AA Mod 2

- **Selection Sort**

Selects smallest element from an unsorted list in each iteration and places it at the beginning of the unsorted list.

Example :- 20, 12, 10, 15, 2 — if its

Set 1st element as min^m. Compare with 2nd element, if its min/smaller, assign it as min^m : 20, 12, 10, 15, 2 → 15 > 10

20, 12, 10, 15, 2 — missing a step → 20, 12, 10, 15, 2

Swap the min^m with first element : 2, 12, 10, 15, 20

Repeat the steps. Iteration 2 :- 2, 12, 10, 15, 20

2, 12, 10, 15, 20 → 2, 12, 10, 15, 20 → 2, 10, 12, 15, 20

→ Indexing starts from 1st unsorted element. 2 is sorted.

Iteration 3 :- 2, 10, 12, 15, 20

2, 10, 12, 15, 20 → 2, 10, 12, 15, 20

Iteration 4 :- 2, 10, 12, 15, 20 → 2, 10, 12, 15, 20

```
Selection Sort (array, size)
repeat (size - 1) times
set the first unordered/unsorted element as the min^m for each of
        unsorted elements.
if element < currentMinimum
set element as new minimum
swap minimum with first unsorted position
end selection Sort
```

Time Complexity :-

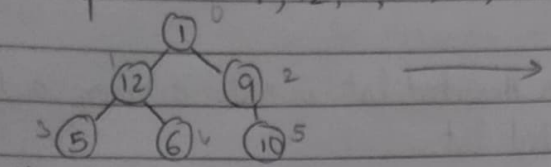
Best : $O(n^2)$: Worst : Average.

- **Heap Sort**

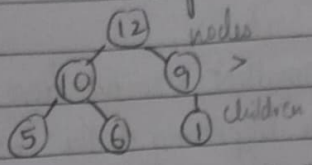
Heap is a complete binary tree, and in heap sort, we eliminate elements one by one from the heap and insert them into sorted part of the list.

0 1 2 3 4 5

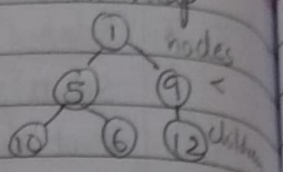
Example :- 1, 12, 9, 5, 6, 10



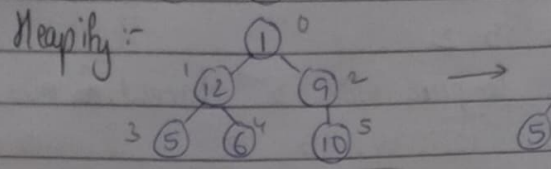
Max Heap

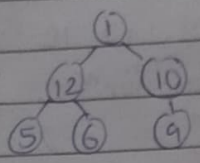


Min Heap



Heapify :-





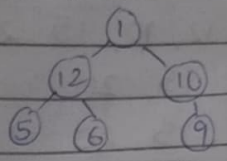
n = 6

i = 6/2 - 1 = 2

index

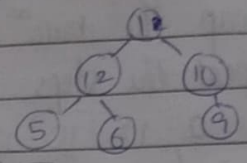
1, 12, 9, 5, 6, 10 → 1, 12, 10, 5, 6, 9

i = 1



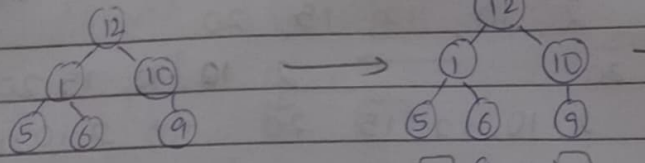
All heapified →

i = 0



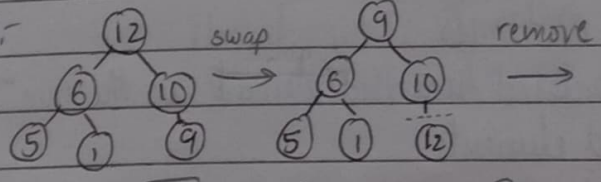
1, 12, 9, 5, 6, 10

i = 0



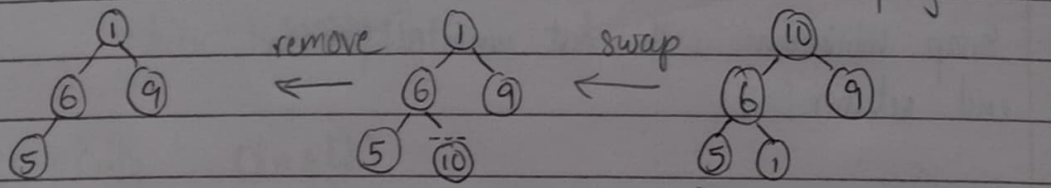
12, 1, 9, 5, 6, 10 → 12, 1, 9, 5, 6, 10 → 12, 6, 9, 5, 1, 10

Heap Sort :-



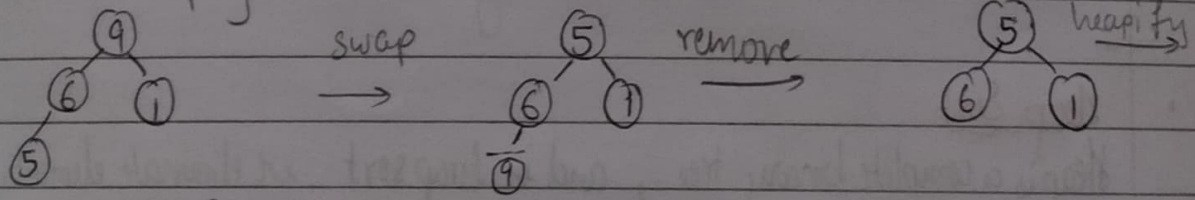
12, 6, 10, 5, 1, 9 → 9, 6, 10, 5, 1, 12 → 9, 6, 10, 5, 1, 12

