UNIT-2

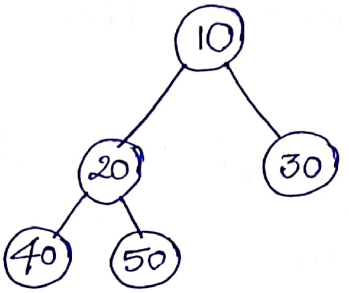
PART-A

Heap Trees (Priority Queues)

What is Heap?

→ A heap tree is a complete binary tree.

→ Complete Binary tree is a binary tree in which all levels except last level must be filled completely.



1. Min Heap Tree :-

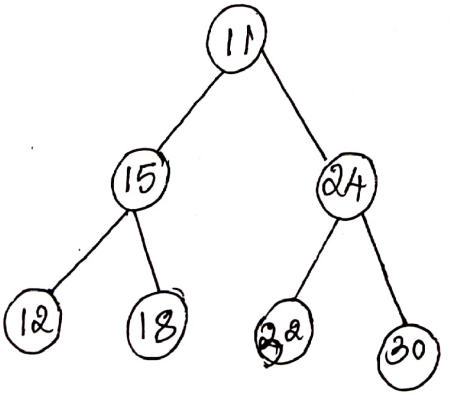
The value of the parent node should be less than (or) equal to either of its children.

(or)

Every node is greater than (or) equal its parent value except the root node.

$$A[Parent(i)] <= A[i]$$

Eg:-



Operations on Min Heap Tree

1. Insert (or) Create
2. Search
3. Delete.

1) Insert :- Method of insertion follows.

step 1 :- Create a new node at the end of heap

step 2 :- Assign new value to the node

step 3 :- Compare the value of this child node with its parent.

step 4 :- If value of parent is greater than child, then swap them.

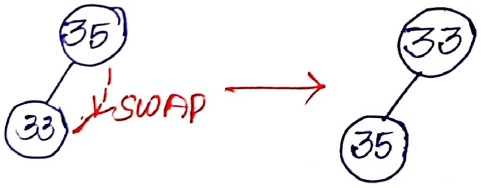
step 5 :- Repeat step 3 & 4 until Heap property holds.

Eg :- Construct a minheap tree with the given elements.

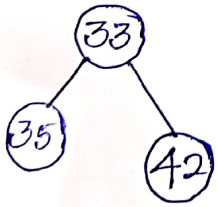
input ← {35, 33, 42, 10, 14, 19, 27, 44, 26, 31}

step 1 :- Insert 35

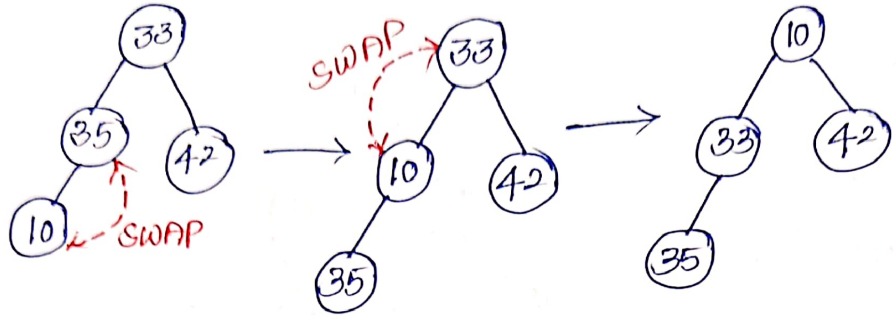
step 2 :- Insert 33



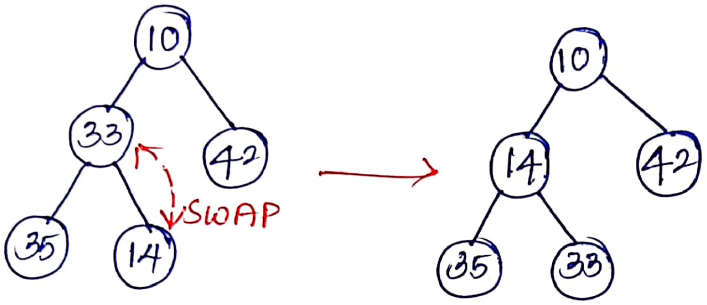
step 3 :- Insert 42



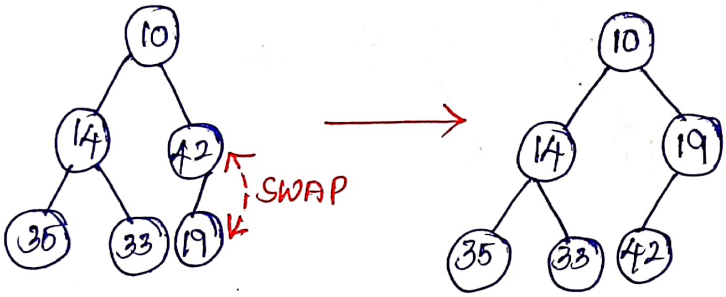
step 4 :- Insert 10



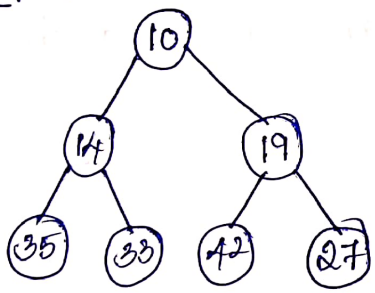
step 5 :- Insert 14



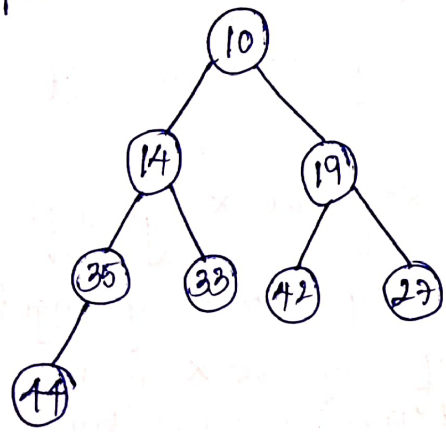
step 6 :- Insert 19



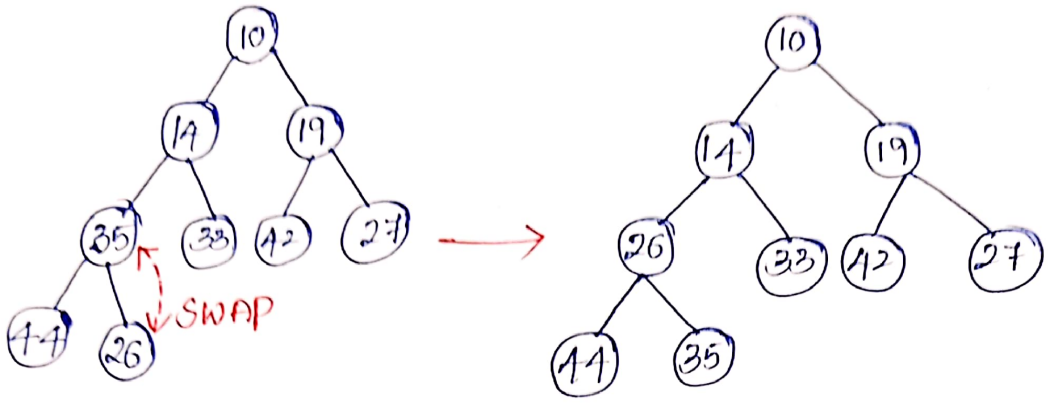
step 7 :- Insert 27



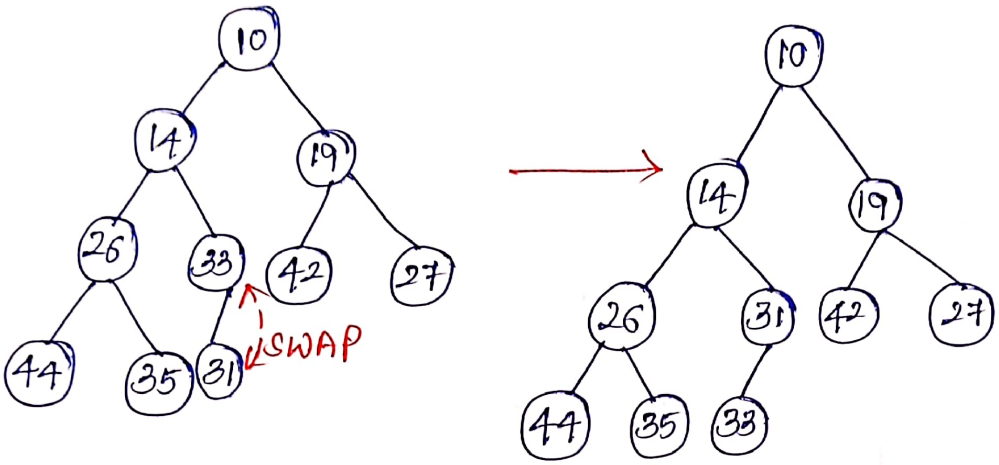
step 8 :- Insert 44



step 9 :- Insert 26



step 10 :- Insert 31

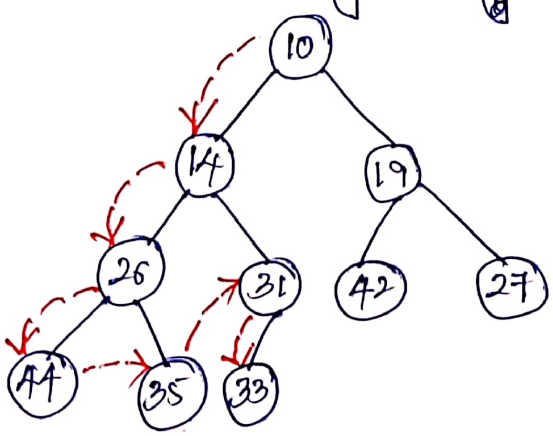


2) Search :-

→ Search in Heap tree is same as search in Binary Tree.

