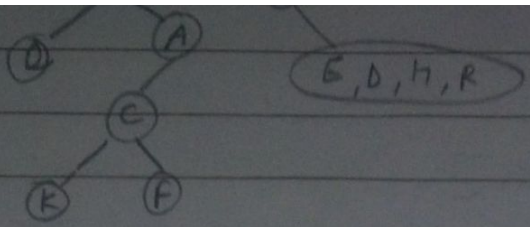
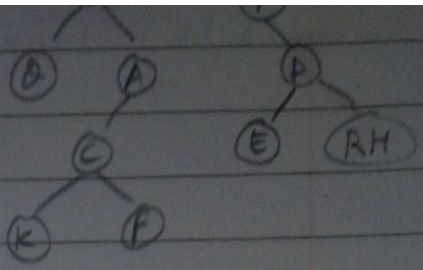
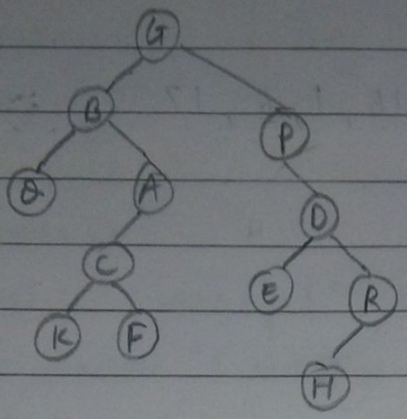


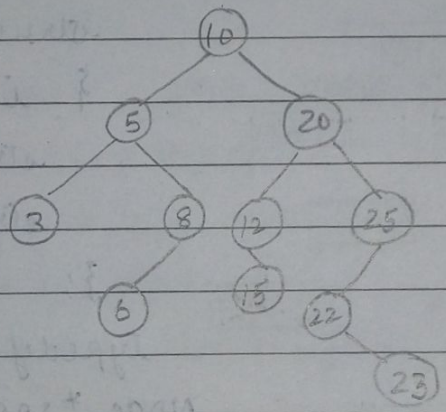


Step 7:



* BINARY SEARCH TREE (B.S.T)

- node value > left subtree values
- node value < right subtree values
- no duplicate values



10 > 5 , 10 < 20

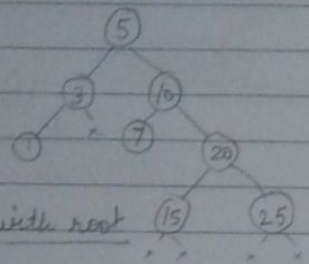
20 > 12 , 20 < 25

5 > 3, 5 < 8

* Binary search tree is a Binary tree in which each node will contain greater value than its values of left subtree and smaller value than values in its right subtree.
this property should be maintained by each subtree of Binary Search tree.
there should be no duplicate values in binary search tree.

- 5, 10, 3, 20, 15, 25, 1, 7: Construct Binary search tree.

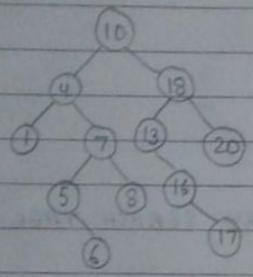
compare with root

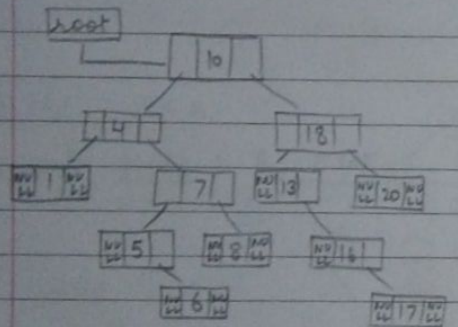


(20) compare with 5 and then with 10.

