12

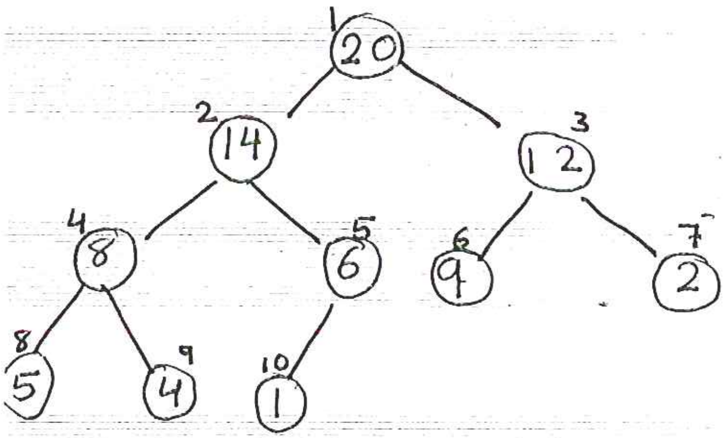
Heaps: Max-Heap, Min-Heap
Building a Heap

COMP 2321
sections : 1 [redacted]
Dr. Majdi M. Mafarja

COMP242

Heap: nearly a complete binary tree

tree is completely filled except at the lowest level, which is filled from left to right.



$n = 10$

$parent(i) = \lfloor i/2 \rfloor$

$left(i) = 2i$

$right(i) = 2i+1$

depending on the complete binary tree property, we add in level 0 first, then to level 1 untill full and so on... From left to right

$$\Rightarrow \begin{array}{cccccccccc} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 \\ 20, & 14, & 12, & 8, & 6, & 9, & 2, & 5, & 5, & 1 \end{array}$$

l

r

From this array we can observe that

$parent(i) = \lfloor i/2 \rfloor$

$left(i) = 2i$

$right(i) = 2i+1$

Max-Heap & Min-Heap

Max-Heap property

maximum element in the Max-Heap is the ROOT

$A[parent(i)] \geq A[i]$

The parent must be greater then (or equals to) its children.

Min-Heap property

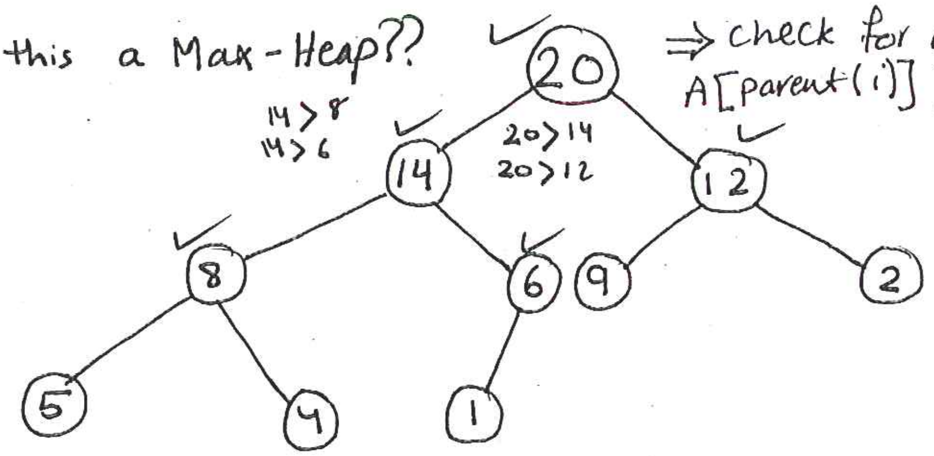
minimum element in the Min-Heap is the ROOT

$A[parent(i)] \leq A[i]$

is this a Max-Heap??

⇒ check for all nodes $A[parent(i)] \geq A[i]$

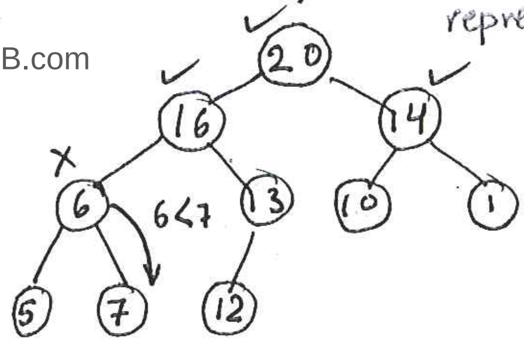
⇒



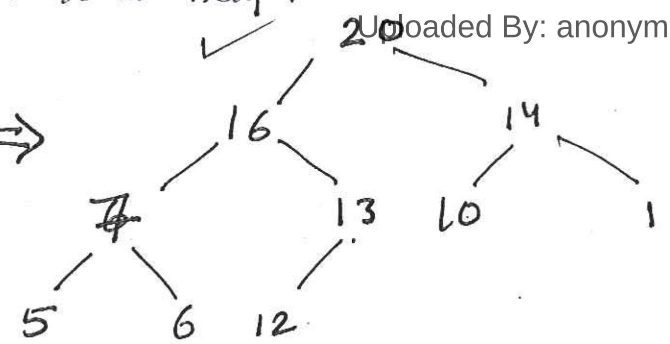
* * *

Example.

is the sequence A[20, 16, 14, 6, 13, 10, 1, 5, 7, 12] represent a Max-Heap?



