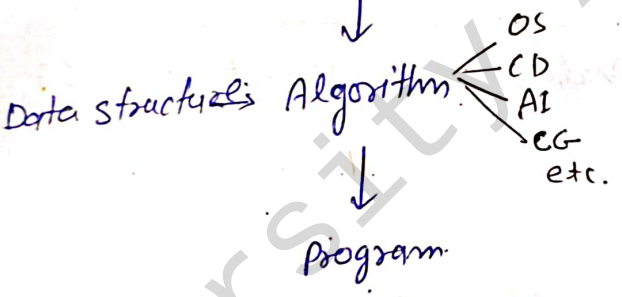
Data Structure

Unit-I

Introduction:

Data structure is logical and mathmatical model of storing and organizing data in a particular way in a computer so that it can be required for designing and implementation of algorithm and program development.
eg. Array, linked list stack, queues etc.



Basic Terminology: Elementary Data Organization:

Data: Data are simply values or set of values. eg.

Student Name: Virat age: 34.

Data items: A data item refers to a single unit of values. Data items are divided into subitems are called groupitem

eg. student-Name divided into First Name, middle Name, last name but age would be treated as single item.

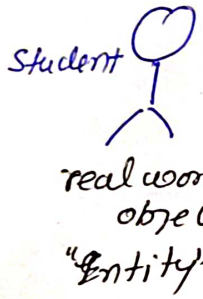
Record: Record is collection of various data item e.g. Record of student; Name, address, course, age, marks can be grouped together to form Record.

File: A File is collection of various record of one type of entity. An entity has certain attributes.
(Real world object)

Field: Field is a single elementary unit of information representing the attribute of an entity.

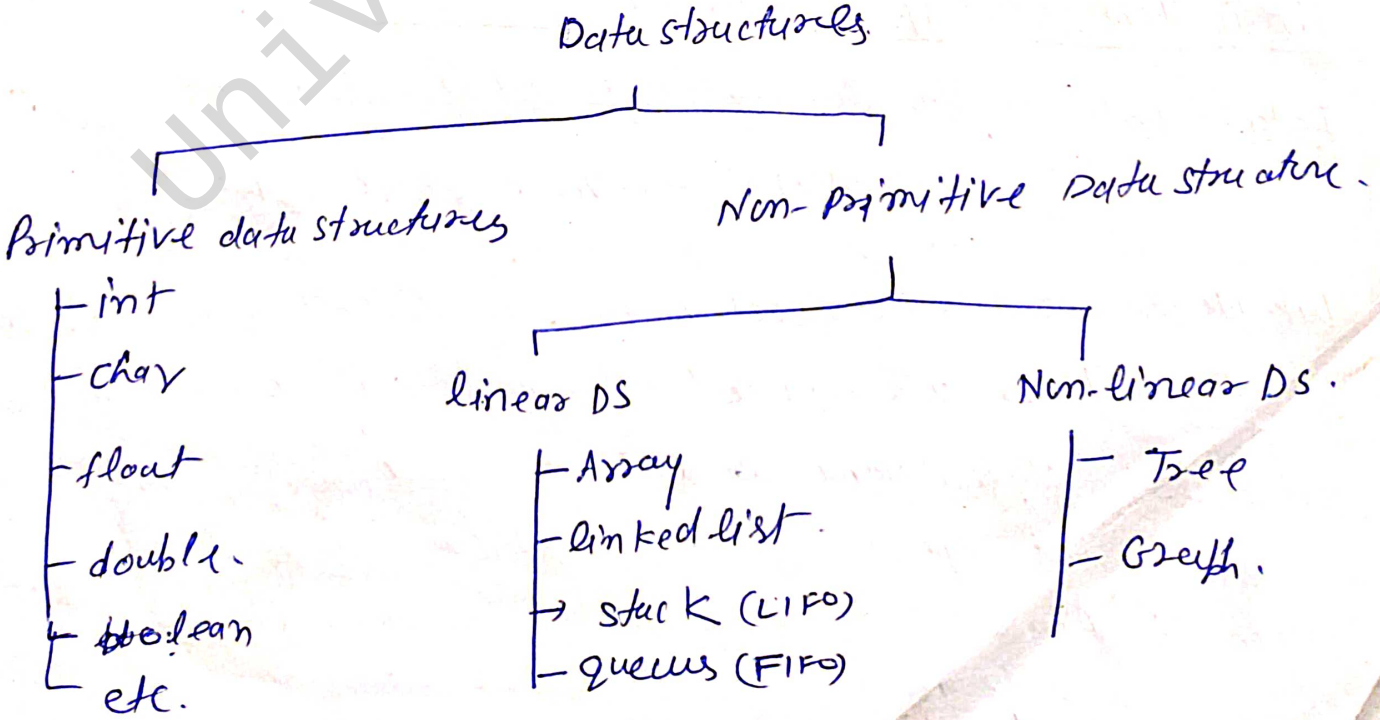
Information: the term information is sometime used for data with given attributes. in other words meeningfull data is called Information.

Example:



Attributes:	Name	age	Roll No.
Value:	Virat	34	310212536

Classification of Data structures.

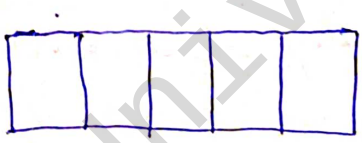


Primitive Data Structures: Primitive data structure are the basic data structure that directly operate upon the machine instruction. these are predefined operations and properties.

Non-Primitive DS: Non-Primitive data structures are more complicated and derived from primitive data structures.

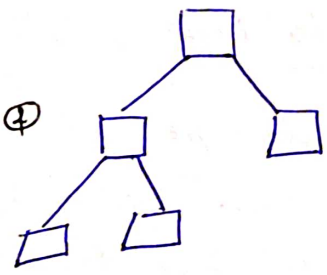
Linear Data structure: A data structure is called linear if all of its elements are arranged in the linear order. where each element has the successors and predecessors except First and Last element.

Non-linear Data structure: This data structure does not form a sequence. data elements are hierarchically connected.

① 

Linear

2. Single level is involved.
3. data element can be traversed in single run.
4. Array, linked list, stack, queue
5. used in system development

④ 

Non-linear

2. multilevel is involved.
3. cant be run single traversal.
4. Graph tree.
5. used in AI and DIP.

Data structure Operations

The following four operation used in data structures.

1- Traversing: Accessing each record exactly once. Sometime called visiting the record.

2. Searching: Finding the location of the record with a given key.

3. Inserting: Adding a new record to the structure.

4. Deleting: Removing a record from structure.

Sometime two more operations may used in a given situation.

1. Sorting: Arranging the record in some logical order.

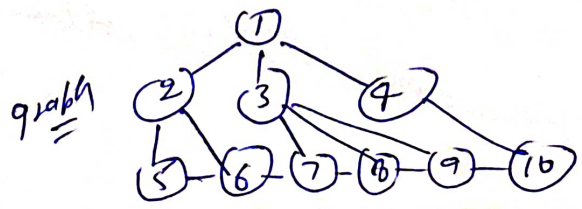
2. Merging: Combining the records in two different sorted file into single sorted file.

Abstract Data Type: (ADT)

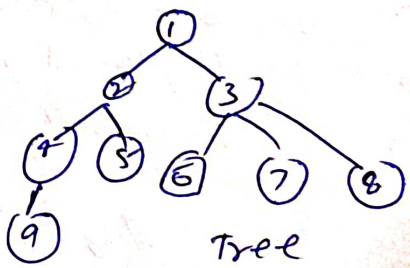
Abstract Data type refers to a set of value and associated operation or function. With ADT we know what a specific data type can do but how it is accually does it is hidden.

Example

list

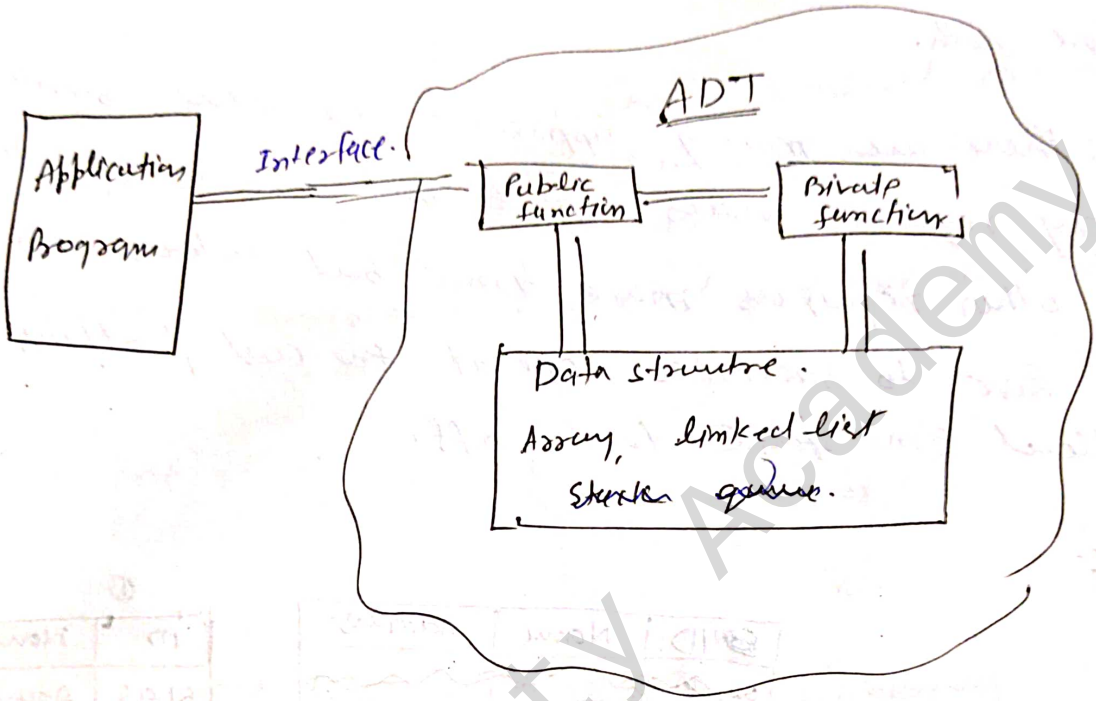


graph



Tree

There are two different part within ADT, one is functions or operations and another is data structure.



Data are entered, accessed, modified and deleted through the external application programming interface.

The datatype is basically a type of data that can used in different computer program. e.g. int, float, char etc.

The abstract data type is special kind of datatype where behaviour is difined by a set of value and set of operations. the keywood "Abstract" is used as we can use these datatype, we can perform different operation

Example Stack<
- push()
pop()

array
sizeof()

queue->
insert()
delete()

List->
size()
insert()

Time Space Trade Off:

The Best algorithm is one that requires less space and less time to execution. but it is not always possible to achive both.

There are many approach to solve same problem. one may require mores space but takes less time. while other requires more time but takes less space. thus we have to sacrifice one at the cost of others. that is called time space trade off.

Example:-

ID	Name	Address
3961	Leena	Delhi
3485	Brijesh	Ghaziabad
6398	Sanjay	Lucknow
5555	Divya	chandigarh
6198	Ajay	Mumbai

①

ID	Name	Adress
6198	Ajay	mumbai
3485	Brijesh	Ghaziabad
5555	Divya	chandigarh
3961	Leena	Delhi
6398	Sanjay	Lucknow

sorted by Name

Name	Pointer
Ajay	•
Brijesh	•
Divya	•
leena	•
Sanjay	•

②

ID	Name	address
3485	Brijesh	Ghaziabad
3961	leena	Delhi
5555	Divya	chandigarh
6198	Ajay	mumbai
6398	Sanjay	Lucknow

Sorted by ID

Array Data Structure:

An array is a collection of element of similar type data eg. int, float. This collection is finite and stored at adjacent memory location.

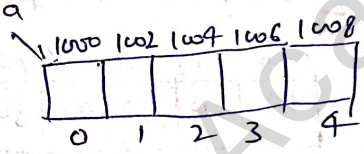
Element: item stored in array is called element

Index: Each location of an element in an array has a numerical index usualy start from 0th to n-1.

Address: numerical value of first byte at which item are stored.

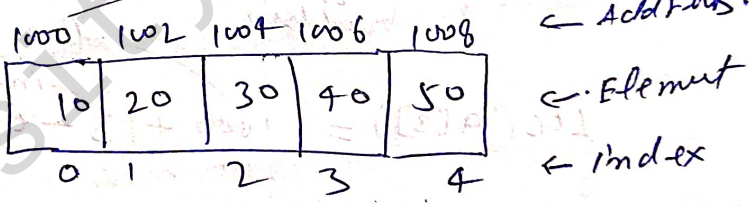
eg.

int a[5];



1000 – 1009 → 10 byte memory is required

int a[5] = {10, 20, 30, 40, 50}



a[0] = 10
a[1] = 20
a[2] = 30
a[3] = 40
a[5] = 50

Base address.
a = 1000

		address	value
a[0] =	a + 0×2 =	1000	= 10
a[1] =	a + 1×2 =	1002	= 20
a[2] =	a + 2×2 =	1004	= 30
a[3] =	a + 3×2 =	1006	= 40
a[4] =	a + 4×2 =	1008	= 50

an integer value take 2 byte in memory.

a[i] = a + i × size in word

Types of Array:

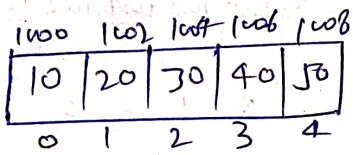
1- One-dimensional Array (1-D)
2. two-dimensional Array (2-D)
3. Three-Dimensional Array (3-D)
4. n-Dimensional Array (n-D)

1. One-Dimensional Array:

An array which has only one subscript is known as 1-D Array.

eg: int a[5];

in a[5] = {10, 20, 30, 40, 50}



How to find location of ..
or address

Base address.

$LOC('A[3]) = 1000 + (3 - 0) \times 2$

index — starting position or index

size of variable.

$= 1000 + 3 \times 2 = 1006$

Example. Consider the array A[1.....100], base address = 1000, and size of element = 4 byte.

find address of A[50].

$LOC(A[50]) = 1000 + (50 - 1) \times 4$

$= 1000 + 49 \times 4 = 1196.$

Example 2 Consider A[-50, ---- +50], base address 999 size of every element 10 byte. Find Address or location of A[49].

$$LOC(A[49]) = 999 + (49 - (-50)) * 10$$
$$= 999 + (49 + 50) * 10$$
$$= 999 + 990$$
$$= 1989.$$

Example 3: Consider the linear Array A(5:50) whose base address is 300 and number of words per memory cell is 4. find the address of A[15].

Ans: 340

---

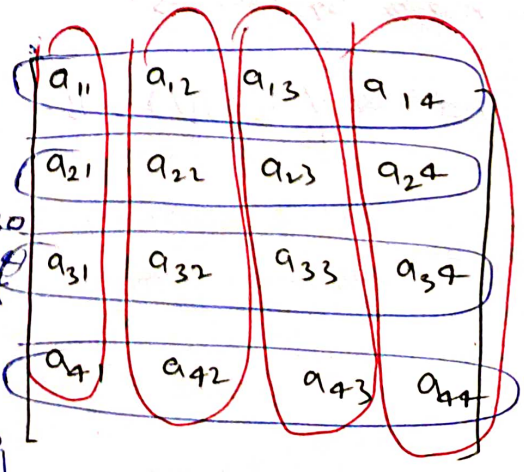
2. Two Dimensional Array (2D) →

An Array which has two subscript is known as 2-D Array.

2-D is also known as matrix.

when 2-D array gets stored in computer memory it can not be 2D way it can store only 1-D way as row major Representation or column major Representation.

eg. int a[4][4]; (row / col.)



row major:

row 1				row 2				row 3				row 4			
1000 1002	04	06		08	10	12	14	16	18	20	22	24	26	28	30
a_{11}	a_{12}	a_{13}	a_{14}	a_{21}	a_{22}	a_{23}	a_{24}	a_{31}	a_{32}	a_{33}	a_{34}	a_{41}	a_{42}	a_{43}	a_{44}
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Column major

Col 1				Col 2				Col 3				Col 4			
1000	02	04	06	08											30
a_{11}	a_{21}	a_{31}	a_{41}	a_{12}	a_{22}	a_{32}	a_{42}	a_{13}	a_{23}	a_{33}	a_{43}	a_{14}	a_{24}	a_{34}	a_{44}

Row major Representation

In the Given 2-D array A[m][n] the address of an element A[i][j] is calculated in row major system as follows

$$\text{Address of } A[i][j] = b + [(i - l_r) * n + (j - l_c)] * w$$

given matrix.

Exam: A[4][5], base address 1020, adr A[3][4] = ?
element size = 2 byte.

b - Base address
i → row of element to be found
j - col of element to be found
w = size of element in byte
l_r = lower limit of Row (if not given assume 0)
l_c = lower limit of Col (if not given assume 0)
m → No of row in matrix
n → no of col in matrix.

$$\text{Address of } A[3][4] = 1020 + [(3-0) * 5 + (4-0)] * 2$$
$$= 1020 + 38 = 1058$$

Column Major Representation

$$\text{Address of } A[i][j] = b + [(i - l_r) + (j - l_c) * m] * w$$

$$A[3][4] = 1020 + [3 + 4 * 4] * 2$$
$$= 1020 + 38 = 1058.$$

Example: a[1....4, 1....4], b = 1000, size = 2, A[4][3] = ?

no of row = 4 - 1 + 1 = 4

no of row = last row no - First row no + 1

① Row major:

$$A[4][3] = 1000 + [(4-1) * 4 + (3-1)] * 2$$
$$= 1000 + [3 * 4 + 2] * 2$$
$$= 1000 + 28 = 1028$$

② Column Major:

$$A[3][4] = 1000 + [(3-1) + (4-1) * 4] * 2 = 1000 + [2 + 12] * 2$$
$$= 1000 + (14 * 2) = 1028$$

2 | 1028

Example An array A[-15 ..... 10, 15 ..... 40] require one byte of storage its beginning location 1500, find the location of A[15][20].

Sol:

No. of rows = last row no. − first row no + 1
= 10 − (−15) + 1 = 10 + 15 + 1 = 26.

no. of col = last col no − first col no + 1
= 40 − 15 + 1 = 26.

(1) Row major wise calculation:

Address A[i][j] = b + [(i − l_r) * n + (j − l_c)] * w

A[15][20] = 1500 + [15 − (−15) * 26 + (20 − 15)] * 1
= 1500 + [30 * 26 + 5] * 1
= 1500 + 785 = 2285 Ans

(2) Column major calculation.

Address A[i][j] = b + [(i − l_r) + (j − l_c) * m] * w

A[15][20] = 1500 + [(15 − (−15)) + (20 − 5) * 26] * 1
= 1500 + [30 + 5 * 26]
= 1500 + 160 = 1660 Ans

3-D Array

An array which as three subscript is known as 3-D array.

in c 3D Array is declared as.

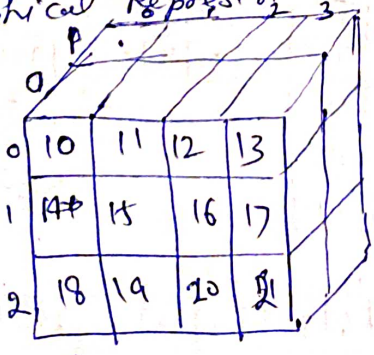
Datatype arrayName [size1] [size2] [size3]

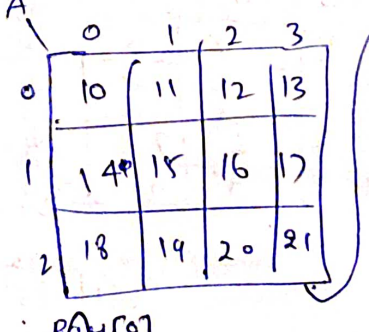
int A[2][3][4]

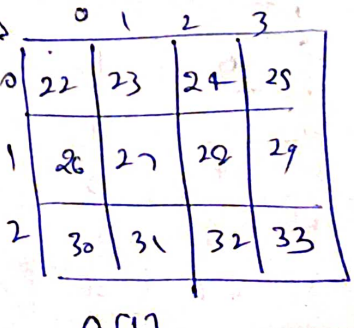
1. The memory allocated to variable A is of data type int
2. Total capacity of 3D array are 2×3×4 = 24 element
3. data is being presented as 2 array with 3 row and 4 column each.

r/c				
A[0]	A[0][0]	A[0][1]	A[0][2]	A[0][3]
	A[1][0]	A[1][1]	A[1][2]	A[1][3]
	A[2][0]	A[2][1]	A[2][2]	A[2][3]
A[1]	A[0][0]	A[0][1]	A[0][2]	A[0][3]
	A[1][0]	A[1][1]	—	—
	A[2][0]	—	—	—

Graphical Representation of array A[2][3][4]







In 3-D Array also address is calculated through two method

1- row-major order 2. Column-major method.

Consider a three dimensional array $A[a:b, c:d, e:f]$

$L_1 \quad L_2 \quad L_3$

① find the length of all dimensions.

L_i = upperbound − lowerbound + 1

② Find Effective Index

suppose we have to find Address of $A[K_1, K_2, K_3]$

$E_i = K_i$ − lowerbound.

Row major:

$$LOC[A[K_1, K_2, K_3]] = B + w[(E_1 L_2 + E_2) L_3 + E_3]$$

Column major:

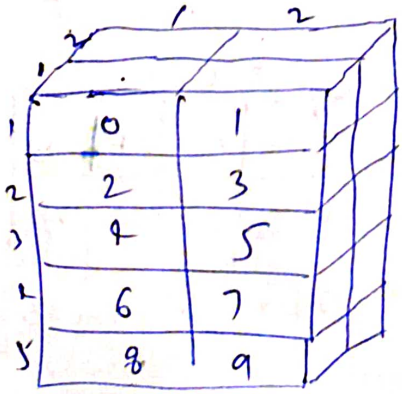
$$LOC(A[K_1, K_2, K_3]) = B + w[(E_3 L_2 + E_2) L_1 + E_1)]$$

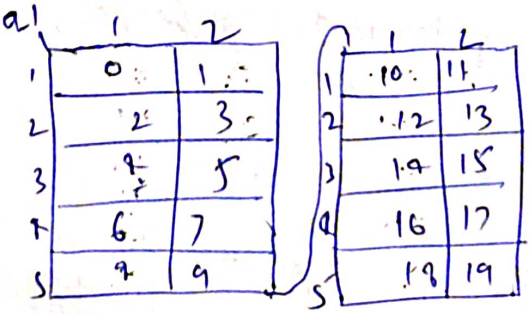
Example: if you have 3D array ? A 3D array is essentially an array of 2D array

$A[2][5][2] = A\_1[5][2], A\_2[5][2]$

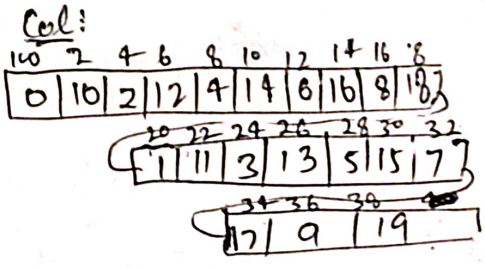
$A\_1[5][2] = \{\{0,1\},\{2,3\},\{4,5\},\{6,7\},\{8,9\}\};$

$A\_2[5][2] = \{\{10,11\},\{12,13\},\{14,15\},\{16,17\},\{18,19\}\};$

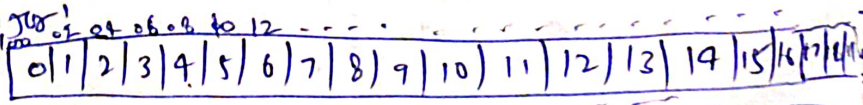




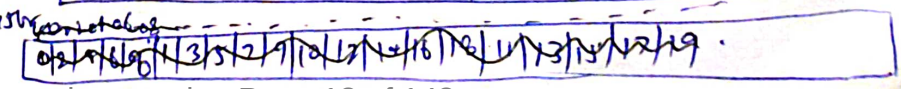
Col:



Row major:



Col major:



Multidimensional Array

An n-dimensional $m_1 \times m_2 \times m_3 \cdots m_n$ array A is collection of $m_1 ; m_2 \; m_3 \cdots m_n$ data element and list of n integer specified as $K_1, K_2 \; K_3 \cdots K_n$ called subscript.

eg: ~~int A[K1][K2]~~ demotes.
the element A with subscript $A[K_1, K_2, K_3 \cdots K_n]$

The programming language will store array in

1. Row major order
2. Column major order.

Let C be an n-D array, the index set for each dimension of C consists of consecutive integer from lower to upper bound. the length L_i of dimension i of C calculated as

$$L_i = \text{upper bound} - \text{lower Bound} + 1$$

For given subscript K_i the effective index E_i calculated as

$$E_i = K_i - \text{lower bound}$$

then address of $LOC(C[K_1, K_2 \cdots K_N])$ of an arbitrary element of C can be calculated by

Col. wise:

$$\text{base address} + w[(((E_N L_{N-1} + E_{N-1}) L_{N-2} + \cdots + E_3) L_2 + E_2) L_1 + E_1]$$

row wise.

$$b + w[(((E_1 L_2 + E_2) L_3 + E_3) L_4 + \cdots + E_{N-1}) L_N + E_N]$$

Q:- Suppose three dimmed array A[2:8, -4:1, 6:10). base=200.

$w = 4$, A[5, -1, 8]
K_1 K_2 K_3

① Find length of all dimmat

$L_1 = 8 - 2 + 1 = 7$

$L_2 = 1 + 4 + 1 = 6$

$L_3 = 10 - 6 + 1 = 5$

total No of elemt = $7 \times 6 \times 5 = 210$.

② Now Find effective index:

$E_i = K_i - \text{lower bound}$

$E_1 = 5 - 2 = 3$

$E_2 = -1 - (-4) = 3$

$E_3 = 8 - 6 = 2$

row wise

$LOC(A[5, -1, 8]) = B + w[(E_1 L_2 + E_2) L_3 + E_3)$

$= 200 + 4 \times [(3 \times 6 + 3) * 5 + 2)]$

$= 200 + 4 + (21 \times 5) + 2)$

$= 200 + 4 (107)$

$= 628$ Ans

Column

$LOC[A[5, -1, 8]] = B + w[(E_3 L_2 + E_2) L_1 + E_1]$

$= B + w[(E_3 L_2 + E_2) L_1 + E_1)]$

$= 200 + 4[(2 \times 6 + 3) 7 + 3$

$= 200 + 4[105 + 3]$

$= 200 + 4(108) = 632$ Ans

Applications of Array

1. Array are used to store list value : 1-D Array used to store list of value.

2. Array are used to Perform Matrix operation. : 2-D Array

3. Array are used to implemut search algorithm.
   1- linear search.
   2- binary search.

4. Array are used to implimut sorting algorithm.
   1- Bubble sort
   2- Insertion sort
   3. Selection sort
   4. Quick Sort
   5. merge sort etc.

5. Array are used to implemut Data structure.
   1- stack
   2- Queue

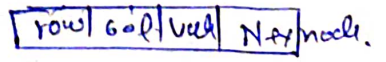
6. Array are used to represent sparse matrices.

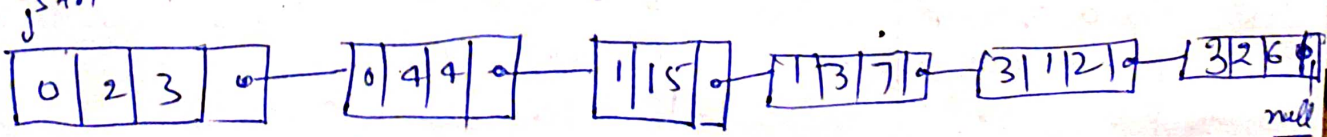
Linked list Representation → in linked list each node has four field as:

Row - index of row of the non zero elmt located
Column - index of column " " "
value → value of non-zero ~~ele~~ element
Next node → address of next ~~element~~ node.

eg:

0	0	3	0
0	0	5	7
0	0	0	0
0	2	6	0





Sparse Matrix

In computer programming a matrix can be defined with a 2D array any matrix can have m row and n column. there may be situation in which a matrix contain more number of zero that Non-zero value, such matrix is known as sparse matrix.

Example: if a matrix of size 100×100 and contain Non-zero elemt only 10. so Ten (10) space are filled with non-zero value.

total size occupied $100 \times 100 \times 2$ byte $= 20000$ bytes.

to access Ten (10) elemt we have to traverse 10000 location

Advantage of sparse Matrix:

1- storage
2- Computing time.

Sparse Matrix Representation:

① - Array Representation.
② Linked list Representation.

Array Representation: 2D array is used to represent a sparse matrix in which there are three row named as.

Row, Column, value.
- value of non-zero elemt.
- index of column where non-zero elemt located
- index of row where non zero elemt located

Exam.

$$\begin{bmatrix} 0 & 0 & 3 & 0 & 4 \\ 0 & 0 & 5 & 7 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 2 & 6 & 0 & 0 \end{bmatrix} \Rightarrow$$

row	0	0	1	1	3	3
column	2	4	2	3	1	2
value	3	4	5	7	2	6

Operations on linear Array:

1. Travarsal
2. Search
3. Insertion
4. deletion
5. Sorting
6. Merging.

1. Traversing linear array:

If we want to print element of array or count number of element in the array. this can be achived by traversing array.

Algo

LA → Linear array
LB - lower bound
UB - upper bound

1. set K = LB
2. Repeat steps 3 and 4 while K ≤ UB
3. Apply PROCESS to LA[K]
4. set K = K+1
5. Exit.

1. Repeat for K = LB to UB
2. Apply PROCESS
3. Exit.

```
void main()
{
int j=0, LA[5]={23, 24, 25, 26, 27};
while (K<5)
  {
  printf("%d", LA[K]);
  K++;
  }
}
```

```
#include<stdio.h>
void main()
{
int i, LA[7]={23, 45, 56, 1, -9, -12, 123};
for (i=0; i<7; i++)
  {
  printf("%d", LA[i]);
  }
}
```

Complexity = O(n)

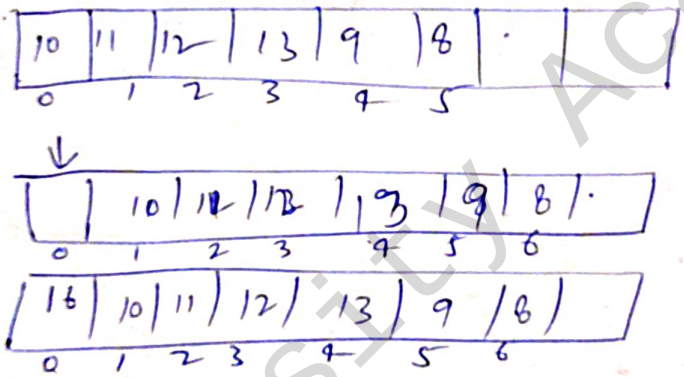
Insertion in Linear Array

Insertion refer to the operation adding another element to linear array,

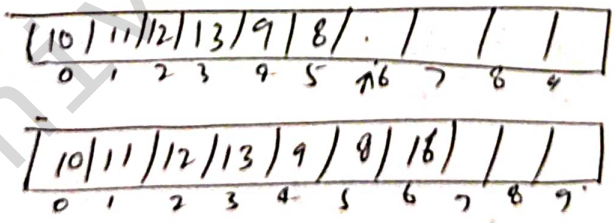
Insertion can be done as three ways!

1. Insert an element at beginning of ~~elen~~ Array
2. Insert an element at end of array
3. Insert ~~at~~ an element at given location (index).

Beginning of array: (worst case)
move all element one position to backwards to make a empty ~~loc~~ index at beginning of array.

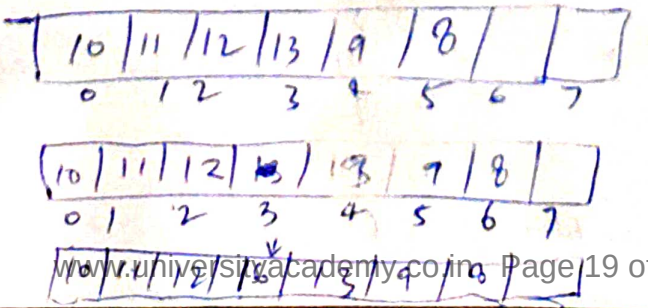


End of array: (Best case)
Just append the element at the location after last element.



At given position (Avg case)
let K be any location in the array:
Starting from K every element move one place backwork so that a empty position created at K. Now insert new element here.

K = 3



Algorithm:

INSERT(LA, N, K, ITEM)

1. set J = N
2. Repeat step 3 and 4 while J ≥ K
3. [move Jth element downward] set LA[J+1] = LA[J]
4. [Decrease Counter] set J = J-1
5. [Insert element] set LA[K] := ITEM
6. [Reset N] set N = N+1.
7. Exit.

Example:

LA: Array

1	2	3	4	5	6	7	8	9	10
21	2	43	14	-5	46	87	8		

N = 8

K = 4, ITEM = 99

1- J = 8

2. Repeat J ≥ K

set LA[J+1] = LA[J]

1	2	3	4	5	6	7	8	9
21	2	43	14	-5	46	87	8	

J = J-1 = 8-1 = 7

1	2	3	4	5	6	7	8	9	10
21	2	43		14	-5	46	87	8	

J = 4-1 = 3

5- Insert set LA[K] = ITEM

LA[4] = 99

1	2	3	4	5	6	7	8	9
21	2	43	99	14	-5	46	87	8

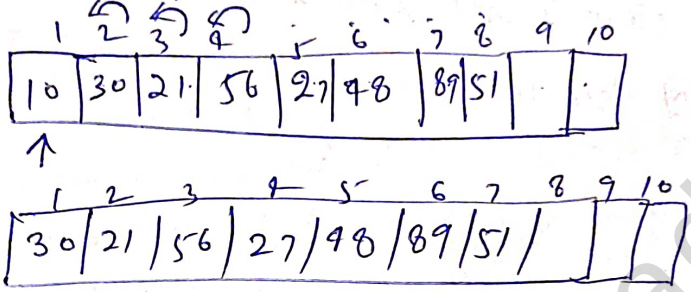
6- N = N+1 = 8+1 = 9

Deletion from Linear Array.

Deletion from Array means removing an element and replacing it with next element. It involve three cases.

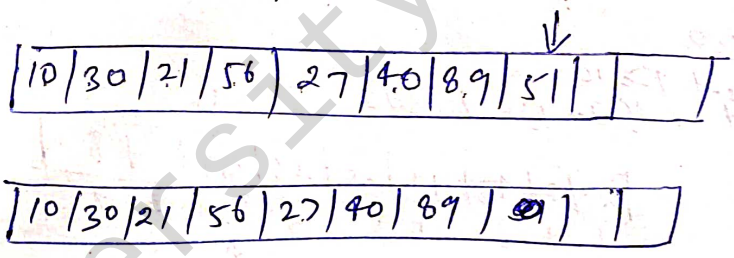
1- From the beginning of array: (worst case)

In this case we move all element one position forward to fill the position of the element at the beginning of array.



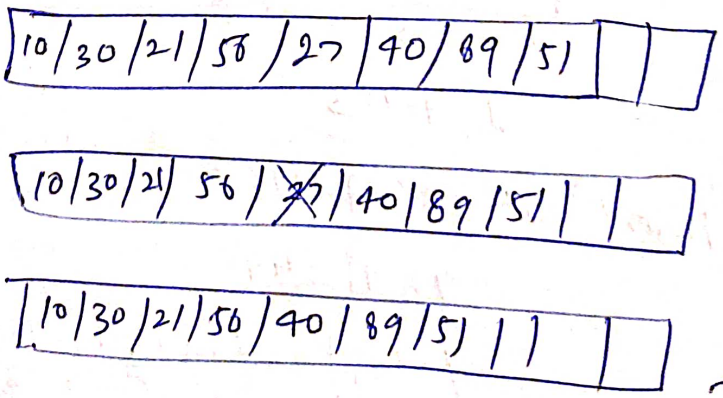
2. From end of array: (best case).

In this case we don't have to move any element. this is done by redefining index of last element of LA = N-1.



3. From Given position:

if we have to delete element at K position. to do this starting from K every element moved one place upword.



Algorithm.

DELETE(LA, N, K, ITEM)

1 - Set ITEM := LA[K]

2 - Repeat for J = K to N-1

move J+1st element upward] Set LA[J] = LA[J+1]

3. Set N = N-1

4 - exit.

Examples

	1	2	3	4	5	6	7	8
LA	21	2	43	14	-5	46	87	8

N=8, K=6 ITEM= 46

1- ITEM = LA[K] ⇒ ITEM = 46

2. J=K to N-1

⇒ K=6 to 7

	1	2	3	4	5	6	7	8
	21	2	43	14	-5	87	8	

J = J+1 = 7

	1	2	3	4	5	6	7
	21	2	43	14	-5	87	8

N = N-1 = 7

SORTING (Bubble Sort)

Let A be a list of n number. Sorting refer to the operation of rearranging the element of A so they are in increasing order

eg

$$A[1] < A[2] < A[3] \cdots < A[N]$$

eg:

original list 8, 4, 19, 2, 7, 13, 5, 16

after sorting 2, 4, 5, 7, 8, 13, 16, 19.

Here we present a very simple sorting algorithm Bubble sort.

Bubble Sort:

Suppose a array A with list A[1], A[2] --- A[N] in memory.

Step 1 - Compare A[1] and A[2] and arrange in order so that A[1] < A[2]

Compare A[2] and A[3] and arrange order so that A[2] < A[3]

⋮

Compare A[N-1] and A[N] and arrange in order so that A[N-1] < A[N]

(N-1 comparison)

after step 1 complete A[N] will contain the largest element.

Step 2 -> Repeat step 1 with one less comparision.

stop after compare

A[N-2] and A[N-1].

(N-2 comparison)

Step[N-1] Compare A[1] with A[2] and arrange them so th A[1] < A[2]

(1 comparison)

After N-1 step list will be sorted.

Example

	1	2	3	4	5	6	7	8
A :	32	51	27	85	66	23	13	57

Pass 1: Compare A[1] and A[2] since A[1] < A[2] ~~the list is not a~~ do nothing.

compare A[2] and A[3] interchange 51 and 27

32 (27) (51) 85 66 23 13 57

compare A[3] and A[4] since A[3] < A[4] do nothing.

compare A[4] and A[5] interchange 85 and 66

32, 27, 51, (66), (85) 23, 13, 57

compare A[5] A[6] interchange.

32, 27, 51, 66, (23) (85) 13, 57

Compare A[6] A[7] interchange.

32, 27, 51, 66, 23, 13, 85, 57

Compare A[7] A[8] interchange.

32, 27, 51, 66 23, 13, 57, 85.

(n-1) comparisons.

at the end of this pass largest element reached at last.

Pass 2. Compare A[1] A[2] no interchange.

32, 27, 51, 66, 23, 13, 57, 85

(n-2) comparisons

A[1] A[2] → 27, 32, 51, 66, 23, 13, 57, 85

A[2], A[3] → 27, 32, 51 66, 23, 13 57, 85

A[3] A[4] → 27, 32, 51, 66, 23, 13, 57, 85

27, 32, 51, 23, 66, 13, 57, 85

27, 32, 51, 23, 13, 66, 57, 85

27, 32, 51, 23, 13, 57, 66, 85

Pass 3

27, 32, 23 51, 13, 57, 66, 85

27, 32, 23, 13, 51, 57, 66, 85

Pass 4:

27, 23, 32, 13, 51, 57, 66, 85

Pass 5:

27, 23, 13, 32, 51, 57, 66, 85

23, 27, 13, 32, 51, 57, 66, 85

23, 13, 27, 32, 51, 57, 66, 85

Pass 6: 13, 23, 27, 32, 51, 57, 66, 85

Pass 7. Sorted :

Complexity of Bubble sort:

the time complexity in term of no. of comparision.

there are. $n-1$ comparision in first pass.

$n-2$ comparision in second pass

:

1 comparision in $(n-1)^{th}$ pass.

$$f(n) = (n-1)+(n-2)+(n+3)\cdots\cdots+2+1 = \frac{n(n-1)}{2} = \frac{n^2-n}{2}$$

$$= O(n^2).$$

Algorithm:

Bubble (Data, N) — Array, No of elements.

1. Repeat Step 2 and 3 for $K=1$ to $N-1$ // Pass.
2. Set PTR = 1
   Repeat while PTR $\leq N-K$ [Execute pass]
   (a) if Data[PTR] > Data[PTR+1] then.
   interchange Data[PTR] and Data[PTR+1]
   b. Set PTR = PTR + 1
3. Exit.

Searching in linear Array

Searching algorithm are designed to check an element or retrive element from data structure.

Let DATA be a collection of ITEM. Searching refer to finding LOC of ITEM in DATA.

generally searching classified in two categories.

1. Linear Search (Sequential search)
2. Binary Search.

Linear Search

- In this algorithm the given element is searched in given input array in sequential order. linear search is mostly used in unsorted array.

```
#include<stdio.h>
void main()
{
 int a[10]={10,23,40,1,2,0,14,13,50,9};
 int i, item, flag;
 printf("Enter the item to be search");
 scanf("%d",&item);
 for(i=0;i<10;i++)
   { if(a[i]==item)
       { loc=i;
         break;
       }
     else loc=-1;
   }
 if(loc==-1)
   printf(item not found)
 else printf(Item found at location %d, flag)
```

Algorithm:

Linear Search (Array A, value X)

1. Set i = 1
2. Repeate step 3 and 4 while i > n
3. if

Linear (DATA, N, ITEM, LOC)

① Repeat Step 2 for K=0 to-

② if DATA[i] == ITEM

Print "item found at location

3 if (i == N)

```
#include<stdio.h>
void main()
{
int a[5]={10,15,12,18,20};
int i, item, loc;
printf("Enter item which to be searched");
scanf("%d", &item)
for(i=0; i<5; i++)
  { if (a[i]==item)
      {loc=i;
       break;
      }
  }
if (i==5)
  printf("Item not found")
else
  printf("item found at location %d", loc);
}
```

Algo

LinearSearch (DATA, ITEM, N, LOC)

1- Repeat for i = 0 to n-1

2. if (DATA[i] == ITEM)
   {LOC = i;
   Break.

3. if (LOC == n)
   Print "Element not found"

else
   Print "Element found at index loc"

Complexity -

Best case = item present at first index: O(1)

Worst case = item present at last index O(n)

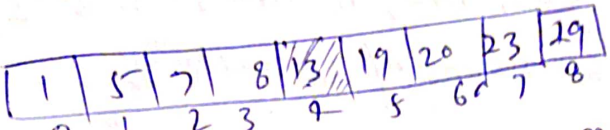
Avg. case = O(n)

Binary Search:

Binary Search technique which works on sorted array. It follow the Divide and Conquer approach.

Exa

Item = 23



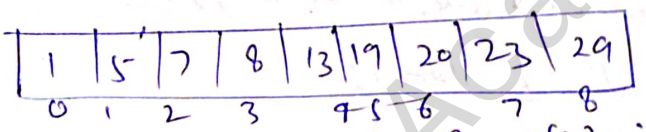
a[mid] = (0+8)/2 = a[4] = 13

23 > 13

beg = mid + 1 = 5

end = 8

mid = (5+8)/2 = 13/2 = 6



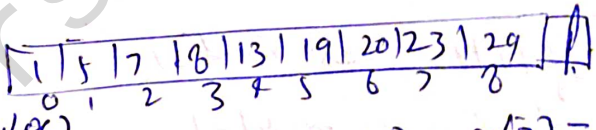
a[mid] = a[6] = 20

23 > 20

beg = mid + 1 = 6 + 1 = 7

end 8.

mid = (7+8)/2 = 7



a[mid] = a[7] = 23

23 = 23

item found

loc = mid = 7.

