Heap Property: A tree is said to possess max-heap property if the key or value of each node is NOT less than that of all of its children. Such a property is helpful if we wish to find the maximum element.

A tree is said to have the min-heap property if the key of each node is NOT greater than that of all of its children. This condition helps us to find the minimum element with a complexity of Θ (log n) (which otherwise would have taken Θ (n)).

A min-heap is a tree with min-heap property and similarly max-heap is a tree with max-heap property. We use the word *heap* to denote one of these.

So, a heap is a tree in which vertices are linked without violating the heap property. The heaps are classified on the basis of "linking" of vertices and few of them are described below

Binary Heaps: When the heap property is maintained in binary trees (a tree with each of its nodes containing NOT more than two children), the resultant data structure is called a binary heap. In a binary heap, both the insertion operation and the extraction of extreme element are of order Θ (log n). The deletion of the extreme element and reordering of heap when the key of one of its nodes is modified are of the order Θ (log n). The merge operation of two binary heaps is done in linear order time i.e., of Θ (n). This can be further reduced by Binomial heaps which from another class of heaps.

Percolation:

Percolation(T(1…….n),i)

k ← i

while $\left(\left(\left\lfloor \frac{i}{2} \right\rfloor\right) \geq 1 \, and \, T\left(\left\lfloor \frac{i}{2} \right\rfloor\right) < T(k)\right)$

k ← $\left\lfloor \frac{i}{2} \right\rfloor$

T(k) ↔ T(i)

i ← k

Makeheap(Using Percolation):

Makeheap1(T(1….n))

for i ← 2 to n

percolation(T(1….n),i)

So, time complexity to make the heap using percolation function is

$$t(n) = \sum_{i=2}^{n} O(\log_2 i) = \sum_{i=2}^{n} O(\log_2 n) = O(n \log_2 n)$$

Because, as p(n), time complexity of percolation func. $\in O(\log_2 i)$ then there exist n_0, with c≥0 such that

p (n) ≤ c $(\log_2 i)$ for all n≥ n_0 , i $\in$ [2,n].

Makeheap(Using Siftdown):

```
Makeheap2(T(1….n))
                for i ← ⌊n/2⌋ down to 1
                    Siftdown(T(1….n),i)
```

Siftdown Function:

```
Siftdown (T(1,….,n),i)
 k ← i
 Repeat
       j ← k
       if(  2j <= n  && T(2j)>T(k)  )
              then k ← 2j
       if(  2j<n   &&   T(2j+1)> T(k) )
              then k ← 2j+1
       T(j)  ↔   T(k)
Until j = = k
```

Remark: [**Max heap**: an element with the greatest key is always in the root node and if *B* is a child node of *A*, then key(*A*) ≥ key(*B*).This is the usual heap property.

Min heap: Alternatively, if the comparison is reversed, the smallest element is always in the root node, results in a min heap]

To Siftdown a node at level 1 we visit the "Repeat" 2-times
To Siftdown a node at level 2 we visit the "Repeat" 3-times
: : :
: : :
To Siftdown a node at level k we visit the "Repeat" (k+1)-times

Also
No. of nodes at level 1 2^{k-1}
No. of nodes at level 2 2^{k-2}

:   :   :
:   :   :
No. of nodes at level k-1   2
No. of nodes at level k   1

=> Total no. of visits to “Repeat” loop = $2.\ 2^{k-1} + 3.2^{k-2} + \ldots\ldots\ldots\ldots + k.2 + (k+1).1$

Therefore,

Time complexity, $t(n) \leq 2.\ 2^{k-1} + 3.2^{k-2} + \ldots\ldots\ldots\ldots + k.2 + (k+1).1$

$$= (-2^k + 2^k) + 2.\ 2^{k-1} + 3.2^{k-2} + \ldots\ldots\ldots\ldots + k.2 + (k+1).1$$
$$= -2^k + 2^{k+1}(2^0.2^{-1} + 2.\ 2^{-2} + 3.2^{-3} + \ldots\ldots\ldots\ldots + (k+1).2^{-(k+1)})$$
$$< -2^k + 2^{k+1}(\ 0.5/\ (1-\ 0.5)^{-2}\ )$$
$$= -2^k + 2^{k+1}.2$$
$$= 2^k\ .3$$

Now, $k \in \Theta(\log_2 n)$.

=> $t(n) < n.3$

Hence, $t(n) \in \Theta(n)$.

Remark:

Hence function Makeheap2 (using Siftdown function) is better than Makeheap1 (which uses percolate function).Note the time complexities in the two cases.

Deleting the maximum node in a heap

Deletemax(H(1,..,n))
  Max $\leftarrow$ H(1)
  H(1) $\leftrightarrow$ H(n)
  n$\leftarrow$n-1
  Siftdown(H(1,..,n),1)

Sorting a heap

Heapsort(T(1,..,n))
    Makeheap(T(1,….,n))   ….O(n)
    for i$\leftarrow$n down to 2   ….n-2+1
        T(1) $\leftrightarrow$ T(i)   ….c_1
        Siftdown(T(1,…,i-1),1)   ….log(i-1)

$t(n) = O(n) + O(n.\log n) \approx O(n.\log n)$

Note: This method of sorting is called <u>inplace</u> sorting i.e. sorting done within the given data structure.

Insert element in a heap

```
Insertheap(H(1,…,n),x)
        H(n+1)← x
        Percolate(H(1,…,n+1),n+1)
```