Design and Analysis of Algorithms (UE18CS251)

Unit III - Transform-and-Conquer

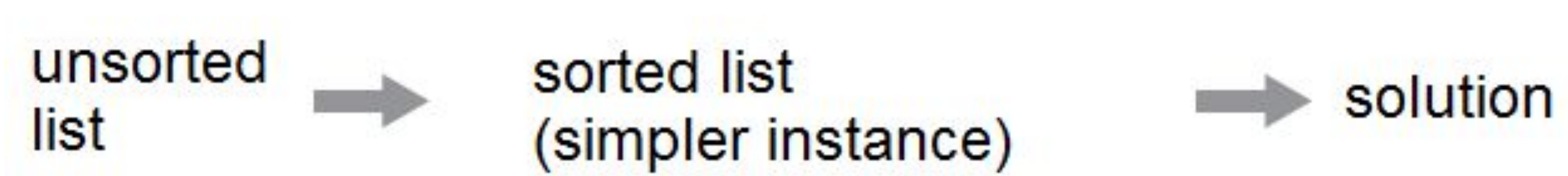
Mr. Channa Bankapur
channabankapur@pes.edu

Transform-and-Conquer:

problem's instance → simpler instance
or
another representation
or
another problem's instance → solution

Presorting:

Interest in sorting algorithms is due, to a significant degree, to the fact that many questions about a list are easier to answer if the list is sorted.



Finding the **largest element** in an array of **n** numbers using the following approaches:

1. Brute Force
2. Decrease-n-Conquer
3. Divide-n-Conquer
4. Transform-n-Conquer (Presorting-based)

Write an algorithm for:

Checking element uniqueness in an array

using **presorting-based** technique.

Analyze its time efficiency.

Compare with the brute force algorithm.

ALGORITHM *PresortElementUniqueness*$(A[0..n-1])$

//Solves the element uniqueness problem by sorting the array first
//Input: An array $A[0..n-1]$ of orderable elements
//Output: Returns "true" if A has no equal elements, "false" otherwise
sort the array A
for $i \leftarrow 0$ **to** $n-2$ **do**
    if $A[i] = A[i+1]$ **return false**
return true

$$\begin{aligned} T(n) = T_{sort}(n) + T_{scan}(n) &\in \Theta(n \log n) + \Theta(n) \\ &= \Theta(n \log n) \end{aligned}$$

Write an algorithm for:
Computing a mode in an array
using **presorting-based** technique.

Analyze its time efficiency.

Compare with the brute force algorithm.

ALGORITHM *PresortMode*$(A[0..n-1])$

```
//Computes the mode of an array by sorting it first
//Input: An array A[0..n − 1] of orderable elements
//Output: The array's mode
sort the array A
i ← 0                          //current run begins at position i
modefrequency ← 0              //highest frequency seen so far
while i ≤ n − 1 do
    runlength ← 1;   runvalue ← A[i]
    while i + runlength ≤ n − 1 and A[i + runlength] = runvalue
        runlength ← runlength + 1
    if runlength > modefrequency
        modefrequency ← runlength;   modevalue ← runvalue
    i ← i + runlength
return modevalue
```

Write an algorithm for:

Computing a mode in an array

using **presorting-based** technique.

Analyze its time efficiency.

Compare with the brute force algorithm.

$$T(n) = T_{sort}(n) + T_{scan}(n) \in \Theta(n \log n) + \Theta(n)$$
$$= \Theta(n \log n)$$

Write an algorithm for:

Searching an element in an array

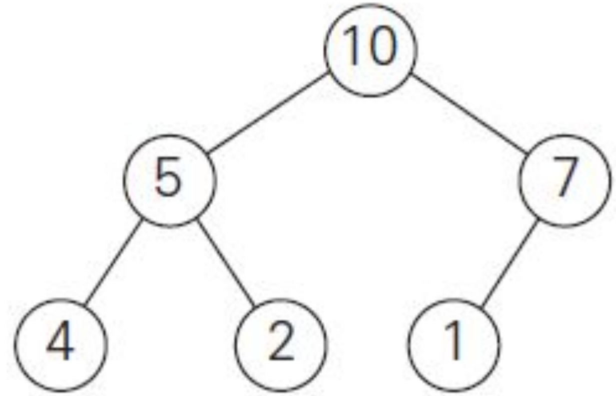
using **presorting-based** technique.

Analyze its time efficiency.

Compare with the brute force algorithm.

$$
\begin{aligned}
T(n) = T_{sort}(n) + T_{search}(n) &\in \Theta(n \log n) + \Theta(\log n) \\
&= \Theta(n \log n)
\end{aligned}
$$

Heaps:

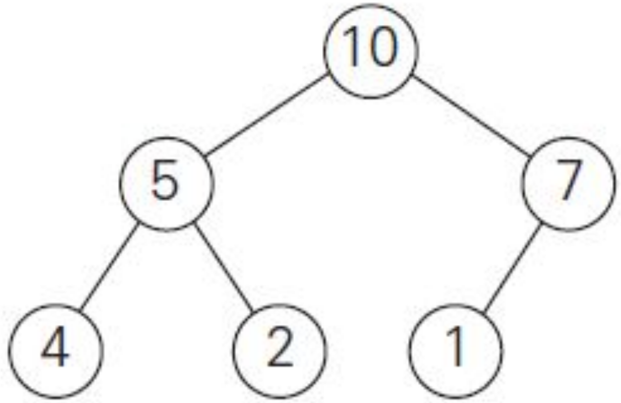


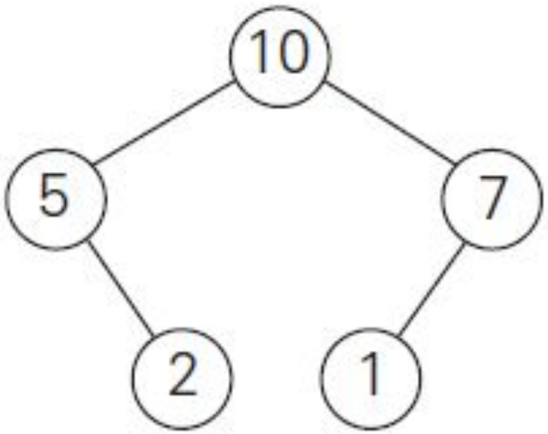
A ***heap*** can be defined as a binary tree with keys assigned to its nodes, one key per node, provided the following two conditions are met:

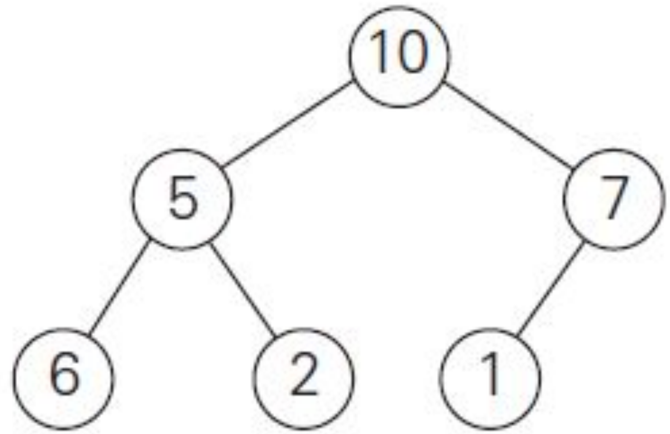
1. The ***shape property***—the binary tree is ***essentially complete*** (or simply ***complete***), i.e., all its levels are full except possibly the last level, where only some rightmost leaves may be missing.

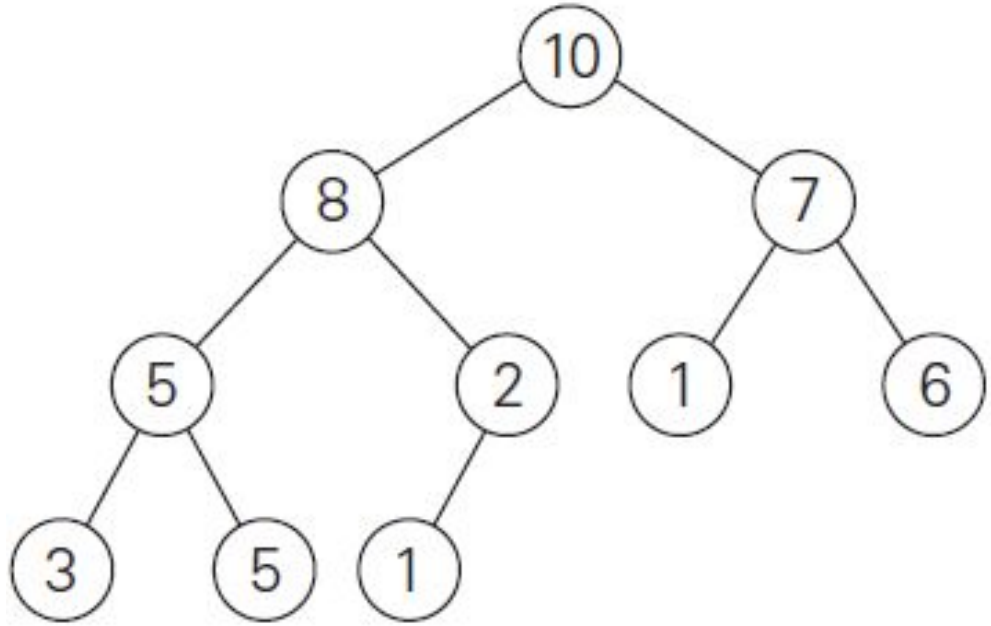
2. The ***parental dominance*** or ***heap property***—the key in each node is greater than or equal to the keys in its children. (This condition is considered automatically satisfied for all leaves.)

