```
else {
    p = front;
    front = front->next;
    free(p);
  }
}
```

Collision Resolution Technique

- Open Addressing
  - linear proping
  - Quadratic proping
  - double
    - ↓ no code
- Closed Hasing
  - ↓ separate

Methods : 4

1. Division
2. Mid - Square
3. folded
4. Multiplication

CO3

1. Hashing
2. Trees
3. Re Hashing
4. Extensible hashing

Hashing — Hash function
Hash methods
collision resolution techniques
linear proping
potratic proping
seperate change
double Hashing.

Trees :

1. Binary search tree (in graphic programming)
2. AVd Tree (code)
3. Splay Tree
4. Expression tree
5. Bi tree
6. Red Black tree
7. Binary hip (hip sort) - C

→ Telecom department

Hashing: Hashing is a process of allocating keys into HashTables (memory) using Hash function is called Hashing.

Eg: keys are some integer values

m stands for size of the table.

H(k) - Hash function.

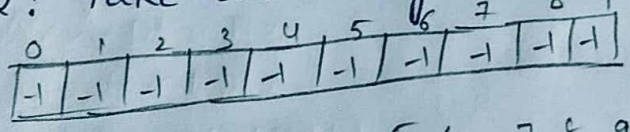
Methods to implement Hashing:

1. Division method
2. Mid-Square
3. folded method
4. Multiplication method

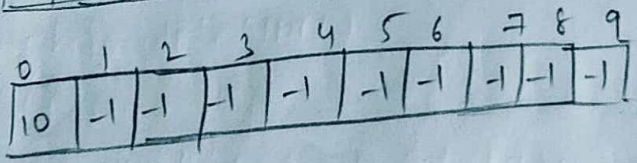
-> Division Method :-> Allocate the following keys 10, 9, 55, 64, 32, 29 into the Hash table with size i.e m = 10 using division Method.

Sol: step-1: The Hash function for division method is given as H(k) = k % m.

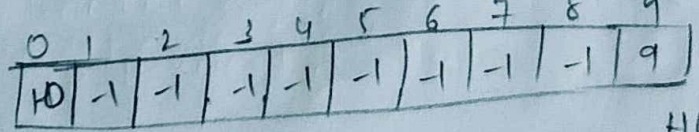
step-2: Take an array



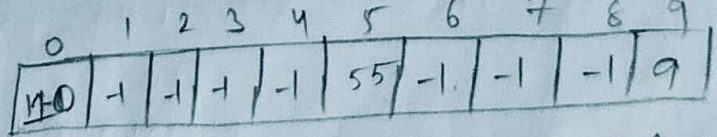
H(10) = 10 % 10
= 0



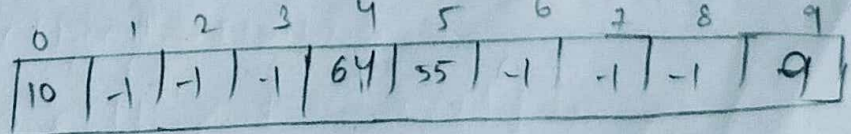
H(9) = 9 % 10
= 9

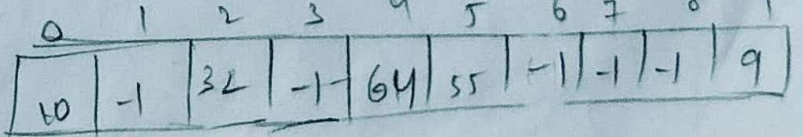


H(55) = 55 % 10



H(64) = 64 % 10





H(32) = 32 %

Q. Mid Square Method:

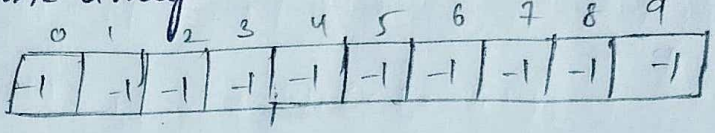
→ Allocate the following keys into Hash table

10, 9, 55, 64, 32, 29

With m = 10 using Mid square method.

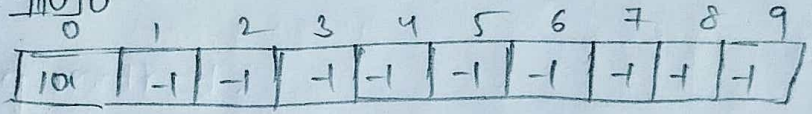
1. The Hash function for mid square method is given as H(K) = K * K.

2. Take the array



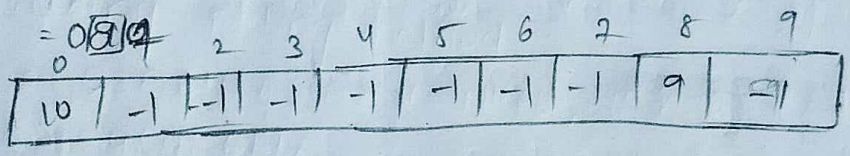
H(10) = 10 * 10

= 100



H(9) = 9 * 9

= 081



H(55) = 55 * 55

= 03025

Collision occured at '0' position

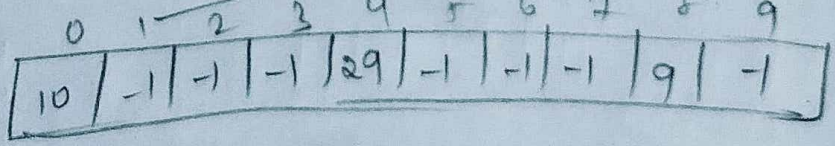
H(64) = 64 * 64 = 04096

Collision occured.

H(32) = 32 * 32 = 01024

Collision occured

H(29) = 29 * 29 = 841

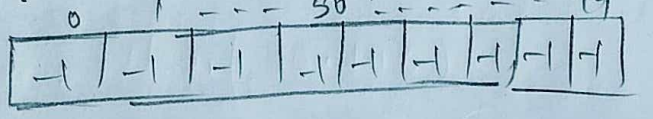


folding method:

→ Allocate the following keys into the Hashtable size 100 using folding method.

8885386142, 9494291146, 9093567812, 6300709101, 9894986142.

step-1:

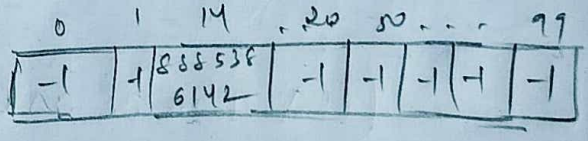


H(8885386142)

2
- 88
- 65
- 38
- 61
- 42

314

ignore 14

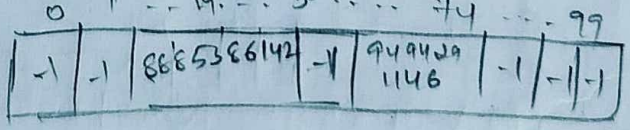


H(9494291146)

2
- 94
- 94
- 29
- 11
- 46

374

ignore



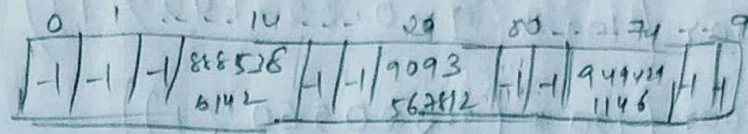
H(9093567812)

ignore

90
93
56
78
12

329

⊠ 29

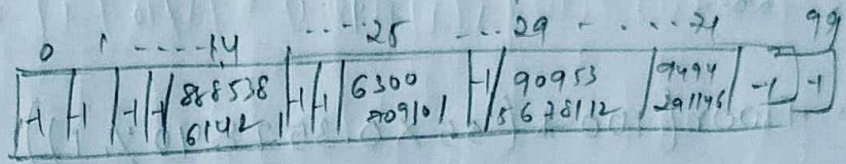


H(6300709101)

63
00
70
91
01

225

⊠ 25



→ Multiplication Method:

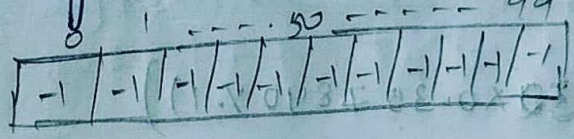
→ Allocate the following keys into Hash table using Multiplication method with m = 100 → 10, 9, 55, 64, 32, 29

the Hash function for multiplication method is given as H(k) = floor(m*(k·A%1))

A is constant value

where, A is 0.357840

Initial array:



step-2: H(10)

floor(100*(10*0.357840%1)

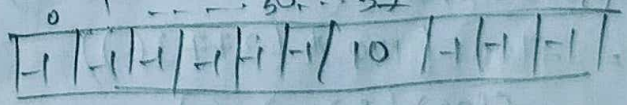
When we do by %1 the remainder part (after point) we will write

floor(100*(3.57840%1))

= 100*(0.57840)

=> floor(57.840)

= 57



$H(9) = \text{floor}(100 * (9 \times 0.357840 \% 1))$

$= \text{floor}(100 * (3.220560 \% 1))$

$= \text{floor}(100 * (0.220560))$

$= \text{floor}(22.0560)$

$= 22.$

0	1	22	50	57	99
-1 -1 -1	9	-1 -1 -1	10	-1 -1 -1 -1 -1	-1

$H(55) = \text{floor}(100 * (55 \times 0.357840 \% 1))$

$= \text{floor}(100 * (19.6812 \% 1))$

$= \text{floor}(100 * 0.6812))$

$= \text{floor}(68.12)$

$= 68$

0	1	22	50	57	68	99
-1 -1 -1 -1	9	-1	10	55	-1	

$H(64) = \text{floor}(100 * (64 \times 0.357840 \% 1))$

$= \text{floor}(100 * (22.90176 \% 1))$

$= \text{floor}(100 * (0.90176))$

$= \text{floor}(90.176)$

$= 90$

0	1	22	57	68	90	99
-1 -1 -1	9	-1 -1 -1	10	55	64	-1

$H(32) = \text{floor}(100 * (32 \times 0.357840 \% 1))$

$= \text{floor}(100 * (11.450880 \% 1))$

$= \text{floor}(100 * (0.450880))$

$= \text{floor}(45.088)$

$= 45$

0	1	22	45	57	68	90	99
-1 -1	9	22	10	55	64	-1	

$H(29) = \text{floor}(100 * (29 \times 0.357840 \% 1))$

$= \text{floor}(100 * (10.37736 \% 1))$

$= \text{floor}(100 * (0.37736))$

$= \text{floor}(37.736) = 37$

0	1	22	37	45	57	68	90
-1 -1	9	29	32	10	55	64	

→ collision Resolution Techniques:

collision: Two keys trying to access the same memory location then we encounter collision.

collision can be resolved in two ways:

(i) open adressing
(ii) closed Hashing

→ open adressing:

(i) linear proping - c
(ii) Quadratic proping - c
(iii) double hashing ×

→ closed Hashing

(i) separate chaining method - c

→ linear proping: The hash function for the linear proping is given as $H(K,i) = (K+i)\%.m$

Allocate the following keys into the hash table with table size 10 and the keys are 29, 9, 99, 5, 65, 11

Sol: Intial array is -1

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

$H(29,0) = (29+0)\%.10$

0	1	2	3	4	5	6	7	8	9
-1	-1	-1	-1	-1	-1	-1	-1	-1	29

$H(9,0) = (9+0)\%.10 = 9.$

collision occurs

increment the value of i

$H(9,1) = (9+1)\%.10 = 0$

0	1	2	3	4	5	6	7	8	9
9	-1	-1	-1	-1	-1	-1	-1	-1	29

$H(99,0) = (99+0)\%.10 = 9$

collision occurs

Increment the value of i

H(99,1) = (99+1) % 10 = 100 % 10 = 0

collision occurs.

H(99,2) = (99+2) % 10 = 101 % 10 = 1

0	1	2	3	4	5	6	7	8	9
9	99	-1	-1	-1	-1	-1	-1	-1	29

H(5,0) = (5+0) % 10 = 5 % 10 = 5

0	1	2	3	4	5	6	7	8	9
9	99	-1	-1	-1	5	-1	-1	-1	29

H(65,0) = (65+0) % 10 = 65 % 10 = 5

collision occurs

increment the value of i.

H(65,1) = (65+1) % 10 = 66 % 10 = 6

0	1	2	3	4	5	6	7	8	9
9	99	-1	-1	-1	5	65	-1	-1	29

H(11,0) = (11+1) % 10 = 12 % 10 = 2

0	1	2	3	4	5	6	7	8	9
9	99	11	-1	-1	5	65	-1	-1	29

Program:

```
#include<stdio.h>
int hashfunction(int ele,int size){
    int key= ele%size;
     return key;
}
int hashfunction2(int ele, int size, int i){
    int key = (ele+i) % size;
     return key;
}
int main(){
   int i,j,table[10], m=10, ele, ke, key;
   for(i=0; i<m; i++){
       table[i]=-1;
```

```
printf("Enter the no.of key elements:\n");
scanf("%d",&ke);
for(j=0;j<ke;j++){
printf("Enter key elements:\n");
scanf("%d",&ele);
key=hashfunction(ele,m);
if(table[key]==-1)
table[key]=ele;
else{
i=1
key=hashfunction2(ele,m,i);
while(table[key]!=-1){
i++;
key=hashfunction2(ele,m,i);
}
table[key]=ele;
}
}
printf("elements in the hash table afting linear
probing:\n");
for(i=0;i<m;i++)
printf("%3d",table[i]);
return 0;
}
```

→ Quadratic proping:

The Hash function for quadratic proping is given by

$H(k,i) = (k + i^2) \% m$

29, 9, 99, 5, 65, 11 with size 10.

SH: $H(29,0) = (29 + 0^2) \% 10 = 29 \% 10 = 9$

0	1	2	3	4	5	6	7	8	9
-1	-1	-1	-1	-1	-1	-1	-1	-1	29

$H(9,0) = (9+0^2) \% 10 = 9 \% 10 = 9$

collision occurs

increment the value of i

$H(9,1) = (9+1^2) \% 10 = (9+1) \% 10 = 10 \% 10 = 0$

0	1	2	3	4	5	6	7	8	9
9	-1	-1	-1	-1	-1	-1	-1	-1	29

$H(99,0) = (99 + 0^2) \% 10 = 99 \% 10 = 9$

collision occurs

increment the value of i.

