SORTING Algorithms

By
Kapil Yadav

Insertion Sort

Merge Sort

Selection Sort

Quick Sort

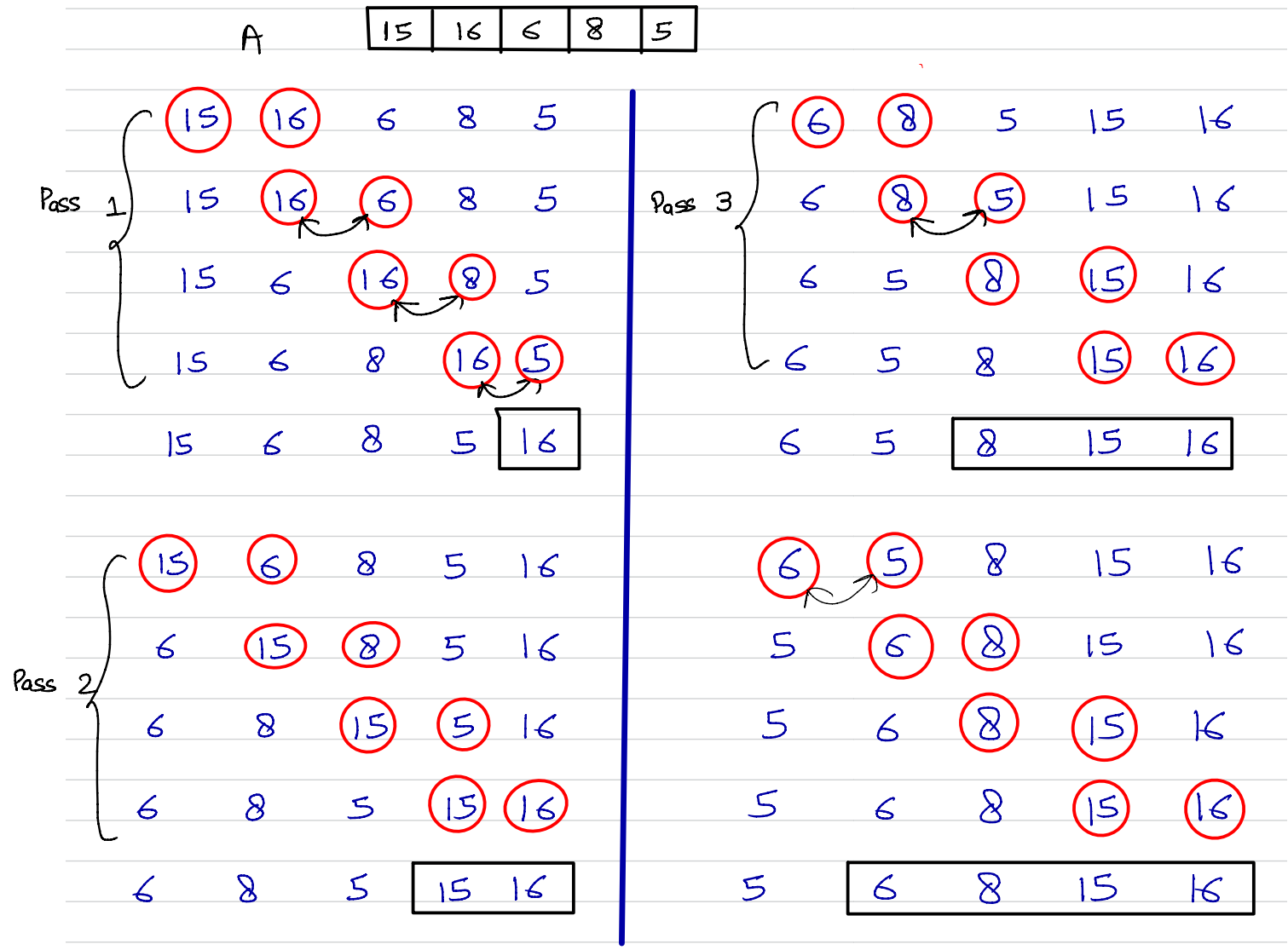
Heap Sort

Bubble Sort

Bubble Sort:

Bubble Sort is a comparison based Sorting Algorithm.
It works by checking its adjacent element whether, it is in sorted order or not.
It is an Inplace Sorting algorithm as we don't need any extra data structure while sorting.
It is stable (the sequence of repeating elements does not change).

Working of Bubble Sort:



No of Required Passes = (n-1)

Total Comparisons = (n-1) + (n-2) + (n-3) + ……1 = ((n-1) * n)/2 = $O(n^2)$ { Every Case}

Total Swaps :

minimum = 0 {elements are in sorted order}

maximum = (n-1) + (n-2) + (n-3) + ……1 = ((n-1) * n)/2 = $O(n^2)$

Time Complexity = maximum(Comparison, Swaps)

= **$O(n^2)$** {Every Case}

Space Complexity : O(1)

```cpp
#include <iostream>
using namespace std;

void bubblesort(int arr[],int n){
    for(int i=0;i<n-1;i++){
        for(int j=0;j<n-i-1;j++){
            if(arr[j]>arr[j+1])
            {
                int temp = arr[j];
                arr[j] = arr[j+1];
                arr[j+1]= temp;
            }
        }
    }
}

void printarr(int arr[],int n){
    for(int i=0;i<n;i++){
        cout<<arr[i];
    }
    cout<<endl;
}

int main() {
    int arr[] = {9,8,7,6,5,4,3,2,1};
    int n = sizeof(arr)/sizeof(int);
    printarr(arr,n);
    bubblesort(arr,n);
    printarr(arr,n);
}

```

Further Optimization:

The Time Complexity of Bubble Sort can be optimized for Best Case.i.e When all the elements of array are in sorted order, then there will be no swaps, So we can use it to identify whether our array is sorted or not.

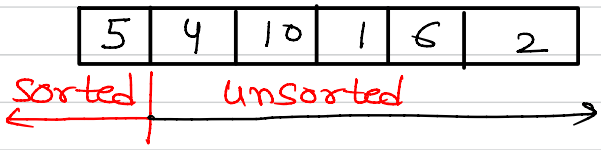
We can use a flag variable with initial value = 0,
If in any pass, there is no swap, just break the loop and we will get sorted elements.

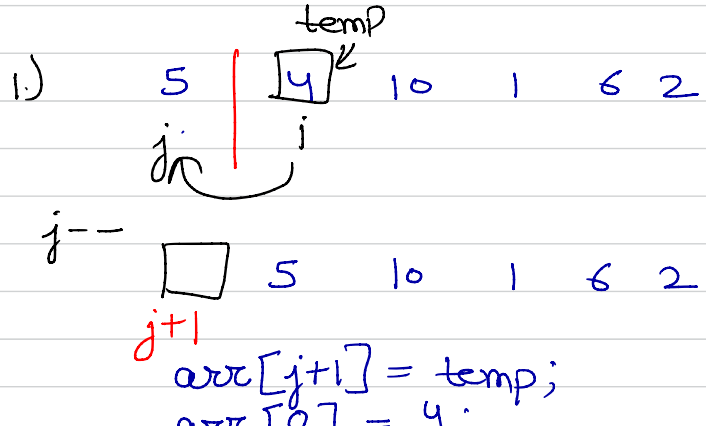
TC: O(N) {Best Case}, $O(n^2)$ in {Average, Worst Case}