→ Searching follows DFS.

→ Search in given tree (key = 33)



1. if key == 10 x go left.
2. if key = 14 x go left.
3. if key = 26 x go left
4. if key = 44 x go right of 26
5. if key = 35 x go to right of 14
6. if key = 31 x go left key = 33

3> Delete :- Deletion in Heap tree is done only at root node.

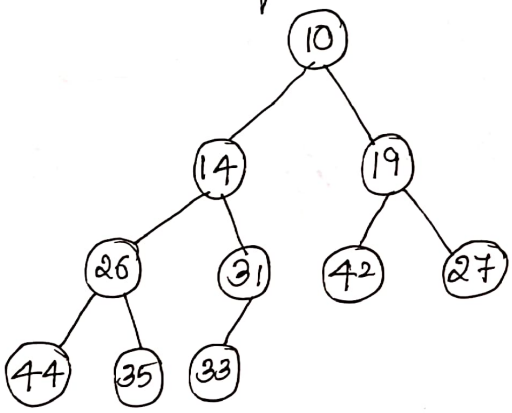
step 1 :- To Delete root node, swap root node with last element of the tree.

step 2 :- Now delete last node.

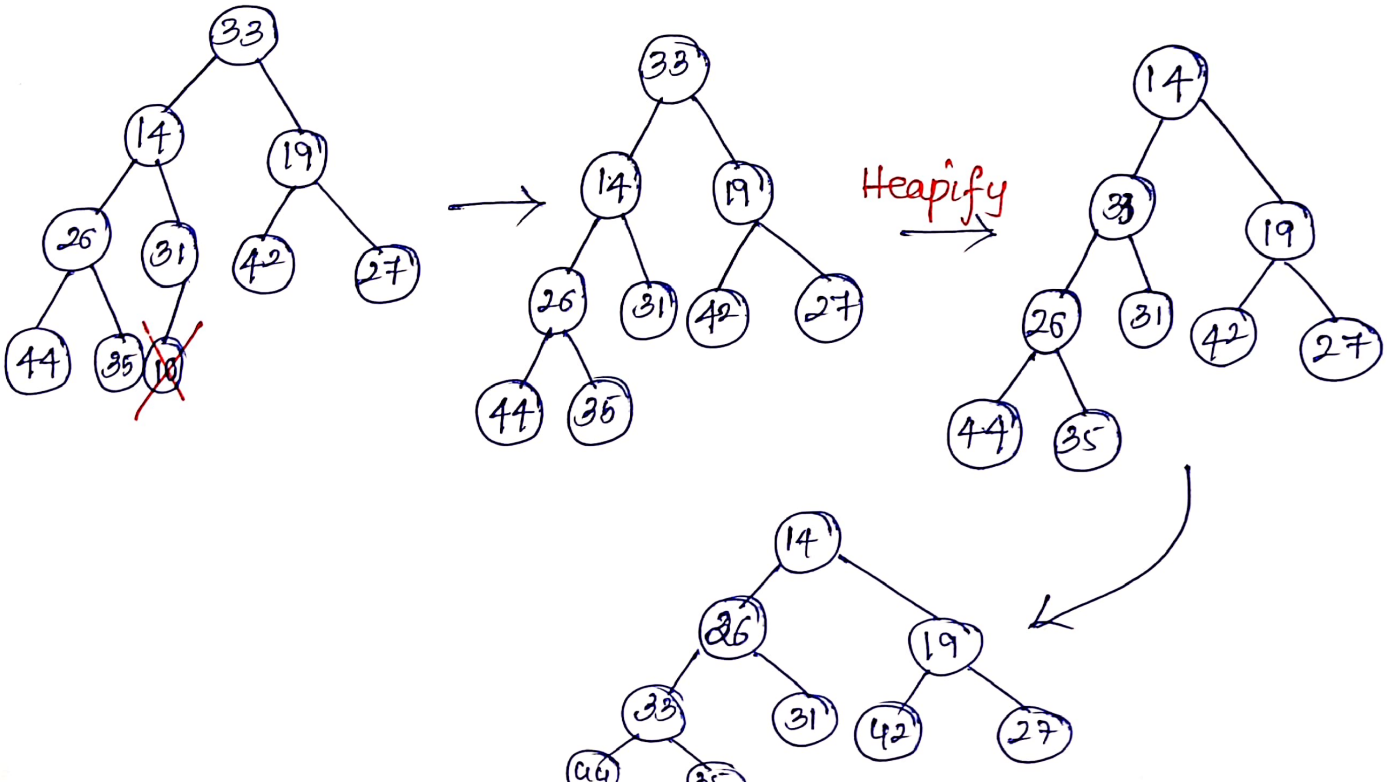
step 3 :- Heapify the tree until it satisfies minHeap property.

step 4 :- Stop.

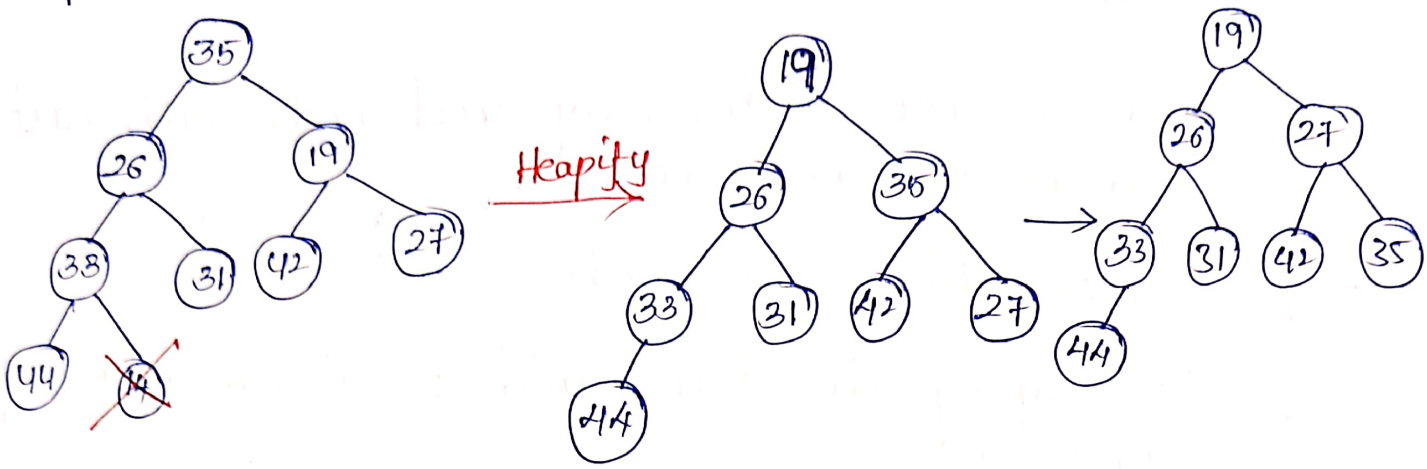
Eg :- Delete first 3 values of given minheap tree



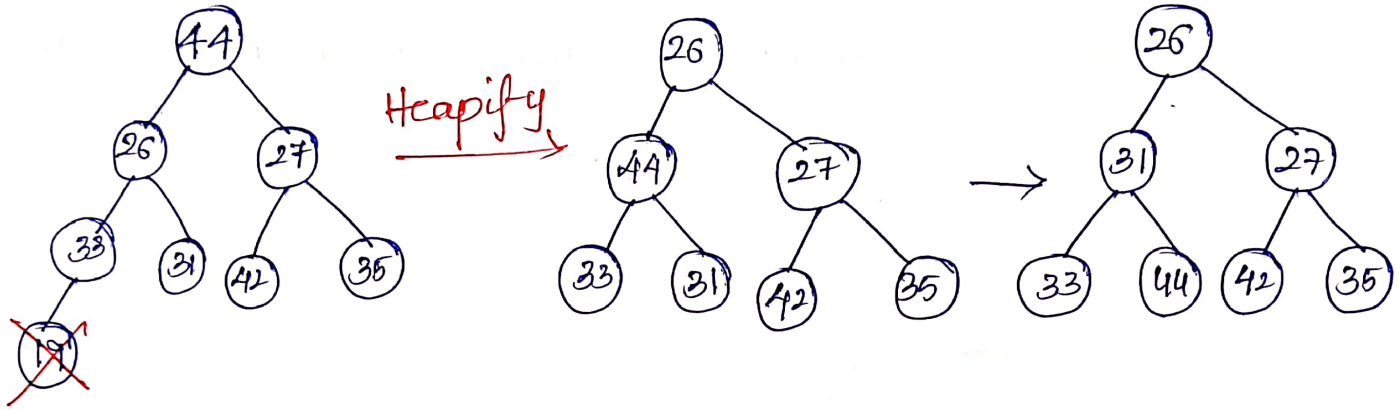
step 1 :- Delete 10.



