Data Structures

L-01

Data structure :- mathematical and logical model of interrelated data

- mathematical → predefined structure of organizing data.
- logical → operations that can be performed on data.

Data structure

- Linear
  Elements are arranged in linear (sequential) fashion
  - Arrays
  - Linked Lists
  - Queue
  - Stack
- Non Linear
  Elements are arranged in non linear fashion.
  - Tree
  - Graph
  - Hash Table

Classification of Data Structure

- Container
  used to store the data.
  - Array
  - Linked list
- Indexing
  data structures that are design especially for searching.
  - Binary search tree
  - AVL Tree
  - Hash Table
  - B Tree
  - B+ Tree
- Priority
  elements have different priorities.
  - Queue
  - Stack.
  - Heap.

Data Structures

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 & atmost once (only once)
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.

Algorithm

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 –

$T(n) = n+2$ linear

$T(n) = n^2+5$ quadriatic

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
⋮	⋮
Exponential	2^n
Factorial	$n!$

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

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

Complexity = $\log_2 n$

Asymptotic Notation

1. Big O :-
   - provides tightest upper bound
   - worst case complexity :(gives idea)

   $O(n)$

2. Omega Ω :-
   - provides tightest lower bound
   - best case complexity (gives idea)

   $\Omega(n)$

3. Theta θ :-
   - provides average bound.

   If $O(n)$ and $\Omega(n)$, then, $\underline{\theta(n)}$

Array

- Collection of homogeneous elements.
- All elements are stored on consecutive memory locations
- All elements can be accessed using a set of indices.

Defining array in C Language –

<datatype> <name> [<size>]

Ex:- int A[5]

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

Defining array in Data Structure
[General notation to depict array irrespective of language]

$L_B \leq U_B$

name $[L_B : U_B]$

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

Size of array = $U_B - L_B + 1$

- Default Lower Bound in array = 0.
  (if not given in ques)

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

Traversing in Array

```
for (i = LB ; i <= UB ; i++) {
    visit A[i];
}
```

Runtime complexity = $\theta(n)$

Space complexity = $\theta(1)$

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

Ans: $16 - 8 + 1 = 9$

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 –

$L_B = -5$
$U_B = 18$
$w = 4$
$B = 1000$

Location$(A[5])$ = $B + (i - L_B) * w$
= $1000 + [5 - (-5)] * 4$
= $1000 + 10 * 4$
= 1040

Finding minimum element in array –

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

```
min = A[LB]
for(i = LB+1; i<= UB; i++){
    if( A[i] < min ){
        min = A[i];
    }
}
```

Runtime complexity = $\theta(n)$

Space complexity = $\theta(1)$

★ Minimum element can also be found by Tournament method [where adjacent elements are in pairs of 2 are compared and smaller one moves to the next stage.

Tournament Method.

Runtime complexity = $\theta(n)$

Space complexity = $\theta(n)$

Finding maximum element in array –

No. of comparisons needed = $n-1$

```
max = A[LB]
for(i = LB+1; i<= UB; i++){
    if( A[i] > max){
        max = A[i];
    }
}
```

Runtime complexity = $\theta(n)$

Space complexity = $\theta(1)$

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 –

- Create 2 lists — small and Large.
- Compare adjacent elements in array in pairs of 2.
- Put the smaller of the pair in list small and larger one in list large.
- Find minimum element from 'small' list and maximum element from 'large' list.

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

Space complexity = $\theta(n)$

Operation	No. of comparisions
Preparing 2 lists	$\frac{n}{2}$
Finding minimum	$\frac{n}{2} - 1$
Finding maximum	$\frac{n}{2} - 1$

$\therefore$ 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 $= \left\lceil \frac{3n}{2} \right\rceil - 2$.

Tournament method (Divide and conquer approach)

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

Runtime complexity $= \theta(n)$

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

G.F.G. Method 1

Best algorithm to find minimum and maximum →

Runtime complexity $= \theta(n)$

Space complexity $= \theta(1)$

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

Linear search

```
int Linear_Search(A[], L_B, U_B, item){
    for( k= L_B ; k<= U_B ; k++){
        if( A[k] == item){
            return k;
        }
    }
    return L_B-1;
}
```

Runtime complexity = $O(n)$

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

Binary search

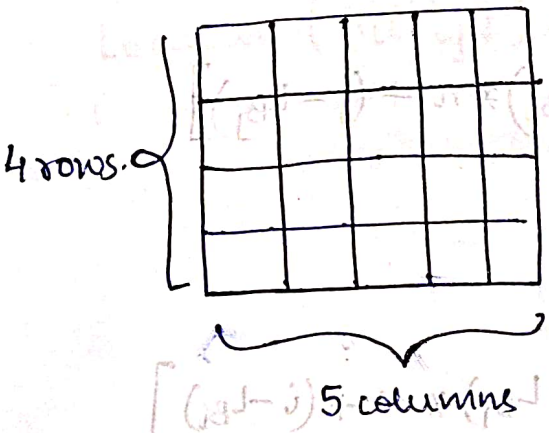
* can be applied only on sorted array

```
int Binary_Search ( A[], L_B, U_B, item) {
    Low = L_B ; High = U_B
    mid = (Low + High)/2;
    while ( Low <= High) {
        mid = Low + (High - Low)/2 ;
        if ( item< A[mid]){
            High = mid -1;
        }
        else if ( item > A[mid]){
            Low = mid+1;
        }
        else{
            return mid;
        }
    }
    return L_B-1;
```

Runtime complexity = $O(\log_2 n)$

2D Array

collection of 1D arrays.



2D Array In C-Language

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

int A[3][4]

row no. = 0, 1, 2 → $L_{Bi} = 0$, $U_{Bi} = 2$

col no. = 0, 1, 2, 3 → $L_{Bj} = 0$, $U_{Bj} = 3$

	0	1	2	3
0	A_{00}	A_{01}	A_{02}	A_{03}
1	A_{10}	A_{11}	A_{12}	A_{13}
2	A_{20}	A_{21}	A_{22}	A_{23}

2D array declaration in C Language.

<name> $[L_{Bi} : U_{Bi}]\ [L_{Bj} : U_{Bj}]$

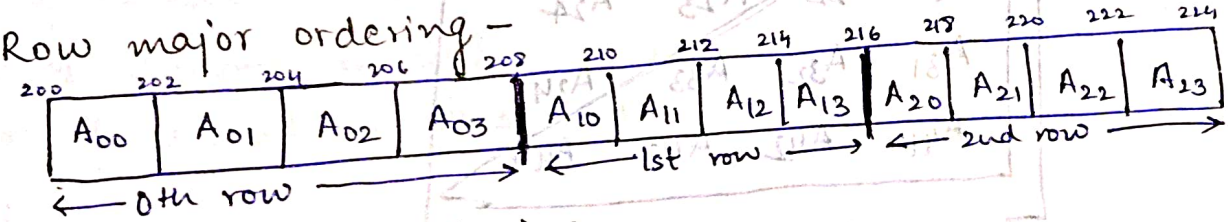
no. of rows (m) = $U_{Bi} - L_{Bi} + 1$

no. of columns (n) = $U_{Bj} - L_{Bj} + 1$

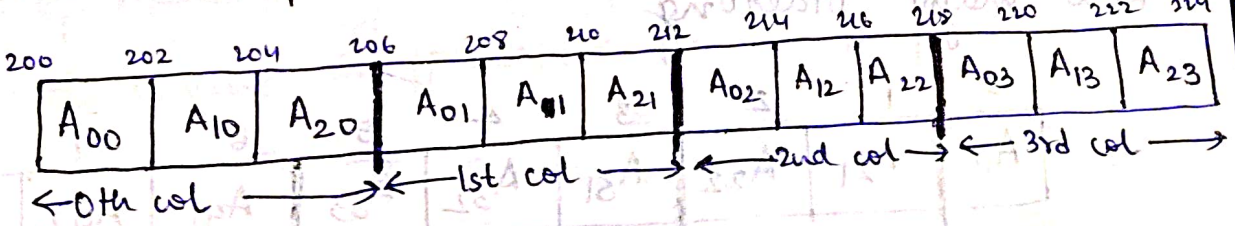
size of array = m x n.

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

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

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

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

Row major ordering

0	1	2	3	4	5	6	7	8	9
A_{11}	A_{21}	A_{22}	A_{31}	A_{32}	A_{33}	A_{41}	A_{42}	A_{43}	A_{44}

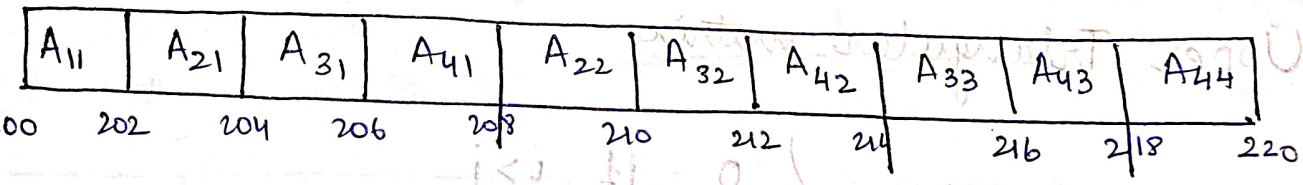
~~Location (A[i][j]) = B + w * [(i - L_{Bi}) * $\frac{i(i)}{2}$ + (j - L_{Bj})]~~

Location (A[i][j]) = B + w * [(~~L_{Bi}~~) * $\frac{i(i-1)}{2}$ + (j - ~~L_{Bj}~~)]

No. of elements in row 1 = 1
" " " " 2 = 2
No. of elements in row i-1 = i-1

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

Column major ordering



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 jth col = n-(j-1)
No. of elements in (j-1)th col = n-[j-1-1]
= n-j+2

~~Total number of elements~~ =

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

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

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

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

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

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

Upper Triangular matrix

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

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 $i-1$ = $n-(i-2)$

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

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

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

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

No. of elements before A[i][j] in ith row = $j-i$

∴ 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 from col 1 to col $j-1$

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

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

Advantages of array —

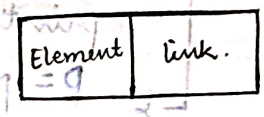
- ↦ random access of elements using index no.
- ↦ other data structures (linked list, stacks, queues, trees, graphs etc) can be implemented.
- ↦ matrices can be easily represented.

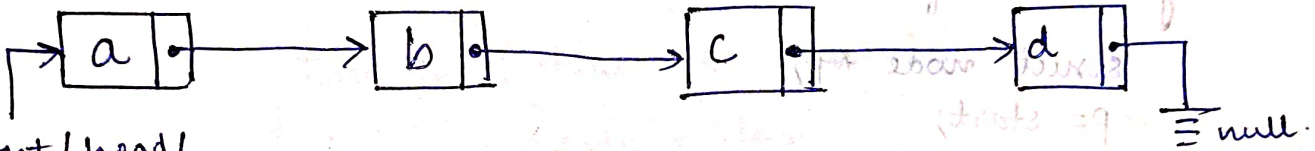
Disadvantages of arrays —

- ↦ 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].

LINKED LIST

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





start / head / first 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){
    Empty list
}
```

or

```
if (!start) {
    Empty list
}
```

Single Node Condition

```
if( start -> link == NULL){
    Single Node
}
```

Traversing a linked list

```
struct node *p;
p = start;
while (p != null) {
    print (p -> data);
    p = p -> link;
}
```

Runtime complexity = $\Theta(n)$

Finding address of last node –

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

Runtime complexity = $\Theta(n)$

* 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 = $O(n)$

Insertion in linked list –

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

→ Insertion in beginning

```
n -> data = item
n -> link = start;
start = n;
```

Runtime complexity = $\theta(1)$

→ Insertion after a given node

```
Insertion (start, loc, item) {
    n -> data = item;
    n -> link = loc -> link;
    loc -> link = n;
}
```

Runtime complexity = $\theta(1)$

→ Insertion before a given node.

```
n -> data = item;
n -> link = loc;
struct node *p = start;
while (p -> link != loc) {
    p = p -> link;
}
p -> link = n;
```

Runtime complexity = $O(n)$

Deletion in linked list

```
struct node *p = <node to be deleted>;
⋮
free(p);
```

→ Deletion of first node –

```
struct node *p = start;
start = start -> link;
free(p);
```

Runtime complexity = $\theta(1)$

→ Deletion of last node –

```
struct node *p = start;
if (start == NULL) UNDERFLOW return.
while (p -> link -> link != NULL) {
    p = p -> link;
}
struct node *q = p -> link;
p -> link = NULL;
free(q);
```

Runtime complexity = $\theta($

→ 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;
}
p->link = loc->link;
free(loc)
```

Runtime complexity = O(n)

→ Deletion after a given node (loc)

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

Runtime complexity = O(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;
free(q);
```

Runtime complexity = O(n)

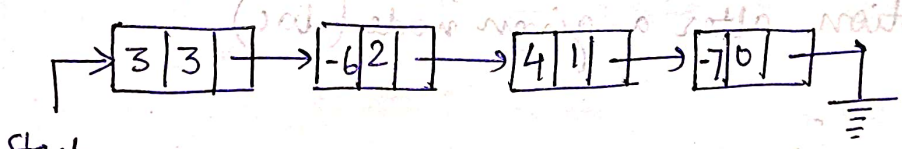
Application of linked list

⊙ To store polynomials

1) Univariate polynomial
single variable.

Ex → $3x^3 - 6x^2 + 4x - 7$

Node = | Coeff | Exponent | →

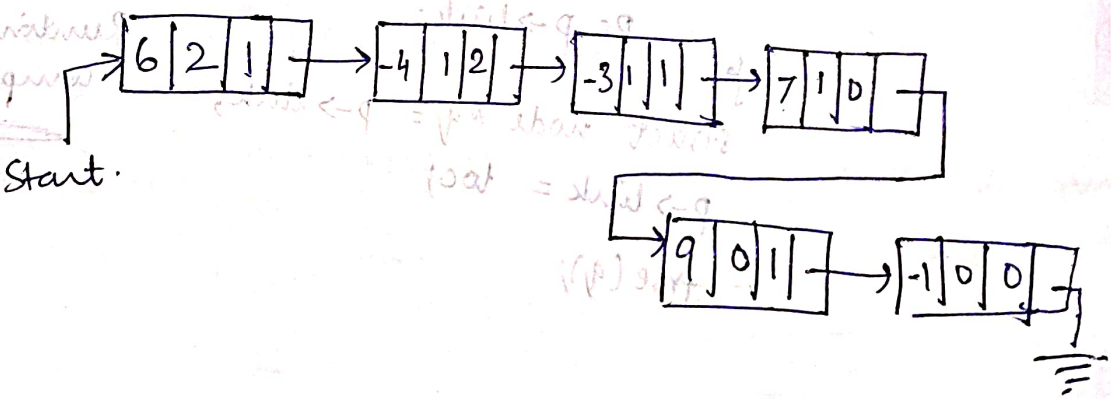


Start.

2) Bivariate polynomial.
2 variable polynomial.

Ex → $6x^2y - 4xy^2 - 3xy + 7x + 9y - 1 = 0$

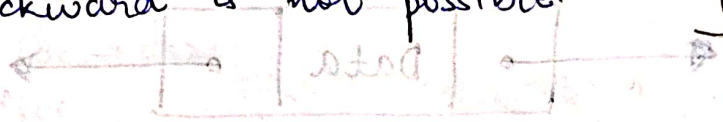
Node = | Coeff | Expx | Expy | →



Start.

Disadvantages of singly linked list

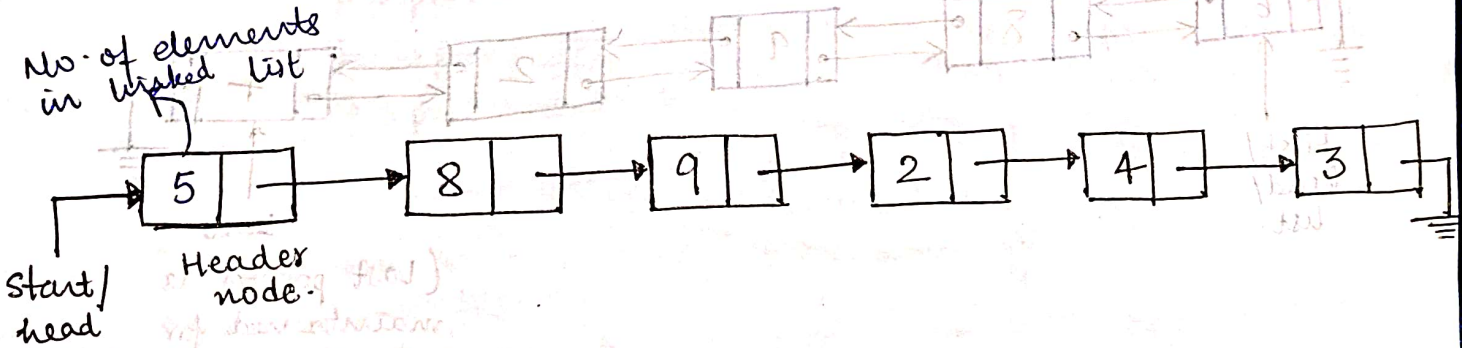
- Link part of last node is not utilized] - circular list
- The address of predecessor is not known. } Doubly linked list
- Stepping backward is not possible.



Header List

- contains special first node called Header node.
- 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

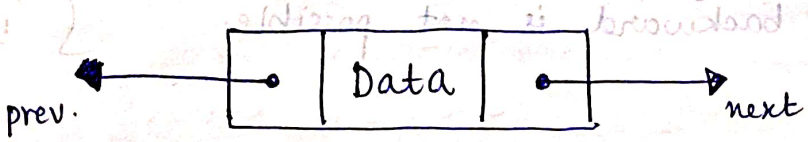
```
struct node *p = list -> link
while (p != NULL) {
    process p -> data;
    p = p -> link;
}
```

Circular

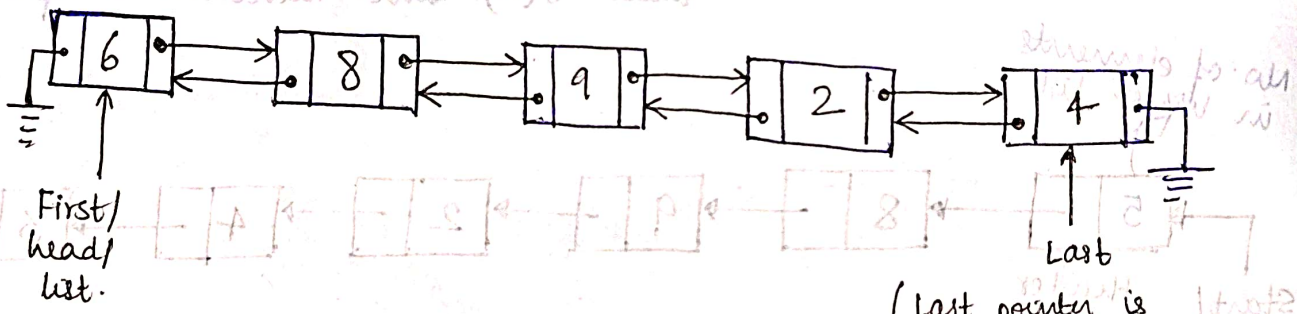
```
struct node *p = list -> link
while ( p != list) {
    process p -> data
    p = p -> link;
}
```

$\Theta(n)$

Doubly Linked List




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



(Last pointer is maintained for backward traversing)

→ Insertion at beginning