$H(99,1) = (99 + 1^2) \% 10 = 100 \% 10 = 0$

collision occurs.

$H(99,2) = (99 + 2^2) \% 10 \Rightarrow (99+4) \% 10 = 103 \% 10 = 3$

0	1	2	3	4	5	6	7	8	9
9	-1	-1	99	-1	-1	-1	-1	-1	29

$H(5,0) = (5 + 0^2) \% 10 \Rightarrow 5 \% 10 = 5$

0	1	2	3	4	5	6	7	8	9
9	-1	-1	99	-1	5	-1	-1	-1	29

$H(65,0) = (65 + 0^2) \% 10 \Rightarrow 65 \% 10 = 5$

collision occurs

increment the value of i

$H(65,1) = (65 + 1^2) \% 10 = 66 \% 10 = 6$

0	1	2	3	4	5	6	7	8	9
9	-1	-1	99	-1	5	65	-1	-1	29

H(11,0) = (11+0²) % 10 = 11 % 10 = 1

0	1	2	3	4	5	6	7	8	9
9	11	-1	99	-1	5	65	-1	-1	29

Program:

```
#include<stdio.h>
int hashfunction(int ele, int size){
  int key = ele % size;
  return key;
}
int hashfunction2(int ele, int size, int i){
  int key=(ele+i*i) % size;
  return key;
}
int main(){
  int i, j, n, table[10], m=10, key, ele;
 for(i=0; i<m; i++)
  table[i]=-1;
 printf("Enter the number of elements:\n");
 scanf("%d",&n);
for(j=0; j<n; j++){
  printf("Enter the key elements:\n");
 scanf("%d", &ele);
 key=hashfunction(ele,m);
 if(table[key]==-1)
  table[key]=ele;
 else{
   i=0;
  key=hashfunction2(ele,m,i);
  while(table[key]!=1){
     i++;
  key=hashfunction2(ele,m,i);
 }
  table[key]=ele;
}}
```

```
printf("Elements in the hash table after linear
    probing:\n");
for(i=0;i<m;i++){
    printf("%3d", table[i]);
}
return 0;
}
```

→ Double Hashing:

In double Hashing, we use two hash functions i.e h_1 & h_2.

Whenever we encounter a collision then second hash function comes into the picture.

the hash function is given as ~~$h(K,i) = H_1(K)$~~

$$H(K,i) = (H_1(K) + i * H_2(K)) \% m$$

where,

$$H_1(K) = K \% m$$

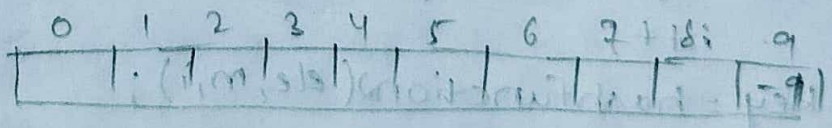
$$H_2(K) = PRIME - (key \% prime)$$

If $m = 10 \Rightarrow 7 - key \% 7$

→ construct a hash table for the following keys with table size 10 and the keys are

29, 9, 99, 5, 65, 11

step-1:



Step-2:

$H(29,0) = 29 \% 10 + 0 * H_2(K) = 9$

0	1	2	3	4	5	6	7	8	9
-1	-1	-1	-1	-1	-1	-1	-1	-1	29

$H(9,0) = 9 \% 10 + 0 * H_2(K)$

$= 9 + 0 = 9$

collision occured

$H(9,1) = (9 \% 10 + 1 * (7 - 9 \% 7) \% 10)$

$= (9 + 1 * 5) \% 10$

$= 14 \% 10 = 4$

0	1	2	3	4	5	6	7	8	9
-1	-1	-1	-1	9	-1	-1	-1	-1	29

$H(99,0) = 99 \% 10 + 0 *$

$= 99 \% 10 = 9$

collision occured

99
2
92

$H(99,1) = (99 \% 10 + 1 * (7 - 99 \% 7) \% 10)$

$(9 + 1 * (7 - 1) \% 10)$ $9 + 1 * ($ $92 \% 10 = 2$

$= 9 + 1 * (5) \% 10$ $9 + 1 * (92) \% 10$

$= 9 + 5 \% 10 = 14 \% 10 =$ 94

$H(99,2) = (99 \% 10 + 2 * (7 - 99 \% 7) \% 10)$

$= 9 + 2 * ($

$H(99,1) = (99 \% 10 + 1(7 - 1)) \% 10$

$= 9 + 6 \% 10$

$= 15 \% 10 = 5$

0	1	2	3	4	5	6	7	8	9
-1	-1	-1	-1	9	99	-1	-1	-1	

$H(5,0) = 5 \% 10$

$= 5$

collision occurs

$H(5,1) = (5 \% 10) + 1 * (7 - (5 \% 7) \% 10)$

$= (5 + 1 * (7 - 5) \% 10)$

$= 5 + 1 * (2) \% 10)$

$= 5 + 2 \% 10$

$= 7 \% 10$

$= 7$

0	1	2	3	4	5	6	7	8	9
-1	-1	-1	-1	9	99	-1	5	-1	-1

$H(65,0) = 65 \% 10$

$= 5$

collision occured

$H(65,1) = (65 \% 10) + 1 * (7 - (65 \% 7) \% 10)$

$= 5 + 1 * (7 - 2) \% 10$

$= 5 + 1 * (5) \% 10$

$= 5 + 5 \% 10$

$= 10 \% 10 = 0$

$H(11,0) = 11 \% 10 = 1$

0	1	2	3	4	5	6	7	8	9
-1	11	-1	-1	9	99	-1	5	-1	-1

-> Separate chaining:

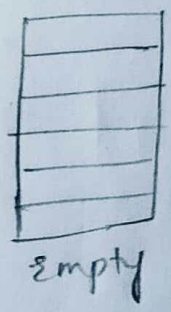
It

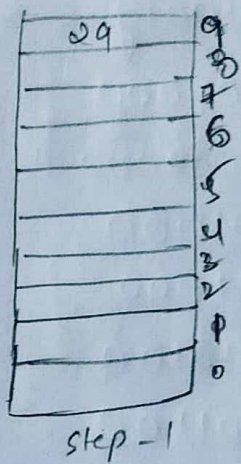
(i) In separate chaining we allocate the keys into hash table using linked list.

-> Allocate the following keys in the H.T with table size 10 with H(k) is given as H(K) = K%.n)

the keys are 29, 9, 99, 12, 72, 54, 34.

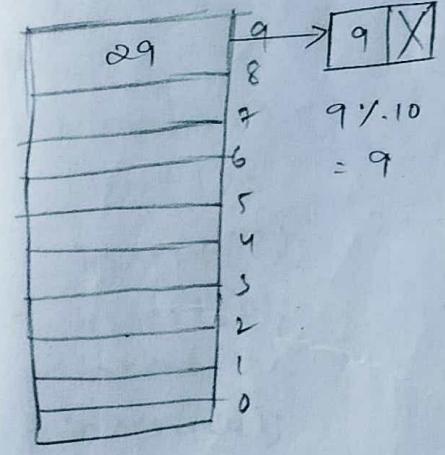
Sol:





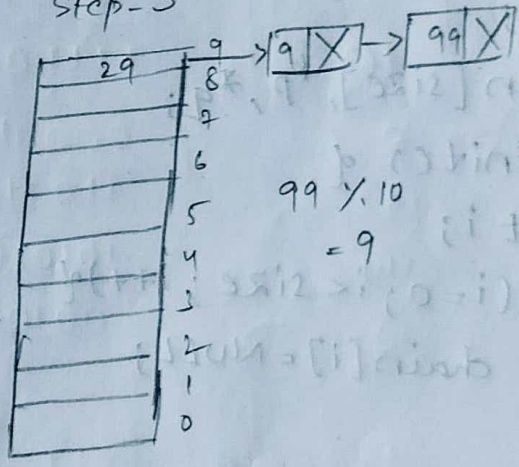
H(29) = 29%.10
= 9

Step-2



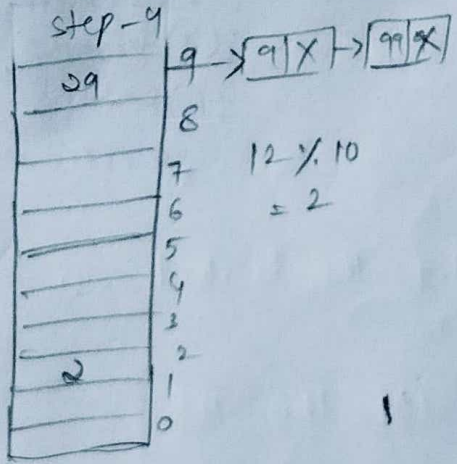
9%.10
= 9

Step-3



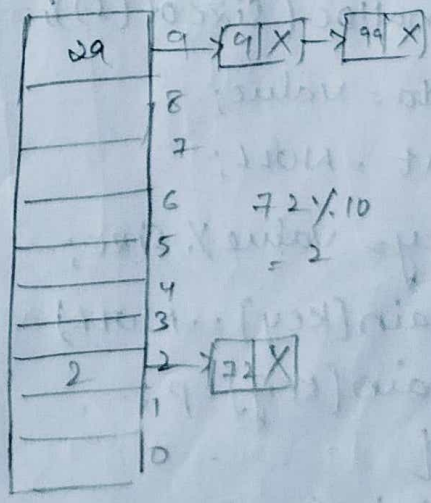
99 %.10
= 9

Step-4



12 %.10
= 2

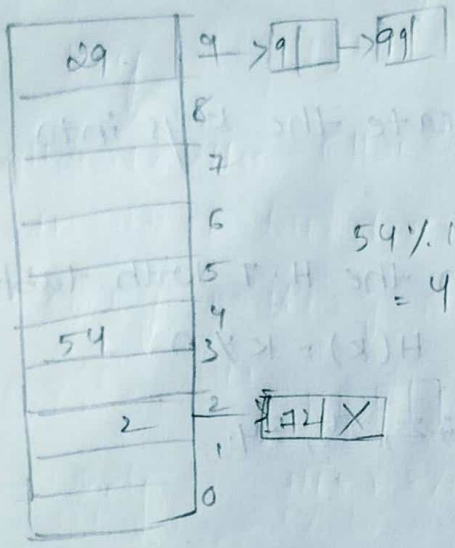
step 5



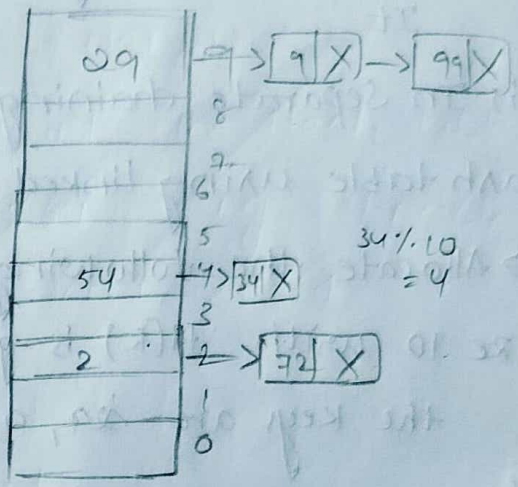
72 %.10
= 2

Collision occured

Step-6



Step-7 :




```
#include<stdio.h>
# include<stdlib.h>
# define size 7
  typedef struct node {
        int data;
        struct node *next;
    };
s*chain[size], *p, *q;
 void init() {
     int i;
     for(i=0; i<size; i++){
         chain[i] = NULL;
 }
 void insert(int value){
   p=(s*) malloc (sizeof(s));
     p->data = value;
     p->next = NULL;
     int key = value % size;
      if(chain[key] == NULL)
          chain[key] = p;
      else {
       q = chain[key];
```

```
while(q->next!=NULL){
    q=q->next;
  }
  q->next=p;
 }
}
void print(){
   int i;
for(i=0;i<size;i++){
    p=chain[i];
  printf("chain[%d]-->",i);
for(p=chain[i];p!=NULL;p=p->next)
   {
  printf("%d-->",p->data);
  }
  printf("NULL\n");
   }
}
int main(){
    init();
   insert(8);
   insert(0);
   insert(16);
  insert(24);
  insert(57);
  ;insert(4);
  insert(5);
    print();
  return 0;
}
```

-> Extensible Hashing:

1. Key H(K) Binary

->Allocate the following keys into the Hash table with Hash function H(K) = K%64 using Extensible Hashing with q=2 and Bucket size = 4

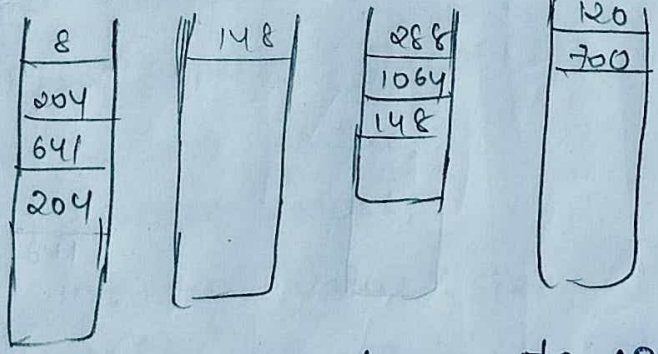
1. 288, H(1

Key	H(k)=K%64	Binary
288	32	100000
8	8	001000
1064	40	101000
120	56	111000
148	20	010100
204	12	001100
641	1	000001
700	60	111100
258	2	000010
1586	15	110010
44	44	101010
10	10	001010

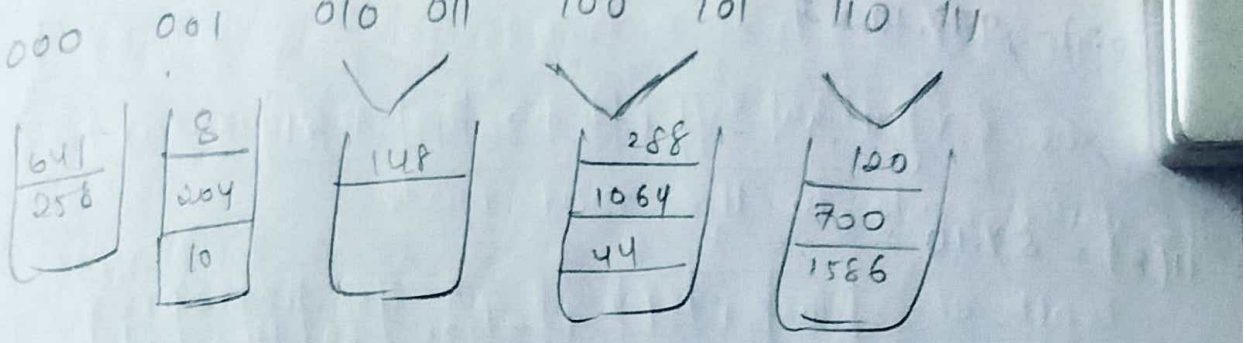
0)64(8
64
0

16 8 4 2 1

step-1: 00 01 10 11



Now we have to split the bucket 00 i.e q=3.



