

CS2351

Data Structures

Lecture 4: Heap

About this lecture

- Introduce **Heap**
 - Shape Property and Heap Property
 - Heap Operations
- **Heapsort**: Use Heap to Sort
- Fixing heap property for all nodes
- Use **Array** to represent Heap
- Introduce **Priority Queue**

Heap

A **heap** (or **binary heap**) is a **binary tree** that satisfies both:

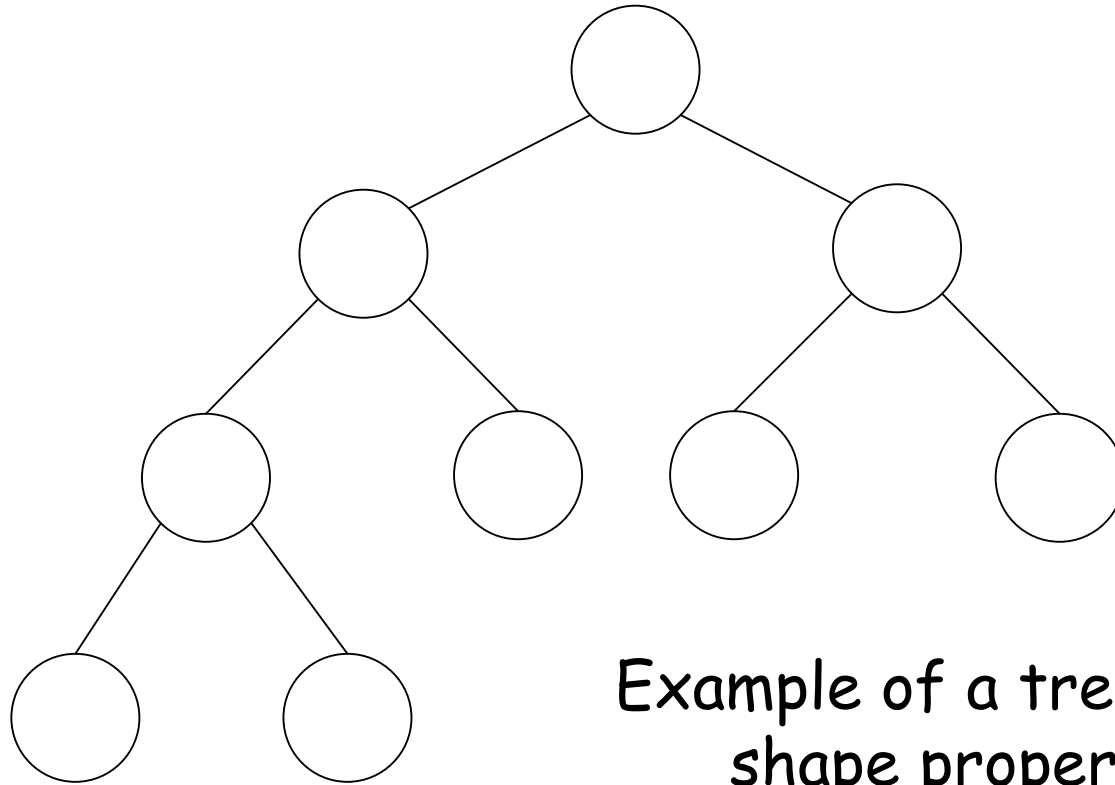
(1) **Shape Property**

- All levels, except deepest, are fully filled
- Deepest level is filled from left to right

(2) **Heap Property**

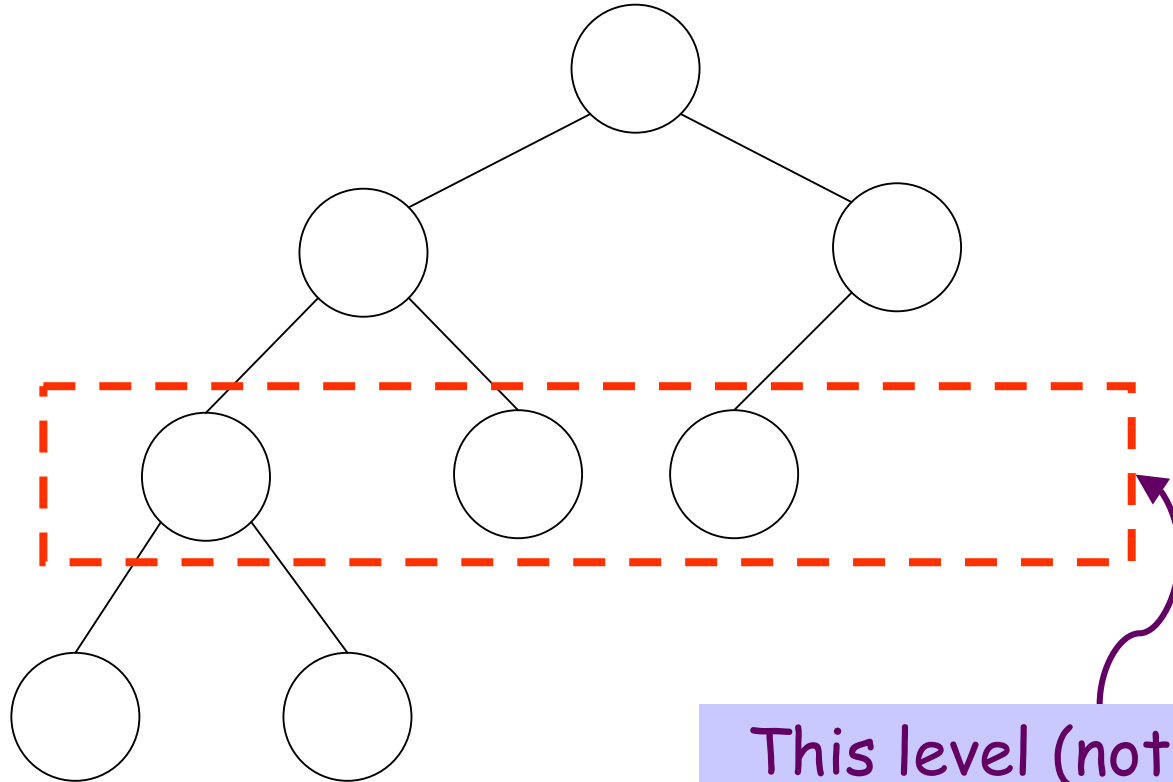
- Value of a node \leq Value of its children

Satisfying Shape Property



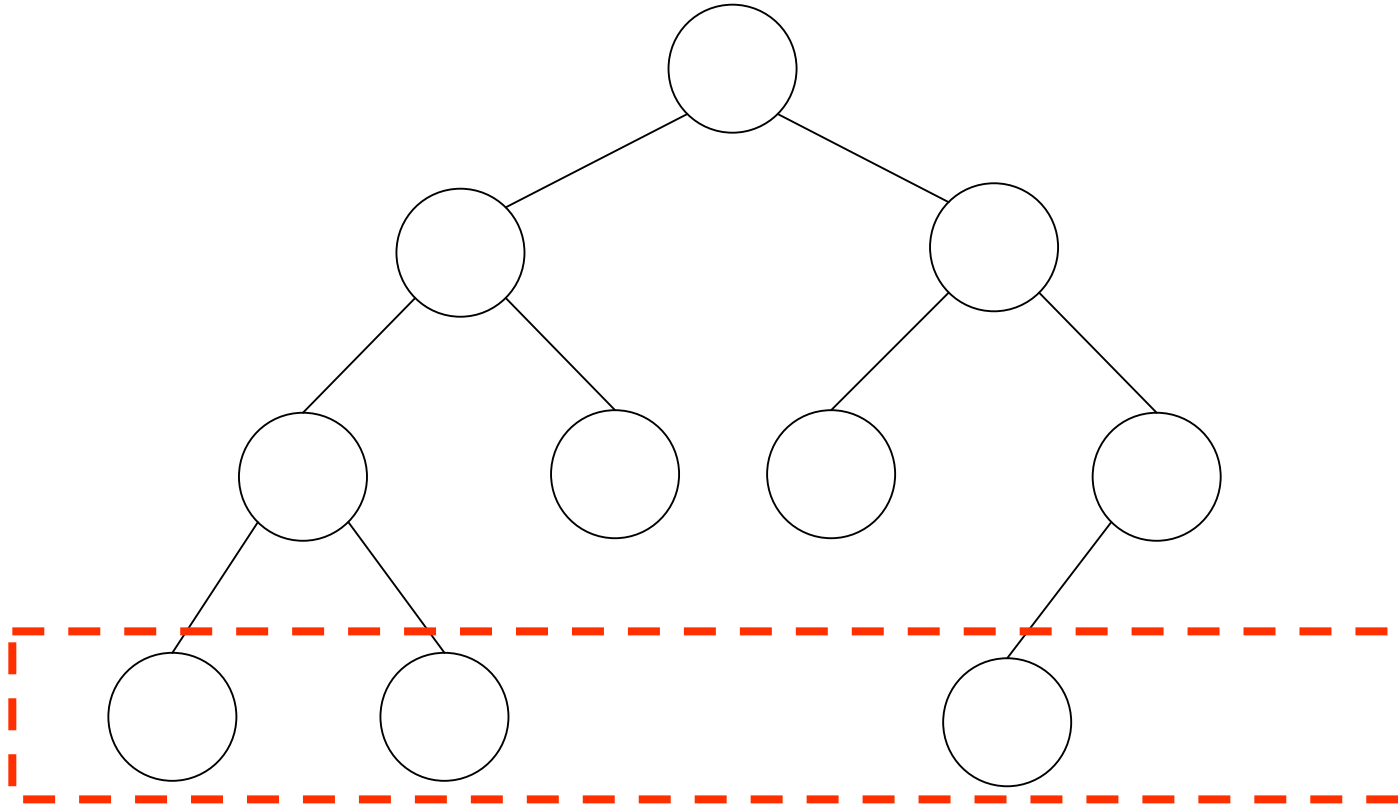
Example of a tree with
shape property

Not Satisfying Shape Property



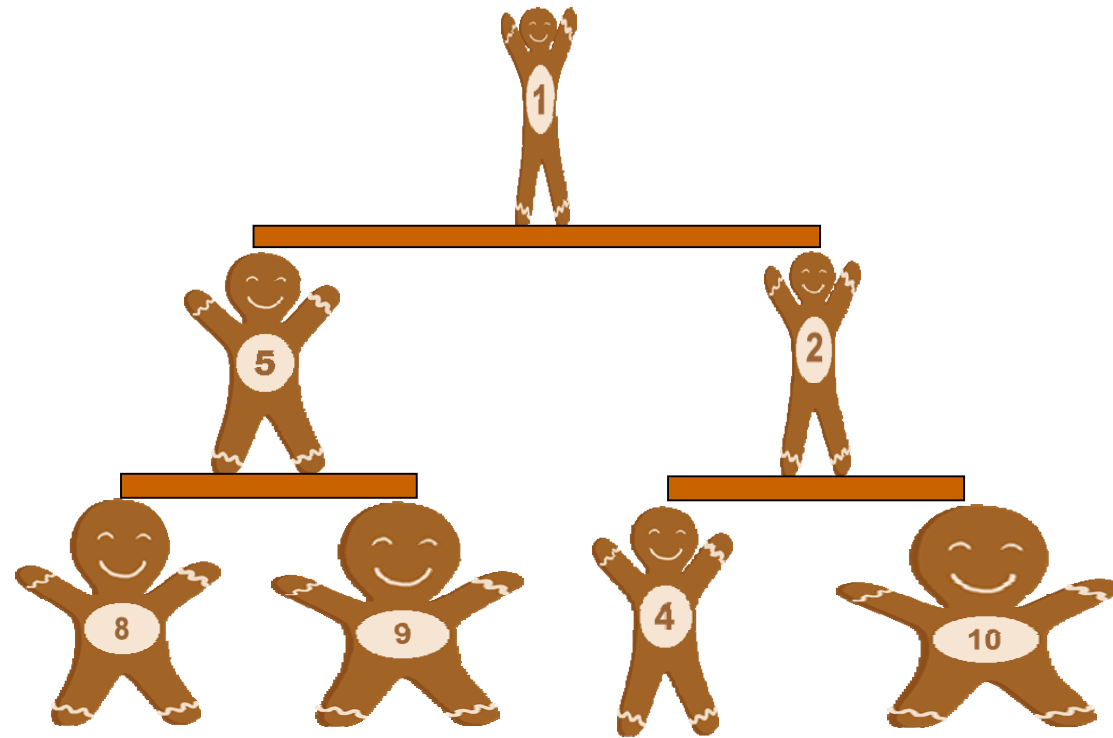
This level (not deepest)
is NOT fully filled