Heaps:
Which of the following are heaps?









the array representation

index	0	1	2	3	4	5	6	7	8	9	10
value		10	8	7	5	2	1	6	3	5	1

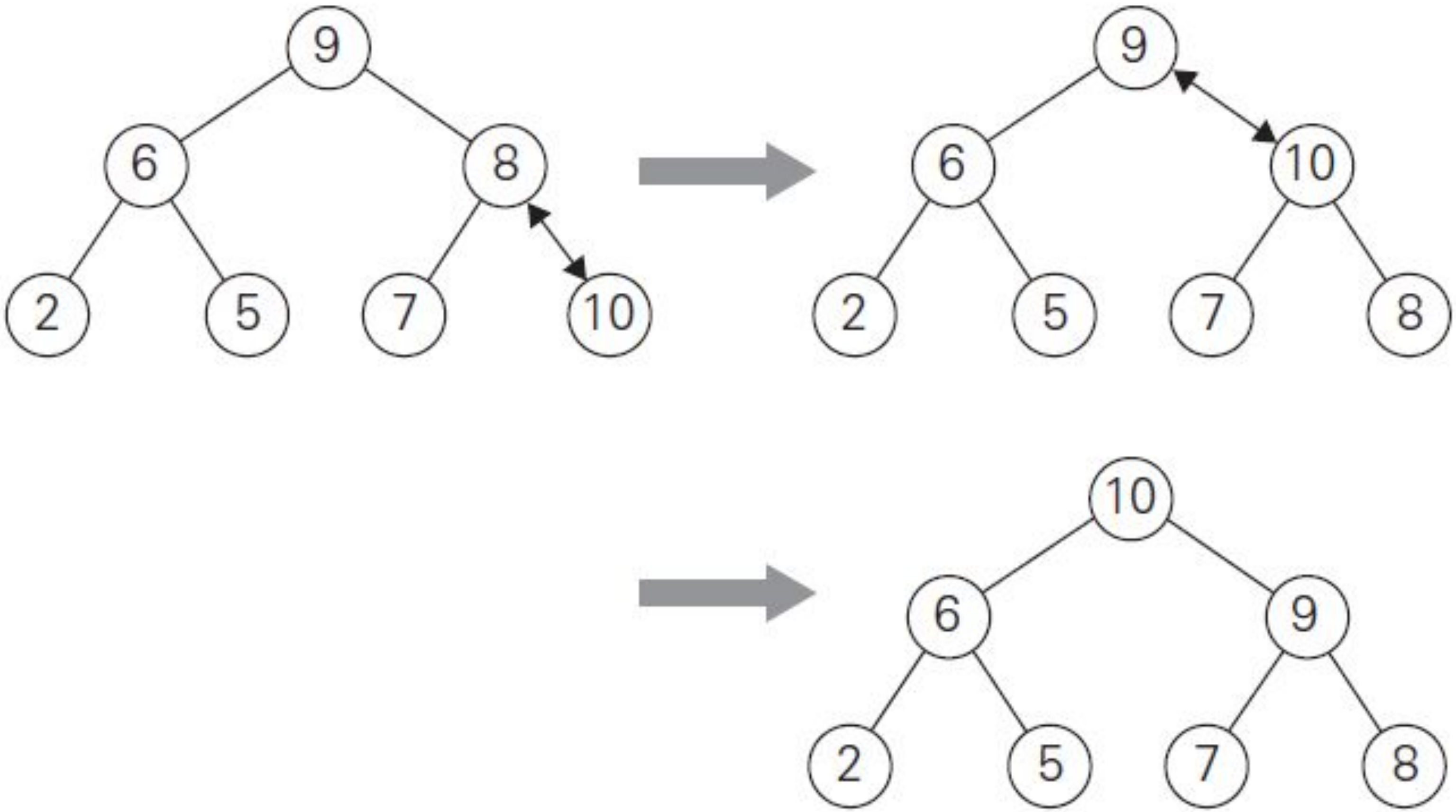
parents | leaves

A heap can be implemented as an array by recording its elements in the top-down, left-to-right fashion (BFS-way). In such a representation (for convenience let's store the heap's elements in positions 1 through n of the array),

- the parental node keys will be in the first $\lfloor n/2 \rfloor$ positions of the array, while the leaf keys will occupy the last $\lceil n/2 \rceil$ positions.

- the children of a key in the array's parental position **i** $(1 \le i \le \lfloor n/2 \rfloor)$ will be in positions **2i** and **2i+1**, and, correspondingly, the parent of a key in position **j** $(2 \le j \le n)$ will be in position $\lfloor j/2 \rfloor$.

Inserting a new element in the heap:
Add new element **‘10’** to the existing heap.



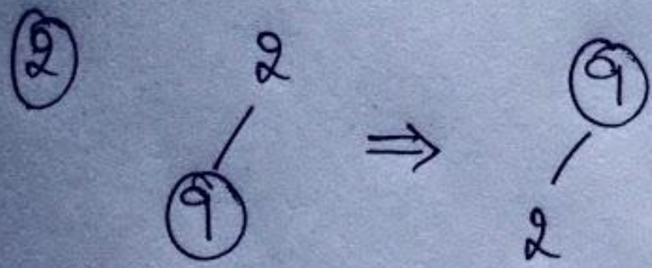
Construction of a heap from **top-down**:

2, 9, 7, 6, 5, 8, 10, 3, 6, 9

2

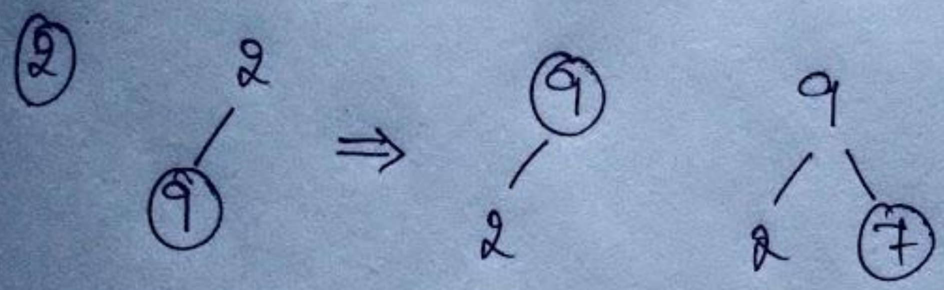
Construction of a heap from **top-down**:

2, 9, 7, 6, 5, 8, 10, 3, 6, 9

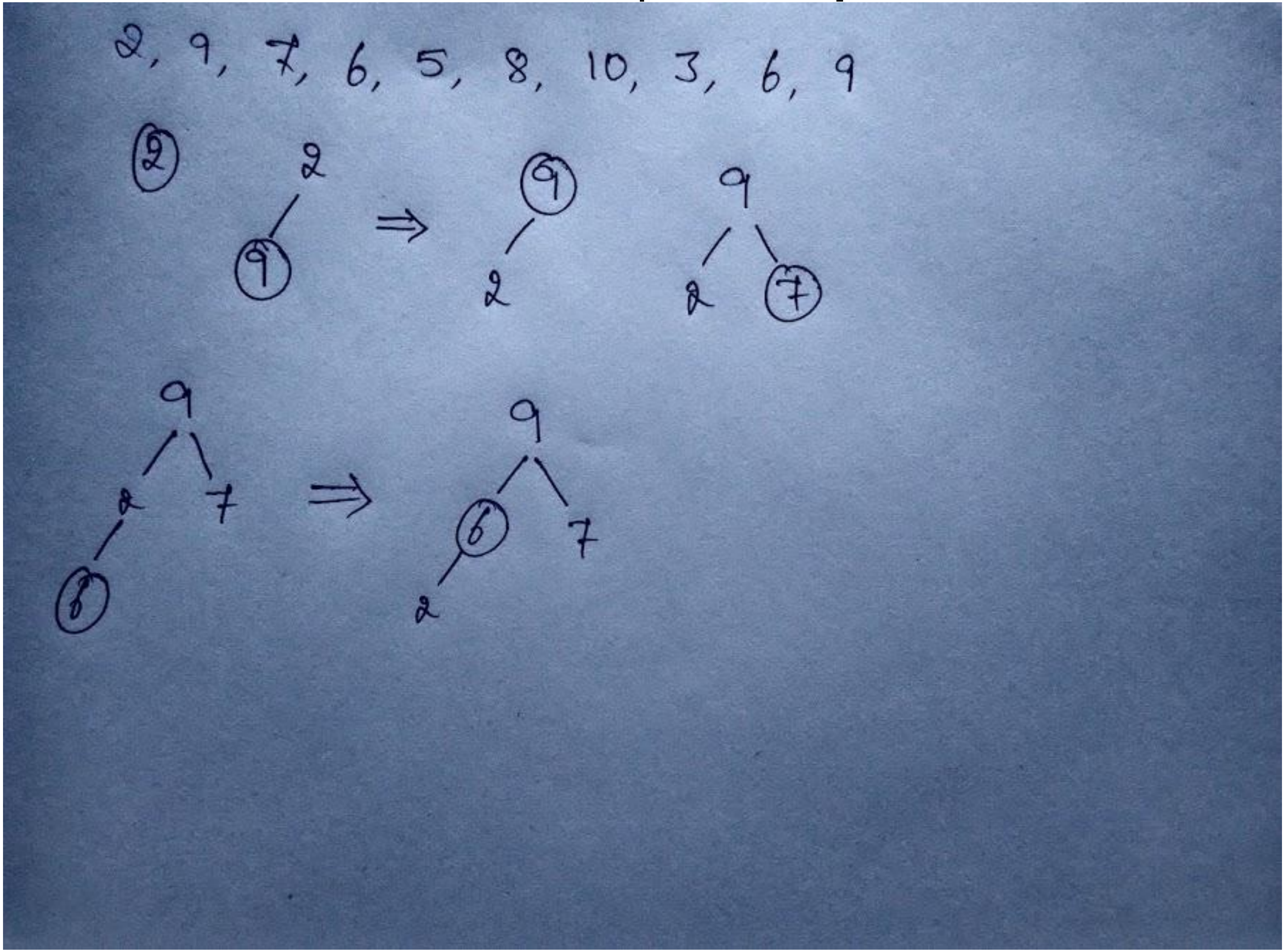


Construction of a heap from **top-down**:

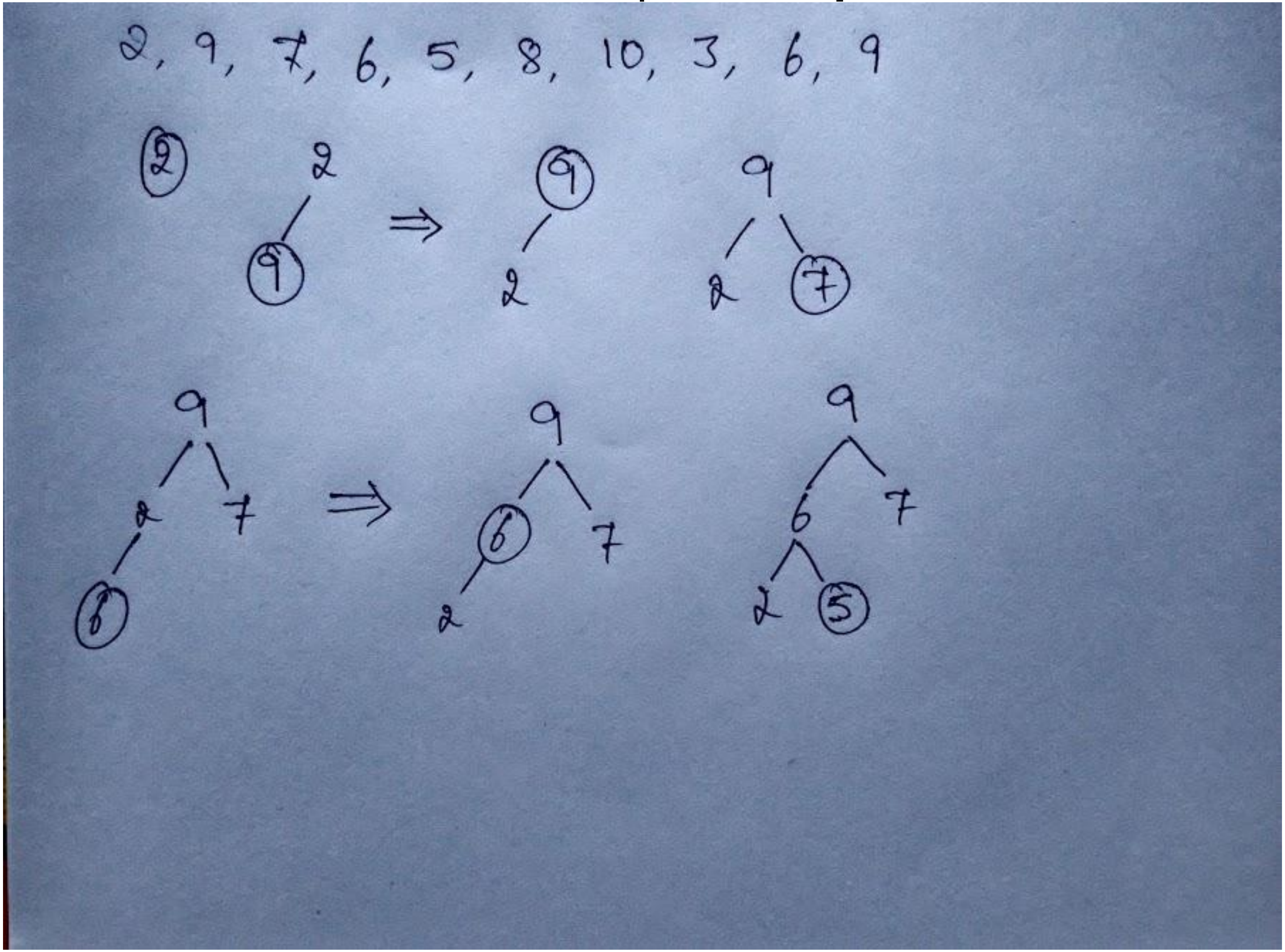
2, 9, 7, 6, 5, 8, 10, 3, 6, 9



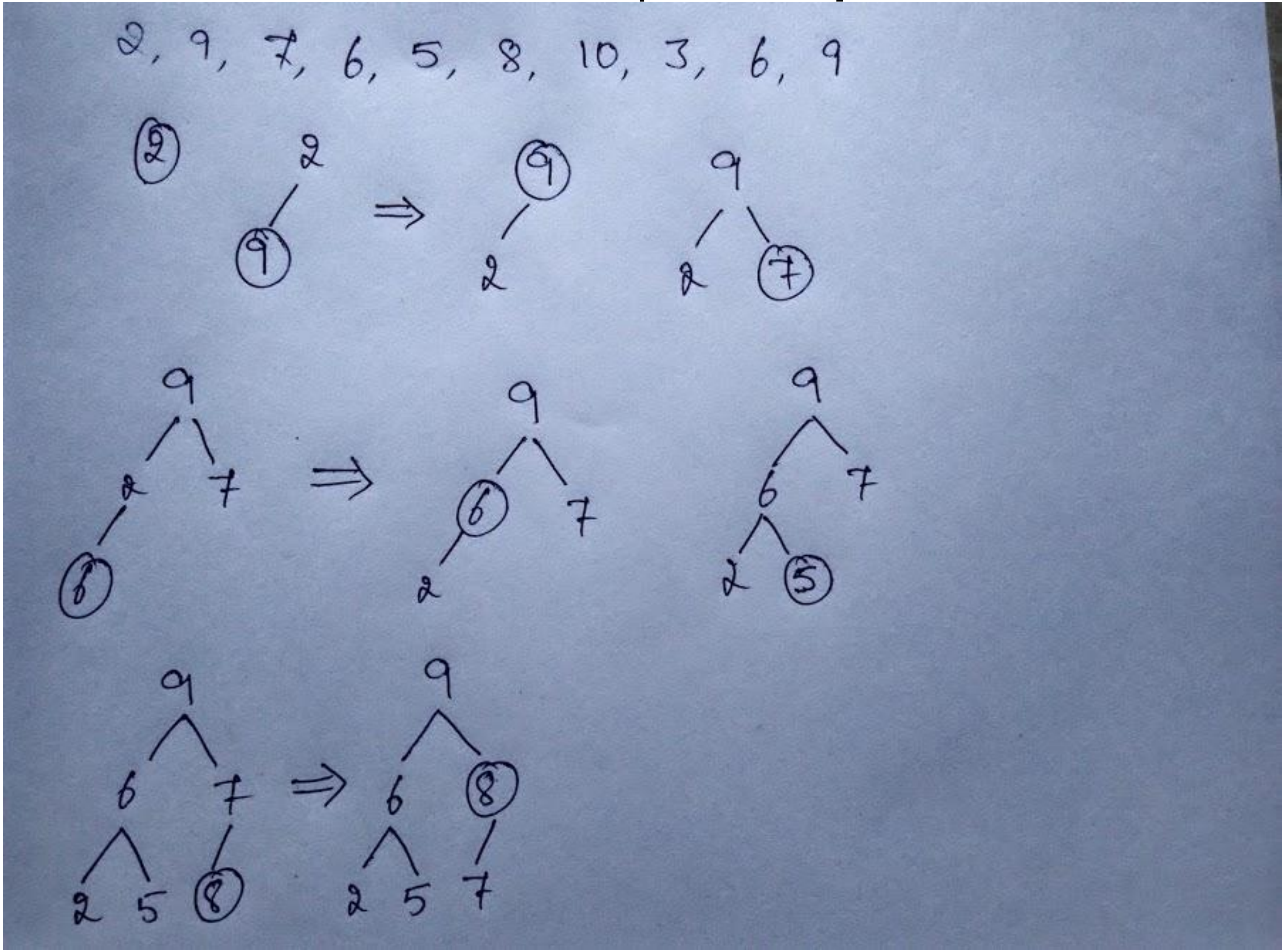
Construction of a heap from **top-down**:



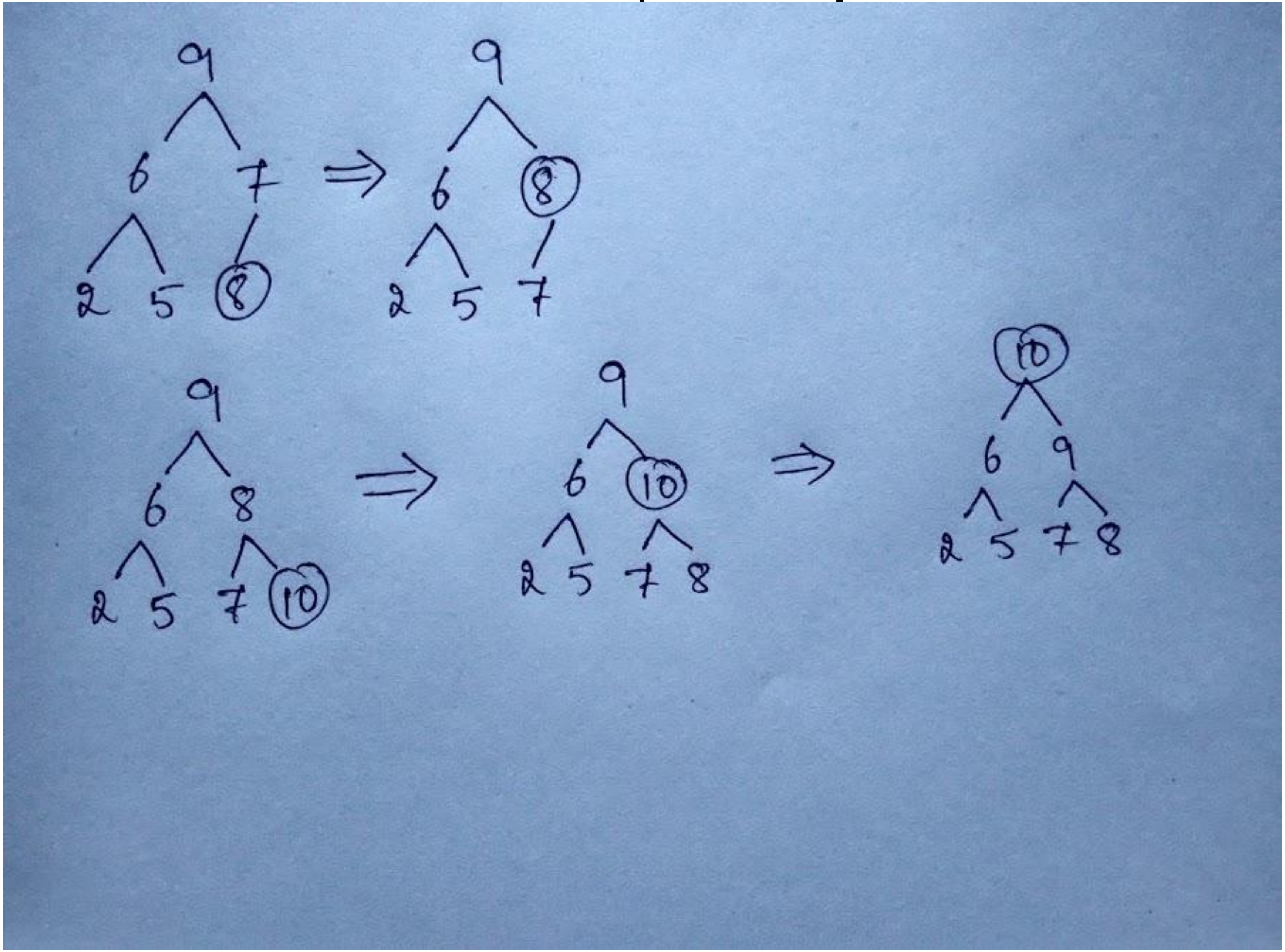
Construction of a heap from **top-down**:



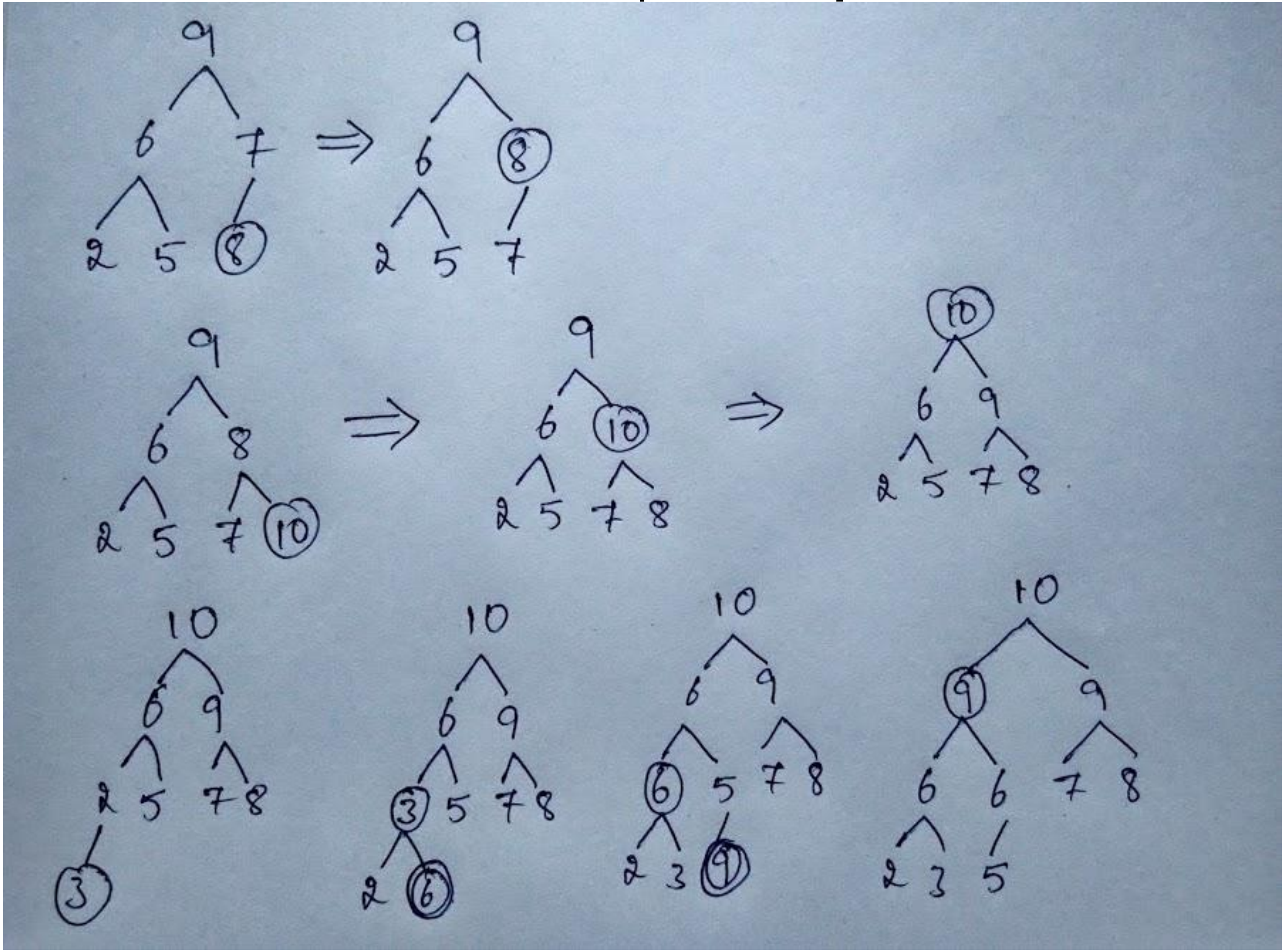
Construction of a heap from **top-down**:



Construction of a heap from **top-down**:

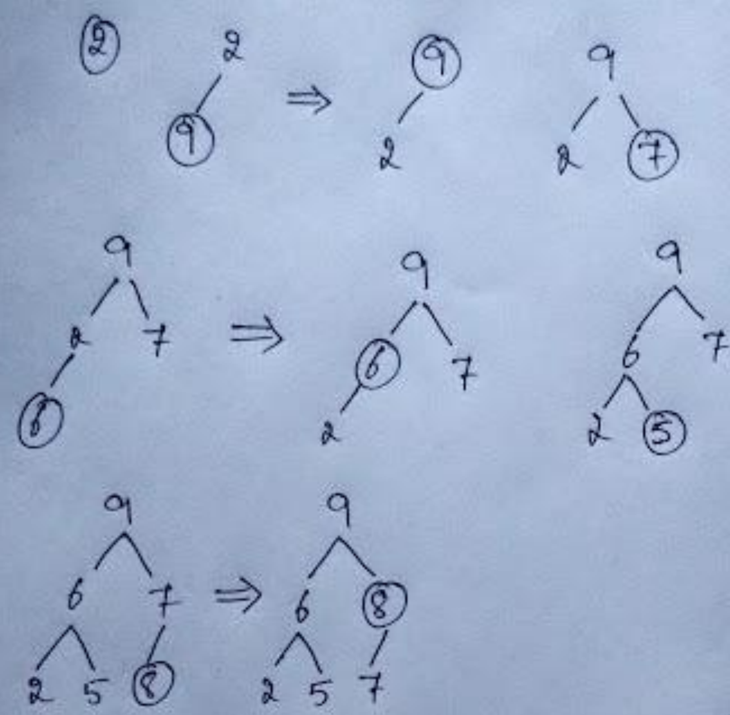


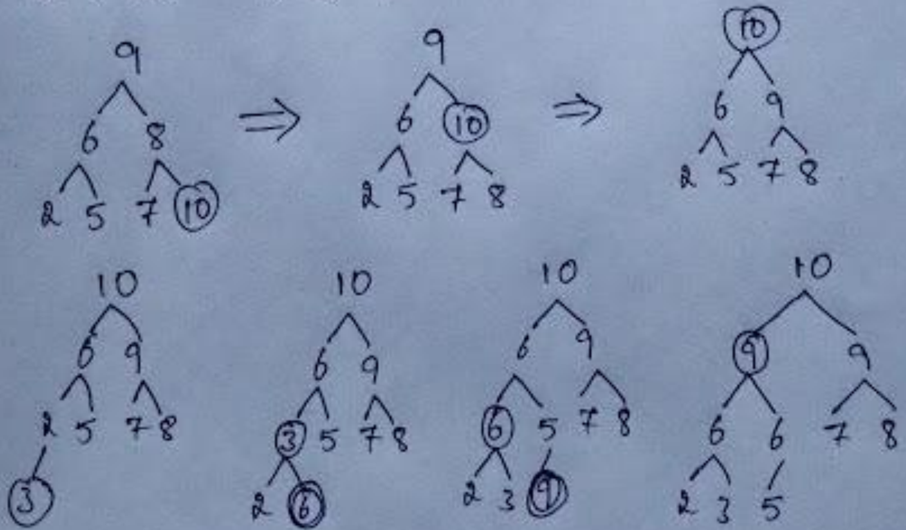
Construction of a heap from **top-down**:



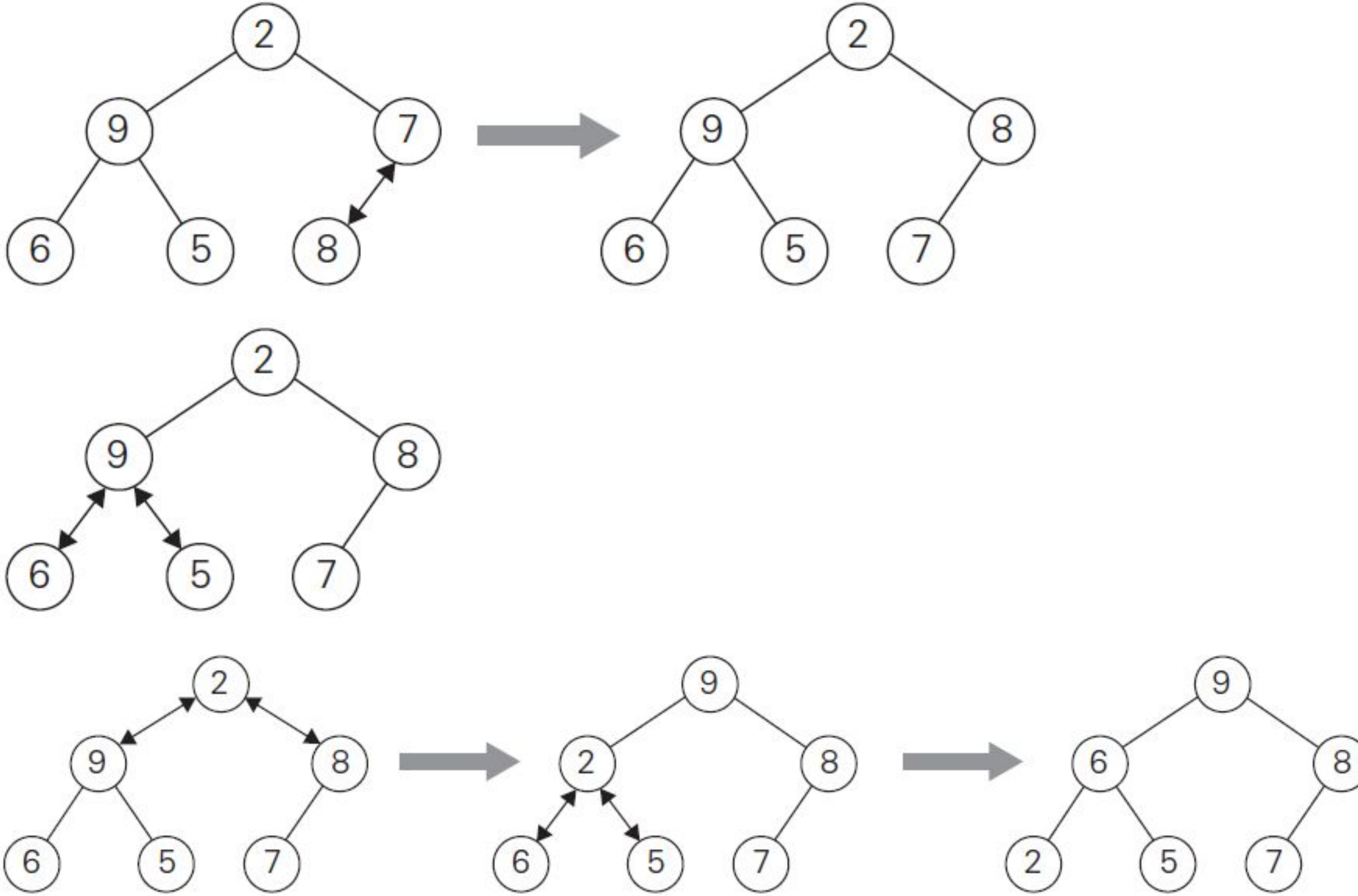
Heap Construction by top-down approach

2, 9, 7, 6, 5, 8, 10, 3, 6, 9



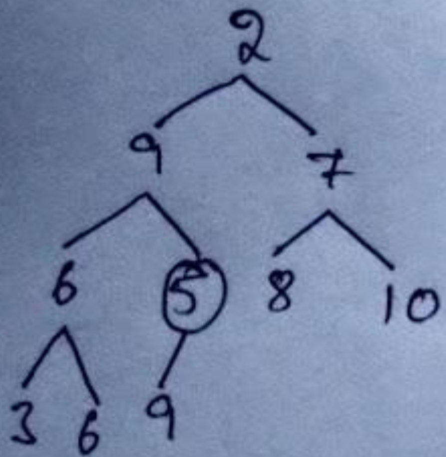


Bottom-up construction of a heap for the list 2, 9, 7, 6, 5, 8.



