UNIT-5

Syllabus: Priority Queue: Definition, operation and their time complexities.
Sorting:- Heap Sort, Mergesort, Radix Sort

Priority Queue:- A queue in which we can insert/delete item from any position depending on some property is called "Priority Queue".

Consider the elements:

Element	priority
A	7
B	4
C	2
D	1
E	3

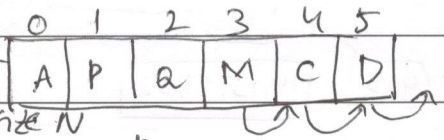
Here lower the number ⇒ Higher is the priority
i.e 1, 2, 3, 4, 7 ⇒ D, C, E, B, A

Types of Priority Queue (PQ): 1 — Ascending PQ (lower no is given Highest Priority)
2 — Descending PQ (Higher no is given Highest Priority)

Implementation of PQ:-
1) Using Arrays
2) using linked list
3) Heap

1. Using Arrays:-

Insertion of an element at a specified position:
To insert an element x at position i in an array A of size N

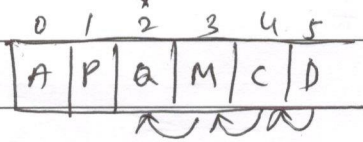


x at position: 3 → Position, element, size of array

Algorithm: array_insert(A[], i, x, N)
BEGIN
FOR j = N-1 to i-1 (j--) (Step-1)
A[j+1] = A[j]
A[i-1] = x
N = N+1
END

O(N) shifts are required in worst case
Ω(1) shift in best case
Θ(N) Average case

Delete an element from an array:-



Algorithm: array_del(A[], N, i)
BEGIN
x = a[i-1]
For j = i TO N-1
A[j-1] = A[j]
N = N-1
Return x
END

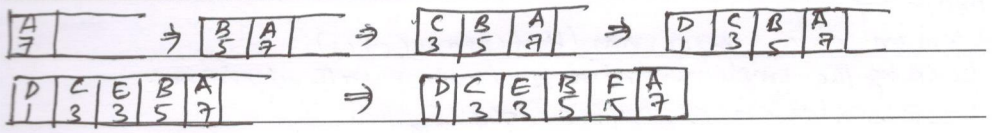
O(N) no of shifts required in worst case
Ω(1) best case (Removal of last element)
Θ(N) Average case.

Akhilesh Kumar Srivastava

Insertion in Priority Queue:-

Consider the following elements with the corresponding priorities

A B C D E F
7, 5, 3, 1, 3, 5



Algorithm for insertion in PQ

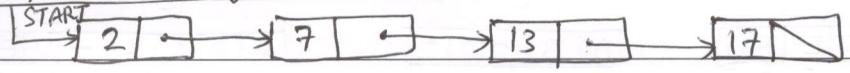
```
PQ_insert (A[], N, PRN)
BEGIN
    i=0
    while PRN >= A[i]
          i++
    array_insert(A[], N, i+1, PRN)
END
```

Algorithm for deletion in PQ.

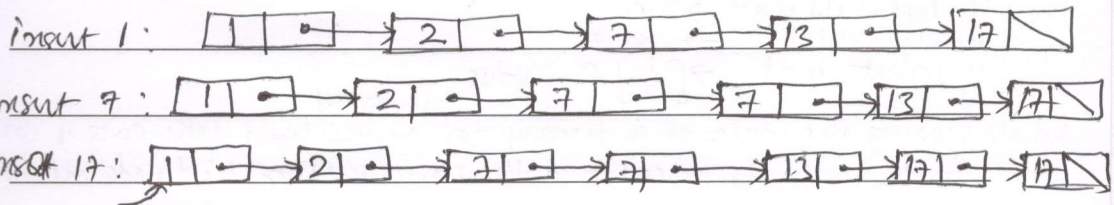
```
PQ_remove (A[], N)
BEGIN
    x = array_del(A[], N, 1)
    return x;
END
```

O(N) Time Complexity for insert and Removal

Priority Queue Using linked list:-



Let us insert the following elements in to the list: 1, 7, 17



q = NULL (initially)

