Data Structure

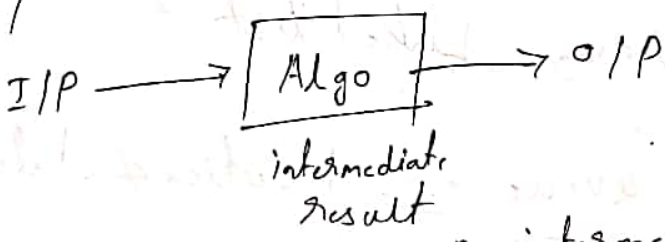
Defination: Mathematical and logical model of organizing the interrelated data.

mathematical → Predefined way of organizing/ storing the data (Stack is last in first out)

Logical → Operations (there is PUSH, POP operation in stack)

interrelated → Data is related to each other

Why DS? : As a programmer we need to solve the problem with minimum effort, time, memory.



We should store the I/P, intermediate results, O/P in a such way that our algo perform in a best way, Thats why we should use a specific structure.

Ex → For sorting data we should not use linked list as we can not randomly access element in linked list.
We use array.

Types :⇒ 2 type of DS are

(i) Linear : Elements are arranged in linear (Sequential) way

ex- array, Linked list, Queues, Stacks

(ii) Non-linear : Elements are arranged in non linear fashion.

ex- Tree, Graph, Hashtable.

Classification based on utility :⇒

We can classify in 3 type

(i) Container :⇒ used for containing data

ex- Array, Linked list

(ii) priority queue :⇒ insertion & deletion is restricted

ex- Stacks, Queue, Heap

(iii) index :⇒ For indexing purpose

ex → Binary Search Tree, Hashtable

Operations :

(i) Traversing → Visiting each and every element atleast & atmost once.
(exact)
(Like Hide & seek)
We use for printing, summation copying

(ii) Insertion → Adding an element in a data structure.
- When data structure is full then we get a error (overflow).

(iii) Deletion → Removing an element from a data structure.
- When there is no data still we want to delete, cause a error called underflow.

(iv) Searching → Finding an element in a data structure.
- output of searching is the location of data (element) if present.
successful (Array (index), Linked List address of node)
unsuccessful (LB or UB+1, Null)

(v) Merging → Combining 2 data structure of same type into one

(vi) Sorting → Arranging elements in ascending or descending order.

Algorithm :⇒ Step by step solution for algorithmic problems.

Problems in computer science that we write a program & run.

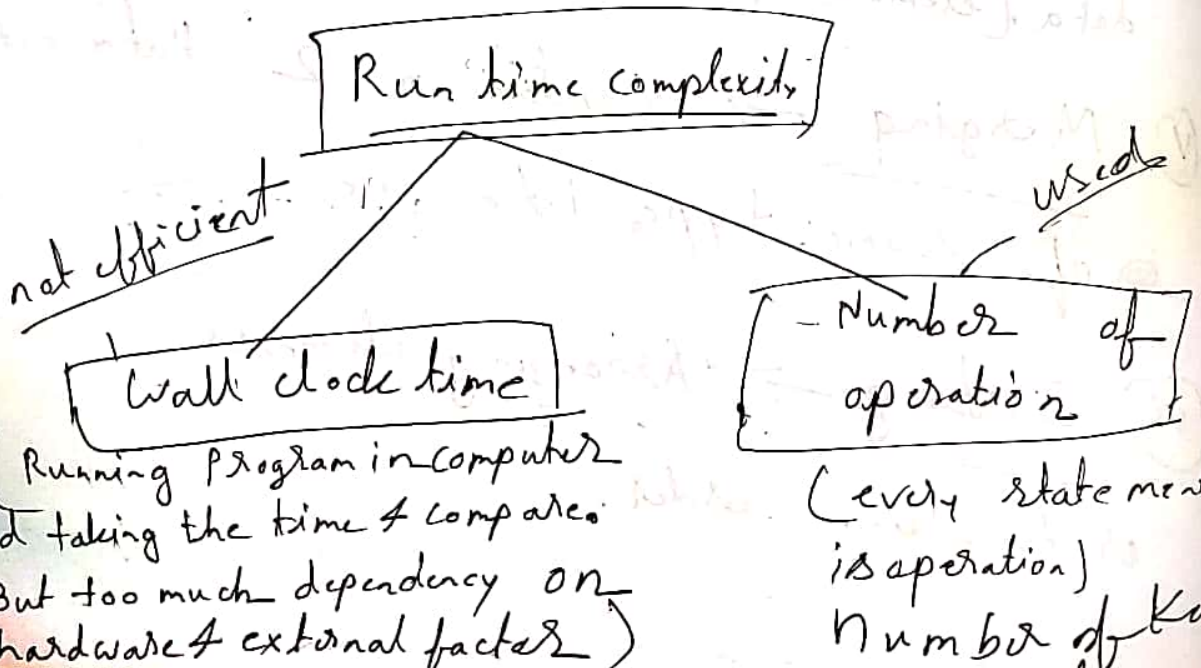
Analysis of Algorithm :⇒ To solve a problem we have multiple solution. To compare the solutions of similar problem we use analysis and pick the best one.

We do analysis by two parameter

(i) Space complexity (how much space to run)
(ii) Runtime complexity (imp as space is not limited).

→ it does not include input space & output space. it only take the space which algo need to run.

→ Time required by algo to run



Ex-1: for (i=1; i<=n; i++) — n times +1

{

x = m+2; —— n times

}

n+1 time (we will check till n time then n+1 will be checked & it will be false)

Total time = 2n+1

ex-2: for (i=1; i<=n; i++) — n+1 times

{ for (j=1; j<=n; j++) n*(n+1)

{ x = m+2; — n*n

}

}

Total - $2n^2 + 2n + 1$

ex-3:

for (i=1; i<n; i=i*2) - log n + 1

{

x = m+2; log n

}

Total - 2 log n + 1

n = 16

i = 1 ✓
i = 2 ✓
i = 4 ✓
i = 8 ✓
i = 16 ✗

} 5 times log n + 1

$\log_2 16 + 1 = 5$

ex- 4 for (i=n; i>1; i=i/2) $\log n + 1$

{ x = m+2; $\log n$

3 ——— Total — $2\log n + 1$

Rate of Growth :⇒

Rate at which the cost of algorithm grows as the size of its input grows.

T(n) = fun on n, n = inputs, T(n) = time complexity

n	T(n)
1	3
2	5
3	7
4	9

T(n) = 2n + 1 — linear function

Complexity

- Constant — 1
- logarithmic — log n
- linear — n
- Linear logarith — n log n
- quadratic — n^2
- cubic — n^3
- :
- Expo — 2^n

low ↓ High

$T(n) = 2n+1$ → complexity = n

$T(n) = 2n^2+2n+1$ — complexity = n^2

$T(n) = 2\log n + 1$ — complexity = $\log n$

Asymptotic Notation :⇒ approaching a given value tends to infinity (asymptotic)

(i) **Big O** :- Tightest upper bound

$O(n)$ means max to max We need linear complexity. We take the worst case.
May be it takes 1 or $n-3, n-2$ but in the worst case it will take n

(ii) **Omega Ω** :- Tightest lower bound

$\Omega(n)$ means it takes minimum n time for run. it may take more than n

(iii) **Theta θ** :- Average bound.

$\theta(n)$ means it takes n time averagely/ exactly n times ← tightest upper bound also n, tightest lower bound also n

$\theta(n) \Rightarrow O(n) \& \Omega(n)$

Array :=>

int [][] arr - in java
int arr[][] - in C

Defination : collection of homogeneous elements / same type

Characteristics :

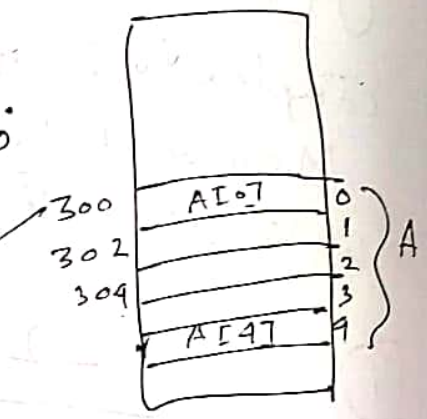
(i) All elements stored on consecutive memory location

(ii) All elements can be accessed using a set of indexes.

Define array In C :=>

datatype name[size];

ex - int A[5];



Lower bound (LB) = 0

upper bound (UB) = (n-1)

Size = upper bound - lowerbound + 1

Define array in D.S. :=>

(*) There is no restriction to be lowerbound as 0.

name[LB : UB] Where LB ≤ UB

ex-

A[-1 : 10] → Size = 10 - 1 + 1 = 10

A[-2 : 8] → Size = 8 - (-2) + 1 = 11

another way Name [LB . . . UB]

ex- Array [1 . . . 10]

it says LB = 1 UB = 10

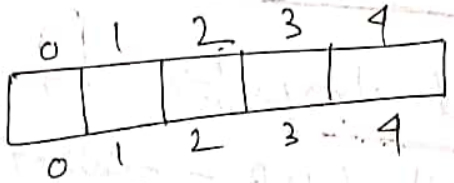
Size = 10 - 1 + 1 = 10

Location of Array element :=>

Location (A[i]) = Base address + W * (No of elements before A[i]) → Relative index

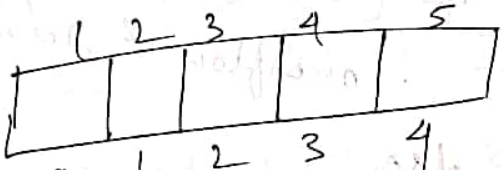
Where W = Size of each element

A[0:4]



RI

A[1:5]



Relative index = index - Lower bound

Q1 consider an array A[-2:15], which is stored in memory starting from location 1000. Each element take 2 location in memory. The location of element A[3] is?

=> LB = -2 UB = 15 BA = 1000 W = 2

RI = 3 - (-2) = 5

A[3] = 1000 + 2 * [5] = 1010

Traversing in Array ⇒ Visiting each element one time.

```
for (K = LB to UB; K++)
  {
    process A[K];
  }
```

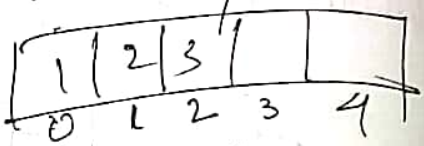
⇒ will run n times

array of n elements

complexity = $\theta(n)$ exactly

↳ copy/print etc.

Insertion In Array ⇒ 2 way



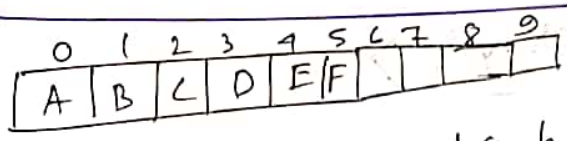
(i) at last always ⇒

```
if (UB == array max)
    overflow & return
else
    UB++;
    Array[UB] = new element
    n++;
```

it doesnot depend of n value So complexity will be constant

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

(ii) Based on given index ⇒ There will be an index i & we have to insert new element at that index only.



We have to insert "x"

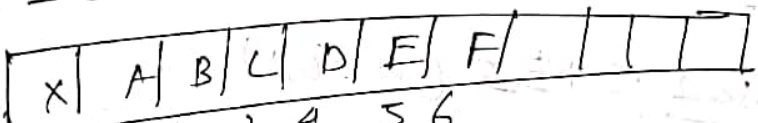
LB = 0
UB = 5
n = 6

Case-1 → i = 6 (Best)

Case-2 → i = 3 (Average)

3 elements shifted toward right

Case 3 ⇒ i = 0 (worst)



all n(6) elements shifted right

Algo-

```
Insert (A, UB, LB, n, ie element)
if (UB == array_max)
    overflow & return
else {
    for (k = UB; k >= i; k--)
        { A[k+1] = A[k]; }
    A[i] = element;
    UB++;
}
n++;
```

depends on value i

loop will run n times as in worst case

Complexity = O(n)

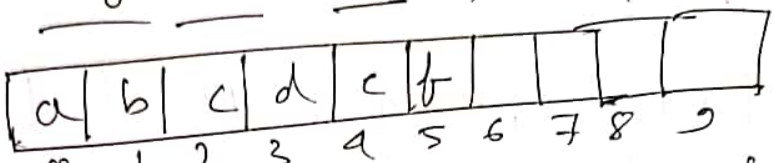
Deletion In Array :⇒

(i) Last element :⇒ to delete last element

```
UB--;
n--;
```

complexity of O(1) or O(1)

(ii) at given index :⇒



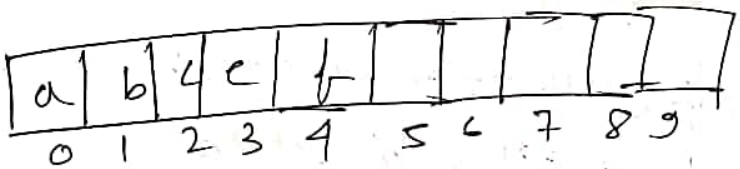
LB = 0, UB = 5, n = 6, i = index of element to be deleted

(LB ≤ i ≤ UB)

Case 1 : i = 5 (i == UB)

```
UB--;
n--;
```

Case 2 :- i = 3



Shift 2 elements left

Case 3 - i = 0

all element shifted towards left

algo is ddelete(AI, LB, UB, n, i)

{
for(K=i; K<UB; K++)
{ A[K] = A[K+1];
}
UB--;
n--;
}

Loop running dependson i
max time it will run n-1 times

So complexity O(n)

Minimum in array ⇒

15, 8, 9, 7, 12, 13, 20, 4, 21, 2, 11, 6, 1, 5

First min = 15 then if 8 < 15 then min = 8
~~then 8<9 true min=8 then 8<7 false~~
9 < 8 false so min still 8
7 < 8 true so min = 7

No of comparison = n - 1

Algo is findmin(AI, LB, UB)

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

exactly (n-1) times

Complexity = θ(n)

Linear Search :=> Sequential search to find whether a number is present in an array or not. Location also returned.

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

if element is present then its index is returned
else if not present then (LB - 1) is returned

algo => Linear_search (A[], LB, UB, item)

```
{
  for (K=LB; K<=UB; K++)
  {
    if (item == A[K])
      return K; // successful
  }
  return LB-1; // unsuccessful search
}
```

for loop: runs max n times (worst case)
So complexity O(n).

Binary Search :=> (Decrease & conquer)

* The array should be sorted (in any order).

Binary search works by comparing the element to be searched by the middle element of the array

case 1 - element = middle, element is found

case 2 - element > middle then search the element in the sub-array starting from middle+1 index to upper bound

case 3 - element < middle then search for element in the subarray starting from Lower bound to middle - 1.

Algo -

```
Binarysearch ( A[I, LB, UB, item )
{  Low = LB
   high = UB
   mid = (Low + High)/2
   While (Low <= high && A[mid] != item)
   {  if (item < A[mid])
         high = mid - 1;
      else
         low = mid + 1;
      mid = (low + high)/2;
   }
```

```
if (A[mid] == item)
    return mid;   // successful
else
    return LB-1;  // unsuccessful
}
```

Time complexity $T(n) = T(n/2) + k$

$\Downarrow$

$O(\log n)$

2 D Array => collection of arrays

in C

int A [m] [n] => m x n = no of cells

m = no of rows

n = no of columns

in DS

name [LB_i : UB_i] [LB_j : UB_j]

m = no of rows = $UB_i - LB_i + 1$

n = no of columns = $UB_j - LB_j + 1$

Size of array = m x n (no of elements)

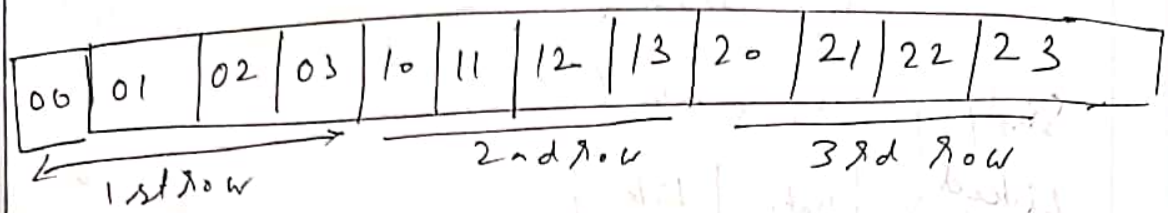
A [2:6] [3:9] => m = 6-2+1 = 5

n = 9-3+1 = 7

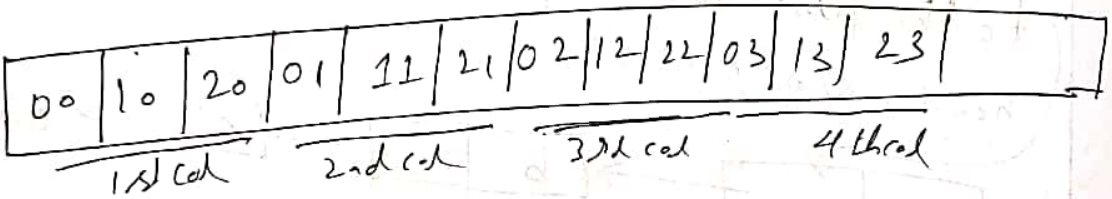
number of elements = 7 x 5 = 35

Row Major order: All the elements are stored row by row

00	01	02	03
10	11	12	13
20	21	22	23



Column Major order: => All the elements are stored column by column



Location in row major order is

$$Loc[A[i][j]] = Base + w * [(i - LB_i) n + (j - LB_j)]$$

n: no of elements in every row

Location in column major order is

$$Loc(A[i][j]) = Base + w * [(j - LB_j) * m + (i - LB_i)]$$

m: no of elements in column

Adv of array — Random access

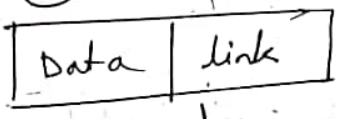
Disadvantage — (i) Static allocation

(ii) Scalability (insert element cost high)

Linked List

- Linear data structure
- Linear order maintained using pointer (link)
- List contain nodes
- Node contain 2 fields
  (i) Data (key or value)
  (ii) Link (address of next node)

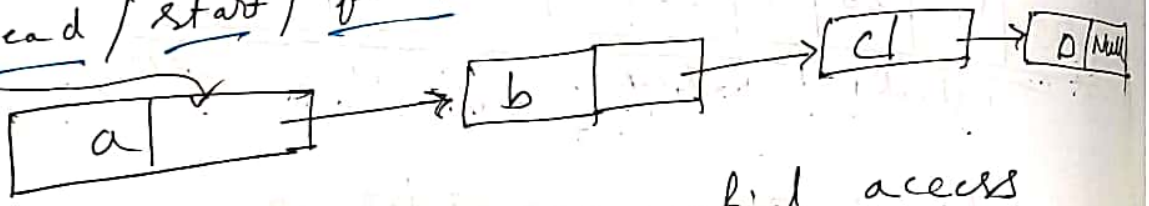
Singly Linked List



node

one link

- There is a list pointer which point to the first node of list. (name can be head / start / first)



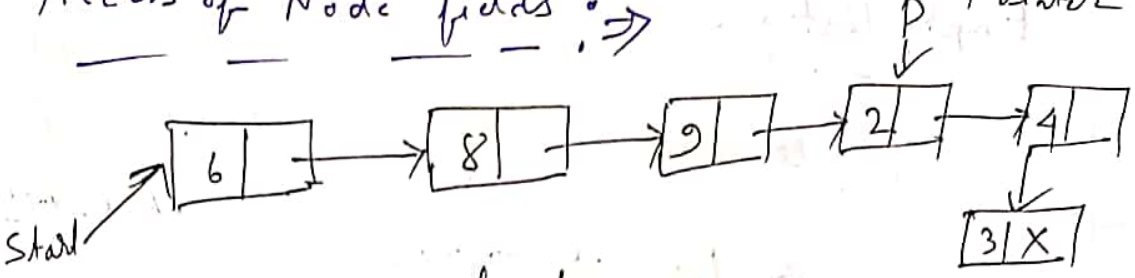
Disadvantage: Only sequential access possible

Structure of node:

```
struct node
{
    char data;          // data type of data
    struct node *link;  // node type pointer
};
```

=> Self Referential structure

Access of Node fields :⇒



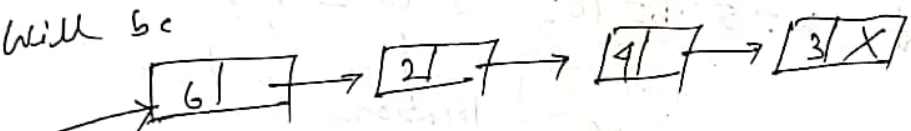
We use arrow pointer to access the value of node

```
start → data ; // 6
start → link ; // link of 1st node
```

if p point to 4th node then

```
printf ("%d", p → data); // 2
printf ("%d", p → link → data); // 4
```

⊛ if we write start → link = p then list will be




```
printf ("%d", start → link → data); // 2
```

Null Pointer Derferencing :⇒

if we write → start → link → data (it will give error

Null (not a value, its having nothing

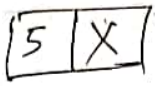
⊛ Null → data } error null pointer
or dereferencing
Null → Link

Empty list :⇒ if there is nothing in list

```
if ( start == NULL )
{ list is empty;
}
```

or

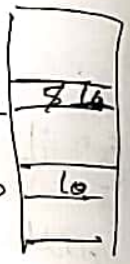
```
if ( ! start )        (false)
{ list is empty
}
```

Single node : 


```
if ( start -> link == NULL )
{ single node list;
}
```

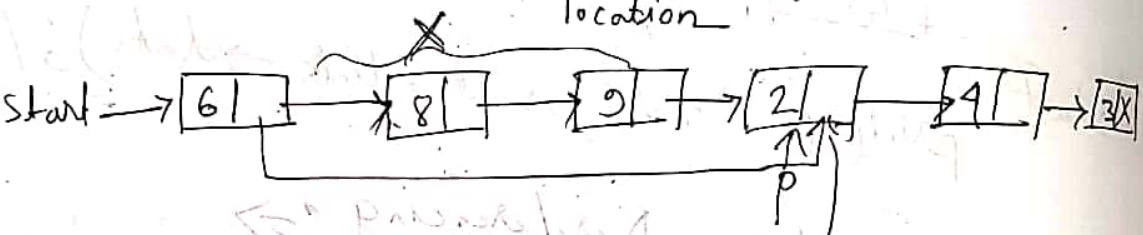
Link assignment :⇒

```
int a = 5, b = 10;
```




```
a = b;
```

(a = location, b = value)



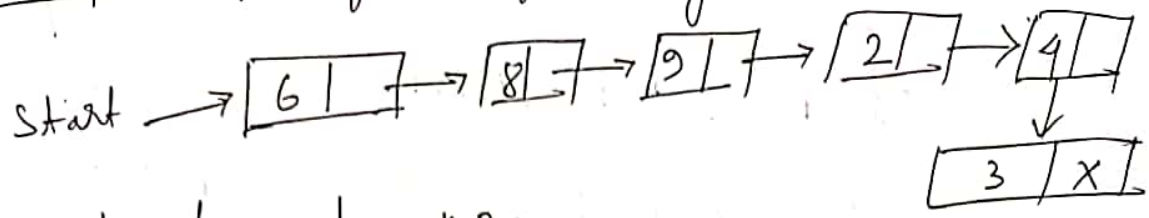
start -> link = p

(p = location of this node)

link part of start pointer

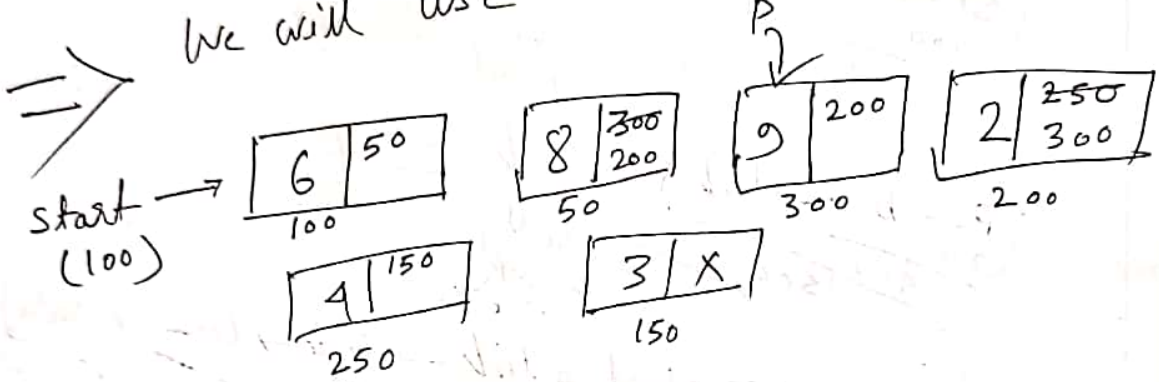
Singly link list — Only Single link

Q1 Output of the following




```
struct node *P;
P = start -> link -> link;
start -> link -> link = P -> link;
P -> link -> link = P;
printf("%d", start -> link -> link -> link -> data);
```

$\Rightarrow$ We will use address



$P = start \rightarrow link \rightarrow link; \Rightarrow P = 300$

(100 → 50 → 300)

$(start \rightarrow link) \rightarrow link = P \rightarrow link$

(100 → 50; 300 → 200; 300 will be updated)

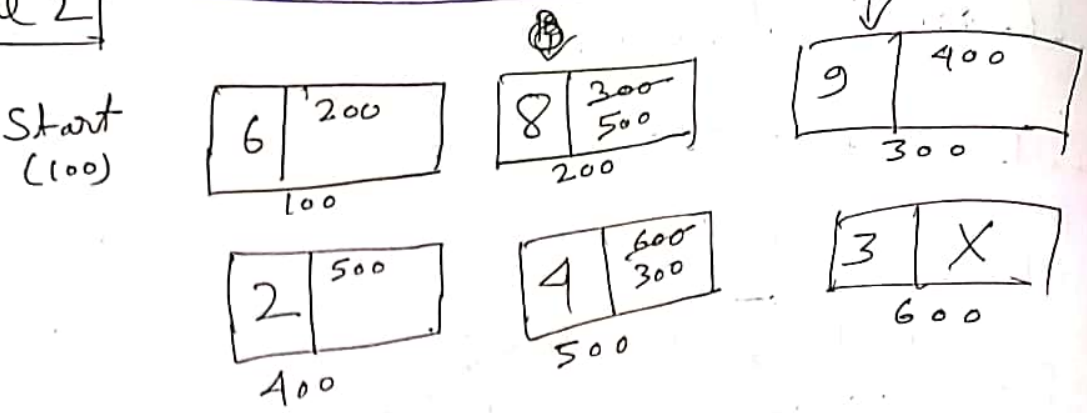
$(P \rightarrow link) \rightarrow link = P;$

(300 → 200; 250 will be updated → 300)

printf("%d", start → link → link → link → data);

(100 → 50 → 200 → 300 → 9)

Q 2



struct node *p;

p = start → link → link ; p = 300
100 200 300

(start → link) → link = p → link → link
100 200 300 will be replaced 300 400 500

(p → link → link) → link = p
300 400 500 600 will be set 300

printf ("%d", start → link → link → link → data)
100 200 500 300 ⇒ 9

Traversing in a Singly Linked List ⇒

struct node *p
p = start
While (P ≠ NULL) or While (P)
{ process P → data
P = P → link
}

Complexity
O(n) exactly

Counting nodes :=>

```
int count = 0;
struct node *p;
p = start;
while ( p ≠ NULL)
    { count++;
      p = p → link;
    }
```

Complexity = $\theta(n)$

Finding address of last node :=>

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

Disad – if we give empty list then p → link will give error Null pointer dereferencing error

To avoid error if (p == NULL)
return NULL;

complexity $\theta(n)$

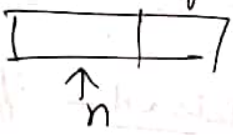
Insertion :⇒ We can create new node dynamically using dynamic memory allocation then we can insert value

Can create a new node →

```
struct node *n = (struct node*) malloc (sizeof (struct node));
```

n → new node pointer
(struct node*) → type (struct node pointer)
sizeof(struct node) → Sizeof node

⇒ address of node is in n.



(if malloc can't create a memory)

```
if (n == NULL)
    {memory full
    }
```

There is 3 cases for insertion
(i) At starting (ii) After a node (iii) at end

(i) At starting :

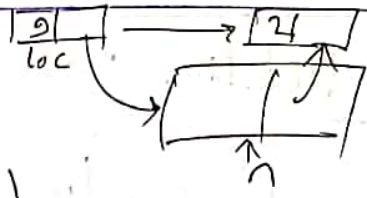
```
insertion (start, element)
{
    n->data = element;
    n->link = start;
    start = n;
}
```

only 3 statement

$\theta(1)$ Constant

(*) This algo can also add new element in empty list.

