HEAPS AND SORTING

The (binary) heap is a data structure which is represented as an array and can be viewed as a binary tree such that

1. Each node of the tree corresponds to an element of the array and the value is stored in the node

2. The tree is completely filled on all levels, except possibly the lowest, which is filled from the left upto a point

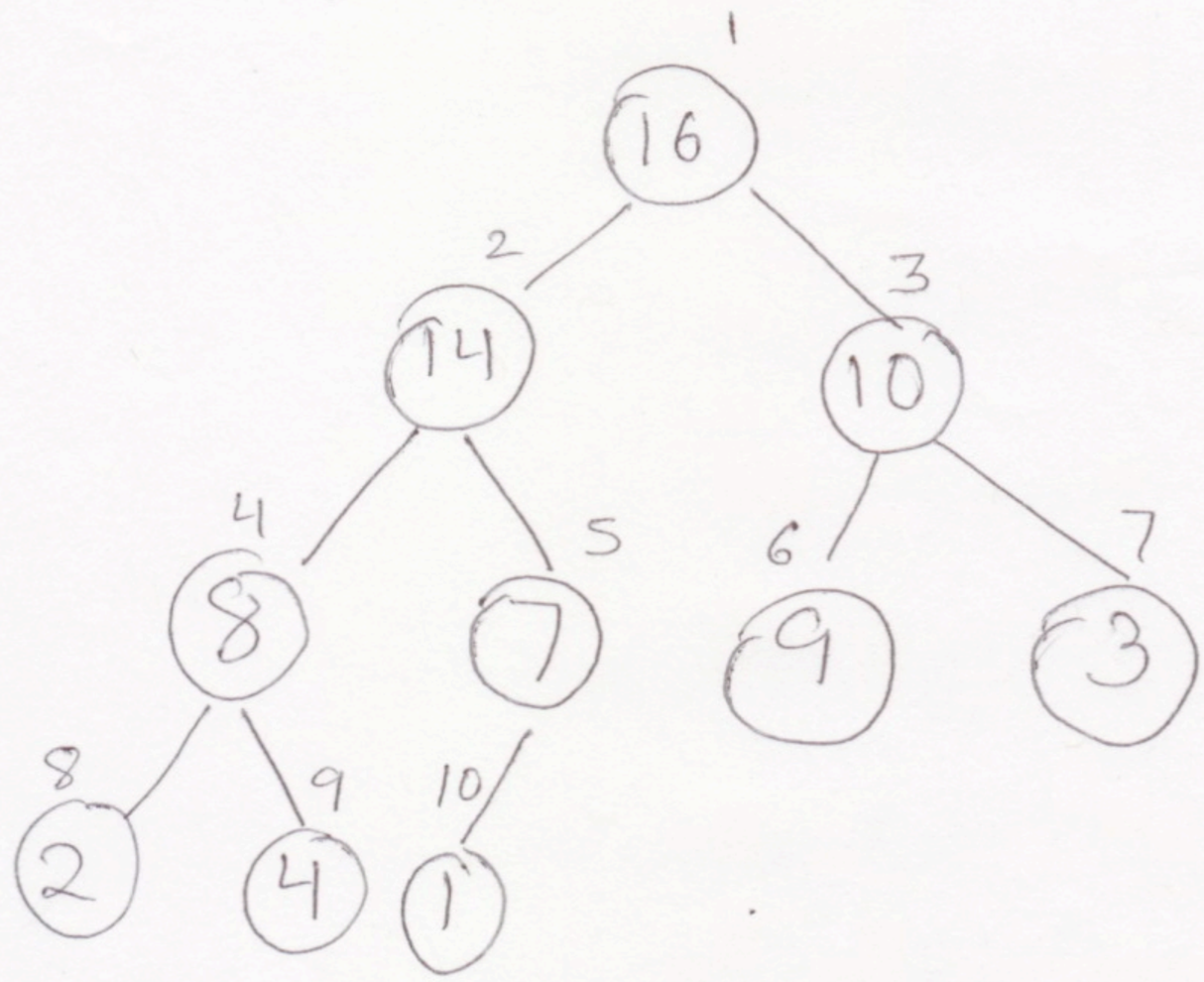
3. The root of the tree is A[1] and given the index i of a node, the indices of its parent and children can be computed as

