Priority Queue ADT

IT5003: Data Structures and Algorithms
(**AY2019/20 Semester 1**)

Lecture Outline

- Priority Queue ADT
  - Motivation
  - Specification
  - Implementation Choices

- Heap
  - Definition
  - Major Operations
  - Insertion/Deletion
  - Building heap from scratch
  - Heap sorting

Priority Queue: Overview

- Special form of a **Queue**:
  - Items are ordered based on **priority**
  - When we perform ***dequeue***:
    - Item with **highest priority** is always removed

- Real life example:
  - Emergency room of hospital:
    - Handle cases with higher priority first even they occurs later than other cases
  - Operating Systems:
    - Choose the task with highest priority when there are multiple tasks to execute

Priority Queue: **Operations**

- The major operations of a priority queue are:

Operation	Functionality
`insert( key, item )`	Add an `item` with `key` as the priority
`delete( ) → item`	Return the `item` with the highest priority and remove it from priority queue

- Compare and contrast with **Queue ADT** and **Table ADT**

Priority Queue: **Implementation Choices**

- Priority Queue is similar to Table ADT:
  - Insertion == same
  - Deletion == find **max key** and delete

- Balanced BST can be a good choice:
  - Insertion = **O(lg N)**
  - Deletion = find max + delete = **O(lg N)**

- However, can we do better by exploiting the **differences** between Priority Queue and Table ADT?

Priority Queue: **Implementation Choices**

- Generally, we use **Table ADT** more for:
  - Adding items then search for item
  - Traversal of items is sometimes needed
  - Deletion occurs less frequently

- **Priority Queue ADT** is used heavily for:
  - Repeated insertion and removal
  - No searching and traversals

- We can improve on the **space efficiency** and **coding complexity** by using **heap**

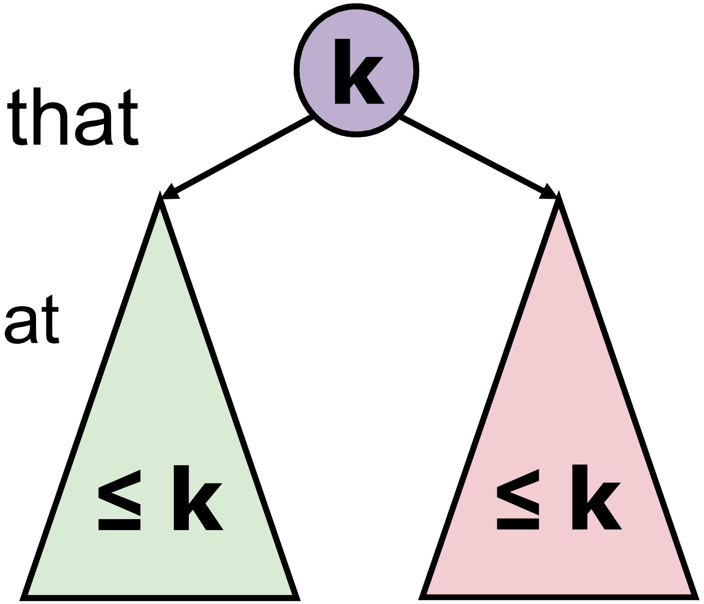
Binary Tree Again!

HEAP

Definition: Heap

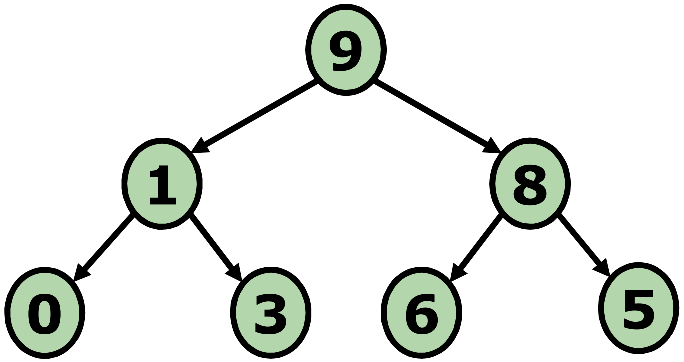
Heap:

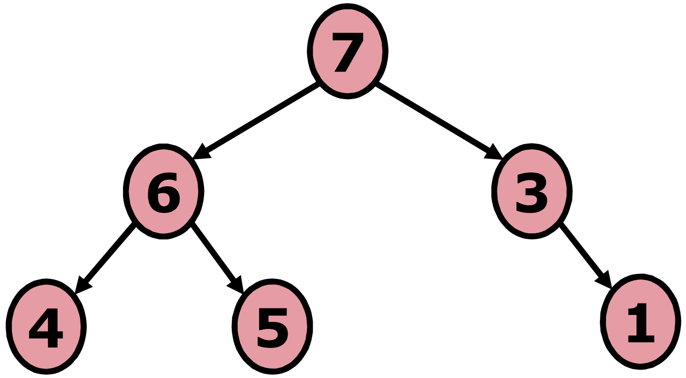
- is a **Complete Binary Tree** that satisfies **heap property**:
  - Every node with key **k** such that
    - Nodes in **both left and right subtrees** have **keys ≤ k**
    - Known as **Max Heap**

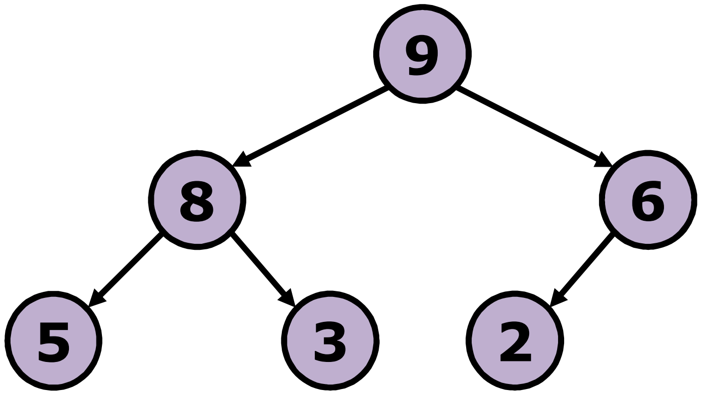


- Remember:
  - Heap is a **binary tree**
  - Heap is **not** a **binary search tree**