(ii) After a given node




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

$\theta(1)$ Constant complexity

(iii) At the end:

if last node location is not known

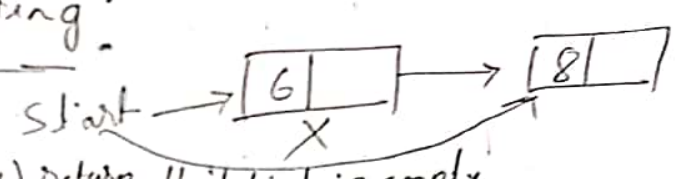
```
Insertion(start, element)
{
  n -> data = element;
  n -> link = NULL;
  if (start == NULL)
  {start = n
   return
  }
  p = start
  While (p -> link)
  { p = p -> link;
  }
  p -> link = n; // address of last node
  return
}
```

$\theta(n)$ (as we have to find the address of last node)

if address of last node is known then $\theta(1)$ constant complexity

Deletion :⇒

(i) at starting:



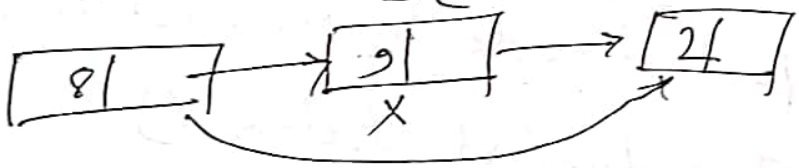

```
if (start == NULL) return // if list is empty
    p = start;
    start = start → link;
    free(p);
```

for deallocating memory

O(1)

complexity

(ii) of a given node:



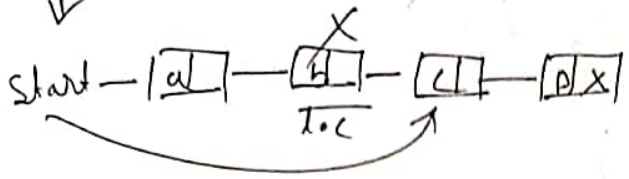

```
if (loc == start)
  { start = start → link // if 1st node
      free(loc)
    return
  }
p = start
while (p → link != loc)
    { p = p → link;
    }
p → link = loc → link;
free (loc)
```

O(n)

not sure so in case upper bound

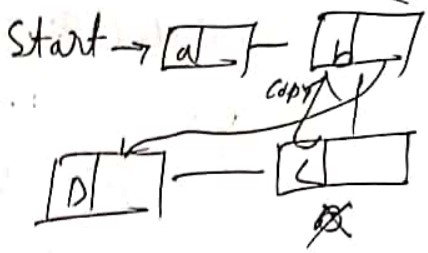
Delete given node without pointer extra (interview)

+ two invalid way



start = loc → link

we delete all the node ~~ca~~ before loc & loc itself



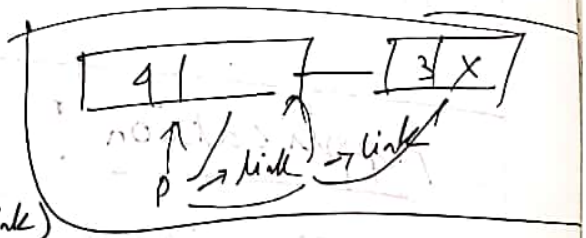
Loc → data = loc → link → data

loc → link = loc → link → link

(Works only if loc is not pointing to null)

(iii) At the end :

```
if (start == null)   // empty list
    return
if (start → link == NULL)   // single element
  { start = NULL
    return                  // converted to empty list
  }
p = start
While (p → link → link)
  { p = p → link ;
  }
p → link = NULL.
```



$\theta(n)$

When no location of last node is given

if location of last node is given then $\theta(1)$

Searching :=> We use linear search as random access not allowed.

```
search (* start, item)
{
    p = start
    while (p)
    {
        if (p -> data == item)
        {
            return p; // successful
        }
        p = p -> link
    }
    return NULL; // unsuccessful
                 /p
```

Complexity O(n) —

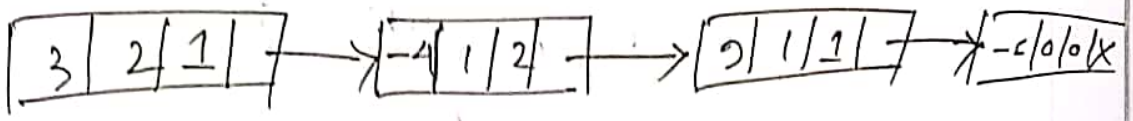
⊗ We can apply binary search but not efficient in linked list as traversing hard.

Application :

- To store polynomial
  - Univariate (Single) + Bivariate (two)

$3x^2 - 5x - 9$

- $3x^2y - 4xy^2 + 9xy - 6$



We can then Subtraction, addition etc.

DPP (Question) :⇒

(1) Set is represented in linked list with element in arbitary order. Which of the operation among union, intersection, membership & cardinality will be slowest

(a) Union only (b) intersection, membership
(c) cardinality membership (d) union, inter

⇒ membership → Searching $O(n)$
Cardinality → no of elements $\Theta(n)$
union $= O(n^2)$
intersection $= O(n^2)$

$S_1 = \{a, b, c\}$
$S_2 = \{b, c, d\}$

$S_1 \cup S_2 = \{a, b, c, d\}$
first if a is present in S_2 then dont add else add
at last add all elements of S_2

$S_1 \cap S_2 = \{b, c\}$ if element of S_1 is also added in S_2 then put in inter section

2] int func (node *start) on a valid will terminate list

{ struct node *p;

p = start;

while (p → link)

{ p = p → link;

}

return 1;

}

(a) 1 always (b) none (c) will cause error

(d) error at 1 ✓

⇒ Start = NULL then return NULL = start // error

Start ≠ NULL then return 1

3] C function takes a singly-linked list of integer as parameter & rearrange it. List containing integer 1, 2, 3, 4, 5, 6, 7. What will final content of list

struct node { int value;

struct node *next; };

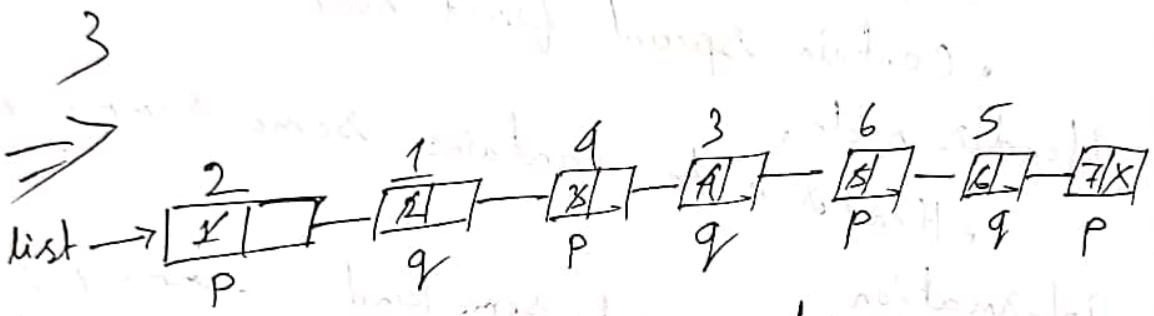
void rearrange (struct node *list)

{ struct node *p, *q;

int temp;

if (!list || !list → next) return;

```
p = list;   q = list → next;
while (q)
{
    temp = p → value;
①   p → value = q → value;
    q → value = temp;
    p = q → next
    q = p ? p → next : 0;
}
```



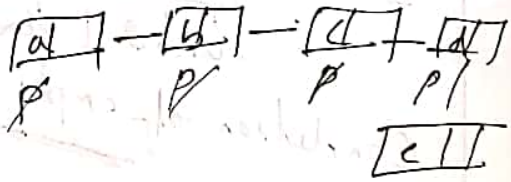
!list ✗ !list → empty ✗ return ✗

① is used for swapping value

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

---

④ O/P will be




```
struct node *p;
p = start;
while (p → link → link)
{
    p = p → link;
}
printf ("%c", p → data);   // data of second last block
```

null error for empty & single node list

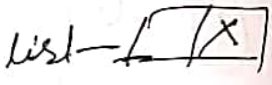
Disadv. of Singly linked list :

- The link part of last node is not utilized [NULL]
- The address of previous node is not known So backward stepping is not possible / easy.
  - Solution - circular linked list
  - Solution - Doubly linked list

Header List :

- Contain special first node called Header node.
- Header node contains some summary information
  * if we need to need some kind summary frequently, then we store it inside the first node itself instead of a variable (useful for big organization team)

Condition of empty list in header list is —

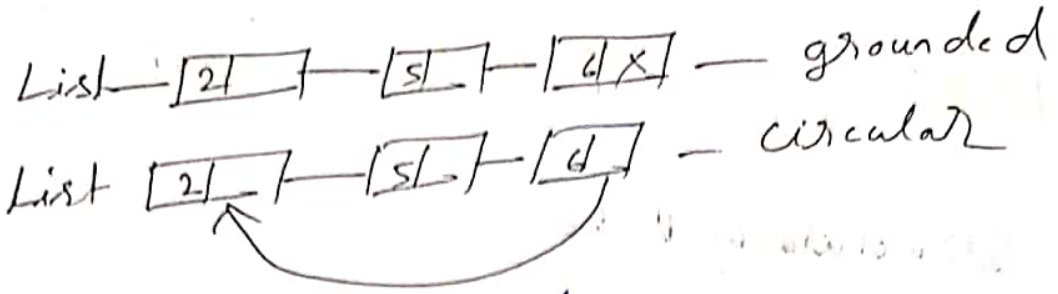



```
if (list -> link == NULL)
    { header list empty
```

3

Types — (i) Grounded : last node link is NULL

(ii) circular : last node link point to header node address



Traversing in Header List :

(i) Grounded :

```
p = list → link
while ( p != NULL)
{ process p → data
  p = p → link
}
```

(ii) circular —

```
p = list → link
while (p != list)
{ process p → data
  p = p → link
}
```

Will try insert, delete later

Doubly linked list :=>

Structure will be

```
struct dnode
{ char data;
  struct dnode *next;
  struct dnode *previous;
};
```

Insertion :=>

(i) at starting :

```
Insertion (First, last, item)
{ n -> data = item
  n -> next = First
  n -> prev = NULL
  First -> prev = n
  First = n
};
```

O(1)

(ii) insertion after node:

```
Insertion ( First, Last, Loc, item )
{
    n → data = item
    n → next = loc → next
    n → prev = loc
    n → next → prev = n
    loc → next = n
}
```

$\theta(1)$

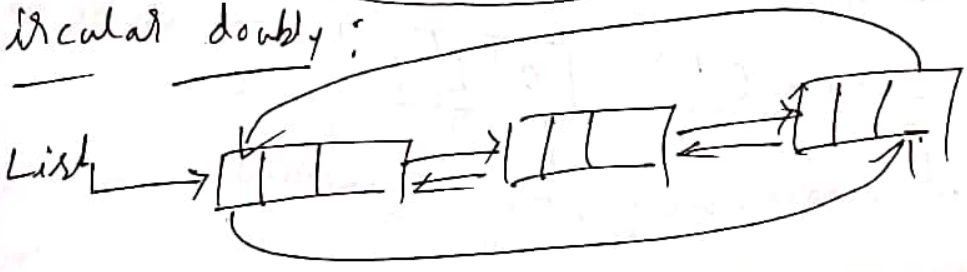
(iii) at end:

```
insertion ( First, Last, item )
{
    n → data = item
    n → next = NULL
    n → prev = Last
    n → prev → next = n
    Last = n;
}
```

$\theta(1)$

Circular linked list

Circular doubly:



Queue

A linear data structure in which insertion can be done from one end (rear end) and deletion is done from other end (front end).

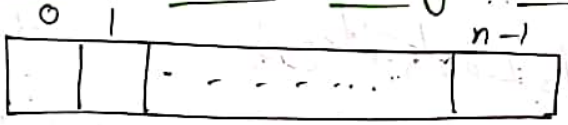
ex- Standing in a line for movie ticket

- It is FIFO (Fast in First out) List.

Insertion is called Enqueue

Deletion is called Dequeue

Implementation using Array :=>



array of size n

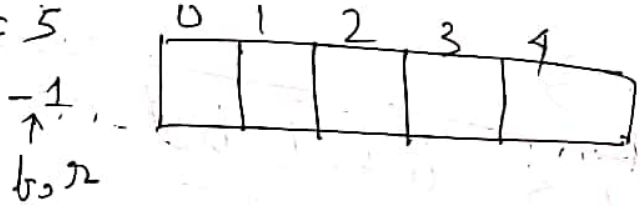
front = points to the first element

rear = points to the last element

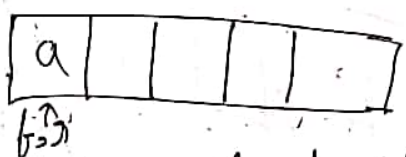
(*) Empty queue :=>

front = rear = -1 (as array starting from 0)

(*) assume n = 5

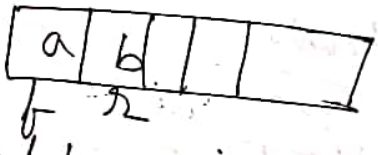


(i) enqueue(a)



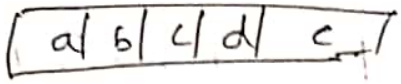
While inserting first element f & r both points to the 0 index (first elmt)

(ii) enqueue(b)



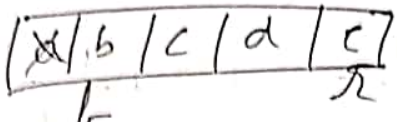
as we have empty space, we insert b

(iii) enqueue(c), enqueue(d) enq(e)

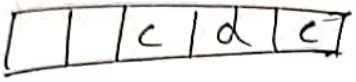


(iv) Enq(f) - overflow as queue is full

if((f==0) && (r==n-1)) (for Linear queue).

(v) Deq() - 

we are doing f++ (we will loose access to a)

(vi) Deq()- 

(vii) Enq(h) - a rear is pointing to the last element so a new element cannot be added in linear queue but can be added if we use circular queue

Queue

Linear

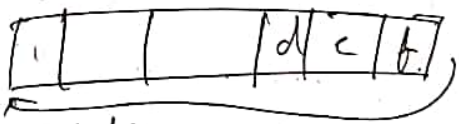
(next element can be enqueued at only next index of rear)

rear end points to last index but there is empty space. instead we cannot add element as its overflow.

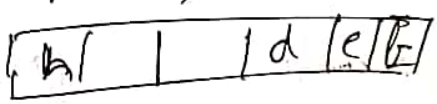
- not a proper way of utilizing array space
- It will take a lot cost for shifting.

Circular

(next element can be inserted on next index of rear in wrap around

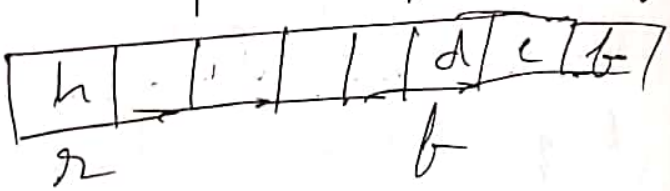


enq(h) =>



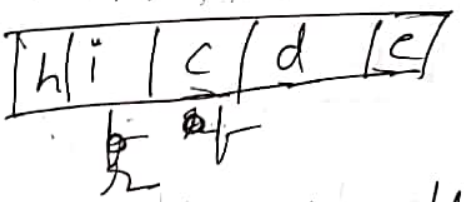
- proper utilization of array space

(*) queue element present from front to rear



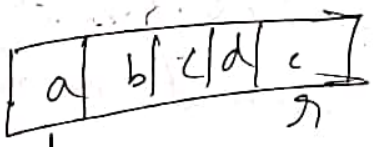
elements are - d, e, f, h

(viii) Enq(i) -



(ix) Enq(j) :⇒ we can't add as there is no space which cause overflow

$f = r + 1$

but if  then $f = r+1$ is not worked so the best will be

$$front = (rear + 1) \bmod n$$

Enqueue Algo :⇒

```
Enqueue (Queue[], f, r, item)
{
For overflow → 1. if (front == (rear+1) mod n)
                    overflow & return
               2. if (front == rear == -1)        θ(1)
For empty           front == rear = 0              constant
               else
                    rear = (rear+1) mod n
For not empty in both linear & circular → 3. Queue[rear] = item
```

Dequeue Algo:

```
Dequeue ( Queue[], front, rear)
{
1. if ( front == rear == -1)
       underflow & return
2. item = Queue[front]
3. if ( front == rear)
       front = rear = -1
   else
       front = (front+1) mod n
}
```

1. for empty
2. For use
3. Single element

complexity = $\theta(1)$

Q1 A[a, b, c, -, -, -, e, g] holding aqueue (circular) you do 2 enq() & 3 deq()
s. f = ? r = ?

⇒ [a, b, c, -, -, -, e, g]
1 2 3 4 5 6 7 8
r f
r = 5, f = 2

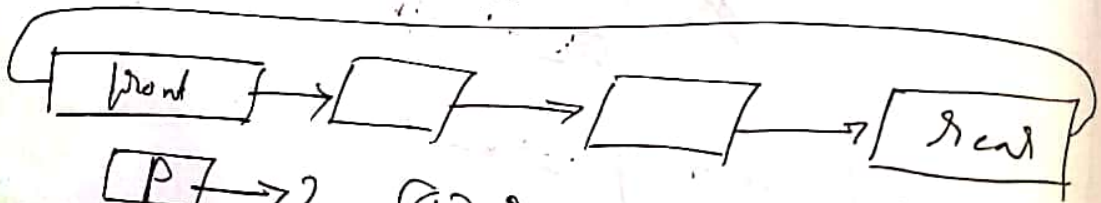
Other Functions :

- QueueFront :- It returns front element of the queue w/o changing the queue
- QueueRear :- It returns the rear element of queue without changing the queue
- IsEmptyQueue :- True when queue is empty
  False when false

Implementation using Linked List

- Enqueue — insertion at the end
  (as as we know the address of last node So Complexity $\theta(1)$)
- Dequeue — Deleting from front end
  Complexity $\theta(1)$

Q A circular linked list is used to represent a queue. A single variable P is used to access the queue. To which node should P point such that both the operation enQueue and deQueue can be performed in constant time ?



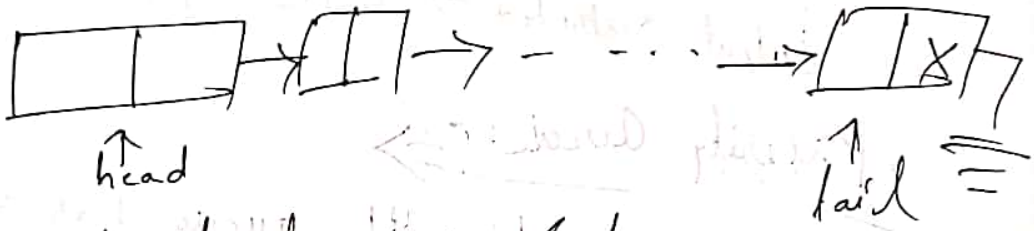
(a) Rear node (b) not possible

⇒ We cant get rear node using front node as we have to use traversing in order to get rear node.

We can get front node using rear without traversing.

→ (a) rear node

Q. A queue is implemented using a non circular singly linked list. Let n denote number of nodes. Let enque & deque be implemented by inserting a new node at the head and deletion a node from tail



time complexity of enque & dequeue

(a) $\theta(1)$ $\theta(1)$ (b) $\theta(1)$, $\theta(n)$

(c) $\theta(n)$ $\theta(1)$ (d) $\theta(n)$, $\theta(n)$

⇒ We can easily insert a new element at head with constant time $\theta(1)$

But in order to delete from tail we need the address of the previous node which can only possible by traversing

So it will take $\theta(n)$

→ (b)

Double Ended Queue:

in which insertion & deletion can be perform from both ends

two types

(i) Input restricted: input is restricted to standard end (Rear End)

(ii) output restricted – output is restricted at standard end (Front end)

	Enqueue	Dequeue
input restricted →	Rear	Both end
output restricted →	Both	Front

Priority Queue :⇒

Every element in the queue has associated priority & always highest priority element is deleted from the queue.

Implementation:

(i) Handling priority at deletion time:

enqueue → insertion at Rear end $O(1)$

dequeue → Search highest priority element & deleted $O(n)$

(ii) Handling Priority at insertion time:

Enqueue – insert the element based on its priority, such that the elements should be arranged in decreasing order of priority

complexity $O(n)$

Dequeue – Delete front end

complexity $\theta(1)$

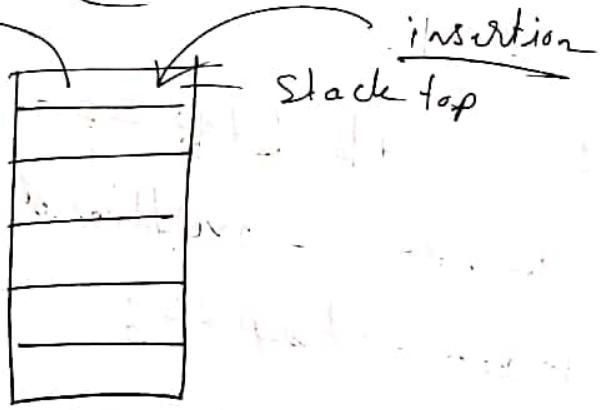
(x) To implement priority queue we need two queue

one queue for elements

another queue for priority

Stack

- A linear data structure in which insertion and deletion both are performed from the same end known as top of the stack
- It is LIFO (Last In First Out)

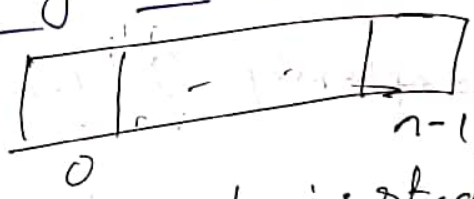


PUSH : Insertion in stack

POP : Deletion in stack

implementation using array:

n size array



top index points to last inserted element

(i) Stack empty :

When top points to -1

top = -1

- Sometime top = -1 is called bottom also (fixed index)

(ii) Stack Full :

if (top == n-1)

overflow & return

PUSH Algo :

```
PUSH (stack[], top, item)
{
1.  if (top == n-1)
        overflow & return
2.  top++;
3.  stack[top] = item;
```

overflow 1.

Complexity = $\theta(1)$

Pop Algo :

```
pop (stack[ ], top)
{
1. if (top == -1)
       underflow & return
2. item = stack[top]
3. top--;
}
```

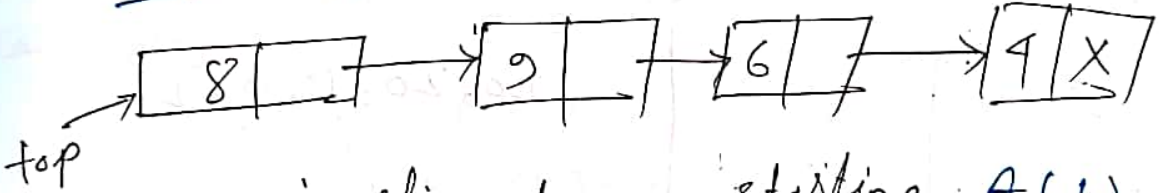
Stack empty

use

deletion

Complexity = $\theta(1)$ constant

Using Linked List :=>



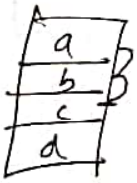
PUSH - insertion from starting $\theta(1)$

pop - Deletion from starting $\theta(1)$ constant

Other Functions :=>

(i) IsEmpty(stack) -> true when empty, false when not empty

(ii) PEEP (stack, top, i) -> Read & returns ith value from top



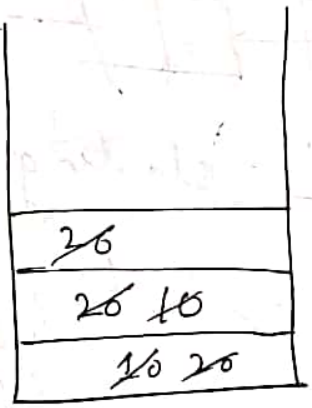
peep (stack, top, 3) -> c

(iii) CHANGE (stack, top, item) :- change the value of ith from top to item.

Q (1.) The following sequence of operations is performed on a stack
PUSH(10), PUSH(20), POP, PUSH(10), PUSH(20), POP, POP, POP, PUSH(20), POP
The sequence of values popped out is

(a) 20, 10, 20, 10, 20
(b) 20, 20, 10, 10, 20
(c) 10, 20, 20, 10, 20
(d) 20, 20, 10, 20, 10

⇒



20, 20, 10, 10, 20

→ (b) ✓

(2) A function f defined on stacks of integers satisfies the following properties
f(ϕ) = 0 and f(push(S, i)) = max(f(S), 0) + i
for all stacks S and integers i. If a stack S contains the integer 2, -3, 2, -1, 2 in order from bottom to top, what is f(S)

(a) 6 (b) 4 (c) 3 (d) 2

$f(S) = 0$

$f(PUSH(S,2)) = \max(0,0) + 2 = 2$

$f(PUSH(S,-3)) = \max(2,0) + (-3) = -1$

$f(PUSH(S,2)) = \max(-1,0) + 2 = 2$

$f(PUSH(S,-1)) = \max(2,0) + (-1) = 1$

$f(PUSH(S,2)) = \max(1,0) + 2 = 3$

→ (C) 3 ✓

Stack Permutation ⇒

If 3 value then there is $3! = 6$ combination

Stack Permutation means in what sequence we can pop out the values from the stack

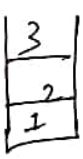
(We can pop out any value any time, it does not to be full everytime)

Case 1

3, 2, 1 ✓

Case 2

1, 2, 3 ✓

Case 6 – 

3, 1, 2 is not possible using Stack

(*) For n input sequence (↗ Catalan number)

Valid stack Permutation = $\frac{2n C_n}{n+1}$

invalid stack Permutation = $n! - \frac{2n C_n}{n+1}$

$^nC_r = \frac{n!}{r!(n-r)!}$

Q1 if the input sequence is 1, 2, 3, 4, 5 identify correct / wrong sequence of permutation as given below

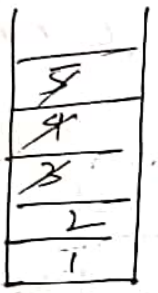
(i) 5, 4, 3, 1, 2 (ii) 4, 5, 3, 2, 1

(iii) 3, 2, 1, 5, 4 (iv) 2, 3, 5, 4, 1

(v) 4, 3, 2, 5, 1

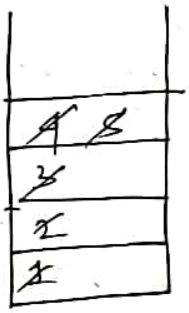
⇒ (i) Here we have to remember in order to remove any value first we have to insert it. And there is no restriction / mandatory that we have to push all elements before popingout

(i)



5, 4, 3. ✗ we cant pop 1 before 2

(ii)



4, 5, 3, 2, 1 ✓ possible

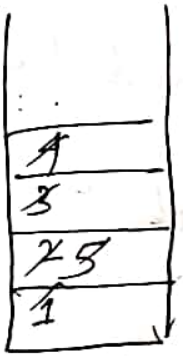
(iii)

3, 2, 1, 5, 4 ✓

(iv)

2, 3, 5, 4, 1 ✓

(v)



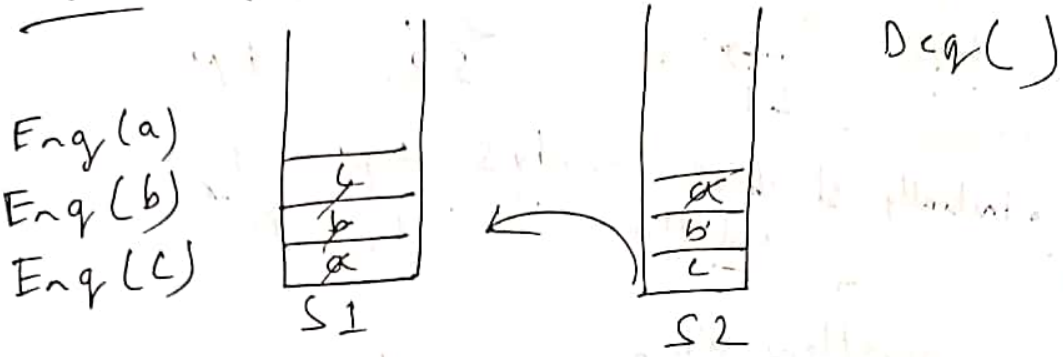
4, 3, 2, 5, 1 ✓

implementation of Queue using stack :- ⇒

(*) minimum number of stack is 2 to implement a queue.

two way -

(i) Regular method :- ⇒

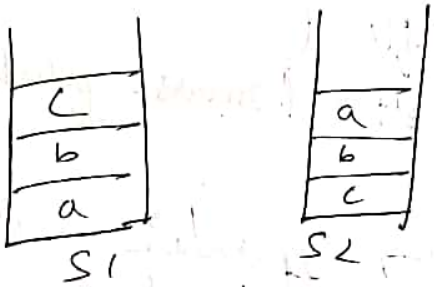


Enqueue → blindly insert in stack S1

Dequeue → Bring all elements from S1 to S2 then pop from S2 & bring back remaining elements from S2 to S1.

Disadv — no of PUSH & POP are a lot

(ii) Efficient method :-



Enqueue — blindly insert in S1

Dequeue — bring all the elements from S1 to S2 & pop from S2 . keep all the elements in S2

. if S2 is not empty pop from S2

. if S2 is empty bring all elements from S1 to S2 & pop .

Multiple Stacks in single Array.

For 2 stack:

start stack from 2 different ends

- initially stack are empty so top1 = -1 (underflow), top2 = n
- overflow when top2 = top1 + 1

Stack Expression & Evaluation

Expression are just representation of mathematical equations. There are 3 type –

(i) Prefix (polish) / warsaw
(ii) Infix
(iii) Postfix (reverse polish)

Prefix → operator is placed before both the operands

Infix → Operator is placed between both of operands

Postfix → Operator is placed after the operands.