$$\text{Parent}(i) = \lfloor i/2 \rfloor$$

$$\text{Left}(i) = 2i$$

$$\text{Right}(i) = 2i + 1$$

Example



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

Fig: Binary tree and array representation of a heap

4. The heap-structure satisfies the heap-property

$\forall$ node i, other than the root

$A[Parent[i]] \geq A[i]$

- value at a node is at most that at its parent
- This is called the max-heap (root has the highest value). You can similarly define the min-heap.

Questions

1. What is the minimum and maximum number of elements in a heap of height h.

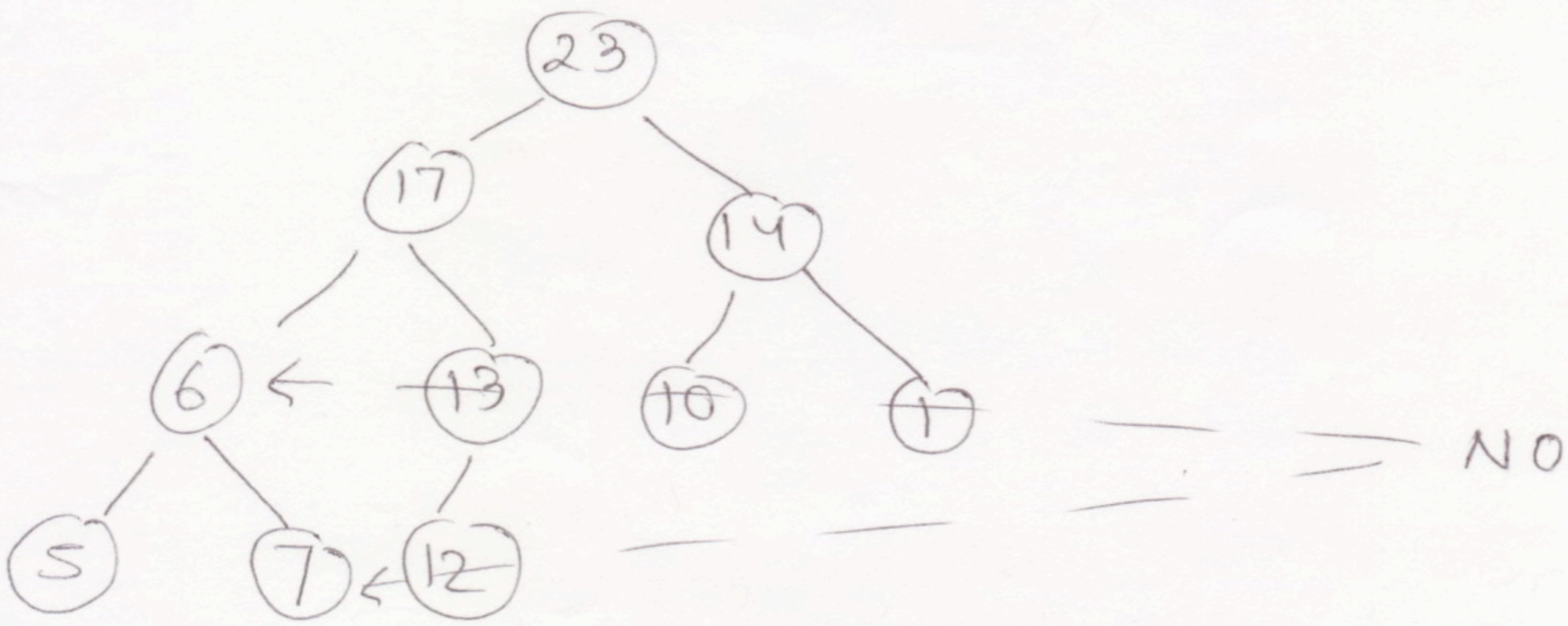
minimum $= 2^h$

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

2. Is the sequence

23, 17, 14, 6, 13, 10, 1, 5, 7, 12

a heap?