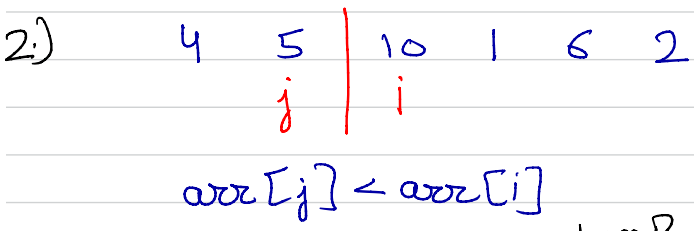
```cpp
#include <iostream>
using namespace std;

void bubbleort(int arr[],int n){
    for(int i=0;i<n-1;i++){
        bool flag = 0;
        for(int j=0;j<n-i-1;j++){
            if(arr[j+1]<arr[j]){
                int temp = arr[j];
                arr[j] = arr[j+1];
                arr[j+1]= temp;
                flag = 1;
            }
        }
        if(flag==0)
        break;
    }
}

void printarr(int arr[],int n){
    for(int i=0;i<n;i++){
        cout<<arr[i];
    }
    cout<<endl;
}

int main() {
    int arr[] = {1,2,3,4,5};
    int n = sizeof(arr)/sizeof(int);
    bubbleort(arr,n);
    printarr(arr,n);
    }
```

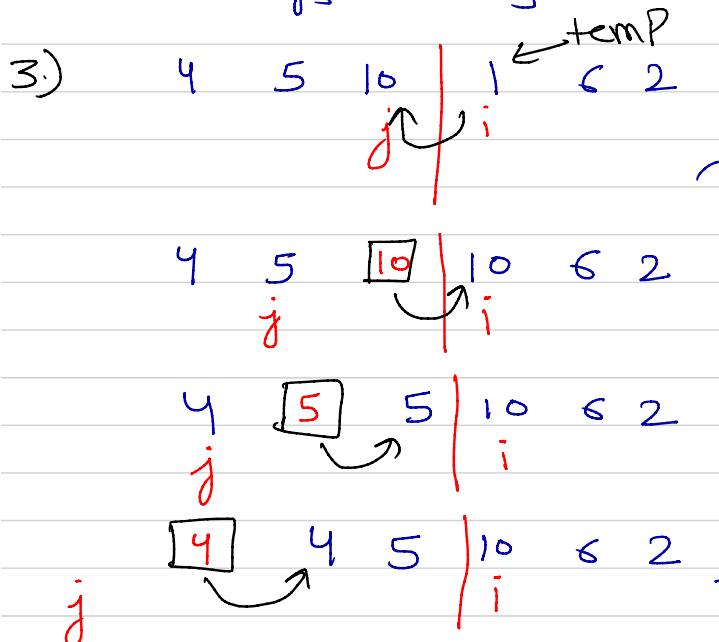
Insertion Sort:

Insertion Sort is another Comparison Based Sorting Algorithm.
It works as similar as the way we sort playing cards.i.e we have unsorted and sorted portions of the cards, and we use to place the unsorted card in its correct position.
Insertion Sort works similar to that.
It is also inplace Sorting algorithm.
It is Stable Sorting Algorithm.
It can be useful when number of elements are very less.
It can be also useful when elements are in almost sorted order.



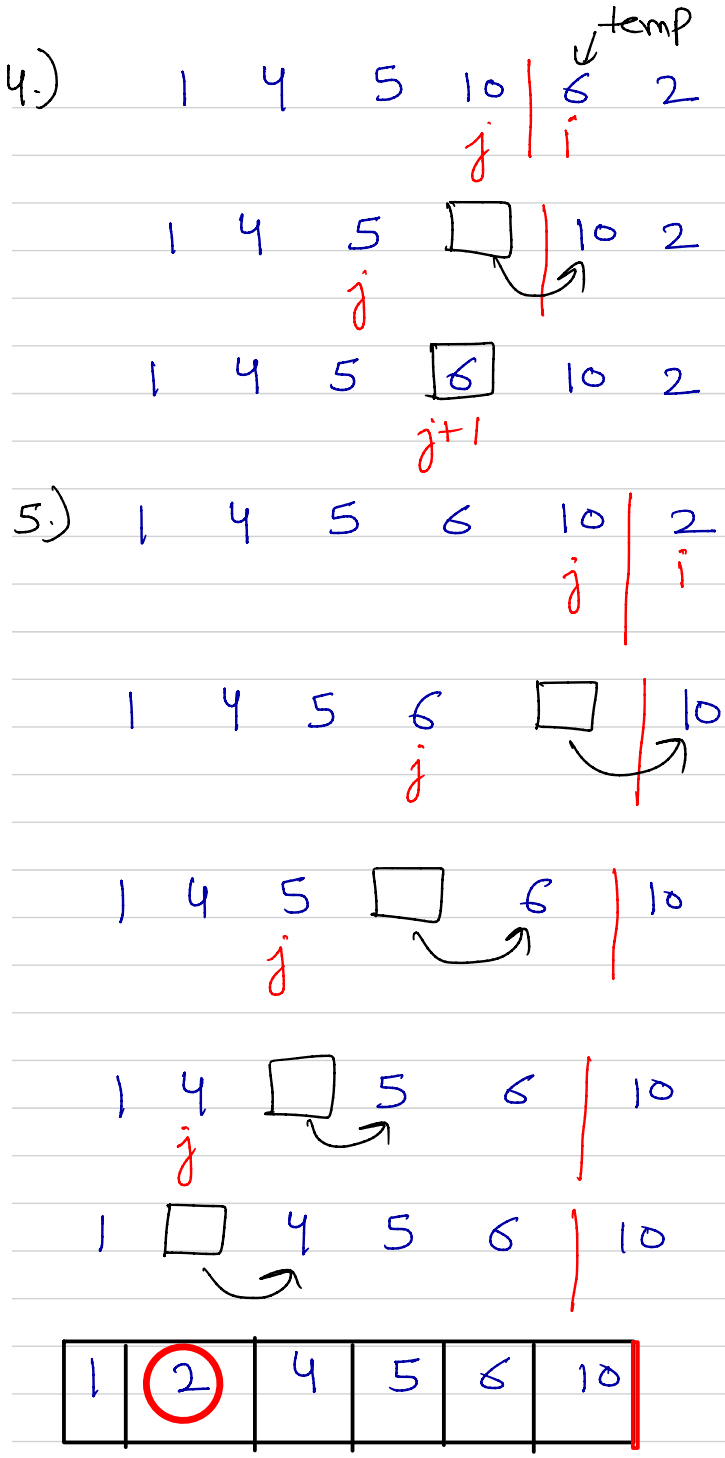






inner loop will run untill j = 0,
if it runs until j=0, means correct position of current i^{th} element is at 0^{th} index.

□ → represents that current j^{th} index before decreament and its value will be same as previous.



Insertion Sort is useful in Best Case Only : TC = O(N) {Best Case}

Total Comparisons : N-1 in Best Case

If Array is almost Sorted Insertion Sort will take O(N)

No of swaps = No of Inversions.

Worst Case of Insertion Sort can be when elements are in Descending Order.

Total Comparisons,swaps in Worst Case = 1+2+3+4+5+……..n-1 = **$O(n^2)$**

Average Case (Half Ascending + Half Descending) = n/2 BC + n/2 WC = $O(n^2)$

```cpp
#include <iostream>
using namespace std;

void insertionsort(int arr[],int n){
    for(int i=1;i<n;i++){
        int temp = arr[i];
        int j=i-1;
    while(j>=0 && arr[j]>temp){
        arr[j+1]=arr[j];
        j--;
    }
    arr[j+1] = temp;
        }
}


void printarr(int arr[],int n){
    for(int i=0;i<n;i++){
        cout<<arr[i];
    }
    cout<<endl;
}

int main() {
    int arr[] = {9,8,7,6,5,4,3,2,1};
    int n = sizeof(arr)/sizeof(int);
    printarr(arr,n);
    insertionsort(arr,n);
    printarr(arr,n);
}

```

Practice Questions:

Q 1.) No of swaps performed by insertion sort in the given array:

1 2 3 5 0 4 10 8 7

Ans.) 8 swaps.