Ex– (a+b) – infix
+ab – Prefix
ab+ – Postfix

Operator Precedence

() → ↑ or ^ → * % / → + or − → =

High to low

if two operator having same precedence then we have to check associativity

2 − 1 − 5 ⇒ 2 − (1 − 5) = 2 − (−4) = 6 ✗

(2 − 1) − 5 = 1 − 5 = −4 ✓

Exponent, Assignment, Unary, Conditional has Right to left others have left to right

- $2^{3^2} = 2^9 = 512$

  ⇓

  2 ↑ 3 ↑ 2 (Right to left)

- a = 5, b = 10, c = 15

  a = b = c

  ⇝

- All unary operator have higher precedence over binary operator

  unary means one operand, binary means two operand.

(i) 2 + 3 * 5 (infix)

⇒ Prefix

+ 2 * 3 5

single operand

postfix

3 5 *

2 3 5 * +

(ii) $2+5*3\uparrow 2+9$

Prefix

$\Rightarrow 2+5*\uparrow 32+9$

$\Rightarrow 2+*5\uparrow 32+9$

left to right

$\Rightarrow +2*5\uparrow 32+9$

$\Rightarrow ++2*5\uparrow 329$

postfix

$2+5*32\uparrow +9$

$= 2+532\uparrow *+9$

$= 2532\uparrow *+9+$

always we will follow associativity

(iii)

	Prefix	Postfix
$-b \rightarrow$	$-b$	$b-$
$\log x \rightarrow$	$\log x$	$x\log$
$x! \rightarrow$	$!x$	$x!$
$\log x! \rightarrow$	$\log !x$	$x!\log$

Note → There is no parentheses in prefix or postfix it just say to evaluate lower precedence if it is under parentheses (in infix)

$a+b*c$	$(a+b)*c$
Pre $-+a*bc$	Pre $*+abc$
post $-abc*+$	post $-ab+c*$

Postfix to infix :⇒

(i) 2532↑*+9+

⇒ 25(3↑2)*+9+

⇒ 2(5*(3↑2))+9+ — we will try to remove the inner bracket. but * ~~is not of same precedence~~ we can remove if only 3↑2 can solved first always ↑ has higher precedence so can remove it.

⇒ 2+(5*3↑2)9+

⇒ (2+(5*3↑2)9+

⇒ 2+5*3↑2+9 ✓

(ii) abc*+de/f*− — Scan from left to right

⇒ a(b*c)+de/f*−

⇒ (a+(b*c))de/f*−

⇒ (a+b*c)de/f*−

⇒ (a+b*c)(d/e)f*−

⇒ (a+b*c)((d/e)*f)−

⇒ (a+b*c)(d/e*f)−

⇒ (a+b*c)−(d/e*f)

⇒ a+b*c−d/e*f ✓

Prefix to infix

(i) $-+9/4\uparrow238$ We scan right to left.

$\Rightarrow -+9/4(2\uparrow3)8$

$\Rightarrow -+9(4/(2\uparrow3))8$

$\Rightarrow -(9+(4/2\uparrow3))8$

$\Rightarrow 9+4/2\uparrow3-8$ ✓

(ii) $+-6\uparrow2*37$

$\Rightarrow +-6\uparrow2(3*7)$

$\Rightarrow +-6(2\uparrow(3*7))$

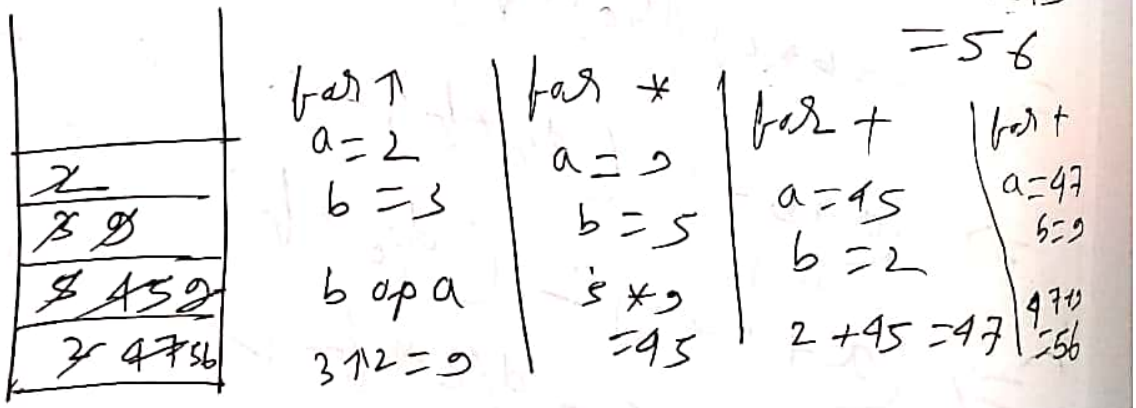
$\Rightarrow +(6-(2\uparrow(3*7)))$

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

Evaluation of postfix Notation using Stack

$25232\uparrow*+9+ \rightarrow 2+5*3\uparrow2+9$

$=56$



Algo is

(i) add right parenthesis) at the end of P

(ii) Scan P from left to right until) is encountered

(i) if an operand is encounter Push on stack

(ii) if an operator is encounter

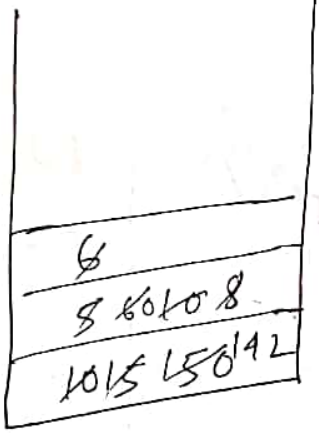
(a) pop first two elements as a, b

(b) evaluate bop a & push result on stack

(iii) Set the result = top of stack.

Q] The result evaluating the postfix expression 10, 5, +, 60, 6, /, *, 8, − is

⇒



for +	for /	for *	for −
a = 10	a = 6	a = 10	a = 8
b = 5	b = 60	b = 15	b = 150
10+5	60/6	15*10	150−8
= 15	= 10	= 150	= 142

Q] The following postfix expression is evaluated using stack

8 2 3 ^ / 2 3 * + 5 1 * −

top element of the stack after the first * is evaluated are

(a) 6, 1 (c) 3, 2

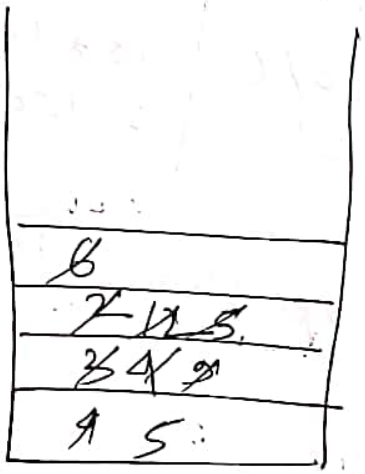
(b) 5, 7 (d) 1, 5

Evaluation of Prefix using Stack

Algo:-

1. Add right parentheses (at the start of P
2. Scan P from right to left until (is encountered
   (i) If an operand is encountered PUSH onto the stack
   (ii) if an operator is encountered
      A. Pop first two element as a & b
      B. evaluate a op b & push result on stack
   (iii) Set the result = top of stack

ex- − + 5 / * 6 2 3 4



for *
a=6
b=2
6*2 = 12

for /
a=12
b=3
12/3 = 4

for +
a=5
b=4
5+4=9

for −
a=9
b=4
9−4=5

Recursion :⇒

Recursion also done with the help of stack

- When a function call itself is called Recursion we use Activation Record

Characteristic

(i) it should have a base/exit/termination condition for which the function doesn't call itself

(ii) Every time the function calls itself, the base condition should come close

```
cx-
fun(int n)
{ if(n==0)
    return;
  print("%d", n);
  fun(n+1);
}
```

⇒ ✗ this is not a good design as its not coming close to the base condition.

Types - (i) direct Recursion : if recursion call is in the body of the same function

(a) tail Recursion → Recursive call is at the end of the function. / last statement

(b) Head Recursion — is not last statement

```
fun()
{
  ...
  fun();
  ...
}
```

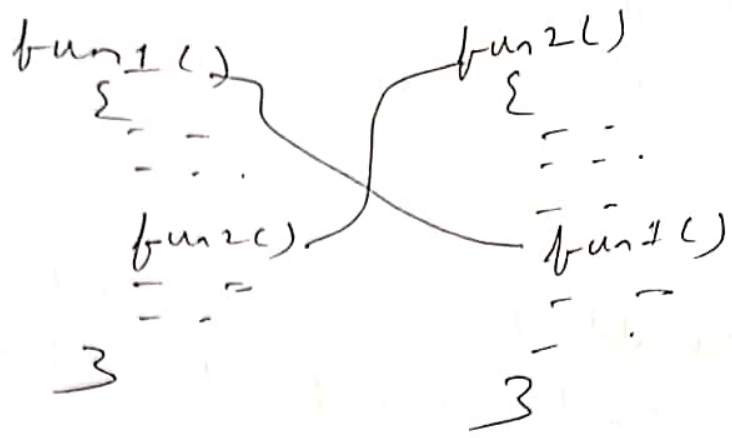
tail

```
fun()
{
  ...
  fun();
}
```

Head

```
fun()
{
  ...
  fun();
  ...
}
```

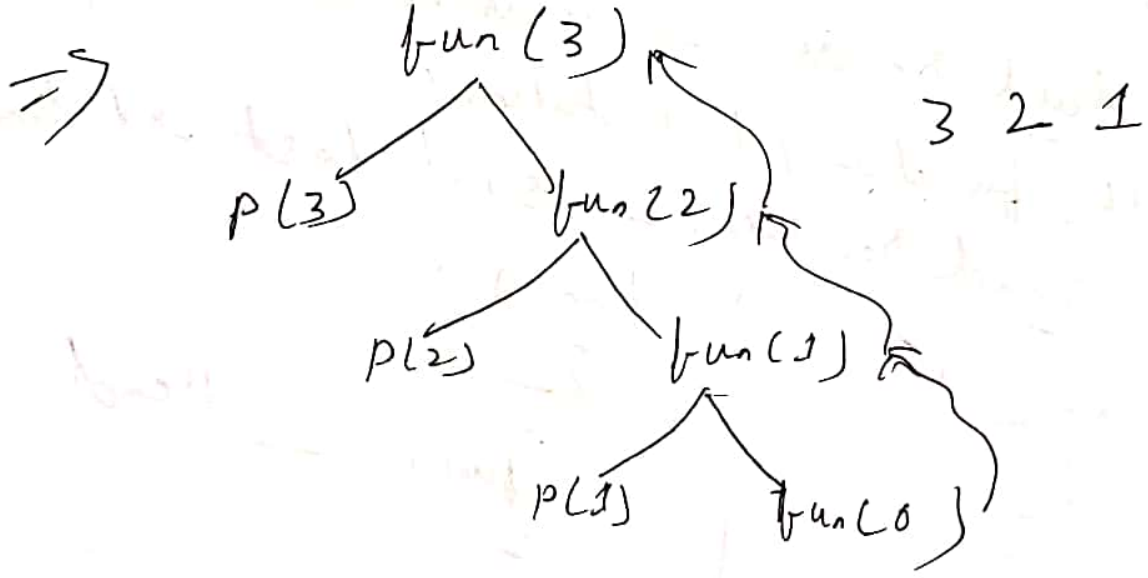
(ii) Indirect recursion:



(iii) Nested Recursion: fun(a, fun(b, c))

Q1

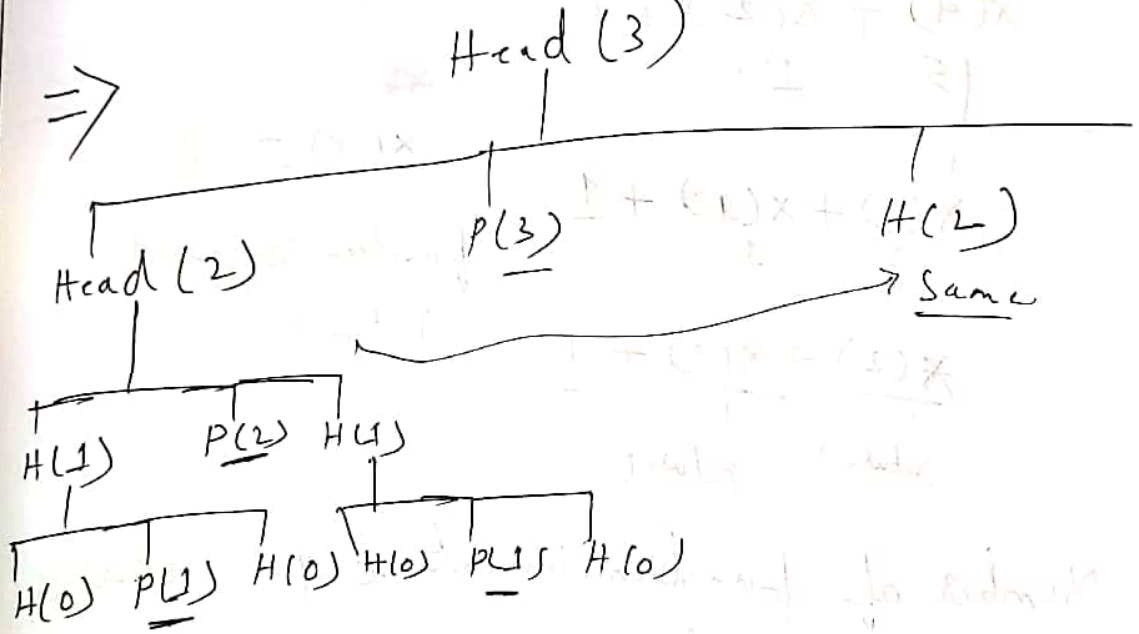
```
Void fun (int x)
{
  if (x>0)
  {
    printf("%d", x);
    fun (x-1);
  }
}
Void main()
{
  fun(3);
}
```



2| Void Head (int n)

```
{ if(n>0)
  { Head (n-1);
    print ("%d", n);
    Head (n-1);
  }
}
Void main()
{ Head (3);
}
```

⇒



1 2 1 3 1 2 1

(*) Function call

printf ("%d", sqrt(16))

printf ("%d", 4);

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

⇒ Return value of X(5) is

X(5)

↓

X(4) + X(2) + 1
(5, 1)

X(3) + X(1) + 1
(3, 1)

X(2) + X(0) + 1
(1, 1)

return 1 return 1

X(5) = 7

function is called also 7 times

Number of function invocations for X(X(5)) is

→ X(5) = 7

X(7) — return X(6) + X(4) + 1
17 (11, 5)

return X(5) + X(3) + 1
(7, 3)

for X(5) the function calls = 7

for X(7) " " " = 17

Tower OF HANOI

```
Void TOH ( N, start, AUX, End )
{
    if (N>0)
    {
        TOH ( N-1, start, End, AUX);
        Move largest disk from start to End;
        TOH (N-1, AUX, start, End)
    }
}
```

Complexity = $O(2^n)$

3

3

a] if n=3 disk

(i) Disk moves = $7 \left(2^n - 1\right)$

(ii) total function call = $15 \left(2^{n+1} - 1\right)$

(iii) After how many function call first move of the disk made = $n+1$

(iv) function call before last move is made = $\left(2^{n+1} - 2\right)$

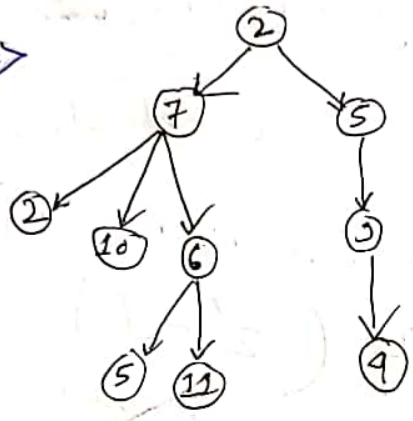
Tree

- Nonlinear hierarchical data structure that consists of nodes connected by edges.

nonlinear means not in sequential order, multiple level are involved, cannot traversed in a single run.

- If there is (n-1) edges then n nodes.

Terms :=>



(i) Root :- the very first node of a tree
ex- (2)

(ii) Edge :- connecting link between two nodes

(iii) Parent :- The node which has atleast one branch from it. ex - (7), (5), (6), (3)

(iv) child :- The node which is descended of some node. ex - (7), (5), (2), (10), (6) .....

(v) Sibling :- Nodes which belong to same node.
ex - (2), (10), (6)

(vi) Degree :- total no of child node it have

(vii) Internal node — node which has atleast one child.

(viii) External/Leaf node :- Node which has no

(ix) ~~Level~~:

Subtree :- Every child node forms a subtree on its parent node.

(*) N = total no of nodes
L = total no of Leaf nodes
I = total no of internal nodes

$$N = L + I$$

[a] Can we have a node in tree which can be internal node & leaf node both?
=> NO

[a] Can we have a node which is neither leaf nor internal node?
=> NO

Parenthesis Representation :=>

root (child1, child2, .. childn) → From left to right separated by comma

If child also have children then we have to make them root also.

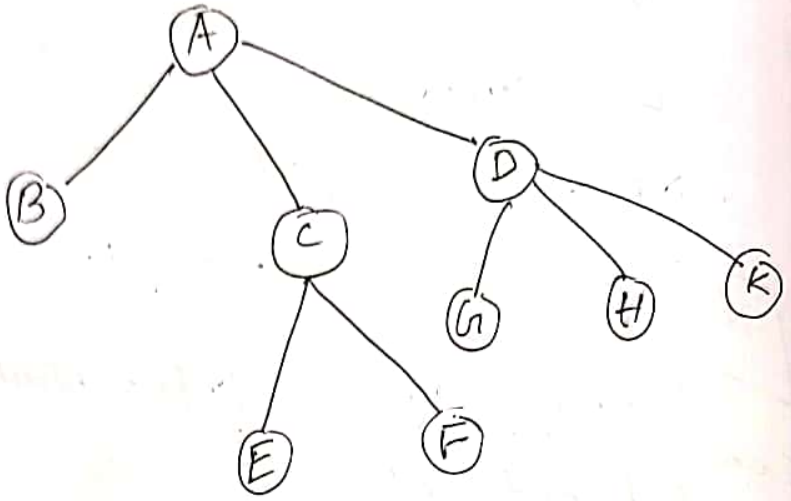
Previous figure representation will be

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

Q] Construct the tree for following parenthesis representation

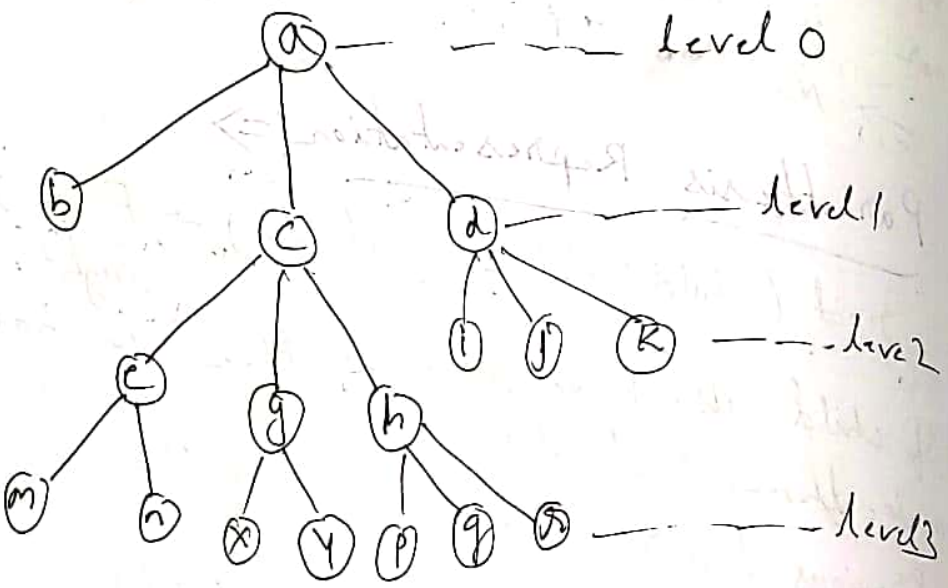
A (B, C (E, F), D (G, H, K))

⇒



Q] a (b, c (d (m, n), g (x, y), h (p, q, r)), d (i, j, k))

⇒



Levels in tree: each step from top to bottom is called as level of a tree.

- The level count start with 0 and increment by 1 each step

Depth :⇒ associated with a node

Depth is its level no

Depth of a root node is 0.

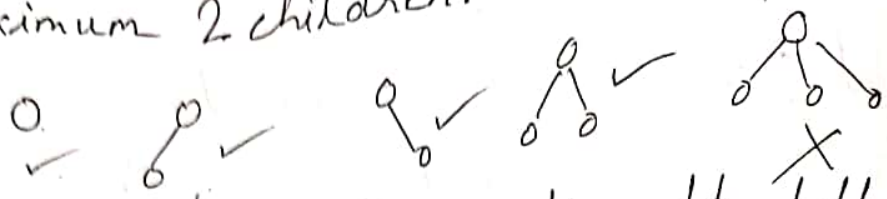
Height :⇒ associated with tree

There is 2 defination

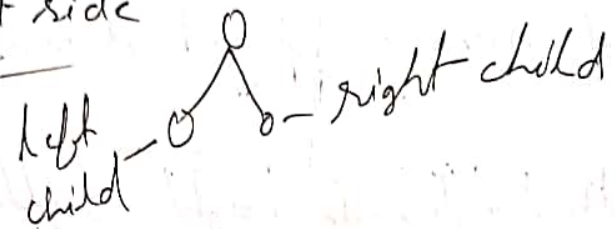
Defination 1	Defination 2
H = Max level number	Max level number + 1
height with single node 0	height with single node 1
height of empty tree −1	Height of empty tree 0
Height is number of edges from root to the farthest leaf node	number of nodes in the path from root to the farthest leaf node.

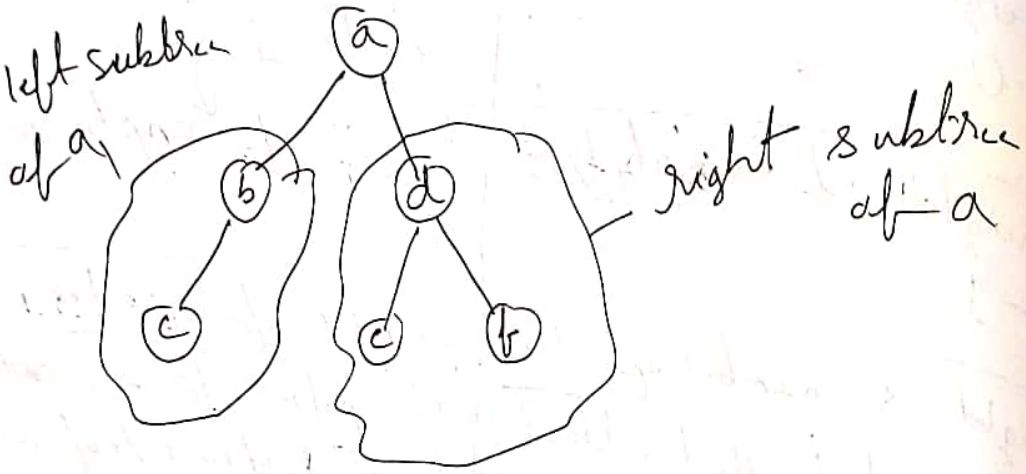
Binary Tree

A tree in which each node can have maximum 2 children.

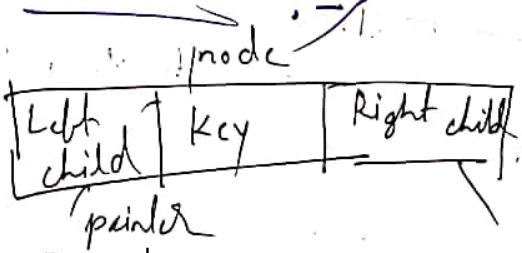


- We should also maintain the left side or right side



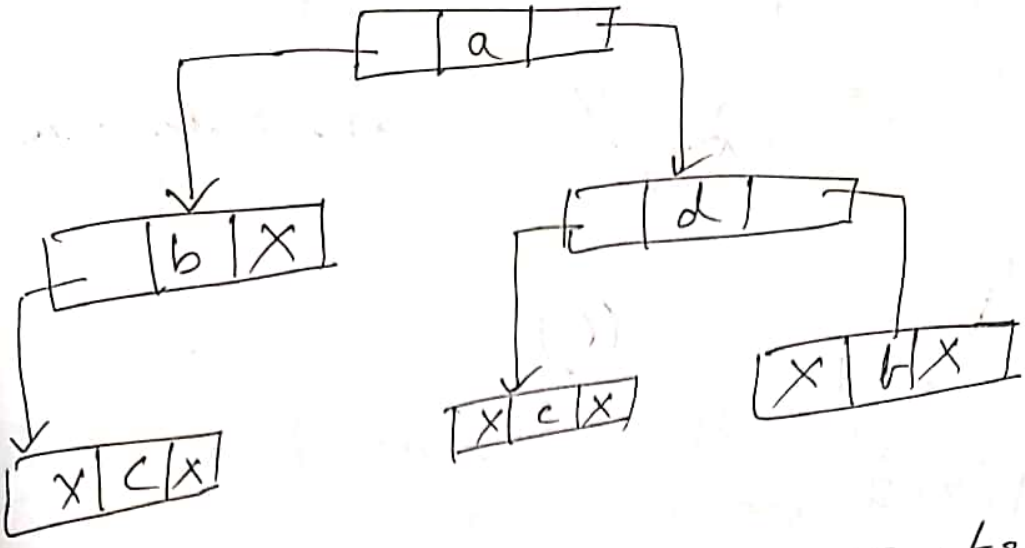


Linked Representation



Self Referential structure

```
struct BTnode
{ char key;
  struct BTnode *leftchild;
  struct BTnode *Rightchild;
};
```



(*) In general tree there is no restriction of number of child thats why general tree can not be represented like this.

Detecting a leaf node :=> if t is pointing a node

if ((t->Lchild == NULL) && (t->Rchild == NULL))

then t is pointing a leaf node

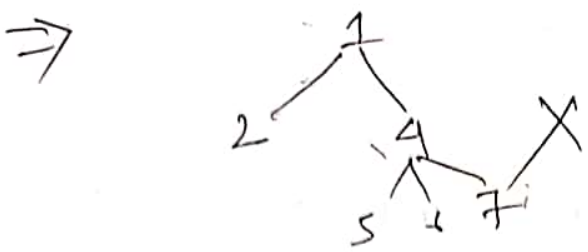
Q] (x y z) indicates y & z are subtree of node x. Which is valid binary tree

a) (1 2 (4 5 6 7))

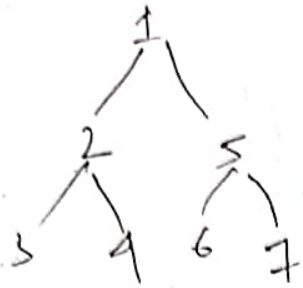
b) (1 (2 3 4) 5 6) 7)

c) (1 (2 3 4) (5 6 7))

d) (1 (2 3 NULL) (4 5))

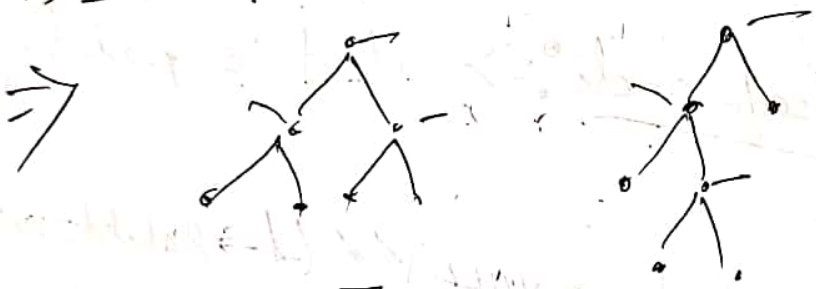


1
2 5 9 – can't be root
X



(L)

Q| A binary tree T has 4 leaves the number of nodes in T having 2 children is 3 ?



Ans - is 3

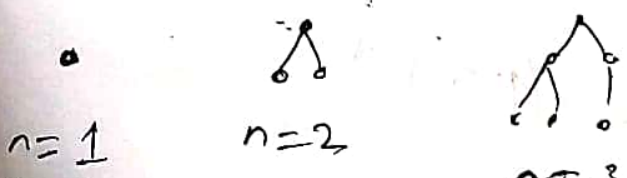
(X) But if there is a large number of leaves then its quite difficult to get

Handshaking Lemma ⇒ if total leaf node is n then internal node with 2 children is (n-1)

internal.

$L = I_2 + 1$

leaf node



n=1 I=0

n=2 I=1

n=3

(*) For binary tree L = Leaf node

$N = L + I$ I_2 — internal node.

$= 2I_2 + 1 + I_2 + I_1$ I_2 - internal nodes with 2 child

$= 2I_2 + 1 + I_1$

Q] In a binary tree, the number of internal nodes of degree 1 is 9 and number of internal node of degree 2 is 18. Total number of nodes in the binary tree is?

