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;

($S_{<x}$, $S_{>x}$)← 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← 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 $A[l \ldots r]$ 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

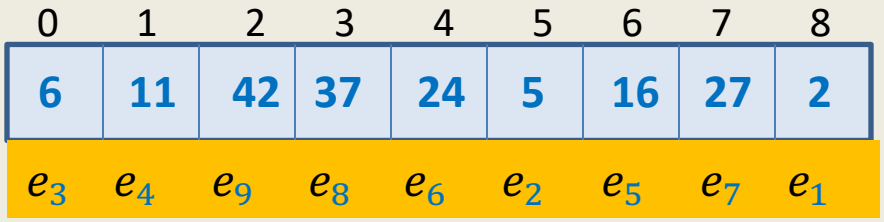
Analyzing average time complexity of QuickSort

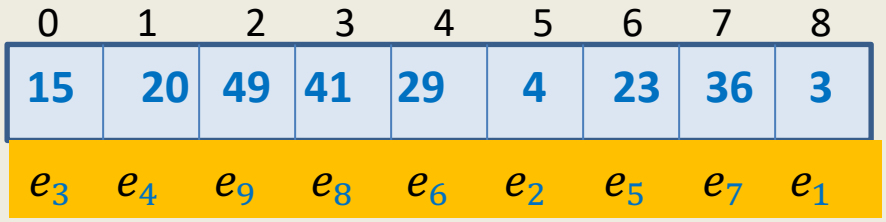
Assumption (just for a neat analysis):

- 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.





Let e_i : ith **smallest** element of 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

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

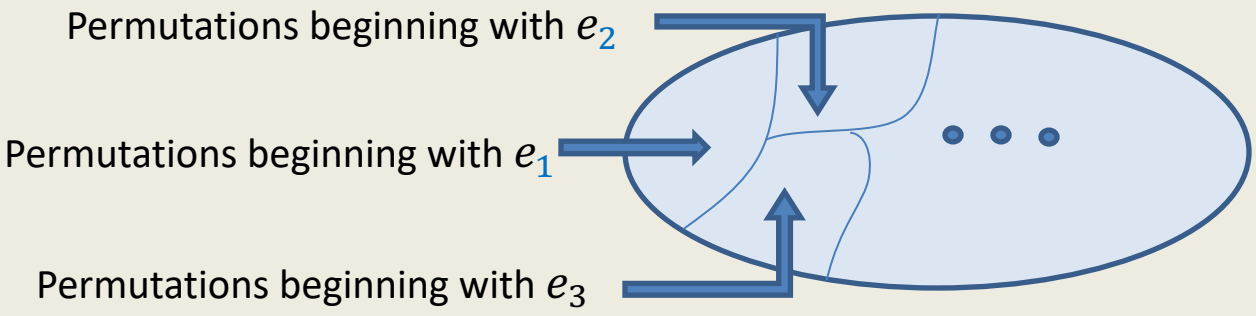
(average over all possible permutations of $\{e_1, e_2, \ldots, e_n\}$)

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

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

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

Analyzing average time complexity of QuickSort



All $n!$ permutations of $\{e_1, e_2, \ldots, e_n\}$

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

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

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

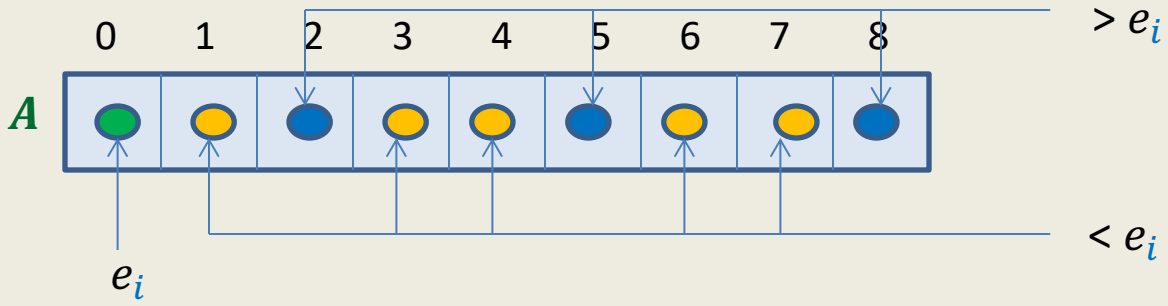
Let $G(n, i)$ be the average running time of **QuickSort** over **P**(i).

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

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

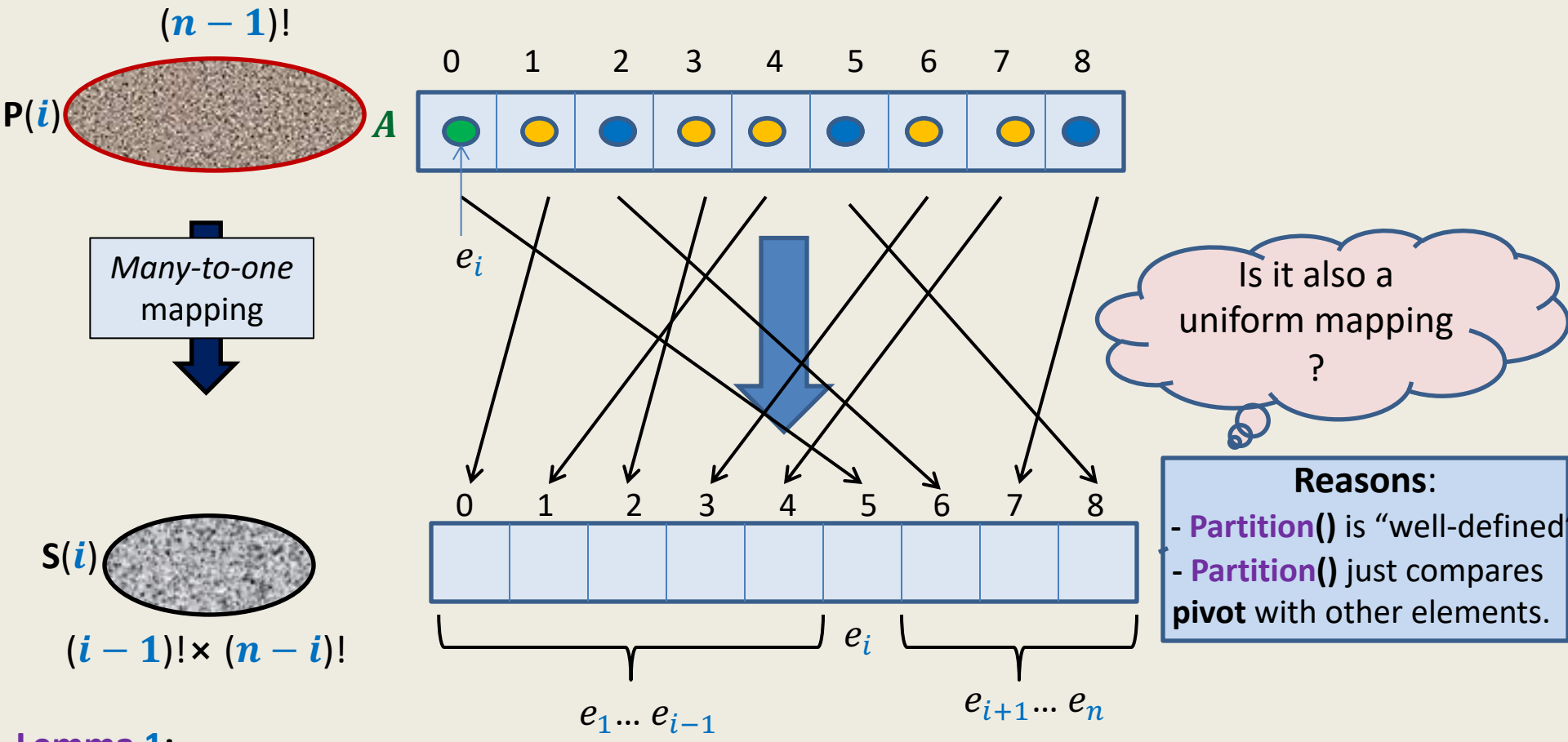
Observation: We now need to derive an expression for $G(n, i)$.For this purpose, we need to have a closer look at the execution of **QuickSort** over **P**(i).

Quick Sort on a permutation from P(i).



What happens during
Partition(A,0,8)

Quick Sort on a permutation from P(i).

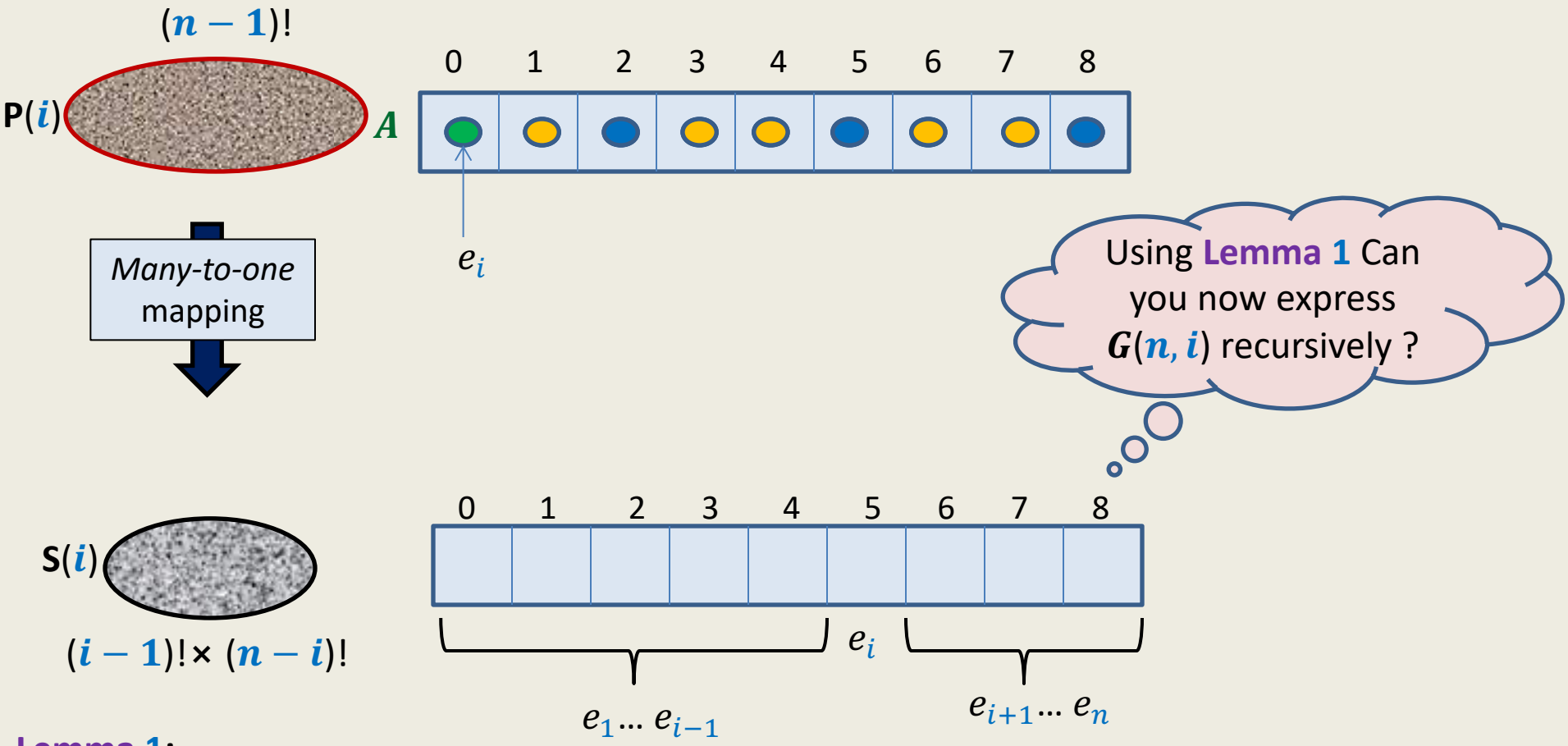


Lemma 1:

There are <u>exactly</u> $\binom{n-1}{i-1}$ 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).



Lemma 1:

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

Analyzing average time complexity of QuickSort

$G(n, i)$ =

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

We showed previously that :

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

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

$T(1) = c$

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

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

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

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

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

$T(n) \le \frac{2}{n}\sum_{i=1}^{n-1}(ai \log i + b) + dn$

$\le \frac{2}{n}(\sum_{i=1}^{n-1} ai \log i) + 2b + dn$

$= \frac{2}{n}(\sum_{i=1}^{n/2} ai \log i + \sum_{i=\frac{n}{2}+1}^{n-1} ai \log i) + 2b + dn$

$\le \frac{2}{n}(\sum_{i=1}^{n/2} ai \log n/2 + \sum_{i=\frac{n}{2}+1}^{n-1} ai \log n) + 2b + dn$

$= \frac{2}{n}(\sum_{i=1}^{n-1} ai \log n - \sum_{i=1}^{n/2} ai) + 2b + dn$

$= \frac{2}{n}\left(\frac{n(n-1)}{2} a \log n - \frac{\frac{n}{2}(\frac{n}{2}+1)}{2} a\right) + 2b + dn$

$\le a(n-1)\log n - \frac{n}{4}a + 2b + dn$

$\le an\log n + b - \frac{n}{4}a + b + dn$

$\le an\log n + b$ for $a > 4(b + d)$

Analyzing average time complexity of QuickSort

Part 3

Solving the recurrence exactly

Some elementary tools

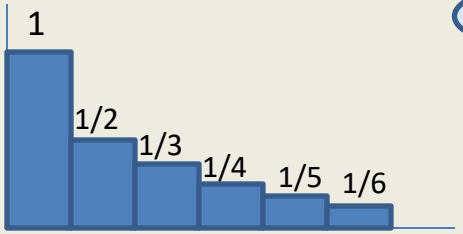
$$\mathbf{H}(n) = \sum_{i=1}^{n} \frac{1}{i}$$

Question: How to approximate $\mathbf{H}(n)$ **?**

Answer: $\mathbf{H}(n) \rightarrow \log_e n +$ $^{\gamma}$, as n increases

where $^{\gamma}$ is **Euler's constant ~0.58**

Hint: →



Look at this figure, and relate it to the curve for function f(x)= 1/x and its integration…

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.

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

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

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

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

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

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

Subtracting 2 from 1, we get

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

➔$nT(n) - (n+1)T(n-1) = 2(n-1)$

Question: How to solve/simplify it further ?

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

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

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

Question: How to simplify RHS ?

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

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

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

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

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

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

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

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

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

$$\ldots \quad = \quad \ldots$$

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

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

Hence $g(n) = \frac{4}{n+1} + (2\sum_{j=2}^{n} \frac{1}{j}) - 2 = \frac{4}{n+1} + (2\sum_{j=1}^{n} \frac{1}{j}) - 4$

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

➔ $T(n) = (n+1)(\frac{4}{n+1} + 2\mathbf{H}(n) - 4)$

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

$$T(n) = 2(n+1)\mathrm{H}(n) - 4n$$
$$= 2(n+1)\log_e n + 1.16\,(n+1) - 4n$$
$$= 2n\log_e n - 2.84\,n + O(1)$$
$$= 2n\log_e n$$

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

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

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 ??

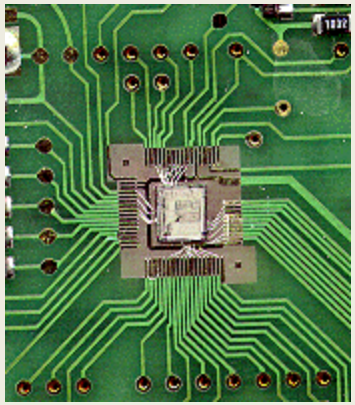
Finding shortest route between cities

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.

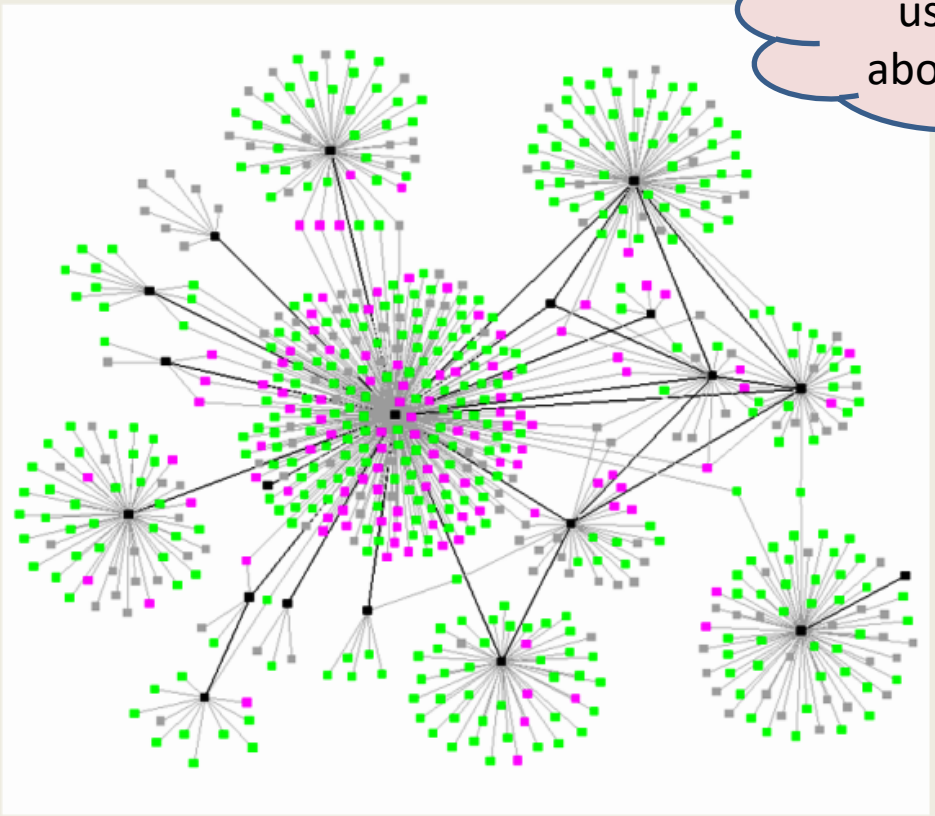
Embedding
an integrated circuit on mother board



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

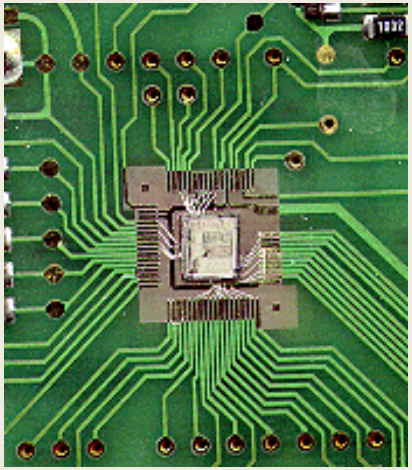
degree distribution

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

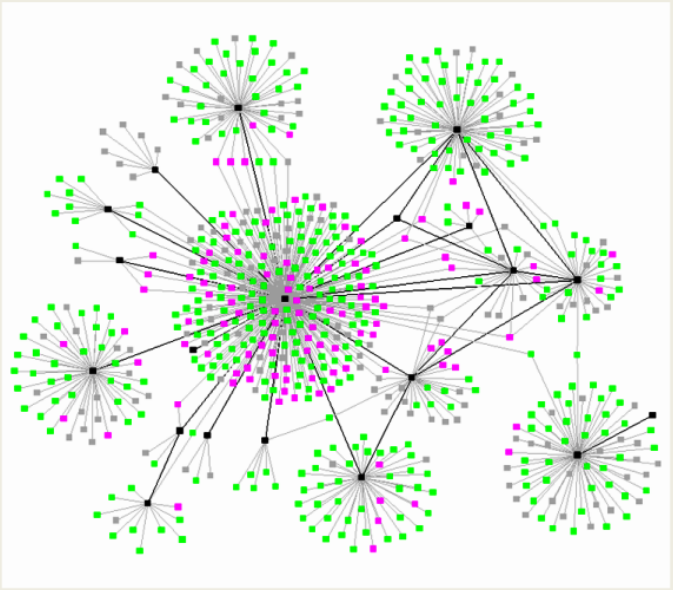
Visit the site https://en.wikipedia.org/wiki/Six_degrees_of_separation

How will you model these problems ?

I

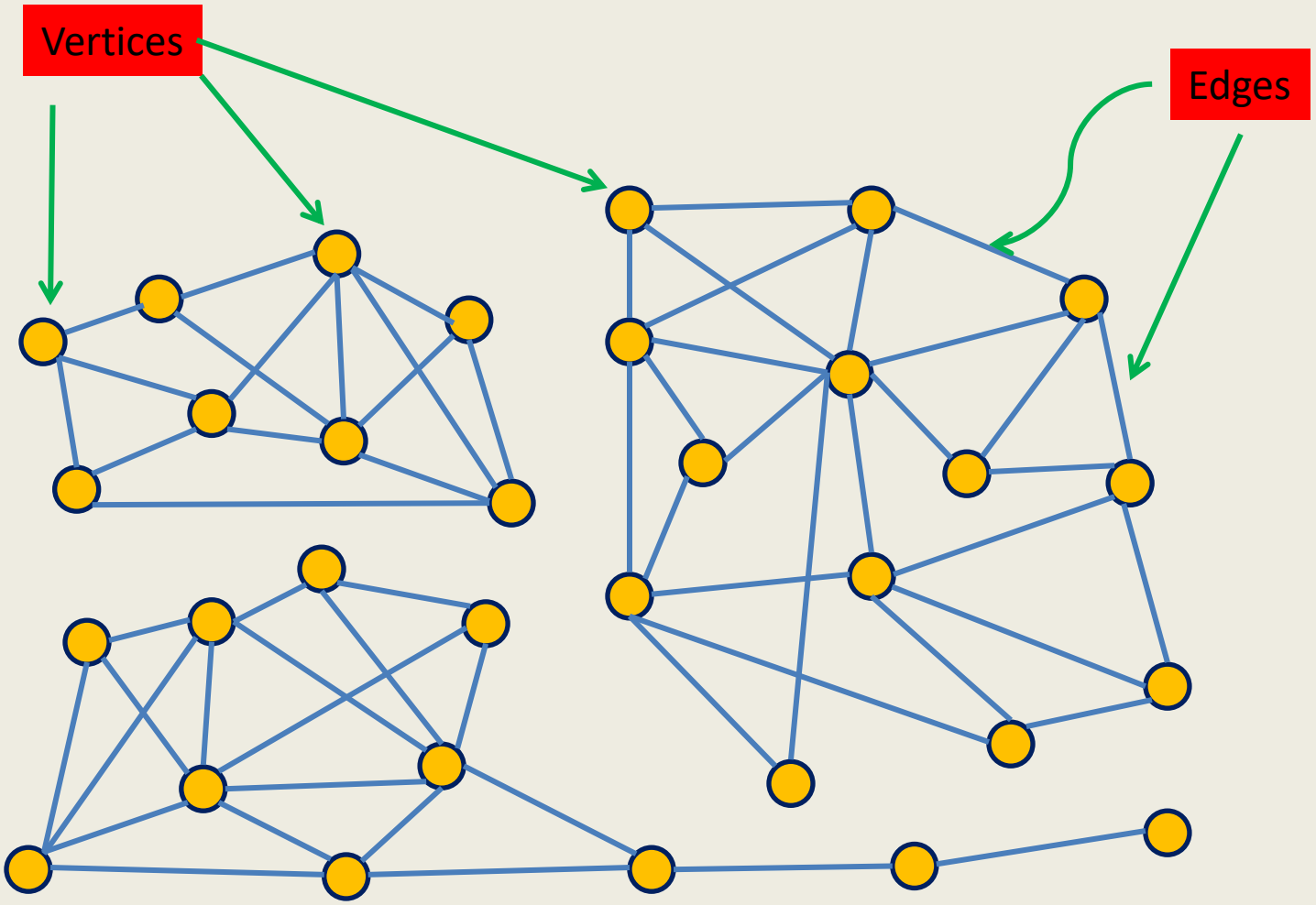


II



III

Graph



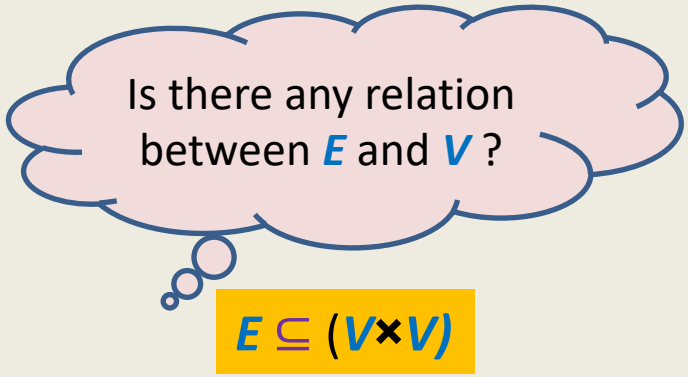
Graph

Definitions, notations, and terminologies

Graph

A graph G is defined by two sets

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

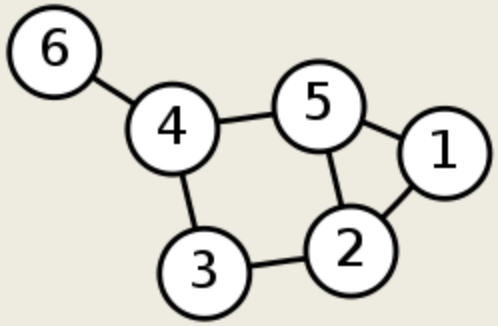


Notation:

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

Types of graphs

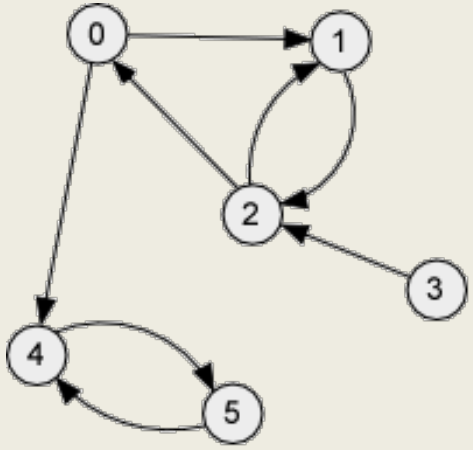
Undirected Graph



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, $m \leq n(n-1)$

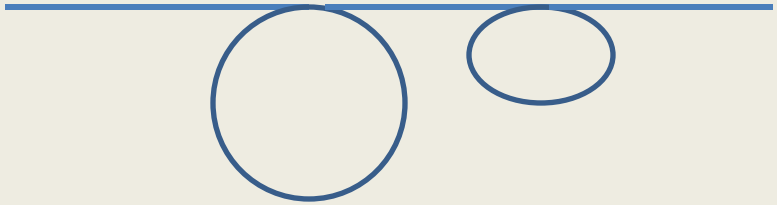
For undirected graphs, $m \leq n(n-1)/2$

Walks, paths, and cycles

Walk:

A sequence <v_0, v_1, …, v_k> of vertices is said to be a **walk** from x to y

- $x = v_0$
- $y = v_k$
- For each $i<k$, $(v_i, v_{i+1}) \in E$

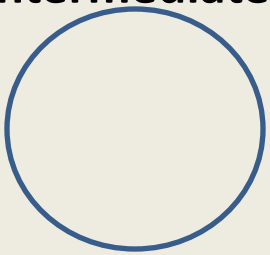


Path:

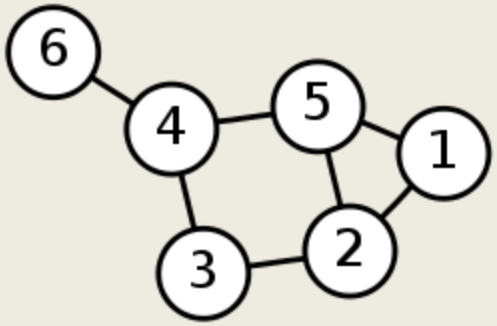
A walk <v_0, v_1, …, v_k> on which no vertex appears twice.

Cycle:

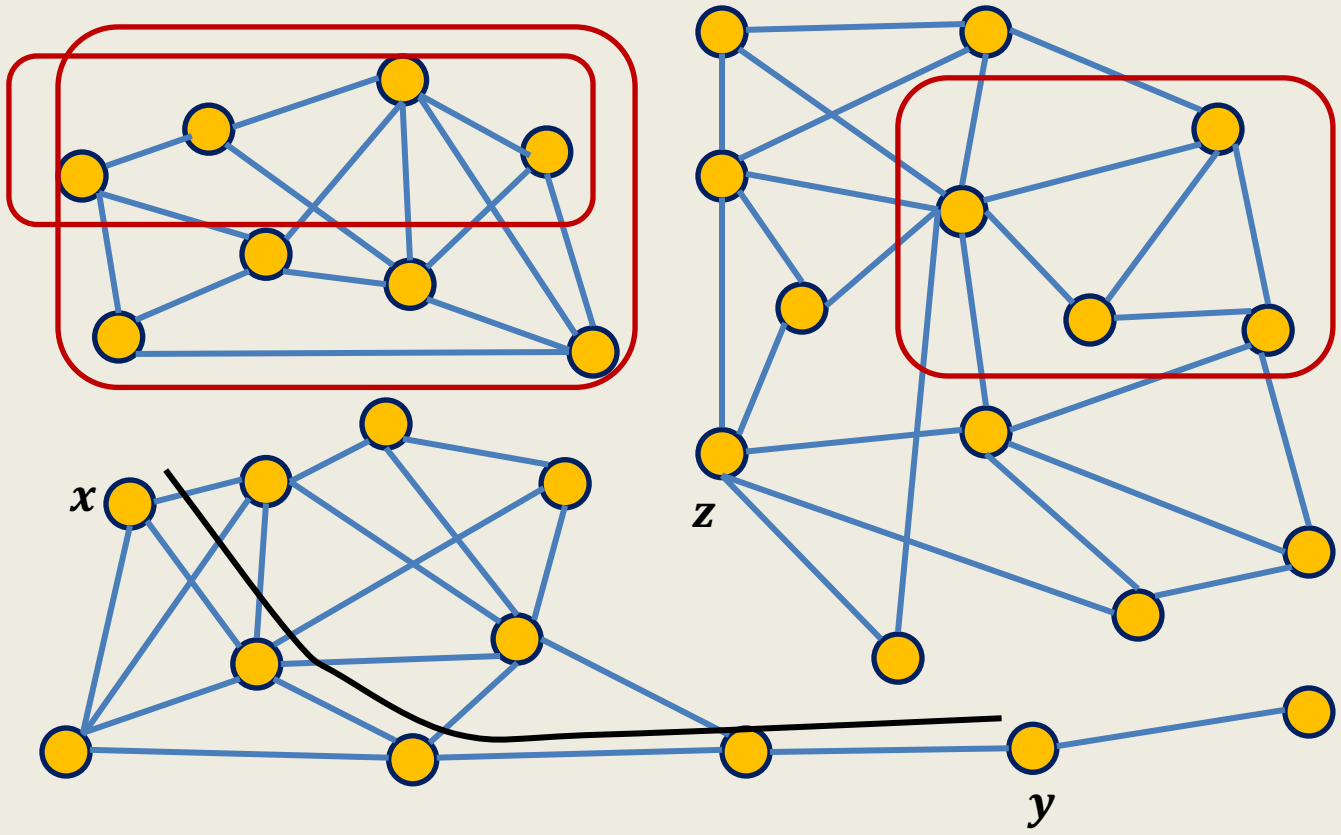
A walk <v_0, v_1, …, v_k> where no **intermediate** vertex gets repeated and $v_0 = v_k$



Examples



- **<1,5,4>** is a **walk** from **1** to **4.**
- **<1,3,2,5>** is **not** a **walk.**
- **<1,2,5,2,3,4,5,4,6>** is a **walk** from **1** to **6.**
- **<1,2,5,4,6>** is a **path** from **1** to **6.**
- **<2,3,4,5,2>** 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.

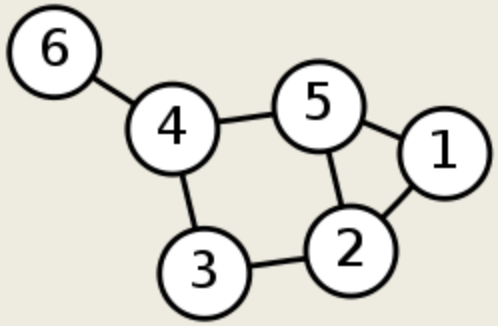
Data Structures for Graphs

Vertices are always numbered

$1, \ldots, n$ Or $0, \ldots, n-1$

Link based data structure for graph

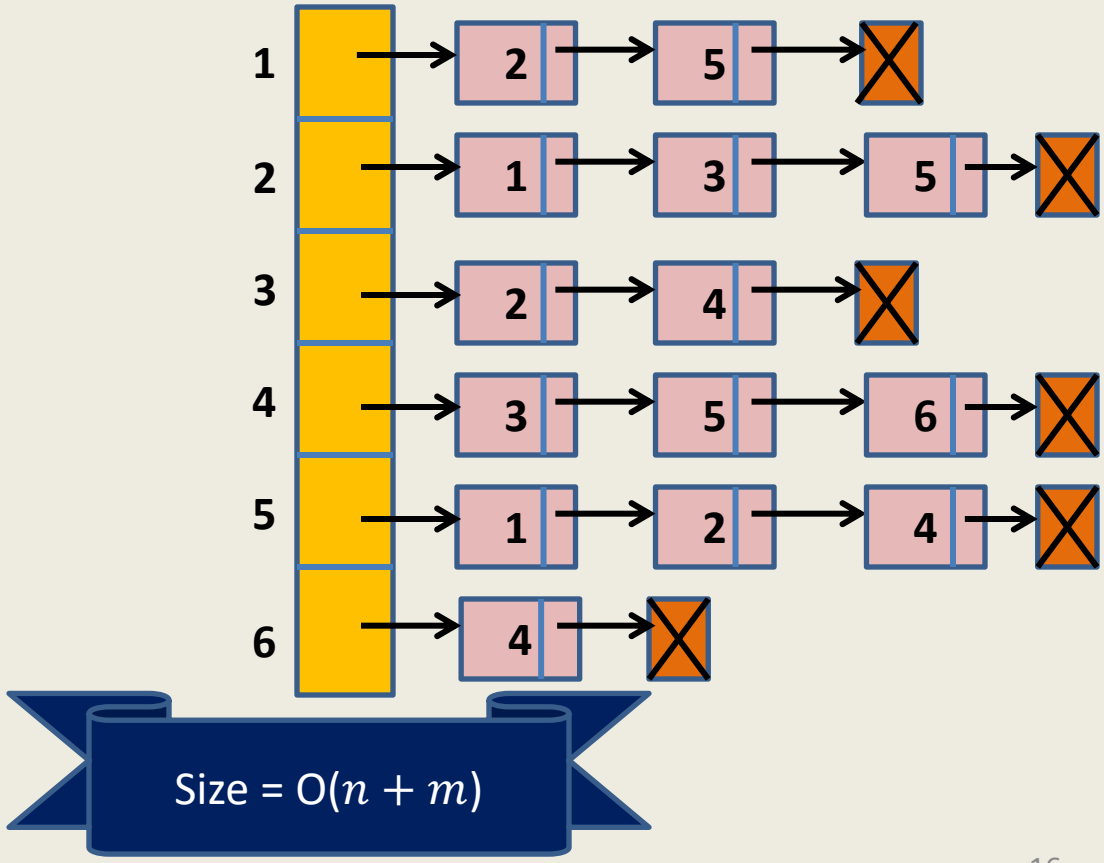
Undirected Graph



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

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

Adjacency Lists



Link based data structure for graph

Advantage of Adjacency Lists :

- Space efficient
- Computing all the neighbors of a vertex in optimal time.

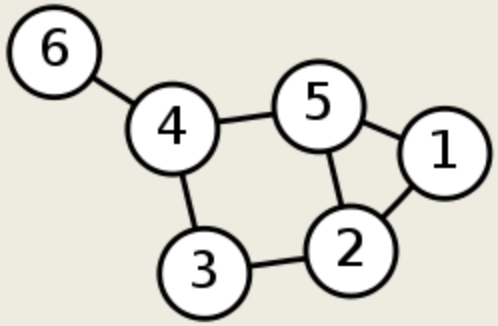
Disadvantage of Adjacency Lists :

- 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*** for any two vertices ***x*** and ***y***. in **O**(1) time

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.

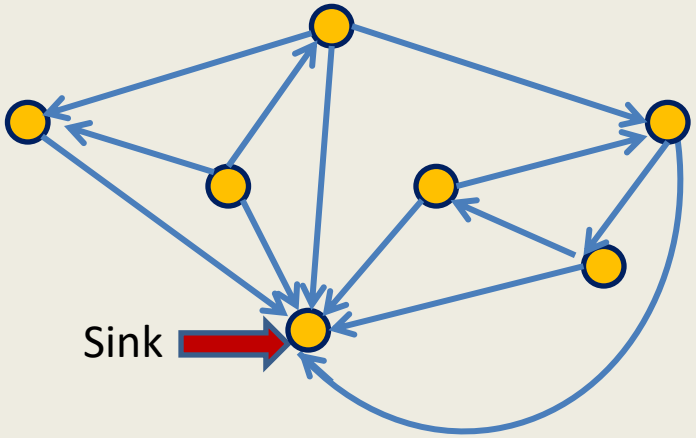
But, there are a few **exceptions**

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***.

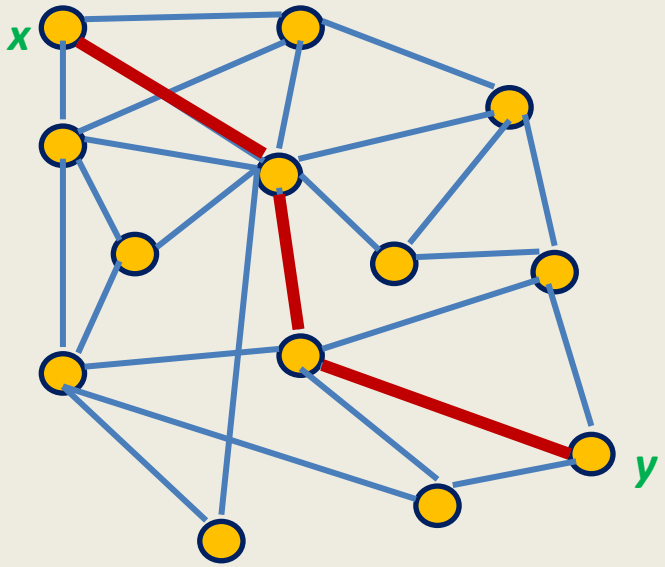
Graph traversal

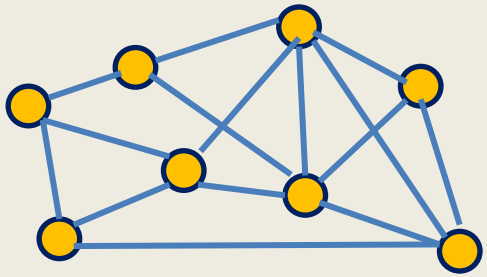
Topic for the next class

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 ?

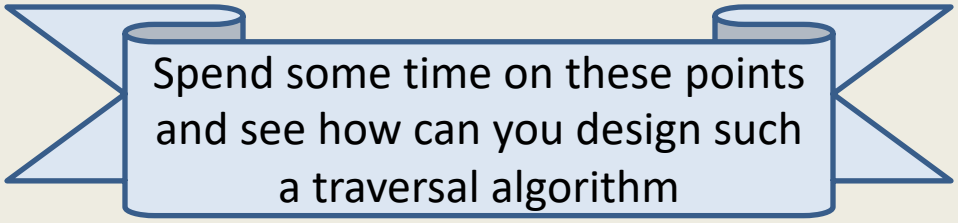
  (keeping track of vertices already visited)

- **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

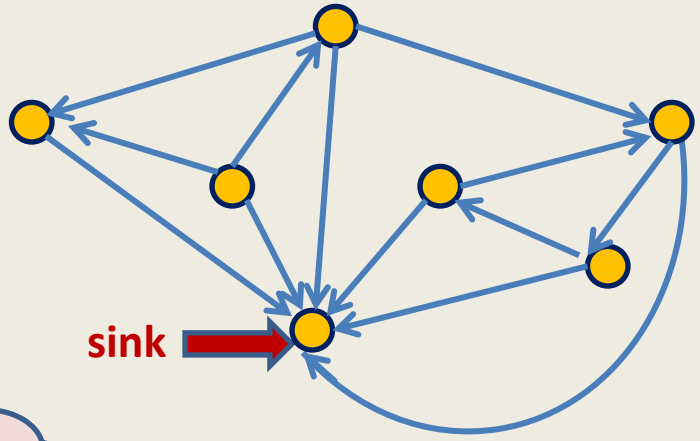
- Finding a sink in a directed graph
- Graph Traversal
  - Breadth First Search Traversal and its simple applications

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*** ?

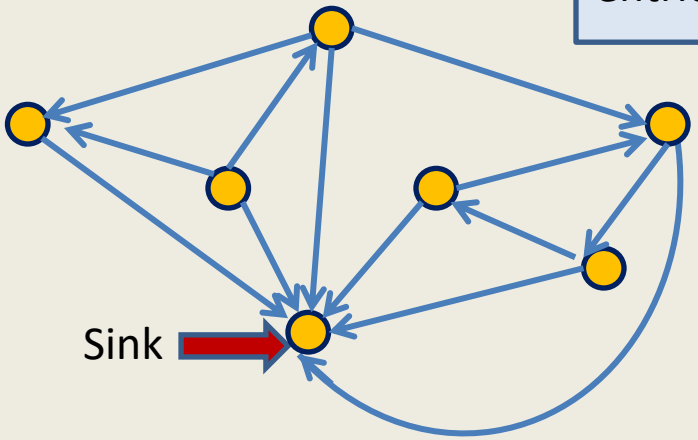
At most **1**.

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**.

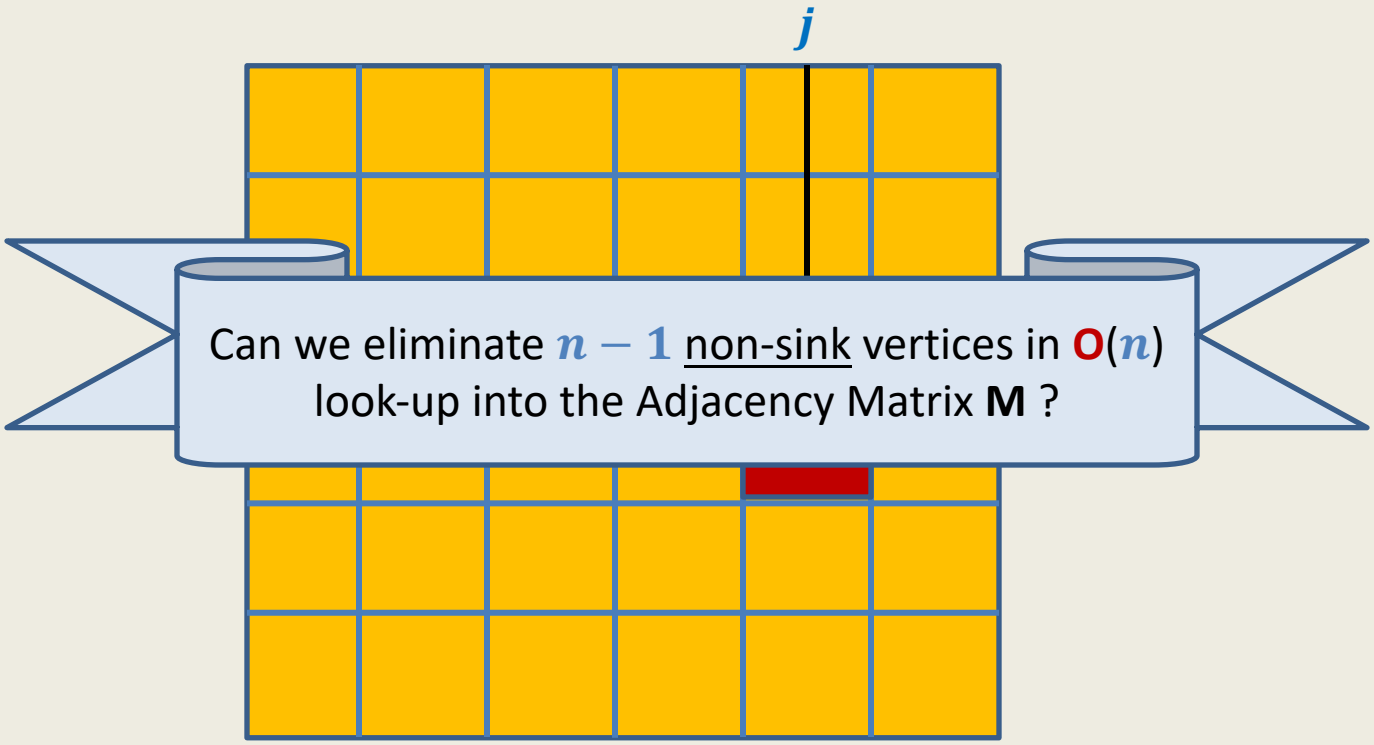


Question: Can we verify efficiently whether any given vertex i is a sink ?

Answer: Yes, in $O(n)$ time only ☺

Look at ith **row** and ith **column** of **M**.