Not Satisfying Shape Property



Deepest level NOT
filled from left to right

Satisfying Heap Property



Min Heap

Q. Given a heap, what is so special about the root's value?

A. ... always the minimum

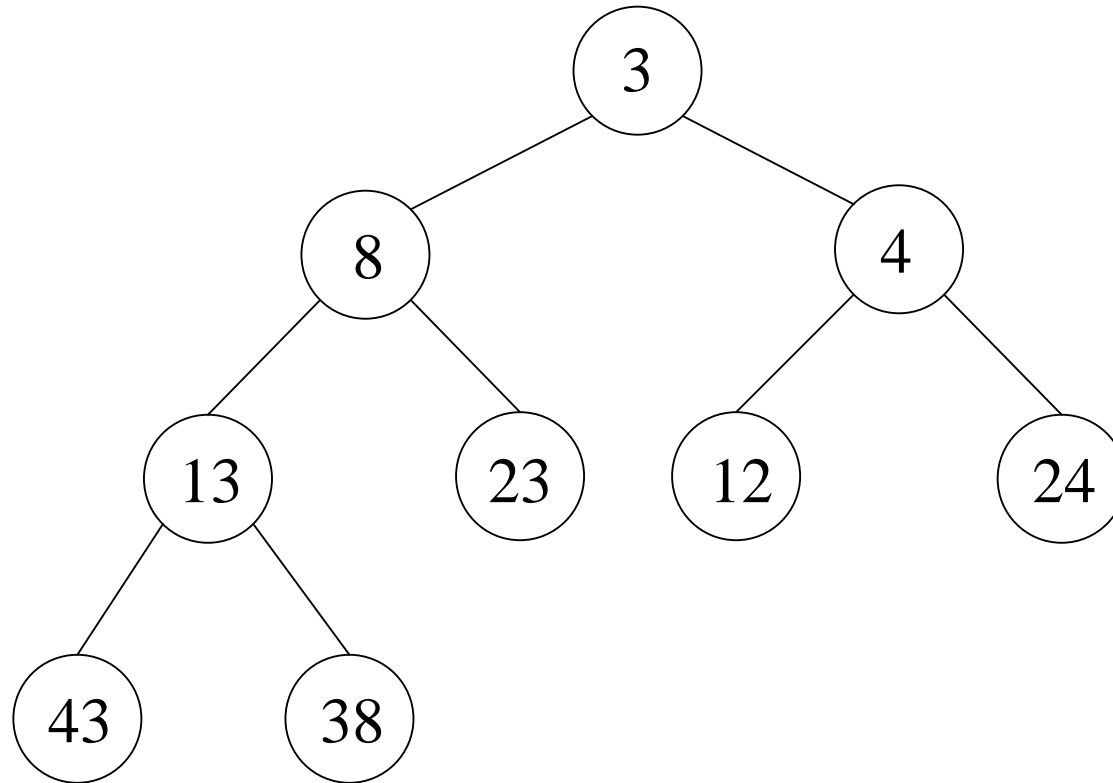
Because of this, the previous heap is also called a **min heap**

Heap Operations

- **Find-Min** : find the minimum value
→ $\Theta(1)$ time
- **Extract-Min** : delete the minimum value
→ $O(\log n)$ time (how??)
- **Insert** : insert a new value into heap
→ $O(\log n)$ time (how??)

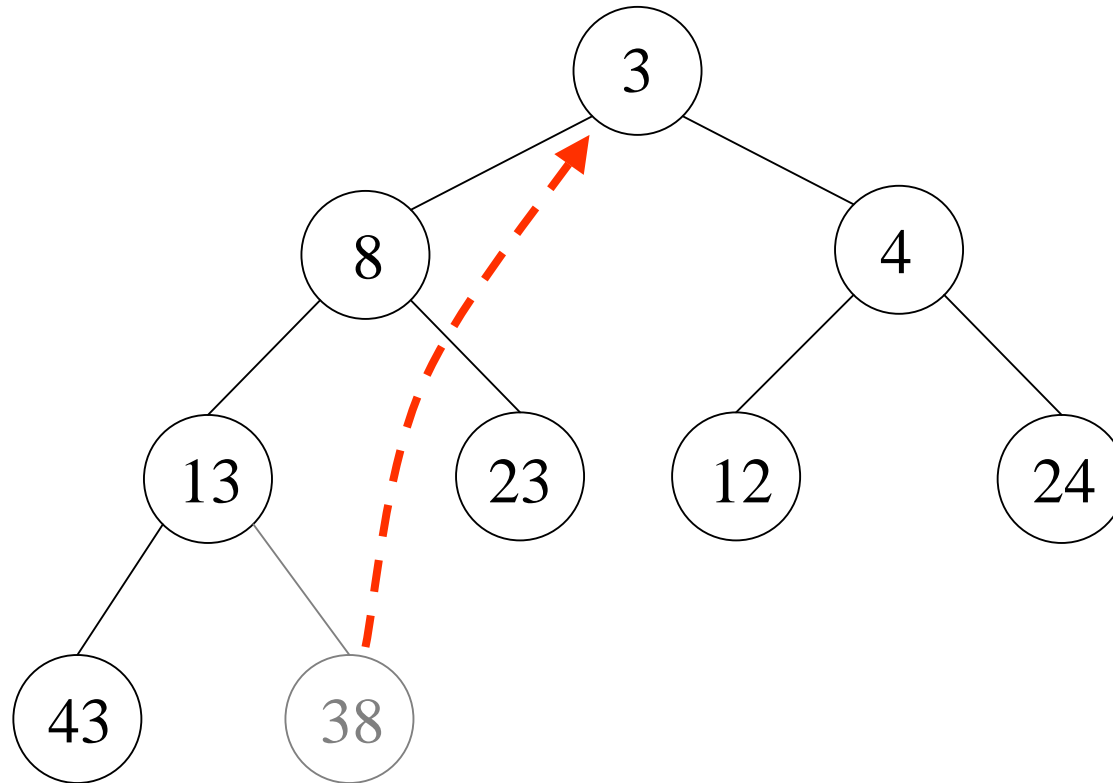
$n = \#$ nodes in the heap

How to do Extract-Min?