→ ReHashing:

- Whenever the Hash table is almost all full and still some keys are left over to Hash in such situation we go for rehashing

i.e we double the table size.

Eg: 75, 86, 97, 8, 19, 24

and $m = 5$.

step 1:

0	1	2	3	4
-1	-1	-1	-1	-1

step 2: $H(k) = k \% m$.

$H(75) = 75 \% 5$

$= 0$

0	1	2	3	4
75	-1	-1	-1	-1

$H(k) = k \% m \Rightarrow H(86) = 86 \% 5$

$= 1$

0	1	2	3	4
75	86	-1	-1	-1

$H(97) = 97 \% 5 = 2$

75	86	97	-1	-1

now, Double the table size with $5 \times 2 = 10$

$H(75) = 75 \% 10 = 5$

0	1	2	3	4	5	6	7	8	9
-1	-1	-1	-1	-1	75	-1	-1	-1	-1

$H(86) = 86 \% 10 = 6$

0	1	2	3	4	5	6	7	8	9
-1	-1	-1	-1	-1	75	86	-1	-1	-1

H(97) = 97 % 10 = 7

0	1	2	3	4	5	6	7	8	9
-1	-1	-1	-1	-1	75	86	97	-1	94

H(8) = 8 % 10 = 8

0	1	2	3	4	5	6	7	8	9
-1	-1	-1	-1	-1	75	86	97	8	-1

H(19) = 19 % 10 = 9

0	1	2	3	4	5	6	7	8	9
-1	-1	-1	-1	-1	75	86	97	8	19

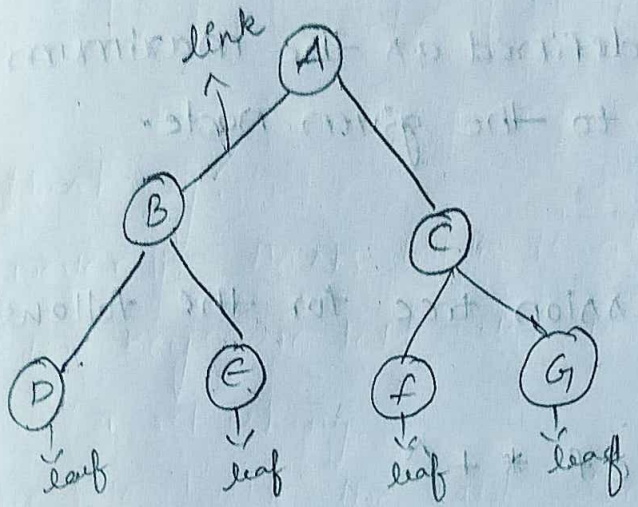
H(24) = 24 % 10 = 4

0	1	2	3	4	5	6	7	8	9
-1	-1	-1	-1	24	75	86	97	8	19

Trees:

Tree: A Tree is a non linear (graph) data structure

→ collection of nodes and link is called a tree.



→ Binary tree:

A Binary tree is divided into three subsets

(i) Root

(ii) left subtree

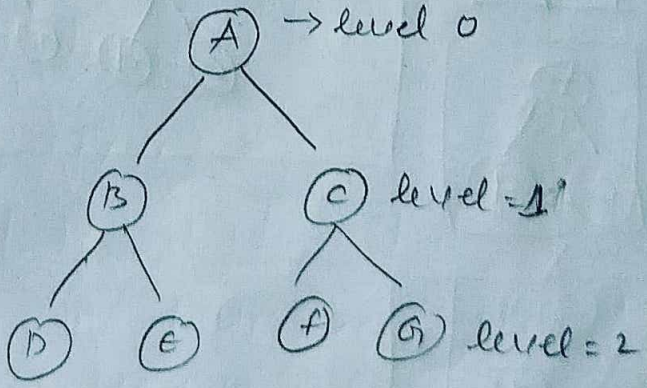
(iii) Right subtree

→ Leaf: A node which doesn't have children.

→ Siblings: Two nodes having same parent are called sibilings.

BC, DE, fG are siblings.

→ Level of a node: The level of a node in defined as one + its parent level



→ Height and depth of a tree:

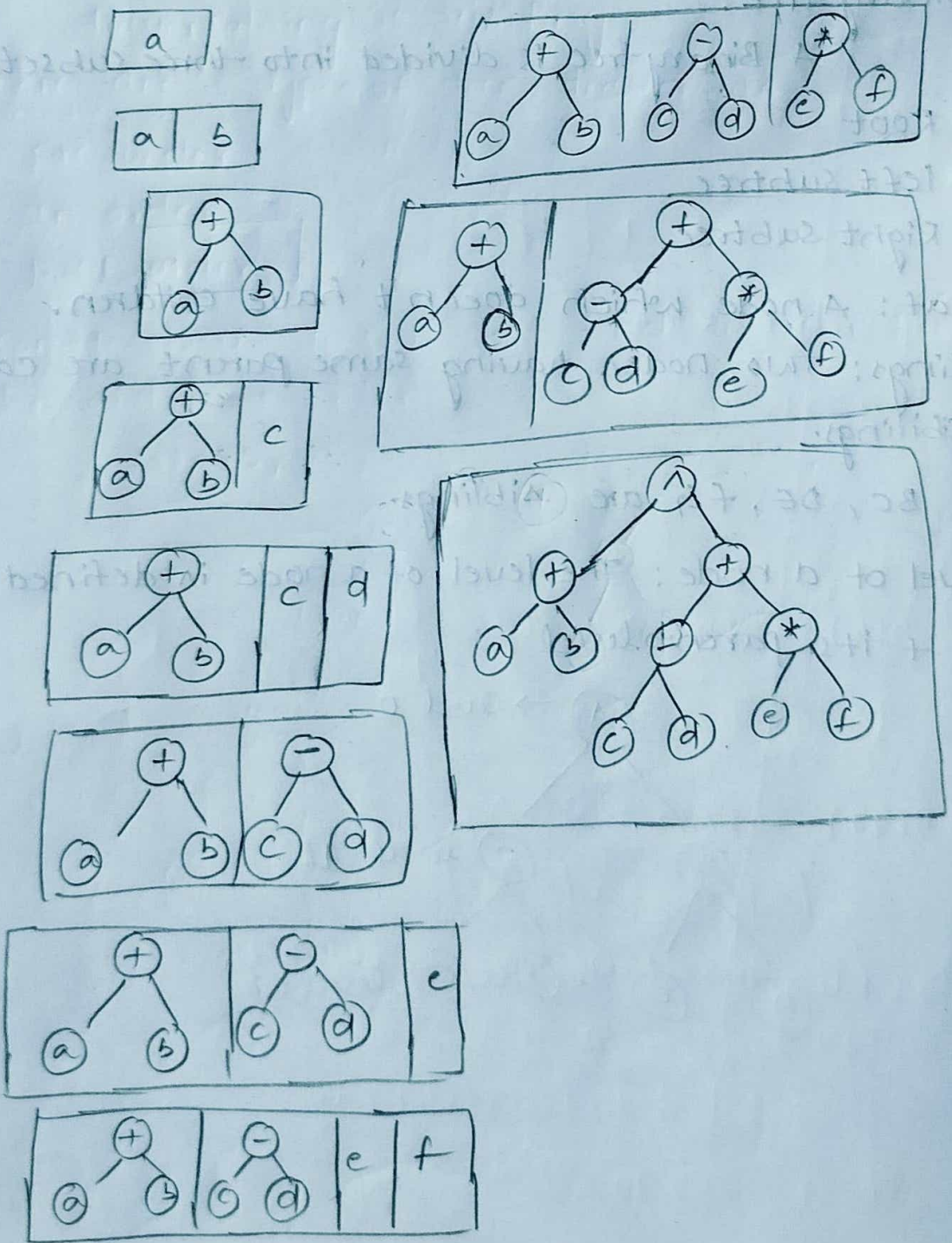
The height is defined as the maximum path from the given node to the root node.

Depth of a tree defined as the maximum path from the root node to the given node.

1. Expression tree :-

Construct an expression tree for the following expression.

a b + c d - e f * + ^



2. Binary Search tree:

A binary Search tree is a binary tree in which the first element is the root node and if the next element is less than the root node then place that node as left child of the of the root node

if it is greater than the root node then place it as right child of the root node.

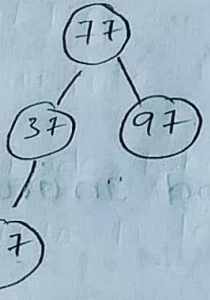
-> construct a B.S.T for the following.

77, 37, 97, 57, 8, 68, 78.

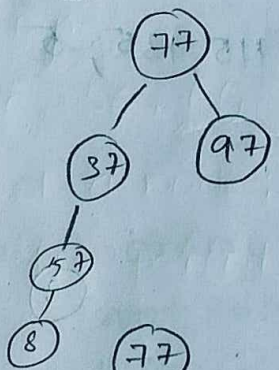
1.

2.

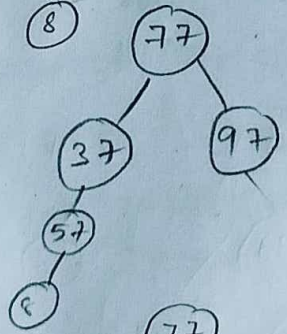
3.



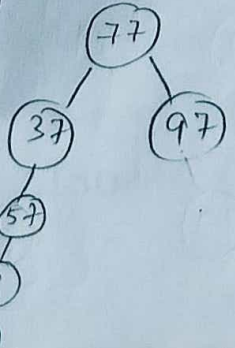
4.



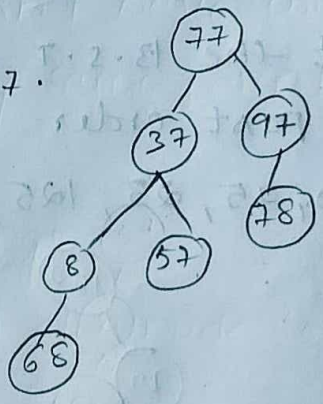
5.



6.

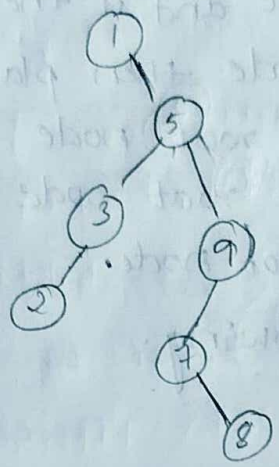


7.



→ Construct a BST for the following

1, 5, 3, 9, 7, 9, 8



Tree traversals:

traversaling is visiting.

There are 3 types of tree traversals:

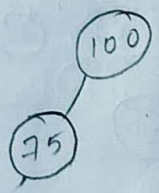
pre order Root Left Right

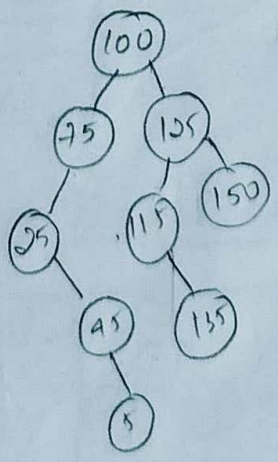
In order Left Root Right

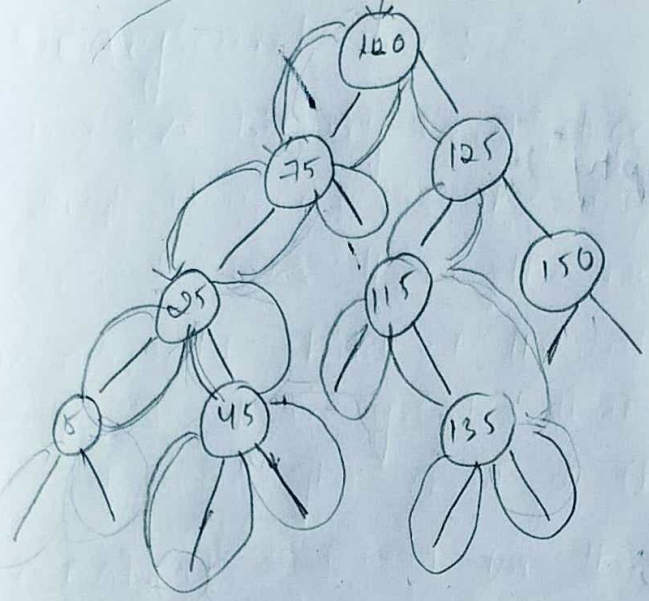
post order Left Right Root

→ Construct the B.S.T and find In order, Pre order and post order.

100, 75, 25, 125, 150, 45, 115, 135, 5







pre : 100, 75, 25, 5, 45, 125

In : 5, 25, 45, 75, 100

post : 5, 45, 25, 75, 135, 115

Recursive function for pre order:

```
Void preOrder() {
   if(root == NULL) {
printf("Tree is Empty");
else {
  Pf("%d ---> ", T->data);
  if ( T->l != NULL)
      preOrder(T->l);
  if (T->R != NULL)
     preOrder(T->R);
   }
 }
}
Void Inorder() {
    if(root == NULL) {
  printf("Tree is Empty");
  else {
    if(T->l != NULL)
       Inorder(T->l);
  printf("%d ---> ", T->data);
  if( T->R != NULL)
     Inorder ( T->R);
   }
 }
```

```
void postOrder() {
    if(root == NULL) {
     printf("Tree is Empty");
    else {
      if(T->L != NULL)
        preOrder(T->L);
      if(T->R != NULL)
        preOrder(T->R);
      printf("%d --->", T->data);
    }
  }
}
```

-> Deleting a node in BST:

→ AVL TREE:

AVL stands for Adelson Velsky Landas.