Algo BINAR(DATA, LB, UB, ITEM, LOC)

1- BEG = LB, END = UB, MID = int((BEG+END)/2)

2. Repeat step 3 and 4 while BEG ≤ END and DATA[MID] ≠ ITEM

3 if ITEM < DATA[MID] then
set END = MID - 1
else
set BEG = MID + 1

4 = set MID = int((BEG + END)/2)

5 = if DATA[MID] = ITEM then
set LOC = MID
else
set LOC = NULL

6 Exit

```
#include<stdio.h>
main()
{
int i, beg, end, mid, n, item, a[20];
printf("Enter the number of elemnt\n");
scanf("%d", &n);
printf("Enter the Elemnt for array)
    for(i=0; i<n; i++)
        scanf("%d", &a[i]);
Printf("Enter the value to find");
scanf("%d", &item);
beg=0;
end = n-1
mid = (beg+end)/2;
while((beg <= end) && (a[mid] != item))
    {
        if(item < a[mid])
            end = mid-1
        else
            beg = mid+1
        mid = (beg + end)/2
    }
if(a[mid] = item)
    printf("item found at location %d", mid);
else
    printf("Item not found");
}
```

Complexity: Best case → O(1) (item found at mid of array (on 1st try))

Worst case → O(log n)

Average case = observe that each comparison divide Array in half.

So how many divide required to reach at 1:

$1 = n/2^x \Rightarrow 2^x = n \Rightarrow \log(2^x) = \log n \Rightarrow x \log 2 = \log n$

$x = \log(n)$

Example

Suppose Data Array contain 1000000 elemt. observe that

$\Rightarrow \log n \cdot r$

$\log_2 1000000 = 20$ comparision Required.

Limitation of Binary Search :-

1- list must be sorted

2. One must have to direct access to the middle elemt.

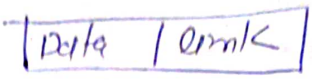
Note:- To remove limitations we have to use linked list or Binary Search tree.

Linked List

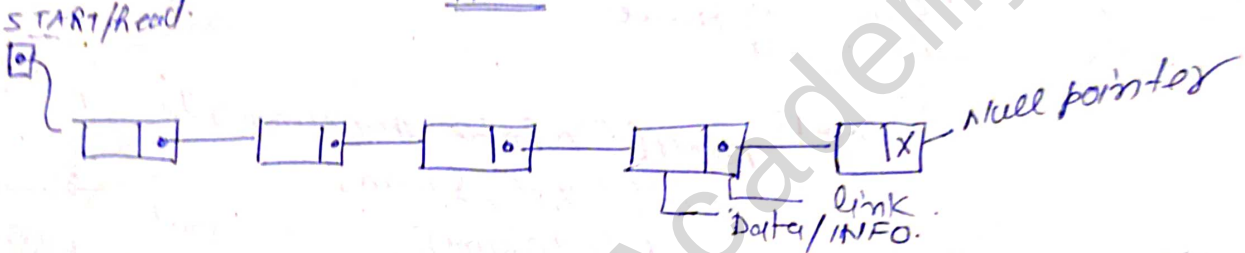
A linked list or one-way list is linear collection of data element called node where the linear order given by pointer.

Each node divided in to two parts. Data and link.

Data store actual information and link used to point to next node.



node



(inactual - 0 or negative no. used for null pointer)

Advantage of linked list.

1. Dynamic size.
2. Ease of insertion and deletion.

Disadvantage.

1. Random access is not allowed so binary search not possible.
2. extra memory at every node.

~~Array~~ Representation.

Struct node

{

int data;

int next;

}

List Require to maintain two linear array one for INFO and other for LINK.

example

	INFO/Data	link
1	K	7
2		
3	D	11
4	M	12
5	A	3
6		
7	L	4
8	G	1
9	S	0
10		
11	F	8
12	N	9

START = 5 INFO = A is first character

LINK = 3 INFO = D is second character

LINK = 11 INFO = F is third ch

LINK = 0, the Null value so list is ended.

```
#include<stdio.h>
void main()
{
  int ch, num;
  do
  {
      printf("\n1 : Create Linked list");
      printf("\n2 : Display ");
      printf("\n3 : Exit");
      printf("\n Enter your choice :");
       scanf("%d", &choice
      switch(choice)
      {
          case 1: printf("\n Enter any number");
                  scanf("%d", &num);
                  createList(num);
                  break;
          case 2: display();
                  break;
          case 3 : exit(0);
          default: printf("Invalid choice");
      }while(1);
  }
void createList(int num)
  {   struct node *q, *temp;
      if(START == NULL)
        {START= (struct node *) malloc (sizeof (struct node));
         START -> data = num;
         START -> next = NULL;
        }
      else
        {  q = START;
           while (q!=NULL)
             {
                q = q -> next;
             }
          temp = (struct node *) malloc (sizeof (struct node));
          temp -> data = num; temp => next = NULL; q -> next = temp;
        }
  }
```

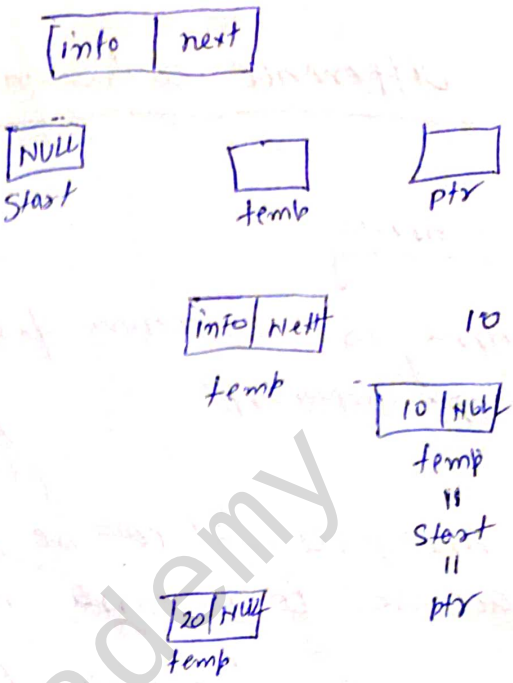
```
struct node
{
  int data;
  struct node *next;
} *START = NULL;
```

```
void creatList(int)
void display (void);
```

```
void display (void)
{
struct node *q;
 for(q=START; q!=ENUL;
        q=q->next);
{ printf("%d\n", q -> data);
}
```

start → NULL

a → NULL
a → next




```
#include<stdio.h>
#include<stdlib.h>
void create();
void display();
struct node
  {
    int info;
    struct node *next;
  };
 struct nod *start= NULL;
int main()
{
      int choice;
      while(1)
       {
        printf("\n 1. Create \n");
        printf("\n 2. Display\n");
        printf("\n 3. Exit \n");
        printf("Enter your choice");
        scanf("%d", &choice);
         switch(choice)
           {
             case 1:  create();
                      break;
             case 2:  Display()
                      break();
             case 3:  exit(0);
                      break;
             default: printf("\n wrong choice");
           }
       return(0);
}
```

```
voide create()
{
  struct node *temp, *ptr;
 temp = (struct node *) malloc(sizeof(struct node));
printf("Enter the value for node:\t");
scanf("%d", &temp->info);
temp->next = NULL;
if (start == NULL)
   {  start = temp
   }
else
{  ptr = start;
while (ptr->next != NULL)
      {  ptr = ptr->next;
      }
   ptr->next = temp;
}
```

```
voiddisplay()
{ struct node *ptr
printf("\n The list of elements:\n");
 for(ptr = start; ptr != NULL; ptr = ptr->next)
    printf("%d\n", ptr->info);
}
```

Difference Between Linked list and Array

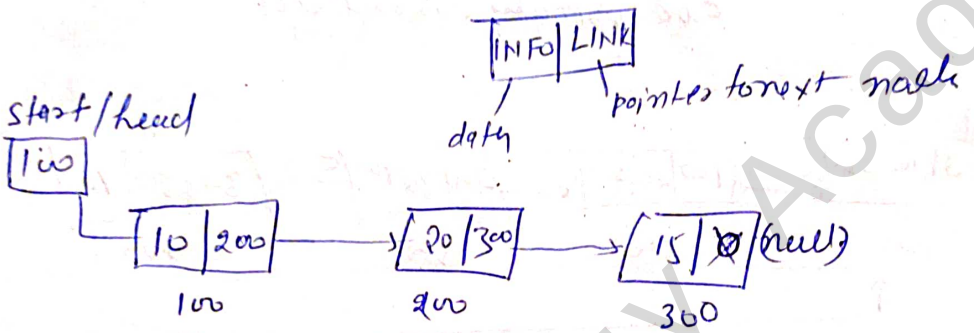
Array	Linked list
1- Array is a collection of similar data type	1- Linked list is an ordered collection of similar data, each element connected using pointer.
2. Array element can be randomly accessed using array index.	2. Random access is not possible, the element can be accessed sequentially.
3. Data element are stored in contiguous location in memory	3. new element can be stored any where and a refence is created using pointer.
4. Insertion deletion is not easy as compare to linked list	4. Insertion deletion is very easy.
5. Memory allocated during compile time (static memory allocation)	5. Memory allocated during run time (dynamic memory Allocation)
6. size of array must be specified at time of declaration.	6. size of linked list grow/shrink as when new element Inserted/deleted.

Types of Linked list

There are three common linked list

1- Singly linked List
2- Doubly linked List
3- Circular Linked list

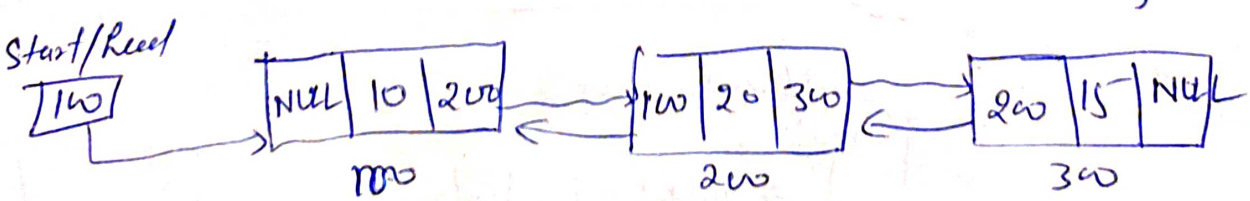
1- Singly linked List : This most common. each node has data and a pointer to the next node. it is one way list




```
struct node
{
int info
struct node * next;
}
```

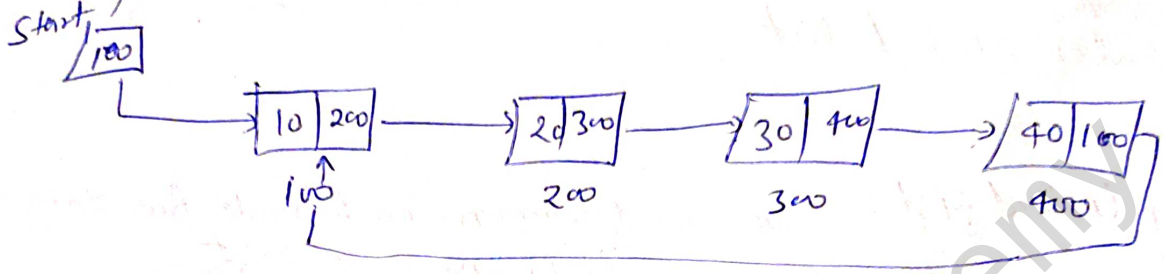
2. Doubly Linked List : Doubly linked list is a variation of Linked list which navigation in both way either forward or backward. node in doubly linked list contain three part. Data, Next, Prev.


```
Struct node
{ int info;
struct node * forw;
struct node * back;
}
```

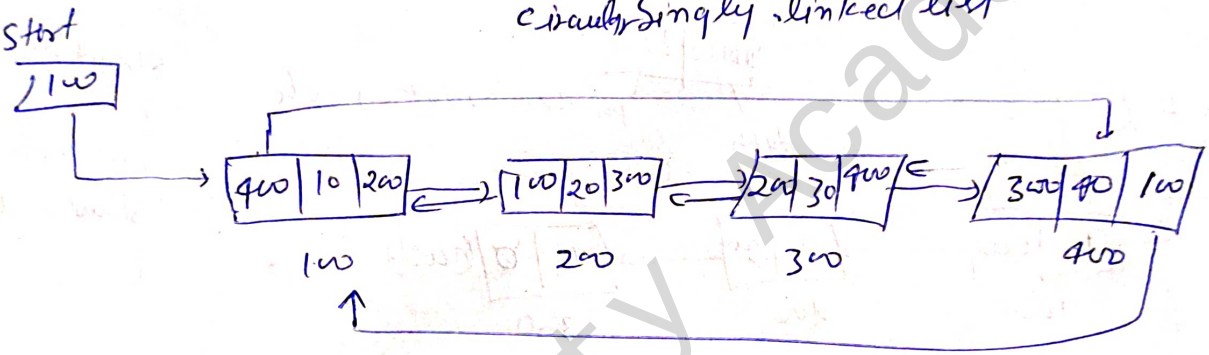


3. Circularly Linked List:

Circularly linked list is a special type of linked list. In circularly linked list, the link field of last node points to the first node of list

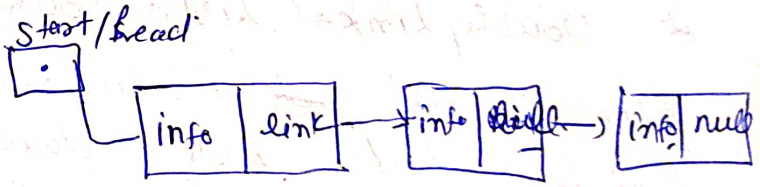


circularly Singly linked list.



Operations on singly list

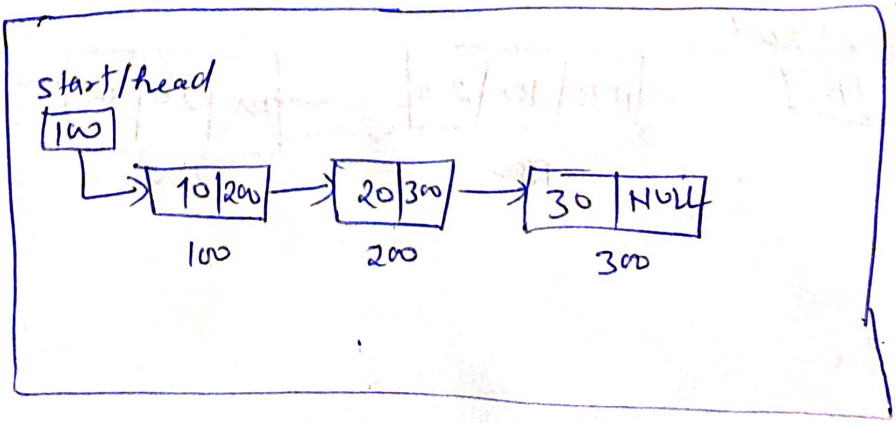
1. Traverse
2. Insert
3. Delete.



start → points to the first node.
info → stored data/information
link → pointer to store address of next element
NULL → shows the last element node.

Example.

```
struct node
{
  int data;
  struct node *next;
};
```



1. Traverse a Linked List.

Step1 start with head of list. Access the content of head node if it is not null

Step2. then go the next node and access the node information.

Step3 continue until no none node.

Algorithm

1. Set PTR := START
2. Repeat Step 3 and 4 while PTR ≠ NULL
3. write: INFO(PTR)
4. set PTR = LINK[PTR]
5. Return.

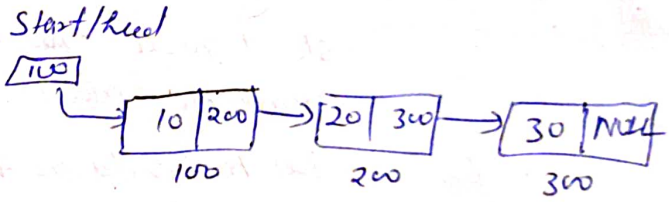
```
struct node *temp = head
printf("\n list of element are \n");
while (temp != NULL)
{
printf("%d ", temp->data);
temp = temp->next;
```

output: 10 20 30

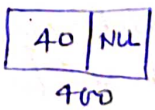
---

2. Insertion into a Linked List.

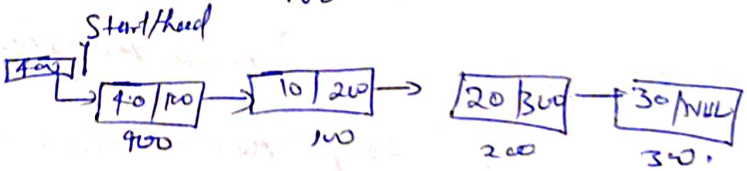
1. Insertion at the beginning
2. Insertion at the end
3. Insertion after a given node.



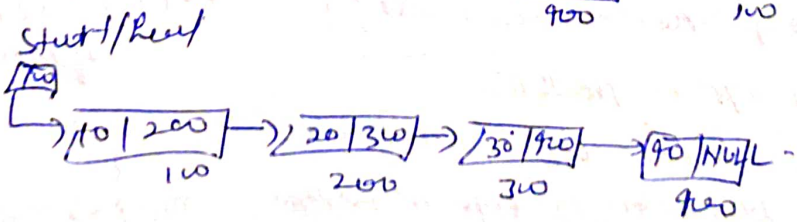
(a) Insert.



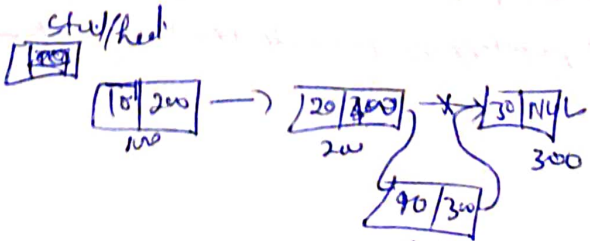
Case1



Case2



Case 3 Insert after 20



(a) Add to beginning :-

1- Allocate memory to new node
2- store data
3- change next of new node to point to head
4- change head to point to recently created node.

```
Struct node *newNode;
newNode = malloc (sizeof (struct node));
newNode -> data = 40;
newNode -> Next = head;
head = newNode;
```

(b) Add to End.

(1) allocate memory for new node.
2. store data.
3- traverse to last node
4. change next of last node to recently created node.

```
Struct node *newNode;
newNode = malloc (sizeof (struct node));
newNode -> data = 40
newNode -> next = NULL;
Struct node *temp = head;
while (temp -> next != NULL)
    {
        temp = temp -> next;
    }
temp -> next = newnode;
```

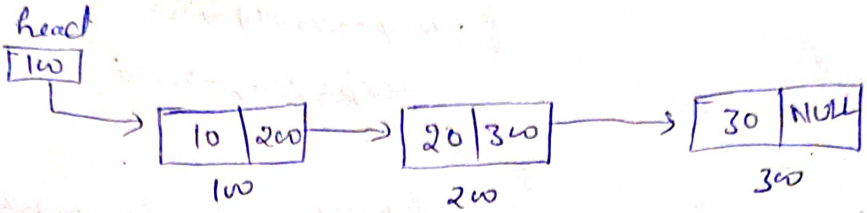
(b) Insertion after a given position.

(1) Allocate memory and store data for new node.
(2) traverse to node just before the required position.
3. change next pointers to include new node between.

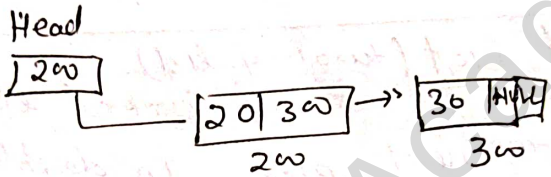
```
Struct node *newNode
newNode = malloc (sizeof(struct node));
newNode -> data = 40;
int position; i;
printf("Enter a position");
Scanf("%d", &position);
Struct node *temp = head;
for (i=2; i<position ; i++) {
    if (temp -> next != NULL)
        temp = temp -> next;
    }
}
newNode -> next = temp -> next
temp -> next = newNode;
```

3. Deletion from a Linked list:

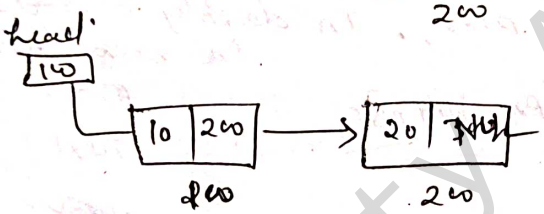
(a) Delete from beginning
(b) Delete from end.
(c) delete element before given position. (Delete from middle):



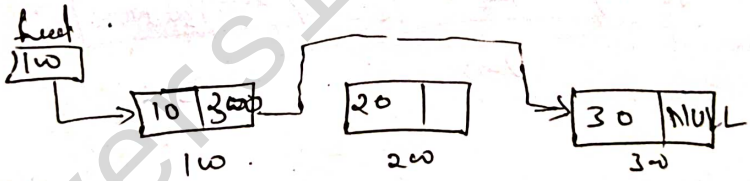
(a)



(b)



(c)



(a) Delete from Beginning:
* Point head to second node.

```
head = head -> next;
```

(b) Delete from end:
* Traverse to second last element
* change its next pointer to null.

```
struct node *temp = head;
while (temp -> next -> next != NULL)
    { temp = temp -> next;
}
temp -> next = NULL;
```

(c) Delete from Middle

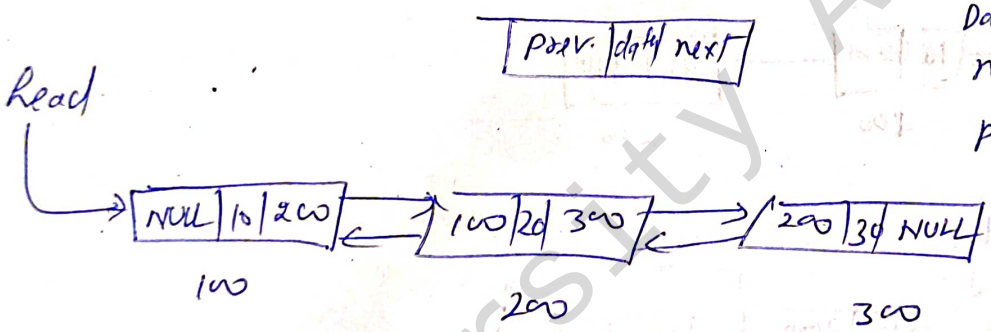
* Traverse to element before the element to be deleted.
* change next pointers to exclude the node from the chain.

```
for(int i=2; i<position; i++) {
    if (temp->next!=NULL) {
        temp=temp->next;
    }
}
temp->next=temp->next->next;
```

Doubly linked list (two way list)

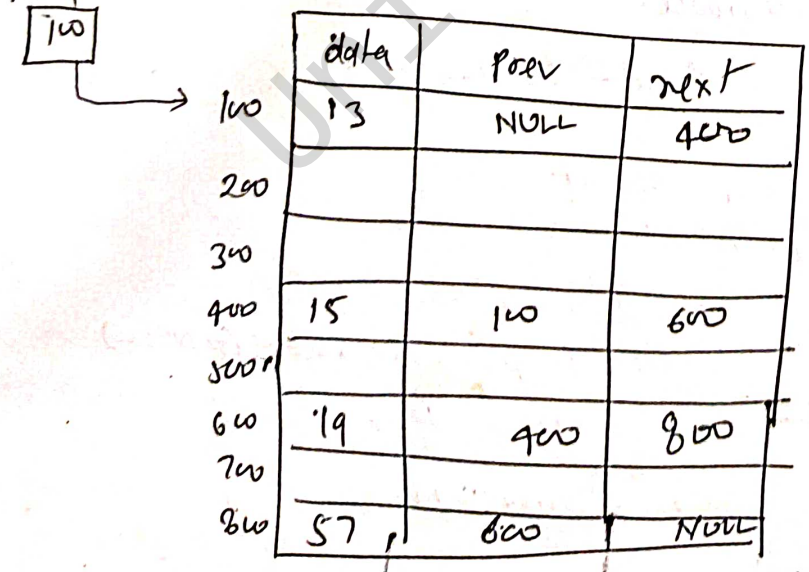
Doubly linked list contain three part data, next, prev,. In doubly linked list we can traverse in both direction.

Data - value stored.
next - pointer to next node
prev - Pointer to previous node.




```
struct node
{
    int data
    struct node *next;
    struct node *prev;
}
```

Memory Representation.

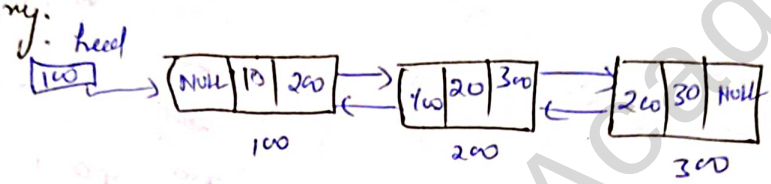


Memory Representation

Doubly linked List operation.

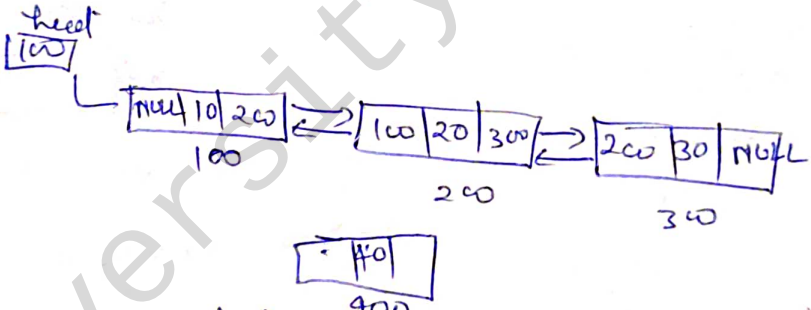
1- Traversing ← Forward, Backward
2- Insertion → at beginning, at end, at location.
3- Deletion. → from beginning, from end, from middle.

i- Traversing.

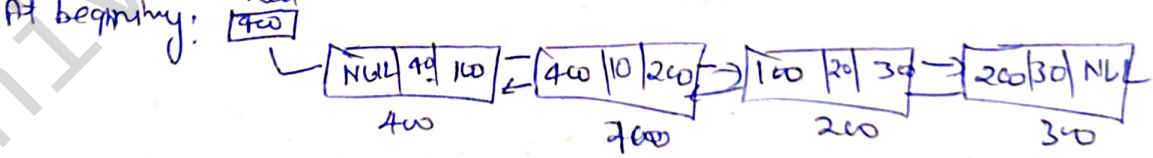


Forward= 10 20 30
Backward 30 20 10

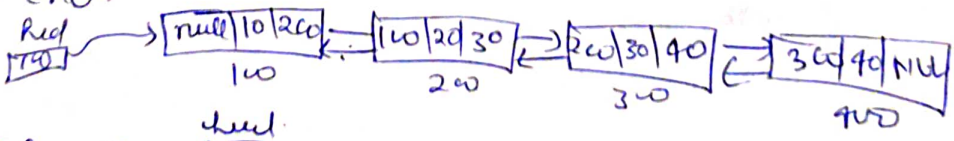
② Insertion.



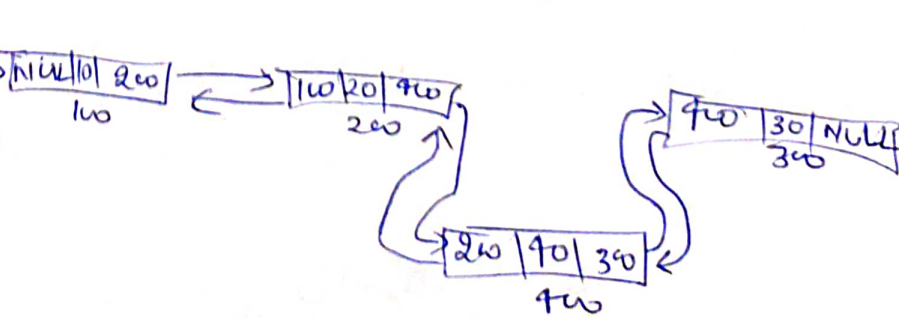
At beginning:



at end:

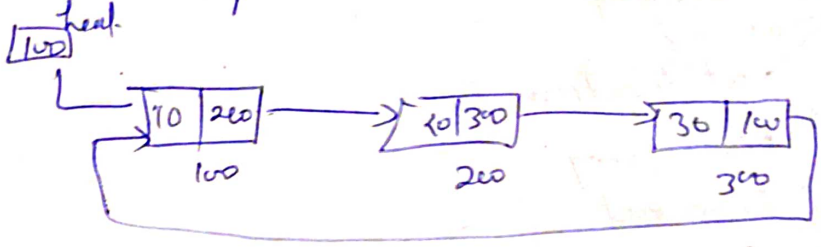


at location:



Circular linked list (singly)

The last node of linked list contain pointer to the first node of the list.



1. Traversal
2. Insertion
3. deletion.

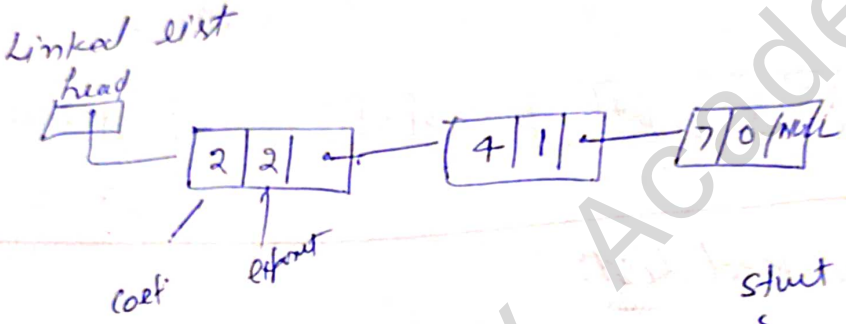
	Data	Next
100	5	
200	10	
300		
400	20	
500		
600	30	
700	40	
800		

Polynomial Representation Using linked list.

The linked list can be used to represent polynomial of any degree. Polynomial is a mathmatical expression that consist of variable and coefficients. eg.

$$2x^2 - 4x + 7$$

Coefficient and exponent of the polynomial are defined as the data node of list




```
struct node
{
int coeff
int pow;
struct Node *next;
};
```

Addition of two polynomials.

$P(1) = 5x^{12} + 2x^9 + 4x^7 + 6x^6 + x^3$

$P(2) = 7x^7 + 2x^7 + 8x^6 + 6x^4 + 2x^2 + 3x + 40$

Step1 Loop arround all value of linked list and follow step 2 and 3.

Step2 if val of node exponet is greater copy this node to result and head towards the next node.

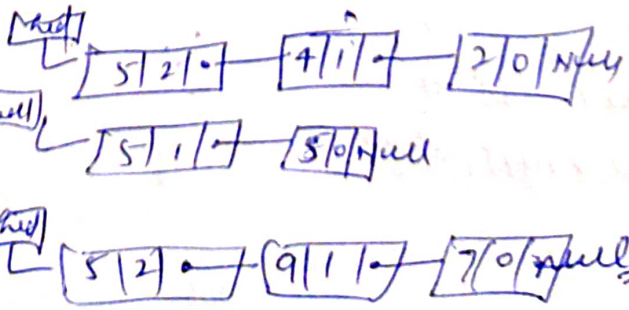
Step3 if the value of both node's exponet is same add the coffient and added to result.

Step4 Print resultant node.

Ex: $P(1) = 5x^2 + 4x + 2$

$P(2) = 5x + 5$

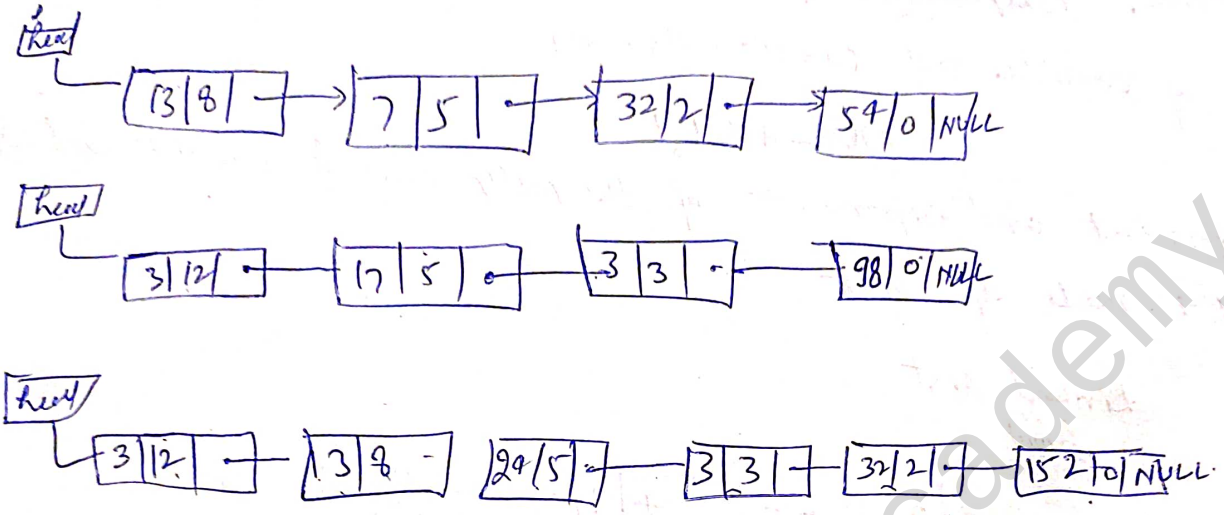
Result $5x^2 + 9x + 7$



Example

$P(1) = 13x^8 + 7x^5 + 32x^2 + 54$

$P(2) = 3x^{12} + 17x^5 + 3x^3 + 98$



Generalized Linked List:

A generalized linked list is useful for ~~multiple~~ representation of multiple variable polynomial equation. A generalized linked list L is defined as finite sequence of $n >= 0$ elements $l_1, l_2 \ldots l_n$ such as l_i are either atom or list of atom.

$L = (l_1, l_2, l_3 \ldots l_n)$. where n is total no. of node in list

Node structure:

Flag: Flag = 1 ⇒ down pointer exist.
Flag = 0 ⇒ Next pointer exist

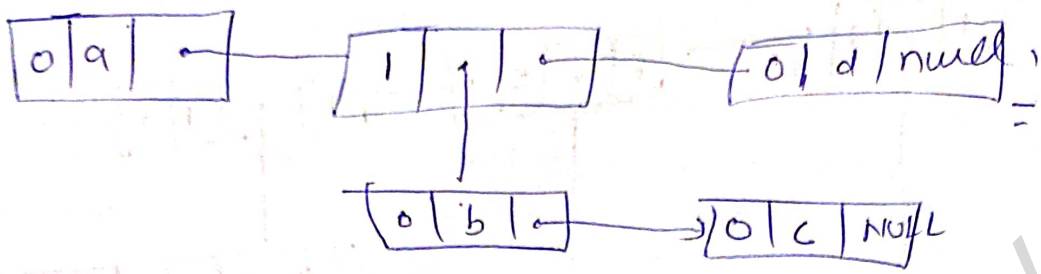
Data ⇒ atom.

Down pointer → address of node which is down to current node

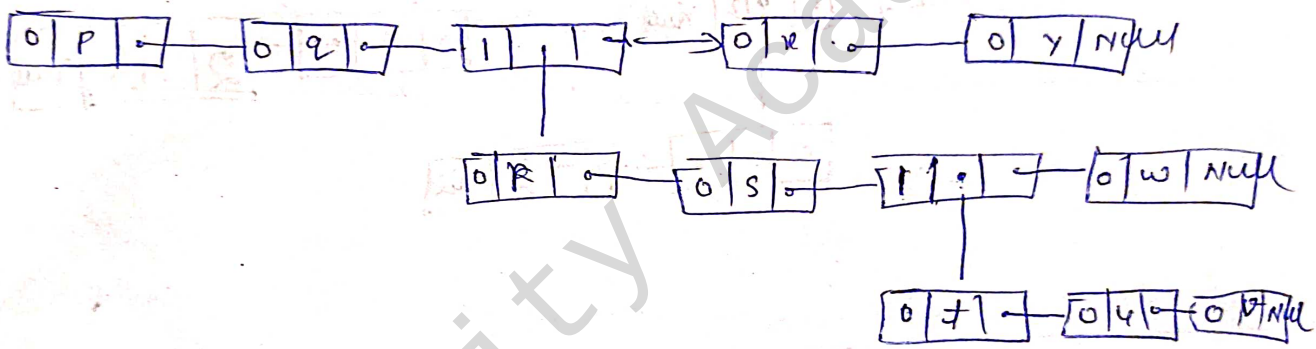
Next pointer → address of next node;

```
struct node
{
  char c;   /data
  int indent /flag
  struct node *next, *down;
}
```

(a) (a, (b, c), d)



(b) L = (P, Q, (R, S, (T, U, V), W) X, Y)

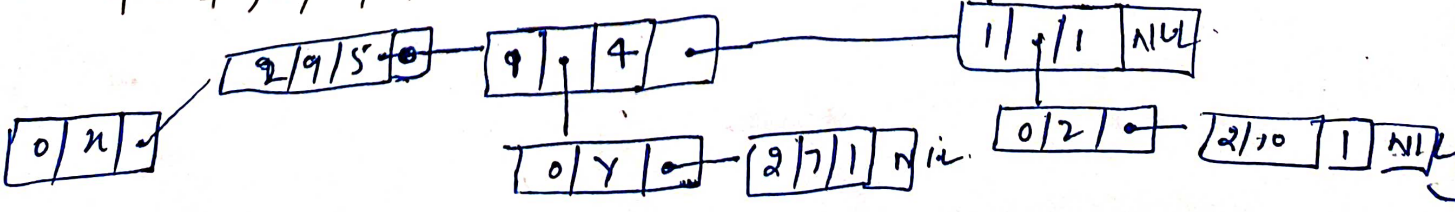


Polynomial Representation using Generalized linked list.

Generalized linked list can be used to represent a multi-variable polynomial.

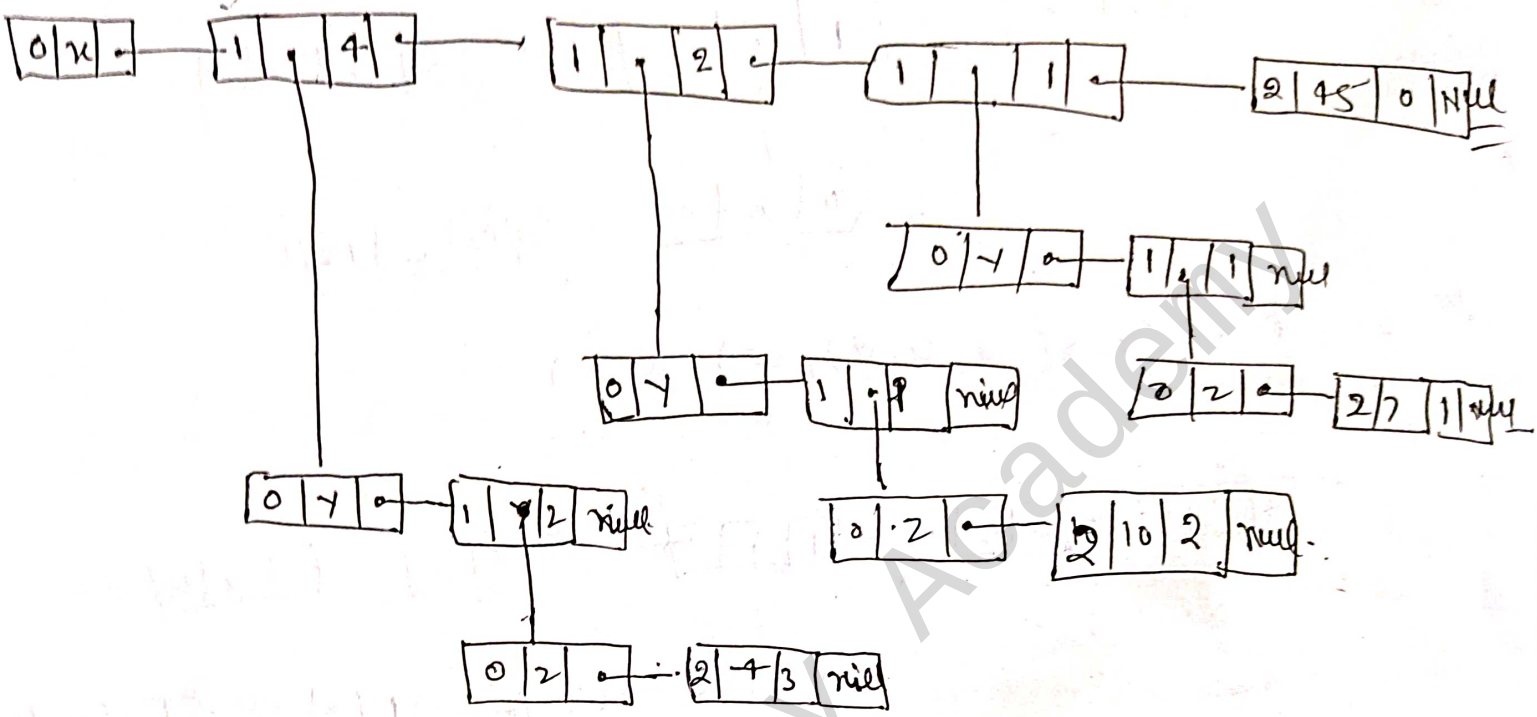
Flag = 0 means variable present.
1 means down pointer present
2 means coefficient, exponent present.

ex: $9x^5 + 7x^4y^2 + 10x^2$



Example: 2

$$-4x^4y^2z^3 + 10x^2yz^2 + 7xyz + 45$$



UNIT - II

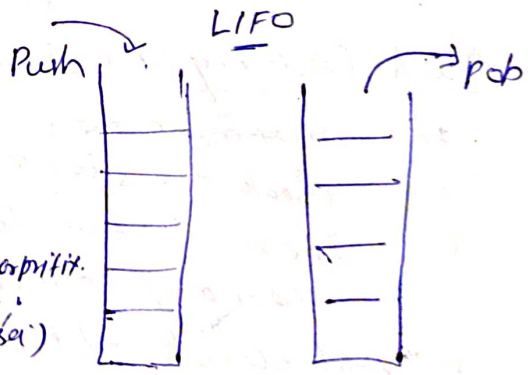
STACK

A stack is list of elements in which an element may be inserted or deleted only at one end called top of stack. Stacks are sometime called Last-in-First-out (LIFO) list i.e. the element which inserted first in the stack will be deleted last from the stack. or FILO - first in last out.

ex. Stack of plate in restorent,
Stack of moulded chairs.

Basic features of stack:

1. Stack is an ordered list of similar data type.
2. Stack is a LIFO structure
3. Push() function used to insert element and pop() function used to remove an element from stack
4. Stack said to be Overflow → when it is completly full and said to be Underflow → when it's completly empty



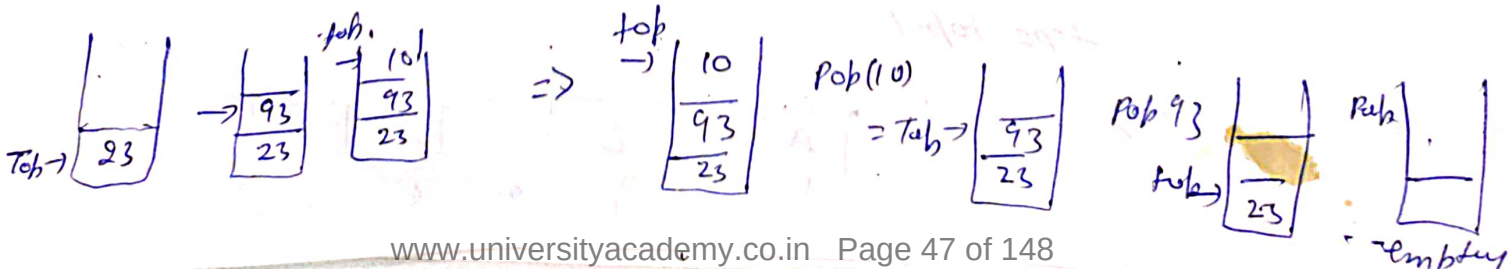
Stack.

Applications of stack.