step 2:- Delete 14



step 3:- Delete 19



2. Max Heap Tree

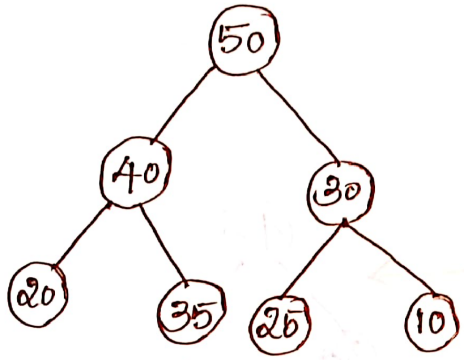
The value of the parent node should be greater than (or) equal to either of its children.

(or)

Every node is lesser than (or) equal to its parent value except the root node.

$$A[Parent(i)] >= A[i]$$

Eg:-



Operations on MaxHeap Tree

1. Insert (or) Create
2. Search
3. Delete.

(1) Insert :-

step 1: Create a new node at the end of heap.

step 2: Assign new value to the node

step 3: Compare the value of this child node with its parent.

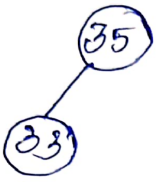
step 4: If the value of parent is less than child then swap them

step 5: Repeat step 3 & 4 until Heap property holds.

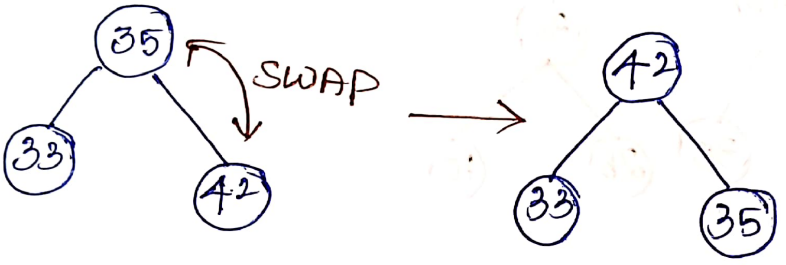
Example :- Construct a MaxHeap tree with the given elements. input ← {35, 33, 42, 10, 14, 19, 27, 44, 26, 31}

step 1 :- Insert 35

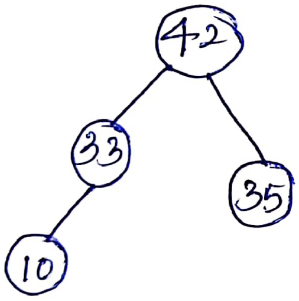
step 2 :- Insert 33



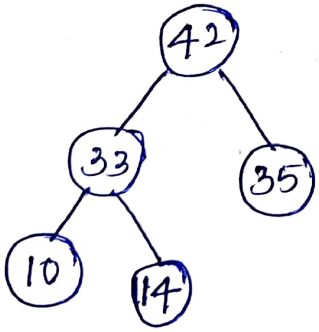
step 3 :- Insert 42



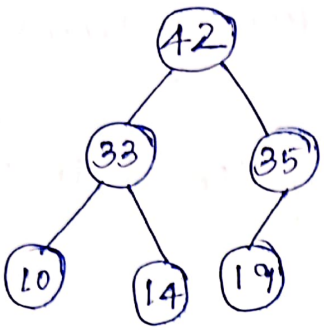
step 4 :- Insert 10



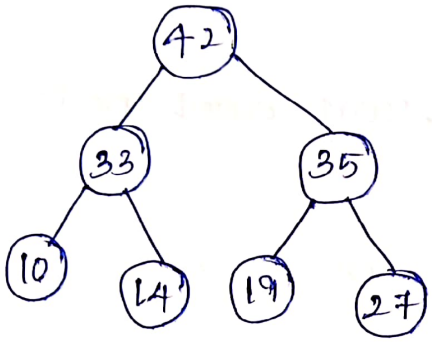
step 5 :- Insert 14

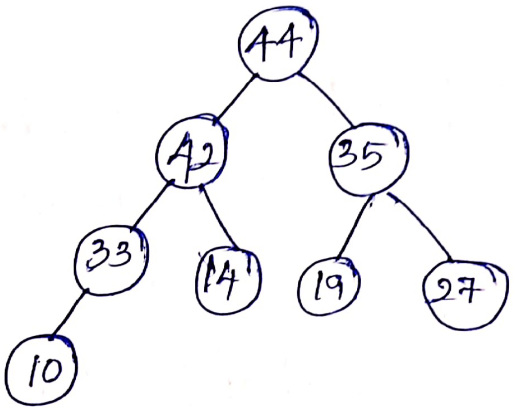


step 6 :- Insert 19

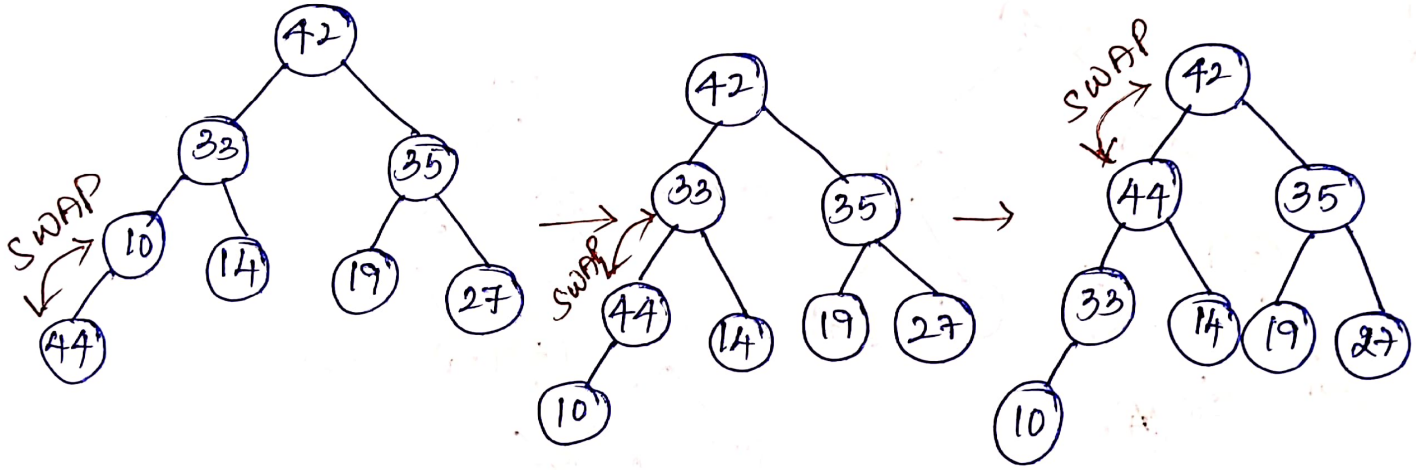


step 7 :- Insert 27

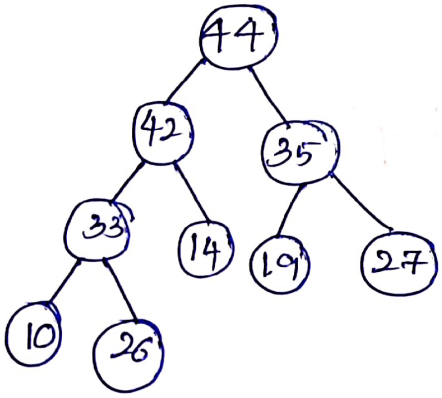




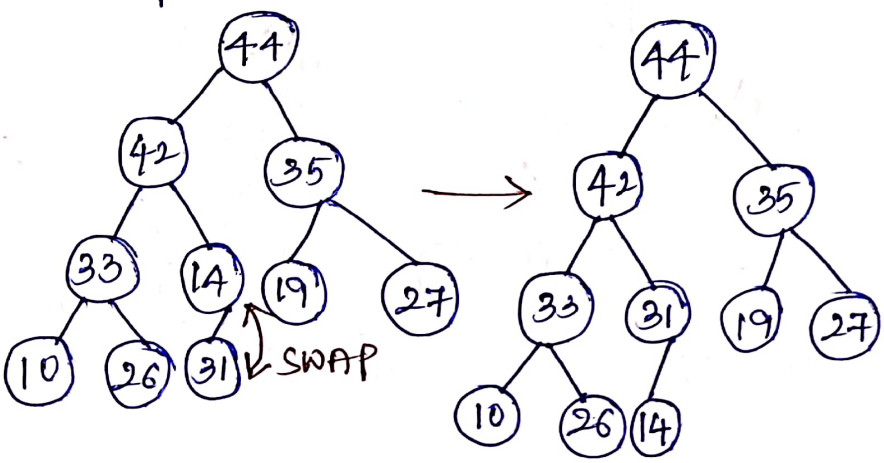
step 8 :- Insert 44



step 9 :- Insert 26.



step 10 :- Insert 31



2) Search

→ Searching an element in MaxHeap tree follows searching technique of Binary Tree.