* always start with root

Q: 10, 18, 4, 7, 20, 5, 13, 8, 16, 1, 6, 17



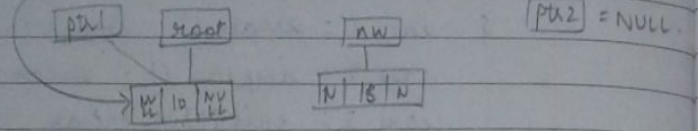


* Binary search tree creation

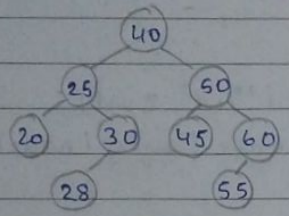
```
struct node
{ int info ;
  struct node *left ;
  struct node *right ;
};
typedef struct node Node ;
Node *root = NULL, *ptr1, *ptr2, *nw;
void create ();
void display ();

void main ()
{ int choice ;
  while (1)
  { printf ("Enter 1 for creation. \n");
    printf ("Enter 2 for displaying. \n");
    printf ("Enter 3 for EXIT. \n");
    printf ("Enter your choice : "); scanf ("%d", &choice);
    switch (choice)
    { case 1 : create (); break ;
      case 2 : display (); break ;
      case 3 : exit (1) ; break ;
      default : printf ("Enter Valid choice !. "):
    } }
}
void create ()
{ int val ; scanf ("%d", &val);
  nw = (node *) malloc (sizeof (node));
  nw -> info = val ;
  nw -> left = NULL ;  nw -> right = NULL ;
  if ( root == NULL)
  { root = nw ;
    nw = NULL ; }
  else
  { ptr1 = root ;  ptr2 = ~~root~~ NULL ;
  while ( ptr1 != NULL)
  { if (ptr1 -> info > val)
    { ptr2 = ptr1 ;
      ptr1 = ptr1-> left ;
    }
    else
    { ptr2 = ptr1 ;
      ptr1 = ptr1 -> right ; }
  }
  if ( ptr2 -> info > val )
  { ptr2 -> left = nw ;  nw = NULL ;
  }
```

```
else
{ ptr2 -> right = nw ;
  nw = NULL ;
}
}
```



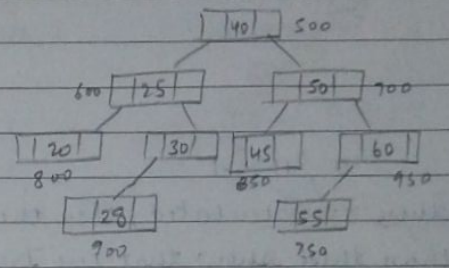
① * searching in Binary search tree

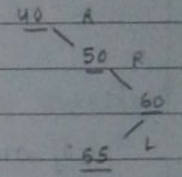



```
void search (
{ node *ptr3,*ptr ;                    // ptr3 = NULL ;
  • if (root == NULL)
    { printf ("Tree is Empty \n"); }   // exit(1);
  • if (root -> info == val)
    { ptr3 = root ; }
  • else if ( val > root -> info)
    { ptr = root -> right ; }
  • else
    { ptr = root -> left ; }
  while (ptr != NULL)
  { - if ( ptr -> info == val)
      { ptr3 = ptr ; break ; }
    - else if ( val > ptr -> info )
      { ptr = ptr -> right ; }
    - else
      { ptr = ptr -> left ; }
```

```
}
• if (ptr3 == NULL)
  { printf (" Element Not Found \n"); }
• else
  { printf (" Node is at %d location \n", ptr3 ); }
}
```

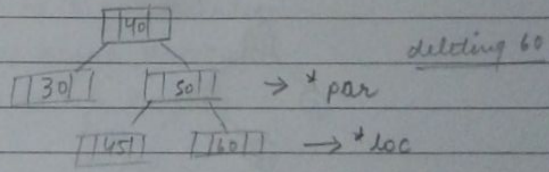
(55) searching





② * deletion in Binary search tree

case 1 : deleting node with no childs.
case 2 : deleting node with one child (left / right)
case 3 : deleting node with two childs

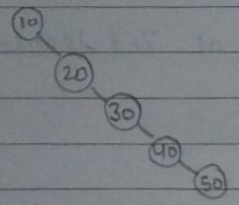


3 : in - order sequence : 20, 30, 35, 40, 41, 42, 45, 50, 60

successor of 50

AVL Height Balance Tree

* Right skew tree : 10, 20, 30, 40, 50



unbalanced Tree : because of only right side.

