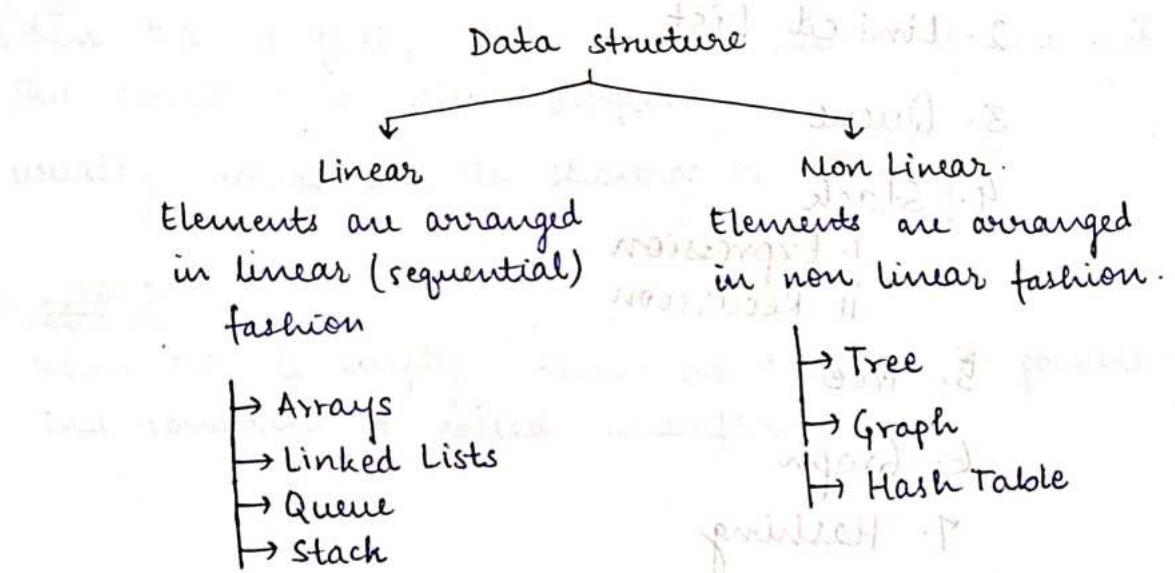


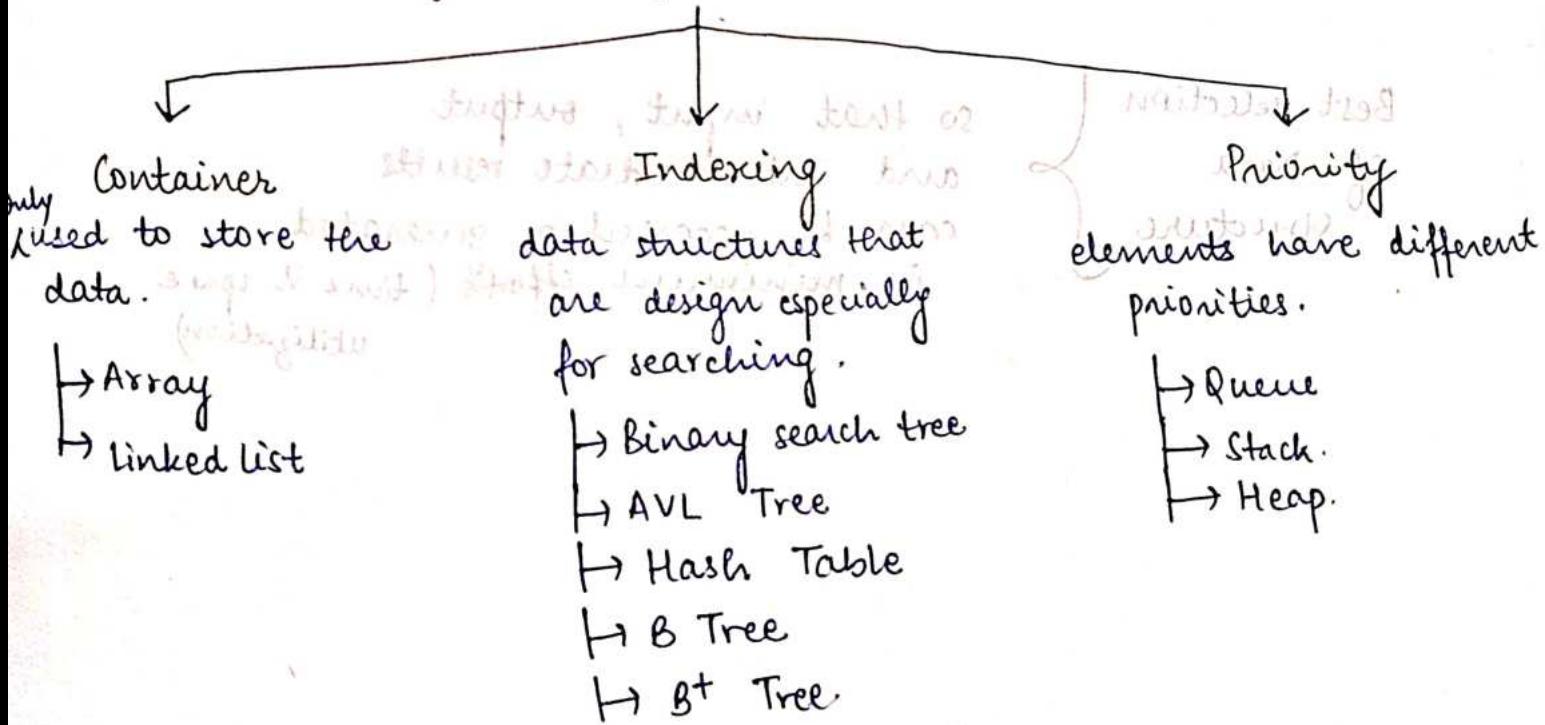
# Data Structures

L-01

Data structure:- Mathematical and logical model of interrelated data  
 predefined structure of organizing data.      operations that can be performed on data.



## Classification of Data Structure



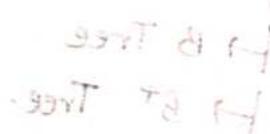
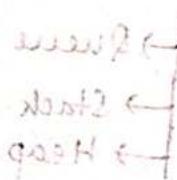
for better representation of data  
looped two dimensional structures also

## Syllabus

1. Arrays
2. Linked List
3. Queue
4. Stack
  - i. Expression
  - ii. Recursion
5. Tree
6. Graph
7. Hashing.

Best selection  
of Data  
Structure

so that input, output  
and intermediate results  
can be accessed or generated  
in minimum efforts (time & space  
utilization)



## Operations on Data Structures

- 1. Traversing :- visiting each element atleast once (only once) & atmost
- 2. Insertion :- adding an element in data structure
- 3. Deletion :- removing an element from a data structure.
- 4. Searching :- finding an element in a data structure.
- 5. Sorting :- arranging elements in ascending or descending order
- 6. Merging :- combining 2 data structures of same type.

### Overflow :-

when D.S. is full, then further insertions are not possible.  
This condition is called overflow.  
usually occurs due to shortage of memory.

### Underflow :-

when D.S. is empty, then, no deletion is possible.  
This condition is called underflow.

## Experiments

### Result of exp

for (int i = 0; i < n; i++)  
    cout << arr[i];

- O(n)

$$S + M = (n)T$$

$$S + M = (n)T$$

= O(n) + O(1) = O(n)

(page 1/2) ~~drawings~~ ~~redundant steps are avoided~~  
Algorithm ~~is a set of instructions to perform a task or to solve a problem -; written & private stat is secret to which we can access -; written & step by step solution for algorithmic problem.~~

~~Analysis of algorithm -~~

To compare multiple algorithms for same problem.

### 1. Space complexity

~~measure of space required by algorithm to generate output.~~

### 2. Run Time complexity

~~measure of time required by the algorithm to generate output~~

### Rate of growth

Measure of rate at which the runtime complexity increases with increment in the input size.

Example-

$$T(n) = n + 2 \quad \text{linear}$$

$$T(n) = n^2 + 5 \quad \text{quadratic}$$

Complexities in ascending order

Rate of growth	complexity
Constant	$1$
Logarithmic	$\log n$
Linear	$(n)$
Logarithmic linear	$n \log n$
Quadratic	$n^2$
Cubic	$n^3$
:	$(n) \Omega$
Exponential	$2^n$
Factorial	$n!$

```
for (i=1; i<n; i=i*2)
    x = m+2;
```

```
for (i=n; i>1; i=i/2)
    x = m+2;
```

complexity =  $\log_2 n$

# Asymptotic Notation

1. Big O :-

- provides tightest upper bound
- worst case complexity (gives idea)

$O(n)$

2. Omega  $\Omega$  :-

- provides tightest lower bound
- best case complexity (gives idea)

$\Omega(n)$

3. Theta  $\Theta$  :-

- provides average bound

If  $O(n)$  and  $\Omega(n)$ , then,  $\Theta(n)$

## Array

- ① Collection of homogeneous elements.
- ② All elements are stored on consecutive memory location.
- ③ All elements can be accessed using a set of indices.

Defining array in C Language -

<datatype> <name> [<size>]

Ex:- int A[5]

- ① Address of starting address of array = Base address (B)
- ② Starting index (Lower Bound)  $L_B = 0$   
Last index (Upper Bound)  $U_B = \text{size} - 1$

Defining array in Data Structure

[General notation to depict array irrespective of language]

name [ $L_B : U_B$ ]

Ex:- A[2:15], B[5:18], C[-2:6], D[-15,-5]

$$\boxed{\text{Size of array} = U_B - L_B + 1}$$

- ① Default Lower Bound in array = 0.

(if not given in ques)

KOTA

Location of element at index i =  $\text{Location}(A[i]) = \text{Base} + w * (i - L_B)$

### Traversing in Array

for ( $i = L_B ; i \leq U_B ; i++$ ) {      }  $L_B \leq i \leq U_B$

(i) visit  $A[i]$ ; Time complexity =  $\Theta(n)$

Space complexity =  $\Theta(1)$

Ques Consider an array  $A[8:16]$ . Total number of elements?

$$\text{Ans: } 16 - 8 + 1 = 9$$

[Explanation]

$$8U \geq 8J$$

Ques Consider an array  $A[-5:18]$  which is stored in array starting from location 1000. Each element takes 4 locations in memory. The address/location of element  $A[5]$  is -

$$L_B = -5$$

$$U_B = 18$$

$$w = 4$$

$$B = 1000$$

$$\text{Location}(A[5]) = \text{Base} + (i - L_B) * w$$

$$= 1000 + [5 - (-5)] * 4$$

$$= 1000 + 10 * 4$$

$$= \underline{\underline{1040}}$$

## Finding minimum element in array -

No. of comparisons needed =  $n-1$ [ $n$  = array size]

$$\min = A[L_B]$$

```
for(i = L_B + 1; i <= U_B; i++) {
    if(A[i] < min) {
```

Runtime complexity =  $\Theta(n)$ 

$$\min = A[i];$$

Space complexity =  $\Theta(1)$ 

\* Minimum element can also be found by Tournament method (where adjacent elements are pairs of 2 are compared and smaller one moves to the next stage).

Tournament method (where adjacent elements are pairs of 2 are compared and smaller one moves to the next stage).

Runtime complexity  $\leq \Theta(n)$  (approx) &

Space complexity  $\approx \Theta(n)$  (approx)

'Max' & 'Min' tournaments (approx) & 'Max' & 'Min' tournaments (approx)

## Finding maximum element in array -

No. of comparisons needed =  $n-1$ 

$$\max = A[L_B]$$

```
for(i = L_B + 1; i <= U_B; i++) {
```

Runtime complexity =  $\Theta(n)$ 

```
    if(A[i] > max) {
```

$$\max = A[i];$$

Space complexity =  $\Theta(1)$ 

$S = \frac{N^2}{2}$  = no. of comparisons for worst case

Finding minimum and maximum with minimum comparisons.

If normal method is used,  $2(n-1)$  comparisons are needed to be done.

Other possible solution can give minimum & maximum in  $\left\lceil \frac{3n}{2} \right\rceil - 2$  comparisons

#### Method -

- ① Create 2 lists — small and large.
- ② Compare adjacent elements in array in pairs of 2.
- ③ Put the smaller of the pair in list small and larger one in list large.

R.T.  
complexity =  $\Theta(n)$

Space  
complexity =  $\Theta(n)$

- ④ Find minimum element from 'small' list and maximum element from 'large' list.

#### Operation

#### No. of comparisons

Preparing 2 lists

$$\frac{n}{2}$$

Finding minimum

$$\frac{n}{2} - 1$$

Finding maximum

$$\frac{n}{2} - 1$$

$$\therefore \text{Total no. of comparisons} = \frac{3n}{2} - 2$$

If  $n$  is odd, last element is copied in both lists.

In this case, no. of comparisons =  $\lceil \frac{3n}{2} \rceil - 2$ .

$\{ f++ \geq 4 \Rightarrow f = 4 \}$  ref.

$\{ n \leq \lceil \frac{3n}{2} \rceil \}$  ref.

## Tournament method (Divide and conquer approach)

No. of comparisons needed =  $\lceil \frac{3n}{2} \rceil - 2$

Runtime complexity =  $\Theta(n)$

Space complexity =  $\Theta(\log n)$  (stack space)

Best algorithm to find minimum and maximum —

Runtime complexity =  $\Theta(n)$

Space complexity =  $\Theta(1)$

No. of comparisons needed =  $\begin{cases} \frac{3n}{2} - 2 & \text{if } n \text{ is even} \\ \lceil \frac{3n}{2} \rceil - 2 & \text{if } n \text{ is odd} \end{cases}$

G.F.G.  
Method 4

Binary search

## Linear search

```
int Linear_Search(A[], LB, UB, item) {
```

```
    for( k = LB; k <= UB; k++) {
```

```
        if( A[k] == item) {
```

```
            return k;
```

Runtime complexity =  $O(n)$

```
}
```

```
return LB-1;
```

- \* Linear search can be applied on any array (both sorted as well as unsorted).

## Binary search

- \* can be applied only on sorted array

```
int Binary_Search ( A[], LB, UB, item) {
```

```
    Low = LB; High = UB
```

```
    mid = (Low + High)/2;
```

```
    while ( Low <= High) {
```

```
        mid = Low + (High - Low)/2;
```

```
        if (item < A[mid]) {
```

```
            High = mid - 1;
```

Runtime complexity =  $O(\log_2 n)$

```
        else if (item > A[mid]) {
```

```
            Low = mid + 1;
```

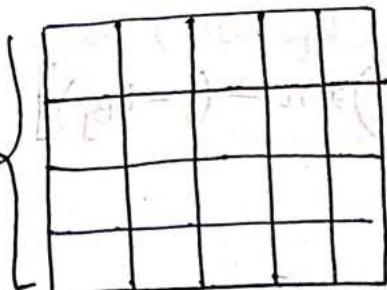
```
        else {
```

```
            return mid;
```

```
}
```

# 2D Array

collection of 1D arrays.



4 rows.

[ (j-i) 5 columns - i ]

2D Array In C-Language

<datatype> <name> [ <row no> ] [ <col no> ]

int A[3][4]

row no. = 0, 1, 2

L<sub>Bi</sub> = 0

U<sub>Bi</sub> = 2

col no. = 0, 1, 2, 3

L<sub>Bj</sub> = 0

U<sub>Bj</sub> = 3

2D array declaration in  
C Language.

<name> [ L<sub>Bi</sub>: U<sub>Bi</sub> ] [ L<sub>Bj</sub>: U<sub>Bj</sub> ]

no. of rows = U<sub>Bi</sub> - L<sub>Bi</sub> + 1

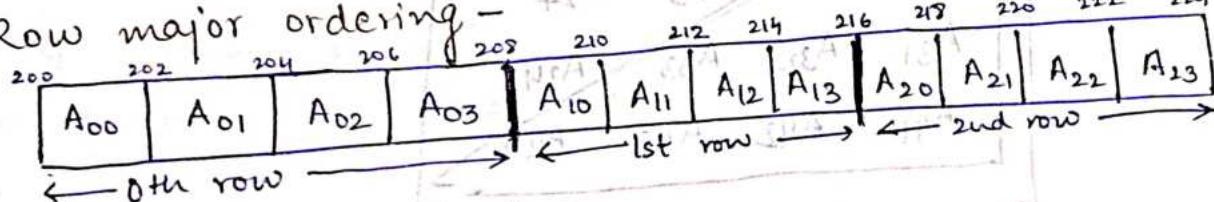
no. of columns = U<sub>Bj</sub> - L<sub>Bj</sub> + 1

size of array = m x n.

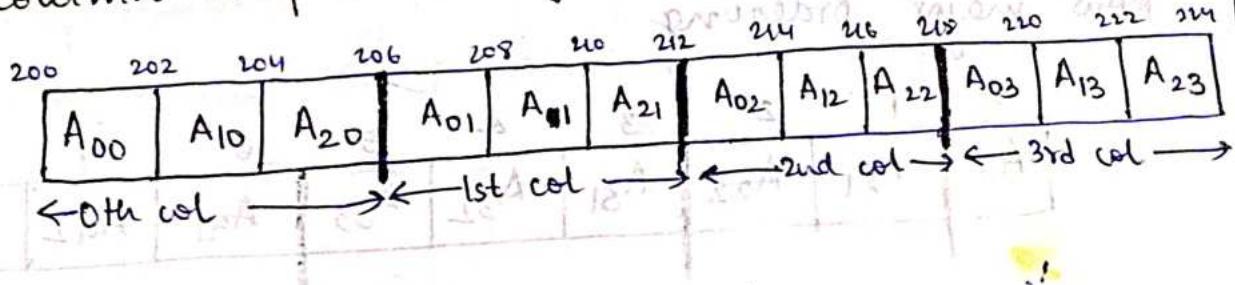
	0	1	2	3
0	A <sub>00</sub>	A <sub>01</sub>	A <sub>02</sub>	A <sub>03</sub>
1	A <sub>10</sub>	A <sub>11</sub>	A <sub>12</sub>	A <sub>13</sub>
2	A <sub>20</sub>	A <sub>21</sub>	A <sub>22</sub>	A <sub>23</sub>

## 2D Array storage

1. Row major ordering -



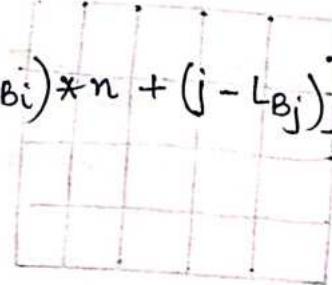
2. Column major ordering -



Address calculation in 2-D array.

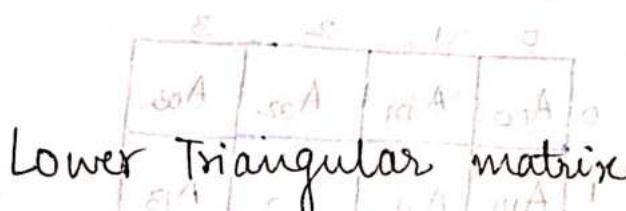
1. Row major ordering

$$\text{Location}(A[i][j]) = B + w * [(i - L_{Bi}) * n + (j - L_{Bj})]$$



2. Column major ordering

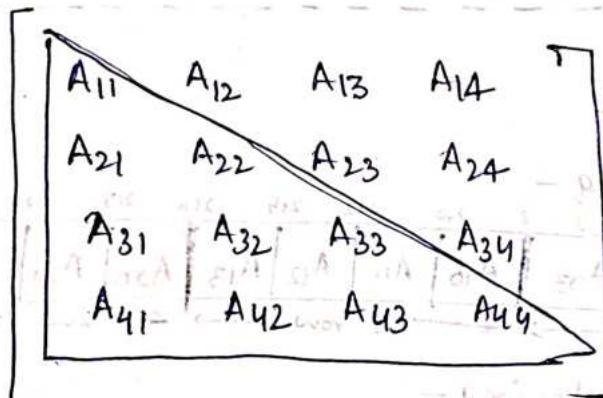
$$\text{Location}(A[i][j]) = B + w * [(j - L_{Bj}) * m + (i - L_{Bi})]$$



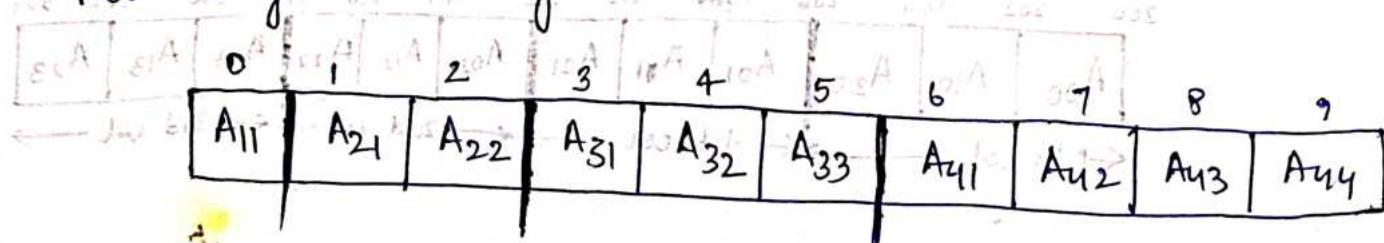
Lower Triangular matrix

$$A[i][j] = \begin{cases} 0 & \text{if } i < j \\ \text{non zero} & \text{otherwise} \end{cases}$$

$$\text{No. of elements stored} = \frac{n(n+1)}{2}$$



Row major ordering



~~$$\text{Location } [A[i][j]] = B + w * [(i - \lceil \frac{i}{n} \rceil) * i(i-1) + (j - \lceil \frac{j}{n} \rceil)]$$~~

~~$$\text{Location } [A[i][j]] = B + w * \left[ \left( \lceil \frac{i}{n} \rceil - 1 \right) * \frac{i(i-1)}{2} + (j - \lceil \frac{j}{n} \rceil) \right]$$~~

No. of elements in row 1 = 1  
No. of elements in row 2 = 2

~~$$(1-i)(s-i) - (1-i)n$$~~

No. of elements in row  $i-1 = i-1$

$$\therefore \text{Total no. of elements} = \frac{i(i-1)}{2}$$

$$[i - s + (i - j)(s - i) - (1 - i)n] * w + B = [i][j][A]_{\text{row major}}$$

Column major ordering

A <sub>11</sub>	A <sub>21</sub>	A <sub>31</sub>	A <sub>41</sub>	A <sub>12</sub>	A <sub>22</sub>	A <sub>32</sub>	A <sub>42</sub>	A <sub>13</sub>	A <sub>23</sub>	A <sub>33</sub>	A <sub>43</sub>	A <sub>14</sub>
200	202	204	206	208	210	212	214	216	218	220		

No. of elements in col 1 = n

No. of elements in col 2 = n-1

No. of elements in col 3 = n-2

⋮

No. of elements in col  $j^{\text{th}}$  col =  $n - (j - 1)$

$$\begin{aligned} \text{No. of elements in } (j-1)^{\text{th}} \text{ col} &= n - [j - 1 - 1] \\ &= n - j + 2 \end{aligned}$$

Total number of elements =

Total no. of elements from column 1 to column  $j-1$

$$= n + (n-1) + (n-2) + \dots + (n-(j-2))$$

$$= n(j-1) - [1+2+3+\dots+(j-2)]$$

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

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

$(i-1)$  : Index of row  $i$

$$\text{Location}(A[i][j]) = B + w * \left[ n(j-1) - \frac{(j-2)(j-1)}{2} + i-1 \right]$$

Upper Triangular matrix

$$A[i][j] = \begin{cases} 0 & \text{if } i > j \\ \text{non zero otherwise.} \end{cases}$$

$$\text{No. of elements stored} = \frac{n(n+1)}{2}$$

$A_{11}$	$A_{12}$	$A_{13}$	$A_{14}$
$A_{21}$	$A_{22}$	$A_{23}$	$A_{24}$
$A_{31}$	$A_{32}$	$A_{33}$	$A_{34}$
$A_{41}$	$A_{42}$	$A_{43}$	$A_{44}$

## Row major ordering

No. of elements in row 1 =  $n$

No. of elements in row 2 =  $n-1$

" " " " row 3 =  $n-2$

No. of elements in row  $i= n - (i-2)$

∴ Total no. of elements from  $\rightarrow$  from row 1 to row  $i-1$

$$= n + n-1 + n-2 + \dots + n-(i-2)$$

$$= n(i-1) - [1+2+\dots+(i-2)]$$

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

No. of elements before  $A[i][j]$  in  $i$ th row  
(beginning of storage address of row) =  $j-i$

$$\therefore \text{Location}(A[i][j]) = B + w * \left[ n(i-1) - \frac{(i-2)(i-1)}{2} + (j-i) \right]$$

## Column major ordering

No. of elements in col 1 = 1

No. of elements in col 2 = 2

" " " " " 3 = 3

:

No. of elements in col  $(j-1) = j-1$

∴ Total no. of elements before from col 1 to col  $j-1$

$$= 1+2+3+\dots+(j-1) = \frac{(j-1)j}{2}$$

$$\therefore \text{Location}(A[i][j]) = B + w * \left[ \frac{j(j-1)}{2} + i(i-1) \right]$$

practical to implement

Advantages of array -

- random access of elements using index no.
- other data structures (linked list, stacks, queues, trees, graphs etc) can be implemented.
- matrices can be easily represented.

$$(S-I) \cdot N + S \cdot N + I \cdot N + N =$$

$$[(S-I)N + SN] + I(N+1) = (I-S)N + I$$

Disadvantages of arrays -

- no. of elements should be known beforehand.
- static (once declared, cannot be modified)
- insertion and deletion are quite difficult.
- Elements must be stored in consecutive locations  
[if consecutive memory locations are not available,  
array cannot be declared]