$\Rightarrow I_1 = 9 \quad I_2 = 18$

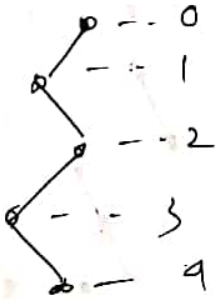
$\therefore N = L + I = 2I_2 + 1 + I_2 + I_1$

$= 2 \times 18 + 9 + 1$

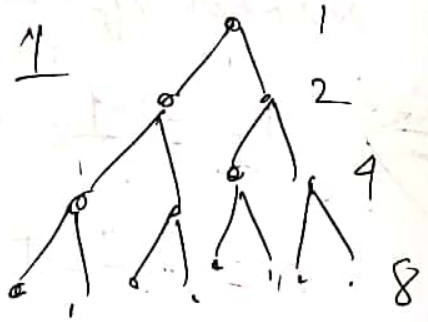
$= 46$

Q] Minimum & maximum number of nodes in a binary tree on a level number L

$\Rightarrow$



Minimum = 1

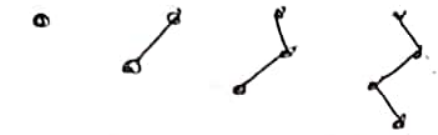


Maximum = 2^L

Q] Minimum and Maximum number of nodes in a binary tree of height H

(H of tree with single node 0)

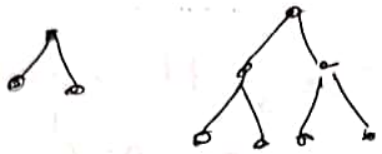
$\Rightarrow$



min number of nodes = H + 1

H =	0	1	2	3
$n_{min\ nodes}$ =	1	2	3	4

$$max = 2^{H+1} - 1$$



H =	0	1	2	3	4
n_{max} =	1	3	7	15	31
	$2^{1}-1$	$2^{2}-1$	$2^{3}-1$		

Q) There is a binary tree with n nodes. The root is only node in the tree having odd no of children.
Number of leaf in tree = ?

$\Rightarrow I_1 = 1$

L = 1

$n = 2I_2 + I_1 + 1$

$= 2(L-1) + 1 + 1$

$= 2L$

L = 2

L = 2

L = 3

or $L = n/2$

Q) How many binary tree can be constructed using 3 unlabelled nodes

⇒  = 5

In general for n — unlabelled node

no of binary Tree possible = $\frac{2nC_n}{n+1}$

Catalan number

Q) How many binary tree can be constructed using 3 distinct keys

⇒ no of tree with unlabelled node

$= \frac{6C_3}{3+1} = 5$

there is 3! = 6 possibilities for labelling

So Answer will be = $n! \times \frac{2nC_n}{n+1}$

$= 3! \times \frac{6C_3}{3+1} = 30$

Tree Traversal

recursively

(i) Preorder :- NLR

(ii) Inorder :- LNR

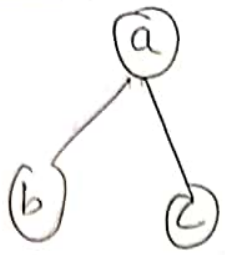
(iii) Postorder - LRN

N = node

L = Left subtree

R = Right subtree.

ex



Preorder - a b c
Inorder - b a c
postorder - b c a

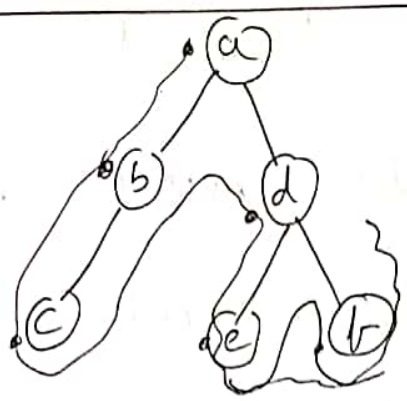
Tree traversal Preorder

```
void preorder (struct BTNode *t)
{
    if (t)
    {
        printf ("%d", t->data);
        preorder (t->Leftchild);
        preorder (t->Rightchild);
    }
}
```

for inorder -

```
void Inorder (struct BTnode *t)
{
    if (t)
    {
        Inorder (t->Leftchild);
        printf ("%d", t->data);
        Inorder (t->Rightchild);
    }
}
```

ex



ⓐ - postorder

ⓐ inorder

preorder — a b c d e f

Inorder — c b a e d f

postorder — c b e f d a

Observation —

(i) first symbol of the preorder traversal is always root.

(ii) Last symbol of the post order traversal is always root.

Q] identify the inorder, preorder, Postorder traversal

1. G H F E A P K D L B X M

2. E G F H A P M L K D X B

3. M P A E F G H X L D K B

⇒ - - - M → postorder

- - - M - - - → inorder

M - - - - - - - → preorder

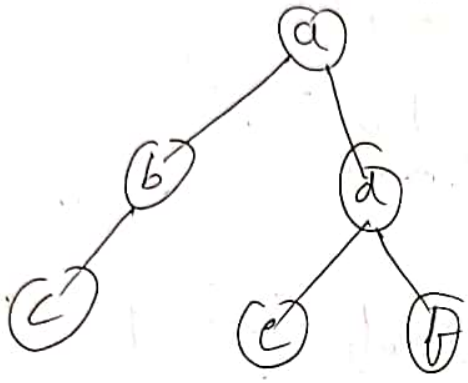
Converse order traversal recursively

(i) Converse preorder — NRL

(ii) Converse inorder — RNL

(iii) Converse Postorder — RLN

Ex



Converse preorder — a d f e b c

converse inorder — f d e a b c

Converse postorder — f e d c b a

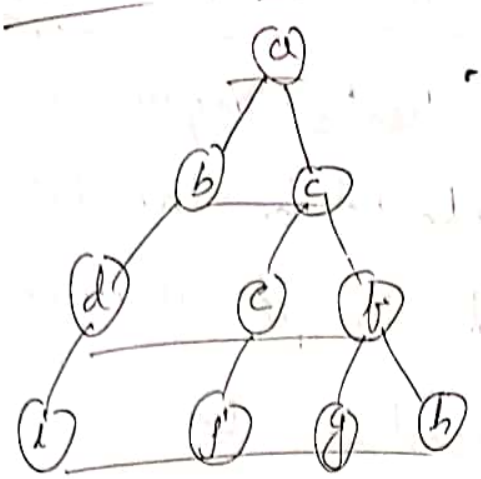
(X) Conventional preorder = Reverse of converse postorder

(X) Conventional postorder = Reverse of converse preorder

(X) convention inorder = Reverse of converse inorder

Vice versa is also same

Level order Traversal :=>



- We traverse each node level by level

All the nodes are traversed from left to right (standard)

Standard a b c d e f i j g h

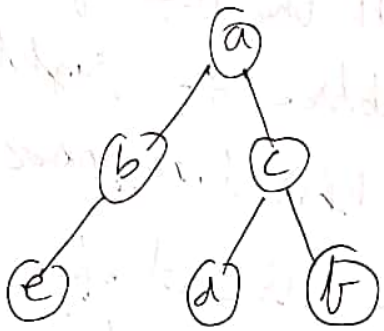
(x) But we can also traverse each node in any level in any possible sequence

a, c, b, e, d, f, i, j, g, h ✓

Double order Traversal :=>

every node is traversed 2 times

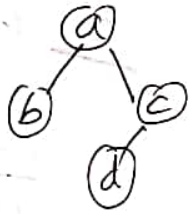
n L R n



a b e e b c d d f f c a

Triple order Traversal - node is traversed 3 times n L n R n

a b b b a c d d d c c a



Construction of Binary Tree :→

Minimum 2 traversals required and one traversal should be inorder traversal. it will be needed to get unique Binary Tree

preorder, inorder ✓

preorder, postorder ✗

postorder, inorder ✓

We will use

Preorder or postorder → root of the tree

Inorder → Left & right subtree

(i) first pick up root from pre or post order

(ii) with the help of inorder classify the left subtree & right subtree

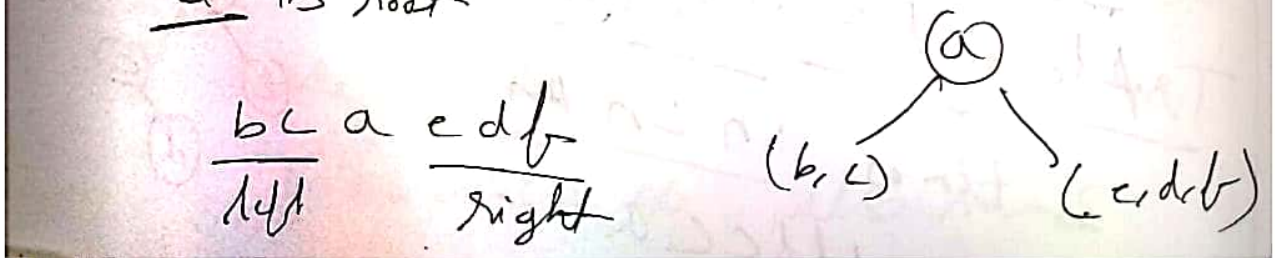
& Do this two process recursively.

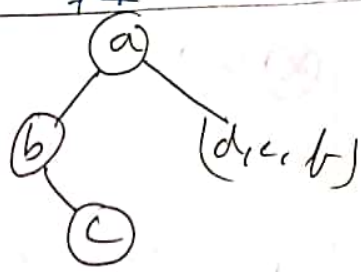
ex Pre — a b c d e f

In — b c a e d f

⇒ We know in preorder first node is root

a is root

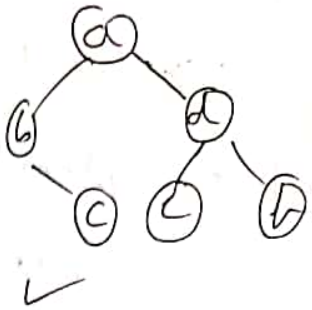




we will do same on b c

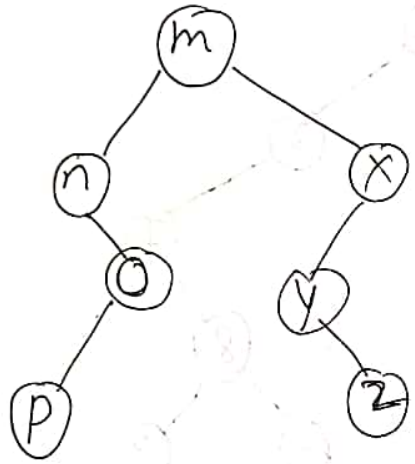
in b c - b is root

- in inorder there is b c & c is right side of b so c will be right child of b
- in d c f - d is the root
- in c d f (in) c is left & f is right



ex 2] pre - m n o p x y z

in - n p o m y z x

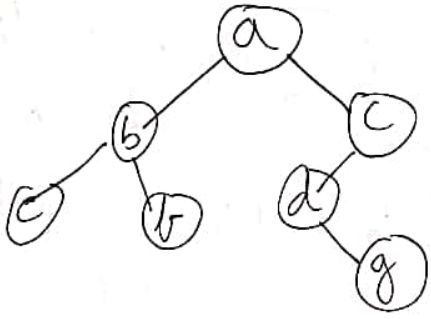


ex-3] post - c f b g d c a

in - e b f a d g c

=> in postorder last element is root

else all same like pre



Ex-4: Converse postorder – 9, 10, 8, 7, 3, 6, 5, 9, 2, 1

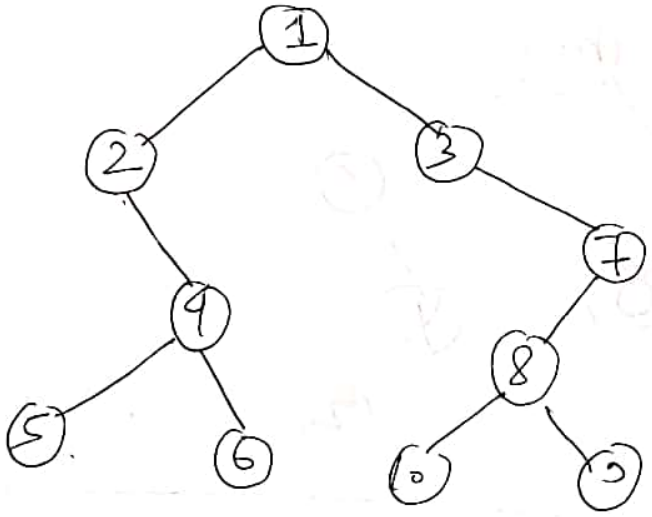
Converse inorder – 7, 3, 8, 10, 3, 1, 6, 9, 5, 2

⇒ We know reverse of converse postorder is preorder

reverse of converse inorder is inorder

∴ Preorder – 1, 2, 9, 5, 6, 3, 7, 8, 10, 3

∴ inorder – 2, 5, 9, 6, 1, 3, 10, 8, 3, 7



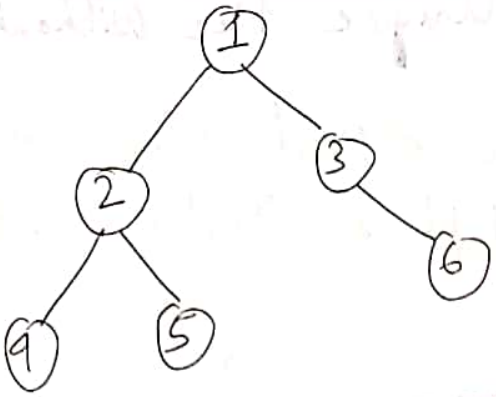
Ex-5: Preorder – 1 2 4 5 3 6

Right pointer

⇒ Right pointer denote the right child of the node.

b is the left child of a

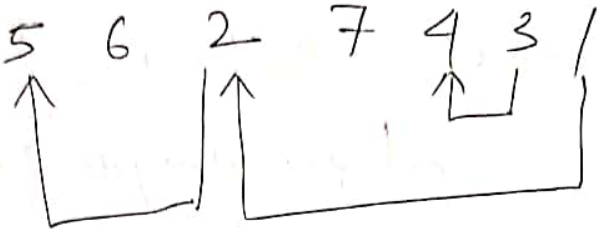
the b is left child of a as it has no arrow



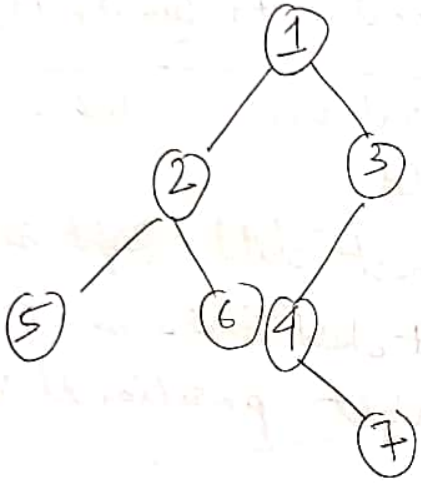
Root | L Subtr | R Subtr..

Ex-6 Postorder — 5 6 2 7 4 3 1

Left pointer



⇒



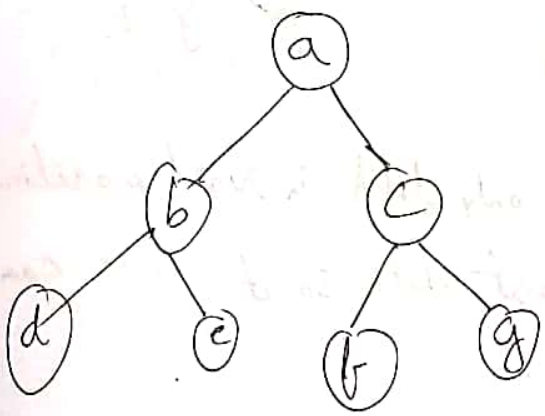
as there is nothing between 3 4 & 7 is after 4 so 7 will be right child of 4

Ex-7 Postorder d e b f g c a

Degree 0 0 2 0 0 2 2

⇒ Degree means number of children

⇒



* Why not unique tree without Inorder →

⇒ Preorder & Postorder can not identify left & right subtree

Pre – a, b, c

Post – c, b, a

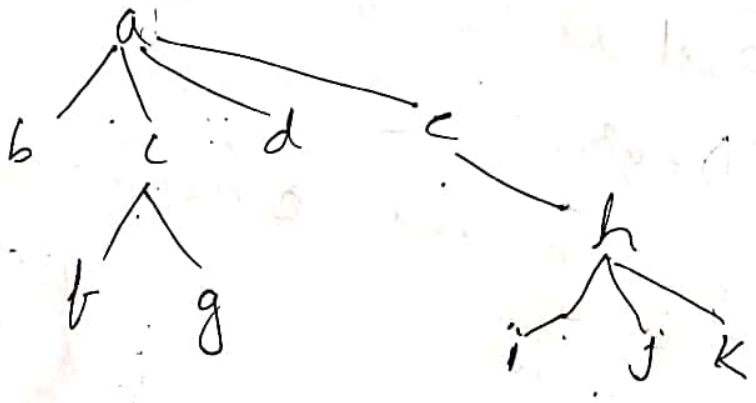
• can not give unique result

Conversion from general to binary Tree ⇒

We can't represent general tree in linked representation as it has no fixed child.

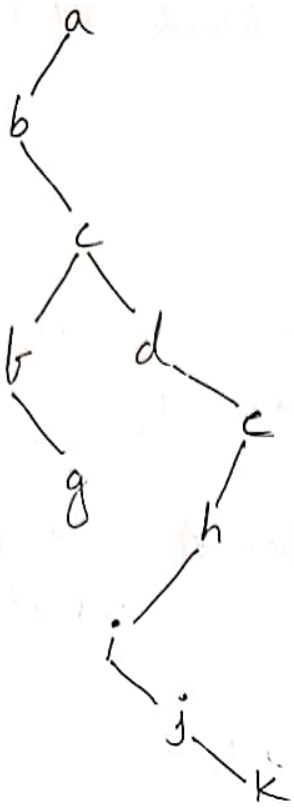
* It is called Leftmost child Right sibling representation (in the left child there will be its left most child & in right position there will be its right sibling)

ex –



•  h is the only child in right position but is still its left most child so it will come in left child part

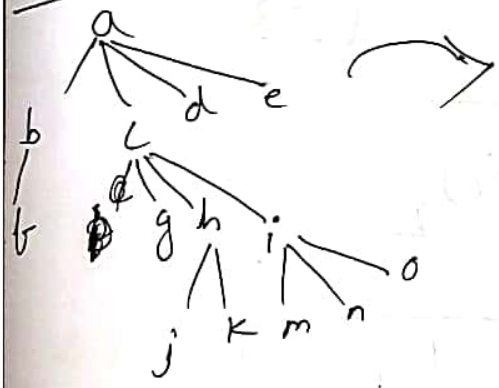
equivalent binary tree will be

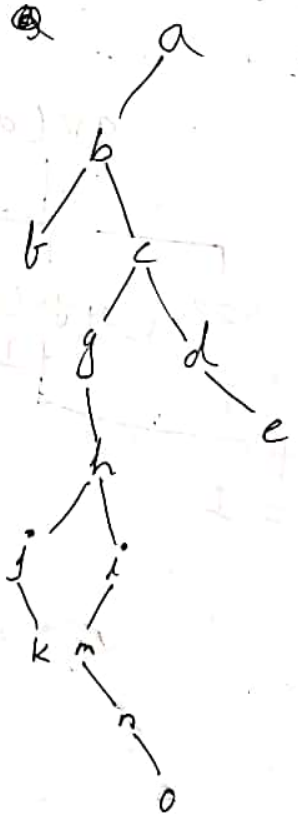



```
struct GBTnode
{ char data;
  struct GBTnode *Leftmost child;
  struct GBTnode * Right Sibling;
};
```

in this representing P->Leftmostchild == null then P points to leaf node (if it has no left child then it is no a leaf node)

Q] Convert the tree -

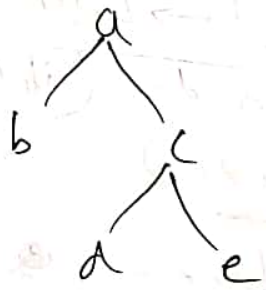




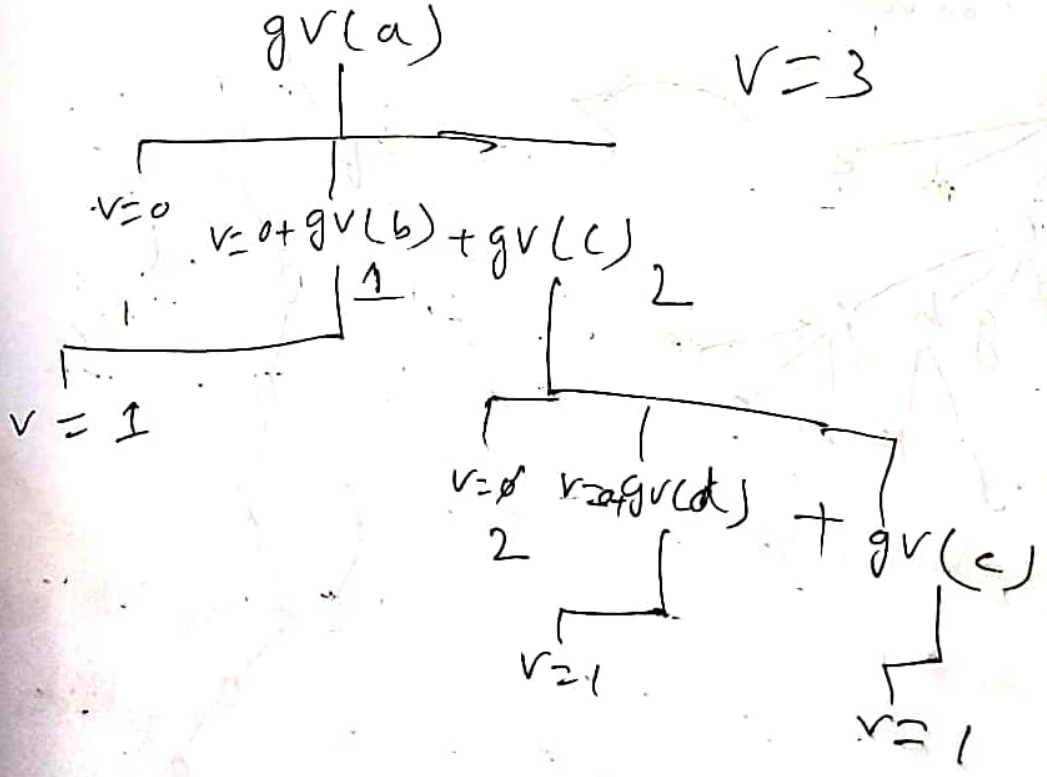
a) What does it return on BT

```
int getvalue (struct BTNode *t)
{
  int value=0;
  if (t)
  {
    if ((t->LC)==NULL) && (t->RC==NULL)
      value = 1;
    else
      value += value + Getvalue(t->LC)
               + Getvalue (t-> RC);
  }
  return value;
}
```

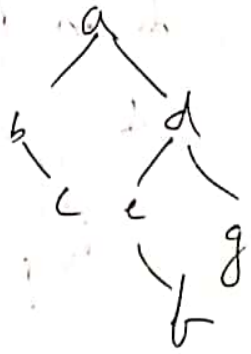
⇒



Ans- Number of leaf node

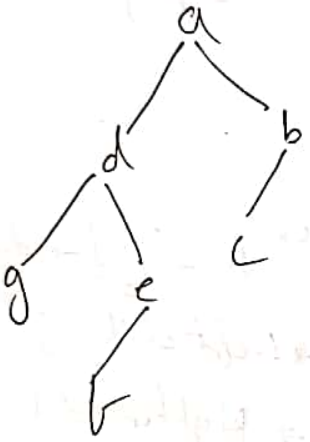


a) Void Do (struct btnode *t)




```
{
  if (t)
  {
    Do(t->LC);
    Do(t->RC);
    swap(t->LC, t->RC);
  }
}
```

⇒ it is swapping the nodes recursively



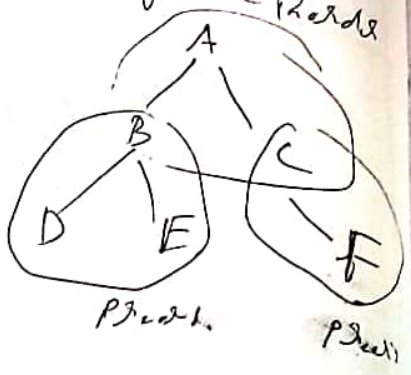
Height calculation in BT

We will recursively calculate height of Left & Right subtree & ~~calc~~ return the maximum.

```
int H(struct BTnode *t)
{
  if (!t)          (when t is empty
    return 0;       tree height is 0)
  int LT = H(t->LC);
  int RT = H(t->RC);
  if (LT>RT) return LT+1; else RT+1;
}
```

Q. Consider following function value of m(t) preorder

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



inorder

preorder

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

postorder

⇒ BDE ACF

Expression Tree :⇒

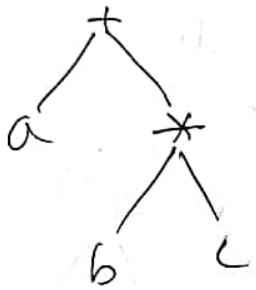
operater – node

operand – children
- Left operand is left child
- Right operand is Right child

* Here also we use precedence & associativity rule

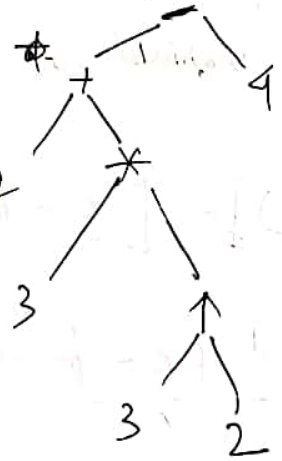
a+b →

a+b*c →

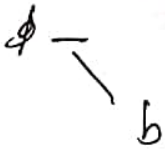


a+[b*c]

2+7*3↑2 – 4 →



–b →

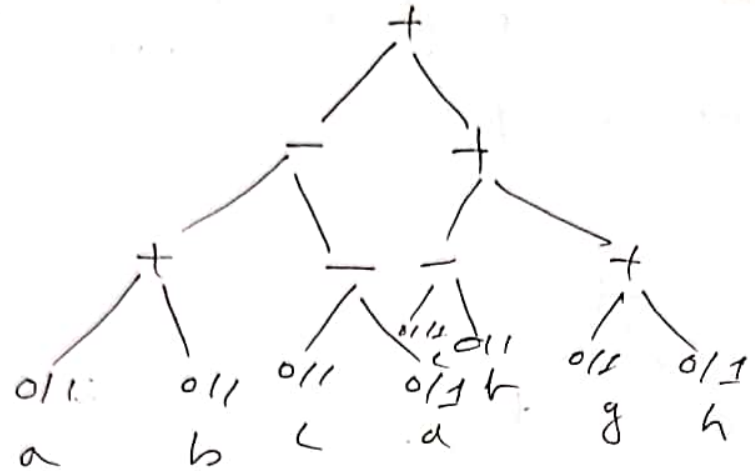


log x! –

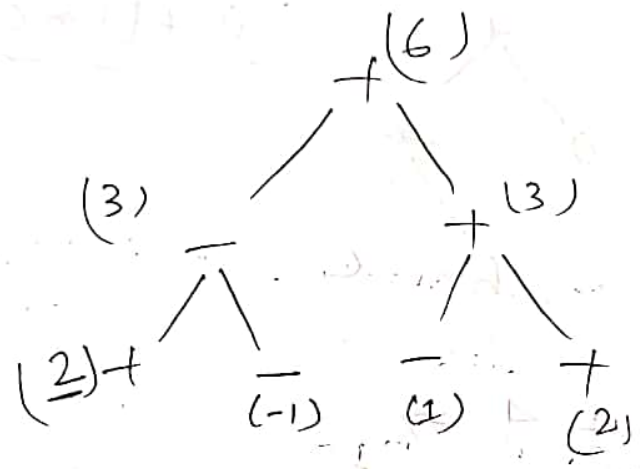
log n –

x! –

Q) Consider the expression tree. Each leaf can be either 0/1. over all all possible choices the maximum possible value is ____



⇒



→ 6

Shortcut is take variable in leaf node

$$[(a+b)-(c-d)]+[(e-f)+(g+h)]$$

$$= a+b-c+d+e-f+g+h$$

all +ve = 1
all -ve = 0

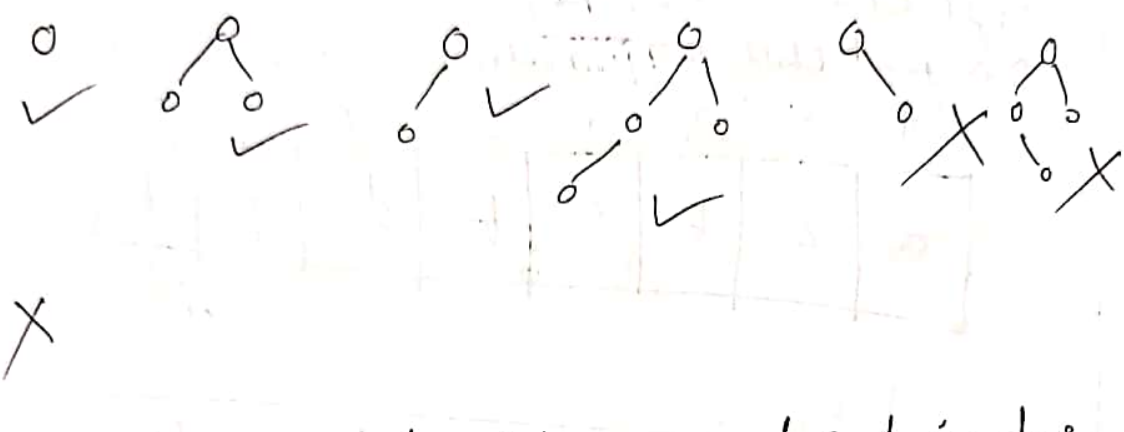
$$= 6\times 1 - 2\times 0 = 6$$

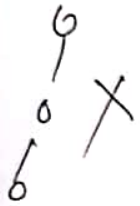
min value $= 6\times 0 - 2\times 1$ ∵ all +ve = 0
all -ve = 1

$$= -2$$

∴ Range = $[-2, 6]$

Complete Binary tree :→ Binary tree in which all the level has maximum number of nodes. Last level may not have maximum number of nodes & all nodes in last node arranged left to right.



