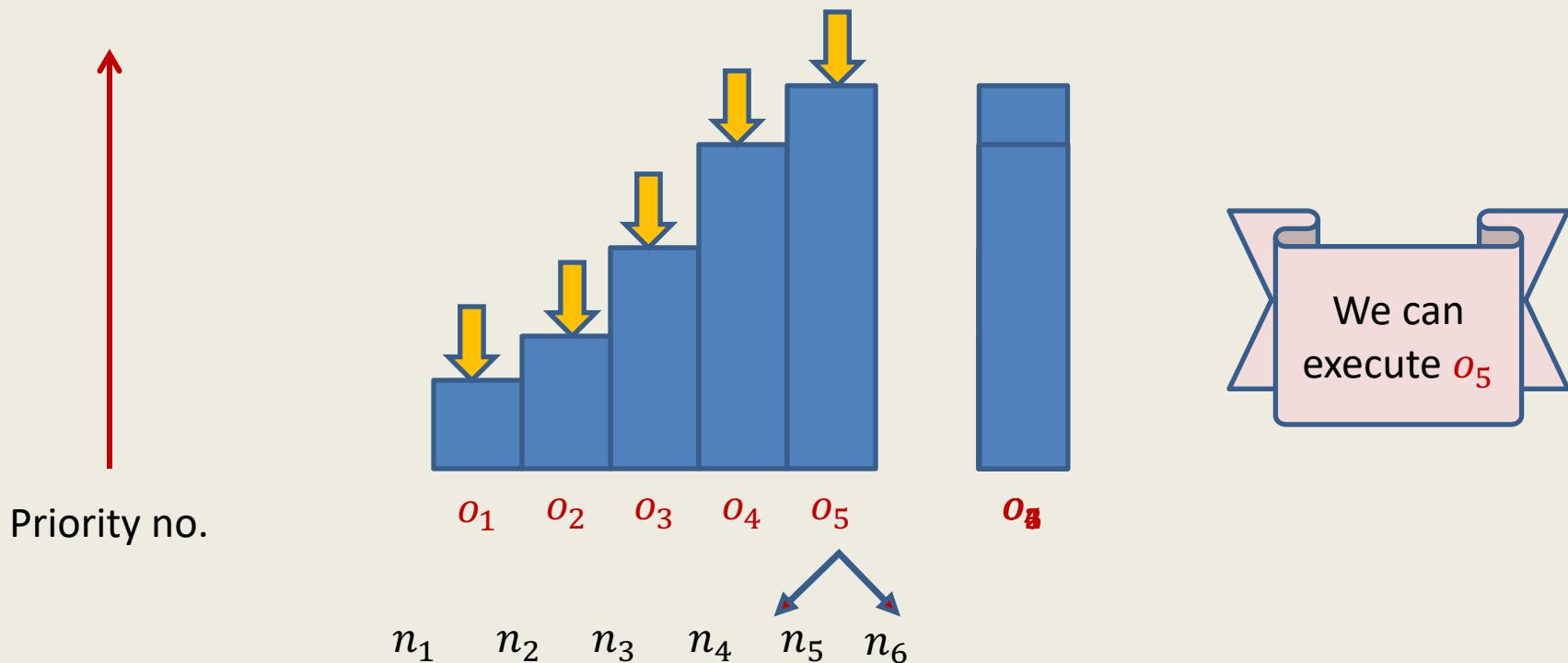
- $\text{priority}(o_i) > \text{priority}(o_{i-1})$
- $\text{priority}(o_i) \geq \text{priority}(o_{i+1})$

Give reasons for \geq
instead of $>$

Question:

How to evaluate expression in a single scan ?

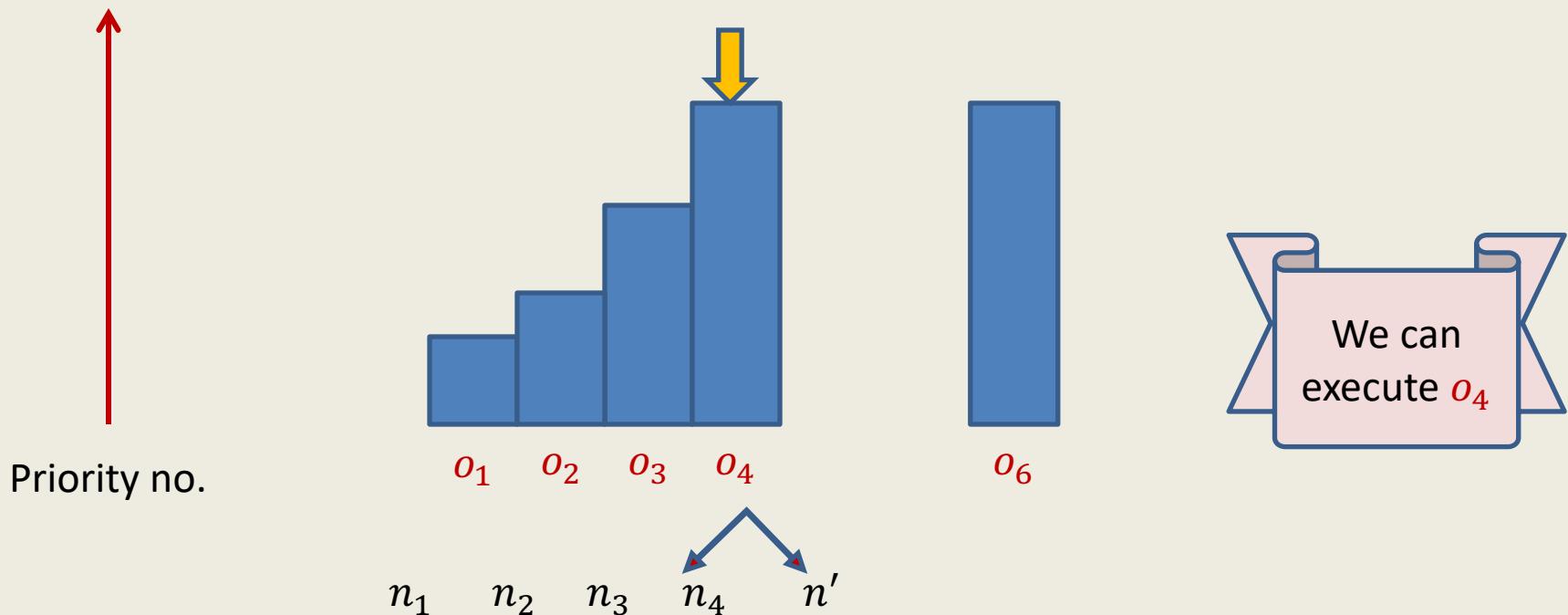
Expression: $n_1 o_1 n_2 o_2 n_3 o_3 n_4 o_4 n_5 o_5 n_6 o_6 \dots$



Question:

How to evaluate expression in a **single scan** ?

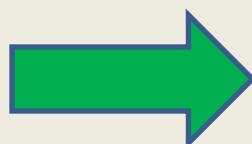
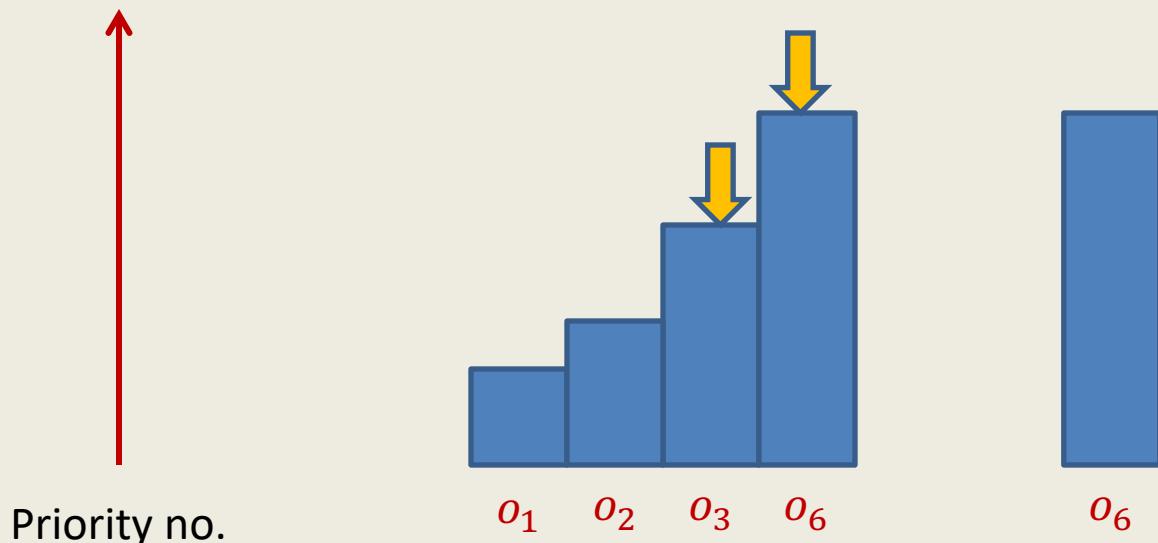
Expression: $n_1 o_1 n_2 o_2 n_3 o_3 n_4 o_4 n_5 o_5 n_6 o_6 \dots$



Question:

How to evaluate expression in a **single scan** ?

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



Homework:

Spend sometime to design an algorithm for evaluation of arithmetic expression based on the insight we developed in the last slides.

(*hint:* use 2 stacks.)

Data Structures and Algorithms

(ESO207)

Lecture 11:

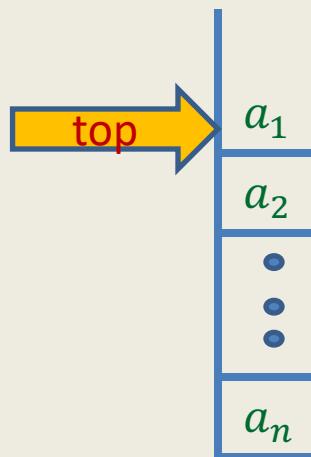
- **Arithmetic expression evaluation: Complete algorithm using stack**
- **Two interesting problems**

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**.



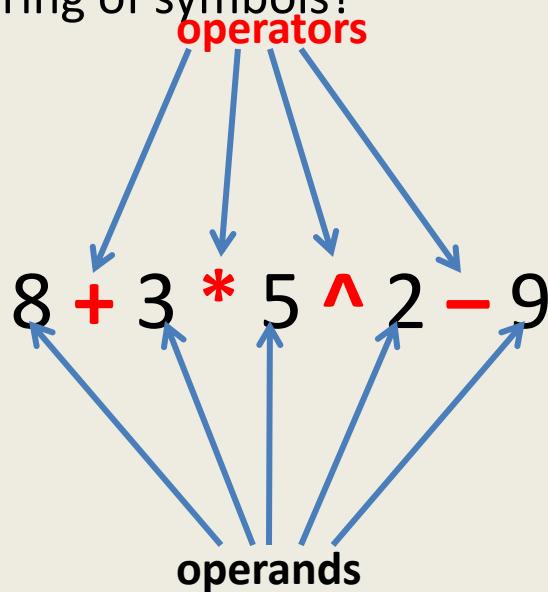
Evaluation of an arithmetic expression

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**

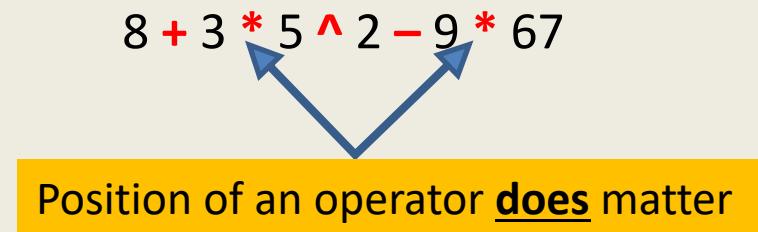
Incorporating precedence of operators through priority number

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



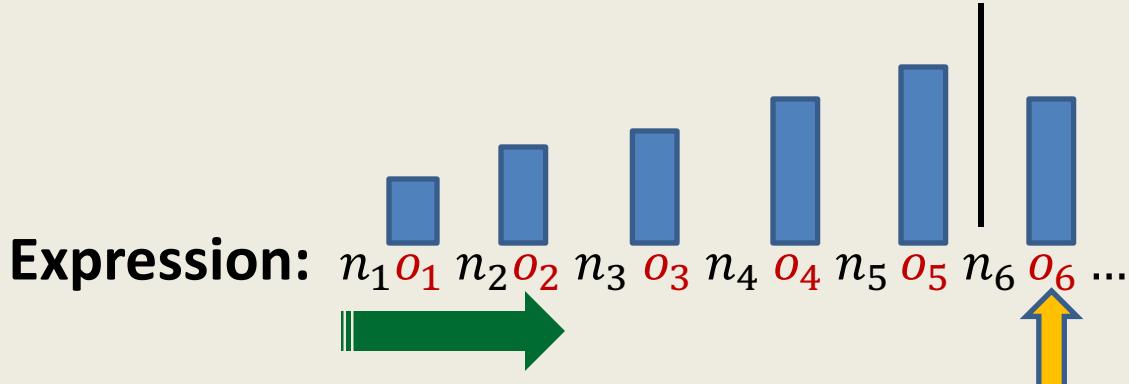
We keep two stacks:



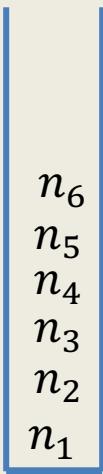
N-stack
for operands



O-stack
for operators



We keep two stacks:



N-stack
for operands

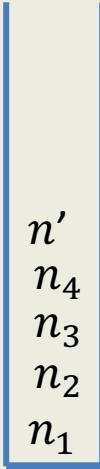


O-stack
for operators

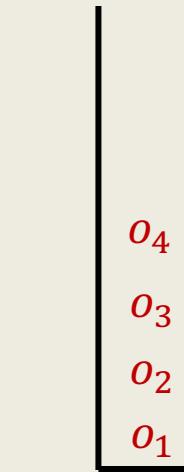
Expression: $n_1 o_1 n_2 o_2 n_3 o_3 n_4 o_4 \dots$



We keep two stacks:

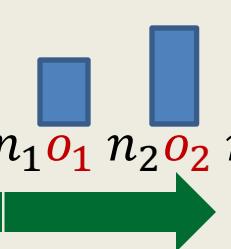


N-stack
for **operands**



O-stack
for **operators**

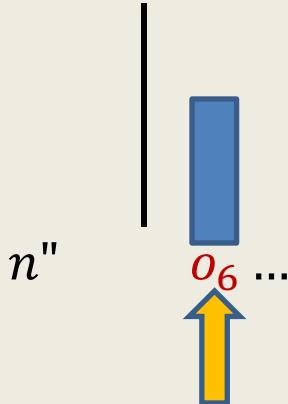
Expression: $n_1 o_1 n_2 o_2 n_3 o_3$



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 ?

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

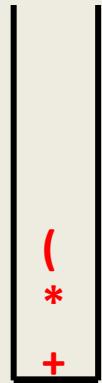
What should we do whenever
we encounter (in the expression ?

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 ?

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

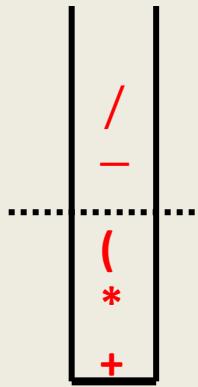
What needs to be done when
(is at the top of the O-stack ?

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

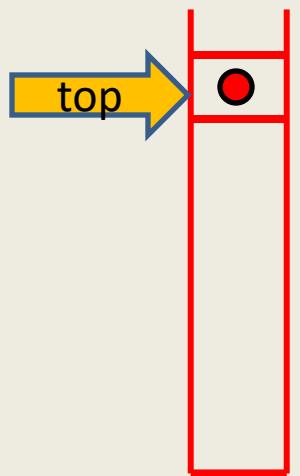
Take a pause for a few minutes to realize **surprisingly that
the **contradicting** requirements for the priority of (**
in fact hints at a **suitable solution for handling (.**

How to handle parentheses ?

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

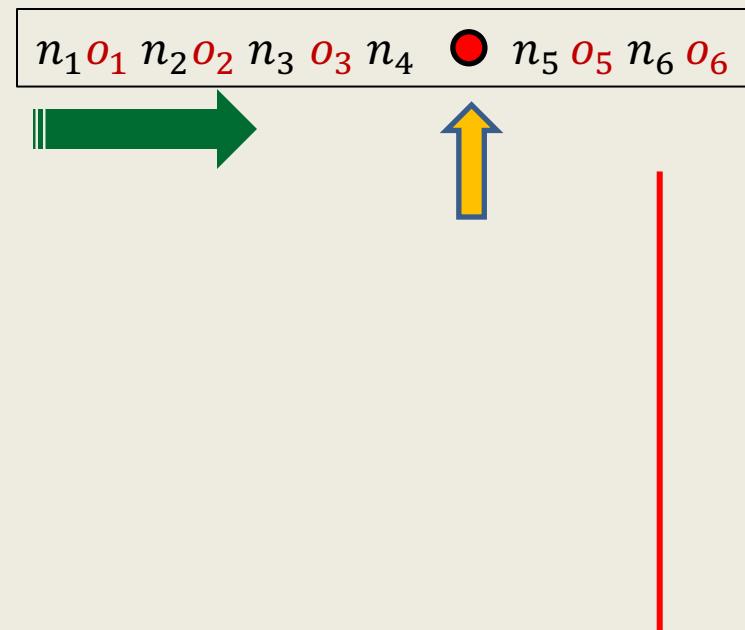
InsideStack priority

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



OutsideStack priority

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



O-stack

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);
    ...
    ...
}
```

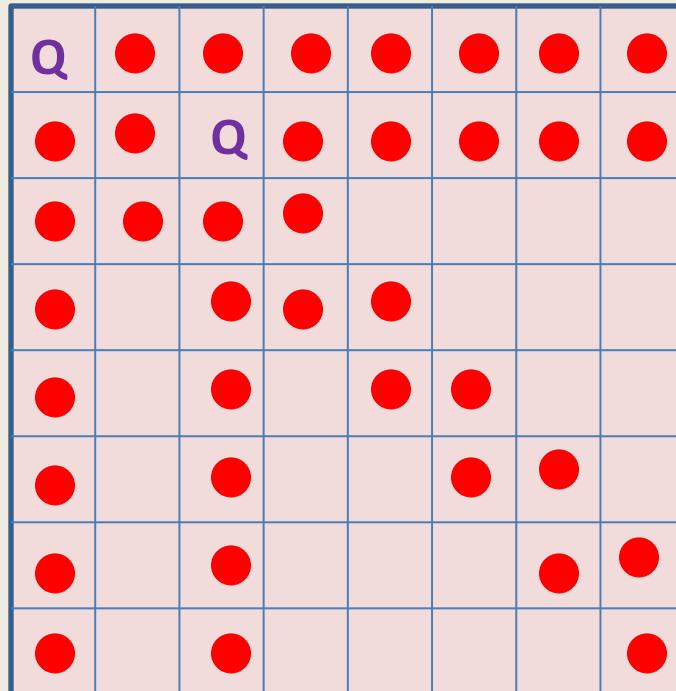
Learn about it from [wikipedia](#) ...

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.

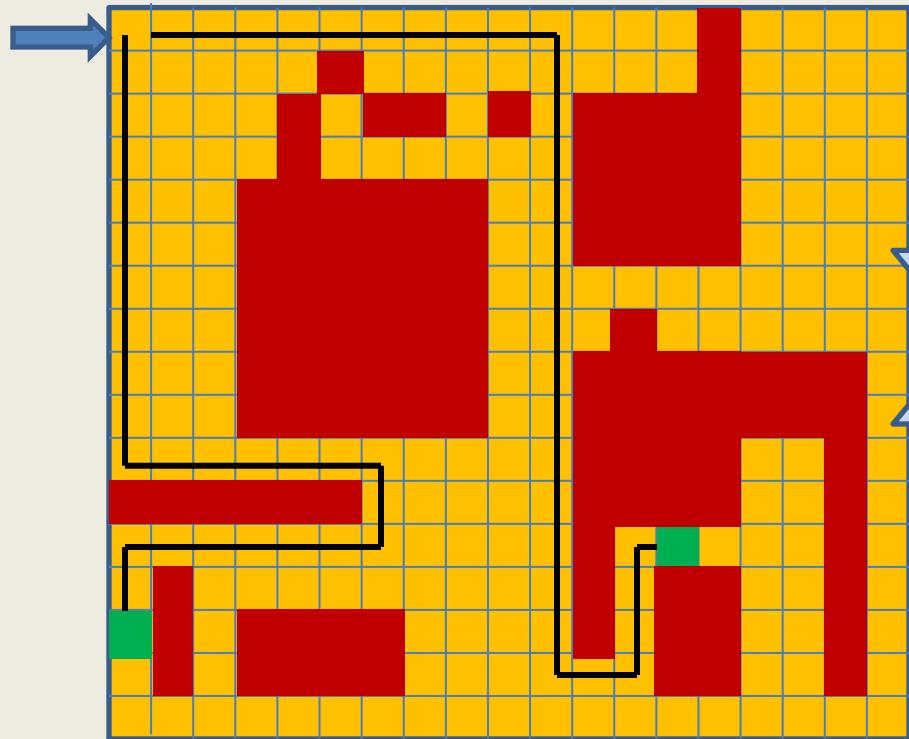


With this sketch/hint,
try to design the
complete algorithm
using stack or
otherwise.

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

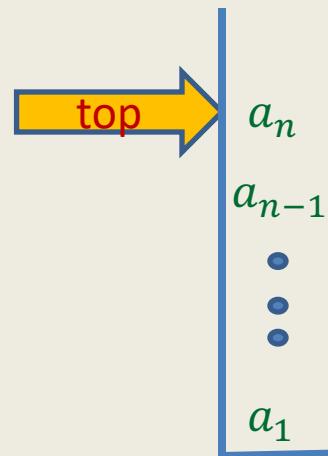
Queue: a new data structure

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:

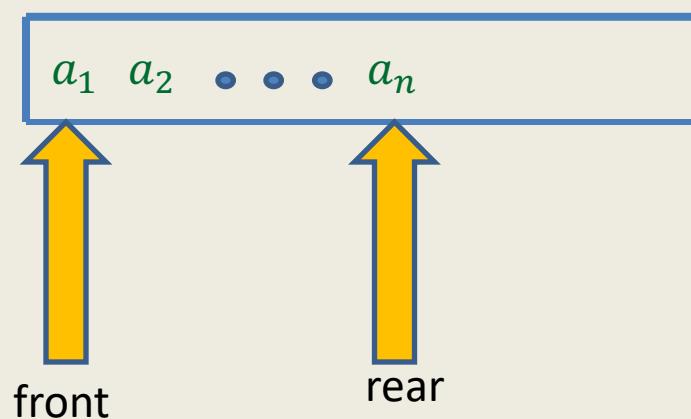
Last in (LIFO)

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`.

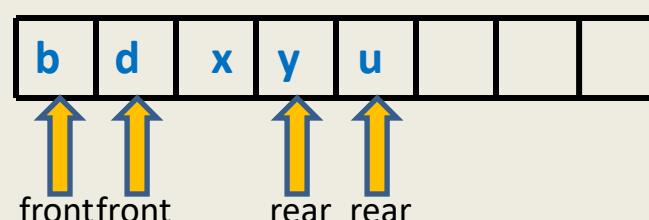
- `front`: the position of the **first** element of the queue in the array.
- `rear`: the position of the **last** element of the queue in the array.

Enqueue(x, Q)

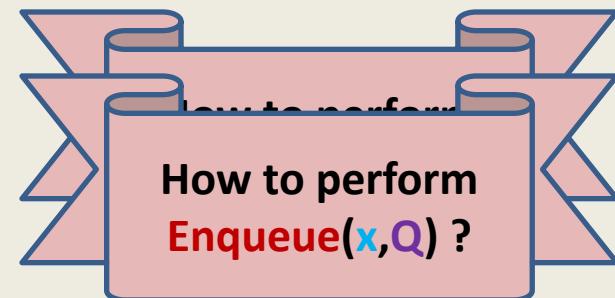
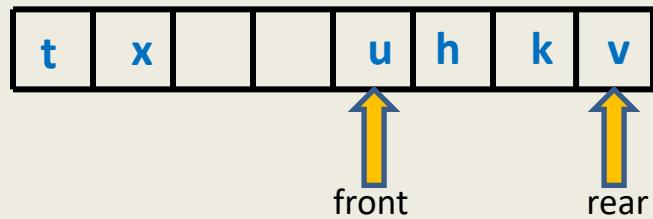
```
{    rear ← rear+1;  
    Q[rear]←x  
}
```

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

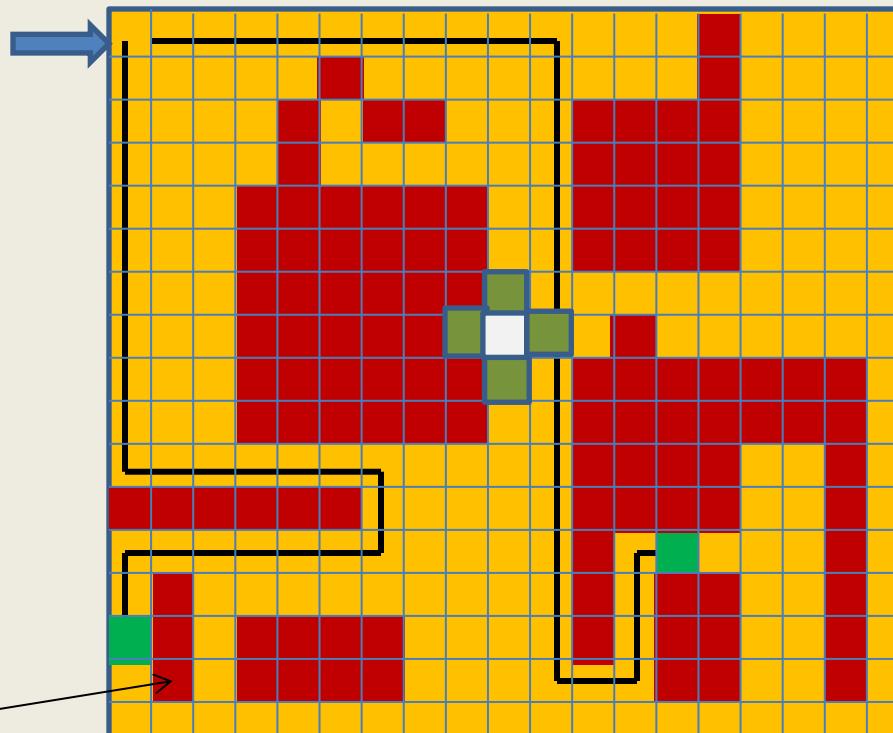
Shortest route in a grid

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.

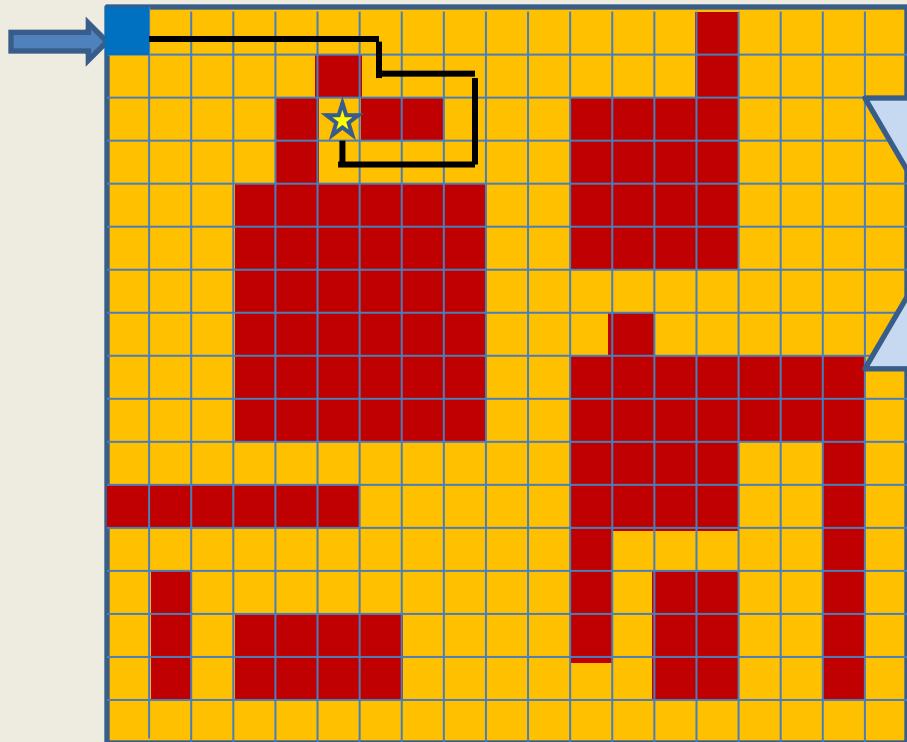


Step 1:

**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 😊

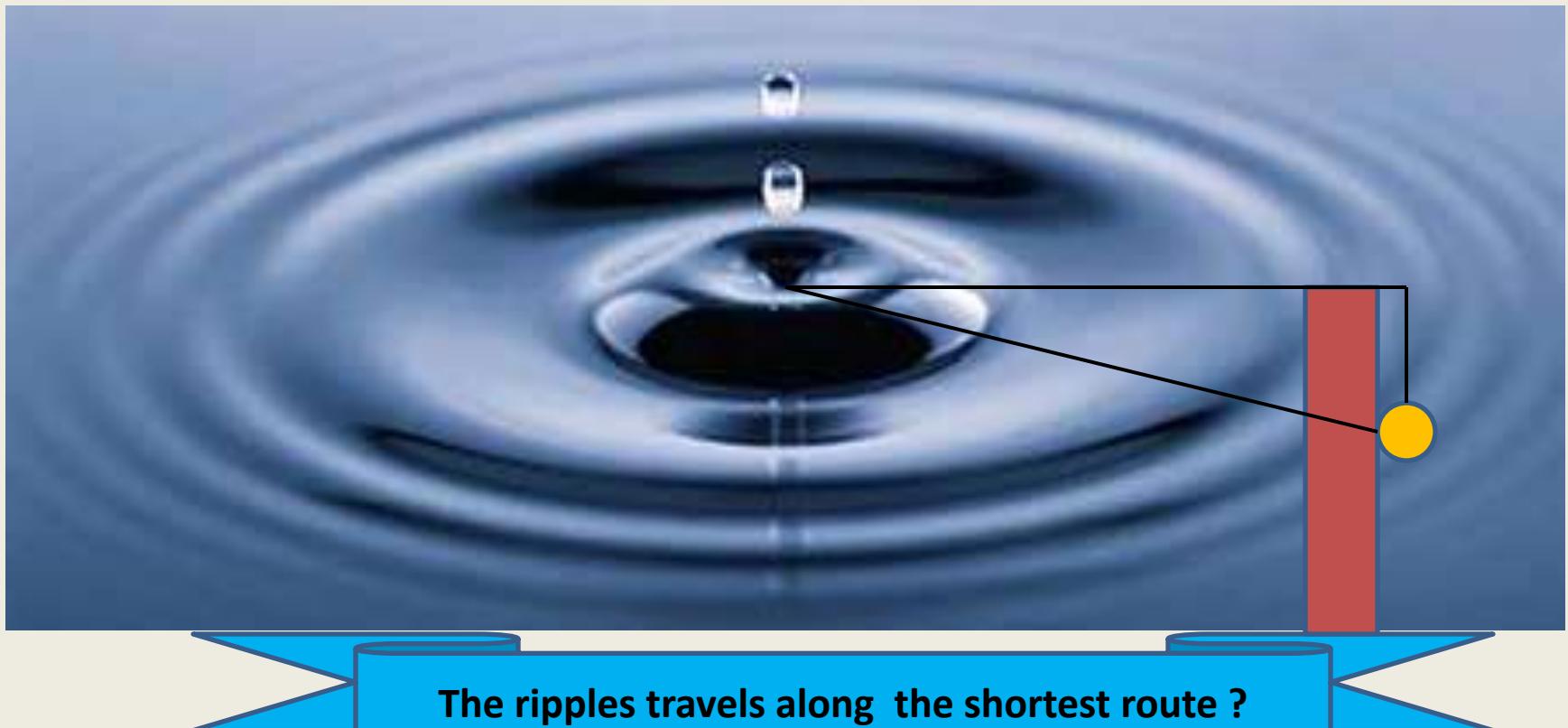
Definition: Distance of a cell c from another cell c'

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

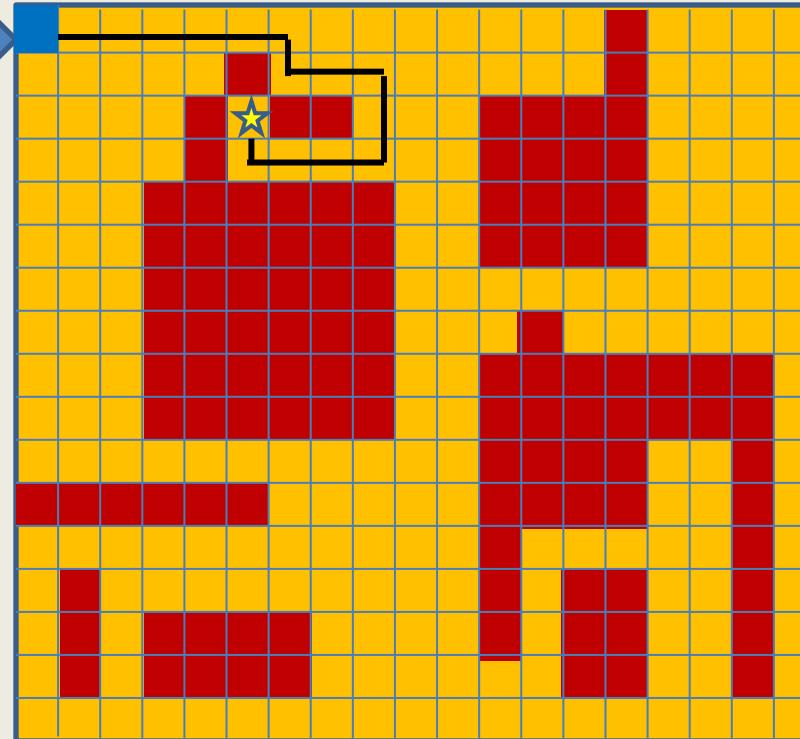


The ripples travels along the shortest route ?

Shortest route in a grid

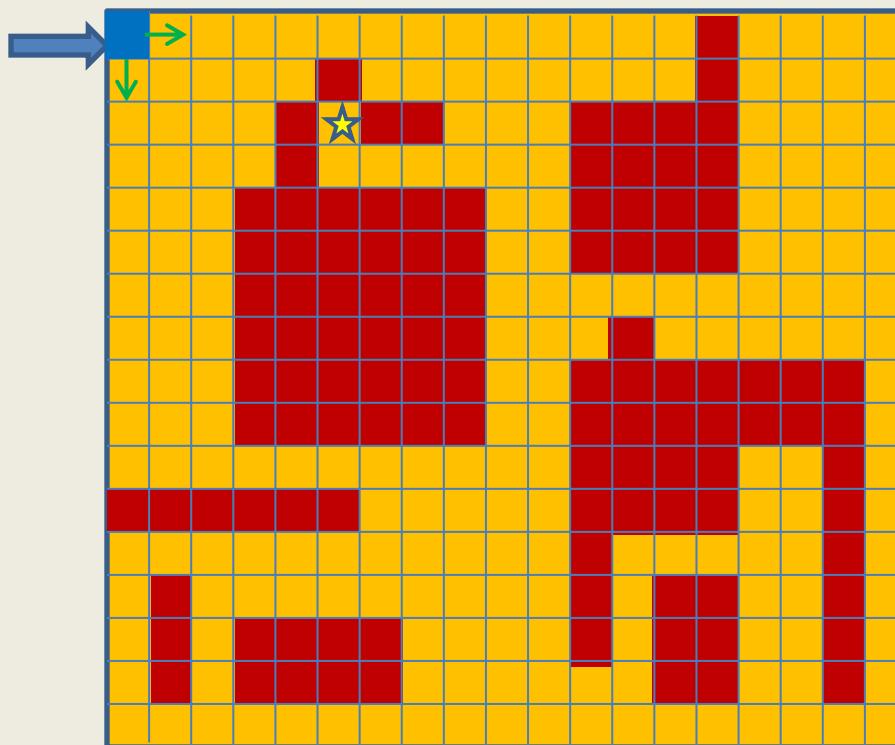
nontriviality of the problem

How to find the shortest route to  in the grid ?

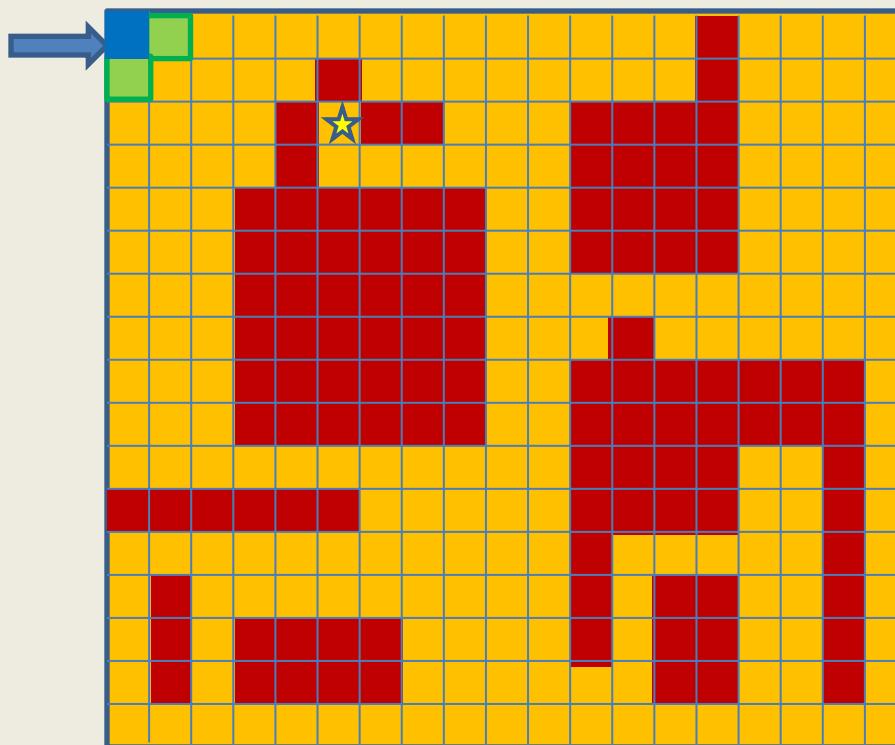


Create a ripple at the start cell and trace
the path it takes to 

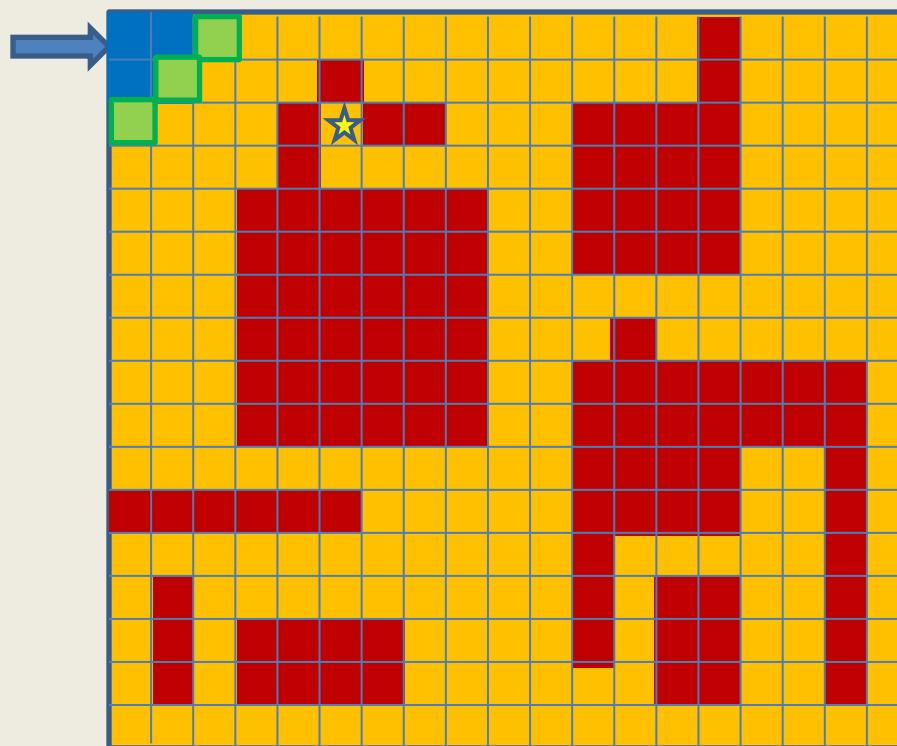
propagation of a ripple from **the start cell**



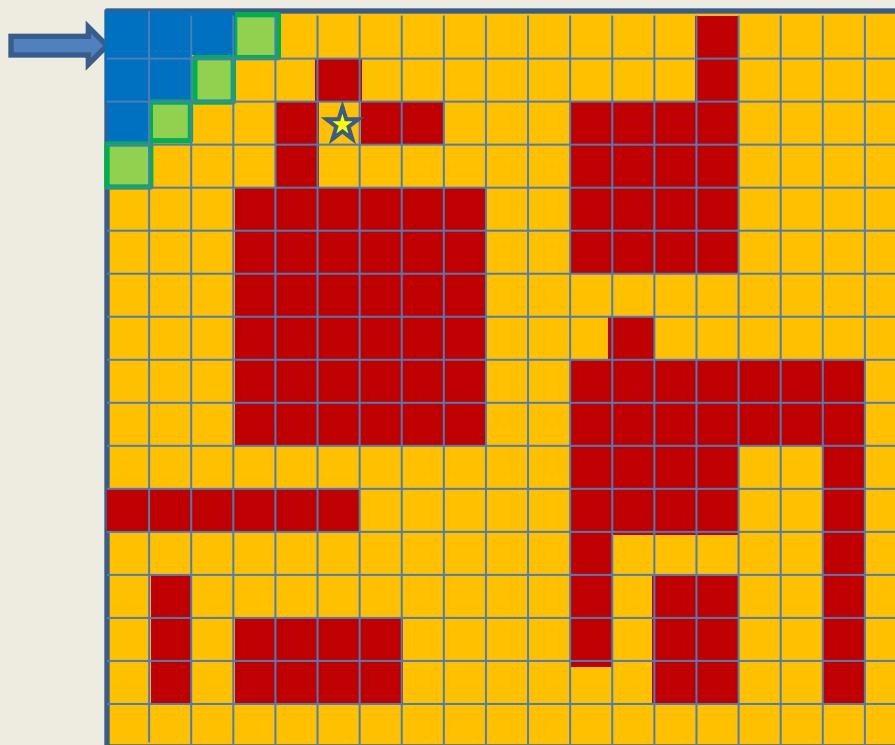
ripple reaches cells at distance 1 in step 1



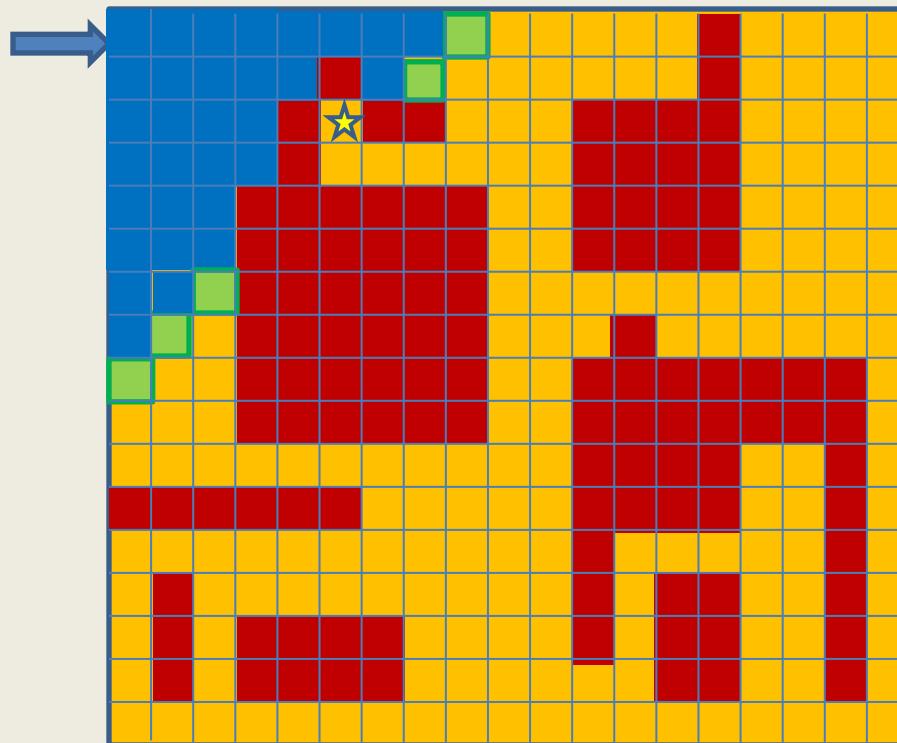
ripple reaches cells at distance 2 in step 2



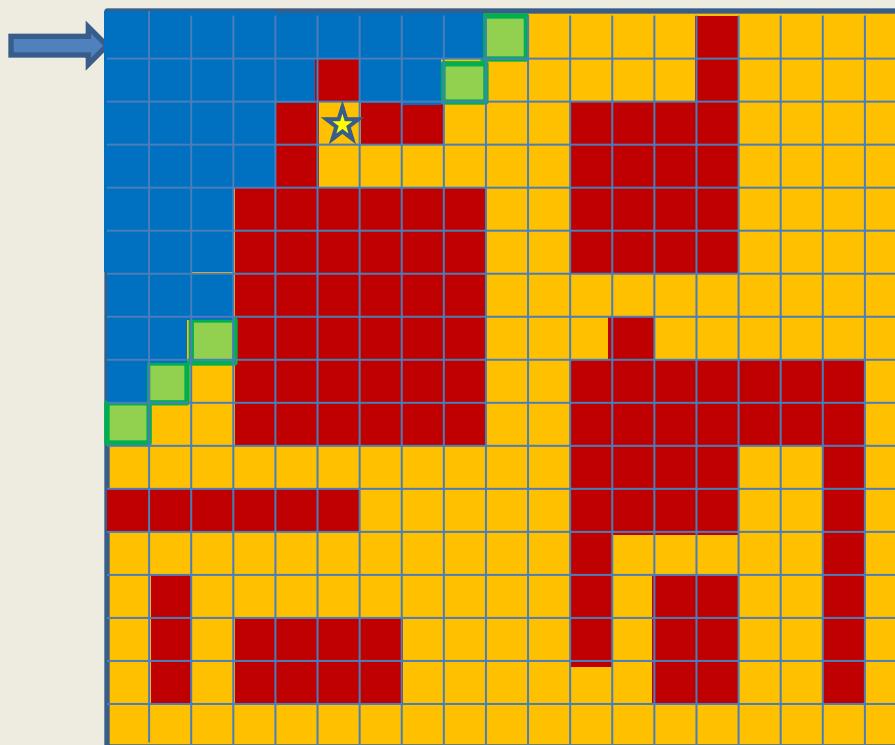
ripple reaches cells at distance 3 in step 3



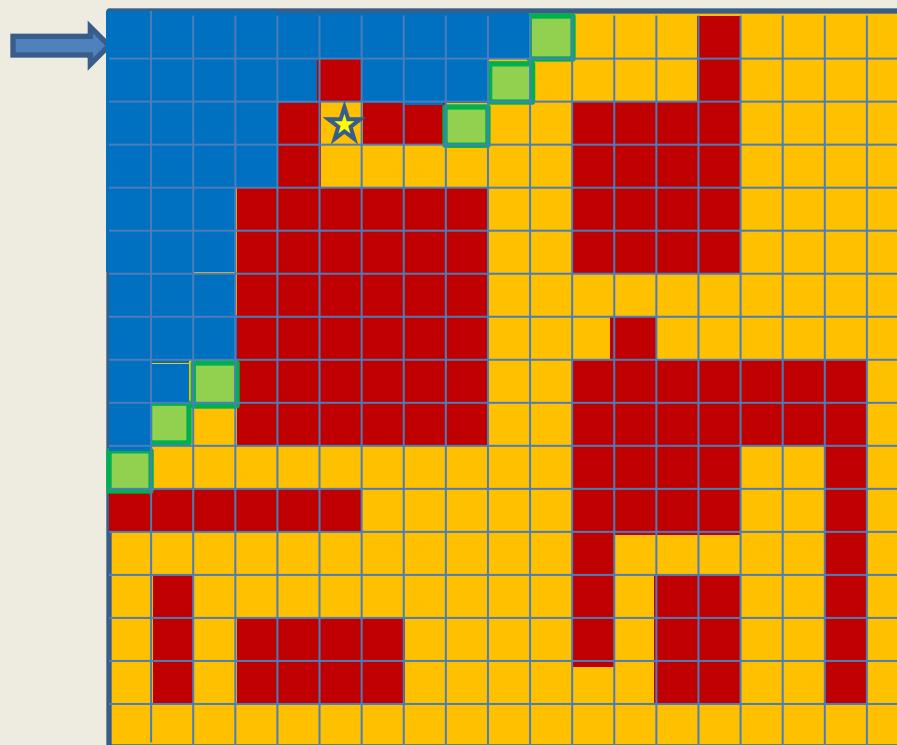
ripple reaches cells at distance 8 in step 8



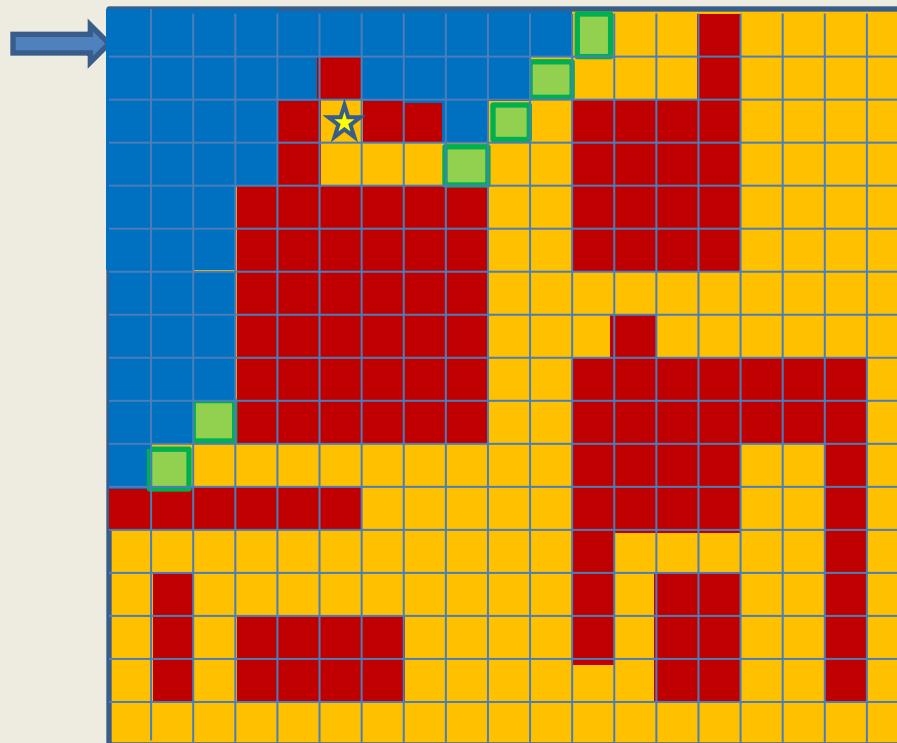
ripple reaches cells at distance 9 in step 9



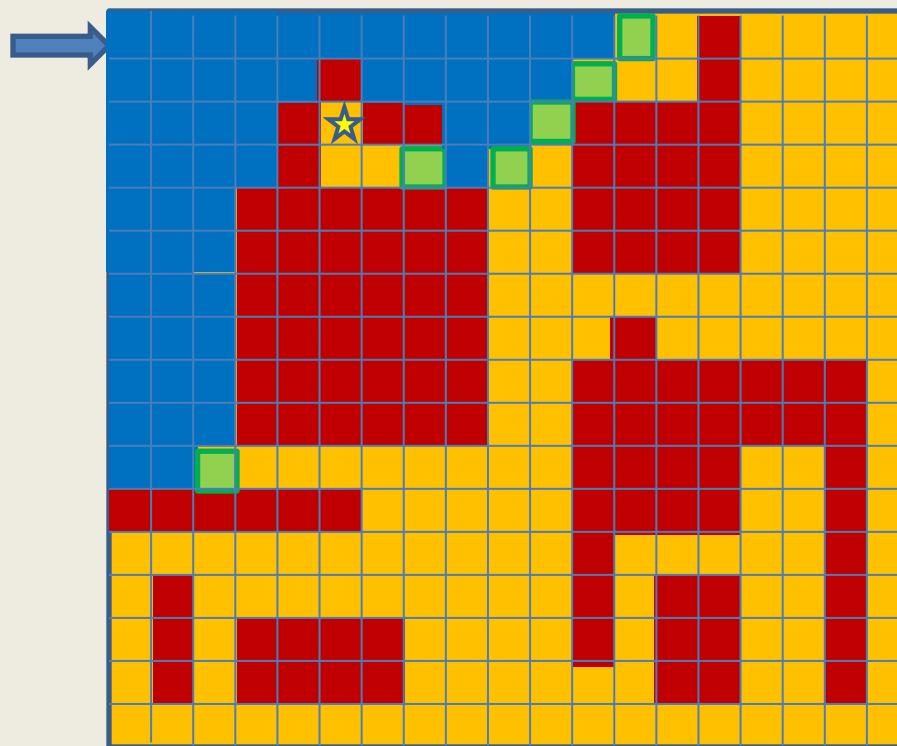
ripple reaches cells at **distance 10** in **step 10**



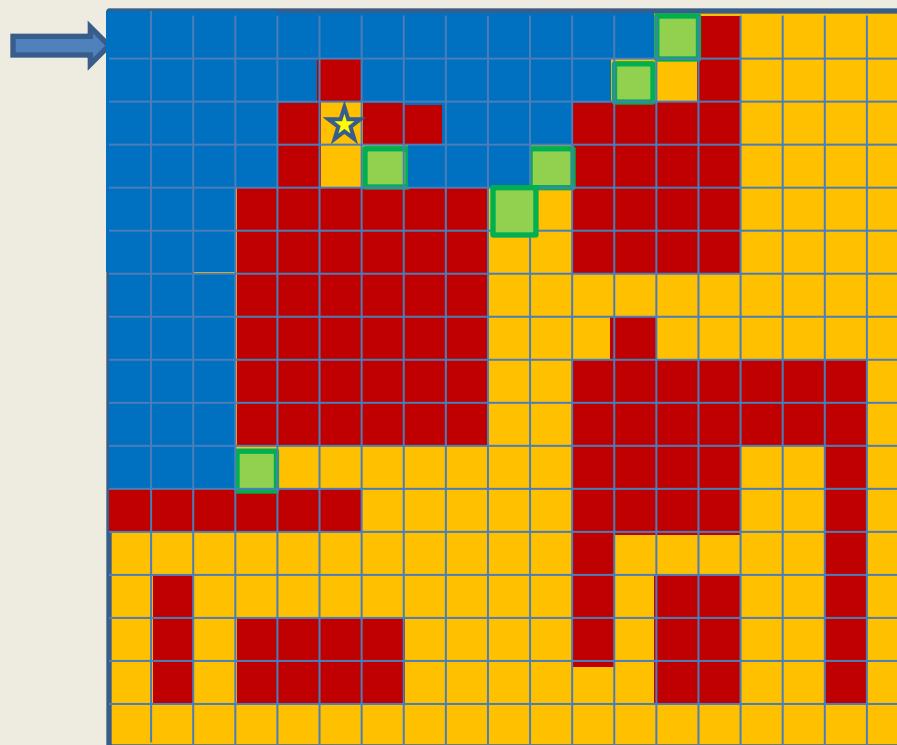
ripple reaches cells at **distance 11** in **step 11**



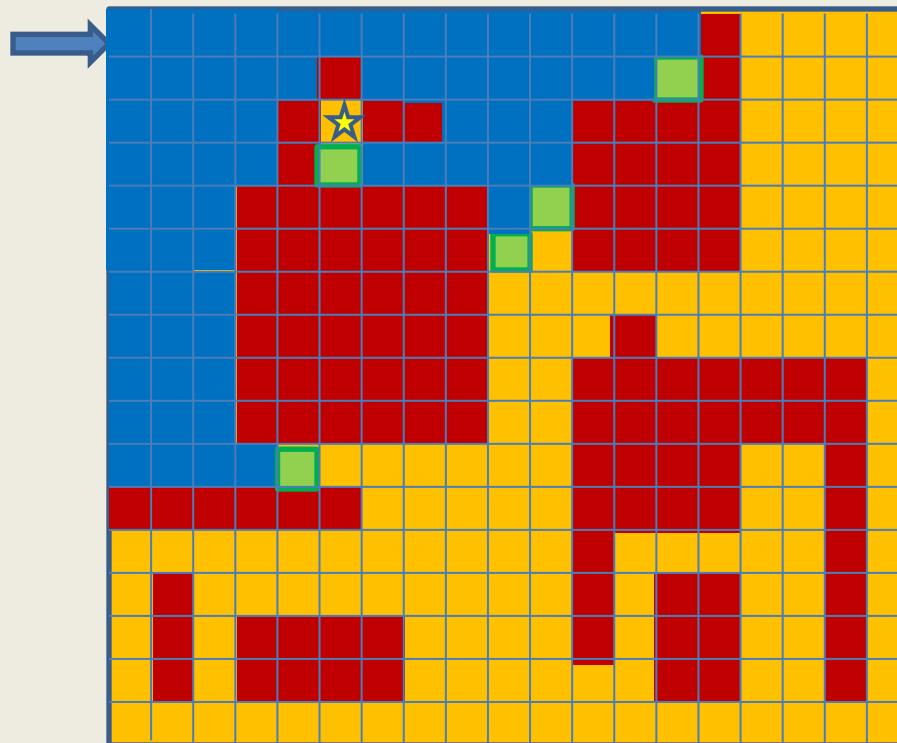
ripple reaches cells at **distance 12** in **step 12**



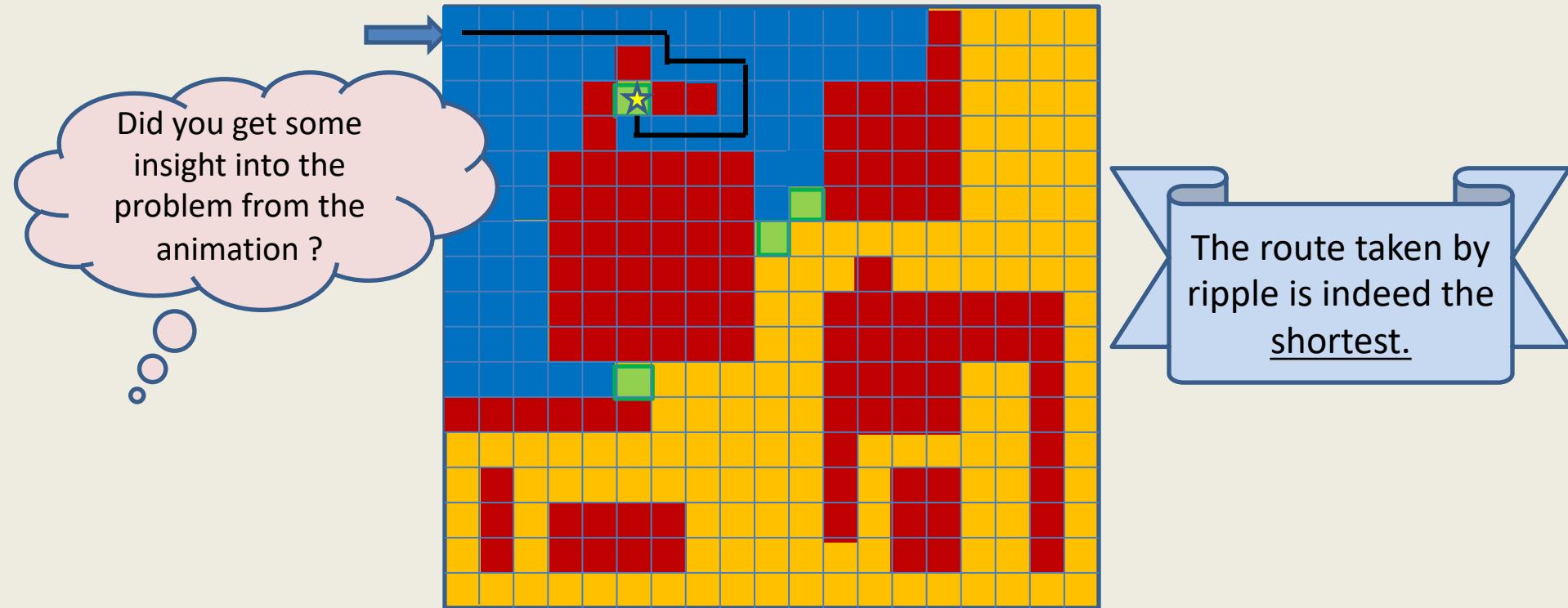
ripple reaches cells at **distance 13** in **step 13**



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 ☺.

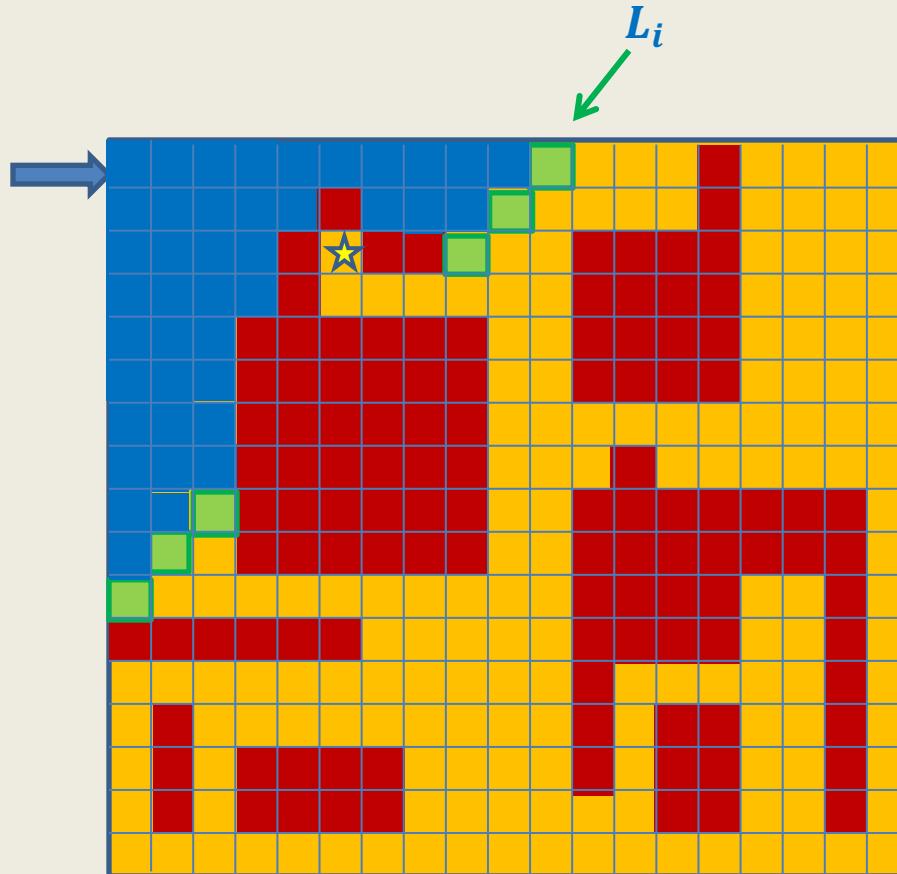
Step 2:

Designing algorithm for distances in grid

(using an insight into propagation of ripple)

A snapshot of ripple after i steps

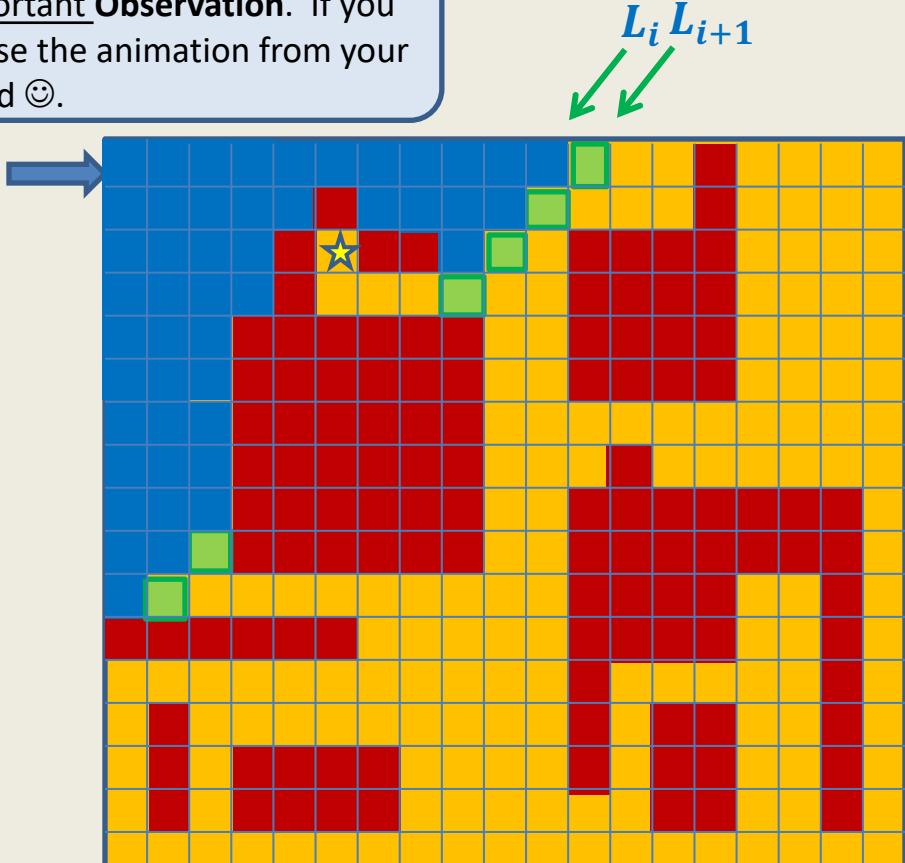
A snapshot of ripple after i steps



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	15	7	8	9	10	11	12	13	14	26	27	28	29			
2	3	4	5	14	13	12	11	12	13					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				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	

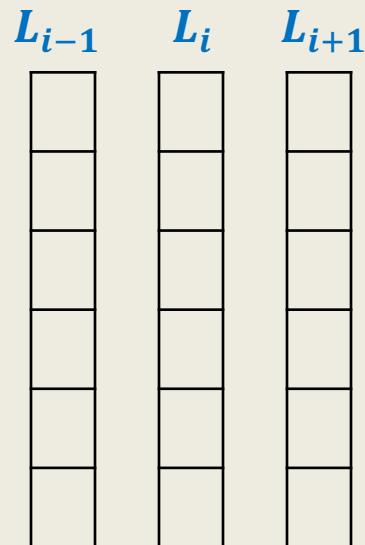
Observation: Each cell of L_{i+1} is a neighbor of a cell in L_i .

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 ?

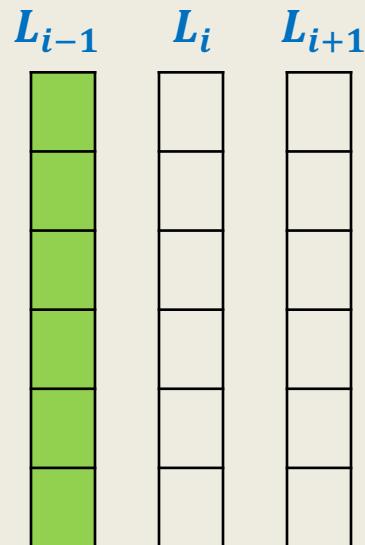
How can we generate L_{i+1} from L_i ?

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



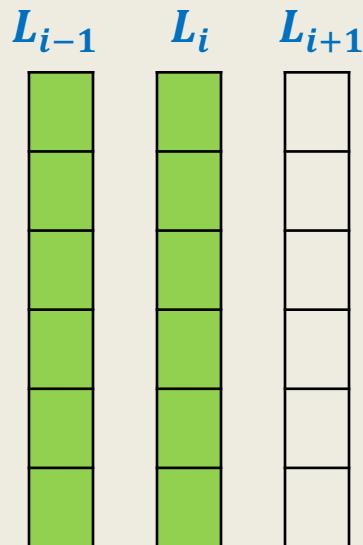
How can we generate L_{i+1} from L_i ?

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



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

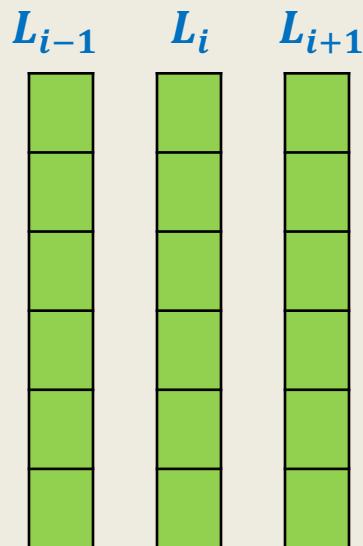
Suppose all cells of L_{i-1} get visited first.
Then all cells of L_i are visited, and



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:

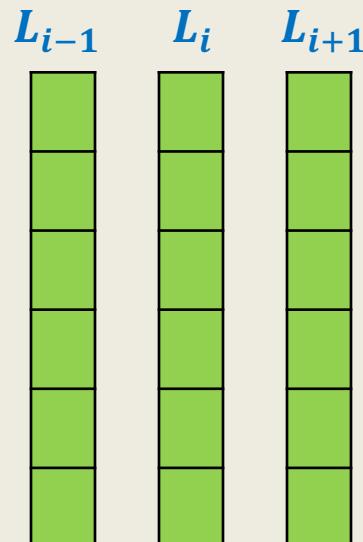
Initialize the distance of all cells except start cell as ∞

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
 $O(n^2)$

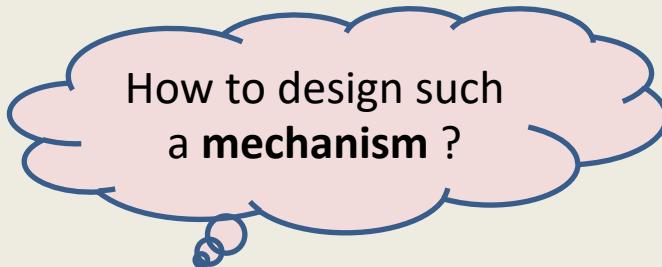
The algorithm is not elegant because of

- So many temporary lists that get created.

Towards an **elegant** algorithm ...

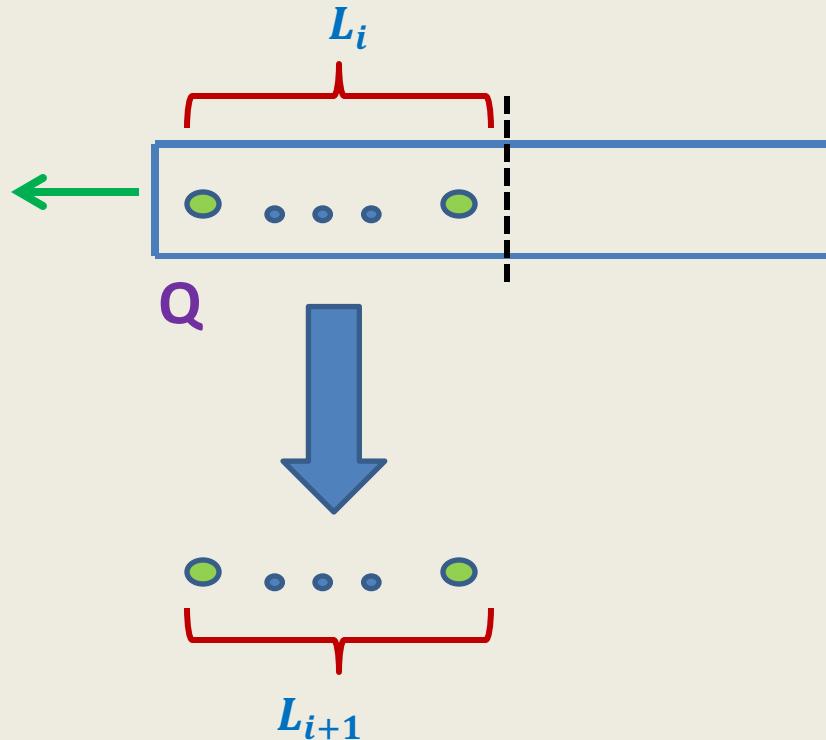
Key points we observed:

- 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,

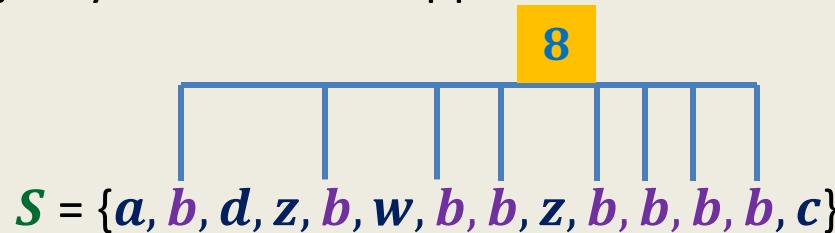
$x \in S$ is said to be majority element if it appears more than $n/2$ times in S .

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

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.

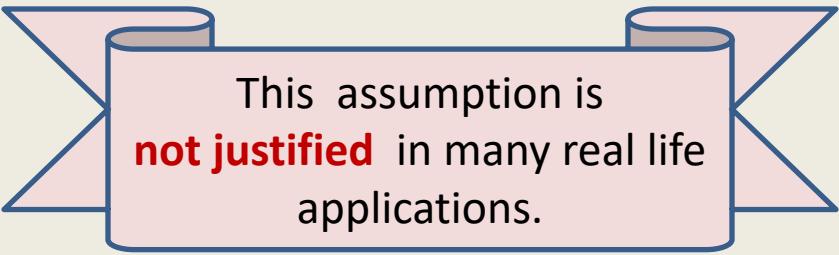
Running time: $\text{O}(n^2)$ time

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



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

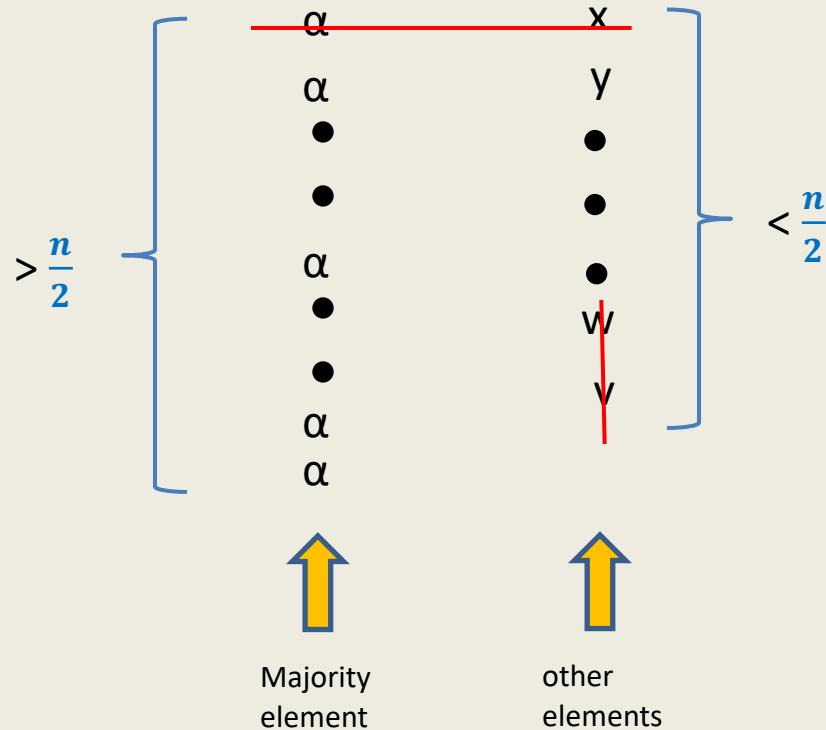
Problem: Given a multiset S of n elements,
where the only relation between any two elements is \neq or $=$,
find the majority element, if any, in S .

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

Answer: $O(n)$ time

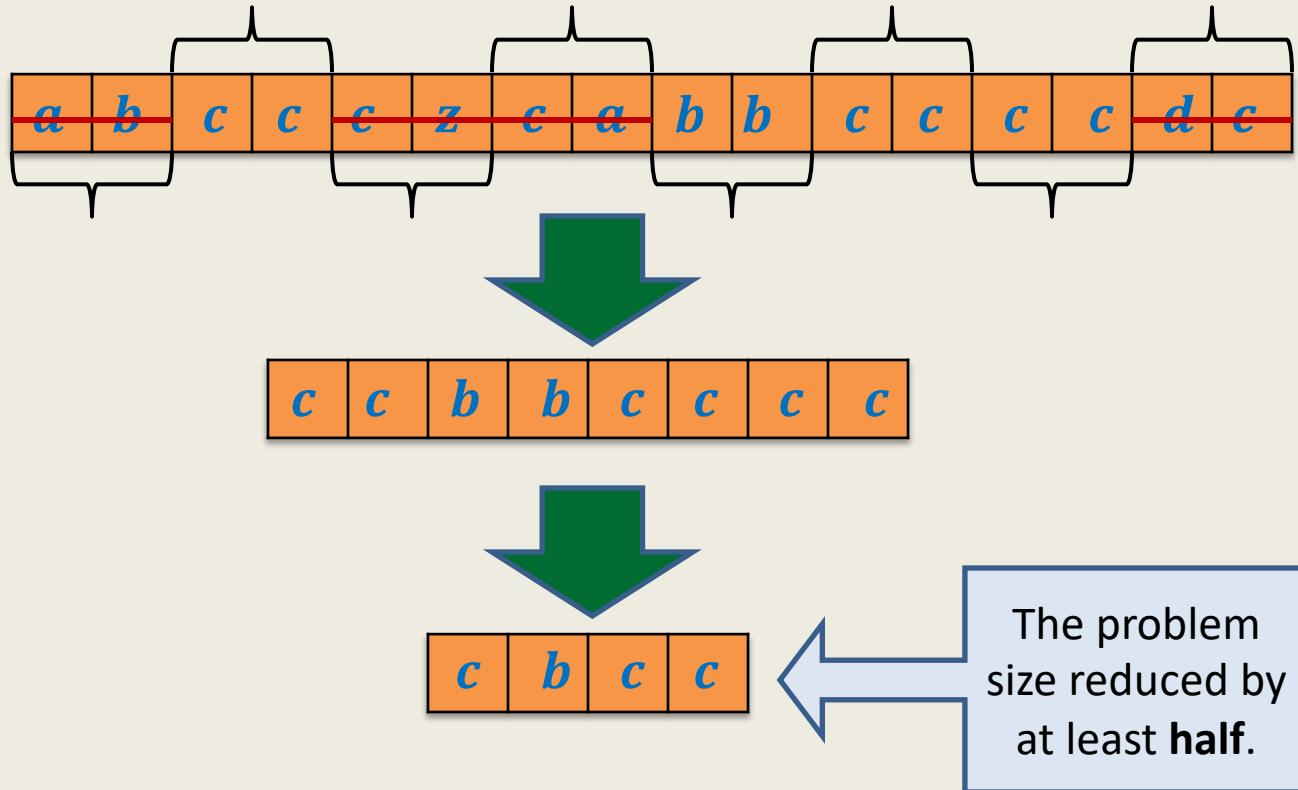
Observation 1: It is easy to verify whether an element is a majority

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 $O(n)$ extra (working) space in this case.

ALGORITHM FOR 2-MAJORITY ELEMENT

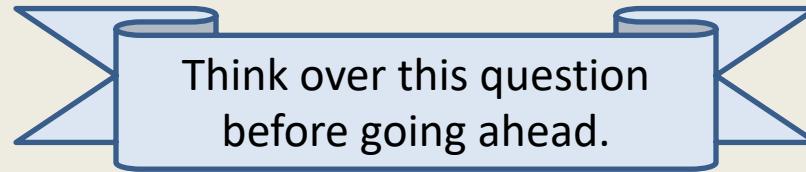
- Single scan and
- $O(1)$ extra space

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

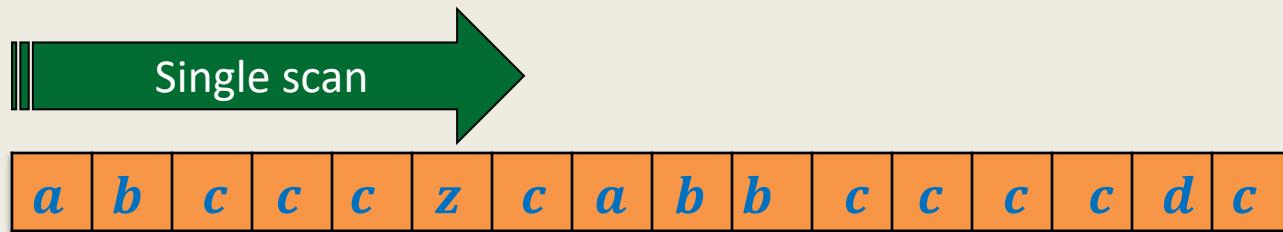
Question: Should we design algorithm from scratch to meet these constraints ?

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( $x \neq A[i]$ ) 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)**

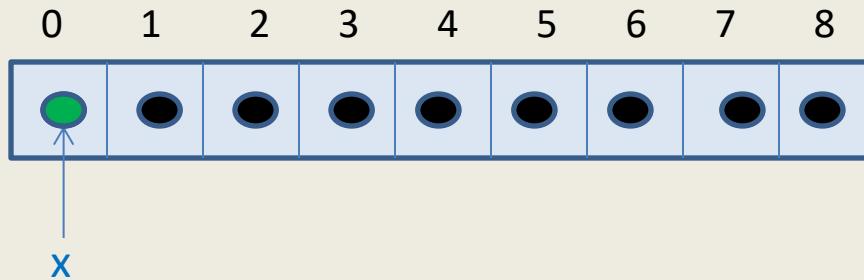
```
}
```

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

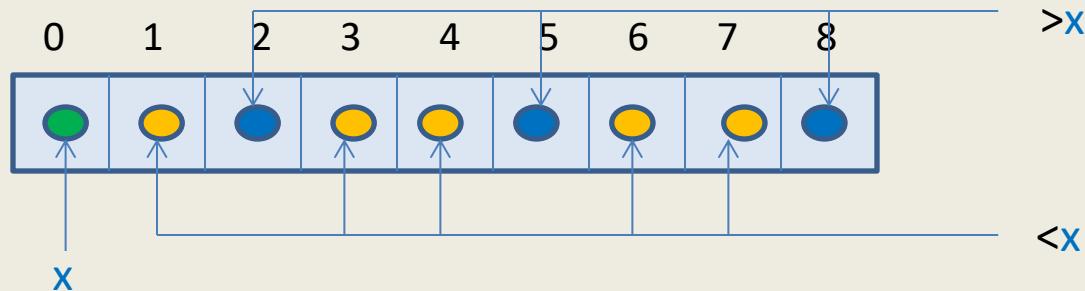
Theorem: There is an algorithm that makes just a **single scan** and uses **O(1)** **extra space** to compute majority element for a given multi-set.

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

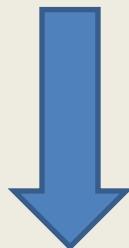
A nice programming exercise ?



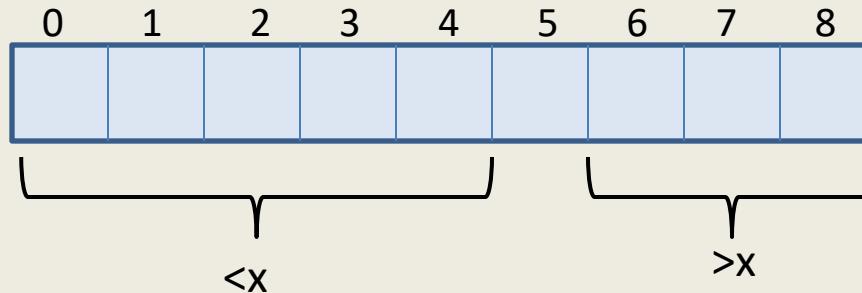
A nice programming exercise ?



A nice programming exercise ?



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



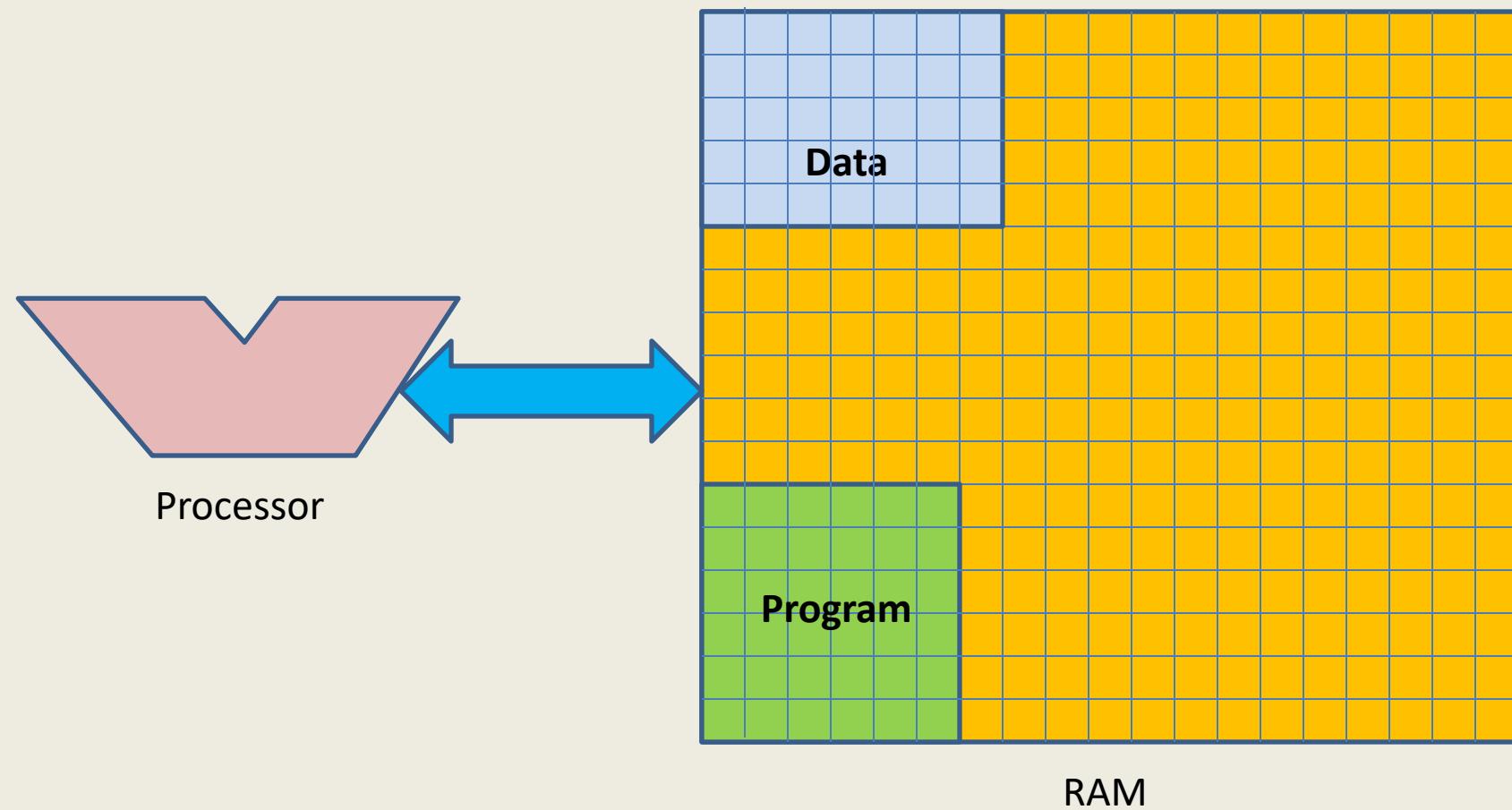
This procedure is called **Partition**.

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

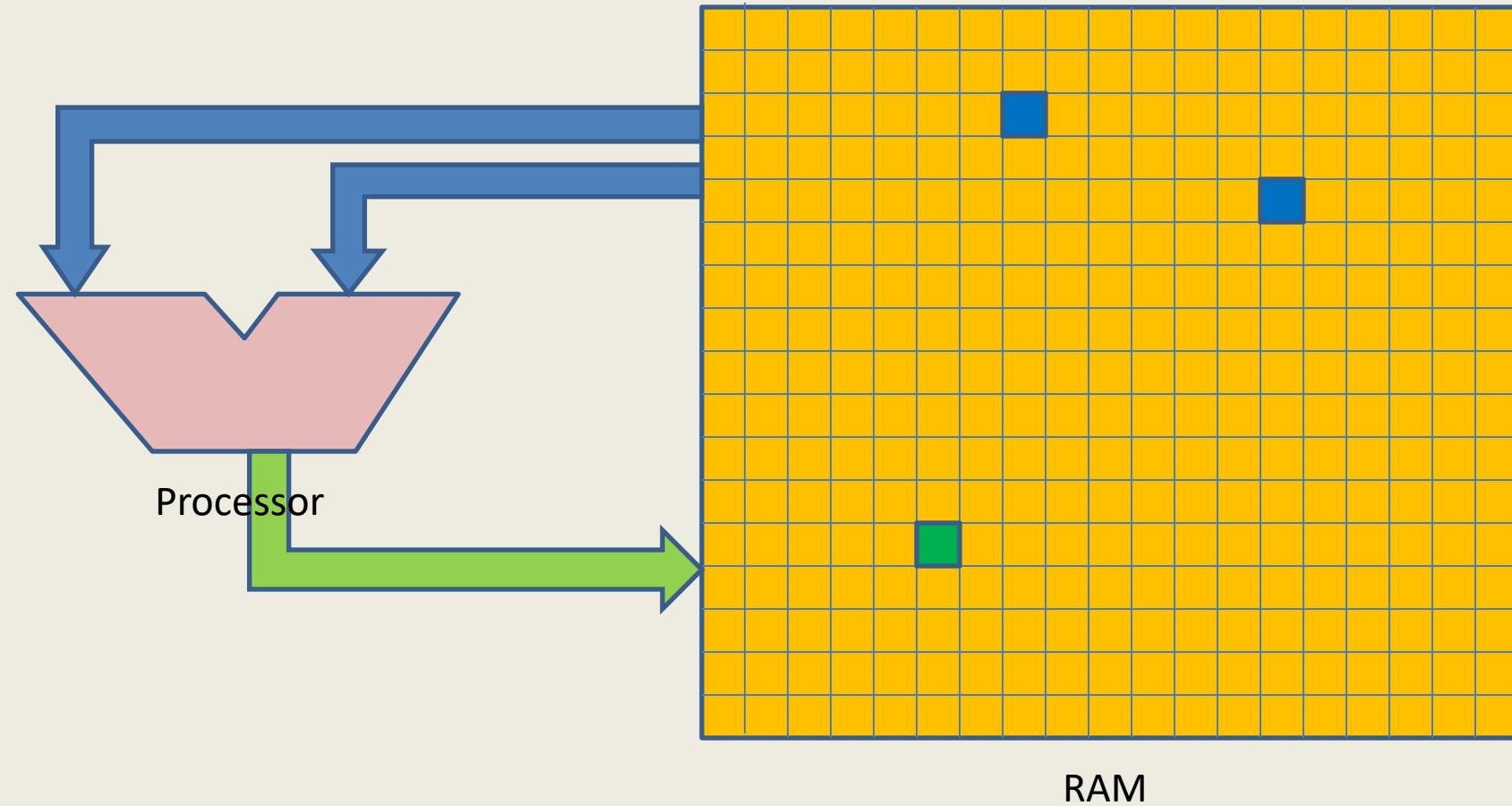
Further refinements

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:

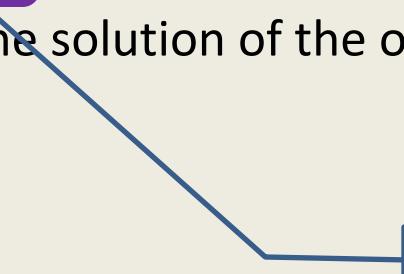
- Divide and Conquer
- Greedy Strategy
- Dynamic Programming
- Local Search

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

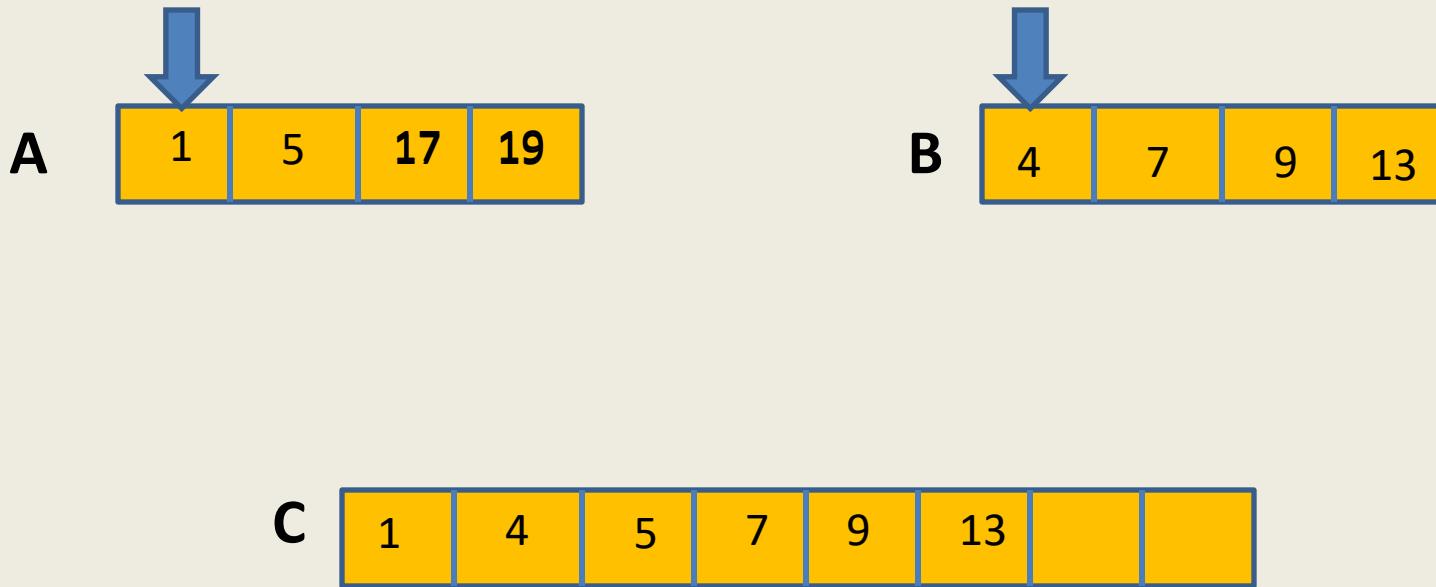
Merging two sorted arrays:

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

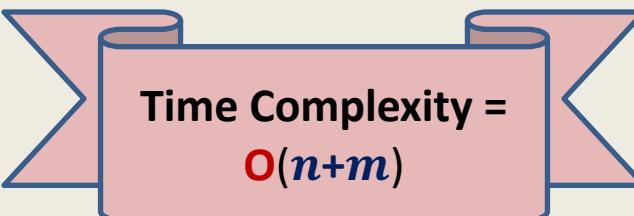


Pesudo-code for Merging two sorted arrays

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 $X+Y$?

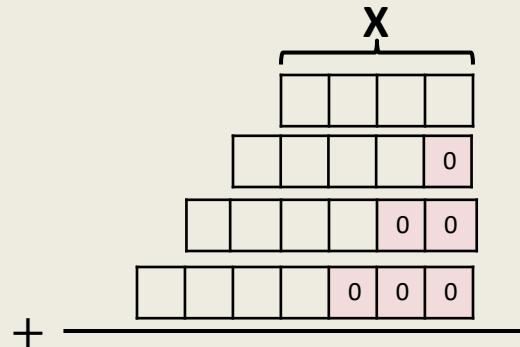
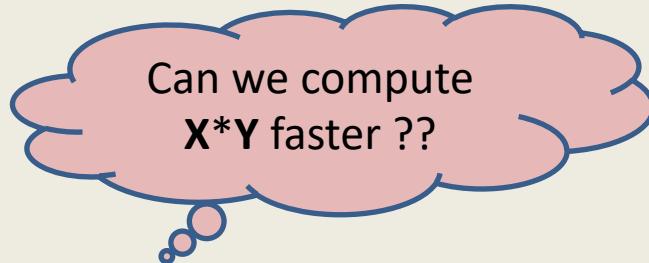
Answer: $O(n)$

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

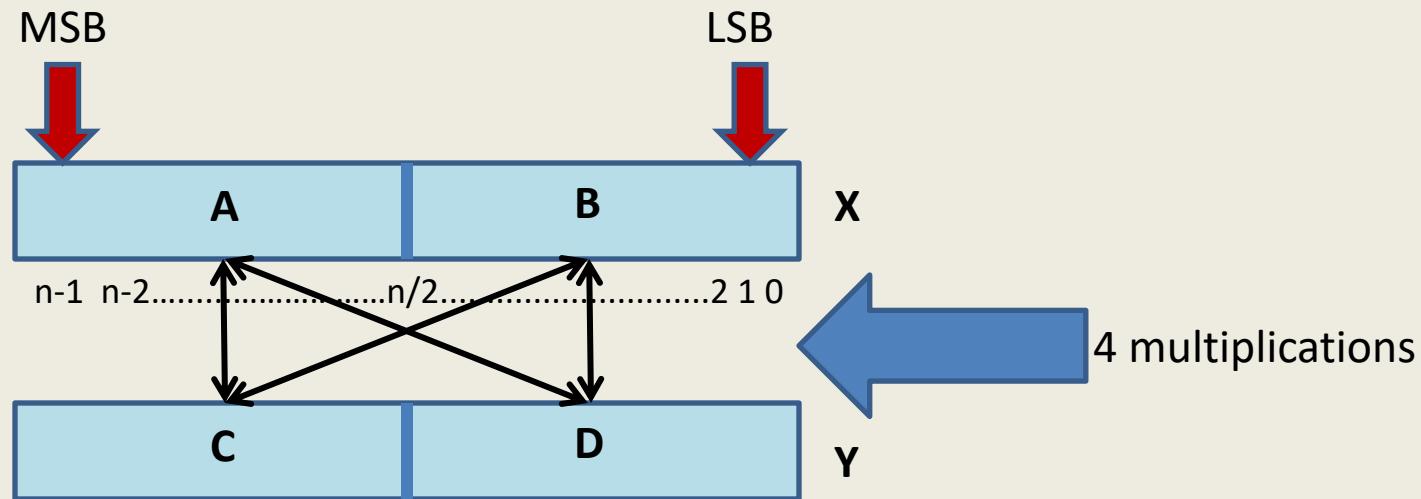
Answer: $O(n)$ [left shift the number X by n places, (do it carefully)]

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 $\{A, B, C, D\}$?

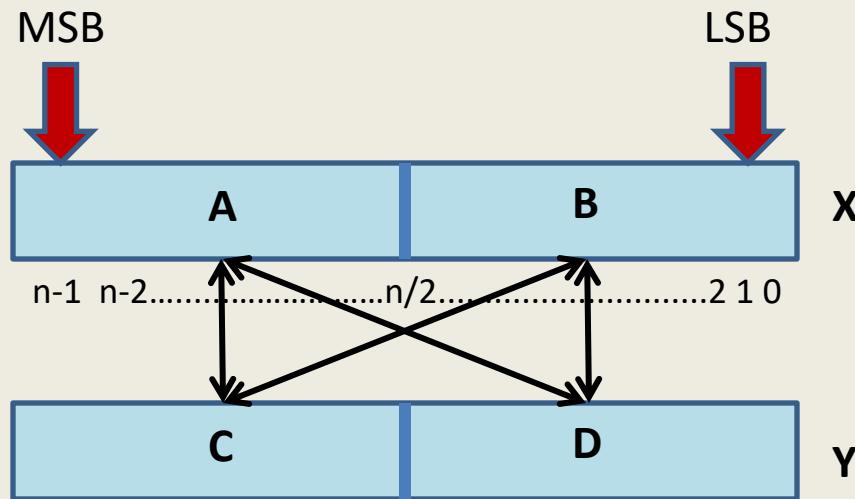
Hint: First Express X and Y in terms of $\{A, B, C, D\}$.

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

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.

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

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

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

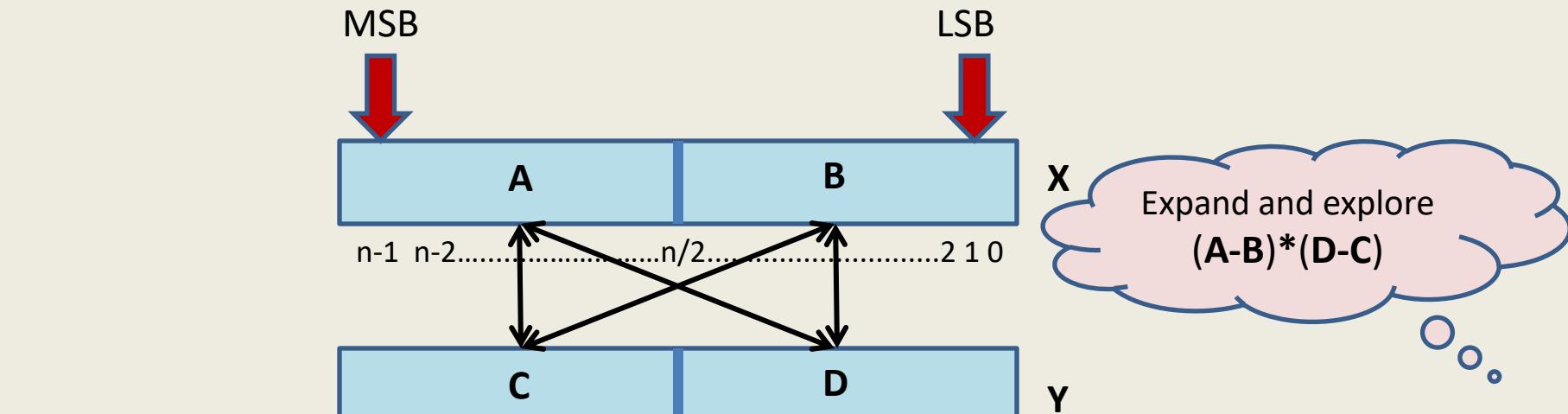
$$= c n + 2c n + 4c n + 8c n + \dots + 4^{\log_2 n} T(1)$$

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



$O(n^2)$ time algo

Pursuing Divide and Conquer approach



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

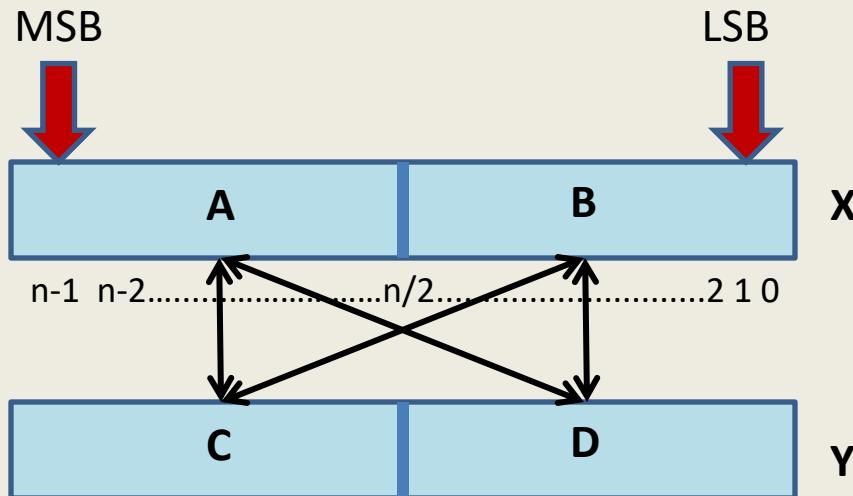
Observation: $A*D + B*C = (A-B)*(D-C) + A*C + B*D$

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 :

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

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

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

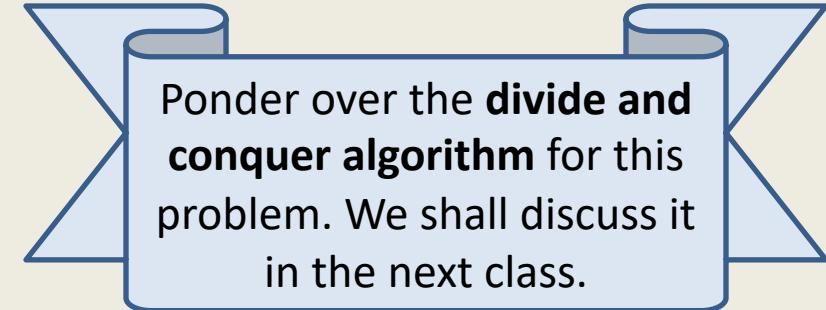
AIM: An efficient algorithm to count the number of inversions in an array A .

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)

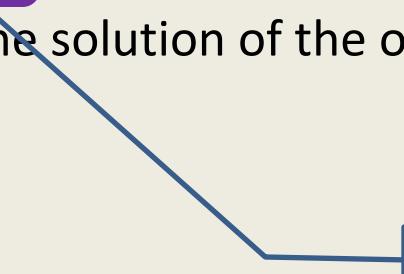
Lecture 15:

- **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

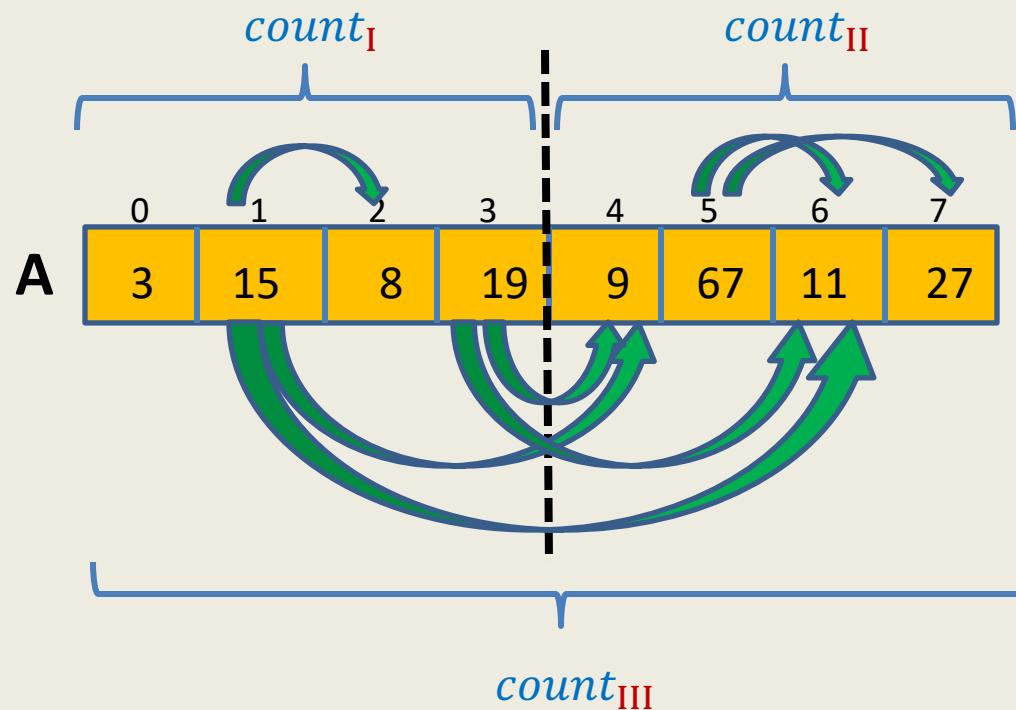
Trivial-algo($A[0..n - 1]$)

```
{ 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);
```

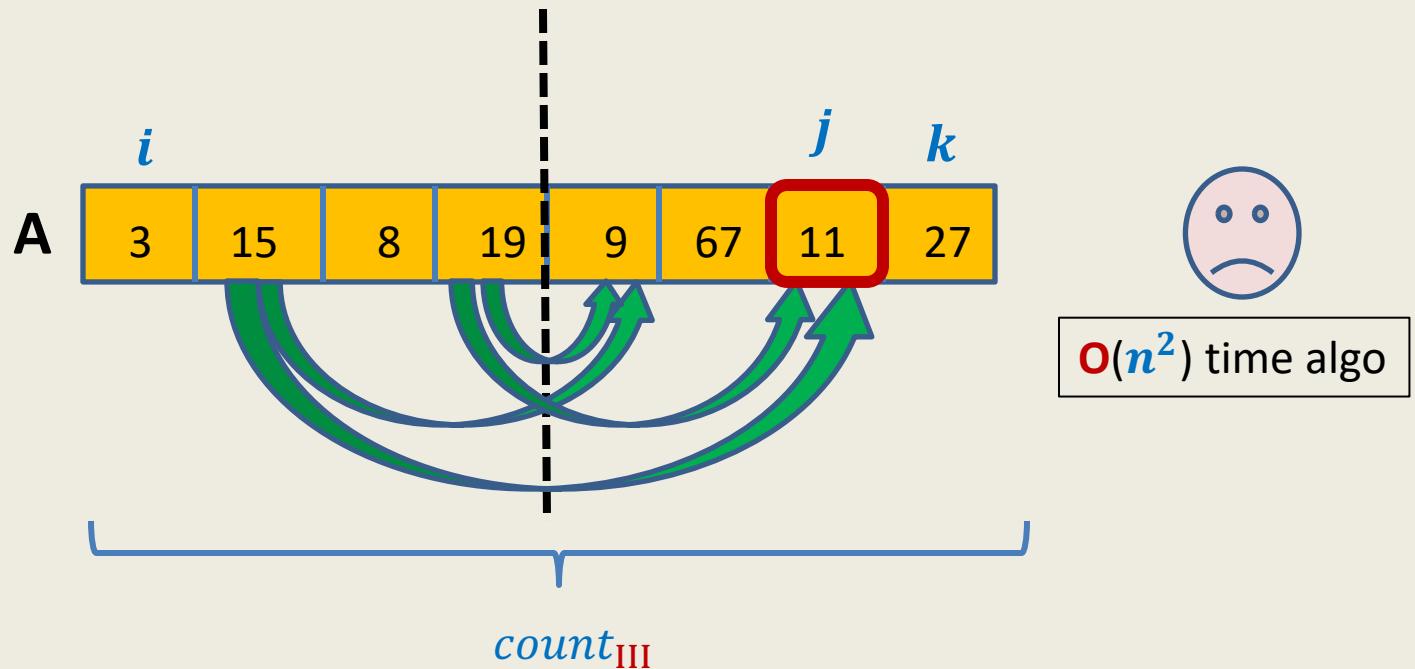
```
    countII ← CountInversion(A,mid + 1,k);
```

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

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

```
}
```

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



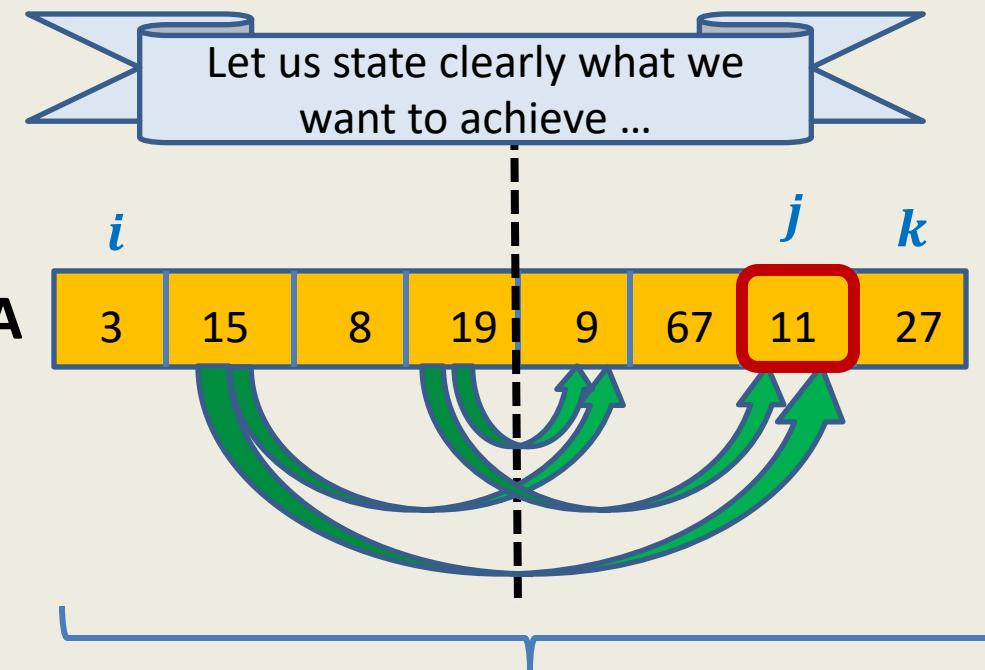
Aim: For each $mid < j \leq k$, count the elements in $A[i..mid]$ that are **greater** than $A[j]$.

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 $n = 1$,

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

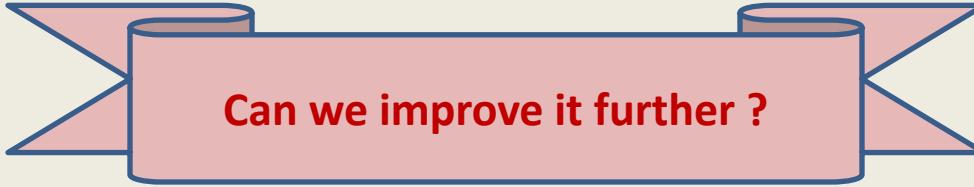
If $n > 1$,

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

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

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

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



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

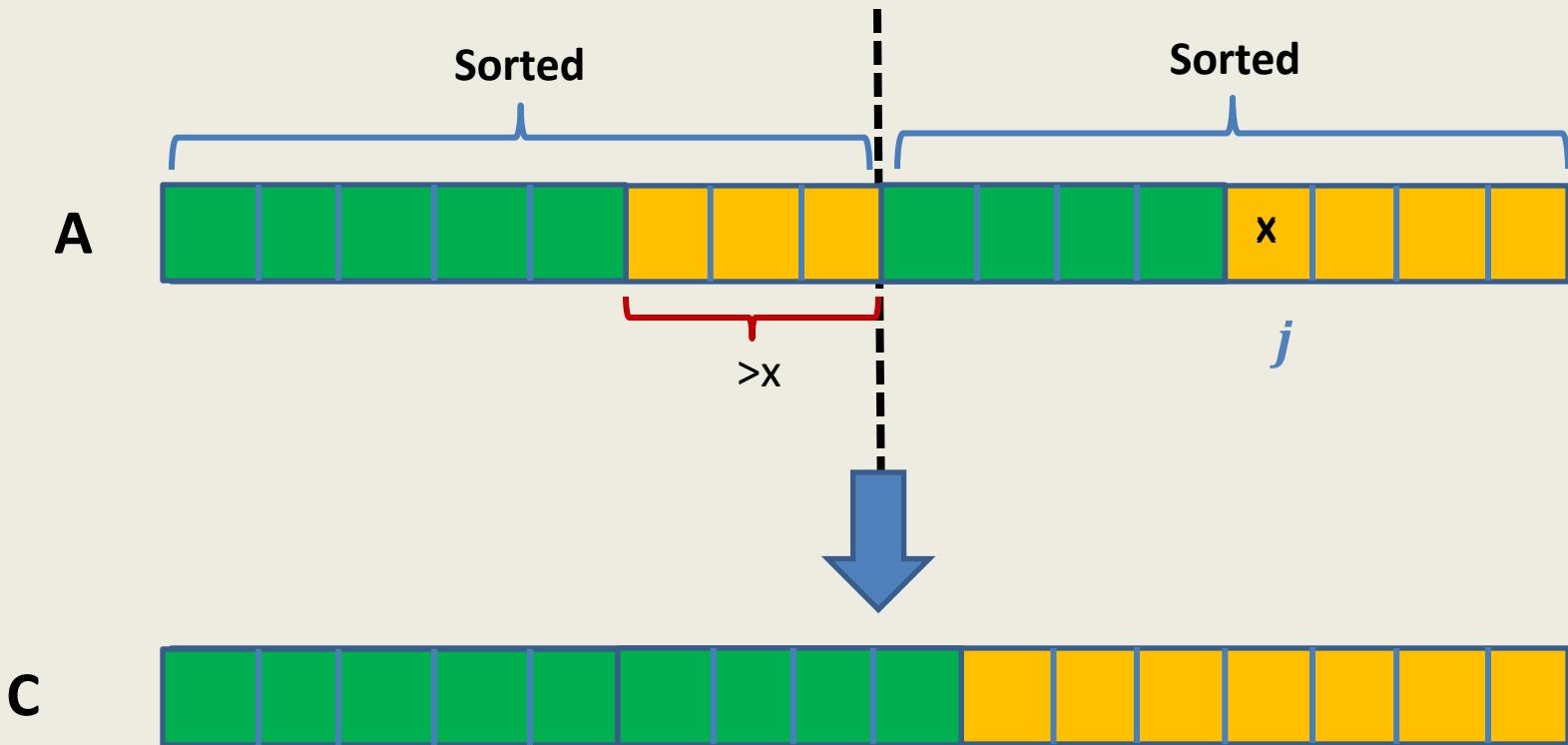
MSort(A, i , k)// Sorting A[$i..k$]

```
{  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*++ }

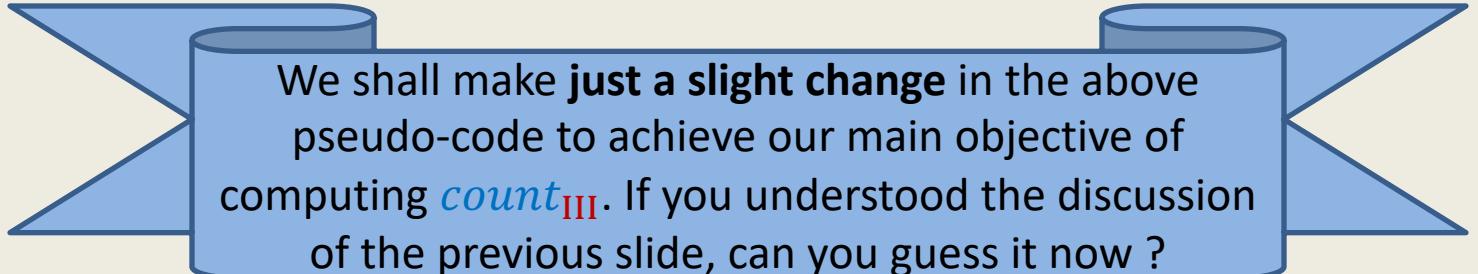
Else { C[*r*] \leftarrow A[*j*]; *r*++; *j*++ }

}

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 $n > 1$,

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

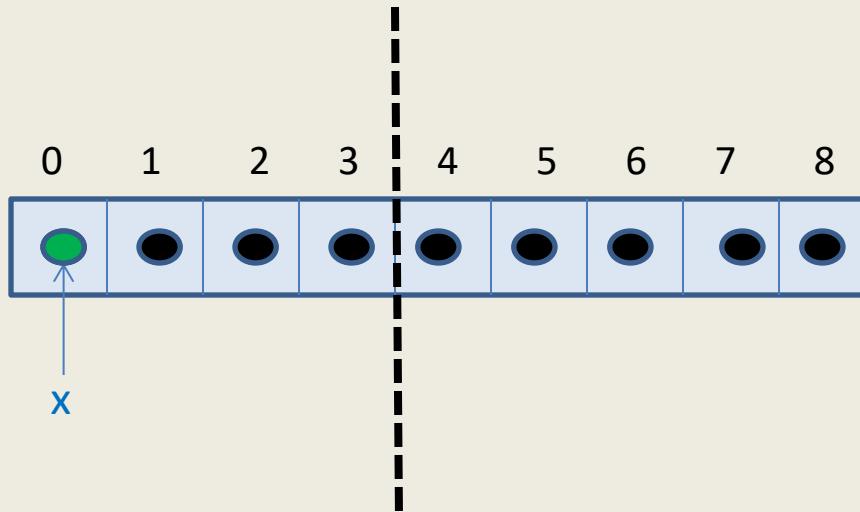
$$= O(n \log n)$$

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 $O(n \log n)$.

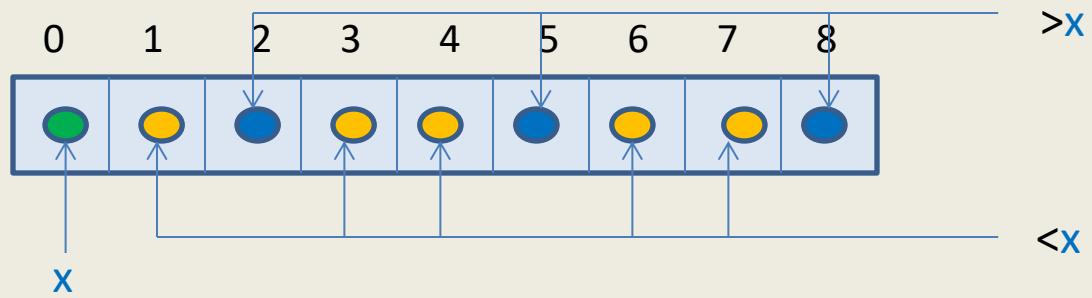
**Another sorting algorithm based on
divide and conquer**

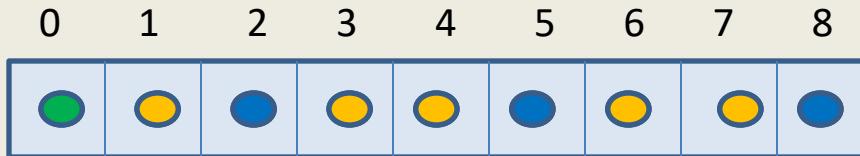
QuickSort

Is there any alternate way to divide ?

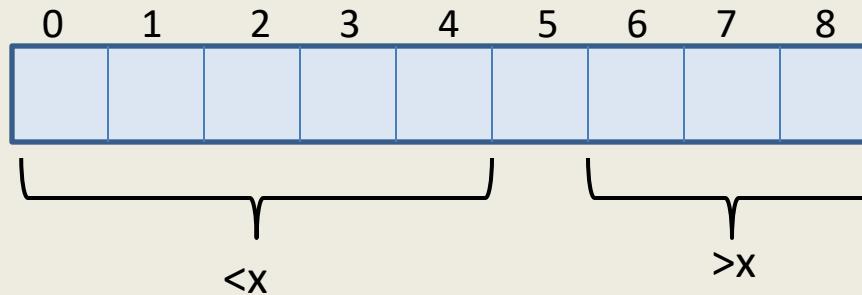
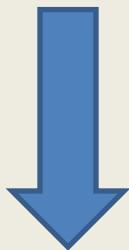


In MergeSort, we divide the input instance in an obvious manner.





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)

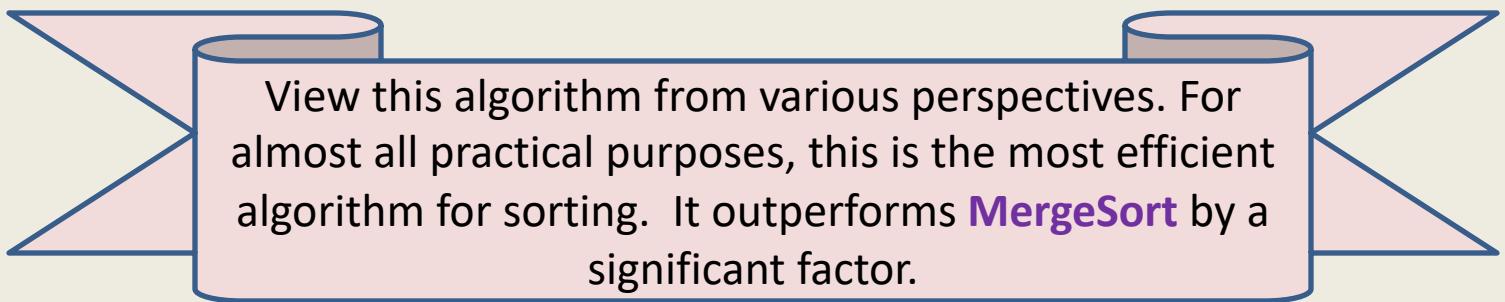
```
{    If ( $|S| > 1$ )
        Pick and remove an element  $x$  from  $S$ ;
         $(S_{<x}, S_{>x}) \leftarrow \text{Partition}(S, x);$ 
        return( Concatenate(QuickSort( $S_{<x}$ ),  $x$ , QuickSort( $S_{>x}$ )))
}
```

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:

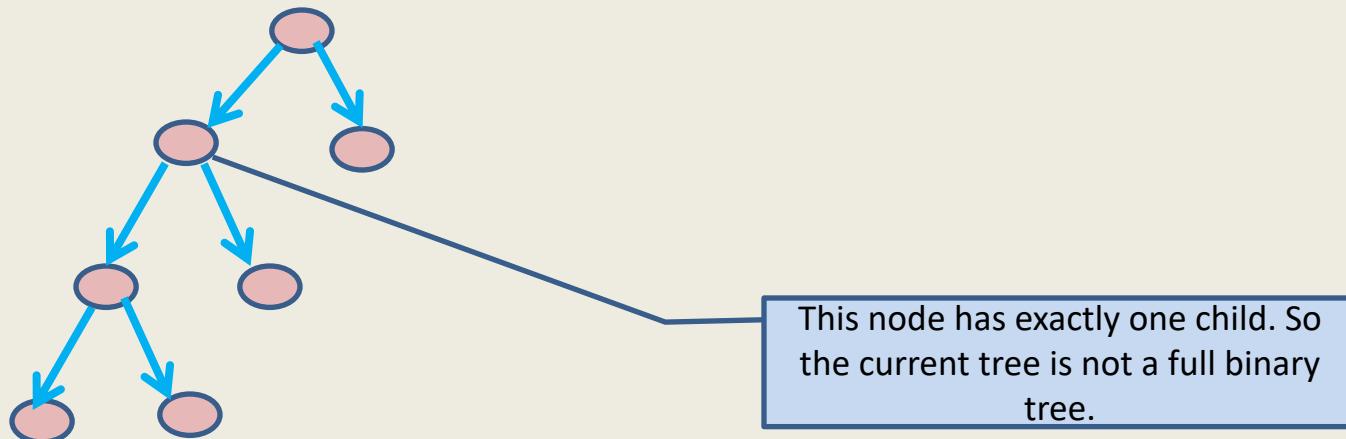
Height balanced BST

- Red-black trees

Terminologies

Full binary tree:

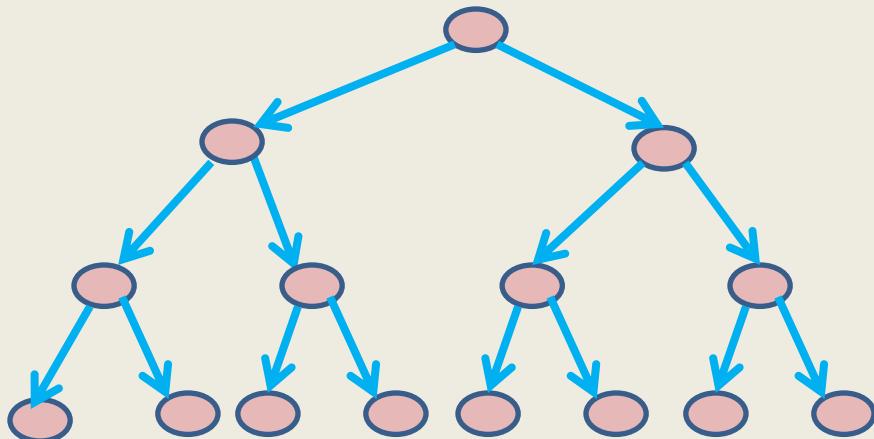
A binary tree where every internal node has exactly two children.



Terminologies

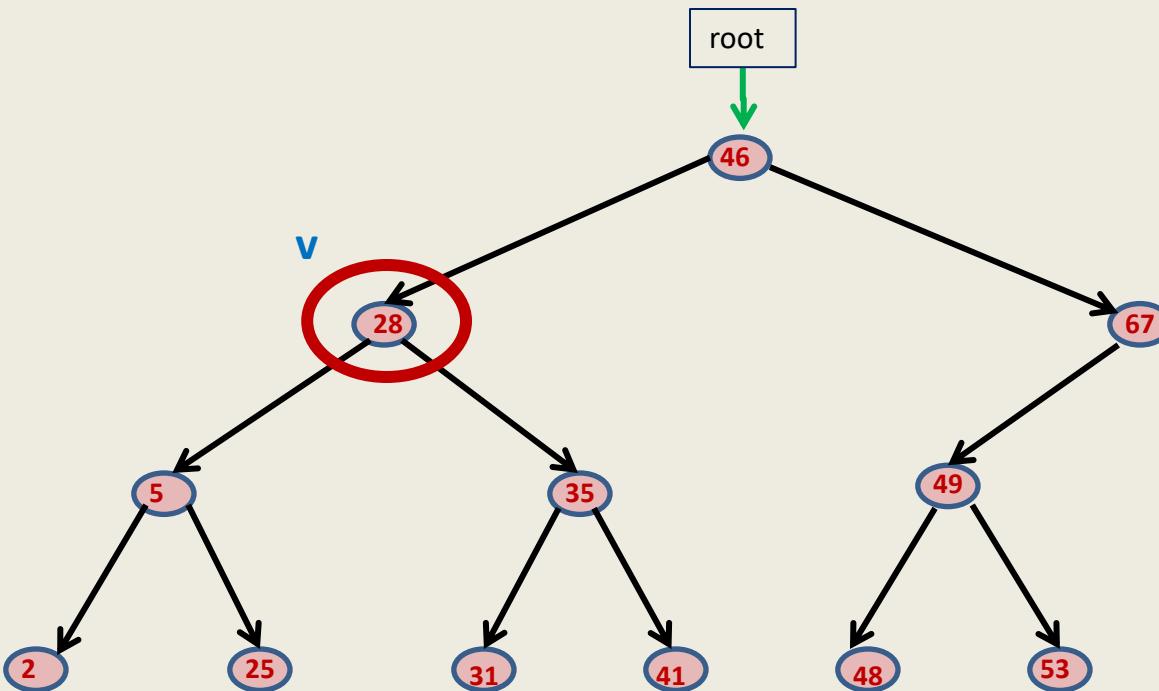
Complete binary tree:

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



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

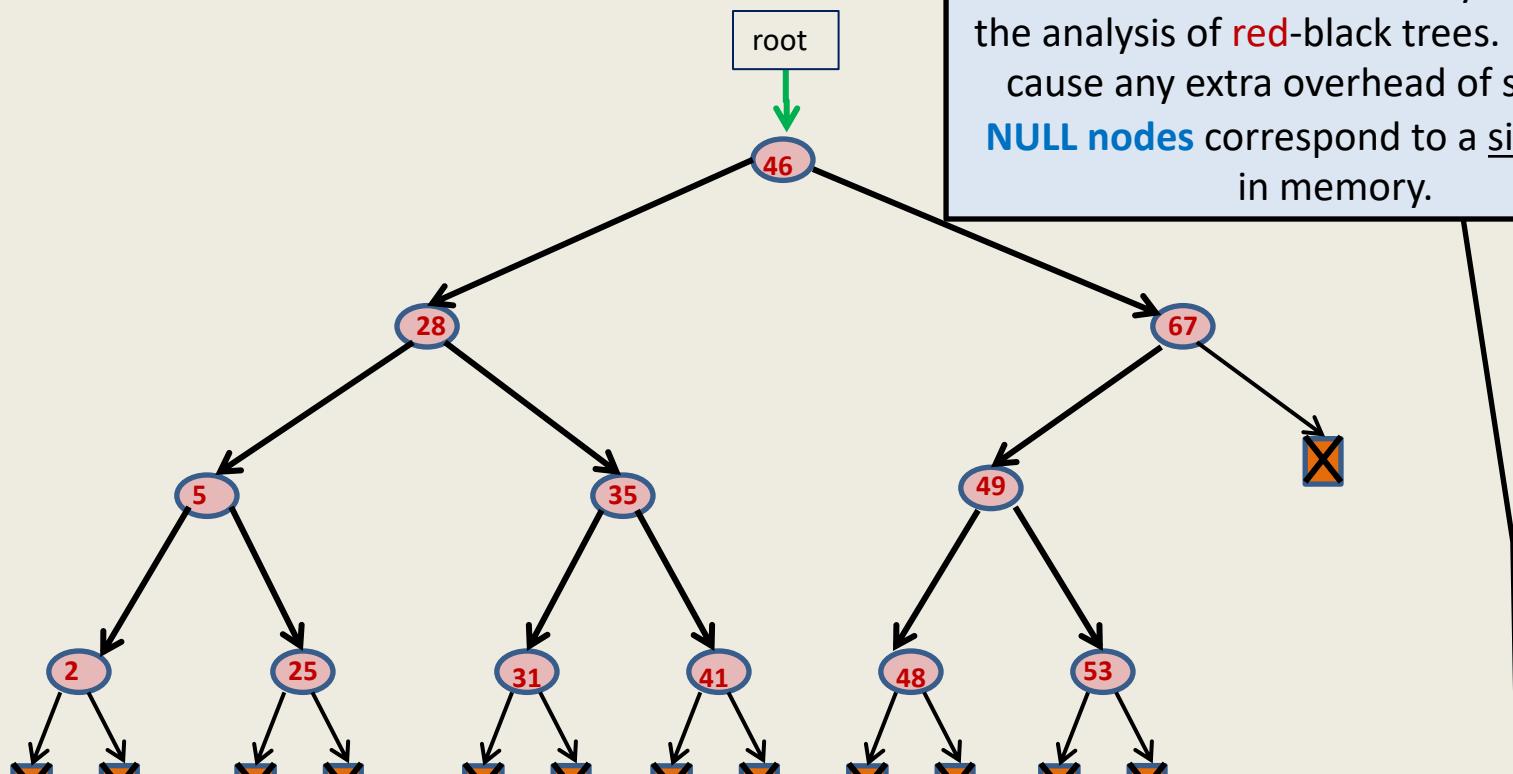
Binary Search Tree



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**.

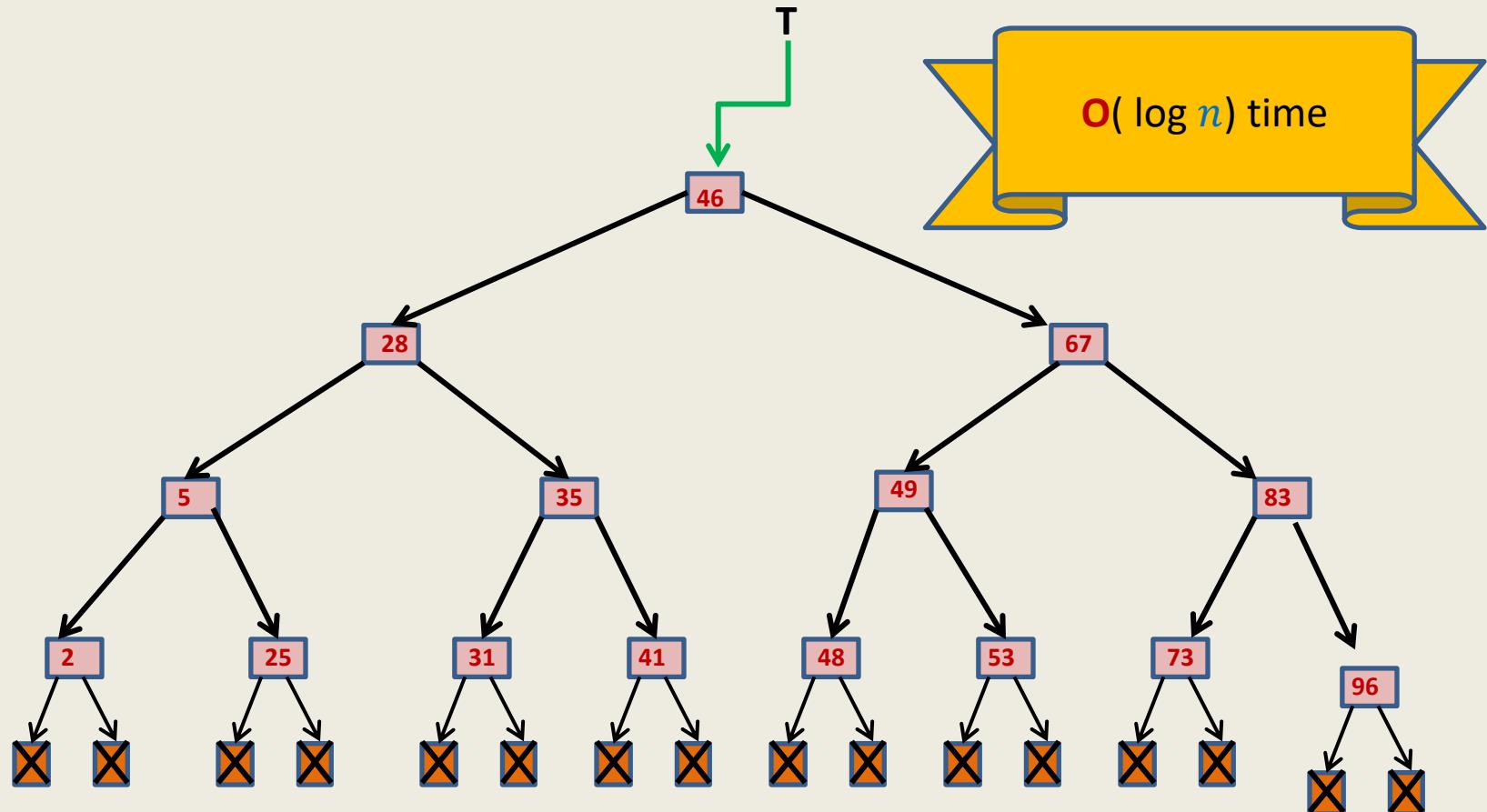
A fact we noticed in our previous discussion on BSTs (Lecture 9)

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

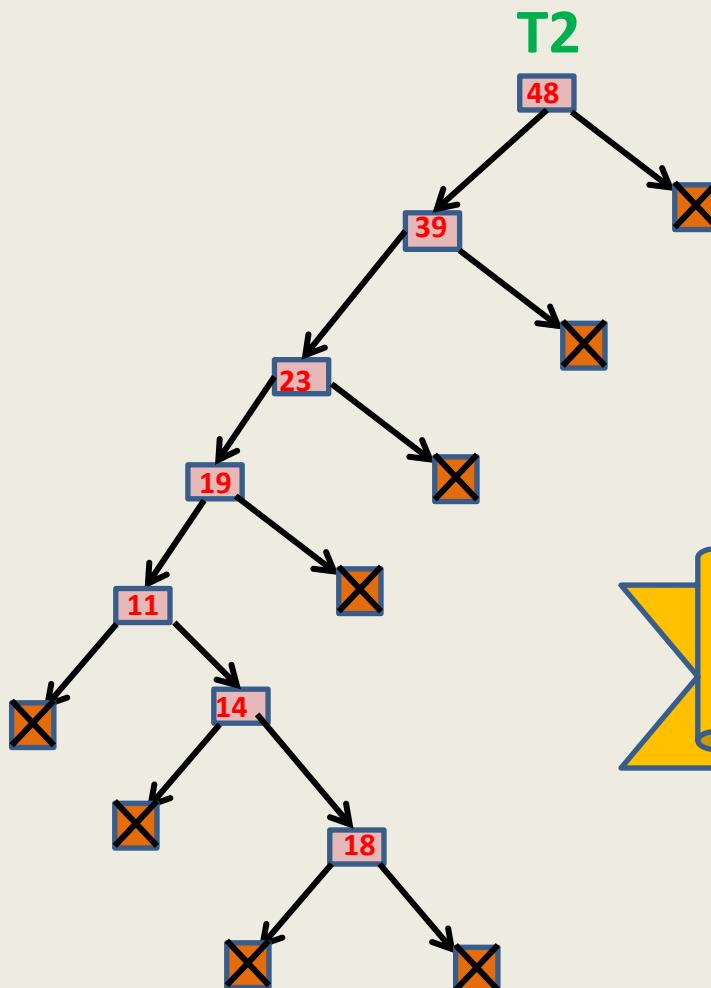
Height(T):

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] 

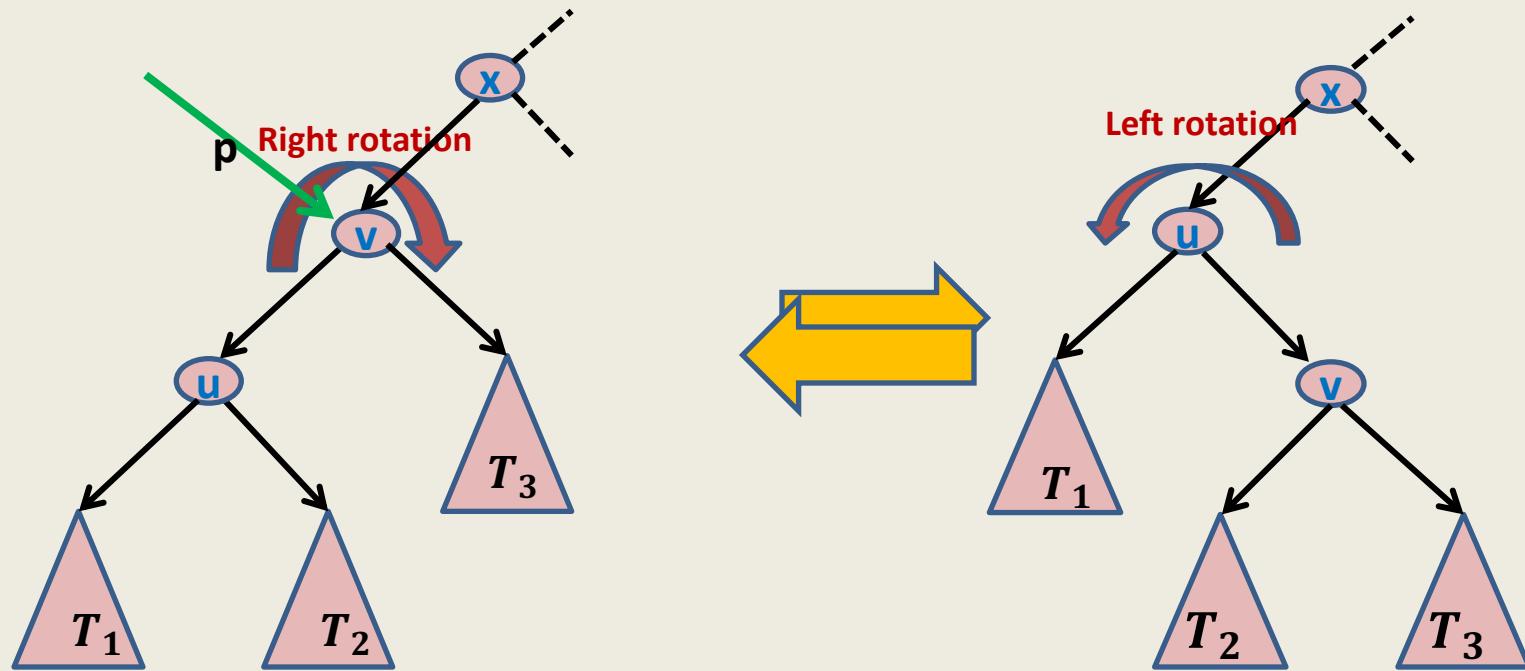
Rotation around a node

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

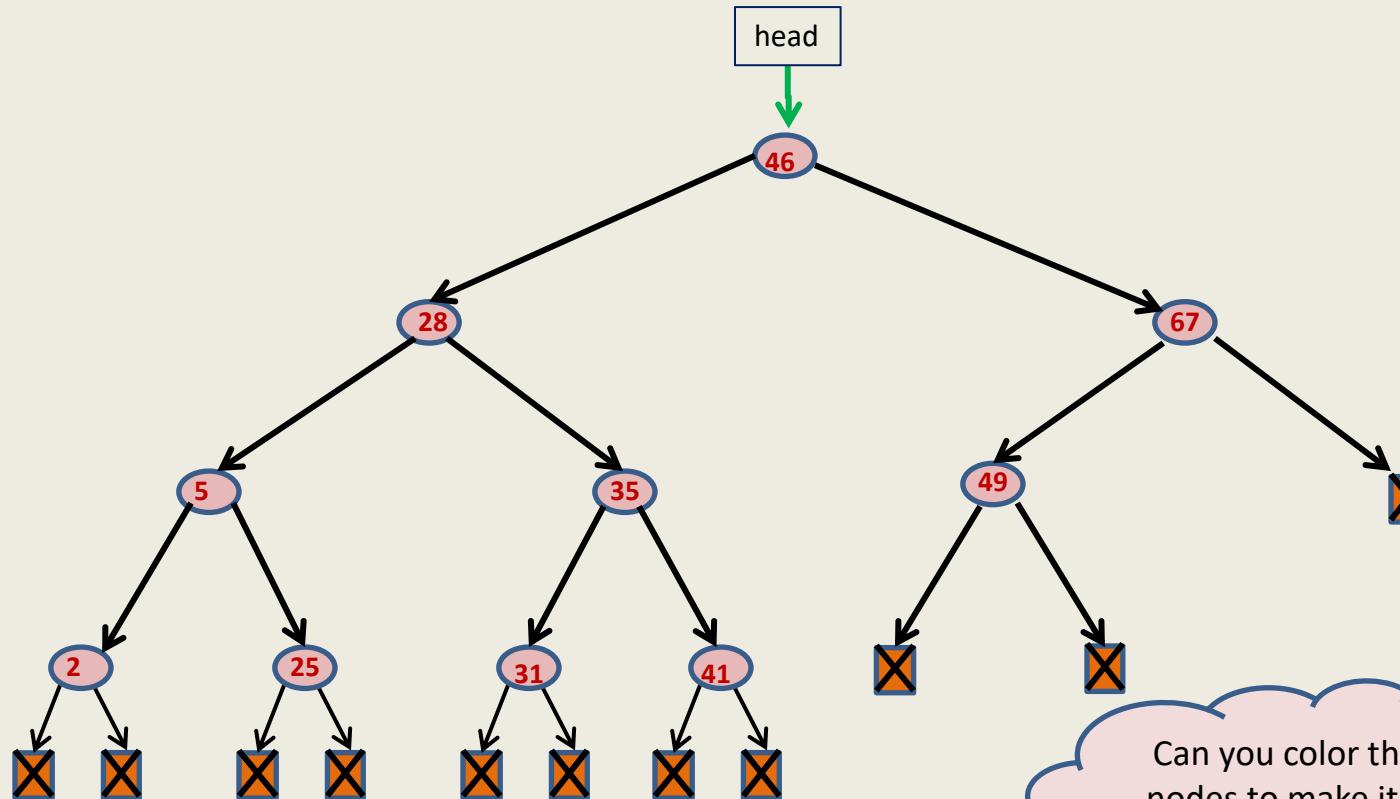
Red Black Tree

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.

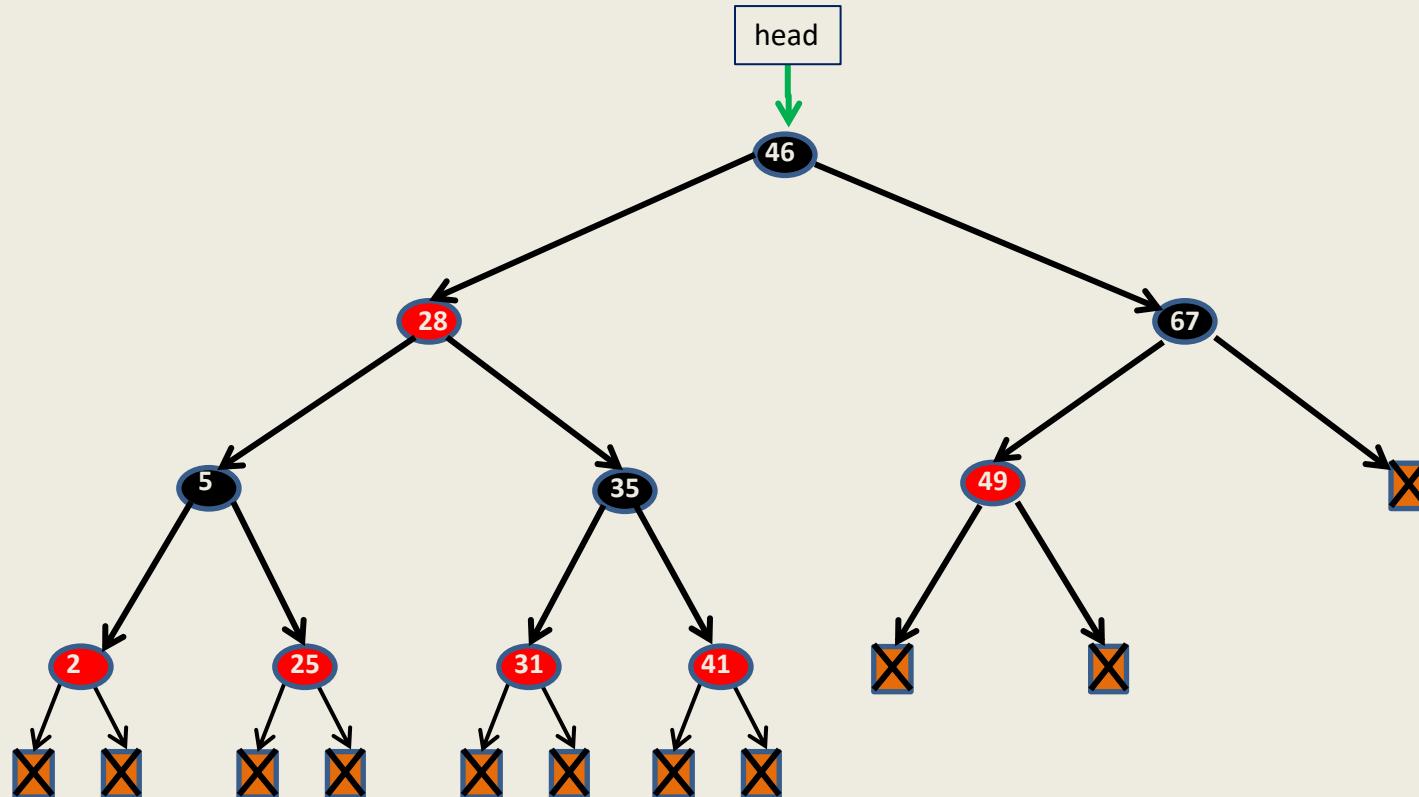
black height

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 ?

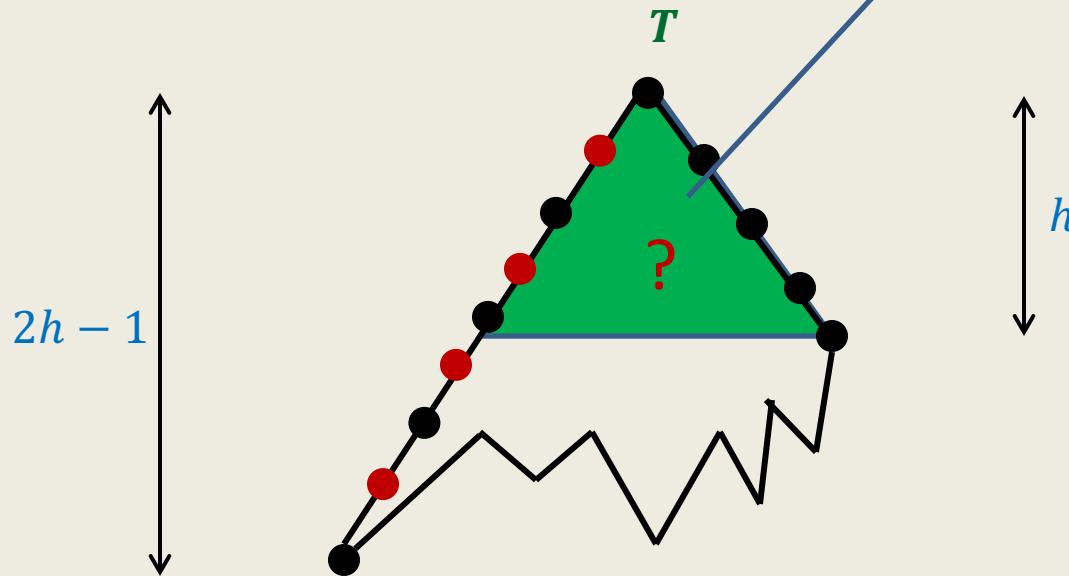
T : a red black tree

h : black height of T .

Question: What can be height of T ?

Answer: $\leq 2h - 1$

What is this “green structure” ?