Algorithm PQinsert (Start, PRN)

```
BEGIN
    p=start
    q=NULL
    while (p!=NULL) && PRN>=info(p) DO
          q=p
          p=next(p)
    If q==NULL then
          insert_begin(start, PRN)
    else  insert_after(q, PRN)
END
```

Algorithm PQRemove (Start)

```
Begin
    x = delbegin(start)
    return x
End;
```

O(1) for removal

Analysis: O(N) no of comparisions neglected in worst case
in best case O(1)

Applications / Usages of Priority Queues:-

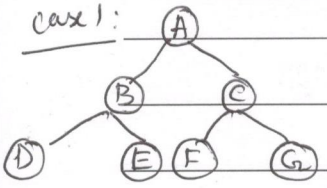
1. Heap sort implementation
2. Used in Operating system for load balancing and for interrupt handling
3. Used in data compression (Huffman Codes)
4. During the implementation of shortest Path algorithms.
5. Implementation of traffic light system

Sorting Techniques:-

Heap Sort, Heapify and Priority Queues:-

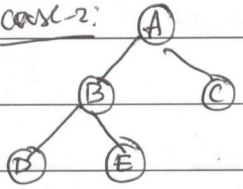
Array representation of Binary Tree:-

case 1:



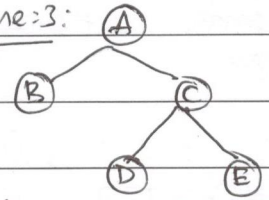
A	B	C	D	E	F	G
1	2	3	4	5	6	7

case 2:



A	B	C	D	E

case 3:



A	B	C	-	-	D	E

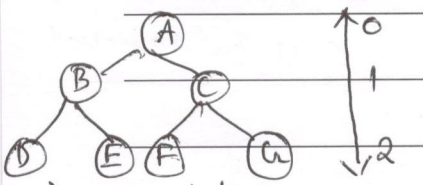
If a node is at index i in array then it satisfied the following:
then its left child is at $\rightarrow 2*i$
its Right child $\rightarrow 2*i+1$
its parent is at $\rightarrow \left[\frac{i}{2}\right]$ floor value

while storing the data of a binary tree in array we should take care of each node with its parent and the corresponding child node information

Consider C $i=3$ ∴ its parent is at $\frac{3}{2} = 1.5$ (Floor Value) $= 1$
left child is at $2*3 = 6$
right child is at $(2*3)+1 = 7$.

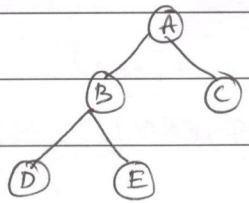
In arrays they are filled level wise. while filling level by level if any nodes were missing leave an empty cell (blank)

Full Binary Tree and Complete Binary Tree:-

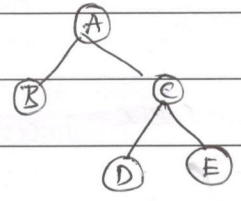


is a Complete B. Tree

A	B	C	D	E	F	G



A	B	C	D	E



A	B	C	-	-	D	E

If a binary tree with height 'h' then it can have $2^{h+1} - 1$ number of nodes.

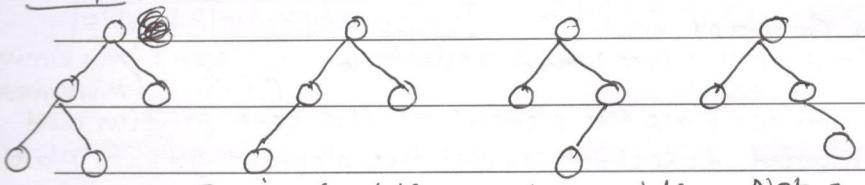
A binary tree with maximum numbers of nods is called full Binary Tree

other wise check the array representation of the binary tree, if there are any missing values (or blank) then we say that tree is complete Binary Tree. (gaps)