Check: **Which Binary Tree is Heap?**

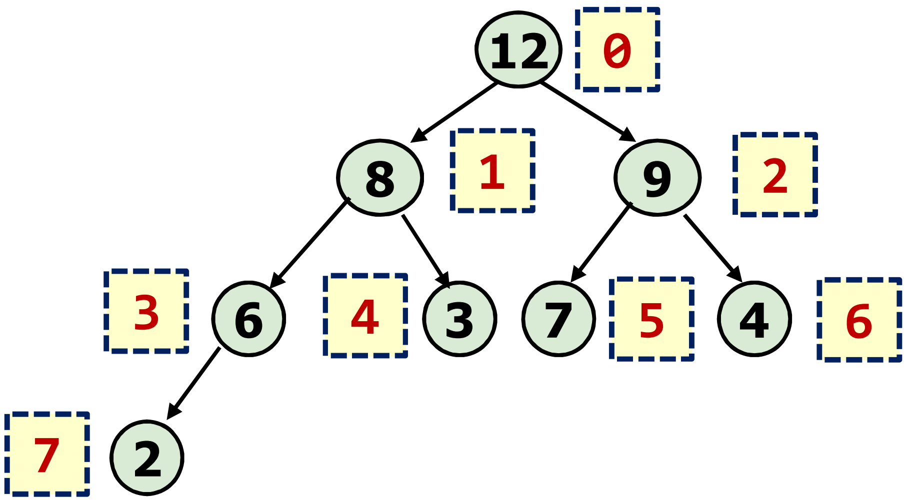






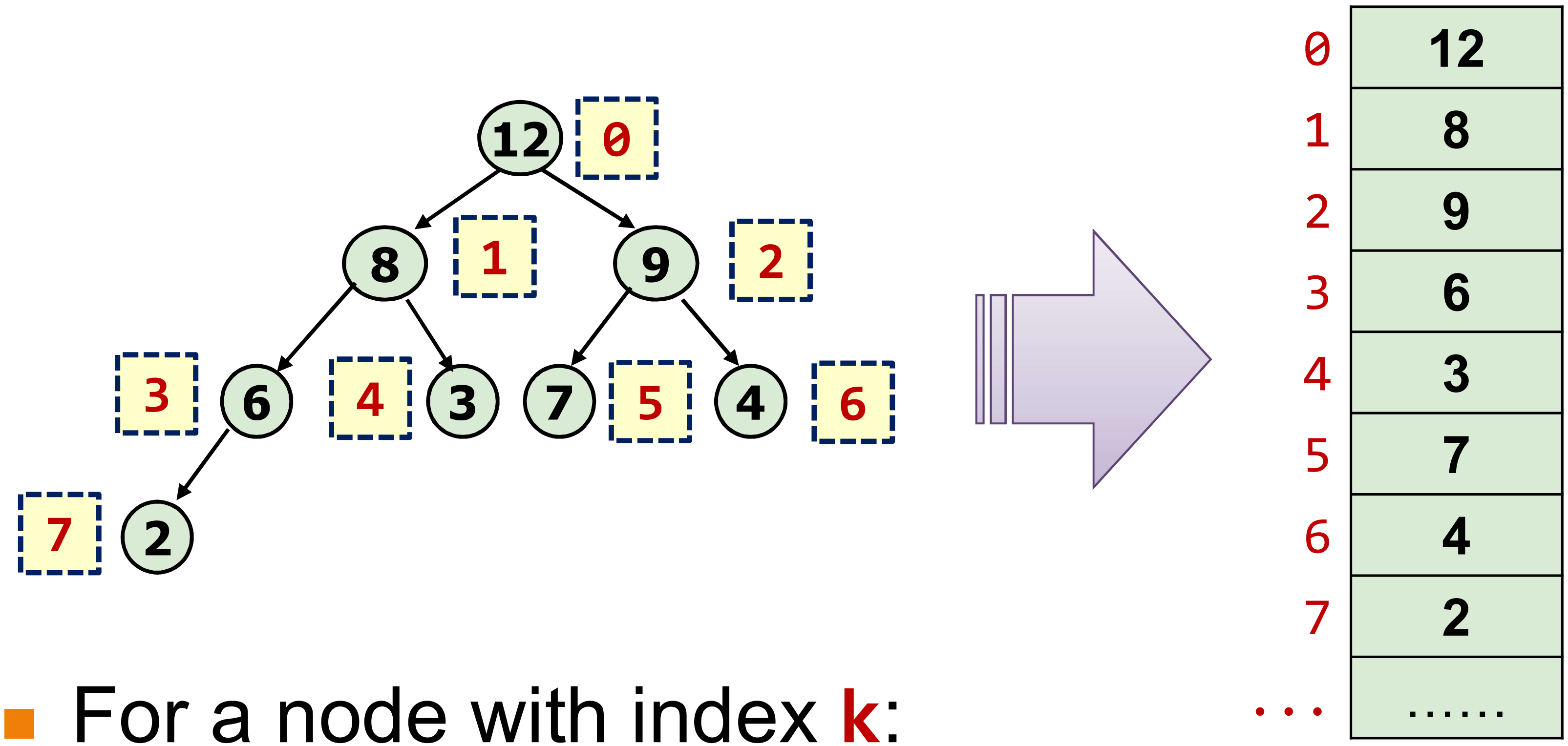
Efficient Array Representation

- Since heap is a complete binary tree, we can use an efficient array representation!
  - See "Tree-Binary Tree" Lecture on "Fixed Index"
- Each node has an index, starting with 0 at root and proceed in level order:



Recall the relationship between the node's index and its left / right child's index

Efficient Array Representation



- For a node with index **k**:

```
leftOf( k )
= 2k + 1
```

```
rightOf( k )
= 2k + 2
```

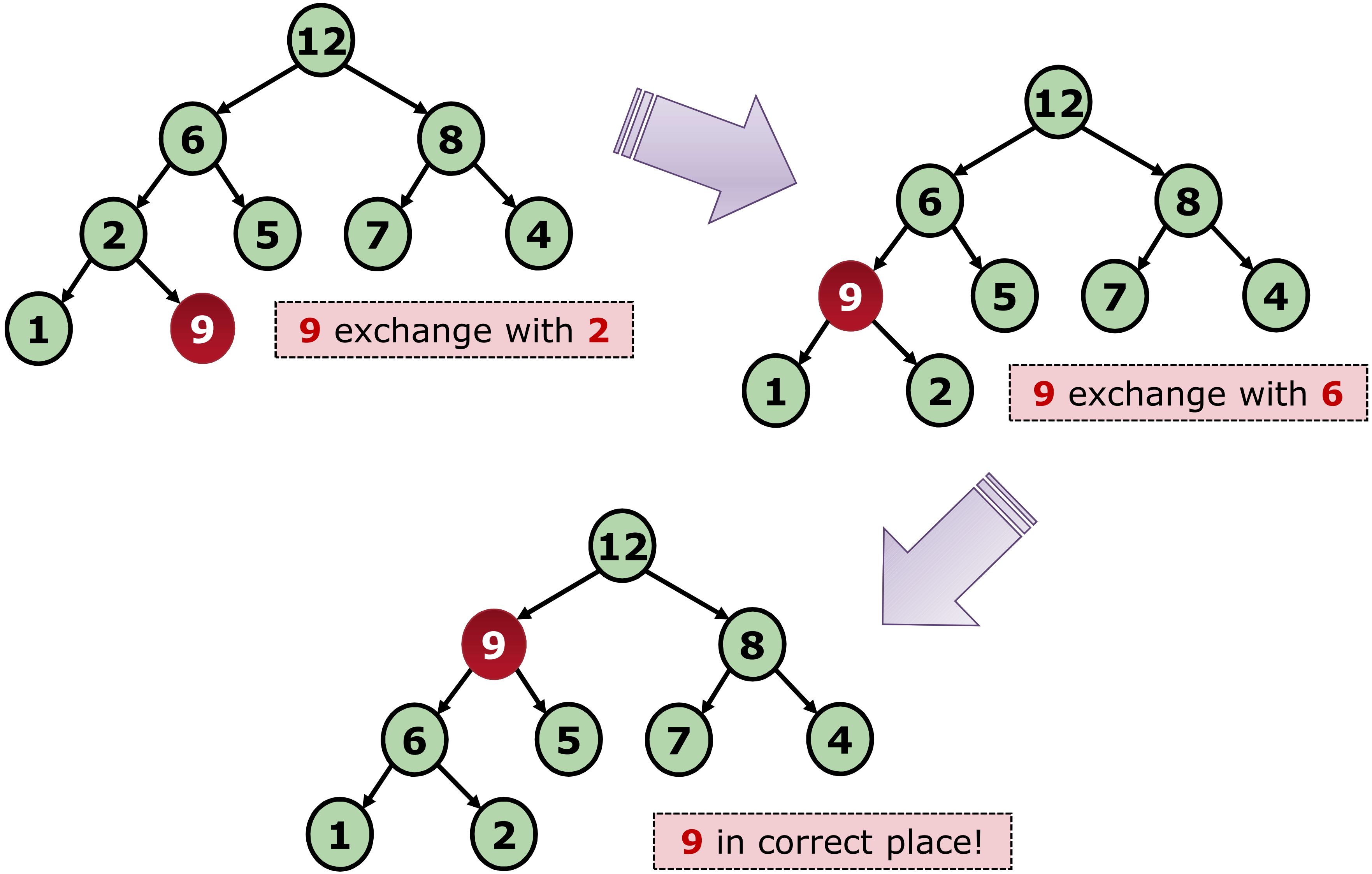
parentOf(k) $= \left\lfloor \frac{k-1}{2} \right\rfloor$

Heap: **Bubbling Mechanism**

- Violation of heap property can be easily corrected:
  - Due to the relaxed ordering in heap

- **Basic Idea:**
  - When an item ***m*** violates heap property we can perform repeated exchange:
    - with its parent (if ***m*** is larger than its parent), **OR**
    - with its children(if ***m*** is smaller than one of its children)
  - known as **bubbling**

