(1) Bubble Sort:

- Bubble Sort is the simplest sorting algorithm that works by repeatedly swapping the adjacent elements if they are in wrong order.
- If we have total n elements, then we need to repeat this process for n-1 times.
- It is known as bubble sort, because with every complete iteration the largest element in the given array, bubbles up towards the last place or the highest index, just like a water bubble rises up to the water surface.

Algorithm:

```
Bubble_sort (A, n)                    // A is input array
{                                     // n is size of array
    for k:=0 to k<n-1
    {
        for i=0 to i<n-k-1
        {
            if (A[i] > A[i+1])        // swap
            {
                temp = A[i];
                A[i] = A[i+1];
                A[i+1] = temp;
            }
        }
    }
}
```

Example:

	0	1	2	3	4
	8	4	6	9	2

k	i	A[i]	A[i+1]	if condition					
0	0	8	4	✓	4	8	6	9	2
	1	8	6	✓	4	6	8	9	2
	2	8	9	x	4	6	8	9	2
	3	9	2	✓	4	6	8	2	9
1	0	4	6	x	4	6	8	2	9
	1	6	8	x	4	6	8	2	9
	2	8	2	✓	4	6	2	8	9
2	0	4	6	x	4	6	2	8	9
	1	6	2	✓	4	2	6	8	9
3	0	4	2	✓	2	4	6	8	9

Time Complexity:

Algorithm	Time Complexity		
	Best	Average	Worst
Bubble sort	$O(n)$	$O(n^2)$	$O(n^2)$

(2) Insertion Sort

- Insertion sort is a simple sorting algorithm, that is appropriate for small inputs.
- The list is divided into two parts: sorted and unsorted.
- In each pass, the first element of the unsorted part is picked up, transtered to the sorted sublist, and inserted at the appropriate place.
- A list of n elements will take at most n-1 passes to sort the data.

Algorithm:

```
Insertion_sort (A, n)
{
  for ( i ← 1  to  i < n-1)
    {
        value ← A[i]
        hole ← i
        while ( hole > 0  &&  A[hole-1] > value)
          {
              A[hole] ← A[hole-1]
              hole ← hole-1
          }
        A[hole] ← value
    }
}
```

Example:

0	1	2	3	4
8	4	6	9	2

i	value	A[j]	hole	while Condition	
					8 _ 6 9 2 value=4
1	4	4	1	✓	8 _ 8 6 9 2
			0	X	4 8 6 9 2
					4 8 _ 9 2 value=6
2	6	6	2	✓	4 _ 8 9 2
			1	X	4 6 8 9 2
					4 6 8 _ 2 value=9
3	9	9	3	X	4 6 8 9 2
					4 6 8 9 _ value=2
4	2	2	4	✓	4 6 8 _ 9
			3	✓	4 6 _ 8 9
			2	✓	4 _ 6 8 9
			1	✓	_ 4 6 8 9
			0	X	2 4 6 8 9

Time Complexity

Algorithm	Time complexity		
	Best	Average	Worst
Insertion Sort	O(n)	$O(n^2)$	$O(n^2)$

(3) Selection Sort:

- Selection sort is an in-place comparison-based algorithm.

- Idea:
  - Find the smallest element in the array.
  - Exchange it with the element in the first position.
  - Find the second smallest element and exchange it with the element in the second position.
  - Continue until the array is sorted.

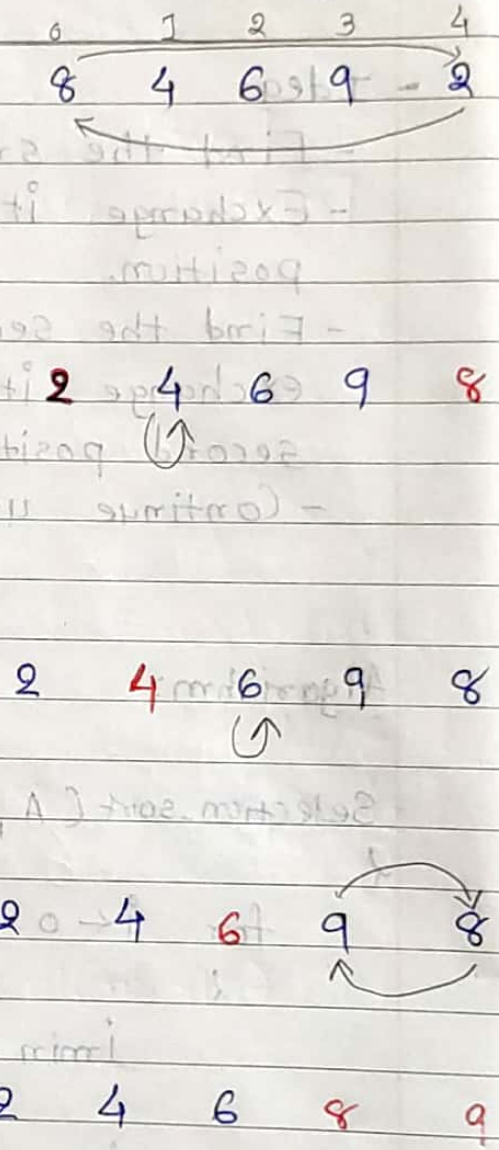
Algorithm:

```
Selection_sort(A, n)
{
  for i ← 0 to [redacted] i < n-1
  {
      imin ← i

      for j ← i+1 to n-1
      {
          if (A[j] > A[imin])
          imin ← j
      }

      temp ← A[i]
      A[i] ← A[imin]
      A[imin] ← temp
  }
}
```