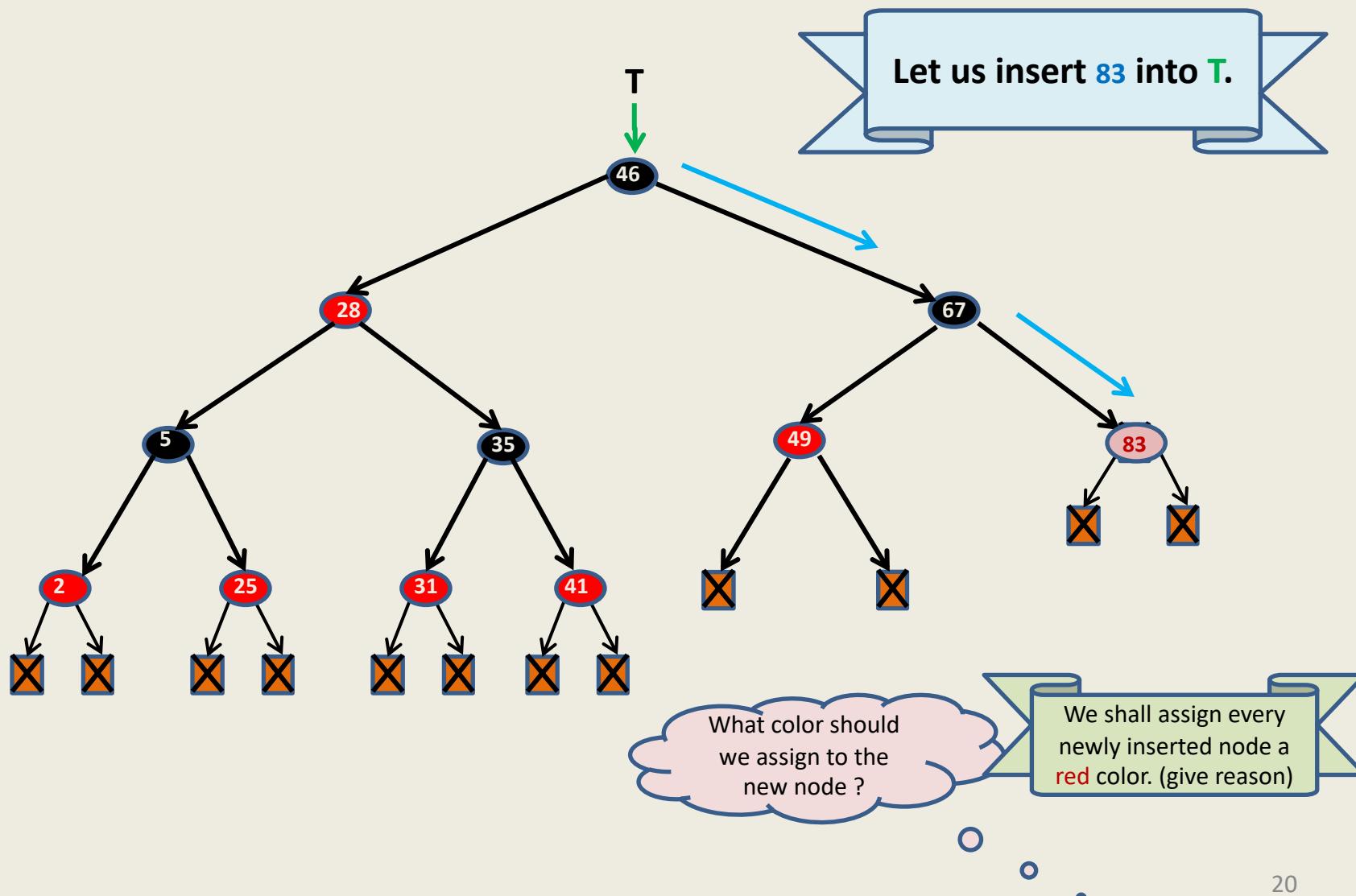
Homework: Ponder over the above hint to prove that T has $\geq 2^h - 1$ elements.¹⁸

Insertion in a Red Black tree

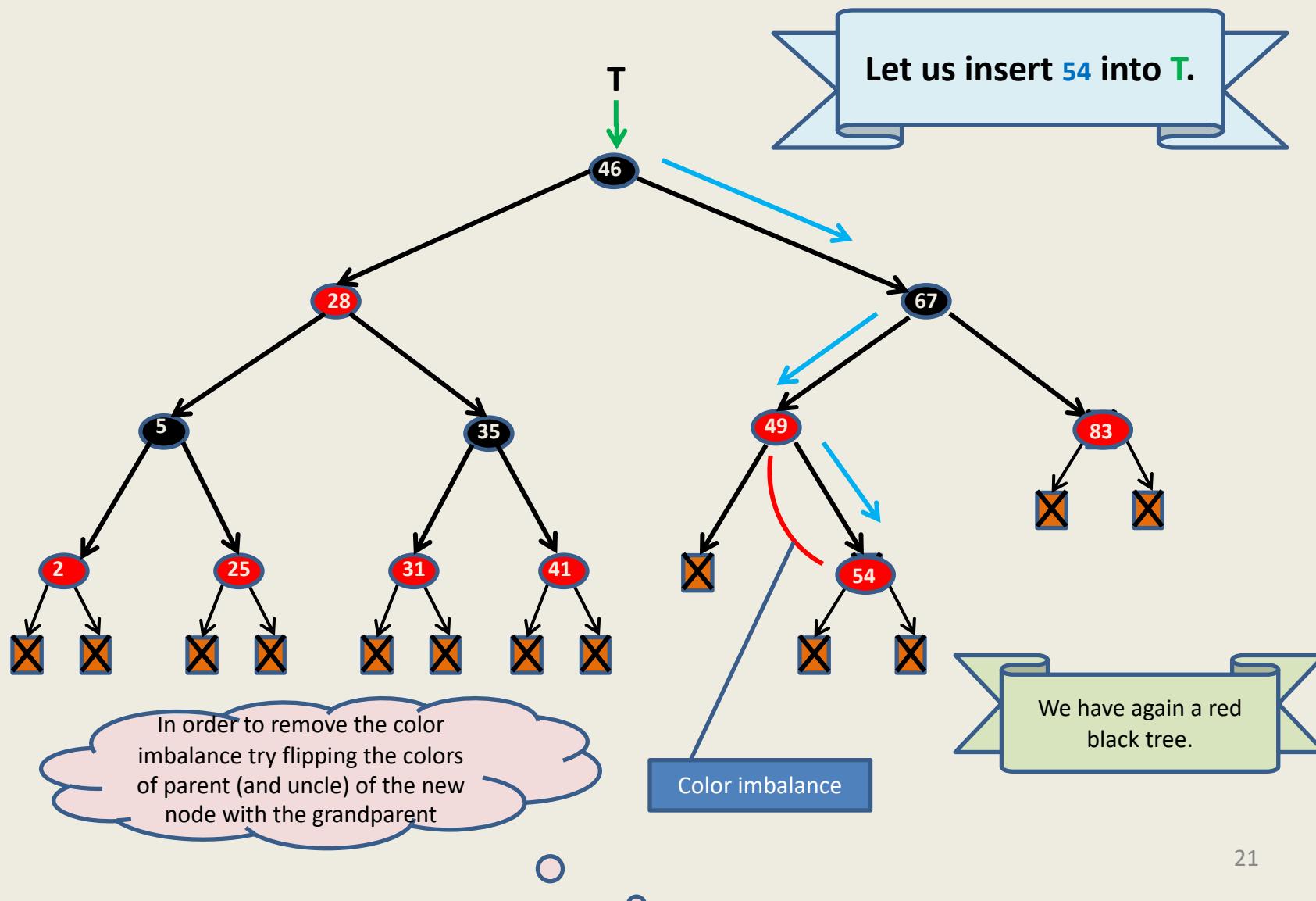
All it involves is

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

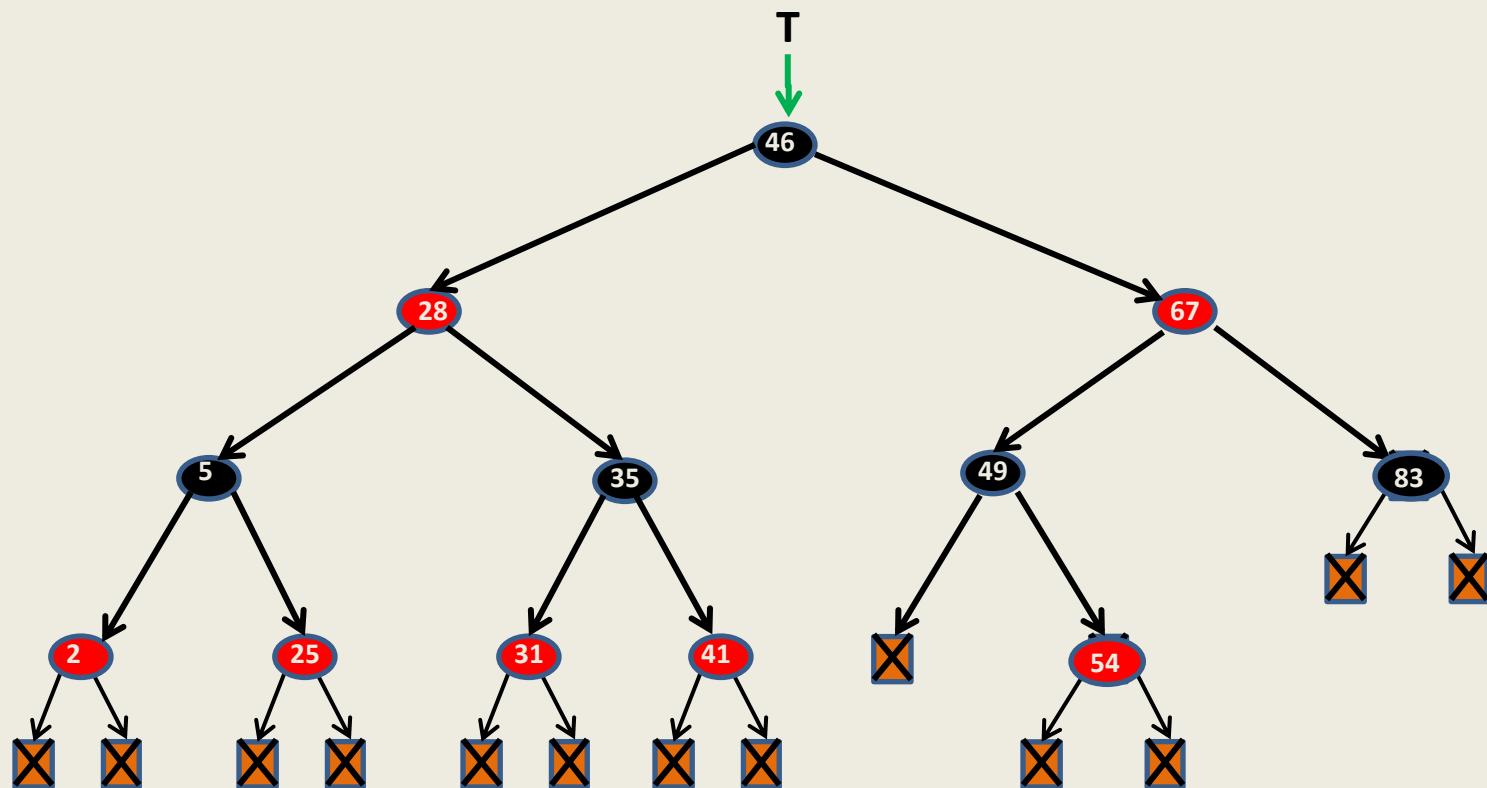
Insertion in a red-black tree



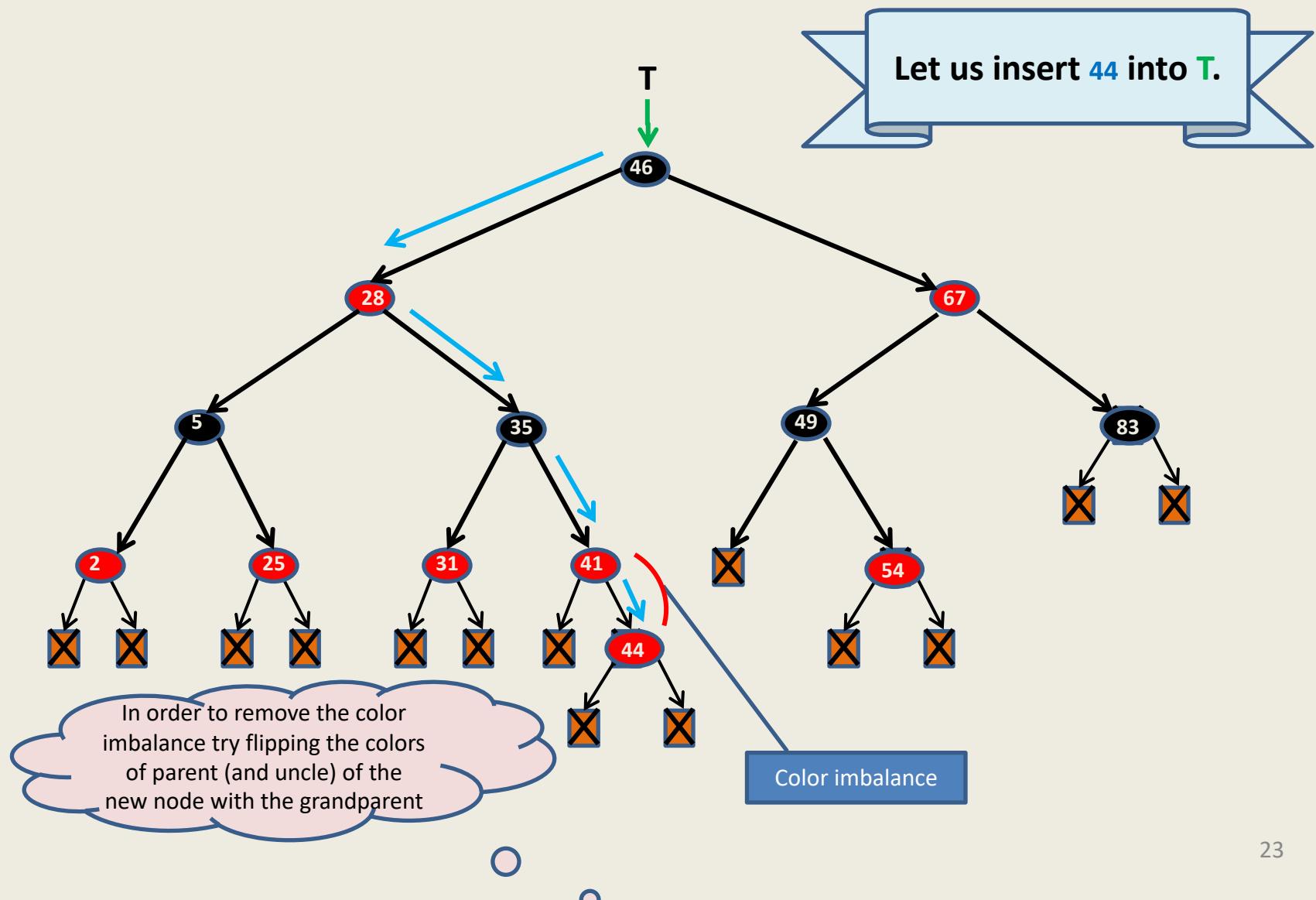
Insertion in a red-black tree



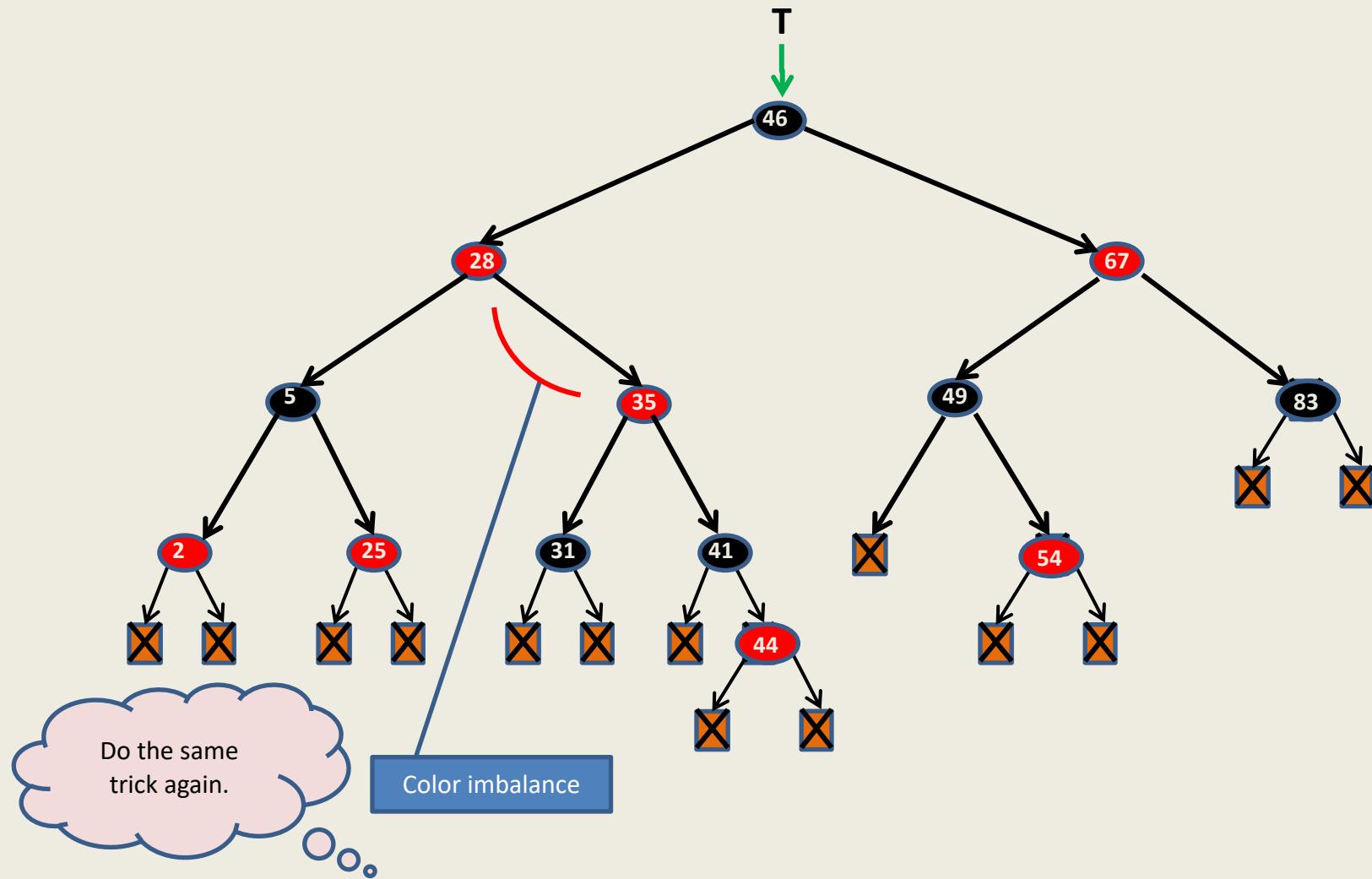
Insertion in a red-black tree



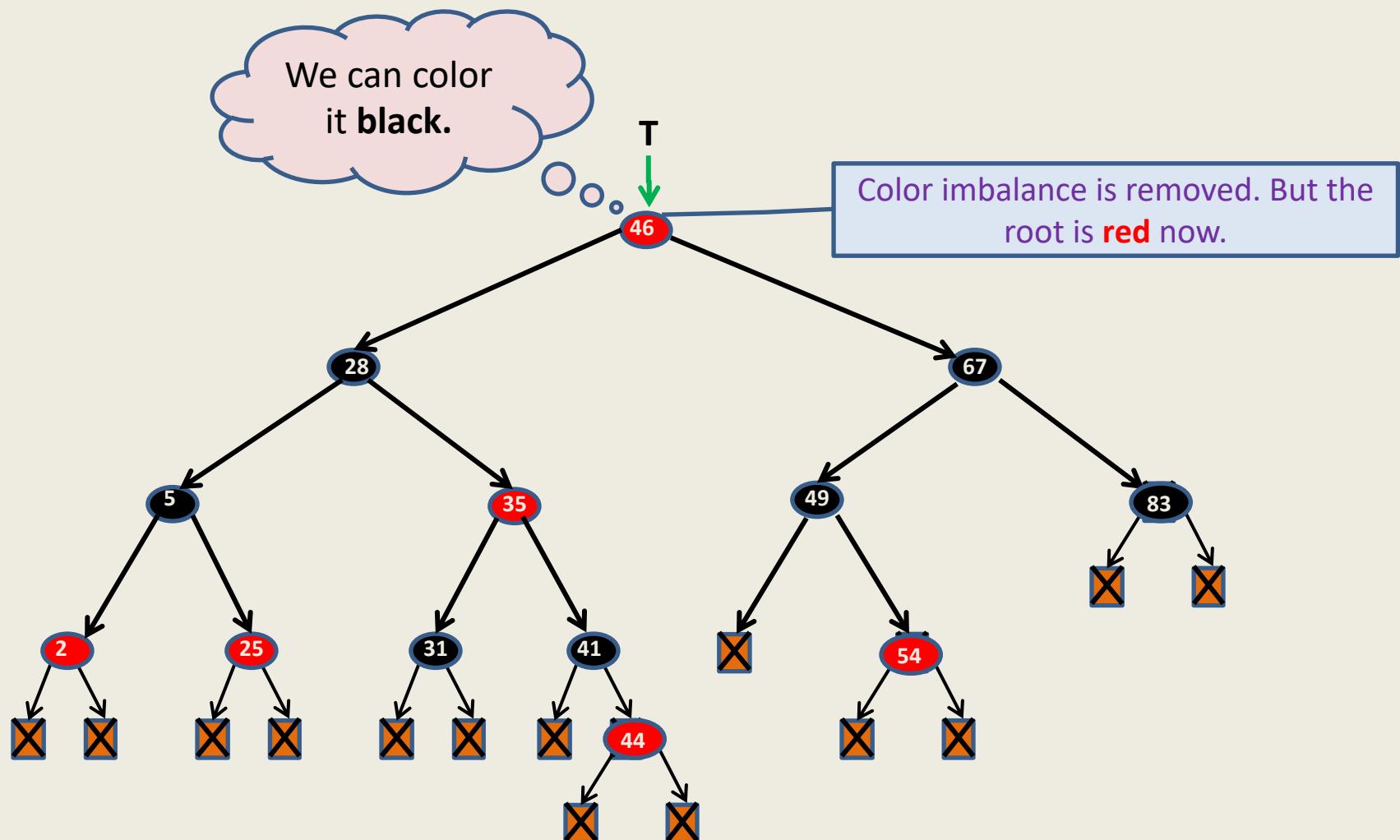
Insertion in a red-black tree



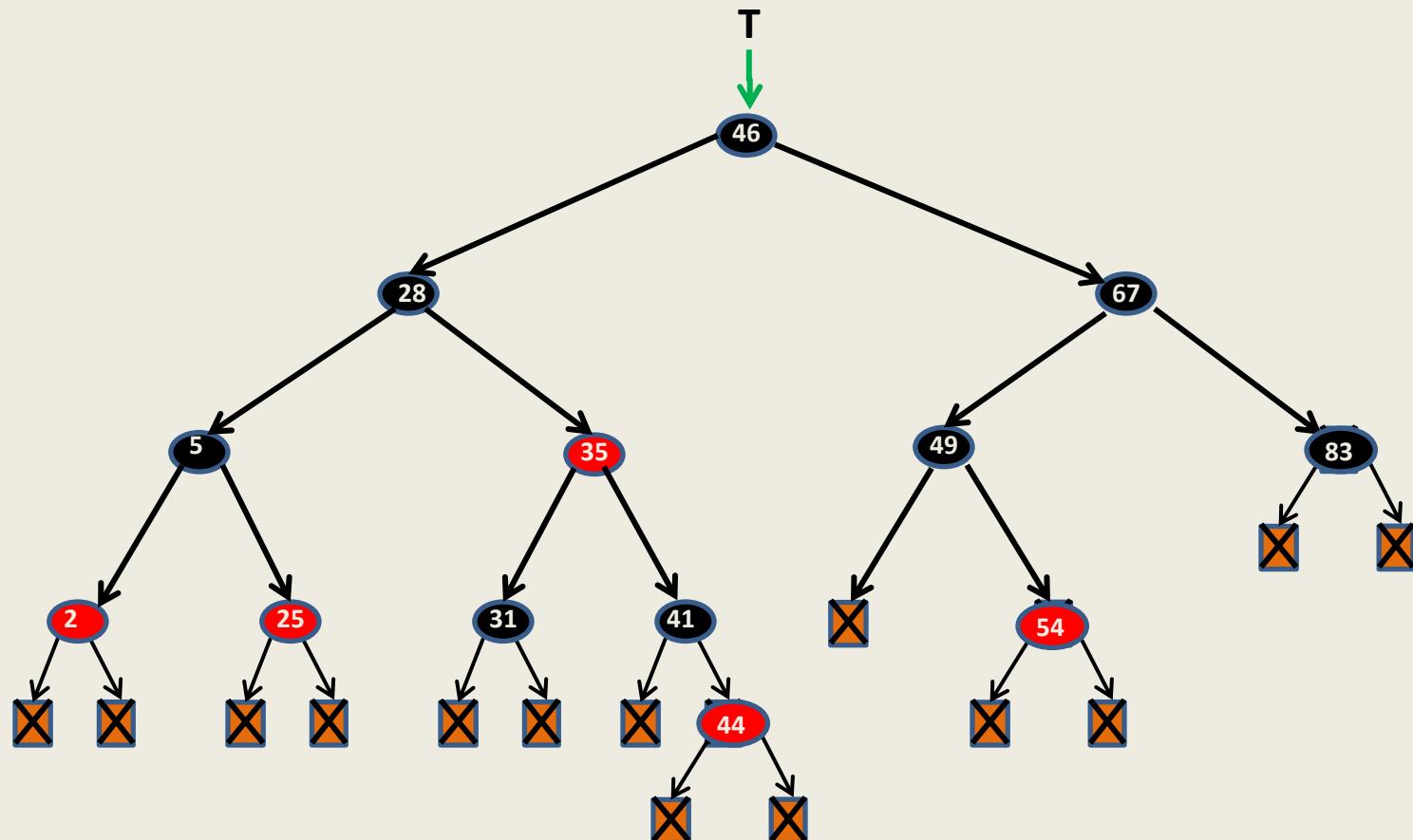
Insertion in a red-black tree



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

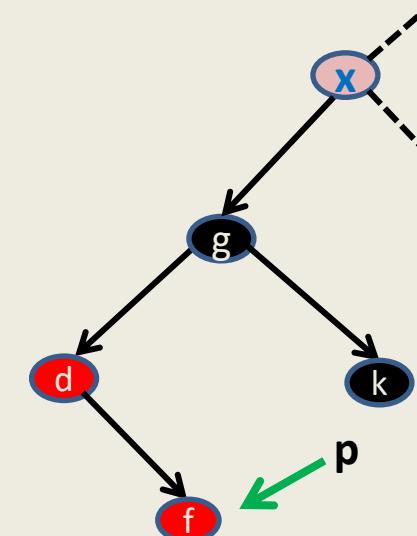
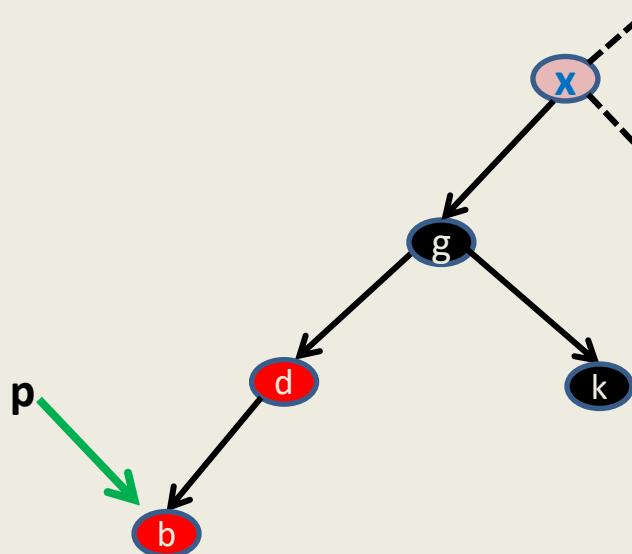
- p is a **red** colored node.
- $\text{parent}(p)$ is also **red**.
- $\text{uncle}(p)$ is **black**.

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



Can you transform
Case 3.2 to
Case 3.1 ?

Case 3.1:

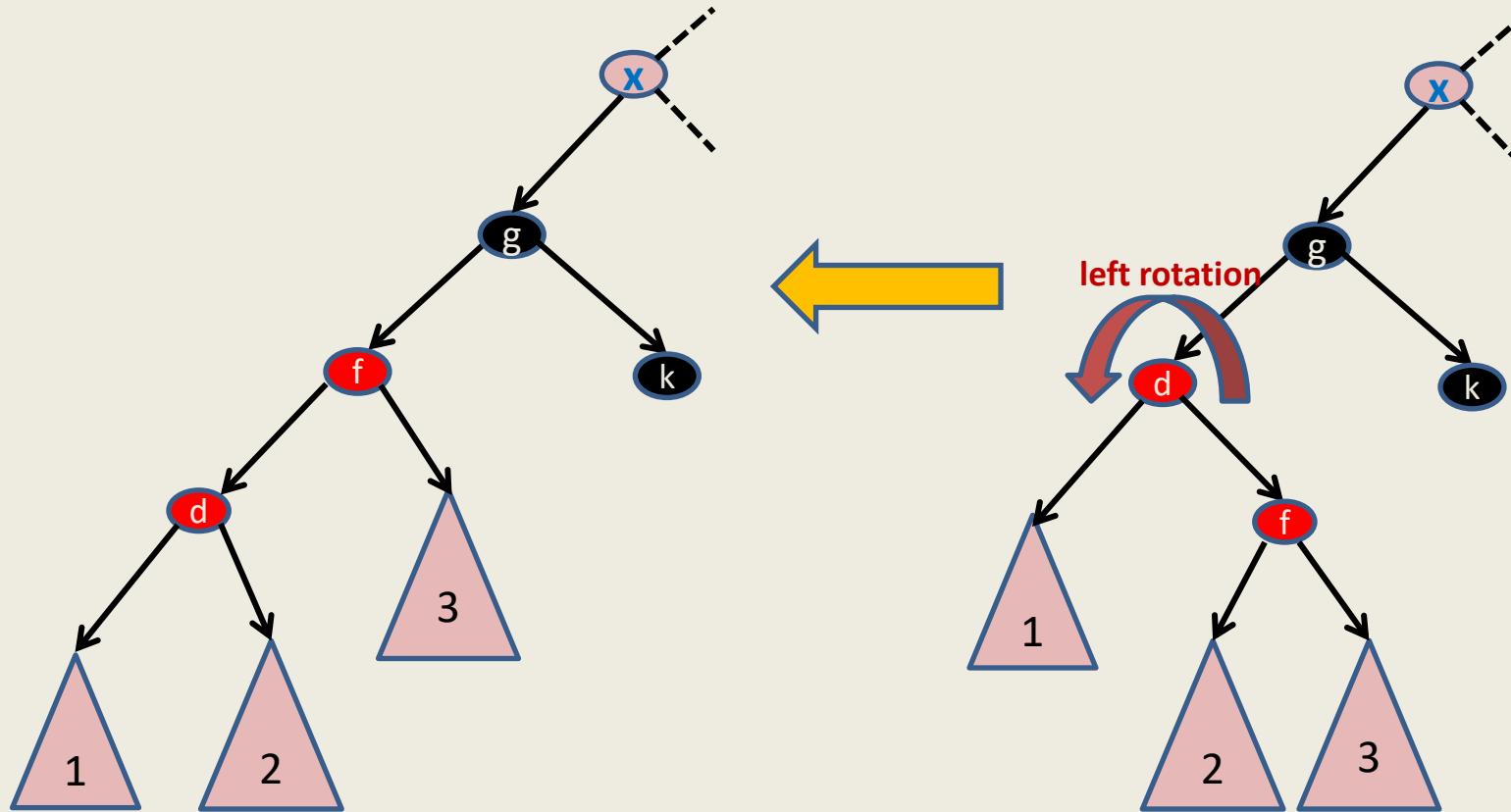
p is **left child** of its parent

Case 3.2:

p is **right child** of its parent

Handling the case 3

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

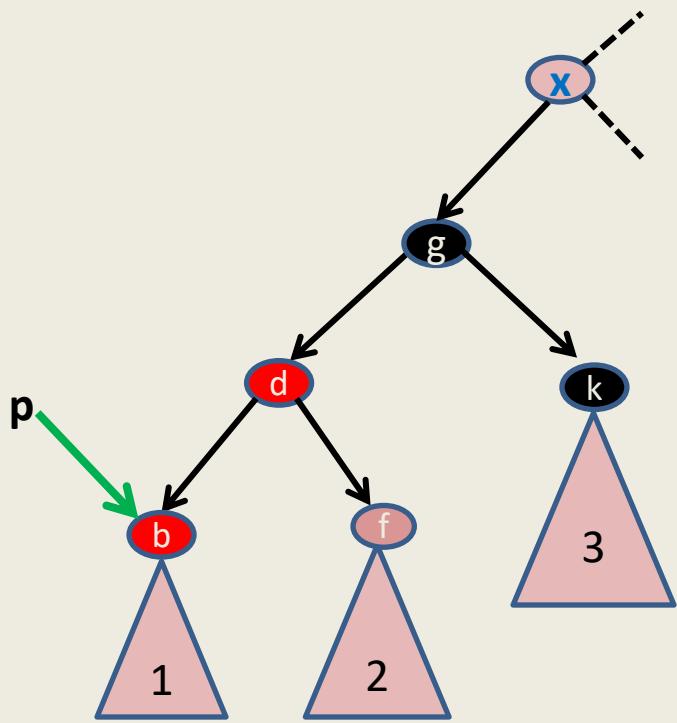


Vow!
This is exactly **Case 3.1**

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

We need to handle only case 3.1

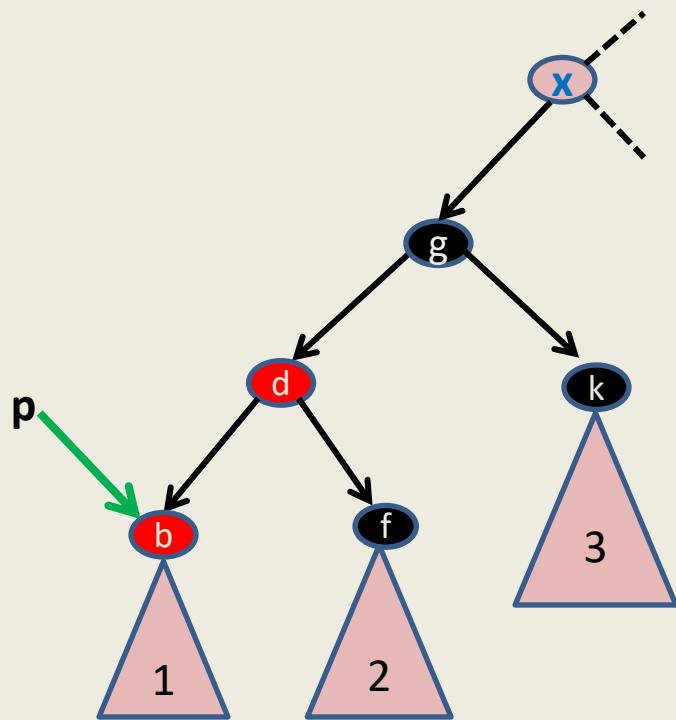
Handling the case 3.1



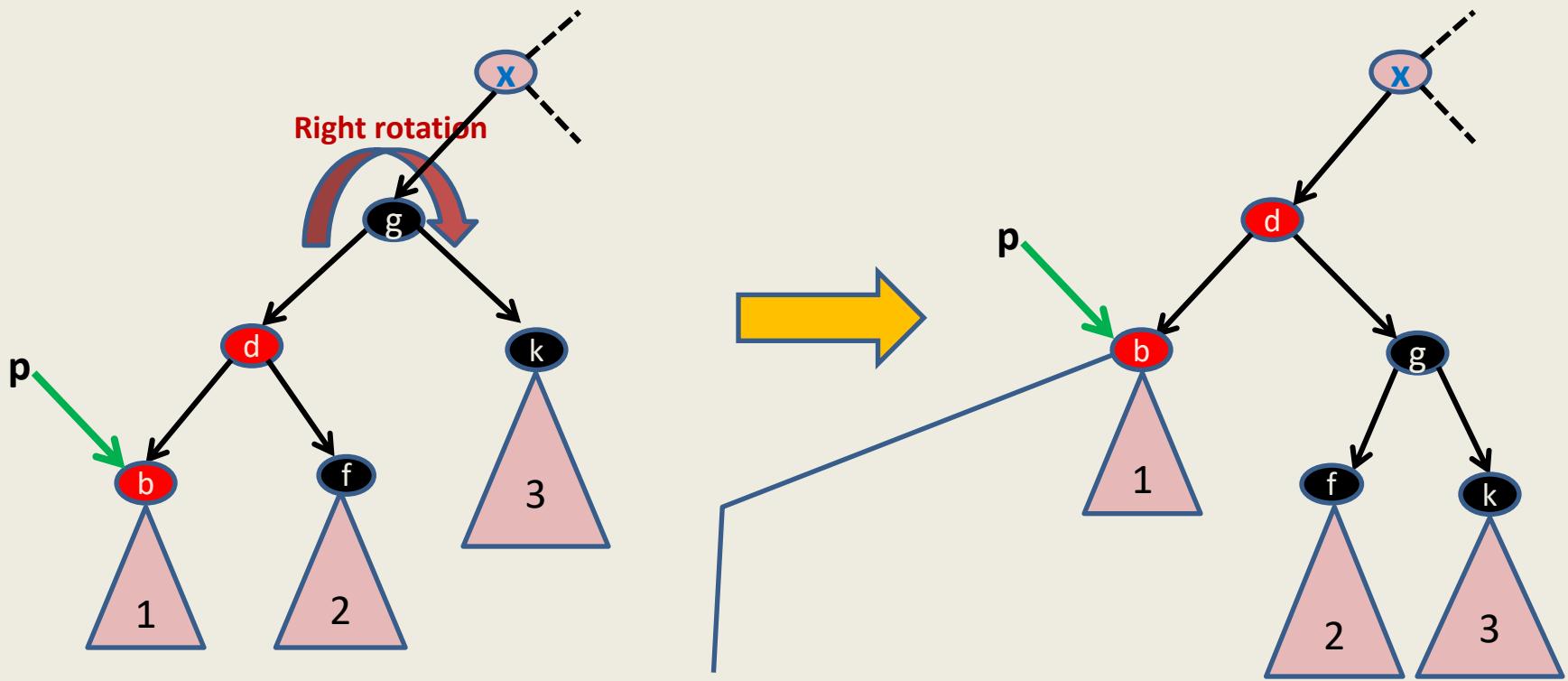
Can we say
anything about
color of node f ?

black

Handling the case 3.1



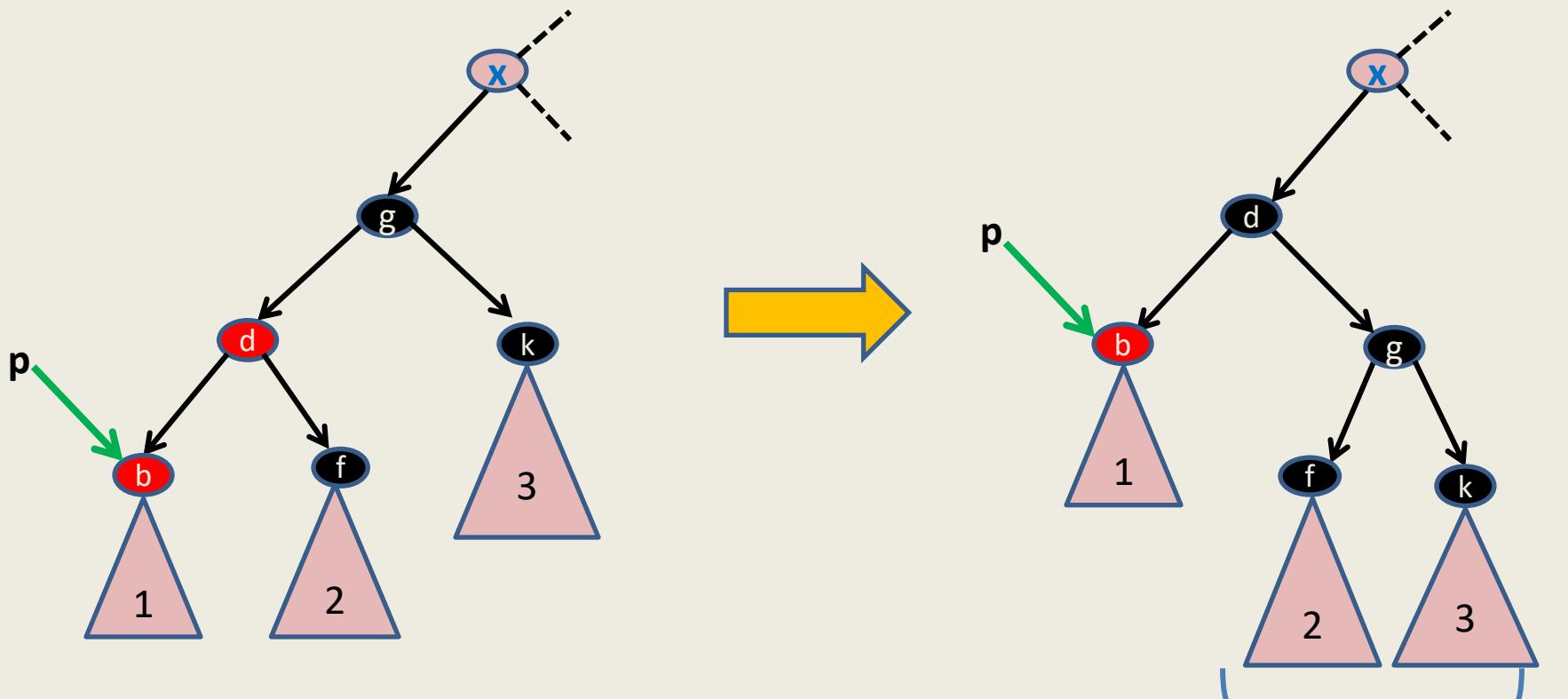
Handling the case 3.1



Now every node in tree 1 has one less **black** node on the path to root !
We must restore it. Moreover, the color imbalance exists even now.
What to do ?

Change color of
node d to **black**

Handling the case 3.1

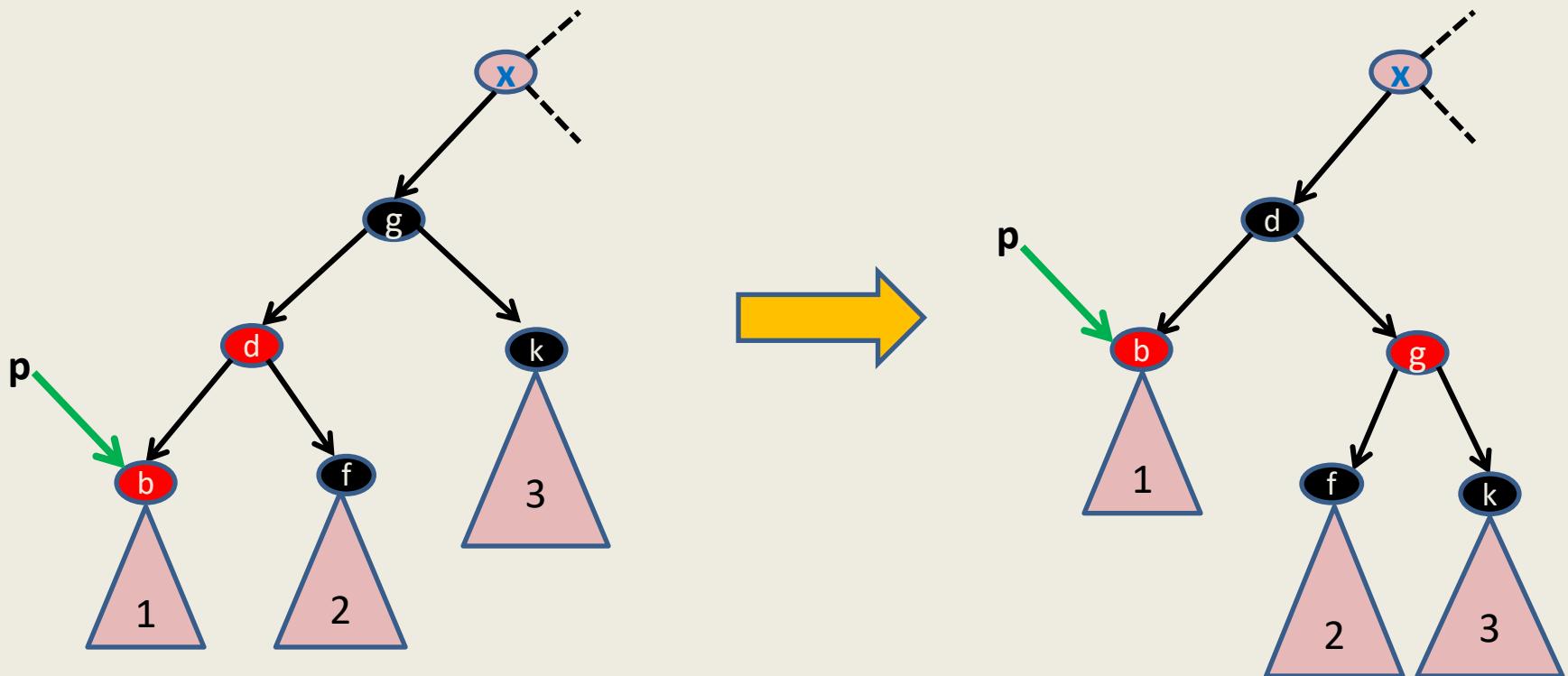


The number of **black** nodes on the path
restored for tree 1. Color imbalance is

But the number of **black** nodes on the path
increased by one for trees 2 and 3. What to do now ?

Color node g **red**

Handling the case 3.1



The black height is
restored for all trees.
This completes **Case 3.1**

Theorem:

We can maintain **red-black** trees under insertion of nodes in $\mathbf{O}(\log n)$ 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.

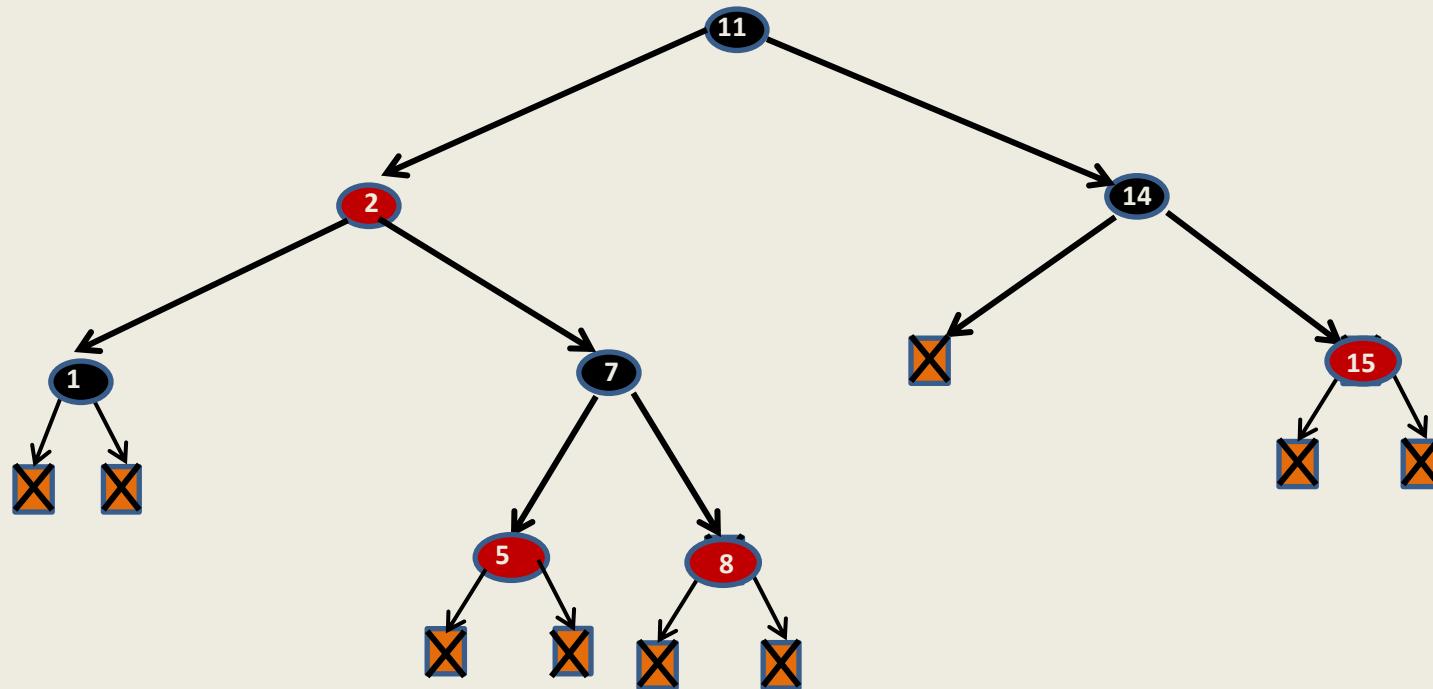
- Why we are handling insertions in “this *particular way*” ?
- Are there *alternative and simpler* ways to handle insertions ?

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 ?

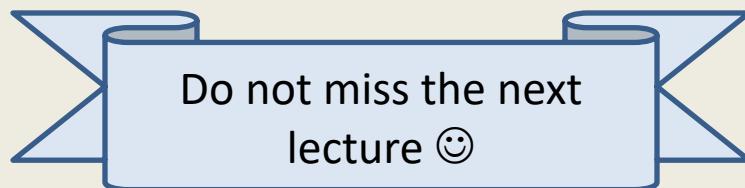


How do will we handle deletion ?

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

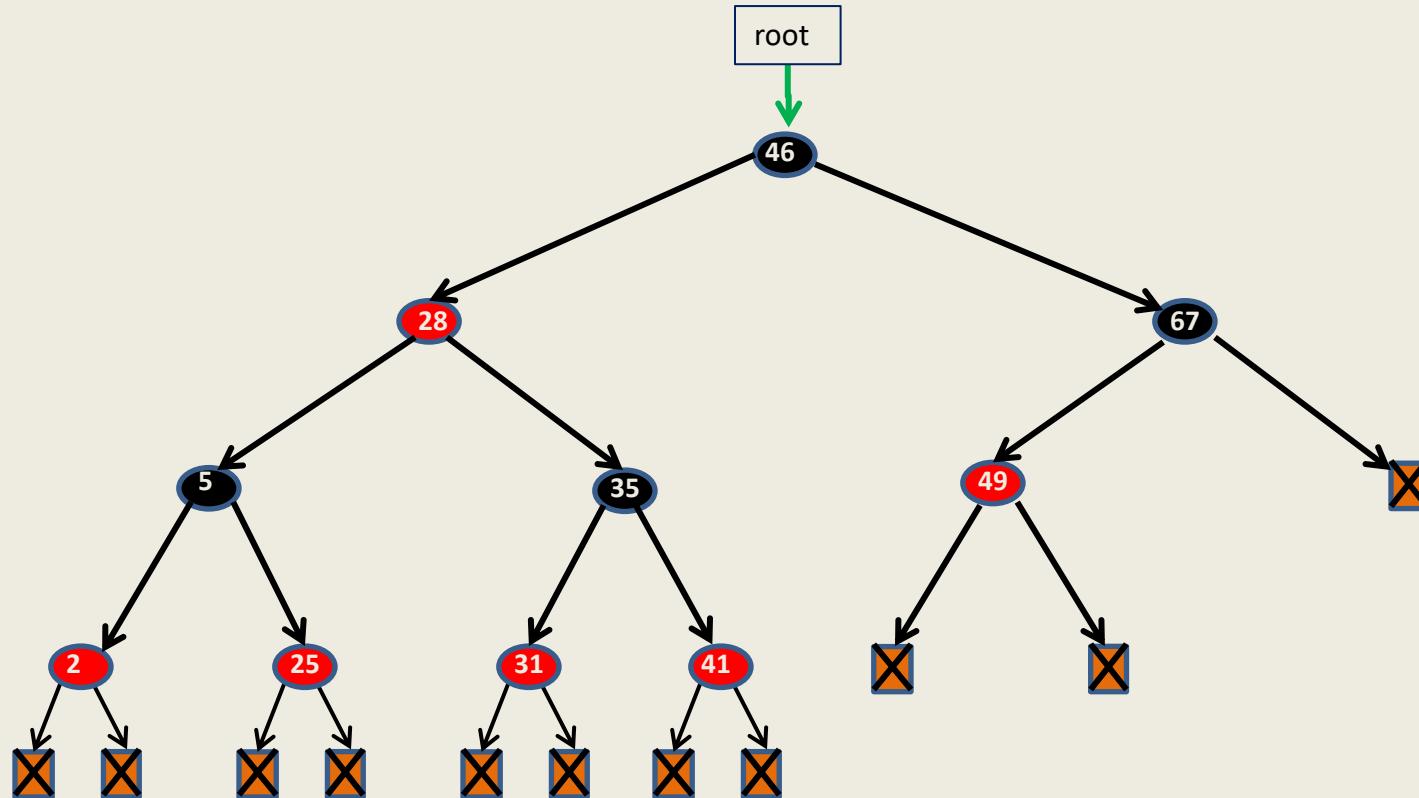
Red Black tree:

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



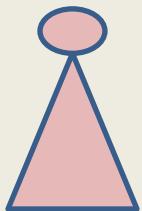
a **black** node



a **red** node



a node whose color is not specified



a BST

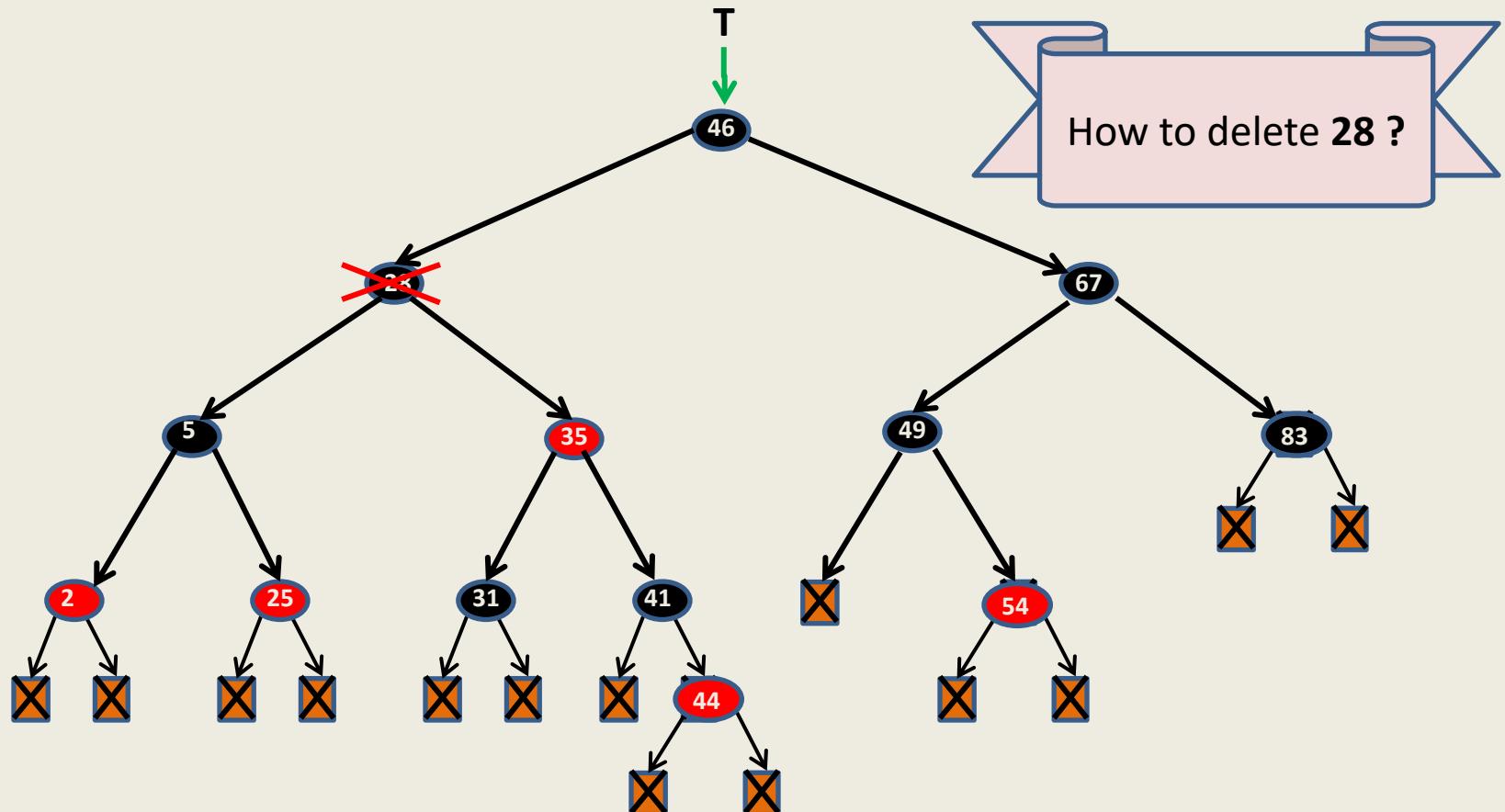


Could potentially be

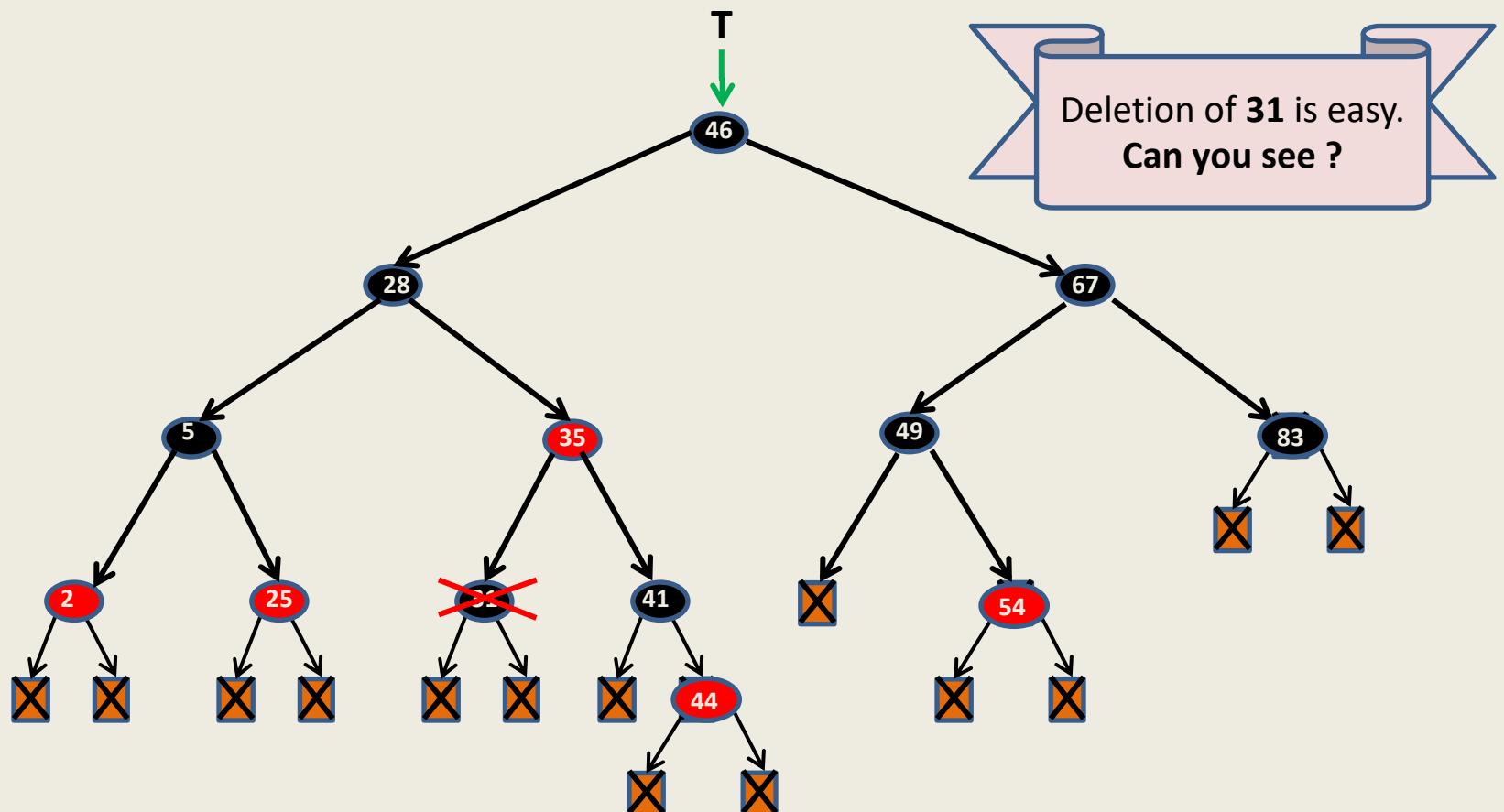


Deletion in a BST is **slightly harder than Insertion**

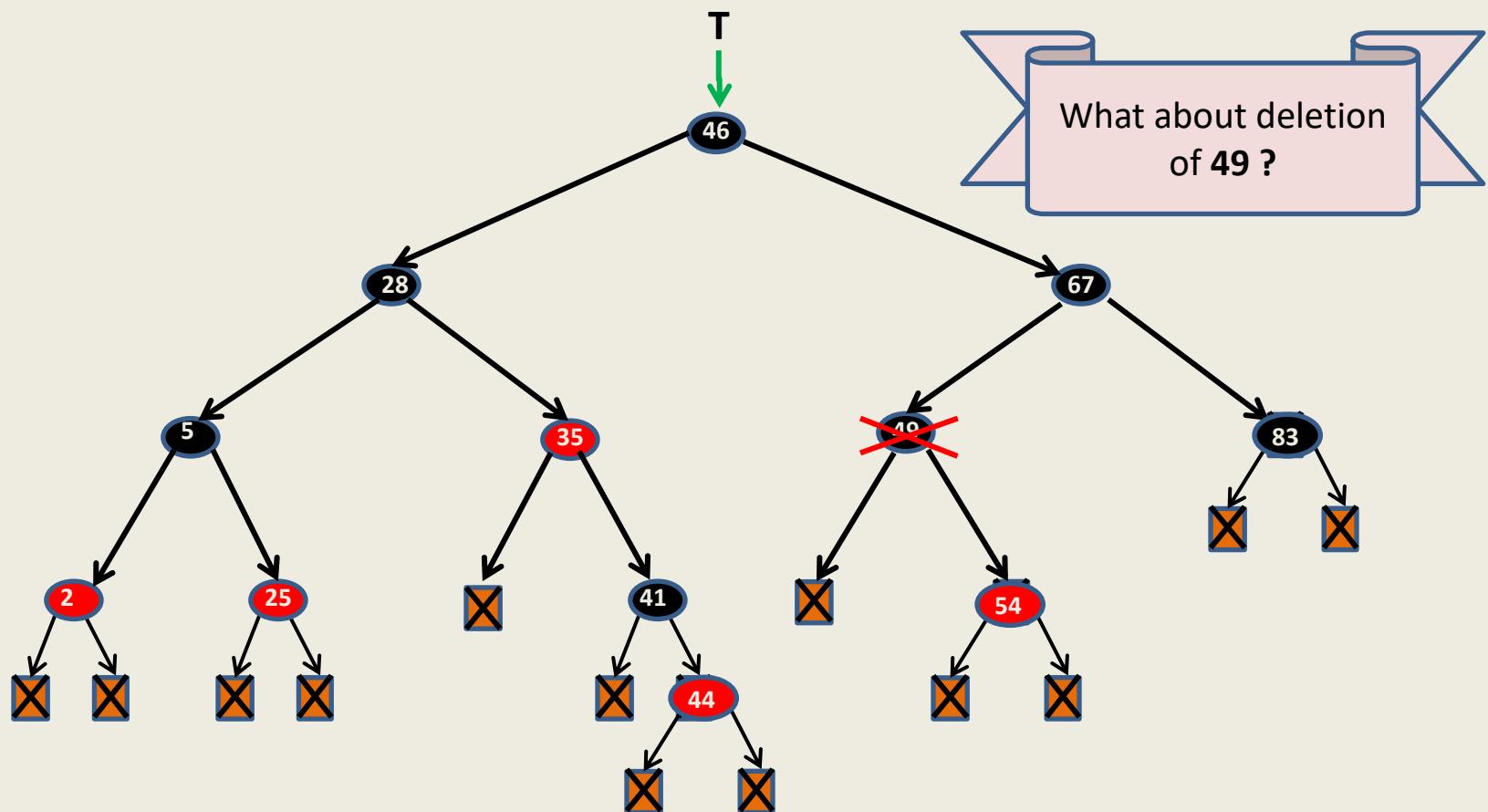
(even if we ignore the **height** factor)



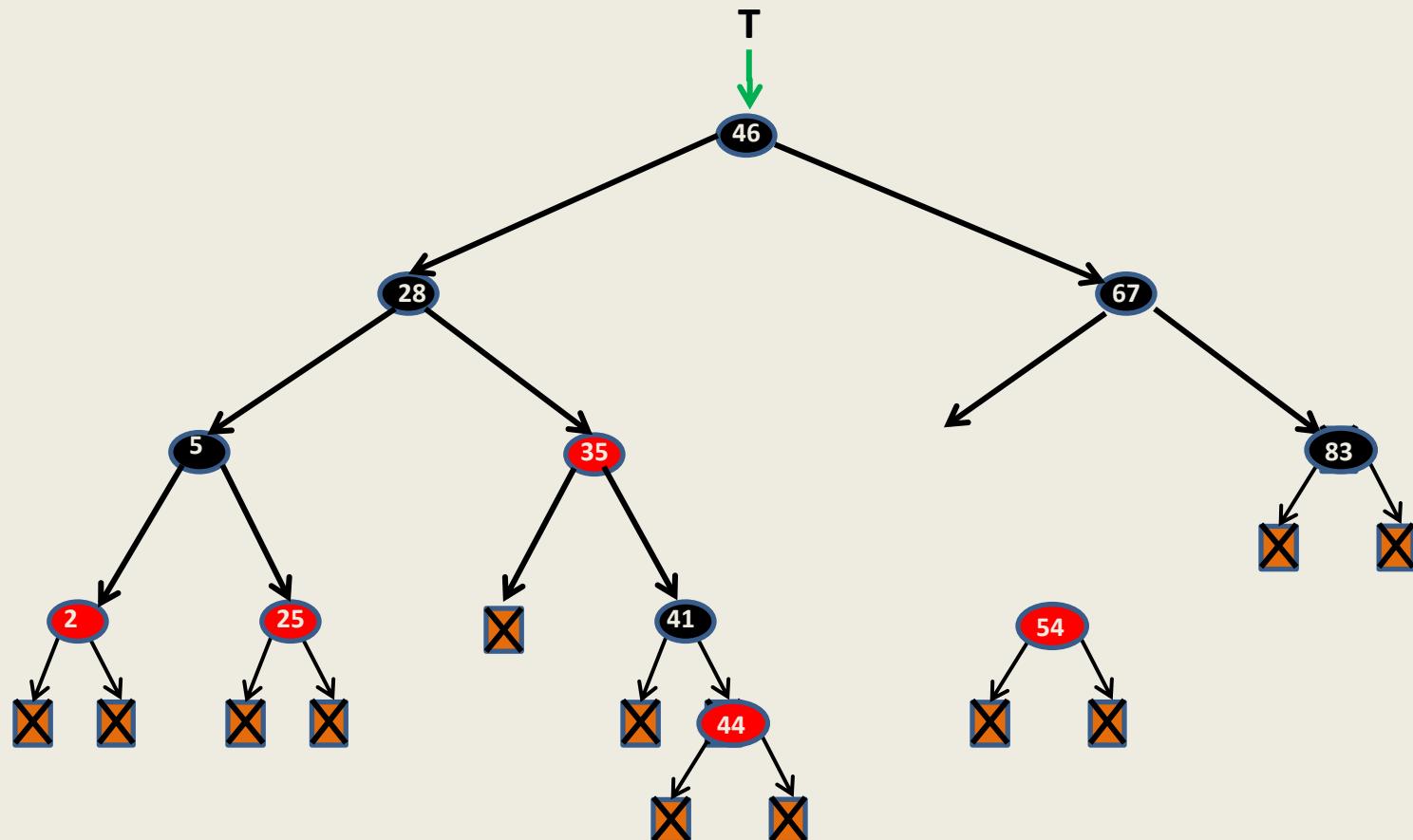
Is deletion of a node easier for some cases ?



Is deletion of a node easier for some cases ?

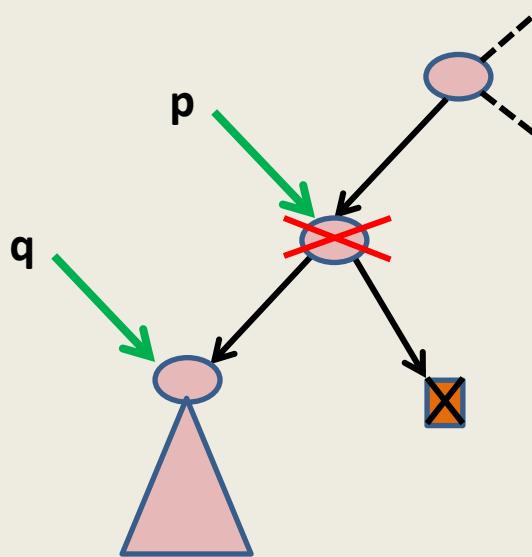


Is deletion of a node easier for some cases ?



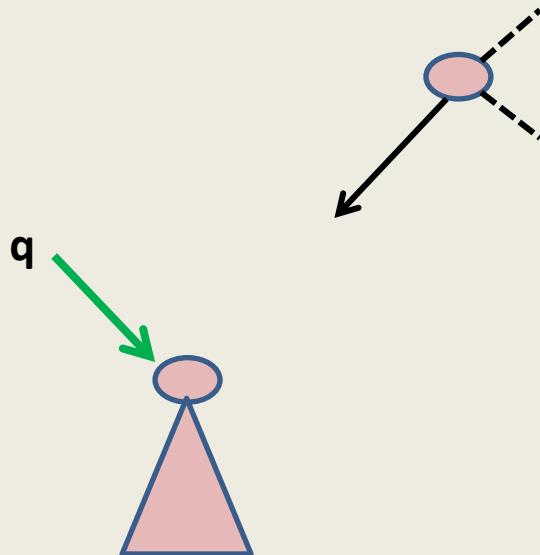
An insight

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

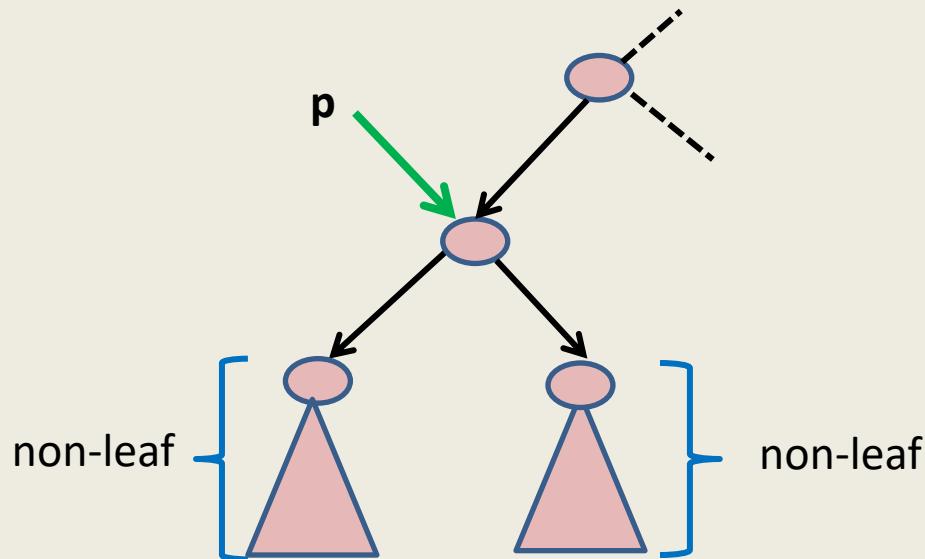
It is easier to maintain a BST under deletion if the node to be deleted has at most one child which is non-leaf.



An important question

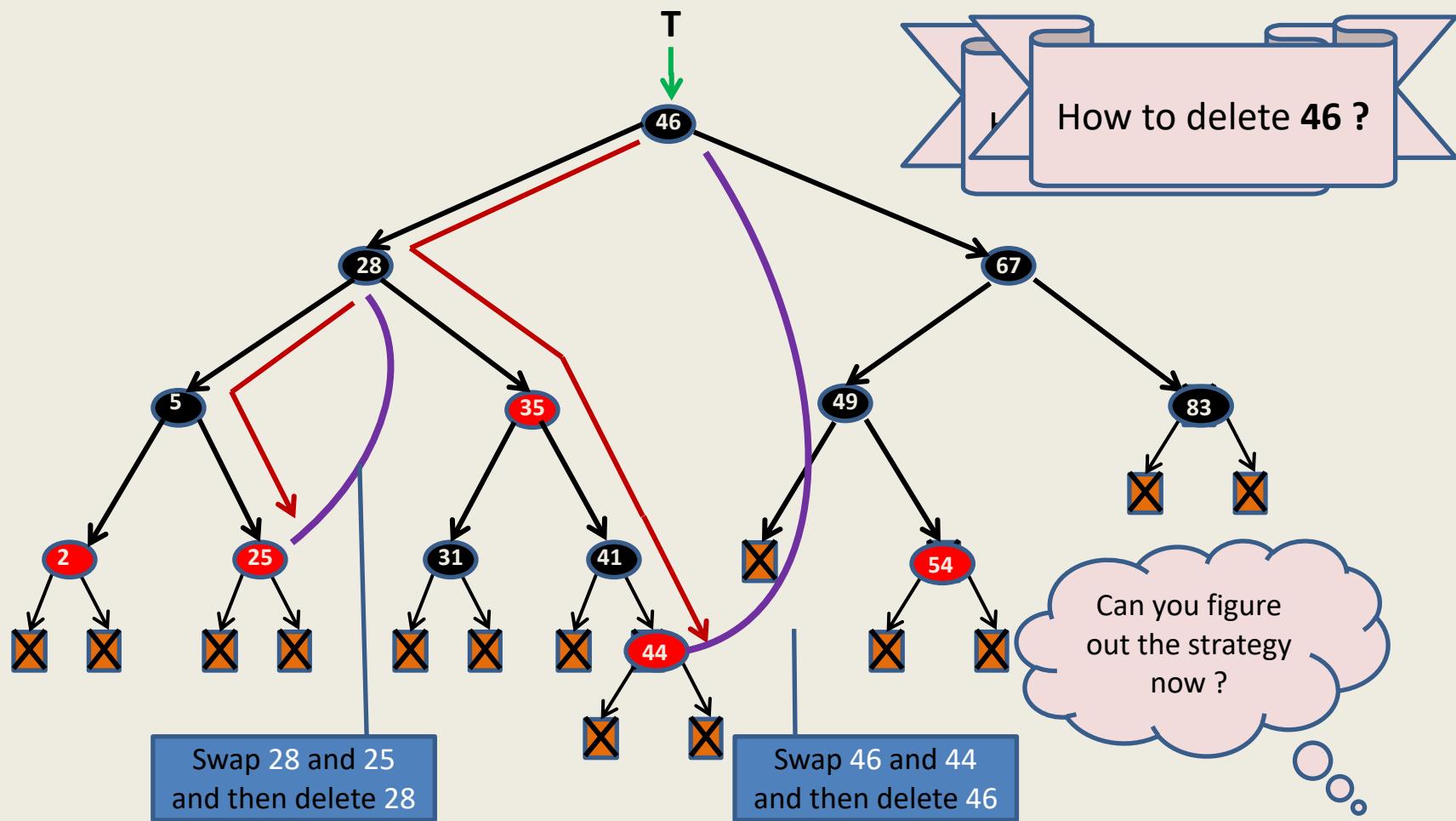
It is easier to maintain a BST under deletion if the node to be deleted has **at most** one child which is **non-leaf**.

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



Answer: ??

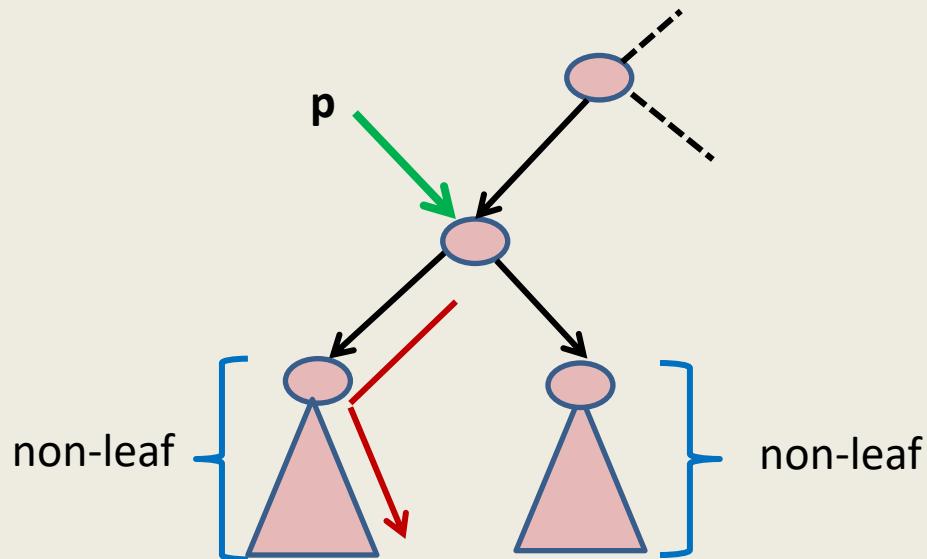
How to delete a node whose both children are non-leaves?



An important observation

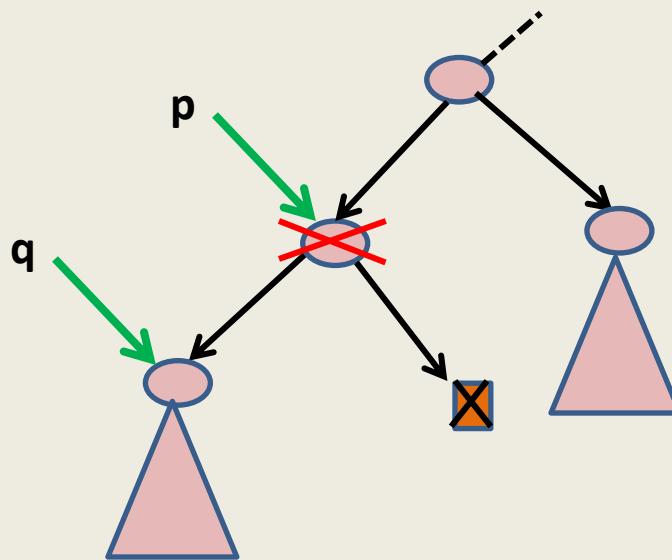
It is easier to maintain a BST under deletion if the node to be deleted has **at most** one child which is **non-leaf**.

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.

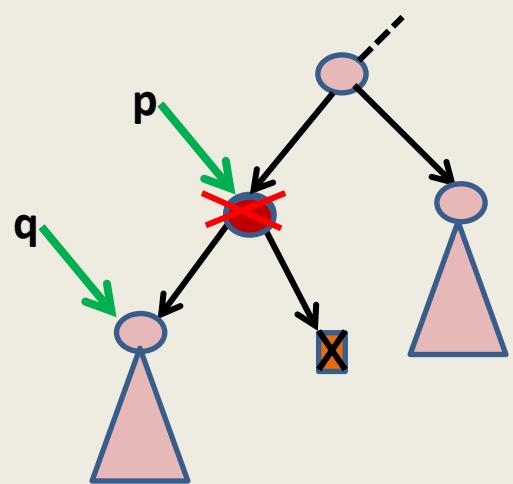
We need to handle deletion only for the following case



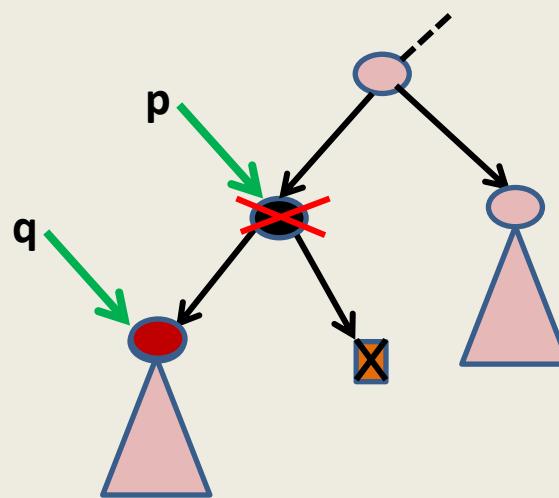
How to maintain a **red-black** tree under deletion ?

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

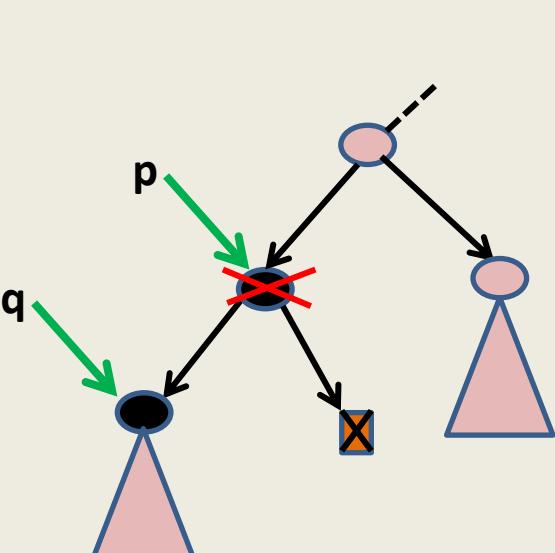
Easy cases and difficult case



Easy case

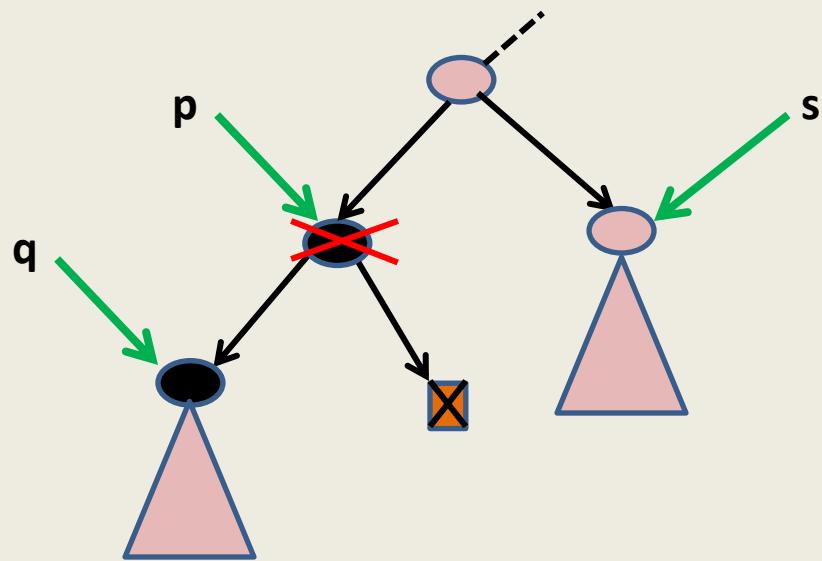


Easy case:
Change color of q to
black

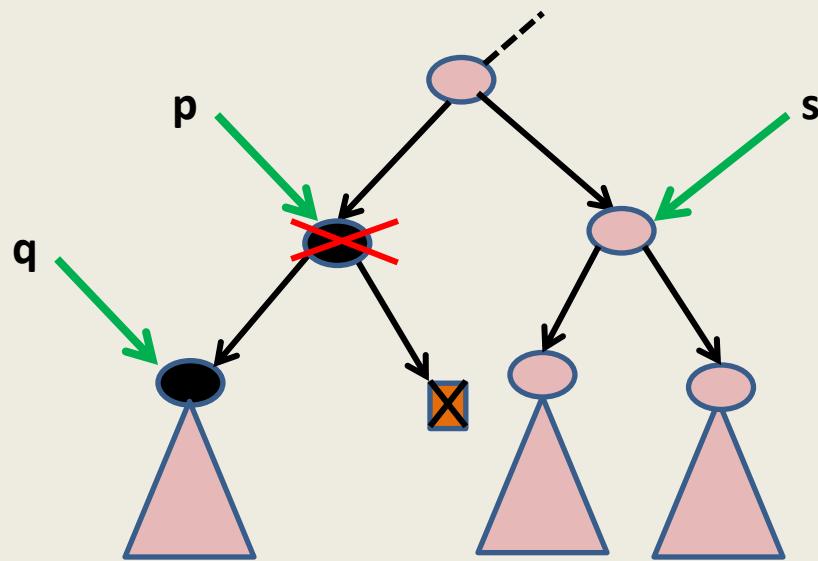


Difficult case

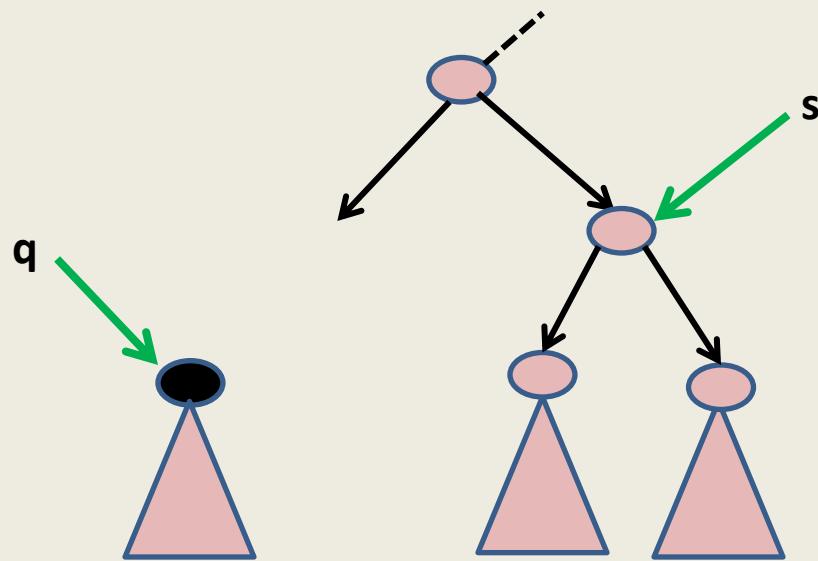
Handling the difficult case



Handling the difficult case

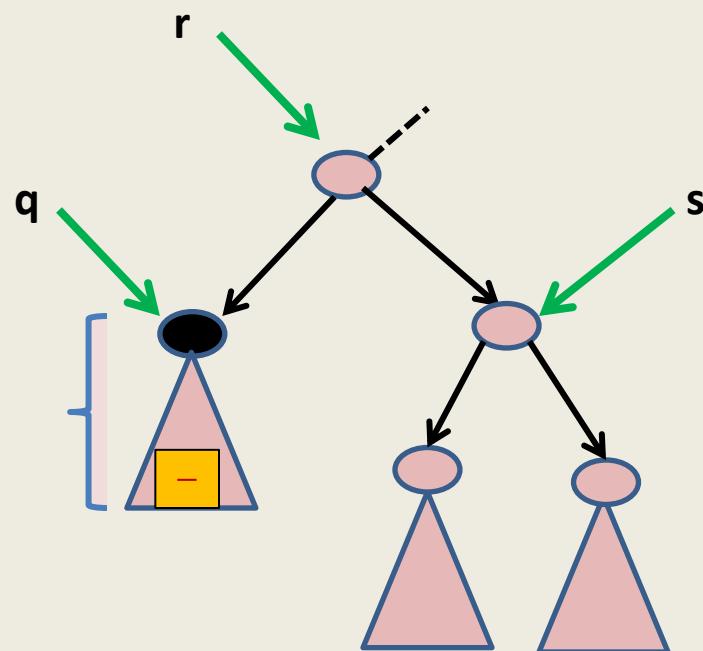


Handling the difficult case

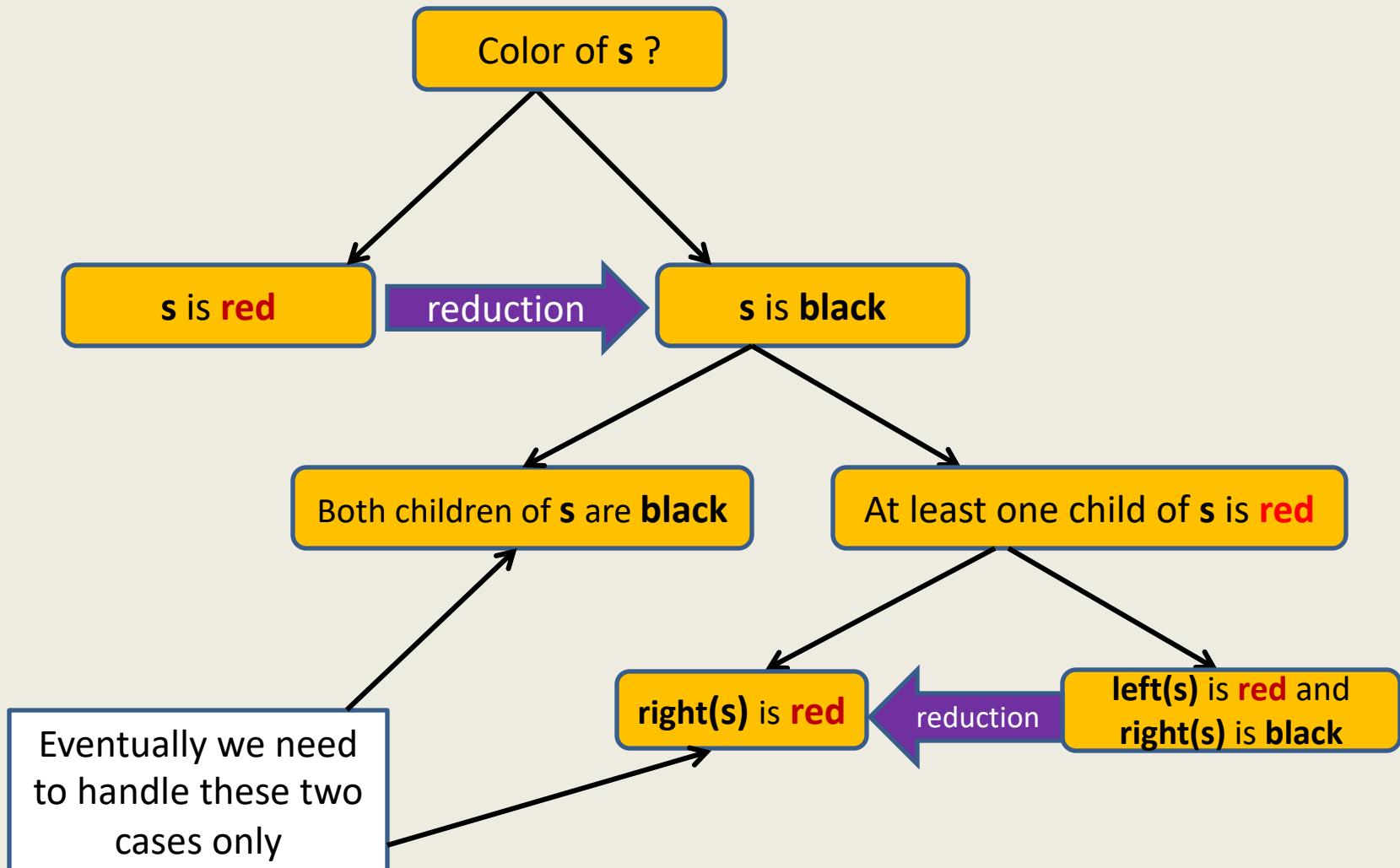


Handling the difficult case

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**.



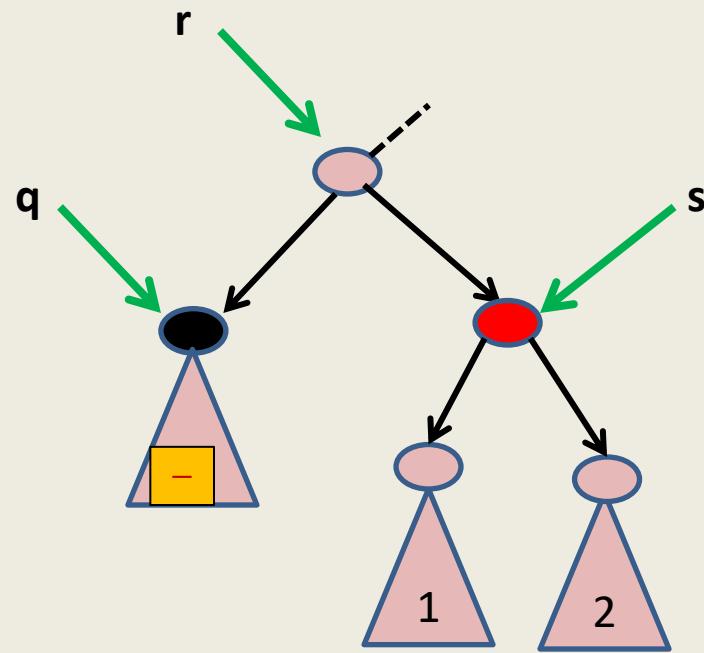
Handling the difficult case: An overview



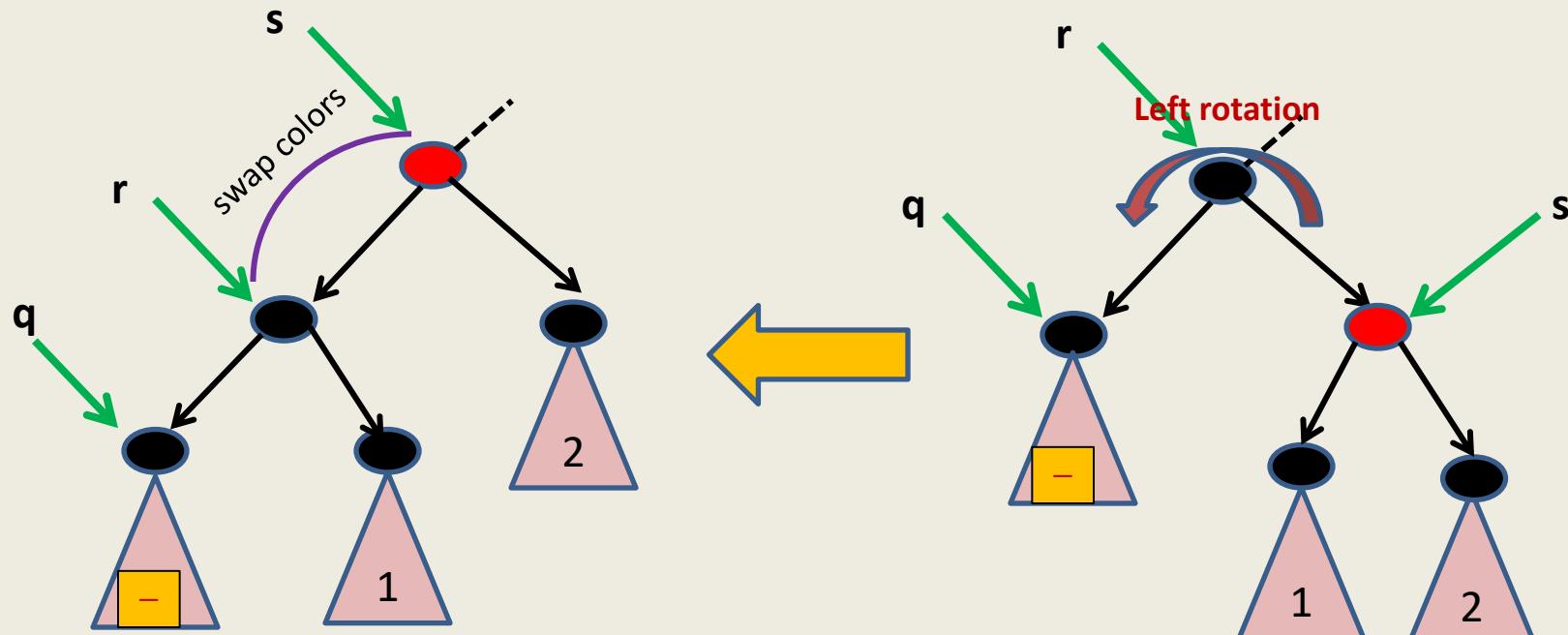
“s is red”  reduction “s is black”

“s is red” → reduction “s is black”

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

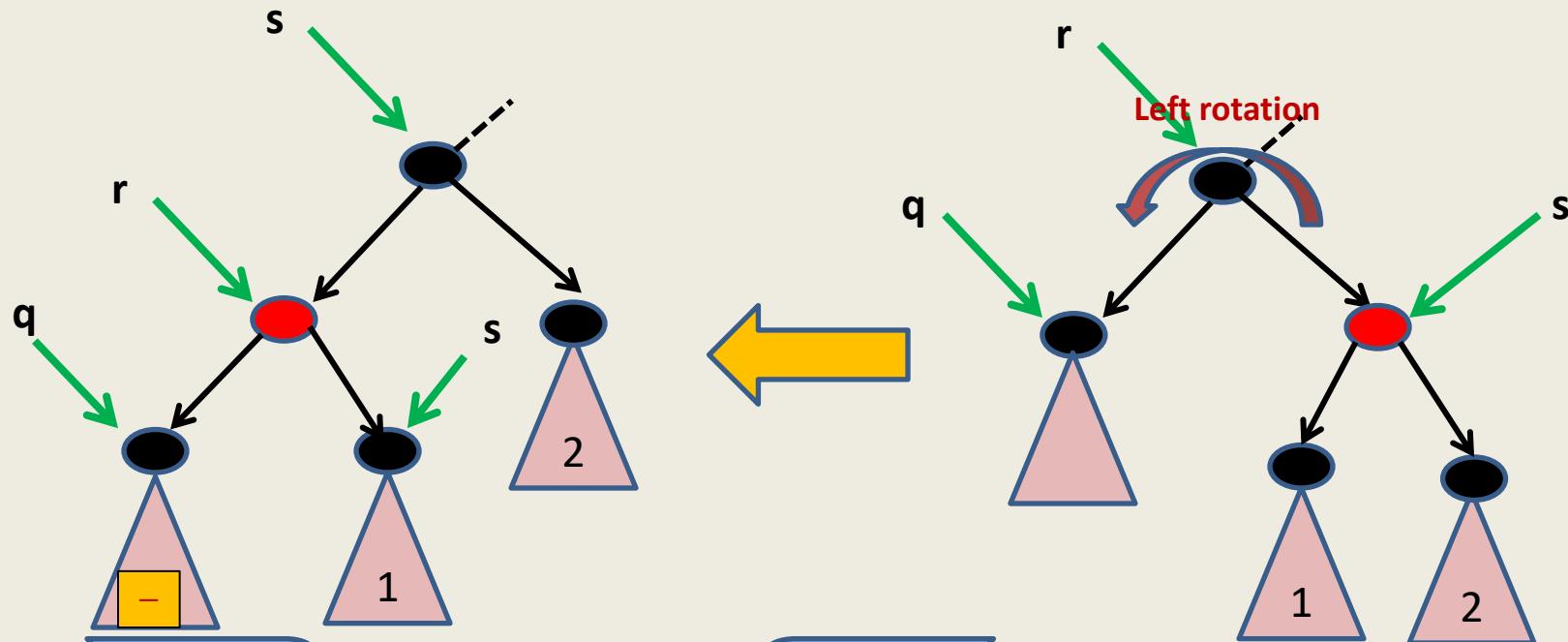


“s is red” → reduction → “s is black”



The new sibling of q is now a black node. But the number of black nodes to leaves of tree 2 have reduced by one. What to do ?

“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.

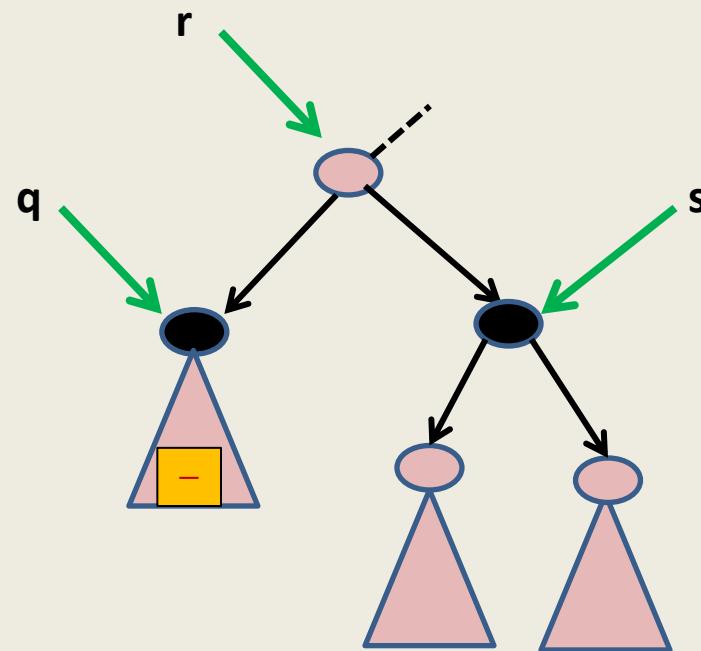
We just need to handle the case

“s is black”

Handling the case: s is black

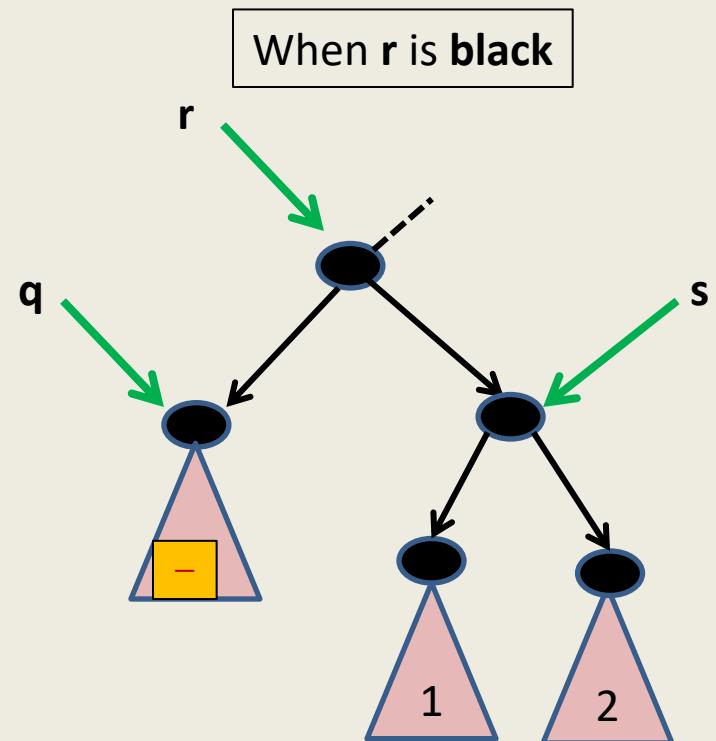
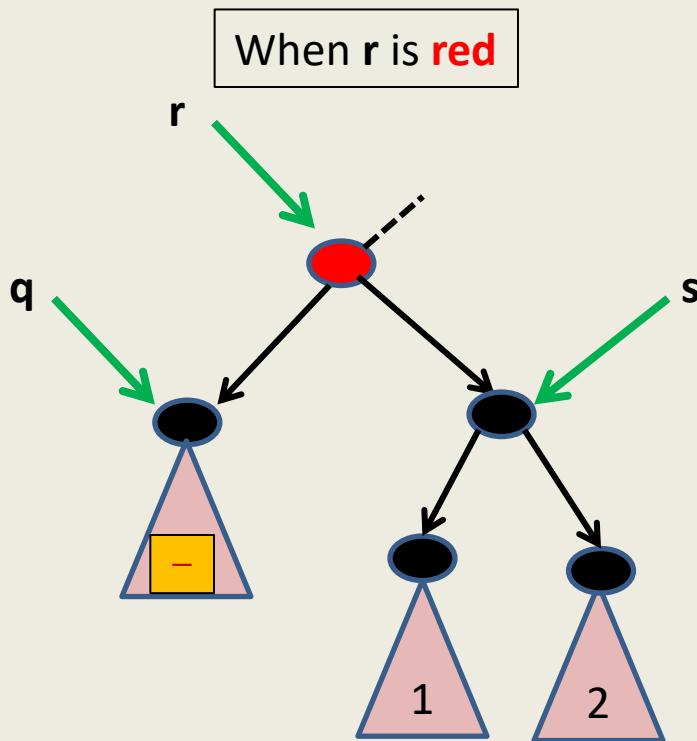
Case 1: both children of s are **black**

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



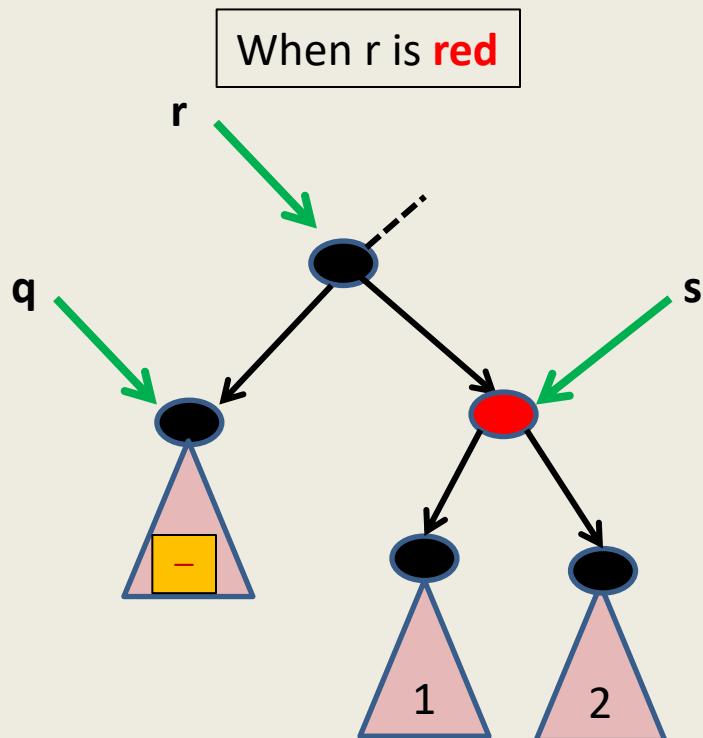
Handling the case:
s is black and both children of s are black

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

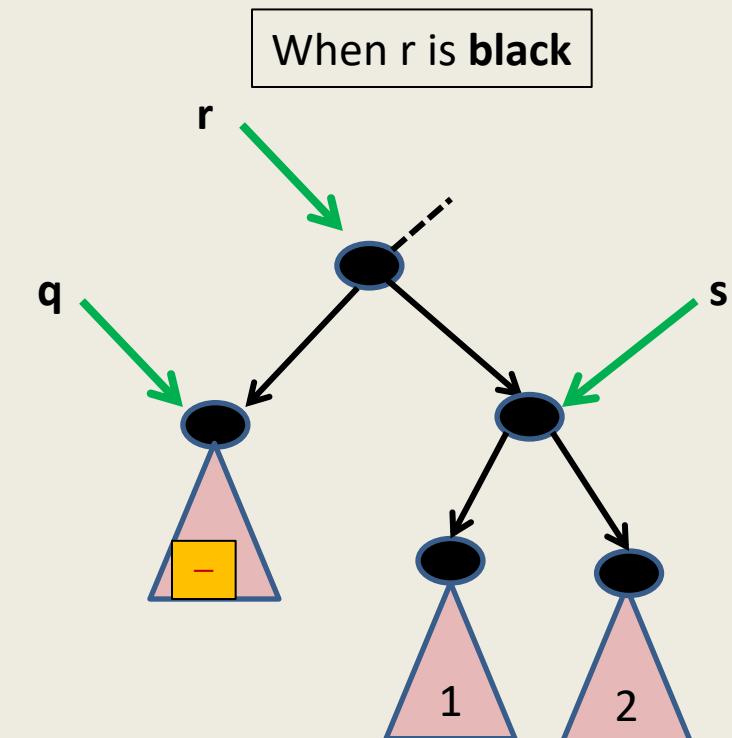


How to handle this case ?

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

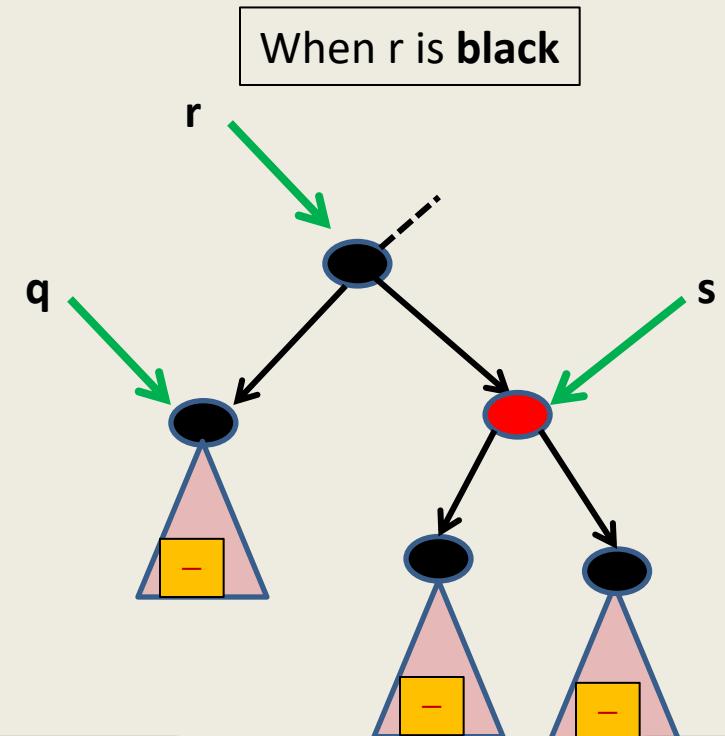
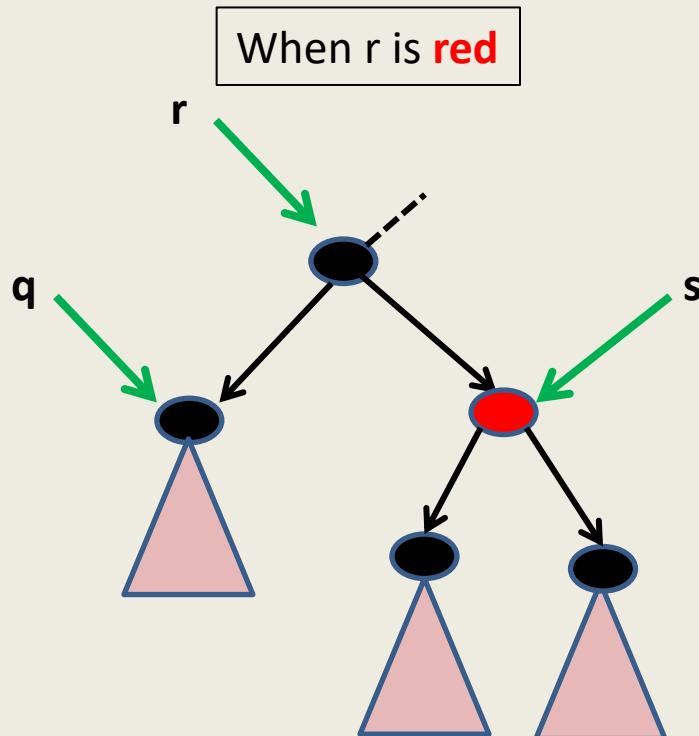


YES.
As a result of swapping the colors, the number of black nodes to the leaves of trees 1 and 2 unchanged. Interestingly, the deficiency of one black node on the path to the leaves of subtree(q) is also compensated. So we are done😊



How to handle this case ?

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

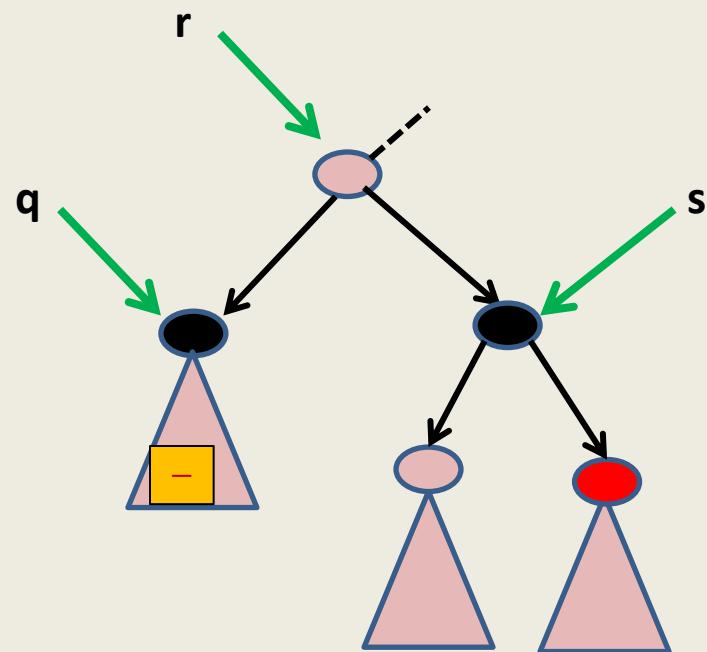


Changing color of s to **red** has reduced the number of black nodes on the path to the root of subtree(s) by one. As a result the imbalance of black height has *propagated* upward. So we process the new q.

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

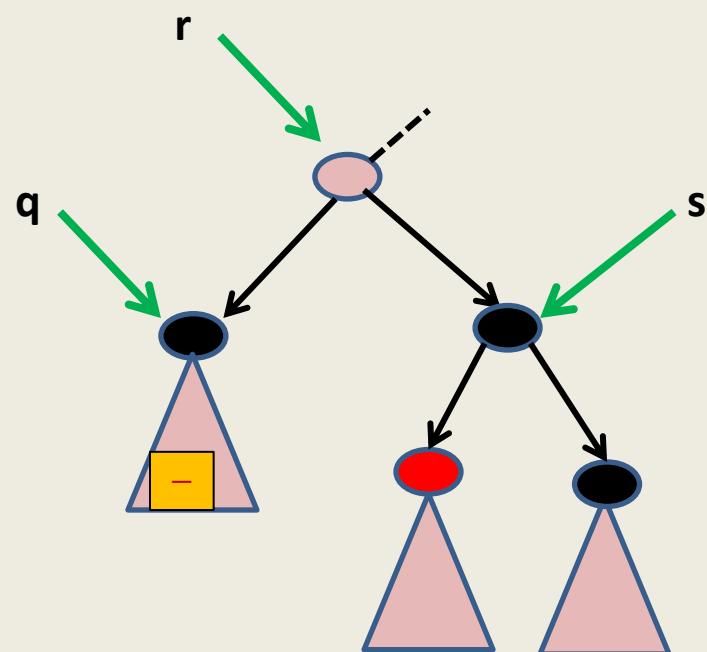
There are two cases

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



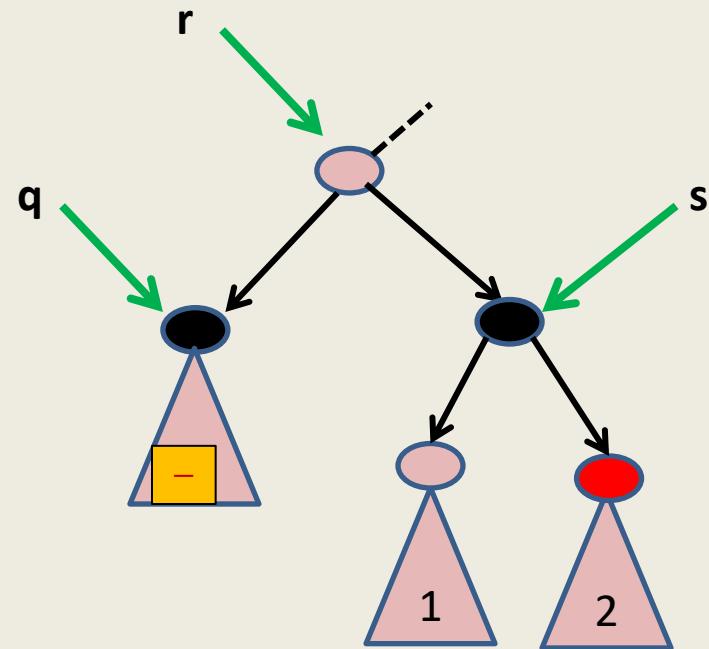
reduction

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



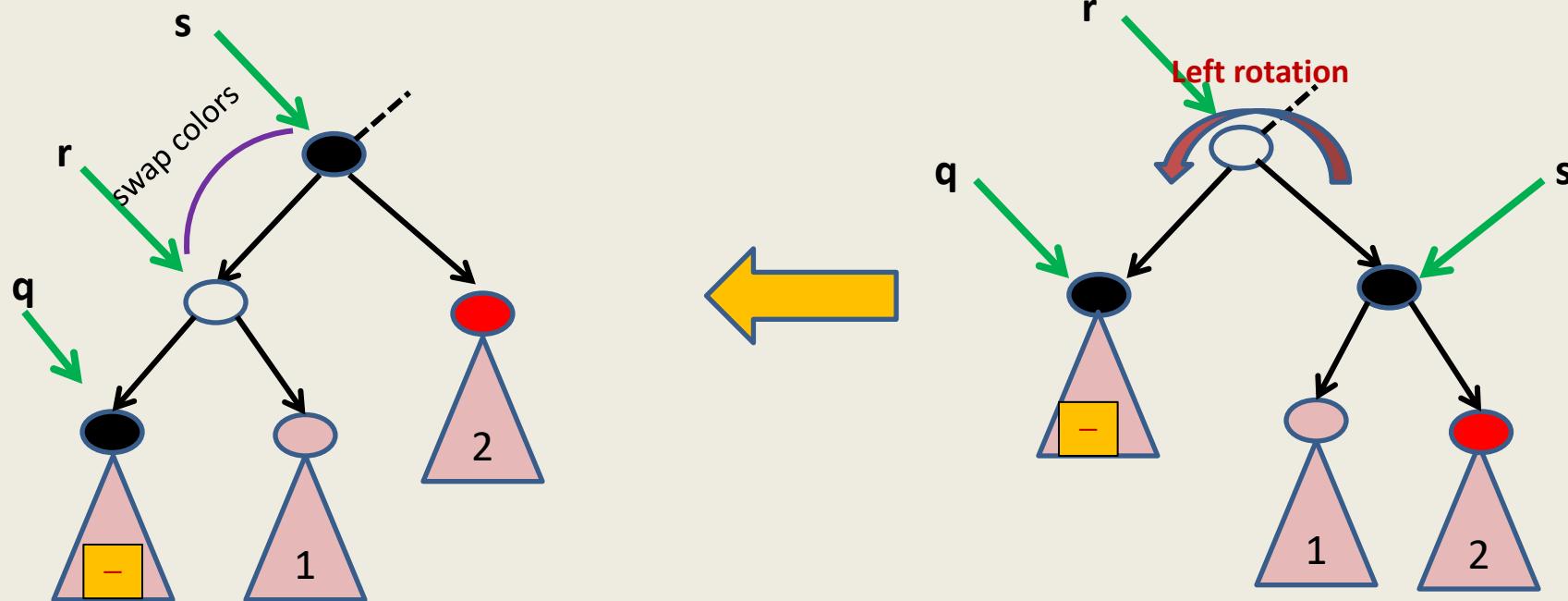
Handling the case: right(s) is red

Handling the case: right(s) is red



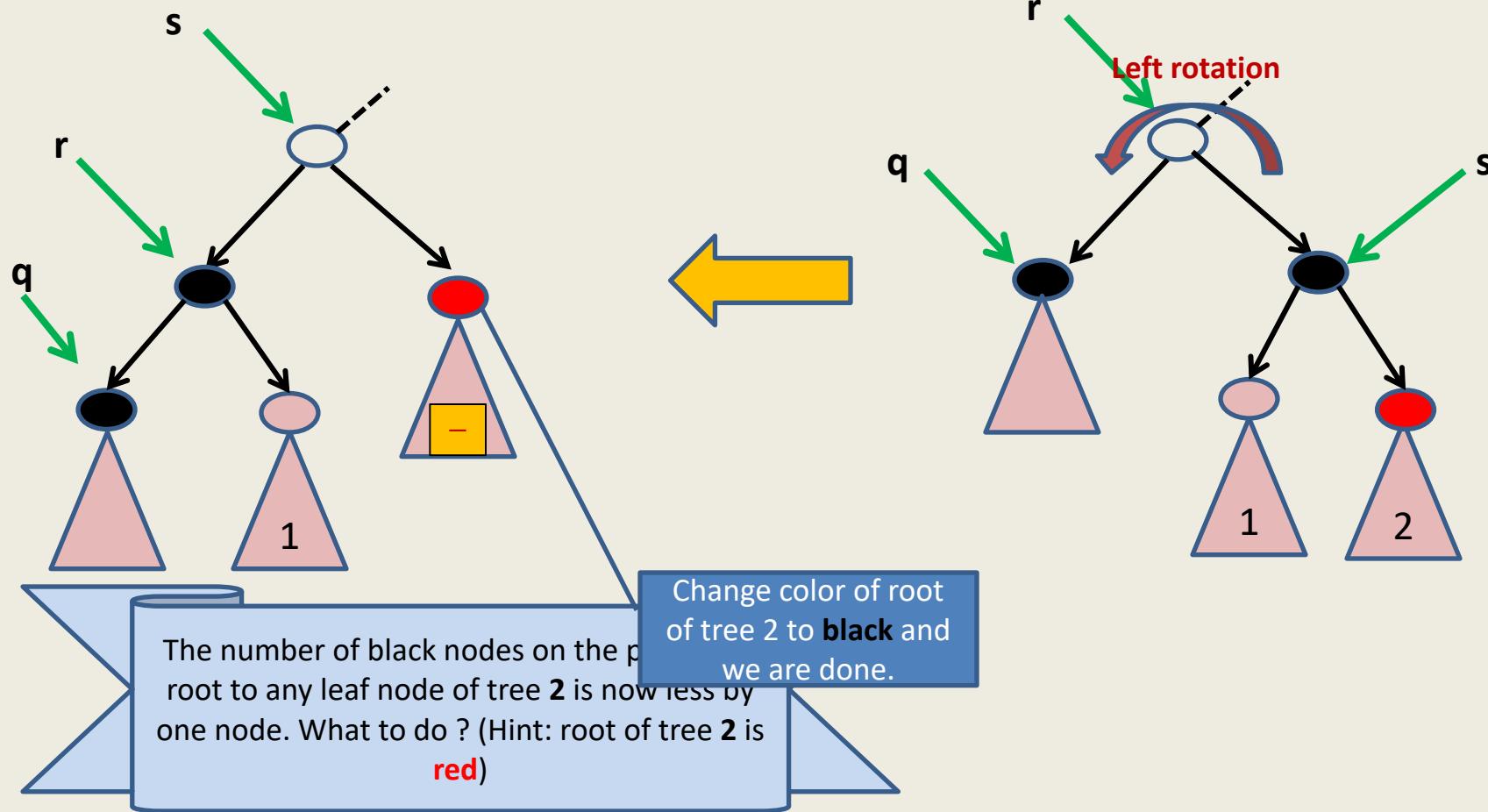
Let $\text{color}(r)$ be c

Handling the case: right(s) is red

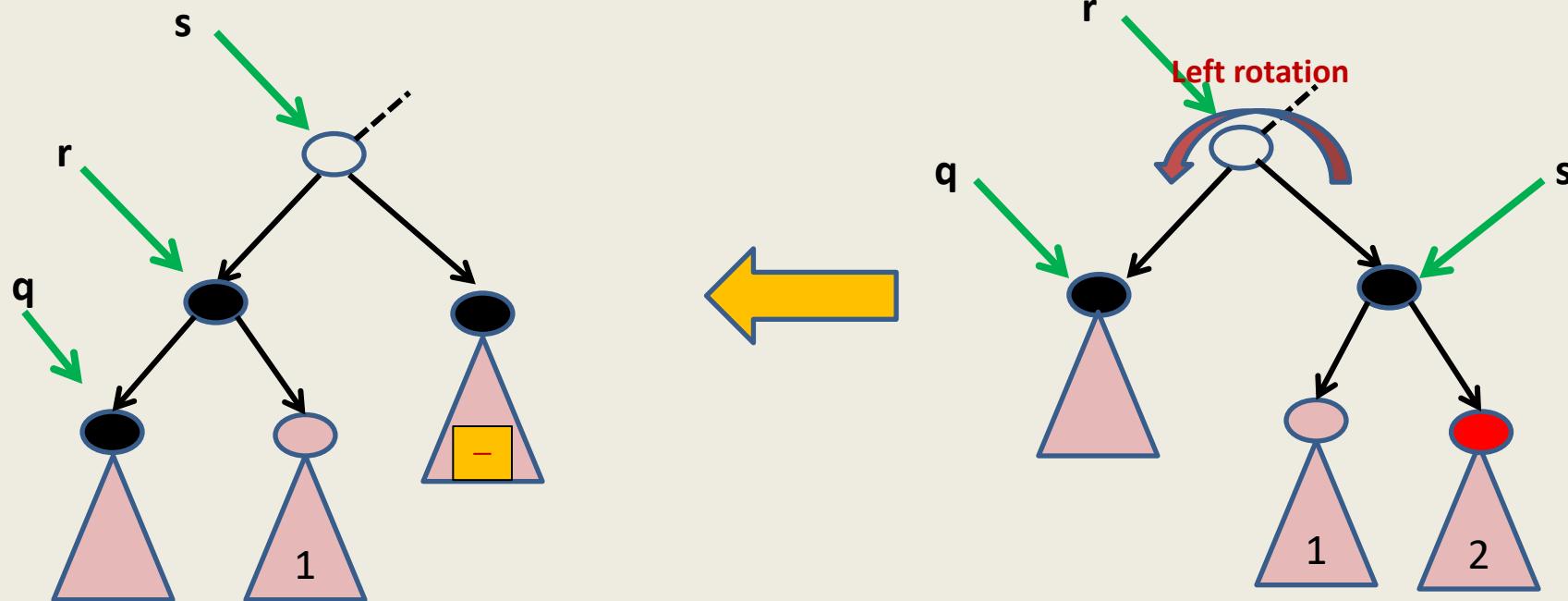


The number of black nodes on the path from root to any leaf node of subtree(q) has increased by one (this is good!), has remained unchanged for leaves of tree 1, and is uncertain for leaves of tree 2(depends upon c). How to get rid of this uncertainty ?

Handling the case: right(s) is red



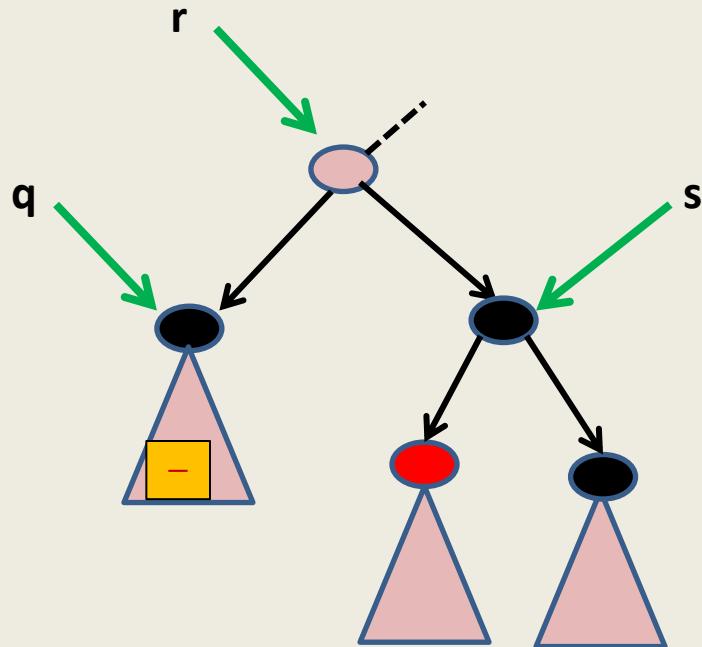
Handling the case: right(s) is red



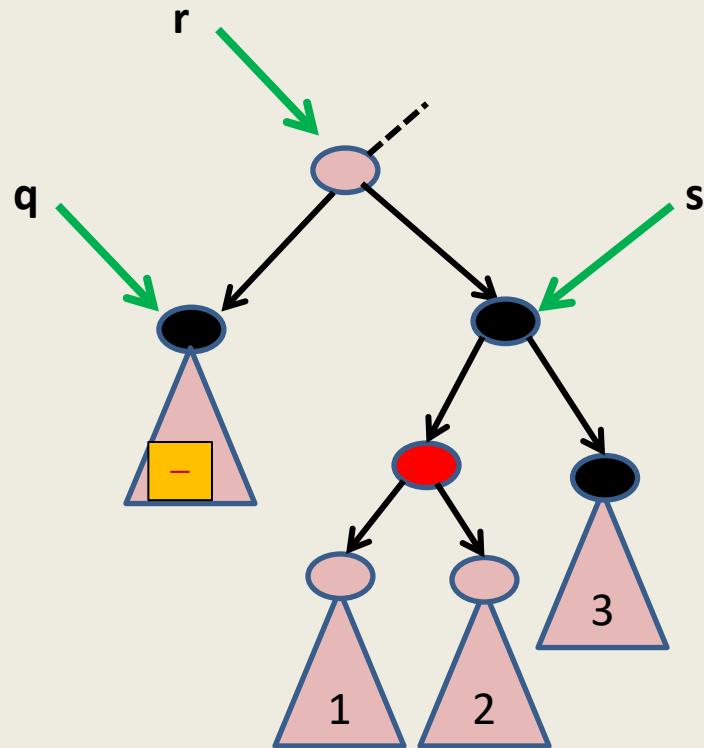
Convince yourself that left rotation at r , followed by color swap of s and r , followed by change of color of root of tree 2 removes the imbalance of black height for all leaf nodes of the subtrees shown.

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

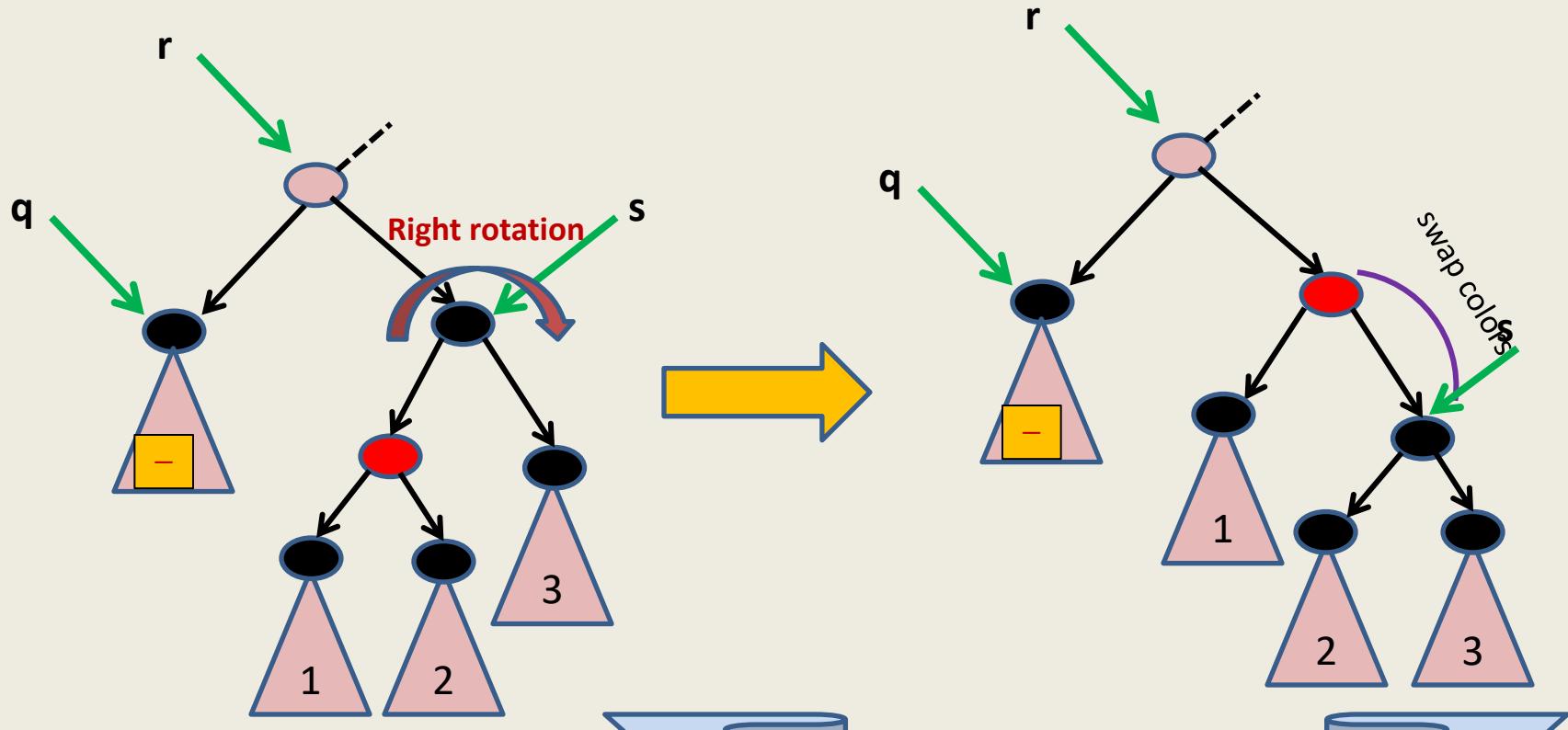
Handling the case: left(s) is red and right(s) is black



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

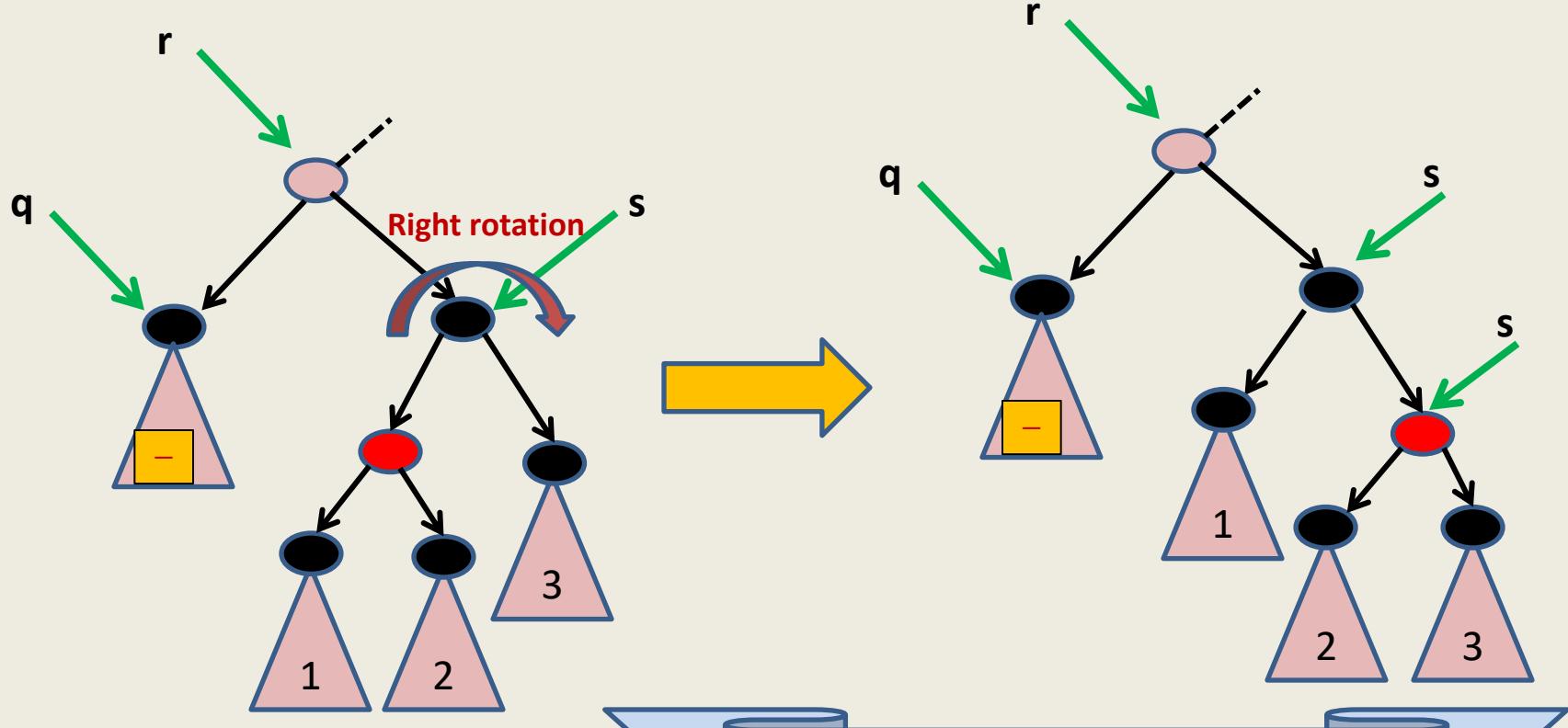


Handling the case: left(s) is **red** and right(s) is **black**



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**

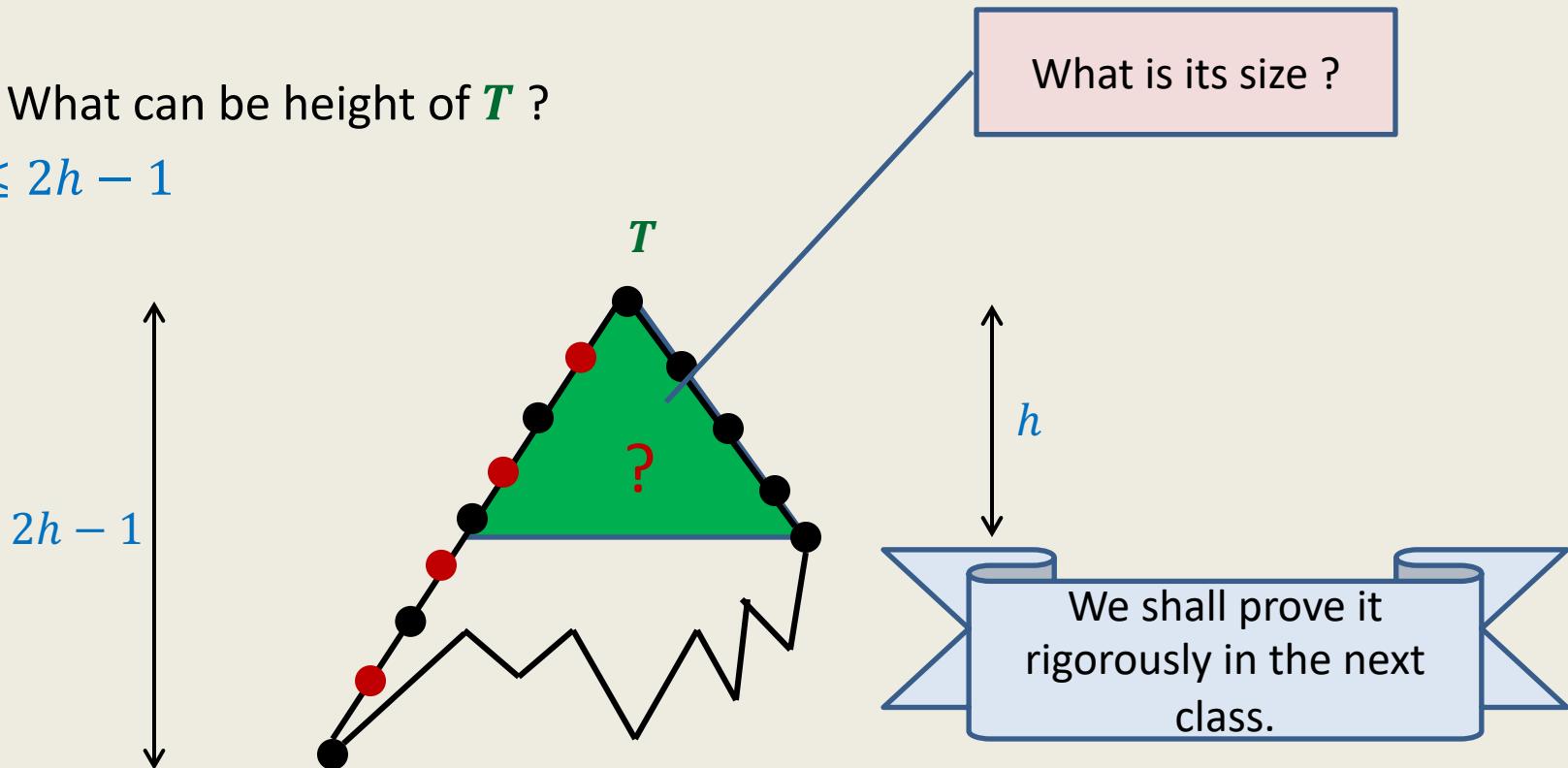
Why is a red black tree height balanced ?

T : a red black tree

h : black height of T .

Question: What can be height of T ?

Answer: $\leq 2h - 1$

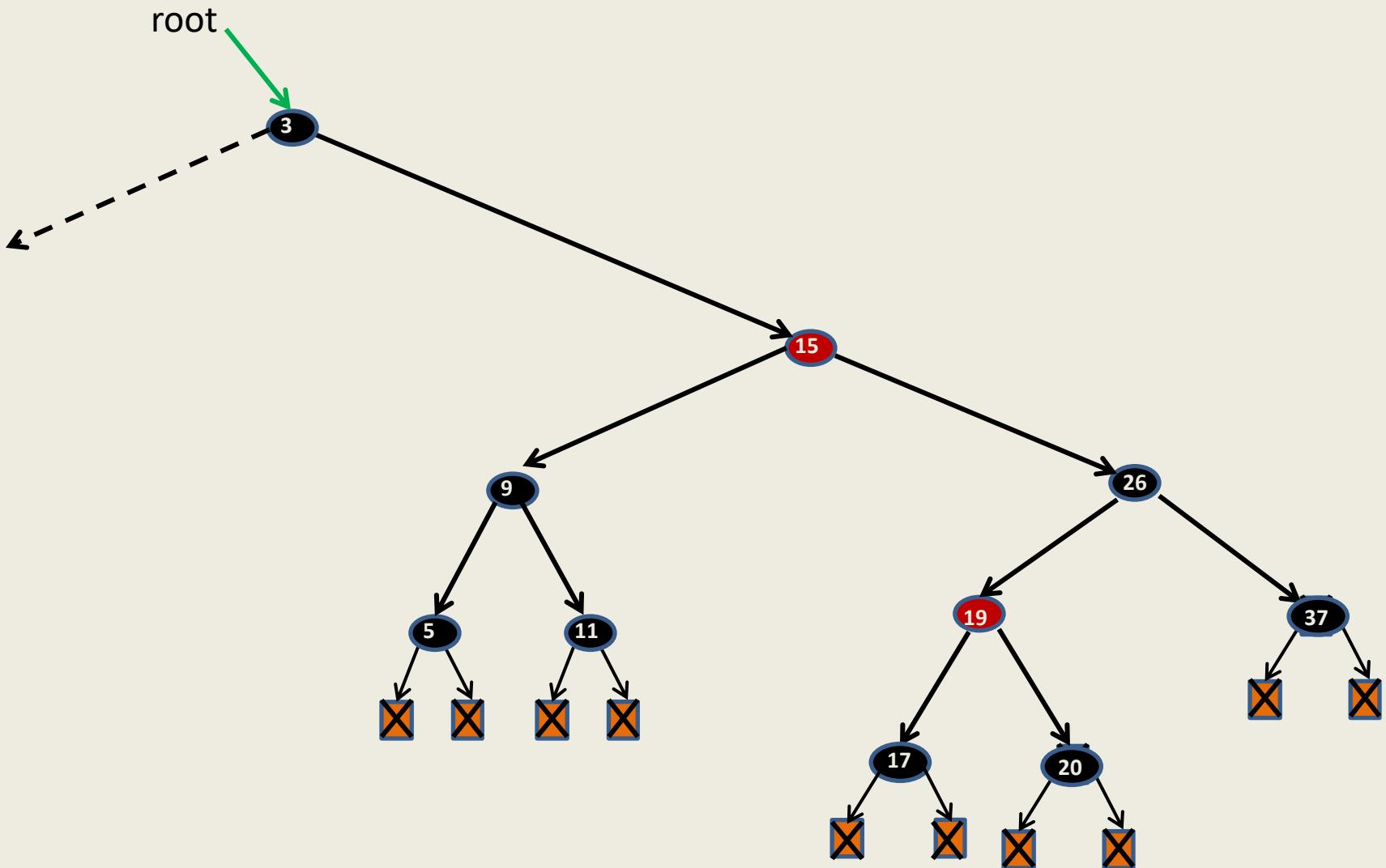


Theorem: The shaded green tree is a complete binary tree & so has $\geq 2^h$ elements.⁴⁷

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

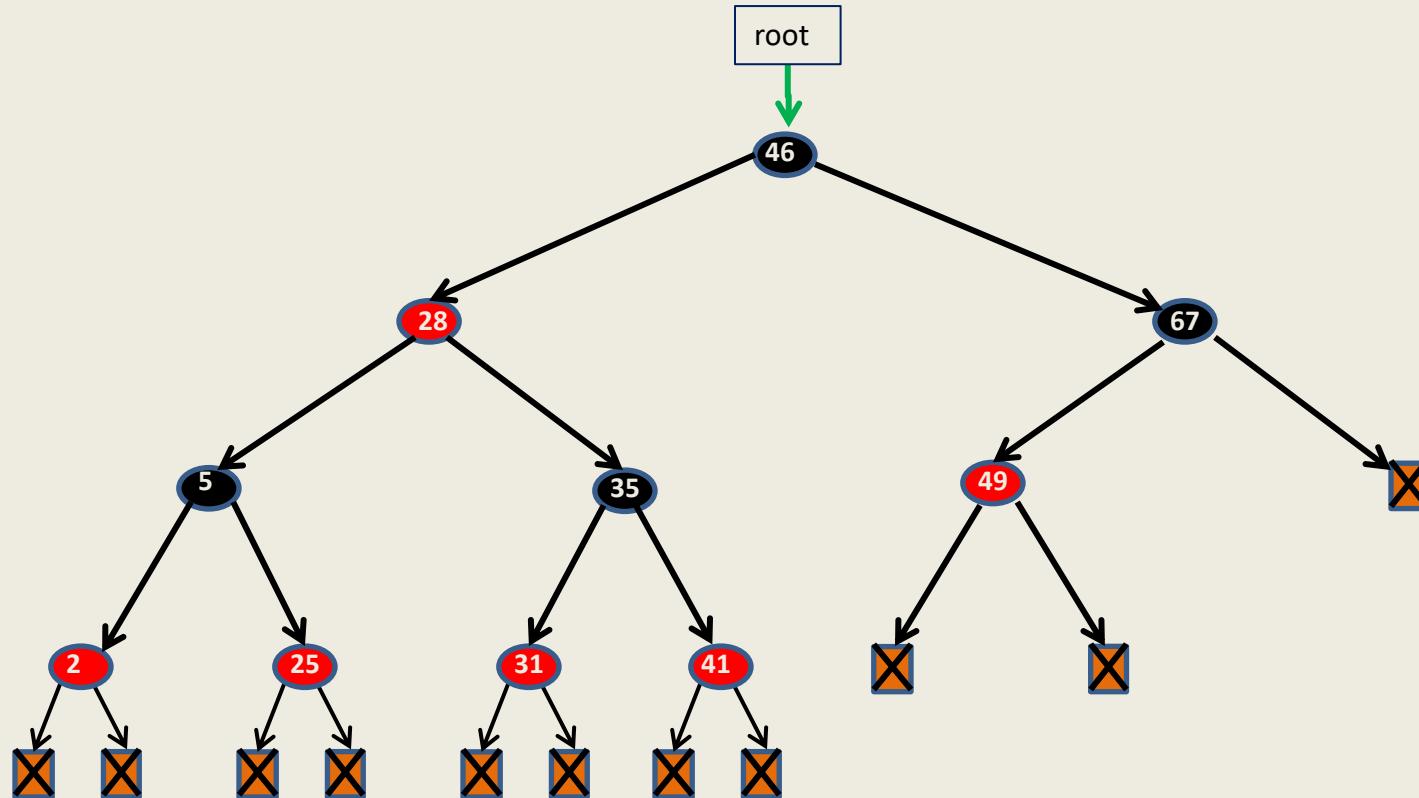
Red Black tree:

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

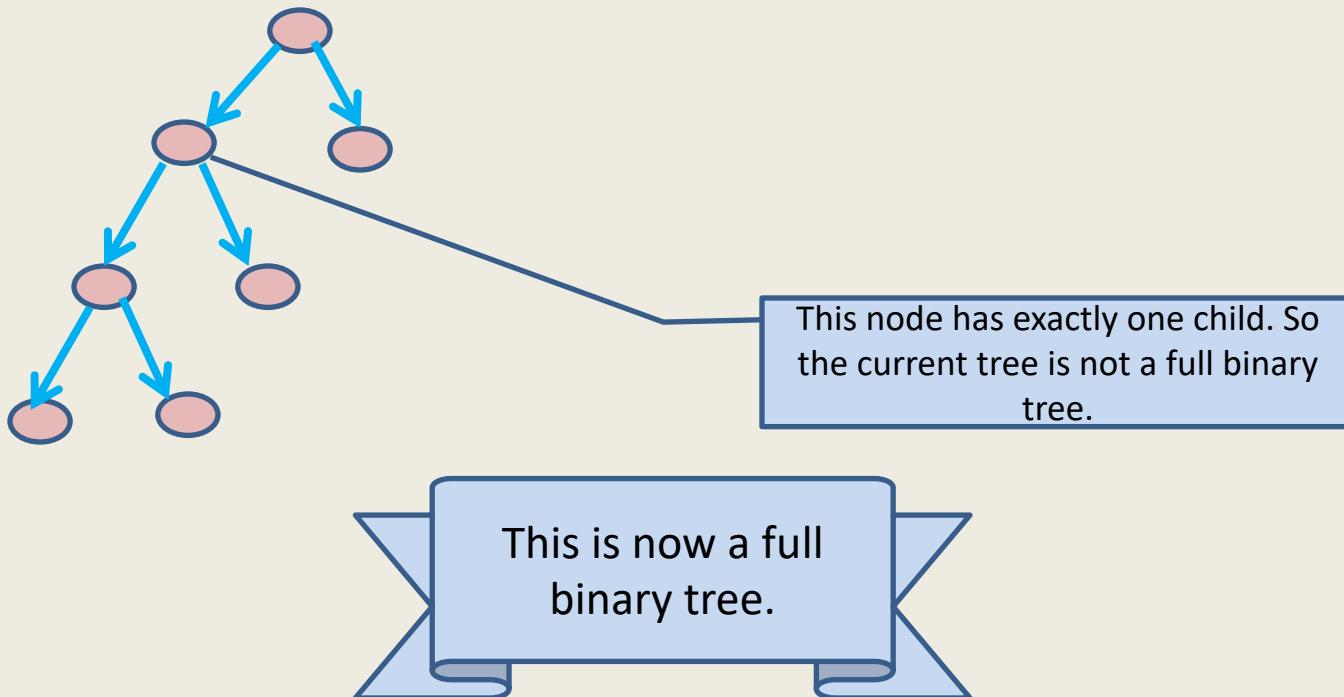
A red-black tree



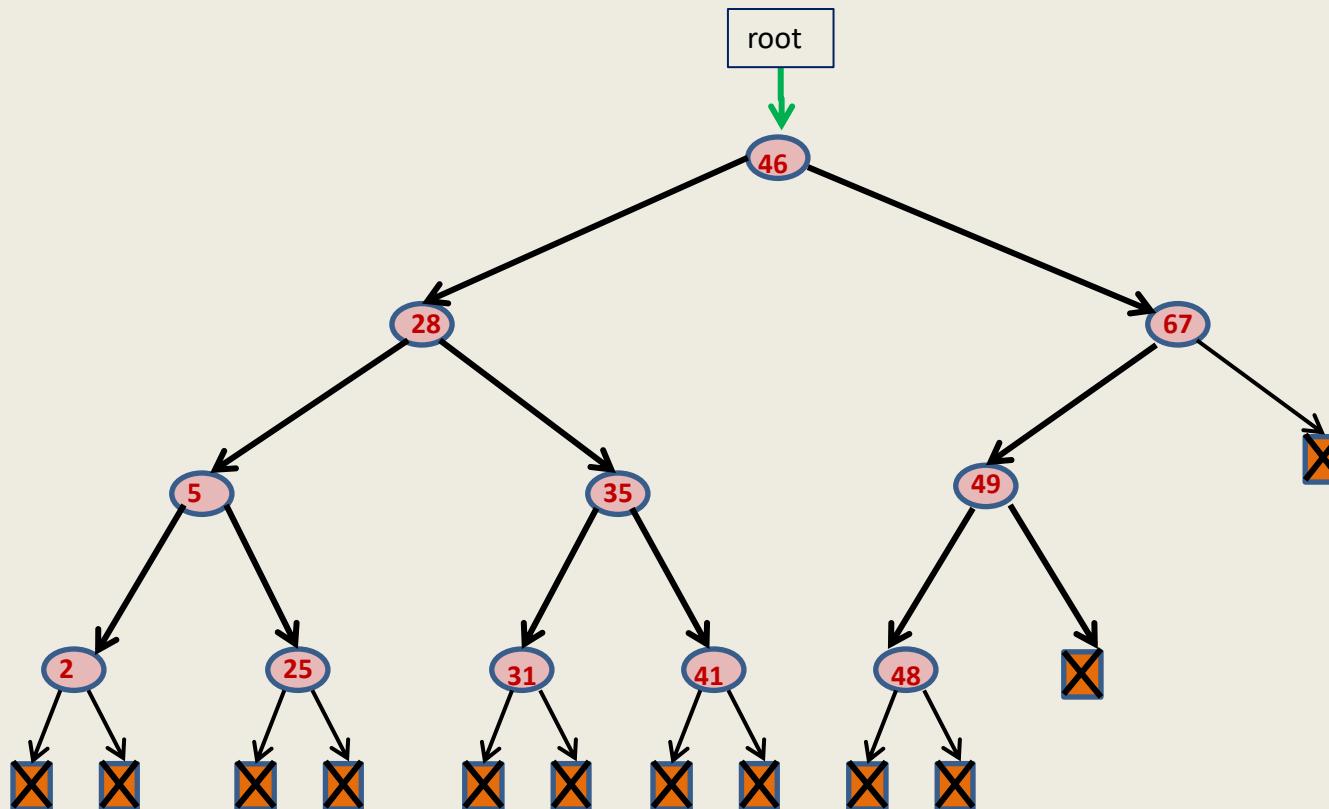
Terminologies

Full binary tree:

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

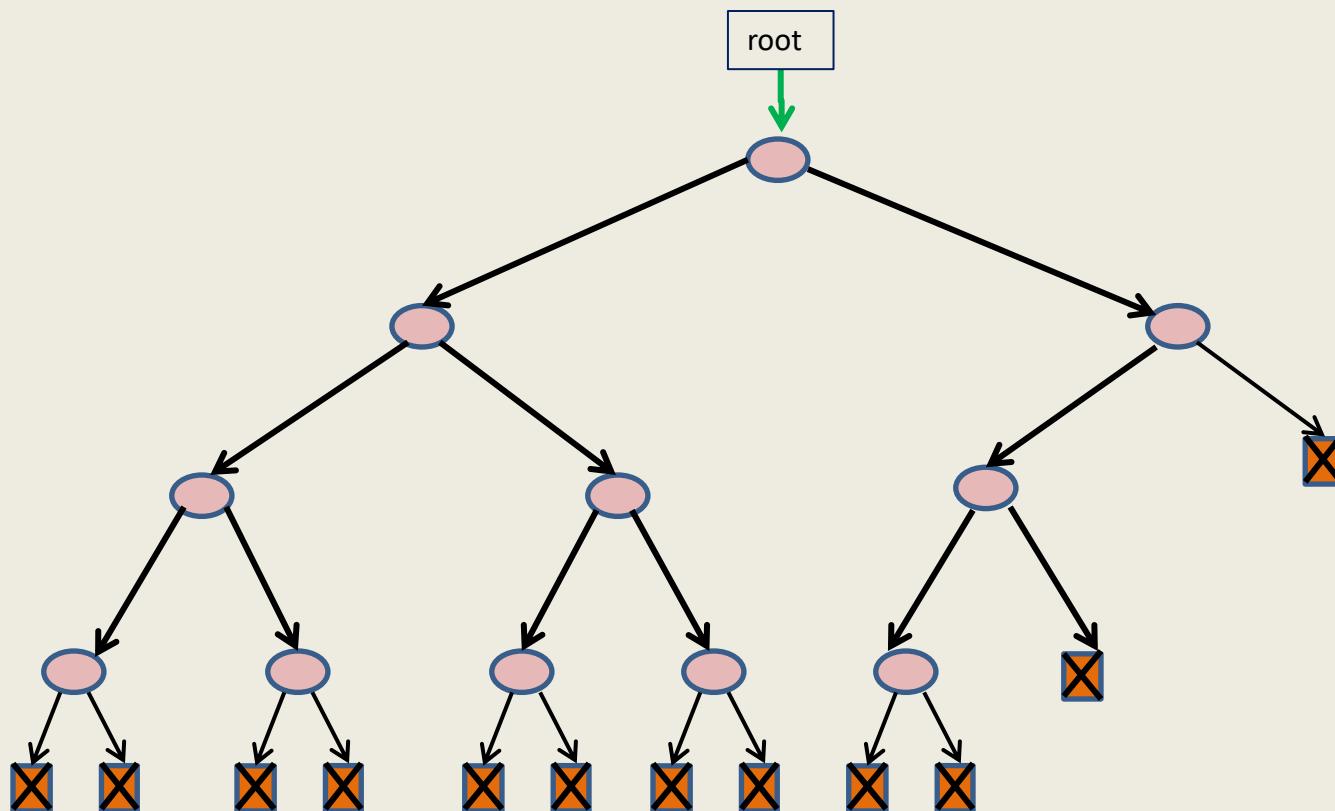


Red-black tree: as a Full Binary Tree



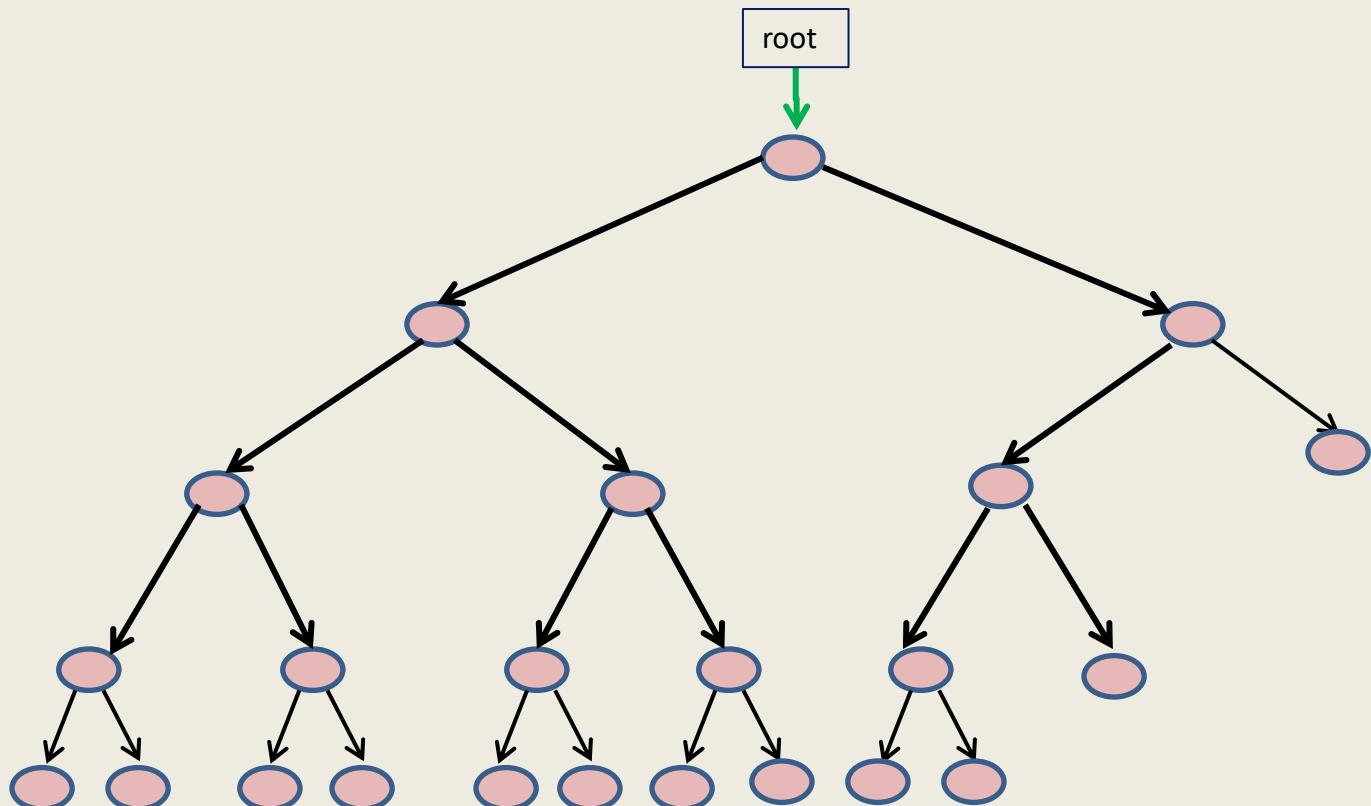
Ignore the values

Red-black tree: as a Full Binary Tree



Ignore the distinction
between internal nodes
and leaf nodes

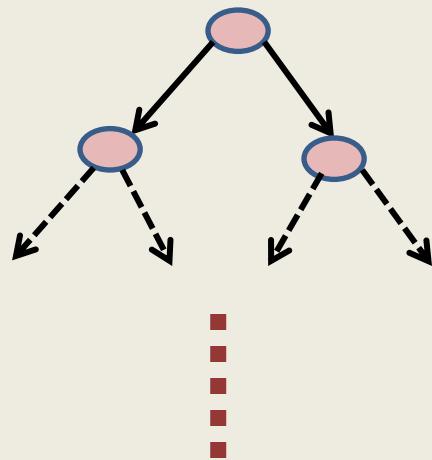
Red-black tree: as a Full Binary Tree



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

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

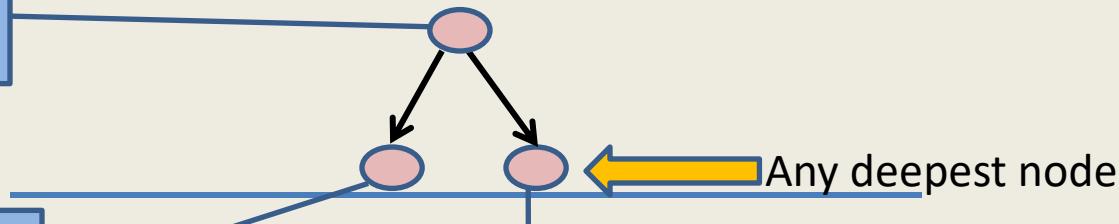
A full binary tree



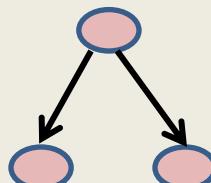
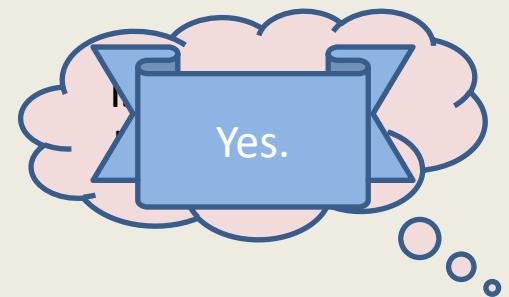
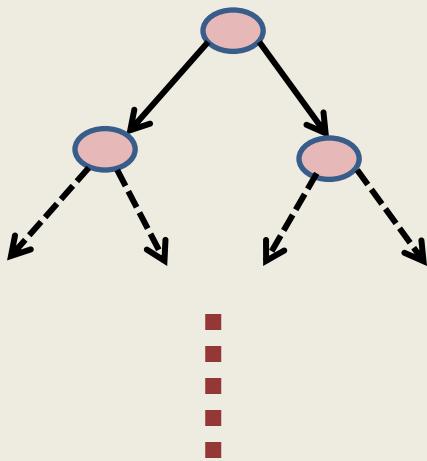
This node must have a left child since the tree is a full binary tree

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

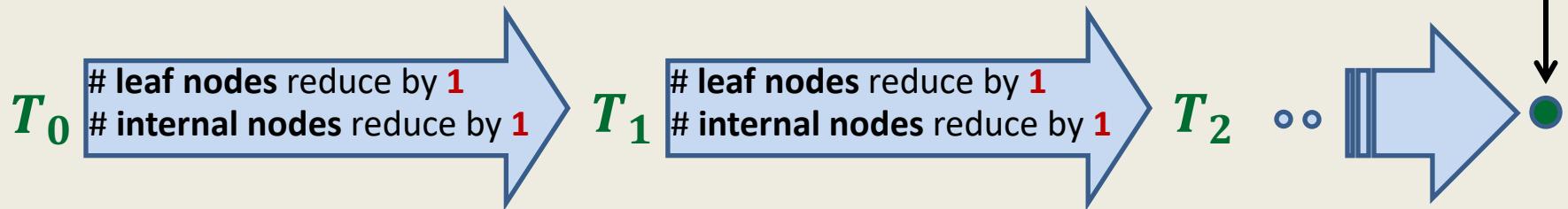
{ Delete the deepest node and its sibling }

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

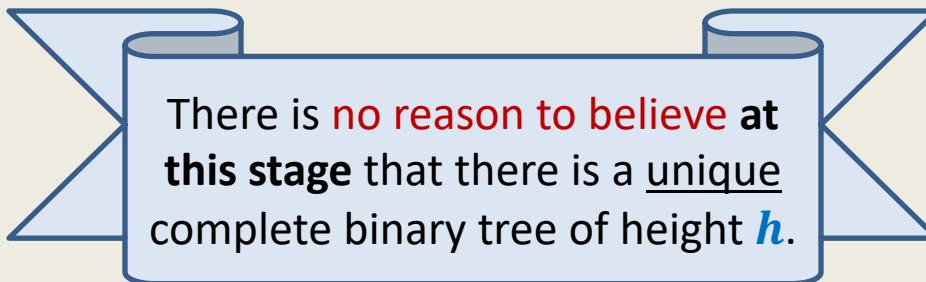
A **complete** binary tree of height h

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

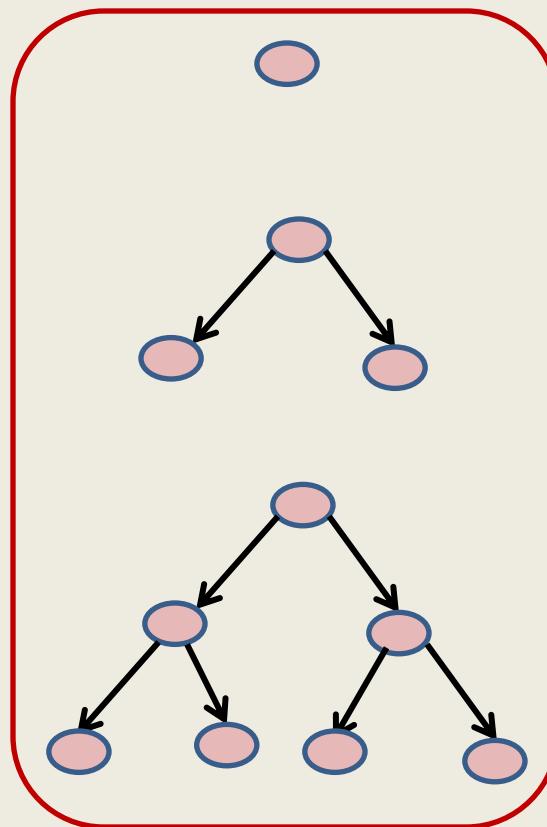
if every leaf node is at depth h .

