

Data Structures and Algorithms

(CS210A)

Semester I – 2014-15

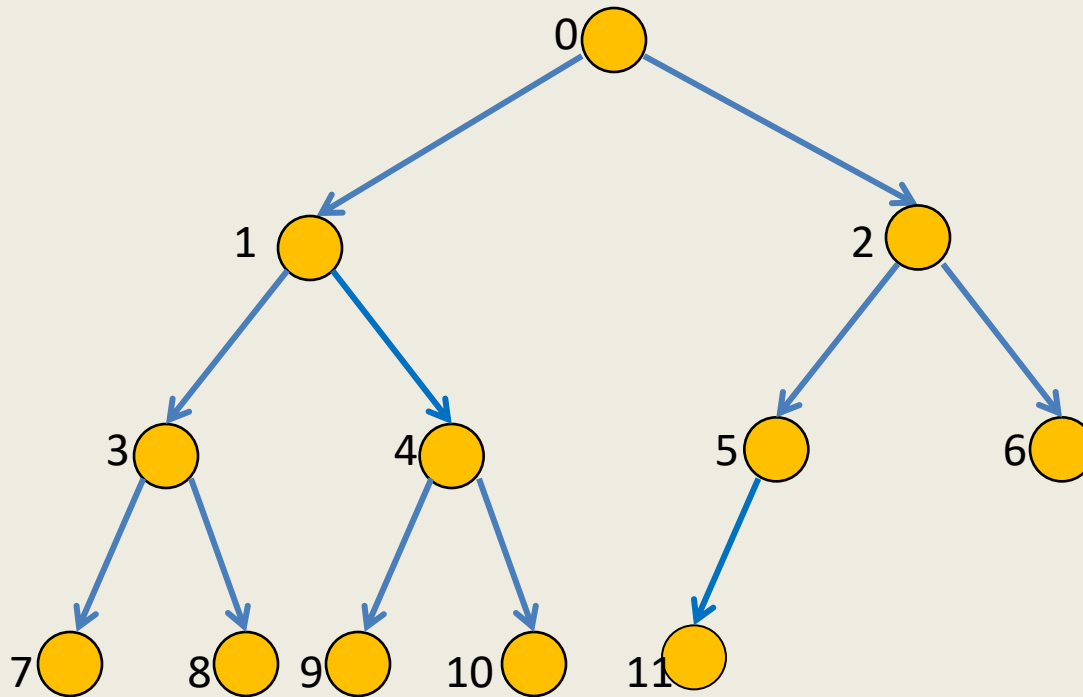
Lecture 29:

- **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

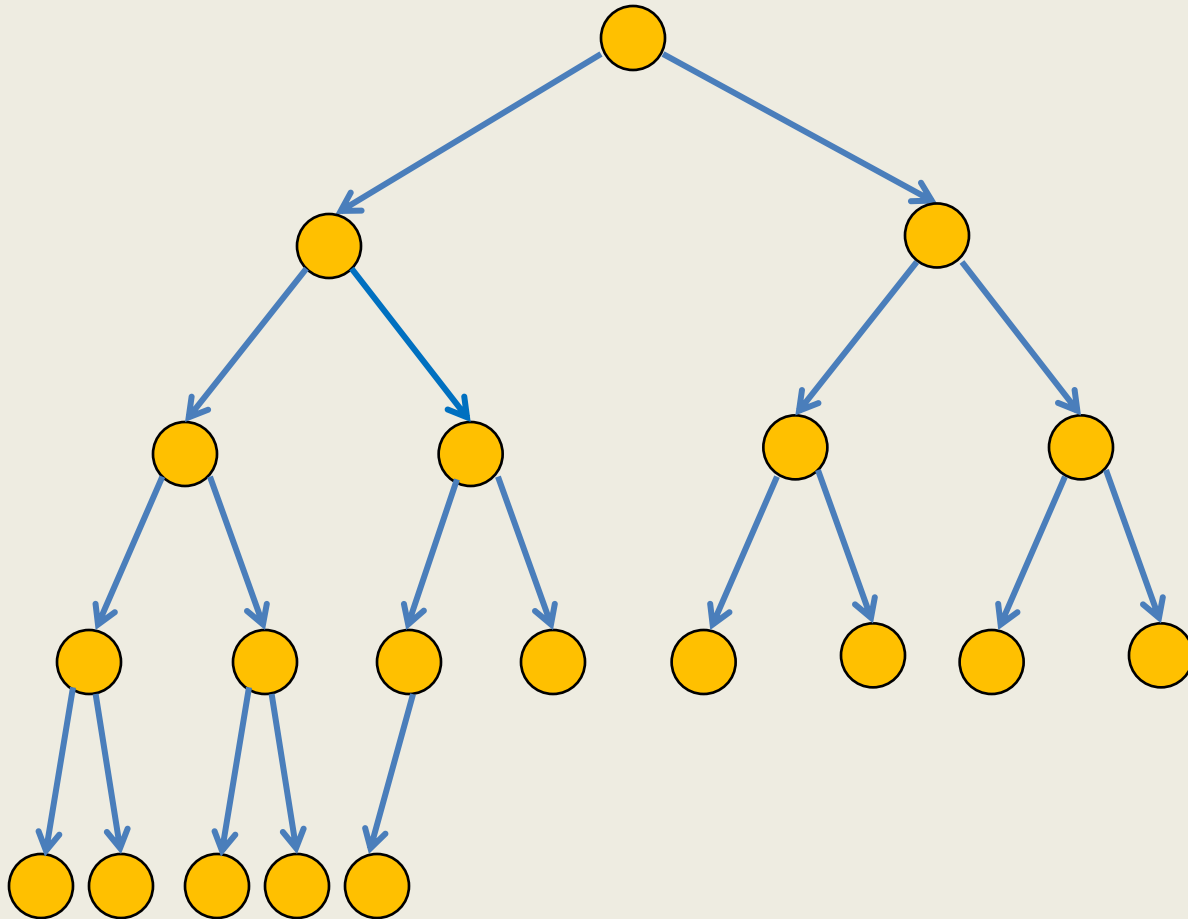
A complete binary tree

A complete binary on 12 nodes.