```
struct dnode *p = (struct node*) malloc(sizeof (struct node));
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);
```

Runtime complexity = O(1)

→ Deletion of a given node (loc)

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

Runtime complexity = O(1)

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

```
struct dnode *p = head;
while (p -> next != null){
    p = p -> next;
}
p -> prev -> link =
p -> prev -> next = NULL;
free(p)
```

Runtime complexity = O(n)

QUEUE

- 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).

- FIFO (First In First Out) list.

  Insertion : Enqueue
  Deletion : Dequeue

- Queue can be implemented by arrays as well as linked list.

  → 9 know!! :)

Queue

- Linear queue
- Circular queue.

	Linear queue	Circular queue
Overflow condition	if (rear == N-1)	if ((front == 0 && rear == N-1) || front == rear+1) if (front == (rear+1) mod n)
Underflow condition	if (front = rear = -1)	if (front = rear = -1)

Applications of queue –

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

Double Ended Queue

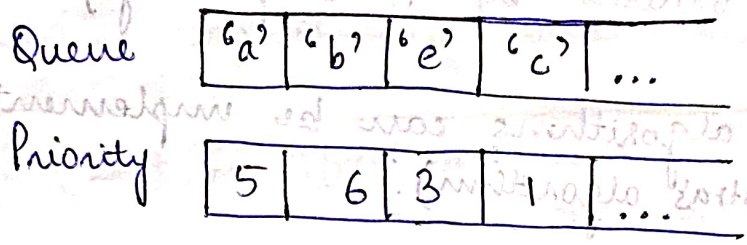
Queue in which insertion and deletion can be performed from both ends.

Types —

- Input Restricted Double Ended Queue → Input is restricted to the standard end (Rear end) <Insertion>
- Output Restricted Double Ended Queue → Output is restricted to the standard end (Front end) <Deletion>

Priority Queue

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



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.
$O(n)$

↳ Deletion –
Delete the front element (which is highest priority element)
$\theta(1)$

★ Priority queue can be used to implement stack and queue

Handle Priority at deletion time

↳ Insertion –
Insert element at rear.
$\theta(1)$

↳ Deletion –
Find the element with highest priority and delete it.
$\theta(n)$

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 (bcoz quickly retreives 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

⊙ A linear data structure in which insertion and deletion both are performed from the same end known as top of the stack.

⊙ LIFO (Last In First Out) list.

⊙ 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 poping elements to obtain different sequences.

If there are n elements in input sequence, ★

catalan number

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

No. of invalid stack permutations	$= n! - \frac{{}^{2n}C_n}{n+1}$

Example :-

Input sequence = 1 2 3

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

3 1 2 Not possible.

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

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

$= 5$

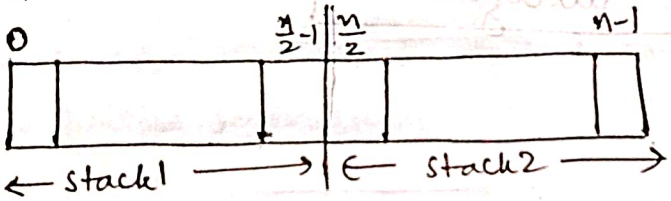
Ques The following sequence of operations is performed on a stack:

- 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

Multiple Stacks in Single Array

Method 1



$0 - \frac{n}{2} - 1$ $\quad$ $\frac{n}{2} - n - 1$

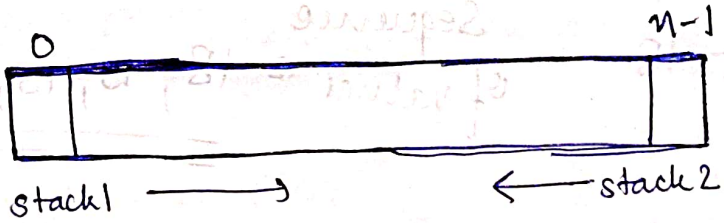
Initially,

top1 = -1 $\quad$ top2 = $\frac{n}{2} - 1$

One stack cannot use the space of other stack which leads to Improper utilization of stack space

Method2

2 stacks start from opposite ends of the array.



Initially, stacks are empty

top1 = -1

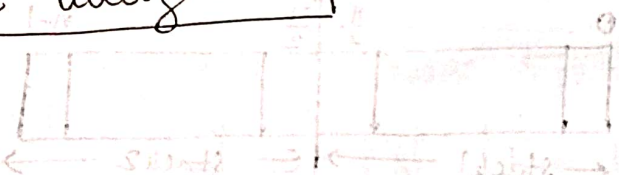
top2 = n

Overflow:

top1 == top2 - 1 or

top2 == top1 + 1

Better space utilization



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

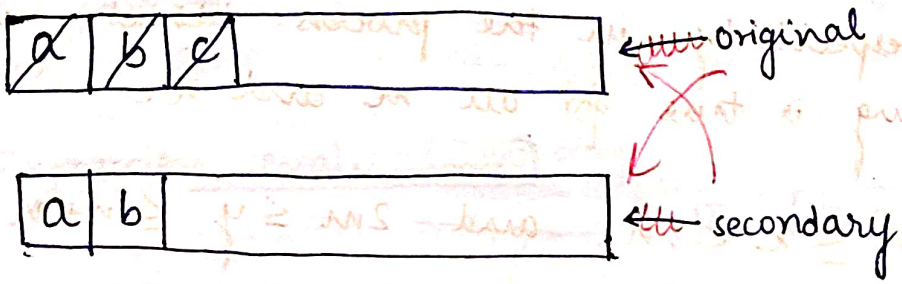
Ans — Two (2)

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.



PUSH

Insert in original queue.

POP

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

→ Queue using stacks

ENQUEUE –

- Push element to S1.

DEQUEUE –

- IF S2 is not empty, pop from S2
- IF S2 is empty, bring all the elements from S1 to S2. And pop from S2.

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 insert operations and then perform m delete operations.

INSERTION — No. of push = n

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

∴ Total no. of PUSH = $n+n = 2n$ 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 = ~~$m+m$~~ $m+n-m = n$

DELETION — No. of push = m
No. of pop = $m+m = 2m$

Total no. of PUSH = $m+n$ Total no. of POP = $2m$

Applications of stack

- Expression Evaluation
- Recursion.

1. Expression evaluation

Expression → Prefix (+ab) — Polish Notation
Expression → Infix (a+b)
Expression → Postfix (ab+) — Reverse Polish Notation

Operator Precedence

Parenthesis	()	Left
Exponent	↑ or ^	Right
Multiplication or Division	* & / & %	Left
Addition or Subtraction	+ & -	Left
Assignment	=	Right

Operators
- Binary — 2 operands
- Unary — 1 operand.

All unary operators have higher precedence than binary operators.

Infix to Prefix and Postfix conversion.

Infix	Prefix	Postfix
$2+3*5-1$	$(2+(3*5))-1$	$(2+(3*5))-1$
	$=(2+(*35))-1$	$=(2+(35*))-1$
	$=(+2*35)-1$	$=(235*+)-1$
	$=-+2*351$	$=235*+1-$

Infix	Prefix	Postfix
$7-6*3\uparrow 2/8+9$	$(7-((6*(3\uparrow 2))/8))+9$	$(7-((6*(3\uparrow 2))/8)+$
	$=(7-((6*(\uparrow 32))/8))+9$	$(7-((6*(32\uparrow))/8)+$
	$=(7-((*6\uparrow 32)/8))+9$	$(7-((632\uparrow *)/8)+$
	$=(7-(/*6\uparrow 328))+9$	$(7-(632\uparrow *8/))+$
	$=(-7/*6\uparrow 328)+9$	$(7632\uparrow *8/-)+9$
	$=+-7/*6\uparrow 3289$	$7632\uparrow *8/-9+$

$-b$	$-b$	$b-$
$\log x$	$\log x$	$x\log$
$x!$	$!x$	$x!$
$\log x!$	$\log !x$	$x!\log$

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

Prefix –

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

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

$= (/*+AB-CDF) - (/*XYZ)$

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

Postfix –

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

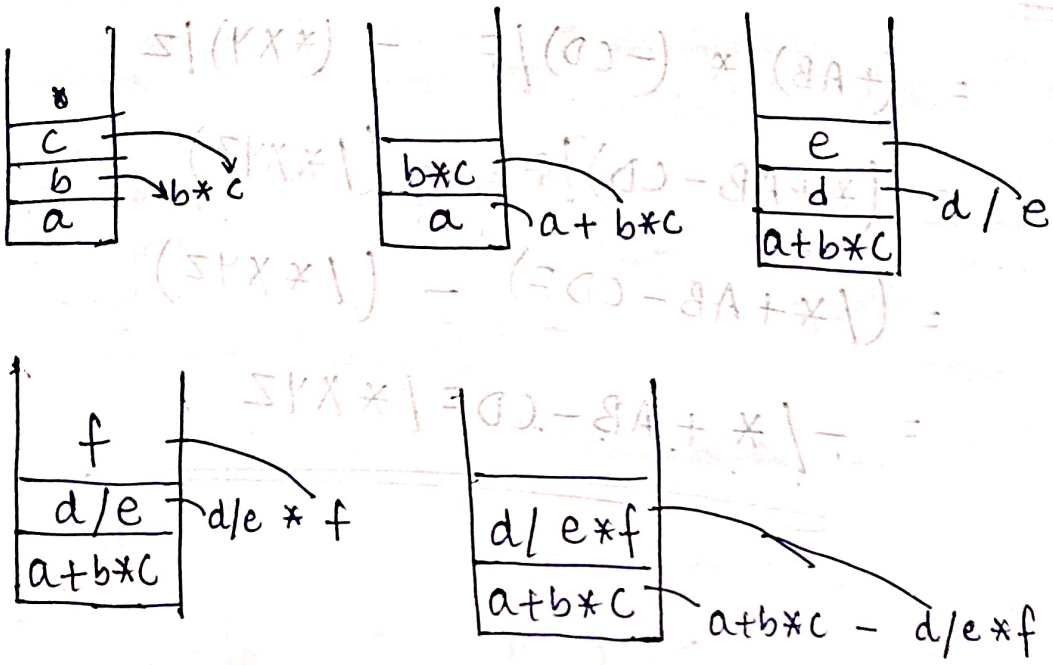
$= (AB+CD-*) / F - (XY*Z/)$

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

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

Postfix to Infix –

abc * + de / f * –



∴ Ans

Infix Expression = a+b*c – d/e*f

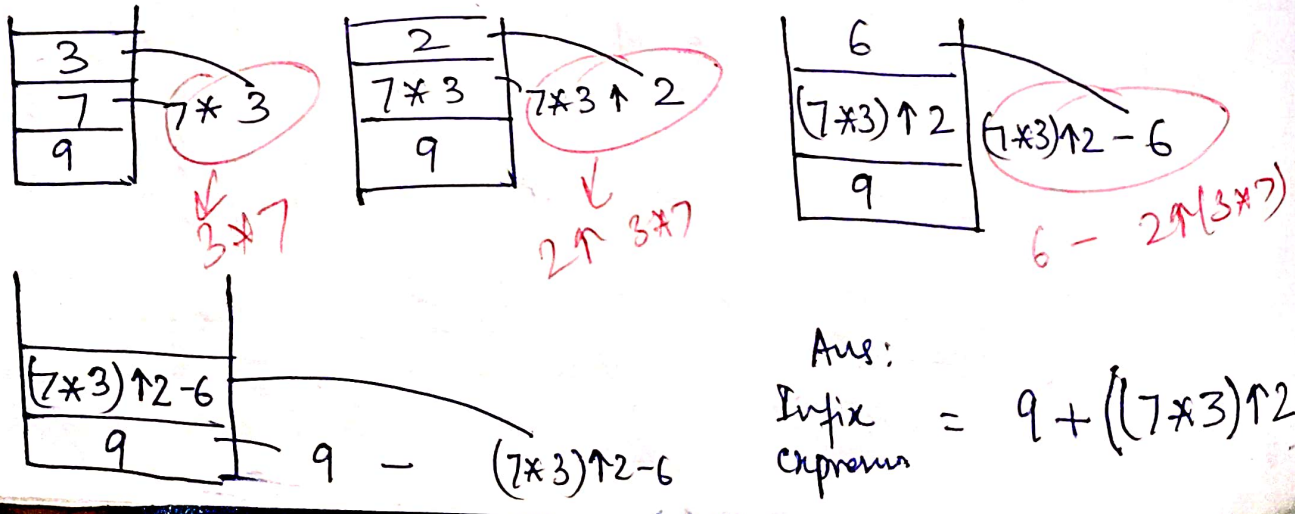
Prefix to Infix –

Reverse the prefix and solve by postfix method.

+ – 6 ↑ 2 * 3 7 9

op1 operand op2

= 9 7 3 * 2 ↑ 6 – +



Ans:

Infix expression = 9 + ((7*3)↑2–6

convert the expression to prefix and postfix

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

Prefix conversion —

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

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

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

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

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

Postfix expression —

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

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

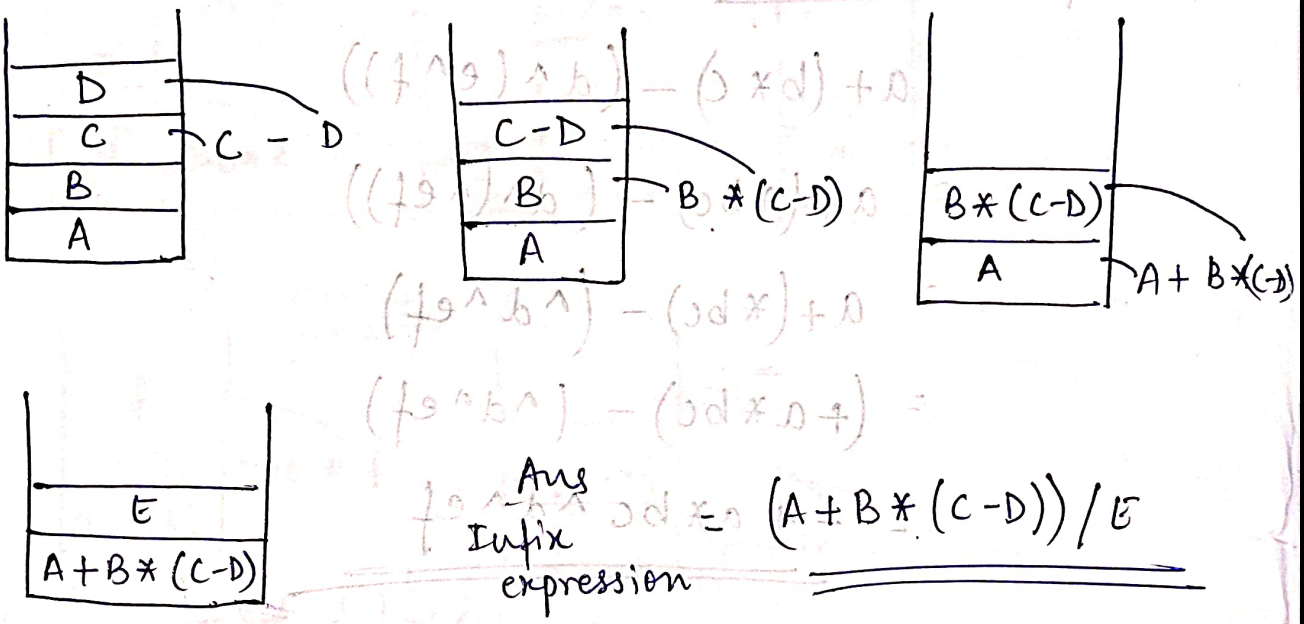
$= a+(bc*)-(def\wedge\wedge)$

$= (abc*+)-(def\wedge\wedge)$

$= abc*+def\wedge\wedge-$

Convert to infix —

A B C D - * + E /

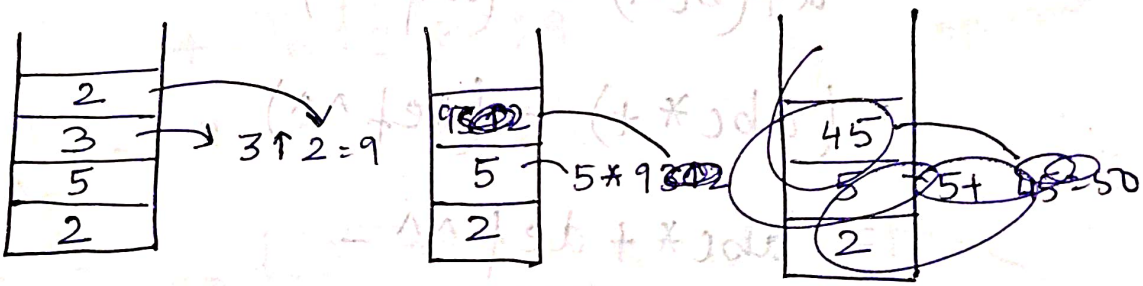


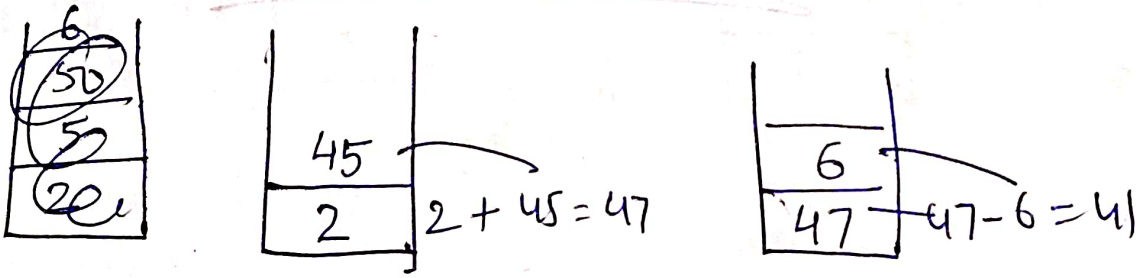
Ans

Infix expression = (A + B * (C - D)) / E

Postfix Evaluation

2 5 3 2 ↑ * + 6 -

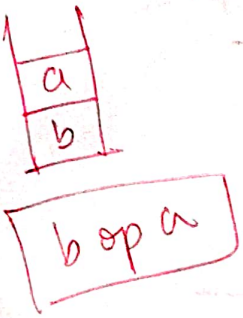




Ans = 41

Algorithm:

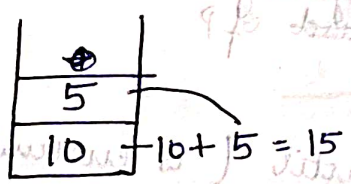
- Add parenthesis) at the end of P.
- Scan P from left to right until) is encountered.
  - ↳ If operand is encountered, PUSH it onto stack.
  - ↳ If operator is encountered,
    i. Pop 2 elements from stack — a & b
    ii. Evaluate b op a and ~~put~~ ~~re~~ push result onto stack.

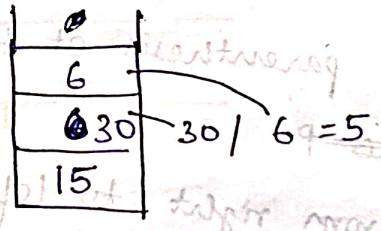


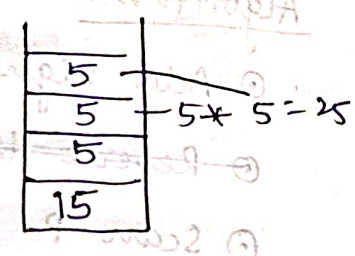
- Result = top of stack.

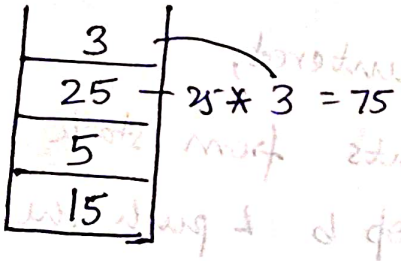
Ques Evaluate postfix expression —

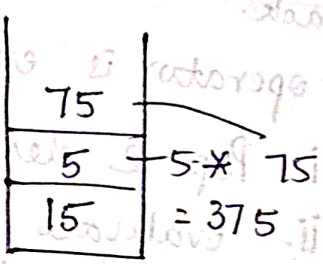
10 5 + 30 6 / 5 5 * 3 * * + 8 −

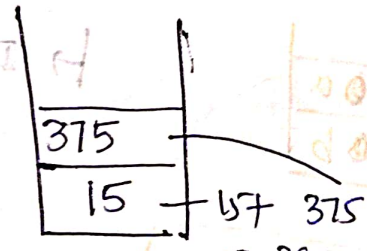


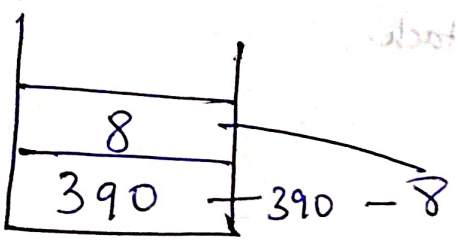










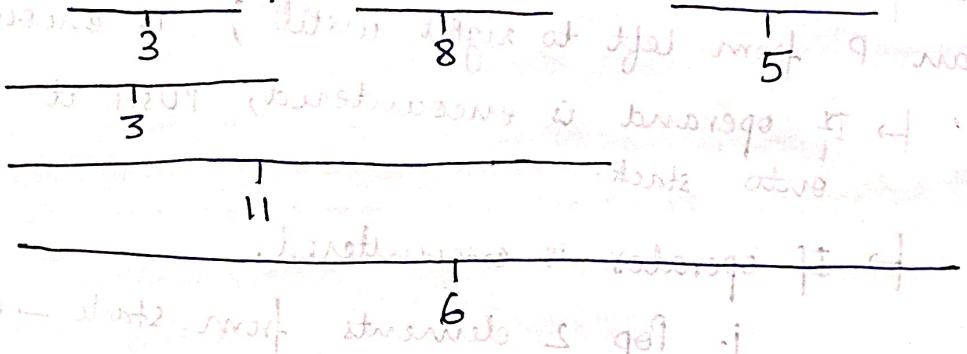


= 382

Ans = 382

Ques

9 3 1 ^ / 4 2 * + 5 1 * -



Ans = 6

Prefix Evaluation

Algorithm:

⊙ Add right parenthesis at the ~~end~~ beginning of P

~~⊙ Reverse the P.~~

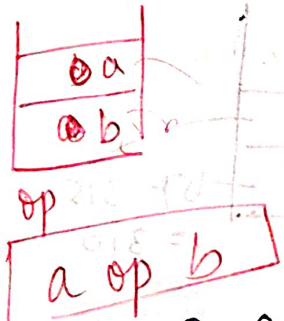
⊙ Scan P from right to left until (is encountered:

↳ 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.



⊙ Set result = top of stack.

Postfix evaluation — b op a

Prefix evaluation — a op b

Ques

$- + 5 / * 6 2 3 4$

12

4

9

5

Ans = 5

Ques

$/ 3 * 9 + 4 + 5 3$

8

12

108

$3/108 = \frac{1}{36}$

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 —

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

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

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

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

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

8 5 → 40 → 10

18 → 6

10, 4 → 4

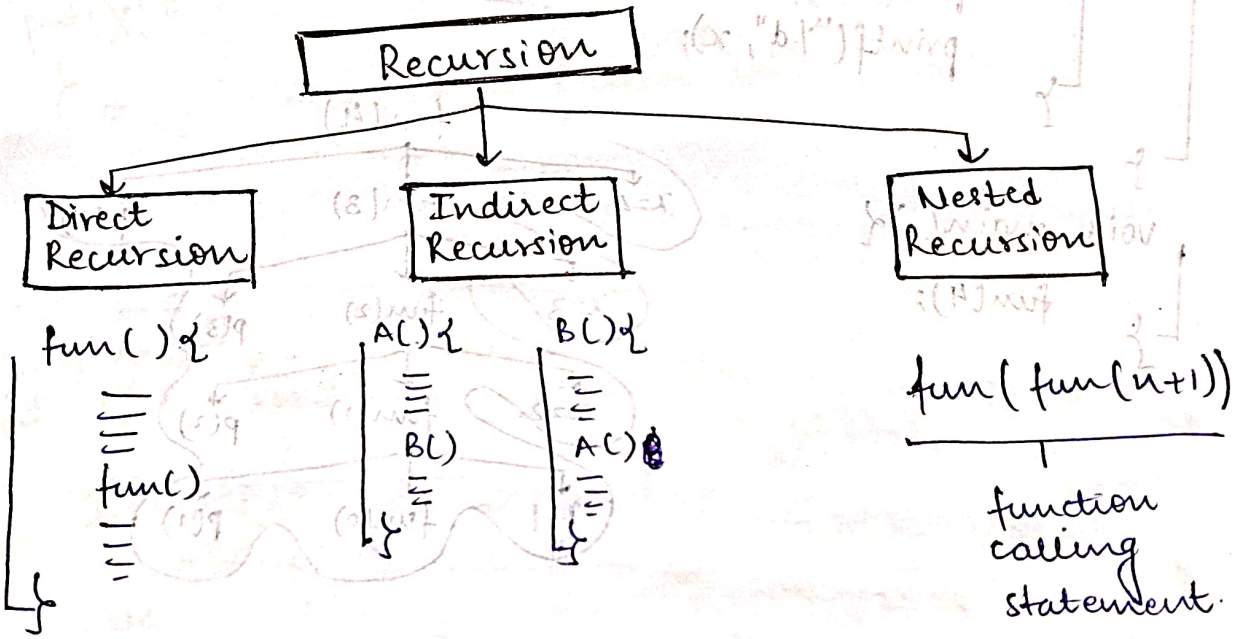
No. of PUSH operations = 2+1+2+1+1+1+1+2+1+1+2
= 15

No. of POP operations = 14 (No. of operators x2)

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 close



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.

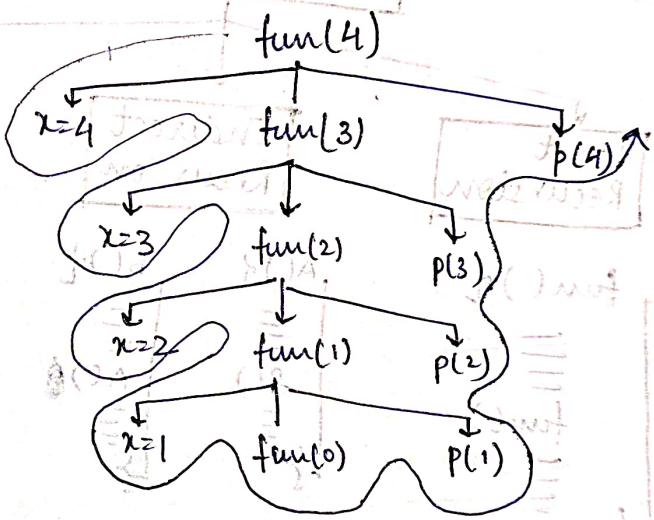
Activation Record contains
- function parameters
- local variable.

Ques