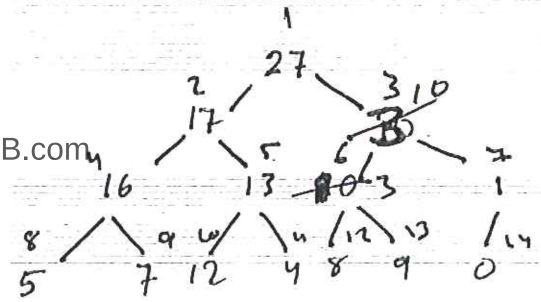
⇒



Maitaining the Heap property.

```
max-heapify (A, i)
{
    l = left(i);   r = right(i);
    if (l ≤ heap-size(A) && A[l] > A[i])
            max = l;
    else
         max = i;
    if (r ≤ heap-size(A) && A[r] > A[max])
          max = r;
    if (max != i) {
         swap(A[i], A[max]);
         max-heapify (A, max);
    }
}
```

Example max-heapify (A, 3) on A = [27, 17, 3, 16, 13, 10, 1, 5, 7, 12, 4, 8, 9, 0]



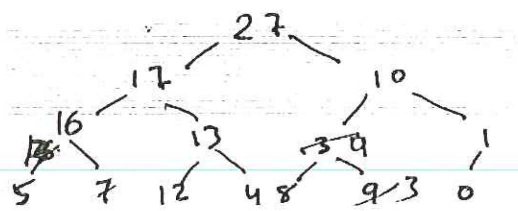
max-heapify (A, 3)

l = 6 , r = 7

A[l] $\overset{?}{>}$ A[i] ? ✓ max = l

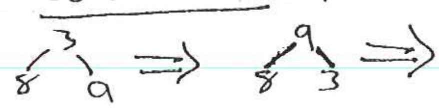
A[r] $\overset{?}{>}$ A[max] ? X

max ≠ i ⇒ swap 3 → 10 ⇒



check again max-heapify (A, max)

max now is 6 b/s it (l) was the max



⇒ 13 is a leaf stop ✓

Buildind A heap:- $\Longrightarrow O(n)$

We can use the Max_heapify in bottom up manner; to convert an array $A[1 \cdots n]$ to Max_heap

Observations:-

The elements $A(\lfloor \frac{n}{2} \rfloor + 1), A(\lfloor \frac{n}{2} \rfloor + 2), A(\lfloor \frac{n}{2} \rfloor + 3), \cdots A(n)$ are all leaves.

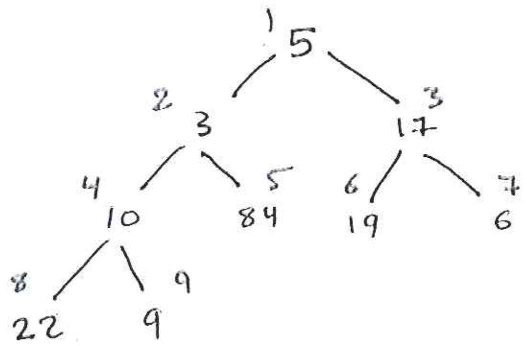
Reson:- n = length of heap = heap_size(A)

any index after $\lfloor \frac{n}{2} \rfloor$ will have left(i) and right(i) > n (does not come in the heap size) so, all are leaves

Example:

$$A = \{ \overset{1}{5}, \overset{2}{3}, \overset{3}{17}, \overset{4}{10}, \overset{5}{84}, \overset{6}{19}, \overset{7}{16}, \overset{8}{22}, \overset{9}{9} \}$$

left(i) = 2i
right(i) = 2i+1
parent(i) = $\lfloor \frac{i}{2} \rfloor$



Build_max_heap(A)

```
for( i = ⌊size(A)/2⌋ ; i >= 1; i--)
    max-heapify(A, i)
```

Running Time :-

Simple upper bound :-

→ Max-heapify costs $O(\log n)$ time

→ There are $O(n)$ such calls

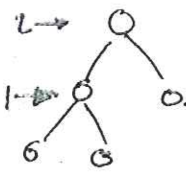
$\Rightarrow O(n \log n)$

However we may produce a tighter bound :-

property one :-

Running time of Max-heapify depends upon the height of the node $O(h)$, and heights of most nodes are small

property Two :-



At most $\lceil n/2^{h+1} \rceil$ nodes are there at any level h

0. bottom up

$5/2^2 = \lfloor 5/4 \rfloor = 1$

$\lfloor \log n \rfloor \Rightarrow$ maximum height of a tree.

max heapify

$$\sum_{h=0}^{\lfloor \log n \rfloor} \left\lceil \frac{n}{2^{h+1}} \right\rceil O(h) = O\left(n \sum_{h=0}^{\lfloor \log n \rfloor} \frac{h}{2^h}\right)$$

$$* \sum_{h=0}^{\infty} \frac{h}{2^h} = 2 \qquad s = 0 + \frac{1}{2^1} + \frac{2}{2^2} + \frac{3}{2^3} + \frac{4}{2^4} + \dots = 2$$

$$\Rightarrow O\left(n \sum_{h=0}^{\lfloor \log n \rfloor} \frac{h}{2^h}\right) < O\left(n \sum_{h=0}^{\infty} \frac{h}{2^h}\right) = O(n)$$

$$< O(2n) = O(n)$$

heape sort

Sorting strategy :-

1- Build Max heap from an unordered array.

