HEAP and HEAPSORT

- Like Merge sort, Heapsort has running time $= O(n \lg n)$.
- Like Insertion sort, Heapsort is in-place sorting algorithm.
  - Only a $O(1)$ number of array elements are sorted outside the input array at any time.
- Heapsort introduces another algorithm-design technique: using a data structure - "HEAP" to manage information. HEAP data structure is also useful making an efficient priority queue.

There exist 3-ary or n-ary Heaps also.

» **Heaps:** The (binary) heap data structure is an array object that we can view as a nearly complete binary tree. Each node of the tree corresponds to an element of the array. An array A that represents a heap is an object with two attributes: A.length, and A.heap-size (i.e., how many elements in the heap are stored within array A). That is, although A[1 : A.length] may contain numbers, only the elements in A[1 : A.heapsize], where $0 \leq$ A.heap-size $\leq$ A.length, are valid elements of the heap.

- Min #elements in a heap of height $h = 2^h$
- Max #elements in a heap of height $h = 2^{h+1} - 1$.

Where, $h = \lfloor \lg n \rfloor$, ($n$ = #elements in the heap).

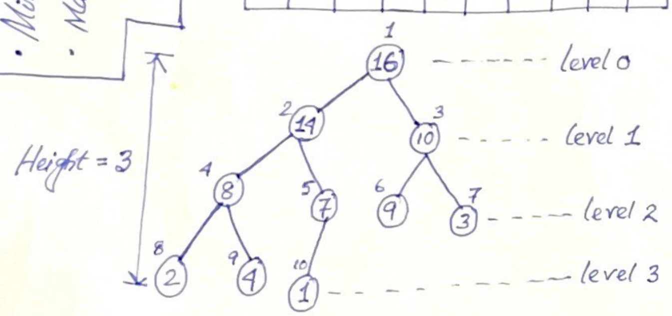
The root of the tree is A[1], and given the index i of a node, we can easily compute the indices of its parent, left child, and right child.

/* Indexing starts from 1 */ $\lfloor i/2 \rfloor$ $\quad 2i \quad (2i+1)$

» An array A represented as a max-heap (binary tree) below:

$i \rightarrow$	1	2	3	4	5	6	7	8	9	10
A	16	14	10	8	7	9	3	2	4	1

For every node i other than the root,
A[PARENT(i)] $\geq$ A[i].
(i.e., the largest element in a max-heap is stored at the root.)



- A complete binary tree.

/* In most computers, $2i$ can be computed as by left shifting binary representation of i, by 1 bit. And $\lfloor i/2 \rfloor$ can be computed as by right shifting binary representation of i, by 1-bit. */

NB). - For HEAP-SORT, algorithm, we usually use Max-Heaps.

- Min-Heaps commonly implement PRIORITY QUEUES.
- Height (HEAP) = Height (Root). Since a heap of n element is based on a complete binary tree, its height is $= \Theta(\lg n)$. We will see that the basic operations on heaps run in time at most ∞ height of the tree and thus take $O(\lg n)$ time.

✓ Smallest element in a Max-Heap must be a leaf node.

When left subtree is a max heap, right subtree is a max heap and we - want to make the root a max heap.

- Maintaining the Heap property: MAX-HEAPIFY (A,i) maintains the max-heap property.

Not Top-Down approach i.e., Bottom up in case of building heap using BUILD MAX-HEAPIFY

MAX-HEAPIFY (A,i)

```
l = LEFT(i)
r = RIGHT(i)
if l ≤ A.heapsize and A[l] > A[i]
    largest = l
else largest = i
if r ≤ A.heapsize and A[r] > A[largest]
    largest = r
if largest ≠ i
    exchange (A[i], A[largest])
    MAX-HEAPIFY (A, largest).
```

PARENT(i)