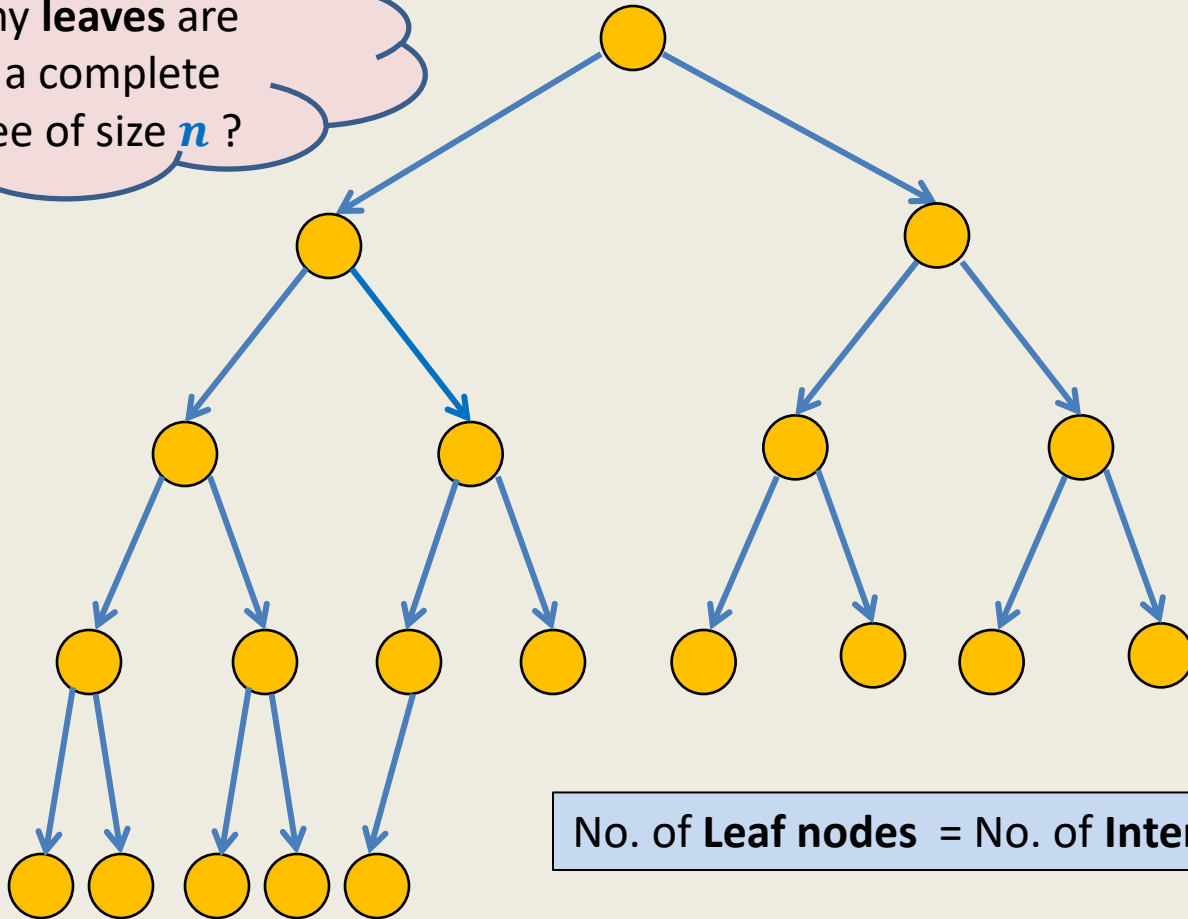
Theorem: A complete binary tree can be **implemented** by an array.

A complete binary tree



A complete binary tree

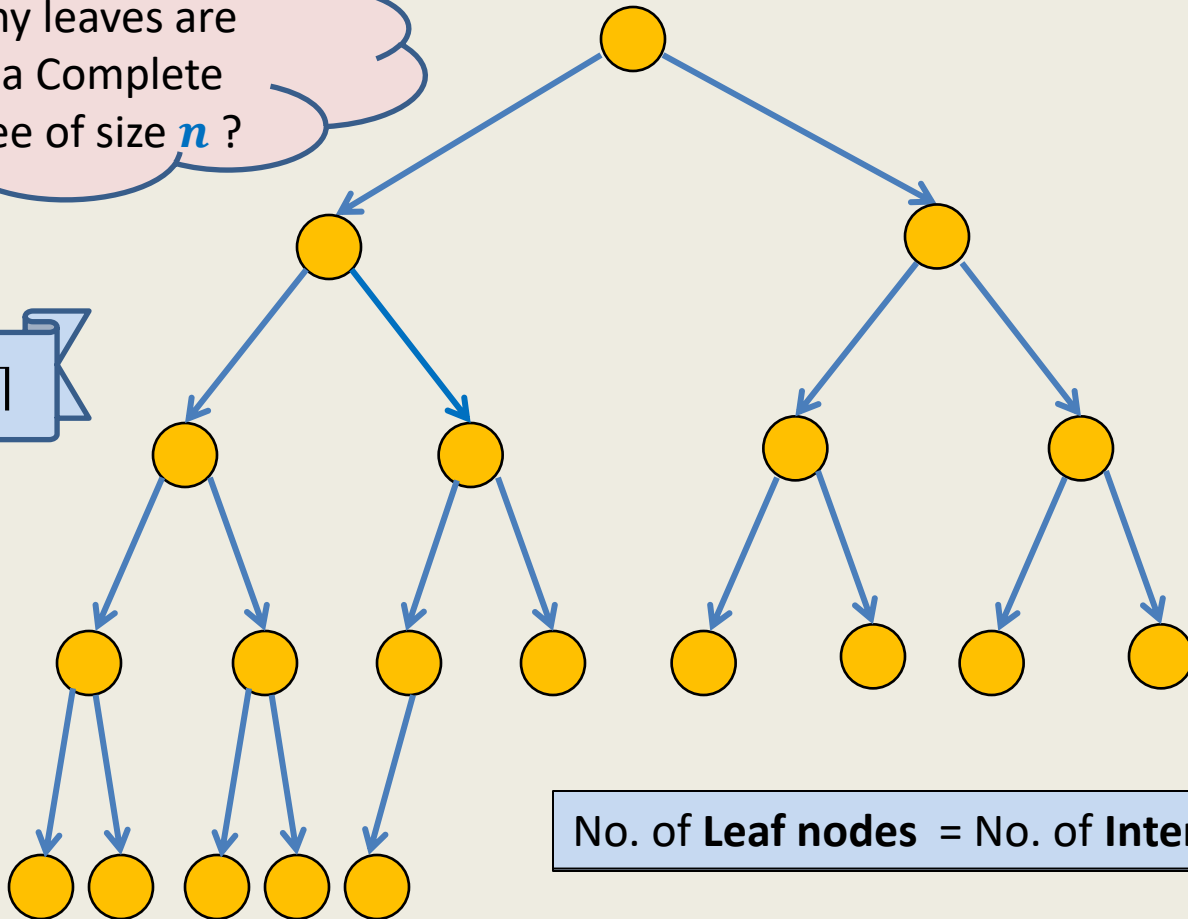
How many **leaves** are there in a complete Binary tree of size n ?



A complete binary tree

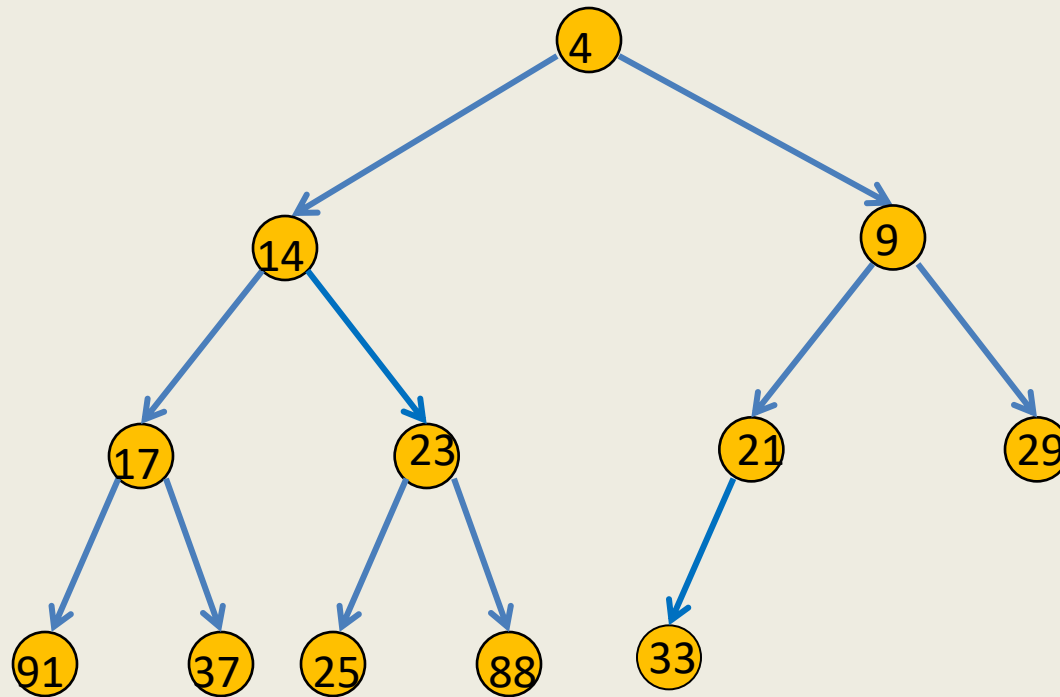
How many leaves are there in a Complete Binary tree of size n ?