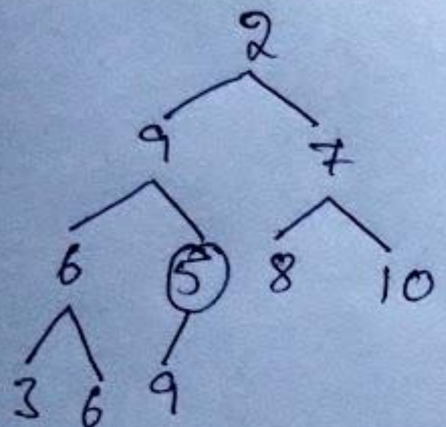
2, 9, 7, 6, 5, 8, 10, 3, 6, 9

Heap Construction by bottom-up approach.

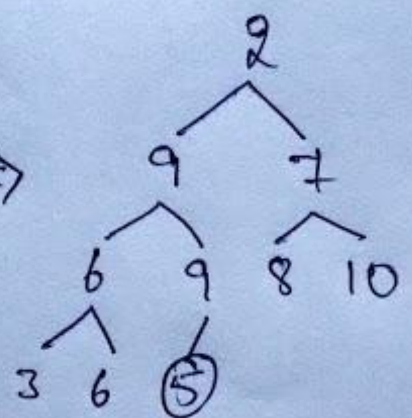


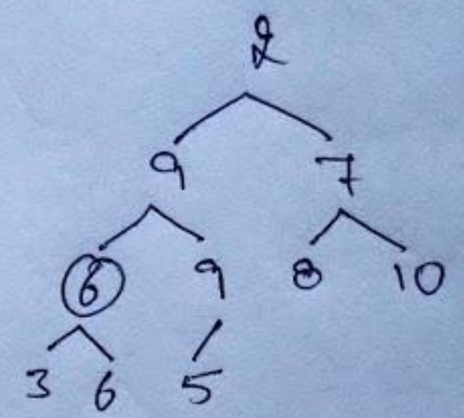
Heap Construction by bottom-up approach.

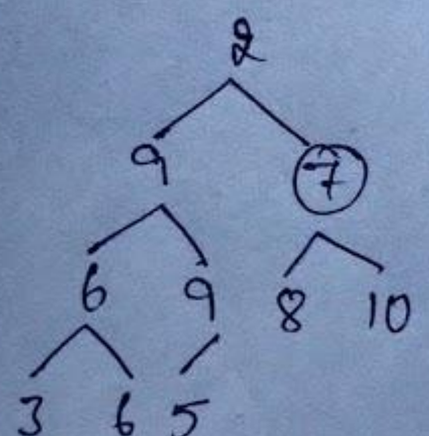
2, 9, 7, 6, 5, 8, 10, 3, 6, 9



⇒

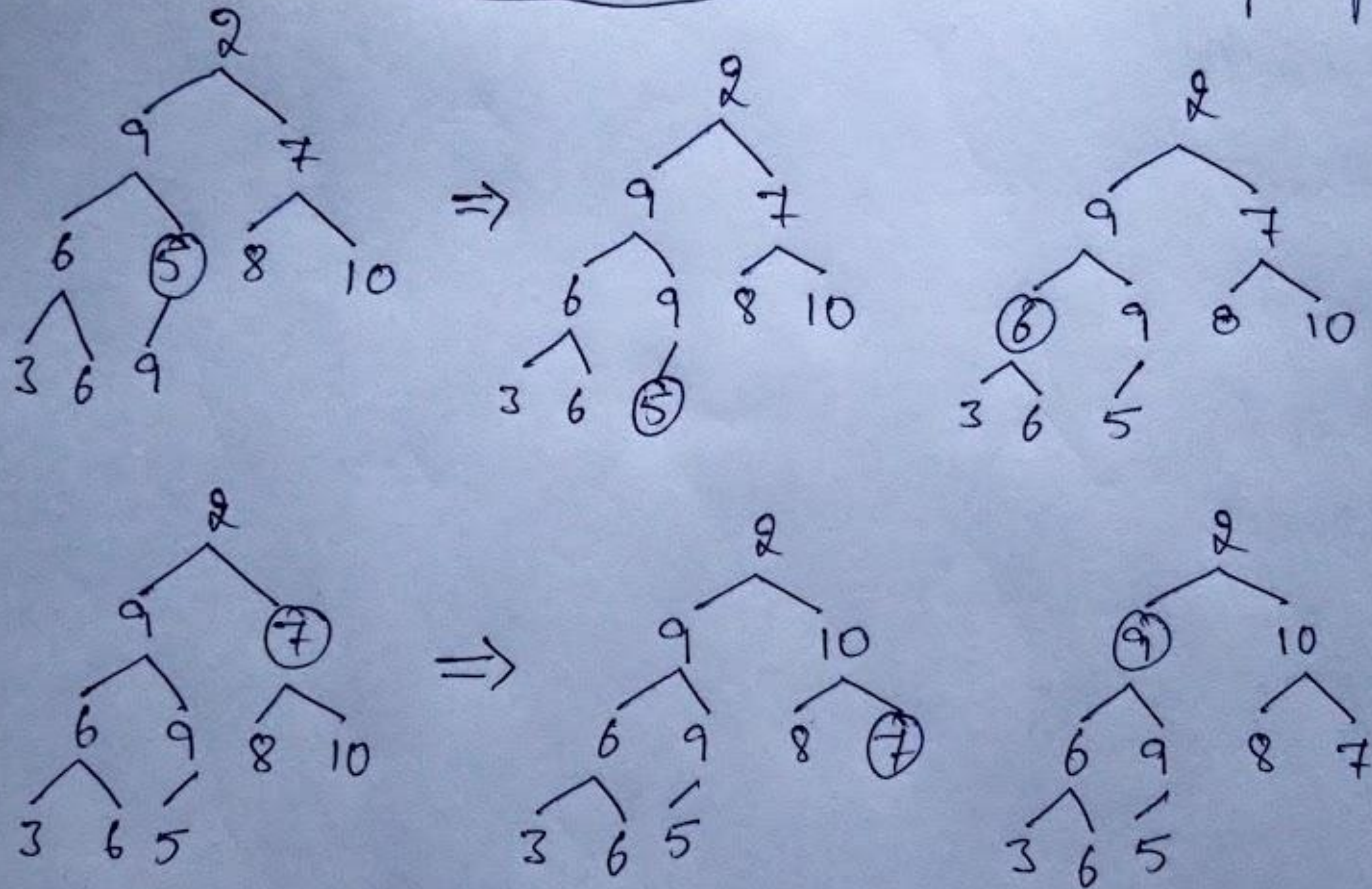






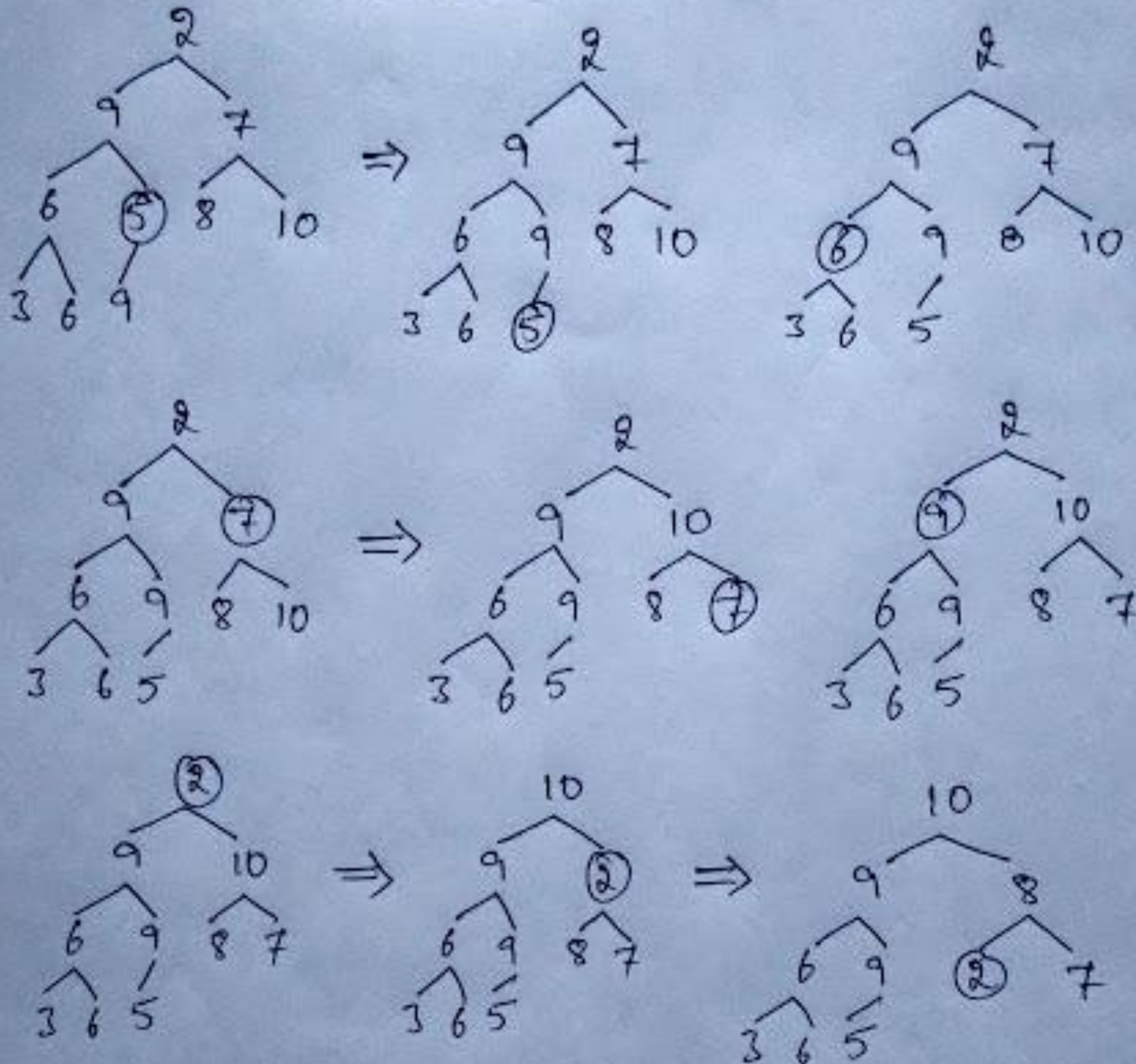
2, 9, 7, 6, 5, 8, 10, 3, 6, 9

Heap Construction by bottom-up approach.



Heap Construction by bottom-up approach.