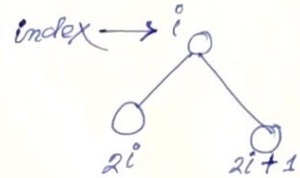
return $\lfloor i/2 \rfloor$

LEFT(i)

return $2i$

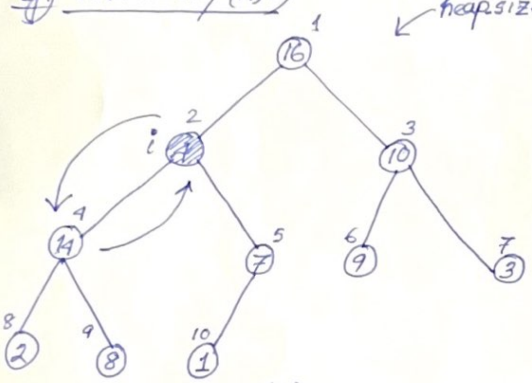
RIGHT(i)

return $(2i+1)$.



eg. MAX-HEAPIFY (A, 2)

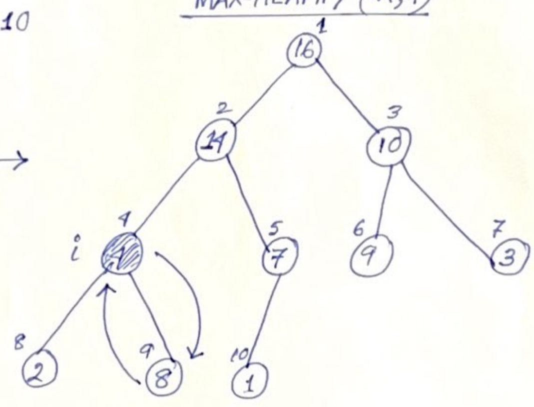
heapsize = 10



(a)

$\longrightarrow$

MAX-HEAPIFY (A, 4)



(b)

Leaf nodes,

$\lfloor n/2 \rfloor + 1$ to n.

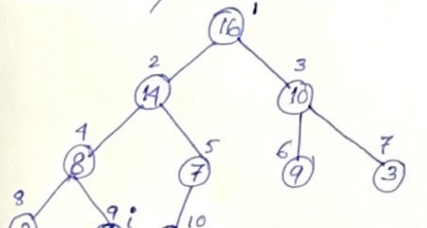
- Space Complexity of MAX-HEAPIFY

= Aux. Space + # Recursive calls. (i.e # levels)

$= O(1) + O(\lg n)$

$= O(\lg n)$.

MAX-HEAPIFY (A, 9)



(c)

/* Running time of MAX-HEAPIFY on a subtree of size n rooted at a given node i is;

$= \Theta(1) + [$time to run MAX-HEAPIFY on a subtree rooted at one of the children of node i (if recursive call occurs)$]$.

Time to fixup the relationship among the elements $A[i]$, $A[LEFT(i)]$, and $A[RIGHT(i)]$.

- The children's subtrees each have size at most $\frac{2n}{3}$ — the worst case occurs when the bottom level of the tree is exactly half full. So, $\boxed{T(n) \leq T(2n/3) + \Theta(1)}$

$\therefore \boxed{T(n) = \Theta(\lg n)}$

nodes in the subtree for which i is the root.

Where, $(\lg n)$ can be written in terms of height of a trees (heaps) node.

$\boxed{T(h) = O(h)}$

Building a Heap: (Bottom-up) /* Using MAX-HEAPIFY to build a heap. */

- We can use the procedure MAX-HEAPIFY in a bottom-up manner to convert an array $A[1:n]$, where $n = A.length$, into a max-heap. The elements in the subarray $A[(\lfloor n/2 \rfloor + 1) : n]$ are all leaves of the tree, and so each is a 1-element heap to begin with. The procedure BUILD-MAX-HEAP goes through the remaining nodes of the tree and runs MAX-HEAPIFY on each one.

HEAPIFY

BUILD-MAX-HEAPIFY (A) /* Used to build max-heap or min heap */.

1. A.heapsize = A.length
2. for $i = \lfloor A.length/2 \rfloor$ downto 1
3. MAX-HEAPIFY (A, i)

/* Loop Invariant: At the start of each iteration of the for loop of lines 2-3, each node $(i+1), (i+2), \ldots, n$ is the root of a max-heap. */

Whenever MAX-HEAPIFY is called on a node, the two subtrees of that node are both max-heaps.