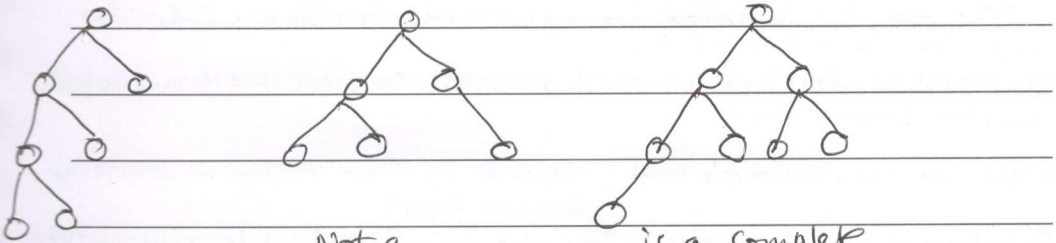
∴ Case1 and Case2 are examples of Complete Binary Trees.

A Complete Binary Tree is a full Binary tree upto level 'h-1'. In the last level the elements are filled from left to right.

Examples:



Is a Complete B.T | is a Complete B.T | Not a Complete B.Tree | Not a Complete B.Tree

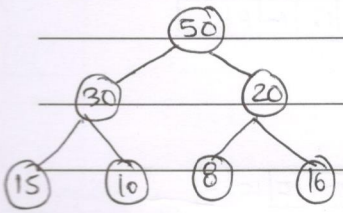


Not a Complete B.Tree | Not a Complete B.Tree | is a Complete B.Tree

The height of a Complete Binary tree is "log n" is always minimum

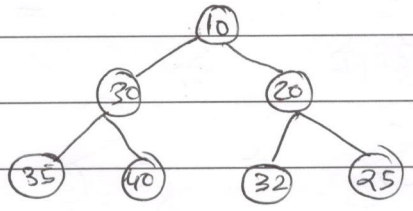
Heap:-

Max Heap



50	30	20	15	10	8	16
1	2	3	4	5	6	7

Min Heap

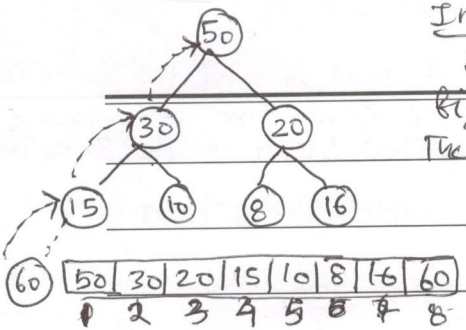


10	30	20	35	40	32	25

Heap is a complete Binary Tree which can be organized as either Max Heap / min Heap

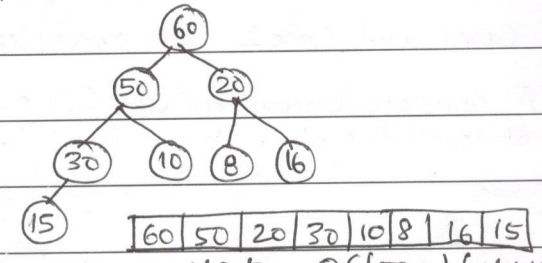
Max Heap: Every node is greater than or equal to its child node
∴ Larger element is at root.

Insertion in Max Heap:-



Insert-60

After placing 60 at the end as shown in figure it is violating the rule of max heap. Then apply adjust with all its predecessors



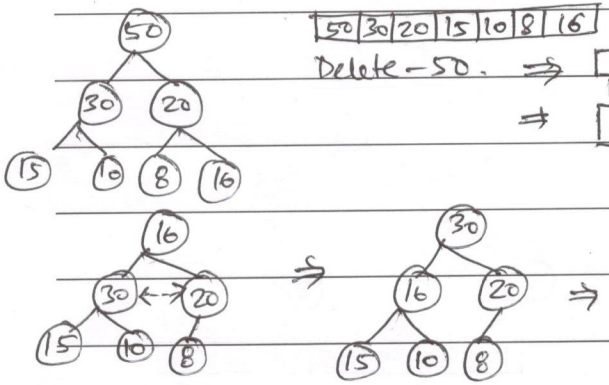
Here the number of swaps is equal to (the height of the tree) and results to O(log n) (Maximum)
O(1) (Minimum)

During the insertion we place the element at the leaf position and adjust it in upward direction until the element gets its place.