```
void fun (int x) {
    if (x>0) {
        fun(x-1);
        printf("%d", x);
    }
}
void main() {
    fun(4);
}
```

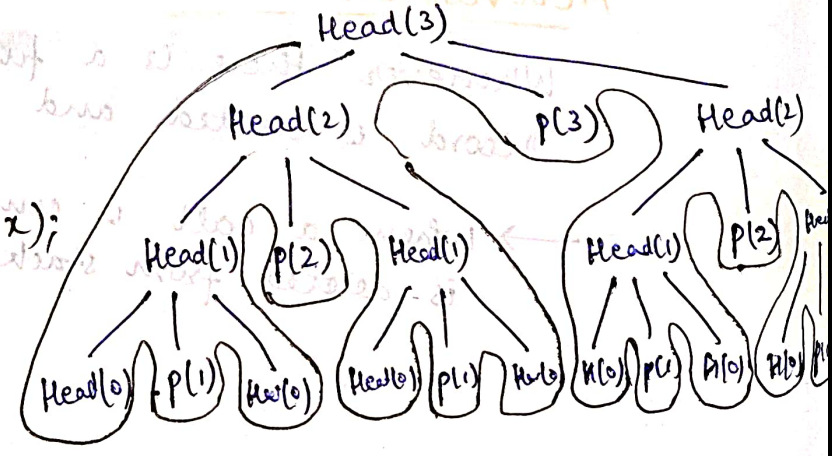
O/P

1 2 3 4



Ques

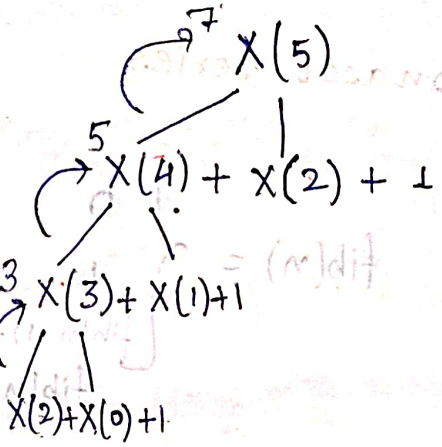
```
void Head (int x) {
    if (x>0) {
        Head(x-1);
        printf("%d", x);
        Head(x-1);
    }
}
void main() {
    Head(3);
}
```



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
}
main(){
    print(X(5));
}
```



O/P → 7

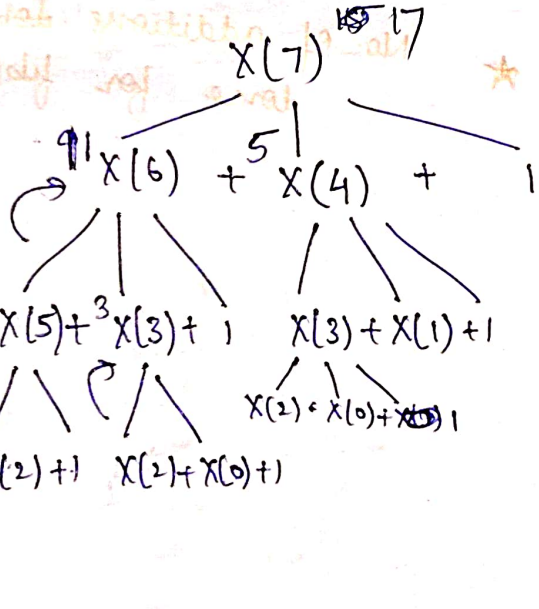
No. of function calls = 7

Ques

```
int X(int N){
    if(N<3)
        return 1
    else
        return (X(N-1) + X(N-3) + 1)
}
```

Return value of X(X(5))

X(5) = 7

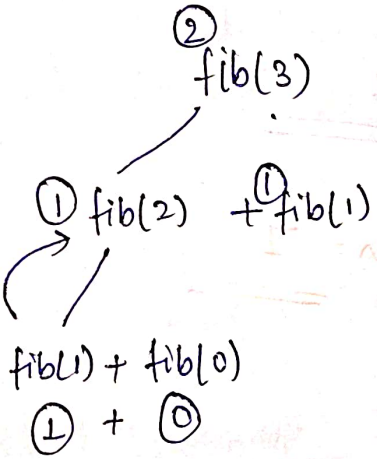


No. of function calls = 17 + 7 = 24

Ans — 17

Fibonacci series

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



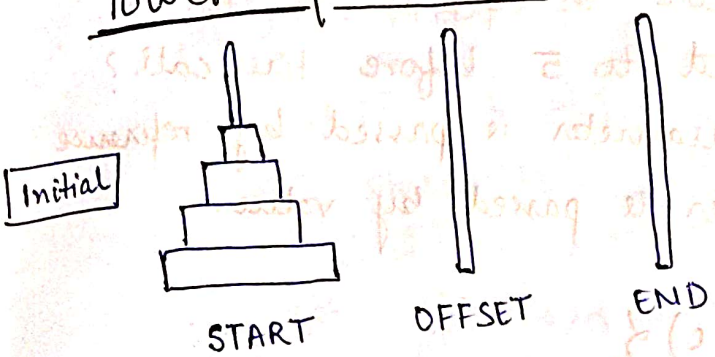
No. of additions for fib(3) = 2

No. of function invocations / calls = 5

★ No. of function calls / invocations for fib(n) = $2 * fib(n+1) - 1$

★ No. of additions ~~done~~ to be done for fib(n) = $fib(n+1) - 1$

Tower of Hanoi



Transfer discs 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

★

No. of disk movements = $2^n - 1$

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

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.

```
int f( int &x, int c) {
    c = c-1
    if(c == 0) return 1
    x = x+1;
    return f(x, c) * x;
}
```

fun(5, 5) x = 5 6 7 8 9, c = 5 4

↓

fun(6, 4) * (x) ← last value of x 729 * 9 = 6561

↓

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

↓

fun(8, 2) * x 9 * 9 = 81

↓

fun(9, 1) * x

1 * 9

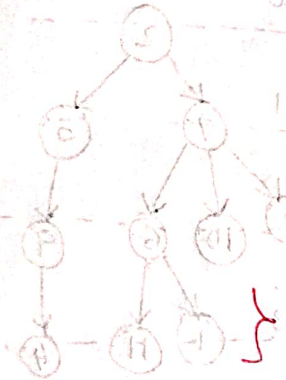
↓

Ans = 6561

GATE 2007

Consider the following C function:

```
int f(int n){
static int r = 0;
if(n<=0) return 1;
if(n>3){
    r = n;
    return f(n-2)+2;
}
return f(n-1)+r
}
```

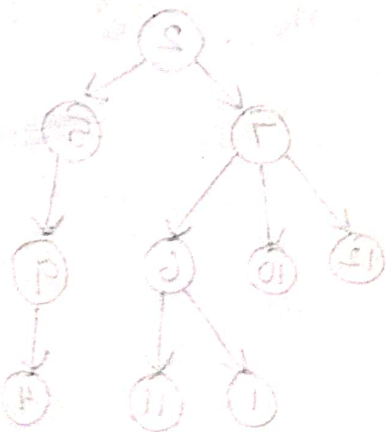


Ans — 18

f(5)

r = ~~0~~ 5

f(5)
|
f(3) + 2 → 16+2 = 18
|
f(2) + 5 → 11+5 = 16
|
f(1) + 5 → 6+5 = 11
|
f(0) + 5 = 1+5 = 6



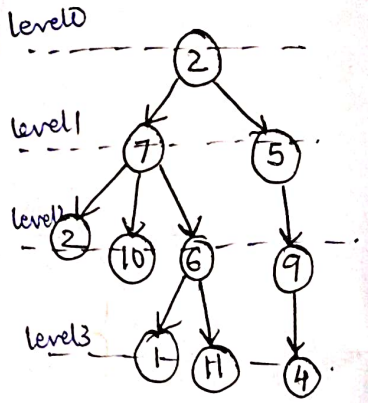
L-22

TREE

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

Tree Terminology –

- Root
- Child
- Parent
- Sibling
- Subtree
- Internal Node
- External Node (leaf node)



* Depth of a node = its level number

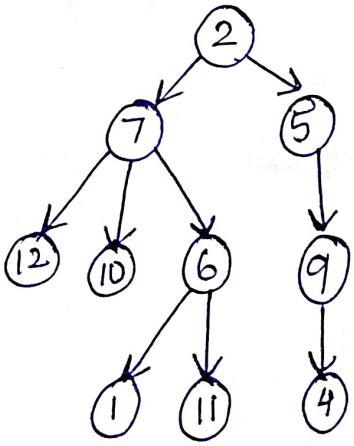
Handshaking Lemma –

Total Number of Nodes (N) = Total no. of Internal nodes (I) + Total no. of External nodes (L)

$$N = I + L$$

Parenthesis Method of Representation

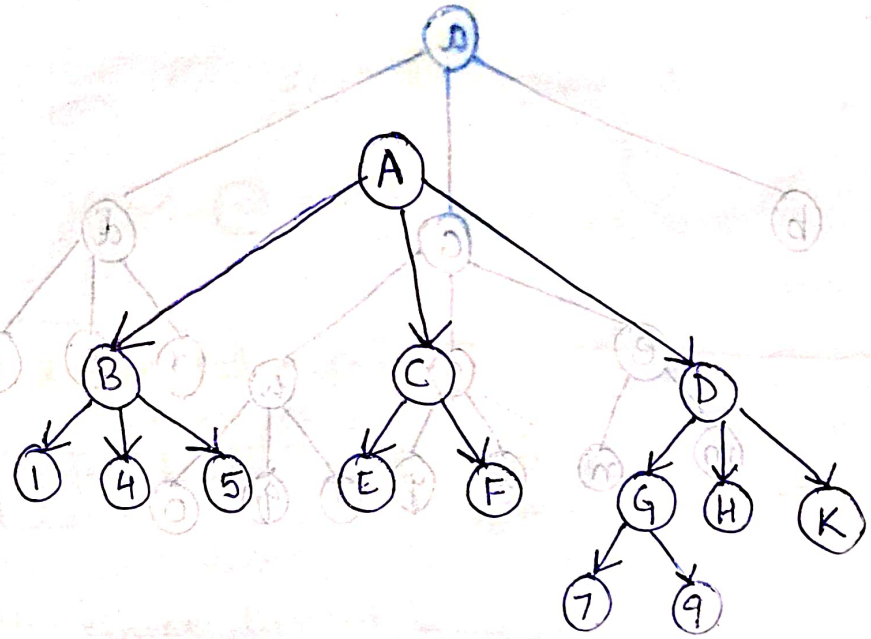
Node (child1, child2 .....)