Example:

	0	1	2	3	4
	8	4	6	9	2

i	imin	j	A[j]	A[imin]	if Condition
0	0	1	4	8	✓
	1	2	6	4	x
	1	3	9	4	x
	1	4	2	4	✓
		imin			
1	1	2	6	4	x
	1	3	9	4	x
	1	4	8	4	x
	imin				
2	2	3	9	6	x
	2	4	8	6	x
	imin				
3	3	4	8	9	✓
		imin			



Time Complexity:

Algorithm	Time Complexity		
Selection Sort	Best	Average	Worst
	$O(n^2)$	$O(n^2)$	$O(n^2)$

(4) Merge Sort:

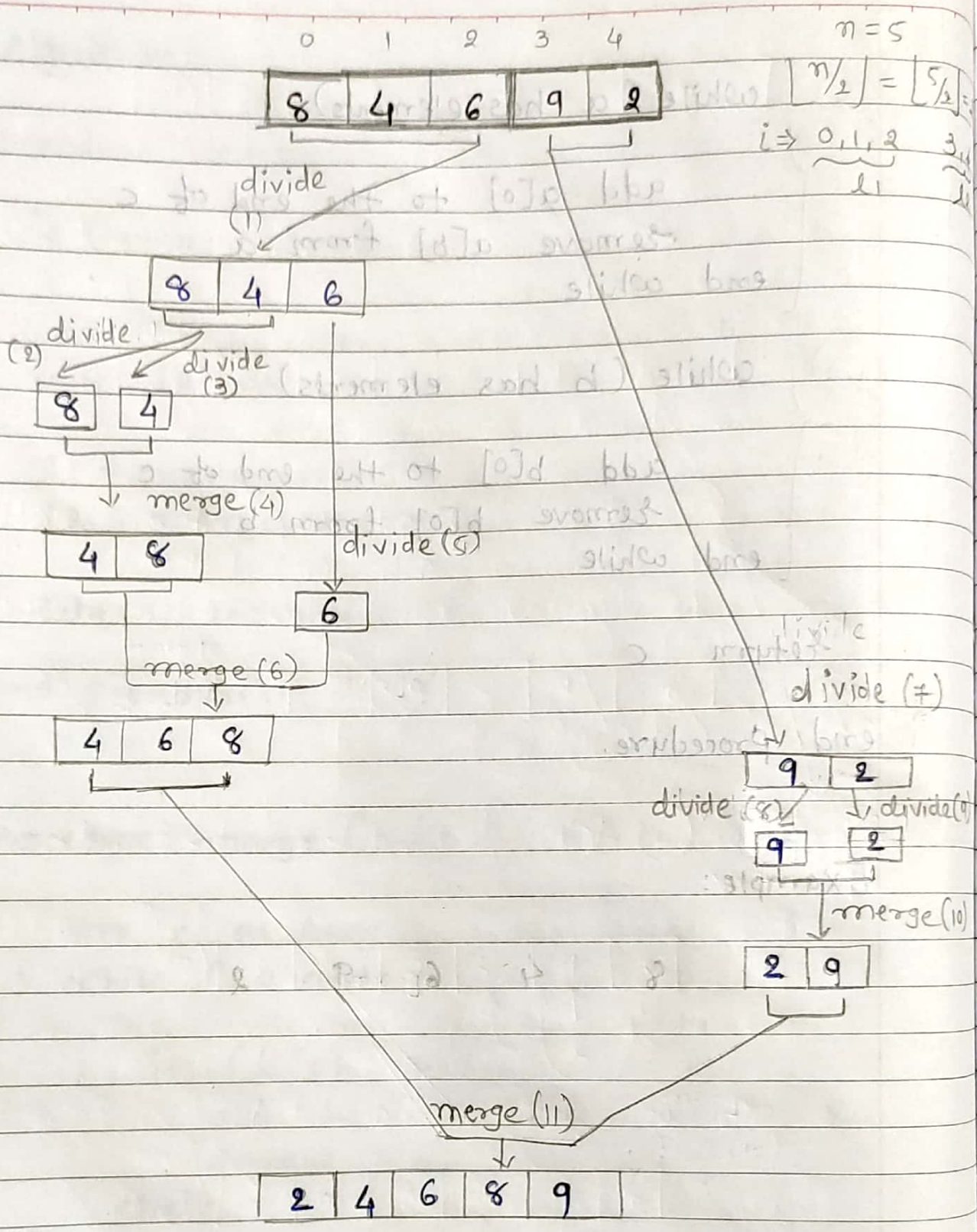
- Merge sort uses strategy of divide and conquer.