↓ heapify

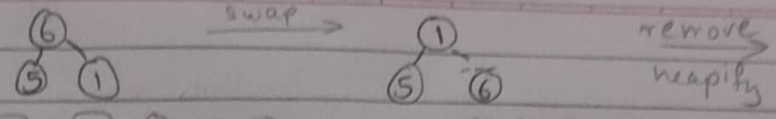


1, 6, 9, 5, 10, 12 ← 1, 6, 9, 5, 10, 12 ← 10, 6, 9, 5, 1, 12

↓ heapify



9, 6, 1, 5, 10, 12 → 5, 6, 1, 9, 10, 12 → 5, 6, 1, 9, 10, 12



[6, 5, 1, 9, 10, 12] [1, 5, 6, 9, 10, 12] [5, 1, 6, 9, 10, 12]

swap, remove

1, 5, 6, 9, 10, 12 ← 1, 5, 6, 9, 10, 12 (1)

Time Complexity :-

Best : O(n log n) = Worst = Average

$T(n) \leq T(2n/3) + \theta(1)$

Heapify → Time complexity O(log n)

```
HeapSort (arr) :
Build Max Heap (arr)
for i := length (arr) to 2
    swap arr[1] with arr[i]
        heap_size[arr] = heap_size[arr] - 1
        Max Heapify (arr, 1)
Build Max Heap (arr):
    heapsize(arr) = length(arr)
    for i := length (arr)/2 to 1
← Max Heapify (arr, i)

Max Heapify (arr, i)
    L = left (i)
    R = right (i)
    if L ≤ heap_size[arr] and arr[L] > arr[i]
    largest = L
    else
    largest = i
    if R ≤ heap_size[arr] and arr[R] > arr[largest]
    largest = R
    if largest != i
    swap arr[i] with arr[largest]
    Max Heapify (arr, largest)
```

- **Radix Sort :-**

Sorts elements by first grouping the individual digits of same place value
Best Case : $O(n+k)$: ar sorted array
Average : jumbled elements : $O(n+k)$
Worst : reverse order : $O(n+k)$
Sort based on 'units' place first. Then 10's then 100's and so on.

Radix Sort (arr)
max = largest element in array
d = number of digits in largest element (or, max)
create d buckets of size 0-9
for i → 0 to d
sort the array elements using counting sort (or any stable sort) according to the digits at the ith place.

Example :-
170, 45, 75, 90, 802, 24, 2, 66
Sorting based on unit digit :-
170, 90, 802, 2, 24, 45, 75, 66
Tens place digit :-
802, 2, 24, 45, 66, 170, 75, 90
100's digit :-
2, 24, 45, 66, 75, 90, 170, 802

- **Counting Sort**

Sorting by counting the number of occurances of each unique element in array. Count is stored in an auxiliary array and sorting is done using this.
Example :-
4, 2, 2, 8, 3, 3, 1 Max : 8
Count array : 0 0 0 0 0 0 0 0 0
0 1 2 3 4 5 6 7 8

Page No.
Date

Add the counts to the array:

0	1	2	2	1	0	0	0	1
0	1	2	3	4	5	6	7	8

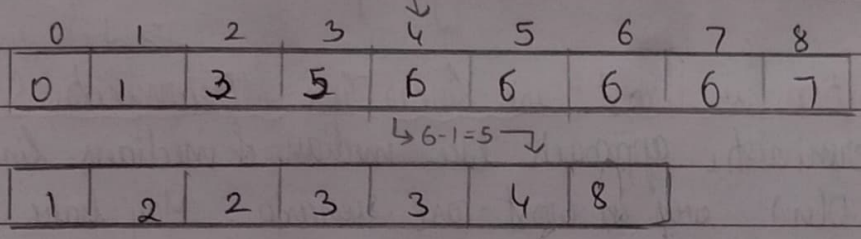
Cumulative sum:

0	1	3	5	6	6	6	6	7

Last step: (Find index of each element of original array in count. Place the element at i-1 index)

array:

4	2	2	8	3	3	1



Time Complexities :-

Best case complexity : $O(n+k)$

Average : $O(n+k)$

Worst case : $O(n+k)$

- **Sorting in Linear Time :-**
  Ability to sort a list of elements in $O(n)$ time complexity where n is the no. of elements in the list. Counting Sort, Radix Sort, Bucket Sort.

- **Lower Bounds for Sorting**
  $O(n \log n)$ is time complexity for n elements to be sorted in worst case, and it solely depends on element comparisons.
  Algorithms: quicksort, mergesort, heapsort.

- **Medians and order Statistics : Minimum & Max^m**
  Minimum and maximum can be found in $O(n)$ time by scanning through the array or in $O(\log n)$ time using divide and conquer

methods. Medians can be found in $O(n \log n)$ time for unsorted arrays, while order statistics like finding the i^{th} smallest or largest element can be achieved in $O(n)$ time.

- **Selection in Expected Linear Time (Randomized Selection)** → faster & simpler
  Utilizes randomized partitioning around a pivot element, similar to quicksort. Achieves an average-case time complexity of $O(n)$ by randomly selecting pivots. May have a worst case time complexity of $O(n^2)$ due to unlucky pivot selection, though highly improbable.

- **Selection in Worst-Case Linear Time (Deterministic Selection)** → efficient, linear complexity
  Deterministic approach like median of medians. Linear time complexity of $O(n)$ even in worst-case scenario. May have higher overhead compared to randomized but offers deterministic performance guarantees.