I = 1st no. element for all

S = size of array for all

E = End " "

H-L = (H-L)N + S for all

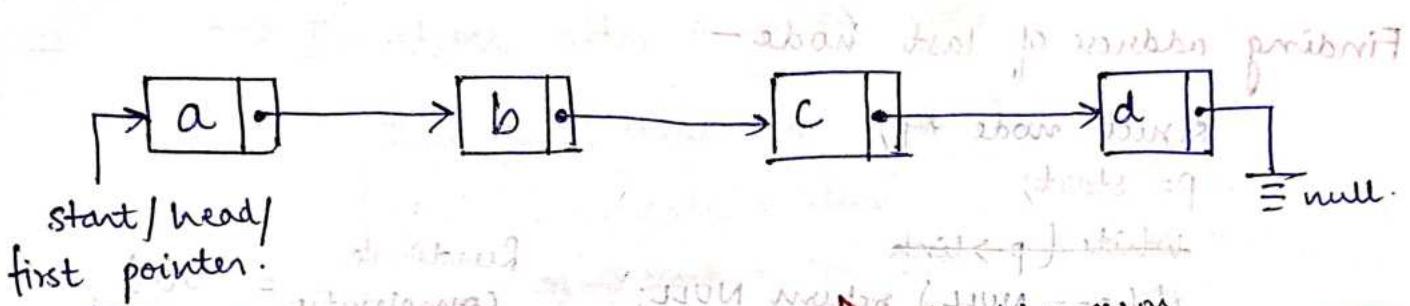
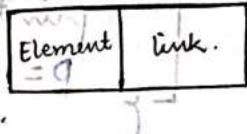
H-L for all has same starting address for all

$$\frac{H-L}{S} = (H-L)N + S + S + 1 =$$

$$H-L + \frac{H-L}{S} (N+S+1) = (H-L)(N+1) + H-L$$

# LINKED LIST

- ① Linear data structure
- ② Linear order is maintained using pointer (link)
- ③ List contains nodes.
- ④ Node contains 2 fields
  - data
  - next pointer



## Linked List Structure

```
struct node {
    char data;
    struct node *link;
};
```

runtime error

NULL → data

NULL → link

NULL pointer  
dereferencing  
error

## Empty List Condition

```
if (head == NULL) {
```

or

(or) Empty list is true

```
if (!start) {
```

Empty list

```
if (start == NULL || start->link == NULL) {
```

if start == NULL

if start->link == NULL

## Single Node Condition

```
if (start->link == NULL) {
```

Single Node

if link == NULL

if link == NULL

## Traversing a linked list

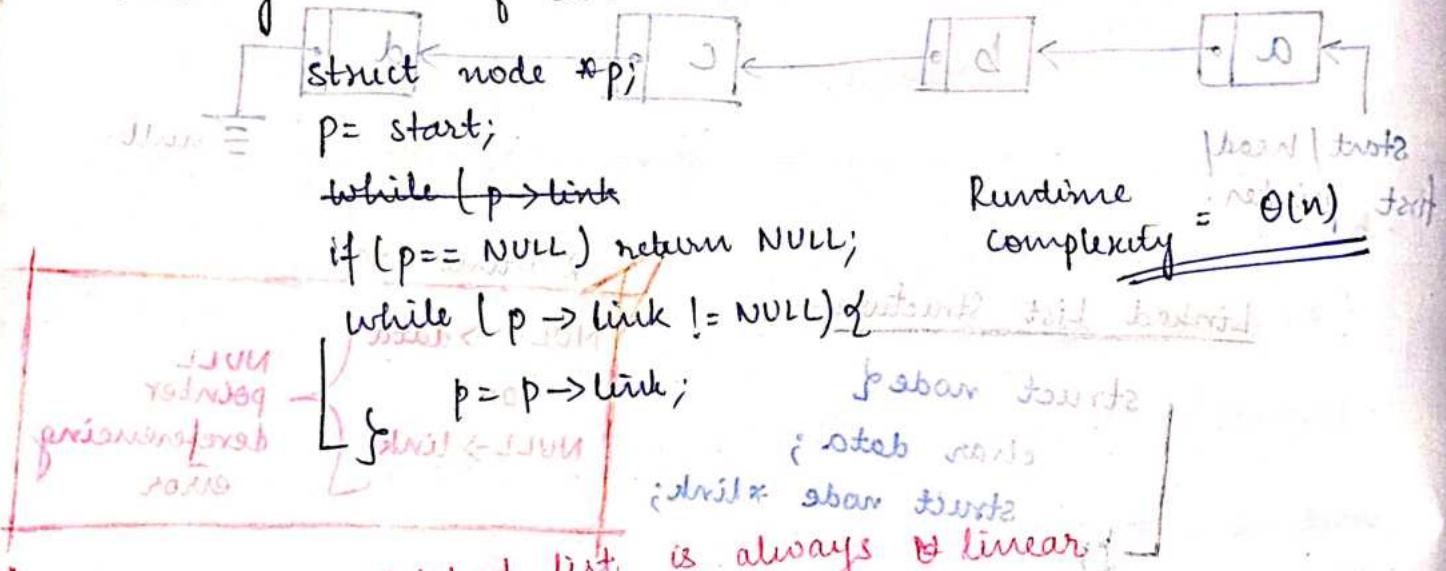
```

struct node *p;
p = start; // print beginning
while (p != null) {
    print (p->data);
    p = p->link; // moving to next
}

```

Runtime complexity =  $\Theta(n)$

Finding address of last node -



\* Searching in linked list is always **linear**.

Searching in linked list.

Searching (start, item) {

```

struct node *p = start;
while (p != null) {
    if (p->data == item) {
        return p;
    }
    p = p->link;
}
return NULL;
}

```

Runtime complexity =  $\Theta(n)$

*Initial state*: *start* points to node A. *item* is null.

*Iteration 1*: *start* points to node A. *item* is null. *item* is set to node A.

*Iteration 2*: *start* points to node B. *item* is node A. *item* is set to node B.

*Iteration 3*: *start* points to node C. *item* is node B. *item* is set to node C.

*Final state*: *start* points to node C. *item* is node C.

## Insertion in linked list -

~~Deletion in linked list~~  
~~insertion in linked list~~  
~~size of struct node~~  
~~start = start + 1~~  
~~return start~~

struct node \*n = (struct node \*) malloc (size of (struct node));

→ Insertion in beginning

$n \rightarrow \text{data} = \text{item}$

$n \rightarrow \text{link} = \text{start};$

$\text{start} = n;$  ~~above line for~~

Runtime complexity =  $\Theta(1)$

→ Insertion after a given node

Insertion (start, loc, item) {

$n \rightarrow \text{data} = \text{item};$

$n \rightarrow \text{link} = \text{loc} \rightarrow \text{link};$

$\text{loc} \rightarrow \text{link} = n;$

Runtime complexity =  $\Theta(1)$

→ Insertion before a given node.

$n \rightarrow \text{data} = \text{item};$

$n \rightarrow \text{link} = \text{loc};$

struct node \*p = start;

while ( $p \rightarrow \text{link} \neq \text{loc}$ ) {

$p = p \rightarrow \text{link};$

}

$p \rightarrow \text{link} = n;$

Runtime complexity =  $\Theta(n)$

(r) Q = ~~answering~~  
~~question~~

## Deletion in linked list

struct node \*p = <node to be deleted>;  
 ...  
 (1)  $\theta(1)$  =  $O(1)$

(1)  $\theta(1)$  =  $O(1)$

→ Deletion of first node -  
 (1)  $\theta(1)$  =  $O(1)$

struct node \*p = start;

start = start → link;

free(p);

Runtime complexity =  $\Theta(1)$

(1)  $\theta(1)$  =  $O(1)$

→ Deletion of last node -

struct node \*p = start;

while if (start == NULL) UNDERFLOW return.

while (p → link != NULL)

p = p → link;

while (p → link → link != NULL)

p = p → link;

} last = q \*; start = start → link;

struct node \*q = p → link;

p → link = NULL;

free(q); q = q

Runtime complexity =  $\Theta(n)$

(n)  $\theta(n)$  =  $O(n)$

→ Deletion of last node if last node pointer is given

struct node \*p = start;

while (p → link != last) {

} p = p → link;

struct node \*q = last;

free(q)

p → link = NULL

last = p;

Runtime complexity =  $\Theta(n)$

→ Deletion of a given node (loc)

```
struct node *p = start;
```

```
while (p->link != loc) {
```

```
    p = p->link;
```

Runtine complexity =  $O(n)$

```
p->link = loc->link;
```

```
free(loc);
```

→ Deletion after a given node (loc)

```
loc->link = loc->link->link;
```

```
free(loc->link);
```

Runtine complexity =  $\Theta(1)$

→ Deletion before a given node (loc)

```
struct node *p = start;
```

```
while (p->link->link != loc) {
```

```
    p = p->link;
```

```
struct node *q = p->link;
```

```
p->link = loc;
```

Runtine complexity =  $O(n)$

```
free(q);
```

## Application of linked list

- To store polynomials

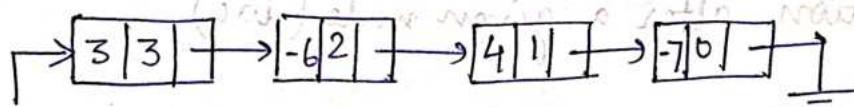
### 1) Univariate polynomial

single variable.

$$\text{Ex} \rightarrow 3x^3 - 6x^2 + 4x - 7 = \text{div} \leftarrow q$$

Node = 

Coeff	Exponent	(→)
-------	----------	-----



Start.

### 2) Bivariate polynomial.

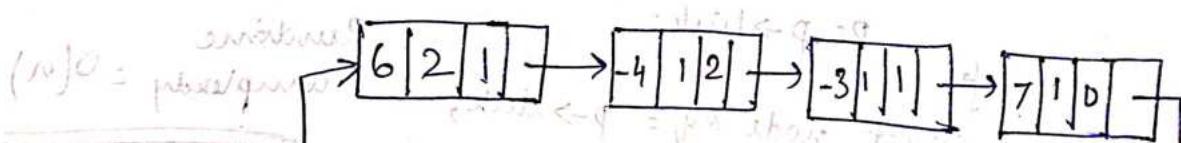
2 variable polynomial.

$$\text{Ex} \rightarrow 6x^2y - 4xy^2 - 3xy + 7x + 9y - 1 = 0$$

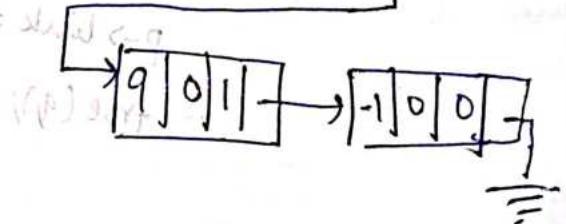
Node = 

Coeff	Exp <sub>x</sub>	Exp <sub>y</sub>	(→)
-------	------------------	------------------	-----

↓ (sat = limit < div < q) list



Start.



Disadvantages of singly linked list

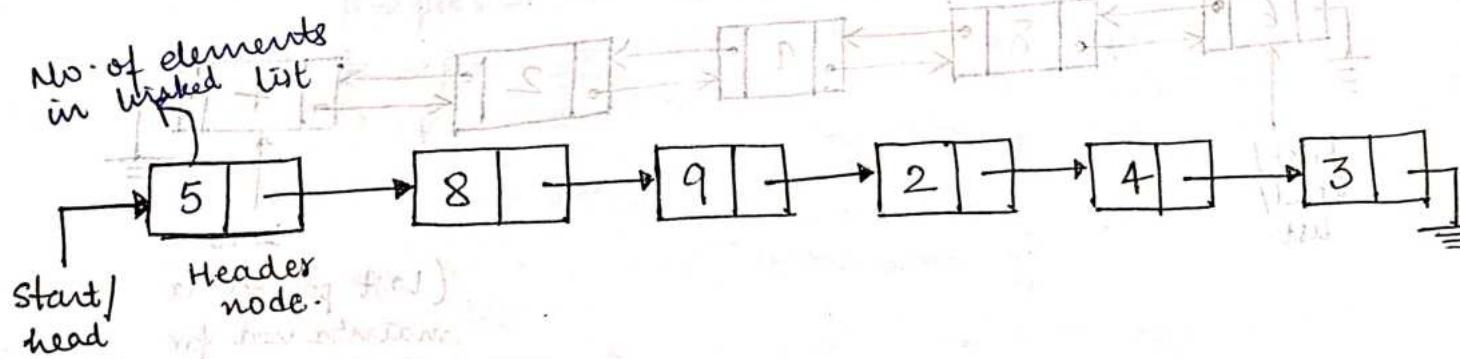
- Link part of last node is not utilized } circular list
- The address of predecessor is not known.
- Stepping backward is not possible.

Doubly linked list

## Header List

- ① contains special first node called Header node.
- ② Header node contains summary information.

(no. of nodes is stored bcoz for most of the operations, no. of nodes is req. and calculating it again and again takes  $\Theta(n)$  time runtime complexity)



Traversing in Header list —

Grounded

struct node \*p = list->link;

while (p != NULL) {

process p->data;

p = p->link;

struct node \*p = list->link

while (p != NULL) {

process p->data;

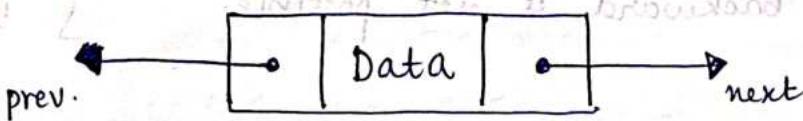
p = p->link;

$\Theta(n)$

$\Theta(n) = \text{very slow}$

$q = \text{fast}$

# Doubly Linked List



struct dnode {

int data; above two lines are comments

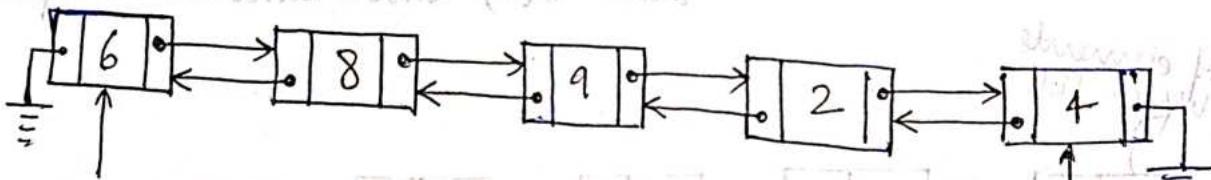
}; right side of colon is structure above itself

four ref. vars for struct dnode \*prev;

prev is above ; } ; everything left to

keep two lines to prevent two

(indicates continue on next line)



First/  
head/  
list.

tail ref

(last pointer is  
maintained for  
backward tracing)

→ Insertion at beginning

```

struct dnode *p = (struct dnode*) malloc(sizeof(struct dnode));
p->data = item;
if(head == NULL)
    p->prev = NULL
    p->next = head
    head = p;
    head->prev = p
    head = p

```

Runtime complexity =  $\Theta(1)$

→ Insertion after a given node

```
struct dnode *p = (struct dnode*) malloc(sizeof(struct dnode));  
p->data = item;  
p->next = loc->next;  
p->prev = loc;  
loc->next = p;  
p->next->prev = p;
```

Runtime complexity =  $\Theta(1)$

→ Insertion before a given node

```
struct dnode *p = (struct dnode*) malloc(sizeof(struct dnode));  
p->data = item;  
p->next = loc;  
p->prev = loc->prev;  
loc->prev = p;  
p->prev->next = p;
```

Runtime complexity =  $\Theta(1)$

→ Insertion at the end (last node not given)

```
struct dnode *n = (struct dnode*) malloc(sizeof(struct dnode));  
n->data = item;  
n->next = NULL;  
struct dnode *p = head;  
while(p->next != NULL){  
    p = p->next;  
}  
p->next = n;  
n->prev = p;
```

Runtime complexity =  $\Theta(n)$

→ Deletion from starting -

```
struct dnode *p = head;  
head = head->next;  
head->prev = NULL;  
free(p);
```

Runtine complexity =  $\Theta(1)$

→ Deletion of a given node (loc)

```
loc->next->prev = loc->prev; // update  
loc->prev->next = loc->next; // update  
free(loc);
```

Runtine complexity =  $\Theta(1)$

→ Deletion at the end if last node is not given.

```
struct dnode *p = head;
```

```
while (p->next != NULL) {
```

```
    p = p->link; p = p->next;
```

```
p->prev->link =
```

```
p->prev->next = NULL;
```

```
free(p);
```

Runtine complexity =  $\Theta(n)$

# QUEUE

- ① A linear data structure in which insertion can be done from one end (rear end) and deletion is done from the other end (front end).
- ② FIFO (First In First Out) list.
- ③ Queue can be implemented by arrays as well as linked list.

Insertion : Enqueue

Deletion : Dequeue

Linear queue

Linear queue

Circular queue.

Overflow condition = if ( $\text{rear} == N - 1$ )

Overflow condition = if (( $\text{front} == 0 \& \text{rear} == N - 1$ ) || ( $\text{front} == \text{rear} + 1$ ))  
if ( $\text{front} == (\text{rear} + 1) \bmod n$ )

Underflow condition = if ( $\text{front} == \text{rear} == -1$ )

Underflow condition = if ( $\text{front} == \text{rear} == -1$ )

at beginning & nothing is inserted & nothing is deleted

## Applications of queue -

QUEUE

- ① Serving requests on a single shared resource, like a printer, CPU task scheduling, disk scheduling etc.
- ② When data is transferred asynchronously b/w 2 processes. Ex → IO buffers, pipes, file IO etc.
- ③ Call center phone systems use queue to hold people.
- ④ Handling of interrupts in real time system.
- ⑤ Breadth First Search (BFS)

## Double Ended Queue

Queue in which insertion and deletion can be performed from both ends.

### Types —

- Input Restricted Double Ended Queue {  
Input is restricted to the standard end (Rear end) (In insertions)
- Output Restricted Double Ended Queue {  
Output is restricted to the standard end (Front end) (Deletions)

# Priority Queue

Every element is inserted with priority attached, and while deletion, highest priority element is deleted.

Queue	'a'	'b'	'e'	'c'	...
-------	-----	-----	-----	-----	-----

Priority	5	6	3	1	...
----------	---	---	---	---	-----

Priority queue : Implementation

Handle Priority at insertion time.

→ Insertion -

Insert element in such a way that elements are arranged in decreasing order of priority from front to rear.

→ Deletion -

Delete the front element (which is highest priority element).

Handle Priority at deletion time

→ Insertion -

A(1) Insert element at rear.

→ Deletion -

O(n) Find the element with highest priority and delete it.

★ Priority queue can be used to implement stack and queue

## Advantages of priority queue

- Elements can be accessed in faster way (since they are ordered by their priority)
- Efficient algorithms can be implemented (Dijkstra's algorithm).
- Used in real time systems (bcz quickly retrieves highest priority element)

Implementation of priority queue.	insert()	delete()
Binary Search Tree (BST)	$O(\log n)$	$O(\log n)$
Binary Heap	$O(\log n)$	$O(\log n)$
Linked list	$O(n)$	$O(1)$
Array	$O(n)$	$O(1)$

# STACK

- ① A linear data structure in which insertion and deletion both are performed from the same end known as top of the stack.
- ② LIFO (Last In First Out) list
- ③ Stack can be implemented using arrays or linked list
  - In case of array, top = index of last element inserted.

★ Some implementations use another index → bottom which stores the index to which top points when stack is empty.

Insertion in stack = PUSH  
Deletion in stack = POP.

## Stack Permutations

sequence of elements obtained by pushing and popping elements to obtain different sequences.

If there are  $n$  elements in input sequence,

~~valid stack permutations with valid A~~

No. of stack permutations possible

$$\frac{2^n C_n}{n+1}$$

catalan number

No. of invalid stack permutations

$$= n! \cdot \frac{2^n C_n}{n+1}$$

Example :-

Input sequence = 1 2 3

existing get stacks of stacks not stacks of stacks

Permutations possible —

3 2 1 (Push Push Push Pop Pop Pop)

1 2 3 (Push Pop Push Pop Push Pop)

2 1 3 (Push Push Pop Pop Push Pop)

2 3 1 (Push Push Pop Push Pop Pop)

1 3 2 (Push Pop Push Push Pop Pop)

existing get stacks of stacks not stacks of stacks

therefore invalid or otherwise

No. of valid stack permutations =  $\frac{2^n C_n}{n+1} = \frac{6 C_3}{4}$

$$= \frac{6}{3 \cdot 2} \cdot \frac{1}{4} = \frac{6 \times 5 \times 4}{3 \times 2} \cdot \frac{1}{4}$$

$$= \underline{\underline{5}}$$

Ques The following sequence of operations is performed on a stack.

- PUSH(15)

- PUSH(-18)

- POP - 18

- PUSH(15) -

- PUSH(-15) -

- POP - 15

- POP - 15

- POP - 15

- PUSH(15)

- POP - 15

Sequence

of values = 18, 15, 15, 15, 15

Starts →

→ 18, 15, 15, 15, 15

↑ = 1st

→ 15 = 2nd

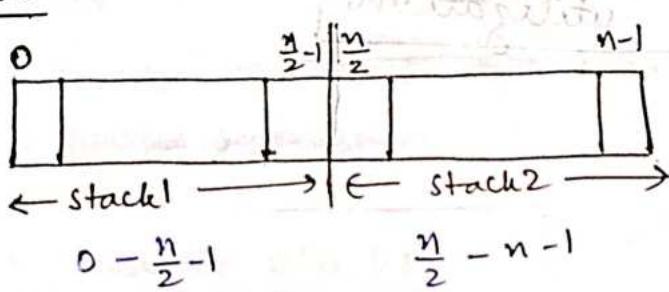
→ 15 = 3rd

→ 15 = 4th

→ 15 = 5th

## Multiple Stacks in Single Array

### Method 1



One stack cannot be used the space of other stack which leads to

Improper utilization of stack space

Initially,

$top1 = -1$

$top2 = \frac{n}{2} - 1$

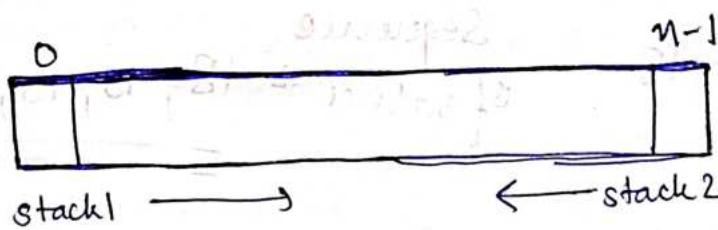
(c) out → w1

(c) out → wA

to use Dequeue & enqueue for insertion & deletion

### Method 2

2 stacks start from opposite ends of the array.



Initially, stacks are empty

$$\text{top1} = -1$$

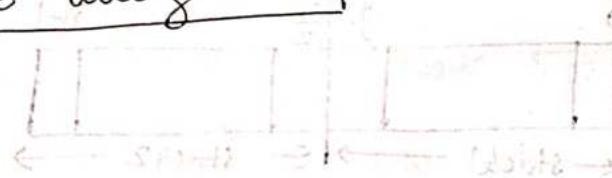
$$\text{top2} = n$$

Overflow:

$$\text{top1} == \text{top2} - 1 \quad \text{or}$$

$$\text{top2} == \text{top1} + 1$$

Better space utilization



Ques

What is the minimum number of queues of size  $n$  required to implement a stack of size  $n$ .

Ans — Two (2)

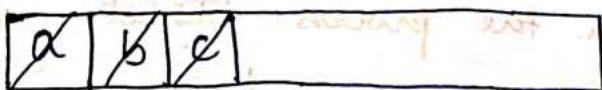
Ques

What is the minimum number of stacks of size  $n$  required to implement a queue of size  $n$ .

Ans — Two (2)

## → Stack using queues

2 queues are required.



original

secondary

### PUSH

Insert in original queue.

### POP

① Dequeue  $n-1$  elements from original queue and enqueue into secondary.

② Dequeue from original queue.

③ Make secondary queue original and original queue secondary.

## → Queue using stacks

### ENQUEUE -

① Push element to  $s_1$ .

### DEQUEUE -

① IF  $s_2$  is not empty, pop from  $s_2$

② IF  $s_2$  is empty, bring all the elements from  $s_1$  to  $s_2$  And pop from  $s_2$ .

~~Gate 2006~~

Let  $n$  insert and  $m$  ( $\leq n$ ) delete operations be performed in an arbitrary order on an empty queue  $Q$ . Let  $x$  and  $y$  be the number of push and pop operations performed respectively in the process. Which one of the following is true for all  $m$  and  $n$ .

- A.  $n+m \leq x < 2n$  and  $2m \leq y \leq n+m$
- B.  $n+m \leq x < 2n$  and  $2m \leq y \leq 2n$
- C.  $2m \leq x < 2n$  and  $2m \leq y \leq n+m$
- D.  $2m \leq x < 2n$  and  $2m \leq y \leq 2n$

B

### Worst case scenario -

Perform  $n$  <sup>insert</sup> operations and then perform  $m$  delete operations.

INSERTION — No. of push =  $n$

DELETION — No. of push =  $n$   
No. of pop =  $n+m$

$$\therefore \text{Total no. of PUSH} = n+n = 2n \quad \text{Total no. of POP} = n+m.$$

### Best case scenario -

Perform  $m$  insert operations, delete them and then insert  $n-m$  items.