Bubble Up: 9 violates Heap Property



Bubble Up: **Pseudo Code**

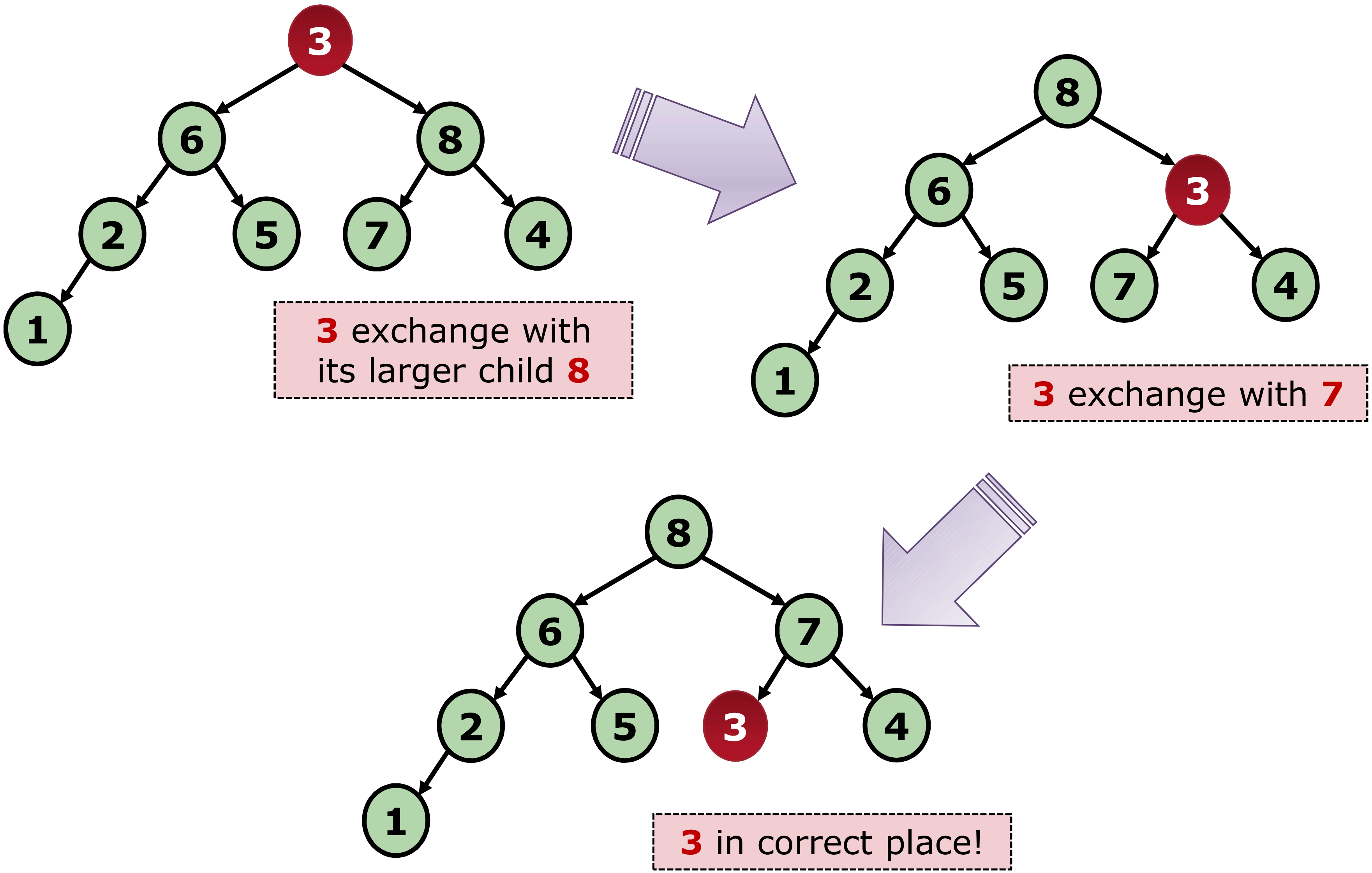
- Suppose we use:
  - items = Python List (i.e. array) to store the heap nodes

```
def bubbleUp( idx ):
    parent = parentOf( idx )

    while parent >= 0 and \
          itemArr[idx] > itemArr[parent]:

        swap( items, idx, parent )

        idx = parent
        parent = parentOf( idx )
```

Get parent's index

Repeatedly exchange with parent (bubble up)

Bubble Down: 3 violates Heap Property



Bubble Down: **Pseudo Code**

```
def bubbleDown( idx ):
    child = leftOf( idx )
    done = False

    while child < size and not done:
        rightC = rightOf( idx )
        if rightC < size and \
           items[child] < items[rightC]:
              child = rightC

        if items[idx] < items[child]:
              swap( items, idx, child )
        else:
              done = True

        idx = child
        child = leftOf( idx )
```

size is the number of nodes in heap

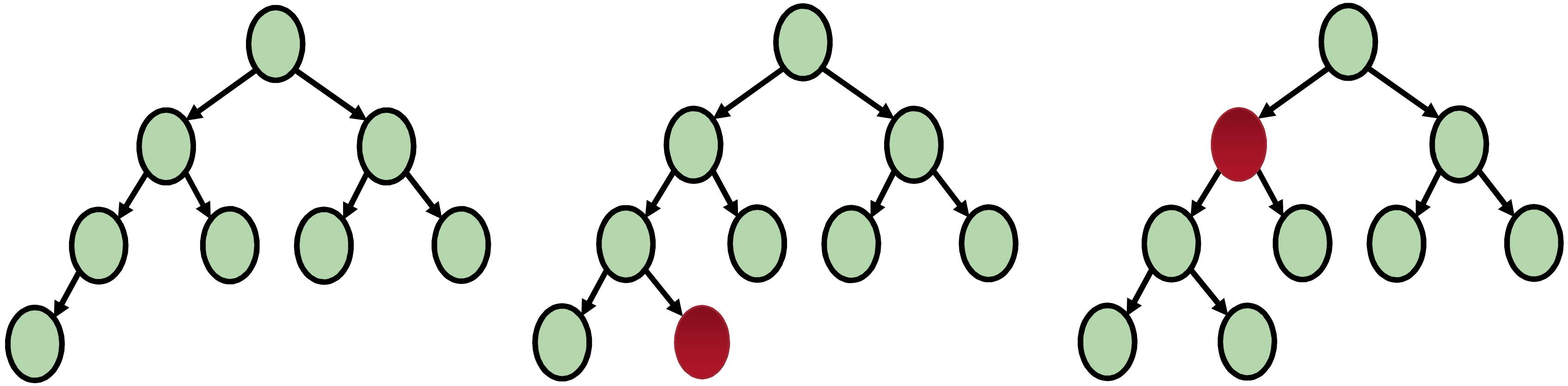
child is the larger of the child nodes

If the node is larger or equal to its larger child == we are done!

HEAP OPERATIONS

Insertion: Pseudo Code

```
def insert( key ):
    items[size] = key
    bubbleUp( size )
    size += 1
```



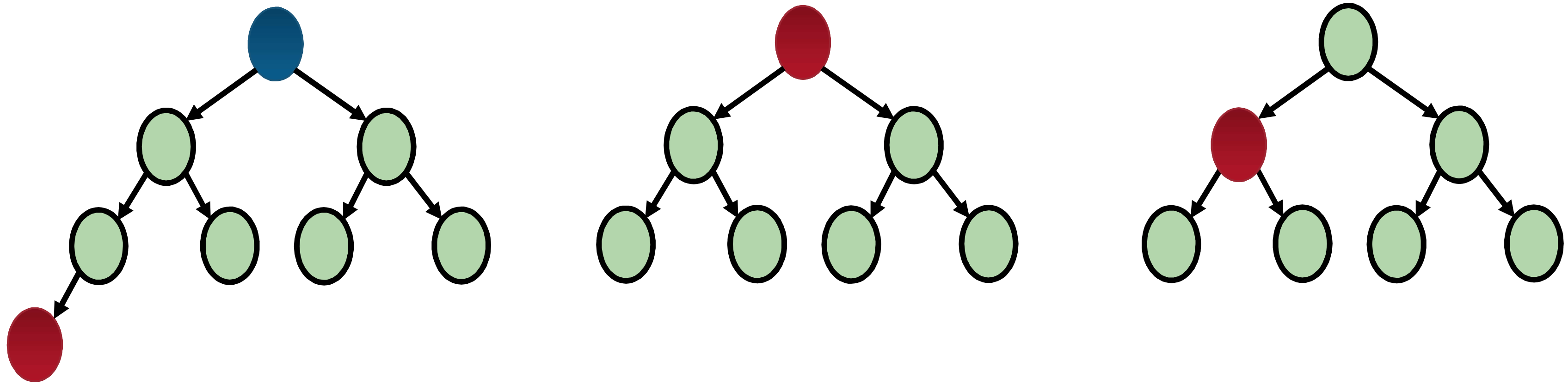
Heap before insertion

Insert new node at the end of heap

Bubble up the new node until it reaches the right location

Deletion: Pseudo Code

```
def delete( ):
    item = items[0]
    item[0] = items[size-1]
    bubbleDown( 0 )
    size -= 1
    return item
```