AVL tree is the balancing tree.

Whenever, we insert a node or delete a node the tree gets imbalanced.

In order to make tree balanced we four types of rotations

We say AVL tree is balanced if the balance factor must be in between {-1, 0, 1}.

A balance factor of an AVL tree is given as height of the left subtree - height of the right subtree.

$$B_f = H_L - H_R$$

In order to make tree balanced.

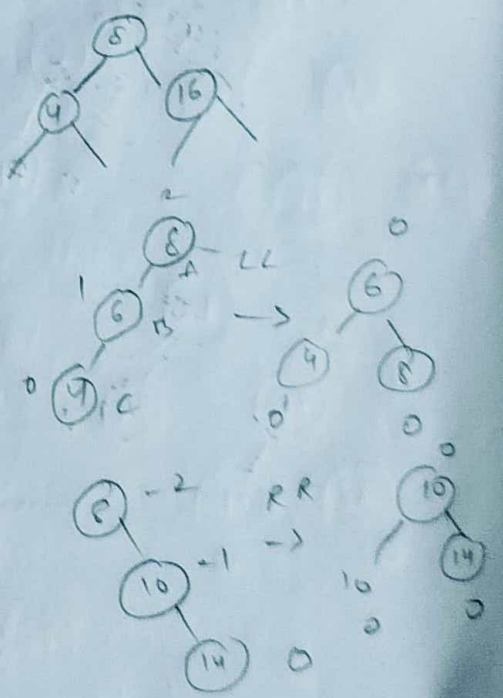
The four rotations are:

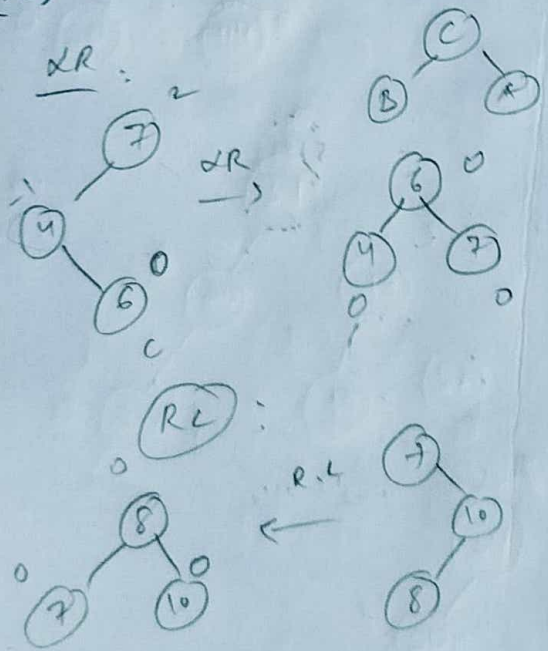
(i) LL - rotation

(ii) RR rotation

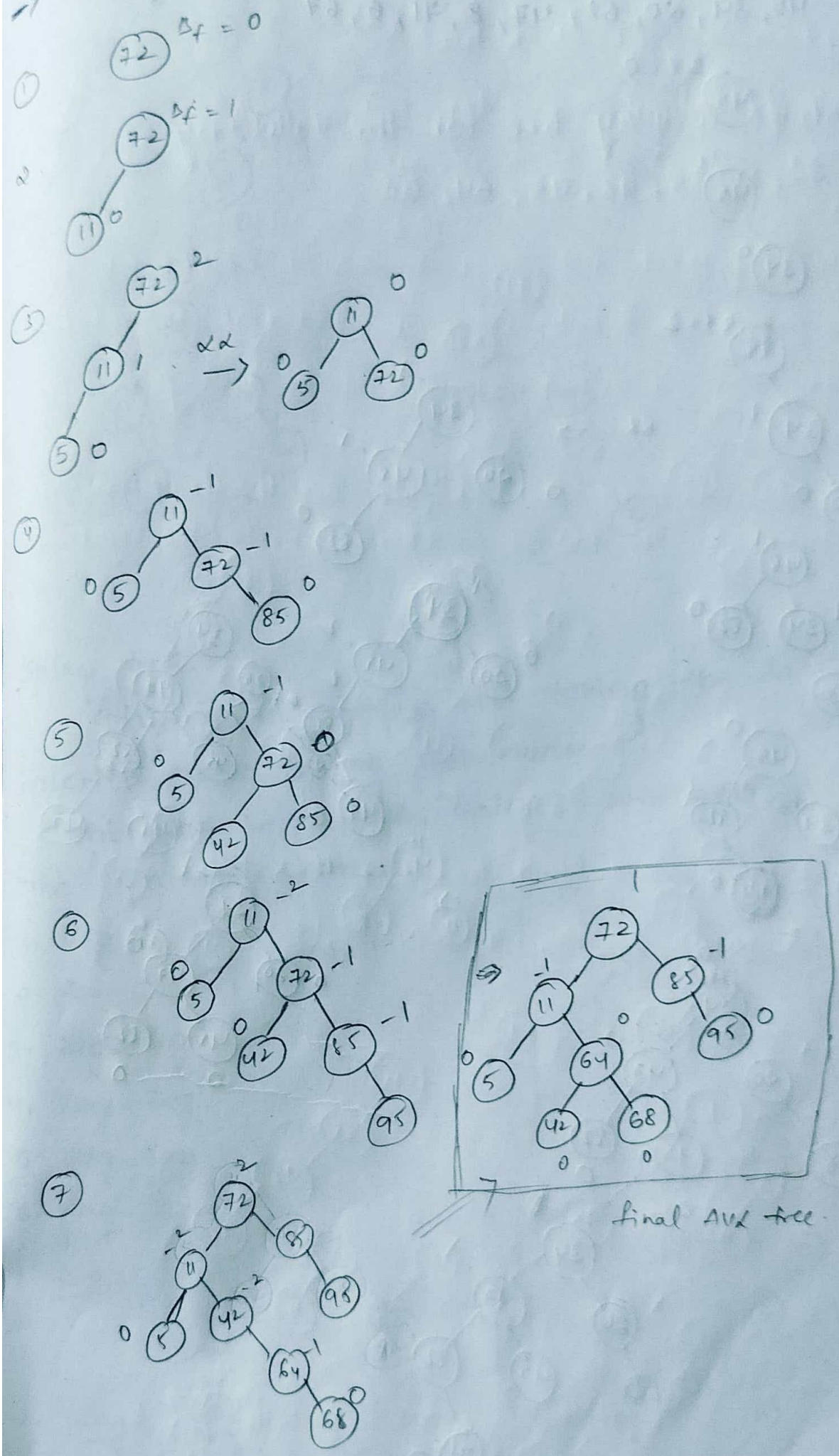
(iii) LR rotation

(iv) RL rotation



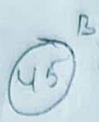


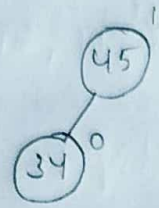
→ 72, 11, 5, 85, 42, 95, 64, 68

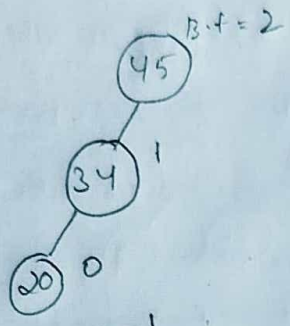


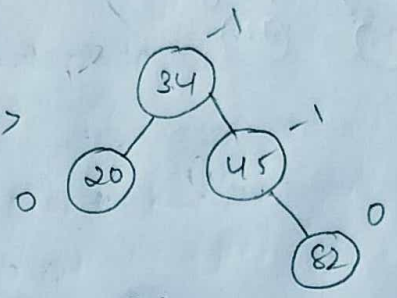
→ construct the AVL tree for the following:

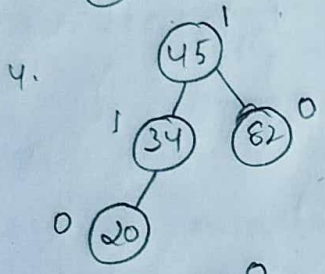
45, 34, 20, 82, 47, 8, 71, 6, 67

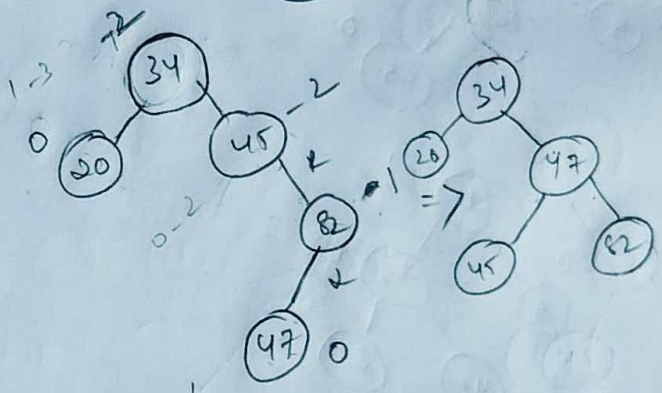
1. 

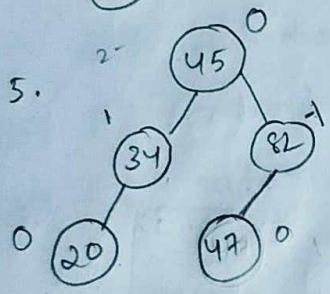
2. 

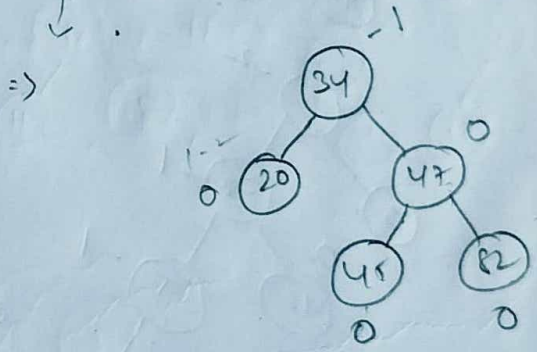
3. 

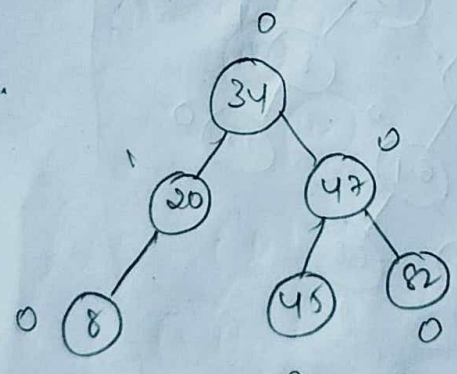
LL => 

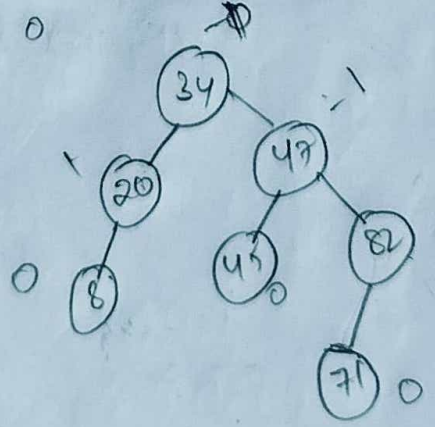
4. 

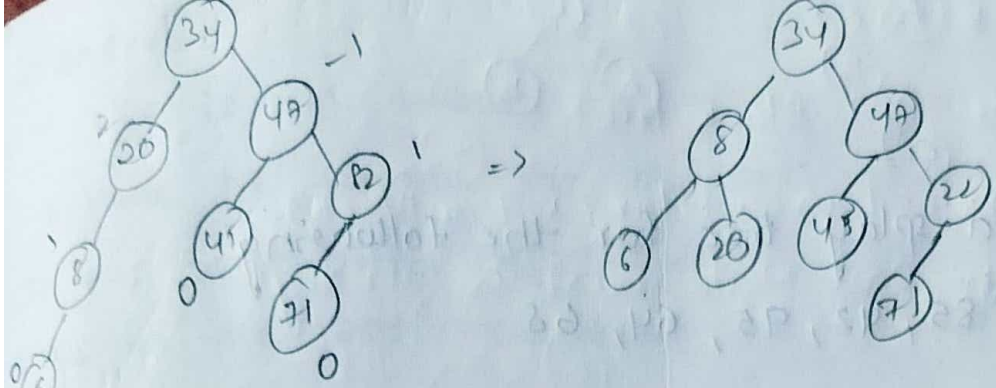


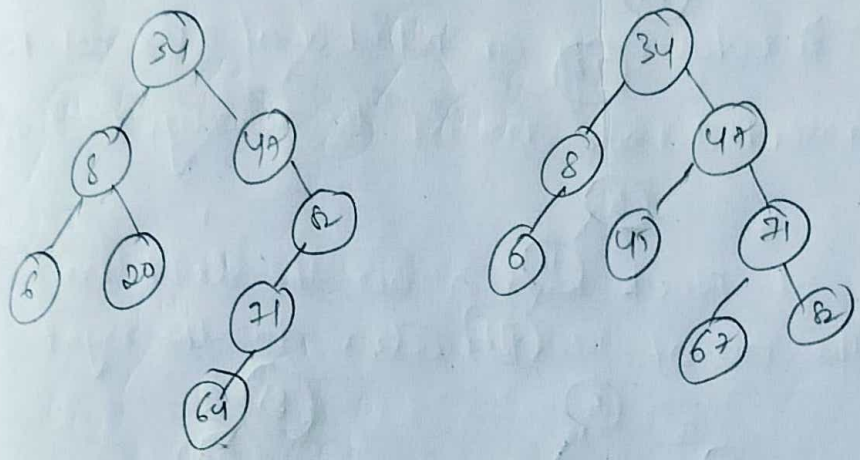
5. 



6. 