Key idea



If **M**[i, j] = **0**, then j can not be sink

If **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← 0;
For(i=1 to n − 1)
 {
        If (M[s,i] = ? ) ….?...;
 }
 Verify if s is a sink and output accordingly.
```

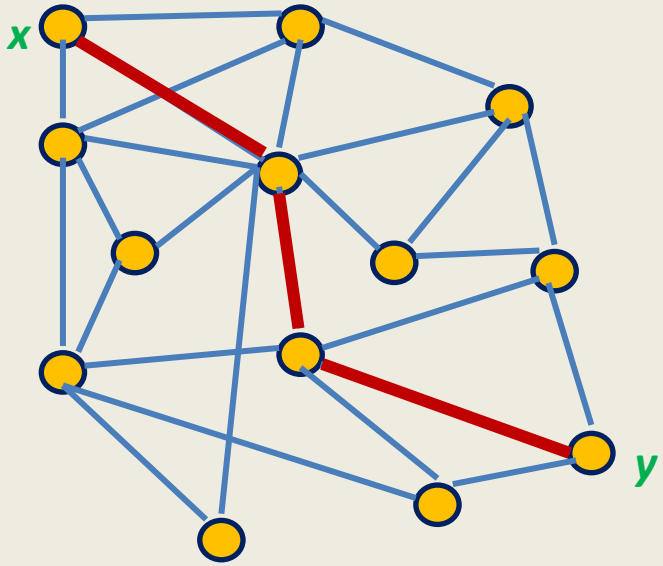
(Fill in the details of this pseudo code as a **Homework**.)

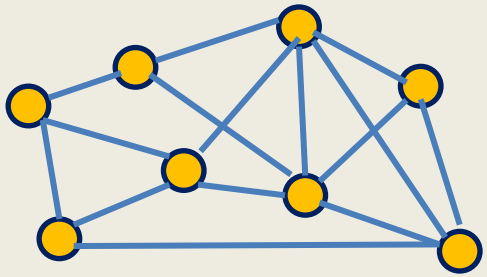
What is Graph traversal ?

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***.

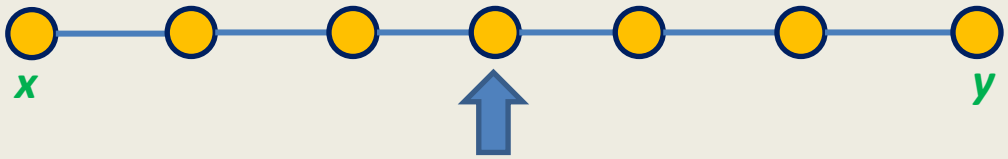
Breadth First Search traversal

We shall introduce this traversal technique through an interesting problem.

computing distances from a vertex.

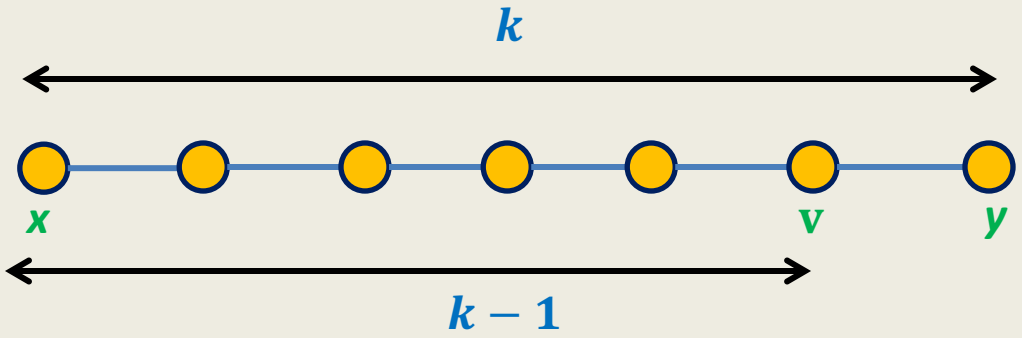
Notations and Observations

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 <x, …,v,y> is a path of length k from x to y,

then what is the length of the path <x, …,v> ?

Answer: $k-1$

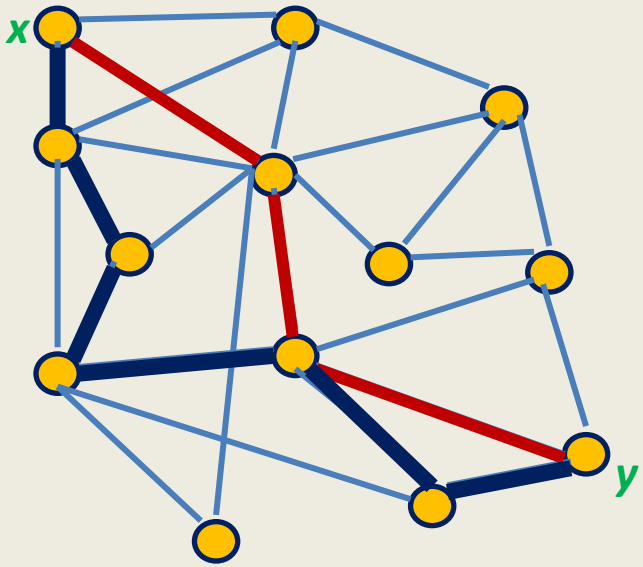
Question: What can be the maximum length of any path in a graph ?

Answer: $n-1$

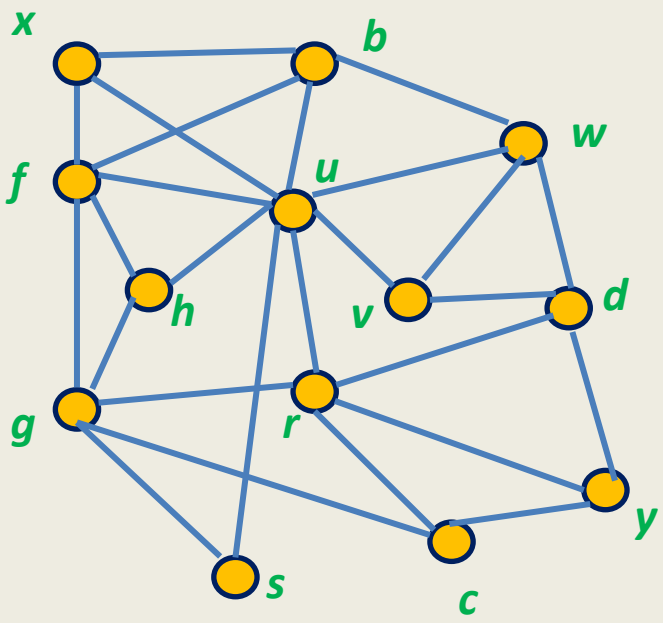
Notations and Observations

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*.



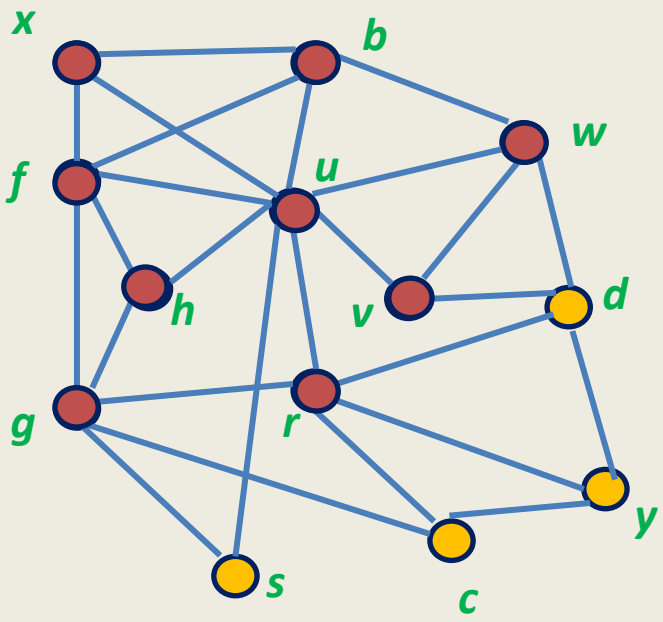
Shortest Paths
in Undirected Graphs



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***: {*x*}

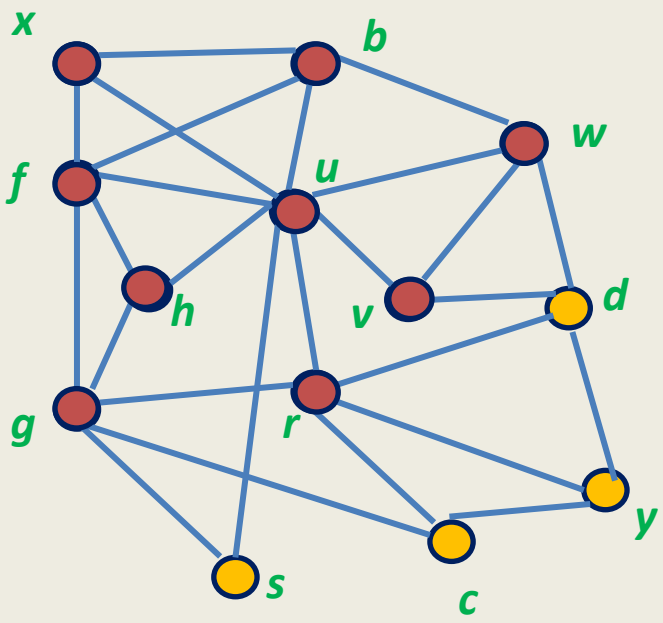
V_1 : Vertices at distance **1** from ***x***: {*f*,*u*,*b*}

V_2 : Vertices at distance **2** from ***x***: {*g*,*h*,*s*,*r*,*v*,*w*}

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



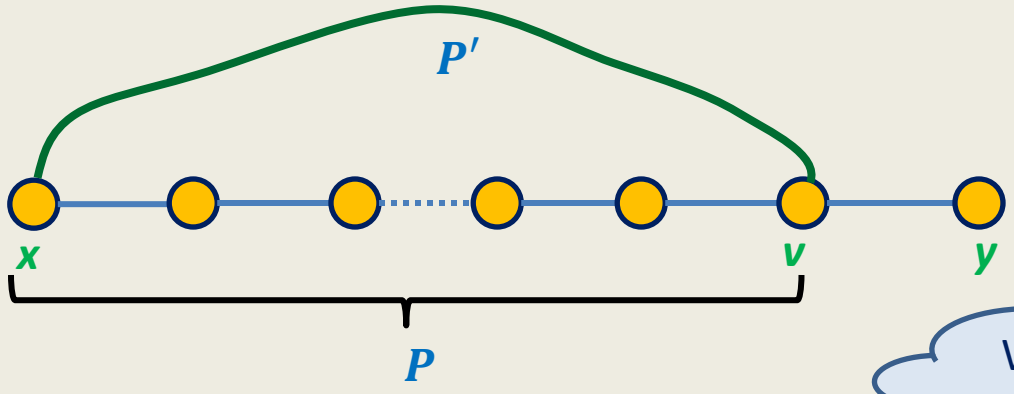
V_0 : Vertices at distance **0** from ***x***: {*x*}

V_1 : Vertices at distance **1** from ***x***: {*f*,*u*,*b*}

V_2 : Vertices at distance **2** from ***x***: {*g*,*h*,*s*,*r*,*v*,*w*}

Why ?

An important property of shortest paths



A shortest path between x and y

What can you say about P ?

Observation:

If $<x, \ldots, v, y>$ is a shortest path from x to y,

then $<x, \ldots, v>$ is also a shortest path.

Proof:

Suppose $P = <x, \ldots, v>$ is not a shortest path between x and v.

Then let P' be a shortest path between x and v.

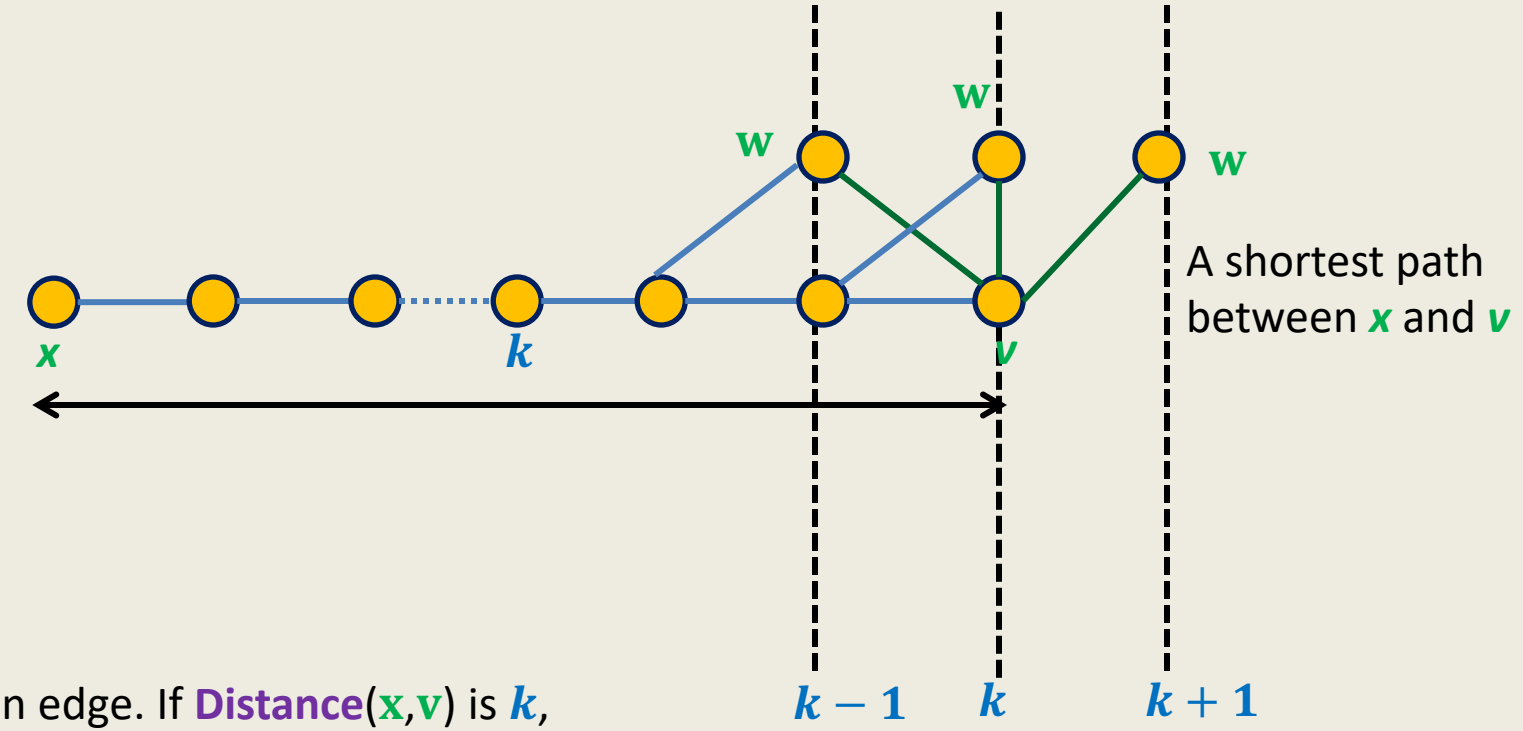
Length(P') < Length(P).

Question: What happens if we concatenate P' with edge (v, y) ?

Answer: a path between x and y shorter than the shortest-path $<x, \ldots, v, y>$.

➔ Contradiction.

An important question



Question:

Let (v,w) be an edge. If Distance(x,v) is k,

then what can be 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 to distinguish the neighbors of V_i which belong to V_{i+1} from those which belong to V_j, $j \leq i$?

How can we compute V_{i+1} ?

Key idea: compute V_i's in increasing order of i.

Initialize **Distance**[**v**] ← ∞ of each vertex **v** in the graph.

Initialize **Distance**[**x**] ← 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 **Distance**[**v**] = a number ≤ i

  If **v** is in V_{i+1}, **Distance**[**v**] = ∞

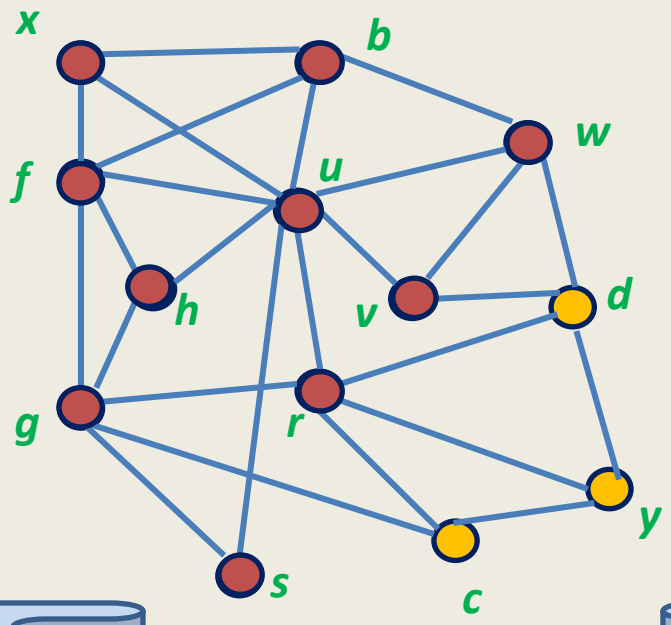
We can thus distinguish the neighbors of V_i which belong to V_{i+1} from those which belong to V_j.

A neat algorithm for computing distances from *x*

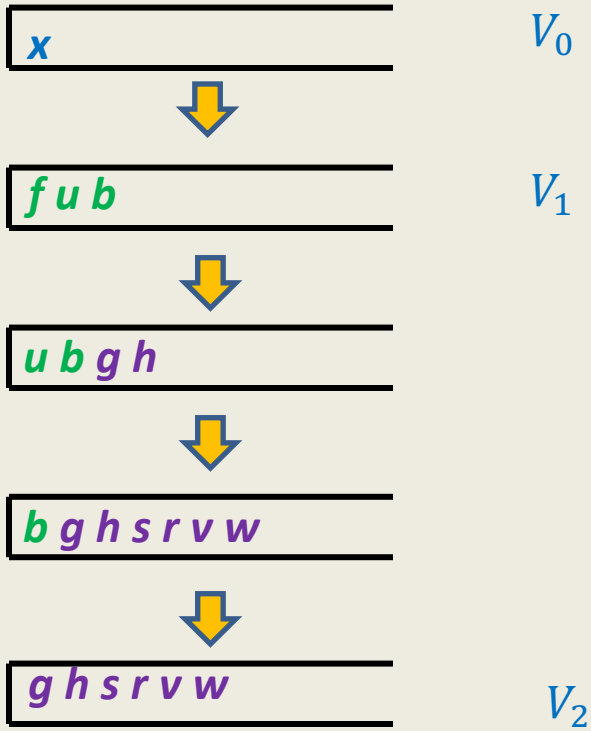
we need an algorithm which traverses/visits the vertices in **non-decreasing** order of **distances** from *x*

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

unvisited, mark it visited and

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) ← 0;
   Enqueue(x,Q);
   While( Not IsEmptyQueue(Q) )
   {        v ←Dequeue(Q);
            For each neighbor w of v
            {
                     if (Distance(w) = ∞)
                     {  Distance(w) ← Distance(v) +1 ;
                        Enqueue(w, Q); ;
                     }
            }
   }
```

Running time of BFS traversal

```
BFS(G, x)
   CreateEmptyQueue(Q);
   Distance(x) ← 0;
   Enqueue(x,Q);
   While( Not IsEmptyQueue(Q) )
   {        v ←Dequeue(Q);
            For each neighbor w of v
            {
                     if (Distance(w) = ∞)
                     {  Distance(w) ← Distance(v) +1 ;
                        Enqueue(w, Q); ;
                     }
            }
   }
```

O(deg(v))

A vertex can enter queue
at most once.
Prove this claim first.

Running time of 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**.

We shall discuss this proof in the next lecture. Please think over it for some time **before next lecture.**

Data Structures and Algorithms

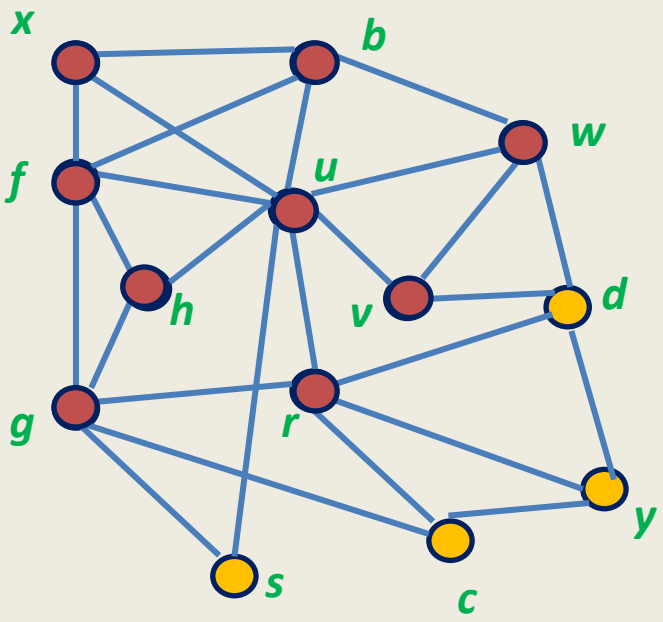
(ESO207)

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) ← ∞ , and Visited(v) ← false.
{  CreateEmptyQueue(Q);
   Distance(x) ← 0;
   Enqueue(x,Q);    Visited(x) ← true;
   While(Not IsEmptyQueue(Q))
   {   v ←Dequeue(Q);
       For each neighbor w of v
       {
           if (Distance(w) = ∞)
           {   Distance(w) ← Distance(v) +1 ;   Visited(w) ← true;
               Enqueue(w, Q);
           }
       }
   }
```

Observations about BFS(x)

Observations:

- Any vertex v enters the queue at most once.
- Before entering the queue, **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

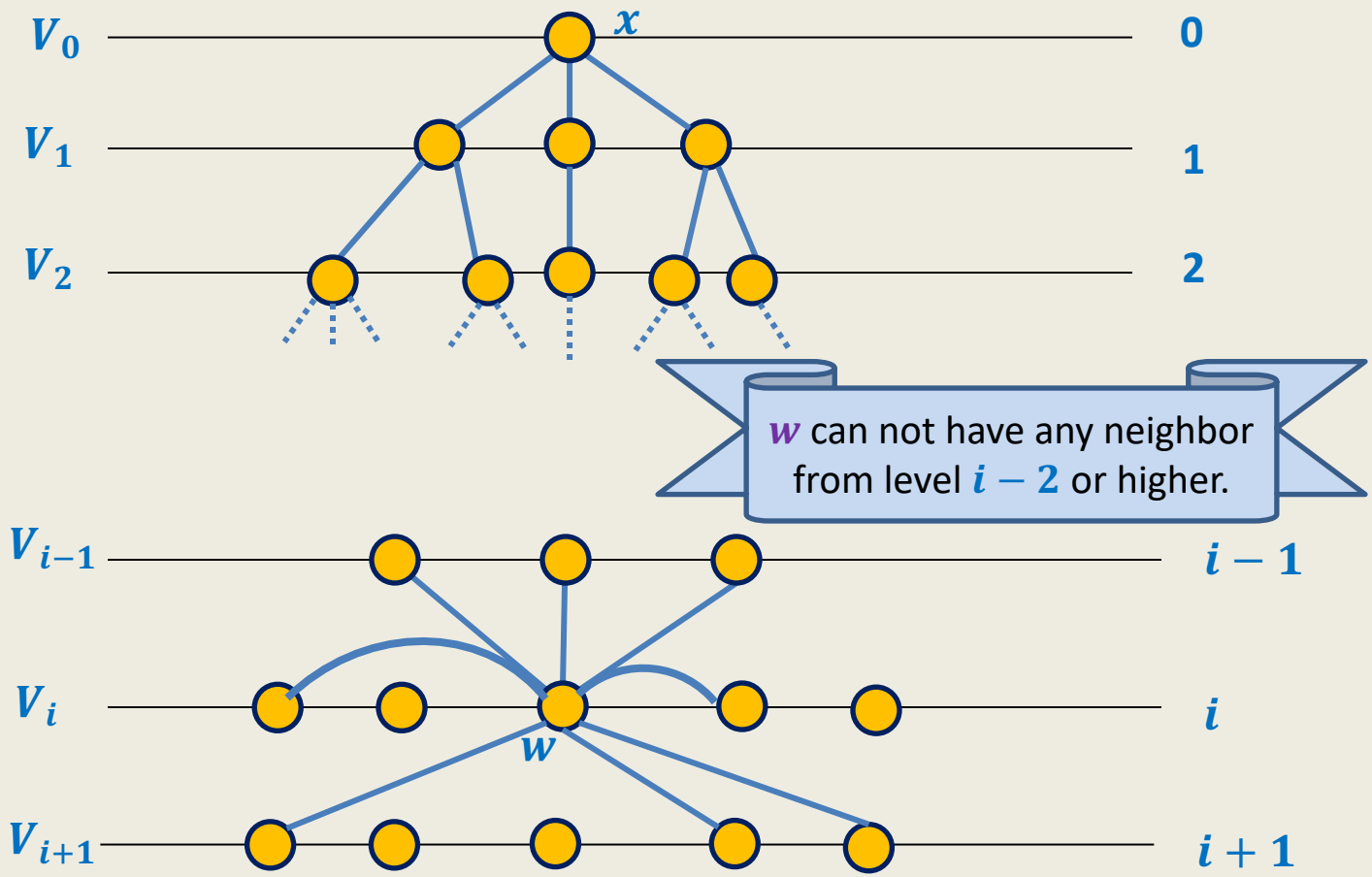
Question: What do we mean by correctness of **BFS**(**G**, x) ?

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, **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 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 **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 **Visited**(x) ← **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.

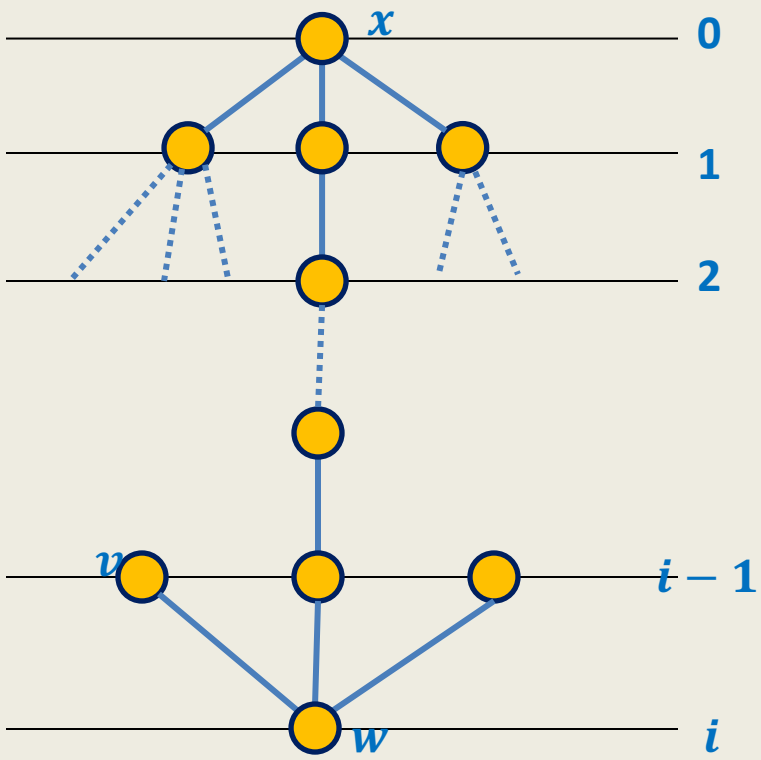
Let $w \in V_i$.

BFS traversal of G from a vertex x

```
BFS(G, x)        //Initially for each v, Distance(v) ← ∞ , and Visited(v) ← false.
{  CreateEmptyQueue(Q);
   Distance(x) ← 0;
   Enqueue(x,Q);    Visited(x) ← true;
   While(Not IsEmptyQueue(Q))
   {   v ←Dequeue(Q);
       For each neighbor w of v
       {
           if (Distance(w) = ∞)
           {   Distance(w) ← Distance(v) +1 ;   Visited(w) ← true;
               Enqueue(w, Q);
           }
       }
   }
```

Induction step:

To prove that **w** ∈ V_i is visited during **BFS**(x)



Let $v \in V_{i-1}$ be any neighbor of w.

By induction hypothesis,

v gets visited during **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.

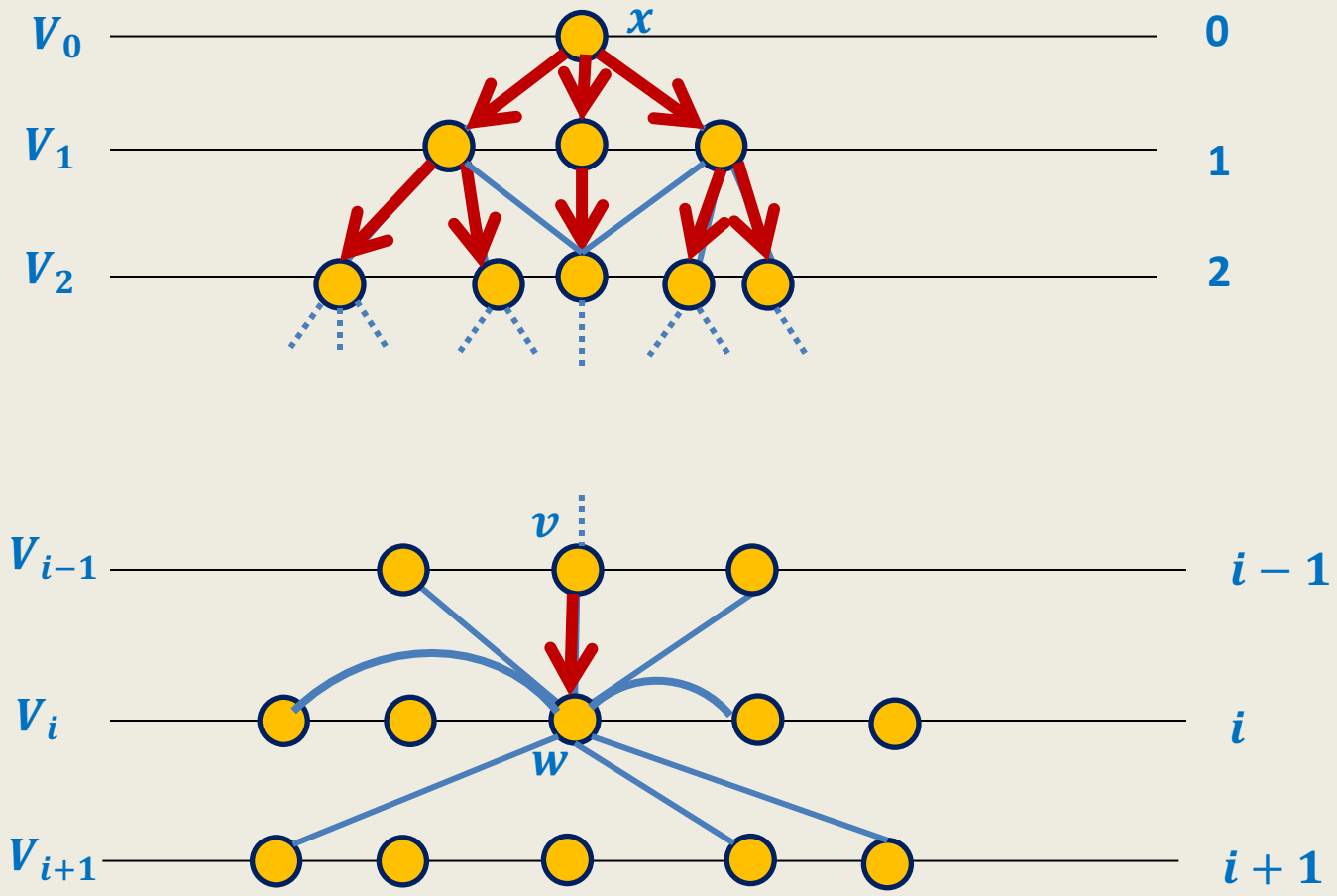
Hence by the principle of mathematical induction, **A**(i) holds for each i.

This completes the proof of **part 1**.

BFS tree

BFS traversal gives a tree

Perform BFS traversal from x.

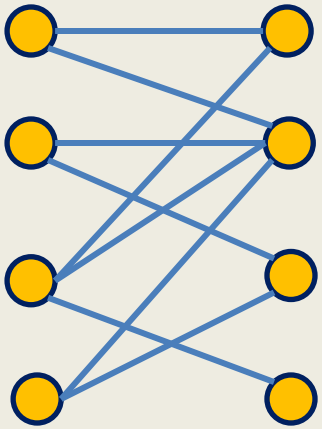


A nontrivial application of BFS traversal

Determining
if a graph is bipartite

Bipartite graph

Definition: A graph **G**=(**V**,**E**) is said to be bipartite
if its vertices can be partitioned into two sets **A** and **B**
such that every edge in **E** has one endpoint in **A** and another in **B**.



A B

YES

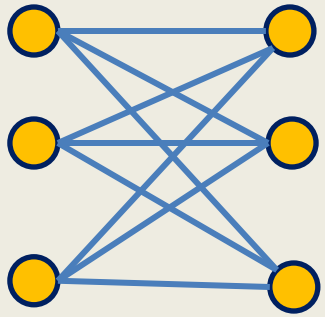
Is this graph bipartite ?

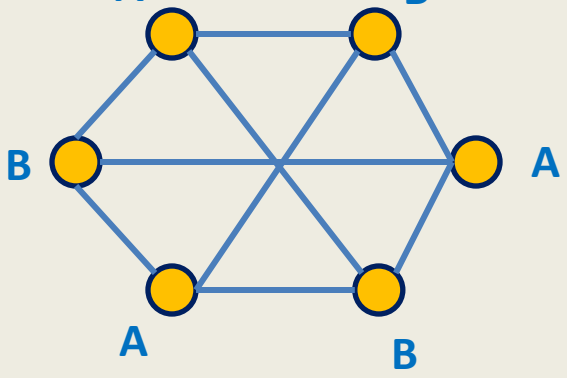
Nontriviality

in determining whether a graph is bipartite

Both are same graph but drawn in different ways. Can you see it ?

Is this graph bipartite ?





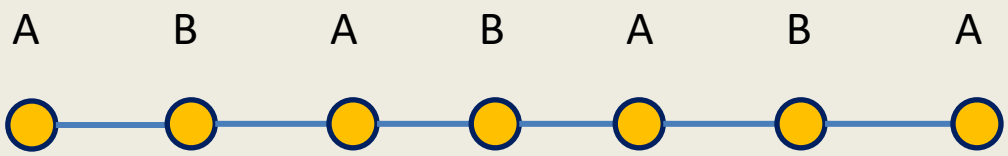
A B

bipartite

bipartite

Bipartite graph

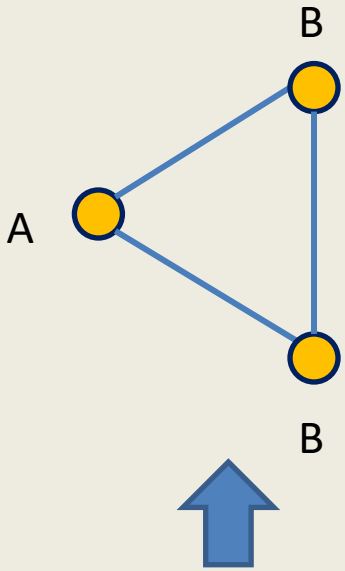
Question: Is a path bipartite ?



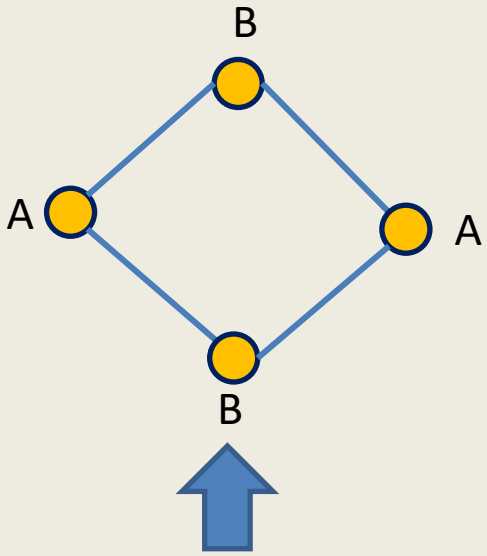
Answer: Yes

Bipartite graph

Question: Is a cycle bipartite ?



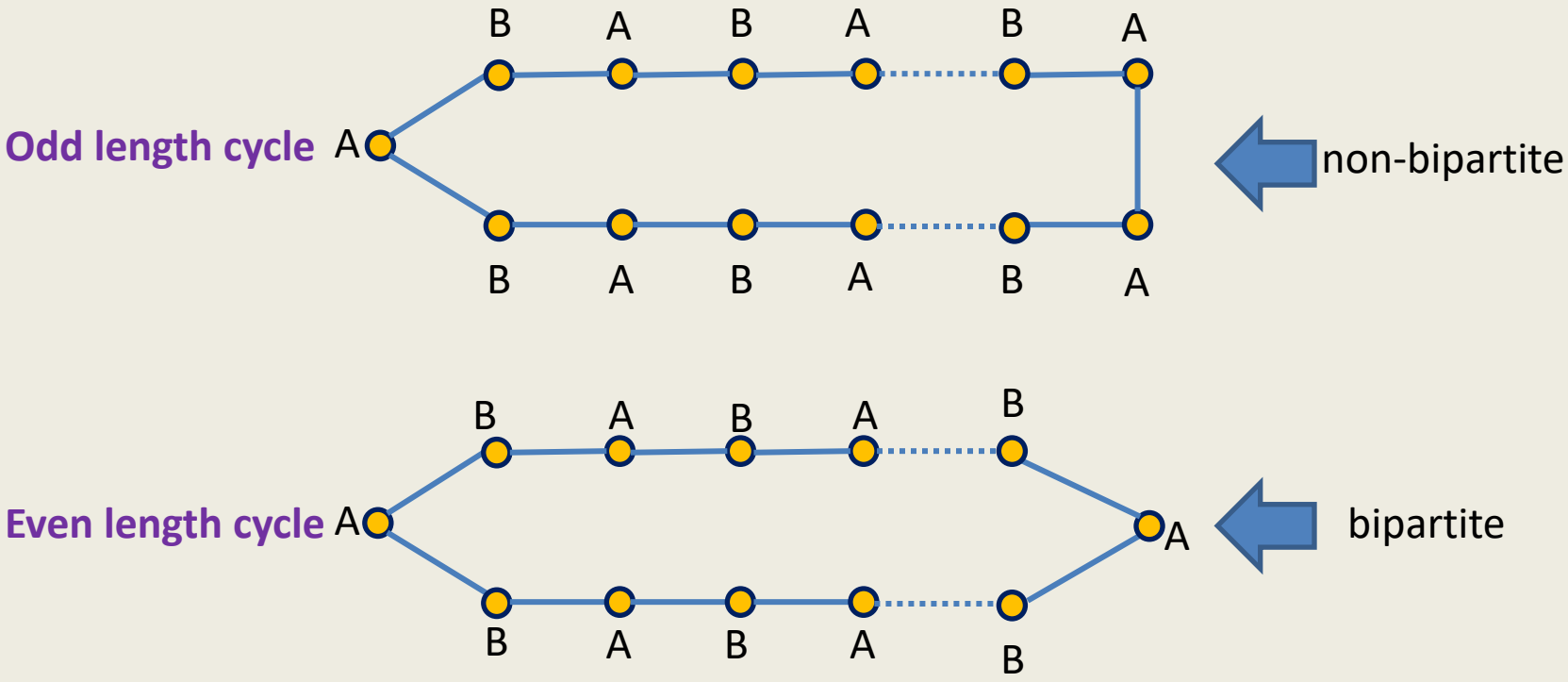
non-bipartite



bipartite

Bipartite graph

Question: Is a cycle bipartite ?



Subgraph

A subgraph of a graph $G=(V,E)$

is a graph $G'=(V',E')$ such that

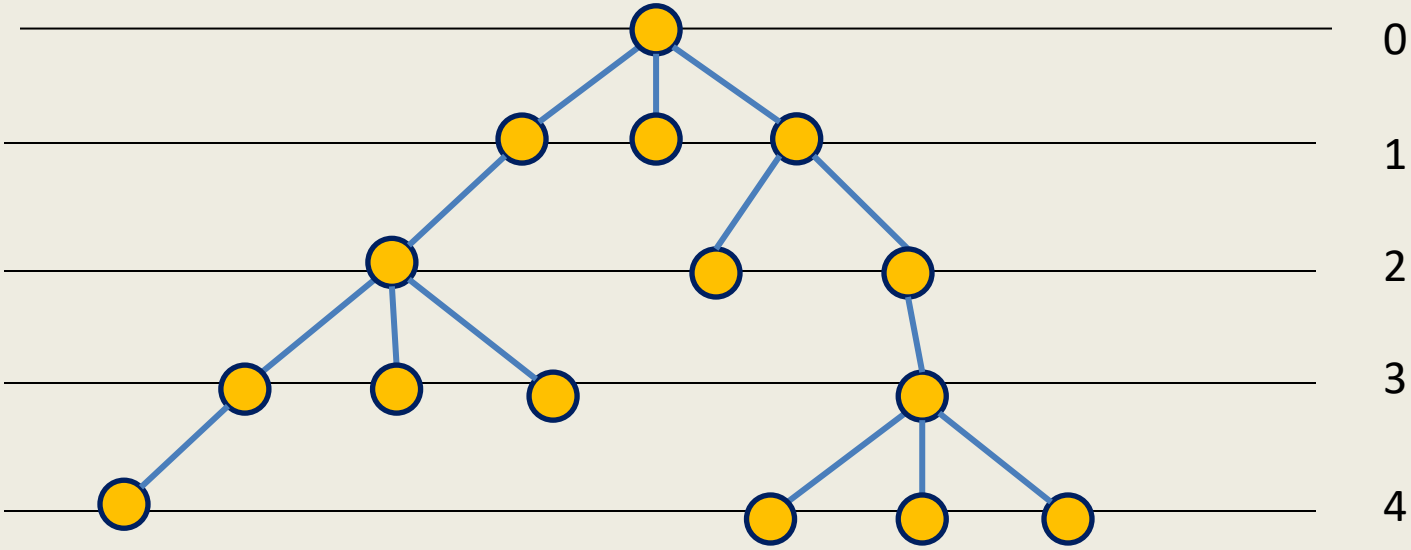
- $V' \subseteq V$
- $E' \subseteq E \cap (V' \times V')$

Question: If G has a subgraph which is **an odd cycle**, is G bipartite ?

Answer: **No**.

Bipartite graph

Question: Is a tree bipartite ?



Answer: Yes

Even level vertices: **A**

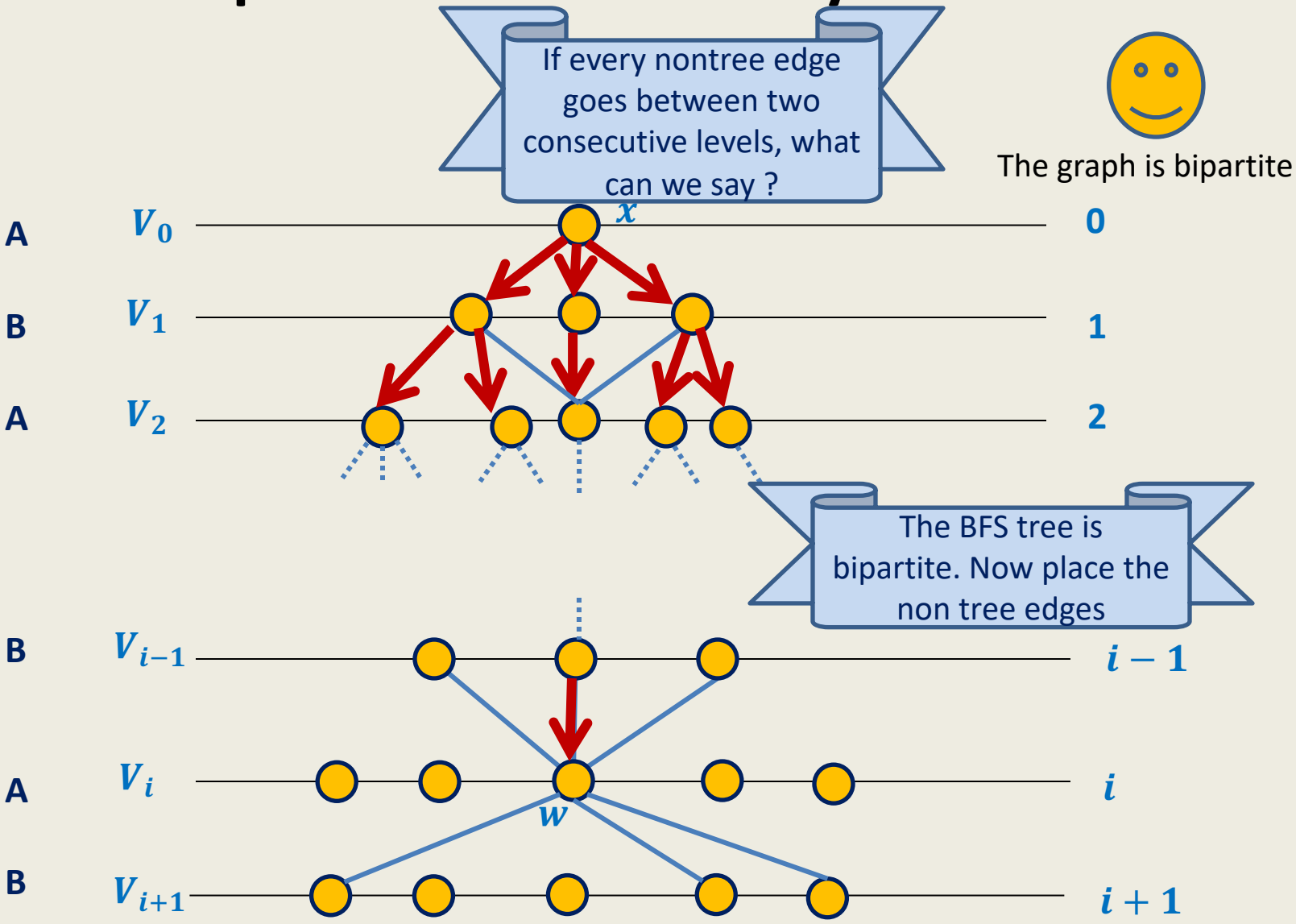
Odd level vertices: **B**

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.



Observation:

If every non-tree edge goes between two consecutive levels of **BFS** tree, then the graph is bipartite.

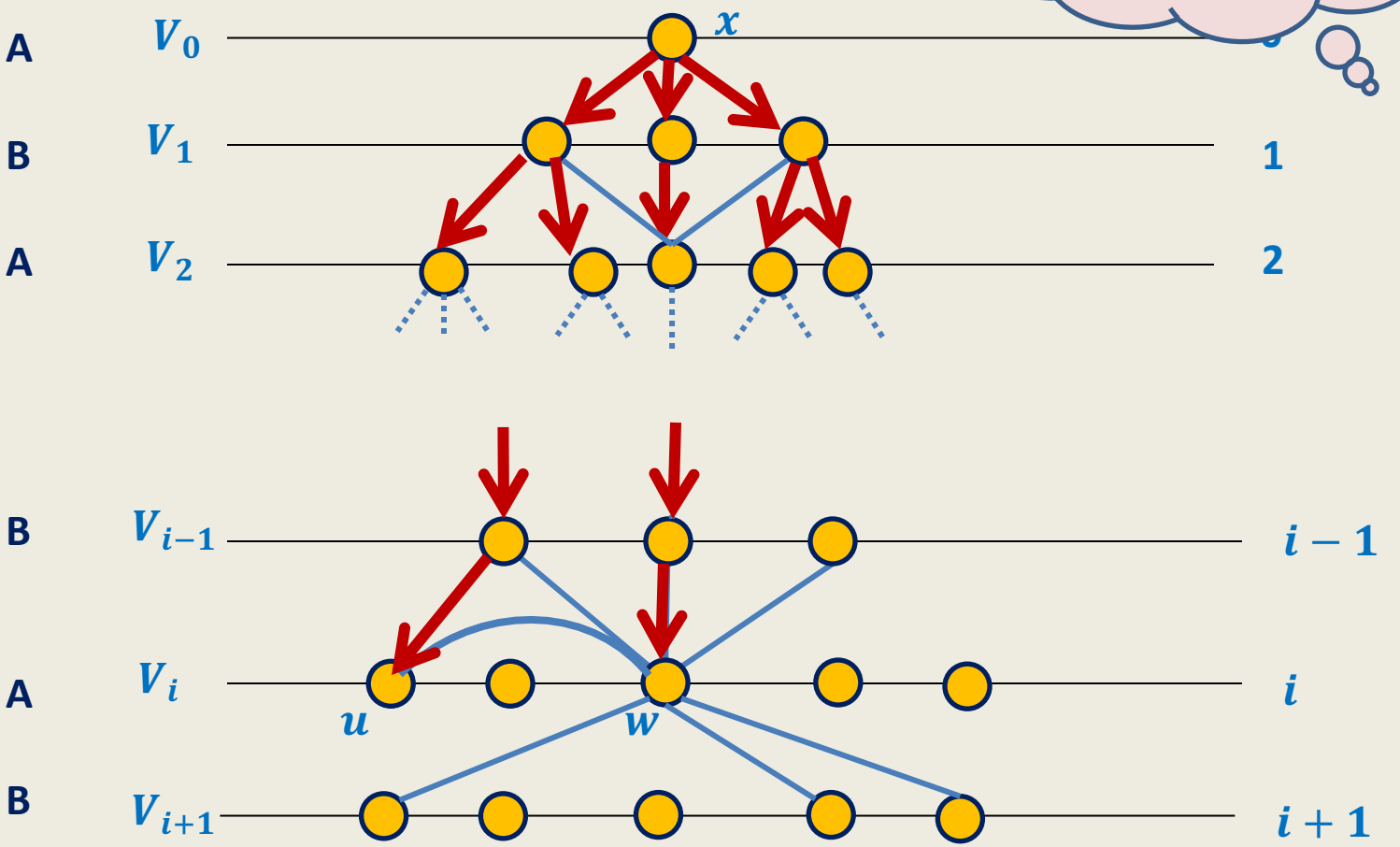
Question:

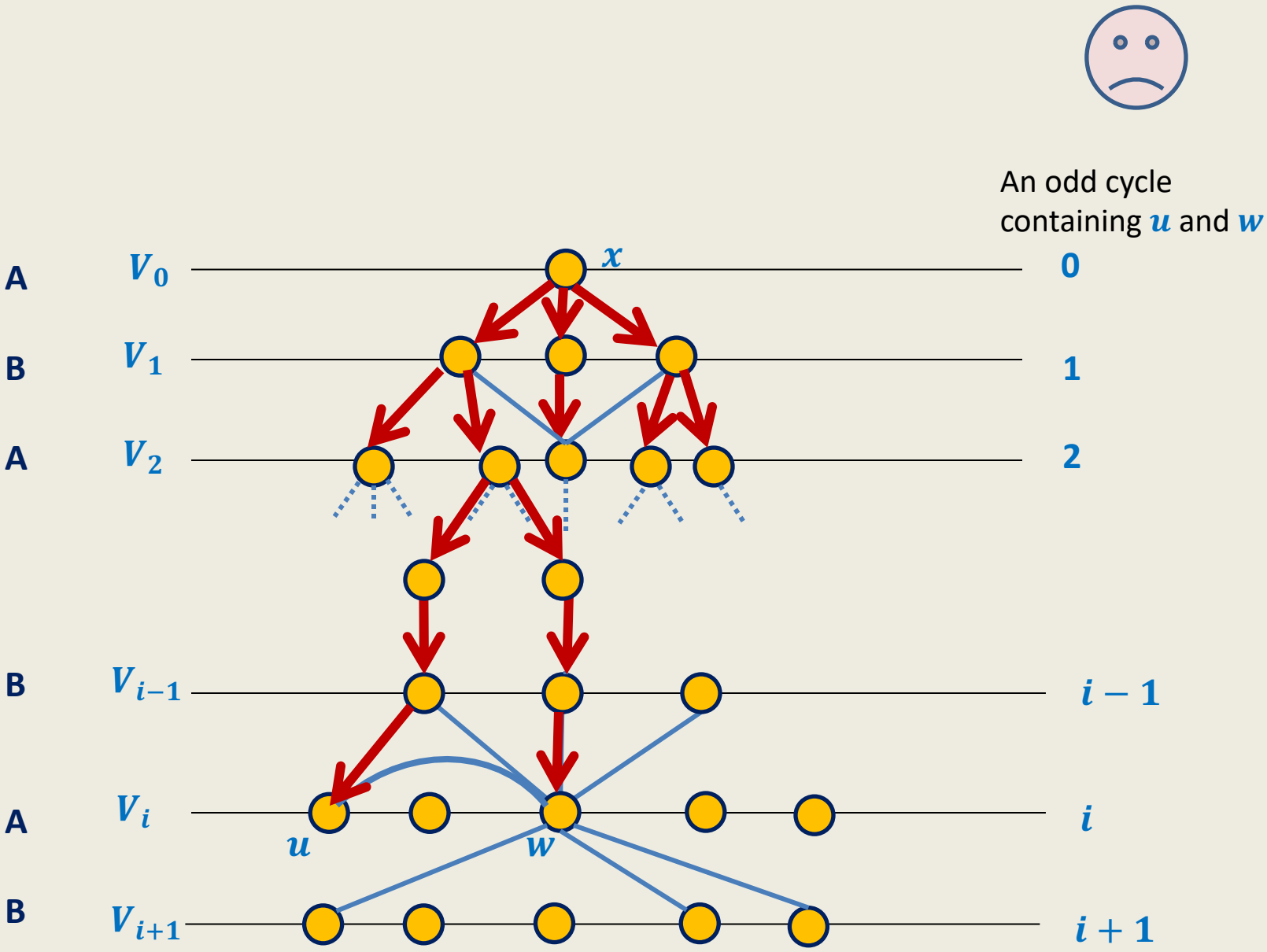
What if there is an edge with both end points at same level ?

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
A
V_0
x
0
B
V_1
1
A
V_2
2
B
V_{i-1}
$i-1$
A
V_i
u
w
i
B
V_{i+1}
$i+1$

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 discus **Depth First Traversal**

: the most nontrivial, elegant graph traversal technique with wide applications.

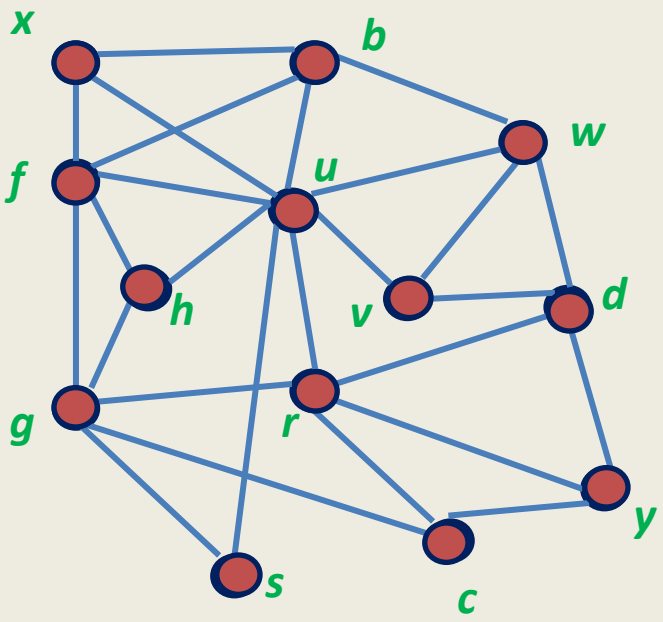
Data Structures and Algorithms

(ESO207)

Lecture 25

- A data structure problem for graphs.
- Depth First Search (DFS) Traversal
- Novel application: computing biconnected components of a graph

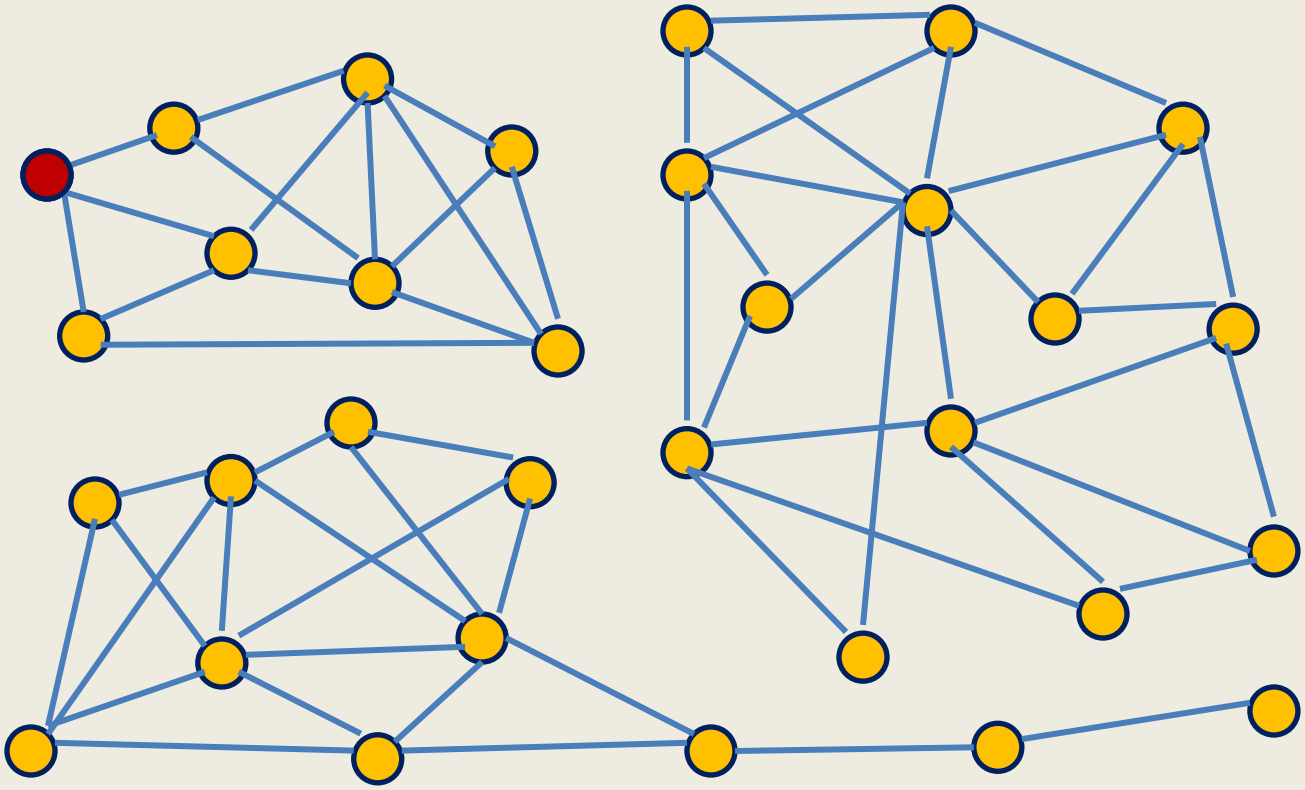
BFS Traversal
in Undirected Graphs



Theorem:

BFS Traversal from ***x*** visits all vertices reachable from ***x*** in the given graph.

Connectivity problem in a Graph

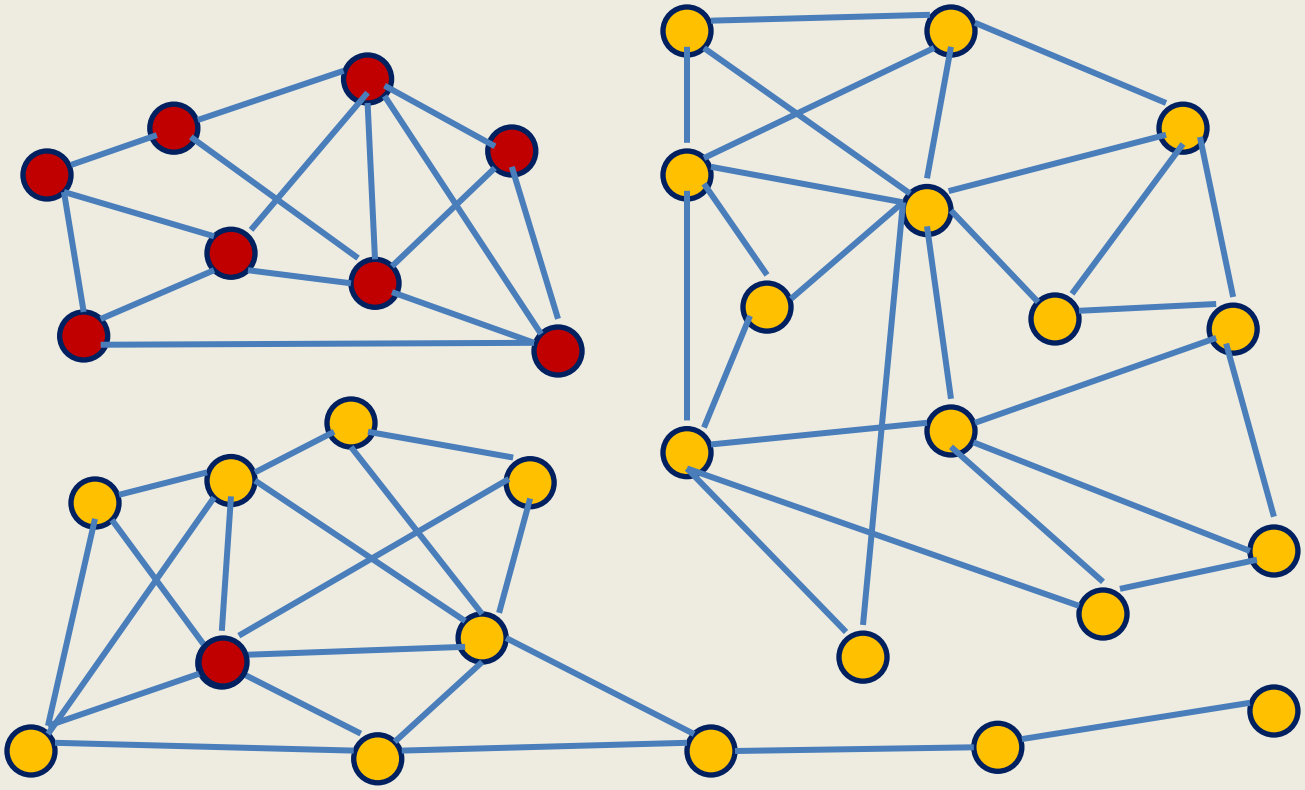


Problem:

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

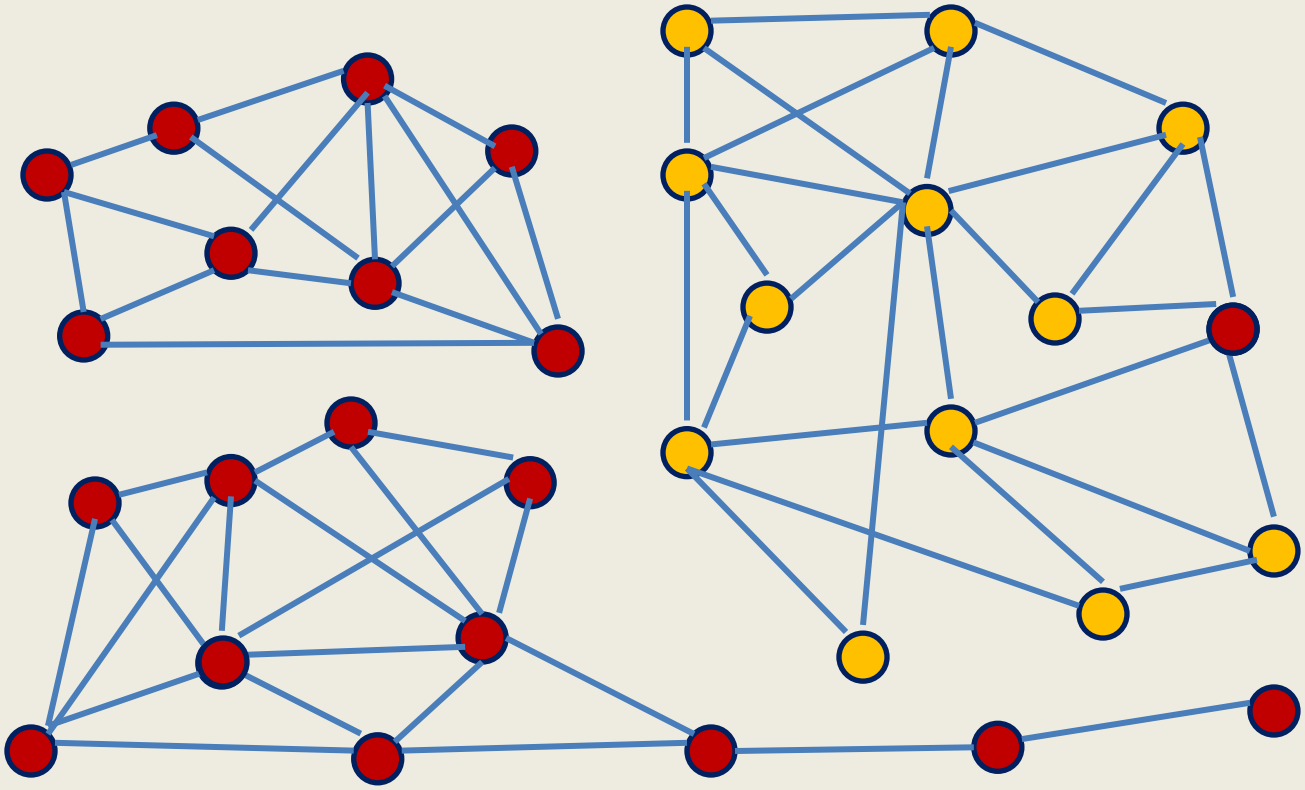


Problem:

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

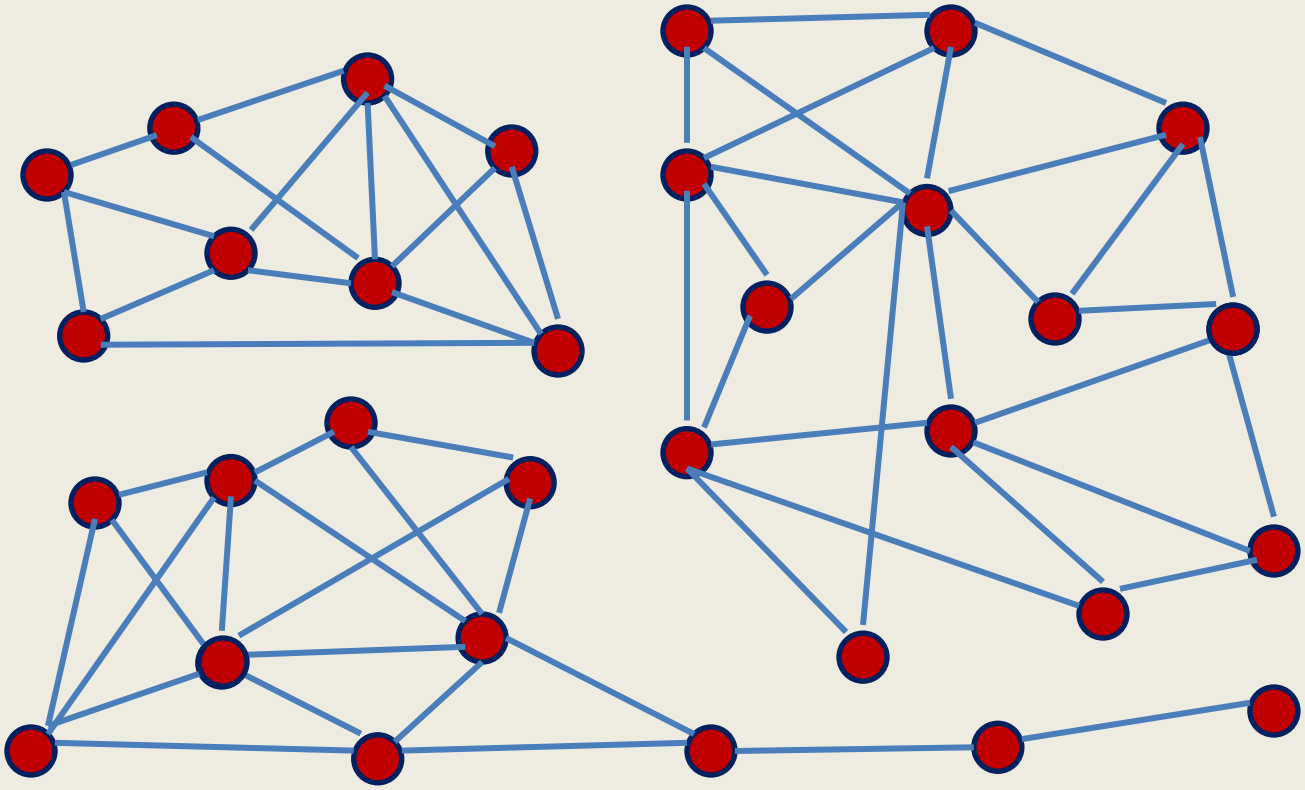


Problem:

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) ← true;   Label[x] ← x;
   Enqueue(x,Q);
   While(Not IsEmptyQueue(Q))
   {          v ←Dequeue(Q);
              For each neighbor w of v
              {       if (Visited(w) = false)
                        {   Visited(w) ←  true ;    Label[w] ← x ;
                            Enqueue(w, Q);
                        }
              }
   }
Connectivity(G)
{  For each vertex x Visited(x)← 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($n + m$)

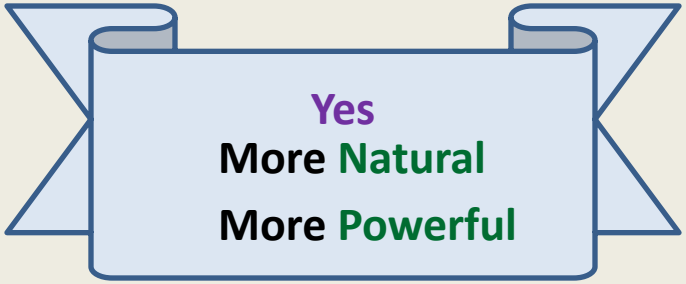
Theorem:

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.

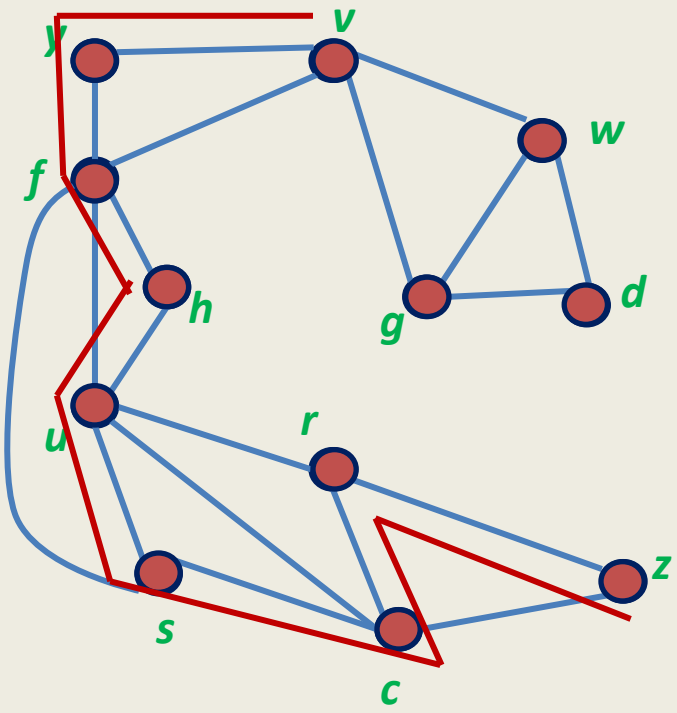
Is there alternate way to traverse a graph ?



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 <u>multiple times</u>

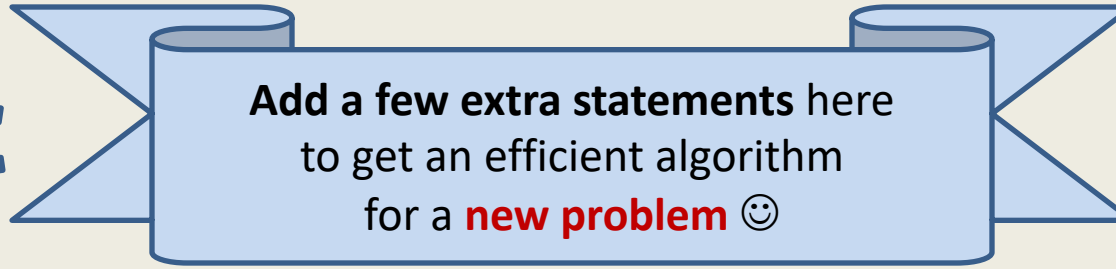
  We can solve it by keeping a label "**Visited**" for each vertex like in BFS traversal.

- **Trace back** in case we reach a dead end.

  Recursion takes care of it ☺

DFS traversal of *G*

