* Data Structures:-

We produce a lot of data on daily Basis. May it be transactional data, sales data (or) records we need to efficiently store and organize them in order to achieve optimal storage and processing.

* Algorithms:-

Algorithms are series of tasks (or) steps of instructions to perform mathematical operations.

Big-O:-

Generally, Processing data in different Machines result in different processing time & Storage Buffer. We need a universal way of quantifying the processing time. So, we choose 'n' the total number of data points in the input.

Note:-

By now, we have achieved very fast processing power in terms of Computing. More time Consumes in traversal of data from Memory to Compute and vice versa.

Algorithms used:-

1. Selection Sort
2. Bubble Sort
3. Insertion Sort
4. Merge Sort
5. Quick sort
6. Binary Search
7. Heapsort.

Data structures Used:-

1. Union Find
2. Stack
3. Queue
4. Linked List
5. Double Linked List
6. Binary Search Tree.
7. Heap.

* Sorting Algorithms:-

Why we need Sorting Algorithms?

Assume a book with page numbers, it is always easy to search a specific page in the Book with help of index & page numbers. Sorting always leads to better searching in the input.

Worst Case :- $O(n^2)$

Best Case :- $O(n \log n)$.

① Selection Sort:-

$O(n^2)$ Algorithm.

Consider Array :-

0	1	2	3
5	4	1	2

n = length(Array)

It compares each element with elements in right and swaps elements whenever it finds elements less than it.

The steps are:-

5 4 1 2 (swap)

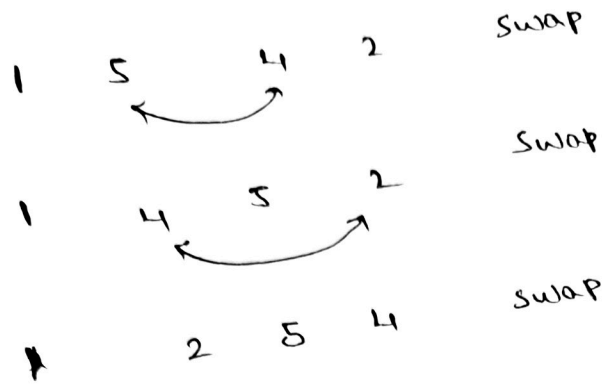
for index 0:-

4 5 1 2 (swap)

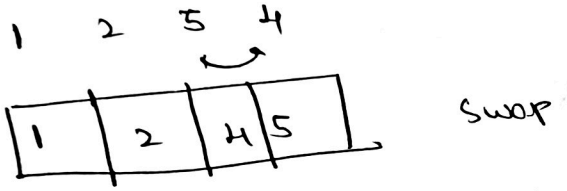
4 5 1 2 (swap)

1 5 4 2 No swap

for index-1:-



for index-2.

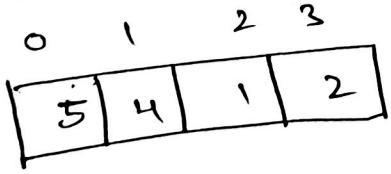


Thus, we achieved a sorted List from selection sort.

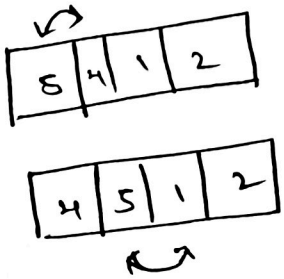
② Bubble Sort:-

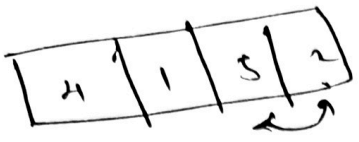
O(n²) Algorithm.

It compares elements side by side & swaps immediately if the next element is smaller than current element.

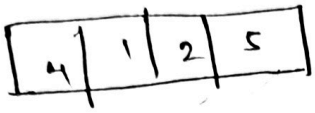


The steps are:- (1st Step).