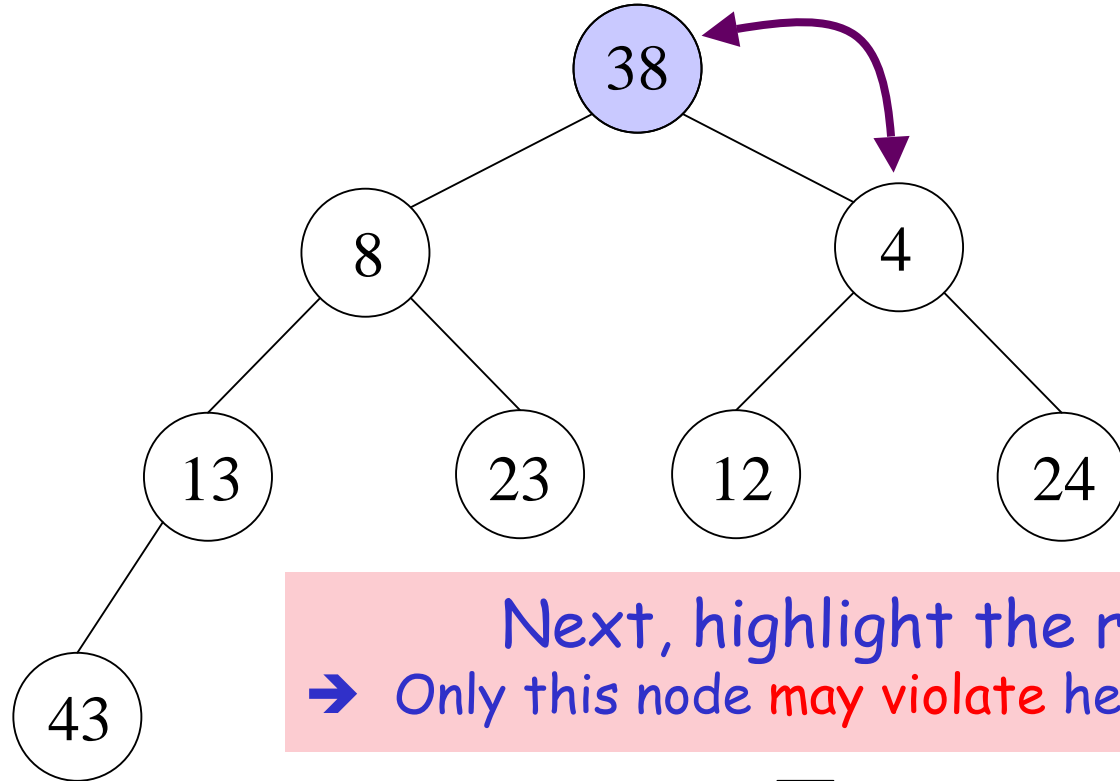
Heap before Extract-Min

Step 1: Restore Shape Property

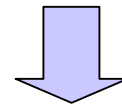


Copy value of last node to root.
Next, remove last node

Step 2: Restore Heap Property

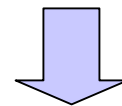
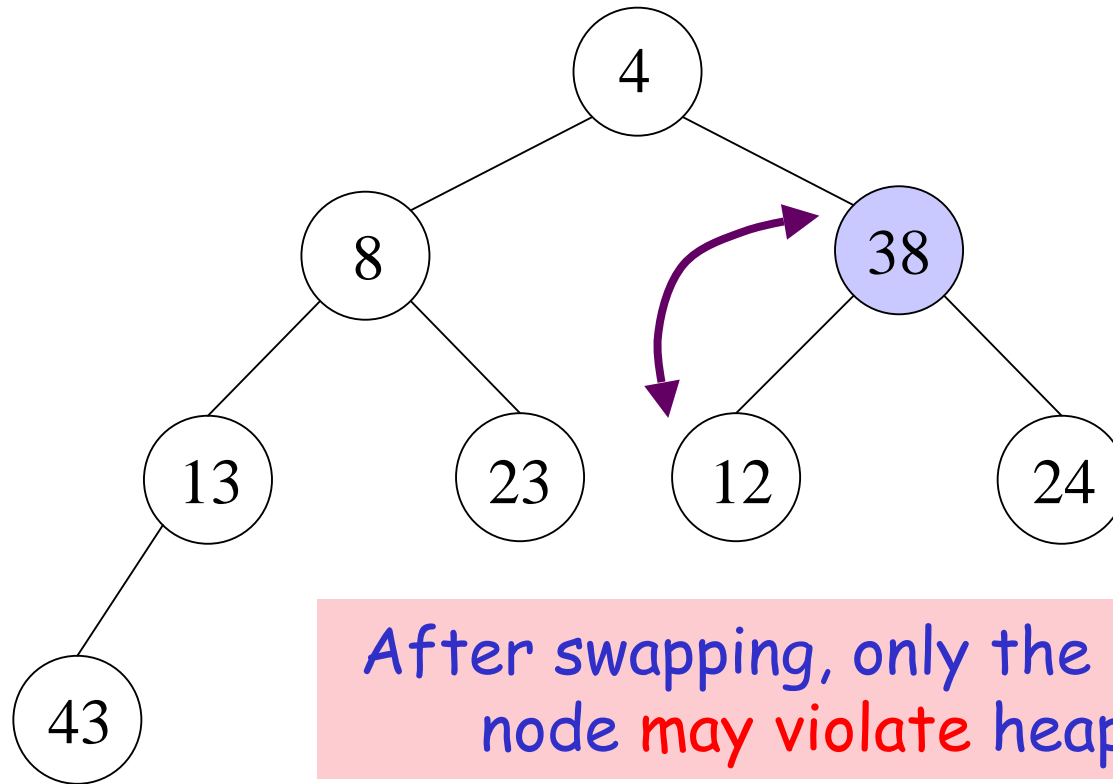


Next, highlight the root
→ Only this node **may violate** heap property



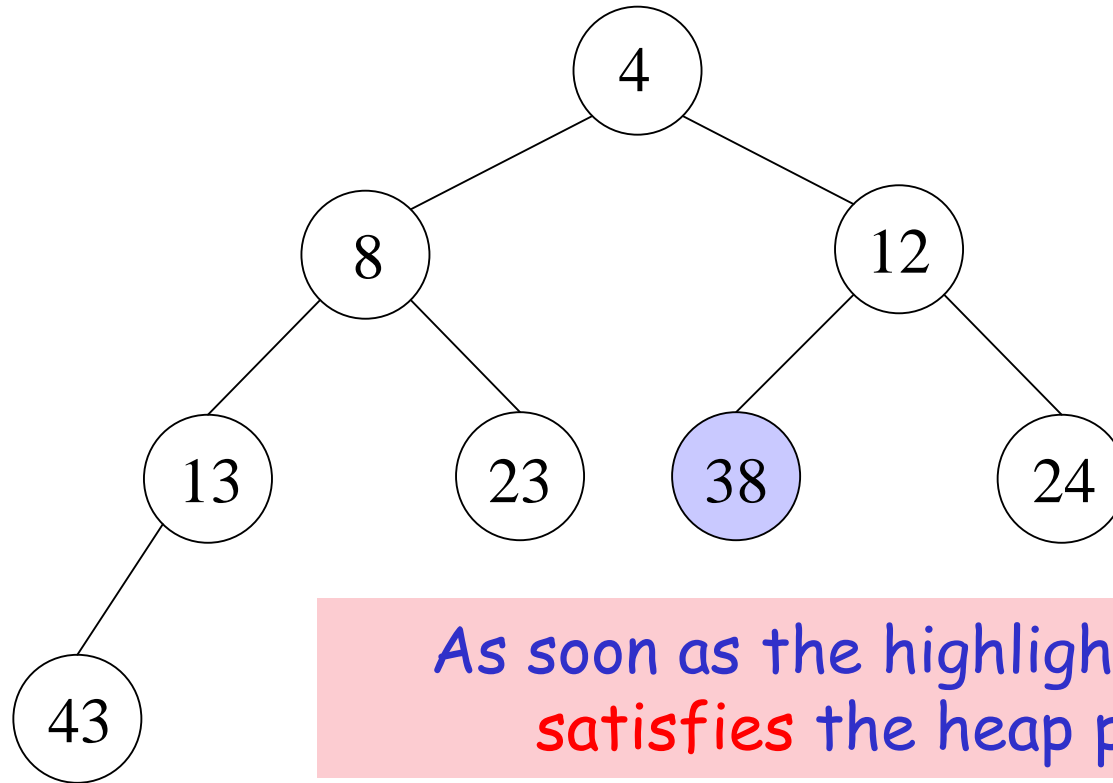
If violates, swap highlighted node with "smaller" child
(if not, everything done)

Step 2: Restore Heap Property

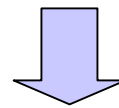


If violates, swap highlighted node with "smaller" child
(if not, everything done)

Step 2: Restore Heap Property

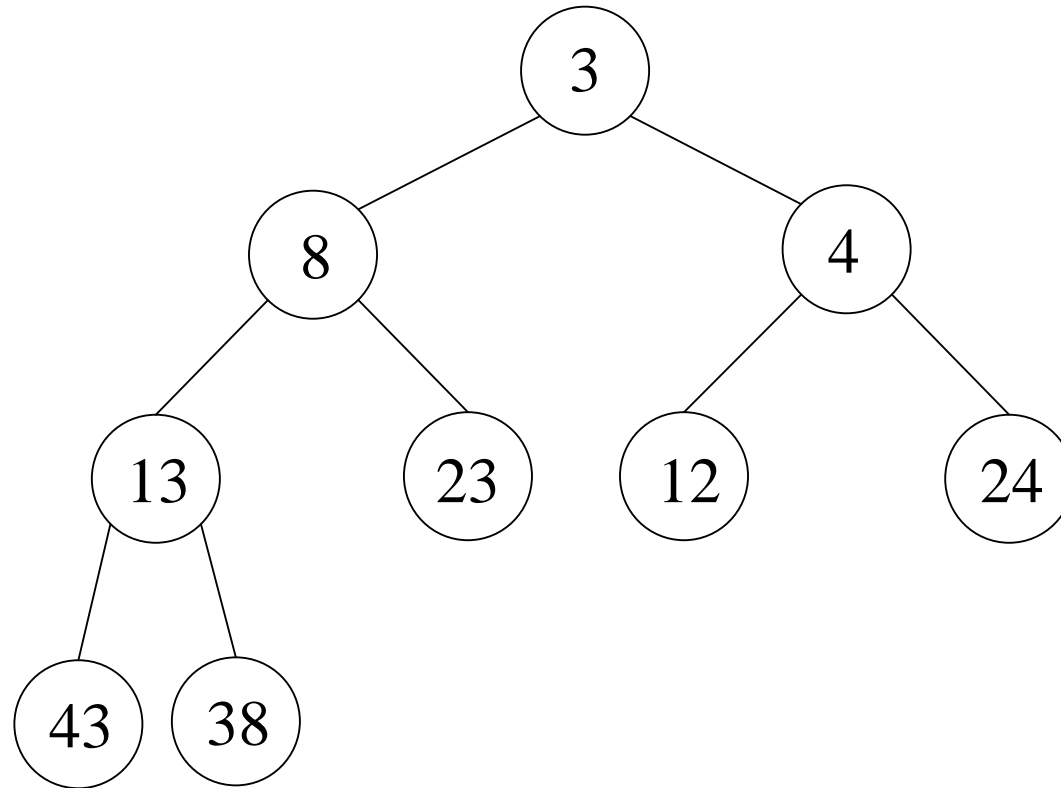


As soon as the highlighted node satisfies the heap property



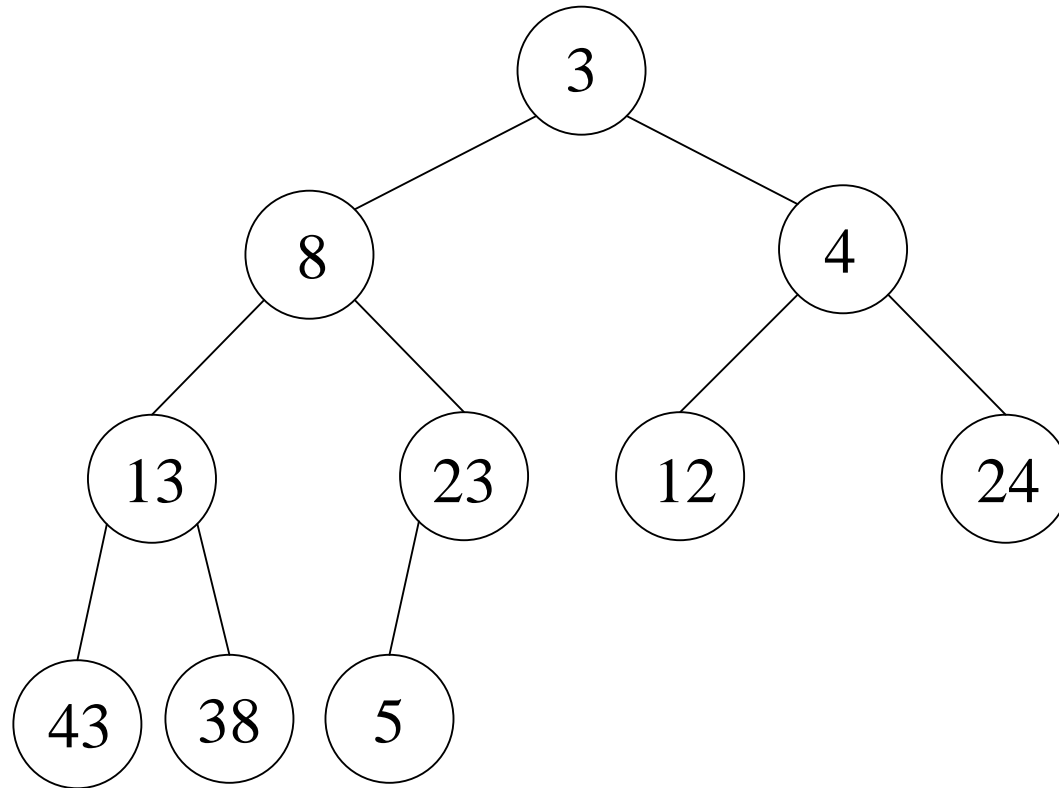
Everything done !!!

How to do Insert?