Heap before deletion

Replace root with last item in heap

Bubble down the new root until it reaches the right location

Insertion & Deletion: Complexity

- The worst case for the bubbling is the **height of the heap**

- Since **heap is complete**
  - height = **O(lg N)**

- Hence,
  - Insertion = **O(lg N)**
  - Deletion = **O(lg N)**

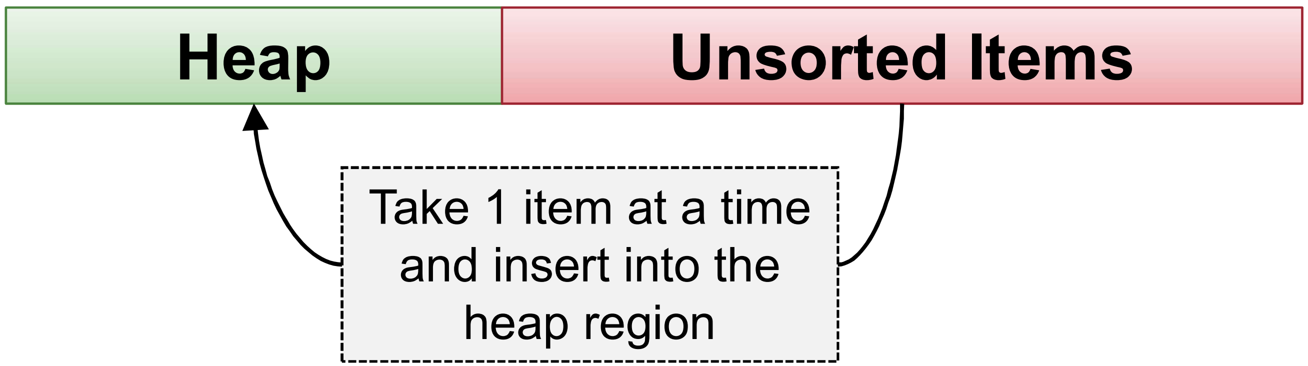
How do we **heapify** an array of unsorted items into heap?

HEAP CONSTRUCTION

Heap Construction: **First Attempt**

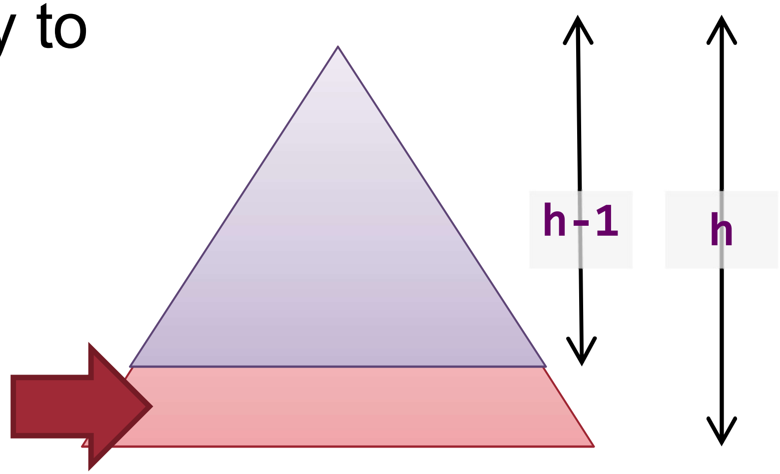
- Given an unsorted array of **N** items
  - How do we turn it into a heap?

- How about the following algorithm?
  1. Start with an empty heap
  2. Insert each of the **N** items into the heap



First Attempt: **Complexity**

- Worst Cast Time complexity:
  - Each item bubbles all the way to the root
  - There are > $\frac{N}{2}$ items at the bottommost level of the heap (why?)



→ Each takes **O(lg N)** exchanges to reach the root

→ **O(N lg N)**

Heap Construction: **Better Approach**

- The **O(N lg N)** complexity is the same as sorting which impose a total ordering on the items

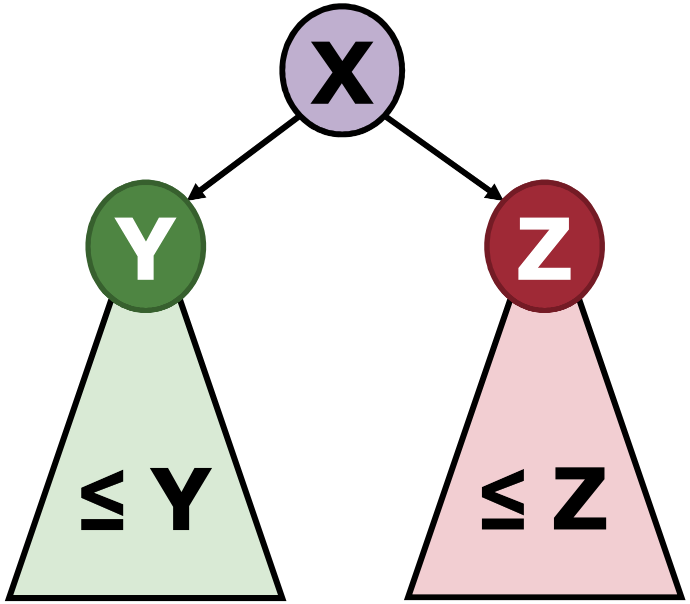
- Since heap items are not totally ordered
  - ➔ Intuitively there should be a better approach that requires lesser time

- Turns out there **is a better solution**
  - Make use of the idea of **semi heap**

Heap Construction: **Semi Heap**

- Given two **heaps**, if we add a **new root node X** on top
  - We have a **semi heap** since the root node may be out of order

- To turn a semi heap into a heap
  - **Simply bubble down the new root node X**



Not a heap as X may be smaller than Y or Z (or both)

Heapify Algorithm

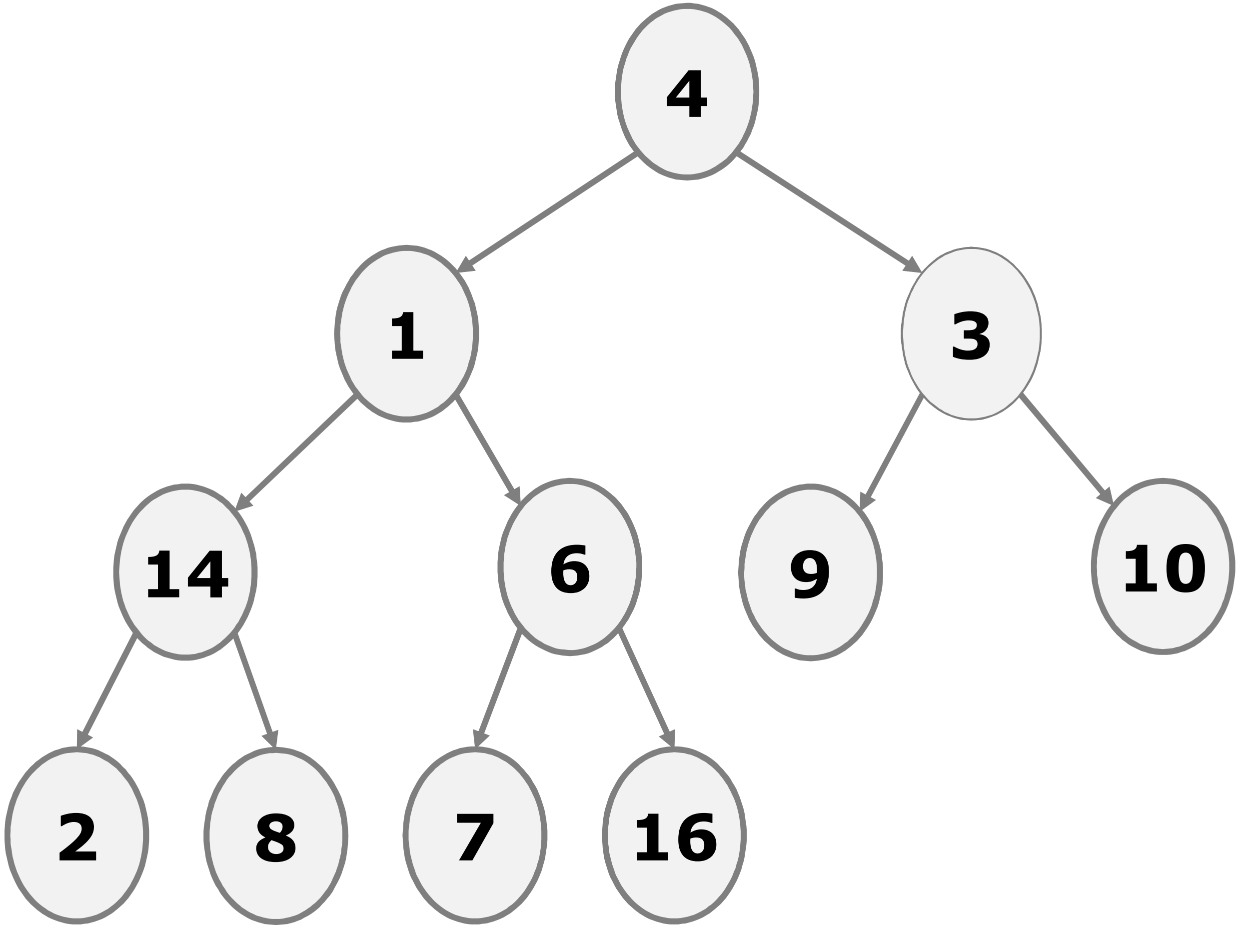
- Make use of the semi heap idea:
  - Take each pair of **height 1 heaps**, grow into a height 2 semi-heap, then convert to height 2 heap
  - Take each pair of **height 2 heaps**, grow into a height 3 semi-heap, then convert to height 3 heap
  - …… (sounds familiar?)

