BITS Pilani

Pilani|Dubai|Goa|Hyderabad

Data Structures and Algorithms Design

DSECLZG519

Parthasarathy

innovate achieve lead

BITS Pilani

Pilani|Dubai|Goa|Hyderabad

Contact Session #5

DSECLZG519 – Heaps & Heap Sort

Agenda for CS #5

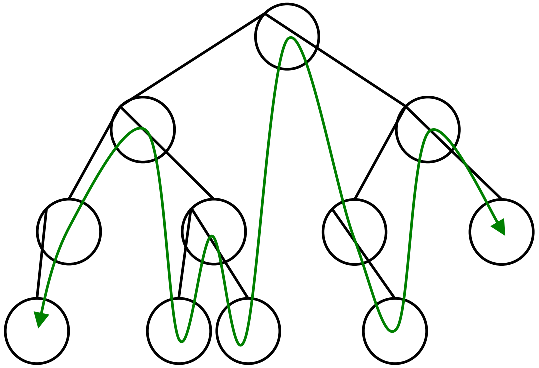
1) Recap of CS#4
2) Introduction to Heap
   - What is Heap ?
   - Types of Heap
   - Heapification
   - Build a Heap
   - Insertion into a Heap
   - Removal from a Heap
   - Exercises
3) Heap Sort
4) Q&A

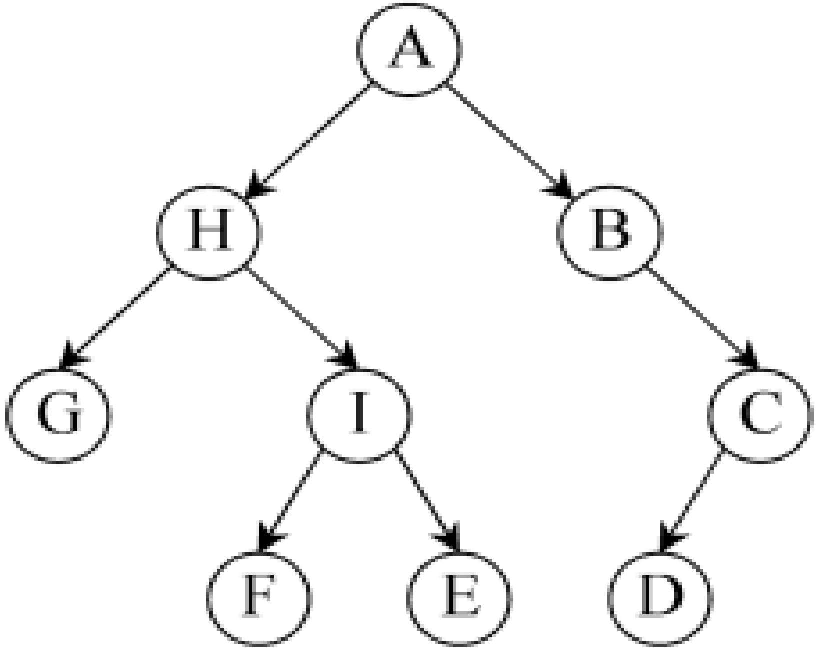
Tree Traversal: In-order

In-order

- Traverse the left sub-tree of R in in-order
- Process the root R
- Traverse the right sub-tree of R in in-order

a.k.a left-node-right (**LNR**)





In-order (LNR) traversal yields:

G, H, F, I, E, A, B, D, C

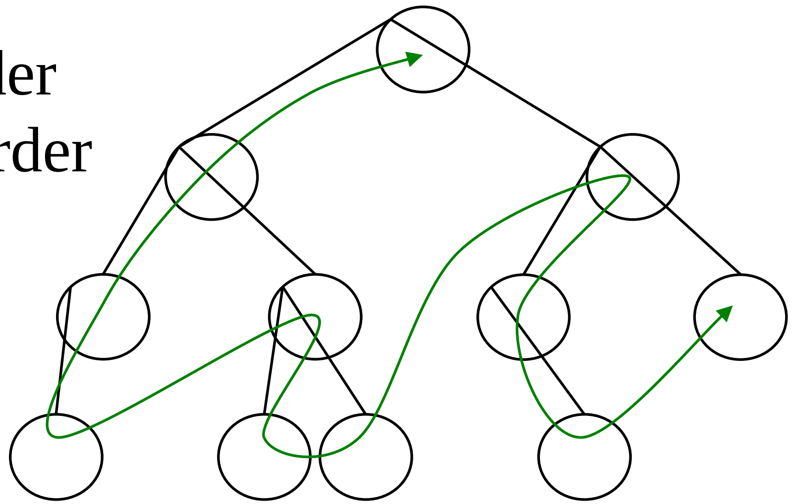
Tree Traversal: Pre-order

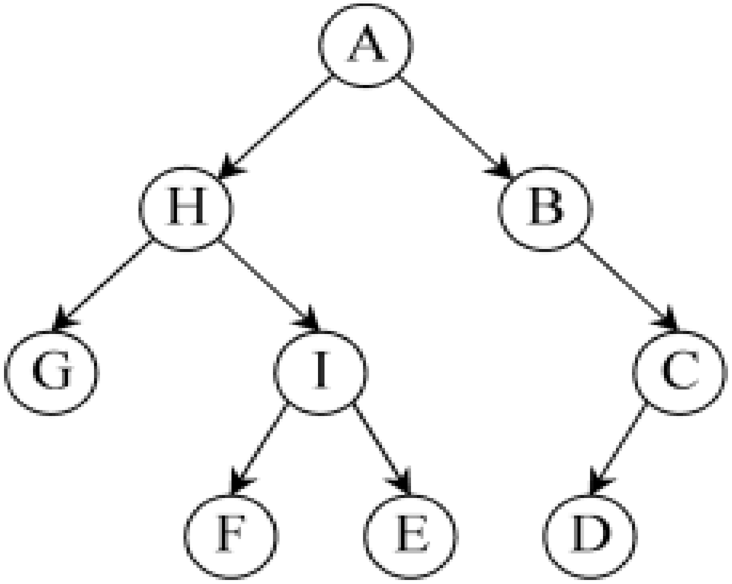
innovate achieve lead

Preorder

- Process the root R
- Traverse the left sub-tree of R in preorder
- Traverse the right sub-tree of R in preorder

a.k.a node-left-right (**NLR**)





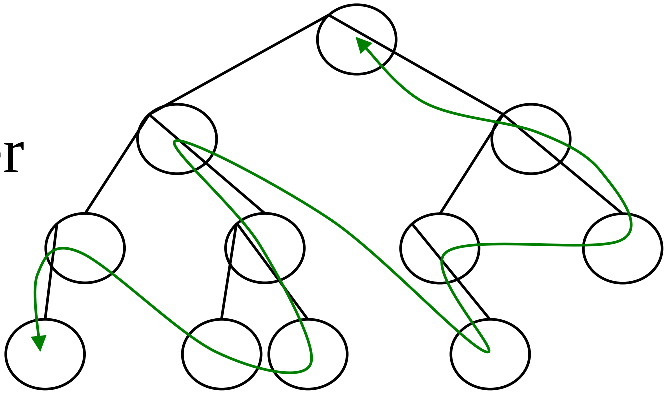
Preorder (NLR) traversal yields: **A, H, G, I, F, E, B, C, D**