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

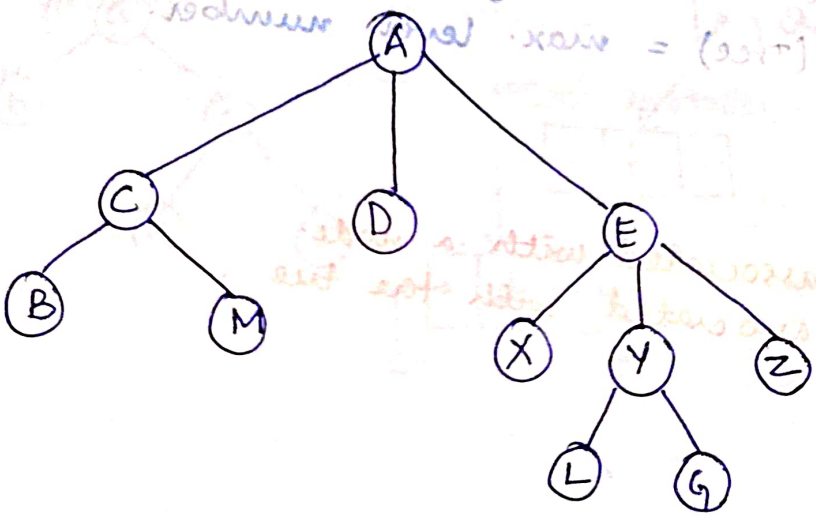
Ques Construct the tree from the following parenthesis representation

A(B(1,4,5), C(E, F), D(G (7,9), H, K))



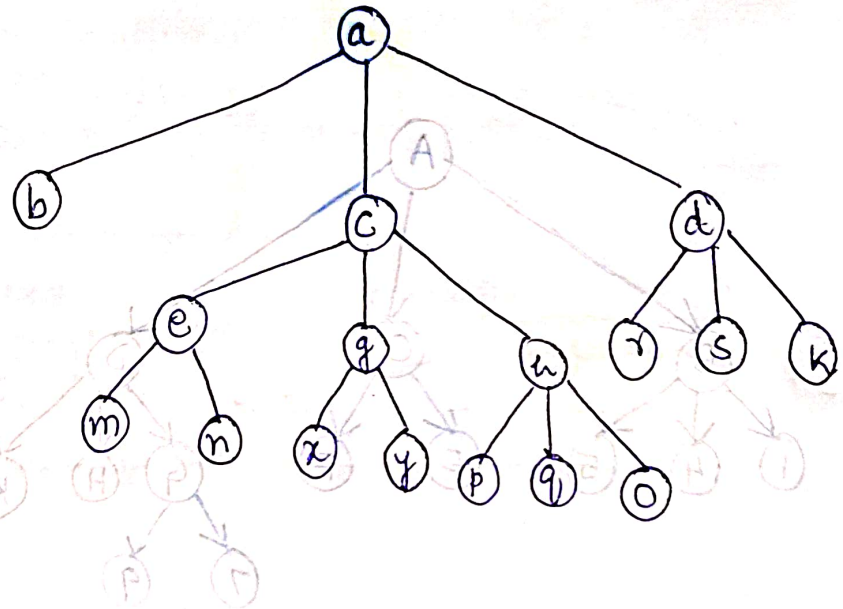
Ques

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



Ques

a(b, c(e(m,n), g(x,y), h(p,q,o)), d(r,s,k))



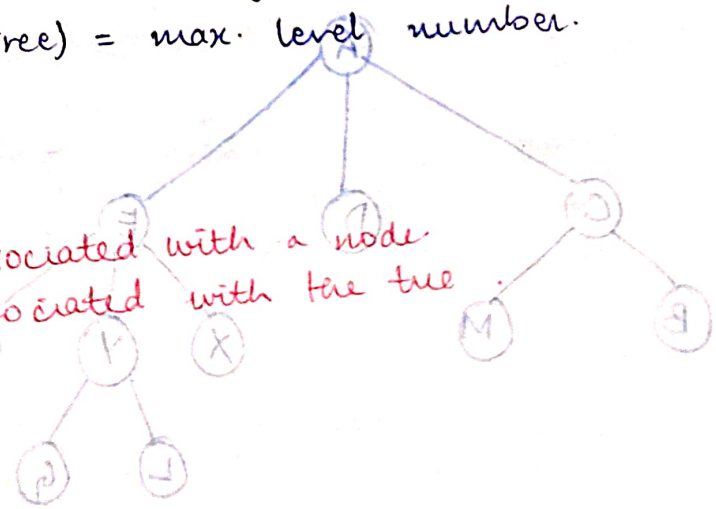
Height of Tree

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

H(Tree) of empty tree = -1

H(Tree) with single node = 0

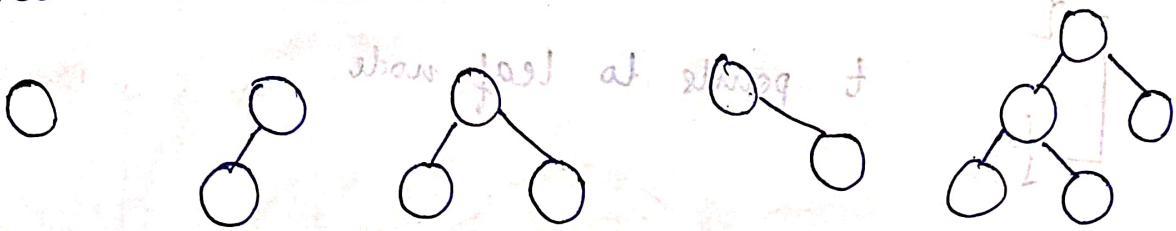
H(Tree) = max. level number.



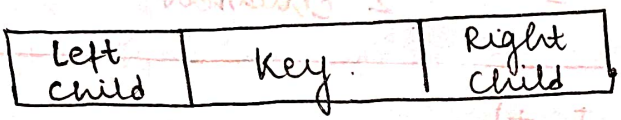
* Depth is associated with a node.
  Height is associated with the tree.

Binary Tree

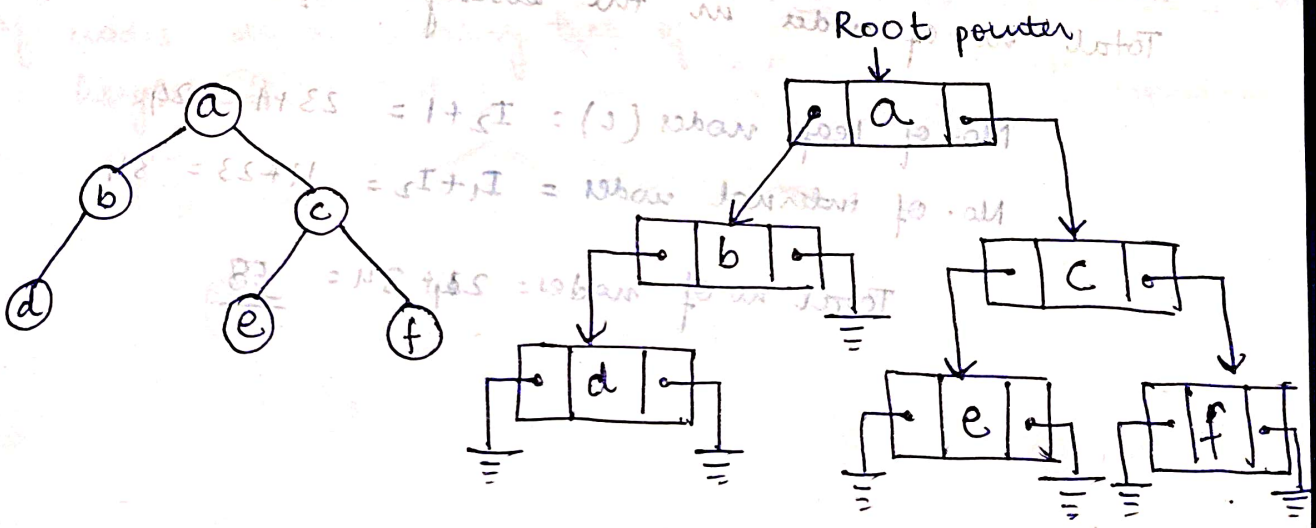
A tree in which each node has maximum of 2 children.



Linked Representation

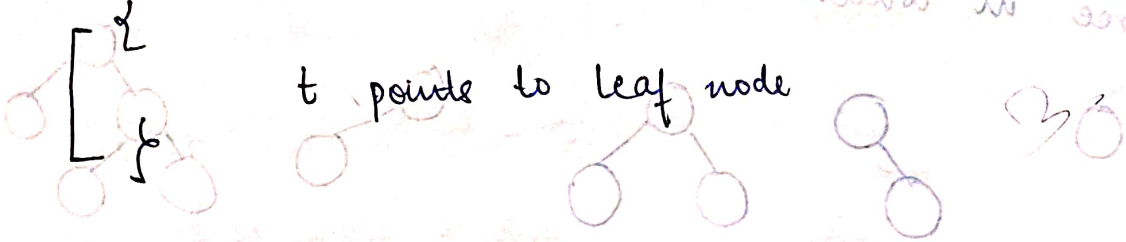



```
struct BTnode {
    char key;
    struct BTnode *Lchild;
    struct BTnode *Rchild;
};
```



How to detect a Leaf Node?

```
if ( t -> Lchild == NULL  && t -> Rchild == NULL)
{
    t points to leaf node
}
```



Number of leaf nodes	=	No. of nodes with 2 children	+ 1

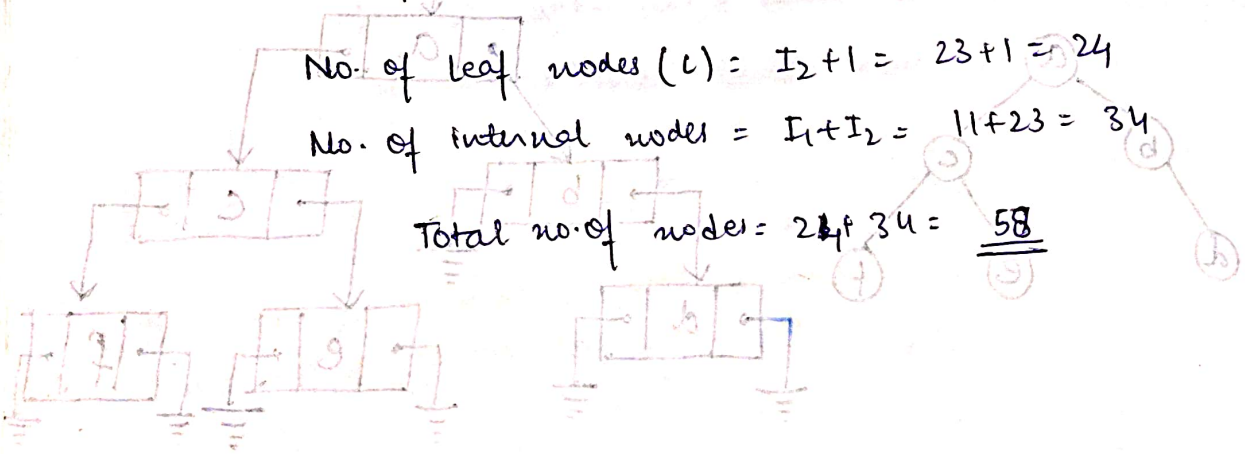
$L = I_2 + 1$

Ques In a binary tree, 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?

No. of leaf nodes (L) = $I_2 + 1 = 23 + 1 = 24$

No. of internal nodes = $I_1 + I_2 = 11 + 23 = 34$

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

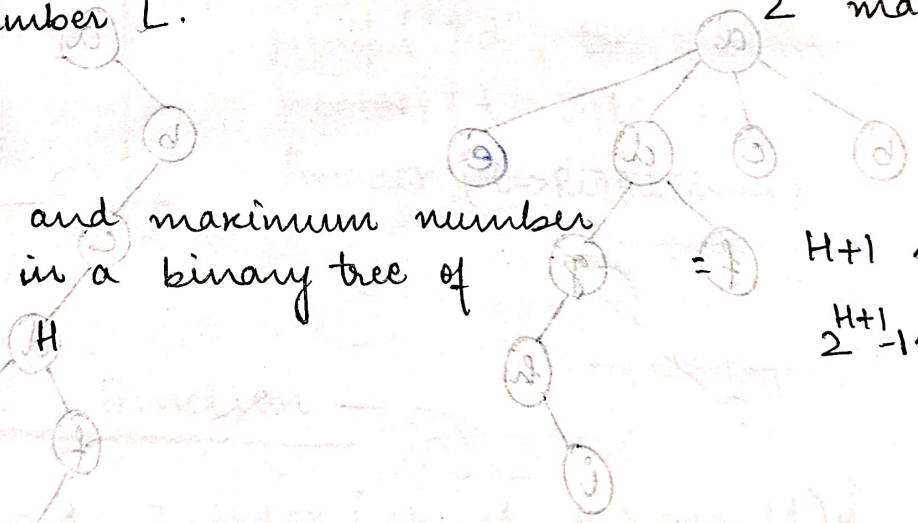


No. of binary trees that can be constructed using n unlabelled nodes $= \frac{2nC_n}{n+1}$, Catalan no.

No. of binary trees that can be constructed with n distinct keys $= \frac{2nC_n}{n+1} * n!$

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

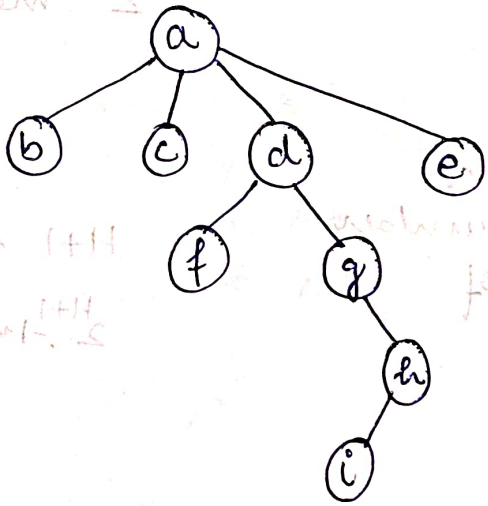
Conversion of General Tree into Binary Tree

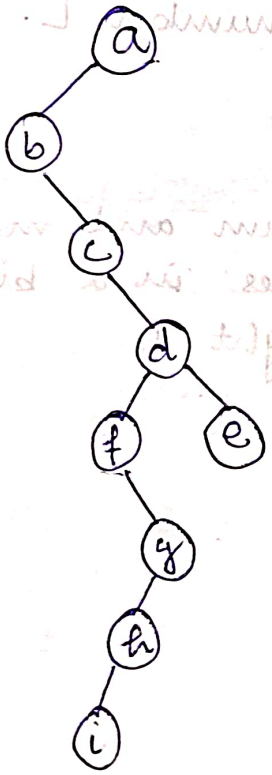
Binary Tree Representation of general tree.

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.





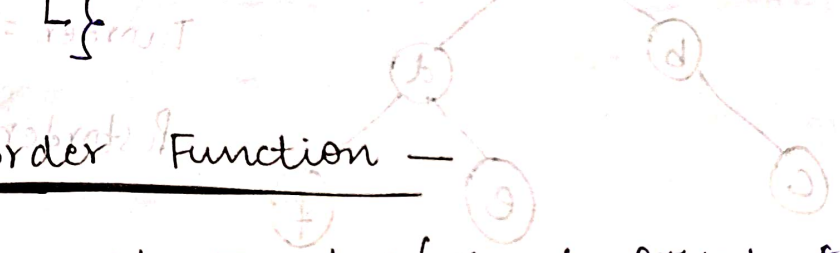
Tree Traversal

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

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);
    }
}
```



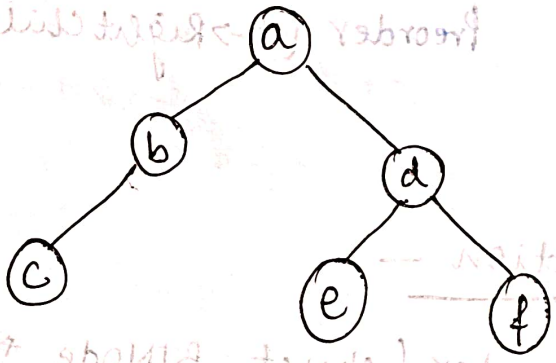
Inorder Function —