Splay tree:

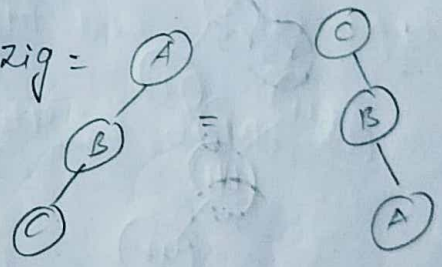
Splaying is nothing but taking the last inserted node to the root node.

This can be done by using 6 rotations:

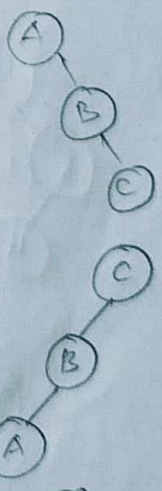
The 6 rotations are:

1. Zig
2. Zag
3. Zig - Zig
4. Zag - Zag
5. Zig - Zag
6. Zag - Zig

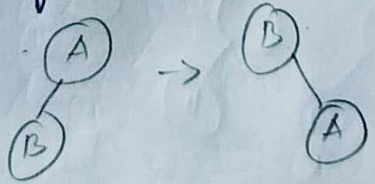
3. Zig - Zig :



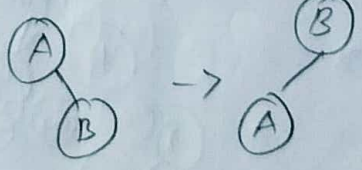
4. Zag - Zag :



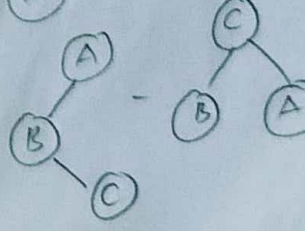
1. Zig :



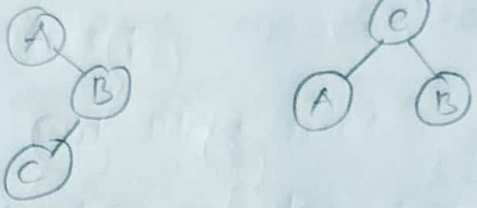
2. Zag :



5. Zig - Zag :

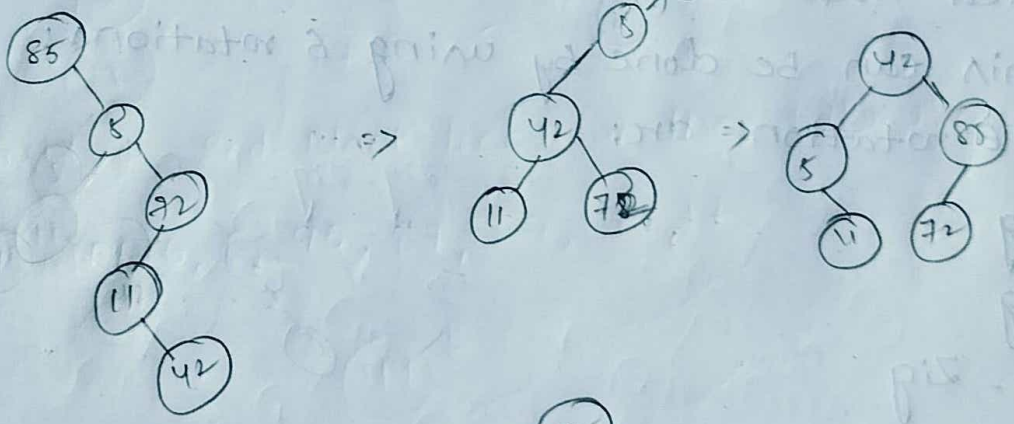


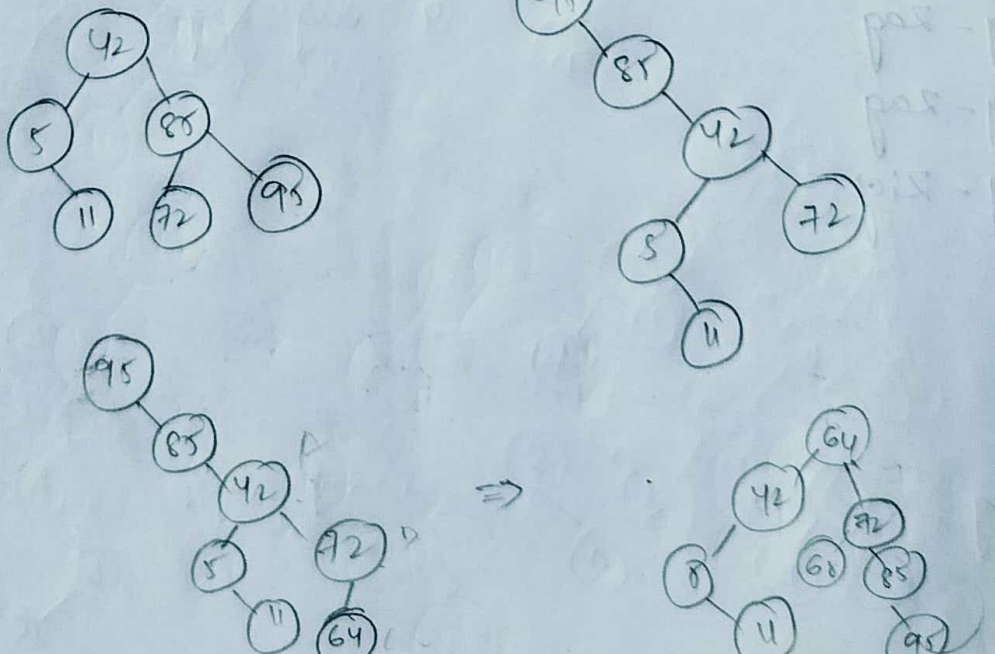
Zag - Zig :



→ construct a splay tree for the following

72, 11, 5, 85, 42, 95, 64, 68





-> B-tree:

It is also known as M way tree.

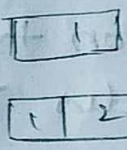
Where M stands for the order of the tree.

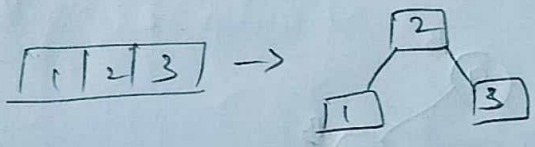
If we have order m(3) then we can have m-1 keys in the root node and m children.

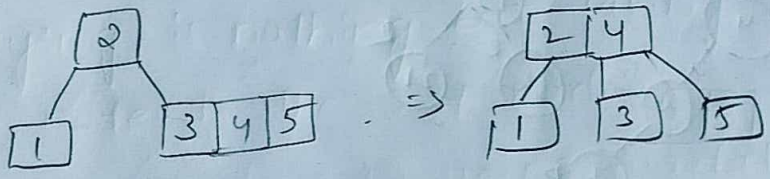
-> construct a B-tree for the following with order 3

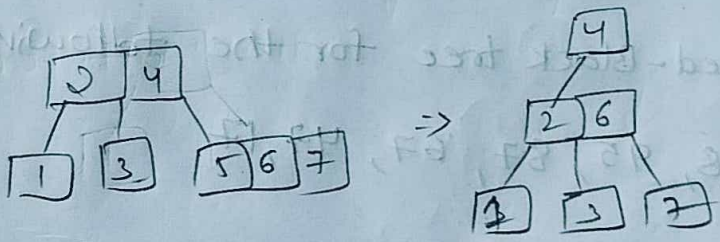
1, 2, 3, 4, 5, 6, 7, 8, 9

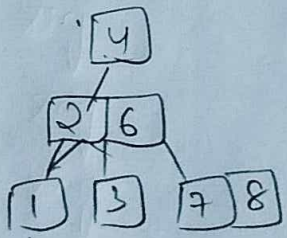
->

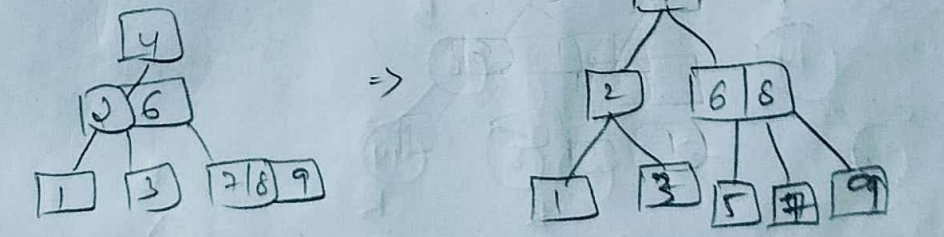












→ Red-Black tree:

Properties of Red-Black tree:

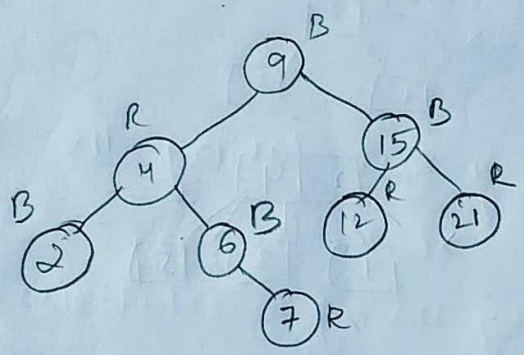
(i) Every node has a color (Red or Black).

(ii) (Every) The Root Node must be always Black in colour.

(iii) No, two adjacent red colour nodes. if exists we have (ivr to perform re colouring or rotations (AVL rotations)

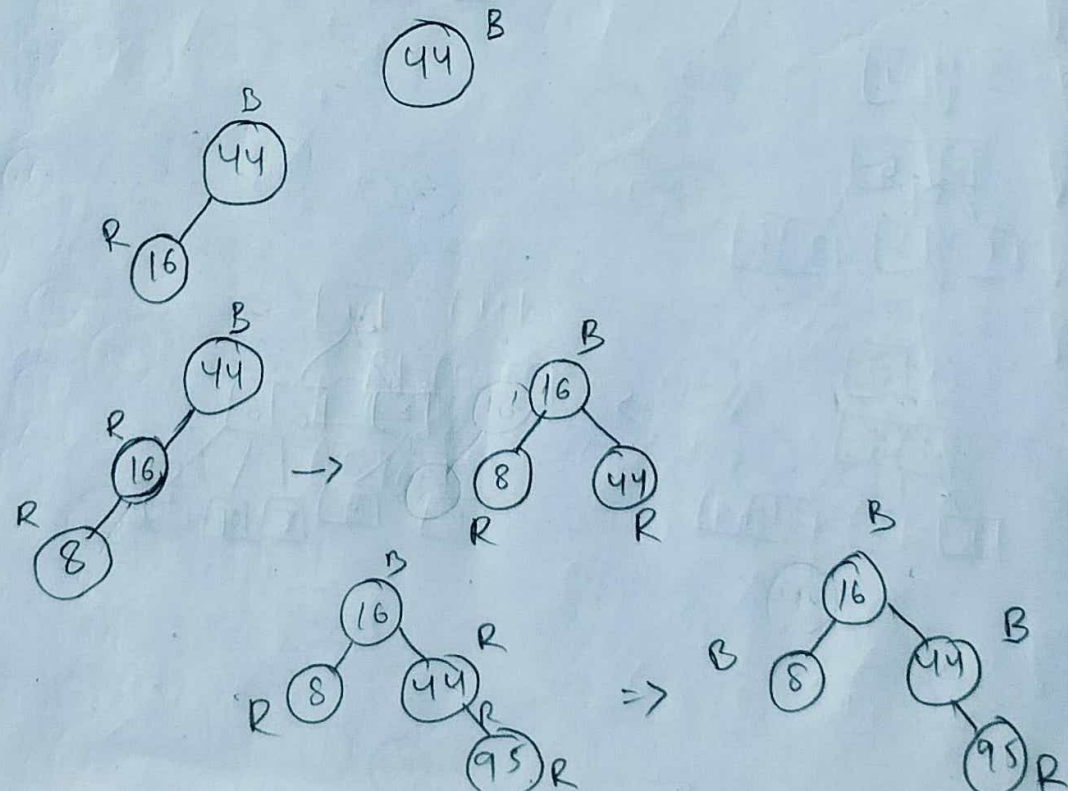
(iv) The last inserted node must be always Red in colour.

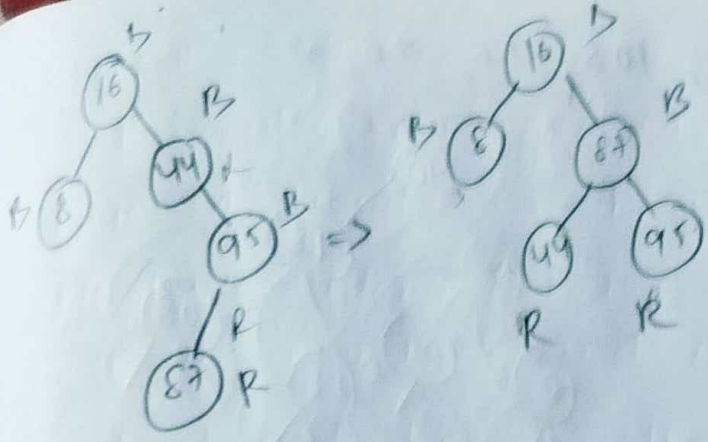
(v) The number of Black nodes on the left must be equal to the no. of Black nodes to the right.