(i) Initialization: Prior to the first iteration of the loop, $i = \lfloor n/2 \rfloor$. Each node $(\lfloor n/2 \rfloor + 1), (\lfloor n/2 \rfloor + 2), \ldots, n$ is a leaf node, and is thus the root of a trivial max-heap.

(ii) Maintenance: To see that each iteration maintains the loop invariant, observe that the Children of node i are numbered higher than i. By the loop invariant, therefore, they are both roots of max-heap. This is precisely the condition required for the call MAX-HEAPIFY (A, i) to make node i a max-heap root. Moreover, the MAX-HEAPIFY call preserves the property that nodes $(i+1), (i+2), \ldots, n$ are all roots of max-heaps. Decrementing i in the for loop update reestablishes the loop invariant for the next iteration.

(iii) Termination: At termination, $i=0$. By the loop invariant, each node $1, 2, \ldots, n$ is the root of a max-heap. In particular, the node 1.

- Running Time: • Each call to MAX-HEAPIFY costs $O(\lg n)$ time, and BUILD-MAX-HEAP makes $O(n)$ such calls. So, running time $= O(n \lg n)$.

  . We can derive a tighter bound by observing that the time for MAX-HEAPIFY to run at a node varies with the height of the node in the tree, and the heights of most nodes are small. Our tighter analysis relies on the properties that an n-element heap has height $\lfloor \lg n \rfloor$ and atmost $\lceil n/2^{h+1} \rceil$ nodes of any height h. The time required by MAX-HEAPIFY when called on a node of height h is $O(h)$, and so we can express the total cost of BUILD-MAX-HEAP as being bounded from above by,

$$\underbrace{\sum_{h=0}^{\lfloor \lg n \rfloor}}_{\text{Range of height.}} \underbrace{\left\lceil \frac{n}{2^{h+1}} \right\rceil}_{\text{Max. no. of nodes at height 'h'.}} O(h) = O\left(n \sum_{h=0}^{\lfloor \lg n \rfloor} \frac{h}{2^h}\right)$$

$$= O\left(n \sum_{h=0}^{\infty} \frac{h}{2^h}\right)$$

$$= O\left(n \sum_{h=0}^{\infty} h \left(\frac{1}{2}\right)^h\right)$$

$$= O\left(n \cdot \frac{1/2}{(1-1/2)^2}\right)$$

$$= O(n.2) = O(n).$$

$$\left[\sum_{k=0}^{\infty} k x^k = \frac{x}{(1-x)^2}, \quad \text{for } |x|<1 \right.$$

If we apply MAX-HEAPIFY on any of the max. no. of nodes the work done is $= O(h)$ ← differs with the height.
(More height = More work done.)

Hence, we can build a max-heap from an unordered array in linear time.

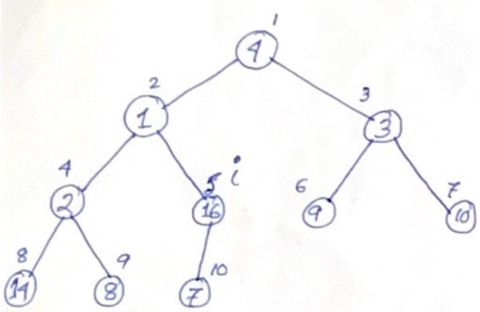
———o———

Ex.) Build MAX-HEAP:

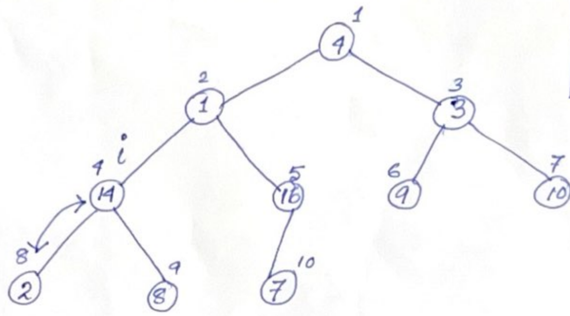
A

4	1	3	2	16	9	10	14	8	7

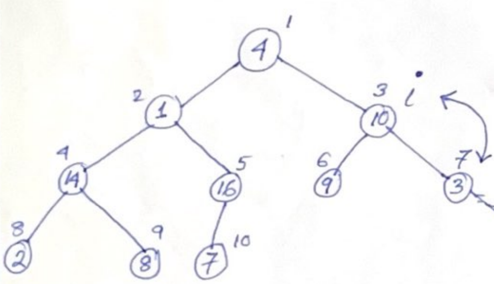
• Space Complexity of BUILD-MAX-HEAP $= O(\log n)$