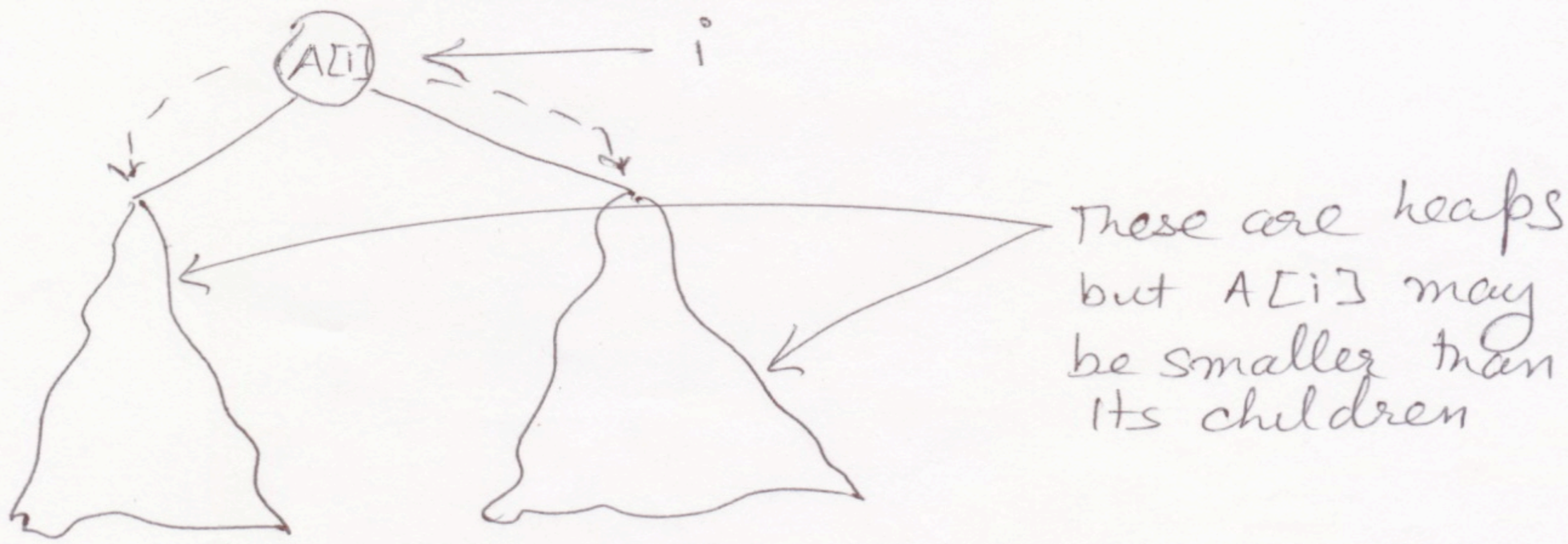
3. Is an array in reverse sorted order a heap?

Yes.

