Date 02/08/19

DATA STRUCTURE

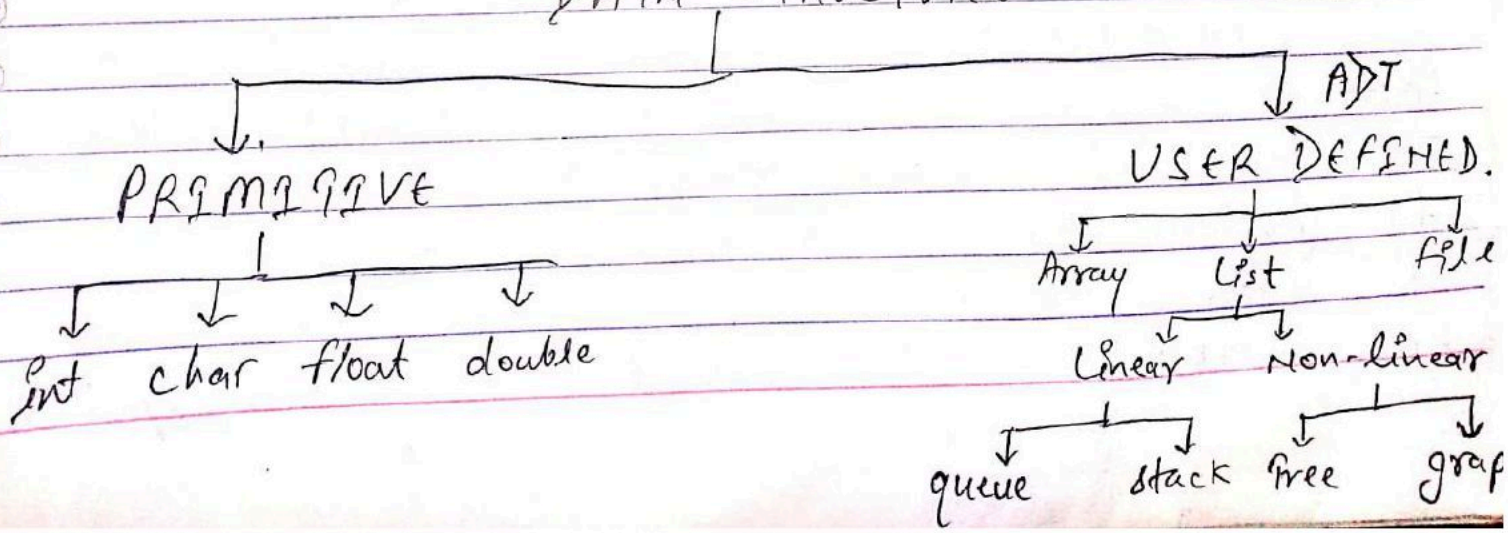
* Data Structure is a way of collecting & organizing data in such a way that we can perform operations on these data effectively.

* Anything that can store data can be called as datastructure, hence int, float, char etc. are the datastructure.

* They are Known as primitive (builtin) datastructure.

* We have some complex datastructure, which are used to store large & connected data. These are Known as abstract datastructure (user defined data structure). for ex- LinkList, tree, graph, stack, Queue.

* All these datastructure allows us to perform different operations on data. We select these datastructure based on which type of operation is required. for example- Queue.



NEED OF DATA STRUCTURE.

* It gives different level of organization data.
* It tells how data store & access in it's elementary level.
* Provide operation on group of data, such as adding an item, deleting an item, updating an item, searching an item etc.
* Provide a means to manage huge amount of data efficiently.
* Provide fast searching & sorting of data.

OPERATIONS ON DATA STRUCTURE.

* Traversing
* Searching
* Inserting
* Deleting
* Sorting
* Merging

ELEMENTARY DATA ORGANIZATION

* Data:- Data can be defined as Representation of facts, concepts or instructions in a formulazid manner suitable for communication, interpretation or processing by human or electric machine.
* Data is represented with the help of characters like alphabets, digit & speial

character.

* Data items - A set of character which are used together to represent specific data element is called data items or fields. for example - name of student in a class represented by dataitem, say Name

* Record - Record is a collection of related dataitem.

* File - file is a collection of logically related records. for example - payroll file might consist of payrecords for a company.

* Entity - An entity is a person, place or thing, event or concepts about which information recorded.

* Attributes - Attributes gives the characteristics or properties of entity.

* Data value - A data value is a actual data or information contained in each attribute

Primary & Secondary Key -

* Primary Key - A field or a collection of fields in a record which identifies a record uniquely is called primary Key. for example - ID NO.

* Secondary Key - A field in a record which identifies the record but not uniquely is called Secondary Key.

ALGORITHM -

* An algorithm is a finite set of instruction kept in a particular order to perform a particular task.

* Algorithm is not a complete code or program. It is just a core logic (solution), of a problem which can be expressed as an high level in formal description. It is also known as pseudo code.

* An algorithm can be expressed in three ways
  (i) In any natural language.
  (ii) In a programming language or in.
  (iii) Or in form of flowchart.

OR PERFORMANCE

* EFFICIENCY of ALGORITHM -

An algorithm is said to be efficient & fast if it takes less time to execute & consumed less memory space. The performance of an algorithm measured on the basis of two properties.

(i) Space Complexity
(ii) Time Complexity.

Programme under execution is known as process.

* SPACE COMPLEXITY -

It is amount of memory space required by an algorithm, during the execution of ~~any~~ algorithm.

Space complexity must be taken seriously for multiuser system & insituation where memory is limited.

An algorithm generally required space for following component.

(a) Instruction Space

It is a space required to store the executable version of programme. This space is fixed but varies depending upon no. of lines of code in a program.

(b) Data Space.

It is the space required to store all the constants & variables (including temperory variable) values.

(c) Environment Space

It is the space required to store the environment information needed to resume the suspended process.

* TIME COMPLEXITY

The time Complexity is a way to represent the amount of time required by the programe to run till its complition.

It is generally a good pratice to try to keep the time required minimum, so that our algorithm complete its algorithm in minimum time possible.

ASYMPTOTIC NOTATION -

*. Asymptotic Notation are languages that allow us to analyze an algorithm running time by identifying it's behaviour as the input size for an algorithm increases.

* This is also known as algorithms growth rate.

* Asymptotic Notation means a way to write down & talk about the fastest & the slowest possible running time of an algorithm.

* BIG O NOTATION

• Big O is an asymptotic Notation for worst case running time.

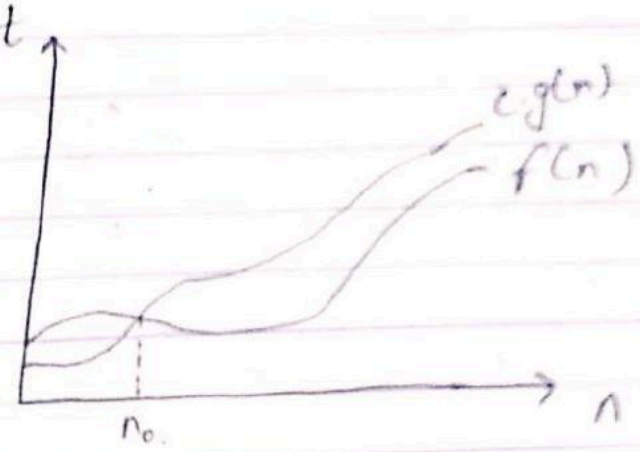
closet upper bound → tight upper bound

- This notation denotes the asymptotic tight upper bound of function $f(n)$.

Let f be a non negative function then we define the three most uncommon asymptotic bound as follow-

$f(n)$ is $O(g(n))$ if there exist a +ve constant c & n_0 such that $f(n) \leq c \cdot g(n)$ where $c > 0$ & $n_0 \leq n$

If $0 \leq f(n) \leq c \cdot g(n)$, we say that $g(n)$ is tight upper bound of $f(n)$.



for example- let $f(n) = 5n + 2$ & $g(n) = n$

for $c = 6$

$5n + 2 \leq 6 \cdot n$

$n = 2$

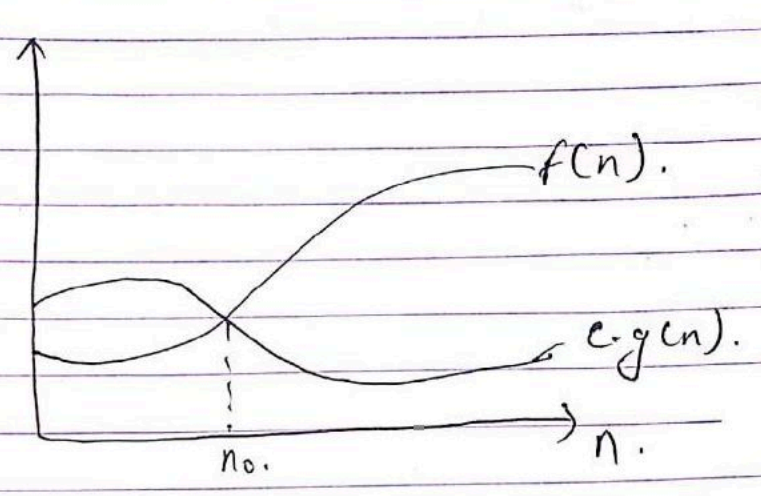
* BIG Ω NOTATION -

- It is an asymptotic notation for t

case running time,

• It provide asymptotically tight lower bound for given function $f(n)$.

• $f(n)$ is said to be $\Omega \cdot g(n)$ & written as $f(n) = \Omega(g(n))$. if & only if (iff) there are +ve constant c & n_0 such that $0 \leq c \cdot g(n) \leq f(n)$. for all $n_0 \leq n$.
if $f(n) = \Omega g(n)$ we say that $g(n)$ is lower bound of $f(n)$.

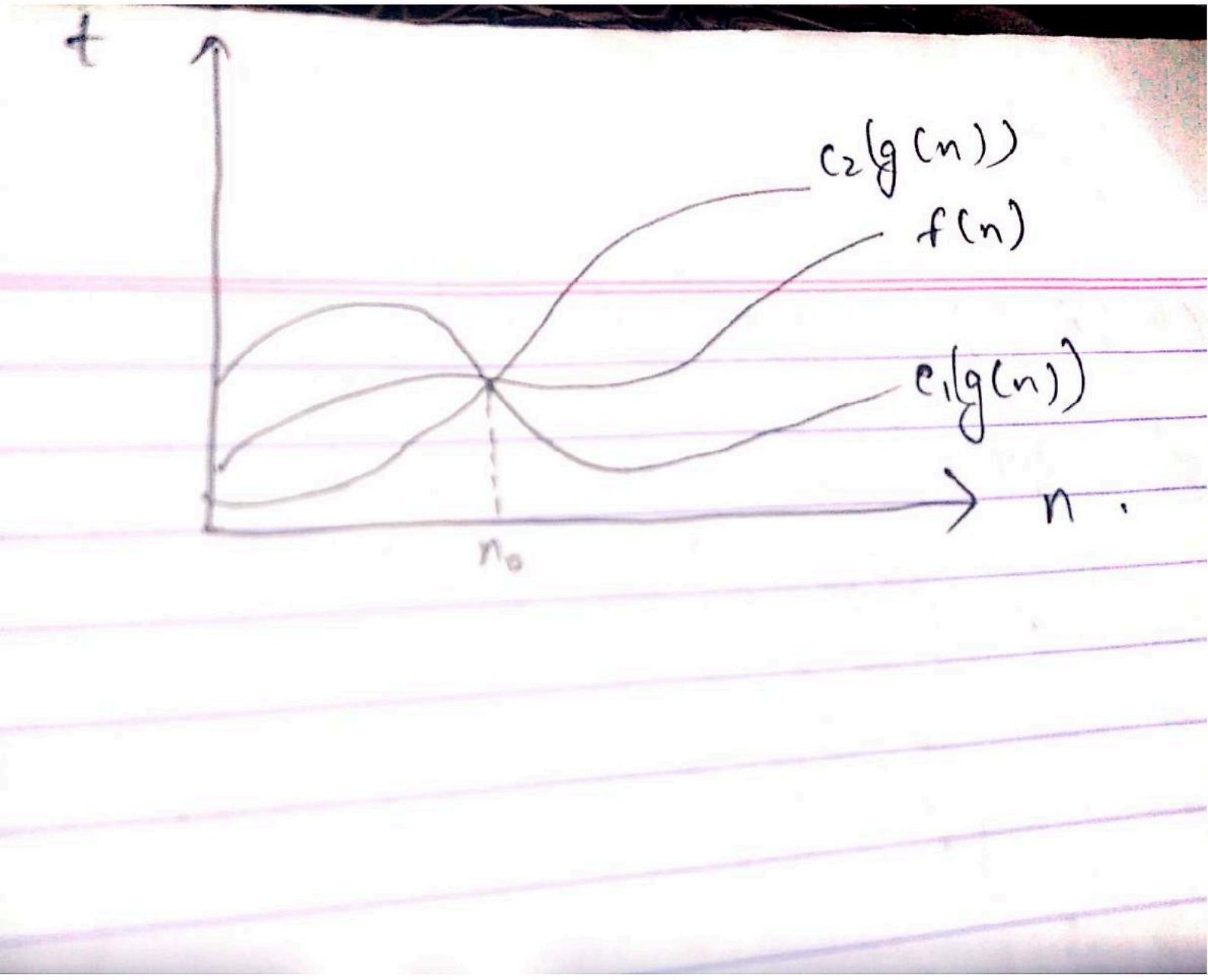


* BIG θ NOTATION -

• It is an asymptotic notation for average case running time.

• It is an asymptotic notation to denote asymptotically tight upper bound & tight lower bound.

• We say $f(n) = \theta(g(n))$ written as $f(n) = \theta(g(n))$ iff there are +ve constant c_1, c_2 & n_0 such that $0 \leq c_1(g(n)) \leq f(n) \leq c_2(g(n))$ $\forall c_1$ & $c_2 > 0$ & $n_0 \leq n$



Q

```
for(i=1; i<=n; i++).
{
    for(j=1; j<=i; j++).
    {
        for(K=1; K<=100; K++)
        {
            printf("ABC");
        }
    }
}.
```

$i = 1$	2	3	.....	K
$j = 1$	2	3		K
$K = 100$	100	100		100
$P = 1 \times 100$	2×100	3×100		$K \times 100$

$1 \times 100 + 2 \times 100 + 3 \times 100 + \ldots\ldots K \times 100$.

$100 (1 + 2 + 3 + 4 + 5. \ldots\ldots + n)$

$100 \left(\frac{n(n+1)}{2} \right) = O(n^2)$.

Q

```
for(i=1; i<=n; i++)
{
    for(j=1; j<=i^2; j++).
    {
        for(K=1; K<=n/2; K++)
        {
            printf("ABC");
        }
    }
}
```

$i = 1$	$j = 2$	$i = 3$	.....	n
$j = 1$	$j = 4$	$j = 9$		n^2
$K = n/2$	$K = n/2 \times 4$	$K = 9 \times n/2$		$n/2$.
$1 \times n/2$	$4 \times \frac{n}{2}$.	$9 \times 9 \times \frac{n}{2}$		

Pf $1 \times n/2 \quad 4 \times n/2 \quad 9 \times n/2 \quad \ldots \quad i^2 \times \frac{n}{2}$.

$\frac{n}{2} + 4 \times n/2 + 9 \times \frac{n}{2} + \ldots\ldots\ldots n^2 \times \frac{n}{2}$.

$\frac{n}{2} (1 + 2^2 + 3^2 + \ldots\ldots n^2)$.

$\frac{n}{2} \times \frac{n(n+1)(2n+1)}{6} = O(n^4)$.

Q
```
for (i=1; i<=n) i = i x 2).
    printf ("ABC");
```

$i = 1 \quad 1 <= n, \; 2 <= n, \; 4 <= n, \; 8 <= n, \; 16 <= n.$

$2^k = n \Rightarrow k = \log_2 n.$

Q1.
```
for ( i= n/2 ; i<=n ; i++)
    for ( j=1 ; j<=n/2; j++ ).
        for(K=1; K<=n ; K= K* 2).        O(n^4)
            printf ("ABC");
```

Q2.
```
for ( i=n/2 ; i<=n ; i++).
    for (j=1; j<=n; j= jx2).
        for ( K=1; K<=n; K= K* 2).       O(n^4)
            printf("Shiv");
```

Q3
```
A ()
{
    while (n >1)                  → n/2 times   O(n)
    {   n = n/2.
        printf("ABC");
    }
}.
```

ABSTRACT DATA TYPE - (ADT)

* Abstract Data Type is a process of providing only essential & hiding the details is known as abstraction.

* Abstract Data Type is a type whose behaviour is defined by set of values & set of operations.

* The definition of ADT only mentions what operation are to be performed but not how these operations will be implemented.

* It involves what can be done in the data, not how has to be done.

* The user of data type only with the knowledge ~~of~~ with values that can take & operation that can be performed on them. without any idea of how these types are implemented.

ARRAY

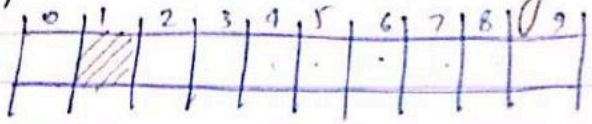
Array is a collection of homogenous data type.

Declaration -

data type a [size];

ex - int a [10];

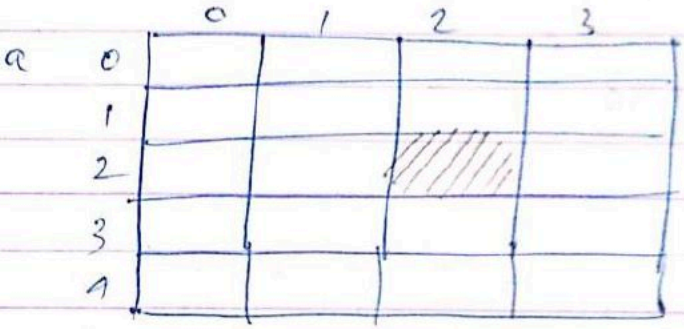
Actual (overview) Representation of memory -



2D - array declaration -

data type a [size1] [size2];

ex - int a [5] [4];



Overview Representation of Memory.

a[2][2].

* ROW MAJOR ORDER.

$$A[i,j] = B.S. + W[M[i] + j].$$

A[i,j] represents the address of ith row & jth column.

B.S. stand for Base Address

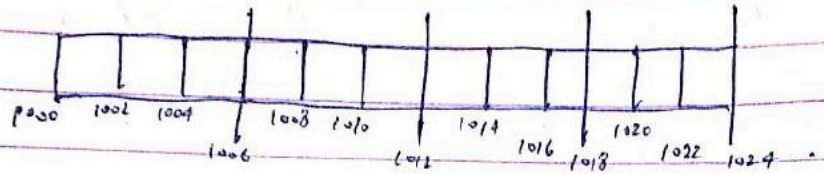
W stand for word size (size of data)

M is a no. of column in an array.

$2000 + 4(6 \times 4 + 3)$

Q a[4][3] whose B.A is 1000. Find B.A of a[3][2].

⇒



a[3][2] = 1000 + 2(3 × 3 + 2).

= 1022.

Q If an array of size 6×5 has an B.A. of 2000. What will be the BA of a[3][4] R.M.O.

a[3][4] = 2000 + 4(5 × 3 + 4).

a[3][4] ⇒ 2000 + 76

⇒ 2076

C.M.O.

a[3][4] = 2000 + 4(6 × 4 + 3)

= 2000 + 108.

= 2108.

* COLUMN MAJOR ORDER.

when the indexing starts from zero.

A[i, j] = B.A + W(R(j) + i).

R. $B.A + W(i-(R_i)C + j-(C_j))$

C $B.A + W(j-(C_j)R + i-(R_i))$

$C = 6(c_n - c_i + 1) \ldots (i+2)$

$R = C_{B.A} + W(i+2).C + j-2$

2 3 4

$\ldots \frac{4-2+1}{4-(-2)+1} \ldots (A[i]) - (C_j)$

SPARSE MATRIX -

* Matrix with maximum zero's element is called sparse matrix.
* A Matrix containing more no zero's value then non-zero's value is called sparse matrix.
* A matrix which is not sparse is called dense matrix.

* REPRESENTATION

A Sparse matrix can be represented by two method (i) Triplet method it is also known as array method - (ii) Link List method.

* Representation of Sparse matrix using array -

- In this representation we consider only non-zero's value with their row & column index values.
- In this representation zeroth row stores total no. of rows, total no. of columns & total no. of non-zero's value in the sparse matrix.

for example- consider a following sparse matrix

$$\begin{array}{c} \\ 0 \\ 1 \\ 2 \\ 3 \end{array} \begin{array}{c} \begin{array}{ccccc} 0 & 1 & 2 & 3 & 4 \end{array} \\ \begin{bmatrix} 0 & 0 & 0 & 10 & \\ 0 & 2 & 0 & 0 & 0 \\ 0 & 0 & 2 & 3 & 0 \\ 0 & 0 & 0 & 0 & 1 \end{bmatrix} \end{array} \quad 4 \times 5$$

row	0	1	2	2	3
col.	3	1	2	3	4
value	1	2	2	3	1

$3 \times n$

where n is non-ze values.

*/ VIRK ANSV -

* Syntax for accessing a field by a pointer variable of structure type.

struct student s1, s2, *p.
p = &s1; (storing address of structure variable to pointer var.)

p→name	s1.name	name
p→roll	s1.roll	roll
p→mark	s1.mark.	marks

both are same.

* DYNAMIC MEMORY ALLOCATION :

Allocation of memory during execution of program is known as dynamic memory allocation. This is done by three special function in C which are as follows.

(i) malloc ()
(ii) calloc ()
(iii) realloc ()
(iv) free ()

(i) MALLOC

Malloc is used to create single unit of memory of given size during run tim.

(12)

S = struct student * malloc (sizeof(struct student));

Syntax for malloc is -

pointer_variable = (cast type*) malloc (size in bytes);

ex- p = int * malloc (sizeof (int));

(ii) CALLOC -

By using calloc function we create n element of corresponding datatype in contiguous memory allocation dynamically.

Syntax for Calloc ( ) is →

pointer_variable = (cast type*) calloc (no of element, size of each element));

ex p = (int*) calloc (10, size of (int));

Q Create a student array using calloc fn →.

struct student *s;

s = (struct student *) calloc (10, size of (struct student));

s = (struct student *) calloc (n, size of (struct stude

s → name	s[0]. name.
(s+1) → name.	s[1]. name .