$\lceil n/2 \rceil$



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			
---	----	---	----	----	----	----	----	----	----	----	----	--	--	--

Binary heap

	Binary heap
Find-min (H)	$O(1)$
Insert (x , H)	$O(\log n)$
Extract-min (H)	$O(\log n)$
Decrease-key (p , Δ , H)	$O(\log n)$
Merge (H ₁ , H ₂)	$O(n)$

Building a Binary heap

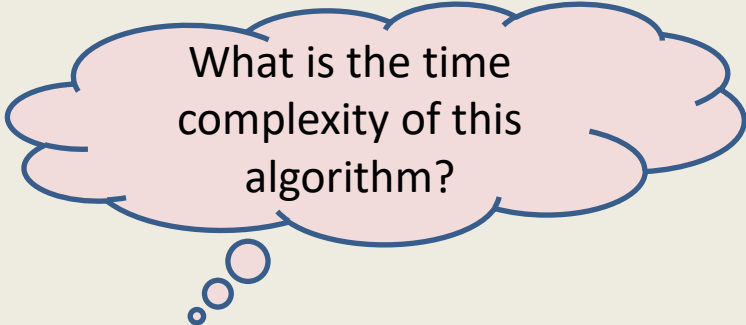
Building a Binary heap

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

Trivial solution:

(Building the Binary heap **incrementally**)

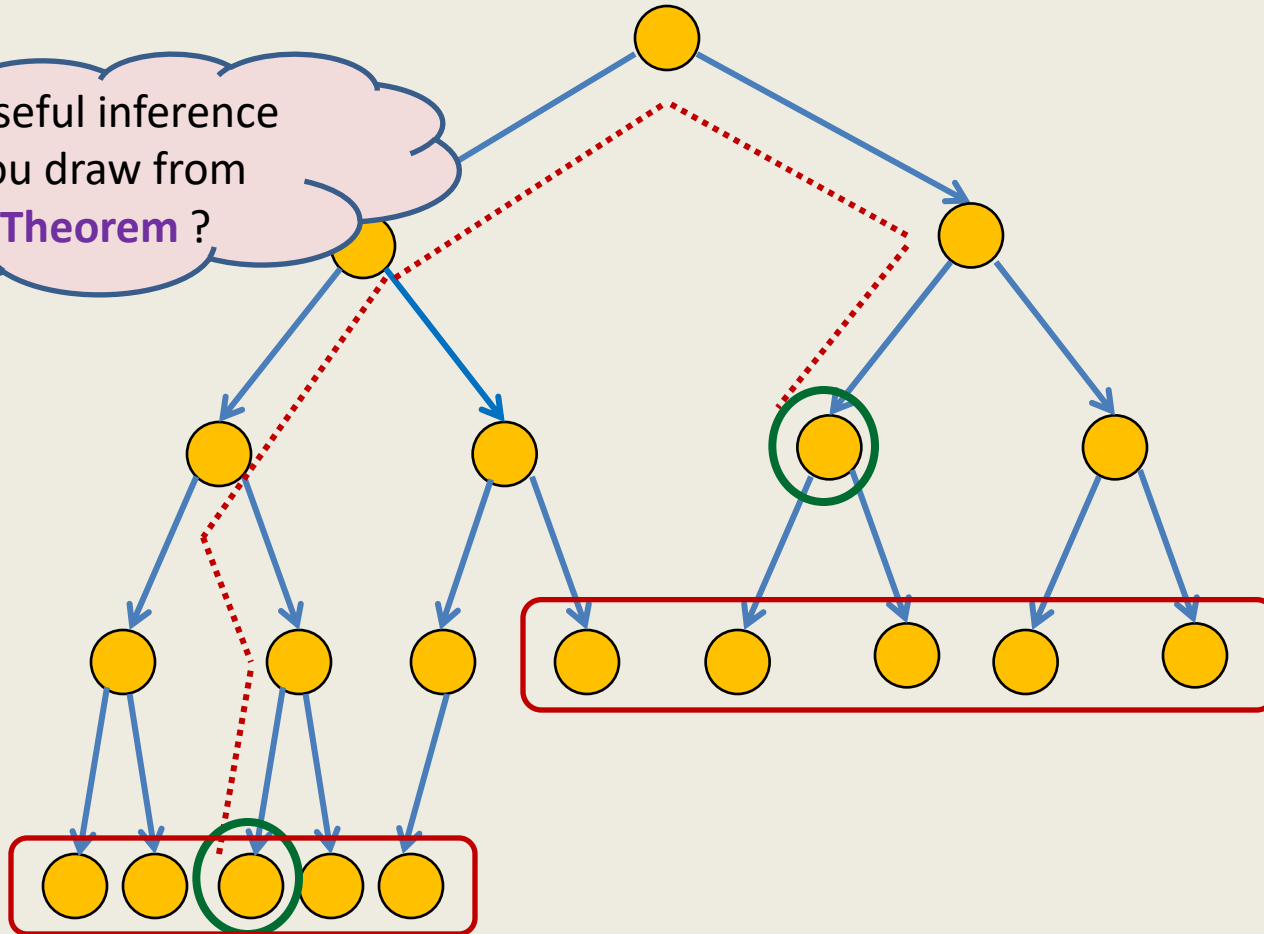
```
CreateHeap(H);  
For(  $i = 0$  to  $n - 1$  )  
    Insert( $x_i, H$ );
```



What is the time complexity of this algorithm?

Building a Binary heap incrementally

What useful inference can you draw from this **Theorem** ?



Top-down
approach

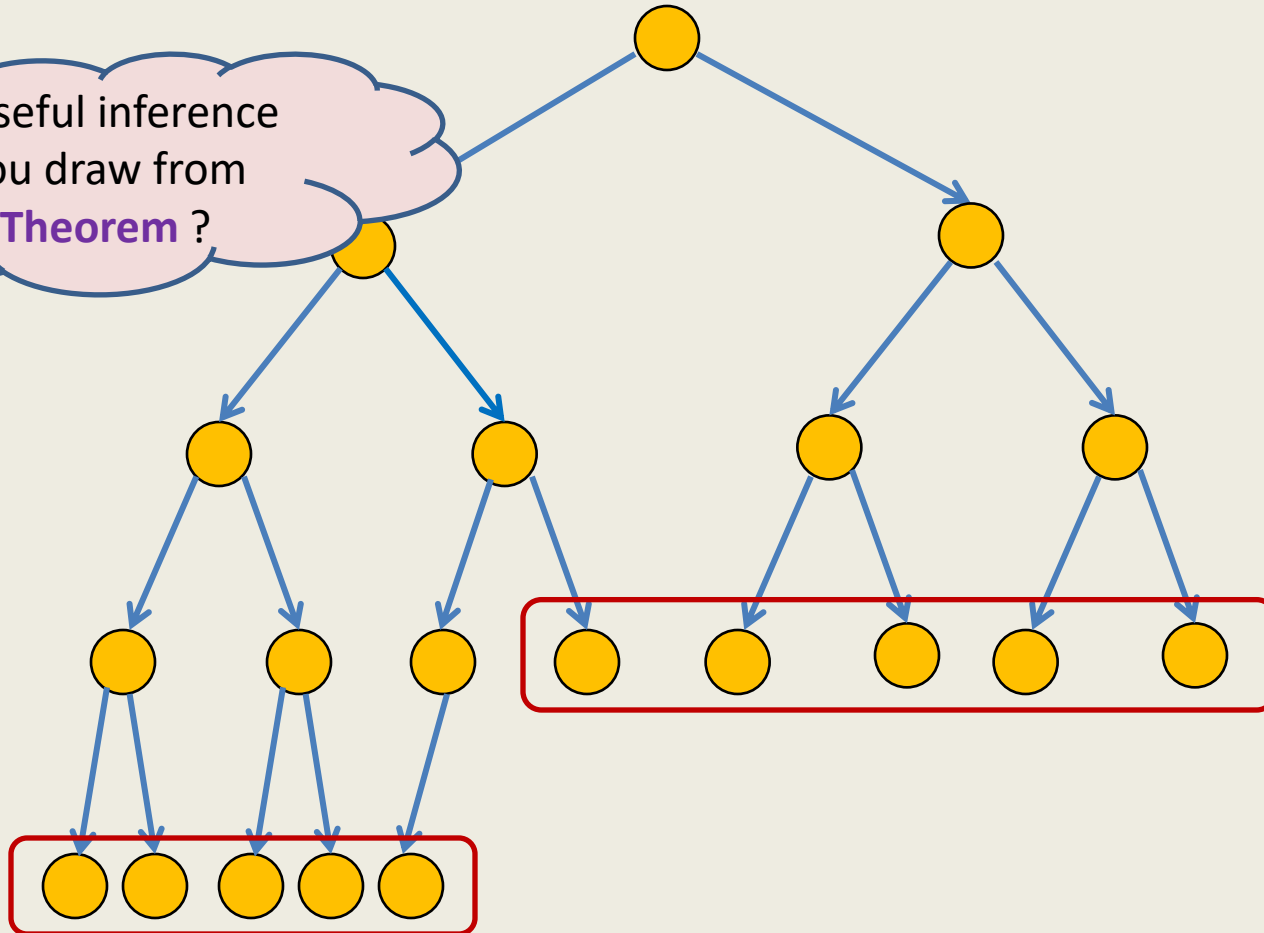
The time complexity for inserting a leaf node = $O(\log n)$