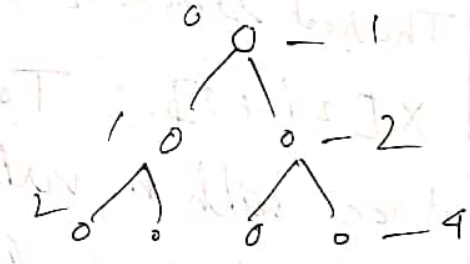


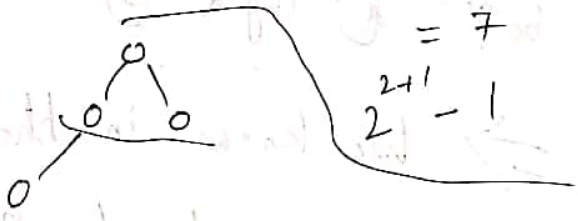
Q] Maximum and minimum number of nodes in a complete binary tree of height h?
(Hof tree with single node is 0)

⇒ $max = 2^{h+1} - 1$

$min = 2^h - 1 + 1$

$= 2^h$





Strictly complete Binary tree : Every level (including last) has maximum number of nodes.

Array Representation of CBT (Complete Binary) =>

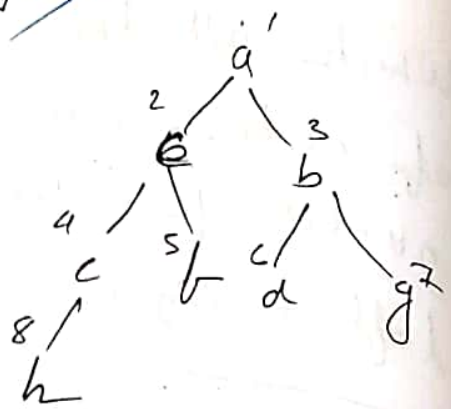
Root = index 1

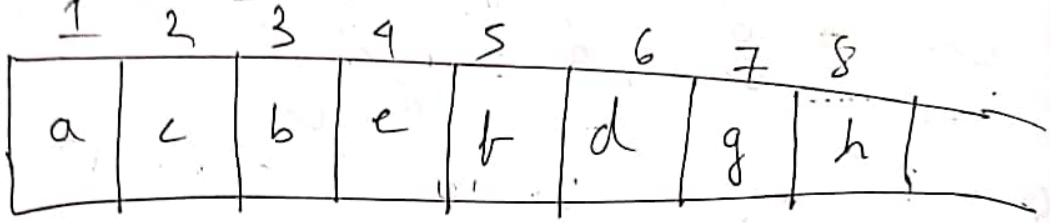
if there is a node in index i

Left child = index $2i$

Right child = index $2i+1$

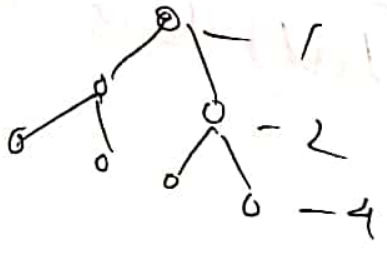
Parent = $\lfloor \frac{\text{Child index}}{2} \rfloor$ floor value





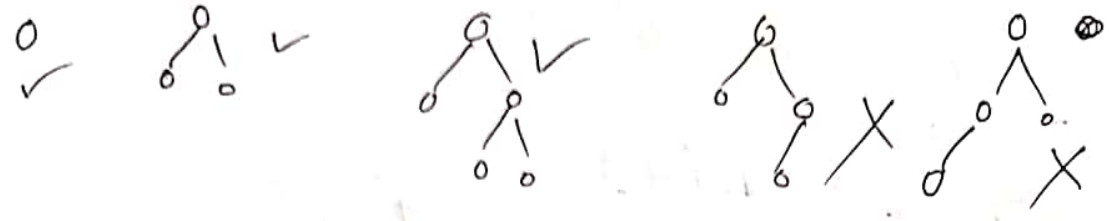
Q] A scheme for storing binary tree in an array X is as follows. indexing of X starts at 1. The root stored in X[1]. Left child in X[2i] & Right child X[2i+1]. To be able to store a binary tree with n vertices, minimum size of X should be? (a) $\log n$ (b) n (c) $2n+1$ (d) $2^n - 1$

=> We know in the worst case scenario maximum number of nodes in a binary Tree is $2^n - 1$ ✓



$1+2+4$
$= 7$
$= 2^3 - 1$

Full Binary Tree :⇒ A Binary tree in which every internal node has 2 children



We previous know $\mathcal{S}_n$ Binary Tree

Total number of leaf node = Total number of intermediate node with 2 children + 1

Total number of nodes = $I_2 + I_1 + L_n$

$2I_2 + 1$

$= 2I_2 + I_1 + 1$

(*) In full binary tree, (as it has no nodes with one child..

Total number of nodes = $I_2 + L_n$

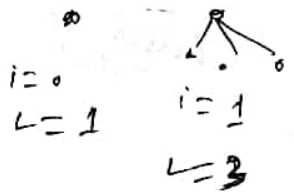
$= I_2 + I_2 + 1$

$= 2I_2 + 1$

(*) Total no of nodes in Full binary tree is always odd

3 Tree :⇒ Every internal node either 0 or 3 children

$L = 2i + 1$



Q The number of leaf nodes in a rooted tree of n nodes, with each node having 0 or 3 children

(a) $n/2$ (b) $(n-1)/3$ (c) $(n-1)/2$ (d) $(2n+1)/3$

$\Rightarrow L = 2i + 1$

or, $L = 2i + 1$

or $n = \frac{L-1}{2}$

$n = L - i$

or $L = n - i = n - \frac{L-1}{2}$

$\rightarrow$ (d) ✓

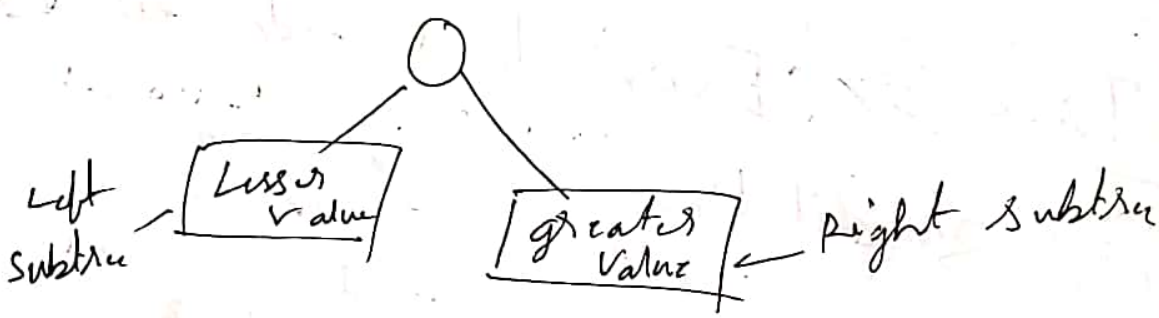
or $L = \frac{2n+1}{3}$ or $2L = 2n - L + 1$

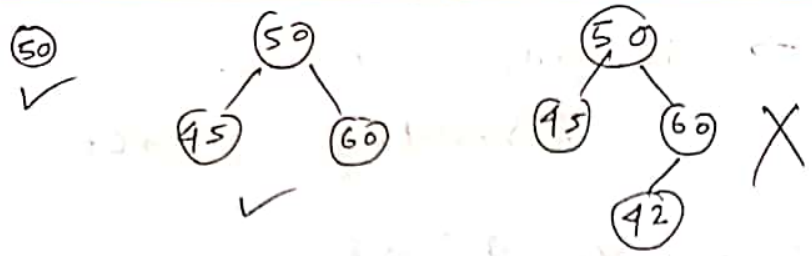
Left Skewed BT: Binary tree in which every internal node has only left child

Right Skewed BT: Binary tree in which every internal node has only right child.

Binary Search Tree (BST)

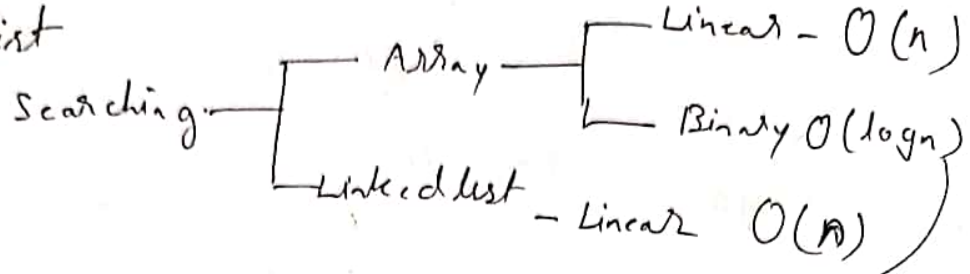
Binary tree whose internal nodes each store a key greater than all the keys in the nodes left subtree and less than those in the right subtree





✓ ✓ ✗ as 42 is in right subtree & it should greater than 50

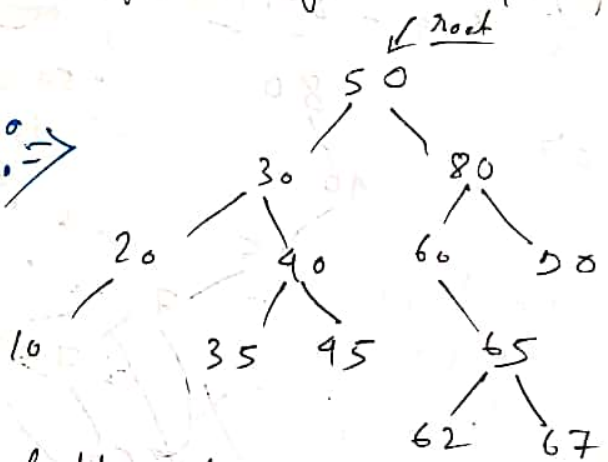
(✗) Before Bst we do search on array & linked list



↳ Array should be sorted — insertion O(n)
Deletion O(n)
Disadvantage
costly

Thats why we need a datastructure in which searching, insertion, deletion can be done in O(log n) time. Therefore we get Binary Search tree (BST)

Searching in BST :=>



Search 40 → root = element then return found

element < root then go towards left

else element > root then go towards right

Do this untill you found

if root = null then terminate & element not found

50, 30, 40 — We compare values

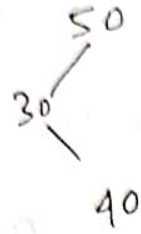
Searching Sequence

Search 67 => 50, 80, 60, 65, 67

Searching sequence

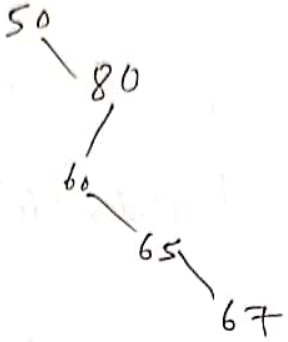
Search 32 - 50, 30, 40, 35

Search for 40



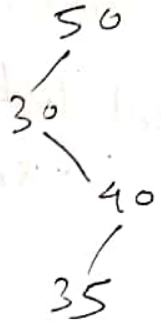
it is also BST

Search for 67



it is also BST

Search for 32



it is also BST

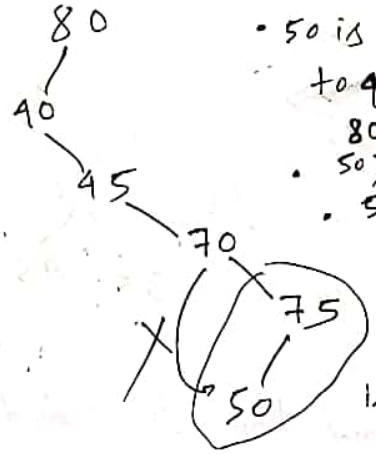
Searching Sequence is also a BST

identifying Valid Searching Sequence:

(1) Search 50

Seq - 80, 40, 45, 70, 75, 50 (wrong)

=>



- 50 is lesser than 80 so we will go to 40 which is also lesser than 80.
- 50 > 40 & 45 > 40 ✓
- 50 > 45 & 70 > 45 ✓

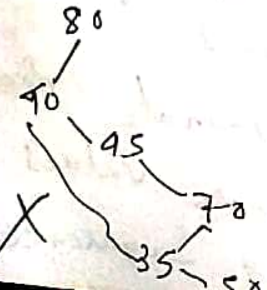
if should go to left than 70

⊗ if direction is wrong then it is not BST.

But if direction is right then we should check BST check

CH-21

80, 40, 45, 70, 35, 50



Q) identify correct/wrong sequence for searching 80 in a BST

(a) 9, 120, 12, 58, 88, 81, 86, 50, 84

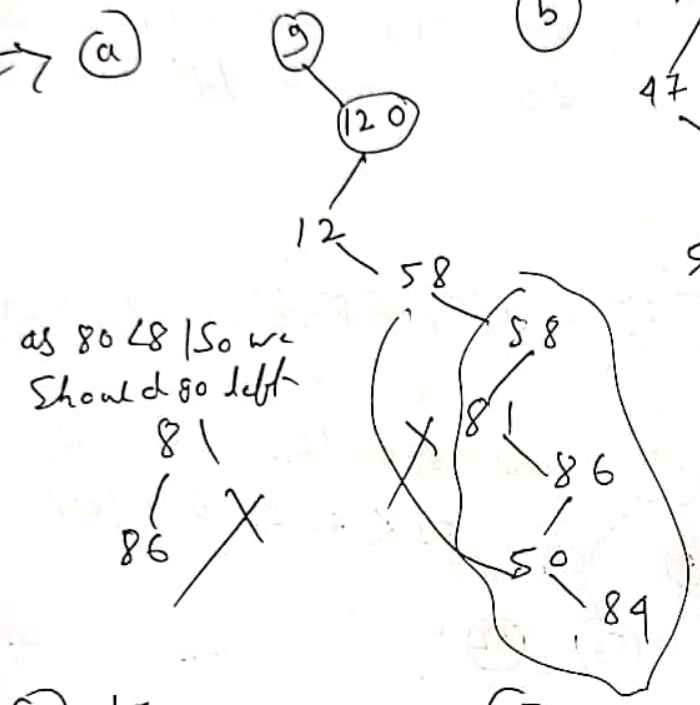
(b) 108, 47, 111, 58, 71, 70, 65, 75, 54, 80

(c) 15, 33, 46, 100, 94, 79, 82

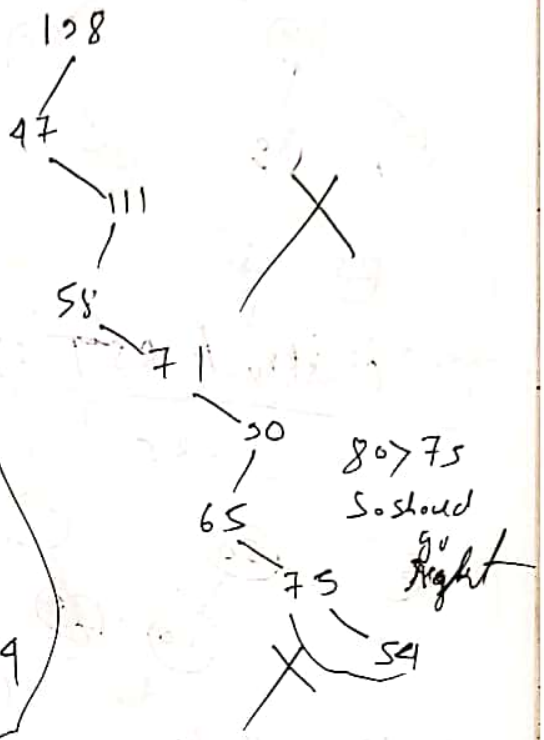
(d) 17, 37, 92, 117, 96, 85, 15, 56, 80

(e) 20, 40, 90, 98, 68, 70, 81

→ (a)

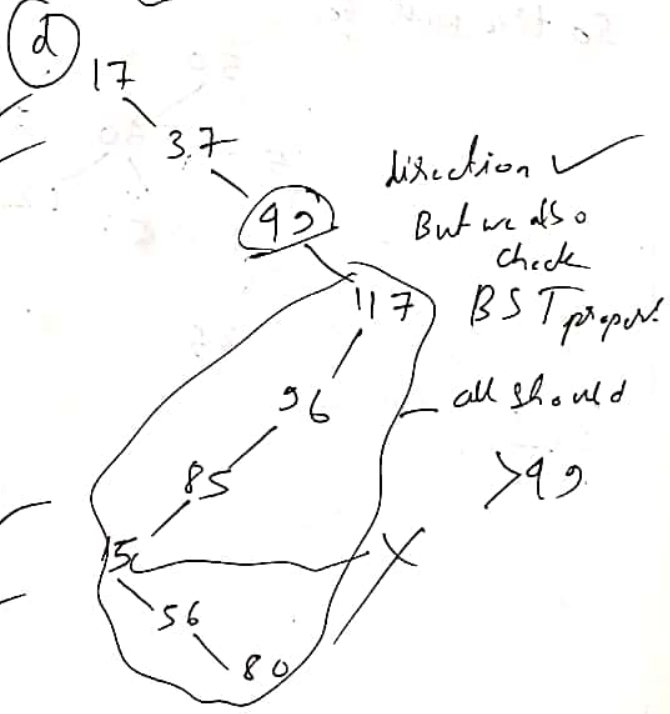


(b)

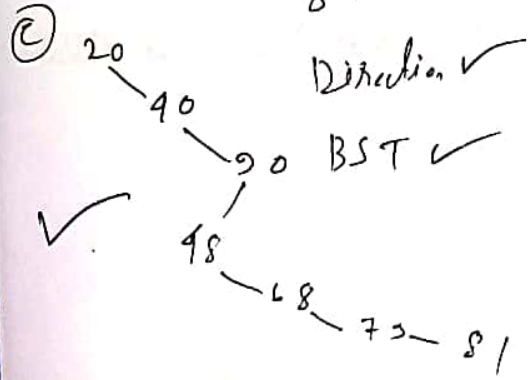


(c)

(d)



(e)

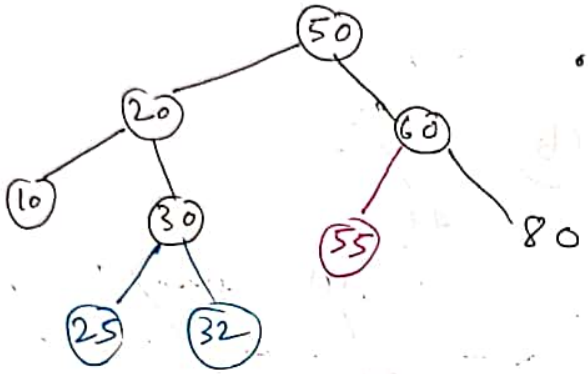


Insertion in BST ⇒

2 way — one is equal ~~key~~ key is kept in left side
— other is equal key is kept in right side

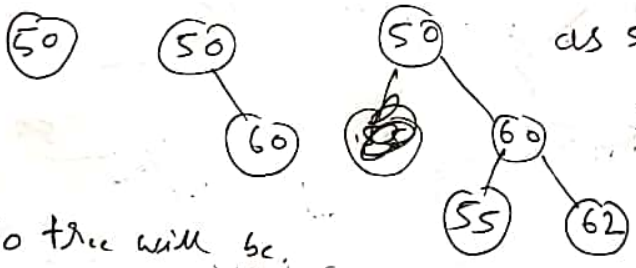
if this is the scenario then what to do it will be given.

insert = 55, 25, 32



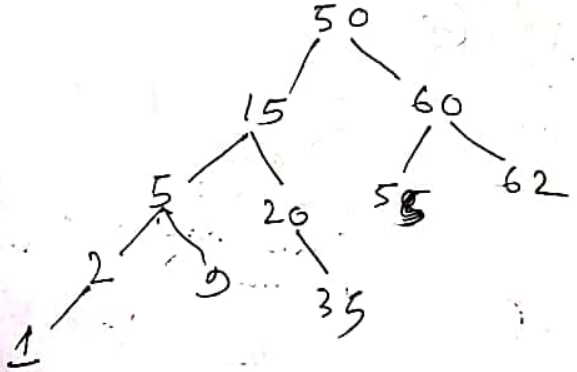
• 55 is > 50 so it will go right side
Then 55 < 60 then it will go in left side.

(i) Construct BST ⇒ 50, 60, 55, 62, 15, 20, 5, 9, 2, 1, 35



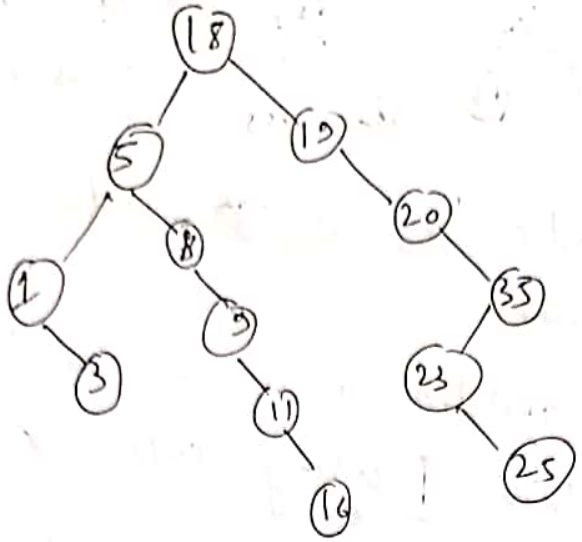
as 55 > 50 then right then 55 < 60 so it is left

So tree will be.



(11) Construct BST 18, 5, 8, 19, 20, 35, 23, 2, 11, 16, 1, 3, 25

⇒



Deletion in BST :⇒

Key will be given. Then we have to search it if found we have to check its number of children

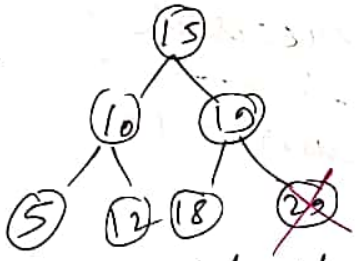
3 cases :⇒

Case 0 (i) Node has 0 child

Case 1 (ii) Node has 1 child

Case 2 (iii) Node has 2 children

(i) Case 0 :

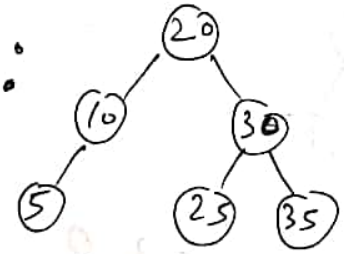


Key = 20

Search successful

20 has no child So We will simply delete it

(ii) case 1 :

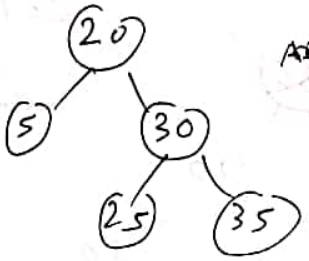


Key = 10

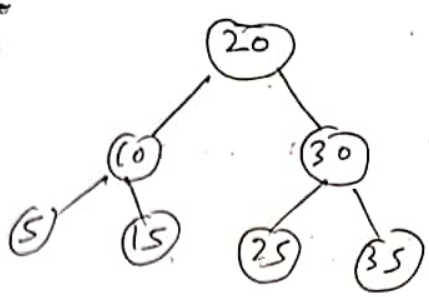
Search successful

10 has one child

10 will be deleted & 5 Will be replaced



(iii) Case 2:



key = 10

Search successful

10 has 2 children

(i) Find inorder successor ~~of~~ inorder predecessor of the node

(ii) Delete inorder successor or inorder predecessor from its place [using case 0 & case 1]

(iii) position it on place of deleted

inorder traversal (BST inorder is ascending order)

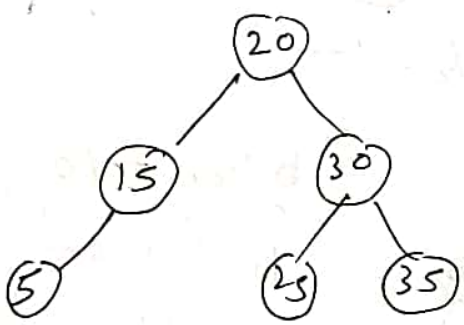
5 [10] 15, 20 25, 30, 35

15 — successor

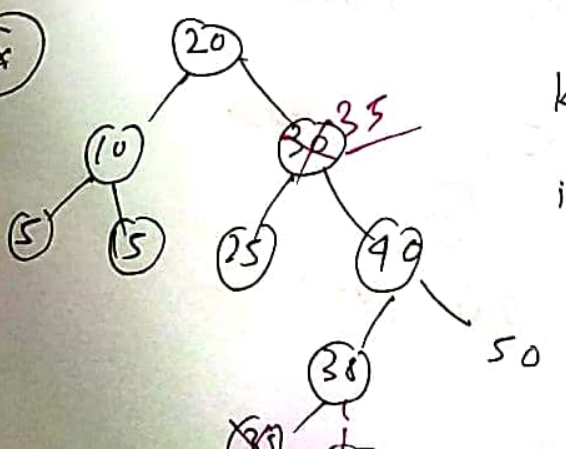
~~successor~~ predecessor

(a) using inorder successor:

inorder Successor = 15



(x)

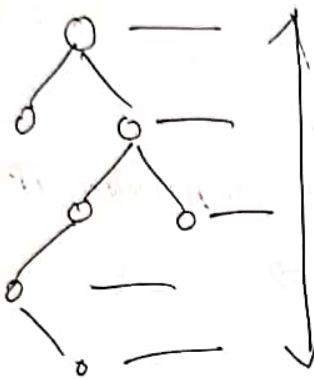


key = 30

inorder successor = 35

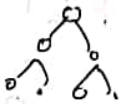
Complexity — O(h) — depends on height

	Searching	insertion	Deletion
Average	$O(\log_2 n)$	$O(\log_2 n)$	$O(\log_2 n)$
worst	$O(n)$	$O(n)$	$O(n)$



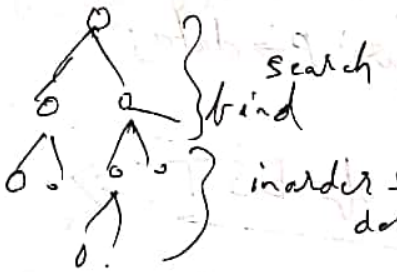
• max height number comparason in every height we compare

O(h) — Average $O(\log n)$, $O(n)$



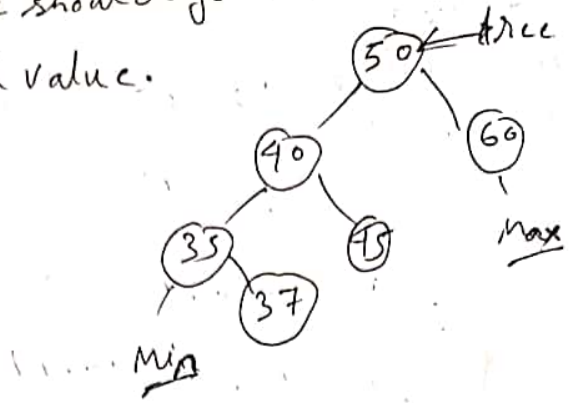
insertion = Searching + Place key/element

deletion = Searching + inorder Predessor + succes



Finding Minimum in BST :⇒

We should go towards left side untill we get null value.



