• Divide and Conquer Paradigm

- **Divide** : Divide the problem into a number of subproblems that are smaller instances of the same problem.

Conquer:- The subproblems by solving them recursively. If the subproblem sizes are small enough, however, just solve the subproblems in a straightforward manner.

Combine:- The solutions to the subproblems into the solution for the original problem.

* :- Sum of n natural numbers $= \frac{n(n+1)}{2}$

$$\sum_{i=1}^{n} i = \frac{n(n+1)}{2} \qquad \sum_{i=1}^{n} (i-1) = \frac{n(n-1)}{2}$$

• **Bit**

→ Left Shift by one = Multiplication by 2

eg: $32 << 1 = 64$ $\quad 64 << 1 = 128$

→ Right Shift by one = Division by 2

eg: $128 >> 1 = 64$ $\quad 64 >> 1 = 32.$

• **Logarithm** – Logarithm basically means in how many times we can divide the value inside parenthesis by the base to make it 1.

eg: $\log_2 8 \rightarrow 8 \rightarrow 4 \rightarrow 2 \rightarrow 1.$ (3 times)
$= 3.$

$\log_{10} 100 \rightarrow 100 \rightarrow 10 \rightarrow 1$ (2 times)
$\hookrightarrow 2.$

• **Master Theorem :** (We use it to solve recurrence problems)

↓

problems/equations of the form:

eg: $T(n) = aT(n/b) + \Theta(n)$

→ $\boxed{T(n) = aT(n/b) + \Theta(n^k \log^p n)}$

where, $a \geq 1$, $b > 1$, $k \geq 0$ and p is real number.

i) if $a > b^k$, then $T(n) = \Theta(n^{\log_b a})$

ii) if $a = b^k$

a) if $p > -1$, then $T(n) = \Theta(n^{\log_b a} \log^{p+1} n)$

b) if $p = -1$, then $T(n) = \Theta(n^{\log_b a} \log\log n)$

c) if $p < -1$, then $T(n) = \Theta(n^{\log_b a})$

iii) if $a < b^k$

a) if $p \geq 0$, then $T(n) = \Theta(n^k \log^p n)$

b) if $p < 0$, then $T(n) = \Theta(n^k)$

eg: $T(n) = 3T(n/2) + n^2$

[if p is not given, then assume $p=0$]

→ $a = 3$, $b = 2$, $k = 2$, $p = 0$

a	b^k
3	2^2
$3 < 4$	

→ condition iii.(a) : $T(n) = \Theta(n^k \log^p n)$

$\Rightarrow T(n) = \Theta(n^2 \log^0 n) \Rightarrow \Theta(n^2)$

eg: $T(n) = 2T(n/2) + n/\log n \Rightarrow 2T(n/2) + n\log^{-1} n$

$\rightarrow a=2,\ b=2,\ k=1,\ p=-1$

$a \quad b^k$

$2 \quad 2^1$

$a = b^k$

$\Rightarrow$ condition (ii)(b)

$T(n) = \Theta(n^{\log_b a} \log\log n)$

$= \Theta(n^{\log_2 2} \log\log n)$

$T(n) = \Theta \log\log n$

- Time Complexity of some common Sorting Algorithms.

Algorithm	Worst-case running time	Average-case/ expected running time
Insertion Sort	$\Theta(n^2)$	$\Theta(n^2)$
Merge Sort	$\Theta(n \log n)$	$\Theta(n \log n)$
Heap Sort	$O(n \log n)$	—
Quick Sort	$\Theta(n^2)$	$\Theta(n \log n)$ (expected)
Counting Sort	$\Theta(k+n)$	$\Theta(k+n$
Radix Sort	$\Theta(d(n+k))$	$\Theta(d(n+k))$
Bucket Sort	$\Theta(n^2)$	$\Theta(n)$ (average-case)

- k: For Counting sort, the items to sort are integers in the set $\{0, 1, \ldots, k\}$.

- **Recurrence:** A recurrence is an equation or inequality that describes a function in terms of it's value on smaller inputs.