2. find the maximum element A[1]

3- Swap elements A[n] and A[1]
   ⇒ now the max element is at the end of an array.

4. Discard node n from heap
   (by decrementing heap size variable)

5. New root may violate max heap property, but its children are max heaps. Run max_heapify to fix this.

6. Go to step 2 unless heap is empty.

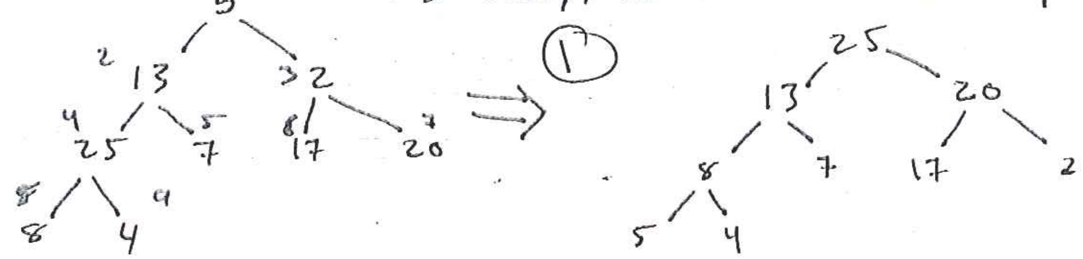
---

heape_sort(A)

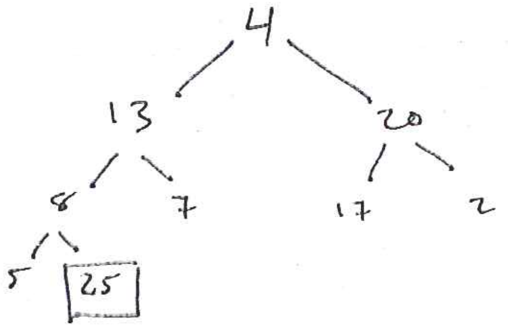
```
Build_max_heap(A);
for(i = size(A); i >= 2 ; i--)
{   swap(A(1), A(i));
    heap_size(A) = heap_size(A)-1 ;
    Max_heapify(A,1);
}
```

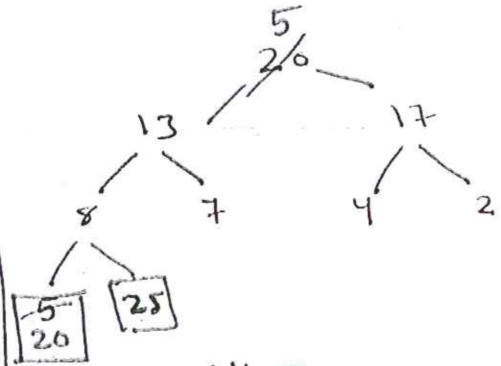
Example: A = {5, 13, 2, 25, 7, 17, 20, 8, 4}