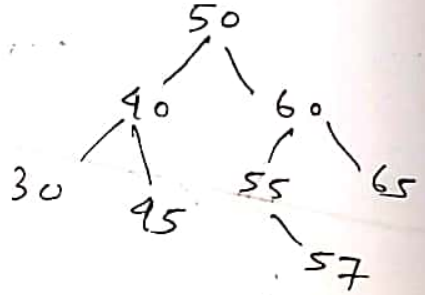

```
p = tree
while (p→Lchild != NULL)
{
    p = p→Lchild;
}
return p→data
```

Find Maximum in BST :⇒

```
p = tree
while (p→Rchild != NULL)
{
    p = p→Rchild;
}
return p→data;
```

Inorder of BST :⇒

always a sorted sequence of the keys in ascending order



30, 40, 45, 50, 55, 57, 60, 65

Q) The preorder traversal of binary search tree is given by 12, 8, 6, 2, 7, 9, 10, 16, 15, 13, 17, 20
Then postorder traversal will be

(a) 2, 6, 7, 8, 9, 10, 12, 15, 16, 17, 13, 20

(b) 2, 7, 6, 10, 9, 8, 15, 17, 20, 13, 16, 12

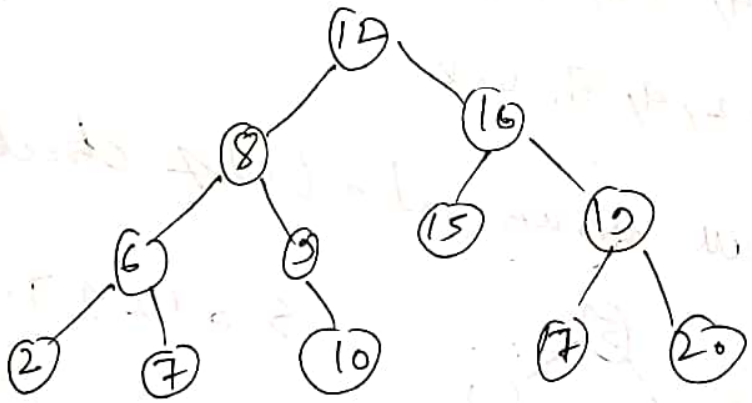
(c) 7, 2, 6, 8, 9, 10, 20, 17, 13, 15, 16, 12

(d) 7, 6, 2, 10, 9, 8, 15, 16, 17, 20, 13, 12

⇒ We know inorder of BST = ascending order of pre (postorder)

∴ inorder of this Bst = 2, 6, 7, 8, 9, 10, 12, 15, 16, 17, 13, 20

We can draw tree using preorder & inorder but here also we can draw using bst only



postorder will be 2, 7, 6, 10, 9, 8, 15, 17, 20, 13, 16, 12

→ (b) ✓

(*) (*)

- if preorder of BST is given → construct BST using one by one insertion from first to last key in preorder
- if postorder of BST given → from last to first

---

(4) A binary Search tree Contain value 1, 2, 3, 4, 5, 6, 7, 8
The tree is traversed preorder, the values are printed out. Which is valid

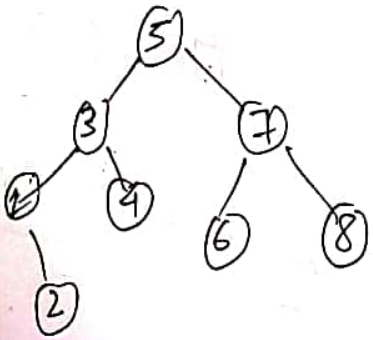
(a) 5, 3, 1, 2, 4, 7, 8, 6

(b) 5, 3, 1, 2, 6, 4, 8, 7

(c) 5, 3, 2, 4, 1, 6, 7, 8

(d) 5, 3, 1, 2, 4, 7, 6, 8

⇒ We will assume it a bst & check by inserting



5 3 1 2 4 7 6 8
~~~~~
✗ not match

(d) ✗
~~~~~

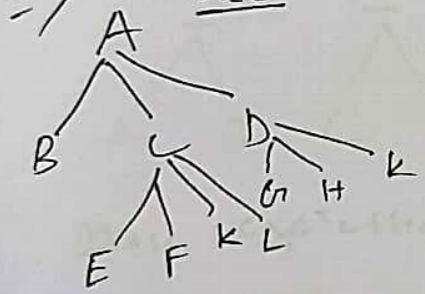
	Unlabelled node	n distinct keys
Binary Tree	$\frac{2nC_n}{n+1}$	$\frac{2nC_n}{n+1} \times n!$
Binary Search Tree	-	$\frac{2nC_n}{n+1} \times 1$

Questions

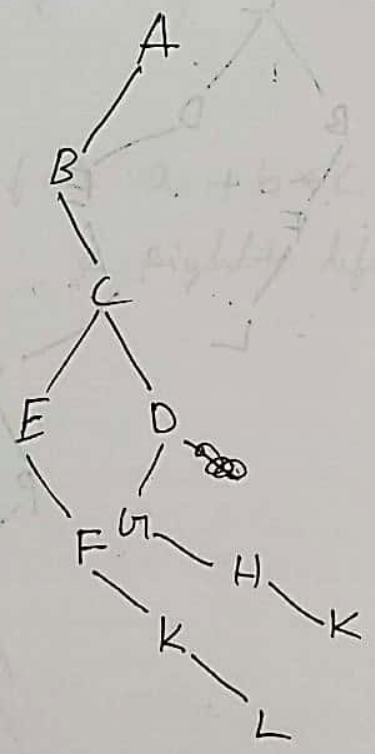
(1.) Construct tree for following parenthesis Representation. If it is not binary tree then Represent these into Leftmost child — Right sibling

A(B, C(E, F, K, L), D(G, H, K))

=> tree



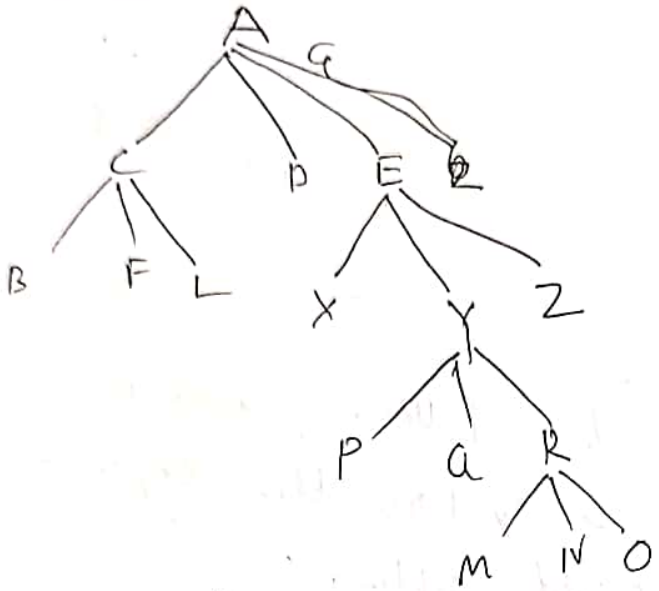
binary tree.



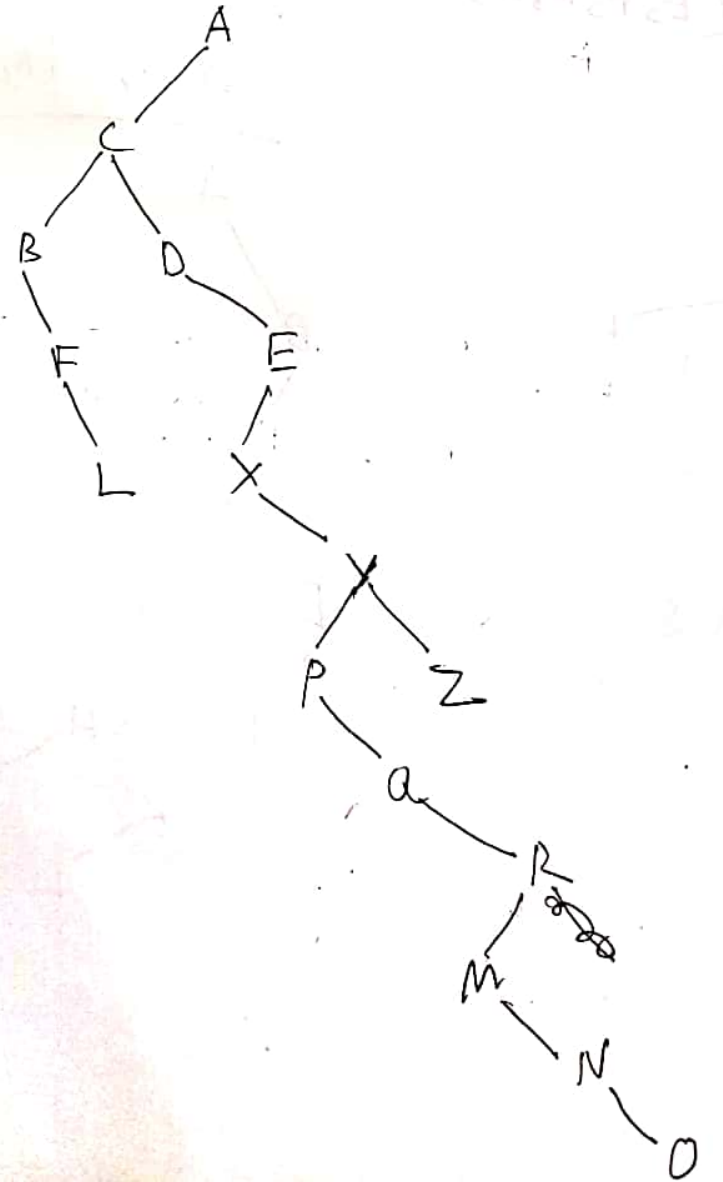
(2) Same question

A (C (B, F, L), D, E (X, Y (P, a, R (M, N, O)), Z))

⇒ tree



Binary tree



(3) Draw expression tree for

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

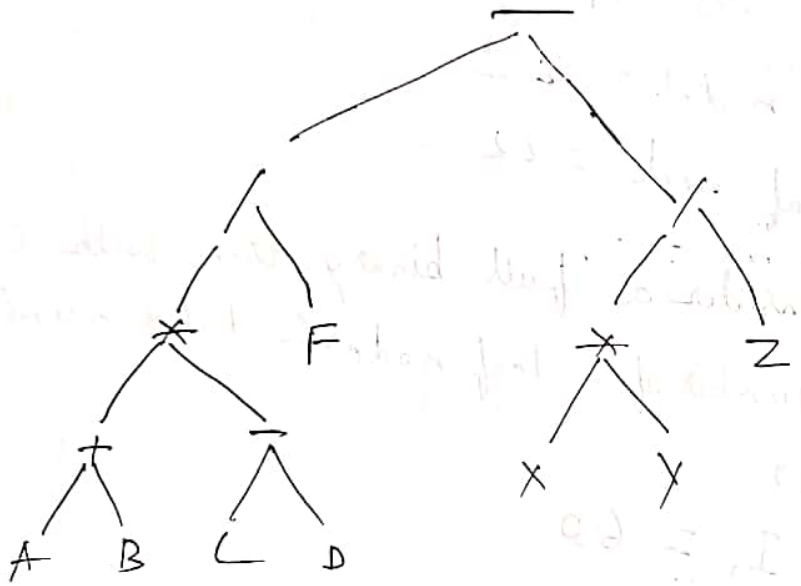
⇒ () high precedence | high to low | Left to Right associativity

* /

\+ —

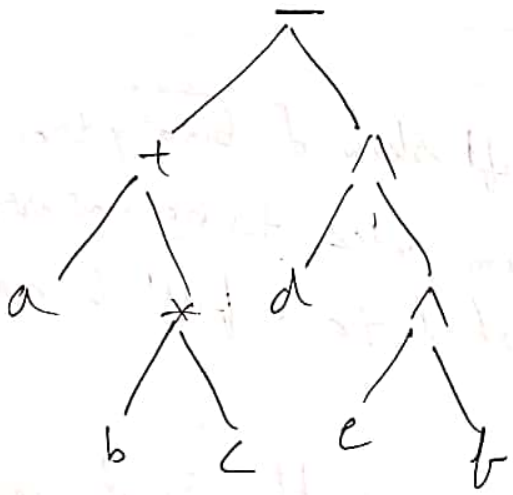
operator = root

operand — child



(4) Draw expression tree for a + b * c - d ^ e ^ f

⇒ ^ high Precedence & Right to left association



(5) Consider a full binary tree with 43 nodes. Number of leaf nodes & internal nodes are?

⇒ Full — every internal node has 2 children only

$$\text{Leaf node} = I_2 + 1$$

$$n = L_n + I_2 = I_2 + 1 + I_2$$

$$\text{or } 43 = 2I_2 + 1$$

$$\text{or, } I_2 = 21$$

∴ internal node = 21

∴ leaf node = 22

(6) Consider a full binary tree with 60 internal nodes. Number of leaf node & total number of nodes are?

⇒ $I_2 = 60$

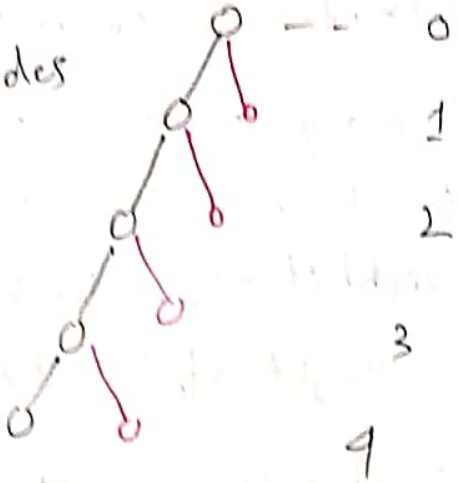
$\therefore L_n = 60 + 1 = 61$

$\therefore n = L_n + I_2 = 121$

(7) Consider a left skewed binary tree of height h. How many minimum nodes to be inserted into this tree to convert it to a full binary tree of height h.

Note - height of tree with single node is 0

⇒ h number of nodes is to be inserted



Total inserted = 4 (h)

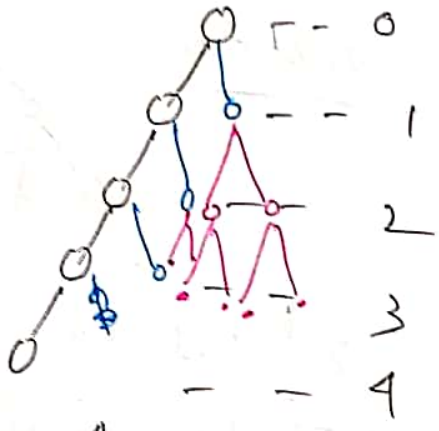
if it was Right Skewed then also it will be same h number of nodes

8] in a left skewed binary tree of height h. how many minimum nodes to be inserted to convert it to complete binary tree of height h
Height of Single node 0

⇒

$$2^h - (h+1)$$

Total node — given number of node



$2^4 - (4+1)$
$= 11$

2] if Previous question was Right Skewed

total nodes $= 2^{h+1} - 1$

already node $= h+1$

So $2^{h+1} - 1 - (h+1)$

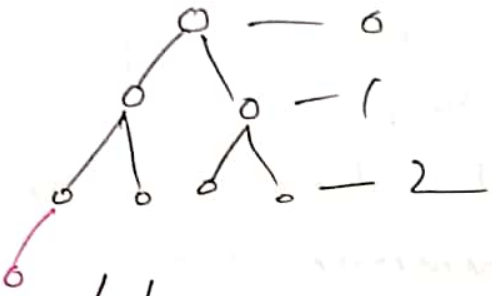
$= 2^{h+1} - (h+2)$

$2^{h+1} - (h+1) - 1$

10] Consider a complete binary tree of height h. How many minimum nodes to be inserted into this tree to convert this tree into complete binary tree of height (h+1). The best case & worst case both.

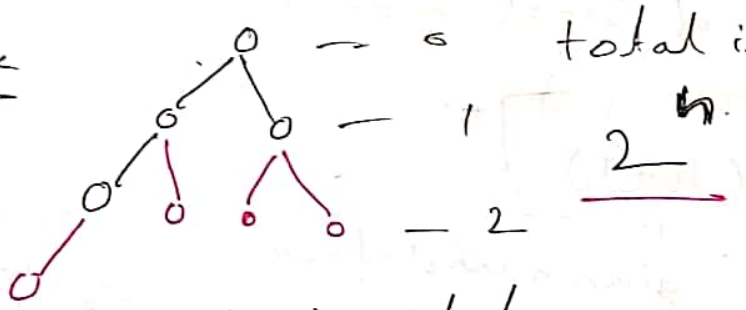
Note - Height of tree with single node 0

=> Best case



1 will be inserted

Worst case



2^h will have to inserted

11] identify correct/wrong Searching Sequence for searching 50 in a bst

(i) 80, 73, 65, 95, 57, 90, 71, 82, 50

(ii) 40, 45, 90, 80, 70, 60, 63, 36, 50

(iii) 24, 33, 46, 98, 47, 80, 75, 61, 48, 50

(iv) 28, 95, 29, 94, 37, 85, 39, 76, 43, 50

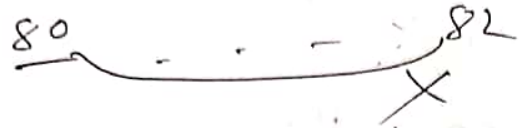
(v) 36, 39, 89, 81, 73, 67, 95, 59, 35, 50

=> We will take first number (80) as 50 is less than 80 so every element after 80 should be less than 80

• next if previous condition fail then it is not valid

number if previous got success then we will take the next something untill got a fail condition

(i) after 80 everything should be less than 80 but there is 82 which is > 80. ✗



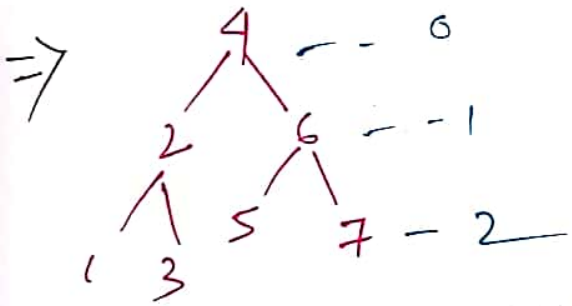
(ii) 40 . . . . . 36 ✗

(iii) ✓

(iv) ✓

(v) 36 . . . 35 ✗

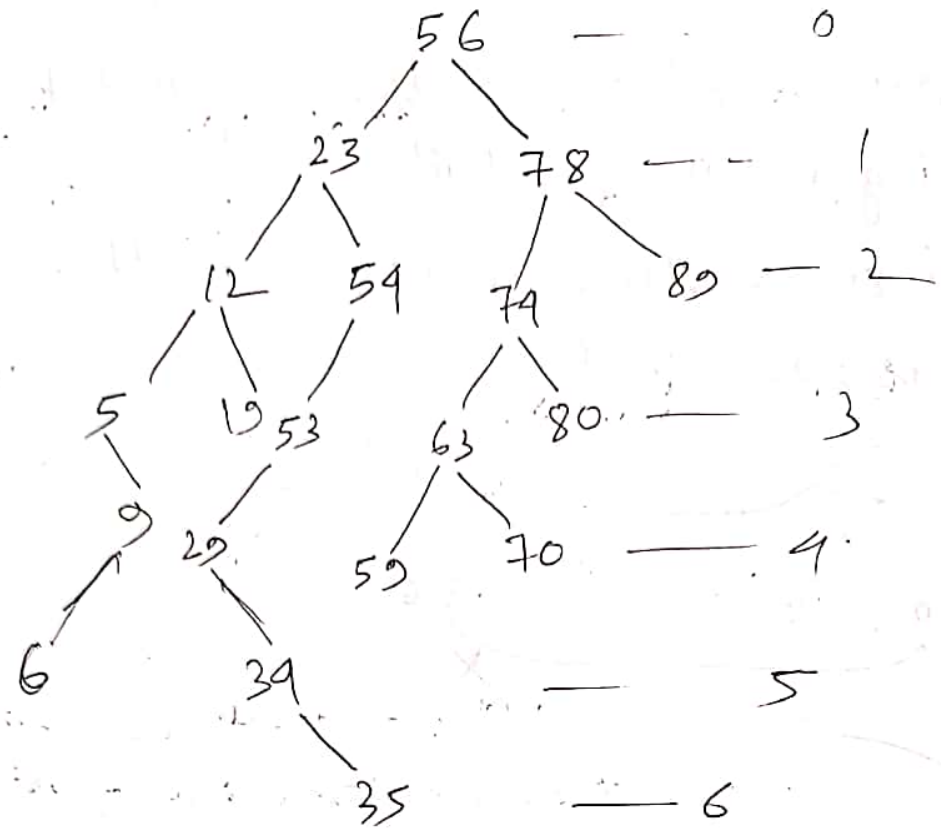
12) Construct BST using keys 4, 2, 6, 5, 1, 3, 7 max level number in tree



Max level = 2

13) inserting the elements 56, 23, 78, 54, 53, 89, 74, 12, 5, 29, 34, 3, 6, 63, 70, 59, 80, 19, 35 in an Empty BST in the sequence. Max level of the tree will be

⇒



Max level = 6

19)

AVL Tree

- named after inventors Adelson, Velsky, Landis
- AVL tree is a self balancing binary search tree (BST)

Need of AVL: - In BST searching complexity is O(h) & in the worst case h=n and then complexity O(n)

- in bst nodes are not arranges equally in both side

So we wanted a Bst where both side almost same number of elements. Therefore AVL came

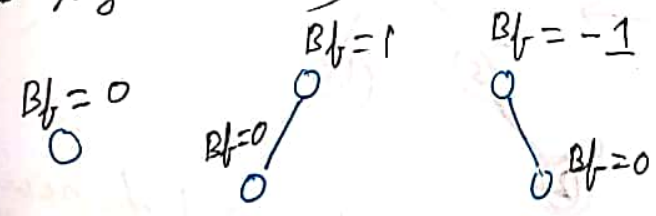
Height of AVL ≃ log n

Balance Factor: ⇒ balanced tree means

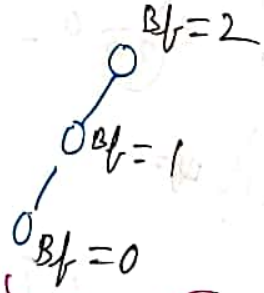
For each node

$$\text{Balance factor} = h_L - h_R = \begin{cases} 1 \\ 0 \\ -1 \end{cases}$$

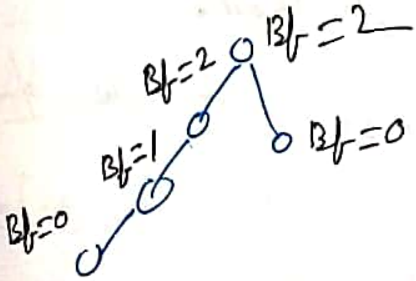
(difference between the height of Left side & right side)



all balanced



Not balanced



Self balancing :- After insertion & deletion self check will be there.

- if tree become imbalanced then restructure the tree so that it becomes balanced

(*) Restructure means some kind of rotation.

Types of imbalances :- ⇒ imbalances can occur if insertion / deletion happen

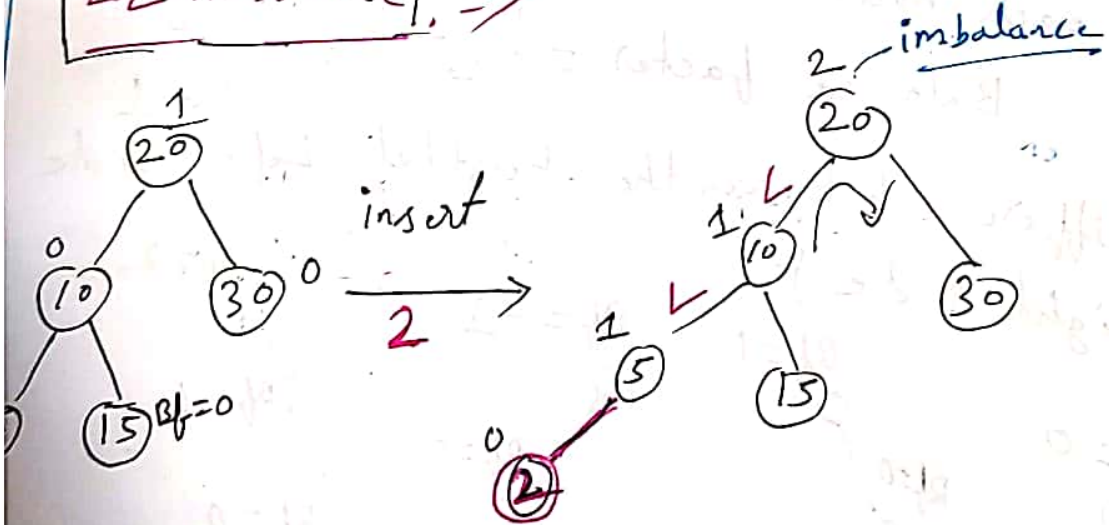
There is 4 type of imbalance

(i) LL : Left Left
(ii) RR : Right Right
(iii) LR : Left Right
(iv) RL : Right Left

Insertion In AVL Tree

- insertion as according to BST
- Find imbalance & recognize type
- if imbalance eliminate it

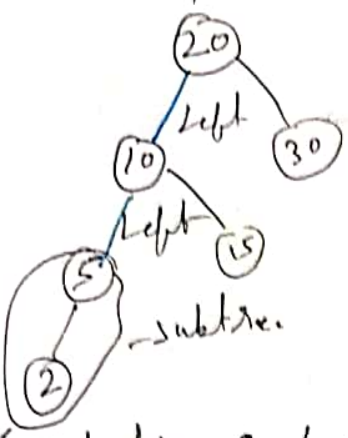
LL imbalance :- ⇒



(xx) check balance factor of the parent of newly erted. if problem solve else check again its parent. o this until you arrive at root.

ere imbalance at node (20) because of tion of node (2)

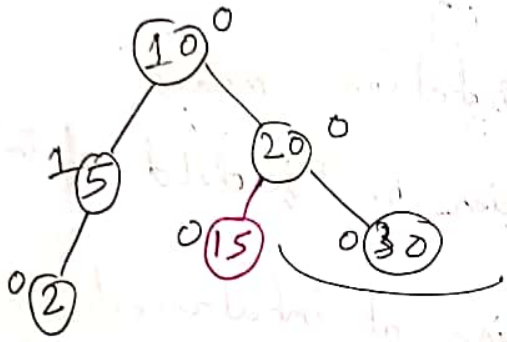
(*) To recognize type of imbalance, From imbalance node you have to check 2 step down which subtree has newly inserted node



- imbalance at 20 because of insertion in 20's left kid's left subtree

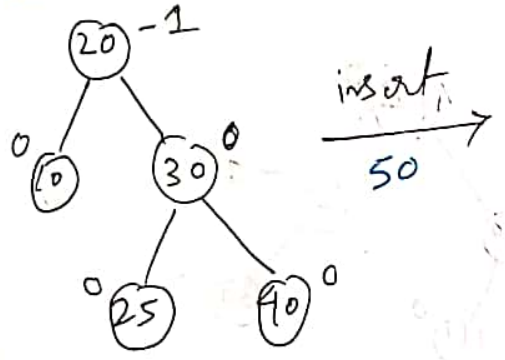
Solution: LL rotation / single right rotation

LL rotation at 20

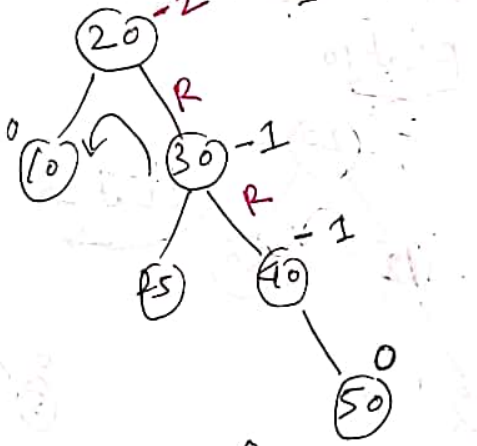


for maintaining BST property

RR imbalance ⇒

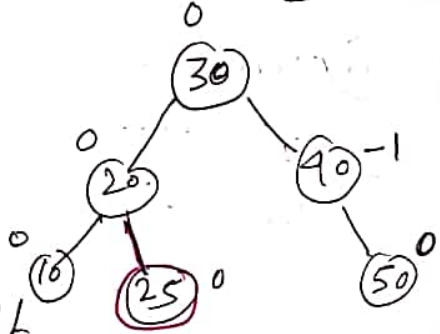


insert 50



balanced tree

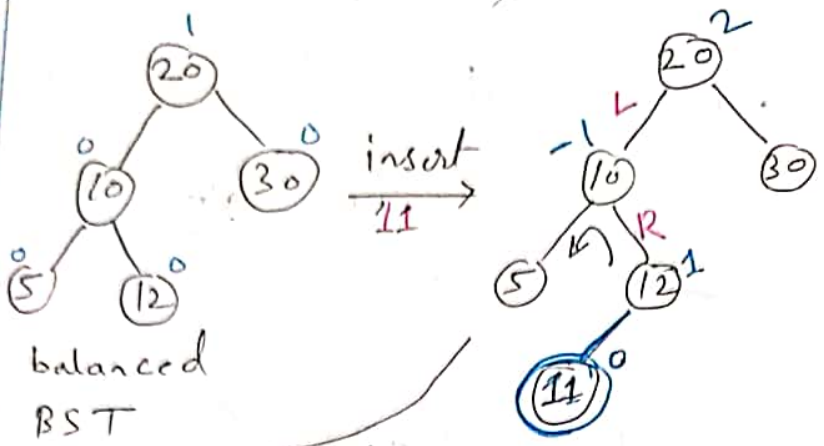
- imbalance at 20 because of insertion in 20's Right kid's Right subtree



Solution: RR Rotation / single left Rotation

balanced

LR imbalance



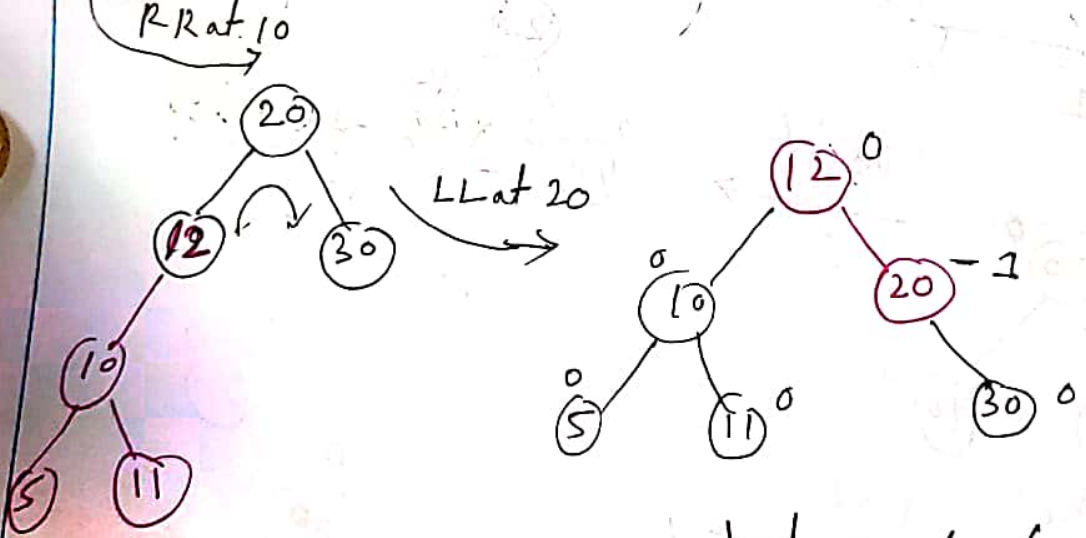
- imbalance at 20 because of inserting in 20's Left & its right Subtree