→ Refer search operation in minHeap (For both it is same)

3) Delete

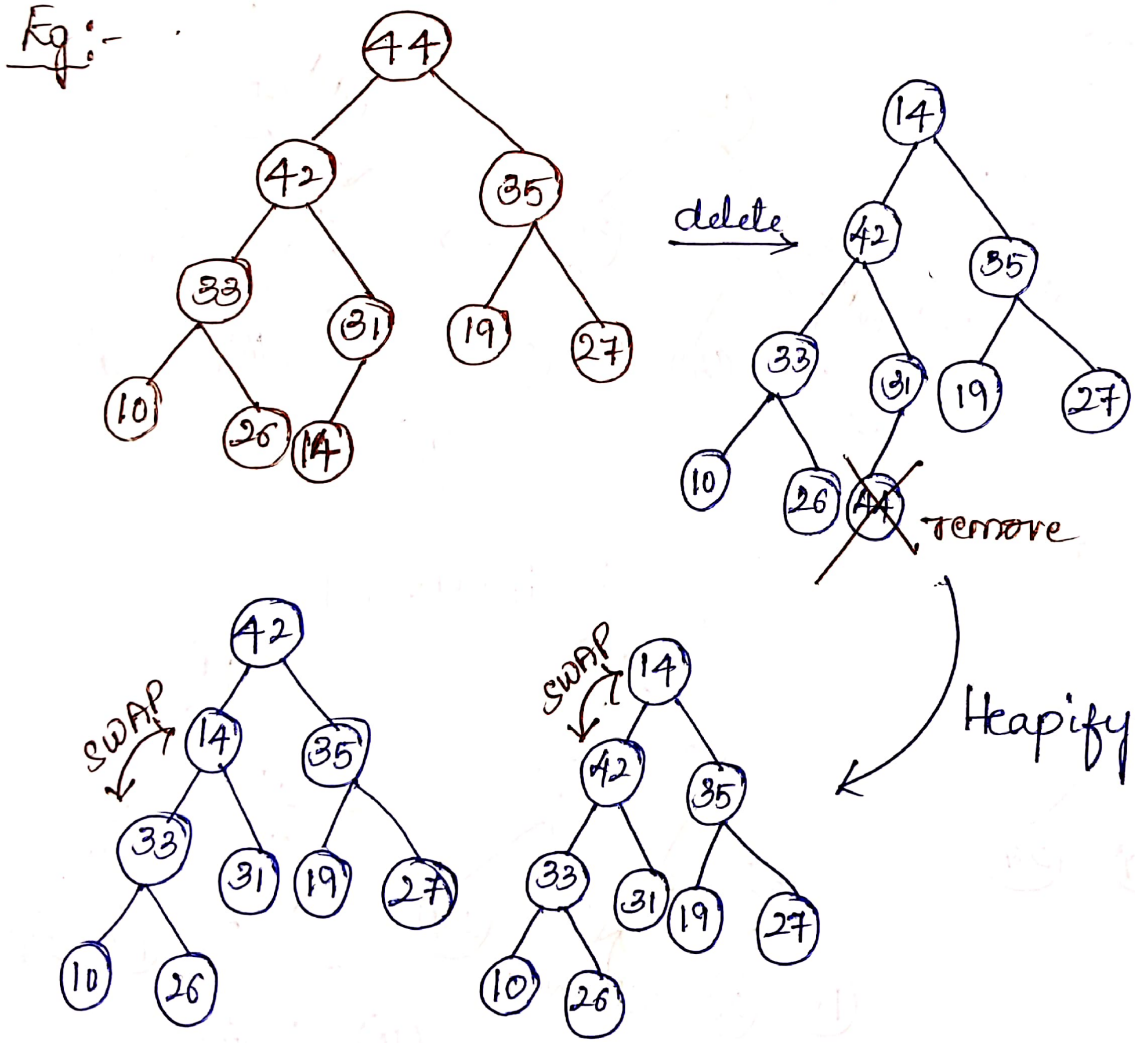
→ In Heap tree the deletion is allowed only at root node.

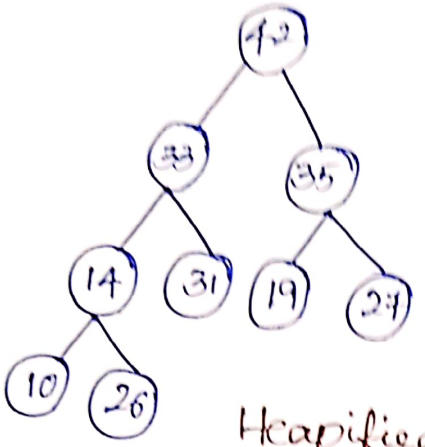
step 1: To delete the root node, swap root node with last element of the tree.

step 2: Now delete last node of the tree.

step 3: Heapify the tree.

Eg:-





Heapified

Advantages of Heaptrees

* Efficient Insertion and deletion
* Efficient priority queue.
* Guaranteed access to the maximum (or) minimum.
* Space efficiency
* Heap-sort Algorithm

Disadvantages of Heap Trees

* Lack of flexibility
* Not ideal for searching
* Not a stable data structure
* Memory management
* Complexity

Applications of Heap Trees (Explain each in detail in your own style)

1. Priority Queues
2. Heapsort Algorithms
3. Job scheduling
4. Graph algorithms

Part-B :- Graphs

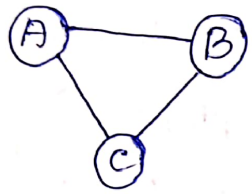
Topic-1 Graph Terminologies

1. Graph :-

A Graph 'G' is a non-empty set of vertices (or nodes) V and a set of edges E, where each edge connects a pair of vertices.

$$G = (V, E)$$

Eg :-



2. Vertex (Node) :-

It represents an entity within the graph.

Eg :- from above graph A, B and C are vertices

3. Edge :-

An edge is a connection between two vertices in a graph. It can be either directed (or) undirected.

Eg :- A-B, B-C, C-A are the edges in above graph.

4. Degree of vertex

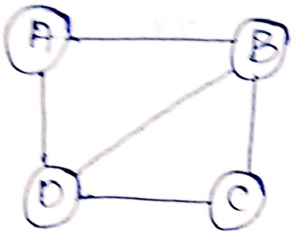
The no. of edges incident to that vertex is the degree of that vertex.

→ In directed graph, the degree is further classified into the "in-degree (d+)" and "out-degree (d-)",

→ In-degree is the no. of incoming edges to the vertex.

→ Outdegree is the no. of outgoing edges from the vertex

Un-Directed Graph.



→ degree of

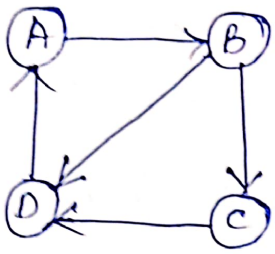
$d(A) = 2$

$d(B) = 3$

$d(C) = 2$

$d(D) = 3$

Directed Graph



→ Indegree and Outdegrees

$d^+(A) = 1$, $d^-(A) = 1$

$d^+(B) = 1$, $d^-(B) = 2$

$d^+(C) = 1$, $d^-(C) = 1$

$d^+(D) = 2$, $d^-(D) = 1$

5) Path :-

A path in a graph is a sequence of vertices where each adjacent pair is connected by an edge.

→ Paths from A to C from above Undirected graph are

1. A-B-C
2. A-D-C
3. A-B-D-C
4. A-D-B-C

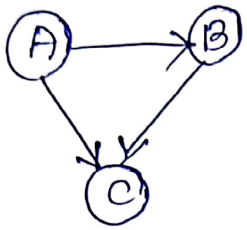
6) Cycle :-

A cycle in a graph is a path that starts and ends at the same vertex.

Eg :- A-B-D-A (from the above graphs)

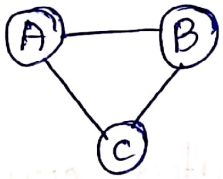
7) Directed Graph (Digraph) :-

A Directed Graph consists of nodes (vertices) connected by directed edges (arcs). Each edges has a specific direction.



8) Undirected Graph :-

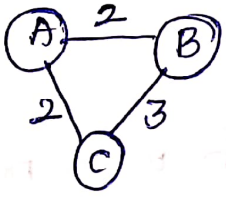
Edges have no direction. They simply connect nodes without any inherit order.