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

➔ **Theorem:** Time complexity of building a binary heap incrementally is $O(n \log n)$. 11

Building a Binary heap incrementally

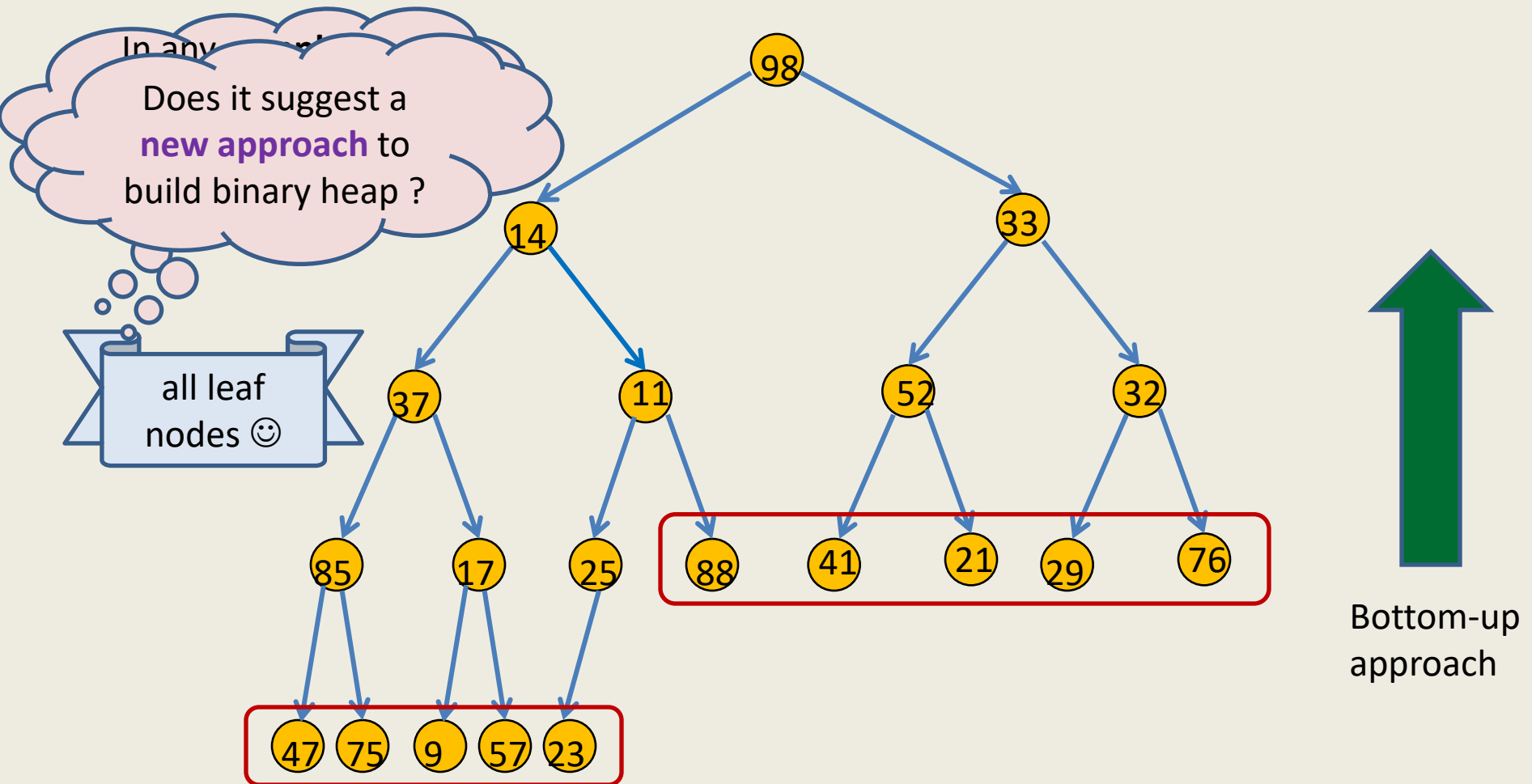
What useful inference
can you draw from
this **Theorem** ?



Top-down
approach

The $O(n)$ time algorithm must take $O(1)$ time
for each of the $\lceil n/2 \rceil$ leaves.

Think of **alternate** approach for building a binary heap



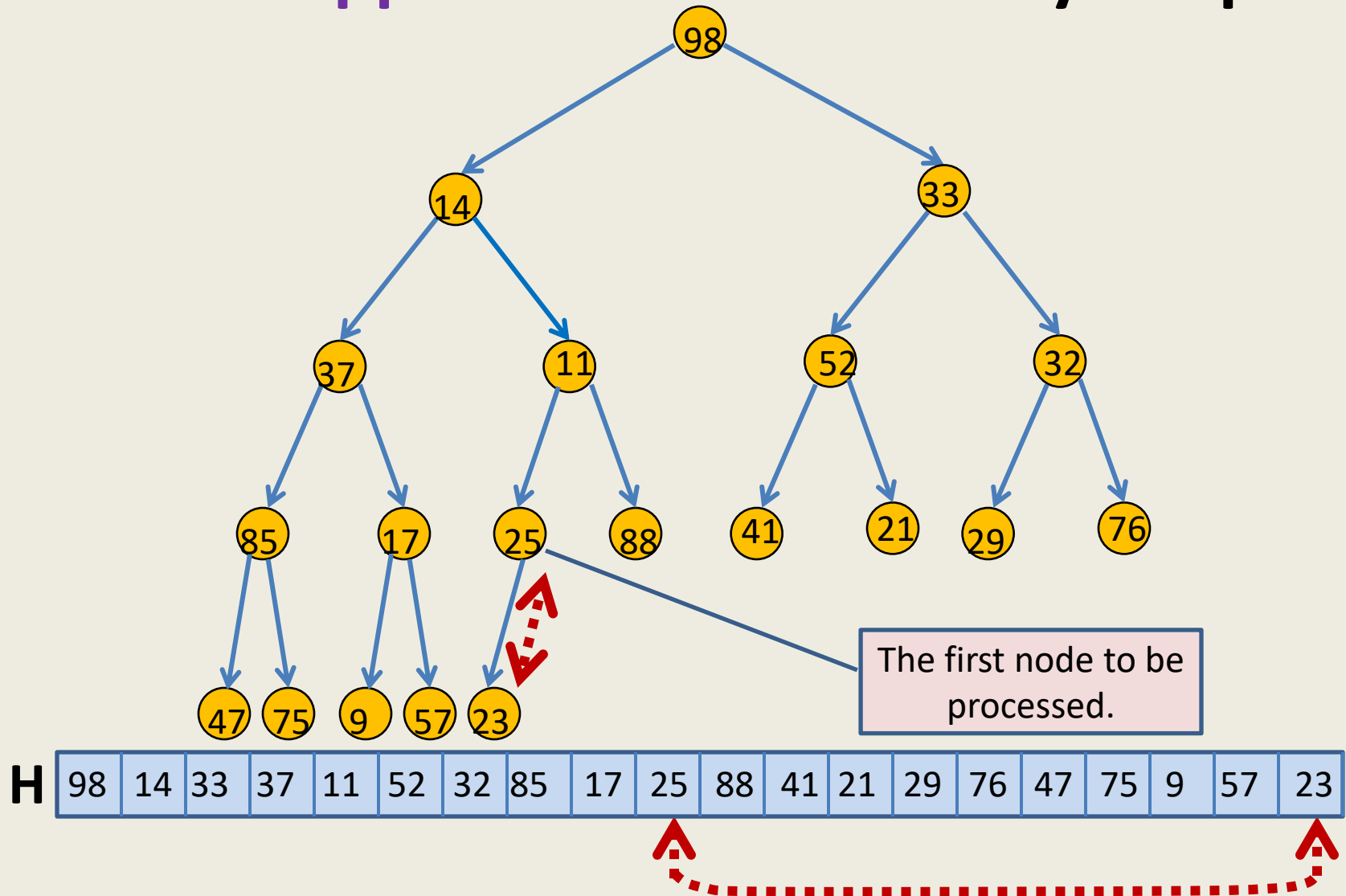
heap property: “Every **node** stores value smaller than its **children**”