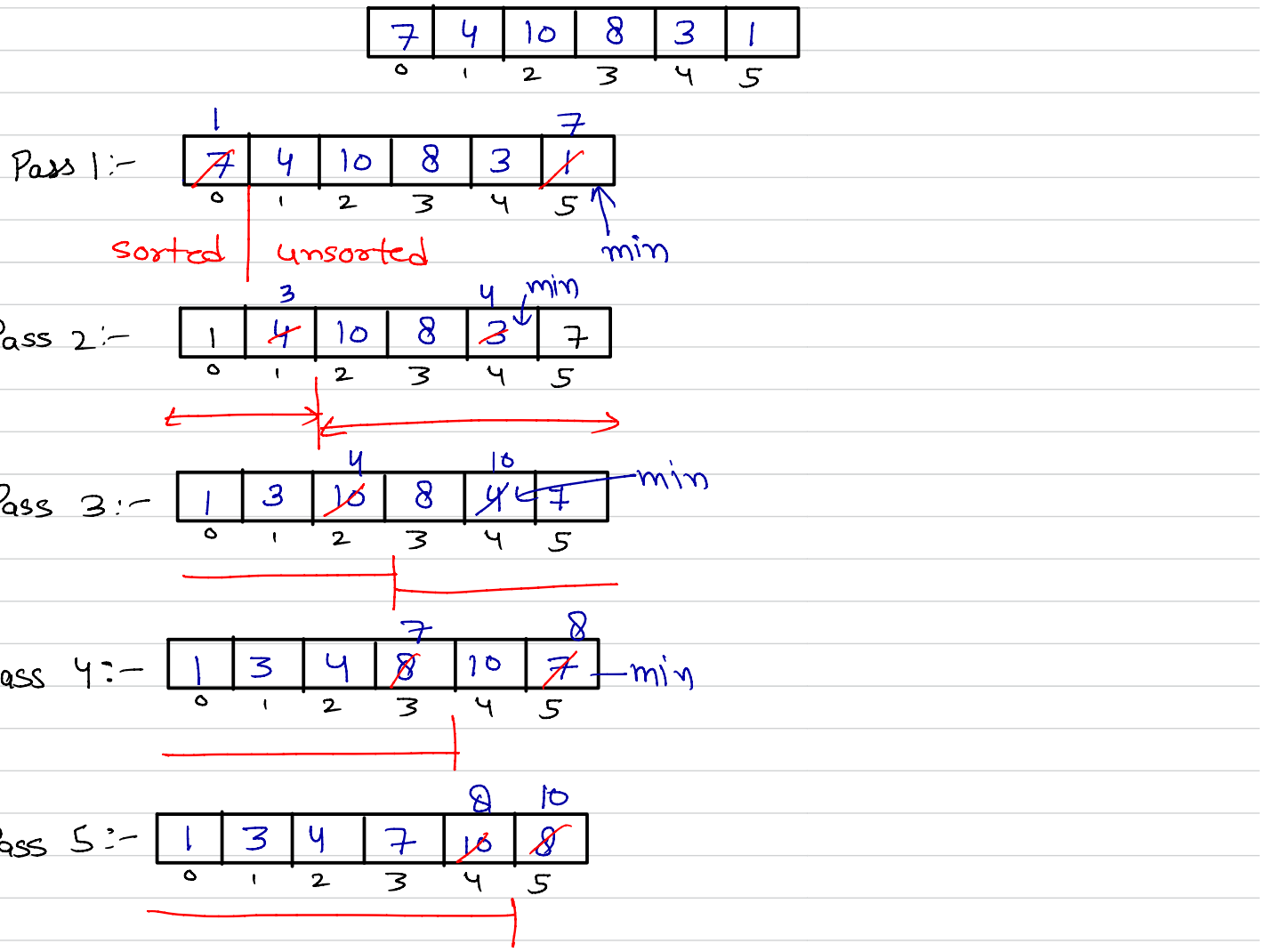
Q 2.) No of comparisons required to perform insertion sort

25 75 95 125 80 5 10

Ans.) 17 comparisons.

Selection Sort:

Selection Sort is another Comparison based Sorting Algorithm.
It also uses the concept of Sorted and Unsorted Subarray.
At each pass, we sort one element by finding minimum element from unsorted array
It is Not Stable as the sequence of repeating element can be change.
It is Inplace as it doesn't require any extra space.



Selection Sort Requires N-1 Passes.
Total Comparisons = ((N-1) * N) /2 = O(n^2) { Every Case},
Total Swaps = N-1 { Every Case}
Time Complexity = O(n^2) {because comparisons are always n^2.}
Selection Sort speciality is No of swaps are N-1 in every case.

```cpp
#include <iostream>
using namespace std;

 void swap(int *a,int *b){
     int temp = *a;
     *a = *b;
     *b = temp;
 }

void selectionsort(int arr[],int n){
    for(int i=0;i<n-1;i++){
        int min =i;
        for(int j=i+1;j<n;j++){
            if(arr[i]>arr[j])
            {
                min = j;
            }
        }
        if(min!=i)
        {
        swap(arr[i],arr[min]);
        }
    }
}

void printarr(int arr[],int n){
    for(int i=0;i<n;i++){
        cout<<arr[i];
    }
    cout<<endl;
}

int main() {
    int arr[] = {9,8,7,6,5,4,3,2,1};
    int n = sizeof(arr)/sizeof(int);
    printarr(arr,n);
    selectionsort(arr,n);
    printarr(arr,n);
}