INSERTION — No. of push =  $m+n-m=n$

DELETION — No. of push =  $m$

No. of pop =  $m+m=2m$

$$\text{Total no. of PUSH} = m+n$$

$$\text{Total no. of POP} = 2m$$

# Applications of stack

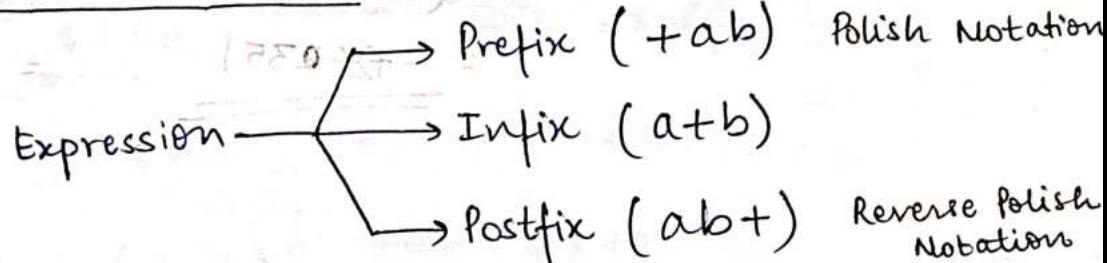
## ① Expression Evaluation

### ② Recursion

$$1 - ((2 + 3) * 5) = 1 - ((2 + 3) * 5) =$$

$$1 - ((2 + 3) * 5) = 1 - ((2 + 3) * 5) =$$

### 1.1 Expression evaluation



$$P + 8 \{ ((S * E) * d) \} - T$$

$$P + 8 \{ ((S * E) * d) \} - T$$

$$P + 8 \{ S * E * d - T$$

### Operator Precedence

$$P + 8 \{ ((S * E) * d) \} - T$$

$$P + 8 \{ ((S * E) * d) \} - T$$

Left

$$P + 8 \{ (S * E * d) \} - T$$

$$P + 8 \{ (S * E * d) \} - T$$

Right

$$P + 8 \{ (S * E * d) \} - T$$

$$P + 8 \{ (S * E * d) \} - T$$

Left

$$P + (-18 * 12 * 5) - T$$

$$P + (-18 * 12 * 5) - T$$

Left

$$P + 8 * 12 * 5 - T$$

$$P + 8 * 12 * 5 - T$$

Right

### Operators

- d

d -

p a r

Unary

Binary

x p a l

1 operand.

2 operands

x 1

1 x

All unary operators have higher precedence than binary operators.

Infix	Prefix	Postfix
$2+3*5-1$	$(2+(3*5))-1$	$(2+(3*5))-1$
	$= (2+(*35))-1$	$= (2+(35*))-1$
	$= (+2*35)-1$	$= (235*+)-1$
notation with $(d+b)$ script	$= +2*35$	$= 235*+1-$
$(d+b)$ script		
another method $(+dn)$ script		

$$7-6*3\uparrow 2/8+9$$

$$(7-((6*(3\uparrow 2))/8))+9$$

$$(7-((6*(3\uparrow 2))/8))+9$$

$$(7-((6*(\uparrow 32))/8))+9$$

$$(7-((6*(32\uparrow))/8))+9$$

$$(7-((6*(32\uparrow))/8))+9$$

$$(7-((6*(32\uparrow))/8))+9$$

$$(7-(1*6\uparrow 328))+9$$

$$(7-(632\uparrow *8/))+9$$

$$(7/(1*6\uparrow 328))+9$$

$$(7632\uparrow *8/-)+9$$

$$+ - 7 / * 6 \uparrow 3289$$

$$\underline{7632 \uparrow * 8 / 9 +}$$

$$-b$$

$$-b$$

$$b -$$

$$\log x$$

$$\log x$$

$$x \log$$

$$x!$$

$$!x$$

$$x!$$

$$\log x!$$

$$\log !x$$

$$x! \log$$

$$(A+B) * (C-D) / F - X * Y / Z$$

Prefix -

$$= (+AB) * (-CD) / F - (*XY) / Z$$

$$= (*+AB-CD) / F - (/*XYZ)$$

$$= /*+AB-CD / F - /*XYZ$$

$$= /*+AB-CD / /*XYZ$$

Postfix -

$$(A+B) * (C-D) / F - X * Y / Z$$

$$= (AB+) * (CD-) / F - (XY*) / Z$$

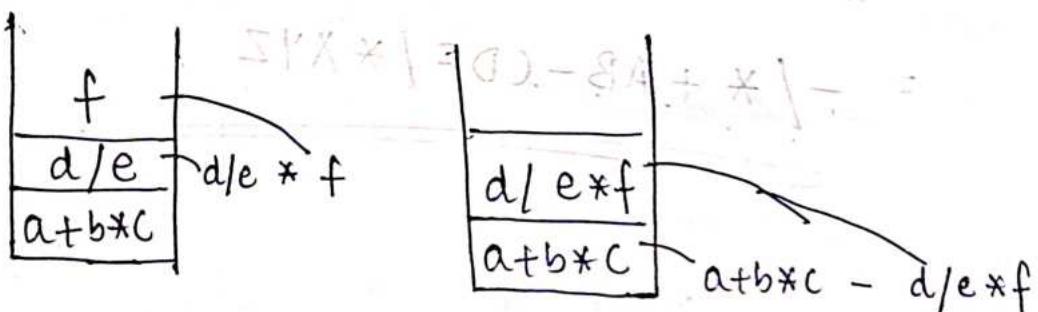
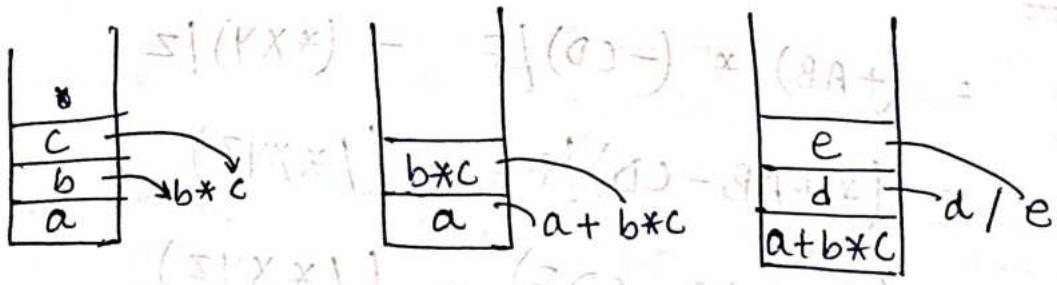
$$= (AB+CD-* / F - (XY*Z) /$$

$$= (AB+CD-*F / ) - (XY*Z / )$$

$$= AB+CD-*F / XY*Z / -$$

Postfix to Infix -  $\therefore PEX = 7 \backslash (0-3) * (8+A)$

$$abc * + de / f * -$$



$\therefore PEX = 7 \backslash (0-3) * (8+A)$

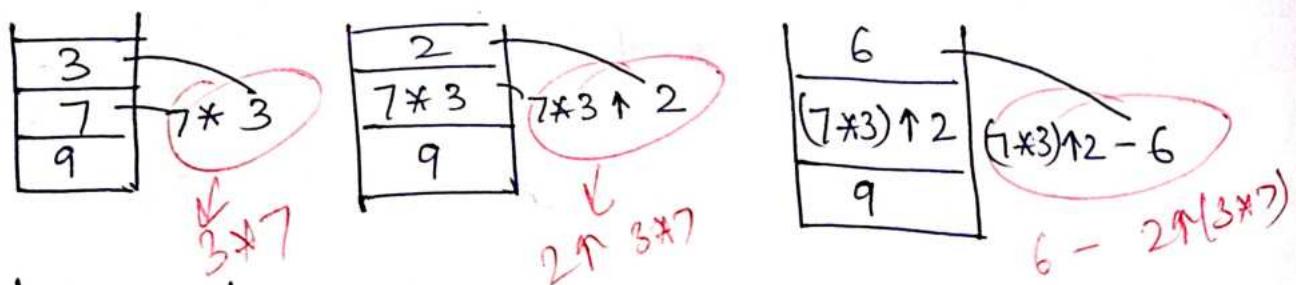
Aus

Infix Expression =  $a+b*c - d/e*f$

Prefix to Infix -

Reverse the prefix and solve by postfix method.

$$\begin{aligned} & + - 6 \uparrow 2 * 3 \uparrow 9 \\ & = 9 7 3 * 2 \uparrow 6 - + \end{aligned}$$



Aus:

Infix Expression =  $9 + ((7 * 3) \uparrow 2 - 6)$

convert the expression to prefix and postfix

$$a + b * c - d \wedge e \wedge f$$

Prefix conversion -

$$a + (b * c) - (d \wedge (e \wedge f))$$

$$a + (*bc) - (d \wedge (\wedge ef))$$

$$a + (*bc) - (\wedge d \wedge ef)$$

$$= (+a * bc) - (\wedge d \wedge ef)$$

$$\underline{\underline{+ a * bc \wedge d \wedge ef}}$$

Postfix expression -

$$a + (b * c) - (d \wedge (e \wedge f))$$

$$= a + (bc*) - (d \wedge (ef^*))$$

$$= a + (bc*) - (def^{**})$$

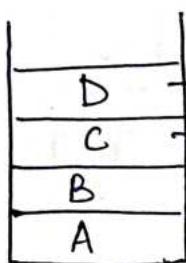
$$= (abc*)+ - (def^{**})$$

$$= abc* + def^{**} -$$

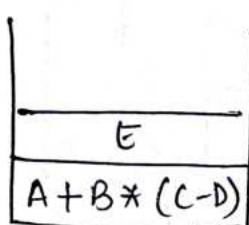
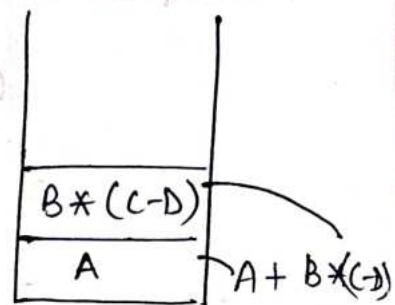
$$(1) = 1000$$

Writing back after evaluating stack frame  
Convert to infix —

$A B C D - * + E /$



$$((f^a)^b) - (d * c) + b /$$

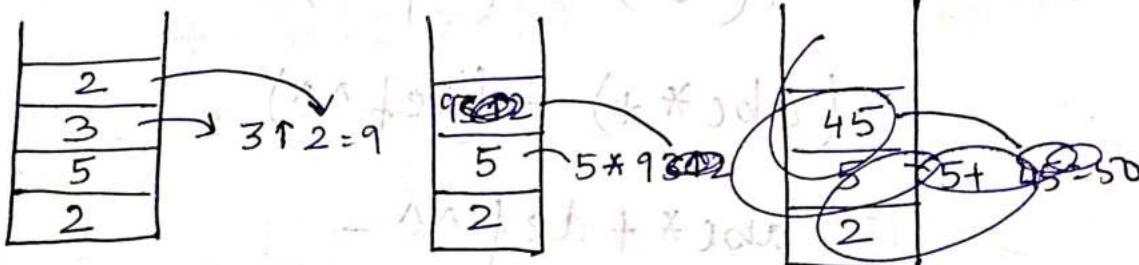


$$\text{Ans} \quad \text{Infix expression} = (A + B * (C - D)) / E$$

## Postfix Evaluation

$$2 \ 5 \ 3 \ 2 \uparrow * + 6 = (2 * 3) + 6 =$$

$$((2 * 3)) + 6 = (6 * 3) + 6 =$$



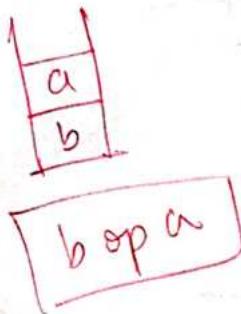
$$45 \quad 2 \quad 2 + 45 = 47$$

$$6 \quad 47 \quad 47 - 6 = 41$$

$$\text{Ans} = 41$$

## Algorithm:

- ① Add parenthesis ) at the end of P.
- ② Scan P from left to right until ) is encountered.
  - If operand is encountered, PUSH it onto stack.
  - If operator is encountered,
    - i. Pop 2 elements from stack → a & b
    - ii. Evaluate b op a and put re push result onto stack.
- ③ Result = top of stack.



Ques

Evaluate postfix expression —

10 5 + 30 6 / 5 5 \* 3 \* \* + 8 -

No if and else

if else

10	5
+ 10 + 5 = 15	

6	30
30 / 6 = 5	

5	5
5 * 5 = 25	

3	25
25 * 3 = 75	

75	5
5 * 75 = 375	

375	15
15 + 375 = 390	

8	390
390 - 8 = 382	

Ans = 382

Ques

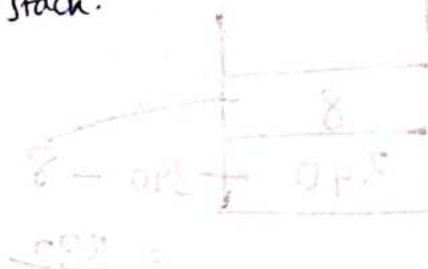
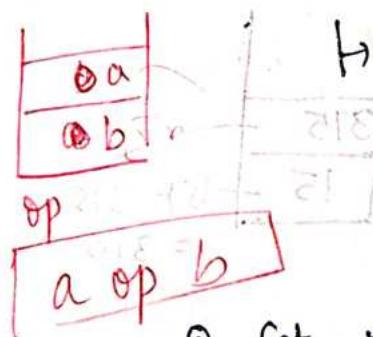
$$\begin{array}{r}
 9 \ 3 \ 1 \ ^\wedge / \ 4 \ 2 \ * + (5 \ 6) \ *
 \\ \hline
 3 \qquad \qquad \qquad 8 \qquad \qquad \qquad 5
 \\ \hline
 3
 \\ \hline
 11
 \end{array}$$

Ans = 6

## Prefix Evaluation

Algorithm:

- ① Add right parenthesis at the beginning of P
- ② Reverse the P.
- ③ Scan P from right to left until ( is encountered:
  - If operand is encountered, push it onto stack.
  - If operator is encountered,
    - i. Pop 2 elements from stack
    - ii. Evaluate a op b & push the result onto the stack.
- ④ Set result = top of stack.



Postfix evaluation — b op a

Prefix evaluation — a op b.

Ques

$$\begin{array}{r} - + 5 | * 6 2 3 4 \\ \hline 12 \\ \hline 4 | \cancel{5} * 2 3 4 | \text{Ans} = 5 \\ \hline 9 | \cancel{5} * 2 3 4 | \\ \hline 5 | (\cancel{5} * 2 3) - 4 | (- 28) * ( + 5) \\ \hline (\cancel{5} * 2 3) - 4 | (+ - 28 + 5) \end{array}$$

Ques

$$\begin{array}{r} / 3 * 9 + 4 + 5 \cancel{3} \\ \hline 8 \\ \hline 12 \\ \hline 108 \\ \hline 3/108 = \frac{1}{36} \\ \hline \end{array}$$

$\text{Ans} = \frac{3}{108}$

Ques

Ques

Convert the following expression into postfix and then evaluate using stack.

$$(2+6)* (8-3) / 4 - 9 * 2 / 3$$

Number of ~~push~~ PUSH and POP operations in evaluation are -

$$(2+6)* (8-3) / 4 - 9 * 2 / 3$$

$$(62+)* (83-) / 4 - (92*)/3$$

$$(62+83-* )/4 - (92*3)$$

$$(62+83-*4) - (92*3)$$

$$62+83-*4/92*3/-$$

$$\begin{array}{r} 6 \\ 8 \quad 5 \\ \hline 40 \end{array}$$

$$\begin{array}{r} 18 \\ 9 \\ \hline 6 \end{array}$$

$$\begin{array}{r} 4 \\ \hline 4 \end{array}$$

$$\frac{1}{38} = 801/2$$

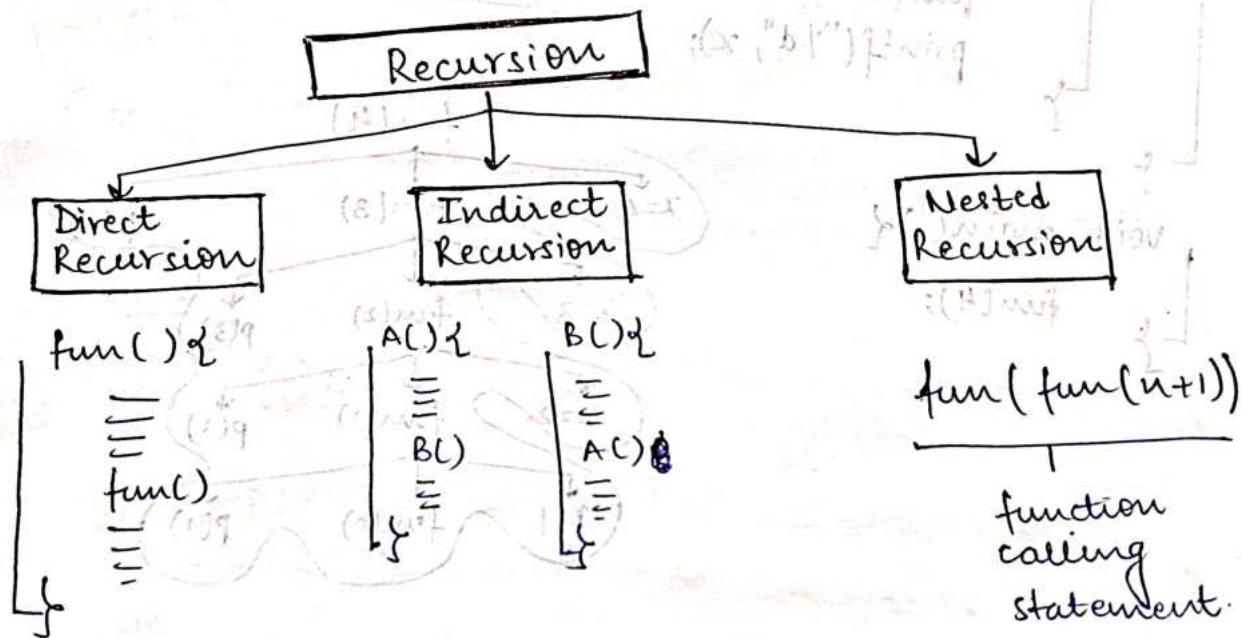
$$\text{No. of PUSH operations} = 2+1+2+1+1+1+1+2+1+1+2 \\ = \underline{\underline{15}}$$

$$\text{No. of POP operations} = \underline{\underline{14}} \quad [\text{No. of operators } \times 2]$$

## 2. Recursion

When a function calls itself, it is called recursion.

- Function should have base/exit/termination condition for which the function does not call itself.
- Every time the function calls itself, the base condition should come closer.



## Activation Record

Whenever there is a function call, activation record is created and pushed onto the stack.

→ when a call is completed, activation record is deleted from stack.

Activation Record contains

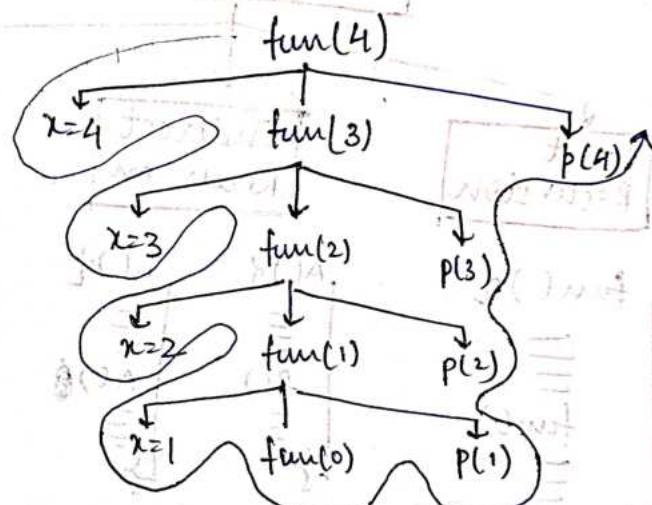
- function parameters
- local variable.

Ques

```
void fun(int x) {
    if(x > 0) {
        fun(x-1);
        printf("%d", x);
    }
}
```

```
void main() {
    fun(4);
}
```

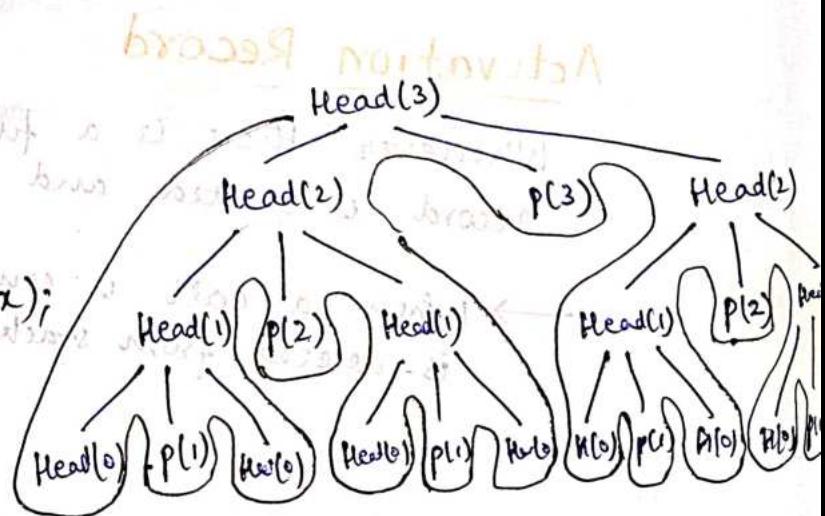
O/P  
1 2 3 4



Ques

```
void Head(int x) {
    if(x > 0) {
        Head(x-1);
        printf("%d", x);
        Head(x-1);
    }
}
```

```
void main() {
    Head(3);
}
```



O/P  
1 2 1 3 1 2 1

Ques

```
int X( int N ) {
```

```
    if ( N < 3 )
```

```
        return 1;
```

```
    else
```

```
        return X(N-1) + X(N-3) + 1 * X(2) + X(0) + 1;
```

X(5)

X(4) + X(2) + 1

X(3) + X(1) + 1

}

```
main ( ) {
```

```
    cout << X(5);
```

}

O/P → 7

No. of function calls = 7

Ques

```
int X( int N ) {
```

```
    if ( N < 3 )
```

```
        return 1;
```

```
    else
```

```
        return ( X(N-1) + X(N-3) + 1 );
```

X(5) = 7

X(7)

X(6) + X(4) + 1

X(5) + X(3) + 1

X(3) + X(1) + 1

X(4) + X(2) + 1

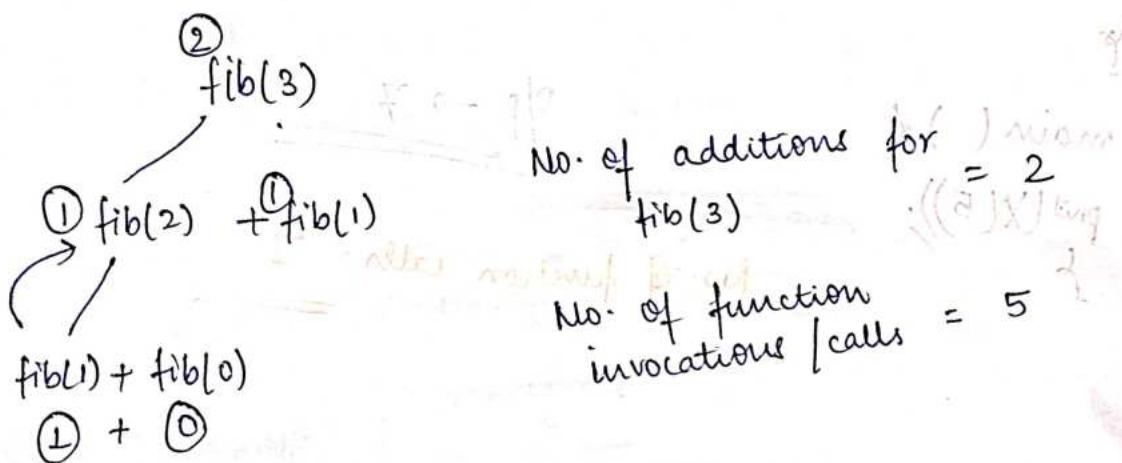
X(3) + X(1) + 1

No. of  
calls = 17 + 7 = 24

Ans = 17

## Fibonacci series

$$\text{fib}(n) = \begin{cases} 0 & \text{if } n=0 \\ 1 & \text{if } n=1 \\ \text{fib}(n-1) + \text{fib}(n-2) & \text{if } n>1 \end{cases}$$

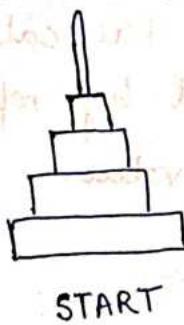


\* No. of function calls / invocations for  $\text{fib}(n)$  =  ~~$2 * \text{fib}(n+1) - 1$~~

\* No. of additions done to be done for  $\text{fib}(n)$  =  ~~$\text{fib}(n+1) - 1$~~

## Tower of Hanoi

Initial



Transfer disc from start to end keeping following rules -

→ Move one disc at a time

→ Larger disc cannot be placed over smaller

TOH( $n$ , Start, Aux, End) {

} If ( $n > 0$ ) {

TOH ( $n-1$ , Start, End, Aux)

Move 1 disk from Start to End.

TOH ( $n-1$ , Aux, Start, End)

Runtime complexity =  $O(2^n)$

For  $n$  disks

\*  $\text{No. of disk movements} = \underline{\underline{2^n - 1}}$

$\text{No. of function calls} = \underline{\underline{2^{n+1} - 1}}$

~~GATE 2013~~

What is the return value of  $f(p, p)$  if the value of  $p$  is initialized to 5 before the call?