- **We recursively build the heap bottom up**
  - Starts with many heaps of smaller height
  - Combine and grow into taller heap
  - Finally get one single heap

Heapify Algorithm: **Illustration**

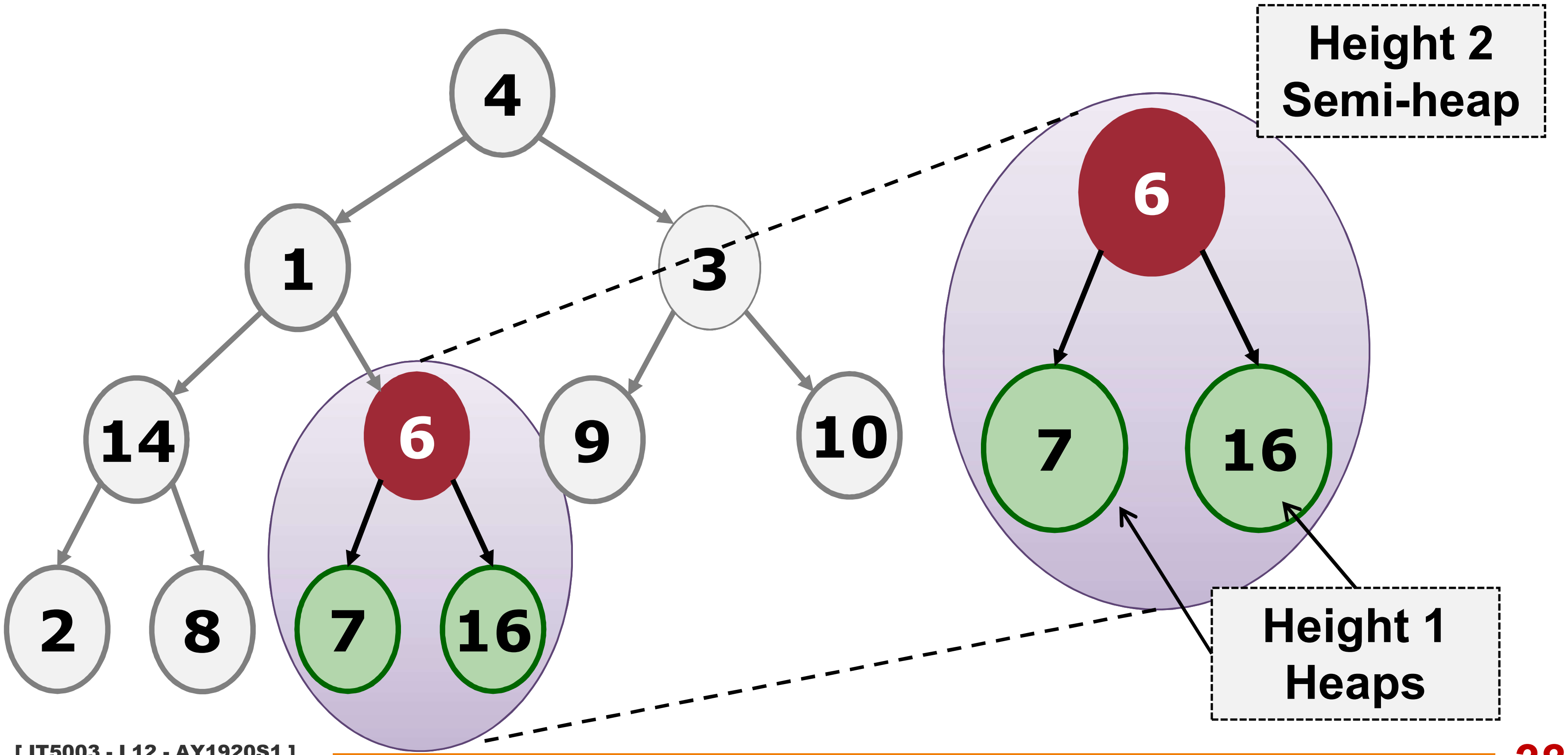
- Starts with the unsorted array

0	4
1	1
2	3
3	14
4	6
5	9
6	10
7	2
8	8
9	7
10	16



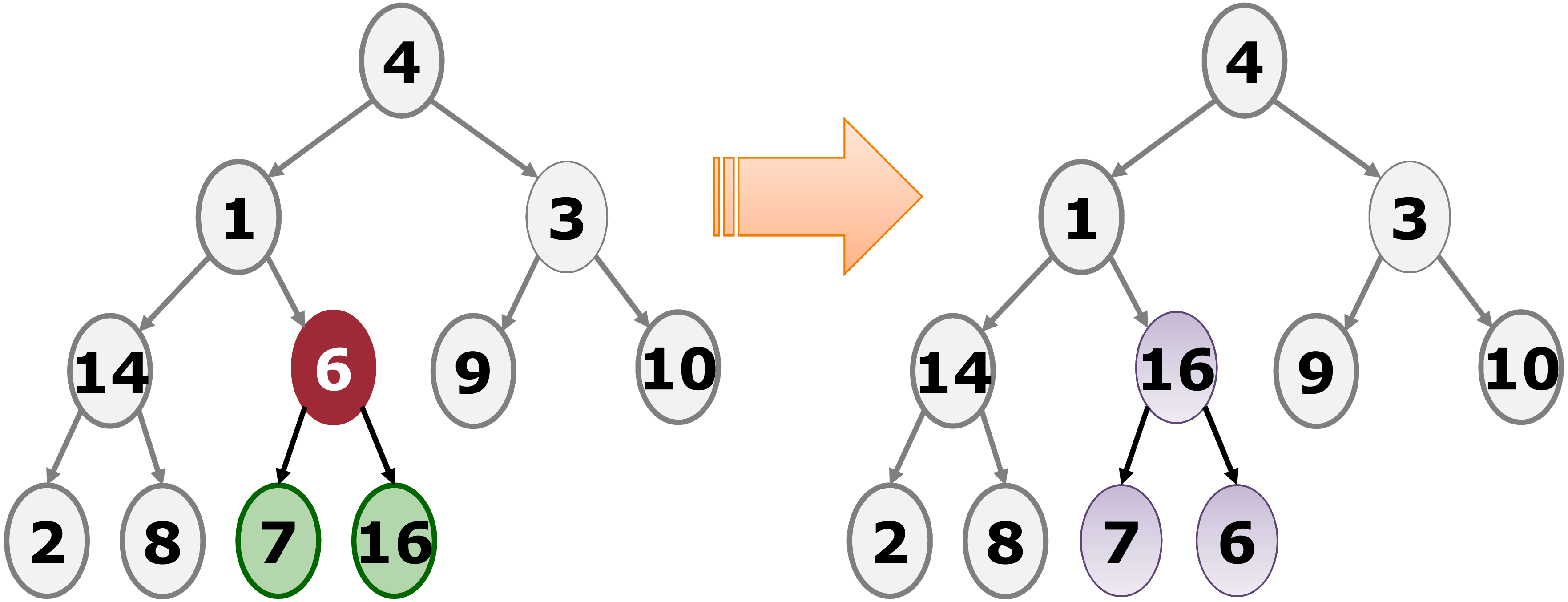
Heapify Algorithm: **Illustration**

- Each leaf node is a height 1 heap:
  - Connect a "root" node to two leaves ➔ height 2 semi-heap