```
void Inorder (struct BTNode *t){
    if (t){
        Inorder (t->leftChild);
        printf ("%d", t->data);
        Inorder (t->RightChild);
    }
}
```

Postorder Function

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

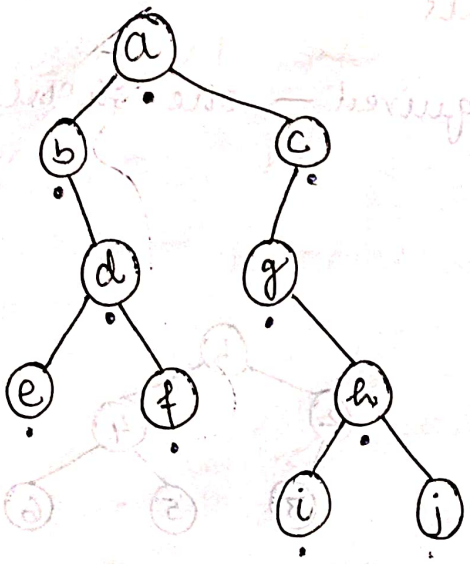
Examples



Preorder = a b c d e f

Inorder = c b a e d f

Postorder = c b e f d a

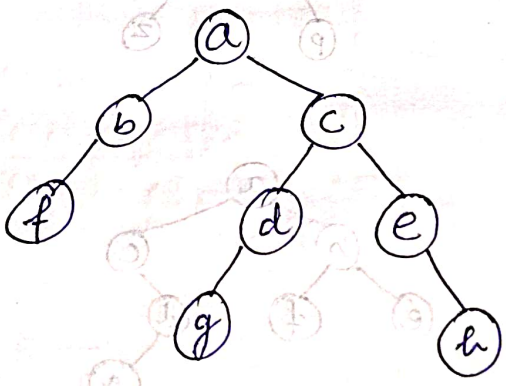


Preorder = a b d e f c g h i j

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

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

Level Order Traversal



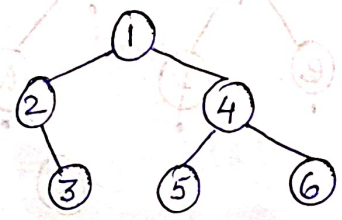
level order Traversal = a b c f d e g h

Constructing Tree using Traversals

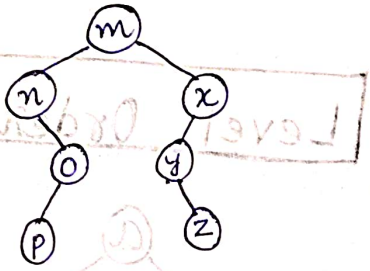
Minimum 2 traversals are required — one should be inorder.

Example-

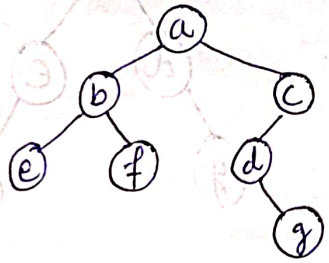
Preorder: 1 2 3 4 5 6
Inorder: 2 3 1 5 4 6



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

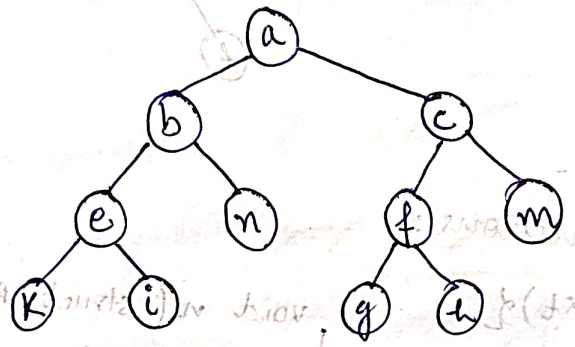


Reverse of converse postorder = Preorder traversal
Reverse of converse inorder = Inorder traversal

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

Preorder: a b e k i n c f g h m

Postorder: k i e n b g h f m c a

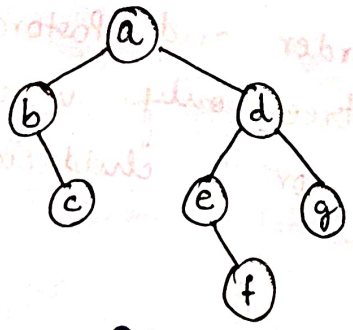


Height of Tree

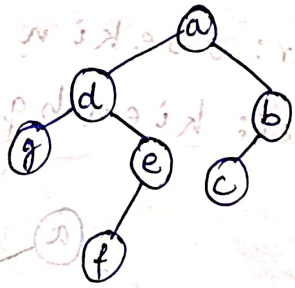
```
int Height (struct BTNode *t){
    if(t==NULL){
        return -1;
    }
    else{
        return max ( Height(t->Lchild),
        Height (t -> Rchild)) +1;
    }
}
```

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?



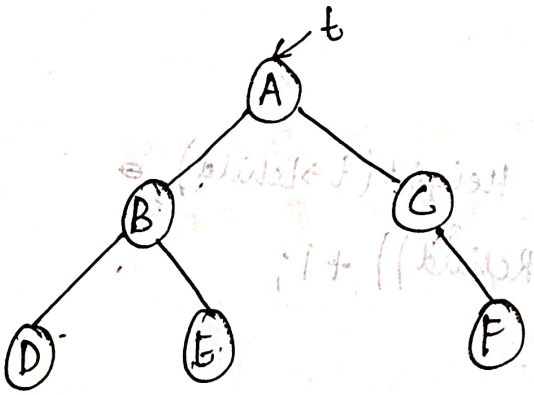
Ques

Consider the following functions:

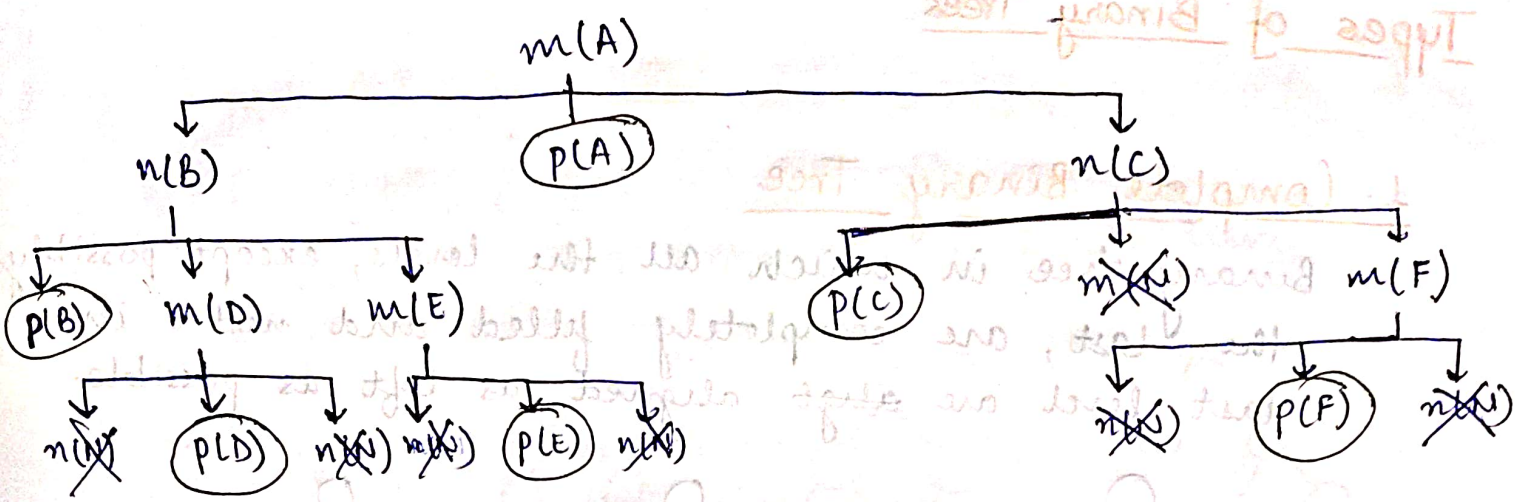
```
void m(struct BTNode *t){
    if(t){
        m(t->leftchild);
        printf("%c", t->data);
        m(t->Rightchild);
    }
}
```

```
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

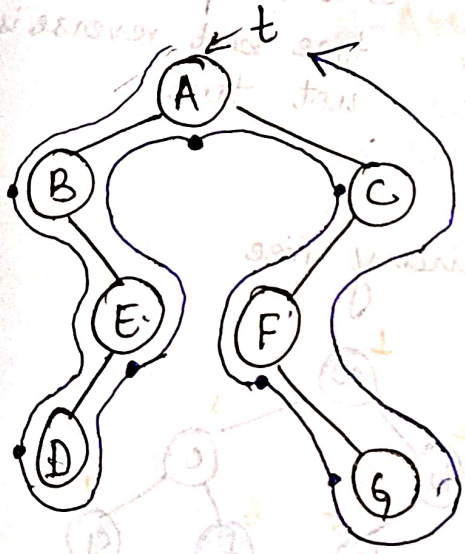


Ques

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->Right child);
    }
}
```



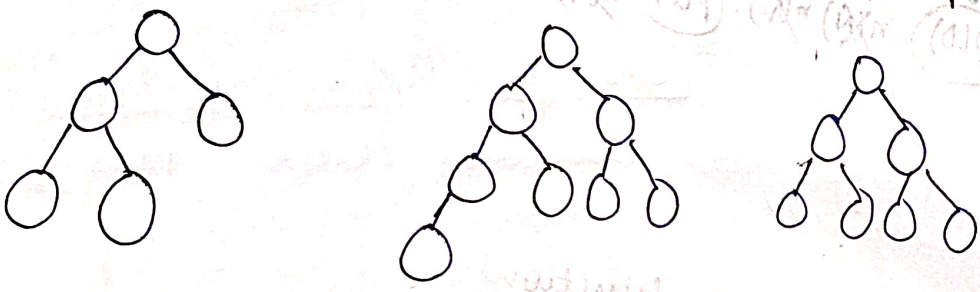
m(t)

B D E A C F G

Types of Binary Trees

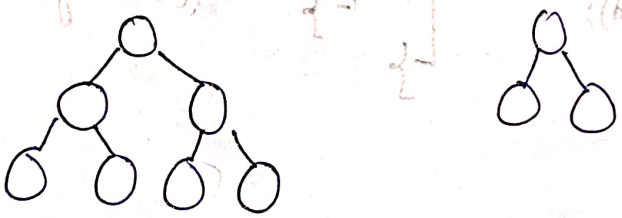
1. Complete Binary Tree

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



2. Strict Complete Binary Tree

Binary tree in which every level is completely filled.



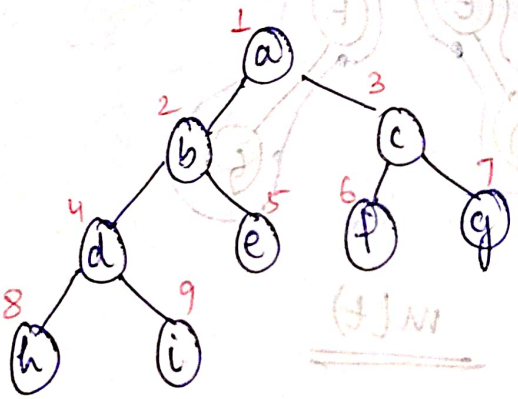
Every strict complete binary tree is a complete binary tree but reverse is not true.

Array Representation of complete Binary Tree

Root at index 1

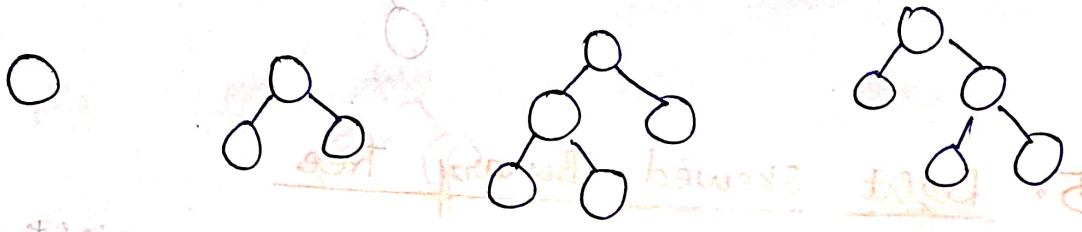
Left child = 2i

Right child = 2i+1



3. Full Binary Tree

- also called proper binary tree.
- 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.

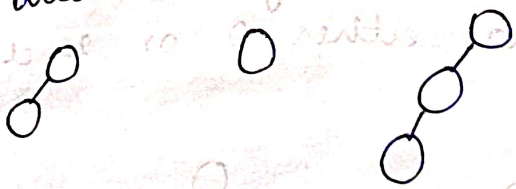
If I = no. of internal nodes

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

$$N = kI + 1$$

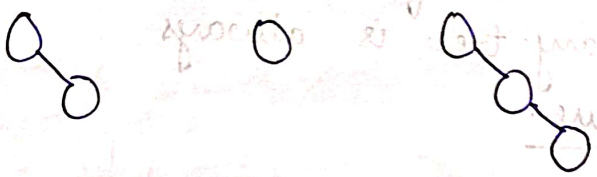
4. Left Skewed Binary Tree

Every internal node has only left child.



5. Right Skewed Binary Tree

Every internal node has only right child.



6. Expression Tree

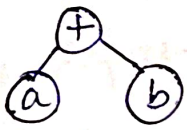
used to represent expression

Operator = root

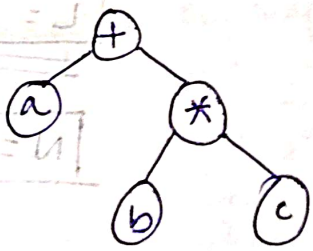
Left operand = left child

Right operand = Right child

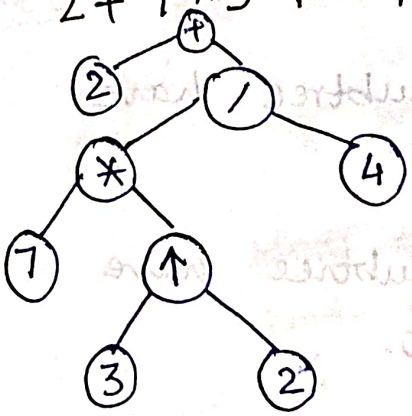
a+b



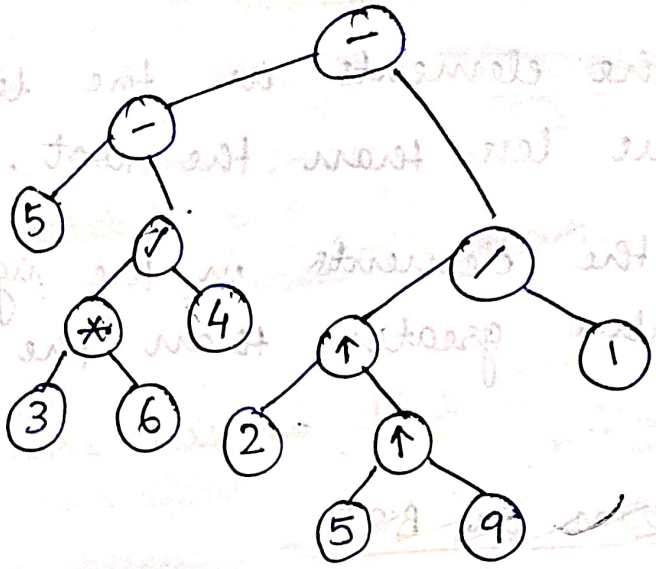
a+b*c



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

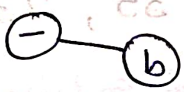


B $5-3*6/4-2\uparrow 5\uparrow 9/1$

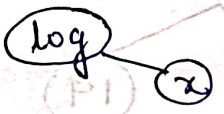


Draw expression tree for:

$-b$

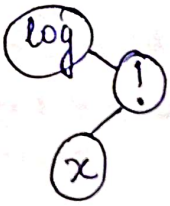


$\log x$



$x!$

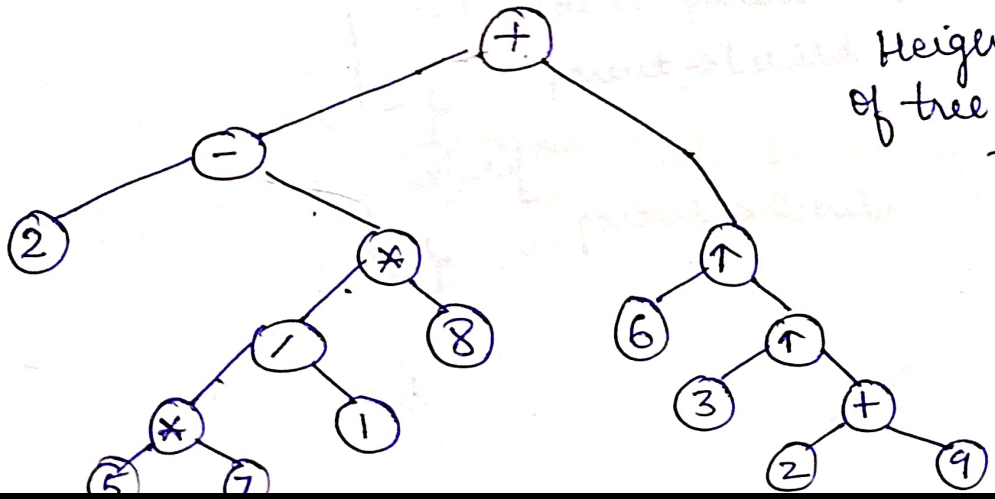
$\log x!$



Ques

Draw expression tree for

$2-5*7/1*8+6\uparrow 3\uparrow(2+9)$



Height of tree = 5

Binary Search Tree

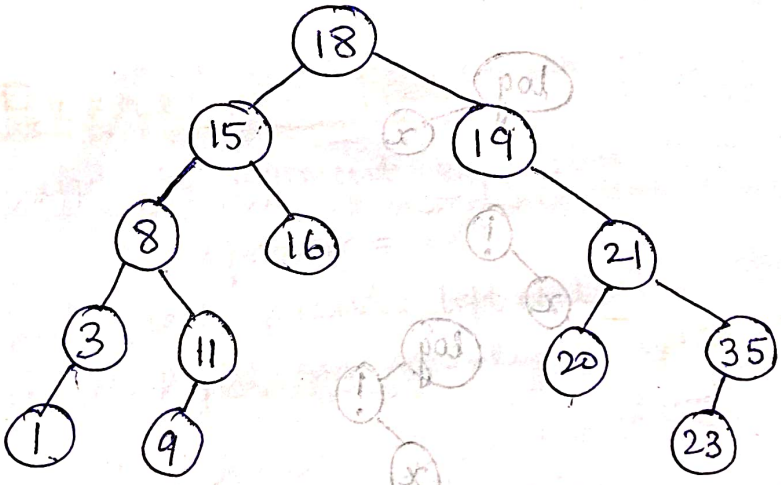
All the elements in the left subtree have value less than the root.

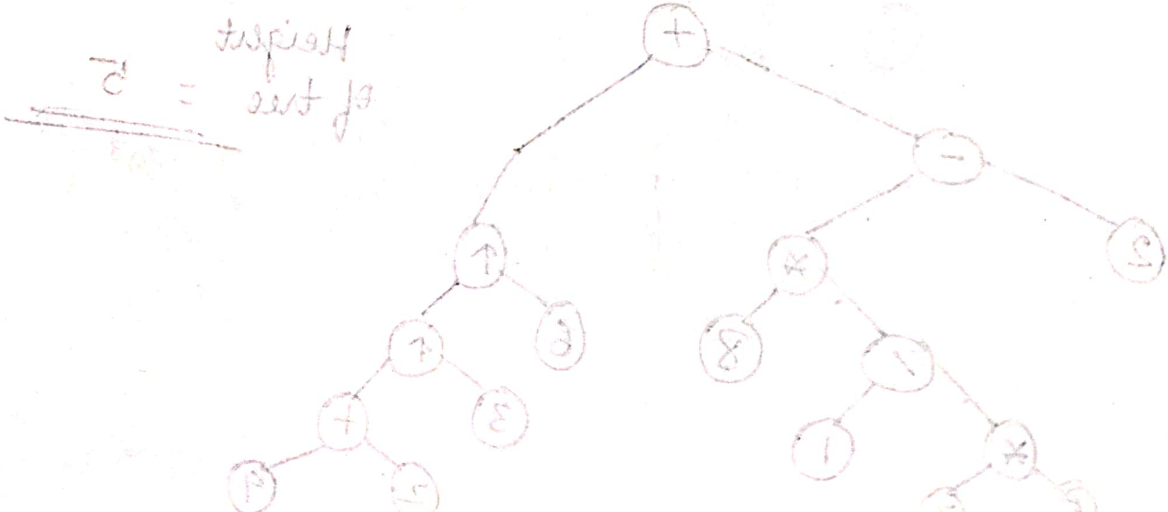
All the elements in the right subtree have value greater than the root.

Insertion in BST

Construct BST using keys

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





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

```
parent
if( node == parent -> Lchild ){
    parent -> Lchild = NULL
}
else{ parent -> Rchild = NULL
}
free(node)
```

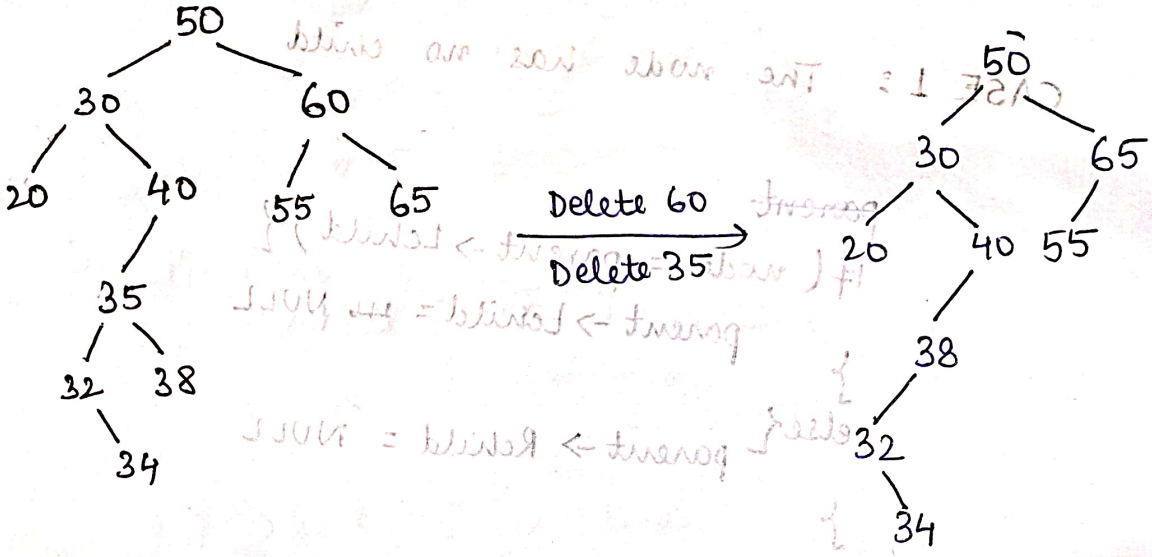
CASE 2: The node has one child

```
if( node -> Lchild != null ){
    child = node -> Lchild;
}
else{
    child = node -> Rchild;
}
parent
if( node == parent -> Lchild ){
    parent -> Lchild = child;
}
else{
    parent -> Rchild = 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)
Insertion	O(H)	$O(\log_2 n)$	O(n)
Deletion	O(H)	$O(\log_2 n)$	O(n)

★ Inorder traversal of BST gives sorted sequence in ascending order.