Q WAP to create n student record having field name, rollno & marks. Print the detail of those student who had scored above 70% marks.

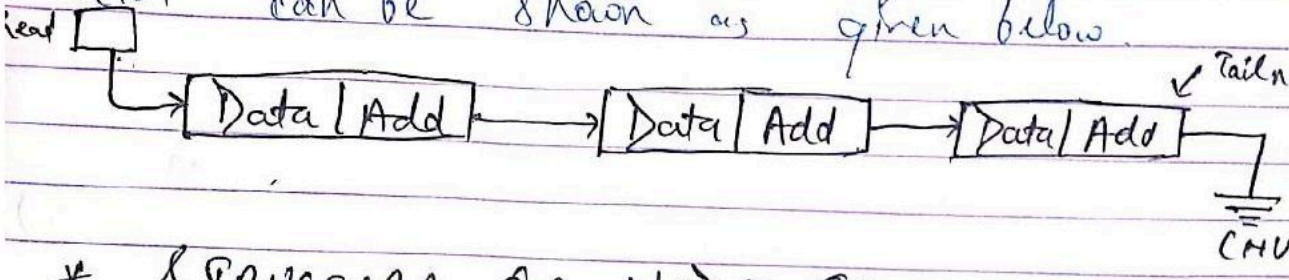
(3) pointer_variable = (cast type *) realloc (ptr, newsize in byte).
free (s)

```
#include <stdio.h>
struct marks
 {
  int m[5];
 };
struct student
 {
  char name[30];
  int rollno;
  int total;
  int per;
 struct marks sub;
 };
 void main()
 {
  int i, j, n;
  struct student s[n], *p;
  printf("Enter no. of Students");
  scanf("%d", &n);
  p = (struct student *) calloc (n, size of(struct student));
  printf("Enter Student Details \n");
  for(i=0; i<n; i++)
  {
   printf("Enter Student %d", i+1);
   printf("Enter student name");
   scanf("%s", (p+i)->name);
   printf("Enter student rollno");
   scanf("%d", (p+i)->rollno);
  for(j=0; j<5; j++)
   {
   printf("Enter Subjects %d", j+1);
```

```
        scanf("%d", (p+i)->sub.m[j]);
      }
    }
    for(i=0; i<n; i++)
    {
      (p+i)->total = 0;
      for(j=0; j<5; j++)
      {
      (p+i)->total = (p+i)->total + (p+i)->sub.m[j];
      }
      (p+i)->per = (float)(p+i)->total/5;
    }
    for(i=0; i<n; i++)
    {
      if((p+i)->per > 70)
      {
      printf("Student Name & Roll No.\n");
      printf("%s %d", (p+i)->name, (p+i)->rollno);
      for(j=0; j<5; j++)
      {
      printf("Subject mark of %d", j+1);
      printf("%d", (p+i)->sub.marks[i]);
      }
      printf("Total Percentage is %f", (p+i)->per);
      }
    }
```

LINK LIST -

Link list is a collection of **nodes** where each node contains two part a data part & a address part where data part contains information of different field of record whereas address part is a self referential field which contains the address of next node. Link list has one special pointer variable known as **head** which contains the address of first node. The last node of link list which is pointing to nothing (null) is known as **tail** of link list. Link list can be shaon as given below.



* STRUCTURE OF NODE IS

```
struct node
{
    datatype  var1 ;
    datatype  var2 ;
       .          .
       .          .
    datatype  varn ;
    struct node * link ;
};
```

Data

Address

* SYNTAX FOR CREATING A NODE -

```
struct node
{
    int
    ~~char~~ a;
    struct node * link;
};  void main()
struct node *p;
p = (struct node *) malloc (sizeof(struct node));
}.
```

for accessing a we use $p \rightarrow a$.
for accessing next node $p \rightarrow$ ~~next~~ link.

* VARIOUS OPERATION PERFORME ON LINKLIST

(i) Searching
(ii) Insert/
(iii) delete
(iv) Traverse

* TRAVERSE -

```
void Traverse (struct node *p)
{
    while( P != Null)
    {
        printf("\n value is %d", p->a);
        } p = p -> link;
}.
```

'null' ka matlab means segmentation fault.

* INSERTION

(i) While inserting a node we create a new node dynamically (ii) To fill the value of each field given by user in data part. (iii) To search the position of node in existing list. (iv) To link the node at desired position.

Insertion is done in three ways -

(i) Insertion at head. -

A new node is always inserted at head position.

(ii) Insertion at tail -

The new node is always inserted at after the last node of existing list.

(iii) Insertion in between (At desired location).

The new node is inserted between two existing node in existing list

(i) At head.

```
void Insert_at_beginning (struct node *p)
{
    struct node *t;
    t = (struct node *) malloc ( sizeof(struct node));
    printf ("Enter any character");
    scanf ("%d", t->a);
    t -> link = p;
    p = t;
}
```

Insertion at desired location.

* Insertion on Tail -

```
Void Insertionontail (struct node *head)
{
    struct node *n, *p;
    n = (struct node *) malloc (sizeof (struct node));
    printf (" Enter the character ");
    scanf (" %c", &n->a)
    n->link = NULL;
    if (head == NULL)
        head = n;
    else
    {
      p = head;
    while (p->link != NULL)
    {
    } p = p->link;

      p-> link = n
    }.
```

* SEARCHING -

```
Void search ( struct node * head, int n).
{
  int f = 0, c = 0 ;
 struct node * p ;
  p = head
  if (p == NULL)
  printf(" \n Element Not found List is Empty ");
   while (p
   else
    {
    while (p != NULL) {
     if (p → a == n)
      {
        printf(" \n Element found in position
 %d ", f+1);
        c++;
      }
     f++;
     p = p → link;
    }
    if (c == 0)
    printf(" Element Not found ");
    }.
}.
```

DELETION

* Deletion of Head Node -

```
void Delete (struct node *head)
{
    struct node *p;
    p = head;
    if (p == NULL)
        printf("\n Link list is empty");
    else
        head = head -> link;
        free(p);
}
```

* Deletion of Tail Node -

```
void Delete (Struct node *head).
{
    struct node *p, *q;
    p = head;
    if (p == NULL)
        printf("\n Link list is empty");
    else
        q = p.
        while (p -> next != NULL)
            q = p;
            p = p -> next
        if (p == q)
        {
            head = NULL
```

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

* Deletion in between.

```
void Delete (struct node * head).
{
  struct node *p,*q;
  int t;
  p = head
  if (p == NULL)
  {
  } printf("\n Link list is empty");
  else.
  {
    printf("Enter The value");
    scanf("%d", &t);
    q = p;
    while(p != NULL)
    {
      if (t == p->a)
        break;
      q = p
```

```
    p = p -> link;
}
if (p == NULL)
    printf("\n Value not found in list No
            Node deleted ");
else
{
    if (p == q)
    {
    head = NULL
    free (p);
    }
    else
    {
      q -> next = p -> next
       free(p);
    }
  }
}
}
```

CIRCULAR LINK LIST

DOUBLY LINK LIST -

(i) Insertion On Head -

```
Void Insert_head ( )
{
  struct node *p, *pre, * next ;
  p = (struct node*) malloc( sizeof(struct node));
  printf("Enter the value \n");
  scanf (" %d ", & p → a) ;
  if (head == NULL )
    {
      head = p ;
      p → pre = NULL ;
      p → next = NULL ;
    }
  else.
    {
      p → pre = NULL;
      p → next = head;
      head → pre = p
      head = p ;
    }
}
```

(ii) Insertion On Tail

```
void Insert_tail ( )
{
  struct node *p, *q, * pre, * next ;
  p = ( struct node * ) malloc (size of (struct node))
  printf (" Enter the value \n");
  scanf (" %d ", & p → a);
```

```
if (head == NULL)
 {
    head = p;
    p → prev = NULL;
    p → NEXT = NULL;
 }
else
 {
    q = head;
    while ( q → next != NULL)
     {
       q = q → next;
     }
    q → next = p;
    p → next = NULL;
    p → pre = q;
 }
}
```

DOUBLY

(iii) Insertion in between on the basis of data.

```
void Insert_in-between ( )
{
   struct node *p, *q;
   p = (struct node *) malloc (s.tue of (
   printf ("Enter the value \n");
   scanf ("%d", &p → a);
   if (head == NULL)
    {
      head = p;
      p->link = NULL;
```

```
} p->pre = NULL;
else {
    q = head;
    if(q->link == NULL)
    {
        if(q->a < p->a)
        {
            q->link = p;
            p->pre = q;
            p->link = NULL;
        }
        else
        {
            p->pre = NULL;
            p->link = q;
            q->pre = p;
            head = p;
        }
    } // end of if.
else {
    while(q->next != NULL && q->a < new->a)
        q = q->next;
    if(q != head)
    {
        new->pre = NULL;
        new->next = q;
        q->pre = new;
        head = new;
    }
    else if(q->next == NULL && q->a < new->a)
```

P.T.O.

Insertion in b/w in decen

```
void insert (void)
{
    p = head
    node *pre, *ne, *new
    new = (node *) malloc (sizeof(node))
    printf("Enter the value");
    scanf("%d", &new->a);
    if (p == NULL)
    {
        head = new
        new->next = NULL;
    }
    else.
    {
        pre = head;
        ne = pre->next;
        if (ne == NULL)
        {
            if (pre->a > new->a)
            {
                new->next = head;
                head = new;
            }
        }
    else
    {
while (ne != NULL && pre->a < new->a &&
        ne->a < new->a).
    {
        pre = ne;
        ne = ne->next;
    }
```

```
new -> next = pre -> next
pre -> next = new;
}
```

In Doubley link list continove.

```
{
    new -> next = NULL;
    new -> pre = q;
    q -> next = new;
}
else
{
    new -> pre = q -> pre
    new -> next = q;
    q -> pre -> next = new;
    q -> pre = new;
}
}
}
```

Date- 4/09/19

UNIT-2

STACK

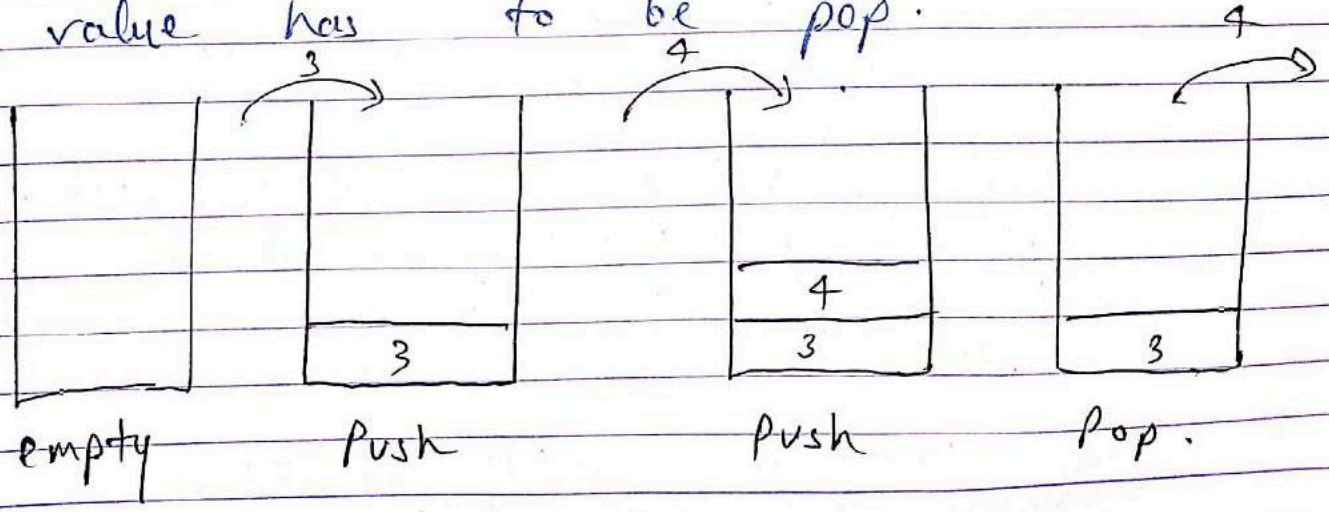
* Stack is the data structure which is used to restore data in LIFO order LIFO stand for Last In First out order.

* The element which is placed last is accessed first

* In stack terminology, insertion operation is called Push operation & removeral operation is called Pop operation.

* Both insertion & removal are allowed at only one end of the stack called TOP of the stack.

* A stack data structure has a special variable called StackTop whose value denotes the position of stack where the value has to be inserted or from the value has to be pop.



* Implementation of stack is done by using Array & LinkList.

* BASIC OPERATIONS OF STACK

(i) Push ()
(ii) Pop ()
(iii) Peep ()
(iv) Isempty ()
(v) Isfull ()

```
#include<stdio.h>

int isfull(void);
int isempty(void);
void push(void);
int peep(void);
int pop(void);
int a[50], top = 0;
void main(void)
{
    int i,n;
    char ch='y';
    while(toupper(ch)=='Y')
    {
    printf("\n 1: PUSH \n 2:POP \n 3: PEEP\n");
    printf("\n Enter your choice ");
    scanf("%d", &i);
    switch(i)
    {
    Case 1:  push();  break;
    Case 2:  n=pop();
```

```
                                    deleted
        printf("\n Value of ~~top~~ is %d", n);
            break;
Case 3 :    n = peep();
        printf("\n Value at top is %d", n);
            break;
default :
        printf("\n Wrong Choice");
}
        printf("\n Do you want to continue
                Y/N");
        scanf("%c", &ch);
    }
}
    int isempty()
    {
        if ( top == 0)
            return 1;
        else
            return 0;
    }.
    int is full ()
    {
        if ( top == 50) // Size of stack array
            return 1;
        else
            return 0;
    }
    void push (void)
    {
        int n;
        if ( isfull() )
```

```
printf("\n stack is full");
else
{
 printf("\n Enter the value ");
 scanf("%d", &n);
  a[top] = n;
   top++;
  }
printf("\n Value inserted successfully");
}
int pop (void)
{
  int n;
  if (isempty())
   printf("\n Stack is empty");
  else
   top--;
   n = a[top];
    return n;
  }
 printf("Value deleted successfully");
}.
int peep (void)
{
  int n;
  if (isempty())
   printf("\n Stack is empty");
  else
   n = a[top-1];
    return n;
}.
```

REVERSE POLISH

+ -

Imp.

INFIX TO POSTFIX CONVERSION

Priority
( )
↑ or ^
* or /
+ or -

Q (A+B/C * (D+E) - F)

Scan Symbol	Stack	Postfix expression
(	(	
A	(	A
+	(+	A
B	(+	A B
/	(+/	A B
C	(+/	A B C
*	(+*(	A B C /
(	(+*(	A B C /
D	(+*(	A B C / D
+	(+*(+	A B C / D
E	(+*(+	A B C / D E
)	(+*(+)	A B C / D E +
-	(+*	A B C / D E + *
F	(+-	A B C / D E + * F
)		A B C / D E + * F - +

ABC/DE+*+F-+ is the postfix expression.

Q ((G * (F ↑ E / D) - C * B) + A)

Scan	Stack	Postfix Expression.
(	(	
(	((	
G	((	G

((G * (F ↑ E / D) - (* B) + A)

*	((*	G
(	((* (	G
F	((* (	GF
↑	((* (↑	GF
E	((* (↑	GFE
/	((* (/	GFE↑
D	((* (/	GFE↑D
)	((* (/)	GFE↑D/
-	((* -	GFE↑D/*
C	((-	GFE↑D/*C
*	((- *	GFE↑D/*C
B	((- *	GFE↑D/*CB
)	((- *)	GFE↑D/*CB*-
+	(+	GFE↑D/*CB*-
A	(+	GFE↑D/*CB*-A
)	(+) → empty	GFE↑D/*CB*-A+

GFE↑D/*CB*-A+ is the postfix expression

INFIX TO PREFIX CONVERSION -

Q ((G * (F ↑ E / D) - C * B) + A)

(A + (B * C - (D / E ↑ F) * G))

Reverse Infix → Postfix.

Postfix ⟷ Reverse.

(A + (B * C - (D / E ↑ F) * G)) . 20

Scan Symbol	Stack	Postfix expression.
(	(	
A	(	A
+	(+	A
(	(+(	A
B	(+(	AB
*	(+(*	AB
C	(+(*	ABC
-	(+(*-	ABC*
(	(+(-(	AB C*
D	(+(-(	AB(*D
/	(+(-(/	AB(*D
E	(+(-(/	AB(*DE
↑	(+(-(/↑	AB(*DE
F	(+(-(/↑	AB(*DEF
)	(+(-(/↑)	AB(*DEF↑/
*	(+(-*	AB(*DEF↑/
G	(+(-*	AB(*DEF↑/G
)	(+(-*)	AB(*DEF↑/G*-
)	(+)	AB(*DEF↑/G*-+

Now Prefix is + - * G / ↑ F E D * C B A

POSTFIX TO INFIX.

Q GFE ↑ D / * C B * - A +

=

Scan	Top of Stack	Expression Stack.
G	G	G
F	F	F, G
E	E	E, F, G
↑	E ↑ F	E↑F, G
D	D	D, E↑F, G
/	D / (E↑F)	D/(E↑F), G
*	(D/(E↑F)) * G	(D/(E↑F))*G
C	C	C, (D/(E↑F))*G
B	B	B, C, (D/(E↑F))*G
*	B * C	B*C, (D/(E↑F))*G
-	(B*C) - ((D/(E↑F))*G)	(B*C)-((D/(E↑F))*G)

A	A	A (B*C)-((D/(E↑F))*G)
+	A+(B*C)-((D/(E↑F))*G)	A+(B*C)-((D/(E↑F))×G)

PREFIX TO INFIX.

Q = + - * A B / C D E

Reverse → E D C / B A * - +

E	E	E
D	D	D E
C	C	C D E
/	C/D	C/D E
B	B	B C/D E
A	A	A B C/D E
*	A*B	A*B C/D E

- $(A*B)-(C/D)$ $\frac{(A*B)-(C/D)}{E}$

+ $((A*B)-(C/D))+E$ $(A*B)-(C/D))+E$

Q) Convert ^Postfix $AB-DE+F*/$ into infix.

A A A

B B $\frac{B}{A}$

$-$ $(B-A)$ $(B-A)$

D D $\frac{D}{(B-A)}$

E E $\frac{E}{\frac{D}{(B-A)}}$

$+$ $(E+D)$ $\frac{(E+D)}{(B-A)}$

F F $\frac{F}{\frac{(E+D)}{(B-A)}}$

$*$ $F*(E+D)$ $\frac{F*(E+D)}{(B-A)}$

F * (E+D) / (B - A) (F * (E+D) / (B-A))

II Sessional Start :-

RECURSION -

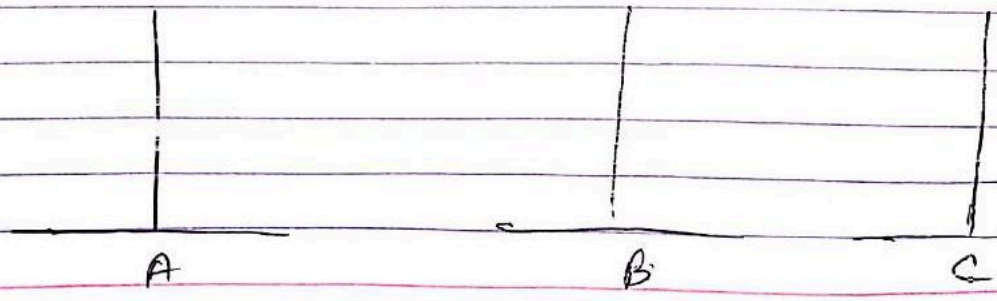
* TOWER OF HANOI

Tower of Hanoi is a mathematical puzzle where we have three poles or rod & n discs. The n disc is placed on one pole in decreasing order from bottom to top. The objective of puzzle is to move entire disc stack to another pole obeying the following rules-

Rule 1- Only one disc can be move from one pole to another pole at a time.

Rule 2- Each move consist of taking the upper disc from one of the stack & placing it on top of another stack it implies the disc can only be moved if it is upper most disc on the stack.

Rule 3- No disc may be placed on the top of smaller disc



In General the pattern of movement of n disc will be -

Step 1 - Move n-1 disc from A → B using C

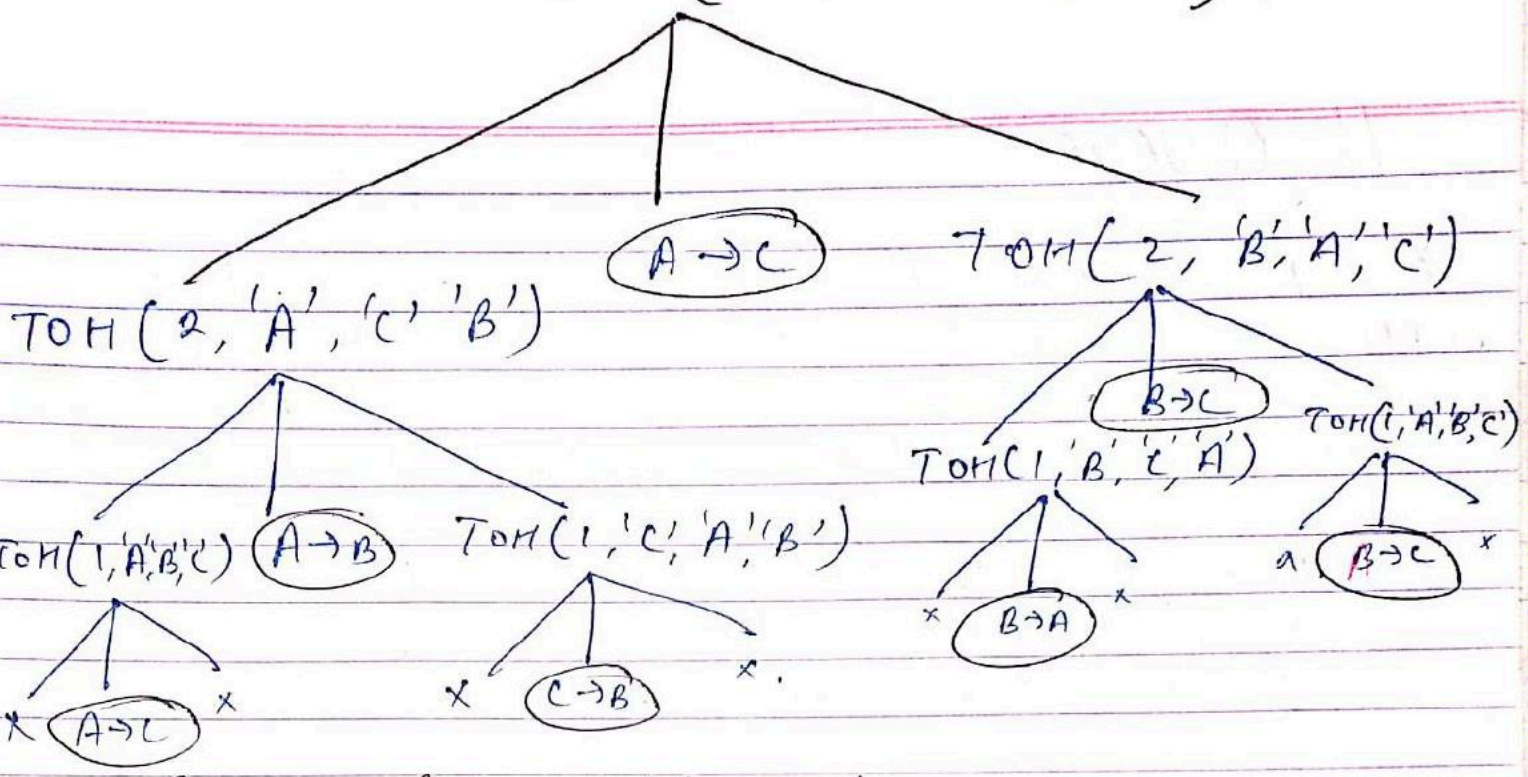
Step 2 - Move a disc from A → C

Step 3 - Move n-1 disc from B → C using A.