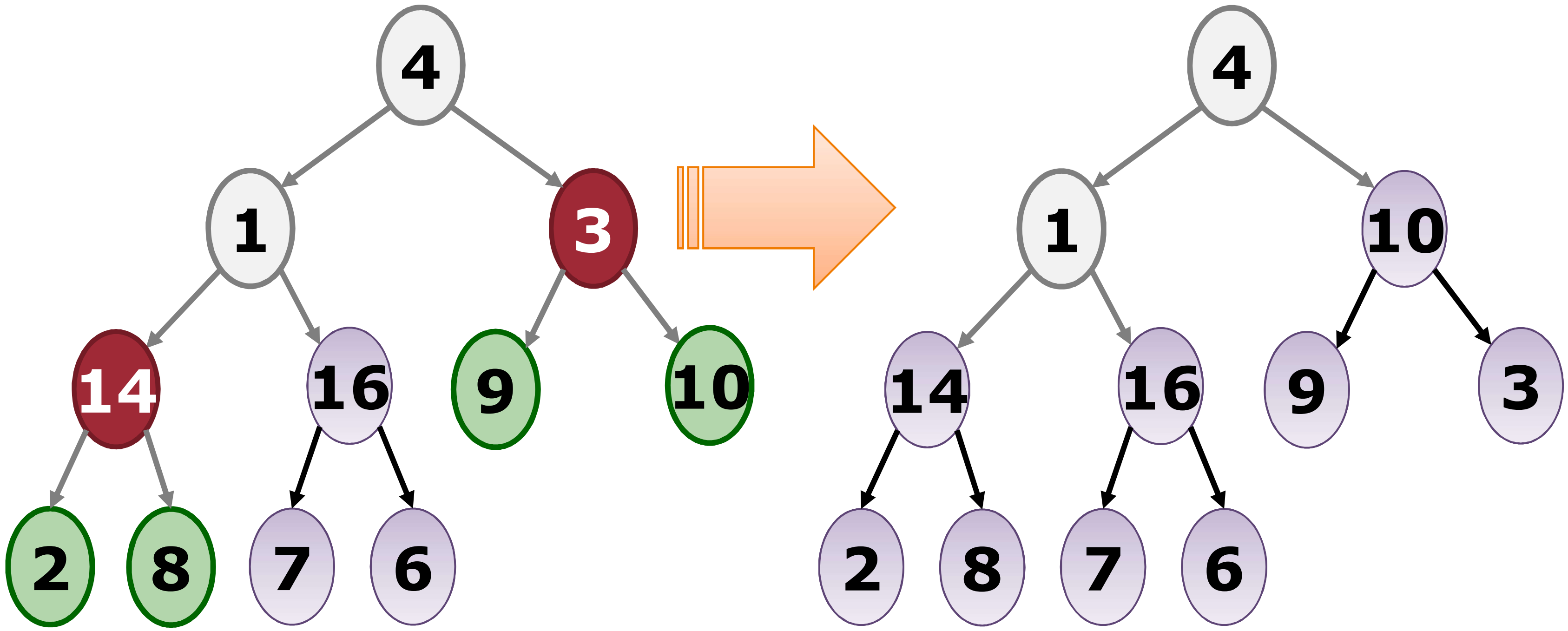
Heapify Algorithm: **Illustration**

- Convert each of the height 2 semi-heap into heap:



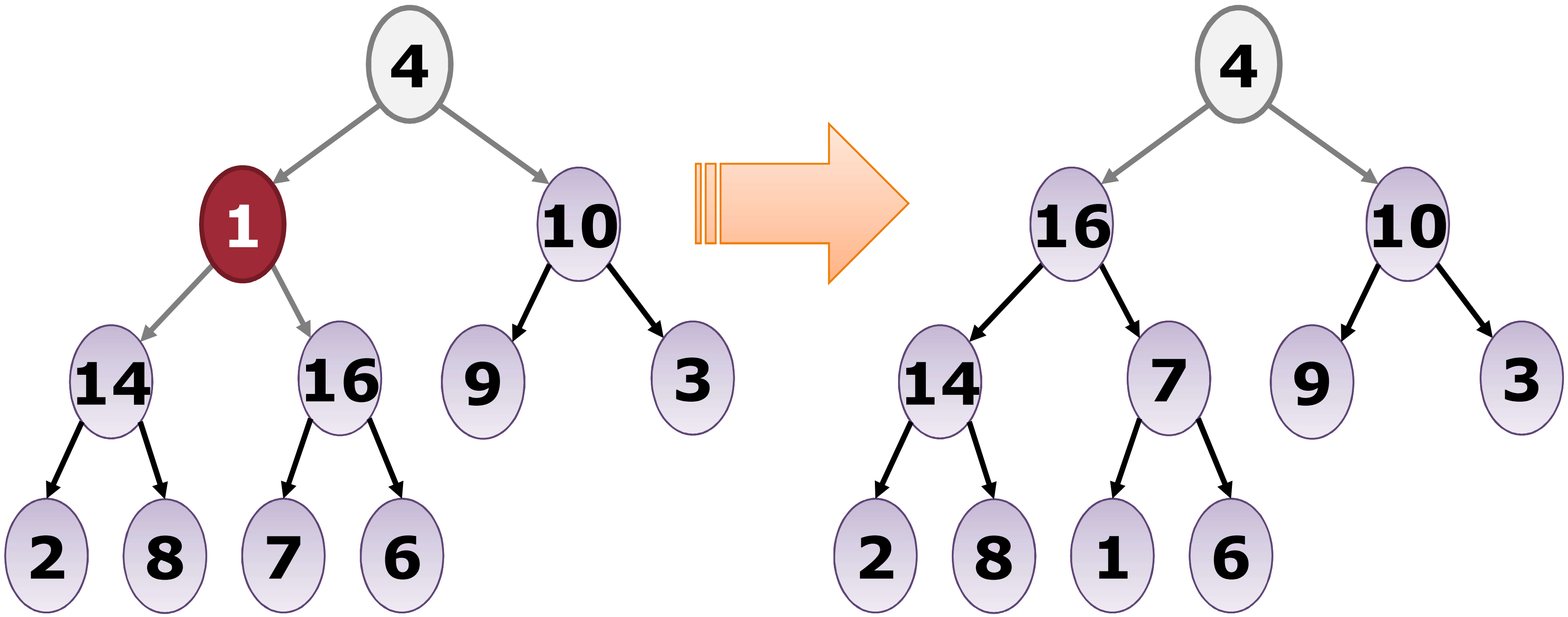
Heapify Algorithm: **Illustration**

- Same thing for height 2 semi-heap at "14" and "3"