Delete operation in Max Heap:- Here we should not delete any other element except the root element without any options. The heap looks like an items placed in a pyramid structure. Only top element need to be taken out.

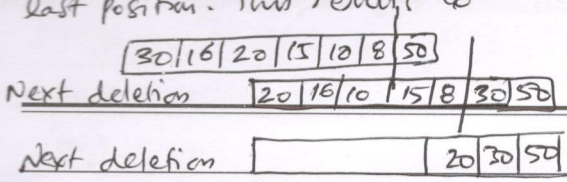
During insertion/deletion we must preserve the tree with a property of "Complete Binary Tree".

1. Delete the maximum element [element in first position in array is deleted first]
2. Last element in the array is brought into the first location of the array. [This is mainly to maintain the property of Complete Binary Tree"
3. Adjust the weights to see that it is "max heap".
   (a) Compare the element (root) with its children. Max element is replaced

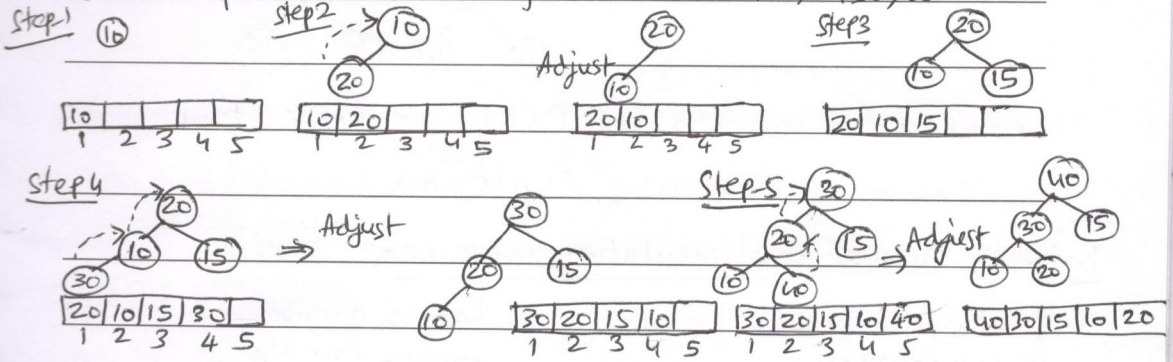


During the deletion the element is adjusted from root to leaf (downwards). The maximum time taken for deletion is O(log n)

To maintain a copy of deleted element, then keep a copy in the last position. This results to (store the element in the free space)

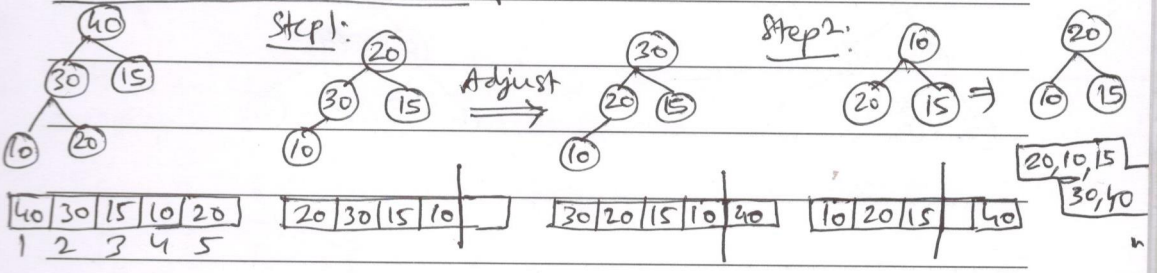


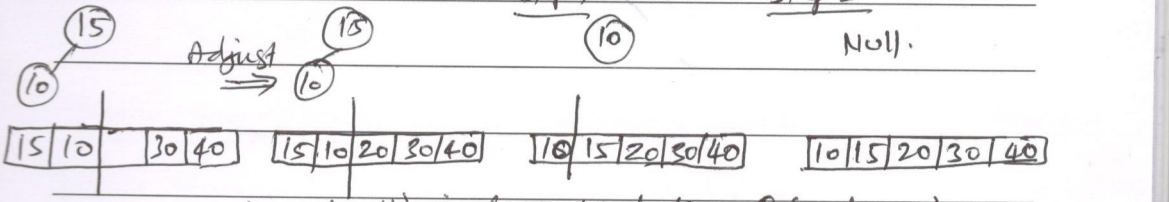
Create Heap with the following elements: 10, 20, 15, 30, 40