```
DFS(v)
{  Visited(v) ← true;
   For each neighbor w of v
   {       if (Visited(w)  = false)
           {   DFS(w) ;
               ……..;
           }
           ……..;
   }
}
```



```
DFS-traversal(G)
{   For each vertex vϵ V  {     Visited(v)← false;      }
    For each vertex v ϵ 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.

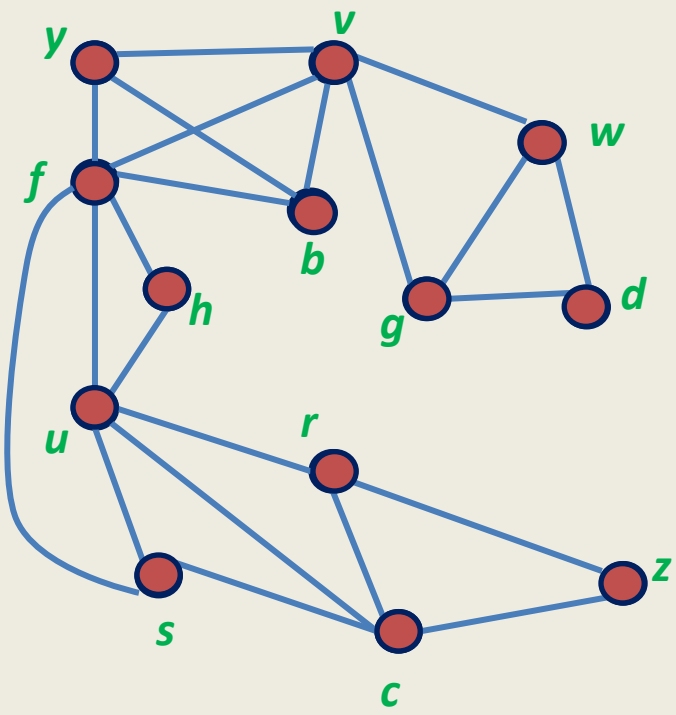
  (Is the connectivity of a graph robust to failure of any node ?)
- Finding **bridges** in a graph.

  (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



DFS(*v*) begins

v visits ***y***

DFS(*y*) begins

y visits ***f***

DFS(*f*) begins

f visits ***b***

DFS(*b*) begins

all neighbors of ***b*** are already visited

DFS(*b*) ends

control returns to **DFS(*f*)**

f visits ***h***

DFS(*h*) begins

…. and so on ….

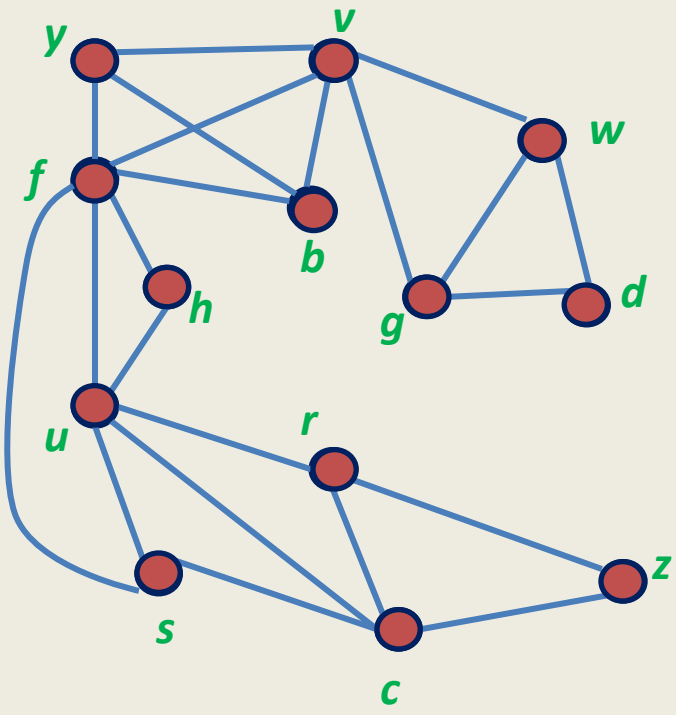
After visiting ***z,*** control returns to ***r→c→s→u→h→f→y→v***

v visits ***w***

DFS(*w*) begins

…. and so on ….

Insight into DFS through an example

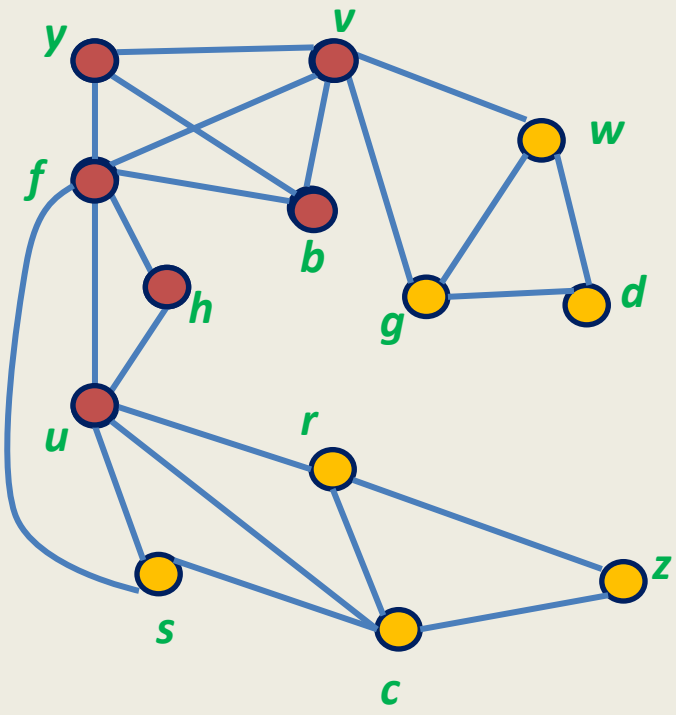


Observation1: (Recursive nature of **DFS**)

If **DFS**(***v***) invokes **DFS**(***w***), then

DFS(***w***) finishes **before** **DFS**(**v**).

Insight into DFS through an example

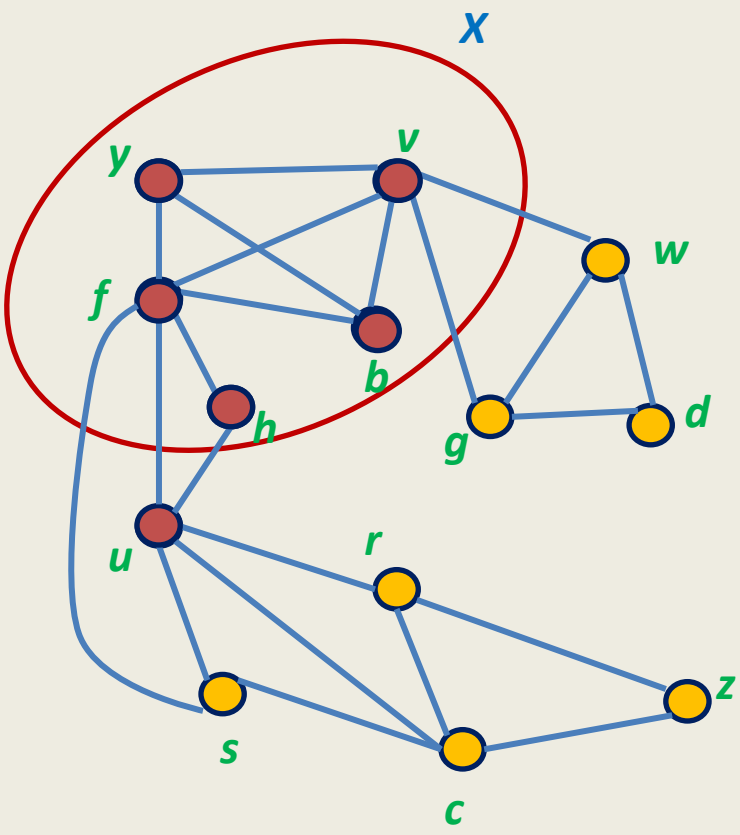


Question :

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*X*.

NOTE:

G*X* is the graph **G** after removal of all vertices ***X*** along with their edges.

Proving that
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.

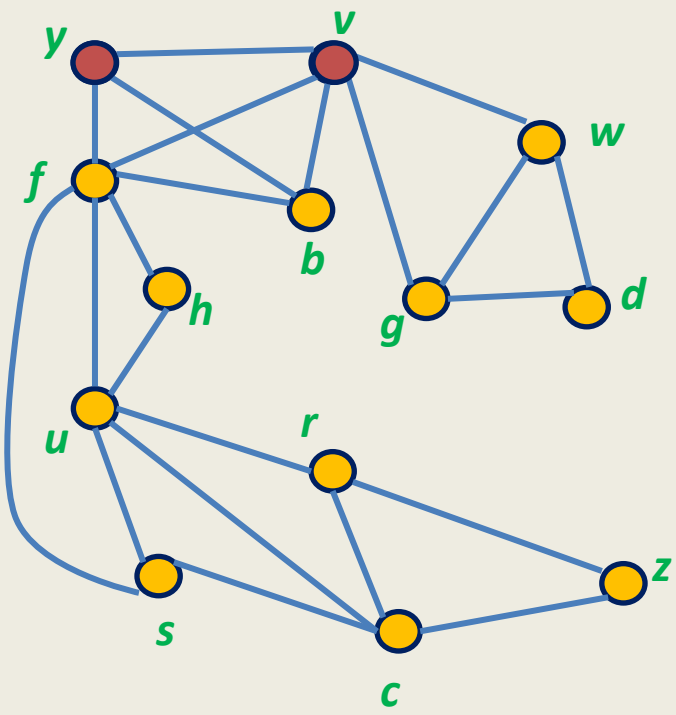
Consider any connected component of size ***i***.

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 $y \notin B$

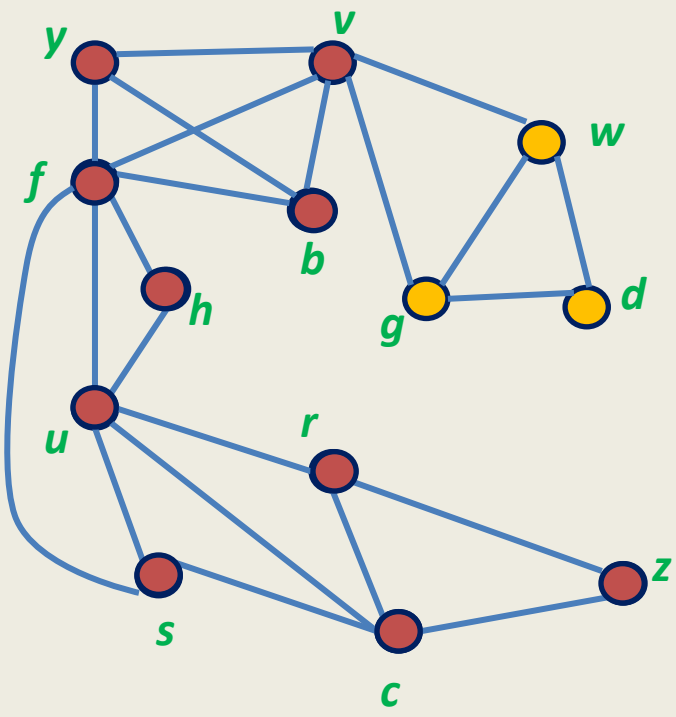
C= {y, b, f, h, u, s, c, r, z} ← $|C| < i$ since $v \notin C$

Question: What is DFS(y) like ?

Answer: 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^*\B.

B= {v, g, w, d}

C= {y, b, f, h, u, s, c, r, z}

|B|< i since y∉ B

|C|< i since v∉ C

Question: What is DFS(y) like ?

Answer: DFS(y) in G\{v}.

Question: What is the connected component of y in G\{v} ?

Answer: C.

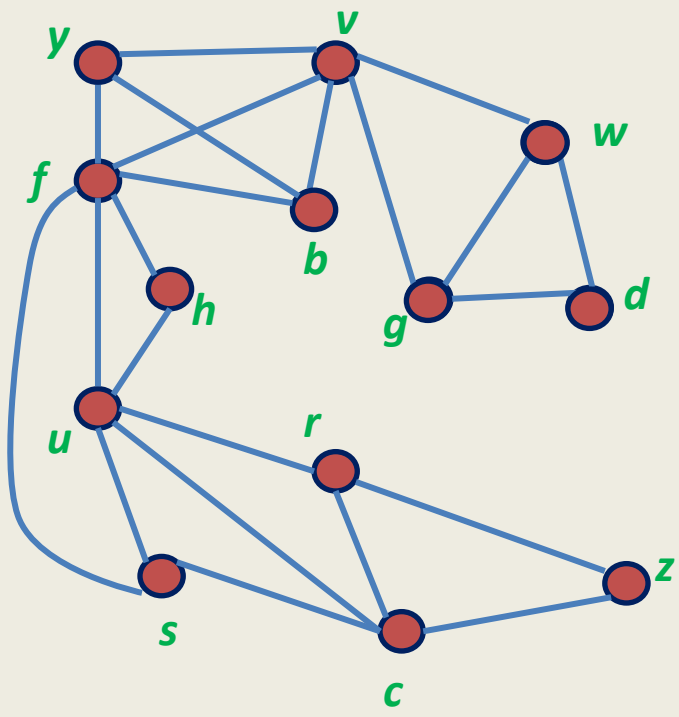
|C|< i, **so by I.H.,** DFS(y) visits entire set C & we return to v.

Question: What is DFS(v) like when DFS(y) finishes ?

Answer: DFS(v) in G\C.

Question: What is the connected component of v in G\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** \B.

B= {*v*, *g*, *w*, *d*} ← |B|< i since y∉ B

C= {*y*, *b*, *f*, *h*, *u*, *s*, *c*, *r*, *z*} ← |C|< i since v∉ C

Question: What is DFS(y) like ?

Answer: DFS(y) in G\{*v*}.

Question: What is the connected component of *y* in G\{v} ?

Answer: C.

|C|< i, **so by I.H.,** DFS(*y*) visits entire set C & we return to v.

Question: What is DFS(*v*) like when DFS(*y*) finishes ?

Answer: DFS(v) in G\C.

Question: What is the connected component of *v* in G\C ?

Answer: B.

|B|< i, **so by I.H.,** DFS(*v*) pursued after finishing DFS(*y*) visits entire set B.

Theorem: 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

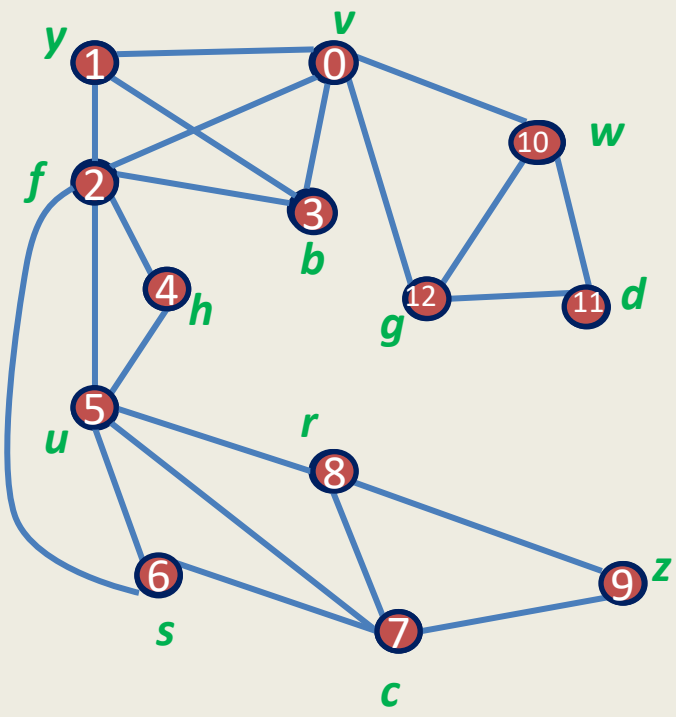
- Depth First Search (DFS) Traversal
- DFS Tree
- Novel application: computing biconnected components of a graph

DFS traversal of *G*

```
DFS(v)
{  Visited(v) ← true;  DFN[v] ← dfn ++;
   For each neighbor w of v
   {        if (Visited(w)  = false)
            {   DFS(w) ;
                ……..;
            }
            ……..;
   }
}
```

```
DFS-traversal(G)
{  dfn ← 0;
   For each vertex vϵ V  {     Visited(v)←  false                }
   For each vertex v ϵ V {     If (Visited(v ) = false)   DFS(v)  }
}
```

DFN number

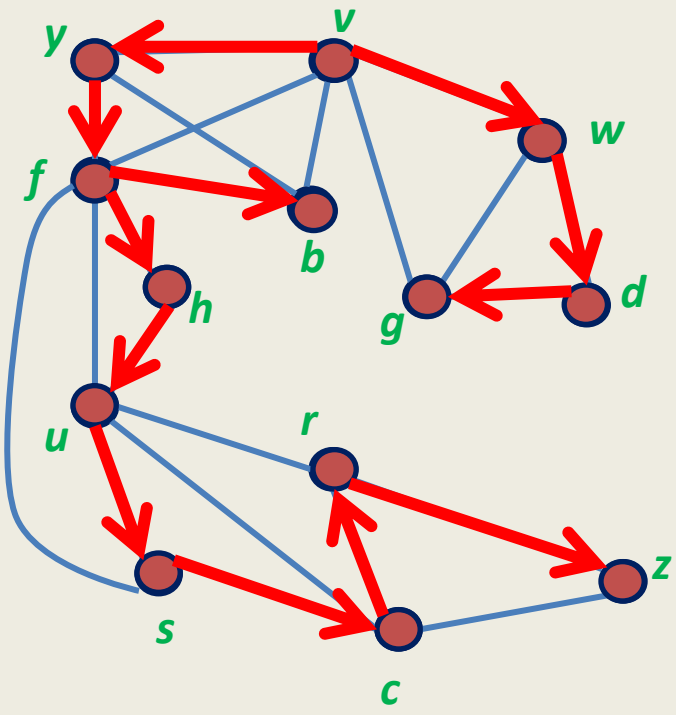


DFN[*x*] :

The number at which *x* gets visited during DFS traversal.

DFS tree

DFS(v) computes a tree rooted at v

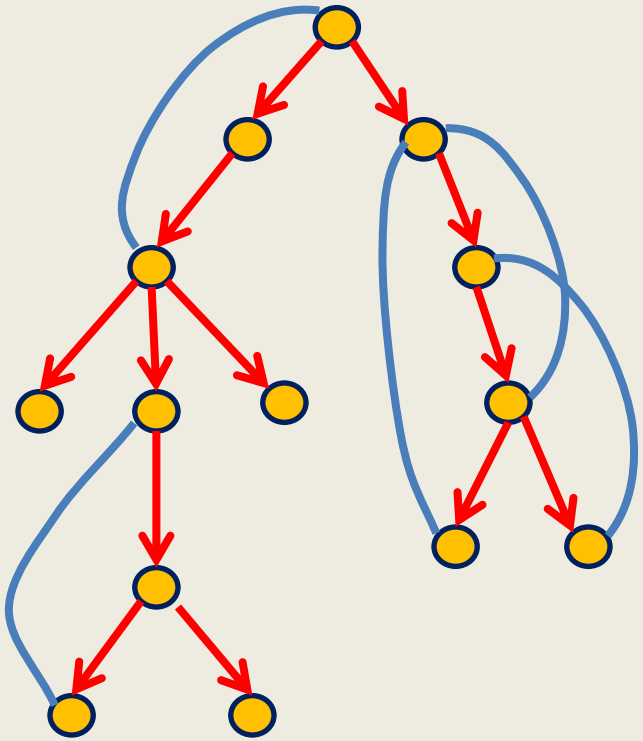


A **DFS** tree rooted at ***v***

Can any rooted tree be obtained through DFS ?

No

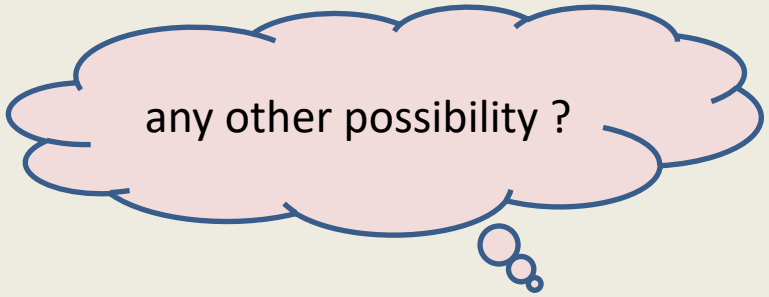
How will an edge appear in DFS traversal ?



- as a **tree-edge**.

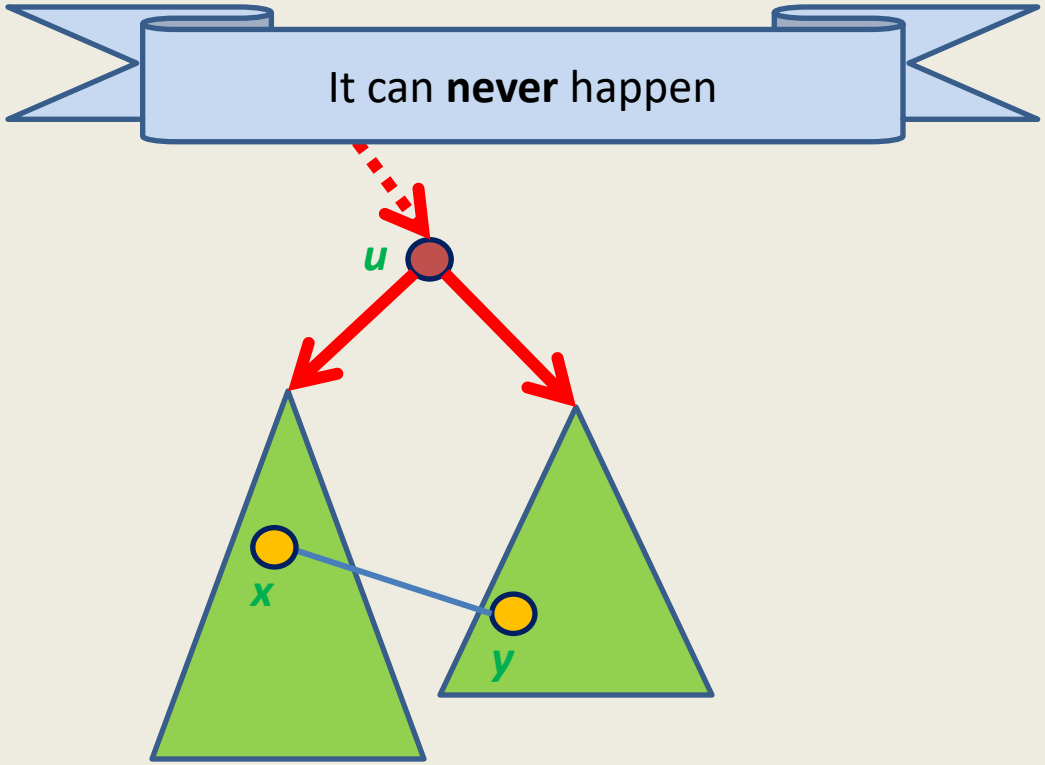
If the edge is a **non-tree** edge :

- Edge between **ancestor** and **descendant** in **DFS** tree.



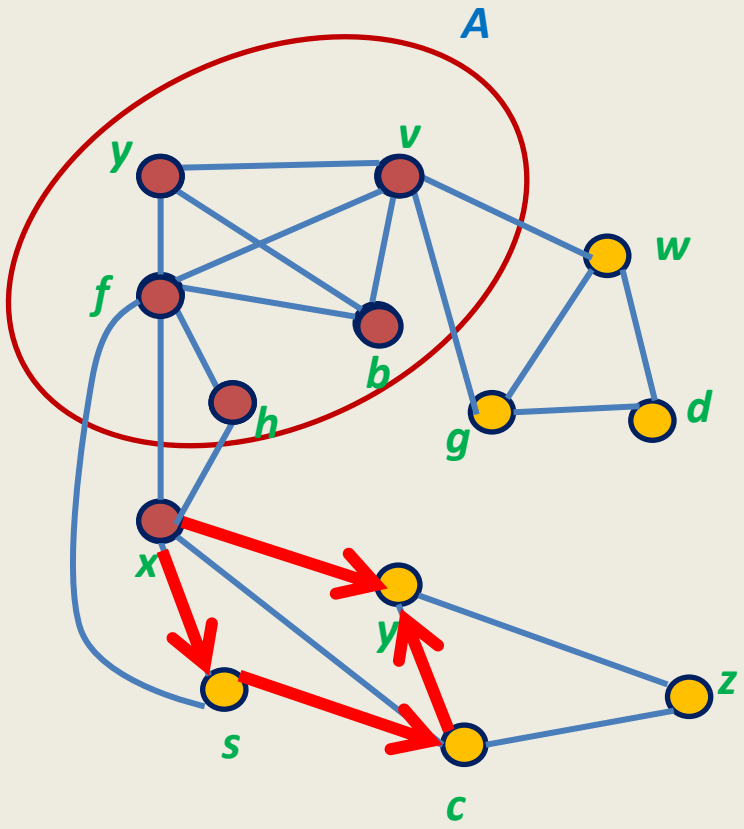
No

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.

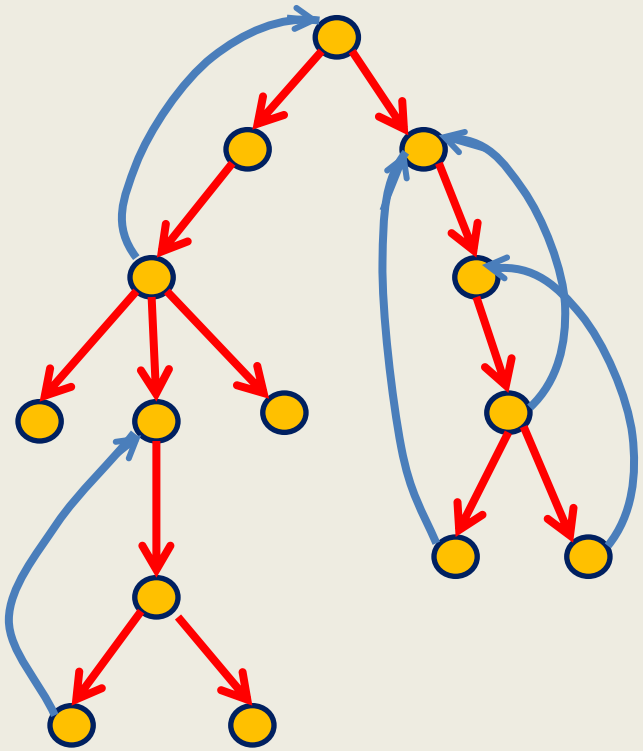
➔

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**.

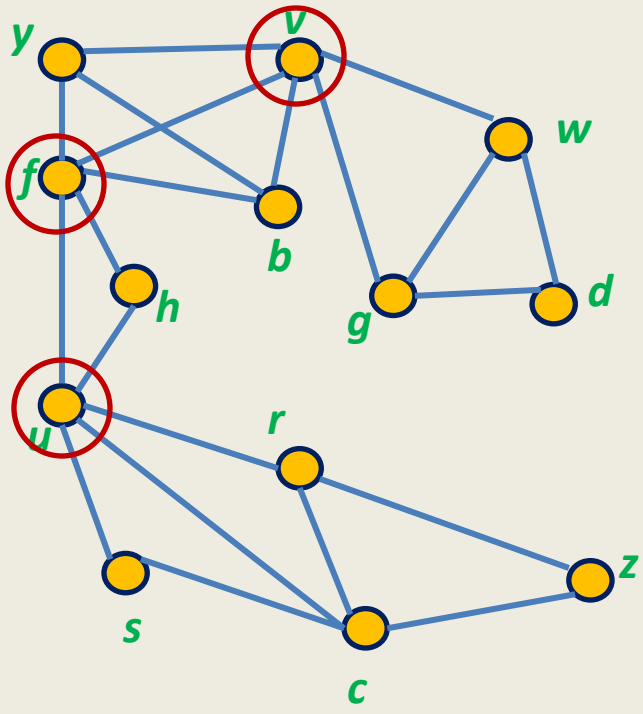
It plays a key role in the design of every efficient algorithm.

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\{*v*}** is connected

  (One may use either **BFS** or **DFS** traversal here)

Time complexity of the trivial algorithm : O(mn)

An **O**($\boldsymbol{m + n}$) time algorithm

A single **DFS** traversal

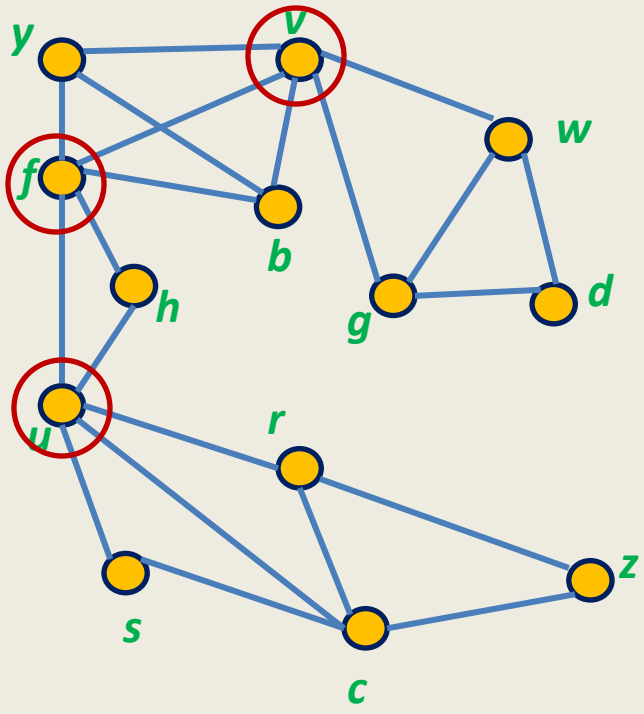
An $O(m + n)$ time algorithm

- 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.

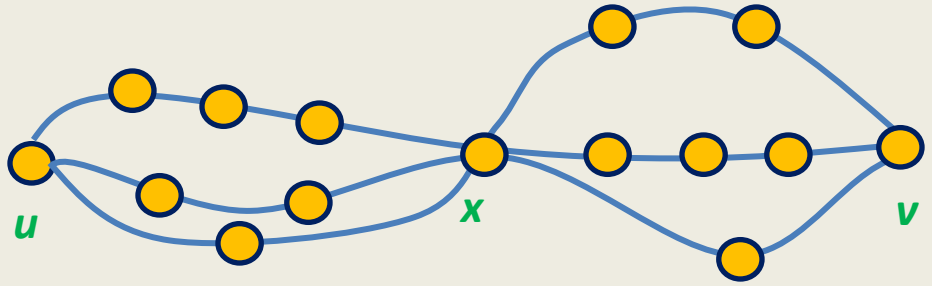
v,*f*,*u* are called the **articulation points** of *G*.

A formal definition of articulaton point

Definition: A vertex ***x*** is said to be **articulation point** if

∃ ***u***,***v*** different from ***x***

such that every path between ***u*** and **v** passes through ***x***.

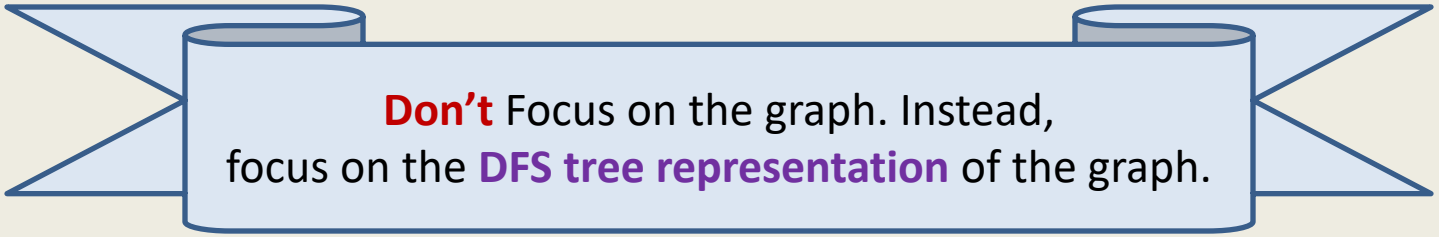


Observation: A graph is biconnected if none of its vertices is an articulation point.

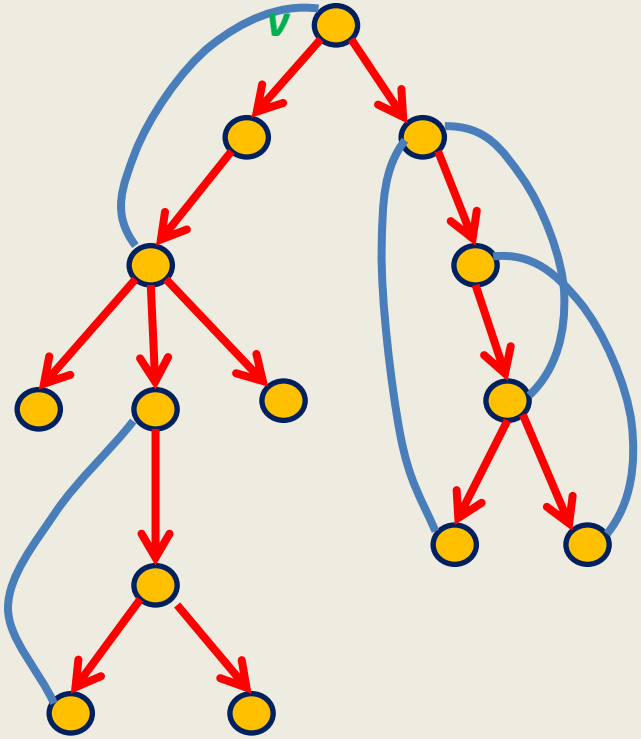
AIM:

Design an **algorithm** to compute all **articulation points** in a given graph.

Articulation points and DFS traversal



Some observations

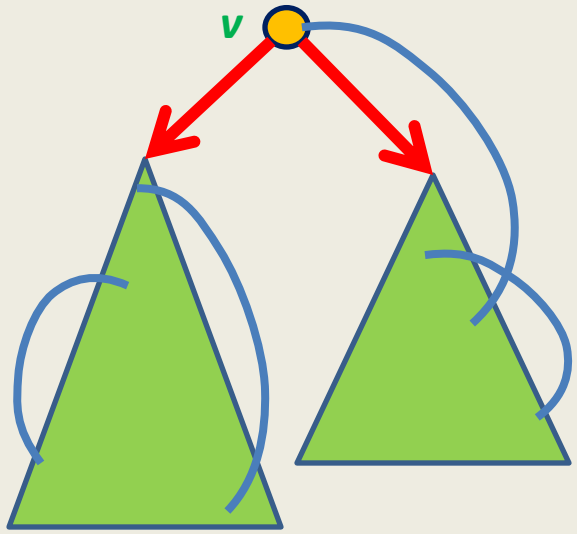


Question: When can **a leaf node** be an **a.p.** ?

Answer: Never

Question: When can **root** be an **a.p**. ?

Some observations

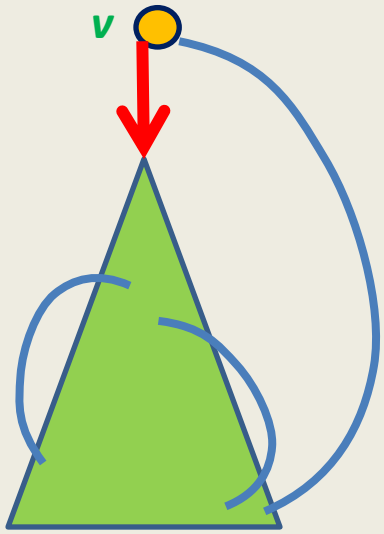


Question: When can **a leaf node** be an **a.p.** ?

Answer: Never

Question: When can **root** be an **a.p**. ?

Some observations



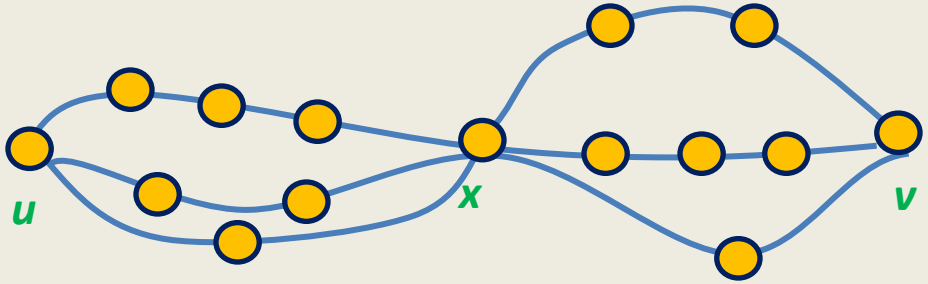
Question: When can **a leaf node** be an **a.p.** ?

Answer: Never

Question: When can **root** be an **a.p**. ?

Answer: Iff it has **two or more** children.

Some observations

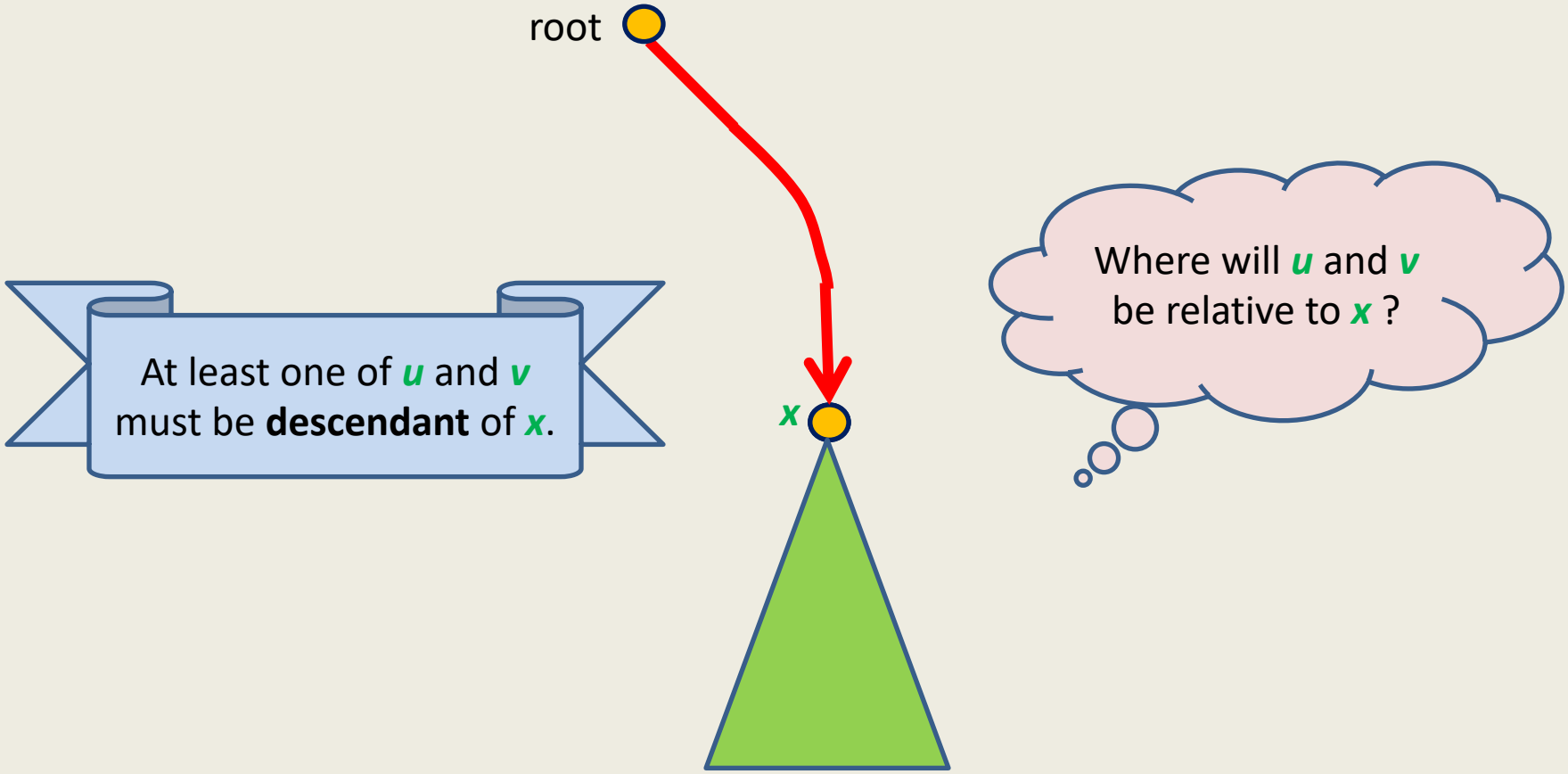


AIM:

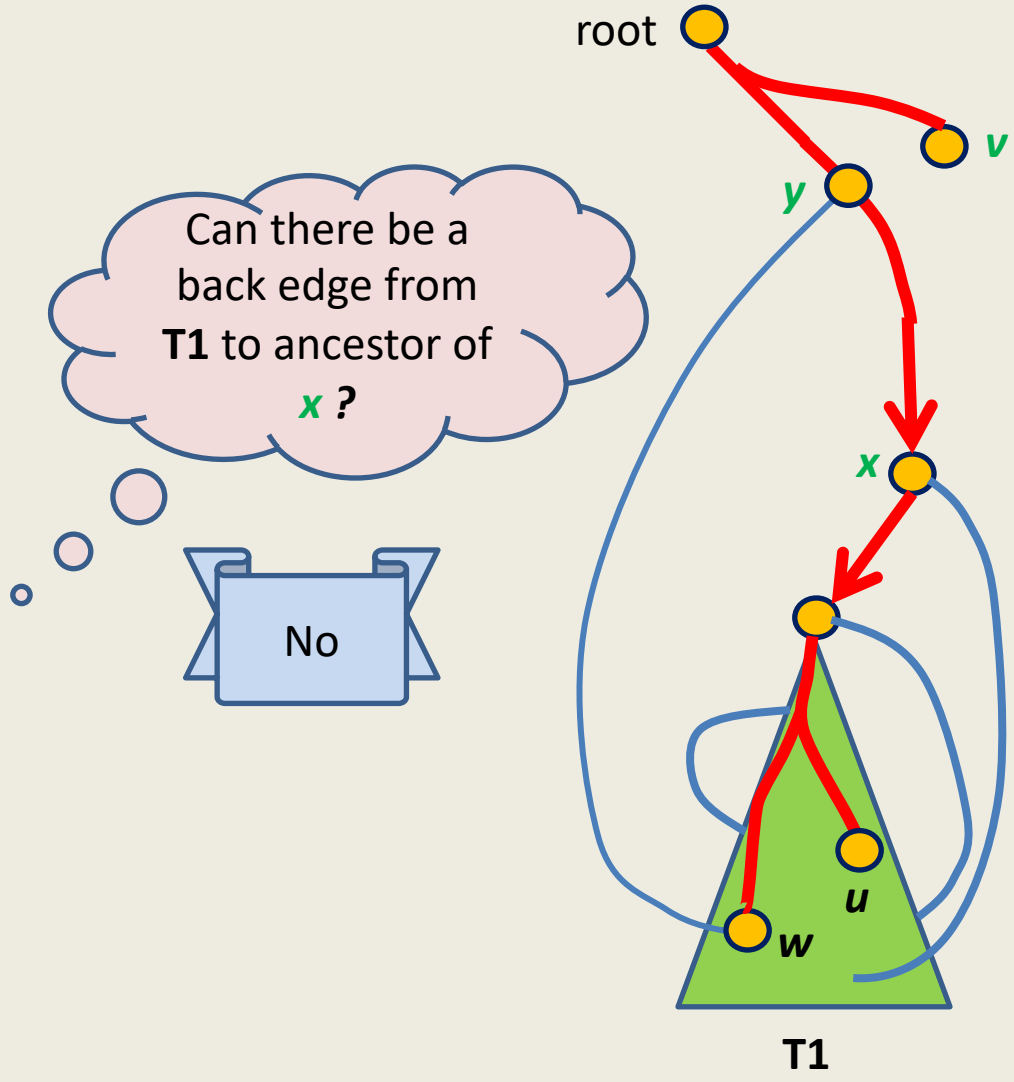
To find **necessary** and **sufficient conditions** for an **internal node** to be **articulation point**.