A complete binary tree of height h

Complete binary tree of height 1 ?

Complete binary tree of height 2 ?

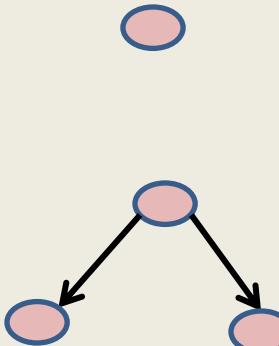
Complete binary tree of height 3 ?



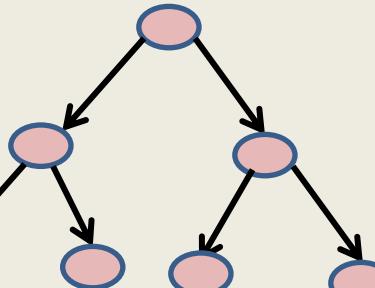
Try to generalize
the type of tree
shown here to
tree of height h .

A complete binary tree of height h

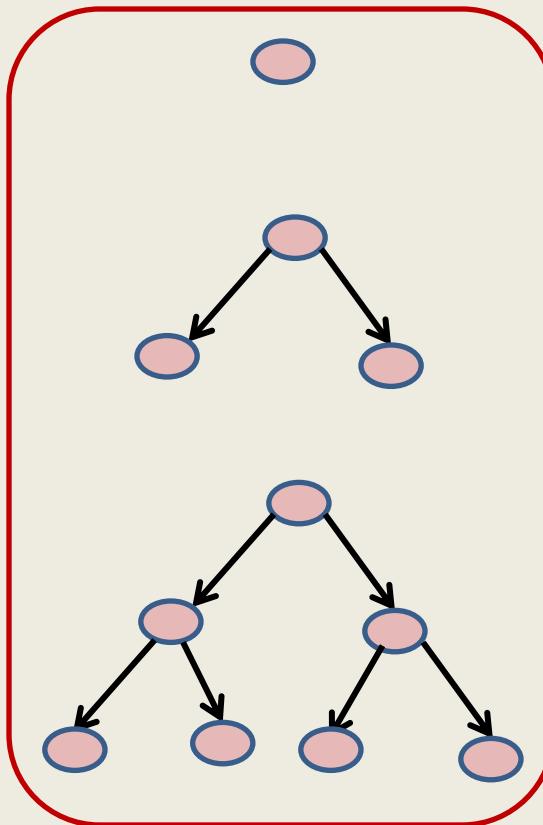
Complete binary tree of height 1 ?



Complete binary tree of height 2 ?



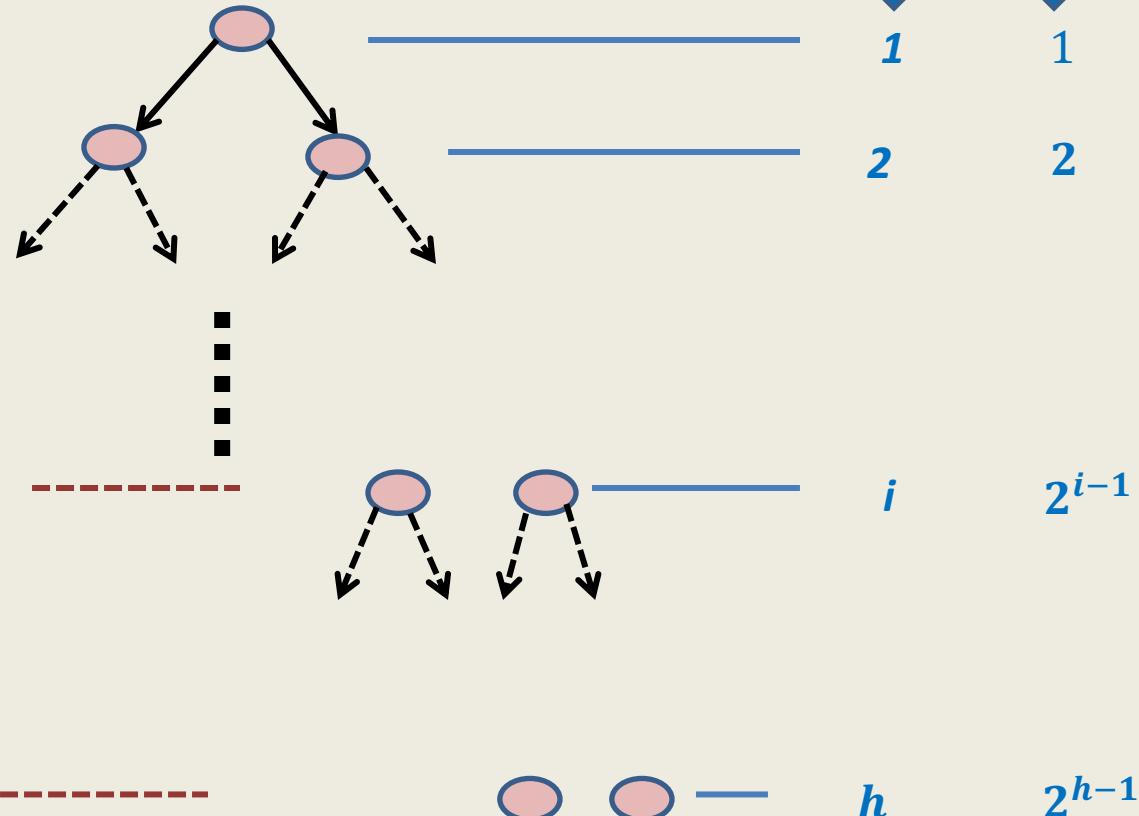
Complete binary tree of height 3 ?



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 .

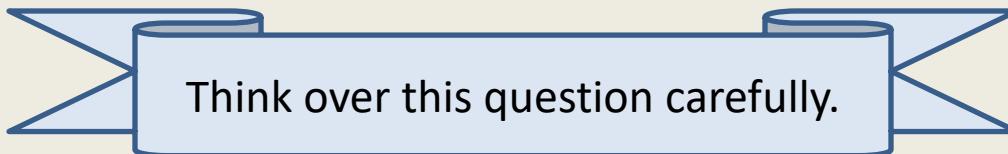
Uniqueness of a **complete** binary tree of height h

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

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

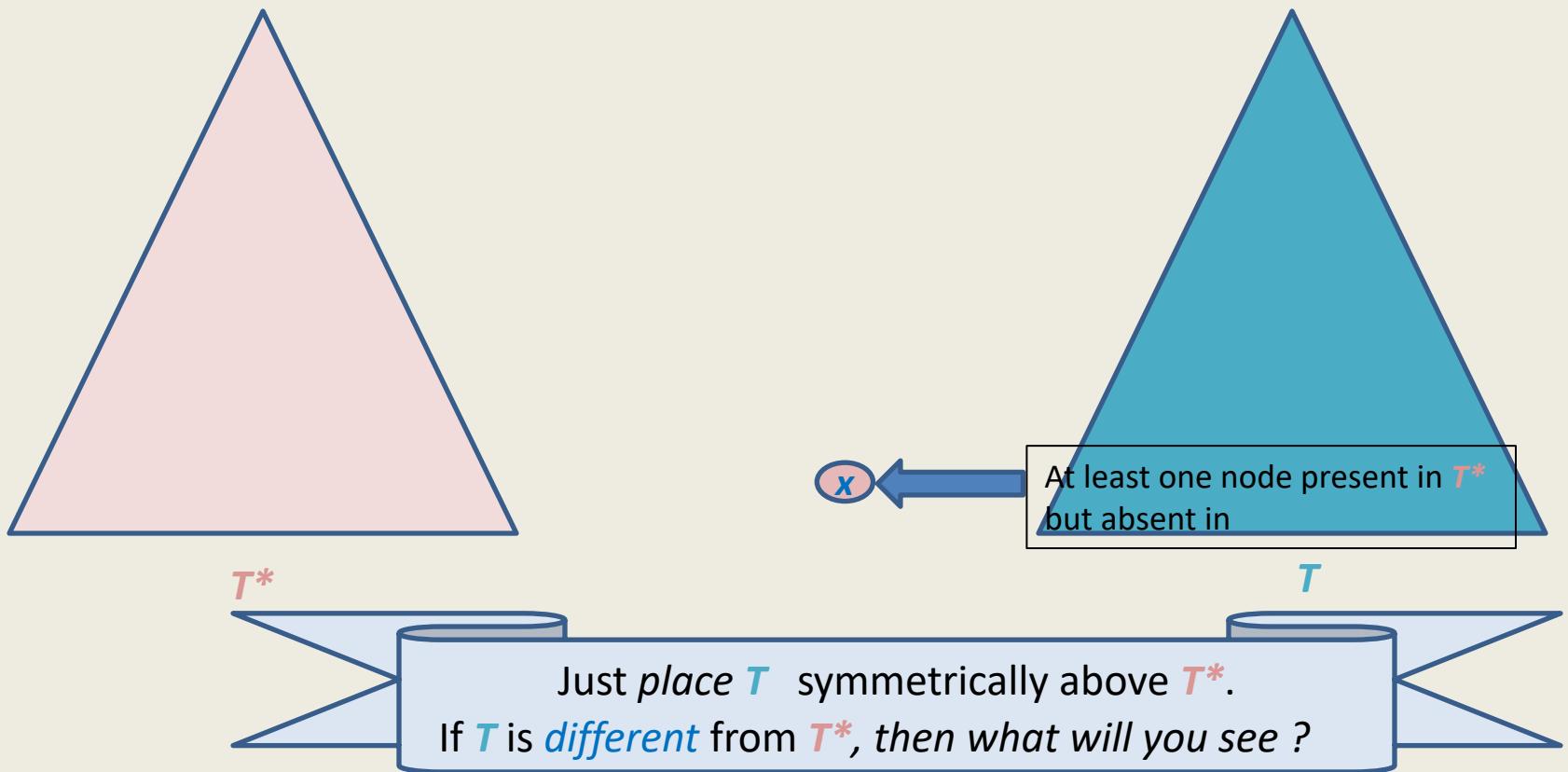
Let T be any other **complete** binary tree of height h different from T^* .

Question: How to show that T can not exist ?

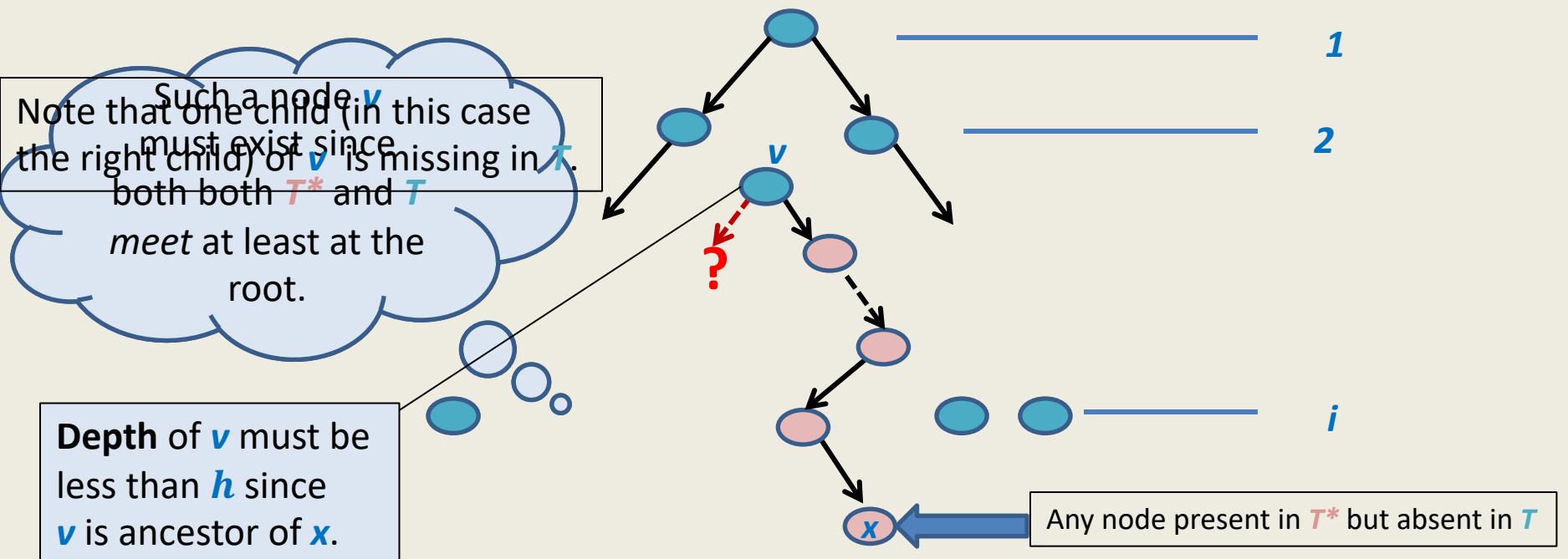


Watch the following slide carefully.

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

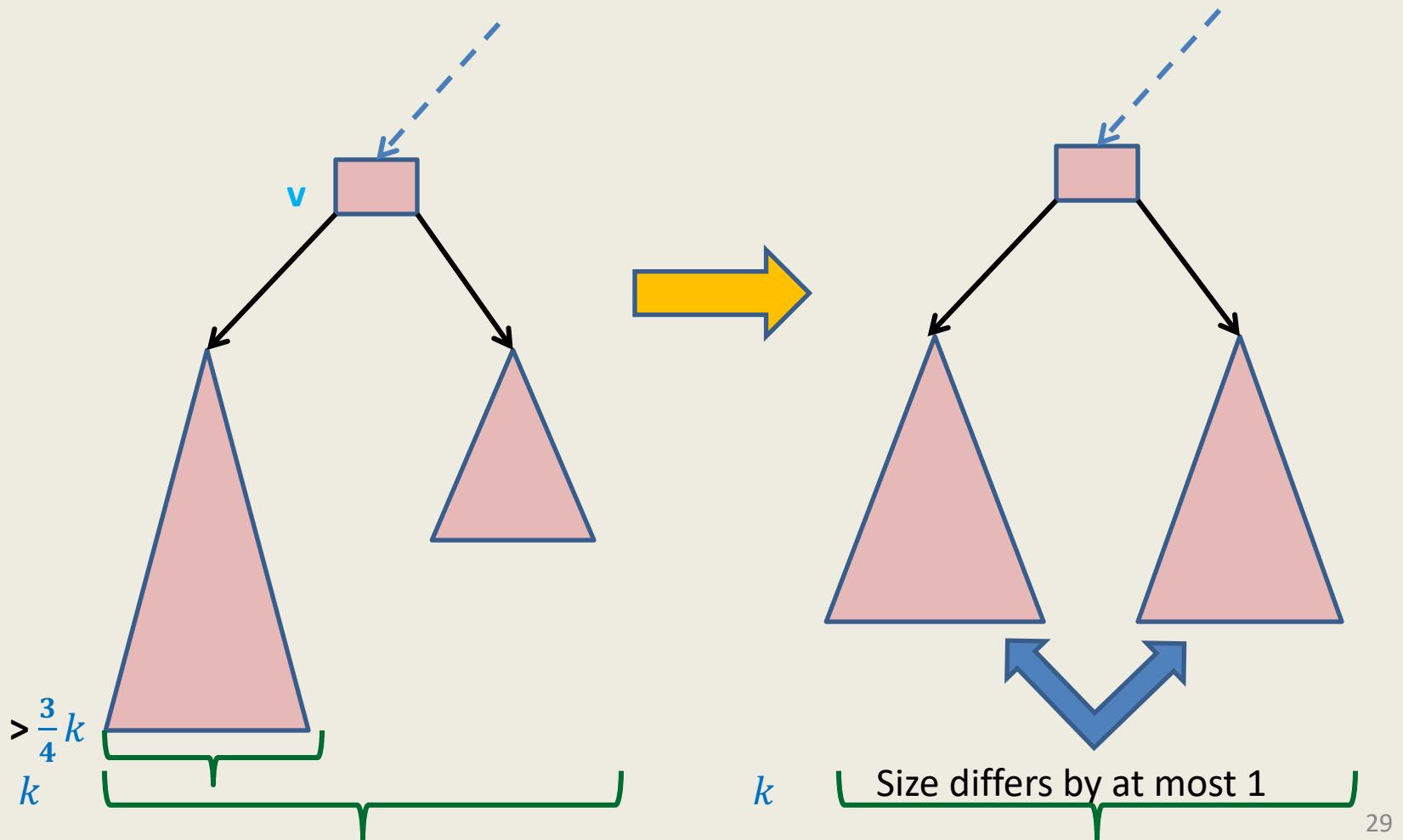
Nearly balanced Binary Search Tree

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 $O(n \log n)$.

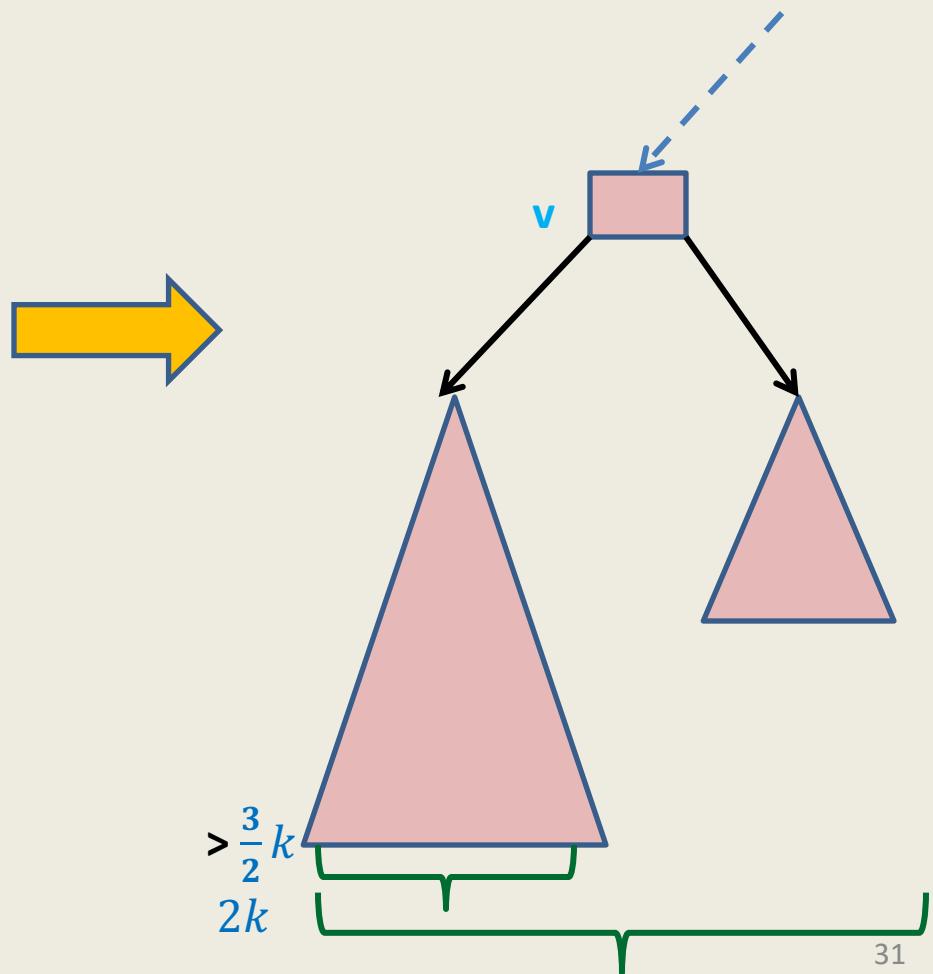
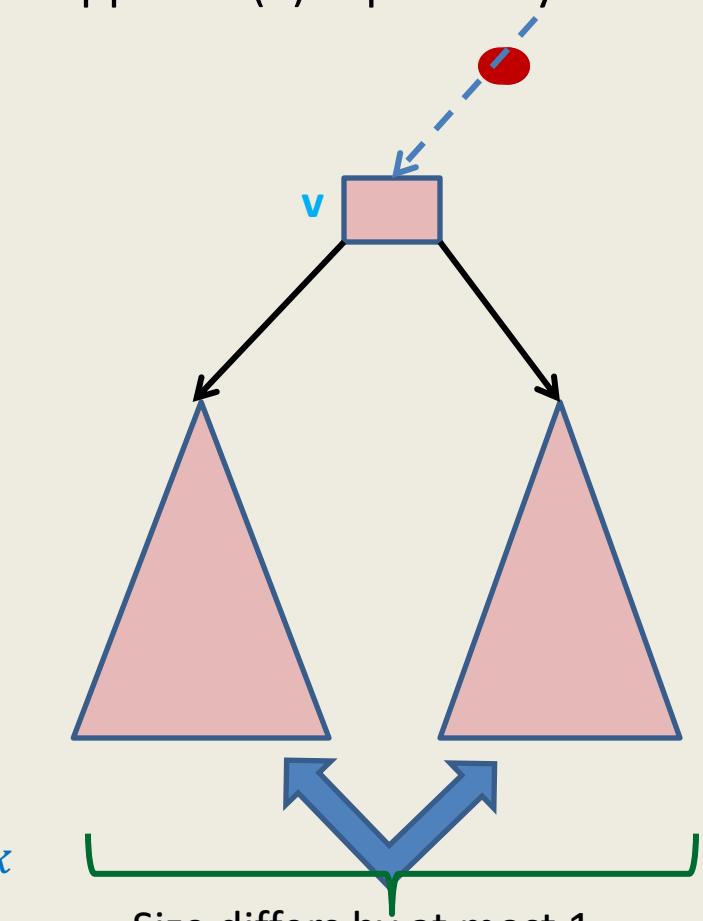
We shall now prove this theorem formally.

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,

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

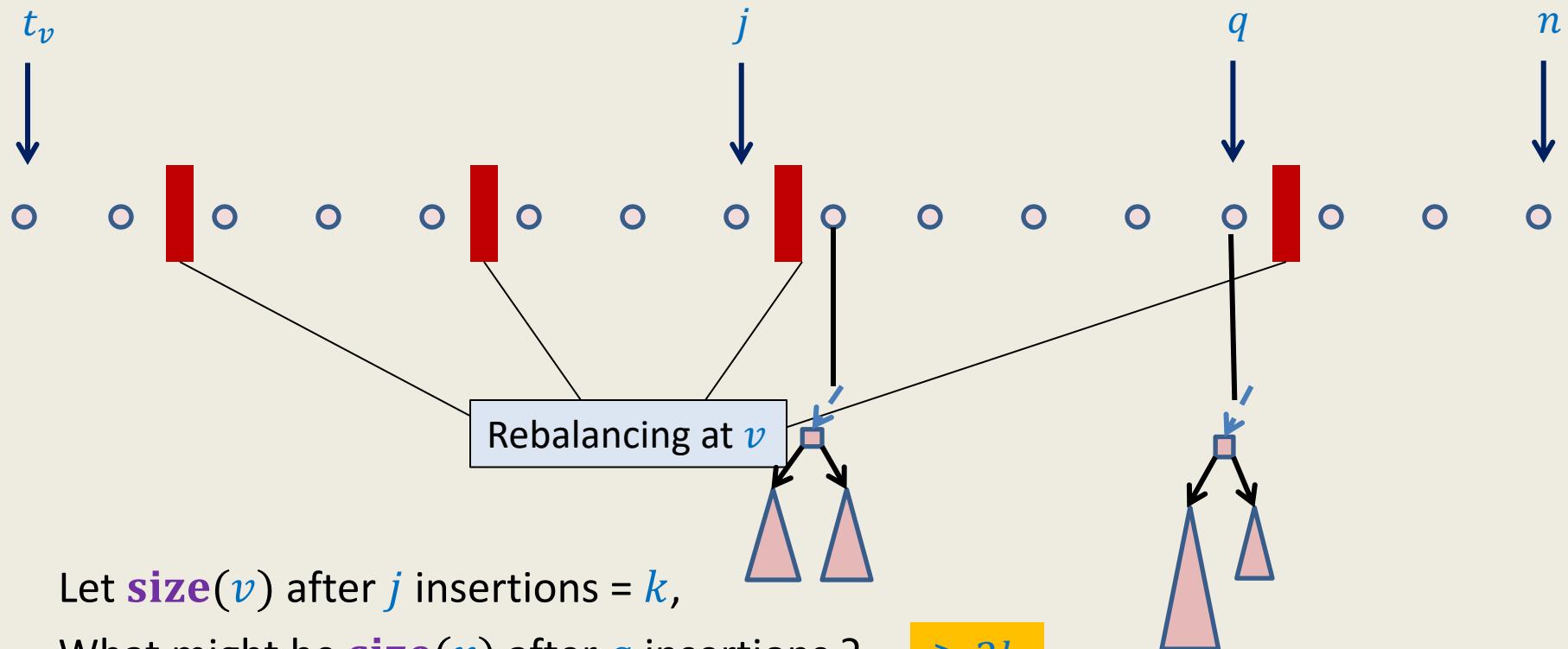
Question: For a nearly balanced BST, what is

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

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)

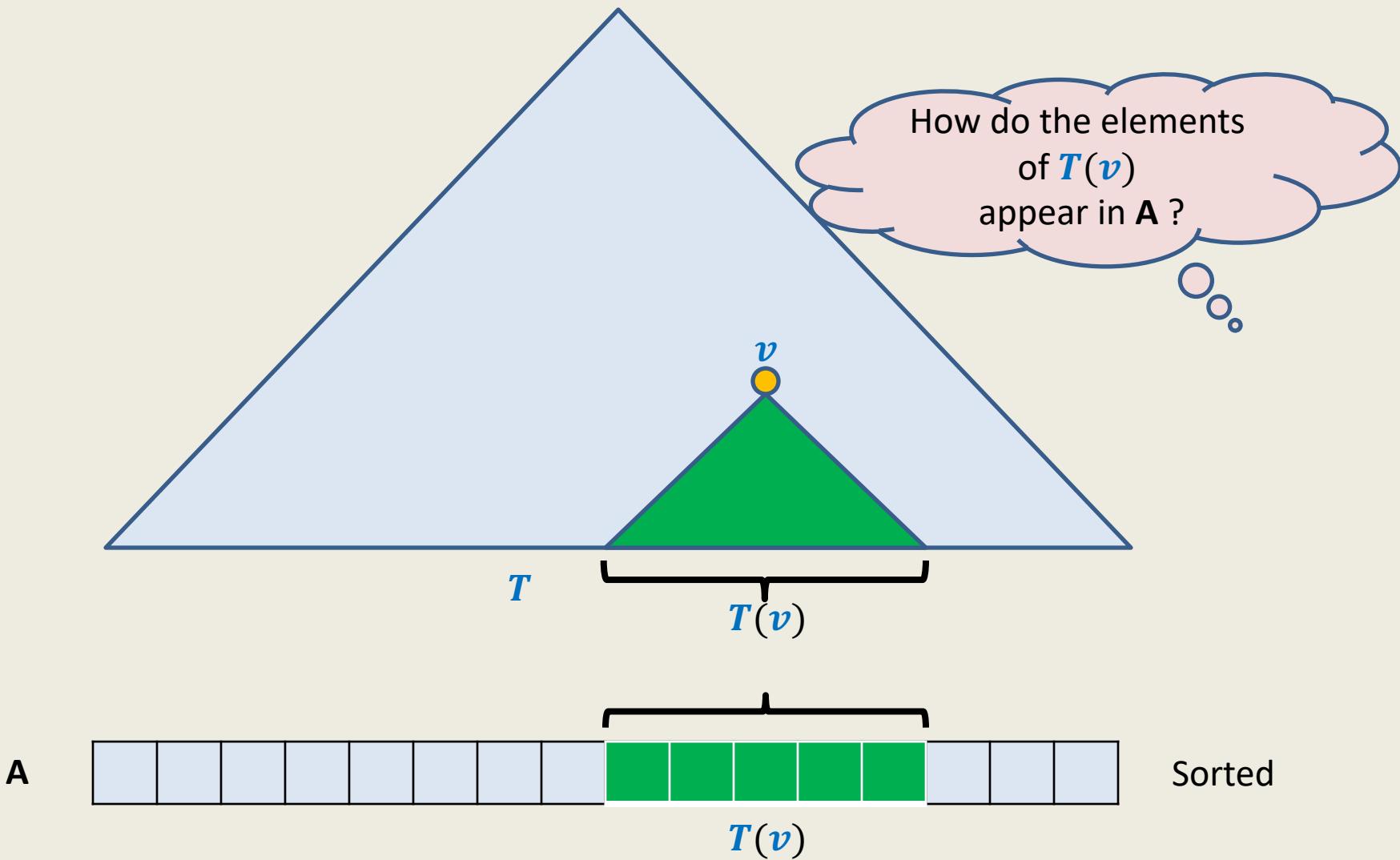
Operations you already know

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

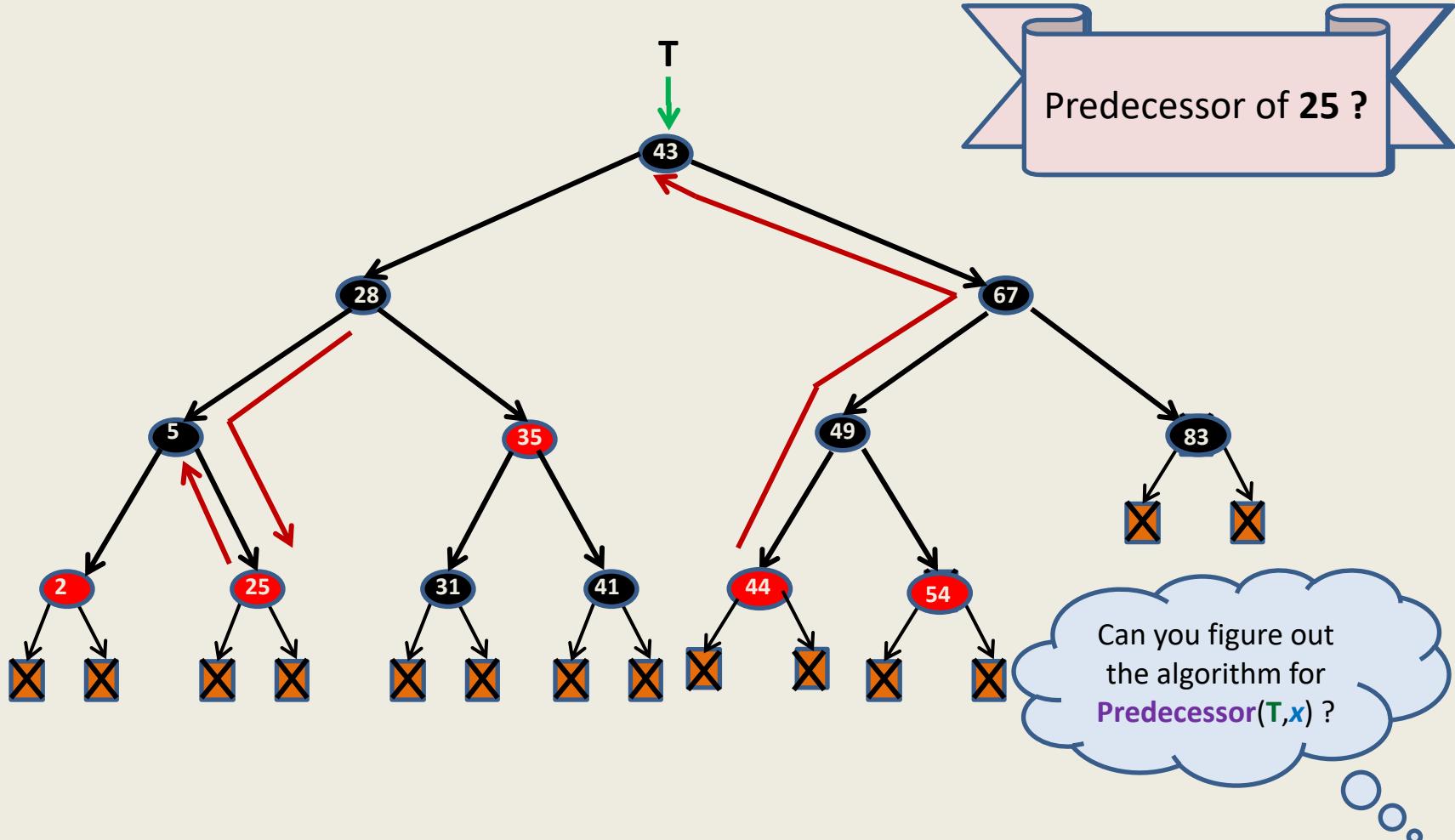
How well have you understood ?



Predecessor(T, x)

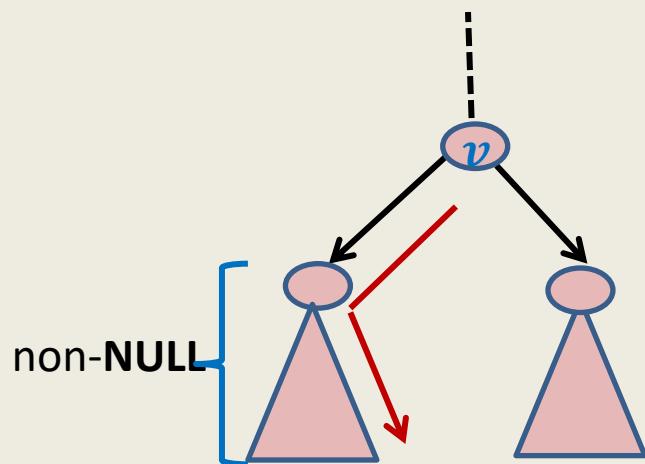
The **largest** element in T which is smaller than x

Predecessor(T, x)



Predecessor(T, x)

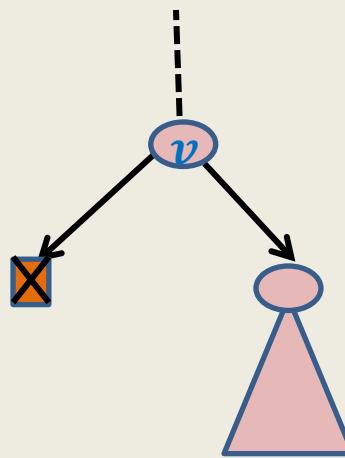
Let v be the **node** of T storing value x .



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

Predecessor(T, x)

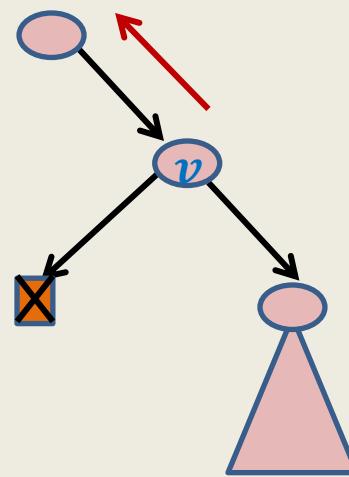
Let v be the **node** of T storing value x .



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 .

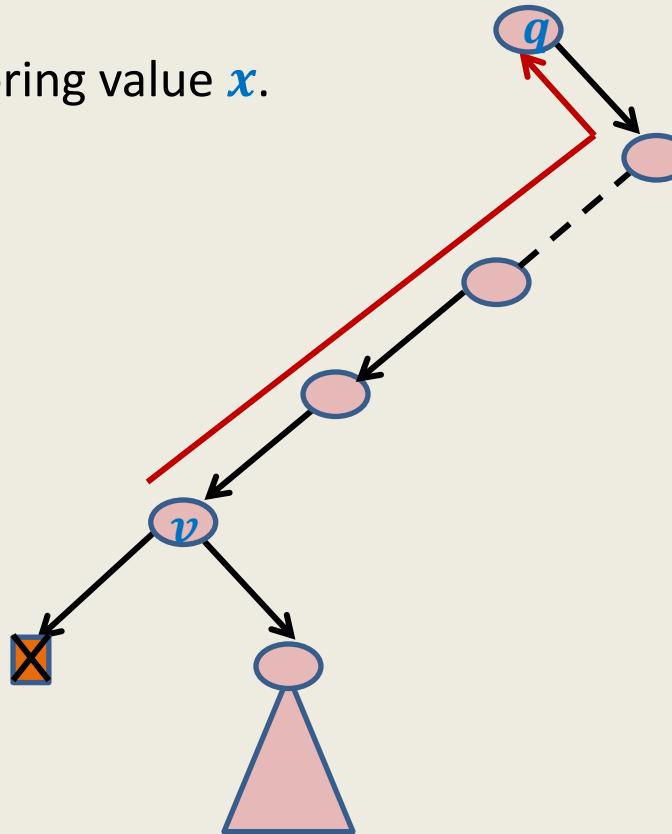


Case 2: $\text{left}(v) == \text{NULL}$, and v is **right child** of its **parent**

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' .

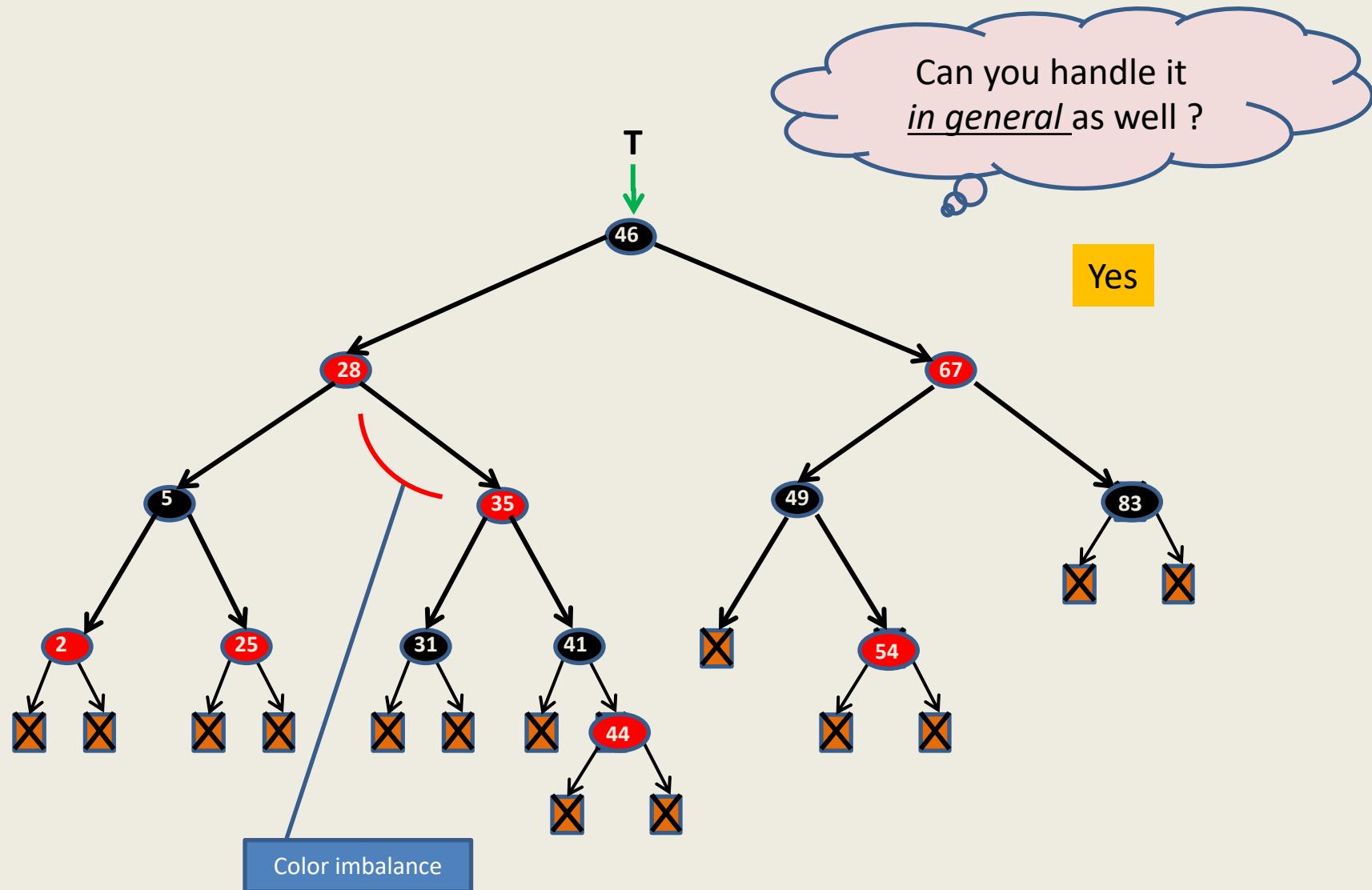


Every operation in $O(\log n)$ time.

Red-Black Tree

How well have you understood ?

Insertion in a red-black tree

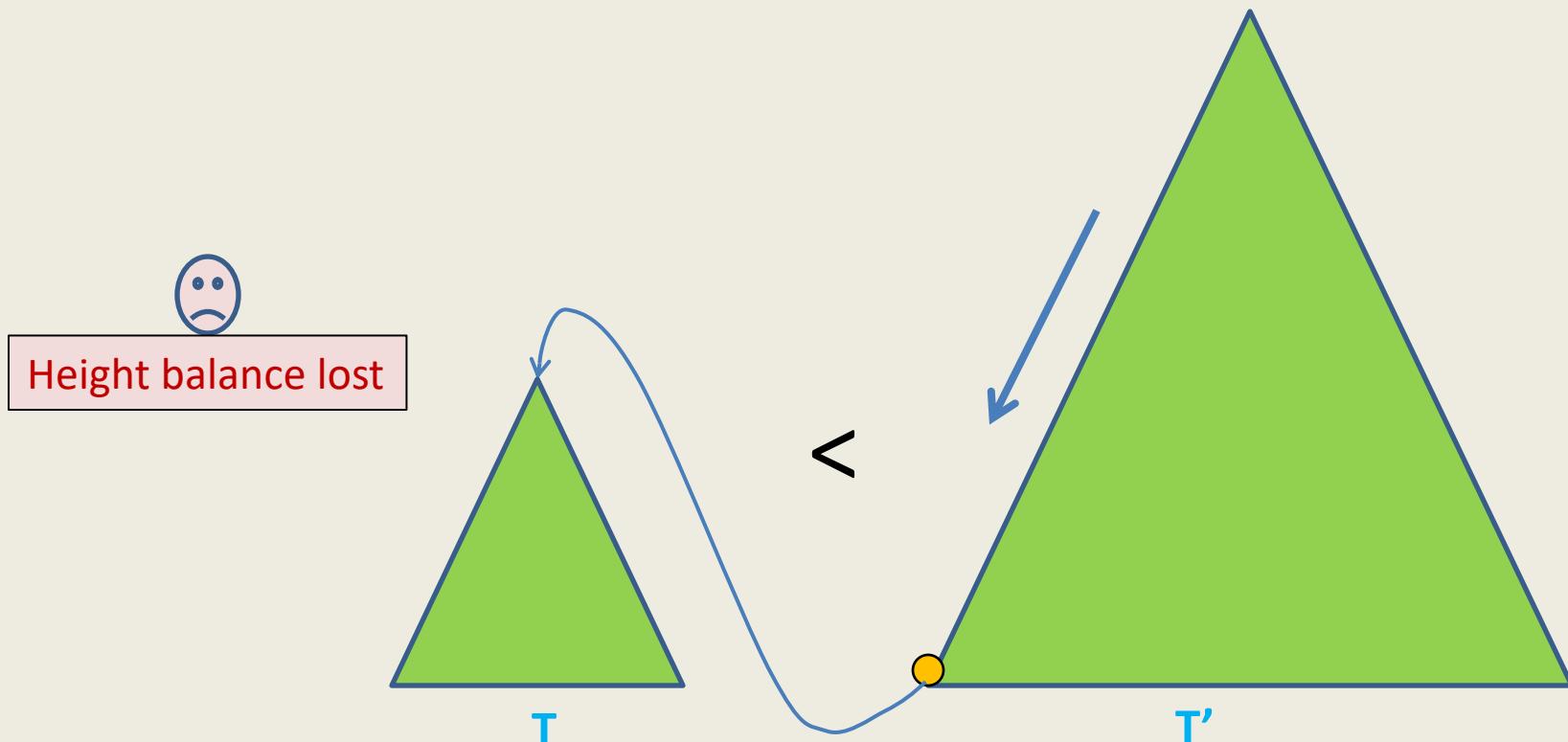


SpecialUnion(T, T')

Remember:

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

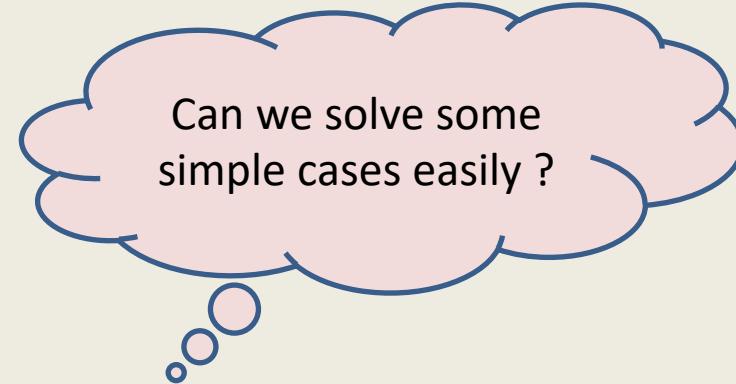
A trivial algorithm that does not work



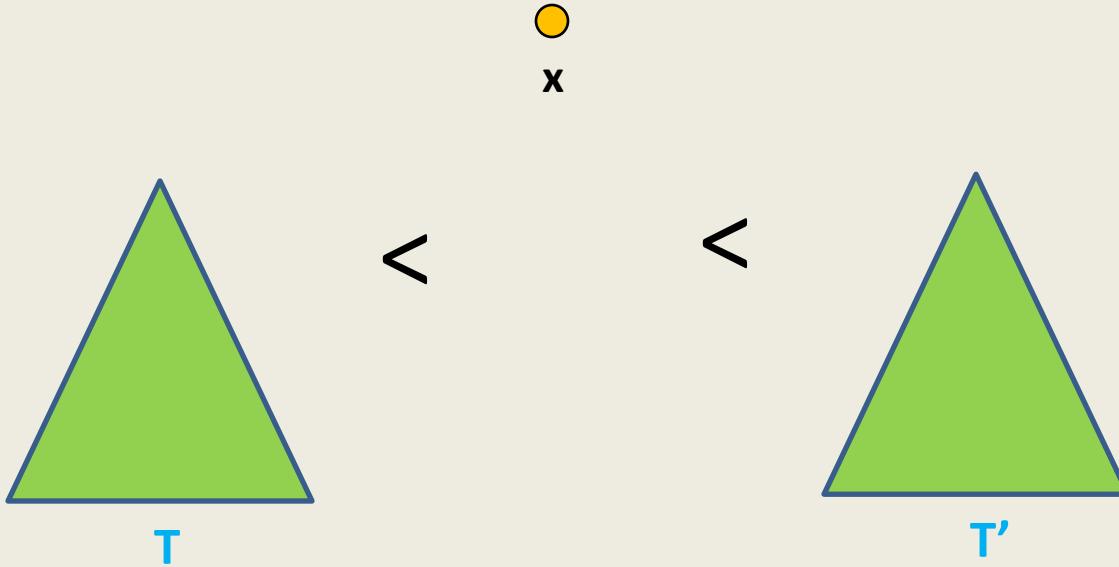
Time complexity: $O(\log n)$

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



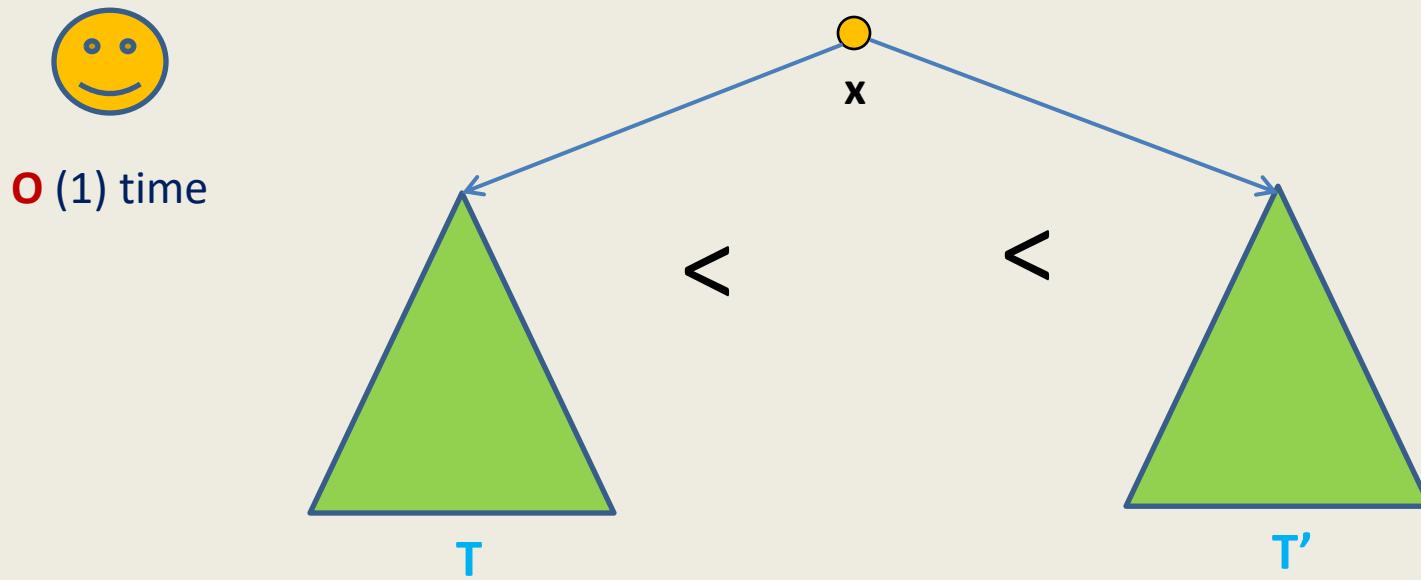
Simplifying the problem



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 $T^* = T \cup \{x\} \cup T'$

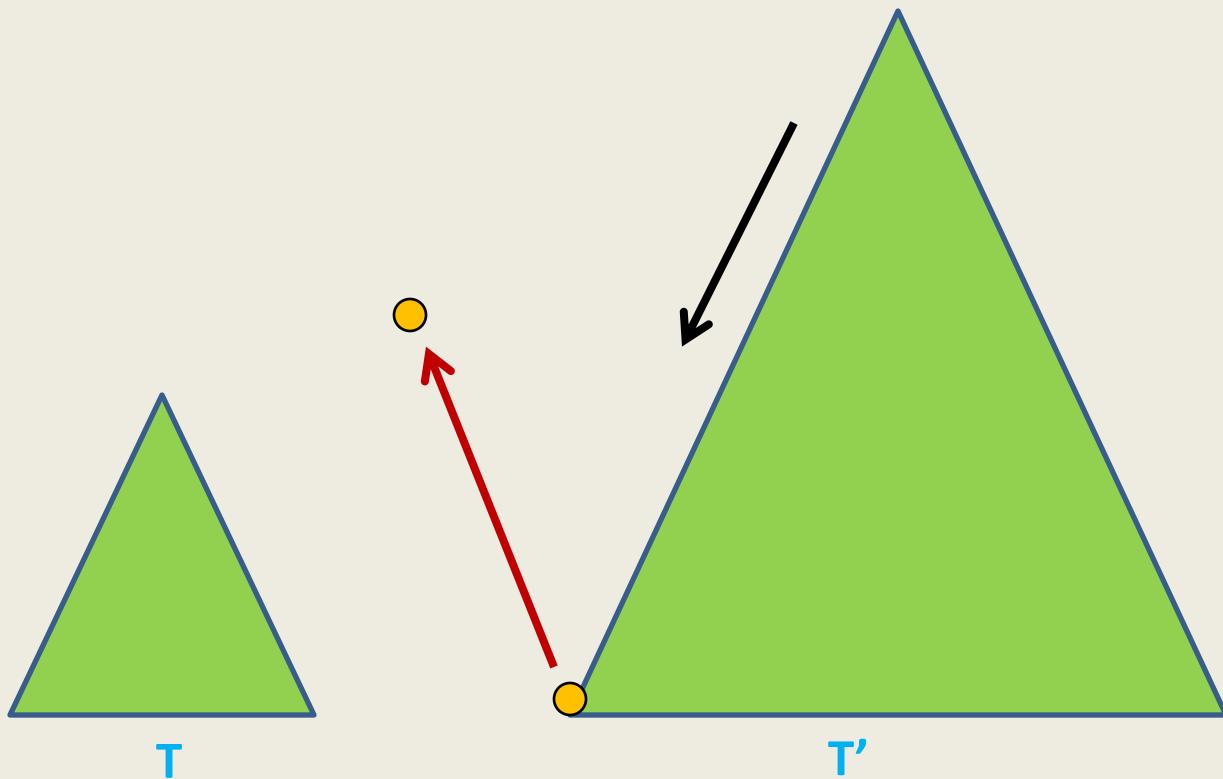
Solving the simplified problem



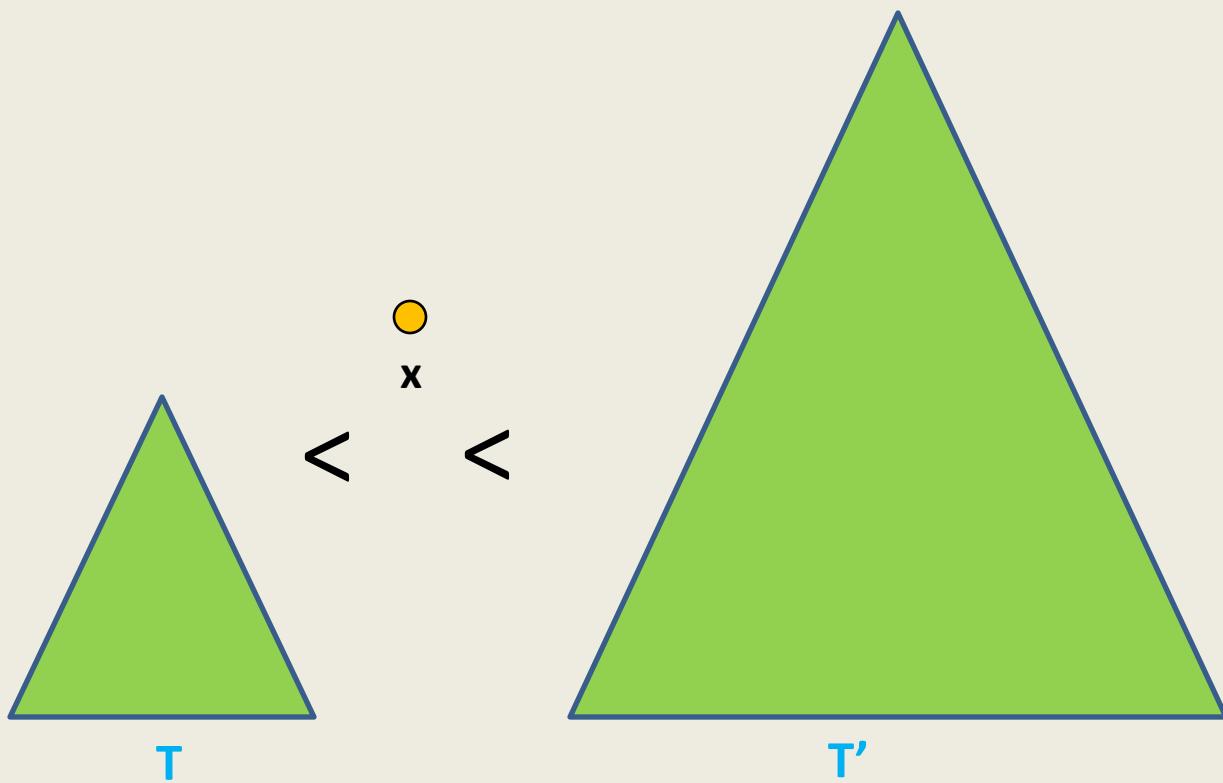
Simplified problem:

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

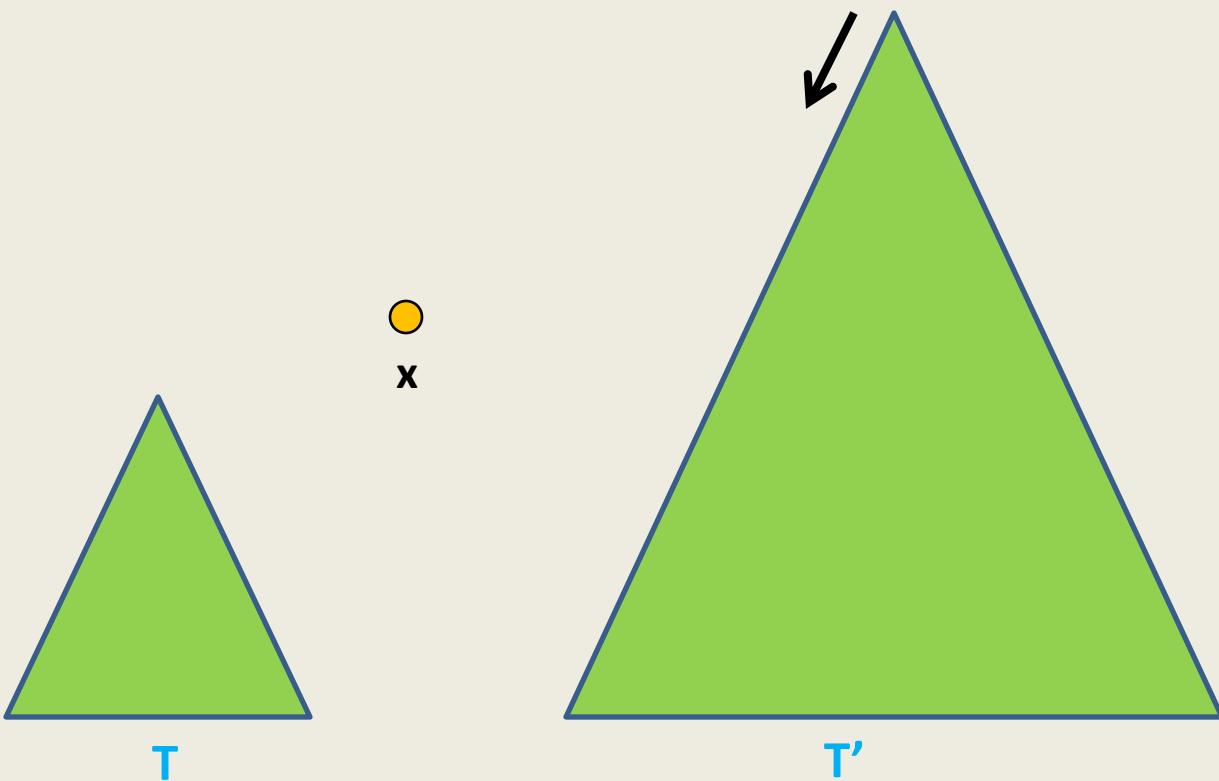
Extending the algorithm to the generic problem



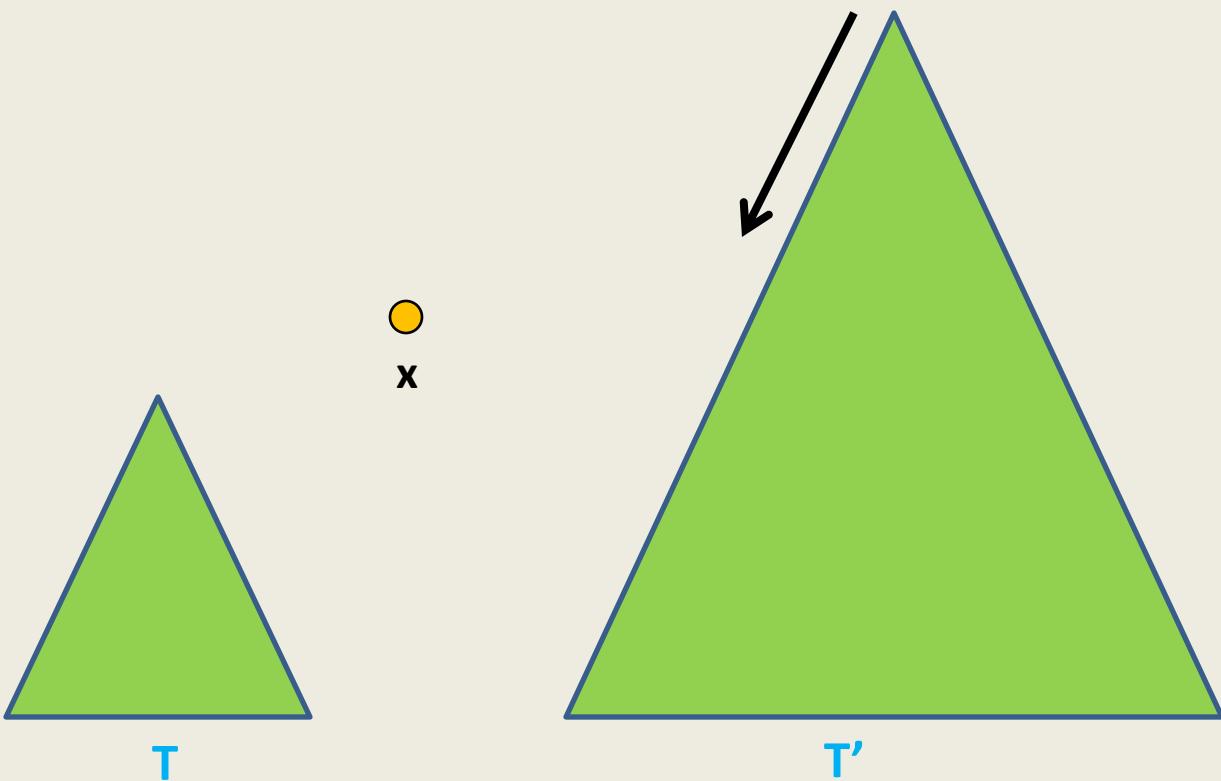
Extending the algorithm to the generic problem



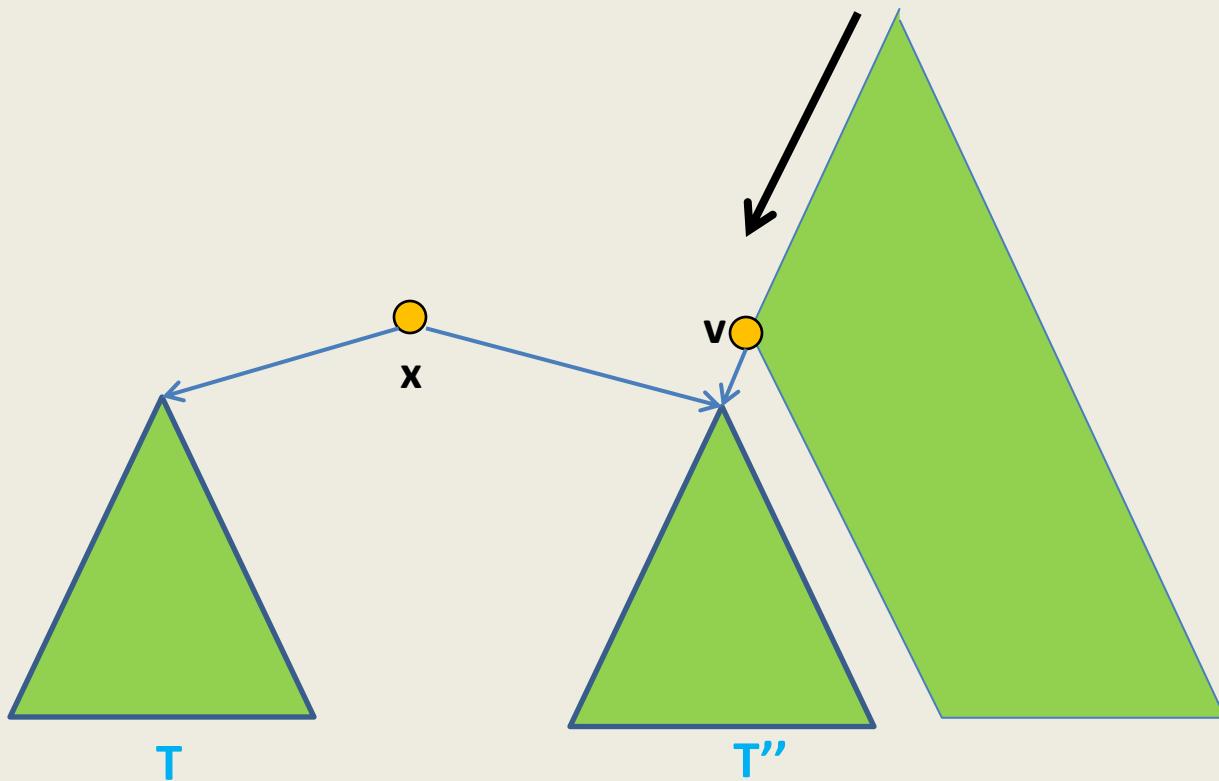
Extending the algorithm to the generic problem



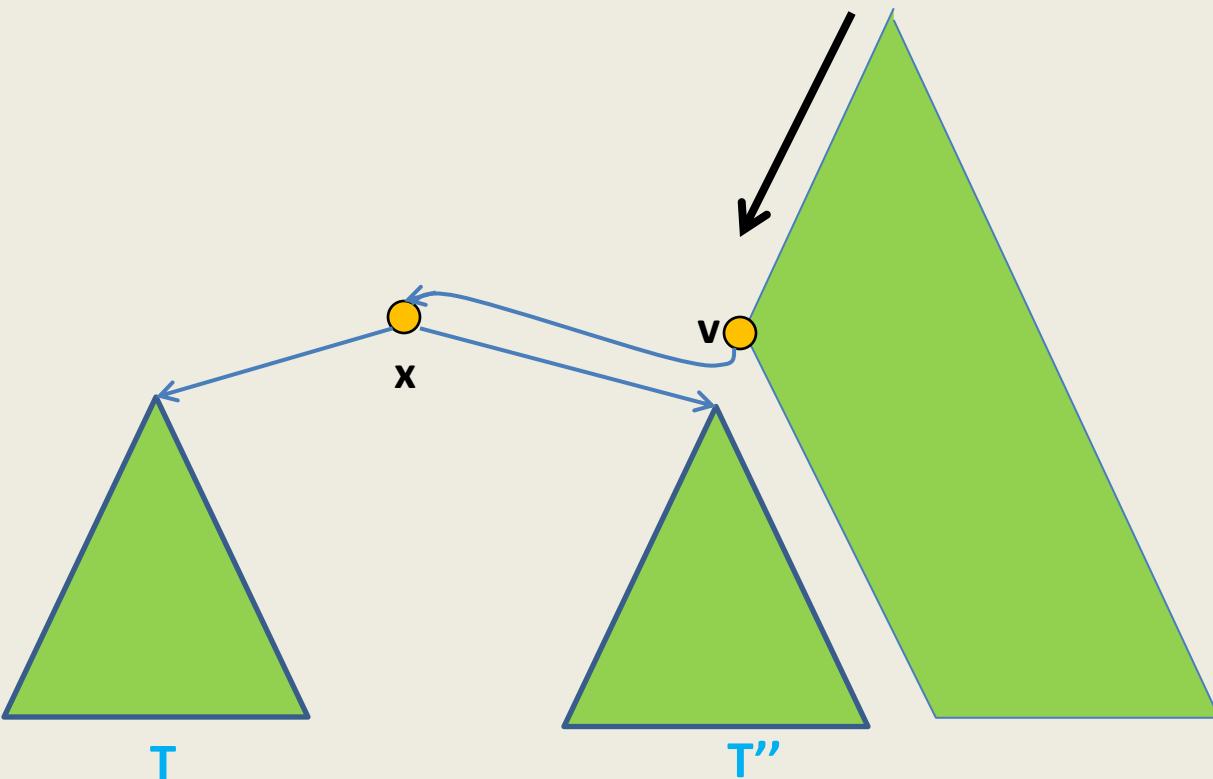
Extending the algorithm to the generic problem



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'

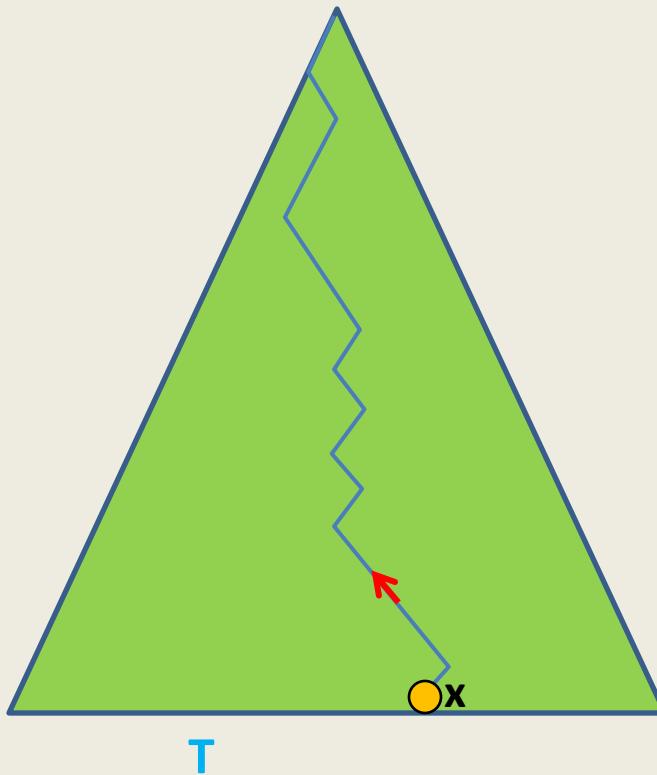
1. Keep following left pointer of T' until we reach a node v such that
 1. $\text{left}(v)$ is black
 2. The subtree T'' rooted at $\text{Left}(v)$ has black height same as that of T
2. $\text{left}(x) \leftarrow T;$
3. $\text{right}(x) \leftarrow T'';$
4. $\text{Color}(x) \leftarrow \text{red};$
5.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

Data Structures and Algorithms

(ESO207)

Lecture 21

- **Analyzing average running time of Quick Sort**

Overview of this lecture

Main Objective:

- Analyzing average time complexity of **QuickSort** using **recurrence**.
 - Using mathematical induction.
 - Solving the recurrence exactly.
- The outcome of this analysis will be quite surprising!

Extra benefits:

- You will learn a standard way of using mathematical induction to bound time complexity of an algorithm. You must try to internalize it.

QuickSort

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 ( $l < r$ )
         $i \leftarrow \text{Partition}(A, l, r);$ 
        QuickSort( $A, l, i - 1$ );
        QuickSort( $A, i + 1, r$ )
}
```

Partition :

$x \leftarrow A[l]$ as a pivot element,

permutes the subarray $A[l \dots r]$ such that
elements preceding x are smaller than x ,

$A[i] = x$,

and elements succeeding x are greater than x .

Analyzing average time complexity of QuickSort

Part 1

Deriving the recurrence

Analyzing average time complexity of QuickSort

Assumption (just for a neat analysis):

- All elements are distinct.
- Each recursive call selects the first element of the subarray as the pivot element.

Analyzing average time complexity of QuickSort

A useful Fact: **Quick sort** is a comparison based algorithm.

0	1	2	3	4	5	6	7	8
6	11	42	37	24	5	16	27	2
e_3	e_4	e_9	e_8	e_6	e_2	e_5	e_7	e_1

0	1	2	3	4	5	6	7	8
15	20	49	41	29	4	23	36	3
e_3	e_4	e_9	e_8	e_6	e_2	e_5	e_7	e_1

Let e_i : i th smallest element of \mathbf{A} .

Observation: The execution of **Quick sort** depends upon the permutation of e_i 's and not on the values taken by e_i 's.

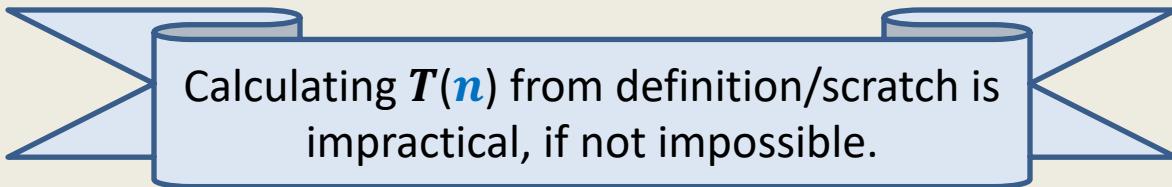
Analyzing average time complexity of QuickSort

$T(n)$: Average running time for Quick sort on input of size n .

(average over all possible permutations of $\{e_1, e_2, \dots, e_n\}$)

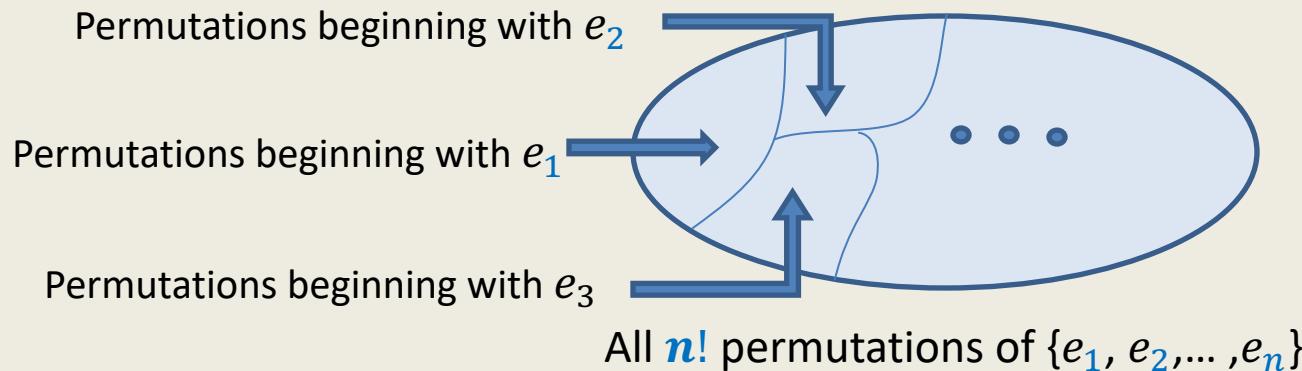
$$\text{Hence, } T(n) = \frac{1}{n!} \sum_{\pi} Q(\pi),$$

where $Q(\pi)$ is the time complexity (or no. of comparisons) when the input is permutation π .



Calculating $T(n)$ from definition/scratch is impractical, if not impossible.

Analyzing average time complexity of QuickSort



Let $P(i)$ be the set of all those permutations of $\{e_1, e_2, \dots, e_n\}$ that begin with e_i .

Question: What fraction of all permutations constitutes $P(i)$?

Answer: $\frac{1}{n}$

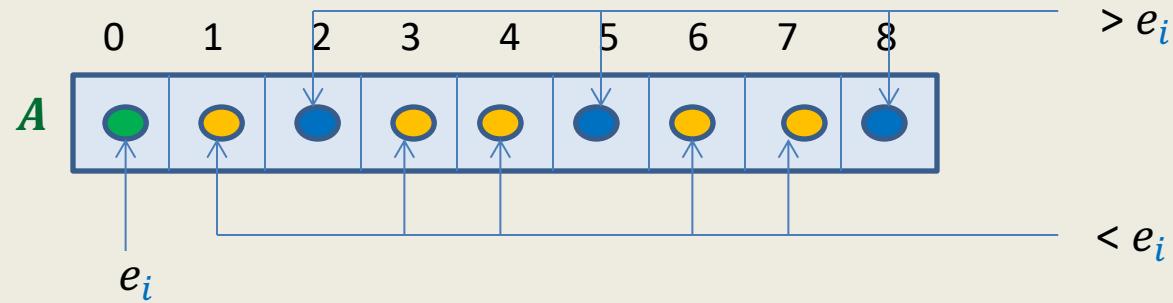
Let $G(n, i)$ be the average running time of QuickSort over $P(i)$.

Question: What is the relation between $T(n)$ and $G(n, i)$'s ?

Answer: $T(n) = \frac{1}{n} \sum_{i=1}^n G(n, i)$

Observation: We now need to derive an expression for $G(n, i)$. For this purpose, we need to have a closer look at the execution of QuickSort over $P(i)$.

Quick Sort on a permutation from $P(i)$.



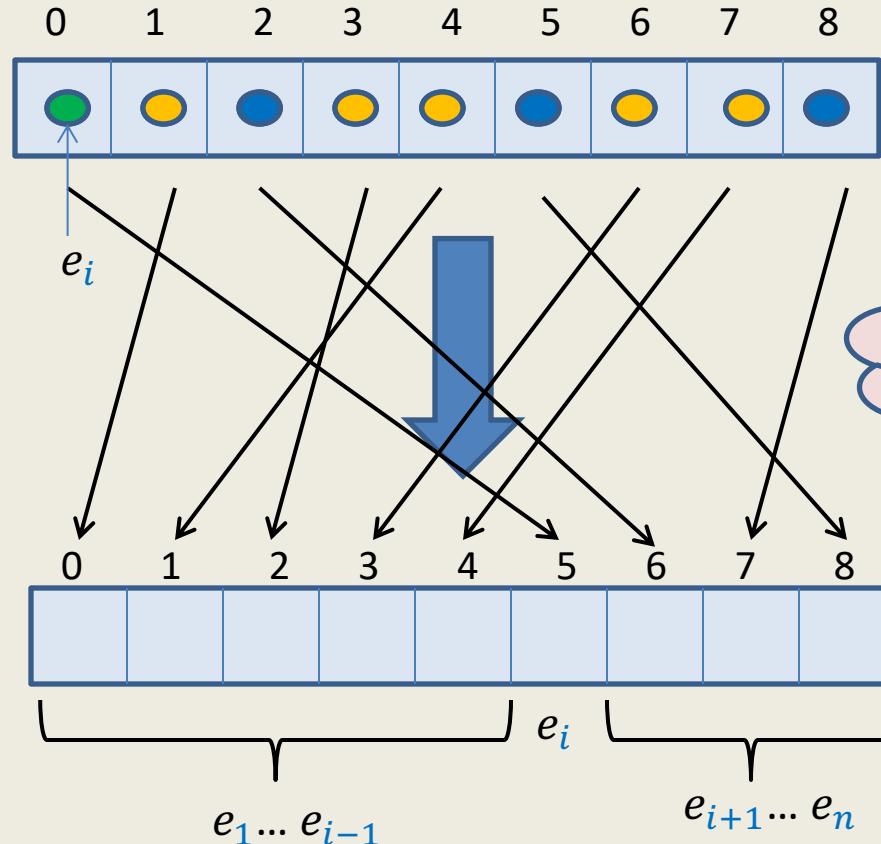
What happens during
Partition($A, 0, 8$)

Quick Sort on a permutation from $P(i)$.

$P(i)$ 
 $(n - 1)!$

Many-to-one
mapping


$S(i)$ 
 $(i - 1)! \times (n - i)!$



Reasons:
- `Partition()` is “well-defined”
- `Partition()` just compares **pivot** with other elements.

Lemma 1:

There are exactly $\binom{n-1}{i-1}$ permutations from $P(i)$ that get mapped to one permutation in $S(i)$.

$S(i)$: Permutations resulting from `Partition()`.

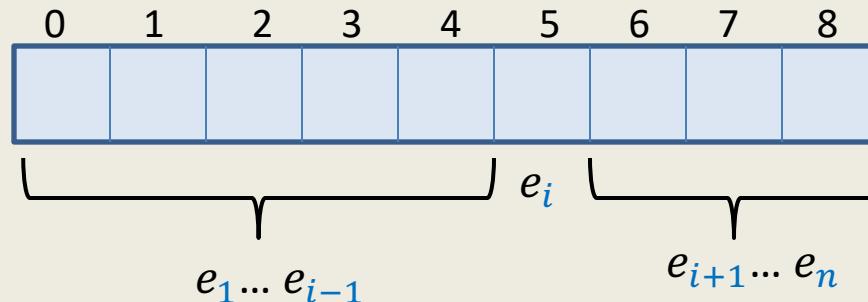
Quick Sort on a permutation from $P(i)$.

$P(i)$ 
 $(n - 1)!$



 Many-to-one mapping
↓

$S(i)$ 
 $(i - 1)! \times (n - i)!$



Using **Lemma 1** Can you now express $G(n, i)$ recursively ?

Lemma 1:

There are exactly $\binom{n-1}{i-1}$ permutations from $P(i)$ that get mapped to one permutation in $S(i)$.

Analyzing average time complexity of QuickSort

$$G(n, i) =$$

$$T(i - 1) + T(n - i) + dn \quad \text{----1}$$

We showed previously that :

$$T(n) = \frac{1}{n} \sum_{i=1}^n G(n, i) \quad \text{----2}$$

Question: Can you express $T(n)$ recursively using 1 and 2?

$$T(n) = \frac{1}{n} \sum_{i=1}^n (T(i - 1) + T(n - i)) + dn$$

$$T(1) = C$$

Analyzing average time complexity of QuickSort

Part 2

Solving the recurrence through
mathematical induction

$$T(1) = c$$

$$\begin{aligned} T(n) &= \frac{1}{n} \sum_{i=1}^n (T(i-1) + T(n-i)) + dn \\ &= \frac{2}{n} \sum_{i=1}^{n-1} T(i) + dn \end{aligned}$$

Assertion A(m): $T(m) \leq am \log m + b$ for all $m \geq 1$

Base case A(0): Holds for $b \geq c$

Induction step: Assuming A(m) holds for all $m < n$, we have to prove A(n).

$$\begin{aligned} T(n) &\leq \frac{2}{n} \sum_{i=1}^{n-1} (ai \log i + b) + dn \\ &\leq \frac{2}{n} \left(\sum_{i=1}^{n-1} ai \log i \right) + 2b + dn \\ &= \frac{2}{n} \left(\sum_{i=1}^{n/2} ai \log i + \sum_{i=\frac{n}{2}+1}^{n-1} ai \log i \right) + 2b + dn \\ &\leq \frac{2}{n} \left(\sum_{i=1}^{n/2} ai \log n/2 + \sum_{i=\frac{n}{2}+1}^{n-1} ai \log n \right) + 2b + dn \\ &= \frac{2}{n} \left(\sum_{i=1}^{n-1} ai \log n - \sum_{i=1}^{n/2} ai \right) + 2b + dn \\ &= \frac{2}{n} \left(\frac{n(n-1)}{2} a \log n - \frac{\frac{n}{2}(\frac{n}{2}+1)}{2} a \right) + 2b + dn \\ &\leq a(n-1) \log n - \frac{n}{4} a + 2b + dn \\ &\leq an \log n + b - \frac{n}{4} a + b + dn \\ &\leq an \log n + b \quad \text{for } a > 4(b+d) \end{aligned}$$

Analyzing average time complexity of QuickSort

Part 3

Solving the recurrence exactly

Some elementary tools

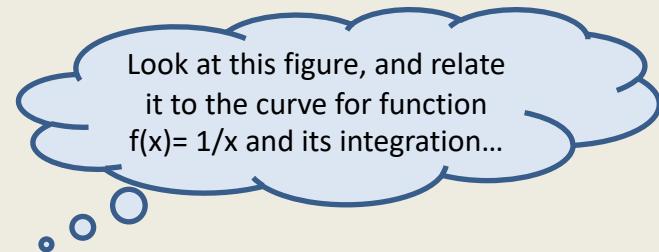
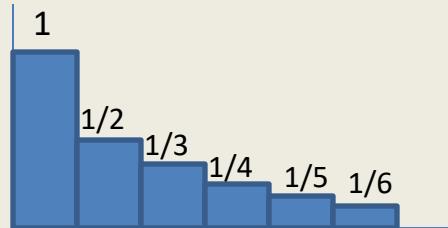
$$H(n) = \sum_{i=1}^n \frac{1}{i}$$

Question: How to approximate $H(n)$?

Answer: $H(n) \rightarrow \log_e n + \gamma$, as n increases

where γ is Euler's constant ~ 0.58

Hint: →



We shall calculate average number of comparisons during **QuickSort** using:

- our knowledge of solving recurrences by substitution
- our knowledge of solving recurrence by unfolding
- our knowledge of simplifying a partial fraction (from JEE days)

Students should try to internalize the way the above tools are used.

$T(n)$: average number of comparisons during **QuickSort** on n elements.

$$T(1) = 0, \quad T(0) = 0,$$

$$\begin{aligned} T(n) &= \frac{1}{n} \sum_{i=1}^n (T(i-1) + T(n-i)) + n - 1 \\ &= \frac{2}{n} \sum_{i=1}^n (T(i-1)) + n - 1 \end{aligned}$$

$$\rightarrow nT(n) = 2 \sum_{i=1}^n (T(i-1)) + n(n-1) \quad \text{----1}$$

Question: How will this equation appear for $n-1$?

$$(n-1)T(n-1) = 2 \sum_{i=1}^{n-1} (T(i-1)) + (n-1)(n-2) \quad \text{----2}$$

Subtracting 2 from 1, we get

$$nT(n) - (n-1)T(n-1) = 2T(n-1) + 2(n-1)$$

$$\rightarrow nT(n) - (n+1)T(n-1) = 2(n-1)$$

Question: How to solve/simplify it further ?

$$\frac{T(n)}{n+1} - \frac{T(n-1)}{n} = \frac{2(n-1)}{n(n+1)}$$

$$\frac{T(n)}{n+1} - \frac{T(n-1)}{n} = \frac{2(n-1)}{n(n+1)}$$

$$\rightarrow g(n) - g(n-1) = \frac{2(n-1)}{n(n+1)}, \quad \text{where } g(m) = \frac{T(m)}{m+1}$$

Question: How to simplify RHS ?

$$\frac{2(n-1)}{n(n+1)} = \frac{2(n+1)-4}{n(n+1)} =$$

$$= \frac{2}{n} - \frac{4}{n(n+1)}$$

$$= \frac{2}{n} - \frac{4}{n} + \frac{4}{n+1}$$

$$= \frac{4}{n+1} - \frac{2}{n}$$

$$\rightarrow g(n) - g(n-1) = \frac{4}{n+1} - \frac{2}{n}$$

$$g(n) - g(n-1) = \frac{4}{n+1} - \frac{2}{n}$$

Question: How to calculate $g(n)$?

$$g(n-1) - g(n-2) = \frac{4}{n} - \frac{2}{n-1}$$

$$g(n-2) - g(n-3) = \frac{4}{n-1} - \frac{2}{n-2}$$

$$\dots \qquad \qquad \qquad = \dots$$

$$g(2) - g(1) = \frac{4}{3} - \frac{2}{2}$$

$$g(1) - g(0) = \frac{4}{2} - \frac{2}{1}$$

$$\begin{aligned} \text{Hence } g(n) &= \frac{4}{n+1} + (2 \sum_{j=2}^n \frac{1}{j}) - 2 = \frac{4}{n+1} + (2 \sum_{j=1}^n \frac{1}{j}) - 4 \\ &= \frac{4}{n+1} + 2H(n) - 4 \end{aligned}$$

$$\begin{aligned} \rightarrow T(n) &= (n+1) (\frac{4}{n+1} + 2H(n) - 4) \\ &= 2(n+1)H(n) - 4n \end{aligned}$$

$$\begin{aligned}
 T(n) &= 2(n+1)H(n) - 4n \\
 &= 2(n+1) \log_e n + 1.16(n+1) - 4n \\
 &= 2n \log_e n - 2.84n + O(1) \\
 &= 2n \log_e n
 \end{aligned}$$

Theorem: The average number of comparisons during **QuickSort** on n elements approaches $2n \log_e n - 2.84n$.

$$= 1.39n \log_2 n - O(n)$$

The best case number of comparisons during **QuickSort** on n elements = $n \log_2 n$

The worst case no. of comparisons during **QuickSort** on n elements = $n(n-1)$

Quick sort versus Merge Sort

No. of Comparisons	Merge Sort	Quick Sort
Average case	$n \log_2 n$	$1.39 n \log_2 n$
Best case	$n \log_2 n$	$n \log_2 n$
Worst case	$n \log_2 n$	$n(n - 1)$

After seeing this table, no one would prefer Quick sort to Merge sort

But **Quick sort** is still the most preferred algorithm in practice. Why ?

Data Structures and Algorithms

(ESO207)

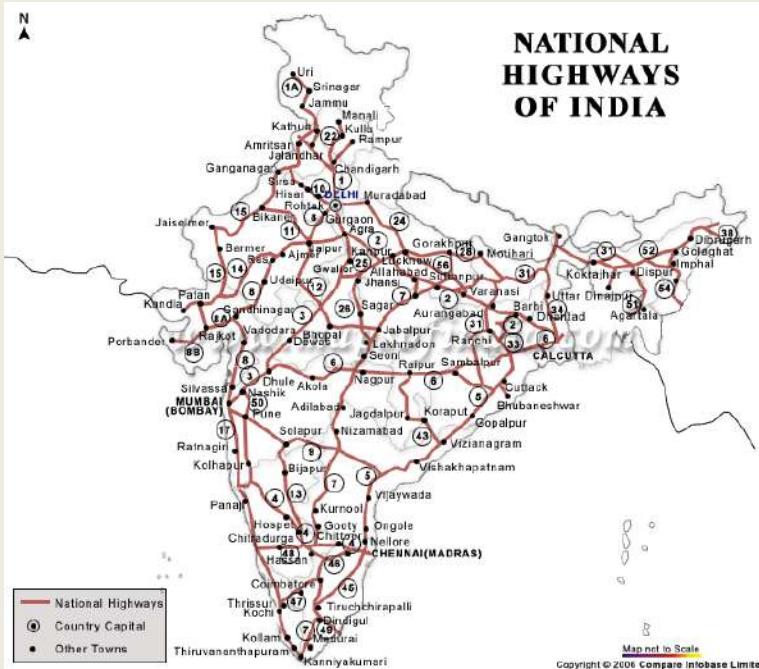
Lecture 22

Graphs

- Notations and terminologies
- Data structures for graphs
- A few algorithmic problems in graphs

Why Graphs ??

Finding shortest route between cities



Given a network of **roads** connecting various cities,
compute the shortest route between any two **cities**.

Just imagine how you would solve/approach this problem.

Embedding an integrated circuit on mother board

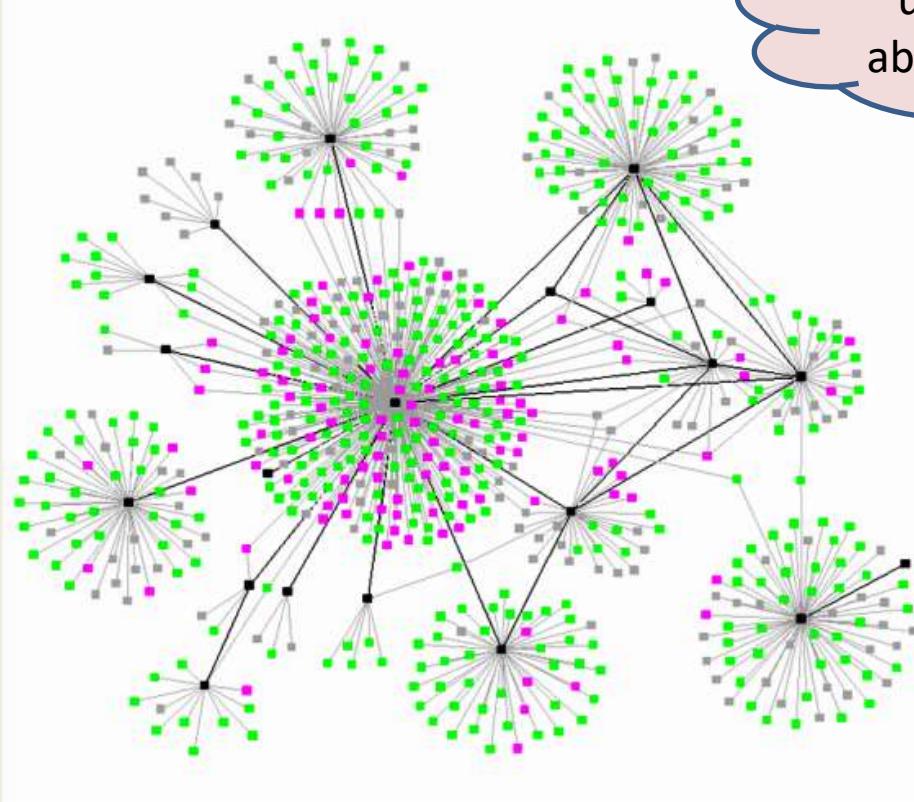


How to embed **ports** of various ICs on a plane and make **connections** among them so that

- No two connections intersect each other
- The total length of all the connections is minimal

A social network or world wide web (WWW)

Can we make some useful observations about such networks ?



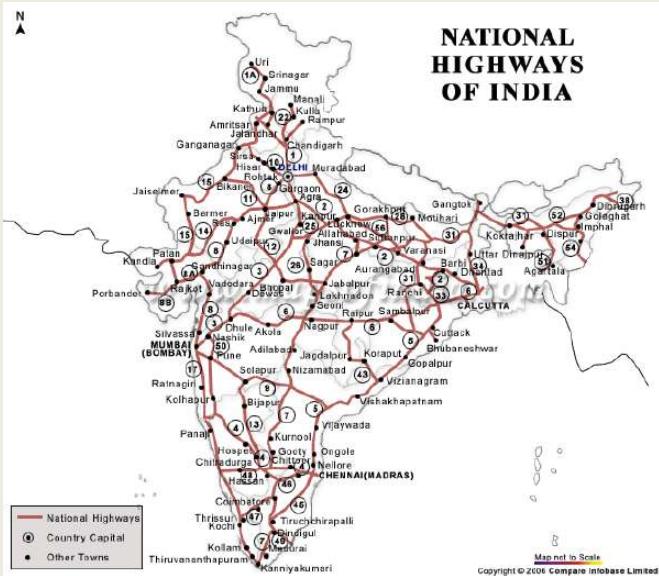
diameter

degree distribution

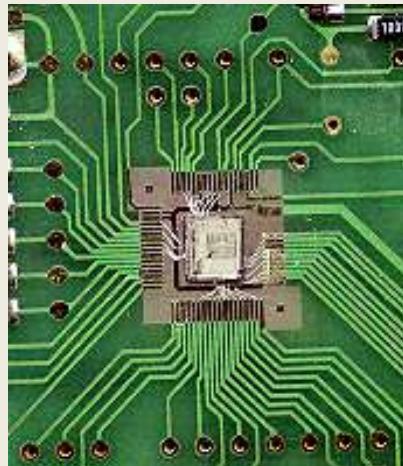
Do you know about the “6 degree of separation principle” of the world ?

Visit the site https://en.wikipedia.org/wiki/Six_degrees_of_separation

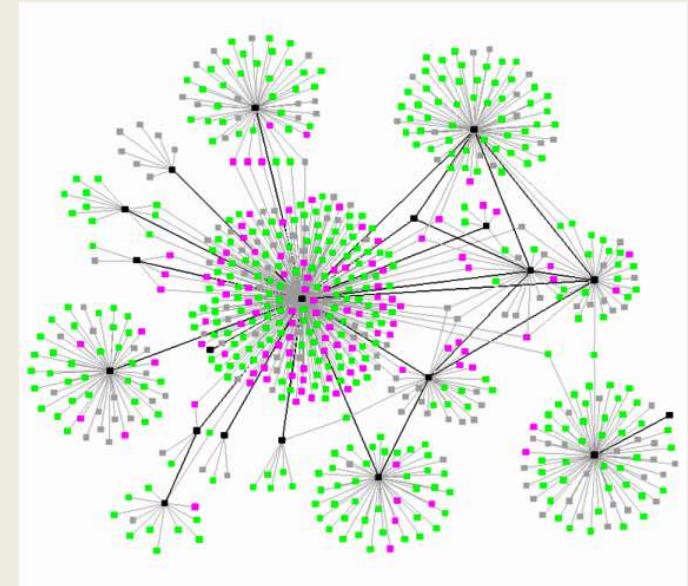
How will you model these problems ?



I

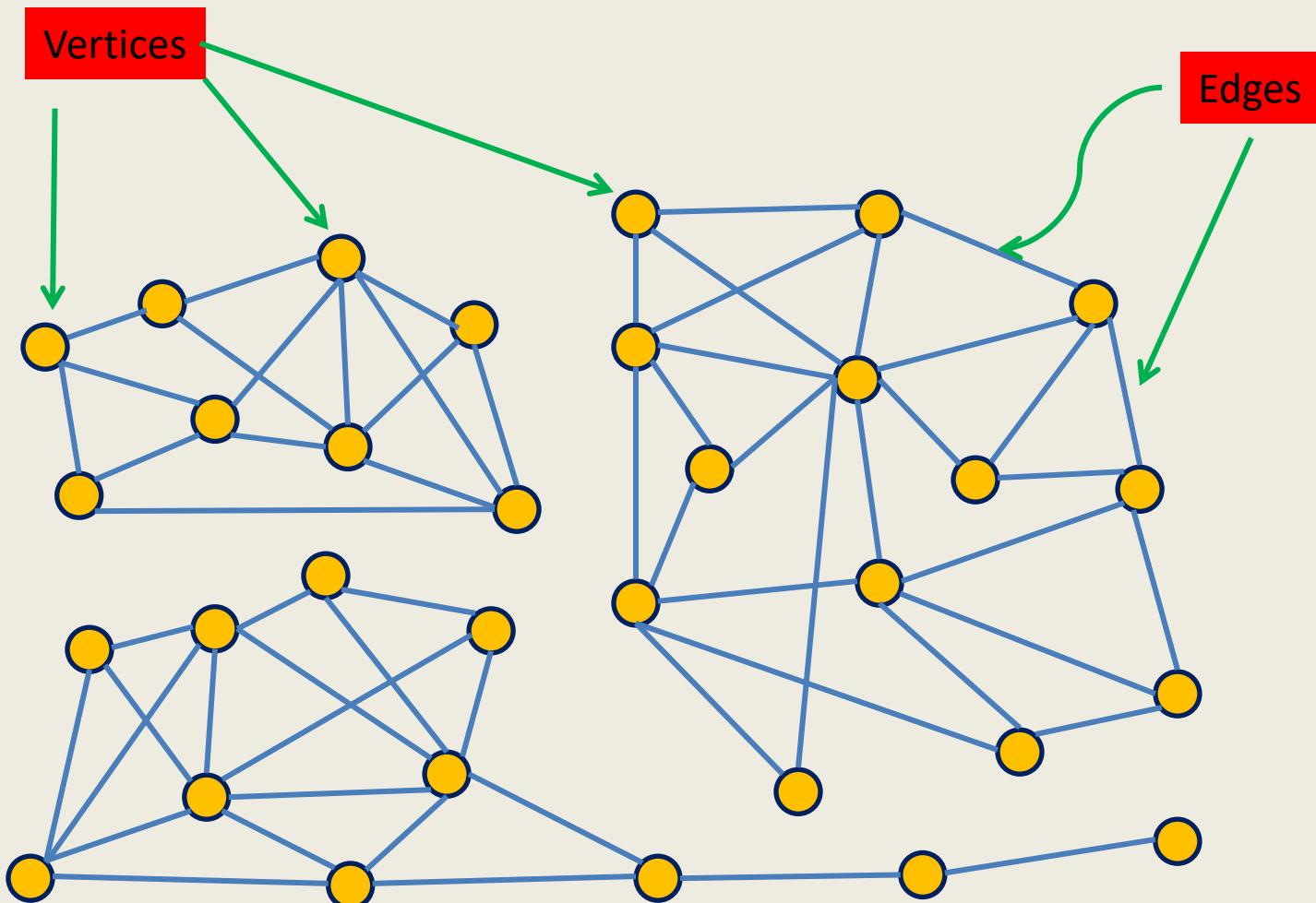


II



III

Graph



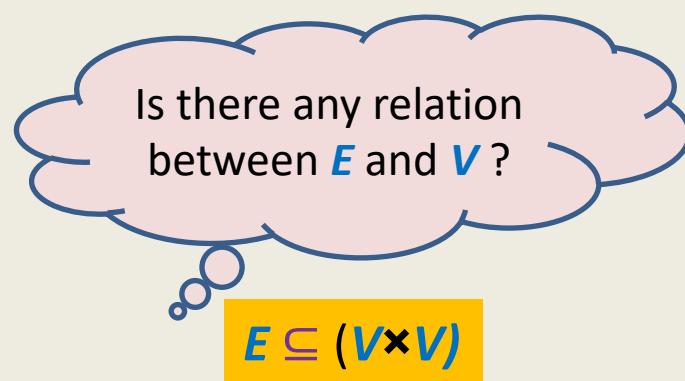
Graph

Definitions, notations, and terminologies

Graph

A graph G is defined by two sets

- V : set of vertices
- E : set of edges

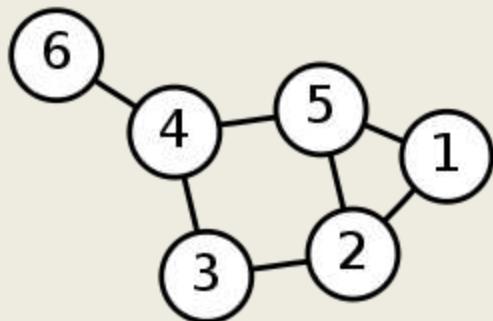


Notation:

- A graph G consisting of vertices V and edges E is denoted by (V, E)

Types of graphs

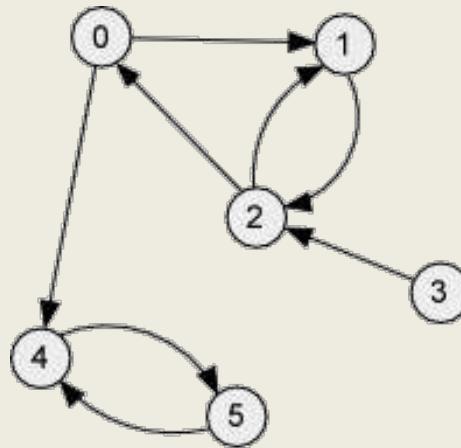
Undirected Graph



$$V = \{1, 2, 3, 4, 5, 6\}$$

$$E = \{(1,2), (1,5), (2,5), (2,3), (3,4), (4,5), (4,6)\}$$

Directed Graph



$$V = \{0, 1, 2, 3, 4, 5\}$$

$$E = \{ (0,1), (0,4), (1,2), (2,0), (2,1), (3,2), (4,5), (5,4) \}$$

Notations

Notations:

- $n = |V|$
- $m = |E|$

Note: For directed graphs, $m \leq n(n-1)$

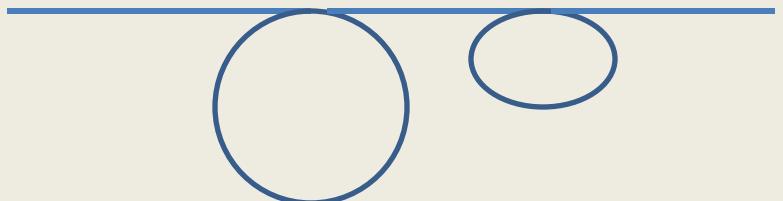
For undirected graphs, $m \leq n(n-1)/2$

Walks, paths, and cycles

Walk:

A sequence $\langle v_0, v_1, \dots, v_k \rangle$ of vertices is said to be a **walk** from x to y

- $x = v_0$
- $y = v_k$
- For each $i < k$, $(v_i, v_{i+1}) \in E$



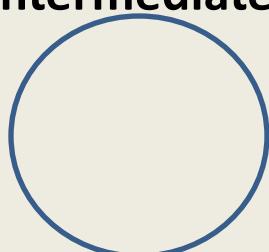
Path:

A walk $\langle v_0, v_1, \dots, v_k \rangle$ on which no vertex appears twice.

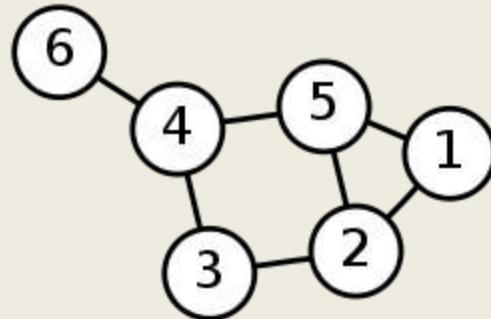


Cycle:

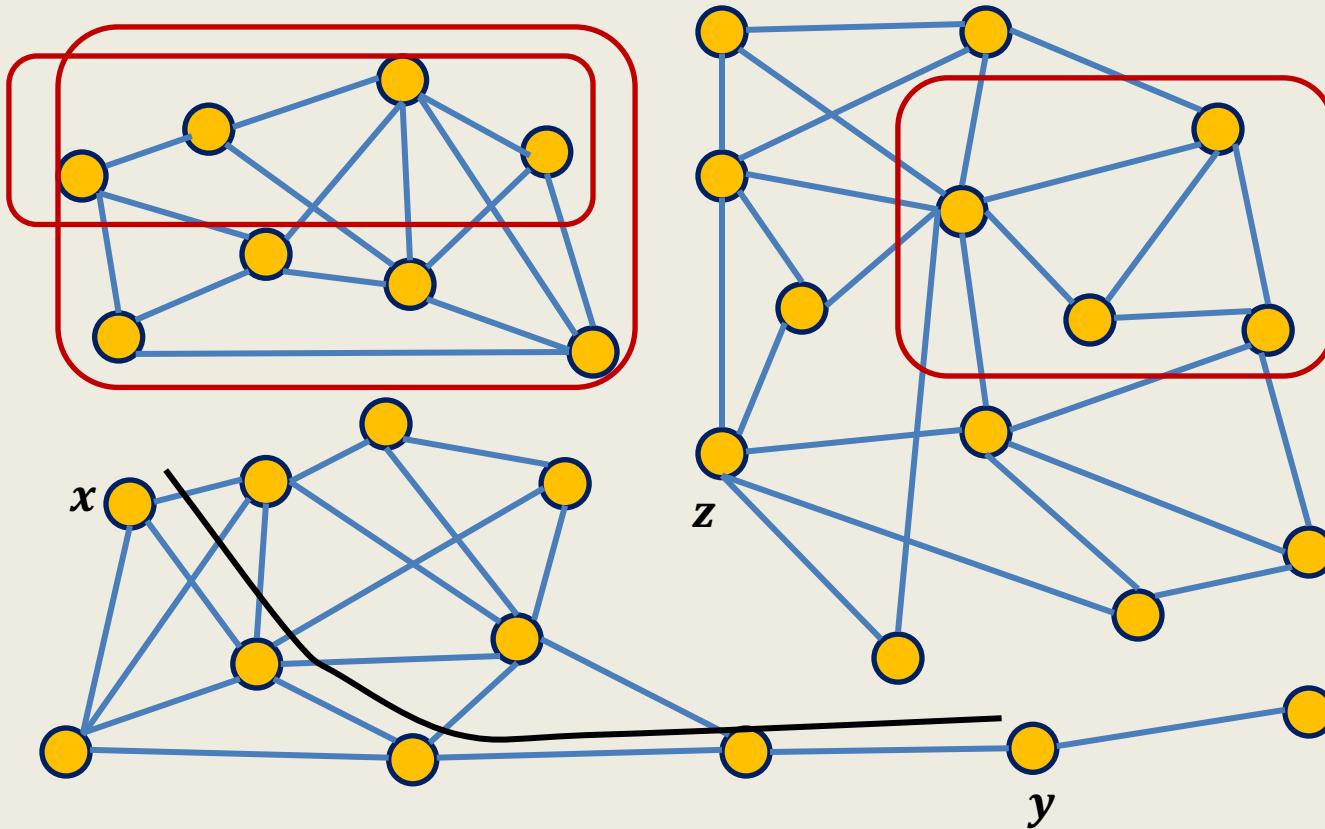
A walk $\langle v_0, v_1, \dots, v_k \rangle$ where no **intermediate** vertex gets repeated and $v_0 = v_k$



Examples



- $\langle 1, 5, 4 \rangle$ is a **walk** from **1** to **4**.
- $\langle 1, 3, 2, 5 \rangle$ is **not** a **walk**.
- $\langle 1, 2, 5, 2, 3, 4, 5, 4, 6 \rangle$ is a **walk** from **1** to **6**.
- $\langle 1, 2, 5, 4, 6 \rangle$ is a **path** from **1** to **6**.
- $\langle 2, 3, 4, 5, 2 \rangle$ is a **cycle**.



two vertices are said to be ***connected*** if there is a **path** between them

Connected component:

A **maximal** subset of connected vertices

You can not add any more vertex to the subset
and still keep it connected.

Data Structures for Graphs

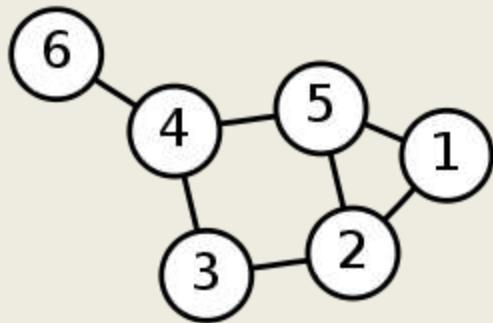
Vertices are always numbered

1, ..., n

Or **0, ..., $n - 1$**

Link based data structure for graph

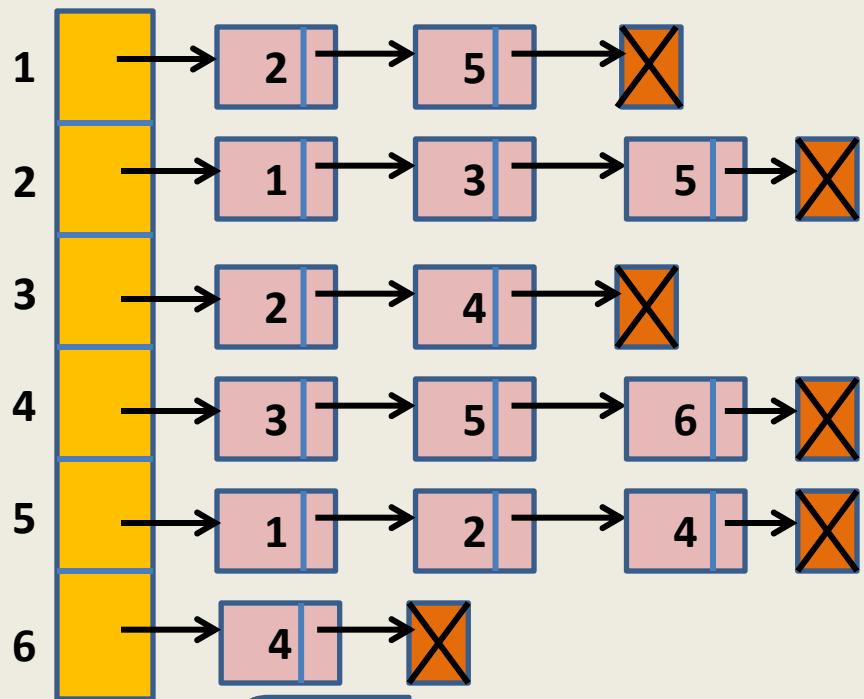
Undirected Graph



$$V = \{1, 2, 3, 4, 5, 6\}$$

$$E = \{(1, 2), (1, 5), (2, 5), (2, 3), (3, 4), (4, 5), (4, 6)\}$$

Adjacency Lists



Size = $O(n + m)$

Link based data structure for graph

Advantage of Adjacency Lists :

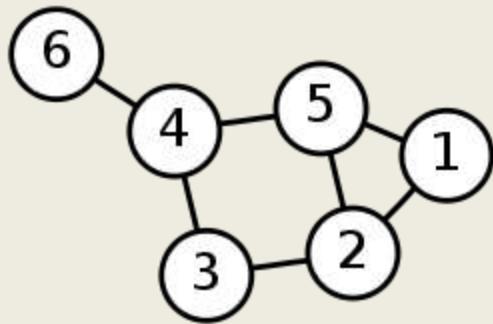
- Space efficient
- Computing all the neighbors of a vertex in optimal time.

Disadvantage of Adjacency Lists :

- How to determine if there is an edge from x to y ?
($O(n)$ time in the worst case).

Array based data structure for graph

Undirected Graph



$$V = \{1, 2, 3, 4, 5, 6\}$$

$$E = \{(1, 2), (1, 5), (2, 5), (2, 3), (3, 4), (4, 5), (4, 6)\}$$

Adjacency Matrix

	1	2	3	4	5	6
1	0	1	0	0	1	0
2	1	0	1	0	1	0
3	0	1	0	1	0	0
4	0	0	1	0	1	1
5	1	1	0	1	0	0
6	0	0	0	1	0	0

Size = $O(n^2)$

Array based data structure for graph

Advantage of Adjacency Matrix :

- Determining whether there is an edge from x to y in $O(1)$ time for any two vertices x and y .

Disadvantage of Adjacency Matrix :

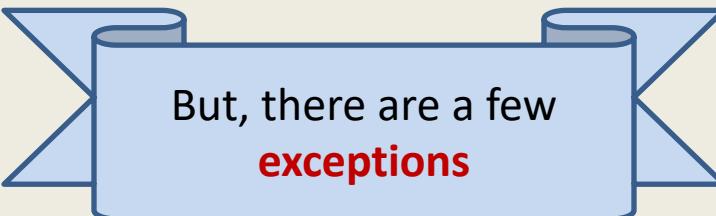
- Computing all neighbors of a given vertex x in $O(n)$ time
- It takes $O(n^2)$ space.

Which data structure is commonly used for storing graphs ?

Adjacency lists

Reasons:

- Graphs in real life are sparse ($m \ll n^2$).
 - Most algorithms require processing neighbors of each vertex.
- Adjacency matrix will enforce $O(n^2)$ bound on time complexity for such algorithm.



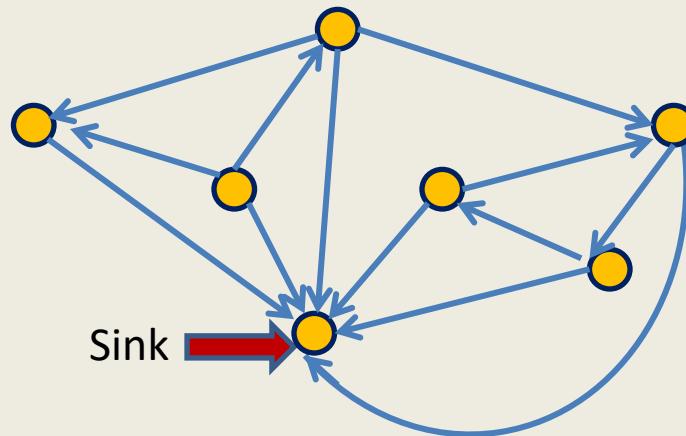
But, there are a few
exceptions

An interesting problem

(Finding a sink)

A vertex x in a given directed graph is said to be a **sink** if

- There is no edge **emanating** from (leaving) x
- Every other vertex has an edge **into** x .



Given a directed graph $G=(V,E)$ in an **adjacency matrix** representation, design an $O(n)$ time algorithm to determine if there is any **sink** in G .

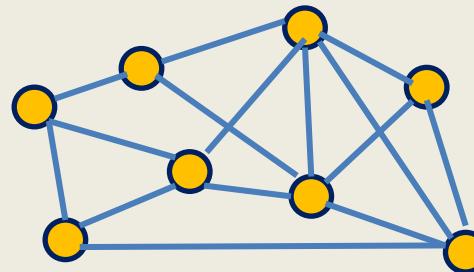
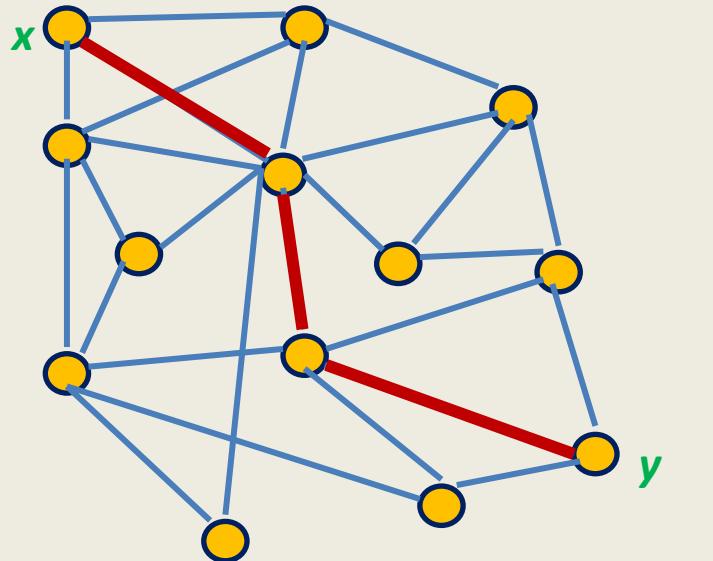
Graph traversal

Topic for the next class

Graph traversal

Definition:

A vertex y is said to be reachable from x if there is a **path** from x to y .



Graph traversal from vertex x : Starting from a given vertex x , the aim is to visit all vertices which are reachable from x .

Non-triviality of graph traversal

- **Avoiding loop:**

How to avoid visiting a vertex multiple times ?

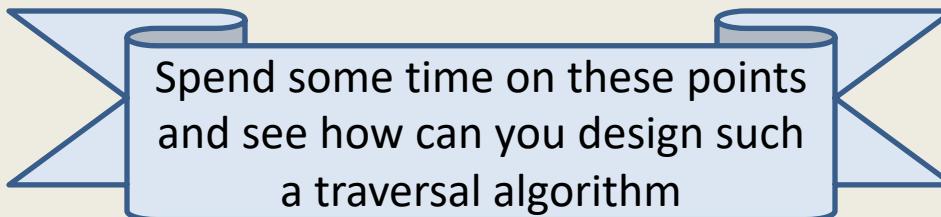
(keeping track of vertices already visited)

- **Finite number of steps :**

The traversal **must stop** in finite number of steps.

- **Completeness :**

We must visit **all** vertices reachable from the start vertex **x**.



A sample of Graph algorithmic Problems

- Are two vertices x and y connected ?
- Find all connected components in a graph.
- Is there is a cycle in a graph ?
- Compute a path of shortest length between two vertices ?
- Is there is a cycle passing through all vertices ?

Data Structures and Algorithms

(ESO207)

Lecture 23

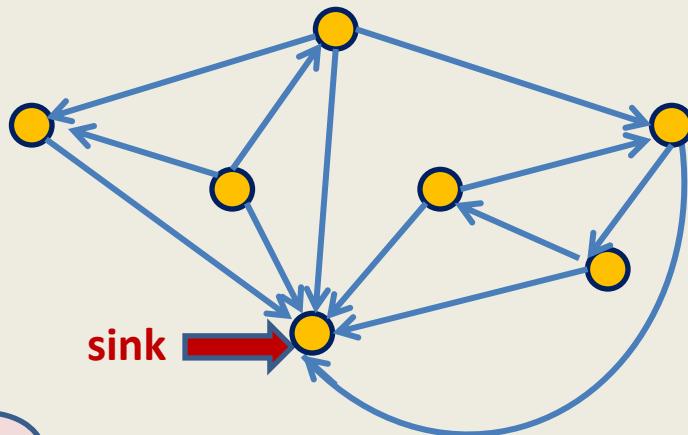
- **Finding a sink in a directed graph**
- **Graph Traversal**
 - **Breadth First Search** Traversal and its simple applications

An interesting problem

(Finding a **sink**)

Definition: A vertex x in a given directed graph is said to be a **sink** if

- There is no edge **emanating from** (leaving) x
- Every other vertex has an edge **into** x .



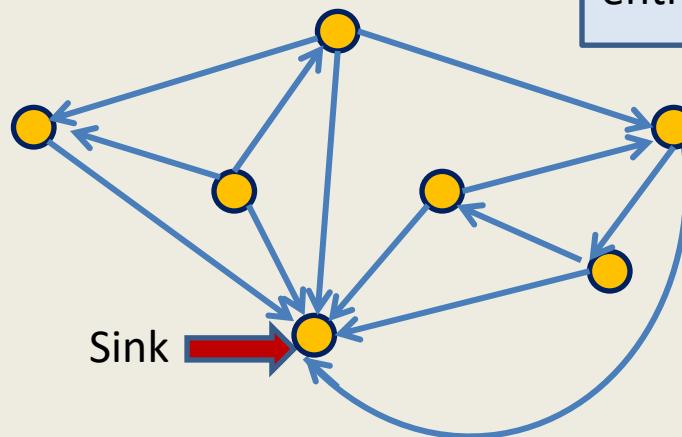
How many
sinks can there
be in G ?

At most 1.

An interesting problem

(Finding a sink)

Problem: Given a directed graph $G=(V,E)$ in an **adjacency matrix** representation, design an $O(n)$ time algorithm to determine if there is any **sink** in G .



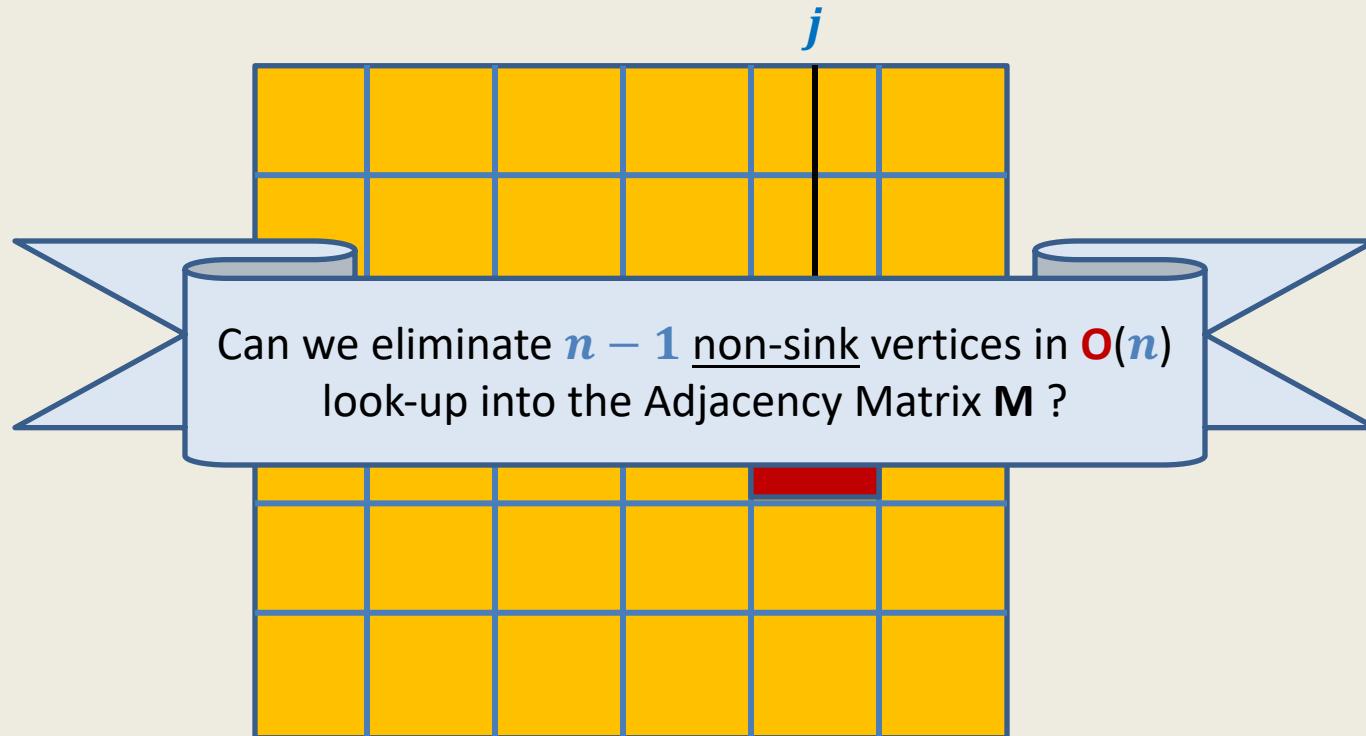
We are allowed to look into only $O(n)$ entries of the **Adjacency matrix M**.

Question: Can we verify efficiently whether any given vertex i is a sink ?

Answer: Yes, in $O(n)$ time only ☺

Look at i th row and i th column of **M**.

Key idea



If $\mathbf{M}[i, j] = 0$, then j can not be sink

If $\mathbf{M}[i, j] = 1$, then i can not be sink



Algorithm to find a **sink** in a graph

Key ideas:

- Looking at a single entry in **M** allows us to discard one vertex from being a sink.
- It takes **O(n)** time to verify if a vertex **i** is a sink.

Find-Sink(M) // **M** is the adjacency matrix of the given directed graph.

s \leftarrow 0;

For(**i**=1 to **n** – 1)

{

If (**M[s,i]** = ?)?...;

}

Verify if s is a sink and output accordingly.

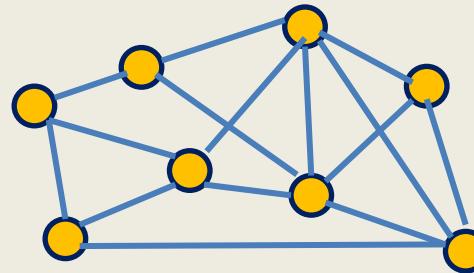
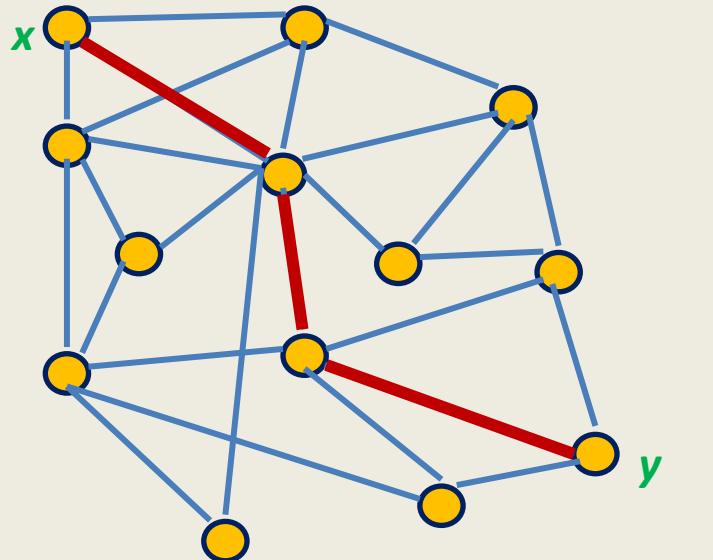
(Fill in the details of this pseudo code as a **Homework.**)

What is Graph traversal ?

Graph traversal

Definition:

A vertex y is said to be reachable from x if there is a **path** from x to y .



Graph traversal from vertex x :

Starting from a given vertex x , the aim is
to visit all vertices which are reachable from x .

Non-triviality of graph traversal

- **Avoiding loop:**
How to avoid visiting a vertex multiple times ?
(keeping track of vertices already visited)
- **Finite number of steps :**
The traversal **must stop** in finite number of steps.
- **Completeness :**
We must visit **all** vertices reachable from the start vertex **x**.

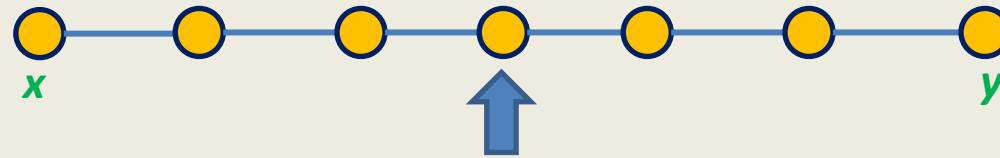
Breadth First Search traversal

We shall introduce this traversal technique through an interesting problem.

computing distances from a vertex.

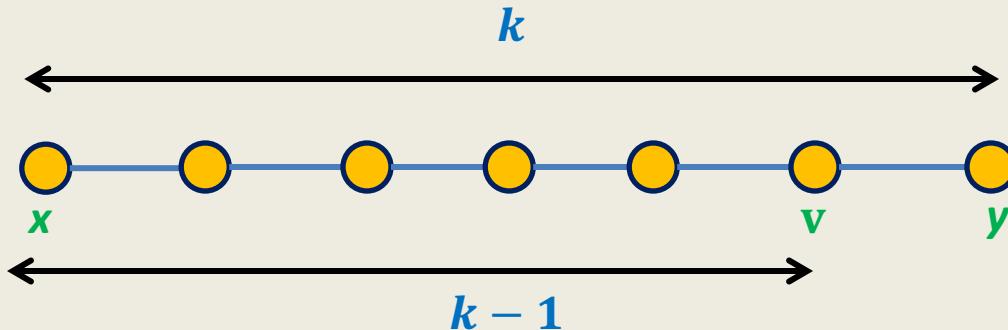
Notations and Observations

Length of a path: the number of edges on the path.



A path of length 6 between x and y

Notations and Observations



Observation:

If $\langle x, \dots, v, y \rangle$ is a path of length k from x to y ,
then what is the length of the path $\langle x, \dots, v \rangle$?

Answer: $k - 1$

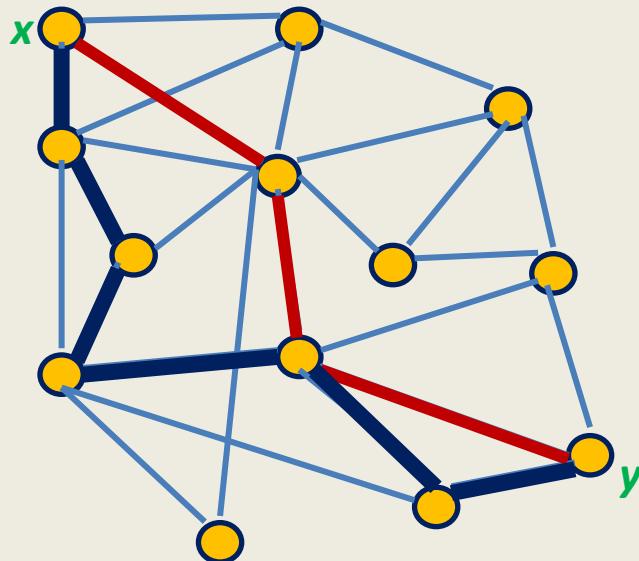
Question: What can be the maximum length of any path in a graph ?

Answer: $n - 1$

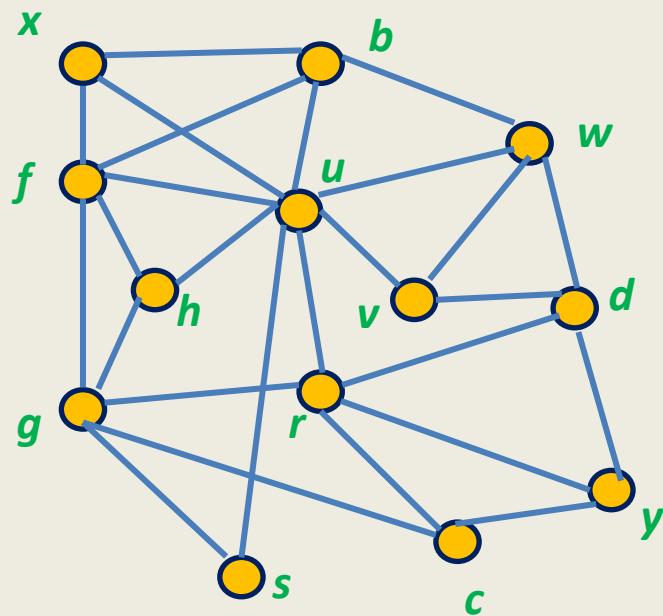
Notations and Observations

Shortest Path from x to y : A path from x to y of least length

Distance from x to y : the length of the shortest path from x to y .



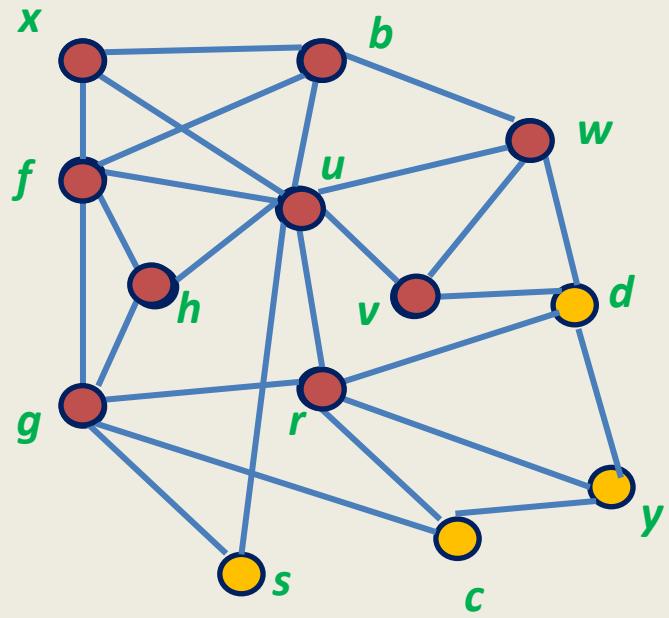
Shortest Paths in Undirected Graphs



Problem:

How to compute distance to all vertices
reachable from *x* in a given undirected graph ?

Shortest Paths in Undirected Graphs



V_0 : Vertices at distance 0 from x :

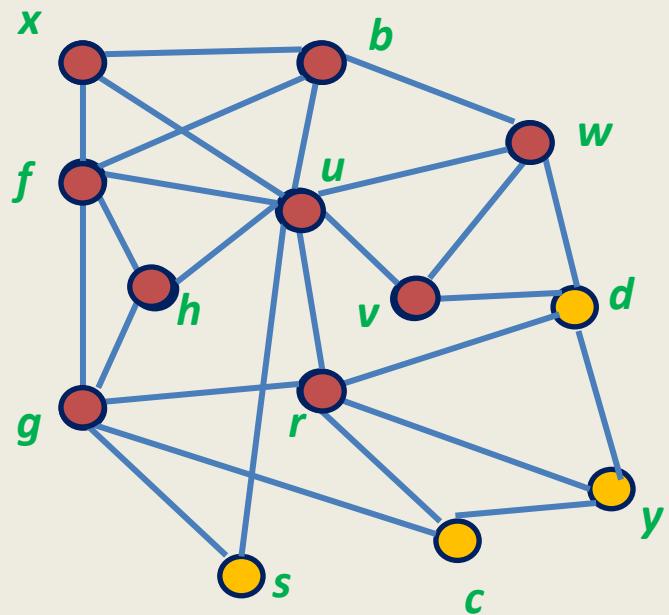
V_1 : Vertices at distance 1 from x :

V_2 : Vertices at distance 2 from x :

Why ?

While reporting V_2 , you have (sub)consciously used an **important property** of shortest paths.
Can you state this property ?

Shortest Paths in Undirected Graphs



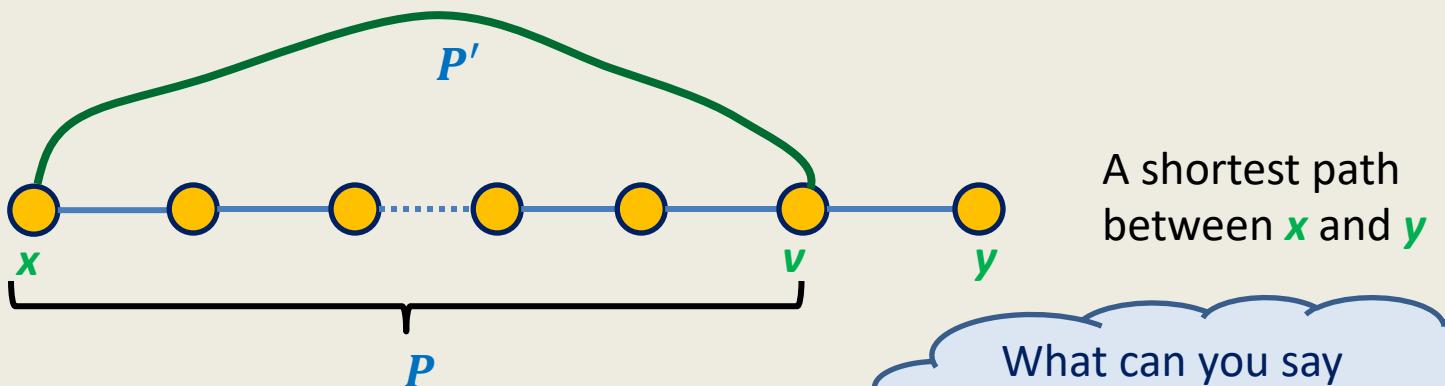
V_0 : Vertices at distance **0** from x :

V_1 : Vertices at distance **1** from x :

V_2 : Vertices at distance 2 from x :

Why ?

An important property of shortest paths



Observation:

If $\langle x, \dots, v, y \rangle$ is a shortest path from x to y ,
then $\langle x, \dots, v \rangle$ is also a shortest path.

Proof:

Suppose $P = \langle x, \dots, v \rangle$ is not a shortest path between x and v .

Then let P' be a shortest path between x and v .

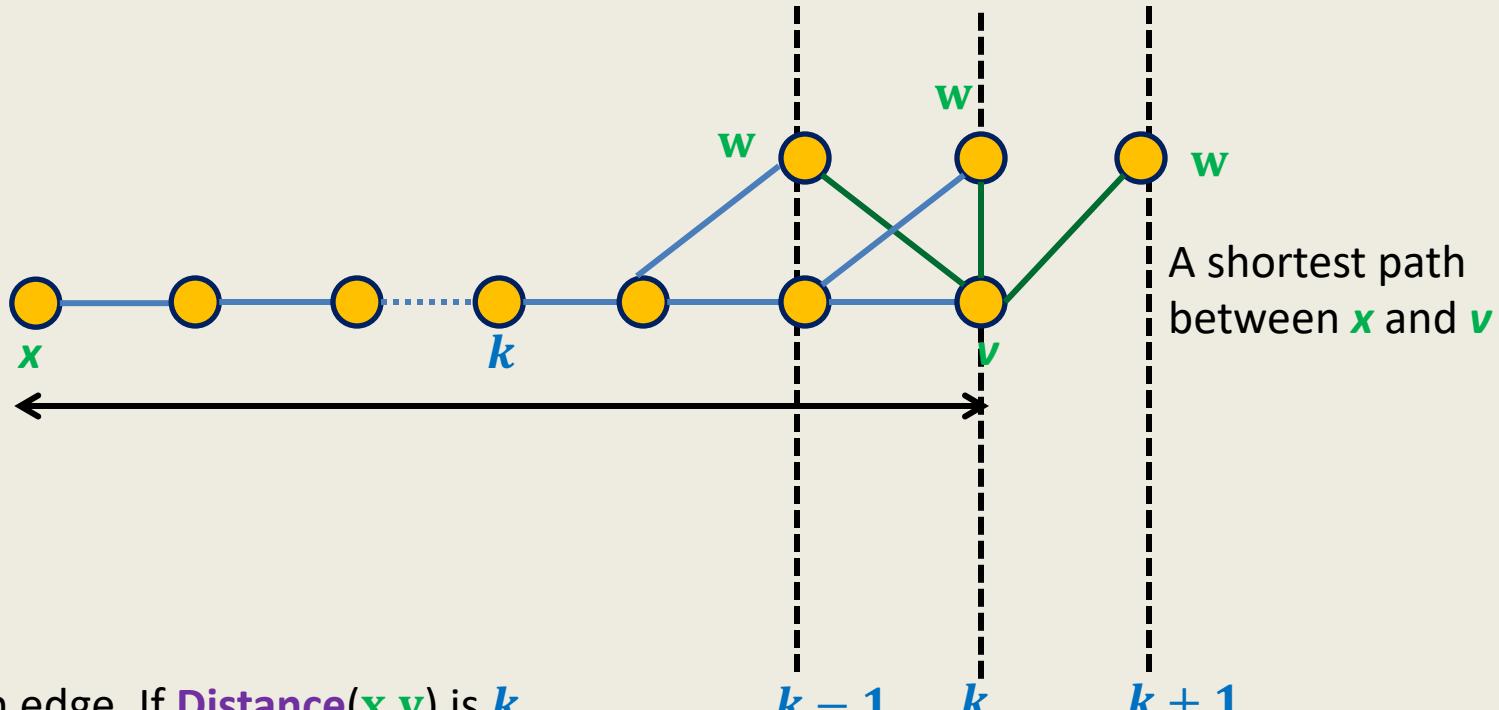
$\text{Length}(P') < \text{Length}(P)$.

Question: What happens if we concatenate P' with edge (v, y) ?

Answer: a path between x and y shorter than the shortest-path $\langle x, \dots, v, y \rangle$.

→ Contradiction.

An important question



Question:

Let (v,w) be an edge. If $\text{Distance}(x,v)$ is k ,

then what can be $\text{Distance}(x,w)$?

Answer: an element from the set $\{k - 1, k, k + 1\}$ only.

Relationship among vertices at different distances from x

V_0 : Vertices at distance **0** from $x = \{x\}$

V_1 : Vertices at distance **1** from $x =$

Neighbors of V_0

V_2 : Vertices at distance **2** from $x =$

Those Neighbors of V_1 which do not belong to V_0 or V_1

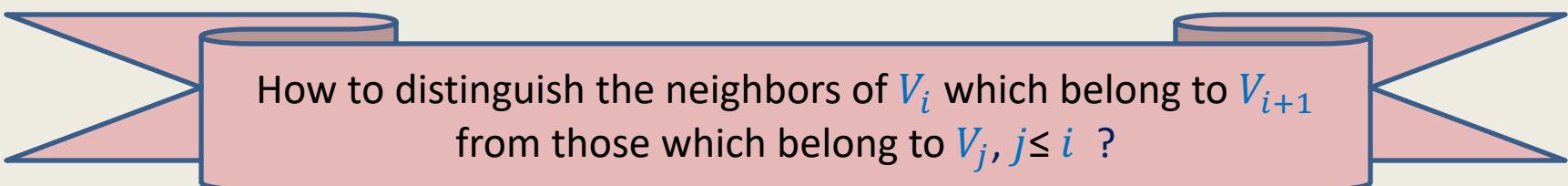
.

.

.

V_{i+1} : Vertices at distance **i+1** from $x =$

Those Neighbors of V_i which do not belong to V_{i-1} or V_i



How can we compute V_{i+1} ?

Key idea: compute V_i 's in increasing order of i .

Initialize $\text{Distance}[v] \leftarrow \infty$ of each vertex v in the graph.

Initialize $\text{Distance}[x] \leftarrow 0$.

- First compute V_0 .
- Then compute V_1 .
- ...
- Once we have computed V_i , for every neighbor v of a vertex in V_i ,

If v is in V_j for some $j \in \{i, i - 1\}$, then $\text{Distance}[v] =$

a number $\leq i$

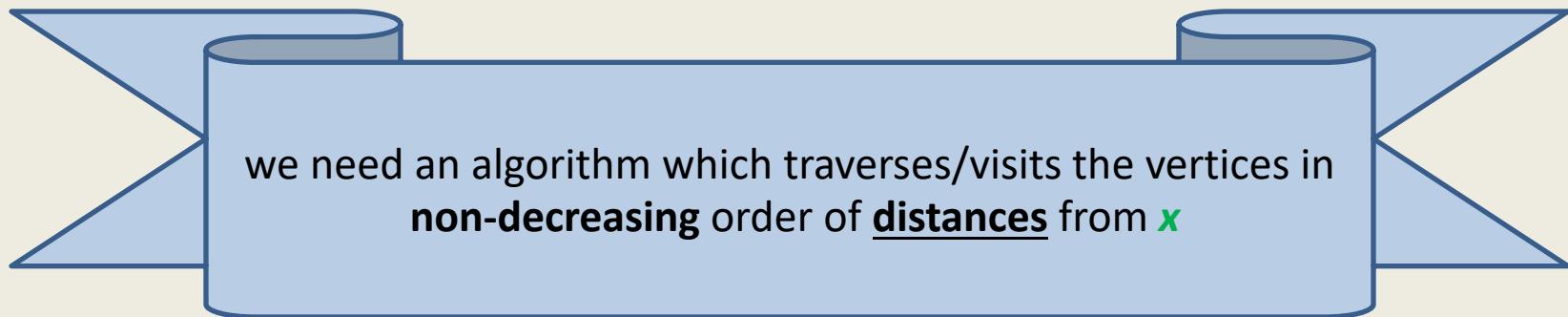
If v is in V_{i+1} , $\text{Distance}[v] =$

∞



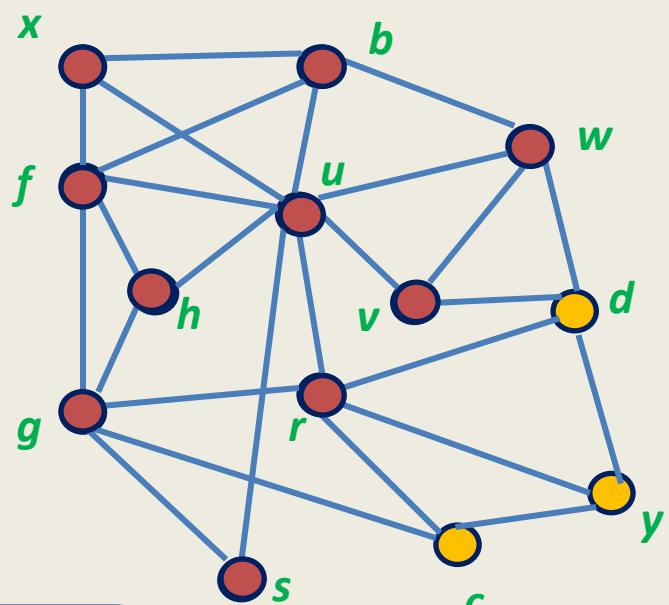
We can thus distinguish the neighbors of V_i which belong to V_{i+1} from those which belong to V_j .

A neat algorithm for computing distances from x

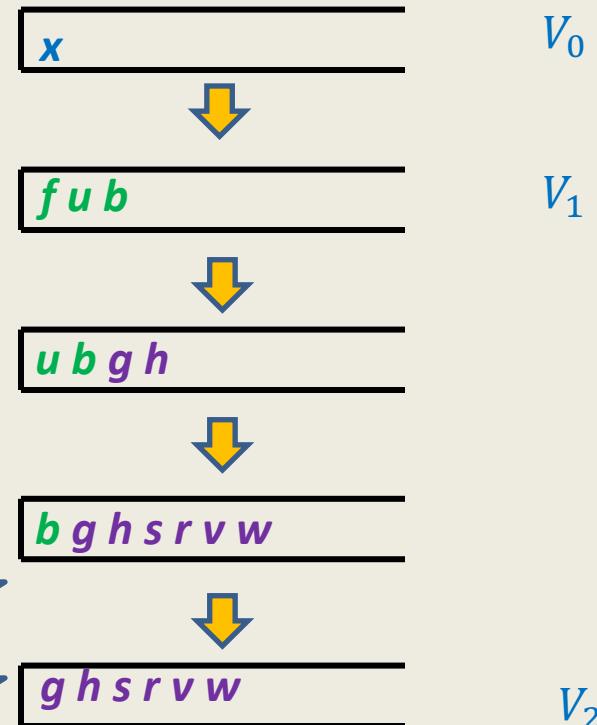


This traversal algorithm is called **BFS** (breadth first search) traversal

Using a queue for traversing vertices in non-decreasing order of distances



Compute distance of vertices from x :



Remove x and for each neighbor of x that was unvisited, mark it visited and put it into queue.

Remove f and for each neighbor of f that was unvisited, mark it visited and put it into queue.

Remove b and for each neighbor of b that was unvisited, mark it visited and put it into queue.

BFS traversal from a vertex

BFS(G, x)

CreateEmptyQueue(Q);

Distance(x) $\leftarrow 0$;

Enqueue(x, Q);

While(Not IsEmptyQueue(Q))

{ **v \leftarrow Dequeue(Q);**

For each neighbor w of v

{

if ($Distance(w) = \infty$)

{ **Distance(w) $\leftarrow Distance(v) + 1$;**

Enqueue(w, Q); ;

}

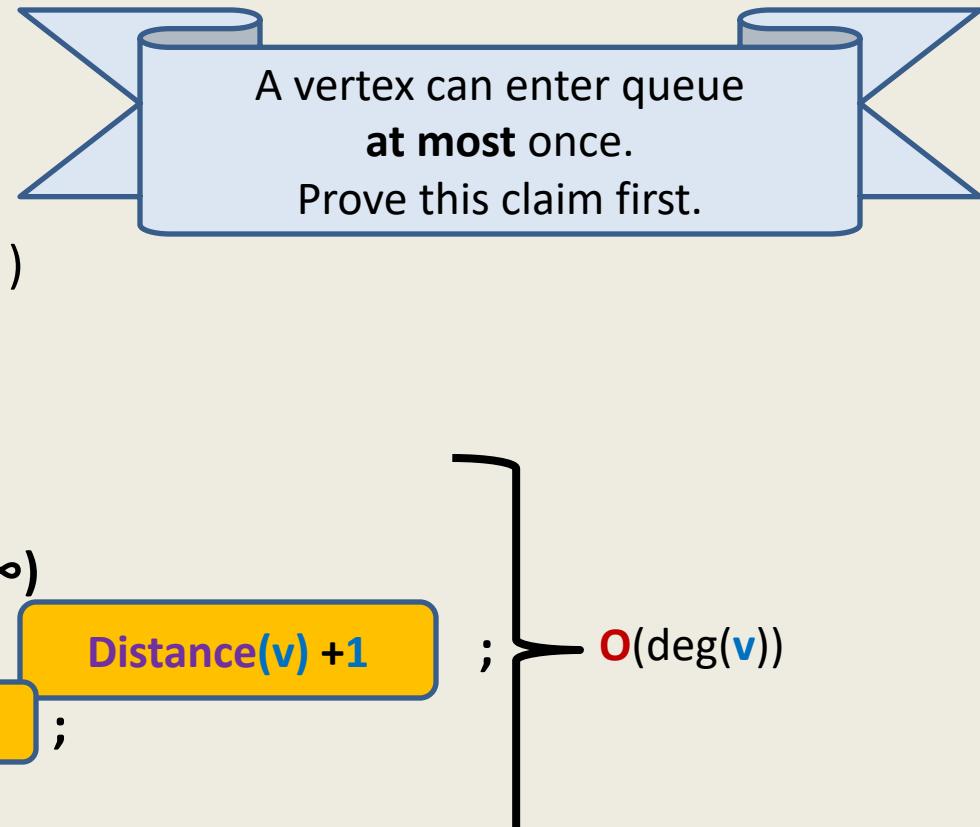
}

}

Running time of BFS traversal

$\text{BFS}(G, x)$