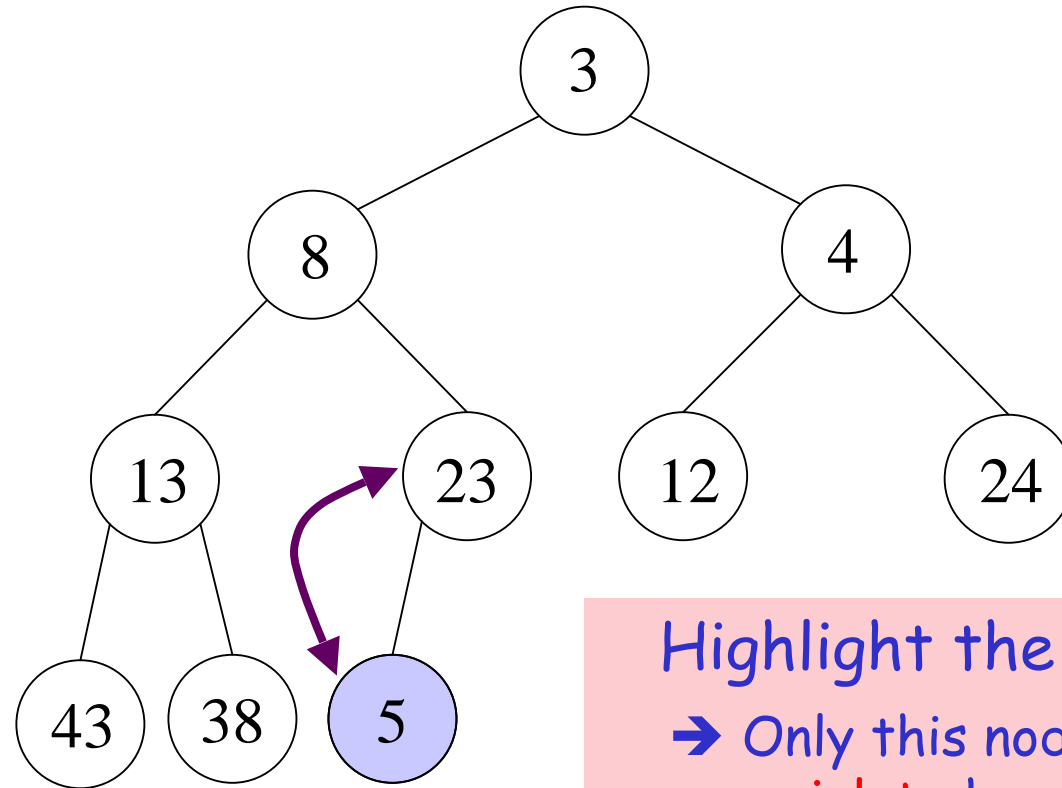
Heap before Insert

Step 1: Restore Shape Property

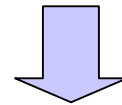


Create a new node with the new value.
Next, add it to the heap at correct position

Step 2: Restore Heap Property

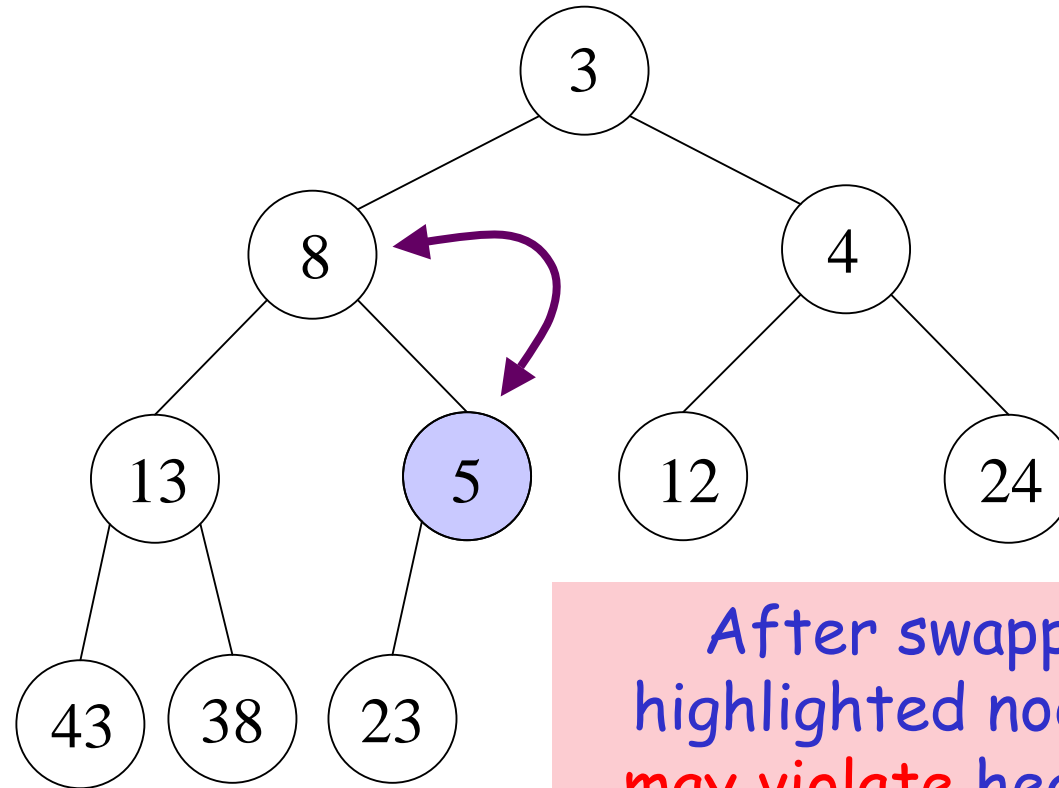


Highlight the new node
→ Only this node's parent
may violate heap property

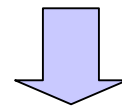


If violates, swap highlighted node with parent
(if not, everything done)

Step 2: Restore Heap Property

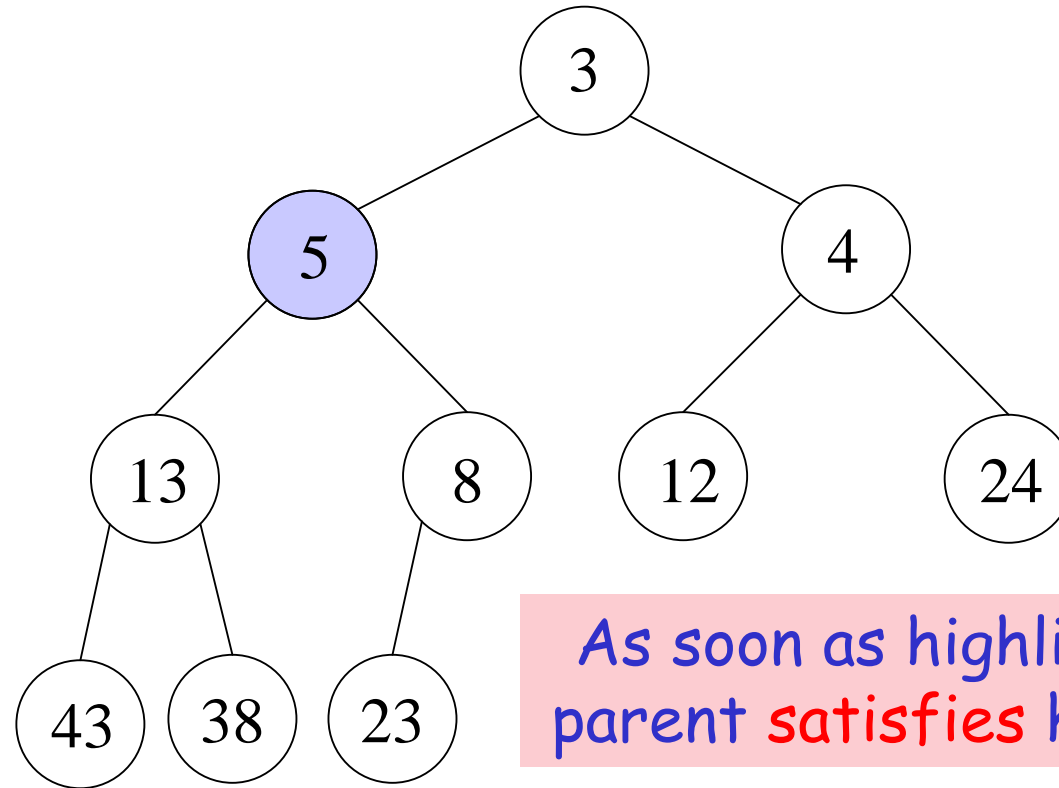


After swapping, only highlighted node's parent may violate heap property

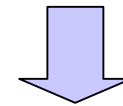


If violates, swap highlighted node with parent
(if not, everything done)

Step 2: Restore Heap Property



As soon as highlighted node's parent **satisfies** heap property



Everything done !!!

Running Time

Let h = node-height of heap

- Both **Extract-Min** and **Insert** require $O(h)$ time to perform

Since $h = \Theta(\log n)$ (why??)

→ Both require $O(\log n)$ time

n = # nodes in the heap

Heapsort

Q. Given n numbers, can we use heap to sort them, say, in ascending order?

A. Yes, and extremely easy !!!

1. Call **Insert** to insert n numbers into heap
2. Call **Extract-Min** n times
→ numbers are output in sorted order

Runtime: $n \times O(\log n) + n \times O(\log n) = O(n \log n)$

This sorting algorithm is called **heapsort**

Challenge

(Fixing heap property for all nodes)

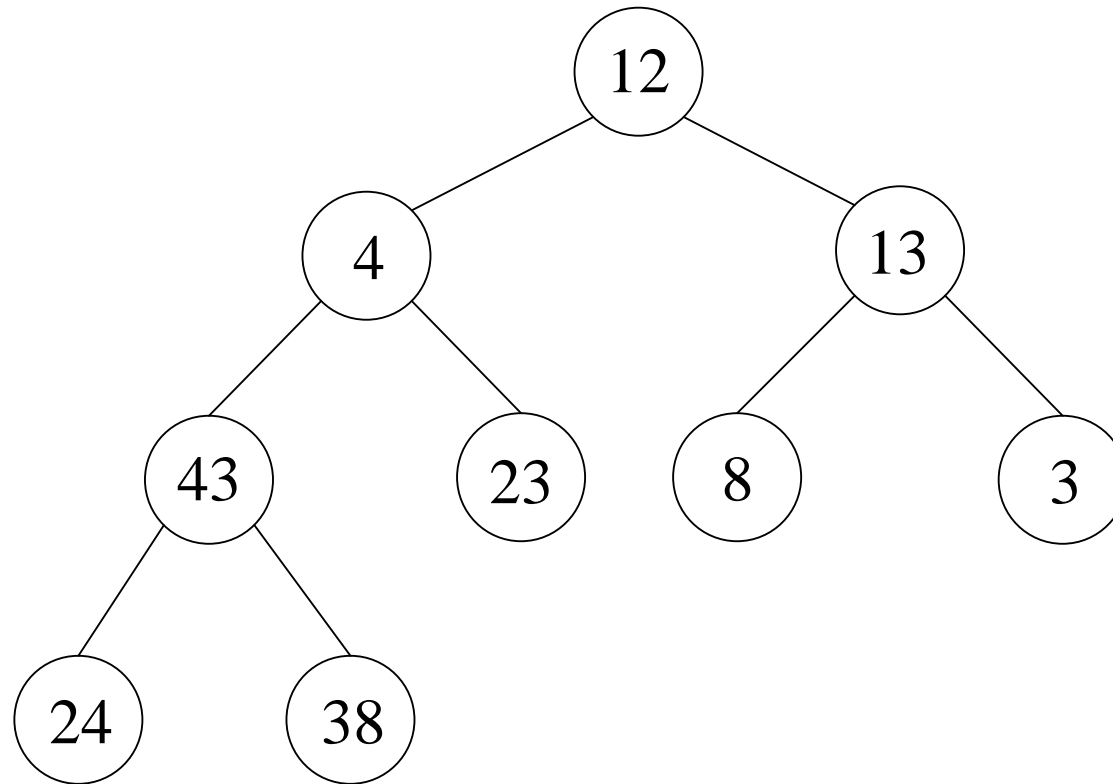
Suppose that we are given a binary tree which satisfies the shape property

However, the **heap** property of the nodes may not be satisfied ...