Note that the first parameter is passed by reference whereas second parameter is passed by value.

To solve this problem,  
int f(int &x, int c) {

for loop starts at  $c = 0$

when  $c = 0$  if ( $c == 0$ ) return 1

$x = x + 1$ ;

return  $f(x, c) + x$ ;  $\{0 \leq N\}$ ,

}  $\{x \text{ is } \text{local variable}, f(x) \text{ is local}\}$

fun(5, 5)  $x = 5, 6, 7, 8, 9$   
 $c = 5, 4, 3, 2, 1$

↓

fun(6, 4) \*  $x$   $\{x \text{ is local variable}\}$   $729 * 9 = 6561$

( $x$ ) ↓

- fun(7, 3) \*  $x = 81 * 9 = 729$

↓

- fun(8, 2) \*  $x = 9 * 9 = 81$

( $x$ ) ↓

- fun(9, 1) \*  $x = 1 * 9$

( $x$ ) ↓

- fun(10, 0) \*  $x$   $\{x \text{ is local variable}\}$

Aust = 6561

~~GATE 2007~~

Consider the following C function:

for statement f ( int n ) {

    static int r = 0;

    if ( n <= 0 ) return 1;

    if ( n > 3 ) {

        r = n;

        return f ( n - 2 ) + 2;

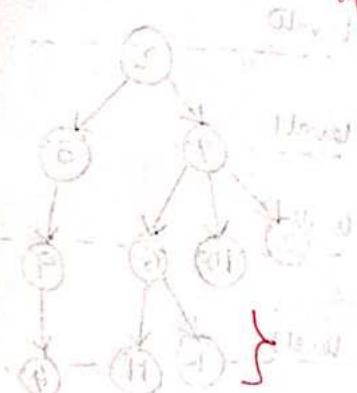
    } else {

        return f ( n - 1 ) + r;

    }

    }

~~Ans - 18~~



f(5)

~~r = 0~~ 5

f(5)

for ( int i = 0 ;

    i < n ; i++ )

    ( i ) is how

f(5)

for ( int i = 0 ;

    i < n ; i++ )

f(3) + 2

f(3) + 2 ( n ) 16 + 2 = 18

f(2) + 5 = 11

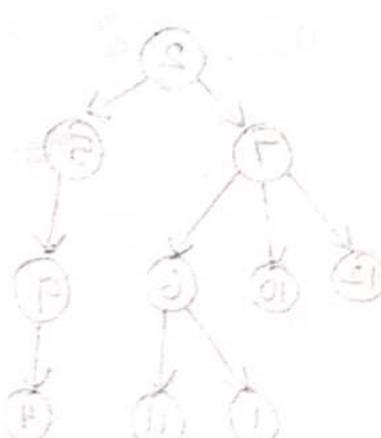
11 + 5 = 16

f(1) + 5 = 6

6 + 5 = 11

f(0) + 5 = 1 + 5 = 6

((((f)P)T, ((H,I)S, (O,S)T))S

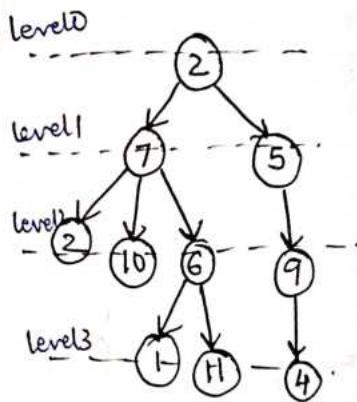


# TREE

Non linear hierarchical data structure that consists of nodes connected by edges

## Tree Terminology -

- ① Root
- ② Child
- ③ Parent
- ④ Sibling
- ⑤ Subtree
- ⑥ Internal Node
- ⑦ External Node  
(leaf node)



\* Depth of a node = its level number

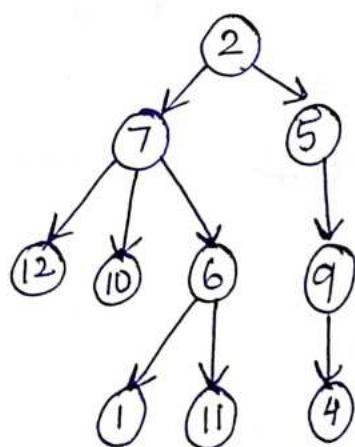
## Handshaking Lemma -

$$\text{Total Number of Nodes (N)} = \text{Total no. of Internal nodes (I)} + \text{Total no. of External nodes (L)}$$

$$N = I + L$$

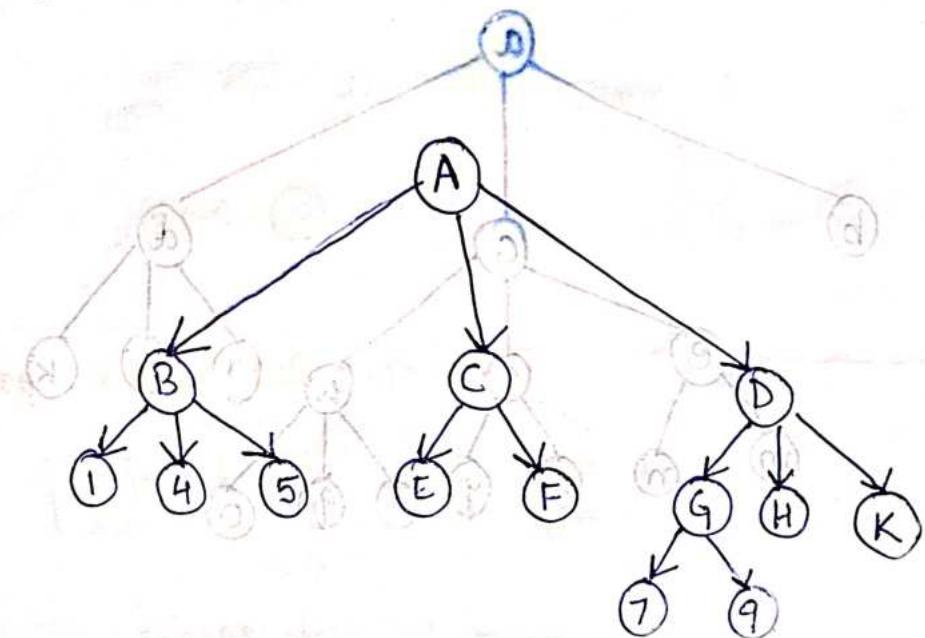
## Parenthesis Method of Representation

Node (child<sub>1</sub>, child<sub>2</sub> .....)



$2(7(12, 10, 6(1, 11)), 5(9(4)))$

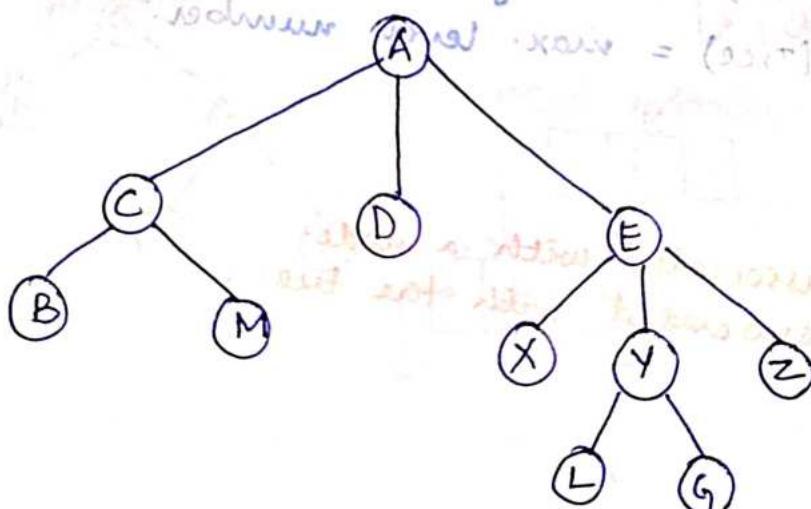
Ques  
Construct the tree from the following parentheses representation  
 $A(B(1,4,5), C(E,F), D(G(7,9), H,K))$



Ques

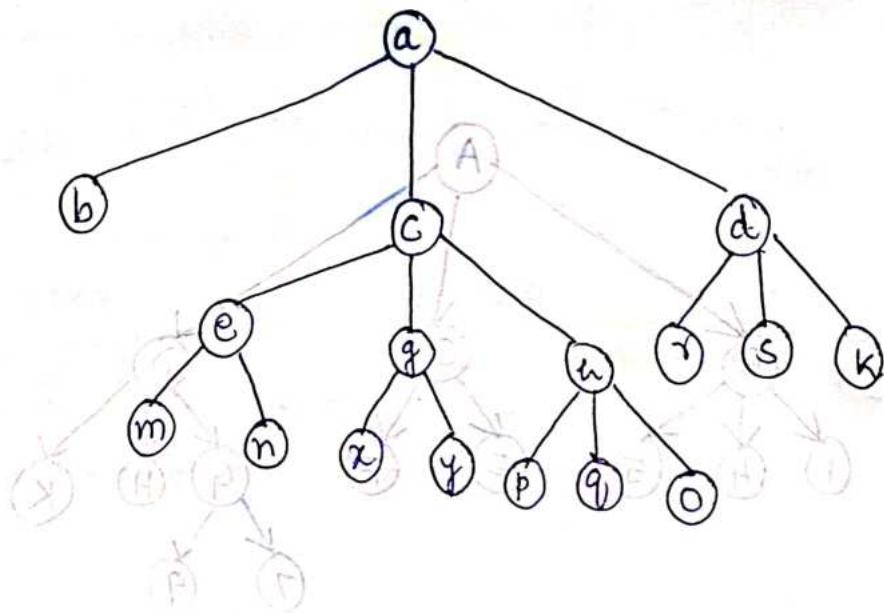
$A(C(B,M), D, E(X, Y(L, G), Z))$

$O = \text{short algorithm} (sort + H)$



Ques

$a(b, c(e(m,n), g(x,y), h(p,q,r,o)), d(r,s,t))$   
 $((x,H,F,G)\in A, F\in B, G\in C)$



### Height of Tree

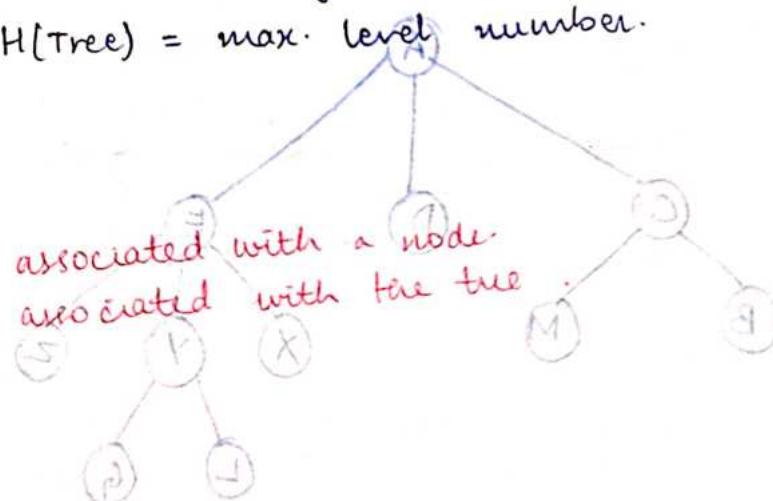
No. of edges in the path from the root to the farthest leaf node.

~~$H(\text{Tree}) \text{ of empty tree} = -1$~~   $(M, S) \rightarrow A$

$H(\text{tree}) \text{ with single node} = 0$

$H(\text{tree}) = \text{max. level number.}$

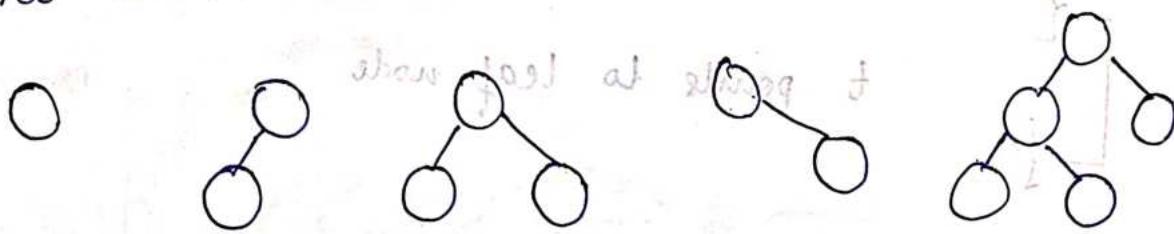
\* Depth is associated with a node.  
 Height is associated with the tree.



# Binary Tree

shows for a depth of 3

(LLH == binary <--> LHL) = maximum of 2 children.  
A tree in which each node has



## Linked Representation

left child	key	right child
------------	-----	-------------

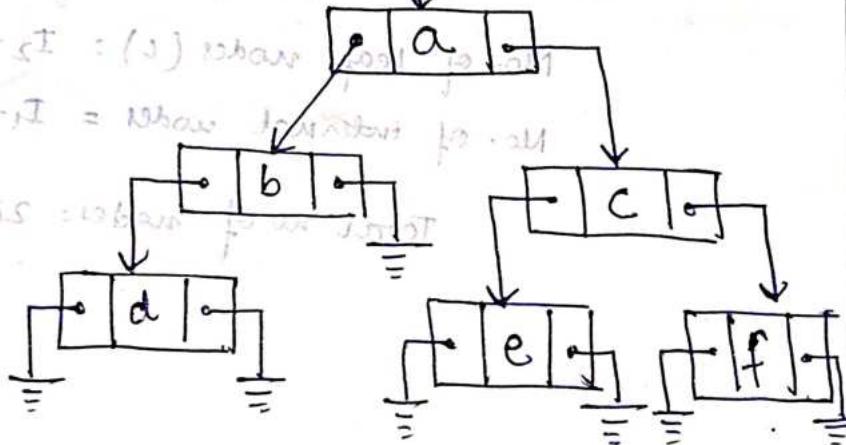
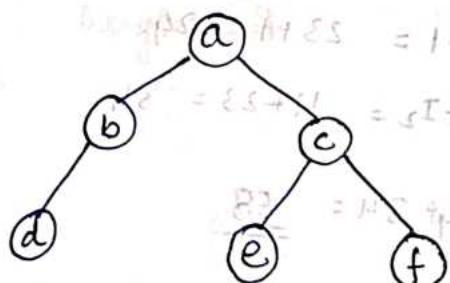
struct BTnode {

char key;

struct BTnode \*Lchild;

struct BTnode \*Rchild;

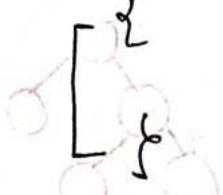
Root pointer



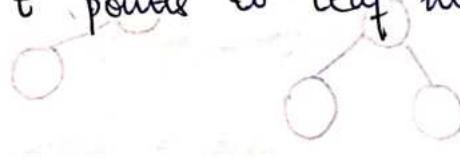
How to detect a Leaf Node?

~~sort priority~~

if ( $t \rightarrow \text{Lchild} == \text{NULL}$  &  $t \rightarrow \text{Rchild} == \text{NULL}$ )



$t$  points to leaf node



30

$$\text{Number of leaf nodes} = \frac{\text{No. of nodes with degree } 1}{2} + 1$$

$I_1$        $I_2$        $I_3$

$$L = I_2 + 1$$

3 leaf nodes

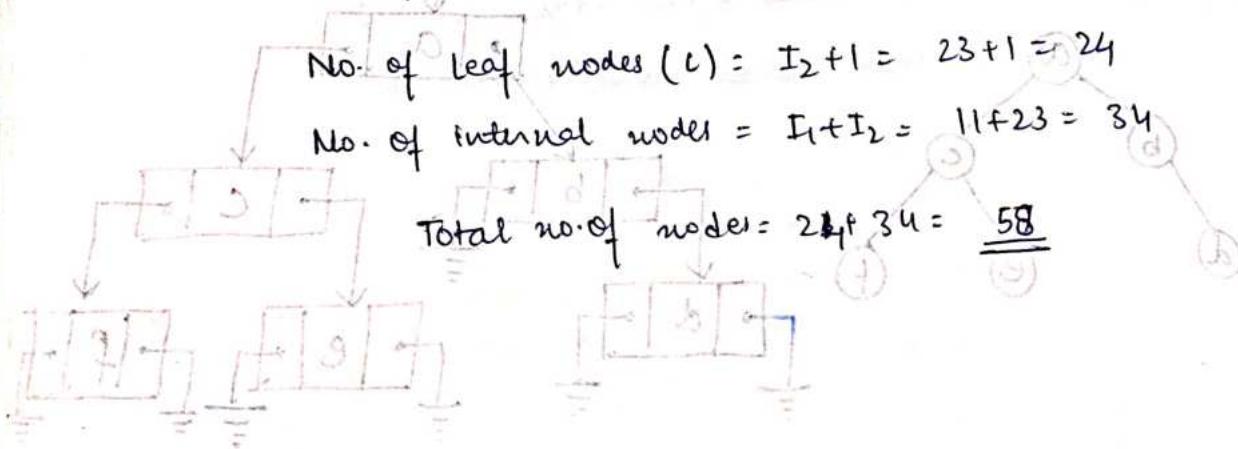
Ques

In a binary tree, if the no. of nodes with degree 1 is 11 and the number of nodes of degree 2 is 23. Total no. of nodes in the binary tree?

$$\text{No. of leaf nodes (L)} = I_2 + 1 = 23 + 1 = 24$$

$$\text{No. of internal nodes} = I_1 + I_2 = 11 + 23 = 34$$

$$\text{Total no. of nodes} = 24 + 34 = \underline{\underline{58}}$$



No. of binary trees that can be constructed using  $n$  unlabelled nodes =  $\frac{2^n C_n}{n+1}$  Catalan no.

No. of binary trees that can be constructed with  $n$  distinct keys =  $\frac{C_n}{n+1} * n!$

Maximum and minimum number of nodes in a binary tree at a level number  $L$ .

1 minimum

$2^L$  maximum

Minimum and maximum number of nodes in a binary tree of height  $H$

$H+1$  minimum

$2^{H+1}-1$  maximum

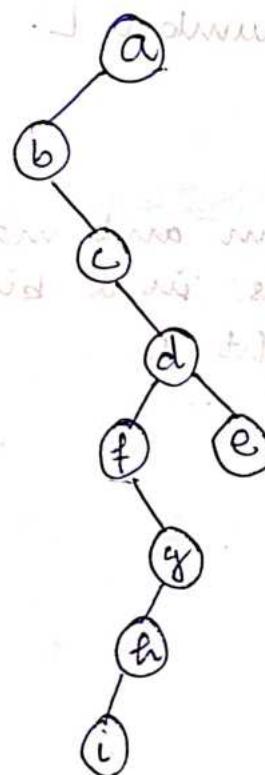
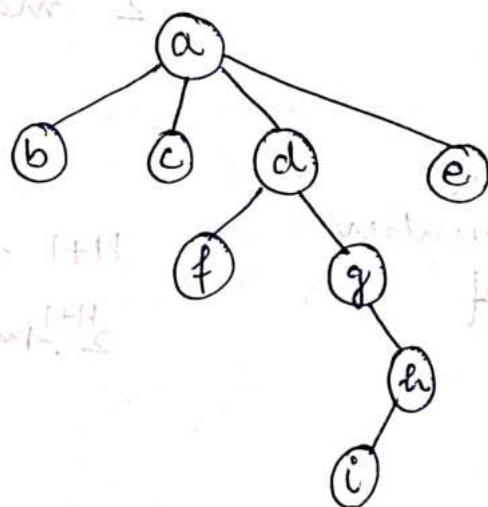
## Conversion of General Tree into Binary Tree

Binary Tree Representation of general tree.

### Leftmost Child Right Sibling Representation

Left child of node of binary tree = leftmost child of the node in general tree.

Right child of node of binary tree = right sibling of the node of general tree.



## Tree Traversal

- Preorder ( +, a, b)
- Inorder ( a, +, b)
- Postorder ( ab, + )

Notation  $t \rightarrow a, t \rightarrow b$

a = left child

t = Root

b = right child

### Preorder Function —

```
void Preorder ( struct BTNode *t ) {
```

if ( t ) {

printf ("%d", t->data);

Preorder ( t->leftchild );

Preorder ( t->rightchild );

### Inorder Function —

```
void Inorder ( struct BTNode *t ) {
```

if ( t ) {

Inorder ( t->leftchild );

printf ("%d", t->data);

Inorder ( t->rightchild );

## Postorder Function

Inorder sort

```

void Postorder (struct BTNode *t) {
    if (t) {
        Postorder (t->left child);
        Postorder (t->right child);
        printf ("%d", t->data);
    }
}

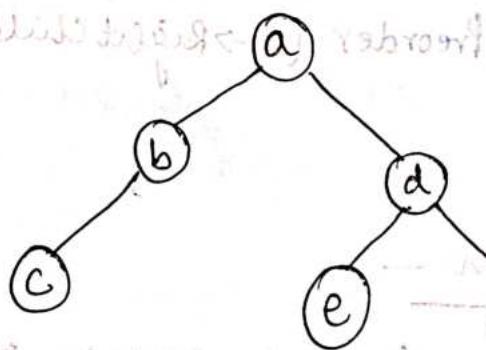
```

Examples

Input: "b..f") printing

((b...f)<-f) reborn

((b...f)<-f) reborn



Preorder = a b c d e f

Inorder = c b a e d f

Postorder = c b e f d a

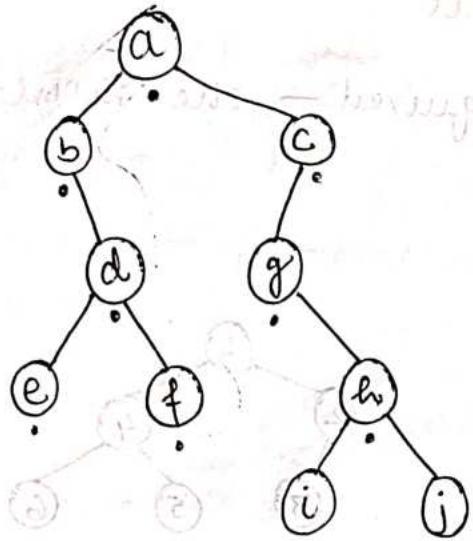
((b...f)<-f) reborn

f(f) II

((b...f)<-f) reborn

((b...f)<-f) reborn

((b...f)<-f) reborn

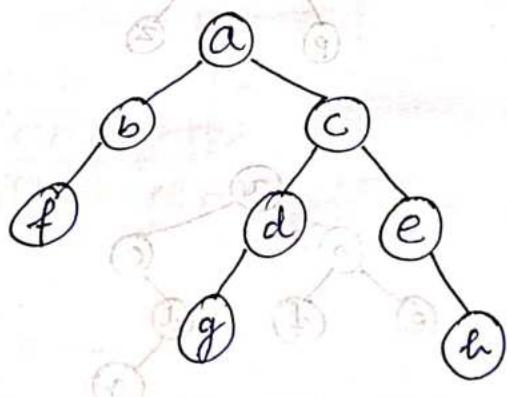


Preorder = a b d e f c g h i j

Inorder = b d e f a g i h j c

Postorder = e f d b i j h g c a

### Level Order Traversal



level  
order  
Traversal

= abc fde ghi

Level order traversal = visiting elements of tree level by level  
Inorder traversal = visiting elements of tree in sorted order

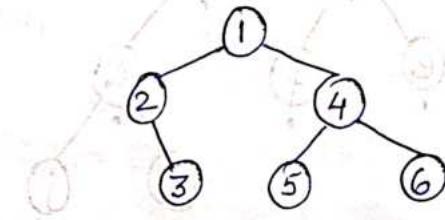
# Constructing Tree using Traversals

Minimum 2 traversals are required - one should be inorder.

## Example

Preorder: 1 2 3 4 5 6

Inorder: 2 3 1 5 4 6

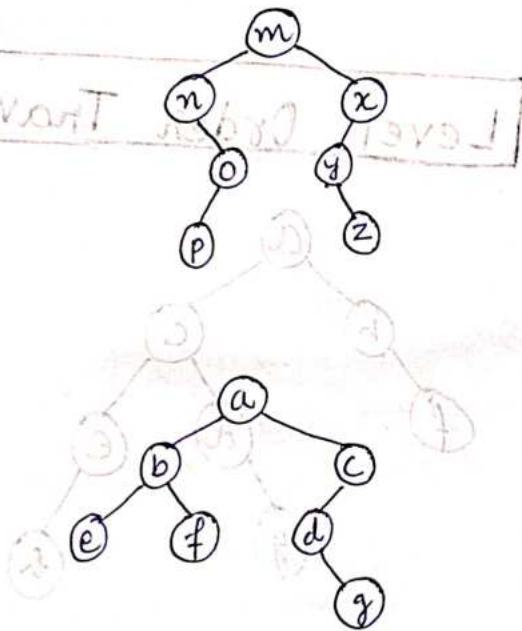


Preorder: m n o p x y z

Inorder: n p o m y z x

Postorder: e f b g d c a

Inorder: e b f a d g c



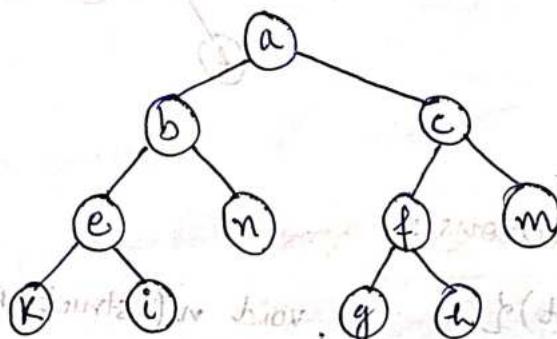
Reverse of converse postorder = Preorder traversal

Reverse of converse inorder = Inorder traversal

\* Preorder and Postorder traversals can uniquely identify a tree only when each node will have either 0 or 2 children.