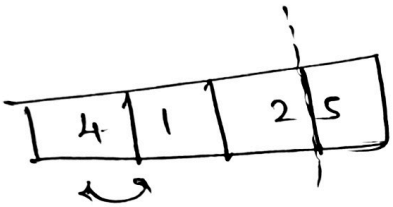




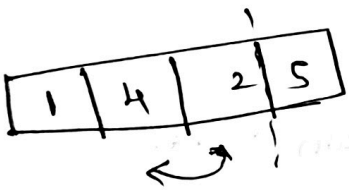
swap



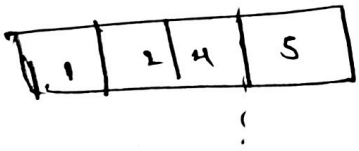
Step-(2) :-



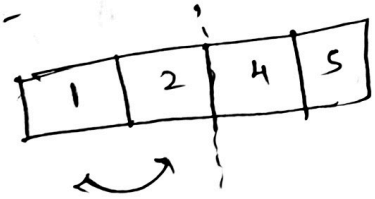
swap



swap



Step-3:-



NO swap.

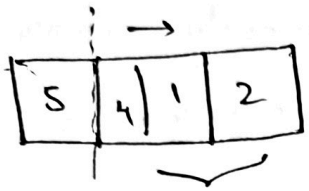
Sorted Array.

Bubble sort swaps neighboring i_s, $(i+1)$ elements with current element if it's high.

③

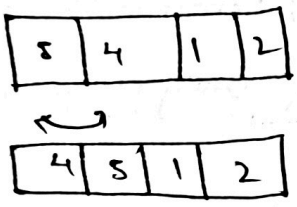
Insertion Sort :-

$O(n^2)$ Algorithm.

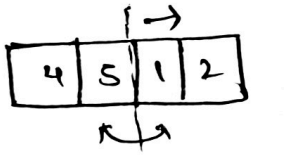


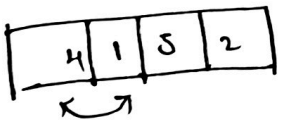
Note:-
Insertion sort works like all the elements in the before part of current index need to be sorted.

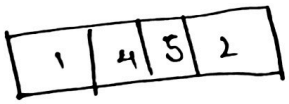
Step-1 :-



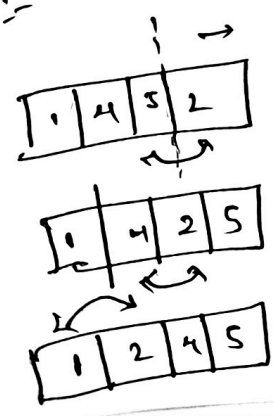
Step-2 :- All elements before index 2 are sorted.







Step-3 :- All elements before index 3 are sorted.