1. Recursion.
2. Expression evaluation (infix to Postfix, prefix to Postfix, Postfix to Prefix, vice versa)
3. Parsing
4. Tree Traversals.
5. Backers:

Example suppose following three item push into to stack and pop each element after all element pushed.

23, 93, 10



Basic operations of stack.

1- Push : Add element to top of stack
2. Pop : : Remove an element from top of stack.
3- Is Empty : check if the stack is empty
4. Is full : check if the stack is full
5. Peek : get the value of top element without removing it.

Array Implementation of stack.

Stack may be represent in computer in various ways. usually represent by one-way list or a linear array.

Program

Top = -1 means stack is empty (underflow)
top = 0 means stack has only one element
Top = n-1 stack is full.
top = n ~~Sta~~ Overflow.

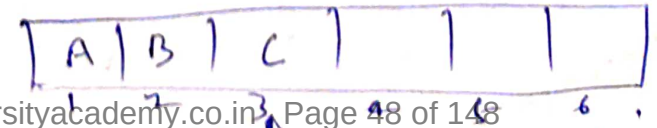
Push. (Adding an element) -O(1)

~~1- increment the~~
1- check the stack is not full.
2. Increment the Top by one to refer next memory location.
3. Add element at the position of incremented top.

Algo.
1- if Top=n (over flow)
2- top = Top+1
3. stack(top) = item
4- end;

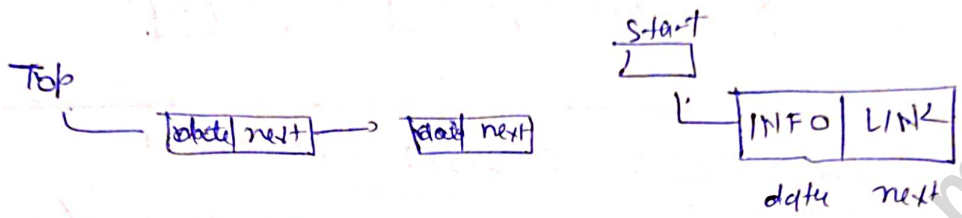
Pop (Removing an element from stack. = O(1)

begin if Top = 0 then underflow (stack is empty)
item = stack(top);
top = top-1;
end;

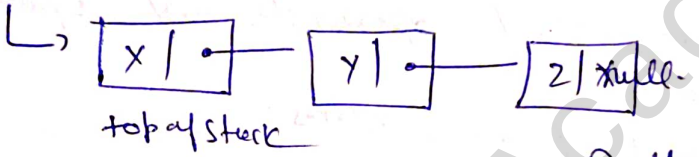


Link list Implementation of stack.

Instead of using array, we can also use linked list to implement stack. linked list allocates the memory dynamically. each node contains data and next part.



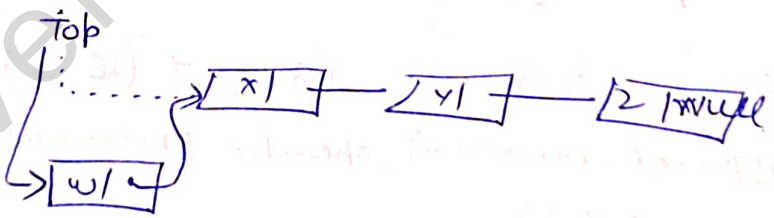
eg. Top (Start)



Push:

1- create node first and allocate memory to it
2. if list is empty then pushed as the start node to the list. assign value to info part and address part is null.
3. if list is not empty add new node to beginning of list.

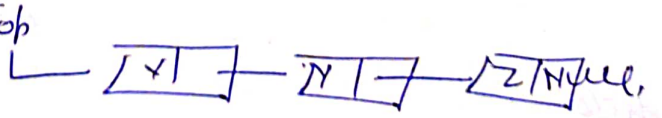
Pus(w) =



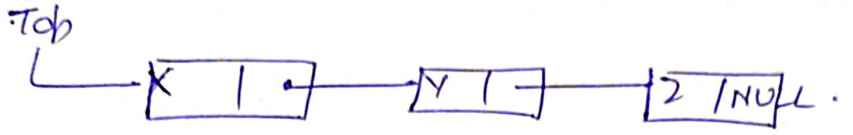
Pop:

1- check the underflow condition.
2. Adjust the head pointer.

Pop(w)



Example- Consider the linked list shown in fig. and perform following operation.
(i) Pass B (ii) POP (iii) Pop (iv) Pus M



ARITHMETIC EXPRESSIONS: POLISH NOTATION

The arithmatic expression invalve constants and operations. Constantant some time called operands and operation some time called operators.
there are tree type of expressions.

1- infix 2- postfix (Reverse polish) 3- Prefix (Polish Notation)

the application of stack in conversion of expression and evaluations.

Infix

infix expression = operand1 operator operand2

eg: (a+b) (a+b) * (k-d) (a+b)/e) * (d+f)

required: Parenthesis, operator precedence; associativity:

Postfix (Reverse Polish): Postfix = operand1 operand2 operator.

eg:

ab+, ab+cd-*, ab+e/ df+*

Prefix (Polish Notation)

+ab, *+ab-cd, */+abe+df.

Note: operation precedence and associativity required to solve infix expression but not required in postfix and prefix.

Example

$$2 \uparrow 3 + 5 * 2 \uparrow 2 - 12 / 6$$

$$= 8 + 5 * 4 - 12 / 6$$

$$= 8 + 20 - 2$$

$$= 26.$$

Assume that three level of Precedence. Highest : - exponent ($\uparrow$)
Next Highest : multiplication and division.
lowest - addition and subtraction.

Infix to Post fix Conversion

Q: A + (B * C - (D / E $\uparrow$ F) * G) * H

Q: A + (B * C - (D / E $\uparrow$ F) * G) * H).

Symbol	stack	output P.
	(	
A	(	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
$\uparrow$	(+ (- (/ $\uparrow$	A B C * D E
)	(+ (-	A B C * D E $\uparrow$ /
*	(+ (- *	A B C * D E $\uparrow$ /
G	(+ (- *	A B C * D E $\uparrow$ / G
)	(+	A B C * D E $\uparrow$ / G * -
*	(*	A B C * D E $\uparrow$ / G * -
H	(*	A B C * D E $\uparrow$ / G * - H
)		A B C * D E $\uparrow$ / G * - H * +

Algo POLISH(Q, P)

1. Push "(" onto stack and add ")" to end of Q.
2. Scan Q from left to right and Repeat step 3 to 6 for each element of Q until stack is empty
3. if an operand encountered add it to P.
4. if left parenthesis encountered push it onto stack.
5. if an operator ⊗ is encountered then
   (a) Repeatly pop from stack and to P each operator which has the same precedence as or higher precedence than ⊗.
   (b) Add ⊗ to stack
6. if Right parenthesis encountered then
   (a) Repeatedly pop from stack and add to P each operation until left parenthesis is encountered.
   (b) Remove left parenthesis.

) left.

Infix to Prefix

1- Reverse the infix expression ie. $(a+b)*(c-d)$ will become

$(d-c)*(b+a)$

2- Apply postfix algorithm and obtain postfix notation.

3. reverse the postfix expression

Example $(A*B)+C$ → 1. Reverse $C+(B*A)$

$Q = C+(B*A))$ 2 Postfix. $CBA*+$

3. Reverse $+*ABC$ Ans

Symbol	Stack	Postfix
Q	(	
C	(	C
+	(+	C
(	(+(	C
B	C+(	CB
*	(+(*	CBA
A	(+(	CBA*
)	(+	CBA*+
)	-	

→ Reverse. $+*ABC$ Ans

Postfix to Infix

1- Read the postfix expression from L → R

2- if we read operand push to stack

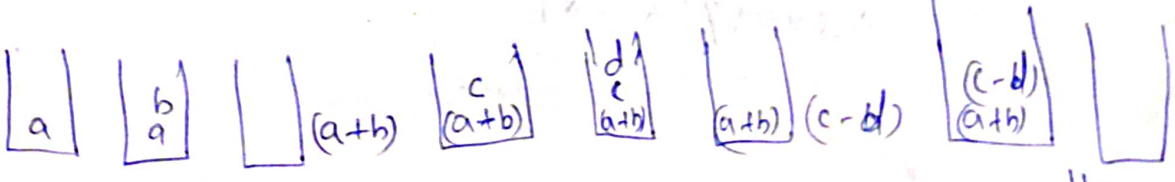
3- if we read operator pop top two operand.

and make expression (operand1 operator operand2)

Push this expression on to stack

4. go to step1 until complete.

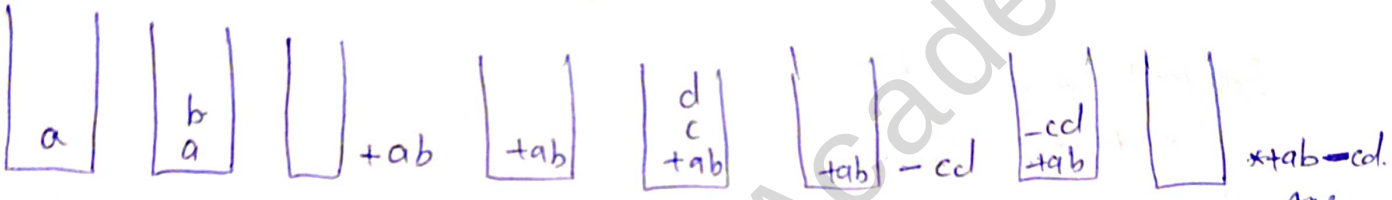
Example $ab+cd-*$



Postfix to Prefix:

1- Read the postfix expression from L – R
2. if we read operand Push it to stack
3. if we read operator pop two operand first pop operand is OP2 and second poped operand as OP1. form it in prefix as (operator OP1 OP2 * then push it to stack.

Exam ab + cd – *



Ans

Prefix to Infix:

1- Reverse the prefix expression
2. Apply postfix to infix conversion process.
3. Reverse infix to obtain final Answer

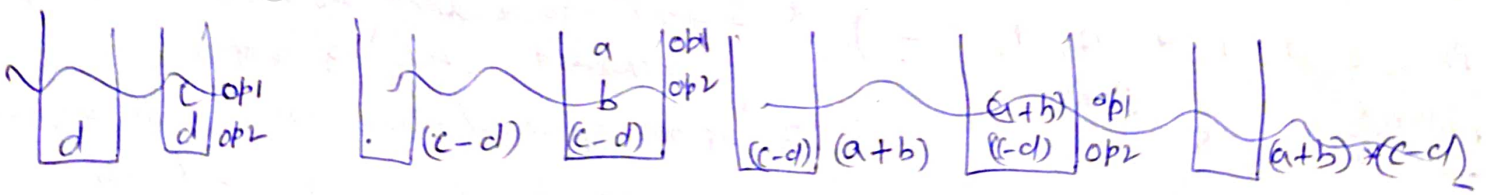
example

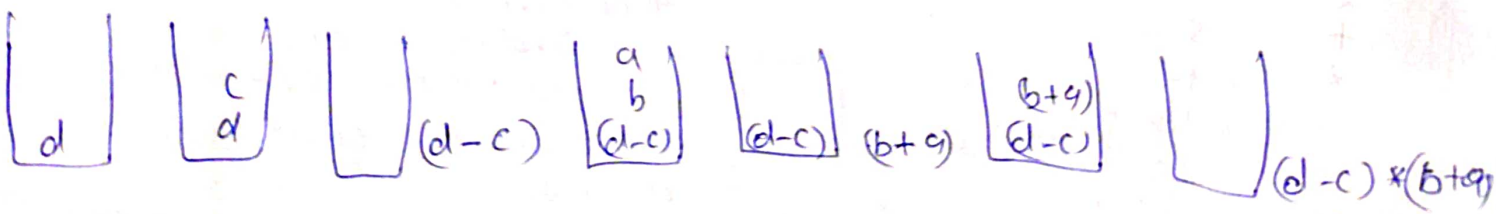
*+ab–cd

Step1 Reverse it

dc – ba + *

2.





Reverse it. (a+b)*(c–d)

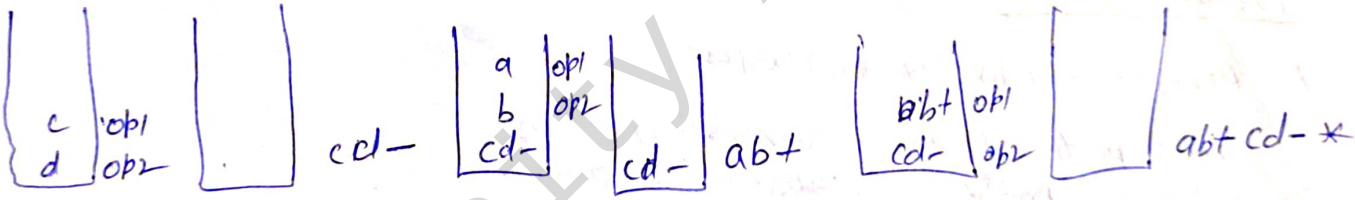
Ans

Prefix to postfix:

1. Reverse the given expression
2. Read from left to right
3. ~~Apply postfix to Prefix~~
   if we Read operand push to stack
4. if we Read operator Pop two top of stack
   First Pop is OP1 second POP is OP2 make it
   "OP1 OP2 operator"

Example ~~*ab+cd-~~
*+ab-cd

1. Reverse it
dc-ba+*



Answer

Evaluation of Postfix Expression:

eg.

P= 5 6 2 + * 12 4 / -

P= 5, 6, 2, +, *, 12, 4, /, -,)

Symbol	Stack
5	5
6	5,6
2	5,6,2
+	5,8
*	40,
12	40,12
4	40,12,4
/	40,3
-	37 ← Answer
)	

Algo:

1. Add Right Parenthesis ')' at the end of P.
2. Scan P from left to right till ')'
   repeat step 3 and 4.
3. if an operand encountered Push to stack
4. if an operator (*) encountered then
   a) Remove TOP two element from stack
      ∴ First Pop is OP1
      Second POP is OP2
   b) Evaluate OP2 (*) OP1
   c) Push Back to Stack.
5. Set value equal to the top of STACK.
6. Exit.

Recursion:

The process in which a function call itself directly or Indirectly is called Recursion. In Recursion a function 'A' either called itself directly or call a function 'B' that turn calles the original function 'A'.

eg.

```
int fun1( )
{
  ----
  fun1( );
  ----
}
```

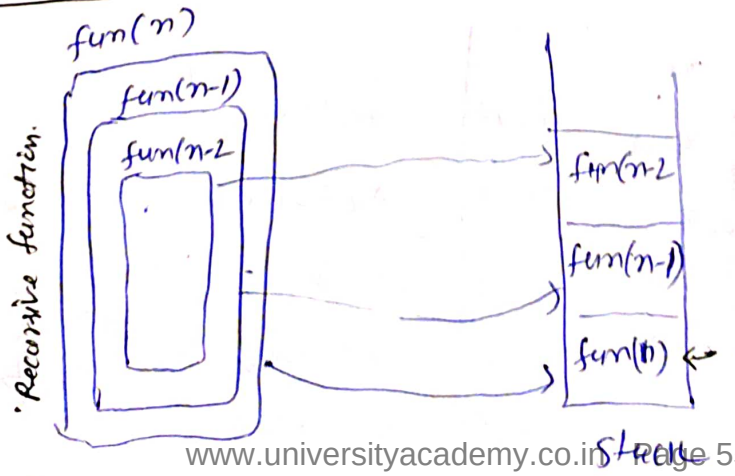
or

```
int fun1( )
{
  ----
  fun2( );
  ----
}
int fun2( )
{
  ----
  fun1( )
  ----
}
```

A Recursive function can go infinite like loop. to avoid infinite running of Recursive function there are two properties that Recursive function must have

1- Base criteria: criteria when function stop.
2- Progressive approach: each time come closer to base criteria.

Stack Implementation:

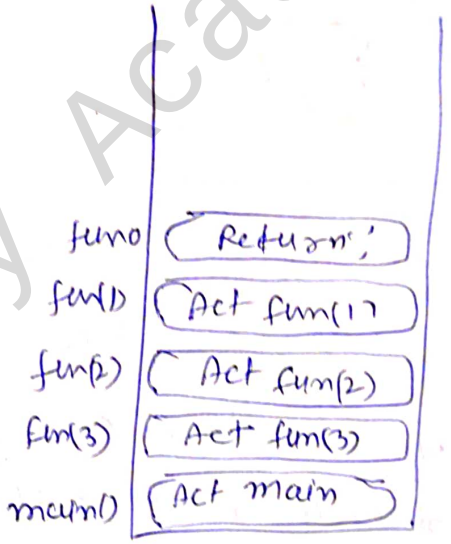


Types of Recursion

1- Direct Recursion
2- Indirect Recursion
3 Tail Recursion
4. Non-Tail Recursion

Tail Recursion: A Recursive function is said to be tail recursive if the recursive call is the last thing done by the function. there is no need to keep Record of previous state.

```
void fun( int n) {
   if (n==0)
     return;
   else
     printf ("%d", n);
   return fun(n-1);
  }
int main( )
  {
   fun(3);
   return 0;
  }
```

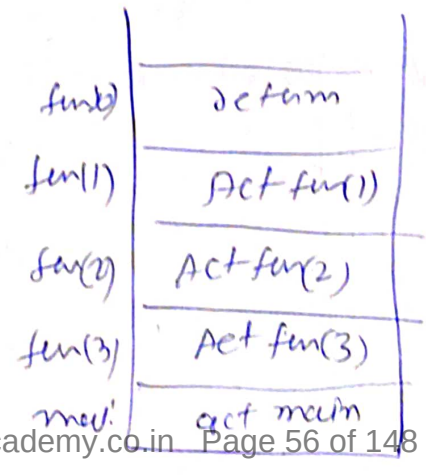


output 3 2 1

Non-Tail Recursion: A recursive function is said to be non-tail Recursion if recursive call is not last thing done by the function. After Returning back there are something left to evaluate

eg

```
void fun(int n){
   if(n==0)
     return;
   fun(n-1);
   printf("%d",n);
  }
int main( )
 {
  fun(3);
  return 0;
 }
```



output: 1 2 3

Example of Recursion:

1. Factorial
2. Fibonacci
3. Towers of Hanoi:

Factorial

The product of positive integer from 1 to n is called factorial of n usually denoted by n!.

$$n! = 1 * 2 * 3 * \cdots (n-1) * n$$

eg. $0! = 1$, $1! = 1$ $2! = 2$ $3! = 6$

$$5! = 1 * 2 * 3 * 4 * 5 = 120$$

Eal calcul 4!

1. $4! = 4 * 3!$
2. $3! = 3 * 2!$
3. $2! = 2 * 1!$
4. $1! = 1 * 0!$
5. $0! = 1$
6. $1! = 1$
7. $2! = 2$
8. $3! = 3 * 2 = 6$
9. $4! = 4 * 6 = 24$

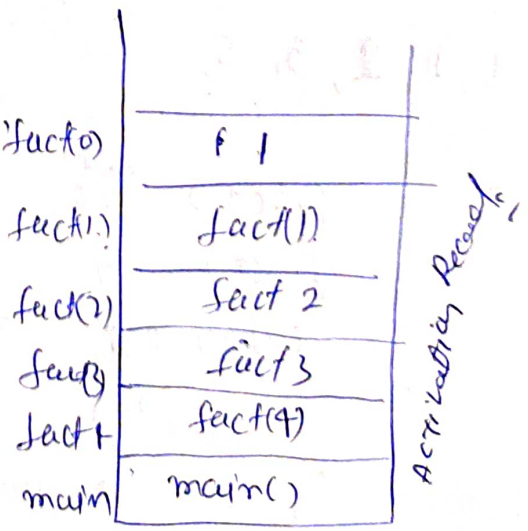
ANS:

Algo

```
int fact (int n)
{
  if (n==0)
    return 1;
  return n * fact(n-1);
}
```

Recursive function

$$fact(n) = \begin{cases} 1 & 0 \\ n * fact(n-1) & n>0 \end{cases}$$



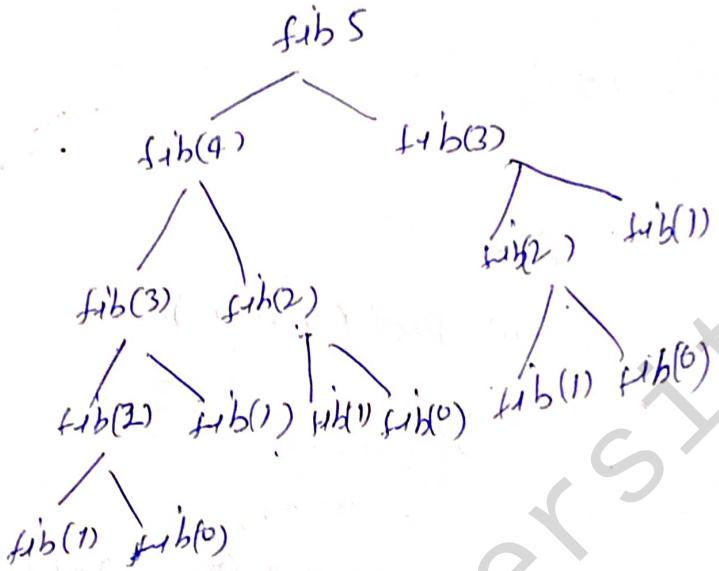
2. Fibonacci Sequence

The fibonacci sequence usually denoted by $(F_0, F_1, F_2 \cdots)$ is as follows.

0 1 1 2 3 5 8 13 21 34 55 ------

Set $F_0 = 1$ $F_1 = 1$ and each succeeding term is sum of two preceding term.

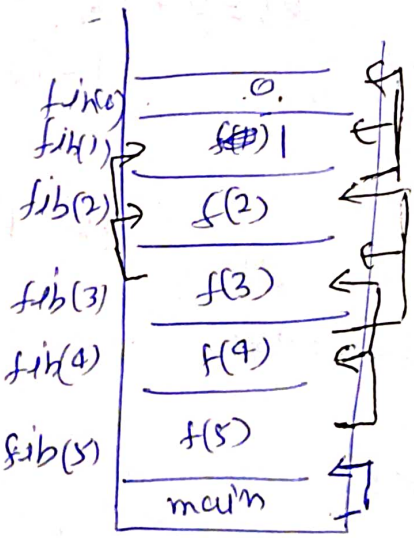
Eg:
n=5



Algo

```
fib (n)
{
 if n <= 1
   return n;
 else
   return fib(n-1) + fib(n-2)
}
```

0, 1, 1, 2, 3, 5,



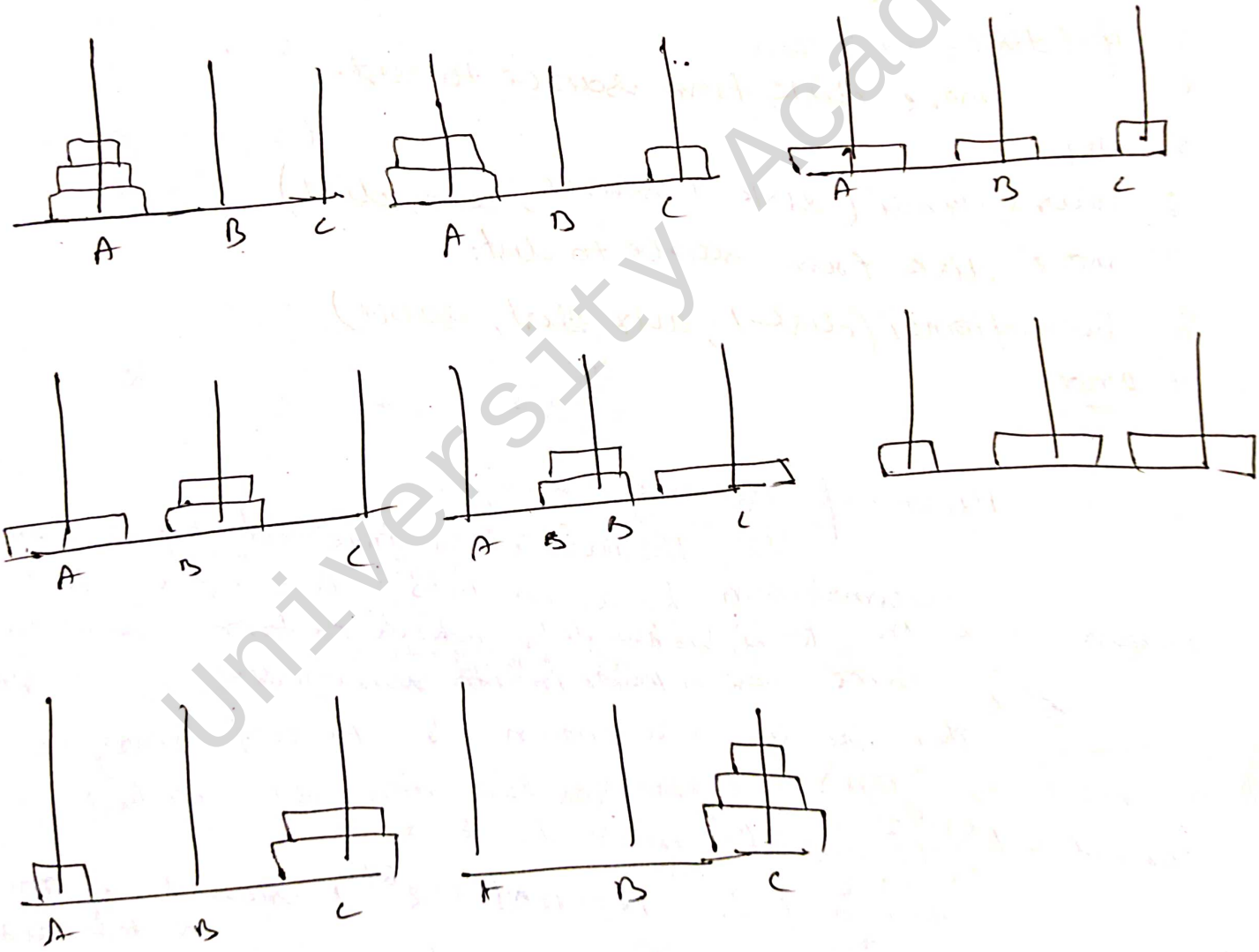
fib(5) = call 15
fib(4) = 9
fib(3) = 5

$O(2^n)$

Tower of Hanoi

Tower of Hanoi is a mathematical puzzle where we have three rods and n disk. The objective of the puzzle is to move the entire stack to another rod, by following rules.

1) One one disk can be moved at a time.
2. Each move consist of taking the upper disk from one of the rod and placing it on top of another rod. or an empty rod.
3) No disk may be placed on top of a smaller disk.



if n=1 A→C one move.
if n=2 A→B; A→C B→C 3 move.
if n=3 7 move.
if n=4 15 move.

for n disk there are $2^n - 1$ move

1- Move n-1 dist from A→B
2 -, move nth dist from A→C
3. Move n-1 dist from B→C

Algo TOWER (N, BEG, AUX, END)

1. if N = 1 then
   a) write BEG → END
   b) Return
2. [move N-1 disk from BEG to AUX]
   Call TOWER(N-1, BEG, EN

1 Start
2 Tower of Hanoi (disk, source dest, aux)
3 if (disk == 1) then
4 move disk from source to dest.
5 else
6 Tower of Hanoi (disk-1, source, aux, dist)
7 move disk from source to dist.
8 Tower of Hanoi (disk-1, aux, dest, sauce)
9 End.

History of Tower of Hanoi:

The puzzle was invented by the french mathematician Lucas in 1883, there is story about an indian temple in Kashi Vishwanath which contain a room with three post or tower in it. surrounded by 64 golden disk. this puzzle also known as tower of Brahma. according to legend, when the last move of the puzzle is completed the world will end.

number of move required: $2^{64} - 1$ second. if one move take one second equal to 585 billion's year.

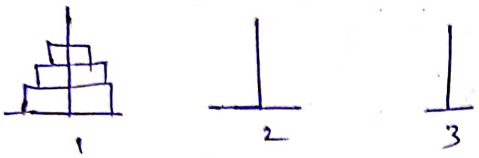
1 billion = 1 "अरब"

42 time the current age of universe.

Example- 3 disk

Procedure:

1. move top 2 disk from A → B using C
2. move disk from A → C
3. move 2 disk from B → C.

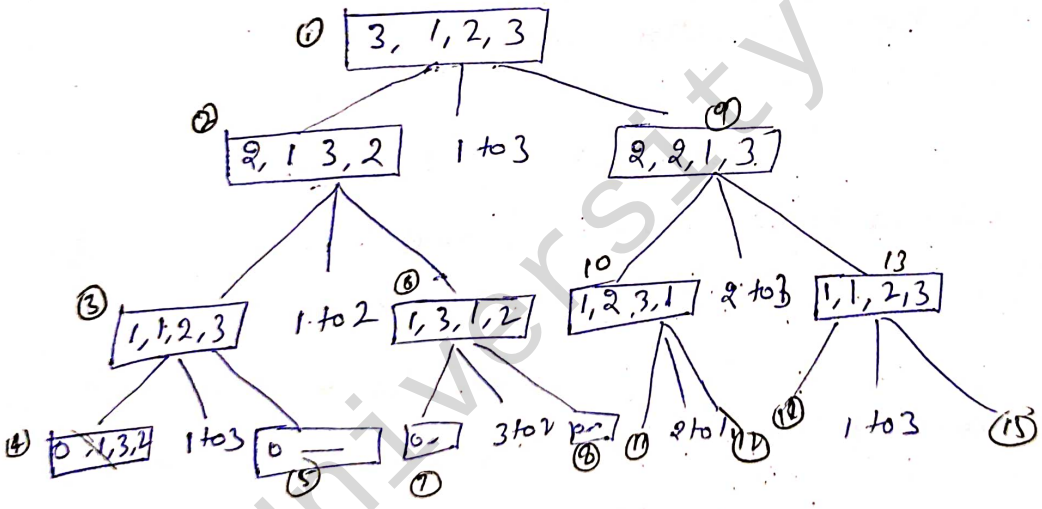


n = 3
A = 1, B = 2 C = 3

TOH (3, 1, 2, 3)
n A B C

Algo

```
void TOH (int n, int A, int B, int C)
          (from) (using) (to
{
  if (n > 0)
  {
    TOH(n-1, A, C, B)
    printf("from %d to %d ", A, C)
    TOH(n-1, B, A, C)
  }
}
```



(1, 3)
(1, 2)
(3, 2)
(1, 3)
(2, 1)
(2, 3)
(1, 3)

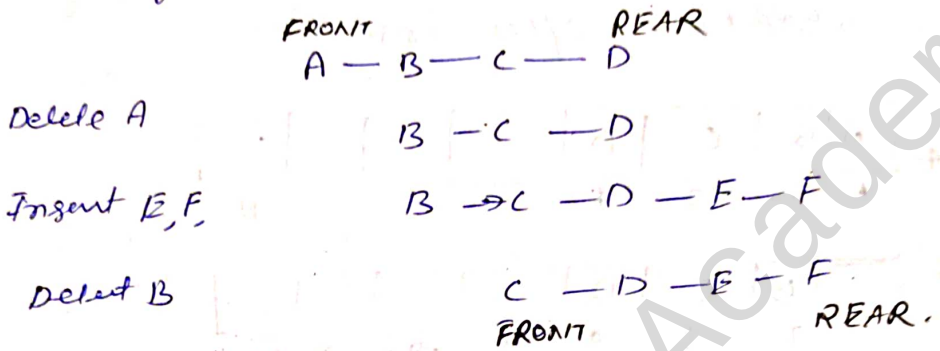
	No of calls	no of move
for n = 3	15 $(2^{3+1}-1)$	7 (2^3-1)
n = 2	7 $(2^{2+1}-1)$	3 (2^2-1)
for n	$(2^{n+1}-1)$	(2^n-1)

Queues-

A Queue is a linear list of element in which deletion can take place at one end called FRONT and insertion can take place at the other end called REAR.

Queue is also called First in First out (FIFO) list.

Eg: Queue at movie ticket counter.
waiting line at Bank ATM.



Basic features of Queue:

1. lik stack Queue is also ordered list of element of similar data type.
2. Queue is FIFO structure.
3. newly inserted element must be remove after the removing the element inserted before the new element.

Applications of Queue.

1. Sharing resource like printer, CPU task scheduling
2. Call center (Phone call)
3. Handling interrupts in real-time System.

Basic operations:

1. enqueue() -> add an element to queue by REAR
2. dequeue() -> remove an item from queue by FRONT.
3. Peek() -> element of FRONT of queue.
4. Is full() -> Que is full (overflow)
5. Is empty -> Queue is empty (underflow)

Array Representation of Queues:

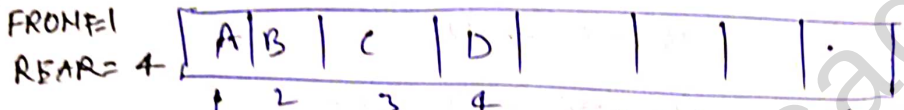
Queue will be maintained by a linear array by two two pointer FRONT : containing location of front element of queue and REAR : containing the location of rear element.

FRONT = NULL shows Queue is empty.

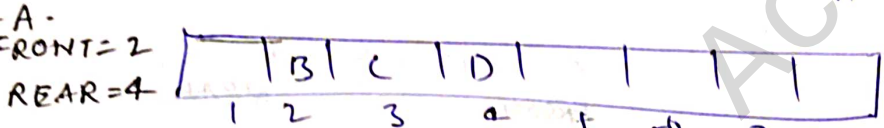
FRONT = FRONT + 1 after deletion.

REAR = REAR + 1 after Insertion.

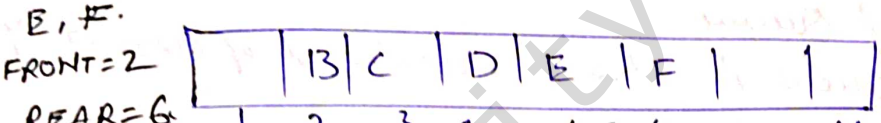
Eg:



delete A.



Insert E, F.



		1	2	3	4	5
(a) Empty	F: 0 R = 0					
(b) A, B, C inserted	F = 1 R = 3	A	B	C		
(c) A deleted	F = 2 R = 3		B	C		
(d) D, E inserted	F = 2 R = 5		B	C	D	E
(e) B, C deleted	F = 4 R = 5				D	E
(f) F inserted	F = 4 R = 1	F			D	E
(g) D deleted	F = 5 R = 1	F				E
(h) G and then H inserted	F = 5 R = 3	F	G	H		E
(i) E deleted	F = 1 R = 3	F	G	H		
F deleted	F = 2 R = 3		G	H		
K inserted	F = 2 R = 4		G	H	K	
G, H deleted	F = 4 R = 4				K	
K deleted	F = 0 R = 0					

INSERT (Q, N, F, R, Item)

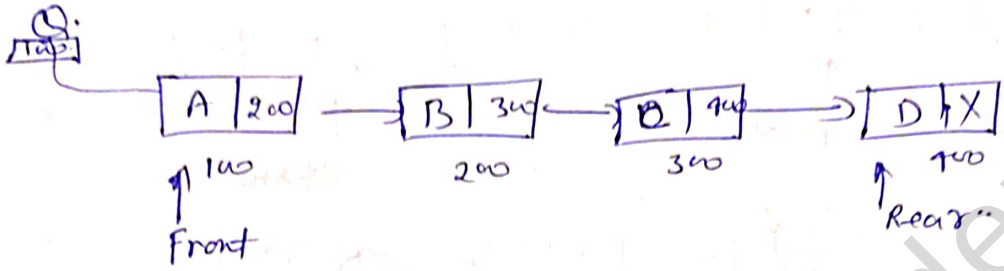
1. if F = 1 and R = N or F = R + 1 then overflow.
2. if F = NULL then:
   F = 1 ; R = 1
   else if R = N then:
   set R = 1 :
   else R = R + 1
3. Set Q[R] = Item :
4. Return.

Delete (Q, N, F, R, Item)

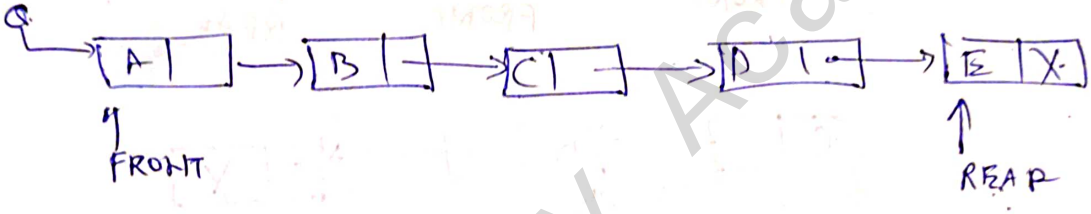
1. if F = NULL write underflow and Return
2. set Item = Q[F]
3. if F = R then, F = NULL, R = NULL
   else if F = N then set F = 1
   else set F = F + 1
4. Return

Linked list Representation of Queue:

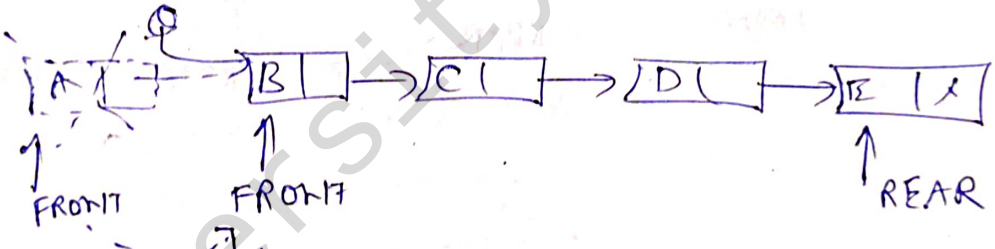
A linked Que is implemented using two pointer variable FRONT and REAR. Each node of linked list contains two part INFO: to hold element and LINK to hold address of neabhoring element in the Que.



Insert E



Deletion Q:



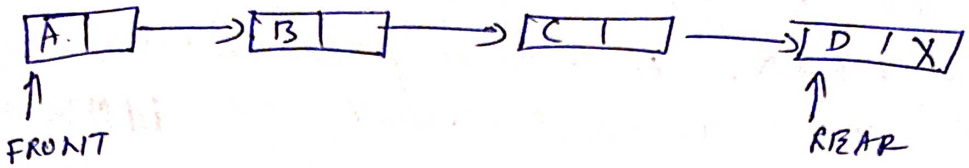
Insert Algo

1. Allocate space for new node "PTR"
2. set PTR → INFO := Item
3. if FRONT = NULL
   set FRONT = REAR = PTR
   set FRONT → LINK = REAR → LINK = NULL
   else
   set REAR → ~~NEXT~~ LINK = PTR
   set REAR = PTR
   set REAR → LINK → NULL.
4. end.

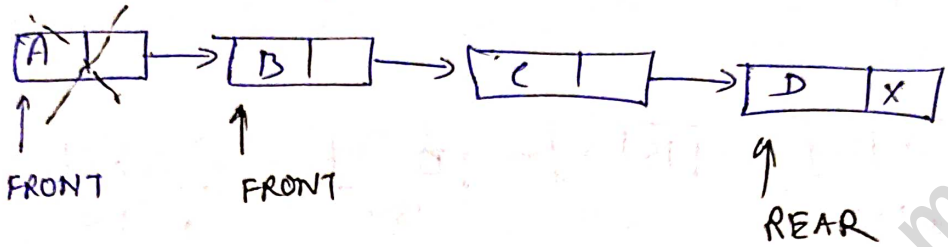
Deletion Algo:

1. if FRONT = NULL
   write underflow.
2. Set PTR = FRONT
3. Set FRONT = FRONT → LINK
4. FREE PTR
5. end.

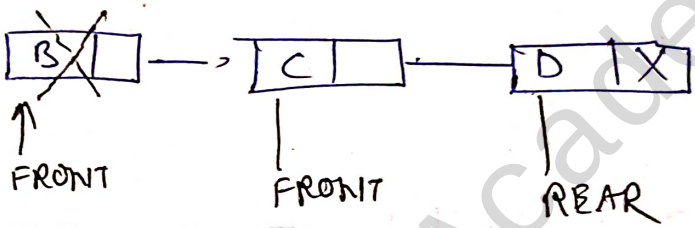
Examples Show the following operation in following linked Queue.
(i) Delete (ii) delete (iii) Insert E.



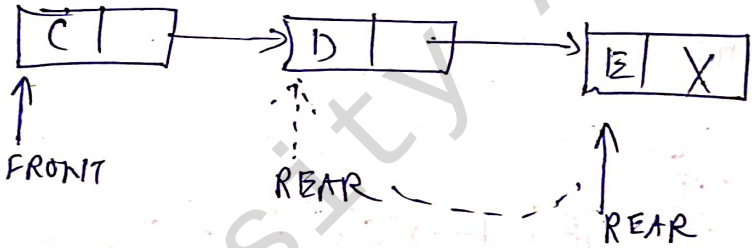
(i) Delete.



(ii) Delete.



(iii) Insert E

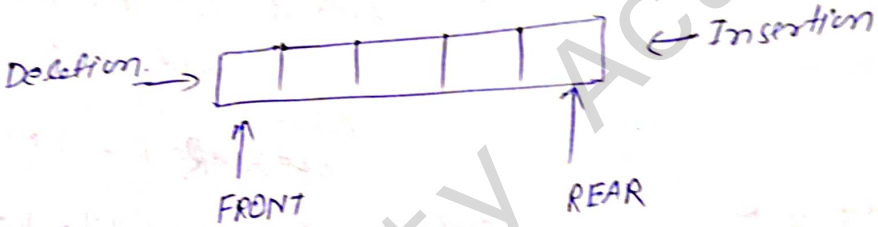


Types of Queue.

1. Simple Queue.
2. Circular Queue
3. Priority Queue.
4. Dequeue (Double Ended Queue)

1. Simple Queue:

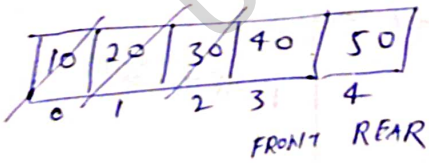
In simple Queue we can perform Insertion at REAR end of Queue. and deletion are performed at the FRONT (beginning) of the Queue list.



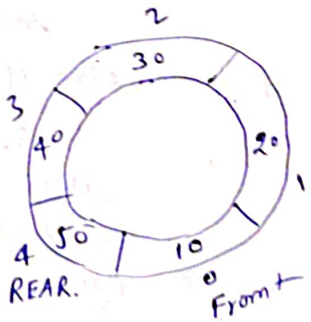
2. Circular Queue: Circular is a linear data structure in which last position is connected back to the first position to make circle. The main advantage of circular Queue is we can utilize the space. of queue fully.

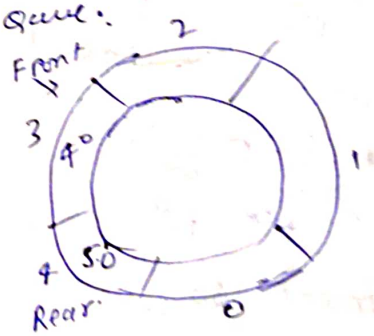
Initially: F = -1, R = -1

Simple Queue.



circular Queue.





1- Initialise the Rear pointer
2. shift the element.

Formulas for FRONT and Rear pointer

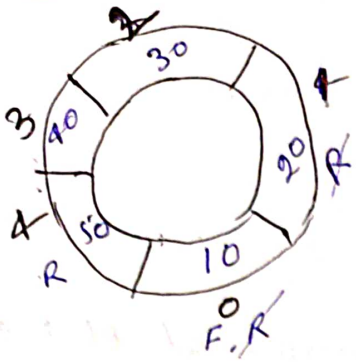
For circular Queue:

FRONT = (FRONT+1) mod size.
REAR = (REAR+1) mod size.

Insert 60: Rear = (Rear+1) mod size
= (4+1) mod 5 = 0
Insert 60 at 0 location.