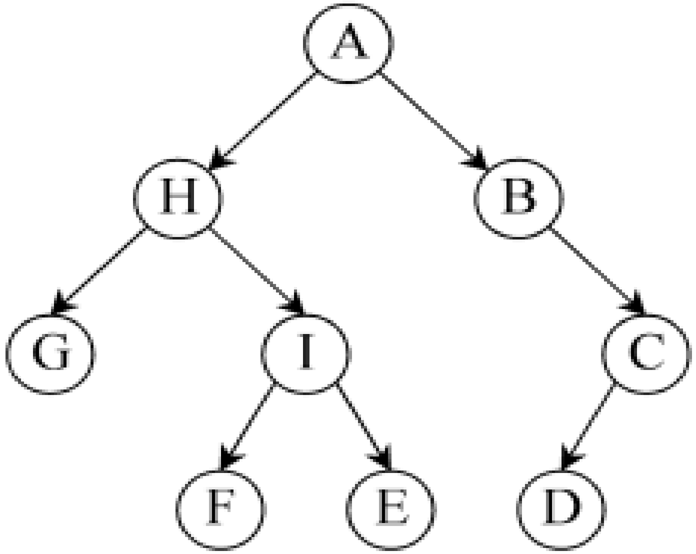
Tree Traversal: Post-Order

Post-order

- Traverse the left sub-tree of R in post-order
- Traverse the right sub-tree of R in post-order
- Process the root R

a.k.a left-right-node (**LRN**)





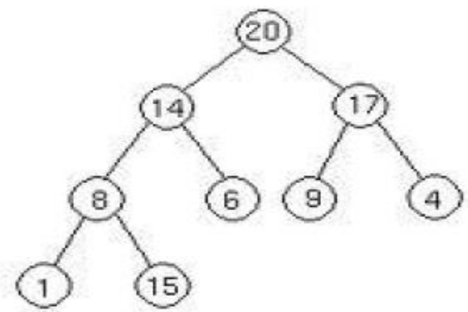
Postorder (LRN) traversal yields: **G, F, E, I, H, D, C, B,A**

Application of Binary Trees

- Data Base indexing
- In video games
- Path finding algorithms in AI applications
- Huffman Coding
- Heaps
- Syntax tree.
- ….
- …

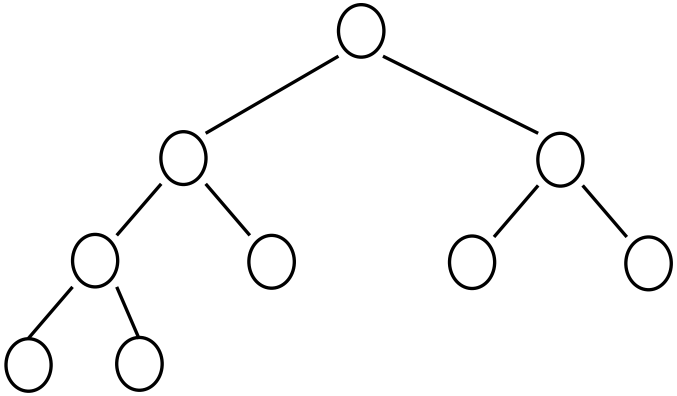
Heap

- Heap is a special tree-based data structure, that satisfies the following special heap properties:
  - *Shape Property*
  - *Heap Property*



Shape Property

Heap data structure is always a complete binary tree, which means all levels of the tree are fully filled till h-1 and at level h (last level), the nodes are filled from left to right.



Complete Binary tree

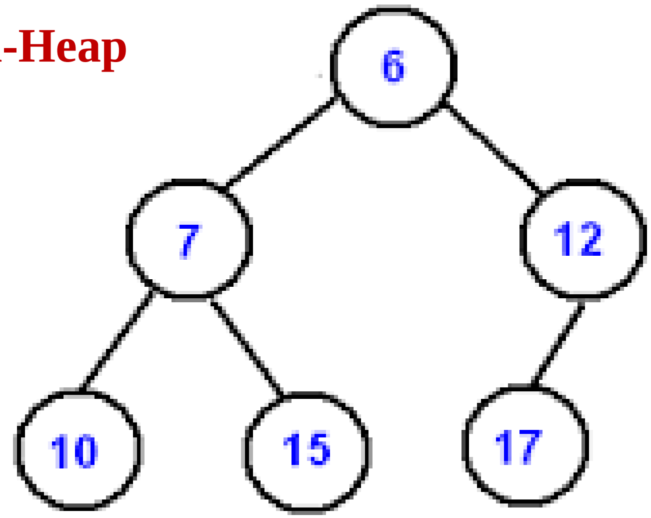
Heap Property

- All nodes are either [greater than or equal to] or [less than or equal to] each of its children.
- If the parent nodes are greater than their children, then such a heap is called: **Max-Heap.** *So,* ***,*** *The root of any sub-tree holds the* ***greatest*** *value in the sub-tree*
- If the parent nodes are smaller than their children, then such a heap is called: **Min-Heap.** *So, the root of any sub-tree holds the* ***least*** *value in that sub-tree.*

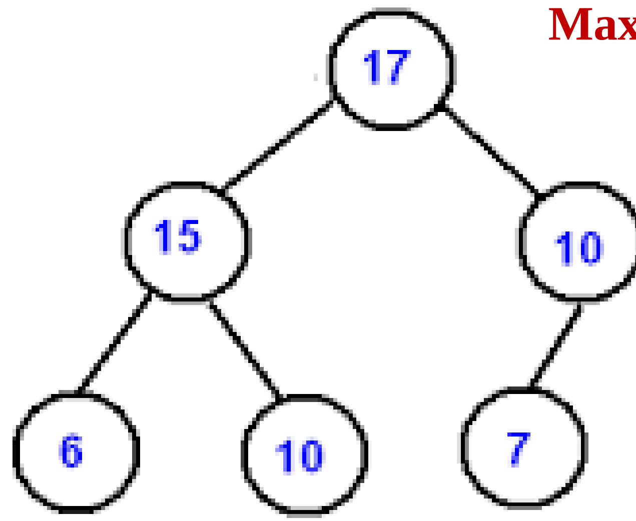
- **Max-Heap**
  ```
  for every node v other than the root
          element(parent(v)) ≥ element(v)
  ```
- **Min-Heap**
  ```
  for every node v other than the root
          element (parent(v)) ≤ element(v)
  ```

Examples of min-heap and max-heap

Min-Heap

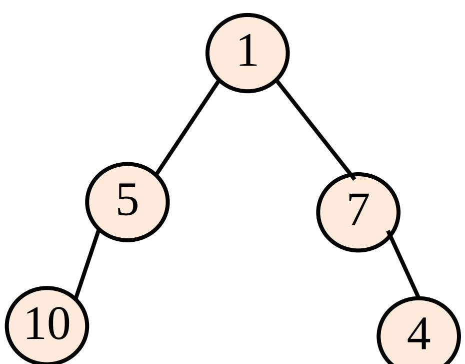


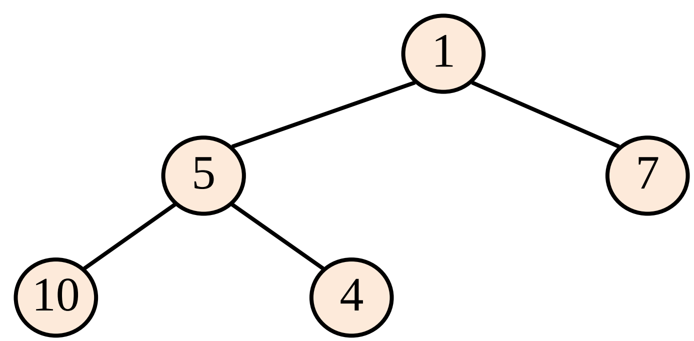
Max-Heap



Check :

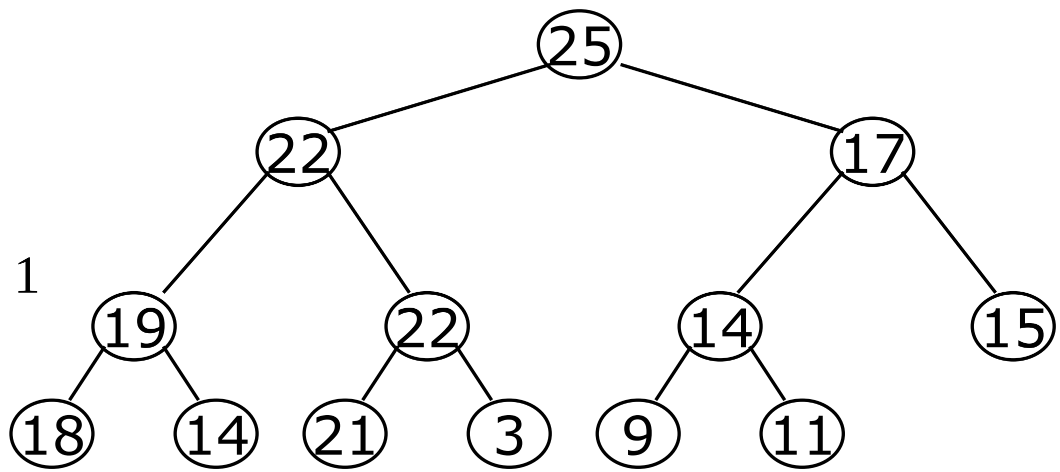
These are not heaps!!