(i)- In that, we divide the unsorted array into 2 halves until the sub-arrays only contain one element.

(ii) - Merge the sub-problem solutions together:
- Compare the sub-array's first elements.
- Remove the smallest element and put into the result array.
- Continue the process until all elements have been put into the result array.

Algorithm:

```
Mergesort (Passed an array)

    if array size >1
        Divide array in half
        call Mergesort on first half.
        call Mergesort on second half.
        Merge two halves.

Merge (Passed two arrays)
        Compare leading element in each array
        Select lower and place in new array.
        (If one input array is empty then
         place remainder of other array in
         Output array)
```



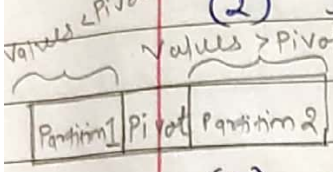
Time complexity

	Best	Average	Worst
Merge sort	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$

(5) Quick Sort.

- Like merge sort, it also uses divide and conquer strategy.

- Quick sort, works as following---
  (1) Pick a 'pivot'.
  (2) Divide list into two lists:
  - One less than or equal to pivot value.
  - One greater than pivot.
  (3) Sort each sub-problem recursively.
  (4) Answer is the concatenation of the two solutions.



Algorithm:

step-1) choose the lowest index value as pivot.

step-2) Take two variables to point left and right of the list excluding pivot.

step-3) Left points to the low index.

step-4) Right points to the high.

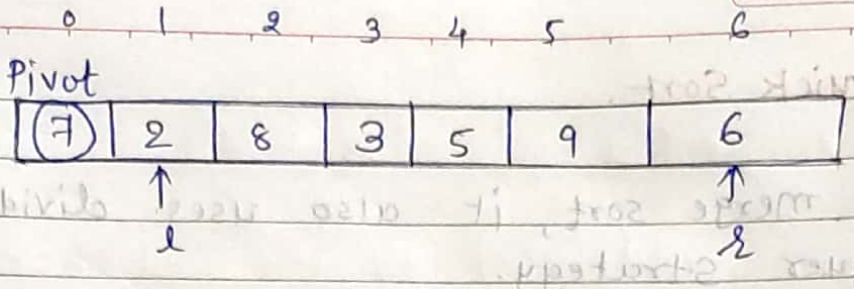
while (left < Right)

step-5) while value at left is less than pivot, move right.

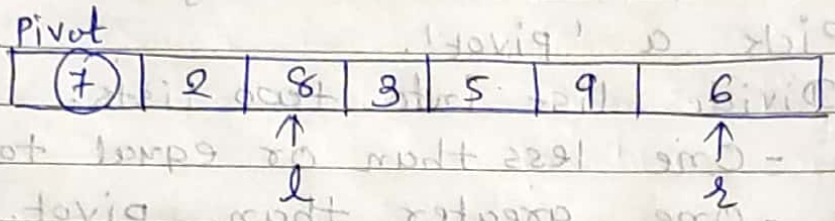
step-6) while value at Right is greater than pivot, move left.

step-7) if both step 5 and 6 not match, swap left and right.