Three methods for solving recurrences:

i. Substitution Method

ii. Recursion-tree Method

iii. Master method

↳ of the form

$$T(n) = aT(n/b) + f(n)$$

It depicts a divide-and-conquer paradigm.

where,

n = size of input

a = number of subproblems in the recursion

n/b = size of each subproblem. All subproblems are assumed to have the same size.

$f(n)$ = cost of the work done outside the recursive call, which includes the cost of dividing the problem and cost of merging the solution.

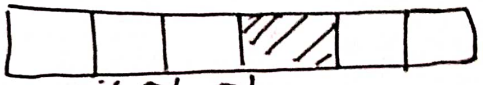
- Here, $a \geq 1$ and $b > 1$ are constants, and $f(n)$ is an asymptotically positive function.

Sorting Algorithms

- **INSERTION SORT :**
  - Time Complexity :
    - Worst case: $O(n^2)$
    - Average case: $O(n^2)$
  - In-place Algorithm

// Pseudo Code

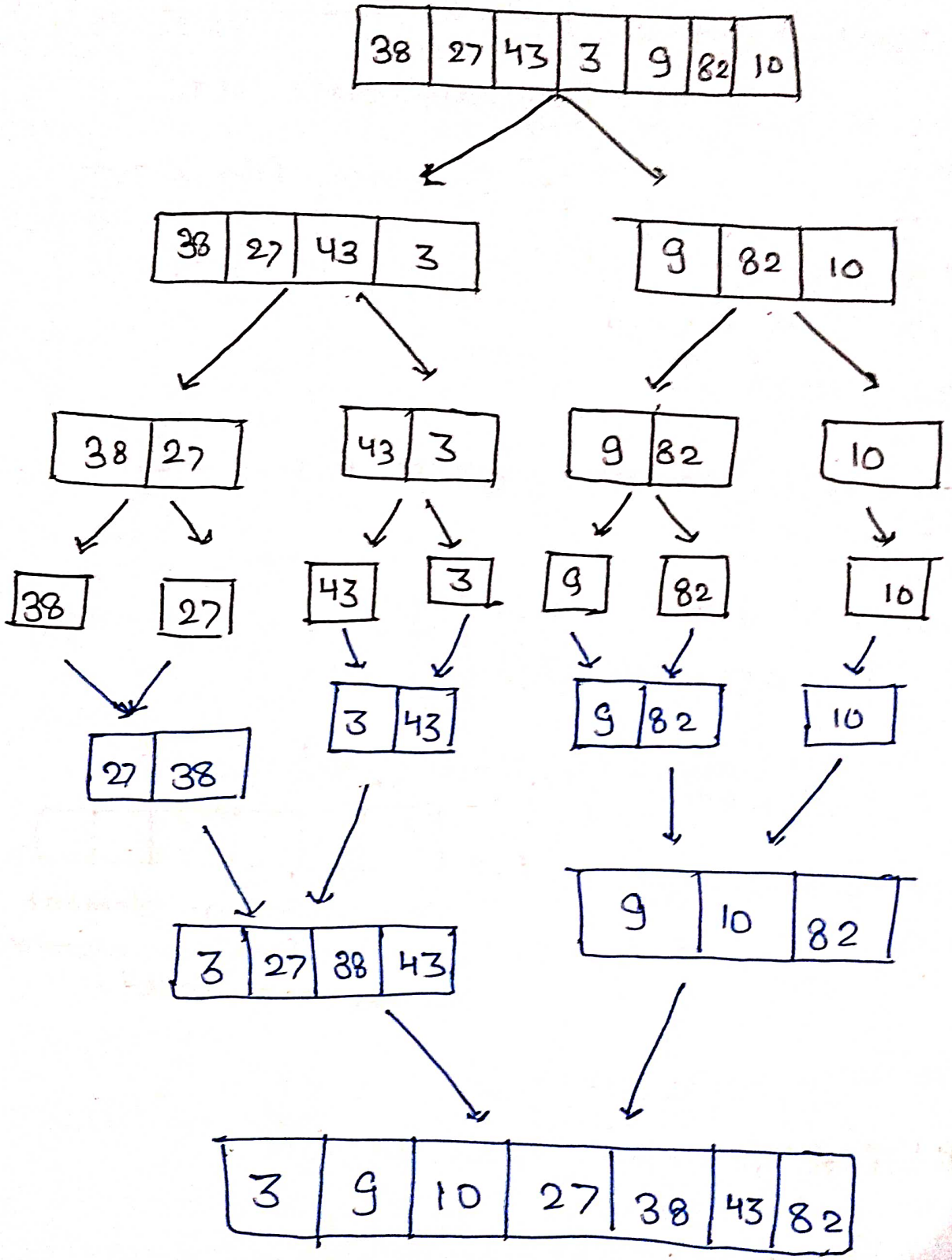
```
INSERTION SORT (A)
    for  j = 2  to  A.length
        Key = A[j]
    // Insert A[j] into the sorted sequence A[1 .... j-1]
        i = j-1
        while  i>0  and  A[i] > Key
            A[i+1] = A[i]
            i = i-1
        A[i+1] = Key
```