```

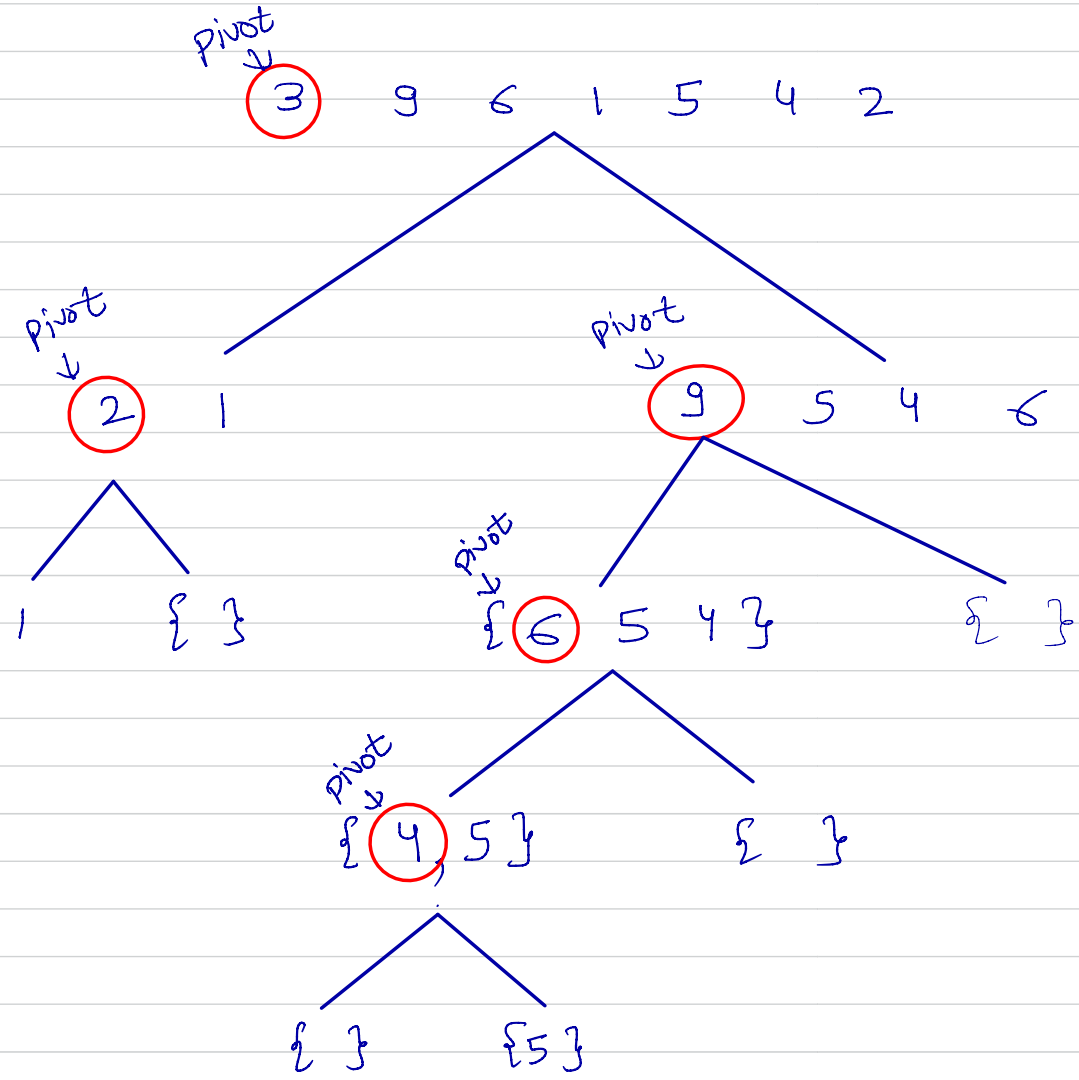
Quick Sort:

It is a Divide and Conquer Algorithm It is also a type of comparison based Sorting Algorithm.
The Partition Algorithm is the heart of Quick Sort Algorithm.
We choose an element as pivot and perform partition algorithm to the pivot algorithm.
The idea to put pivot element at it's correct position, which means the left side elements of pivot elements will be less than or equal to pivot element and the right side elements will be greater than pivot element.
NOTE: Left and Right side elements need not to be in Sorted Order, to Sort them, we again call Quick Sort for both the partitions.
It is considered to be inplace algorithm as it only used recursive space.
It is not Stable.

Pivot Selection can be done in Many ways:

- Always Select first element as pivot
- Always Select last element as pivot
- Pick Random element as pivot
- Pick Median as pivot.

Let's choose first element as pivot

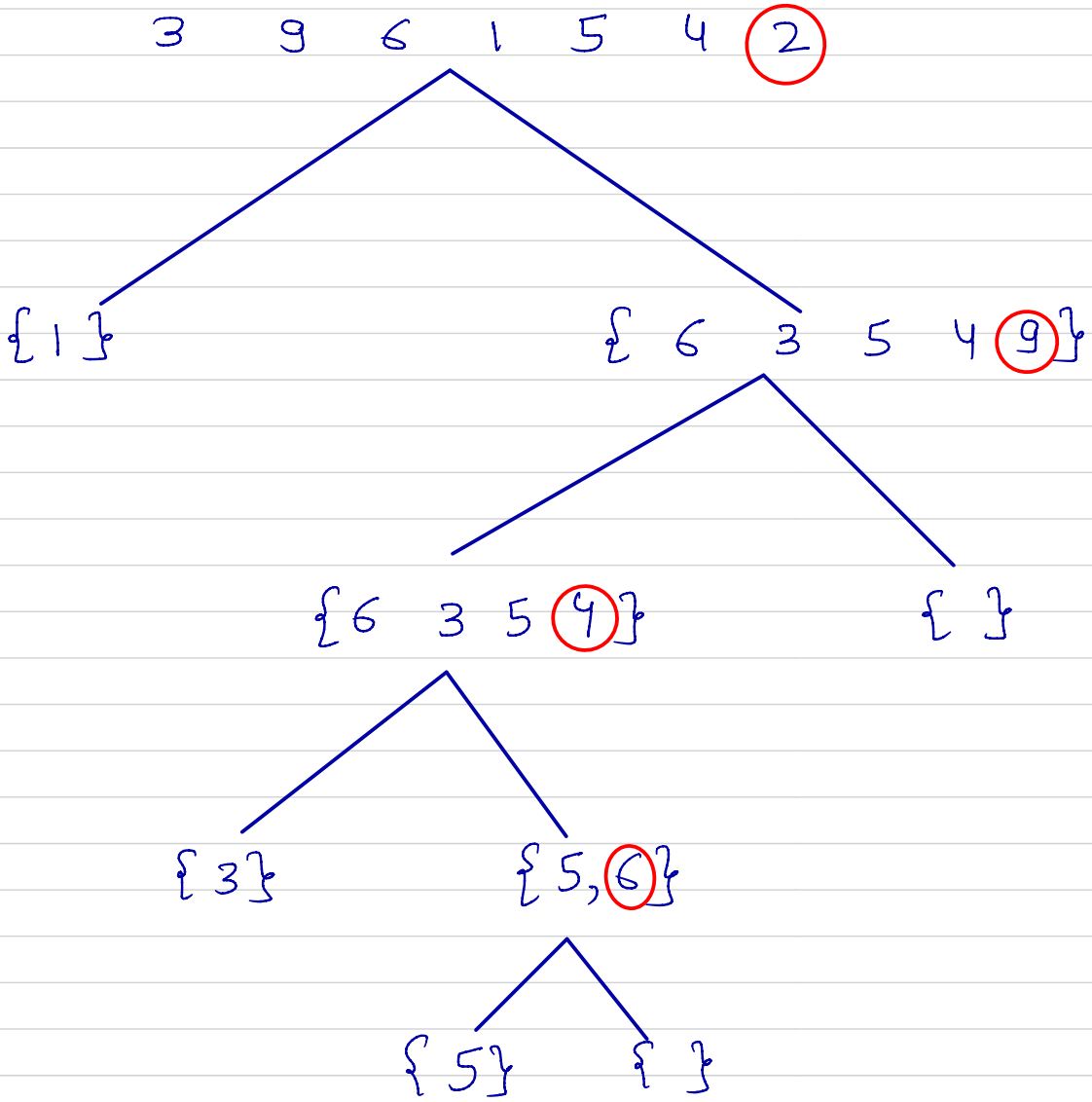


1.) When first element is selected as pivot:

Take i = p = index of first element.
Take second element index as j.
Check if a[j]<= a[p], if yes, increament i, and swap a[j] & a[i].
At the end of the pass swap a[i] and a[p].