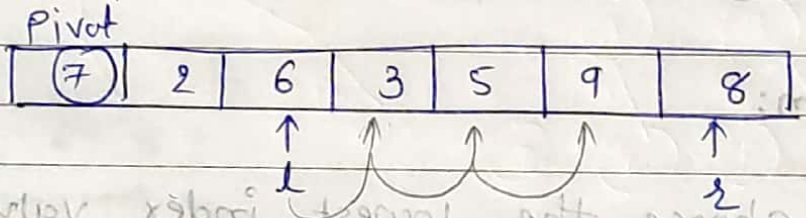
step-8) if left > right, the point where they met is new pivot. (swap value of pivot and right)



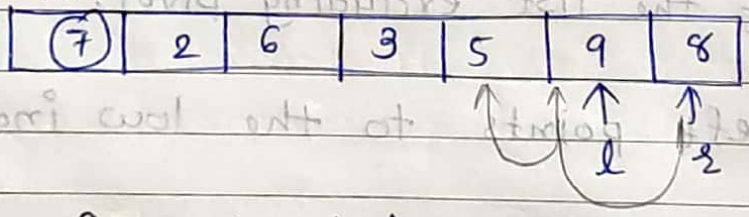
$2 < 7$, so increment l



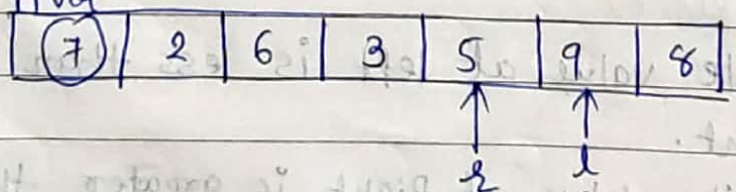
Now, $8 \not< 7$, so stop l
$6 \not< 7$, so stop r.
Now, swap values at l and r



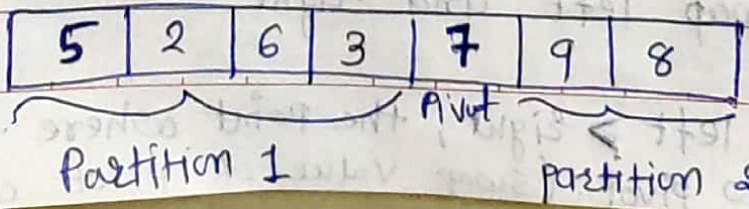
$6 < 7$, $3 < 7$, $5 < 7$, but $9 \not< 7$, so stop l at value 9.



$8 > 7$, $9 \not> 7$, but $5 \not> 7$, so stop r at value 5



Now, $l > r$
So, swap value of pivot and 'r'