How will *x* look like
in **DFS tree** ?

conditions for an **internal node** to be **articulation point**.

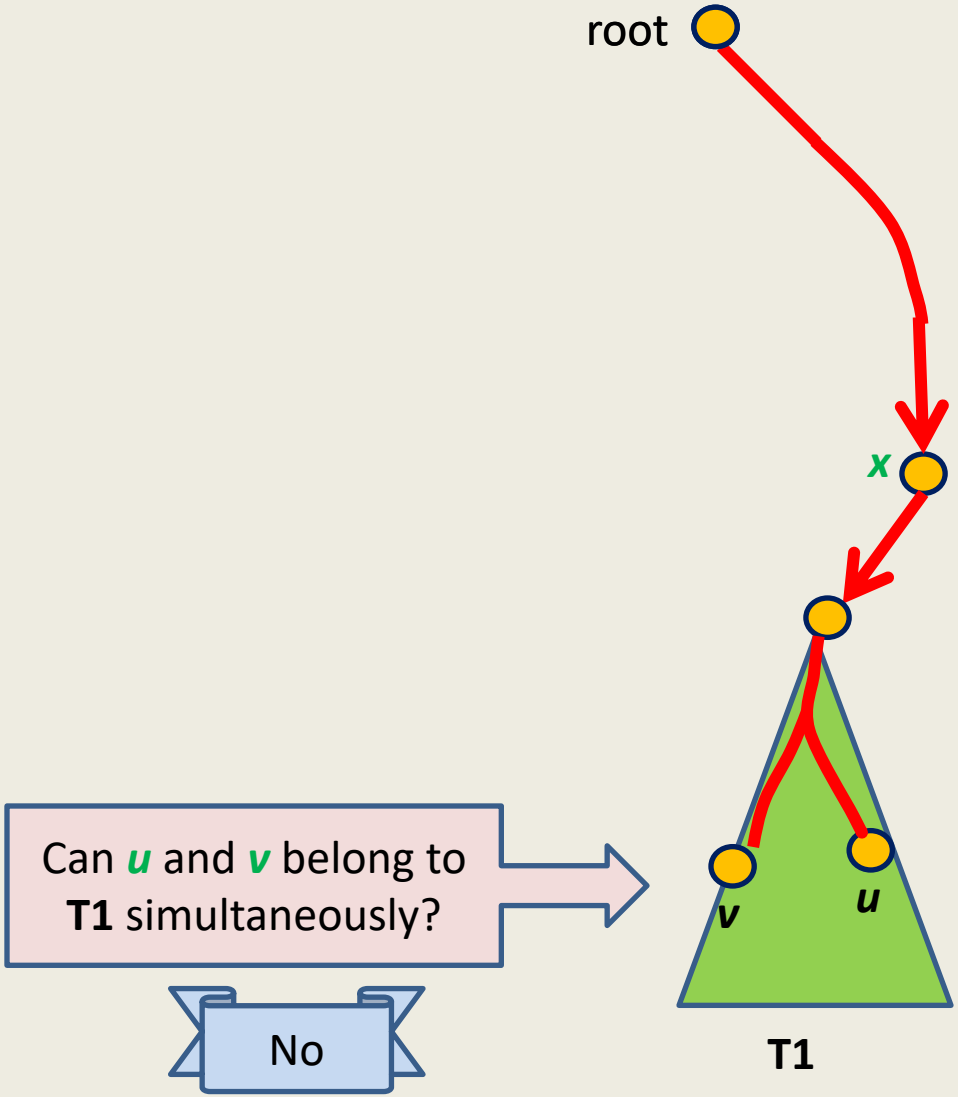


Case 1: Exactly one of u and v is a descendant of x in DFS tree

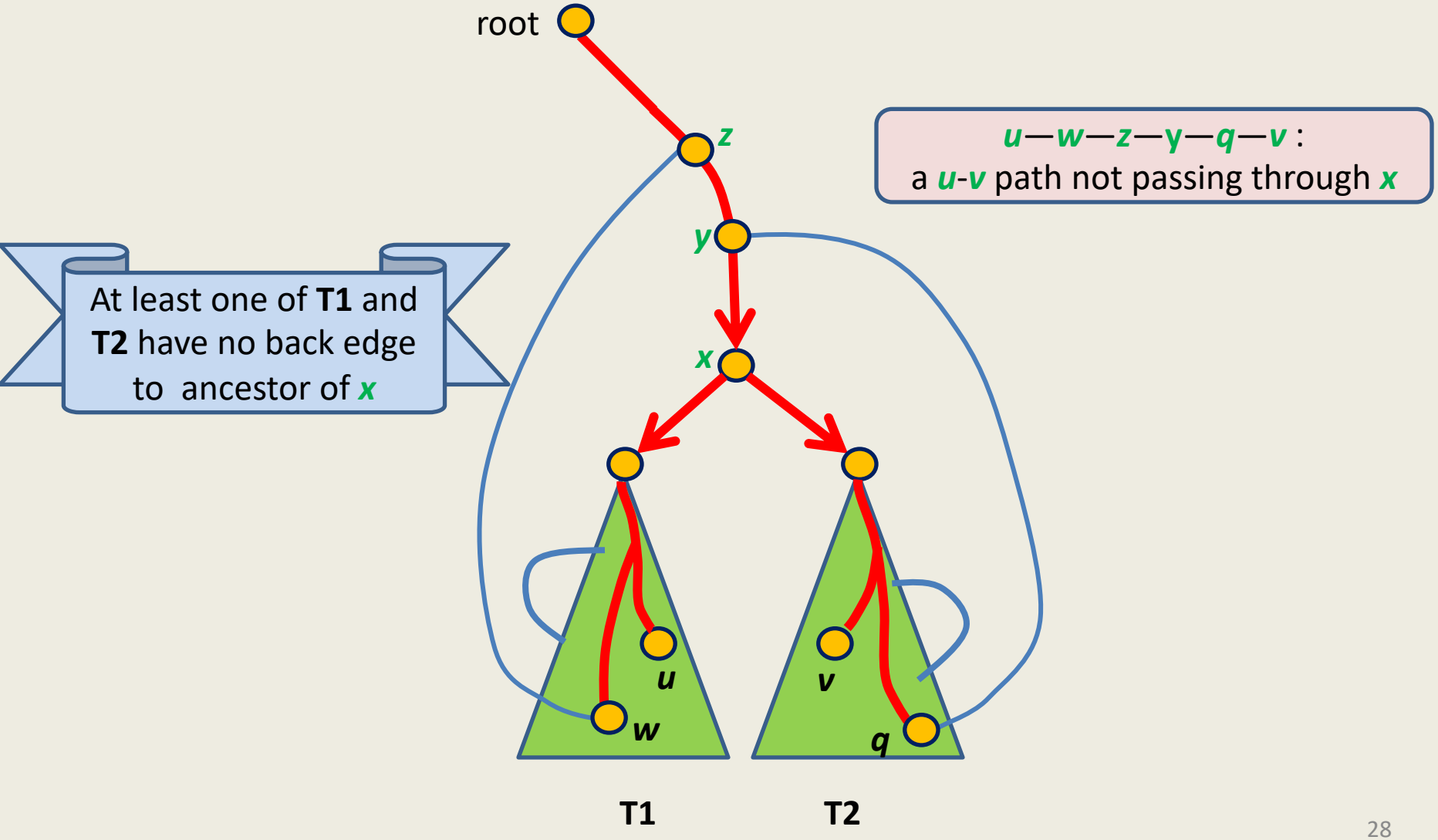


u—*w*—*y*—*v* :
a *u*-*v* path not passing through *x*

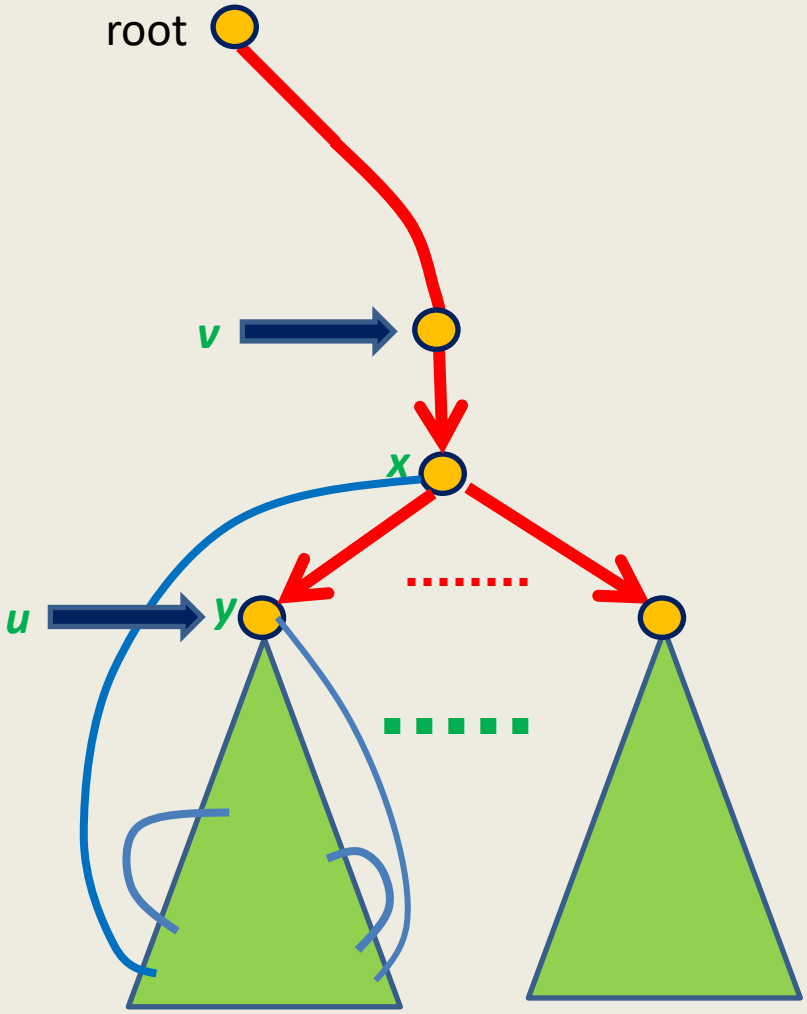
Case 2: both u and v are descendants of x in DFS tree



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*.

Is this condition **sufficient** also ?

yes.

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

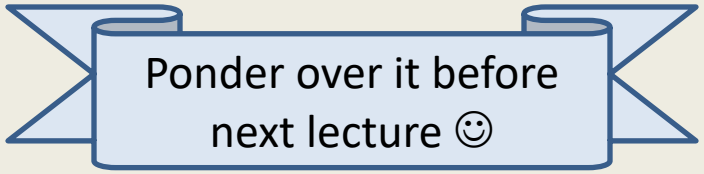
there is **no** back edge

from **subtree**(y) to any ancestor of x.

Efficient algorithm for Articulation points

Use Theorem 1

Exploit recursive nature of DFS



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

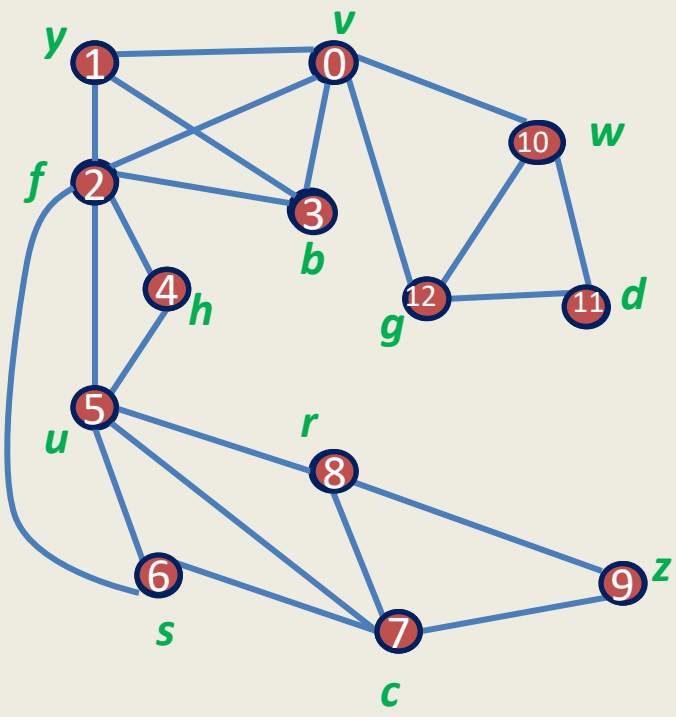
Quick revision of
Depth First Search (DFS) Traversal

DFS traversal of *G*

```
DFS(v)
{  Visited(v) ← true;  DFN[v] ← dfn ++;
   For each neighbor w of v
   {        if (Visited(w)  = false)
            {   DFS(w) ;
                ……..;
            }
            ……..;
   }
}
```

```
DFS-traversal(G)
{  dfn ← 0;
   For each vertex vϵ V  {     Visited(v)←  false                  }
   For each vertex v ϵ V {     If (Visited(v ) = false)   DFS(v)   }
}
```

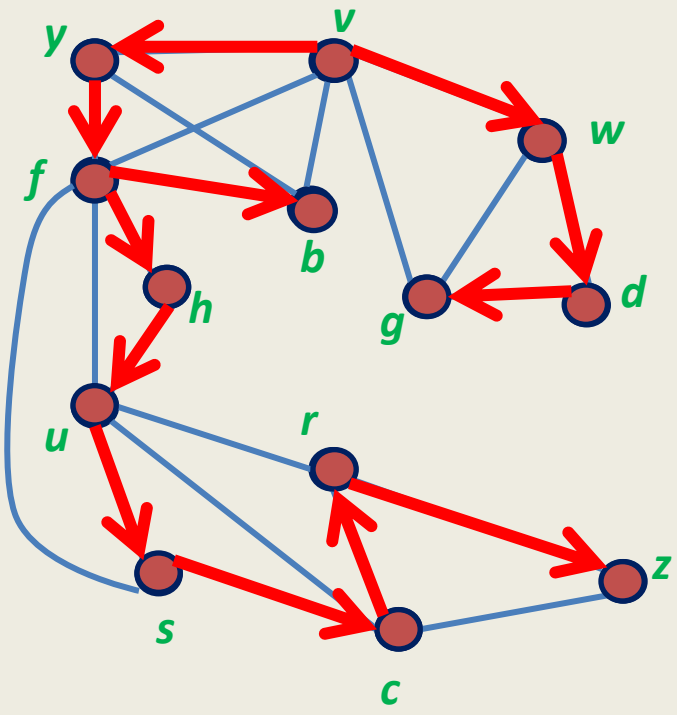
DFN number



DFN[*x*] :

The number at which *x* gets visited during DFS traversal.

DFS(v) computes a tree rooted at v



A **DFS** tree rooted at ***v***

If **x** is ancestor of **y** then

DFN[***x***] < **DFN**[***y***]

Question: Is a **DFS** tree unique ?

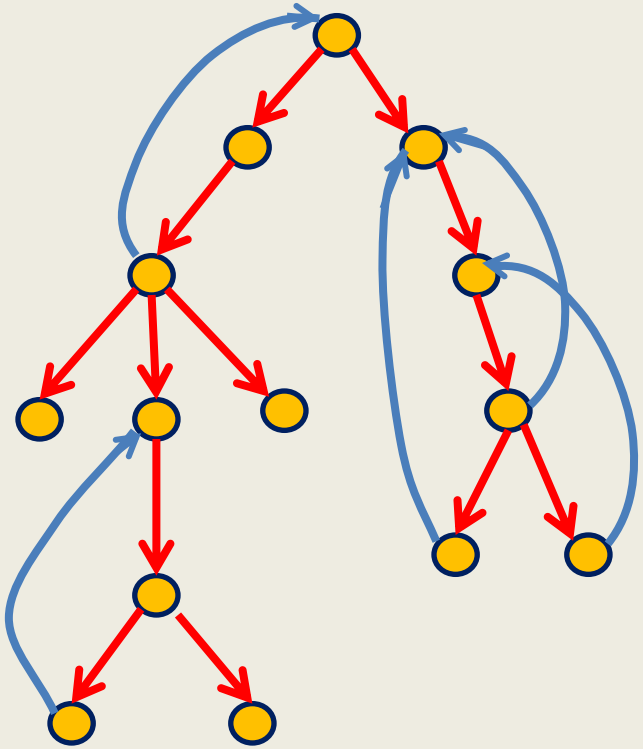
Answer: No.

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

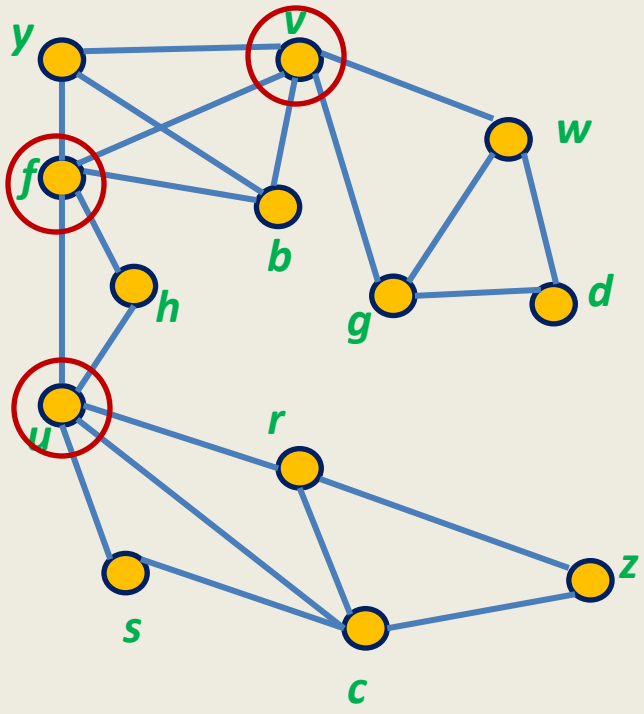
An $O(m + n)$ time algorithm

- 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.

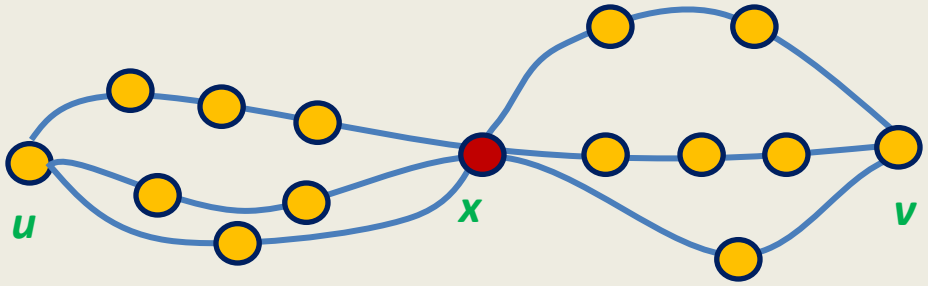
v,***f***,***u*** are called the **articulation points** of ***G***.

A formal definition of articulaton point

Definition: A vertex ***x*** is said to be **articulation point** if

∃ ***u***,***v*** different from ***x***

such that every path between ***u*** and **v** passes through ***x***.

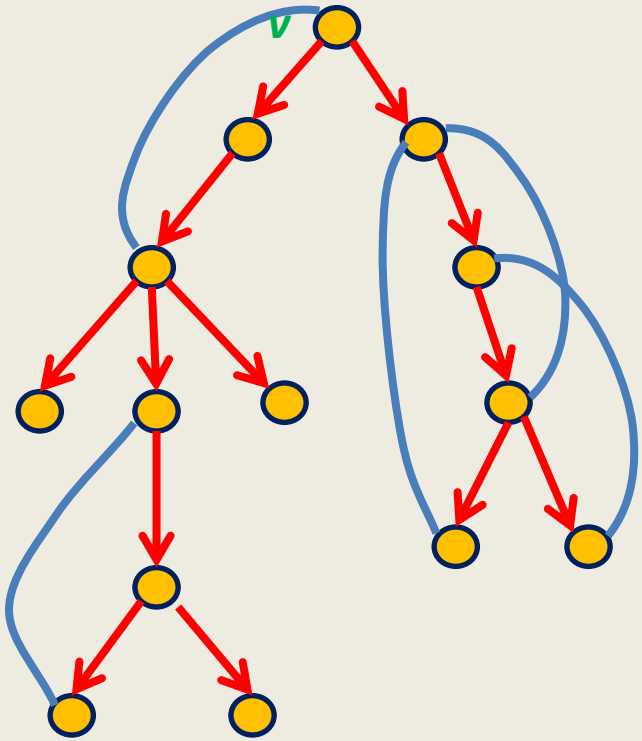


Observation: A graph is biconnected if none of its vertices is an articulation point.

AIM:

Design an **algorithm** to compute all **articulation points** in a given graph.

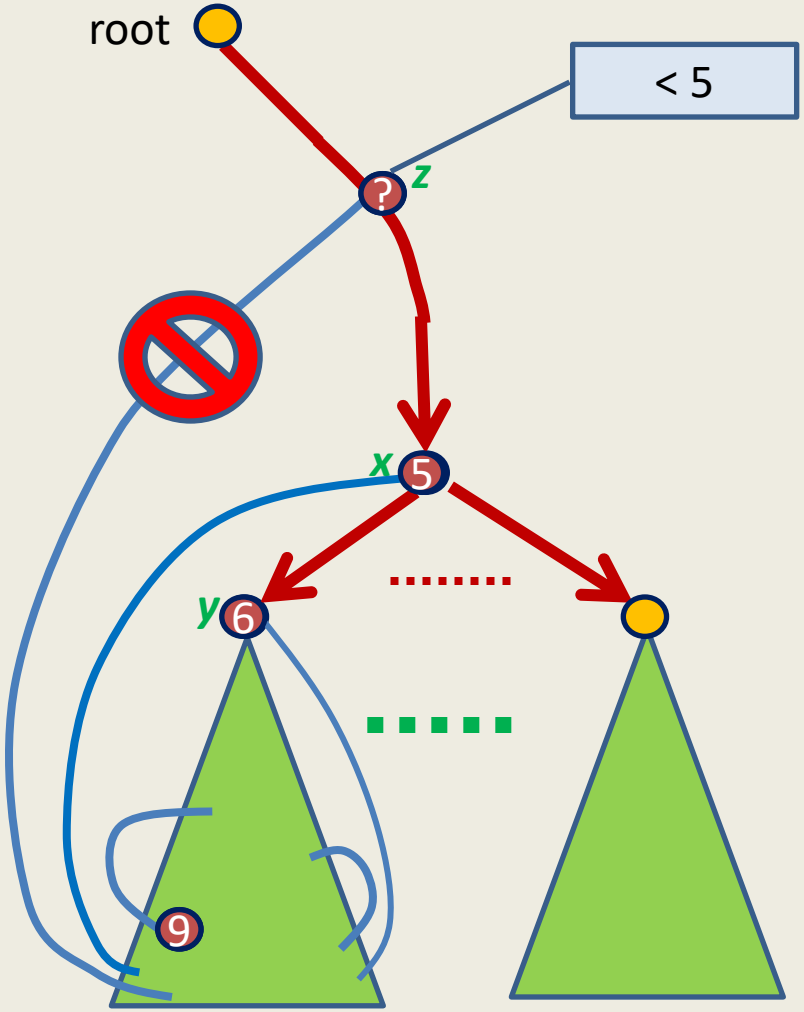
Some observations



- **A leaf node** can never be an **a.p.** ?
- **Root** is an **a.p.** iff it has two or more children.

What about an internal node ?

Necessary and Sufficient condition for *x* to be articulation point



Theorem1:

An internal node ***x*** is **articulation point** iff

x has **at least** one child ***y*** s.t.

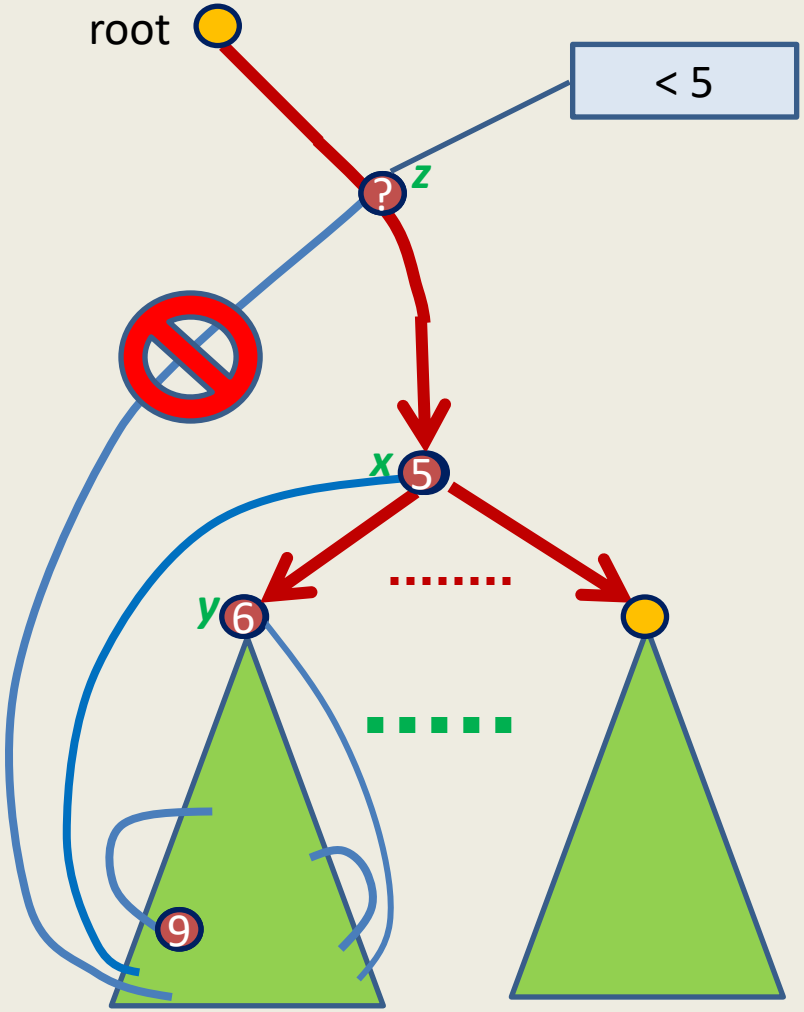
no back edge from **subtree**(**y**) to **ancestor** of ***x***.

➔ 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



Theorem1:

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***).

Theorem2:

An internal node ***x*** is **articulation point** **iff**

it has a child, say **y**, in **DFS** tree such that

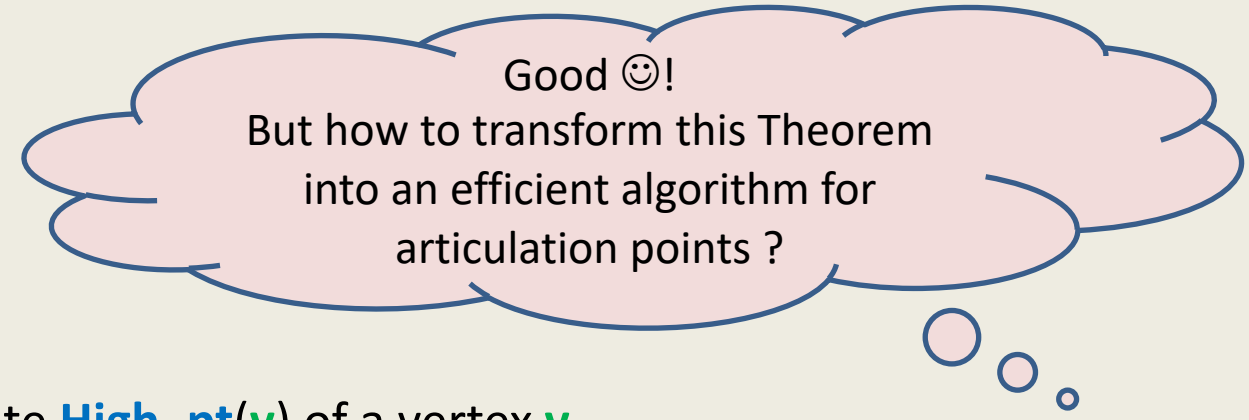
High_pt(**y**) $\geq$ **DFN**(***x***).

Theorem2:

An internal node ***x*** is **articulation point** **iff**

it has a child, say **y**, in **DFS** tree such that

High_pt(**y**) ≥ **DFN**(***x***).



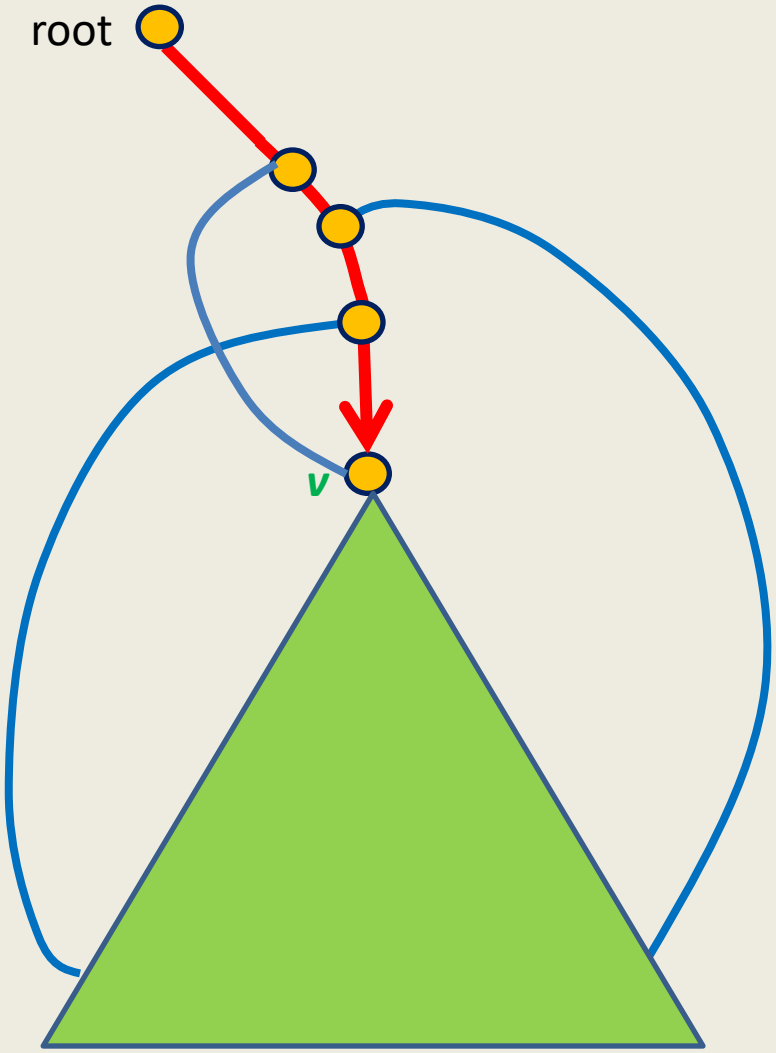
In order to compute **High_pt**(**v**) of a vertex **v**,

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 ☹

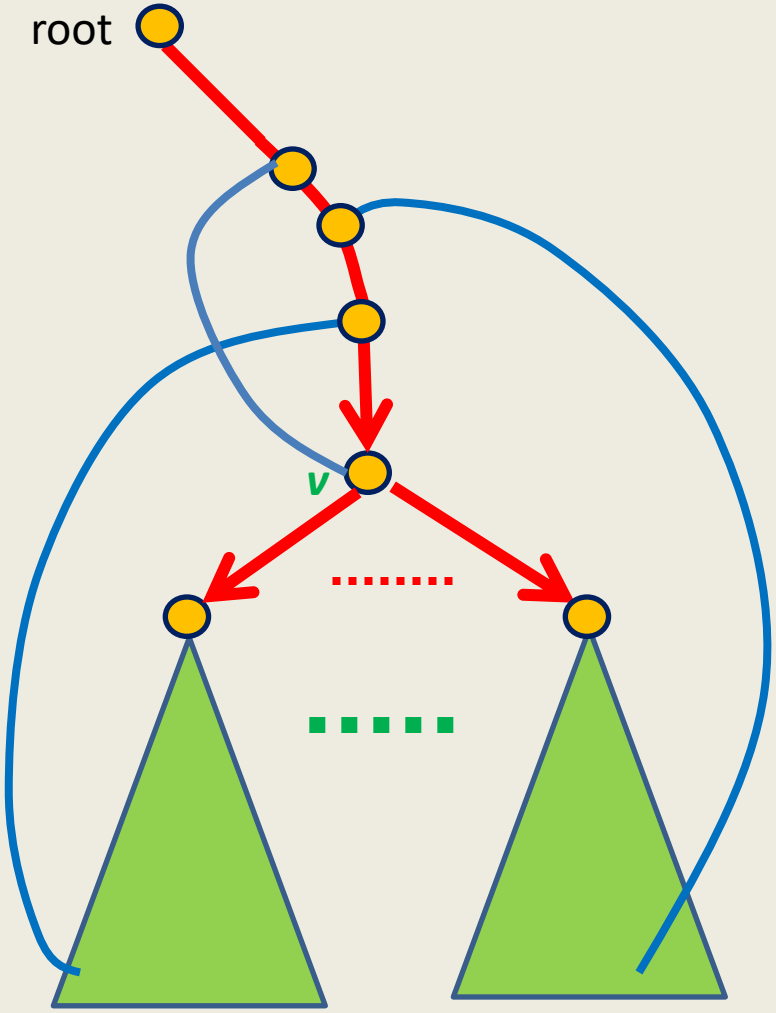
How to compute High_pt(v) efficiently ?



Question: Can we express **High_pt**(**v**) in terms of its **children** and **proper ancestors**?

Exploit
recursive structure of
DFS tree.

How to compute High_pt(v) efficiently ?



Question: Can we express **High_pt**(**v**) in terms of its **children** and **proper ancestors**?

High_pt(***v***) =

$$\min_{(v,w)\in E} \begin{cases} \text{High\_pt}(w) & \text{If } w=\text{child}(v) \\ \text{DFN}(w) & \text{If } w = \text{proper ancestor of } v \\ ? & \end{cases}$$

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) ← true; DFN[v] ← dfn ++;  High_pt[v]←∞ ;
   For each neighbor w of v
   {       if (Visited(w)  = false)
           {   DFS(w) ;  Parent(w) ← v;

               ……..;

               ……..;
           }

           ……..;
   }
}
```

```
DFS-traversal(G)
{  dfn ← 0;
   For each vertex vϵ V  {     Visited(v)←  false;   AP[v]← false   }
   For each vertex v ϵ V {     If (Visited(v ) = false)   DFS(v)        }
}
```

Algorithm for articulation points in a graph *G*

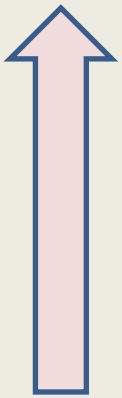
```
DFS(v)
{  Visited(v) ← true; DFN[v] ← dfn ++;  High_pt[v]←∞ ;
   For each neighbor w of v
   {       if (Visited(w)  = false)
           {    Parent(w)← v;   DFS(w);
                High_pt(v) ← min(  High_pt(v)   ,  High_pt(w)  );
                If  High_pt(w) ≥ DFN[v]       AP[v] ← true
           }
           Else if ( Parent(v) ≠  w  )
                      High_pt(v) ← min(   DFN(w)   ,   High_pt(v)   )
   }
}
```

```
DFS-traversal(G)
{  dfn ← 0;
   For each vertex vϵ V  {     Visited(v)←  false;   AP[v]← false   }
   For each vertex v ϵ 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



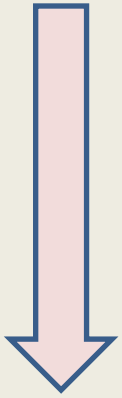
Simplicity

Lists: (arrays, linked lists)

Binary Heap

Binary Search Trees

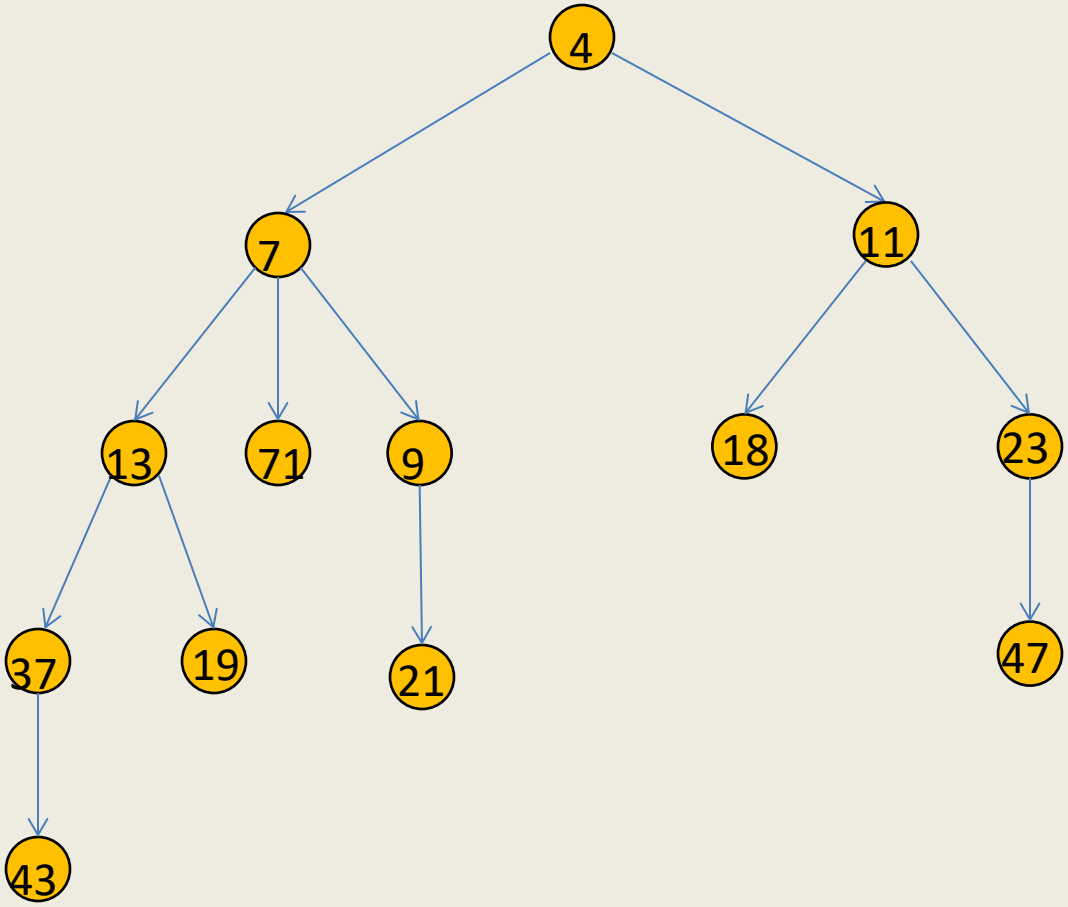
Range of efficient functions



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

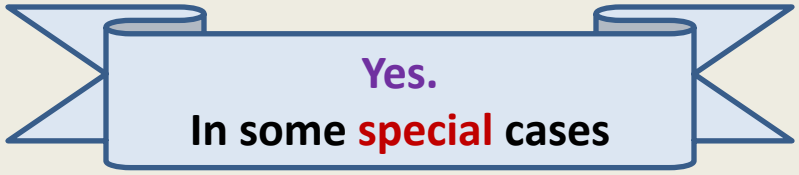
-- much **simpler** and

-- more **efficient**

Existing heap data structures

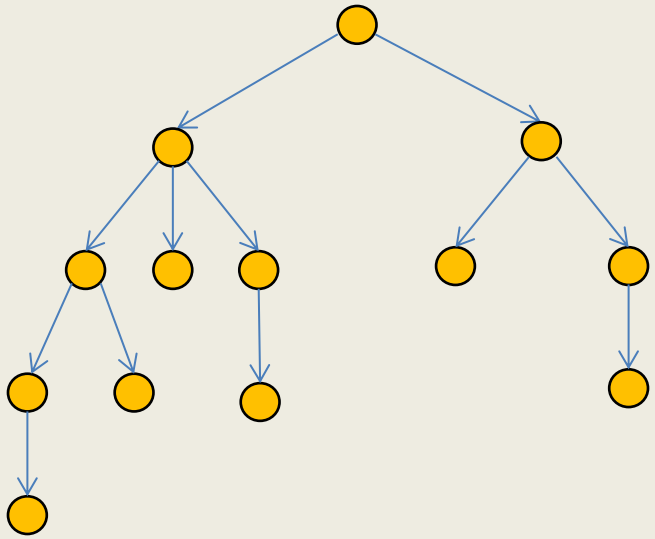
- **Binary heap**
- **Binomial heap**
- **Fibonacci heap**
- **Soft heap**

Can we implement a binary tree using an array ?



fundamental question

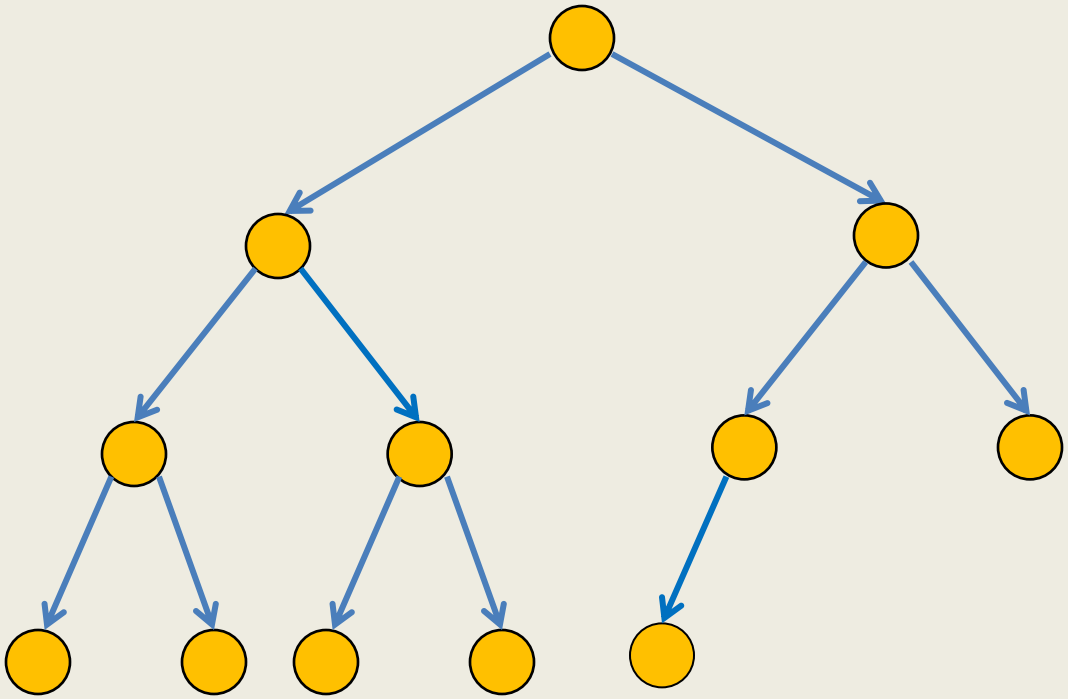
Question: What does the implementation of a tree data structure require ?



Answer: a mechanism to

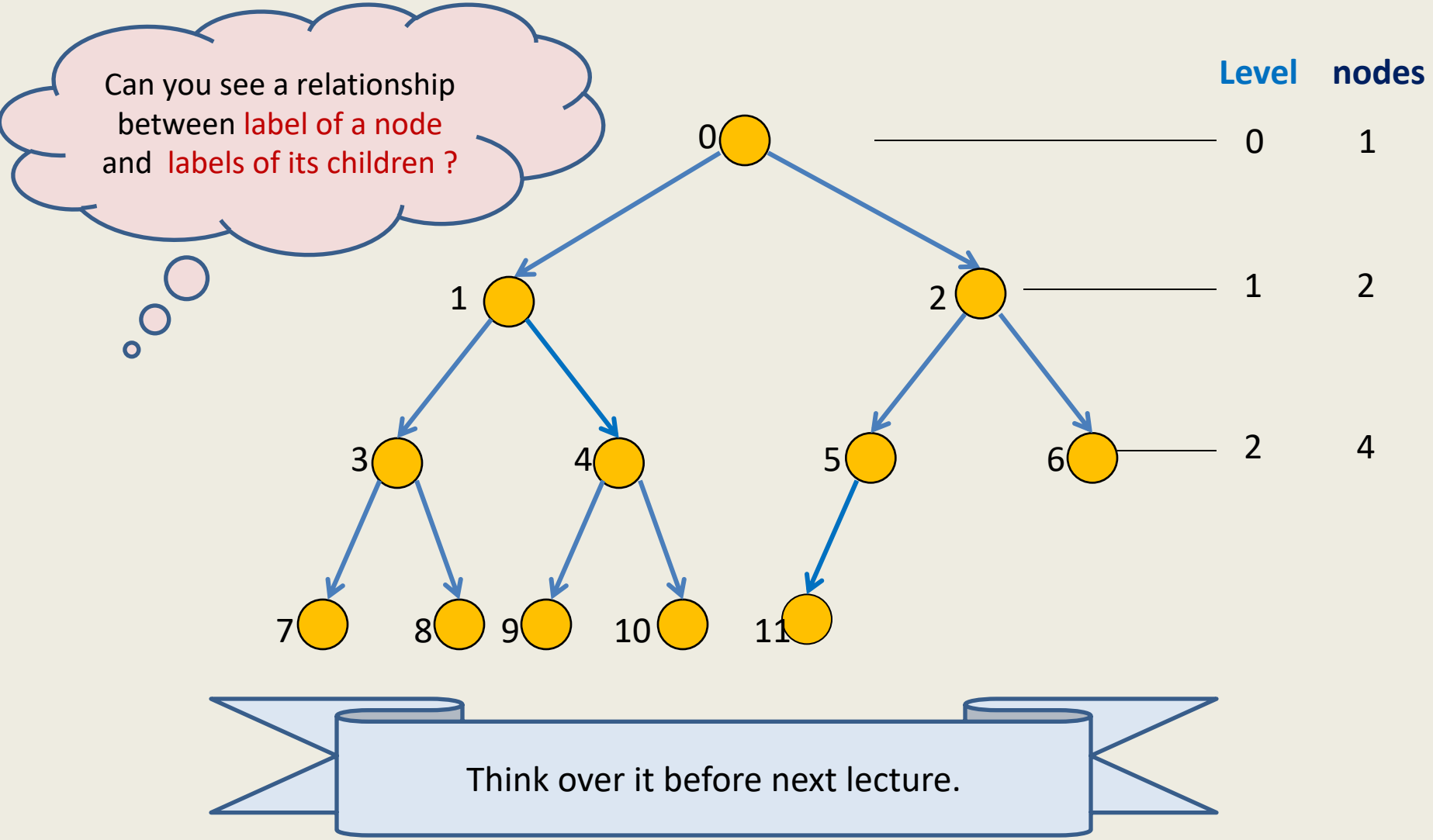
- access **parent** of a node
- access **children** of a node.

A complete binary tree



A complete binary of 12 nodes.

A complete binary tree



Data Structures and Algorithms

(ESO207)

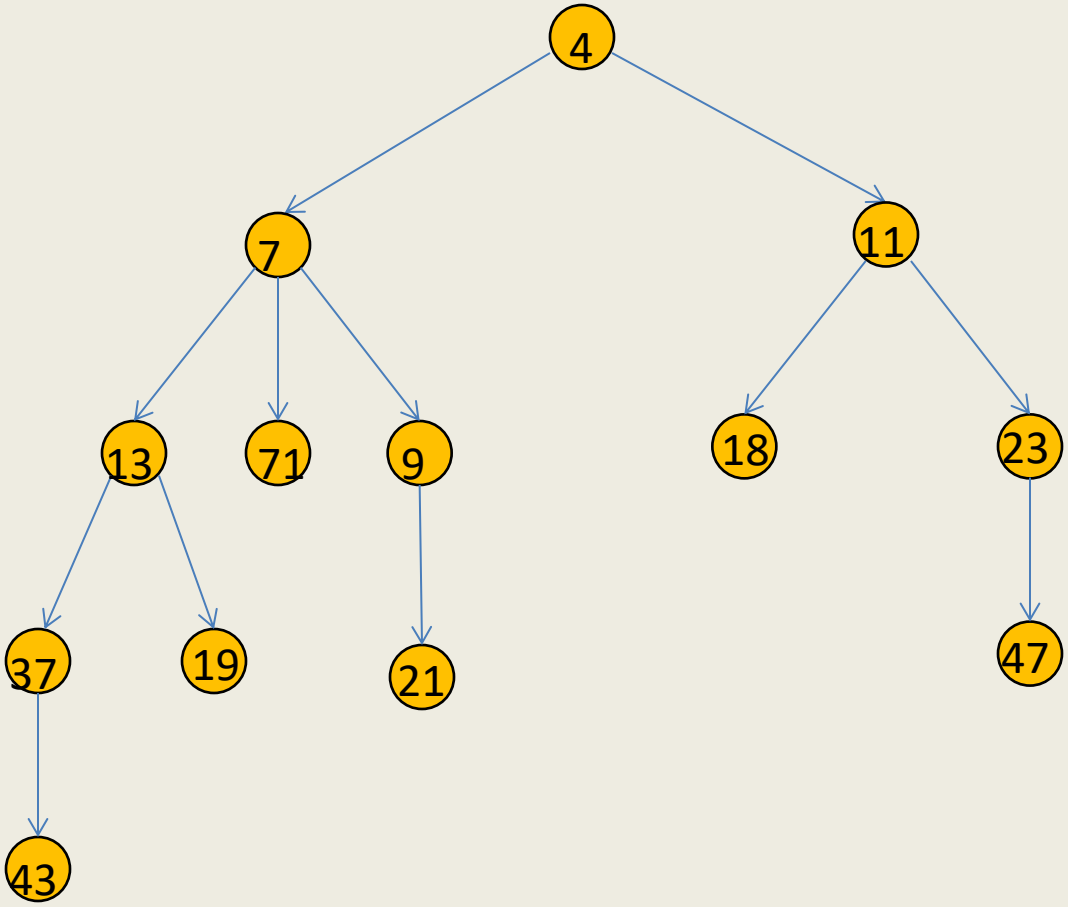
Lecture 28:

- **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

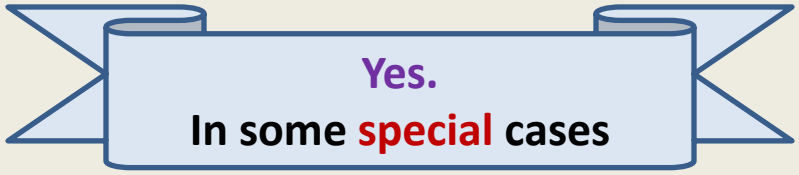
Query Operations

- **Find-min**: report the smallest key stored in the heap.

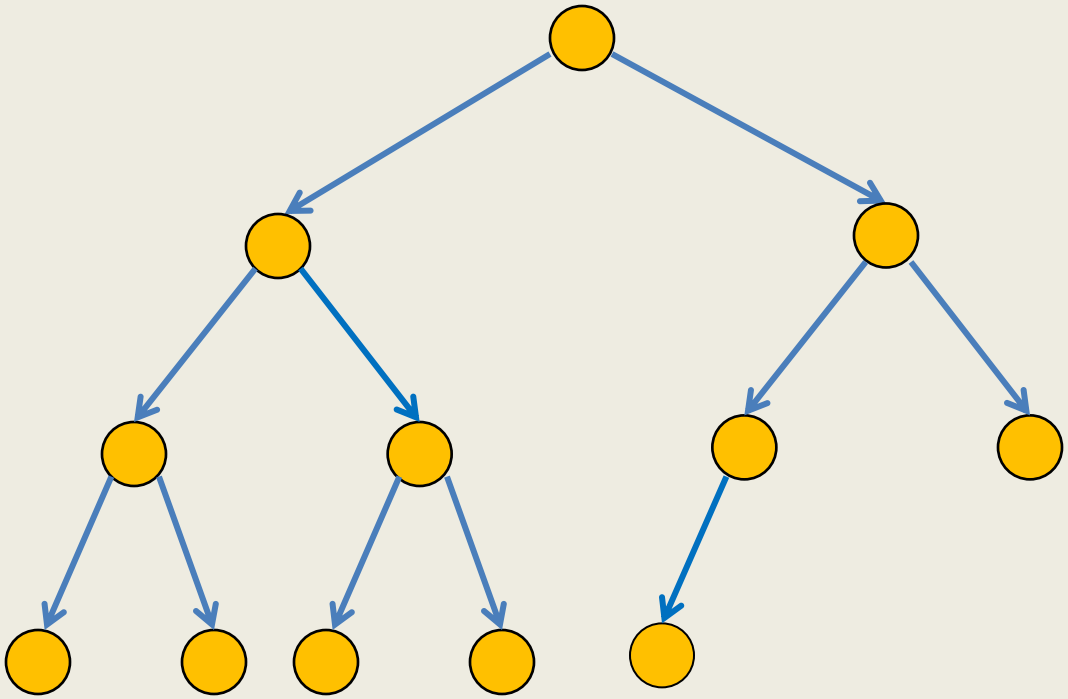
Update Operations

- **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 ?