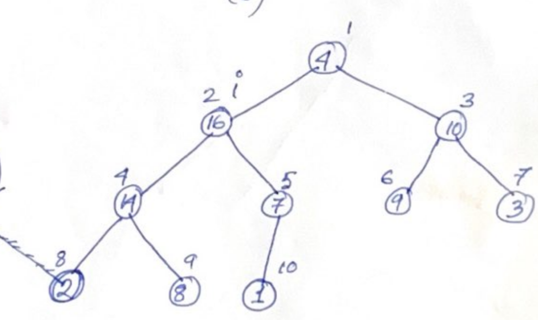
(a)



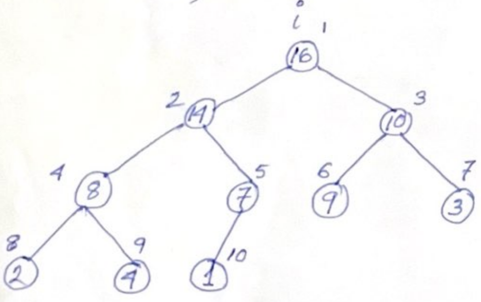
(b)



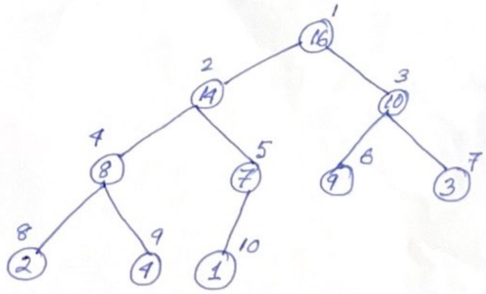
(c)



(d)



(e)



(f) Final Max-Heap.

———o———

Heapsort: The heapsort algorithm starts by using BUILD-MAX-HEAP to build a max-heap on the input array $A[1:n]$, where $n = A.length$.

HEAPSORT (A)

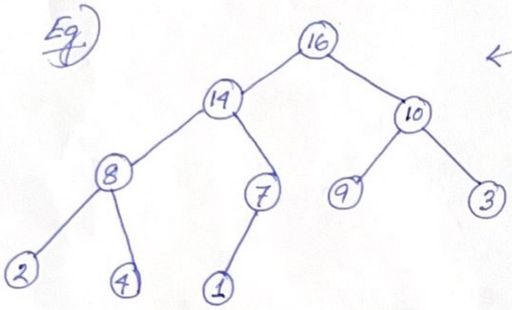
1. BUILD-MAX-HEAP (A)
2. For $i = A.length$ downto 2
3. exchange $(A[1], A[i])$
4. $A.heapsize = A.heapsize - 1$
5. MAX-HEAPIFY $(A, 1)$.

/* **Time Complexity of HEAPSORT:**

$= O(n \lg n)$

Call to Build heap (BUILD-MAX-HEAP) takes time $O(n)$ and each of the $(n-1)$ calls to MAX-HEAPIFY takes time $O(\lg n)$.

Eg.