this is a heap, convert it to max-heap



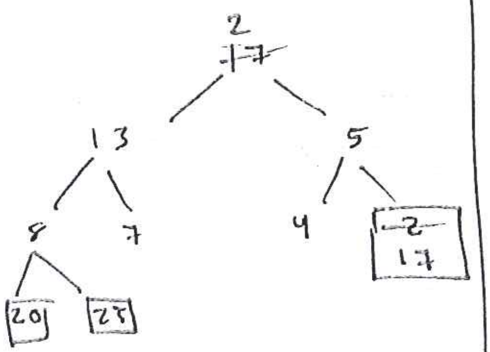
(2) swap 25 with 4



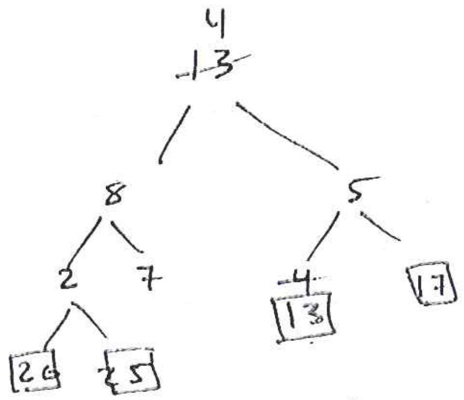


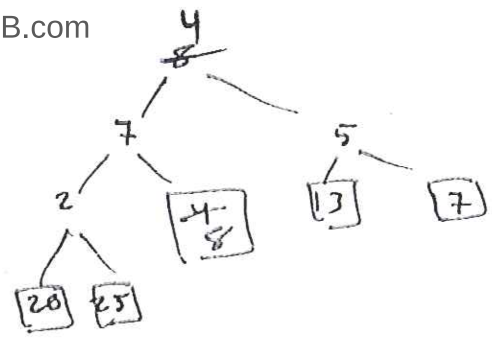
swap 20 with 5

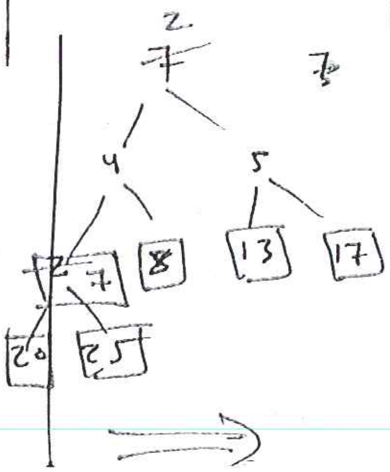
3- discard 25 from the heap.
25 is not apart heap now

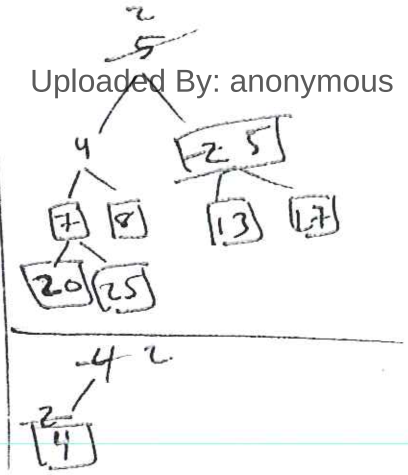


swap 17 and 2









25, 20, 17, 13, 8, 7, 5, 4, 2

{2, 4, 5, 7, 8, 13, 17, 20, 25}

(*) we have n elements, for each of them we call max-heapify $\log n$

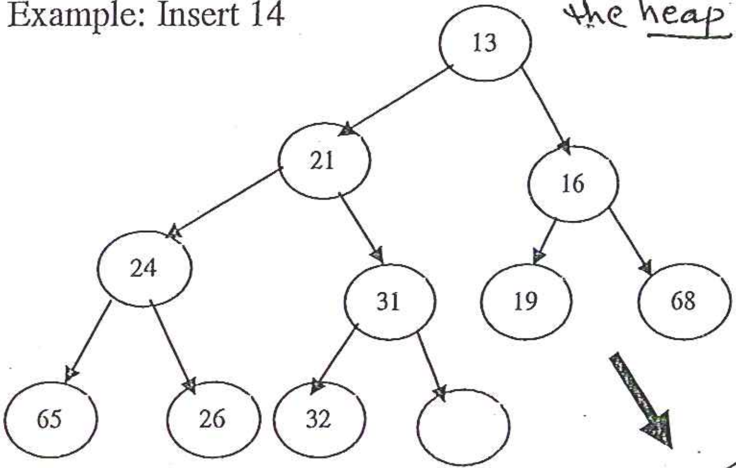
$\Rightarrow O(n \log n)$ time.