By above movement we can see there is a recursive call in Step 1 & step 3. The algorithm / function of Tower of Hanoi will be -

```
void TOH(int n, char A, char B, char C).
{
    if (n > 0)
    {
        TOH(n-1, 'A', 'C', 'B')
        printf("\n Move Disc %c to %c ", A, C);
        TOH(n-1, 'B', 'A', 'C');
    }
}
```

TOH(3, 'A', 'B', 'C').



Order of execution step. —

(A→C), (A→B), (C→B), (A→C), (B→A), (B→C), (B→C)

Note - Total no. of movement of n disc will be $2^n - 1$.

In every step we have to show poles.

QUEUE -

* A queue is a linear data structure where the ~~first~~ element is inserted from one end called rear & deleted from other end called front.

* front points to the beginning of the queue whereas rear points to the end of the queue.

* Queue follows the FIFO (First in First out)

* According to its FIFO structure element inserted first will also be removed first.

* In Queue, one end is always used to insert data known as enqueue & the other end is used to delete data known as dequeue. The enqueue & dequeue are the two important function for any insertion & deletion operation performed on queue.

FUNCTION ( ).

⇒
```
int a[5], front = -1, rear = -1;
int isempty (void)
{
  if( front == -1 || fraant > Rear).
```

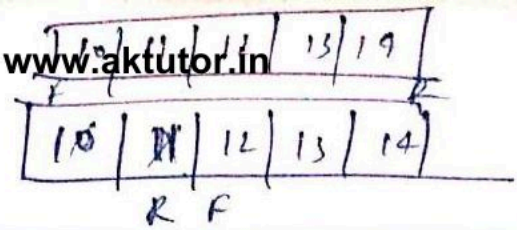
```
    return 1;
else
    return 0;
}

int isfull(void)
{
    if ( Rear >= (Size -1)   (Size → array.)
        return 1;
    else
        return 0;
}

void enqueue(int x)
{
    if (isfull())
        printf("\n Queue is full");
    else if ( Rear == -1 || front == -1)
        Rear = front = 0;
    else
        Rear++;
        a[Rear] = x;
    }.a[Rear] = x;
}

int
void dequeue(void x)
{   int t;
    if (isempty())
        printf("\n Queue is Empty");
    else
    {
```

```
    t = a[front]
    f++;
    return t;
}

void display (void)
{
    int i;
    if(isempty(1)
        printf("Queue is empty");
    else
    {
        for(i=front; i<=Rear; i++)
        {
            }.printf("%d", a[i]));
        }
    }
```

CIRCULAR QUEUE.

⇒ FUNCTION ()

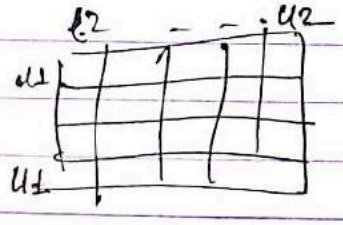
```
int isempty (void.)
{
    if(front == -1)
        return 1;
    else
        return 0;
}.
```




```
int isfull(void)
{
  if ( (front == 0 && R == size-1) || (front == Rear+1))
      return 1;
  else
      return 0;
}

void enqueue (int x)
{
  if(isfull())
  {
    printf("\n Queue is full");   // Overflow
  }
  else
  {
    if( Rear == -1)
    {
      front = 0;
      Rear = 0;
    }
    else if( Rear == size-1)
    {
      Rear = 0;
    }
    else
    {
      R = R+1;
    }
    a[R] = x;
  }
}
```

int a[3] = {21, }

a[ ] = {21 }



a[l1 : u1]

(u1 - l1 + 1) (u2 - l2 + 1)

a[l1 : u1]

[u1 - l1 + 1]

5 - 1 + 1 = 5

5 - 0 + 1 = 6

```
int cdequeue (void)
{
    int x;
    if( isempty ())
    {
        printf ("\n Underflow ");
        return;
    }
    else
    {
        x = a[front];
        if ( front == Rear) // Single element.
        {
            front = -1;
            Rear = -1;
        }
        else if ( front == size -1)
            {
                front = 0;
            }
        else
            {
                front = front + 1;
            }
        return x;
    }
}

void cdisplay (void)
{
    int i;
    if( isempty ())
```

Date - 21/09/19

UNIT - 3

QUICK SORTING -

```
int partition(char *A, int p, int r)
{
    int x, i, j, t;
    x = a[r];
    i = p-1;
    for(j=p; j<=r-1; j++)
    {
        if(a[j] <= x)
        {
            i = i+1;
            t = a[i];
            a[i] = a[j];
            a[j] = t;
        }
    }
    t = a[i+1];
    a[i+1] = a[r];
    a[r] = t;
    return i+1;
}
```

```
Quicksort(char *A, int p, int r)
{
    int q;
    if(p < r)
    {
        q = partion(char *A, p, r)
        Quicksort(A, p, q-1);
        Quicksort(A, q+1, r);
    }
}
```

23/09/19

MERGE SORT - ($n \log_2 n$)

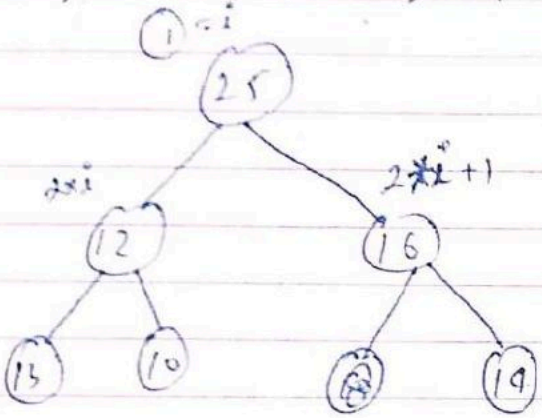
```
Merge (A, p, q, r)
{
    n1 = q - p + 1;
    n2 = r - q;
int L1[n1+1], L2[n2+1], L1[n1+1] = ∞, L2[n2+1] = ∞;
for (i = p; i <= n1; i++)
    L[i] = A[i];
for (j = 0; j <= n2; j++)
    R[j] = A[j].
i = 0;
j = 0;
for (K = p; K <= r; K++)
{
    if (L[i] <= R[j])
    {
        A[K] = L[i];
        i++;
    }
    else
        A[K] = R[j];
        j++;
} }.

Merge-sort (A, P, r)
{
    if (P < r)
    {
        q = (p + r)/2;
```

$h = \log_2 n$ $\quad (2^{h+1} - 1) = n$ (nodes)

```
Merge_sort ( A, p, q );
Merge_sort ( A, q+1, r );
Merge ( A, p, q, r );
}
}
```

HEAP SORT.

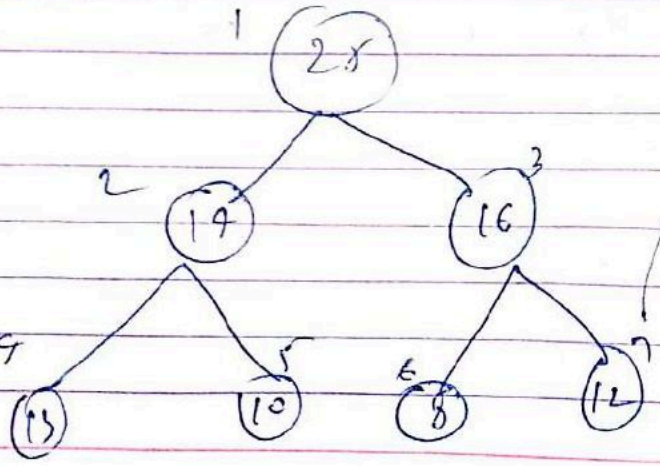
Construct almost binary tree of array
25, 12, 16, 13, 10, 8, 14



$P[i] = \left[\frac{i}{2}\right]$.

→ Heapify ← Max Heapify (for increasing order)
Min Heapify (for decreasing order)

Q = 25, 14, 16, 13, 10, 8, 12. find Heap size



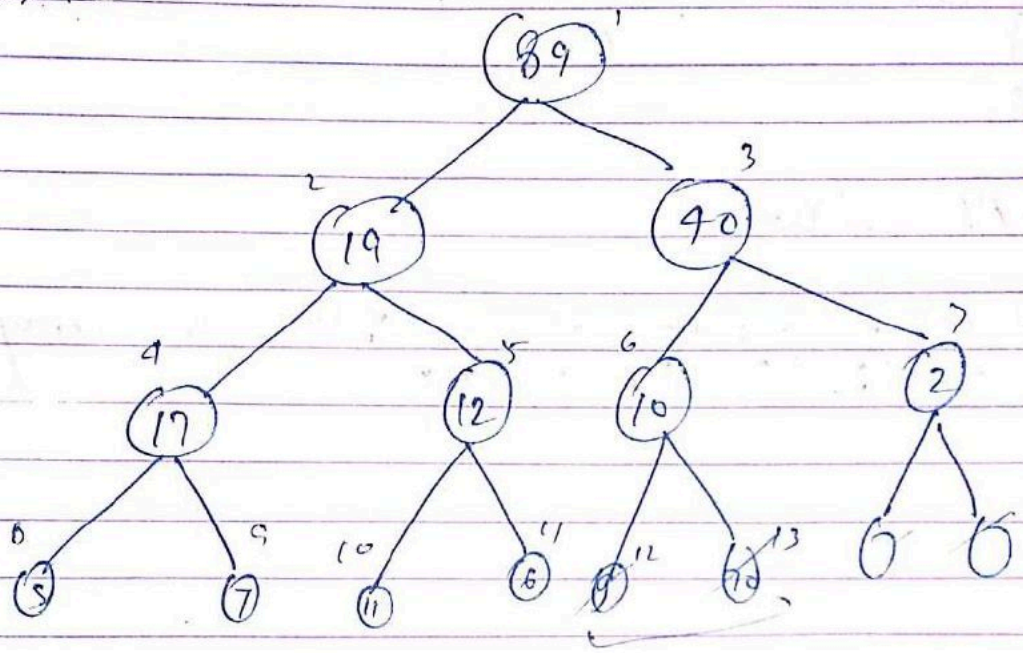
Heap size = 7.

Heap length = 7.
array

If heap size = heap length then it is either max or min heapify.

Q What is the heap size & heap length of given array - 89, 19, 40, 17, 12, 10, 2, 5, 7, 11, 6, 9, 70

⇒



Heap Size = 2. & Heap length = 13.
(Condition validates - 1)

Parent node from 1 to $\left\lceil \frac{\text{Size of array}}{2} \right\rceil$

Leaf node from $\left\lceil \frac{\text{Size of array}}{2} \right\rceil + 1$ to n.

ALGORITHM OF HEAP SORT -

```
Heap-sort (A)
{
(1) Build_max heap (A)
    for (i = A.length  down to 2
    {
        exchange a[1] with a[i]
```

```
        A.heapsize--;
        max_heapify(A,1)
    }
}
build_maxheap(A)
{
    A.heapsize = A.length
    for(i = A.heapsize)
    for(i = A.length/2 ; down to 1)
        max_heapify(A,i).
}
max_heapify(A,i)
{
    l = 2*i
    r = 2*i+1
    if(l <= A.heapsize && A[l] > A[i])
        largest = l else largest = i
    else if((r <= A.heapsize && A[r] > A[largest])
        largest = r.
    if(largest != i)
    exchange A[i] with A[largest]
        max_heapify(A, largest())
}
```

Date - 30/08/19

HASHING -

Q What is hashing?

Answer: * Hashing is the process of maping large amount of data item to smaller table with the help of hashing function.

* Hashing is also known as message digest function.

* It is a technique to convert a range of key values into range of indexes of an array.

* It is used to facilitate next level searching method when compared with linear & binear search.

* Hashing allow to update & retrive data entry in a constant time i.e., O(1). with the help of hashing function.

* Constant time O(1) means operation doesnt depend on size of data.

* Hashing is used with a database to enable to be retrive more quickly.

Q What is hash function ?.

Answer A fixed process that convert a key & map it to a value of certain length which is called hash value or hash.

Hash value represents the original string of character, but it is normally smaller than the original value. The parameters of good hashing function are i.e., easy to complute, even distribution & minimizle collision

Q What is hash table ?.

Answer * Hash table or hash map is a datastructure used to store key value pairs.

* It is a collection of item store to make it easy to find them later.

* It uses a hash function to compute an index into an array of buckets or slot from which the desired value can be formed.

* It is an array of list where each list is known as bucket (a node of link list

* It contains values based on the key.

* Hash table is used to implement the map interface & extends the dictionary class.
* Hash table is synchronized & contains only unique elements.

HASH FUNCTION TYPES - (Algorithm) (H.W)

(i) 1> DIVISION -

M = 1000 (0 - 999) (No. of memory slots)
Key = 12345789
H(Key) = Key % M

(ii) M1> SQUARE METHOD -

M = 1000 (0 - 999)
Key = 8492
H(Key) = $(8492)^2$ = 72114064
= 114 or 140.

(iii) Digit Extraction Method -

M = 10 (0 - 9)	M = 1000 (0 - 999)
Key = 1 2 3 4 5 6 7 8 9 (0 1 2 3 4 5 6 7 8 9)	Key = 1 2 3 4 5 6 7 8 9 10 (0 1 2 3 4 5 6 7 8 9)
H(Key) = 5 (4th position)	H(Key) = 260 (1, 5, 7)

(iv) Folding Method -

- Field Boundary
- Fold & Shifting method.

(a) Fold Boundary Method -

M = 1000 (0-999)

Key = 123456789

H(Key) = 123 + 789 = 912 (index)

Key = 789456789.

H(Key) = 789 + 789 = 1578 → 157 + 8 = 165 (index)

(b) Fold Shifting Method -

M = (0-999)

Key = 123456789

H(Key) = 123 + 456 + 789 = 1368 → 136 + 8 = 144 (index)

```
  123
  456
  789
 ----
 1368
```

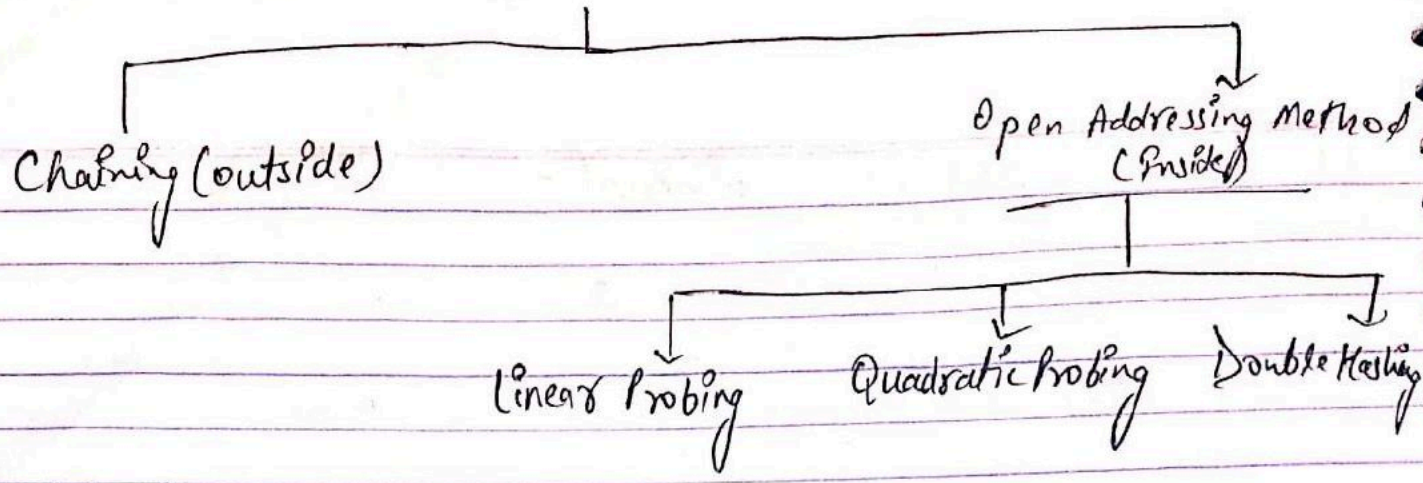
(v) Binning Method -

M = 10 (0-9)

Key Value Range (0-999)

0-99	Slot 0.
100-199	Slot 1
200-299	Slot 2
300-399	Slot 3
:	:
:	:
900 999	slot 9

COLLISION RESOLUTION TECHNIQUES -



COLLISION -

In hashing hash function used to compute hash value for a Key. Has value is then used as an index to store the Key in the hash table. Hash function may return the same hash value for two or more Key. When the hash value of a Key map to an already occupy bucket of hash table it is called collision

COLLISION RESOLUTION TECHNIQUE

Collision Resolution Technique is used for resolving or handling the collision. These techniques are classified as above given.

* Separate Chaining -

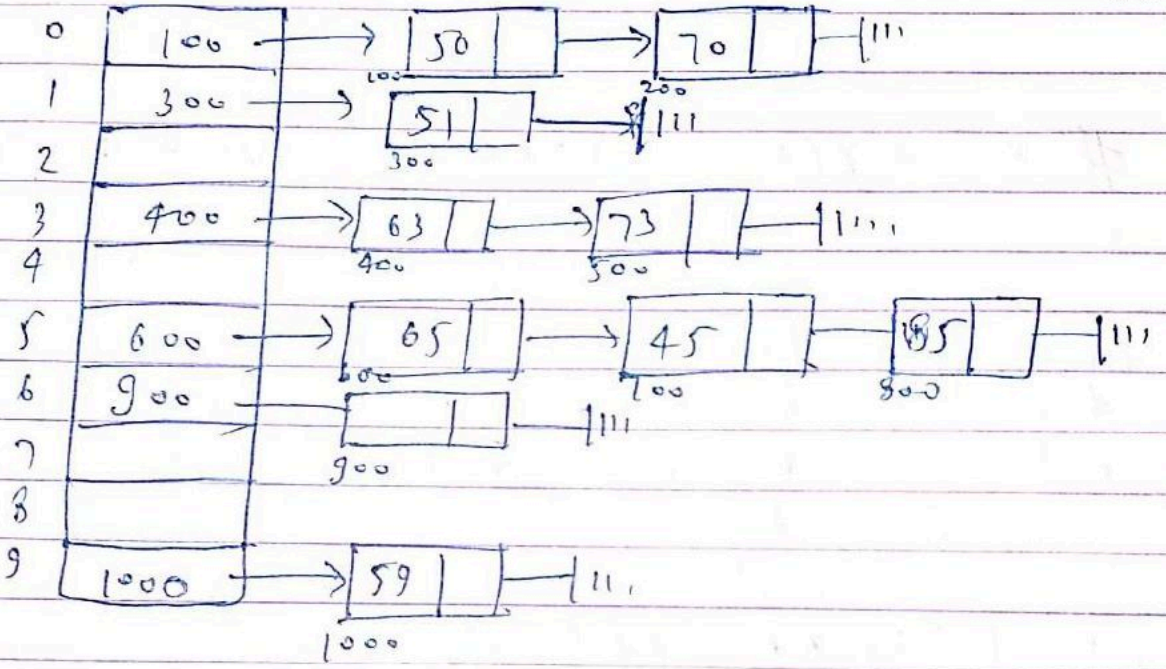
To handle the collision this technique creates a link list to the slot for which collision occurs. The new Key is then inserted in the link list. These link list to the slot appears like a chain that's why this technique is known as

Seperate chaining.

NOTE - The hash table contains array of pointers (addresses) which points to the first node of list of corresponding hash Key index.

ex- m = 10. (0-9)

elements = 50, 70, 51, 59, 65, 45, 85, 63, 73, 86



- TIME COMPLEXITY FOR SEARCHING -

(a) In worst case all the Key might mapped to the same bucket of hash table.

(b) In such case all Key will present in single link list. So, time taken for searching in worst case is O(n)

- LOAD FACTOR (α)

$$\alpha = \frac{\text{No. of element present in hash table}}{\text{Total size of hash table}}$$

Time complexity = $O(\alpha)$

ex- If elements lies (0-99) & $M = 10$.

$\therefore \alpha = \frac{100}{10} = 10.$

* OPEN ADDRESING METHOD -

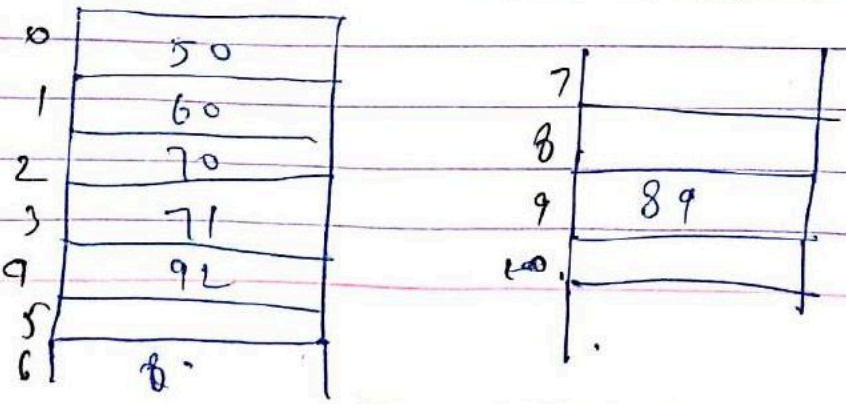
In open addressing unlike separate chaining all the Keys are stored inside the hash table i.e., no Key is stored outside the hash table memory.