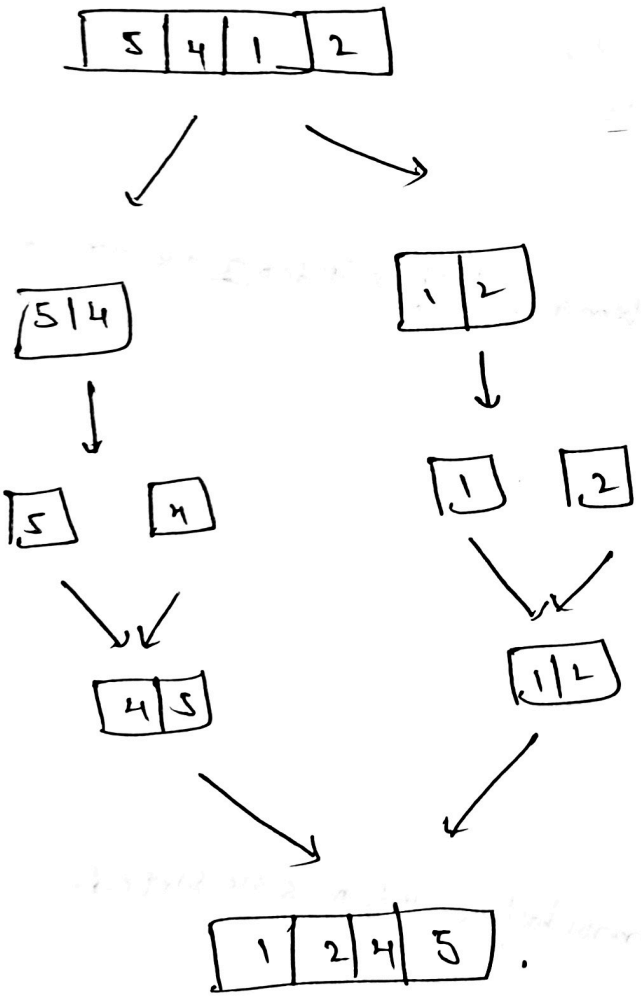


(4) Merge sort:-

O(n log n) Algorithm.

Merge sort mainly works because we can linearly merge two arrays which are already sorted. So n for merging and log n for the number of merges.

So,



First we will split all elements to size of 1 & then combine them using Merge Function.

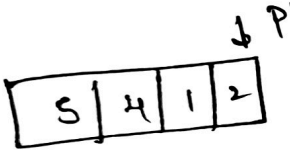
(5)

Quicksort:-

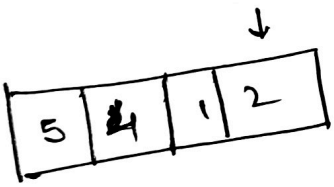
O(n log n)

In Quick sort we have an element called pivot from the array.

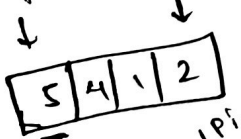
It can be either start or end indexed element. Once pivot is selected we will place it in the correct index that it should belong and the left and right side of the pivot are also placed in correct indexes recursively. Theoretically its O(n log n) Algorithm.



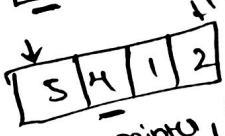
Step 1:-



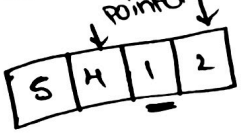
The better way to do is have a pointer, that count all elements which are lesser than the pivot. the pointer will show where the element from pivot should be present exactly. When we find some element < pivot we can swap the pointer with the current element.



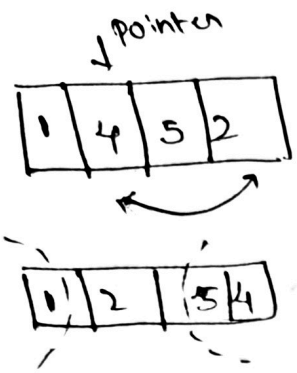
5 > 2 No increment in pointer.



4 > 2 No increment in pointer.



1 < 2 increment the pointer. & swap 1 with 5

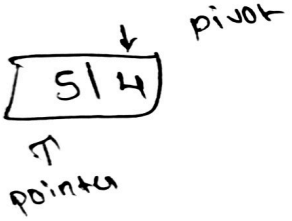


swap (pointer, pivot)

Left part & Right Part need to be solved recursively.

Left part is already sorted.

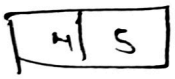
Right Part :-

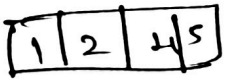


5 > 4, no pointer increase.

And pointer didn't change.

swap (pivot, pointer)



So, Combinedly the result is 

. This is an inplace algorithm.

⑥

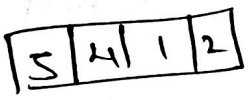
Binary Search:-

Complexity O(logn)

Once, we sort any array we can search any element by logn steps.

So,

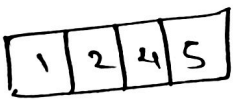
Consider our array.