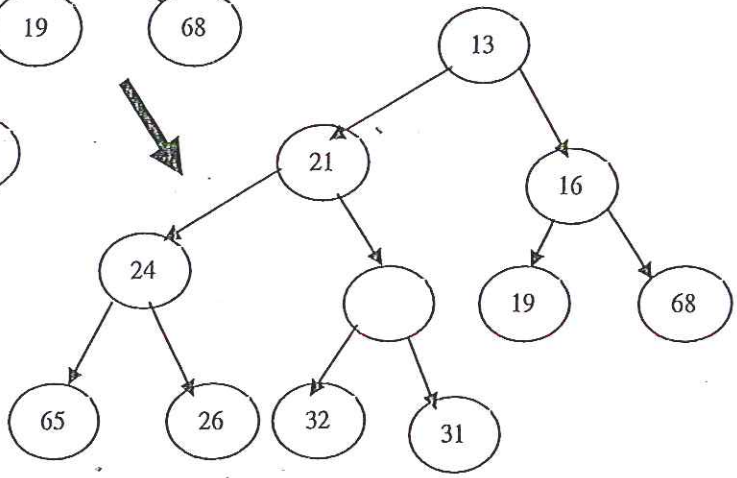
Basic Heap Operation

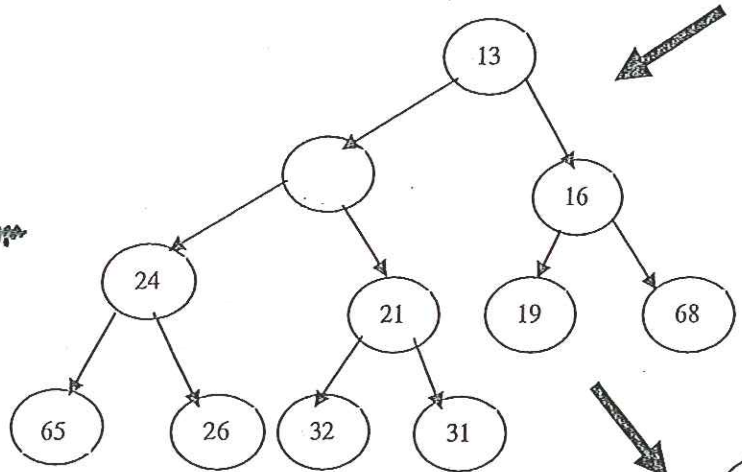
→ Insert

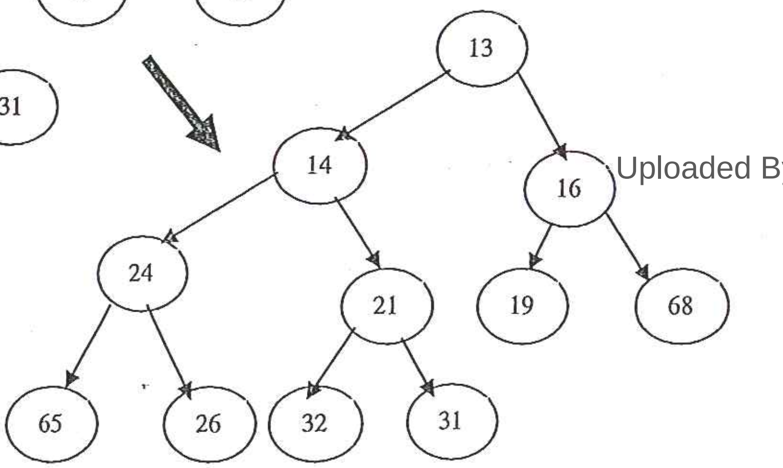
Example: Insert 14

any element can be inserted from the bottom level ~~from~~ starting from the left, then you have to check the heap property.