Various type of open adressing method are-

(i) LINEAR PROBING -

When a collision occurs we linearly probe for the next bucket. We Keep probing untill an empty bucket is found.

ex- $M = 10$, elements = 50, 60, 70, 71, 89, 92,



(ii) QUADRATIC PROBING.

In Quadratic Probing when collision occurs, we probe for i^2 th bucket in ith operation iteration. We keep probing untill an empty bucket is found.

Ex:- $m = 10$, elements - 50, 60, 70, 71, 92, 95, 89.

50 mod 10 = 0.
60 mod 10 = 0
$(60 + 1^2) \bmod 10 = 1$.
70 mod 10 = 0 coll.
$(70 + 1^2) \bmod 10 = 1$ coll.
$(70 + 2^2) \bmod 10 = 4$ coll.

0	50
1	60
2	71
3	92
4	70
5	95
6	
7	
8	
9	89.

(iii) DOUBLE HASHING -

In Double hashing we use another hash function say $h_2(x)$ & look for i th $h_1(x)$ bucket in ith iteration. It requires more computation time as two hash function needed to be computed.

UNIT-4

Date - 10/10/19

GRAPH

A Graph consist of two sets.

(i) A non-empty set V whose elements are called vertices.

(ii) A set of E of edges such that $e \in E$ associated with ordered or unordered pair of element of V. The set $E(G)$ is called edges set of G.

The Graph with vertices V & edges E is written as $G(V, E)$ or $G = (V, E)$

* If an edge $e \in E$ is associated with an ordered or unordered pair (u, v) where $u, v \in V$. Then, E is said to connect u & v; u & v are called end point of edge e.

* An edge is said to be incident with the vertices it join.

* The edge e that join vertices u & v it said to be incident on each of its end point u & v. i.e., connected by edge. e in a graph is called adjacent vertix.

ISOLATED VERTEX -

In a graph vertix that ~~have~~ is not adjacent to any vertix is called isolated vertix.

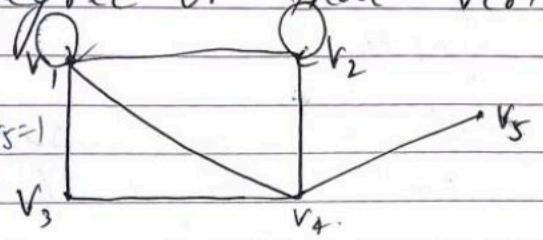
DEGREE OF VERTEX -

The degree of a vertix of an undirected graph is the number of edges incident with it

NOTE -

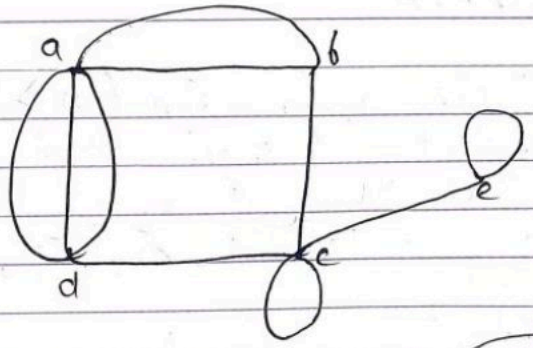
The loop at vertix contributes twice to the degree of that vertix.

ex -



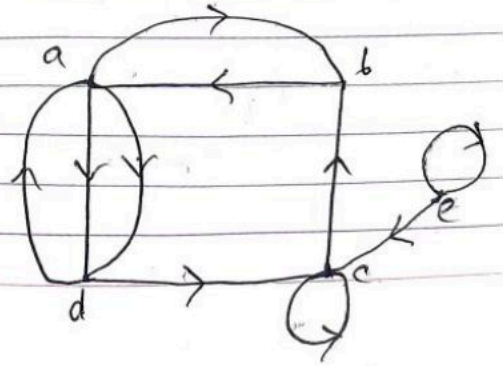
$v_1 \to 5$ | $v_3 = 2$ | $v_5 = 1$
$v_2 = 4$ | $v_4 = 4$ |

ex -



deg(a) = 5
deg(b) = 3
deg(c) = 5
deg(d) = 4
deg(e) = 3

Indeg(a) = 2, Outdeg(a) = 3
Indeg(b) = 2, Outdeg(b) = 1
Indeg(c) = 3, Outdeg(c) = 2
Indeg(d) = 2, Outdeg(d) = 2
Indeg(e) = 1, Outdeg(e) = 2



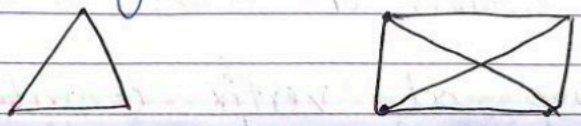
NULL GRAPH

Null graph is a graph which contain isolated ~~graph~~ vertix.

COMPLETE GRAPH

If all vertix are interconnected to all remaining vertix.

ex-

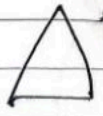
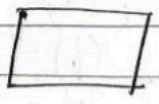
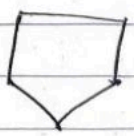


REGULAR GRAPH

If all vertix of graph have same degree then this graph is known as regular graph

CYCLE GRAPH

The Cycle graph C_n $(n \geq 3)$ of length n is a connected graph which consist of n vertices say $v_1, v_2, v_3, \ldots v_n$ & n edges $E_n = \{\{v_1, v_2\}, \{v_2, v_3\}, \ldots \{v_{n-1}, v_n\}, \{v_n, v_1\}\}$.

ex- C_3  , C_4  , C_5 

WALK

A walk in a graph G is a finite alternative sequence say $v_0 e_0, v_1 e_1, v_2 e_2, \ldots\ldots v_n e_n$ of vertices & edges of the graph such that edge e_i in the sequence joins vertices v_{i-1}, v_i where $1 \le i \le n$.

The end vertices v_0 & v_n are called end terminals of the walk whereas $v_1, v_2, \ldots\ldots v_{n-1}$ are called internal vertices of walk.

The number of edges in walk is called length of a walk.

* OPEN WALK

A walk is called open when its end terminals are distinct.

* CLOSED WALK

A walk is called closed when its end terminals are same.

NOTE- A walk may repeat both edge & vertices.

* TRAIL

A walk is called trail if all its edges are distinct but it can repeat the

Closed loop = Circuit

vertices.

A Trail is a open or closed depends on whether its end vertices are distinct or same.

* PATH -

A walk does not contain repeated vertix & edges is called Path.

SPANNING TREE -

A Spanning Tree is a subset of Graph G which has all the vertices covered with minimum possible no. of edges. Hence, a spanning tree does not have cycles & it cannot be disconnected.

Every connected & undirected graph G(V, E) has atleast one spanning tree.

A disconnected graph does not have any spanning tree as it cannot be spann to all its vertices

* PROPERTIES OF SPANNING TREE

(i) A graph G can have more than one spanning tree.

(ii) A complete undirected graph can have $(n^{(n-2)})$ where n spanning tree where n is no. of vertices.

(iii) All possible spanning tree of Graph G have same no. of vertices & edges.

... make graph disconnected it implies
Spanning tree minimally connected sub graph.
) Adding one edge to the spanning tree will create a circuit or closed loop it implies spanning tree is minimally acyclic subgraph.

MATHEMATICAL PROPERTIES OF S.T.

i) From complete graph by removing maximum (e - n + 1) edges we can construct a spanning tree. where e is no. of edges & n is no. of vertices

ii) A ~~con~~ Spanning tree has (n-1) edges where n is no. of vertices.

APPLICATIONS OF S.T.

i) Civil network planning.

ii) Computer network routing protocol.

iii) Cluster analysis.

MINIMAL SPANNING TREE -

~~Inwaited~~ Inweighted graph, a minimal spanning tree is a spanning tree that have minimum weight, that all other spanning

(iv) Removing one edge from spanning tree will make graph disconnected it implies spanning tree minimally connected sub graph.

(v) Adding one edge to the spanning tree will create a circuit or closed loop it implies spanning tree is minimally acyclic subgraph.

* MATHEMATICAL PROPERTIES OF S.T.

(i) From complete graph by removing maximum $(e-n+1)$ edges we can construct a spanning tree. where e is no. of edges & n is no. of vertices

(ii) A Spanning tree has $(n-1)$ edges where n is no. of vertices.

* APPLICATIONS OF S.T.

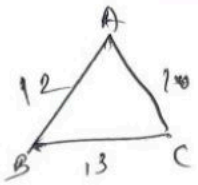
(i) Civil network planning.

(ii) Computer network routing protocol.

(iii) Cluster analysis.

MINIMAL SPANNING TREE -

In weighted graph, a minimal spanning tree is a spanning tree that have minimum weight, that all other spanning

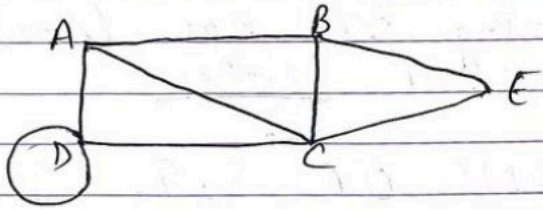


tree of same graph. In real world situation, this weight can be measured as distance, congestion, traffic load or any arbitrary value denoted to edge.

ADJACENCY MATRIX -

A connected graph G(V, E) can be represent into a 2-Dimensional adjacency matrix of size nxn where n is no. of vertices. By following value. A[i][j] = 1 if there exist an edge b/w from ith vertix to jth vertix. A[i][j] = 0. if there is no edge b/w ith & jth vertix.

Ex-



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

5x5

SPANNING TREE (ALGORITHM FOR CONSTRUCTING) USING BREADTH FOR SEARCH (BFS).

Step 1 - Choose (Arbitrary) a vertix & designate it as the root.

Step 2 - Add all edges incident to this vertix, such that addition of edges doesnot produce cycle or circuit.

Step 3 - A new vertices added in previous

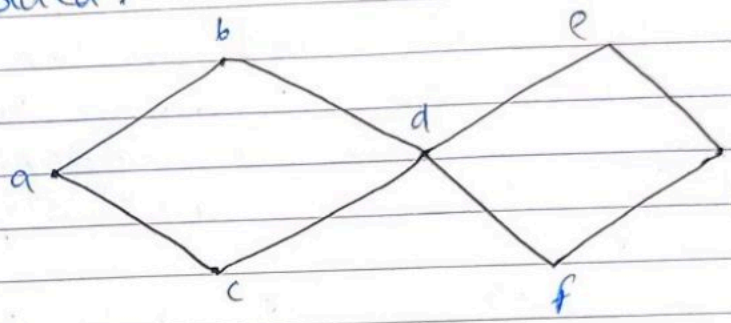
step become vertices at level 1 in th spanning tree, arbitrary order them

Step 4 - for each vertix of level 1 visit in order, add each edge incident to this vertix to the tree as long a it doesnot produce any cycle or circ

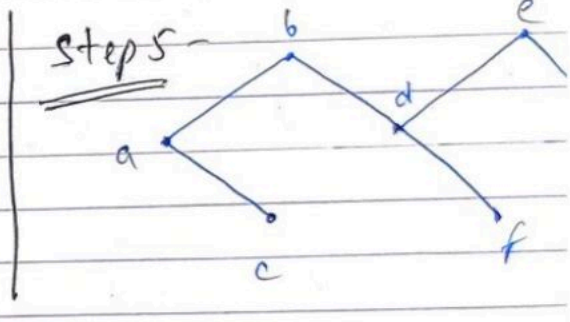
Step 5 - Arbitrary order the vertices at level 2.

Step 6 - Continue the same procedure until all the vertices in the tree have been added.

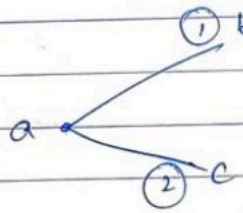
ex -



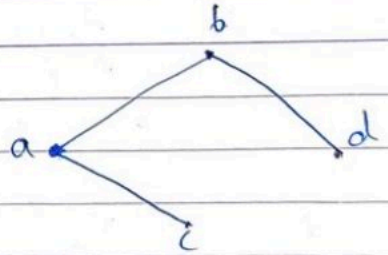
Step 1 -



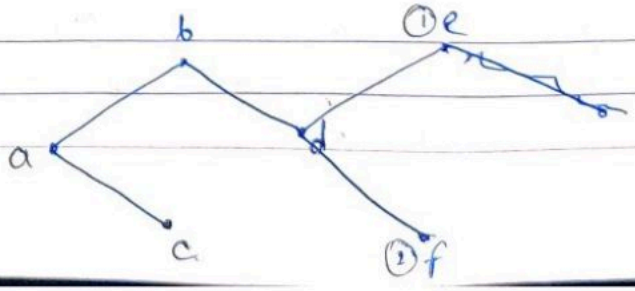
Step 2 -



Step 3 -



Step 4 -



Step 5 -

DEPTH FIRST SEARCH ALGORITHM

Step 1- Arbitrary choose a vertix from a graph G & designate it as root.

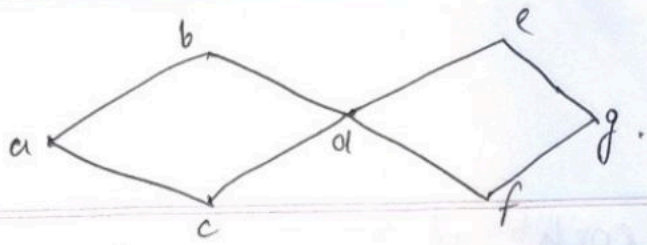
Step 2- form a path starting at this vertix by successively adding edges as long as possible where each new edge is incident with the last vertix in the path without producing any cycle.

Step 3- If the path goes through all the vertices of graph. The tree consisting of this path is a spanning tree. Otherwise go to the next step.

Step 4- Move back to the next to last vertix in the path & if possible, form a new path starting at this vertix passing through vertices that where not already visited.

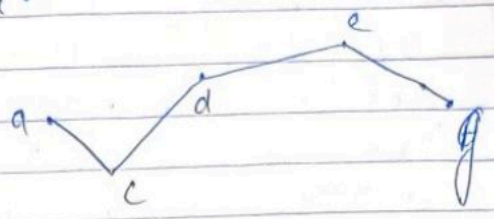
Step 5- If this cannot be done move back to another vertix in the path i.e., two vertix back in the path, repeat this procedure beginning at the last vertix visited.

Step 6- Moving back up the path one vertix at a time, forming new path that are as long as possible untill no more edges can be added.

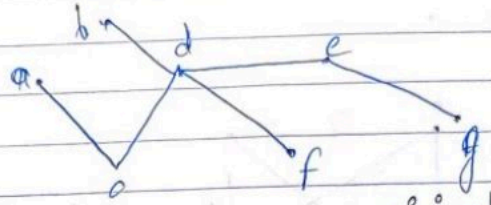


Step 1- a

Step 2-



Step 3-



Loop → within a vertex.

Circuit →

MST (MINIMUM SPANNING TREE).- (Based upon minimum weight).
→ or minimal

~~KRUSKAL's~~

KRUSKAL's ALGORITHM -

This Algorithm provides an acyclic graph T of a connected waited graph G which is minimum spanning tree of G.

I/P

O/P - MST T

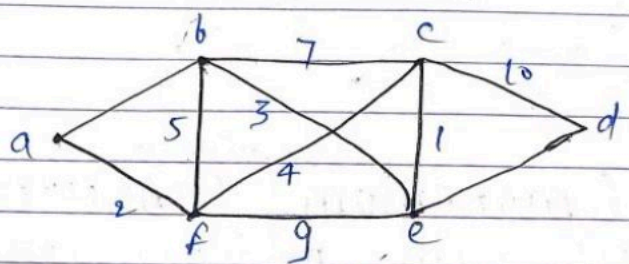
Step 1- List all the edges (which do not form a loop) of G in non-decreasing (increasing) order of their.

Step 2- Select an edge of minimal weight (if more than one edge of minimum weight, arbitrary choose one of them.

Step 3 - At each stage, select an edge of minimum weight from all the remaining edge of G. If doesnot form a circuit with the previously selected edge in T

Step 4 - Repeat the step 3 untill (n-1) edges are selected where n is no. of vertices.

Ex -

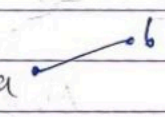


Step 1 -

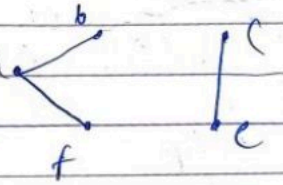
edges	ab	ce	af	be	cf	de	bf	bc	ef	cd
weight	1	1	2	3	4	4	5	7	9	10

Step 2 - $T = \{ab\}$

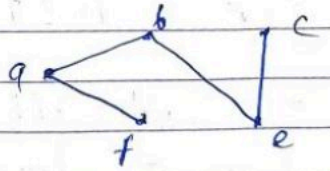
$T = \{(ab), (ce)\}$.



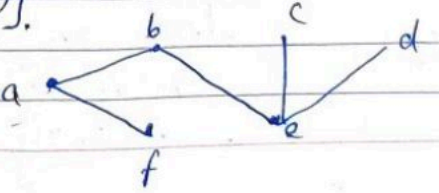
Step 3 - $T = \{(a,b), (c,e), (af)\}$



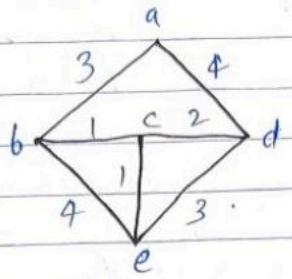
$T = \{(ab), (ce), (af), (be)\}$



$T = \{(ab), (ce), (af), (be), (de)\}$.



Q Draw the MST of the given graph with help of Kruskal's algorithm.



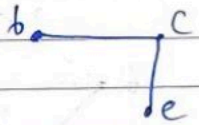
Answer Step 1:-

edge	ab	bc	ce	cd	de	ab	be	ad
weight	1	1	2	3	3	4	4	

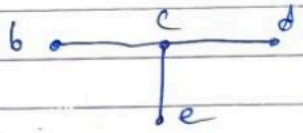
Step 2 -

$T = \{bc\}$.

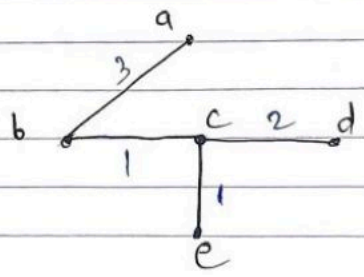
$T = \{(bc), (ce)\}$



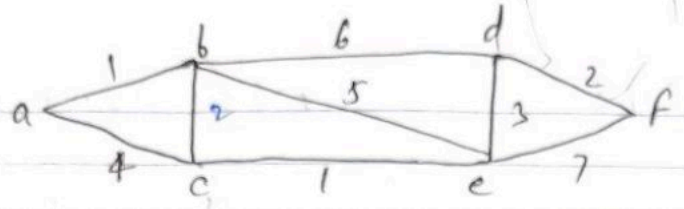
Step 3 - $T = \{(bc), (ce), (cd)\}$



$T = \{(bc), (ce), (cd), (ab)\}$



ex-



Initial label is given by table.

Vertex	a	b	c	d	e	f
L(V)	0	∞	∞	∞	∞	∞
T	a	b	c	d	e	f

* Iteration 1- u = a has L(u) = 0. T becomes T-{a}. There are 2 edges incident with a i.e., ab & ac where b & c ∈ T.

$L(b) = \min(\text{old}(L(b)), L(a) + w(a,b))$
$= \min(\infty, 0+1) = 1$

$L(c) = \min(\text{old}(L(c)), L(a) + w(a,c))$
$= \min(\infty, 0+4) = 4$

Since, L(b) < L(c) minimum label is of b. So, b vertex is choosen L(b) = 1.

Vertex	a	b	c	d	e	f
L(V)	0	1	4	∞	∞	∞
T		b	c	d	e	f

* Iteration 2- u = b has L(u) = 1. T becomes T-{b}. There are 3 edges incident with b i.e., bc, bd, be where c, d, e ∈ T.

$L(c) = \min(\text{old}(L(c)), L(b) + w(b,c))$
$= \min(4, 1+2) = 3.$

$L(d) = \min(\infty, 1+6) = 7$

$L(e) = \min(\infty, 1+5) = 6$

Since, $L(c) < L(e) < L(d)$. minimum label is of c. So, c vertex is choosen $L(c) = 3$.

Vertex	a	b	c	d	e	f
L(V)	0	1	3	7	6	∞
T			c	d	e	f

* Iteration 3 - $u = c$ has $L(u) = 3$. T becomes $T - \{c\}$. There are only one edge incident with e i.e., ce where $ce \in T$.

$L(e) = \min(\text{old}(L(e)), L(c) + w(c,e))$
$= \min(6, 3+1) = 4$

Since, only one vertex is there. minimum label is of e. So, e vertex is choosen $L(e) = 4$

Vertex	a	b	c	d	e	f
L(V)	0	1	3	7	4	∞
T				d	e	f

* Iteration - 4 $u = e$ has $L(e) = 4$. T becomes $T - \{e\}$. There are two edges incident with e i.e., ed & ef where $d \& f \in T$.

$L(d) = \min(\text{old}(L(d)), L(e) + w(e,d))$
$= \min(7, 4+3) = 7$.
$L(f) = \min(\infty, 4+7) = 11$.

Since, $L(d) < L(f)$ minimum label is of d So, d vertex is choosen $L(d) = 7$.

Vertex	a	b	c	d	e	f
L(v)	0	1	3	7	4	18
T				d		f

* Iteration - 5 $u = d$. has $L(d) = 7$. T becomes $T - \{d\}$. There are only edge incident with d i.e., df where $f \in T$.

$L(f) = \min(\text{old}(L(f)), L(d) + w(d,f))$
$= \min(18, 7+2) = 9$.