Maintaining the Heap property

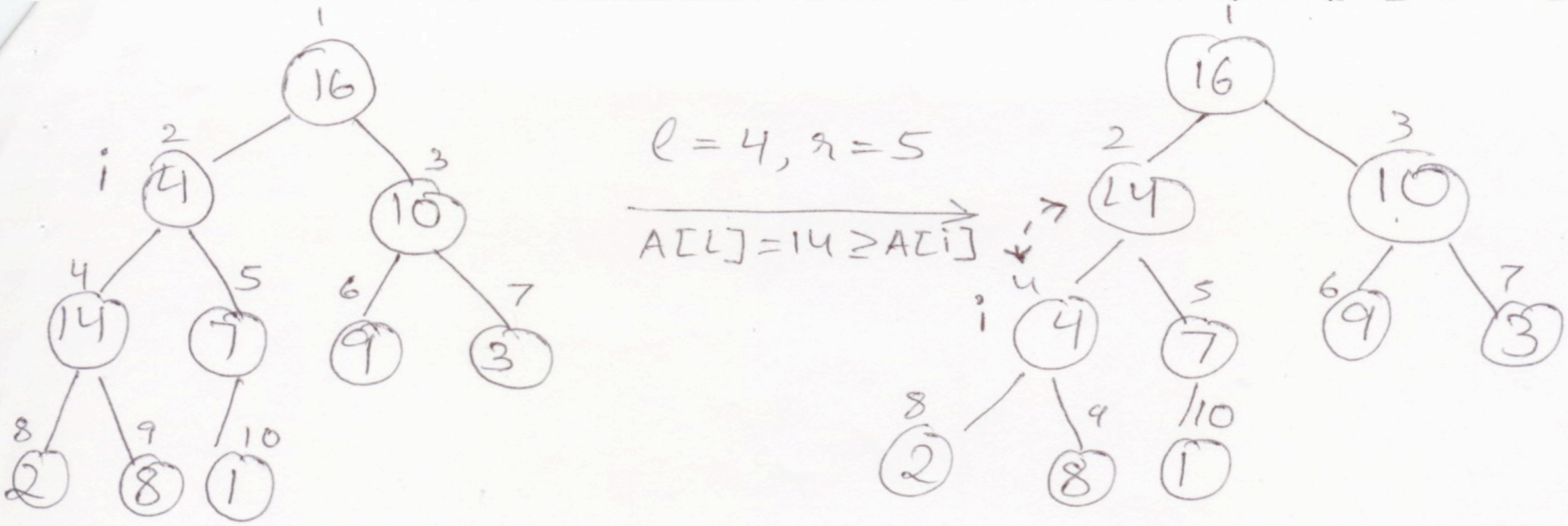
The routine Heapify(·,·) is used to maintain the