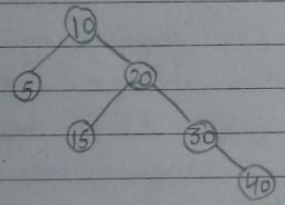
* BALANCE FACTOR : 0, +1, -1

If any of these are calculated, then the tree is balanced else other than these show that the Tree is unbalanced.

* FORMULA :

* $H_L - H_R$ L = left, R = Right

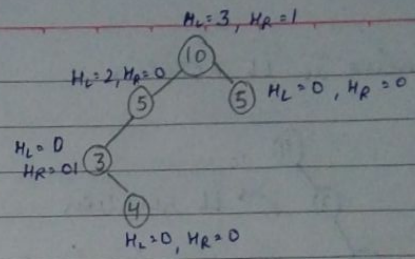
* Height of left sub Tree - ~~left sb~~ Height of Right sub Tree.



10 = $H_L = 1$, $H_R = 3$] 1 - 3 = -2 Unbalanced.

20 = $H_L = 1$, $H_R = 2$] 1 - 2 = -1 Balanced.

30 = $H_L = 0$, $H_R = 1$] 0 - 1 = -1 Balanced



* ROTATIONS : 10, 20, 30

step 1 : 10 0 - 0 = 0

step 2 : 10 0 - 1 = -1 ∴ 10 = -1
20 0 - 0 = 0 20 = 0

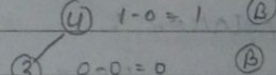
step 3 : 10 0 - 2 = -2 Unbalanced
20 0 - 1 = -1 Balanced
30 0 - 0 = 0 Balanced.

∴ i) Left to Left Rotation
ii) Right to Right Rotation
iii) Left to Right Rotation
iv) Right to Left Rotation.

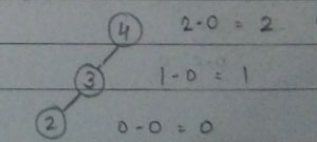
* 1) LL Rotation : 4, 3, 2

step 1 : 4 0 - 0 = 0

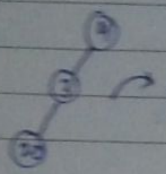
step 2 : 3 is inserted



step 3 : 2 is inserted

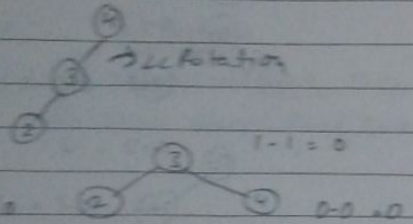


Balancing in LL Rotation :



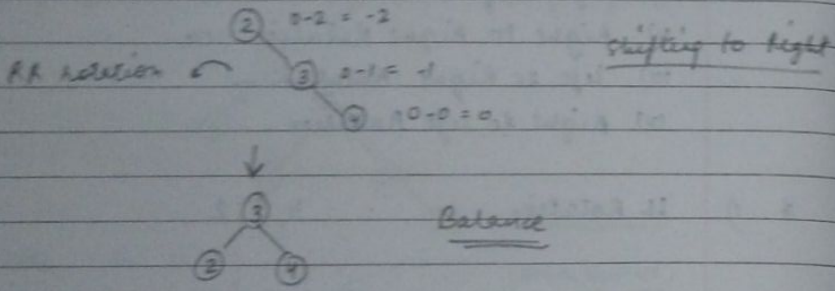
shifting in Right.

if the unbalanced node is in the left of left of left to the root.



Balance.