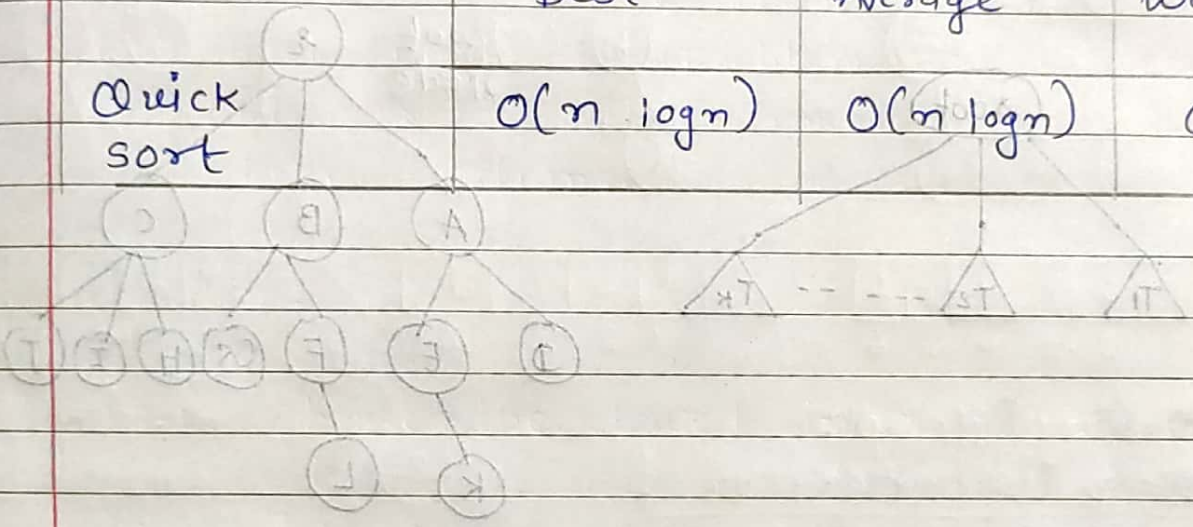


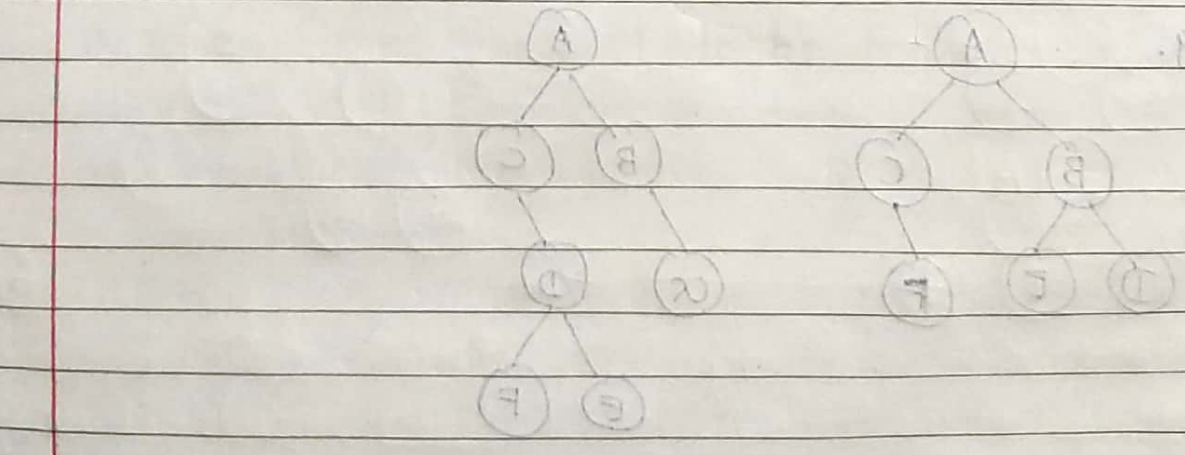
⇒ Recursively sort each partitions.
You will get sorted list like this

2	3	5	6	7	8	9

Time Complexity

Algorithm	Time Complexity		
	Best	Average	Worst
Quick sort	$O(n \log n)$	$O(n \log n)$	$O(n^2)$

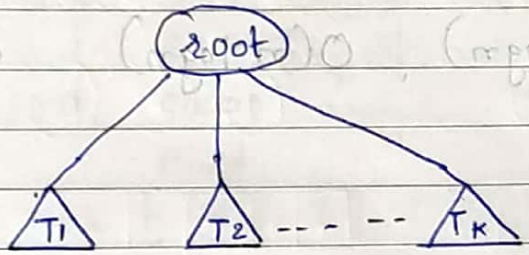


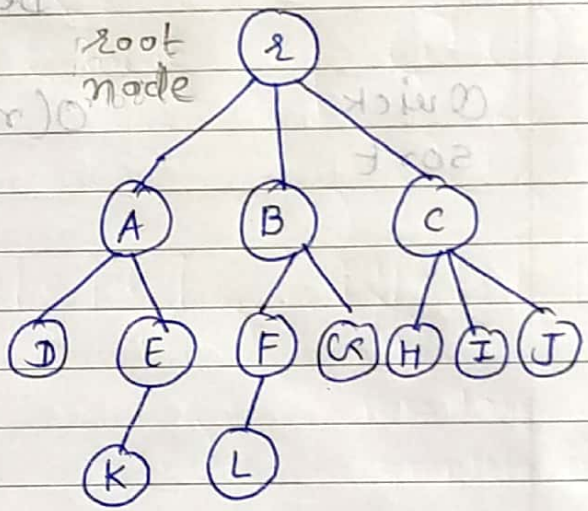


(6) Heap sort:

*) Basic Concepts:-

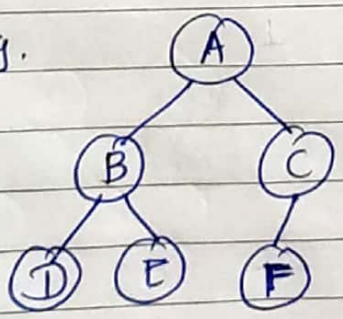
(1) Tree:
A tree is a collection of elements called "nodes", one of which is distinguished as a root say r, along with a relation "parenthood" that places a hierarchical structure on the nodes.

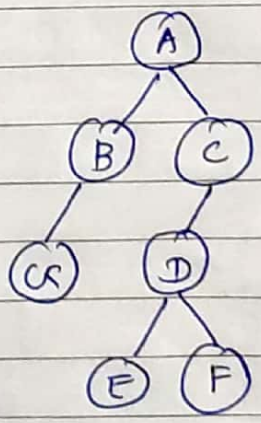




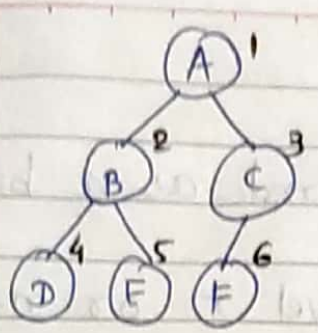
(2) Binary Tree:
A tree is binary if each node of the tree can have maximum of two children.

e.g.