heap property. Its inputs are

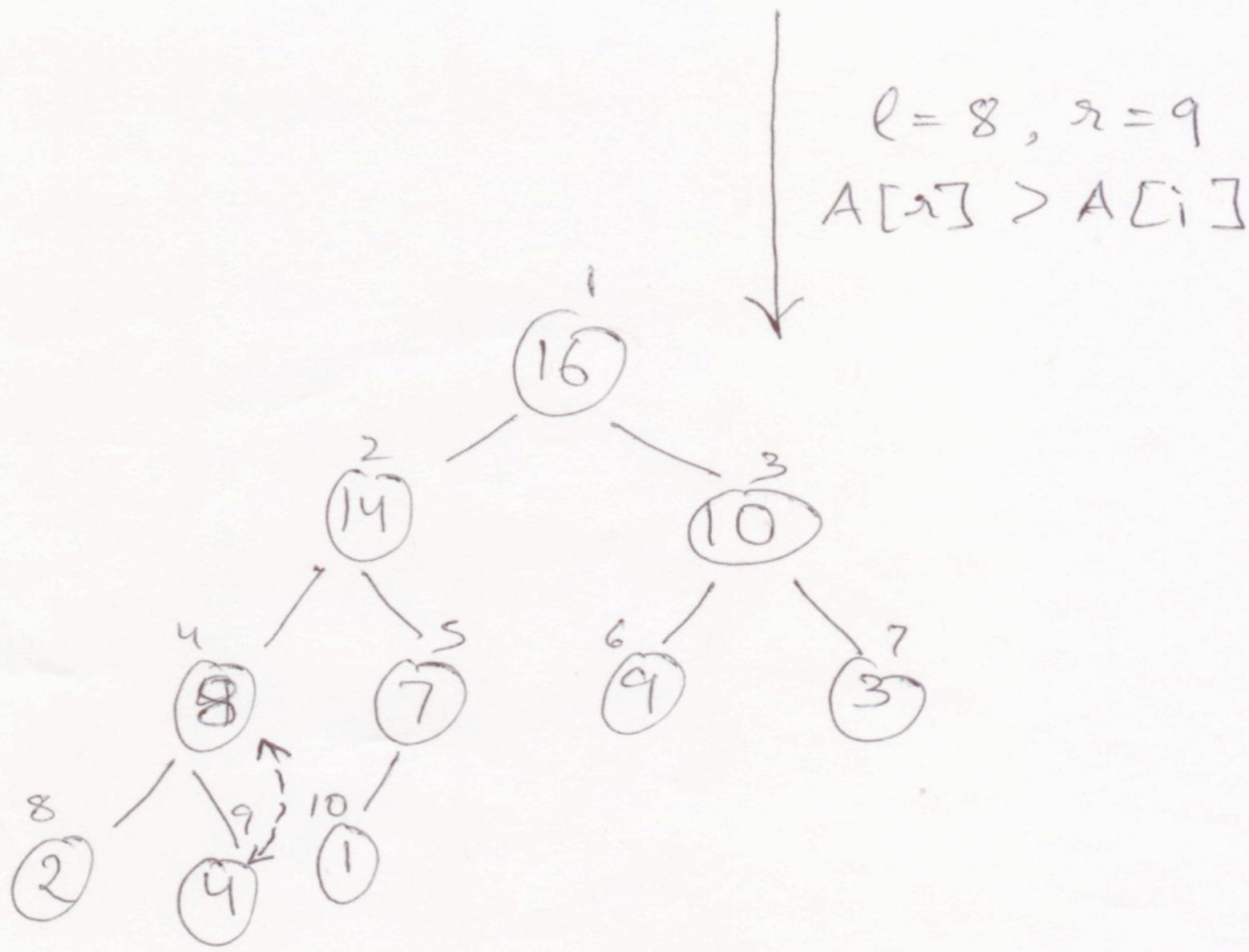
- The array A and an index i
- Heapify assumes:



- idea: Make A[i] "float-down" such that the subtree rooted at i becomes a heap

```
Heapify (A, i)
    l ← left(i)
    r ← right(i)
    if l ≤ heap_size(A) and A(l) > A(i)
        then largest ← l
        else largest ← i
    if r ≤ heap_size(A) and A(r) > A(largest)
        then largest ← r
    if largest ≠ i
        Then exchange A[i] ↔ A[largest]
```





<u>Complexity</u>

Running time of heapify on a subtree of size n rooted at i is

$O(1)$ to fix the relationship amongst elements $A[i]$, $A[l]$, $A[r]$ +

running time of heapify on a subtree rooted at one of the children of i

$= O(1) + O(h)$

$= O(h) = O(\lg n)$

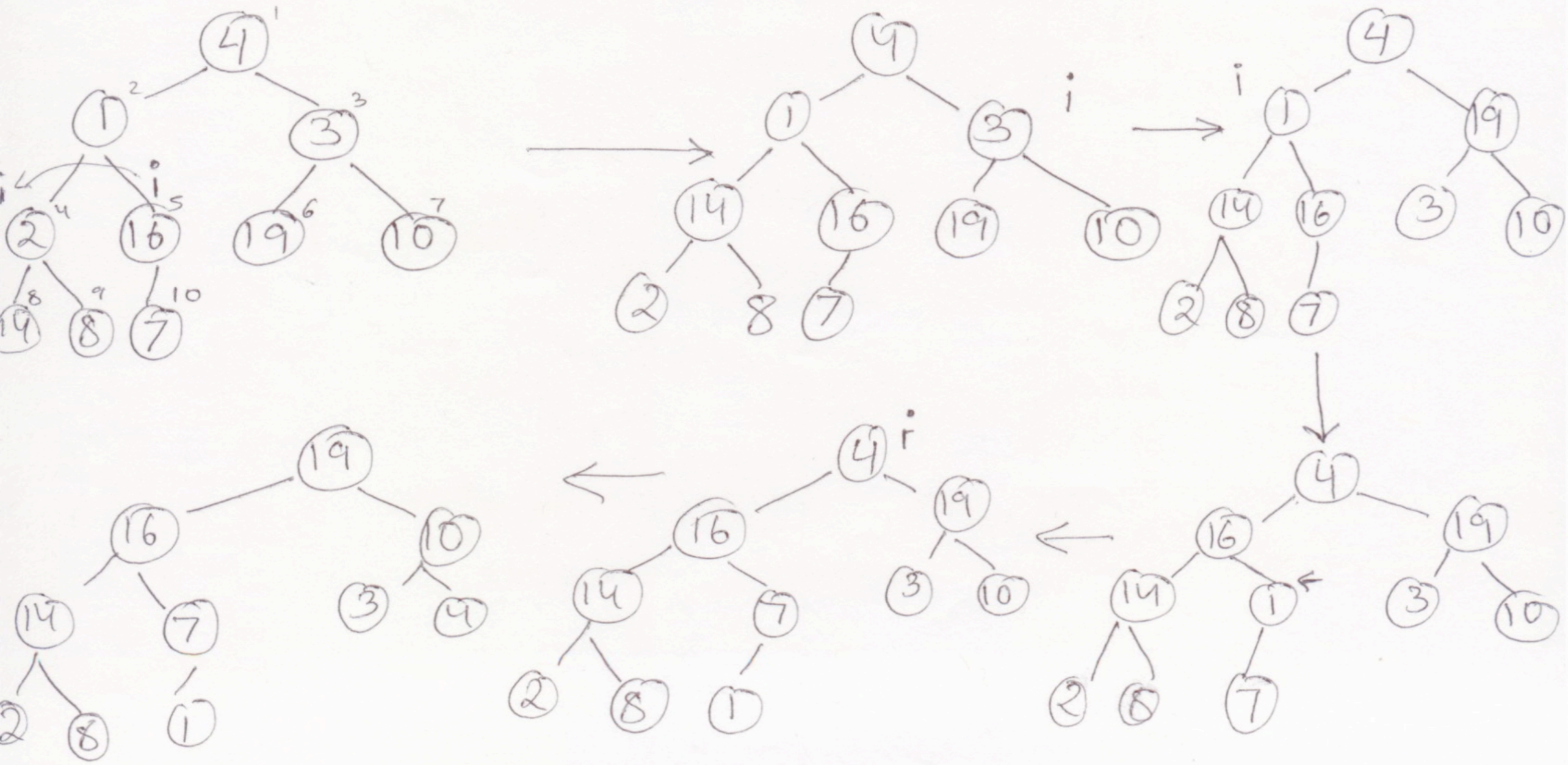
Building a heap

- Heapify can be used from bottom-up to convert an array $A[1, \ldots n]$, into a heap.

  - Note: Elements in the subarray $A[(\lfloor \frac{n}{2} \rfloor + 1), \ldots n]$ are all leaves (and therefore a 1-element heap)

```
Build-Heap (A)
    heap-size(A) ← length(A)
    for i = ⌊length(A)/2⌋ downto 1
        do Heapify (A, i)
```