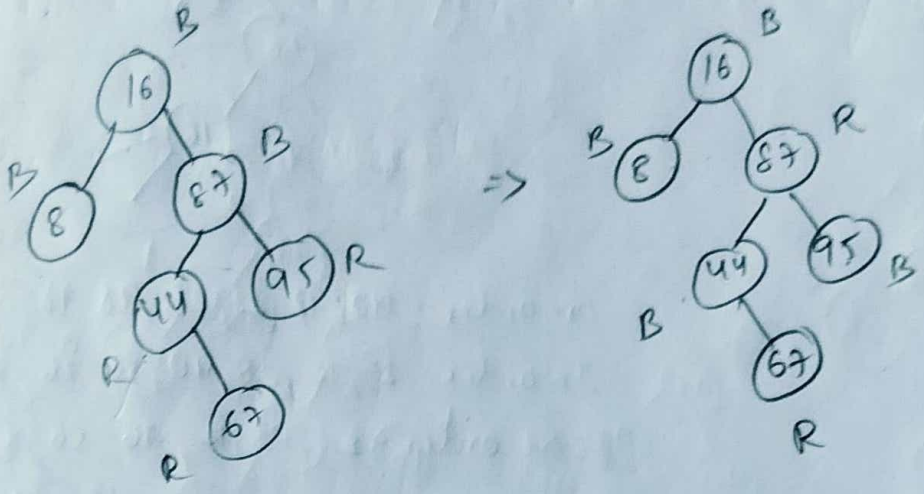


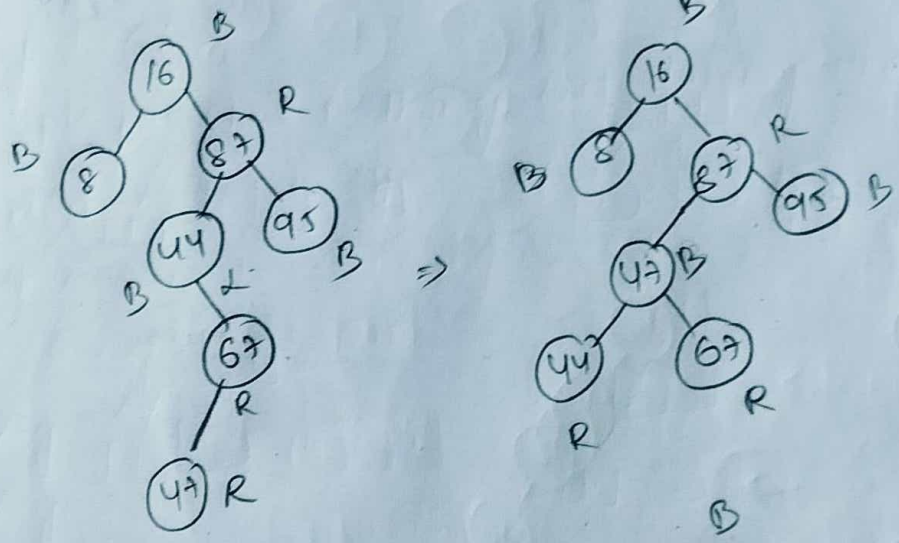
→ Construct a Red-Black tree for the following elements

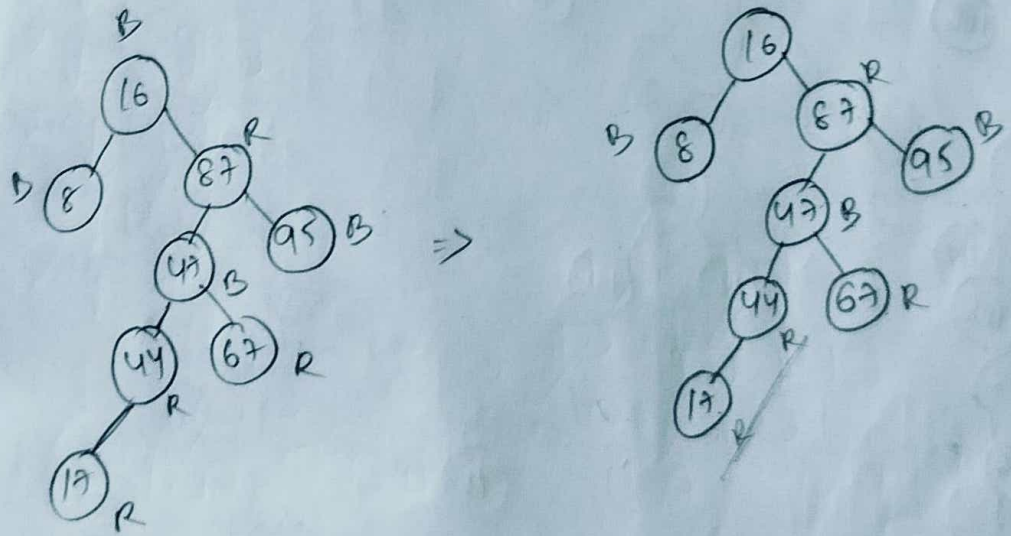
44, 16, 8, 95, 87, 67, 47, 17




16
8
44
95
87
=>
16
8
87
44
95

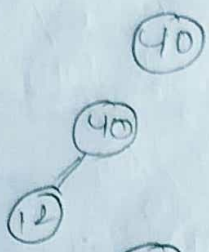

16
8
87
44
95
67
=>
16
8
87
44
95
67

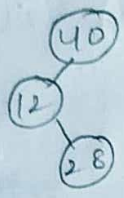

16
8
87
44
95
67
47
=>
16
8
87
44
95
67
47

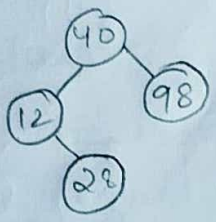

16
8
87
44
95
67
47
17
=>
16
8
87
44
95
67
47
17

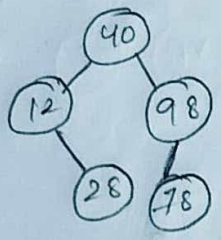
→ 40, 12, 28, 98, 78, 108, 56, 76, 2.

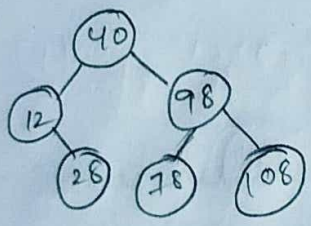
Binary Search Tree:

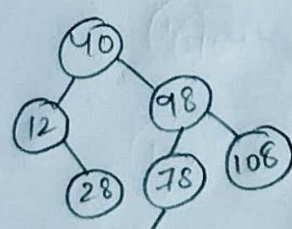


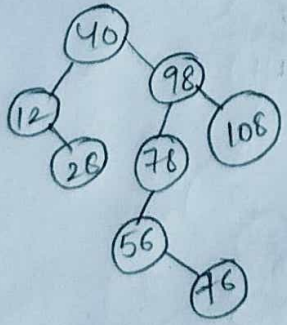


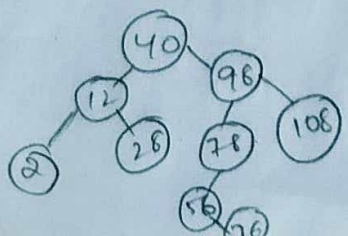


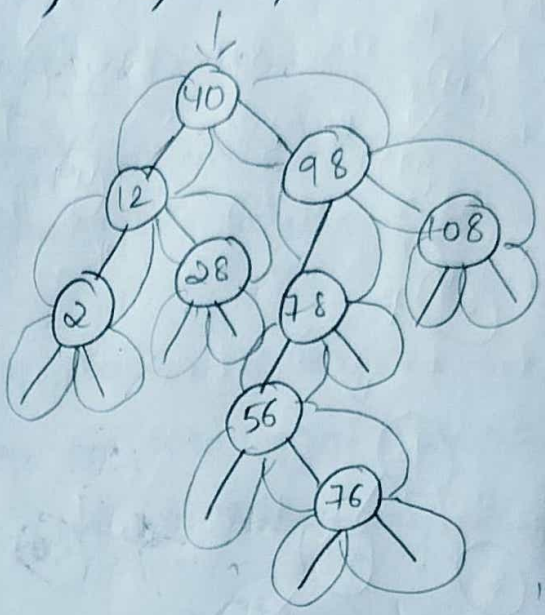












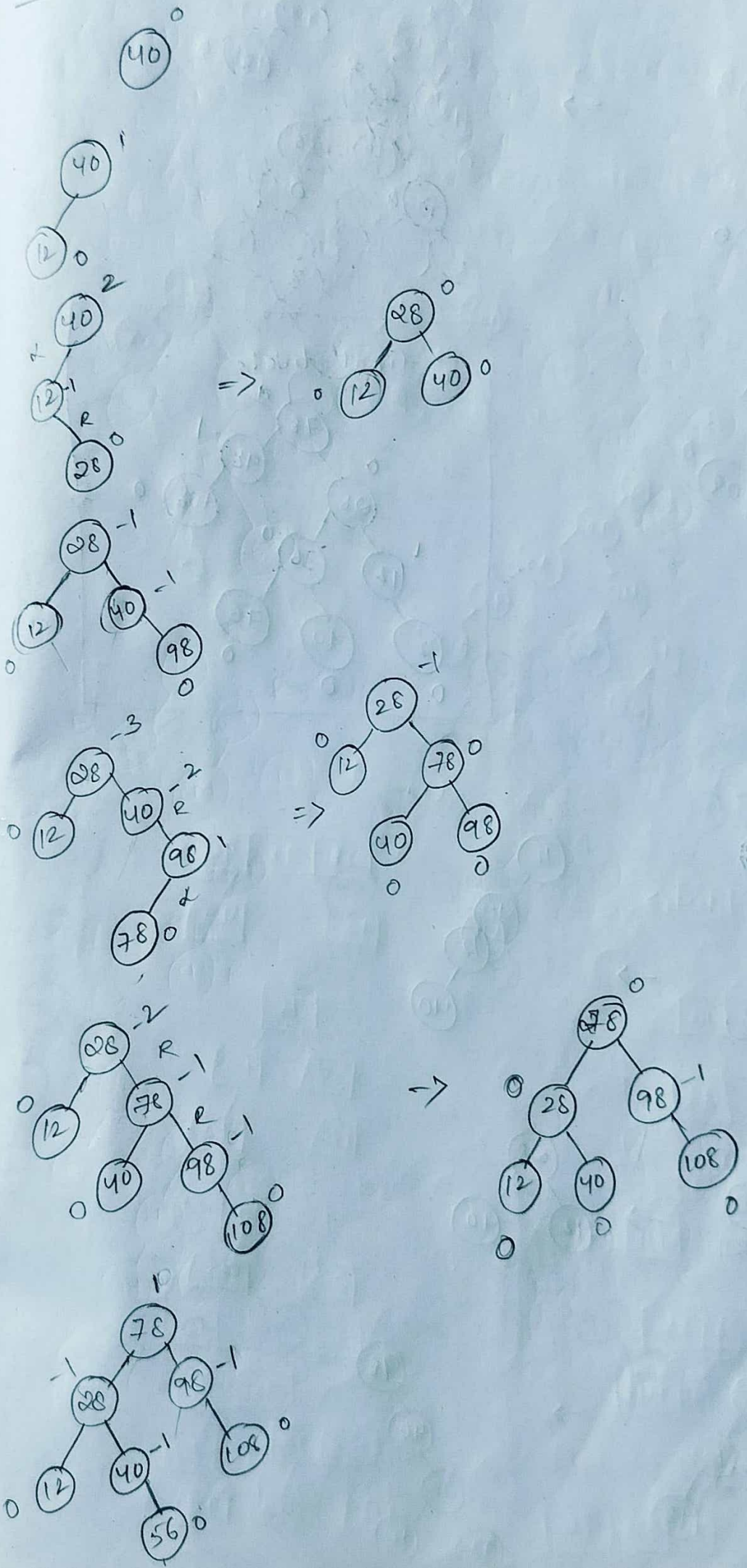
108

preorder: 40, 12, 2, 28, 98, 78, 56, 76, 108

Inorder: 2, 12, 28, 40, 56, 76, 78, 98, 108

Post order: 2, 28, 12, 76, 56, 78, 108, 98, 40

AVL Tree :


40
40
12
40
12
R
28
=>
28
12
40
28
12
40
98
26
12
78
40
98
28
12
40
98
78
=>
26
12
78
40
98
28
12
40
98
108
->
78
28
98
12
40
108
78
28
98
12
40
108
56

Graphs

→ Heap Sort : — Code

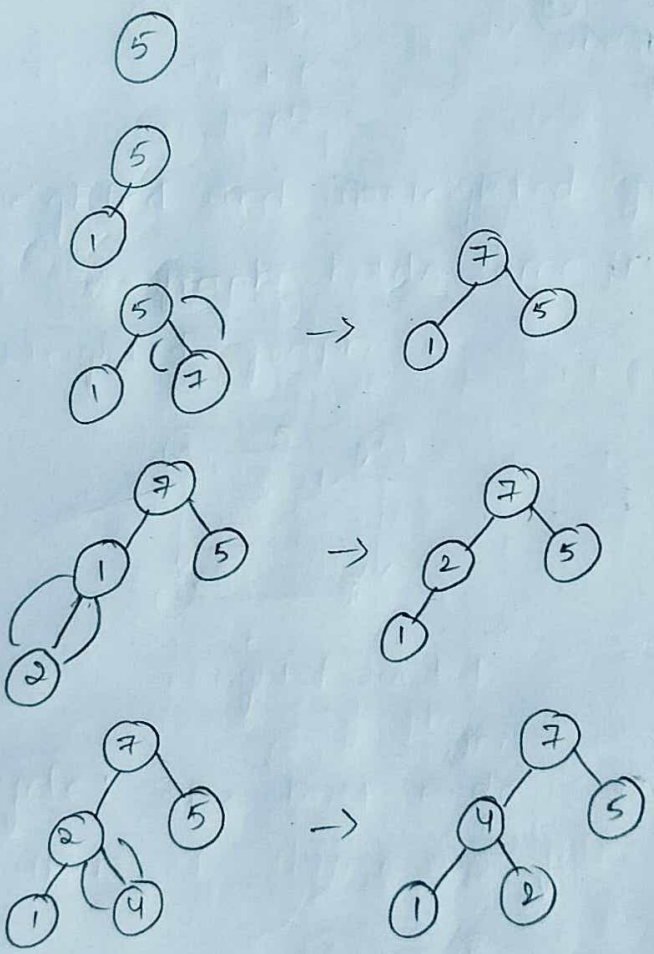
We can construct a Heap sort either by using max heap property or min Heap property.

Max Heap : In the max heap property the child nodes are always less than its parent nodes.

In Min Heap, the child node are always greater than its parent node.

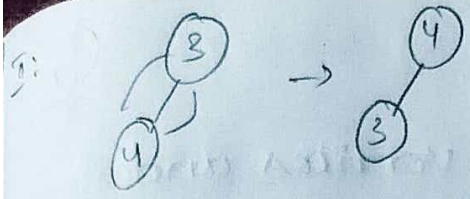
Construct the Max Heap for the following and sort the data.