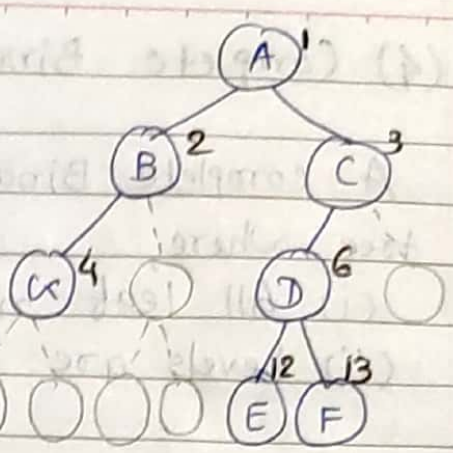




Array Representation:



A	B	C	D	E	F
1	2	3	4	5	6



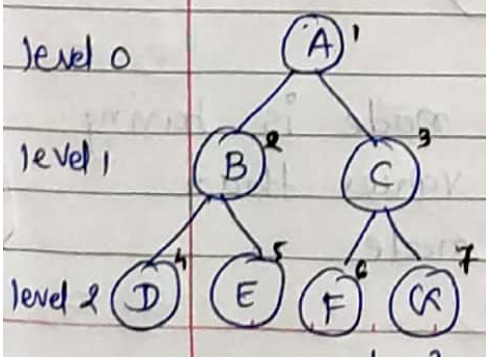
A	B	C	G	10	D	10	10	10	10	10	E	F
1	2	3	4	5	6	7	8	9	10	11	12	13

For any node at index = i
Left child of i^{th} node = $2*i$
Right child of i^{th} node = $(2*i)+1$
Parent node of i^{th} node = $\lfloor \frac{i}{2} \rfloor$

e.g for node D, i = 6
left child = $2*i = 2*6 = 12 \Rightarrow$ node E
Right child = $(2*i)+1 = (2*6)+1 = 13 \Rightarrow$ node F
Parent node = $\lfloor \frac{i}{2} \rfloor = \lfloor \frac{6}{2} \rfloor = 3 \Rightarrow$ node C

(3) Full Binary Tree:

A binary tree is said to be full binary tree if each of node has either two children or no child at all.



Total no. of nodes in full binary tree
$= 2^{h+1} - 1 = 2^{2+1} - 1 = 2^3 - 1 = 7$

Array Representation:

A	B	C	D	E	F	G
1	2	3	4	5	6	7

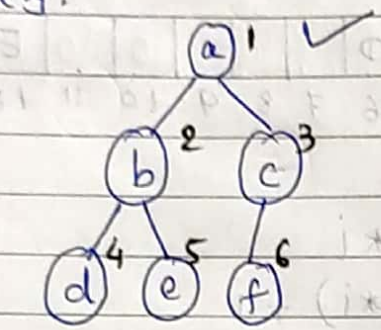
* (4) Complete Binary tree:

A complete Binary tree is defined as a binary tree where,

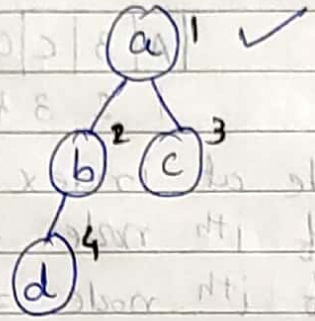
(i) All leaf nodes are on level n or n-1

(ii) Levels are filled from left to right

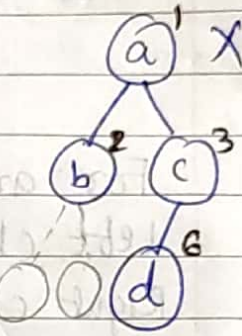
e.g.



✓



✓



X

a	b	c	d	e	f
1	2	3	4	5	6

a	b	c	d
1	2	3	4

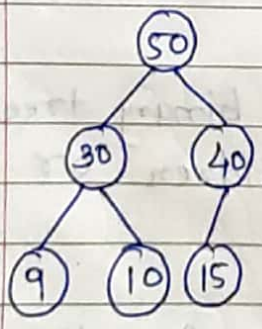
a	b	c	10	10	d
1	2	3	4	5	6

This tree is not Complete b.T.

Binary Heap:

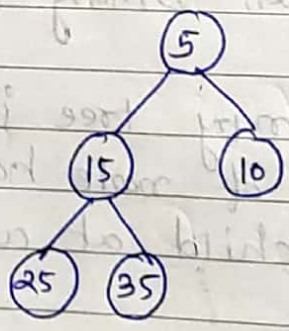
→ BHeap is an example of Complete binary tree.