```
CreateEmptyQueue(Q);  
Distance(x)  $\leftarrow 0$ ;  
Enqueue(x, Q);  
While(      Not IsEmptyQueue(Q) )  
{      v  $\leftarrow$  Dequeue(Q);  
        For each neighbor w of v  
        {  
            if (Distance(w) =  $\infty$ )  
            {      Distance(w)  $\leftarrow$  Distance(v) + 1  
                Enqueue(w, Q);      ;  
            }  
        }  
}
```



Running time of $\text{BFS}(x)$ = no. of edges in the connected component of x.

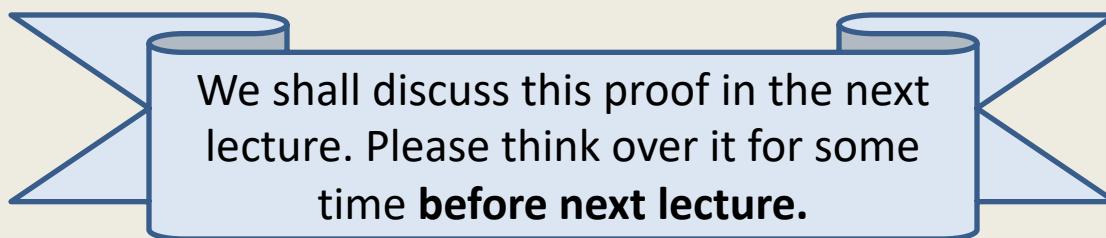
Correctness of BFS traversal

Question: What do we mean by correctness of BFS traversal from vertex x ?

Answer:

- All vertices reachable from x get visited.
- Vertices get visited in the non-decreasing order of their distances from x .
- At the end of the algorithm,

Distance(v) is the distance of vertex v from x .



Data Structures and Algorithms

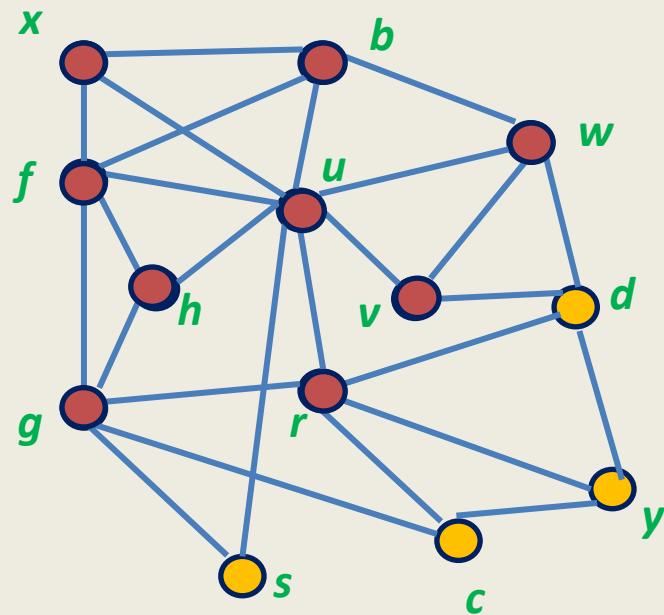
(ESO207)

Lecture 24

- BFS traversal (proof of correctness)
- BFS tree
- An important application of BFS traversal

Breadth First Search traversal

BFS Traversal in Undirected Graphs



BFS traversal of G from a vertex x

```
BFS( $G, x$ )      //Initially for each  $v$ , Distance( $v$ )  $\leftarrow \infty$  , and Visited( $v$ )  $\leftarrow \text{false}$ .  
{   CreateEmptyQueue(Q);  
    Distance( $x$ )  $\leftarrow 0$ ;  
    Enqueue( $x, Q$ );    Visited( $x$ )  $\leftarrow \text{true}$ ;  
    While(Not IsEmptyQueue(Q))  
    {       $v \leftarrow \text{Dequeue}(Q)$ ;  
        For each neighbor  $w$  of  $v$   
        {  
            if (Distance( $w$ ) =  $\infty$ )  
            {              Distance( $w$ )  $\leftarrow \text{Distance}(v) + 1$  ;  Visited( $w$ )  $\leftarrow \text{true}$ ;  
                Enqueue( $w, Q$ );  
            }  
        }  
    }  
}
```

Observations about $\text{BFS}(x)$

Observations:

- Any vertex v enters the queue at most once.
- Before entering the queue, $\text{Distance}(v)$ is updated.
- When a vertex v is dequeued, v processes all its unvisited neighbors as follows
 - its distance is **computed**,
 - It is **enqueued**.
- A vertex v in the queue is surely removed from the queue during the algorithm.

Correctness of BFS traversal

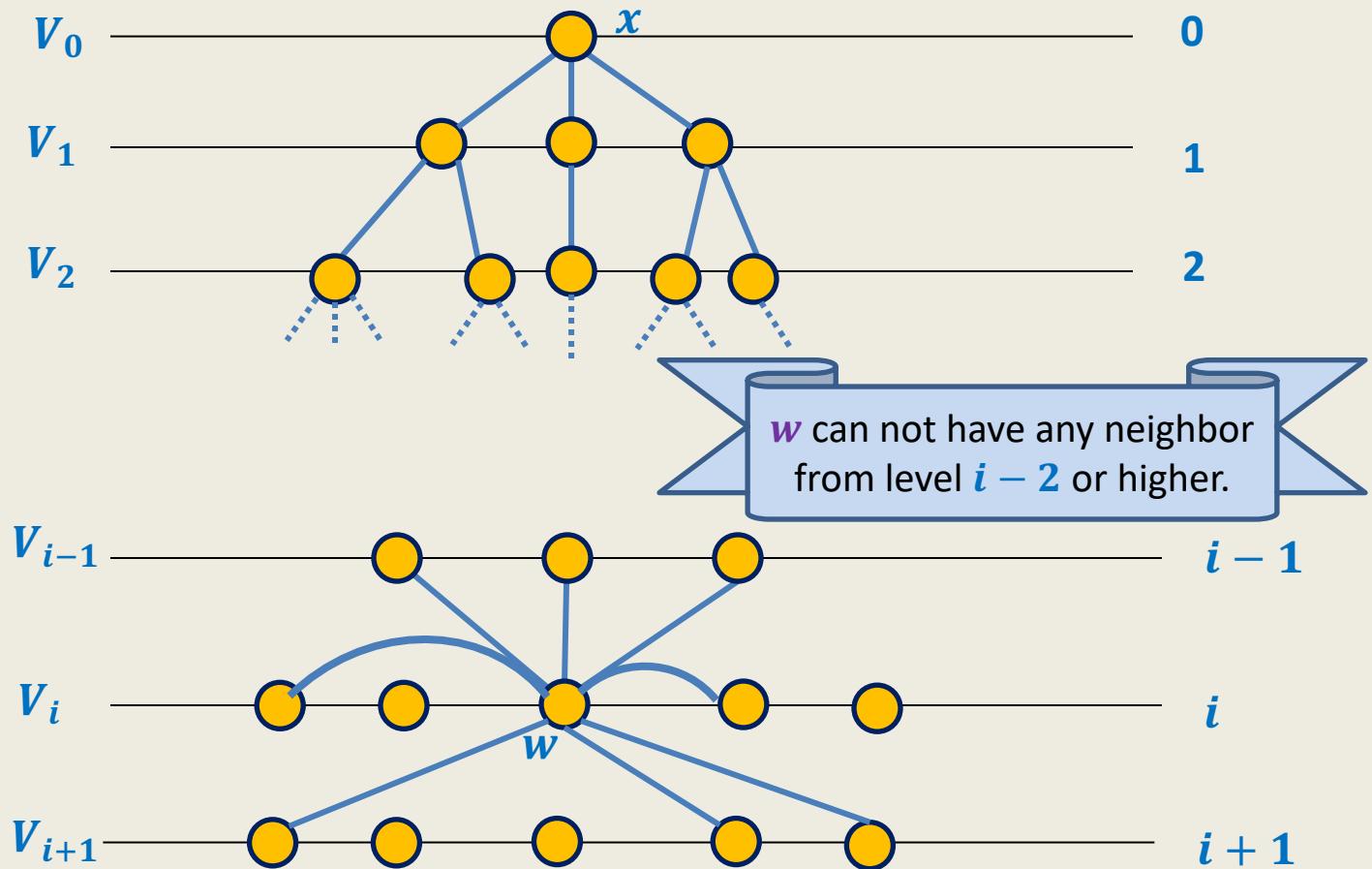
Question: What do we mean by correctness of $\text{BFS}(G, x)$?

Answer:

- All vertices reachable from x get visited.
- Vertices are visited in non-decreasing order of distance from x (Try as exercise).
- At the end of the algorithm, $\text{Distance}(v)$ is the distance of vertex v from x (Try as exercise).

The key idea

Partition the vertices according to their distance from x .



Correctness of $\text{BFS}(x)$ traversal

Part 1

All vertices reachable from x get visited

Proof of Part 1

Theorem: Each vertex v reachable from x gets visited during $\text{BFS}(G, x)$.

Proof:

(By **induction** on **distance from x**)

Inductive Assertion A(i) :

Every vertex v at distance i from x get visited.

Base case: $i = 0$.

x is the only vertex at distance 0 from x .

Right in the beginning of the algorithm $\text{Visited}(x) \leftarrow \text{true}$;

Hence the assertion $A(0)$ is true.

Induction Hypothesis: $A(j)$ is true for all $j < i$.

Induction step: To prove that $A(i)$ is true.

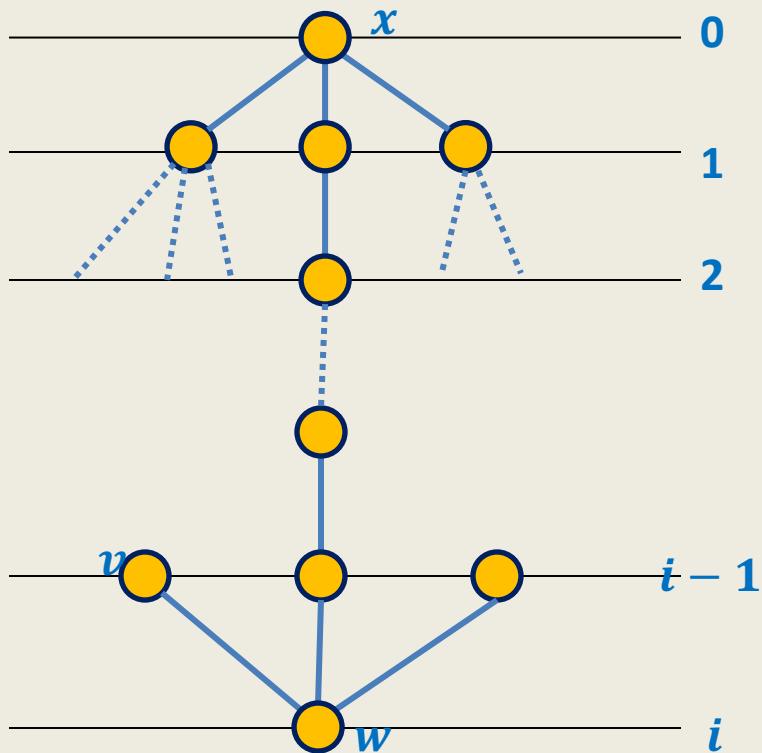
Let $w \in V_i$.

BFS traversal of G from a vertex x

```
BFS( $G, x$ )      //Initially for each  $v$ , Distance( $v$ )  $\leftarrow \infty$  , and Visited( $v$ )  $\leftarrow \text{false}$ .  
{   CreateEmptyQueue(Q);  
    Distance( $x$ )  $\leftarrow 0$ ;  
    Enqueue( $x, Q$ );    Visited( $x$ )  $\leftarrow \text{true}$ ;  
    While(Not IsEmptyQueue(Q))  
    {       $v \leftarrow \text{Dequeue}(Q)$ ;  
        For each neighbor  $w$  of  $v$   
        {  
            if (Distance( $w$ ) =  $\infty$ )  
            {              Distance( $w$ )  $\leftarrow \text{Distance}(v) + 1$  ;  Visited( $w$ )  $\leftarrow \text{true}$ ;  
                Enqueue( $w, Q$ );  
            }  
        }  
    }  
}
```

Induction step:

To prove that $w \in V_i$ is visited during $\text{BFS}(x)$



Let $v \in V_{i-1}$ be any neighbor of w .

By induction hypothesis,

v gets visited during $\text{BFS}(x)$.

So v gets Enqueued.

Hence v gets dequeued.

Focus on the moment when v is dequeued,



v scans all its neighbors and marks all its unvisited neighbors as visited.

Hence w gets visited too

This proves the induction step. If not already visited.

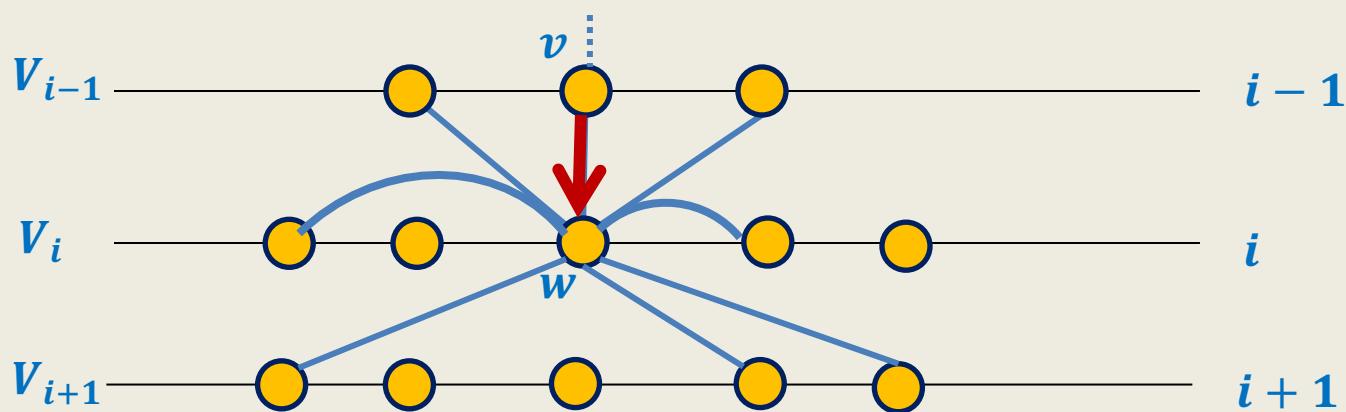
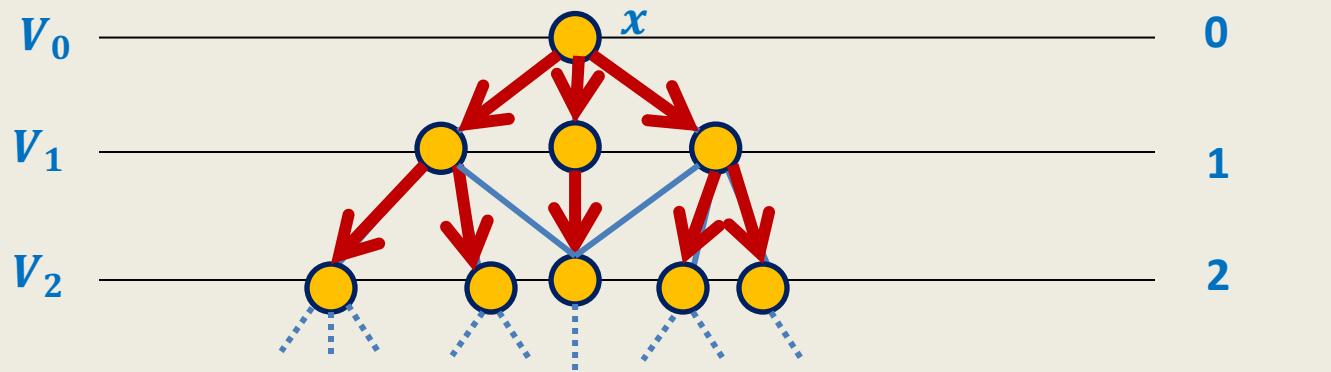
Hence by the principle of mathematical induction, $A(i)$ holds for each i .

This completes the proof of **part 1**.

BFS tree

BFS traversal gives a tree

Perform BFS traversal from x .

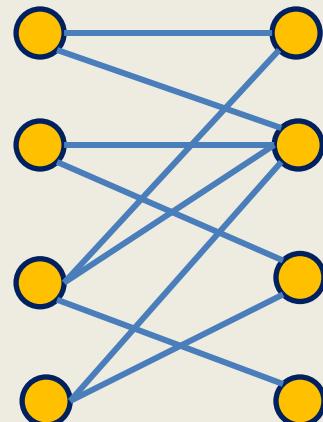


A nontrivial application of BFS traversal

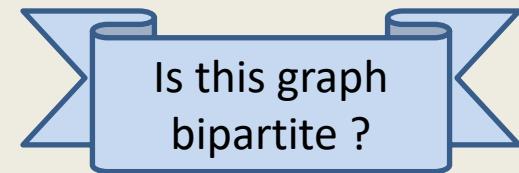
Determining
if a graph is bipartite

Bipartite graph

Definition: A graph $\mathbf{G}=(\mathbf{V},\mathbf{E})$ is said to be bipartite if its vertices can be partitioned into two sets \mathbf{A} and \mathbf{B} such that every edge in \mathbf{E} has one endpoint in \mathbf{A} and another in \mathbf{B} .

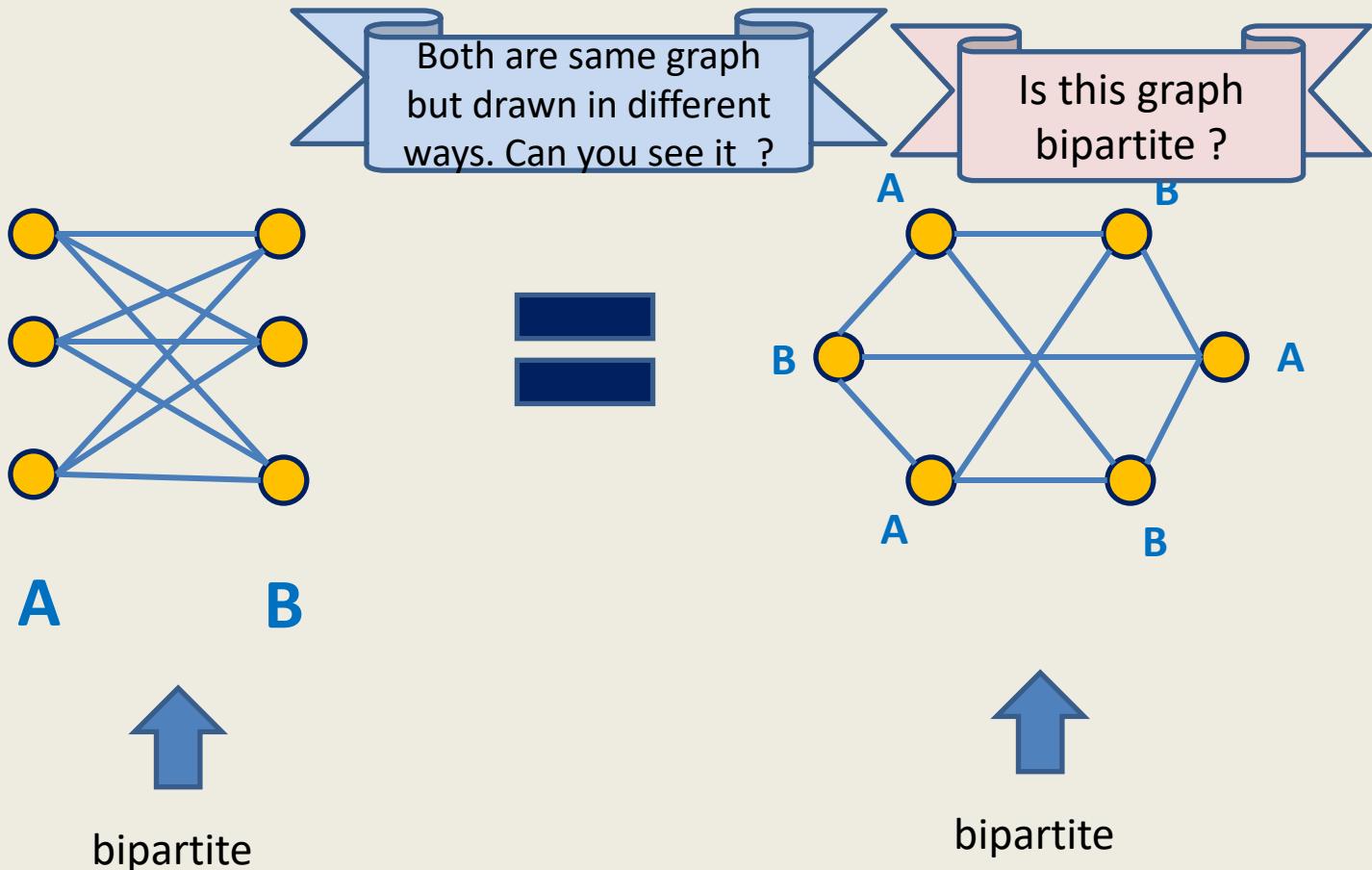


A B



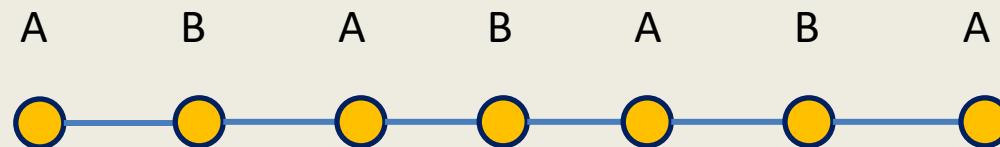
YES

Nontriviality in determining whether a graph is bipartite



Bipartite graph

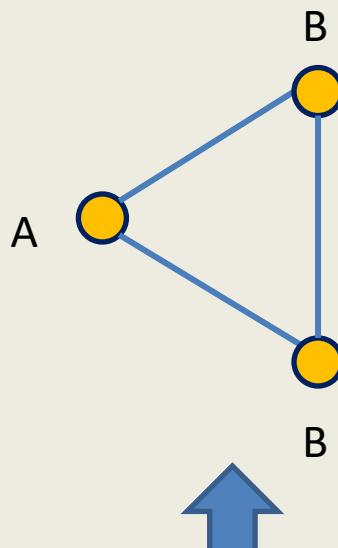
Question: Is a path bipartite ?



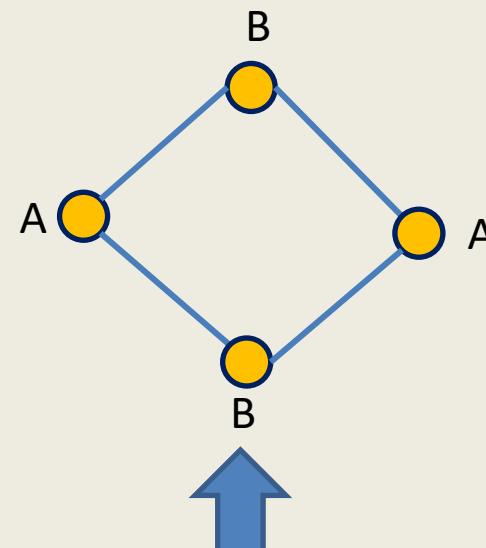
Answer: Yes

Bipartite graph

Question: Is a cycle bipartite ?



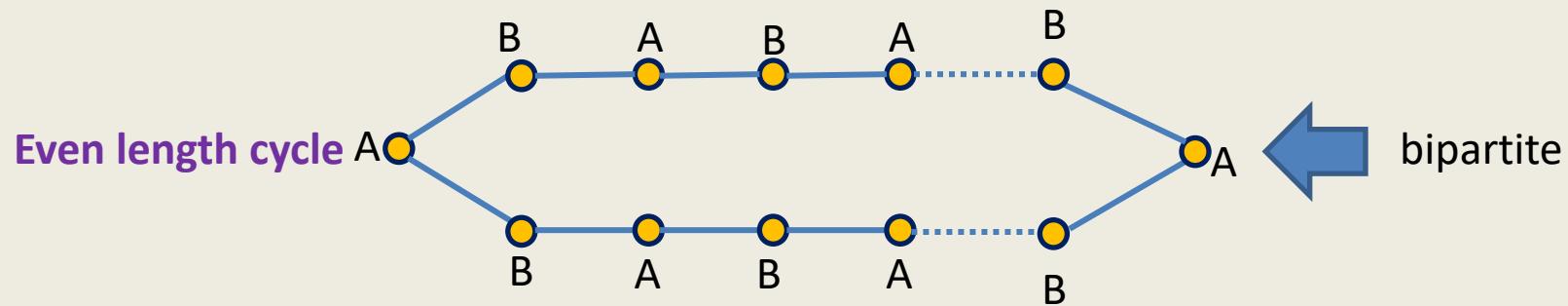
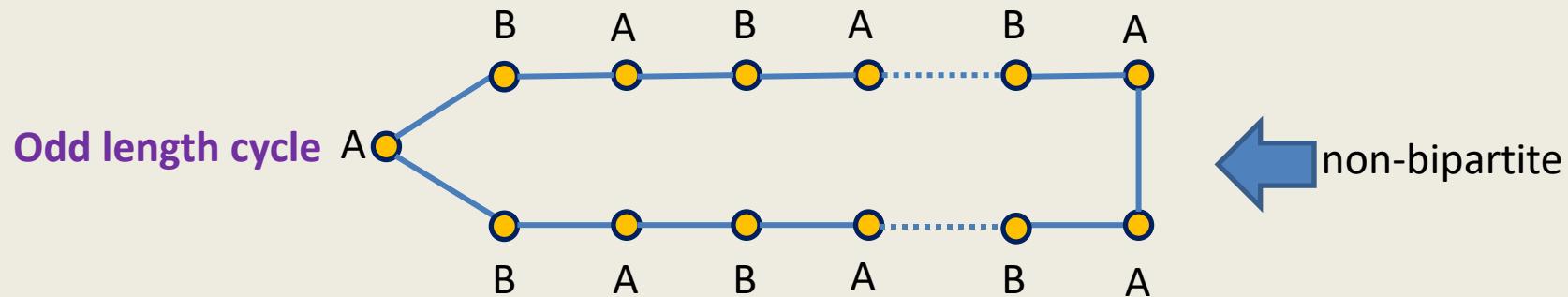
non-bipartite



bipartite

Bipartite graph

Question: Is a cycle bipartite ?



Subgraph

A subgraph of a graph $\mathbf{G}=(\mathbf{V},\mathbf{E})$

is a graph $\mathbf{G}'=(\mathbf{V}',\mathbf{E}')$ such that

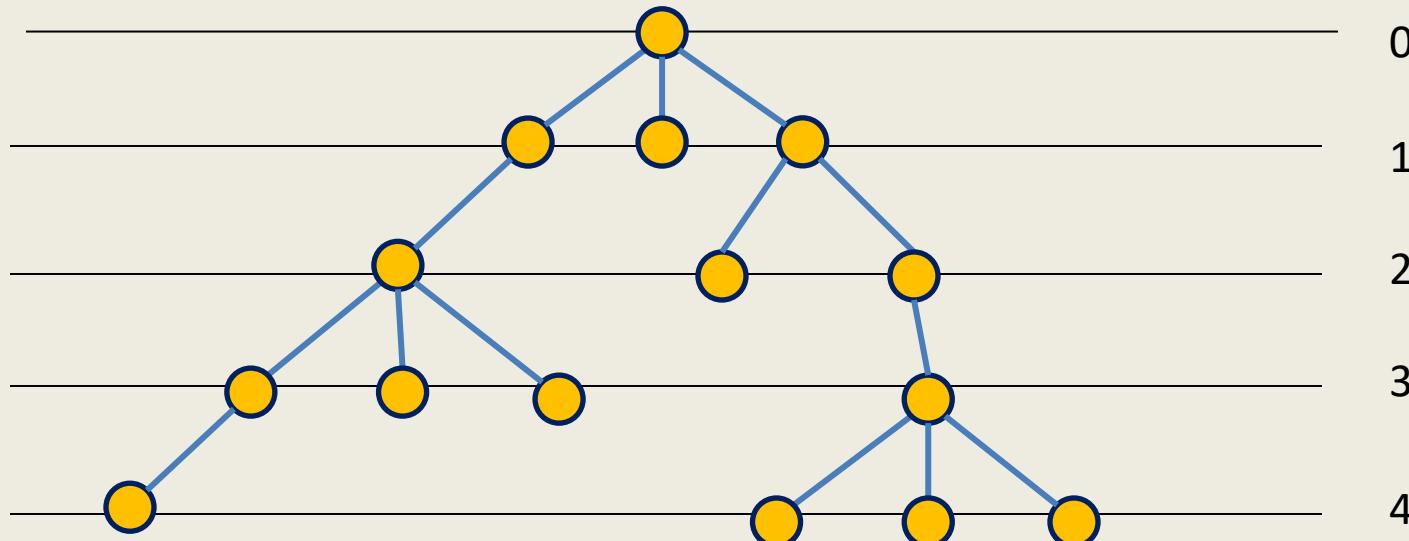
- $\mathbf{V}' \subseteq \mathbf{V}$
- $\mathbf{E}' \subseteq \mathbf{E} \cap (\mathbf{V}' \times \mathbf{V}')$

Question: If \mathbf{G} has a subgraph which is **an odd cycle**, is \mathbf{G} bipartite ?

Answer: **No.**

Bipartite graph

Question: Is a tree bipartite ?



Answer: Yes

Even level vertices: A

Odd level vertices: B

An algorithm for determining if a given graph is bipartite

Assumption:

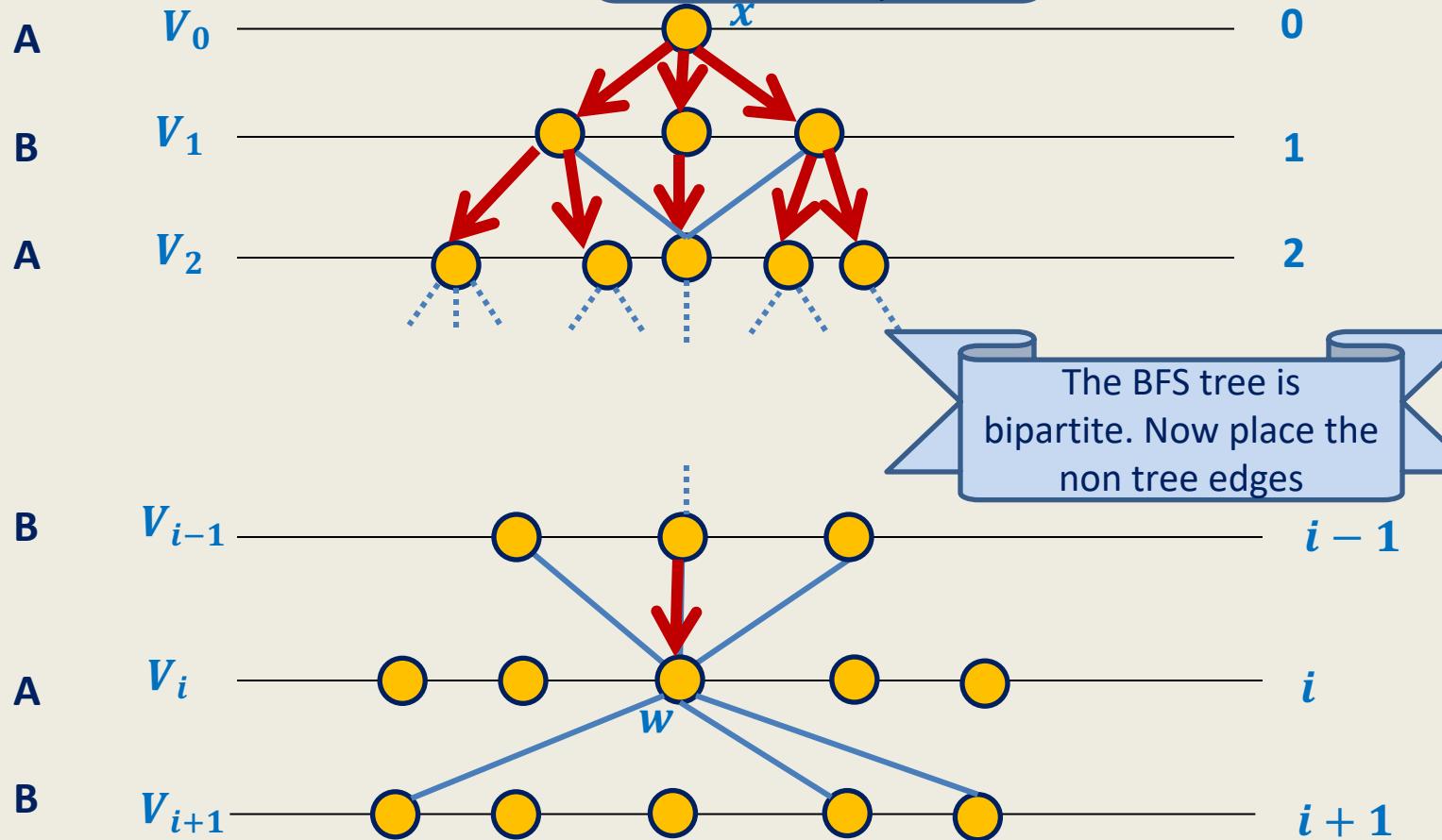
the graph is a single connected component

Compute a BFS tree at any vertex x .

If every nontree edge goes between two consecutive levels, what can we say ?



The graph is bipartite



Observation:

If every non-tree edge goes between two consecutive levels of **BFS** tree, then the graph is bipartite.

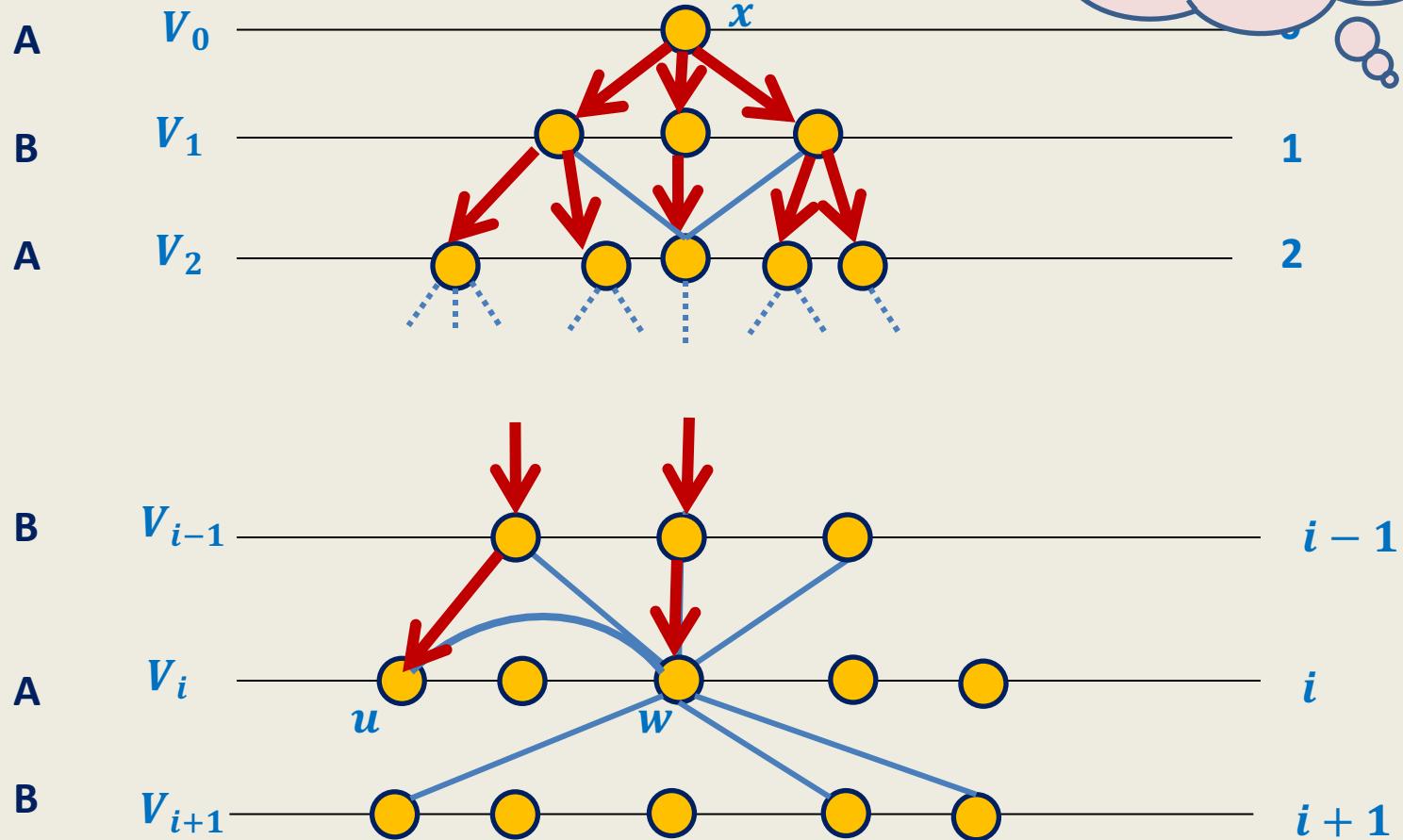
Question:

What if there is an edge with both end points at same level ?

What if there is an edge with both end points at same level ?

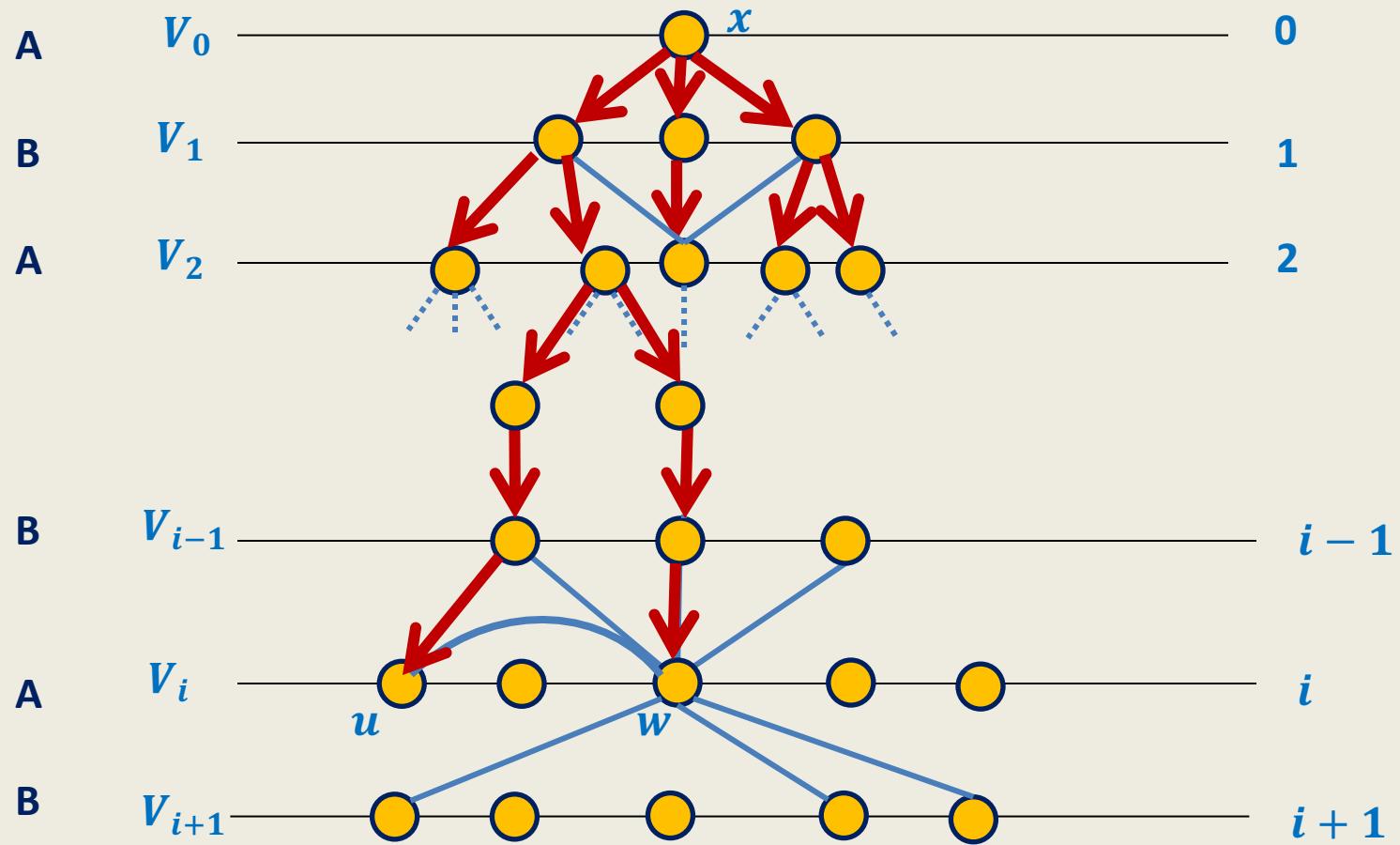
Keep following parent pointer from u and w simultaneously until we reach a common ancestor. What do we get ?

Can you spot an odd length cycle here ?





An odd cycle
containing u and w



Observation:

If there is **any** non-tree edge with **both endpoints at the same level** then the graph has **an odd length cycle**.

Hence the graph is **not** bipartite.

Theorem:

There is an $O(n + m)$ time algorithm to determine if a given graph is **bipartite**.

In the next 3 lectures, we are going to discuss **Depth First Traversal** : the most nontrivial, elegant graph traversal technique with wide applications.

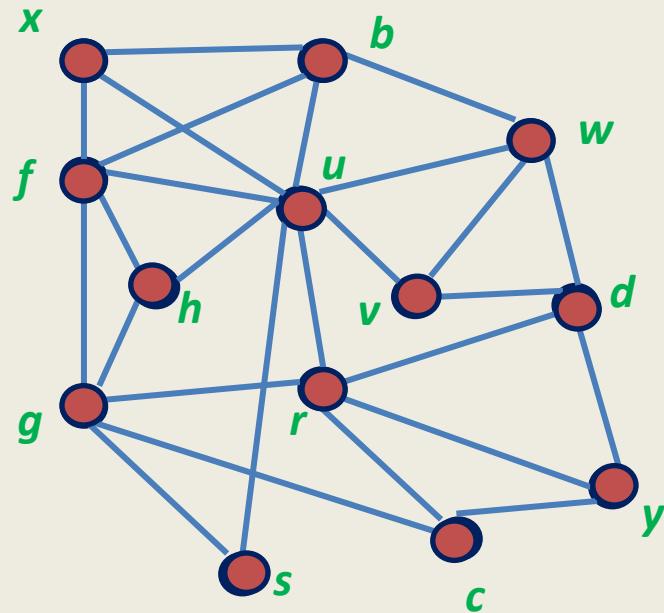
Data Structures and Algorithms

(ESO207)

Lecture 25

- A data structure problem for graphs.
- Depth First Search (DFS) Traversal
- Novel application: computing biconnected components of a graph

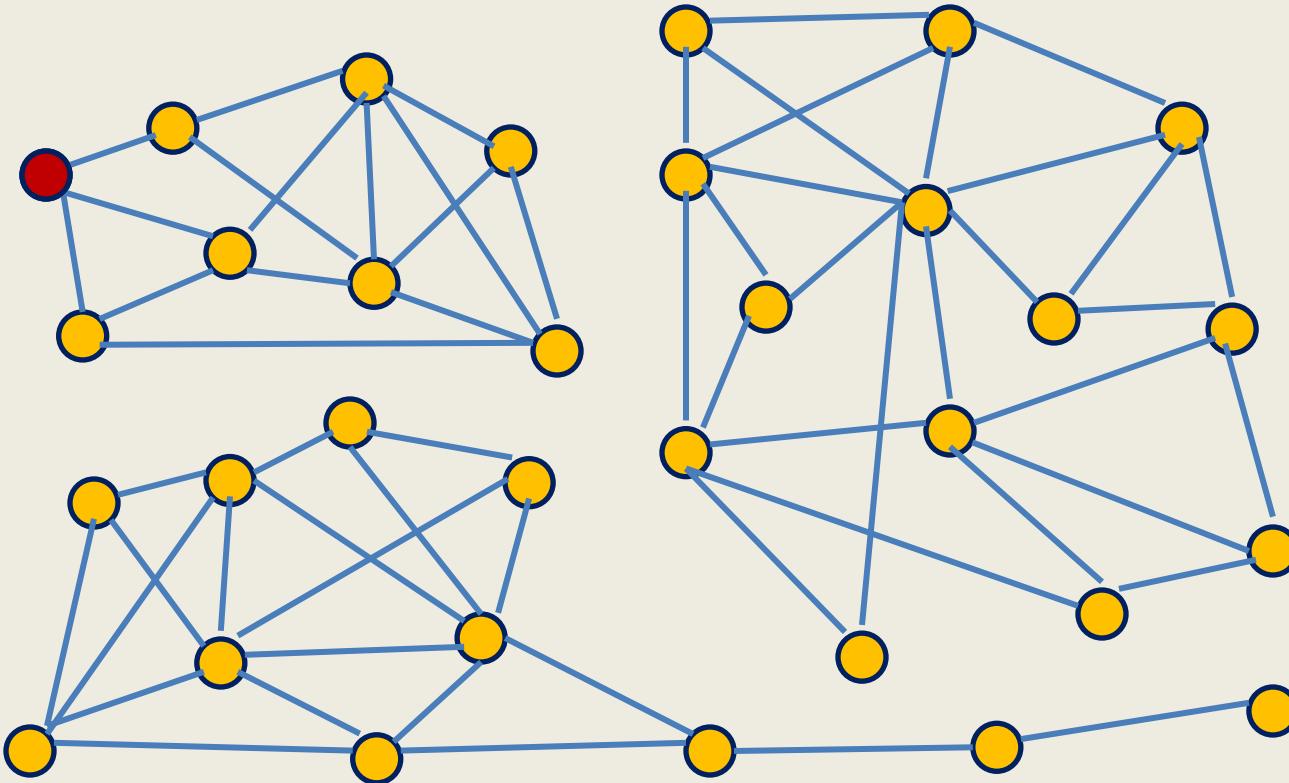
BFS Traversal in Undirected Graphs



Theorem:

BFS Traversal from **x** visits all vertices reachable from **x** in the given graph.

Connectivity problem in a Graph

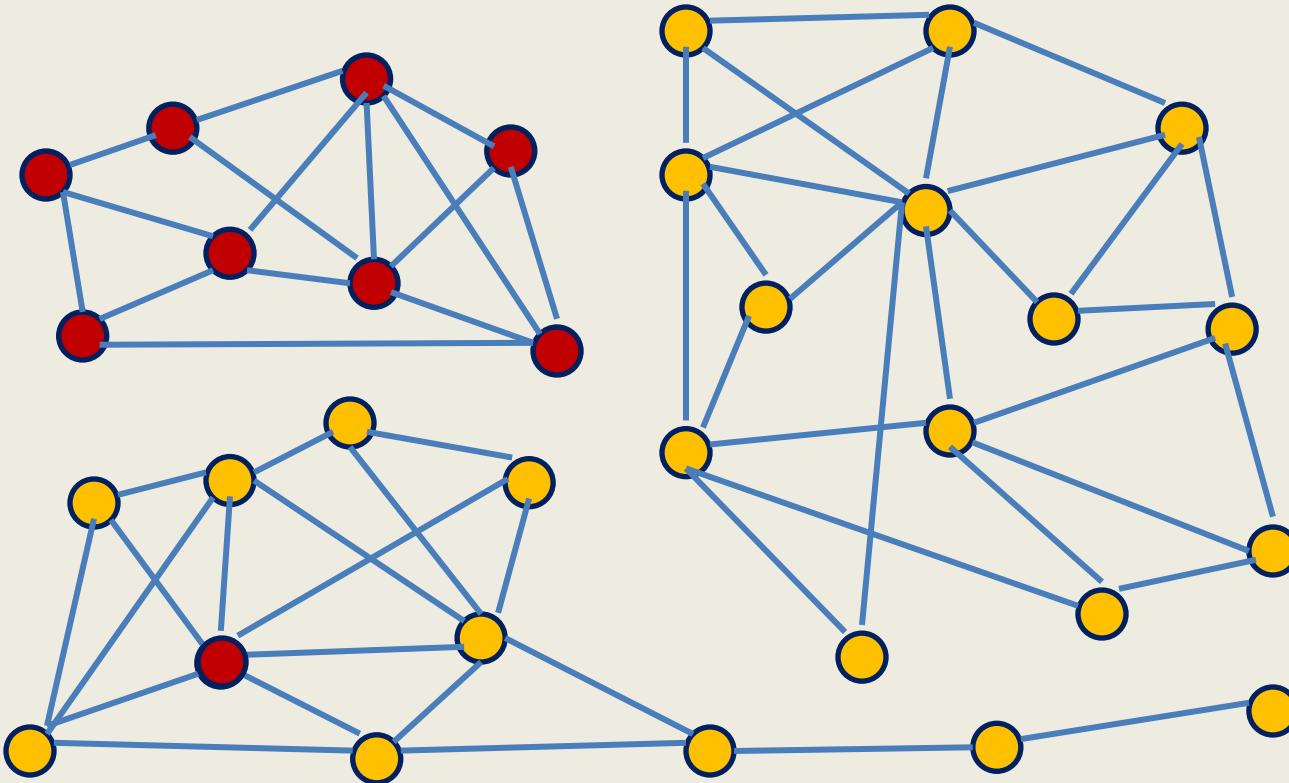


Problem:

Build an $O(n)$ size data structure for a given undirected graph s.t.
the following query can be answered in $O(1)$ time.

Is vertex i reachable from vertex j ?

Connectivity problem in a Graph

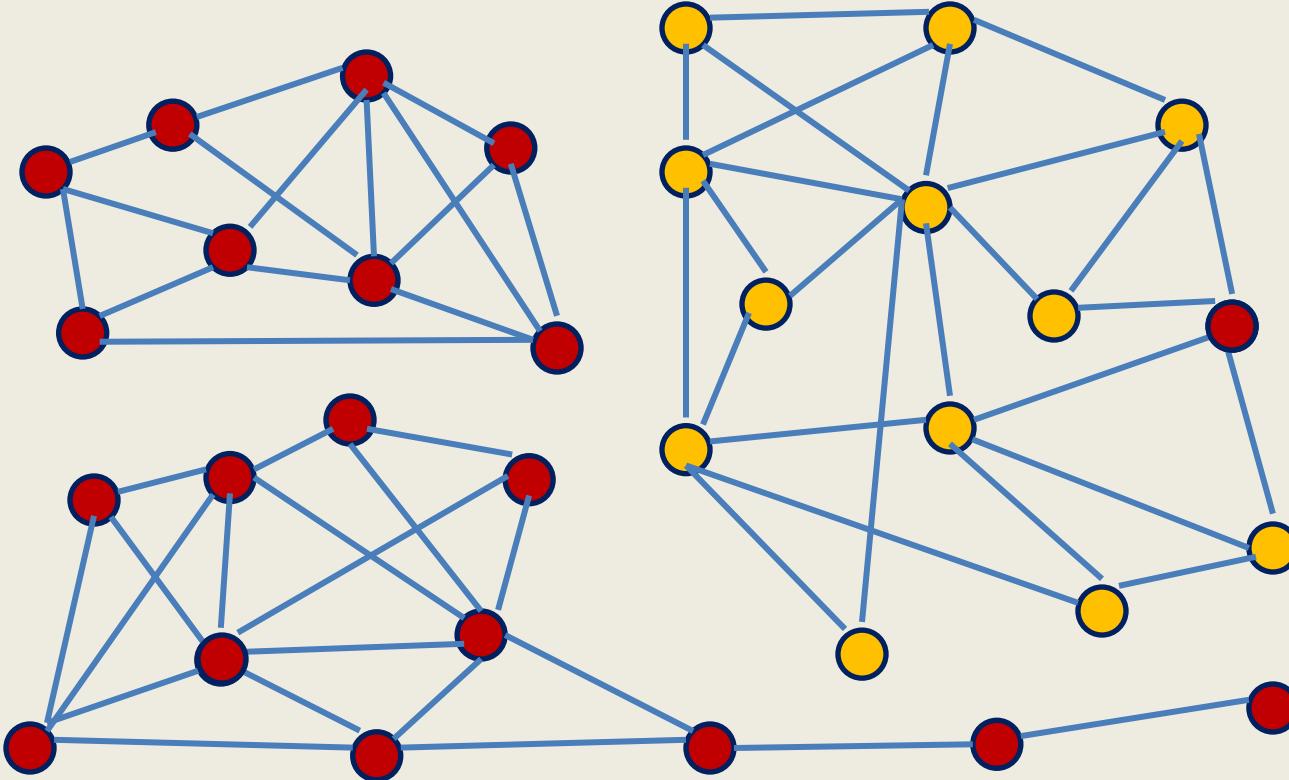


Problem:

Build an $O(n)$ size data structure for a given undirected graph s.t.
the following query can be answered in $O(1)$ time.

Is vertex i reachable from vertex j ?

Connectivity problem in a Graph

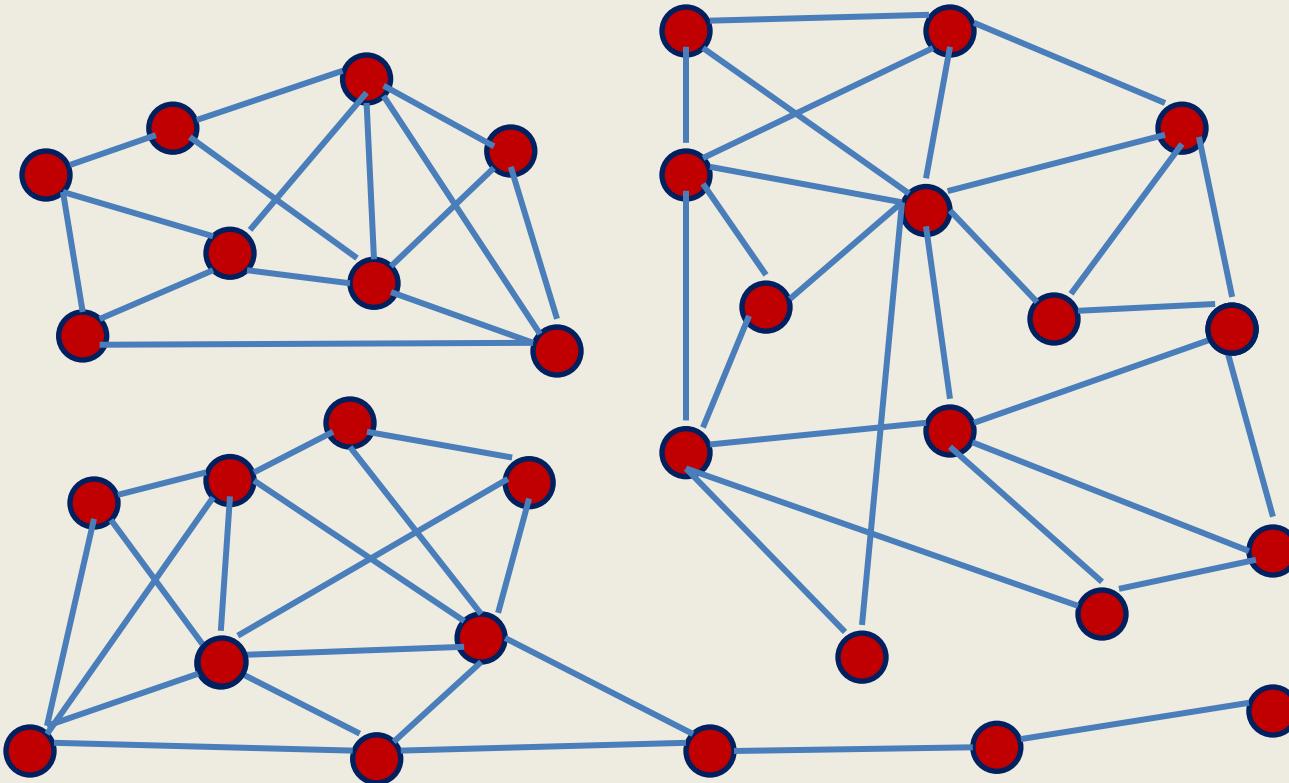


Problem:

Build an $O(n)$ size data structure for a given undirected graph s.t.
the following query can be answered in $O(1)$ time.

Is vertex i reachable from vertex j ?

Connectivity problem in a Graph



Problem:

Build an $O(n)$ size data structure for a given undirected graph s.t.
the following query can be answered in $O(1)$ time.

Is vertex i reachable from vertex j ?

Connectivity problem in a Graph

BFS(x)

```
CreateEmptyQueue(Q);
Visited( $x$ )  $\leftarrow$  true; Label[ $x$ ]  $\leftarrow$   $x$ ;
Enqueue( $x$ , Q);
While(Not IsEmptyQueue(Q))
{
     $v \leftarrow$  Dequeue(Q);
    For each neighbor  $w$  of  $v$ 
    {
        if (Visited( $w$ ) = false)
        {
            Visited( $w$ )  $\leftarrow$  true ; Label[ $w$ ]  $\leftarrow$   $x$  ;
            Enqueue( $w$ , Q);
        }
    }
}
```

Connectivity(G)

```
{ For each vertex  $x$  Visited( $x$ )  $\leftarrow$  false; Create an array Label;
    For each vertex  $v$  in  $V$ 
        If (Visited( $v$ ) = false) BFS( $x$ );
    return Label;
}
```

Analysis of the algorithm

Output of the algorithm:

Array **Label[]** of size **O(*n*)** such that

Label[x]=Label[y] if and only if **x** and **y** belong to same connected component.

Running time of the algorithm :

$$O(n + m)$$

Theorem:

An undirected graph can be processed in **O(*n* + *m*)** time
to build an **O(*n*)** size data structure
which can answer any connectivity query in **O(1)** time.

Is there alternate way to traverse a graph ?

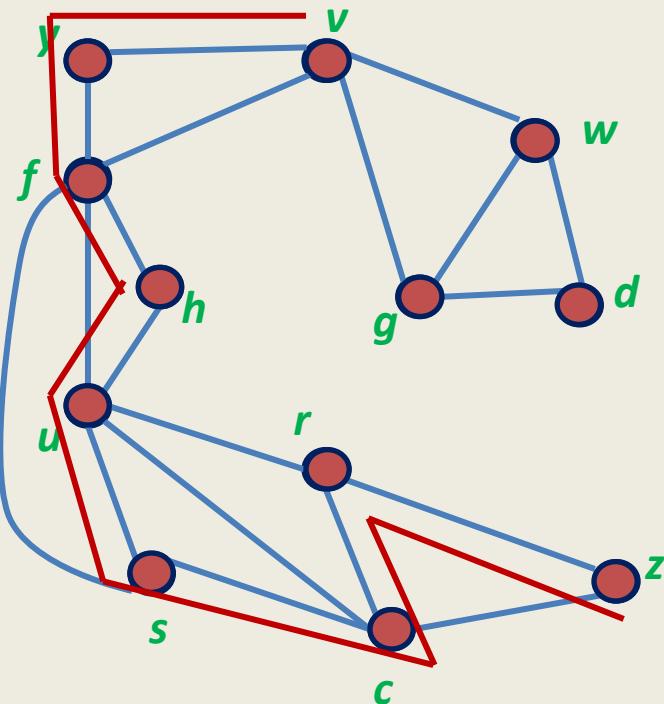


Try to get inspiration from your “human executable method”
to design
“**a machine executable algorithm**” for traversing a graph.



How will you do it without any map or
asking any one for directions ?

A recursive way to traverse a graph



We need a **mechanism** to

- **Avoid** visiting a vertex multiple times
We can solve it by keeping a label “Visited” for each vertex like in BFS traversal.
- **Trace back** in case we reach a dead end.
Recursion takes care of it ☺

DFS traversal of G

$\text{DFS}(v)$

```
{ Visited(v)  $\leftarrow$  true;  
  For each neighbor w of v  
  {    if (Visited(w) = false)  
        {      DFS(w);  
            .....;  
        }  
        .....;  
    }  
}
```

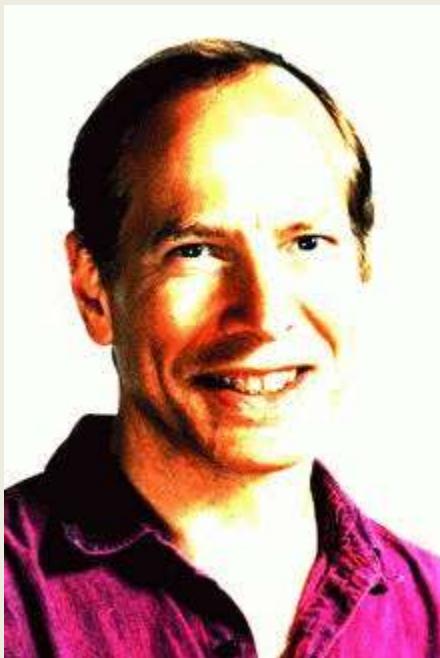
Add a few extra statements here
to get an efficient algorithm
for a new problem ☺

$\text{DFS-traversal}(G)$

```
{ For each vertex  $v \in V$  { Visited(v)  $\leftarrow$  false; }  
  For each vertex  $v \in V$  {  
    If (Visited(v) = false) DFS(v);  
  }  
}
```

DFS traversal

a **milestone** in the area of graph algorithms



Invented by **Robert Endre Tarjan** in 1972

- One of the **pioneers** in the field of data structures and algorithms.
- Got the **Turing award** (equivalent to **Nobel prize**) for his fundamental contribution to data structures and algorithms.
- **DFS traversal** has proved to be a very powerful tool for graph algorithms.

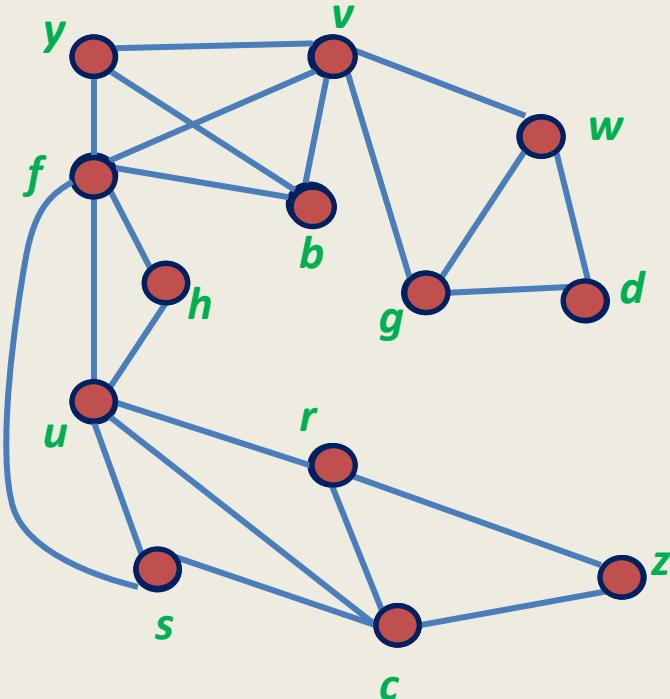
DFS traversal

a **milestone** in the area of graph algorithms

Applications:

- **Connected** components of a graph.
- **Biconnected** components of a graph.
(Is the connectivity of a graph robust to failure of any node ?)
- Finding **bridges** in a graph.
(Is the connectivity of a graph robust to failure of any edge)
- **Planarity testing** of a graph
(Can a given graph be embedded on a plane so that no two edges intersect ?)
- **Strongly connected** components of a directed graph.
(the extension of connectivity in case of directed graphs)

Insight into DFS through an example



$\text{DFS}(v)$ begins

v visits y

$\text{DFS}(y)$ begins

y visits f

$\text{DFS}(f)$ begins

f visits b

$\text{DFS}(b)$ begins

all neighbors of b are already visited

$\text{DFS}(b)$ ends

control returns to $\text{DFS}(f)$

f visits h

$\text{DFS}(h)$ begins

.... and so on

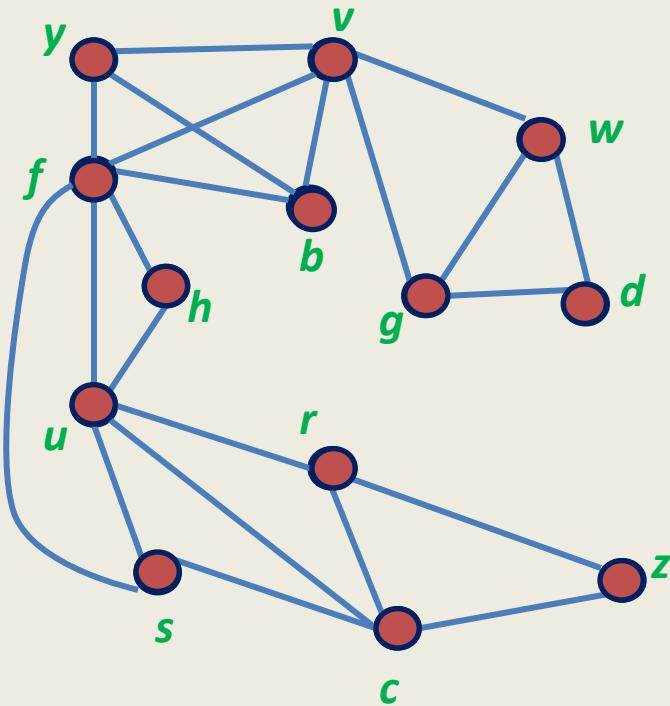
After visiting z , control returns to $r \rightarrow c \rightarrow s \rightarrow u \rightarrow h \rightarrow f \rightarrow y \rightarrow v$

v visits w

$\text{DFS}(w)$ begins

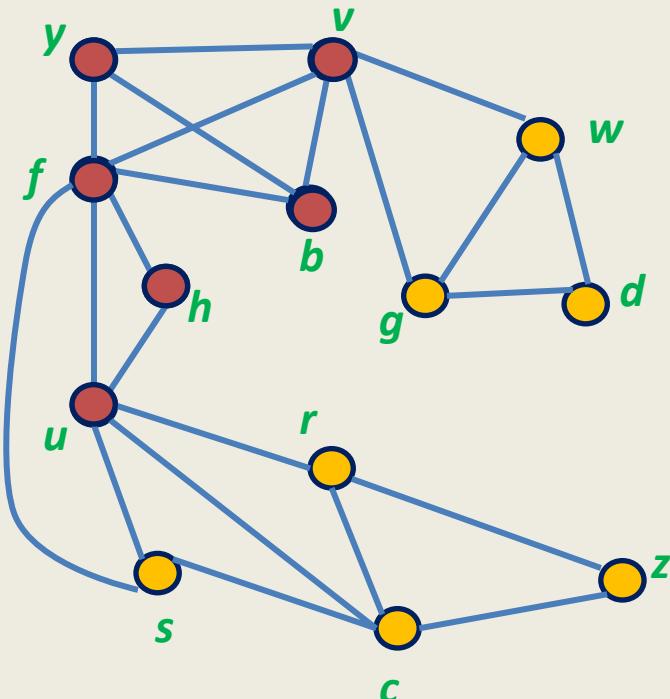
.... and so on

Insight into DFS through an example



Observation1: (Recursive nature of DFS)
If $\text{DFS}(v)$ invokes $\text{DFS}(w)$, then
 $\text{DFS}(w)$ finishes **before** $\text{DFS}(v)$.

Insight into DFS through an example

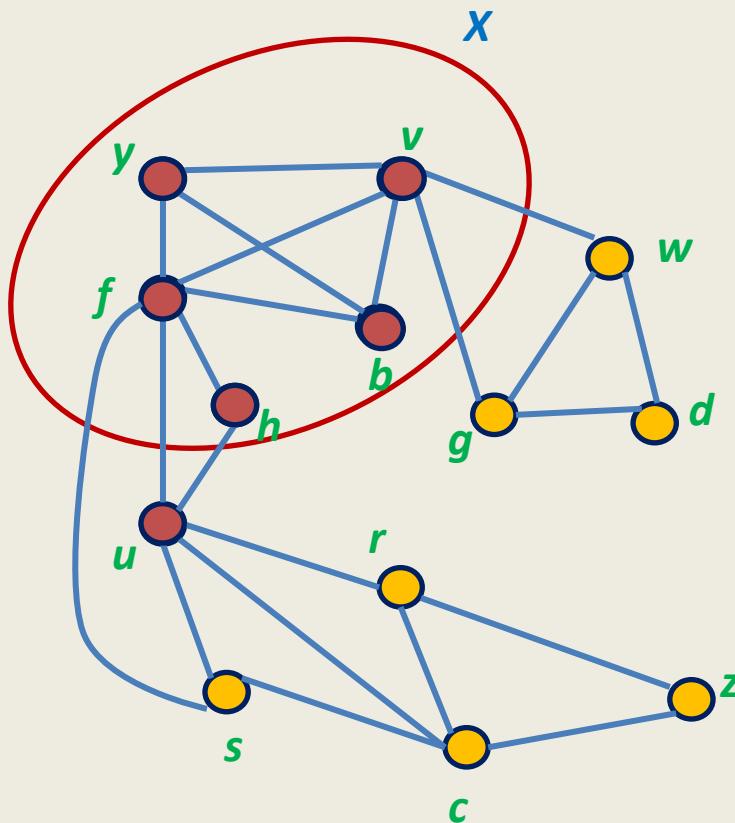


Question :

When **DFS** reaches a vertex **u**, what is the role of vertices already visited ?

The traversal will not proceed along the vertices which are **already visited**. Hence the visited vertices act as a **barrier** for the traversal from **u**.

Insight into DFS through an example



Observation 2:

Let X be the set of vertices visited before **DFS** traversal reaches vertex u for the first time.

The **DFS(u)** pursued now is like

fresh **DFS(u)** executed in graph $G \setminus X$.

NOTE:

$G \setminus X$ is the graph G after removal of all vertices X along with their edges.

Proving that $\text{DFS}(v)$ visits all vertices reachable from v

By **induction** on the

size of connected component of v

Can you figure out the **inductive assertion** now?

Think over it. It is given on the following slide...

Inductive assertion

A(*i*):

If a connected component has **size = *i***, then **DFS** from any of its vertices **will visit** all its vertices.

PROOF:

Base case: *i* = 1.

The component is $\{v\}$ and the first statement of **DFS(*v*)** marks it visited.

So **A(1)** holds.

Induction hypothesis:

If a connected component has **size < *i***, then **DFS** from any of its vertices **will visit** all its vertices.

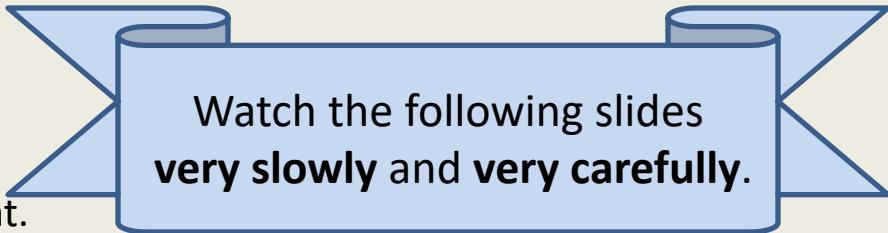
Induction step:

We have to prove that **A(*i*)** holds.

Consider any connected component of size *i*.

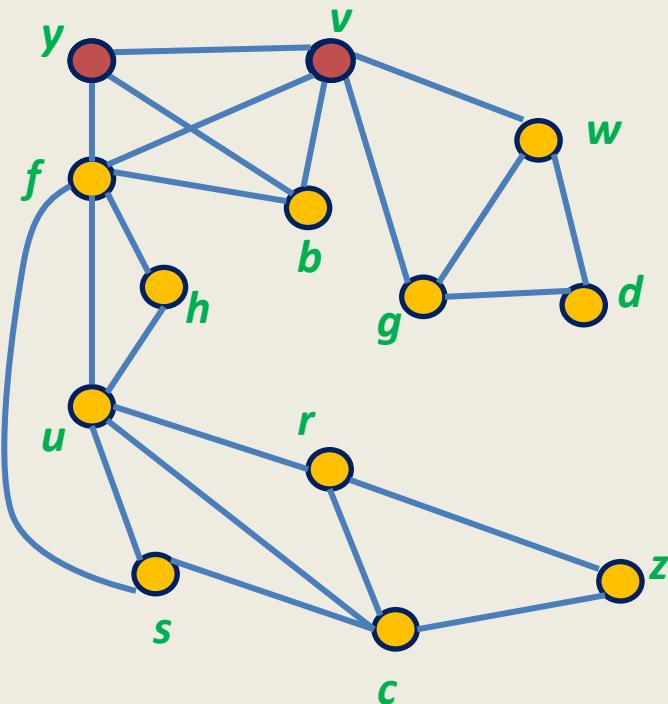
Let V^* be the set of its vertices. $|V^*| = i$.

Let *v* be any vertex in the connected component.



Watch the following slides
very slowly and very carefully.

DFS(v)



Let y be the first neighbor visited by v .

$B =$ the set of vertices such that **every path** from y to them passes through v .

$$C = V^* \setminus B.$$

$$B = \{v, g, w, d\}$$

$|B| < i$ since $y \notin B$

$$C = \{y, b, f, h, u, s, c, r, z\}$$

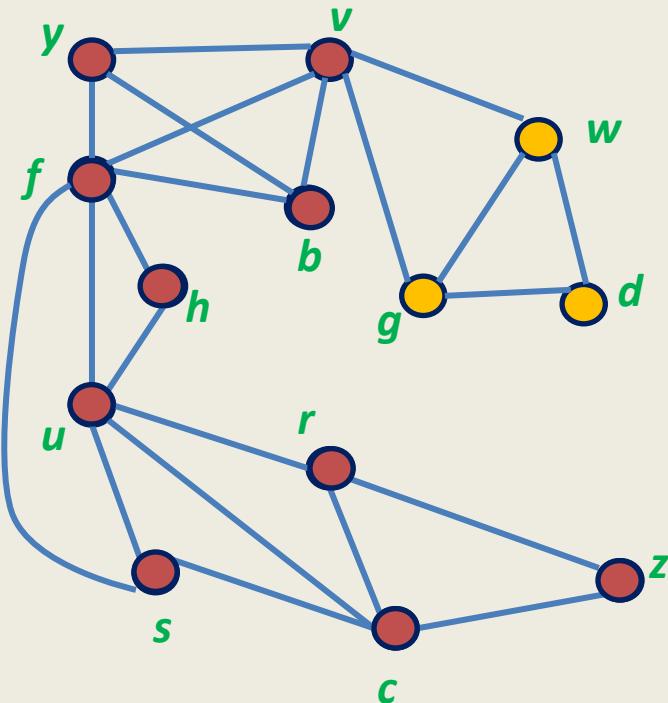
$|C| < i$ since $v \notin C$

Question: What is $\text{DFS}(y)$ like ?

Answer: $\text{DFS}(y)$ in $G \setminus \{v\}$.

Question: What is the connected component of y in $G \setminus \{v\}$?

DFS(v)



Let y be the first neighbor visited by v .

$B =$ the set of vertices such that **every path** from y to them passes through v .

$$C = V^* \setminus B.$$

$$B = \{v, g, w, d\}$$

$|B| < i$ since $y \notin B$

$$C = \{y, b, f, h, u, s, c, r, z\}$$

$|C| < i$ since $v \notin C$

Question: What is $\text{DFS}(y)$ like ?

Answer: $\text{DFS}(y)$ in $G \setminus \{v\}$.

Question: What is the connected component of y in $G \setminus \{v\}$?

Answer: C .

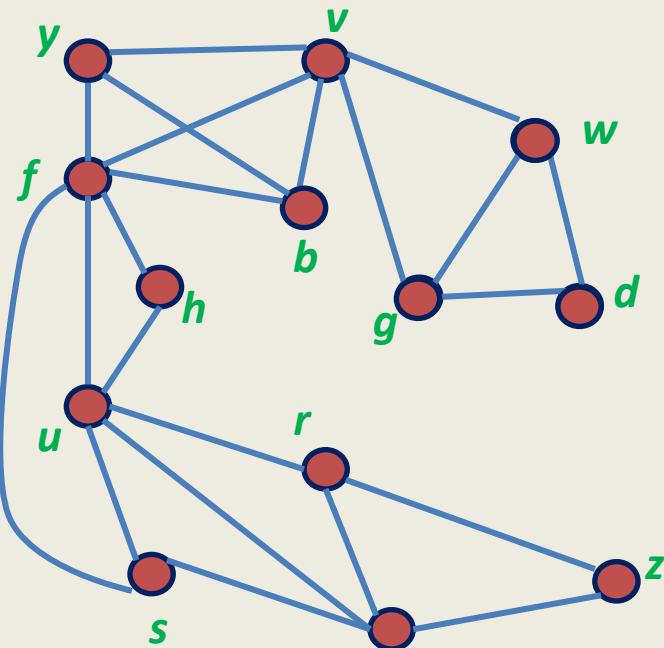
$|C| < i$, so by I.H., $\text{DFS}(y)$ visits entire set C & we return to v .

Question: What is $\text{DFS}(v)$ like when $\text{DFS}(y)$ finishes ?

Answer: $\text{DFS}(v)$ in $G \setminus C$.

Question: What is the connected component of v in $G \setminus C$?

DFS(v)



Hence entire component
of v gets visited

Let y be the first neighbor visited by v .

$B =$ the set of vertices such that **every path** from y to them passes through v .

$$C = V^* \setminus B.$$

$$B = \{v, g, w, d\}$$

$|B| < i$ since $y \notin B$

$$C = \{y, b, f, h, u, s, c, r, z\}$$

$|C| < i$ since $v \notin C$

Question: What is $\text{DFS}(y)$ like ?

Answer: $\text{DFS}(y)$ in $G \setminus \{v\}$.

Question: What is the connected component of y in $G \setminus \{v\}$?

Answer: C .

$|C| < i$, so by I.H., $\text{DFS}(y)$ visits entire set C & we return to v .

Question: What is $\text{DFS}(v)$ like when $\text{DFS}(y)$ finishes ?

Answer: $\text{DFS}(v)$ in $G \setminus C$.

Question: What is the connected component of v in $G \setminus C$?

Answer: B .

$|B| < i$, so by I.H., $\text{DFS}(v)$ pursued after finishing $\text{DFS}(y)$ visits entire set B .

Theorem: $\text{DFS}(v)$ visits all vertices of the connected component of v .

Homework:

Use **DFS** traversal to compute all connected components of a given **G** in time **$O(m + n)$** .

Data Structures and Algorithms

(ESO207)

Lecture 26

- Depth First Search (**DFS**) Traversal
- **DFS Tree**
- **Novel application:** computing **biconnected components of a graph**

DFS traversal of G

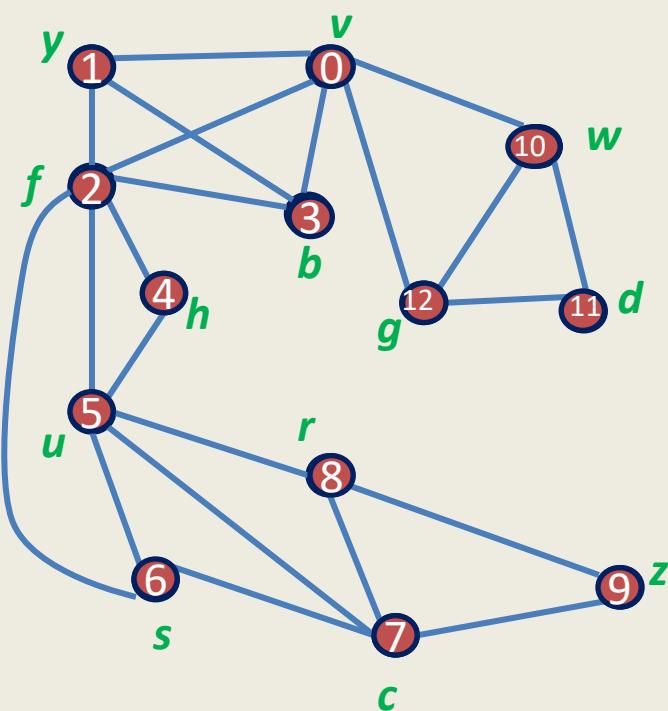
DFS(v)

```
{ Visited( $v$ )  $\leftarrow$  true; DFN[ $v$ ]  $\leftarrow$  dfn ++;  
    For each neighbor  $w$  of  $v$   
    {      if (Visited( $w$ ) = false)  
        {    DFS( $w$ ) ;  
            .....;  
        }  
        .....;  
    }  
}
```

DFS-traversal(G)

```
{ dfn  $\leftarrow$  0;  
    For each vertex  $v \in V$  { Visited( $v$ )  $\leftarrow$  false }  
    For each vertex  $v \in V$  { If (Visited( $v$ ) = false) DFS( $v$ ) }  
}
```

DFN number

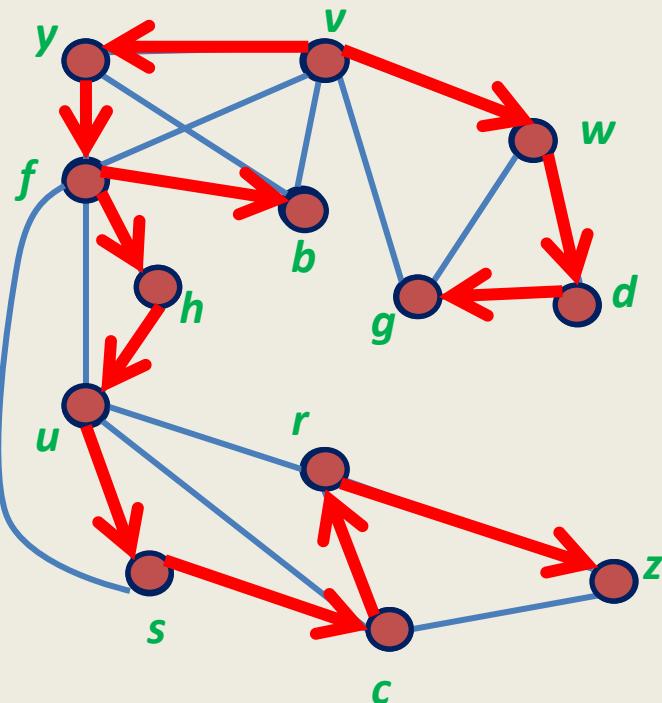


DFN[x] :

The number at which x gets visited during DFS traversal.

DFS tree

DFS(v) computes a tree rooted at v

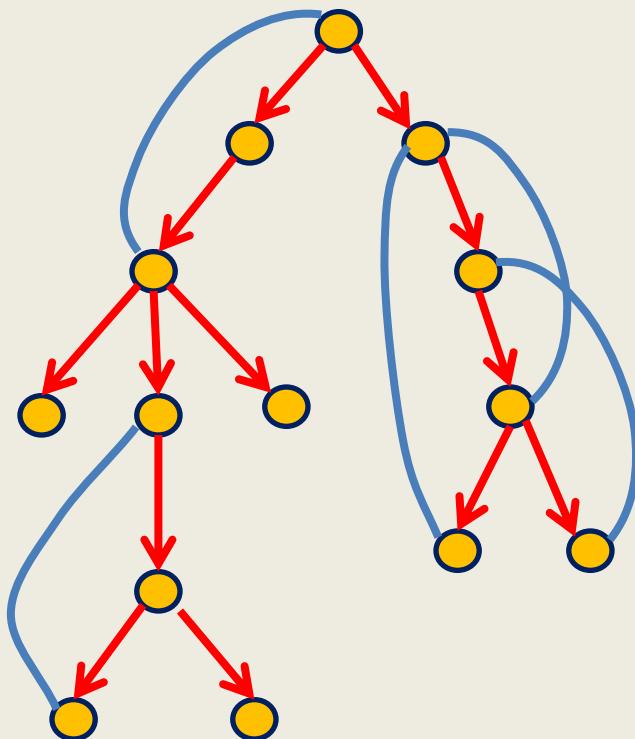


A DFS tree rooted at v

Can any rooted tree be obtained through DFS ?

No

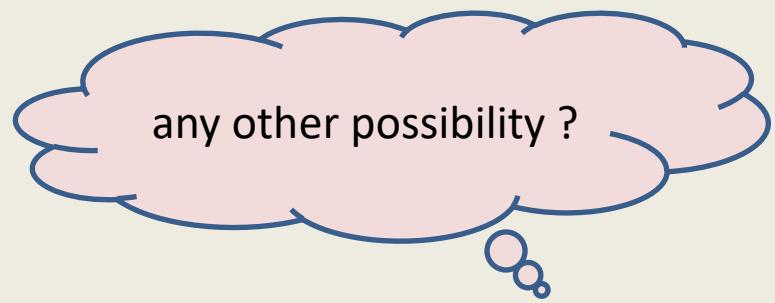
How will an edge appear in DFS traversal ?



- as a **tree-edge**.

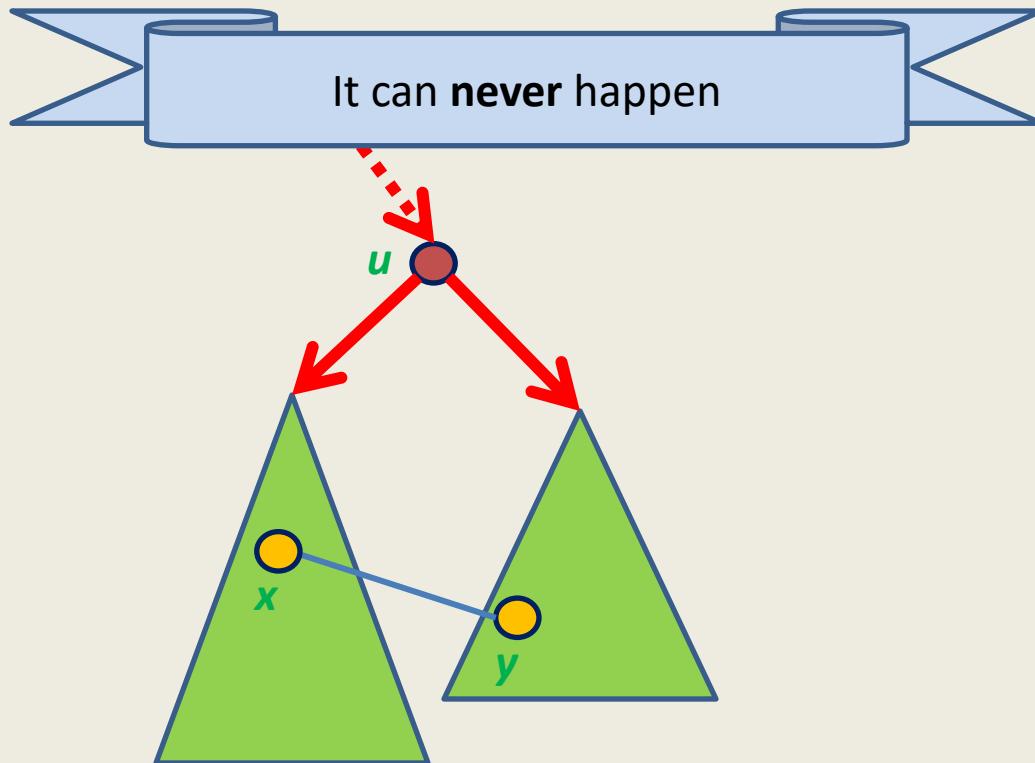
If the edge is a **non-tree** edge :

- Edge between **ancestor** and **descendant** in DFS tree.



No

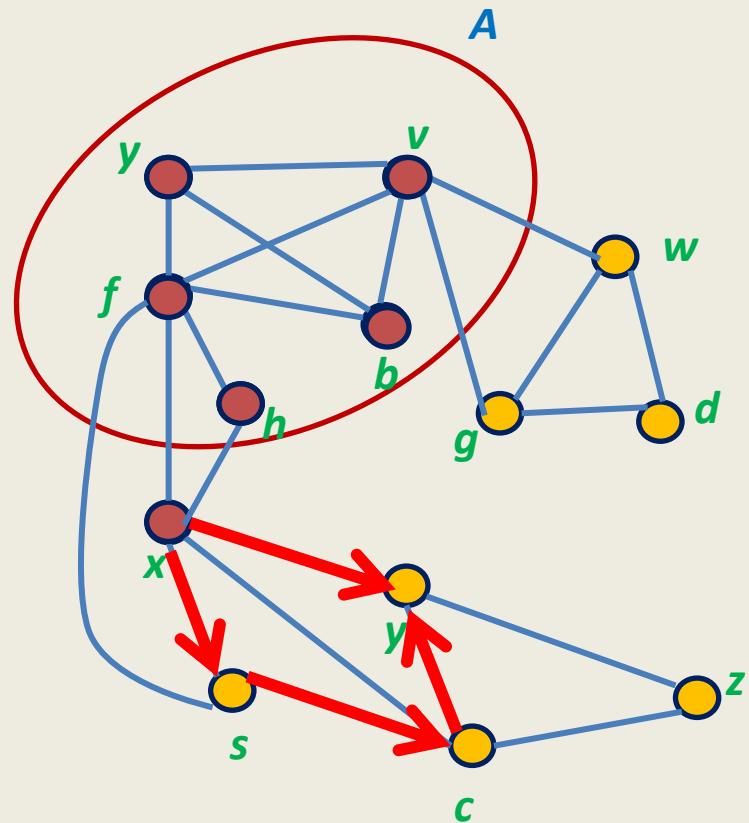
How will an edge appear in DFS traversal ?



How will an edge appear in DFS traversal ?



How will an edge appear in DFS traversal ?



How will an edge appear in DFS traversal ?

A short proof:

Let (x,y) be a non-tree edge.

Let x get visited before y .

Question:

If we remove all vertices visited prior to x , does y still lie in the connected component of x ?

Answer: yes.

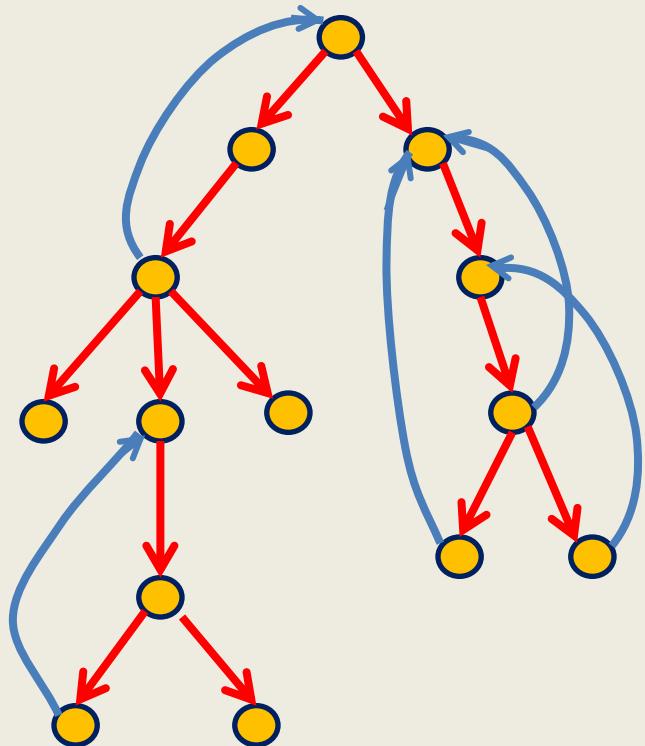


DFS pursued from x will have a path to y in DFS tree.

Hence x must be ancestor of y in the DFS tree.

Always remember

the following **picture** for DFS traversal



non-tree edge → **back** edge

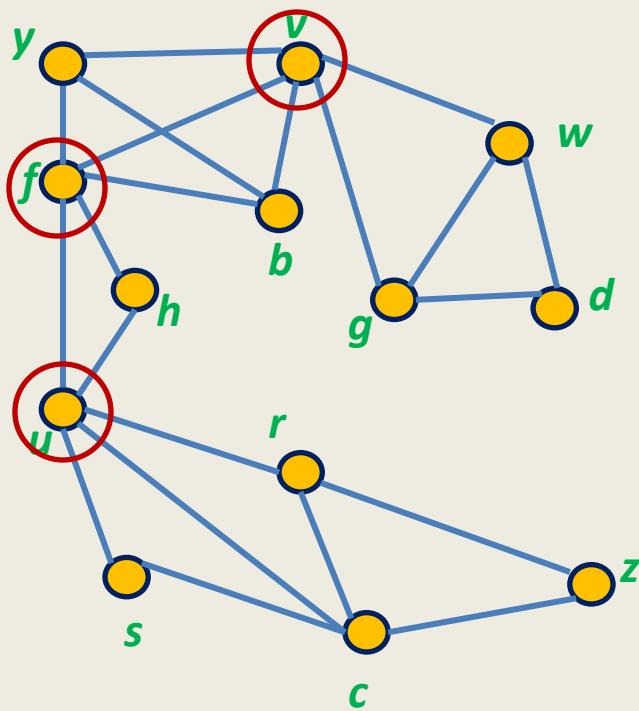
This is called **DFS representation** of the graph.
It plays a key role in the design of every efficient algorithm.

A novel application of DFS traversal

Determining if a graph G is **biconnected**

Definition: A connected graph is said to be **biconnected** if there does not exist any vertex whose removal disconnects the graph.

Motivation: To design **robust** networks
(immune to any single node failure).



Is this graph biconnected ?

No.

A trivial algorithms for checking bi-connectedness of a graph

- For each vertex v , determine if $G \setminus \{v\}$ is connected
(One may use either **BFS** or **DFS** traversal here)

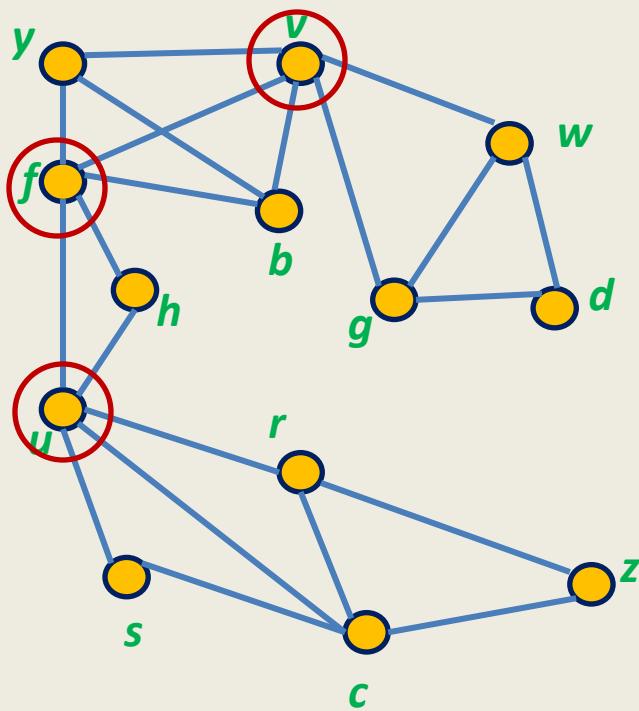
Time complexity of the trivial algorithm : **O(mn)**

An $\mathbf{O(m + n)}$ time algorithm

A single DFS traversal

An $\mathbf{O}(m + n)$ time algorithm

- A formal **characterization** of the problem.
(articulation points)
- Exploring relationship between articulation point & DFS tree.
- Using the relation **cleverly** to design an efficient algorithm.



This graph is NOT **biconnected**

The removal of any of $\{v, f, u\}$ can destroy connectivity.

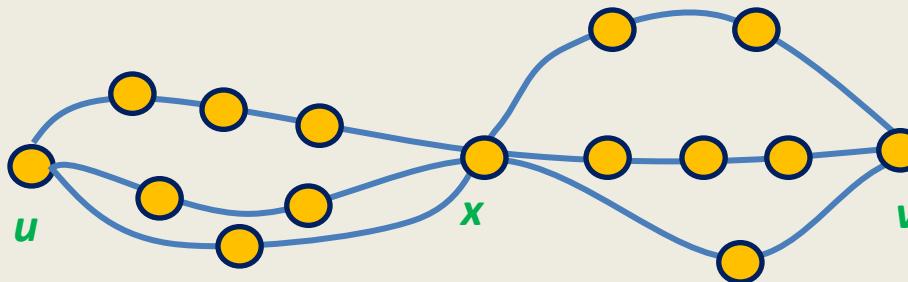
v, f, u are called the **articulation points** of G .

A formal definition of articulation point

Definition: A vertex x is said to be **articulation point** if

$\exists \ u, v$ different from x

such that every path between u and v passes through x .

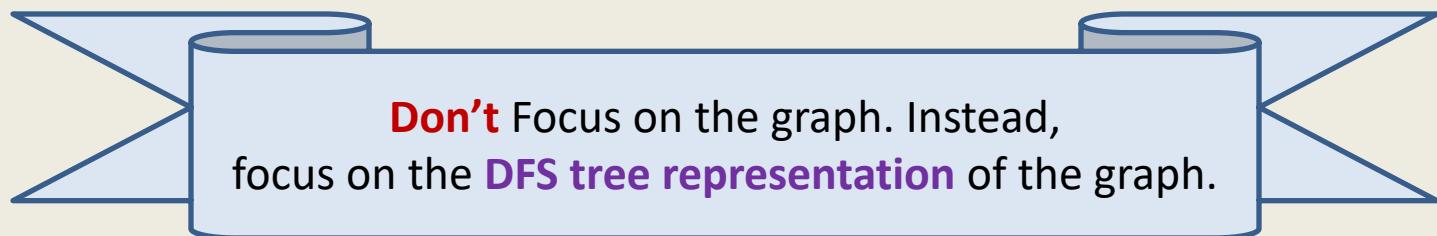


Observation: A graph is biconnected if none of its vertices is an articulation point.

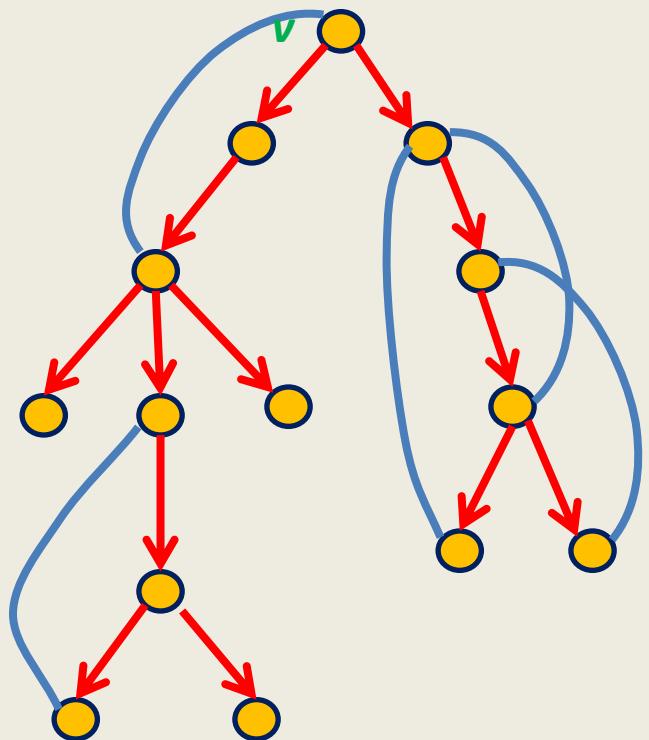
AIM:

Design an **algorithm** to compute all **articulation points** in a given graph.

Articulation points and DFS traversal



Some observations

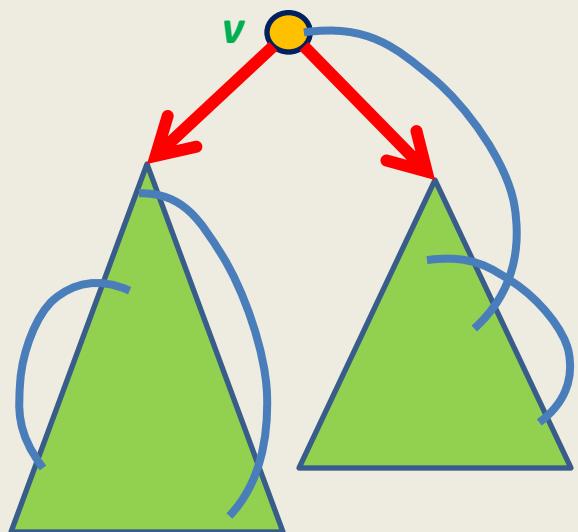


Question: When can a leaf node be an a.p. ?

Answer: Never

Question: When can root be an a.p. ?

Some observations

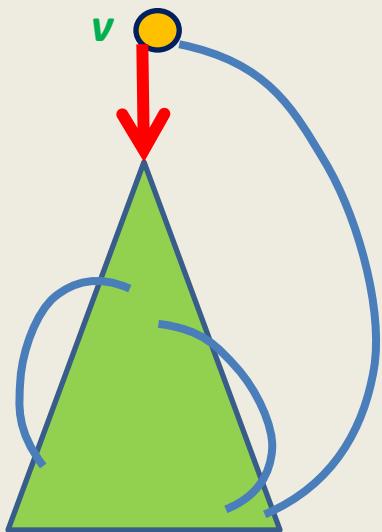


Question: When can a **leaf node** be an **a.p.** ?

Answer: Never

Question: When can **root** be an **a.p.** ?

Some observations



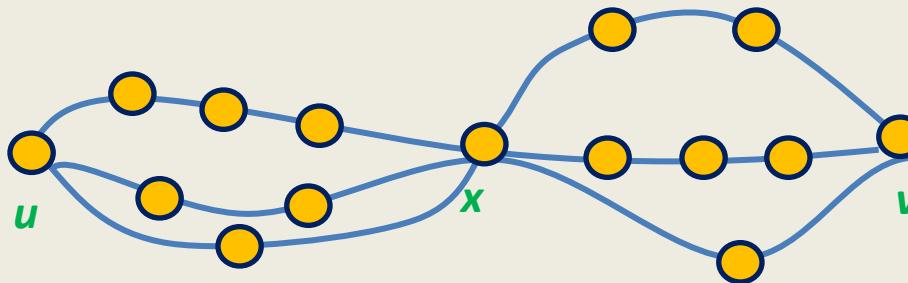
Question: When can a **leaf node** be an a.p. ?

Answer: Never

Question: When can **root** be an a.p. ?

Answer: Iff it has two or more children.

Some observations

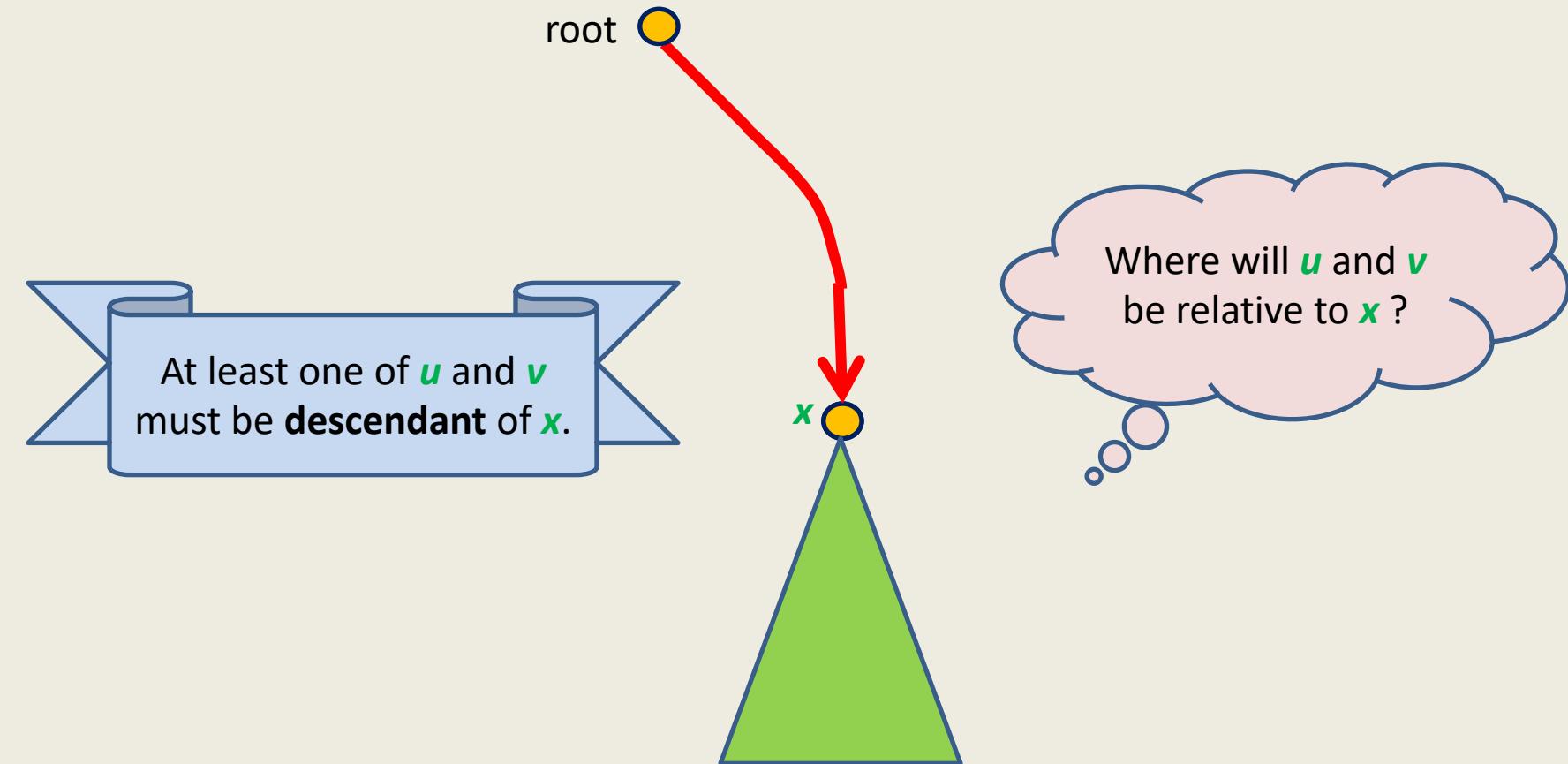


AIM:

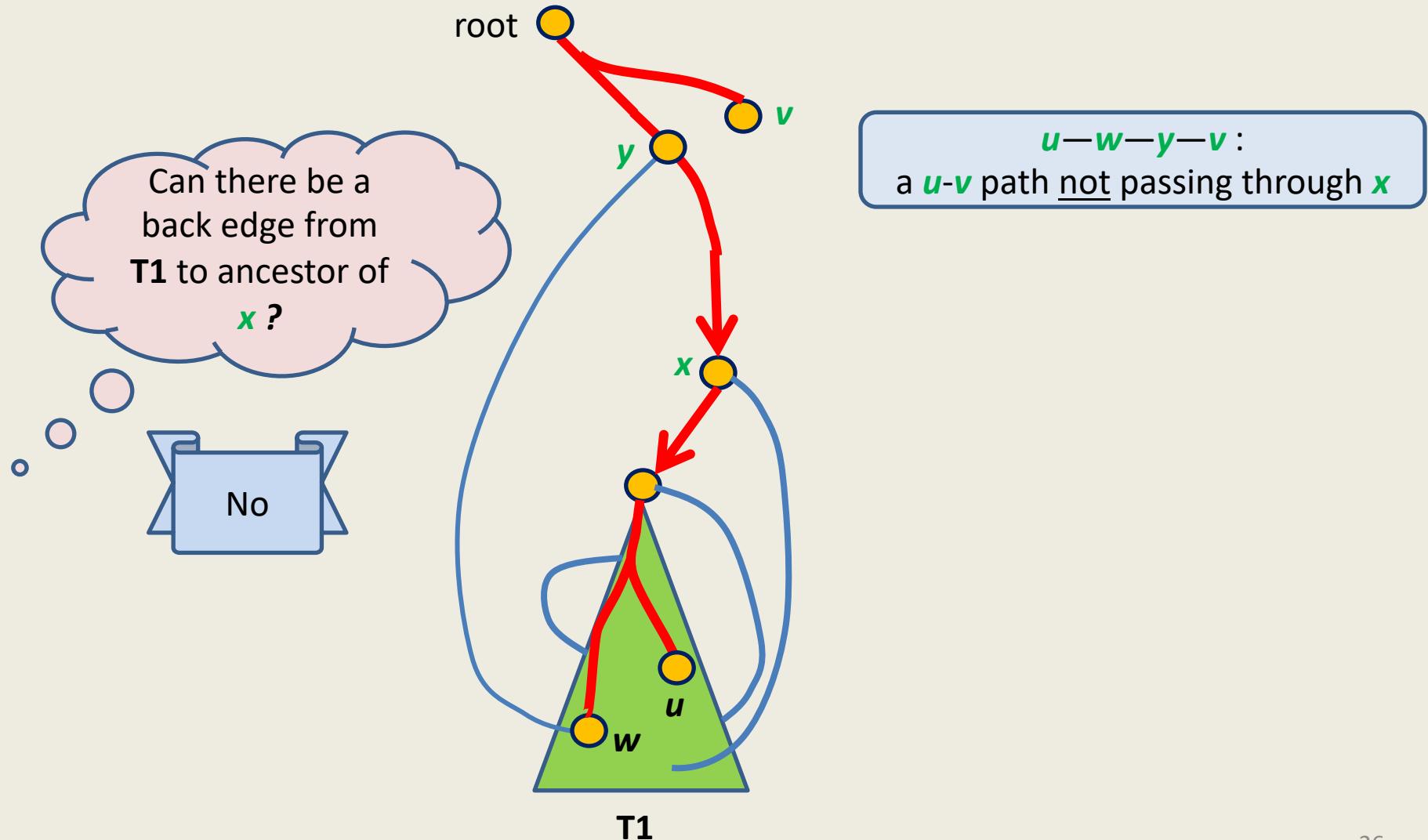
To find **necessary and sufficient conditions** for an **internal node** to be **articulation point**.

How will **x** look like
in **DFS tree** ?

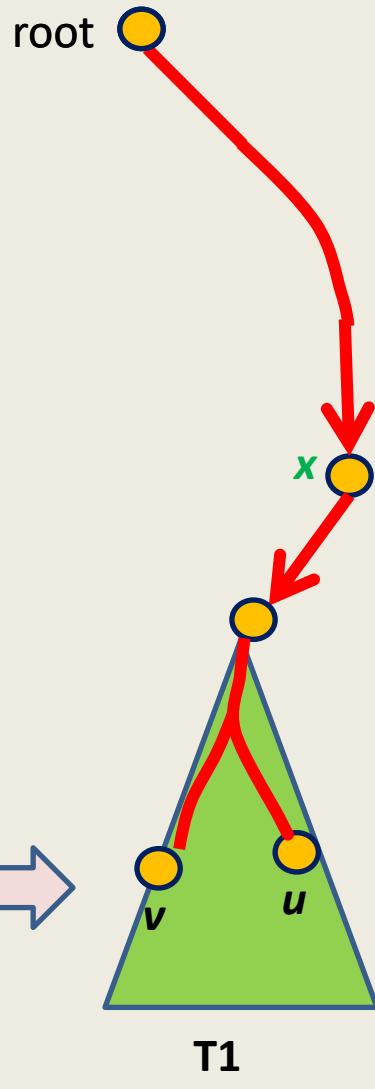
conditions for an internal node to be articulation point.



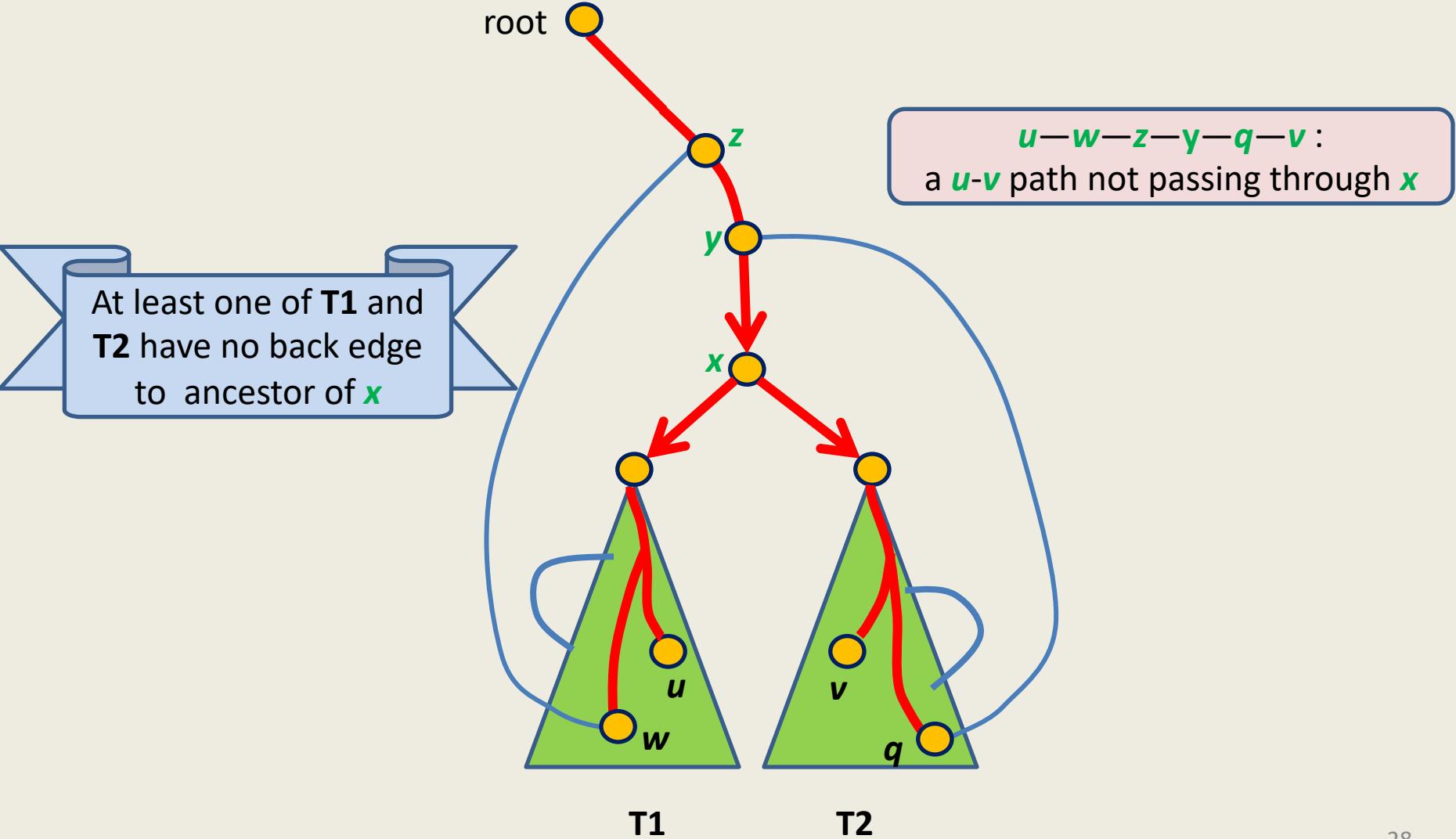
Case 1: Exactly one of u and v is a descendant of x in DFS tree



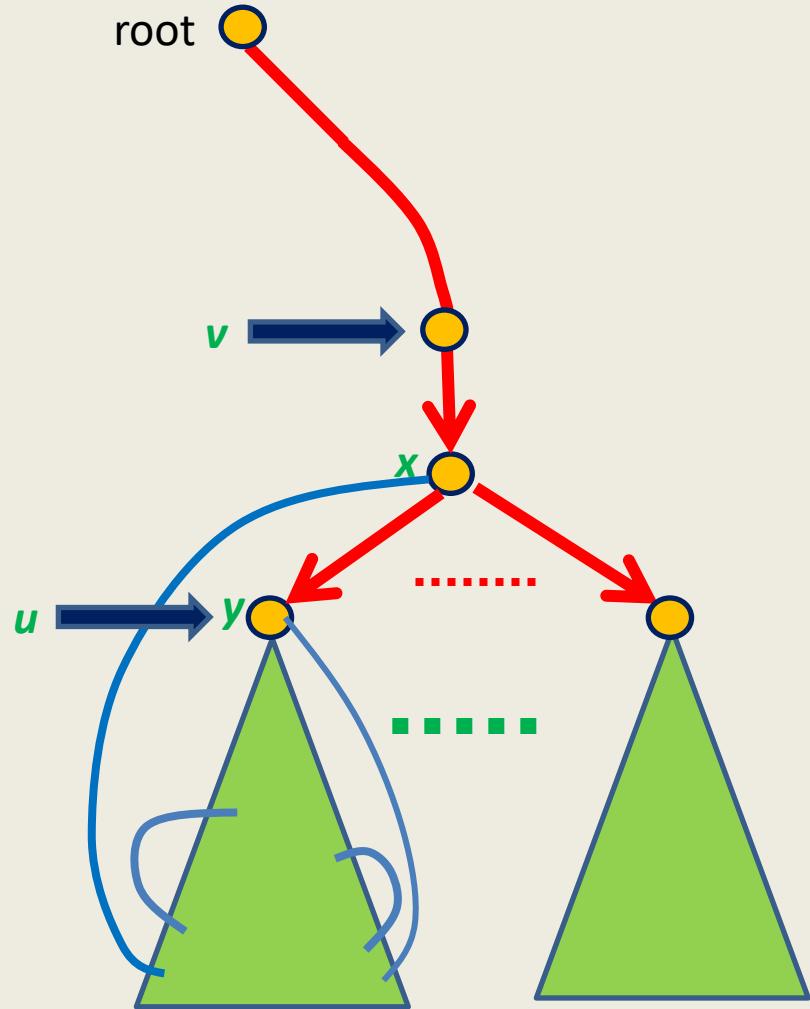
Case 2: both u and v are descendants of x in DFS tree



Case 2: both u and v are descendants of x in DFS tree

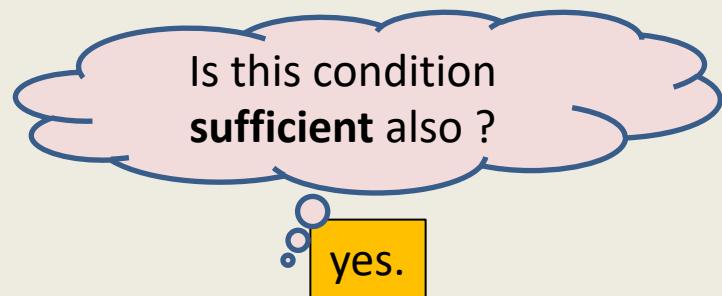


Necessary condition for x to be articulation point



Necessary condition:

x has **at least one child y** s.t.
there is **no** back edge
from **subtree(y)** to **ancestor of x** .



Articulation points and DFS

Let $G=(V,E)$ be a connected graph.

Perform **DFS** traversal from any graph and get a DFS tree T .

- No leaf of T is an **articulation point**.
- root of T is an **articulation point** if and only if it has more than one child.
- For any internal node ... ??

Theorem1 : An internal node x is **articulation point if and only if**

it has a child y such that

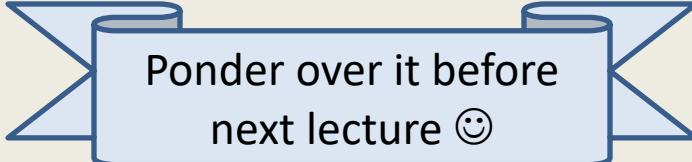
there is **no** back edge

from **subtree(y)** to any ancestor of x .

Efficient algorithm for Articulation points

Use Theorem 1

Exploit recursive nature of DFS



Ponder over it before
next lecture ☺

Data Structures and Algorithms

(ESO207)

Lecture 27

- Quick revision of Depth First Search (**DFS**) Traversal
- An $O(m + n)$:algorithm for **biconnected components** of a graph

Quick revision of Depth First Search (DFS**) Traversal**

DFS traversal of G

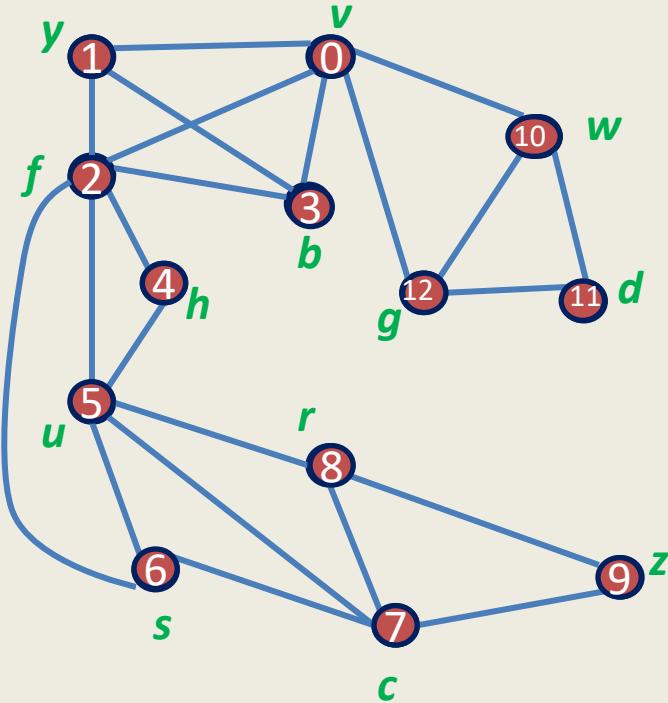
DFS(v)

```
{ Visited( $v$ )  $\leftarrow$  true; DFN[ $v$ ]  $\leftarrow$  dfn ++;  
    For each neighbor  $w$  of  $v$   
    {      if (Visited( $w$ ) = false)  
        { DFS( $w$ ) ;  
            .....;  
        }  
        .....;  
    }  
}
```

DFS-traversal(G)

```
{ dfn  $\leftarrow$  0;  
    For each vertex  $v \in V$  { Visited( $v$ )  $\leftarrow$  false }  
    For each vertex  $v \in V$  { If (Visited( $v$ ) = false) DFS( $v$ ) }  
}
```

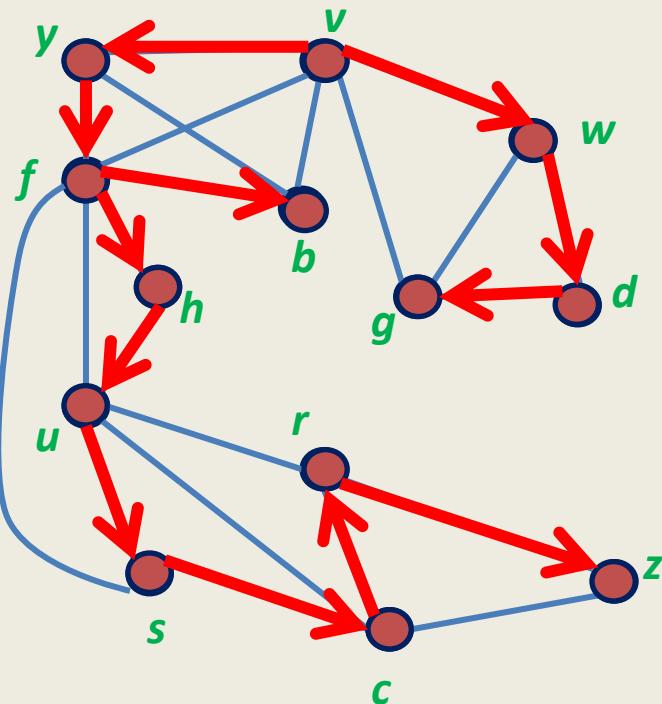
DFN number



DFN[*x*] :

The number at which *x* gets visited during DFS traversal.

$\text{DFS}(v)$ computes a tree rooted at v



A DFS tree rooted at v

If x is ancestor of y then

$$\text{DFN}[x] < \text{DFN}[y]$$

Question: Is a DFS tree unique ?

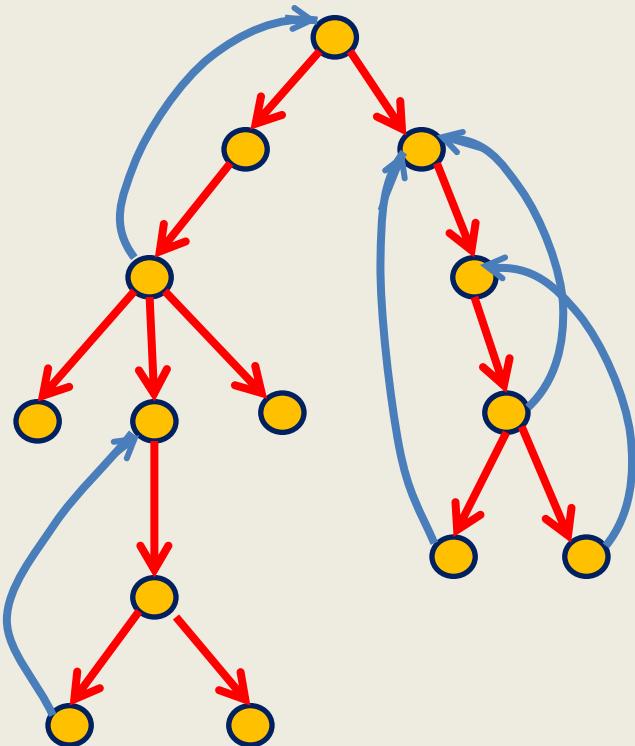
Answer: No.

Question:

Can any rooted tree be obtained through DFS ?

Answer: No.

**Always remember
this picture**



non-tree edge → **back** edge

A DFS representation of the graph

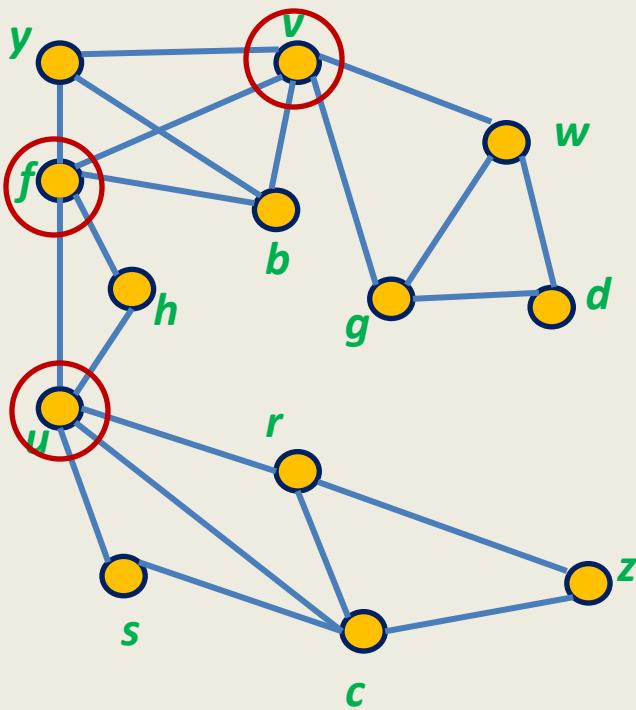
Verifying bi-connectivity of a graph

An $O(m + n)$ time algorithm

A single **DFS** traversal

An $\mathbf{O}(m + n)$ time algorithm

- A formal **characterization** of the problem.
(articulation points)
- Exploring relationship between articulation point & DFS tree.
- Using the relation **cleverly** to design an efficient algorithm.



This graph is NOT **biconnected**

The removal of any of $\{v, f, u\}$ can destroy connectivity.

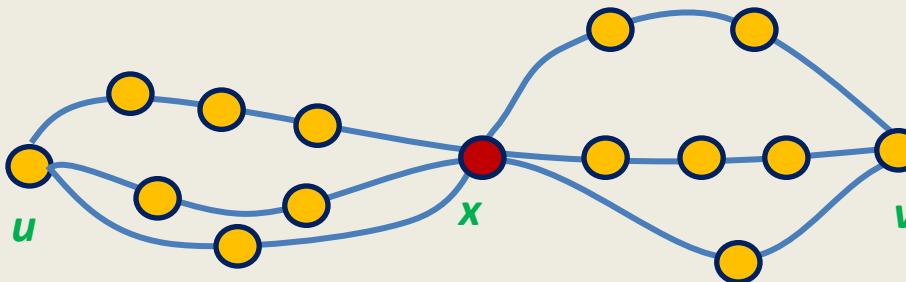
v, f, u are called the **articulation points** of G .

A formal definition of articulation point

Definition: A vertex x is said to be **articulation point** if

$\exists \ u, v$ different from x

such that every path between u and v passes through x .

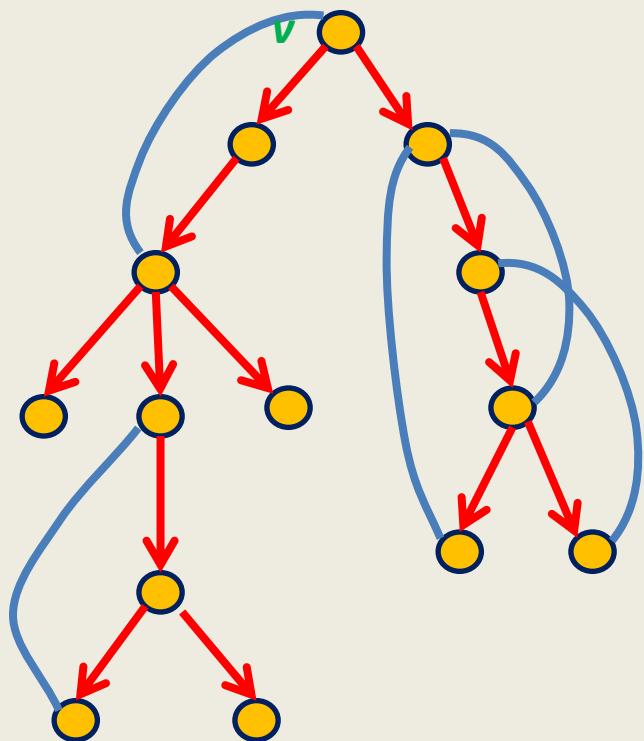


Observation: A graph is biconnected if none of its vertices is an articulation point.

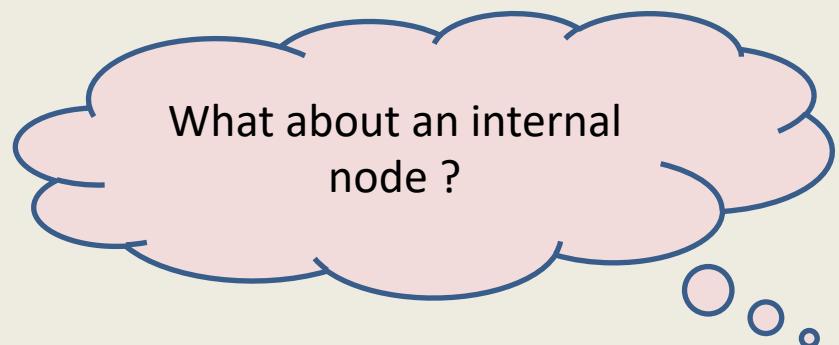
AIM:

Design an **algorithm** to compute all **articulation points** in a given graph.

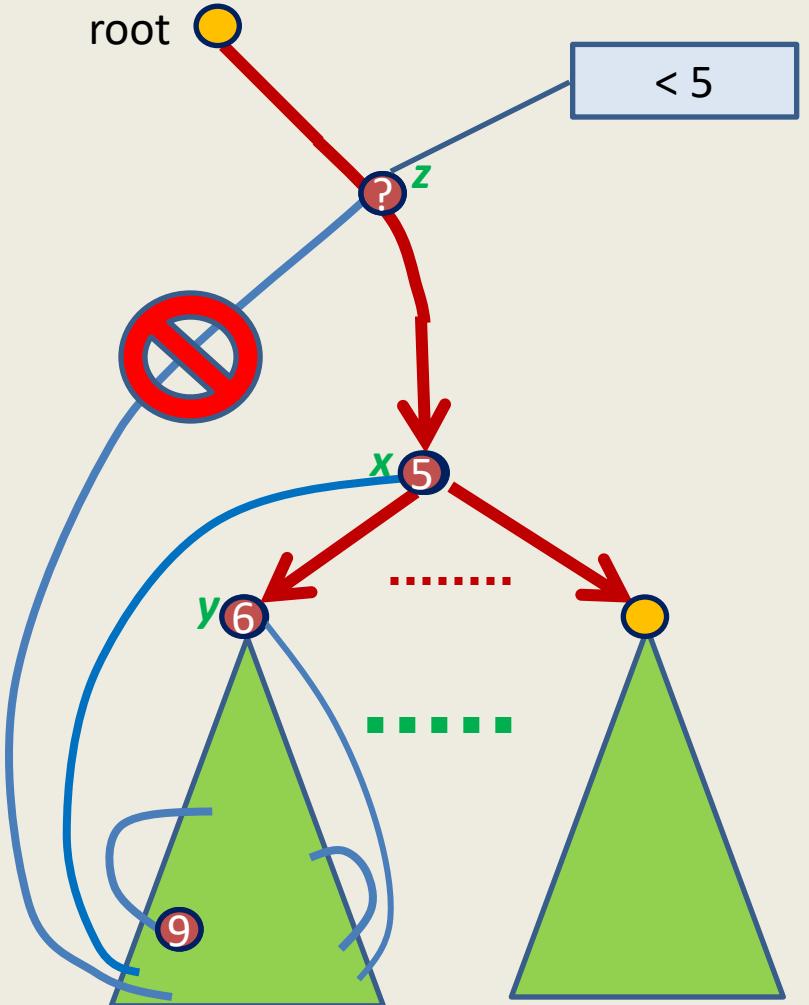
Some observations



- A **leaf node** can never be an **a.p.** ?
- **Root** is an **a.p.** iff it has two or more children.



Necessary and Sufficient condition for x to be articulation point



Theorem 1:

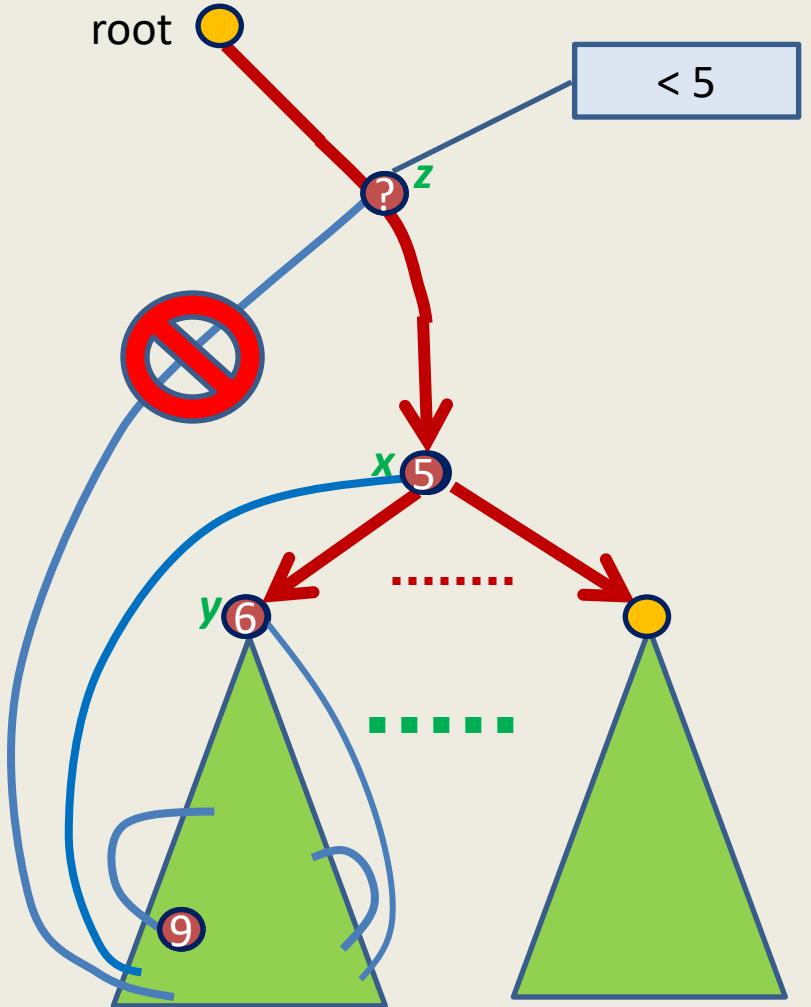
An internal node x is **articulation point** iff
 x has at least one child y s.t.
no back edge from **subtree(y)** to **ancestor of x** .

→ No back edge from subtree(y) going to a vertex “higher” than x .

How to define the notion “higher” than x ?

Use DFN numbering

Necessary and Sufficient condition for x to be articulation point



Theorem 1:

An internal node x is **articulation point** iff
 x has at least one child y s.t.
no back edge from **subtree(y)** to **ancestor of x**

Invent a new function

High_pt(v):

DFN of the highest ancestor of v
to which there is a back edge from **subtree(v)**.

Theorem 2:

An internal node x is **articulation point** iff it has a child, say y , in **DFS** tree such that

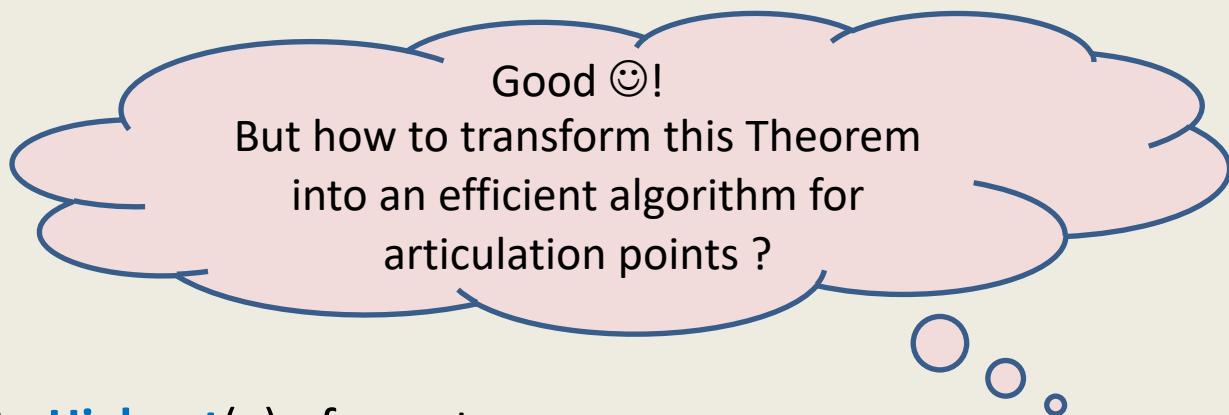
$$\text{High_pt}(\mathbf{y}) \geq \text{DFN}(\mathbf{x}).$$

Theorem2:

An internal node x is **articulation point iff**

it has a child, say y , in **DFS** tree such that

$$\text{High_pt}(y) \geq \text{DFN}(x).$$

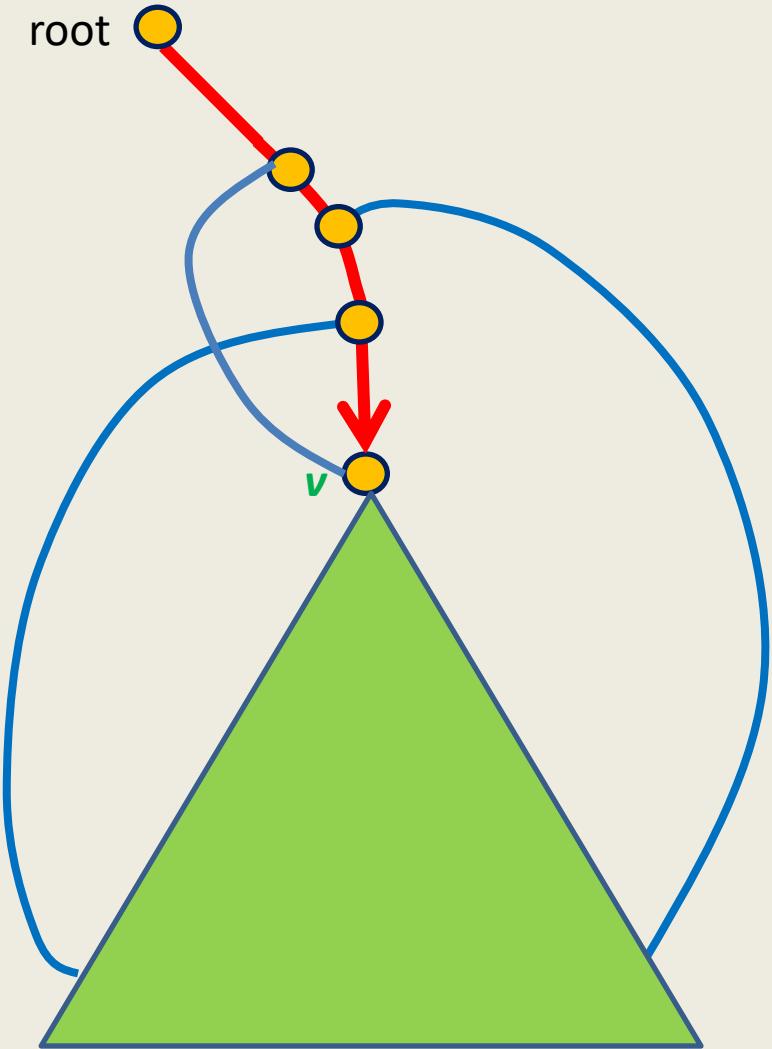


In order to compute **High_pt(v)** of a vertex v ,
we have to traverse the adjacency lists of all vertices of subtree $T(v)$.

→ **O(m)** time in the worst case to compute **High_pt(v)** of a vertex v .

→ **O(mn)** time algorithm 😞

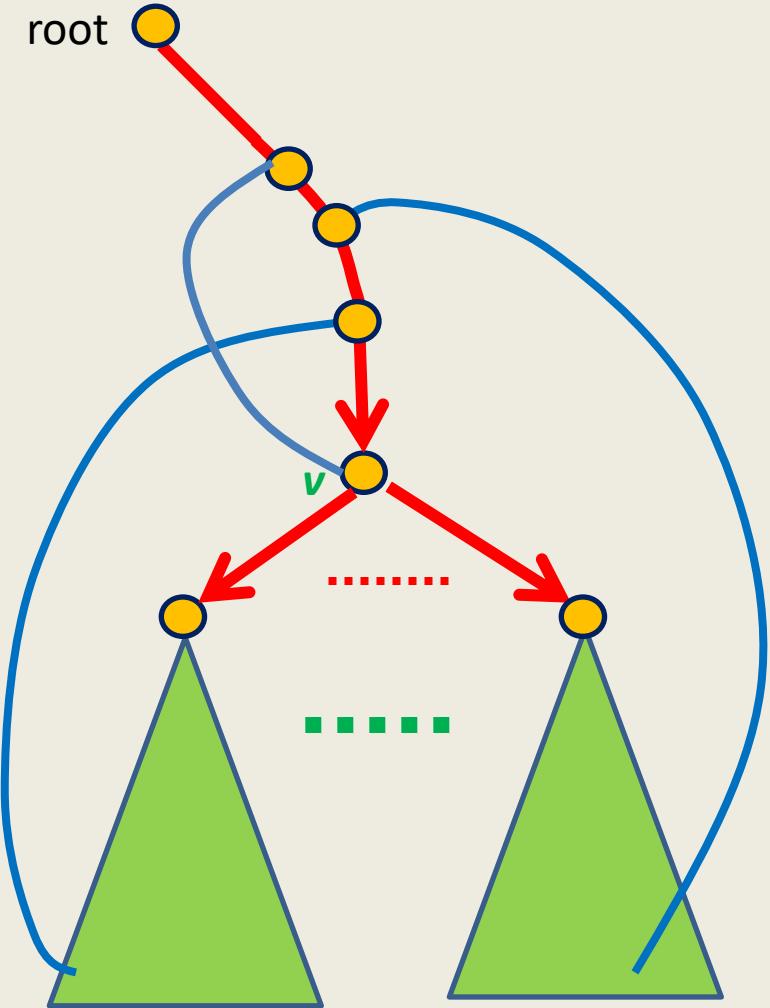
How to compute $\text{High_pt}(v)$ efficiently ?



Question: Can we express $\text{High_pt}(v)$ in terms of its **children** and **proper ancestors**?

Exploit
recursive structure of
DFS tree.

How to compute $\text{High_pt}(v)$ efficiently ?



Question: Can we express $\text{High_pt}(v)$ in terms of its **children** and **proper ancestors**?

$\text{High_pt}(v) =$

$$\min_{(v,w) \in E} \left\{ \begin{array}{l} \text{High_pt}(w) \\ \text{DFN}(w) \end{array} \right. \begin{array}{l} \text{If } w = \text{child}(v) \\ \text{If } w = \text{proper} \\ \text{ancestor of } v \end{array}$$

The **novel** algorithm

Output : an array **AP[]** s.t.

AP[*v*]= true if and only if *v* is an articulation point.

Algorithm for articulation points in a graph G

DFS(v)

```
{ Visited( $v$ )  $\leftarrow$  true; DFN[ $v$ ]  $\leftarrow$  dfn ++; High_pt[ $v$ ]  $\leftarrow$   $\infty$  ;
```

For each neighbor w of v

```
{ if (Visited( $w$ ) = false)
```

```
{   DFS( $w$ ) ; Parent( $w$ )  $\leftarrow$   $v$ ;
```

```
.....;
```

```
.....;
```

```
}
```

```
.....;
```

```
}
```

```
}
```

DFS-traversal(G)

```
{ dfn  $\leftarrow$  0;
```

For each vertex $v \in V$ { Visited(v) \leftarrow false; AP[v] \leftarrow false }

For each vertex $v \in V$ { If (Visited(v) = false) DFS(v) }