Ques

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

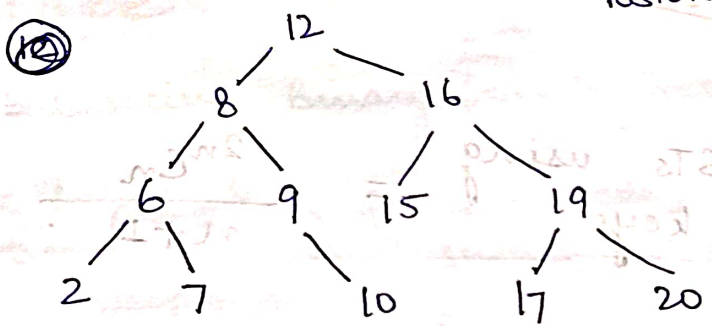
5 9 10 12 15 16 19 (ascending order)

~~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?

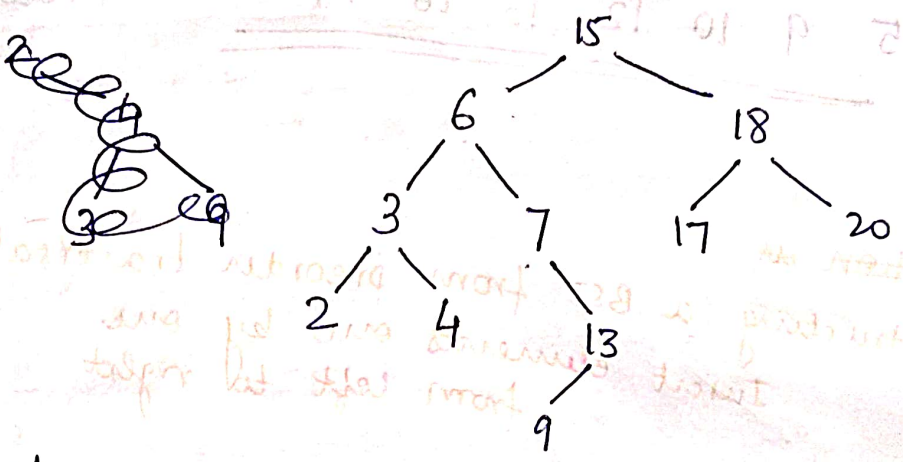
~~postorder traversal~~



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 = $\frac{2nC_n}{n+1}$.

	n unlabelled nodes	n distinct keys.
Binary Tree	$\frac{2nC_n}{n+1}$	$\frac{2nC_n}{n+1} * n!$
Binary Search Tree	—	$\frac{2nC_n}{n+1}$

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.

No. of elements less than 60 = 4

No. of " greater " " = 4

$\therefore$ No. of orders possible = $\frac{7!}{4!\,4!} = \frac{7\times6\times5\times4\times3\times2\times1}{4\times3\times2\times1}$

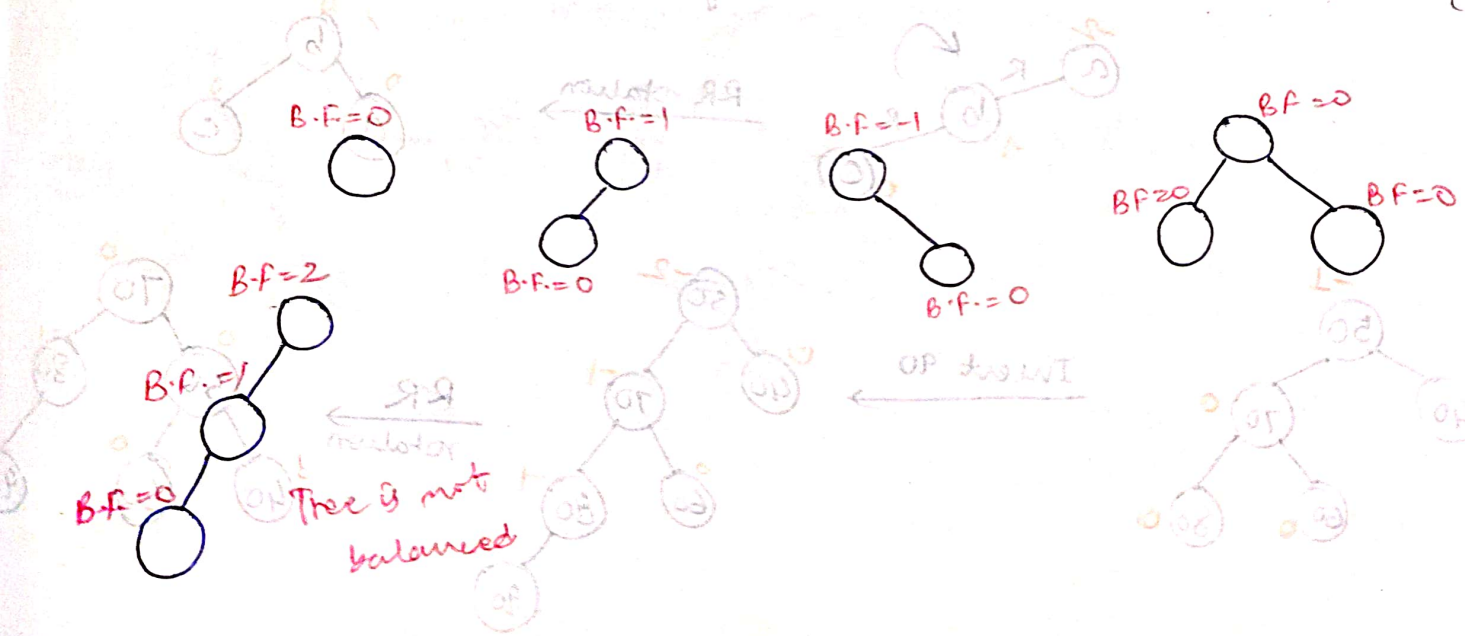
$= 70$

AVL Tree

Self balancing Binary Search Tree.

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

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



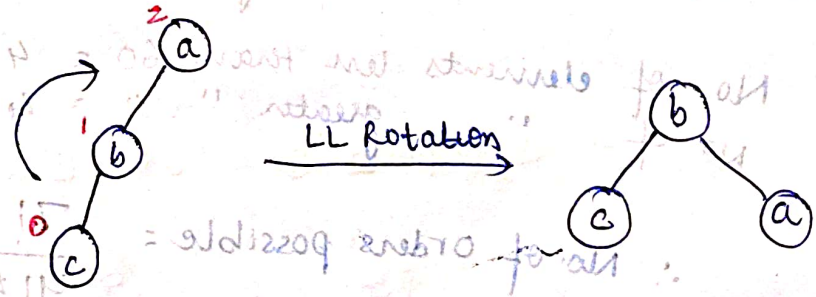
Tree can have imbalance due to insertion or deletion.

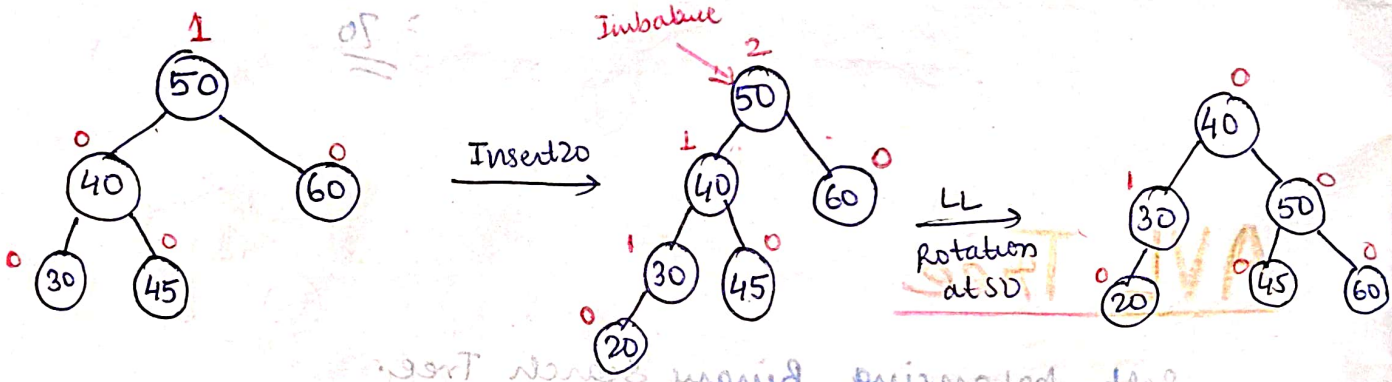
Types of Imbalances

1. LL Imbalance (Left left imbalance)

Solution = LL Rotation

Single Right rotation.

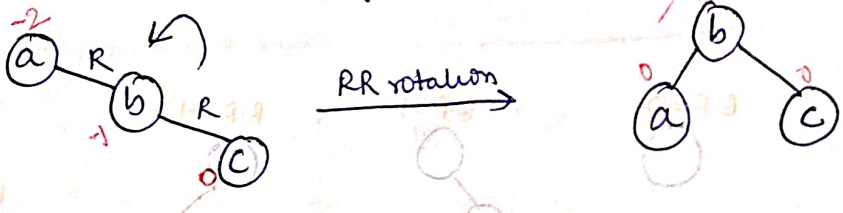


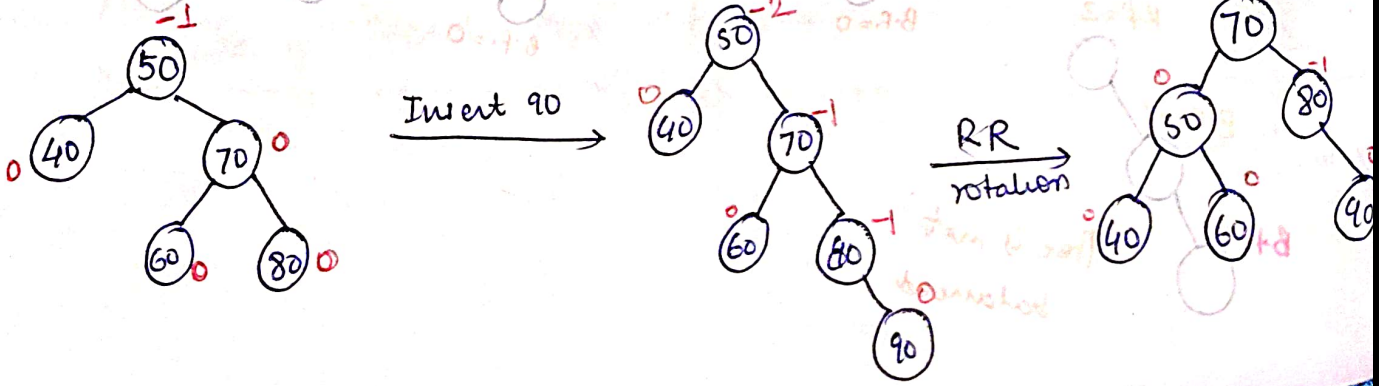


2. RR Imbalance (Right Right imbalance)

Solution = RR Rotation

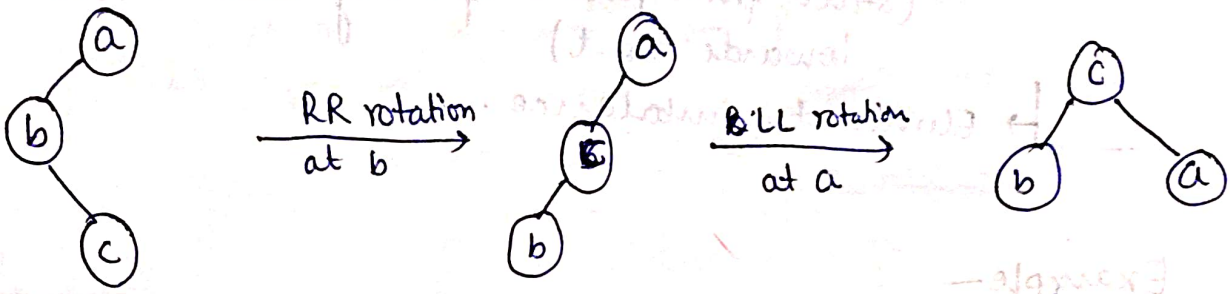
Single left rotation

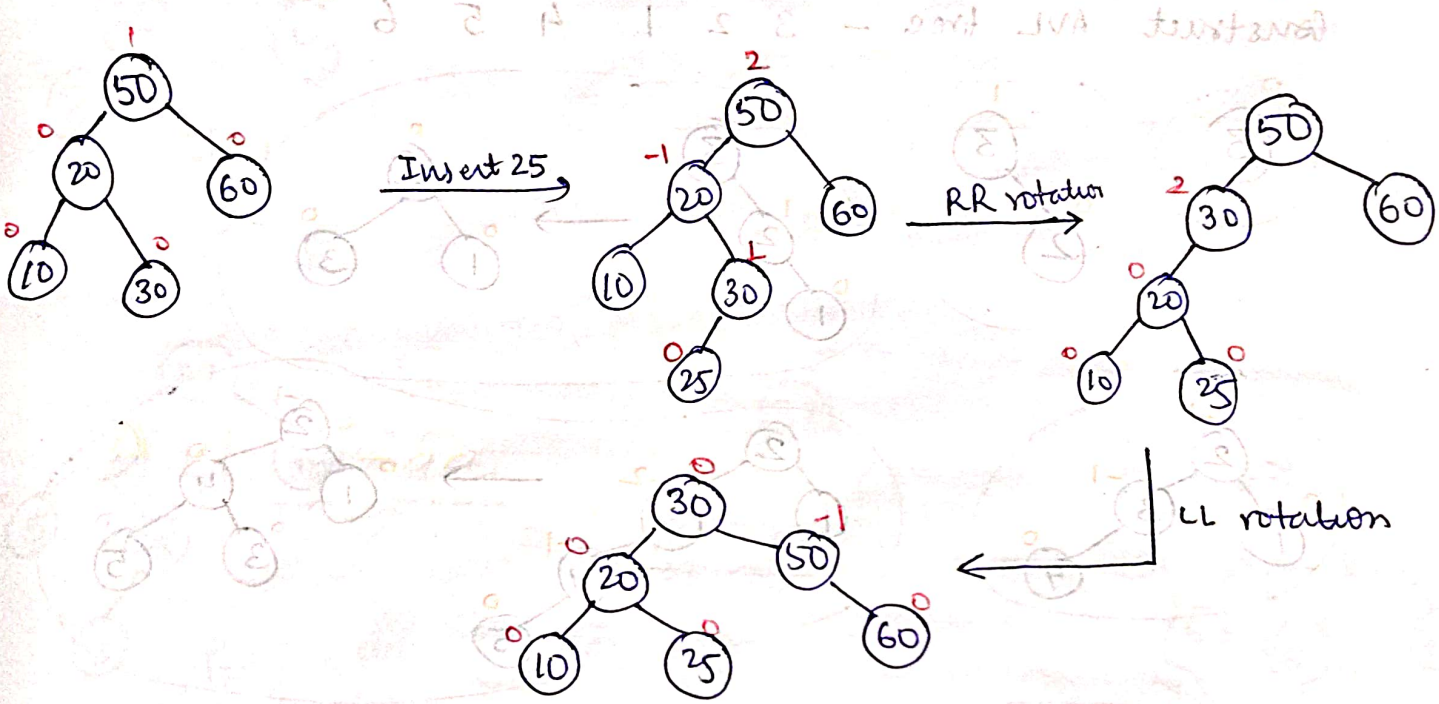




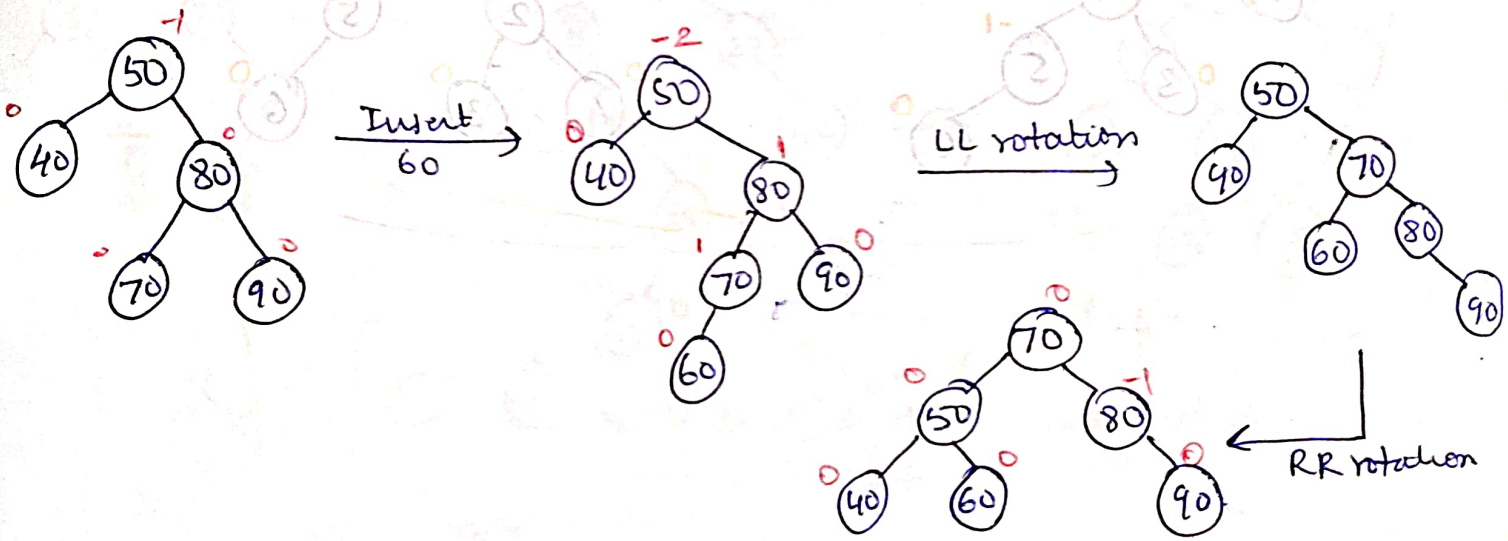
3. LR Imbalance (Left Right Imbalance)