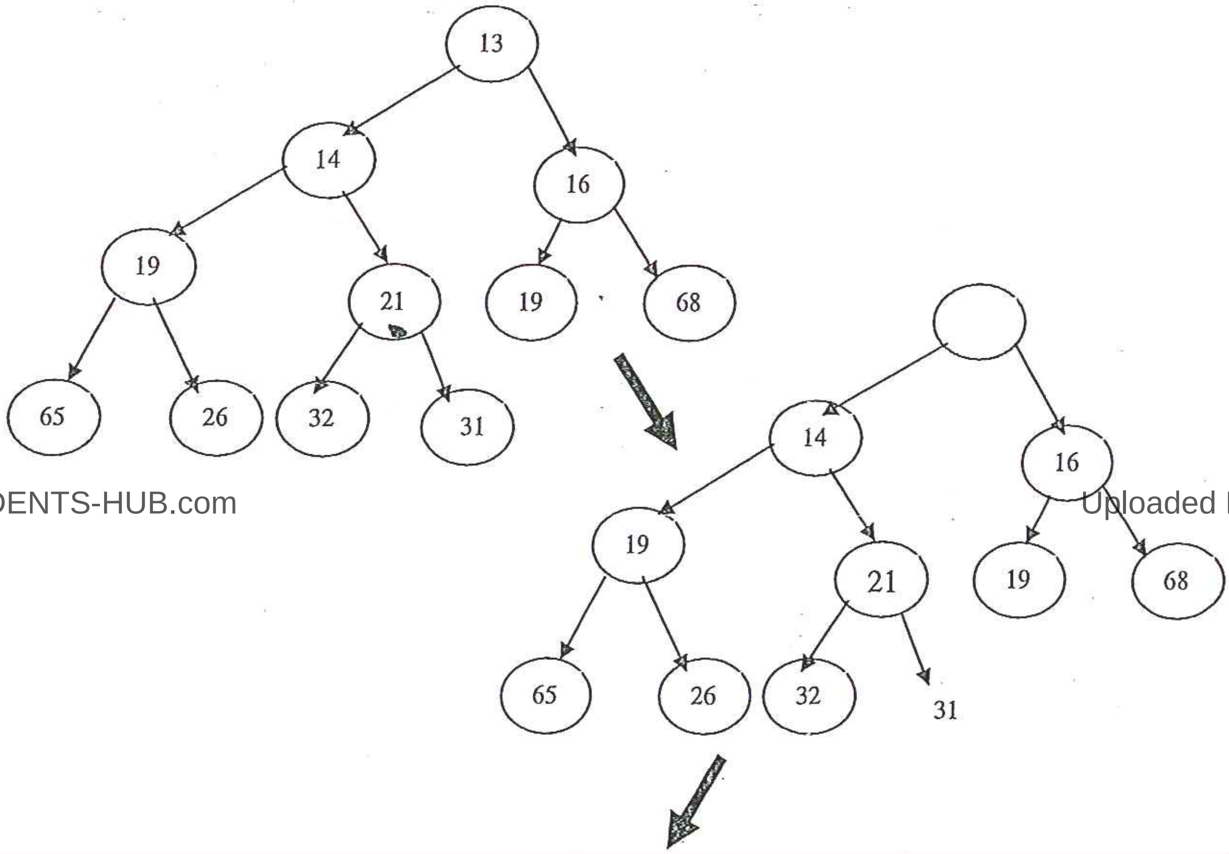


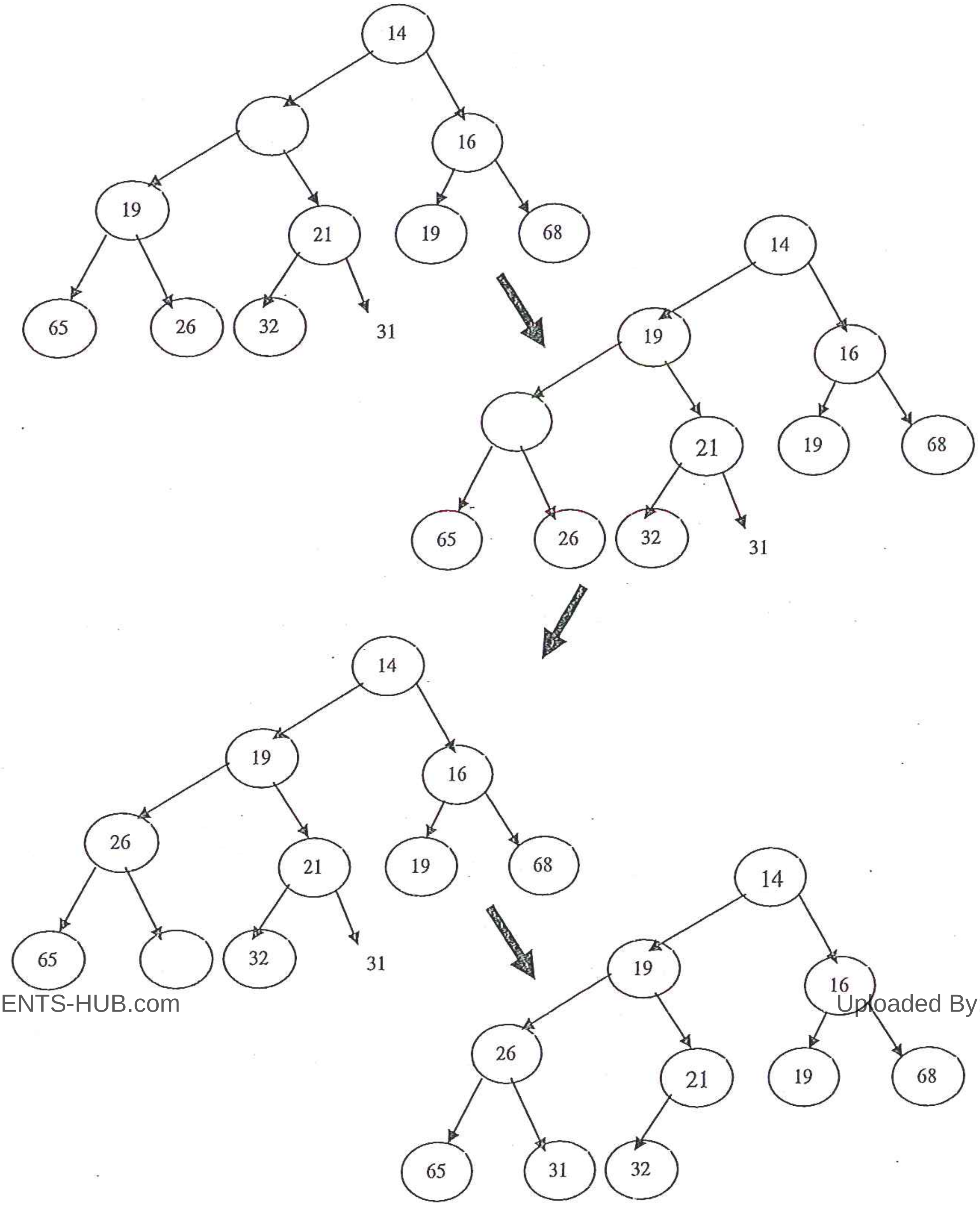


```
void insert ( element_type x, PRIORITY_QUEUE   H)
{
        unsigned int i;
        if ( is_full (H))
                cout << "Priority Queue is full" << endl;
        else
        {
                i = ++H→size;
                while ( H→element[i/2] > x )
                {
                        H→element[i] = H→element[i/2];
                        i/=2;
                }
                H→element[i] = x;
        }
}
```

→ Delete minimum

Example:




14
16
19
21
19
68
65
26
32
31
14
19
16
21
19
68
65
26
32
31
14
19
16
26
21
19
68
65
32
31
14
19
16
26
21
19
68
65
31
32

```
element_type   delete_min  ( PRIORITY_QUEUE   H)
{
        unsigned  int  i,  child;
        element_type    min_element, last_element;
        if ( is_empty(H))
        {
                cout << "Priority Queue is empty" << end;
                return H->element[0];
        }
        min_element = H->element[0];
        last_element = H->element[H->size--];
        for (i=1; i*2 <= H->size ; i = child)
        {
                child = i * 2;
                if ( ( child != H->size )
                        && ( H->element[child+1] < H->element[child])))
                                child++;
                if ( last_element > H->element[child])
                        H->element[i] = H->element[child];
                else
                        break;
        }
        H->element[i] = last_element;
        return min_element;
}
```

Algorithm to maintain heap property

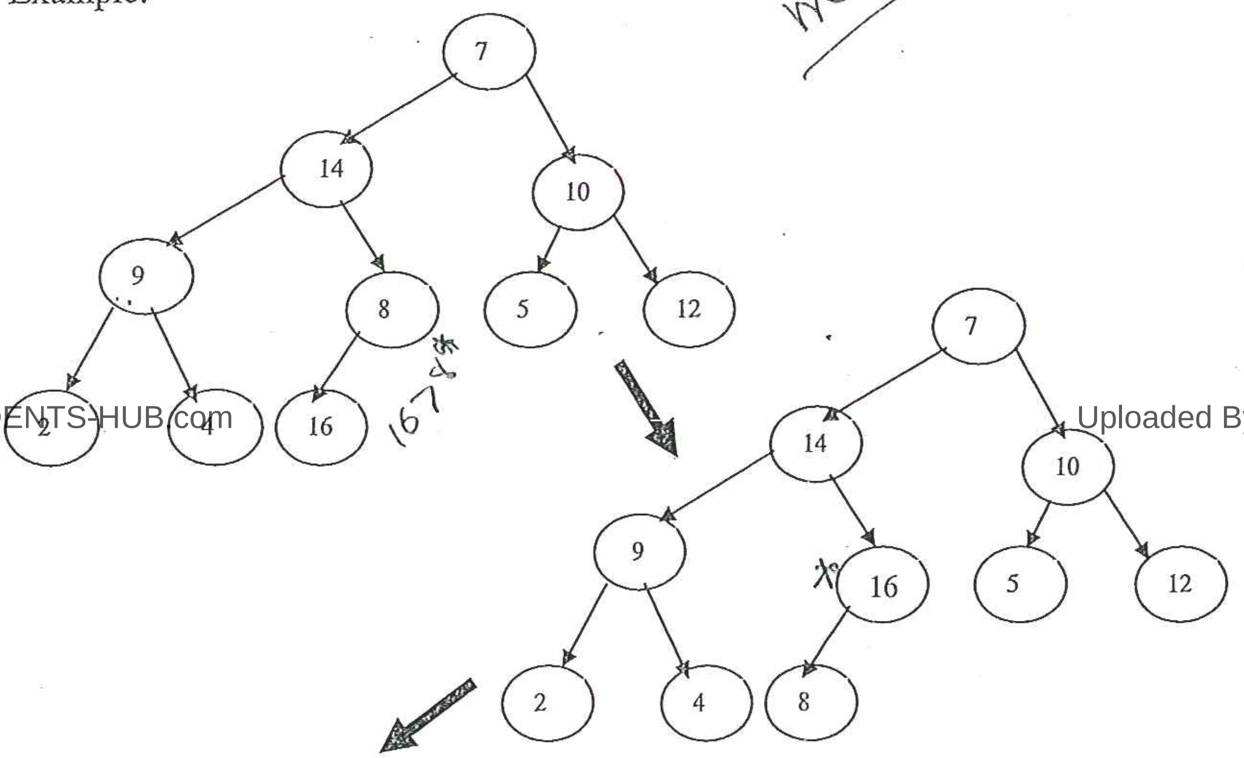
```
Procedure ment_heap_prop ( A, i)
        L  → Left(i)   i*2;
        R  → Right(i)  i*2+1;
If  L <= heap_size   and   A[L] > A[i]  then
        Largest = L;
Else
        Largest = i;
If  R <= heap_size  and   A[R] >= A[Largest])  then
        Largest  = R;
If  Largest  <> i then
        Exchange( A[i] , A[Largest])
Ment_heap_prop( A, Largest)
```