FRONT = (3+1) mod 5 = 4 mod 5 = 4.
item deleted:

Example-



Insert 10

insert 20

set R = (0+1) mod 5
= 1

similarly
insert 30, 40, 50

insert 60

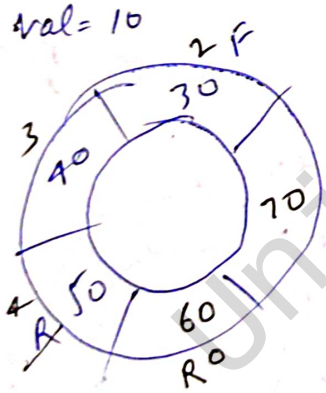
(4+1) mod 5 = F

5 mod 5 = 0 = F

overflow.

---

val = 10



insert 60.

insert 70

n = 5

Initially F = -1, R = -1 Empty.

~~Circular Queue full:~~

~~F = -1 Rear = n-1~~

Step 1 if (R+1)% n = F
write overflow.

step 2: if F = -1 R = -1
set F = R = 0
else if R = n-1 and F != 0
set R = 0
else
~~set R~~ = (R+1) % n

R = (R+1)%n

step 3 set Queue[R] = val.

step 4. exit.

Deletion

① if FRONT = -1
overflow.

2. set val = Queue[F]

③. if F = R
set F = R = -1
else if F = n-1
set F = 0
else
set F = F+1

F = (F+1) % n.

4- exit

Priority Queue:

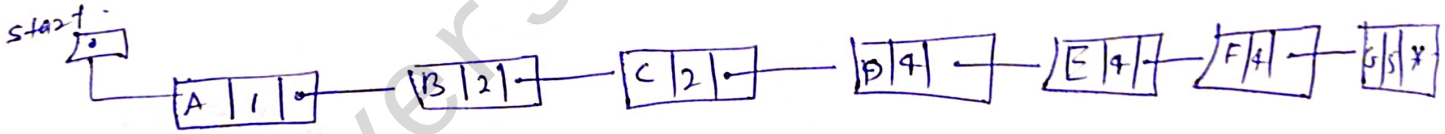
A priority Queue is a collection of element such that each element has been assigned a priority .. and ~~such that the order in which element are deleted and processed comes from following rule~~

1. ~~A element of higher priority is processed before lower priority.~~
2. and serve according to its priority. if element with the same priority occur they are served according to their order the ques.

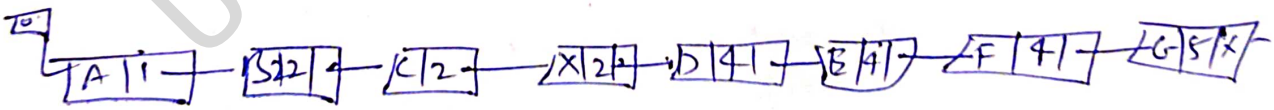
there are various ways to represent priority Que. but we discuss two of them.

1. linked list (one way list)
2. Array (multiple ques).

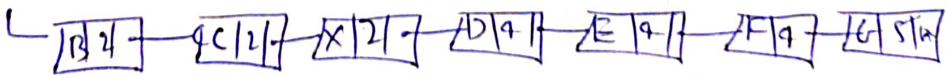
1- One way list Representation:



insert [X] with priority 2



delete:

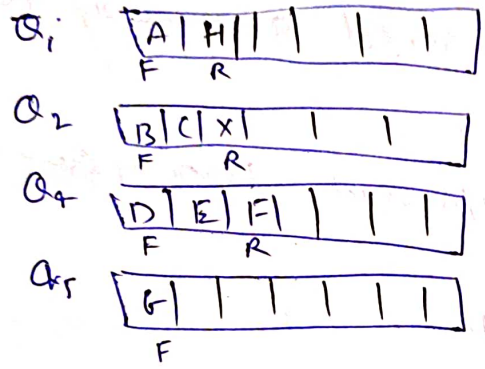


similarly delete B then C then X. and so on.

Array Representation of Priority Queue:

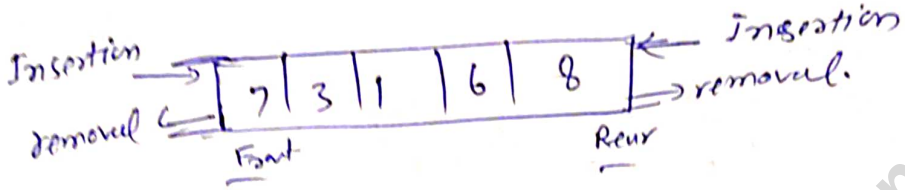
Another way to maintain a priority queue in memory is to use a seperate queue for each priority. each queue in its circular array and must have pointer FRONT and REAR.

A	B	C	X	D	E	F	G	H
1	2	2	2	4	4	4	5	1



DEQUE

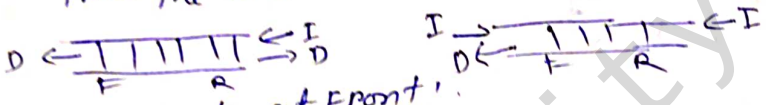
Deque or Double-Ended queue is a type of Queue. in which insertion and removal of elements can be perform from either from front to rear it does not follow FIFO Rule.



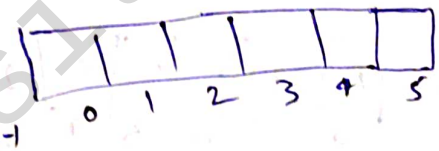
there are two type of Deque

1. Input Restricted Deque: ~~input~~ insertion at only one end "Rear" but allow Deletion from Both end.

2. output Restricted Deque: Allows deletion at only one end of "FRONT" the list but allows insertion at both ends of the list.

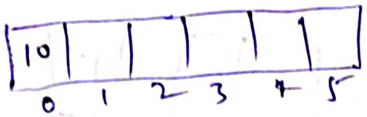


Insert Element at Front:



FRONT=-1
REAR=-1

Insert=>10



F=0
R=0

Inset 12 at Front.

Insert Front
IF FRONT=-1
set F=0 R=0
else IF FRONT=0
set FRONT=size-1
else
FRONT=FRONT-1
Rear
DEQUE[FRONT]=ITEM

Insert Rear
if ~~FRONT~~=-1
set F=0 R=0
else if Rear=size-1
set Rear=0
else
Rear=rear+1
DEQUE[REAR]=ITEM

Delet FRONT
if Front=Rear
set F=-1 R=-1.
else if
(F=size-1)
F=0
else
F=F+1

Delete Rear
if F==R
set F=-1 R=-1
else if (R==0)
R=size-1
else
R=R-1

- Insert Front
- Insert Rear
- Delete Front
- Delete Rear

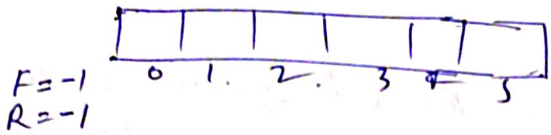
] is full / is empty

- FRONT=0 && REAR==size-1 or FRONT=Rear+1
- FRONT==-1

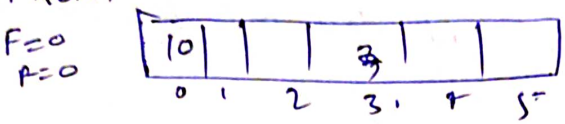
Size = 6

Example

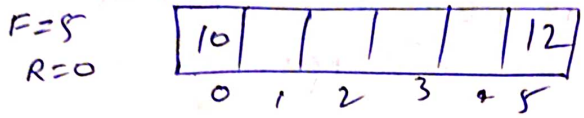
initially



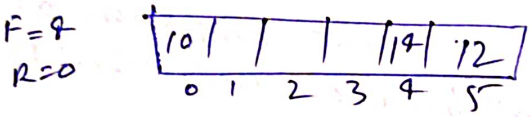
insert in FRONT.



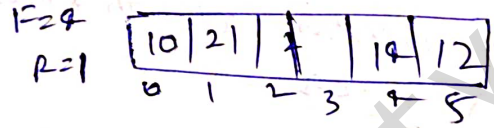
insert 12 at FRONT.



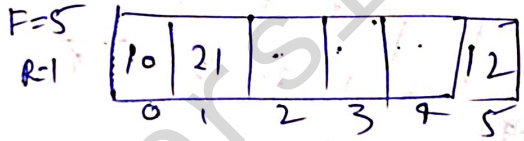
insert 14 at FRONT



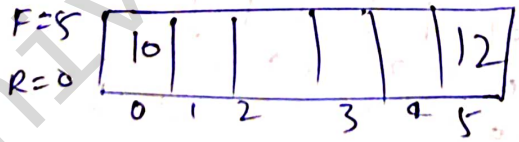
Insert 21 at REAR.



Delete FROM FRONT.



Delete From REAR

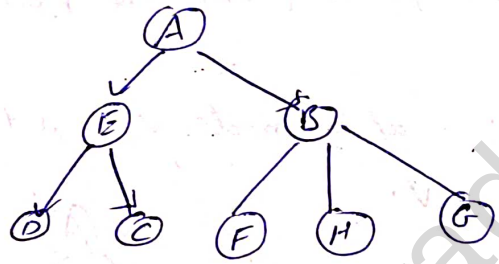


UNIT – III

TREE :- Tree is a non-linear data structure. this is used to represent ~~to~~ hierarchical Relationship between elements.

A tree is a finite set of nodes together with a finite set of directed edges that define parent-child Relationship. and there are no any circuit.

e.g:



Node :- A, B, C, D, E, F G H

Edges: (A,B), (A,E), (E D), (E C)
(B,F) (B,H), (B,G)

Properties of Tree :

(a) It has one designated node called Root, that has no parent.

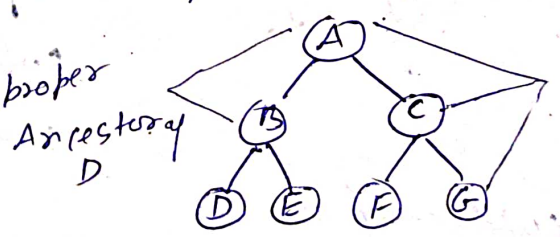
(b) Every node, except root, has exactly one parent.

(c) A node may have zero or more children.

(d) There is a unique path from root to each node.

Tree Terminology :-

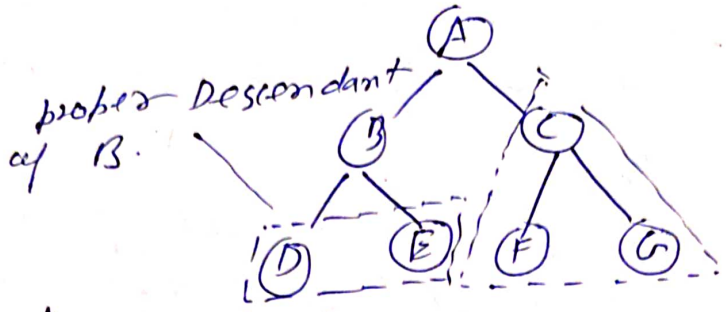
(i) Ordered tree :- A tree in which the children of each node are linearly ordered (left to right).



(ii) Ancestor :- Ancestor of a node V: any node including V itself on the path from root to the node.

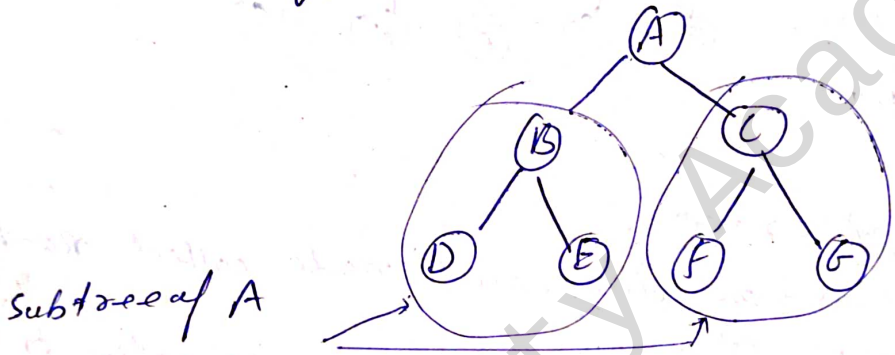
(iii) Proper Ancestor : of a node V: any node, excluding V, on the path from root to the node.

(IV) Descendant : Descendant of node v, any including v itself, on any path from the node to a leaf node.



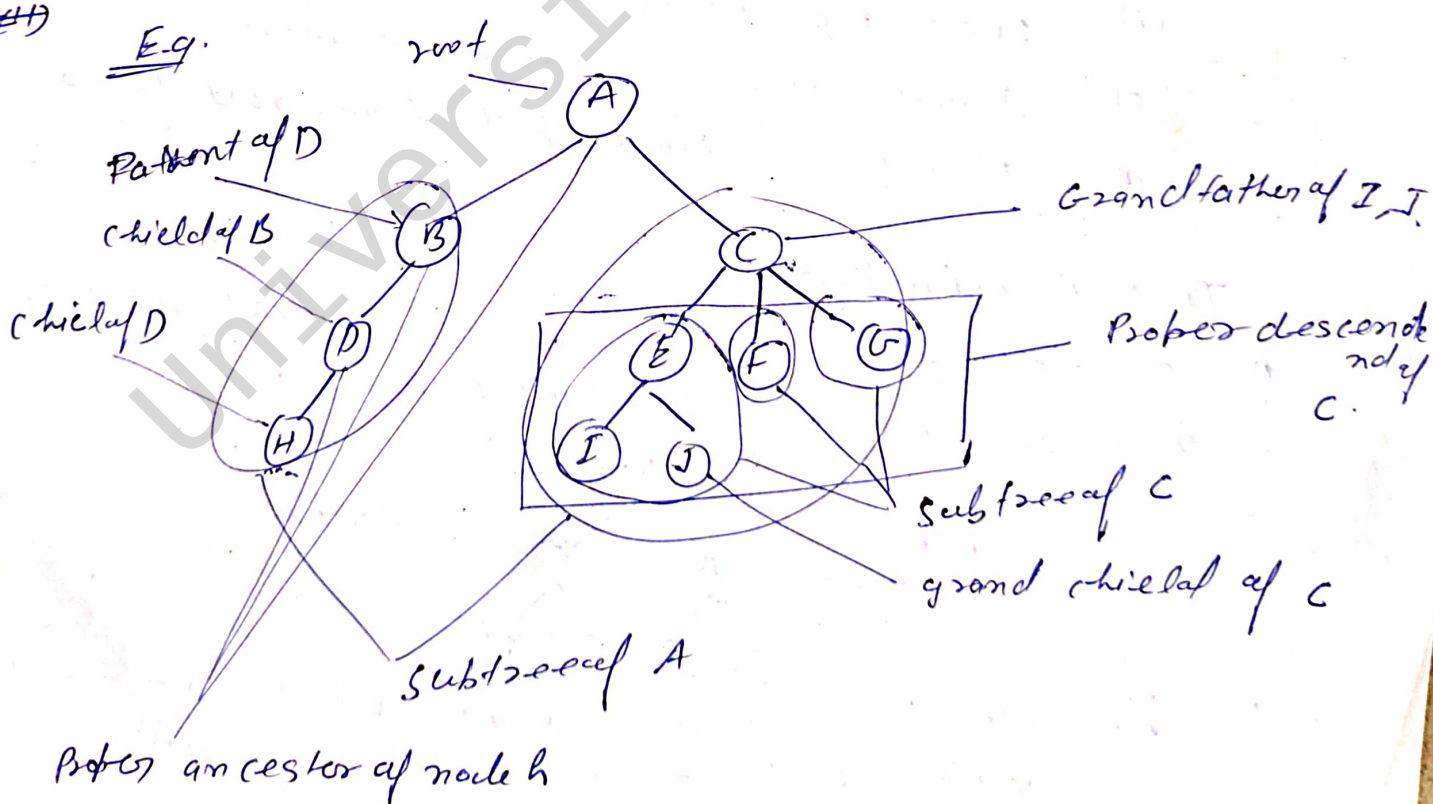
(V) Proper Descendant : of a node v, any node, excluding v, on any path from the node to leaf node.

(VI) Subtree : of a node v, A tree rooted at a child of v.



~~(VII)~~

E.g.



(viii) Leaf node: terminal node is leaf node.

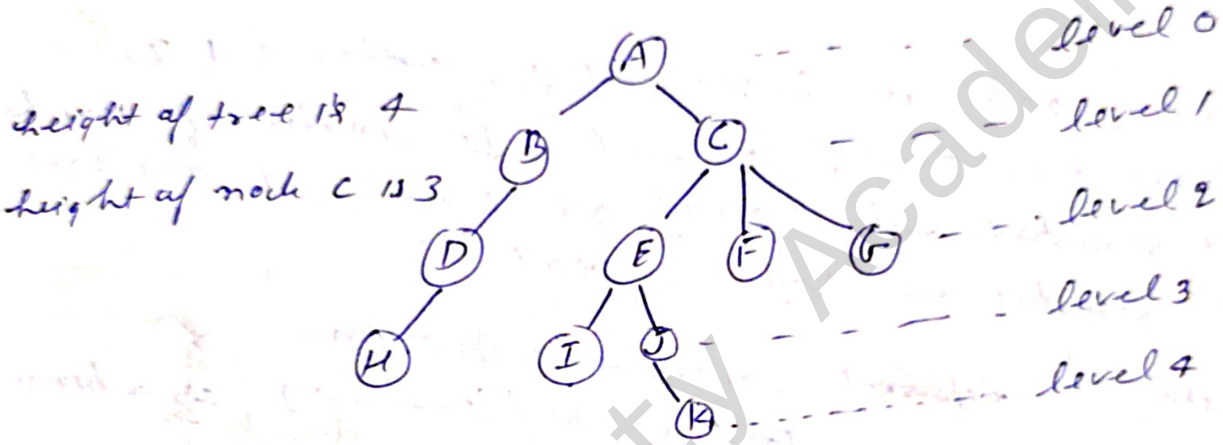
(viii) Internal node: except leaf node.

(ix) Sibling: node that have same parent.

(x) Size: Number of Node in the tree.

(xi) Level (depth) – length of path from the root to node.

(xii) Height → the length of longest path from ~~Root~~ node to leaf node.



BINARY TREE

A binary tree is defined as a finite set of element (node) such that

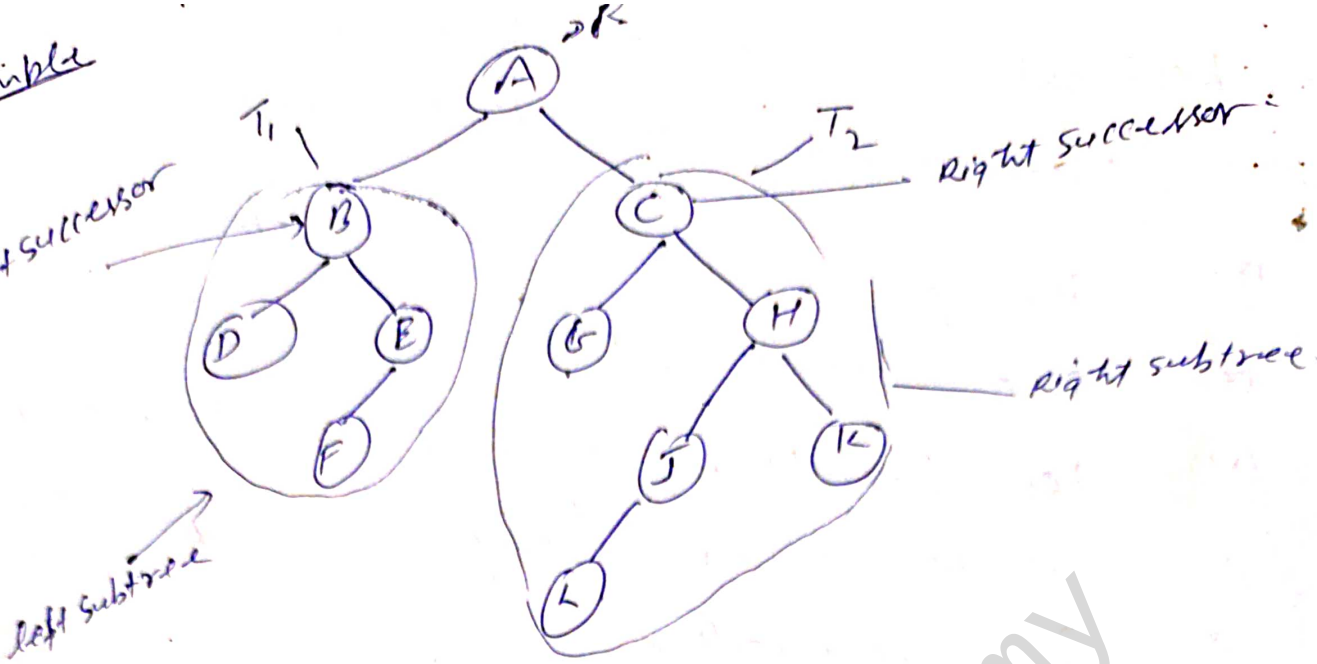
(1) Tree is empty (null tree, empty tree)

(2) Tree contains root node and at most two non-empty binary subtree.

If Binary tree T contain a root R the two subtree T_1 and T_2 are called respectively left and right subtree of R.

If T_1 & T_2 is nonempty ∧ then its root of T_1 is called left successor of R similarly if T_2 is nonempty then its root is called right successor of R.

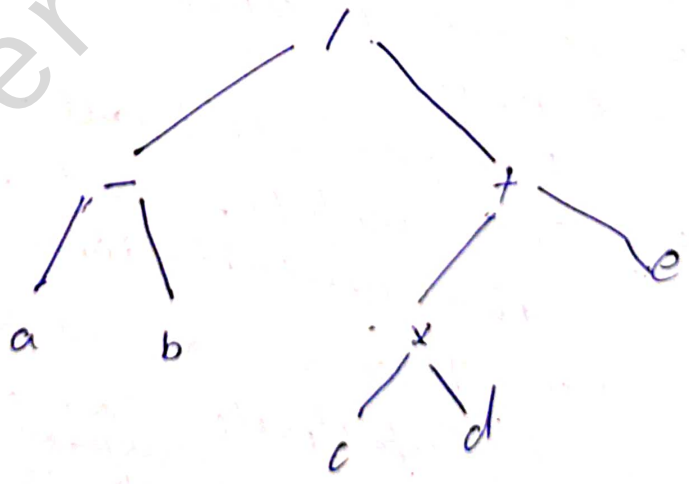
Example



Note → any node in binary tree has either 0, 1, or 2 successor.

Binary tree T and T' are said to be similar if they have same structure and T and T' are copied if they have same structure and same content.

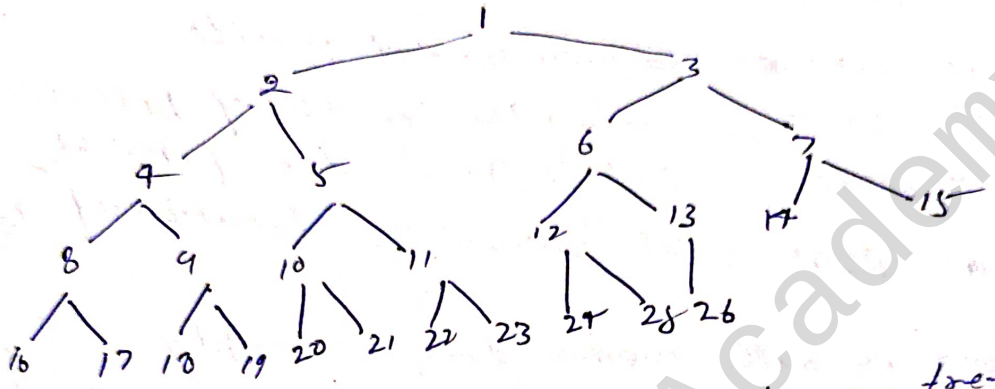
Question Represent Algebraic Expression (a-b) / ((c*d) + e) as a binary tree.



Complete Binary tree:

The tree T is said to be complete if all its levels except possibly the last, have maximum number of possible node. the each level have at most 2^r node. where r is level.

the complete binary tree T_{26} with 26 node is.



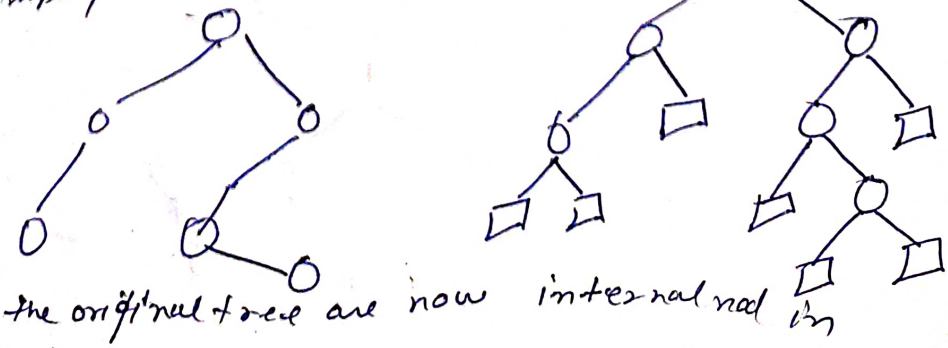
the depth d_n of complete binary tree T_n with n nodes is given $D_n = \lfloor \log_2 n + 1 \rfloor$

Extended Binary tree: (2-Tree)

A binary tree is said to be a 2-tree or extended binary tree if each node N has either 0, or 2 children.

In such case the nodes with 2 children are called internal node and nodes with '0' children called external nodes.

The Tree may be converted into 2-Tree by replacing each empty subtree by new node.



Note: we observe that the original tree are now internal node in extended binary tree.

Representation of Binary tree in Memory

1- Link Representation of Binary tree :

2- Sequential Representation of Binary tree :

1- LINK REPRESENTATION OF BINARY TREE :-

Binary tree will be maintained in memory by linked representation which uses three parallel array. INFO, LEFT, RIGHT, and a pointer variable ROOT.

(1) INFO[K] Contain the data at node.

(2) LEFT Contain the location of left child :

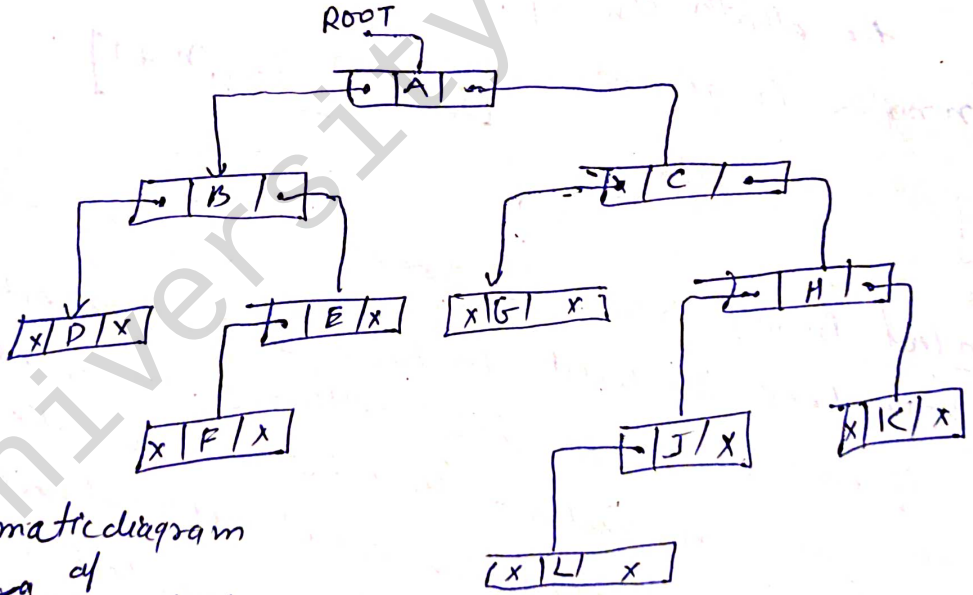
(3) RIGHT Contain the location of Right child.

(4) ROOT will contain the location of root of tree

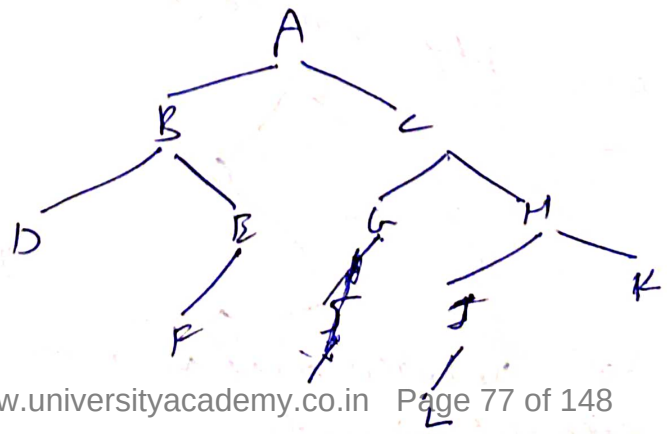
if ROOT is null means tree is empty.

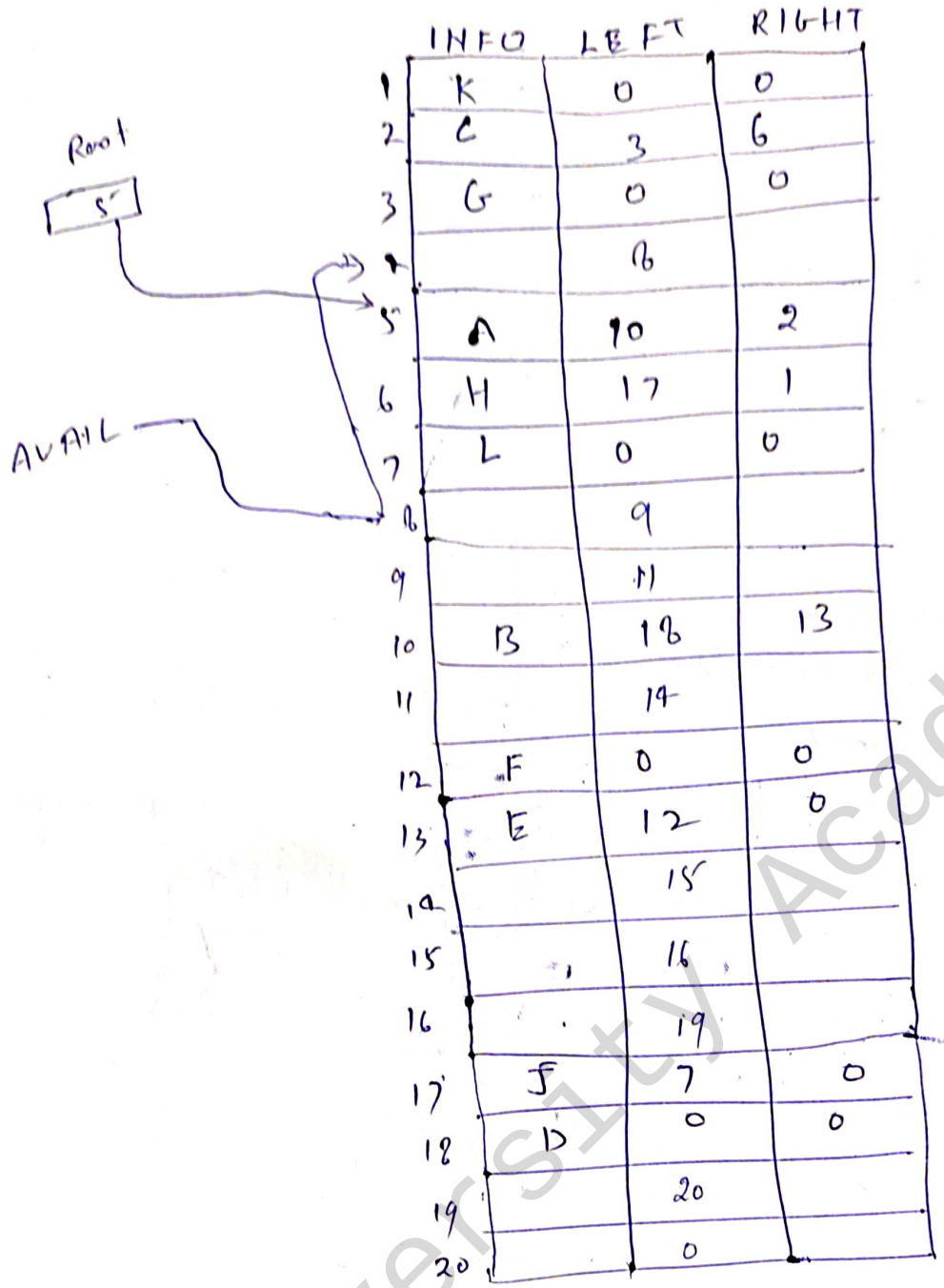
if any subtree is empty then corresponding pointer contain null

Example:



Is a schematic diagram of link Representation of following tree.





SEQUENTIAL REPRESENTATION OF BINARY TREE:

Suppose T is a Binary tree that is complete or nearly complete. then there is an efficient way to maintaining in memory called sequential representation.

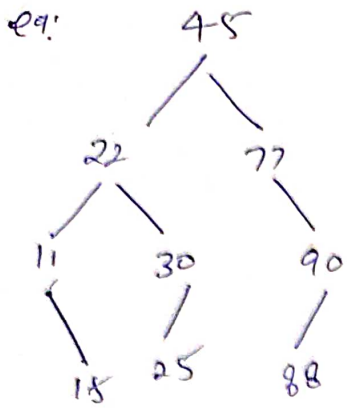
This Representation uses only a single linear array Tree as follow.

(a) The root of tree T is stored at TREE[1]

(b) if a node N occupies TREE[K], then its left child is stored in Tree[2*K] and its Right child is stored in TREE[2*K+1]

Note: if TREE[1] = NULL then tree is empty.

Eg:



TREE

1	45
2	22
3	77
4	11
5	30
6	.
7	90
8	.
9	15
10	25
11	
12	
13	
14	88
15	
16	
17	
18	
19	
20	

A tree with depth d will require an array with approx 2^{d+1} element

Operation on Binary tree.

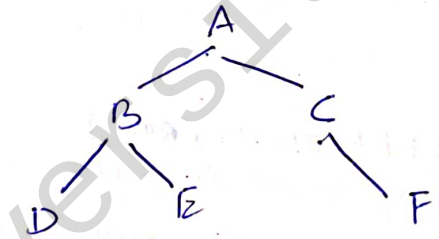
1- Traversing Binary tree
2. Searching ~~Binary~~ tree.
3- Insertion & Deletion on Binary tree.
4 Seach Tree :- BST (Binary Search tree)
- AVL trees.

1- TRaversing Binary tree:

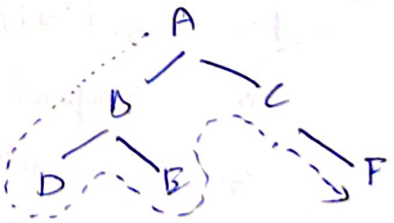
there are three standard way of traversing a binary tree.

1. Preorder	2- Inorder	3. Postorder,
1. Process the root R	1- Traverse the left Subtree of R Inorder	1. traverse the left subtree of R in postorder
2- traverse the Left Subtree of R in Preorder	2 - process the root R	2- traverse the Right subtree of R in postorder.
3- Traverse the Right Subtree of R in Preorder	3- Traverse the ~~Right~~ ~~left~~ Subtree of R inorder	3. Process the root R.

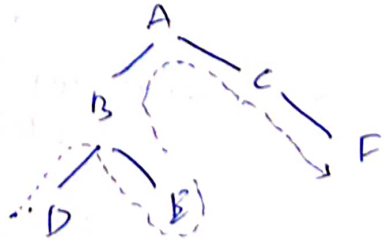
eg:



a- Preorder traversal: A B D E C F
↳ Node-left-Right (NLR)

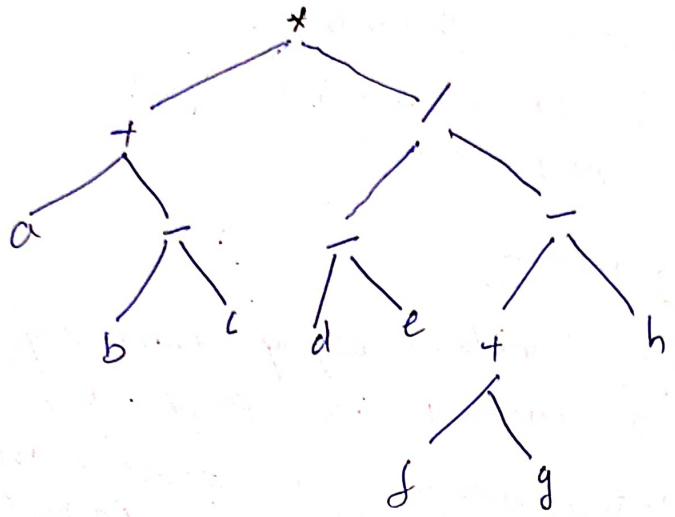


(b) inorder traversal:
LNR (Left-node-Right) D B E A C F



C- Post order traversal
LRN - Left Right-node D E B F C A

Example: Algebric equation: $[a+(b-c)] * [(d-e) / (f+g-h)]$



preorder (prefix) ⇒ * + a - b c / - d e - + f g h

inorder (infix) ⇒ a + b - c * d - e / f + g - h

postorder (postfix) = a b c - + d e - f g + h - / *

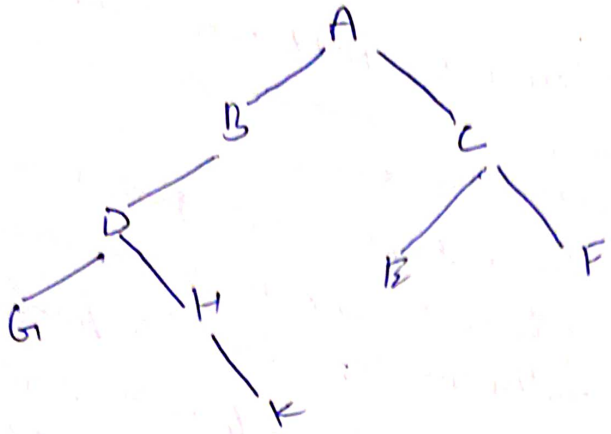
TRAVERSAL ALGORITHM USING STACKS

1- Preorder-

Algorithm PREORD(INFO, LEFT, RIGHT, ROOT)
An array STACK is used to temporarily hold the address of node.

1- Set TOP = 1, STACK[1] = NULL and PTR = ROOT
2- Repeat step 3 to 5 while PTR ≠ NULL
3- Apply process to PTR. INFO[PTR].
4- [RIGHT child]
IF RIGHT[PTR] ≠ NULL then [Push on STACK]
set TOP = TOP + 1 and STACK[TOP] = RIGHT[PTR]
5- [Left child]
if LEFT[PTR] ≠ NULL then
set PTR = LEFT[PTR]
else [POP from stack]
set PTR = STACK[TOP] and TOP = TOP - 1
[end of step 2 Loop]
6- exit

Example:



1- initially push NULL onto stack

STACK: ∅

PTR = A Root of Tree:

TOP: 1

2- proceed left most path rooted at PTR := A

(i) Process A and push it's Right child to Stack: ∅ C.

(ii) Process B: No Right child.

(iii) Process D: Push it Right child to stack

stack: ∅ C H

(iii) Proceed G: No right child.

→ No other process since there are no any children left to be

(2) Backtrack: pop the top element of stack: H.

stack: ∅ C

(v) process H and put it right child to stack

∅ C. K

Backtrack: pop K

∅ C.

process K there are no Rig child.

there are no then Left child.

pop C

→ ∅.

Process C and push it Right child to stack

∅ K

process E. there is no right child.

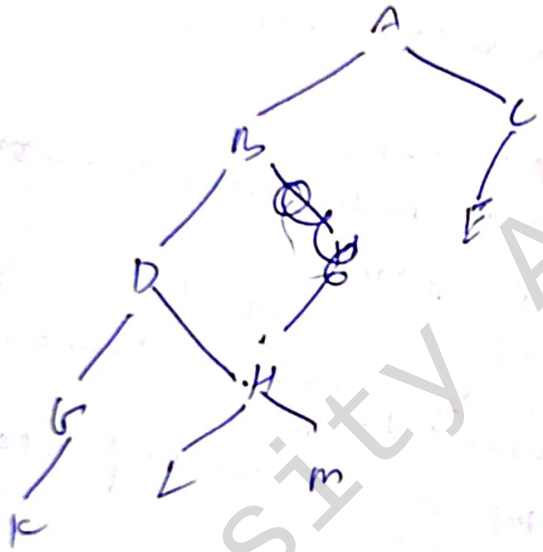
pop F

process ∅ there no left and Right child.

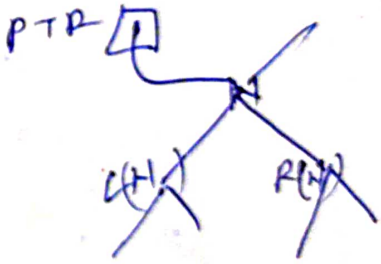
Inorder Initially Push NULL onto STACK and set PTR := ROOT. Repeat following steps until NULL is popped from STACK.

(a) Proceed down left-most path rooted at PTR. Pushing each node N onto STACK and stopping when a node N with no left child is pushed onto stack

(b) Pop and process the node onto STACK : if NULL is popped the exit. if node N with a right child is R(N) is processed, set PTR=(RN) and Return to step (a).



K G D L H M B A E C



STACK : ϕ A B D G K

K G D popped and process

D has right-

stack ϕ A B H L

L and H become.

ϕ A B

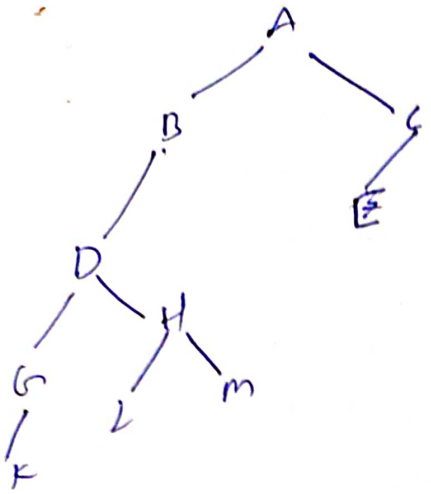
ϕ A B M

M pram M B A also ϕ

ϕ C E

E L popped

INORDER TRAVERSAL



1- STACK: ϕ, PTR: A TOP=1

2- STACK: ϕ A B D G K

no other is pushed onto STACK since K has no left child.

3. the node K, G, D are popped and processed now

STACK: ϕ A B

we stop the processing at D, since D has a right child. then set

PTR := H

STACK ϕ A B H L

(No other node is pushed onto STACK since L has no left child,

Pop L, H and processing.

H has Right child since

STACK ϕ A B m

Pop m; B, A.

Stop A since A has Right child.

STACK ϕ C E

pop E C .

INORD (INFO, LEFT, RIGHT, ROOT)

1-[PUSH NULL onto STACK and Initialize PTR]
Set, TOP=1, STACK[1]:=NULL, PTR:= ROOT

2. Repeat while PTR ≠ NULL [Push Left most path onto stack]

(a) Set TOP: TOP+1 and STACK[TOP] = PTR.

(b) Set PTR = LEFT [PTR]

3. Set PTR := STACK[TOP] and TOP = TOP-1 [Pop node from STACK]

4. Repeat 5 to 7 while PTR ≠ NULL [Backtrack]

5. Apply PROCESS to INFO[PTR]

6 [Right child] if RIGHT[PTR] ≠ NULL

(a) Set PTR := RIGHT[PTR]

(b) goto step 2

7- Set PTR = STACK[TOP] and

TOP = TOP - 1

8. Exit.

Steps

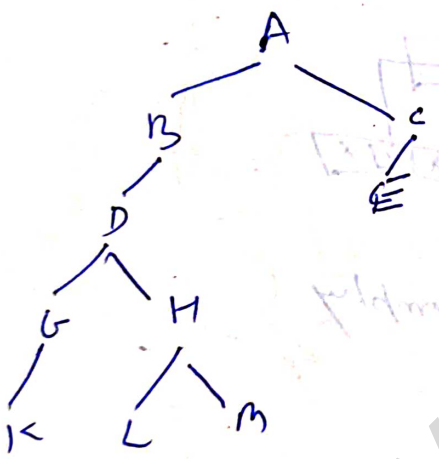
Initially push NULL onto stack and then set PTR := ROOT. Repeat following.