2, 9, 7, 6, 5, 8, 10, 3, 6, 9



```
HeapBottomUp(H[1..n])
   if(n ≤ 1) return
   for i ← ⌊n/2⌋ downto 1 do
      Heapify(H, i)

//For the subtree rooted at k, it sifts down H[k]
//as much as possible until it becomes a heap.
Heapify(H[1..n], k)
   if(2*k > n) return  //if H[k] is a leaf
   j ← 2*k  //j points to left child of H[k]
   if(j+1 ≤ n)  //if there exists a right child of H[k]
      if(H[j+1] > H[j]) j ← j+1
   if(H[j] > H[k])  //if greater child is greater than H[k]
      H[j] ↔ H[k]
      Heapify(H, j)  //Heapify the subtree rooted at j
```

ALGORITHM $HeapBottomUp(H[1..n])$

```
//Constructs a heap from elements of a given array
// by the bottom-up algorithm
//Input: An array H[1..n] of orderable items
//Output: A heap H[1..n]
for i ← ⌊n/2⌋ downto 1 do
    k ← i;   v ← H[k]
    heap ← false
    while not heap and 2 ∗ k ≤ n do
        j ← 2 ∗ k
        if j < n   //there are two children
            if H[j] < H[j + 1] j ← j + 1
        if v ≥ H[j]
            heap ← true
        else H[k] ← H[j];   k ← j
    H[k] ← v
```

Efficiency of construction of heap from bottom-up:

Let h be the height of the tree.

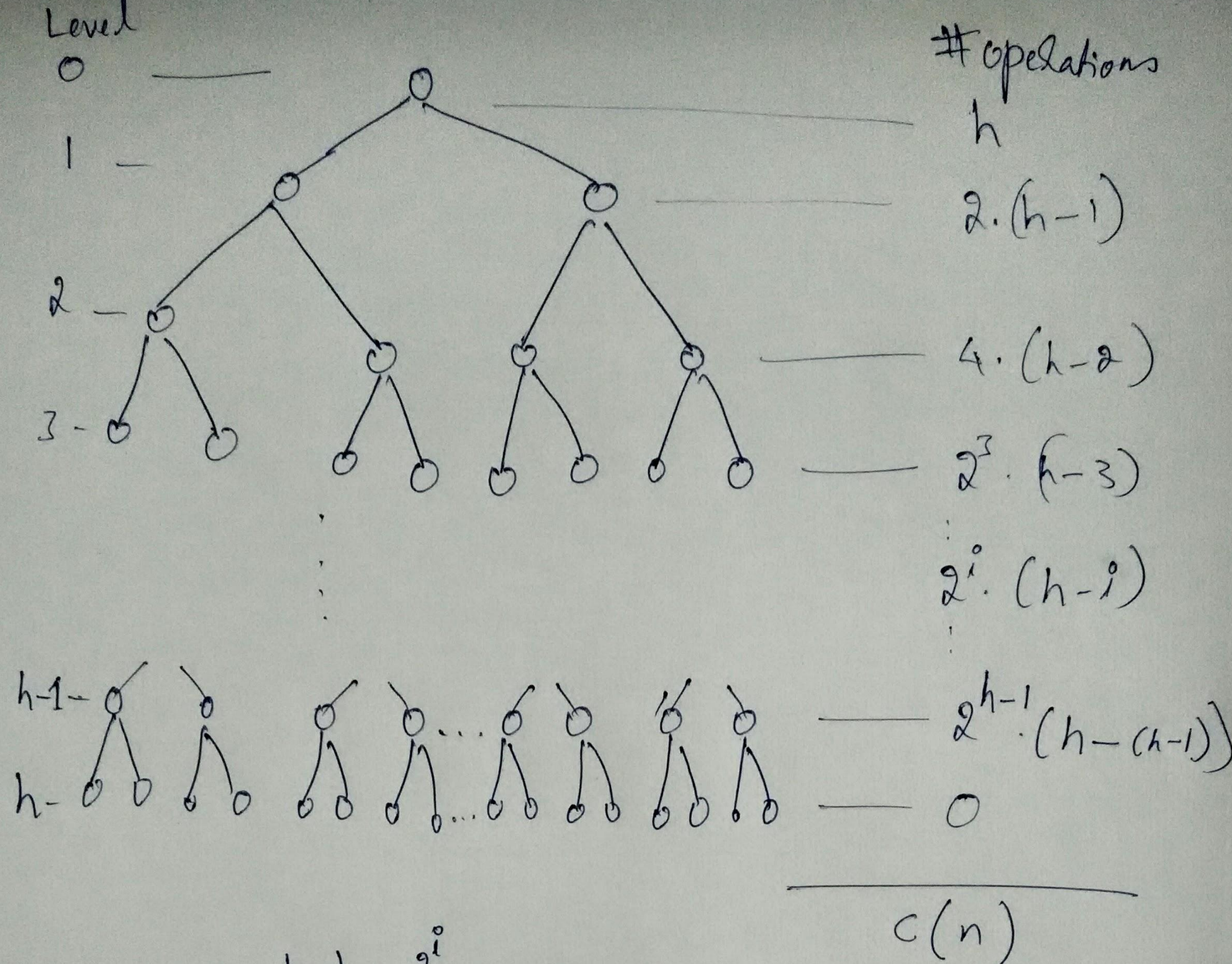
Two key comparisons at level of trickle down of an element.

$$h = \lfloor \log_2 n \rfloor$$

$$C_{worst}(n) = \sum_{i=0}^{h-1} \sum_{\text{level } i \text{ keys}} 2(h-i)$$

$$= \sum_{i=0}^{h-1} 2(h-i)2^i$$

≅ 2n ∈ Θ(n)



$$c(n) = \sum_{i=0}^{h-1} \sum_{j=1}^{2^i} (h-i)$$

$$c(n) = \sum_{i=0}^{h-1} 2^i (h-i)$$

$$C(n) = \sum_{i=0}^{h-1} 2^i (h-i)$$

$$= \left(h \sum_{i=0}^{h-1} 2^i\right) - \left(\sum_{i=0}^{h-1} i \cdot 2^i\right)$$

$$= h\left(2^h - 1\right) - \left((h-2)2^h + 2\right)$$

$$\because \sum_{i=1}^{n} i \cdot 2^i = (n-1)\,2^{n+1} + 2$$

$$C(n) = \cancel{h \cdot 2^h} - h - \cancel{h \cdot 2^h} + 2^{h+1} - 2$$

$$= 2^{h+1} - h - 2$$

$$h = \lfloor \log_2 n \rfloor$$

$$C(n) = 2^{1+\lfloor \log_2 n \rfloor} - h - 2 \in \Theta(n)$$

$$\sum_{i=1}^{n} i\,2^i = (n-1)\,2^{n+1} + 2 \quad ?$$

$$= 1\cdot 2^1 + 2\cdot 2^2 + 3\cdot 2^3 + 4\cdot 2^4 + \dots + n\cdot 2^n$$

$$\left.\begin{aligned}
= \; & 2^1 + 2^2 + 2^3 + 2^4 + \dots + 2^n \\
& \quad + 2^2 + 2^3 + 2^4 + \dots + 2^n \\
& \qquad + 2^3 + 2^4 + \dots + 2^n \\
& \qquad \vdots \qquad\qquad \vdots \\
& \qquad\qquad + 2^{n-1} + 2^n \\
& \qquad\qquad\qquad + 2^n
\end{aligned}\right\} n \text{ lines}$$

$$= \left(2^{n+1} - 2^1\right) + \left(2^{n+1} - 2^2\right) + \left(2^{n+1} - 2^3\right) + \dots + \left(2^{n+1} - 2^{n-1}\right) + \left(2^{n+1} - 2^n\right)$$

$$= n\cdot 2^{n+1} - \left(2^1 + 2^2 + \dots + 2^n\right)$$

$$= n\cdot 2^{n+1} - \left(2^{n+1} - 2\right) = \boxed{(n-1)\,2^{n+1} + 2}$$

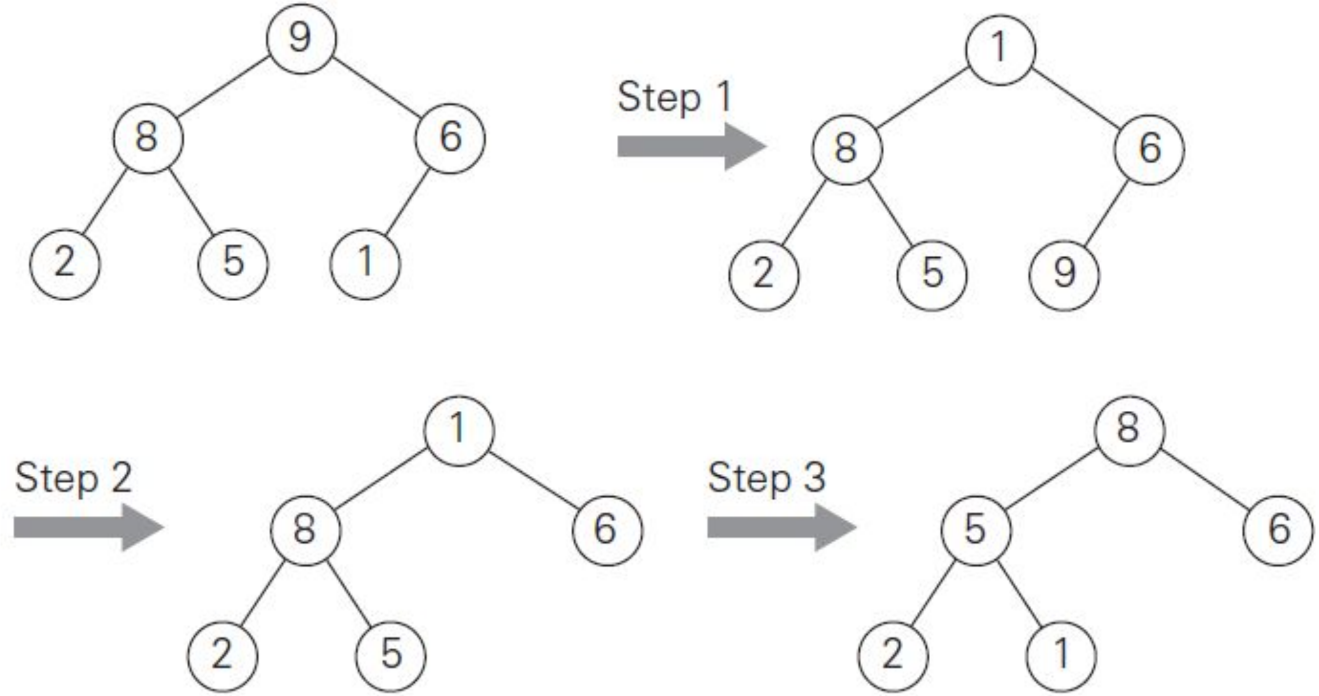
Heapsort discovered by J. W. J. Williams

This is a two-stage algorithm

Stage 1 (heap construction):
Construct a heap for a given array.

Stage 2 (maximum deletions):
Apply the root-deletion operation $n - 1$ times to the remaining heap.

Maximum
Key Deletion
from a heap



1. Exchange the root's key with the last key K of the heap.

2. Decrease the heap's size by 1.

3. "Heapify" the smaller tree by sifting K down the tree exactly in the same way we did it in the bottom-up heap construction algorithm.