Heap Representation

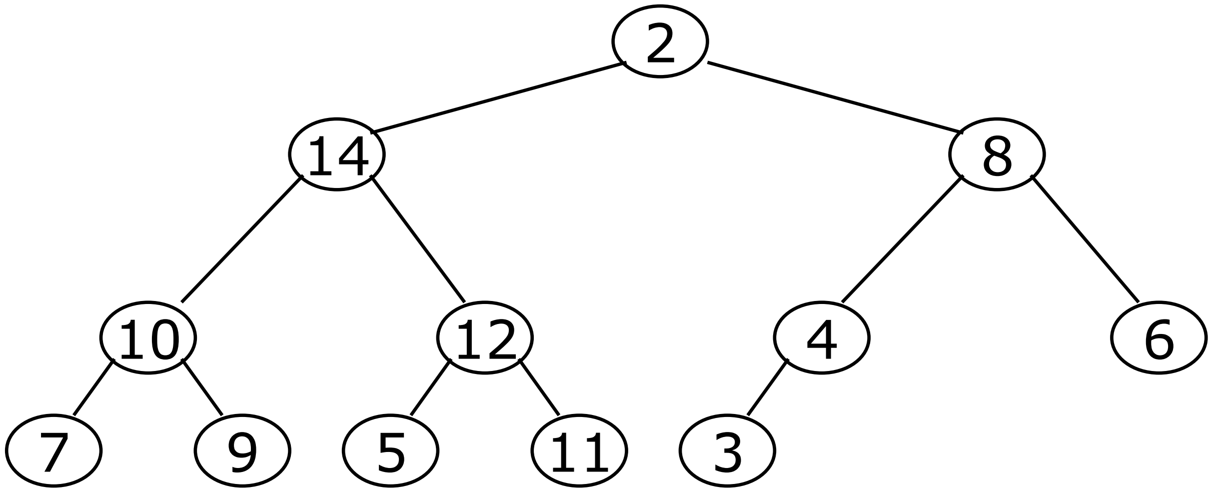
- Since, a heap is always a complete binary tree, can we represent heap as array easily and effectively ? *YES*
- Similar to array implementation of binary trees
- Root is at index 1
- For any node at index ***i***
  - The left child is at index 2***i***
  - The right child is at index 2***i*** + 1
  - Parent is at floor(***i***/2)



1	2	3	4	5	6	7	8	9	10	11	12	13
25	22	17	19	22	14	15	18	14	21	3	9	11

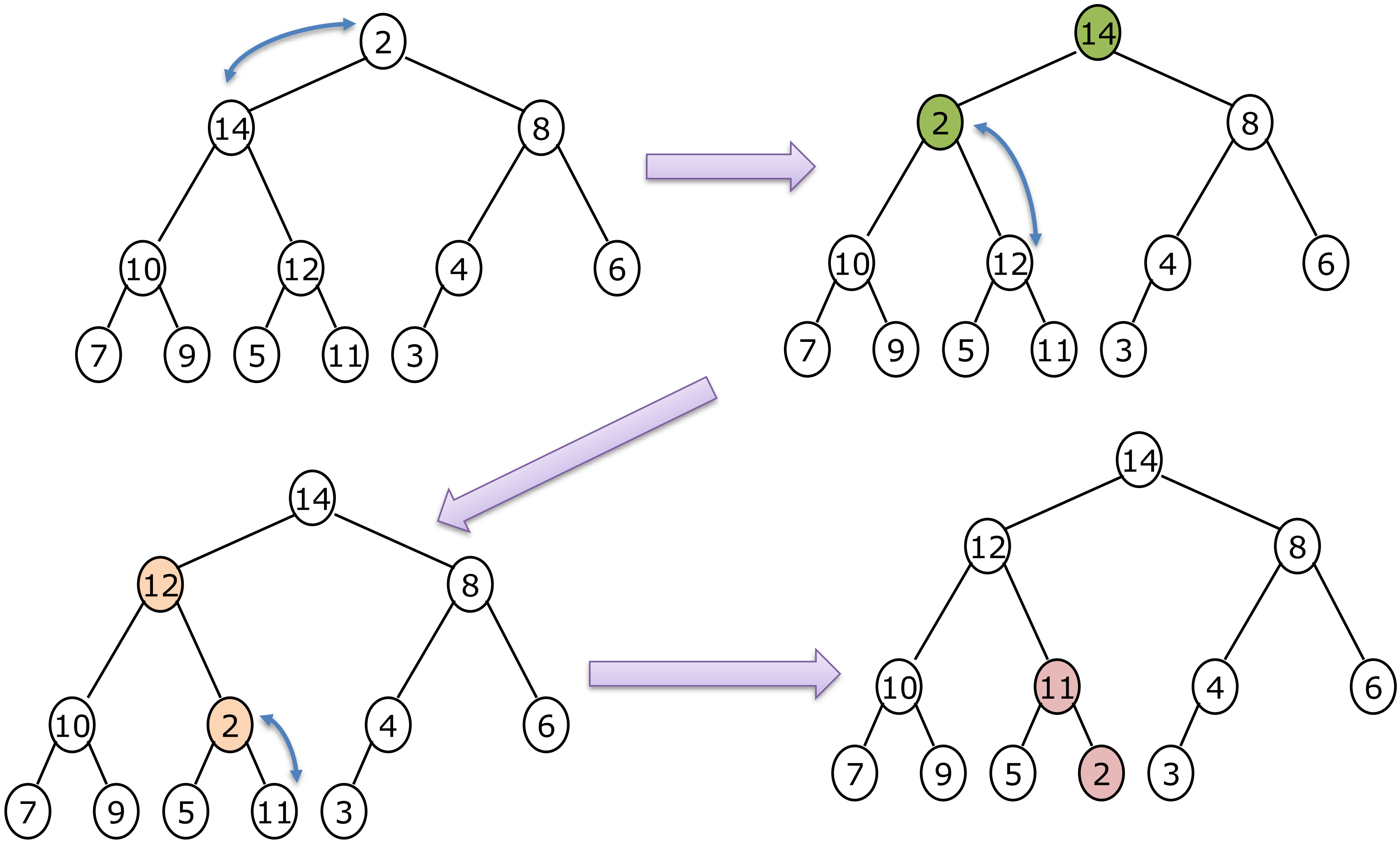
Heapification (Max-Heapify)

- Before discussing the method for building heap of an arbitrary complete binary tree, we discuss a simpler problem.
- *Let us consider a binary tree in which left and right subtrees of the root satisfy the heap property but not the root*. See the following fig:



- Now the Question is how to transform the above tree into a Heap ?
- Heapification !! Commonly referred as Max-Heapify()

Sequence Depicting the Heapification process



Algorithm: Max-Heapify(B, s)

Algorithm 1: Max-Heapify Pseudocode

Data: B: input array; s: an index of the node
Result: Heap tree that obeys max-heap property

```
Procedure Max-Heapify(B, s)
    left = 2s;
    right = 2s + 1;
    if left ≤ B.length and B[left] > B[s] then
        largest = left;
    else
        largest = s;
    end
    if right ≤ B.length and B[right] > B[largest] then
        largest = right;
    end
    if largest ≠ s then
        swap(B[s], B[largest]);
        Max-Heapify(B, largest);
    end
end
```

The time complexity of max-Heapify is **O(log n)**

**Since the complete binary tree is perfectly balanced, shifting up a single node takes O(log n) time.*

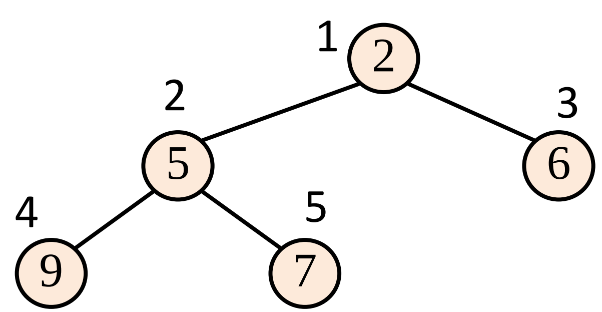
Build Heap

- Heap building can be done efficiently with **bottom up fashion**.
- Given an arbitrary complete binary tree, we can assume each leaf is a heap
- Start building the heap from the *parents of these leaves* i.e., Max-Heapify subtrees rooted at the parents.
- The Heapify process continues till we reach the root of the tree.