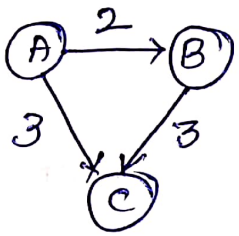


9) Weighted Graph :-

Weighted graphs assign numerical values (weights) to edges.

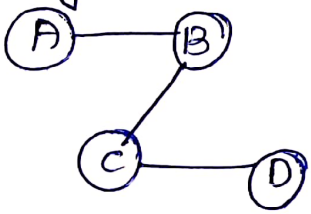
→ These weights represent some property associated with the connection between nodes.





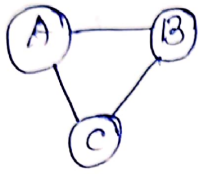
10) Acyclic Graph

An acyclic graph contains no cycles (closed loops).



11. Cyclic Graph

A cyclic graph has at least one cycle. You can traverse edges and eventually return to the same node.



12. Connected Graph:-

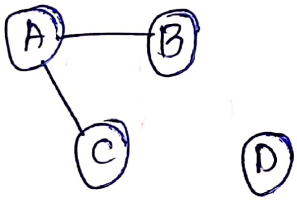
A graph is connected if there is a path between any pair of nodes.

→ In other words, you can reach any node from any other node.

13. Disconnected Graph:-

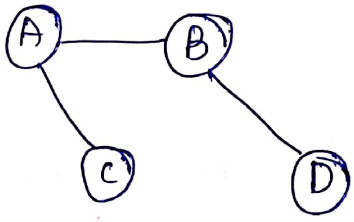
A disconnected graph has isolated components that are not connected to each other.

→ These components are separate subgraphs.



14. Tree:-

A Tree is a Connected graph with no cycles.



Topic-2:- Representation

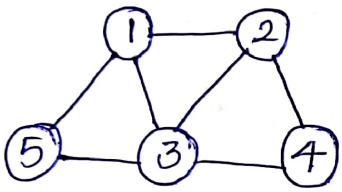
Here are the two most common ways to represent a graph. (For Unweighted Graphs)

1. Adjacency Matrix
2. Adjacency List

1. Adjacency Matrix

An adjacency matrix is a way of representing a graph as a matrix of boolean (0's and 1's)

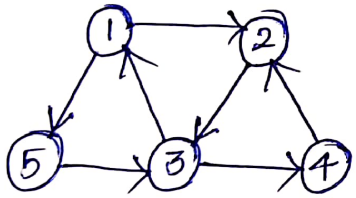
Graph Representation of Undirected graph to Adjacency Matrix



→

	1	2	3	4	5
1	0	1	1	0	1
2	1	0	1	1	0
3	1	1	0	1	1
4	0	1	1	0	0
5	1	0	1	0	0

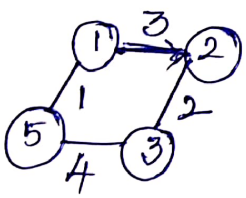
Graph Representation of Directed graph to Adjacency Matrix



→

	1	2	3	4	5
1	0	1	0	0	1
2	0	0	1	0	0
3	1	0	0	1	0
4	0	1	0	0	0
5	0	0	1	0	0

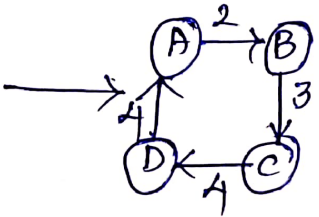
Graph Representation of Weighted Undirected graph to Adjacency Matrix



→

	1	2	3	5
1	0	3	0	1
2	3	0	2	0
3	0	2	0	4
5	1	0	4	0

Graph Representation of weighted Directed graph



→

	A	B	C	D
A	0	2	0	0
B	0	0	3	0
C	0	0	0	4
D	4	0	0	0

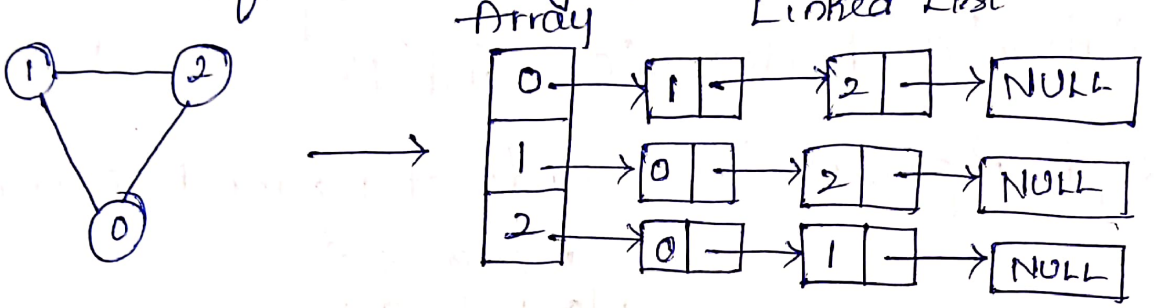
2) Adjacency List :-

→ An array of Lists is used to store edges between two vertices.

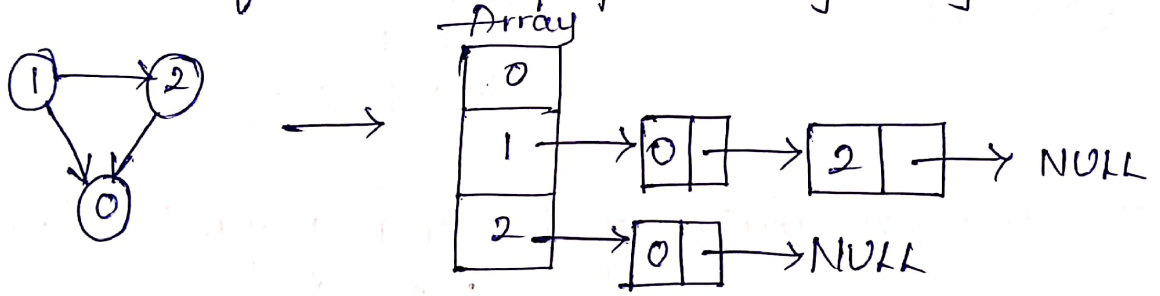
→ The size of array is equal to the number of vertices.

→ Each index in this array represents a specific vertex in the graph.

* Representation of Undirected graph as Adjacency list



* Representation of Directed Graph as Adjacency list



Topic-3:- Basic Search and Travels

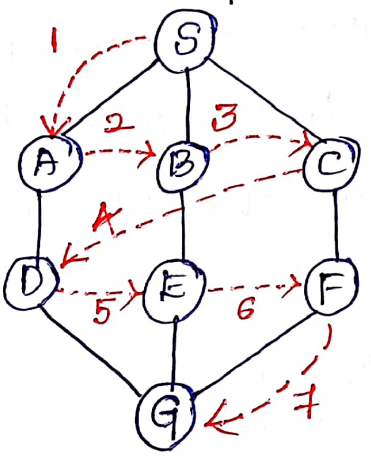
"The process of traversing all the nodes (or) vertices on graph is called graph traversal".

→ We have two traversal techniques on graphs.

1. Breadth First Search (BFS)
2. Depth First Search (DFS)

(1) Breadth First Search

BFS algorithm starts at the tree root and explores all nodes at the present depth prior to moving on to the nodes at the next depth level.



→ As in the example given above, BFS algorithm traverses from A to B to C to D first then to E and to F lastly to G.

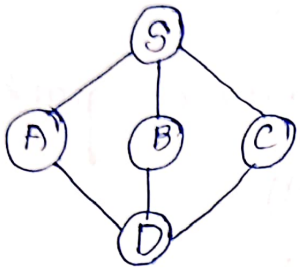
Rule 1:- Visit the adjacent unvisited vertex Mark it as visited. Display it. Insert it in a queue.

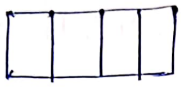
Rule 2:- If no adjacent vertex is found, remove the first vertex from the queue.

Rule 3:- Repeat Rule 1 and Rule 2 until the queue is empty.

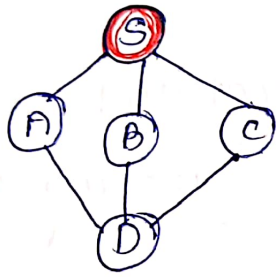
Example

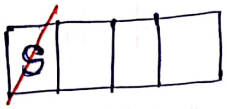
step 1:- Initialize the Queue.



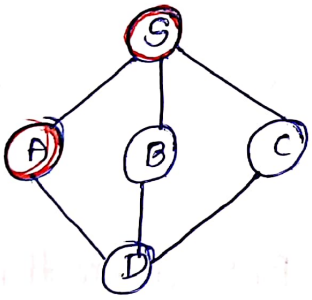


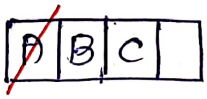
step 2:- We start from visiting S, and make it as visited.