```
}
```

Algorithm for articulation points in a graph G

DFS(v)

{ Visited(v) \leftarrow true; DFN[v] \leftarrow dfn ++; High_pt[v] \leftarrow ∞ ;

For each neighbor w of v

{ if (Visited(w) = false)

{ Parent(w) $\leftarrow v$; DFS(w);

High_pt(v) $\leftarrow \min(\text{High_pt}(v), \text{High_pt}(w))$;

If High_pt(w) \geq DFN[v] AP[v] \leftarrow true

}

Else if (Parent(v) $\neq w$)

High_pt(v) $\leftarrow \min(\text{DFN}(w), \text{High_pt}(v))$

}

}

DFS-traversal(G)

{ dfn $\leftarrow 0$;

For each vertex $v \in V$ { Visited(v) \leftarrow false; AP[v] \leftarrow false }

For each vertex $v \in V$ { If (Visited(v) = false) DFS(v) }

}

Conclusion

Theorem2 : For a given graph $G=(V,E)$, all **articulation points** can be computed in $O(m + n)$ time.

Data Structures

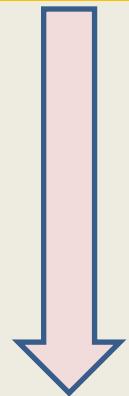
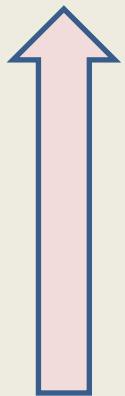
Lists: (arrays, linked lists)

Range of
efficient functions

Binary Heap

Binary Search Trees

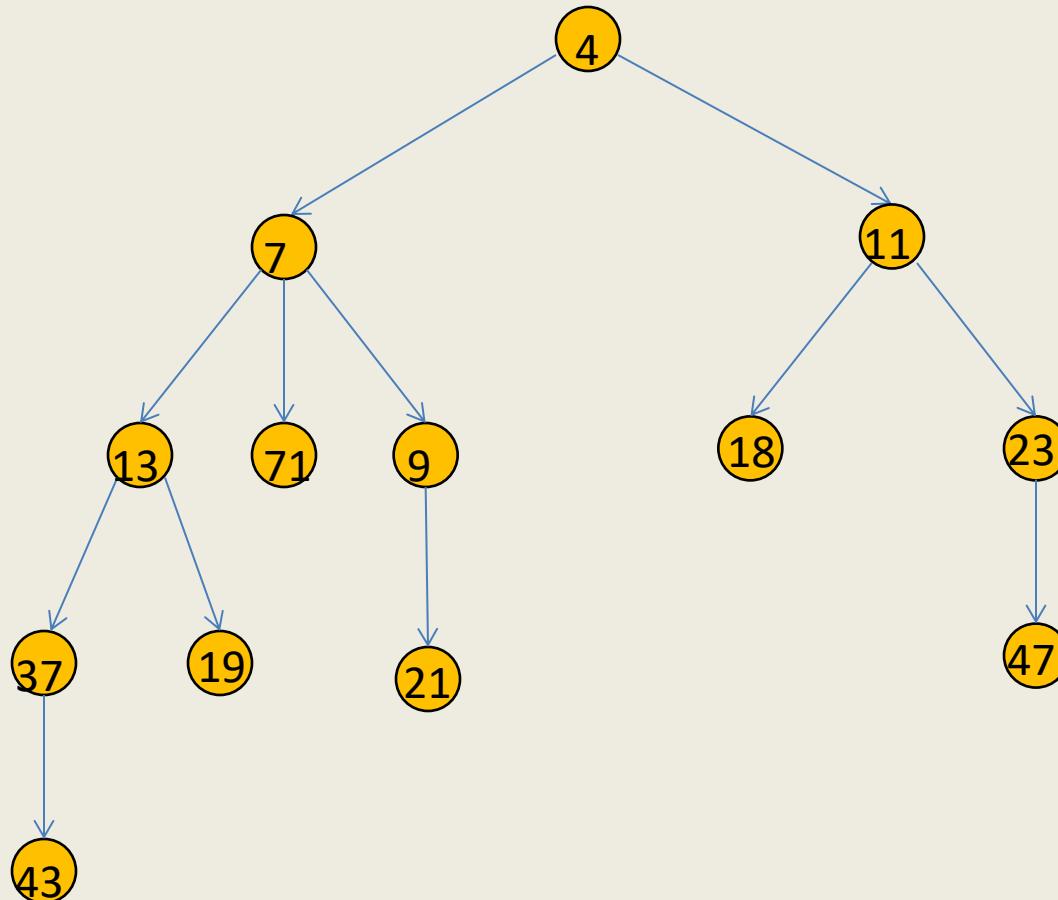
Simplicity



Heap

Definition: a tree data structure where :

value stored in a node < value stored in each of its children.



Operations on a heap

Query Operations

- **Find-min**: report the smallest key stored in the heap.

Update Operations

- **CreateHeap(H)** : Create an empty heap H .
- **Insert(x, H)** : Insert a new key with value x into the heap H .
- **Extract-min(H)** : delete the smallest key from H .
- **Decrease-key(p, Δ, H)** : decrease the value of the key p by amount Δ .
- **Merge(H_1, H_2)** : Merge two heaps H_1 and H_2 .

Why heaps when we can use a binary search tree ?

Compared to binary search trees, a heap is usually

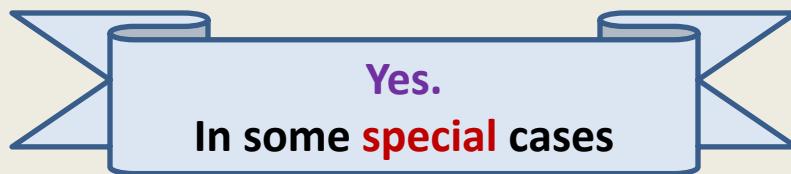
-- much simpler and

-- more efficient

Existing heap data structures

- **Binary heap**
- **Binomial heap**
- **Fibonacci heap**
- **Soft heap**

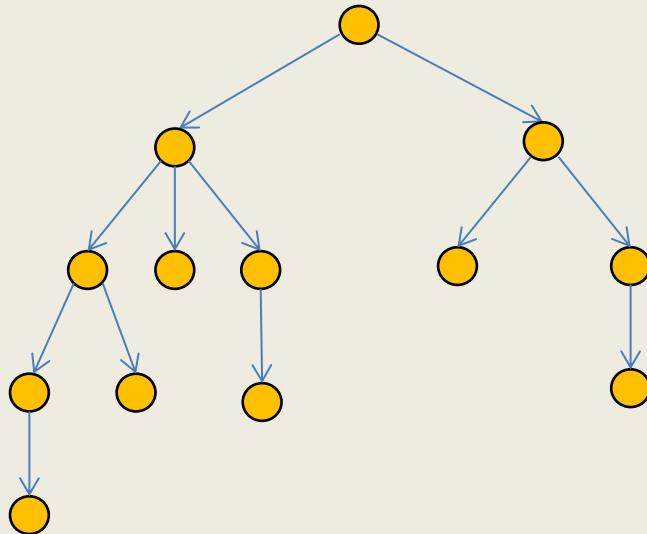
Can we implement a binary tree using an array ?





fundamental question

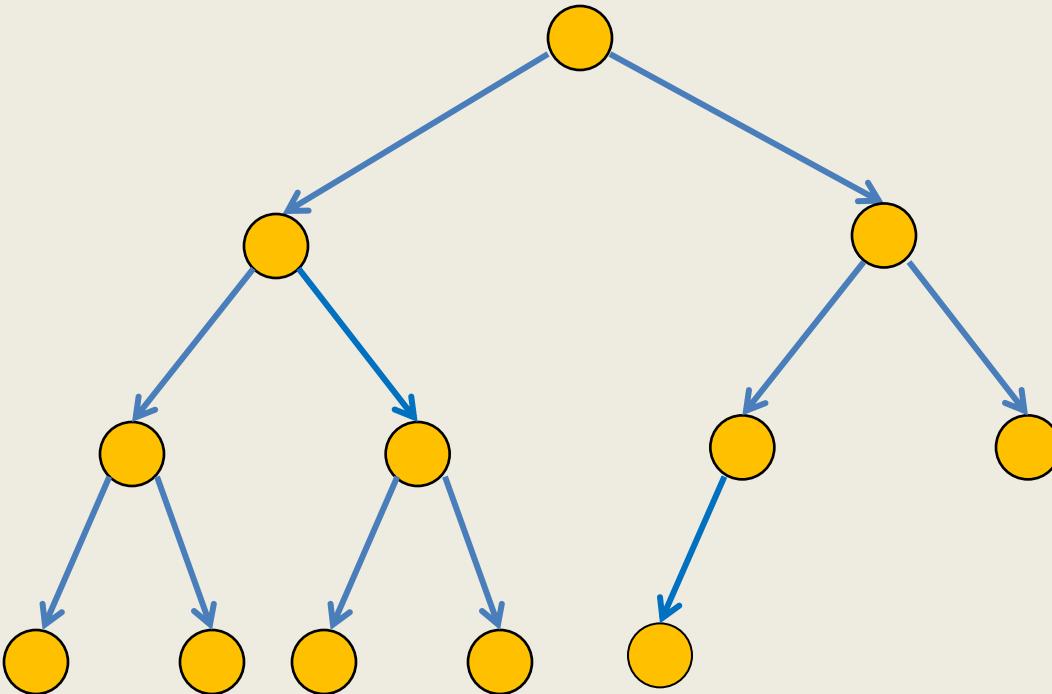
Question: What does the implementation of a tree data structure require ?



Answer: a mechanism to

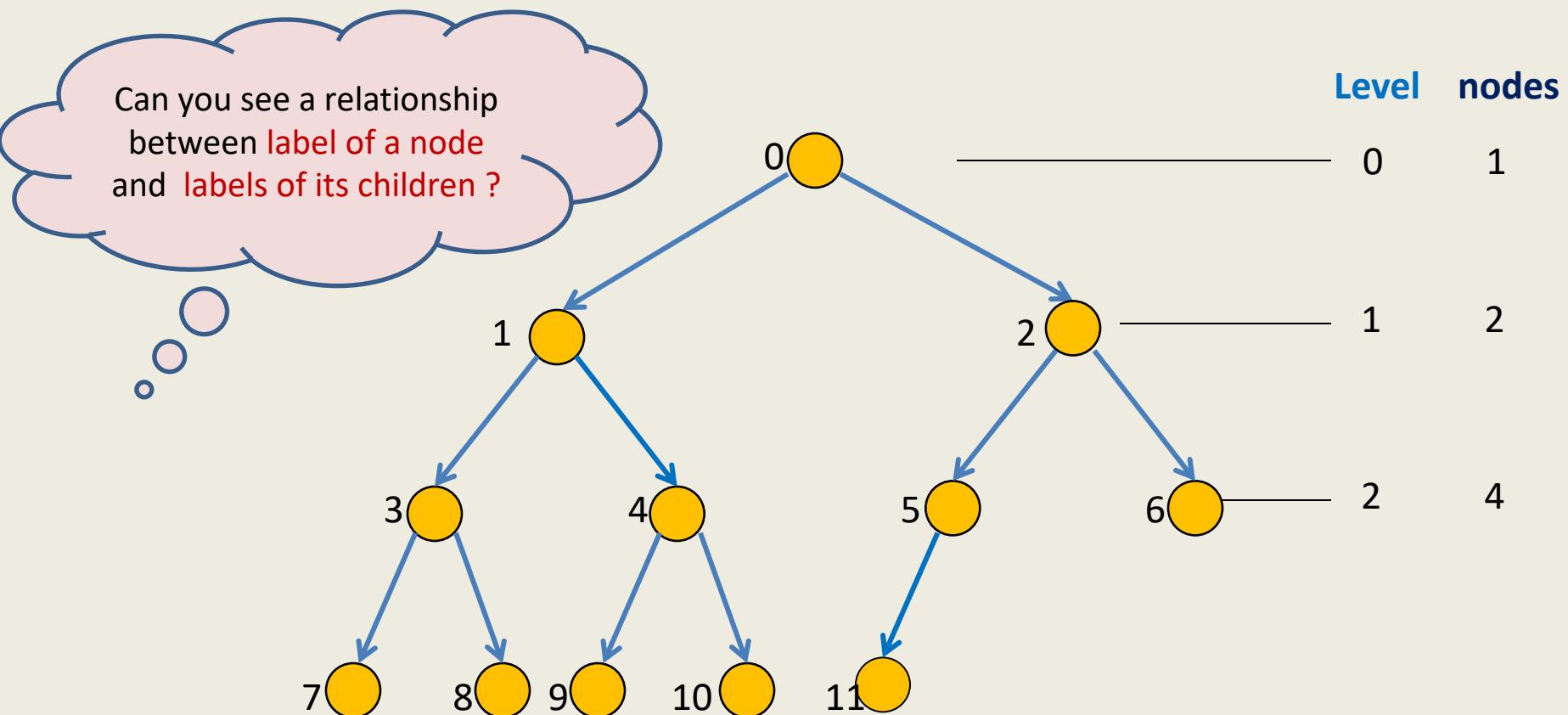
- access **parent** of a node
- access **children** of a node.

A **complete** binary tree



A complete binary of 12 nodes.

A complete binary tree



Think over it before next lecture.

Data Structures and Algorithms

(ESO207)

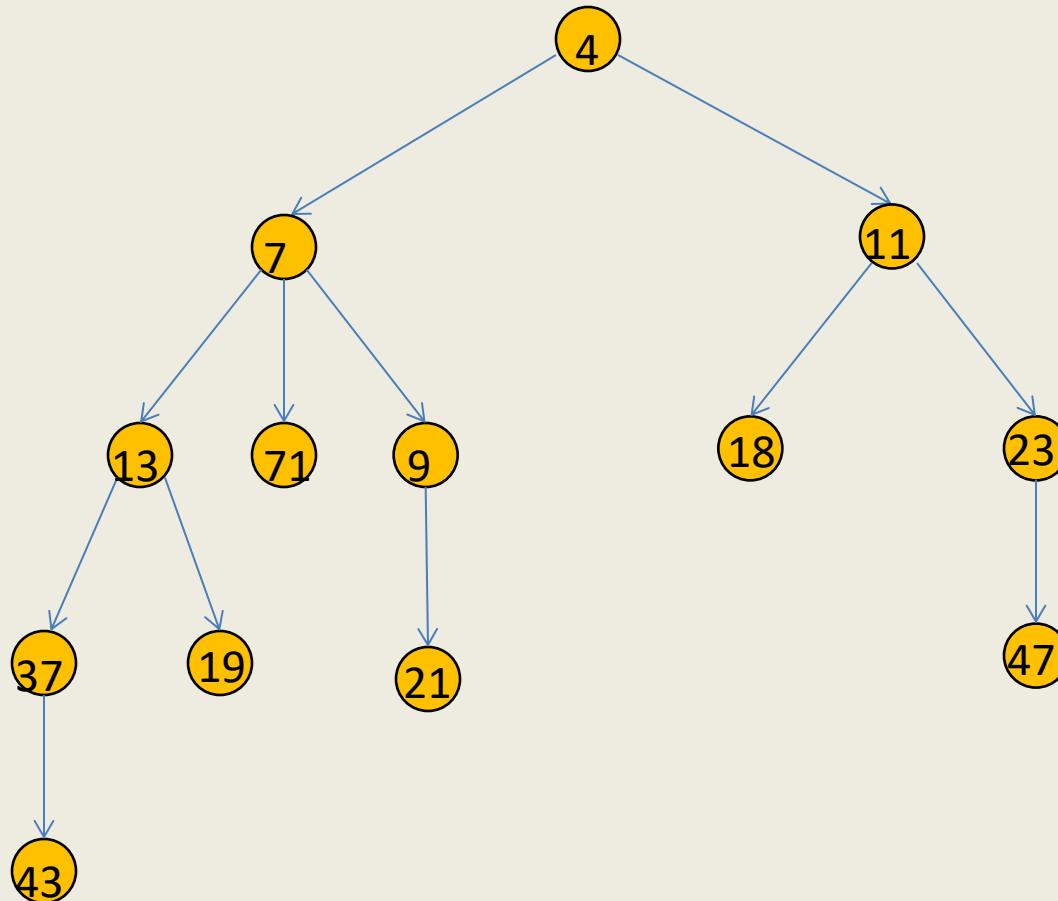
Lecture 28:

- **Heap** : an important tree data structure
- Implementing some **special binary tree** using an **array** !
- **Binary heap**

Heap

Definition: a tree data structure where :

value stored in a node < value stored in each of its children.



Operations on a heap

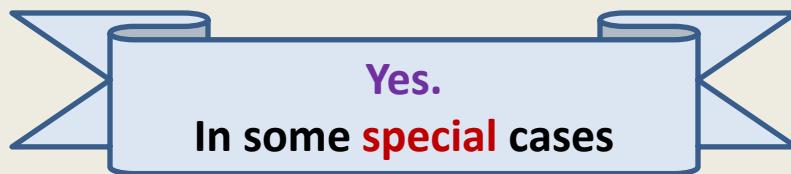
Query Operations

- **Find-min**: report the smallest key stored in the heap.

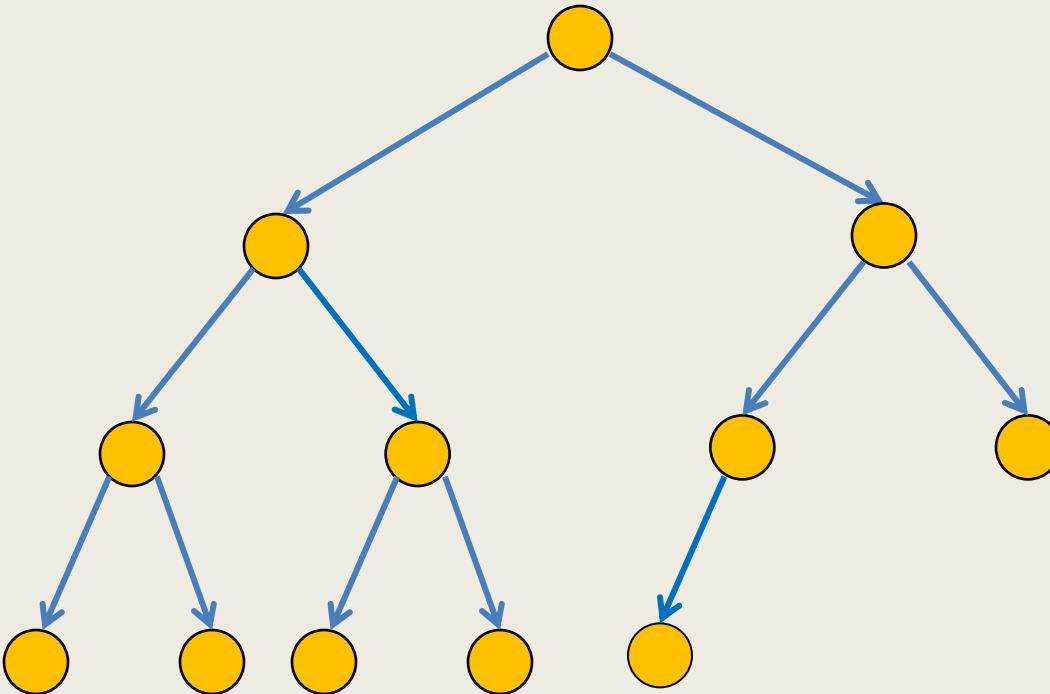
Update Operations

- **CreateHeap(H)** : Create an empty heap H .
- **Insert(x, H)** : Insert a new key with value x into the heap H .
- **Extract-min(H)** : delete the smallest key from H .
- **Decrease-key(p, Δ, H)** : decrease the value of the key p by amount Δ .
- **Merge(H_1, H_2)** : Merge two heaps H_1 and H_2 .

Can we implement a binary tree using an array ?

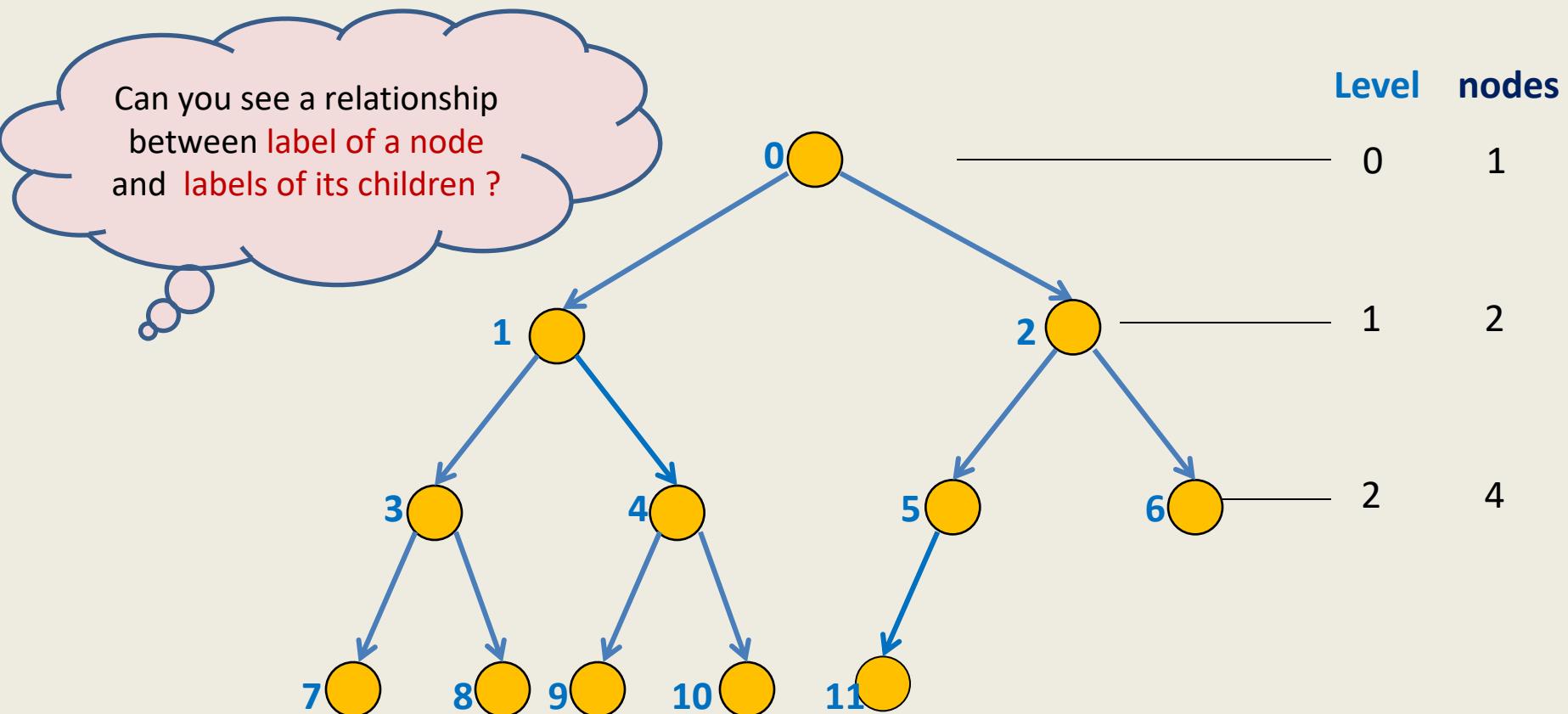


A **complete** binary tree



A complete binary of 12 nodes.

A complete binary tree



The label of the **leftmost node** at level $i = 2^i - 1$

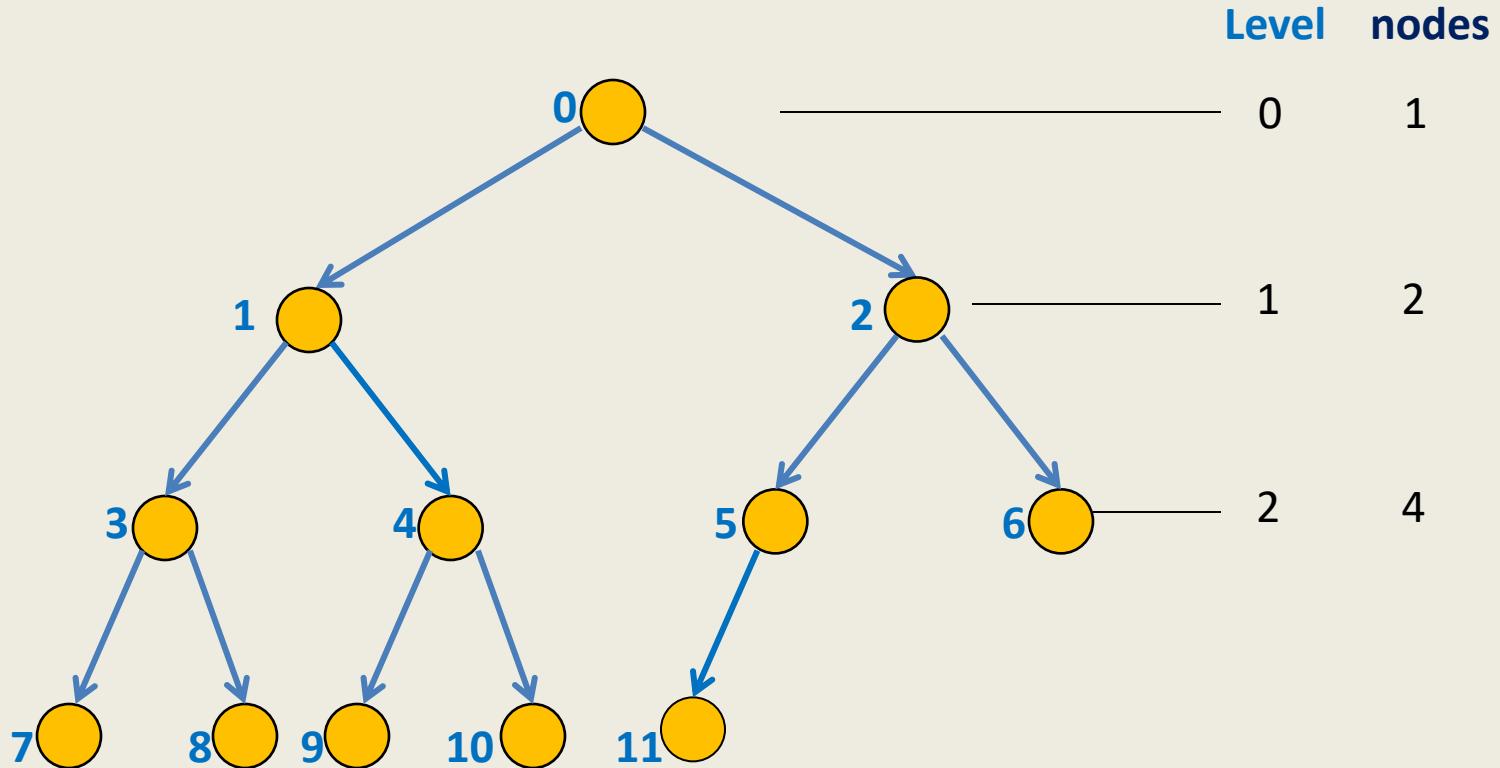
The label of a **node v** at level i

The label of the **left** child of v is= $2^{i+1} - 1 + 2(k - 1)$

The label of the **right** child of v is= $2^{i+1} + 2k - 2$

$i - 2$

A complete binary tree



Let v be a node with label j .

$$\text{Label of left child}(v) = 2j + 1$$

$$\text{Label of right child}(v) = 2j + 2$$

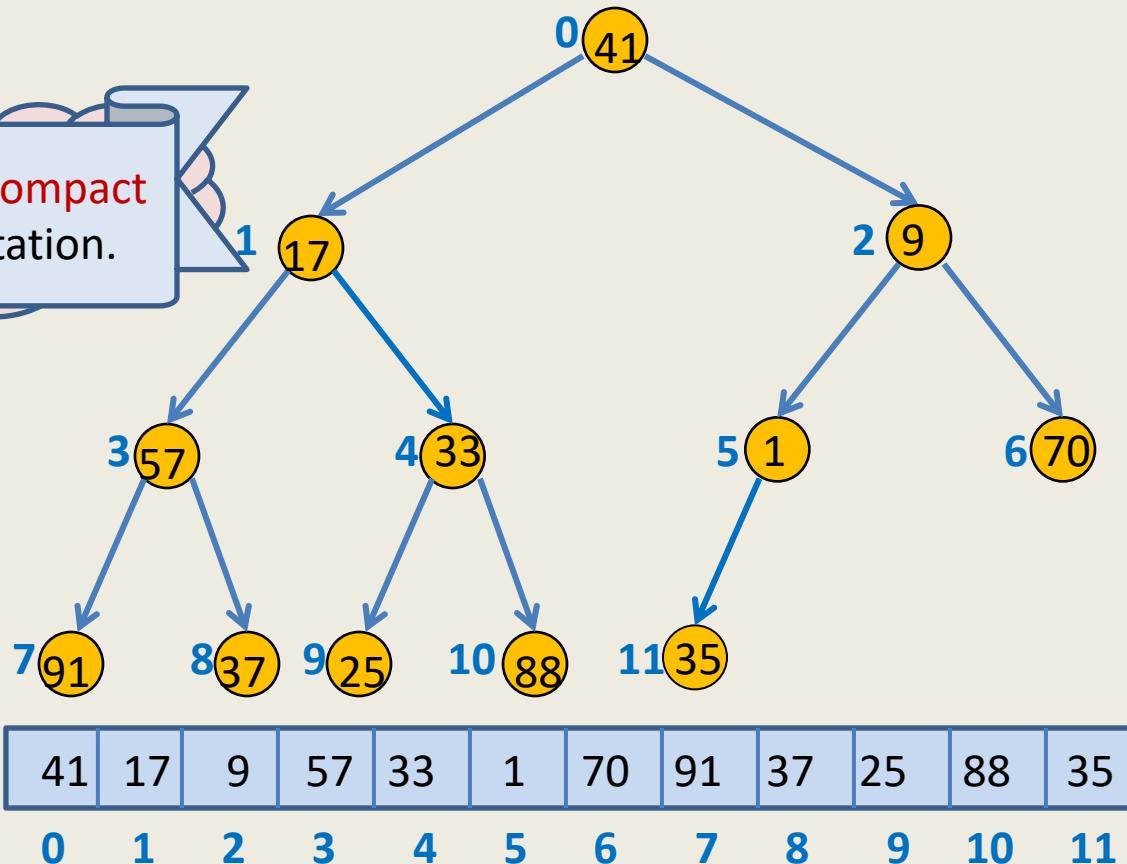
$$\text{Label of parent}(v) = \lfloor (j - 1)/2 \rfloor$$

A complete binary tree and array

Question: What is the relation between a complete binary trees and an array ?

Answer: A complete binary tree can be **implemented** by an array.

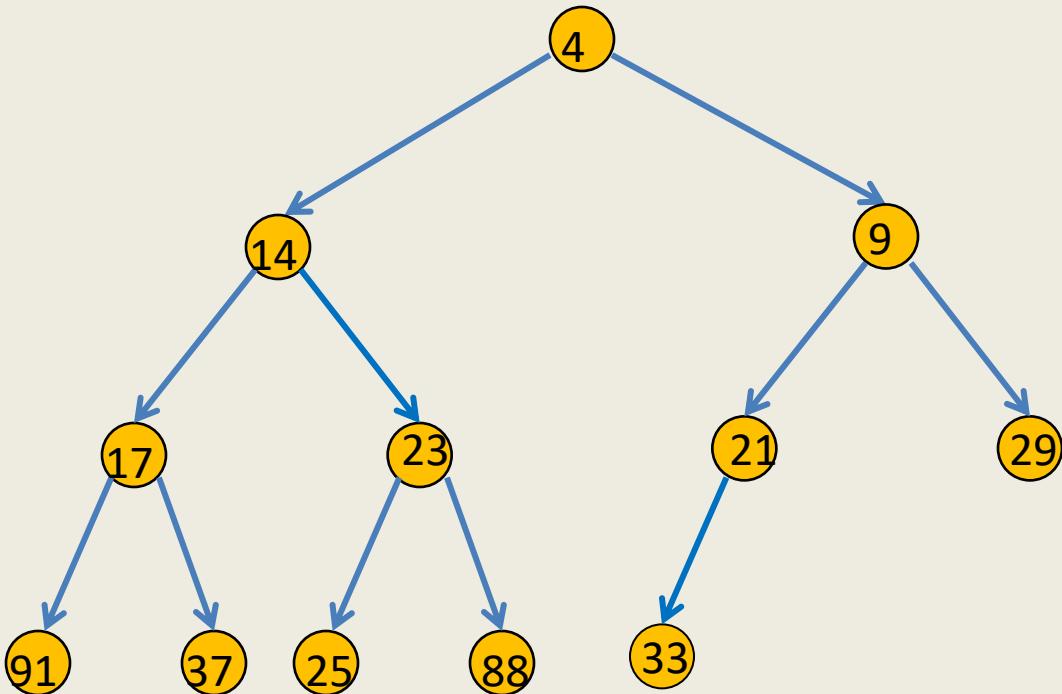
The most **compact** representation.



Binary heap

Binary heap

a **complete binary tree** satisfying **heap** property at each node.



H

4	14	9	17	23	21	29	91	37	25	88	33			
---	----	---	----	----	----	----	----	----	----	----	----	--	--	--

Implementation of a Binary heap

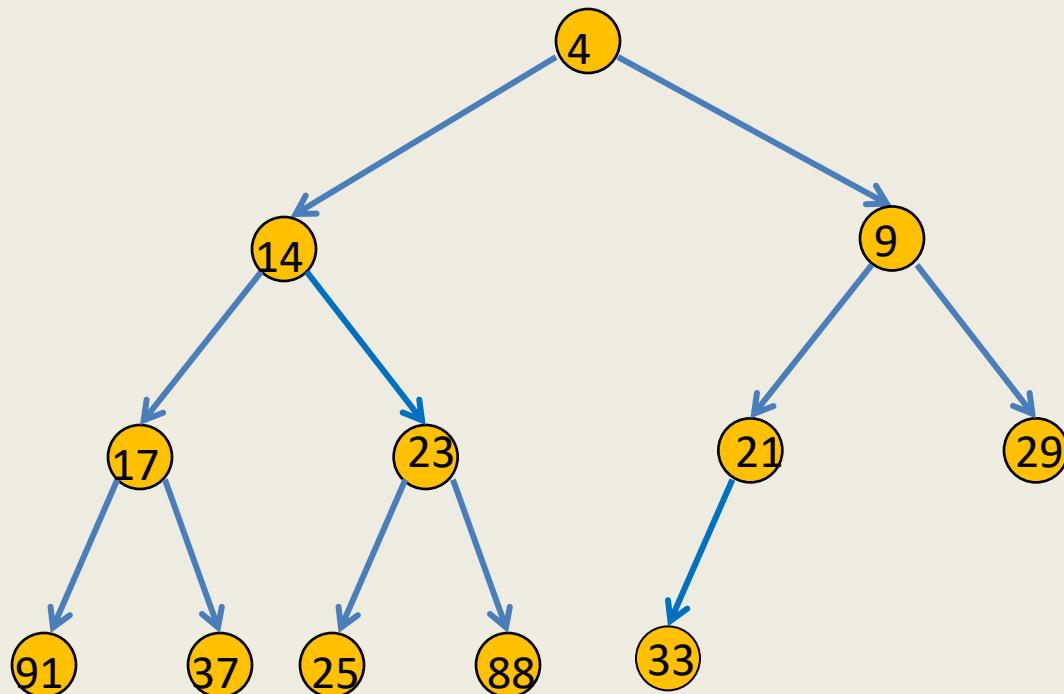
n

then we keep

- **H[]** : an **array** of size **n** used for storing the binary heap.
- **size** : a **variable** for the total number of keys currently in the heap.

Find_min(H)

Report $H[0]$.



H

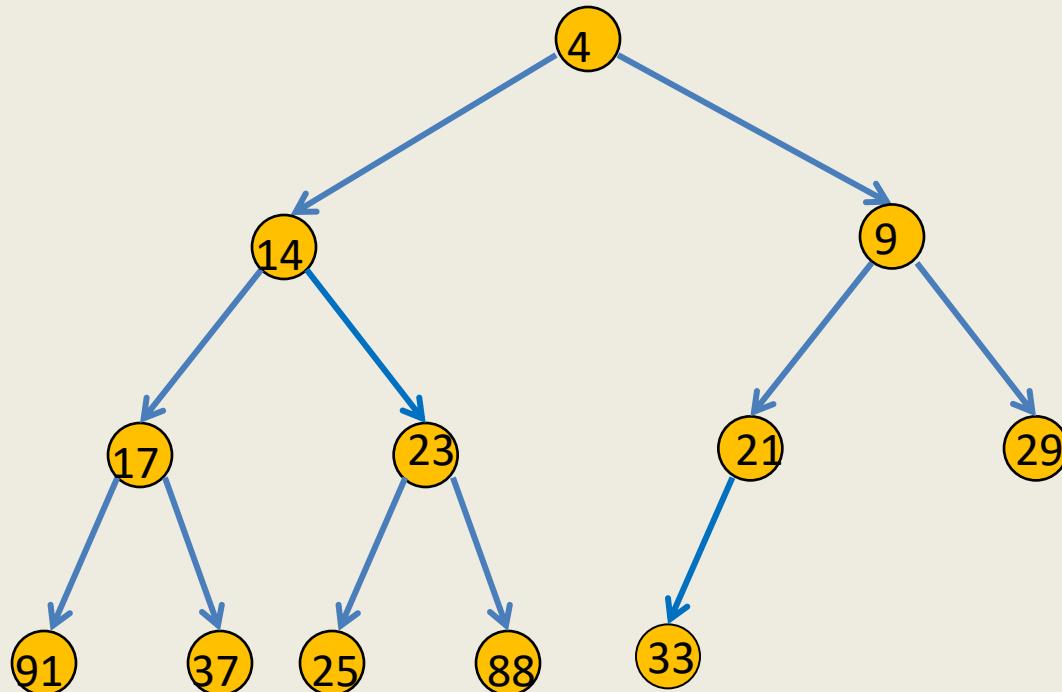
4	14	9	17	23	21	29	91	37	25	88	33			
---	----	---	----	----	----	----	----	----	----	----	----	--	--	--

Extract_min(H)

Think hard on designing efficient algorithm for this operation.

The challenge is:

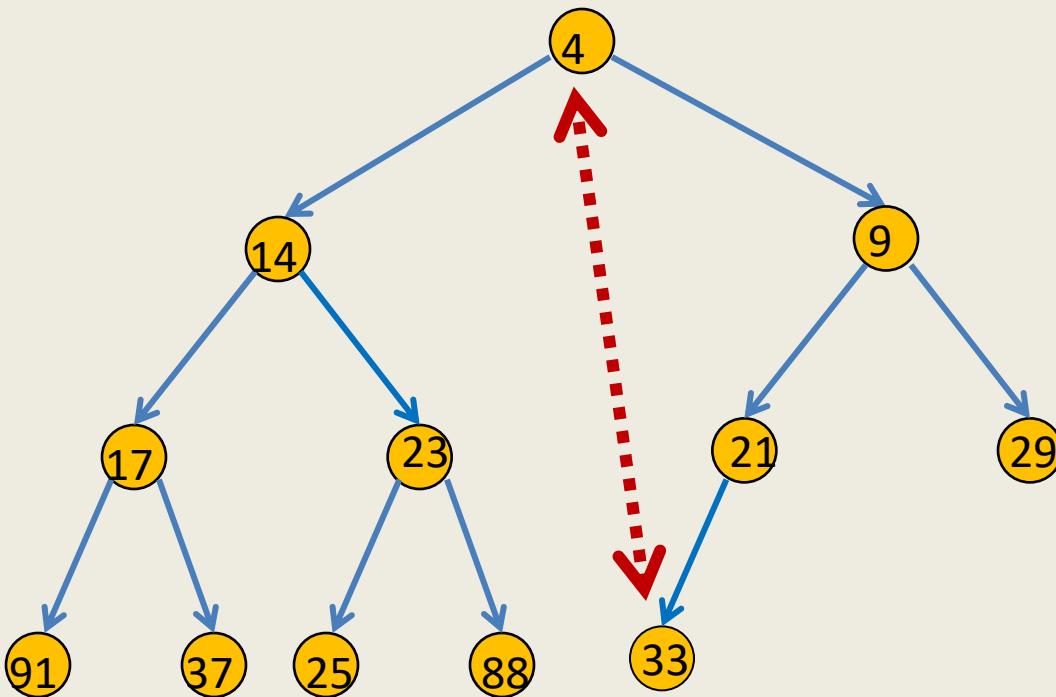
how to preserve the complete binary tree structure as well as the heap property ?



H

4	14	9	17	23	21	29	91	37	25	88	33			
---	----	---	----	----	----	----	----	----	----	----	----	--	--	--

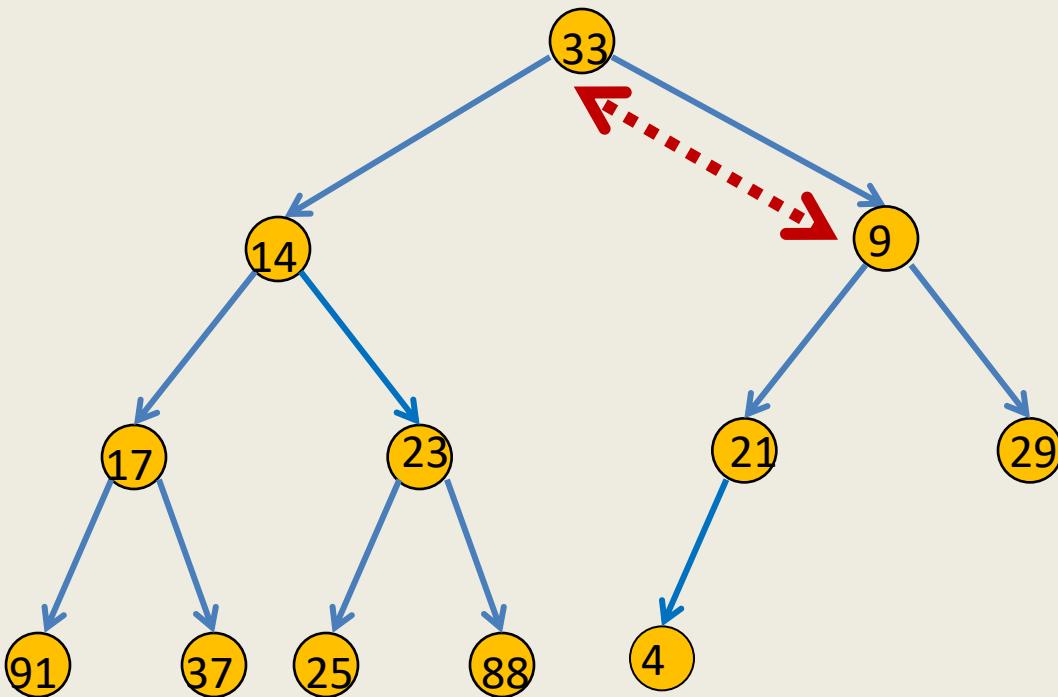
Extract_min(H)



H

4	14	9	17	23	21	29	91	37	25	88	33			
---	----	---	----	----	----	----	----	----	----	----	----	--	--	--

Extract_min(H)

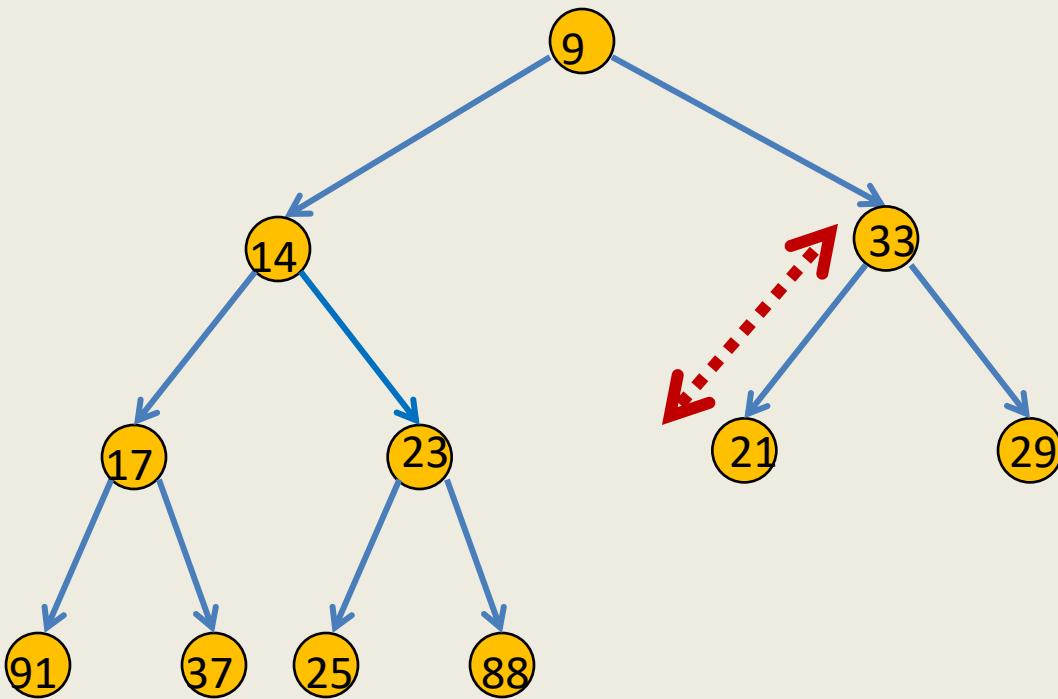


H

33	14	9	17	23	21	29	91	37	25	88	4				
----	----	---	----	----	----	----	----	----	----	----	---	--	--	--	--



Extract_min(H)



H

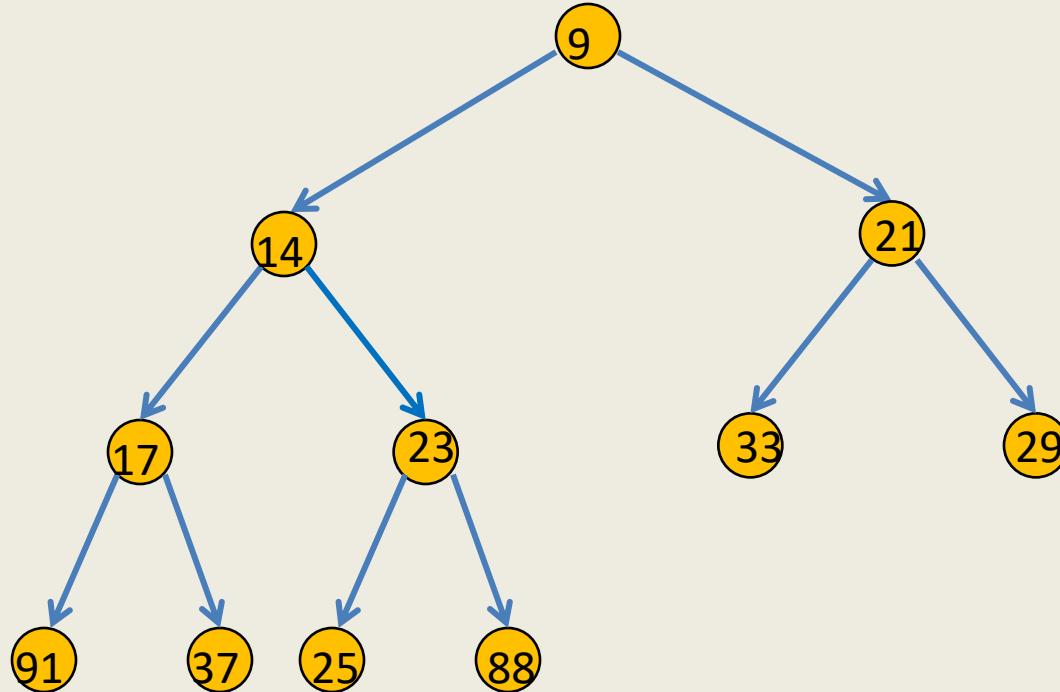
9	14	33	17	23	21	29	91	37	25	88				
---	----	----	----	----	----	----	----	----	----	----	--	--	--	--

Extract_min(H)

We are done.

The no. of operations performed = **O**(no. of levels in binary heap)

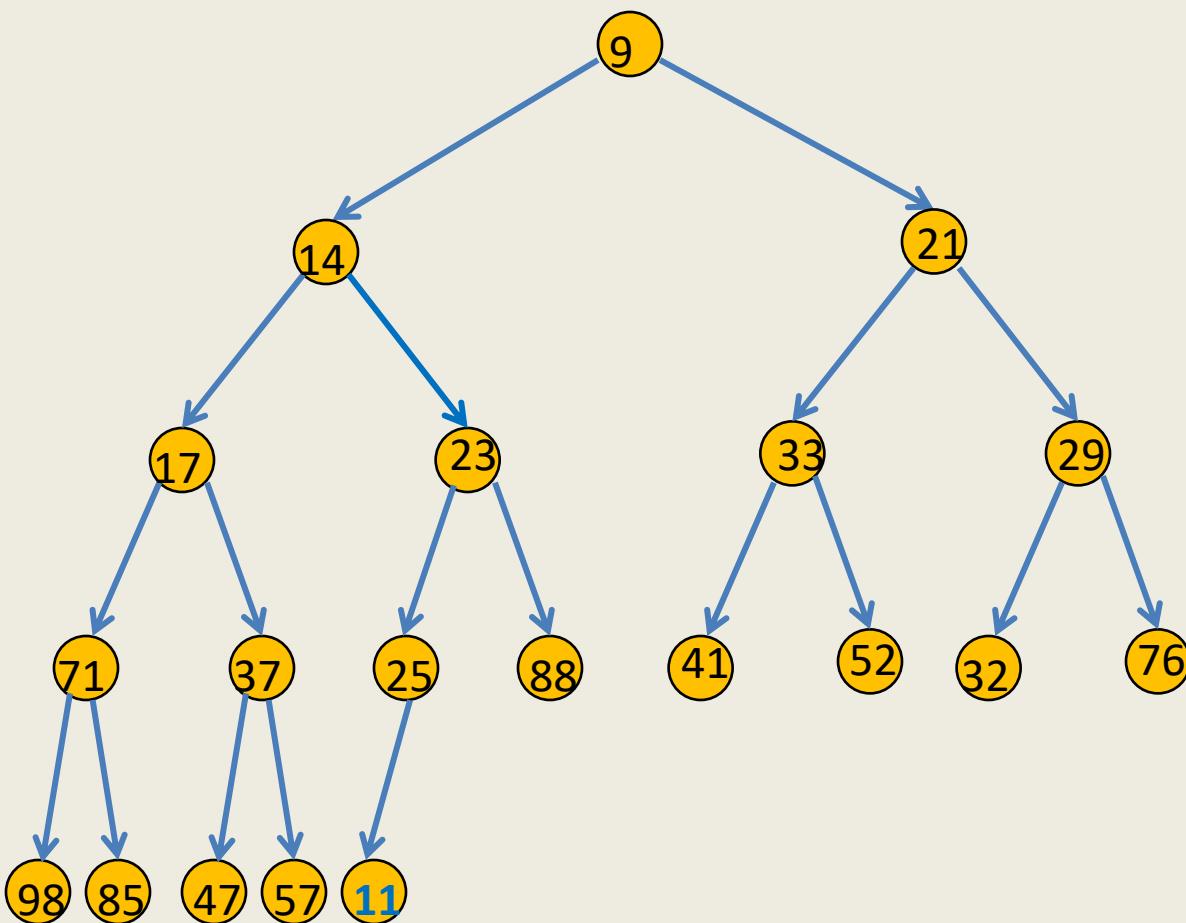
= **O**($\log n$) ...show it as an **homework exercise**.



H

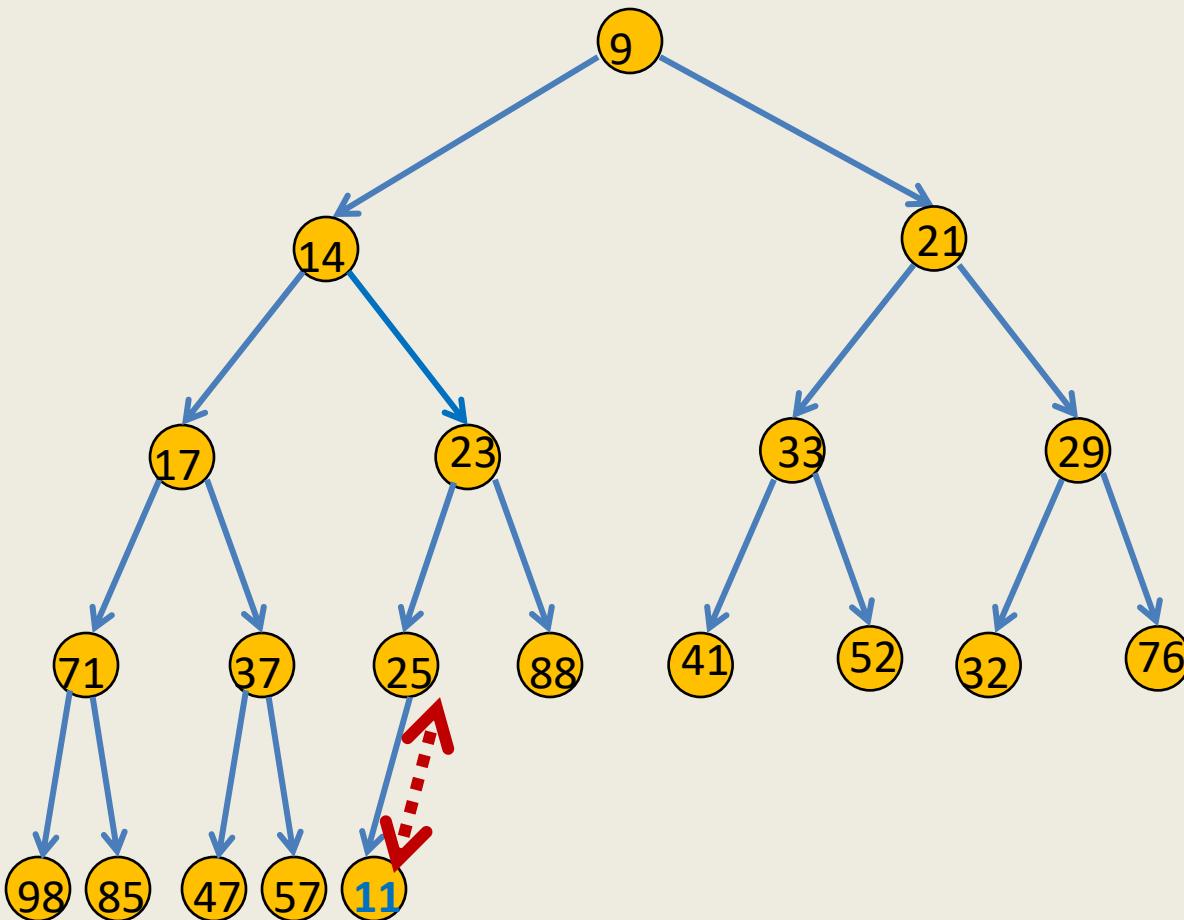
9	14	21	17	23	33	29	91	37	25	88				
---	----	----	----	----	----	----	----	----	----	----	--	--	--	--

Insert(x , H)



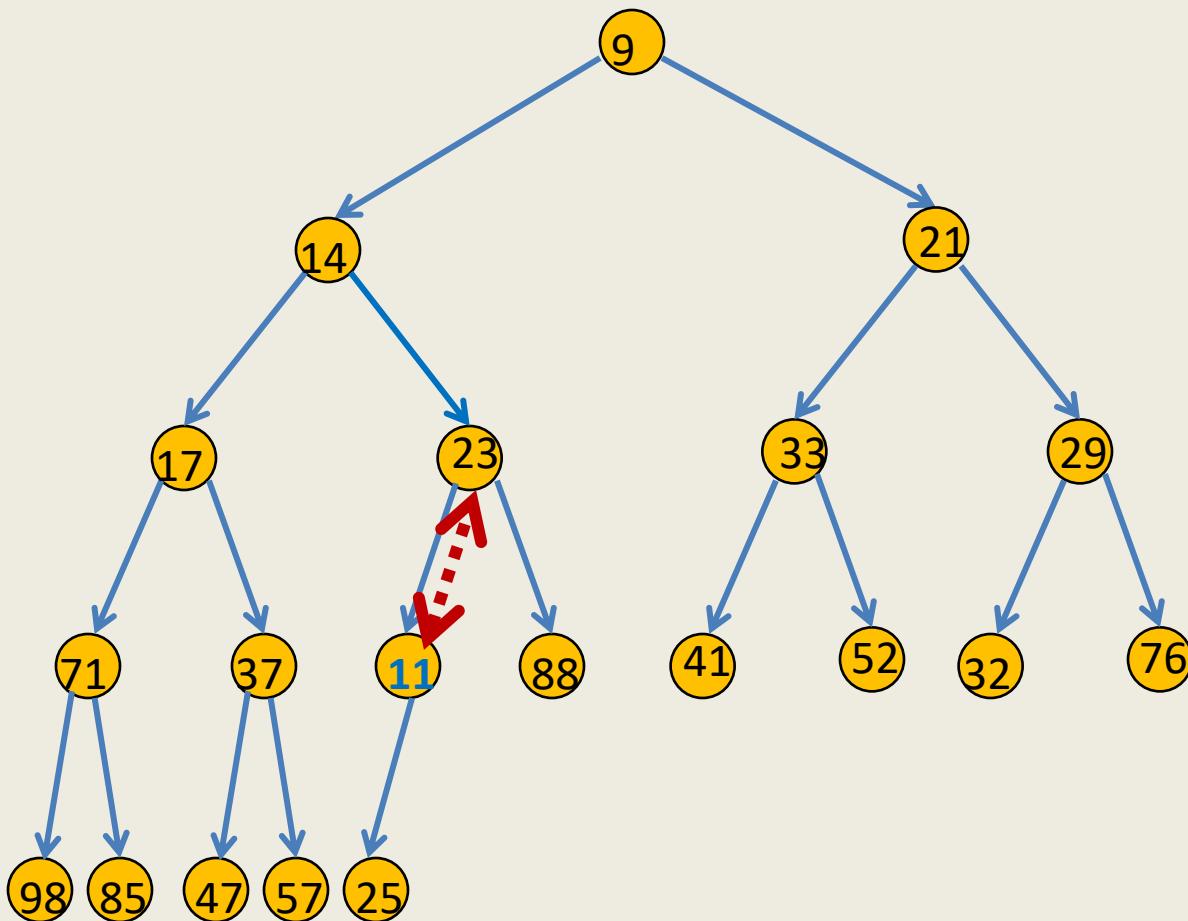
H	9	14	21	17	23	33	29	71	37	25	88	41	52	32	76	98	85	47	57	11
----------	---	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	-----------

Insert(x , H)

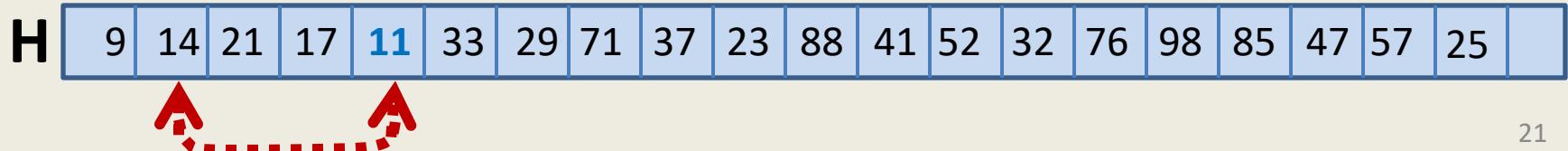
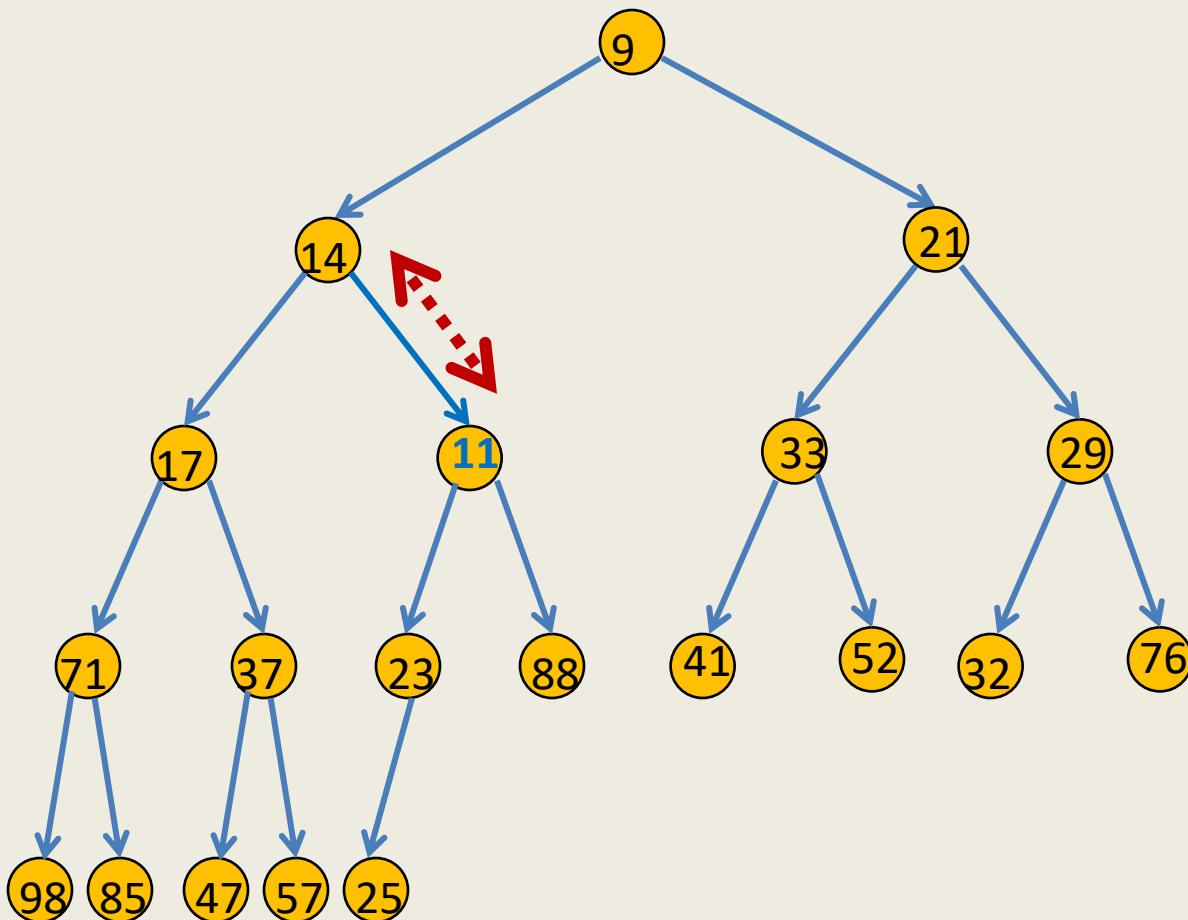


H	9	14	21	17	23	33	29	71	37	25	88	41	52	32	76	98	85	47	57	11
----------	---	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----

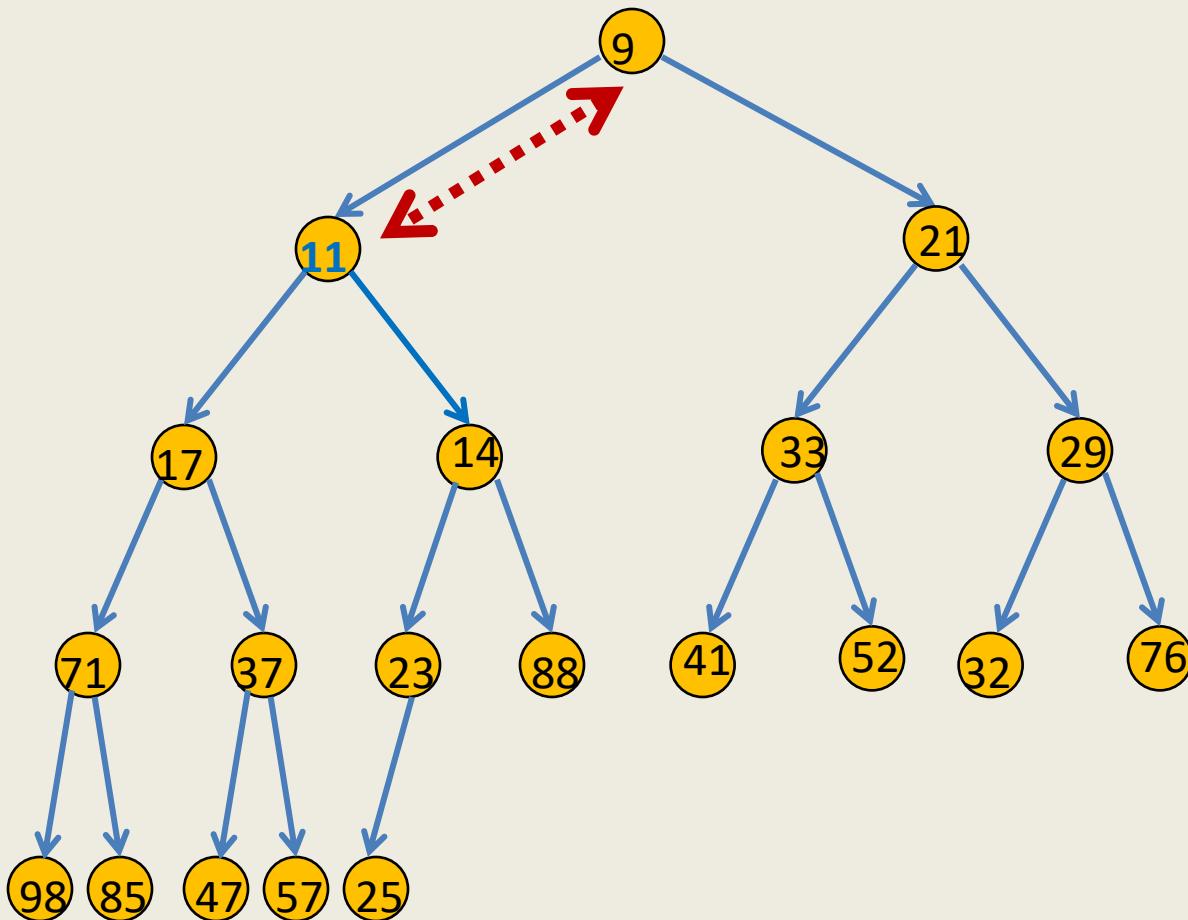
Insert(x , H)



Insert(x, H)



Insert(x, H)



Insert(x,H)

Insert(x,H)

```
{   i < size(H);  
    H(size) < x;  
    size(H) < size(H) + 1;  
    While(           i > 0           and  H(i) < H([(i - 1)/2]) )  
    {  
        H(i) ↔ H([(i - 1)/2]);  
        i <- [(i - 1)/2];  
    }  
}
```

Time complexity: $O(\log n)$

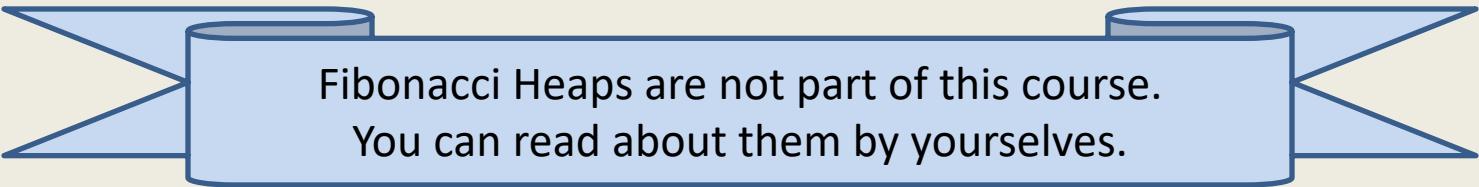
The remaining operations on Binary heap

- **Decrease-key(p, Δ, H):** decrease the value of the key p by amount Δ .
- Similar to **Insert(x, H)**.
- **$O(\log n)$ time**
- Do it as an exercise
- **Merge(H_1, H_2):** Merge two heaps H_1 and H_2 .
- **$O(n)$ time** where n = total number of elements in H_1 and H_2
(This is because of the array implementation)

Other heaps

Fibonacci heap : a link based data structure.

	Binary heap	Fibonacci heap
Find-min(H)	$O(1)$	$O(1)$
Insert(x, H)	$O(\log n)$	$O(1)$
Extract-min(H)	$O(\log n)$	$O(\log n)$
Decrease-key(p, Δ, H)	$O(\log n)$	$O(1)$
Merge(H_1, H_2)	$O(n)$	$O(1)$



Fibonacci Heaps are not part of this course.
You can read about them by yourselves.

Building a Binary heap

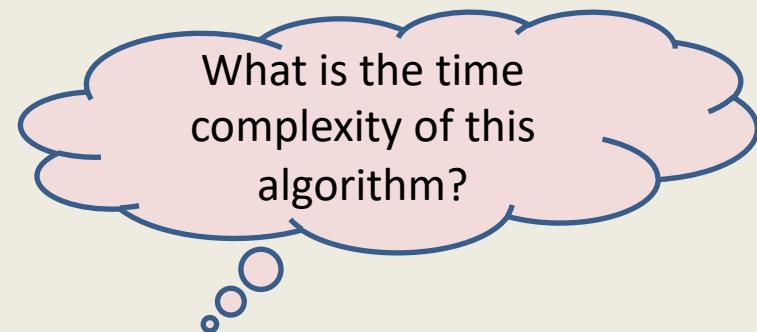
Building a Binary heap

Problem: Given n elements $\{x_0, \dots, x_{n-1}\}$, build a binary **heap H** storing them.

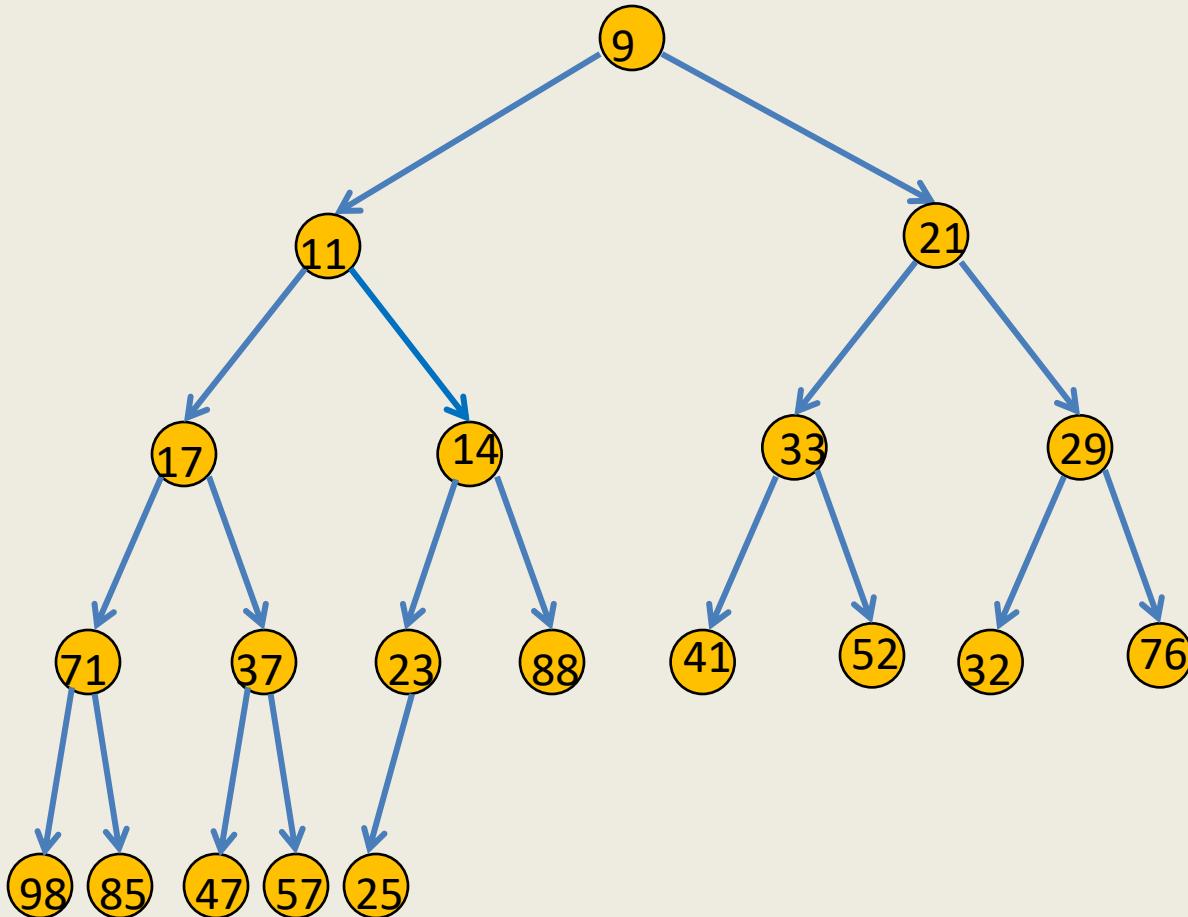
Trivial solution:

(Building the Binary heap **incrementally**)

```
CreateHeap( $H$ );  
For(  $i = 0$  to  $n - 1$  )  
    Insert( $x_i, H$ );
```



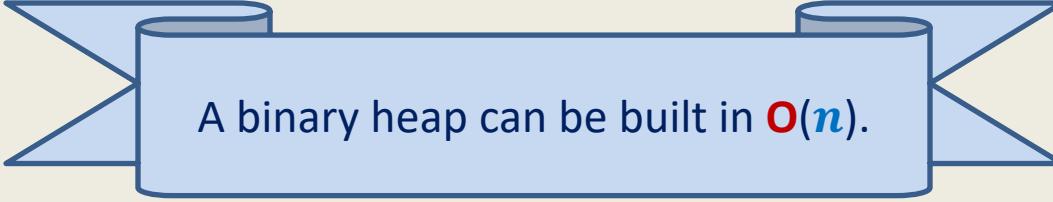
Building a Binary heap incrementally



H	9	11	21	17	14	33	29	71	37	23	88	41	52	32	76	98	85	47	57	25
---	---	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----

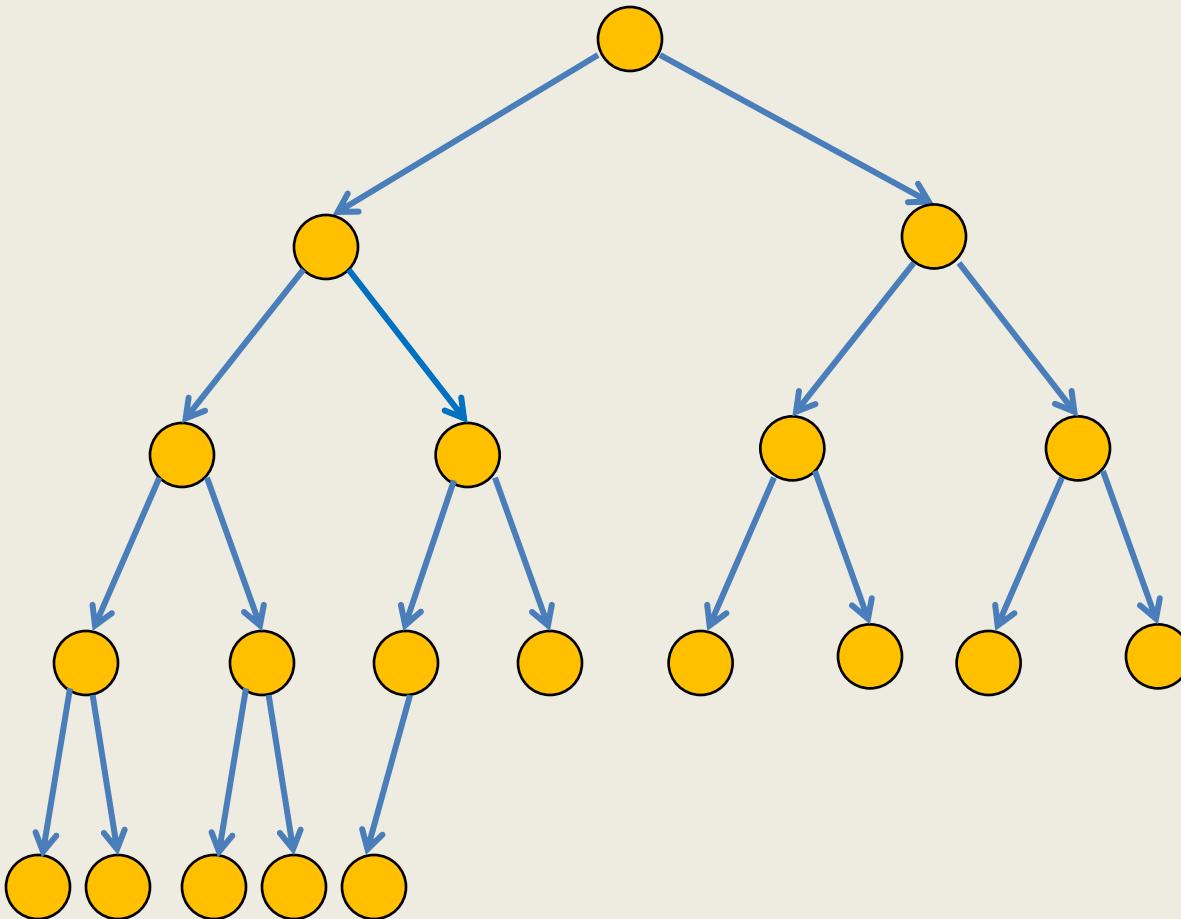
Time complexity

Theorem: Time complexity of building a binary heap **incrementally** is $O(n \log n)$.



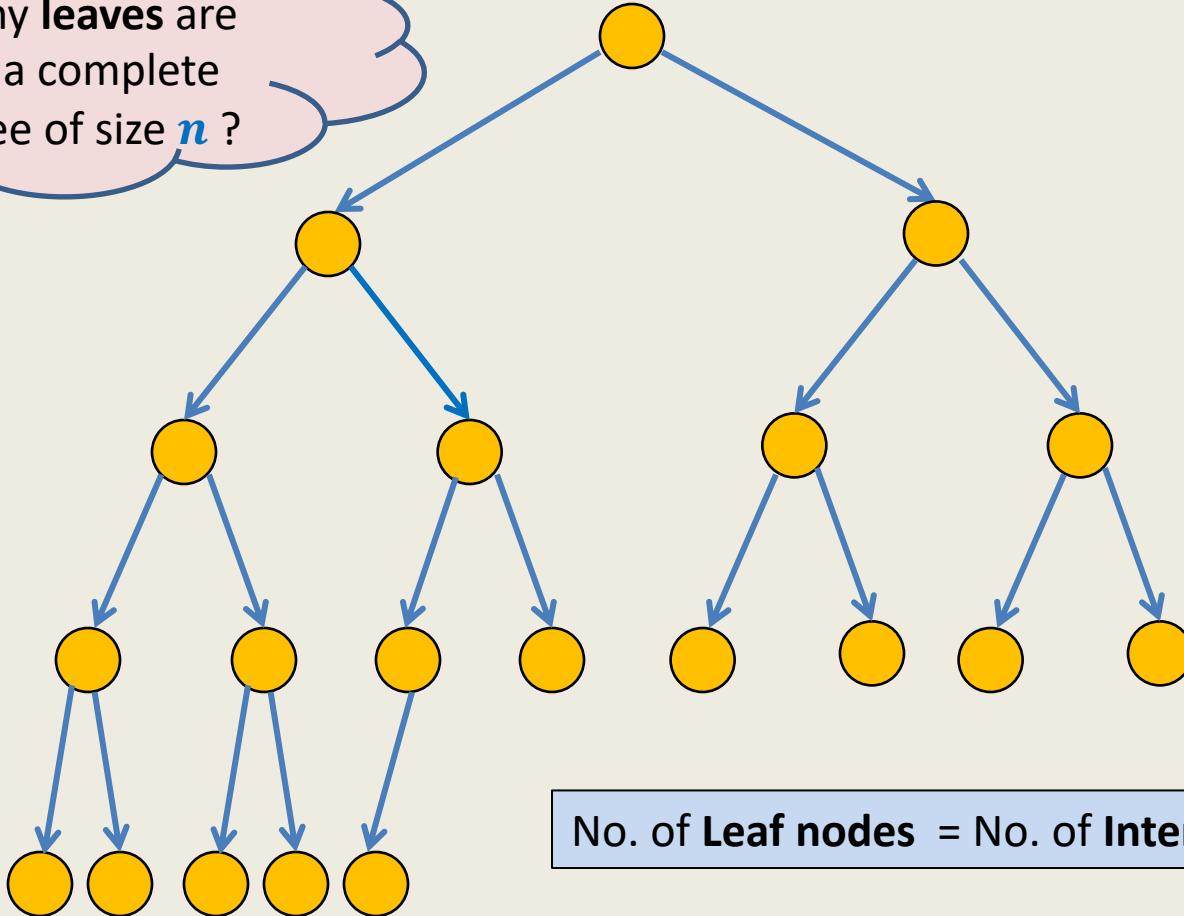
A binary heap can be built in $O(n)$.

A complete binary tree



A complete binary tree

How many **leaves** are there in a complete Binary tree of size n ?

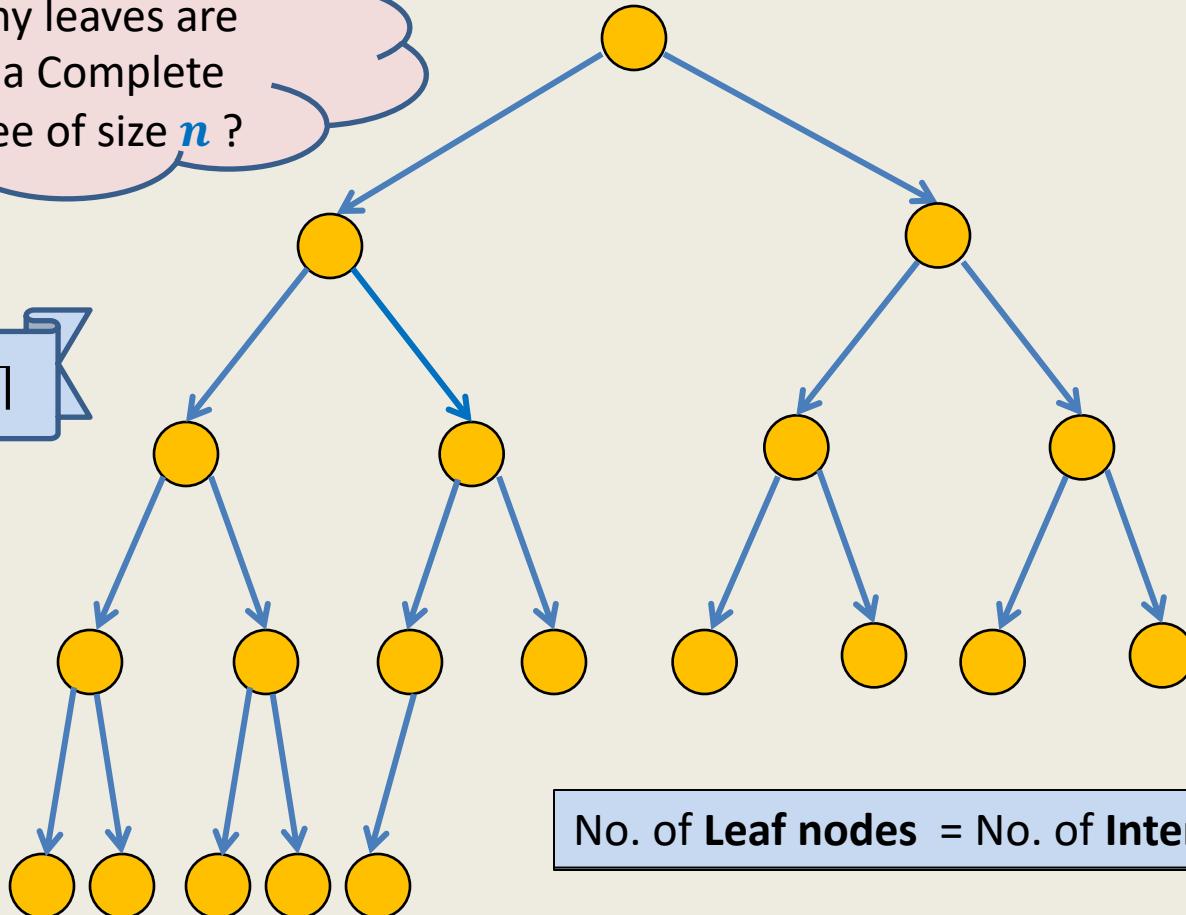


No. of Leaf nodes = No. of Internal nodes + 1

A complete binary tree

How many leaves are there in a Complete Binary tree of size n ?

$[n/2]$

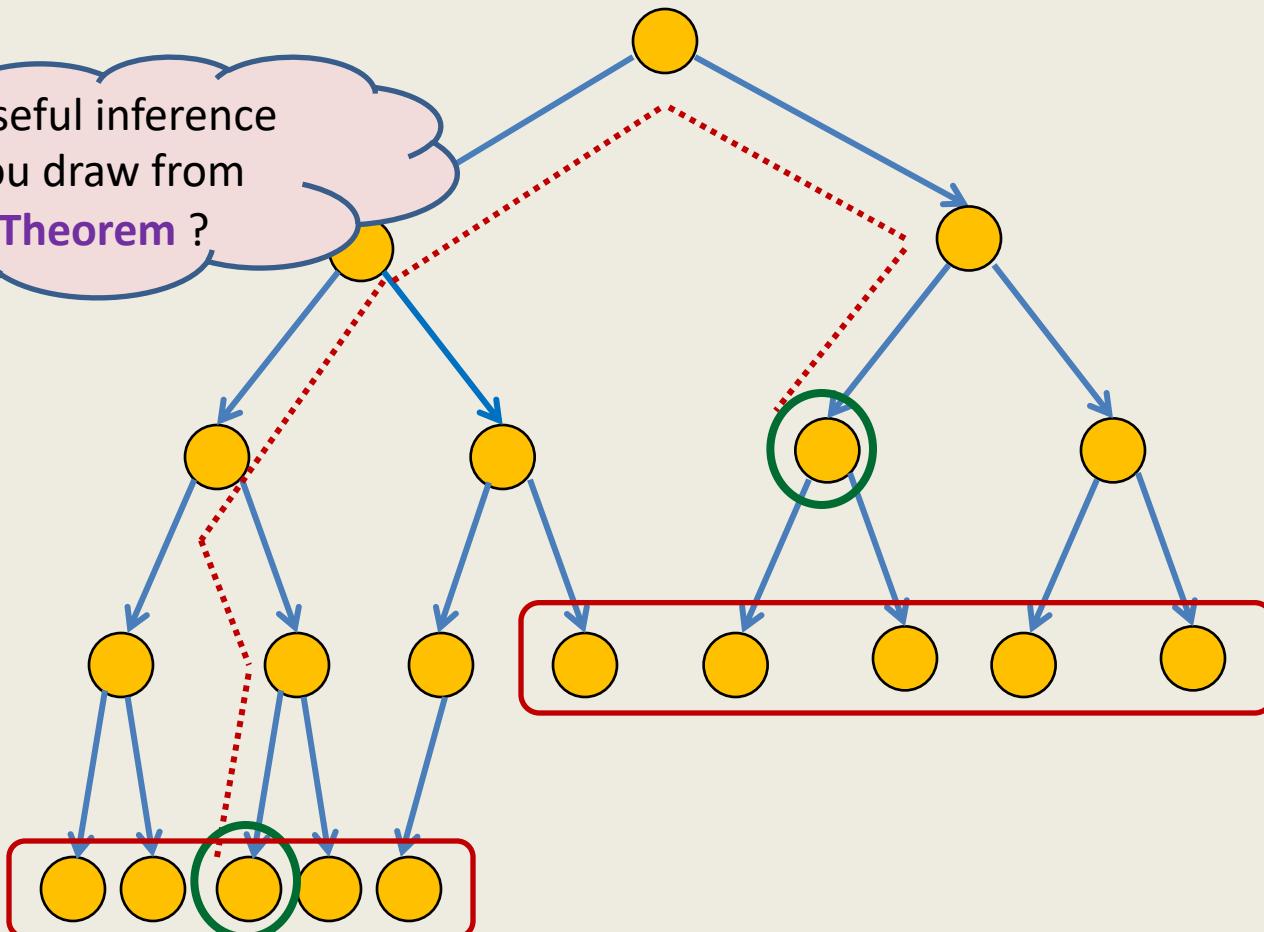


No. of Leaf nodes = No. of Internal nodes

1

Building a Binary heap incrementally

What useful inference
can you draw from
this **Theorem** ?



Top-down
approach

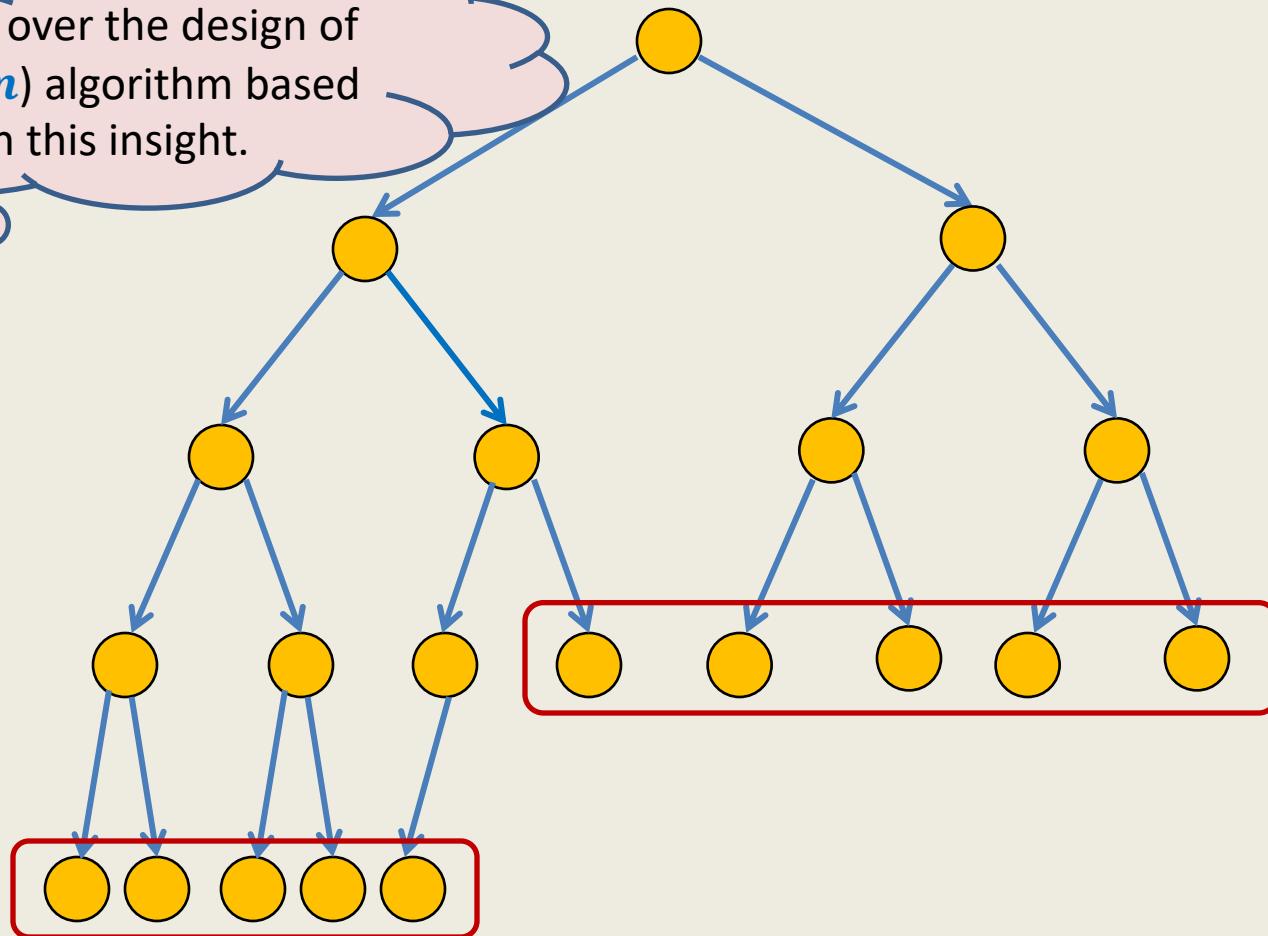
The time complexity for inserting a leaf node = $O(\log n)$

leaf nodes = $\lceil n/2 \rceil$,

→ **Theorem:** Time complexity of building a binary heap incrementally is $O(n \log n)$.

Building a Binary heap incrementally

Ponder over the design of the $O(n)$ algorithm based on this insight.



Top-down approach

The $O(n)$ time algorithm must take $O(1)$ time for each of the $[n/2]$ leaves.

Data Structures and Algorithms

(ESO207)

Lecture 29:

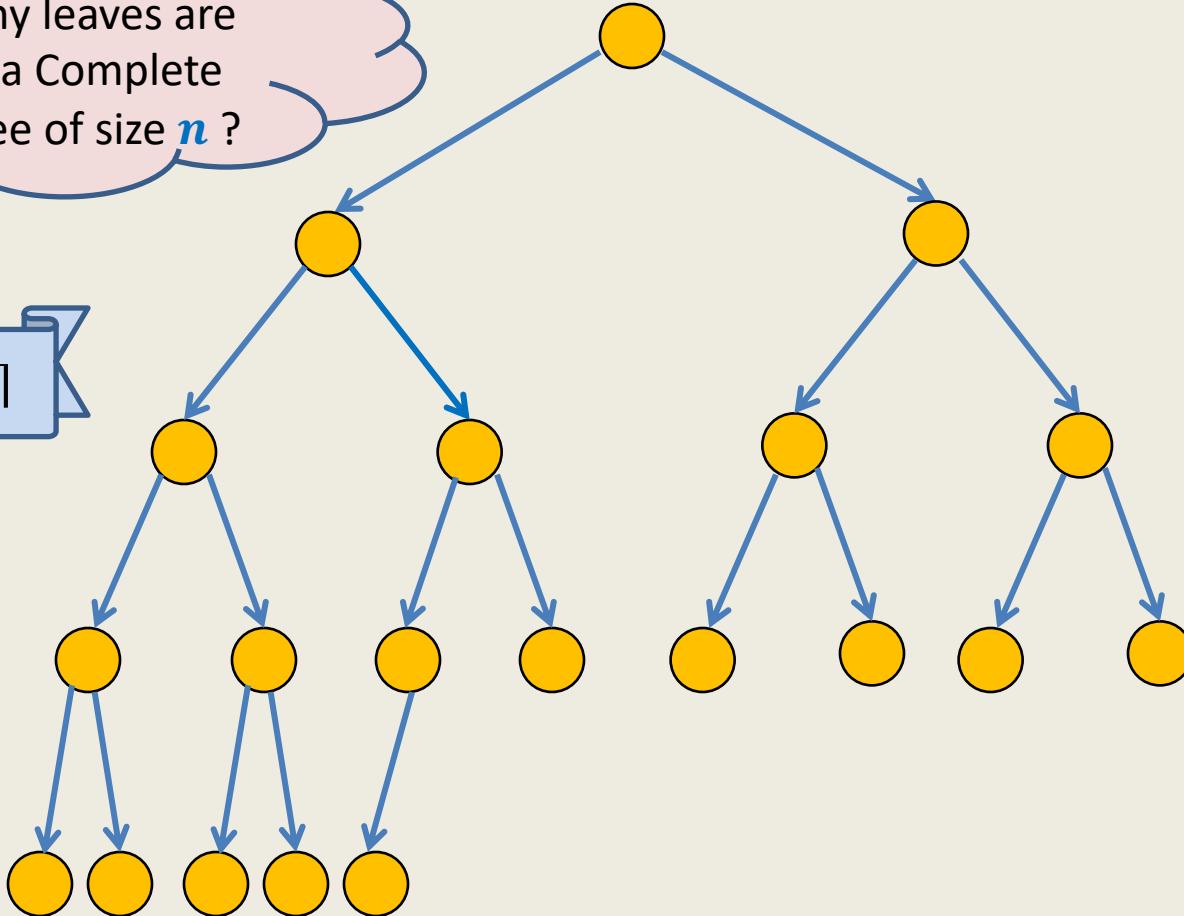
- Building a Binary heap on n elements in $O(n)$ time.
- Applications of Binary heap : sorting
- Binary trees: beyond searching and sorting

Recap from the last lecture

A complete binary tree

How many leaves are there in a Complete Binary tree of size n ?

$[n/2]$



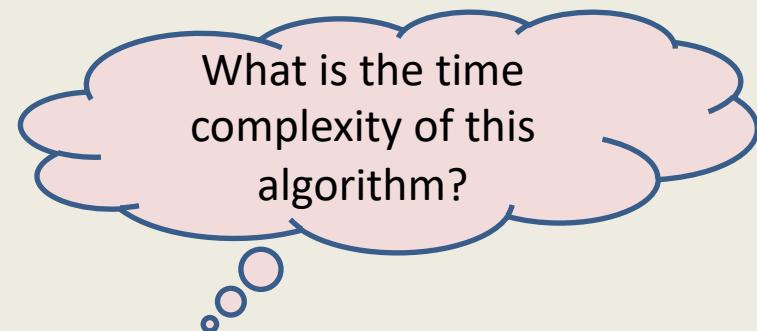
Building a Binary heap

Problem: Given n elements $\{x_0, \dots, x_{n-1}\}$, build a binary heap H storing them.

Trivial solution:

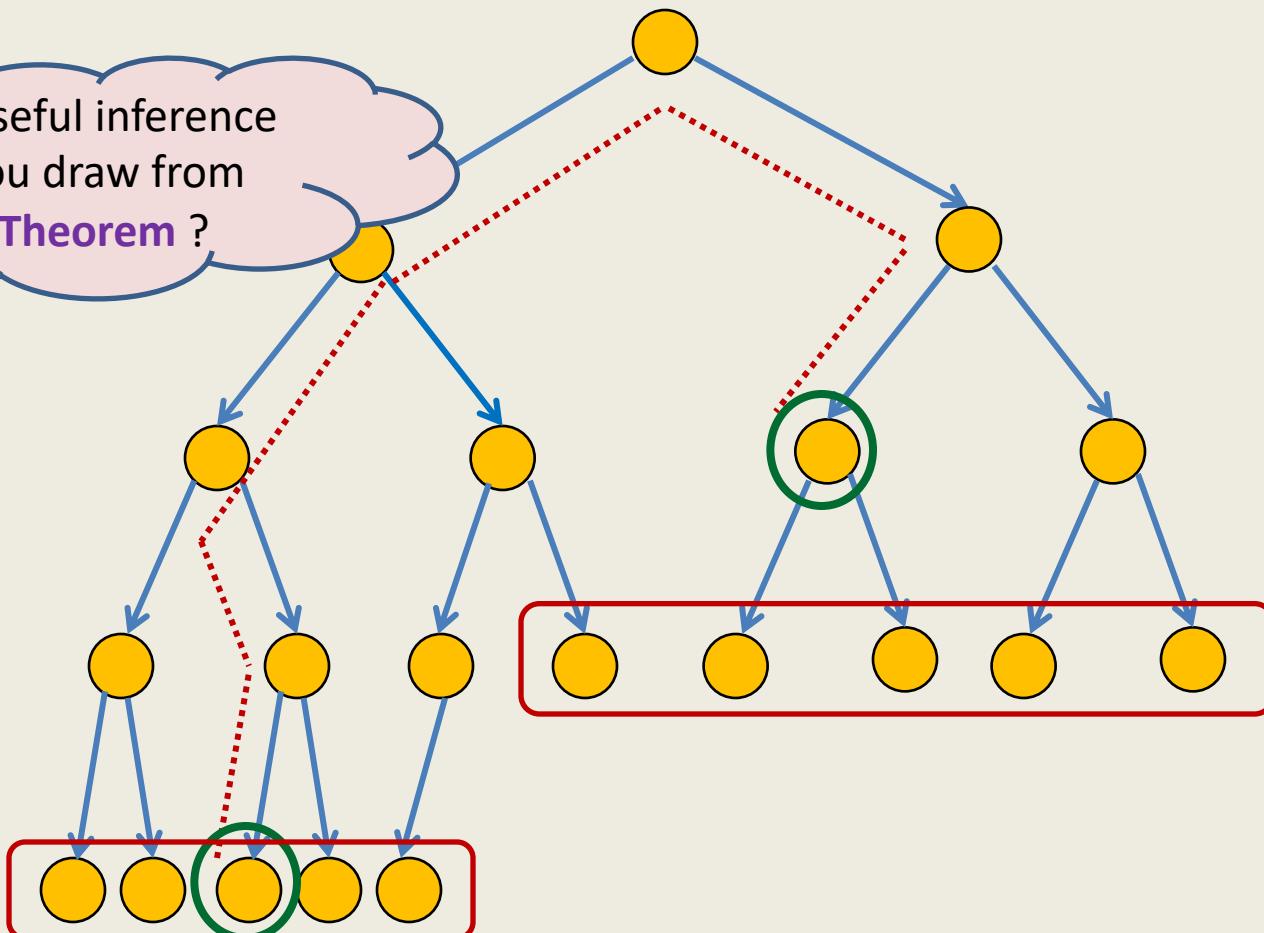
(Building the Binary heap **incrementally**)

```
CreateHeap(H);  
For(  $i = 0$  to  $n - 1$  )  
    Insert( $x_i, H$ );
```



Building a Binary heap incrementally

What useful inference
can you draw from
this **Theorem** ?



Top-down
approach



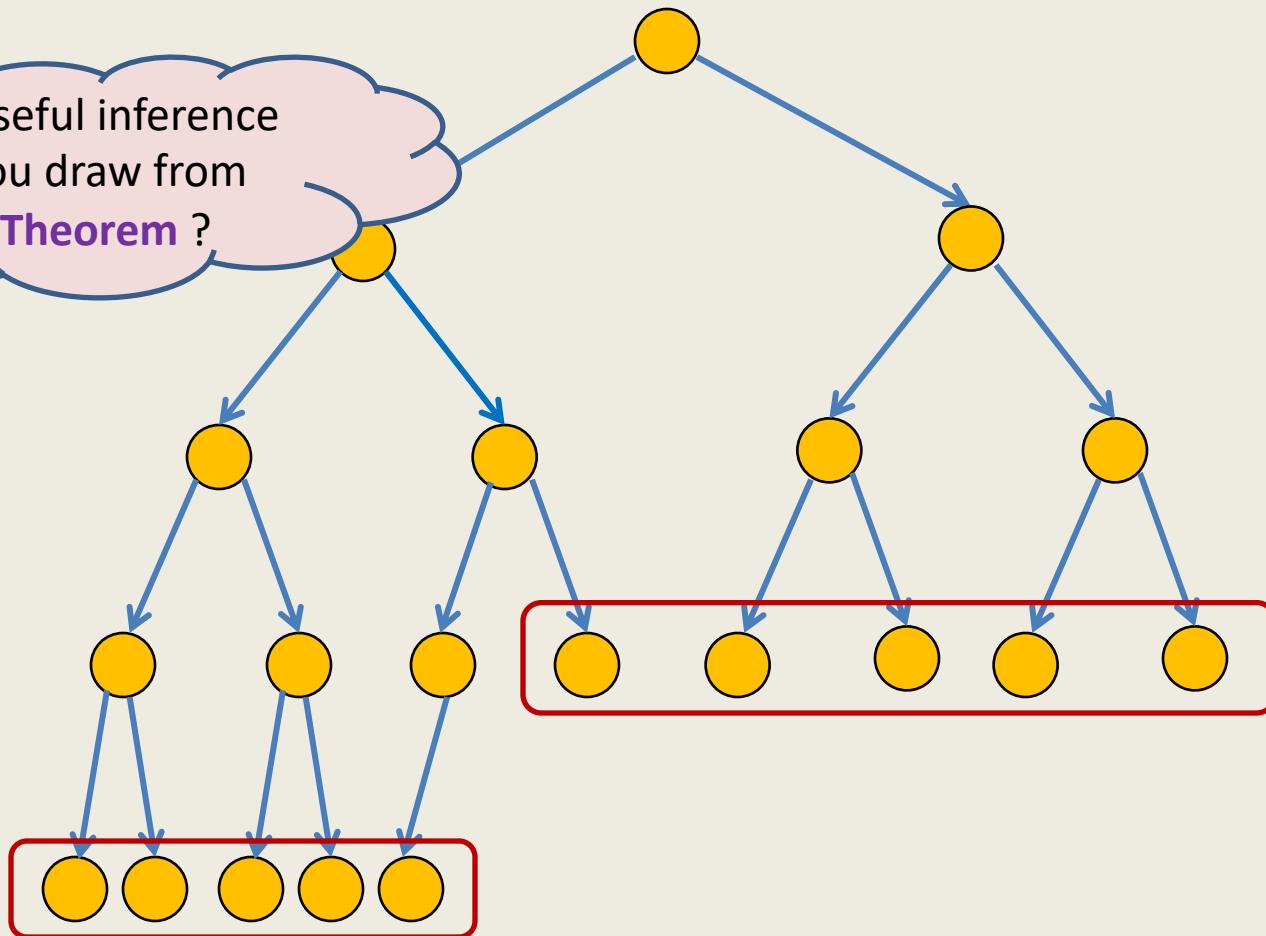
The time complexity for inserting a leaf node = $O(\log n)$

leaf nodes = $\lceil n/2 \rceil$,

→ **Theorem:** Time complexity of building a binary heap incrementally is $O(n \log n)$.

Building a Binary heap incrementally

What useful inference
can you draw from
this **Theorem** ?

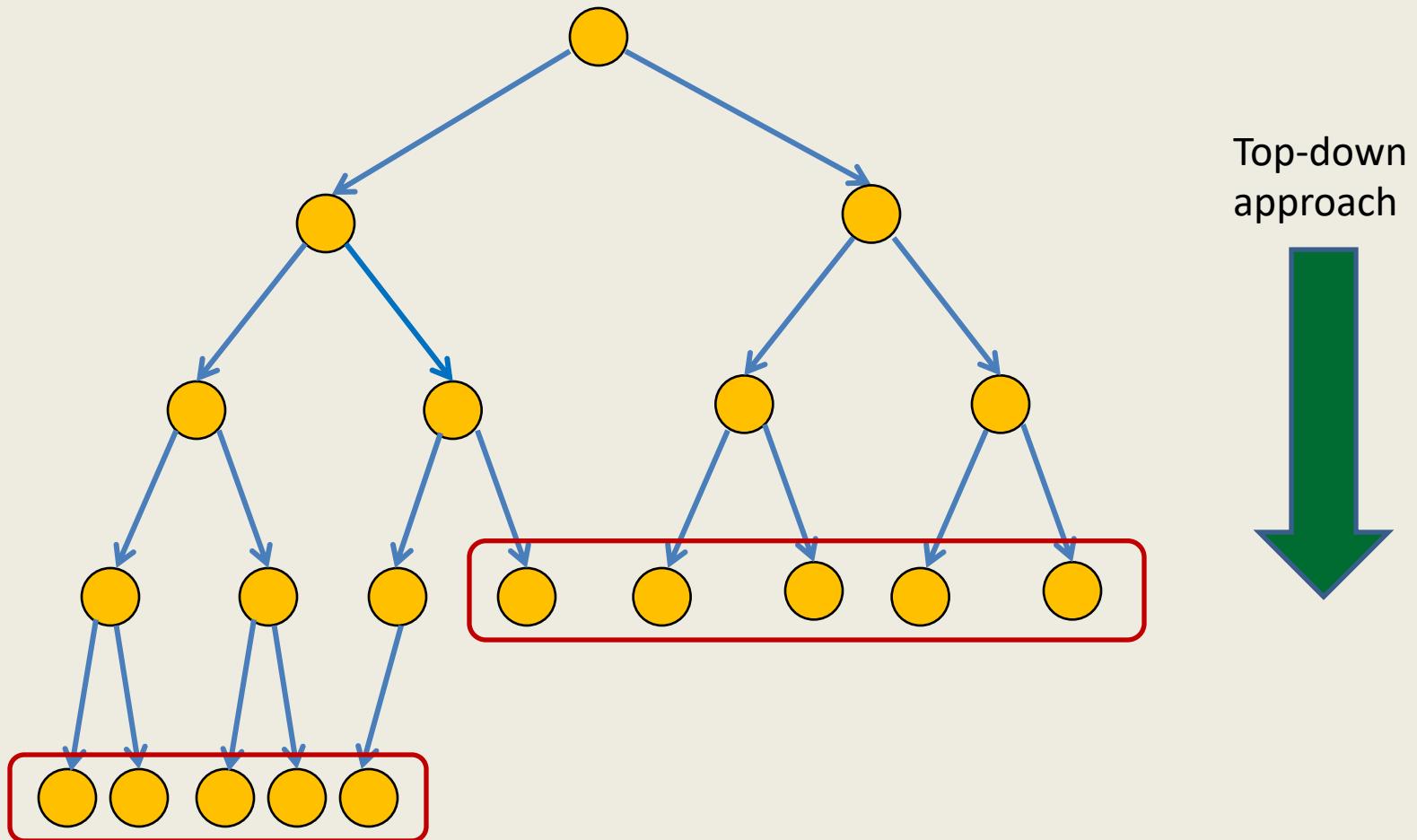


Top-down
approach

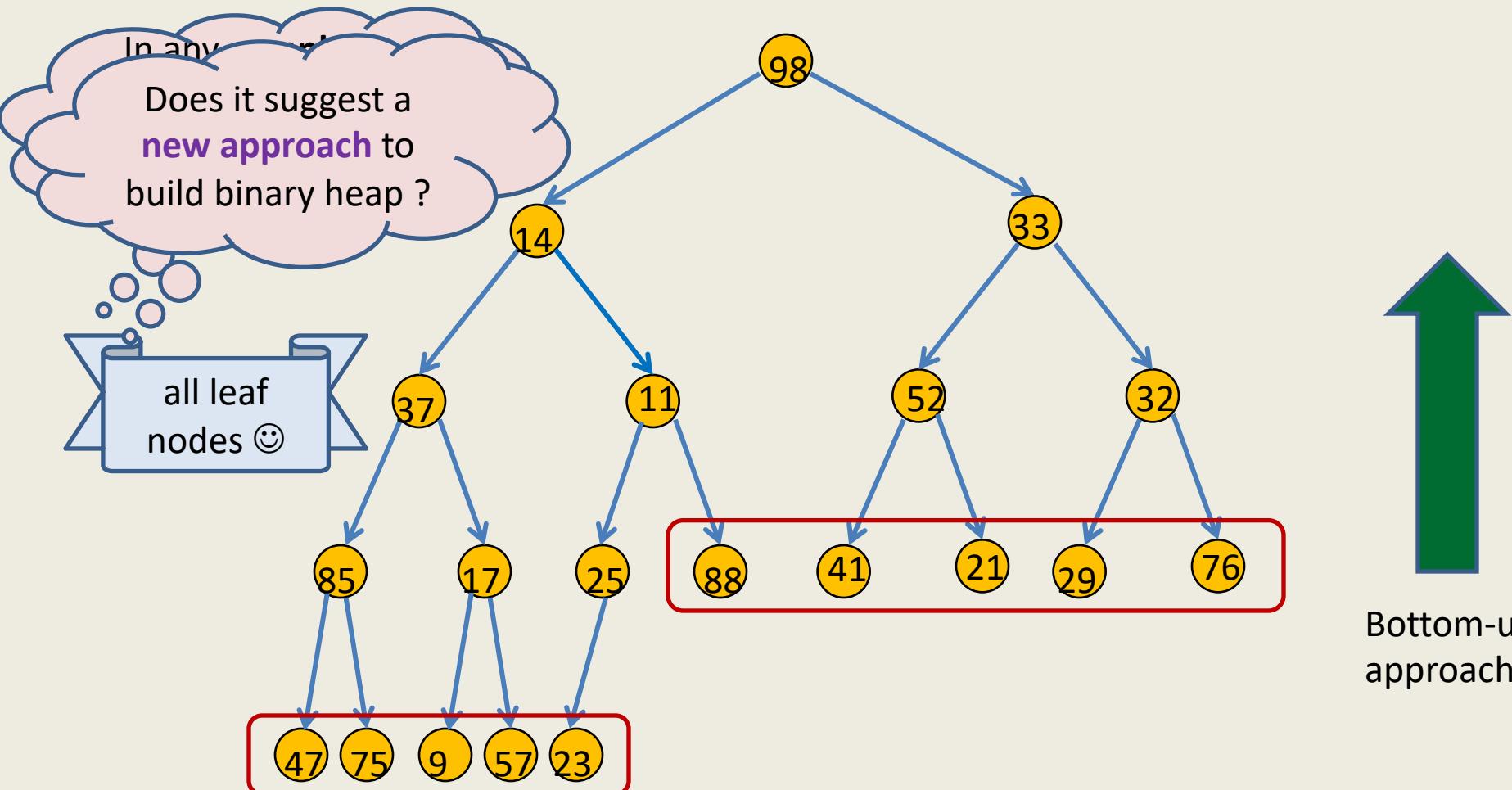


The **O(n)** time algorithm must take **O(1)** time
for each of the **[n/2]** leaves.

Building a Binary heap incrementally



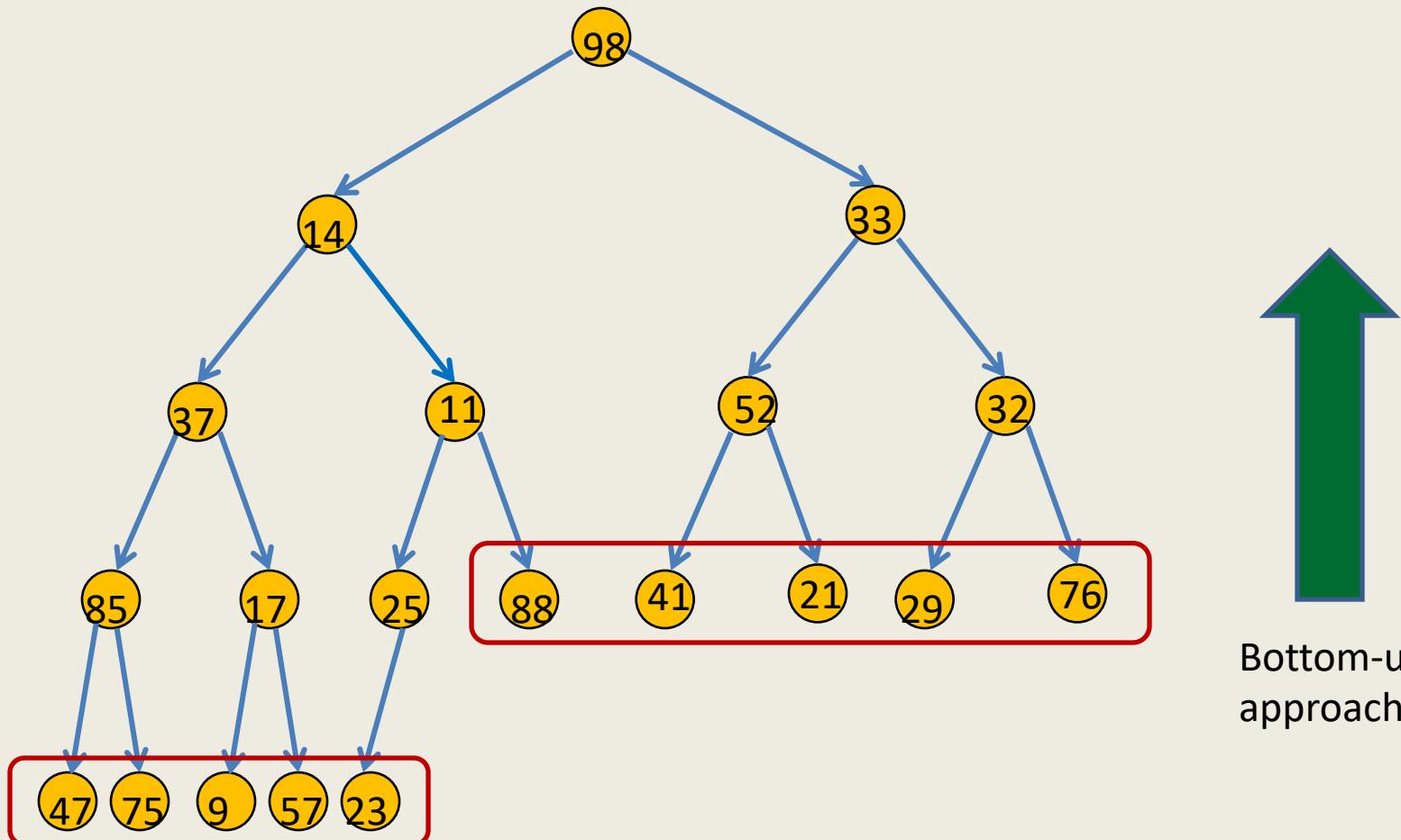
Think of alternate approach for building a binary heap



heap property: “Every **node** stores value smaller than its **children**”

We just need to ensure this property at each node.

Think of alternate approach for building a binary heap



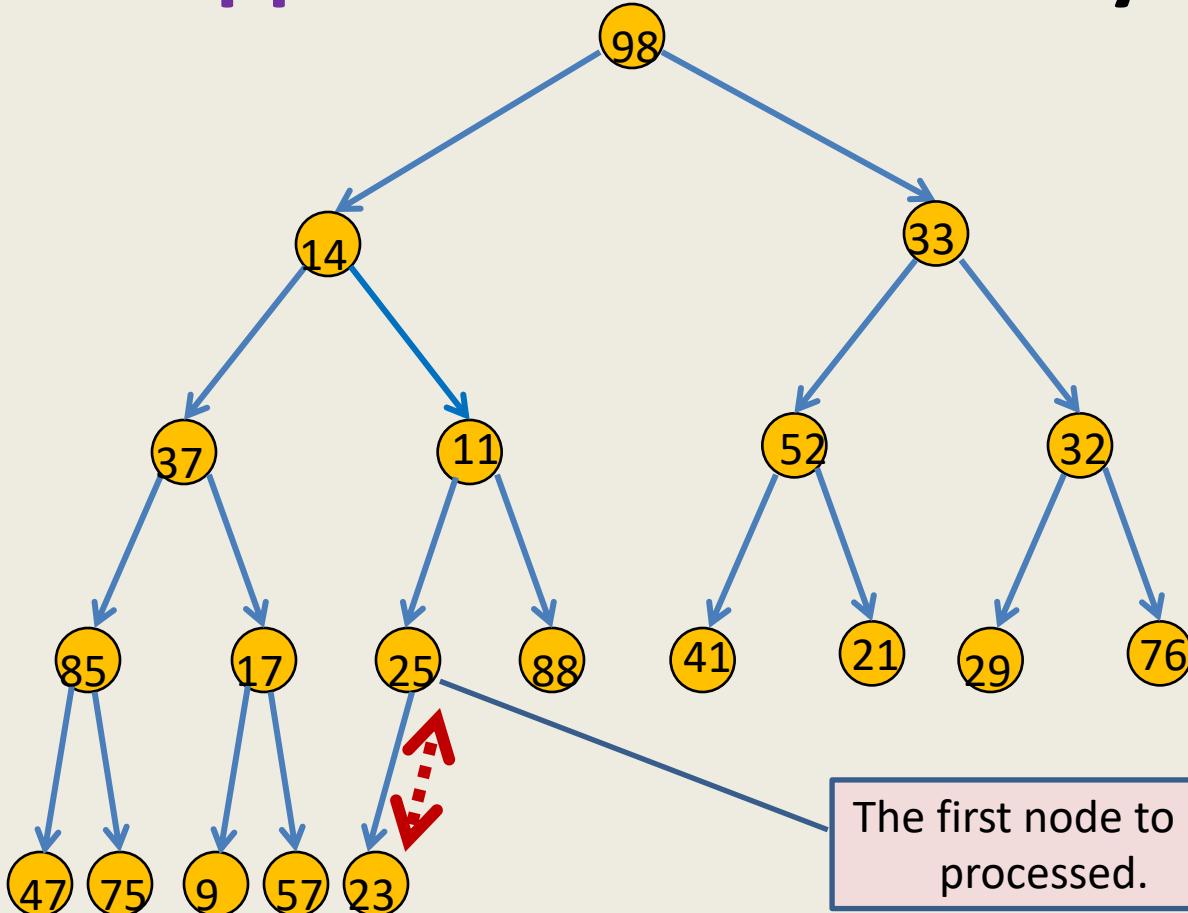
heap property: “Every **node** stores value smaller than its **children**”

We just need to ensure this property at each node.

A new approach to build binary heap

1. Just copy the given n elements $\{x_0, \dots, x_{n-1}\}$ into an array H .
2. The **heap property** holds for all the leaf nodes in the corresponding complete binary tree.
3. Leaving all the leaf nodes,
process the elements in the decreasing order of their numbering
and set the heap property for each of them.

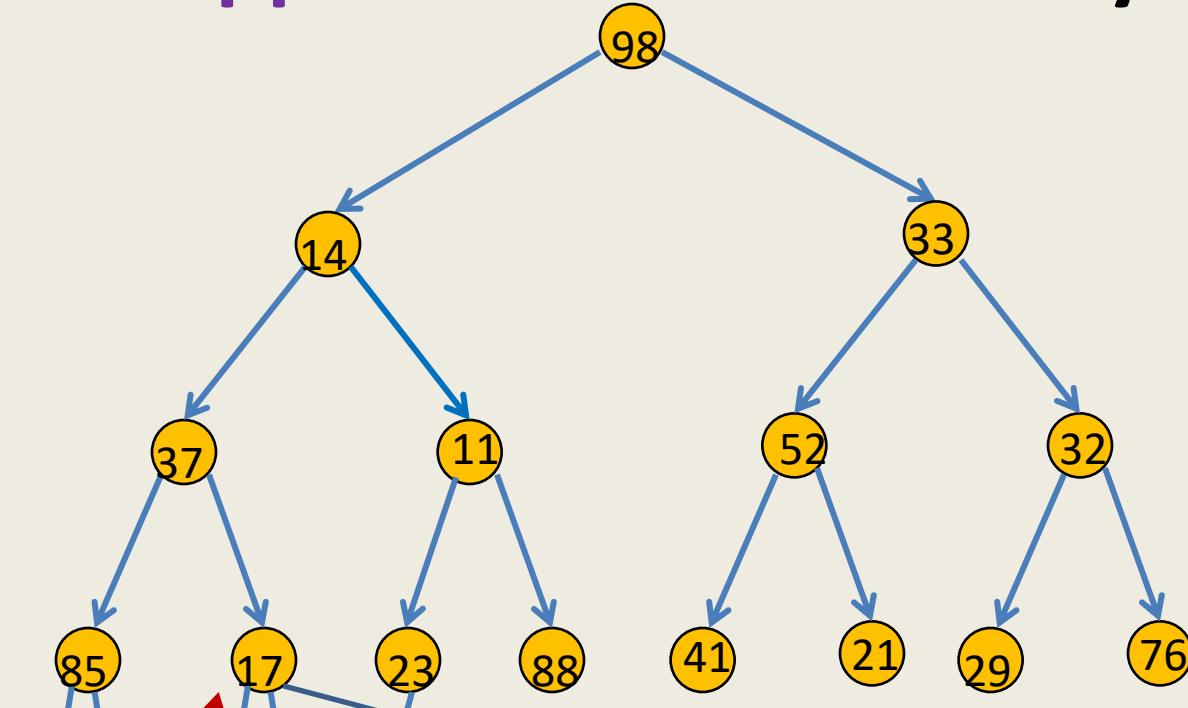
A new approach to build binary heap



H	98	14	33	37	11	52	32	85	17	25	88	41	21	29	76	47	75	9	57	23
---	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	---	----	----



A new approach to build binary heap

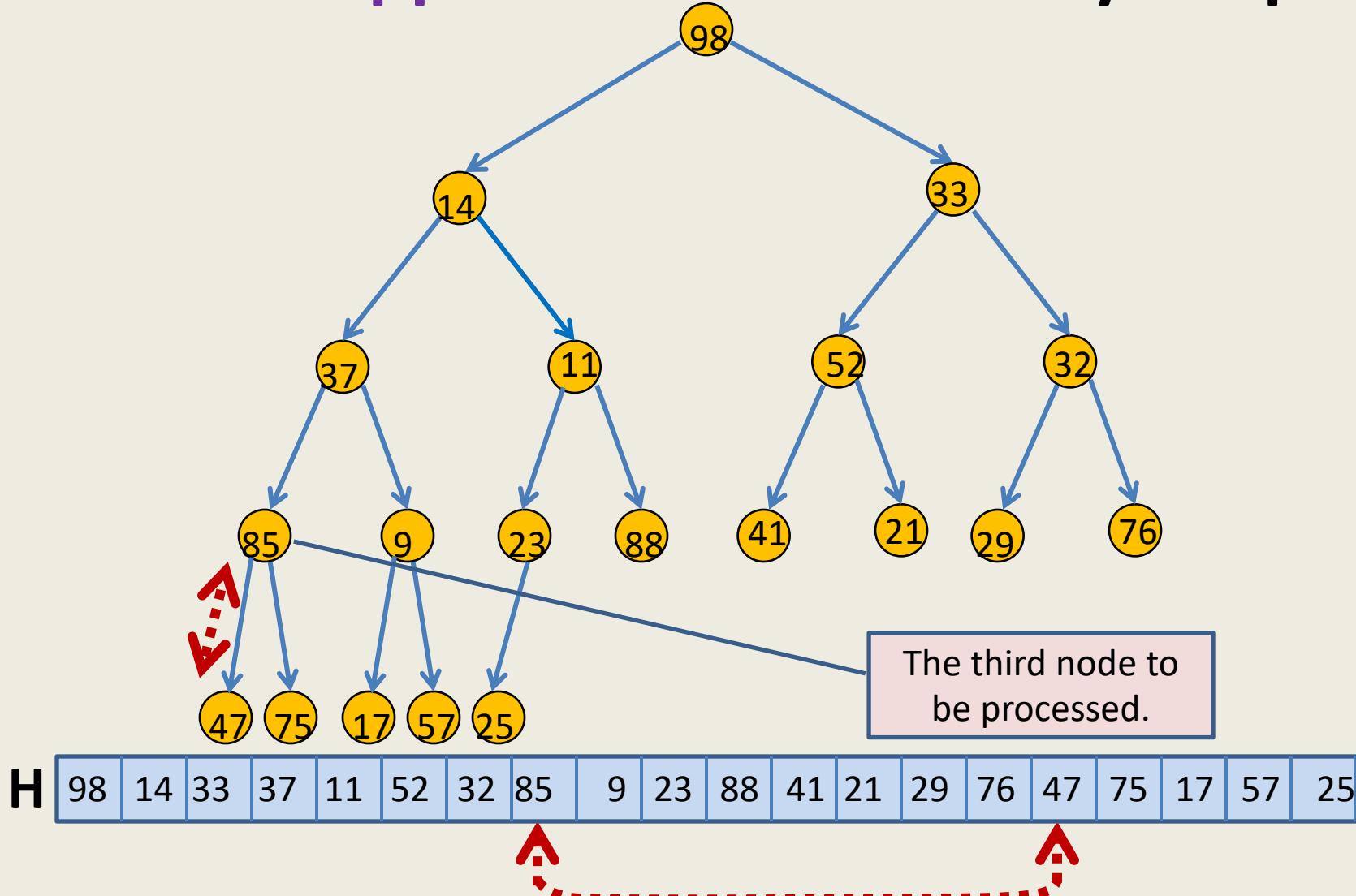


The second node to
be processed.

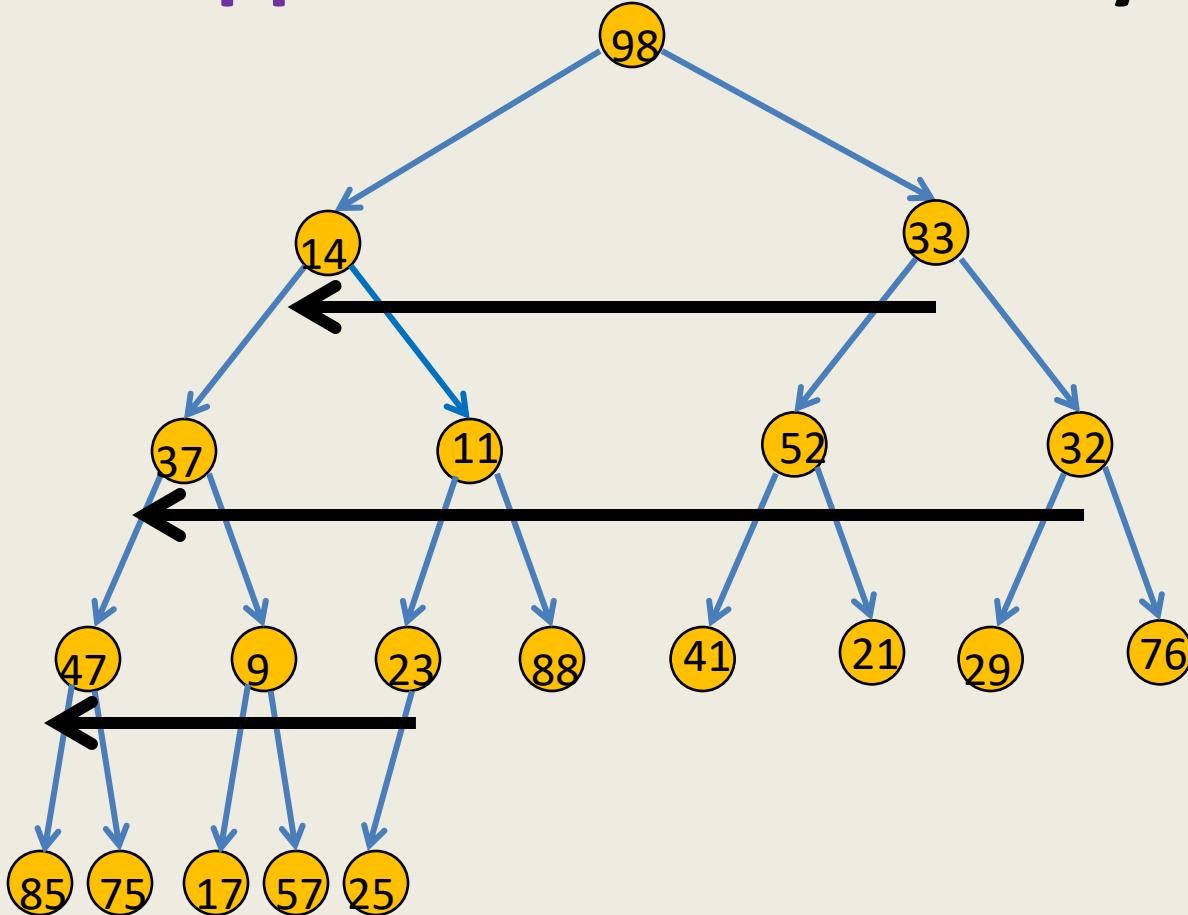
H	98	14	33	37	11	52	32	85	17	23	88	41	21	29	76	47	75	9	57	25
---	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	---	----	----



A new approach to build binary heap

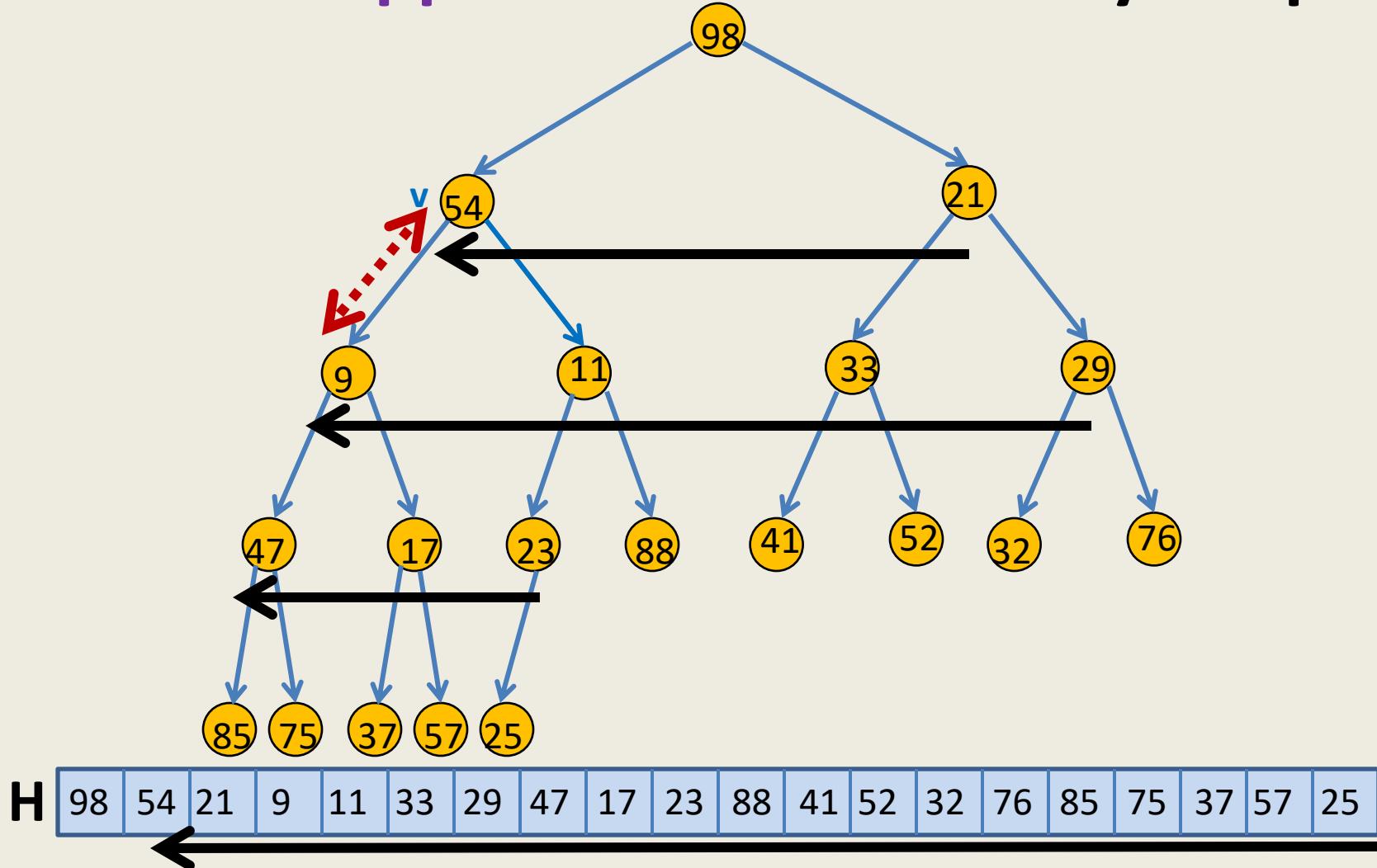


A new approach to build binary heap

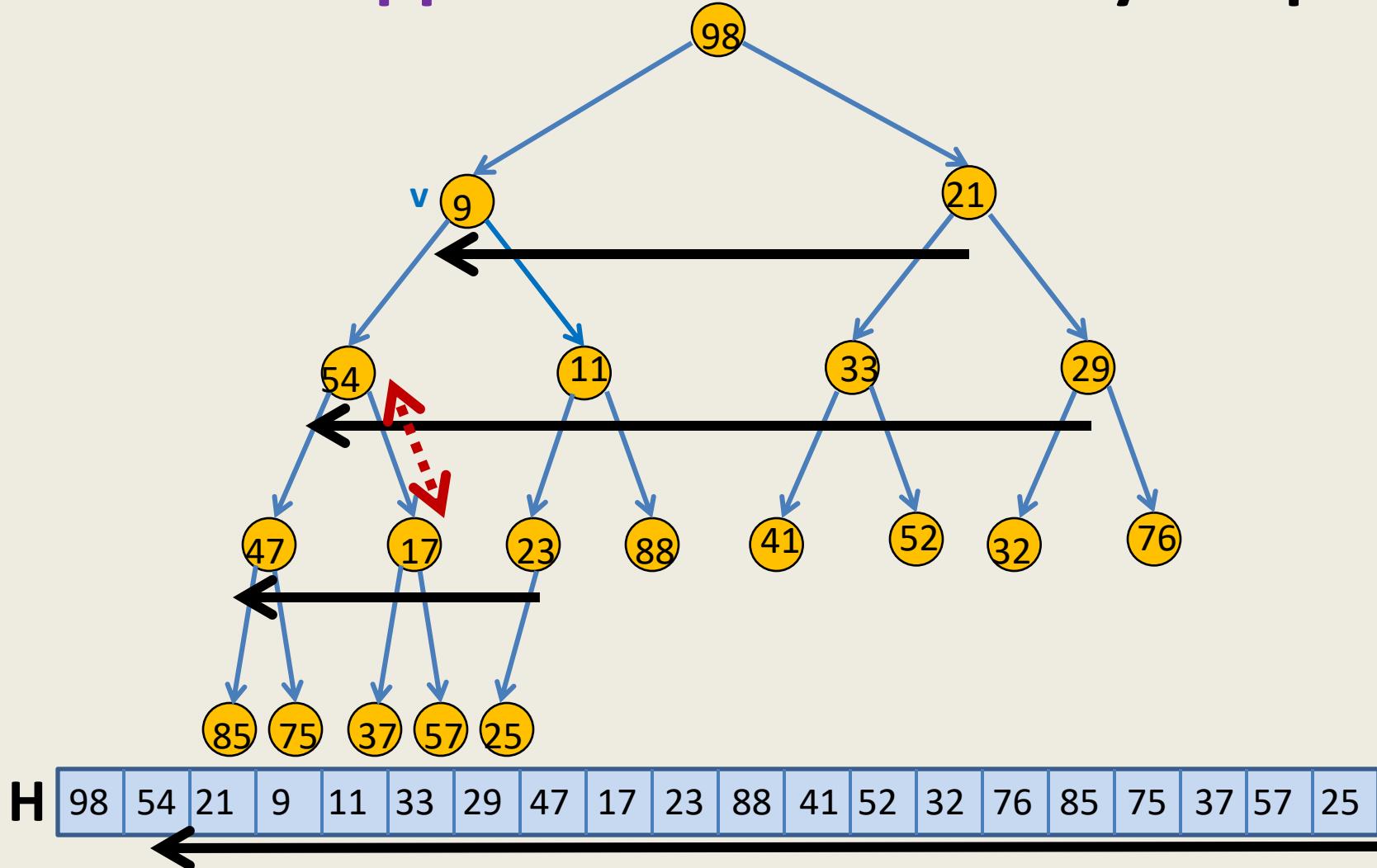


H	98	14	33	37	11	52	32	47	9	23	88	41	21	29	76	85	75	17	57	25
---	----	----	----	----	----	----	----	----	---	----	----	----	----	----	----	----	----	----	----	----

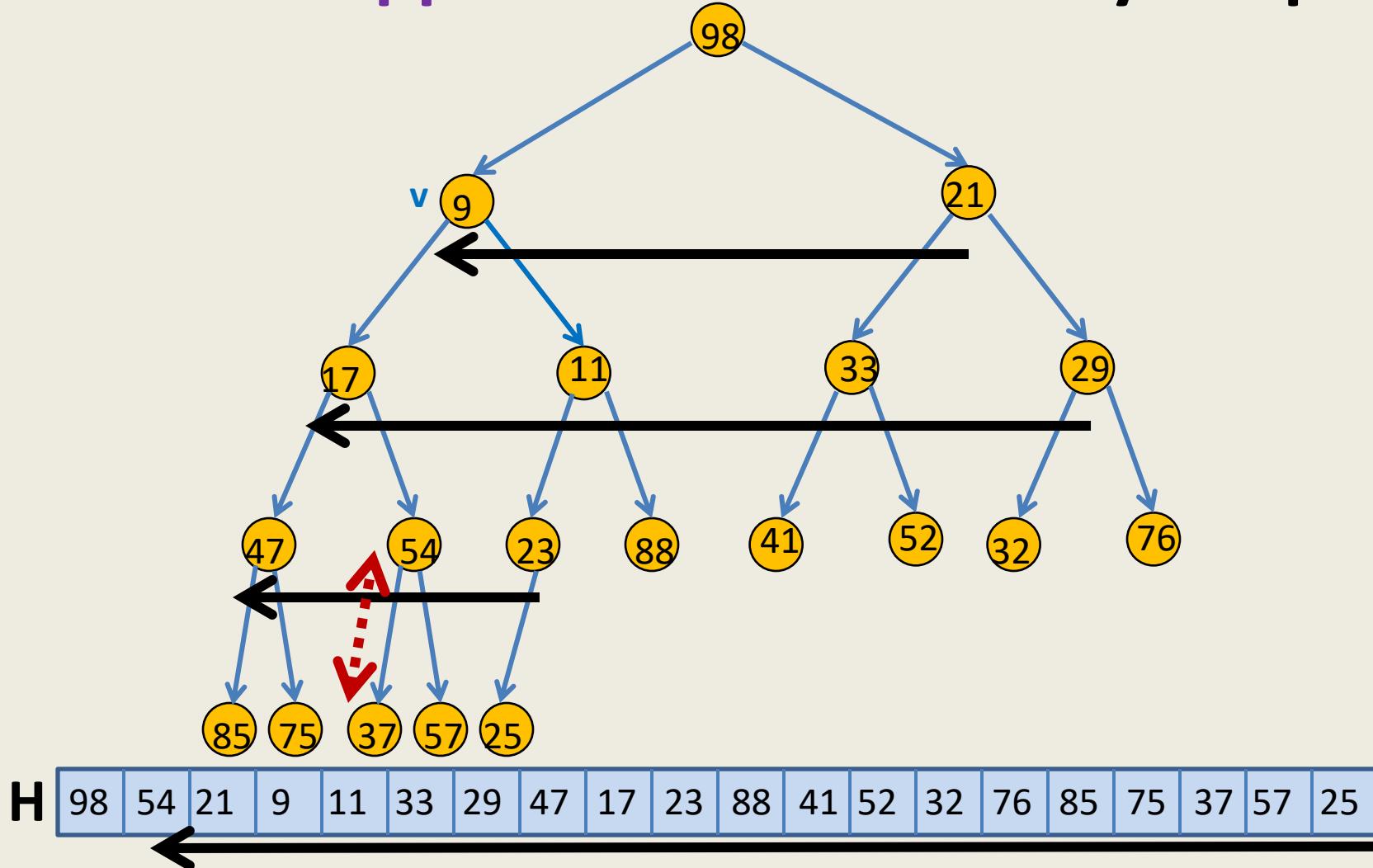
A new approach to build binary heap



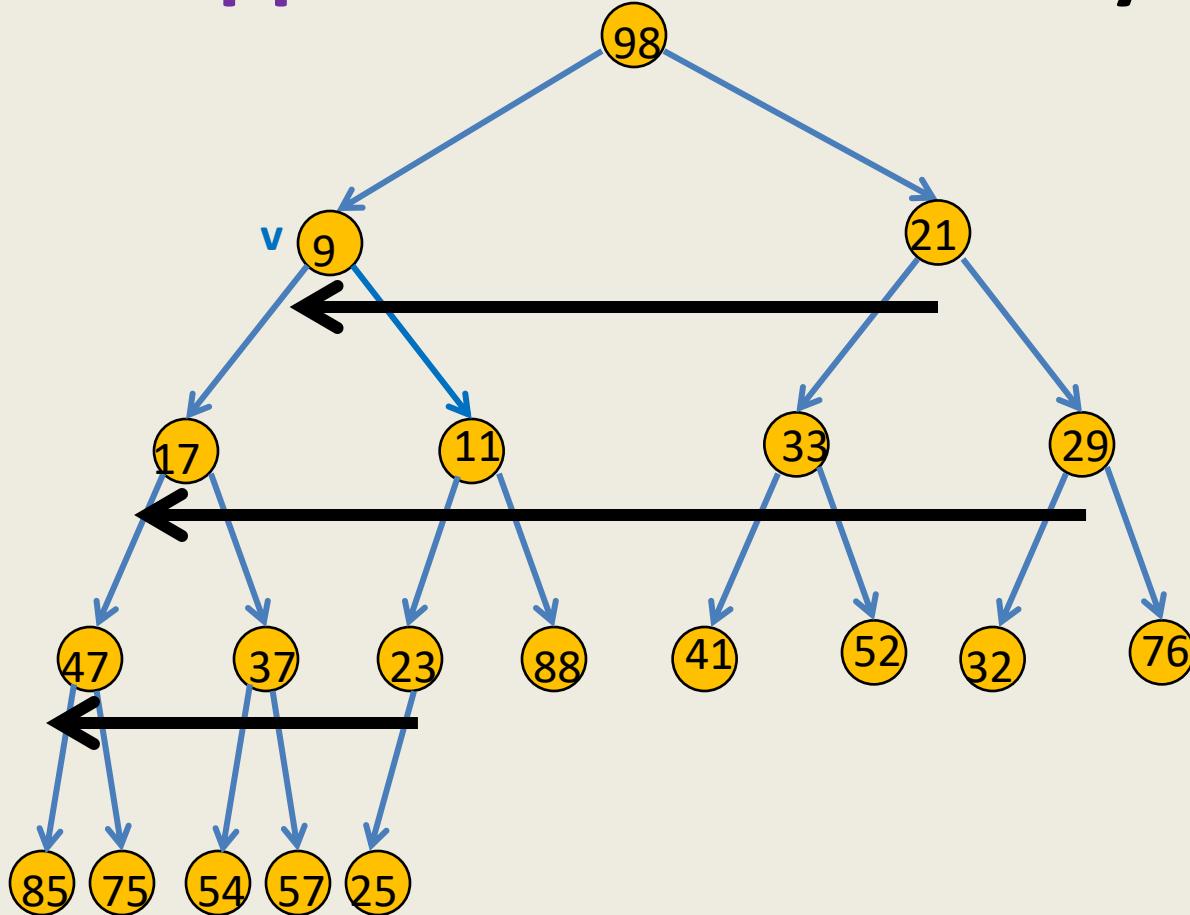
A new approach to build binary heap



A new approach to build binary heap



A new approach to build binary heap



Let v be a node corresponding to index i in H .
The process of restoring heap property at i called **Heapify(i, H)**.

Heapify(i, H)

Heapify(i, H)

{ $n \leftarrow \text{size}(H) - 1$;

While (? and ?)

{

For node i , compare its value with those of its children

If it is smaller than any of its children

→ Swap it with **smallest** child
and move down ...

Else stop !

}

}

Heapify(*i*,H)

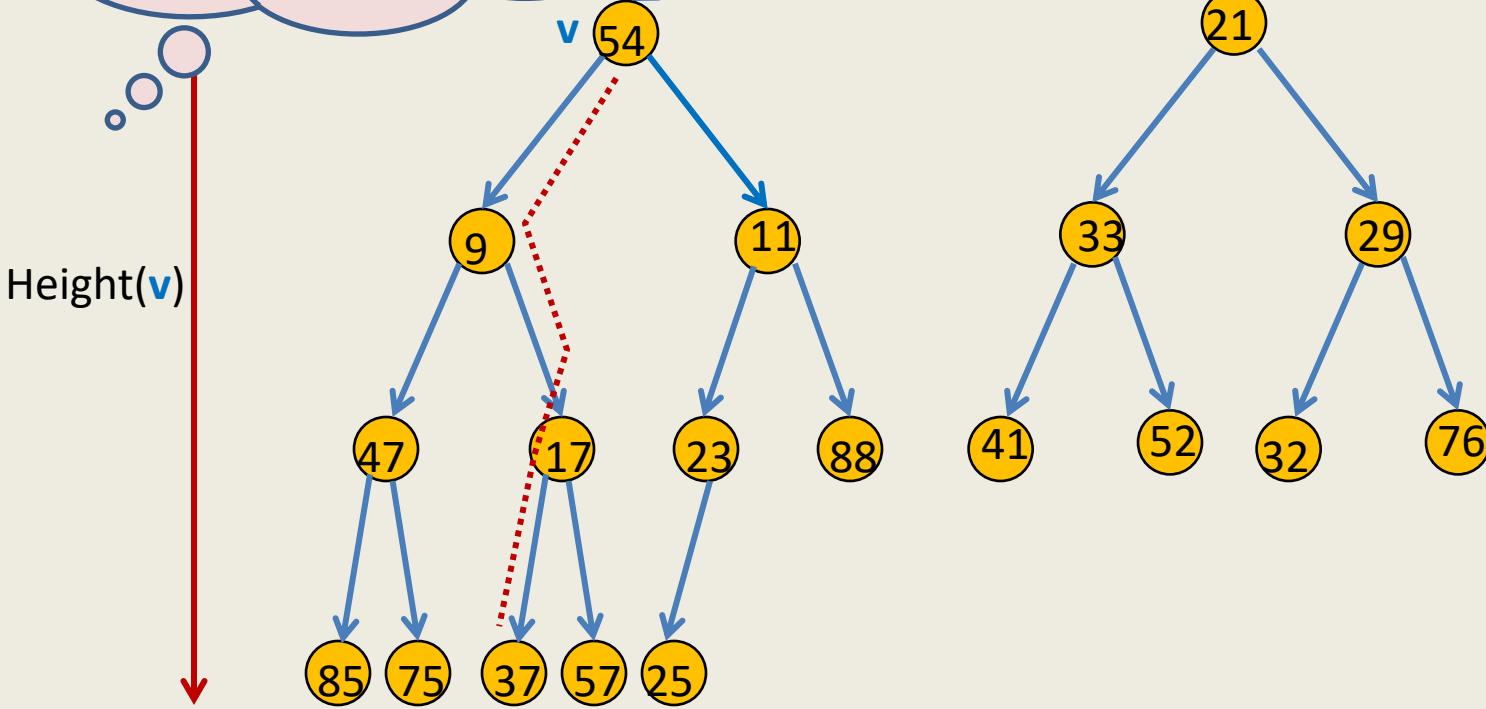
Heapify(*i*,H)

```
{   n ← size(H) -1 ;
    Flag ← true;
    While ( i ≤ ⌊(n-1)/2⌋ and Flag = true )
    {
        min ← i;
        If( H[i]>H[2i + 1] )      min ← 2i + 1;
        If( 2i + 2 ≤ n and H[min]>H[2i + 2] )  min ← 2i + 2;
        If(min ≠ i)
        {
            H(i) ↔ H(min);
            i ← min; }
        else
            Flag ← false;
    }
}
```

Building Binary heap in $O(n)$ time

How many nodes of height h can there be in a complete Binary tree of n nodes ?

Time to heapify node v ?



H	98	54	21	9	11	33	29	47	17	23	88	41	52	32	76	85	75	37	57	25
----------	----	----	----	---	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----

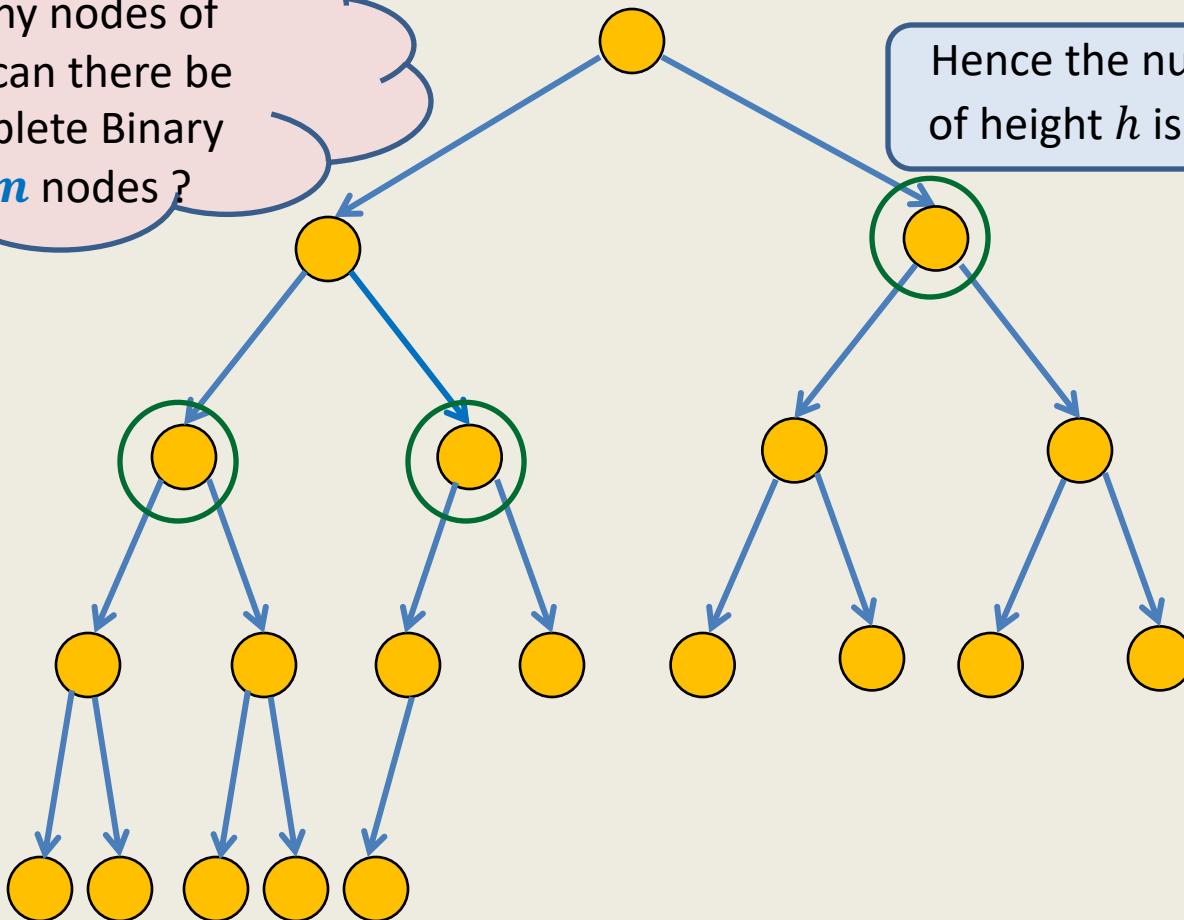
Time complexity of algorithm = $\sum_h O(h) \cdot N(h)$

No. of nodes of height h

A complete binary tree

How many nodes of height h can there be in a complete Binary tree of n nodes?

Hence the number of nodes of height h is bounded by $\frac{n}{2^h}$



Each subtree is also a complete binary tree.

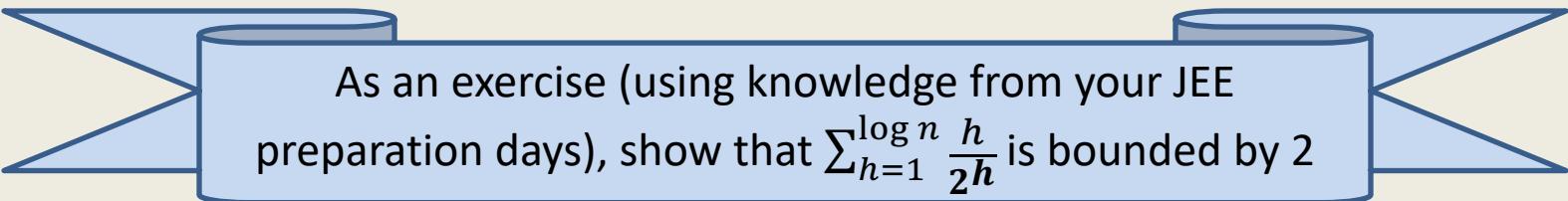
→ A subtree of height h has at least 2^h nodes

Moreover, no two subtrees of height h in the given tree have any element in common

Building Binary heap in $O(n)$ time

Lemma: the number of nodes of height h is bounded by $\frac{n}{2^h}$.

$$\begin{aligned}\text{Hence Time complexity to build the heap} &= \sum_{h=1}^{\log n} \frac{n}{2^h} O(h) \\ &= n \cdot c \cdot \sum_{i=1}^{\log n} \frac{h}{2^h} \\ &= O(n)\end{aligned}$$



As an exercise (using knowledge from your JEE preparation days), show that $\sum_{h=1}^{\log n} \frac{h}{2^h}$ is bounded by 2

Sorting using a Binary heap

Sorting using heap

Build heap H on the given n elements;

While (H is not empty)

```
{    $x \leftarrow \text{Extract-min}(H);$ 
    print  $x$ ;
}
```

This is **HEAP SORT** algorithm

Time complexity : $O(n \log n)$

Question:

Which is the best sorting algorithm : (**Merge** sort, **Heap** sort, **Quick** sort) ?

Answer: Practice programming assignment ☺

Binary trees: beyond searching and sorting

- **Elegant solution for two interesting problem**
- **An important lesson:**

Lack of **proper understanding** of a problem is a big hurdle to solve the problem

Two interesting problems on sequences

What is a sequence ?

A sequence $\mathbf{S} = \langle x_0, \dots, x_{n-1} \rangle$

- Can be viewed as a mapping from $[0, n]$.
- Order does matter.

Problem 1

Multi-increment

Problem 1

Given an initial sequence $S = \langle x_0, \dots, x_{n-1} \rangle$ of numbers,
maintain a compact data structure to perform the following operations:

- **ReportElement(i):**

Report the current value of x_i .

- **Multi-Increment(i, j, Δ):**

Add Δ to each x_k for each $i \leq k \leq j$

Example:

Let the initial sequence be $S = \langle 14, 12, 23, 12, 111, 51, 321, -40 \rangle$

After **Multi-Increment(2,6,10)**, S becomes

$\langle 14, 12, 33, 22, 121, 61, 331, -40 \rangle$

After **Multi-Increment(0,4,25)**, S becomes

$\langle 39, 37, 58, 47, 146, 61, 331, -40 \rangle$

After **Multi-Increment(2,5,31)**, S becomes

$\langle 39, 37, 89, 78, 177, 92, 331, -40 \rangle$

Problem 1

Given an initial sequence $S = \langle x_0, \dots, x_{n-1} \rangle$ of numbers,
maintain a compact data structure to perform the following operations:

- **ReportElement(i):**

Report the current value of x_i .

- **Multi-Increment(i, j, Δ):**

Add Δ to each x_k for each $i \leq k \leq j$

Trivial solution :

Store S in an array $A[0.. n-1]$ such that $A[i]$ stores the current value of x_i .

- **Multi-Increment(i, j, Δ)**

```
{  
    For ( $i \leq k \leq j$ )      A[k]  $\leftarrow$  A[k] +  $\Delta$ ;  
}
```