The time complexity for creation of heap is $O(n \log n)$

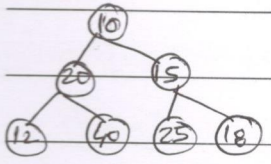
Deletion of elements from Heap:-





Deletion of all 'n' elements takes $O(n \cdot \log n)$

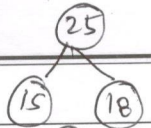
Heapify:- It is the procedure of creation of Heap with different procedure.



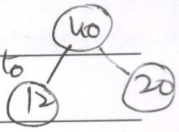
Here we scan the elements in the array from right to left and adjust the elements towards downwards

10 | 20 | 15 | 12 | 40 | 25 | 18

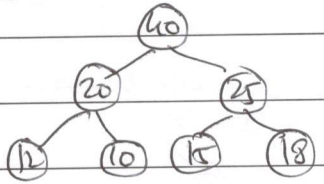
For 18, 25, 40 and 12 there were no child nodes. So no adjustment required. Move to the next element 15, compare with its children and adjust.



then element 20, compare with its children nodes results to



and in the final phase we get



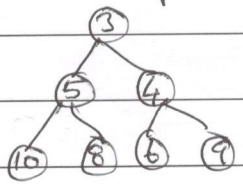
Time taken by heapify is $O(n)$ takes less time

Creating a Heap is $O(n \log n)$

Priority Queue implementation using Heap

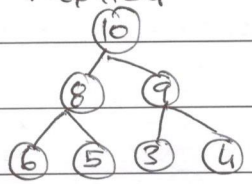
Smaller numbers
Higher Priority

Min Heap



Larger Numbers
Higher Priority

Max Heap



Quicksort: This sorting is quicker than other sorting algorithms. It uses "partition and exchange" method. Here we choose one element as "pivot" element (which can be first element / last element / random element. It is based on "divide-and-conquer" strategy. This method divides the array into smaller sub-arrays until the list is sorted. Here all the elements <= pivot are placed to the ~~right~~ left of the pivot and all elements > pivot are placed to the right of pivot.

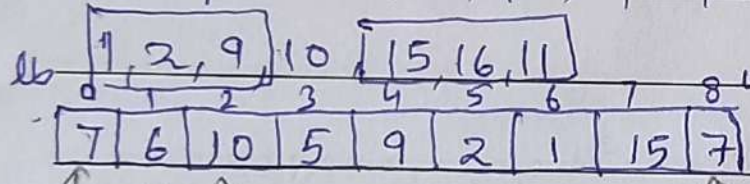
Partition-1	Pivot element	Partition-2

Values are <= Pivot

Values are > Pivot element

Apply the same principle on the partition-1 and Partition-2

Example: 10, 15, 1, 2, 9, 16, 11 if pivot = 10



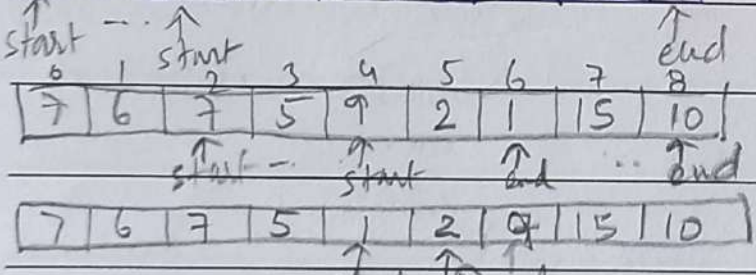
start move from left to right / end (right to left)