Before sorted,

if we want to find 2 we should go linearly & take O(n) steps

Once we sort,

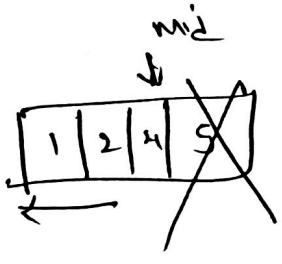


If we want to search 2, then we need to check. Mid = $\frac{0+\text{len(array)}}{2}$

if Arr[mid] > 2. We will cut the right side part of array and look in left half only similarly we recursively explore the left side of array. If Arr[mid] = Val we will return the element

So,

1st Step, Mid = $\frac{0+4}{2} = 2$.



Now left point is [1] [2],

So,

now mid = $\frac{0+2}{2}$ → (1)

If the element is mid return element. It took (2) steps to explore. where as linearly it would have been 4 steps to find (2) in the given array.

Note:-
we will Discuss about Heap DataStructure & Heap sort at the end.

Data Structures:-

① Union Find:-

It has two key methods ① Union : O(1) complexity
② find : O(1) complexity

Understand this way a road exists between "A" & "B" Then "B" is connected to "C". Then "A" is connected to "C".

Union operation performs addition of a connection. find operation checks if there exists a specific connection.

we use dictionaries to store the values.

For example if we initiate a list of 3 cities,

list = ['Ongole', 'Guntur', 'Vijayawada']

we need to give keys as their index at start.

dict = {"Ongole": 0, "Guntur": 1, "Vijayawada": 2}.

So, if we make a connection between Ongole & Guntur we need to match their keys. Union("Ongole", "Guntur").

dict = {"Ongole": 1, "Guntur": 1, "Vijayawada": 2}

~~Union(Guntur, Vijayawada)~~

Union (Vijayawada, Guntur)

dict = {"Ongole": 1, "Guntur": 1, 'Vijayawada': 1}.

Now if we do a find on, find(Ongole, Vijayawada)

it will result Connection exists.

Note:-

This data structure is coded in sequence in mind (a) if the order of cities in Union function changes we will get a different dictionary & Connection won't work properly.

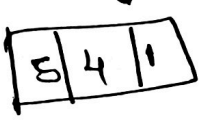
② Stack:-

Stack is a Data Structure where recent elements pops first when deleting the record even while inserting. These operations are called Push & POP.

① Push.

② POP.

Assume we have this array

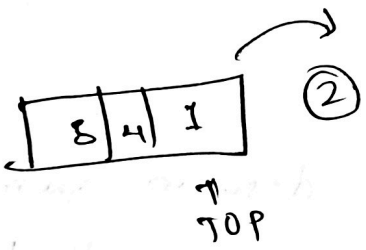


top is a pointer, which we will use as the index where the new element is added at index 1.

The top will get (+1) updated after we add this element.

When we pop the element :-

The element at TOP index will be removed & TOP will be decremented (-1).



Stack is a LIFO Data Structure.

Queue:-

Queue is a data structure, where the elements which are added first in Queue get out first when an element need to be deleted. Whereas the elements adding on Queue get added on end of the list. It has two operations.

① Enqueue O(1)

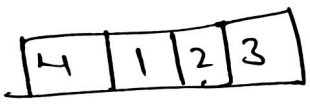
② Dequeue O(1).

Queue:-

Assume the Queue.

If we perform Queue.dequeue.

If we perform enqueue (3)



(3) Linked List:-

Linked list is a data structure where the data in the memory are connected by links & We need to traverse through the list when we need to find an element in a linked list. Even the deleting operation needs traversing through the linked list.

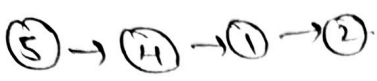
(1) Add - O(n)

(2) Traverse - O(n)

(3) Delete - O(n).

(1)

Adding (2) will end up in creating a new link.