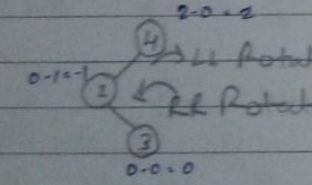
* ii) RR Rotation : 2, 3, 4



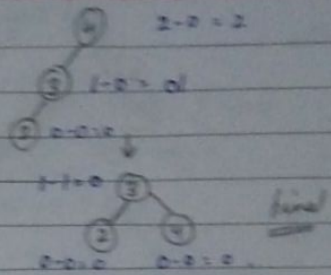
shifting to right

* iii) Left to Right Rotation : 4, 2, 3

step 1 :
step 2 :
step 3 :



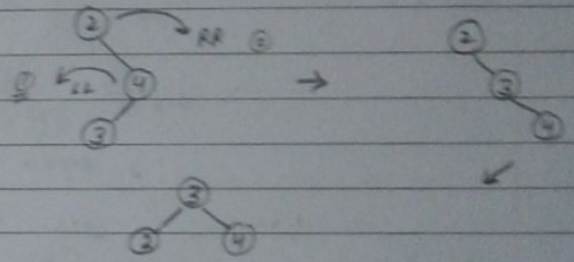
i) first apply RR on child
ii) apply LL on parent



* firstly apply RR rotation onto the child and after that apply LL rotation onto the parent.

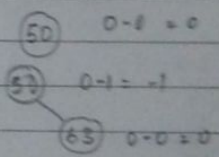
* iv) Right to left Rotation : 2, 4, 3

first apply LL rotation on child then apply RR rotation on the parent.

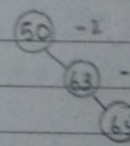


Q: 50, 63, 65, 69, 71, 72, 200, 0, 7 and 68

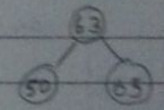
step 1 :
step 2 :



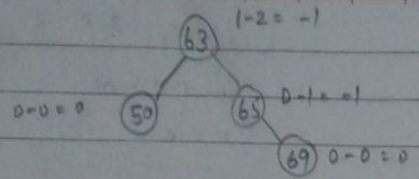
step 3 :



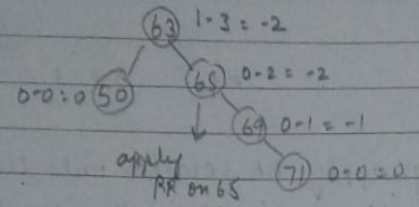
step 4 =



Step 5:

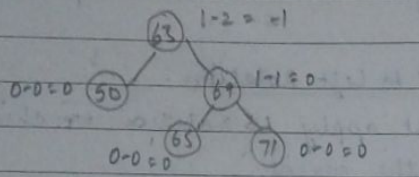


step 6:



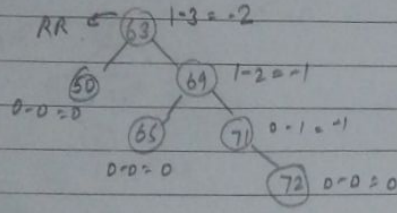
Unbalance

step 7:

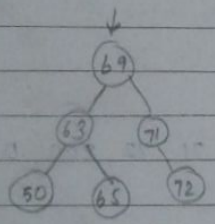


Balance.

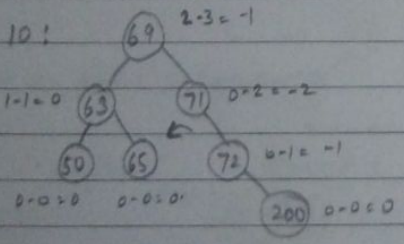
step 8:



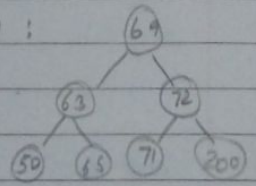
step 9:



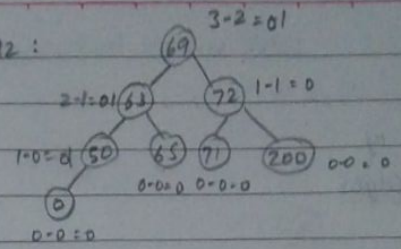
step 10:



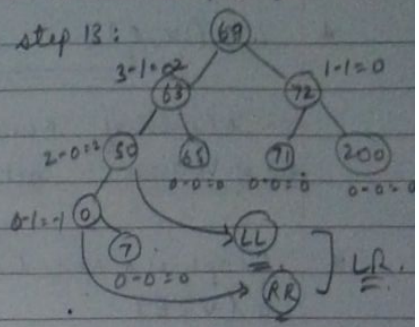
step 11:



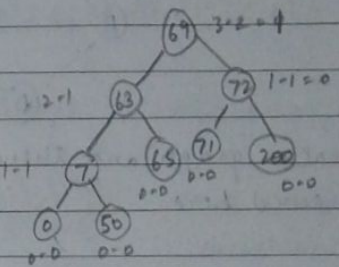
step 12:



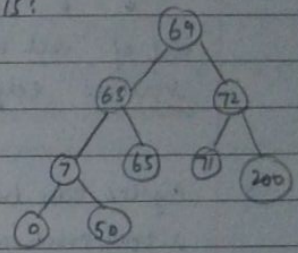
step 13:



step 14:



step 15:

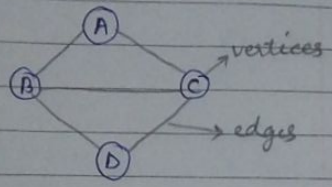


01/2019

GRAPH

* G (V, E)
  - V: Number of set of Vertices
  - E: Number of set of edges

* Graph is a Non-linear data structure which is represented by G (V, E).
  - V: set of vertices
  - E: set of edges

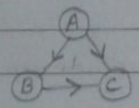


* Cycles can be present in graph and there is no hierarchical relationships followed here.

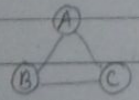
A, B, C, D - vertices
— - edges

* Graph are used mostly in Networking algorithms.
* Graphs can have double edges between two vertices.
* CATEGORIES:

i) **Directed Graph** : Graphs in which direction of every edge is given.
- Unidirectional



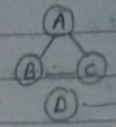
ii) **Undirected Graph** : Graphs in which direction of edges are not given.



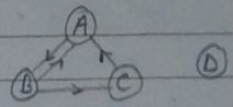
- Bidirectional.

- SOURCE
- S

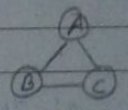
* **Isolated Vertex** :



Isolated vertex

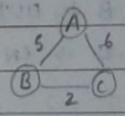


V = { A, B, C, D }
E = { (A,B) (B,A) (B,C) (C,A) }



E = { (A,B) (B,A) (B,C) (C,B) (C,A) (A,C) }

iii) **Weighted Graph** : Every edge has some value with it

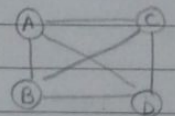


its a weighted graph with un-directional graph (combination).

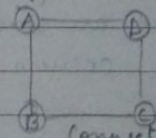
iv) **Complete Graph and Connected Graph**

* PATH and WALK:
  - PATH: vertex can be repeated in this
  - WALK: not has repeated vertex in the way.

There should be any direct edge in between any two vertices.

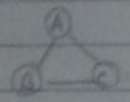


Complete Graph



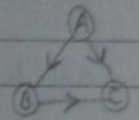
Connected graph.

* Degree of Vertex : degree (vertex)
- no of edges connected to a vertex is known as degree of vertex.



A = 2
B = 2
C = 2

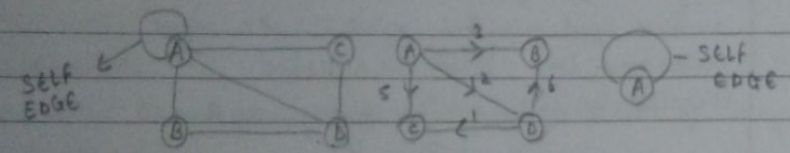
a) INDEGREE : inputs of degrees coming.
b) OUTDEGREE : outputs of degrees going.



A ⇒ IN = 0
OUT = 2
B ⇒ OUT = 1
C ⇒ IN = 2

* Representation of Graph :
Graph is represented using ADJACENCY MATRIX.