POST ORDER

Algorithm Initially push NULL onto STACK and set PTR=ROOT then Repeat following steps until NULL is poped from STACK

(a) Proceed down left most path Rooted at PTR. At each node N of the path push N onto STACK and if N has a Right child R(N) then push -R(N) on to stack.

(b) Pop and process positive node on STACK. if NULL is popped then exit. if a negative node is popped that is if PTR = -N for some node N, set PTR=N, set PTR=N and return to step(a).



K G L M H D B E C A

1- Proceed down left-most path Rooted at PTR=A, and right child push -R(H) onto stack

STACK: ~~$, A B C~~
$ A, -C, B, D, -H, G, K,

Pop K and G and process.

Now PTR = -H ~~Reset~~ PTR=H

STACK $ A, -C, B, D, H

Now Push M and L

STACK $ A, -C, B, D, H, -M, L

Pop L and process.

STACK: $ A, -C, B, D, H, ~~M~~,

Pop M and process.

$ A, -C, B, D, H

Pop H, D B and proce

stack $ A, C

Push E

stack $ A C E

pop E C A : any

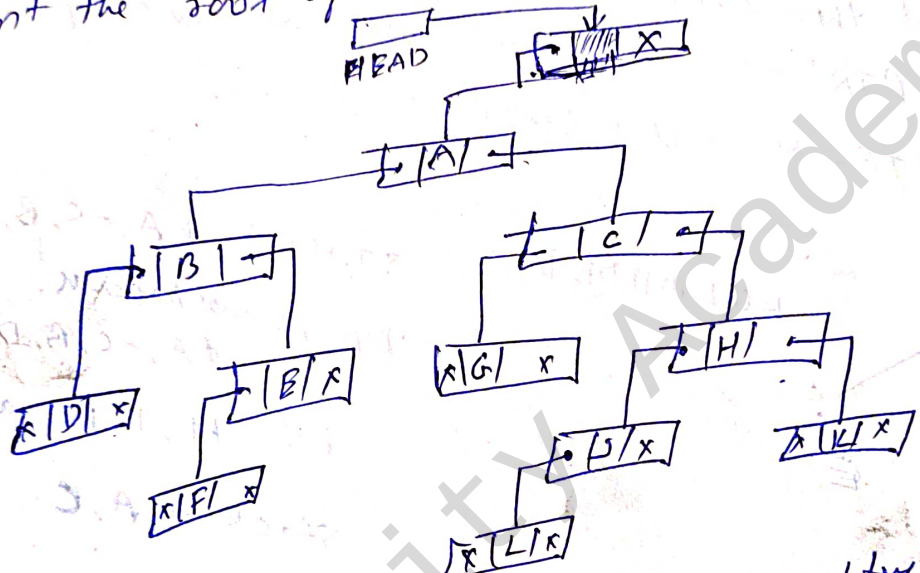
Retu

Header Nodes & Treads:

Header Nodes:

Suppose a binary tree in memory by means of link representation. some time an extra node called header node is added to the beginning of T. When the extra node is used the tree pointer variable which we will call HEAD will be point to header node and left pointer of header node point the root of tree.

eg:



if LEFT[HEAD] = NULL, tree is empty

Threaded Binary tree:

Consider again the link representation of binary tree. Approximately half of the entries in the field LEFT and RIGHT will contain null element. this space may be more efficiently used by replacing the null entries by some other type of Information.

we will Replace certain nul entries by special pointers which point to node higher in the tree. this pointer is called threads and binary tree is called treaded binary tree.

there are many ways to ~~tread~~ thread a binary tree.

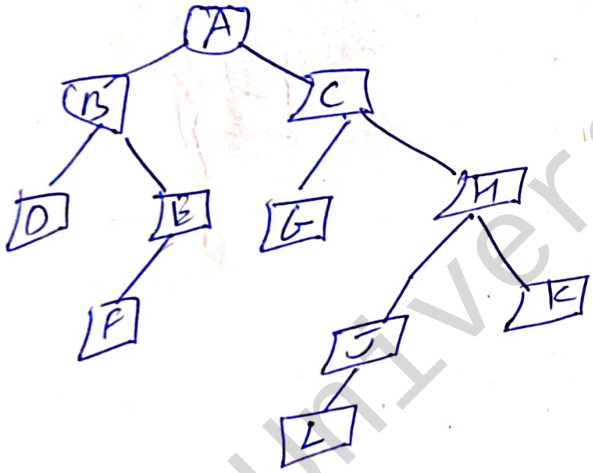
A binary tree is treaded by making all Right child pointer that would normally be null point to the inorder successor of the node. and all the left child pointer that would normally be null pointer to the inorder predecessor of the node.

In computation a treaded binary tree is a binary tree variant that allows fast traversal.

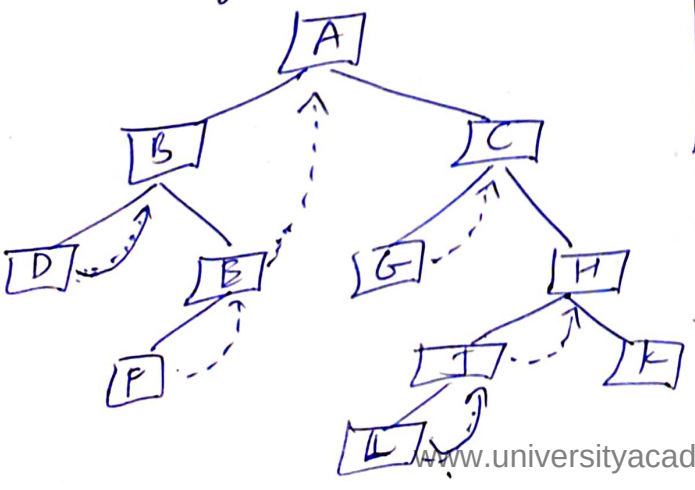
there are many ways to thread a binary tree.

1. one way threading
2. two way threading

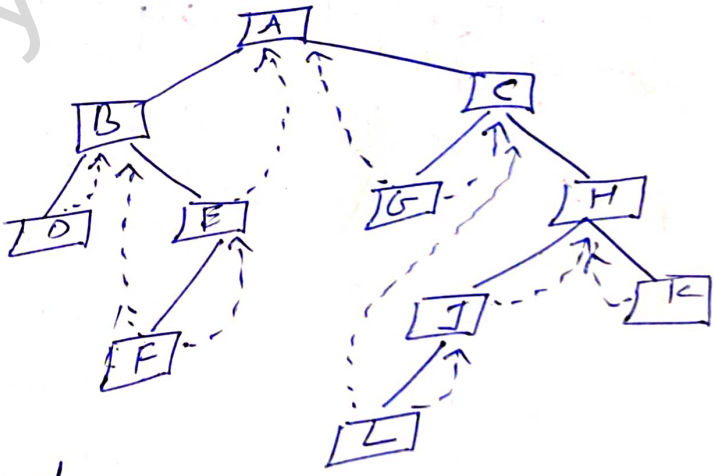
Consider a tree:

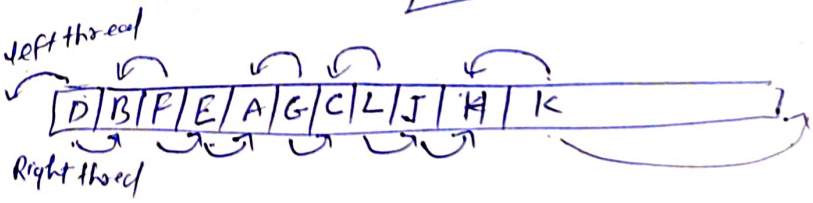


(a) one-way inorder threading

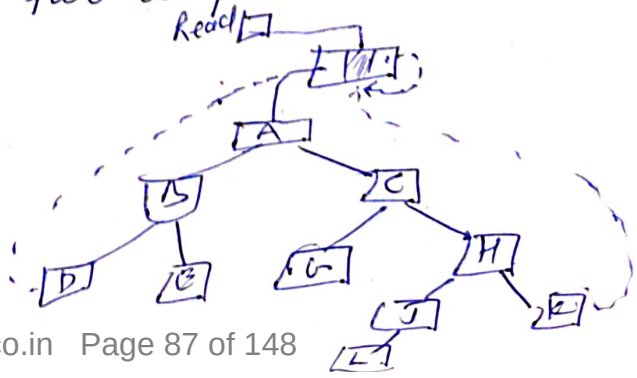


(b) two way Inorder threading

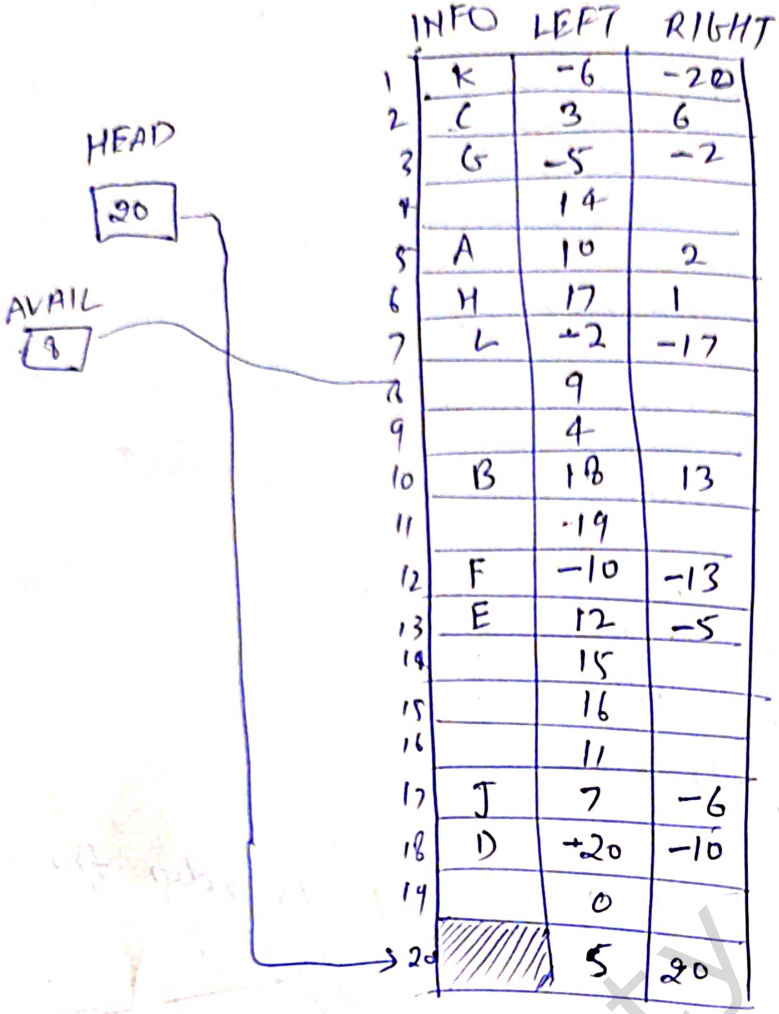




(c) two way threading with Header node



Memory Representation of threaded binary tree:



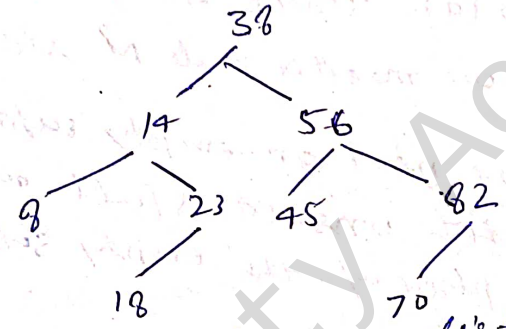
Binary Search tree:

Suppose T is binary tree then T is called Binary search tree if each node N of T has following properties.

(1) the value at N is greater than every value in the left subtree and N is less than every value in the right subtree of N.

Note: this's properties guarantees that the inorder traversal of T will yield a ~~stored~~ sorted listing of the elements of T.

eg.



here all the node values are distinct.

In the case of duplicates, each node has the following properties:

The value at N is greater than every value in left subtree of N and is less than or equal to every value in the right subtree of N.

SEARCHING AND INSERTING IN BINARY SEARCH TREE

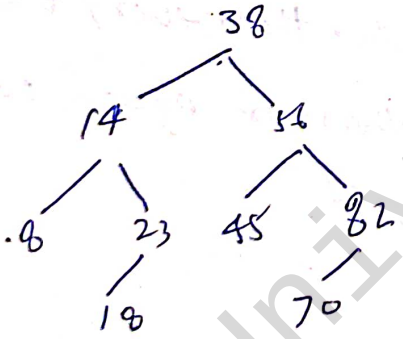
the searching and inserting will be given by a single search and insertion algorithm.

Suppose a ITEM of information is given. the following steps to find the location of ITEM in binary search tree or, or insert ITEM as a new node in its appropriate place in the tree.

(a) Compare ITEM with the root node N of the tree
   (i) if ITEM < N proceed to the left child of N
   (ii) if ITEM > N proceed to the Right child of N.

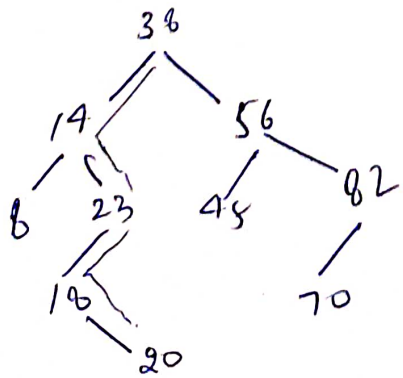
(b) Repeat step (a) until one of following occurs
   (i) we meet a node N such that ITEM = N.
   (ii) we meet an empty subtree, which indicate search unsuccessfull and we insert ITEM in place of the empty subtree.

Consider the tree

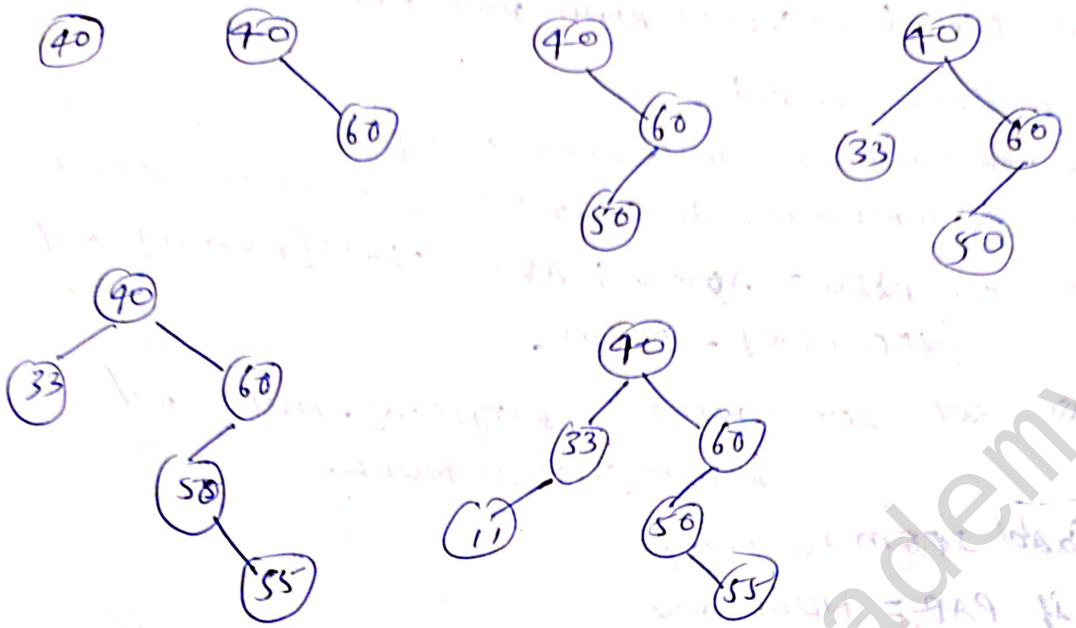


we want to search or Insert ITEM = 20 is given.

1- Compare ITEM = 20 with Root 38 of tree
   20 < 38 proceed to left child.
2- Compare 20 with 14, 20 > 14 proceed to right child.
3- Compare it with 23, 20 < 23 proceed to left.
4- Compare to 18, 20 > 18, since 18 does not have right child Insert 20 as the right child of 18.



Question Insert following six number in order into an empty binary search tree: 40, 60, 50, 33, 55, 11



Procedure:

FIND (INFO, LEFT, RIGHT, ROOT, ITEM, LOC, PAR)

(i) LOC = NULL and PAR = NULL tree is empty

(ii) LOC ≠ NULL and PAR = NULL ITEM is Root.

(iii) LOC = NULL and PAR ≠ NULL will indicate ITEM is not in T.

1) [Tree is empty]
IF ROOT = NULL then set LOC := NULL and PAR = NULL and Return.

[2] [ITEM at ROOT]
if ITEM = INFO[ROOT], then set LOC := ROOT and PAR := NULL and Return.

3. [initialize PTR and SAVE]
if ITEM < INFO[ROOT] then:
set PTR = LEFT[ROOT] and SAVE = ROOT.
else set PTR = RIGHT[ROOT] and SAVE = ROOT

4. Repeat step 5 and 6 while PTR ≠ NULL

5. [ITEM found]
if ITEM = INFO[PTR] the set LOC = PTR and PAR = SAVE and Return

6 if ITEM < INFO[PTR] then:
Set SAVE := PTR and PTR := LEFT[PTR]
else SAVE = PTR and PTR = RIGHT[PTR]
[End of step 4 Loop]

7. [Search unsuccessful] Set LOC := NULL and PAR := SAVE

8. exit.

Algorithm 7.5:

INSBST (INFO, LEFT, RIGHT, ROOT, AVAIL, ITEM, LOC)

1- Call FIND (INFO, LEFT, RIGHT, ROOT, ITEM, LOC, PAR)

2. if LOC ≠ NULL then exit.

3- [Copy ITEM into new node AVAIL LIST]

(a) if AVAIL = NULL then write: OVERFLOW and exit.

(b) set NEW = AVAIL ; AVAIL = LEFT[AVAIL] and INFO[NEW] = ITEM.

(c) set LOC = NEW; LEFT[NEW] = NULL and RIGHT[NEW] = NULL.

4- [Add ITEM to Tree]

if PAR = NULL then
    ROOT = NEW.
else if ITEM < INFO[PAR] then
    set LEFT[PAR] = new
else
    set RIGHT[PAR] = New

(5) exit.

Complexity of BST:

the average running time $f(n)$ to search for an element in a binary tree T with n element is proportional to $\log_2 n$ that $f(n) = O(\log_2 n)$.

Deleting from a Binary Search tree!

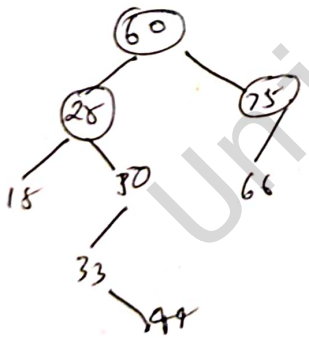
The deletion algorithm first to find the location of node N which contain the ITEM and also location of parent node P(N). The way N is deleted from the tree depends primarily on the number of children of node N.

Case 1 - N has no children. Then N is deleted from Tree by simply replacing the location of N in the parent node P(N) by a null pointer.

Case 2. N has exactly one child. then N is deleted from T by simply replacing the location of N in P(N) by the location of only child of N.

Case 3: N has two children, let S(N) denote the inorder successor of N. then N is deleted from Tree by first deleting S(N) from tree and then replacing node N in tree by the node S(N).

Example!

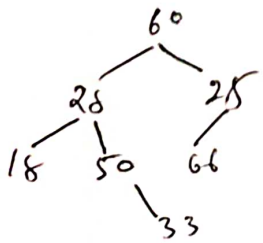


ROOT 3

AVAIL 5

	INFO	LEFT	RIGHT
1	33	0	9
2	25	4	10
3	60	2	7
4	66	8	0
5		6	
6		0	
7	75	4	0
8	18	0	0
9	44	0	0
10	50	1	0

Suppose we delete 44 from tree!

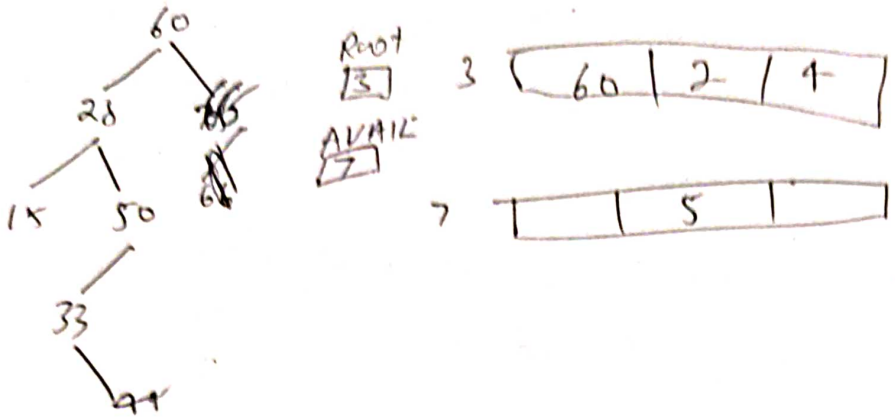


ROOT 3

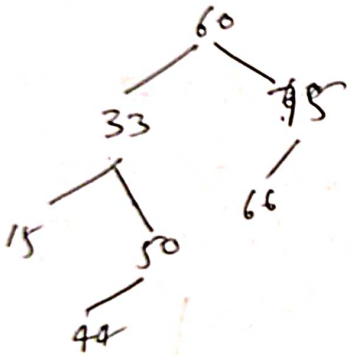
AVAIL 9

1	33	0	0
9			5

(b) delete 75 from tree:



(c) delete 25 from T:



Huffman's Algorithm:

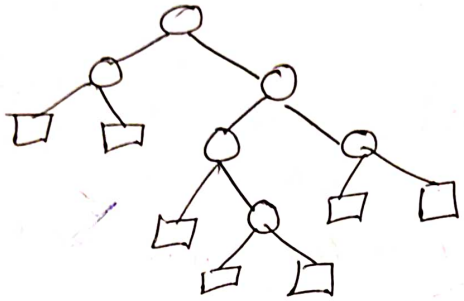
Recall the extended binary tree in which each node has either 0 or 2 children. the node with 0 children are called external nodes, and the node with 2 children are called internal nodes.

Internal node denoted by → circle.

External node denoted by → square.

$$N_E = N_I + 1$$

eg.



$N_E = 7$

$N_I = 6$

External path length $L_E = 2+2+3+4+4+3+3 = 21$

Internal path length $L_I = 0+1+1+2+2+3 = 9$

observe the

$$L_I + 2n = 9 + 2.6 = 9+12 = 21 = L_E$$

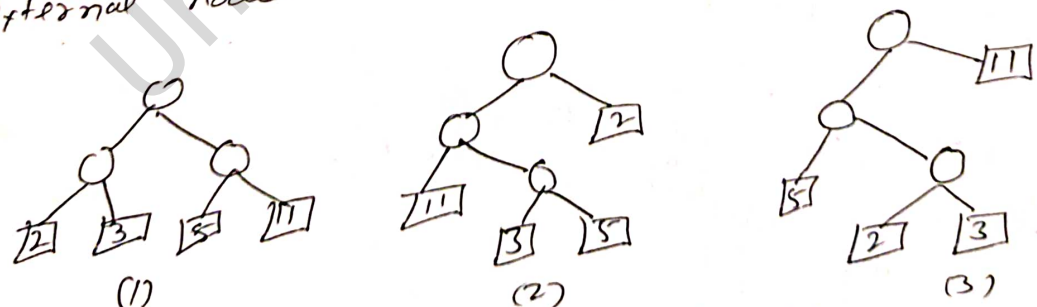
$$L_E = L_I + 2n$$

- Suppose binary T is a 2-tree with n external nodes, and suppose each of the external node is assigned a (non-negative) weight

The external weighted path length $P = w_1 L_1 + w_2 L_2 + \cdots w_n L_n$

w_i and L_i denote respectively weight and length of an external node.

eg.



$P_1 = 2*2 + 3*2 + 5*2 + 11*2 = 42$

$P_2 = 11*2 + 3*3 + 5*3 + 2*1 = 48$

$P_3 = 11*1 + 3*3 + 2*3 + 5*2 = 36$

the general problem that we want to solve is as follows.
suppose a list of n weights is given

$w_1, w_2 \cdots w_n$

among all the 2-Trees with n external node and with a given n weights, find a tree T with a minimum weighted path length.

Huffman gave an algorithm ~~which are~~ to find such a tree T.

Algorithm

suppose w_1 and w_2 are two minimum weight among the n given weight $w_1, w_2 \cdots w_n$. find a tree T' which give a solution for n-1 weight.

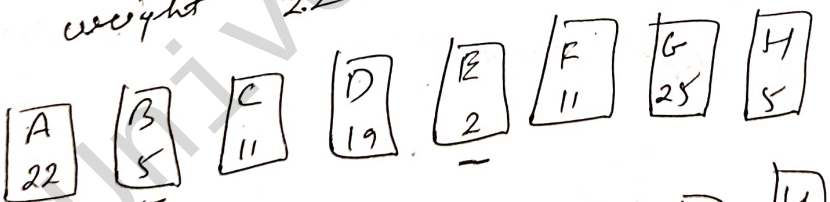
$w_1 + w_2, w_3, w_4 \cdots w_n$

represent $w_1 + w_2$ by the subtree

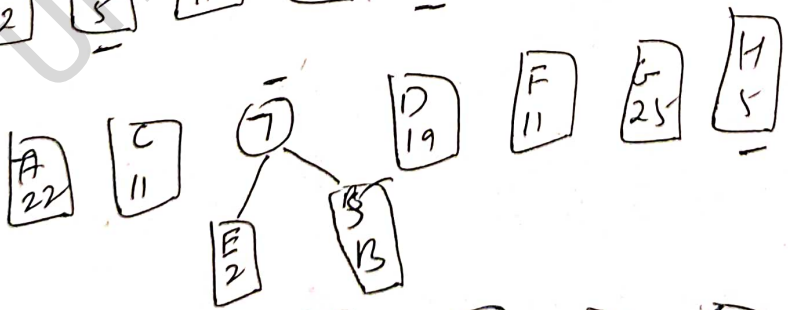
Eg. Suppose A, B, C, D, E, F, G, and H are 8 data item and suppose they are assigned weight as follow.

Data item	A	B	C	D	E	F	G	H
weight	22	5	11	19	2	11	25	5

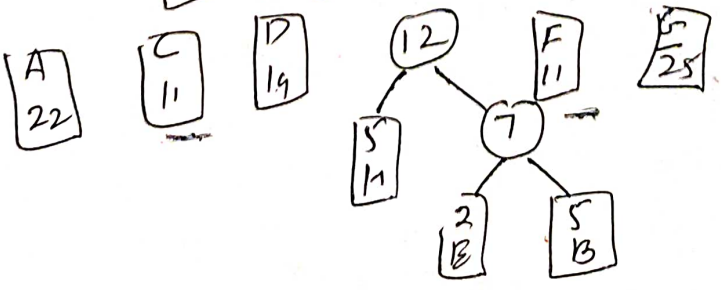
(a)



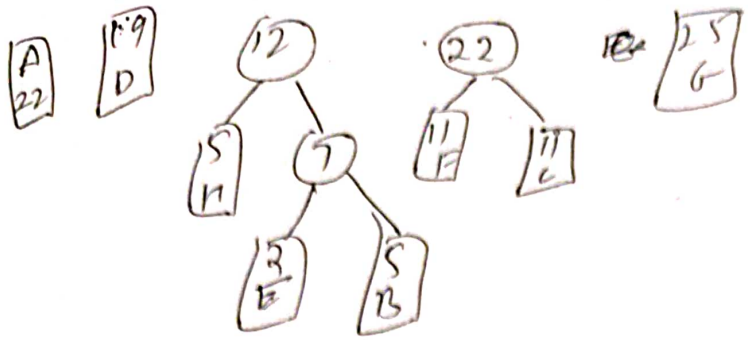
(b)



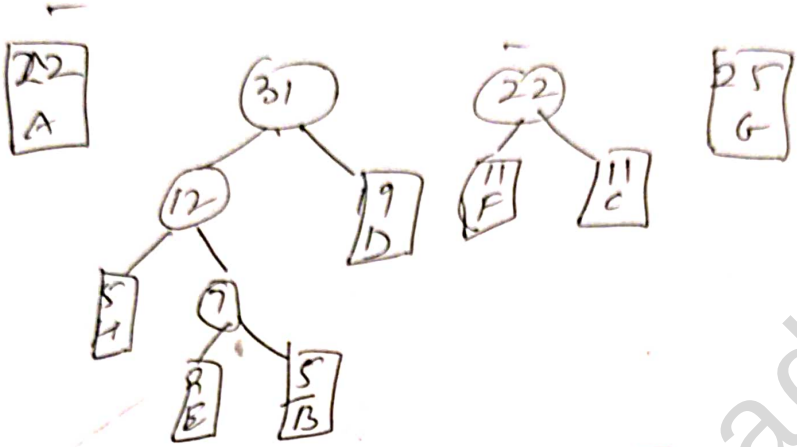
(c)



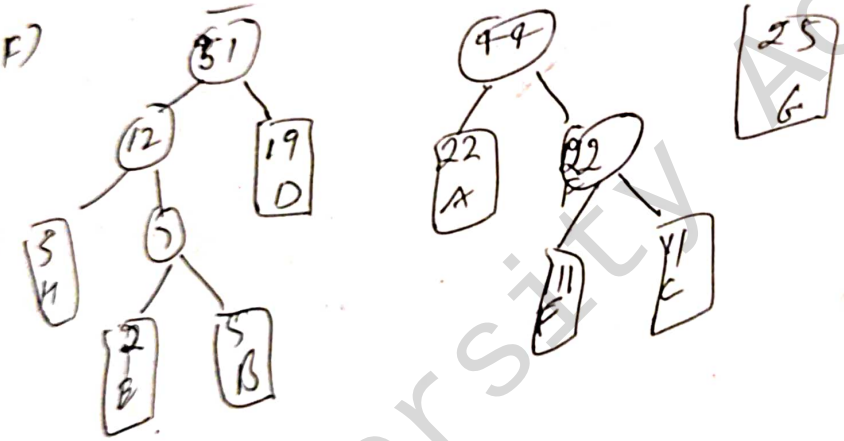
(D)



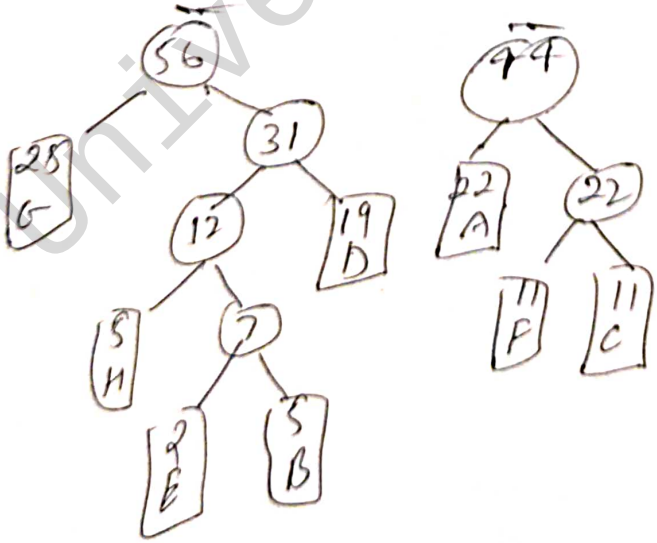
(E).



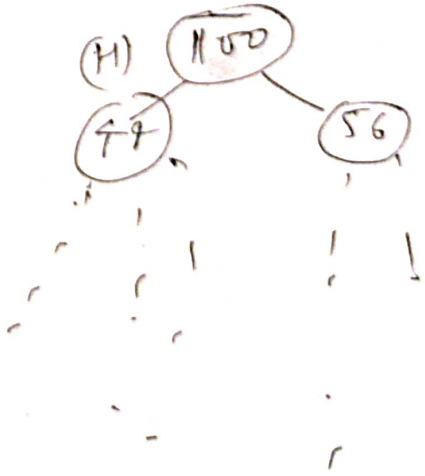
(F)



(G)



(H)



AVL Tree:

Problem with binary tree:

if we insert element A, B, C ... Z into BST and if we insert element Z, Y, X ... A into BST the tree shows as bellow

we observe that the worst case to search an element in BST is O(n). that is the disadvantage of BST. The AVL tree Remove the disadvantage of BST. and the complexity of AVL tree in all case will be O(log n). the AVL tree is most popular balance tree was introduce in 1962 by Adelson - Velski's and Landis.

Definition:

⇒ An empty binary search tree is an AVL tree.

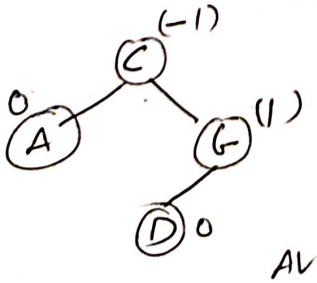
⇒ A non-empty binary tree has T^L and T^R and $h(T^L)$ and $h(T^R)$ are AVL Tree iff

$$|h(T^L) - h(T^R)| \leq 1$$

T^L — Left Subtree
T^R - Right subtree
$h(T^L)$ → height of Left Subtree
$h(T^R)$ → height of Right subtree
$h(T^L) - h(T^R)$ → Balance factor

Note: for all AVL tree the balance factor of a node can be either 0, +, or -1.

e.g.

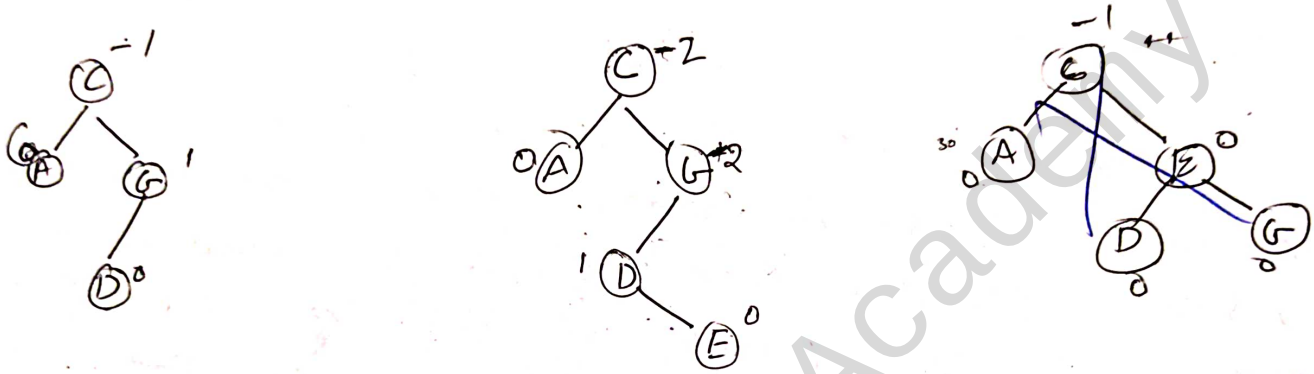


AVL tree

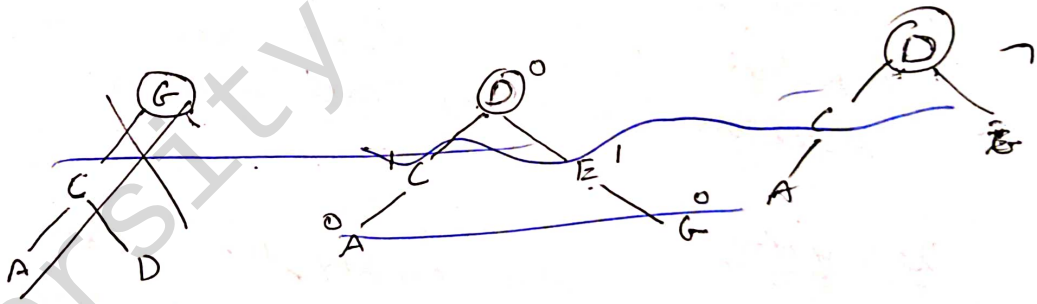
Insertion in an AVL Search tree..

Inserting an element in AVL search tree in it's first phase is similar to ~~BST~~ BST. however after the insertion the balance factor of any node in tree it's affected so we have to solve this by the techniques called Rotation.

Suppose if we insert 'E' into the tree.



So we have to perform Rotation to form a AVL Search tree.



there are four type of Rotation possible in AVL tree.

LL Rotation: inserted node is in the left subtree of left subtree of A

RR Rotation: inserted node in the Right subtree of Right subtree of A

LR Rotation: inserted node in Right subtree of left subtree of A.

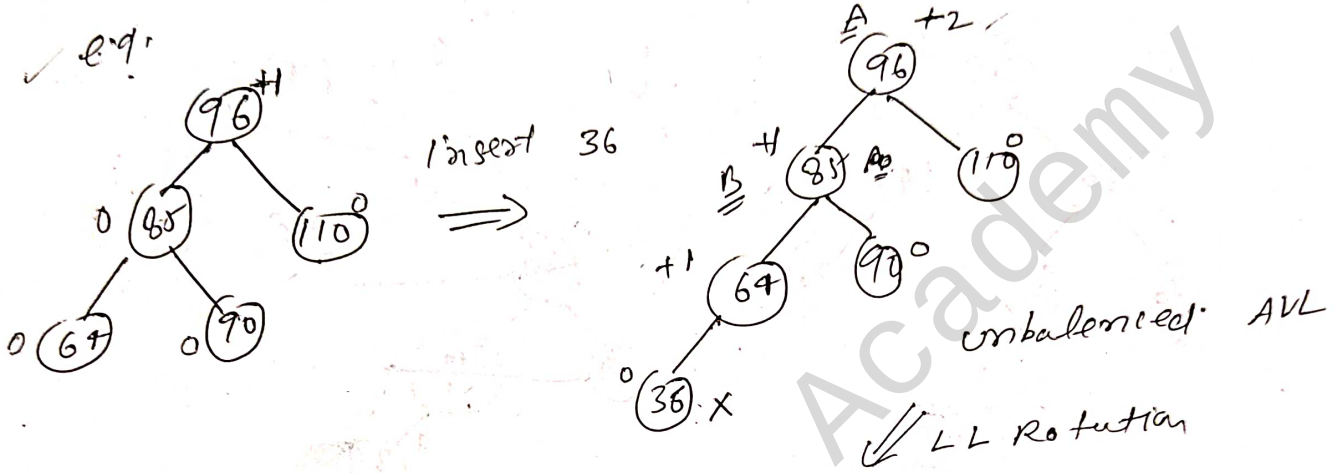
RL Rotation inserted node in left subtree of Right subtree of A.

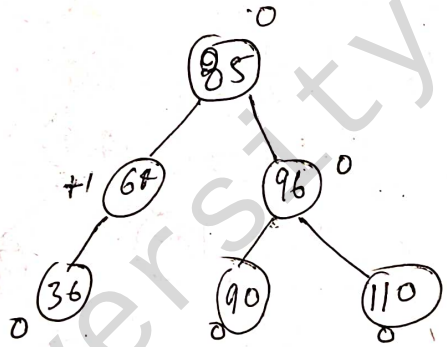
LL Rotation

→ the new element X is inserted in the left subtree of left subtree of A, the closest ancestor node whose BF(A) becomes +2 after iteration.

→ B to be root with B_L and A to be its left subtree and right child. and B_R and A_R to be ~~Right~~ the ~~left~~ and right ~~child~~. subtree of A.

e.g.

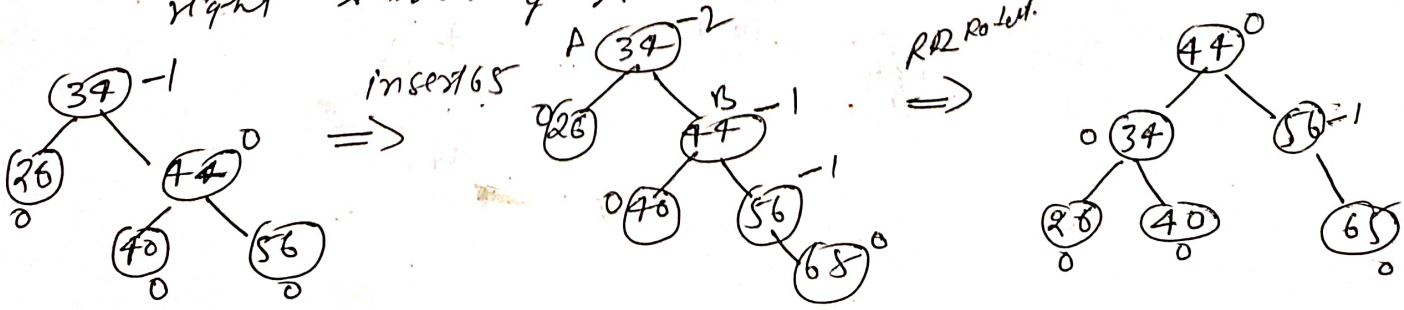




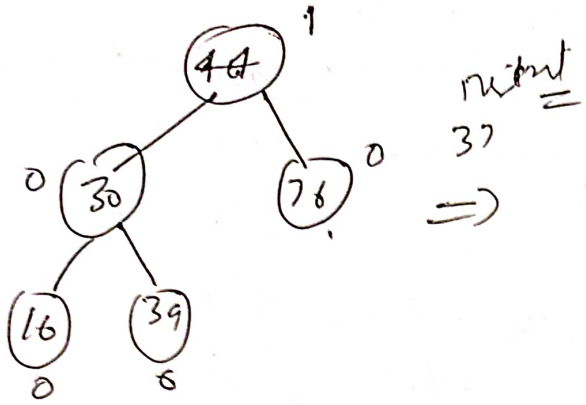
RR Rotation

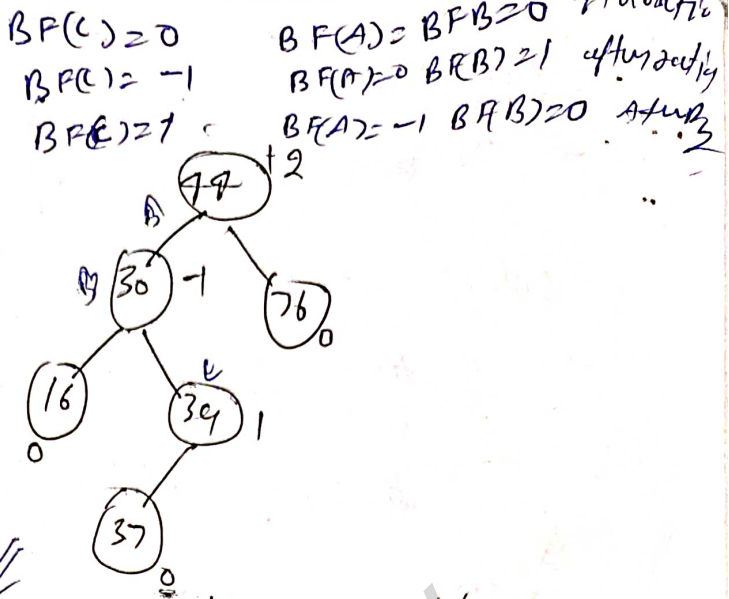
→ the new element X is in the right subtree of Right subtree of A.

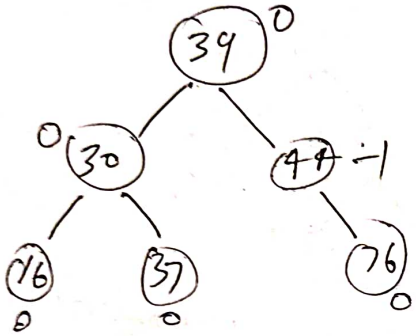
→ Pushes B up to the root with A as its left child and B_R as its right subtree and A_L and B_L as left and right subtree of A.