Algorithm 2: Building a Max-Heap Pseudocode

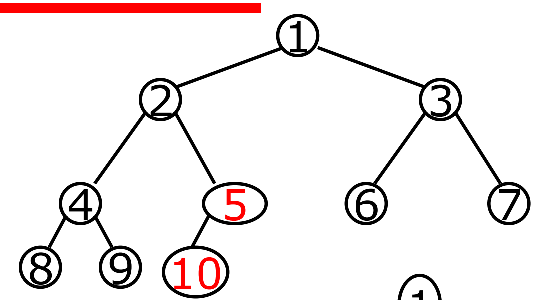
```
Data:   B: input array
Result: Heap tree
Procedure Max-Heap-Building(B)
    B.heapsize = B.length;
    for k = B.length/2 down to 1 do
        Max-Heapify(B, k);
    end
end
```

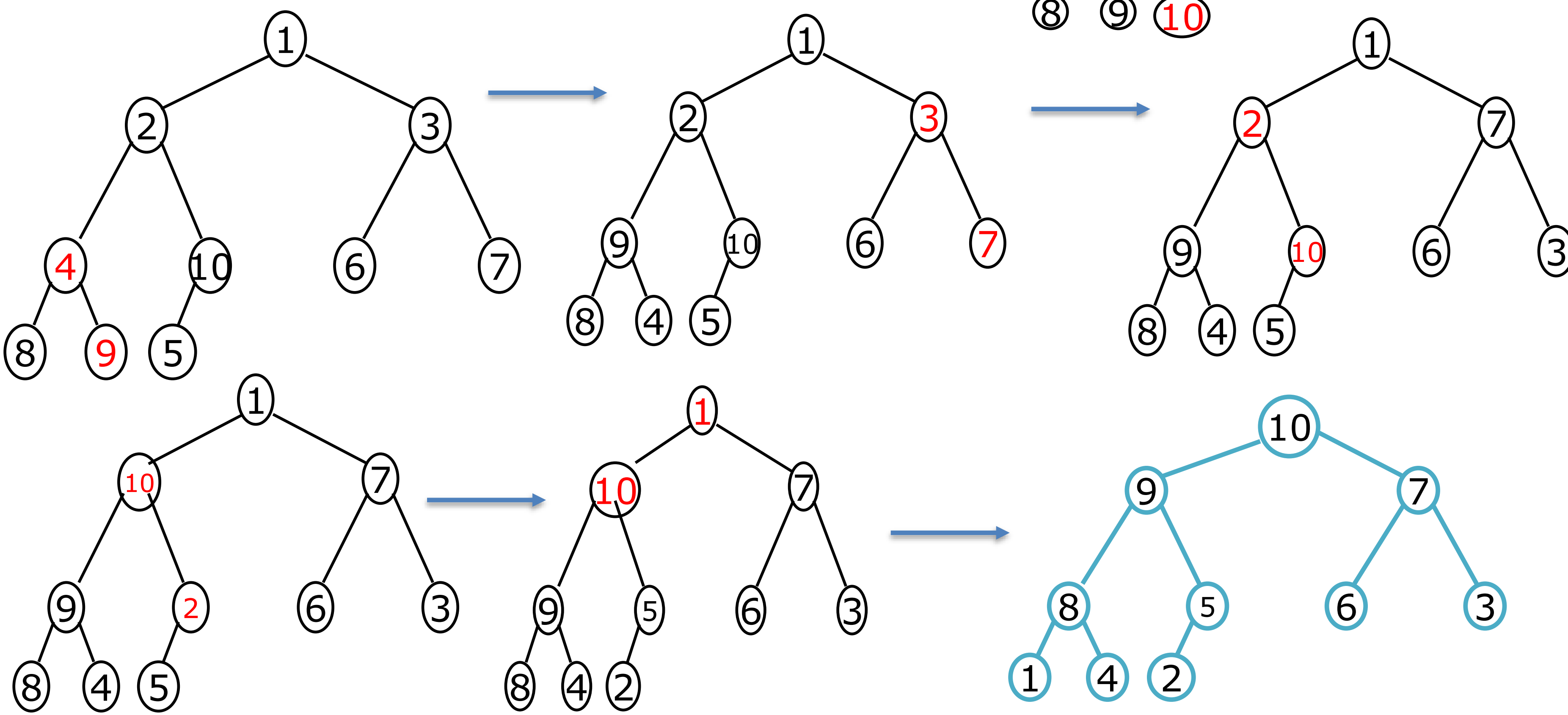
All leaf nodes are from $\lfloor n/2 \rfloor + 1$ *to* n
All non-leaf nodes are from 1 to $\lfloor n/2 \rfloor$



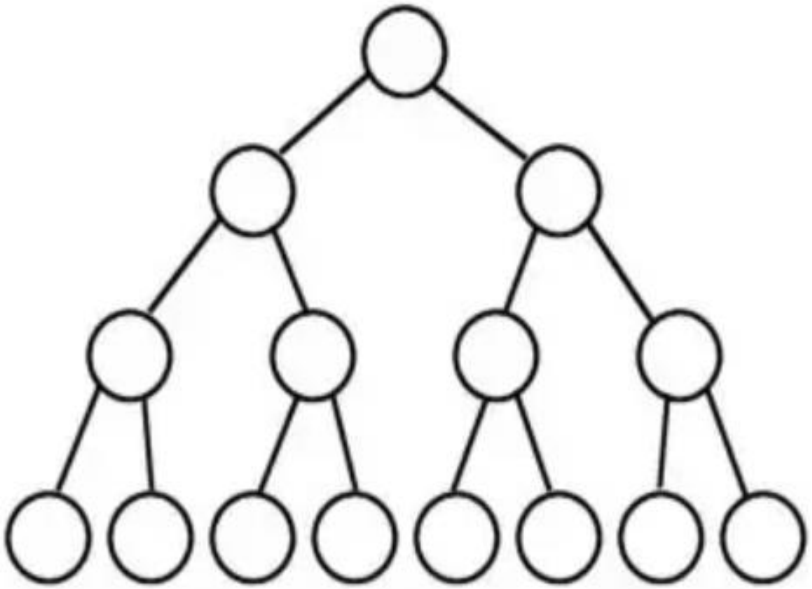
Build Heap

1	2	3	4	5	6	7	8	9	10
1	2	3	4	5	6	7	8	9	10





Build Heap Analysis



We are calling MAX_HEAPIFY() on n/2 nodes (as leaf's are already a heap).

Since we call MAX_HEAPIFY O(n) times and MAX_HEAPIFY takes O(log n), the overall complexity is O(n log n).

But his is not tight! We can also prove that building a heap is not O(nlogn) but just ***O(n).***

There is a beautiful proof about this in CLRS using progression. Please refer the same and start a discussion if required!