We just need to ensure this property at each node.

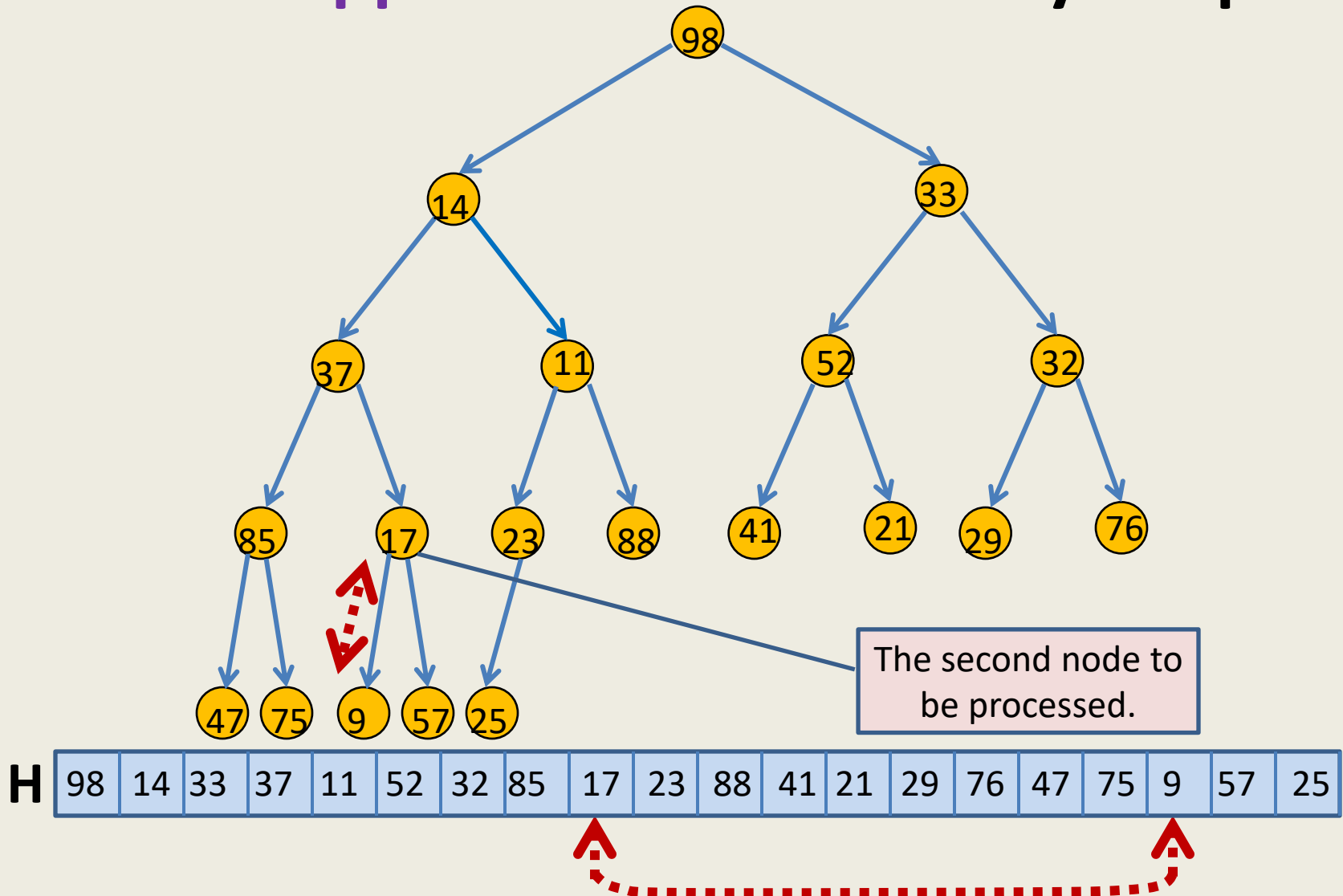
A new approach to build binary heap

1. Just copy the given n elements $\{x_0, \dots, x_{n-1}\}$ into an array H .
2. The **heap property** holds for all the leaf nodes in the corresponding complete binary tree.
3. Leaving all the leaf nodes, process the elements in the decreasing order of their numbering and set the heap property for each of them.