```
ReportElement( $i$ ){      return A[i]  }
```

$O(j - i) = O(n)$

$O(1)$

Problem 1

Given an initial sequence $S = \langle x_0, \dots, x_{n-1} \rangle$ of numbers,
maintain a compact data structure to perform the following operations:

- **ReportElement(i):**
Report the current value of x_i .
 - **Multi-Increment(i, j, Δ):**
Add Δ to each x_k for each $i \leq k \leq j$
-

Trivial solution :

Store S in an array $A[0.. n-1]$ such that $A[i]$ stores the current value of x_i .

Question: the source of difficulty in breaking the $O(n)$ barrier for **Multi-Increment()** ?

Answer: we need to explicitly maintain S .

Question: who asked/inspired us to maintain S explicitly.

Answer: 1. incomplete understanding of the problem
2. conditioning based on incomplete understanding

Towards efficient solution of Problem 1

Assumption: without loss of generality assume n is power of 2.

Explore ways to maintain sequence S **implicitly** such that

- **Multi-Increment(i, j, Δ)** is efficient
- **Report(i)** is efficient too.

Main hurdle: To perform **Multi-Increment(i, j, Δ)** efficiently

Problem 2

Dynamic Range-minima

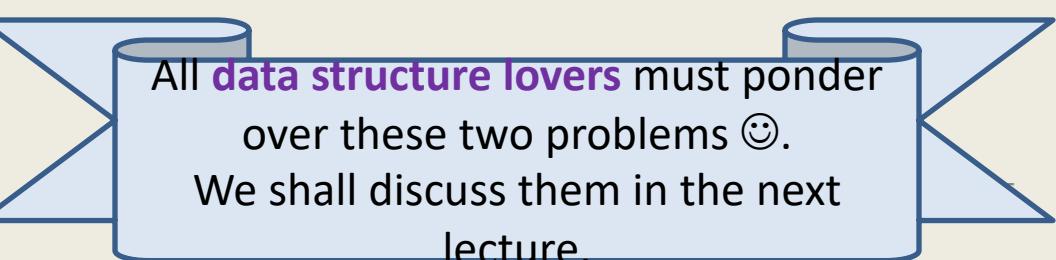
Problem 2

Given an initial sequence $S = \langle x_0, \dots, x_{n-1} \rangle$ of numbers,
maintain a compact data structure to perform the following operations efficiently for
any $0 \leq i < j < n$.

- **ReportMin(i, j):**
Report the minimum element from $\{x_k \mid \text{for each } i \leq k \leq j\}$
- **Update(i, a):**
 a becomes the new value of x_i .

AIM:

- $O(n)$ size data structure.
- ReportMin(i, j) in $O(\log n)$ time.
- Update(i, a) in $O(\log n)$ time.



All **data structure lovers** must ponder
over these two problems 😊.
We shall discuss them in the next
lecture.

Data Structures and Algorithms

(ESO207)

Lecture 30

Binary Trees

Magical applications

Two interesting problems on sequences

Problem 1

Multi-increment

Problem 1

Given an initial sequence $S = \langle x_0, \dots, x_{n-1} \rangle$ of n numbers,
maintain a compact data structure to perform the following operations efficiently :

- **Report(i):**
Report the current value of x_i .
 - **Multi-Increment(i, j, Δ):**
Add Δ to x_k
-

Example:

Let the initial sequence be $S = \langle 14, 12, 23, 12, 111, 51, 321, -40 \rangle$

After **Multi-Increment(2,6,10)**, S becomes

$\langle 14, 12, 33, 22, 121, 61, 331, -40 \rangle$

Trivial solution discussed in the last class :

- $O(n)$ time per **Multi-Increment(i, j, Δ)**
- $O(1)$ time per **Report(i)**

Towards efficient solution of Problem 1

Explore ways to maintain sequence S **implicitly** such that

- **Multi-Increment(i, j, Δ)** is efficient.
- **Report(i)** is efficient too.

Main hurdle: To perform **Multi-Increment(i, j, Δ)** efficiently

Assumption: without loss of generality assume n is power of 2.

A SYSTEMATIC JOURNEY TO THE SOLUTION

A motivating problem

$$S = \{1, 2, 3, \dots, 2^n\}$$

Question:

Can we have a small set $X \subset S$ of numbers s.t.

Every number from S can be expressed as a sum of a few numbers from X ?

Answer: $X = \{1, 2, 4, 8, \dots, 2^n\}$

$$|X| = n$$

1 0 0 0 0 0 0 0 0

1 0 0 0 0 0 0

1 0 0 0 0

1 0 0 0

1

1 0 0 1 0 1 1 0 0 1

If it is too trivial, try to answer the problem of next slide. ☺

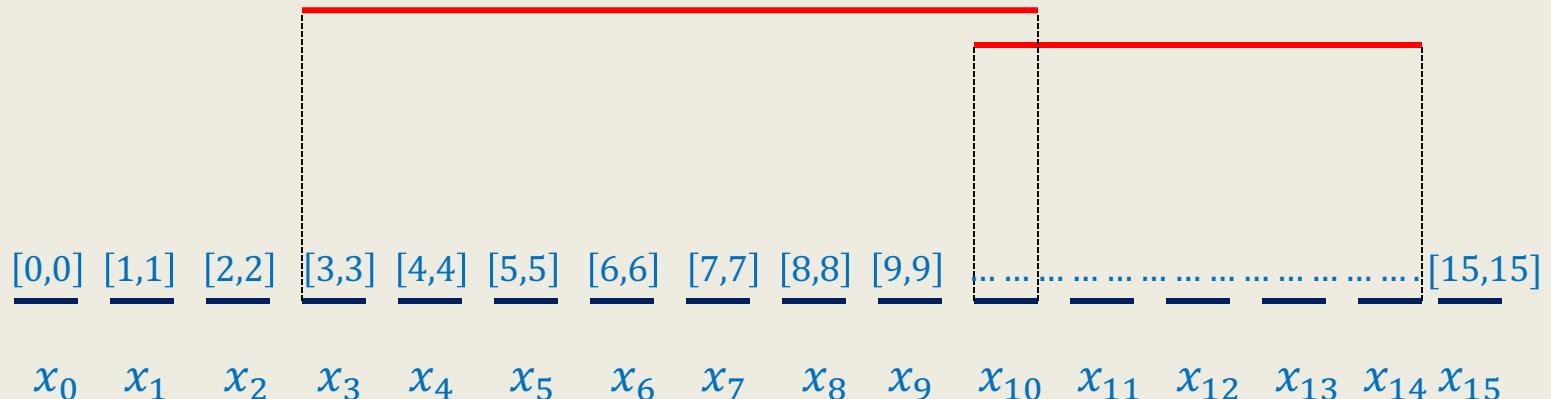
Extension to intervals

$$S = \{[i, j], 0 \leq i \leq j < n\}$$

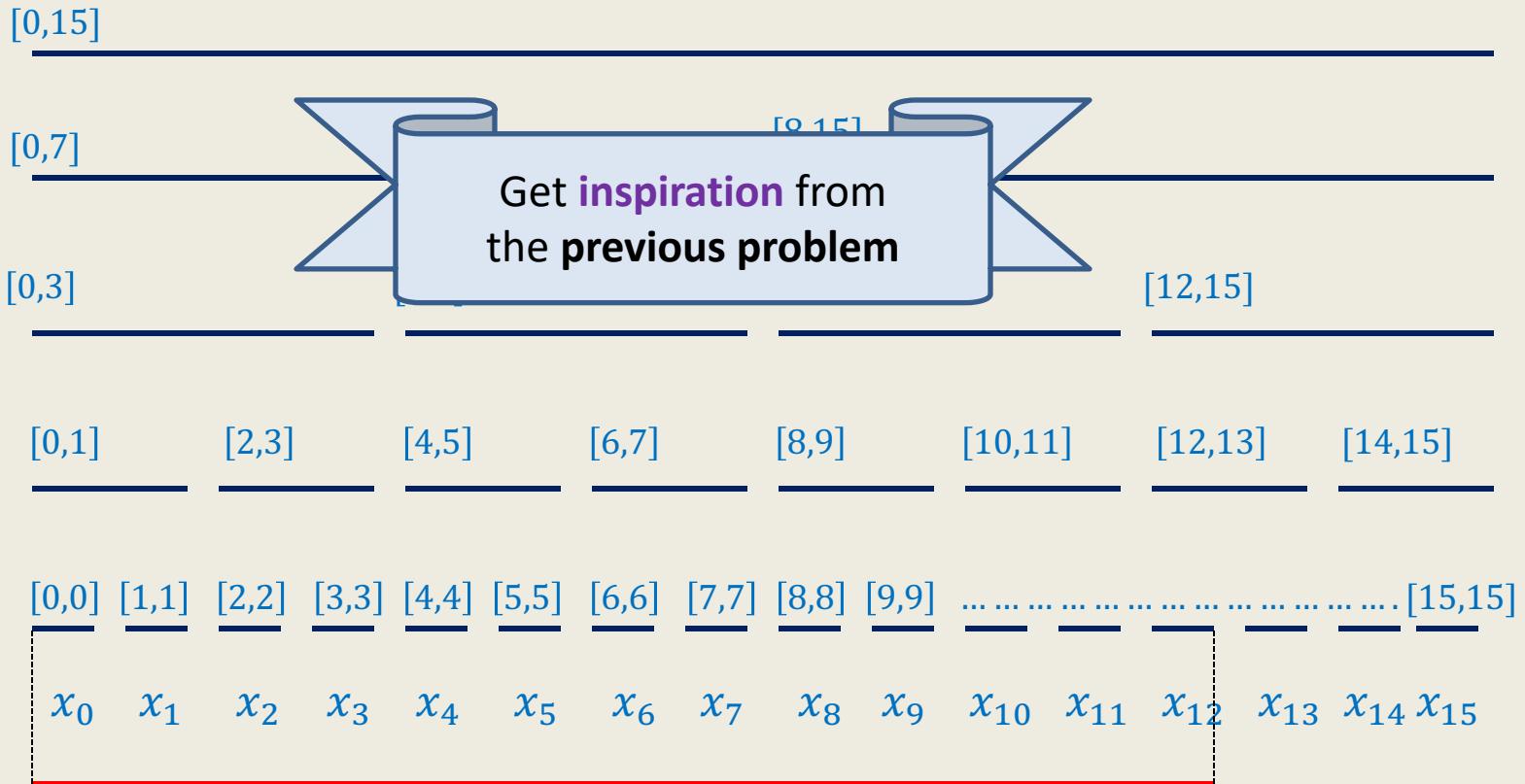
Question:

Can we have a small set $X \subset S$ of **intervals** s.t.

every interval in S can be expressed as a union of a few intervals from X ?

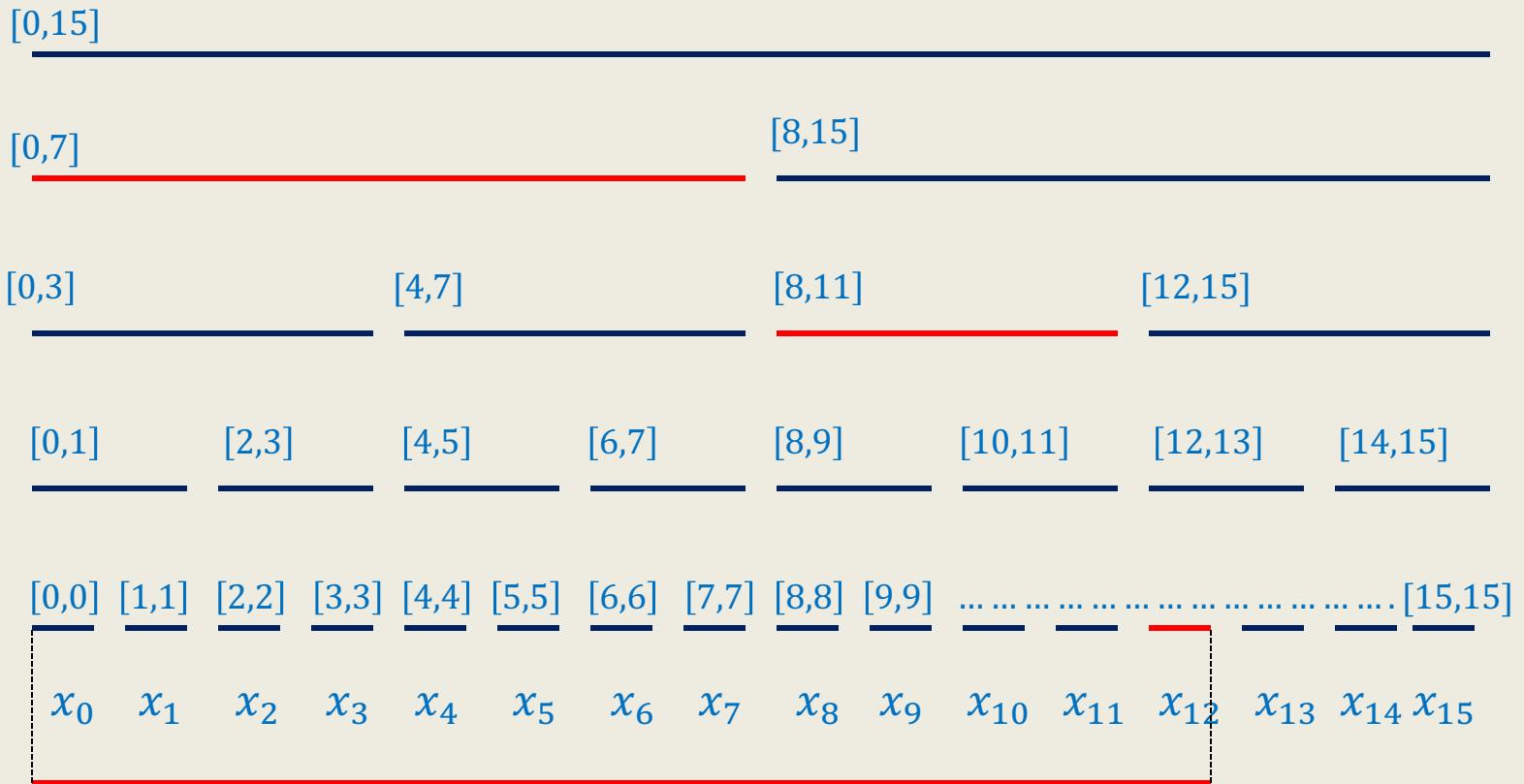


Extension to intervals



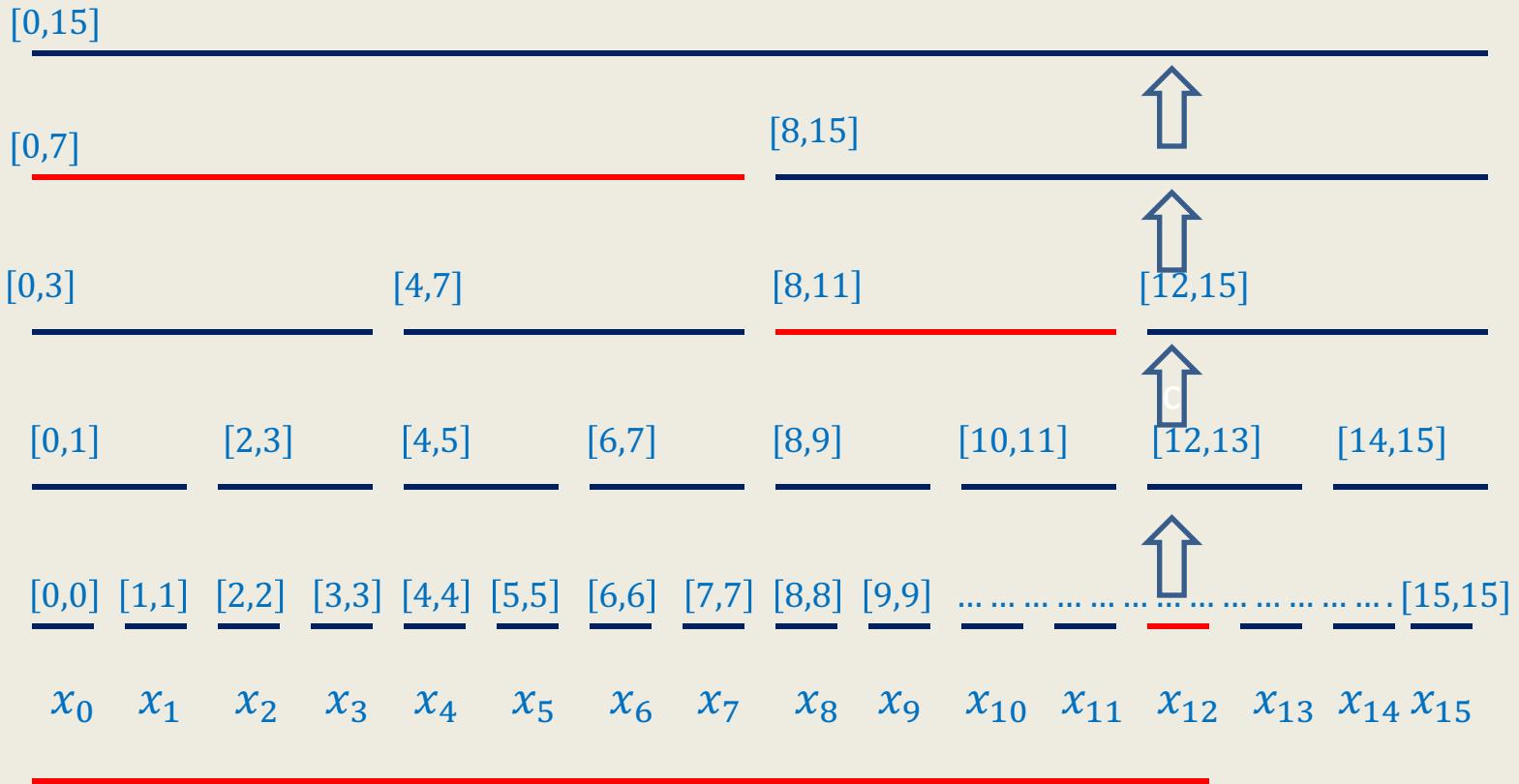
How to express [0, 12] ?

Extension to intervals



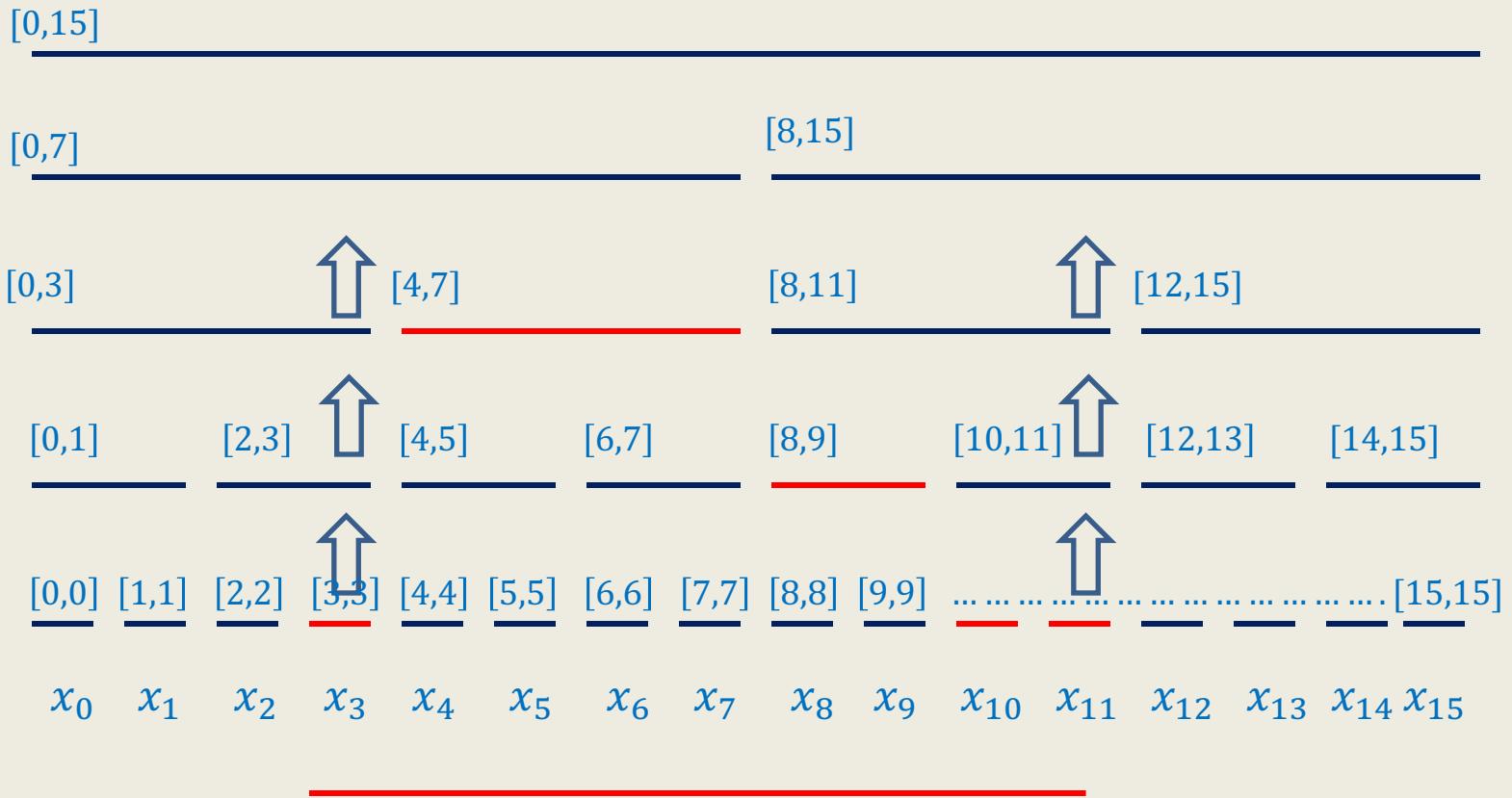
How to express [0, 12] ?

Extension to intervals



How to express $[0, 12]$?

Extension to intervals



How to express $[3, 11]$?

How to use this **Observation** to perform Multi-Increment(i, j, Δ) efficiently?

[0,15]

[0,7]

[8,15]

[0,3]

[4,7]

[8,11]

[12,15]

[0,1]

[2,3]

[4,5]

[6,7]

[8,9]

[10,11]

[12,13]

[14,15]

[0,0]

[1,1]

[2,2]

[3,3]

[4,4]

[5,5]

[6,6]

[7,7]

[8,8]

[9,9]

..... [15,15]

x_0

x_1

x_2

x_3

x_4

x_5

x_6

x_7

x_8

x_9

x_{10}

x_{11}

x_{12}

x_{13}

x_{14}

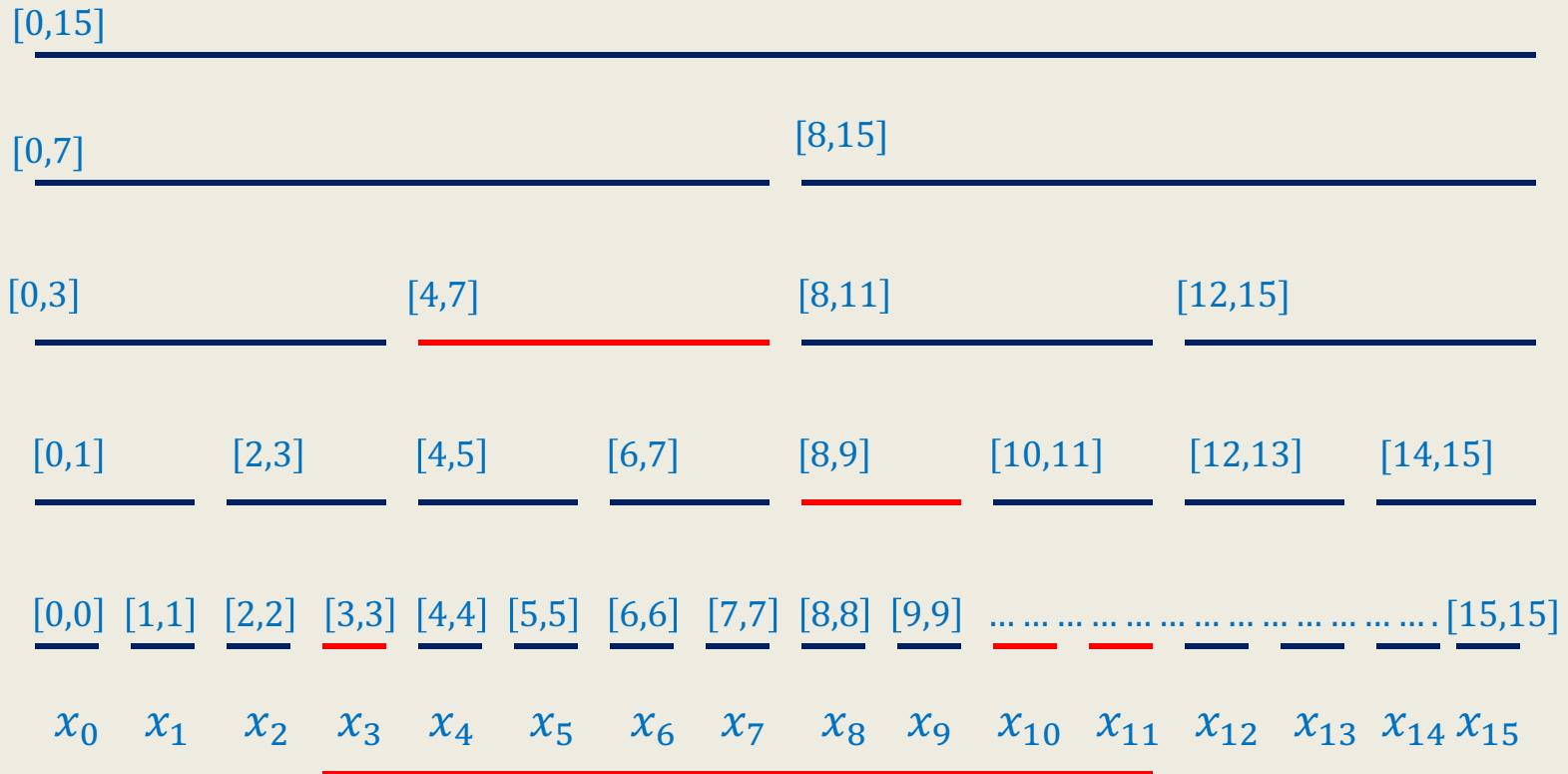
x_{15}

Observation:

There are $2n$ intervals such that

any interval $[i, j]$ can be expressed as union of $O(\log n)$ basic intervals ☺

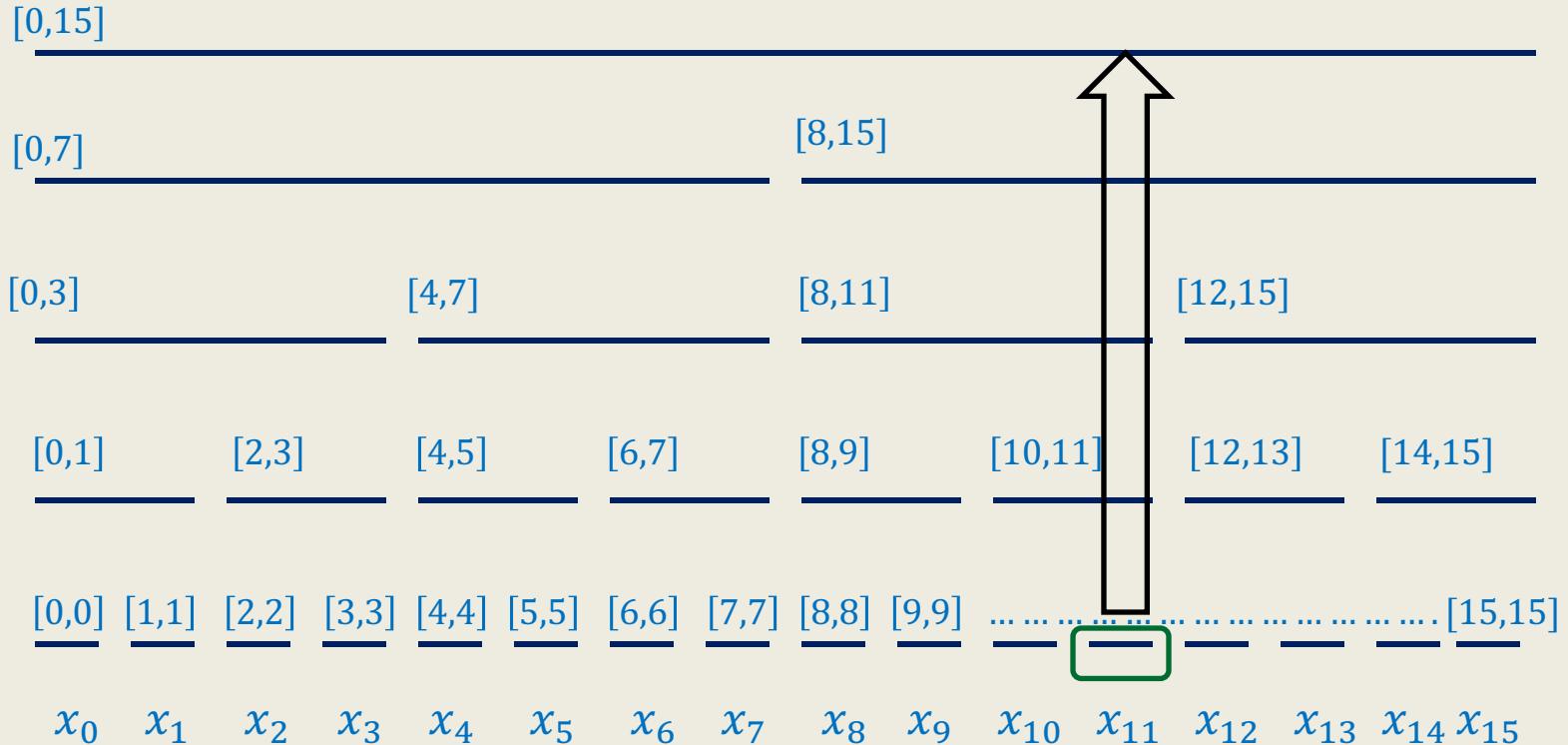
How to use this **Observation** to perform Multi-Increment(i, j, Δ) efficiently?



Maintain $2n$ intervals with a field **increment**

Multi-Increment(i, j, Δ) \Rightarrow add Δ to **increment** field of its $O(\log n)$ intervals.

How to use this **Observation** to perform Multi-Increment(i, j, Δ) efficiently?

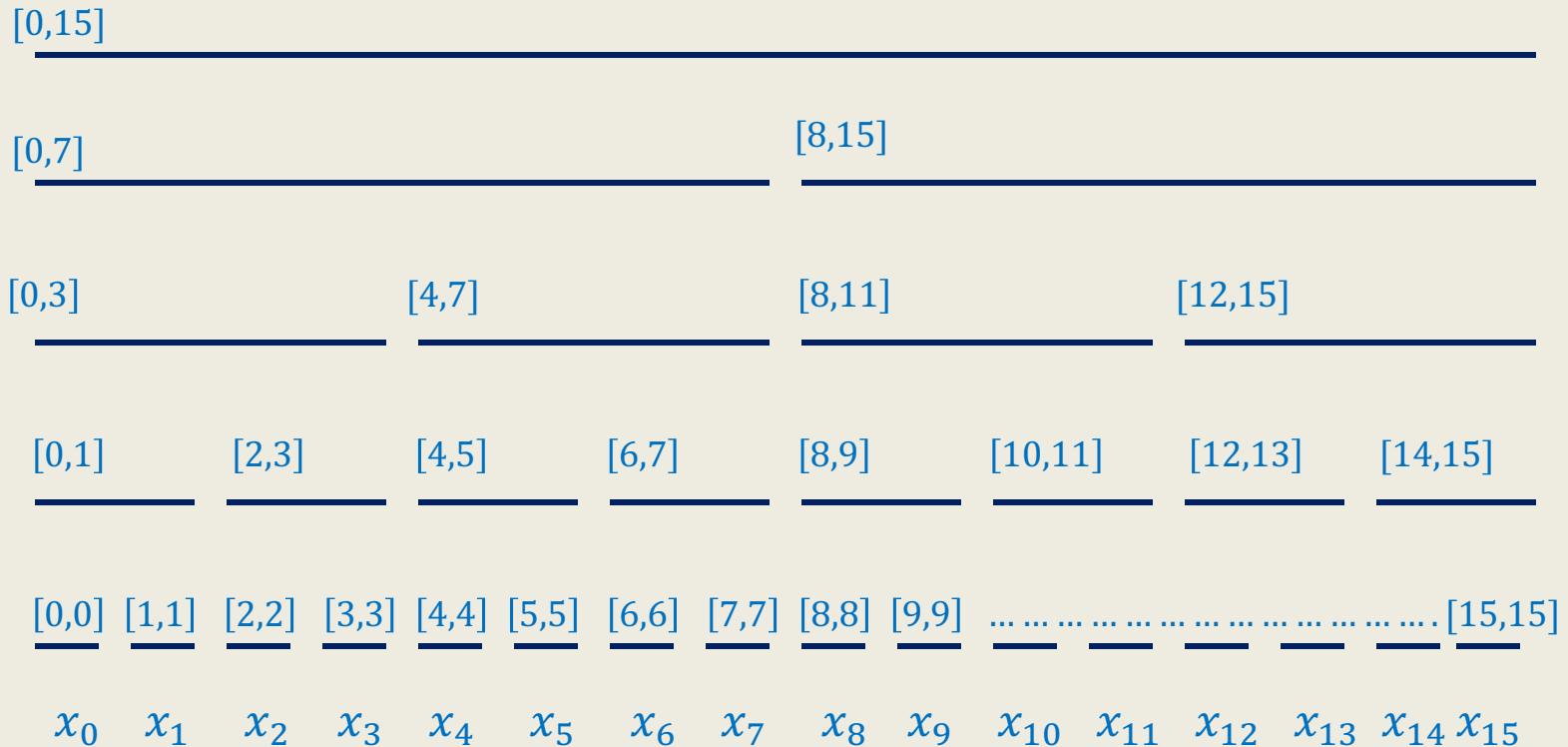


Maintain $2n$ intervals with a field **increment**

Multi-Increment(i, j, Δ) \Rightarrow add Δ to **increment** field of its $O(\log n)$ intervals.

How to perform **Report(i)** ?

How to use this **Observation** to perform Multi-Increment(i, j, Δ) efficiently?



Maintain $2n$ intervals with a field **increment**

Multi-Increment(i, j, Δ) \Rightarrow add Δ to **increment** field of its $O(\log n)$ intervals.

How to perform **Report(i)** ?

What data structure to use ?

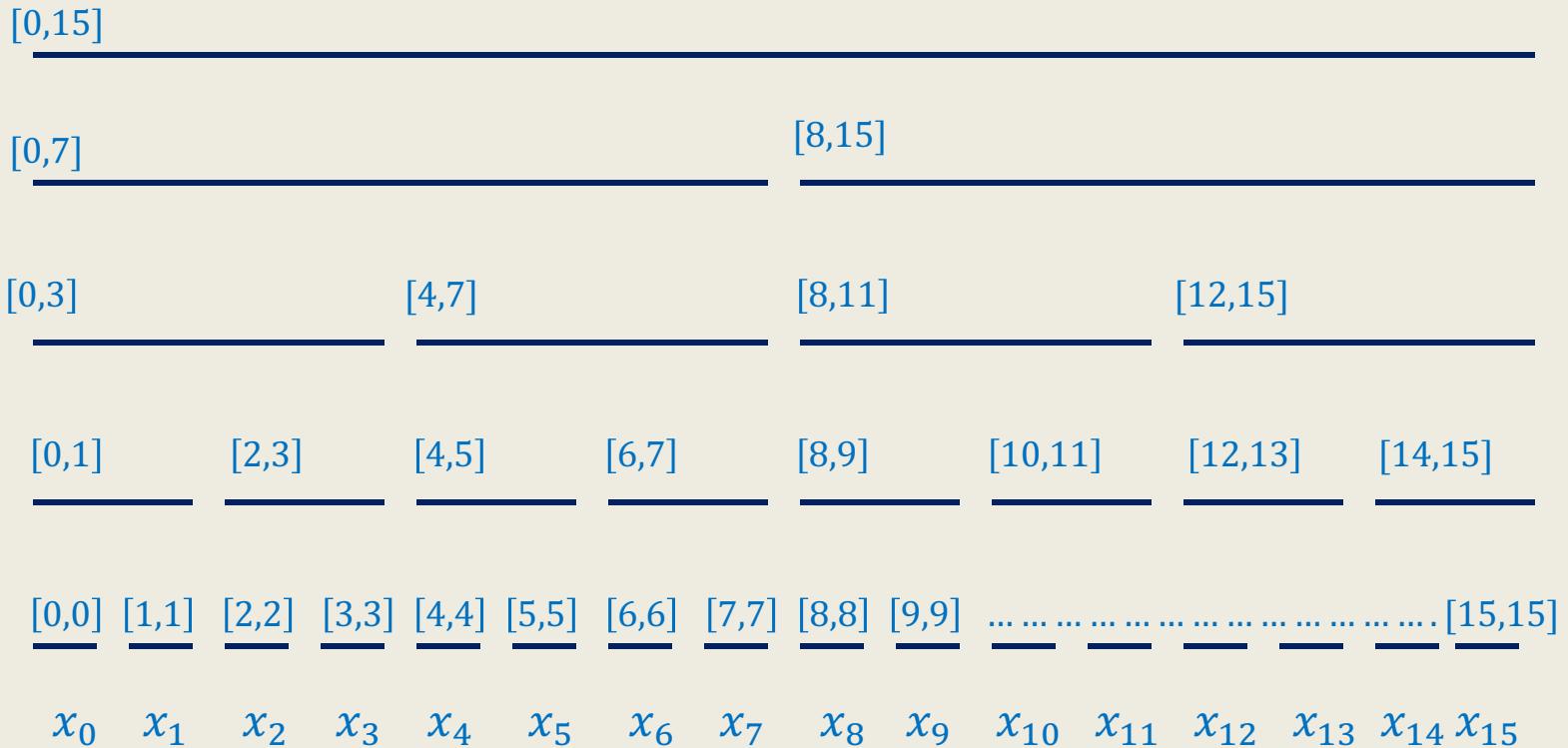
You might like to have another look on the last slide to answer this question.
I have **reproduced** it for you again in the next slide

Have another look,

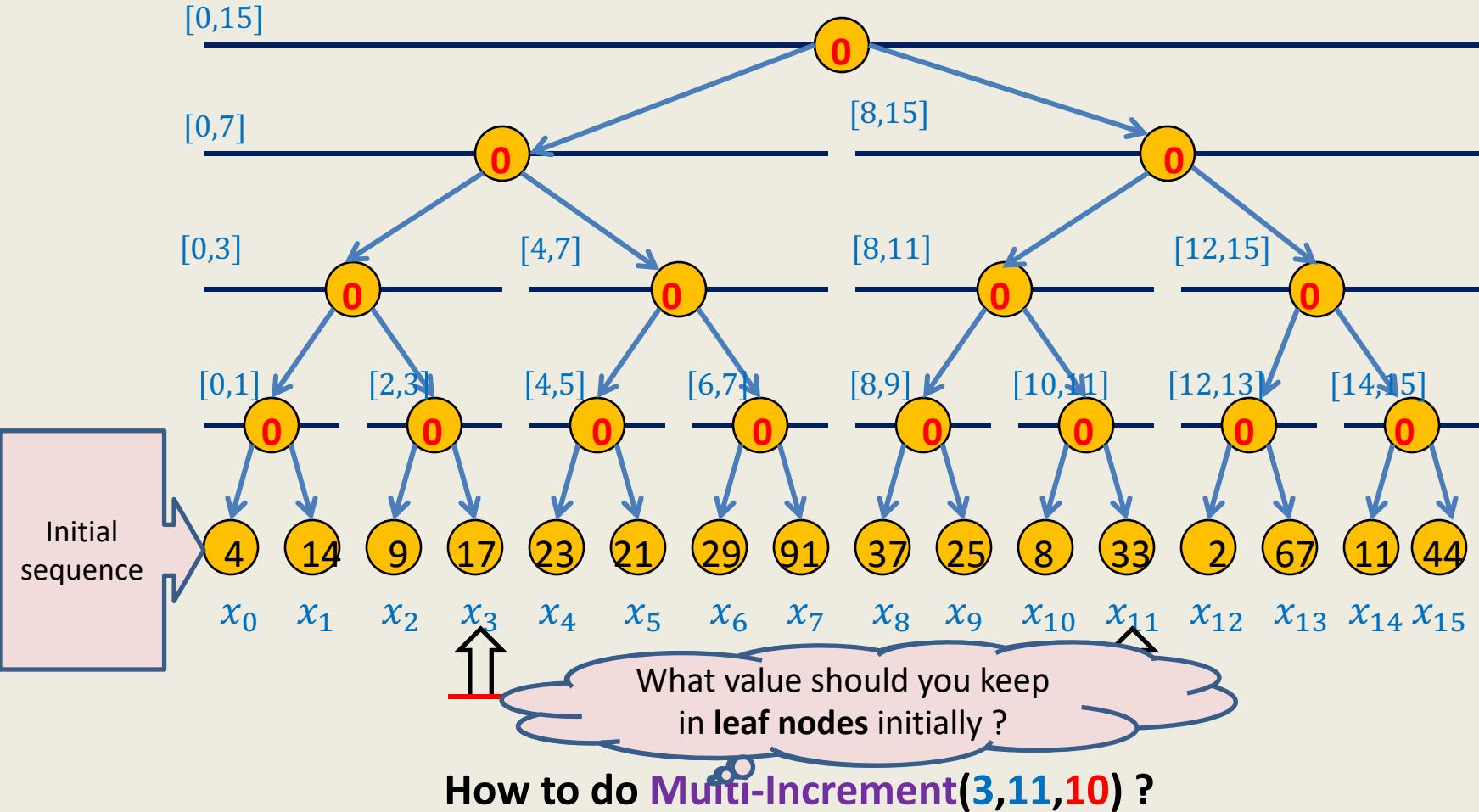
think for a while ...

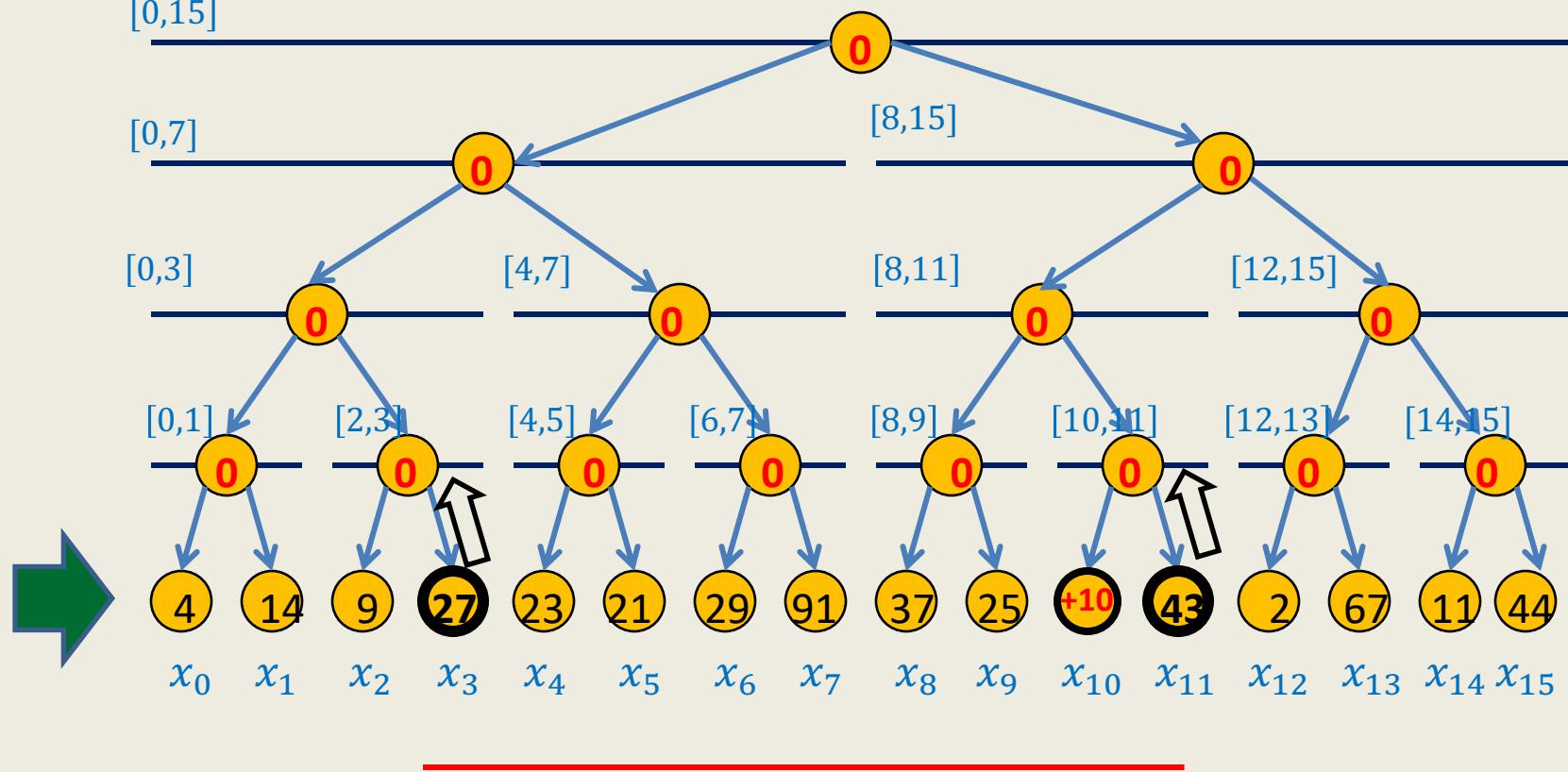
and then only proceed.

Which data structure emerges ?

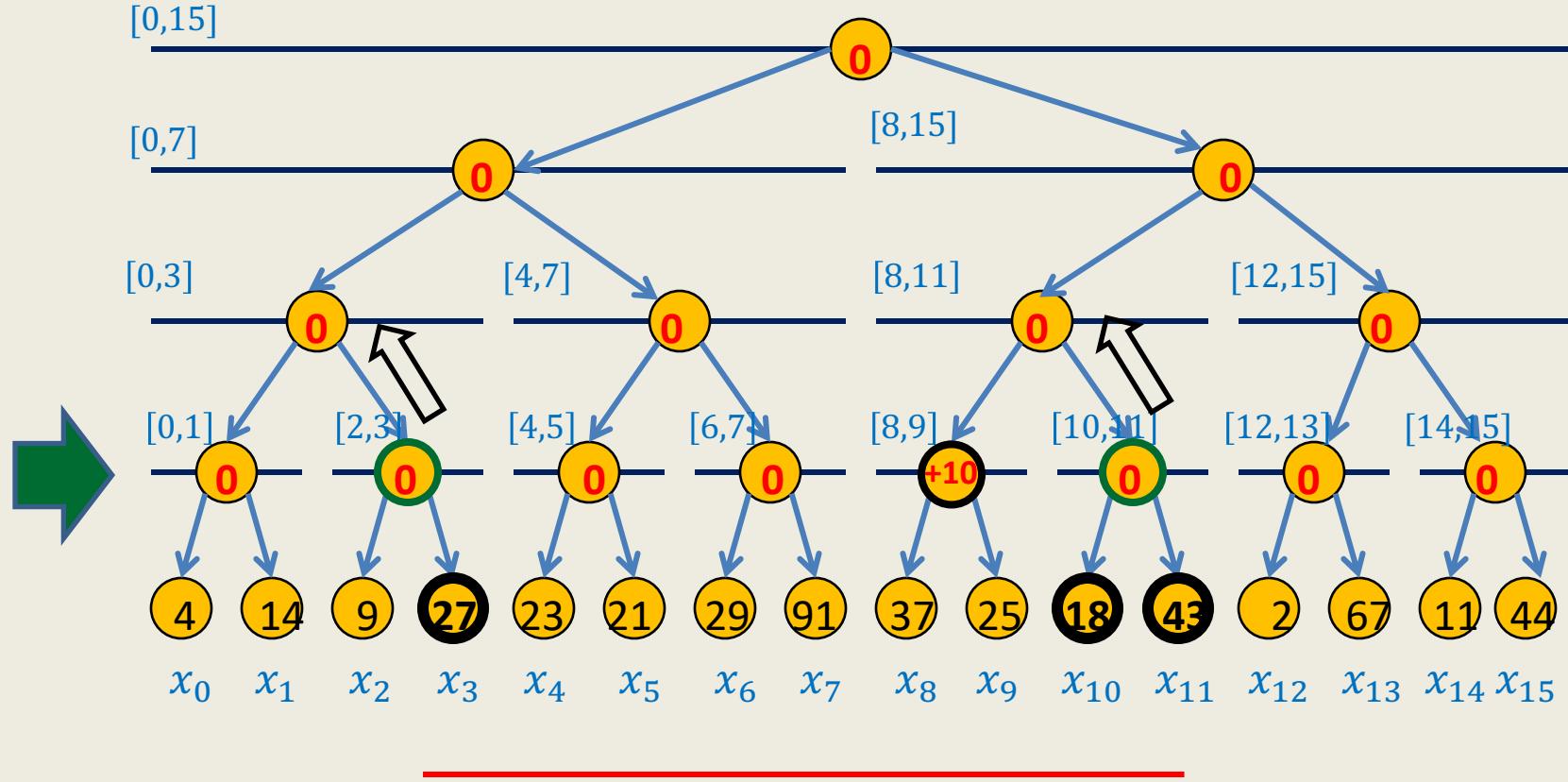


Isn't it a **Binary tree** that you thought ?

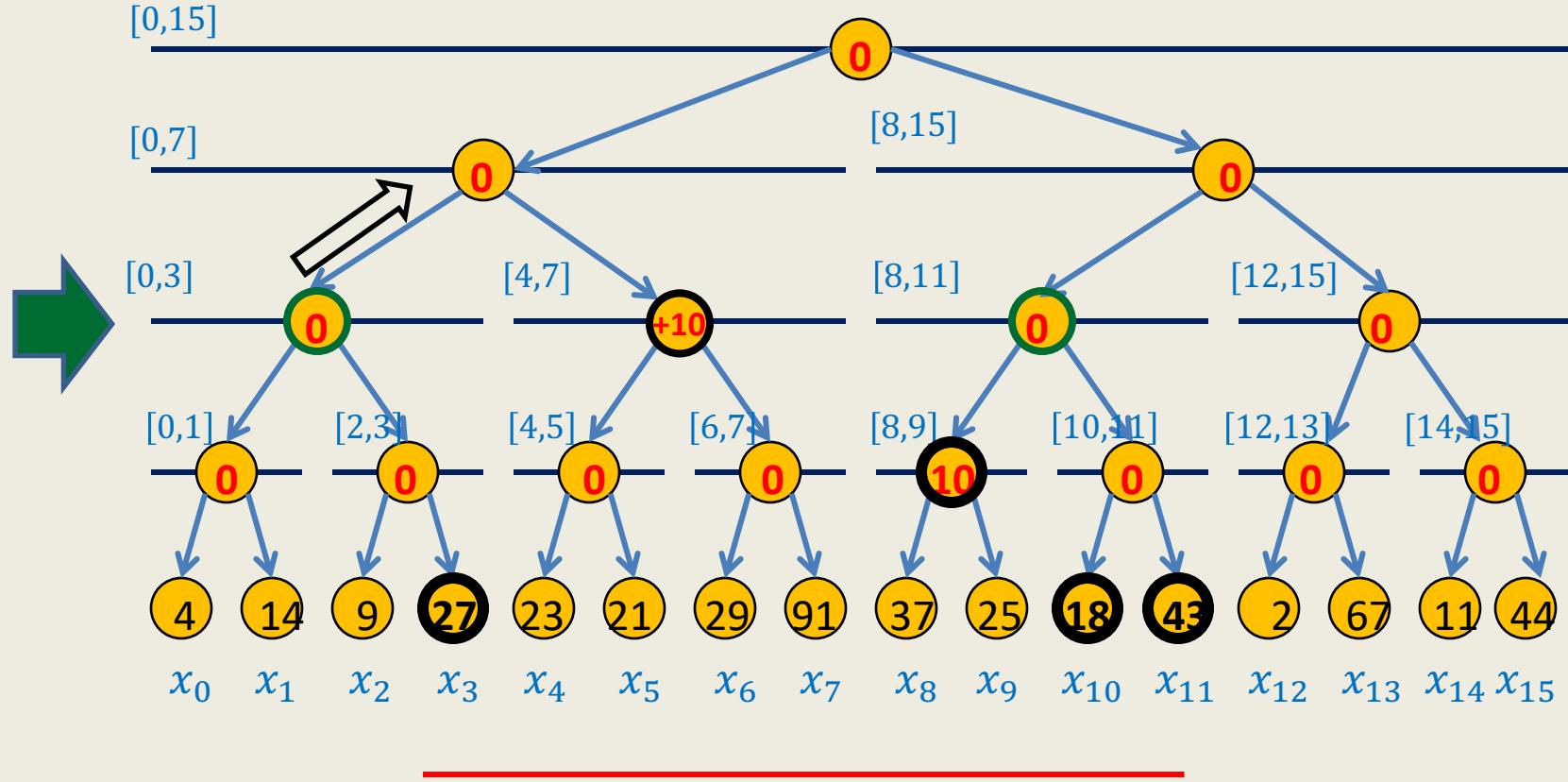




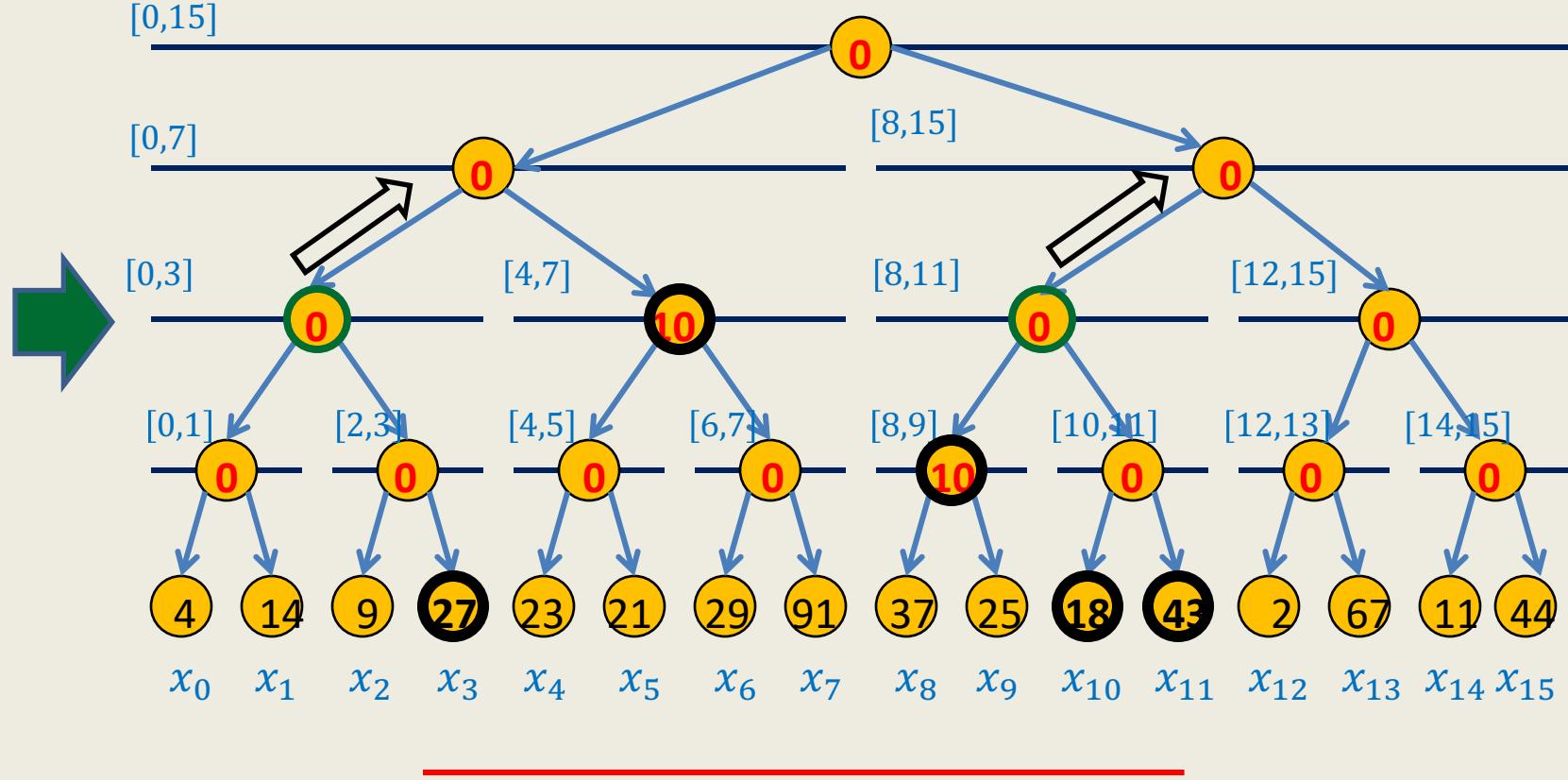
How to do Multi-Increment(3,11,10) ?



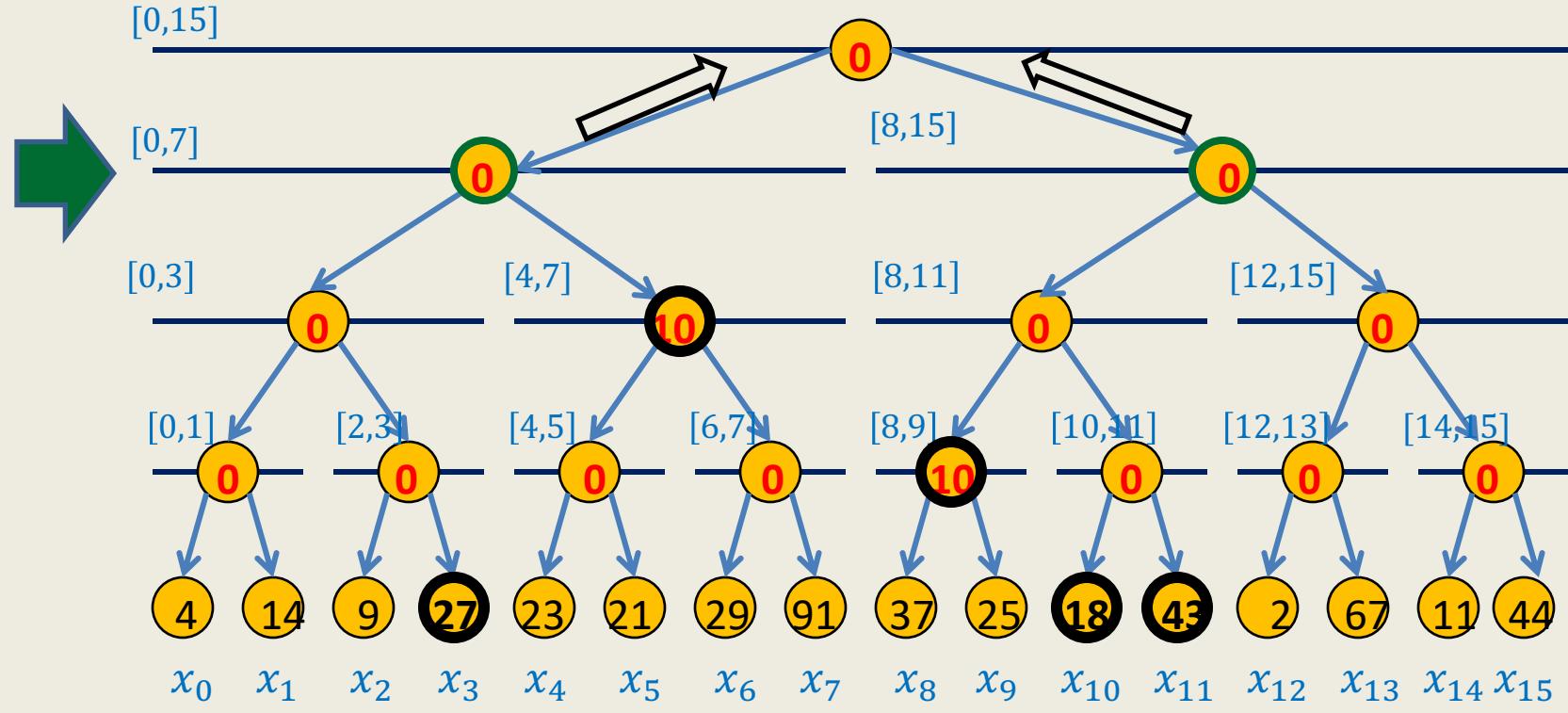
How to do Multi-Increment(3,11,10) ?



How to do Multi-Increment(3,11,10) ?

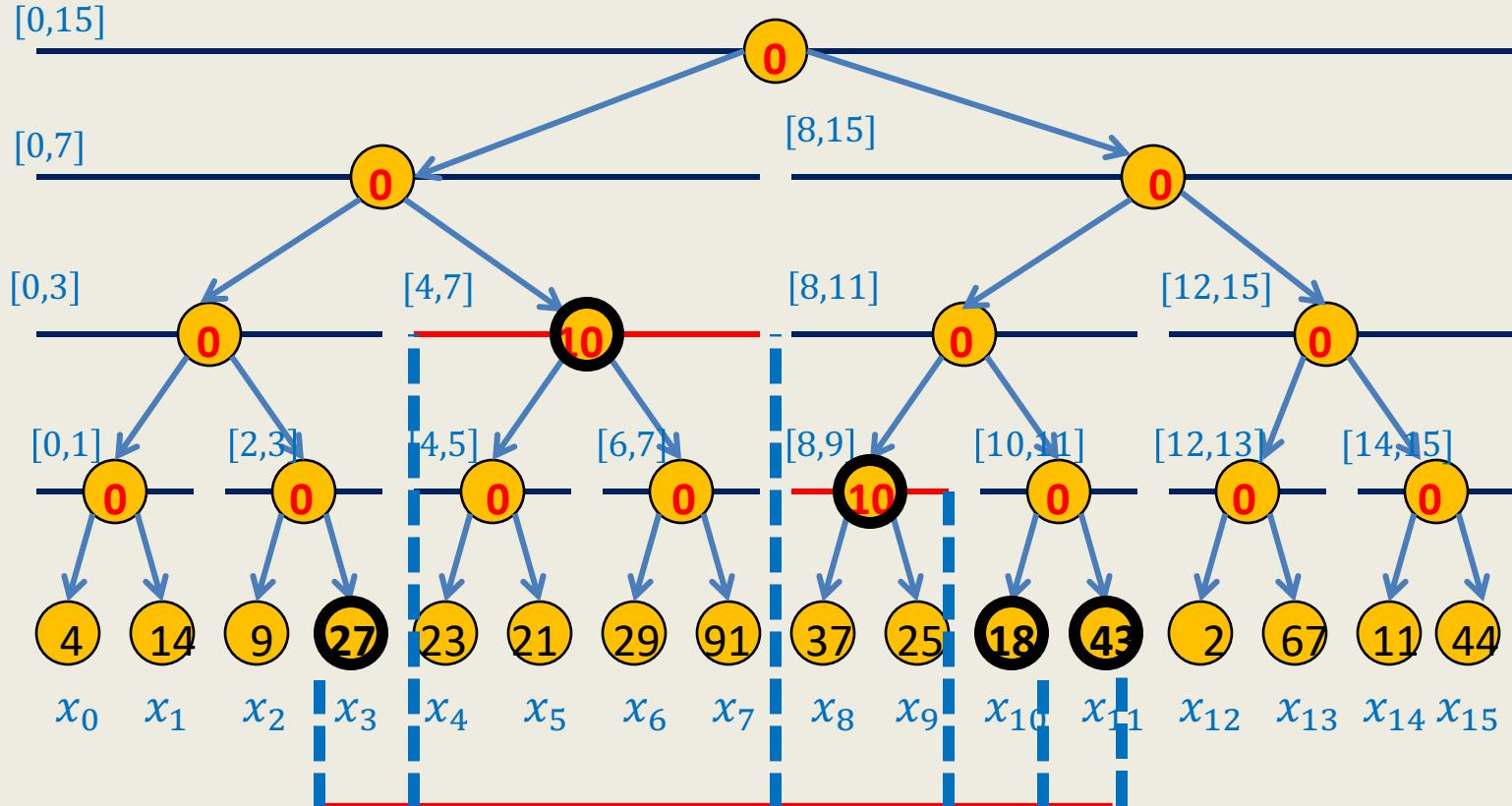


How to do Multi-Increment(3,11,10) ?



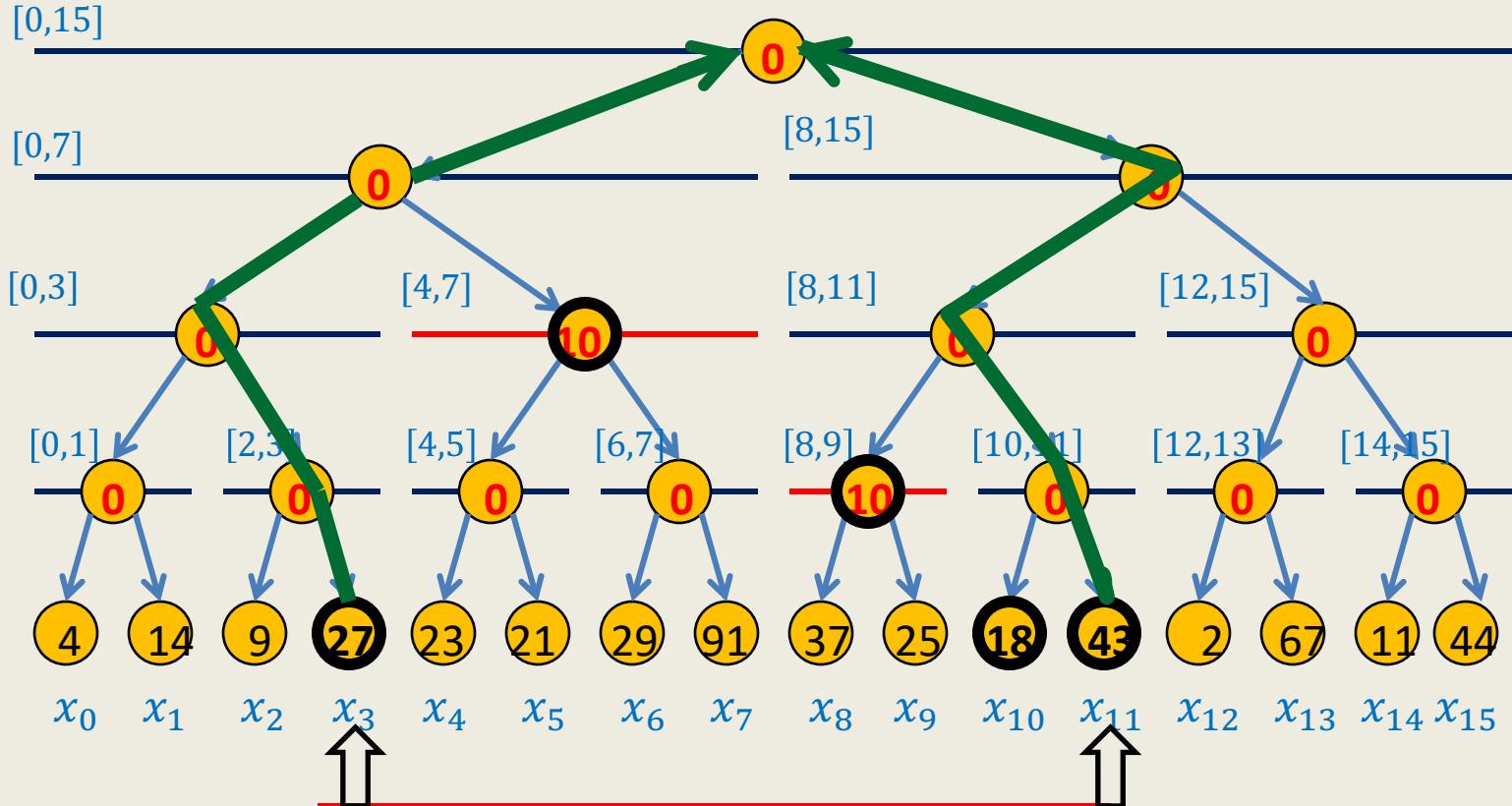
How to do Multi-Increment(3,11,10) ?

Are we done ?



How to do Multi-Increment(3,11,10) ?

Yes



How to do Multi-Increment(3,11,10) ?

What path was followed ?

Multi-Increment(i, j, Δ) efficiently

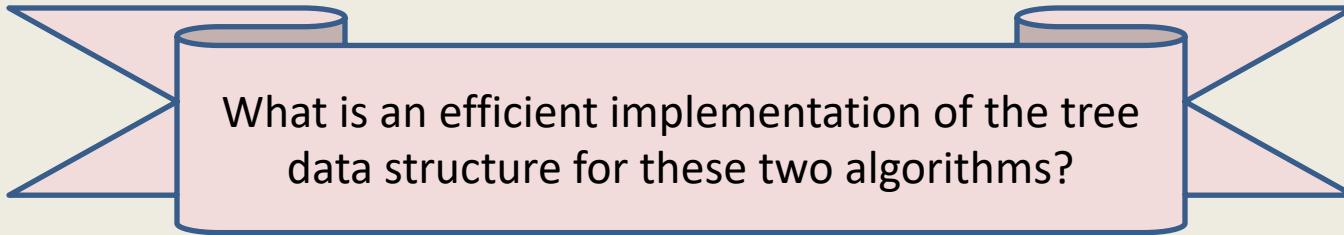
Sketch:

1. Let \mathbf{u} and \mathbf{v} be the leaf nodes corresponding to x_i and x_j .
2. Increment the value stored at \mathbf{u} and \mathbf{v} .
3. Keep repeating the following **step** as long as $\text{parent}(\mathbf{u}) <> \text{parent}(\mathbf{v})$
Move up by one step simultaneously from \mathbf{u} and \mathbf{v}
 - If \mathbf{u} is **left child** of its parent, increment value stored in sibling of \mathbf{u} .
 - If \mathbf{v} is **right child** of its parent, increment value stored in sibling of \mathbf{v} .

Executing Report(i) efficiently

Sketch:

1. Let u be the leaf nodes corresponding to x_i .
2. $\text{val} \leftarrow 0$;
3. Keep moving up from u and keep adding the value of all the nodes on the path to the root to val .
4. Return val .

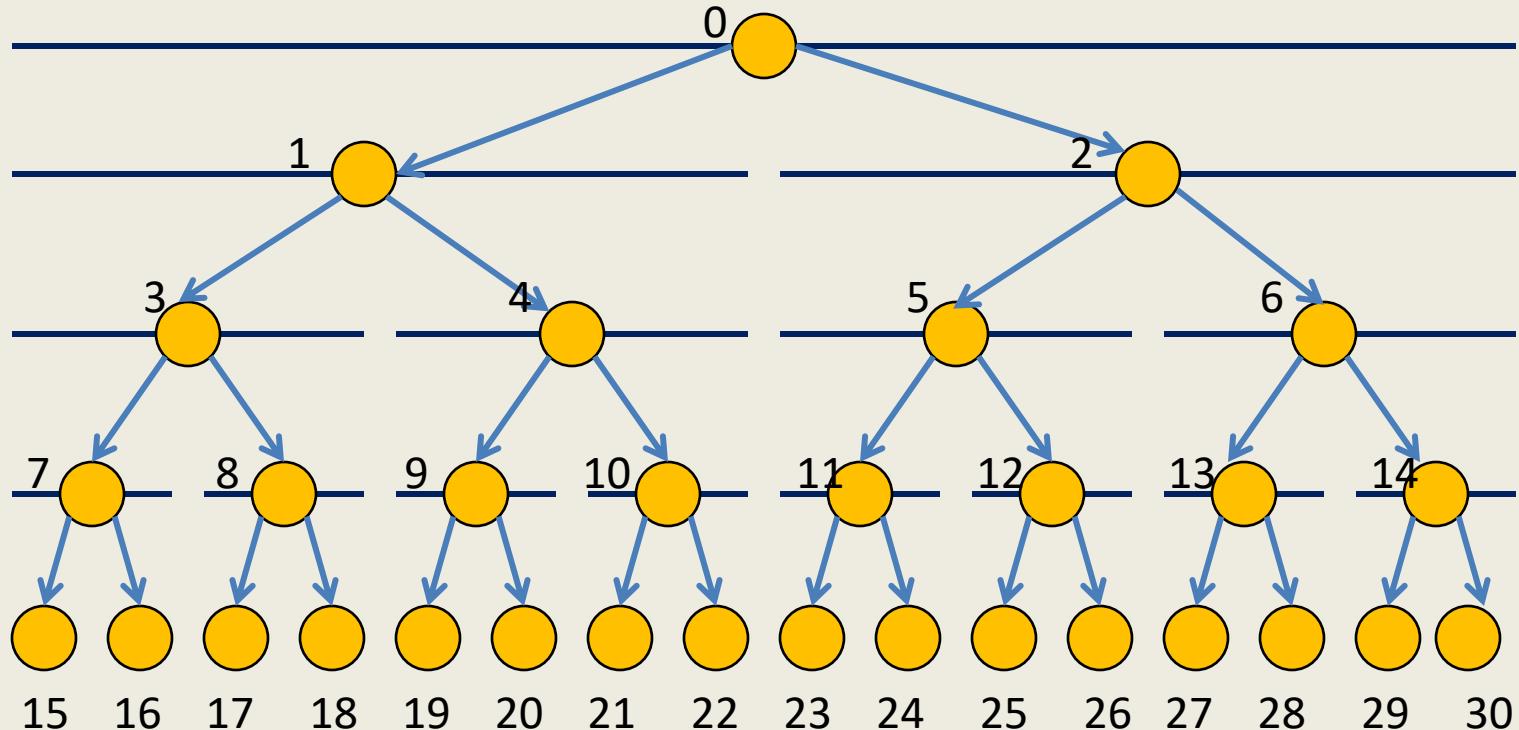


What is an efficient implementation of the tree data structure for these two algorithms?



Realize that it was a **complete binary tree**.

Exploiting complete binary tree structure



Data structure: An array A of size $2n-1$.

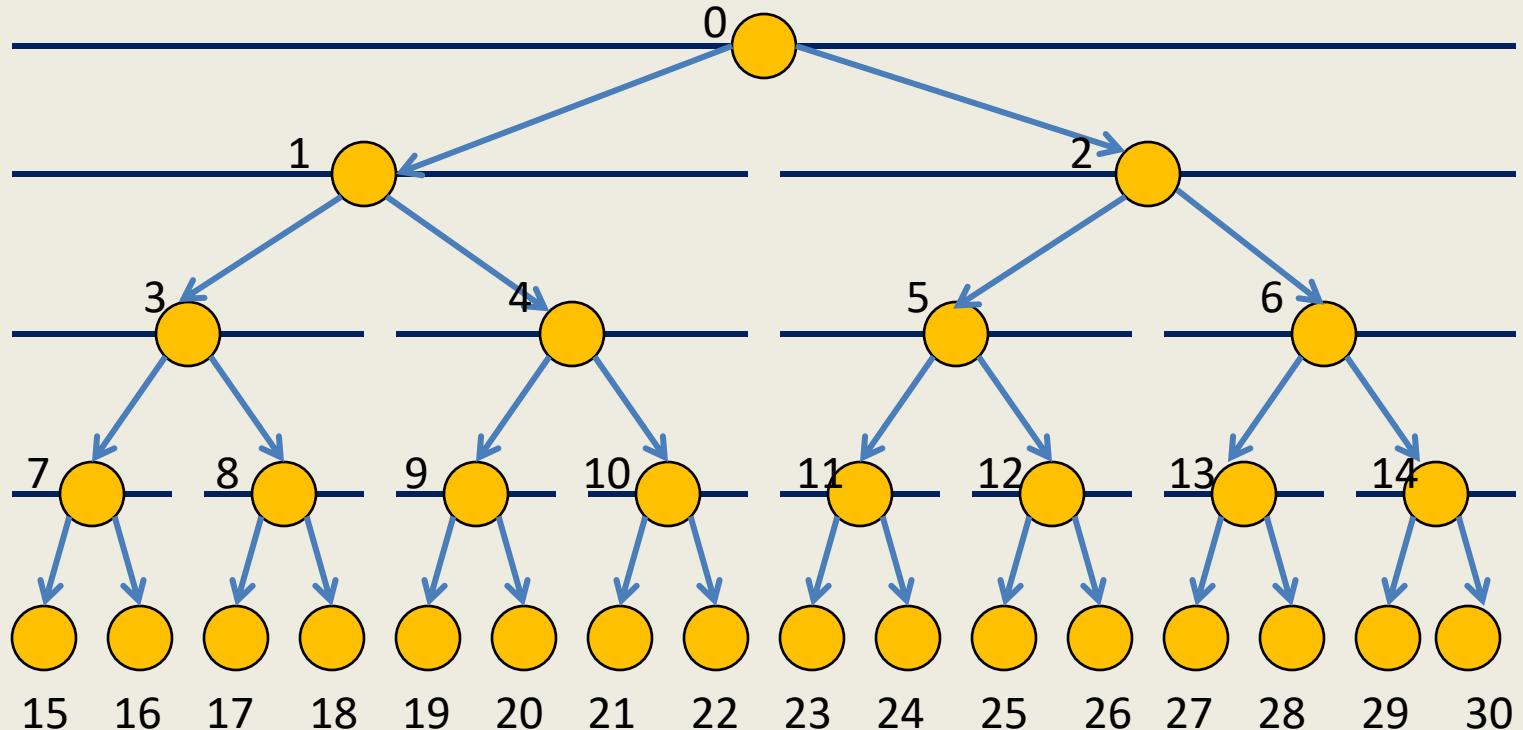
Copy the sequence $S = \langle x_0, \dots, x_{n-1} \rangle$ into $A[n-1] \dots A[2n-2]$

Leaf node corresponding to $x_i = A[(n-1) + i]$

How to check if a node is **left child or right child** of its parent ?

(if index of the node is odd, then the node is left child, else the node is right child)

Exploiting complete binary tree structure



Data structure: An array \mathbf{A} of size $2n-1$.

Copy the sequence $\mathbf{s} = \langle x_0, \dots, x_{n-1} \rangle$ into $\mathbf{A}[n-1] \dots \mathbf{A}[2n-2]$

Leaf node corresponding to $x_i = \mathbf{A}[(n-1) + i]$

How to check if a node is **left child or right child** of its parent ?

Multilncrement(i, j, Δ)

Multilncrement(i, j, Δ)

```
i < (n - 1) + i ;
j < (n - 1) + j ;
A(i) < A(i) + Δ;
If (j > i)
{
    A(j) < A(j) + Δ;
    While( ⌊(i - 1)/2⌋ <> ⌊(j - 1)/2⌋ )
    {
        If( i%2=1 )    A(i + 1) < A(i + 1) + Δ;
        If( j%2=0 )    A(j - 1) < A(j - 1) + Δ;
        i < ⌊(i - 1)/2⌋ ;
        j < ⌊(j - 1)/2⌋ ;
    }
}
```

Report(*i*)

Report(*i*)

```
i ← (n − 1) + i ;  
val ← 0;  
While(i > 0 )  
{  
    val ← val + A[i];  
    i ← ⌊(i − 1)/2⌋ ;  
}  
return val;
```

The solution of Multi-Increment Problem

Theorem:

There exists a data structure of size $\mathbf{O}(n)$
for maintaining a sequence $S = \langle x_0, \dots, x_{n-1} \rangle$ such that
each **Multi-Increment()** and **Report()** operation takes $\mathbf{O}(\log n)$ time.

Problem 2

Dynamic Range-minima

Problem 2

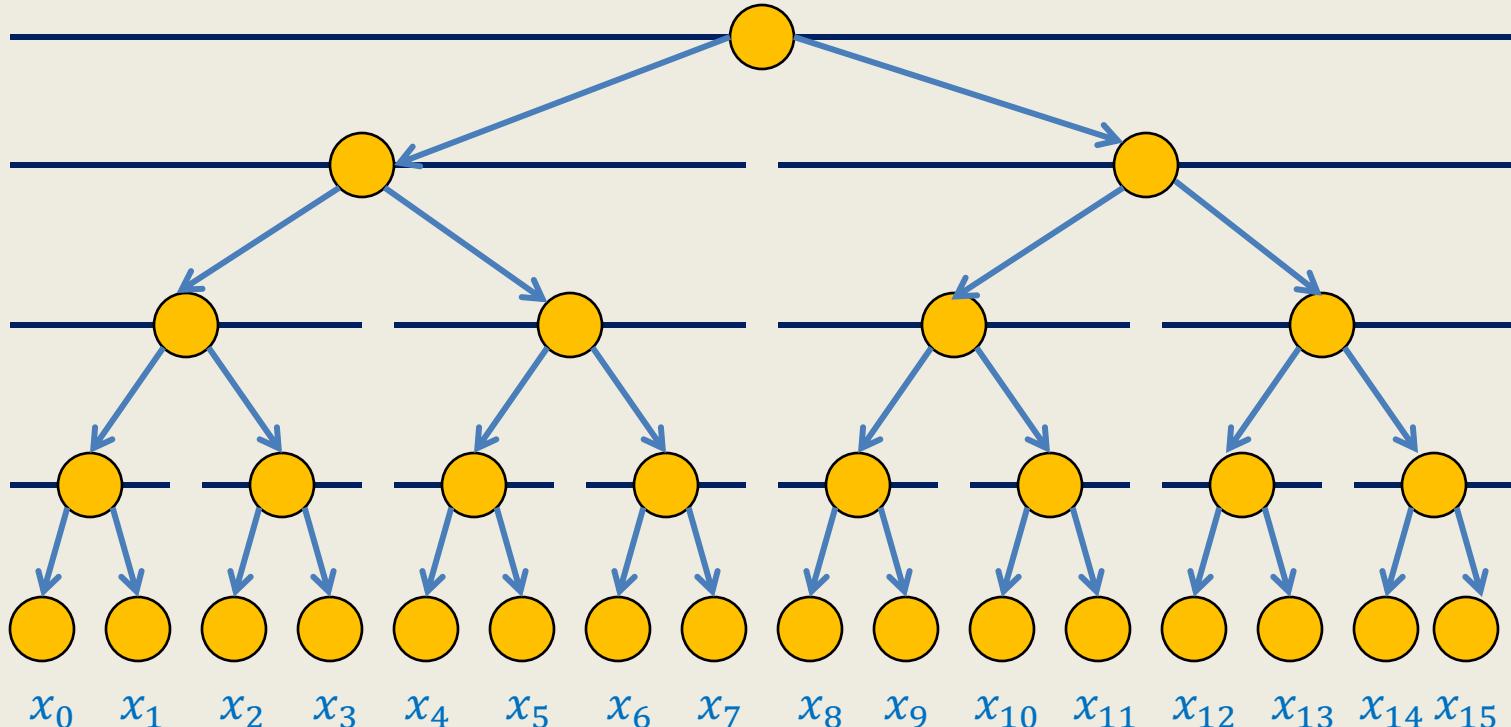
Given an initial sequence $S = \langle x_0, \dots, x_{n-1} \rangle$ of numbers, maintain a compact data structure to perform the following operations efficiently for any $0 \leq i < j < n$.

- **ReportMin(i, j):**
Report the minimum element from $\{x_k \mid \text{for each } i \leq k \leq j\}$
- **Update(i, a):**
 a becomes the new value of x_i .

AIM:

- **$O(n)$** size data structure.
- **ReportMin(i, j)** in **$O(\log n)$** time.
- **Update(i, a)** in **$O(\log n)$** time.

Efficient dynamic range minima



What to store at internal nodes ?

How to perform **ReportMin(i, j)** ?

How to perform **Update(i, a)** ?

Make sincere attempts to solve the problem. We shall discuss it in next lecture.

Data Structures and Algorithms

(ESO207)

Lecture 31

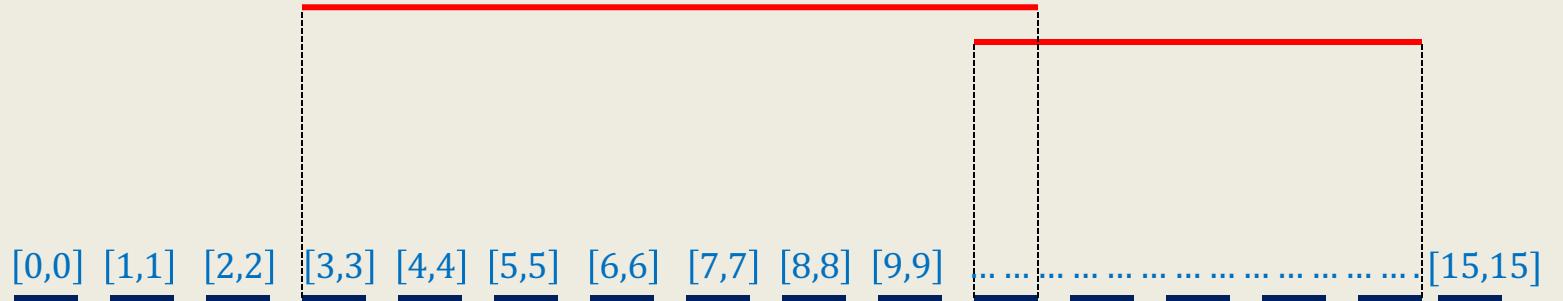
- **Magical applications of Binary trees -II**

RECAP OF LAST LECTURE

Intervals

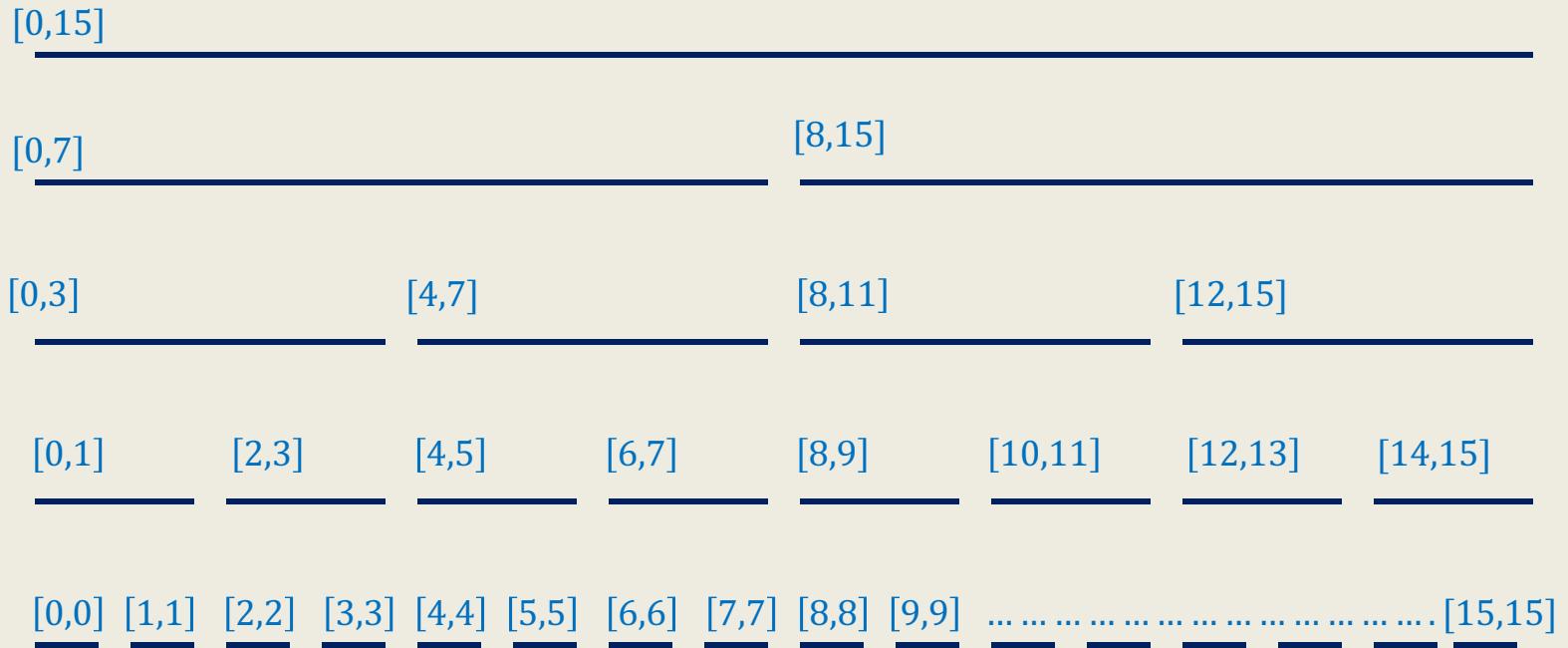
$$\mathbf{S} = \{[\mathbf{i}, \mathbf{j}], 0 \leq i \leq j < n\}$$

Question: Can we have a small set $\mathbf{X} \subset \mathbf{S}$ of **intervals** s.t.
every interval in \mathbf{S} can be expressed as a union of a few intervals from \mathbf{X} ?

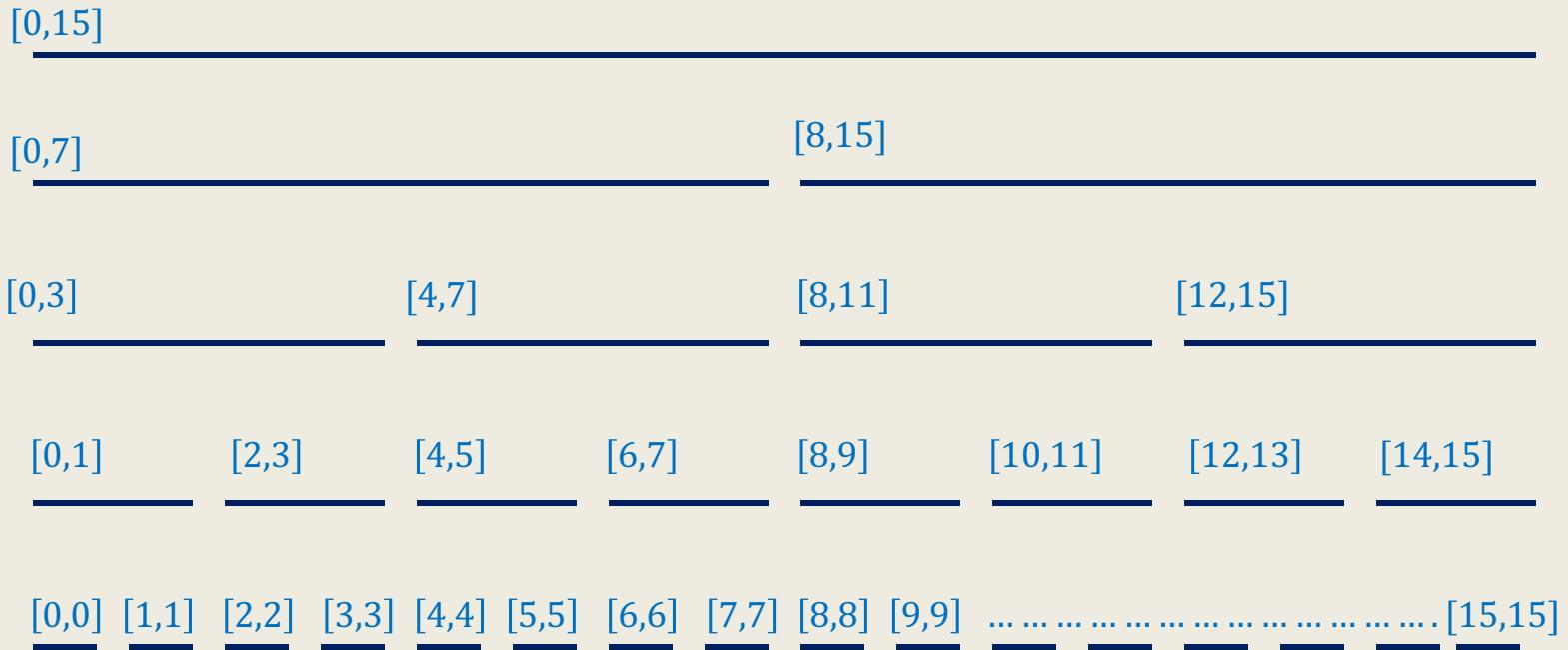


Answer: yes😊

Hierarchy of intervals

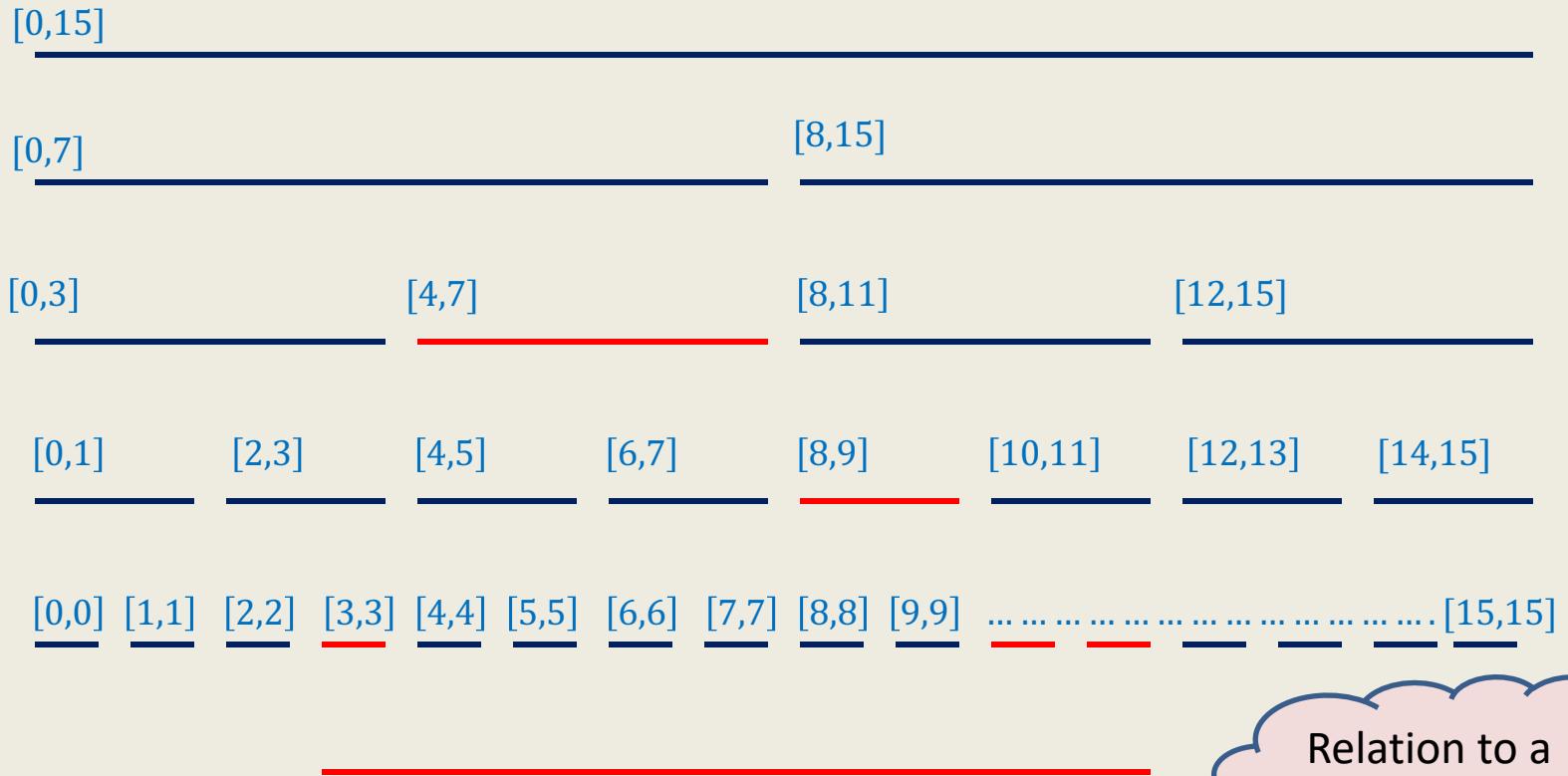


Hierarchy of intervals



Observation: There are $2n$ intervals such that
any interval $[i, j]$ can be expressed as **union** of $O(\log n)$ basic intervals ☺

Hierarchy of intervals



Observation: There are $2n$ intervals such that

any interval $[i, j]$ can be expressed as union of $O(\log n)$ basic intervals 😊

Relation to a
sequence ?

Which data structure emerges ?

[0,15]

[0,7]

[8,15]

[0,3]

[4,7]

[8,11]

[12,15]

[0,1]

[2,3]

[4,5]

[6,7]

[8,9]

[10,11]

[12,13]

[14,15]

[0,0]

[1,1]

[2,2]

[3,3]

[4,4]

[5,5]

[6,6]

[7,7]

[8,8]

[9,9]

.....

.....

.....

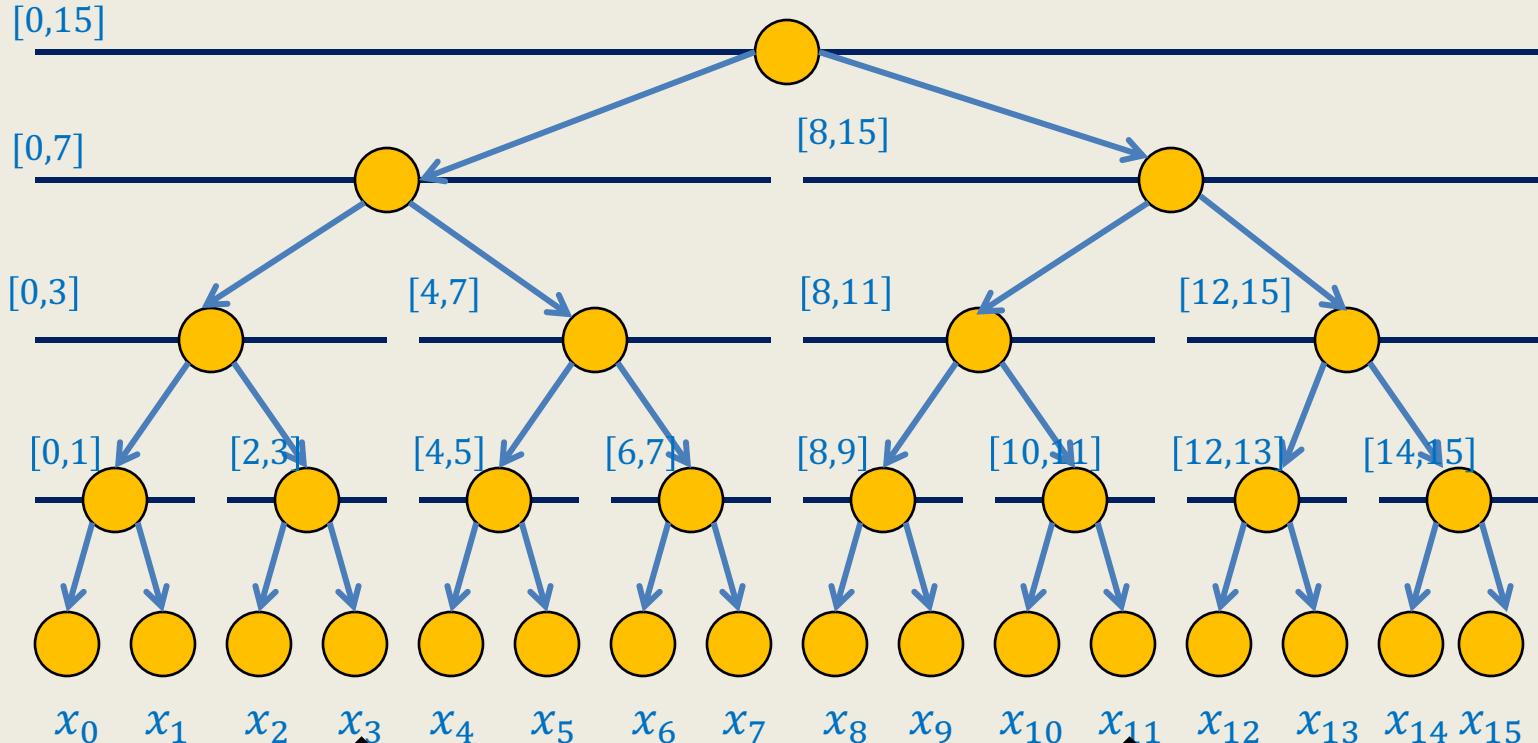
.....

[15,15]

Sequence

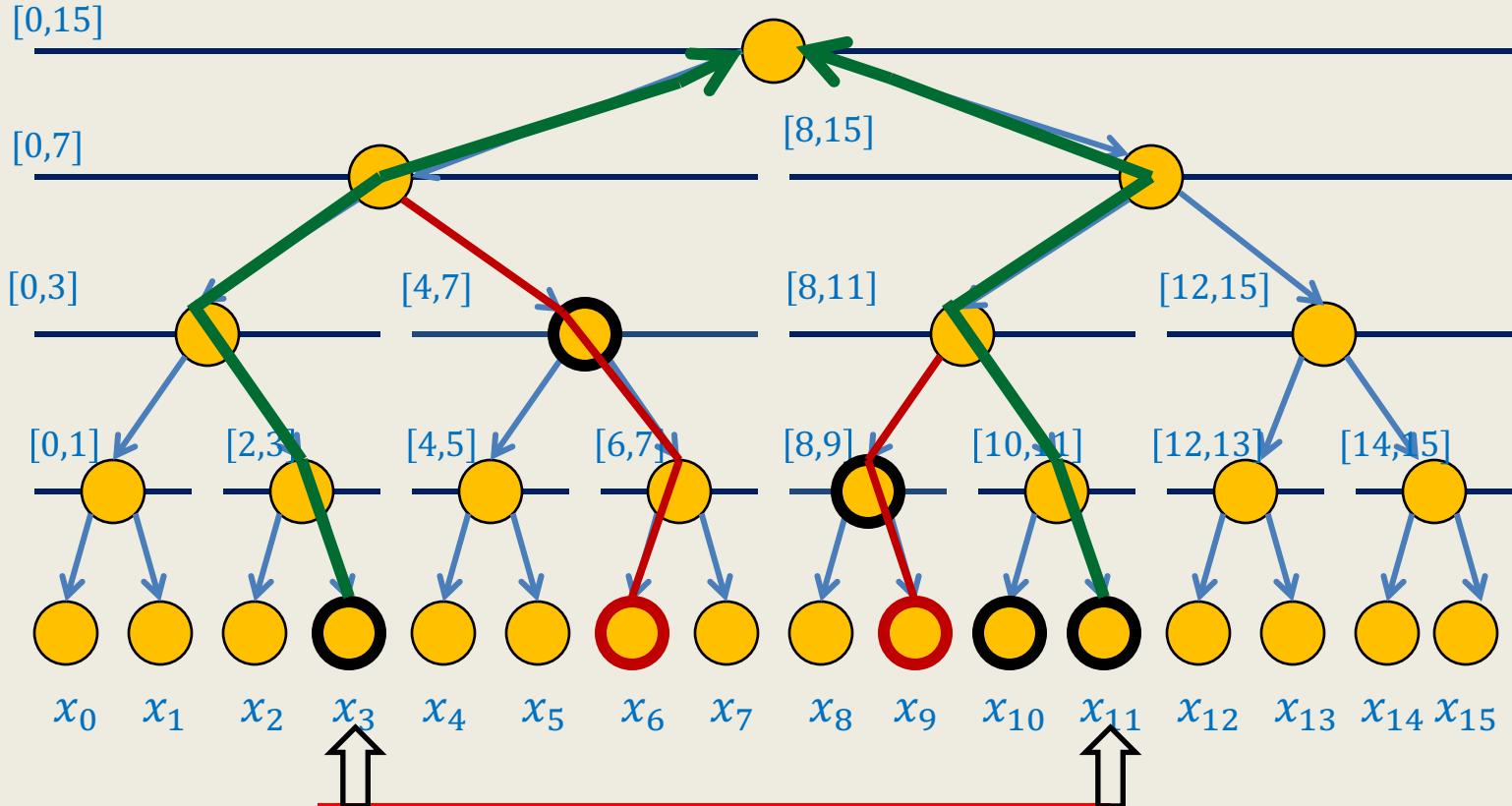
$x_0 \quad x_1 \quad x_2 \quad x_3 \quad x_4 \quad x_5 \quad x_6 \quad x_7 \quad x_8 \quad x_9 \quad x_{10} \quad x_{11} \quad x_{12} \quad x_{13} \quad x_{14} \quad x_{15}$

A Binary tree



What value should you keep
in **internal nodes** ?

How to perform Operation on an interval ?



How to perform Operation on an interval ?

Problem 2

Dynamic Range-minima

Dynamic Range Minima Problem

Given an initial sequence $S = \langle x_0, \dots, x_{n-1} \rangle$ of numbers, maintain a compact data structure to perform the following operations efficiently for any $0 \leq i < j < n$.

- **ReportMin(i, j):**
Report the minimum element from $\{x_k \mid \text{for each } i \leq k \leq j\}$
 - **Update(i, a):**
 a becomes the new value of x_i .
-

Example:

Let the initial sequence be $S = \langle 14, 12, 3, 49, 4, 21, 322, -40 \rangle$

ReportMin(1, 5) returns **3**

ReportMin(0, 3) returns **3**

Update(2, 19) update S to $\langle 14, 12, 19, 49, 4, 21, 322, -40 \rangle$

ReportMin(1, 5) returns **4**

ReportMin(0, 3) returns **12**

Dynamic Range Minima Problem

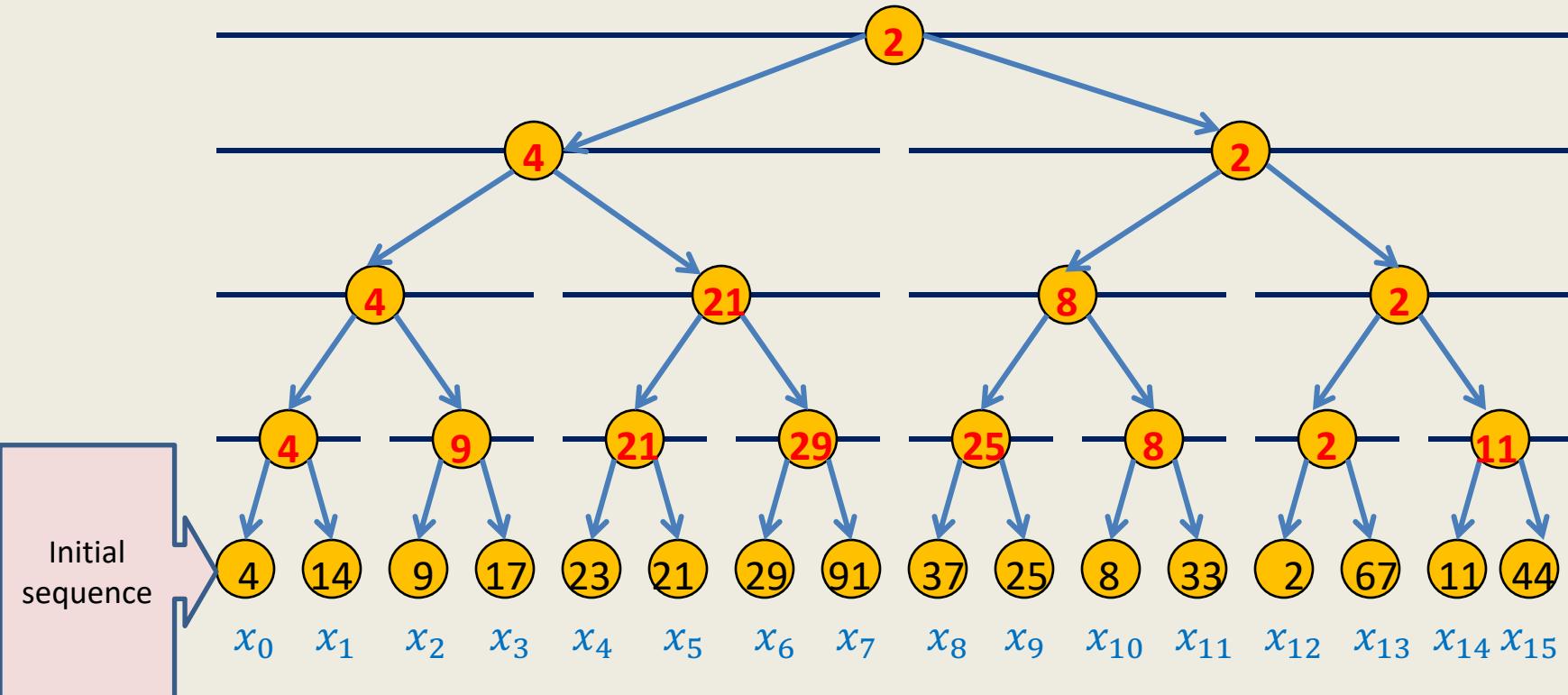
Given an initial sequence $S = \langle x_0, \dots, x_{n-1} \rangle$ of numbers, maintain a compact data structure to perform the following operations efficiently for any $0 \leq i < j < n$.

- **ReportMin(i, j):**
Report the minimum element from $\{x_k \mid \text{for each } i \leq k \leq j\}$
 - **Update(i, a):**
 a becomes the new value of x_i .
-

AIM:

- $O(n)$ size data structure.
- ReportMin(i, j) in $O(\log n)$ time.
- Update(i, a) in $O(\log n)$ time.

Data structure for dynamic range minima

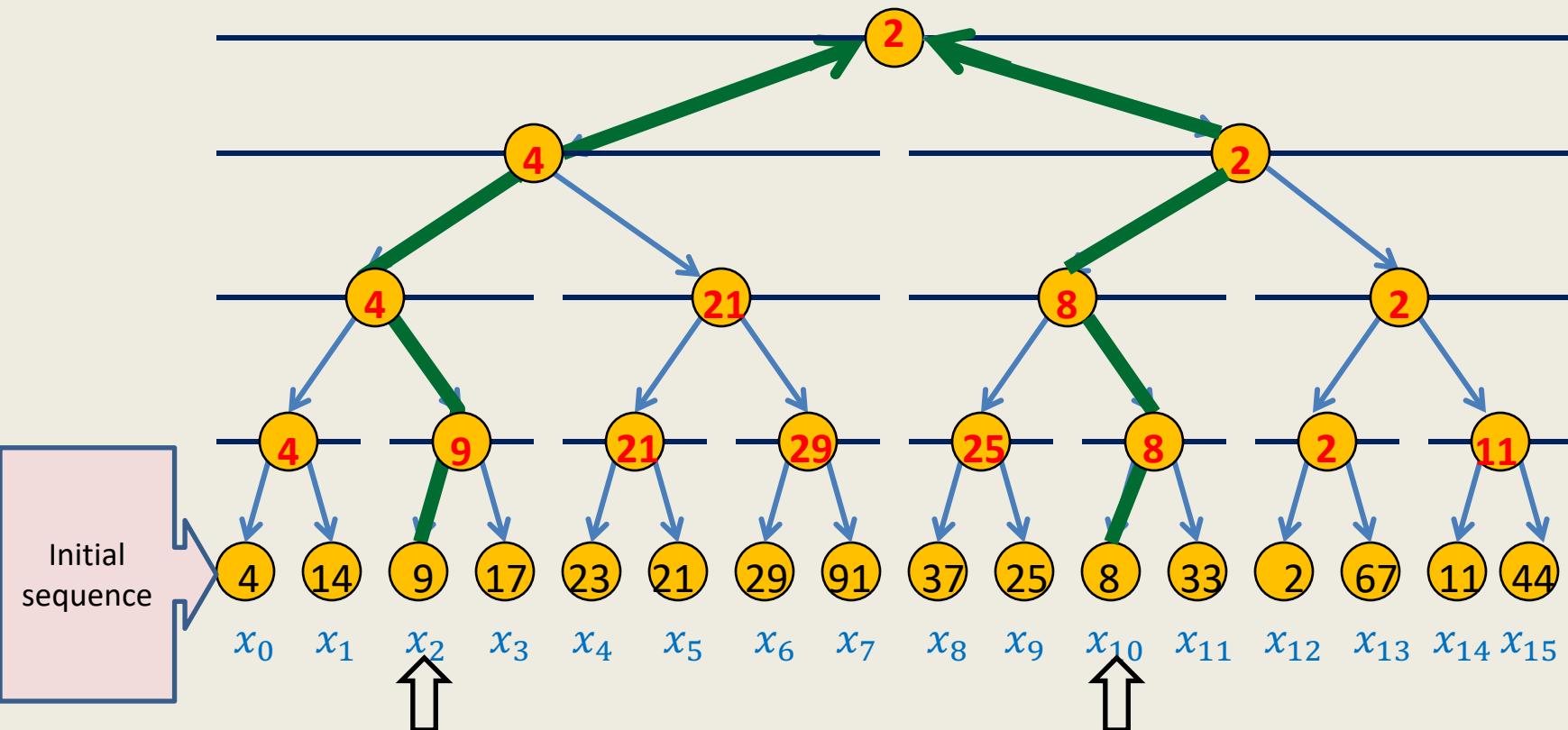


Question: What should be stored in an internal node v ?

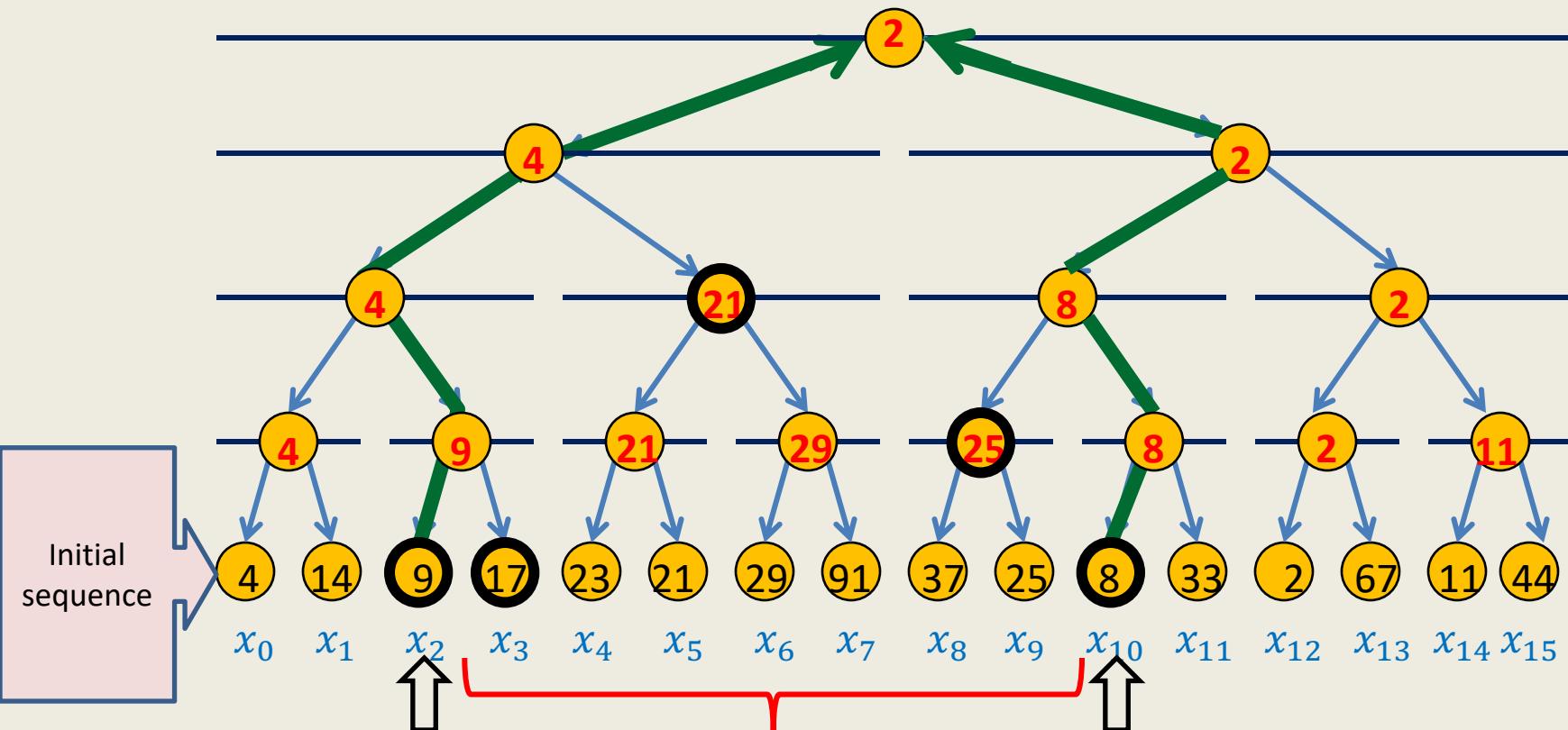
Answer:

minimum value

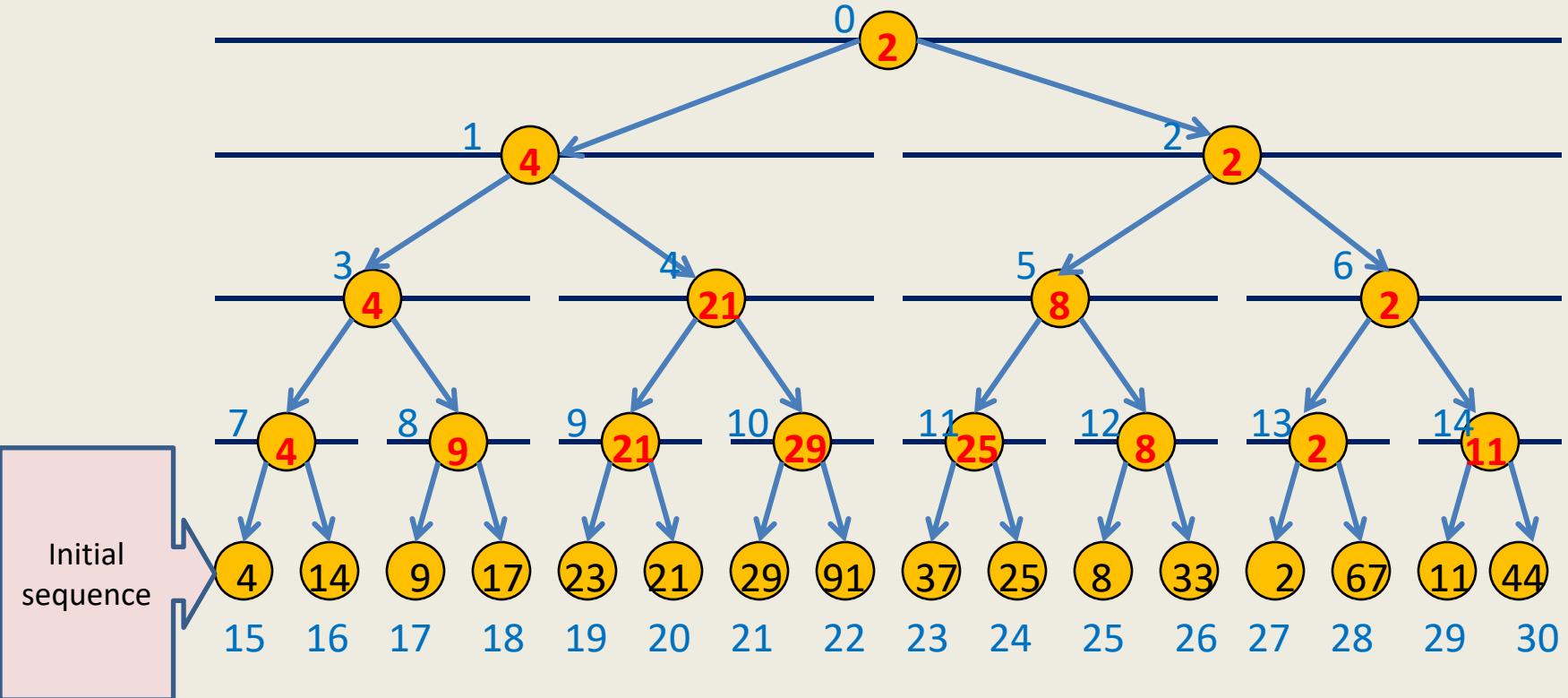
Data structure for dynamic range minima



Data structure for dynamic range minima



Data structure for dynamic range minima



Data structure: An array A of size $2n-1$.

Copy the sequence $S = \langle x_0, \dots, x_{n-1} \rangle$ into $A[n-1] \dots A[2n-2]$

Leaf node corresponding to $x_i = A[(n-1) + i]$

How to check if a node is left child or right child of its parent ?

(if index of the node is odd, then the node is left child, else the node is right child)

Update(*i, a*)

Update(*i, a*)

i \leftarrow (*n - 1*) + *i* ;

A[*i*] \leftarrow *a* ;

i \leftarrow [(*i - 1*)/2] ;

While(??)

{

Homework

At the end of the lecture slides, an **incorrect** pseudocode is given for Update(*i, a*).
It is followed by a **correct** one.

Ponder over it ☺

}

Report-Min(*i,j*)

Report-Min(*i,j*)

```
i < (n - 1) + i ;  
j < (n - 1) + j ;  
min <- A(i) ;  
If (j > i)  
{     If (A(j) < min)   min <- A(j) ;  
    While(  $\lfloor (\iota - 1)/2 \rfloor \neq \lfloor (j - 1)/2 \rfloor$  )  
    {  
        If(   i%2=1 and A(i + 1) < min   )   min <- ;  
        If(   j%2=0 and A(j - 1) < min   )   min <- ;  
        i < ;  
        j < ;  
    }  
}  
return min ;
```

Another interesting problem on sequences

Practice Problem

Given an initial sequence $S = \langle x_0, \dots, x_{n-1} \rangle$ of n numbers,
maintain a compact data structure to perform the following operations efficiently :

- **Report_min(i, j):**

Report the minimum element from $\{x_i, \dots, x_j\}$.

- **Multi-Increment(i, j, Δ):**

Add Δ to each x_k for each $i \leq k \leq j$

Example:

Let the initial sequence be $S = \langle 14, 12, 3, 12, 111, 51, 321, -40 \rangle$

- **Report_min(1, 4):**

returns 3

- **Multi-Increment(2, 6, 10):**

S becomes $\langle 14, 12, 13, 22, 121, 61, 331, -40 \rangle$

- **Report_min(1, 4):**

returns 12

An **challenging problem** on sequences

**For summer vacation
(not for the exam)**

* Problem

Given an initial sequence $S = \langle x_0, \dots, x_{n-1} \rangle$ of n numbers,
maintain a compact data structure to perform the following operations efficiently :

- **Report_min(i, j):**

Report the minimum element from $\{x_i, \dots, x_j\}$.

- **Multi-Increment(i, j, Δ):**

Add Δ to each x_k for each $i \leq k \leq j$

- **Rotate(i, j):**

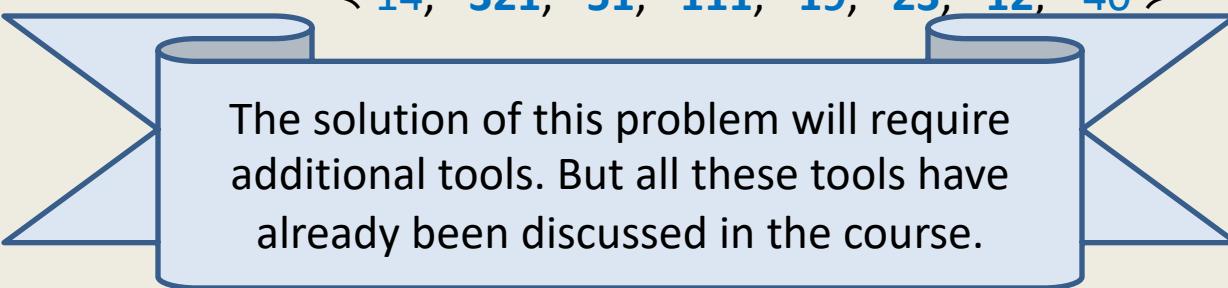
$x_i \leftrightarrow x_j, x_{i+1} \leftrightarrow x_{j-1}, \dots$

Example:

Let the initial sequence be $S = \langle 14, 12, 23, 19, 111, 51, 321, -40 \rangle$

After **Rotate(1,6)**, S becomes

$\langle 14, 321, 51, 111, 19, 23, 12, -40 \rangle$



The solution of this problem will require additional tools. But all these tools have already been discussed in the course.

Problem 4

A data structure for sets

Sets under operations

Given: a collection of n singleton sets $\{0\}, \{1\}, \{2\}, \dots \{n - 1\}$

Aim: a compact data structure to perform

- **Union(i, j):**
Unite the two sets containing i and j .
 - **Same_sets(i, j):**
Determine if i and j belong to the same set.
-

Trivial Solution 1

Keep an array **Label[]** such that

$\text{Label}[i] = \text{Label}[j]$ if and only if i and j belong to the same set.

- **Same_sets(i, j):** $O(1)$ time
- check if $\text{Label}[i] = \text{Label}[j]$?
- **Union(i, j):** $O(n)$ time
- For each $0 \leq k < n$
- if ($\text{Label}[k] = \text{Label}[i]$) $\text{Label}[k] \leftarrow \text{Label}[j]$)

Sets under operations

Given: a collection of n singleton sets $\{0\}, \{1\}, \{2\}, \dots \{n - 1\}$

Aim: a compact data structure to perform

- **Union(i, j):**
Unite the two sets containing i and j .
 - **Same_sets(i, j):**
Determine if i and j belong to the same set.
-

Trivial Solution 2

Treat the problem as a graph problem: ??

Connected component

- $V = \{0, \dots, n - 1\}$, $E =$ empty set initially.
- A set \Leftrightarrow
- Keep array **Label[]** such that **Label[i] = Label[j]** iff i and j belong to the same component.



Union(i, j) :

O(n) time

add an edge (i, j) and
recompute connected components using **BFS/DFS**.

Sets under operations

Given: a collection of n singleton sets $\{0\}, \{1\}, \{2\}, \dots \{n - 1\}$

Aim: a compact data structure to perform

- **Union(i, j):**
Unite the two sets containing i and j .
 - **Same_sets(i, j):**
Determine if i and j belong to the same set.
-

Efficient solution:

- A data structure which supports each operation in $O(\log n)$ time.
- **An additional heuristic**
→ time complexity of an operation :

We shall discuss it in the next
lecture.

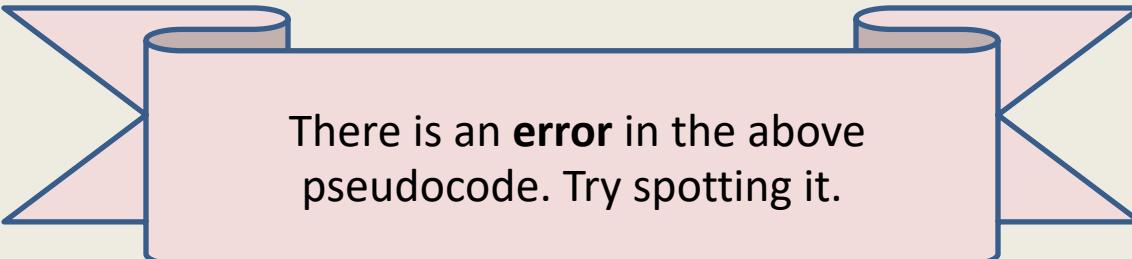
Homework

For **Dynamic Range-minima** problem

Update(*i, a*)

Update(*i, a*)

```
i < (n - 1) + i ;  
A[i] <= a ;  
i <= [(i - 1)/2] ;  
While(           i ≥ 0           )  
{  
    If(a < A[i])  A[i] <= a;  
    i <= [(i - 1)/2] ;  
}
```

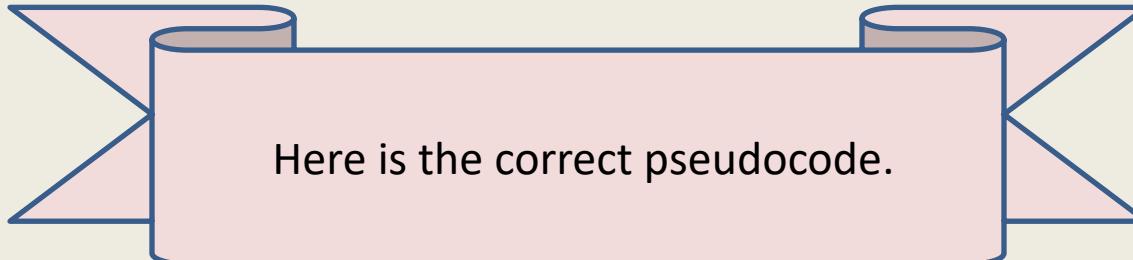


There is an **error** in the above pseudocode. Try spotting it.

Update(i, a)

Update(i, a)

```
i ← (n - 1) + i ;  
A[i] ← a ;  
i ← ⌊(i - 1)/2⌋ ;  
While(           i ≥ 0          )  
{  
    If( A[2i + 1] < A[2i + 2])  
        A[i] ← A[2i + 1]  
    else  
        A[i] ← A[2i + 2] ;  
    i ← ⌊(i - 1)/2⌋ ;  
}
```



Here is the correct pseudocode.

Data Structures and Algorithms

(ESO207)

Lecture 32

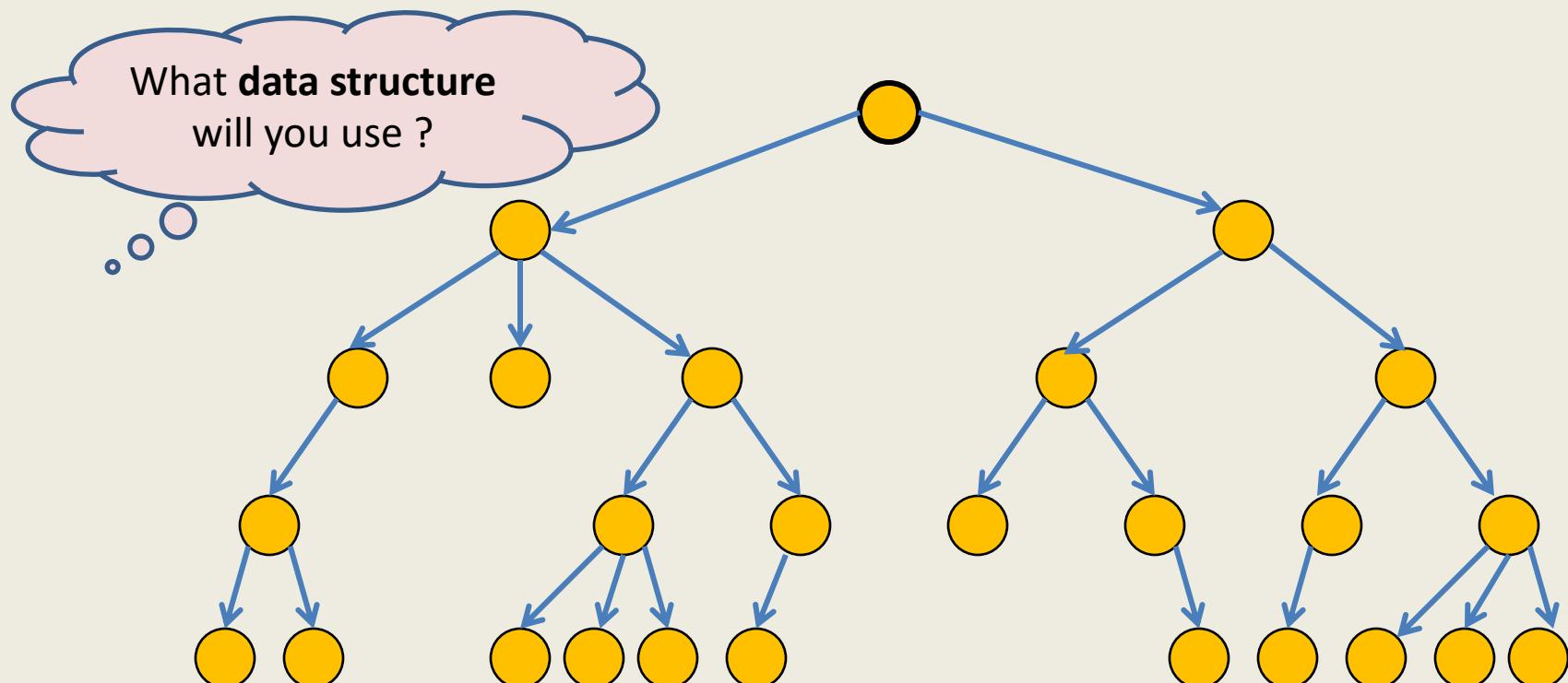
- **Magical application of binary trees – III**

Data structure for sets

Rooted tree

Revisiting and extending

A typical rooted tree we studied



Definition we gave:

Every vertex, except **root**, has exactly one incoming edge and has a path **from** the root.

Examples:

Binary search trees,

DFS tree,

BFS tree.

A typical rooted tree we studied

Question: what data structure can be used for representing a rooted tree ?

Answer:

Data structure 1:

- Each node stores a list of its children.
- To access the tree, we keep a pointer to the root node.
(there is no way to access any node (other than root) directly in this data structure)

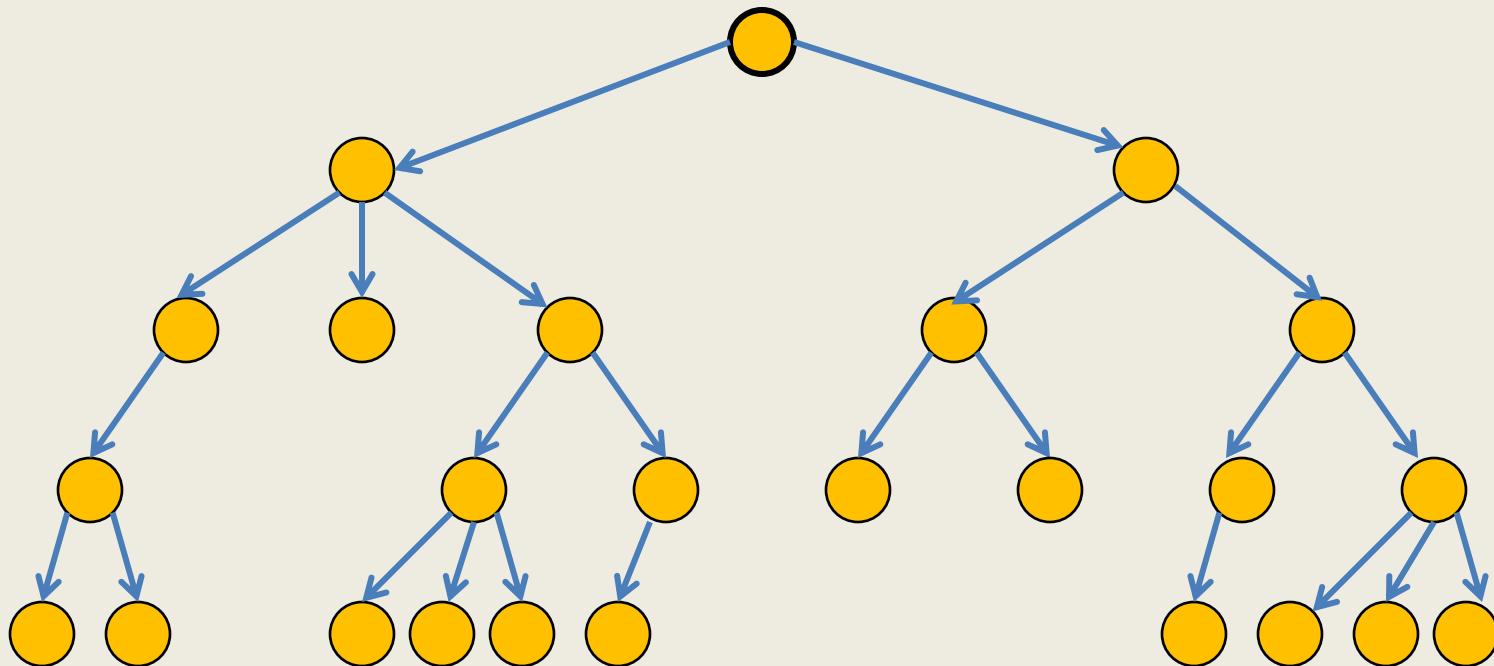
Data structure 2: (If nodes are labeled in a contiguous range [0..n-1])

rooted tree becomes an instance of a **directed graph**.

So we may use **adjacency list** representation.

Advantage: We can access each node directly.

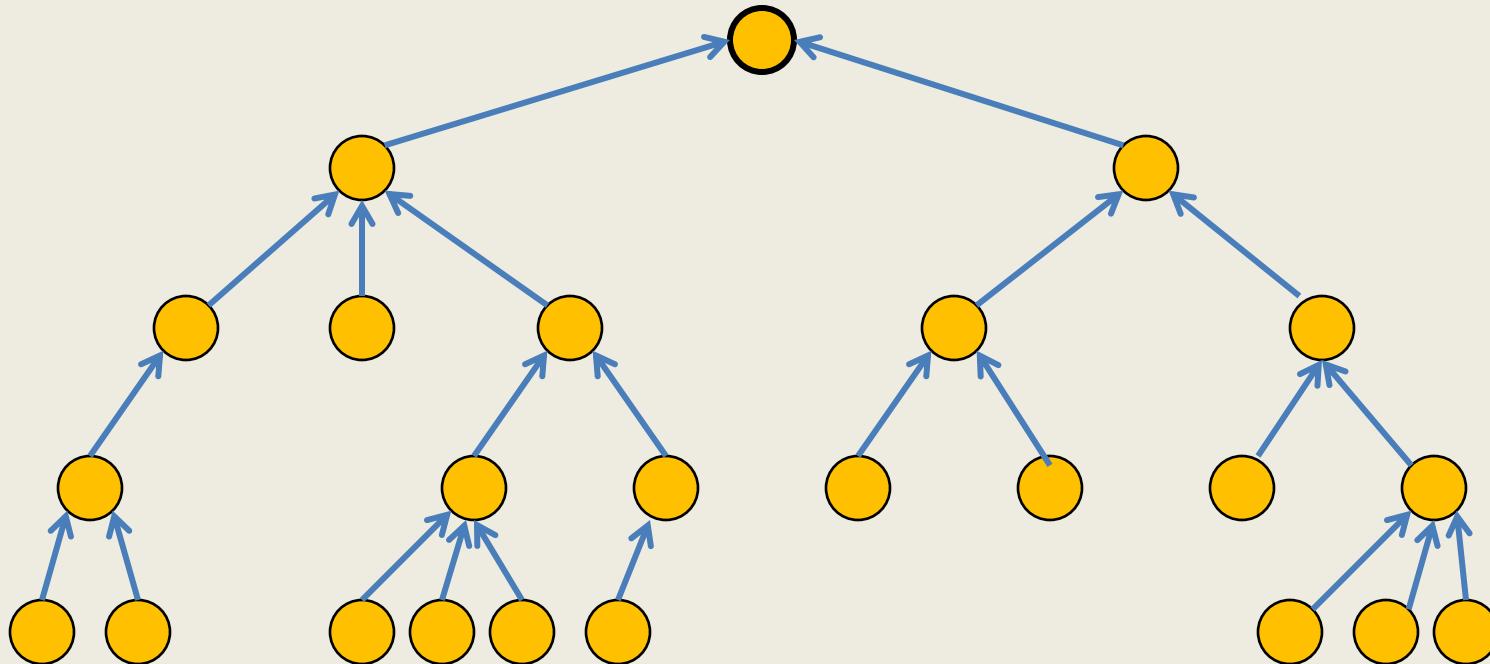
Extending the definition of rooted tree



Extended Definition:

Type 1: Every vertex, except **root**, has exactly one incoming edge and has a path **from** the root.

Extending the definition of rooted tree



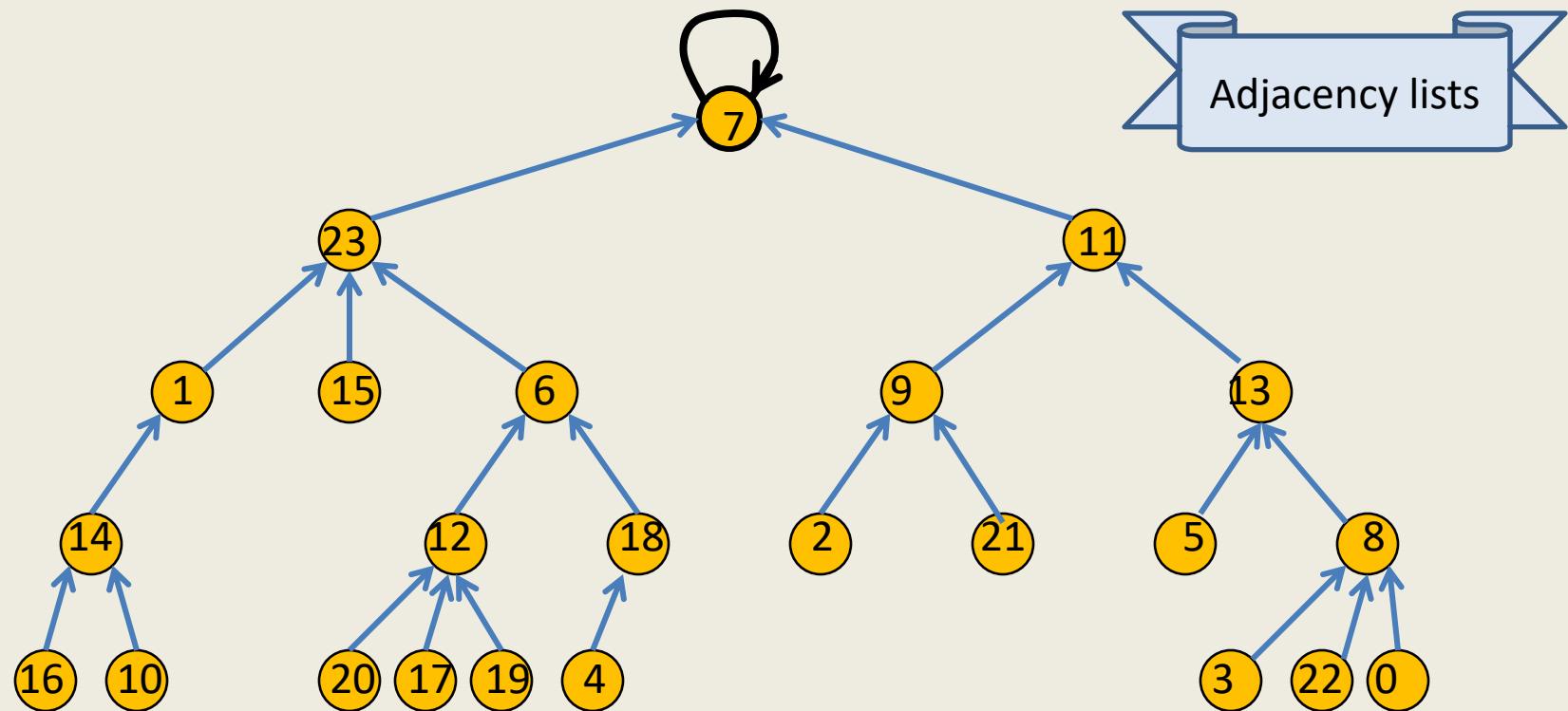
Extended Definition:

Type 1: Every vertex, except **root**, has exactly one incoming edge and has a path **from** the root.

OR

Type 2: Every vertex, except root, has exactly one outgoing edge and has a path **to** the root.

Data structure for rooted tree of type 2



If nodes are labeled in a contiguous range [0..n – 1],
there is even simpler and more compact data structure

Guess ??

Parent	8	23	9	8	18	13	23	7	13	11	14	7	6	11	1	23	14	12	6	12	12	9	8	7
	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23

Application of rooted tree of **type 2**

Maintaining sets

Sets under two operations

Given: a collection of n singleton sets $\{0\}, \{1\}, \{2\}, \dots \{n - 1\}$

Aim: a compact data structure to perform

- **Union(i, j):**
Unite the two sets containing i and j .
 - **Same_sets(i, j):**
Determine if i and j belong to the same set.
-

Trivial Solution

Treat the problem as a graph problem: **Connected component**

- $V = \{0, \dots, n - 1\}$, $E =$ empty set initially.
- A set \Leftrightarrow
- Keep array **Label[]**



Union(i, j):

if (**Same_sets(i, j)** = false)
add an edge (i, j)

O(n) time

Sets under two operations

Given: a collection of n singleton sets $\{0\}, \{1\}, \{2\}, \dots \{n - 1\}$

Aim: a compact data structure to perform

- **Union(i, j):**
Unite the two sets containing i and j .
 - **Same_sets(i, j):**
Determine if i and j belong to the same set.
-

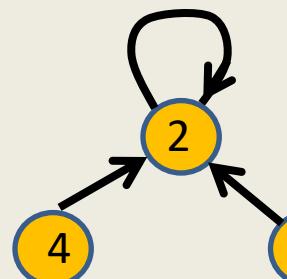
Efficient solution:

- A data structure which supports each operation in $O(\log n)$ time.
- **An additional heuristic**
→ time complexity of an operation : practically $O(1)$.

Data structure for sets

Maintain each set as a rooted tree .

Union(i, j) ?



{2,4,6}

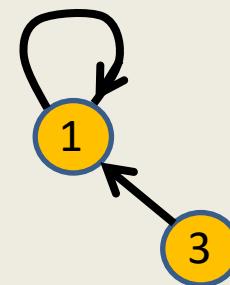


{0}



{5}

SameSet(i, j) ?



{1,3}

Data structure for sets

Maintain each set as a rooted tree .

Question: How to perform operation $\text{Same_sets}(i, j)$?

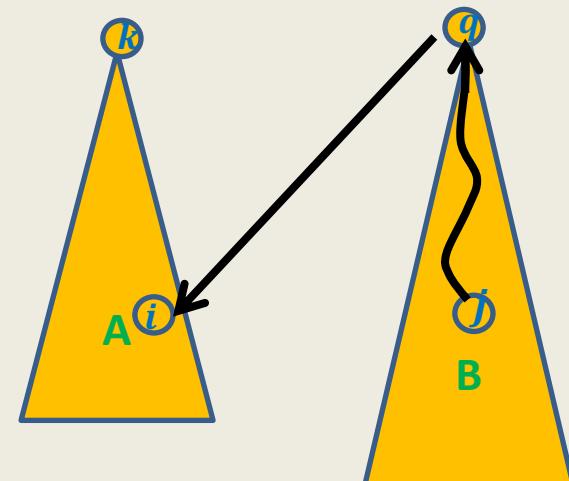
Answer: Determine if i and j belong to the same tree.

→ find root of i

Question: How to perform $\text{Union}(i, j)$?

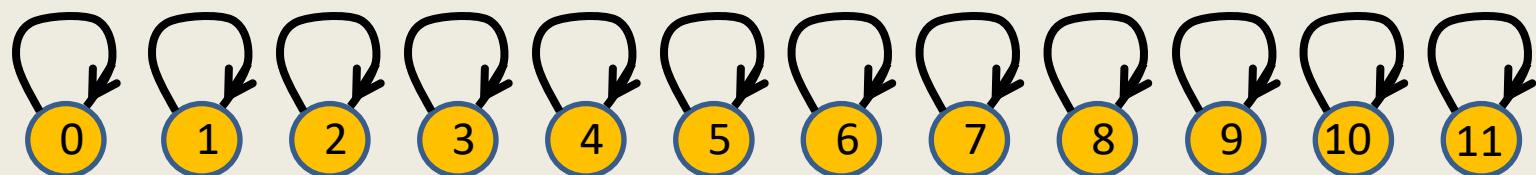
Answer:

- find root of j ; let it be q .
- $\text{Parent}(q) \leftarrow i$.



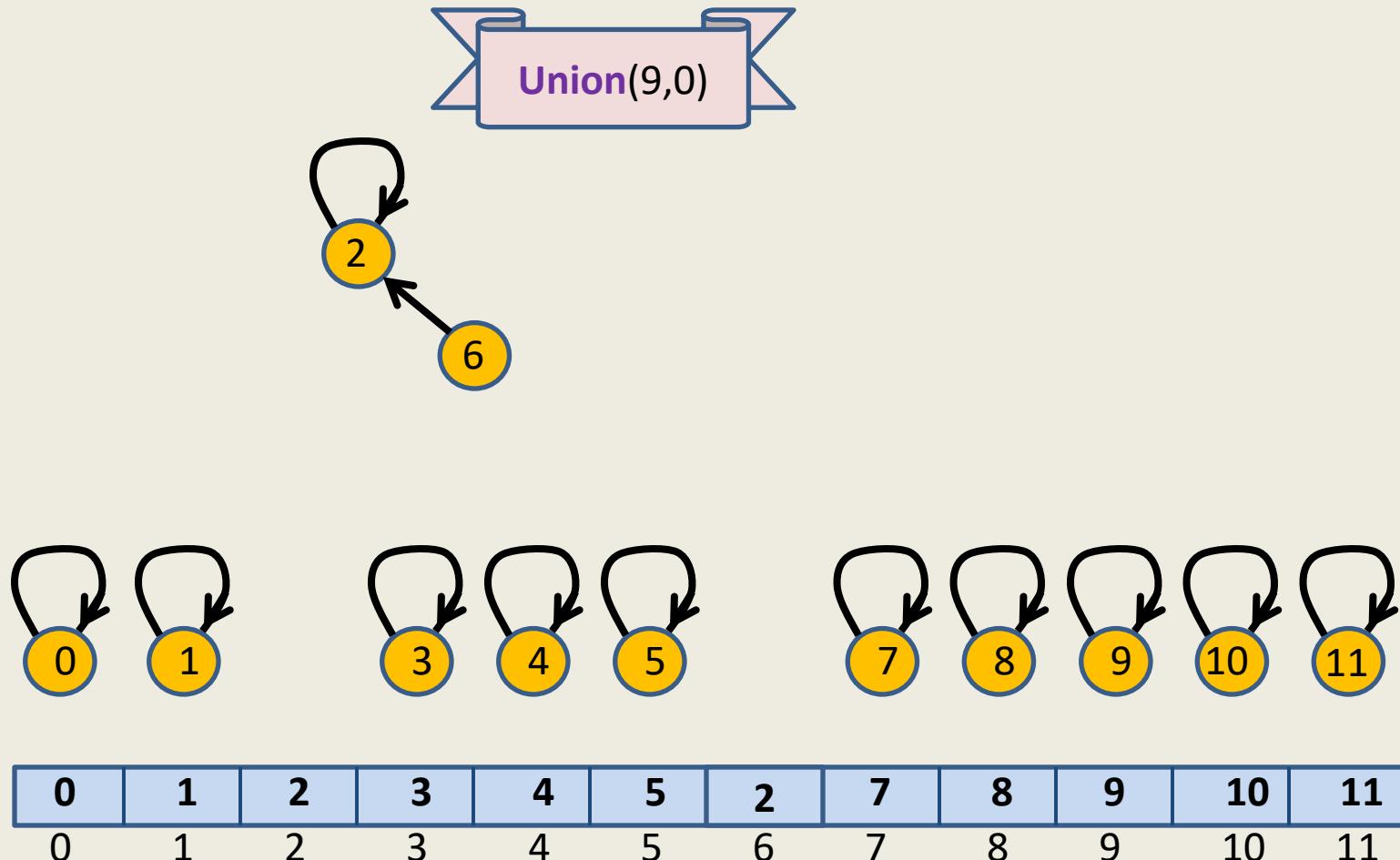
A rooted tree as a data structure for sets

Union(2,6)

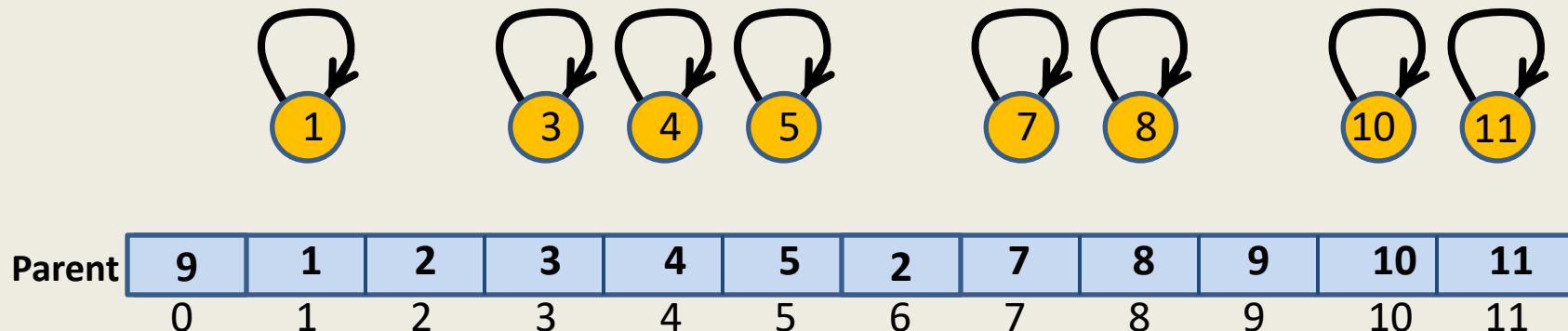
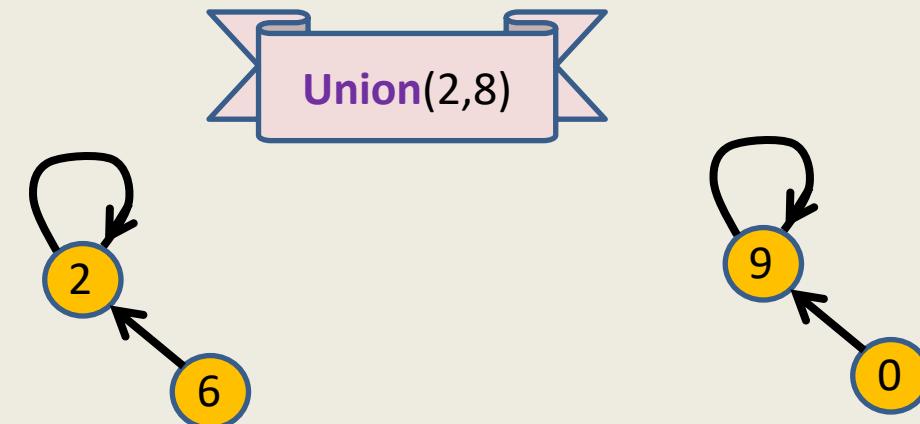


Parent	0	1	2	3	4	5	6	7	8	9	10	11
	0	1	2	3	4	5	6	7	8	9	10	11

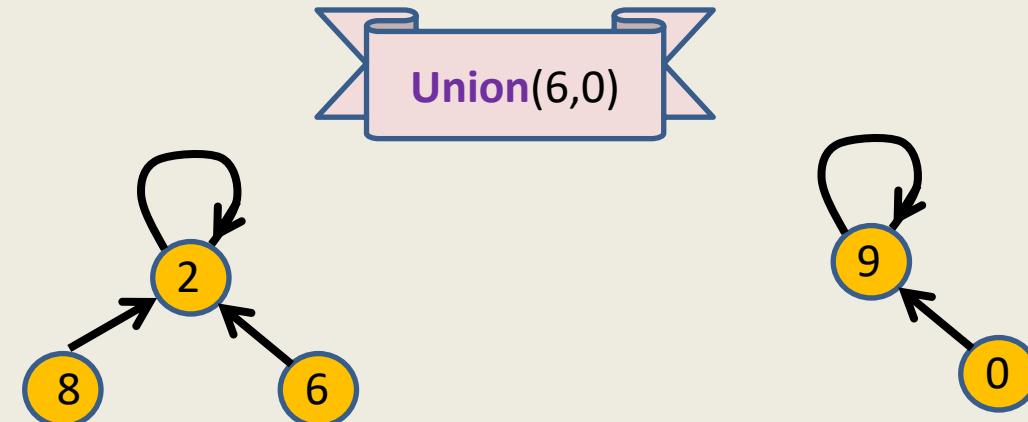
A rooted tree as a data structure for sets



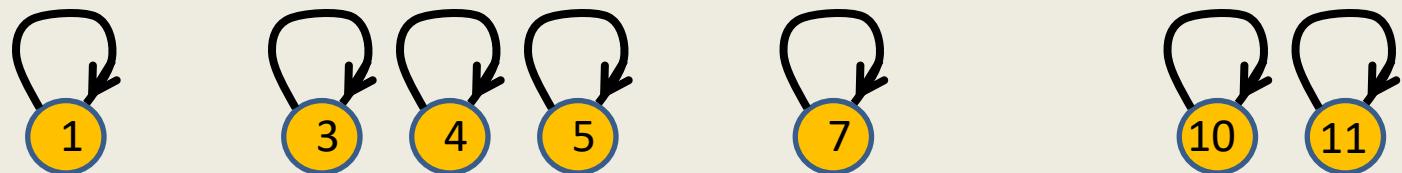
A rooted tree as a data structure for sets



A rooted tree as a data structure for sets

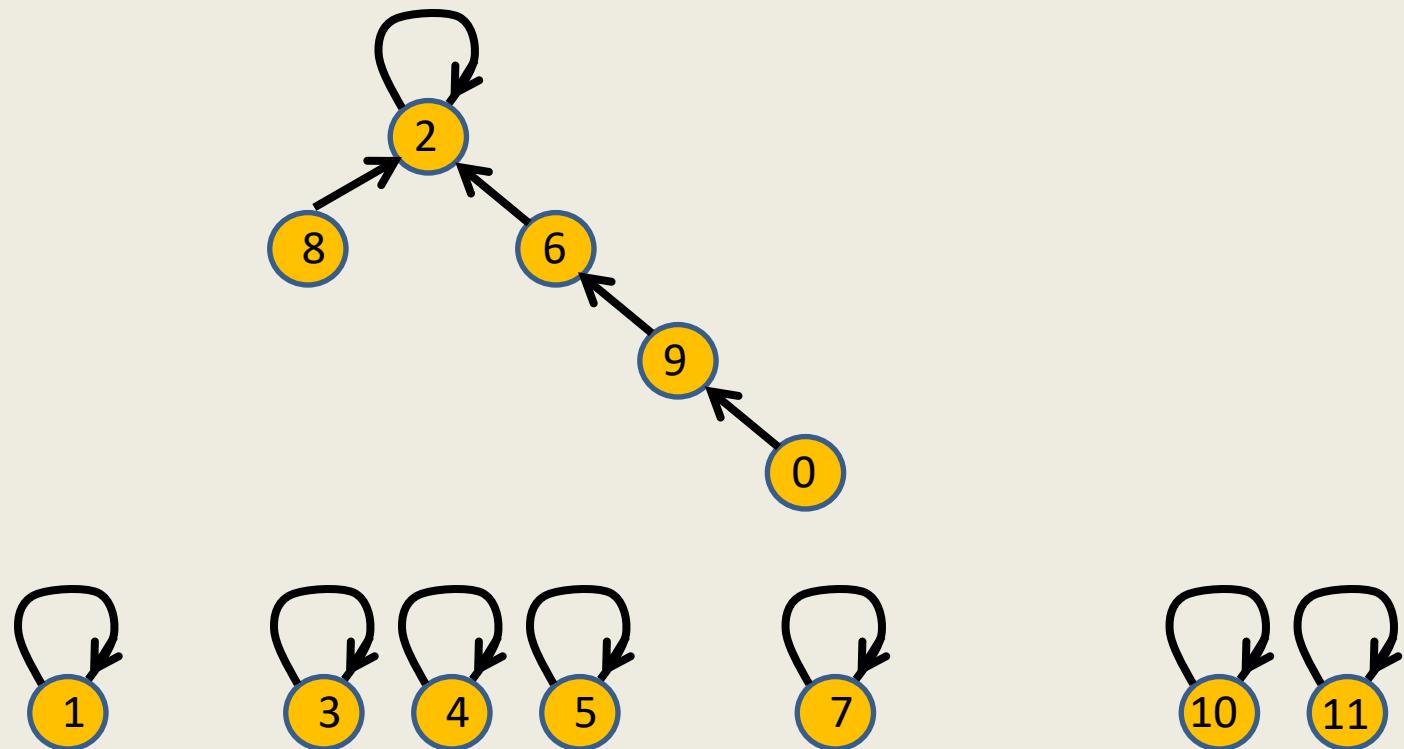


Union(6,0)



Parent	9	1	2	3	4	5	2	7	8	9	10	11
	0	1	2	3	4	5	6	7	8	9	10	11

A rooted tree as a data structure for sets



Parent	9	1	2	3	4	5	2	7	2	6	10	11
	0	1	2	3	4	5	6	7	8	9	10	11

Pseudocode for Union and SameSet()

Find(*i*) // subroutine for finding the root of the tree containing *i*

```
If (Parent(i) = i)    return i ;  
else return Find(Parent(i));
```

SameSet(*i, j*)

```
k  $\leftarrow$  Find(i);  
l  $\leftarrow$  Find(j);  
If (k = l)    return true else return false
```

Union(*i, j*)