1×1 , 2×2 , 3×3 etc. ∴ rows × column



WEIGHT MATRIX : for weighted Matrix graph. both rows and columns are similar / same.
rows = 4, column = 4

if a edge is in between A to A, 01 will be called.
if there is no edge in between then 0 will come.

should contain DIRECT EDGE (Adjacency Matrix)

A×A

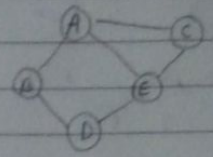
	A	B	C	D
A	0	1	1	1
B	1	0	0	1
C	1	0	0	1
D	1	1	1	0

UNDIRECTED

	A	B	C	D
A	0	1	1	1
B	0	0	0	0
C	0	0	0	0
D	0	1	1	0

DIRECTED

* Ex :



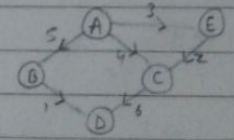
	A	B	C	D	E
A	0	1	1	0	1
B	1	0	0	1	0
C	1	0	0	0	1
D	0	1	0	0	1
E	1	0	1	1	0

UNDIRECTED

PATH MATRIX

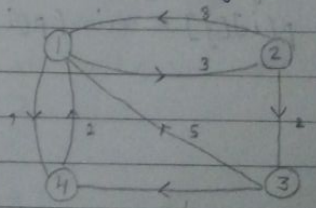
	A	B	C	D	E
A	1	1	1	1	1
B	1	1	1	1	1
C	1	1	1	1	1
D	1	1	1	1	1
E	1	1	1	1	1

UNDIRECTED



A → B → D
A → B 5 5 + 1 = 6
3 + 2 + 6 = 11
A → C → D → B
4 + 6 = 10
A → C → D
> MINIMUM DISTANCE

* WARSHALL's Shortest Path Magic Algorithm :



ADJACENCY MATRIX

A =

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

PATH MATRIX

P_0 =

∞	3	∞	7
8	∞	2	∞
5	∞	∞	1
2	∞	∞	∞

PM

P_1 =

∞	3	∞	7
8	∞	2	(15)
5	(8)	∞	1
2	(5)	∞	∞

PM

P_2 =

∞	3	(5)	7
8	∞	2	15
5	8	∞	1
2	5	(7)	∞

PM

P_3 =

∞	3	5	(6)
(7)	∞	2	(3)
5	8	∞	1
2	5	7	∞

P_4 =

∞	3	5	6
(5)	∞	2	3
(3)	(6)	∞	1
2	5	7	∞

Result (FINAL) Matrix

* No. of Vertices = No. of Path Matrix.

* Step 1: via 1 (calculating Path) (P_1)

2 -> 4 = 2 -> 1 then 1 -> 4
8 + 7 = 15

3 -> 2 : 3 -> 1 then 1 -> 2
5 + 3 = 8

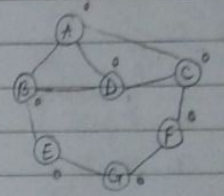
```
for (k=1 ; k<=n ; k++)
 { for (i=1 ; i<=n ; i++)
    { for (j=1 ; j<=n ; j++)
       { p[i][j] = min (p[i][j] , p[i][k] + p[k][j]);
       }
    }
}
```

* parameters should be passed.

```
min ()
{ if ( p[i][j] < p[i][k] + p[k][j])
    { p[i][j] = p[i][j] ; }
  else
    { p[i][j] = p[i][k] + p[k][j] ;
    }
}
```

* TRAVERSING IN GRAPH :

I) Breadth first search (BFS) : through Queue
II) Depth first search (DFS) : through Stack



status = 0 (Not processed)
status = 1
status = 2

I) going level wise
II) going randomly

① status

1) BFS

Step 1: starting vertex is A.

Step 2: | ~~A~~ | ~~B~~ | D | C | E | | inserting into queue.
① and set status to ①

Step 3: Repeat till no vertex's status is 0.