we compare an element with it's previous elements in insertion sort.

- **MERGE SORT:**
  - Time Complexity:
    - Worst Case : $O(n \log n)$
    - Average case: $O(n \log n)$
    - Space Complexity : $O(n)$ (space required for merge)
  - Out-of-place Algorithm

// Pseudocode


38 27 43 3 9 82 10
38 27 43 3
9 82 10
38 27
43 3
9 82
10
38
27
43
3
9
82
10
27 38
3 43
9 82
10
3 27 38 43
9 10 82
3 9 10 27 38 43 82

```
MERGE (A, p, q, r)
    n1 = q - p + 1
    n2 = r - q
    let L[1....n1+1] and R[1....n2+1] be new arrays
    for i = 1 to n1
        L[i] = A[p + i - 1]
    for j = 1 to n2
        R[j] = A[q + j]
```

} Copying elements into the newly created arrays.

```
    L[n1 + 1] = ∞
    R[n2 + 1] = ∞
    i = 1
    j = 1
    for k = p to r
        if L[i] ≤ R[j]
            A[k] = L[i]
            i = i + 1
        else A[k] = R[j]
            j = j + 1
```

} Copying/Combining elements in the array A (in sorted order)

```
MERGE-SORT (A, p, r)        (p: low, r: high)
    if p < r
        q = ⌊(p + r)/2⌋      (q: mid)
        MERGE-SORT (A, p, q)
        MERGE-SORT (A, q+1, r)
        MERGE (A, p, q, r)
```

- Analyzing some different kind of operations for different kind of Inputs.

	insert	Search	Find Min	Delete min
Unsorted Array	O(1)	O(n)	O(n)	O(n)
Sorted Array	O(n)	O(logn)	O(1)	O(n)
Unsorted Linked list	O(1)	O(n)	O(n)	O(n)
Min Heap	O(logn)		O(1)	O(logn)

- Observing some of the above examples, heaps seems to be an efficient one among others.

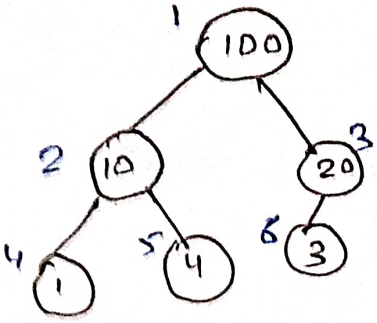
• Heap

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. The tree is completely filled on all levels except possibly the lowest (leaf nodes), which is filled in the left to right order.

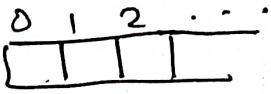
Eg:

1	2	3	4	5	6
100	10	20	1	4	3

Note: We usually assume 1 based indexing while going through heaps.