start will stop at if $x > Pivot$

increment if $x <= Pivot$

end will decrement if $x > pivot$

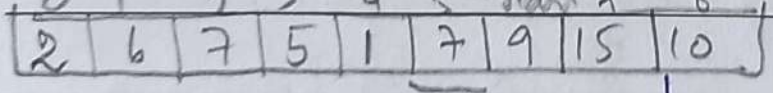
stop if $x < pivot$



Here start >= end (no swapping)

if start < end (swapping)

else swap (pivot and end)




```
quicksort(A, lb, ub)
{ if(lb < ub)
  { loc = partition(A, lb, ub);
    quicksort(A, lb, loc-1);
    quicksort(A, loc+1, ub);
  }
}
```

```
partition(A, lb, ub)
{ pivot = a[lb];
  start = lb; end = ub;
  while(start < end)
  { while(a[start] <= pivot)
    { start++;
    }
    while(a[end] > pivot) { end--; }
```

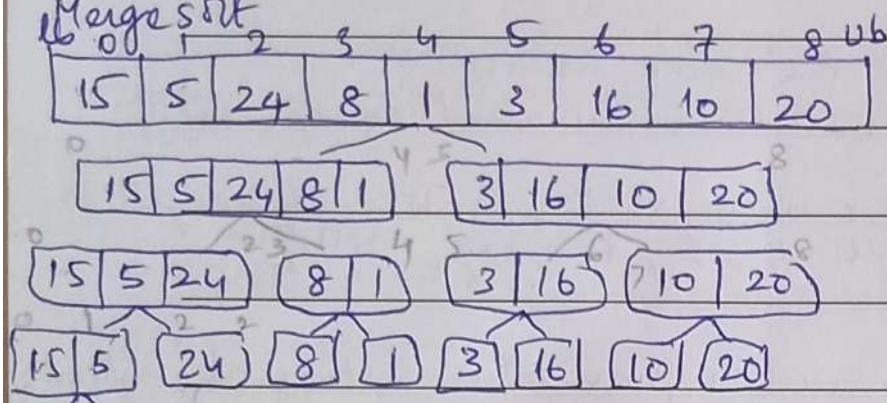
```
if(start<end)
{swap(a[start] a[end]);
else?
swap(a[lb], a[end]);
}
return end;
}
```

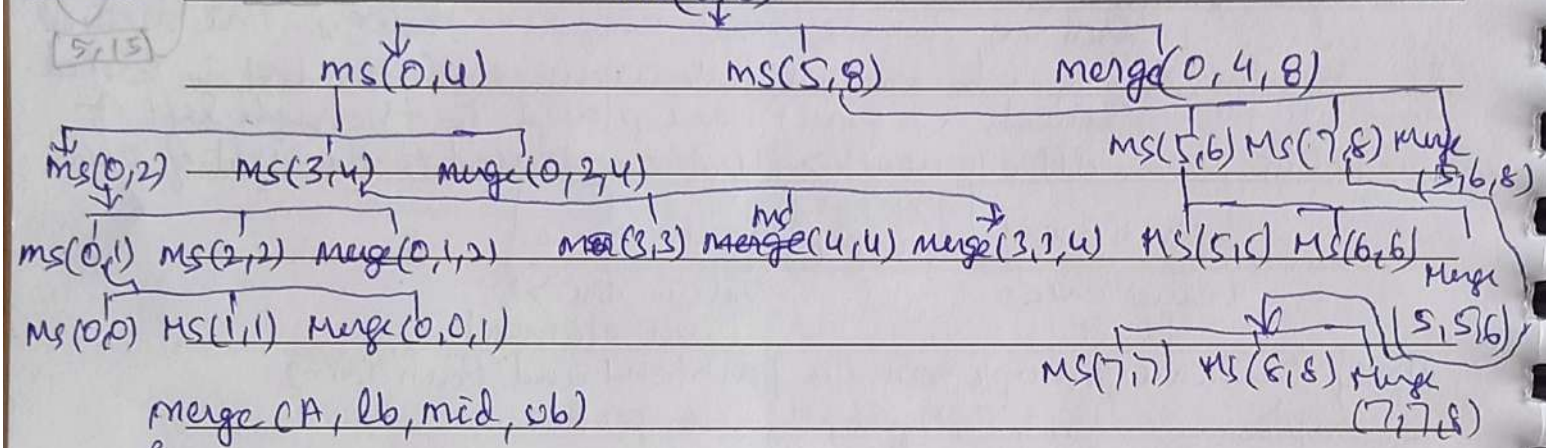
Time complexity:
worst case : $O(n^2)$
Average case : $O(n \log n)$
where n is the number of elements