2. When last element is selected as pivot:

Take i = index of first element.
Take p = index of last element.
Take j = index of first element, run j loop from j=0 to n-1.
if a[i]<=a[p], swap a[i] and a[j], and then increment i by 1.
At the end, swap a[i] and a[p].



```cpp
#include <iostream>
using namespace std;

void swap(int *a,int *b){
    int temp = *a;
    *a = *b;
    *b = temp;
}
int partition(int arr[],int start, int end){
    int pivot = arr[start];  // take first element as pivot
    int i = start;
    for(int j = start+1;j<=end;j++){
        if(arr[j]<=pivot){
            i++;
            swap(&arr[i],&arr[j]);
        }
    }
        swap(&arr[i],&arr[start]);
        return i;
}


void Quicksort(int arr[],int start,int end){
    if(start >= end){
        return ;
    }

    int k = partition(arr,start,end);  //pivot element
    Quicksort(arr,start,k-1);  //left sort
    Quicksort(arr,k+1,end);   //right sort
}

void printarr(int arr[],int n){
    for(int i=0;i<n;i++){
        cout<<arr[i];
    }
    cout<<endl;
}

int main() {
    int arr[] = {9,8,7,6,5,4,3,2,1};
    int n = sizeof(arr)/sizeof(int);
    printarr(arr,n);
    Quicksort(arr,0,n-1);
    printarr(arr,n);
}

```

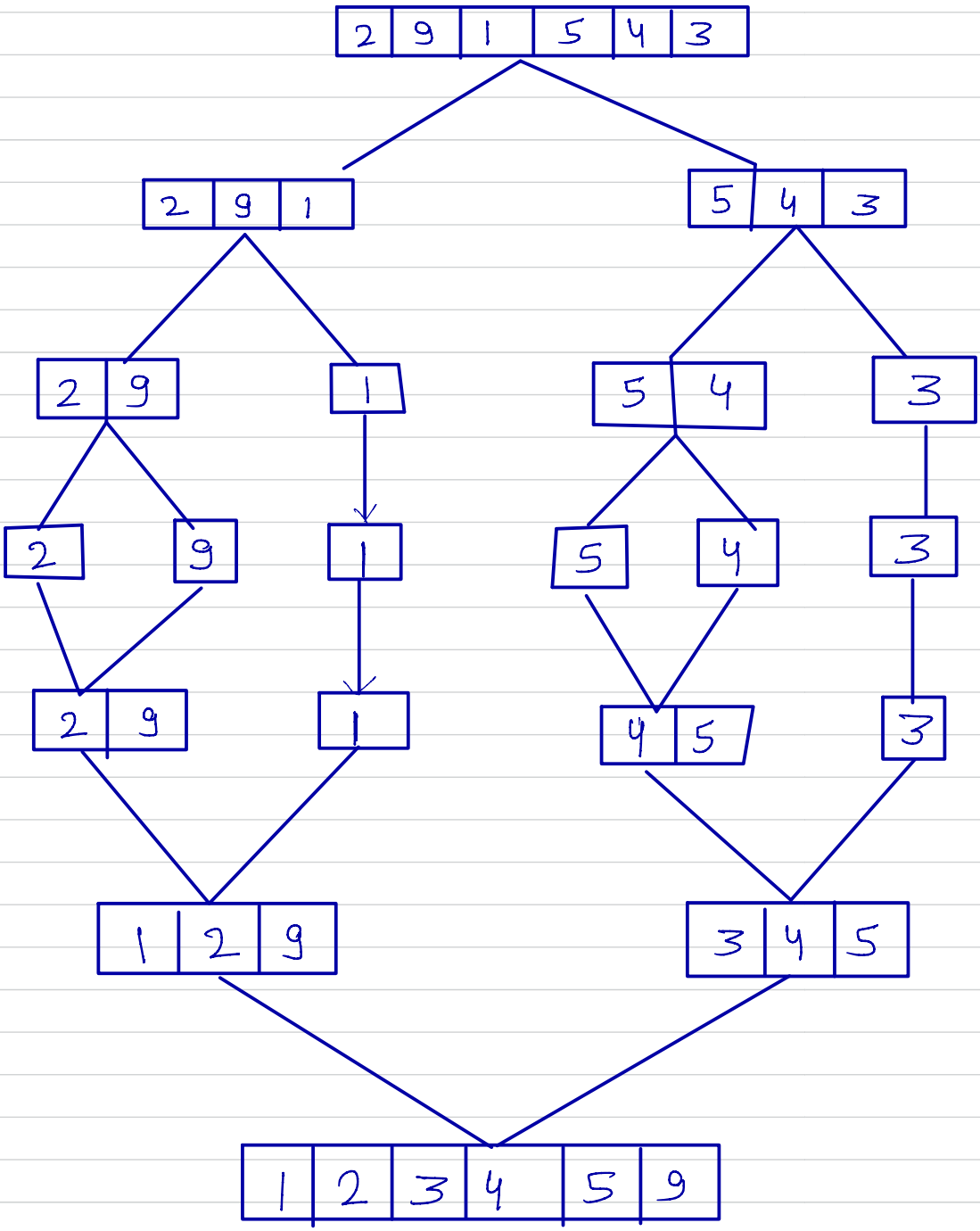
Time Complexity of Quick Sort:

T(n) = T(K-p) + T(q-m) + n + O(1),
where K is the pivot element index after partition, p is the starting index, q is the last index, O(n) is the time required for partition algorithm, O(1) for choosing pivot element.

Best Case = T(n/2) + T(n/2) + n, when after partition, there will be equal elements on both sides.
Worst Case + T(n-1) + n, when after partition, the elements are skewed on one side only.

Merge Sort:

It is another comparison based sorting algorithm.
It is based on the Divide and conquer technique.
It is Outplace algorithm as we need an additional array to store the temp output of the sorted array.
It is Stable Sort.



```cpp
#include <iostream>
using namespace std;

void Merge(int arr[],int start,int mid, int end){
    int i=start;
    int j =0;
    int k = mid+1;
    int b[end-start+1];
    while(i<=mid && k<=end){
        if(arr[i]<=arr[k]){
            b[j]=arr[i];
            i++;
        }
        else {
            b[j] = arr[k];
            k++;
    }
    j++;
    }


    while(i<=mid){
        b[j] = arr[i];
        j++; i++;
    }
    while(k<=end){
        b[j] = arr[k];
        k++;
        j++;
    }
     int h=start;
       for(int i=0;i<end-start+1;i++)
       {
           arr[h]=b[i];
           h++;
       }

}
```

```cpp
void Mergesort(int arr[],int start,int end){
    if(start<end){
    int mid = start+(end-start)/2;
    Mergesort(arr,start, mid);
    Mergesort(arr,mid+1,end);
    Merge(arr,start,mid,end);
    }
}
void printarr(int arr[],int n){
    for(int i=0;i<n;i++){
        cout<<arr[i];
    }
    cout<<endl;
}

int main() {
    int arr[] = {9,8,7,6,5,4,3,2,1};
    int n = sizeof(arr)/sizeof(int);
    printarr(arr,n);
    Mergesort(arr,0,n-1);
    printarr(arr,n);
}
```

→ O(1)
→ T(n/2)
→ T(n/2)
O(n)

Time Complexity of Merge algorithm is : O(N) for outplace,
O(n/2*n/2) = O(n2) for inplace.