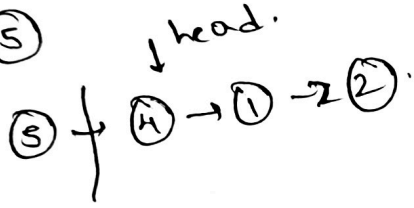
(2) Traverse:-

This will print the elements in insertion order.

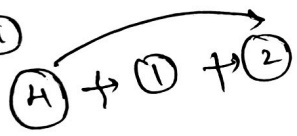
5, 4, 1, 2.

Delete:-

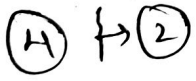
Delete (5)



Delete (1)



Delete (2)



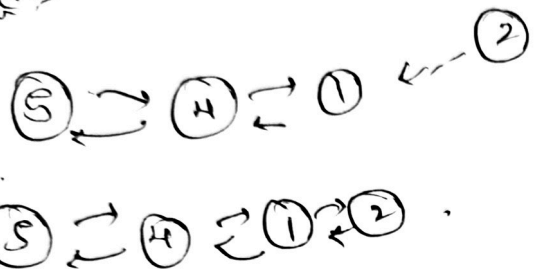
(4) Double Linked list:-

A Double Linked list is similar to Linked list & we can traverse back and forth with double links.

Binary Search Tree :-

(1) Add
(2) Traverse
(3) Delete.

(1) Add :-



(2) Traverse

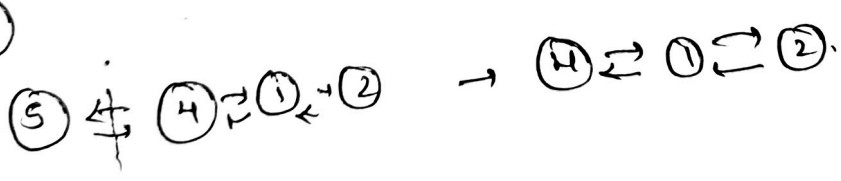
5, 4, 1, 2

Reverse Traverse

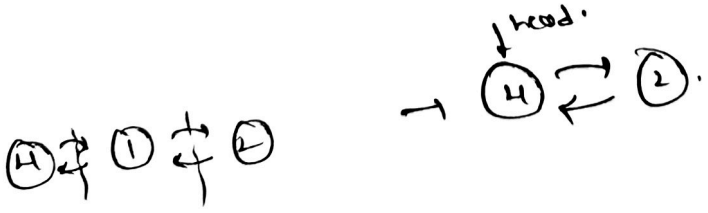
2, 1, 4, 5.

(3) Delete :-

(5)



(1)



(2)

Binary Search Tree :-

Binary Search Tree is a datastructure in which data is Organized in a tree fashioned Manner & all the elements to the left of the root is less than root & all the elements on the right are greater than root. And each subtree other than root follows the rules like the elements on left should be less than root & right greater than root.

Adding elements ① Add :- O(logn)

② preOrder - O(n)

③ postOrder - O(n)

④ inOrder - O(n).

⑤ Delete - O(logn)

⑥ Max value - O(logn)

⑦ Min value - O(logn).

Adding elements:

5, 3, 7, 2, 4, 6, 8.

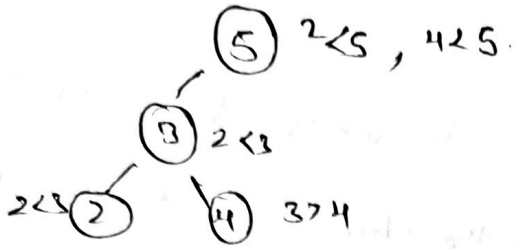
① 5 <5 left

3

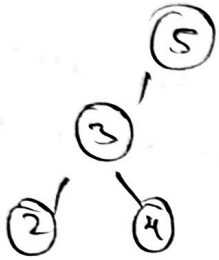
② 5 7>5 right.