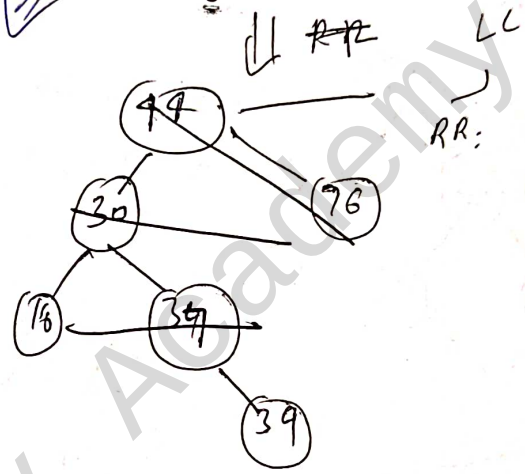


LR Rotation

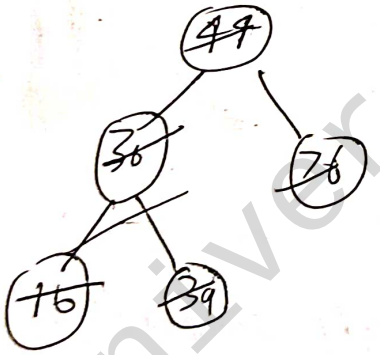


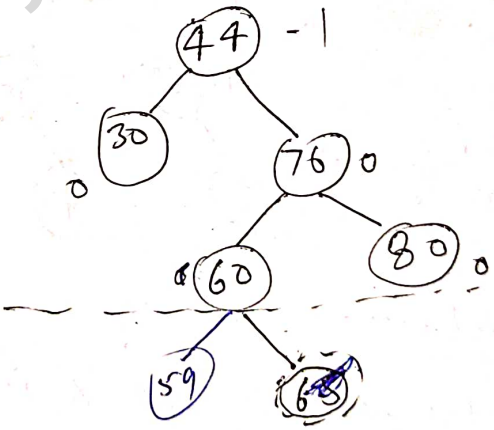


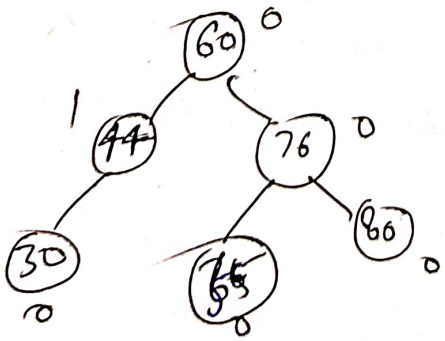


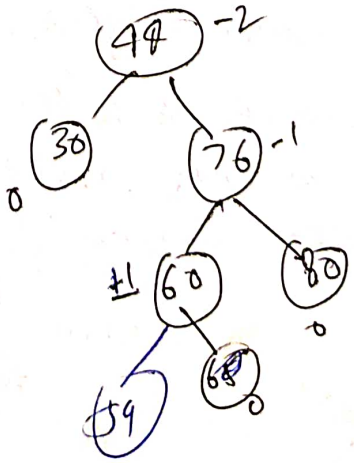


RL Rotation

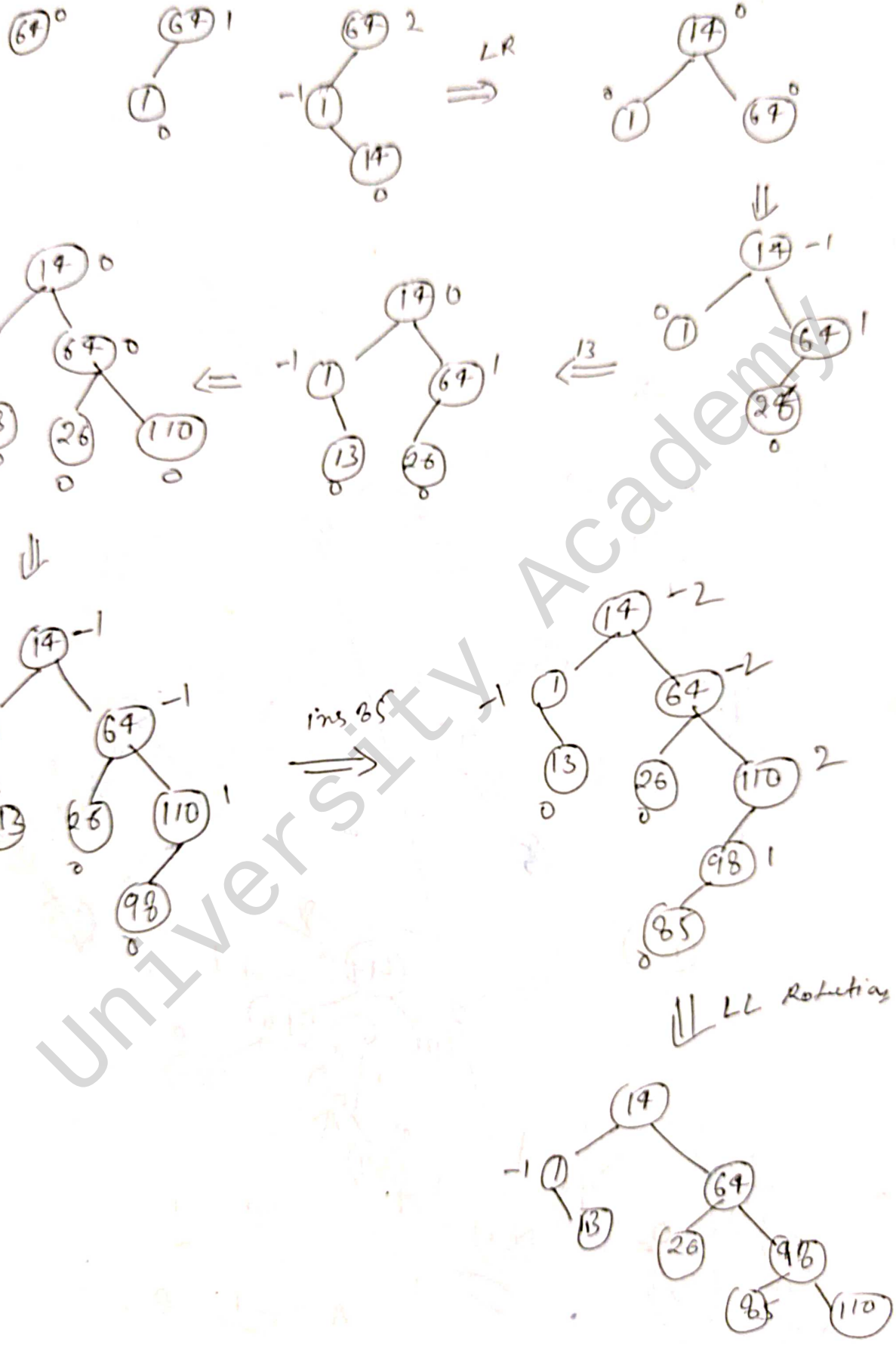








Exam 64, 1, 14, 26, 13, 110 98 85

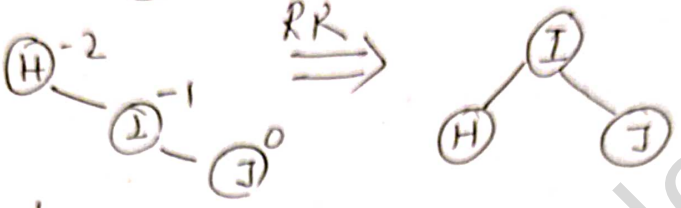


H, I, J, B, A, E, C, F, D, G, K, L

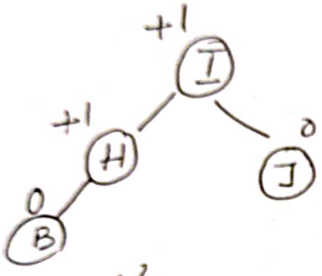
H →

I →

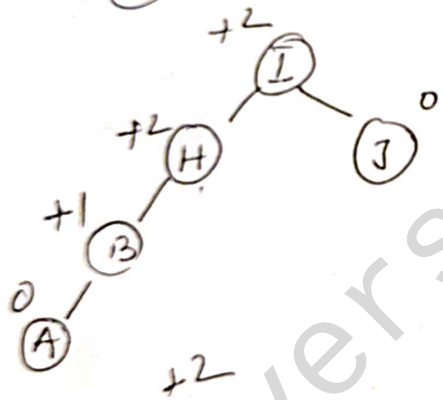
J →



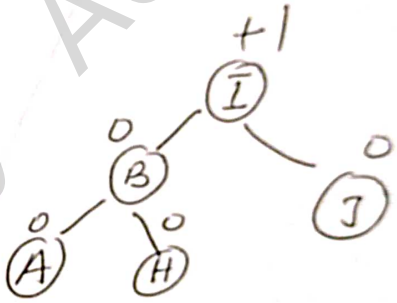
B →



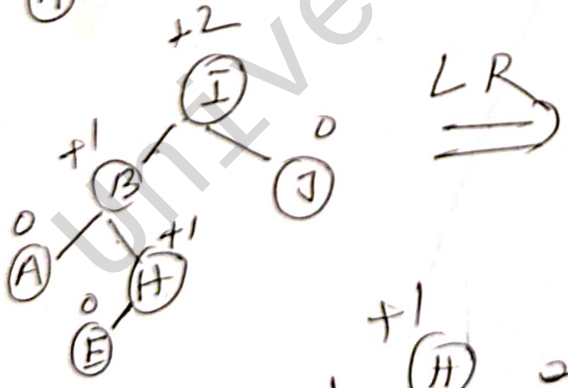
A →



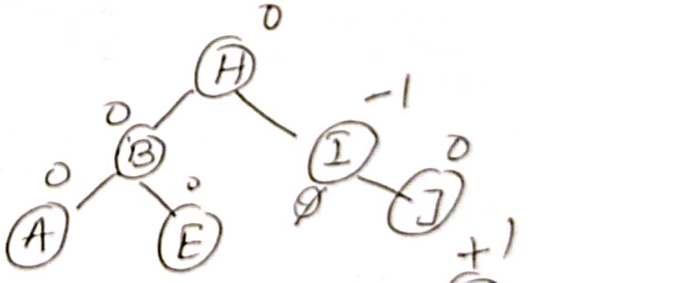
LL



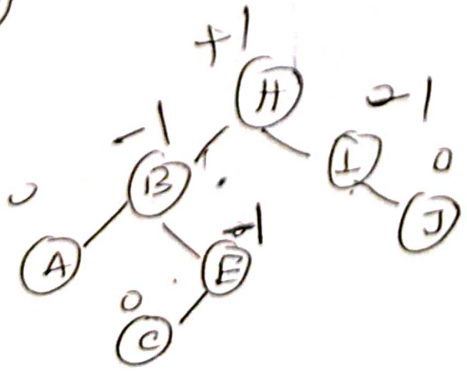
E →



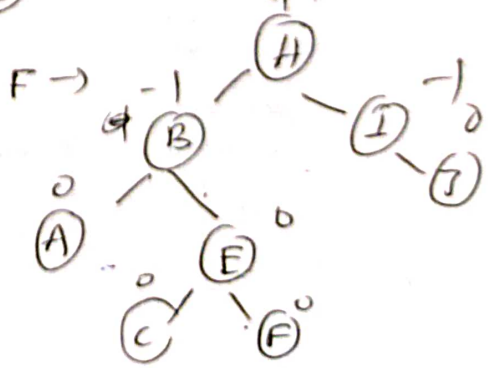
LR



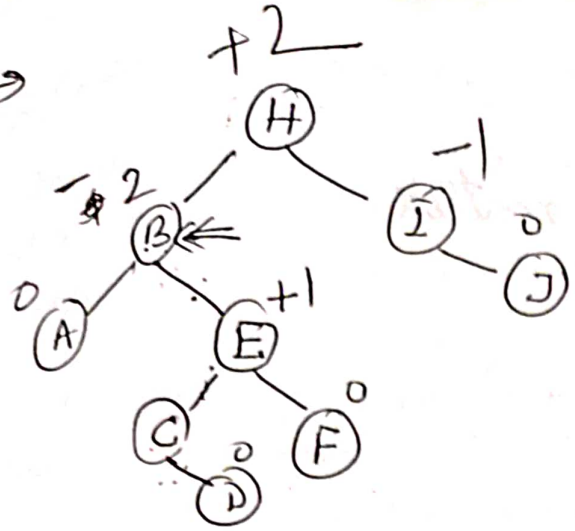
L →



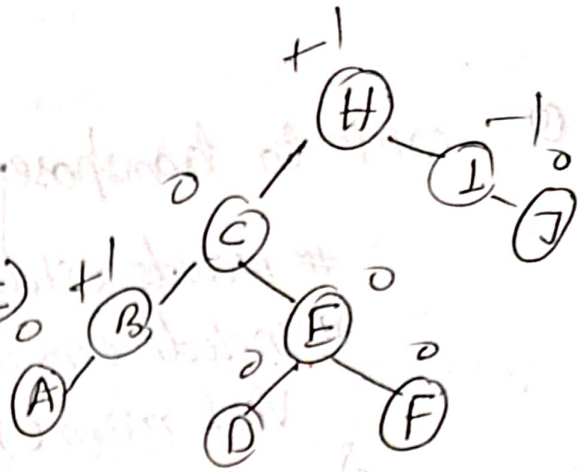
F →



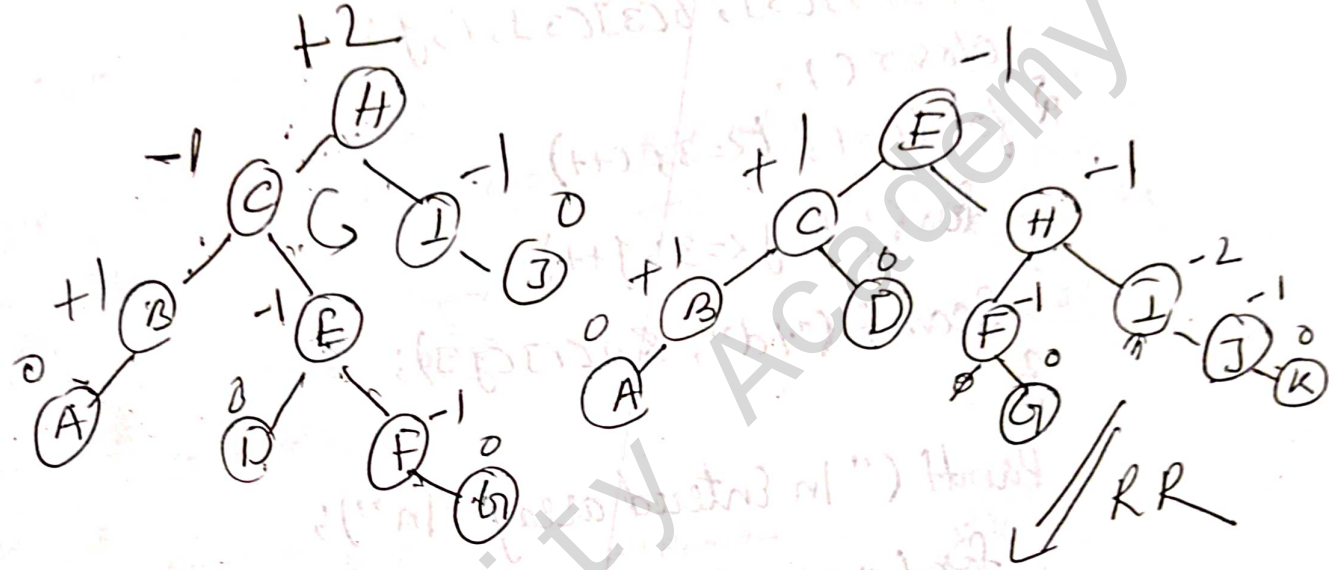
D →



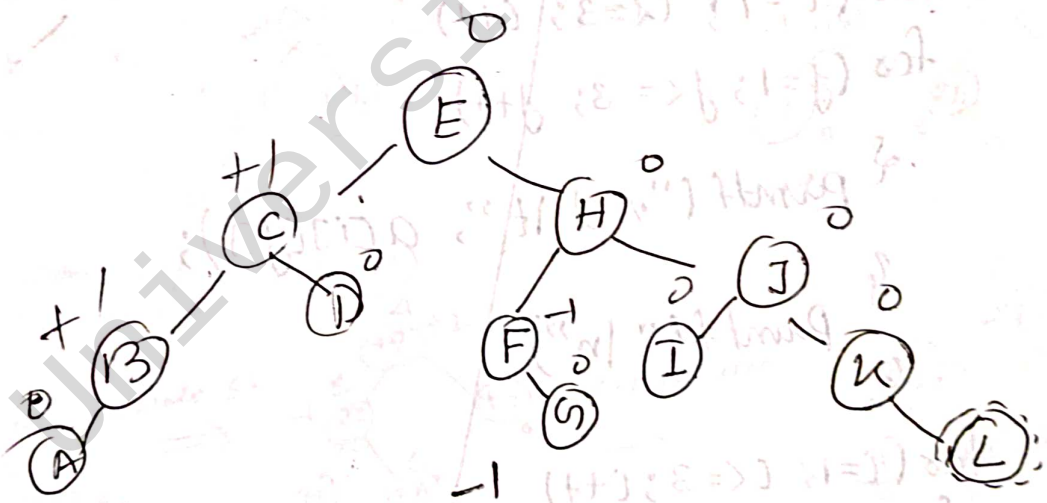
RL



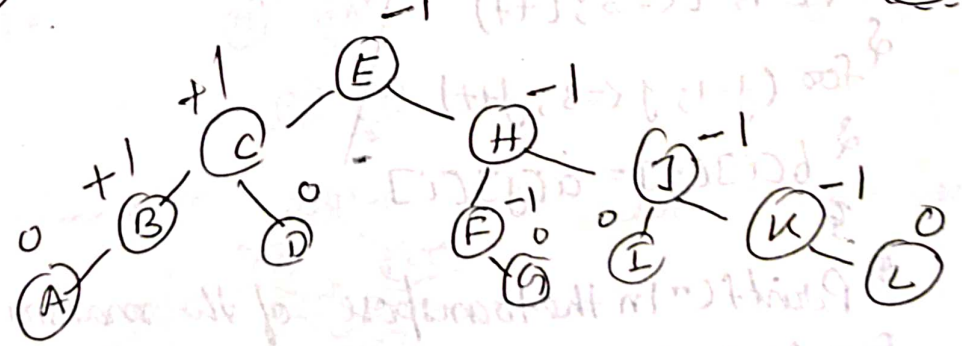
G →



K →



L →

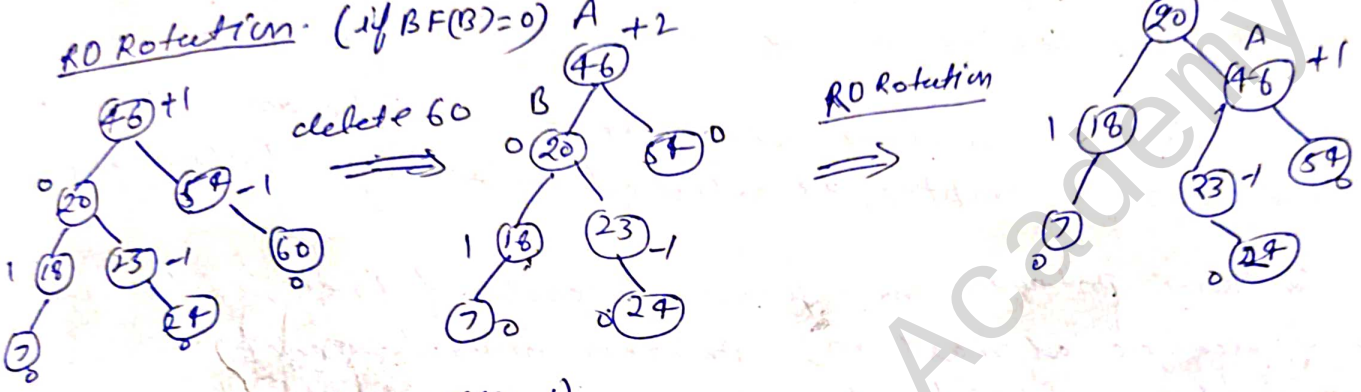


Deletion in an AVL search tree:

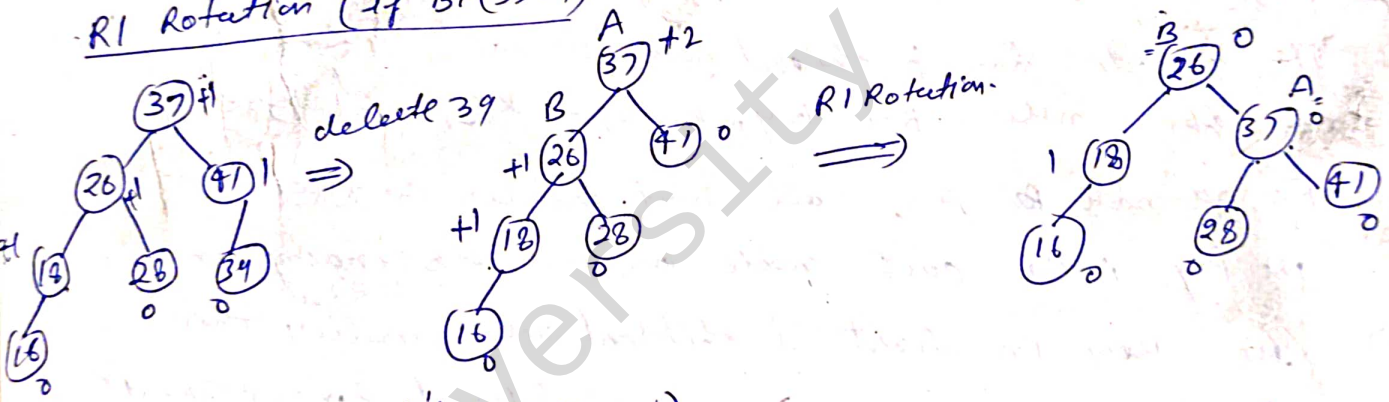
On deletion of a node X from AVL tree let A be the closest Ancestor node on the path node X to root node with a BF of +2 or -2 to restore BF the rotation is required.

the deletion may occured on left or right subtree of A, for deletion from right subtree of A, R0, R1 or R-1 rotations are required and for deletion from left subtree of A, L0, L1 or L-1 rotation are required.

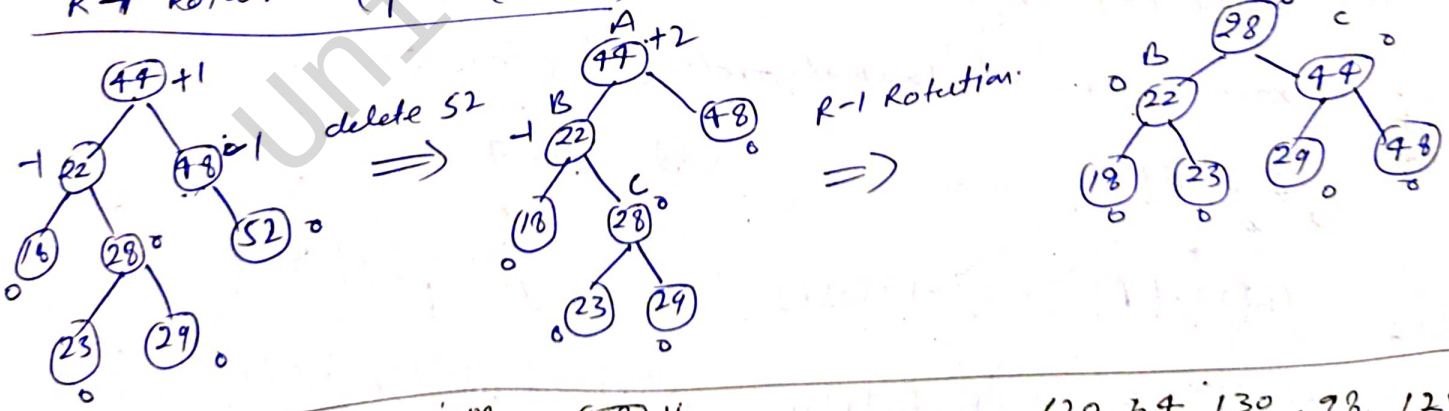
R0 Rotation (if BF(B)=0)



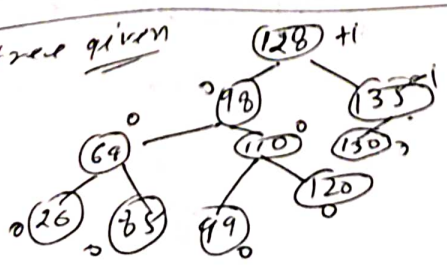
R1 Rotation (if BF(B)=1)



R-1 Rotation (if BF(B)=-1)



Example. AVL tree given



delete - 120, 64, 130, 98, 128

first in the stack

m-Way search tree.

All the data structure such as array, link list, stack, queue BST, AVL etc. so far favor data stored in the internal memory. for the data retrieval and manipulation of data stored in secondary storage device such as disk. there is a need to support some special data structure. ~~called~~ such as m-way search tree, B trees and B^+ tree.

m-way search trees are generalized version of BST. the goal of m-way search tree is to minimize the accesses while retrieving a key from file.

An m-way search tree may be an empty tree. if tree is not empty, it satisfied following properties.

(i) for some integer m, known as order of tree, each node is of degree at most m, in other word each node has at most m child and m-1 key.

(ii) ~~if a node has k child node where $k \le m$ then the node can have only k-1 key, $k_1, k_2 \dots k_{k-1}$ and $k_i < k_{i+1}$~~

(iii) ~~For a node A, all key values in the subtree~~

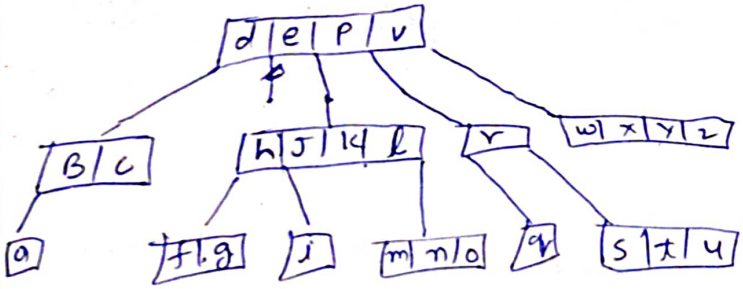
(iv) the key in the each node are in ascending order

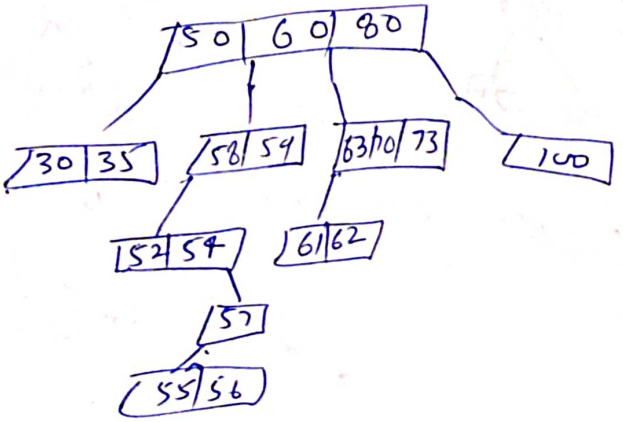
(iii) the key in first i children are smaller than ith key

(iv) The key in the last m-i children larger than ith key.

eg. (1) m-way search tree of order 5 (m=5)

(2) m=4

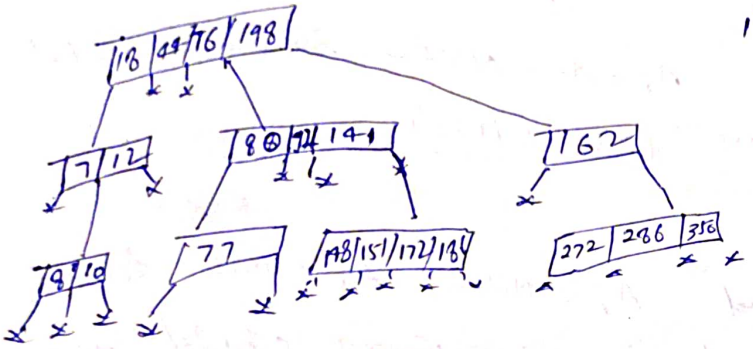




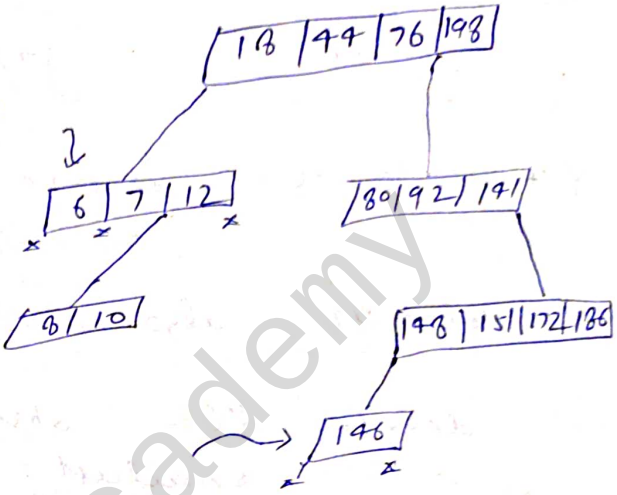
m-way search tree: Insert / Delete

Insertion: search for a key in m-way search tree as BST manner. and if key is not found insert the key at appropriate place.

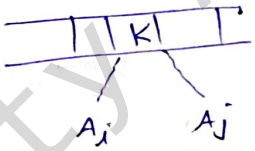
e.g. 5-way tree



insert 6 and 146



deletion look the situation:



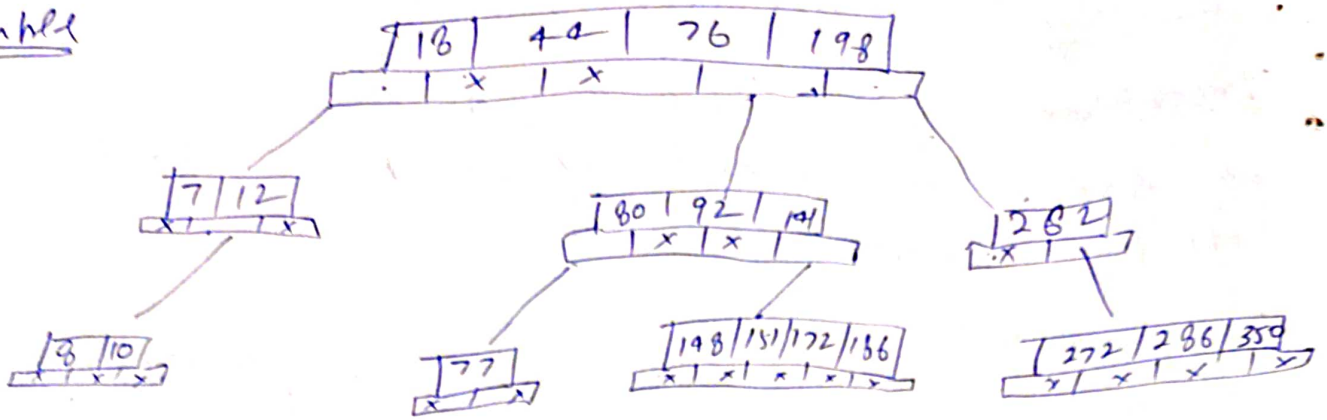
(1) if $A_i = A_j = NULL$ then delete k.

(2) if $A_i \neq NULL$, $A_j = NULL$ then chose largest of key element k' in the child node pointed by A_i delete key k' and Replace k by k'.

(3) if $A_i = NULL$, $A_j \neq NULL$ then chose smallest of key element k' in the child node pointed by A_j delete k' and Replace k by k'

(4) if $A_i \neq NULL$ and $A_j \neq NULL$ then chose either step 2 or 3.

Example



(i) delete 151 observe that $A_i = A_j = NULL$ so simply delete. and node became 148 | 172 | 186

(ii) delet 92 also similar.

(iii) delete 262 observe that $A_i = NULL$ $A_j \neq NULL$ hence smallest element 272 from child node 272 | 286 | 350 delet and Replace 262 by 272.

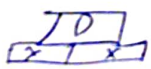
(iv) delet 272 – same as (iii).

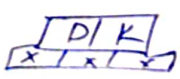
(v) delet 12 observe that $A_i \neq NULL$ $A_j = NULL$ hence largest element 10, from child node 8 | 10 delete and Replace 12 with 10.

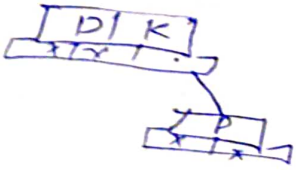
(vi) deled 198 observe that $A_i \neq NULL$ $A_j \neq NULL$ so abbly any of Condition 2 or 3.

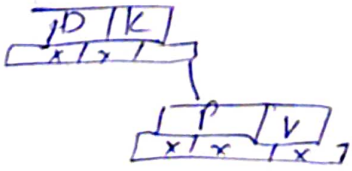
Question ↑ 3-way search tree constructed out of an empty search tree. with following key in the order shown.

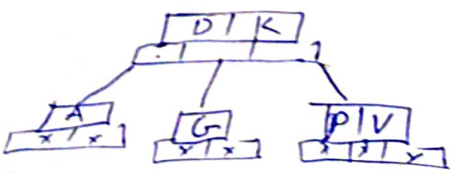
D, K, P, V, A, G

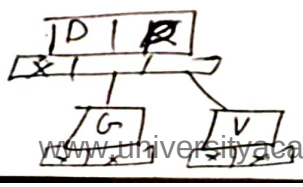
insert D: 

insert K 

insert P 

insert V 

insert A and G: 

delete A and K 

B-Tree:

B tree is balanced m-way tree. The improvement over m-way tree in case of height of tree. B-tree try to kept height of tree down as possible.

A B Tree of order m if non empty, is an m-way search tree in which.

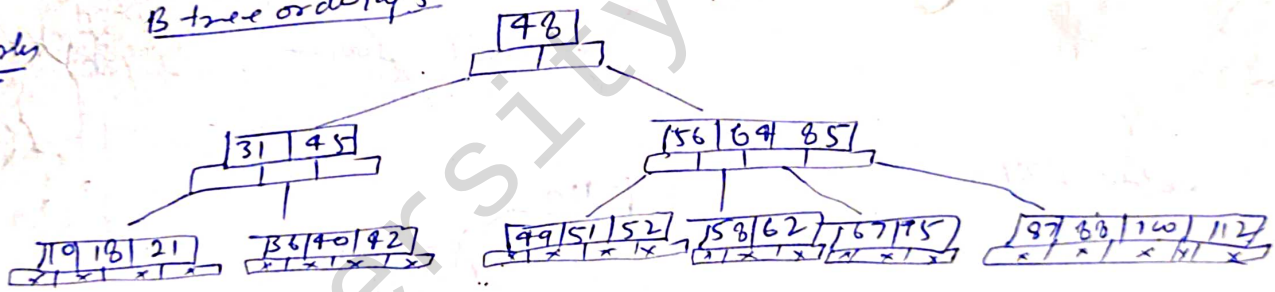
(i) the root has at least two child node and at most m child node

(ii) the Internal node except the root have at least $\lceil \frac{m}{2} \rceil$ child node and at most m child node.

(iii) the number of key in the each internal node is one less than the number of child node and partition of key in a manner similar to that of m way search trees.

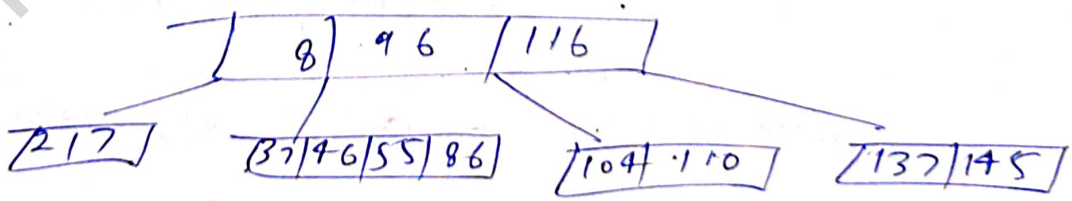
(iv) all leaf nodes are on same level.

Example: B tree order of 5

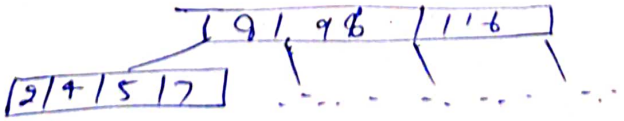


Insertion in a B Tree:

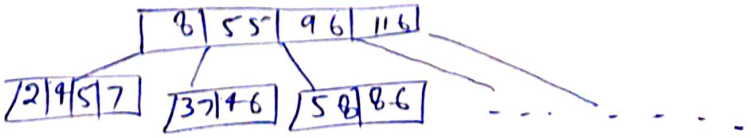
Consider the B Tree of order 5 shown in fig insert the element 4, 5, 58, 6 in the order given.



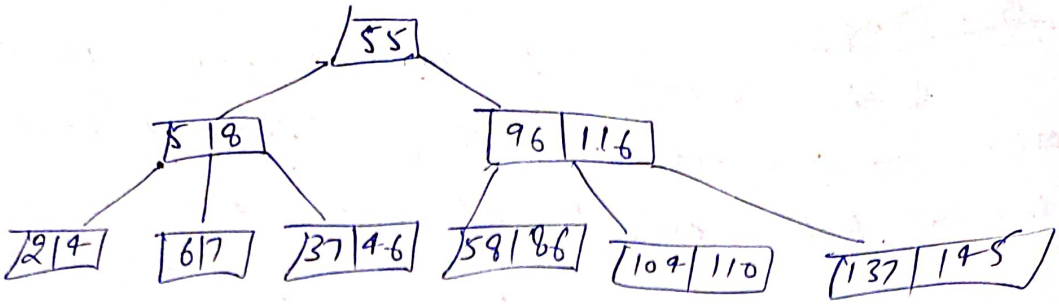
insert 4 and 5



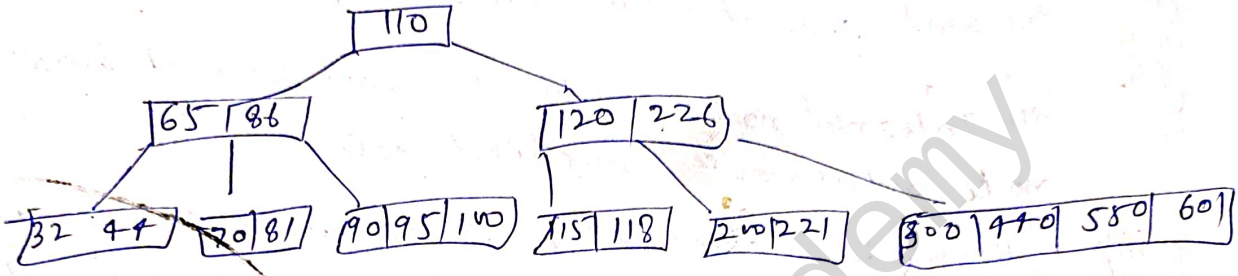
insert 58



Insert 6

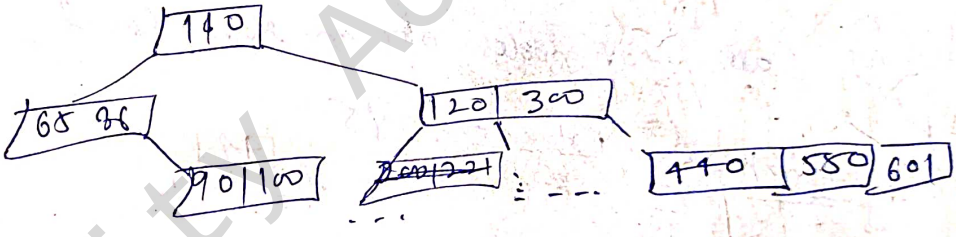


Deletion in a B Tree-

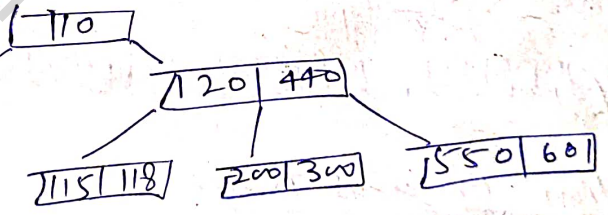


delete 95, 226, 221 and 70 in given order 5 B-Tree!

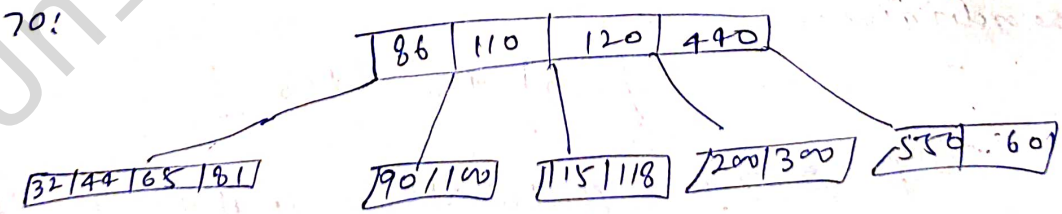
delete 95, 226



delet 221



delet 70:



B^+ Tree:

A slight modification over B Tree is known as B^+ tree. The primary distinction between two approaches is that, a B^+ tree allows redundant storage of key value. A B^+ tree have following properties:

1- Every node has one more references than it has keys.

2. All leaves are at the same distance from root.

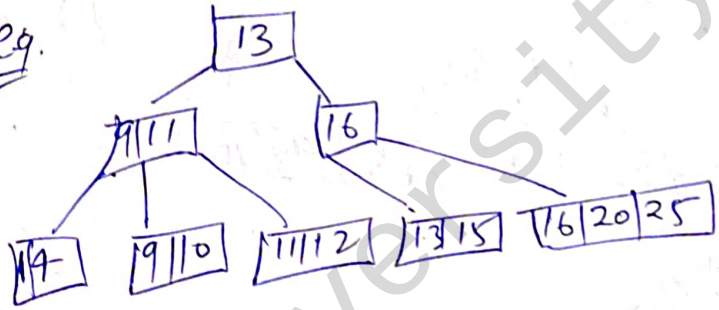
3- The root has atleast two children.

4- Every non-leaf, non-root node has atleast $\lceil m/2 \rceil$ children.

5. ~~Each leaf contain at least $\lceil \frac{m}{2} \rceil$ key.~~

6- Every key from the table appears in a leaf, in left-to-right sorted order.

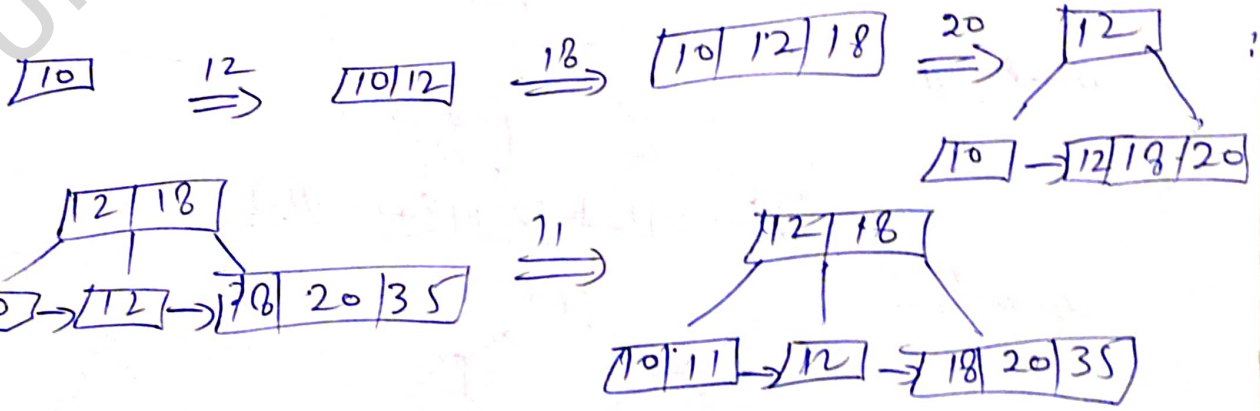
eg.