Preorder: abekincfghm

Postorder: kienbgfhmcda



### Height of Tree

```
int Height (struct BTNode *t) {
```

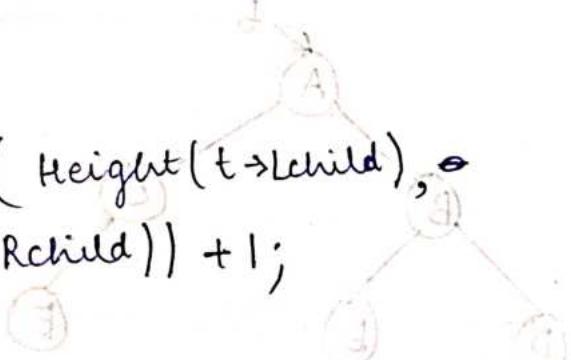
```
    if (t == NULL) {
```

```
        return -1;
```

```
    else {
```

```
        return max ( Height(t->Lchild),
```

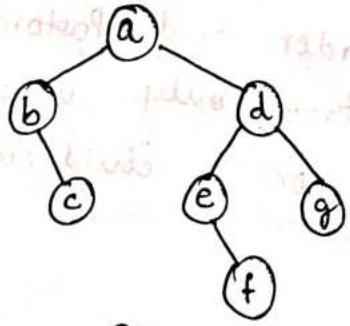
```
                    Height (t->Rchild)) + 1;
```



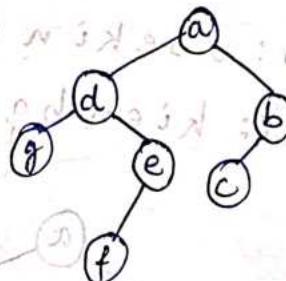
Ques

```
void Do (struct btNode *t) {
```

```
    if (t) {  
        Do(t->leftchild);  
        Do(t->rightchild);  
        swap (t->leftchild, t->rightchild);  
    }
```



what will be the final tree after the above function is executed for given tree?



Ques

Consider the following functions:

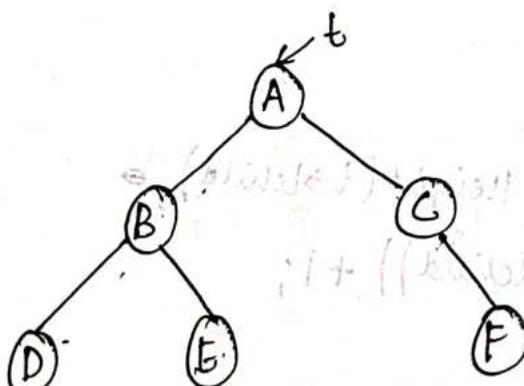
```
void m (struct BTNode *t) {
```

```
    if (t) {  
        n(t->leftchild);  
        printf ("%c", t->data);  
        n(t->rightchild);  
    }
```

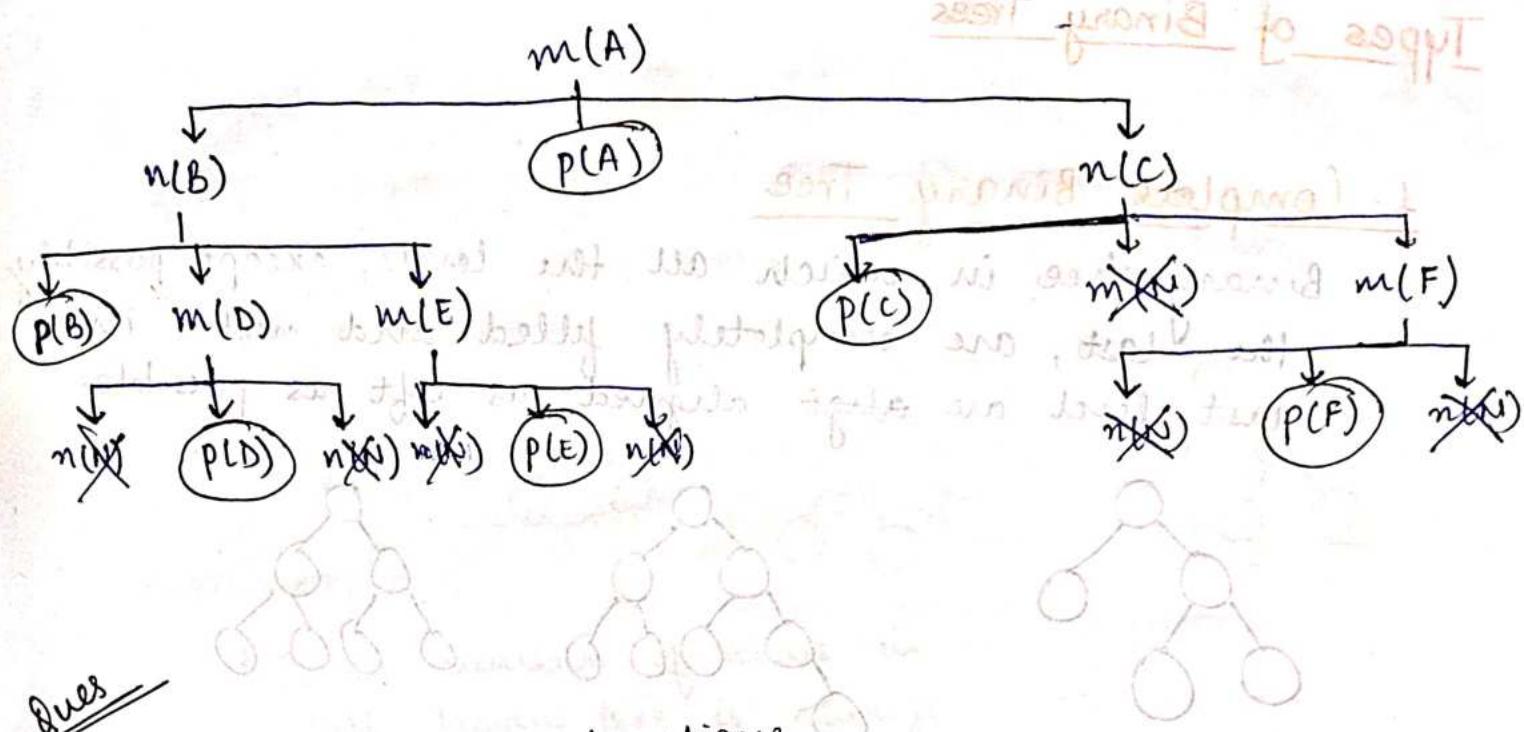
```
void n (struct BTNode *t) {
```

```
    if (t) {  
        printf ("%c", t->data);  
        m (t->leftchild);  
        m (t->rightchild);  
    }
```

Get the value of m(t) for the given tree -



B D E A C F



Ques

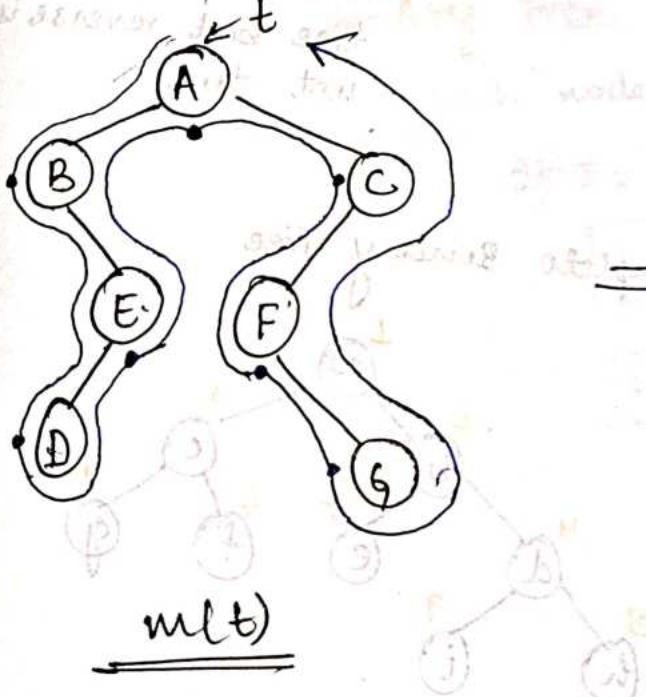
Consider the following functions

```

void m(struct BTNode *t) {
    if(t) {
        n(t->leftchild);
        printf("%c", t->data);
        n(t->rightchild);
    }
}
  
```

```

void n(struct BTNode *t) {
    if(t) {
        printf("%c", t->data);
        m(t->leftchild);
        m(t->rightchild);
    }
}
  
```



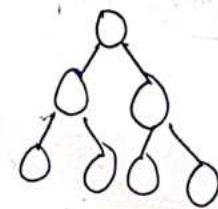
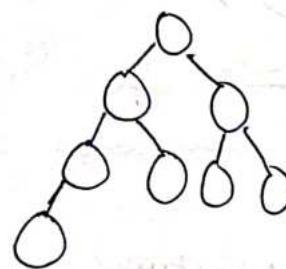
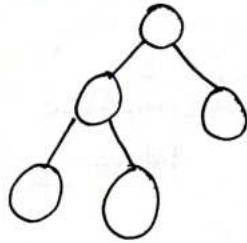
B D E A C F G

+ Xero is to root  
Ib = left child  
Irb = right child

## Types of Binary Trees

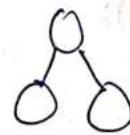
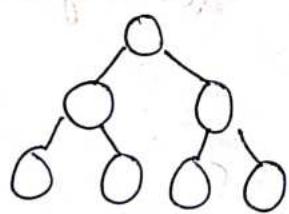
### 1. Complete Binary Tree

Binary tree in which all the levels, except possibly the last, are completely filled and nodes in last level are left aligned as left as possible.



### 2. Strict Complete Binary Tree

Binary tree in which every level is completely filled.



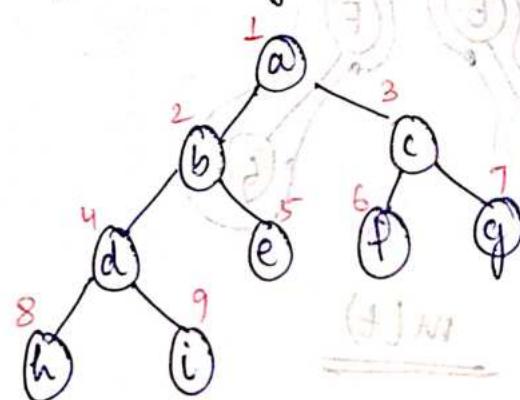
Every strict complete binary tree is a complete binary tree but reverse is not true.

### Array Representation of complete Binary Tree

Root at index 1

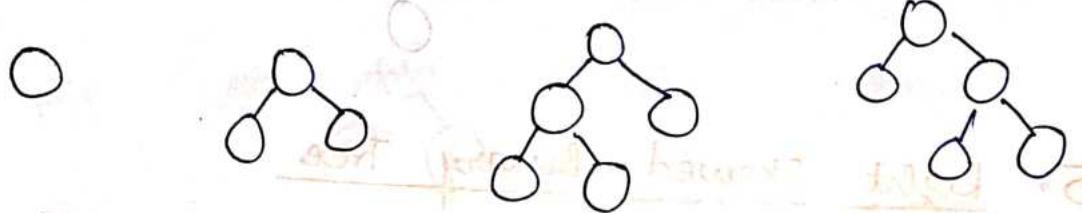
Left child =  $2i$

Right child =  $2i+1$



### 3. Full Binary Tree

- ① also called proper binary tree.
- ② every node has either 0 or 2 children.



\* Total number of nodes in a full binary tree is always odd value.

### 3-Ary Tree

each node has 0 or 3 children.

If there are  $I$  internal nodes (3 children),  
No. of leaf nodes =  $2I + 1$

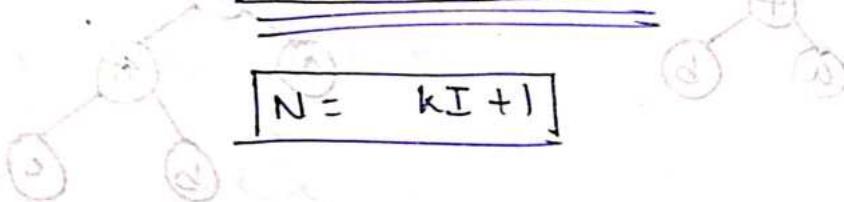
### K-Ary Tree

each node has either 0 or  $k$ -children.

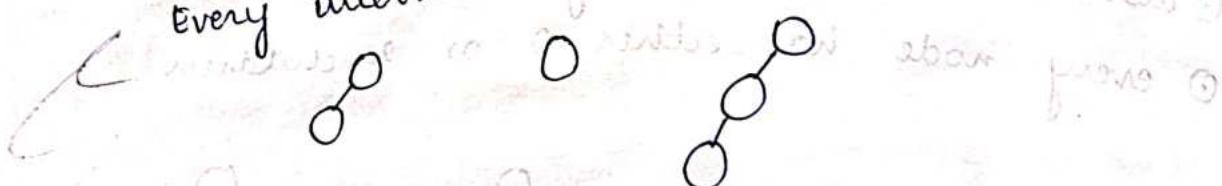
If  $I =$  no. of internal nodes

$$L = (k-1)I + 1$$

$$N = kI + 1$$



4. Left skewed Binary Tree  
Every internal node has only left child.



5. Right skewed Binary Tree

Every internal node has only right child.



## 6. Expression Tree

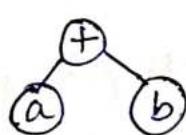
used to represent expression of PI

operator = root

left operand = left child

right operand = right child

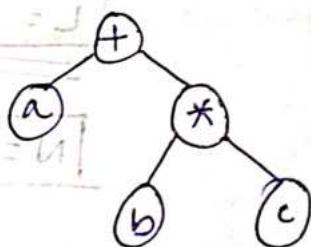
when iteration plan  $a+b*c$



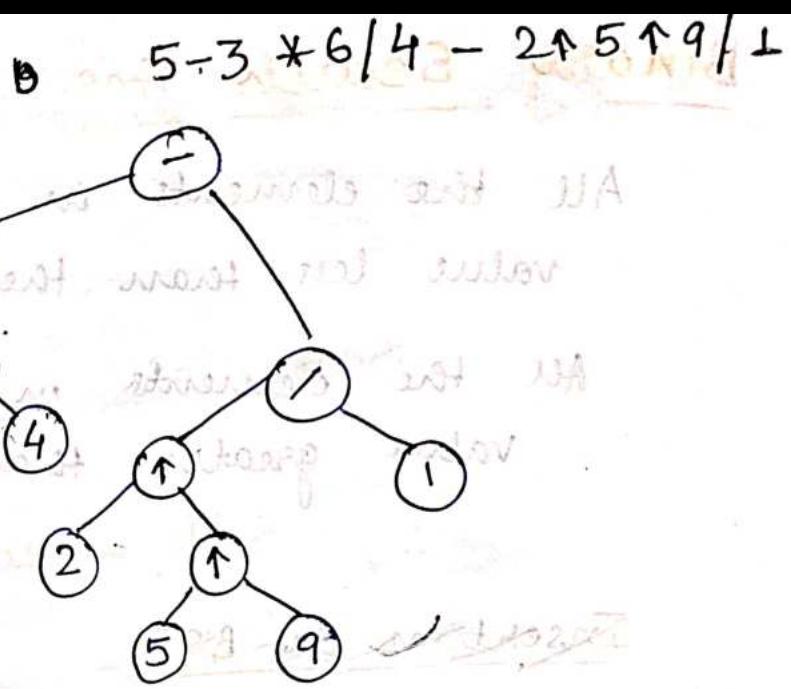
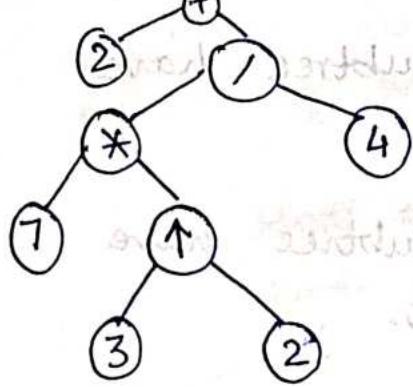
$a+b*c$

$$[(1+I)(1-x)] = J$$

$$[1+Ix - 1] = J$$



$$2 + 7 * 3 \uparrow 2 / 4$$



Draw expression tree for:

$$-b$$

$$\log x$$

$$x!$$

$$\log x!$$

$$\log$$

$$(P1) x$$

$$!$$

$$x$$

$$\log$$

$$!$$

$$x$$

$$81$$

$$71$$

$$61$$

$$5$$

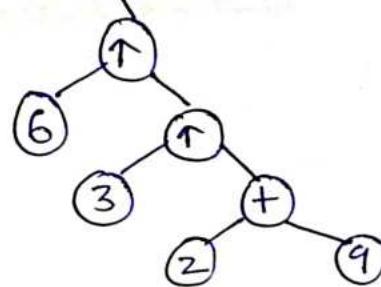
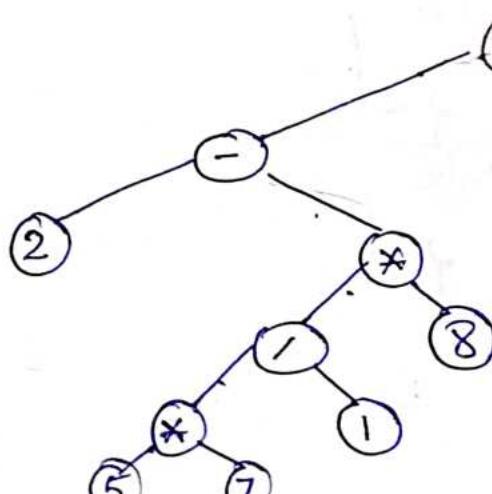
$$4$$

$$3$$

$$2$$

$$1$$

Height  
of tree = 5



## Binary Search Tree

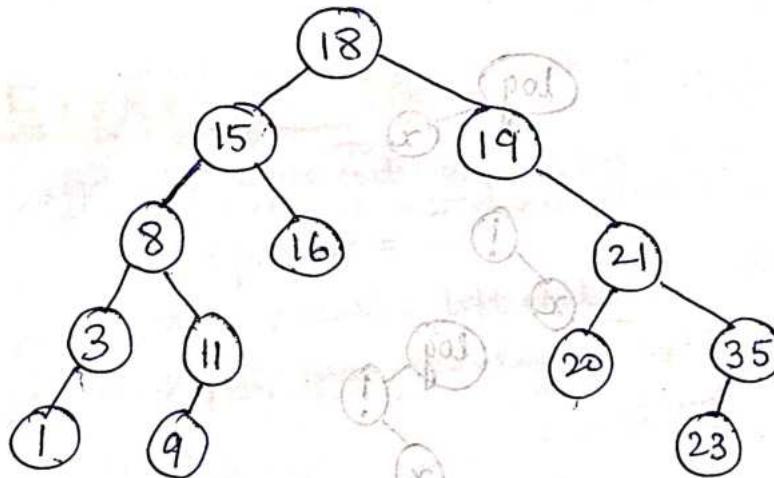
All the elements in the left subtree have value less than the root.

All the elements in the right subtree have value greater than the root.

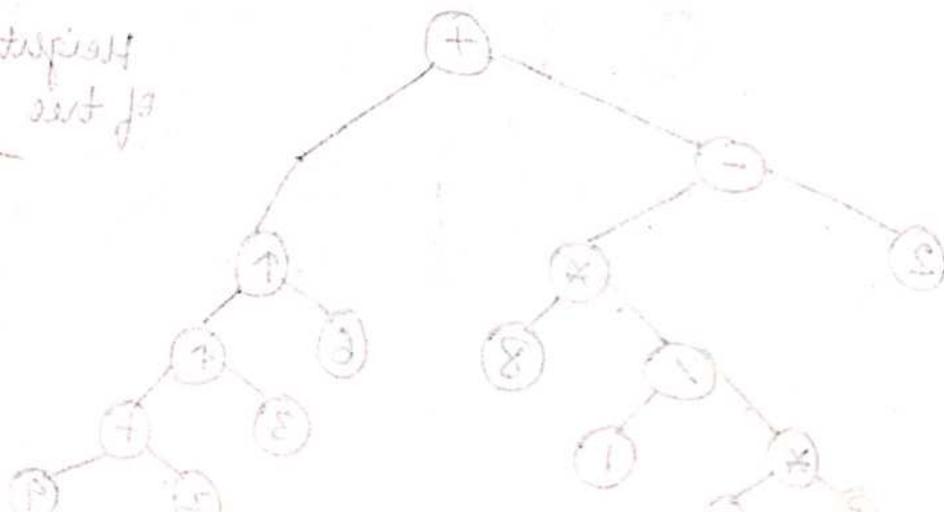
### Insertions in BST

Construct BST using keys

18, 15, 8, 19, 21, 35, 23, 11, 9, 16, 3, 1, 20



~~$c = \text{left } f_3$~~



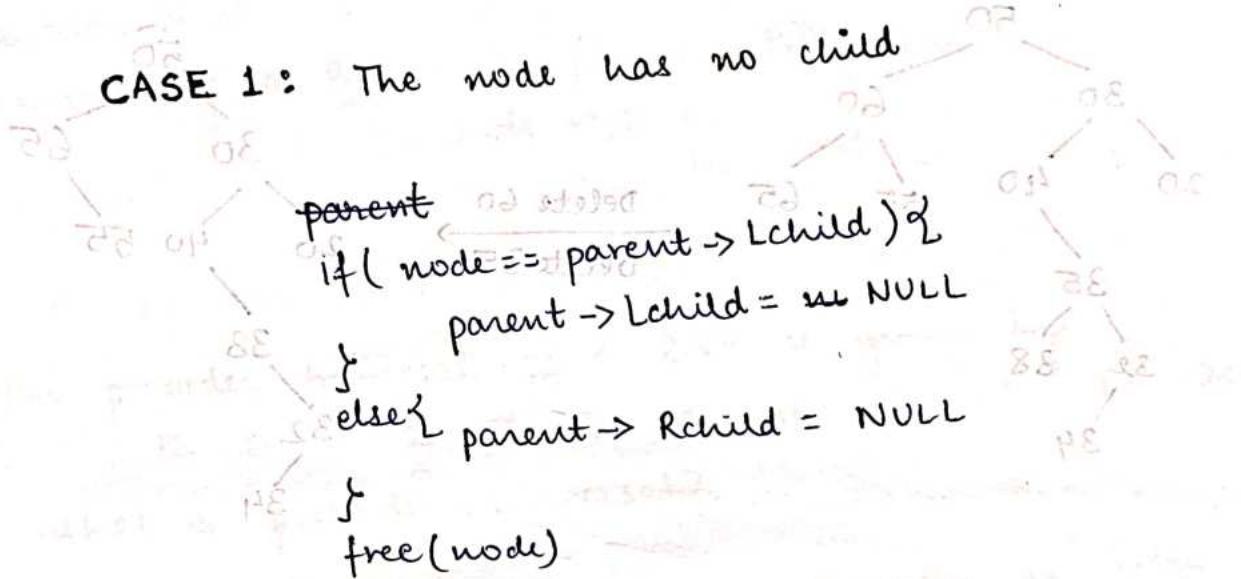
## Deletion in BST

Search for the node in BST. Also, keep track of the parent of the node.

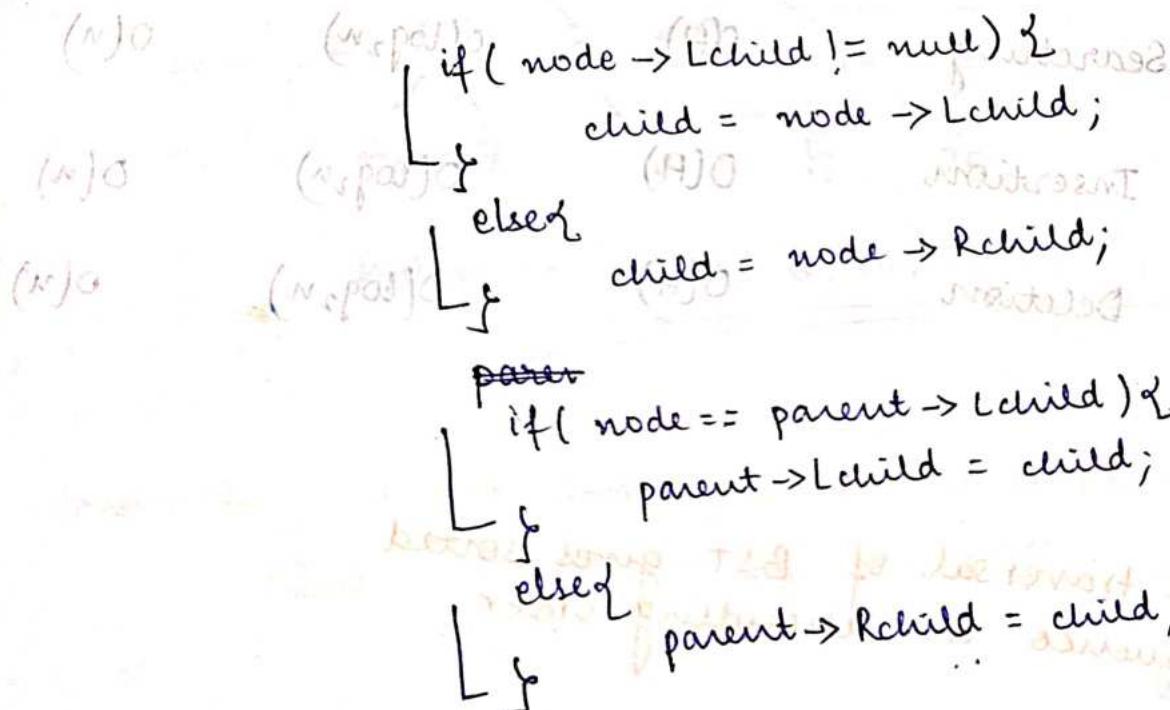
Node to be deleted = \*node

Node to Parent of the node = \*parent.