Question: Can we make the tree into a heap in $O(n)$ time?

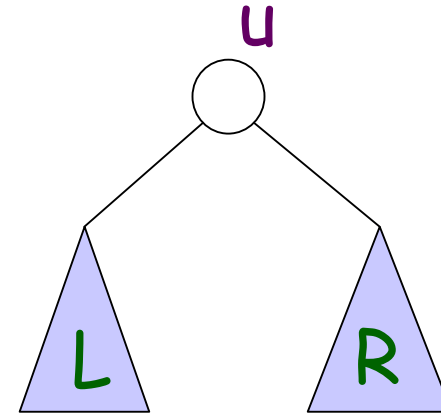
$n = \#$ nodes in the tree

How to make it a heap?



Observation

- u = root of a binary tree
- L = subtree rooted at u 's left child
- R = subtree rooted at u 's right child



Obs: If L and R satisfy heap property, we can make the tree rooted at u satisfy heap property in $O(\max\{\text{height}(L), \text{height}(R)\})$ time.

We denote the above operation by **Heapify**(u)

Heapify

Then, for any tree T , we can make T satisfy the heap property as follows:

Step 1. $h = \text{node_height}(T)$;

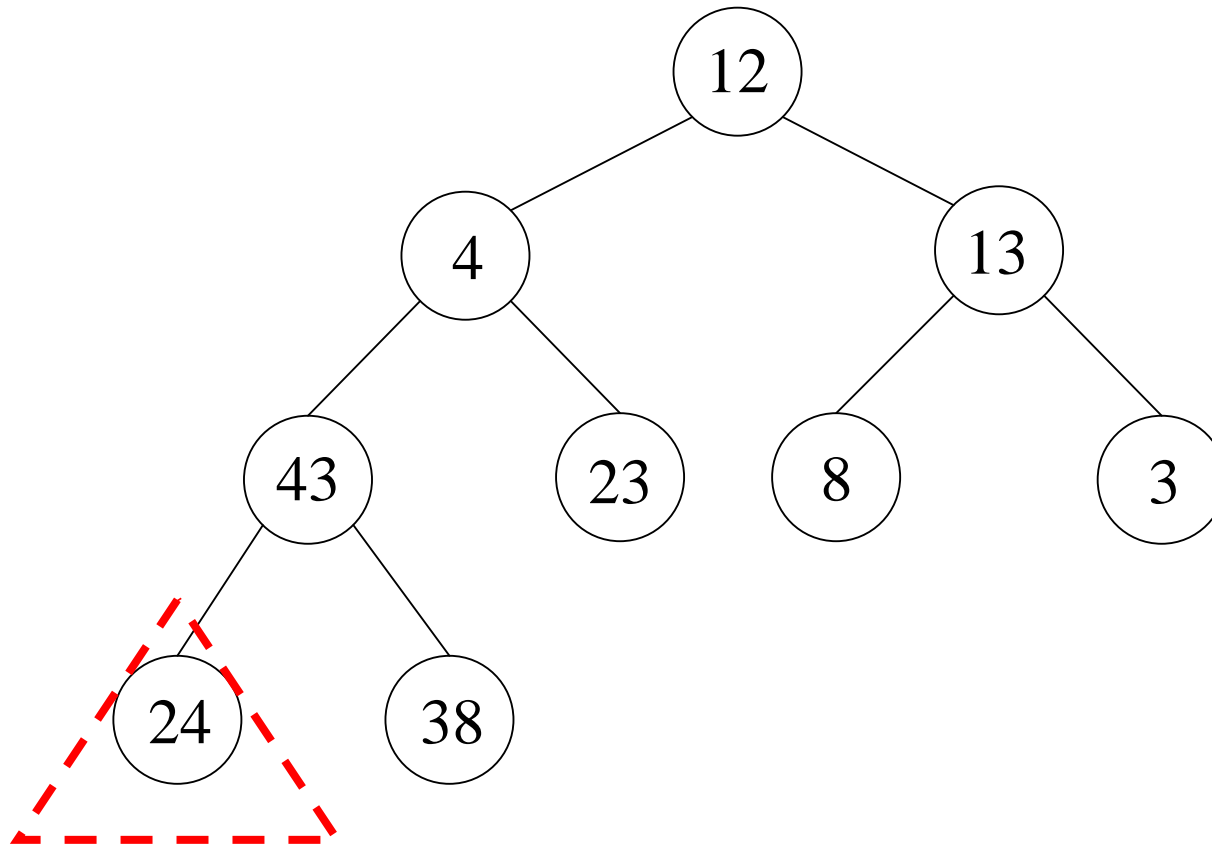
Step 2. for $k = h, h-1, \dots, 1$

 for each node u at level k

 Heapify(u) ;

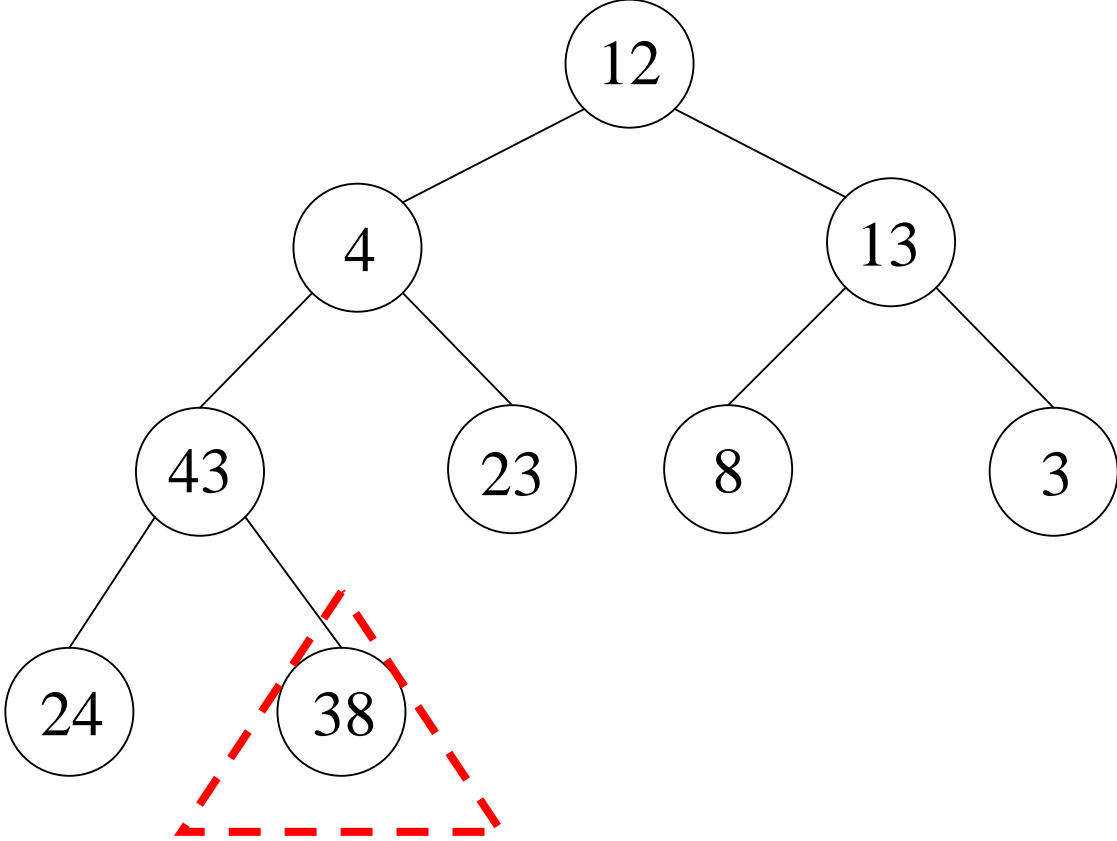
Why is the above algorithm correct?