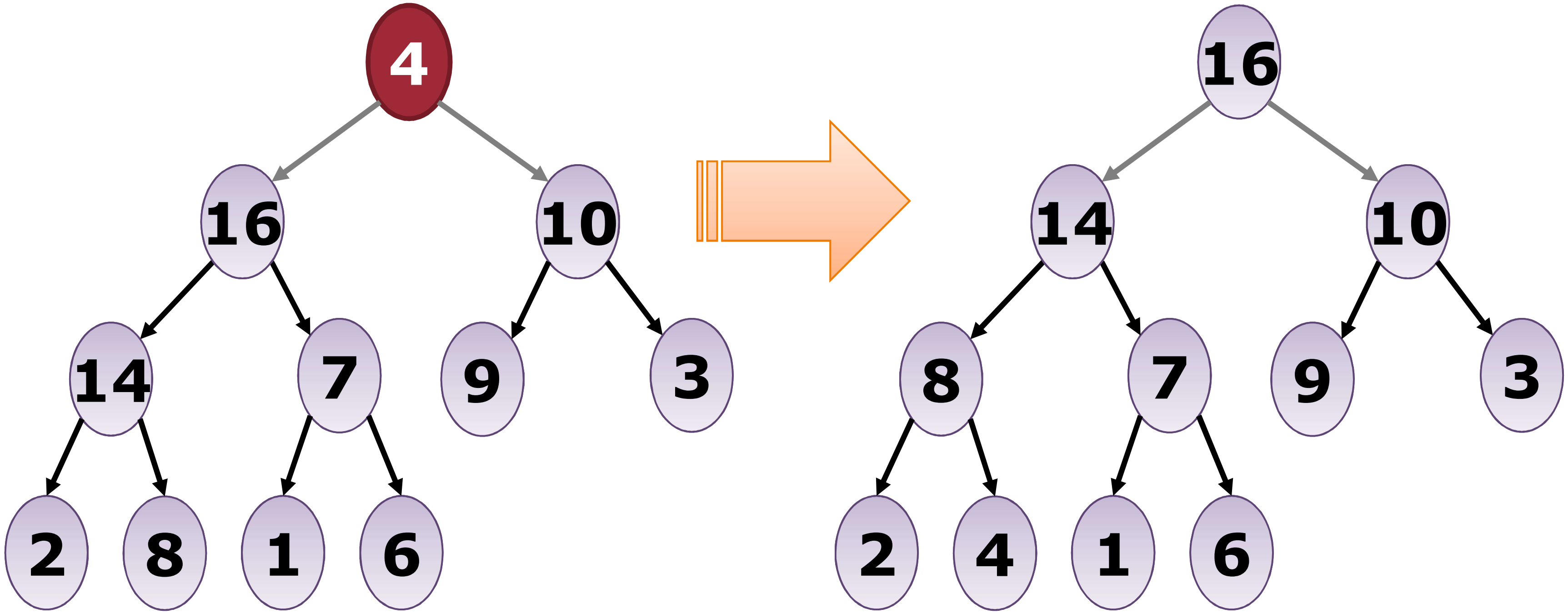
Heapify Algorithm: **Illustration**

- Continue the process:
  - Bottom-up, from right to left



Heapify Algorithm: **Illustration**

- When we reach the root node
  ➔ There is now a single heap!



Heapify Algorithm: **Pseudo Code**

```
def heapify( ):
    for idx in range( size // 2 – 1, -1, -1)
        bubbleDown( idx )
```

- Does this algorithm improves the time complexity?
- Intuitively, only the root node requires **O(lg N)** swaps in the worst case
  - Unlike the insertion method where all the leaves can causes **O(lg N)** swaps in the worst case
  - How do we calculate the time complexity?

Heapify Algorithm: **Analysis I**

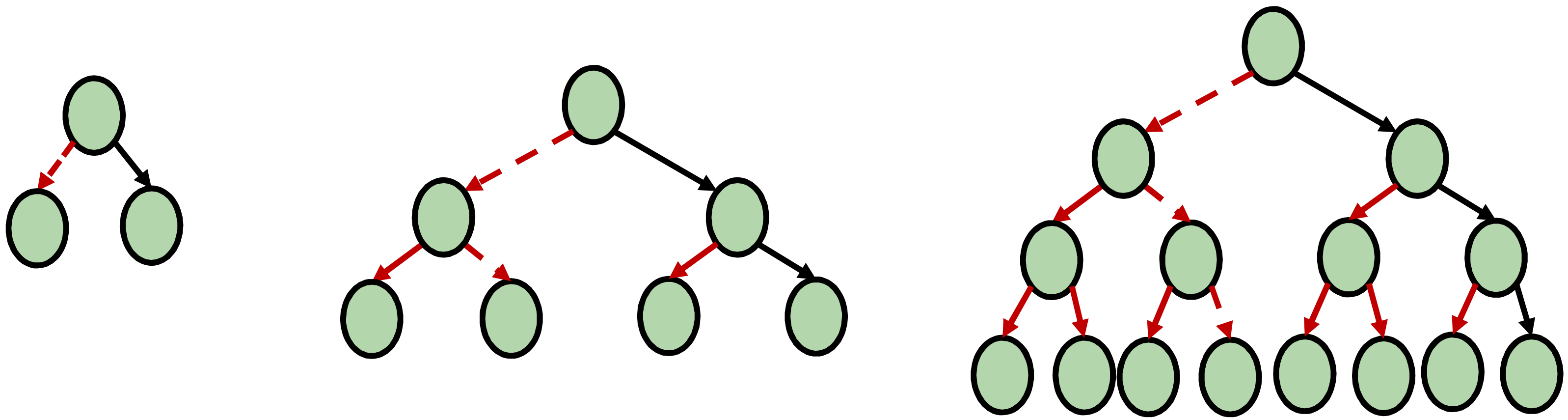
Number of Semi-Heap	Max Swaps	Height of Heap Formed
$\frac{N/2}{2} = \frac{N}{2^2}$	1	2
$\frac{N/2^2}{2} = \frac{N}{2^3}$	2	3
$\frac{N}{2^4}$	3	4
…..	…..	…..
$\frac{N}{2^{\lg}} = 1$	lg N – 1	lg N, i.e. **H**

- Total swapping in the worst case:
  - $1 \times \frac{N}{2^2} + 2 \times \frac{N}{2^3} + 3 \times \frac{N}{2^4} + \ ... + (\lg N \ - 1) \times \frac{N}{N}$

$$= N \ \times \left(\frac{1}{2^2} + \frac{2}{2^3} + \frac{3}{2^4} + \ ... + \frac{(\lg N \ - 1)}{N} \right) = \boldsymbol{O(\ N \ )}$$

Heapify Algorithm: **Analysis II**

- A more visual way to understand the complexity:
  - Let's **color an edge** whenever swapping occurs



- Whenever the heap grows 1 level to height H, we have 2H free edges to use:
  - Since worst case uses H edges ➔ there is still H free edges to pass to the next level
  - At root level, we used up (N-H) edges ➔ O(N) swapping!

Example: **An Application**

- The google search engine gives each webpage a **pagerank** according to search terms
  - Higher **pagerank** == more relevant to the search

- Search usually returns a huge amount of hits
  - What if we want to display 10 hits at a time in order of descending page rank scores?

- Let's assume the hits are stored in an array
  - Stores the page rank, the webpage address, etc
  - Not ordered by page rank

Example: **Algorithm 1 – Use Sort**

- We can use the following steps:
  - 1. Sort the array in descending order using page rank as key
    - **O(N lg N)** using mergesort
  - 2. Display the first **k** items in the array
    - **O(k)**
  - If we display all items (**k** items at a time)
    - **O(N/k * k) → O(N)**

- **Overall cost:**
  - **O(N lg N)** for fixed **k**
  - **O(N lg N)** for all items (k at a time)

Example: Algorithm 2 – Use Heap

- Alternative steps:
  - 1. Heapify the array
    - **O(N)**
  - 2. Display the first **k** items in the array
    - **O(k lg N) → O(lg N)** if **k** is constant
  - If display all items (**k** items at a time)
    - **O(N/k *k lg N) → O(N lg N)**

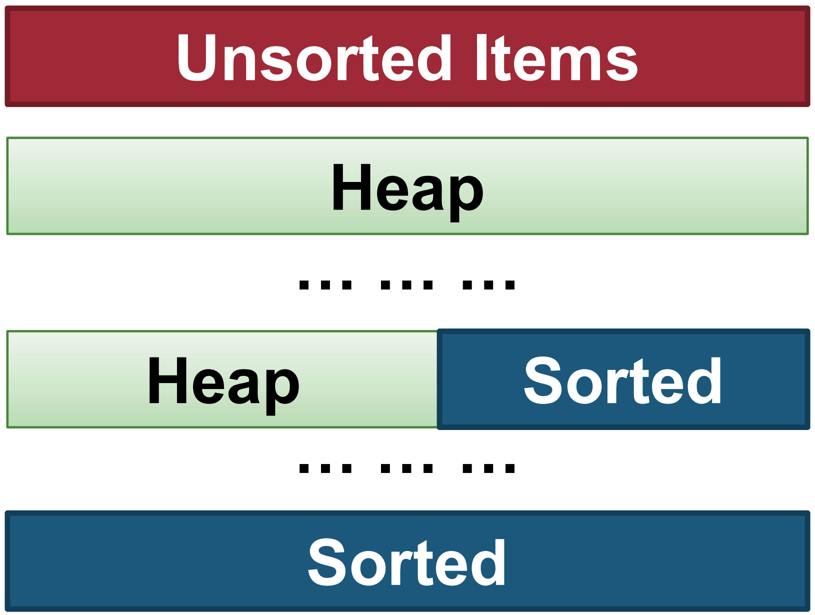
- **Overall cost:**
  - **O(N)** for fixed **k** --- Faster!
  - **O(N lg N)** for all items --- Same

Sorting using heap

HEAPSORT

Heapsort

- Basic Idea:
  - Modification of the selection sort
  - Use heap deletion to select the **maximum value**
    - More efficient than the original
- Illustration:



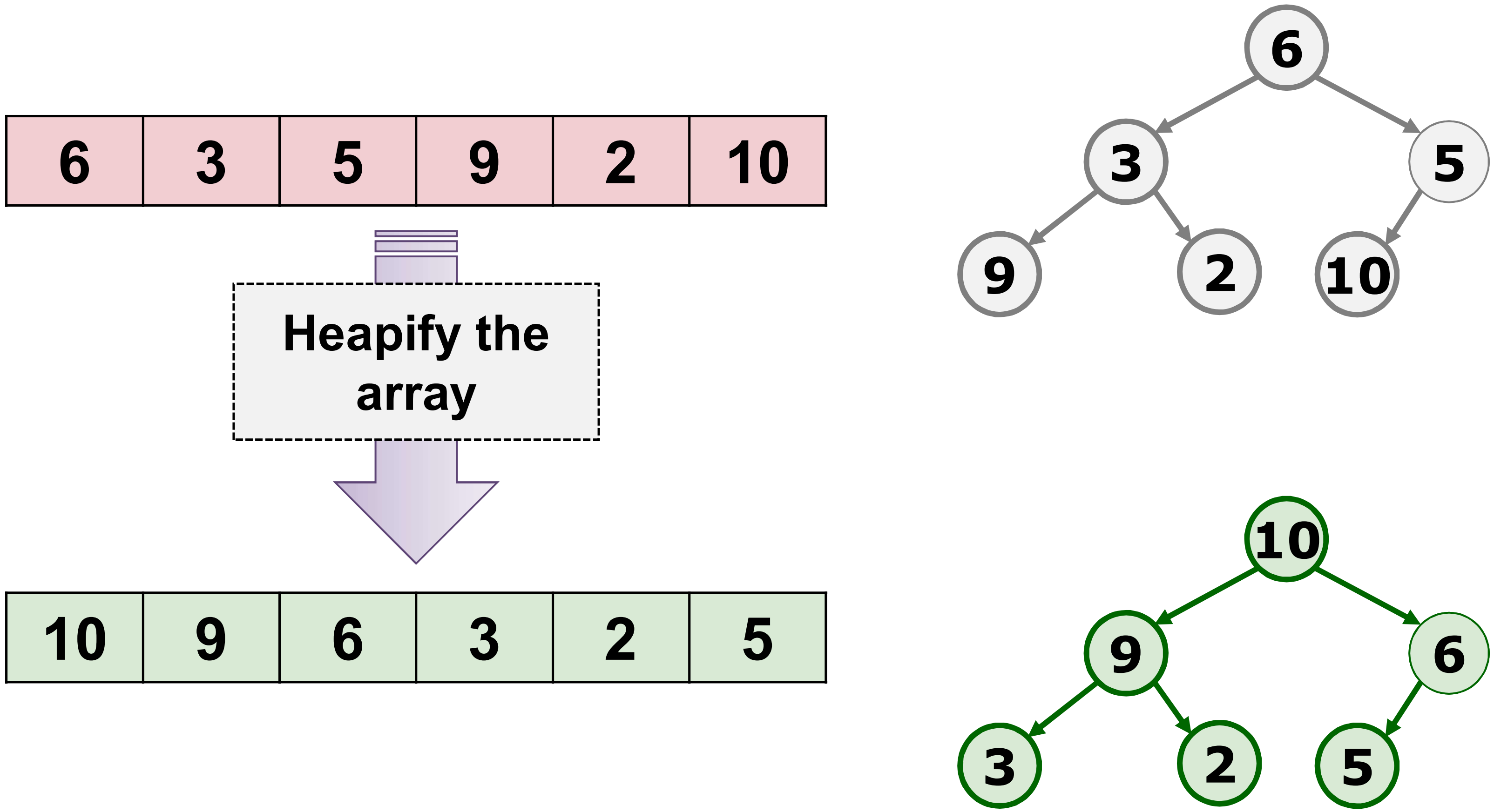
1. Heapify the array

2. Delete from heap, place item at the end of array

3. Repeat step 2. Item removed placed at the end of sorted region

4. Eventually, the whole array is sorted!

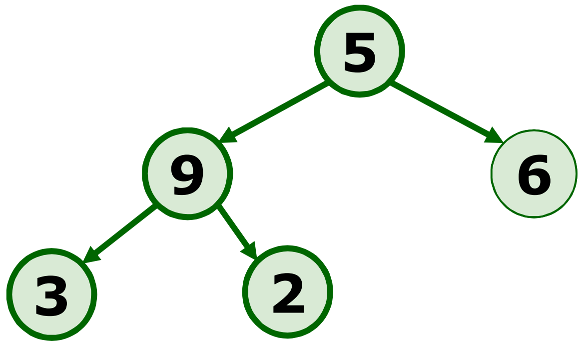
Example: Step 1. Heapify the array



Example: Step 2. Heap Delete

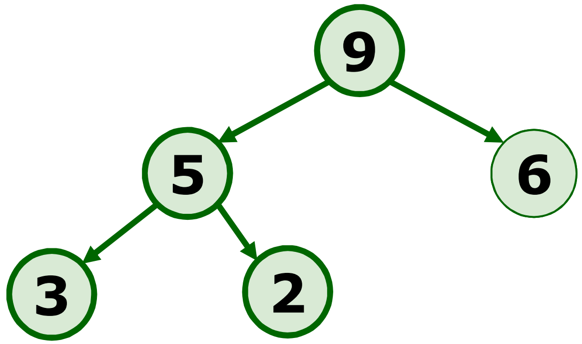
Delete "10" from heap

10	9	6	3	2	5



5	9	6	3	2	

Bubble down "5"



9	5	6	3	2	

Place "10" at the end of array

9	5	6	3	2	10

Example: **Repeat Step 2**

9	5	6	3	2	10

Delete "9" from heap, bubble down "2"

6	5	2	3	9	10

Delete "6" from heap, bubble down "3"

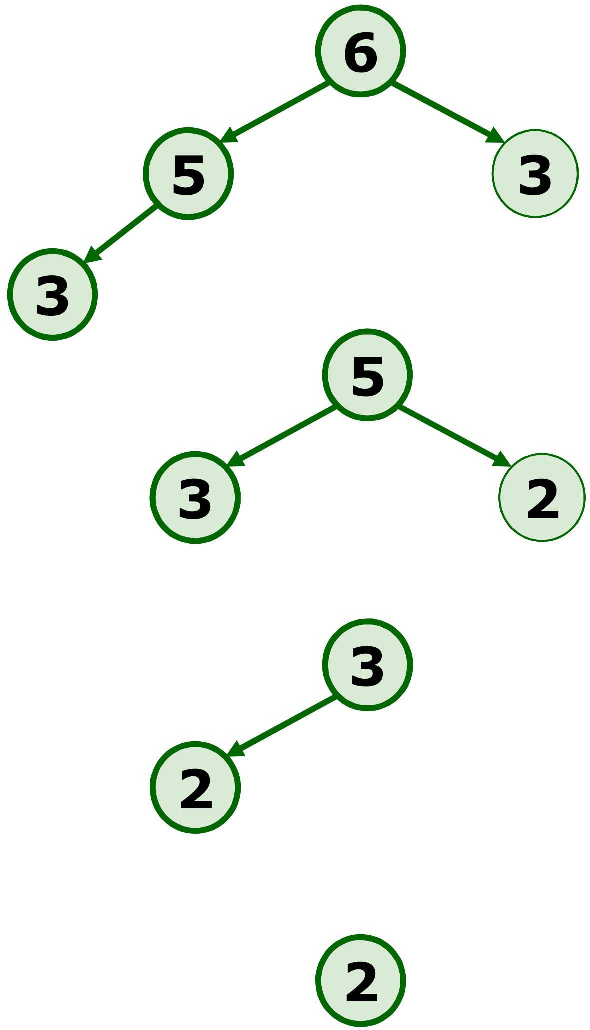
5	3	2	6	9	10

Delete "5" from heap, bubble down "2"

3	2	5	6	9	10

Delete "3" from heap, bubble down "2"

2	3	5	6	9	10



Heapsort: Complexity

- Self exercise:
  - What is the complexity of the algorithm?
  - Is the sorting in place?
  - Is the sorting stable?

Summary

- Priority Queue is a specialized queue where items are ordered by priority (highest priority = front)

- Heap is a variant of binary tree
  - Bubbling Mechanism
  - Insertion / Deletion
  - Heapify
  - Heapsort

END