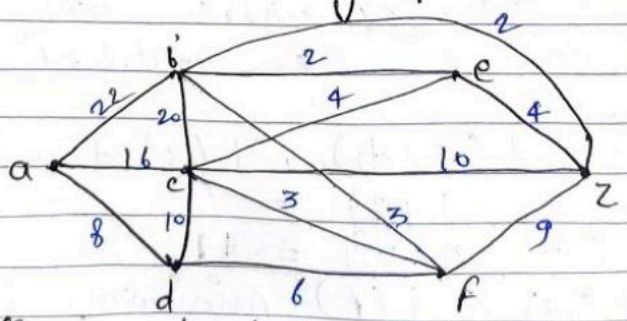
Since, only one vertex is there so, minimum label is of f, f vertex is choosen $L(f) = 9$

Vertex	a	b	c	d	e	f
L(v)	0	1	3	7	4	9
T						f

Min. Path for moving a to f.

$a \to b \to c \to e \to d \to f$

Q In below graph



find the shortest distance b/w a & z.
Using Digikstra's Rule -

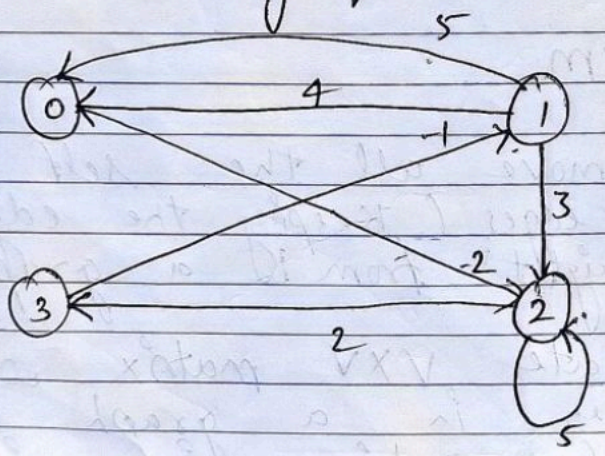
STEP-3

For each vertex K, where K is an intermediate vertex from i to j vertex. Calculate a distance matrix.

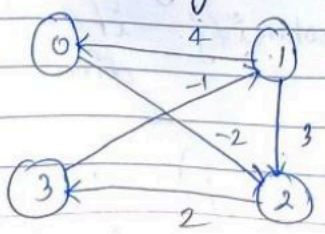
```
for(K=0 to K<=V-1)
    for( i=0 to i<=V-1)
        for (j=0 to V-1)
            if((d[i][K] + d[K][j]) < d[i][j])
                d[i][j] = d[i][K] + d[K][j]
```

Q Consider a graph G.

=



Step 1- Remove all self loop & parallel edges - the graph will be -



$$D = \begin{array}{c} \\ 0 \\ 1 \\ 2 \\ 3 \end{array} \begin{array}{c} \begin{array}{cccc} 0 & 1 & 2 & 3 \end{array} \\ \begin{bmatrix} 0 & \infty & -2 & \infty \\ 4 & 0 & 3 & \infty \\ \infty & \infty & 0 & 2 \\ \infty & -1 & \infty & 0 \end{bmatrix} \end{array}$$

Weight Matrix (Vertex X Vertex) of a Graph

$$D_0 = \begin{array}{c} \\ 0 \\ 1 \\ 2 \\ 3 \end{array} \begin{array}{c} \begin{array}{cccc} 0 & 1 & 2 & 3 \end{array} \\ \begin{bmatrix} 0 & \infty & -2 & \infty \\ 4 & 0 & 2 & \infty \\ \infty & \infty & 0 & 2 \\ \infty & -1 & \infty & 0 \end{bmatrix} \end{array}$$

$$D_1 = \begin{array}{c} \\ 0 \\ 1 \\ 2 \\ 3 \end{array} \begin{array}{c} \begin{array}{cccc} 0 & 1 & 2 & 3 \end{array} \\ \begin{bmatrix} 0 & \infty & -2 & \infty \\ 4 & 0 & 2 & \infty \\ \infty & \infty & 0 & 2 \\ 3 & -1 & 1 & 0 \end{bmatrix} \end{array}$$

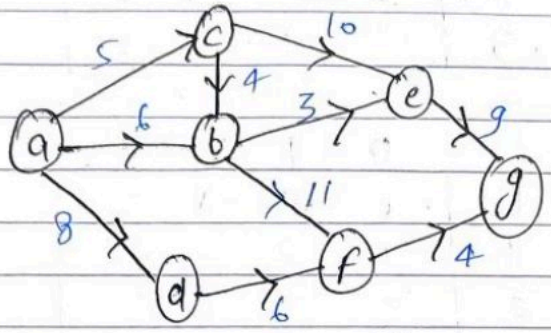
$$D_2 = \begin{array}{c} \\ 0 \\ 1 \\ 2 \\ 3 \end{array} \begin{array}{c} \begin{array}{cccc} 0 & 1 & 2 & 3 \end{array} \\ \begin{bmatrix} 0 & \infty & -2 & 0 \\ 4 & 0 & 2 & 4 \\ \infty & \infty & 0 & 2 \\ 3 & -1 & 1 & 0 \end{bmatrix} \end{array}$$

$$D_3 = \begin{array}{c} \\ 0 \\ 1 \\ 2 \\ 3 \end{array} \begin{array}{c} \begin{array}{cccc} 0 & 1 & 2 & 3 \end{array} \\ \begin{bmatrix} 0 & -1 & -2 & 0 \\ 4 & 0 & 2 & 4 \\ 5 & 1 & 0 & 2 \\ 3 & -1 & 1 & 0 \end{bmatrix} \end{array}$$

Above matrix is required matrix.

Sorting RMO CMO, Postfix,

Q Find the minimum distance b/w every pair of vertices of following graph.



$D =$

	a	b	c	d	e	f	g
a	0	6	5	8	∞	∞	∞
b	∞	0	∞	∞	3	11	∞
c	∞	4	0	∞	10	∞	∞
d	∞	∞	∞	0	∞	6	∞
e	∞	∞	∞	∞	0	∞	9
f	∞	∞	∞	∞	∞	0	4
g	∞	∞	∞	∞	∞	∞	0

$D_a =$

	a	b	c	d	e	f	g
a	0	6	5	8	∞	∞	∞
b	∞	0	∞	∞	3	11	∞
c	∞	4	0	∞	10	∞	∞
d	∞	∞	∞	0	∞	6	∞
e	∞	∞	∞	∞	0	∞	9
f	∞	∞	∞	∞	∞	0	4
g	∞	∞	∞	∞	∞	∞	0

3, 4 or 5

$D_b =$

	a	b	c	d	e	f	g
a	0	6	5	8	9	17	∞
b	∞	0	∞	∞	3	11	∞
c	∞	4	0	∞	7	15	∞
d	∞	∞	∞	0	∞	6	∞
e	∞	∞	∞	∞	0	∞	9
f	∞	∞	∞	∞	∞	0	4
g	∞	∞	∞	∞	∞	∞	0

$D_c =$

	a	b	c	d	e	f	g
a	0	6	5	8	9	17	∞
b	∞	0	∞	∞	3	11	∞
c	∞	4	0	∞	7	∞	∞
d	∞	∞	∞	0	∞	6	∞
e	∞	∞	∞	∞	0	∞	9
f	∞	∞	∞	∞	∞	0	4
g	∞	∞	∞	∞	∞	∞	0

$D_d =$

	a	b	c	d	e	f	g
a	0	6	5	8	9	14	∞
b	∞	0	∞	∞	3	11	∞
c	∞	4	0	∞	10	∞	∞
d	∞	∞	∞	0	∞	6	∞
e	∞	∞	∞	∞	0	∞	9
f	∞	∞	∞	∞	∞	0	4
g	∞	∞	∞	∞	∞	∞	0

$D_e =$

	a	b	c	d	e	f	g
a	0	6	5	8	9	14	18
b	∞	0	∞	∞	3	11	12
c	∞	4	0	∞	10	∞	19
d	∞	∞	∞	0	∞	6	∞
e	∞	∞	∞	∞	0	∞	9
f	∞	∞	∞	∞	∞	0	4
g	∞	∞	∞	∞	∞	∞	0

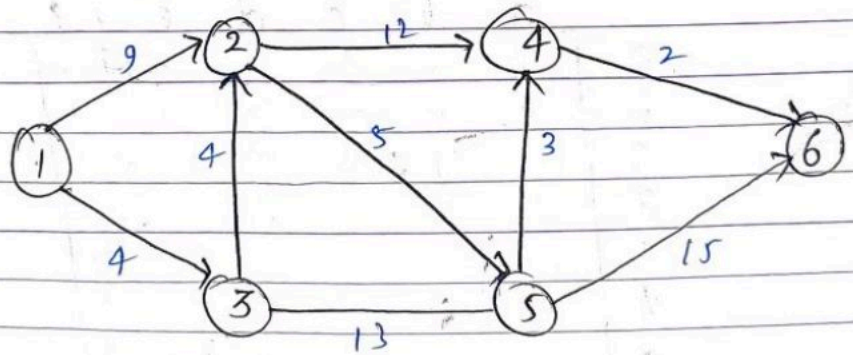
P_f =

	a	b	c	d	e	f	g
a	0						14
b							11
c							∞
d							6
e							∞
f	∞	∞	∞	∞	∞	0	4
g							∞

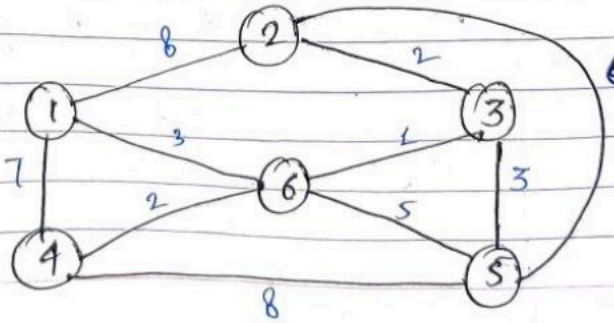
P_g =

a							
b							
c							
d							
e							
f							
g							

Q 1 –



Q 2 –



Solution - 2

$D =$

	1	2	3	4	5	6
1	0	8	∞	7	∞	3
2	8	0	2	∞	6	∞
3	∞	2	0	∞	3	1
4	7	∞	∞	0	8	2
5	∞	6	3	8	0	5
6	3	∞	1	2	5	0

$D_1 =$

	1	2	3	4	5	6
1	0	8	∞	7	∞	3
2	8	0	2	15	6	11
3	∞	2	0	∞	3	1
4	7	15	∞	0	8	2
5	∞	6	3	8	0	5
6	3	11	1	2	5	0

$D_2 =$

	1	2	3	4	5	6
1	0	8	10	7	14	3
2	8	0	2	15	6	11
3	10	2	0	17	3	1
4	7	15	17	0	8	2
5	14	6	3	8	0	5
6	3	11	1	2	5	0

$D_3 =$

	1	2	3	4	5	6
1	0	8	10	7	13	3
2	8	0	2	15	5	3
3	10	2	0	17	3	1
4	7	15	17	0	8	2
5	13	5	3	8	0	4
6	3	3	1	2	4	0

$D_4 =$

	1	2	3	4	5	6
1	0	8	10	7	13	3
2	8	0	2	15	5	3
3	10	2	0	17	3	1
4	7	15	17	0	8	2
5	13	5	3	8	0	4
6	3	3	1	2	4	0

$D_5 =$

	1	2	3	4	5	6
1	0	8	10	7	13	3
2	8	0	2	13	5	3
3	10	2	0	11	3	1
4	7	13	11	0	8	2
5	13	5	3	8	0	4
6	3	3	1	2	4	0

$D_6 =$

	1	2	3	4	5	6
1	0	6	4	5	7	3
2	6	0	2	5	5	3
3	4	2	0	3	3	1
4	5	5	3	0	6	2
5	7	5	3	6	0	4
6	3	3	1	2	4	0

Date - 23/10/19

UNIT - V

TREE

Definition

Properties

Basic Terminology -

(a) Root (b) Edge (c) Parent (d) Child (e) Sibling
(f) Degree (g) Internal Nodes (h) Leaf Node
(i) Level (j) Height (k) Depth (l) Subtree
(m) Forest.

BINARY TREE -

Definition
Unlabelled Binary Tree → formula depend upon height depth
Labelled Binary Tree
Types of Binary Tree -

(a) Rooted B.T.
(b) Full / Strictly B.T.
(c) Complete B.T.
(d) Almost Complete B.T.
(e) Skewed B.T.
Binary Search Tree.
Various formula or Properties of B.T.

A Graph in which no circuit is formed or self looping is K/a tree.
A Node is collection of data & address.

BINARY TREE -

Binary Tree is the tree in which every node can have atmost two children. Each node can have 0, 1 or 2 children but not more than two

* Unlabelled Binary Tree -

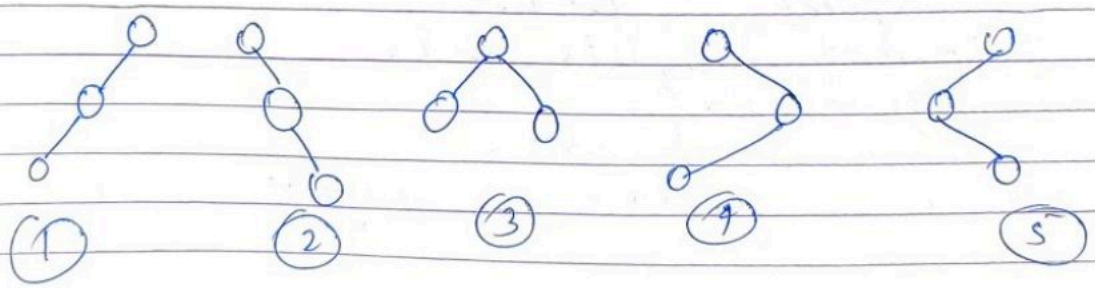
A binary tree is unlabelled if its node are not assign any label.

No. of different binary tree possible with n unlabelled node $= \frac{2nC_n}{n+1}$.

ex- If we have three node then how many binary tree possible.

$$\frac{2\times3C_3}{3+1} = \frac{6!}{3!(6-3)!} \Big/ 4 = 5.$$

$\frac{6\times5\times4\times3!}{4\times3!3!}$

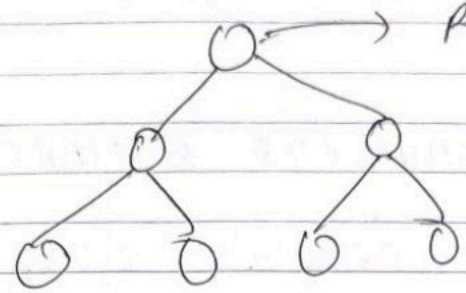


No. of binary tree possible with n labelled nodes $= \left(\frac{2nC_n}{n+1}\right) * n!$

* TYPES OF BINARY TREE -

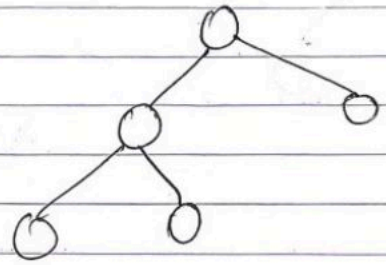
(i) ROOTED BINARY TREE -

A rooted binary tree is one that has a root node & every node has atmost two children



(ii) FULL OR STRICTLY BINARY TREE -

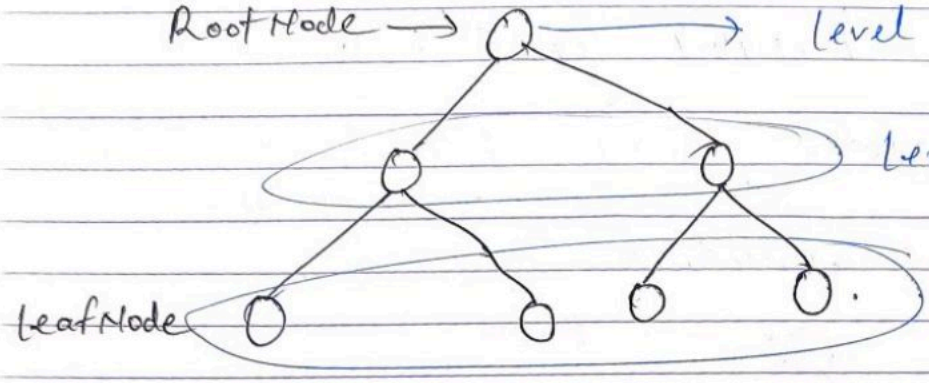
A full or strictly binary tree is a tree in which every ~~children has~~ nodes has 0 or 2 children



(iii) COMPLETE BINARY TREE (PERFECT B.T.)

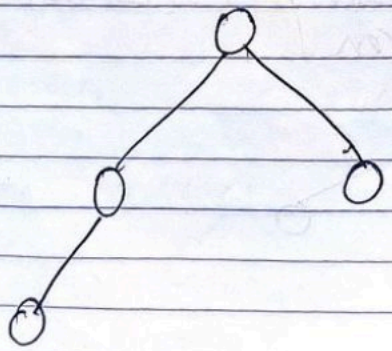
A Complete or perfect binary tree is a tree in which every internal node has exactly two children & all ~~nod~~ leaf

node are at same level



(iv) ALMOST COMPLETE BINARY TREE-

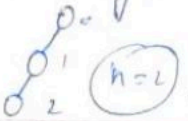
An almost complete binary tree is a tree in which all the levels are completely filled except possibly the last level & the last level must be strictly filled from left to right.



(v) SKEWED BINARY TREE -

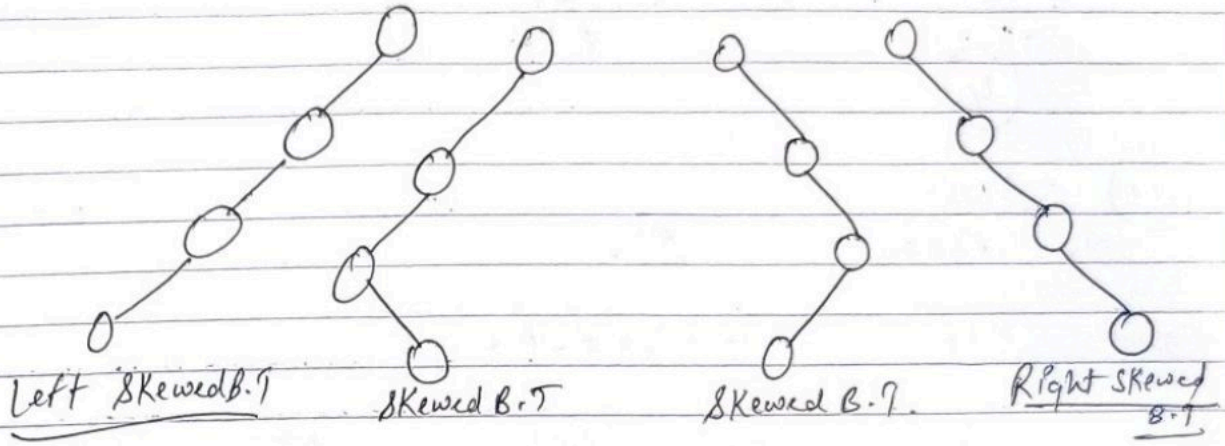
A skewed binary tree is the tree in which all the node except one node have one & only one child the

Height of the tree is a max. no. of edges b/w root node to last node.



remaining nodes has no child.

A SKewed binary tree is a tree of n node of n node such that it depth is n-1



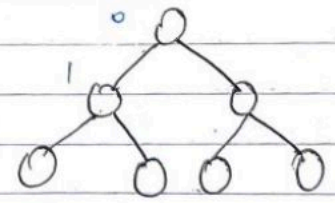
* PROPERTIES OF BINARY TREE -

(i) Minimum no. of node in a binary tree of height h is equal to h+1.

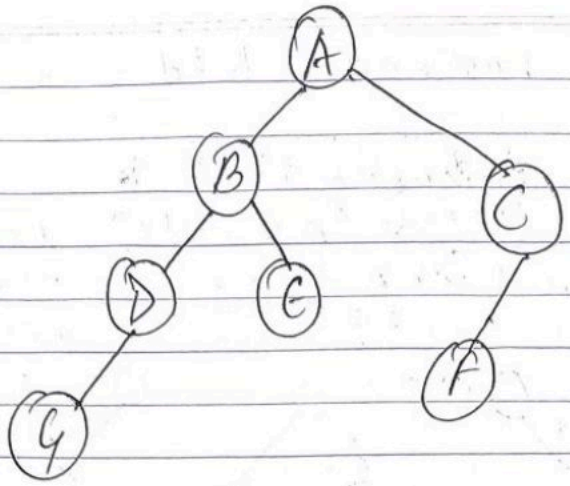
(ii) Maximum no. of node in a binary tree of height h is equal to $2^{h+1} - 1$

(iii) Total no. of leaf node in a binary tree is equal to total no. of degree two node + 1

Ex-



h = 3 =>
leaf node = 3+1 = 4.



(iv) Maximum no. of node at level l is equal to 2^l.