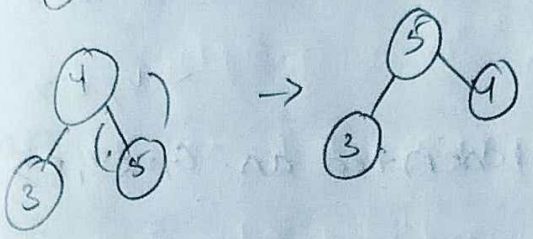
5, 1, 7, 2, 4, 3

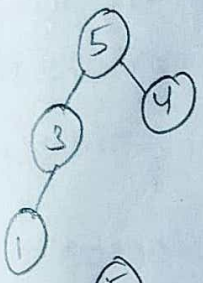


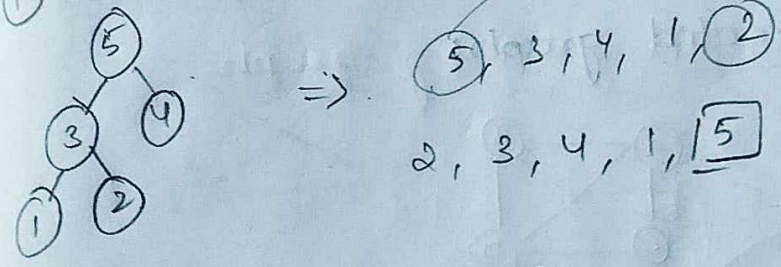
=> 7, 4, 5, 1, 2, 3

=> 3, 4, 5, 1, 2, 7

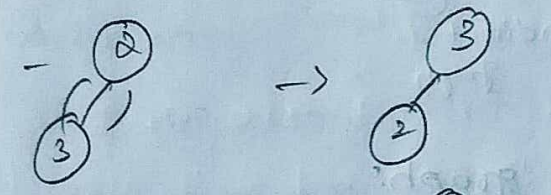


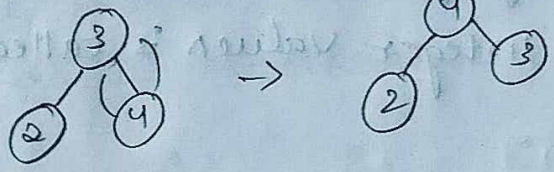


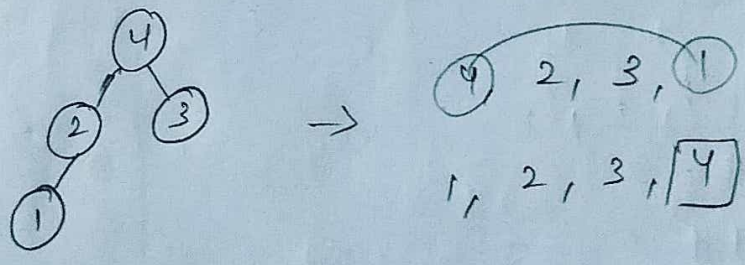




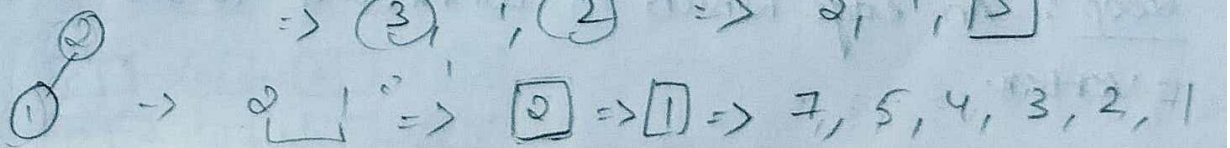
III







IV.

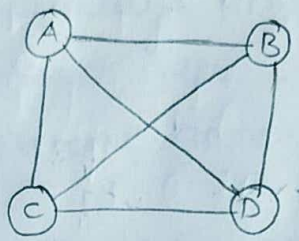


CO4 - Graphs

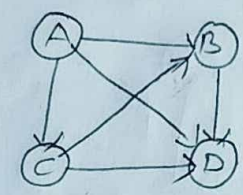
A graph is a collection of vertices and edges. $<V, E>$

Mathematically it is represented as $G=<V, E>$

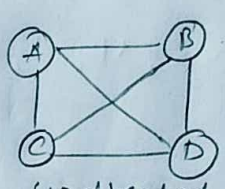
Eg:



Directed and undirected graph:



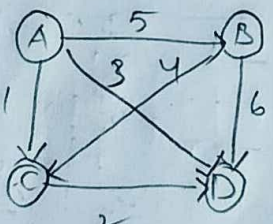
directed graph



undirected graph

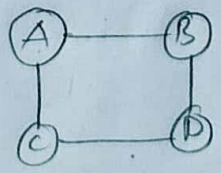
Weighted and unweighted graph:

A graph holds some integer values is called weighted graph.



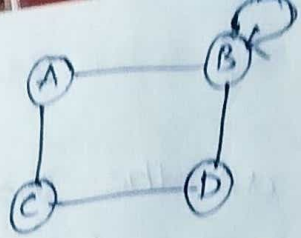
weighted directed graph.

Cycle: The last vertex node connects to the first node then it forms a cycle.



Loop: An edge which is connected to the same vertex.

Eg:



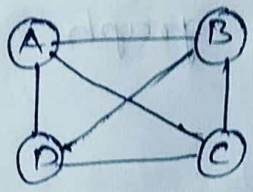
Graph Representation techniques:

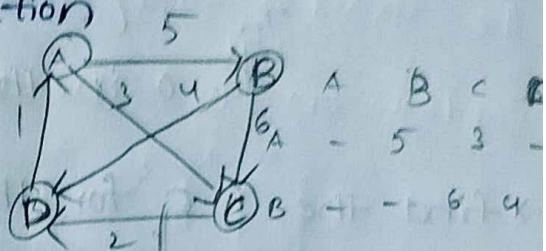
A Graph can be represented in two ways:

i) using Adjacency matrix

ii) using Linkedlist representation

consider



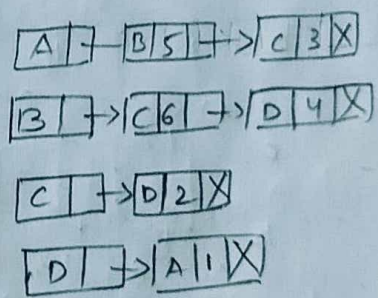


1. Adjacency matrix:

An Adjacency matrix is which if there is a edge between one vertex to another then represent it as 1 otherwise 0.

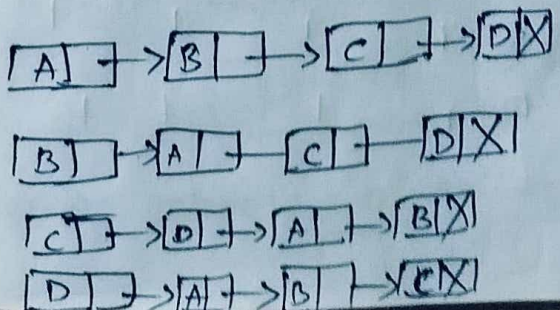
	A	B	C	D
A	0	1	1	1
B	1	0	1	1
C	1	1	0	1
D	1	1	1	0

Linked list:



2. using Linked List:

Ans



→ Transitive closure:

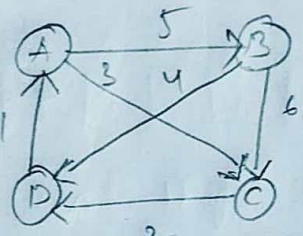
Transitive closure is defined as the shortest path between the nodes in a graph.

Mathematically it is represented as T(i, j) = 1 If there is a direct (or) and indirect edge from that vertex = 0

$T_{ij} = 1$ (...

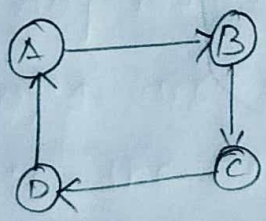
= 0 if there is no edge

→ find the T.C for the following graph:



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

Ex: 2:



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

→ Graph Traversals:

A Graph can be traversed in two ways.

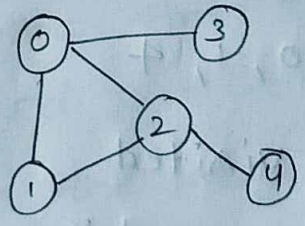
(i) DFS (Depth first search)

(ii) BFS (Breadth first search)

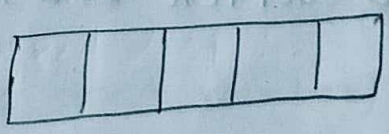
i) DFS:

1. Depth first search uses a strategy called stack.
2. choose any arbitrary vertex from the graph. (random).
and put that vertex of the top of stack.
3. Take the top element from the stack and put in the visited list.
4. ~~(clear)~~ the list of that vertex adjacent nodes. create
5. Add the ones(vertex) which are not visited on top of the stack.
6. Keep repeating the above two steps until the stack is empty.
7. And all the vertices are visited.
8. stop.

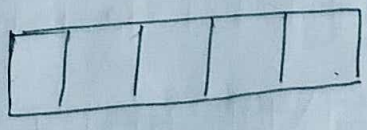
→ Traverse the following graph using DFS:



step-1:

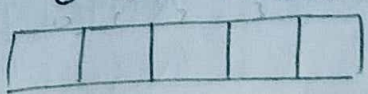


stack

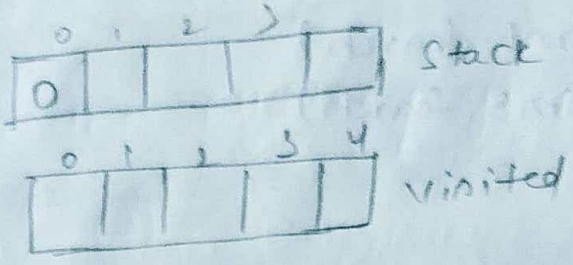


visited

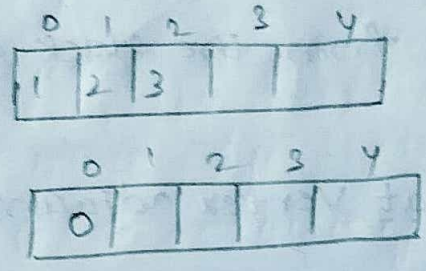
step-2: choose '0' an abbitrary vertex, the dfs



algorithm starts by putting the top element in the stack.



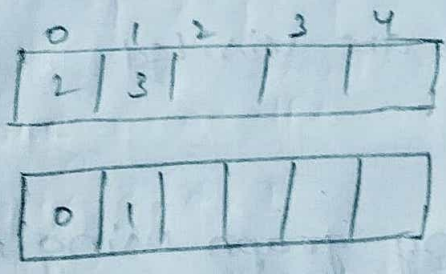
Pop the top element in stack and write in the visited, and also adjacent nodes of that vertex. on to the stack.



3. choose 1 as the vertex.

the adj of nodes of 1 are 0, 2

since, 0 is already visited and 2 is already in stack, then,

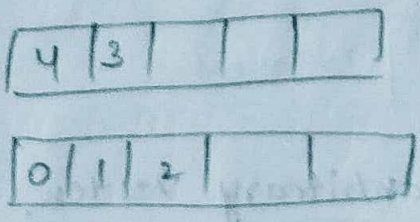


4. choose '2' as the vertex.

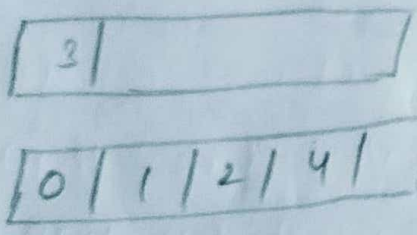
the adj nodes are 0, 1, 4.

since 0, 1 are already visited.

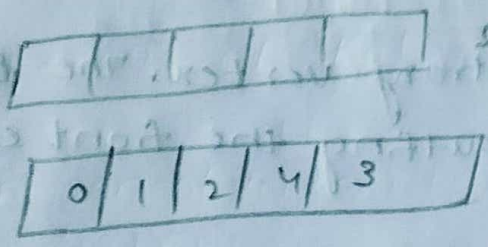
and 4 is the new vertex and it on the top of the stack



5. choose '4' as the verten
the adj vertican are 2
which in already visited then pop the '4' to the visited list.



6. choose '3' as the verten
the adj vertecen are 0
which in already visited then pop the 3 to the visited list



Then the stack is empty. then all elements are visited

0 → 1 → 2 → 4 → 3

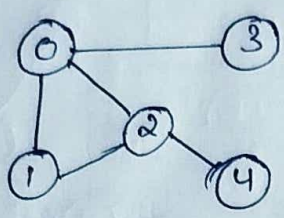
(ii) BfS :

1. BfS uses a strategy called queue.
2. start by choosing any Arbitrary vertex from the graph and put in the queue.
3. Take the first or front element from the queue and put it on the visited list.
3. create the list of that vertex adjacent nodes.
4. Add the ones (vertex) which are not visited on the queue.
5. Keep repeating the above two steps until the queue is empty.

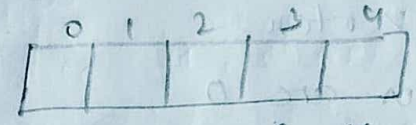
8. And all the vertices are visited

9. stop.

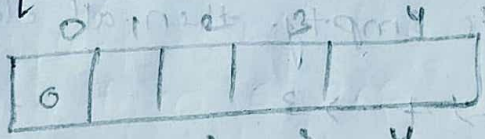
→ Traverse the following graph using Bfs.

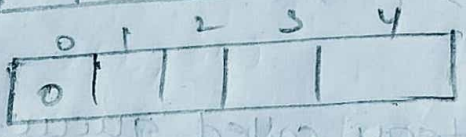


Step-1 :



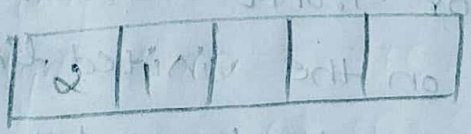
choose '0' as arbitrary vertex. The Bfs algorithm starts by putting the front element in the queue.