Solution — RR rotation + LL rotation





4. RL Imbalance (Right Left Imbalance)

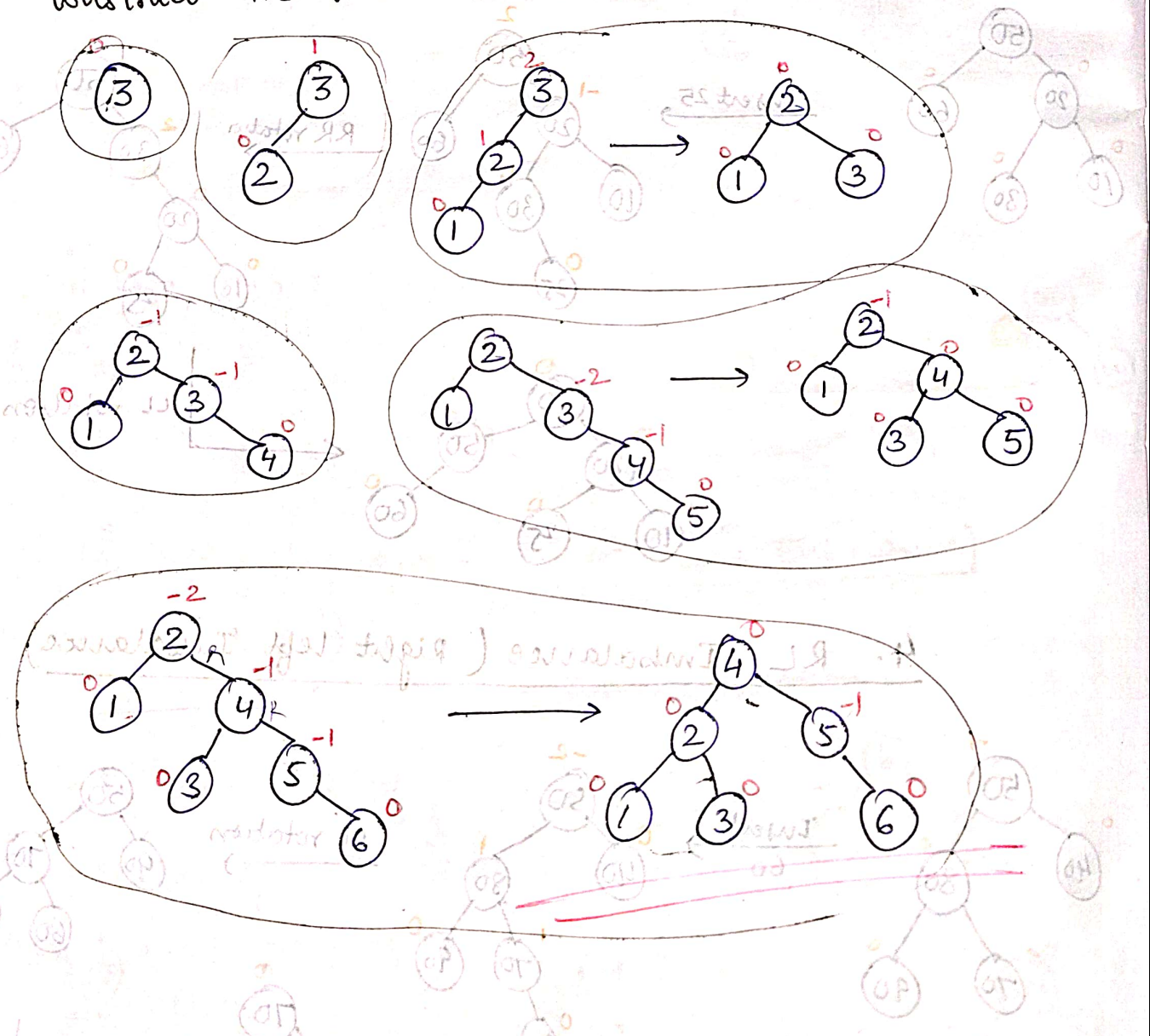


AVL Tree Insertion –

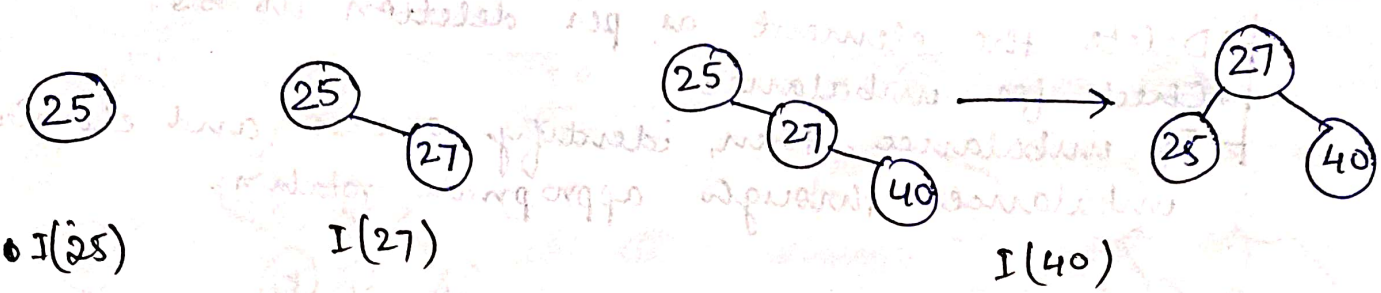
- Insert key as per BST insertion.
- Identify imbalance, if any
  (start from parent of newly inserted node & move towards root)
- Eliminate imbalance.

Example –

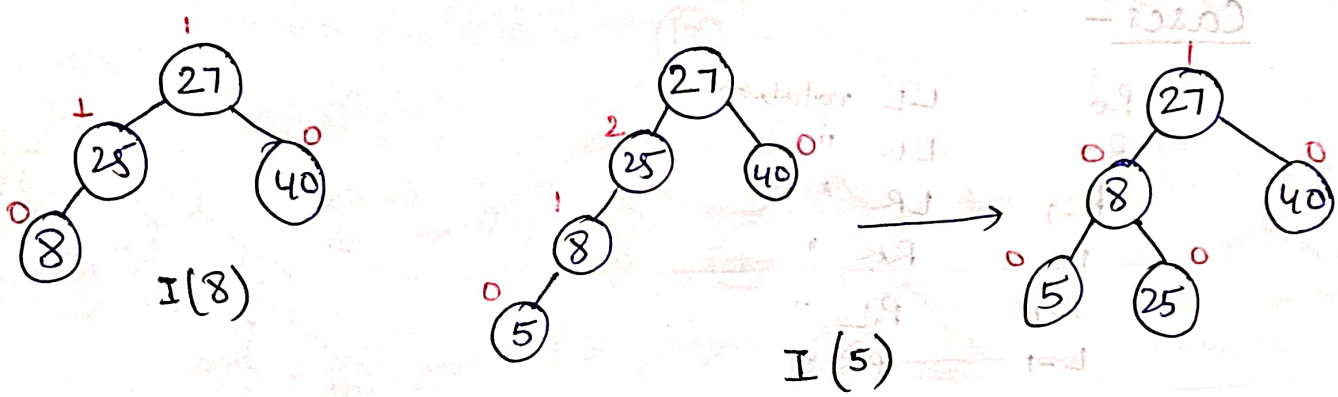
Construct AVL tree – 3 2 1 4 5 6

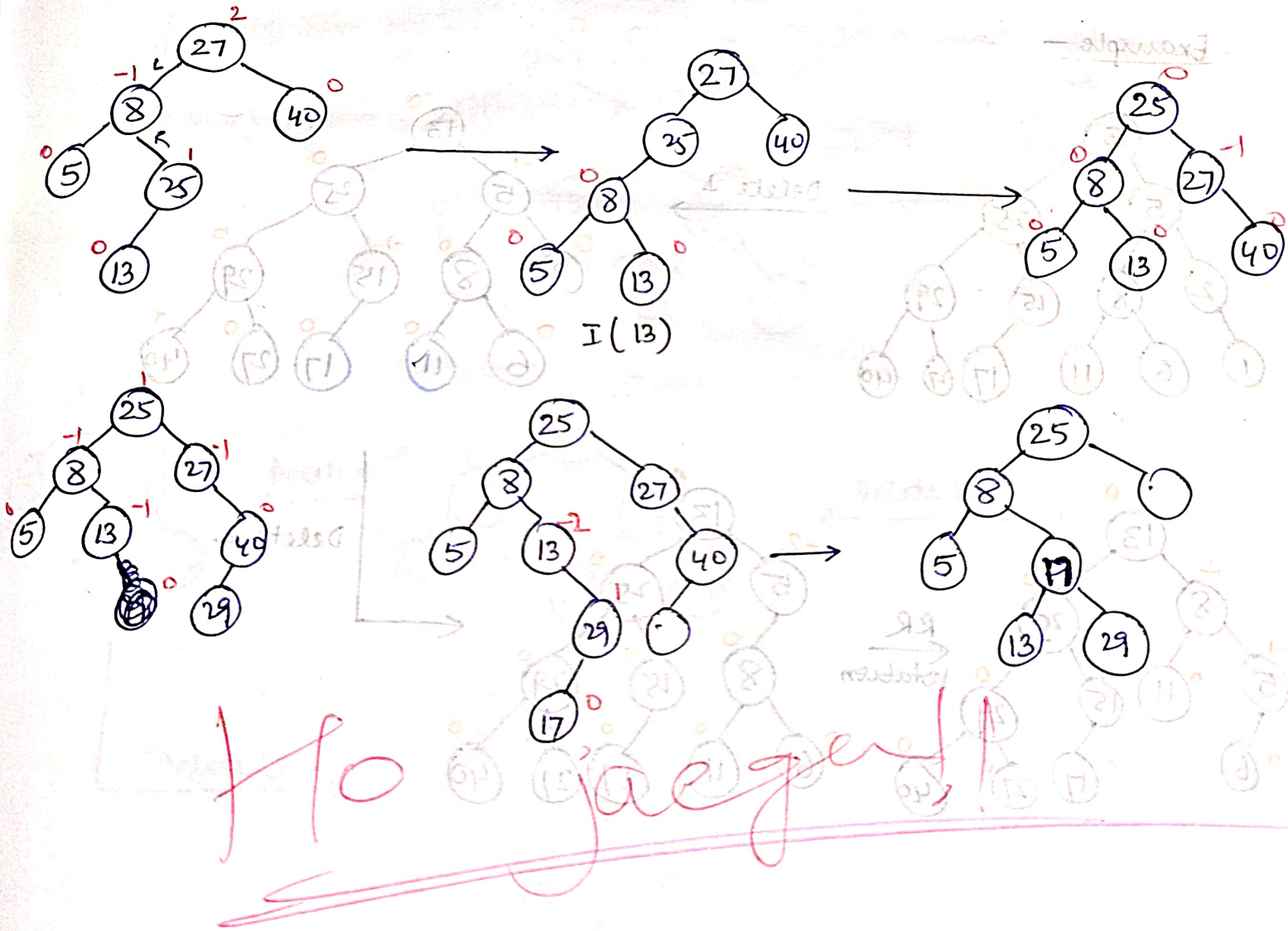


25, 27, 40, 8, 5, 13, 29, 17, 15, 11, 1, 2, 6



• I(25) I(27) I(40)





Hogaegeri!

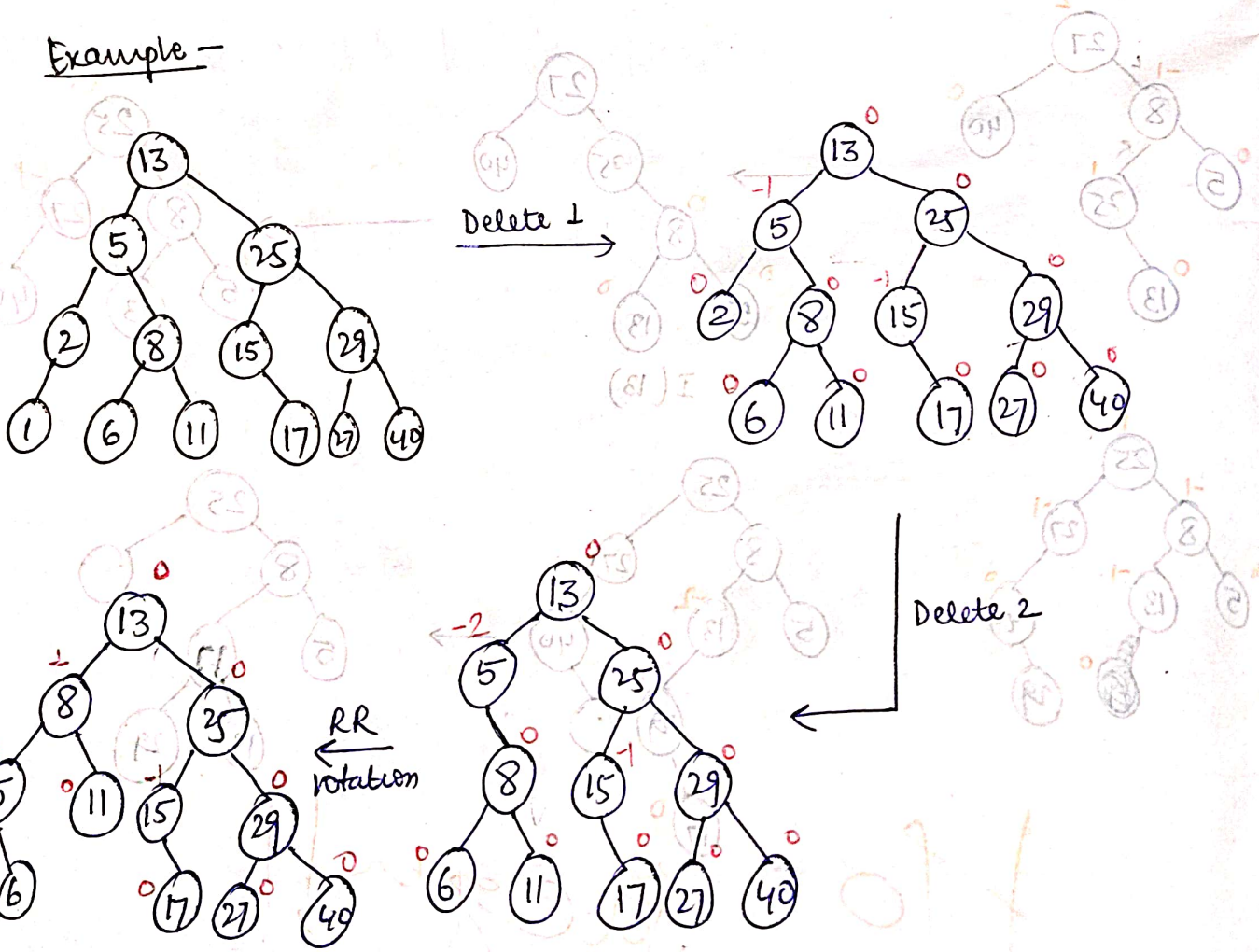
AVL Tree Deletion

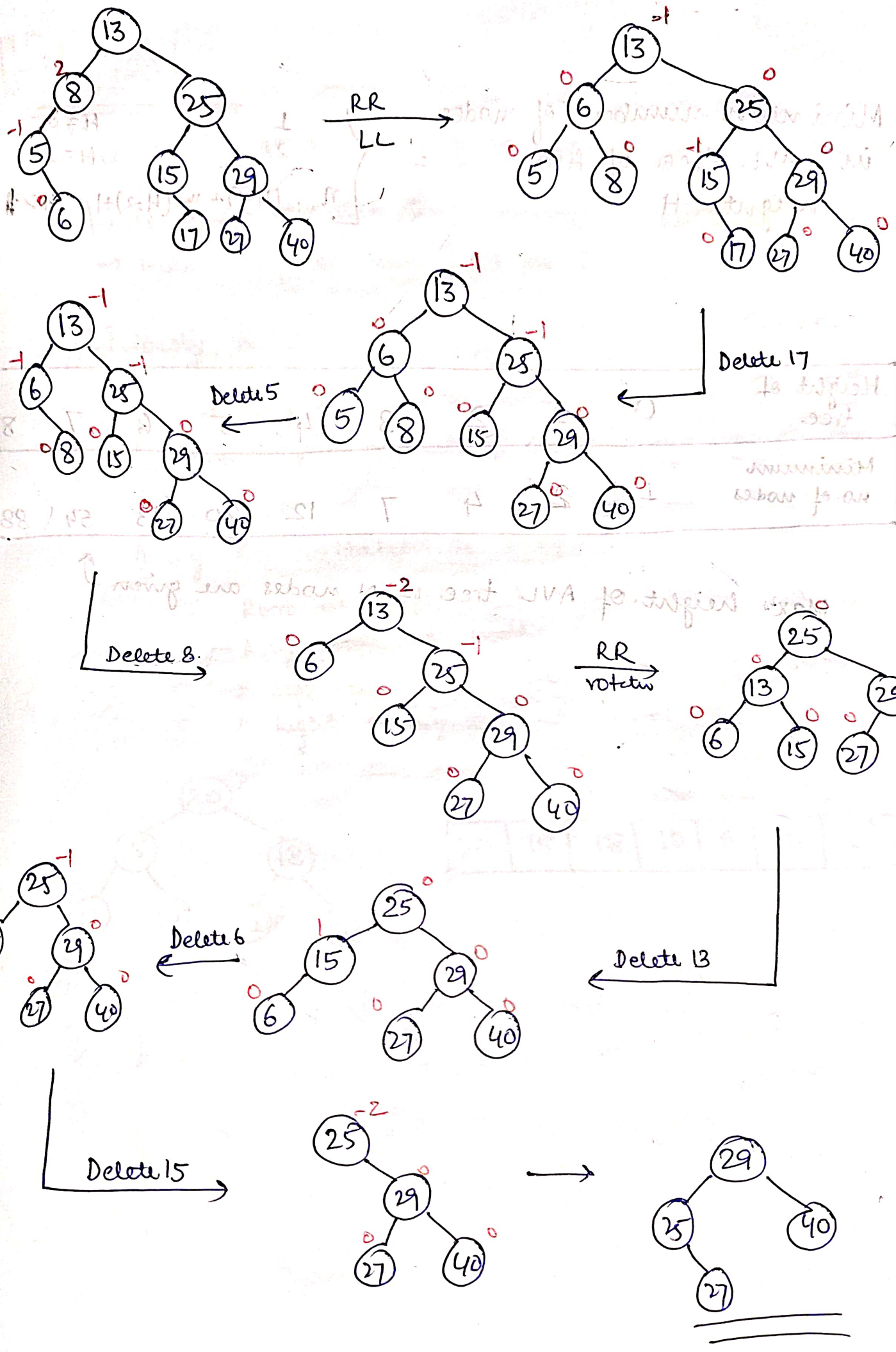
- Delete the element as per deletion in BST
- Check for imbalance
- If imbalance, then, identify a case and eliminate imbalance through appropriate rotation.