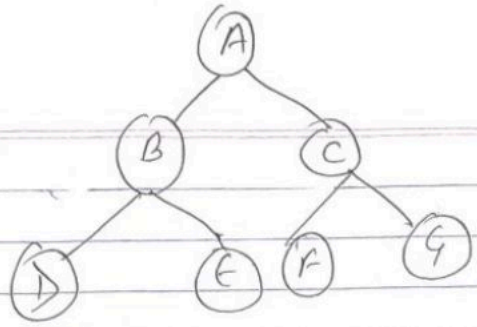
TREE TRAVERSAL -

It is also known as tree search or tree display. Tree traversal is a form of graph traversal & refirst to the process of visiting each node in a tree data structure exactly once. Tree traversal techniques are depth first search which includes three techniques (i) Preorder Tree Traversal
(ii) Inorder " "
(iii) Postorder " "

* ALGORITHM OF TREE TRAVERSAL

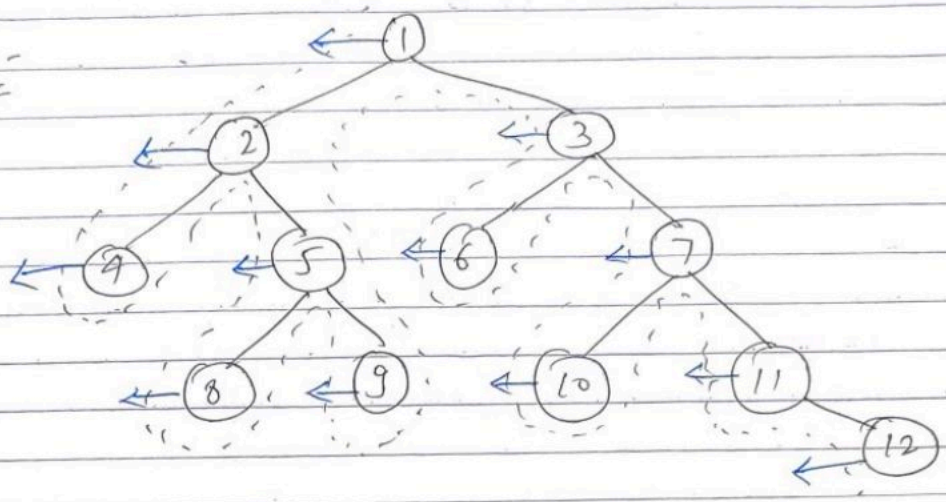
(i) Visit the root.
(ii) Traverse left subtree i.e., called Preorder left subtree.
(iii) Traverse right subtree i.e., called Preorder right subtree.

ex-



A B D E C F G.

ex-



1, 2, 4, 5, 8, 9, 3, 6, 7, 10, 11, 12

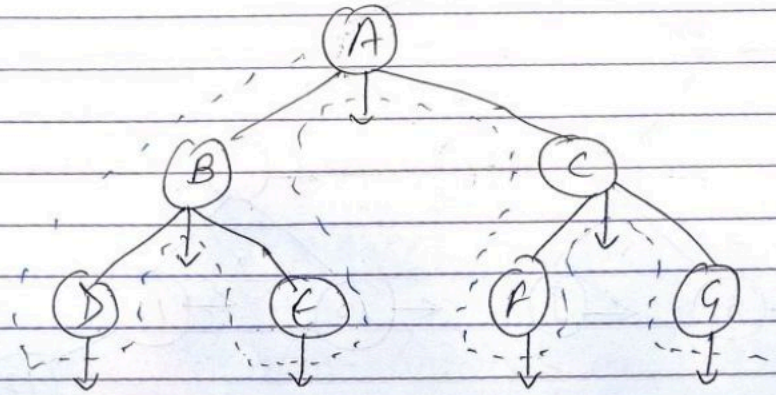
Write a algo to traverse a tree in preorder.

```
void Preorder (struct node *root)
{
    if (root != NULL)
    {
    printf ("%d \t", root -> data);
    preorder (root -> left);
    preorder (root -> right);
    }
}
```

* ALGORITH FOR INORDER.

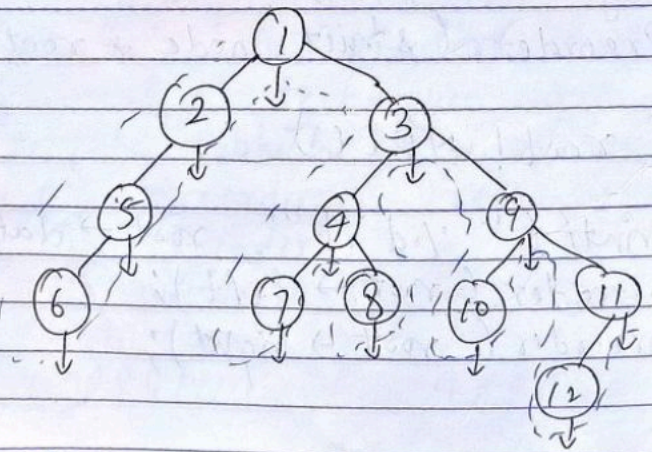
(1) TRAVERSE the left subtree i.e., call the inorder left subtree
(2) Visit the root.
(3) Traverse the right subtree i.e., call inorder right subtree.

ex -



D, B, E, A, F, C, G.

ex -



6, 5, 2, 1, 7, 4, 8, 3, 10, 9, 12, 11.

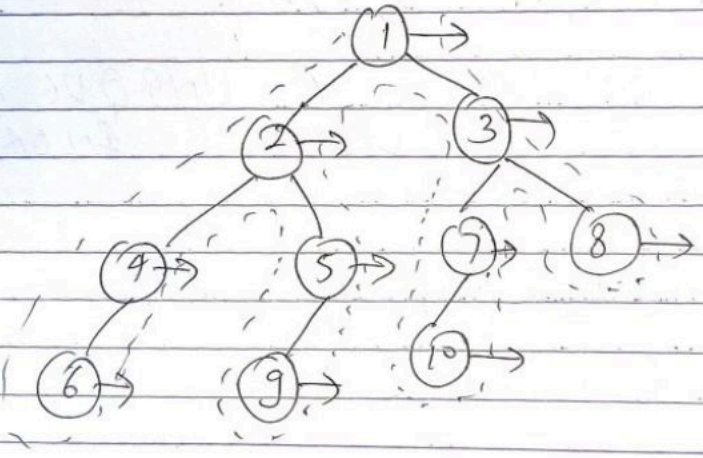
Q WAP in C to implement algo of inorder tree traversal.

```
void inorder ( struct node * root )
{
    if ( root != NULL)
    {
        inorder (root → left);
        printf ("%d\t", root → data);
        inorder ( root → right );
    }
}.
```

ALGORITHM OF POSTORDER

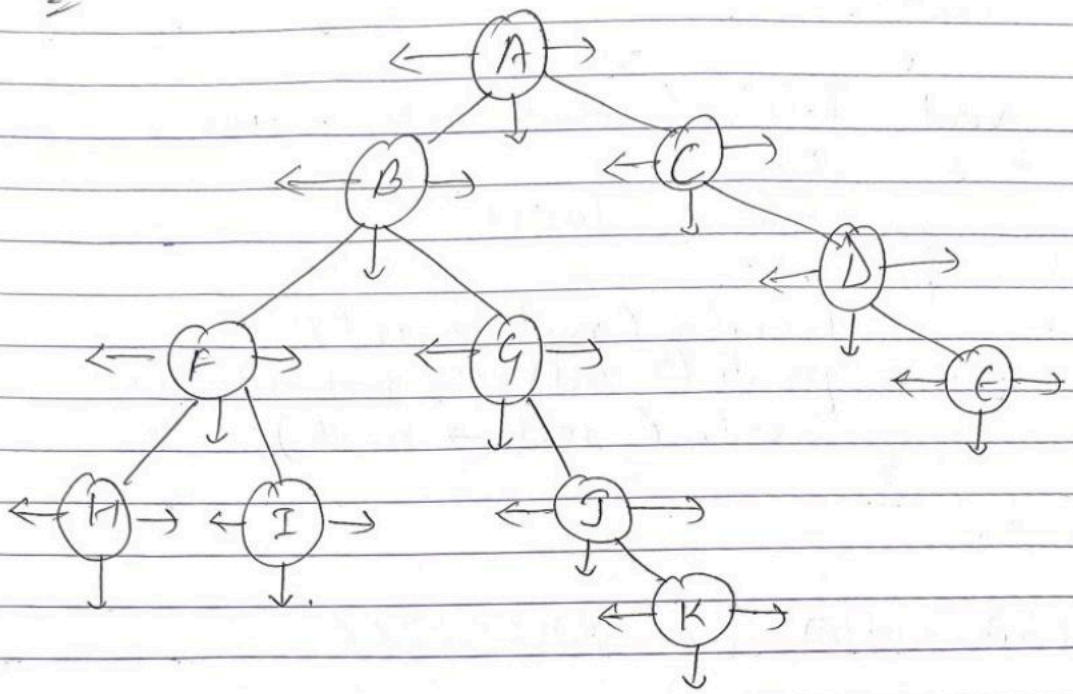
1. Traverse the left subtree i.e., called postorder left subtree.
2. Traverse the right subtree i.e., call postorder right subtree.
3. Visit the root.

Ex -



6, 4, 9, 5, 2, 10, 7, 8, 3, 1.

Q. Construct Preorder, Inorder & Postorder.



Pre A, B F, H, I, G, J, K, C, D, E.

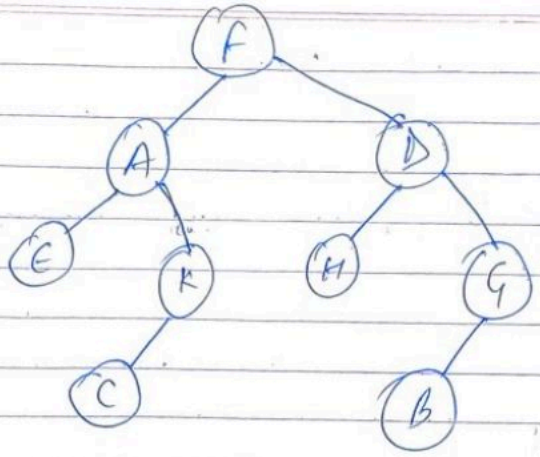
In H, F, I, B, G, J, K, A, C, D, E

Post H, I, F, K, J, G, B, E, D, C, A.

Q # TO CONSTRUCT A UNIQUE BINARY TREE WHEN PREORDER & INORDER IS GIVEN-

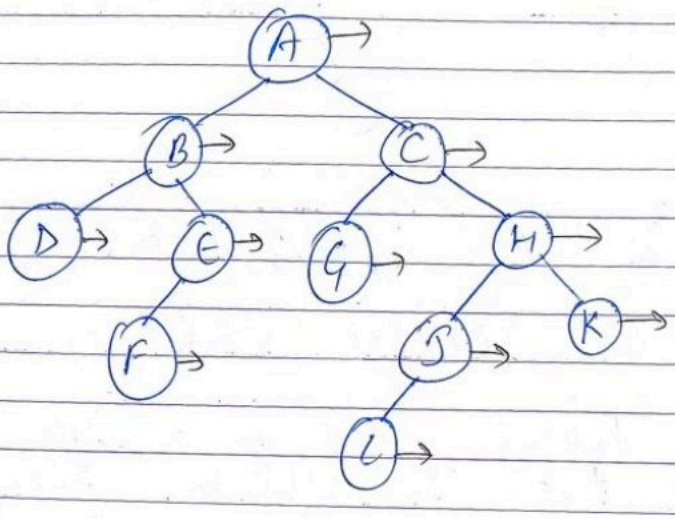
INORDER - E, A, C K, F, H, D, B, G.

PREORDER - F, A, E, K, C, D, H, G, B.



INORDER - D, B, F, E, A, G, C, L, J, H, K.

PREORDER - A, C, H, K, J, L, G, B, E, F, D.



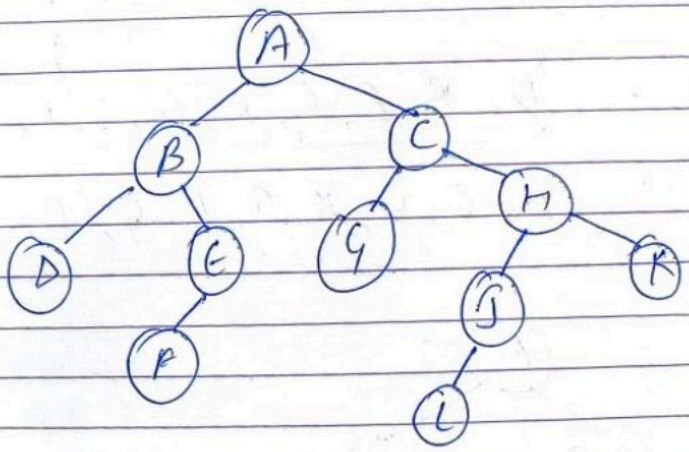
D, F, E, B, G, L, J, K, H, C, A.

HOW TO CONSTRUCT A TREE WHEN POSTORDER & INORDER ARE GIVEN -

INORDER D, B, F, E, A, G, C, L, J, H, K

POSTORDER D, F, E, B, G, L, J, K, H, C, A.

FIND PREORDER.



Preorder - A, B, D, E, F, C, G, H, J, L, K.

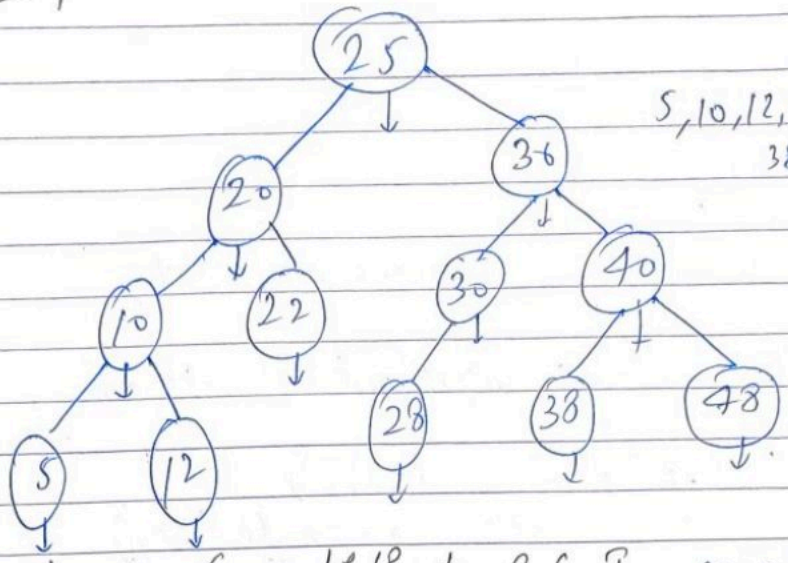
BINARY SEARCH TREE - (B.S.T.)

Binary Search Tree is a special kind of binary tree in which every node are arranged in specific orders.

In binary Search Tree, each node contains :

1. Only smaller value in its left sub tree.
2. Only greater value in its right sub tree.

for example: 25, 20, 10, 5, 12, 22, 36, 30, 28, 40, 38, 48.



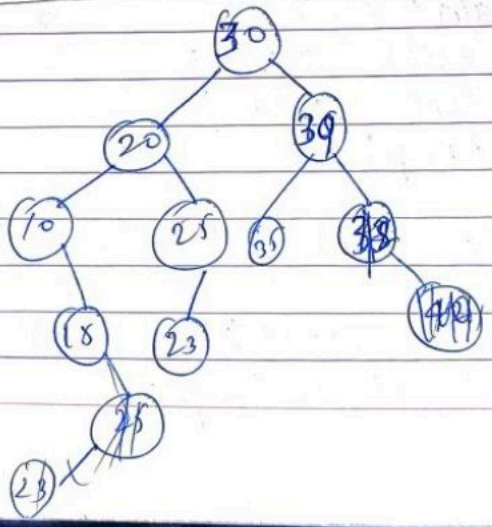
5, 10, 12, 20, 22, 25, 28, 30, 36, 38, 40, 48.

* No. of distinct B.S.T. possible with n distinct Key = $\frac{2nC_n}{n+1}$

BINARY SEARCH TREE TRAVERSAL -

Q Preorder of binary search tree is
30, 20, 10, 15, 25, 23, 39, 35, 42
find the Postorder of B.S.T.

10, 15, 20, 23, 25, 30, 35, 39, 42



OPERATION OF B.S.T. - INSERTION, DELETION & UPDATION -

```
#include<stdio.h>
struct node;
{
    int data;
    struct node *left, *right;
};
struct node * insert(struct node *new, int Key)
{
    if(new == NULL)
        return newnode(Key);
    if( new->data < Key)
        new->right = insert(new->right, Key)
    if(new->data > Key)
        new->left = insert(new->left, Key)
    return node;
}
struct node * newnode (int Key)
{
    struct node *temp;
    temp = (struct node *) malloc(sizeof(struct node));
    temp->data = Key;
    temp->right = NULL;
    temp->left = NULL;
    return temp;
}
struct node *search ( struct node *temp, int value)
{
    if( temp == NULL || temp->Key == value)
        return temp;
```

Insert, particular element search, traversal (inorder - preorder, postorder

```
if ( temp → Key > value )
    search (temp → left, value);
else
    search (temp → right, value);
}
```

HEIGHT OF A TREE

Longest Path from root to leaf node i.e., in that path the no. of edges.

AVL.

Balance factor of node = height of left subtree - height of Right subtree.

TIME COMPLEXITY OF B.S.T. –

The time complexity of B.S.T. for insertion, searching & deletion operation is O(h). where h is the height of B.S.T.

* Worst Case Analysis of B.S.T –

The worst case of B.S.T. is SKewed B.S.T. It happens when values are inserted either in increasing or decreasing order occurence. In this (SKewed B.S.T) case the height of B.S.T. becomes equivalent to n where n is no. of values inserted in B.S.T.. Hence, for searching, insertion & deletion

order will be O(n).

* BEST CASE ANALYSIS OF B.S.T. -

The Best case of Analysis of B.S.T. will be a Balanced B.S.T. where the height of Balanced B.S.T. will be $h = \log_2 n$. where n is the no. of nodes or value inserted. So, by the above analysis we find if we balanced the tree by insertion & deletion operation we can reduce the time complexity for further searching, insertion & deletion operation. for balancing their are many algorithm that are -

(i) AVL Tree
(ii) Red-Black Tree.
(iii) B-Tree.

AVL TREE -

AVL Tree are self balancing B.S.T. where the difference of height of left subtree & right subtree of any node is not greater than 1.

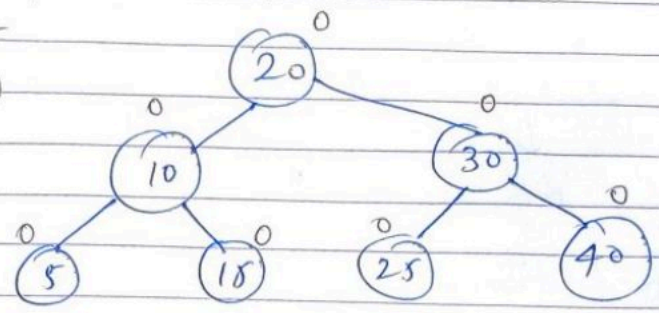
* Balanced Factor.

Balanced factor of are node in B.S.T. is equal to height of left subtree - height of right subtree.

So, AVL tree are those B.S.T. where Balance factor of each node will be either 0, -1 or 1.

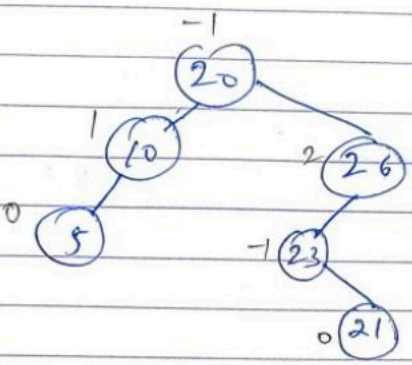
for example -

(i)



← AVL Tree

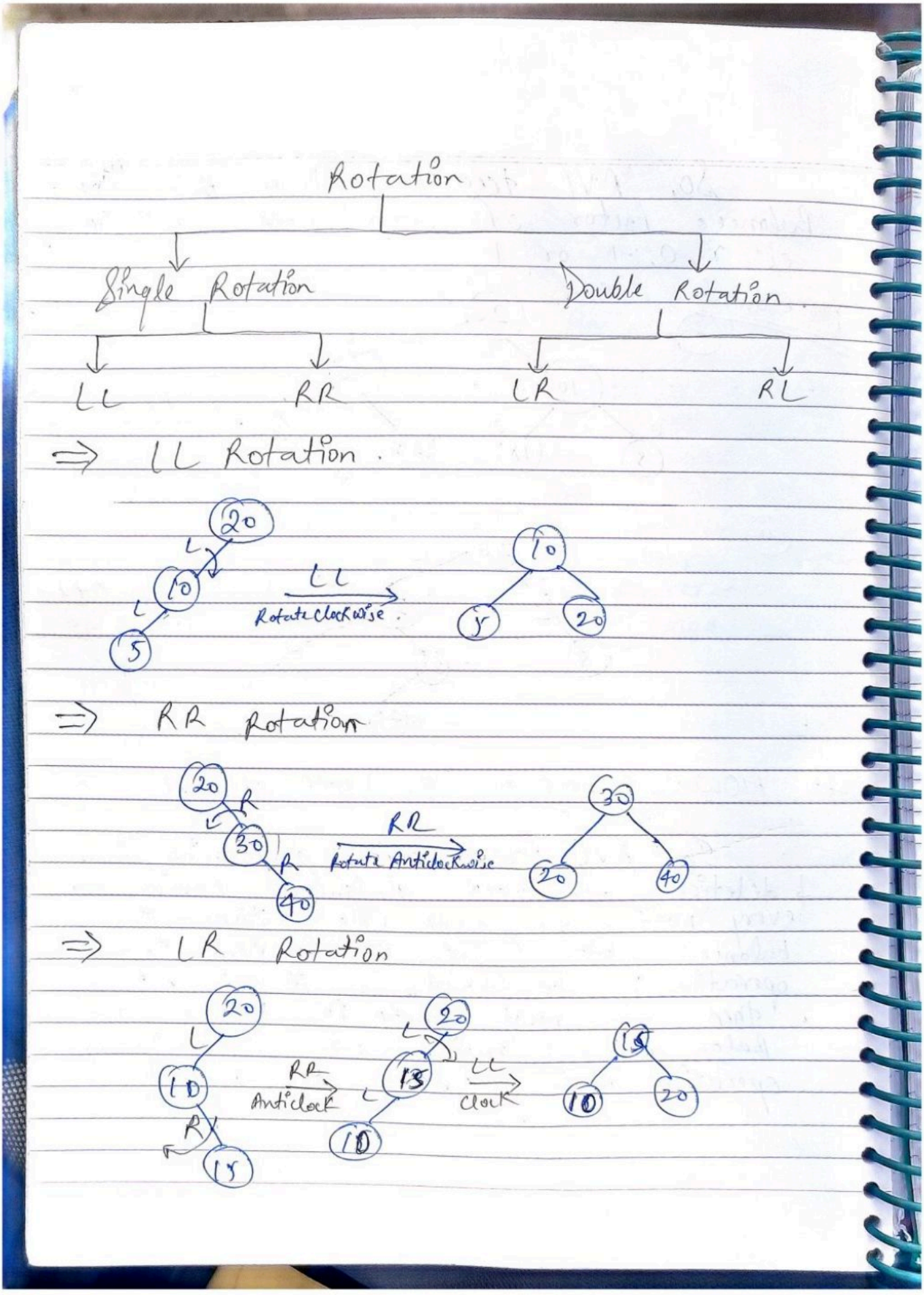
(ii)



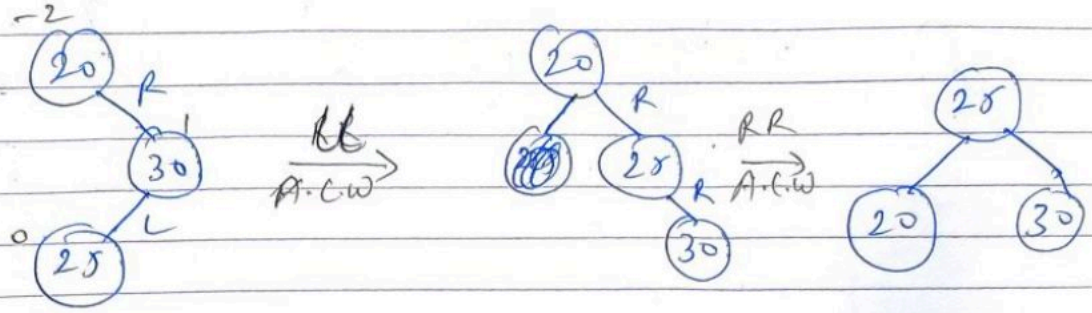
Not a AVL Tree.