worst case is seen when the list is imbalanced (one of the partition is always empty)
Best case (leads to binary tree and set of log n)

```
mergesort(A, lb, ub)
{ if(lb<ub)
  { mid = (lb+ub)/2;
    mergesort(A, lb, mid);
    mergesort(A, mid+1, ub);
    merge(A, lb, mid, ub);
  }
}
```

Mergesort :-



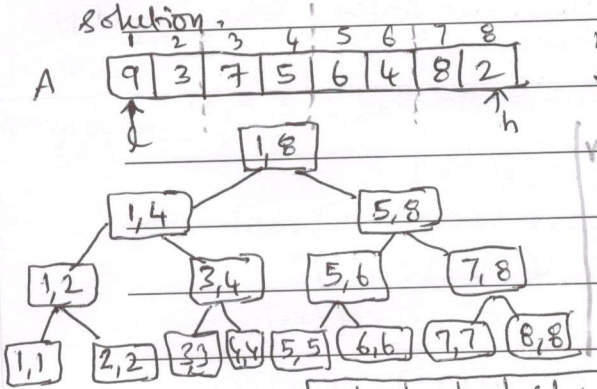



```
merge(A, lb, mid, ub)
{ i = lb;
  j = mid+1;
  k = lb;
  while(i<=mid && j<=ub)
  { if(a[i]<=a[j])
    { b[k] = a[i];
      i++;
    }
    else
    { b[k] = a[j];
      j++;
    }
    k++;
  }
  if(i>mid)
  { while(j<=ub)
    { b[k] = a[j]; j++; k++; }
  }
  else
  { while(i<=mid)
    { b[k] = a[i];
      i++;
      k++;
    }
  }
  for(k=lb; k<=ub; k++)
  { a[k] = b[k];
  }
}
```

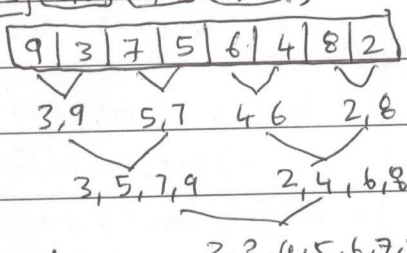
Time complexity (in case of both worst case & Avg case)

$O(n \log n)$

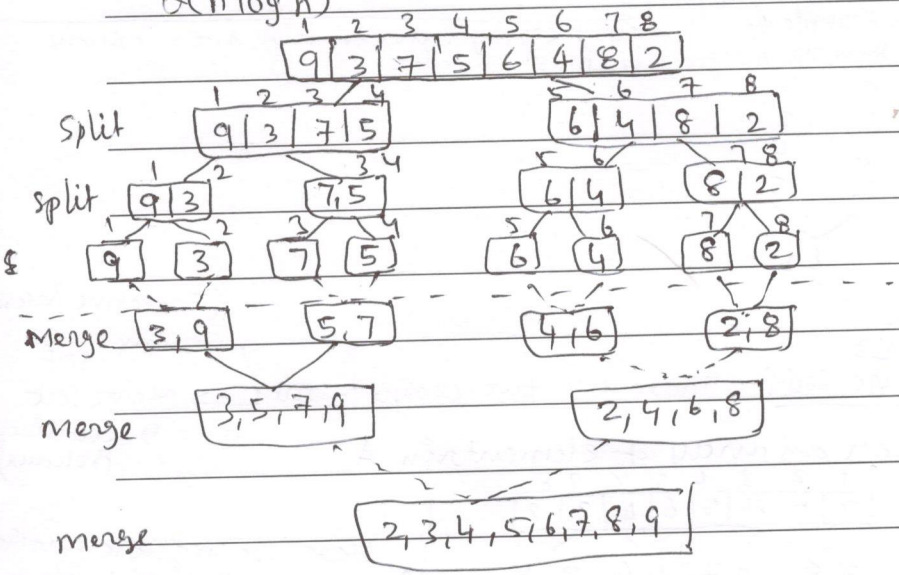
2. MergeSort :- It is a recursive procedure and follows divide-and-conquer strategy. It follows that "If the problem is large then divide the problem into smaller sub-problems and solve them and combine the results of each sub-problem to get the final solution".




```
Algorithm MergeSort(l, h)
{
  if(l < h)
  {
    mid = (l+h)/2;
    mergesort(l, mid);
    mergesort(mid+1, h);
    merge(l, mid, h);
  }
}
```