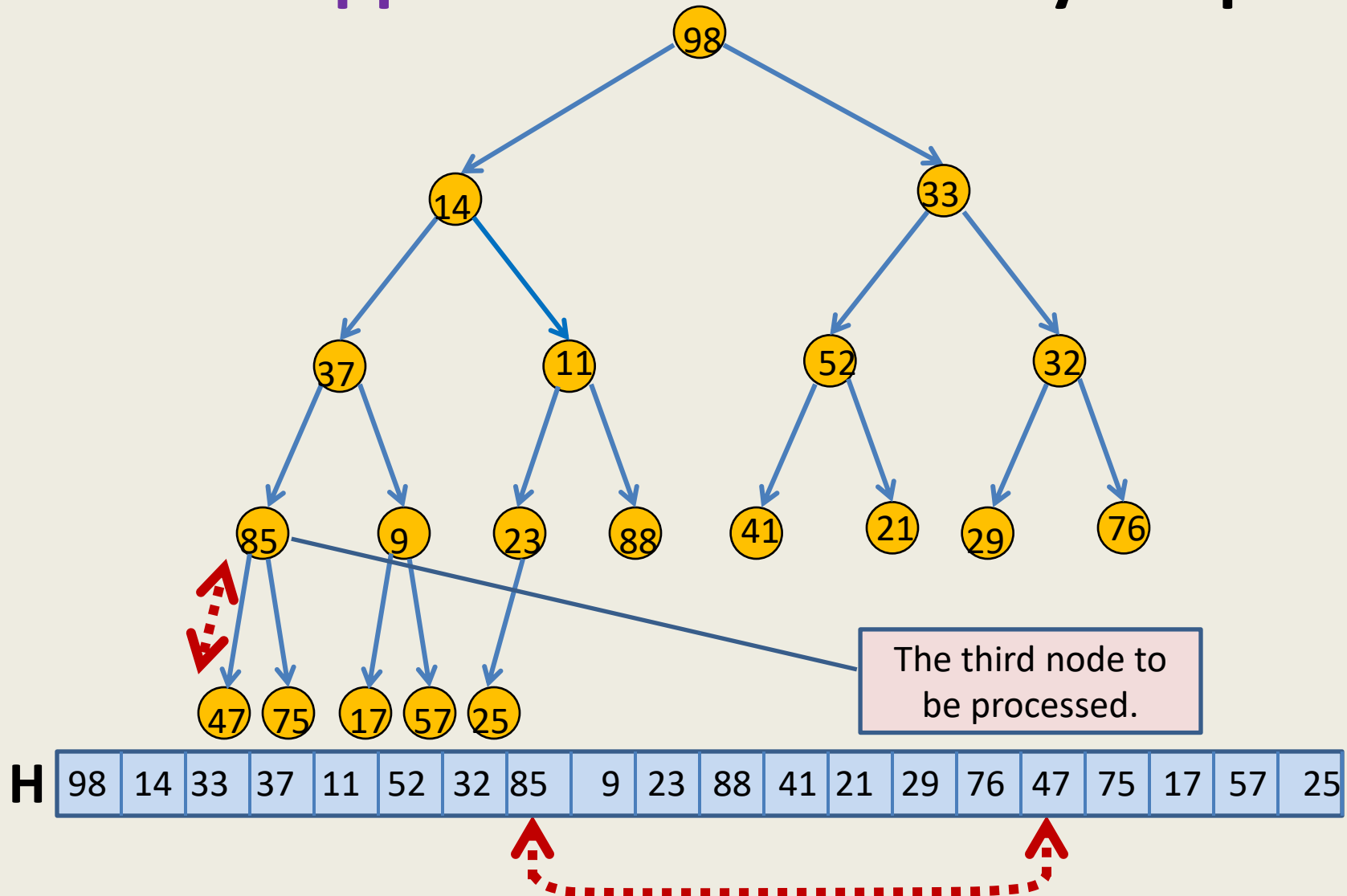
new approach to build binary heap



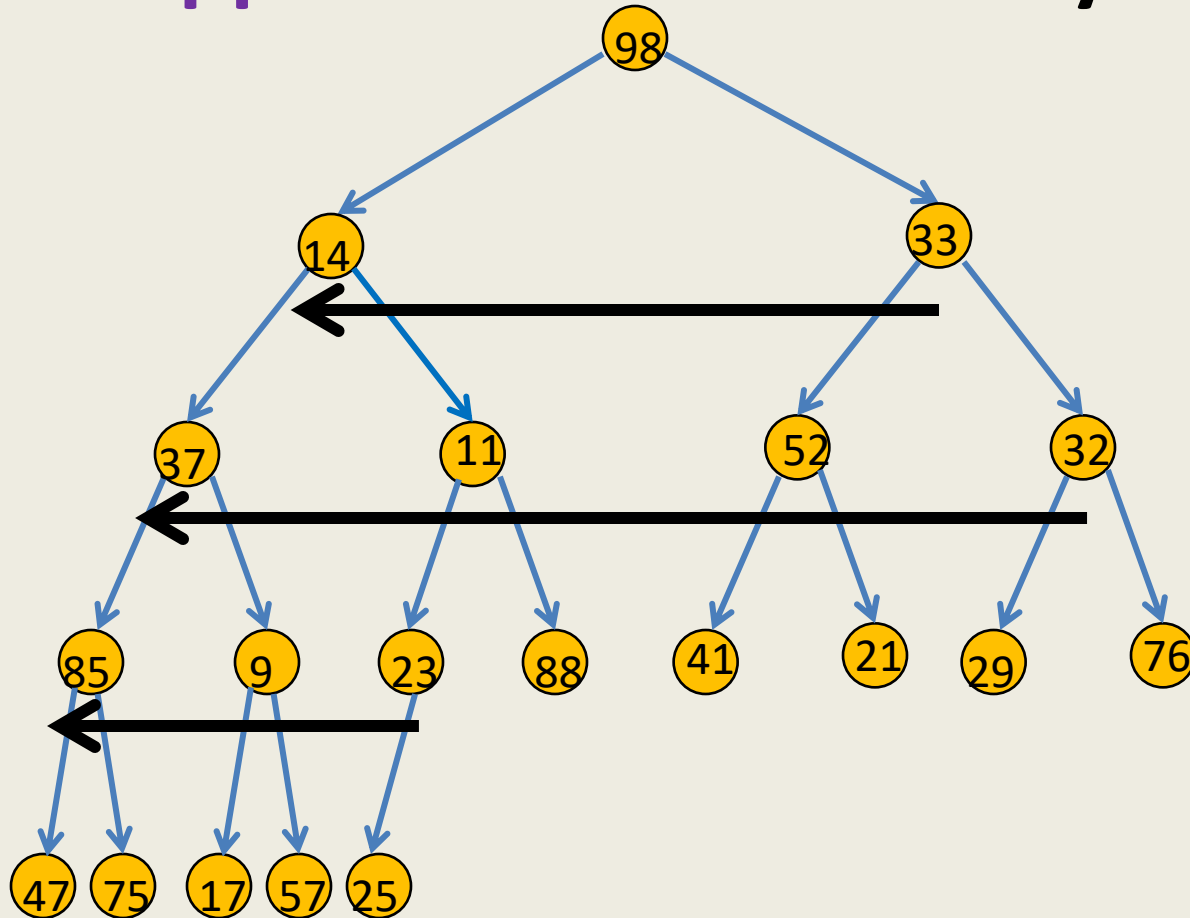
new approach to build binary heap



new approach to build binary heap



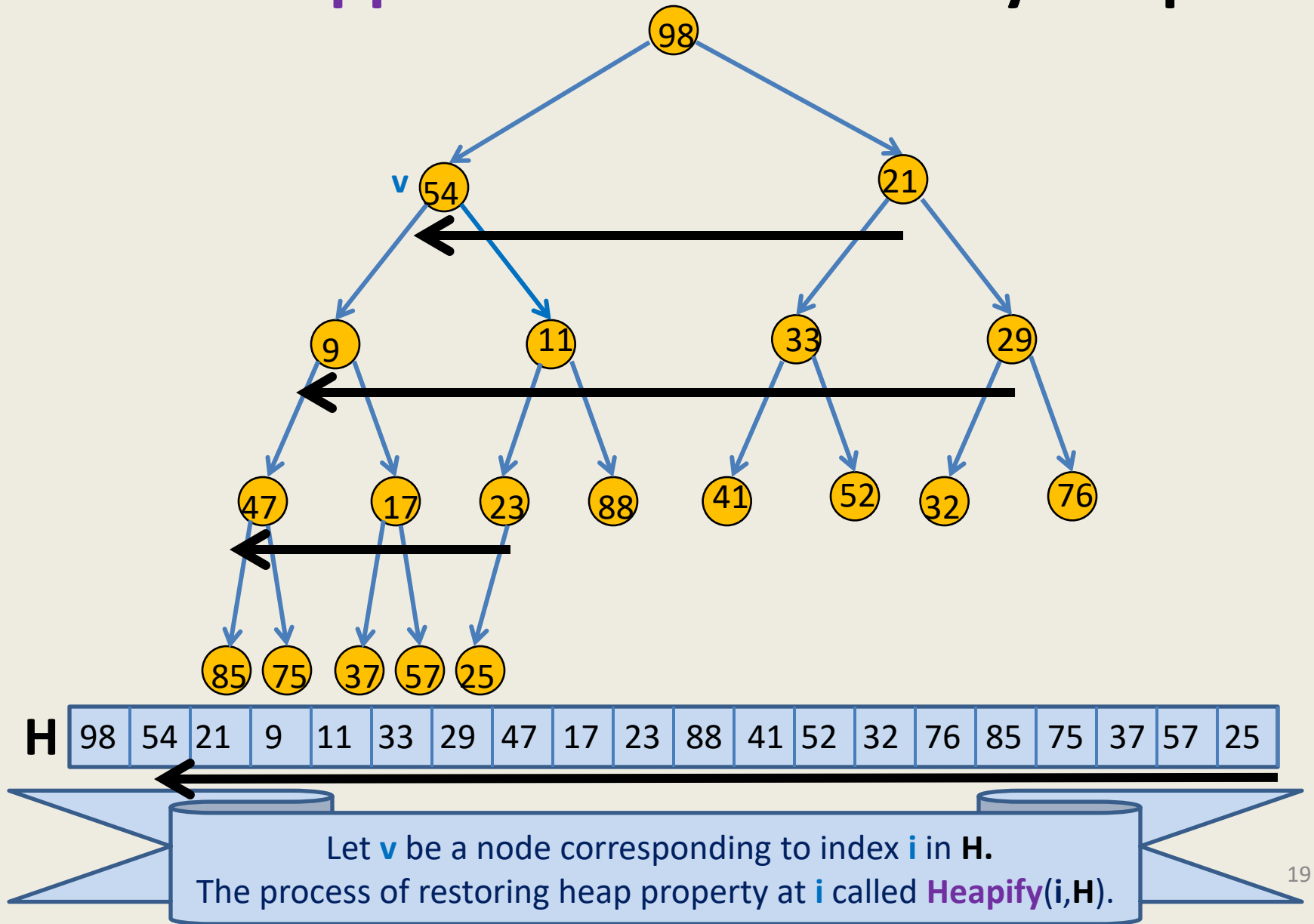
new approach to build binary heap



H

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

new approach to build binary heap



Heapify(*i*,H)

Heapify(*i*,H)

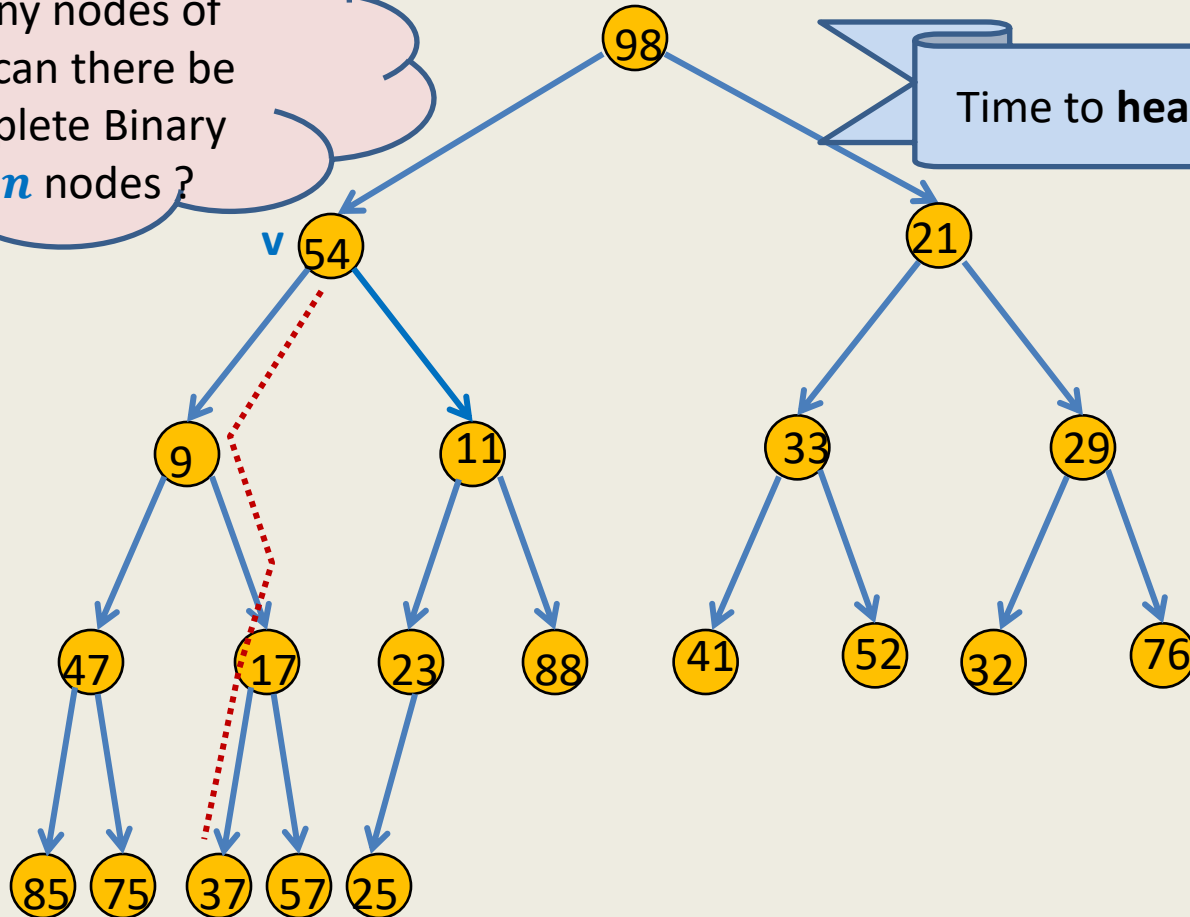
```
{  n ← size(H) - 1 ;
  Flag ← true;
  While( i ≤ [(n-1)/2]  and Flag = true  )
  {  min ← i;
    If(H[i] > H[2i + 1])  min ← 2i + 1;
    If( For node i, compare its value with those of its children + 2;
    If( If it is smaller than any of its children  → Swap it with smallest child
      {  H(i) ↔ H(min);
        and move down ...
      }
      Else stop ! in; }
    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 ?

Height(v)



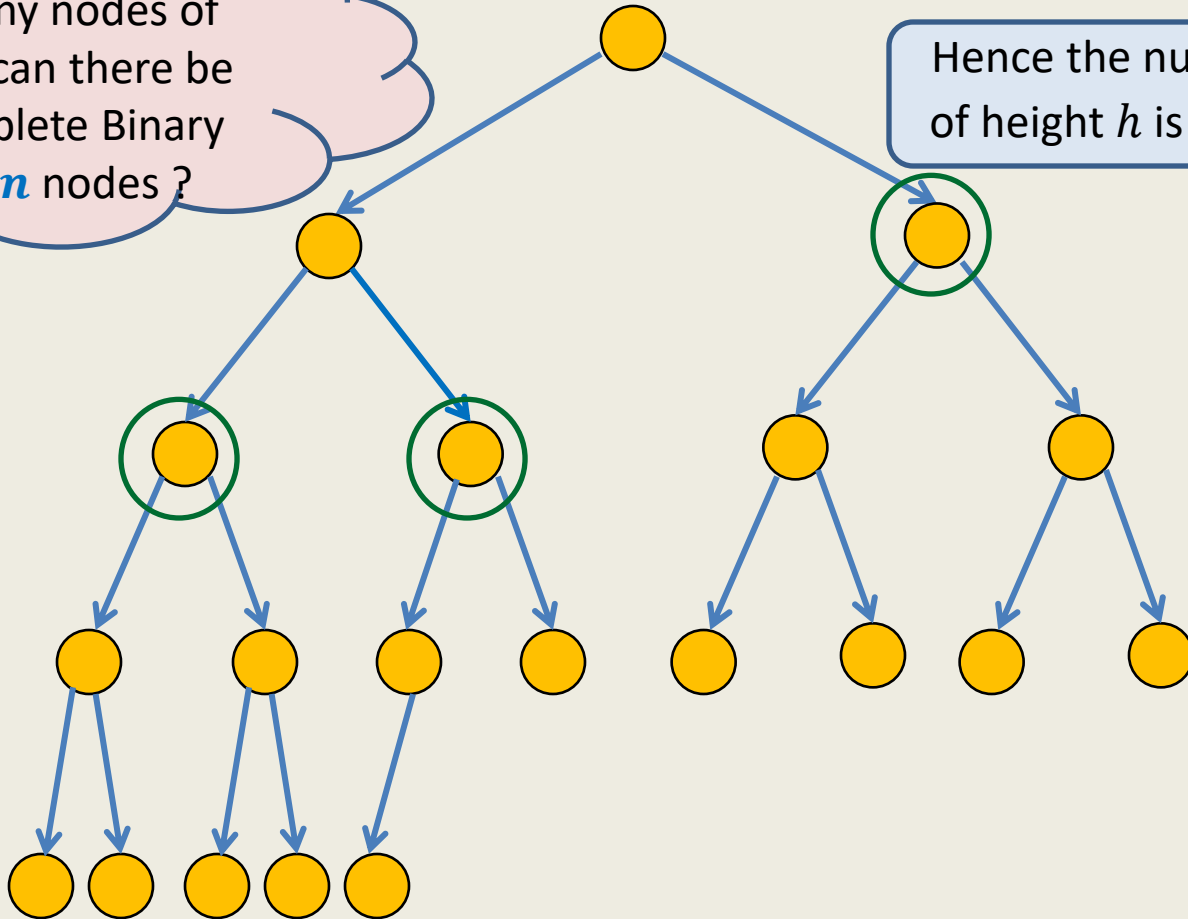
H 98 54 21 9 11 33 29 47 17 23 88 41 52 32 76 85 75 37 57 25