Example Run



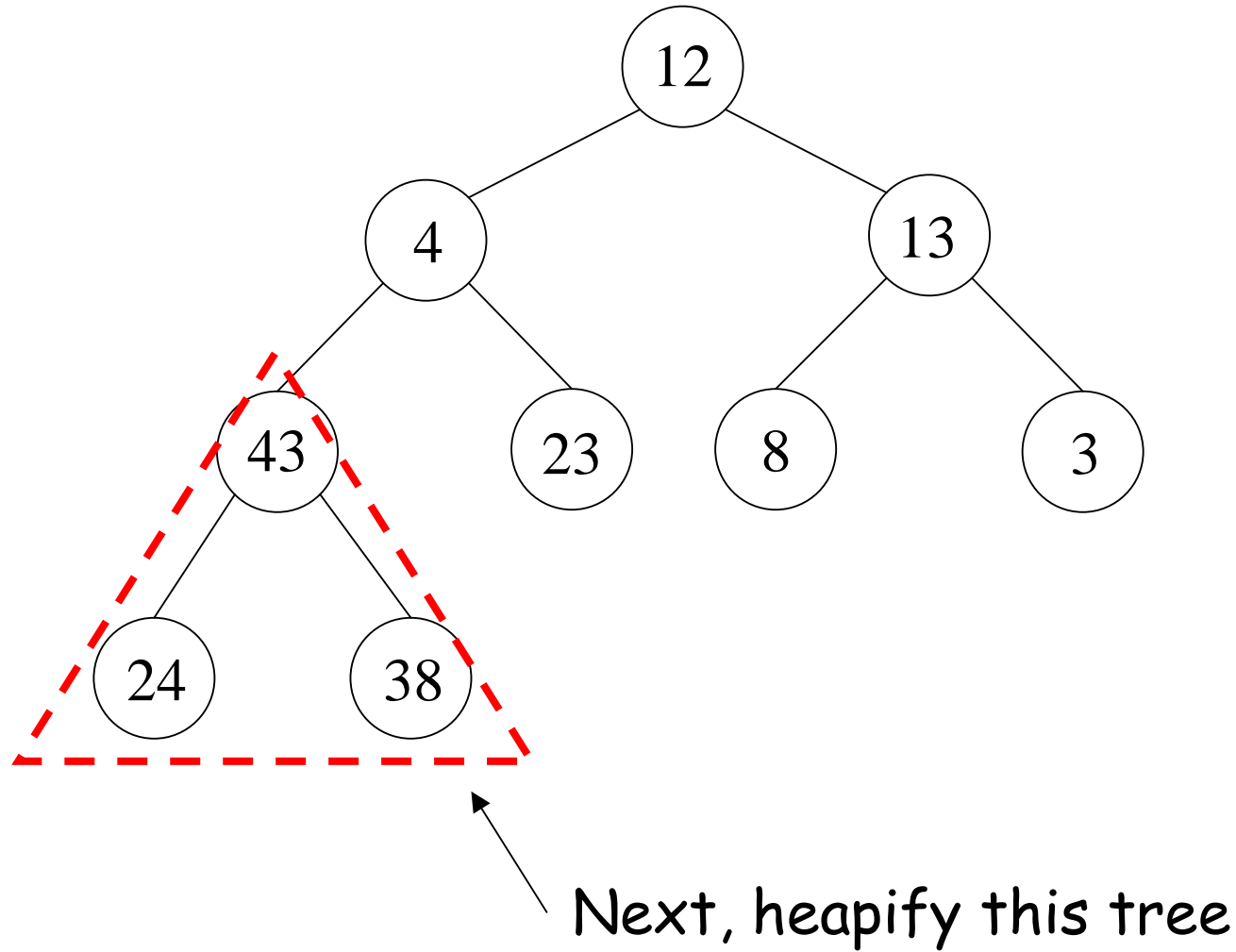
First, heapify this tree

Example Run

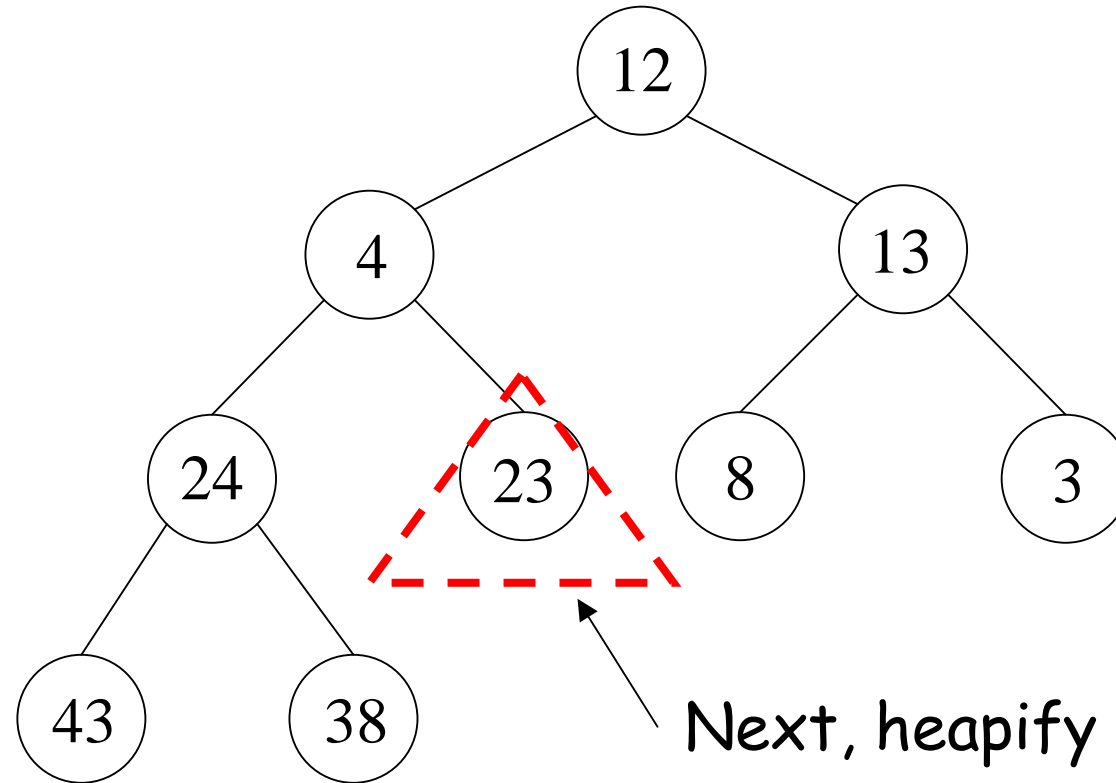


Next, heapify this tree

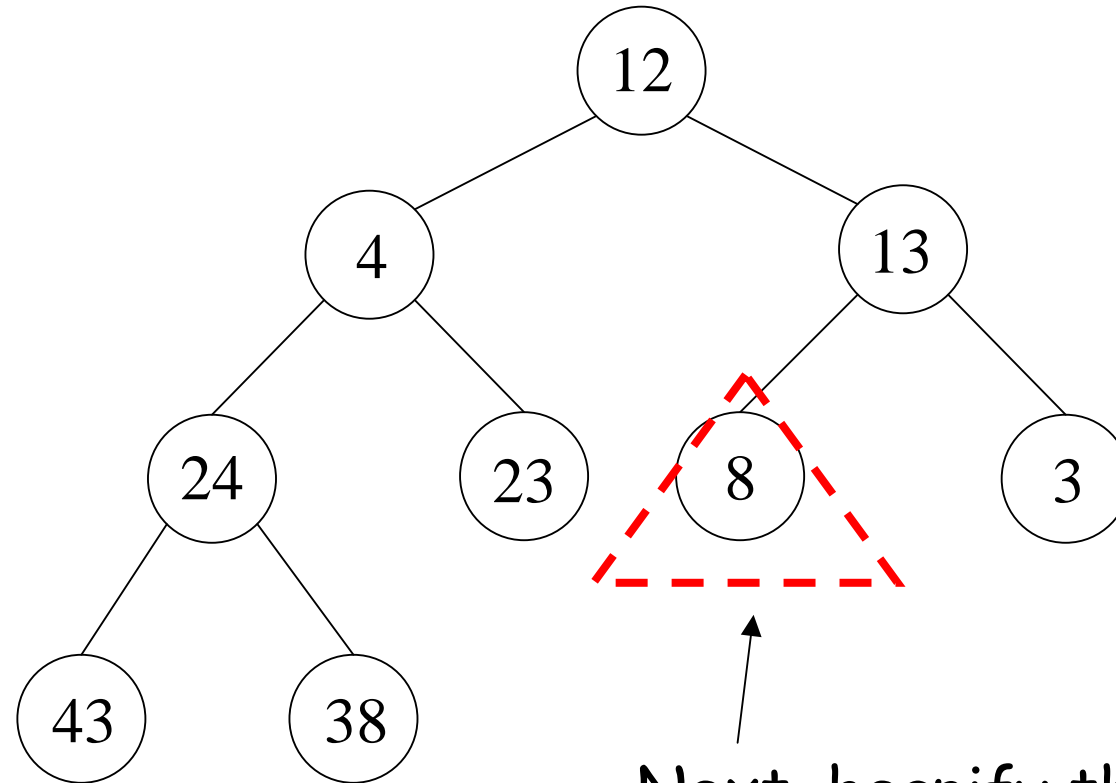
Example Run



Example Run

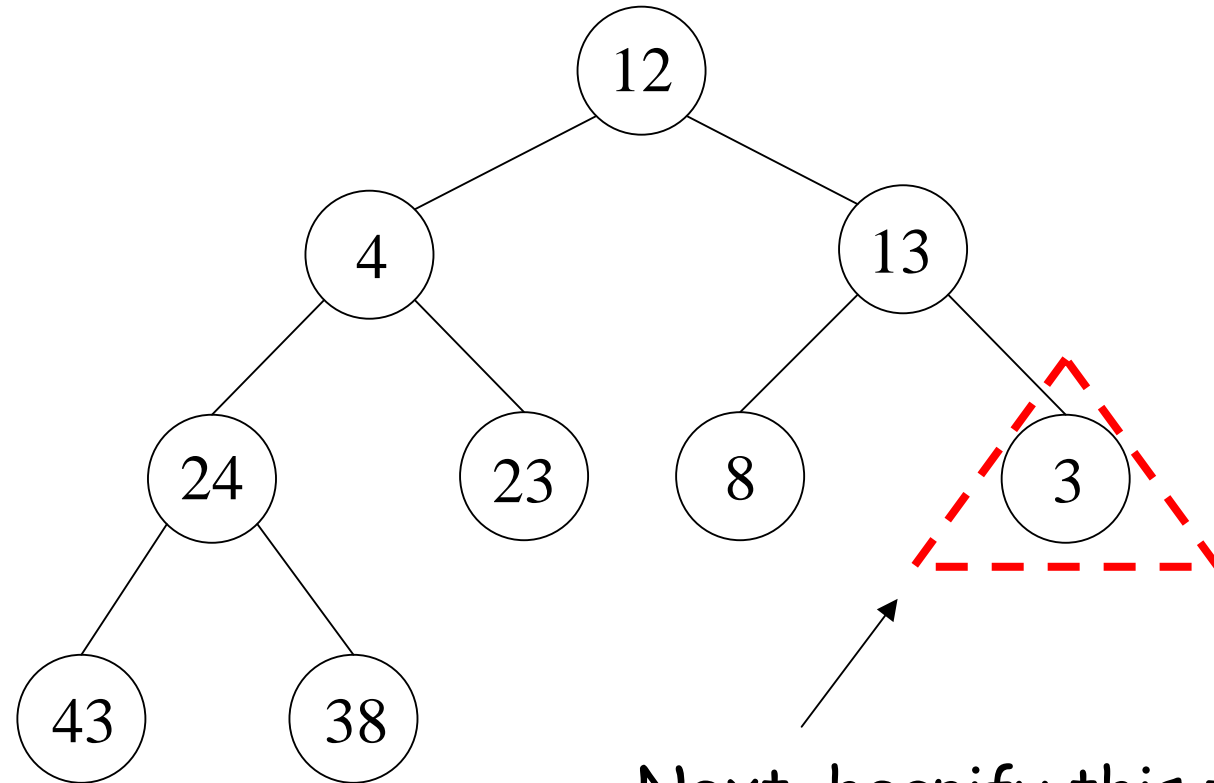


Example Run



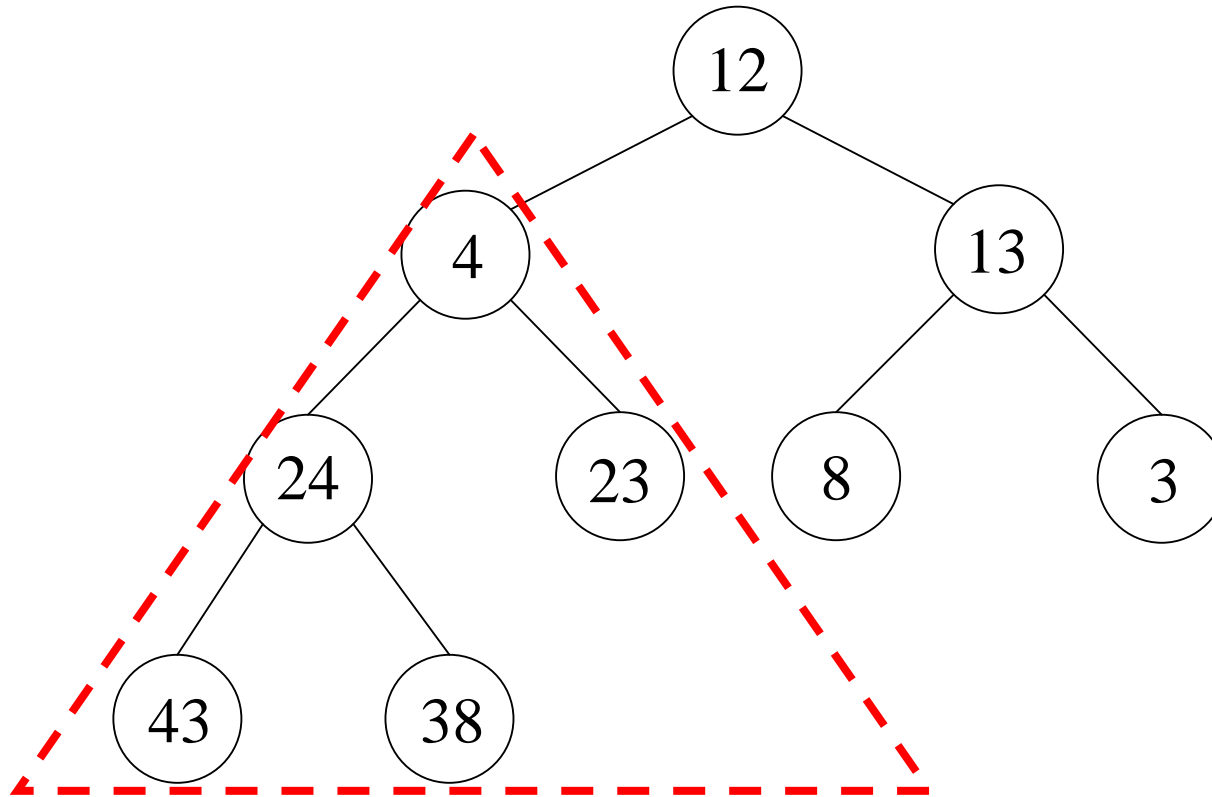
Next, heapify this tree

Example Run



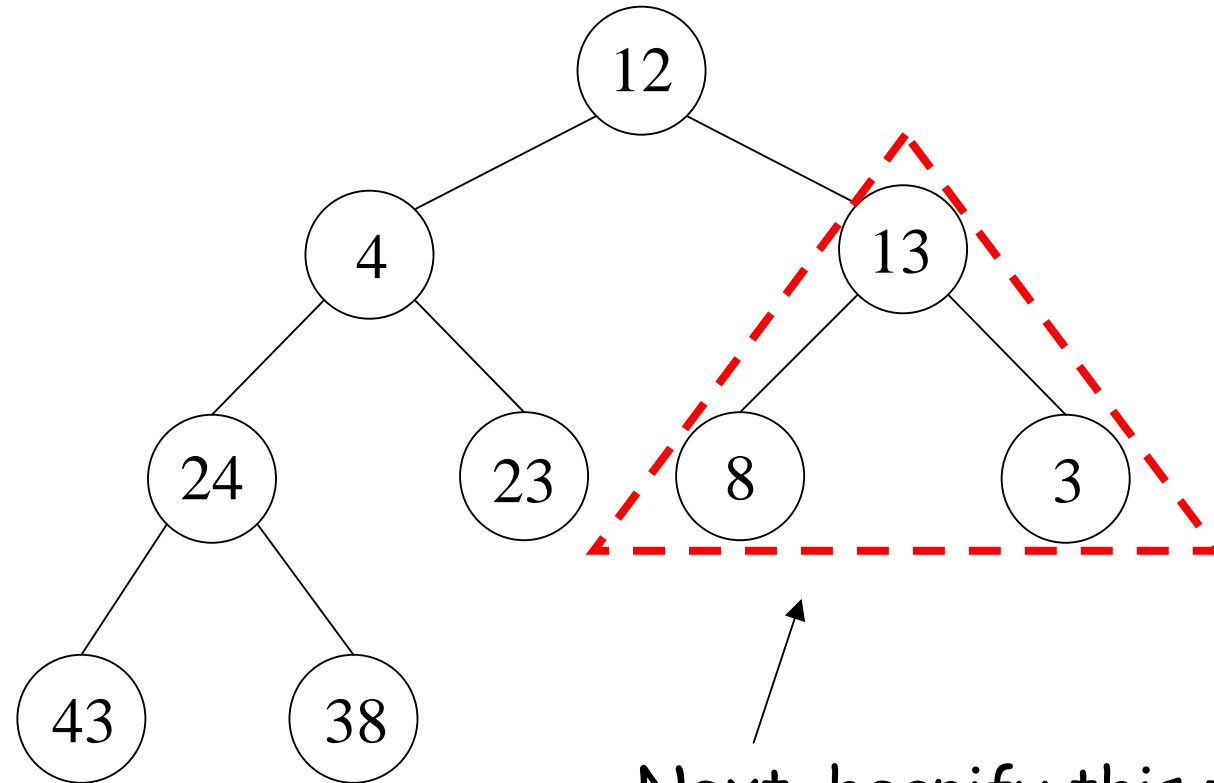
Next, heapify this tree