Solution: LR Rotation means

(i) RR Rotation on left child of imbalanced node

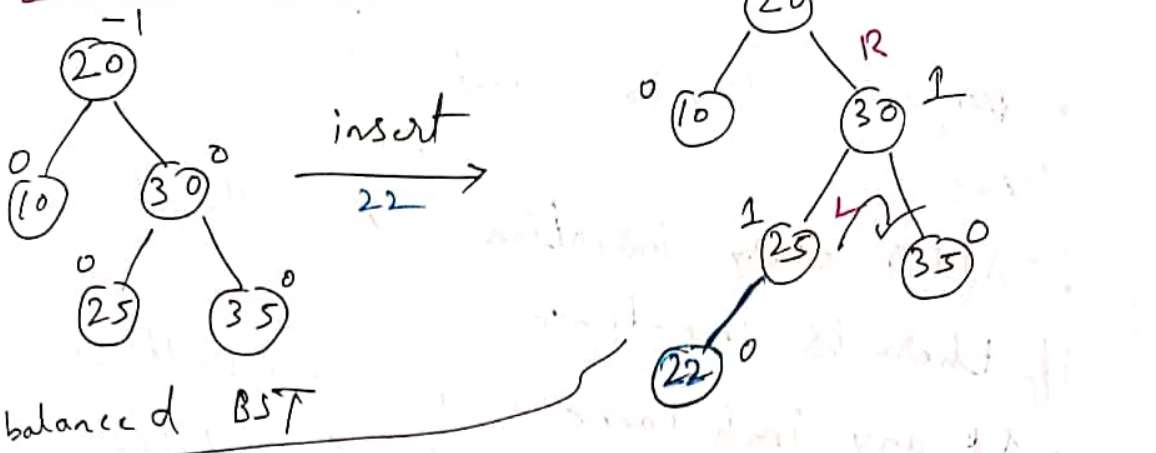
(ii) LL Rotation at imbalanced node

- Here left child of 20 is 10



balanced tree

RL imbalance

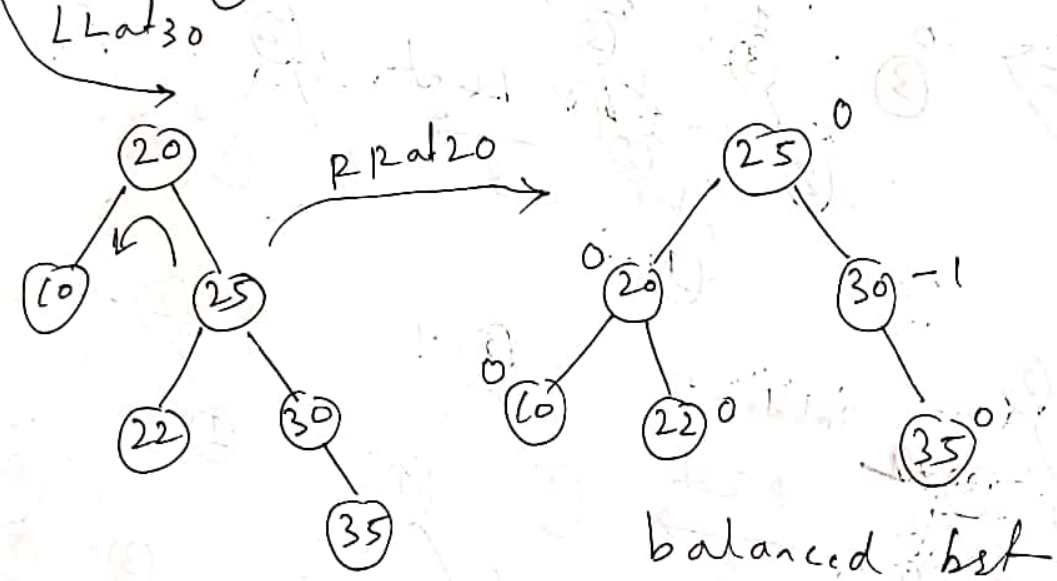


balanced BST

Solution :- RL rotation means

(i) LL rotation at right child of imbalance node

(ii) RR Rotation at imbalance node

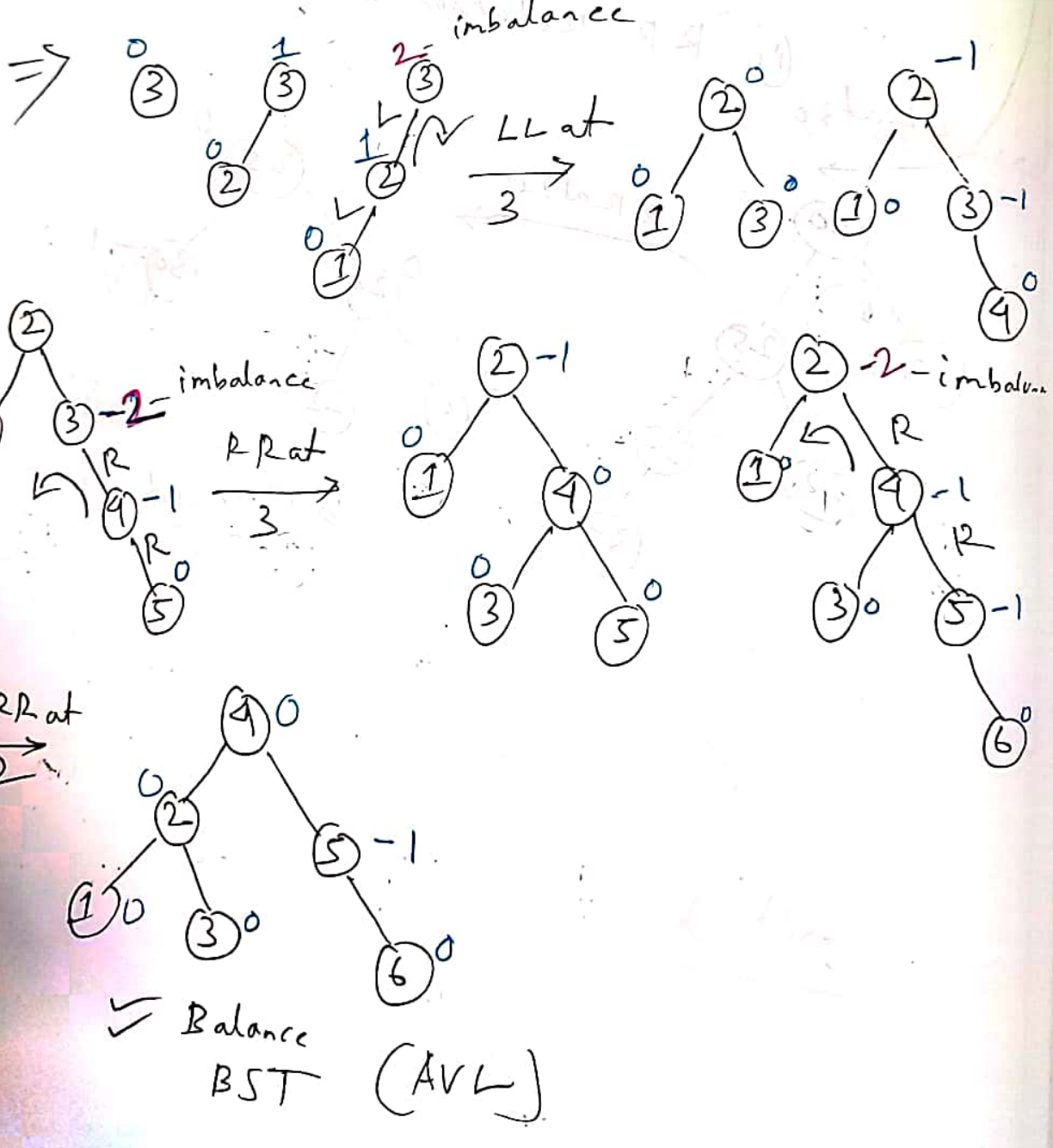


balanced bst

Create AVL Tree

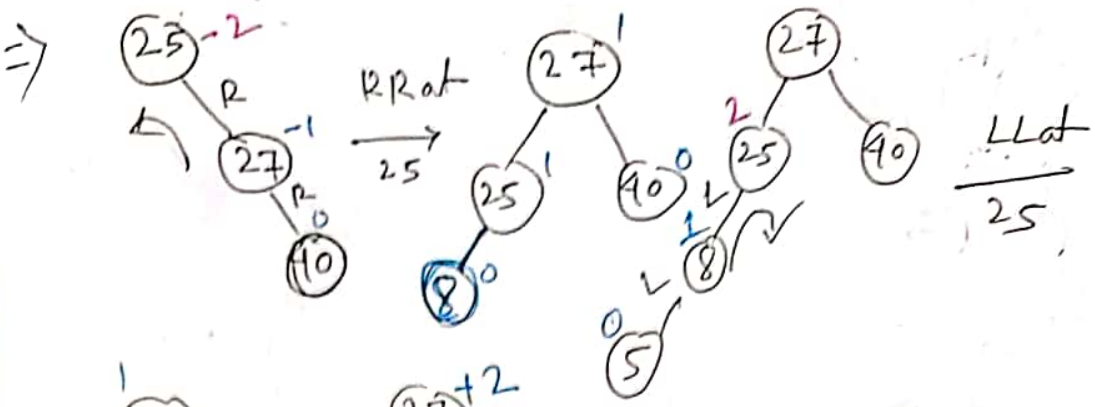
- We will try to create AVL Tree using keys
- After every insertion we will check if there is imbalance
- If any imbalance occur we will solve it using imbalance solution (Rotation)

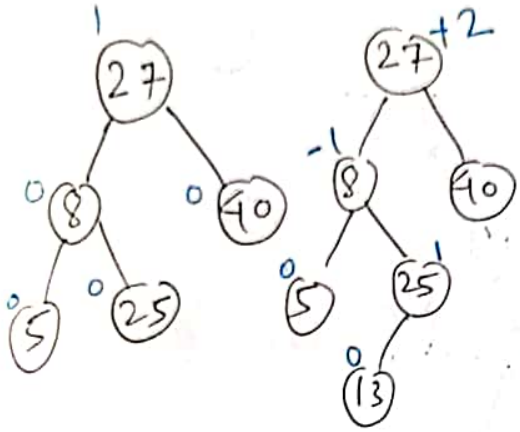
Ex Create AVL tree using keys
3, 2, 1, 4, 5, 6

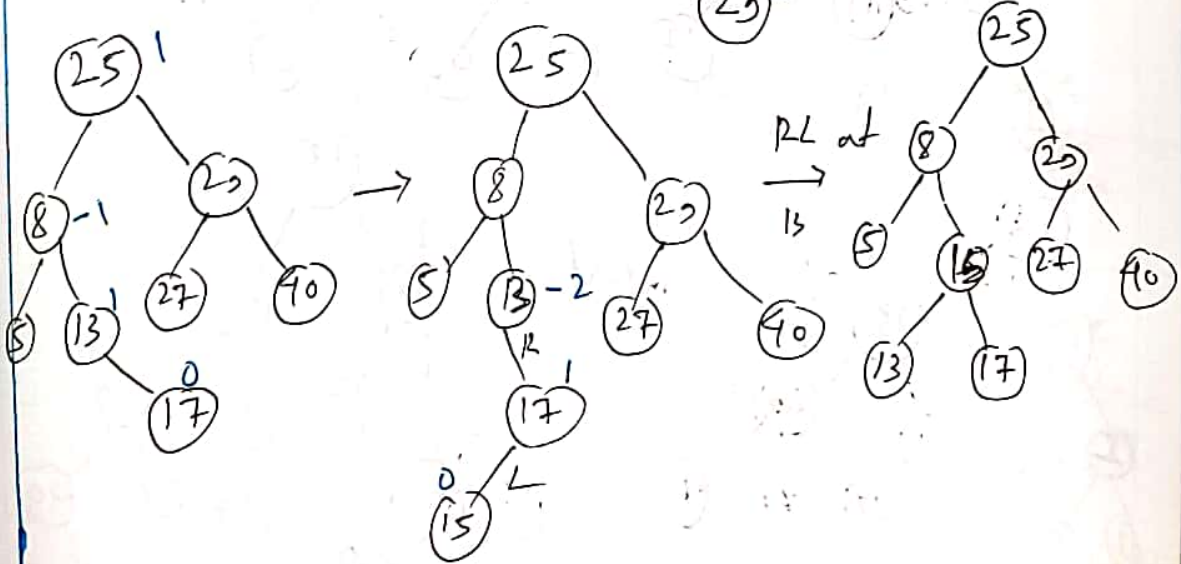


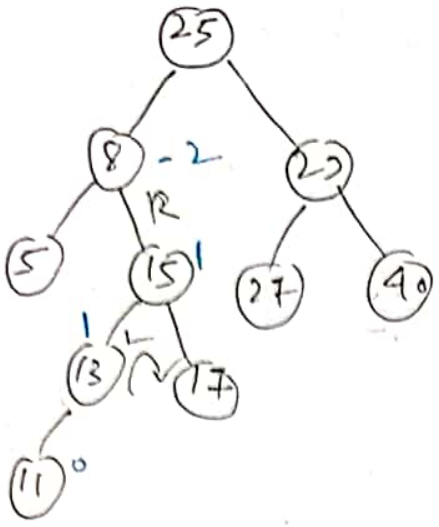
Ex: Create AVL tree using keys

25, 27, 40, 8, 5, 13, 20, 17, 15, 11, 1, 2, 6



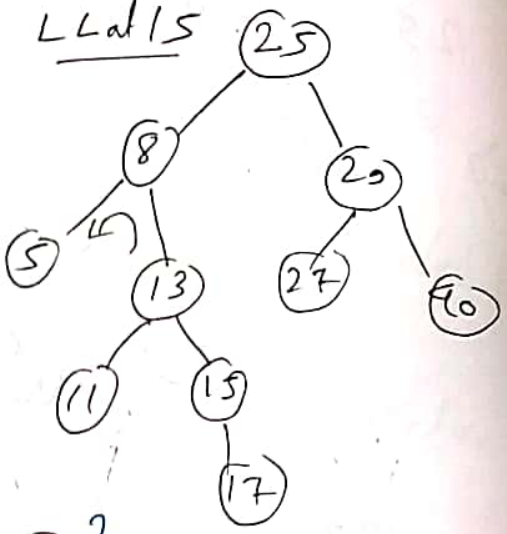




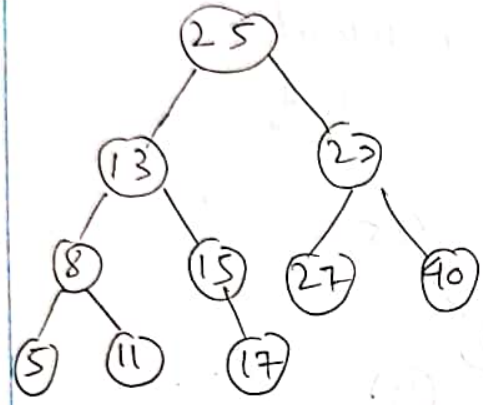


RL at 8

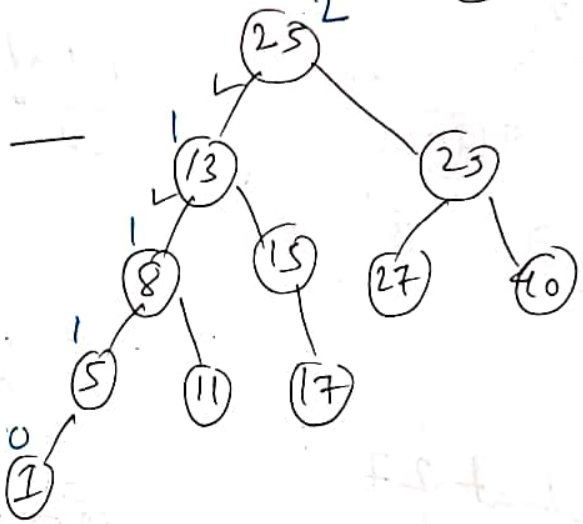
LL at 15



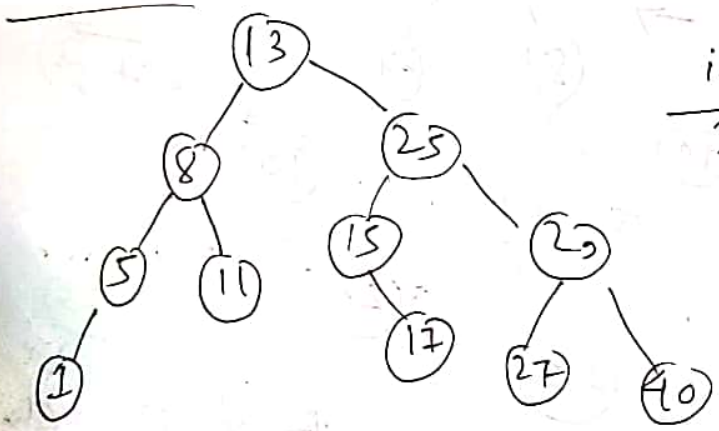
RR at 8



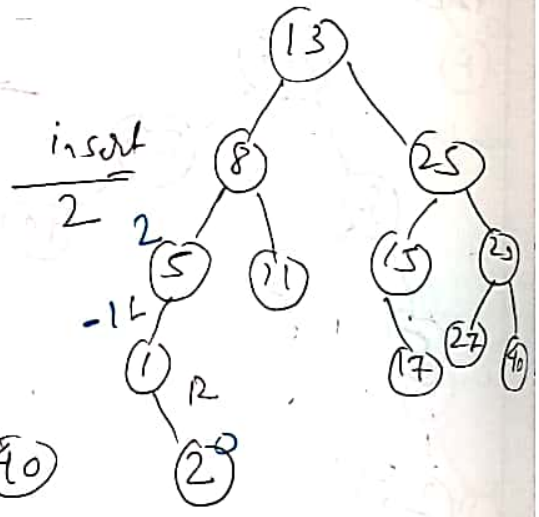
—

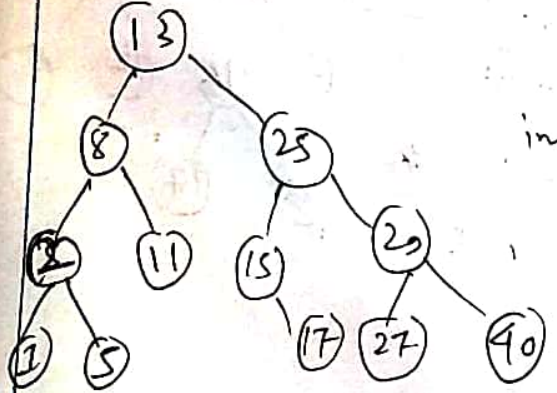


LL at 25

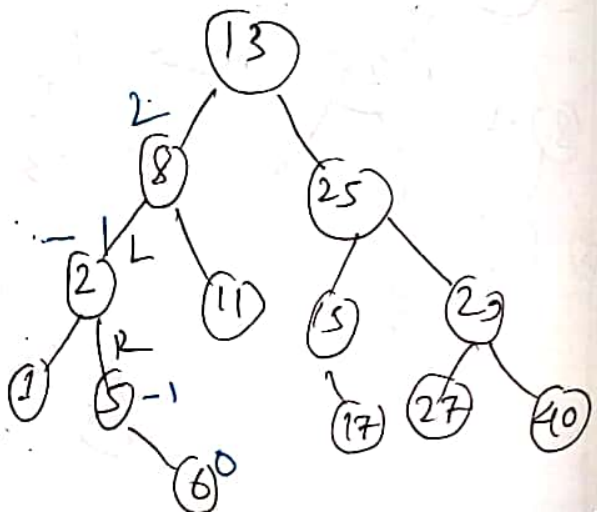


insert 2



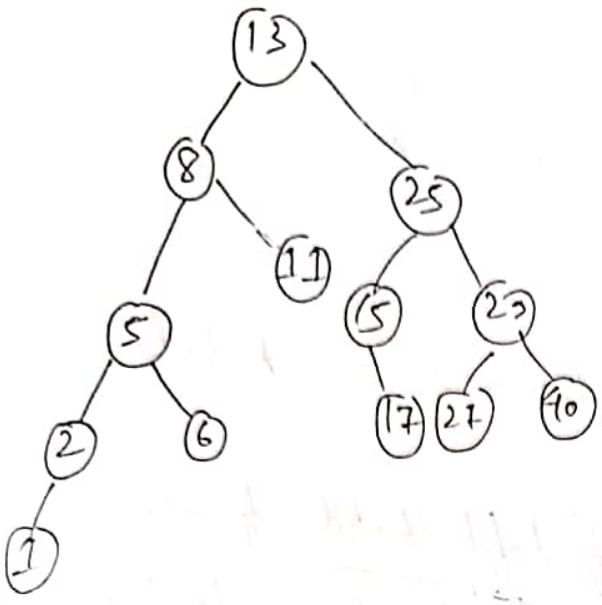


insert 6

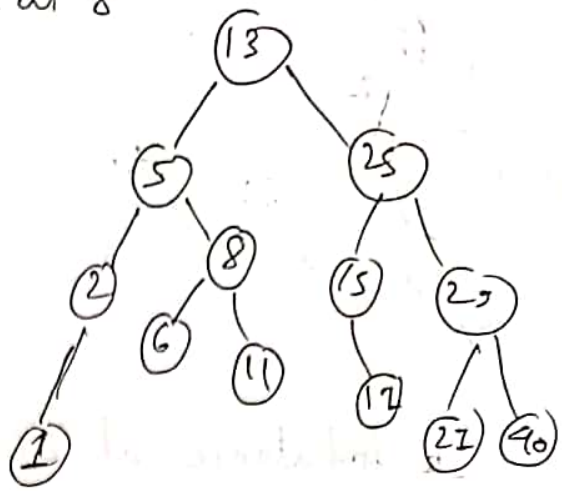


LR at 8

RR at 2



LL at 8



Final

Deletion in AVL

In BST there is 3 cases in deletion

(i) Case 0 no child (ii) Case 1 1 child (iii) case 2 2 child

Deletion in AVL three steps

(i) Delete node using BST cases

(ii) Detect if any imbalance & recognize it

(iii) Eliminate imbalance using appropriate rotation.

0, 1, -1 denotes loading of the subtree

There is 6 cases

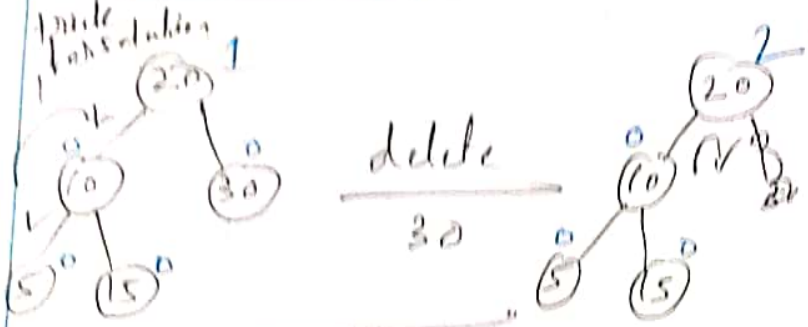
(i) R_0(LL) (iv) L_0(RR) (*) Here R/L is the Right subtree/Left subtree of imbalance node where deletion took place

(ii) R_1(LL) (v) L_1(RL)

(iii) R_{-1}(LR) (vi) L_{-1}(RR)

(*) 0, 1, -1 is the balanced factor of opposite side child of imbalanced node.

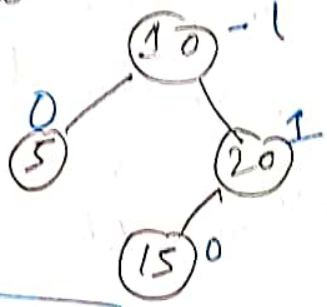
(i) R_0 case :-



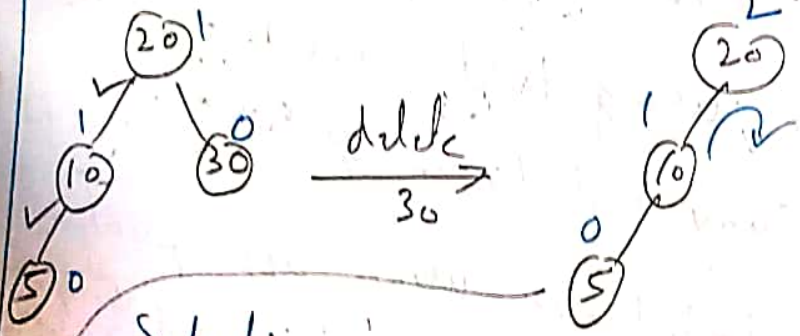
- imbalance at 20, because of deletion in 20's Right subtree
  Balance factor of left child of 20 is 0

Solution - LL rotation (Single Right Rotation)

LL at 20



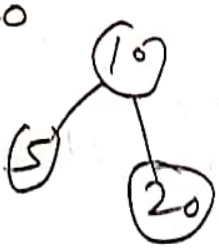
(ii) R_1 case :-



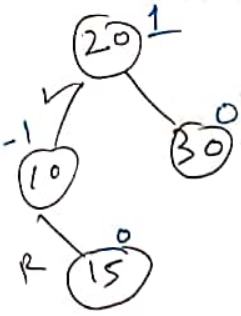
- deletion in Right
- Load of 20's Left child 10 is 1

Solution - LL Rotation (Single Right Rotation)

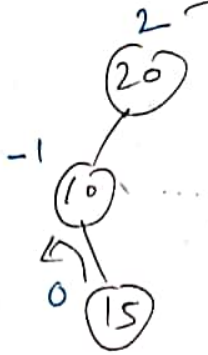
LL at 20



(iii) **R-1 Case** :=>



delete 30



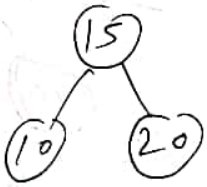
- inbalance
- deletion in right
- Bf of 20's left child 10 → -1

Solution → LR Rotation

- Here LR at 20 — (i) RR at 10
  (ii) LL at 20

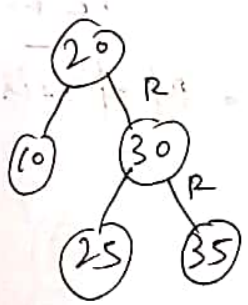
RR at 10

LL at 20



Balanced

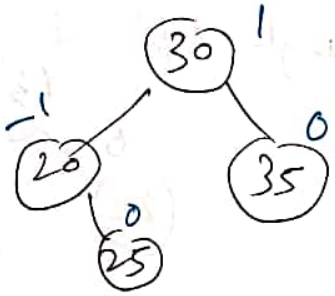
(iv) **L0 Case** :=>



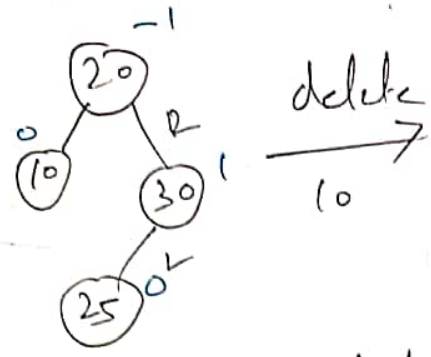
delete 10

- deletion is in left
- Balance factor of Right child of 20 is -0

Solution → RR rotation (Single Left rotation) / RR at 20



(V) L_1 case :⇒



- deletion is in left
- Bf of 30 is 1

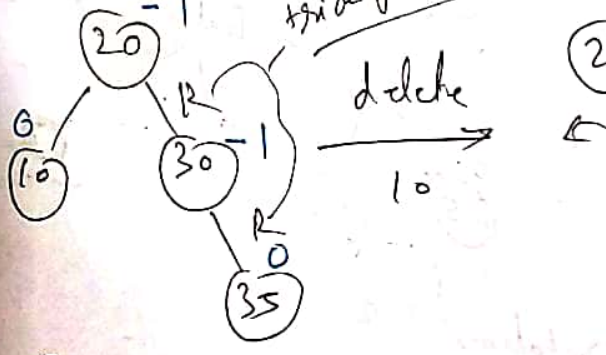
Solution - RL Rotation (Single left Rotation)

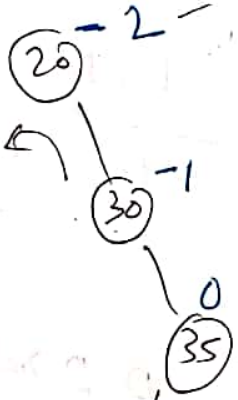
RR at 20

RL at 20 means LL at 30 then RR at 20

Balanced.