```
k  $\leftarrow$  Find(j);  
Parent(k)  $\leftarrow$  i;
```

Observation: Time complexity of **Union(*i, j*)** as well as **Same_sets(*i, j*)** is governed by the time complexity of **Find(*i*)** and **Find(*j*)**.

Question: What is time complexity of **Find(*i*)** ?

Answer: **depth** of the node *i* in the tree containing *i*.

Time complexity of $\text{Find}(i)$

$\text{Union}(0,1)$

$\text{Union}(1,2)$

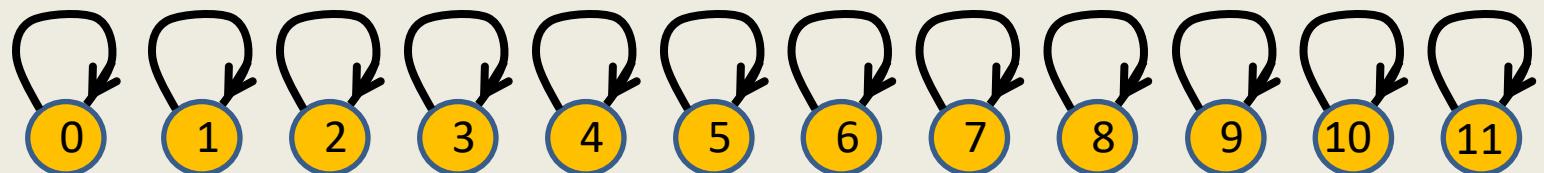
$\text{Union}(2,3)$

...

$\text{Union}(9,10)$

$\text{Union}(10,11)$

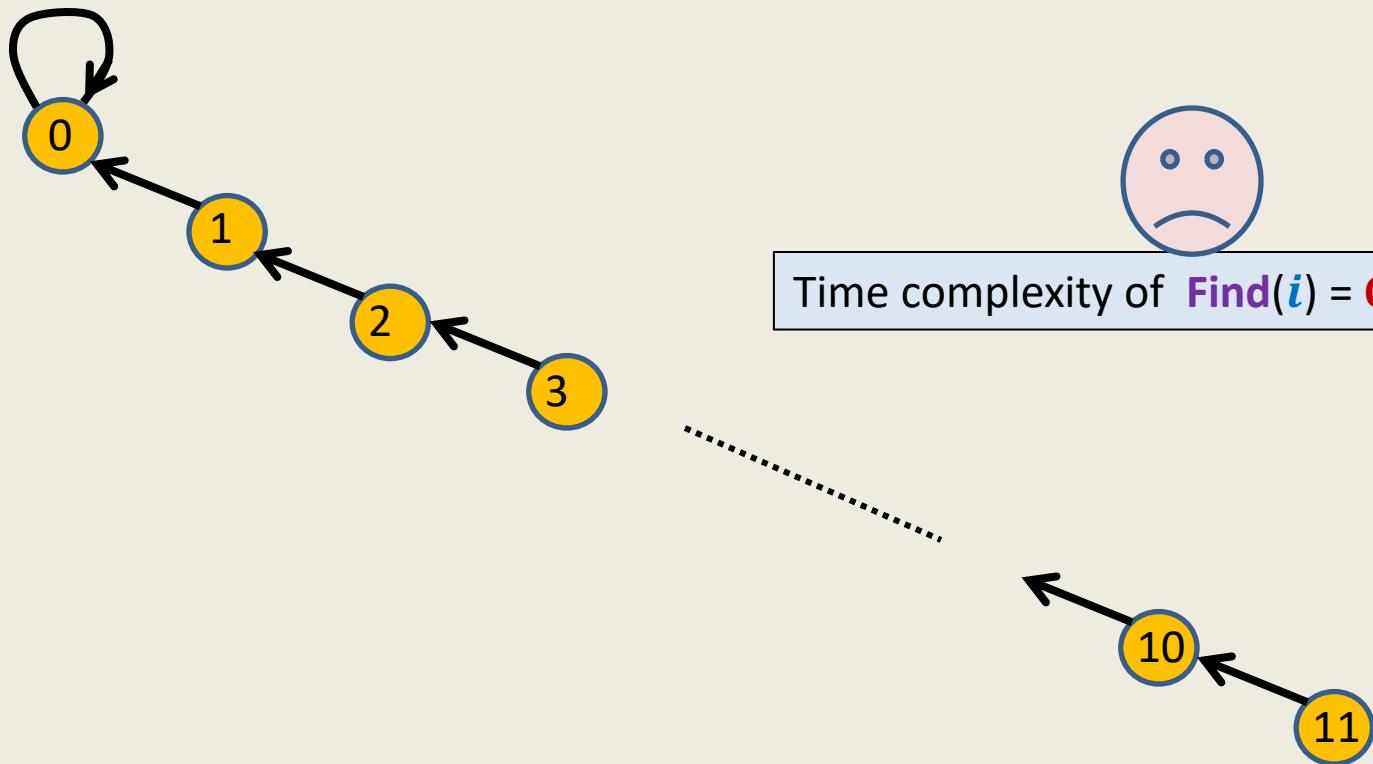
What will be the rooted tree structures
after these union operations ?



Parent	0	1	2	3	4	5	6	7	8	9	10	11
	0	1	2	3	4	5	6	7	8	9	10	11

Time complexity of $\text{Find}(i)$

$\text{Union}(0,1)$
 $\text{Union}(1,2)$
 $\text{Union}(2,3)$
...
 $\text{Union}(9,10)$
 $\text{Union}(10,11)$

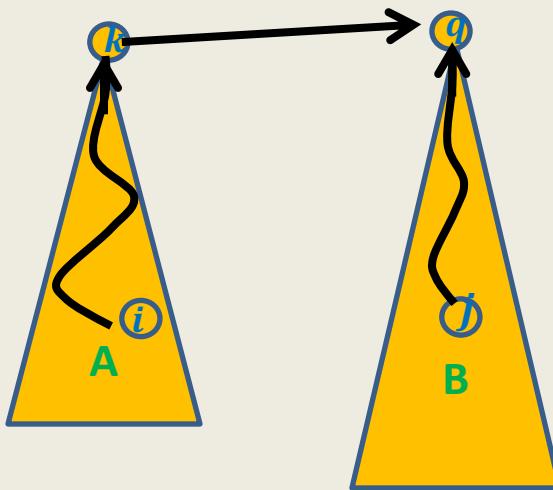


Parent	0	0	1	2	3	4	5	6	7	8	9	10
	0	1	2	3	4	5	6	7	8	9	10	11

Improving the time complexity of Find(*i*)

Heuristic 1: Union by size

Improving the Time complexity



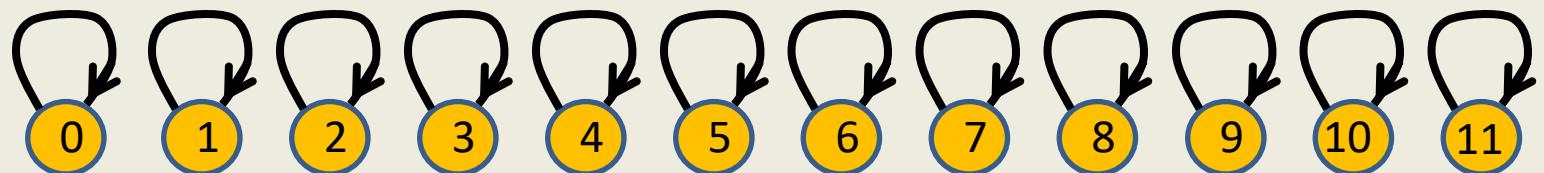
Key idea: Change the **union(*i,j*)** .

While doing **union(*i,j*)**, hook the **smaller size tree to the root of the bigger size tree**.

For this purpose, keep an array **size[0,..,n-1]**

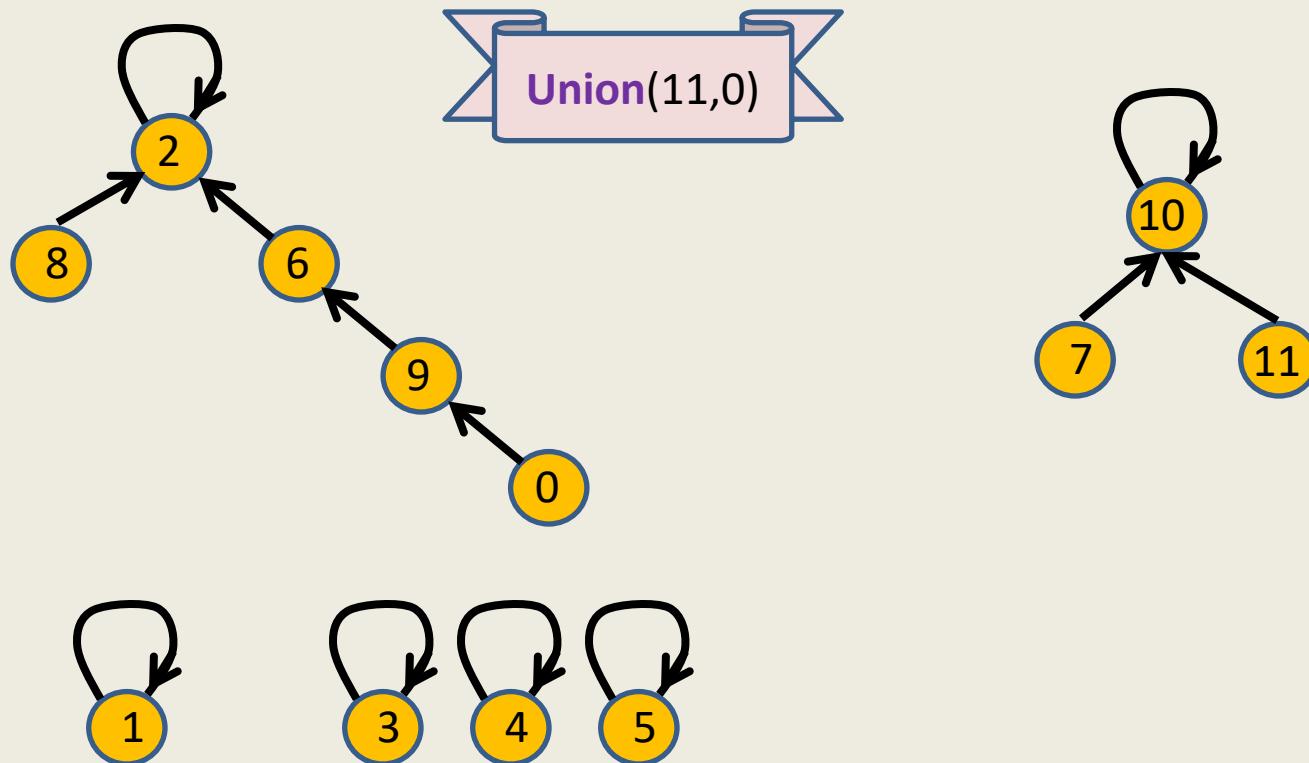
size[*i*] = number of nodes in the tree containing *i*
(if *i* is a **root** and zero otherwise)

Efficient data structure for sets



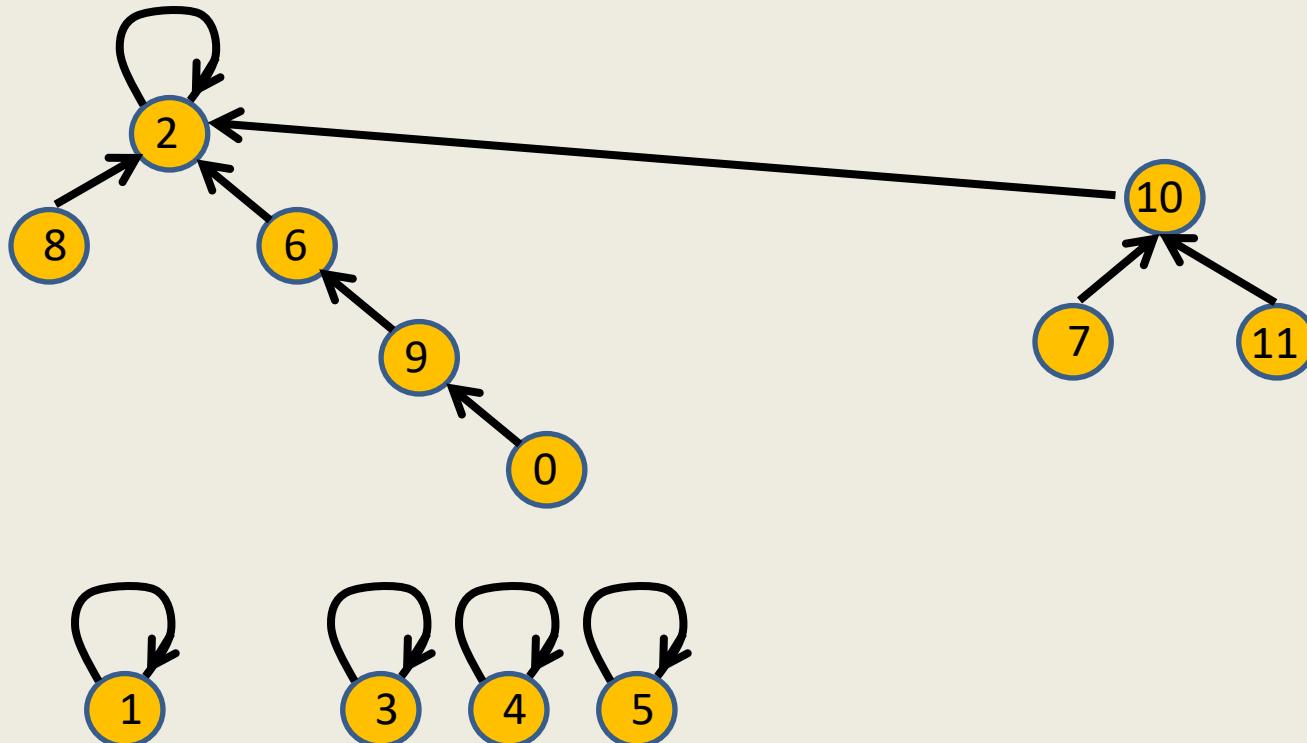
Parent	0	1	2	3	4	5	6	7	8	9	10	11
size	1	1	1	1	1	1	1	1	1	1	1	1

Efficient data structure for sets



Parent	9	1	2	3	4	5	2	10	2	6	10	10
size	0	1	2	3	4	5	6	7	8	9	10	11
size	0	1	5	1	1	1	0	0	0	0	3	0

Efficient data structure for sets



Parent	9	1	2	3	4	5	2	10	2	6	10	10
size	0	1	5	1	1	1	0	0	0	3	0	

Pseudocode for modified Union

Union(i, j)

$k \leftarrow \text{Find}(i);$

$l \leftarrow \text{Find}(j);$

If(size(k) < size(l))

$l \leftarrow \text{Parent}(k);$

 size(l) \leftarrow size(k) + size(l);

 size(k) \leftarrow 0;

Else

$k \leftarrow \text{Parent}(l);$

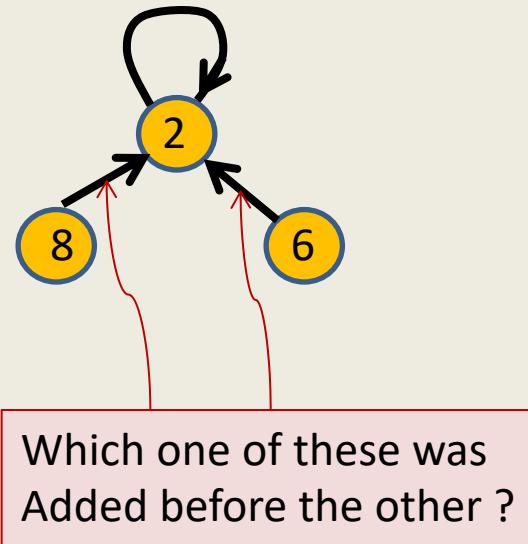
 size(k) \leftarrow size(k) + size(l);

 size(l) \leftarrow 0;

Question: How to show that **Find(i)** for any i will now take $O(\log n)$ time only ?

Answer: It suffices if we can show that **Depth(i)** is $O(\log n)$.

Can we infer history of a tree?

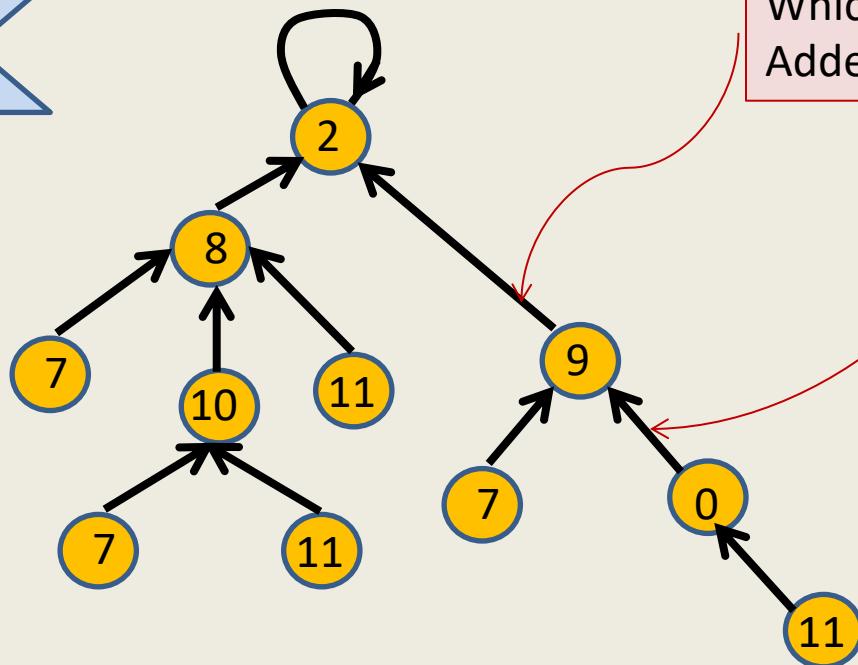


Answer: Can not be inferred with any certainty 😔.

Can we infer history of a tree?

During union, we join **roots** of two trees.

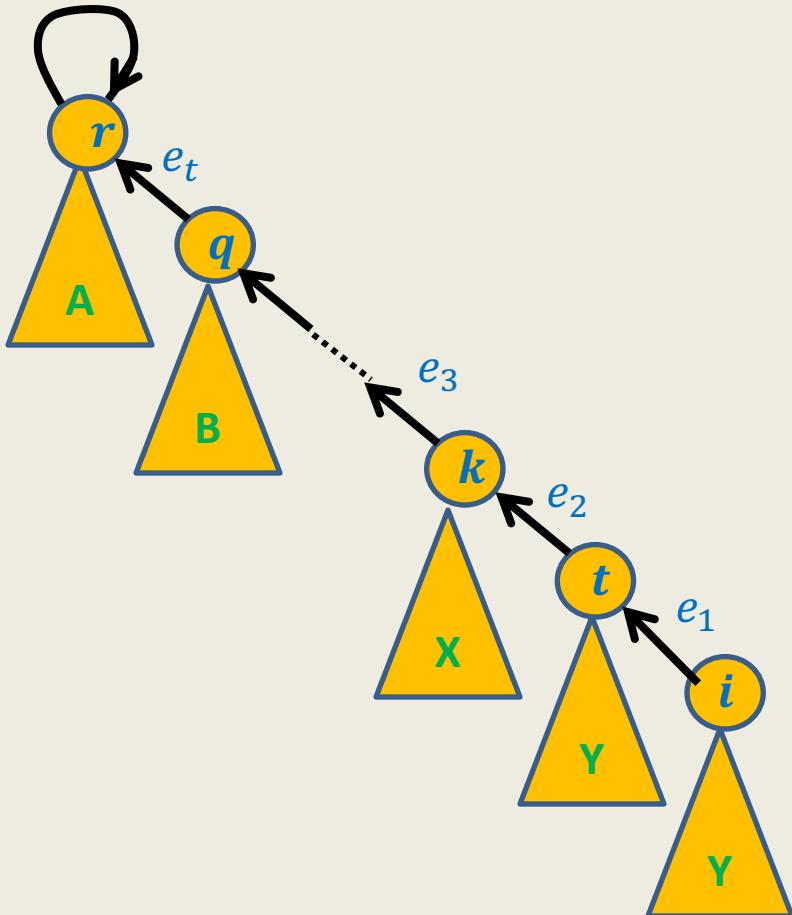
Which one of these was
Added before the other ?



→ $(0 \rightarrow 9)$ was added **before** $(9 \rightarrow 2)$.

Theorem: The edges on a **path** from node v to root were inserted in the order they appear on the **path**.

How to show that depth of any element = $O(\log n)$?



Let e_1, e_2, \dots, e_t be the edges on the path from i to the **root**.

Let us visit the history.
(how this tree came into being ?).

Edges e_1, e_2, \dots, e_t would have been added in the order:

e_1
 e_2
...
 e_t

How to show that depth of any element = $O(\log n)$?

Let e_1, e_2, \dots, e_t be the edges on the path from i to the root.

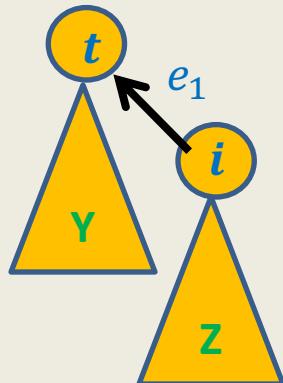
Consider the moment just before edge e_1 is inserted.

Let no. of elements in subtree $T(i)$ at that moment be n_i .

We added edge $i \rightarrow t$ (and not $t \rightarrow i$).

→ no. of elements in $T(t) \geq n_i$.

→ After the edge $i \rightarrow t$ is inserted,
no. of element in $T(t) \geq 2n_i$

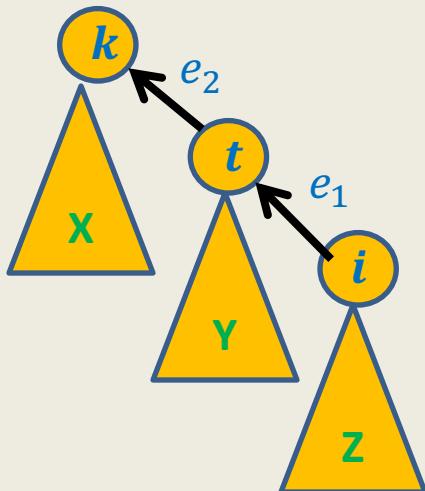


How to show that depth of any element = $O(\log n)$?

Let e_1, e_2, \dots, e_t be the edges on the path from i to the root.

Consider the moment just before edge e_2 is inserted.

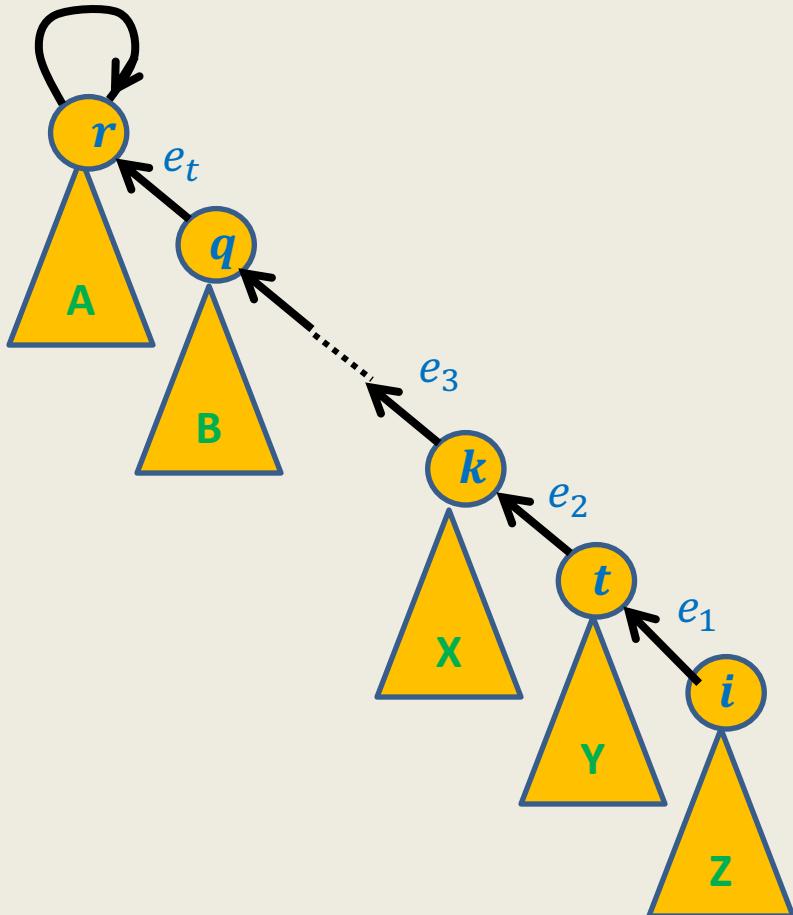
no. of element in $T(t) \geq 2n_i$



We added edge $t \rightarrow k$ (and not $k \rightarrow t$).

- # elements in $T(k) \geq 2n_i$.
- After the edge $t \rightarrow k$ is inserted,
no. of element in $T(k) \geq 4n_i$

How to show that depth of any element = $O(\log n)$?



Let e_1, e_2, \dots, e_t be the edges on the path from i to the root.

Arguing in a similar manner for edge $e_3, \dots, e_t \rightarrow$

elements in $T(r)$ after insertion of $e_t \geq 2^t n_i$

Obviously $2^t n_i \leq n$

→

Theorem: $t \leq \log_2 n$

Theorem: Given a collection of n singleton sets followed by a sequence of **union** and **find** operations, there is a data structure based that achieves $O(\log n)$ time per operation.

Question: Can we achieve even better bounds ?

Answer: Yes.

A new heuristic for better time complexity

Heuristic 2: Path compression

This is how this heuristic got invented

- The time complexity of a **Find(*i*)** operation is proportional to the depth of the node *i* in its rooted tree.
- If the elements are stored closer to the root, faster will the **Find()** be and hence faster will be the overall algorithm.

The algorithm for **Union** and **Find** was used in some application of **data-bases**.

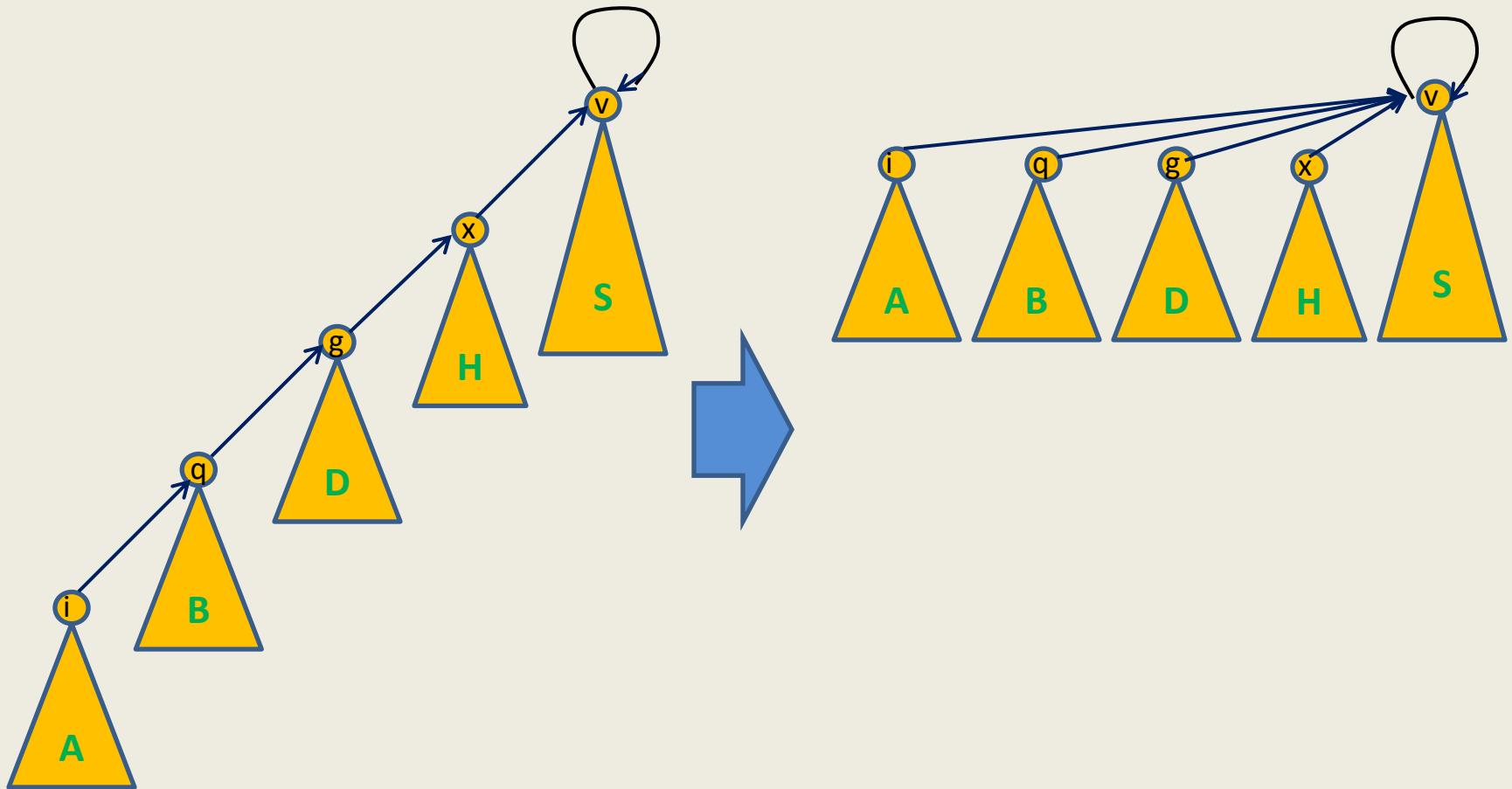
A clever programmer did the following modification to the code of **Find(*i*)**.

While executing **Find(*i*)**, we traverse the path from node *i* to the root. Let v_1, v_2, \dots, v_t , be the nodes traversed with v_t being the root node. At the end of **Find(*i*)**, if we update parent of each v_k , $1 \leq k < t$, to v_t , we achieve a reduction in depth of many nodes. This modification increases the time complexity of **Find(*i*)** by at most a constant factor. But this little modification increased the overall speed of the application very significantly.

The heuristic is called **path compression**. It is shown pictorially on the following slide.

It remained a mystery for many years to provide a theoretical explanation for its practical success.

Path compression during Find(i)



Pseudocode for the modified Find

Find(*i*)

If (**Parent(*i*)** = *i*) return *i* ;

else

j \leftarrow **Find(Parent(*i*));**

Parent(*i*) \leftarrow *j*;

return *j*

Data Structures and Algorithms

(ESO207)

Lecture 33

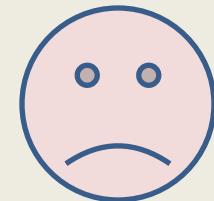
- Algorithm for i th order statistic of a set S .

Problem definition

Given a set S of n elements
compute i th smallest element from S .

Applications:

Trivial algorithm:



But sorting takes $O(n \log n)$ time
and appears to be an overkill
for this simple problem.

AIM: To design an algorithm with $O(n)$ time complexity.

Assumption (For the sake of **neat description** and **analysis** of algorithms of this lecture):

- All elements of S are assumed to be **distinct**.

A motivational background

Though it was **intuitively appealing** to believe that there exists an $O(n)$ time algorithm to compute i th smallest element, it remained a challenge for many years to design such an algorithm...

In **1972**, five well known researchers: **Blum, Floyd, Pratt, Rivest, and Tarjan** designed the $O(n)$ time algorithm. It was designed during a **lunch break** of a conference when these five researchers sat together for the first time to solve the problem.

In this way, the problem which remained unsolved for many years got solved in less than an hour. But one should not ignore the efforts these researchers spent for years before arriving at the solution ... It was their effort whose fruit got ripened in that hour 😊.

Notations

We shall now introduce some notations which will help in a neat description of the algorithm.

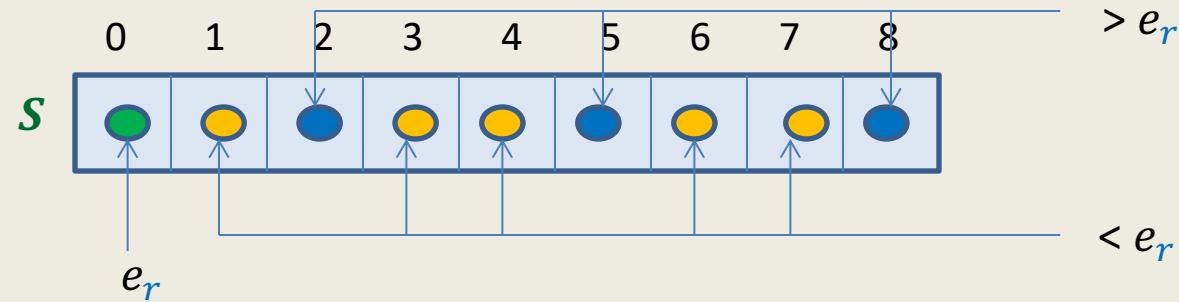
Notations

- S :
the given set of n elements.
- e_i :
 i th **smallest** element of S .
- $S_{<x}$:
subset of S consisting of all elements **smaller than** x .
- $S_{>x}$:
subset of S consisting of all elements **greater than** x .
- $\text{rank}(S,x)$:
 $1 +$ number of elements in S that are smaller than x .
- **Partition**(S,x):
algorithm to partition S into $S_{<x}$ and $S_{>x}$;
this algorithm returns $(S_{<x}, S_{>x}, r)$ where $r = \text{rank}(S,x)$.

Why should such an algorithm exist ?

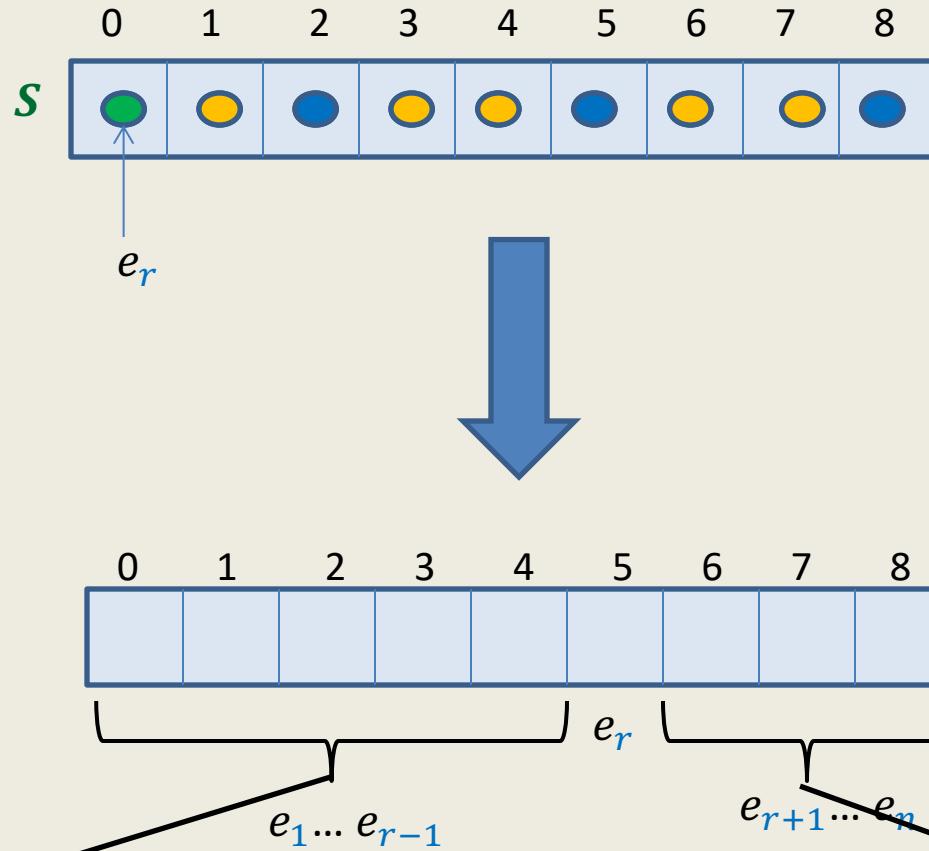
(inspiration from **QuickSort**)

QuickSelect(S, i)



What happens during
Partition($S, 0, 8$)

QuickSelect(S, i)



If $i > r$ we can discard these elements.

If $i < r$ we can discard these elements.

Pseudocode for QuickSelect(S, i)

```
QuickSelect( $S, i$ )
{
    Pick an element  $x$  from  $S$ ;
     $(S_{<x}, S_{>x}, r) \leftarrow \text{Partition}(S, x);$ 
    If( $i = r$ ) return  $x$ ;
    Else If ( $i < r$ )
        QuickSelect( $S_{<x}, i$ )
    Else
        QuickSelect(  $S_{>x}, i - r$  );
}
```

Average case time complexity: $O(n)$

Worst case time complexity : $O(n^2)$

Analysis is simpler than Quick Sort.

Towards worst case $O(n)$ time algorithm ...

Key ideas

- Inspiration from some recurrences.
- Concept of approximate median
- Learning from QuickSelect(S, i)

isn't it surprising that knowledge of recurrence can help in the design of an efficient algorithm? 😊

This is the usual trick:
When a problem appears difficult, weaken the problem and try to solve it.

This is also a natural choice.
Can we fine tune this algorithm to achieve our goal?

Learning from recurrences

Question: what is the solution of recurrence $T(n) = cn + T(9n/10)$?

Answer: $O(n)$.

Sketch (by gradual unfolding):

$$\begin{aligned}T(n) &= cn + c \frac{9n}{10} + c \frac{81n}{100} + \dots \\&= cn[1 + \frac{9}{10} + \frac{81}{100} + \dots] \\&= O(n)\end{aligned}$$

Lesson 1 :

Solution for $T(n) = cn + T(an)$ is $O(n)$ if $0 < a < 1$.

Learning from recurrences

Question: what is the solution of recurrence

$$T(n) = cn + T(n/6) + T(5n/7) ?$$

Answer: $O(n)$.

Sketch: (by induction)

Assertion: $T(n) \leq c_1 n$.

Induction step: $T(n) = cn + T(n/6) + T(5n/7)$

$$\leq cn + \frac{37}{42}c_1 n$$

$$\leq c_1 n \text{ if } c_1 \geq \frac{42}{5} c$$

Lesson 2 :

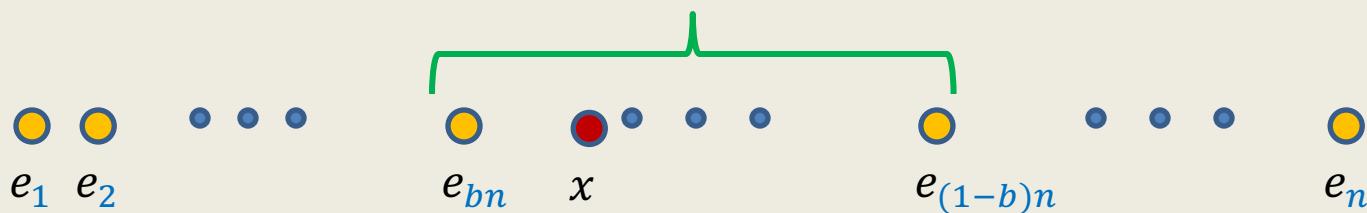
Solution for $T(n) = cn + T(an) + T(dn)$ is $O(n)$ if $a + d < 1$.

Concept of approximate median

Definition: Given a constant $0 < b \leq 1/2$,

an element $x \in S$

if $\text{rank}(x, S)$



Learning from QuickSelect(S, i)

QuickSelect(S, i)

```
{   Pick an element  $x$  from  $S$ ;  
     $(S_{<x}, S_{>x}, r) \leftarrow \text{Partition}(S, x);$   
    If( $i = r$ ) return  $x$ ;  
    Else If ( $i < r$ )  
        QuickSelect( $S_{<x}, i$ )  
    Else  
        QuickSelect( $S_{>x}, i - r$ );  
}
```

$O(n)$

$T((1 - b)n)$

Lesson 1

Answer: $T(n) = cn + T((1 - b)n)$
 $= O(n)$

What is time complexity
of the algorithm

Algorithm 2

Select(S, i)

(A linear time algorithm)

Overview of the algorithm

Select(S, i)