Example Run



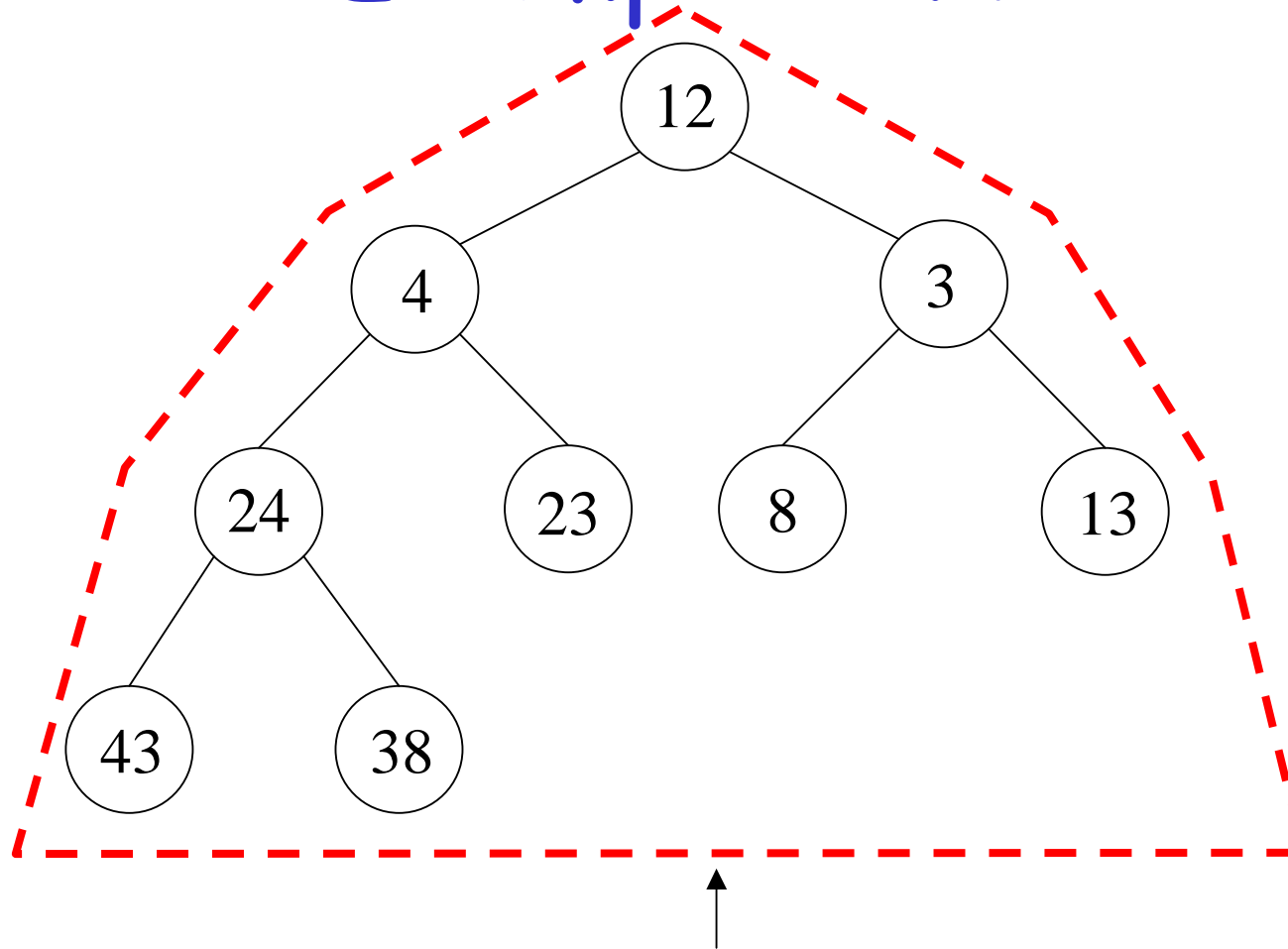
Next, heapify this tree

Example Run



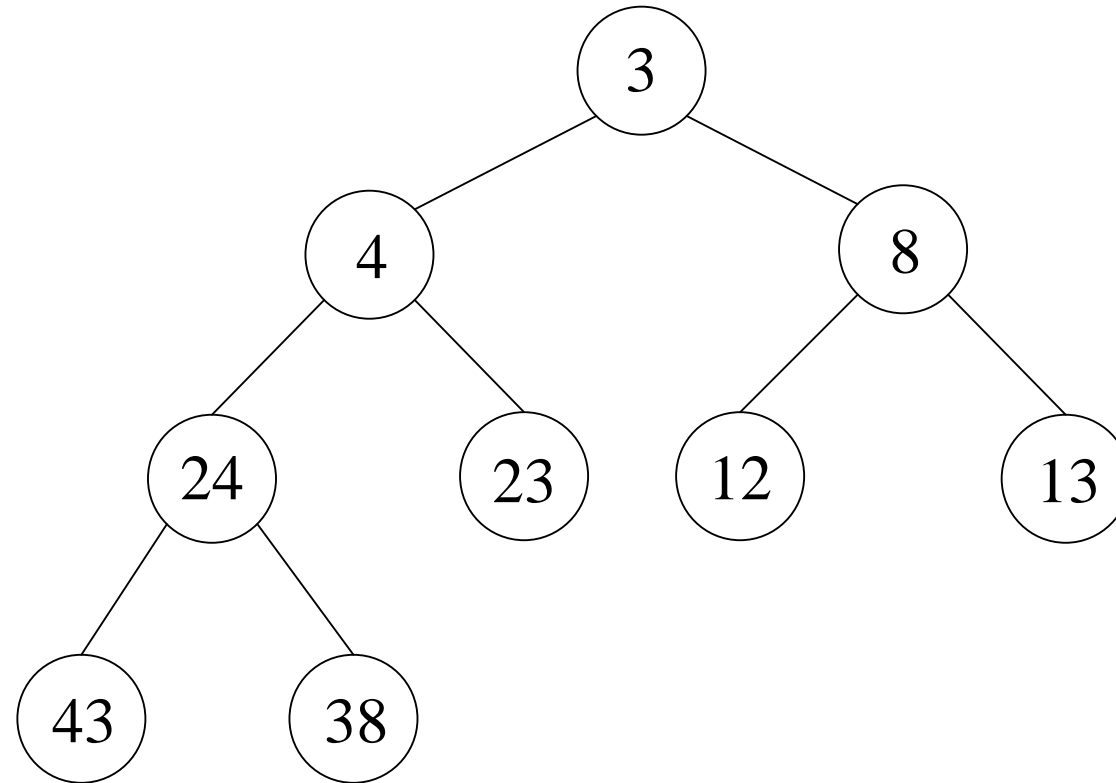
Next, heapify this tree

Example Run



Finally, heapify the whole tree

Example Run



Everything Done !

Back to the Challenge

(Fixing heap property for all nodes)

Suppose that we are given a binary tree which satisfies the shape property

However, the **heap** property of the nodes may not be satisfied ...