3 7

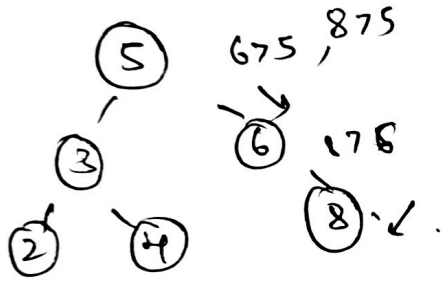
(3)



(4)



(5)



(2) Pre Order:-

The Order of elements are printed in the Order.

* root left right.

* 5 3 2 4 6 8.

(3) Post Order:-

The Order of elements are printed in Order.

* left right root.

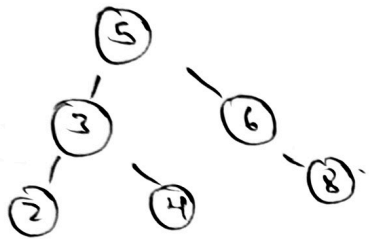
* 2 4 3 8 6 5.

(A) InOrder:

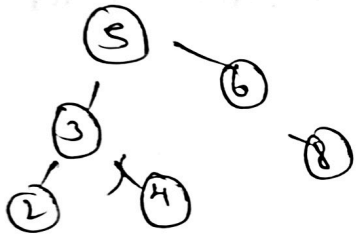
left root right.

2 3 4 5 6 8.

(5) Delete :-



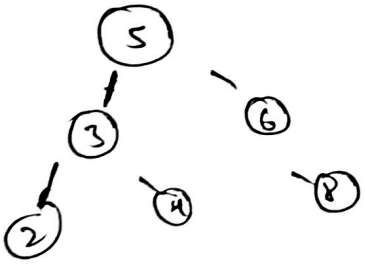
Case-1 :- Deleting a leaf



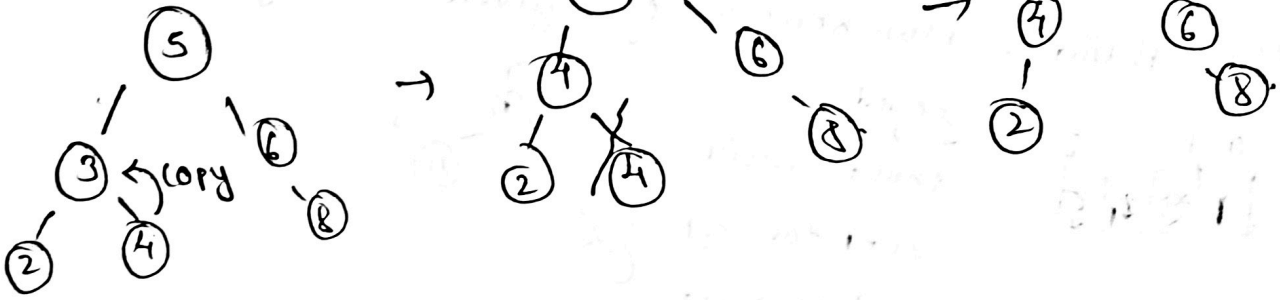
Delete (4)

Adding (4) Back to get the correct working.