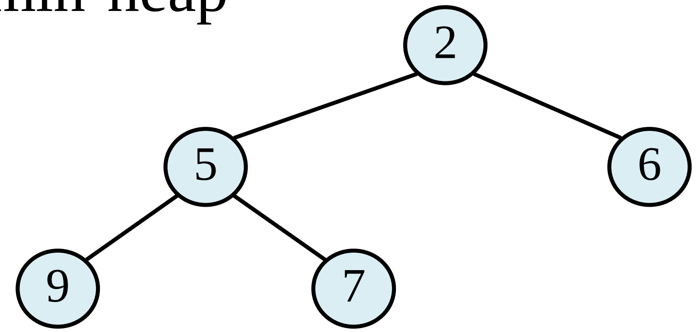
Insertion into a Heap

Inserting an element ***e*** in the heap has

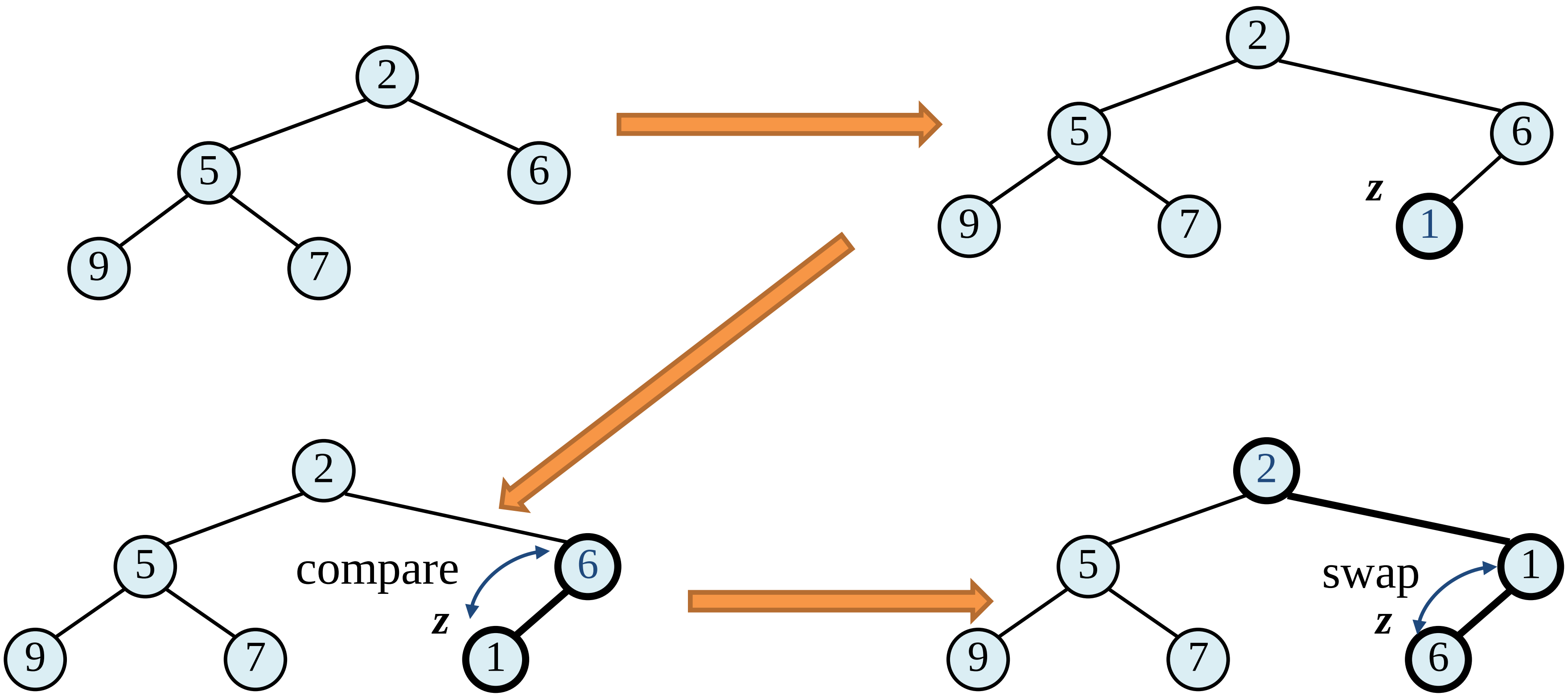
- ***Three steps***
  - ➢ Find the insertion point ***z***
    - o So that we maintain complete binary tree property
  - ➢ Store ***e*** at insertion point ***z***
  - ➢ Check if the heap follows heap-order property
    - o Restore the heap-order property by *Up-heap bubbling*

Example:

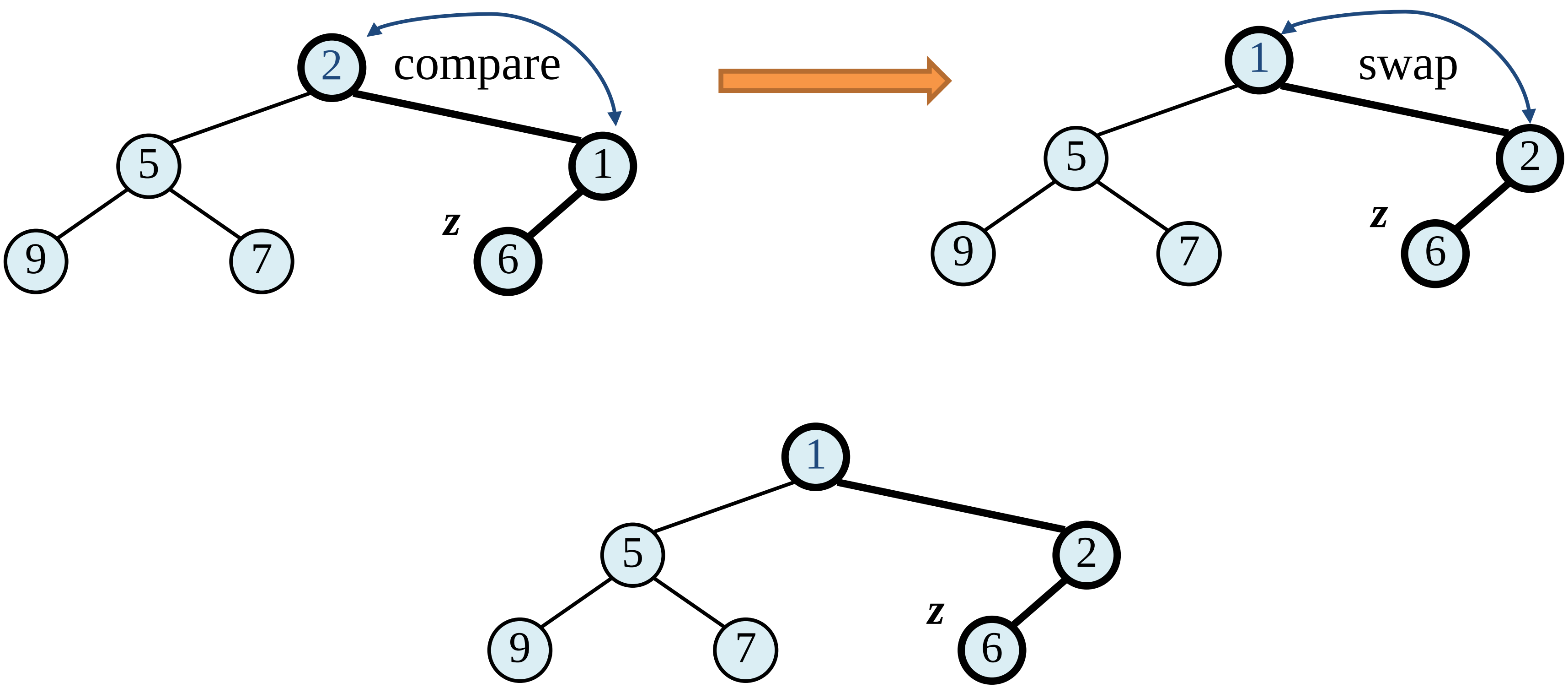
- ➢ Insert the element 1 into the min-heap



Insertion into a Heap



Insertion into a Heap



Insertion into a Heap

- After the insertion of a new element ***e***, the heap-order property may be violated.
- Up-heap bubbling restores the heap-order property
  - Compare and swap ***e*** along an upward path from the insertion point
  - Up-heap bubbling terminates when the element ***e*** reaches
    - the root
    - a node where the heap order property is satisfied
- Since the heap has a complete binary tree structure, its height = log n (where n is no of elements). In the worst case (element inserted at the bottom has to be swapped at every level from bottom to top up to the root node to maintain the heap property), 1 swap is needed on every level. Therefore, the maximum no of times this swap is performed is log n. Hence, Insertion in a heap takes **O(log n) time**.