← Run HEAPSORT, finally sorted array A is,

A										
	1	2	3	4	7	8	9	10	14	16

—o—

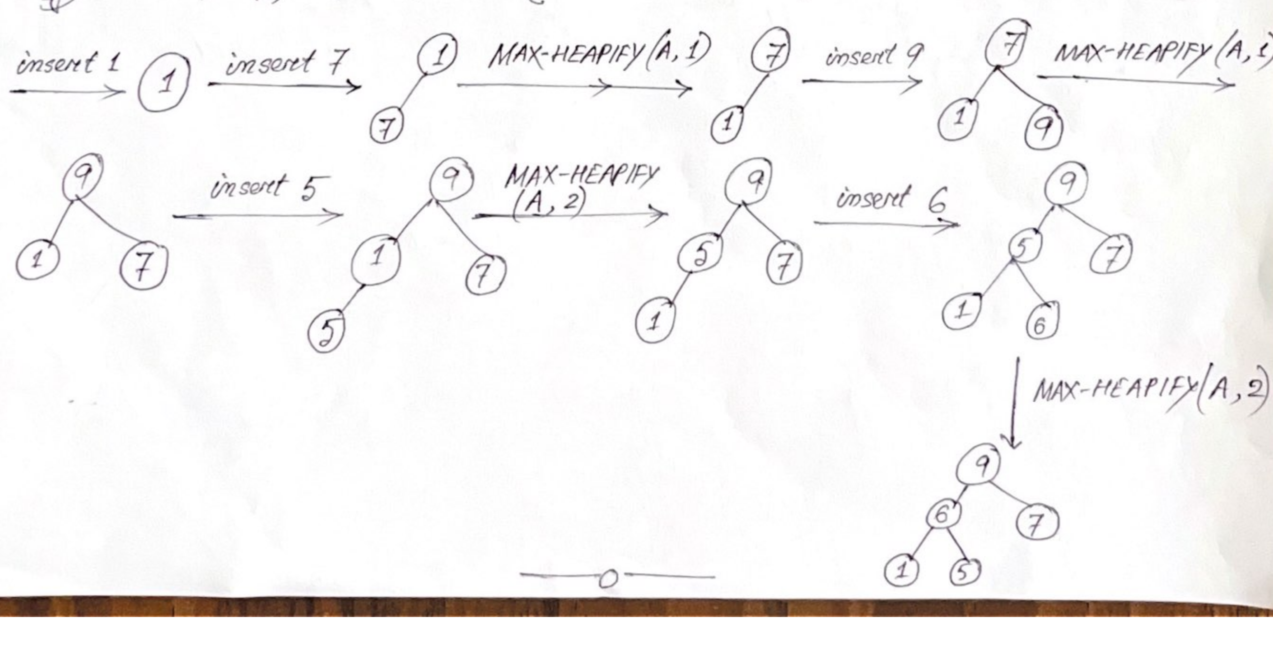
Insertion and Deletion in a Heap: /* Time taken for insertion, deletion = $\boxed{O(\lg n)}$ */.

→ Always insert at the last position, apply HEAPIFY if needed.

Always delete from Root.

→ Swap the Root with the last element, then delete the last node, apply MAX-HEAPIFY at Root and so on.

eg. 1, 7, 9, 5, 6. (Construct heap by inserting keys in Top-Down)



—o—

Iterative_MAX-HEAPIFY (A, i)

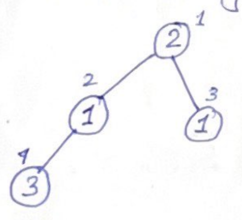
```
While (i < A.heapsize)
    l = 2i
    r = 2i + 1
    largest = i
    if (l ≤ A.heapsize and A[l] > A[i])
        largest = l
    if (r ≤ A.heapsize and A[r] > A[i])
        largest = r
    if (largest ≠ i)
        exchange (A[i], A[largest])
Return A
```

/* The code for MAX-HEAPIFY (Recursive) is quite efficient in terms of constant factors, except possibly for the recursive call, which might cause some compilers to produce inefficient code. Therefore, an iterative version is given here. */

Questions: (i) Why do we want the loop index i in line 2 of BUILD-MAX-HEAP to decrease from $\lfloor A.length/2 \rfloor$ to 1, rather than increase from 1 to $\lfloor A.length/2 \rfloor$?

Ans: If we had started at 1, we wouldn't be able to guarantee that the max-heap property is maintained. For example,

	1	2	3	4
A	2	1	1	3



- Call of MAX-HEAPIFY (A, 1) i.e, no exchange of elements.
- But call of MAX-HEAPIFY (A, 2) will swap (A[2], A[4]). So, now, max-heapify property at root node is violated.