Recurence Relation :
1.) for Outplace:

$$T(n) = \begin{cases} O(1) & \text{if } n = 1 \\ O(1) + 2T\left(\frac{n}{2}\right) + n & \text{if } n > 1 \end{cases}$$

TC = O(n log n)

2:) for inplace:

$$T(n) = \begin{cases} O(1) & n=1 \\ O(1) + 2T\left(\frac{n}{2}\right) + \frac{n^2}{4} & n>1 \end{cases}$$

$$TC = O(n^2).$$

Space Complexity :- input + extra

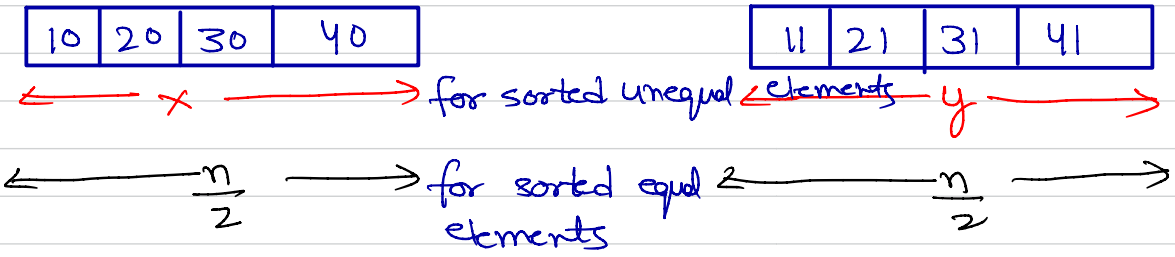
input → n

extra → stack ($\log n$), Merge (n)

$$= O(n).$$

→ Merge Sort is advisable for larger size arrays. for smaller size, Insertion sort is preferable.

① Merge Algorithm worst case in outplace :-



Comparisons	Moves
$(4,4) = 4+4-1$	$(4,4) = 4+4$
$\left(\frac{n}{2},\frac{n}{2}\right) = n-1$	$\left(\frac{n}{2},\frac{n}{2}\right) = n$
$(x,y) = x+y-1$	$(x,y) \Rightarrow x+y$

$TC = \max(\text{Comparisons, moves})$

$= n$ (for equal sorted elements)

(2.) Merge Algorithm Best case outplace :-

Comparison	Moves
$(x,y) = \min(x,y)$	$(x,y) = x+y$
$\left(\frac{n}{2},\frac{n}{2}\right) = \frac{n}{2}$	$\left(\frac{n}{2},\frac{n}{2}\right) = n$

Heap Sort:

Heap Sort is a comparison-based Sorting Technique.
It is based on Binary Heap.
Heap sort is an in-place algorithm.
It is not stable, because the heap operations can change the relative ordering of the elements.
We use heapify method in Heap Sort.
It is similar to selection sort, but we can maximum element from heap in constant time.

1. If Complete Binary tree contains K-levels, Total Nodes = $2^k - 1$
2. Total leaf nodes in CBT = $\lceil n/2 \rceil$ (n = total nodes)
3. Total Internal nodes in CBT = $\lfloor n/2 \rfloor$

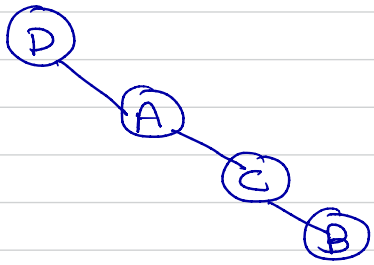
Representation of Binary Tree:

1.Array
2. Linked List

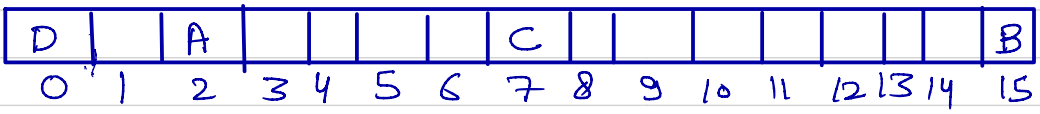
NOTE: If node is stored in 1st place of the array then (Array starting from 1. {different when index changes})
1.) Parent of i = $\lfloor i/2 \rfloor$
2.) Left child of i = 2 * i
3.) Right child of i = (2*1) + 1

NOTE: Array is better for complete and almost complete Binary Tree.
Linked List is better idea for any gap in Nodes.

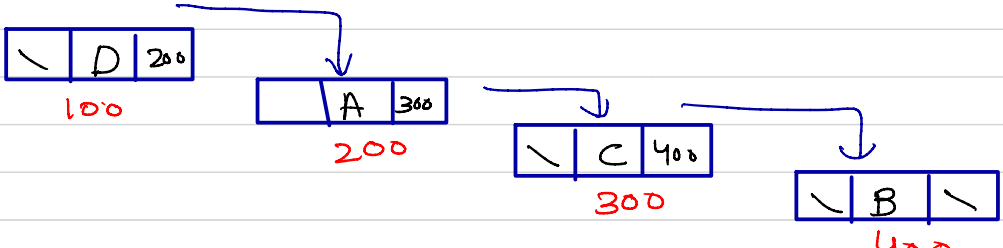
for example :-



(1) Using Array :-



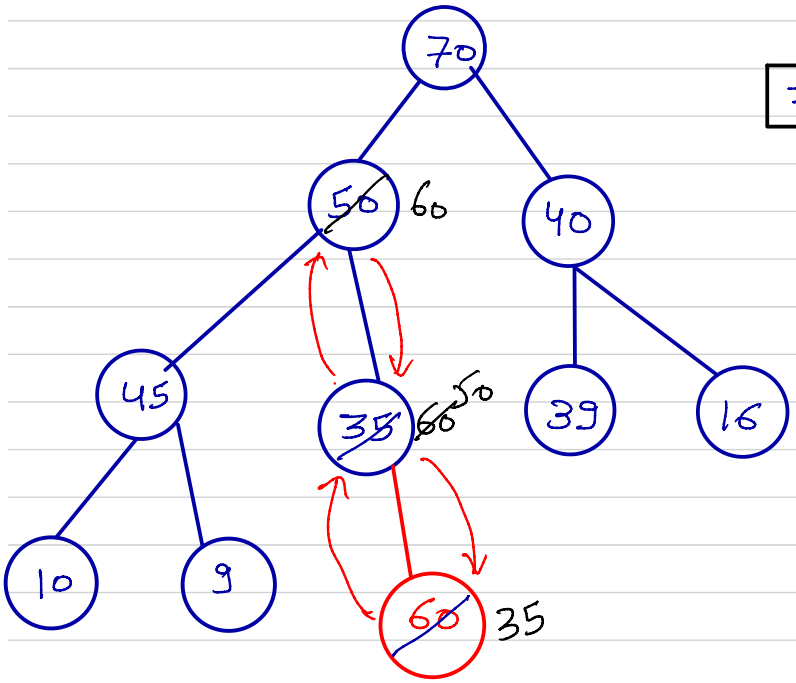
(2) Using Linked List :-

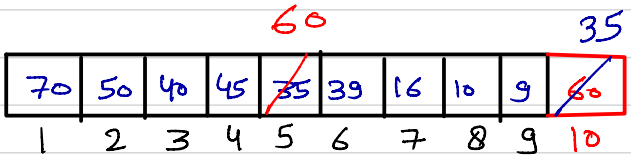


MaxHeap: Root is maximum or equal to children at every level.
for ith max, Total comparison = $i(i-1)/2 = O(i^2)$.

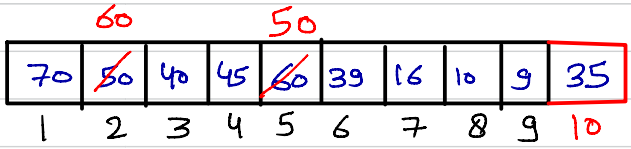
In Bubble Sort, every pass is costly O(n), but in heap sort, every comparison cost is O(1).

Insertion in MaxHeap (Bottom to top Approach)**:**

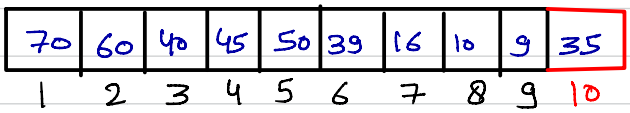




parent of 10 = $\lfloor \frac{i}{2} \rfloor = 5$.