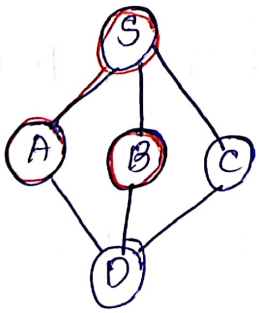


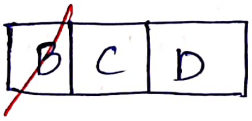
step 3: We then see an unvisited adjacent nodes from S. In this example, we have three nodes but alphabetically we choose A, mark it as visited and enqueue it.



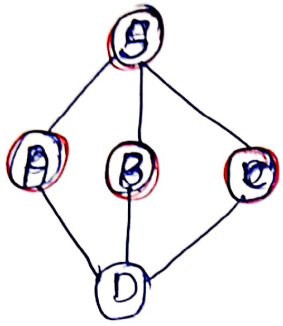


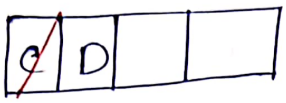
step 4:- Next, the unvisited adjacent node from A, S. is B, C, D we mark it as visited and enqueue it.



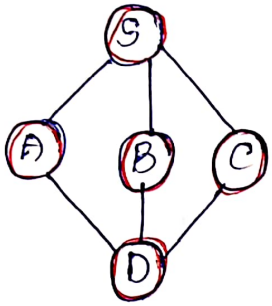


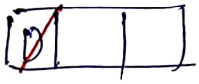
step5:- Next, the unvisited adjacent node from S, A, B is C, D. We mark it as visited and enqueue it.





step 6:- Now, the unvisited adjacent nodes from S, A, B & C is D, visit D.





→ Time Complexity of BFS = O(V + E)

where V ← No. of vertices
E ← No. of Edges.

→ Space Complexity of BFS = O(V).

where V ← No. of vertices

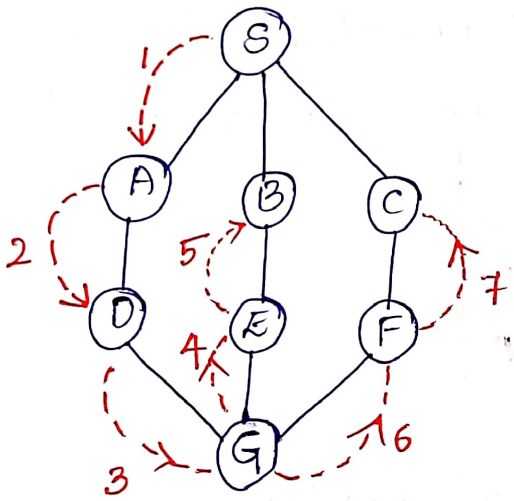
Algorithm of BFS

```
BFS(rootnode)
{
 create queue
 create list of visited nodes
 mark root node as visited
 enqueue root node
 while (queue is not empty)
 {
  x = queue.pop()
  queue.pop()
  for (all immediate neighbors of x)
  {
    if (not visited)
    {
      enqueue
      mark as visited
    }
  }
 }
}
```

2. Depth-First Search

→ DFS is a recursive algorithm for searching all the vertices of a graph.

→ This algorithm traverse a graph in a depthward motion and uses a stack to remember to get the next vertex to start a search, when a dead end occurs in any iteration.

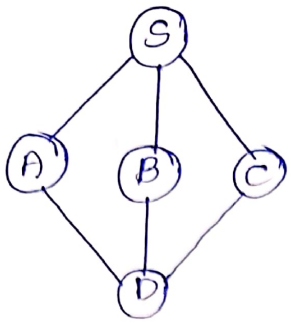


Rule 1:- Visit the adjacent unvisited vertex Mark it as visited. Display it. Push it in a stack.

Rule 2:- If no adjacent vertex is found, pop up a vertex from the stack. (It will pop up all the vertices from the stack, which do not have adjacent vertices

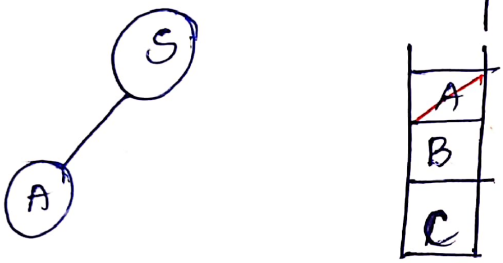
Rule 3:- Repeat Rule 1 and Rule 2 until the stack is empty.

Example

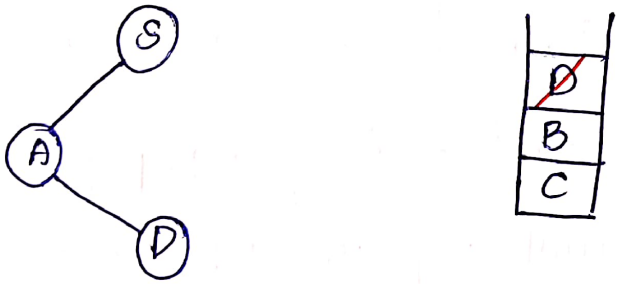


step 1:- Initialize a stacks and push start node to start.

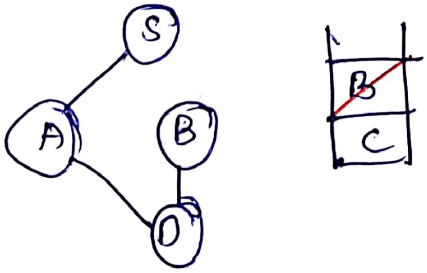
step 2:- Now find adjacent unvisited vertices of S into stack. and visit top most value.



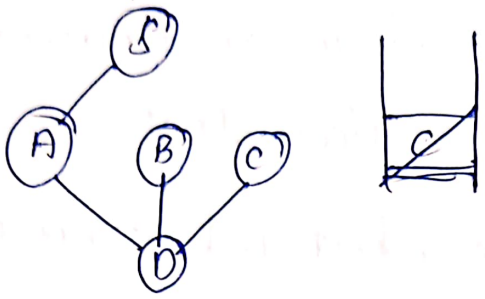
step 3:- Now find adjacent unvisited vertices of top most deleted node A. and add it to stack and visit top most value.



step 4:- No unvisited adjacent vertices to add into stack and visit top most node.



step5:- Visit top most node from stack.



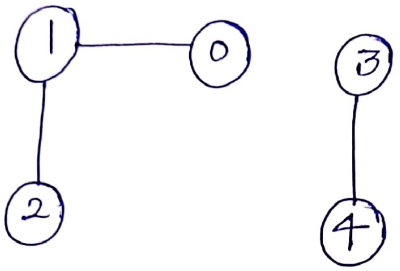
step6:- No nodes to visit, Since stack is empty.

Algorithm for DFS

```
DFS(rootnode)
{
 create stack
 create list of visited nodes
 mark rootnode as visited
 push rootnode into stack
 while (stack is not empty)
 {
    x = stack.top()
    stack.pop()
    for(all immediate neighbors of x)
    {
      if (not visited)
        {
          push into stack
          mark as visited
        }
    }
 }
}
```

Topic-4:- Connected Components

Connected component in an undirected graph refers to a group of vertices that are connected to each other through edges, but not connected to other vertices outside the group.



→ For example, in the graph shown ~~below~~ above {0, 1, 2} form a connected component and {3, 4} form another connected component.

Characteristics of Connected Components

* Set of vertices in a graph that are connected to each other.
* A graph can have multiple connected components.
* Inside a component, each vertex is reachable from every other vertex in that component.

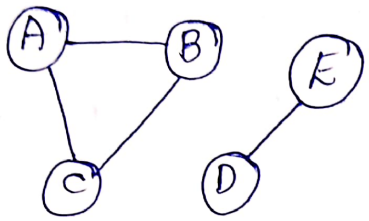
How to identify Connected Component

→ There are several algorithms to identify Connected Components in a graph. They are

1) Depth-First-Search (DFS)
2) Breadth-First-Search (BFS)
3) Union-Find Algorithm (Also known as Disjoint)

→ How to find the no. of connected Components in a given graph ?

→ Run the following DFS algorithm, here the no. of times the DFS function is called.