(5) Max heap



Parent node is having higher values than child node

(6) Min heap



Parent node is having lower values than child node.

Heap Sort

- To perform heap sort, we use a max-heap.
- Suppose, we have given unsorted list as follows,
  7, 5, 6, 2, 4 (N=5)

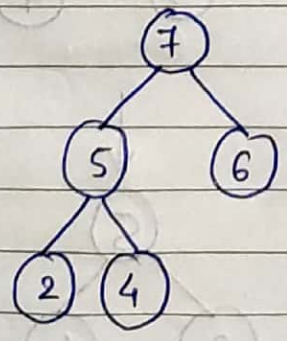
Now, we need to perform sorting using heap sort...

→ Using Binary heaps for sorting...

- Build a max-heap
- DO N DeleteMax operations and store each Max element as it comes out of the heap.
  [Data comes out in largest to Smalles order, where we put the elements as they are removed from heap?]
- Every time we do a DeleteMax, the heap gets smaller by one node, and we have one more node to store.
  - Store the data at the end of the heap array.
    [Not 'in heap', but it is in the heap array]

e.g 7, 5, 6, 2, 4

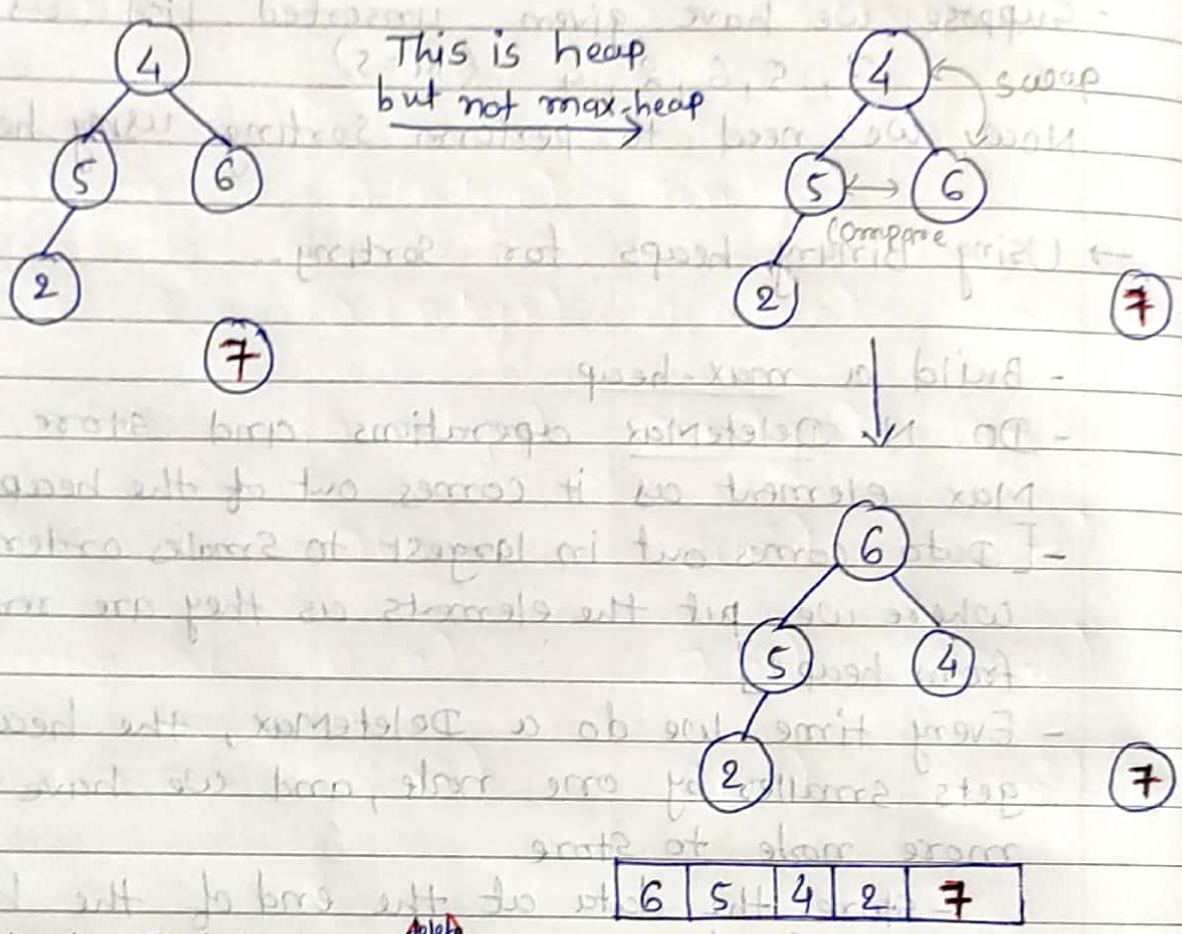
→ Build a max-heap



7	5	6	2	4
1	2	3	4	5

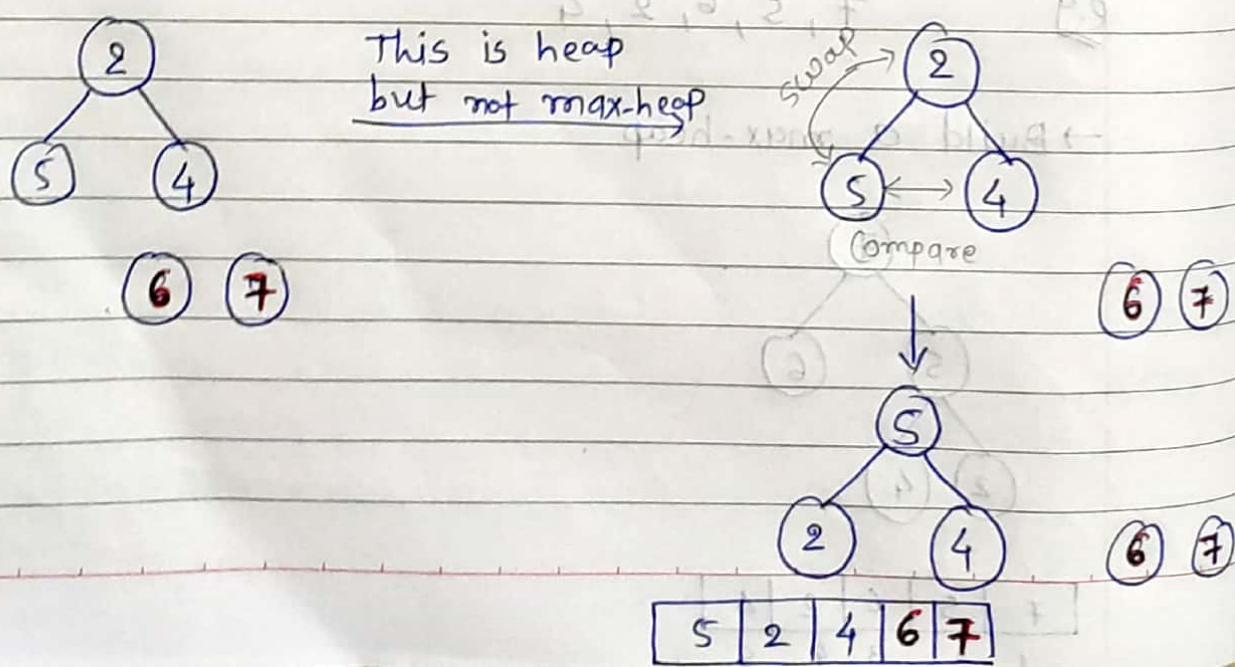
→ Now, DeleteMax, (delete 7)

(If we delete 7, New root will be very last element, i.e. 4)