Time complexity is $O(n \log n)$



NOTE:-
Merging is done in post order traversal in this method.

Radix Sort :-

Radix sort is a non-comparative sorting algorithm. It avoids comparision by creating and distributing elements into buckets according to their radix (base of a numbers). For elements with more than one significant digit, this bucketing process is repeated for each digit, while preserving the ordering of the prior step, until all digits have been considered. For this reason radix sort is also called "bucket sort" or "digital sort".

$n = 10$

A →

0	1	2	3	4	5	6	7	8	9
432	008	530	090	088	231	011	045	677	199

Pass-1 count Array

0	1	2	3	4	5	6	7	8	9
2	4	5	5	5	6	6	7	9	10

update count array to fix the location of the elements in B

B →

0	1	2	3	4	5	6	7	8	9
530	090	231	011	432	045	677	008	088	199

Pass-2

0	1	2	3	4	5	6	7	8	9
1	2	2	5	6	6	6	7	8	10

B →

0	1	2	3	4	5	6	7	8	9
008	011	530	231	432	045	677	088	090	199

Pass-3

0	1	2	3	4	5	6	7	8	9
5	6	7	7	8	9	10	10	10	10

B →

0	1	2	3	4	5	6	7	8	9
008	011	045	088	090	199	231	432	530	677

```
Algorithm radixsort(a,n)
{ int max=getmax(a,n);
  for(pos=1; max/pos>0; pos*10)
      countsort(a,n,pos);
}
countsort(int a[], int n, int pos)
{ int count[10]={0};
  for(i=0; i<n; i++)
      ++count[(a[i]/pos)%10];
  for(i=1; i<=k; i++)
      count[i]= count[i]+count[i-1];
  for(i=n-1; i>=0; i--)
      b[--count[(a[i]/pos)%10]] = a[i];
  for(i=0; i<n; i++)
      a[i]=b[i];
}
```

Time complexity of the algorithm:

$O(n+k)$ — $3n+k$ — 3 times for loop

$O(d*(n+k))$

K : number of distinct key elements.

Here $K=10$.

Description:- ① Here sorting is done by the place value of the digit.
Ex: For a 3 digit number sorting is done digit by digit for three times.

Example:- 199

MSB ← LSB
Most Significant Bit — Least Significant Bit.

Sorting is done from LSB to MSB in case of digits.
If strings/characters we follow MSB to LSB

The number of sorting (Passes) is based on the highest number present in the list.
Ex:- 699 (3 digits) Hence sorting is done for 3 times.

② Make all numbers in the list equated with the maximum number as follows: 8 ⇒ 008 and 88 ⇒ 088

③ Pass-1: Use count array and initialize it with zeros. The size of the count=10 (to meet all types of digits which occur in any number)
Increment the value in counter based on the presence of value in the one's place from left to right.

④ Update the counter: Update the counter so that it reflect the actual exact position of the value in the array.

⑤ Build the output array
Output array is similar to the original array. Scan each element based on one's place from right to left and map it with counter index value, decrement it by 1 and copy the value into the output array.

⑥ Repeat steps 3 to 5 until all significant bits from LSB to MSB is completed.