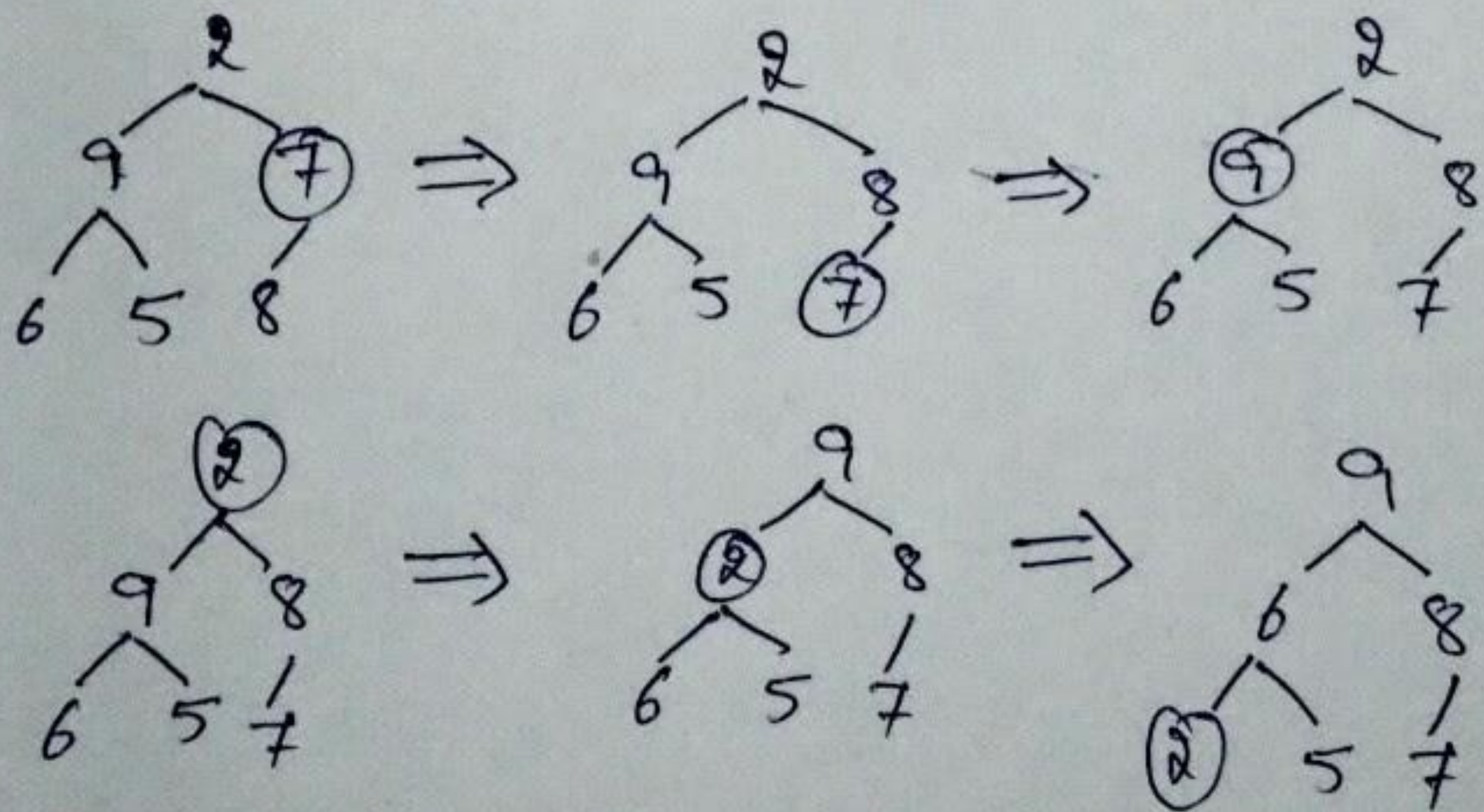
```
HeapSort(H[1..n])
   HeapBottomUp(H[1..n])  //Construct heap
   for i ← n downto 2 do
      H[1] ↔ H[i]  //H[1] has the max element.
      Heapify(H[1..i-1], 1)  //Sift down H[1]

HeapBottomUp(H[1..n])
   if(n ≤ 1) return
   for i ← ⌊n/2⌋ downto 1 do
      Heapify(H, i)

Heapify(H[1..n], k)
   if(2*k > n) return  //if H[k] is a leaf
   j ← 2*k  //j points to left child of H[k]
   if(j+1 ≤ n)  //if there exists a right child of H[k]
      if(H[j+1] > H[j]) j ← j+1
   if(H[j] > H[k])  //if greater child is greater than H[k]
      H[j] ↔ H[k]
      Heapify(H, j)  //Heapify the subtree rooted at j
```

2	9	7	6	5	8

Unsorted array

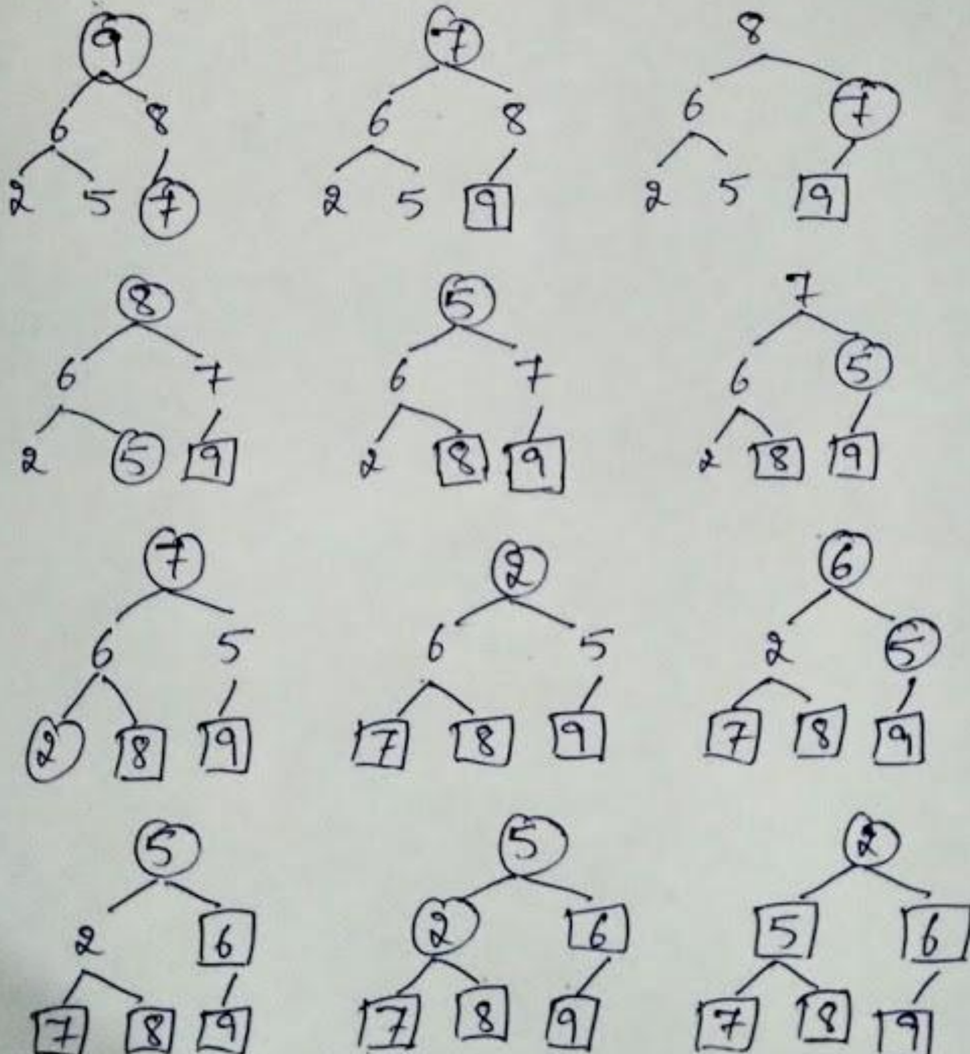


9	6	8	2	5	7

Heap

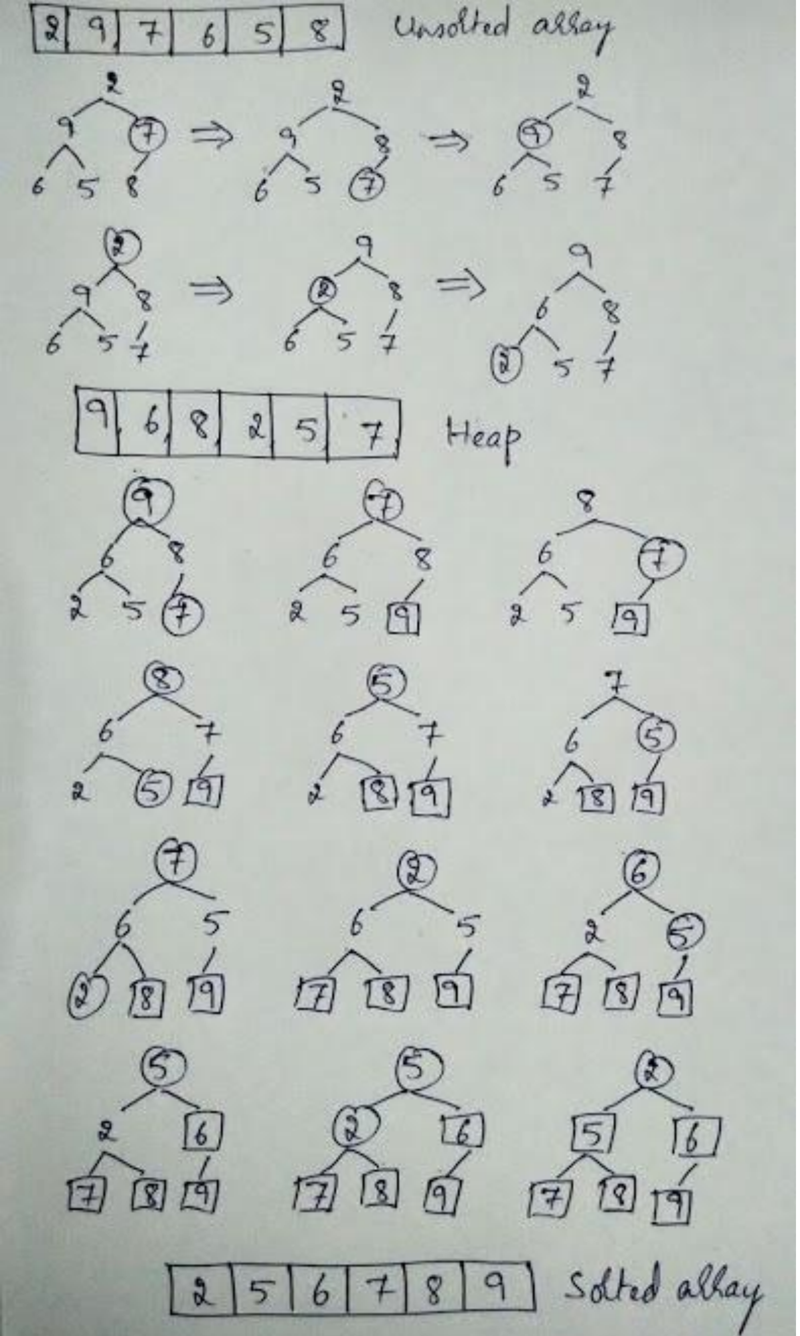
9	6	8	2	5	7

Heap



2	5	6	7	8	9

Sorted array



Analysis of Heapsort:

$$T_{Heapsort}(n) = T_{Heap}(n) + T_{Sort}(n)$$

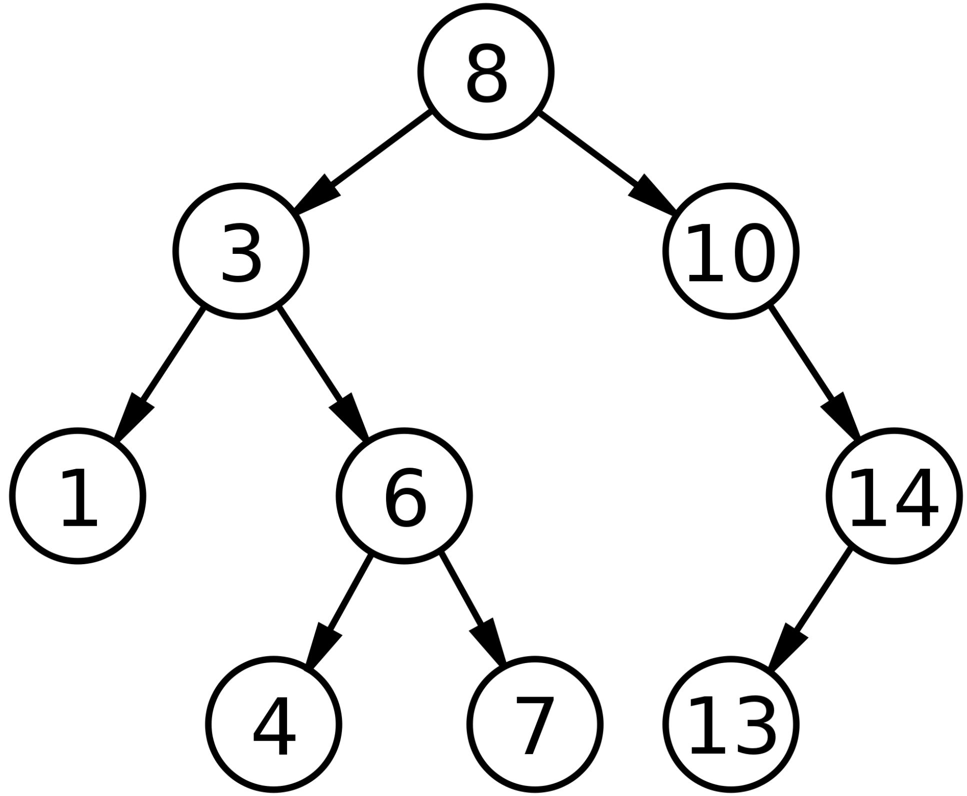
$$T_{Heapsort}(n) \in \max\{\Theta(n), \Theta(n \log n)\}$$

$$T_{Heapsort}(n) \in \Theta(n \log n)$$

Binary Trees

Binary Search Trees (BST)

What do we "conquer" by transforming a Binary Tree into a BST?



[Optional]

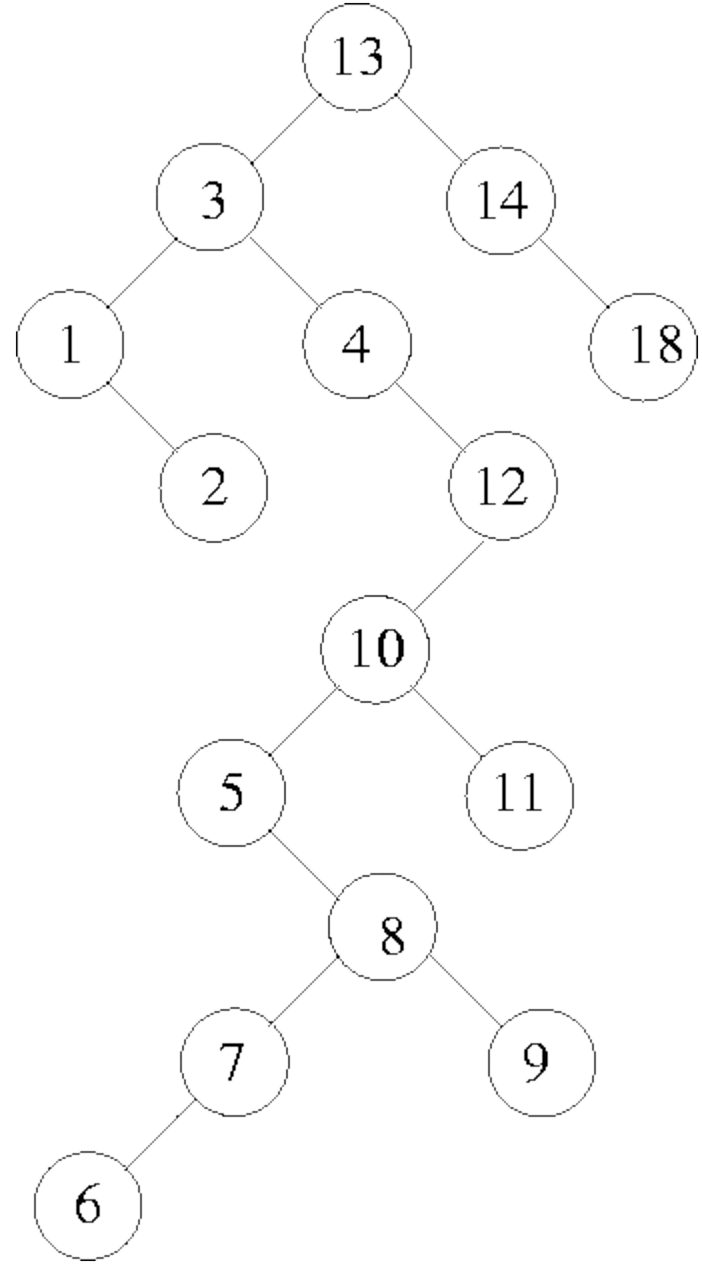
- Boolean isBST(BinaryTree t);
- BST BT2BST(BinaryTree t); //do it in-place

Binary Trees

Binary Search Trees (BST)

What do we “conquer” by transforming a Binary Tree into a BST?
Search!

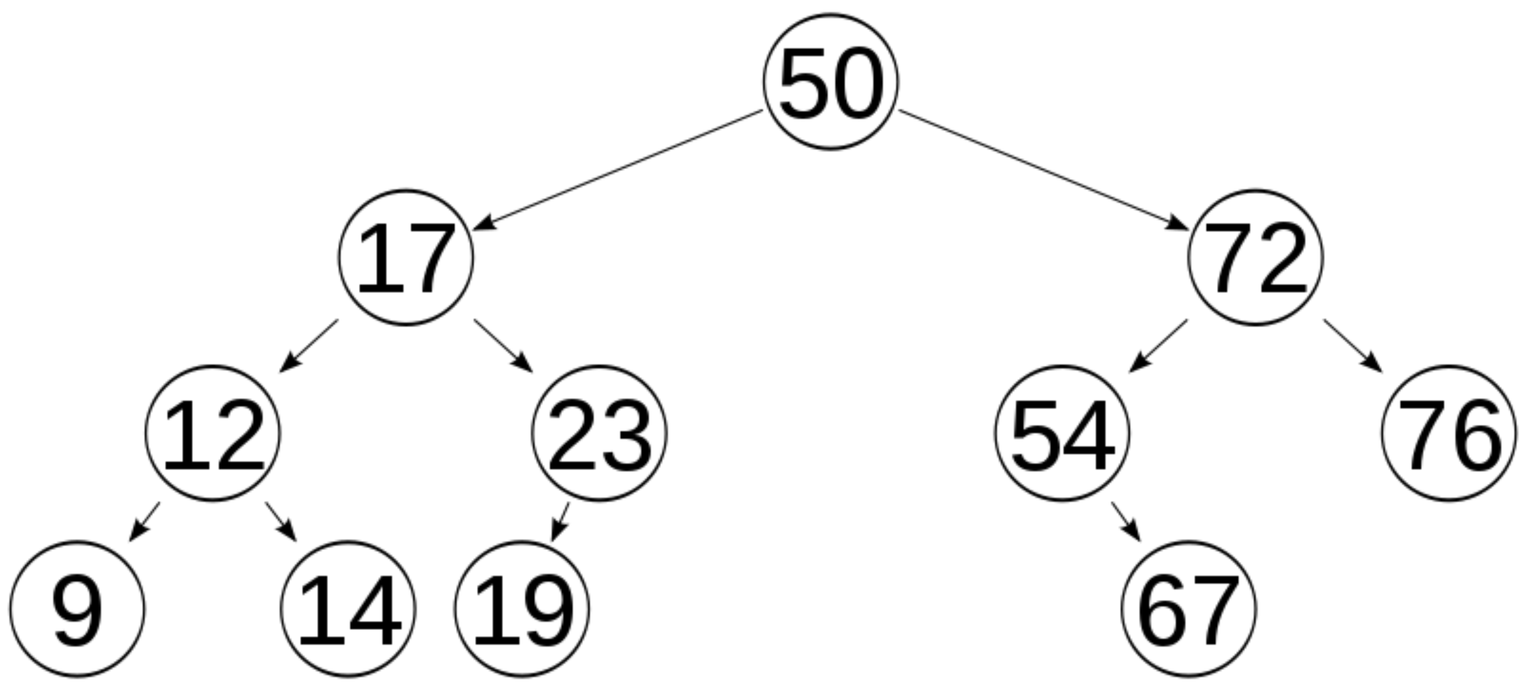
Time complexity of
- Inserting an element into a BST:
- Searching for an element in a BST:
  - Average case:
  - Worst case:



Balanced Binary Search Trees:

Time complexity of worst-case of search in a BST is **O(n)**

How can we keep the BST balanced so that the worst-case is just **O(log n)** because the height of the tree is limited to **O(log n)**?



Balanced Binary Search Trees:

Time complexity of worst-case of search in a BST is **O(n)**

How can we keep the BST balanced so that the worst-case is just **O(log n)** because the height of the tree is limited to **O(log n)**?

1. AVL Trees
2. Red-Black Trees
3. Splay Trees
4. 2-3 Trees
   a. Not exactly a BST.
      It’s not even a Binary Tree.
      It’s a **Balanced Search Tree**.

Transform for good!

</ End of Transform-n-Conquer >