- How to know, left child, right child for a node with index i -



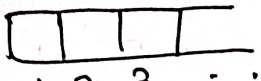
For 0 based Indexing:

Left Child = $2 * i + 1$

Right Child = $2 * i + 2$

Parent Node = $(i-1)/2$

For 1^ based indexing

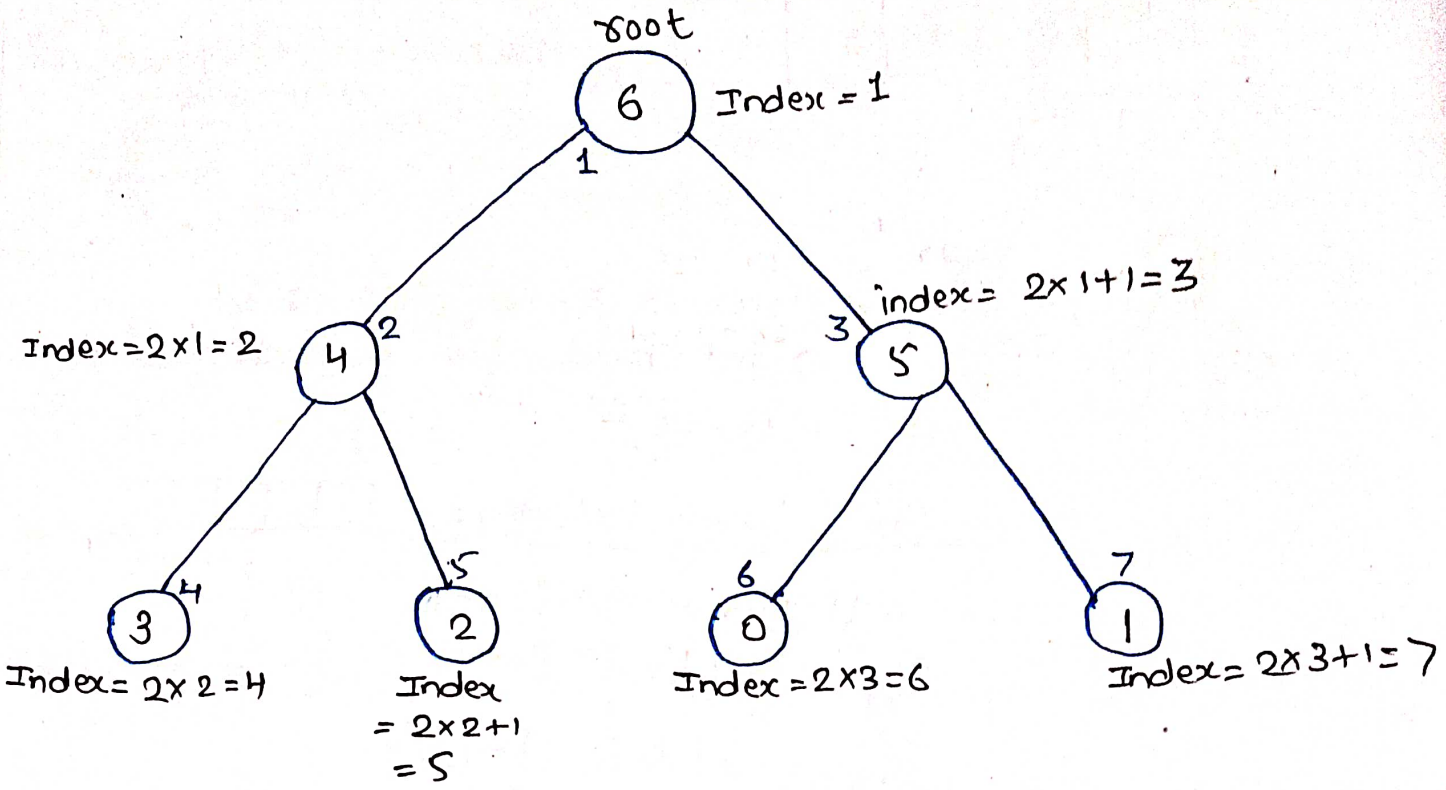


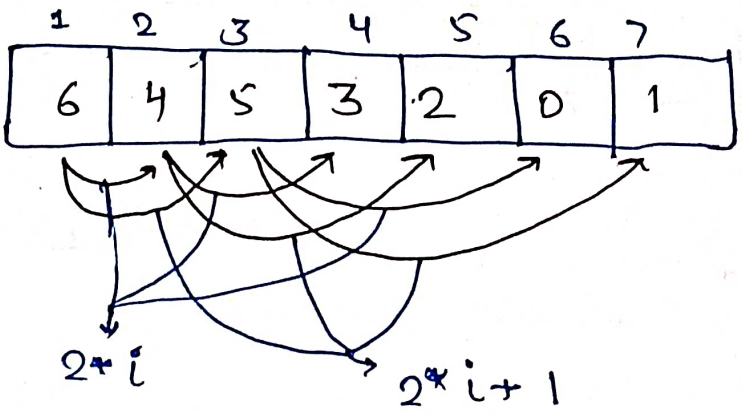
Left Child = $2 * i$

Right Child = $2 * i + 1$

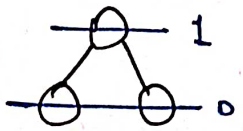
Parent Node = $i/2$

Note: An array can be used to simulate a tree in the following way. If we are storing one element and index i in the array Arr, then its parent will be stored at index $i/2$ (unless its a root, as root has no parent) and can be accessed by Arr[$i/2$], and its left child can be accessed by Arr[$2*i$] and its right child can be accessed by Arr[$2*i+1$]. Index of root will be 1 in an array.

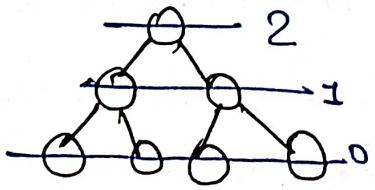




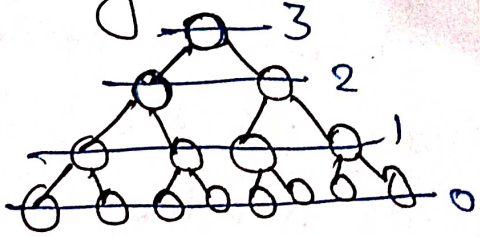
- Some properties of Complete Binary Trees:



Height = 1
maxNodes = 3



Height = 2
maxNodes = 7



Height = 3
maxNodes = 15

Height	1	2	3
maxNodes	3	7	15

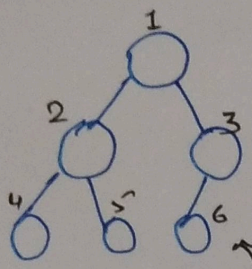