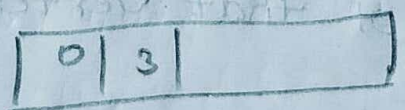




choose 0 as the verten then the adjacence nodes are 3, 2, 1

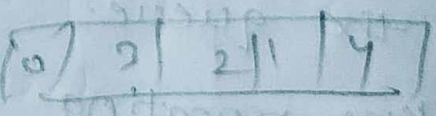
choose '3' as the arbitrary vertex





(2) → 1, 4

(1) → (4)

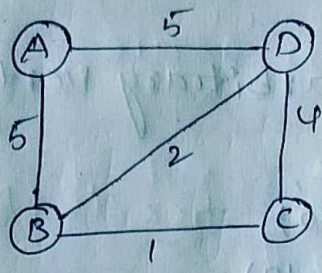


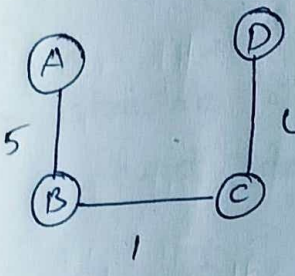
-> Minimal Spanning Tree (MST):

The minimum spanning Tree is a subset of the edges of a connected, weighted graphs, which connects all the vertices together without any cycles and with minimum total edge weights.

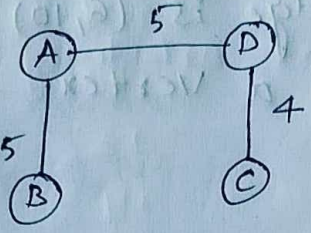
for a graph, we can have or construct more than one MST.

Eg:

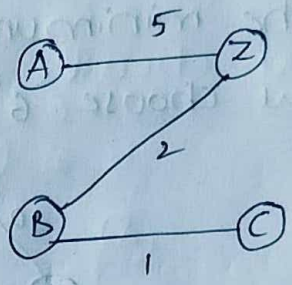




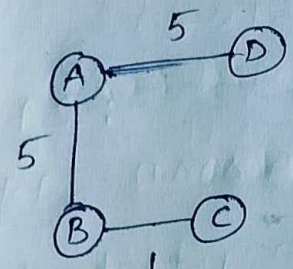
MST = 10



MST = 14



MST = 8



MST = 11

1. prims Algorithm:

prims algorithm comes under ~~(prims)~~ greedy technique.

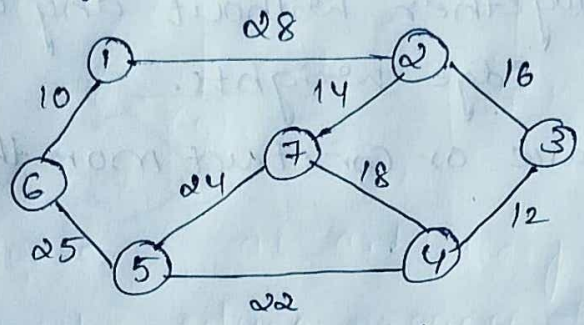
Algorithm:

choose any arbitrary vertex from the given graph.

Now we have to find all the edges that connect the tree to new vertices. from that find the minimum edge and add to the tree.

Repeat step 2 untill we get Minimum Spanning Tree.

→ find the MST for the following graph using prims Algorithm.



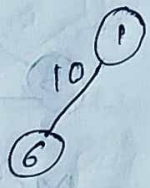
Step - 1 : choose '1' as an arbitrary vertex.

The edges from 1 are,

1 → (2, 28) (6, 10)

the minimum edge is (6, 10).

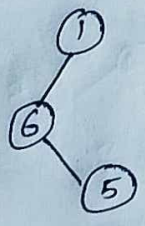
→ Now choose '6' as a vertex.



The edges from 6 are

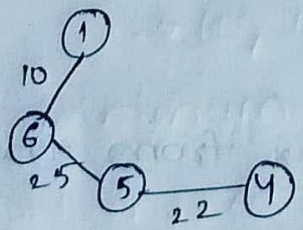
6 → (5, 25)

→ choose '5' as the vertex



The edges from 5 are,

5 → (4, 22) (7, 24)



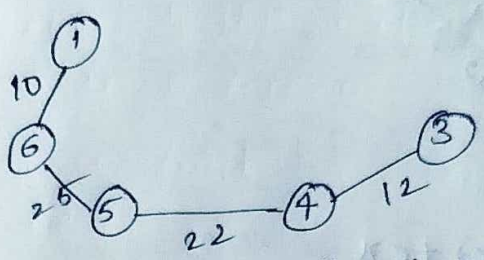
The minimum edge is (4, 22)

→ Now choose 4 as the vertex

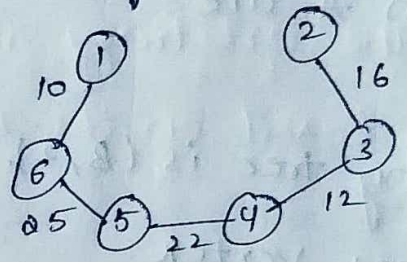
The edges passing from 4 are.

4 -> (7, 18) (3, 12)

-> choose (3, 12) as a vertex

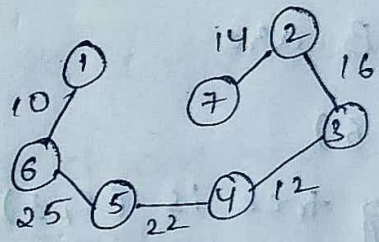


The edges passing from 3 are (2, 16)



-> choose 2 as a vertex

and edges are (7, 14)

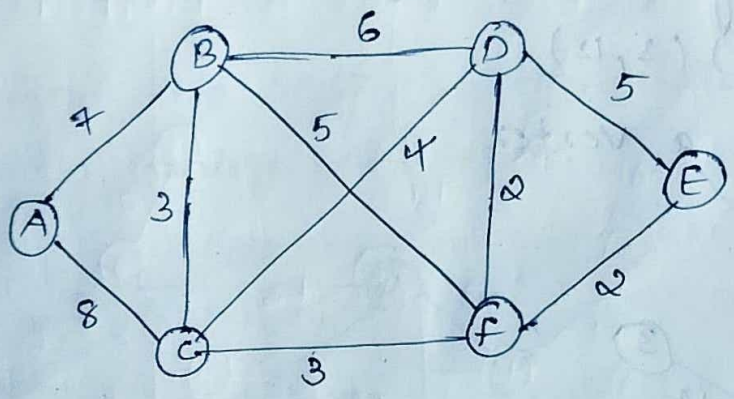


since all the vertices have been covered the final MST = 10 + 25 + 22 + 12 + 16 + 14

= 99

-> Construct a MST for the following graph using prims Algorithm.

(A)



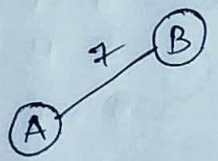
Choose 'A' as arbitrary Vertex

The edges passing from 'A' are

A → (B, 7) (C, 8)

The minimum spanning tree is (B, 7).

Now choose 'B' as arbitrary vertex

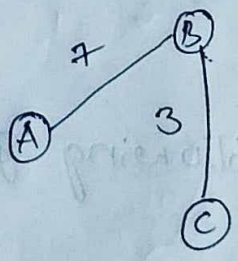


The Edges passing from B are

B → (C, 3) (f, 5) (D, 6)

The minimum spanning tree is (C, 3)

Now choose 'C' as arbitrary vertex

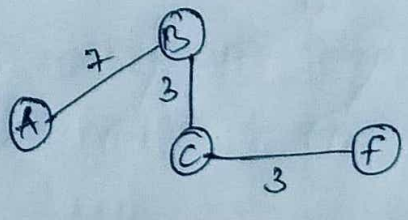


The Edges passing from C are

C → (D, 4), (f, 3)

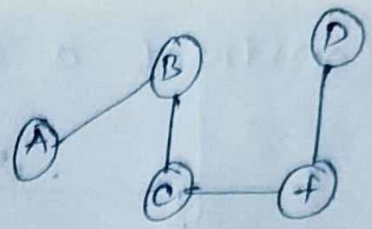
The minimum spanning is (f, 3)

Now choose f as arbitrary vertex



the edges from f are ~~f to D~~

$f \to (D, 2)\ (E, 2)$



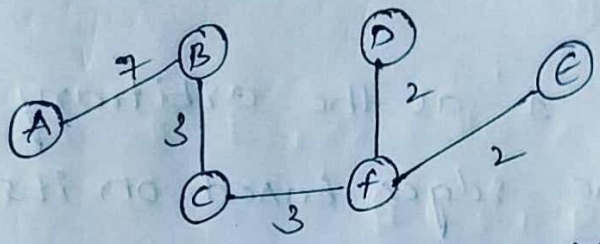
Now the M.S.T is $(D, 2)$

Now choose f an Arbitrary vertex

$f \to (D, 5),\ (E, 2)$

The minimum MST $(E, 2)$

Now choose E an vertex



since all the vertices are covered the final MST is $7 + 3 + 3 + 2 + 2$

$= 17$

→ Kruskals Algorithm.

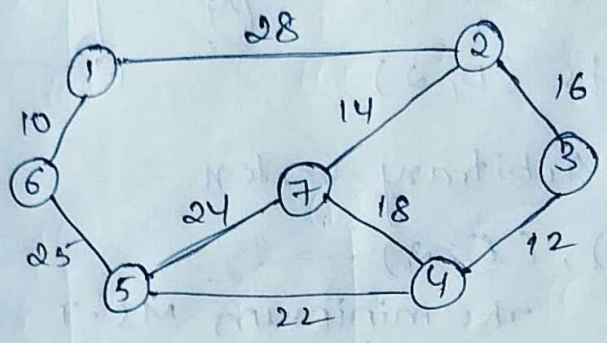
step-1: choose any arbitrary vertex from the given graph.

step-2: Sort all the edges based on its weight

step-3: choose the minimum weight edge and added to the tree untill all the edges are covered without cycle.

step-4: stop.

→ Construct a MST for the following graph.



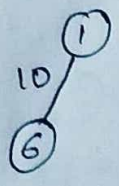
Sol:

Step-1: Choose '1' as the arbitrary vertex

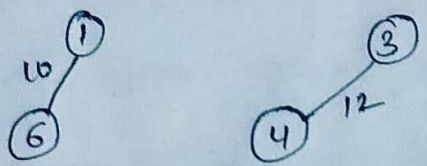
step-2: Sort the edges based on its weight.

Edge	wt
1-6	10
4-3	12
7-2	14
2-3 (Colsed	16
7-4	18
5-4	22
7-5	24
6-5	25
1-2	28

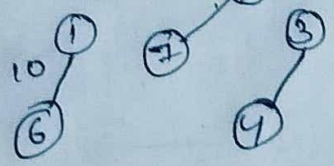
step-3:

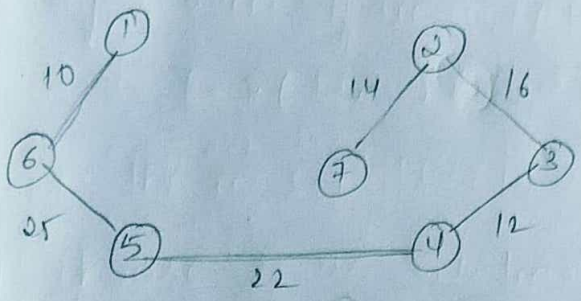


step-4:



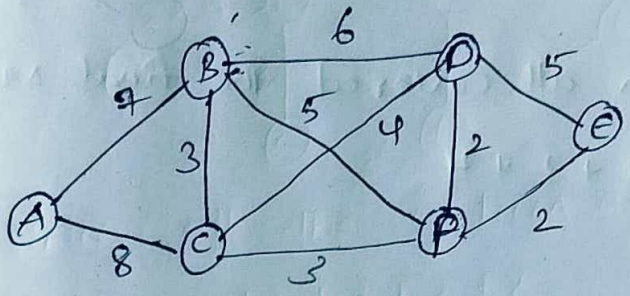
5:





$10 + 25 + 22 + 12 + 16 + 14 = 99$

→ Construct a MST for the following graph.

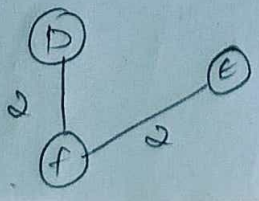


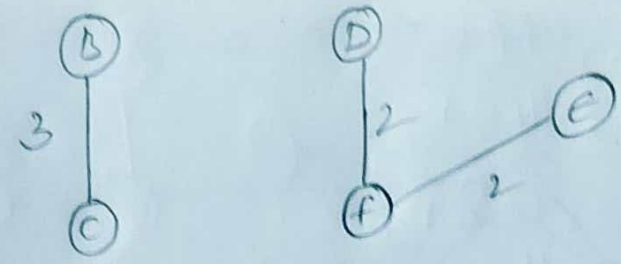
1. Choose A as arbitrary vertex
2. Sort the edges based on its weights.

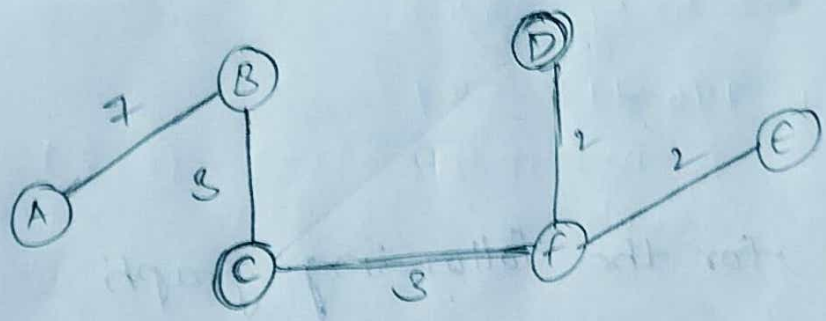
Edge	Wt
D - f	2
f - E	2
B - C	3
C - f	4
C - D	5
B - f	5
D - E	6
B - D	7
A - B	8
A - C	

1.

2.







Since all edges all covered the final MST is

$$7 + 3 + 3 + 2 + 2$$

$$= 17$$