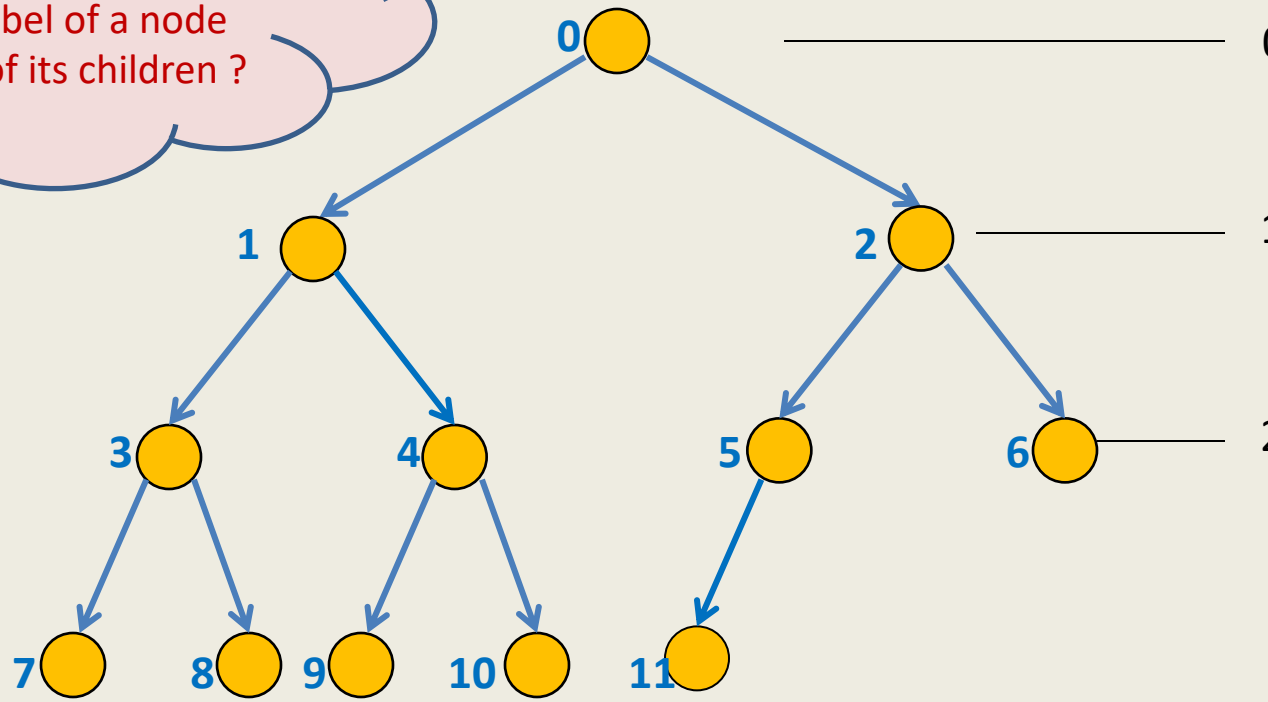
A complete binary tree



A complete binary of 12 nodes.

A complete binary tree

Can you see a relationship between label of a node and labels of its children ?



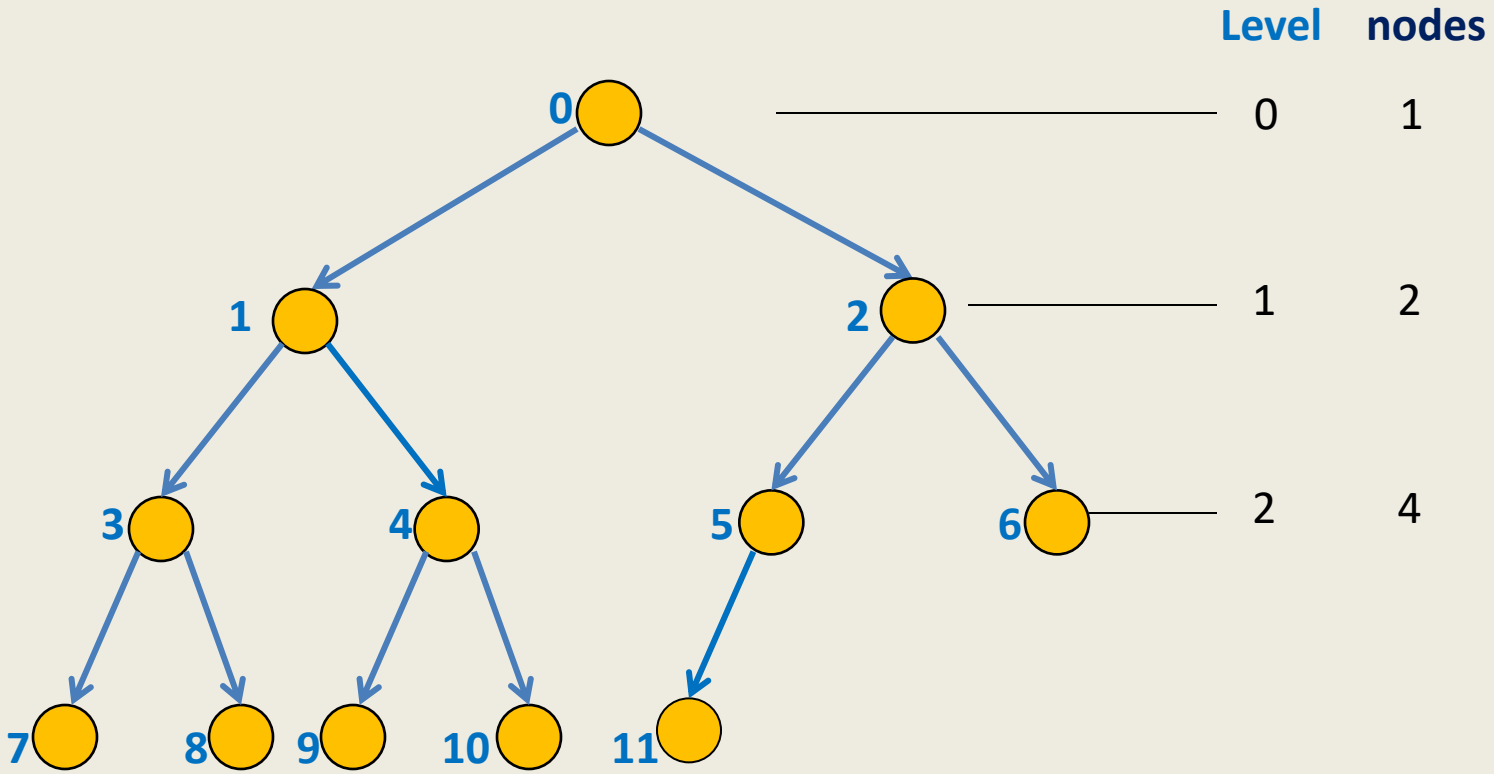
The label of the **leftmost node** at level i = $2^i - 1$

The label of a **node** **v** at level i $k - 2$

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= $2^{i+1} + 2k - 2$

A complete binary tree



Let **v** be a node with label j.

Label of **left child(v)** = $2j+1$

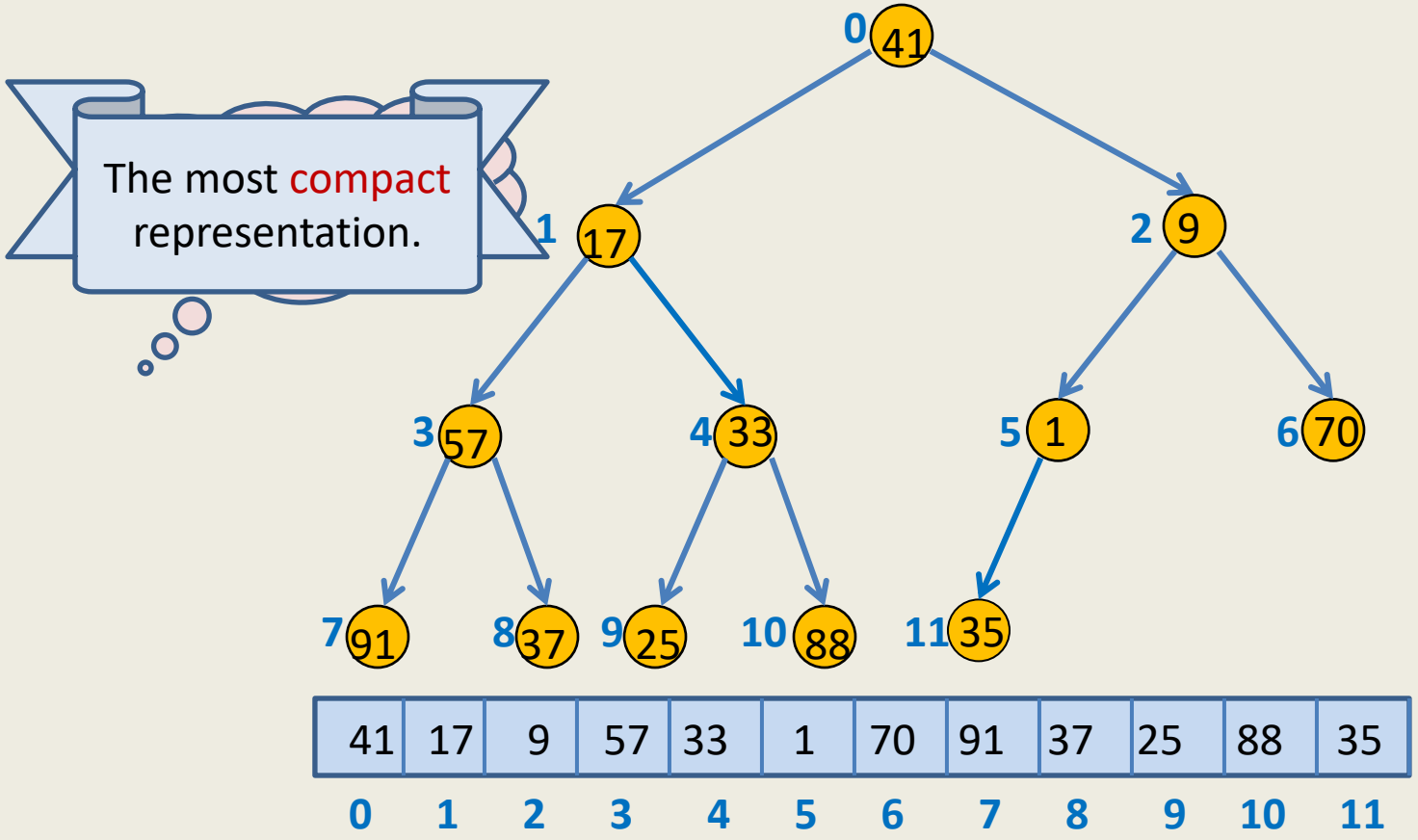
Label of **right child(v)** = $2j+2$

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 ?

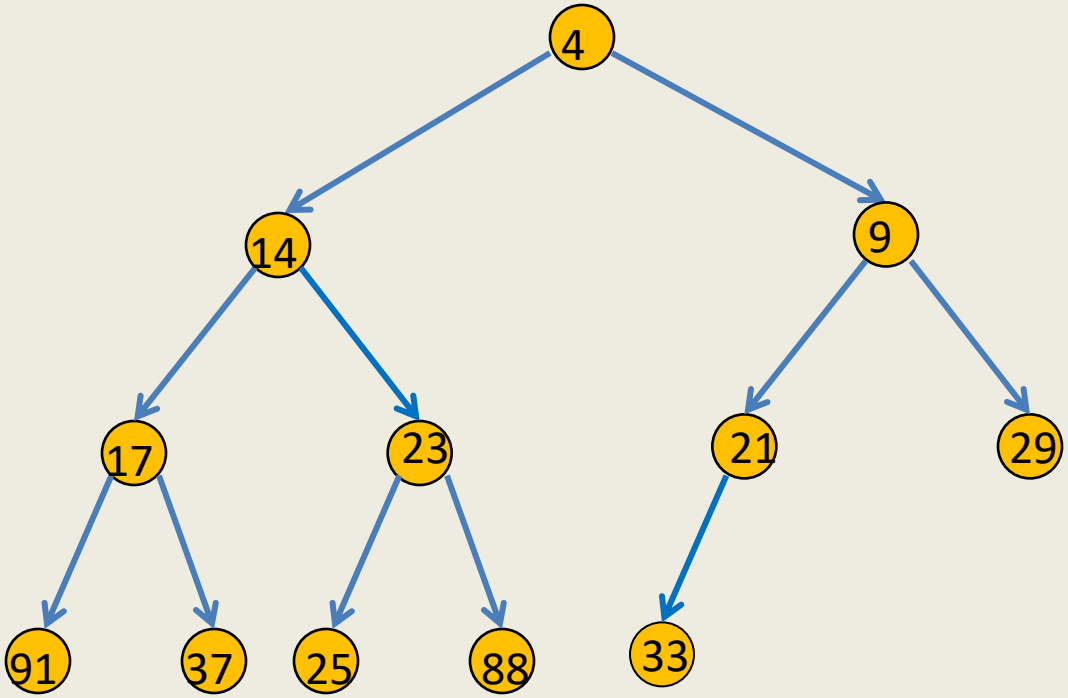
Answer: A complete binary tree can be **implemented** by an array.



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

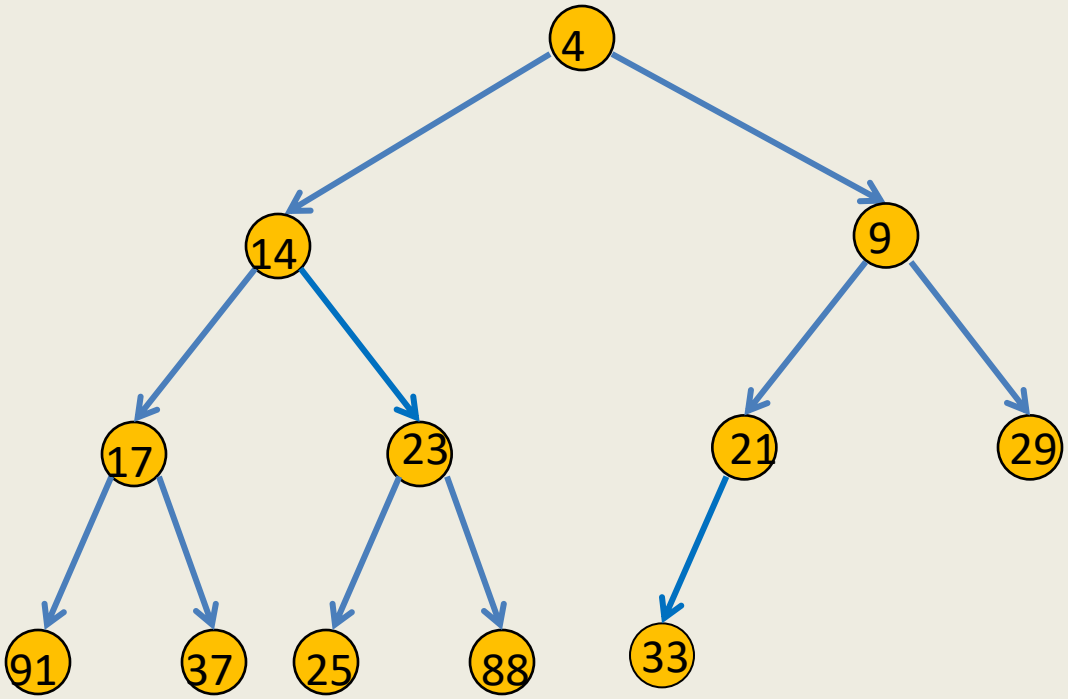
n

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].



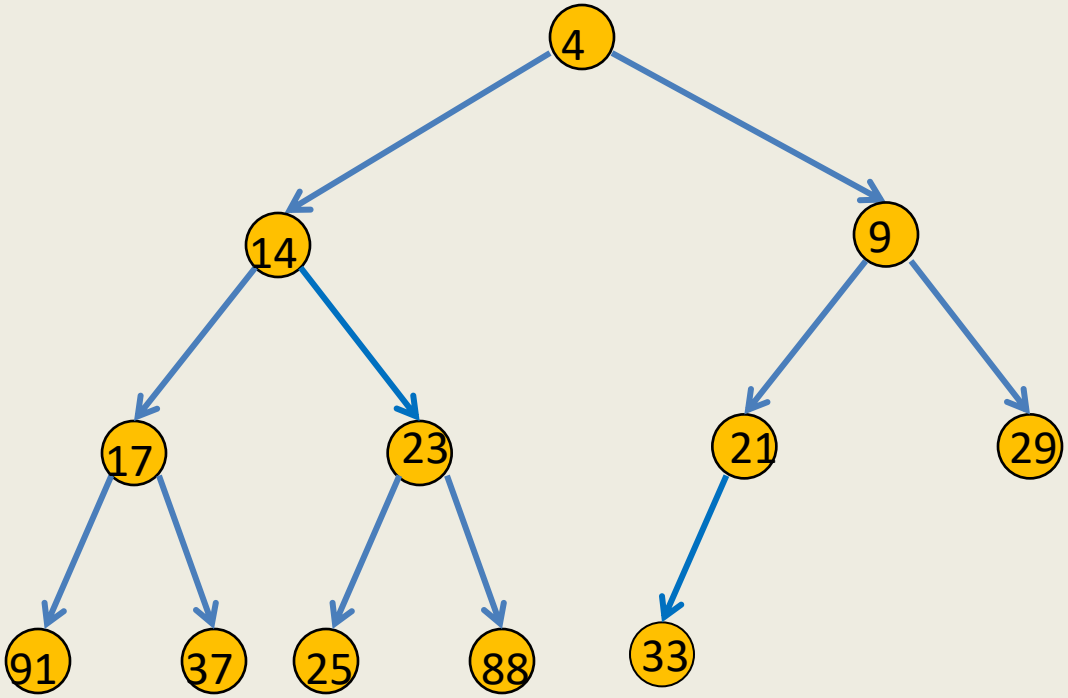
H	4	14	9	17	23	21	29	91	37	25	88	33			

Extract_min(H)

Think hard on designing efficient algorithm for this operation.

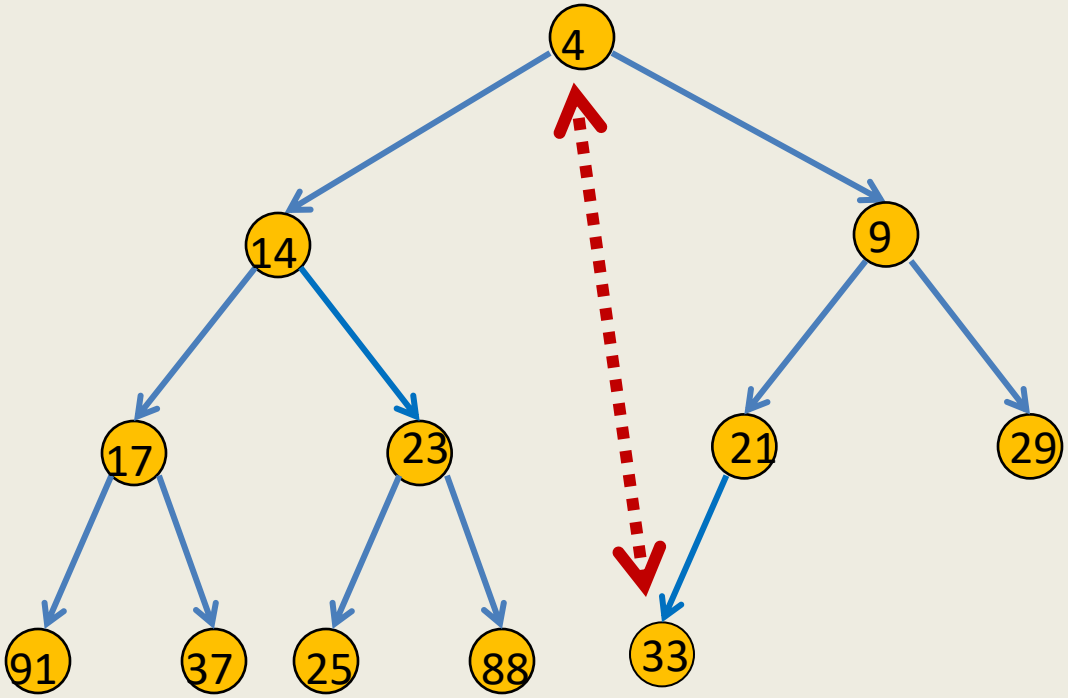
The challenge is:

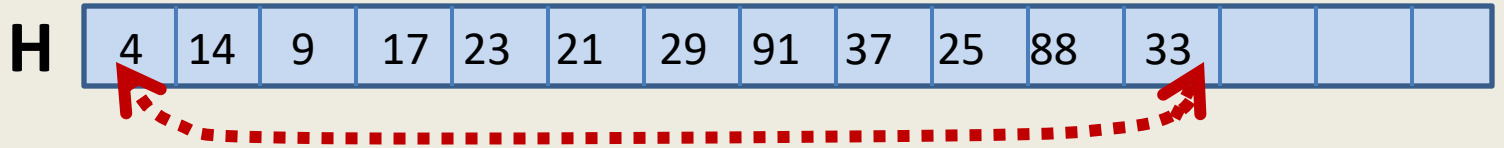
how to preserve the complete binary tree structure as well as the heap property ?



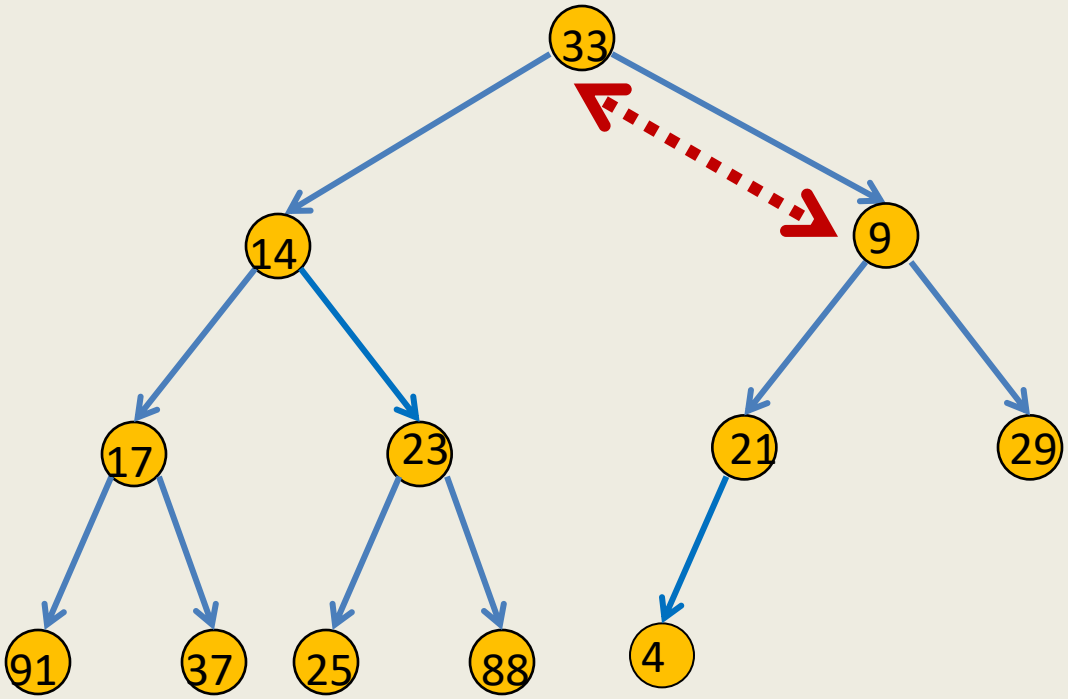
H	4	14	9	17	23	21	29	91	37	25	88	33			

Extract_min(H)





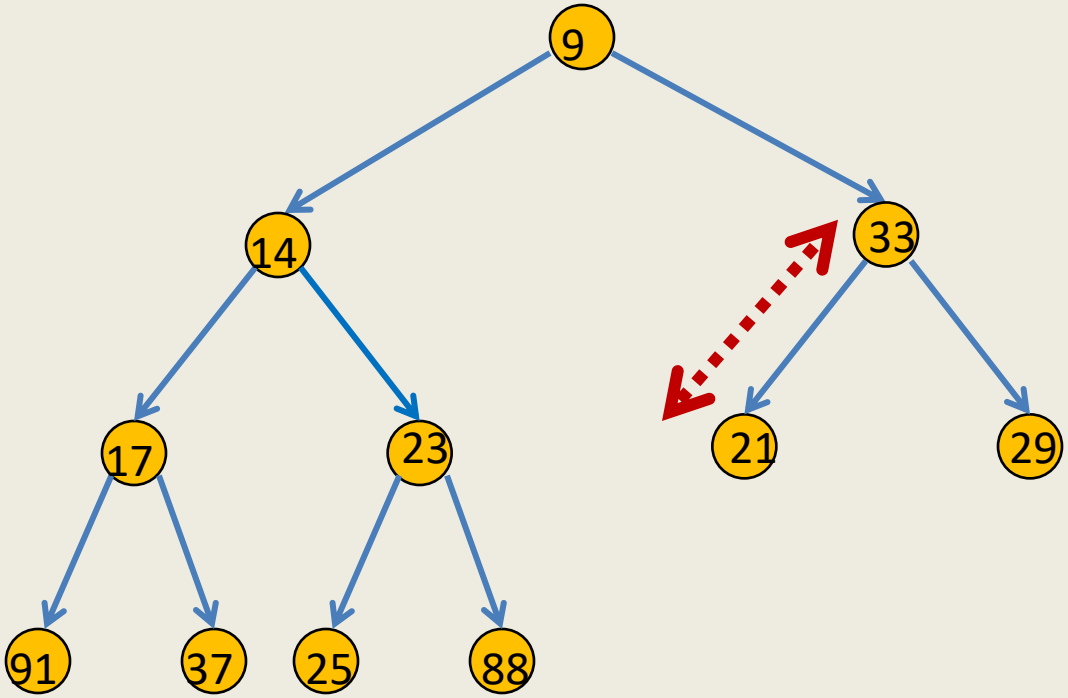
Extract_min(H)

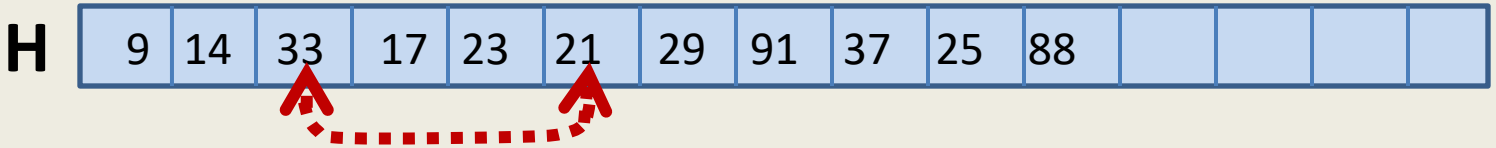


H

33	14	9	17	23	21	29	91	37	25	88	4			

Extract_min(H)



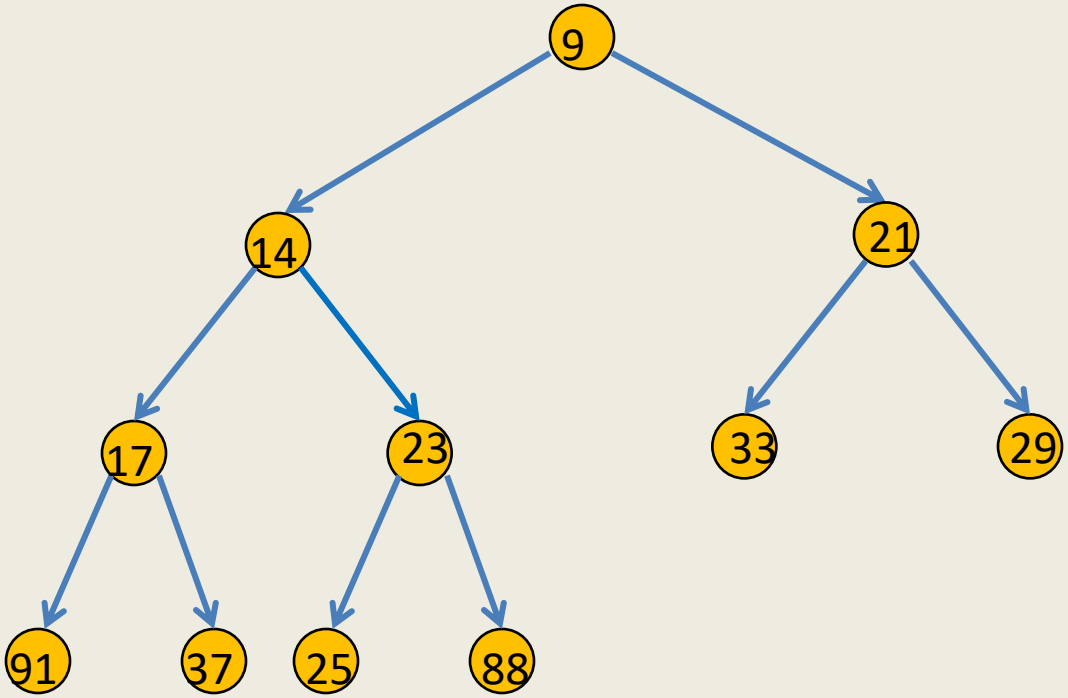


Extract_min(H)

We are done.

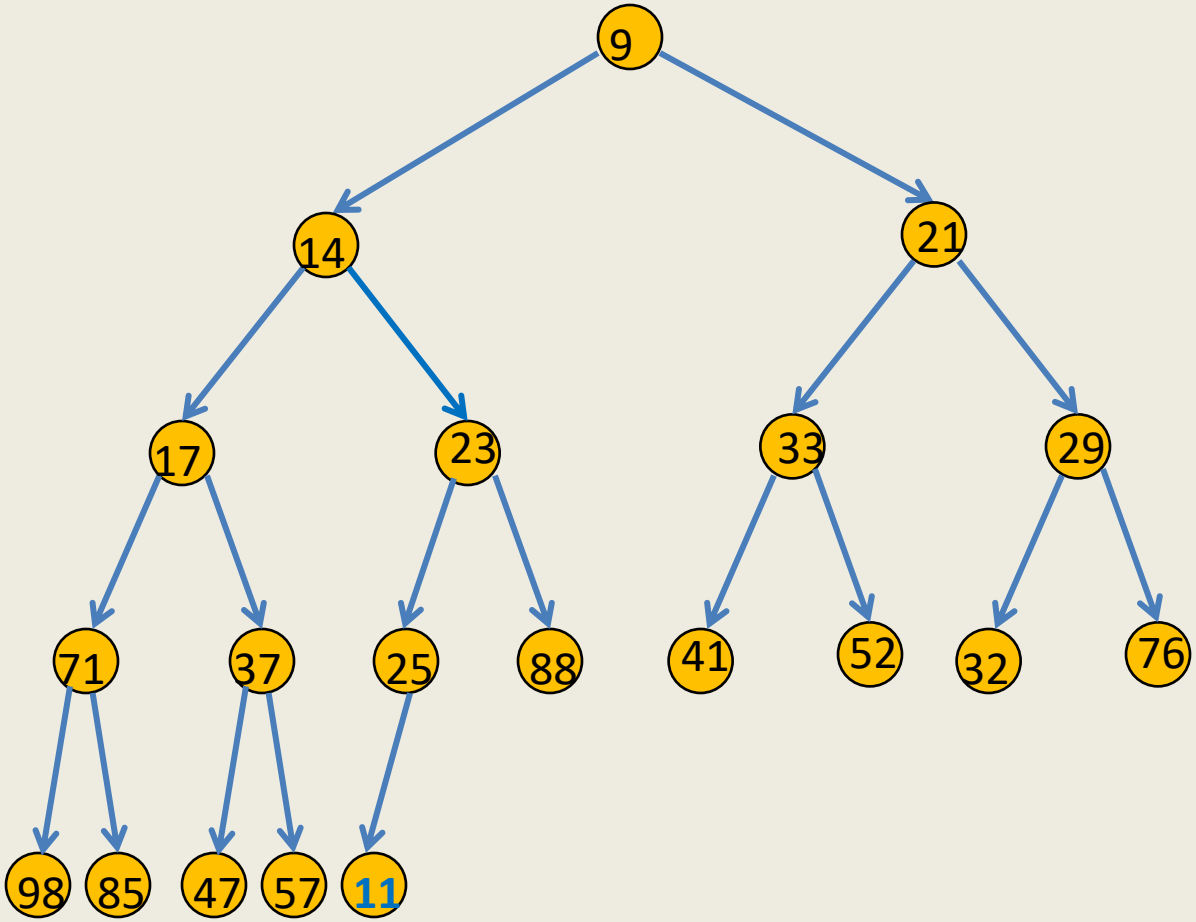
The no. of operations performed = **O**(no. of levels in binary heap)

= **O**(log n) …show it as an **homework exercise**.



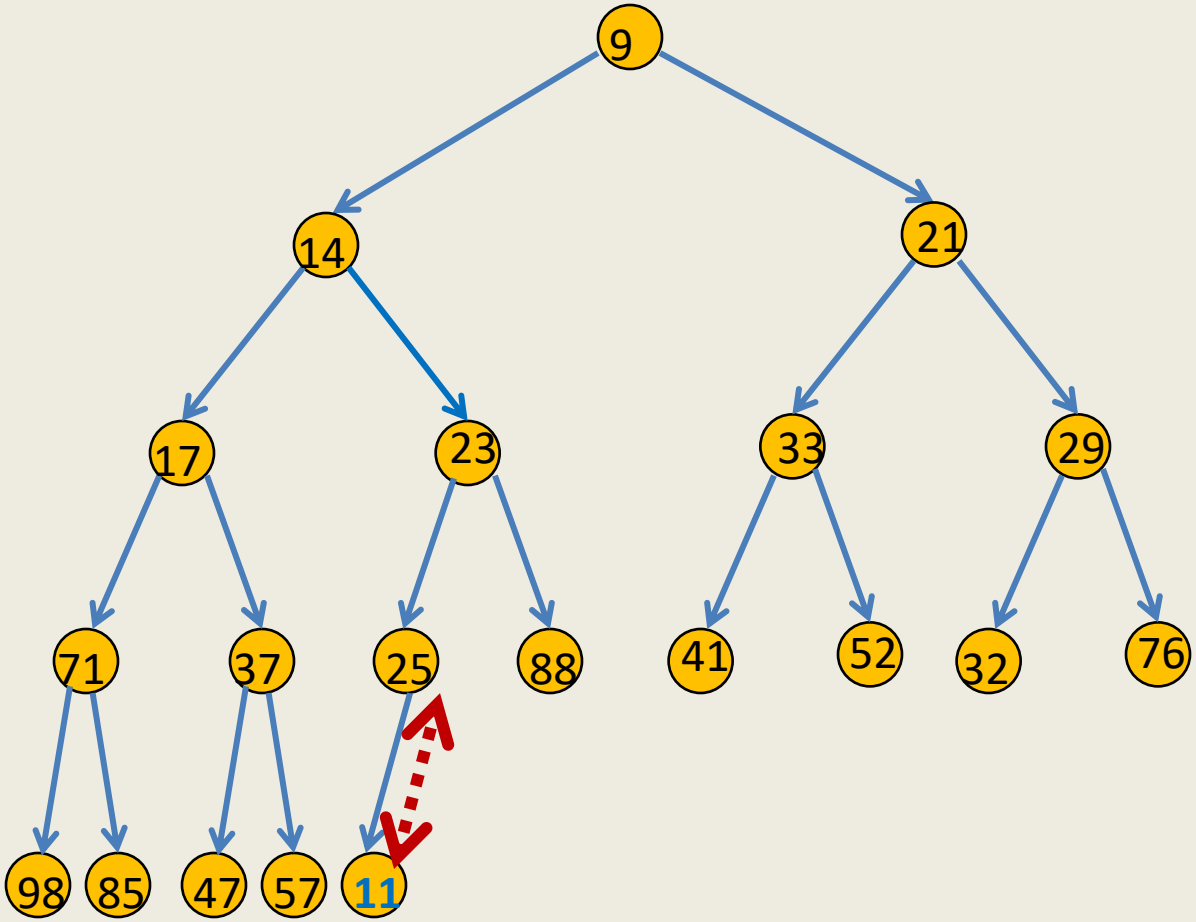
H	9	14	21	17	23	33	29	91	37	25	88				

Insert(x,H)



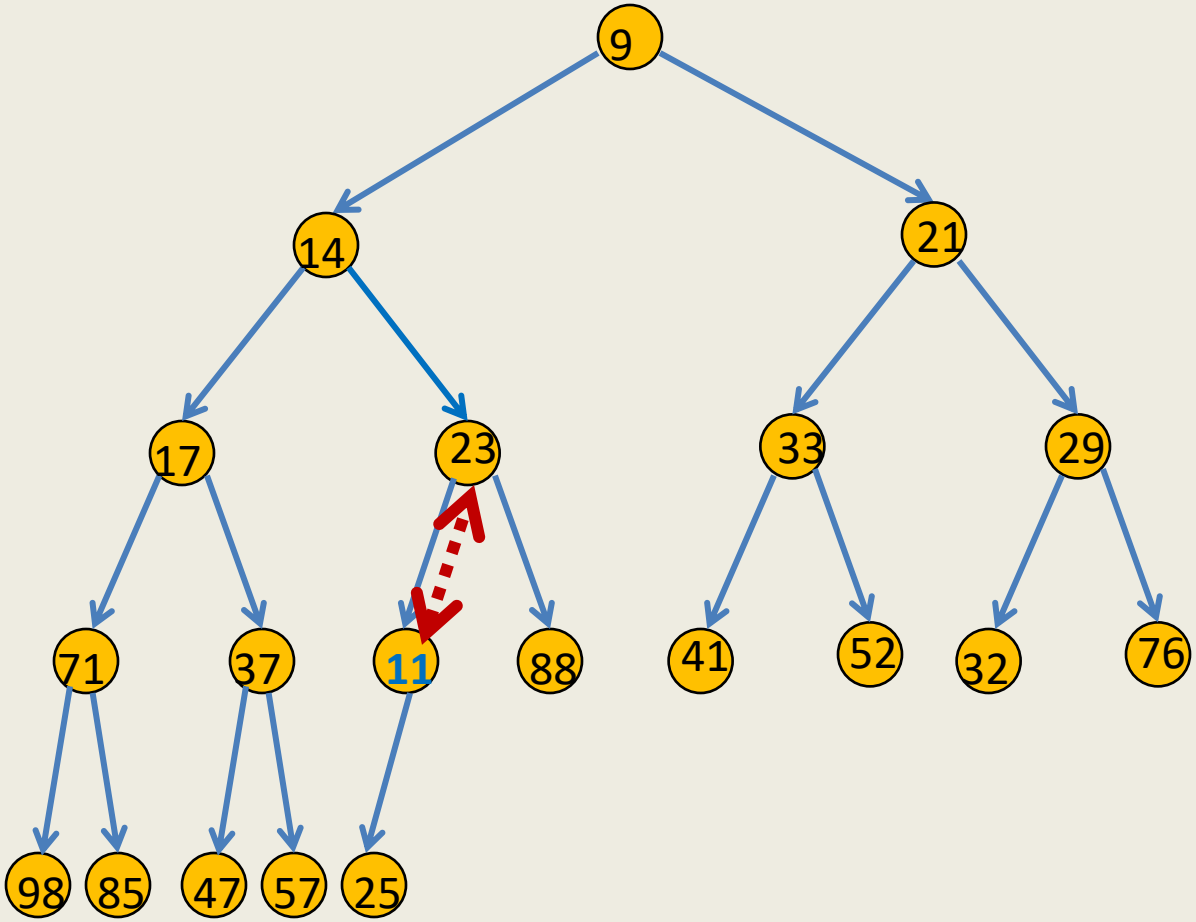
H	9	14	21	17	23	33	29	71	37	25	88	41	52	32	76	98	85	47	57	11	

Insert(x,H)


9
14
21
17
23
33
29
71
37
25
88
41
52
32
76
98
85
47
57
11

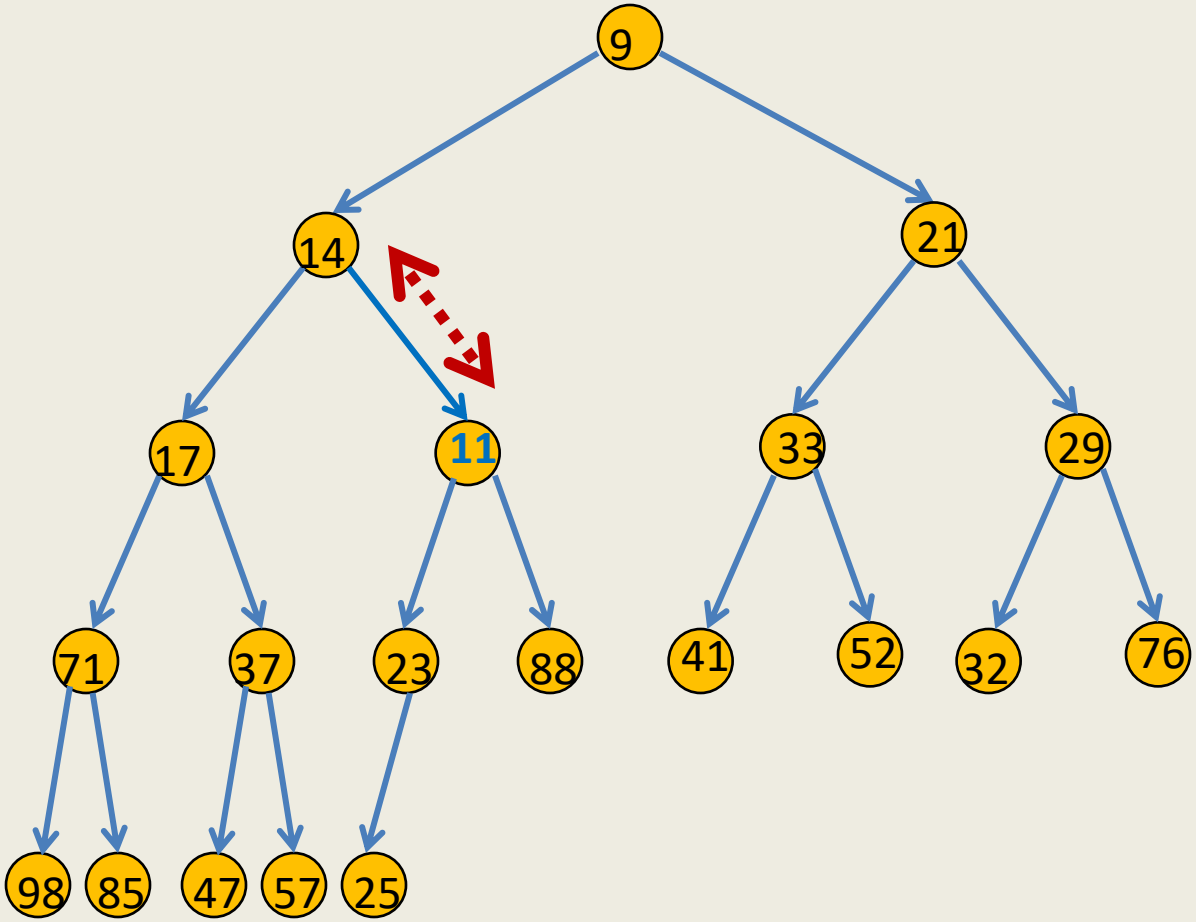

H
9
14
21
17
23
33
29
71
37
25
88
41
52
32
76
98
85
47
57
11

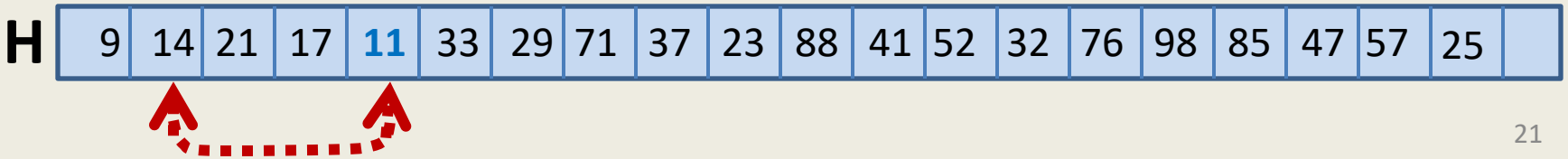
Insert(x,H)



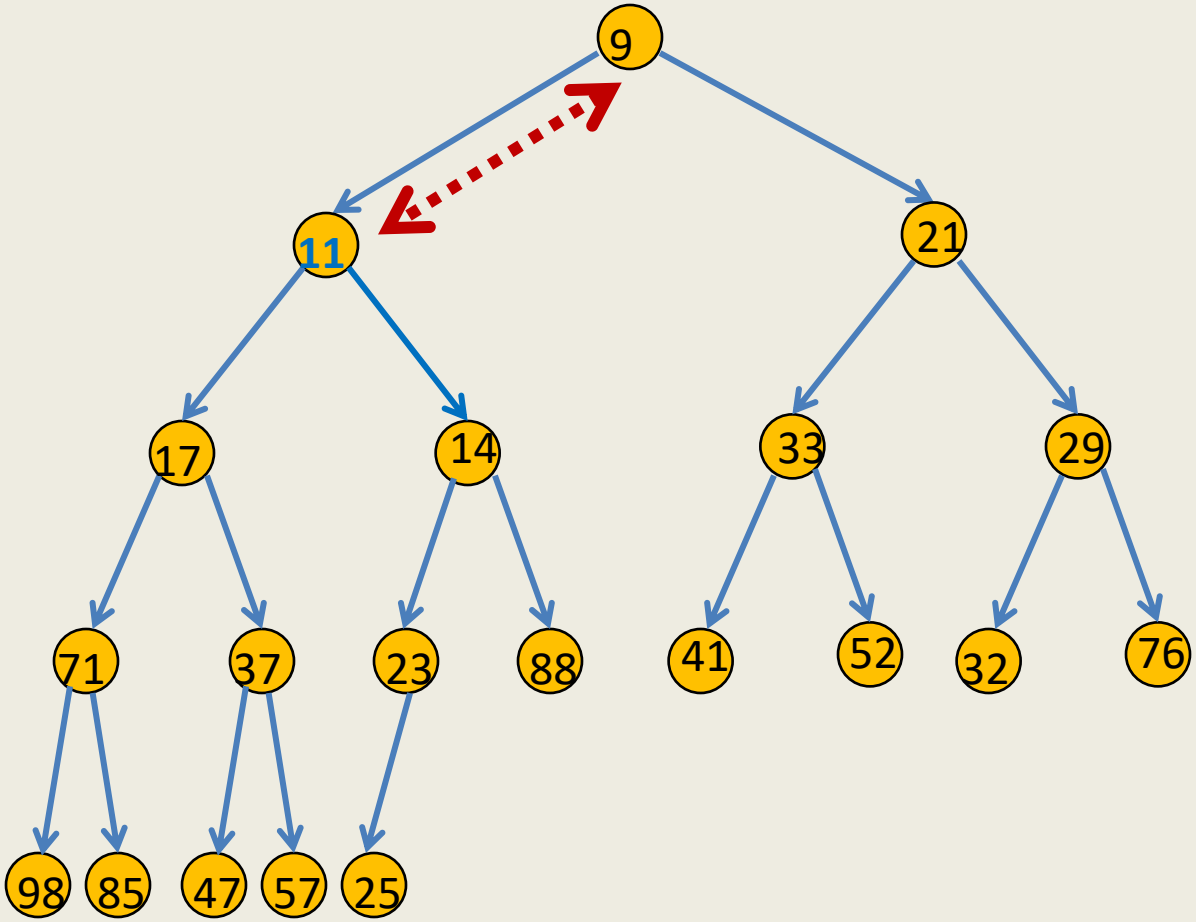
H	9	14	21	17	23	33	29	71	37	11	88	41	52	32	76	98	85	47	57	25	

Insert(x,H)


9
14
21
17
11
33
29
71
37
23
88
41
52
32
76
98
85
47
57
25


H
9
14
21
17
11
33
29
71
37
23
88
41
52
32
76
98
85
47
57
25

Insert(x,H)



H

9	11	21	17	14	33	29	71	37	23	88	41	52	32	76	98	85	47	57	25	

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**(**H1,H2**): Merge two heaps **H1 and H2**.
  - **O(**n**) time** where n = total number of elements in **H1** and **H2**

    (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(**H1**,**H2**)	**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 $\{x_0, \ldots, x_{n-1}\}$, build a binary **heap H** storing them.

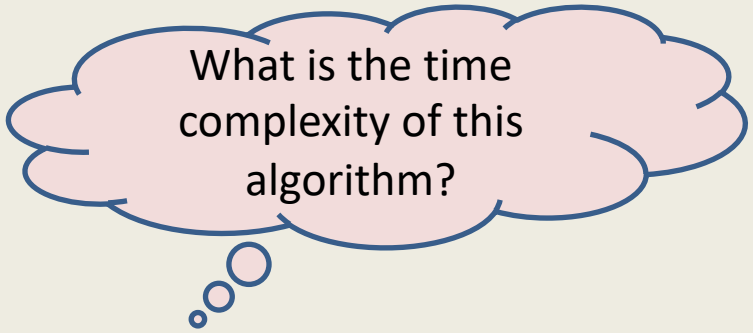
Trivial solution:

(Building the Binary heap incrementally**)**

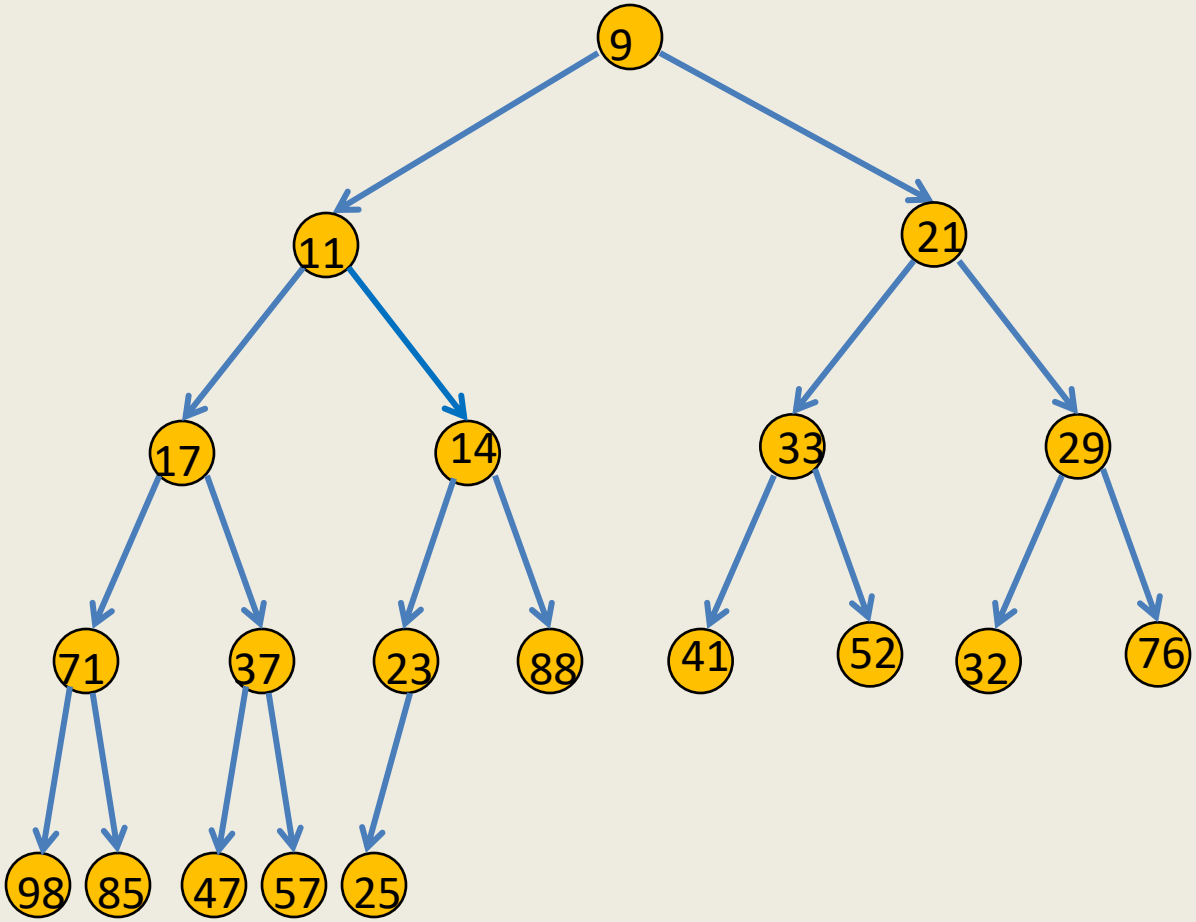
CreateHeap(**H**);

For(i = 0 to $n-1$ **)**

Insert(x_i,**H);**



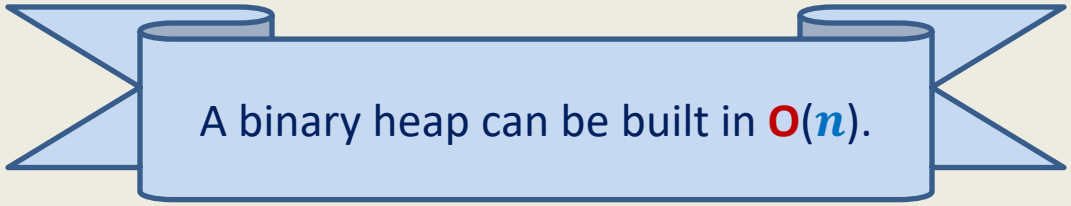
Building a Binary heap incrementally



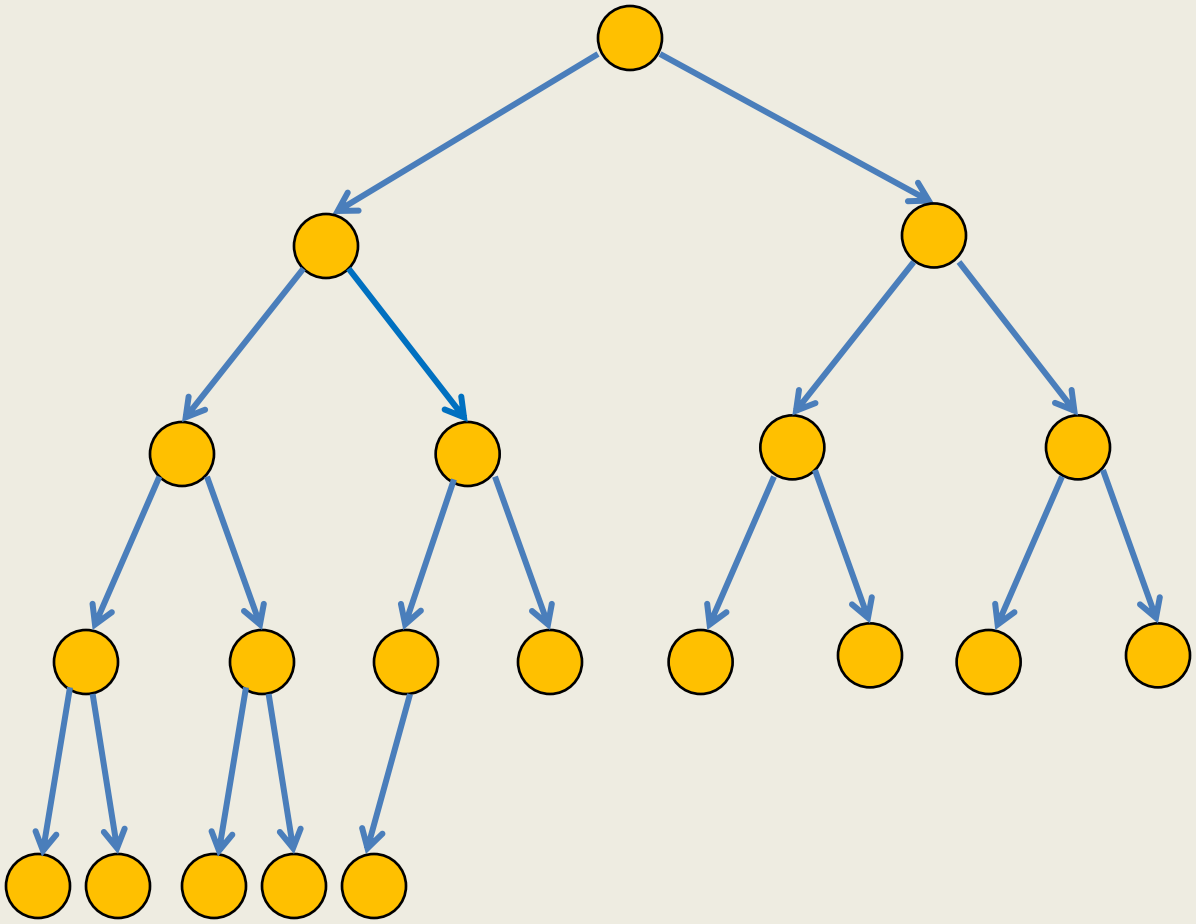
H	9	11	21	17	14	33	29	71	37	23	88	41	52	32	76	98	85	47	57	25

Time complexity

Theorem: Time complexity of building a binary heap **incrementally** is **O**(n log n).