HOW BALANCING IS DONE IN AVL TREE?

In AVL Tree after performing insertion & deletion, we check Balanced factor of every node. If every node satisfy the balanced factor (i.e., 0, -1, 1) then the operation is concluded, if it is not so then we must make it balance. ~~for~~ ~~Bala~~ for Balancing we use rotation operation which can be classified as.

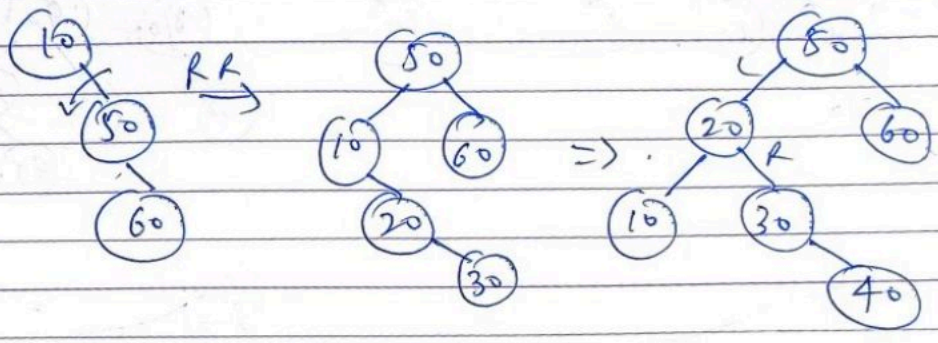


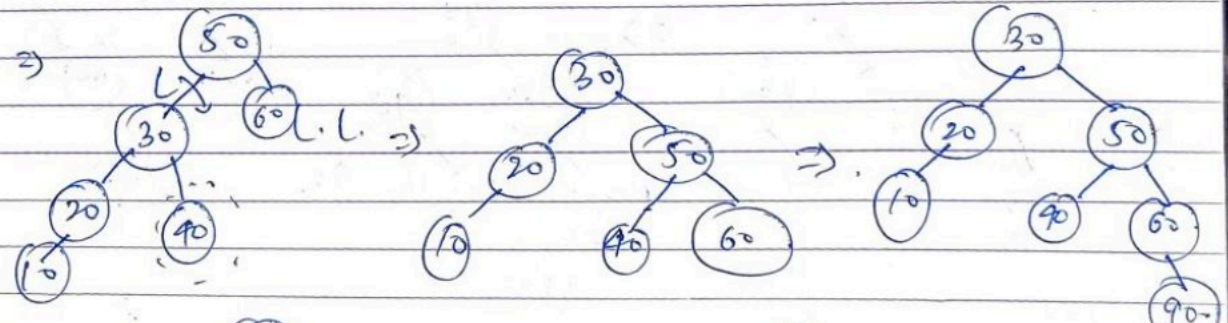
⇒ RL Rotation -

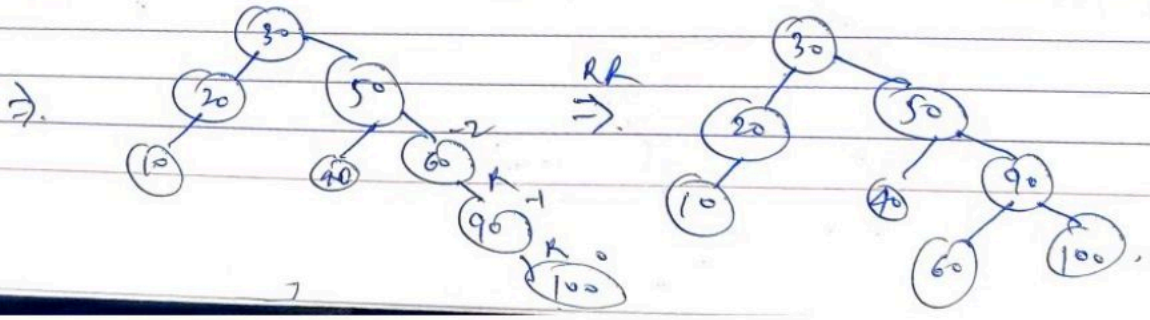


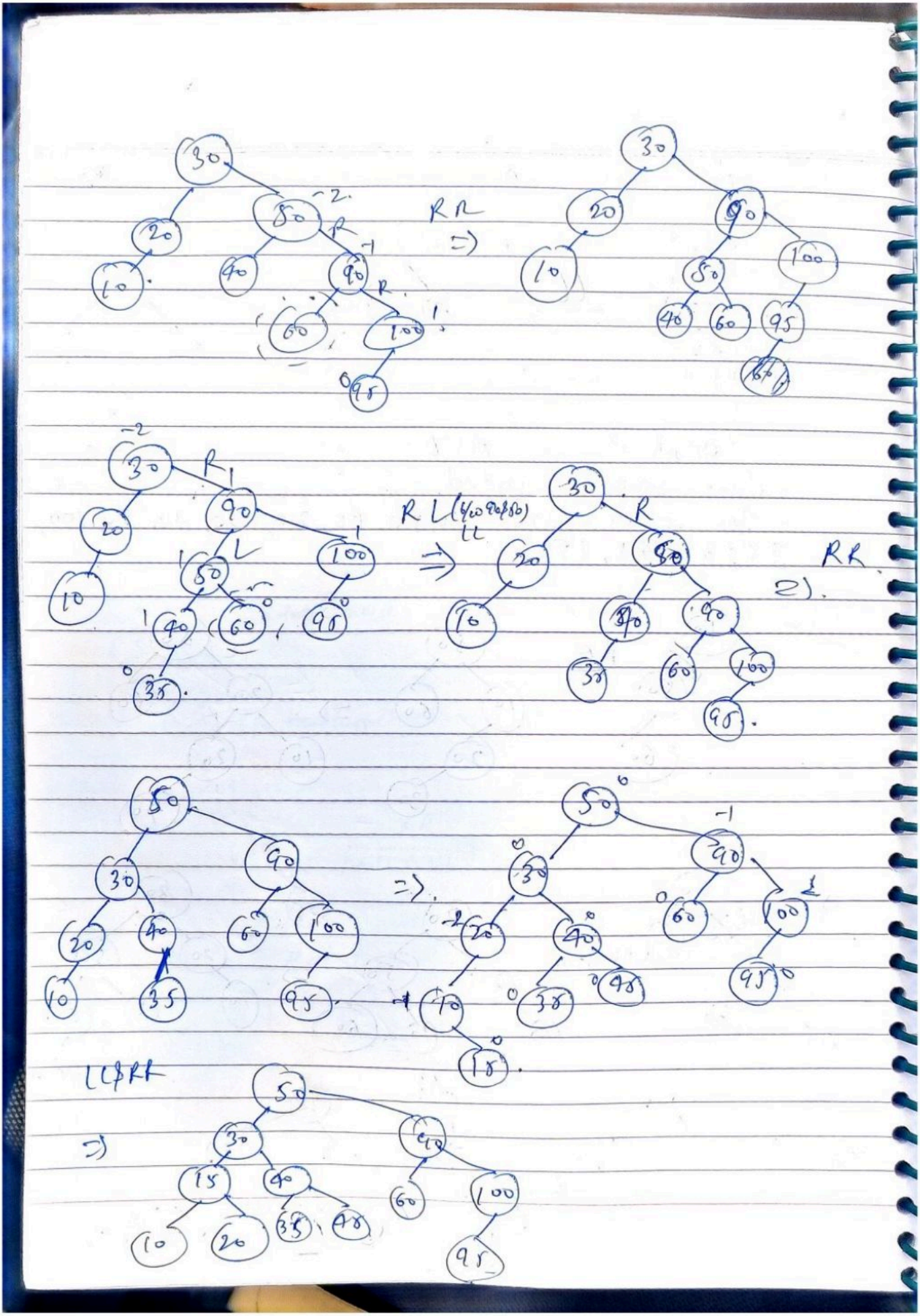
Ex- Construct a AVL Tree when the values are inserted in following sequence & The values are 10, 50, 60, 20, 30, 40, 90, 100, 95, 35, 45, 15.

Solution-



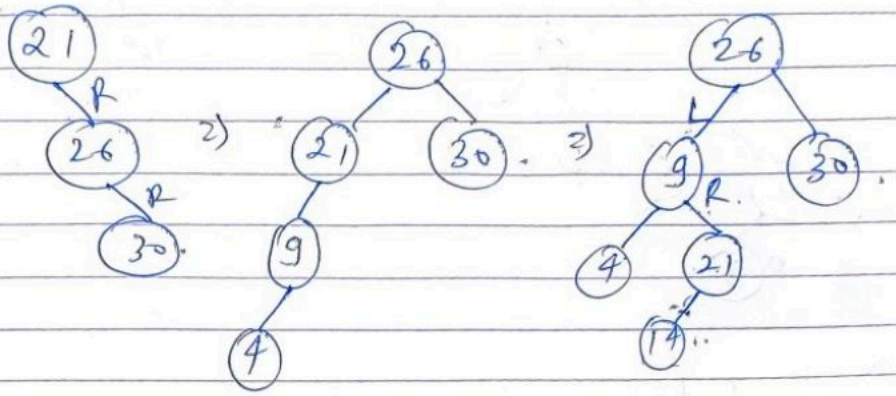




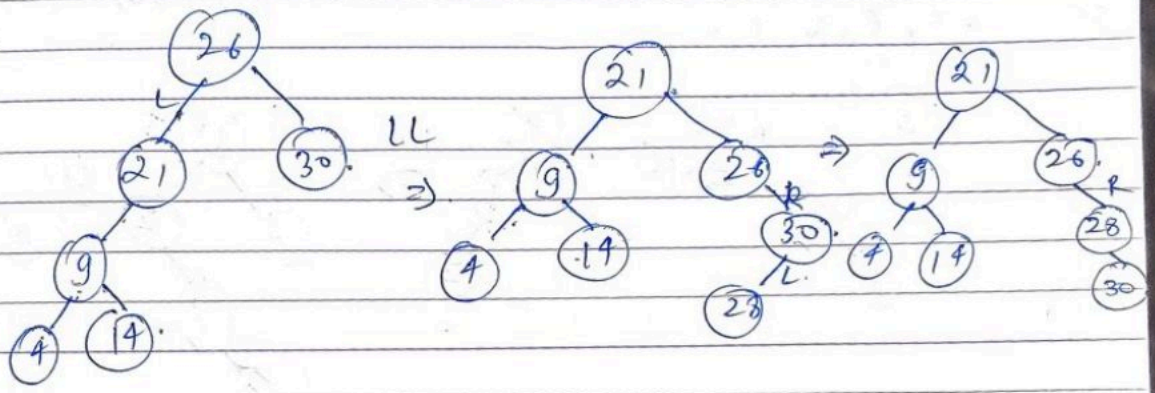


Q. AVL :→

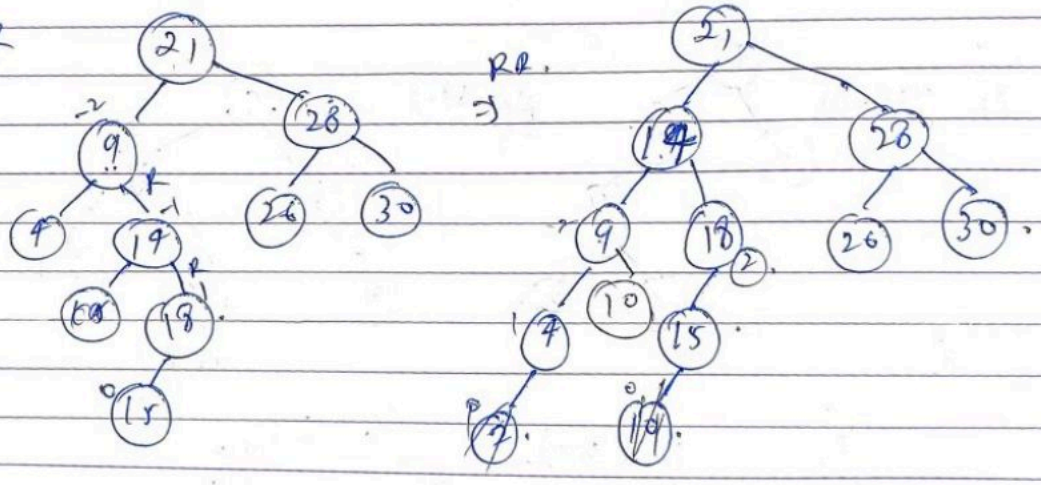
21, 26, 30, 9, 4, 14, 28, 18, 15, 10, 2, 3, 7

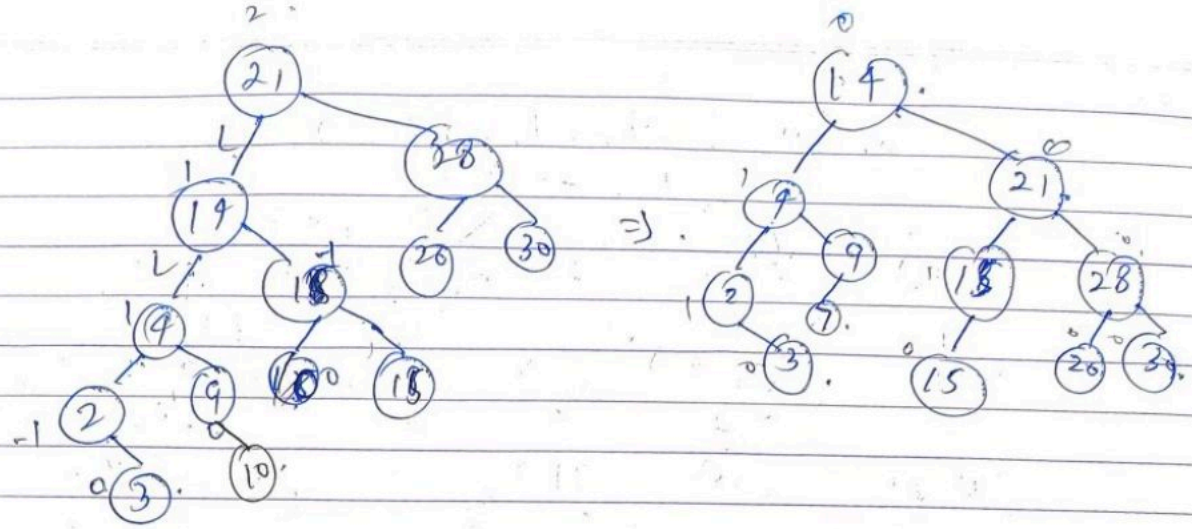


3)

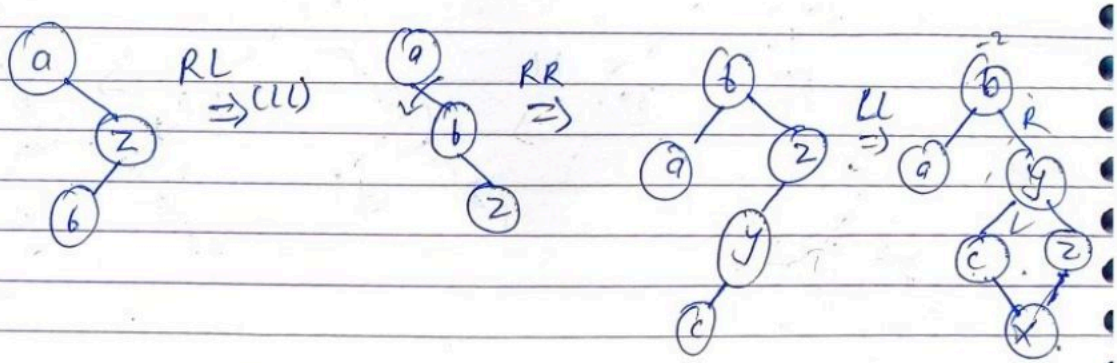


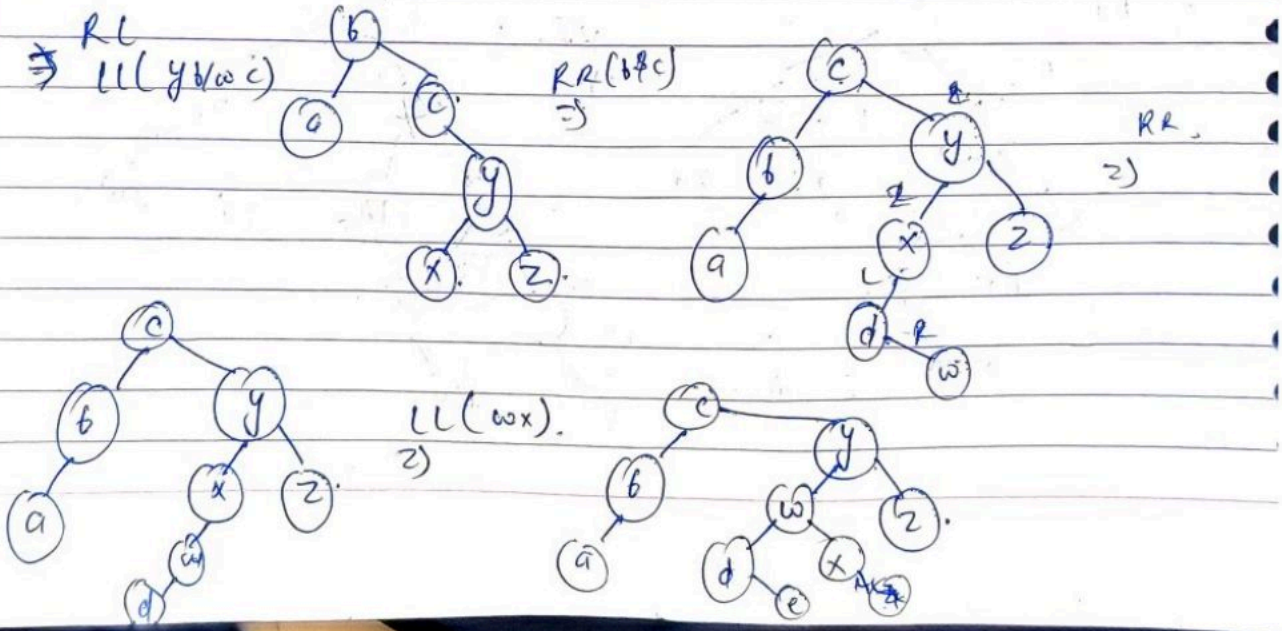
4) RR



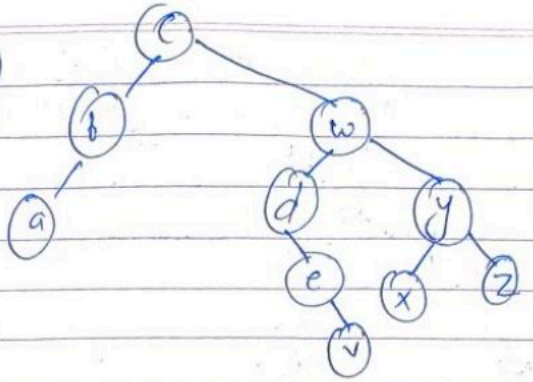


Q Find the AVL tree of a, z, b, y, c, x, d, w, e, v, f.

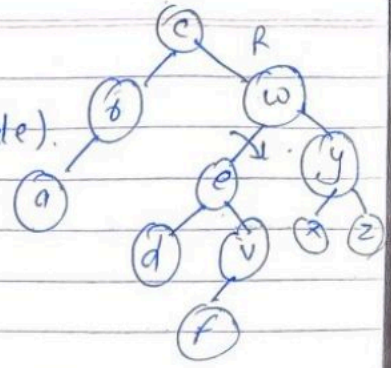




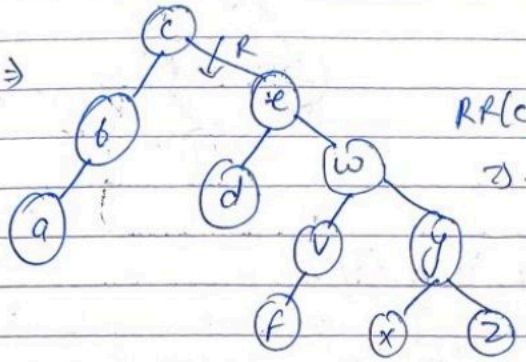
LL(y ω)
⇒



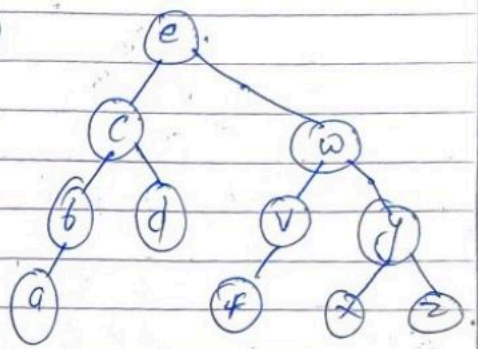
RR(ele).
⇒.



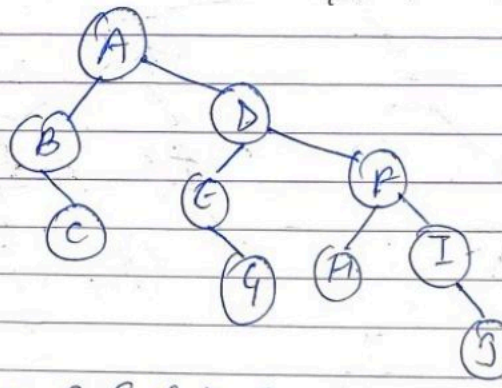
RL
LL(ωte) ⇒



RR(cte)
⇒.