Deleting a parent of leaf Case-2

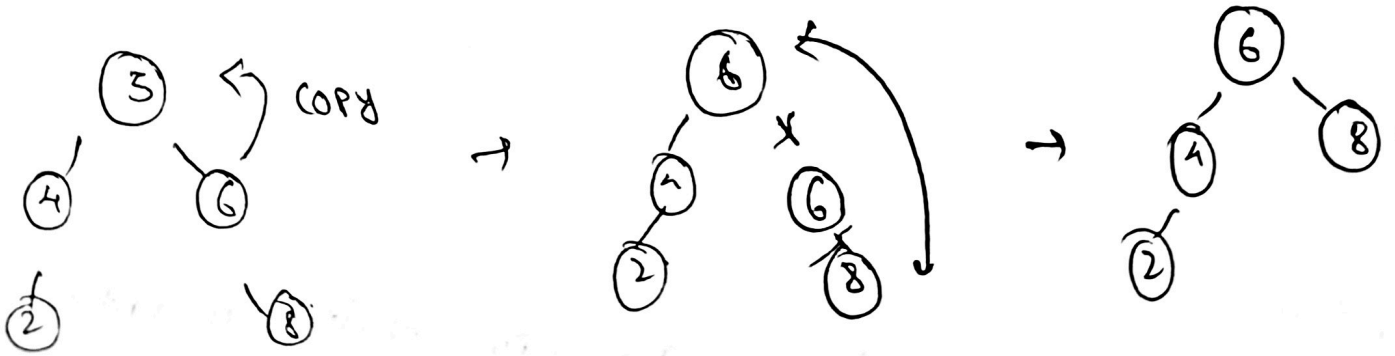


Delete (3)



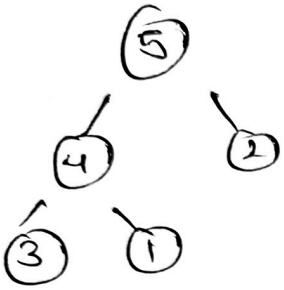
Case-3

Delete - (5).



(7) Heap & Heapsort:-

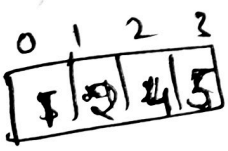
Heap is a Data structure where the root of the heap contains the maximum element & this applies to subtree of the heap.



As we this is a Max heap.

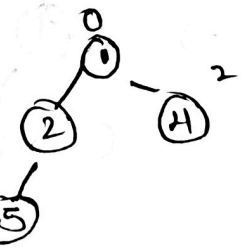
(1) Heapify O(logn)
(2) Build Max heap. O(N)
(3) Heap sort. O(N log N).

→ step (1) Build a Max heap out of given Array.



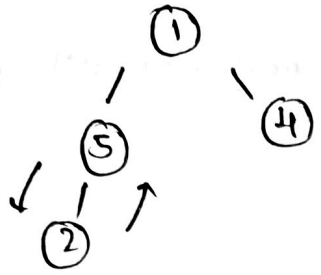
see indexes they represent

2i+1 for left
2i+2 for right.

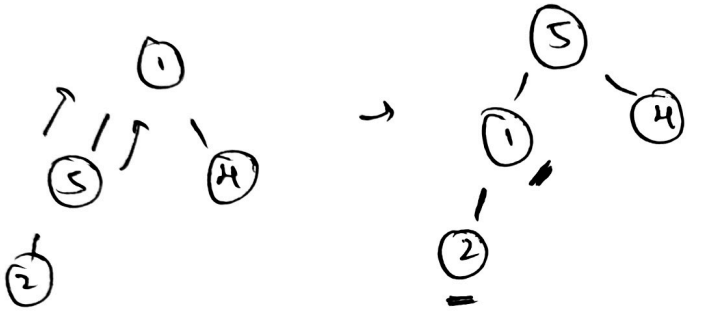


We should start from leaves:-

Q:

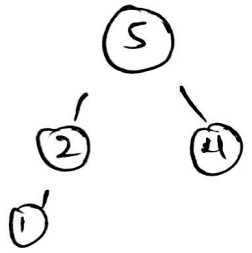


① If the leaf value is greater than root change it.



→ It still not a Max heap

Heapify (left).



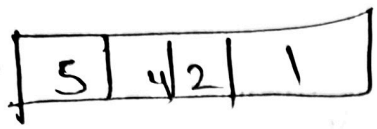
. Max heap.

Once Max heap is build removal of root and reheapify will only take (Ogn order.

(32-16+1) (32-16+1-1)

③ Heap sort.

Similarly we will pop the root & reheapify the elements until we get a sorted array in reverse order in our case.



Heap sort use heapify to continue for next elements going forward.