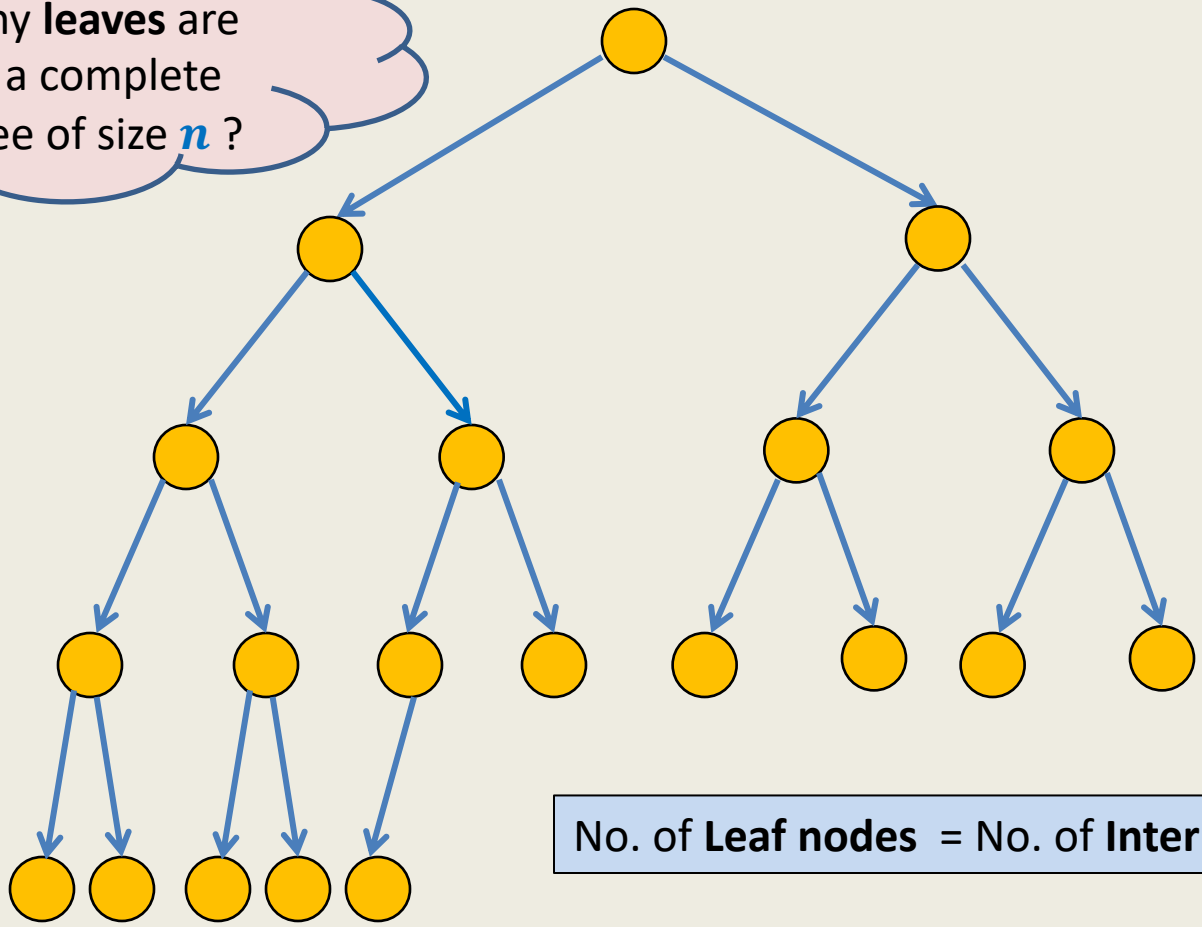


A complete binary tree



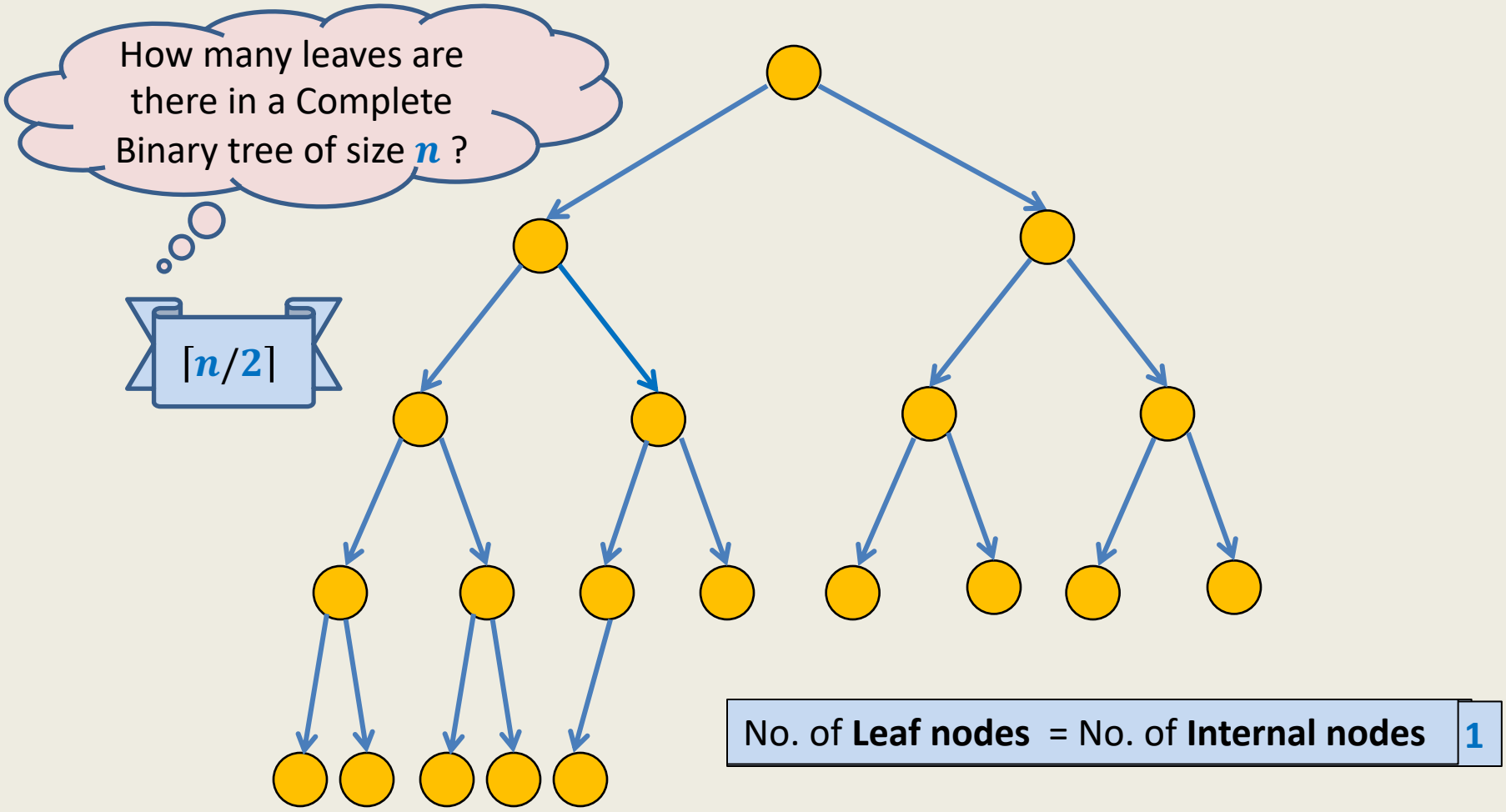
A complete binary tree

How many **leaves** are there in a complete Binary tree of size n ?

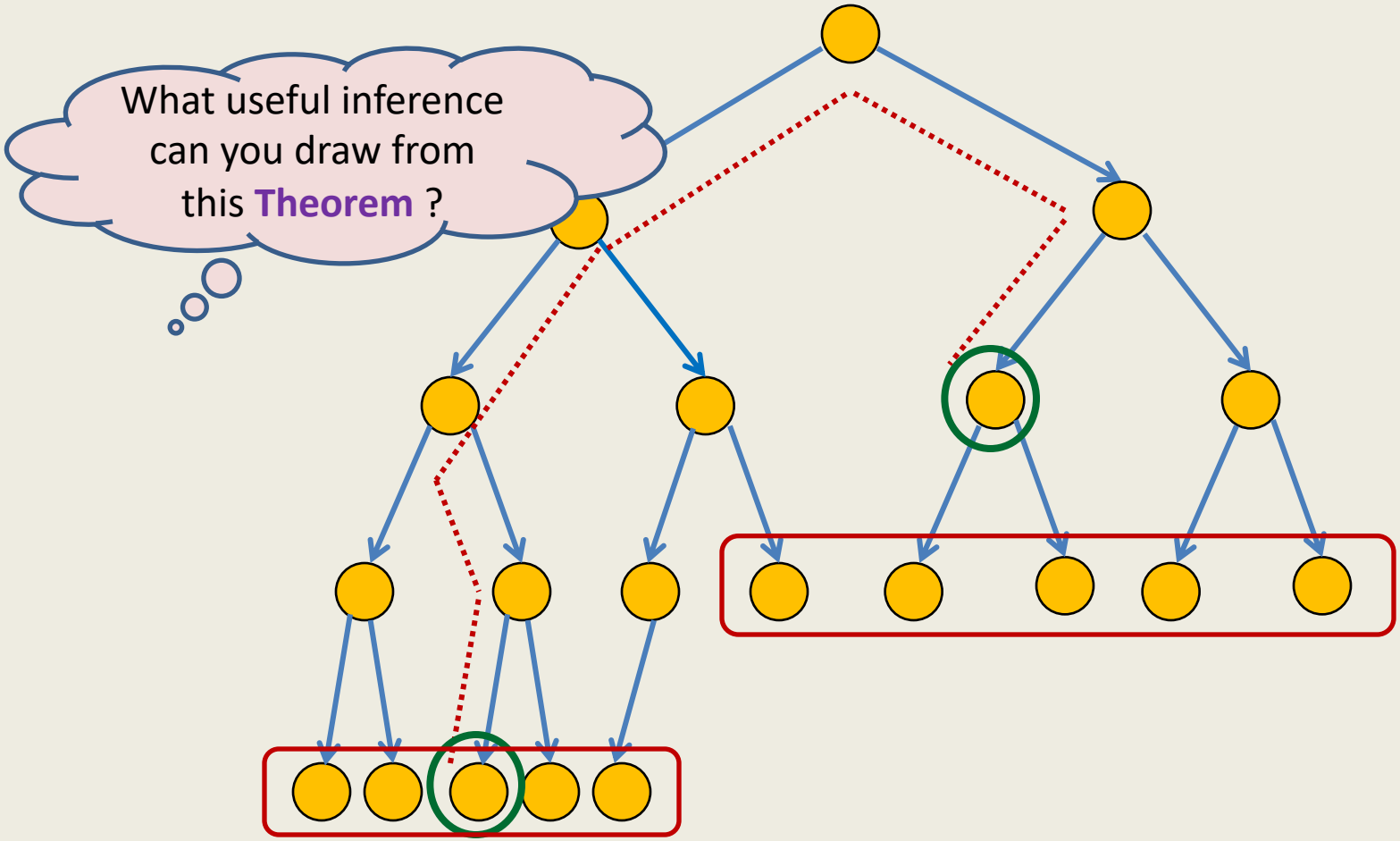


No. of **Leaf nodes** = No. of **Internal nodes** + **1**

A complete binary tree



Building a Binary heap incrementally



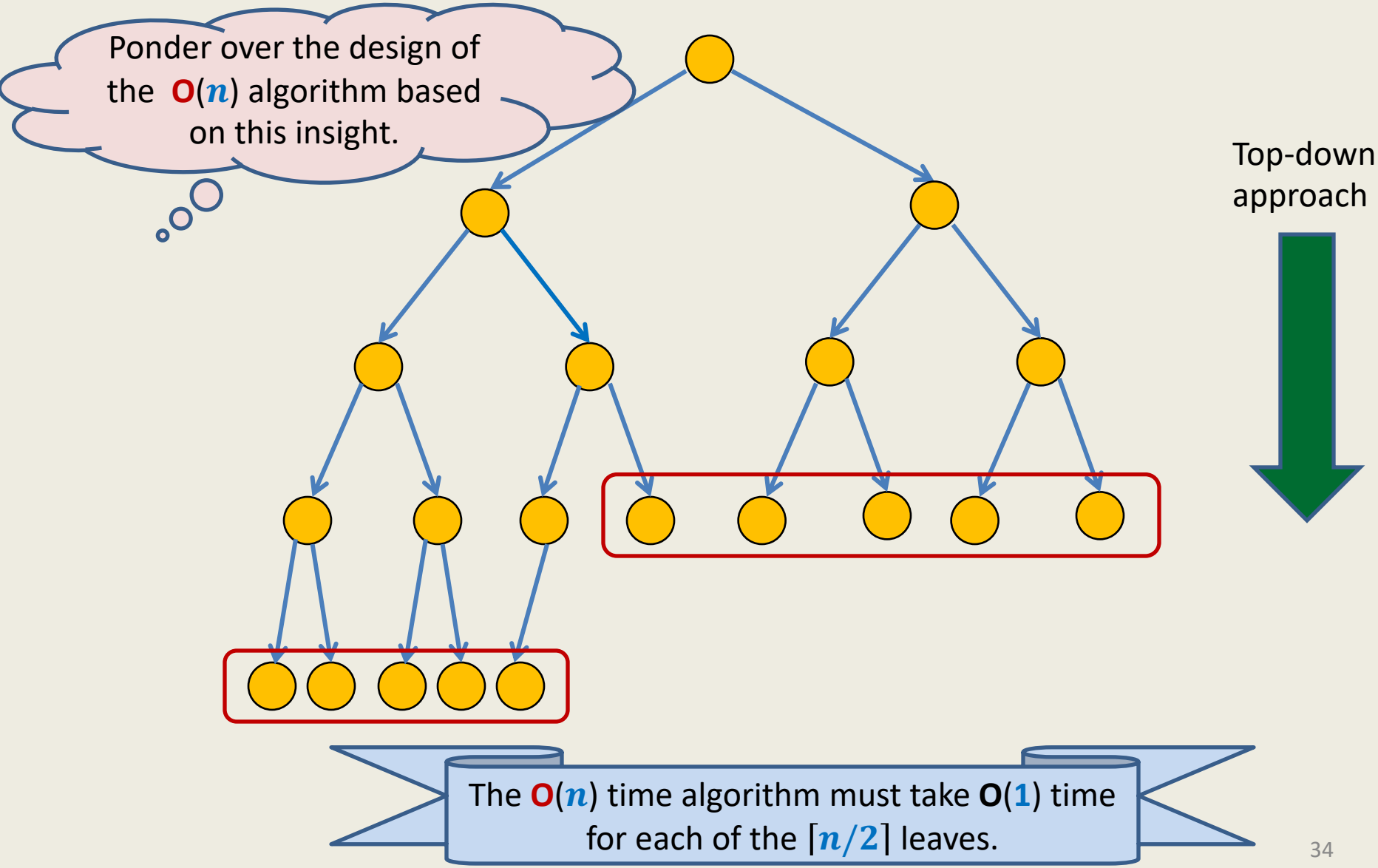
Top-down approach

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



Data Structures and Algorithms

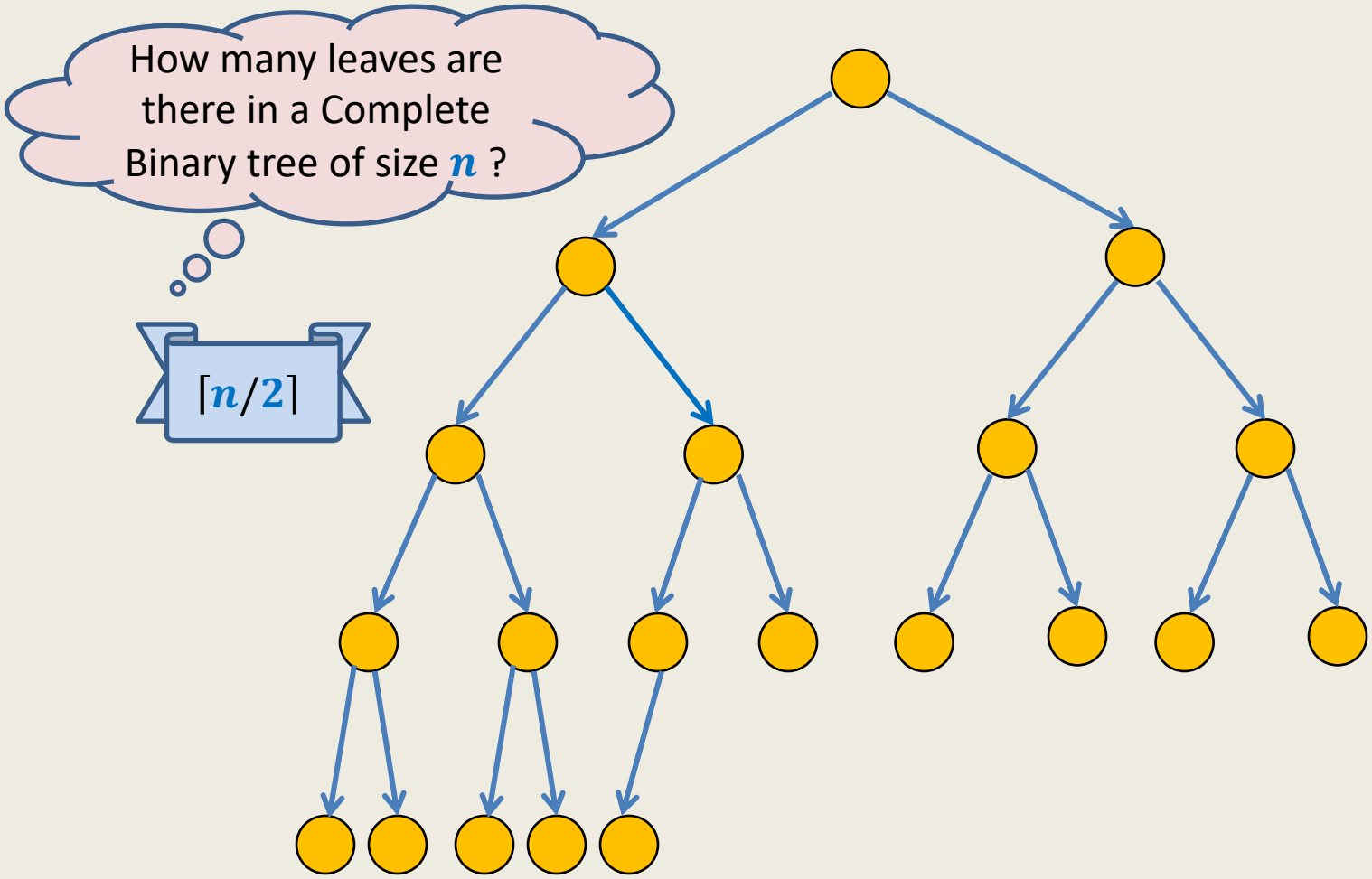
(ESO207)

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



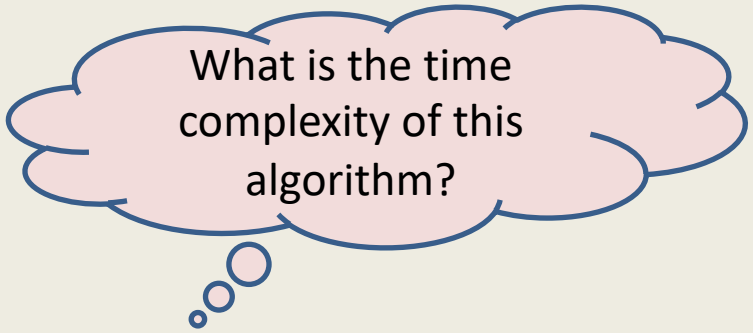
Building a Binary heap

Problem: Given n elements $\{x_0, \ldots, x_{n-1}\}$, build a binary **heap H** storing them.

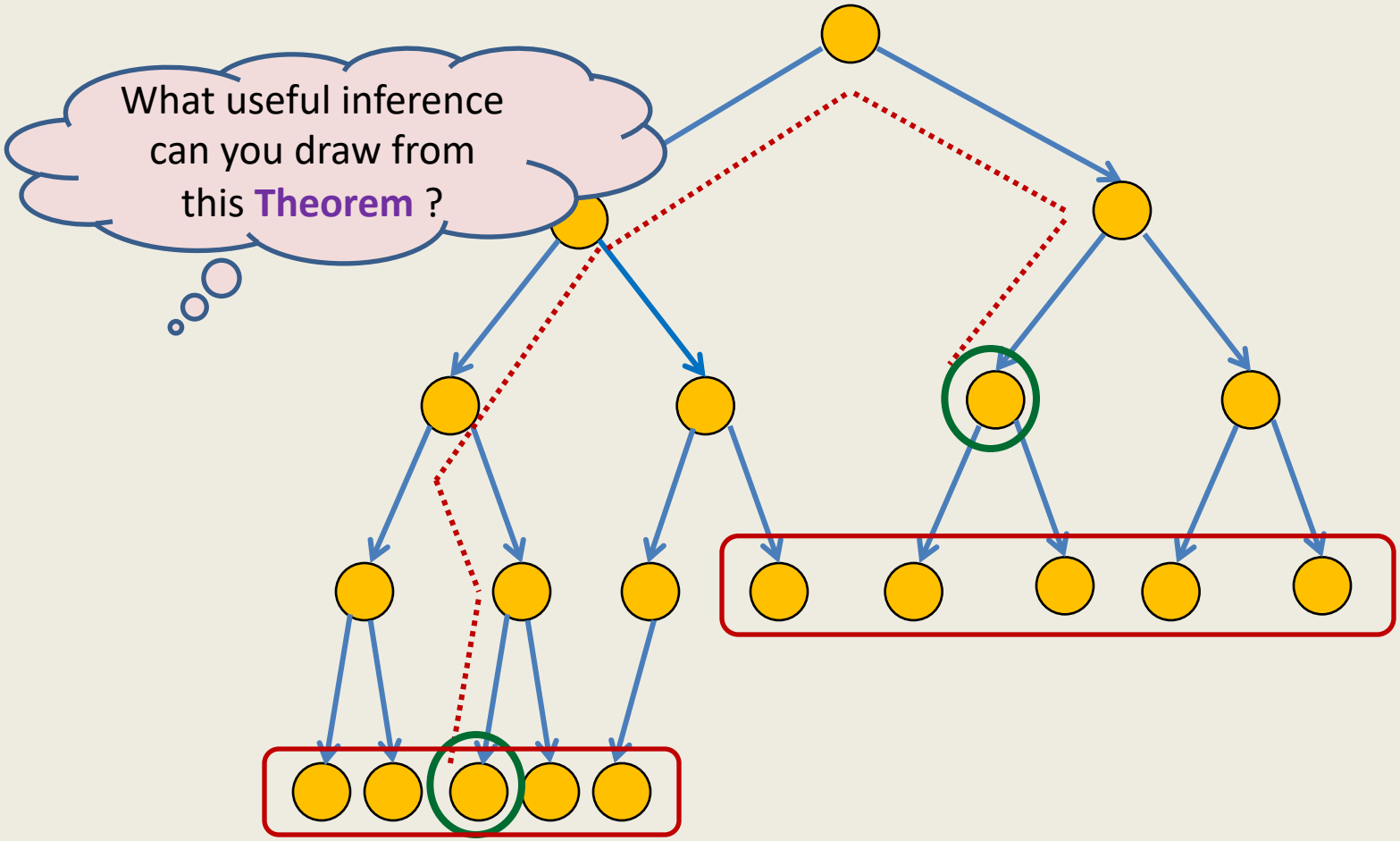
Trivial solution:

(Building the Binary heap incrementally**)**

```
CreateHeap(H);
For( i = 0 to n − 1 )
    Insert(x_i,H);
```



Building a Binary heap incrementally



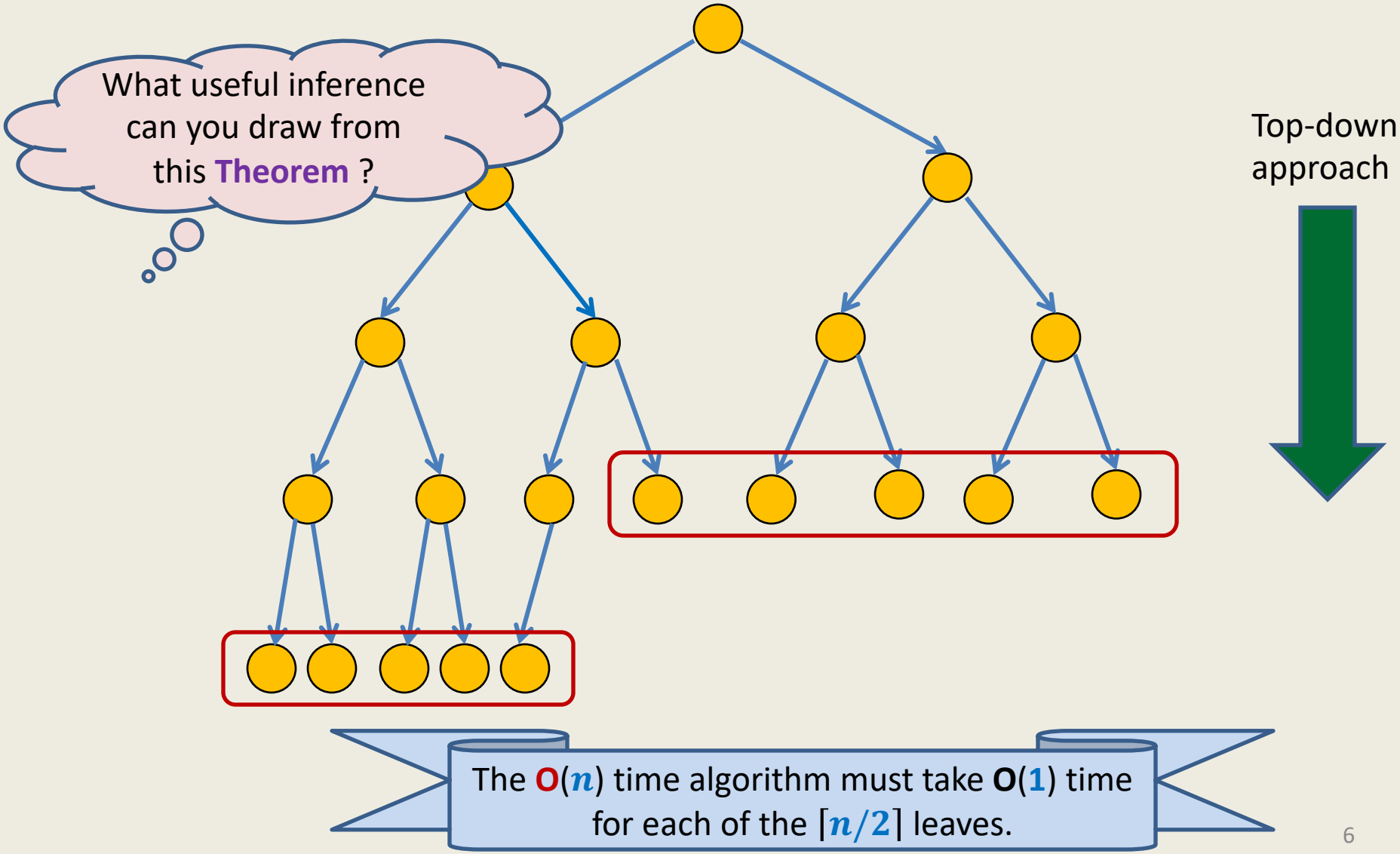
Top-down approach

The time complexity for inserting a leaf node = **O**(log n)

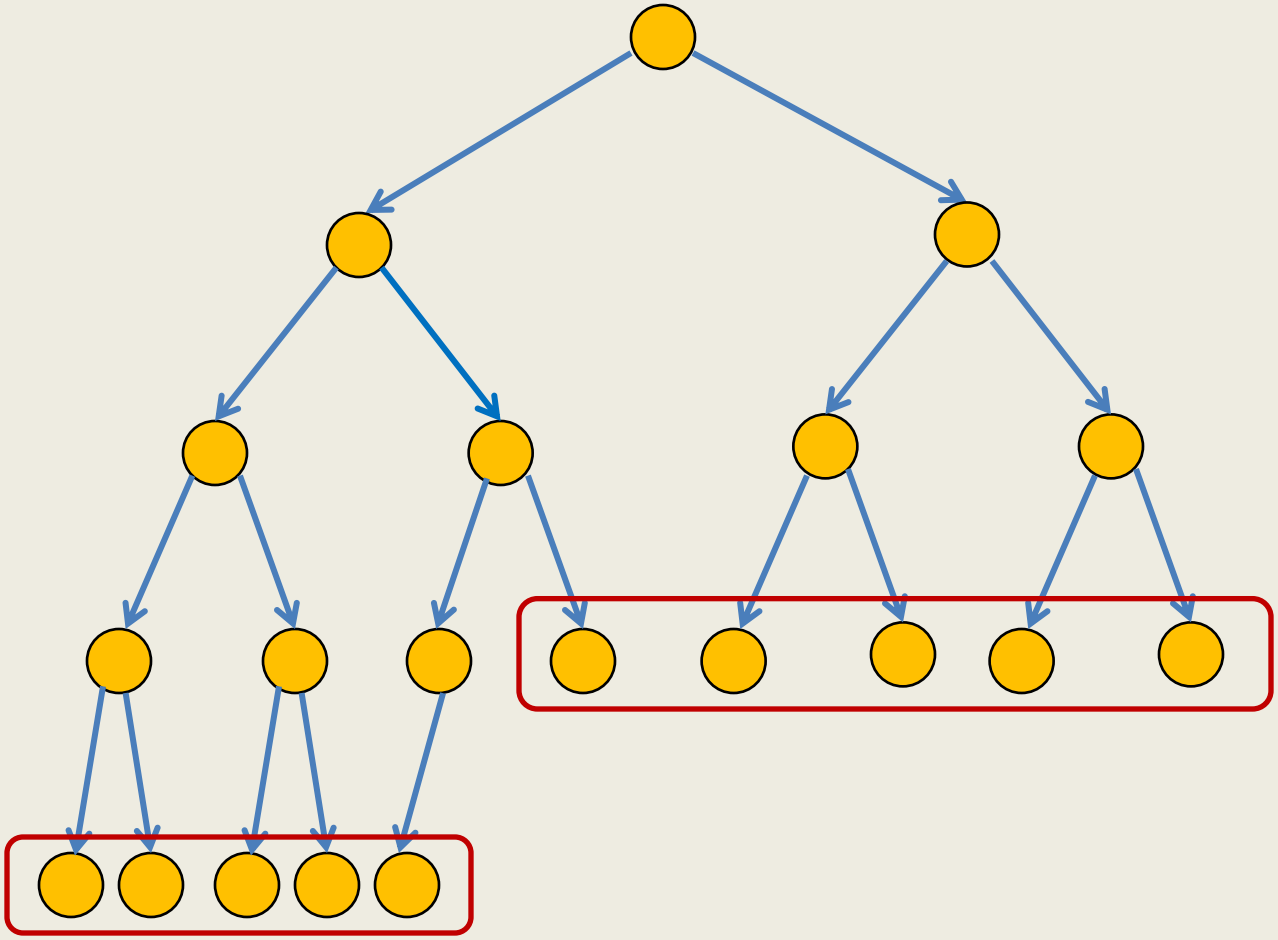
leaf nodes = $\lceil n/2 \rceil$,

➔ **Theorem**: Time complexity of building a binary heap incrementally is **O**(n log n).

Building a Binary heap incrementally

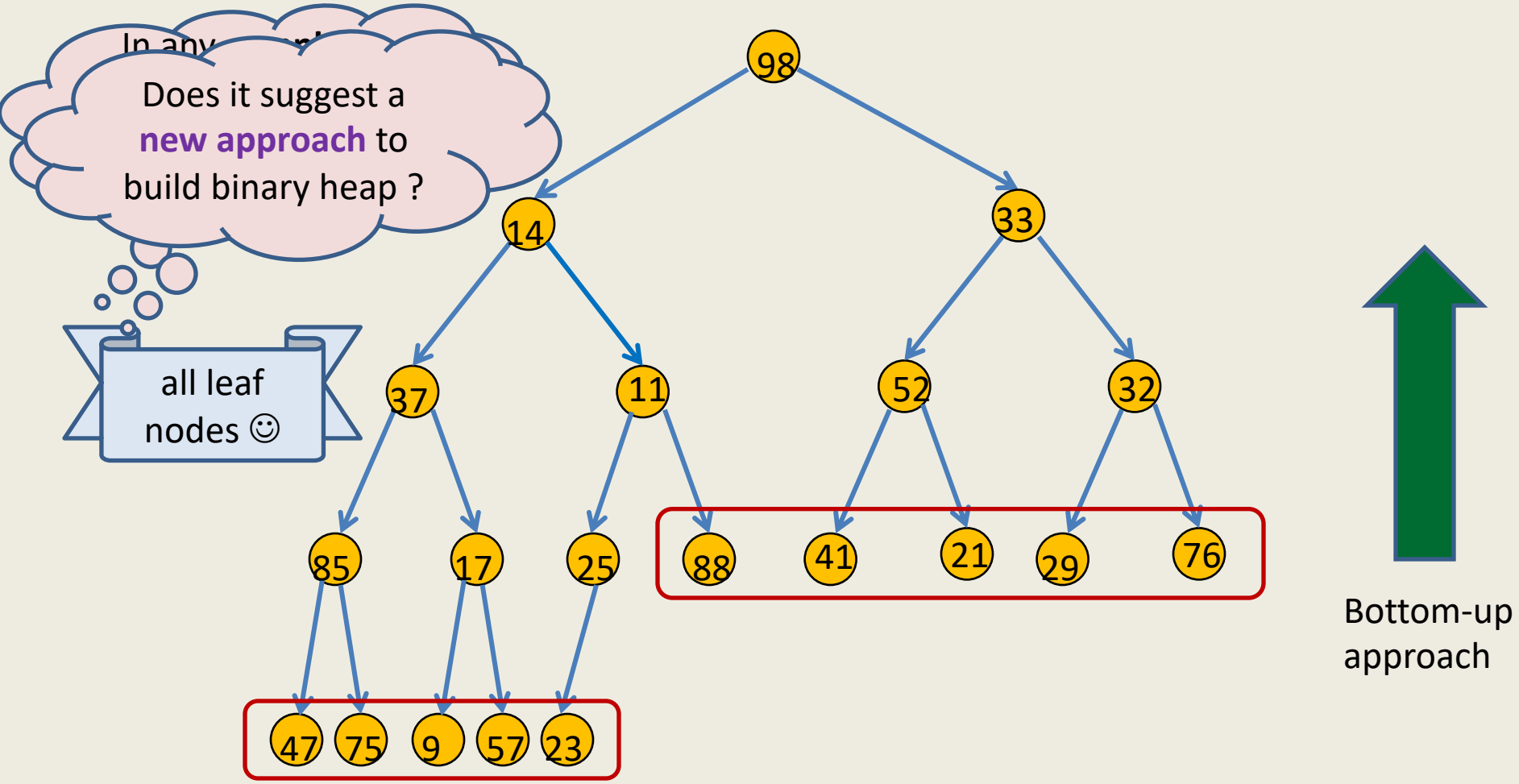


Building a Binary heap incrementally



Top-down approach

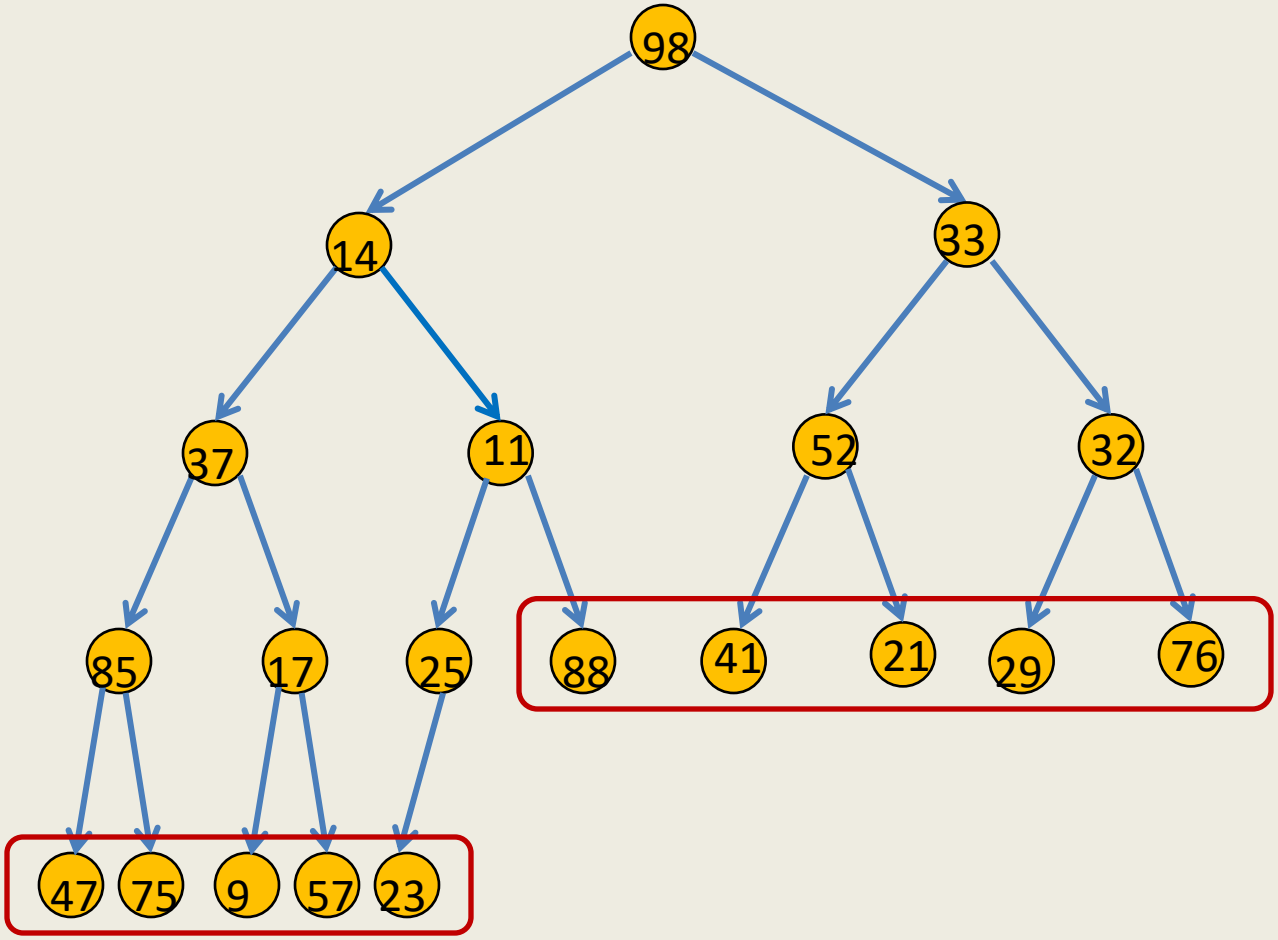
Think of alternate approach for building a binary heap



heap property: *"Every **node** stores value <u>smaller</u> than its **children**"*

We just need to ensure this property at each node.

Think of alternate approach for building a binary heap



Bottom-up approach

heap property: *"Every **node** stores value <u>smaller</u> than its **children**"*

We just need to ensure this property at each node.

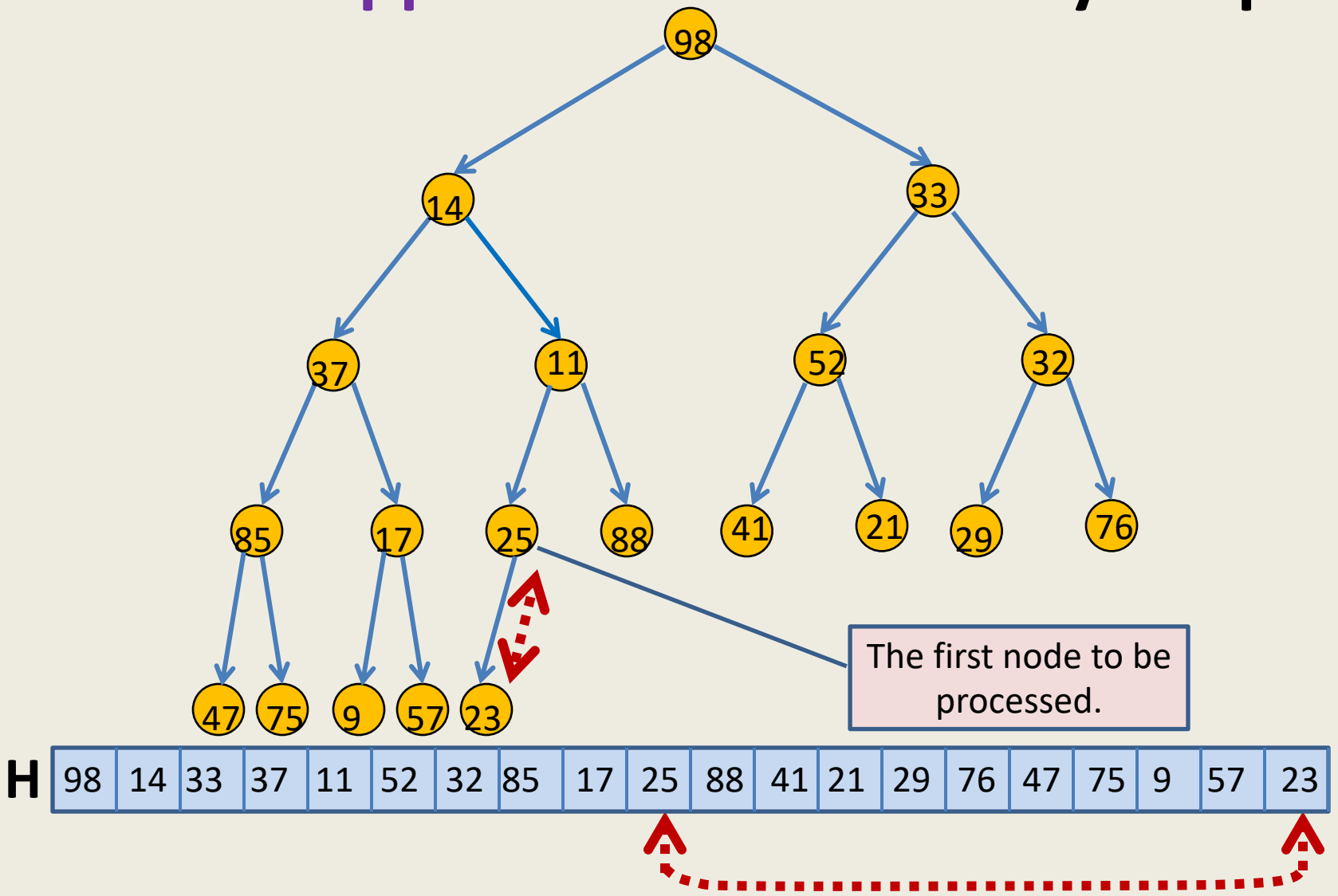
A new approach to build binary heap

1. Just copy the given n elements $\{x_0, \ldots, x_{n-1}\}$ into an array **H**.

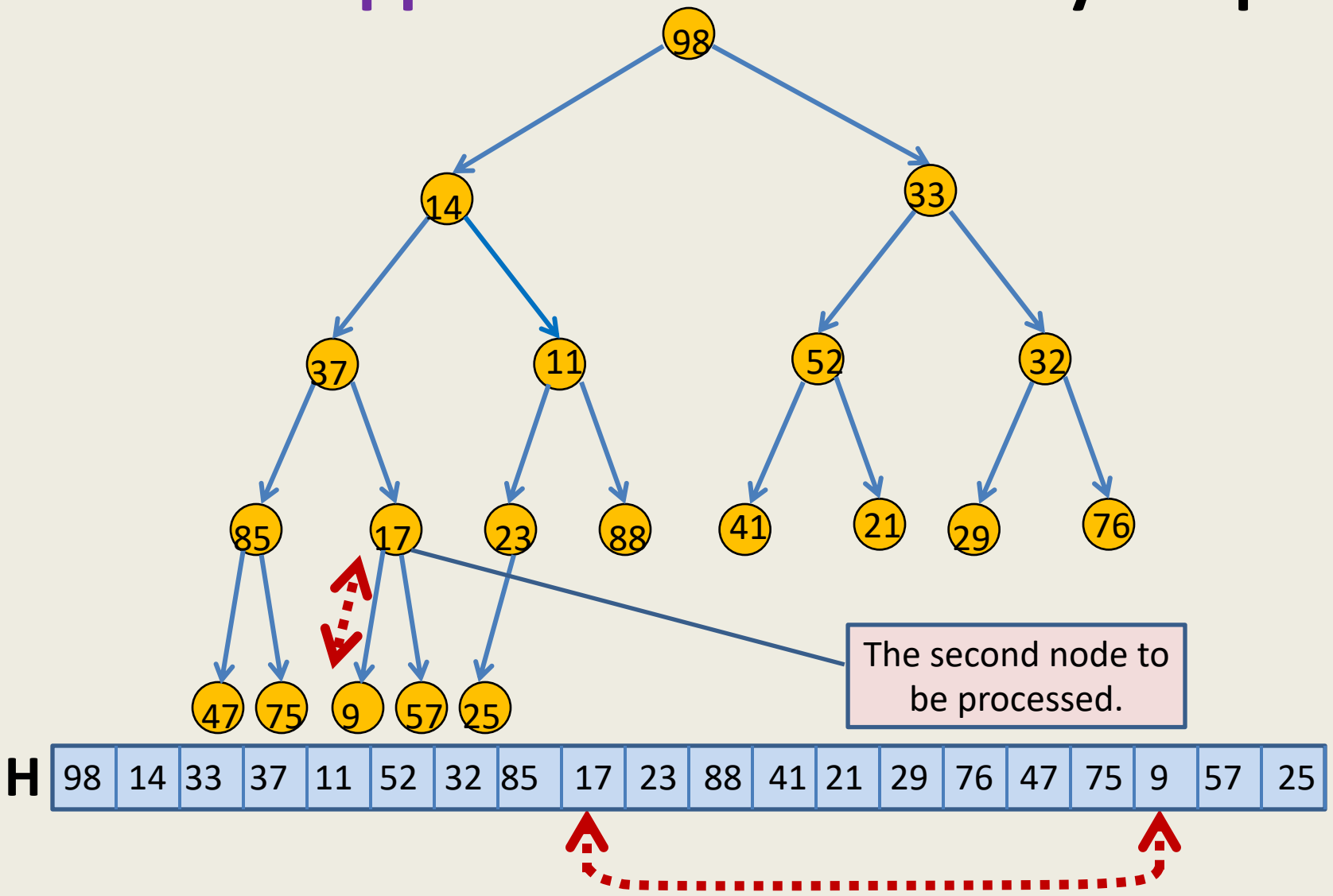
2. The **heap property** holds for all the leaf nodes in the corresponding complete binary tree.

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.