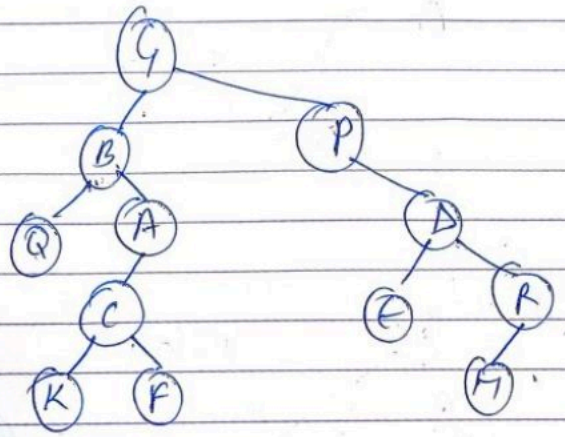
Q Inorder → B, C, A, E, G, D, H, F, I, J.
= Preorder → A, B, C, D, E, G, F, H, I, J.



Post - C, B, G, E, H, J, I, F, D, A.

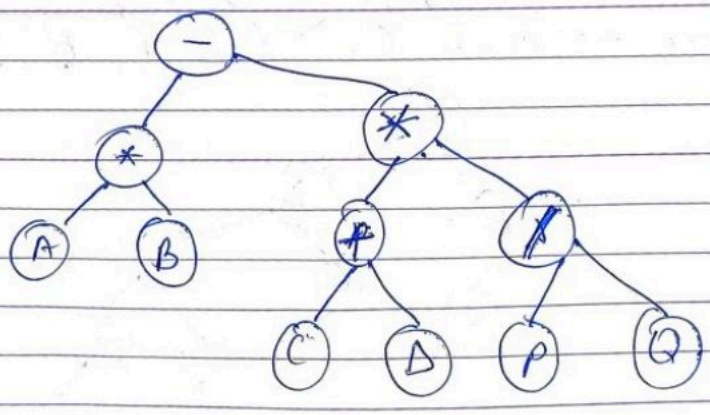
Q Inorder ⇒ Q, B, K, C, F, A, G, P, E, D, H, R.
Preorder ⇒ G, B, Q, A, C, K, F, P, D, E, R, H.
Postorder ⇒ ?

Solution:-



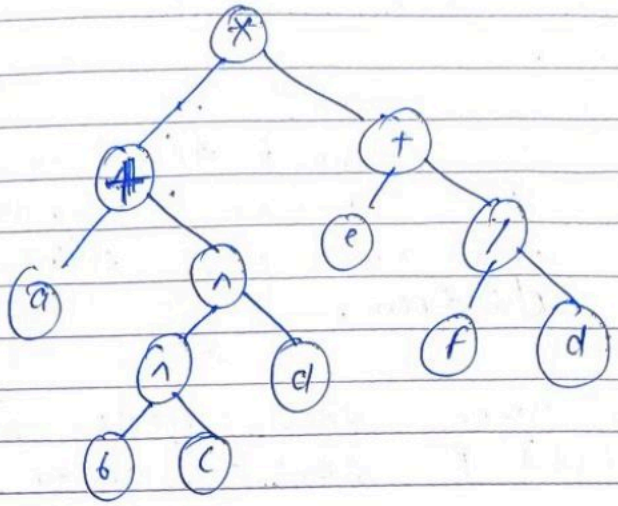
Post - Q, K, F, C, A, B, E, H, R, D, P, G.

Q Draw the binary tree of the following expression -
A * B - (C + D) * (P / Q).



Preorder = - * A B.

Q Convert the infix string (a+b^c^d)*(e+f/d). to reverse polish.



a, b, c, ^, d, ^, +, e, f, d, /, +, *.

Q 6+2^3+9/3-4*5.

Imp

B-TREE → Self Balance Tree.

(i) B-Tree is a balanced 'm'-way tree where m is known as order of B-Tree.

(ii) B-Tree is a generalization of binary search tree (B.S.T) in which a node can have more than one key elements & more than two children.

(iii) In B-Tree each node maintain data (keys value) in sorted order.

(iv) All leaf node should be at same level

(v) B-Tree of order m has following property-

(a) Every node has maximum 'm' children.
(b) Minimum children: leaf → 0.
Root → 2.
Internal Node → $\left\lceil \frac{m}{2} \right\rceil$

ex- $\left\lceil \frac{5}{2} \right\rceil = \lceil 2.5 \rceil = 3$.

(c) Every node has maximum (m-1) keys.
(d) Minimum keys: root → 1.
all other node → $\left\lceil \frac{m}{2} \right\rceil - 1$.

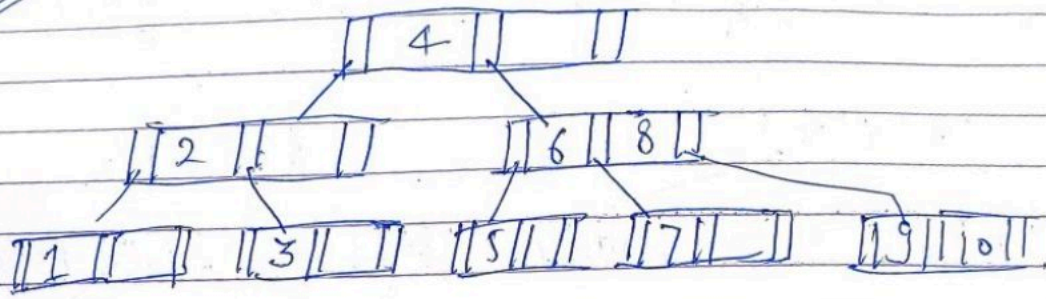
(vi) Every node would be inserted on leaf.

NOTE - B-Tree will grow upward

Q Construct a B-Tree of order 3. by inserting values 1 to 10.

Solution -

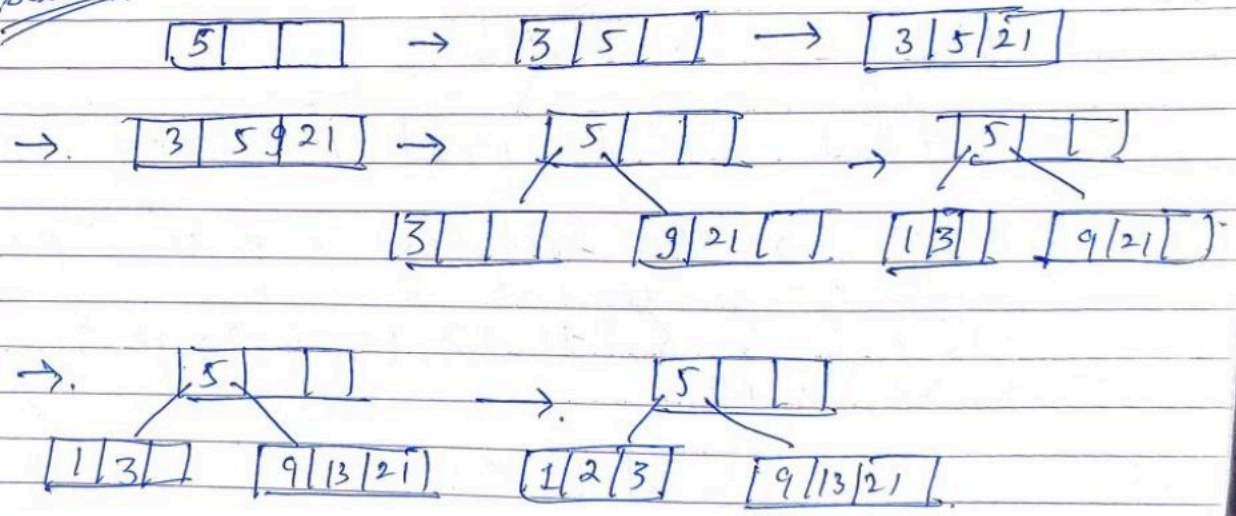
$m = 3$, $K = 2$.

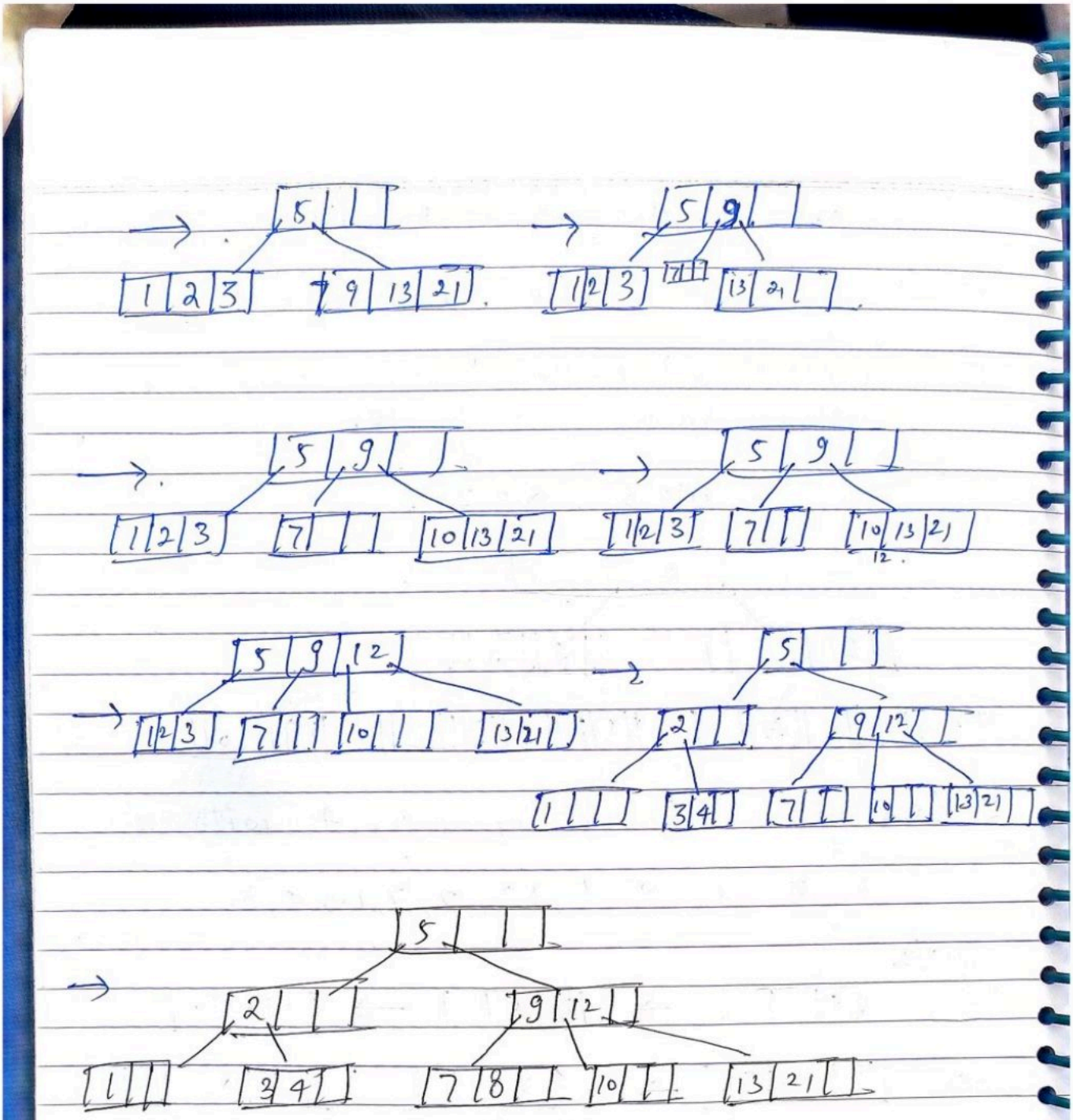


Q Construct a B-Tree of order 4 with following element -

5, 3, 21, 9, 1, 13, 2, 7, 10, 4, 8.

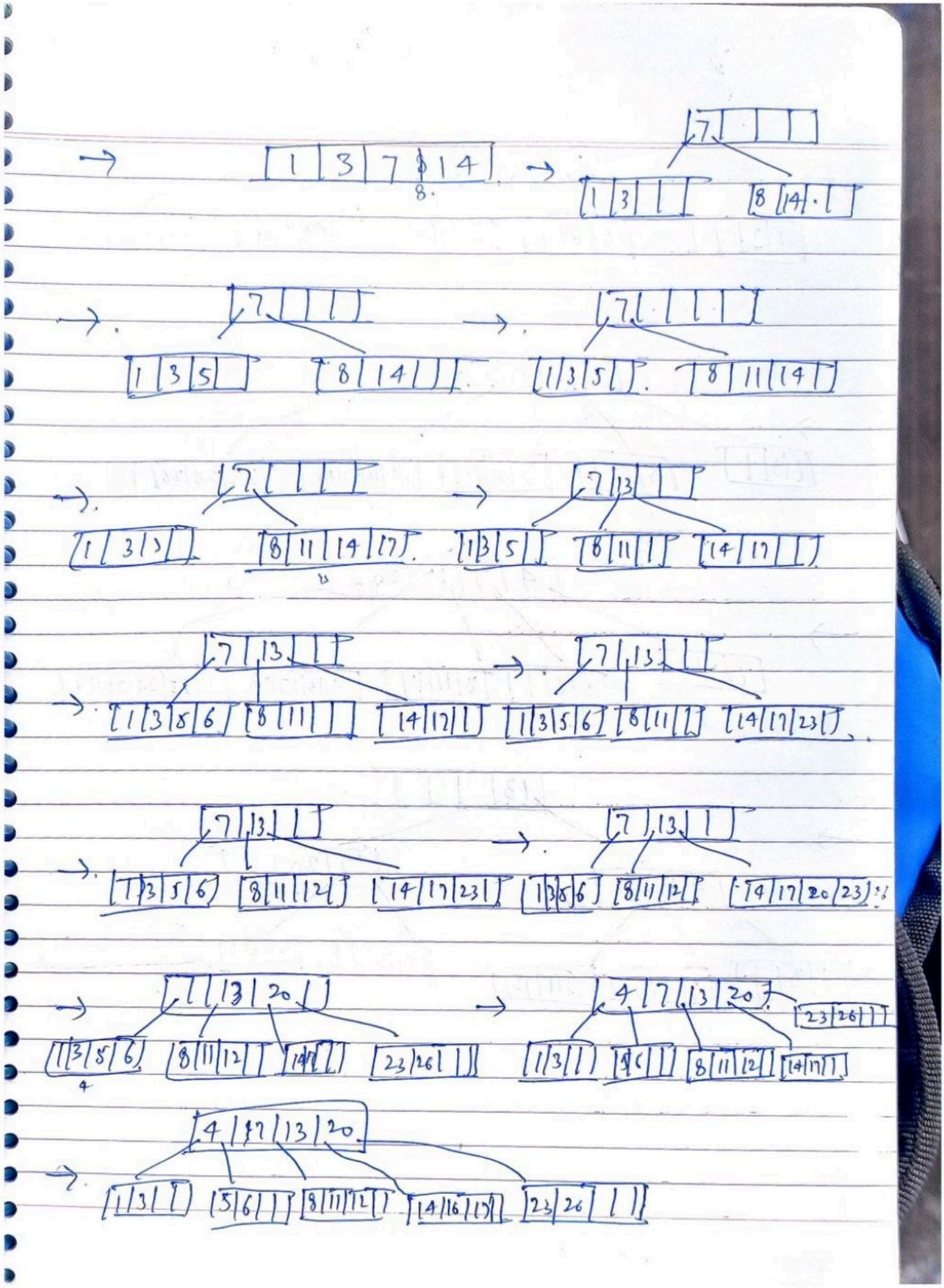
Solution -

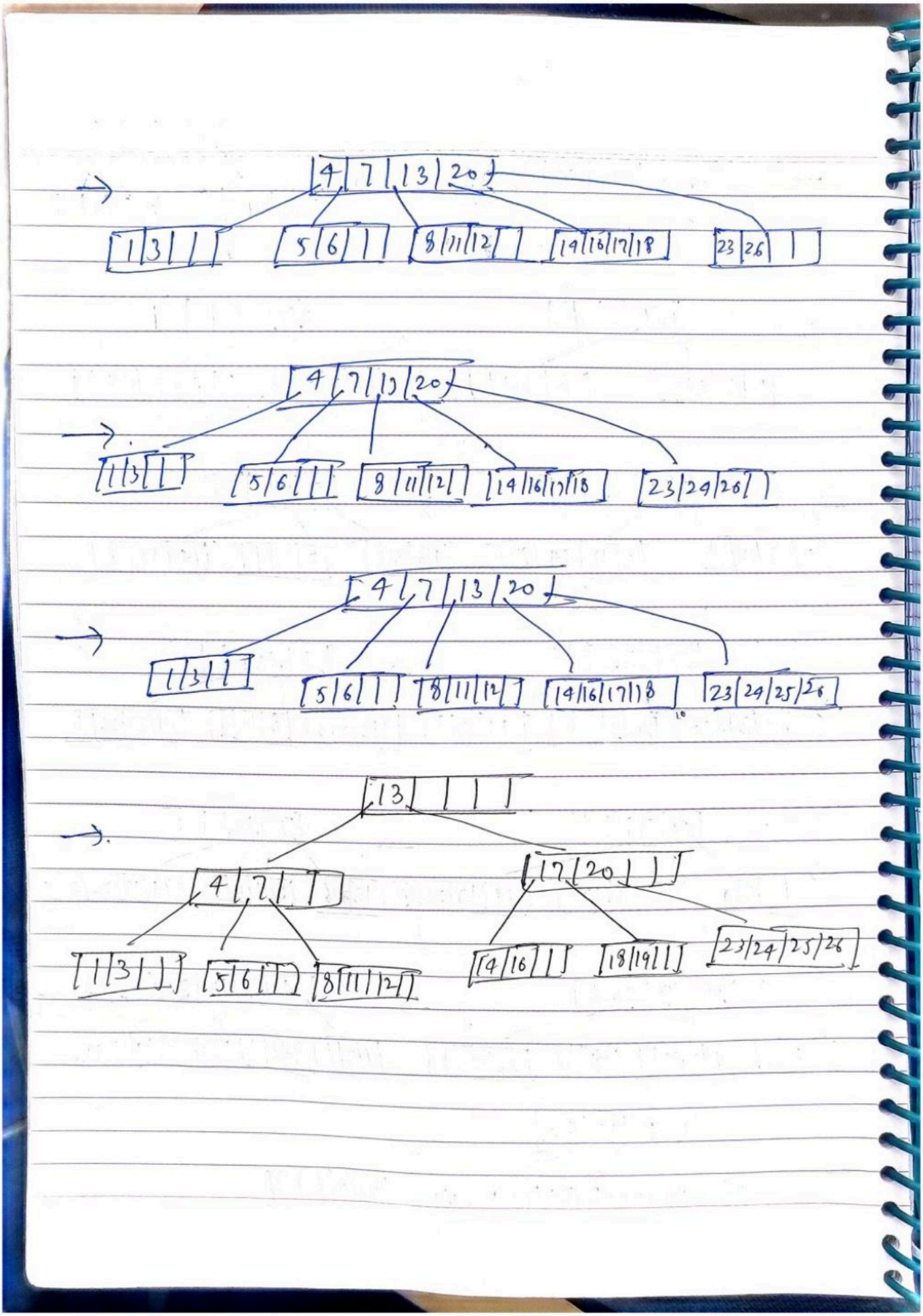




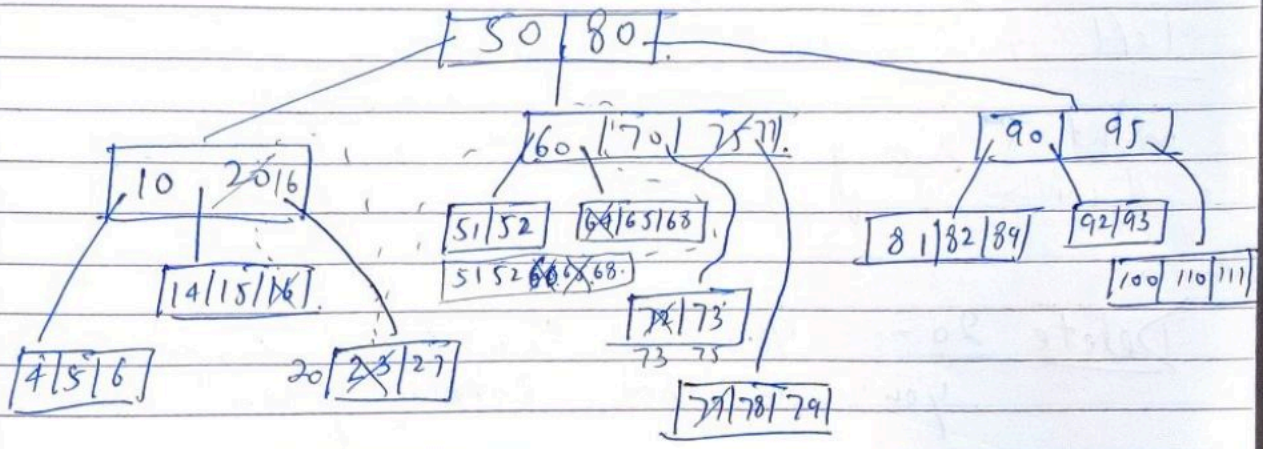
Q Construct a B- Tree of order 5 inserting the following sequence elements.
3, 14, 7, 1, 8, 5, 11, 17, 13, 6, 23, 12, 20, 26, 4, 16, 18, 24, 25, 19.

⇒





Q Deletion Operation on B~~inary~~ Tree.



B. Tree of order 5.

Min. Child = $\lceil \frac{m}{2} \rceil = 3$.

Max. Child = $m = 5$.

Min. Key = $\frac{m}{2} - 1 = 2$.

Max. Key = $m - 1 = 4$.

The value can found on

1. Root.
2. Intermediate.
3. Leaf.

More than min. Keys. | Exactly min Key.

Left Sibling borrow | Right Sibling borrow.

Left & Right both having min Key.

Left Borrow –

Left Sibling max. value will move to its parent & parent Key will move to target node & then delete the target node.

Right Borrow –

Right Sibling min. value will move to its parent & parent Key will move to target node in sorted order &.

delete the target node.

Left & Right Borrow :- Merge the target node with its right sibling or left sibling along with its parent.

Delete 20 - This is the case where you can not borrow parent node.