Example: | 4 | 1 | 3 | 2 | 16 | 19 | 10 | 14 | 8 | 7 |



Exercise : Illustrate the operation of build-heap on

$A = [5, 3, 17, 10, 84, 19, 6, 22, 9]$

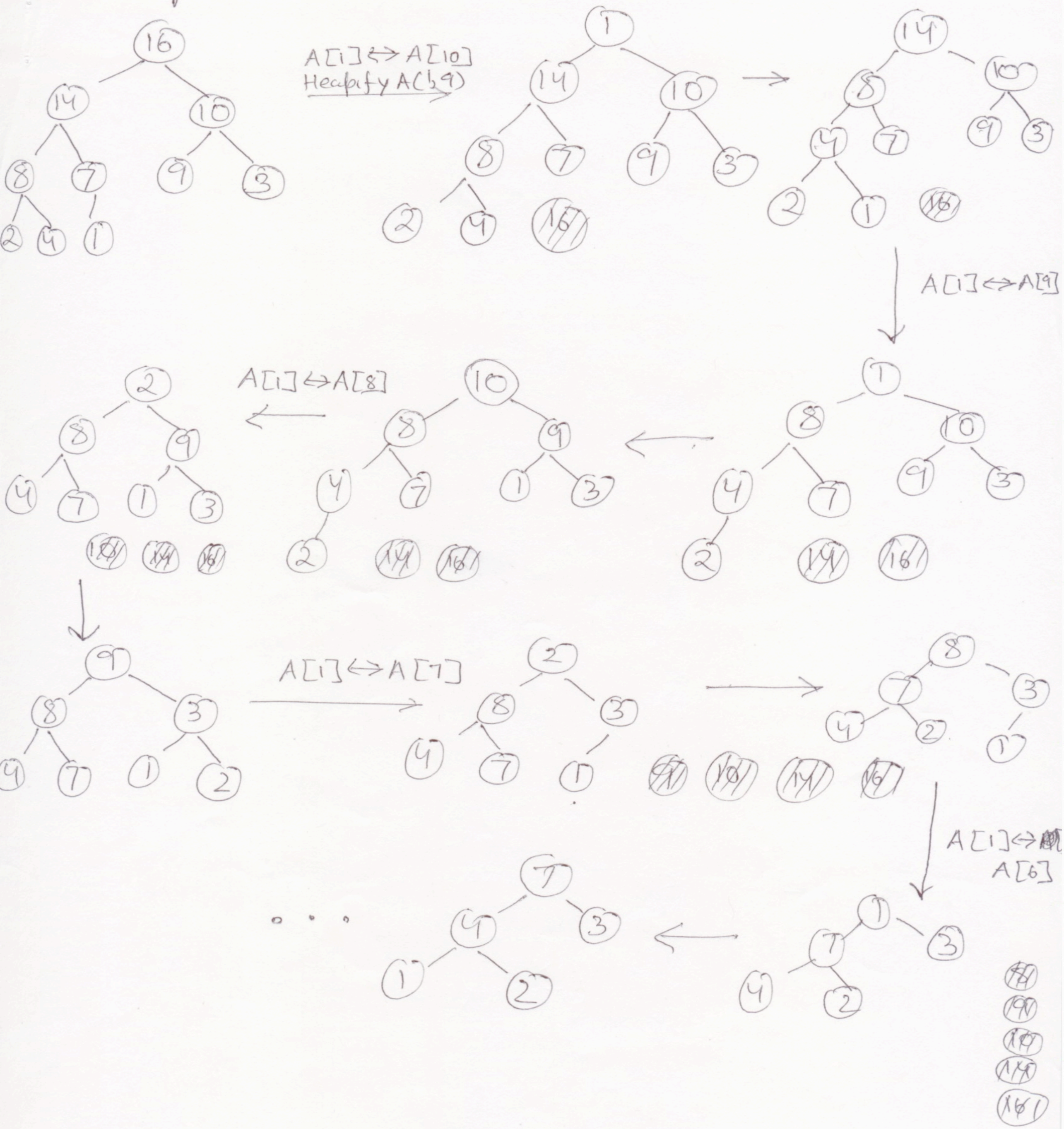
The Heapsort Algorithm

<u>Idea</u>

- Use build-heap to construct a heap on the input array $A[1, \ldots n]$ where $n = length(A)$
- Now the maximum element lies at the root
- Exchang the root with $A[n]$. This puts the largest element in its correct spt (at the end of the array)
- Heapify $A[1, \ldots n-1]$ by calling the routine at $A[1]$ since this is the only place where the heap property is violated.
- Repeat this process down to heap size of 2

```
HeapSort (A)
    Build-Heap (A)
    for i ← length (A) downto 2
        do  exchange A[1] ↔ A[i]
            heap-size[A] ← heap-size[A] - 1
            Heapify(A, 1)
```

Example



Complexity of heapsort

① Complexity of build-heap +
(n-1) calls to Heapify

= Complexity of build-heap + $O((n-1)\lg n)$
+ $O(n \lg n)$