I) Print vertex with status 2.
II) insert all vertex connected to it with status 0.
III) Delete the first vertex.

Output = A -> B -> D -> C -> E -> F -> G
Output

X = Printed / Displayed.

i) A

ii) X B D C

iii) X X D C E

iv) X X X C E

v) X X X X E F

vi) X X X X X F G

vii) X X X X X X G

viii) X X X X X X X

ii) DFS

i)	A	A
ii)	X B D C	A
iii)	X B D X F	A -> C
iv)	X B A X X G	A -> C -> F
v)	X B D X X X E	A -> C -> F -> G
vi)	X B X X X X X	A -> C -> F -> G -> E -> D
vii)	X X X X X X X	A -> C -> F -> G -> E -> D -> B

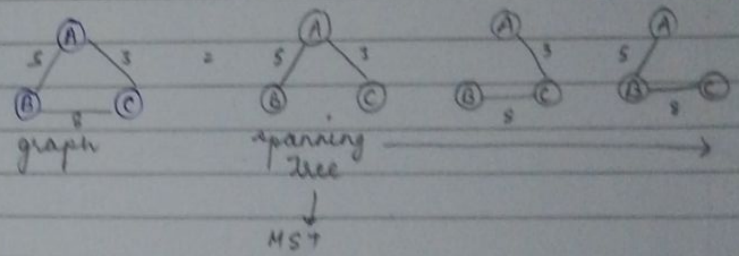
Output

09/11/2019

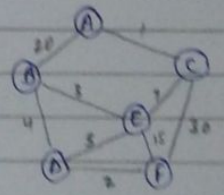
Spanning Tree

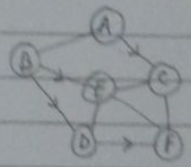
* Minimum Spanning Tree (MST)
  - PRIM's ALGORITHM
  - KRUSKAL's ALGORITHM

* Spanning tree is a special graph in which all vertexes are considered but not necessary that all the edges will be considered.



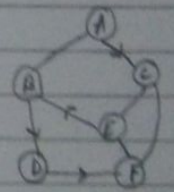
EXAMPLE :





KRUSKAL

A -> C



PRIM

A -> C -> E -> B -> D -> F

minimum

X = Printed / Displayed.

i) A

ii) X B D C

iii) X X D C E

iv) X X X C E

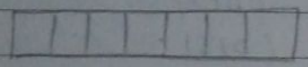
v) X X X X E F

vi) X X X X X F G

vii) X X X X X X G

viii) X X X X X X X

ii) DFS



i)	A	A
ii)	X B D C	A
iii)	X B D X F	A -> C
iv)	X B D X X G	A -> C -> F
v)	X B D X X X E	A -> C -> F -> G
vi)	X B X X X X X	A -> C -> F -> G -> E -> D
vii)	X X X X X X X	A -> C -> F -> G -> E -> D -> B

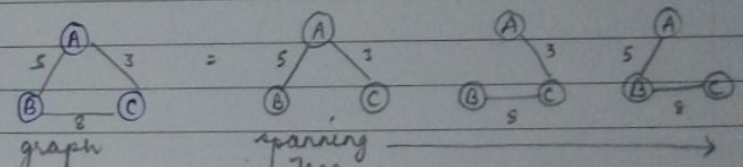
Output

23/11 2019

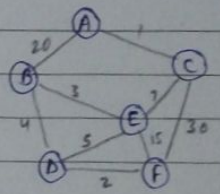
Spanning Tree

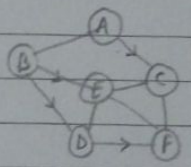
* Minimum Spanning Tree (MST)
  - PRIM's ALGORITHM
  - KRUSKAL's ALGORITHM

* Spanning Tree is a special graph in which all vertexes are considered but not necessary that all the edges will be considered.



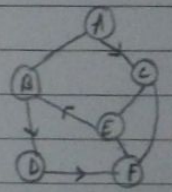
EXAMPLE :





KRUSKAL

A -> C



PRIM

A -> C -> E -> B -> D -> F

minimum