Insertion in B^+ tree. (10, 12, 18, 20, 35, 11).

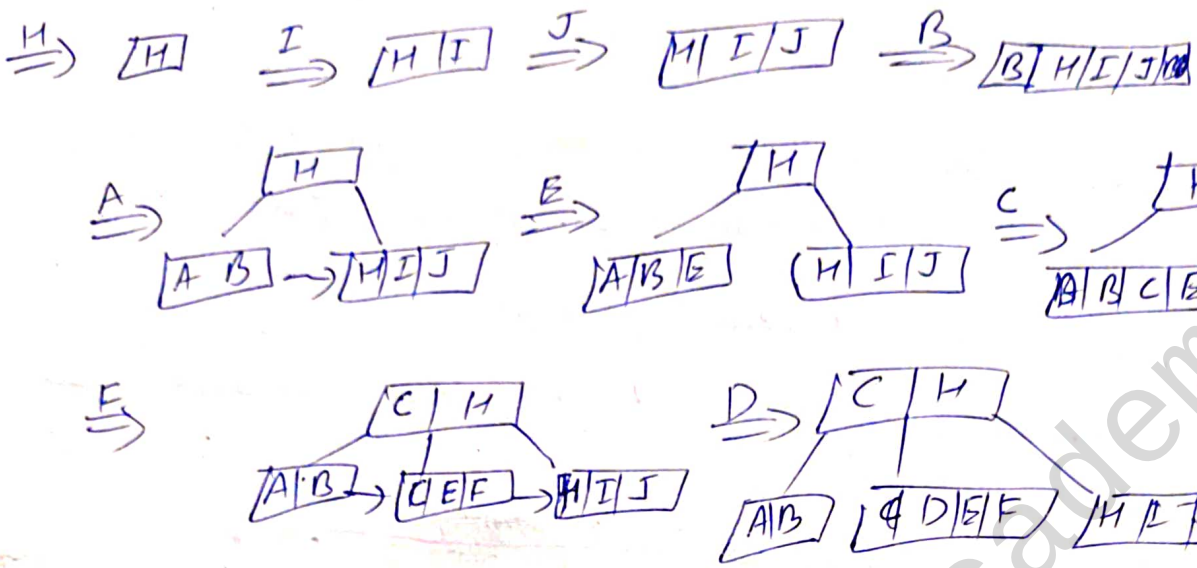
m = 4

Insert 10 ⇒



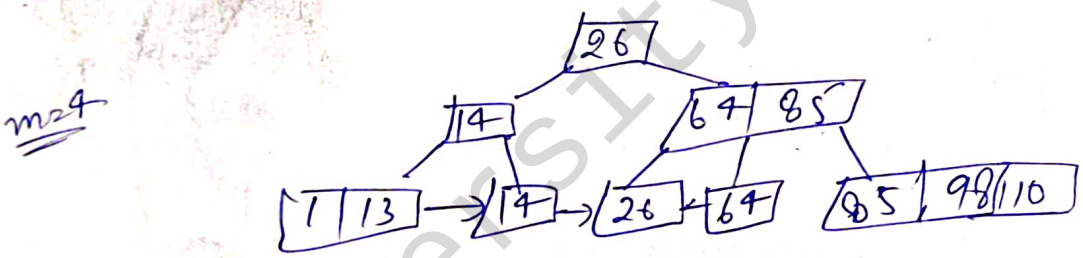
Example Insert the following elemnt in given order in B+ tree

H, I, J, B, A E C, F D,

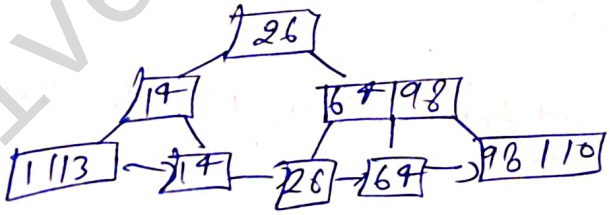


Deletion in B+ tree.

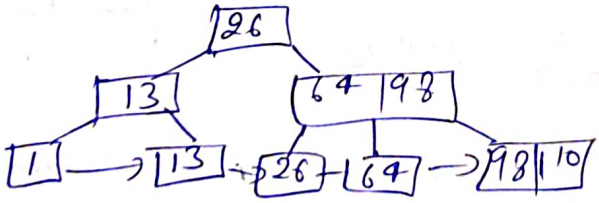
64, 1, 14, 26, 13, 110, 98, 85



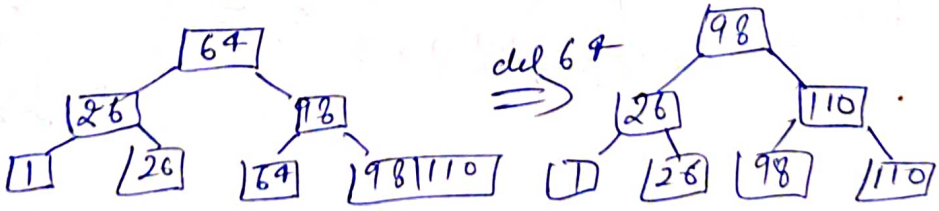
delet: 85



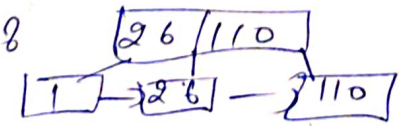
delet: A



delet: 13



del 98



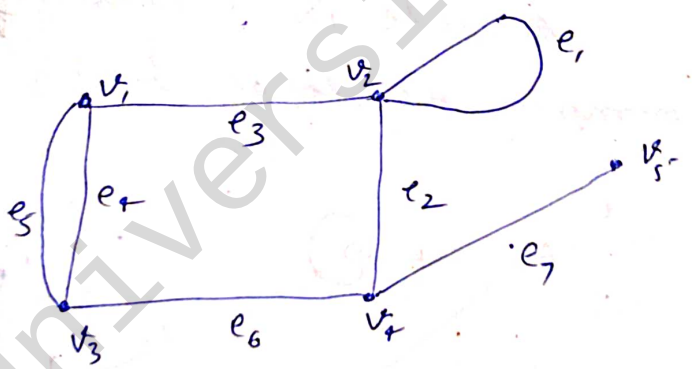
Graph.

A graph $G=(V,E)$ consist of a set of V element called node (point or vertices) and another set E of edges such that each ~~eqt~~ edge e in E is identified with unique pair $[u,v]$ of node in V. denoted by $e=[u,v]$.

Suppose $e=[u,v]$ then node u and v are called endpoint of e, and u and v are said to be adjacent node or neighbores. the degree of node u, written $deg(u)$, is the number of edges containing u. if $deg(u)=0$ ~~that~~ then u is called isolate node.

A path P of length n from a node u to a node v is defined as a sequence of n+1 node

$$P=(v_0, v_1, v_2 \dots v_n)$$



eg. $e_6=[v_3\ v_4] \Rightarrow v_3, v_4$ are Adjacent node.

degree of node $deg(v_3)=3$

path $P=(v_1\ v_2\ v_4\ v_5)$, 3 edge and 4 node on the path from $v_1 \to v_5$.

A graph is connected if if there is a path between any two of its node. other wise graph is disconnected.

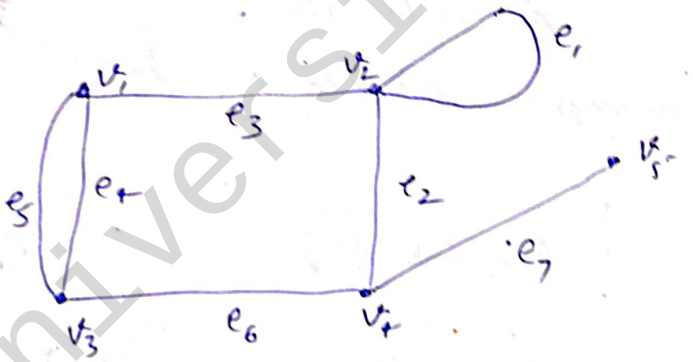
Graph.

A graph $G=(V,E)$ consist of a set of V element called node (point or vertices) and another set E of edges such that each edge e in E is identified with unique pair $[u,v]$ of node in V. denoted by $e=[u,v]$.

Suppose $e=[u,v]$ then node u and v are called endpoint of e, and u and v are said to be adjacent node or neighbores. the degree of node u, written $deg(u)$, is the number of edges containing u. if $deg(u)=0$ then u is called isolate node.

A path P of length n from a node u to a node v, is defined as a sequence of n+1 node

$$P = (v_0, v_1, v_2 \dots v_n)$$



eg. $e_6 = [v_3\ v_4] \Rightarrow v_3, v_4$ are Adjacent node.

degree of node $deg(v_3) = 3$

path $P=(v_1\ v_2\ v_4\ v_5)$, 3 edge and 4 node on the path from $v_1 \to v_5$.

A graph is connected if if there is a path between any two of its node. other wise graph is disconnected.

Complete graph: A graph is complete if every node u in G is adjacent to every other node v in G. A complete graph will have $n(n-1)/2$ edges.

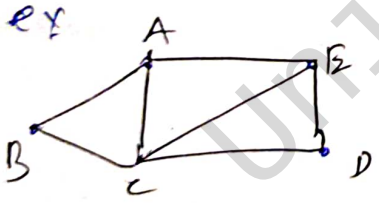
Labeled graph A graph is said to be labeled if it's edges are assigned data. In the labeled graph some weight assigned to each edges.

multi-edges graph: If edges e and e' have same end point, e and e' are parallel edges.

Loop: An edge e is called loop if it has identical end point i.e. $e = [u, u]$.

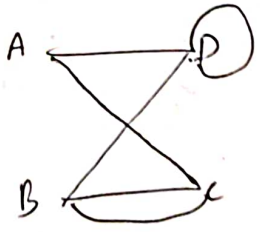
~~Finite and~~ **Finite and Infinite graph:** the graphs have finite no. of vertices and edges called finite graph. ~~and~~ otherwise if is Infinite graph.

Cycle in graph A cycle is a closed simple path with length 3 or more. A cycle of length k is called a k-cycle.

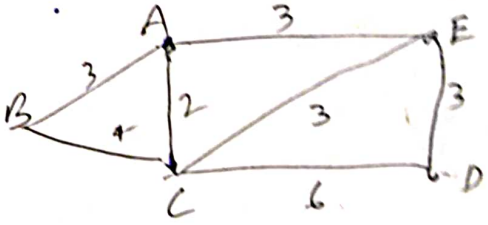


No. of node - 5
no. of edge : 6
path from B to E
⇒ BAE, BCE
BCDE

4-cycle in the graph is
[A B C E A]
&
[A C D E A]



multigraph.
having self loop
and parallel edges.



weighted graph.

Q: Identify tree and graph.

Directed Graph.

A directed graph G, also called a digraph or graph is same as multigraph except that each edges e in G is assigned a direction.

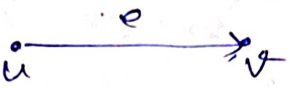
the following terminology is used in Digraph:

(1) e begin at u and end at v

(2) u is the origin of e and v is the destination of.

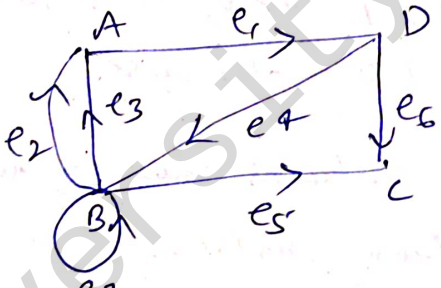
(3) u is predecessor of v and v is a successor of u

(4) u is adjacent to v and v is adjacent to u



the outdegree of node u, written as outdeg(u), is number of edges beginning at u. similarly indegree u, indeg(u) is number of edge ending at u

eg:



Parallel edge: e_2, e_3

selfLoop e_7

Path (D → A): D B A

indeg(d) = 1

outdeg(d) = 2

A directed graph is strongly connected if for each pair u, v of nodes in graph there is a path from u to v and there is a also a path from v to u.

Note: A directed graph G is said to be simple directed graph if G has no parallel edges. it can have a loop.

Graph Representation in memory.

There are two standard way to maintaining a graph in the memory of a computer.

(1) Sequential Representation

(2) linked Representation:

1- Sequential Representation:

The sequential Representation achive by the adjacency matrix of graph.

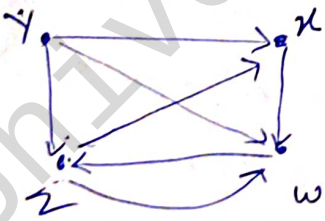
Suppose G is a simple directed graph with m nodes and suppose the node of G have been ordered and are called $v_1, v_2 \ldots v_m$. then

Adjacency matrix $A = (a_{ij})$ of graph G is the $m \times m$ matrix defined as:

$$a_{ij} = \begin{cases} 1 & \text{if } v_i \text{ is adjacent to } v_j \\ 0 & \text{otherwise} \end{cases}$$

such matrix which contains only 0, and 1's called bit matrix or Boolean matrix.

Example.



Adjacency Matrix A =

$$A = \begin{array}{c} \\ x \\ y \\ z \\ w \end{array} \begin{array}{c} \begin{array}{cccc} x & y & z & w \end{array} \\ \begin{bmatrix} 0 & 0 & 0 & 1 \\ 1 & 0 & 1 & 1 \\ 1 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{bmatrix} \end{array}$$

Consider the powers. $A\ A^2 A^3 \ldots$

$$A^2 = \begin{array}{c} \\ x \\ y \\ z \\ w \end{array} \begin{array}{c} \begin{array}{cccc} x & y & z & w \end{array} \\ \begin{bmatrix} 0 & 0 & 1 & 0 \\ 1 & 0 & 1 & 2 \\ 0 & 0 & 1 & 1 \\ 1 & 0 & 0 & 1 \end{bmatrix} \end{array}$$

$$A^3 = \begin{bmatrix} 1 & 0 & 0 & 1 \\ 1 & 0 & 2 & 2 \\ 1 & 0 & 1 & 1 \\ 0 & 0 & 1 & 1 \end{bmatrix} \qquad A^4 = \begin{bmatrix} 0 & 0 & 1 & 1 \\ 2 & 0 & 2 & 3 \\ 1 & 0 & 1 & 2 \\ 1 & 0 & 1 & 1 \end{bmatrix}$$

$$B_4 = \begin{bmatrix} 1 & 0 & 2 & 3 \\ & & & \end{bmatrix}$$

B_r shows the no. of path of length r or less from node v_i to v_j

(2) Link Representation of graph:

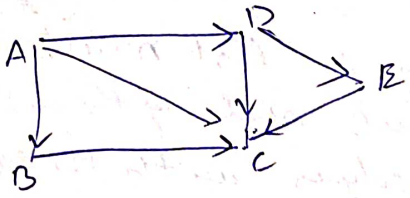
there are some drawbacks in sequential Representation..

1. It may be difficult to insert and delete node in it. because the size of matrix A may need to be changed and node need to be reordered.

2. if the number of edges is equal to no. of node then the matrix A will be sparse (will contain many zero) hence greate deal of space will be wasted.

Now the Above drawback of sequential Representation can be remove by link Representation also called an Adjacency structure.

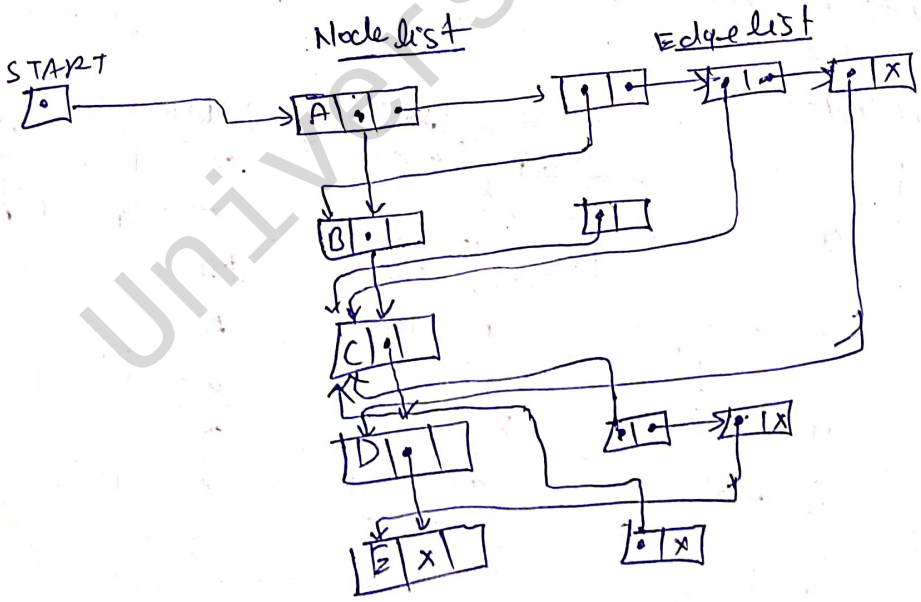
Consider the graph.



graph

Node	Adjacency list
A	B C D
B	C
C	C E
D	C
E	

adjacency list of Graph.



Link Representation of G in memory will contain two list. Node list and edge list.

Node list :- Each element in the Node list will correspond to a node in G. and it will be a record of the form.

NODE	NEXT	ADJ	

NODE → will be name or key value of the node

NEXT → will be a pointer to the next node in NODE list.

ADJ → will be a pointer to the first element in adjacency list

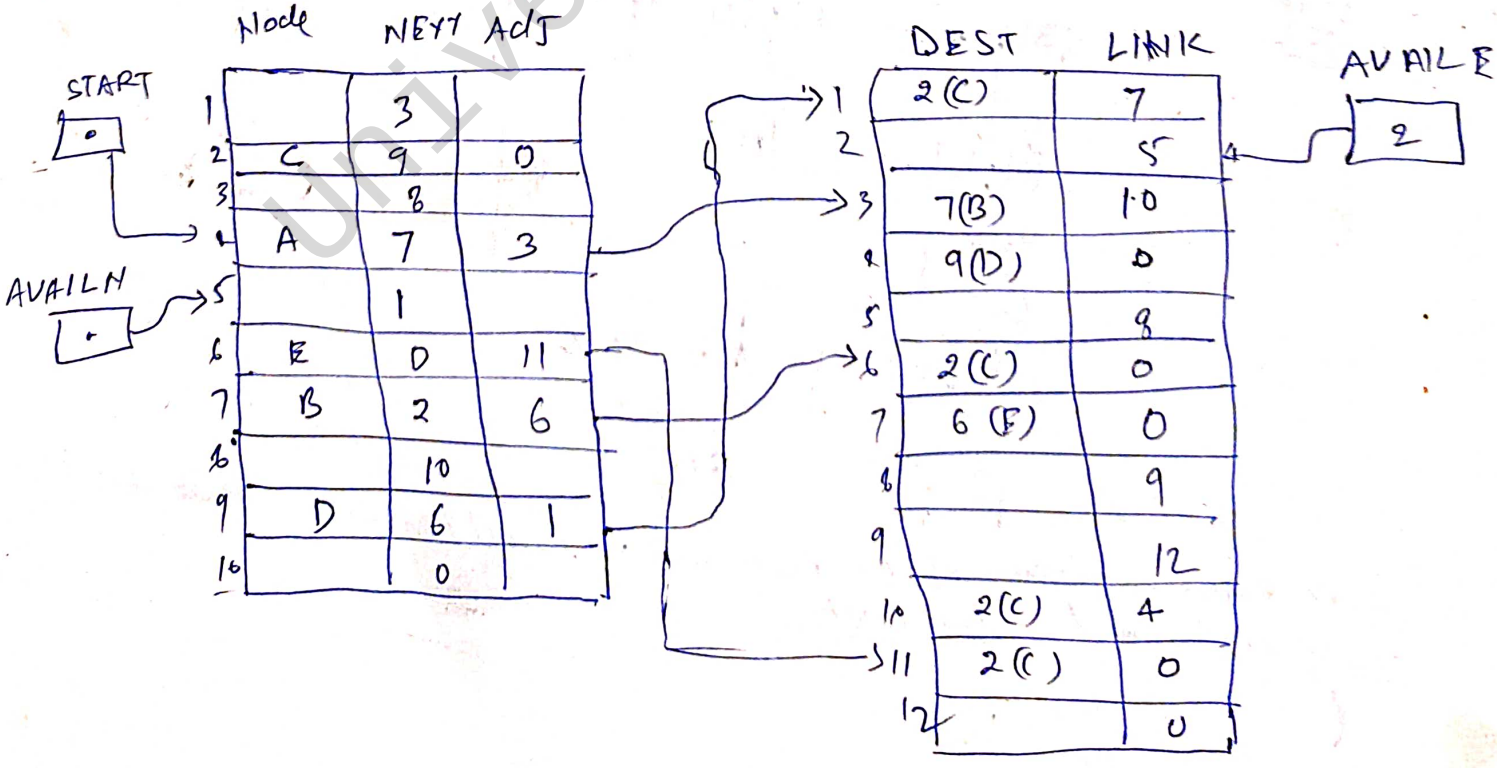
shaded area indicate that there may be other information in the record such as degree of node.

Edge list :- each element in the list will correspond to an edge of G. and will be a record of the form.

DEST	LINK	

DEST - will point the location in list NODE of destination or terminal node of the edges.

LINK - will link together the edges with the same initial node. the node in the same adjacency list.



Traversing A Graph.

There are two standard ways to traverse a graph.

1- Breadth - First search. → uses queue.
2- Depth - First search. → uses stack.

During the execution of both algorithm each node in graph will in one of the three ~~states~~ status..

STATUS = 1: (Ready state). Initial state of the node N

STATUS = 2: (waiting status). the node N is on the Queue or stack, waiting to be processed.

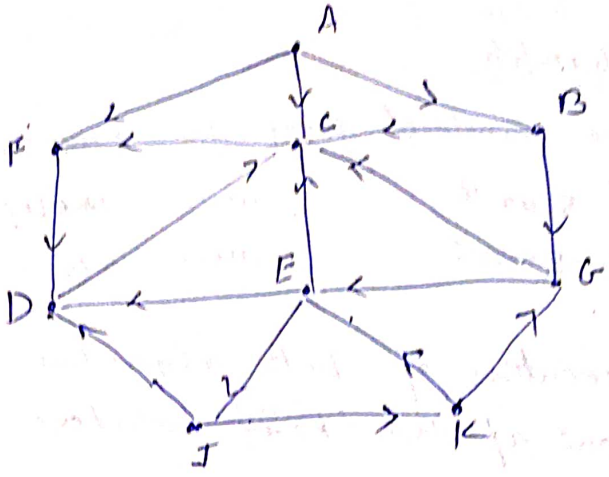
STATUS = 3: (Processed state) the node has been processed.

1- Breadth - First search: (BFS)

Algorithm: This algorithm executes a breadth-First search on a graph G beginning at a starting node A.

1- Initialize all node to be ready state (STATUS = 1)
2- Put the starting node A in QUEUE and change its state to (STATUS = 2).
3- Repeat step 4 and 5 until QUEUE is empty.
4- Remove the FRONT node A of QUEUE. Process A and change the status of A to (STATUS = 3)
5- Add to the REAR of QUEUE all the neighbors of A that are in the ready state (STATUS = 1). and change their status to the waiting status (STATUS = 2)
6- exit.

e.g



Adjacency list

A	F C B
B	G C
C	F
D	C
E	D C J
F	D
G	C E
J	D K
K	E G

traversing from A ⟹ J

(a) Initially add A to QUEUE and NULL to ORIG as follows

FRONT = 1 QUEUE : A
REAR = 1 ORIG : ϕ

(b) Remove A and add to QUEUE the neighbors of A.

FRONT = FRONT + 1 since
FRONT = 2 QUEUE : A F C B
REAR = 4 ORIG : ϕ A A A

(c) Remove the front element F and insert neighbors of F to QUEUE

FRONT = 3 QUEUE : A F C B D
REAR = 5 ORIG : ϕ A A A F

(d) Remove C from QUEUE and add neighbors of C.

FRONT = 4 QUEUE : A F C B D
REAR = 5 ORIG : ϕ A A A F

(e) Remove B and add neighbors to Que.

FRONT = 5 QUEUE : A F C B D G
REAR = 6 ORIG : ϕ A A A F B

(f) Remove D and add neighbors.

FRONT = 6 QUEUE : A F C B D G
REAR = 6 ORIG : ϕ A A A F B

(g) Remove G and add neighbors.

FRONT = 7 QUEUE : A F C B D G E
REAR = 7 ORIG : ϕ A A A F B G

(h) Remove E

FRONT = 8 QUEUE : A F C B D G E J
REAR = 8 ORIG : ϕ A A A F B G E

we stop as soon as J is added to Queue since J is our final destination.

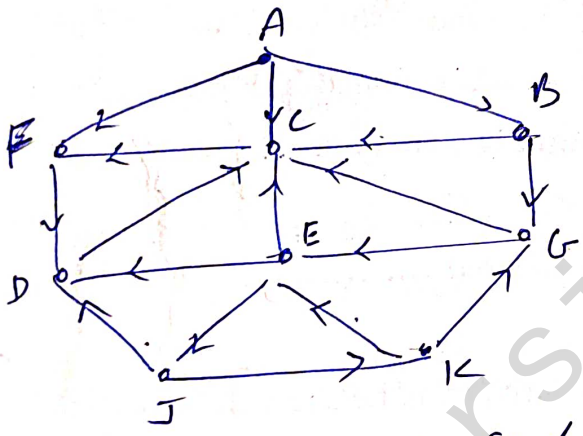
Path from A → J is (J ← E ← G ← B ← A) Ans

Depth - First - Search.

Algorithm: beginning node A.

1- Initialize all nodes to the Ready state (STATUS = 1)
2. Push the starting node A to STACK and change its state (STATUS: 2)
3. Repeat step 4 and 5 until stack is empty.
4- Pop the top node A of STACK. Process A and change its status to the (STATUS: 3)
5- Push onto to STACK all the neighbors of A that are still in the ready state; and change their status to the waiting (STATUS = 2)

6 exit



Adjacency list
A : F C B
B : G C
C : F
D : G
E : D C J
F : D
G : C E
J : D K
K : E G

Suppose we want to find and print all the nodes reachable from the node J.

(a) Initial Push J onto STACK
STACK: J

(b) Pop J and push neighbors of J
Print J , STACK: D, K

(c) Pop K and Push neighbors of K
Print J K STACK: D E G

(d) Pop G and Push neighbor of G
Print J K G STACK: D E C

e). Pop C and Push neighbors of C
Print J, K, G, C STACK: D E F

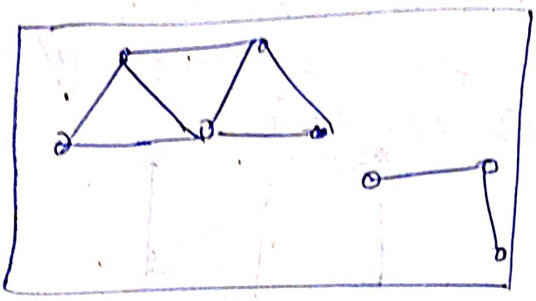
(f) Pop F and push neighbors of F
Print J, K, G, C F STACK: D E

(g) Pop E and push neighbor of E
Print J K G, C, F E, STACK: D

(h) Pop D and Push neighbor of D
Print J K G C F E D

Strongly Connected Component

A undirected graph is not connected is naturally decomposition in several connected component



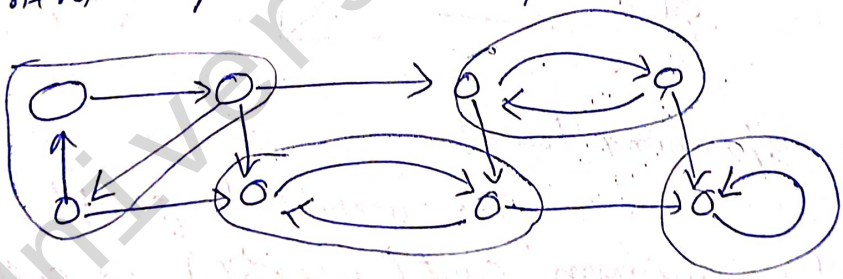
In a directed graph the component is ~~of~~ graph. is connect i.e. the path from u → v the are also a path u → v also from v → u). is called strongly connected component.

~~A directed graph is acyclic~~

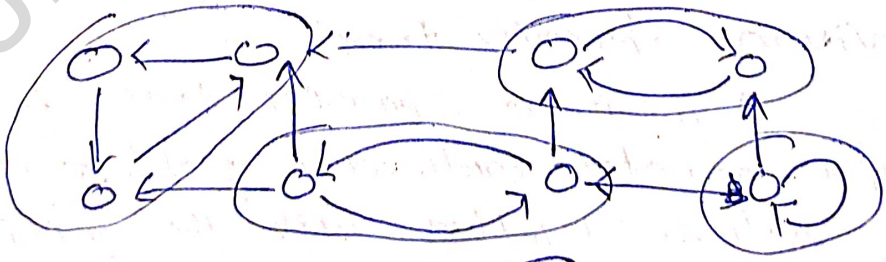
Now consider directed graph : decomposition a directed graph in to its strongly connected components. this a classical application of DFS.

Here we can see the decomposition using two ~~if~~ DFS. after decomposition the ~~are~~ Algorithm is run separately on each strongly connected component. the solutions are then combined according to the structure component ~~of a directed~~ of connection bet'n components.

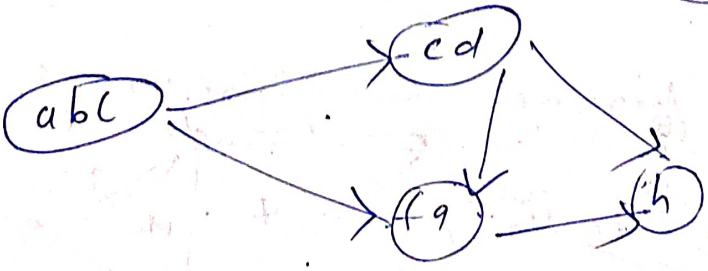
G:



G^T

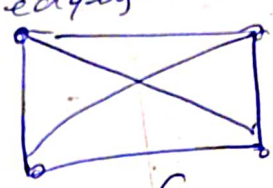


C.

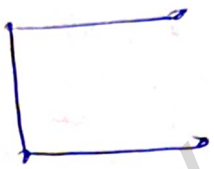


Spanning tree.

A tree T is said to be spanning tree of a connected graph G if T is a sub graph of G and T contains all vertices of G. All spanning tree have exactly (V-1) edges.

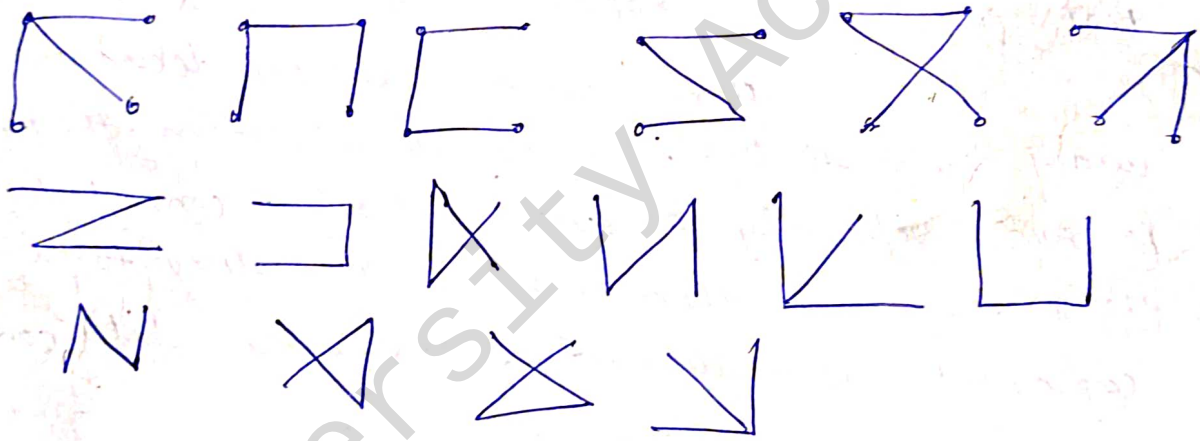


G



spanning tree.

for a complete graph with n vertex. there are n^{n-2} spanning tree.

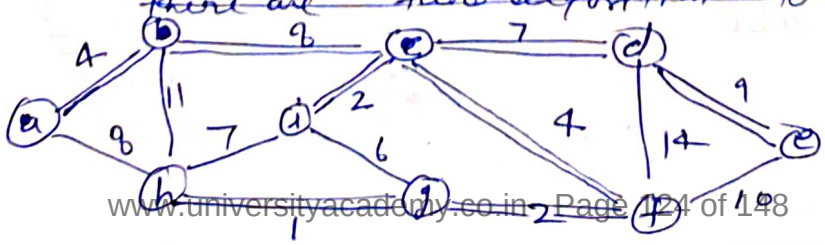


Minimum spanning tree!

In weighted graph, the spanning tree with minum cost (weight) is called minimum spanning tree.

A minimum spanning tree is a spanning tree of a connected undirected graph. it connects all the vertices togather with the minimum total weight.

~~there are two algorithm to finding~~



There are two algorithm to finding MST of graph. (7)

1- Kruskal's algorithm.

2. Prim's algorithm.

Kruskal's Algorithm

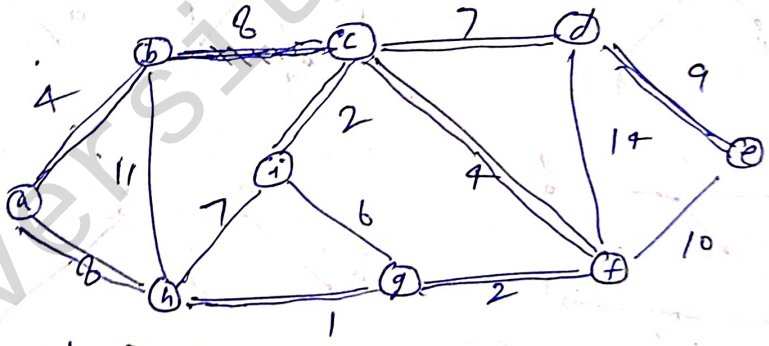
Procedure:

1- Sort all the edges in non-decreasing order of their weight.

2. Pick the smallest edges. check if it's form a cycle with spanning tree discard it otherwise include this edge.

3- Repeat step. 2 until there are V-1 edges in the spanning tree.

eg:



Sorted edges list

(h, g) = 1
(i, c) = 2
(g, f) = 2
(a, b) = 4
(c, f) = 4
(i, g) = 6
(h, i) = 7
(c, d) = 7
(b, c) = 8
(a, h) = 8
(d, e) = 9
(e, f) = 10
(h, b) = 11
(d, f) = 14

end

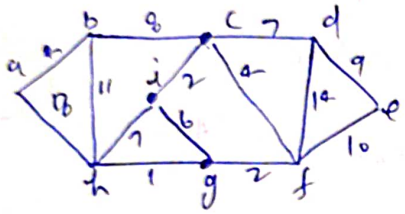
Prim's Algorithm:

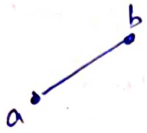
The prims algorithm may informally be described as performing the following steps.

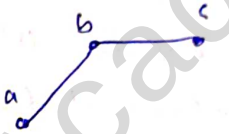
1- Initialize a tree with a single vertex, chosen arbitrarily from the graph.

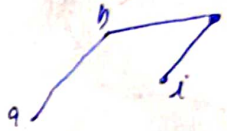
2- Grow the tree by one edge: of the edges that connect tree to vertices not yet in the tree, find, minimum-weight edges and transfers it to the tree.

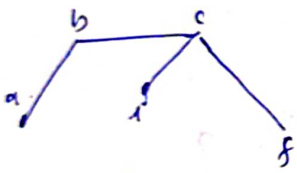
3- Repeat setep 2 until all vertices are in the tree.

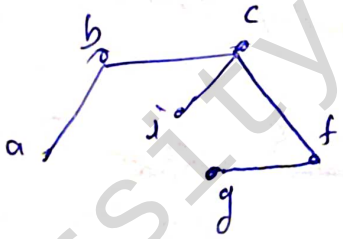


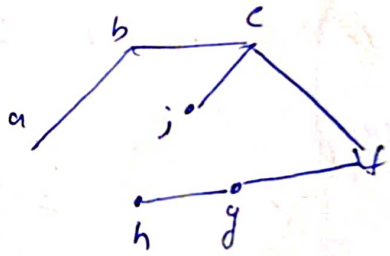


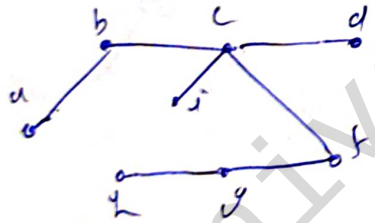


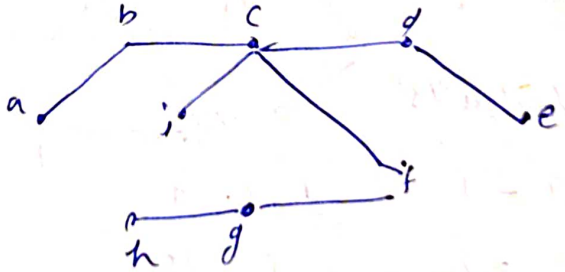




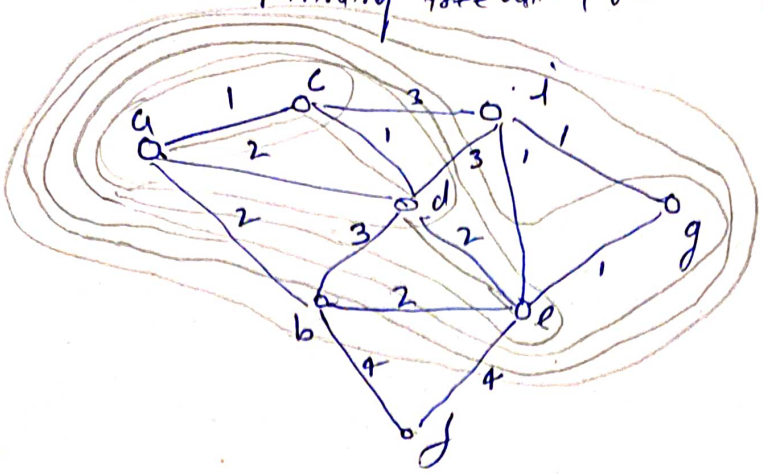


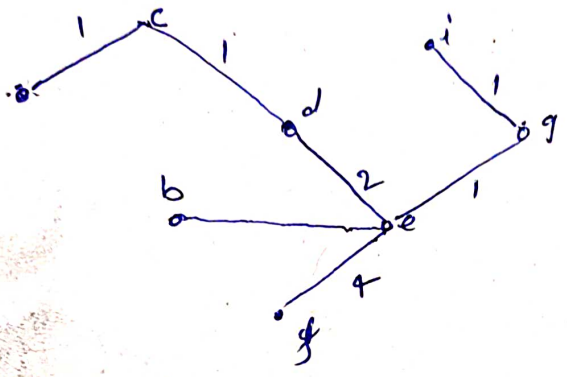




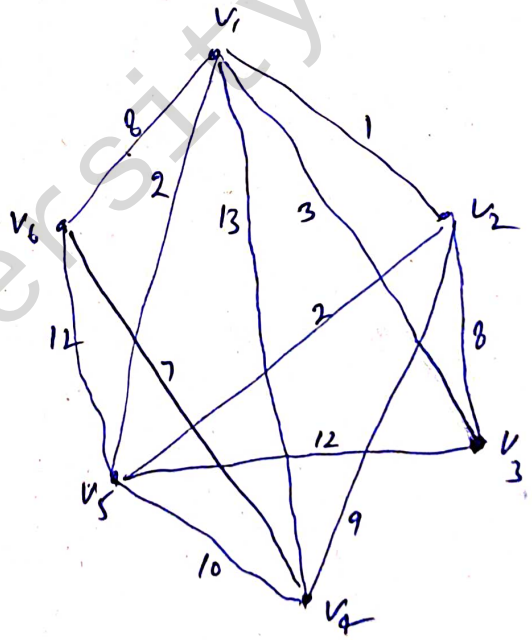


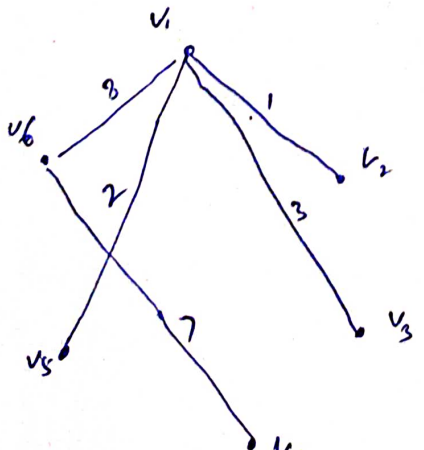
Q:- find minimum spanning tree using prims algorithm:





Q:- find the minimum spanning tree using kruskal algorithm.



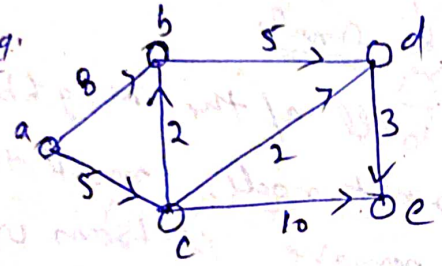


Single source - shortest paths.

The shortest path problem is the problem of finding a path between two nodes in a graph directed weighted graph such that the sum of weight of its constituent edges is minimized.

Single source shortest-path problem: it's the problem of finding shortest path from a given source vertex $s \in V$ to each vertex $v \in V$.

eg.

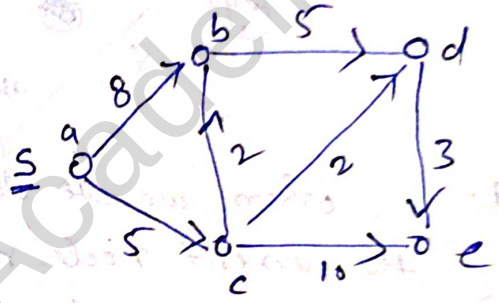


there are two path from c to e. (i) $c \xrightarrow{10} e$ $w = 10$

(ii) $c \xrightarrow{2} d \xrightarrow{3} e$ $w = 5$

the $c \to d - e$ is shortest path. from c to e.

The path fromm $c \to a$ No path. since $w = \infty$.



shortest path from

$s(a) \to b$: $\{s \to c \to b\}$

$s \to d$: $\{s \to c \to d\}$.

if there are no path from s to any node the the weight is ∞.

Single source shortest Algorithm: -

1- Bellman-Ford-algorithm: in this case edges weight may be negative.

2- Dijkstra Algorithm: in this case edges weight are non negative.

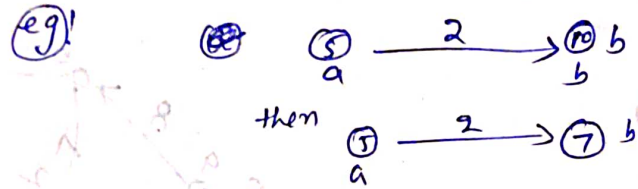
Dijkstra Algorithm.

1- Assign to every node a tentative distance value.
set it to zero for initial node and infinity to others.

2- set initial node as current, mark all other nodes unvisited.
create unvisited list.

3- for the current node, consider all of its unvisited neighbors and calculate their tentative distances.