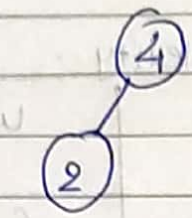
→ Now, DeleteMax, (delete 6)

(If we delete 6, New root will be 2)

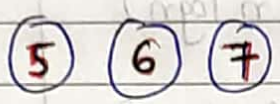


→ Now, DeleteMax, (delete 5)

(If we delete 5, New root will be 4)



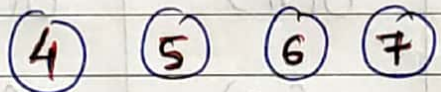
This is max heap,



4	2	5	6	7

→ Now, DeleteMax, (delete 4)

This is max-heap.



2	4	5	6	7

→ Now, DeleteMax, (delete 2)

2	4	5	6	7

⇐ Sorted array

After all the DeleteMax, the heap is gone but the array is full and is in sorted order

Time Complexity:

Algorithm	Time Complexity		
	Best	average	worst
Heap sort	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$

⇒ Time Complexity comparision of Sorting Algorithms

Algorithm	Time Complexity		
	Best	Average	Worst
Quick Sort	$O(n \log n)$	$O(n \log n)$	$O(n^2)$
Merge Sort	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$
Heap Sort	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$
Bubble Sort	$O(n)$	$O(n^2)$	$O(n^2)$
Insertion Sort	$O(n)$	$O(n^2)$	$O(n^2)$
Selection Sort	$O(n^2)$	$O(n^2)$	$O(n^2)$