Cases –

R_0	LL rotation
R_1	LL "
R_{-1}	LR "
L_0	RR "
L_1	RL "
L_{-1}	RR "

Example –





Minimum number of nodes in AVL tree of height H

$$= \begin{cases} 1 & H=0 \\ 2 & H=1 \\ n_{min}(H-1) + n_{min}(H-2) + 1 & H>1 \end{cases}$$

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

tree based data structure

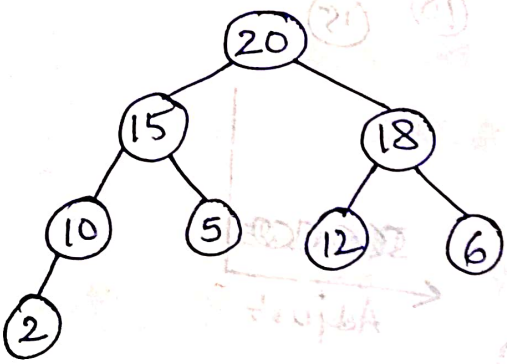
↳ Complete binary tree
↳ Max heap or min heap property

⊙ Priority based data structure

Max heap → value of root is greatest.
Min heap → value of root is minimum

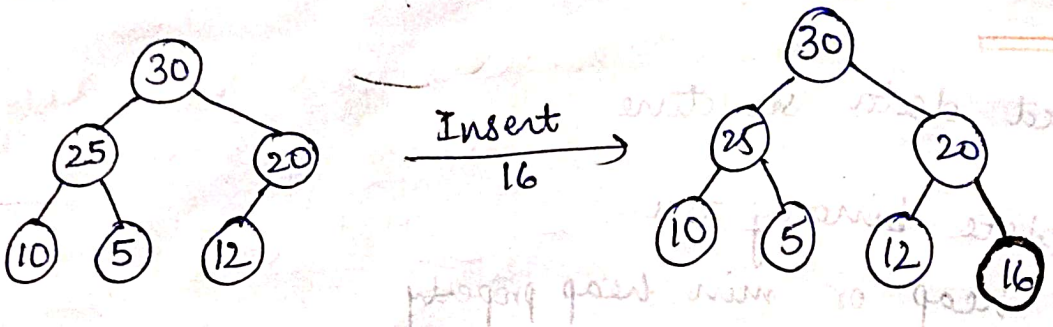
Array Representation of heap –

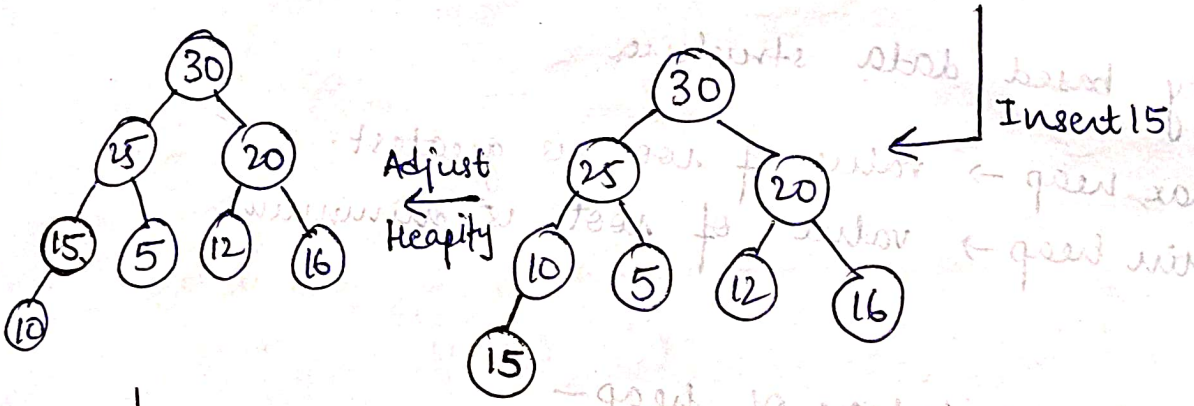
Root at index ⇒ 1
→ Any node at index i
→ left child on index 2i
→ Right child on index 2i+1

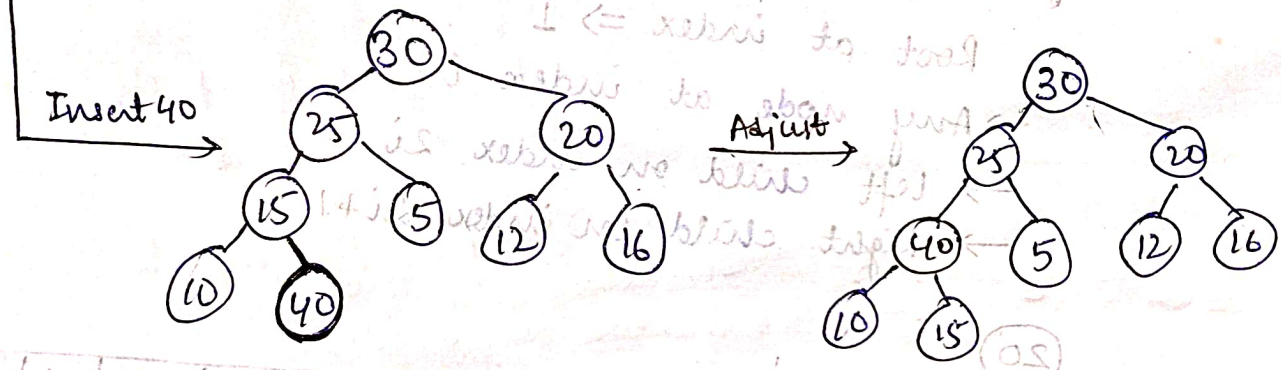


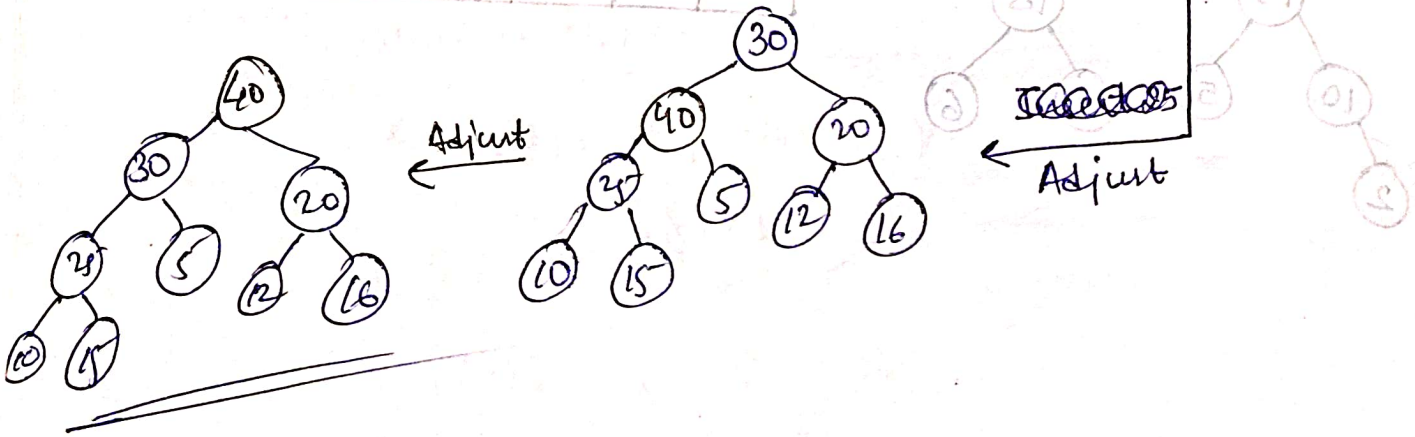
20	15	18	10	5	12	6	2

Insertion in max heap

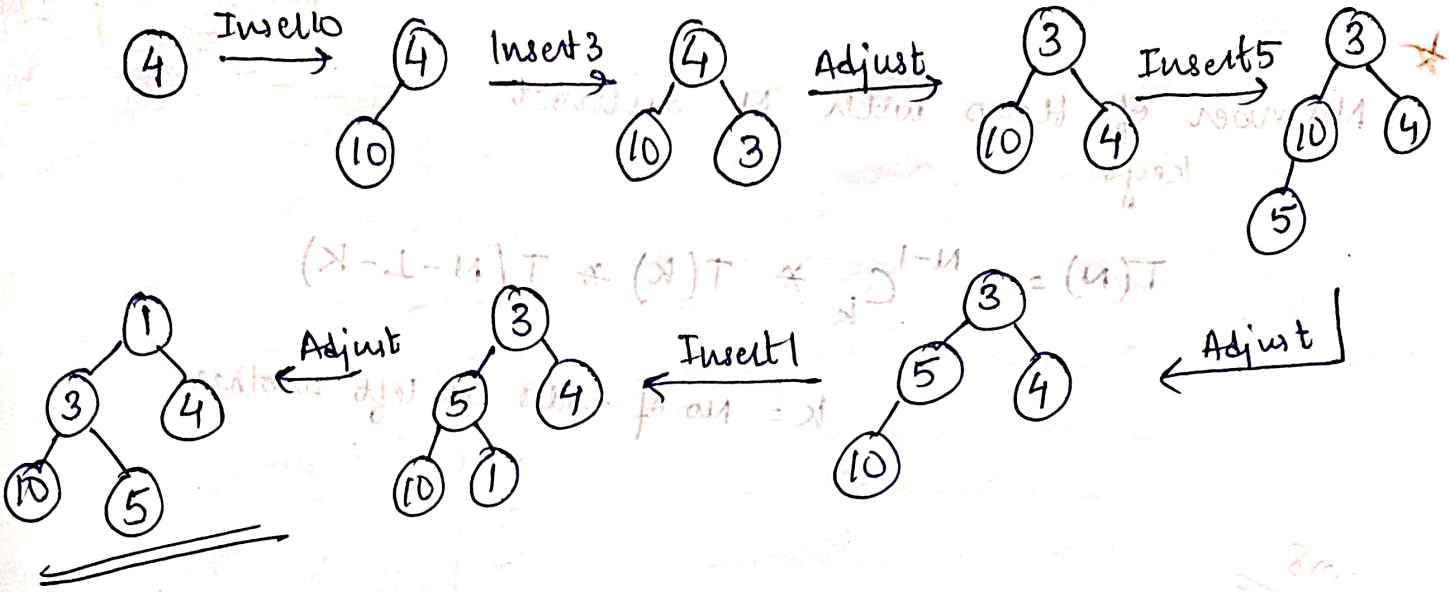








Build Min heap: 4 10 3 5 1



Runtime complexity —

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

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

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

Deletion in heap —

- Delete root element & place the last node of CBT. at root.
- Heapify the root. $O(\log_2 n)$

★ Number of Heap with N distinct keys

$$T(N) = {}^{N-1}C_k * T(k) * T(N-1-k)$$

k = No. of nodes in left subtree

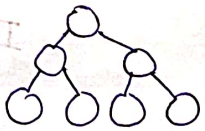
Gate 2018

The number of possible min heaps containing each value from {1, 2, 3, 4, 5, 6, 7} exactly once is ?

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

To calculate T(3) —

$T(3) = {}^{2}C_1 \; T(1) \; T(1)$

$= 2 * 1 * 1 = 2$

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

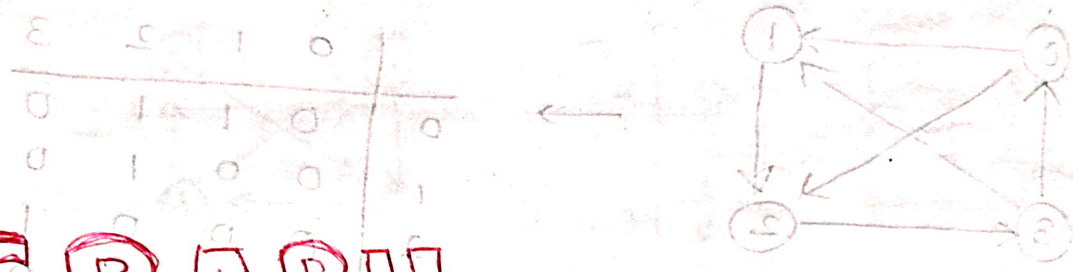
$= \frac{6}{3!3!} * 2 * 2 = \frac{6 \times 5 \times 4}{3 \times 2} \times 2 \times 2$

$= 80$

Heap sort

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

Space complexity = $\Theta(n)$

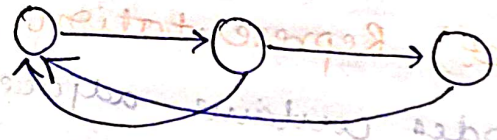


GRAPH

non linear data structure consisting of nodes and edges.

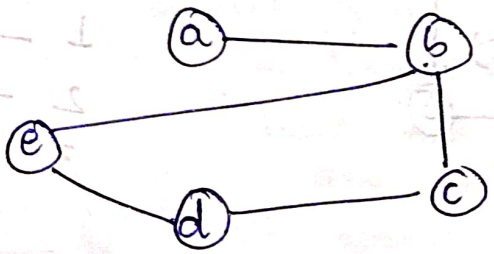
Directed graph –

Each edge has one direction.



Undirected graph –

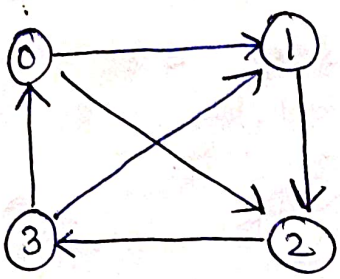
Edges do not have any direction.



Graph Representation —

1. Adjacency Table or Matrix

nxn matrix where n = 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

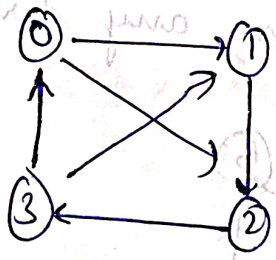
If weighted graph is given,

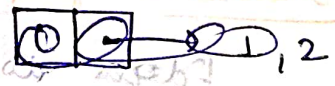
adj[i][j] = ∞ 0 if no edge b/w i & j

w if edge of weight w exists.

2. Linked (List) Representation

Every nodes contains adjacency list for outgoing edges.





0 → 1, 2

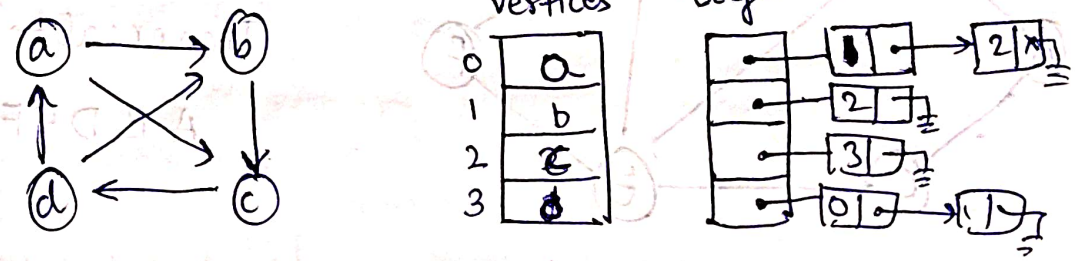
1 → 2

2 → 3

3 → 0, 1

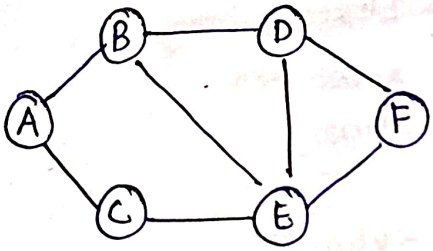
3. Hybrid Representation

Vertices in array.
Edges in linked list.



Traversing in graph

1. Breadth First search (BFS)
   implemented using queue

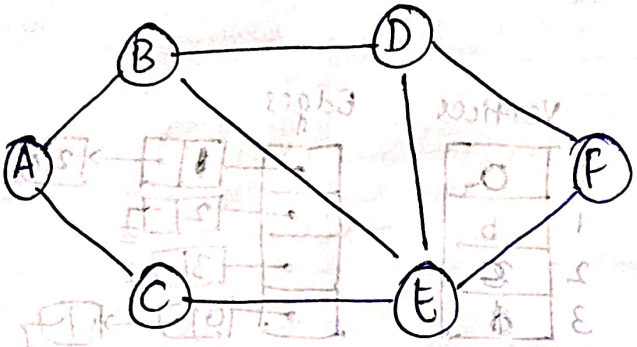


Source = C

C A E B F D

```
Bfs(u) {
    Mark u as visited and Enqueue(u)
    while q is not empty
        x = Dequeue
        Find all neighbours of x which
        are unvisited and enqueue
        them.
}
```

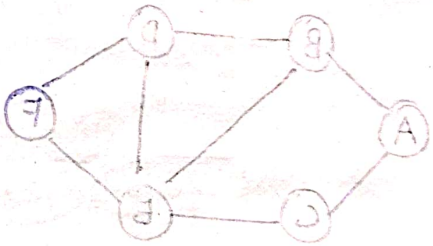
2. Depth First Search (DFS) implemented using stack.



Source A

Traversal –

A B D E F C



HASHING

Searching technique which can provide result in constant time

[Hash function is used to generate the 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

$H(k) = k - x$ x is a +ve integer.

If location range is 0 to 99. $x = 3$

possible keys $\rightarrow$ 3 to 102

Disadv:- keys are limited by location range.

*3. DIVISION METHOD (Modulo Division method)

$L = H(k) = k \bmod m$ [m is usually prime]

Result → 0 to m-1.

Adv:- unlimited range of keys can be stored.

Disadv:- collision is possible.

4. FOLD SHIFTING METHOD

Location range ⇒ 000 - 999.

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

key :- 163 121 382

163
121
382
666 → Location.

If sum ≥ 1000, L = (sum) % 1000

5. FOLD BOUNDARY METHOD

Location range ⇒ 000-999

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

key :- 1234 12 635

123
635
758 → Location

If sum ≥ 1000, L = (sum) % 1000

6. DIGIT EXTRACTION METHOD

Location range – 000 – 999

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

Ex → key : 6 2 9 8 4 3 5

694 – location

7. MID SQUARE METHOD

Take square of the key and pick middle digits.

Collision

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

Example-

$H(k) = k \% 10$

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.

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

Applications

- Symbol Table
- caching
- Databases

* Linear probing — Primary clustering
* Quadriatic probing — Secondary clustering

2. Quadriatic probing–

$L = H(k)$

if there is a collision at L

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

Suffers from secondary clustering.

3. Random probing–

r = random number.

$L = H(k)$

if there is a collision at location L

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

4. Double Hashing–

$L = H_1(k)$

if collision at location L

$r = H_2(k)$

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

Chaining

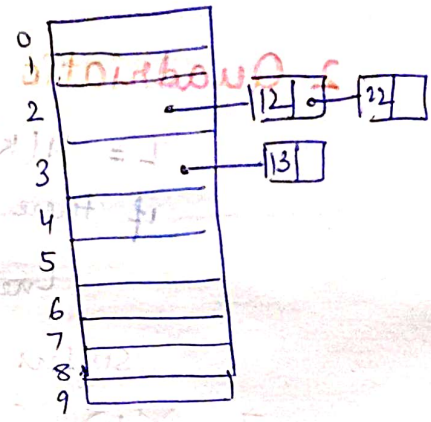
elements which have same hash value are stored in linked list at the same location.

$H(k) = k \bmod 10$

$12 \bmod 10 = 2$

$13 \bmod 10 = 3$

$22 \bmod 10 = 2$



Open Addressing (closed Hashing)	Closed Addressing (Open Hashing)
⊙ If collision occurs at one location, then, other addresses are open to be generated.	⊙ No other location is open to be generated.
⊙ Limited number of elements can be stored.	⊙ Unlimited elements can be stored.
⊙ Overflow possible.	⊙ No overflow.
⊙ Deletion not possible.	⊙ Deletion possible.

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.

→ Quadriatic probing.

Load Factor

Average number of keys per slot.

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

Space Utilization →

No. of slots occupied as compared to total slots.

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

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

For open addressing,

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

for closed addressing,

$$\alpha \geq 0$$

Perfect Hashing

if n keys are inserted in n slots without any collision, then, it is perfect hashing

Minimal Perfect Hashing

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

~~Minimum~~ n keys inserted w/o collision after that collision occur for m-n elements.

Uniform Hashing

hash function that can generate all locations.