```
{    Compute a  $b$ -approximate median, say  $x$ , of  $S$ ;      }  $O(n)$ 
     $(S_{<x}, S_{>x}, r) \leftarrow \text{Partition}(S, x);$           }  $O(n)$ 
    If( $i = r$ ) return  $x$ ;
    Else If ( $i < r$ )
        Select( $S_{<x}, i$ )
    Else
        Select( $S_{>x}, i - r$ );
}
```

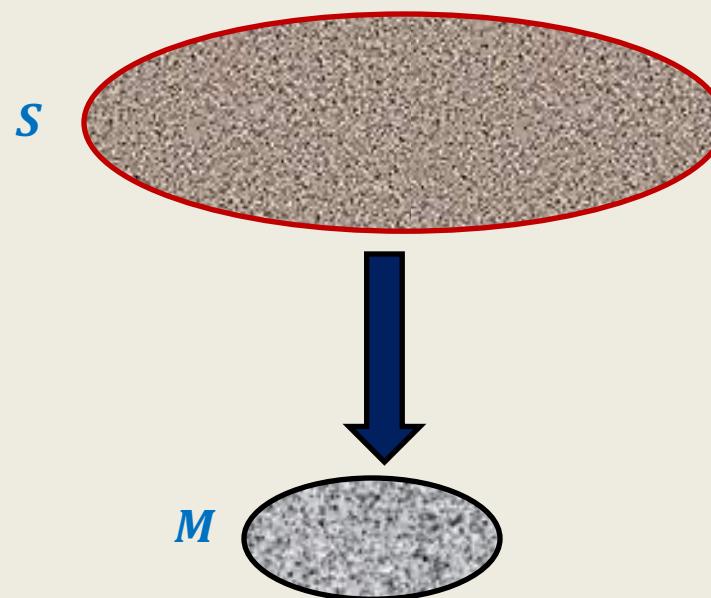
Observation: If we can compute b -approximate median in $O(n)$ time, we get $O(n)$ time algo.
But that appears too much to expect from us. Isn't it ?
So what to do 😔 ?

Hint: use **Lesson 2**

Observation: If we can compute b -approximate median
time complexity of the algorithm will still be $O(n)$.

Spend some time on this observation to infer what it hints at.

AIM: How to compute a b -approximate median of S
in $O(n) + T(dn)$ time



Question: Can we form a set M of size dn such that
exact median of M is b -approximate median of S ?

Forming the subset M with desired parameters

This step forms the core of the algorithm and is indeed a brilliant stroke of inspiration. The student is strongly recommended to ponder over this idea from various angles.

- Divide S into **groups** of **5** elements;
 - Compute median of each group by sorting;
 - Let M be the set of medians;
 - Compute median of M , let it be x ;
- $\left. \begin{matrix} \\ \\ \\ \end{matrix} \right\} O(n)$
 $\left. \begin{matrix} \\ \end{matrix} \right\} T(n/5)$

Question: Is x an approximate median of S ?

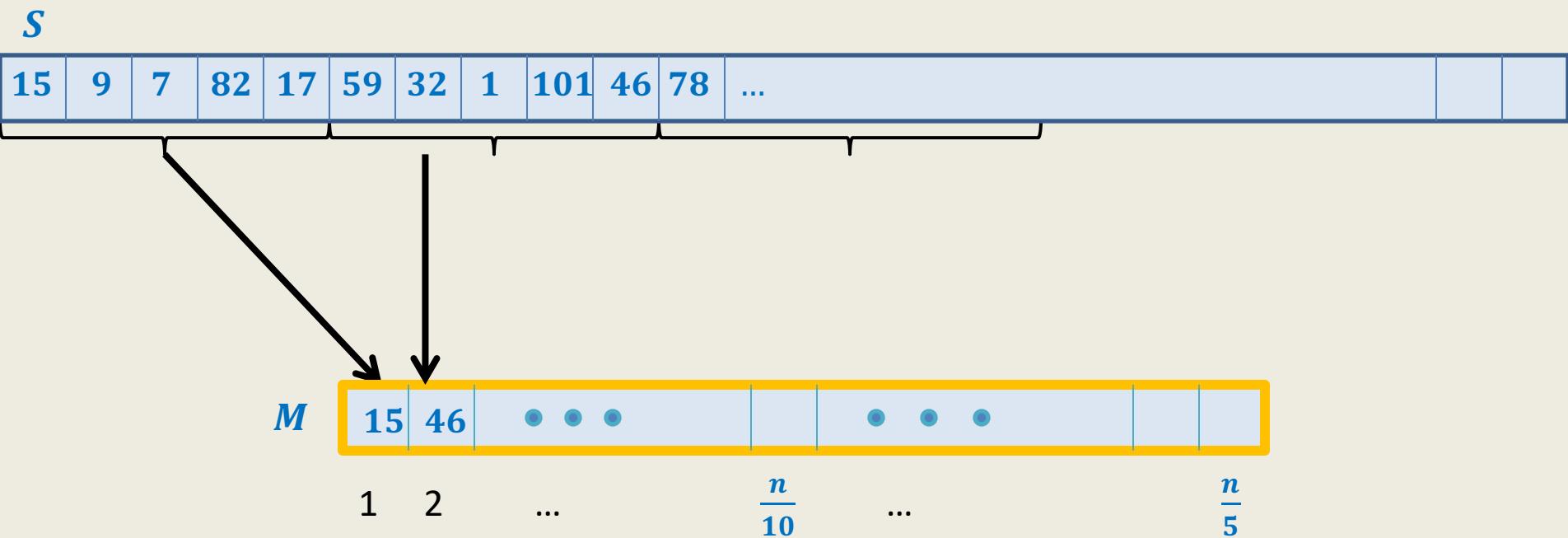
Answer: indeed.

The rank of x in M is $n/10$. Each element in M has two elements smaller than itself in its respective group. Hence there are at least $\frac{3n}{10} - 1$ elements in S which are **smaller** than x .

In a similar way, there are at least $\frac{3n}{10} - 1$ elements in S which are **greater** than x . Hence, x is $\frac{3n}{10}$ -approximate median of S .

(See the animation on the following slide to get a better understanding of this explanation.)

Forming the subset M



- Divide S into **groups** of 5 elements;
- Compute median of each group by sorting;

$\left. \right\} O(n)$

Forming the subset M

S

15	9	7	82	17	59	32	1	101	46	78	...



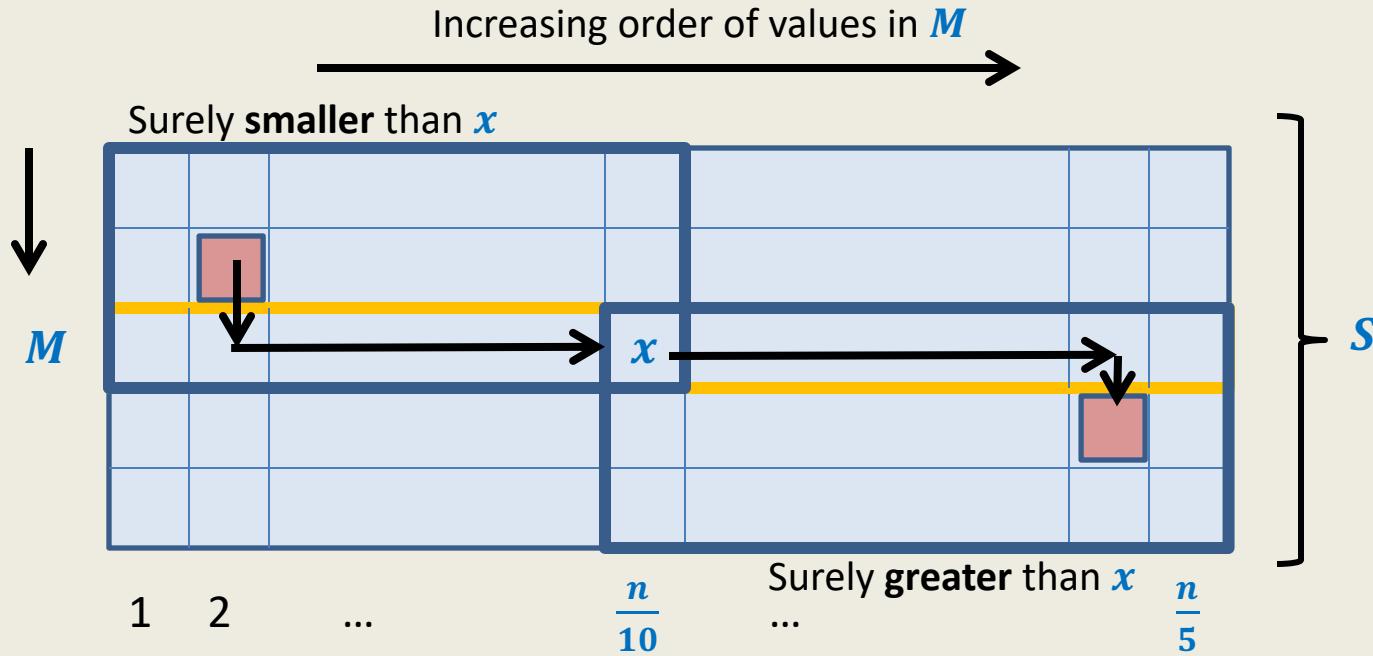
- Divide S into **groups** of 5 elements;
- Compute median of each group by sorting;
- Let M be the set of medians;
- Let x be median of M .

What can we say about rank of x in S ?

Spend some time to answer this question before moving ahead.

Forming the subset M

Bring back the remaining 4 elements associated with each element of M



→ x is $\left(\frac{3n}{10}\right)$ –approximate median of S .

Time required to form M : $O(n)$

Pseudocode for $\text{Select}(S, i)$

$\text{Select}(S, i)$

$M \leftarrow \emptyset;$

Divide S into **groups** of 5 elements;

Sort each group and add its **median** to M ;

$x \leftarrow \text{Select}(M, |M|/2);$

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

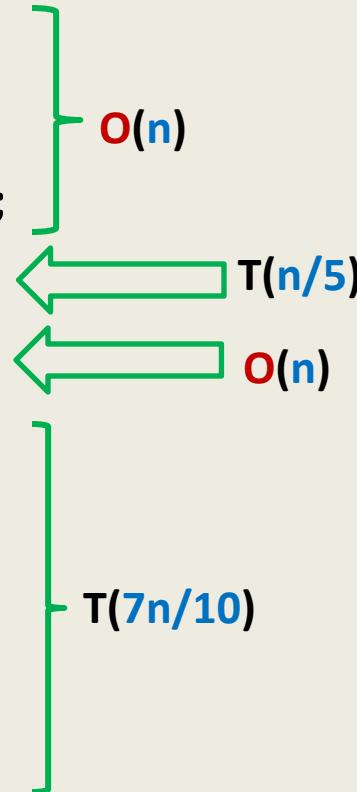
If($i = r$) return x ;

Else If ($i < r$)

$\text{Select}(S_{<x}, i)$

Else

$\text{Select}(S_{>x}, i - r);$



Analysis

$$\begin{aligned} T(n) &= cn + T(n/5) + T(7n/10) \\ &= O(n) \quad [\text{Learning from Recurrence of type II}] \end{aligned}$$

Theorem: Given any S of n elements, we can compute i th smallest element from S in $O(n)$ worst case time.

Exercises

(Attempting these exercises will give you a better insight into the algorithm.)

- What is magical about number **5** in the algorithm ?
- What if we divide the set **S** into groups of size **3** ?
- What if we divide the set **S** into groups of size **7** ?
- What if we divide the set **S** into groups of even size (e.g. **4** or **6**) ?

Data Structures and Algorithms

(ESO207)

Lecture 34

- A new algorithm design paradigm: Greedy strategy
part I

Path to the solution of a problem



But there is a **systematic approach** which usually works ☺

Today's lecture will demonstrate this approach 😊

Problem : JOB Scheduling

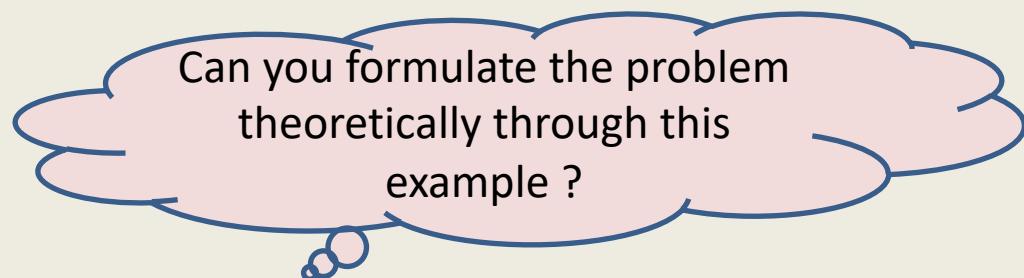
Largest subset of non-overlapping job

A motivating example

Antaragni 2021

- There are ***n*** large-scale activities to be performed in Auditorium.
- Each large scale activity has a **start time** and **finish time**.
- There is **overlap** among various activities.

Aim: What is the largest subset of activities that can be performed ?



Formal Description

A job scheduling problem

INPUT:

- A set J of n jobs $\{j_1, j_2, \dots, j_n\}$
- job j_i is specified by two real numbers
 - $s(i)$: start time of job j_i
 - $f(i)$: finish time of job j_i
- A single server

Constraints:

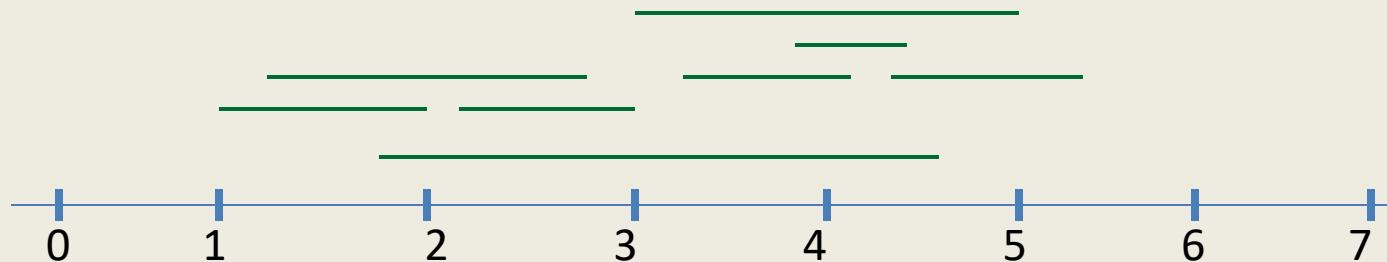
- Server can execute at most one job at any moment of time.
- Job j_i , if scheduled,

Aim:

To select the **largest** subset of non-overlapping jobs which can be executed by the server.

Example

INPUT: $(1, 2)$, $(1.2, 2.8)$, $(1.8, 4.6)$, $(2.1, 3)$, $(3, 5)$, $(3.3, 4.2)$, $(3.9, 4.4)$, $(4.3, 5.4)$



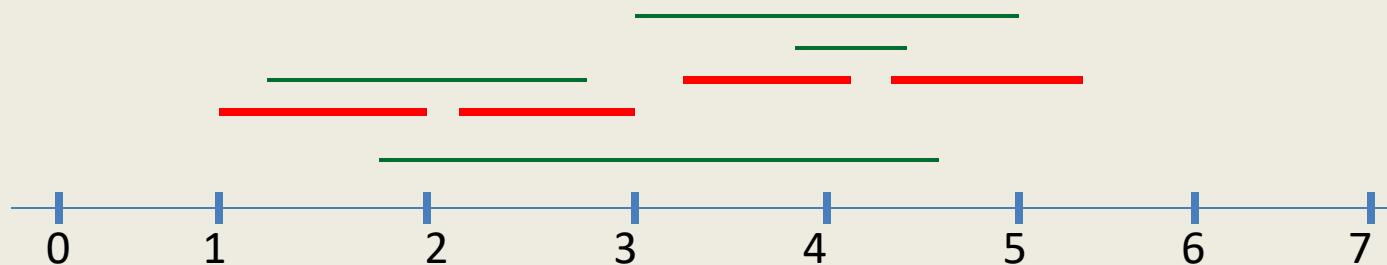
It makes sense to work with pictures than these numbers😊

job i is said to be **non-overlapping** with job k if

Try to find solution for the above example.

Example

INPUT: $(1, 2), (1.2, 2.8), (1.8, 4.6), (2.1, 3), (3, 5), (3.3, 4.2), (3.9, 4.4), (4.3, 5.4)$



job i is said to be **non-overlapping** if $\text{f}(k) \cap \text{f}(i) = \emptyset$

What strategy come
to your mind?

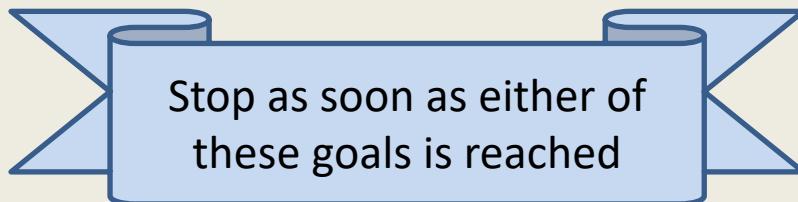
Designing algorithm for any problem

1. Choose a strategy based on some intuition
2. Transform the strategy into an algorithm.

Try to prove
correctness
of the
algorithm

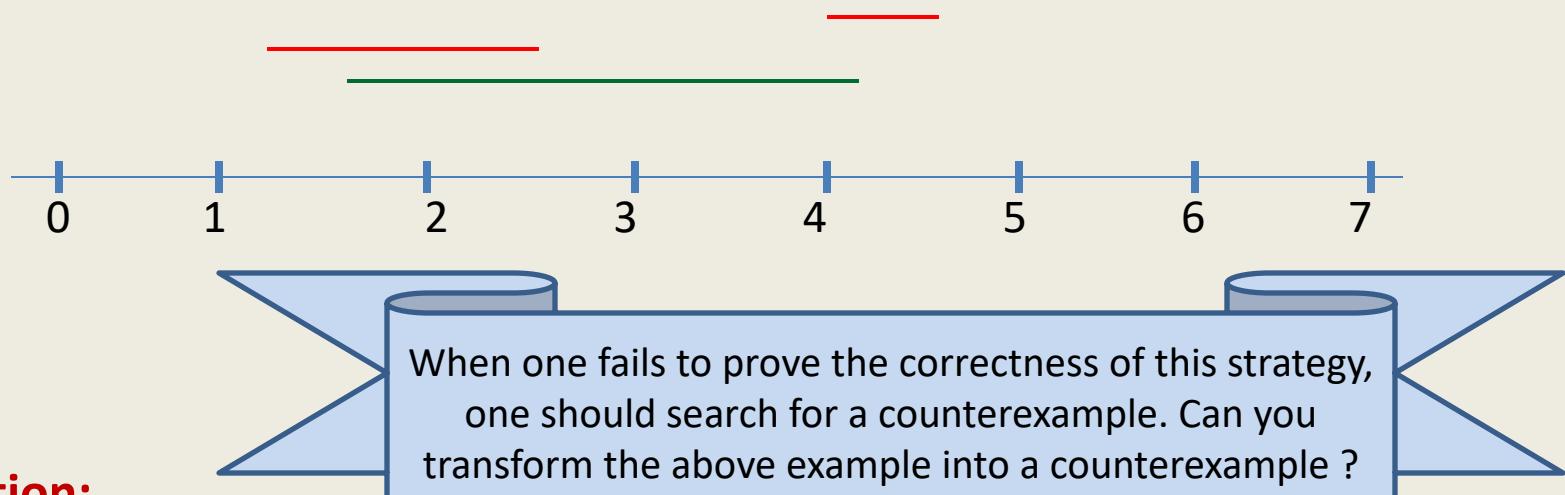


Try to design a
conterexample



Designing algorithm for the problem

Strategy 1: Select the earliest start time job

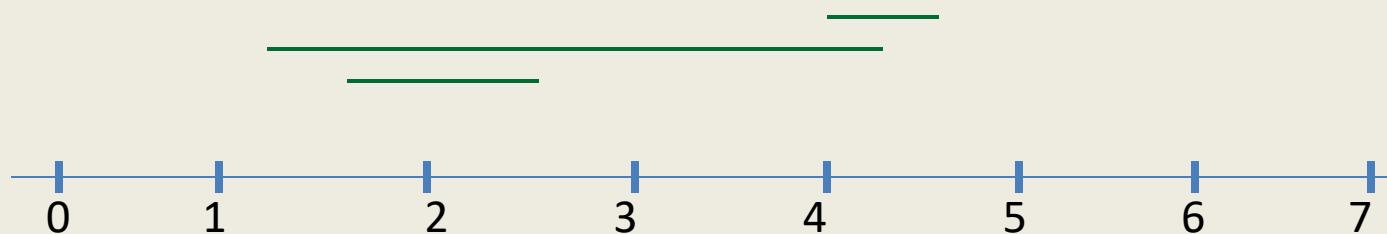


Intuition:

It might be better to assign jobs as early as possible so as to make optimum use of server.

Designing algorithm for the problem

Strategy 1: Select the earliest start time job

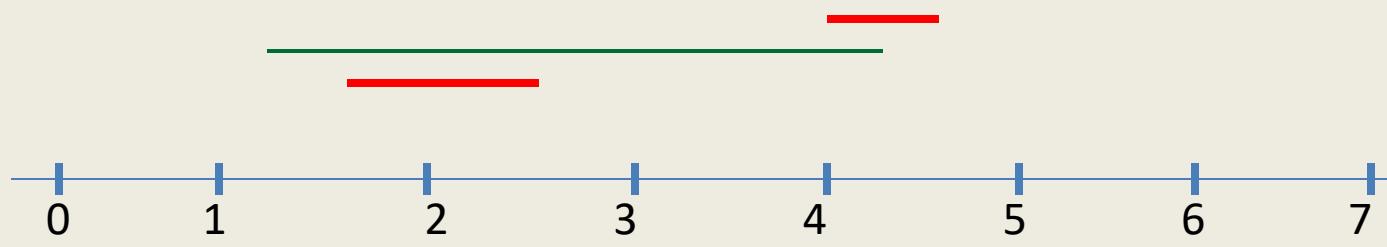


Intuition:

It might be better to assign jobs as early as possible so as **to make optimum use of server**.

Designing algorithm for the problem

Strategy 1: Select the earliest start time job



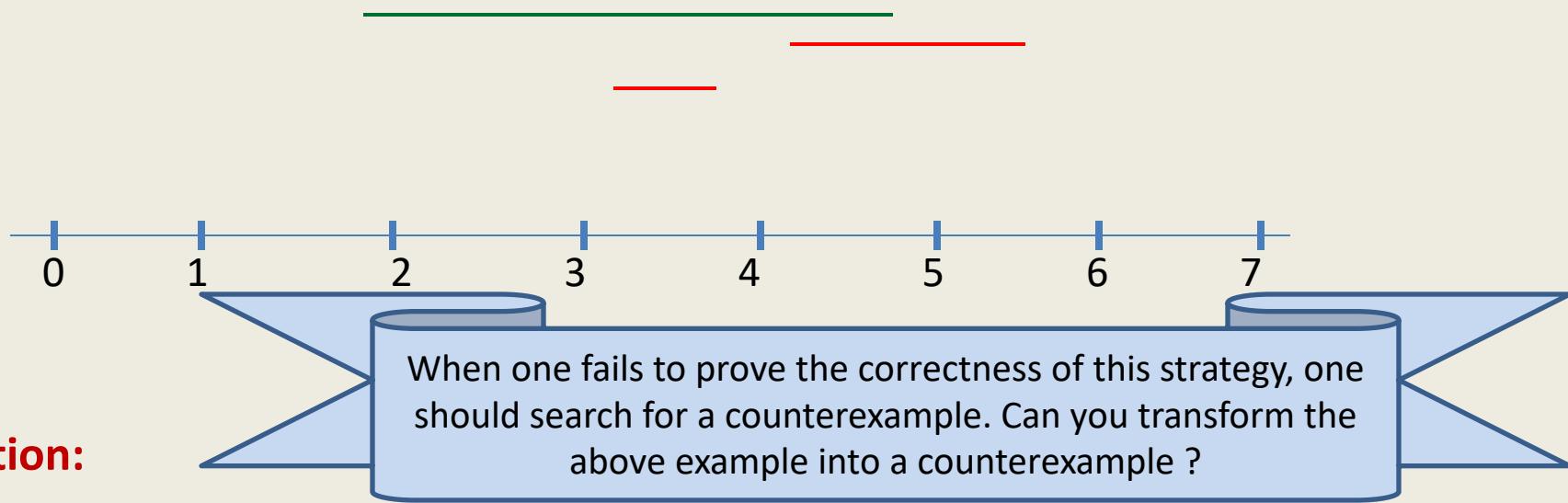
Intuition:

Instead of getting disappointed, try to realize that this counterexample points towards some other strategy which might work.

It might be better to assign jobs as early as possible so as to make optimum use of server.

Designing algorithm for the problem

Strategy 2: Select the job with **smallest duration**



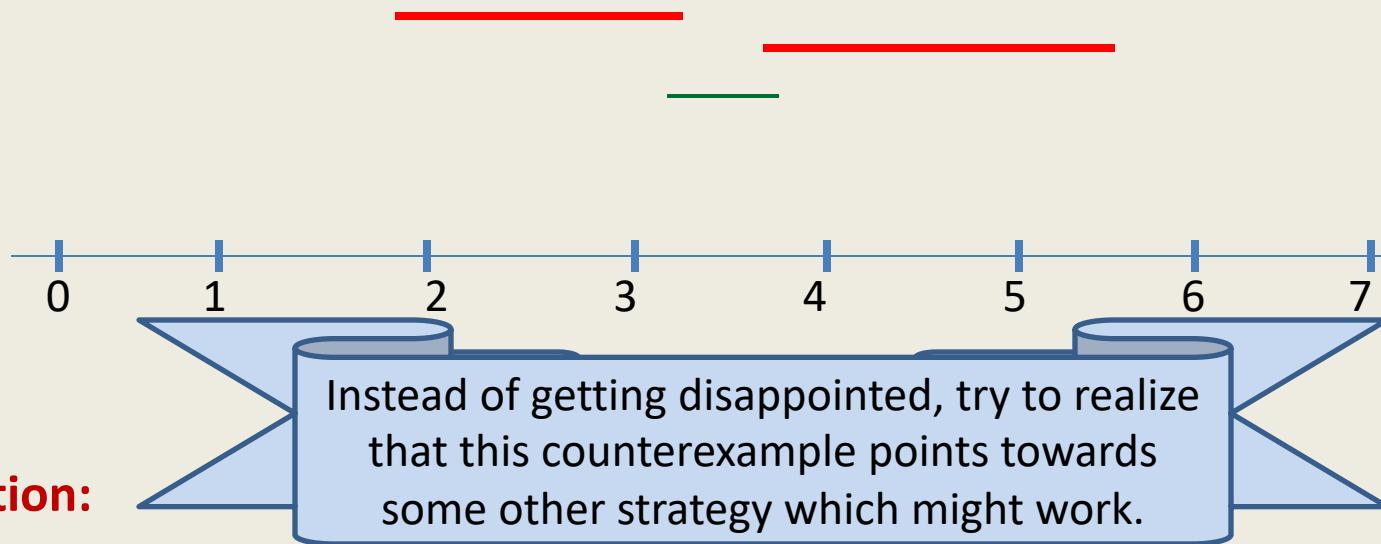
Intuition:

Such a job will make **least use of the server**

→ this might lead to larger number of jobs to be executed

Designing algorithm for the problem

Strategy 2: Select the job with **smallest duration**



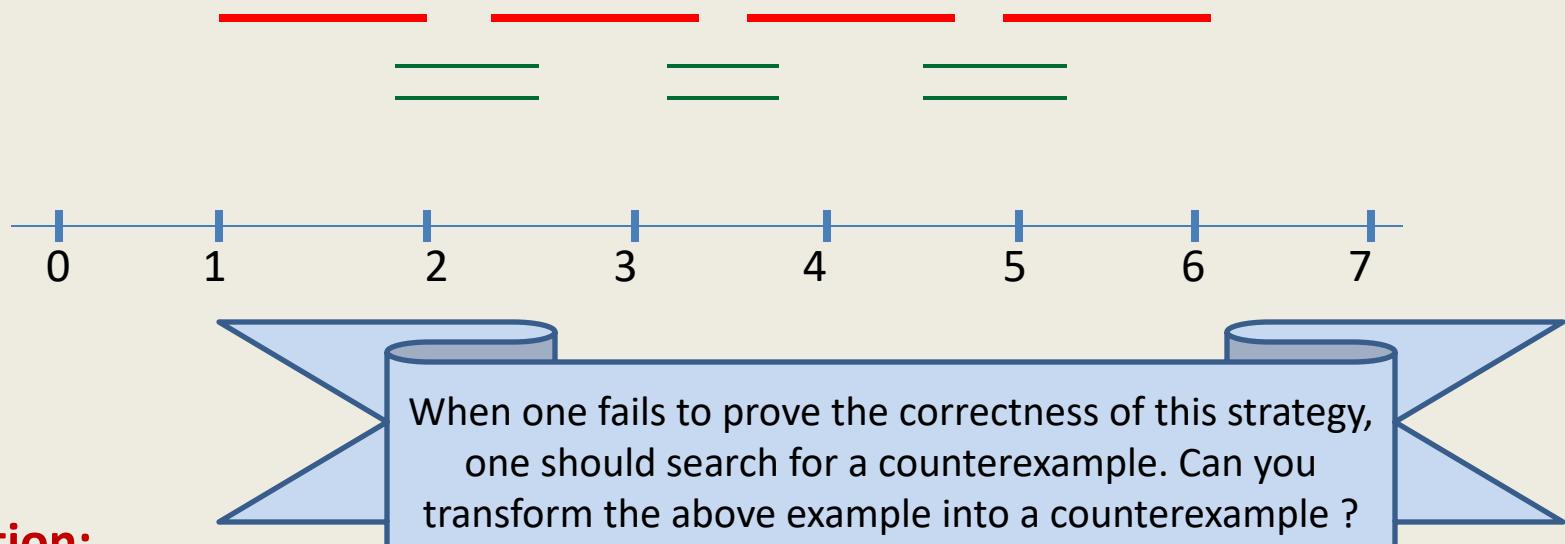
Intuition:

Such a job will make **least use of the server**

→ this might lead to larger number of jobs to be executed

Designing algorithm for the problem

Strategy 3: Select the job with **smallest no. of overlaps**

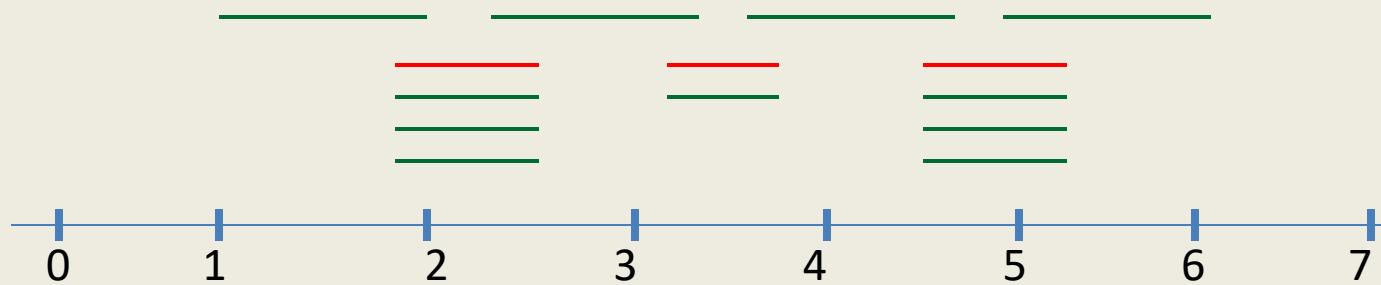


Intuition:

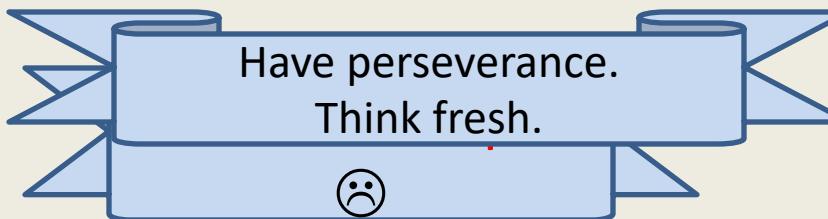
Selecting such a job will result in **least number of other jobs to be discarded**.

Designing algorithm for the problem

Strategy 3: Select the job with **smallest no. of overlaps**



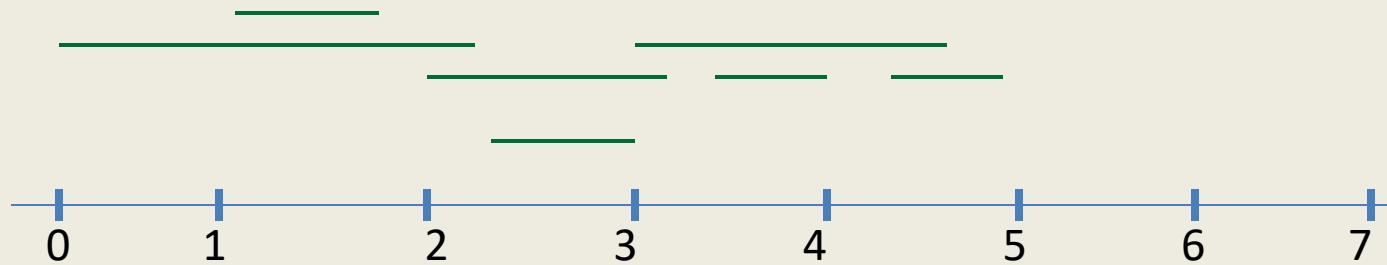
Intuition:



Selecting such a job will result in **least number of other jobs to be discarded**.

Designing algorithm for the problem

Strategy 4: Select the job with **earliest finish time**



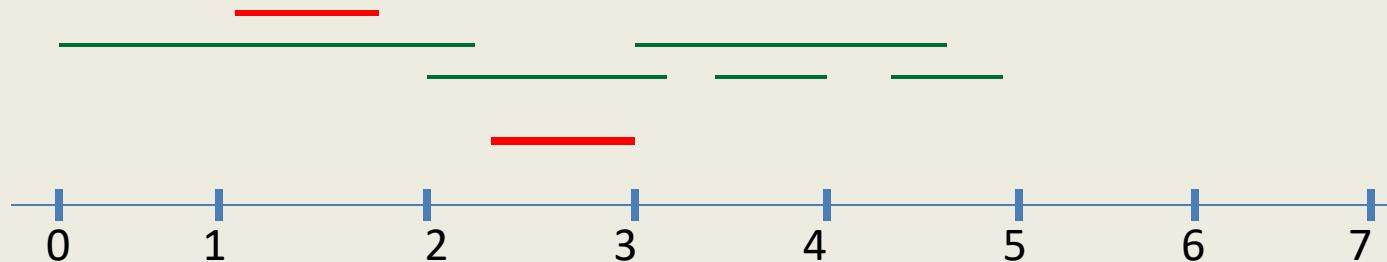
Intuition:

Selecting such a job will **free** the server **earliest**

→ hence more no. of jobs might get scheduled.

Designing algorithm for the problem

Strategy 4: Select the job with **earliest finish time**



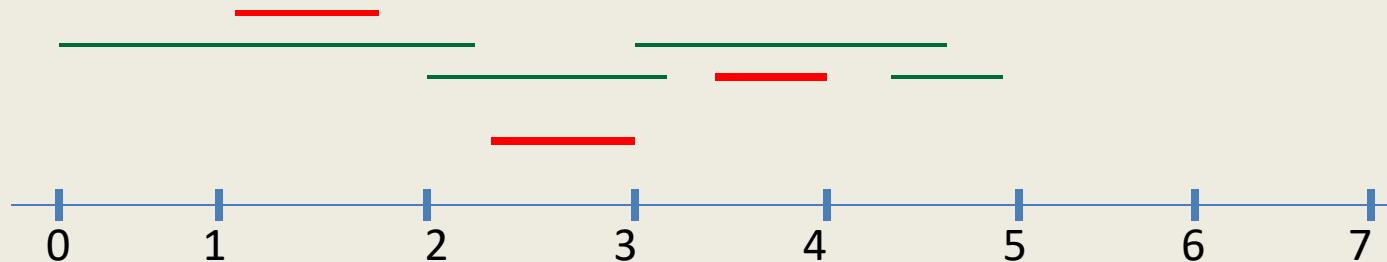
Intuition:

Selecting such a job will **free** the server **earliest**

→ hence more no. of jobs might get scheduled.

Designing algorithm for the problem

Strategy 4: Select the job with **earliest finish time**



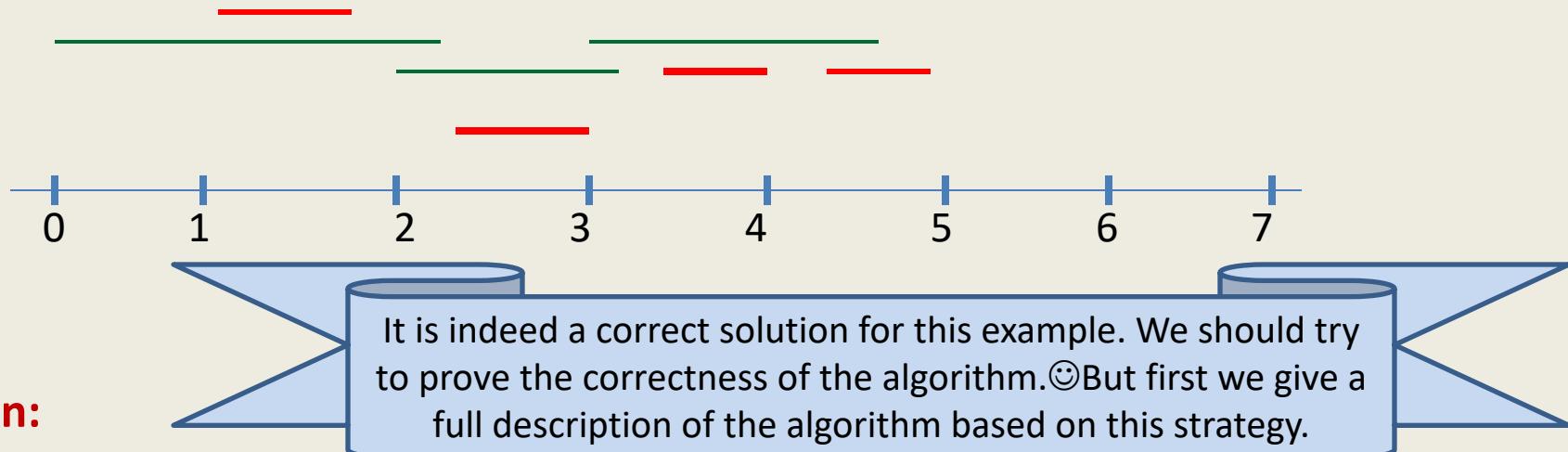
Intuition:

Selecting such a job will **free** the server **earliest**

→ hence more no. of jobs might get scheduled.

Designing algorithm for the problem

Strategy 4: Select the job with earliest finish time



Selecting such a job will **free** the server **earliest**
→ hence more no. of jobs might get scheduled.

Algorithm “earliest finish time”

Description

Algorithm (Input : set J of n jobs.)

1. Define $A \leftarrow \emptyset$;
2. While $J \neq \emptyset$ do
 - { Let x be the job from J with earliest finish time;
 $A \leftarrow A \cup \{x\}$;
Remove x and all jobs that overlap with x from set J ;
 - }
3. Return A ;

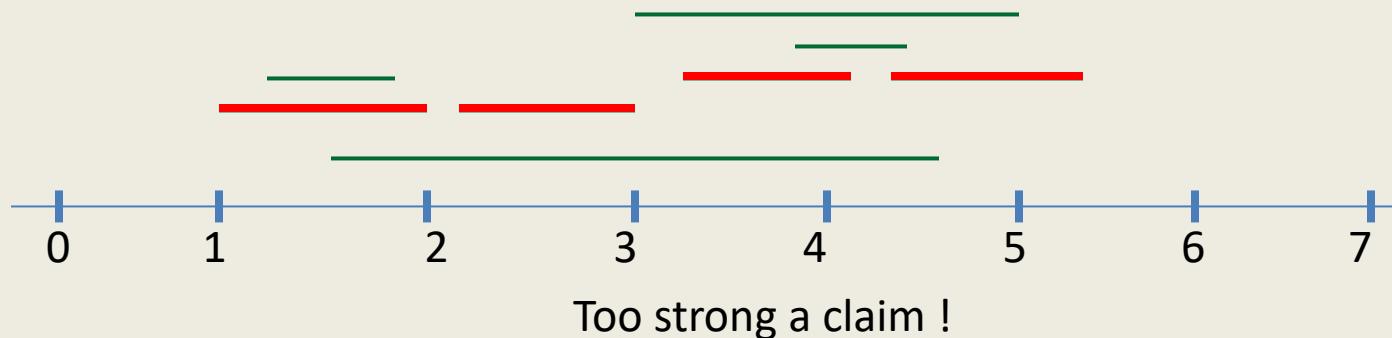
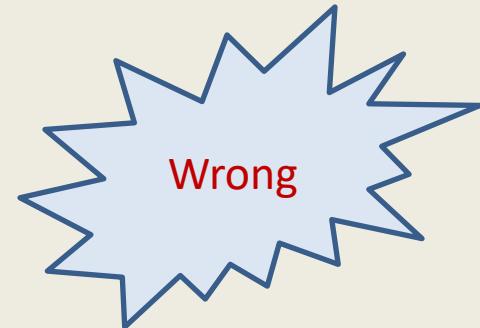
Running time for a trivial implementation of the above algorithm: $O(n^2)$

Algorithm “earliest finish time”

Correctness

Let x be the job with earliest finish time.

Lemma1: x is present in the optimal solution for J .



Algorithm “earliest finish time”

Correctness

Let x be the job with earliest finish time.

Lemma1: There exists an optimal solution for J in which x is present.

Proof: Consider any optimal solution O for J .

Let y be the job from O with earliest finish time.

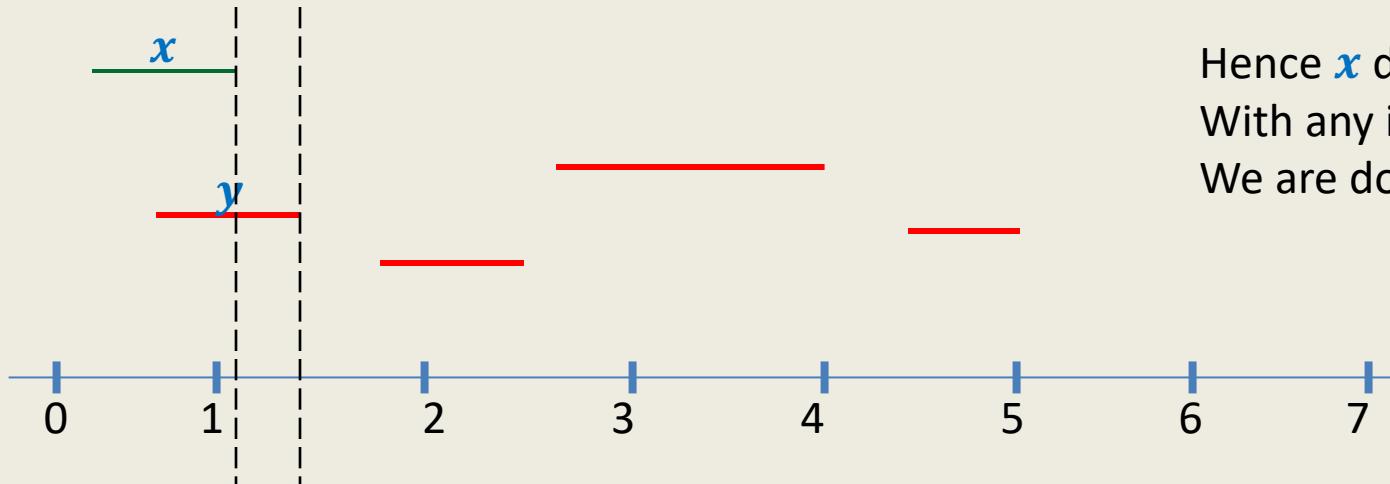
Let $O' \leftarrow O \setminus \{y\}$

$$\rightarrow f(y) < s(z) \forall z \in O'$$

$$\rightarrow f(x) \leq f(y)$$

$O' \cup \{x\}$ is also an optimal solution.

Reason: $O' \cup \{x\}$ has no overlapping intervals. Give arguments.



Hence x does not overlap
With any interval of O' .
We are done 😊

Homework

Spend 30 minutes today on the following problems.

1. Use **Lemma1** to complete the proof of correctness of the algorithm.
2. Design an **$O(n \log n)$** implementation of the algorithm.

Data Structures and Algorithms

(ESO207)

Lecture 35

- A new algorithm design paradigm: Greedy strategy
part II

Continuing Problem from last class

JOB Scheduling

Largest subset of non-overlapping job

A job scheduling problem

Formal Description

INPUT:

- A set J of n jobs $\{j_1, j_2, \dots, j_n\}$
- job j_i is specified by two real numbers
 - $s(i)$: start time of job j_i
 - $f(i)$: finish time of job j_i
- A single server

Constraints:

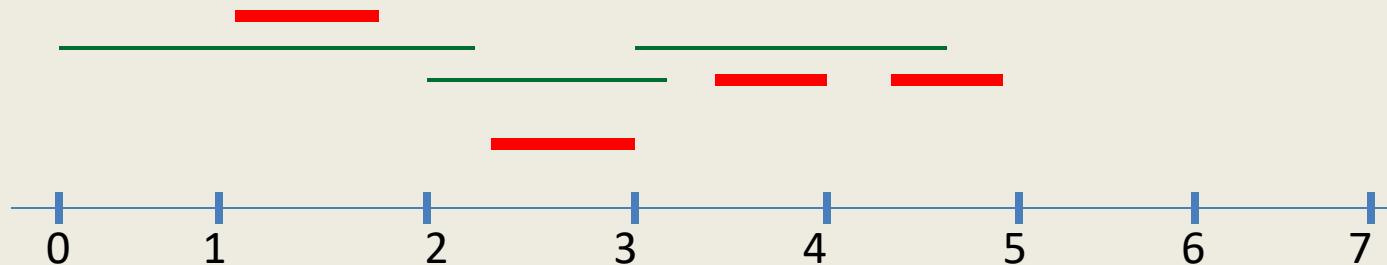
- Server can execute at most one job at any moment of time and a job.
- Job j_i , if scheduled, has to be scheduled during $[s(i), f(i)]$ only.

Aim:

To select the **largest** subset

Designing algorithm for the problem

Strategy 4: Select the job with **earliest finish time**



Intuition:

Selecting such a job will **free** the server **earliest**

→ hence more no. of jobs might get scheduled.

Algorithm “earliest finish time”

Proof of correctness ?

Let $x \in J$ be the job with earliest finish time.

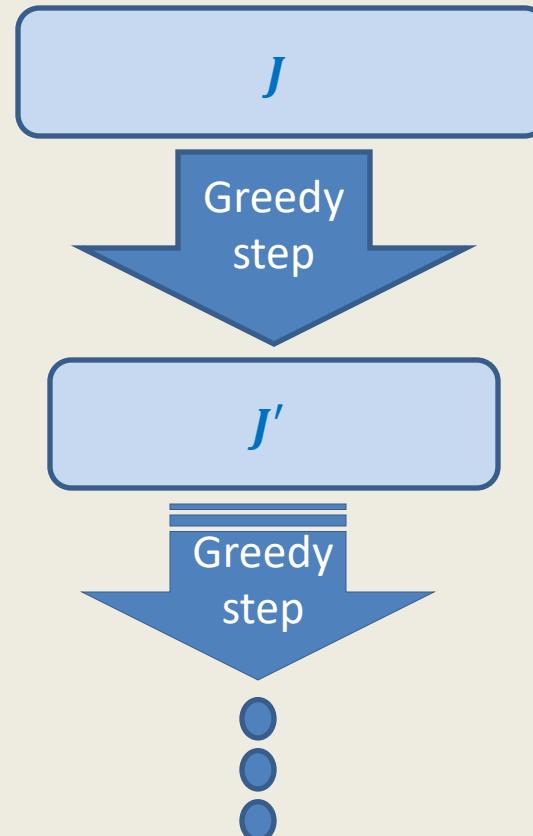
Let $J' = J \setminus \text{Overlap}(x)$

Algorithm (Input : set J of n jobs.)

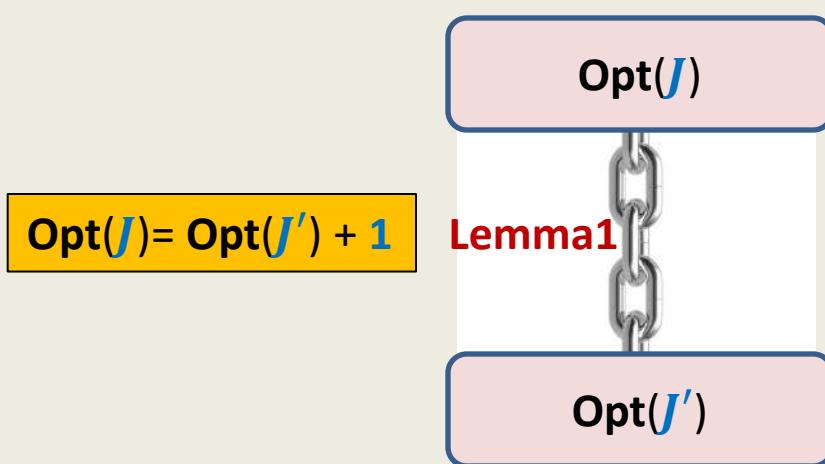
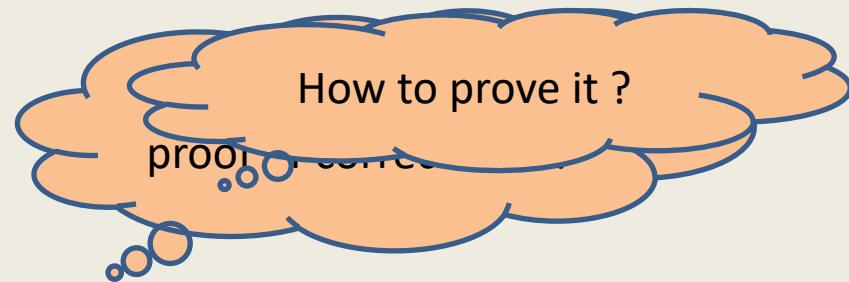
1. Define $A \leftarrow \emptyset$;
2. While $J \neq \emptyset$ do
 - { Let $x \in J$ has earliest finish time;
 $A \leftarrow A \cup \{x\}$;
 $J \leftarrow J \setminus \text{Overlap}(x)$;
 - }
3. Return A ;

Lemma1 (last class):

There exists an optimal solution for J containing the **earliest finish time** job.

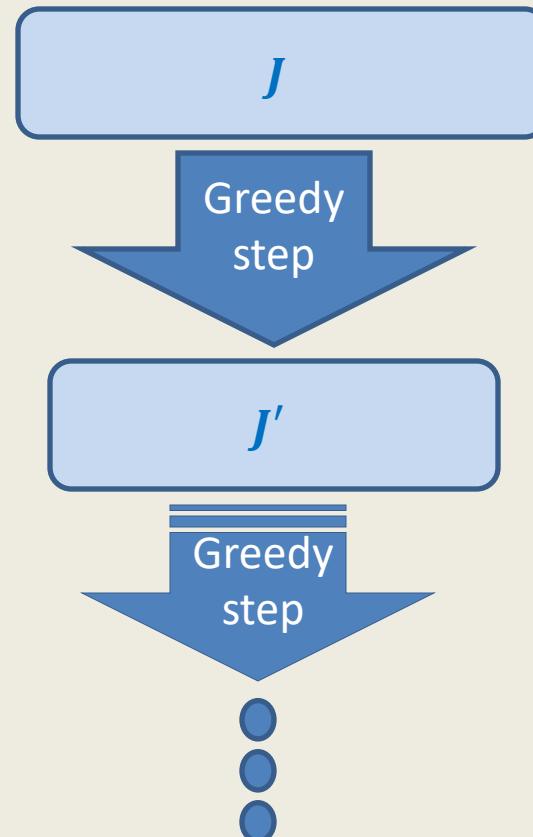


Algorithm “earliest finish time”



Proof of correctness ?

Let $x \in J$ be the job with earliest finish time.
Let $J' = J \setminus \text{Overlap}(x)$



Notation:

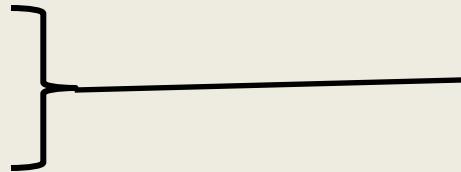
$\text{Opt}(J)$:

Theorem: $\text{Opt}(J) = \text{Opt}(J') + 1$.

- Proof has two parts

$$\text{Opt}(J) \geq \text{Opt}(J') + 1$$

$$\text{Opt}(J') \geq \text{Opt}(J) - 1$$



Try to give a physical interpretation to these inequalities.

- Proof for each part is a proof **by construction**

Algorithm “earliest finish time”

Proving $\text{Opt}(J) \geq \text{Opt}(J') + 1$.

Observation: start time of every job in J' is greater than finish time of x .

Let O' be any optimal solution for J' .

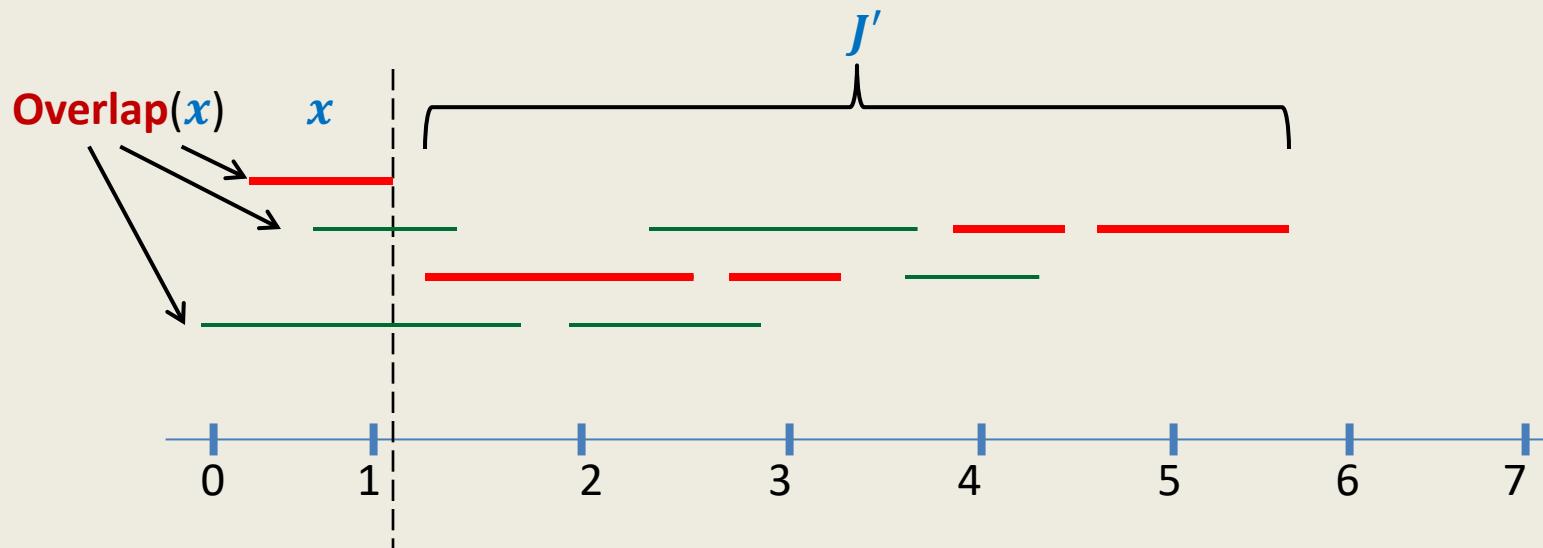
None of the jobs in

From an optimal solution of J'

Hence $O' \cup \{x\}$ is a

can you derive a solution for J with one extra job?

Therefore $\text{Opt}(J) \geq |O'| + 1$



Algorithm "earliest finish time"

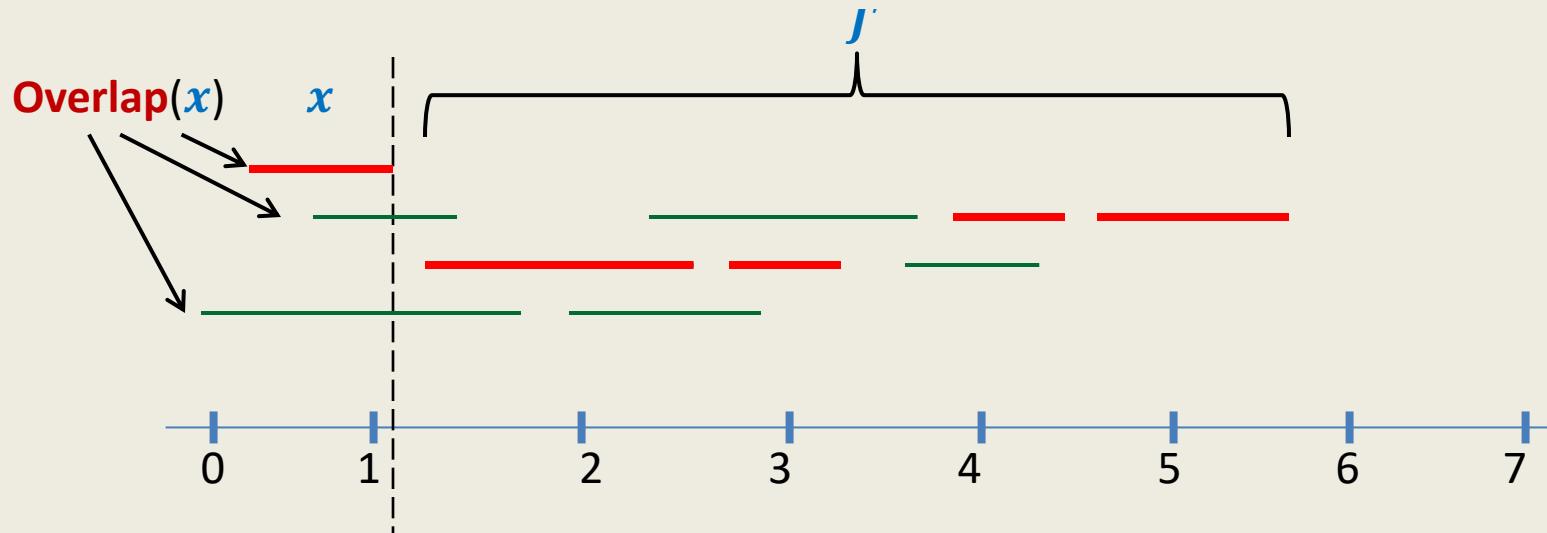
Lemma1 (last class): There exists an optimal solution for J in which x is present.

Let O be an optimal solution for J containing x .

None of the jobs in O overlaps with x .

→ Every job from O can you derive a solution for J' with one job less?
Hence $O \setminus \{x\}$ is a subset of non-overlapping jobs from J .

Therefore $\text{Opt}(J') \geq |O| - 1$:



Theorem:

Given any set J of n jobs,

the algorithm based on “earliest finish time” approach
computes the largest subset of non-overlapping job.

$O(n \log n)$ implementation of the Algorithm

This is not the only way to achieve $O(n \log n)$ time. It can be done other ways as well.

Algorithm (Input : set J of n jobs.)

1. Define $A \leftarrow \emptyset$;
2. While $J \neq \emptyset$ do

{ Let $x \in J$ have earliest finish time;

$A \leftarrow A \cup \{x\}$;

$J \leftarrow J \setminus \text{Overlap}(x)$;

}

3. Return A ;

Maintain a **binary min-heap** for J based on **finish time** as the key.

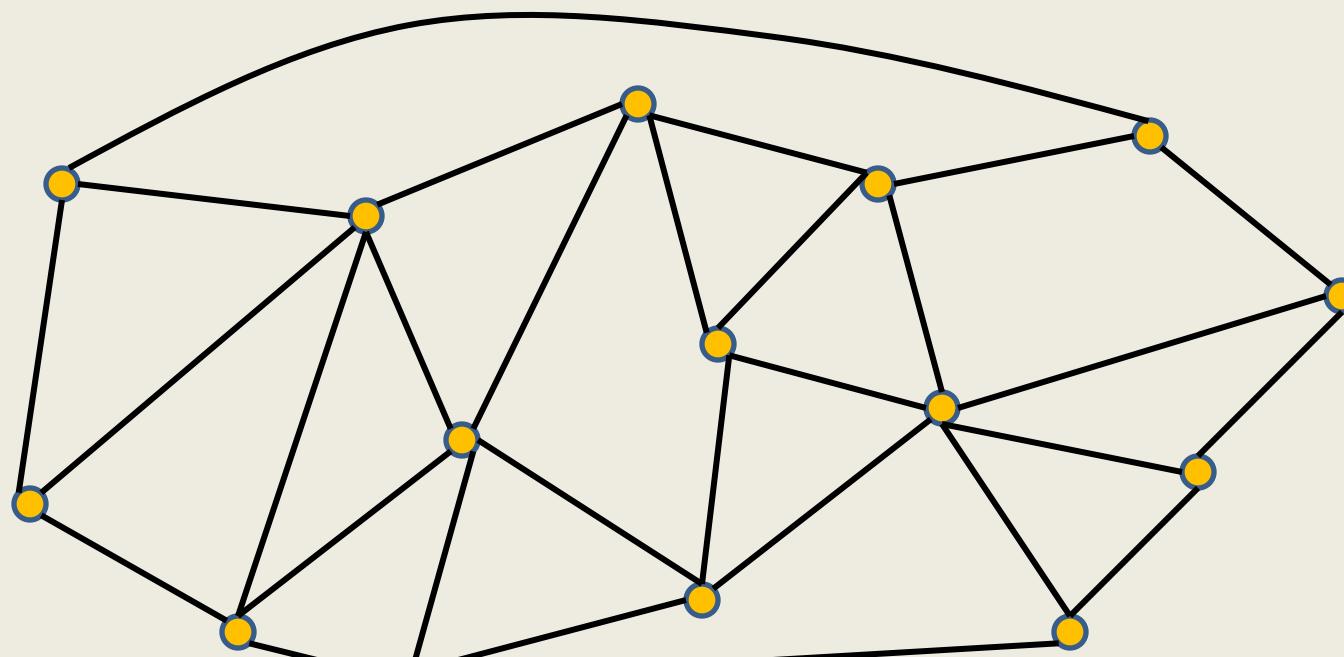
Sort J in increasing order of **start time**.

→ $O(n^2)$ time complexity is obvious

Problem 2

First we shall give motivation.

Motivation: A road or telecommunication network



Suppose there is a collection of possible links/roads that can be laid.
But laying down each possible link/road is costly.

Aim: To lay down **least number** of links/roads to ensure **connectivity** between each pair of nodes/cities.

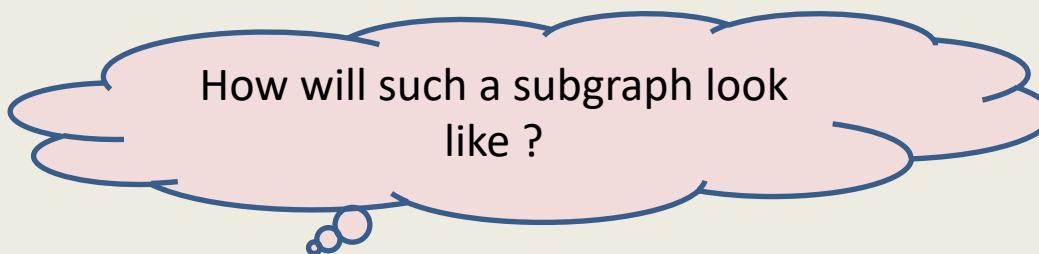
Motivation

Formal description of the problem

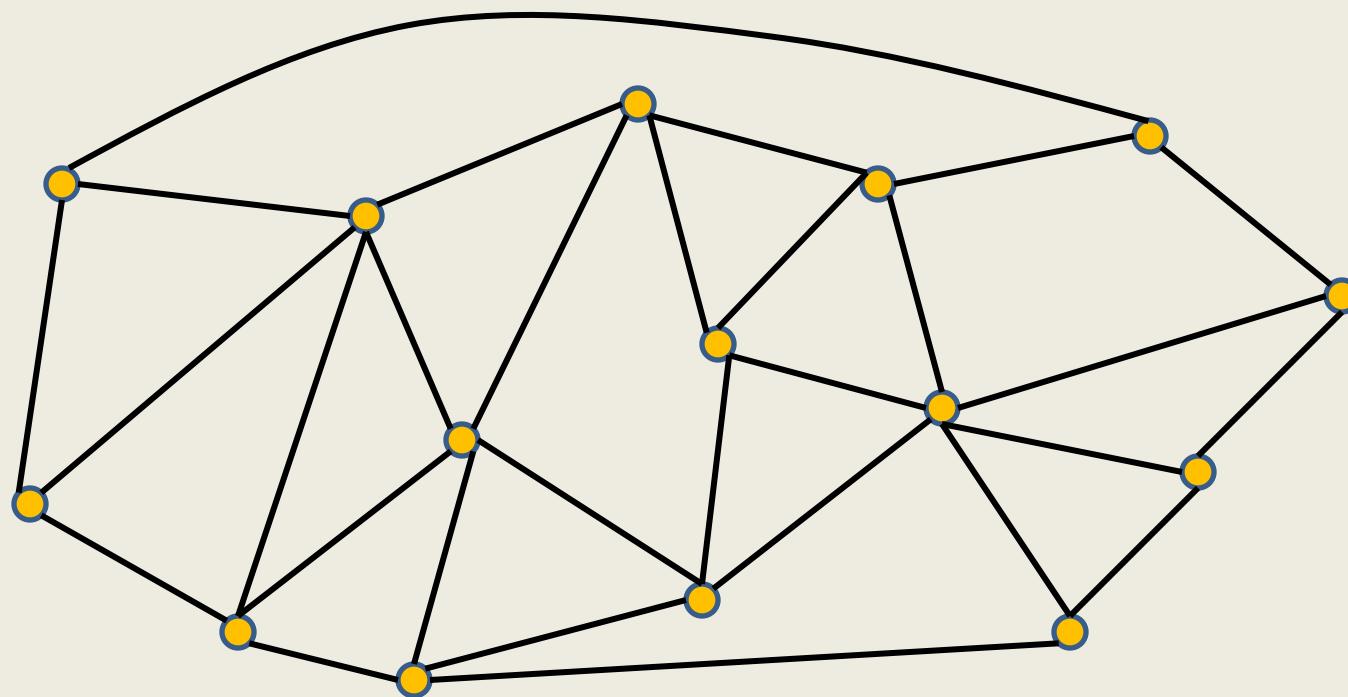
Input: an undirected graph $\mathbf{G}=(\mathbf{V},\mathbf{E})$.

Aim: compute a **subgraph** $(\mathbf{V}',\mathbf{E}')$, $\mathbf{E}' \subseteq \mathbf{E}$ such that

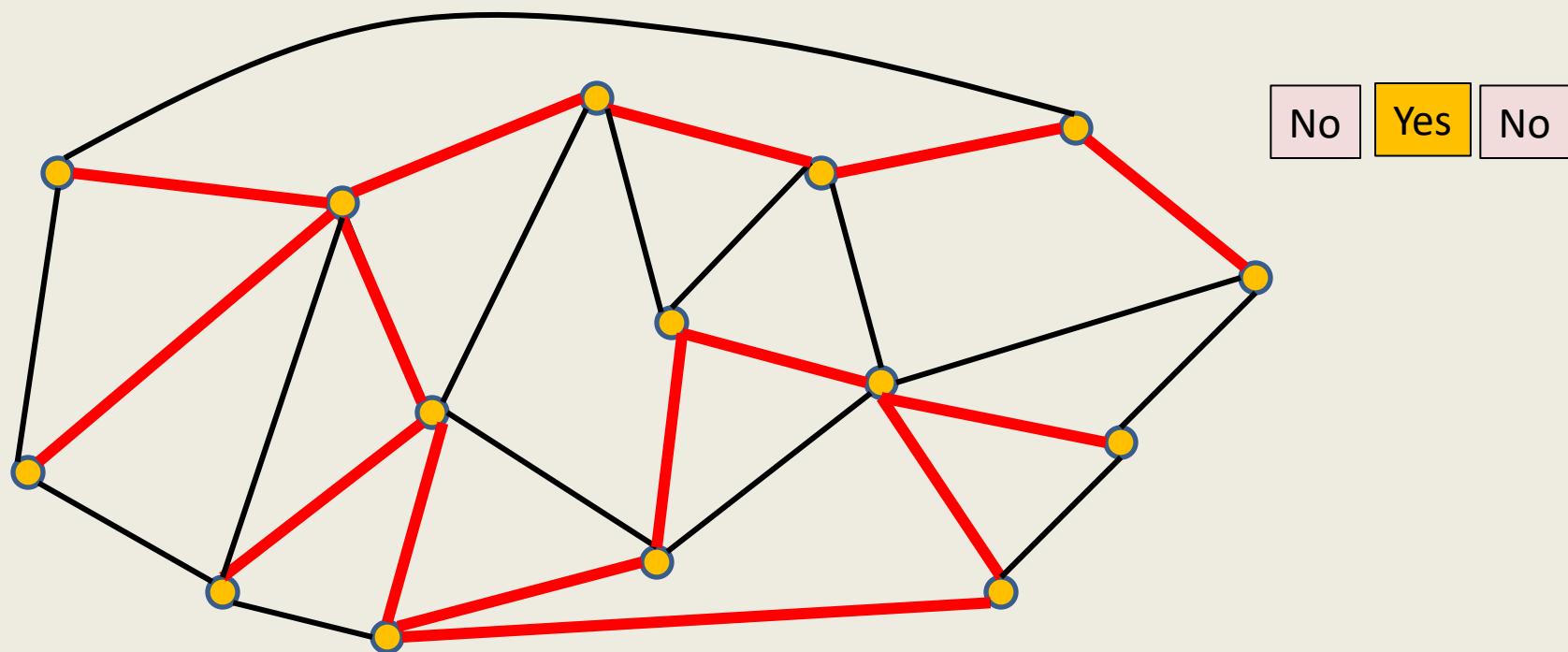
- **Connectivity** among all \mathbf{V} is guaranteed in the **subgraph**.
- $|\mathbf{E}'|$ is **minimum**.



A road or telecommunication network



A road or telecommunication network



Is this subgraph meeting our requirement ?

A tree

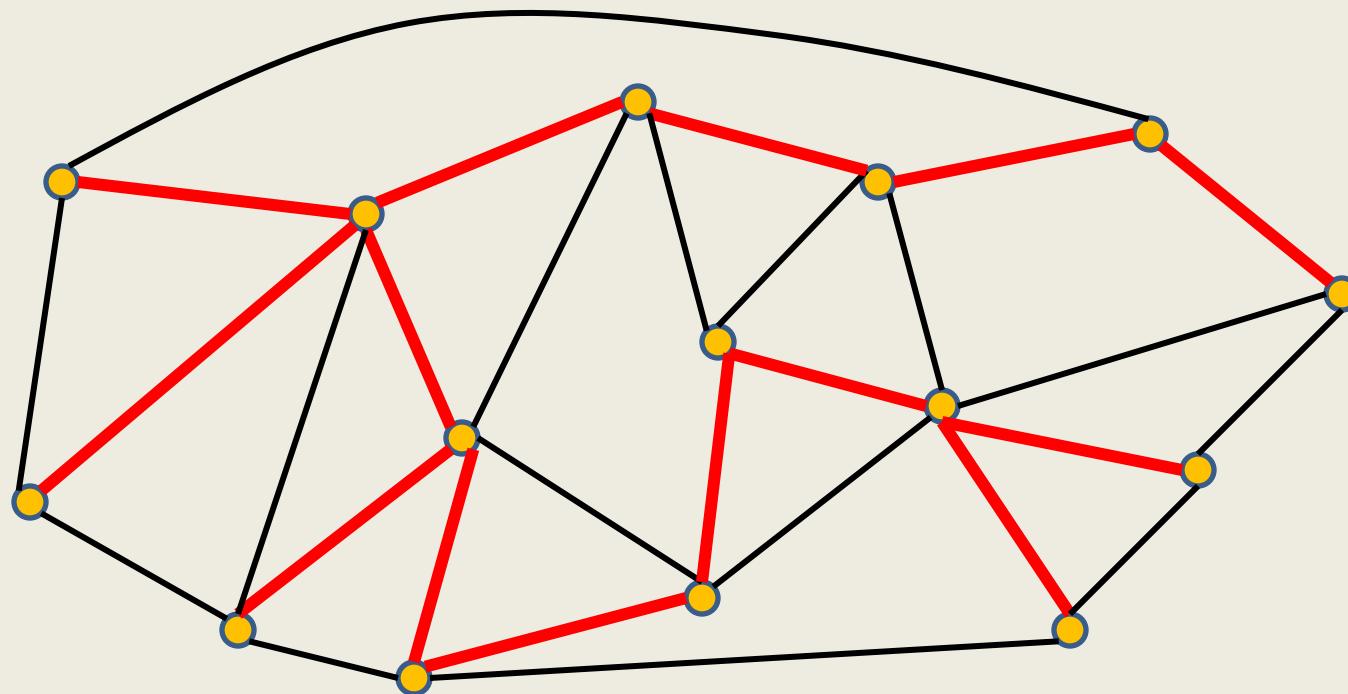
The following definitions are **equivalent**.

- An undirected graph which is **connected**
- An undirected graph where each pair of vertices has
- An undirected **connected** graph on n vertices and $n - 1$ edges
- An undirected graph on n vertices and $n - 1$ edges and

A Spanning tree

Definition: For an undirected graph (V, E) ,

A spanning tree is a **subgraph** (V, E') , $E' \subseteq E$ which is a tree.

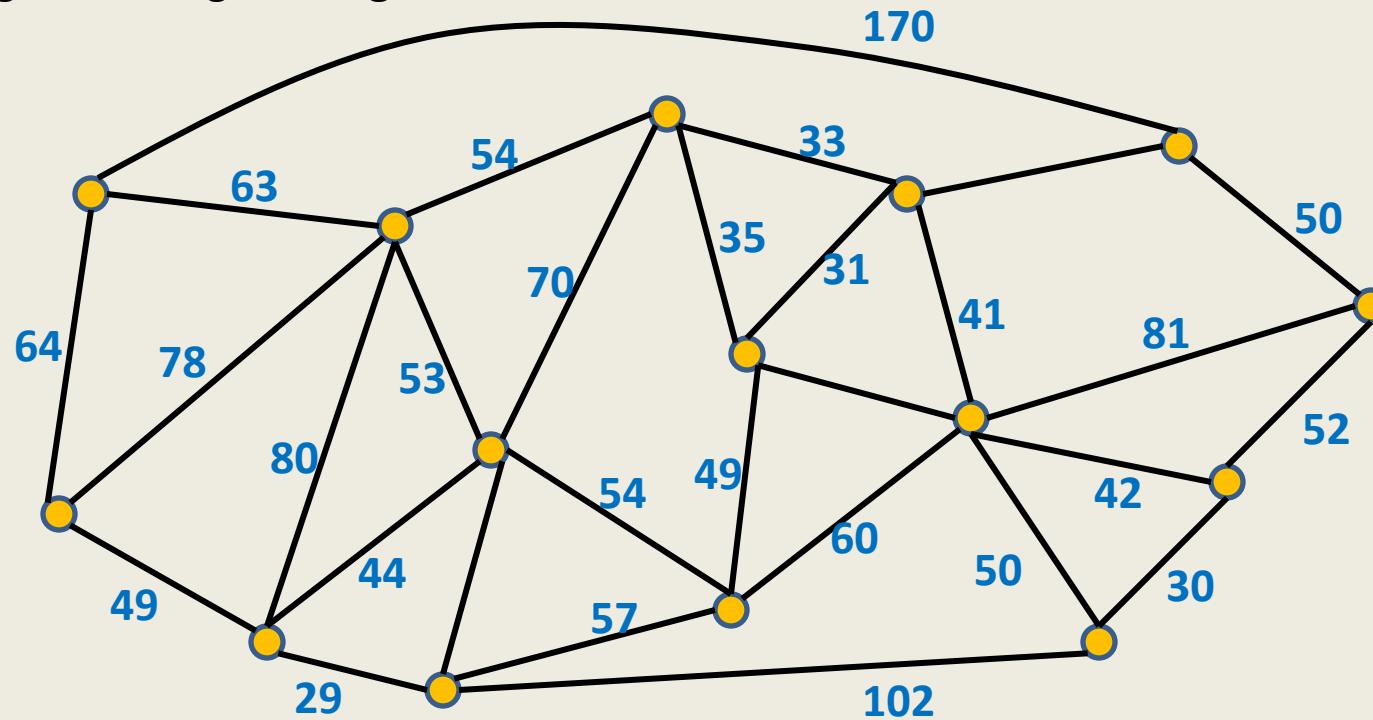


Observation: Given a spanning tree T of a graph G , adding a nontree edge e to T creates a unique cycle.

There will be total $m - n + 1$ such cycles. These are called **fundamental cycles** in G induced by the spanning tree T .

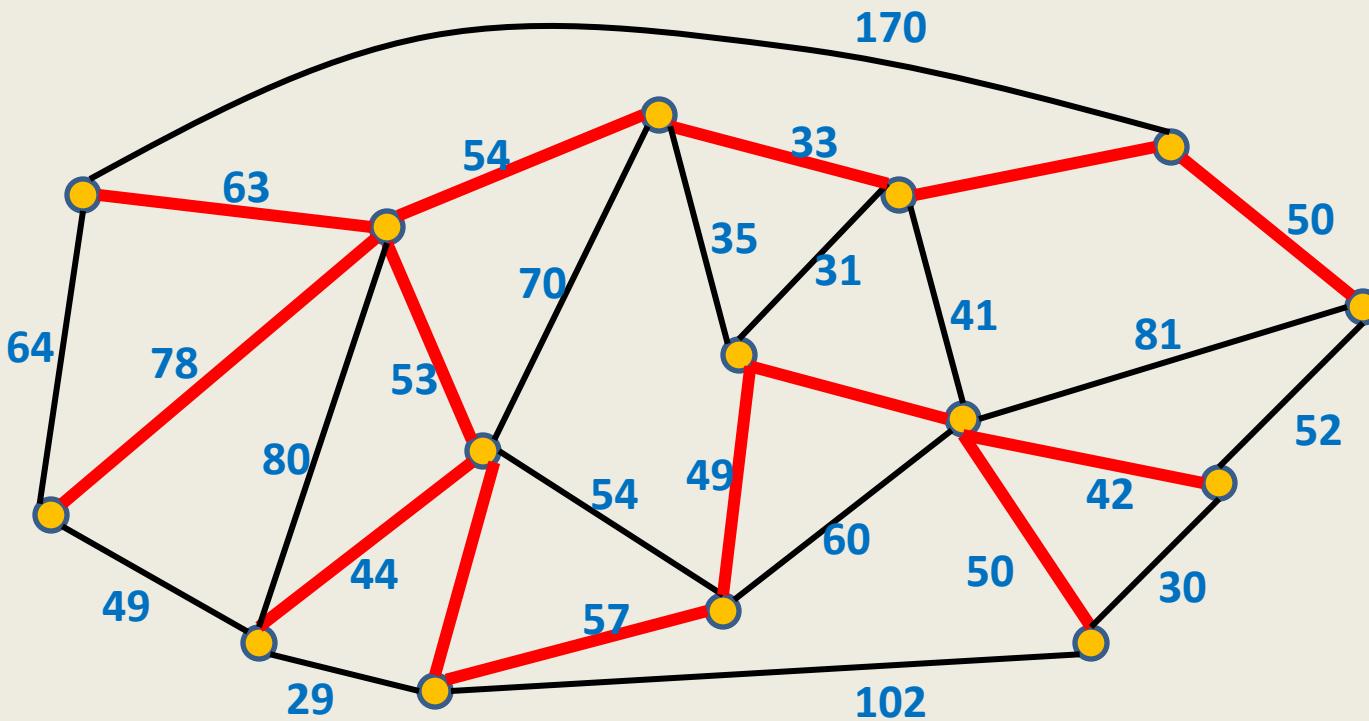
A road or telecommunication network

Assign each edge a **weight/cost**.



Adding more reality to the problem

A road or telecommunication network



Any arbitrary spanning tree (like the one shown above) will not serve our goal 😞.

We need to select the spanning tree with **least weight/cost**.

Problem 2

Minimum spanning tree

Problem Description

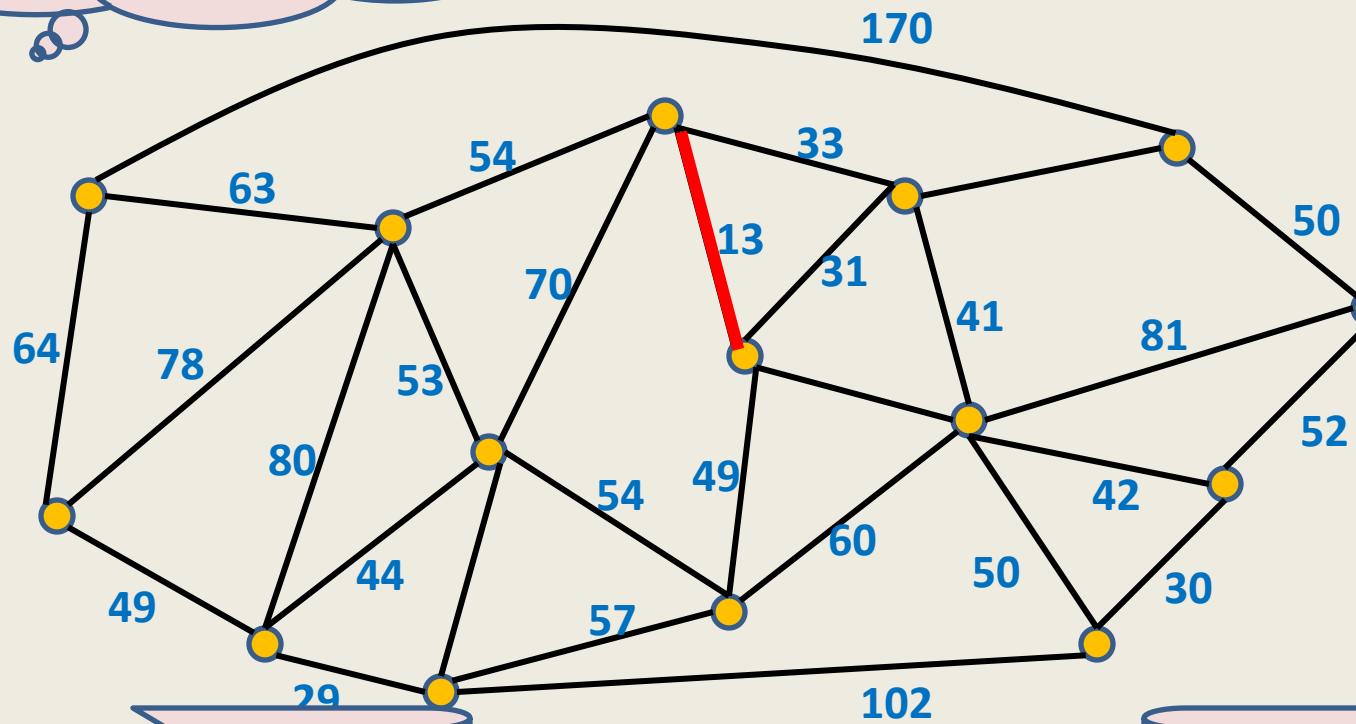
Input: an undirected graph $\mathbf{G}=(\mathbf{V}, \mathbf{E})$ with $w: \mathbf{E} \rightarrow \mathbb{R}$,

Aim: compute a **spanning tree** $(\mathbf{V}, \mathbf{E}')$, $\mathbf{E}' \subseteq \mathbf{E}$ such that

$\sum_{e \in \mathbf{E}'} w(e)$ is **minimum**.

How to compute a MST ?

The least weight edge
should be in MST.
But why ?



Look at the graph. Is there any edge for which you feel strongly to be present in MST ?

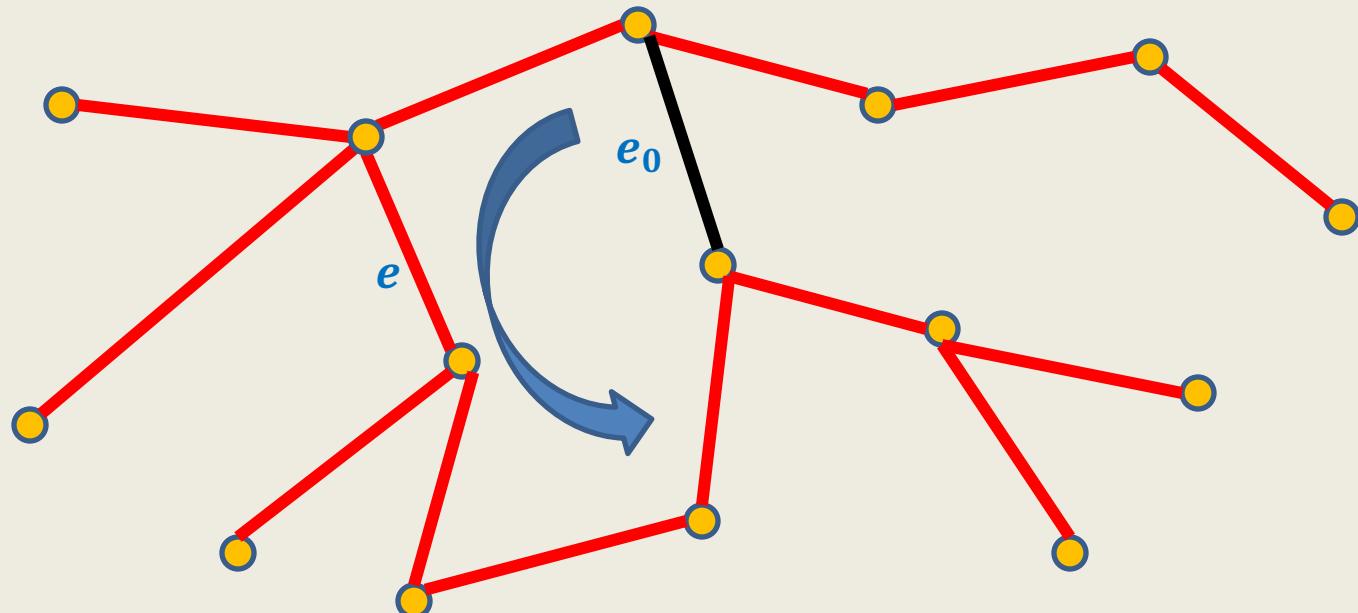
Let $e_0 \in E$ be the edge of least weight in the given graph.

Lemma2: There is a MST T containing e_0 .

Proof: Consider any MST T . Let $e_0 \notin T$.

Consider the fundamental cycle C defined by e_0 in T .

Swap e_0 with any edge $e \in T$ present in C .



Let $e_0 \in E$ be the edge of least weight in the given graph.

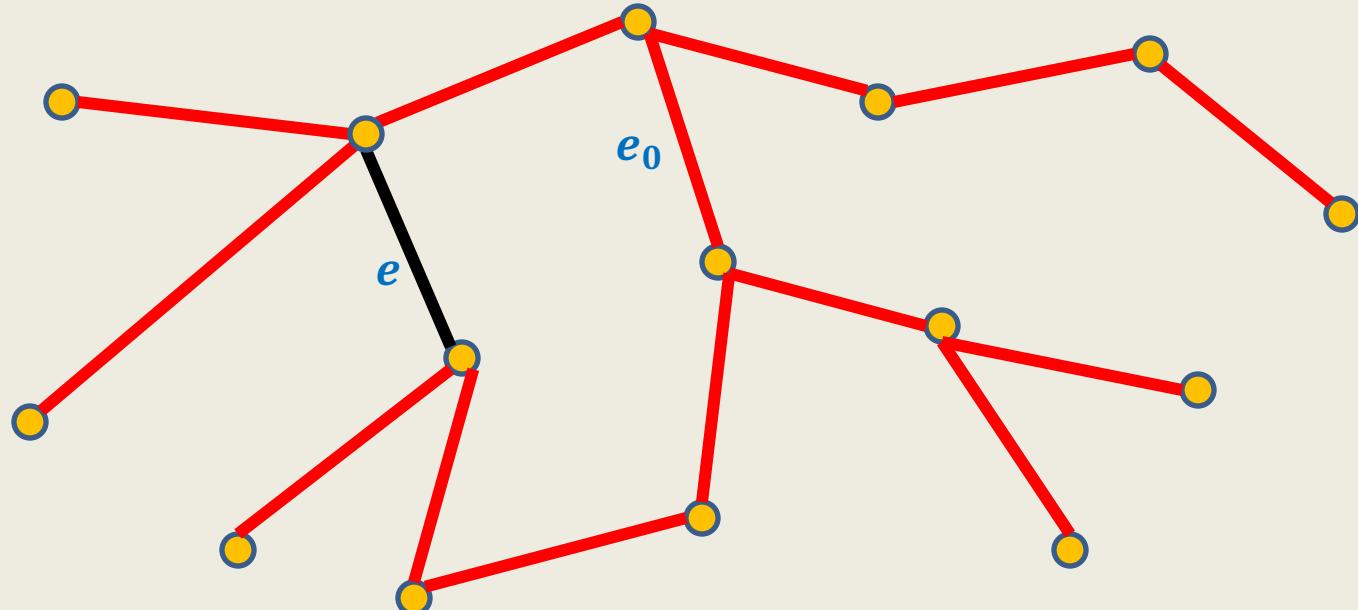
Lemma2: There is a MST T containing e_0 .

Proof: Consider any MST T . Let $e_0 \notin T$.

Consider the fundamental cycle C defined by e_0 in T .

Swap e_0 with any edge $e \in T$ present in C .

We get a spanning tree of weight $\leq w(T)$.



Try to translate Lemma2 to an algorithm for MST ?

with **inspiration** from the job scheduling problem ☺

Data Structures and Algorithms

(ESO207)

Lecture 36

- A new algorithm design paradigm: Greedy strategy
part III

Continuing Problem from last class

Minimum spanning tree

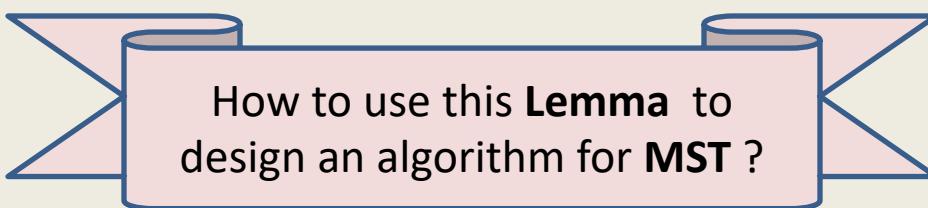
Problem Description

Input: an undirected graph $G=(V,E)$

Aim: compute a **spanning tree** (V, E')

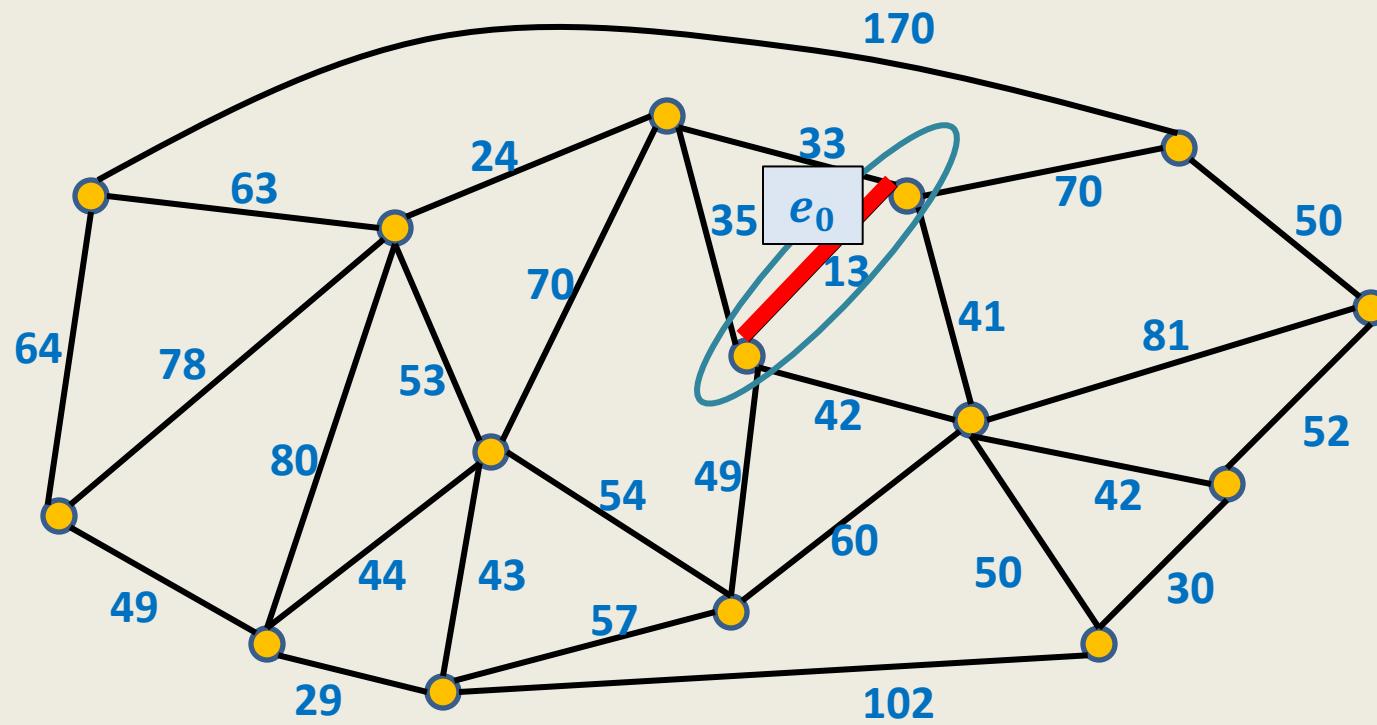
Lemma (proved in last class):

If $e_0 \in E$ is the edge of **least weight** in G ,
then there is a **MST** T containing e_0 .

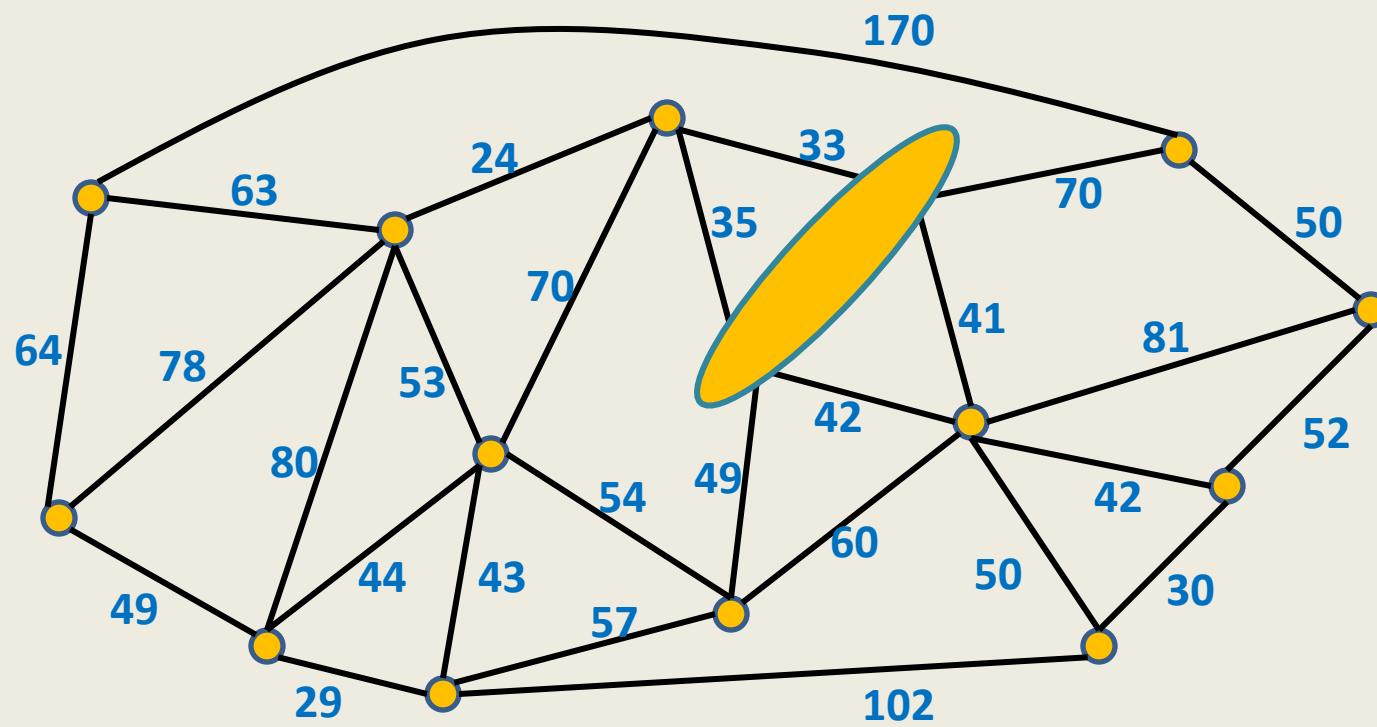


How to use this **Lemma** to
design an algorithm for **MST** ?

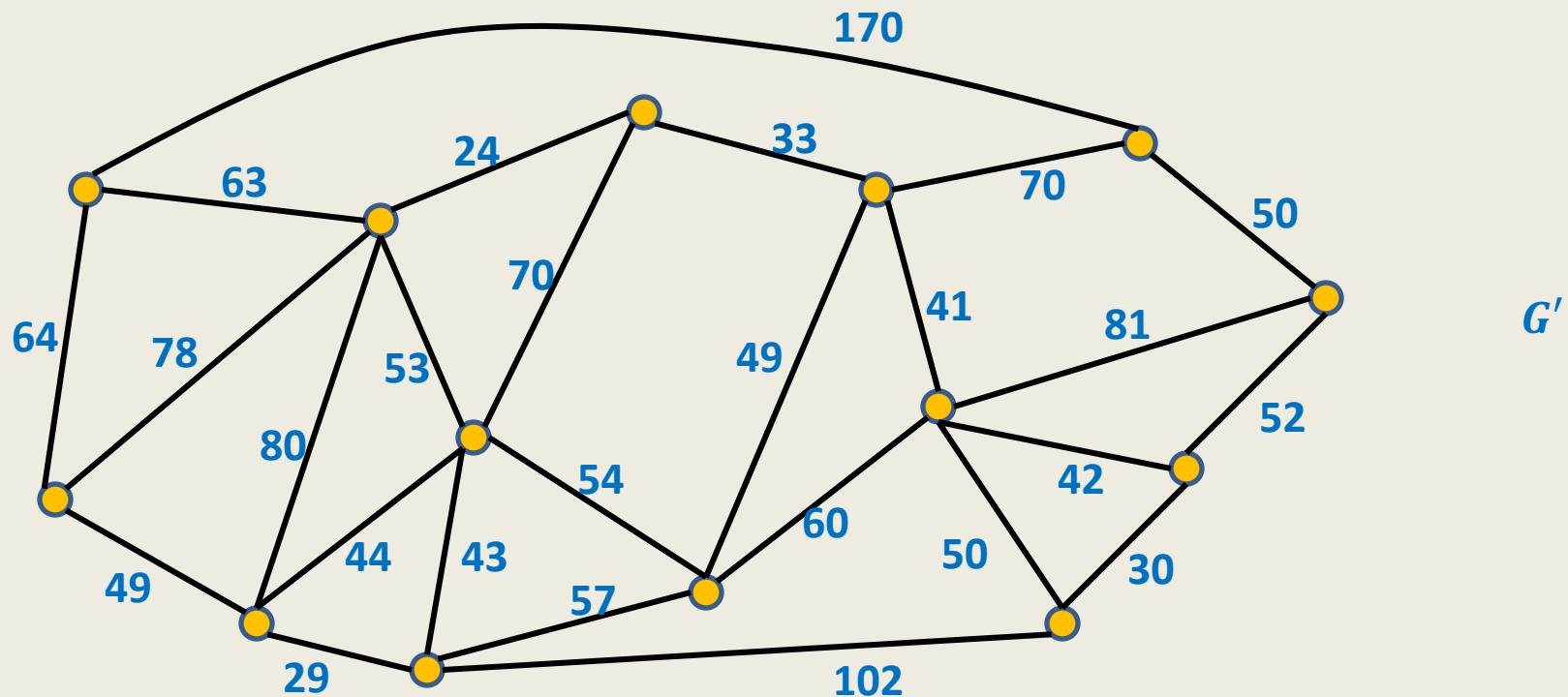
Minimum Spanning Tree (MST)



Minimum Spanning Tree (MST)



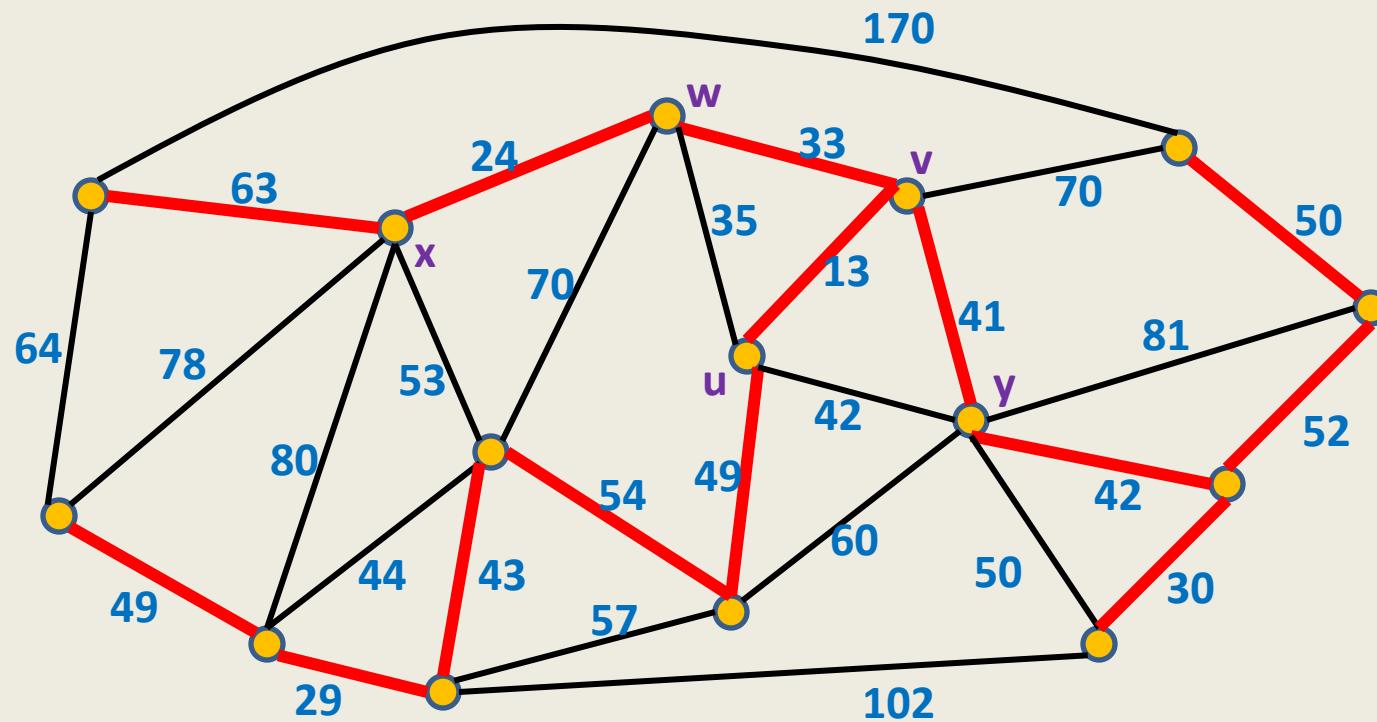
Minimum Spanning Tree (MST)



Theorem:

$$w(\text{MST}(G)) =$$

Minimum Spanning Tree (MST)



A useful lesson for design of a graph algorithm

If you have a complicated algorithm for a graph problem, ...

- search for **some graph theoretic property**

to design simpler and more efficient algorithm

Two graph theoretic properties of MST

- Cut property
- Cycle property



Every algorithm till date is based on
one of these properties!

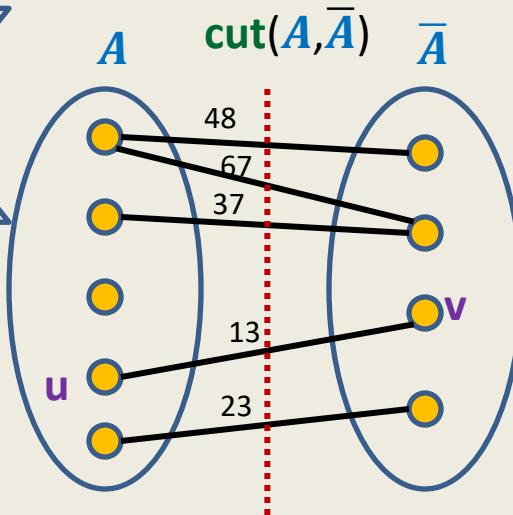
Cut Property

Cut Property

Definition: For any subset $A \subseteq V$

$$\text{cut}(A, \bar{A}) = \{ (u, v) \in E \mid$$

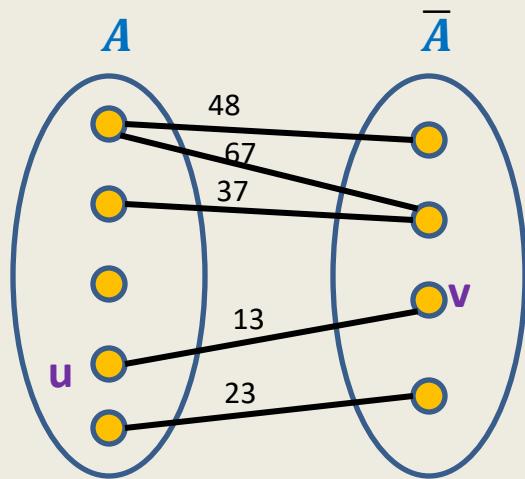
Pursuing **greedy strategy** to minimize weight of MST, what can we say about the edges of $\text{cut}(A, \bar{A})$?



Cut-property:

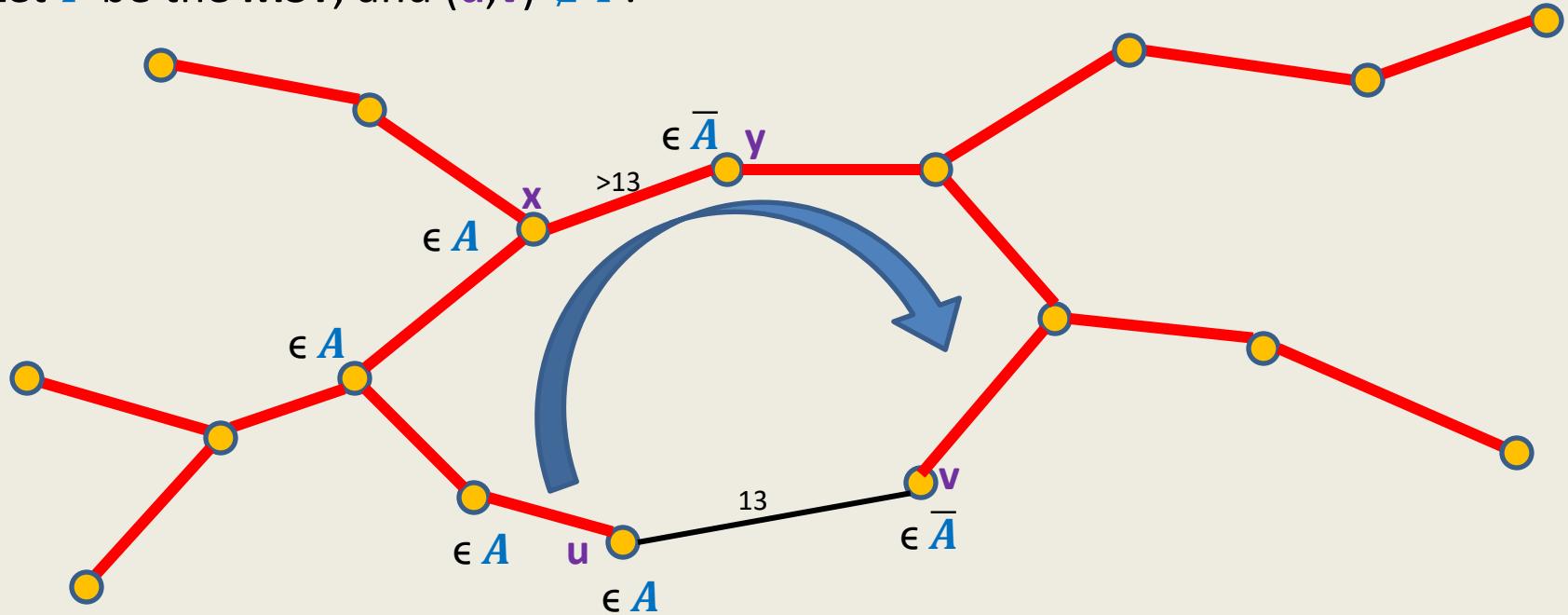
The **least weight edge** of a $\text{cut}(A, \bar{A})$ must be in **MST**.

Proof of cut-property



Proof of cut-property

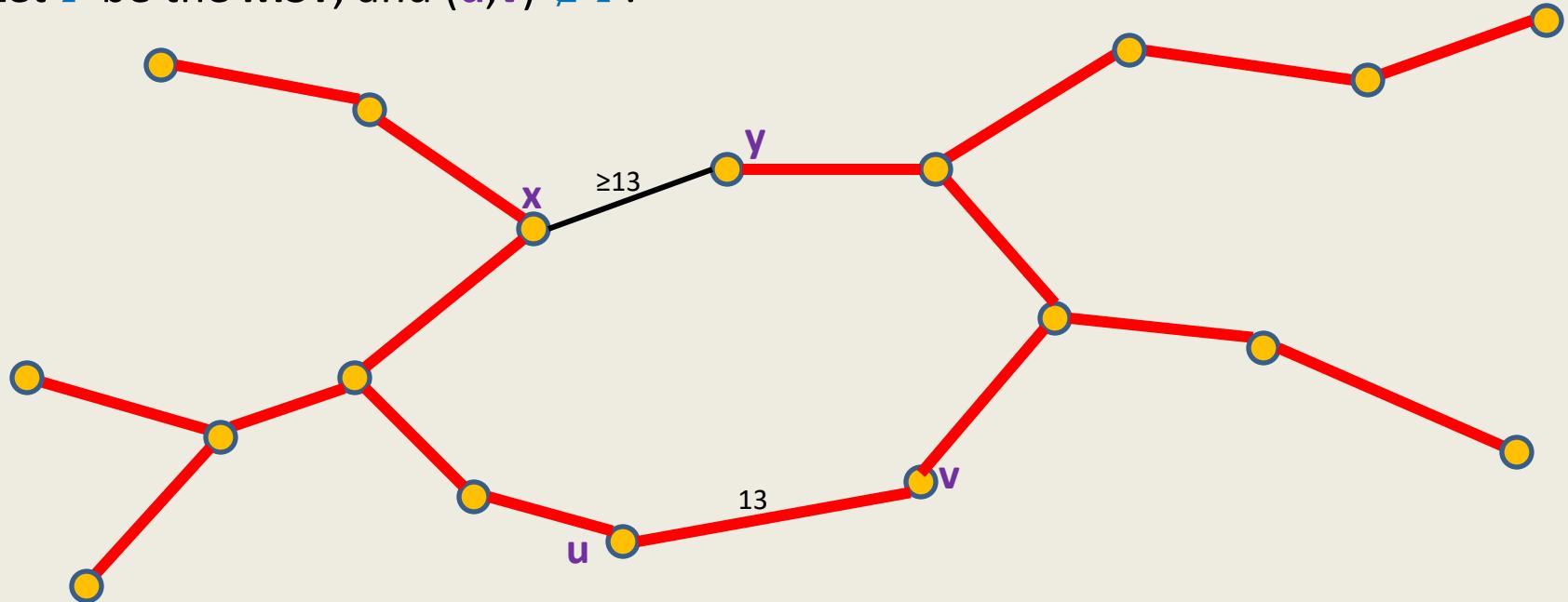
Let T be the MST, and $(u, v) \notin T$.



Question: What happens if we remove (x, y) from T , and add (u, v) to T .

Proof of cut-property

Let T be the MST, and $(u, v) \notin T$.



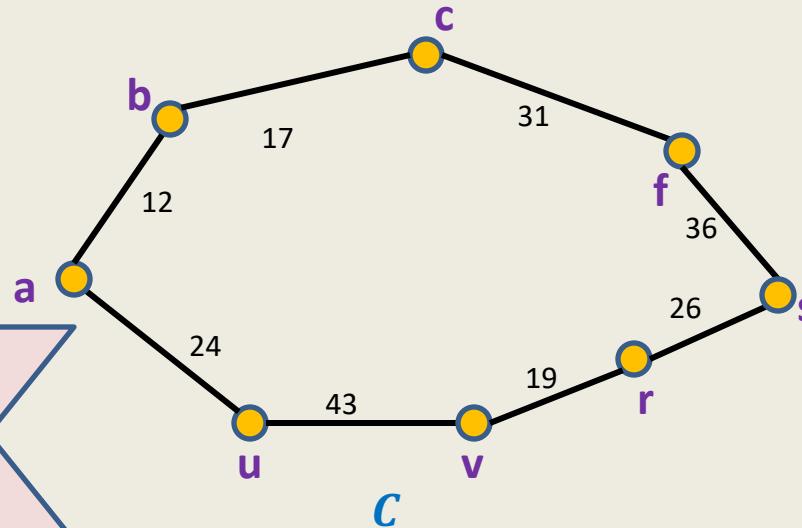
Question: What happens if we remove (x, y) from T , and add (u, v) to T .

We get a spanning tree T' with weight < $\text{weight}(T)$
A contradiction !

Cycle Property

Cycle Property

Let \mathcal{C} be any cycle in G .

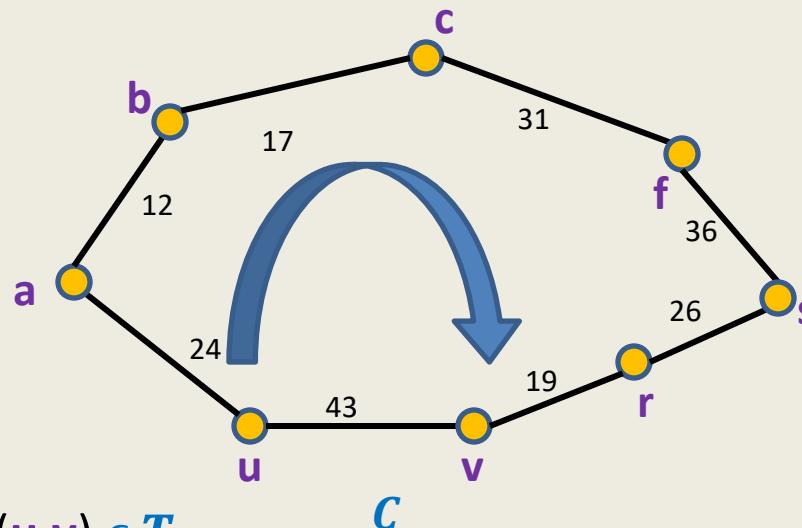


Pursuing **greedy strategy** to minimize weight of MST, what can we say about the edges of cycle \mathcal{C} ?

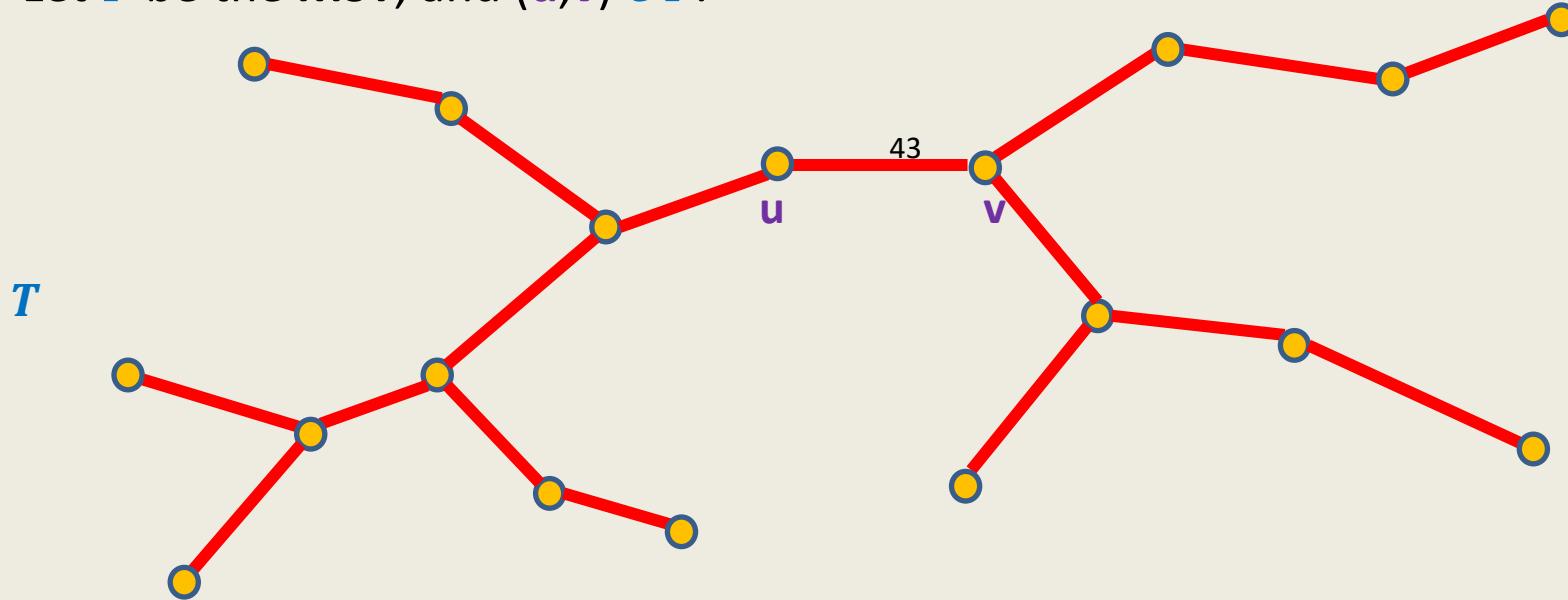
Cycle-property:

Maximum weight edge of any cycle \mathcal{C} **can not** be present in **MST**.

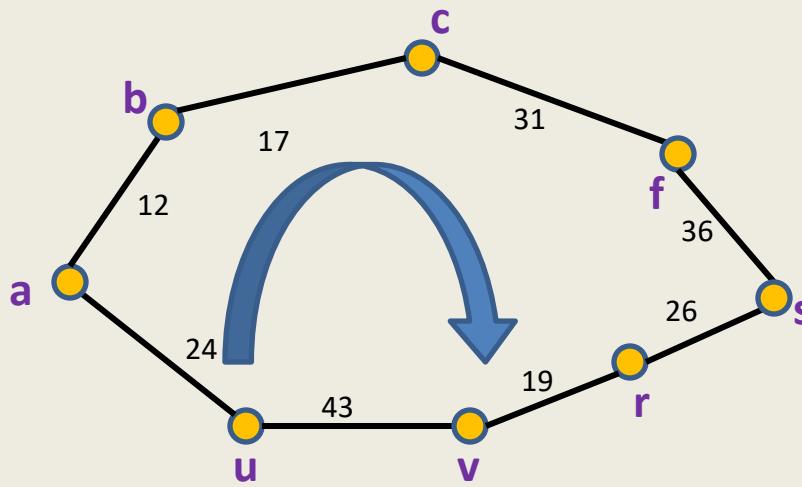
Proof of Cycle property



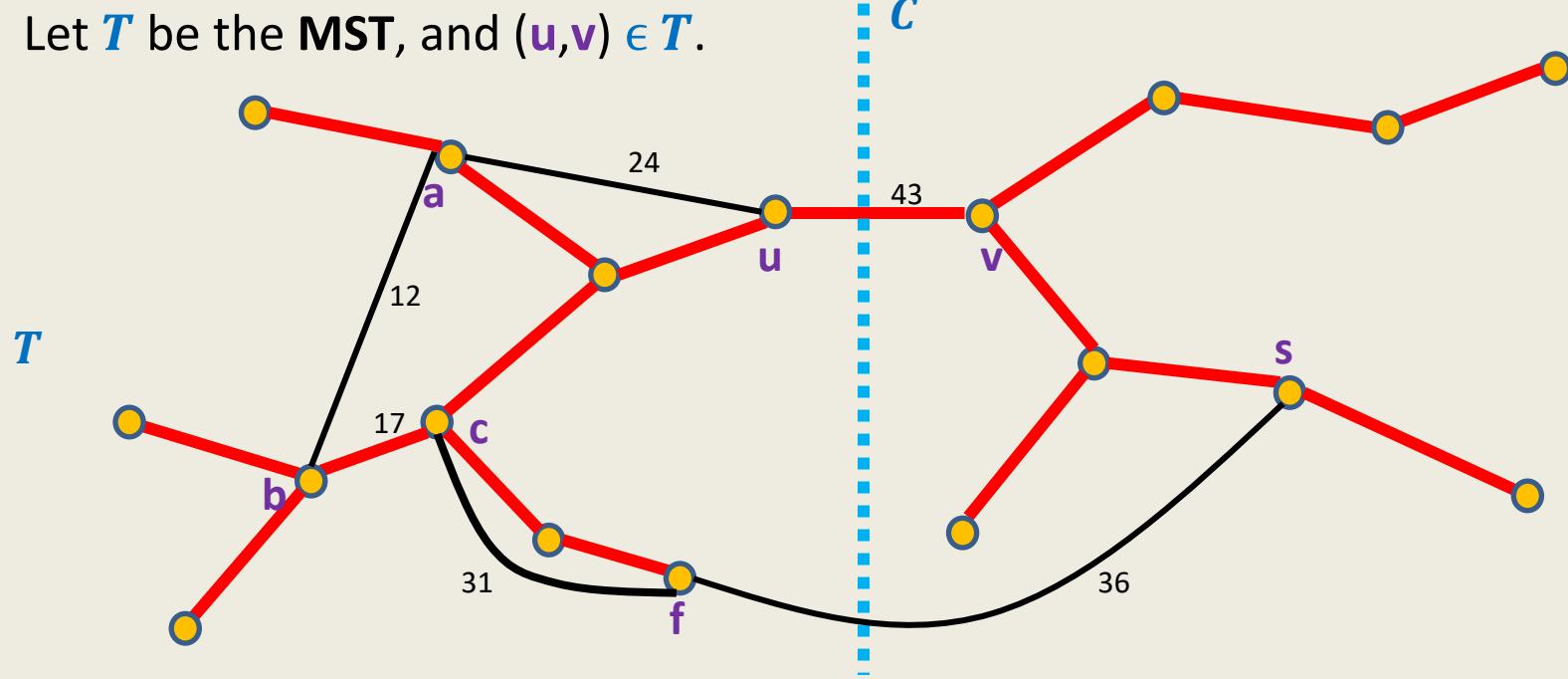
Let T be the MST, and $(u,v) \in T$.



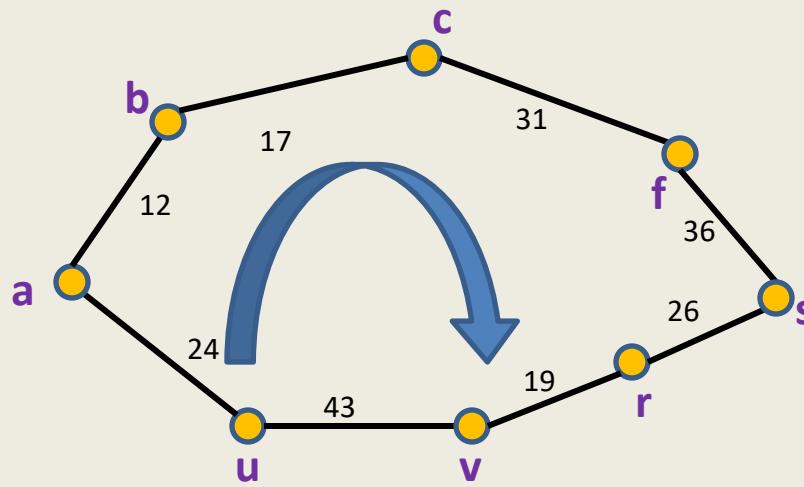
Proof of Cycle property



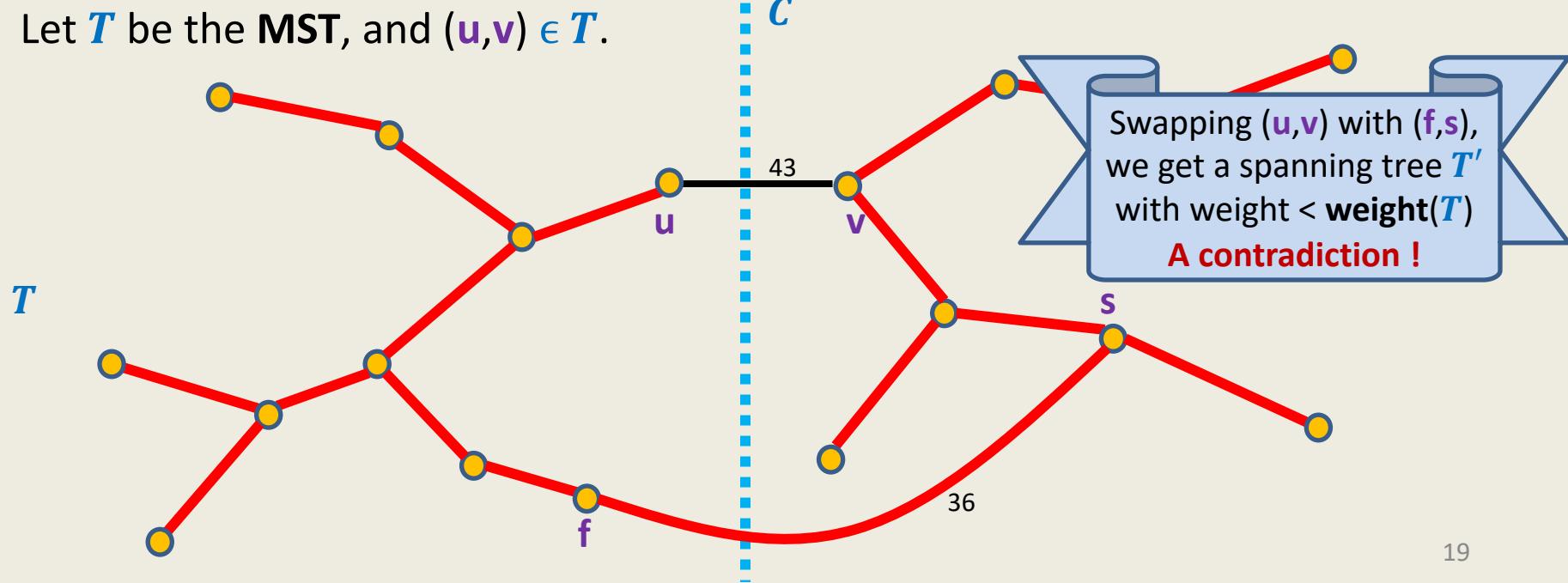
Let T be the MST, and $(u,v) \in T$.



Proof of Cycle property

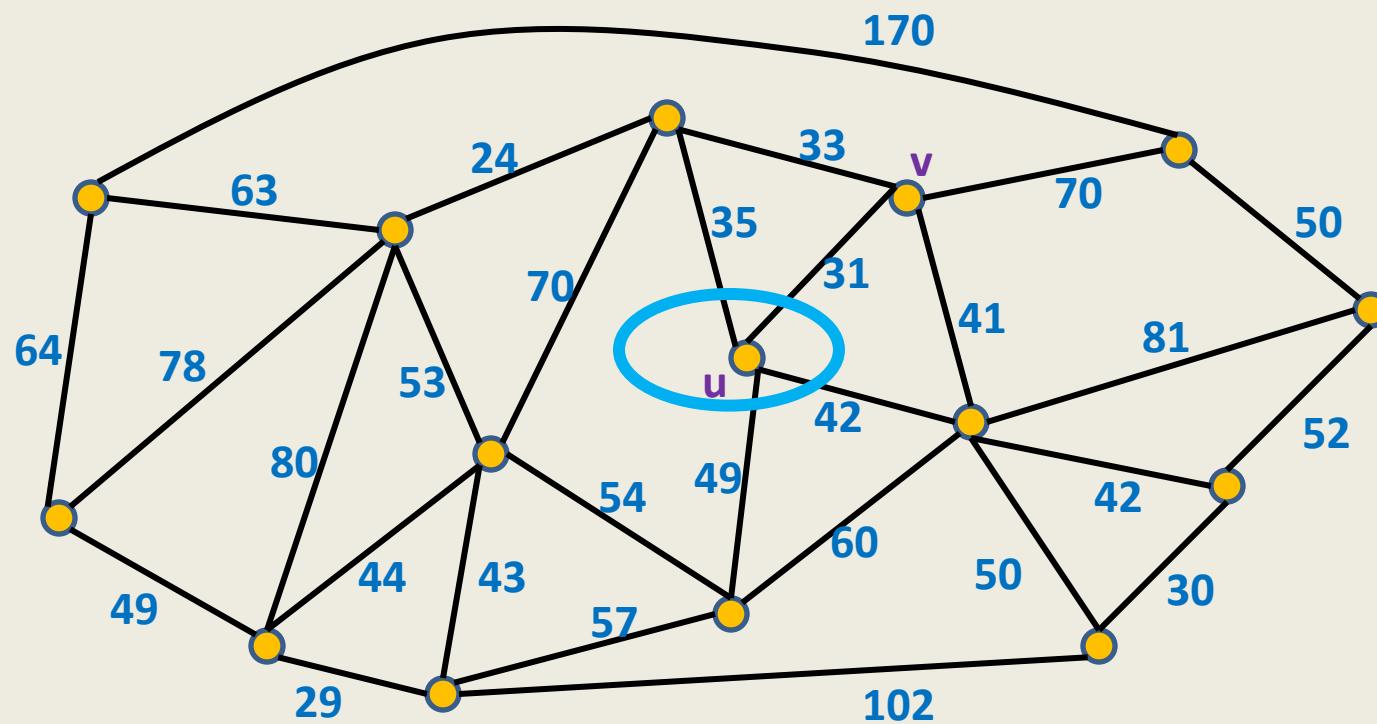


Let T be the MST, and $(u,v) \in T$.

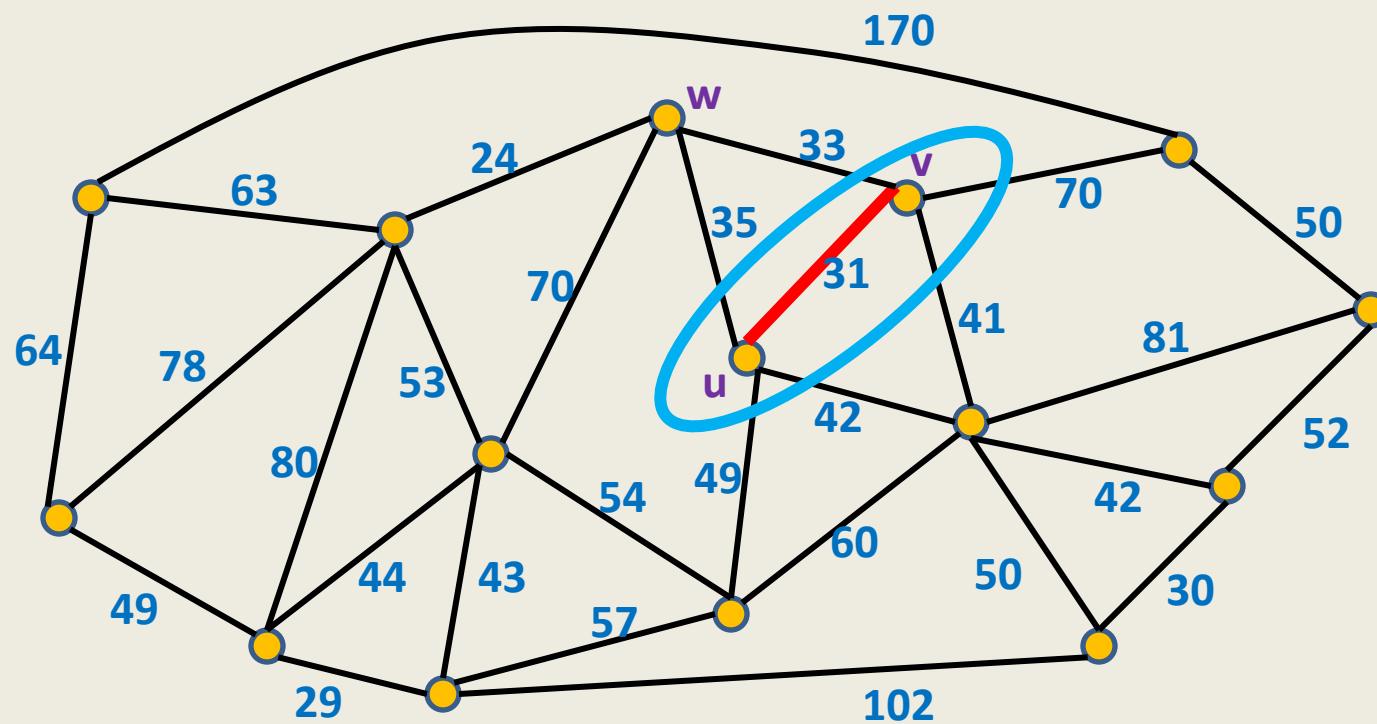


Algorithms based on cut Property

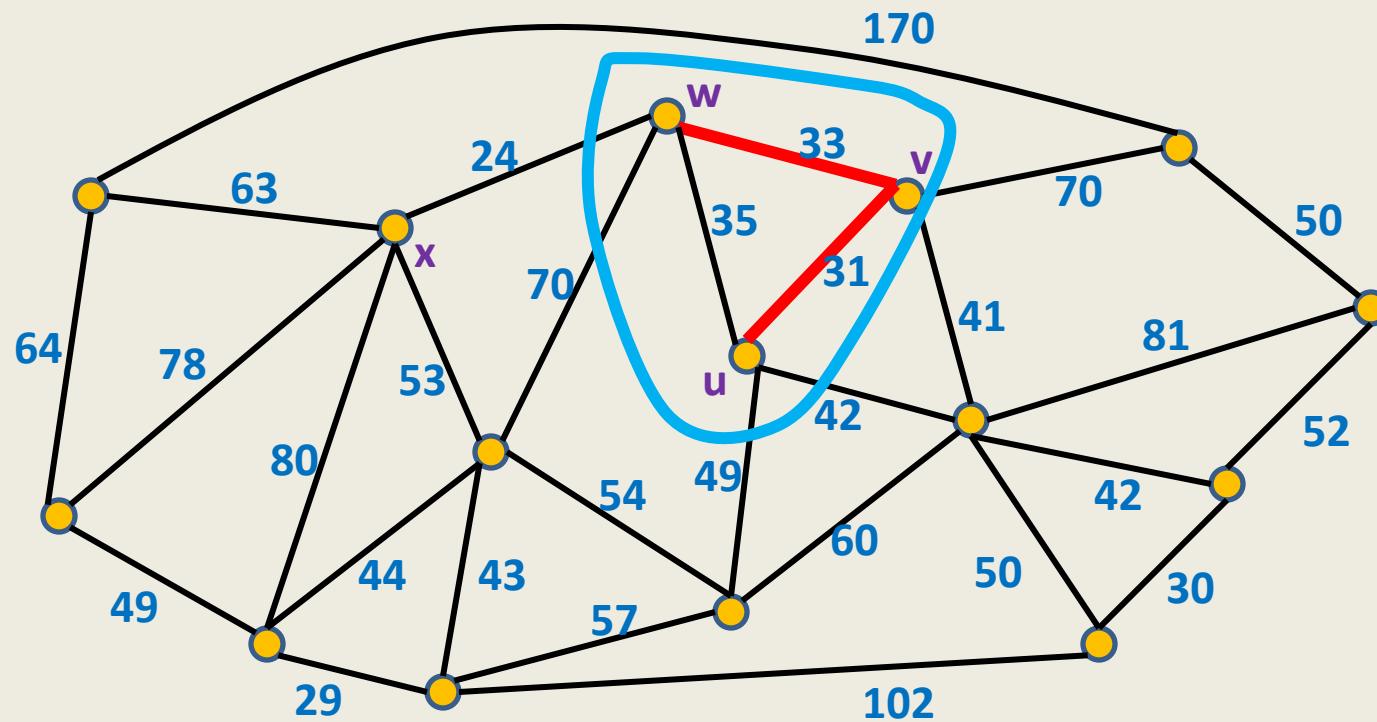
How to use cut property to compute a MST ?



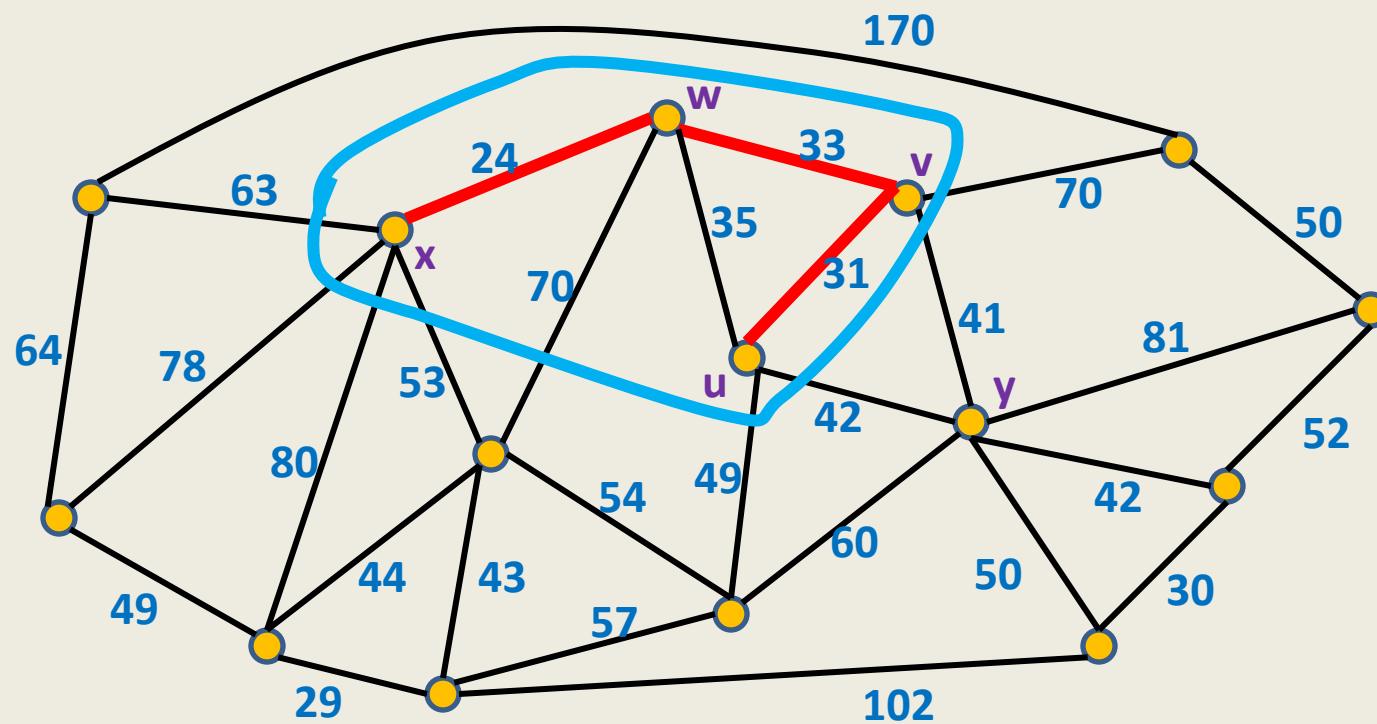
How to use cut property to compute a MST ?



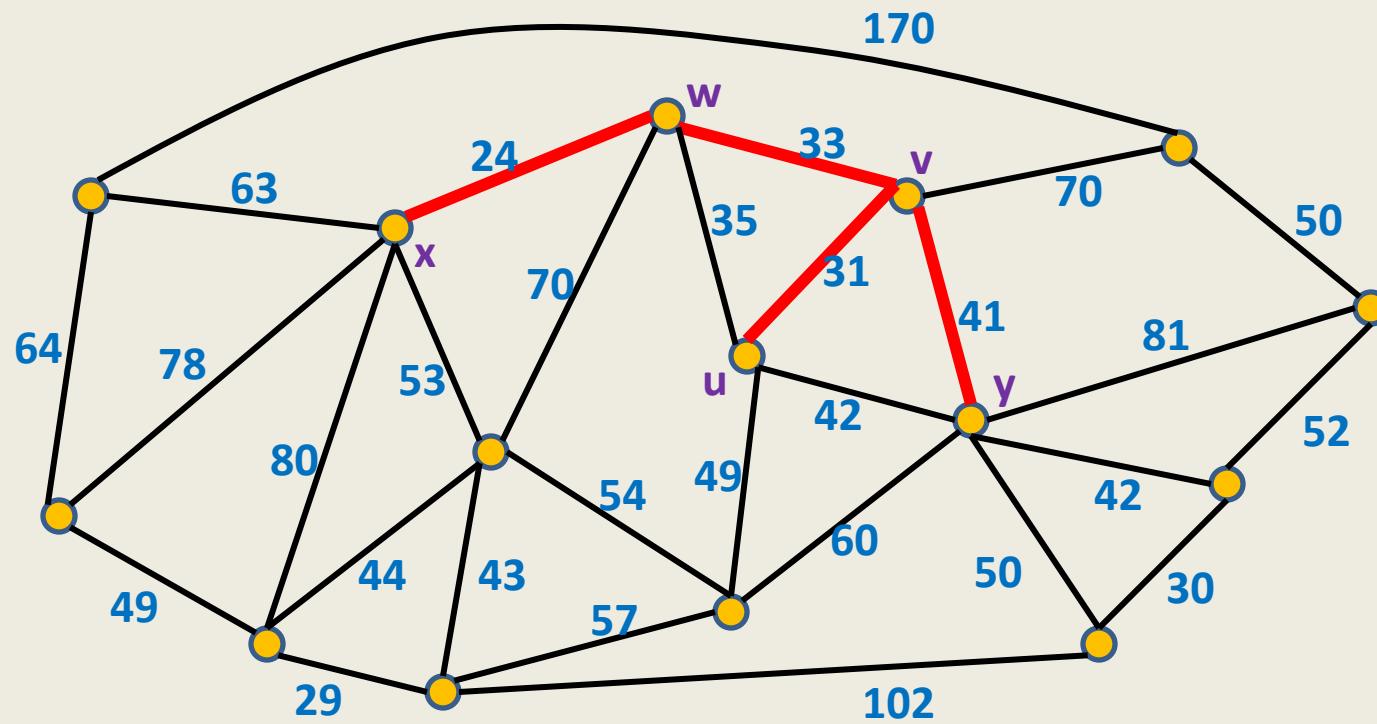
How to use cut property to compute a MST ?



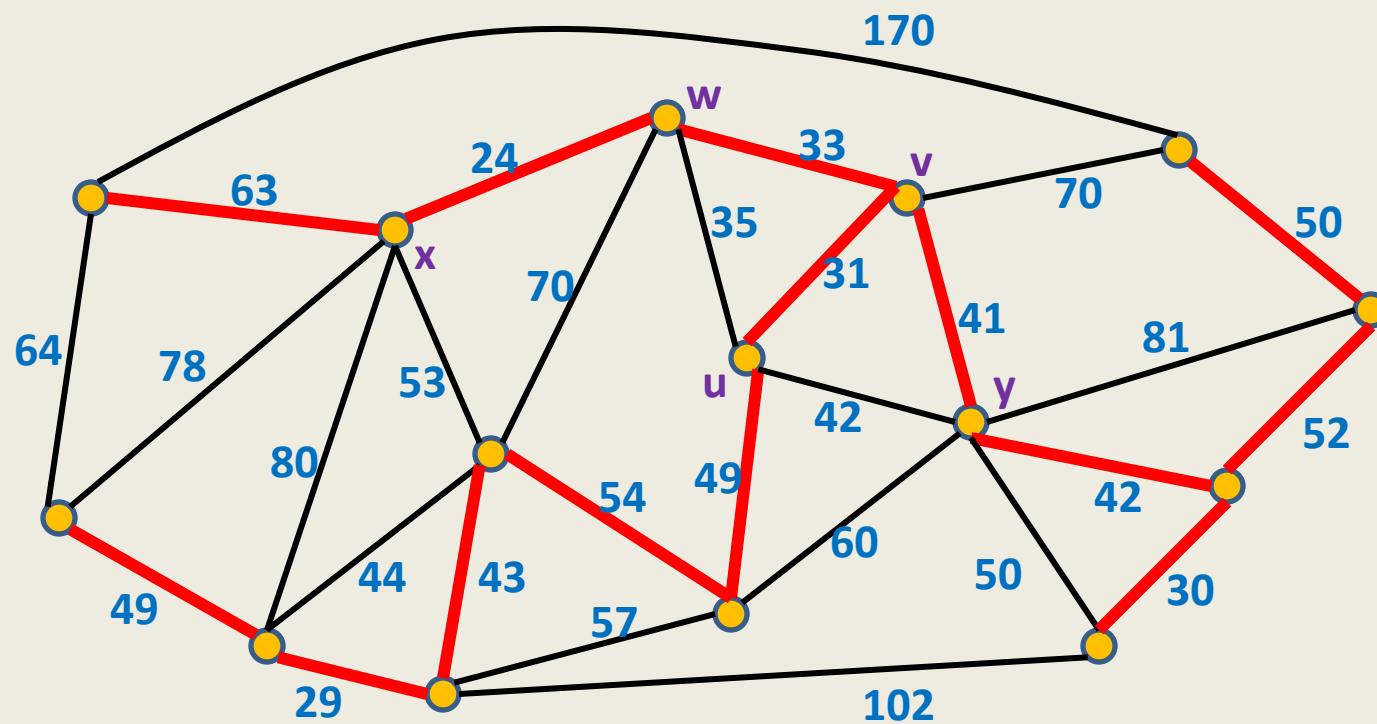
How to use cut property to compute a MST ?



How to use cut property to compute a MST ?



How to use cut property to compute a MST ?



An Algorithm based on **cut** property

Algorithm (Input: graph $G = (V, E)$ with **weights** on edges)

$T \leftarrow \emptyset;$

$A \leftarrow \{u\};$

While ($A <> V$) **do**

{ Compute the least weight edge from $\text{cut}(A, \bar{A})$;

Let this edge be (x, y) , with $x \in A, y \in \bar{A}$;

$T \leftarrow T \cup \{(x, y)\};$

$A \leftarrow A \cup \{y\};$

}

Return T ;

Number of iterations of the **While** loop : $n - 1$

Time spent in one iteration of While loop: $O(m)$

→ Running time of the algorithm: $O(mn)$

Algorithm based on cycle Property

An Algorithm based on cycle property

Description

Algorithm (Input: graph $G = (V, E)$ with weights on edges)

While (E has any cycle) do

{ Compute any cycle C ;

Let (u, v) be the maximum weight edge of the cycle C ;

Remove (u, v) from E ;

}

Return E ;

Number of iterations of the While loop : $m - n + 1$

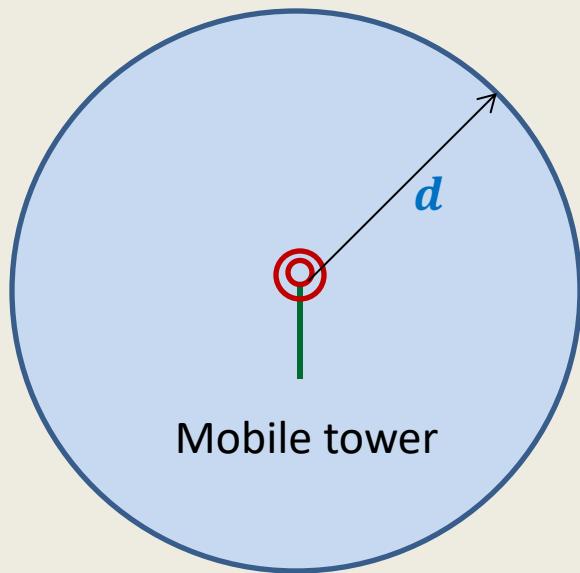
Time spent in one iteration of While loop: $O(n)$

→ Running time of the algorithm: $O(mn)$

Problem 3

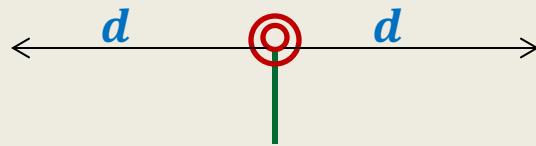
Mobile towers on a road

Mobile towers on a road



A mobile tower can cover any cell phone within radius d .

Mobile towers on a road



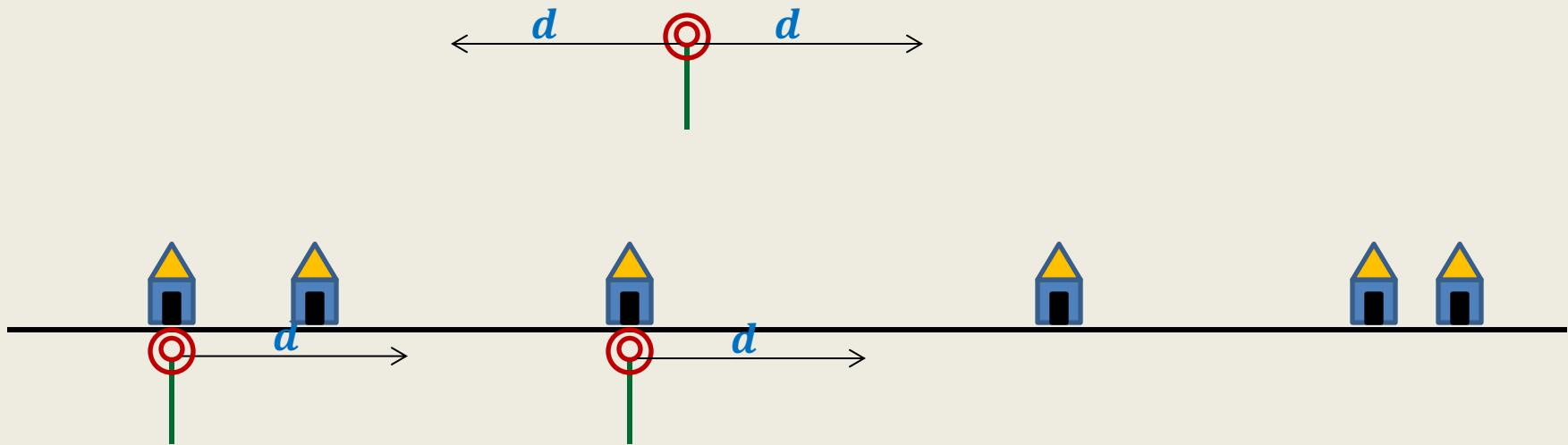
Problem statement:

There are n houses located along a road.

We want to place mobile towers such that

- Each house is covered by at least one mobile tower.
- The number of mobile towers used is **least** possible.

Mobile towers on a road



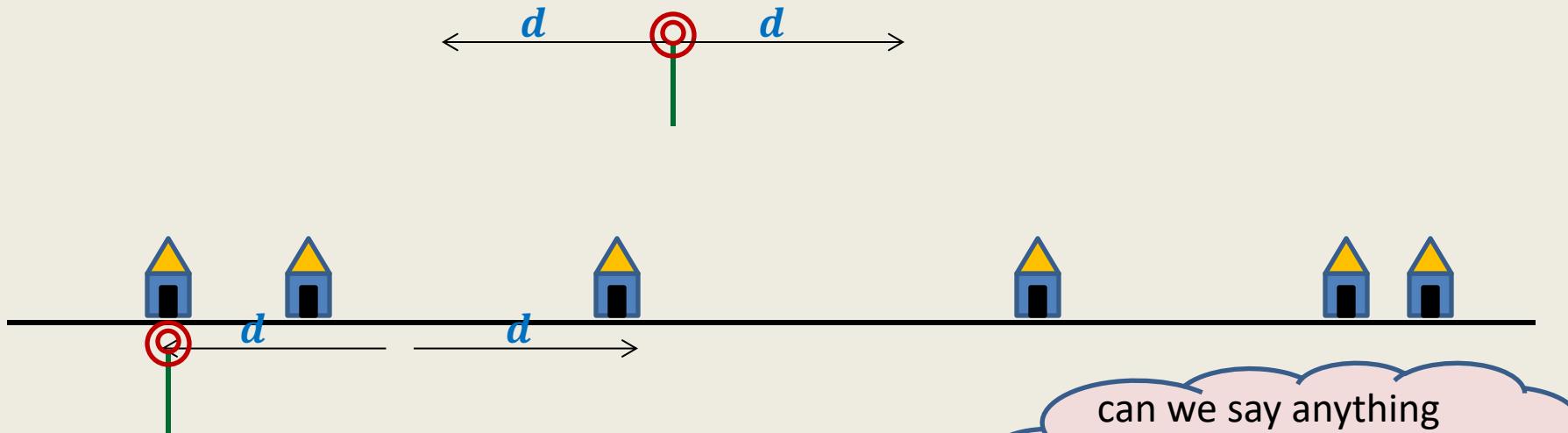
Strategy 1:

Place tower at first house,

Remove all houses covered by this tower.

Proceed to the next uncovered house ...

Mobile towers on a road



Strategy 2:

Place tower at distance d to the right of the first house;

Remove all houses covered by this tower;

Proceed to the next uncovered house along the road...

Lemma: There is an optimal solution for the problem in which the leftmost tower is placed at distance d to the right of the first house

can we say anything
about the optimal
solution ?

Homework ...

Ponder over the following questions before coming for the next class

- Use **cycle property** and/or **cut property** to design a **new algorithm for MST**
- Use some data structure to improve the running time of the algorithms discussed in this class to **$O(m \log n)$**

Data Structures and Algorithms

(ESO207)

Lecture 37

- A new algorithm design paradigm: Greedy strategy
part IV

Problems solved till now

1. Job Scheduling Problem

2. Mobile Tower Problem

3. MST



Did you notice
anything common in
their solutions ?

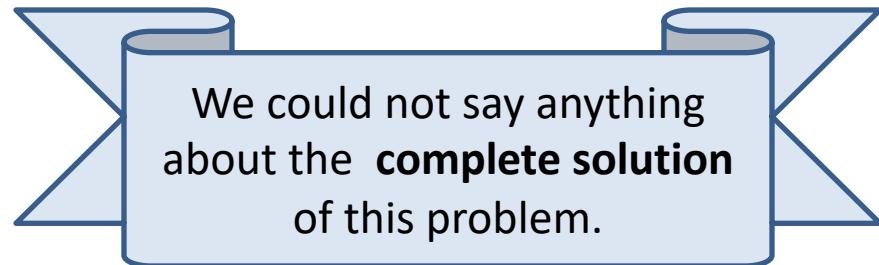
Ponder over this question before moving ahead

Problem 1

Job scheduling Problem

INPUT:

- A set J of n jobs $\{j_1, j_2, \dots, j_n\}$
- job j_i is specified by two real numbers
 - $s(i)$: start time of job j_i
 - $f(i)$: finish time of job j_i
- A single server



We could not say anything about the **complete solution** of this problem.

Constraints:

- Server can execute at most one job at any moment of time and a job.
- **Job j_i** , if scheduled, has to be scheduled during $[s(i), f(i)]$ only.

Aim:

To select the **largest** subset of non-overlapping jobs which can be executed by the server.

Problem 1

Job scheduling Problem

INPUT:

- A set J of n jobs $\{j_1, j_2, \dots, j_n\}$
- job j_i is specified by two real numbers
 - $s(i)$: start time of job j_i
 - $f(i)$: finish time of job j_i
- A single server

Constraints:

- Server can execute at most one job at any moment of time and a job.
- **Job j_i** , if scheduled, has to be scheduled during $[s(i), f(i)]$ only.

Aim:

To select the **largest** subset of non-overlapping jobs which can be executed by the server.

All that we could do was to make a local observation

Let $x \in J$ be the job with earliest finish time.

Lemma1 : There exists an optimal solution for J in which x is present.

Let $J' = J \setminus \text{Overlap}(x)$

Lemma 1 gives very small information
about the optimal solution ☹
How to use it to compute this solution ?

J (original instance)

$\text{Opt}(J)$

$$\text{Opt}(J) = \text{Opt}(J') + 1 \quad \text{-- (i)}$$

Greedy
step

J' (smaller instance)

Lemma1

$\text{Opt}(J')$

Equation (i) hints at recursive solution of the problem ☺

Theorem: $\text{Opt}(J) = \text{Opt}(J') + 1$.

- Proof has two parts

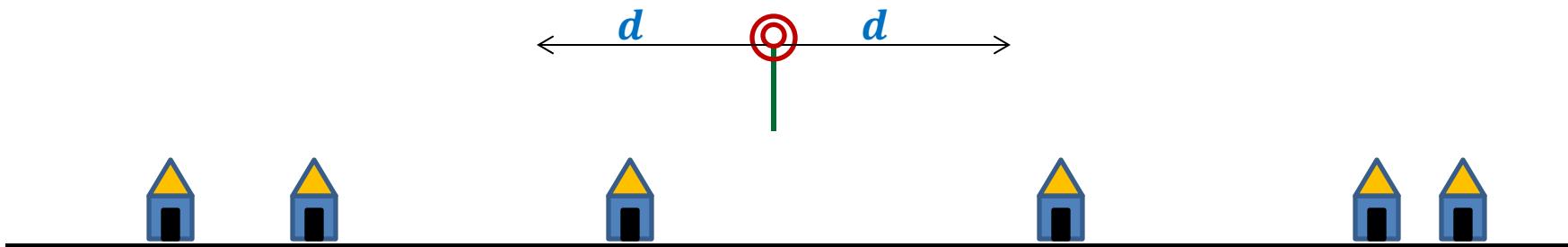
$$\text{Opt}(J) \geq \text{Opt}(J') + 1$$

$$\text{Opt}(J') \geq \text{Opt}(J) - 1$$

- Proof for each part is a proof **by construction**

Problem 2

Mobile Tower Problem



Problem statement:

There is a set H of n houses located along a road.

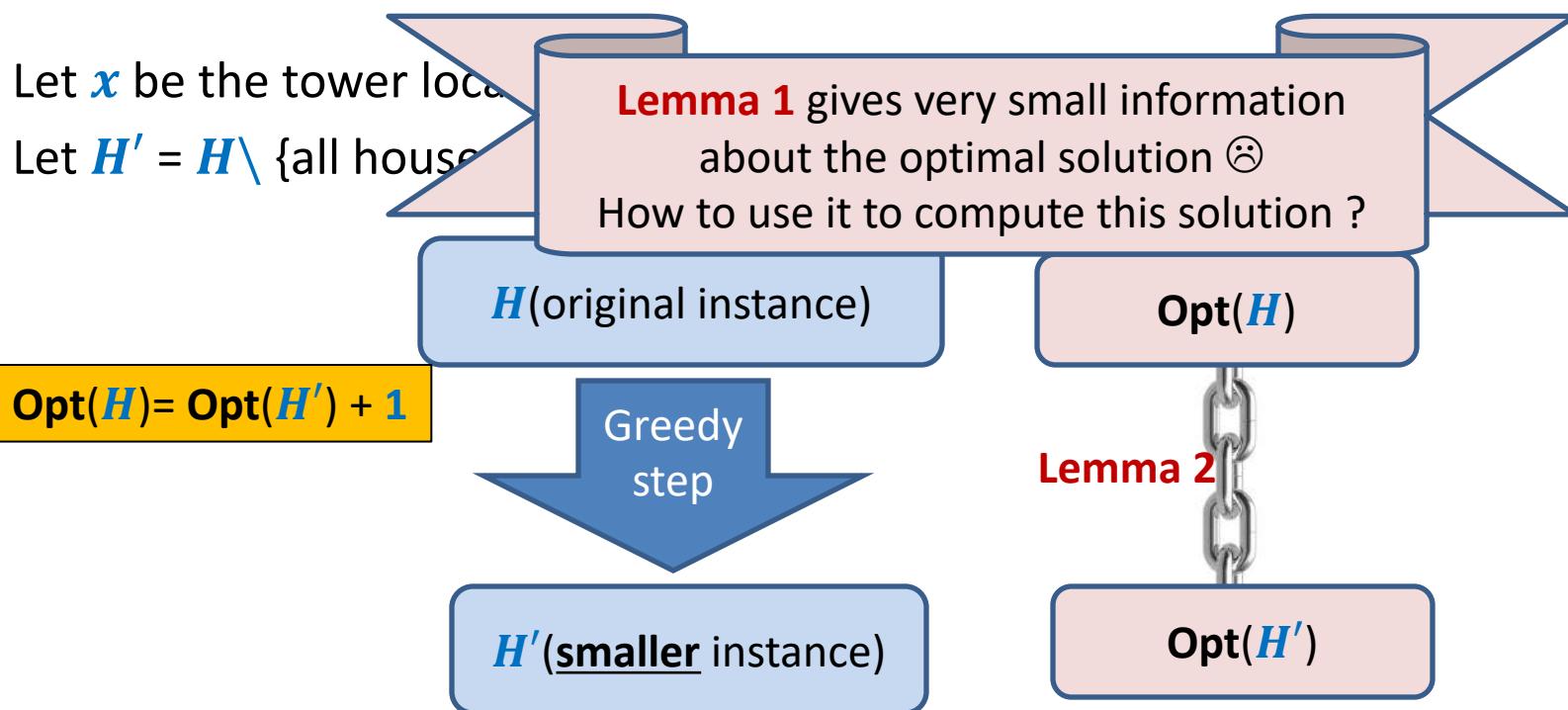
We want to place mobile towers such that

- Each house is covered by at least one mobile tower.
- The number of mobile towers used is **least** possible.

We could not say anything about the **complete solution** of this problem.

All that we could do was to make a local observation

Lemma 2: There is an optimal solution for the problem in which the leftmost tower is placed at distance d to the right of the first house.



Equation (i) hints at recursive solution of the problem 😊

What is a **greedy strategy** ?

A strategy that is

- Based on some **local** approach
- With the **objective to optimize** some function.

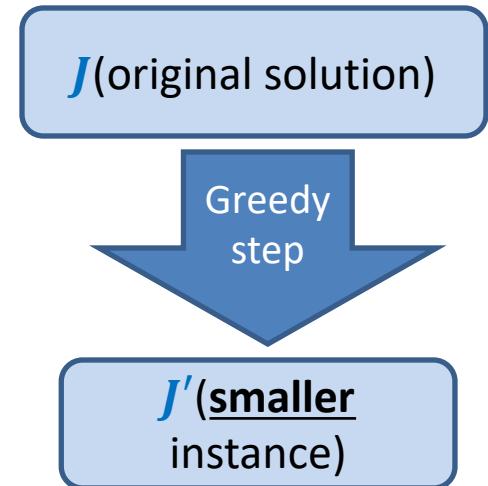
Note:

Recall that the divide and conquer strategy takes a **global approach**.

Design of a greedy algorithm

Let \mathbf{A} be an instance of an optimization problem.

1. Make a **local observation** about the solution.
2. Use this observation to express optimal solution of \mathbf{A} in terms of
 - Optimal solution of a smaller instance \mathbf{A}'
 - Local step
3. This gives a recursive solution.
4. Transform it into iterative one.



MST

Input: an undirected graph $G=(V,E)$ with $w: E \rightarrow \mathbb{R}$,

Aim: compute a **spanning tree** (V, E') , $E' \subseteq E$ such that $\sum_{e \in E'} w(e)$ is **minimum**.

Lemma 2

If you have understood a generic way to design a greedy algorithm, then try to solve the MST problem.

If $e_0 \in E$ is the edge of least weight in G , then there is a MST T containing e_0 .

How to use this **Lemma** to design an algorithm for **MST** ?

Problem 4

Overlapping Intervals

The aim of this problem is to make you realize that it is sometime very nontrivial to design a greedy algorithm. In particular, it is quite challenging to design the smaller instance.

Problem 4

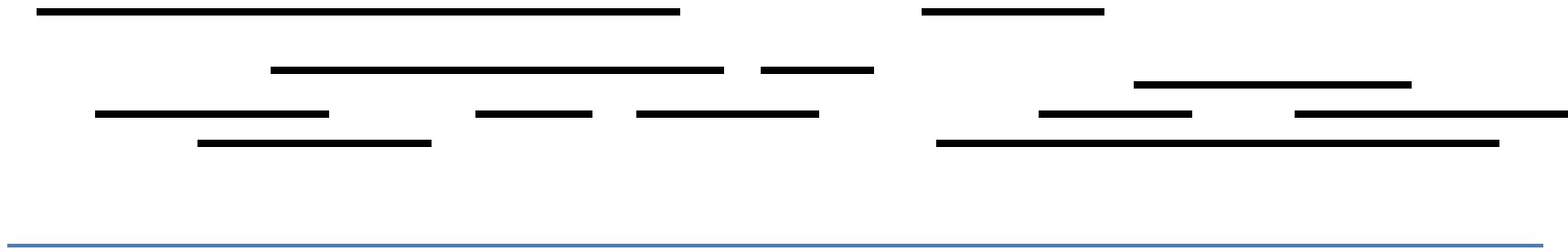
Overlapping Intervals

Overlapping Intervals

Problem statement:

Given a set **A** of n intervals, compute smallest set **B** of intervals so that for every interval **I** in $A \setminus B$, there is some interval in **B** which overlaps/intersects with **I**.

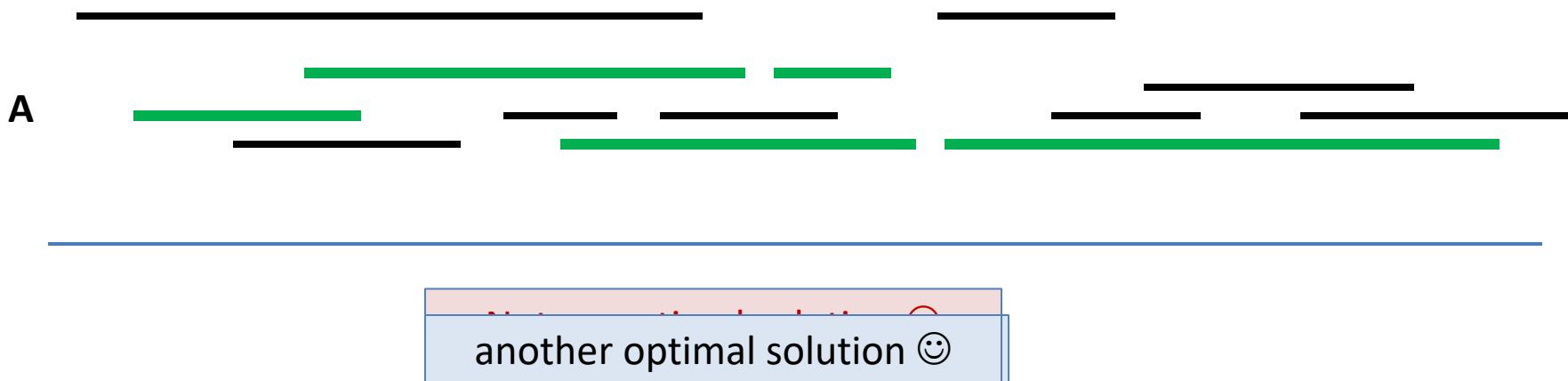
A



Overlapping Intervals

Problem statement:

Given a set **A** of n intervals, compute smallest set **B** of intervals so that for every interval **I** in $A \setminus B$, there is some interval in **B** which overlaps/intersects with **I**.



Overlapping Intervals

Strategy 1

Interval with maximum length should be there in optimal solution



Intuition:

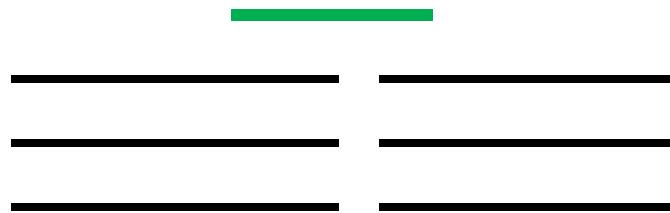
Selecting such an interval will **cover maximum** no. of other intervals

There is a counter example 😞

Overlapping Intervals

Strategy 1

Interval with maximum length should be there in optimal solution



Intuition:

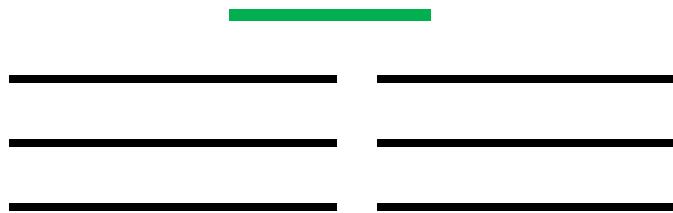
Selecting such an interval will **cover maximum** no. of other intervals

There is a counter example 😞

Overlapping Intervals

Strategy 2

Interval that overlaps maximum no. of intervals should be there in optimal solution



Intuition:

Selecting such an interval will **cover maximum** no. of other intervals

There is a counter example 😞

Overlapping Intervals

Strategy 2

Interval that overlaps maximum no. of intervals should be there in optimal solution



Overlapping Intervals

Strategy 2

Interval that overlaps maximum no. of intervals should be there in optimal solution



Not an optimal solution 😞

Overlapping Intervals

Strategy 2

Interval that overlaps maximum no. of intervals should be there in optimal solution



An optimal solution has size 2.

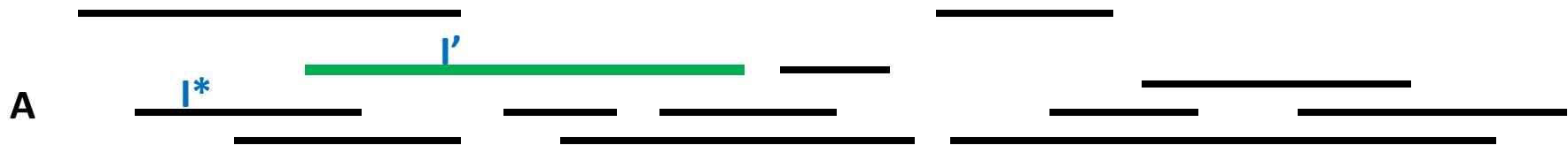
Think for a while :

After failure of two strategies, how to proceed to design the algorithm.

Overlapping Intervals

Let I^* be the interval with earliest finish time.

Let I' be the interval with **maximum** finish time overlapping I^* .



Lemma1: There is an optimal solution for set A that contains I' .

Proof:(sketch) :

If I^* is overlapped by any other interval in the optimal solution, say I^\wedge ,

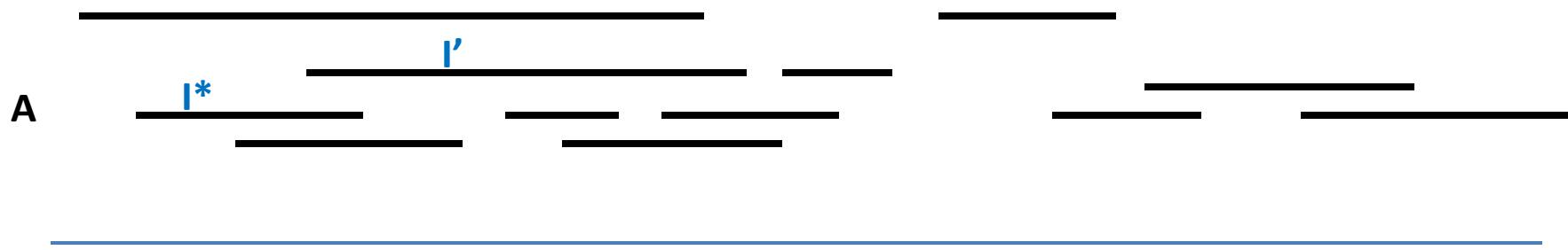
I' will surely overlap all intervals that are overlapped by I^\wedge .

→ Swapping I^\wedge by I' will still give an optimal solution.

Exploit the fact that
 I^* has earliest finish
time for this claim.

Overlapping Intervals

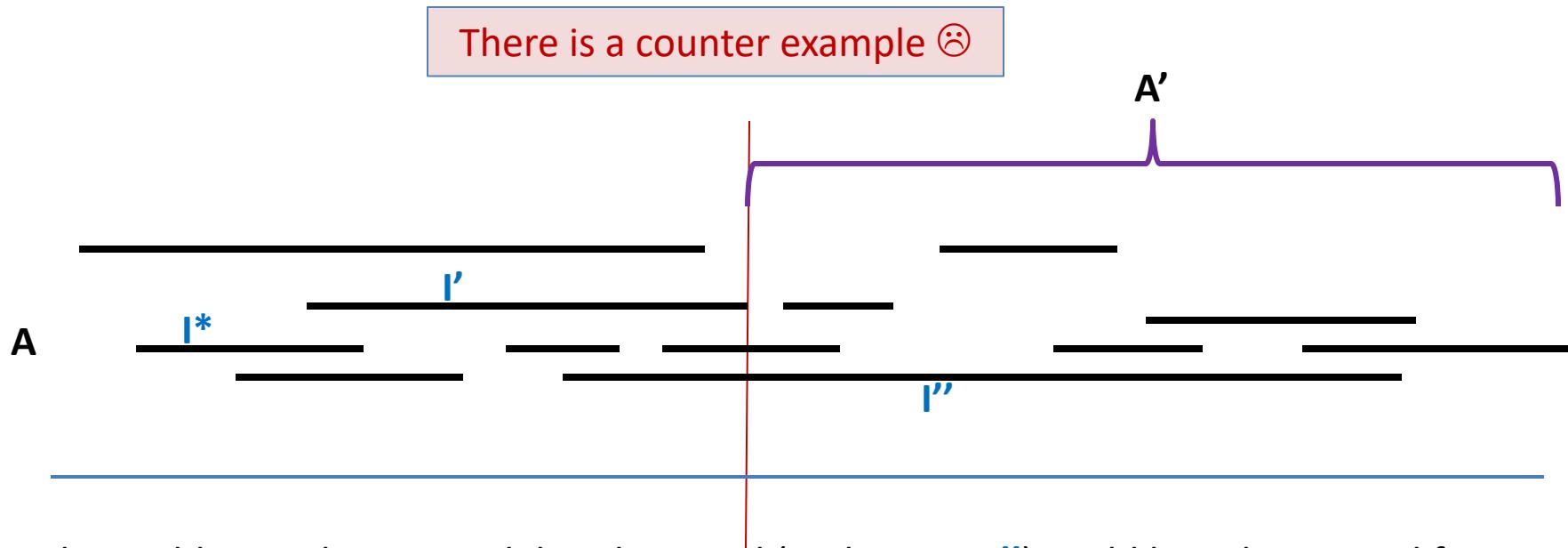
Question: How to obtain smaller instance A' using **Lemma 1** ?



Overlapping Intervals

Question: How to obtain smaller instance A' using **Lemma 1** ?

Naive approach : remove from A all intervals which overlap with I' . This is A' .



The problem is that some deleted interval (in this case I'') could have been used for intersecting many intervals if it were not deleted. But deleting it from the instance disallows it to be selected in the solution.

Homework

- How will you form the smaller instance ?
- Design an algorithm for the problem.
- Give a neat, concise, and formal proof of correctness of the algorithm.

Data Structures and Algorithms

(ESO207)

Lecture 38

- An interesting problem:
shortest path from a **source** to **destination**

SHORTEST PATHS IN A GRAPH

A fundamental problem

Notations and Terminologies

A directed graph $G = (V, E)$

- $\omega: E \rightarrow R^+$
- Represented as **Adjacency lists** or **Adjacency matrix**
- $n = |V|$, $m = |E|$

Question: what is a path in G ?

Answer: A sequence v_1, v_2, \dots, v_k such that $(v_i, v_{i+1}) \in E$ for all $1 \leq i < k$.



Length of a path $P = \sum_{e \in P} \omega(e)$

Notations and Terminologies

Definition:

The path from u to v of minimum length is called the **shortest path** from u to v

Definition: Distance from u to v is the length of the shortest path from u to v .

Notations:

$\delta(u, v)$: distance from u to v .

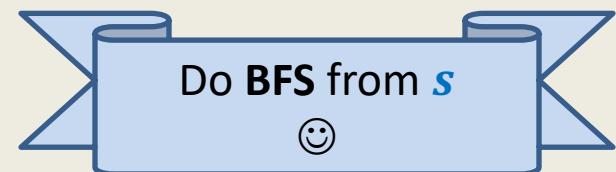
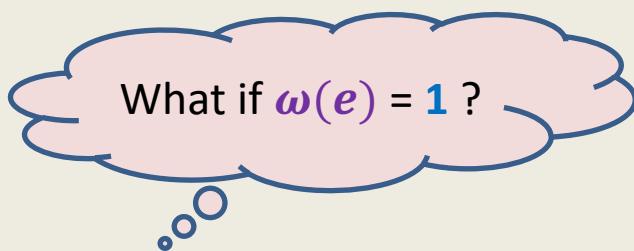
$P(u, v)$: The shortest path from u to v .

Problem Definition

Input: A directed graph $G = (V, E)$ with $\omega: E \rightarrow R^+$ and a source vertex $s \in V$

Aim:

- Compute $\delta(s, v)$ for all $v \in V \setminus \{s\}$
- Compute $P(s, v)$ for all $v \in V \setminus \{s\}$



Problem Definition

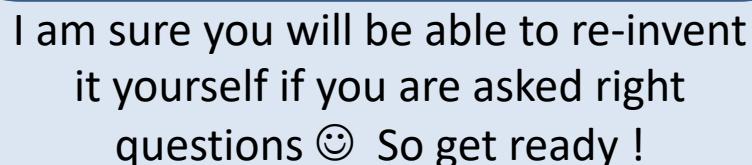
Input: A directed graph $G = (V, E)$ with $\omega: E \rightarrow R^+$ and a source vertex $s \in V$

Aim:

- Compute $\delta(s, v)$ for all $v \in V \setminus \{s\}$
- Compute $P(s, v)$ for all $v \in V \setminus \{s\}$

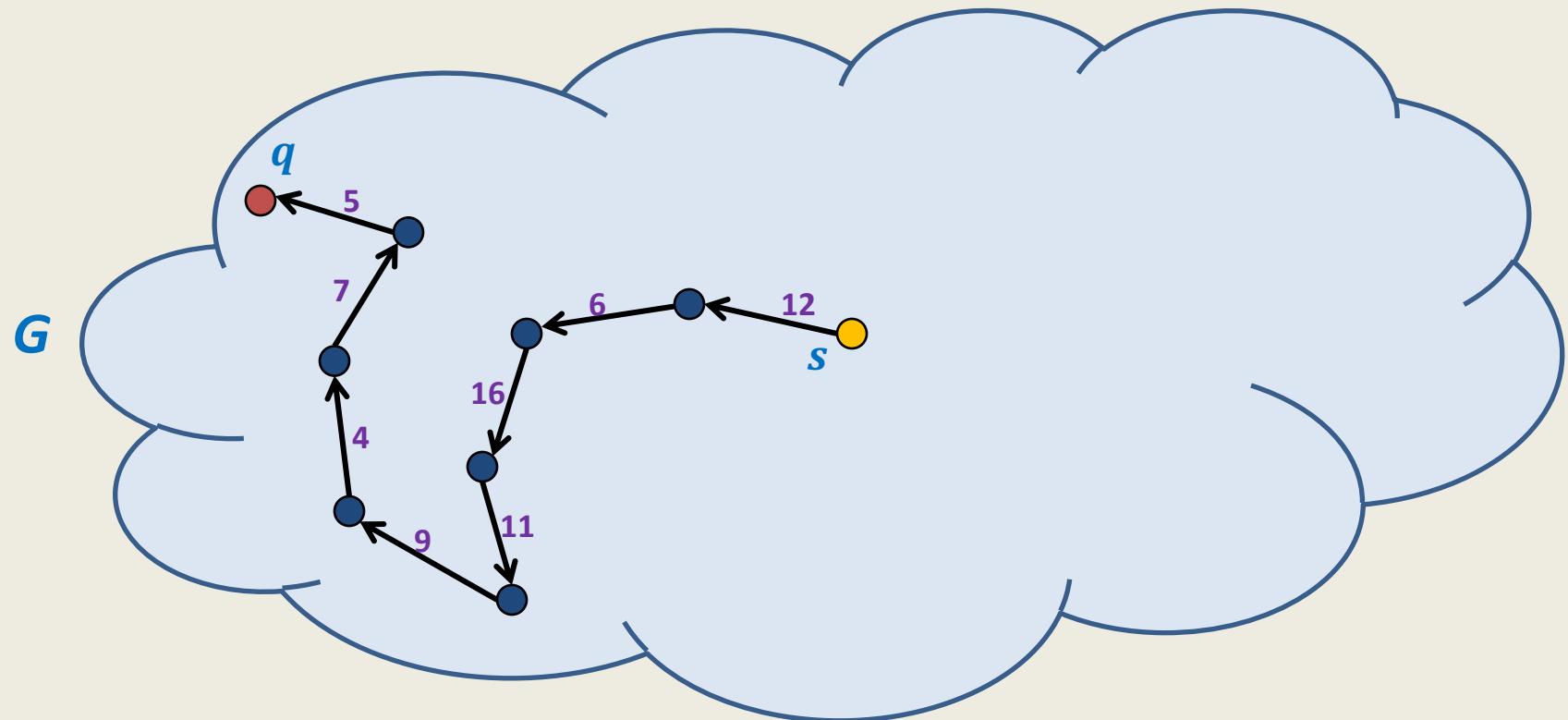
First algorithm : by Edsger Dijkstra in 1956

And still the best ...



I am sure you will be able to re-invent
it yourself if you are asked right
questions ☺ So get ready !

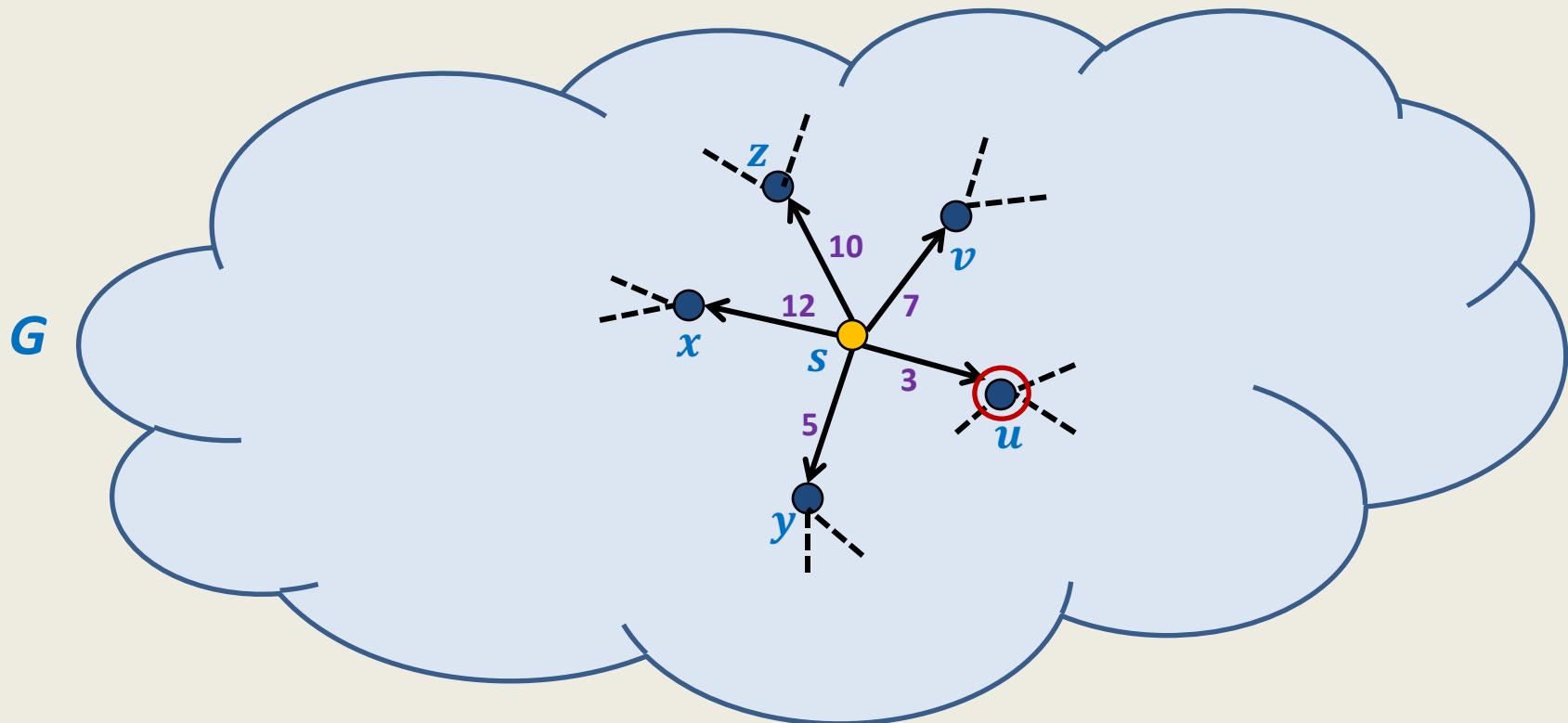
An example to get an insight into this problem



Inference:

The distance to any vertex depends upon global parameters.

An example to get an insight into this problem

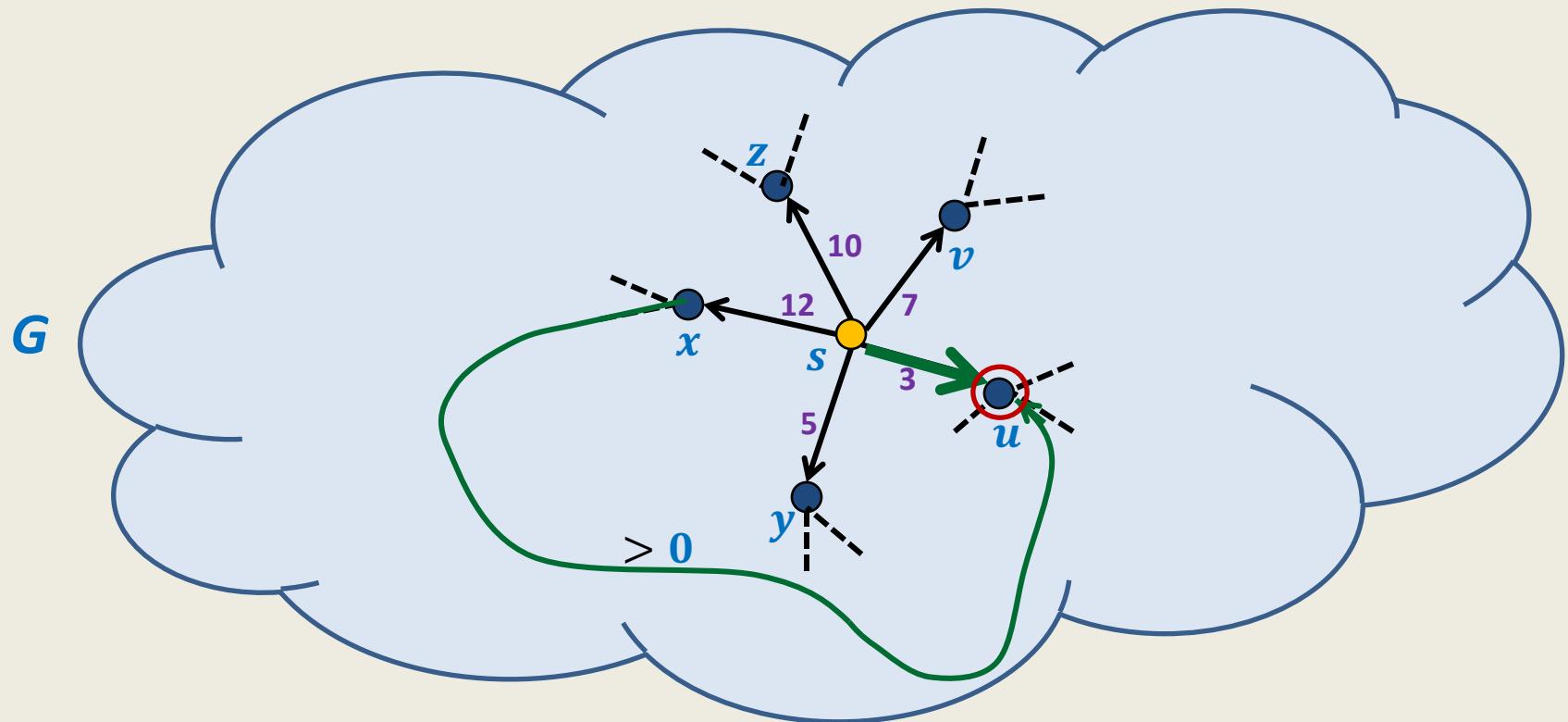


Question: Is there any vertex in this picture for which you are certain about the distance from s ?

Answer: vertex u .

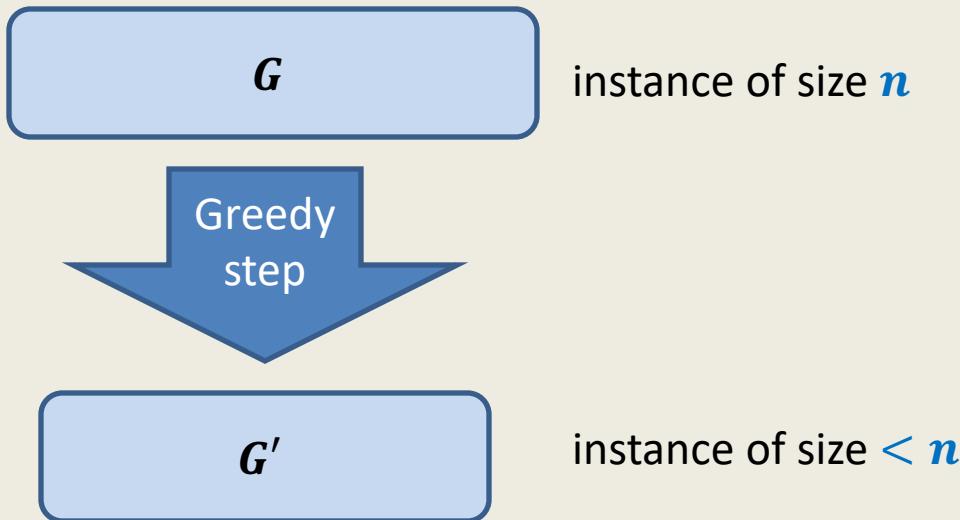
Give reasons.

An example to get an insight into this problem



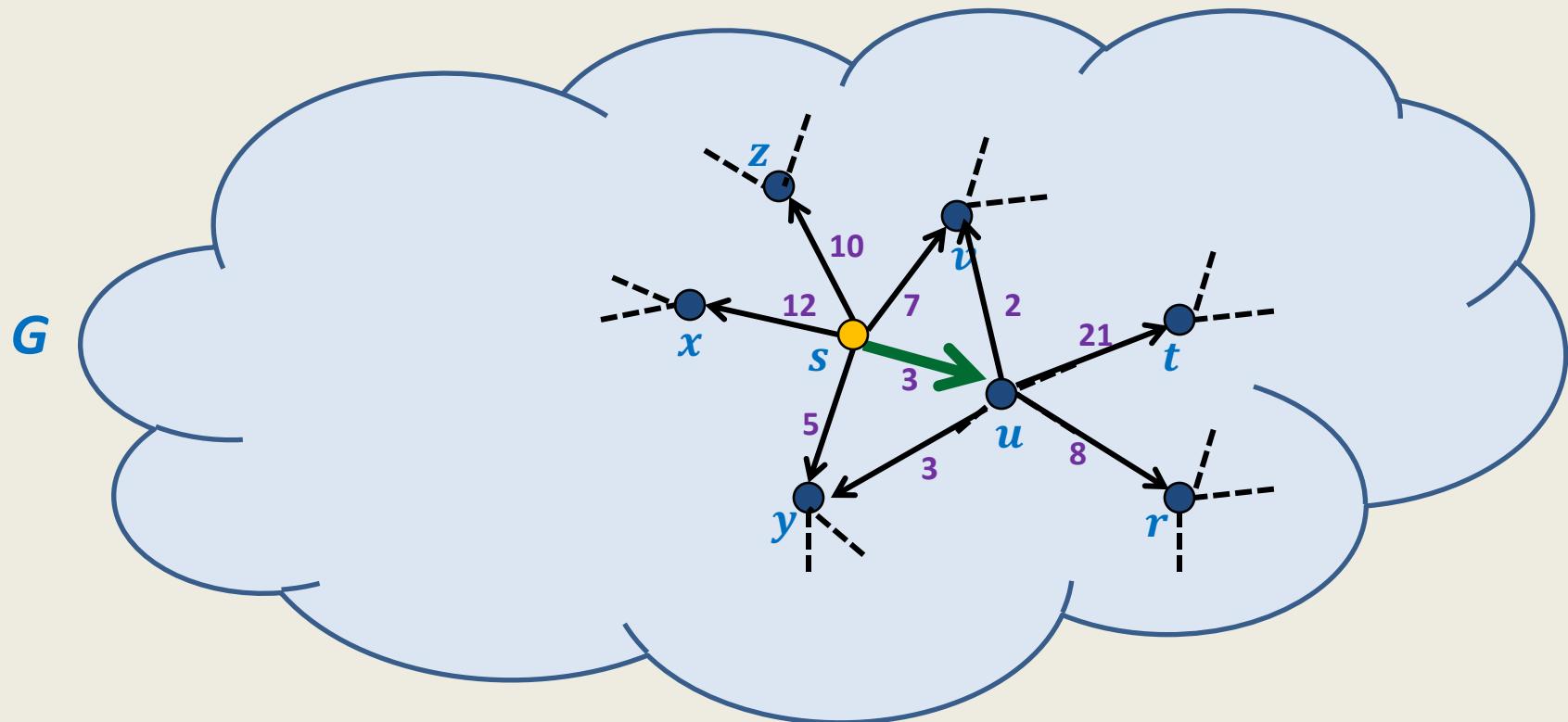
→ The shortest path to vertex u is edge (s, u) .

Designing a greedy algorithm for shortest paths



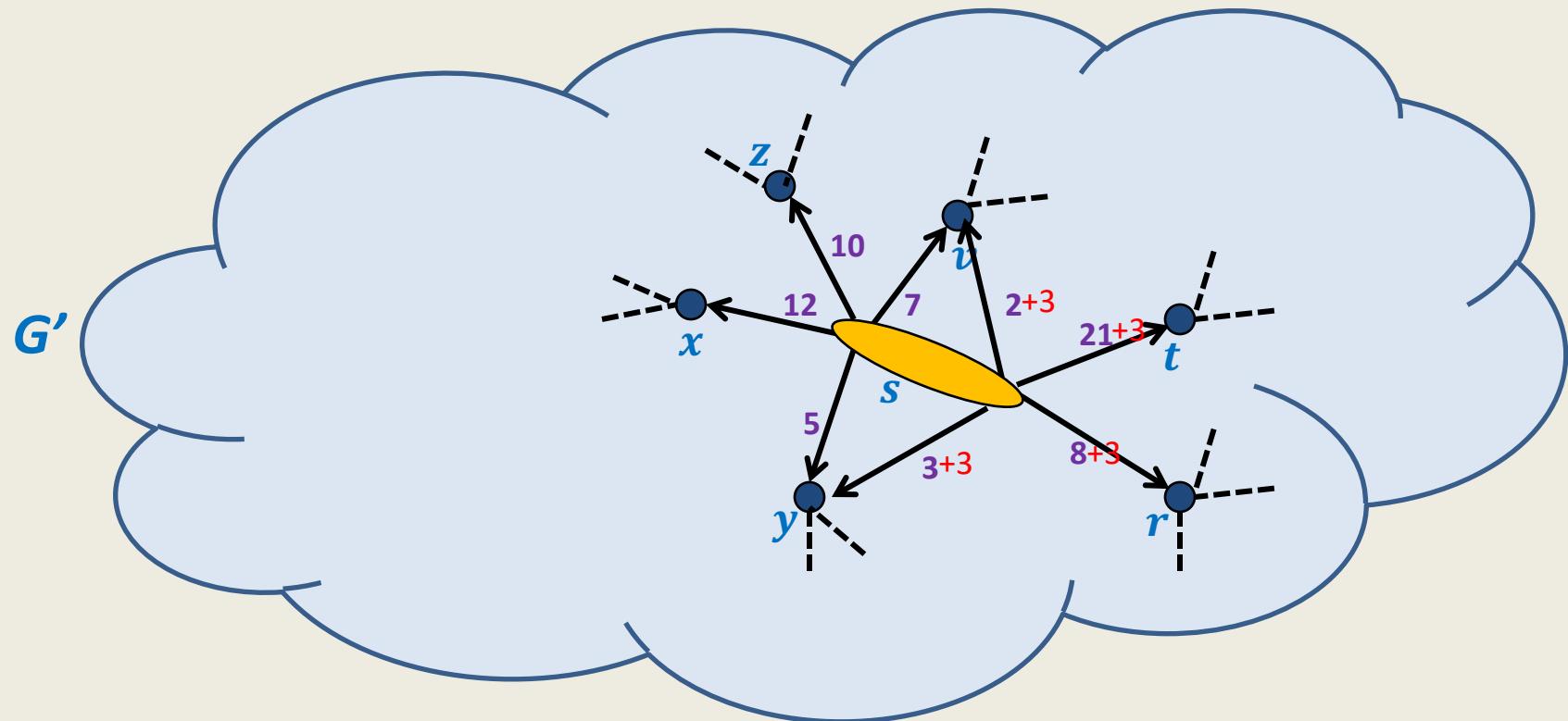
1. Establish a relation between
and
shortest paths in G'

An example to get an insight into this problem

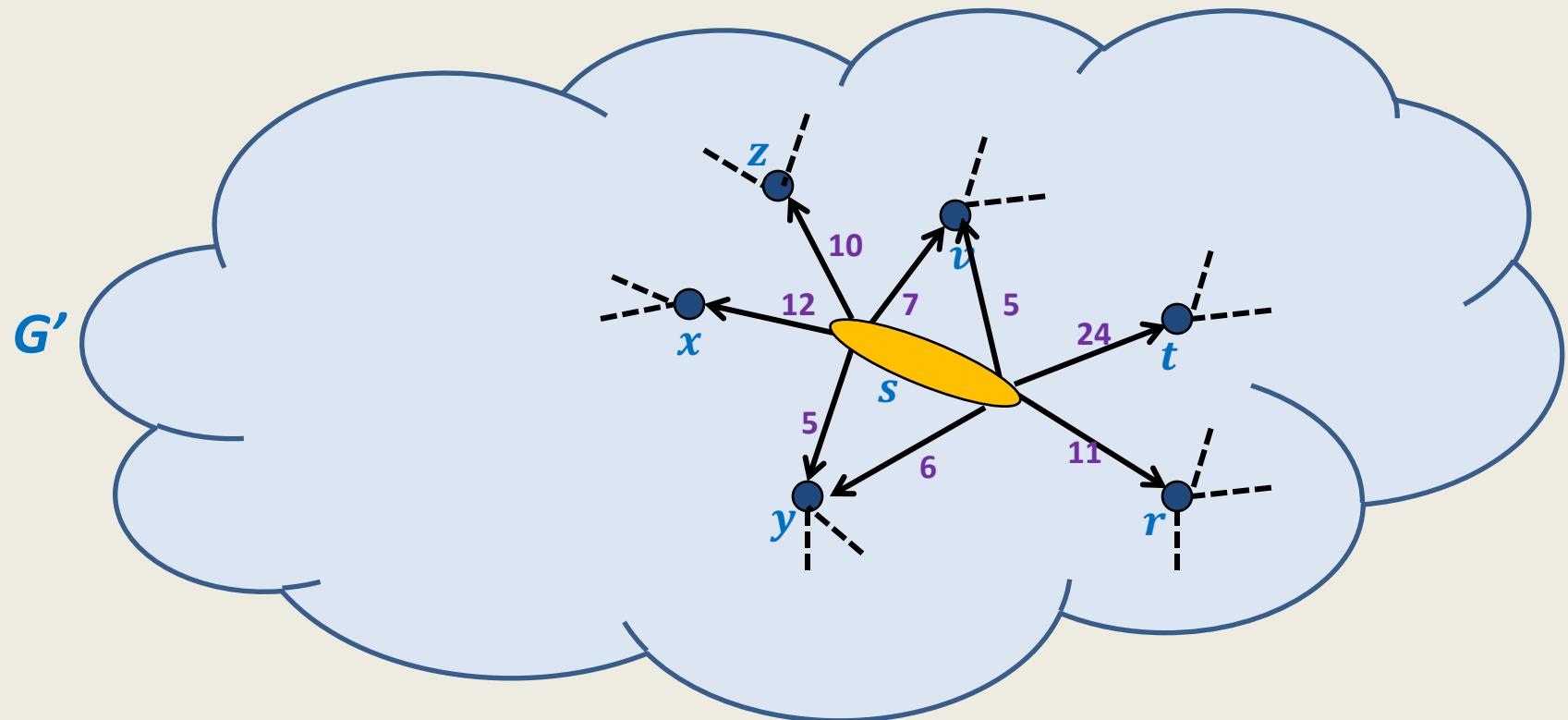


Question: Can you remove vertex u without affecting the distance from s ?

An example to get an insight into this problem



An example to get an insight into this problem



How to compute instance G'

Let (s, u) be the **least weight edge** from s in $G = (V, E)$.

Transform G into G' as follows.

1. For each edge $(u, x) \in E$,

add edge (s, x) ;

$$\omega(s, x) \leftarrow \omega(s, u) + \omega(u, x);$$

2. In case of two edges from s to any vertex x

3. Remove vertex u .



Theorem: For each $v \in V \setminus \{s, u\}$,

$$\delta_G(s, v) = \delta_{G'}(s, v)$$

→ an algorithm for **distances** from s

Can you see some **negative points** of this algorithm ?

Shortcomings of the algorithm

- **No insight** into the (beautiful) structure of shortest paths.
- **Just convinces** that we can solve the shortest paths problem in polynomial time.
- **Very few options** to improve the time complexity.

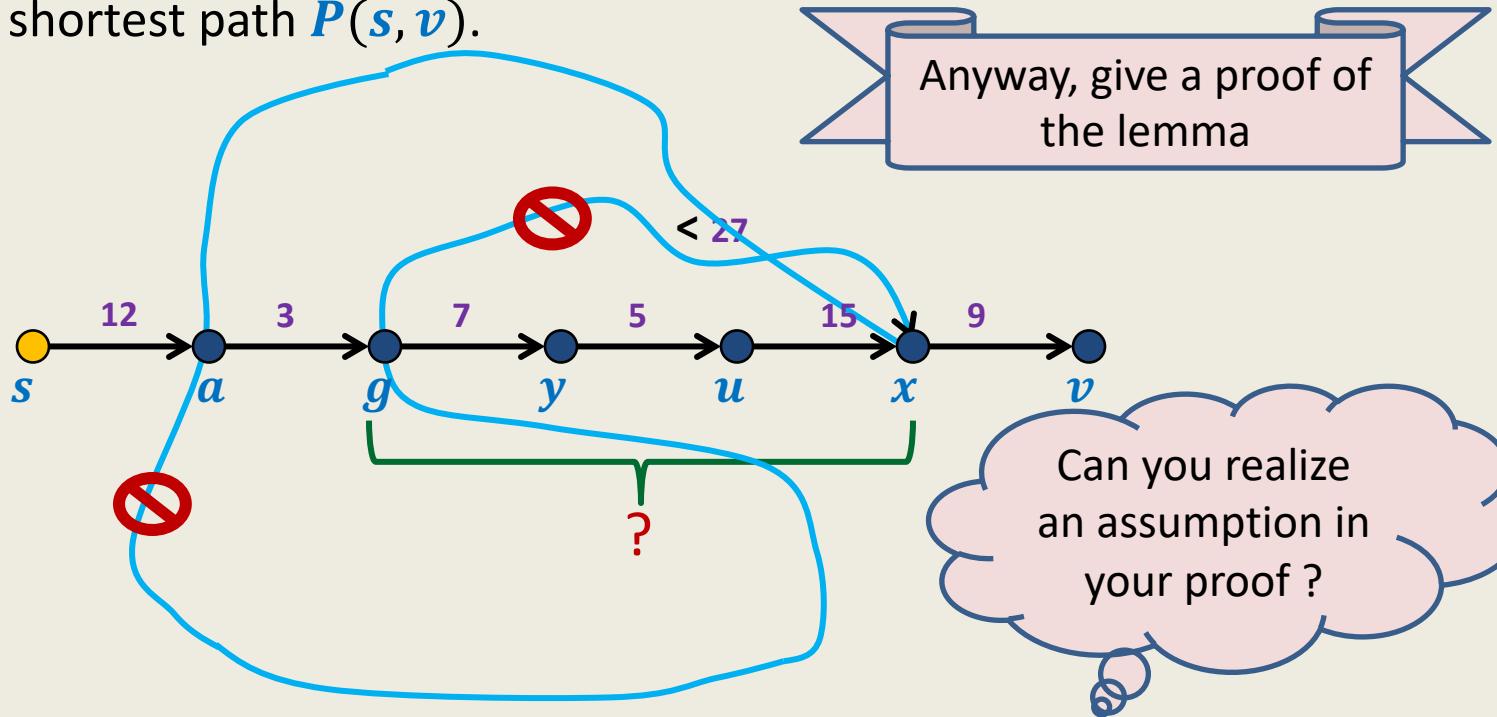
We shall now design a very **insightful** algorithm based on **properties** of shortest paths.

PROPERTY OF A SHORTEST PATH

Optimal subpath property

Consider any shortest path $P(s, v)$.

$$\delta(s, v) = 51$$



Lemma 1: Every **subpath** of a shortest path is also a shortest path.

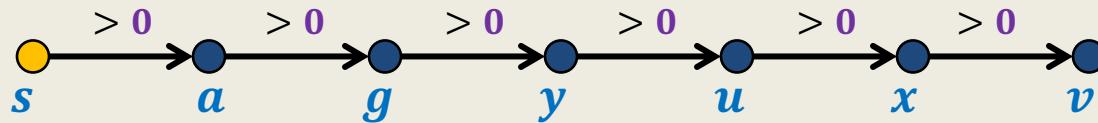
NOTE: Does the lemma use the fact that the edge weights are positive?

If yes, can you locate the exact place where it used it?

Homework: Write a complete and formal proof for **Lemma 1**

Exploiting the positive weight on edges

Consider once again a shortest path $P(s, v)$.



$$\rightarrow \delta(s, a) < \delta(s, g) < \delta(s, y) < \delta(s, u) < \delta(s, x) < \delta(s, v) \quad (1)$$

\rightarrow The first nearest vertex of s must be its **neighbor**.

More insights ...

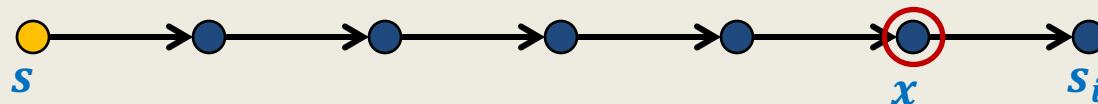
Let s_i :

$$s_0 = s.$$

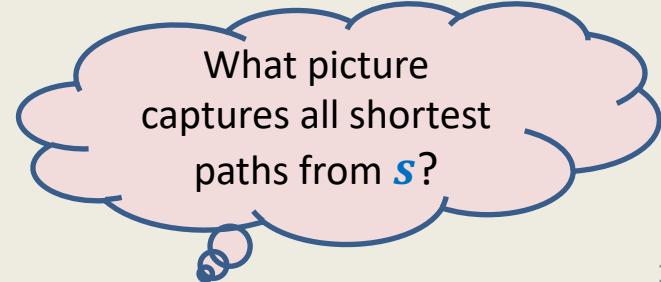
Consider the shortest path $P(s, s_i)$.



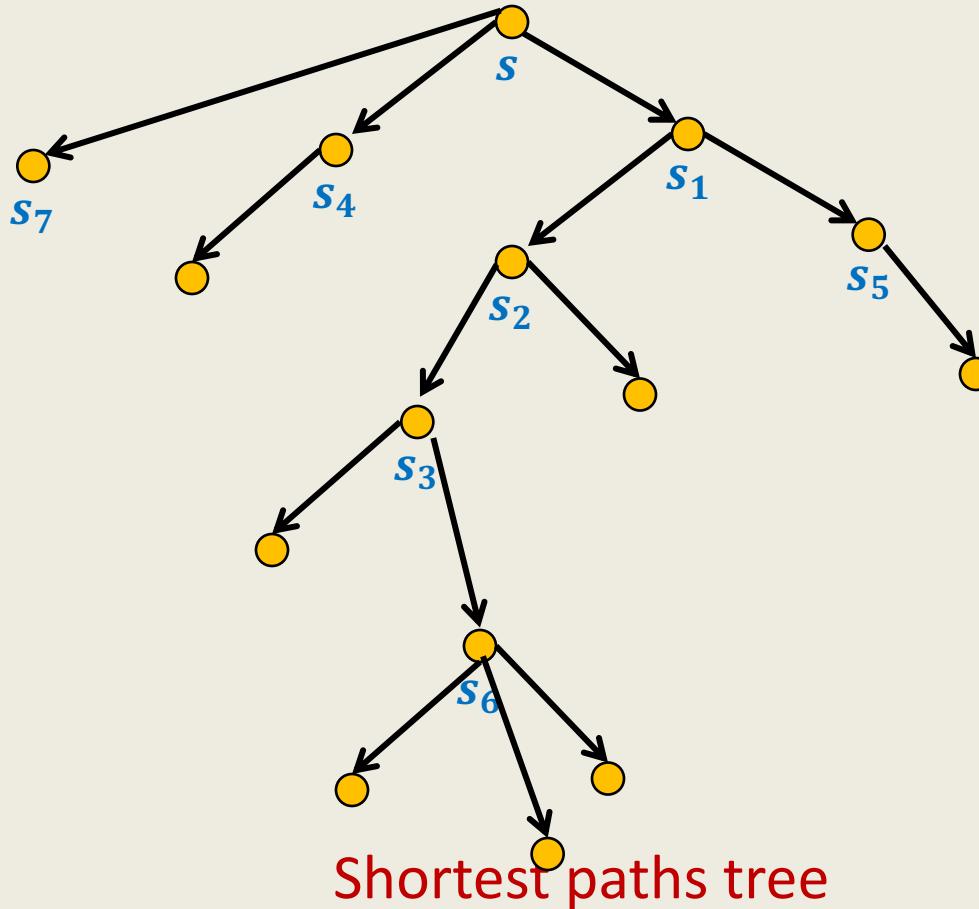
x must be s_j for some $j < i$.



Lemma 2: $P(s, s_i)$ must be of the form



Complete picture of all shortest paths ?



Designing the algorithm ...

Lemma 2: $P(s, s_i)$ must be of the form $s \rightsquigarrow s_j \rightarrow s_i$ for some $j < i$.

shortest

Question: Can we use **Lemma 2** to design an algorithm ?

Incremental way to compute shortest paths.

Ponder over it before going to the next slide 😊

Designing the algorithm ...

Lemma 2: $P(s, s_i)$ must be of the form $s \rightsquigarrow s_j \rightarrow s_i$ for some $j < i$.

shortest

Question: Can we use **Lemma 2** to design an algorithm ?

Incremental way to compute shortest paths.

The next slide explains it precisely.

If shortest paths to $s_j, j < i$ is known, we can compute s_i .

But how ?

All we know is that

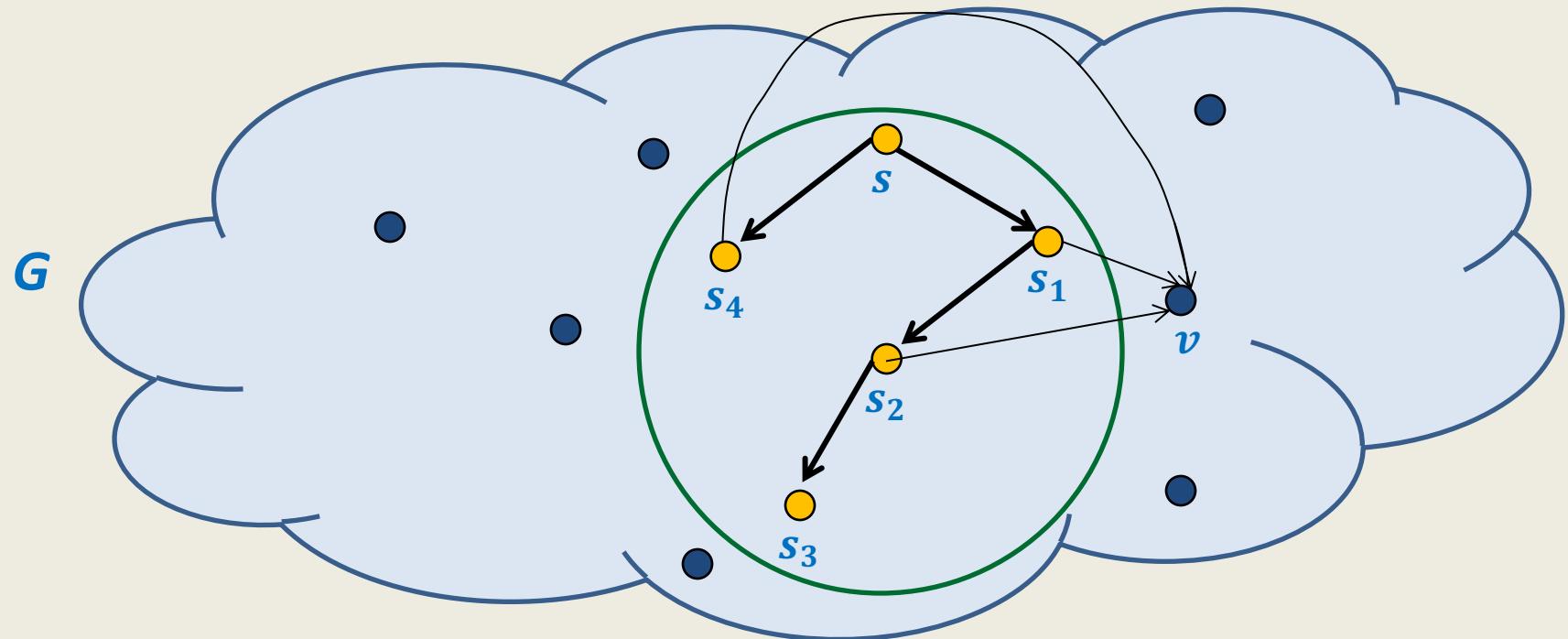
But which neighbor ?

Hint:

For each neighbor, compute some *label* based on
Lemma 2 s.t. s_i is the neighbour with least label.

Suppose we have computed s_1, s_2, \dots, s_{i-1} .

We can compute s_i as follows.



For each $v \in V \setminus \{s_1, s_2, \dots, s_{i-1}\}$

$$L(v) = \min_{(s_j, v) \in E} (\delta(s, s_j) + \omega(s_j, v))$$

s_i is the vertex with **minimum** value of L .

Dijkstra's algorithm

Dijkstra-algo(s, G)

{ $U \leftarrow V \setminus \{s\}$;

$S \leftarrow \{s\}$;

For $i = 1$ to $n - 1$ do

{ For each $v \in U$ do

{ $L(v) \leftarrow \infty$;

 For each $(x, v) \in E$ with $x \in S$ do

$L(v) \leftarrow \min(L(v), \delta(s, x) + \omega(x, v))$

}

$y \leftarrow$ vertex from U with minimum value of L ;

$\delta(s, y) \leftarrow L(y)$;

 move y from U to S ;

}

In this algorithm, we first compute $L(v)$ for each $v \in U$ and then find the vertex with the least L value.

Try to rearrange its statements so that in the beginning of each iteration, we have L values **computed already**.

This rearrangement will be helpful for improving the running time.

So please try it on your own first before viewing the next slide.

Dijkstra's algorithm

Dijkstra-algo(s, G)

{ $U \leftarrow V$; $L(v) \leftarrow \infty$ for all $v \in U$;

$L(s) \leftarrow 0$;

For $i = 0$ to $n - 1$ do

{ $y \leftarrow$ vertex from U with minimum value of L ;

$\delta(s, y) \leftarrow L(y)$;

move y from U to S ;

For each $v \in U$ do

{ $L(v) \leftarrow \infty$;

For each $(x, v) \in E$ with $x \in S$ do

$L(v) \leftarrow \min(L(v), \delta(s, x) + \omega(x, v))$

}

}

a lot of re-computation

Dijkstra's algorithm

Dijkstra-algo(s, G)

{ $U \leftarrow V$; $L(v) \leftarrow \infty$ for all $v \in U$;

$L(s) \leftarrow 0$;

For $i = 0$ to $n - 1$ do

{ $y \leftarrow$ vertex from U with minimum value of L ;

$\delta(s, y) \leftarrow L(y)$;

move y from U to S ;

For each $v \in U$ do

{

For each $(x, v) \in E$ with $x \in S$ do

$L(v) \leftarrow \min(L(v), \delta(s, x) + \omega(x, v))$

}

}

Only neighbors of y

What are the vertices whose L value may change in this iteration ?

Dijkstra's algorithm

Dijkstra-algo(s, G)

{ $U \leftarrow V$; $L(v) \leftarrow \infty$ for all $v \in U$;

$L(s) \leftarrow 0$;

For $i = 0$ to $n - 1$ do

{ $y \leftarrow$ vertex from U with minimum value of L ;

$\delta(s, y) \leftarrow L(y)$;

move y from U to S ;

For each $(y, v) \in E$ with $v \in U$ do

{

$L(v) \leftarrow \min(L(v), \delta(s, y) + \omega(y, v))$

}

}

1 extract-min operation

deg(y) Decrease-key operations

Time complexity of Dijkstra's algorithm

Total number of **extract-min** operation : n

Total **Decrease-key** operations : m

Using **Binary heap** to maintain the set U , the time complexity: $O(m \log n)$

Theorem: Given a directed graph with positive weights on edges, we can compute all shortest paths from a given vertex in $O(m \log n)$ time.

Fibonacci heap supports **Decrease-key** in $O(1)$ time and **extract-min** in $O(\log n)$.

→ Total time complexity using Fibonacci heap: $O(m + n \log n)$

Data Structures and Algorithms

(ESO207)

Lecture 39

- Integer sorting
- Counting Sort and Radix Sort

Integer sorting

Algorithms for Sorting n elements

- **Insertion** sort: $\mathcal{O}(n^2)$
- **Selection** sort: $\mathcal{O}(n^2)$
- **Bubble** sort: $\mathcal{O}(n^2)$
- **Merge** sort: $\mathcal{O}(n \log n)$
- **Quick** sort: worst case $\mathcal{O}(n^2)$, average case $\mathcal{O}(n \log n)$
- **Heap** sort: $\mathcal{O}(n \log n)$

Question: What is common among these algorithms ?

Answer: All of them use only **comparison** operation to perform sorting.

Theorem (we will not prove it in this course):

Every comparison based sorting algorithm

must perform at least $n \log n$ comparisons in the worst case.

Question: Can we sort in $O(n)$ time ?

The answer depends upon

- the model of computation.
- the domain of input.

Integer sorting

Counting sort: algorithm for sorting integers

Input: An array A storing n integers in the range $[0 \dots k - 1]$.

$$k = O(n)$$

Output: Sorted array A .

Running time: $O(n + k)$ in word RAM model of computation.

Extra space: $O(k)$

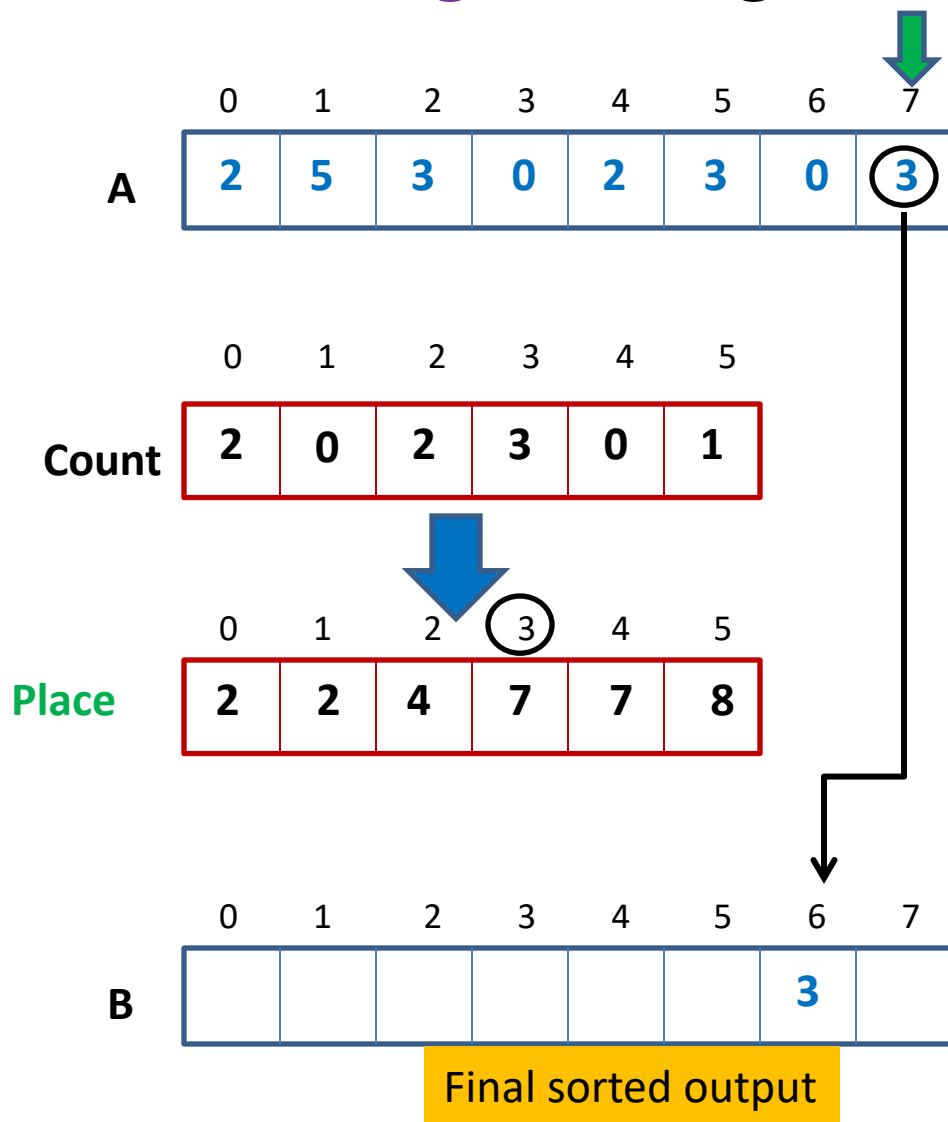
Motivating example: Indian railways

There are 13 lacs employees.

Aim : To sort them list according to DOB (date of birth)

Observation: There are only 14600 different date of births possible.

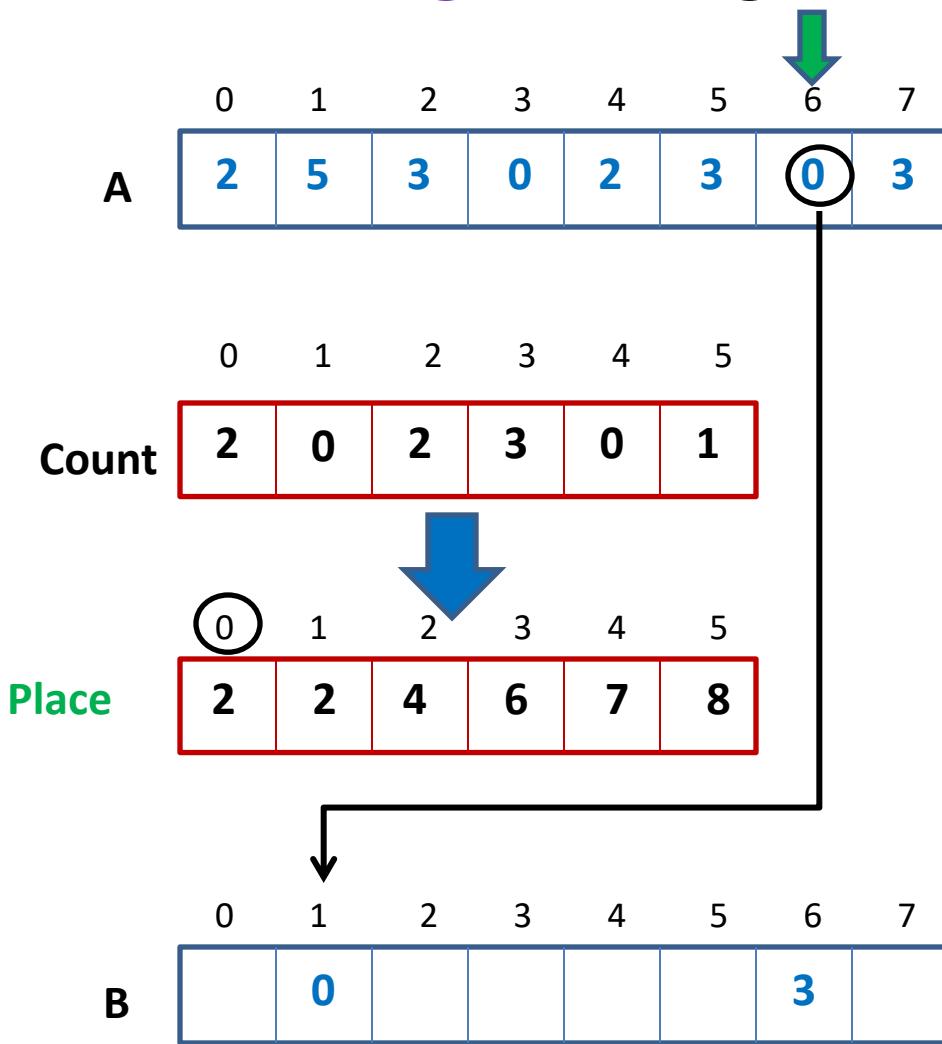
Counting sort: algorithm for sorting integers



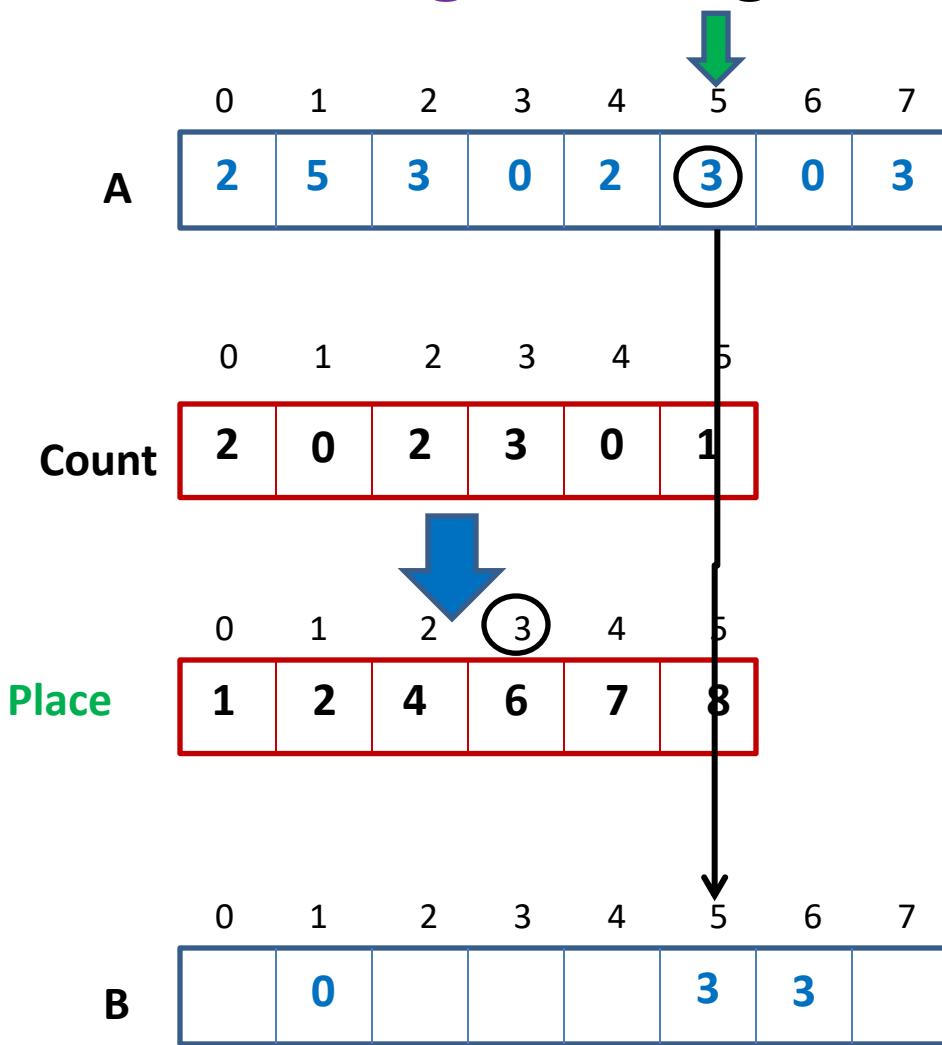
If $A[i]=j$,
where should $A[i]$ be
placed in B ?

Certainly after all those elements in A
which are smaller than j

Counting sort: algorithm for sorting integers



Counting sort: algorithm for sorting integers



Types of sorting algorithms

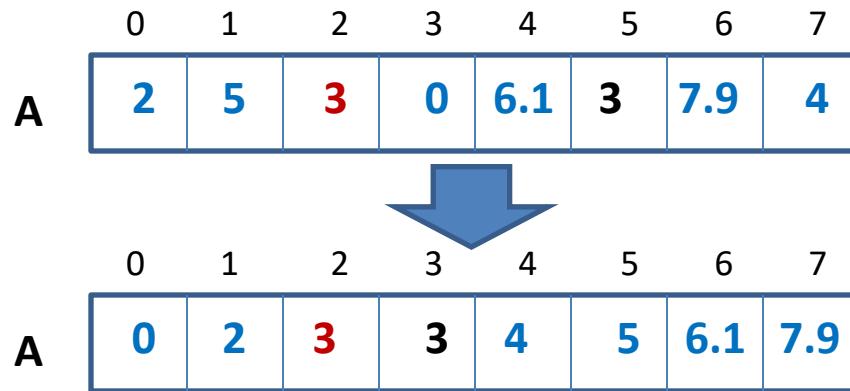
In Place Sorting algorithm:

A sorting algorithm which uses

Example: Heap sort, Quick sort.

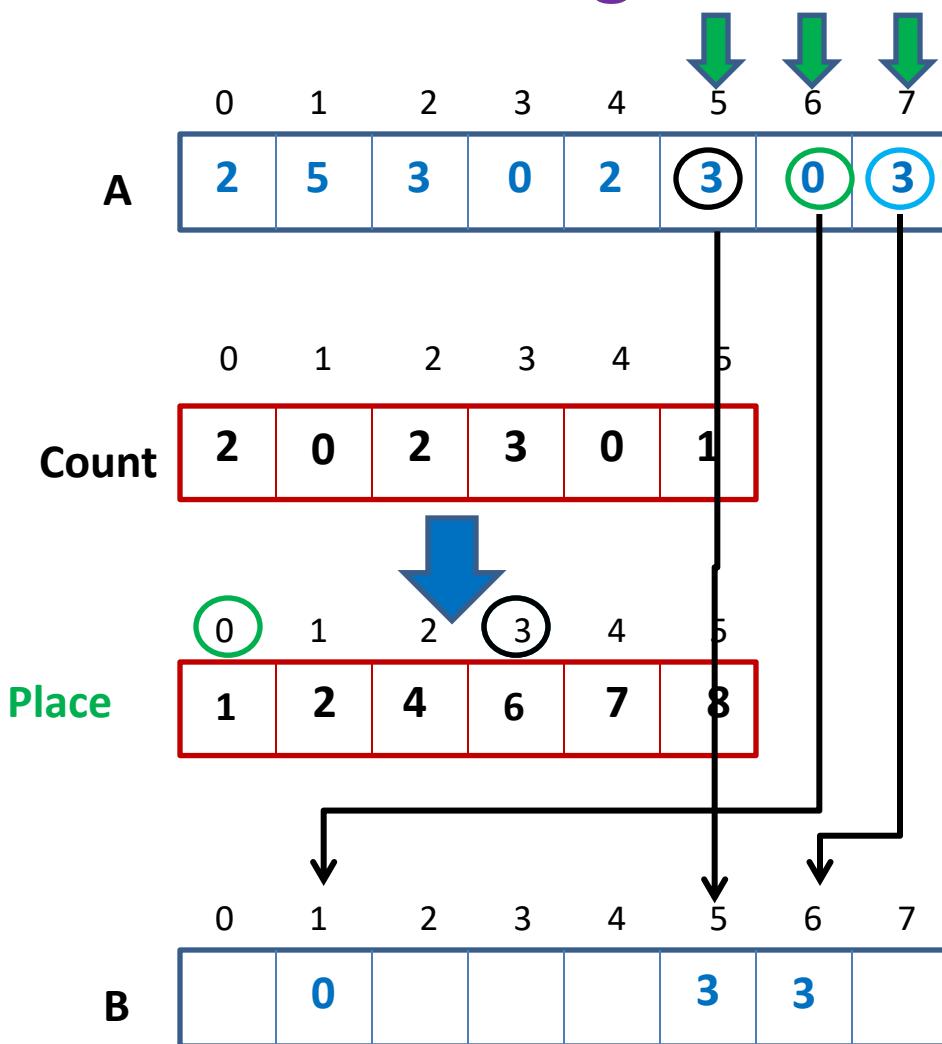
Stable Sorting algorithm:

A sorting algorithm which preserves



Example: Merge sort.

Counting sort: a visual description



Why did we scan elements of **A** in reverse order (from index $n - 1$ to **0**) while placing them in the final sorted array **B**?

Answer:

- To ensure that Counting sort is **stable**.
- The reason why stability is required will become clear soon 😊

Counting sort: algorithm for sorting integers

Algorithm ($A[0 \dots n - 1], k$)

For $j=0$ to $k - 1$ do $\text{Count}[j] \leftarrow 0$;

For $i=0$ to $n - 1$ do $\text{Count}[A[i]] \leftarrow \text{Count}[A[i]] + 1$;

$\text{Place}[0] \leftarrow \text{Count}[0]$;

For $j=1$ to $k - 1$ do $\text{Place}[j] \leftarrow \text{Place}[j - 1] + \text{Count}[j]$;

For $i=n - 1$ to 0 do

{ $B[\text{Place}[A[i]] - 1] \leftarrow A[i]$;

$\text{Place}[A[i]] \leftarrow \text{Place}[A[i]] - 1$;

}

return B ;

Each arithmetic operations

involves $O(\log n + \log k)$ bits

Counting sort: algorithm for sorting integers

Key points of Counting sort:

- It performs arithmetic operations involving $O(\log n + \log k)$ bits
($O(1)$ time in word RAM).
- It is a **stable** sorting algorithm.

Theorem: An array storing n integers in the range $[0..k - 1]$
can be sorted in $O(n+k)$ time and
using total $O(n+k)$ space in word RAM model.

- For $k \leq n$,
- For $k = n^t$,
(too bad for $t > 1$. ☹)

Question:

How to sort n integers in the range $[0..n^t]$ in

Radix Sort

Digits of an integer

507266

No. of digits = 6

value of digit $\in \{0, \dots, 9\}$

The binary representation of the decimal number 507266 is shown as a sequence of bits: 1011000101011111. Four horizontal brackets above the sequence group the bits into four groups of three, indicating they represent nibbles or bytes. This visualizes how integers can have different digit counts depending on the base used.

1011000101011111

No. of digits = 4

value of digit $\in \{0, \dots, 15\}$

It is up to us how we define digit ?

Radix Sort

Input: An array \mathbf{A} storing n integers, where

- (i) each integer has exactly d digits.
- (ii) each digit has value $< k$
- (iii) $k < n$.

Output: Sorted array \mathbf{A} .

Running time:

$O(dn)$ in word RAM model of computation.

Extra space:

$O(n + k)$

Important points:

- makes use of a count sort.
- Heavily relies on the fact that count sort is a stable sort algorithm.

Demonstration of Radix Sort through example

A



2	0	1	2
1	3	8	5
4	9	6	1
5	8	1	0
2	3	7	3
6	2	3	9
9	6	2	4
8	2	9	9
3	4	6	5
7	0	9	8
5	5	0	1
9	2	5	8



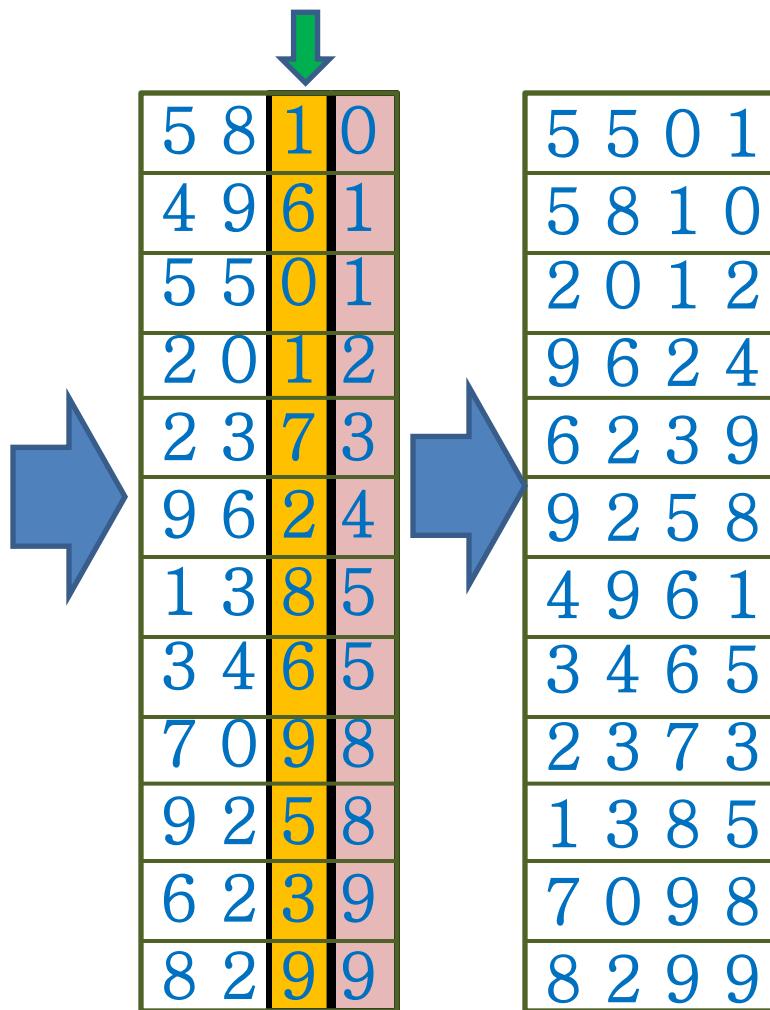
5	8	1	0
4	9	6	1
5	5	0	1
2	0	1	2
2	3	7	3
9	6	2	4
1	3	8	5
3	4	6	5
7	0	9	8
9	2	5	8
6	2	3	9
8	2	9	9

$d = 4$
 $n = 12$
 $k = 10$

Demonstration of Radix Sort through example

A

2	0	1	2
1	3	8	5
4	9	6	1
5	8	1	0
2	3	7	3
6	2	3	9
9	6	2	4
8	2	9	9
3	4	6	5
7	0	9	8
5	5	0	1
9	2	5	8



5	5	0	1
5	8	1	0
2	0	1	2
9	6	2	4
6	2	3	9
9	2	5	8
4	9	6	1
3	4	6	5
2	3	7	3
1	3	8	5
7	0	9	8
8	2	9	9

Demonstration of Radix Sort through example

A

2 0 1 2
1 3 8 5
4 9 6 1
5 8 1 0
2 3 7 3
6 2 3 9
9 6 2 4
8 2 9 9
3 4 6 5
7 0 9 8
9 2 5 8
5 5 0 1
9 2 5 8

5 8 1 0
4 9 6 1
5 5 0 1
2 0 1 2
2 3 7 3
9 6 2 4
1 3 8 5
3 4 6 5
7 0 9 8
9 2 5 8
6 2 3 9
8 2 9 9
4 9 6 1
3 4 6 5
5 5 0 1
9 2 5 8
6 2 3 9
8 2 9 9

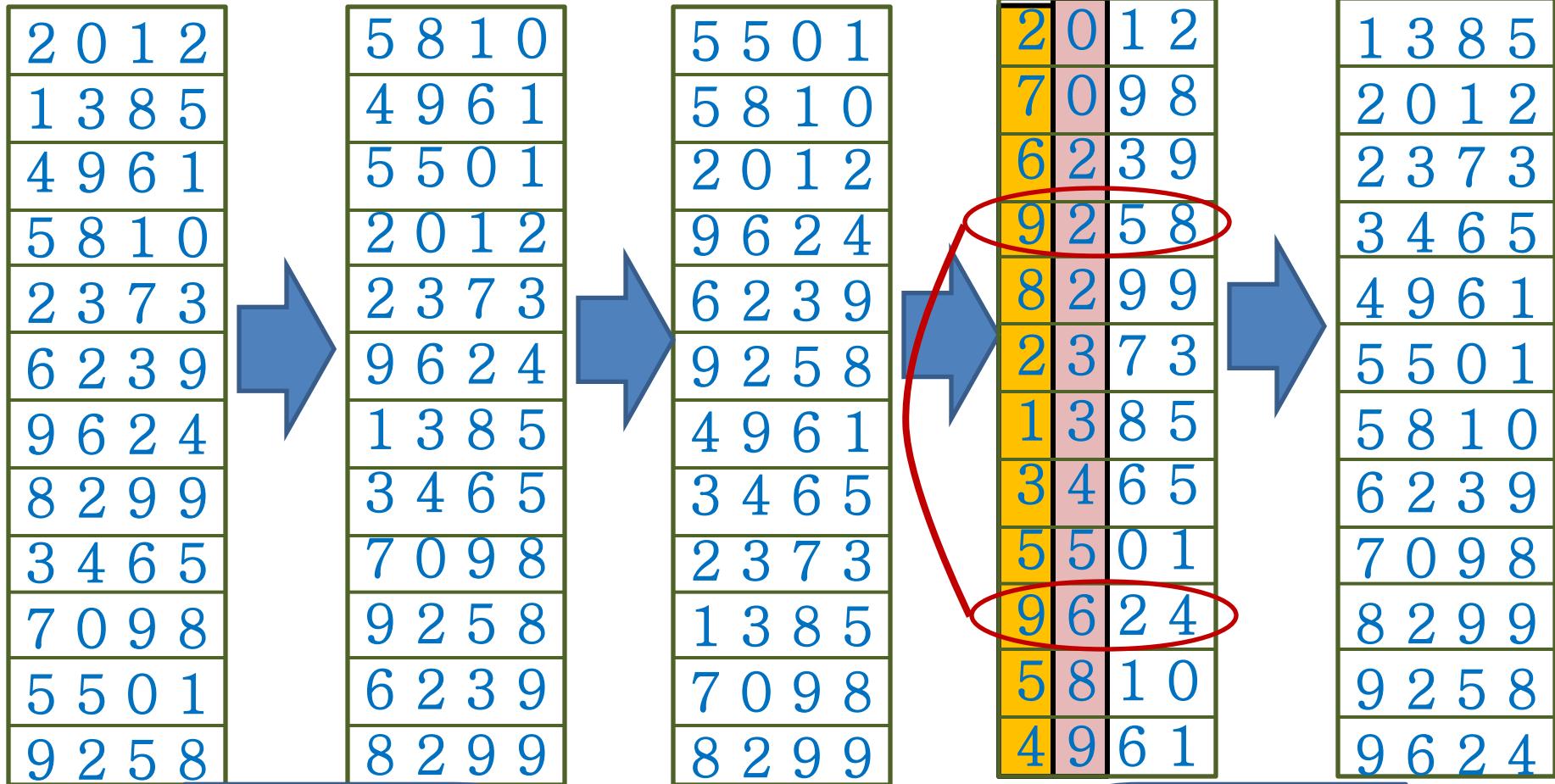
↓

5	5	0	1
5	8	1	0
2	0	1	2
9	6	2	4
6	2	3	9
9	2	5	8
4	9	6	1
3	4	6	5
2	3	7	3
1	3	8	5
7	0	9	8
8	2	9	9

2 0 1 2
7 0 9 8
6 2 3 9
9 2 5 8
2 3 7 3
1 3 8 5
3 4 6 5
5 5 0 1
9 6 2 4
5 8 1 0
4 9 6 1

Demonstration of Radix Sort through example

A



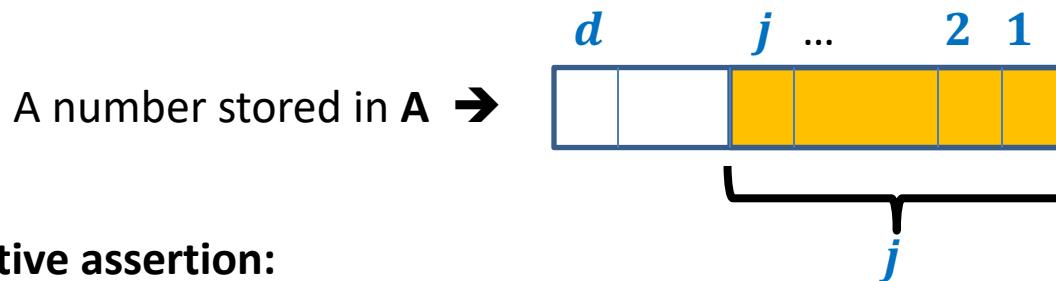
Can you see where we are exploiting the fact that
Countsort is a **stable** sorting algorithm ?

Radix Sort

RadixSort(A[0... $n - 1$], d , k)

```
{  For  $j=1$  to  $d$  do
    Execute CountSort(A,  $k$ ) with  $j$ th digit as the key;
    return A;
}
```

Correctness:



Inductive assertion:

At the end of j th iteration, array **A** is sorted according to the last j digits.

During the induction step, you will have to use the fact that **Countsort** is a **stable** sorting algorithm.

Radix Sort

RadixSort(A[0... $n - 1$], d , k)

```
{  For  $j=1$  to  $d$  do
    Execute CountSort(A, $k$ ) with  $j$ th digit as the key;
    return A;
}
```

Time complexity:

- A single execution of **CountSort(A, k)** runs in $O(n + k)$ time and $O(n + k)$ space.
- For $k < n$,
 - a single execution of **CountSort(A, k)** runs in $O(n)$ time.
 - Time complexity of radix sort = $O(dn)$.
- → Extra space used = $O(n)$

Question: How to use Radix sort to sort n integers in range $[0..n^t]$ in $O(tn)$ time and $O(n)$ space ?

Answer:

The diagram illustrates the conversion of bit widths. On the left, a blue box labeled "1 bit" has a green arrow pointing to a white box labeled "log n bits". Below this, a larger blue box labeled " $t \log n$ bits" has a green arrow pointing to a white box labeled " t bits".

d	k	Time complexity
$t \log n$	2	$O(tn \log n)$ 😕
t	n	$O(tn)$ 😊



Power of the word RAM model

- **Very fast** algorithms for **sorting integers**:
Example: n integers in range $[0..n^{10}]$ in $O(n)$ time and $O(n)$ space ?
- **Lesson:**
Do not always go after **Merge sort** and **Quick sort** when input is integers.
- **Interesting programming exercise:**
Compare **Quick sort** with **Radix sort** for sorting **long** integers.

Data Structures and Algorithms

(ESO207)

Lecture 40

- **Search data structure for integers : Hashing**

Data structures for searching

in **O(1)** time

Motivating Example

Input: a given set S of 1009 positive integers

Aim: Data structure for searching

Example

```
{  
123, 579236, 1072664, 770832456778, 61784523, 100004503210023, 19,  
762354723763099, 579, 72664, 977083245677001238, 84, 100004503210023,  
...  
}
```

Data structure : Array storing S in sorted order

Searching : Binary search

$\mathbf{O}(\log |S|)$ time

Can we perform
search in $\mathbf{O}(1)$ time ?

Problem Description

\mathcal{U} :

$\mathcal{S} \subseteq \mathcal{U}$,

$n = |\mathcal{S}|$,

$n \ll m$

A search query:

Aim: A data structure for a given set \mathcal{S}

that can facilitate search in $O(1)$ time

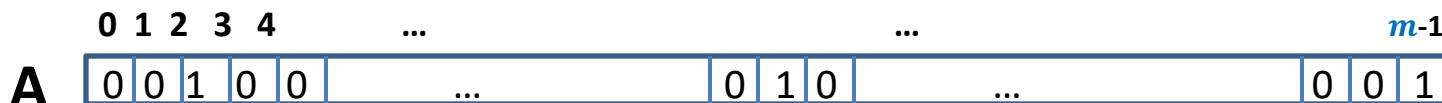
A trivial data structure for $O(1)$ search time

Build a 0-1 array A of size m such that

$A[i] = 1$ if $i \in S$.

$A[i] = 0$ if $i \notin S$.

Time complexity for searching an element in set S : $O(1)$.



This is a totally Impractical data structure because $n \ll m$!

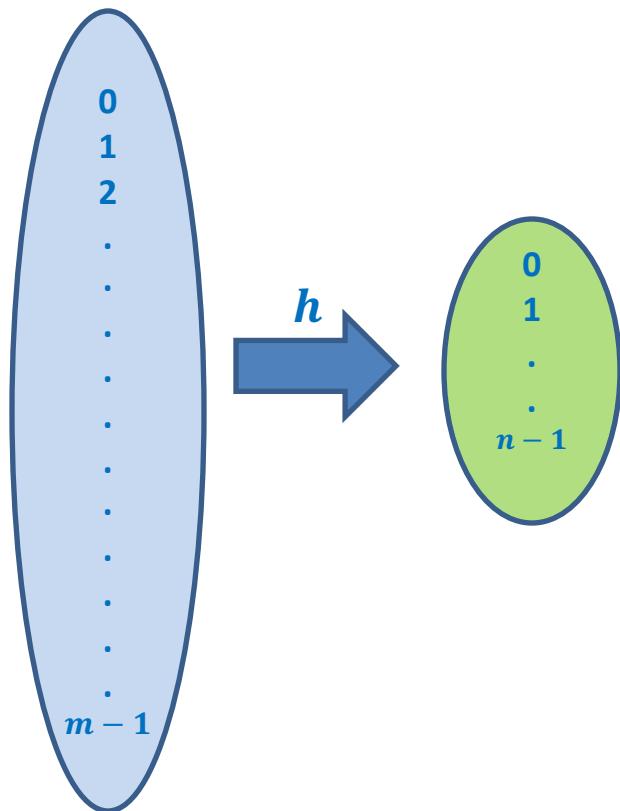
Example: n = few thousands, m = few trillions.

Question:

Can we have a data structure of $O(n)$ size that can answer a search query in $O(1)$ time ?

Answer: Hashing

Hash function



Hash function:

h is a mapping from U to $\{0, 1, \dots, n - 1\}$ with the following characteristics.

- Space required for h
- $h(i)$ computable in $O(1)$

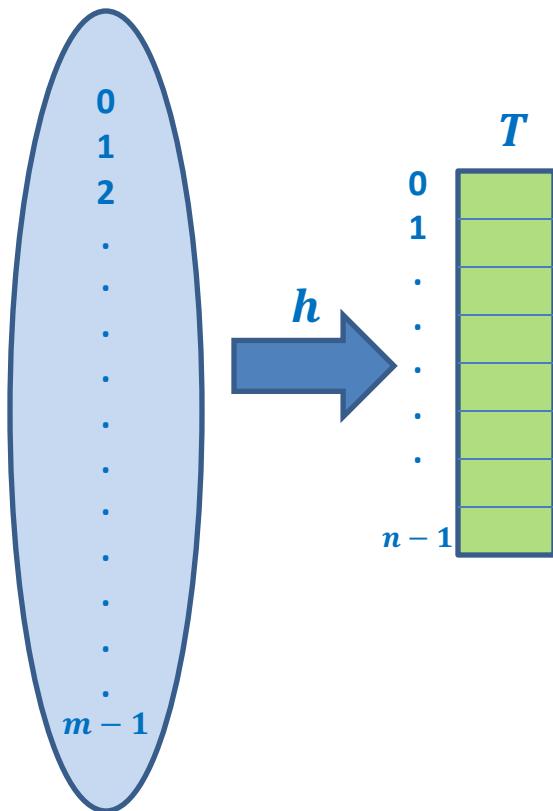
Example:

Hash value:

For a given hash function h , and $i \in U$.

$h(i)$ is called hash value of i

Hash function, hash value



Hash function:

h is a mapping from U to $\{0, 1, \dots, n - 1\}$ with the following characteristics.

- Space required for h : a few words.
- $h(i)$ computable in $O(1)$ time in word RAM.

Example: $h(i) = i \bmod n$

Hash value:

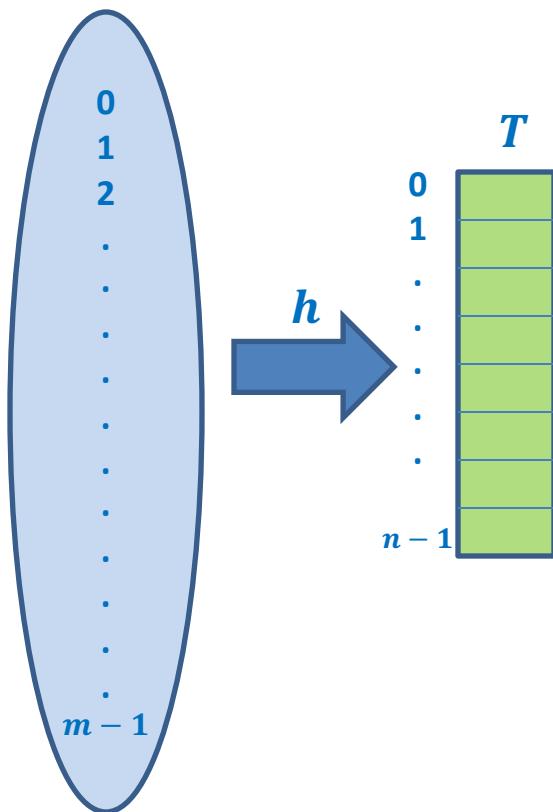
For a given hash function h , and $i \in U$.

$h(i)$ is called hash value of i

Hash Table:

An array $T[0 \dots n - 1]$

Hash function, hash value, hash table



Hash function:

h is a mapping from U to $\{0, 1, \dots, n - 1\}$ with the following characteristics.

- Space required for h : a few words.
- $h(i)$ computable in $O(1)$ time in word RAM.

Example: $h(i) = i \bmod n$

Hash value:

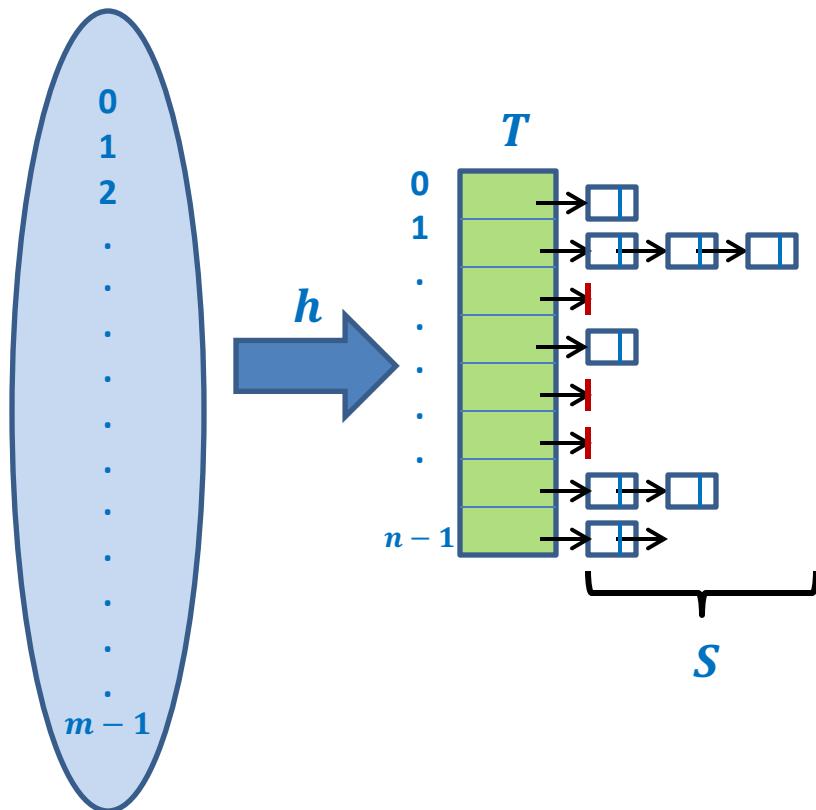
For a given hash function h , and $i \in U$.

$h(i)$ is called hash value of i

Hash Table:

An array $T[0 \dots n - 1]$

Hash function, hash value, hash table



Hash function:

h is a mapping from $\textcolor{blue}{U}$ to $\{0, 1, \dots, \textcolor{blue}{n} - 1\}$ with the following characteristics.

- **Space** required for h : a few **words**.
 - $h(i)$ computable in **O(1)** time in **word RAM**.

Example: $h(i) = i \bmod n$

Hash value:

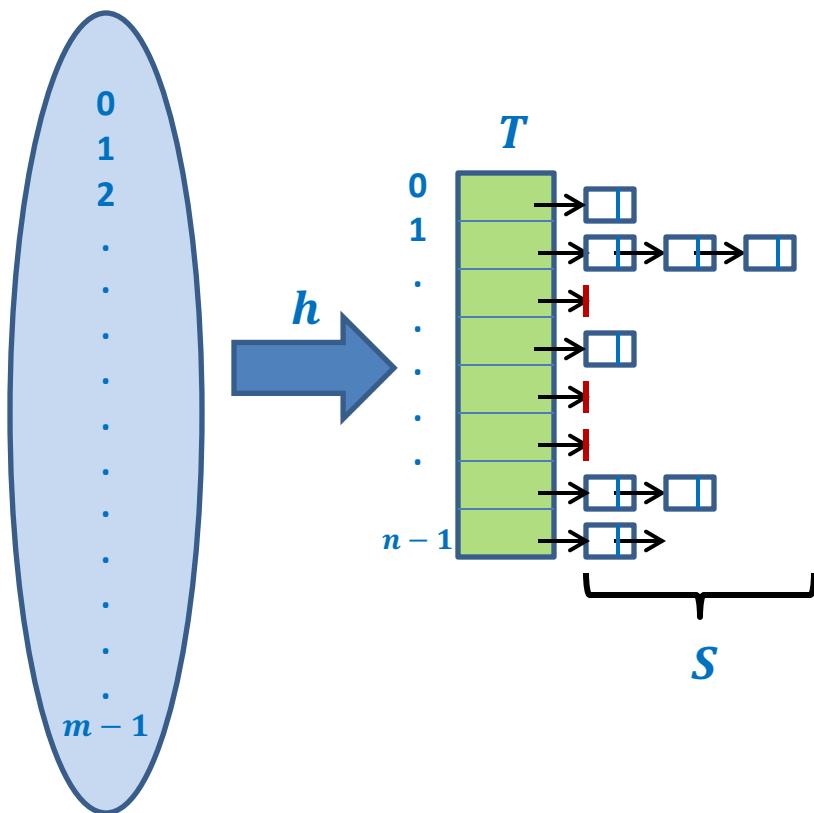
For a given hash function h , and $i \in U$.

$h(i)$ is called hash value of **i**

Hash Table:

An array $T[0 \dots n - 1]$ of pointers storing S .

Hash function, hash value, hash table



Question:

How to use (h, T) for searching an element $i \in U$?

Answer:

$$k \leftarrow h(i);$$

Search element i in the list $T[k]$.

Time complexity for searching:

O(length of the **longest** list in T).

Efficiency of Hashing depends upon hash function

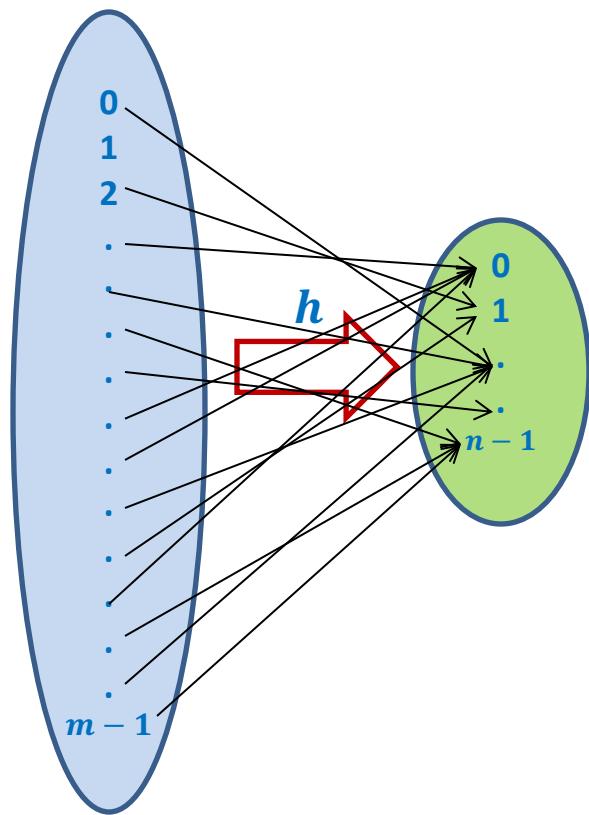
A hash function h is good if it can **evenly** distributes S .

Aim: To search for a good hash function for a given set S .



There **can not be** any hash function h which is good for every S .

Hash function, hash value, hash table



For every h , there exists a subset of $\left\lceil \frac{m}{n} \right\rceil$ elements from U which are hashed to same value under h .

So we can always construct a subset S for which all elements have same hash value

- All elements of this set S are present in a single list of the hash table T associated with h .
- $O(n)$ worst case search time.

Why does hashing work **so well** in Practice ?

$$h(i) = i \bmod n$$

Because the set **S** is usually a **uniformly random** subset of **U**.

Let us do a **theoretical analysis**
to prove this fact.

Why does hashing work so well in Practice ?

$$h(x) = x \bmod n$$

Let y_1, y_2, \dots, y_n denote n elements selected randomly uniformly from U to form S .

Question:

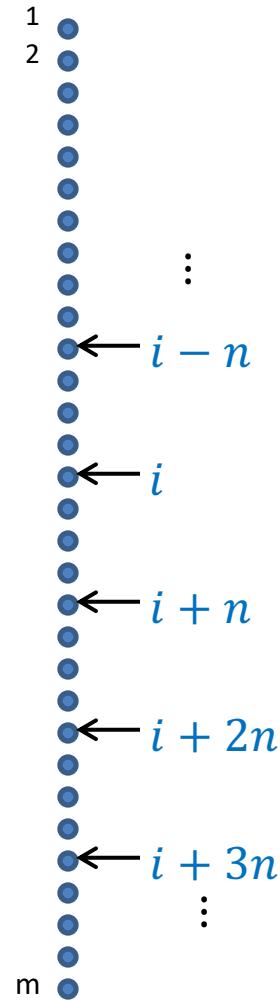
What is expected number of elements of S colliding with y_1 ?

Answer: Let y_1 takes value i .

$P(y_j \text{ collides with } y_1) = ??$

How many possible values can y_j take ? m - 1

How many possible values can collide with i ?



Why does hashing work so well in Practice ?

$$h(x) = x \bmod n$$

Let y_1, y_2, \dots, y_n denote n elements selected randomly uniformly from U to form S .

Question:

What is expected number of elements of S colliding with y_1 ?

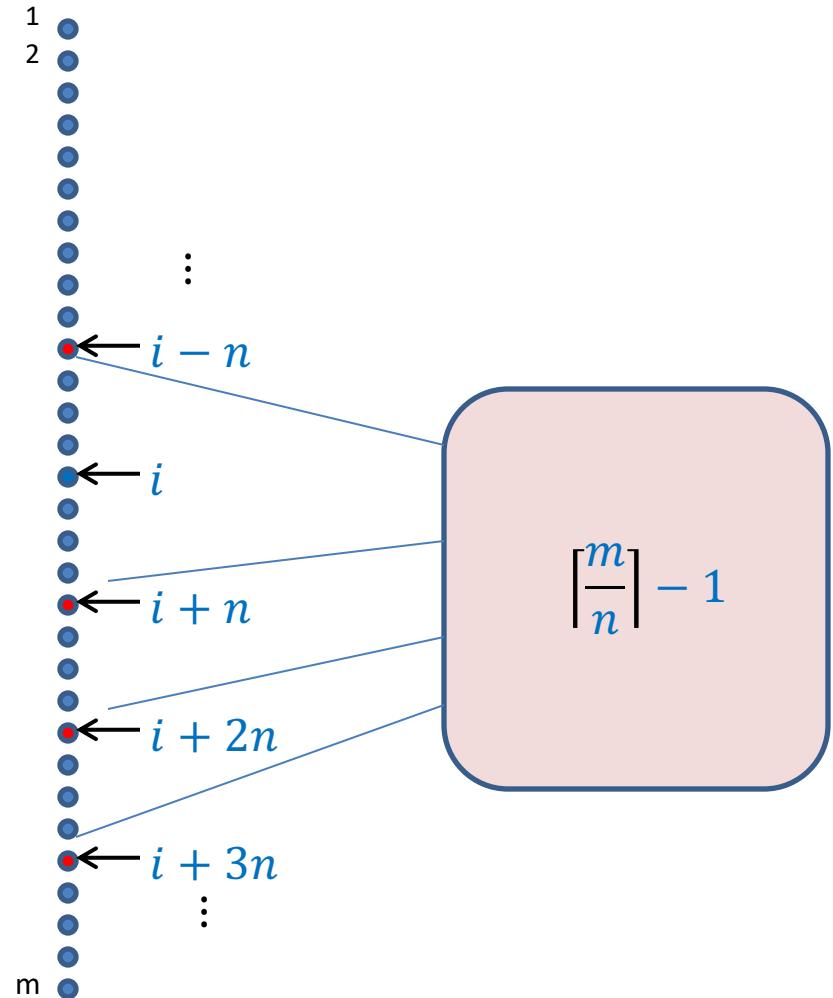
Answer: Let y_1 takes value i .

$P(y_j \text{ collides with } y_1) =$

$$\frac{\left[\frac{m}{n}\right] - 1}{m-1}$$

Expected number of elements of S colliding with y_1 =

$$\begin{aligned} &= \frac{\left[\frac{m}{n}\right] - 1}{m-1} (n-1) \\ &= O(1) \end{aligned}$$



Why does hashing work **so well** in Practice ?

Conclusion

1. $h(i) = i \bmod n$ works so well because
for a uniformly random subset of U ,
the **expected** number of collision at an index of T is $O(1)$.

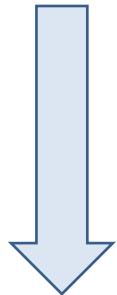
It is easy to fool this hash function such that it achieves $O(s)$ search time.
(do it as a simple exercise).

This makes us think:

“How can we achieve worst case $O(1)$ search time for a given set S .”

Hashing: theory

1953



1984

$U : \{0, 1, \dots, m - 1\}$

$S \subseteq U,$

$n = |S|,$

Theorem [FKS, 1984]:

A hash table and hash function can be computed in average $O(n)$ time for a given S s.t.

Space : $O(n)$

Query time: worst case $O(1)$

Ingredients :

- elementary knowledge of **prime numbers**.
- The algorithms use **simple randomization**.

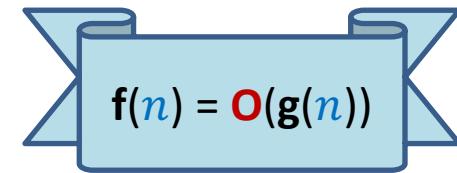
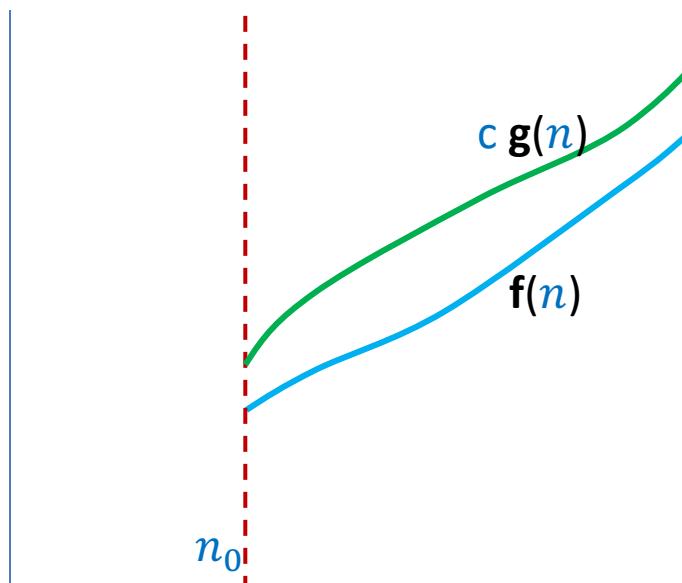
Order notation

Definition: Let $f(n)$ and $g(n)$ be any two increasing functions of n .

$f(n)$ is said to be

if there exist constants c and n_0 such that

$$f(n) \leq c g(n) \quad \text{for all } n > n_0$$



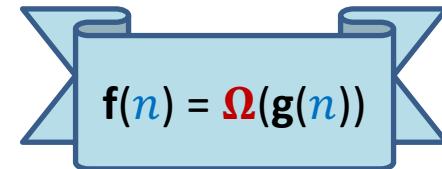
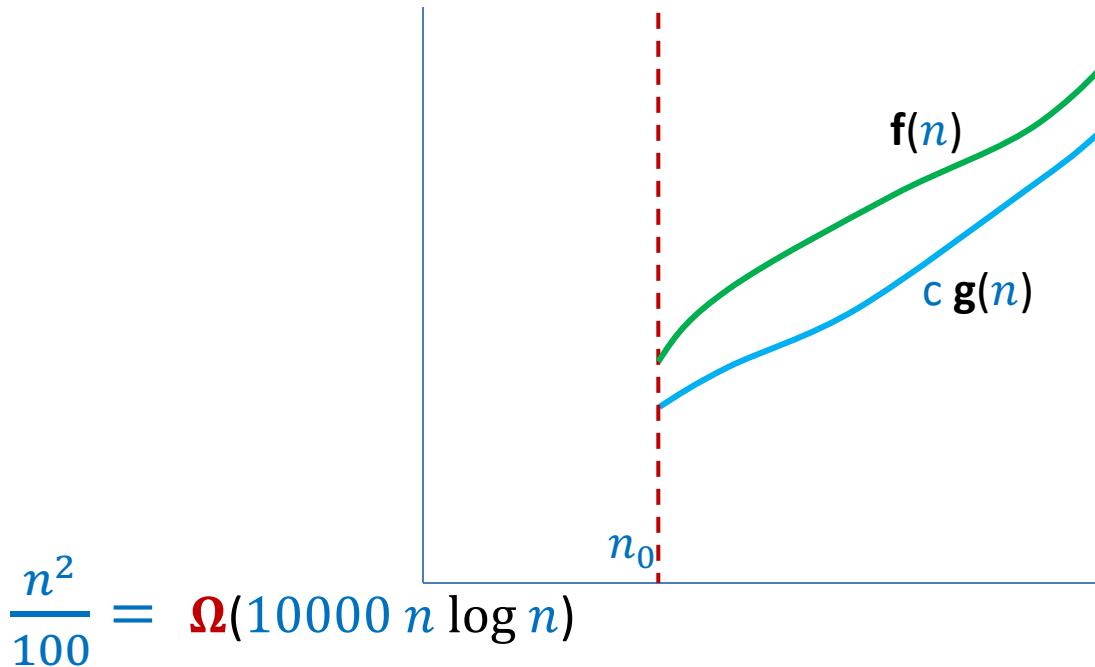
Order notation extended

Definition: Let $f(n)$ and $g(n)$ be any two increasing functions of n .

$f(n)$ is said to be

if there exist constants c and n_0 such that

$$f(n) \geq c g(n) \quad \text{for all } n > n_0$$



Order notation extended

Observations:

- $f(n) = O(g(n))$ if and only if $g(n) = \Omega(f(n))$

One more Notation:

If $f(n) = O(g(n))$ and $g(n) = O(f(n))$, then

$$g(n) = \Theta(f(n))$$

Examples:

- $\frac{n^2}{100} = \Theta(10000 n^2)$

- Time complexity of Quick Sort is $\Omega(n \log n)$
- Time complexity of Merge sort is $\Theta(n \log n)$