Time complexity of algorithm = $\sum_h O(h) \cdot \underbrace{N(h)}_{\text{No. of nodes of height } h}$

A complete binary tree

How many nodes of height h can there be in a complete Binary tree of n nodes?

Hence the number of nodes of height h is bounded by $\frac{n}{2^h}$



Each subtree is also a complete binary tree.

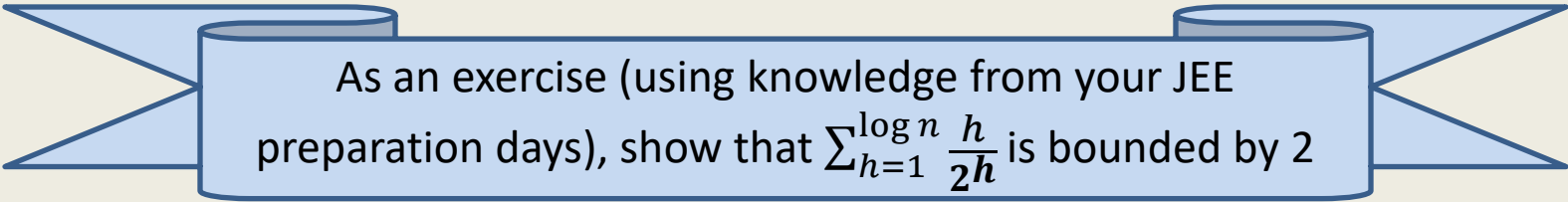
➔ A subtree of height h has at least 2^h nodes

Moreover, no two subtrees of height h in the given tree have any element in common

Building Binary heap in $O(n)$ time

Lemma: the number of nodes of height h is bounded by $\frac{n}{2^h}$.

$$\begin{aligned}\text{Hence Time complexity to build the heap} &= \sum_{h=1}^{\log n} \frac{n}{2^h} O(h) \\ &= n \cdot c \sum_{h=1}^{\log n} \frac{h}{2^h} \\ &= O(n)\end{aligned}$$



As an exercise (using knowledge from your JEE preparation days), show that $\sum_{h=1}^{\log n} \frac{h}{2^h}$ is bounded by 2

Sorting using a Binary heap

Sorting using heap

Build heap **H** on the given n elements;

While (**H** is not empty)