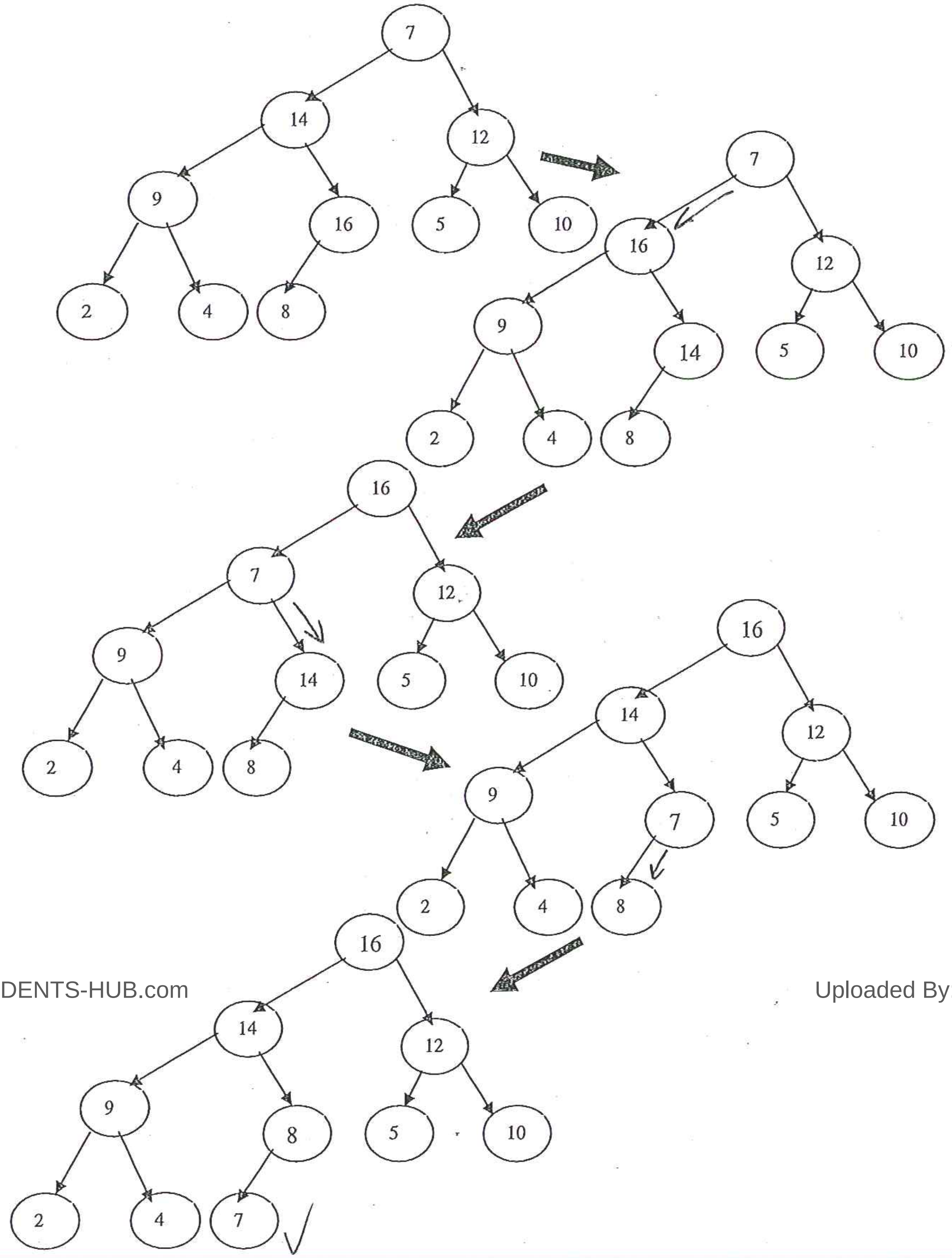
→ Building a heap tree

```
heap_size = length_Array
for ( i= heap_size/2; i >= 1; i--)
        ment_heap_prop(A, i);
```

Example:

max_heap





Sorting Using heap tree

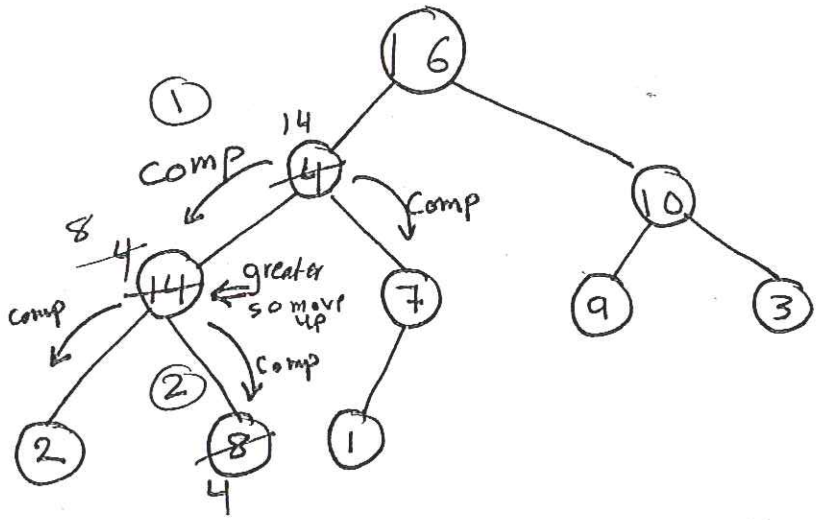
```
Procedure heap_sort(A)
        Build_heap;
        For(i=length(A); i==2; i--)
        {
                Exchange(A[i], A[1]);
                Dec(heap_size);
                Ment_heap_prop(A,i);
        }
```

→Analysis

ment_heap_prop → O(log n)

for statement → O(n)

Heap sort → O(n log n)



this is not a max heap, node with key 4 violates the max heap property.