**CASE 1:** The node has no child

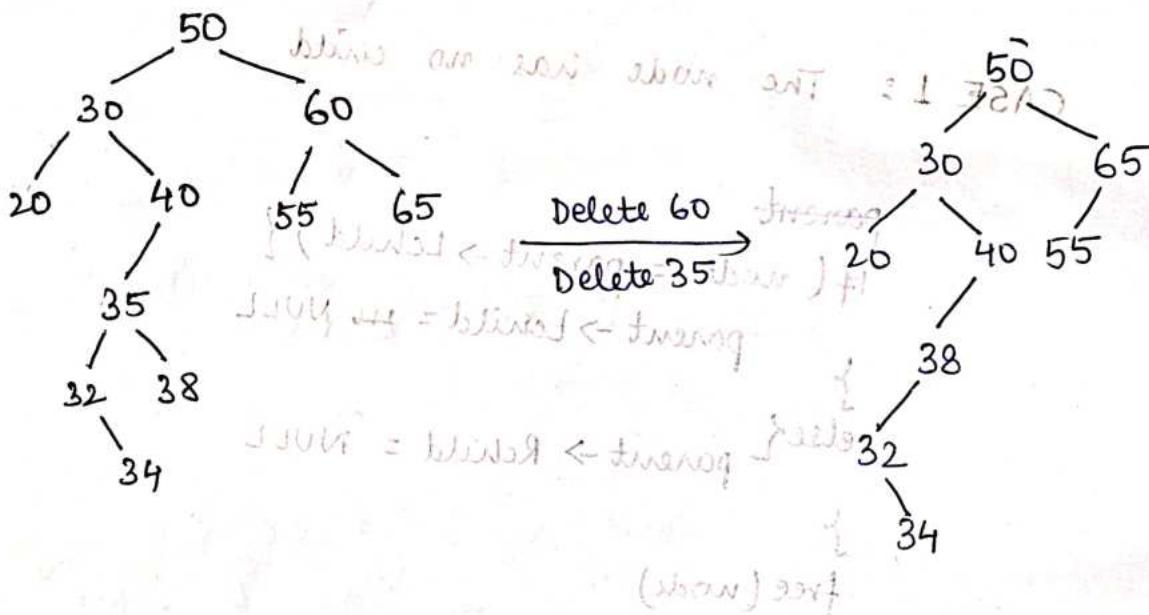


**CASE 2:** The node has one child



CASE 3: The node has 2 children.  
 Replace the node with inorder successor  
 or predecessor.

Delete the inorder successor/predecessor



Runtime complexity of BST -

Searching  $O(H) = O(\log_2 n)$   $O(n)$

$\{ \text{blinks} < than} = \text{blinks}$

Insertion  $O(H)$   $O(\log_2 n)$   $O(n)$

$\{ \text{blinks} < than} = \text{blinks}$

Deletion  $O(H)$   $O(\log_2 n)$   $O(n)$

$\{ \text{blinks} < than} = \text{blinks}$

$\{ \text{blinks} = \text{blinks} < than}$

\* Inorder traversal of BST gives sorted sequence in ascending order.

If preorder traversal of BST 15, 10, 5, 9, 12, 19, 16  
What is inorder traversal?

5 9 10 12 15 16 19

(ascending order)

Insertion in

Constructing a BST from preorder traversal — SIMPLE

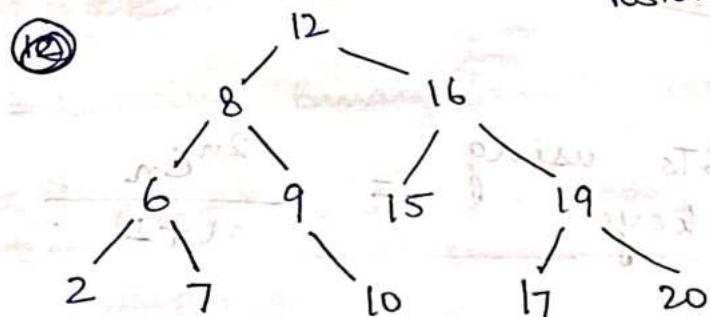
Insert elements one by one  
from left to right

Ques

The preorder traversal of a BST is given by  
12 8 6 2 7 9 10 16 15 19 17 20

What is postorder traversal?

postorder traversal



Postorder: 2, 7, 6, 10, 9, 8, 15, 17, 20, 19, 16, 12

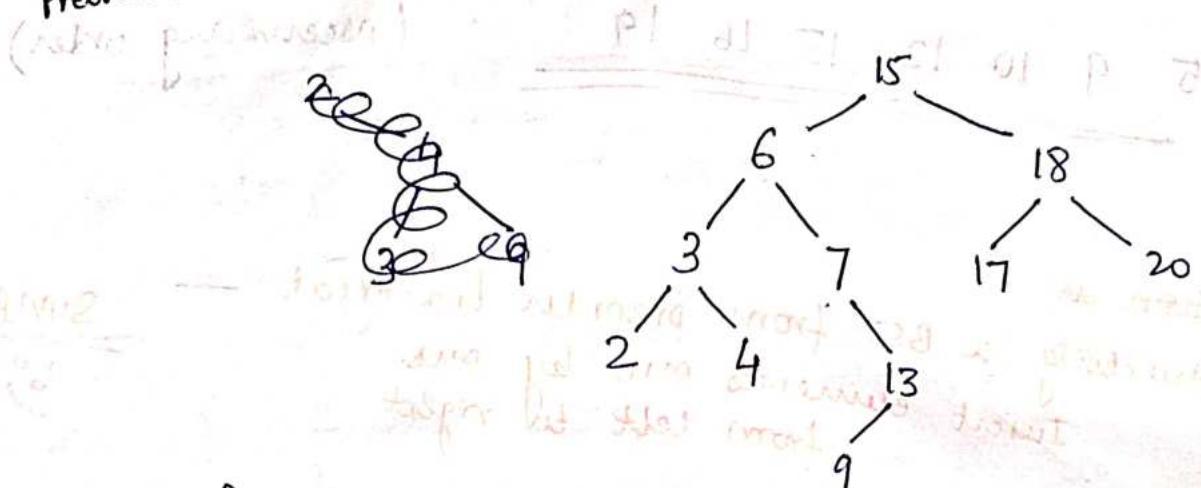
Constructing a BST from postorder traversal —

Insert elements one by one from  
right to left

SIMPLE!!

Ques

Postorder traversal :- 2 4 3 9 13 7 6 17 20 18 15  
 Preorder traversal?



Preorder

traversal :- 15 6 3 2 4 7 13 9 18 17 20

\* Unique BST cannot be constructed if only inorder traversal is given

Number of BSTs using  
n distinct keys

$$\frac{2^n C_n}{n+1}$$

$n$ unlabeled nodes	$n$ distinct keys.
$\frac{2^n C_n}{n+1}$	$\frac{2^n C_n \times n!}{n+1}$
Binary Tree	Binary Search Tree

Catalan number

~~Ques~~ When searching for a key value 60 in a BST, nodes containing values 10 20 40 50 70 80 90 are traversed (not necessarily in same order).

How many different orders are possible in which these key values can occur on the search path from the root to the key containing value 60.

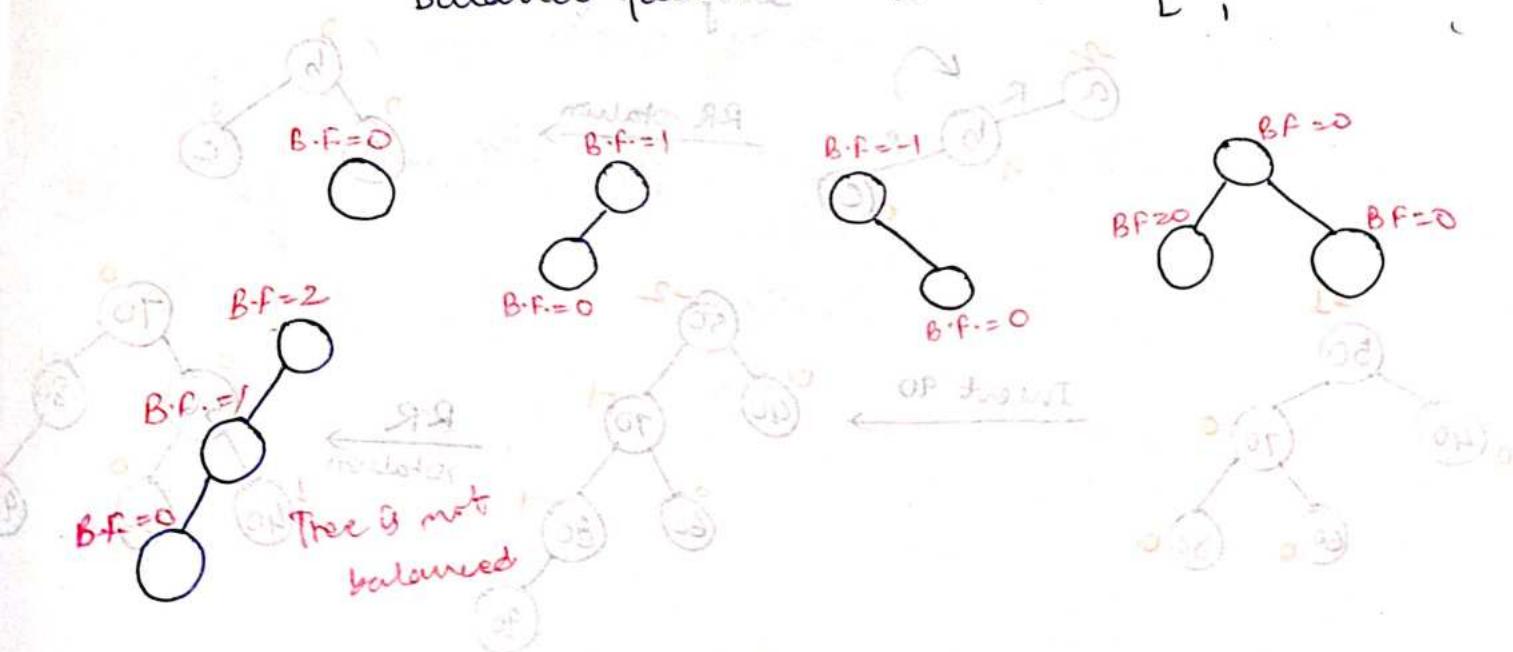
$$\begin{aligned} \text{No. of elements less than } 60 &= 4 \\ \text{No. of "greater" } " " &= 4 \\ \therefore \text{No. of orders possible} &= \frac{7!}{4!4!} = \frac{7 \times 6 \times 5 \times 4 \times 3 \times 2 \times 1}{4 \times 3 \times 2 \times 1 \times 4 \times 3 \times 2 \times 1} \\ &= 35 \end{aligned}$$

## AVL Tree

Self balancing Binary Search Tree.

Height Balanced tree  $\Rightarrow$  for each node, balance factor should be  $b/w -1$  and 1.

$$\text{Balance factor} = h_L - h_R \in \{-1, 0, 1\}$$



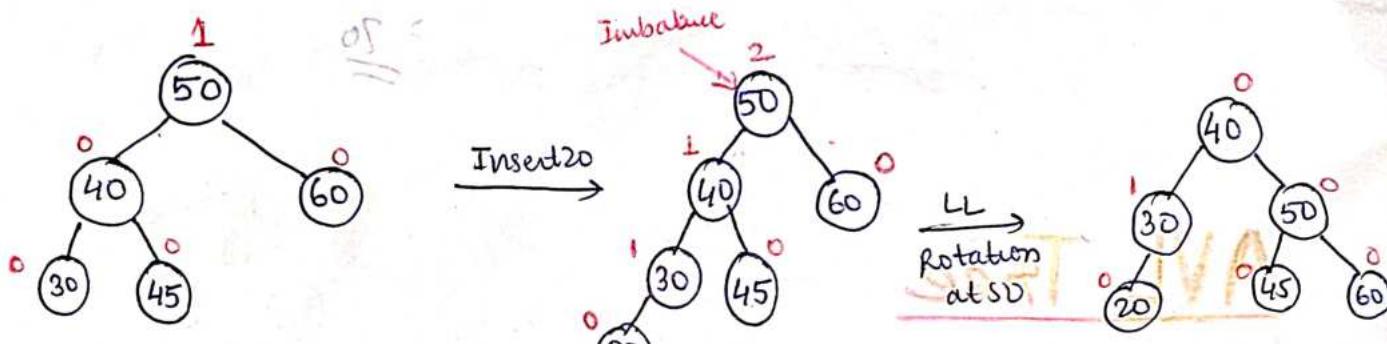
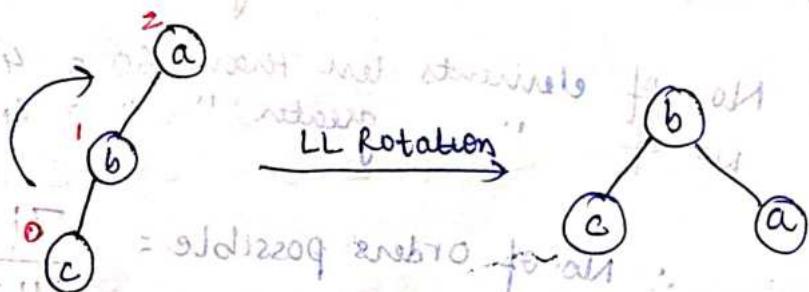
Tree can have unbalance due to insertion or deletion.

## Types of Imbalances

### 1. LL Imbalance (Left Left imbalance)

Solution = LL Rotation

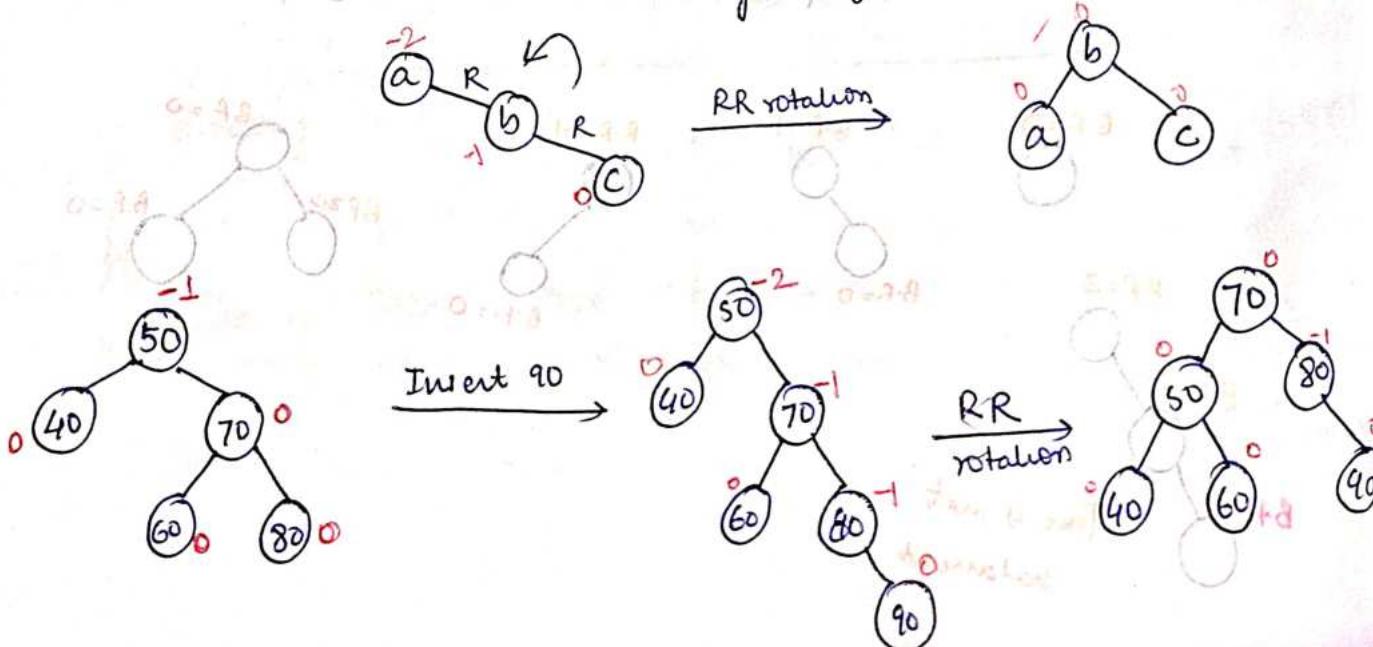
Single Right rotation



### 2. RR Imbalance (Right Right imbalance)

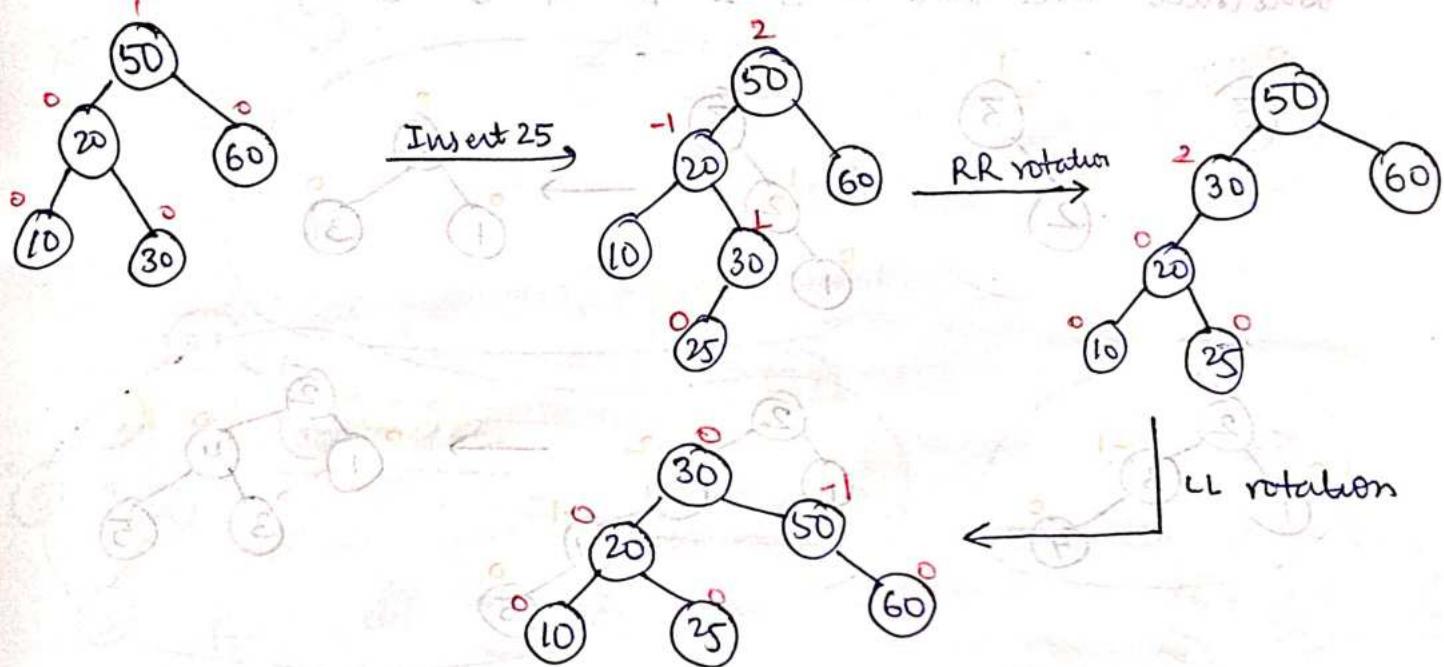
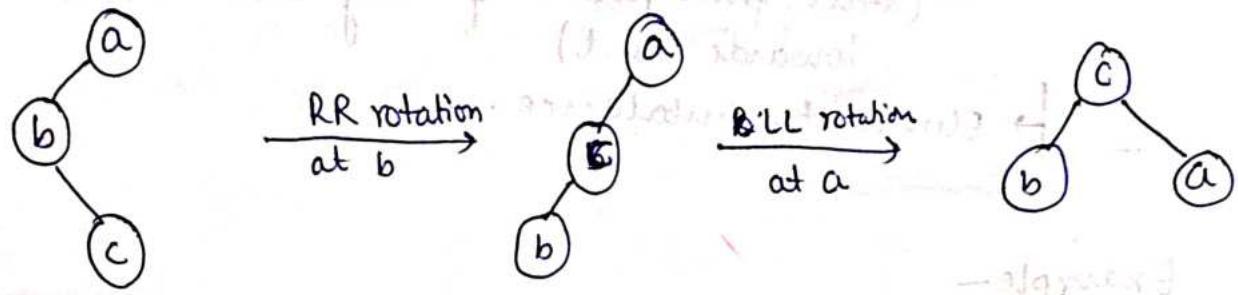
solution = RR Rotation

Single left rotation

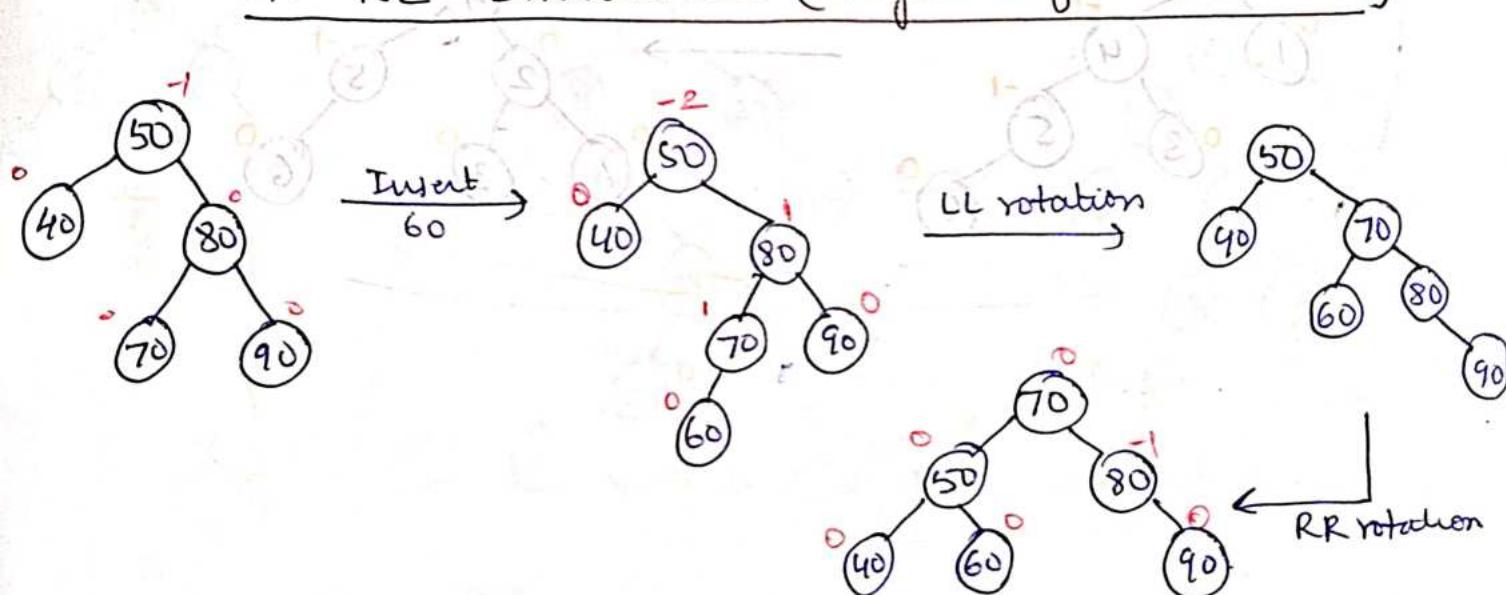


### 3. LR Imbalance (Left Right Imbalance)

solution - RR rotation + LL rotation



### 4. RL Imbalance (Right Left Imbalance)



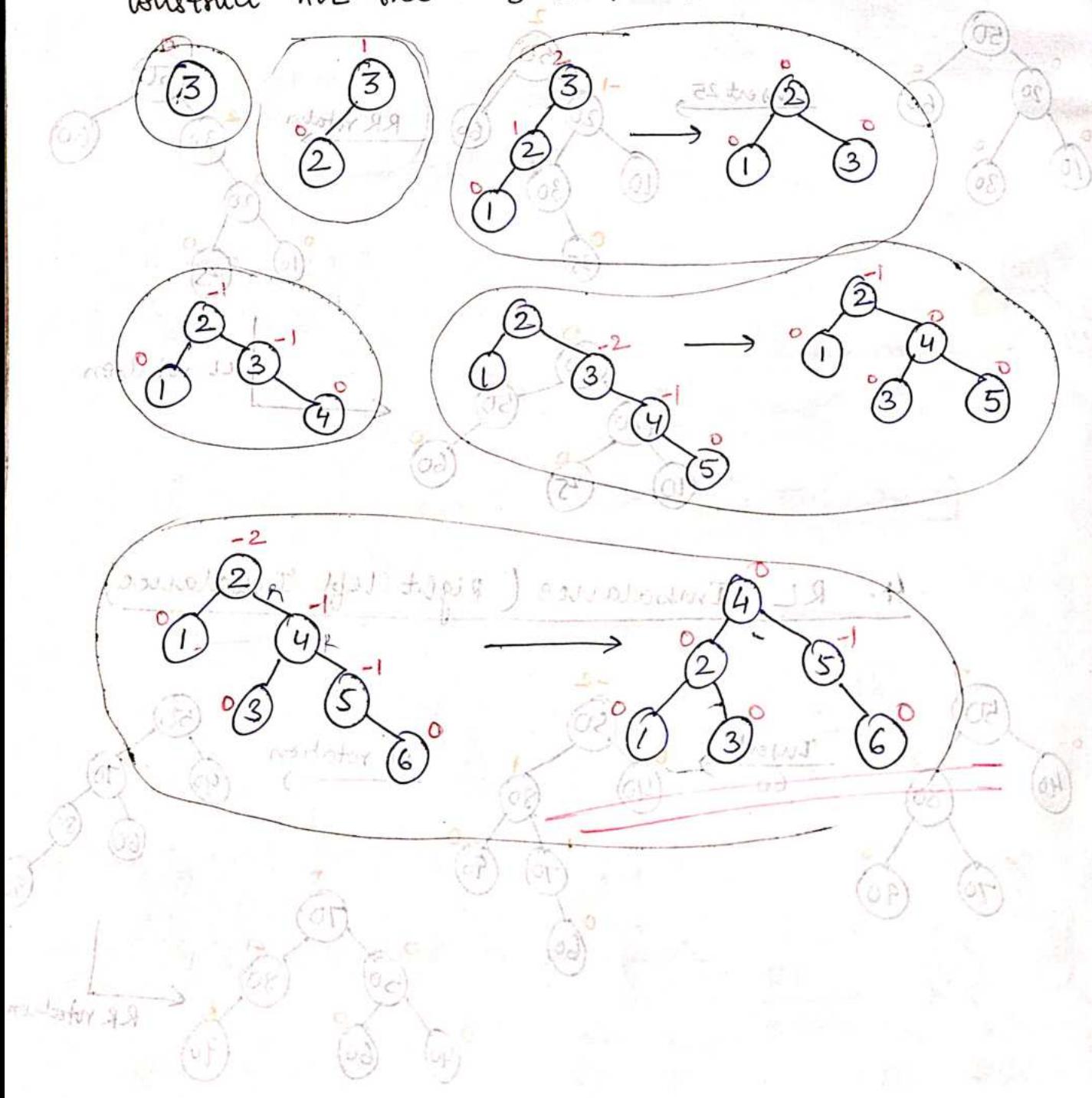
## AVL Tree Insertion -

- Insert key as per BST insertion.
- Identify imbalance, if any (start from parent of newly inserted node & move towards root)