```
{   $x \leftarrow \text{Extract-min}(\mathbf{H})$ ;  
  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: Next programming assignment 😊

Binary trees: beyond searching and sorting

- **Elegant solution for two interesting problem**
- **An important **lesson**:**

Lack of **proper understanding** of a problem is a big hurdle to solve the problem

Two interesting problems on sequences

What is a sequence ?

A sequence $\mathbf{S} = \langle x_0, \dots, x_{n-1} \rangle$

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

Problem 1

Multi-increment

Problem 1

Given an initial sequence $S = \langle x_0, \dots, x_{n-1} \rangle$ of numbers, maintain a compact data structure to perform the following operations:

- **ReportElement**(i):
Report the current value of x_i .
 - **Multi-Increment**(i, j, Δ):
Add Δ to each x_k for each $i \leq k \leq j$
-

Example:

Let the initial sequence be $S = \langle 14, 12, 23, 12, 111, 51, 321, -40 \rangle$

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

$\langle 14, 12, 33, 22, 121, 61, 331, -40 \rangle$

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

$\langle 39, 37, 58, 47, 146, 61, 331, -40 \rangle$

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

$\langle 39, 37, 89, 78, 177, 92, 331, -40 \rangle$

Problem 1

Given an initial sequence $S = \langle x_0, \dots, x_{n-1} \rangle$ of numbers, maintain a compact data structure to perform the following operations:

- **ReportElement**(i):
Report the current value of x_i .
 - **Multi-Increment**(i, j, Δ):
Add Δ to each x_k for each $i \leq k \leq j$
-

Trivial solution :

Store S in an array $A[0..n-1]$ such that $A[i]$ stores the current value of x_i .

Multi-Increment(i, j, Δ)

```
{  
    For ( $i \leq k \leq j$ )     $A[k] \leftarrow A[k] + \Delta$ ;  
}
```

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

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

} $O(1)$

Problem 1

Given an initial sequence $S = \langle x_0, \dots, x_{n-1} \rangle$ of numbers, maintain a compact data structure to perform the following operations:

- **ReportElement(i):**
Report the current value of x_i .
 - **Multi-Increment(i, j, Δ):**
Add Δ to each x_k for each $i \leq k \leq j$
-

Trivial solution :

Store S in an array $A[0.. n-1]$ such that $A[i]$ stores the current value of x_i .

Question: the source of difficulty in breaking the $O(n)$ barrier for **Multi-Increment()** ?

Answer: we need to explicitly maintain 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

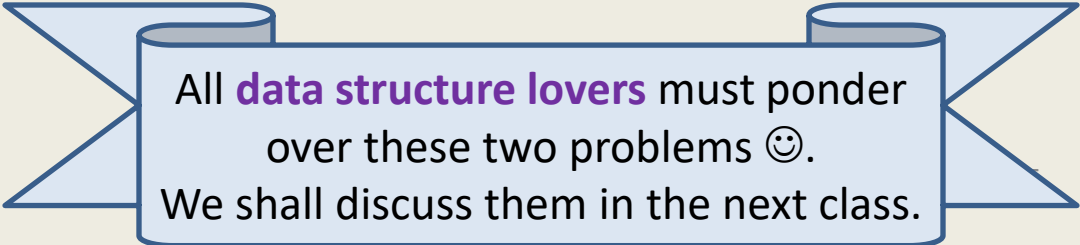
Problem 2

Given an initial sequence $S = \langle x_0, \dots, x_{n-1} \rangle$ of numbers, maintain a compact data structure to perform the following operations efficiently for any $0 \leq i < j < n$.

- **ReportMin**(i, j):
Report the minimum element from $\{x_k \mid \text{for each } i \leq k \leq j\}$
- **Update**(i, a):
 a becomes the new value of x_i .

AIM:

- $O(n)$ size data structure.
- **ReportMin**(i, j) in $O(\log n)$ time.
- **Update**(i, a) in $O(\log n)$ time.



All **data structure lovers** must ponder over these two problems 😊.
We shall discuss them in the next class.