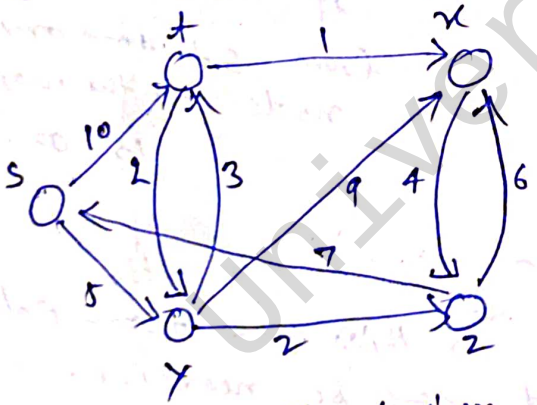
Compare newly calculated tentative distance to current assigned value and assign the smaller one



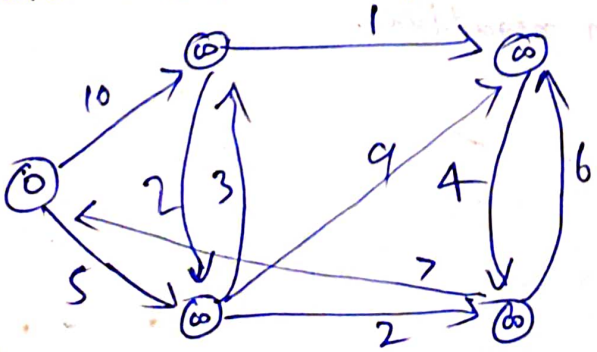
4- when we are done considering all of the neighbors of the current node, mark the current node, mark the current node as visited and remove it from unvisited list.

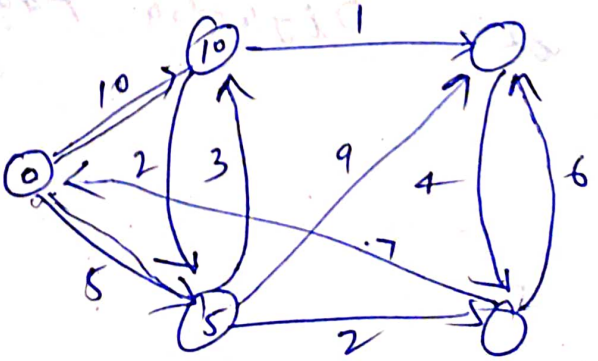
5. if distination node has been marked visited then stop.

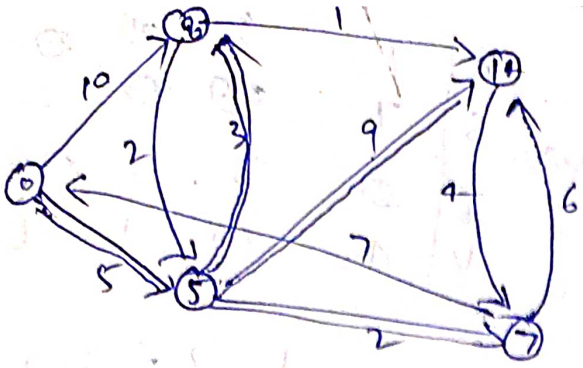
6 otherwise ~~select~~ goto step 3.

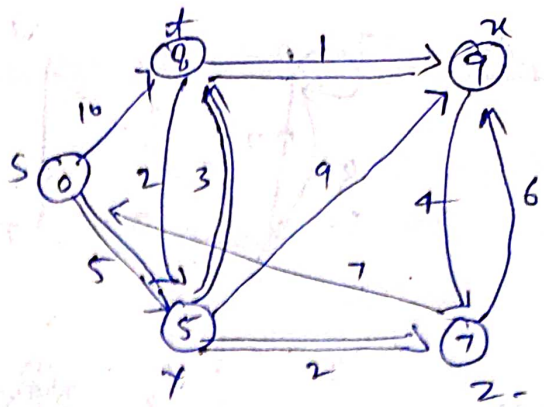


Source is! s if not given
assume source is leftmost vertex!

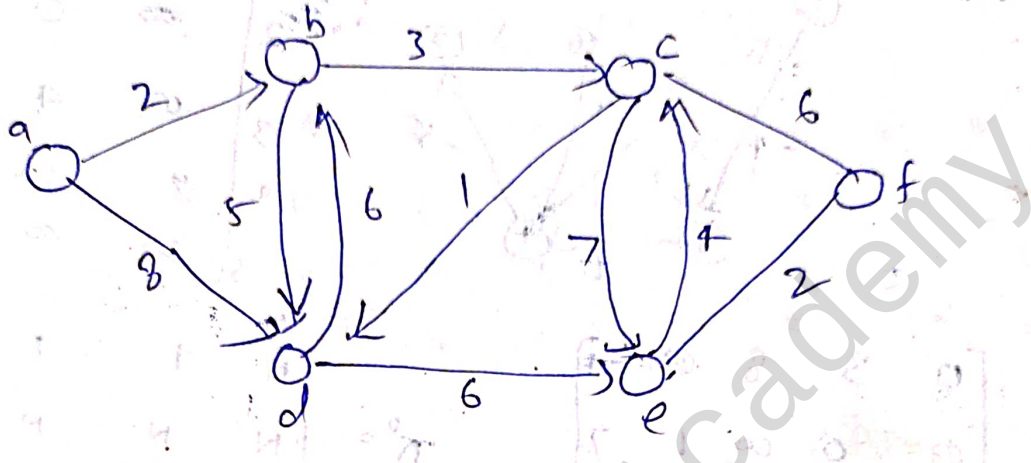








Q:



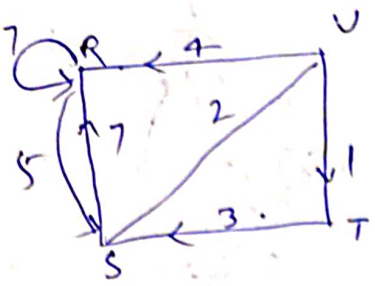
All pair shortest path.

The all pair shortest path problem can be considered the mother of all routing problem. It aim to compute the shortest path from each vertex to every other vertex in the graph may have non-negative weight or negative weight if negative weight is in the graph. then graph contains no negative weight cycle.

To solve the all pair shortest paths problem on weighted directed graph $G = (V, E)$. the resulting algorithm known as the Floyd warshall algorithm.

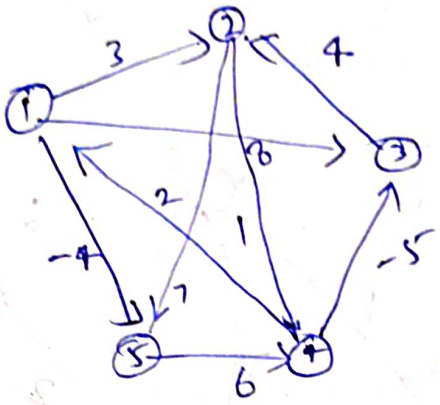
```
FLOYD-WARSHALL(w)
1.  n ← rows[W]
2.  D(0) ← W
3.  for k ← 1 to n
4.      for i ← 1 to n
5.          for J ← 1 to n
6.              do d_ij^(k) ← min(d_ij^(k-1), d_ik^(k-1) + d_kj^(k-1))
7.  Return D(n)
```

example:



$$W = \begin{array}{c} \\ R \\ S \\ T \\ U \end{array} \begin{array}{c} \begin{array}{cccc} R & S & T & U \end{array} \\ \begin{bmatrix} 7 & 5 & \infty & \infty \\ 7 & \infty & \infty & 2 \\ 0 & 3 & 0 & 0 \\ 4 & 6 & 1 & 0 \end{bmatrix} \end{array}$$

example:



$$\begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \end{array} \begin{array}{c} \begin{array}{ccccc} 1 & 2 & 3 & 4 & 5 \end{array} \\ \begin{bmatrix} 0 & 3 & 8 & \infty & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & \infty & \infty \\ 2 & \infty & -5 & 0 & \infty \\ \infty & \infty & \infty & 6 & 0 \end{bmatrix} \end{array}$$

$$D^{(0)} = \begin{bmatrix} 0 & 3 & 8 & \infty & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & \infty & \infty \\ 2 & \infty & -5 & 0 & \infty \\ \infty & \infty & \infty & 6 & 0 \end{bmatrix}$$

$$\pi^0 = \begin{bmatrix} N & 1 & 1 & N & 1 \\ N & N & N & 2 & 2 \\ N & 3 & N & N & N \\ 4 & N & 4 & N & N \\ N & N & N & 5 & N \end{bmatrix}$$

$$D^{(1)} = \begin{bmatrix} 0 & 3 & 8 & \infty & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & \infty & \infty \\ 2 & 5 & -5 & 0 & \infty 2 \\ \infty & \infty & \infty & 6 & 0 \end{bmatrix}$$

$$\pi^{(1)} = \begin{bmatrix} N & 1 & 1 & N & 1 \\ N & N & N & 2 & 2 \\ N & 3 & N & N & N \\ 4 & 1 & 4 & N & 1 \\ N & N & N & 5 & N \end{bmatrix}$$

$$D^{(2)} = \begin{bmatrix} 0 & 3 & 8 & 4 & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & 5 & 11 \\ 2 & 5 & -5 & 0 & -2 \\ \infty & \infty & \infty & 6 & 0 \end{bmatrix}$$

$$\pi^2 = \begin{bmatrix} N & 1 & 1 & 2 & 1 \\ N & N & N & 2 & 2 \\ N & 3 & N & 2 & 2 \\ 4 & 1 & 4 & N & 1 \\ N & N & N & 5 & N \end{bmatrix}$$

$$D^{(3)} = \begin{bmatrix} 0 & 3 & 8 & 4 & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & 5 & 11 \\ 2 & -1 & -5 & 0 & -2 \\ \infty & \infty & \infty & 6 & 0 \end{bmatrix}$$

$$\pi^3 = \begin{bmatrix} N & 1 & 1 & 2 & 1 \\ N & N & N & 2 & 2 \\ N & 3 & N & 2 & 2 \\ 4 & 3 & 4 & N & 1 \\ N & N & N & 5 & N \end{bmatrix}$$

$$D^{(4)} = \begin{bmatrix} 0 & 3 & -1 & 4 & -4 \\ 3 & 0 & -4 & 1 & -1 \\ 7 & 4 & 0 & 5 & 3 \\ 2 & -1 & -5 & 0 & -2 \\ 8 & 5 & 1 & 6 & 0 \end{bmatrix} \qquad \pi^4 = \begin{bmatrix} N & 1 & 4 & 2 & 1 \\ 4 & N & 4 & 2 & 1 \\ 4 & 3 & N & 2 & 1 \\ 4 & 3 & 4 & N & 1 \\ 4 & 3 & 4 & 5 & N \end{bmatrix}$$

$$D^5 = \begin{bmatrix} 0 & 1 & -3 & 2 & -4 \\ 3 & 0 & -4 & 1 & -1 \\ 7 & 4 & 0 & 5 & 3 \\ 2 & -1 & -5 & 0 & -2 \\ 8 & 5 & 1 & 6 & 0 \end{bmatrix} \qquad \pi^5 = \begin{bmatrix} N & 3 & 4 & 5 & 1 \\ 4 & N & 4 & 2 & 1 \\ 4 & 3 & N & 2 & 1 \\ 4 & 3 & 4 & N & 1 \\ 4 & 3 & 4 & 5 & N \end{bmatrix}$$

Constructing a path:

Q9.

1- Path from node 2 to 3.

$3 \leftarrow 4 \leftarrow 2 \quad = -4$

2. path from node 4 to 5.

$5 \leftarrow 4 \leftarrow \quad = -5$

3. path $5 \leftarrow 2$

$2 \leftarrow 3 \leftarrow 4 \leftarrow 5 \quad = 5$

4- path from 3 to 5

$5 \leftarrow 1 \leftarrow 4 \leftarrow 2 \leftarrow 3 \quad =$

5 path from 4 to 3

$3 \leftarrow 4 \quad = -5$

6- path from 1 to 3

$3 \leftarrow 4 \leftarrow 5 \leftarrow 1 \quad = 3$

7- path from 2 to 4

$4 \leftarrow 2$

8- path from 5 to 1

$1 \leftarrow 4 \leftarrow 5 \leftarrow \quad = 8$

UNIT-5

Searching & sorting

Searching: Searching refers to the operation of finding the location of item in array.

1- linear search. — O(n)
2- Binary Search. O(log n)

1- Linear ~~Ass~~ Search

(Linear search) LINEAR (DATA, N, ITEM, LOC).

1- [Insert ITEM at end of DATA] set DATA[N+1] := ITEM
2- set LOC := 1.
3- Repeat while DATA[LOC] ≠ ITEM
   set LOC = LOC+1
4- [successfully] if LOC = N+1, then set LOC = 0
5 exit.

NAME

(2) Binary Search ( )

The complexity is measured by number of f(n) comparission observe that each comparision Reduce the size in half. he we required at most f(n) comparision to locate item ~~in~~ in array of size n. $2^{f(n)} > n$ comparision

taking log both side

$\log f(n) > \log n$

$f(n) = \log n$ + constant.

so $\Theta(\log n)$

Question How many comparision Required for binary search in an array containing 1000000 elem

$$f(n) = \log_2 1000000$$
$$= 20$$

Limitation

(1) the list must be sorted

(2) to store data in sorted order we may used different data structure such as Binary Search tree.

Sorting.

	Best	avg	worst
1- Bubble sort	$O(n)$	$O(n^2)$	$O(n^2)$
2- Insertion sort	$O(n)$	$O(n^2)$	$O(n^2)$
3- Selection sort	$O(n^2)$	$O(n^2)$	$O(n^2$
3 Quick sort	$n \log n$	$n \log n$	n^2
4- merge sort	$n \log n$	$n \log n$	$n \log n$
5- Heap sort	$n \log n$	$n \log n$	$n \log n$
6- Radix sort	$O(nk)$	$O(nk)$	$O(nk)$

1- Bubble sort: Analysis

blows last at elem at last

first pass $n-1$ comparision
second pass $n-2$ com ---
:
! comparision.

$$f(n) = (n-1) + (n-2) + \cdots 2+1 = \frac{n(n-1)}{2} = \frac{n^2+n}{2}$$
$$= O(n^2)$$

(2) INSERTION SORT:

Suppose an Array A with n element A[1], A[2]...A[N] is in memory. The Insertion sort Algorithm scans A from A[1] → A[N], inserting each element A[K] into its proper position: that is

pass 1: A[1] is sorted.

pass 2: A[2] is inserted either before or after A[1] so A[1], A[2] is sorted

pass 3: A[3] is inserted either before A[1], between A[1] and A[2] or after A[2] so the A[1] [2] A[3] are sorted.

:

Pass N.

Algorithm:

INSERTION (A, N)

1. Set A[0] := -∞
2. Repeat steps 3 to 5 for K = 2, 3, ... N
3. set TEMP := A[K] and PTR := K-1
4. Repeat while TEMP < A[PTR]
   (a) Set A[PTR+1] := A[PTR]
   (b) Set PTR := PTR - 1
5. Set A[PTR+1] = TEMP
6. Return.

Complexity of Insertion Sort:

$$f(n) = 1 + 2 + \cdots + (n-1) = \frac{n(n-1)}{2}$$

$$= O(n)^2$$

exam:

77, 33, 44, 11, 88, 22, 66, 55

Pass	A[0]	A[1]	A[2]	A[3]	A[4]	A[5]	A[6]	A[7]	A[8]
K=1	-∞	(77)	33	44	11	88	22	66	55
K=2	-∞	77	(33)	44	11	88	22	66	55
K=3	-∞	33	77	(44)	11	88	22	66	55
K=4	-∞	33	44	77	(11)	88	22	66	55
K=5	-∞	11	33	44	77	(88)	22	66	55
K=6	-∞	11	33	44	77	88	(22)	66	55
K=7	-∞	11	22	33	44	77	88	(66)	55
K=8	-∞	11	22	33	44	66	77	88	(55)
Sorted	-∞	11	22	33	44	55	66	77	88

$S_n = \frac{n}{2}[2a + (n-1)d]$

a = 1

d = 1

upto N term $= \frac{n}{2}[2 + (n-1)1]$

$=$

SELECTION SORT

Suppose an array A with n element $A[1], A[2], \ldots A[N]$ is in memory selection sort algorithm for sorting works as follows.

pass 1 : Find the location LOC of smallest element in list. $A[1], A[2] \ldots A[N]$ and then Interchange $A[LOC]$ and $A[1]$ then $A[1]$ is sorted.

pass 2 : Find the location LOC of smallest in the sublist $N-1$ $A[2], A[3], \ldots A[N]$ and then Interchange $A[LOC]$ and $A[2]$ then $A[1], A[2]$ is sorted since $A[1] \leq A[2]$

pass N-1 Find the loc of smaller element $A[N-1]$, $A[N]$ and then Interchange $A[LOC]$ and $A[N-1]$ then $A[1], A[2], \ldots A[N]$ is sorted.

SELECTION (A, N)

1- Repeat step 2 and 3 for K = 1, 2, ... N-1
2 Call MIN (A, K, N, LOC)
3- [Interchang A[K] and A[LOC]]
Set TEMP = A[K], A[K] = A[LOC] and A[LOC] = TEMP
4 exit.

77, 33, 44, 11, 88, 22, 66, 55

PASS	A[1]	A[2]	A[3]	A[4]	A[5]	A[6]	A[7]	A[8]
K=1 LOC=4	(77)	33	44	(88)	88	22	66	55
K=2, LOC=6	11	(33)	44	77	88	22	66	55
K=3, LOC=6	11	22	(44)	77	88	(33)	66	55
K=4, LOC=6	11	22	33	(77)	88	(44)	66	55
K=5 LOC=8	11	22	33	44	(88)	77	66	(55)
K=6 LOC=7	11	22	33	44	55	(77)	(66)	88
K=7 LOC=7	11	22	33	44	55	66	(77)	88

Analysis: observe min(A, K, N, LOC) requires

$f(n) = (n-1) + (n-2) + \ldots 2 + 1$

$= \frac{n(n-1)}{2}$

$= O(n^2)$

Quick sort

Quicksort is based on divide and conquer method. The three step of divide and conquer process for sorting a array $A[p \ldots r]$.

Divide: Partition the array $A[p \ldots r]$ into two subarray $A[p \ldots q-1]$ and $A[q+1 \ldots r]$ such that $A[p \ldots q-1] \leq A[q] \leq A[q+1 \ldots r]$

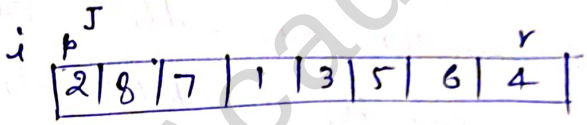
Conquer: sort the two subarray recursively.

Combine: since the subarray are sorted in place, so no work is needed to combine them: the entire array $A[p \ldots r]$ is now sorted.

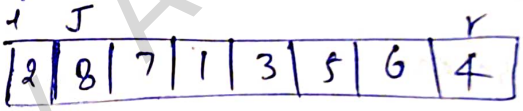
```
QUICKSORT(A,p,r)
1- if p<r
2- then q ← Partition(A,p,r)
3       QUICKSORT(A,p,q-1)
4       QUICKSORT(A,q+1,r)
```

```
PARTITION(A,p,r)
1- x ← A[r]
2- i ← p-1
3- for j ← p to r-1
4      do if A[j] ≤ x
5           then i ← i+1
6                exchange A[i] ↔ A[j]
7  exchang A[i+1] ↔ A[r]
8  return i+1
```

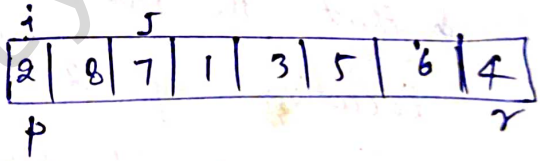
(a) $x = 4$



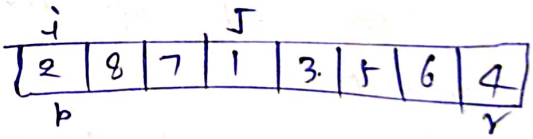
(b)



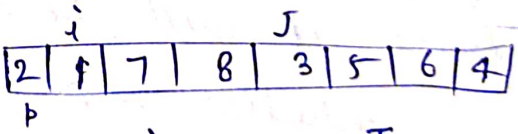
(c)



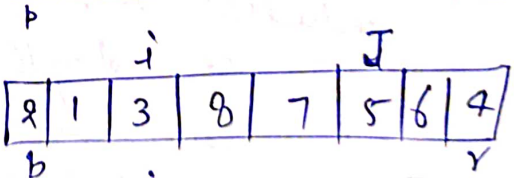
(d)



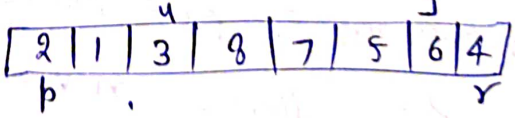
(e)



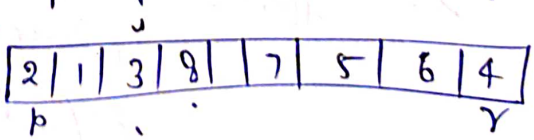
(f)



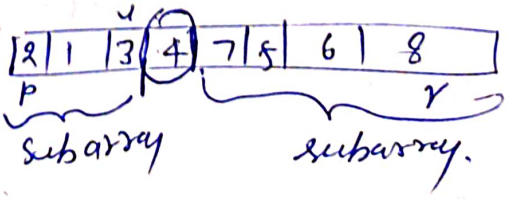
(g)



(h)



(i)



Analysis of quicksort.

the worst case occurs when list is already sorted. then the first sublist will be empty but ~~first~~ second sub list will have n-1 element, ~~so~~ and so on.

$$f(n) = n + (n-1) + \cdots 2+1 = \frac{n(n+1)}{2} = O(n^2)$$

but in average case complexity will calculated as:

(1) Reducing the initial list places 1 element and produce two sublist

(2) Reducing the two sublist places 2 element and produce 4 sublists

(3) Reduce the four sublist places 4 element and produce 8 sublists

(4) Reducing the 8 sublist places 8 element and produces 16 sublist.

and so on.

Observe that the reduction step in the k^{th} level find the location of 2^{k-1} element.

eg. of level (4) we find the location of $2^{4-1} = 2^3 = 8$ element.

Hence it will be approximately $\log_2 n$ level of reduction step.

eg. $\log_2 16 = 4$

further more, each level uses at most n comparision.

so $f(n) = n * \log n$

$= n \log n$.

$= O(n \log n)$.

Merge sort:

merge sort is based on divide and conquer paradigm.

the three steps of divide and conquer process for sorting Array A[p...r]. as follows.

Divide! divide the n element sequence to be sorted into two subsequence of n/2 element

conquer! sort the two subsequences recursively using merge sort.

combine! merge the two sorted subsequence to produce the sorted array.

example

66, 33, 40, 22, 55, 88, 60, 11, 80, 20, 50, 44, 77, 30.

Sloor

66, 33, 40, 22, 55, 88, 60 | 11, 80, 20, 50, 44, 77, 30

66, 33, 40, 22 | 55, 88, 60 | 11, 80, 20, 50 | 44, 77, 30

66, 33 | 40, 22 | 55, 88 | 60 | 11, 80 | 20, 50 | 44, 77 | 30

66 | 33 | 40 | 22 | 55 | 88 | 60 | 11 | 80 | 20 | 50 | 44 | 77 | 30

33, 66 | 22, 40 | 55, 88 | 11, 60 | 20, 80 | 44, 50 | 30, 77

22, 33, 40, 66 | 11, 55, 60, 88 | 20, 44, 50, 80 | 30, 77

11, 22, 33, 40, 55, 60, 66, 88 | 20, 30, 44, 50, 77, 80

Ans 11, 20, 22, 30, 33, 40, 50, 55, 60, 66, 77, 80, 88

Analysis of merge sort.

the algorithm require $\log_2 n$ pass to sort n element and each pass atmost n comparison done. so

$$f(n) = n \log n.$$

$$= O(n \log n)$$

Heap Sort.

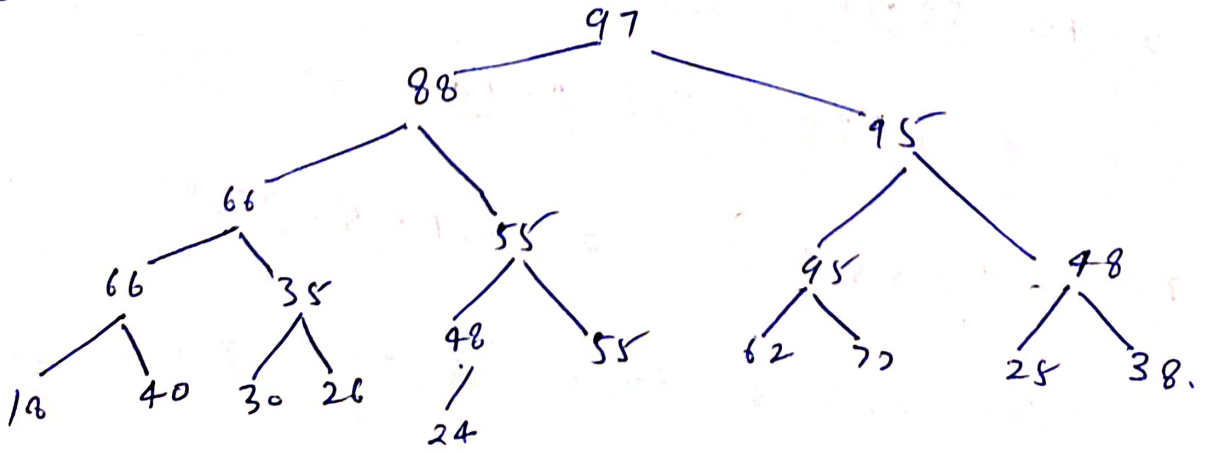
there is a special tree structure called heap. The heap is used in an elegant sorting algorithm called heap sort.

Suppose H is a complet tree with n element. Then H is called heap or a max heap if each node N of heap has the following property:

the value at N is greater than or equal to the value at each of children N.

A Min heap defined as the value of N is less than or equal to the value at any of the children of N.

eg.



Deleting the Root of a Heap:

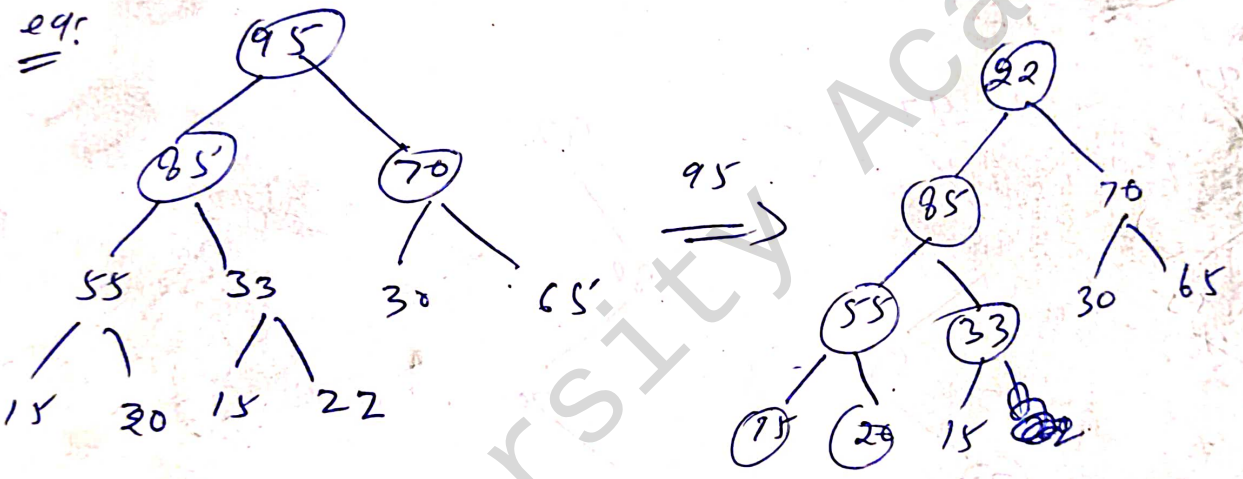
Suppose a heap with N element and suppose we want to delete the R of H.

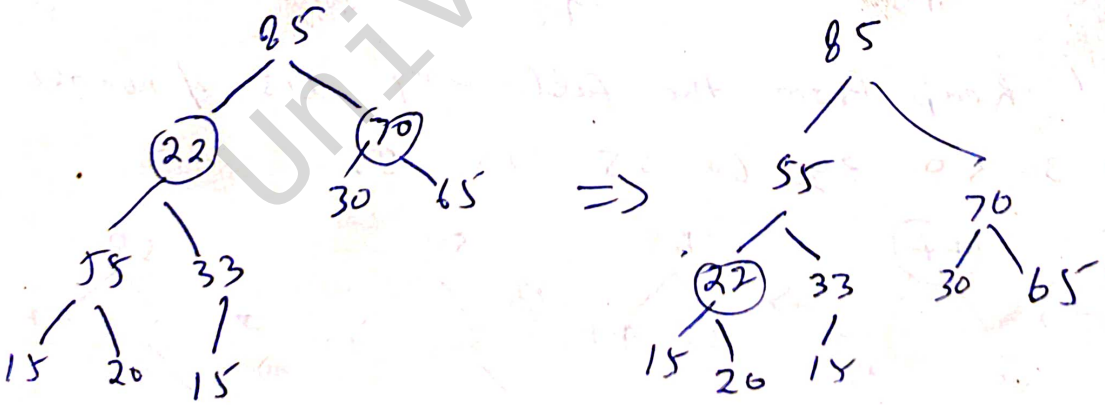
1- Assign Root R to same variable ITEM

2. Replace the deleted node R by the last node(L) of Heap(H) so H is still complete tree but not necessarily heap.

3- (Reheap) let L sink to its appropriate place so H is finally a heap.

eg:





Heapsort Algorithm to sort Array A consist of two following phases

1 ~~Phase A~~ : Build a heap H out of the element A

2 ~~Phase B~~ Repeatedly delete Root of H.

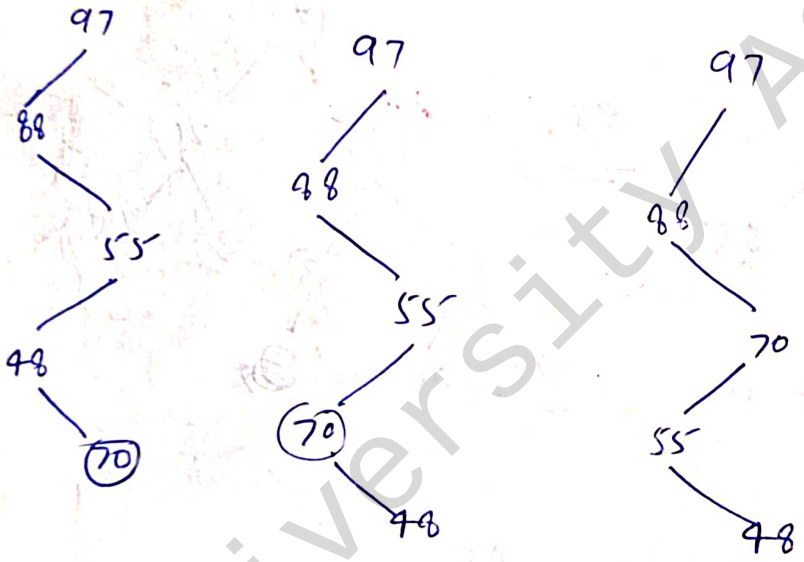
Inserting into heap

Suppose H is a heap with N element and suppose an ITEM of information is given. we insert ITEM into heap H as follows:

1- First adjoin ITEM at the end of H so H is still complet tree but necessarily a heap.

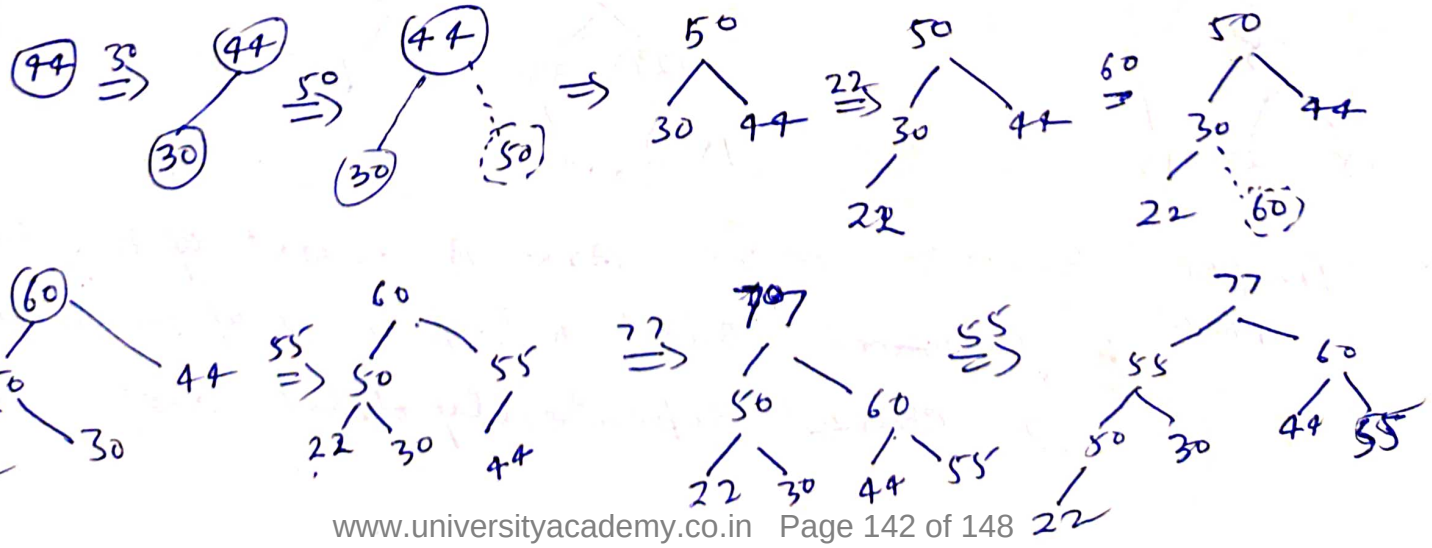
2. Then let ITEM rise to it's appropriate place, in Heap so the H is finally heap.

e.g. Consider the ~~this~~ above heap structure, suppose we want to add ITEM = 70 to H.



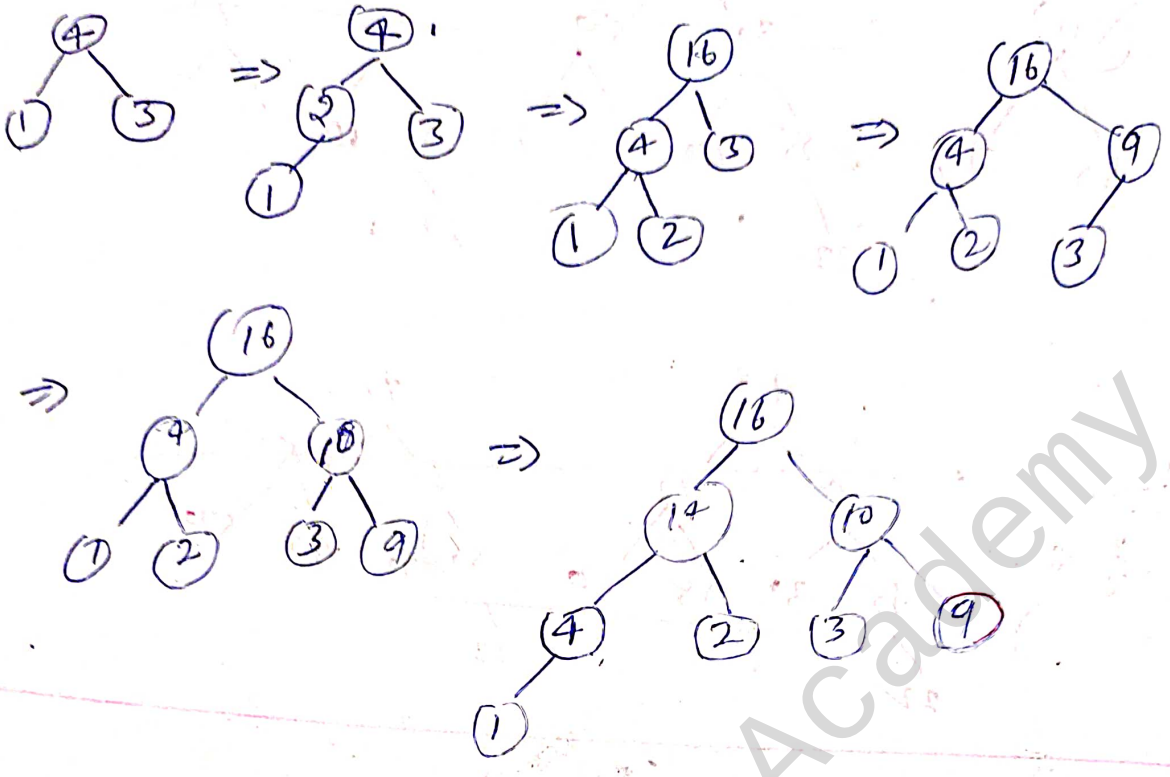
Caution Build heap from the following list of numbers

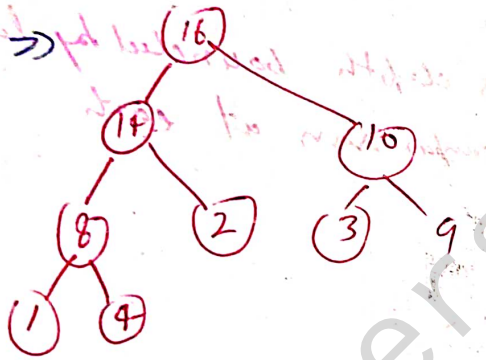
44 30 50 22 60 55 77 55

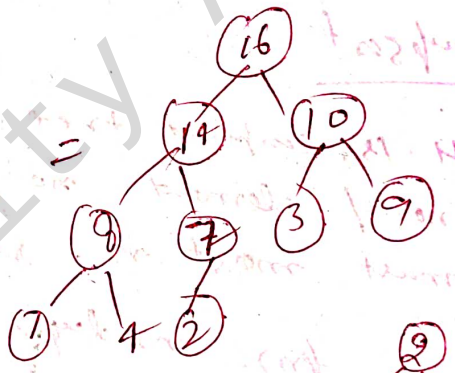


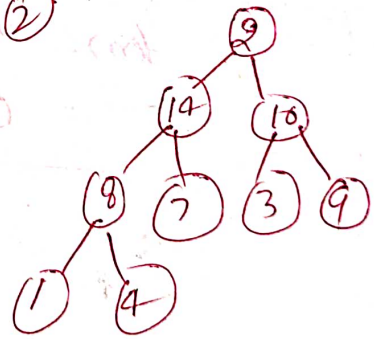
Heap Sort:

4, 1, 3, 2, 16, 9, 10, 14, 8, 7

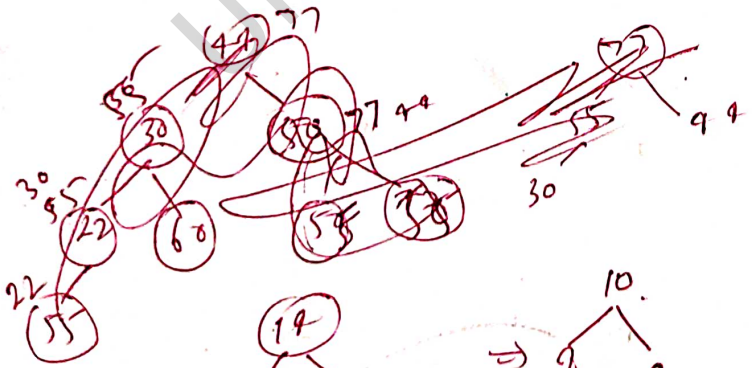


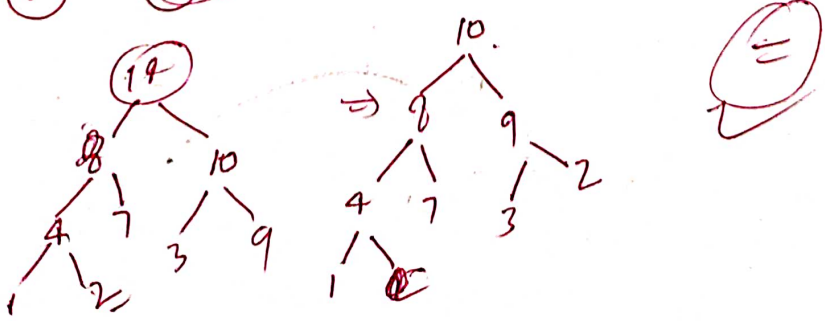






1, 14, 16





Analysis of Heapsort.

Since H is complete tree its depth bounded by $\log n$ where n is no. of element. no of comparisons at each depth at most most n. so

$$f(n) = n \times \log n$$
$$= O(n \log n)$$

Radix Sort

Radix sort is the method that many people intuitively use or begin to use when alphabetizing a large list of name. that is the names are arranged in 26 classes where the first class consist of those name begin with "A" and second class consist of those name that begin with "B" and so on.

Example.

Suppose 9 cards numbers as follows

348, 143, 361, 423, 538, 128, 321, 543, 366

input	0	1	2	3	4	5	6	7	8	9
348					143				348	
143										
361		361								
423				423	423				538	
538									128	
128										
321		321								
543				543						
366							366			

Pass 1

input	0	1	2	3	4	5	6	7	8
361							361		
321			321		143				
143									
423			423		543		366		
543									
366					348				
348				538					
538									
128			128						

Pass 2

Input	0	1	2	3	4	5	6	7	8	9
321				321						
423			423		423					
128		128				538				
538										
143		143				543				
543										
398				398						
361				361						
366				366						

a. In the first pass the unit digit are sorted in pocket

(b) In the second pass tens digit are sorted into pocket

(c) Third pass hundred digit are sort in to pocket

Complexity of Radix

Suppose a list of n item A_1, A_2, A_n given. Let d denote the Radix (eg. d=10 for decimal, d=26 for letters. d= 2 for bits) and suppose each item A_i.

The Radix sort algorithm will require s passes, the number of digit in each item. pass K will compare each of digit. Hence the number C(n) of comparisions for algorithm follows: $C(n) \leq d * s * n$

Hashing

the searching time of each algorithm discussed so far depends on the number n of elements of list. hashing or hash addressing is searching technique which is essentially independent of the number n.

suppose a company with 68 employees assigns a 4-digit number to each employee which is used as the primary key in company file. use the employee number as the address of the Record in memory. this technique will require 10000 memory location.

the general idea of using key to determine the address of record is an excellent idea. this modification take the form of a function H from set K of keys into L of memory addresses such function..

$$H: K \rightarrow L$$

is called hash function. it is possible that two different K_1 and K_2 yield same hash address. this situation is called collision.

the Hashing is divided into two parts.

(1) hash function.

(2) collision resolutions.

HASH FUNCTION:

there are two principle criteria used in seleting function $K: K \rightarrow L$ as follows,

(1) function H should be very easy and compute quickly.

(2) minimum collisions occurs.

there are some popular hash function are:

(i) division method:

chose number m. larger than

$H(k) = k \pmod m$

(b) midsquire method:

$H(k) = l$

c- folding method:

$K(k) = k_1 + k_2 + k_3 + k_r$

Consider the example:

the compary have 68 employee is assigned unique 4-digit emp. number. suppose L consist two digit memory address: 00, 01 . . . 99.

3205, 7148, 2345

(a) division method: chose the prime number 'm' close to 99. so m=97

$H(3205) = 3205 \bmod 97$

$= 4$

$H(7148) = 67$, $H(2345) = 17$

(b) Mid Square method:

k	3205	7148	2345
k^2:	10272025	51093904	5499025
H(k):	72	93	99

C- Folding method.

$H(3205) = 32+05 = 37$

$H(7148) = 71+48 = 19$,

$H(2345) = 23+45 = 68$