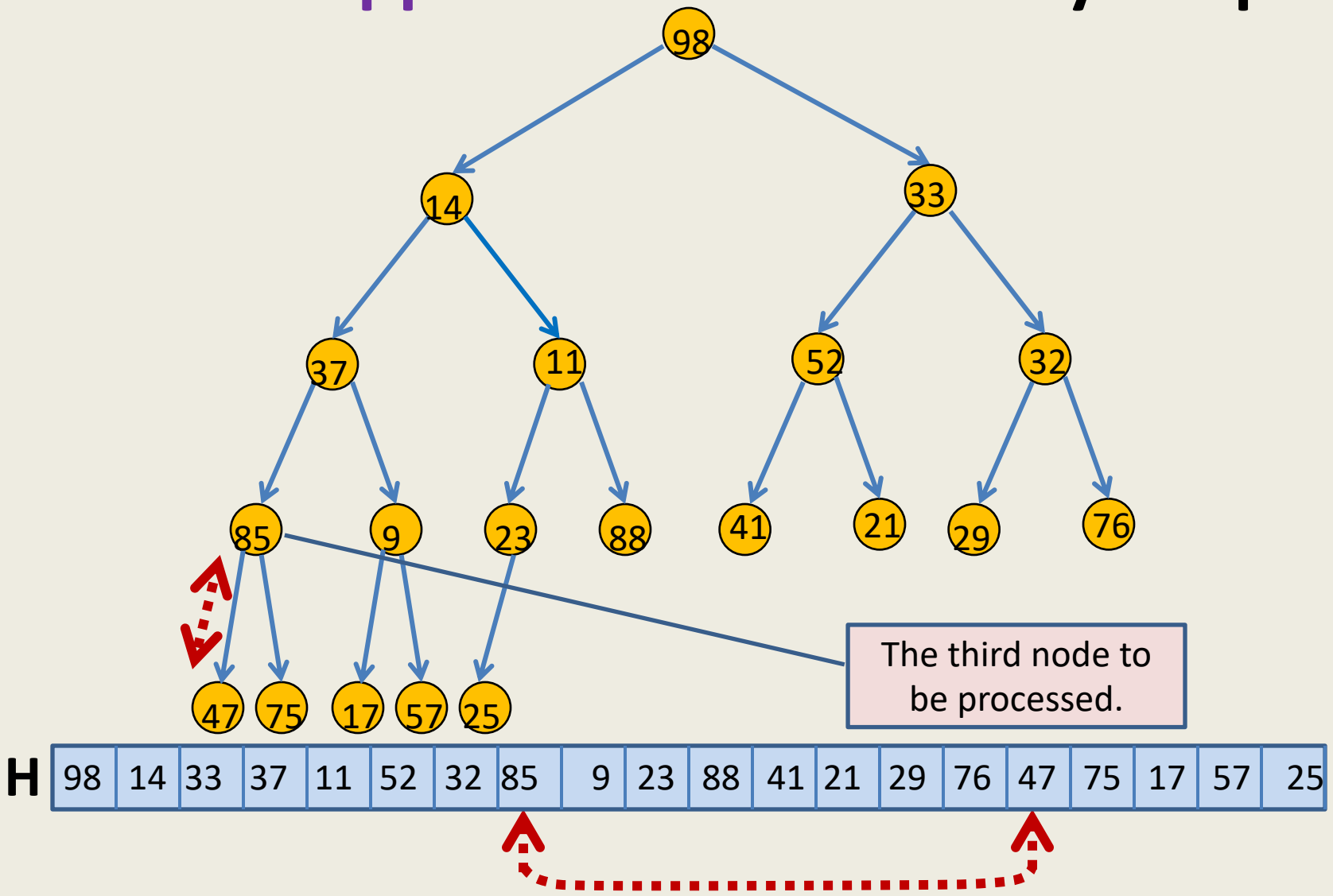
A new approach to build binary heap



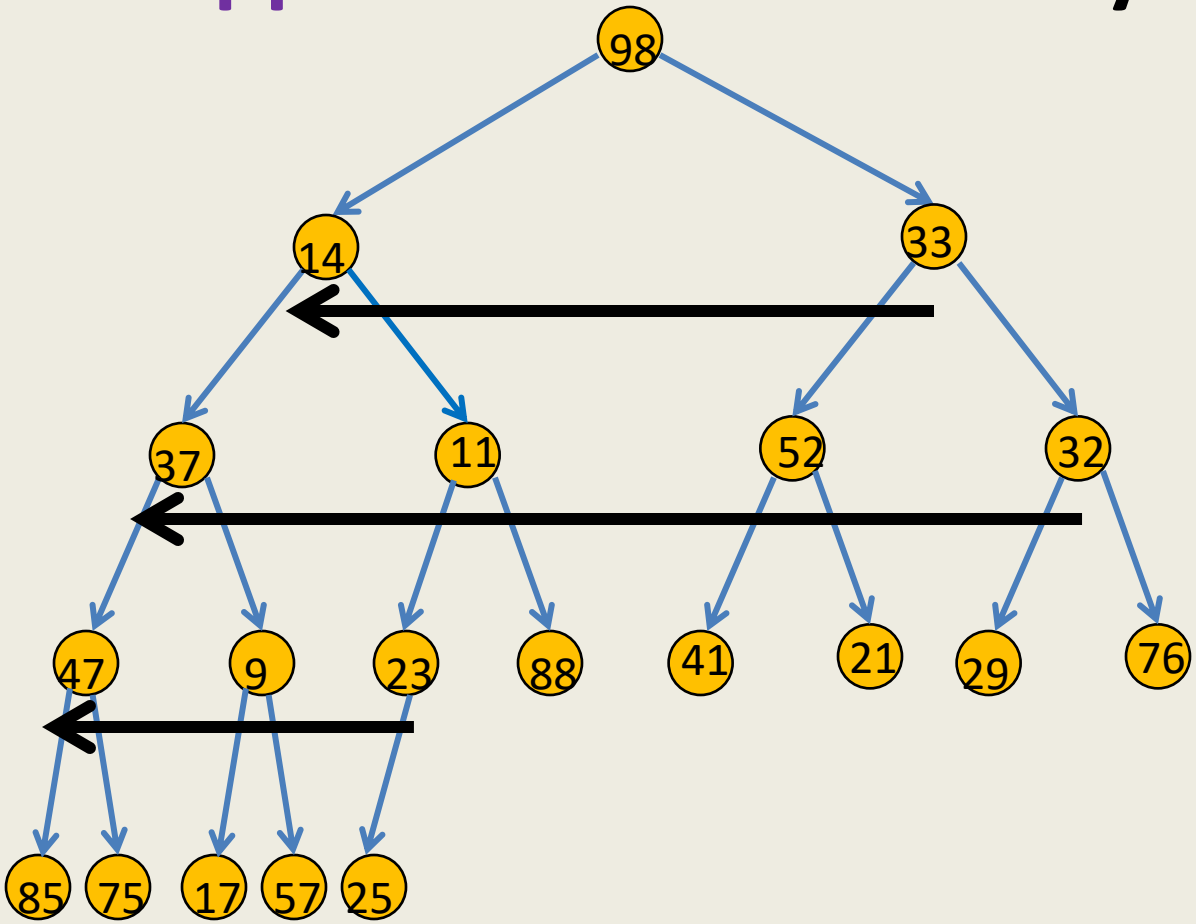
A new approach to build binary heap



A new approach to build binary heap

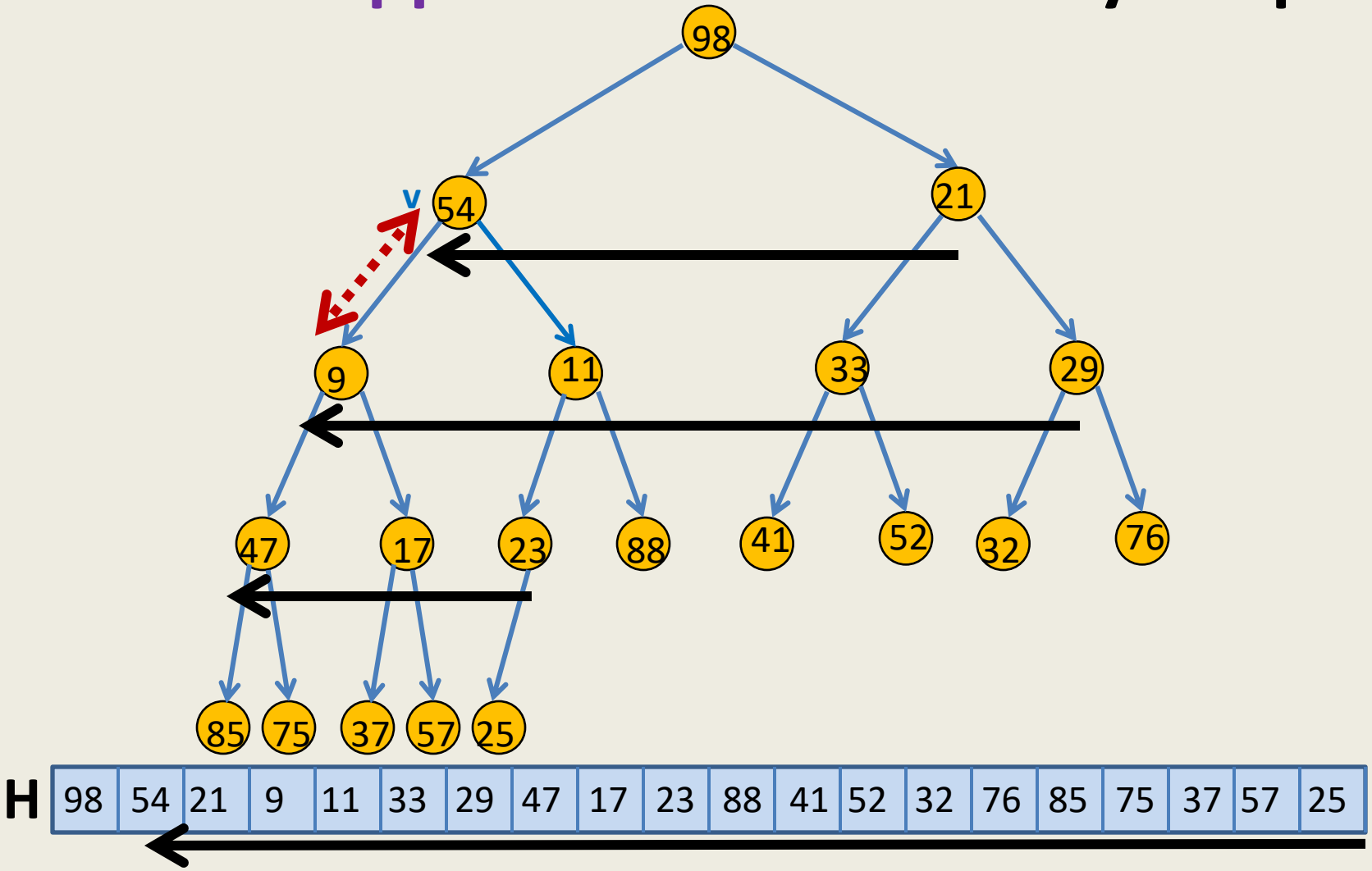


A new approach to build binary heap

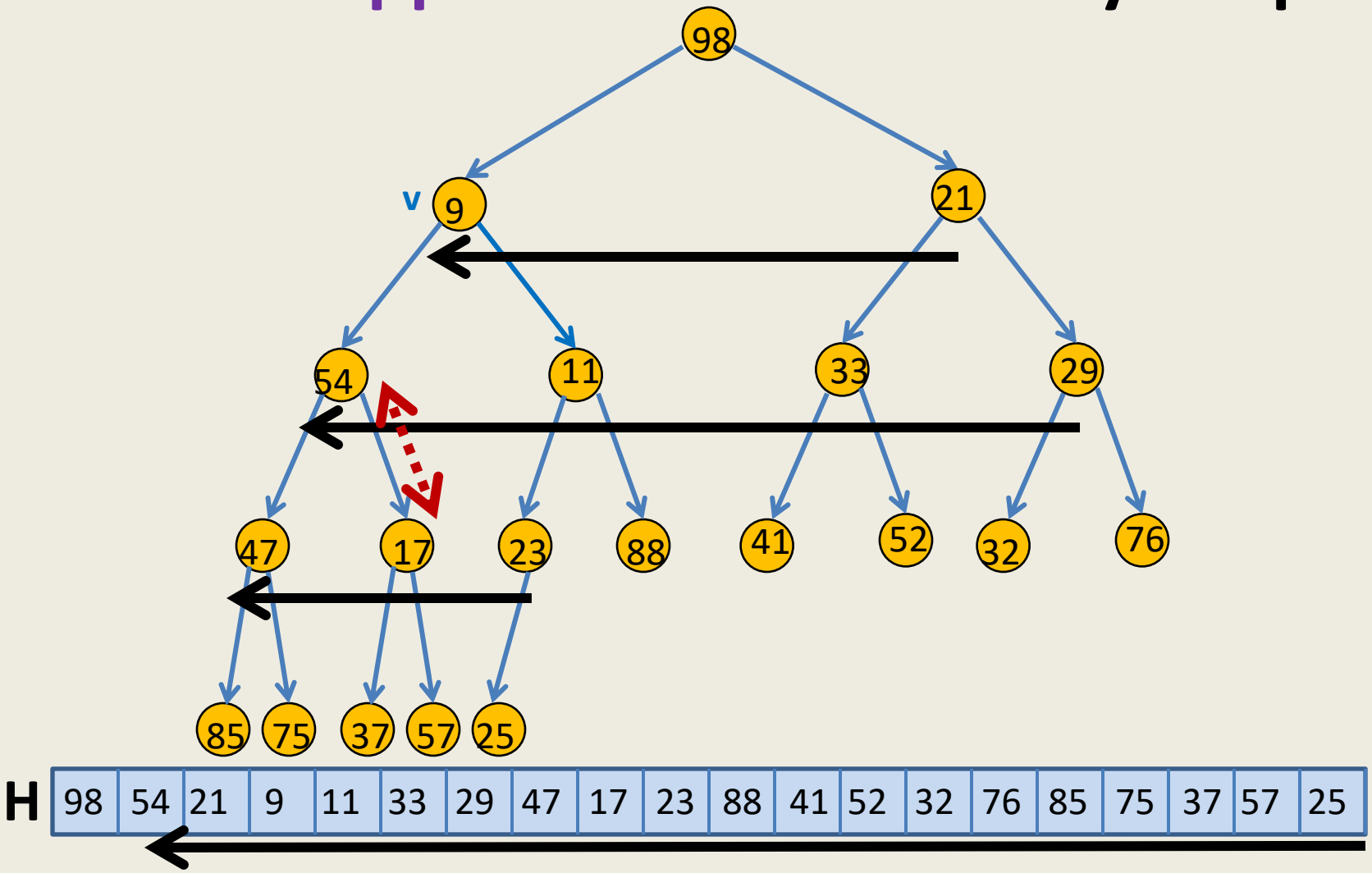


H	98	14	33	37	11	52	32	47	9	23	88	41	21	29	76	85	75	17	57	25

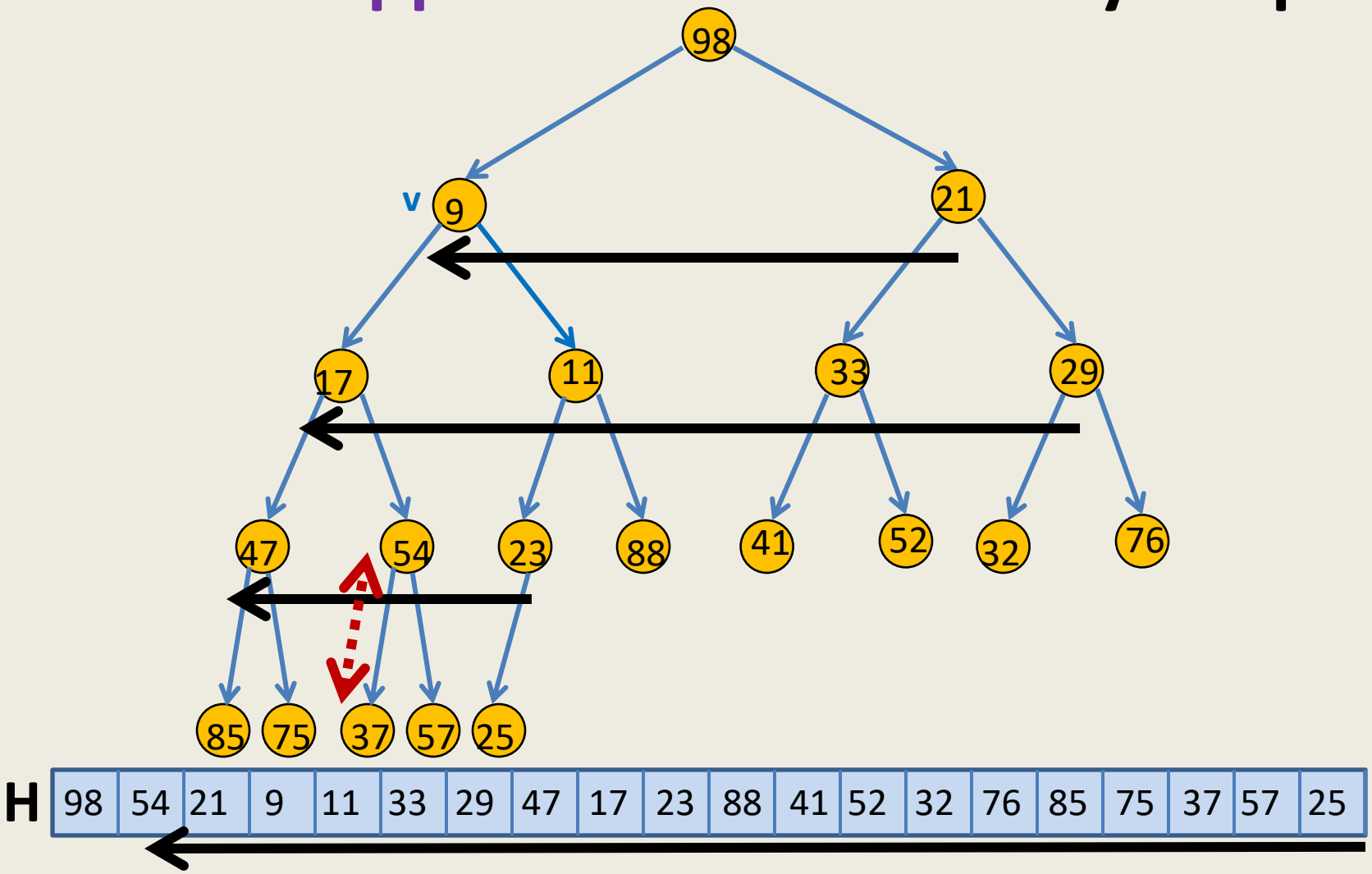
A new approach to build binary heap



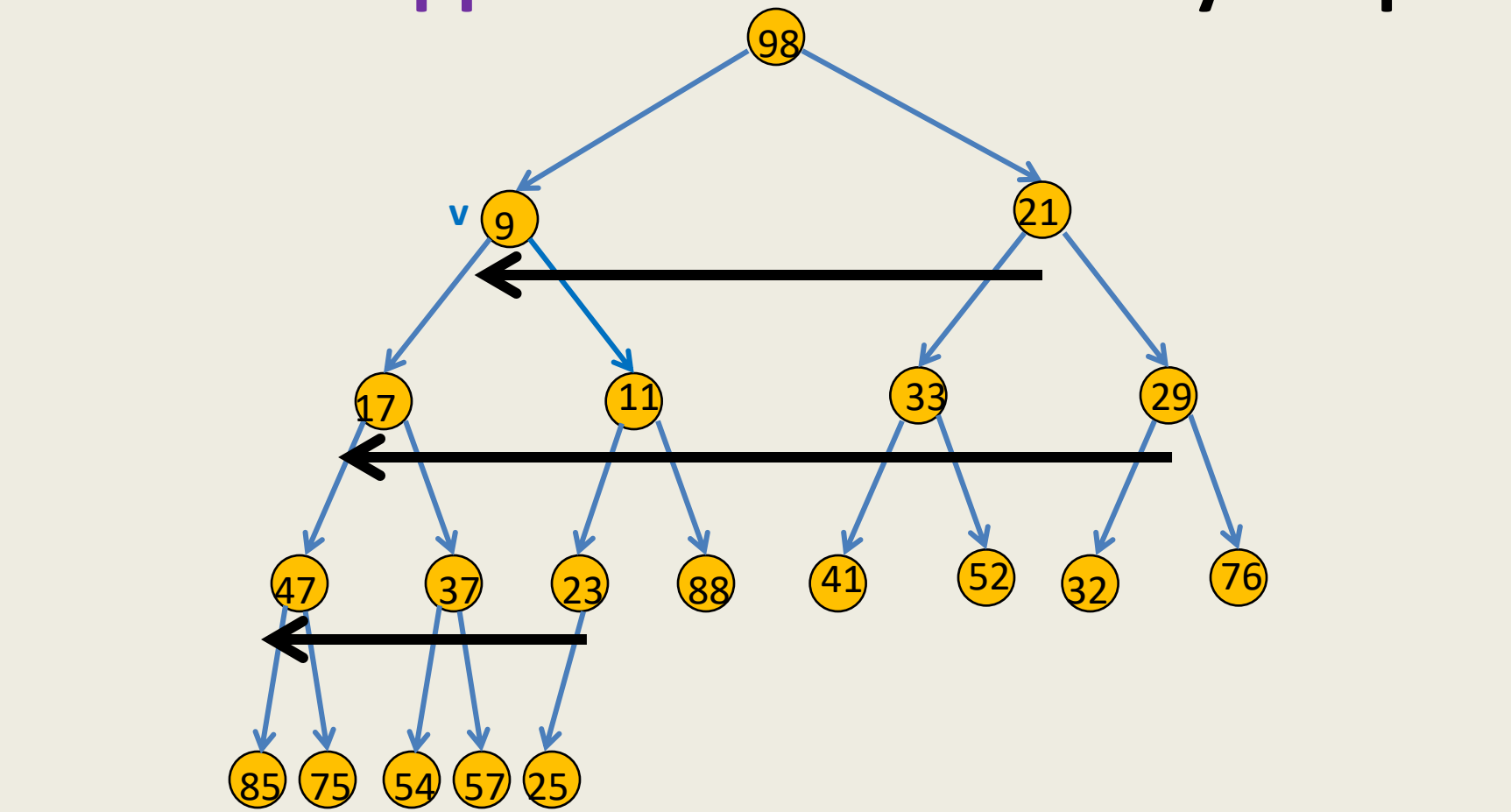
A new approach to build binary heap



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 ← 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 !

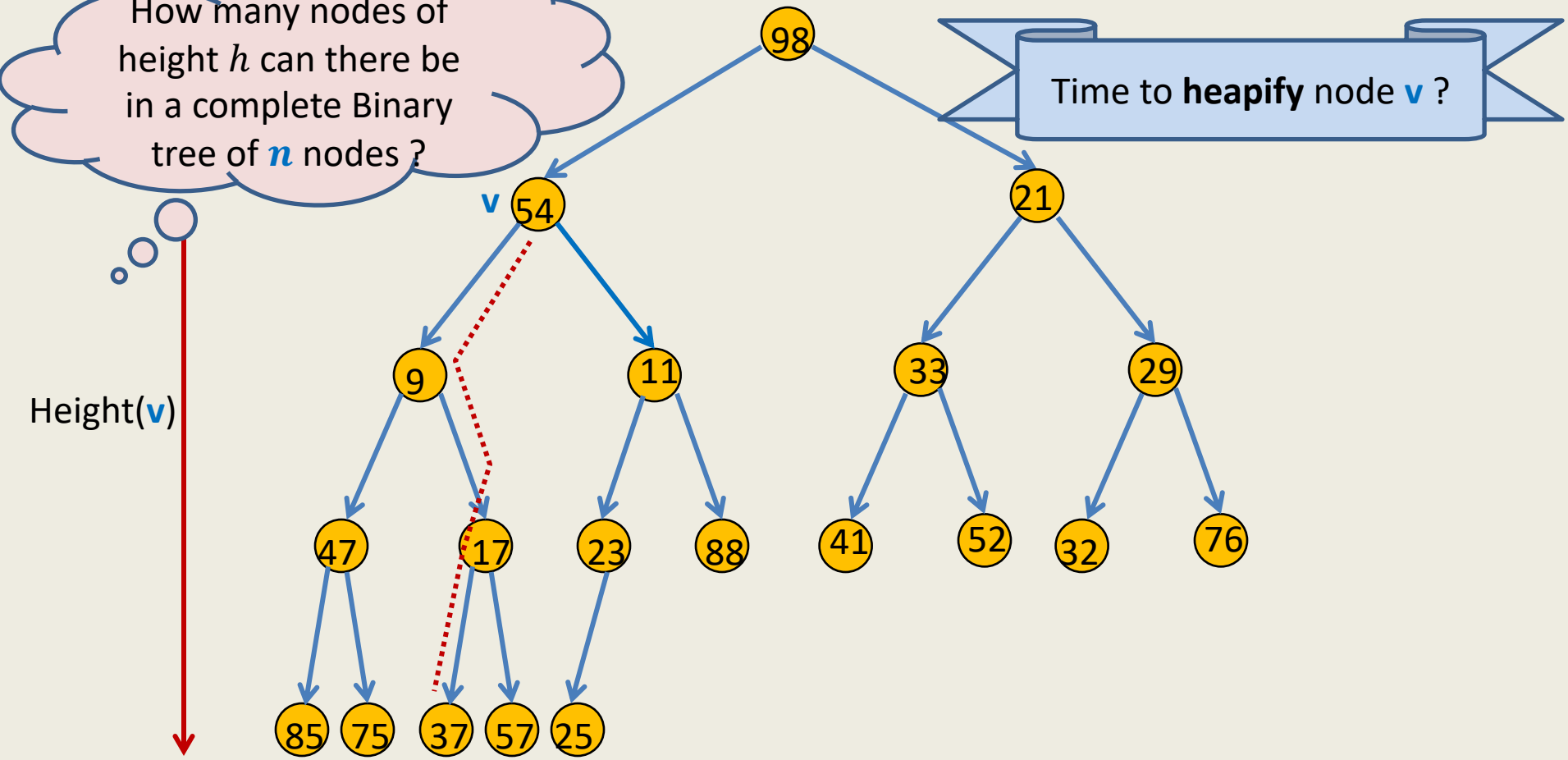
Heapify(i,H)

```
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** ?



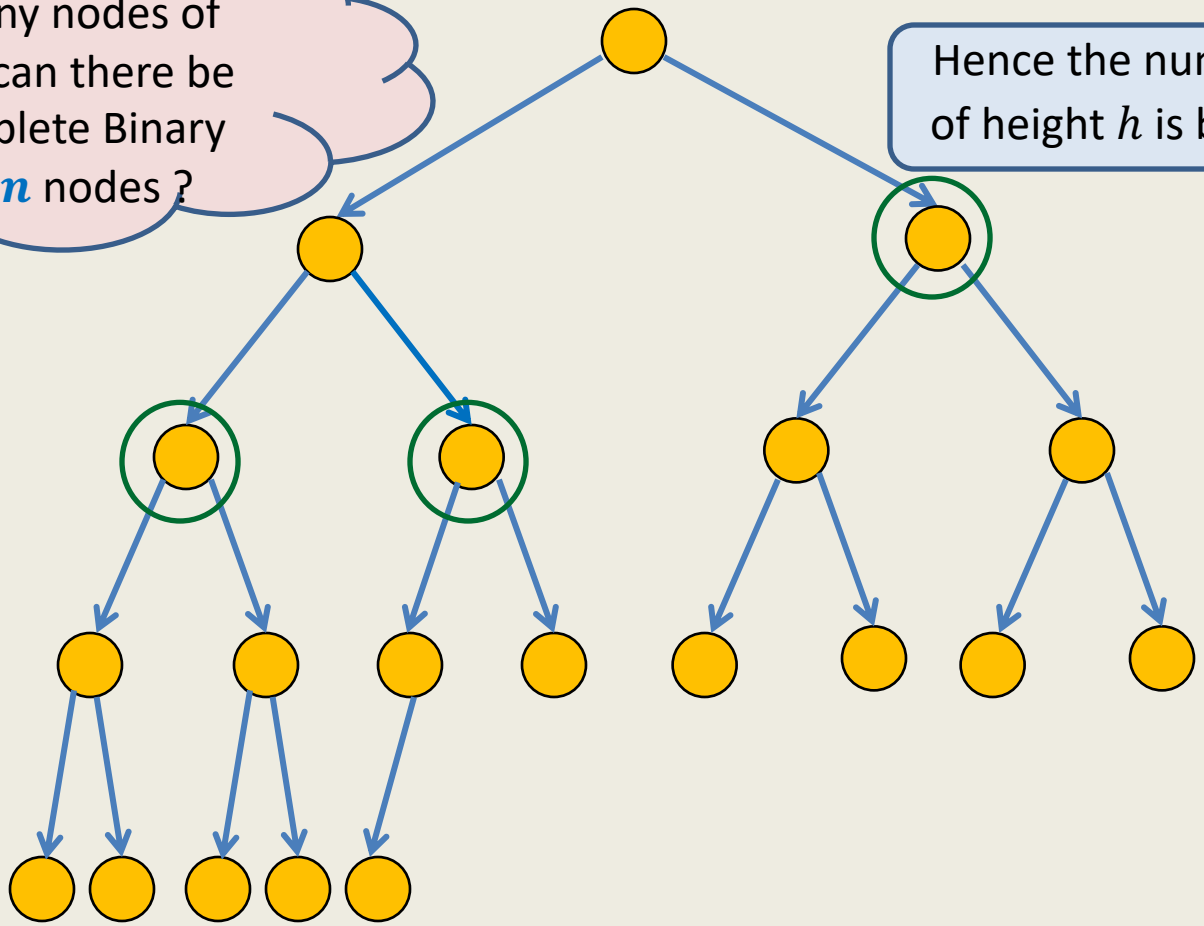
H	98	54	21	9	11	33	29	47	17	23	88	41	52	32	76	85	75	37	57	25

Time complexity of algorithm = $\sum_{h} O(h) \cdot \underbrace{N(h)}$

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 $\frac{n}{2^h}$.

Hence Time complexity to build the heap = $\sum_{h=1}^{\log n} \frac{n}{2^h} O(h)$

$= n \; c \; \sum_{i=1}^{\log n} \frac{h}{2^h}$

$= O(n)$

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** ← 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 **S** = $\prec x_0, \ldots, x_{n-1} \succ$

- Can be viewed as a mapping from [0, n].
- Order does matter.

Problem 1

Multi-increment

Problem 1

Given an initial sequence **S** = $\prec x_0, \ldots, x_{n-1} \succ$ 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** = $\prec$ 14, 12, 23, 12, 111, 51, 321, -40 $\succ$

After **Multi-Increment**(2,6,10), **S** becomes

$\prec$ 14, 12, **33**, **22**, **121**, **61**, **331**, -40 $\succ$

After **Multi-Increment**(0,4,25), **S** becomes

$\prec$ **39**, **37**, **58**, **47**, **146**, 61, 331, -40 $\succ$

After **Multi-Increment**(2,5,31), **S** becomes

$\prec$ 39, 37, **89**, **78**, **177**, **92**, 331, -40 $\succ$

Problem 1

Given an initial sequence **S** = $\prec x_0, \ldots, x_{n-1} \succ$ 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 ≤ k ≤ j)        A[k] ← A[k] + Δ;
}
```

O($j - i$) = **O**(n)

```
ReportElement(i){      return A[i]    }
```

O(1)

Problem 1

Given an initial sequence **S** = $\prec x_0, \ldots, x_{n-1} \succ$ 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** in **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** = $\prec x_0, \ldots, x_{n-1} \succ$ 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** = $\prec x_0, \ldots, x_{n-1} \succ$ 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** = $\prec$ 14, 12, 23, 12, 111, 51, 321, -40 $\succ$

After **Multi-Increment**(2,6,10), **S** becomes

$\prec$ 14, 12, **33**, **22**, **121**, **61**, **331**, -40 $\succ$

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 = $\{1,2,3, \ldots, 2^n\}$

Question:

Can we have a small set **X**⊂**S** of numbers s.t.

Every number from **S** can be expressed as a **sum** of **a few** numbers from **X** ?

Answer: **X** = $\{1,2,4,8, \ldots, 2^n\}$

|**X**| = n

$$1\,0\,0\,0\,0\,0\,0\,0\;\;0\,0\,0$$
$$1\,0\,0\,0\,0\,0\,0$$
$$1\,0\,0\,0\,0$$
$$1\,0\,0\,0$$
$$1$$
$$1\,0\,0\,1\,0\,1\,1\;\;0\,0\,1$$

If it is too trivial, try to answer the problem of next slide.☺

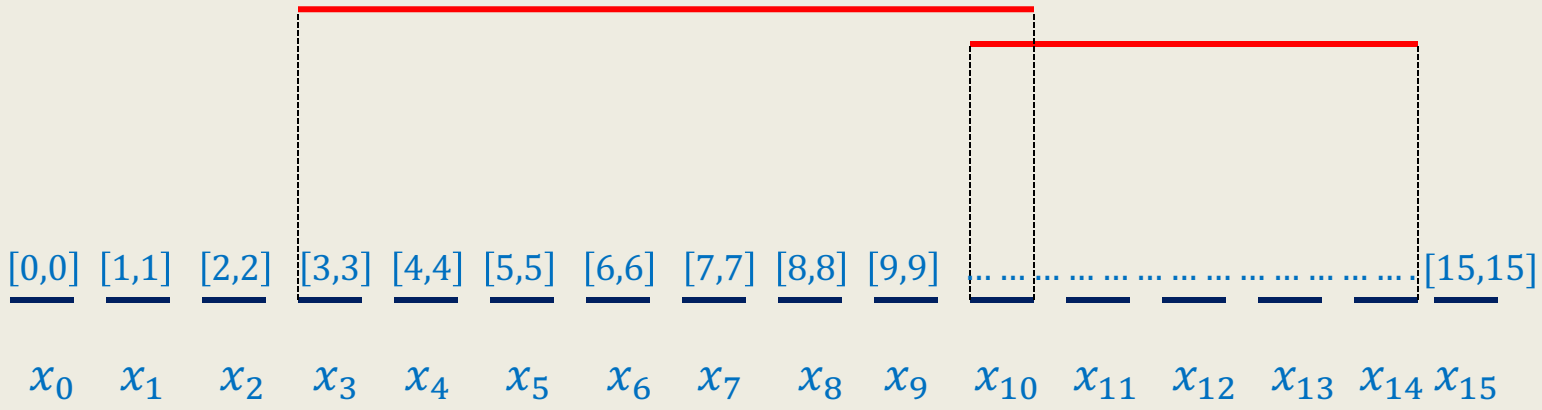
Extension to intervals

S = {[***i***, ***j***], $0 \leq i \leq j < n$}

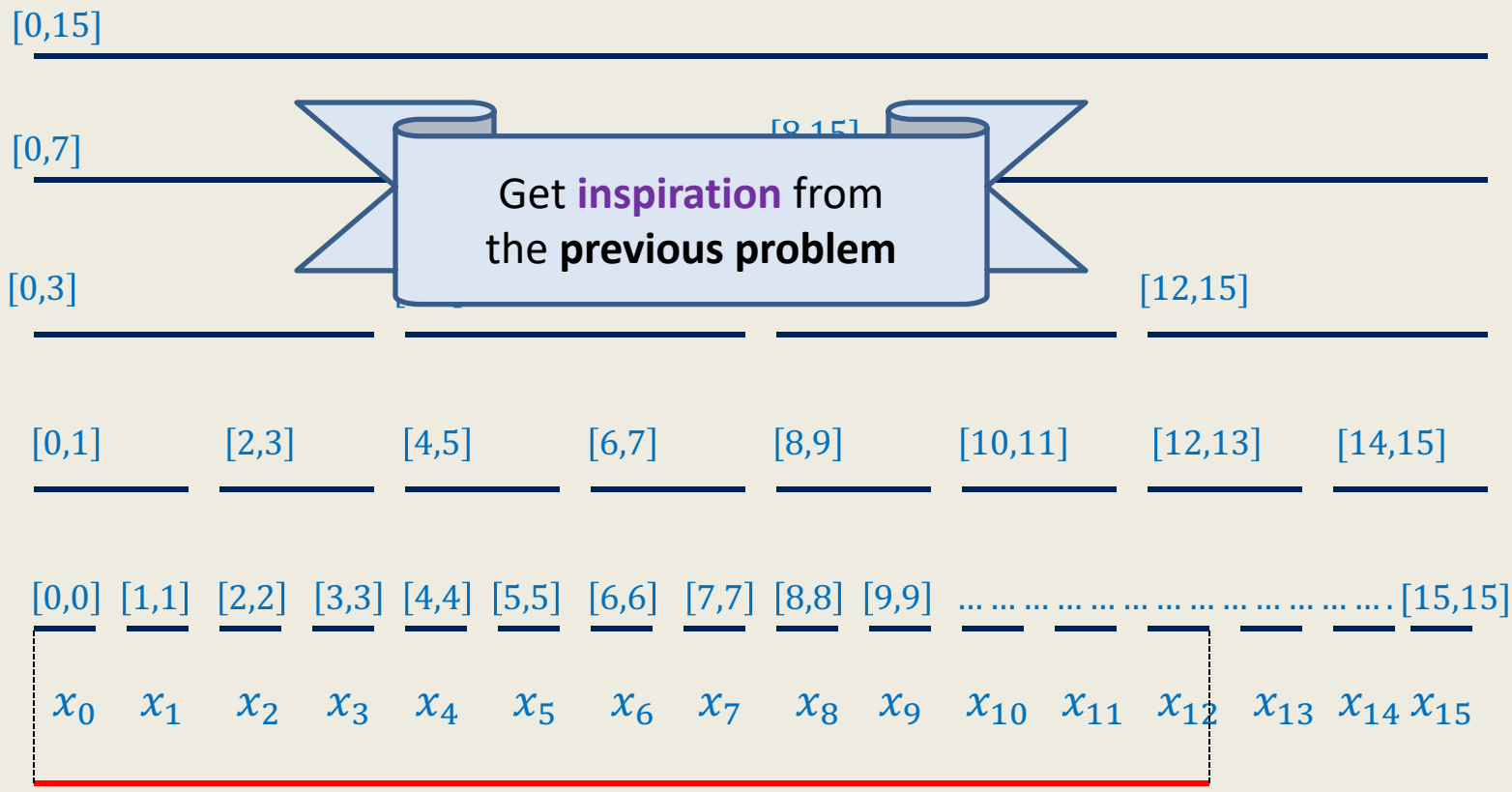
Question:

Can we have a small set **X**⊂**S** of **intervals** s.t.

every interval in **S** can be expressed as a **union** of a few **intervals** from **X** ?

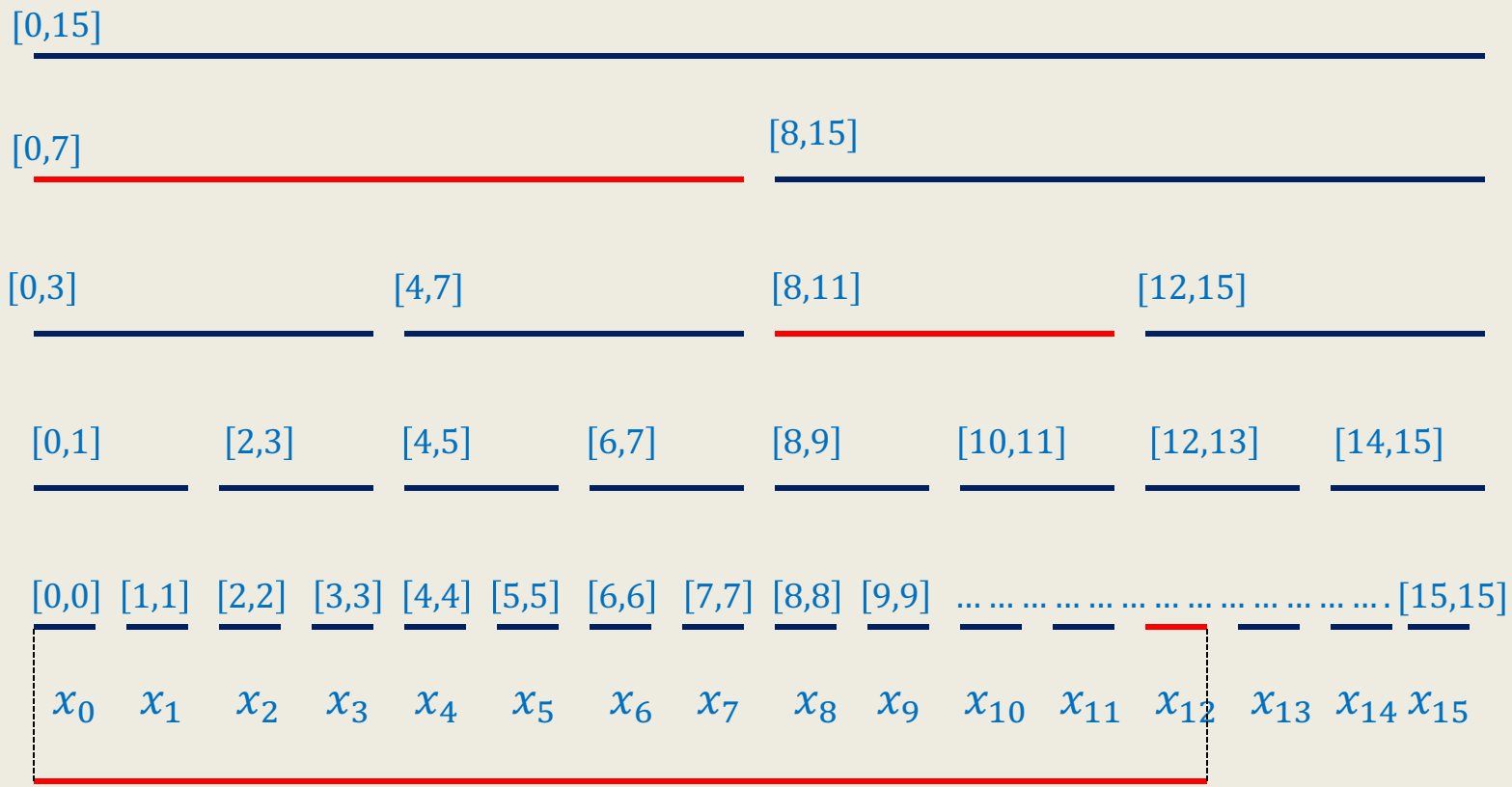


Extension to intervals



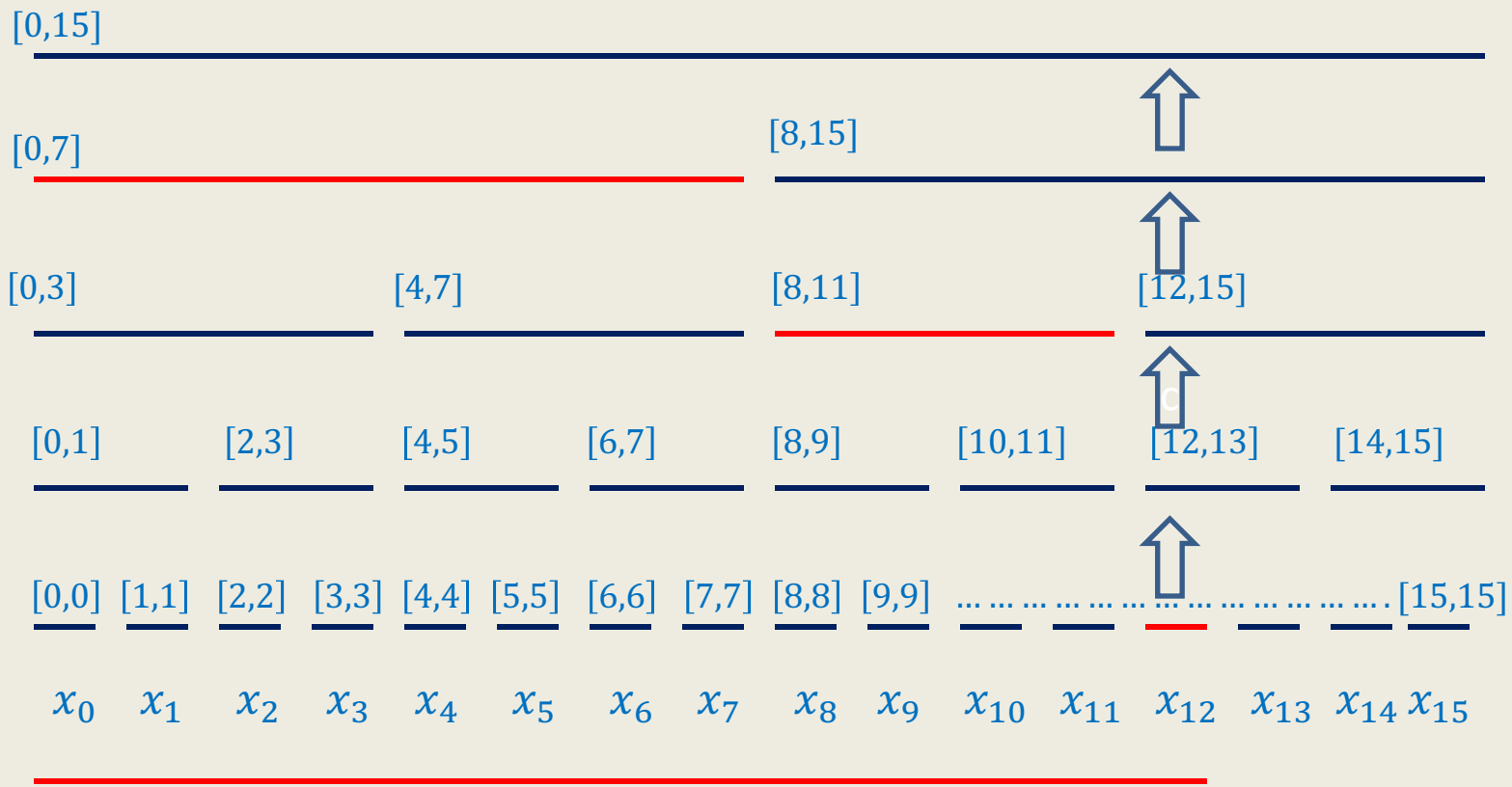
How to express [0, 12] ?

Extension to intervals



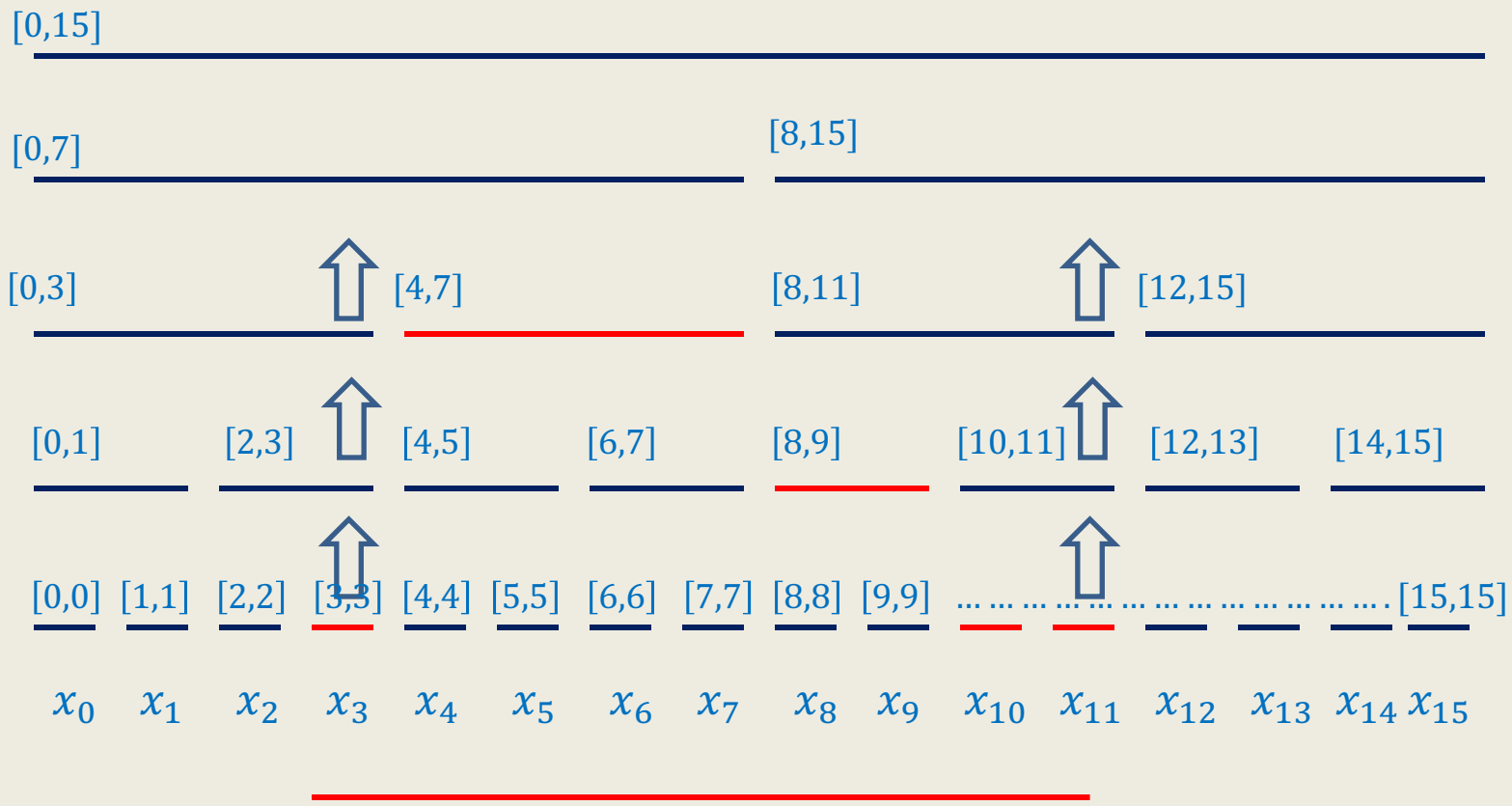
How to express [0, 12] ?

Extension to intervals



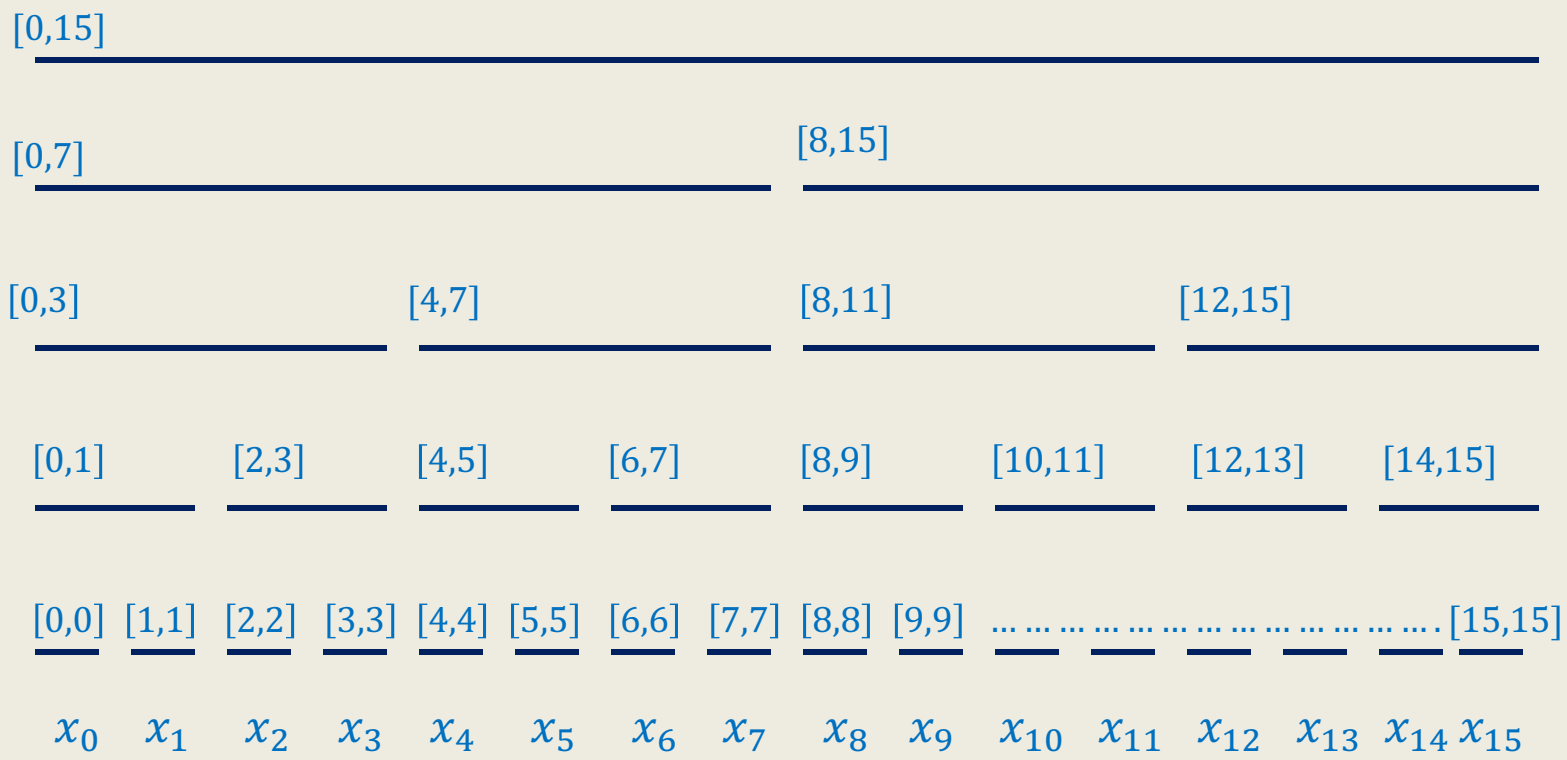
How to express [0, 12] ?

Extension to intervals



How to express [3, 11] ?

How to use this Observation to perform Multi-Increment(i, j, Δ) efficiently?