→ The No. of connected components in the above graph is 2, because the following DFS method is called 2 times to visit all the nodes.

```
dfs(v)
{
  mark[v] = 1
  print(v)
  for each (v, u) ∈ Adj[v]
      if (mark[u] = 0)
          dfs(u)
}
```

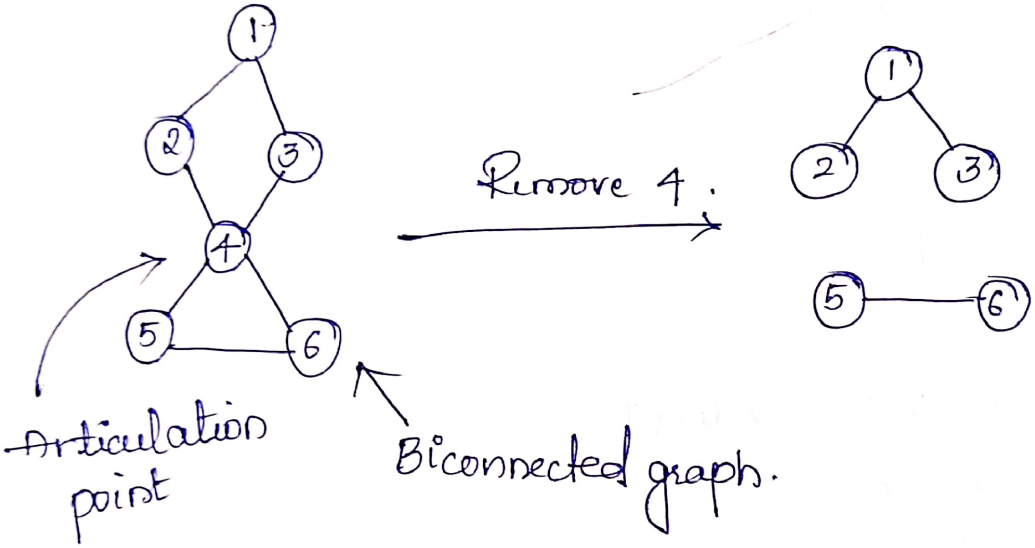
```
Algorithm DEPTH_FIRST_SEARCH(V, E)
{
  for each v∈V
      mark[v] = 0
  for each v∈V
      if(mark[v] = 0) then
          dfs(v)
}
```

Biconnected Components

A Biconnected Components of a graph is a connected subgraph that cannot be broken into disconnected pieces by deleting any single node.

→ An articulation point is a node of a graph whose removal would cause an increase in the number of connected components.

→ If Removal of any node makes a graph into disconnected graph, then such graphs are called Biconnected graph.



UNIT-2 PART-C

DIVIDE & CONQUER

A Divide and Conquer algorithm is a strategy of solving large problem by

1) Breaking the problem into smaller sub problems
2) Solving the sub-problems
3) Combining them to get the desired output.

→ To use divide and conquer recursion is used.

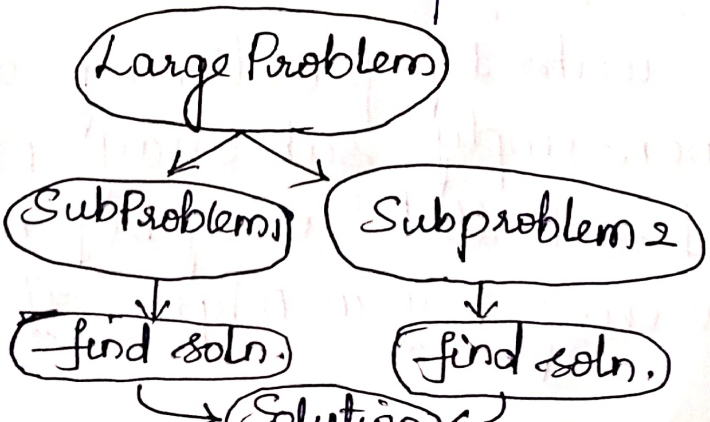
How Divide and Conquer Algorithm work ?

Here are the steps involved

1) Divide :- Divide the given problem into sub-problem using recursion until it is small enough to solve.

2) Conquer :- Solve the smaller subproblems recursively If the subproblem is small enough, then solve it directly.

3) Combine :- Combine the solutions of the sub-problems that are part of recursive process to solve the actual problem.



Topic-1 :- Quick Sort

It is a divide and conquer approach used to sort and arrange the elements in ascending (or) descending order.

Advantage :-

→ It uses only a small auxiliary stack.
→ It requires only (nlogn) times to sort n items.
→ It has extremely short inner loops.
→ This algorithm has been subjected to a through mathematical analysis, a very precise statement can be made about performance issues.

Disadvantages :-

→ It is recursive, especially if recursion is not available, the implementation is extremely complicated.
→ It requires quadratic (i.e, n2) time in the worst - case.
→ It is fragile, i.e a simple mistakes in the implementation can go unnoticed and cause it to perform badly.

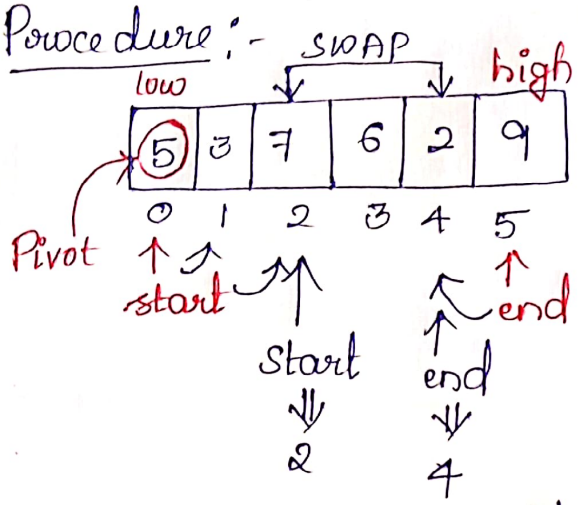
Working Procedure :-

→ Quick sort works by partitioning a given array A[p]. into two non-empty sub array A[p...q] is ~~less~~ and A[q+1...r] such that every key in A[p..q] is lessthan or equal to every key in A[q+1...r].

→ These two sub-arrays are sorted by recursive calls to QuickSort.

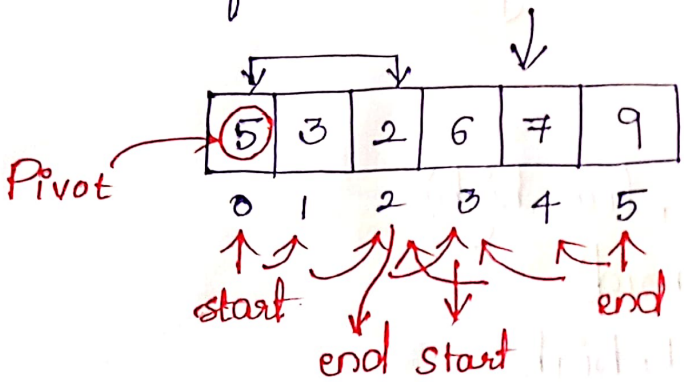
→ The exact position of the partition depends on the given array and index q is computed as a part of partitioning procedure.

Procedure :- SWAP

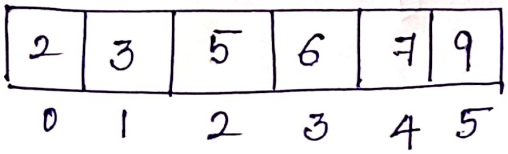



```
while(pivot >= a[start])
{
    start++
}
while(pivot < a[end])
{
    end--
}
```

if start < end then swap(a[start], a[end])

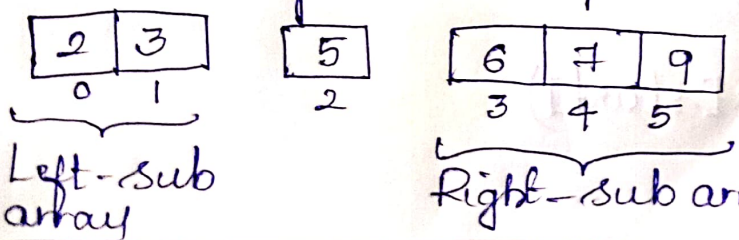


if start > end then swap(pivot, a[end])



→ Here pivot element moved to its appropriate position

→ So divide array as two parts as shown below.



→ Recursively call QuickSort until all elements are sorted.

∴ The final result is

2	3	5	6	7	9

Time Complexity :-

$$T(n) = \begin{cases} 1 & n=1 \\ 2T(\frac{n}{2})+2 & n \geq 2 \end{cases}$$

↓

∴ $O(n \log n)$

Algorithm

```
Algorithm QuickSort(a[], low, high)
{
 if (low < high)
   {
    mid = partitioning(a[], low, high)
    QuickSort(a[], low, mid)
    QuickSort(a[], mid+1, high)
   }
 }
int partitioning(a[], low, high)
{
 start = low
 end = high
 pivot = a[low]
 while (start < end)
 {
  while (pivot >= a[start])
    {
     start++;
```