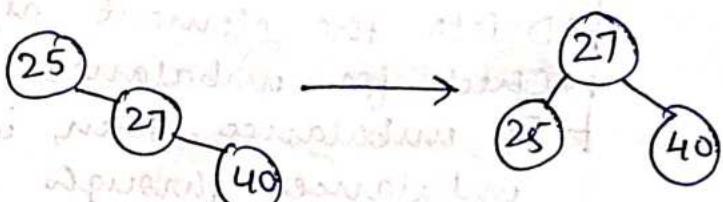
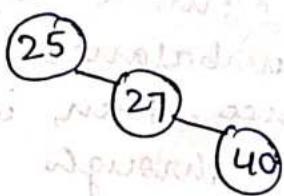
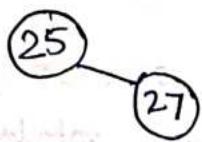
- Eliminate imbalance

### Example -

Construct AVL tree - 3 2 1 4 5 6



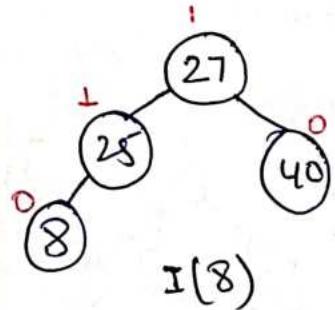
25, 27, 40, 8, 5, 13, 29, 17, 15, 11, 1, 2, 6



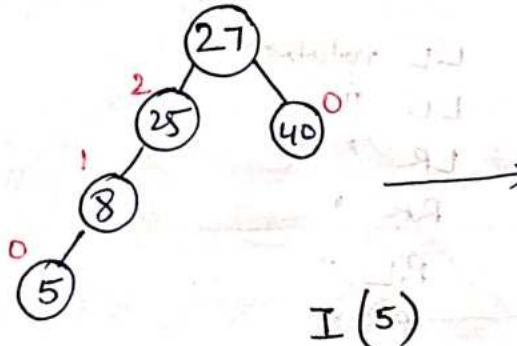
I(25)

I(27)

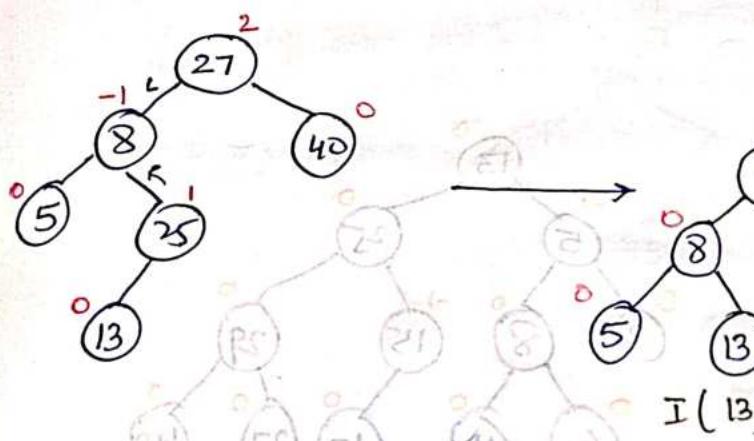
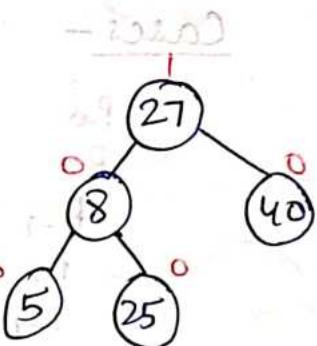
I(40)



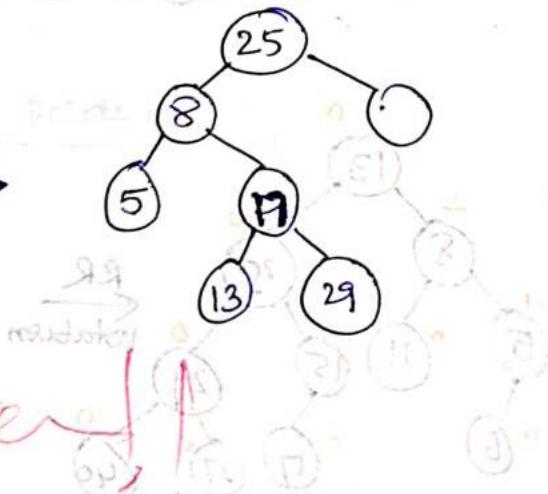
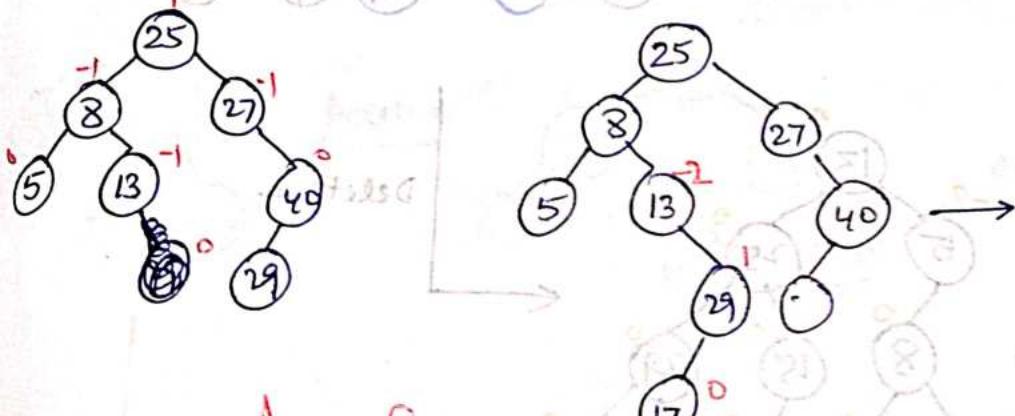
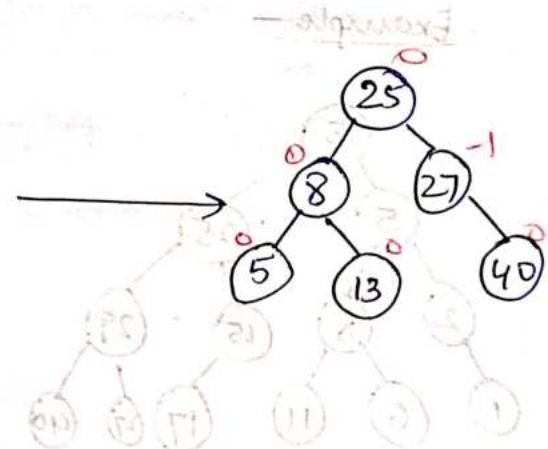
I(8)



I(5)



I(13)

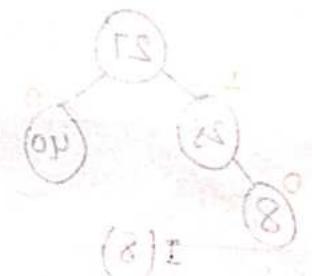
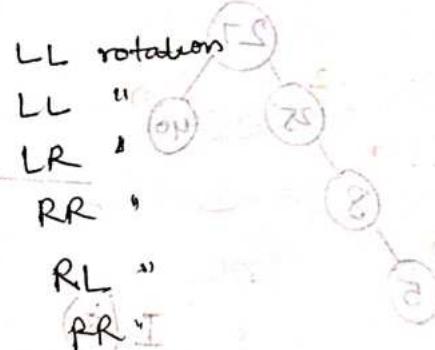
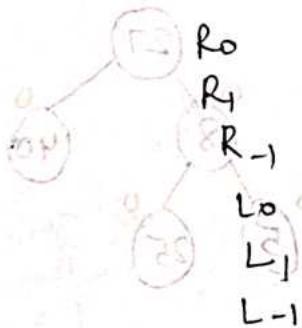


Ho jaeger

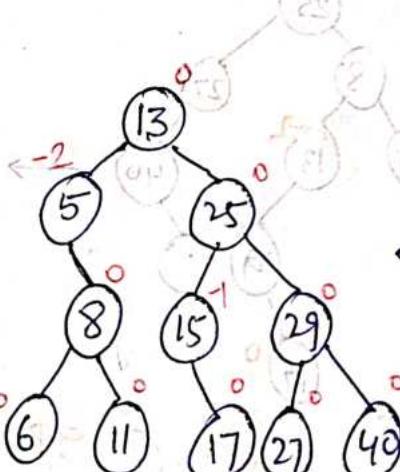
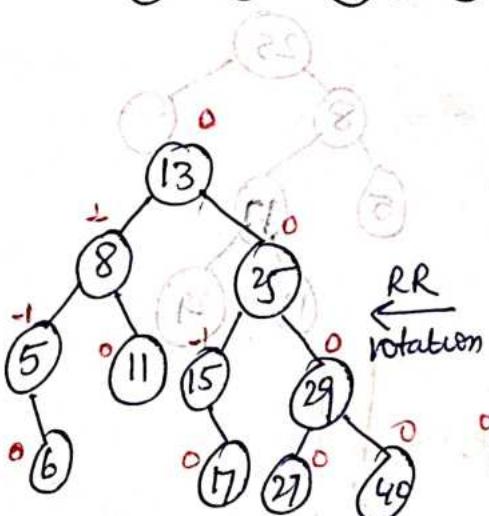
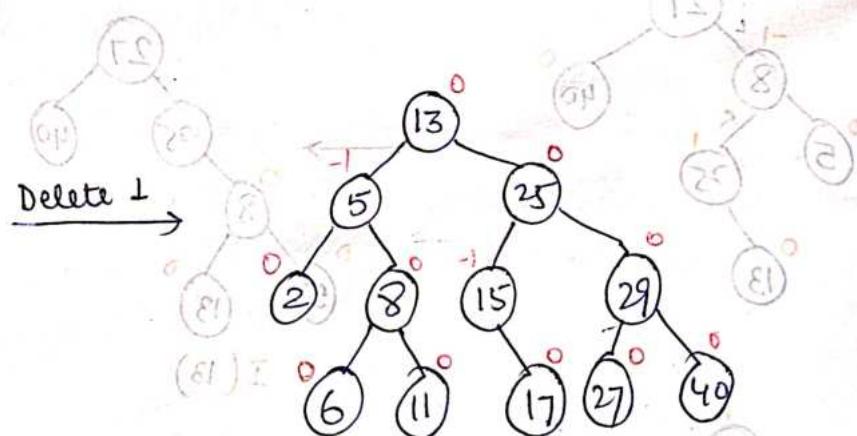
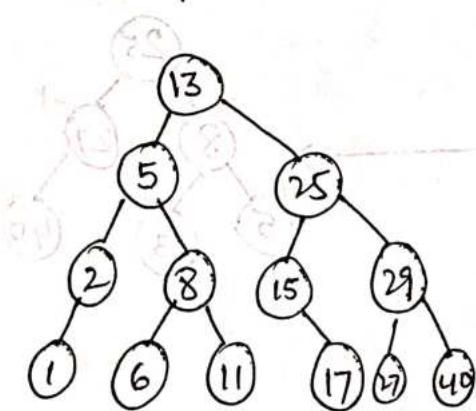
## AVL Tree Deletion

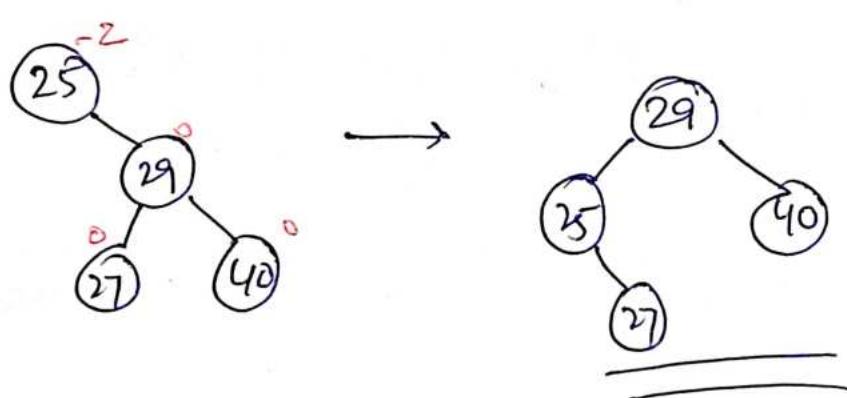
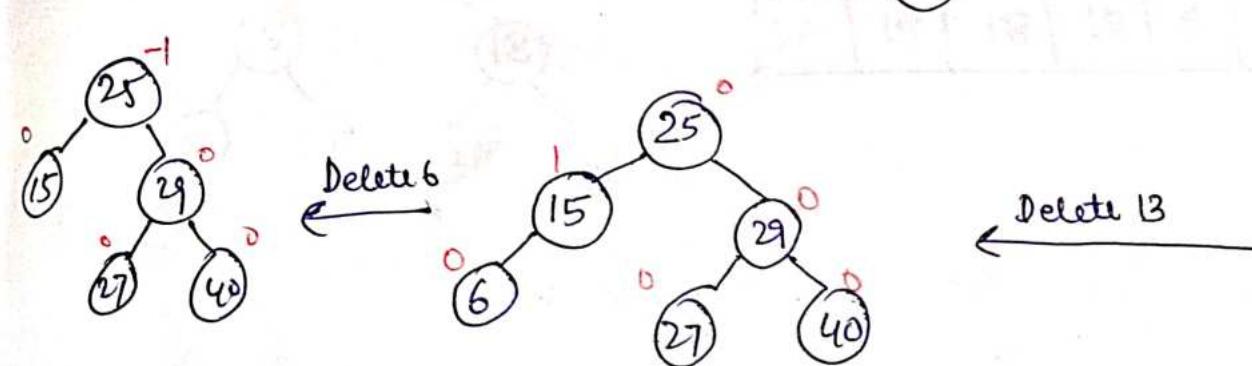
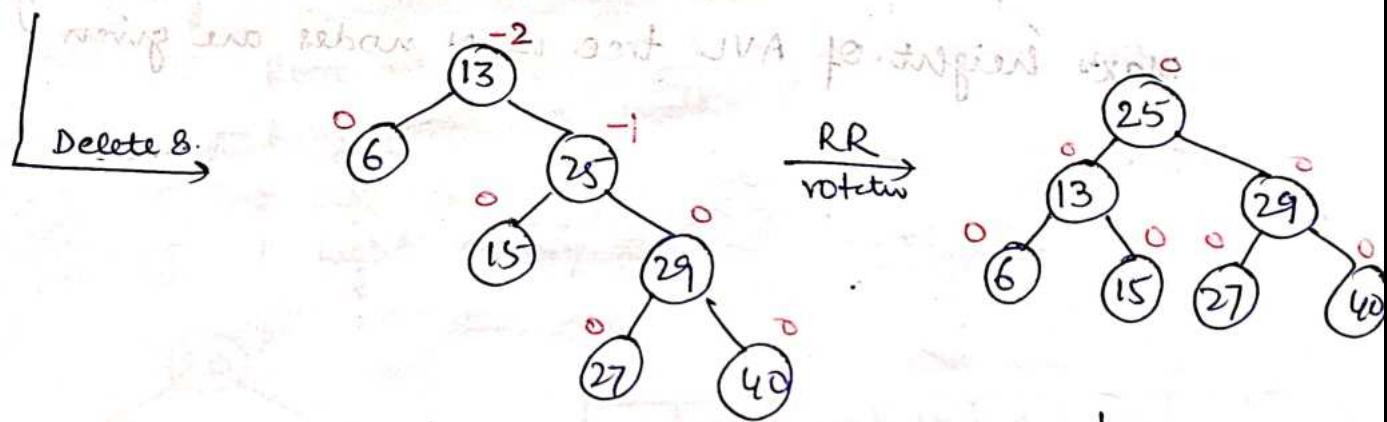
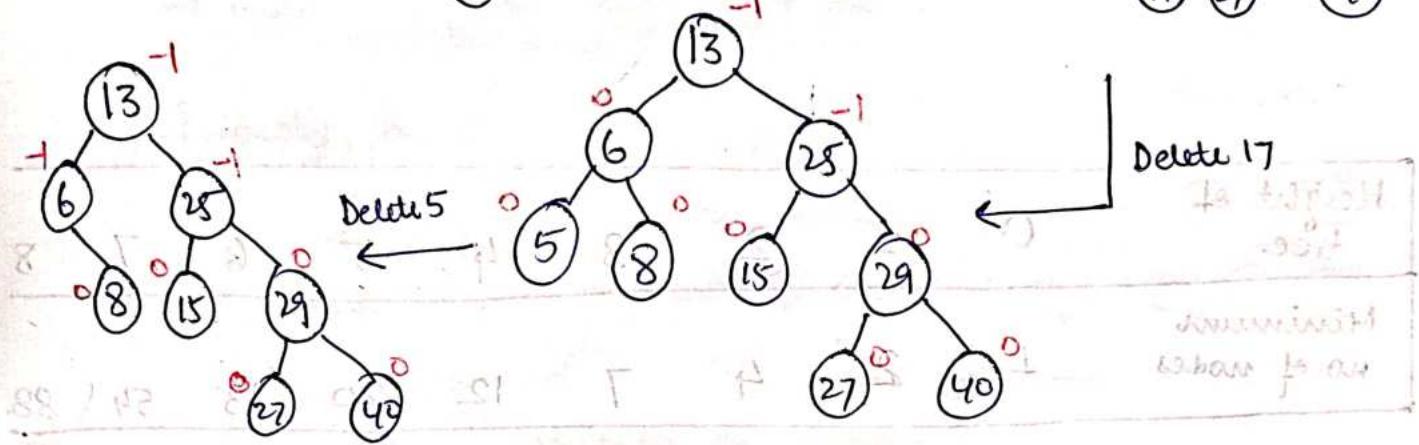
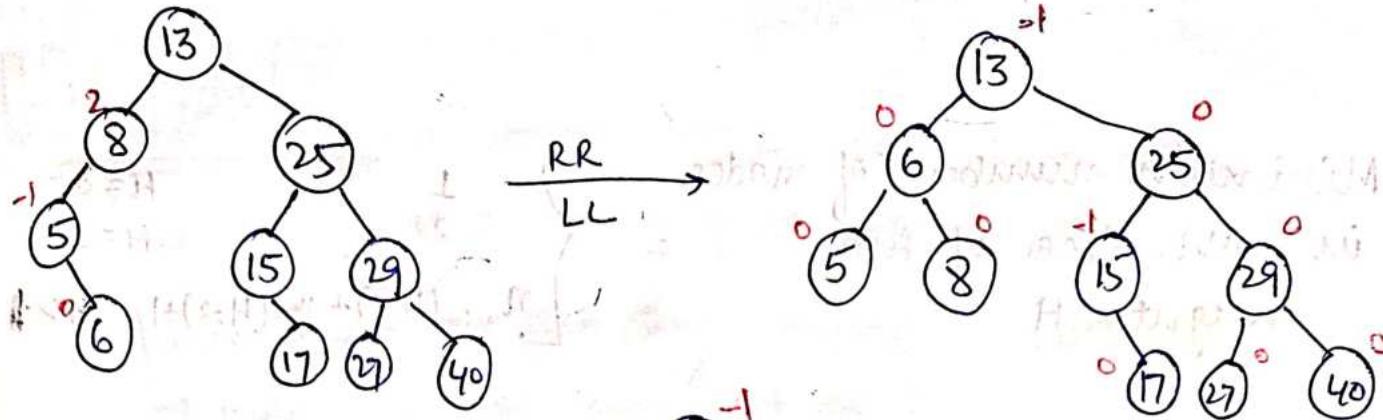
- Delete the element as per deletion in BST
- Check for imbalance
- If imbalance, then, identify a case and eliminate imbalance through appropriate rotation.

### Cases -



### Example -



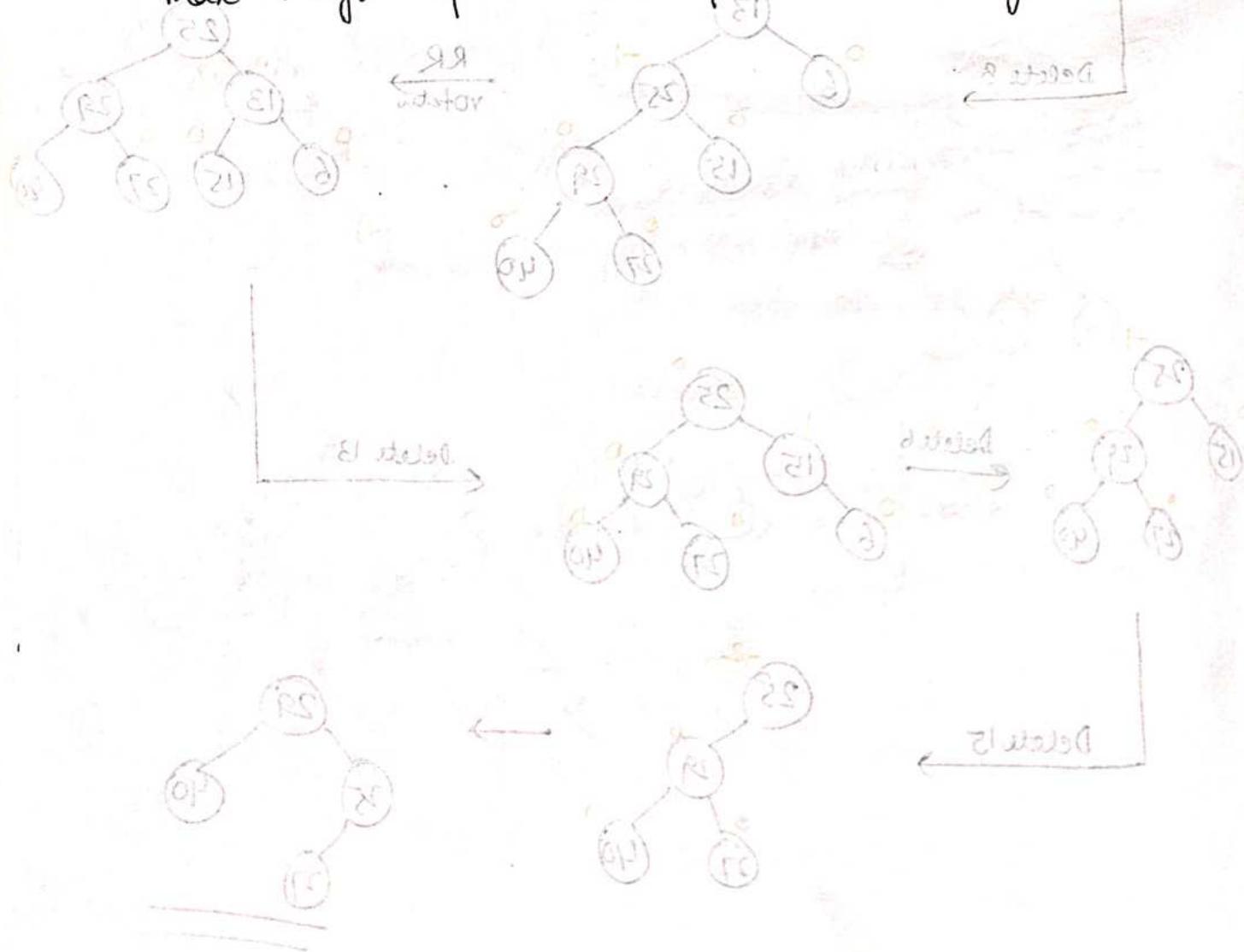


Minimum number of nodes  
in AVL tree of height H

$$= \left\{ \begin{array}{l} 1 \\ 2 \\ n_{min}(H-1) + n_{min}(H-2) + 1 \end{array} \right. \quad H > 1$$

Height of tree	0	1	2	3	4	5	6	7	8
Minimum no. of nodes	1	2	4	7	12	20	33	54	88

Max height of AVL tree of N nodes are given ↗



# HEAP

open down air max 92.5T

tree based data structure

- Complete binary tree
- Max heap or min heap property

Priority based data structure

Max heap → value of root is greatest.