Question: Can we make the tree into a heap in $O(n)$ time?

$n = \#$ nodes in the tree

Back to the Challenge

(Fixing heap property for all nodes)

Let h = node-height of tree

So, $2^{h-1} \leq n \leq 2^h - 1$ (why??)

For a tree with shape property,

at most 2^{h-1} nodes at level h ,

exactly 2^{h-2} nodes at level $h-1$,

exactly 2^{h-3} nodes at level $h-2$, ...

Back to the Challenge

(Fixing heap property for all nodes)

Using the previous algorithm to solve the challenge, the total time is at most

$$\begin{aligned} & 2^{h-1} \times 1 + 2^{h-2} \times 2 + 2^{h-3} \times 3 + \dots + 1 \times h \quad [\text{why??}] \\ &= 2^h \left(1 \times \frac{1}{2} + 2 \times \left(\frac{1}{2}\right)^2 + 3 \times \left(\frac{1}{2}\right)^3 + \dots + h \times \left(\frac{1}{2}\right)^h \right) \\ &\leq 2^h \sum_{k=1 \text{ to } \infty} k \times \left(\frac{1}{2}\right)^k = 2^h \times 2 \leq 4n \end{aligned}$$

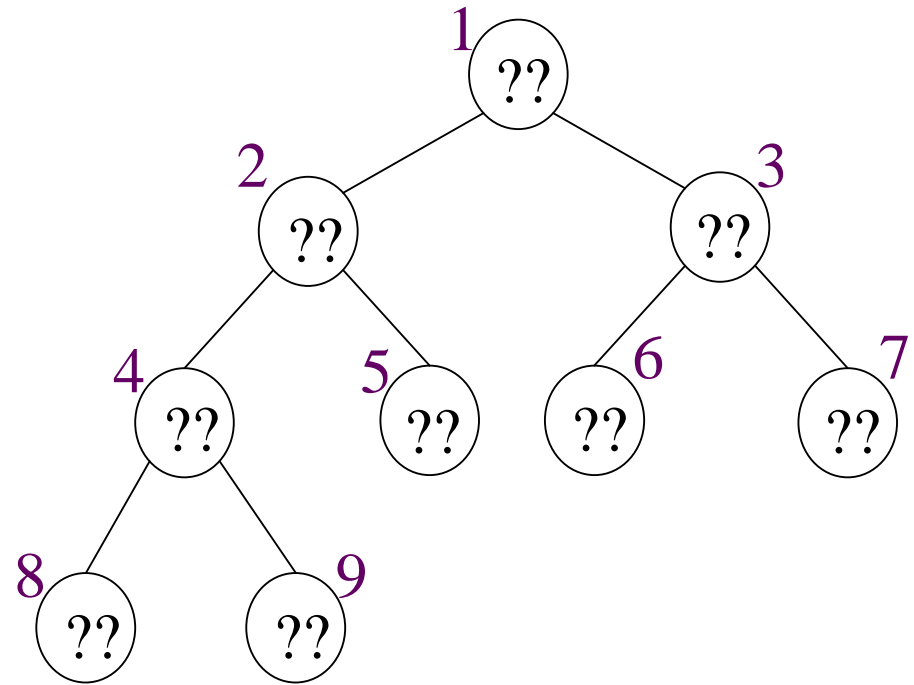
→ Thus, total time is $O(n)$

Array Representation of Heap

Given a heap.

Suppose we mark the position of root as **1**, and mark other nodes in a way as shown in the right figure. (BFS order)

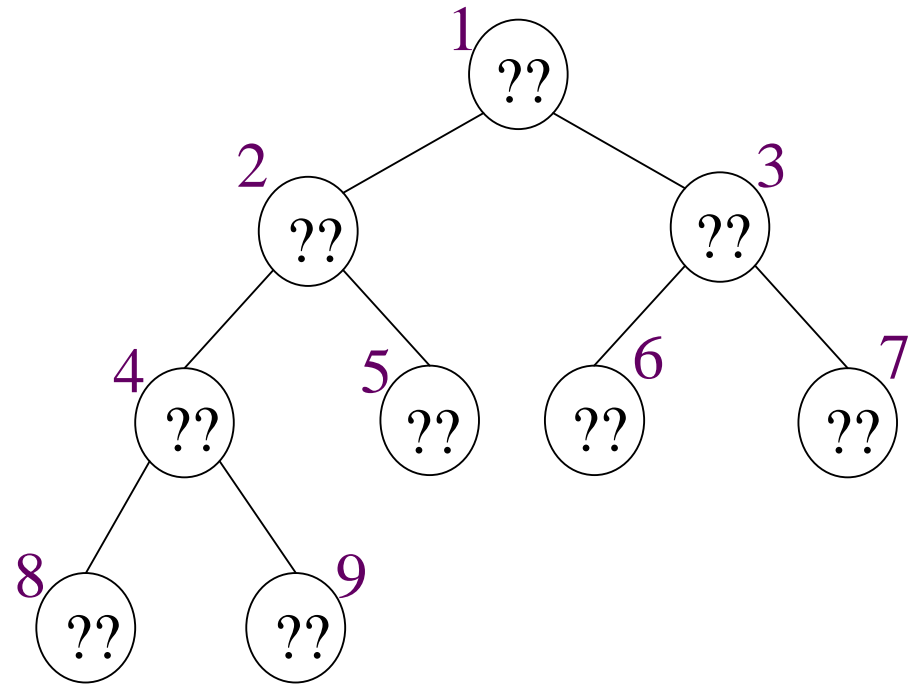
Anything special about this marking?



Array Representation of Heap

Yes, something special:

1. If the heap has n nodes, the marks are from 1 to n
2. Children of x , if exist, are $2x$ and $2x+1$
3. Parent of x is $\lfloor x/2 \rfloor$



Array Representation of Heap

- The special properties of the marking allow us to use an array $A[1..n]$ to store a heap of size n

Advantage:

Avoid storing or using tree pointers !!

Try this at home:

Write codes for **Insert** and **Extract-Min**, assuming the heap is stored in an array

Max Heap

We can also define a **max heap**, by changing the heap property to:

Value of a node \geq Value of its children

Max heap supports the following operations:

(1) Find Max, (2) Extract Max, (3) Insert

Do you know how to do these operations?

Priority Queue

Consider S = a set of items, each has a key

Priority queue on S supports:

Min(): return item with min key

Extract-Min(): remove item with min key

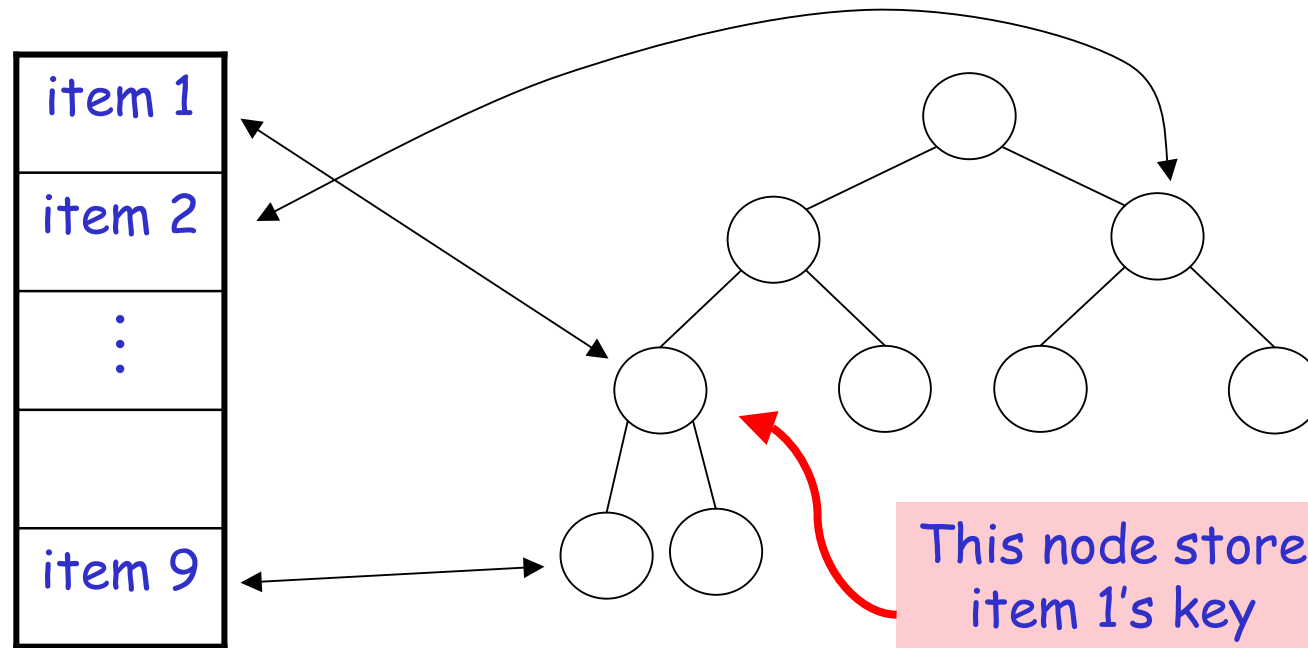
Insert(x,k): insert item x with key k

Decrease-Key(x,k): decrease key of x to k

Using Heap as Priority Queue

1. Store the items in an array
2. Use a **heap** to store keys of the items
3. Store links between an item and its key

E.g.,



Using Heap as Priority Queue

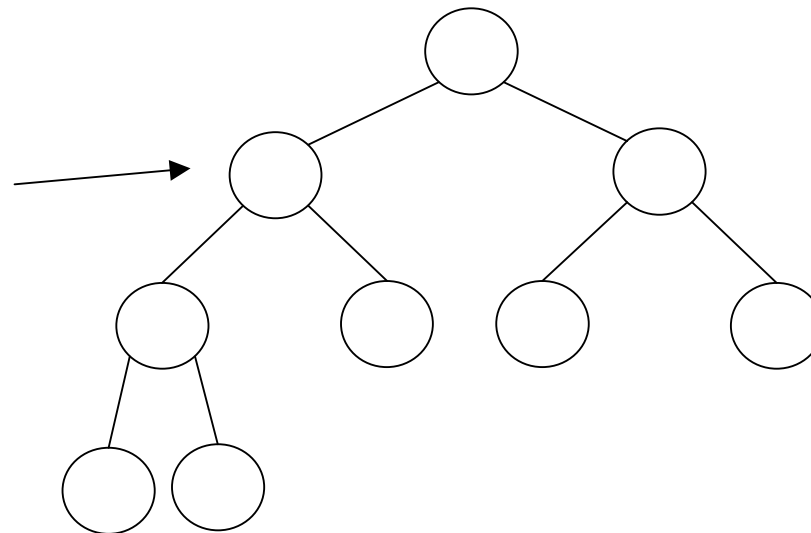
Previous scheme supports **Min** in $O(1)$ time,
Extract-Min and **Insert** in $O(\log n)$ time

It can support **Decrease-Key** in $O(\log n)$ time

E.g.,

Node storing key
value of item x

How do we decrease
the key to k ??



Other Schemes?

- In algorithm classes (or perhaps later lectures), we will look at other ways to implement a priority queue
 - with different time bounds for the operations

Remark: Priority Queue can be used for finding MST or shortest paths, and job scheduling