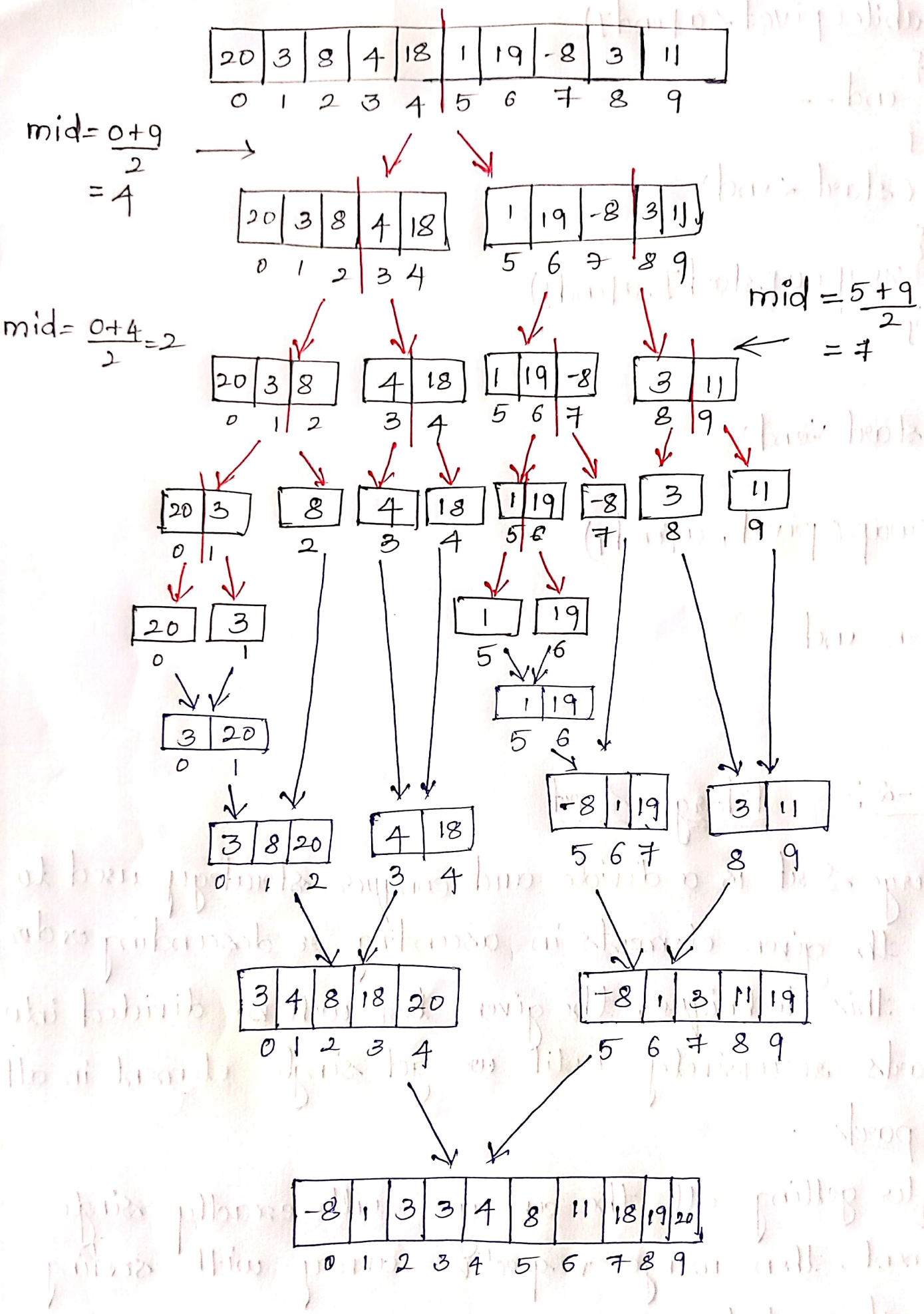
```
while (pivot < a[end])
  {
    end--
  }
if (start < end)
  {
    swap(a[start], a[end])
  }
}
if (start >= end)
  {
    swap(pivot, a[end])
  }
return end
}
```

Topic-2 :- Merge Sort

→ Merge Sort is a divide and conquer strategy used to sort the given elements in ascending or descending order.

→ In this technique, the given list will be divided into 2 parts recursively until we get single element in all the parts.

→ After getting all the arrays with exactly single element, then merge respective array with sorting as shown below.


20 3 8 4 18 1 19 -8 3 11
0 1 2 3 4 5 6 7 8 9
mid = 0+9/2 = 4
20 3 8 4 18
0 1 2 3 4
1 19 -8 3 11
5 6 7 8 9
mid = 0+4/2 = 2
mid = 5+9/2 = 7
20 3 8
0 1 2
4 18
3 4
1 19 -8
5 6 7
3 11
8 9
20 3
0 1
8
2
4
3
18
4
1 19
5 6
-8
7
3
8
11
9
20
0
3
1
1
5
19
6
3 20
0 1
1 19
5 6
3 8 20
0 1 2
4 18
3 4
-8 1 19
5 6 7
3 11
8 9
3 4 8 18 20
0 1 2 3 4
-8 1 3 11 19
5 6 7 8 9
-8 1 3 3 4 8 11 18 19 20
0 1 2 3 4 5 6 7 8 9

Algorithm

```
Algorithm MergeSort(a[], low, high)
{
  if (low < high)
  {
    mid = (low + high) / 2;
    MergeSort(a[], low, mid)
    MergeSort(a[], mid+1, high)
    Merge(a[], low, mid, high)
  }
}

Merge(a[], low, mid, high)
{
  i = low
  j = mid+1
  k = low
  while(i<=mid && j<=high)
  {
    if (a[i]<=a[j])
    {
      b[k] = a[i]
      i++; k++;
    }
    else
    {
      b[k] = a[j]
      j++; k++;
    }
  }

  if(j > high)
  {
    while(j<=high)
    {
      b[k] = a[j];
      j++; k++;
```

```
    }
  if (j > high)
    {
     while (i<=mid)
       {
         b[k] = a[i];
          i++;
          k++;
       }
    }
  for (k=low; k<high; k++)
    {
     a[k] = b[k]
    }
}
```

Time Complexity

$$T(n) = \begin{cases} 1 & n=1 \\ 2T(\frac{n}{2})+2 & n>1 \end{cases}$$

$T(n) = 2T(\frac{n}{2})+2 \longrightarrow (1)$

$T(n) = 2\left[2T(\frac{n}{4})+2\right]+2$

$= 4T(\frac{n}{4})+4+2$

$T(n) = 2^2 T(\frac{n}{2^2}) + 2^2 + 2 \longrightarrow (2)$

$\vdots$

$T(n) = 2^k T(\frac{n}{2^k}) + 2^k + 2^{k-1} + \ldots + 2^2 + 2$

$= 2^k T(\frac{n}{2^k}) + 2^k - 2$

Assume $\frac{n}{2^k} = 1$

$\therefore n = 2^k$

$k = \log n$

$T(n) = 2^{\log n} T(\frac{n}{2^{\log n}}) + \log n \times n$

$= n \log n$

$O(n \log n)$

Topic-3: Strassen's Matrix Multiplication

Let us consider two matrices x and y. We want to calculate the resultant matrix z by multiplying x and y.

1) Iterative Method:-

```
Algorithm MatrixMultiplication (x, y, z)
{
 for i = 1 to p do
   for j = 1 to r do
       z[i,j] = 0
       for k = 1 to q do
          z[i,j] = z[i,j] + x[i,j] * y[j,k]
}
```

Complexity:- $O(n^3)$ ← Time Complexity

2) Strassen's Matrix Multiplication

→ Using this approach the time complexity can be improved (i.e time is reduced).

→ It is a divide and conquer approach used to multiply 2 matrices of size n where n is in terms of 2^n.

→ Here both matrices should be square matrices of size n, where n is multiples of 2^n.

When $n=2$

$$A = \left[\begin{array}{c|c} A_{11} & A_{12} \\ \hline A_{21} & A_{22} \end{array}\right] \qquad B = \left[\begin{array}{c|c} B_{11} & B_{12} \\ \hline B_{21} & B_{22} \end{array}\right]$$

$A \times B \Rightarrow C$

where $C = \begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix}$

When $n=4$

$$A = \left[\begin{array}{cc|cc} A_{11} & A_{12} & A_{13} & A_{14} \\ A_{21} & A_{22} & A_{23} & A_{24} \\ \hline A_{31} & A_{32} & A_{33} & A_{34} \\ A_{41} & A_{42} & A_{43} & A_{44} \end{array}\right] \qquad B = \left[\begin{array}{cc|cc} B_{11} & B_{12} & B_{13} & B_{14} \\ B_{21} & B_{22} & B_{23} & B_{24} \\ \hline B_{31} & B_{32} & B_{33} & B_{34} \\ B_{41} & B_{42} & B_{43} & B_{44} \end{array}\right]$$

$C \leftarrow A \times B$

$$C = \begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix}$$

→ To find the $C_{11}, C_{12}, C_{21}, C_{22}$. Strassen has given these formulas.

$$P = (A_{11} + A_{22})(B_{11} + B_{22})$$
$$Q = (A_{21} + A_{22}) B_{11}$$
$$R = A_{11}(B_{12} - B_{22})$$
$$S = A_{22}(B_{21} - B_{11})$$
$$T = (A_{11} + A_{12}) B_{22}$$
$$U = (A_{21} - A_{11})(B_{11} + B_{12})$$
$$V = (A_{12} - A_{22})(B_{21} + B_{22})$$

$\Rightarrow$

$$C_{11} = P + S - T + V$$
$$C_{12} = R + T$$
$$C_{21} = Q + S$$
$$C_{22} = P + R - Q + U.$$

→ The Time Complexity of this approach using Masters Theorem is

$$\boxed{O(n^{2.81})}$$

Refer class works for example problems.