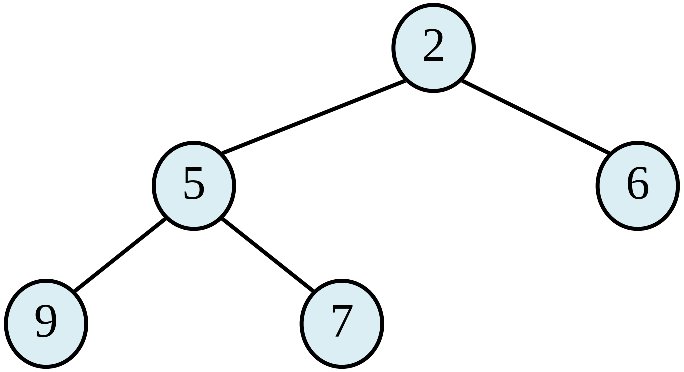
Removal from a Heap

Three steps

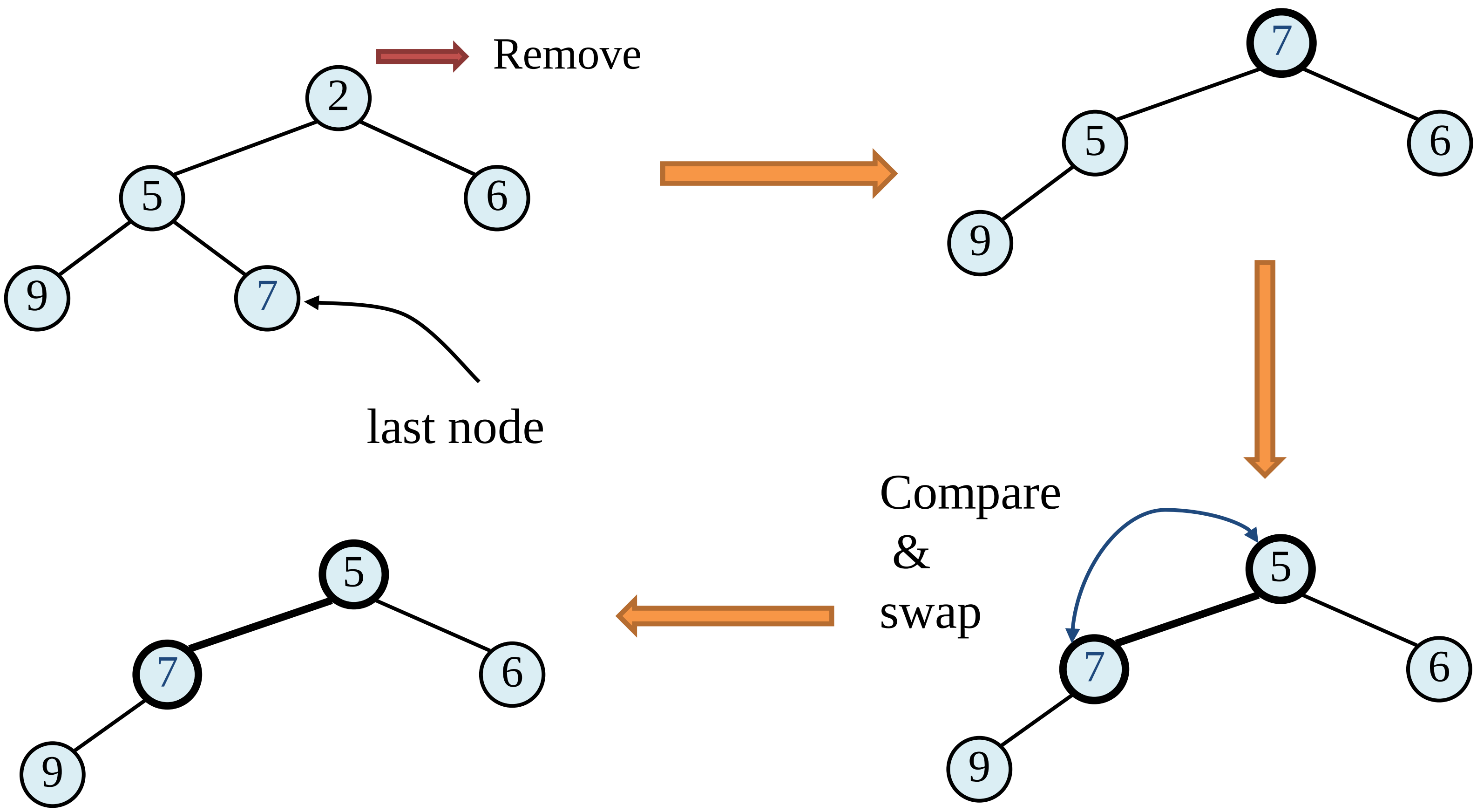
- Remove the element **at the root node** from the heap
- Fill the root node with the element from **the last node**
  - maintain Complete binary tree property
- Check if the heap follows the heap-order property
  - Restore the heap-order property by *down-heap bubbling*

Removal from a Heap

Example: Perform a delete operation on the given min-heap



Removal from a Heap



Removal from a Heap

- After replacing the root with the element from the last node ***e***, the heap-order property may be violated
- Down-heap bubbling restores the heap-order property
  - Compare and swap ***e*** along a downward path from the root node
    - Choose the eligible (min/max) child of ***e*** and swap it with ***e***
  - Down-heap terminates when the element ***e*** reaches
    - A leaf
    - A node where the heap order property is satisfied
- Here again, in worst case, we may have to perform down-heap bubbling till the node reaches the leaf. Complexity is **O(log n)**

Exercise 1

Min-Heap

- Illustrate the result of inserting the elements 35, 33, 42, 10, 14, 19 and 27 one at a time, into an initially empty binary min-heap in that order. Draw the resulting min-heap after each insertion.
- Perform 2 delete operation for the min-heap constructed in the earlier example.

Max Heap

- Illustrate the result of inserting the elements 35, 33, 42, 10, 14, 19 and 27 one at a time, into an initially empty binary max-heap in that order. Draw the resulting max-heap after each insertion.
- Perform 2 delete operation for the max-heap constructed in the earlier example.

Exercise 2

Consider a binary max-heap implemented using an array. Which one of the following array represents a binary max-heap?

25, 14, 16, 13, 10, 8, 12
25, 12, 16, 13, 10, 8, 14
25, 14, 12, 13, 10, 8, 16
25, 14, 13, 16, 10, 8, 12

Draw the heap structure and find out the right answer/s

Heap-Sort Algorithm

innovate achieve lead

In-Place: A sorting algorithm is said to be "in-place" if it moves the items within the array itself and, thus, requires only a small O(1) amount of extra storage.

- Heap sort is one of the best sorting methods being in-place and with no quadratic worst-case scenarios.
- Heap sort is divided into two basic parts :
  - Creating a heap of the unsorted list
  - Then a sorted array is created by repeatedly removing the largest/smallest element form the heap and inserting it into the array
  - *Heap is reconstructed after each removal*

Why study Heapsort?

- It is a well-known, traditional sorting algorithm you will be expected to know!!
- Heapsort is *always* O(n log n)
- Heapsort is a *really cool* algorithm!