No. of Nodes

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

- For 3-ary trees
  $max = 3^{h+1} - 1$
- For n-ary trees
  $maxNodes = n^{h+1} - 1$

- Height of any heap / binary tree: $O(\log n)$
- Total No. of leaves in a complete Binary Tree

$n = 6$.



↑ In this case

Total no. of leaves = $\lfloor n/2 \rfloor + 1$ to n

leaves = $\lfloor n/2 \rfloor + 1$ to n = $\lfloor 6/2 \rfloor + 1$ to 6 = 4 to 6.

∴ leaf nodes = 4, 5, 6.

★★ • And all the nodes from $\lfloor n/2 \rfloor$ down to 1 (root node) are non-leaf nodes.

- Note: All leaf nodes are ~~binary~~ heaps in themselves of one element.

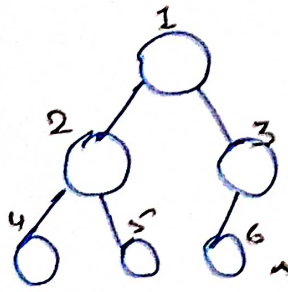
Conclusion: Since we know that leaf nodes are heaps in themselves, thus we only need to check/calculate heap for nodes other than the leaf nodes. Which, in the case of complete binary trees are: $\lfloor n/2 \rfloor$ down to 1 (root

- For 3-ary trees
  $max = 3^{h+1} - 1$
- For n-ary trees
  $maxNodes = n^{h+1} - 1$

- Height of any heap / binary tree: $O(\log n)$

- **Total No. of leaves in a complete Binary Tree**

$n = 6$.