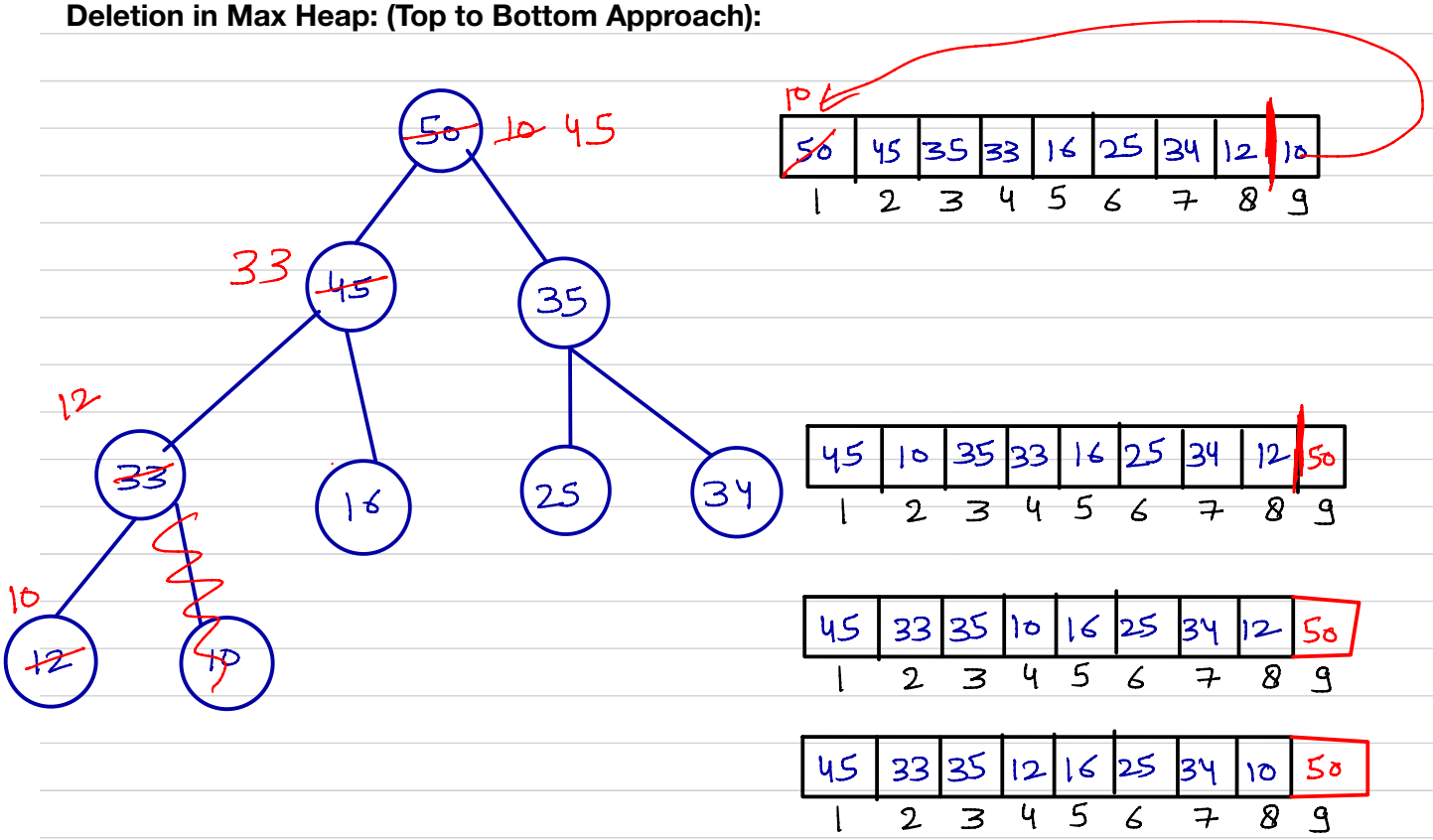
parent of 5 = $\lfloor \frac{5}{2} \rfloor = 2$



parent of 2 is = $\lfloor \frac{2}{2} \rfloor = 1$.

NOTE :- Inserting and Deleting an element into maxheap, which already contains n elements will take order of logn.

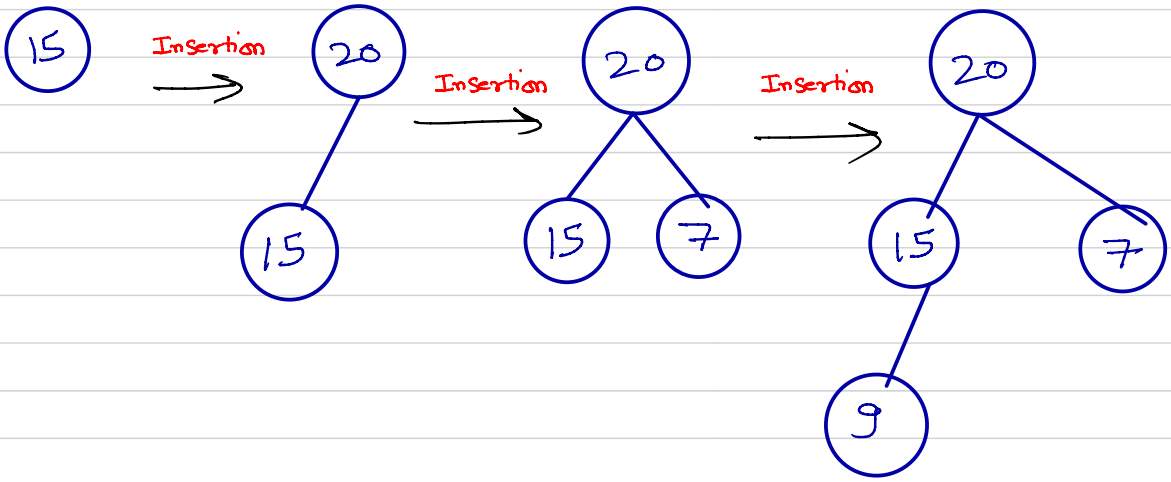
Deletion in Max Heap: (Top to Bottom Approach):