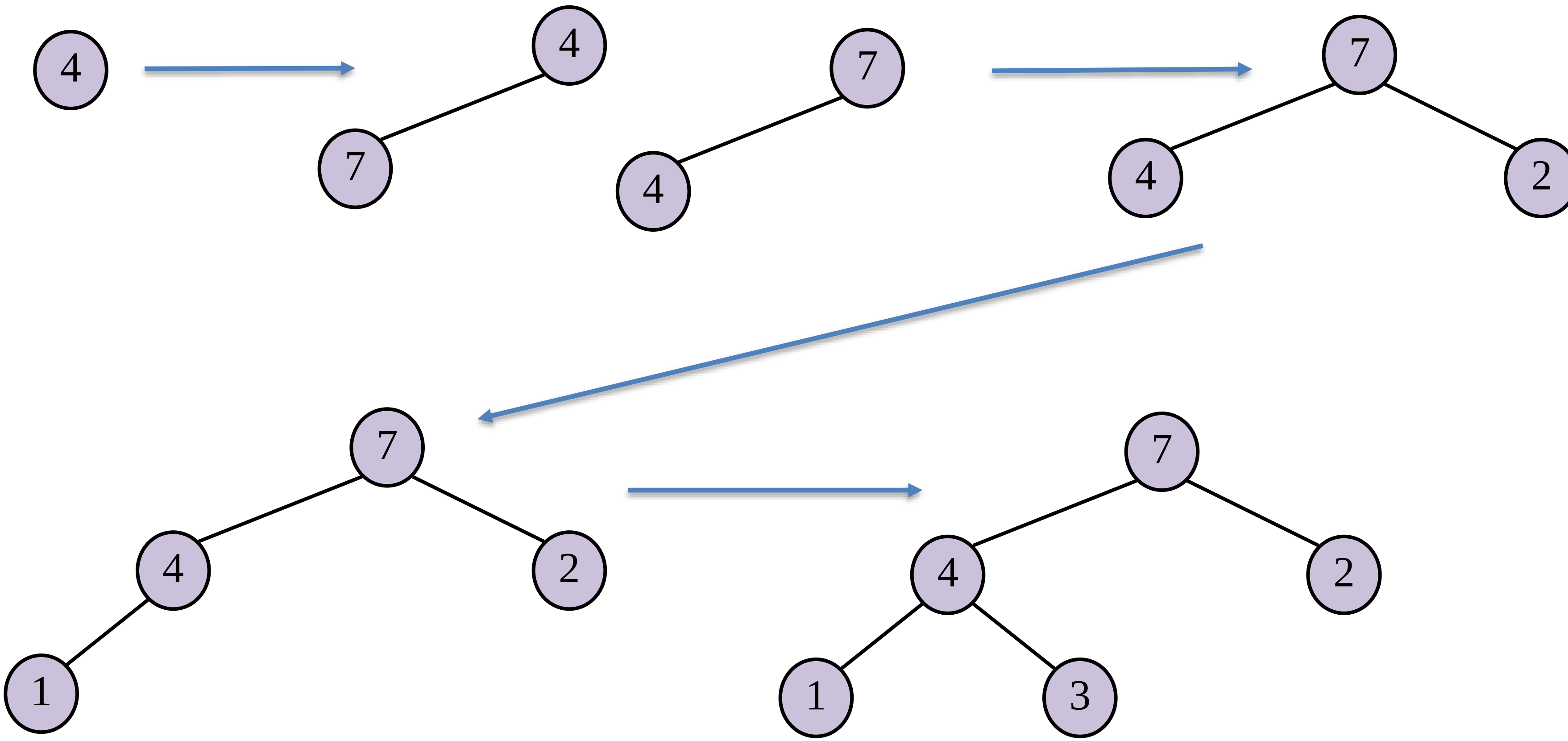
Heap Sort

- Given an array of n element, first we build the heap ..
- The largest element is at the root, but its position in sorted array should be at last. So swap the root with the last.
- We have placed the highest element in its correct position we left with an array of n-1 elements. Repeat the same of these remaining n-1 element to place the next largest elements in its correct position.
- Repeat the above step till an elements are placed in their correct positions.
- For increasing (ascending) order → Create a Max-Heap
- For a decreasing (descending) order → Create a Min-Heap

Heap Sort Example

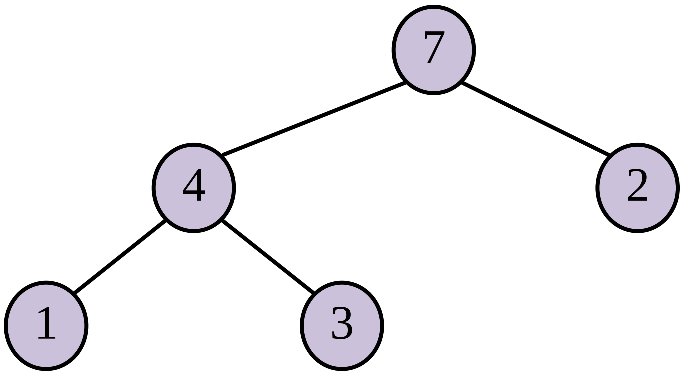
Illustrate heap sort for the given array S= [4, 7, 2, 1, 3]. Sort S in increasing order.

<u>First phase: Build max-heap</u>



Heap Sort Example

	S[1]	S[2]	S[3]	S[4]	S[5]
Initial	4	7	2	1	3
i=1	4	7	2	1	3
i =2	7	4	2	1	3
i =3	7	4	2	1	3
i =4	7	4	2	1	3
i =5	7	4	2	1	3
	Represents heap				
	Represents array elements not in heap				