Total no. of leaves = $\lfloor n/2 \rfloor$ 1 to n

↑ In this case

leaves = $\lfloor n/2 \rfloor + 1$ to $n = \lfloor 6/2 \rfloor + 1$ to $6 = 4$ to 6.

∴ leaf nodes = 4, 5, 6.

** • And all the nodes from $\lfloor n/2 \rfloor$ down to 1 (root node) are non-leaf nodes.

- Note: All leaf nodes are ~~binary~~ heaps in themselves of one element.

Conclusion: Since we know that leaf nodes are heaps in themselves, thus we only need to check/calculate heap for nodes other than the leaf nodes. Which, in the case of complete binary trees are: $\lfloor n/2 \rfloor$ down to 1 (root node)

- There can be two types of heap:

i. Max Heap:

In this type of heap, the value of parent node will always be greater than or equal to the value of child node across the tree and the node with highest value will be root node of the tree.

Implementation :

//PseudoCode

```
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] with A[largest]
      MAX-HEAPIFY (A, largest)
```

- Building a heap

```
BUILD-MAX-HEAP (A)
  A.heapsize = A.length
  for i = ⌊A.length/2⌋ downto 1
      MAX-HEAPIFY (A, i)
```

// It means to heapify only the nodes other than the leaf nodes.

// Code:

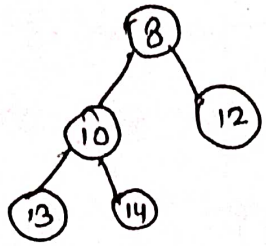
```
void max_heapify (int Arr[], int i)
{
    int left = 2*i;        //left child
    int right = 2*i+1;     // right child
    if (left <= heapsize && Arr[left] > Arr[i])
        largest = left;
    else
        largest = i;
    if (right <= heapsize && Arr[right] > Arr[largest])
        largest = right;
    if (largest != i)
    { swap(Arr[i], Arr[largest]);
      max_heapify (Arr, largest);
    }
}
```

Complexity: O(logn)

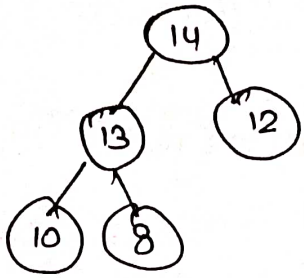
```
void build_maxheap (int Arr[])
{
    for ( i = ⌊A.length/2⌋ ; i>=1 ; i--)
    {
        max_heapify (Arr, i);
    }
}
```

Complexity: O(N)

- 8, 10, 12, 13, 14 → If the array is in ascending order, then it is always a minHeap.



- 14, 13, 12, 10, 8 → If the array is in descending order, then it's always a max-Heap.



- Applications:

Heap Sort:

Idea
- Initially we will build a max heap of elements in Arr.
- Now the root element that is Arr[1] contains maximum element of Arr. After that, we will exchange this element with the last element of Arr and will again build a max heap excluding the last element which is already in its correct position and will ~~dec~~ decrease the length of heap by one.
- We will repeat the step 2, until we get all the elements in their correct position.
- We will get a sorted array.

Implementation:

// Pseudocode

```
HEAPSORT (A)
 BUILD-MAX-HEAP(A)
 for i = A.length downto 2
      exchange A[1] with A[i]
      A.heapsize = A.heapsize -1
       MAX-HEAPIFY (A,1)
```

// Code

```
void heap_sort (int Arr[])
 {  int heap_size = N;
    build_maxheap (Arr);
    for(int i=N ; i>=2 ; i--)
      {  swap(Arr[1], Arr[i]);
         heap_size = heap_size-1;
         max_heapify (Arr,1)
      }
} Complexity : O(n logn)
                 |      |
          build maxheap  max_heapify.
```

- It is an in-place sorting Algorithm.

where

merge sort is an out of place Algorithm.

O(NlogN)