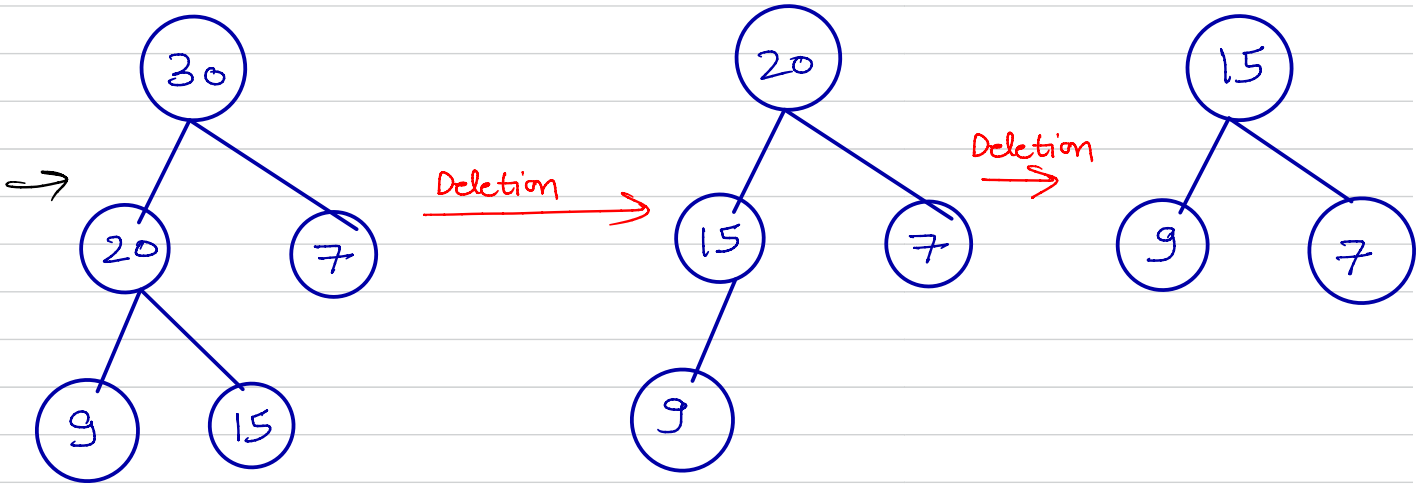


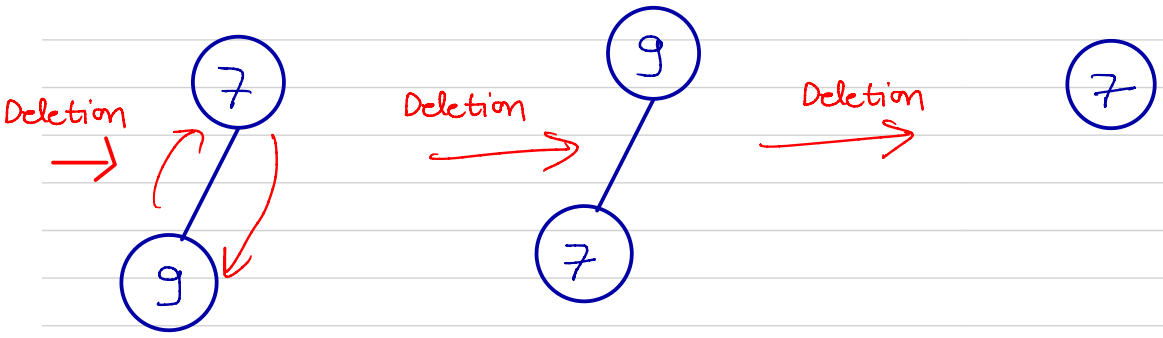
NOTE: To delete an element , we can simply put the deleted element at last place, and reduce the loop to one place (from 1-8). As soon as we deleted the element, we will get the elements in ascending order (because we are placing max element at last place.)

Heap Sort:

	1	2	3	4	5
A	15	20	7	9	30



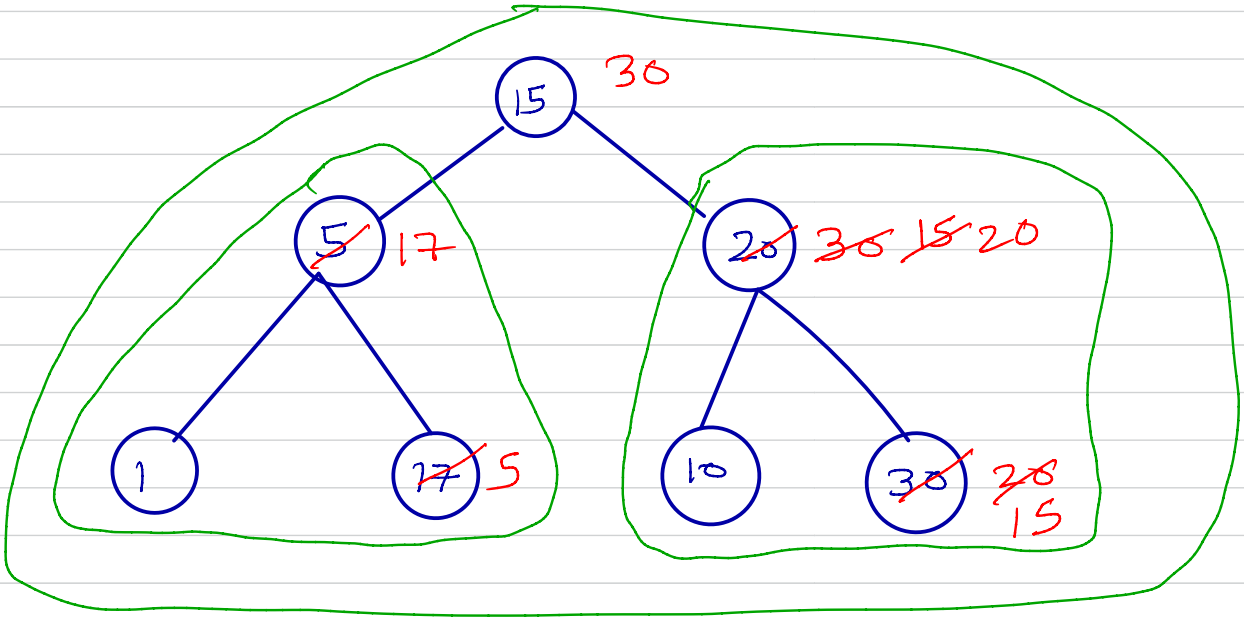




For Insertion in a max heap or min heap, O(logn) times requires for an single element.
for n elements = O(nlogn).
for Deletion in a max or min heap, O(logn) times require for an single element.
for deletion of n elements = O(nlogn)
For heap sort, creation and deletion requires for n elements = O(2nlogn).

NOTE: This method takes O(nlogn) to create a max heap, but there is another method, which is heapify method and it takes O(n) to create a Build Heap.

Heapify Method: We apply Heapify Method for non-Leaf Nodes.




```
1
2 void maxheapify(int arr[], int N, int i){
3     int largest = i;
4     int left = 2 * i + 1;
5     int right = 2 * i + 2;
6     if (left < N && arr[left] > arr[largest])
7         largest = left;
8
9     if (right < N && arr[right] > arr[largest])
10        largest = right;
11
12    if (largest != i) {
13        swap(&arr[i], &arr[largest]);
14        maxheapify(arr, N, largest);
15    }
16 }
```

Heapify Method takes O(logn) time

```c
void heapSort(int arr[], int N)
{
    for (int i = N / 2 - 1; i >= 0; i--)
        maxheapify(arr, N, i);
    for (int i = N - 1; i >= 0; i--) {
        swap(&arr[0], &arr[i]);
        maxheapify(arr, i, 0);
    }
}
```

Build max heap → O(n)

Deletion

$TC = O(n) + O(n \log n)$

↓

Build max heap

```c
#include <stdio.h>
void swap(int* a, int* b)
{   int temp = *a;
    *a = *b;
    *b = temp;
}

void maxheapify(int arr[], int N, int i){
    int largest = i;
    int left = 2 * i + 1;
    int right = 2 * i + 2;
    if (left < N && arr[left] > arr[largest])
        largest = left;

    if (right < N && arr[right] > arr[largest])
        largest = right;

    if (largest != i) {
        swap(&arr[i], &arr[largest]);
        maxheapify(arr, N, largest);
    }
}


void heapSort(int arr[], int N)
{
    for (int i = N / 2 - 1; i >= 0; i--)
        maxheapify(arr, N, i);
    for (int i = N - 1; i >= 0; i--) {
        swap(&arr[0], &arr[i]);
        maxheapify(arr, i, 0);
    }
}

void printArray(int arr[], int N)
{
    for (int i = 0; i < N; i++)
        printf("%d ", arr[i]);
    printf("\n");
}

int main()
{
    int arr[] = {15,5,20,1,17,10,30};
    int N = sizeof(arr) / sizeof(arr[0]);

    heapSort(arr, N);
    printf("Sorted array is \n");
    printArray(arr, N);
}
```