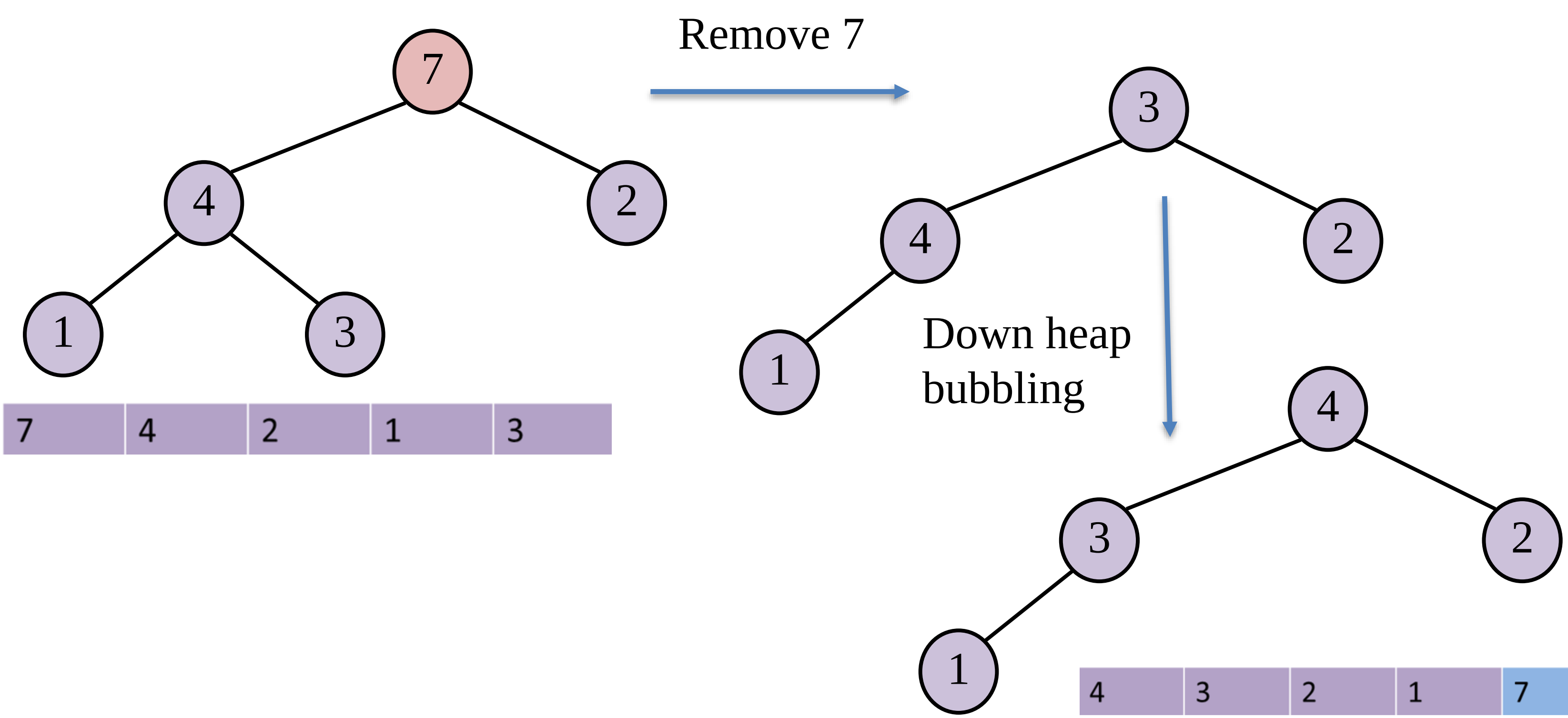


Phase 1: Heap Creation is completed!

Heap Sort Example

Heap S= [7, 4, 2, 1, 3]

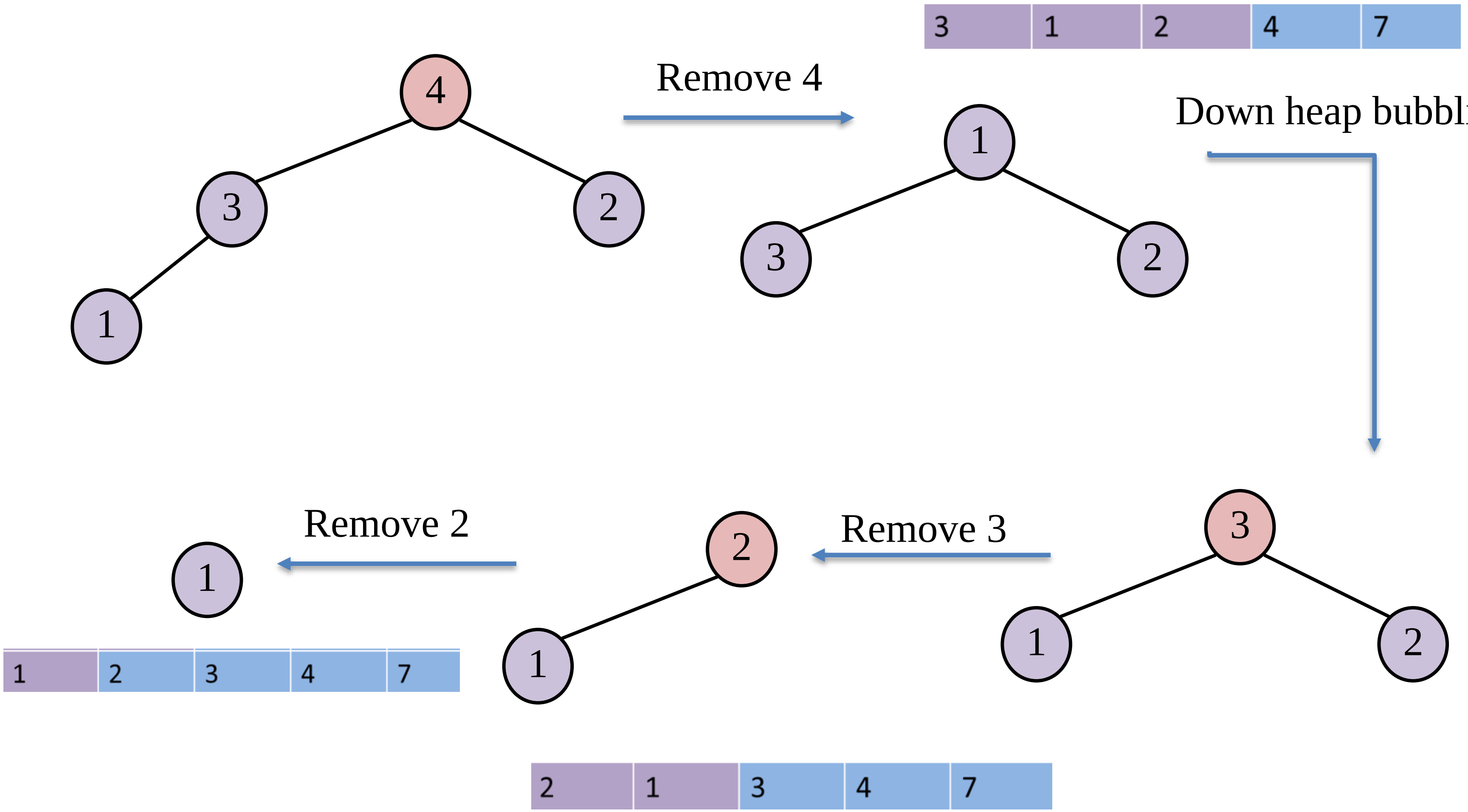
<u>Second phase:</u> Remove elements from heap and add them to the sorted array



7	4	2	1	3

4	3	2	1	7

Heap Sort Example



Heap Sort Example

	S[1]	S[2]	S[3]	S[4]	S[5]
Initial	7	4	2	1	3
i=1	4	3	2	1	7
i =2	3	1	2	4	7
i =3	2	1	3	4	7
i =4	1	2	3	4	7
i =5	1	2	3	4	7
	Represents heap				
	Sorted array				

Phase 2: Heap Deletion is completed! And result is sorted elements!!

Pseudocode for Heap Sort

innovate achieve lead

Heapsort(A)

1. Build-Max-Heap(A)
2. for i ← *length*[A] **downto** 2
3. **do** exchange A[1] ↔ A[i]
4. *heap-size*[A] ← *heap-size*[A]-1
5. Max-Heapify(A,1)

The time complexity of the heap sort algorithm is in ?
Phase 1 – Building the heap takes O(n)
Phase 2 – Remove the roots till we are left with only 1 element (log n)

Overall Complexity : O (n log n)

Exercise 3

- Given set of elements: 16,14,10,8,7,9,3,2,4,1

Exercise 1:

- Implement Heap Sort by showing each step and the resultant must be in increasing order. Hint: Create max-heap!

Exercise 2:

- Implement Heap Sort by showing each step and the resultant must be in decreasing order. Hint: Create min-heap!

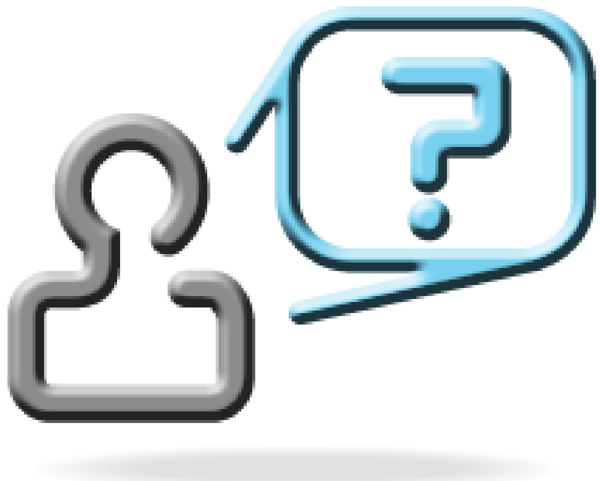
Exercise 4

Find K'th smallest element in an array using Heap.

```
Input:

arr = [7, 4, 6, 3, 9, 1]
k = 3

Output:

k'th smallest element in the array is 4
```

Procedure

1. Construct a min-heap of size 'N'
2. Pop first K-1 elements from it
3. Now K'th smallest element will reside at the root of the min-heap.



See you in the next class to explore Graphs!

Thank You for your time & attention !

Contact : parthasarathypd@wilp.bits-pilani.ac.in