Min heap → value of root is minimum

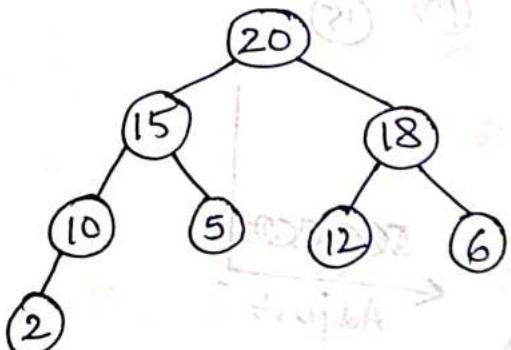
Array Representation of heap -

Root at index  $\Rightarrow 1$

→ Any node at index  $i$

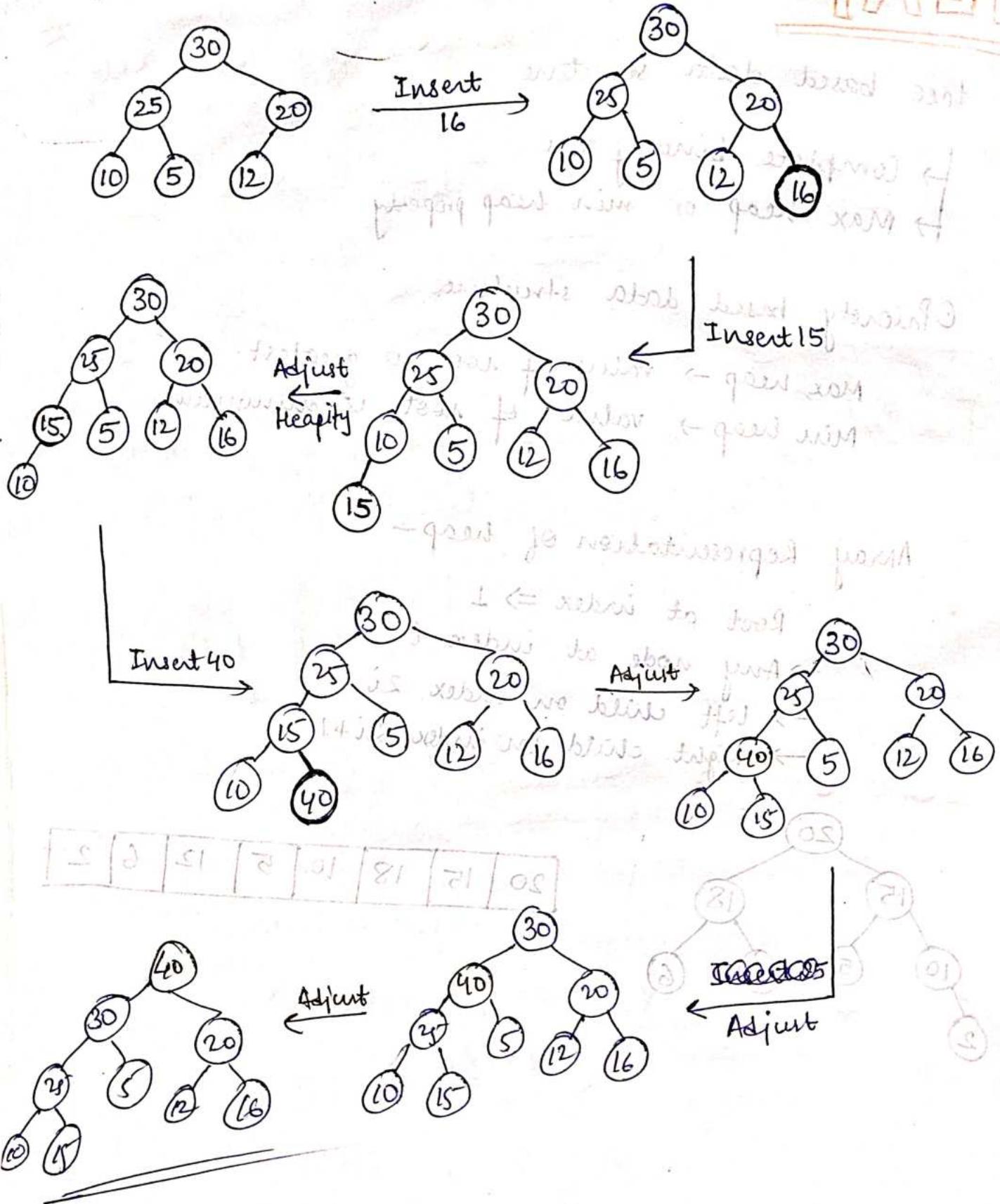
→ left child on index  $2i$

→ right child on index  $2i+1$

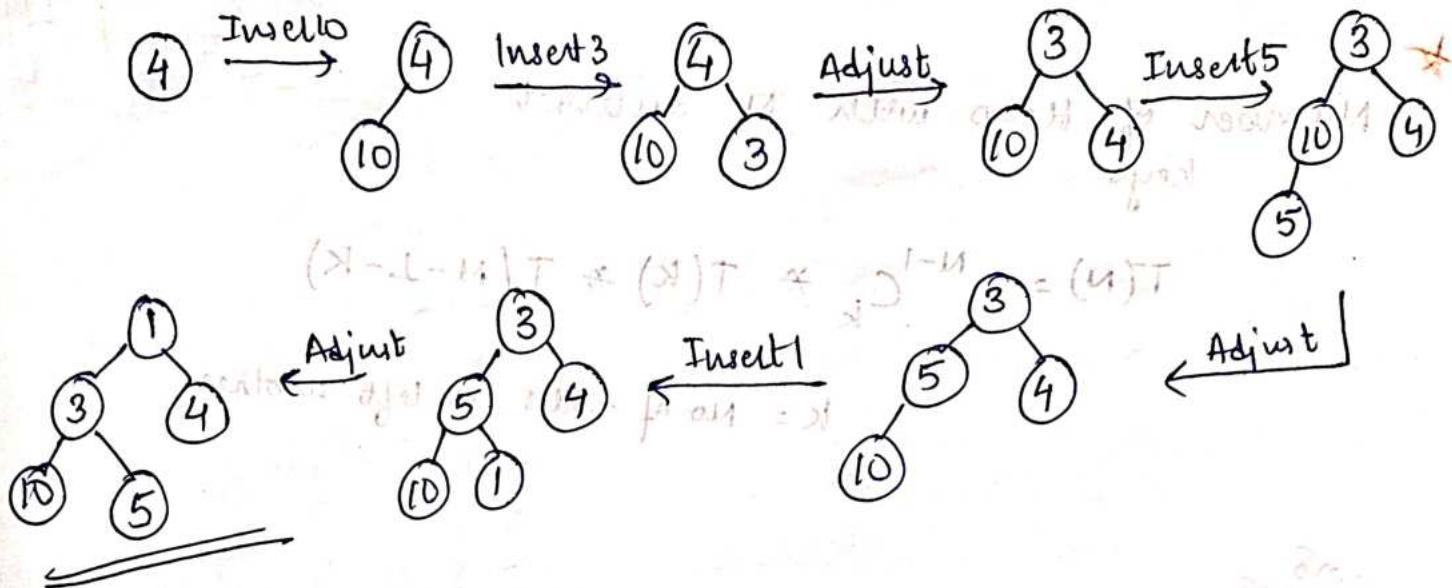


20	15	18	10	5	12	6	2
----	----	----	----	---	----	---	---

# Insertion in max heap



Build Min heap: 4 10 3 5 1



Runtime complexity —

Inserting 1 element =  $O(\log n)$

Inserting  $n$  elements =  $O(n \log n)$

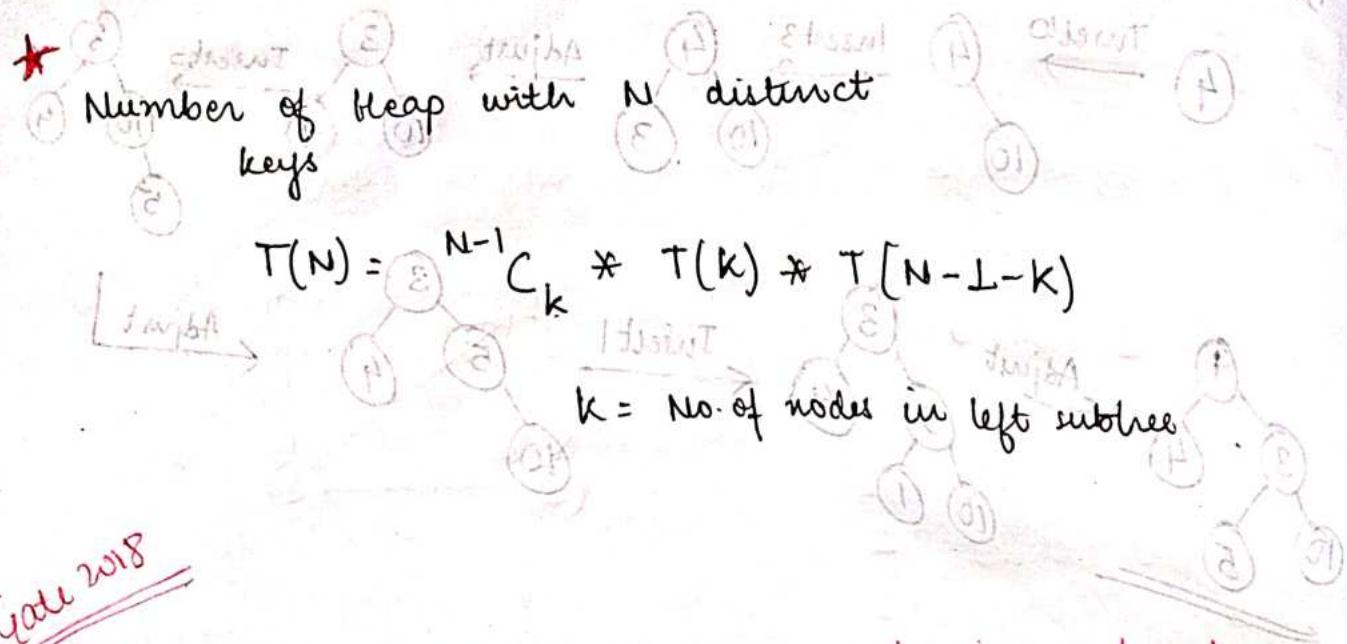
Building heap (one by one insertion) =  $O(n \log n)$

Building heap by using heapify method =  $O(n)$

Deletion in heap —

→ Delete root element & place the last node of C.B.T. at root.

→ Heapify the root  $O(\log n)$



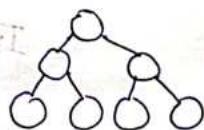
Gate 2018

The number of possible minheaps containing each value from  $\{1, 2, 3, 4, 5, 6, 7\}$  exactly once?

Root of the tree = 1 (since min heap)

No. of nodes in left subtree = 3

No. of nodes in right subtree = 3



No. of ways in which left subtree can be formed =  ${}^6C_3 \cdot T(3) \cdot T(3)$

To calculate  $T(3)$  -

$T(0)$

$$T(3) = 2 * T(1) * T(1)$$

$$= 2 * 1 * 1 = 2$$

$$\therefore T(6) = {}^6C_3 \cdot T(3) \cdot T(3)$$

$$= \frac{16}{3! \cdot 3!}$$

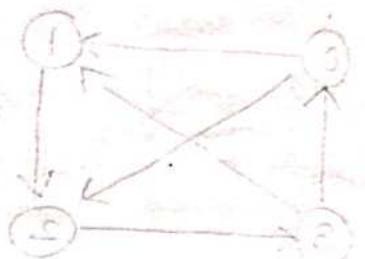
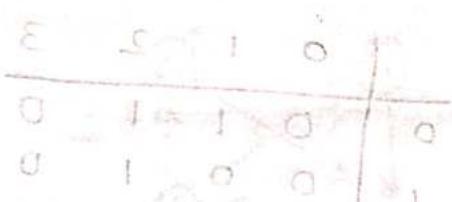
$$* 2 * 2 = \frac{8 \times 5 \times 4}{3 \times 2} \times 2 \times 2$$

$$= \underline{\underline{80}}$$

## Heap sort

Runtime complexity =  $O(n + n \log n) = O(n \log n)$

Space complexity =  $\Theta(n)$

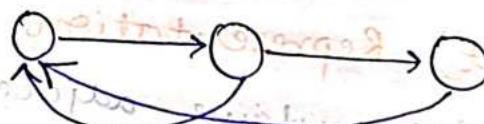


## GRAPH

non linear data structure consisting of nodes and edges.

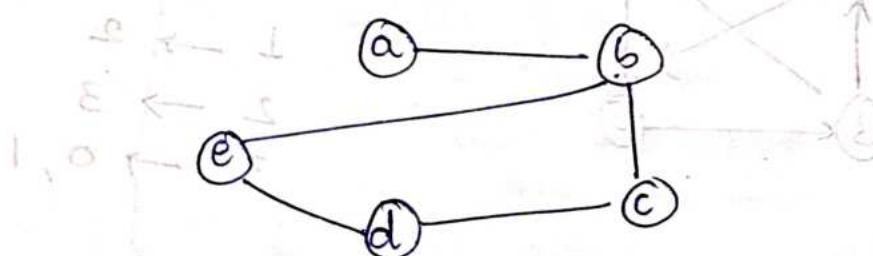
### Directed graph -

Each edge has one direction.



### Undirected graph -

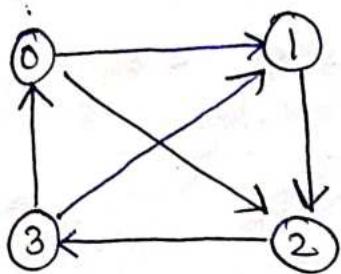
Edges do not have any direction.



## Graph Representation -

### 1. Adjacency Table or Matrix

$n \times n$  matrix where  $n = \text{no. of nodes/vertices}$ .



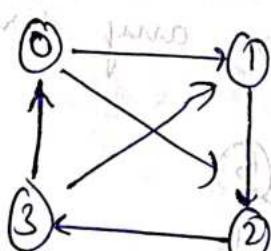
	0	1	2	3
0	0	1	1	0
1	0	0	1	0
2	0	0	0	1
3	1	1	0	0

If weighted graph is given,

$\text{adj}[i][j] = \infty 0$  if no edge b/w i & j  
 $w$  if edge of weight w exists.

### 2. Linked (List) Representation

Every node contains adjacency list for outgoing edges.



Adjacency List



0 → 1, 2

1 → 2

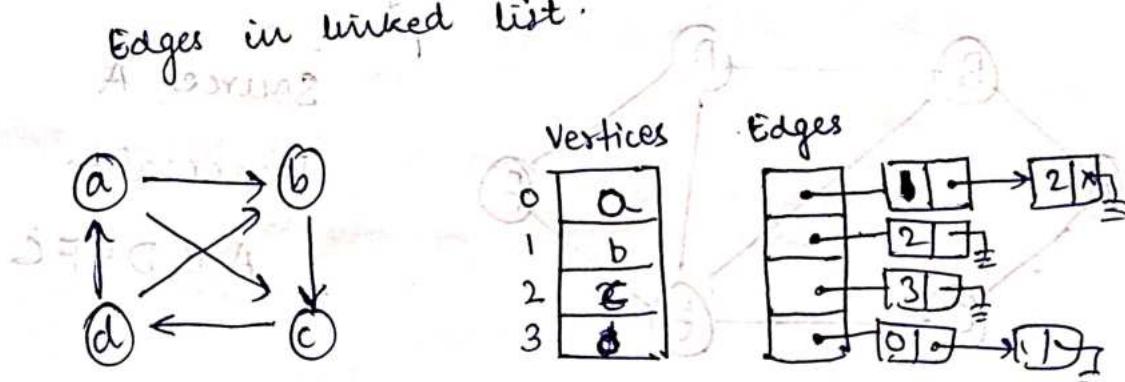
2 → 3

3 → 0, 1

### 3. Hybrid Representation

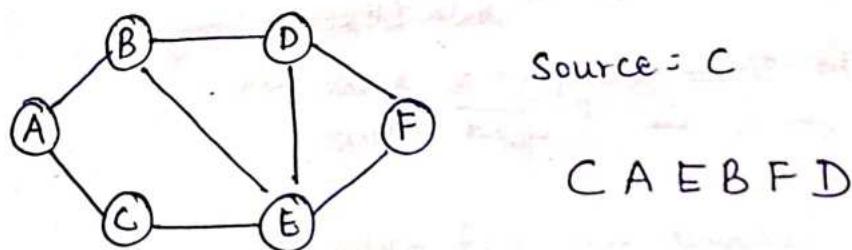
Vertices in array between zigzag lines.

Edges in linked list.



### Traversing in graph

1. Breadth first search (BFS)  
implemented using queue



BFS( $u$ ) {

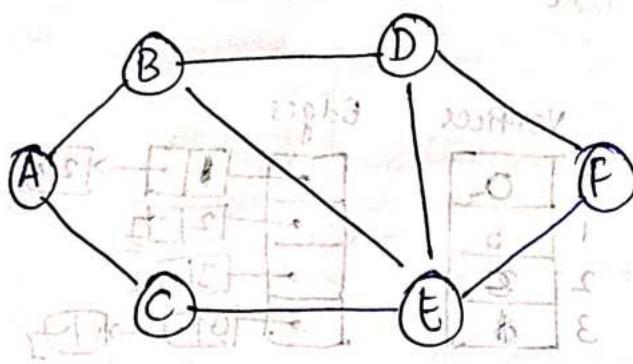
Mark  $u$  as visited and Enqueue by  
while  $q$  is not empty

$x = \text{Dequeue}$

Find all neighbours of  $x$  which  
are unvisited and enqueue  
them.

## 2. Depth First Search (DFS)

implemented using stack.



Source A

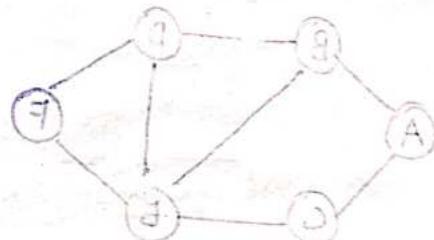
Traversal -

A B D E F C

(278) address from above

comes from between

C A E B F D



5 (u) 278

(u) answer E has been used so don't  
choose for it p view

answer A is correct because the level  
number from beginning is 1



# HASHING

Divide & Conquer Method

Searching technique which can provide result in constant time

[Hash function is used to generate free address / location of element]

Hashing Techniques -

1. Direct Hashing
2. Subtraction Hashing
3. Division Hashing
4. Fold Shifting Method
5. Fold Boundary Method
6. Digit Extraction Method
7. Mid Square Method

## 1. DIRECT HASHING

$$H(k) = k$$

if there is a hash table of range 00 to 99,  
possible keys to be stored are  $0 \rightarrow 99$ .

Disadv:- keys are limited by location range.

## 2. SUBTRACTION HASHING

$$H(k) = k - x^k \quad x \text{ is a +ve integer.}$$

If location range is 0 to 99.  $x=3$   
possible keys  $\rightarrow 3 \text{ to } 102$

Disadv:- keys are limited by location range

### \* 3. DIVISION METHOD (Modulo Division method)

$$L = H(k) = k \bmod m$$

usually  
fn is prime]

Result  $\rightarrow$  0 to  $m-1$ .

Adv:- unlimited range of keys can be stored.

Disadv:- collision is possible.

### \* 4. FOLD SHIFTING METHOD

Location range  $\Rightarrow$  000 - 999

Form groups of 3 and add them to get the location in hash table.

key: 163 121 382  $\rightarrow$  location.  
163  
121  
382  
666

If sum  $\geq 1000$ , ~~L = (sum) % 1000~~

### 5. FOLD BOUNDARY METHOD

Location range  $\Rightarrow$  000-999

For groups of size 3 only at the boundaries and add them to get the location.

key: 123 412 635

123  
635  
758  
 $\rightarrow$  location

If sum  $\geq 1000$ ,  $L = (\text{sum}) \% 1000$

## 6. DIGIT EXTRACTION METHOD

Location range - 000-999

$$L = H(k)_{1,3,5} \quad (\text{1st, 3rd, 5th})$$

Ex → key = 629 843,5 = 011H

694 - Location

## 7. MID SQUARE METHOD

Take square of the key and pick middle digits.

$(12345^2 = 152275)$  material to generate random numbers  
and no. of digits from left to right  
equal to number of digits in key

or base (any) = any

material for any base  
generator  
seed

## Collision

situation that occurs when two distinct keys have the same hash value (location)

Example -

$$H(k) = k \cdot 10$$

Keys to insert - 12, 52      collision occurs

## Collision Resolution techniques

Open Addressing  
(closed hashing)

- Linear probing
- Quadratic probing
- Random probing
- Double Hashing

Closed Addressing  
(open hashing)

- chaining

### 1. Linear probing

if collision occurs at location  $L (= H(k))$ , algorithm looks for next available slot in the hash table to store collided key.

#### Applications

- Symbol Table
- caching
- Databases

$$L_{\text{new}} = (L+i) \bmod m$$

- \* Linear probing — Primary clustering
- \* Quadratic probing — Secondary clustering

↳ not good for linear search and insertion & deletion  
→ mostly used in file deletion etc.

## 2. Quadratic probing —

$$L = H(k)$$

if there is a collision at  $L$

$$L_{\text{new}} = (L + i^2) \bmod m \quad \text{where } i = 1, 2, 3, 4, \dots$$

suffers from secondary clustering.

## 3. Random probing —

$r$  = random number.

$$L = H(k)$$

if there is a collision at location  $L$

$$L_{\text{new}} = [L + (i * r)] \bmod m \quad i = 1, 2, 3, 4, \dots$$

## 4. Double Hashing —

$$L = H_1(k)$$

if collision at location  $L$

$$r = H_2(k)$$

$$L_{\text{new}} = [L + (i * r)] \bmod m \quad i = 1, 2, 3, \dots$$

## Chaining

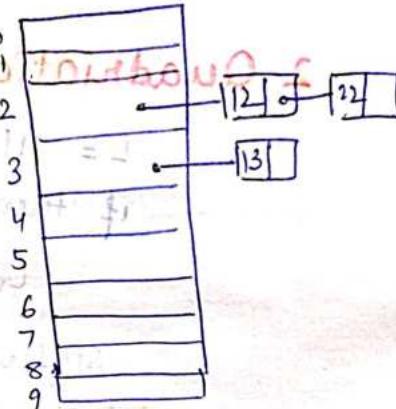
elements which have same hash value are stored in linked list at the same location.

$$H(k) = k \bmod 10$$

$$12 \bmod 10 = 2$$

$$13 \bmod 10 = 3$$

$$22 \bmod 10 = 2$$



### Open Addressing (closed Hashing)

### Closed Addressing (open Hashing)

- |   |   |
|---|---|
| <ul style="list-style-type: none"> <li>① If collision occurs at one location, then, other addresses are open to be generated.</li> <li>② Limited number of elements can be stored.</li> <li>③ Overflow possible.</li> <li>④ Deletion not possible.</li> </ul> | <ul style="list-style-type: none"> <li>① No other location is open to be generated.</li> <li>② Unlimited elements can be stored.</li> <li>③ No overflow.</li> <li>④ Deletion possible.</li> </ul> |
|---|---|

# Clustering -

## 1. Primary clustering -

long sequence of preoccupied positions  
still becoming longer at one place.

↳ → linear probing.

## 2. Secondary clustering -

long sequence of preoccupied positions  
still become longer, primarily at  
different places.

→ Quadratic probing.

# Load Factor

Average number of keys per slot

$$\text{load factor} = \frac{\text{No. of keys}}{\text{No. of slots}}$$

## Space Utilization →

No. of slots occupied as compared to total no. slots.

$$\text{Space utilization} = \frac{\text{No. of keys}}{\text{No. of slots}}$$

$$0 \leq \text{Space utilization} \leq 1$$

For open addressing,

$$0 \leq \alpha \leq 1$$

To for closed addressing,

$$0 < \alpha \geq 0$$

## Perfect Hashing

If  $n$  keys are inserted in  $n$  slots without any collision, then, it's perfect hashing

## Minimal Perfect Hashing

$n$  slots in hash table,  $m$  keys to be inserted where  $m > n$ ,

Minimize  $n$  keys inserted w/o collision after that collision occurs for  $m-n$  elements.

## Uniform Hashing

hash function that can generate all locations.