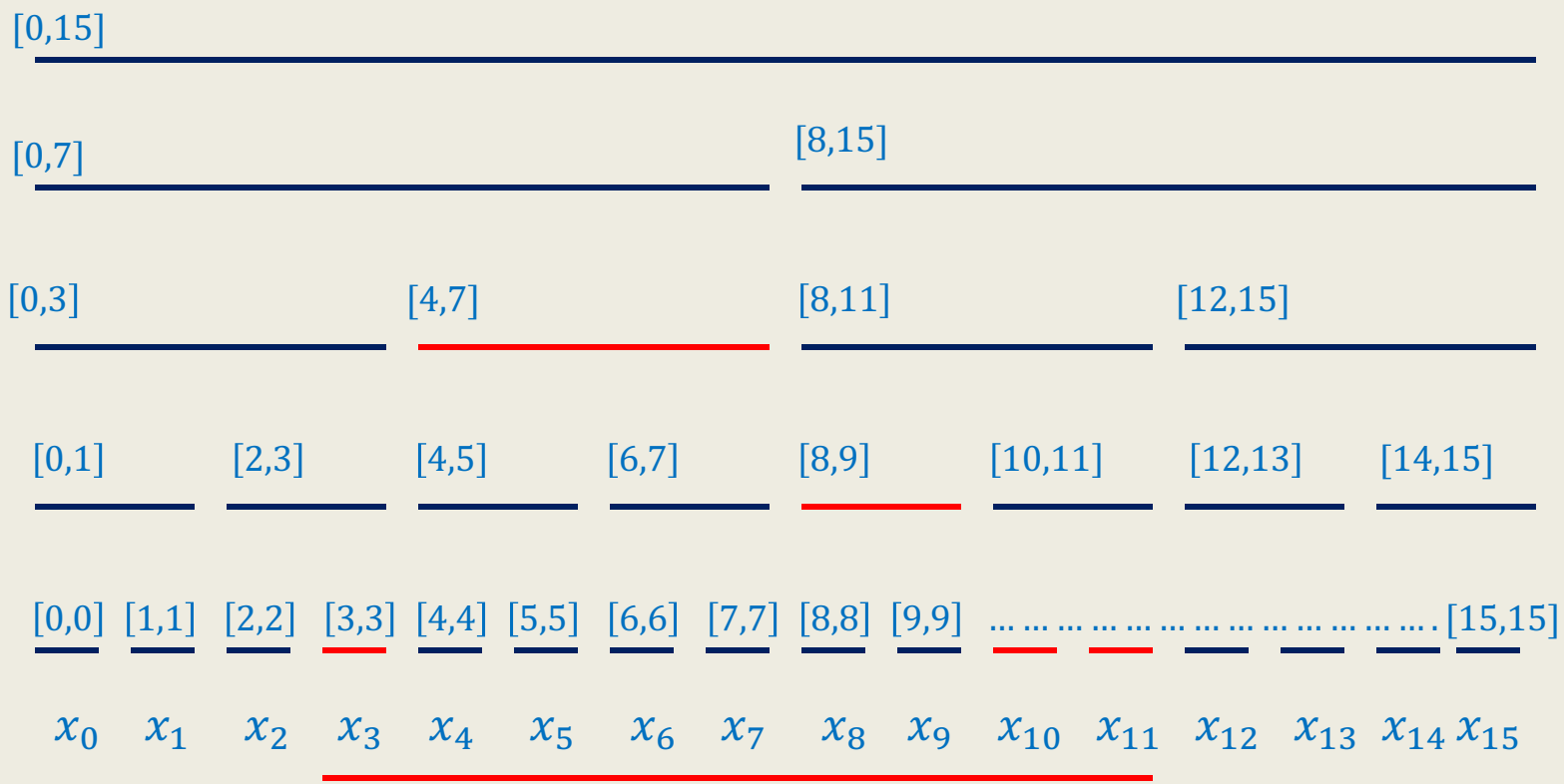


Observation:

There are $2n$ intervals such that

any interval [i, j] can be expressed as **union** of **O**(log n) basic intervals ☺

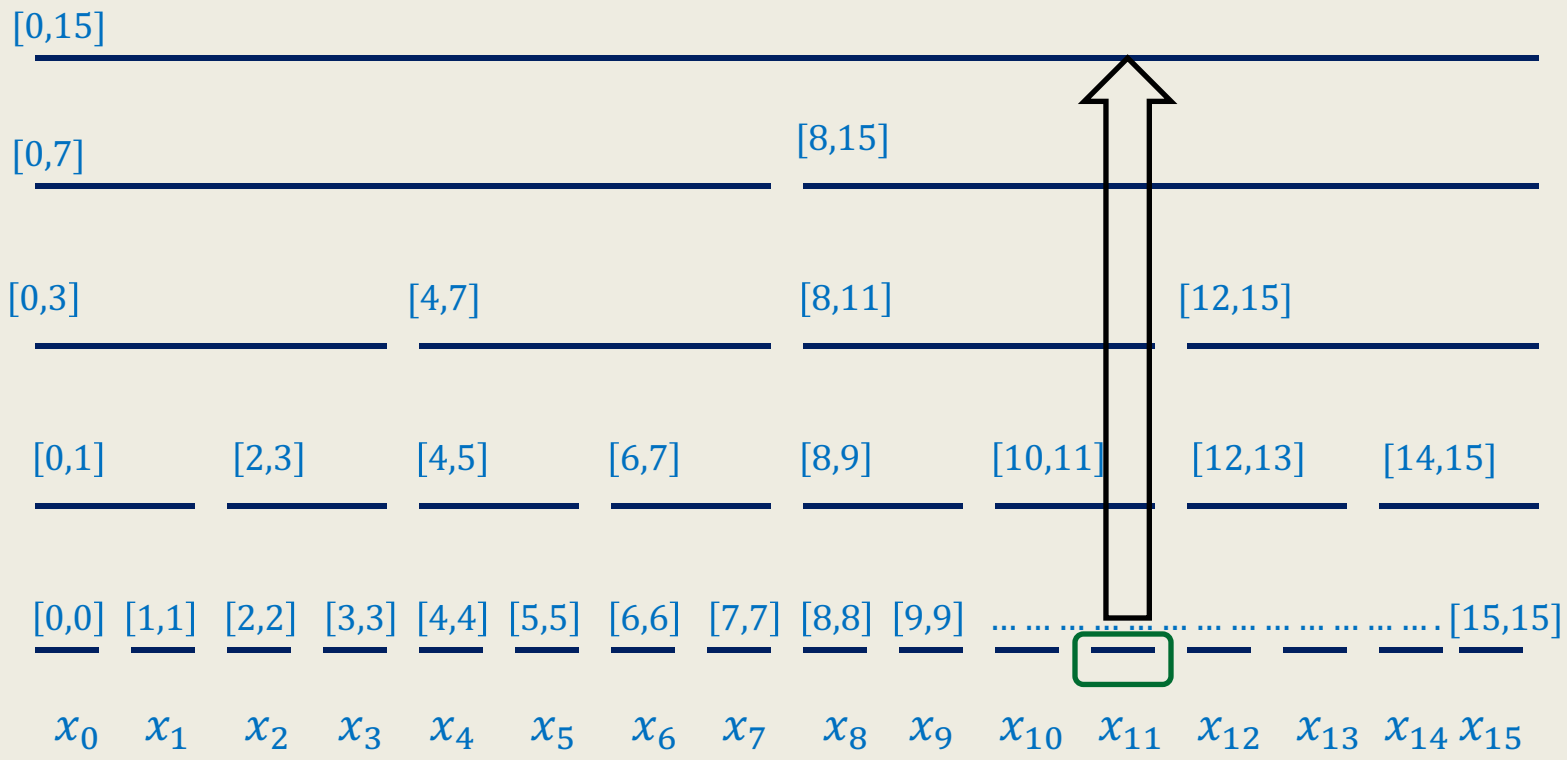
How to use this Observation to perform Multi-Increment(i, j, Δ) efficiently?



Maintain $2n$ intervals with a field **increment**

Multi-Increment(i, j, Δ) ➔ add Δ to **increment** field of its **O**(log n) intervals.

How to use this Observation to perform Multi-Increment(i, j, Δ) efficiently?

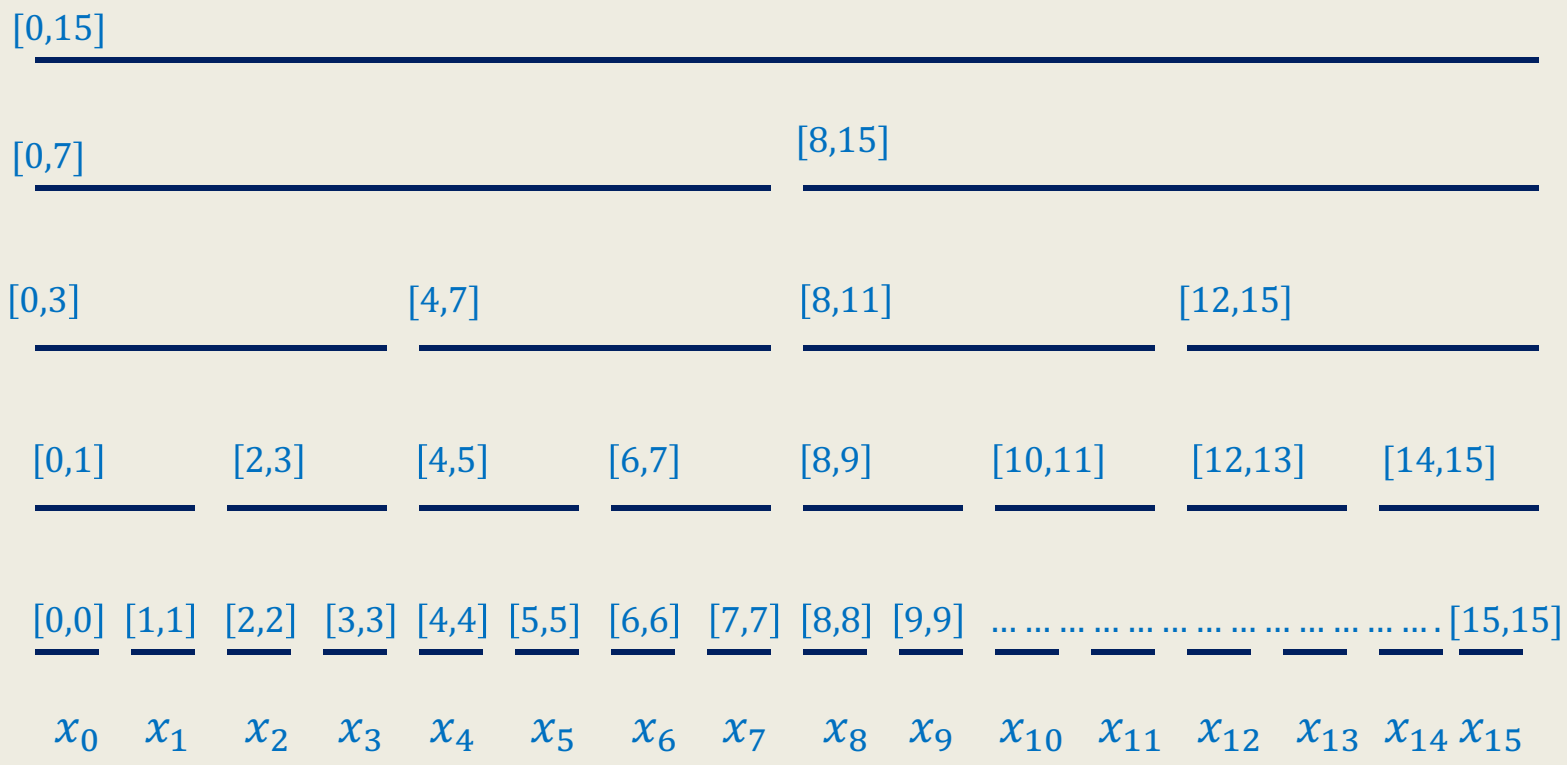


Maintain $2n$ intervals with a field **increment**

Multi-Increment(i, j, Δ) ➔ 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) ➔ 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.

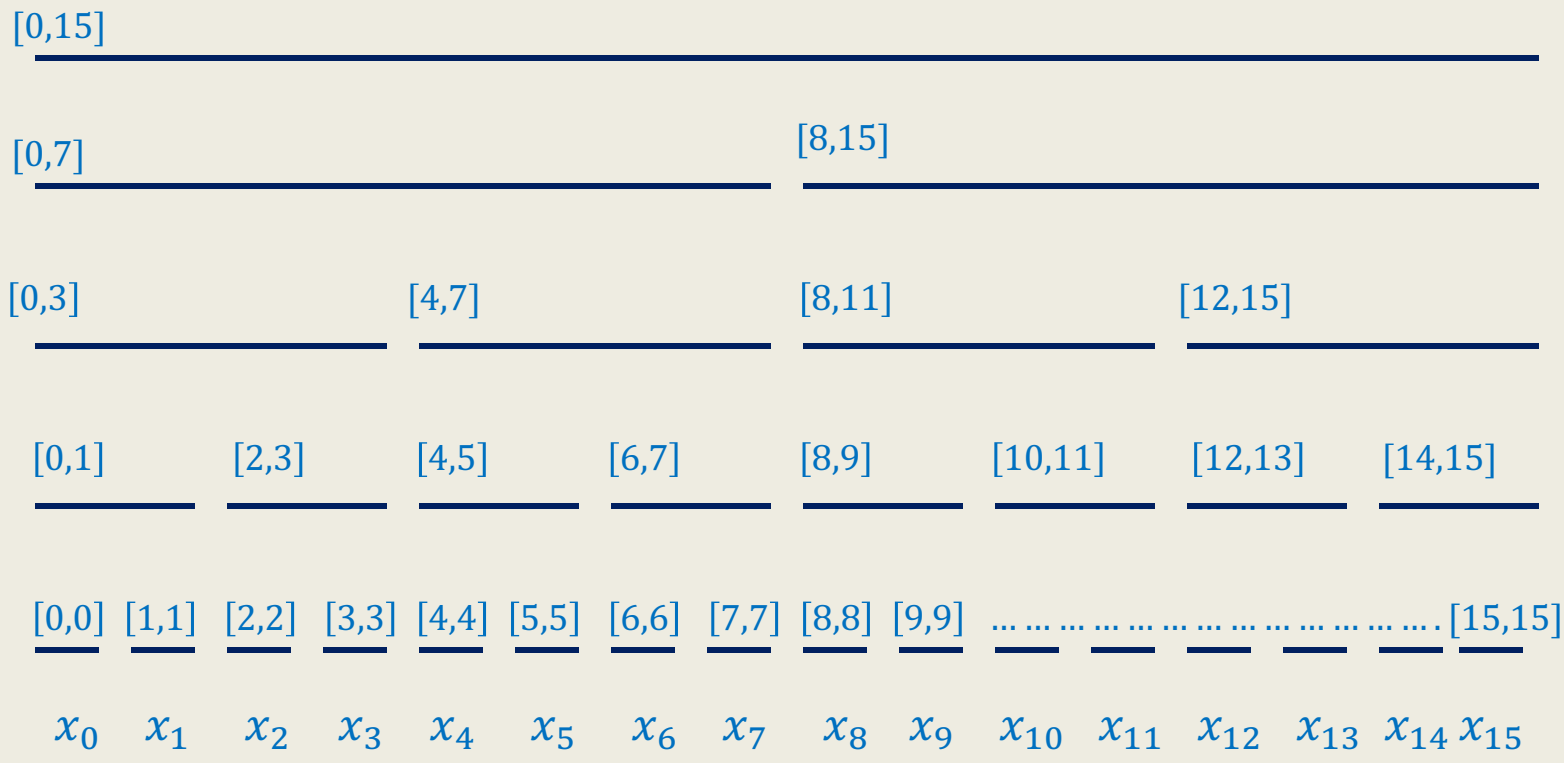
I have **reproduced** it for you again in the next slide

Have another look,

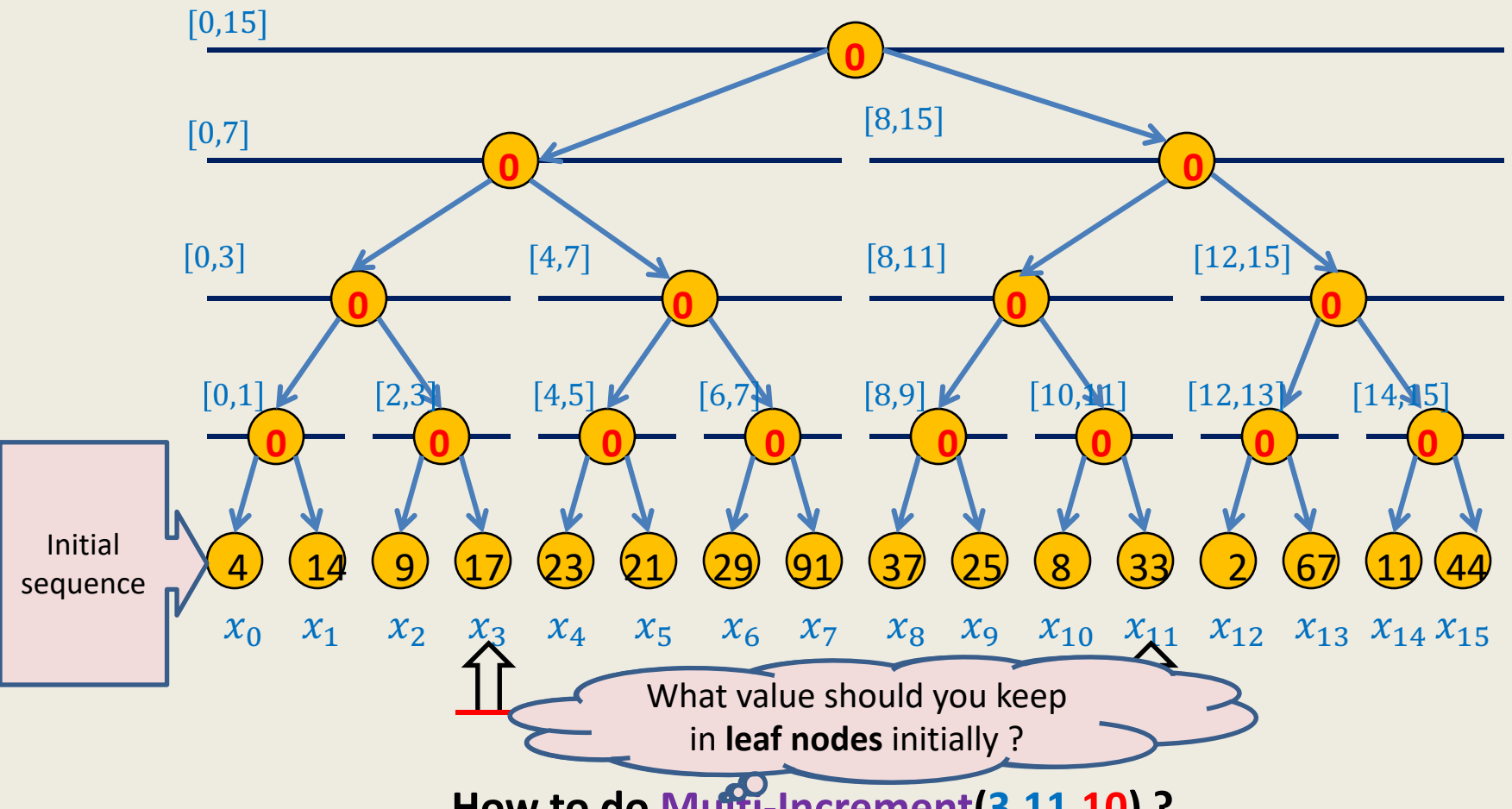
think for a while …

and then only proceed.

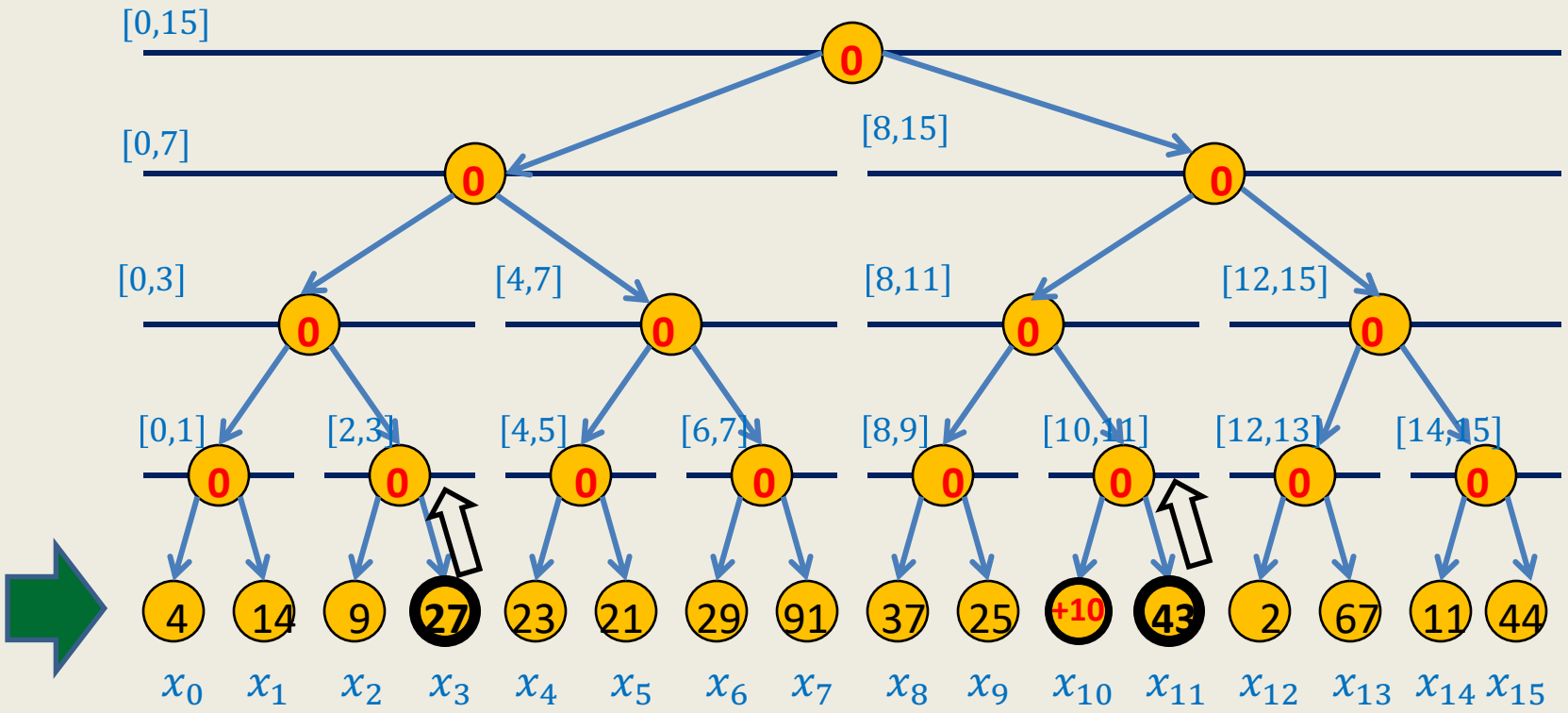
Which data structure emerges ?



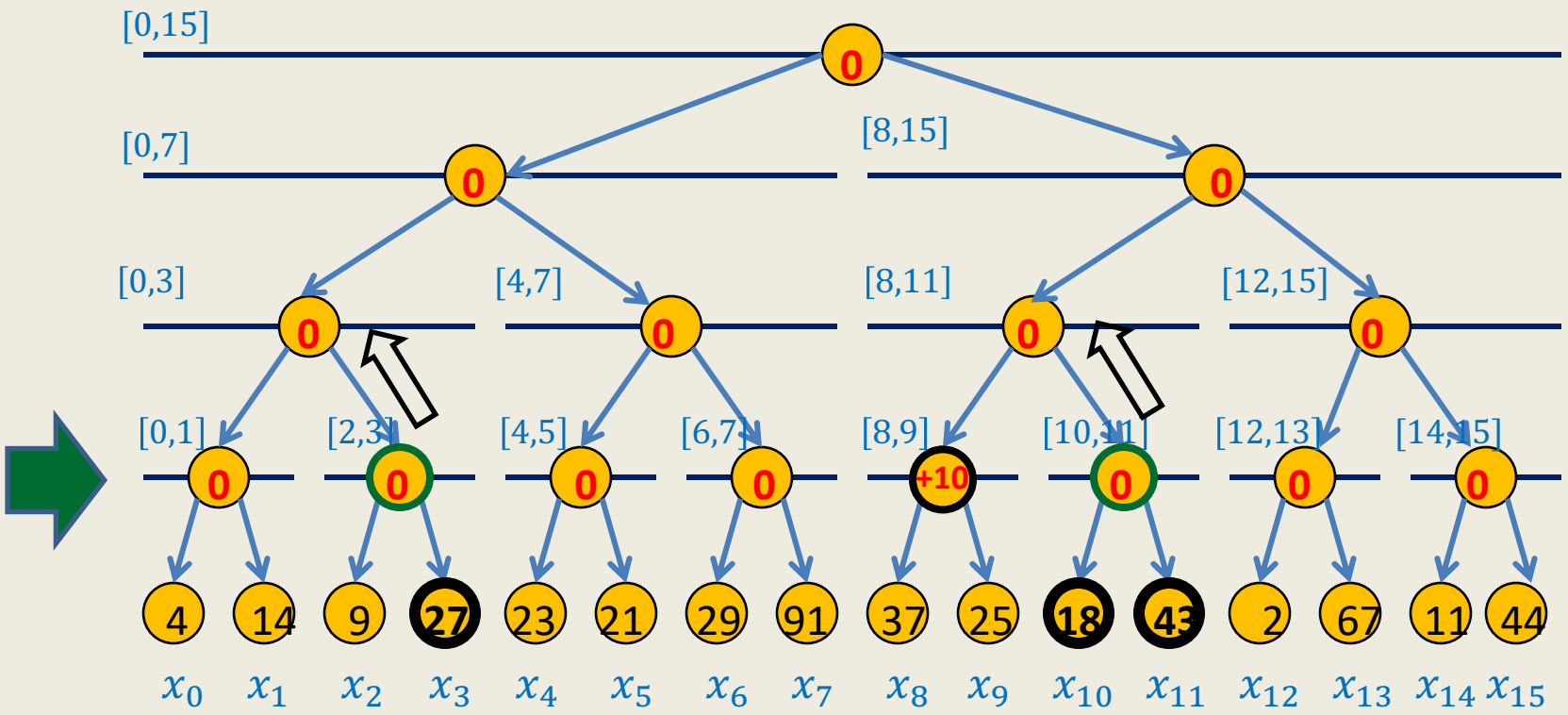
Isn't it a **Binary tree** that you thought ?



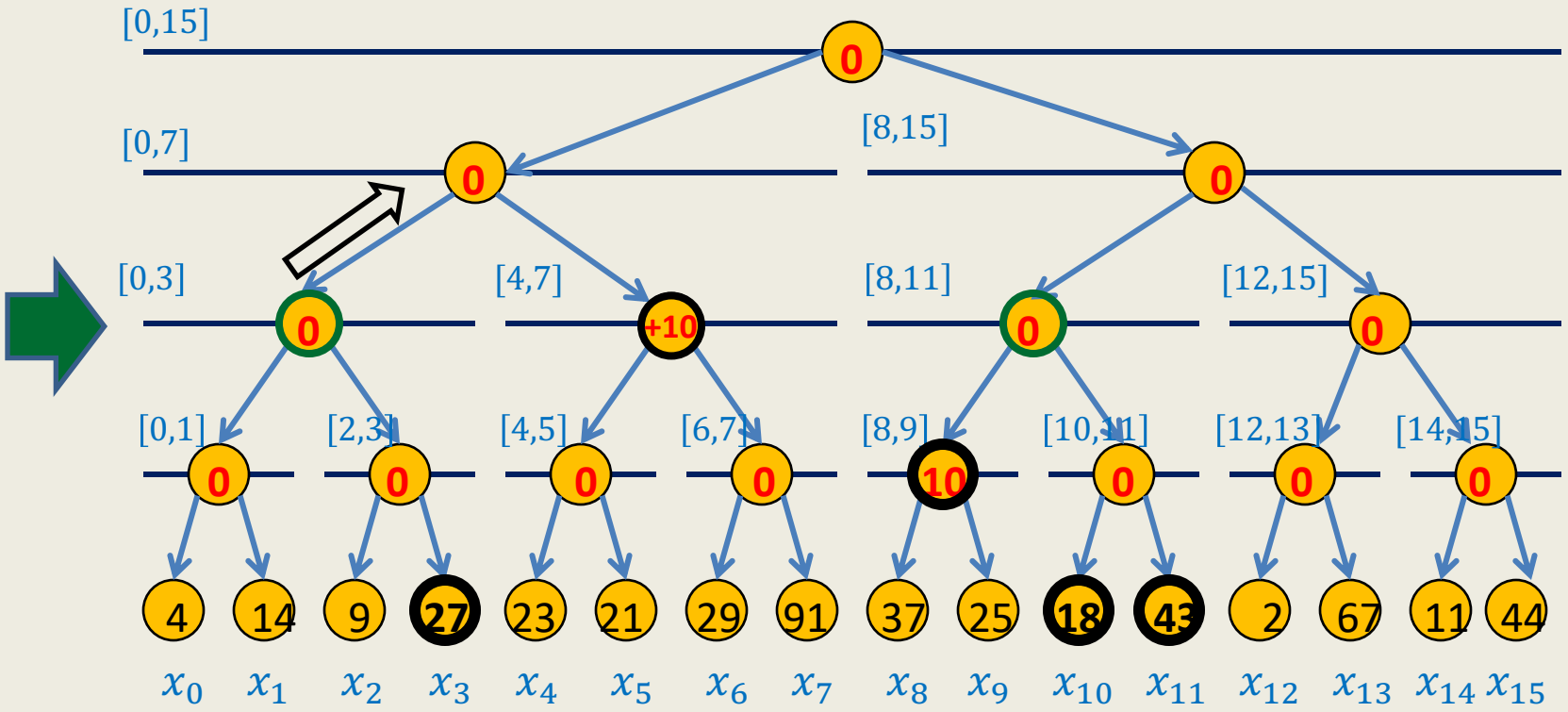
How to do Multi-Increment(3,11,10) ?



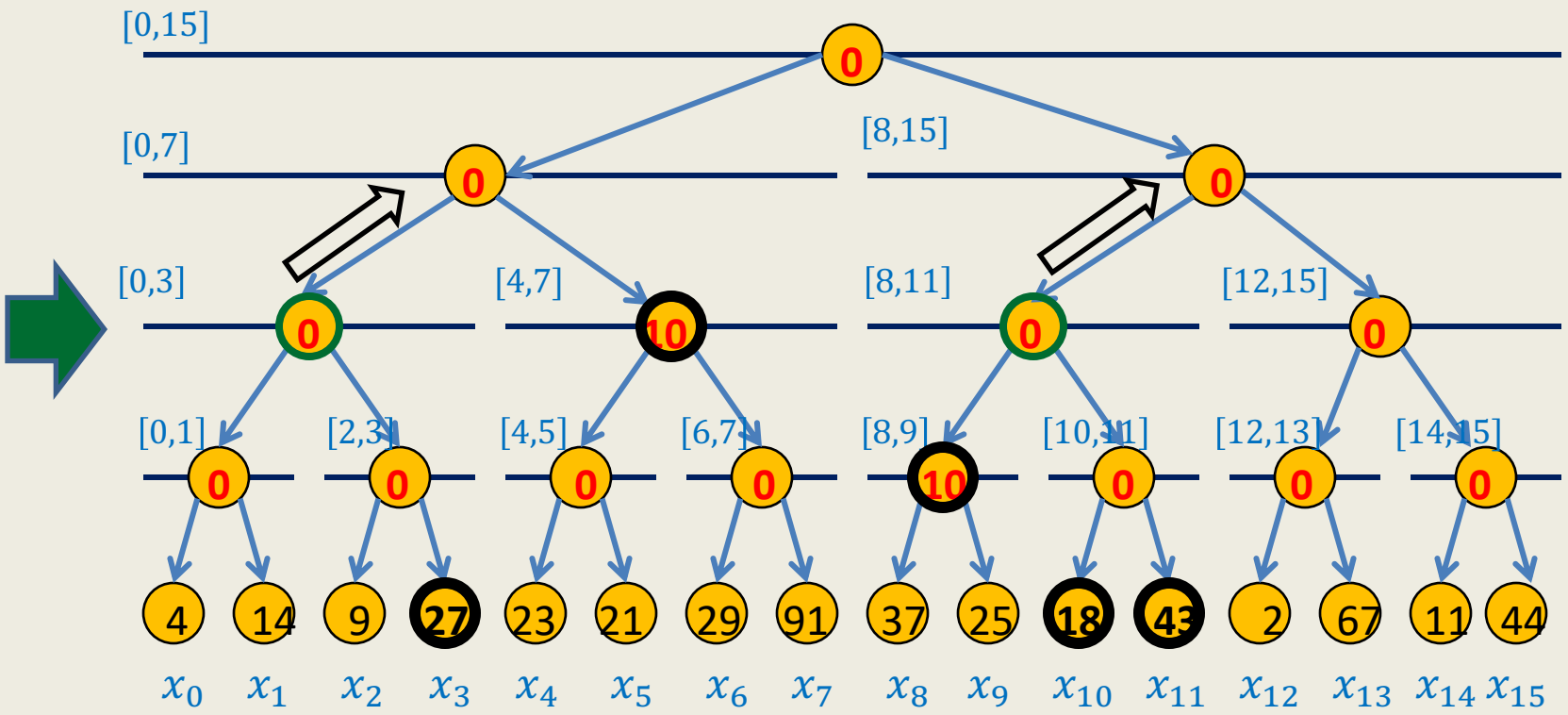
How to do Multi-Increment(3,11,10) ?



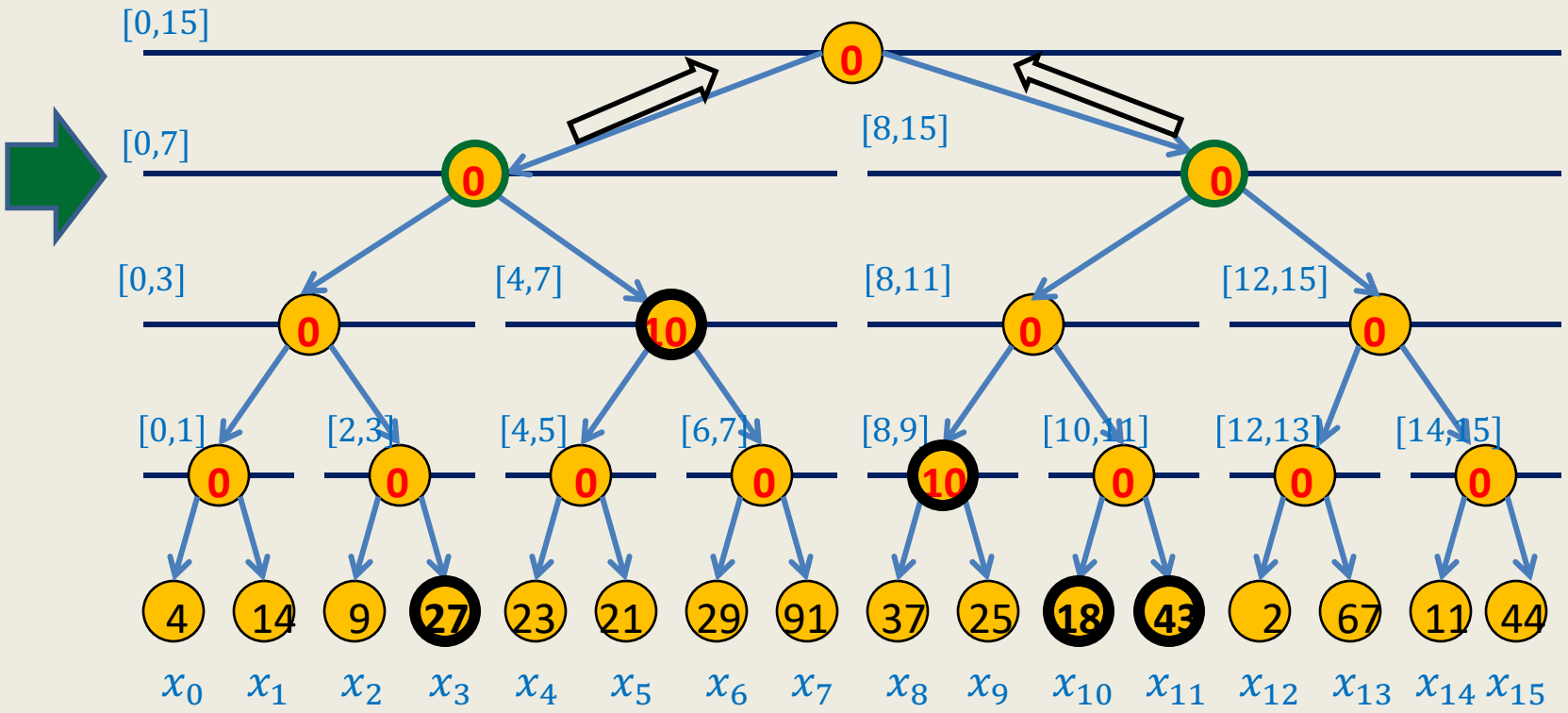
How to do Multi-Increment(3,11,10) ?



How to do Multi-Increment(3,11,10) ?

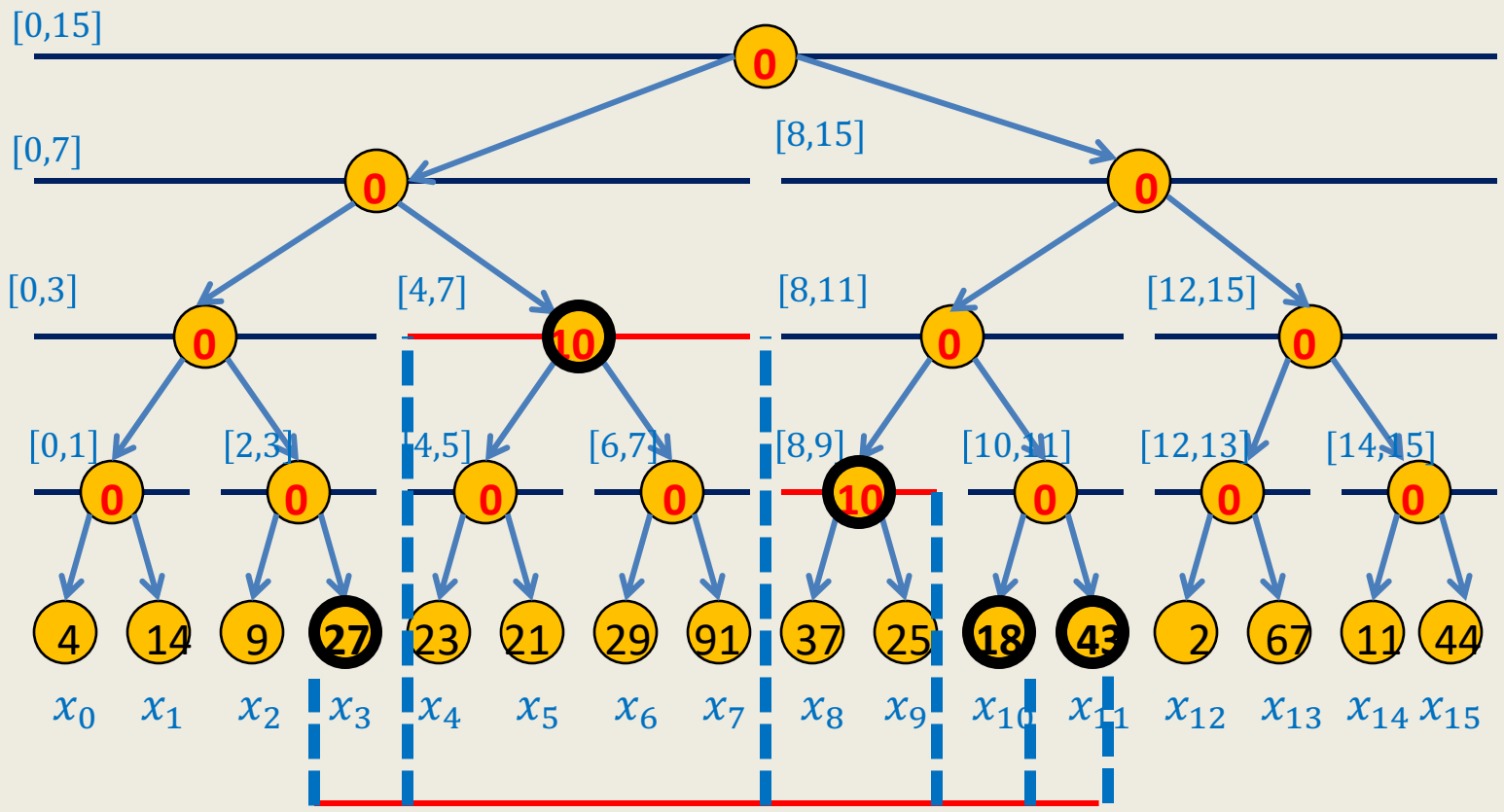


How to do Multi-Increment(3,11,10) ?



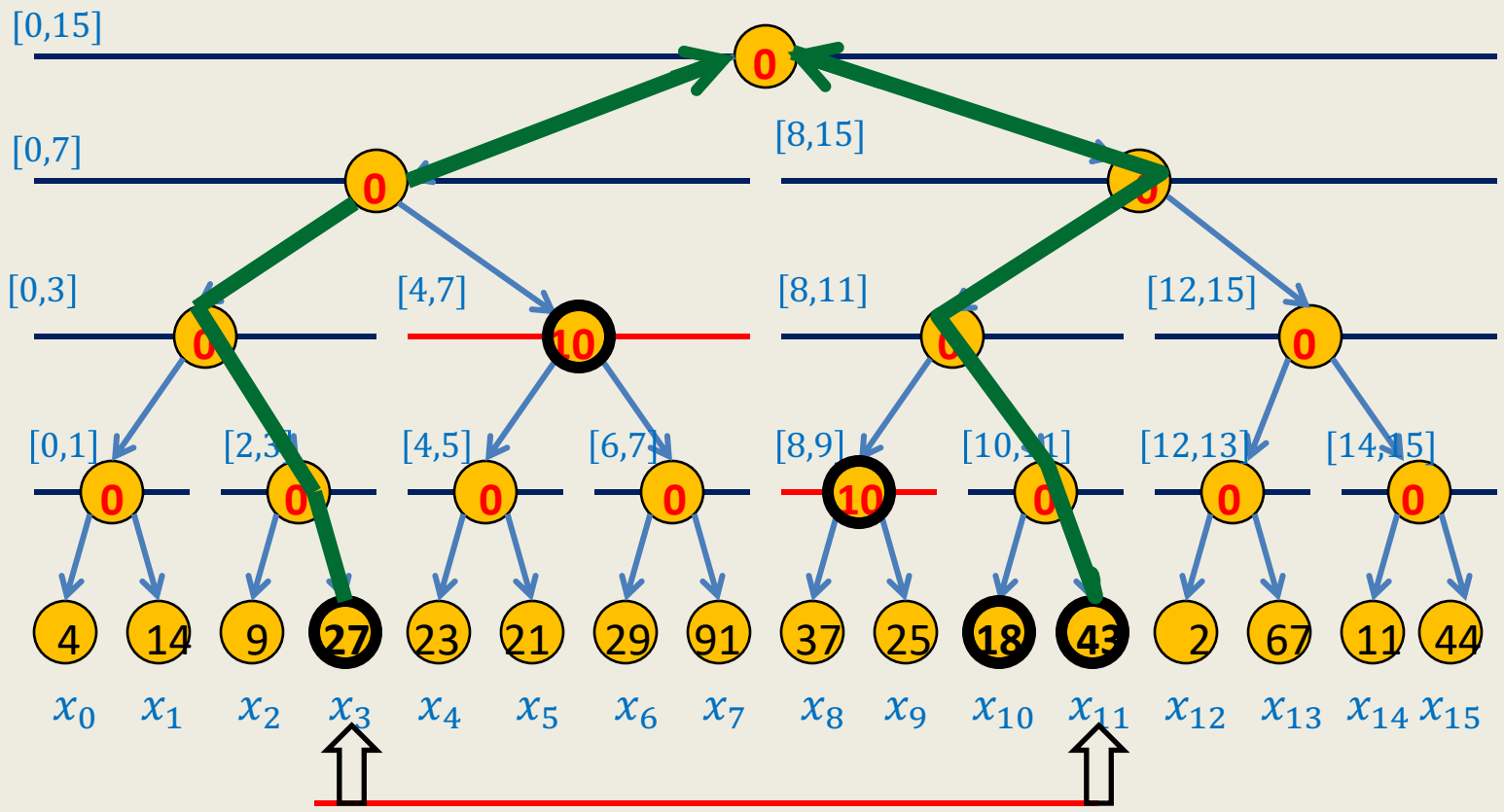
How to do Multi-Increment(3,11,10) ?

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 **u** and **v** be the leaf nodes corresponding to x_i and x_j.
2. Increment the value stored at **u** and **v**.

3. Keep repeating the following **step** as long as parent(**u**) <> parent(**v**)

   Move up by one step simultaneously from **u** and **v**

   - If **u** is **left child** of its parent, increment value stored in sibling of **u**.
   - If **v** is **right child** of its parent, increment value stored in sibling of **v**.

Executing Report(i) efficiently

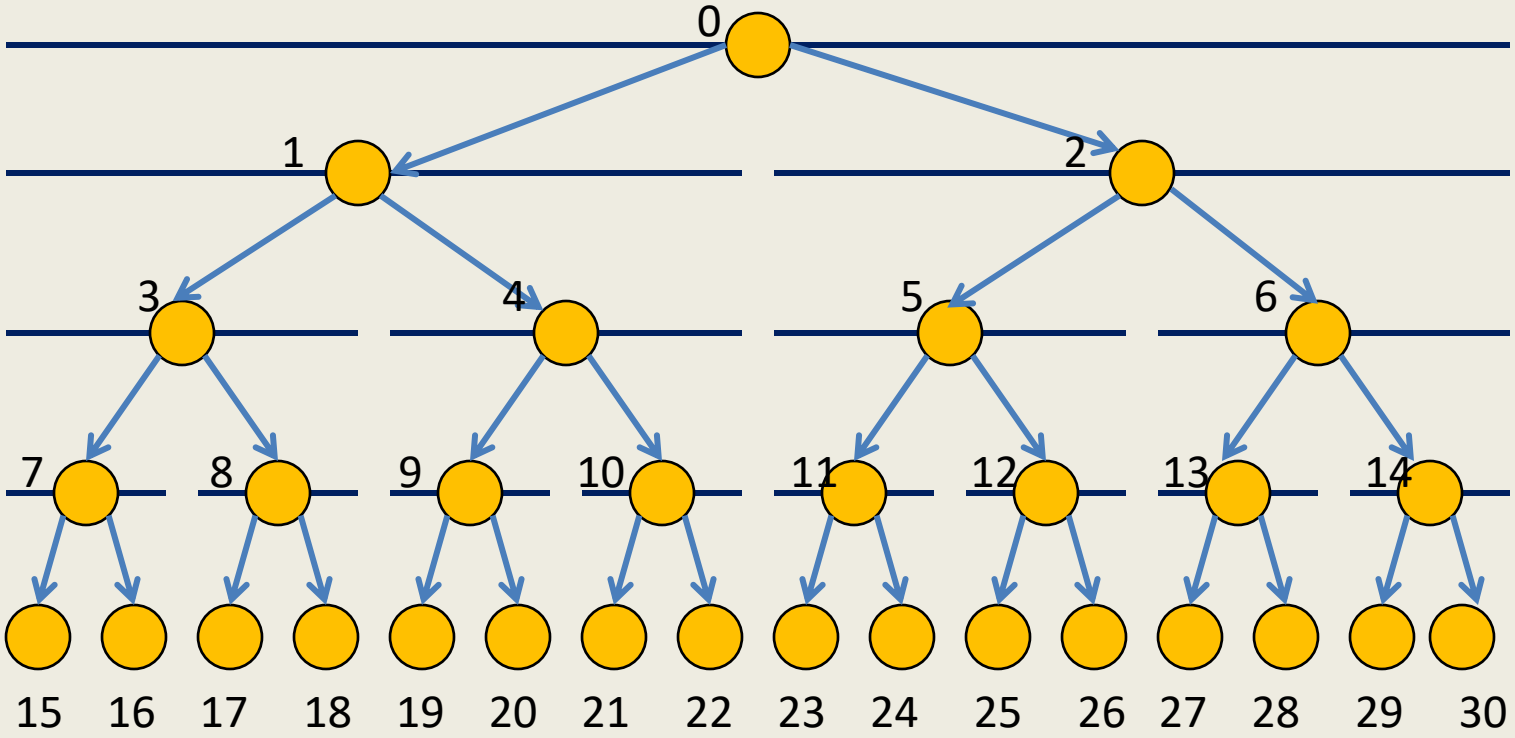
Sketch:

1. Let **u** be the leaf nodes corresponding to x_i.
2. **val** ← **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 2**n**-1.

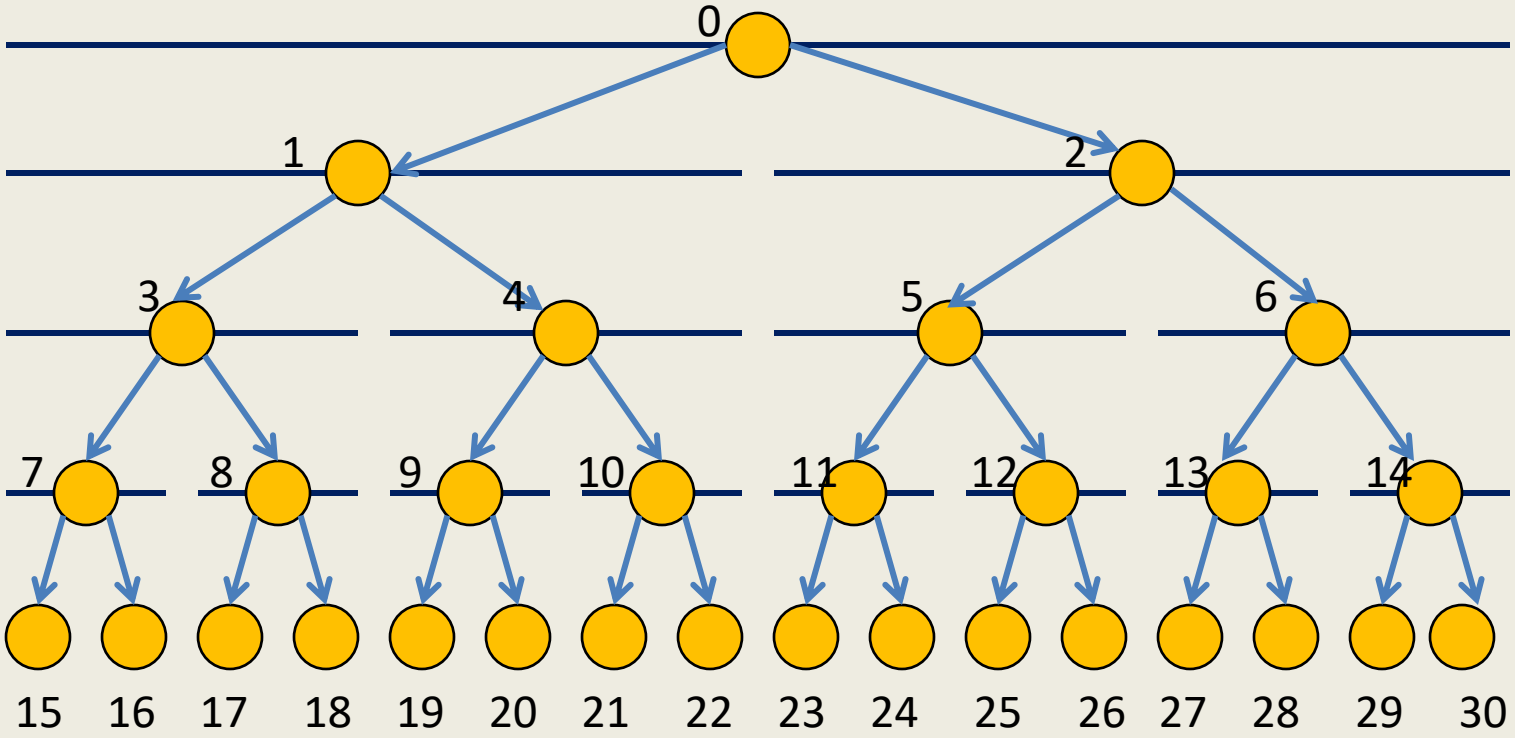
Copy the sequence **S** = $\prec x_0, \ldots, x_{n-1} \succ$ into **A**$[n-1]$…**A**$[2n-2]$

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 **A** of size 2**n**-1.

Copy the sequence **S** = $\prec x_0, \ldots, x_{n-1} \succ$ into **A**$[n-1]$…**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 ?

MultiIncrement(i, j, Δ)

MultiIncrement(i, j,Δ)

$i \leftarrow (n-1) + i$;

$j \leftarrow (n-1) + j$;

A(i) ← A(i) + Δ;

If (j > i)

{ A(j) ← A(j) + Δ;

While($\lfloor (i-1)/2 \rfloor$ <> $\lfloor (j-1)/2 \rfloor$)

{

If(i%2=1) A($i+1$) ← A($i+1$) + Δ;

If(j%2=0) A($j-1$) ← A($j-1$) + Δ;

$i \leftarrow \lfloor (i-1)/2 \rfloor$;

$j \leftarrow \lfloor (j-1)/2 \rfloor$;

}

}

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 **O**(n)

for maintaining a sequence **S** = $\prec x_0, \ldots, x_{n-1} \succ$ such that

each **Multi-Increment**() and **Report**() operation takes **O**(log n) time.

Problem 2

Dynamic Range-minima

Problem 2

Given an initial sequence **S** = $\prec x_0, \ldots, x_{n-1} \succ$ 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 $\{x_k \mid \text{for each } i \leq k \leq j\}$

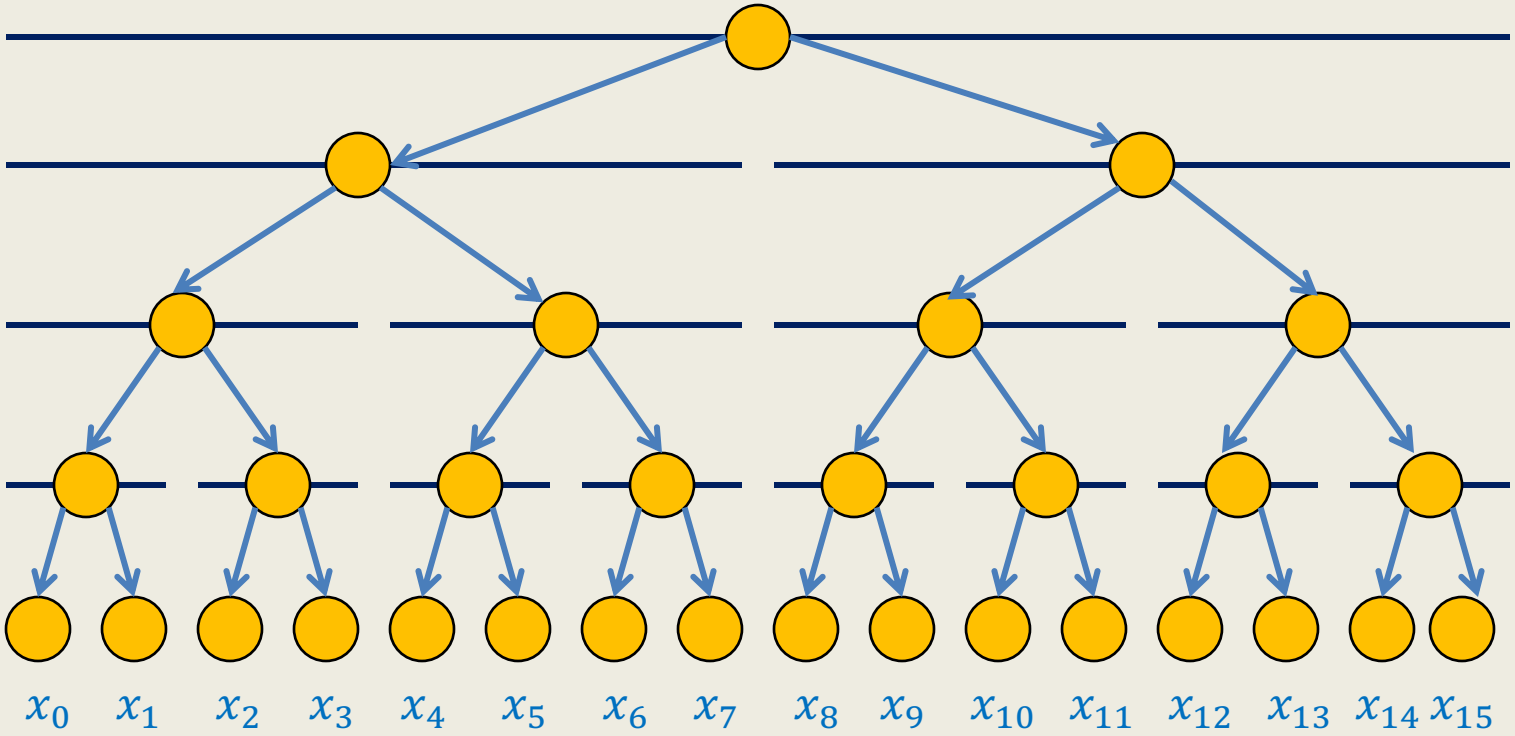
- **Update**(i, **a**):

  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.