(VI) L_{-1} case :⇒

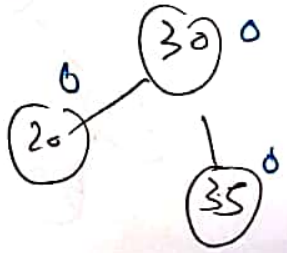




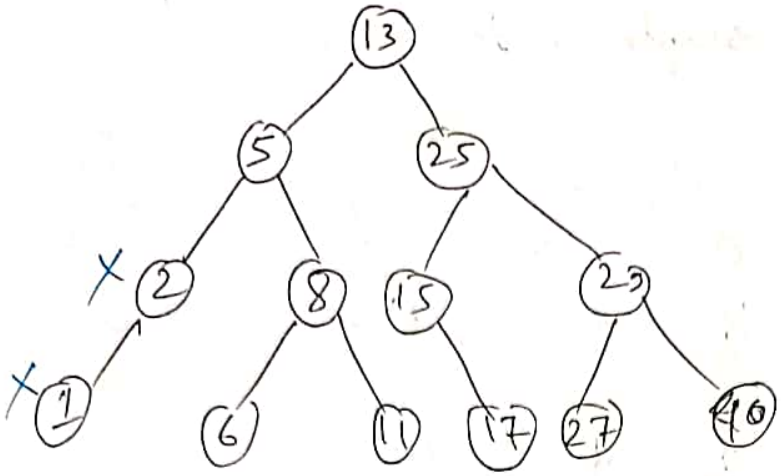
- deletion is in left
- Bf of 30 is -1

Solution - RR Rotation (Single left

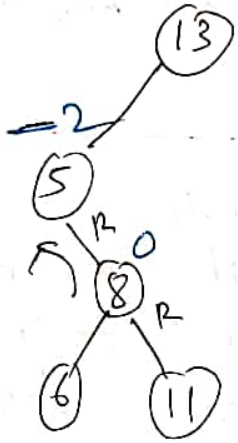
RR at 20



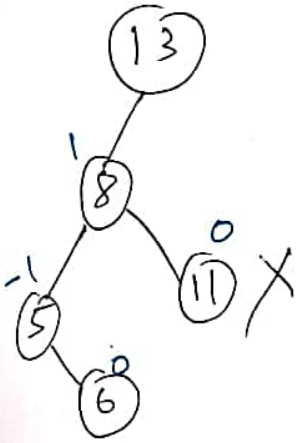
Q) Delete the keys from the given AVL tree 1, 2, 11, 17, 5, 8, 13, 6, 15



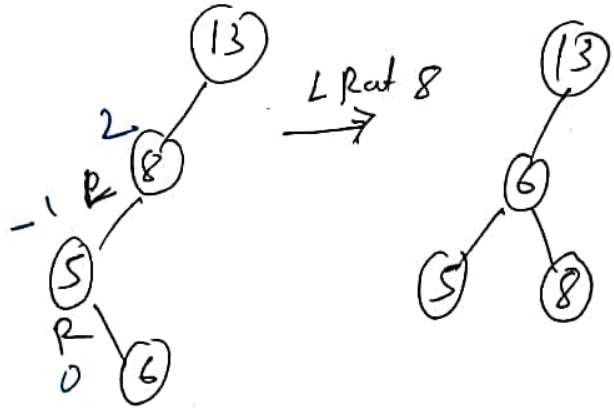
⇒ after deleting 1 no problem but after deleting 2. Balance factor of 5 becomes -2



this is L0 case Solution is RR Rotation (Single left Rotation)

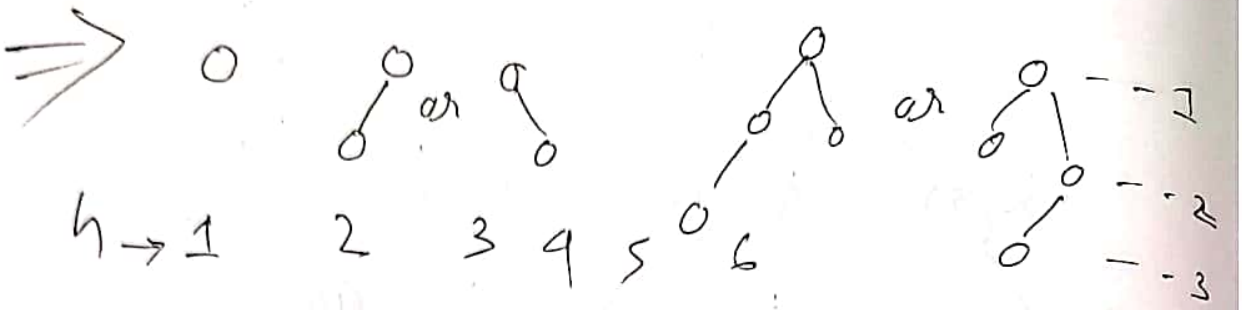


after deleting 11 again imbalance occur at 8



This continues

Q| The minimum number of nodes in AVL tree with height H is ___ ?
Note - With single node height is 1

⇒ 

$h \to$ 1 2 3 4 5 6

$n_{min} \to$ 1 2 4 7 12 20

$$n_{min}(h) = \begin{cases} 1 & \text{if } h = 1 \\ 2 & h = 2 \\ n_{min}(h-1) + n_{min}(h-2) + 1 & h > 2 \end{cases}$$

Complete / dense graph :- each node connected with every node

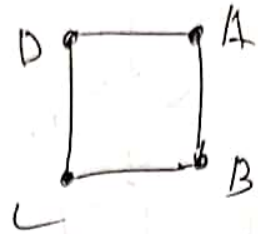
~~Equal degree~~

Regular graph - equal degree of every node

Graph

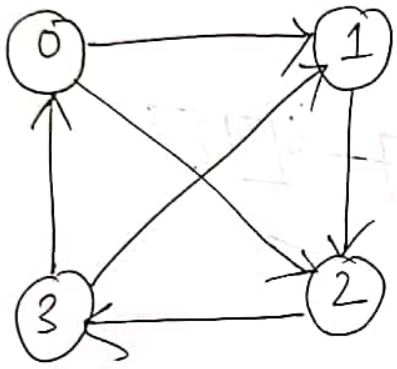
- A graph is a non-linear data structure consisting of nodes (vertices) and edges.

$G = \{ V, E \}$



Graph Representation :=>

(1) Adjacency Matrix or Table :=>



it is directed graph

n = no of vertices

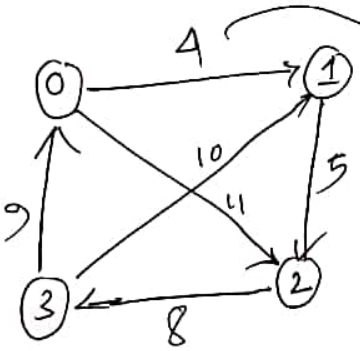
n x n matrix.

	0	1	2	3
1	F	F	T	F
2	F	F	F	T
3	T	T	F	F
0	F	T	T	F

- If it is undirected then ⟷ both possible

Adv:

- If we have weighted graph then instead of true we can keep weight of edge.



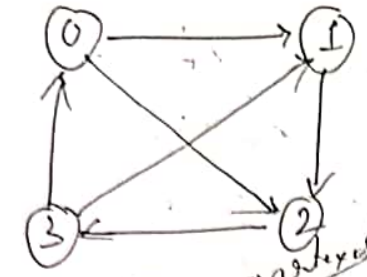
weight

	0	1	2	3
0	0	4	11	0
1	0	0	5	0
2	0	0	0	8
3	9	10	0	0

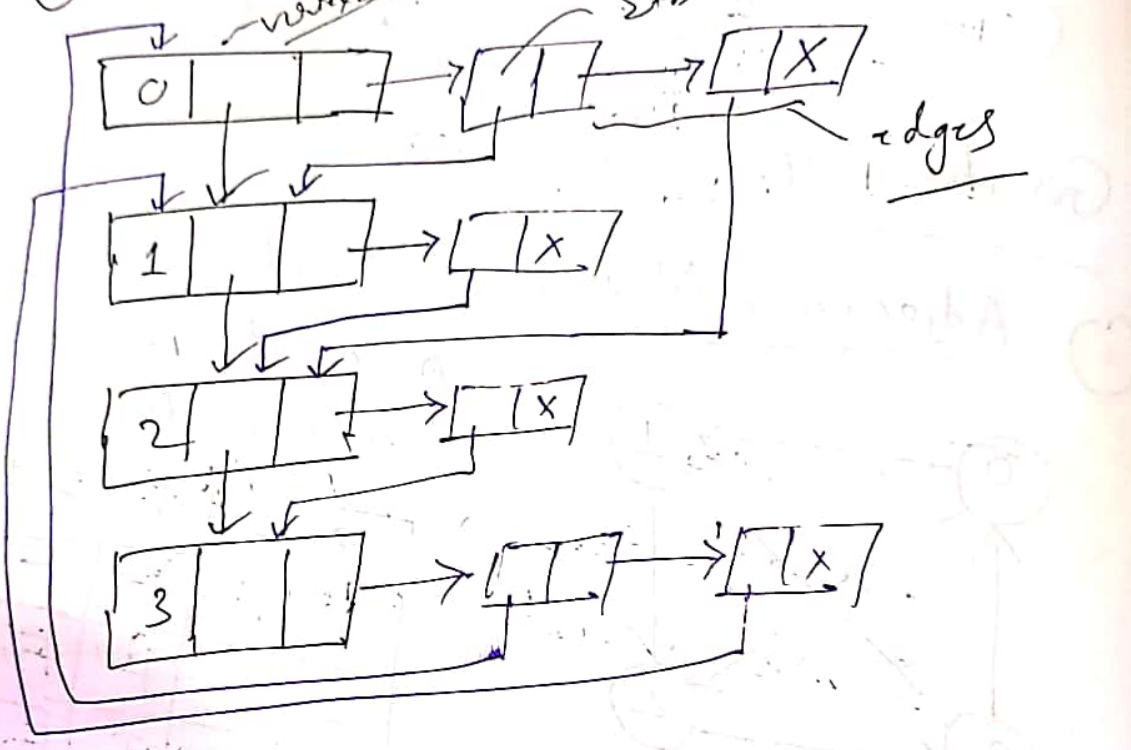
Dis :-

- To do operation on each edge we have to traverse all possibilities (all edge + non edges) (complexity $\Theta(n^2)$)

(ii) List (Linked list) Representation :=>



• 2 type of list Needed



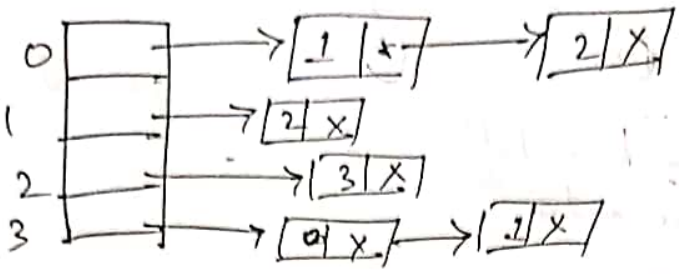
Dis :=> Though we can perform operation only on these edges which actually exist.

But there is also one problem.

All the vertices are stored in sequential order (Linked list). We can't reach to any vertex directly.

(iii) Hybrid Representation :=>

- Vertices represent in array (can jump to any vertex
- Edges are represented in list (perform operation only on existing edges).



Traversing In Graph:

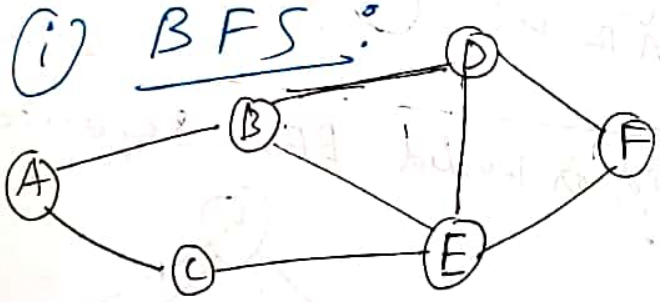
Two types

(i) Breadth First Search (BFS)

(ii) Depth First Search / Traversal (DFS)

(*) In both search a source (particular vertex) will be given

(i) BFS:

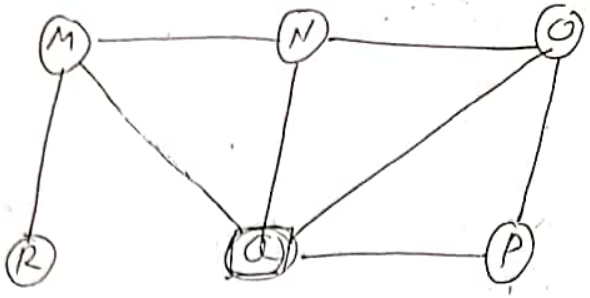


Suppose source - A

A B C D E F
A B C E D F
A C B E D F

- First we will write its neighbour connecting them & write them in any order
- Then we will take next vertices and write their neighbour
- This will be go on until all traversed.

Q| BFS has been implemented using queue data structure.



Which one of the following is a possible order of visiting the nodes

(i) M N O P Q R

(ii) N Q M P O R

(iii) Q M N R O P

(iv) P O Q N M R

⇒ if it started with M

M N Q R (i) X

N O Q M (ii) X

Q O N M X (ii)

P O Q N M R ✓ (iv) ✓

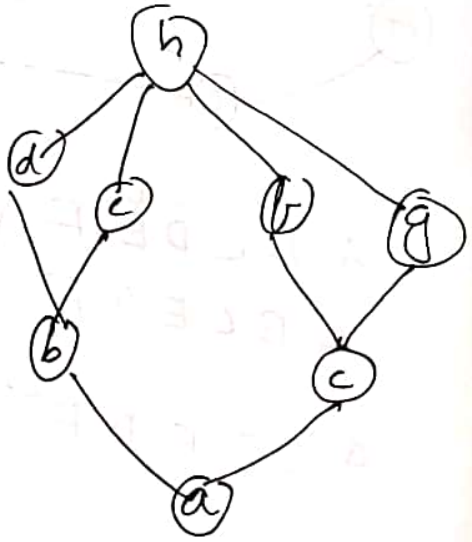
Q| identify Valid or Invalid BFS Sequence for given graph

(i) a, b, c, d, e, f, g, h

(ii) a, c, b, f, g, d, e, h

(iii) a, c, b, d, e, g, f, h

(iv) a, c, b, g, f, e, d, h



⇒ (i) a, b, c, d, e, f, g, h ✓

(ii) a, e, b, f, g, d, c, h ✓

(iii) a, c, b, d, c ✗

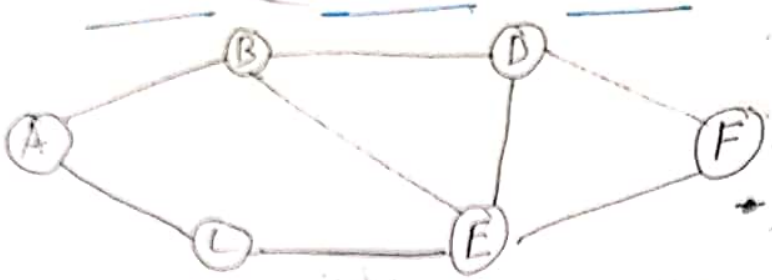
first c neighbour should be written

(iv) a, c, b, g, f, e, d, h ✓

BFS using Queue :→

```
BFS (v)
{
  • Mark v as visited and Enq (v)
  • While (queue not empty)
      • x = dequeue()
      • Find all unvisited neighbours of x
        mark them visited & enqueue them
}
```

DFS (Depth First Search) in Graph



- Selecting any node as arbitary node as root node & explore as far as possible along each of its neighbours before backtracking.
- Start from the root & mark the node & move to the adjacent unmarked node & continue this loop untill there is no unmarked adjacent node. Then backtrack and check for other unmarked nodes and traverse them.

Finally print the node in Path

⇒ Source — A

A B E F D C

Source was A then it will go to its adjacent node B. Then from B it will go E, Then F then D As D has no adjacent node unmarked So it will return to F. F also has no adjacent node unmarked So it will return to E. E has one adjacent node unmarked that is C. it will go to C. Then return E. Then return B then return A.

Other posibilities are.

A B E C D F ✓

A C E B D F ✓

A C E D B F ✓

A C E F B D ✗ — After going to F there is one unmarked node D So D should traversed first not B

A B D F C E ✓

A B D E F C ✓

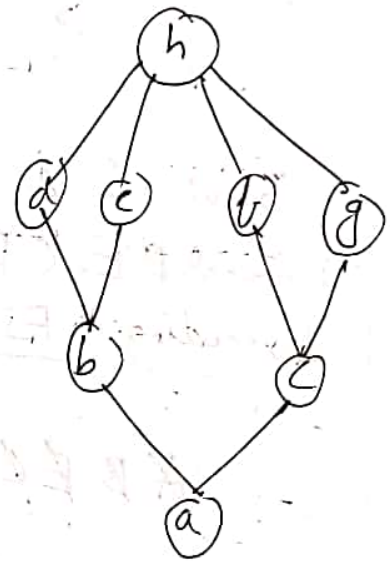
(*) (a) identify valid or invalid DFS Sequence

(i) a, b, e, h, f, g, d, c

(ii) a, c, f, h, g, e, b, d

(iii) a, b, c, h, d, f, c, g

(iv) a, c, f, h, e, b, d, g



→ (i) a, b, e, h, f, g, d, c

after f there is one neibour adjacent node c but instead of it goes to g

(ii) a, c, f, h, g, e, b, d ✓

(iii) a, b, e, h, d, f, c, g ✓

(iv) a, c, f, h, e, b, d, g ✓

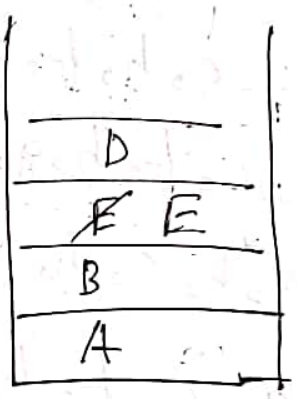
DFS using Stack:

```
DFS(U)
  { i. Mark U as visited
    ii. while( there is neighbours v of
                u which is unvisited)
        { DFS(V)
        }
  }
```

Source – A

Sequence-A B E C D F

Executing DFS:



~~A~~ ~~B~~ ~~E~~ ~~C~~ ~~E~~ ~~D~~ ~~F~~ ~~D~~ ~~E~~, ~~B~~, A Stack

C has no adjacent unmarked it will return and pop out E

F has no adjacent it will return to D, D has no neighbour unvisited then return to E & after doing this stack will be empty

- Nodes that pushed on to Stack A B, E, D
- Nodes that not pushed on to stack - C, F
- nodes pushed more than once E

Q) Perform DFS on given graph with source as vertex 1

vertex	Adjacency
1	2 3
2	1, 4, 5
3	1, 6, 7
4	2, 8
5	2, 8
6	3, 8
7	3, 8
8	4, 5, 6, 7

- Traversal Sequence
- Vertex which are not pushed onto the stack
- vertex which are pushed onto stack more than once.

⇒ • in such kind of representation we should take care of sequence

ex - 1 → 2, 3 if it is unvisited

We have to first go to 2 from 1 then can go to 3.

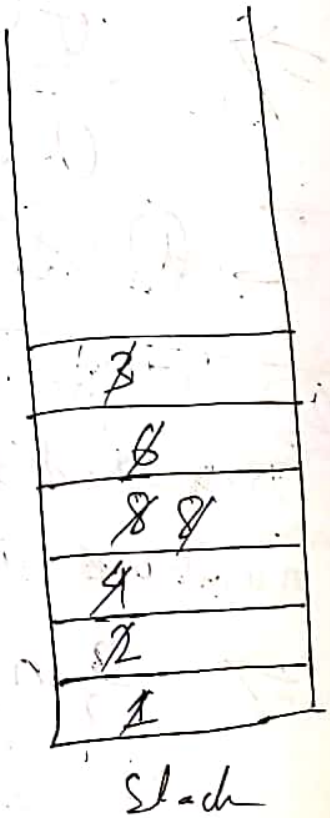
(X) Traversal Sequence:

12485637

DFS Executing

~~1 2 4 8 5 8 6 3 7 3 6 8 4 2 1~~

- vertex not pushed - 5, 7
- vertex pushed on stack - 8



Stack

Complexity	BFS	DFS
Adjacency matrix	$O(V^2)$	$O(V^2)$
Linked Representation	$O(V+E)$	$O(V+E)$

Questions

(1) Consider an algo which takes n number of inputs and performs an operation on its which require $n-1$. The best possible run time complexity

(a) $O(n)$ (b) $\theta(n)$ (c) $O(n \log_2 n)$

(d) A, 4 B both

⇒ $\theta(n)$ – more appropriate as it says exactly

$O(n)$ – it also can

→ (d) ✓

(2) The minimum number of comparison required to find min & max of 500 numbers is ____?

⇒ $\frac{3}{2}n - 2$

$= \frac{3}{2} \times 500 - 2 = 748$

(3) Consider an array of n elements. ~~Total~~ min comparison required to calculate second minimum element of array if n = 64?

⇒ min comparison = $n + \log_2 n - 2$

in this case = $64 + \log_2 64 - 2$

= 68

(4) Consider an array A[-6 ... 15] which is stored in memory starting from location 1000. Assume each element in memory is stored on 4 location. Then the location of element A[2]

⇒ l.c of A[2] = Base address + Size * number of previous element / relative index

= 1000 + 4 * [2 - (-6)]

= 1032

(5) consider a 2d array A[-3 ... 7][6 ... 12] The starting address is 1000. Each element occupies 4 memory locations. What is address of elements A[0][5] in row major & column major order?

$\Rightarrow$ m = rows = 7 − (−3) + 1

= 11

n = columns = 12 − 6 + 1

= 7

Base = 1000 Size W = 4

R.M.O

Loc of A[i][j] = Base + W * [(i − LB$_1$) * n + (j − LB$_2$)]

= 1000 + 4 * [(0 − (−3)) * 7 + (0 − 6)]

= 1096

C.M.O

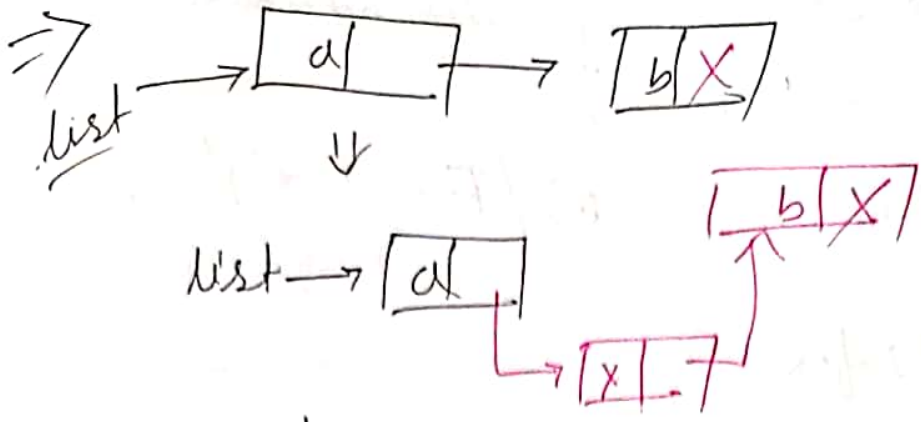
Loc of A[i][j] = base + W * [(j − LB$_2$) * m + (i − LB$_1$)]

= 1000 + 4 * [(6 − 6) * 11 + (0 − (−3))]

= 1149

6) Consider a singly linked list of n elements (n > 1) The run time complexity to insert a new node after first node in the list is

(a) O(1) (b) O(log n) (c) O(n) (d) none



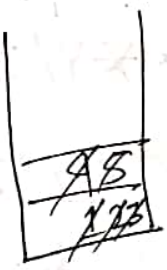
* . newnode → link = list → link
  list → link = newnode
  } 2 statement

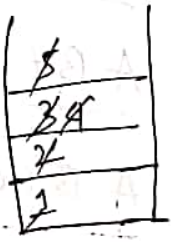
→ (a). O(1) ✓

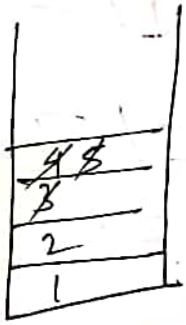
---

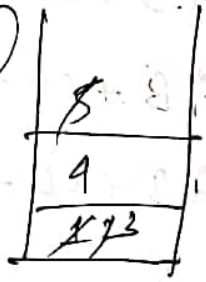
7. Which of the following permutation can be obtained in the output (in the same order) using a stack assuming that the input is the sequence 1 2 3 4 5 in that order?

(a) 1 2 4 5 3
(b) 4 5 3 1 2
(c) 3 5 4 2 1
(d) 1 2 5 3 4

⇒ (a)  1 2 4 5 3 ✓

(c)  3 5 4 2 1 ✓

(b)  4 5 3 ✗
We can't pop out 1 before 2

(d)  1 2 5 ✗

8 Convert following infix notation into prefix and postfix

⇒ (A+B) * (C-D) / F - X * Y / Z

⇒ Prefix

⇒ +AB * (C-D) / F - X * Y / Z

⇒ +AB * -CD / F - X * Y / Z

⇒ *+AB-CD / F - X * Y / Z

⇒ /*+AB-CDF - *XY / Z

⇒ /*+AB-CDF - /*XYZ

⇒ -/*+AB-CDF/*XYZ

Postfix

⇒ AB+ * (C-D) / F - X * Y / Z

⇒ AB+ * CD- / F - X * Y / Z

⇒ AB+CD-* / F - X * Y / Z

⇒ AB+CD-*F/ - X * Y / Z

⇒ AB+CD-*F/ - XY*/Z

⇒ AB+CD-*F/ - XY*Z/

⇒ AB+CD-*F/XY*Z/-

2. Convert following infix into prefix & postfix

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

⇒ prefix will be

⇒ $a+b*c-d\wedge e\wedge f$

⇒ $a+b*c-d\wedge \wedge ef$

⇒ $a+b*c-\wedge d\wedge ef$

⇒ $a+*bc-\wedge d\wedge ef$

⇒ $+a*bc-\wedge d\wedge ef$

⇒ $-+a*bc\wedge d\wedge ef$

postfix will be

⇒ $a+b*c-d\wedge e\wedge f$

⇒ $a+b*c-d\wedge ef\wedge$

⇒ $a+b*c-def\wedge\wedge$

⇒ $a+bc*-def\wedge\wedge$

⇒ $abc*+-def\wedge\wedge$

⇒ $abc*+def\wedge\wedge-$

10| Convert following notation into infix

⇒ A B C D − * + E /

⇒ This is a postfix notation we have to scan left to right

A B (C − D) * + E /

⇒ A (B * (C − D)) + E /

⇒ (A + B * (C − D)) E /

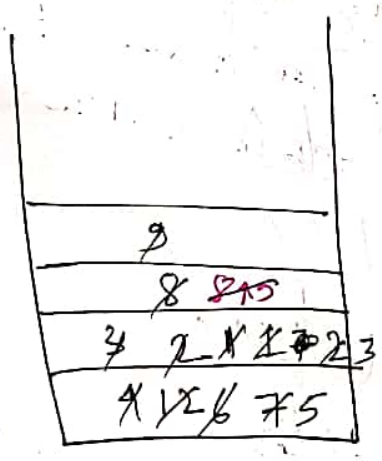
⇒ (A + B * (C − D)) / E

bracket will be written as same meaning

11| Evaluate following expression using Stack

4 3 * 2 / 1 8 9 ↑ ↑ + 2 − 3 +

⇒



4 * 3 = 12

12 / 2 = 6

8 ↑ 9 =

1 ↑ (8 ↑ 9) = 1

6 + 1 = 7

7 − 2 = 5

5 + 3 = 8

12. Void fun (struct node * list)
{
if (! list → link)
return list
return fun (list → link) ;
}

What does the above function do on singly linked list?

⇒ add of last node for non empty list.
or
null point dereferencing error for empty list

13. Consider a binary tree which has 60 internal nodes with degree 1 and total 90 leaf nodes. Total number of nodes in the tree is

⇒ $i_1 = 60$ $l_n = 90$

we know = $i_2 + 1 = l_n$

or $i_2 = l_n - 1 = 90 - 1 = 89$

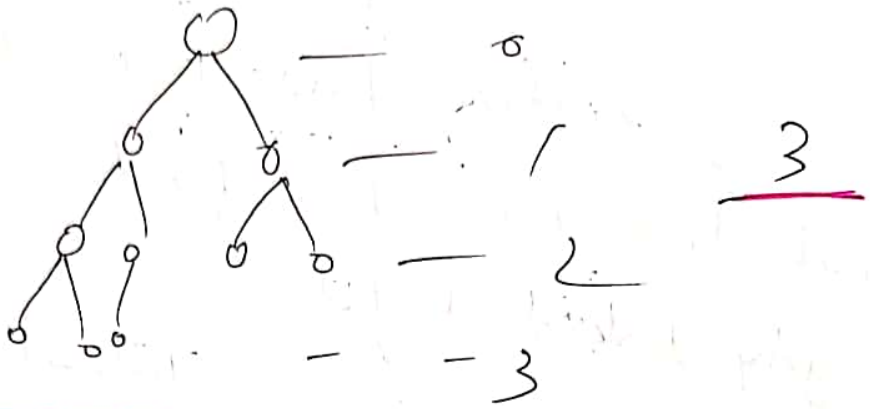
∴ Total node = $i_2 + i_1 +$ leaf node

$= 89 + 60 + 90 = 239$

14] The maximum height of AVL tree with 10 nodes is ____

Note — Single node height 0

⇒



3

15] Consider a sorted array of integer of size n with duplicates elements. You have given an element k. What is time complexity to find the element k is appeared at least n/4 times in array (a) O(n) (b) O(log n) (c) O(1) (d) none

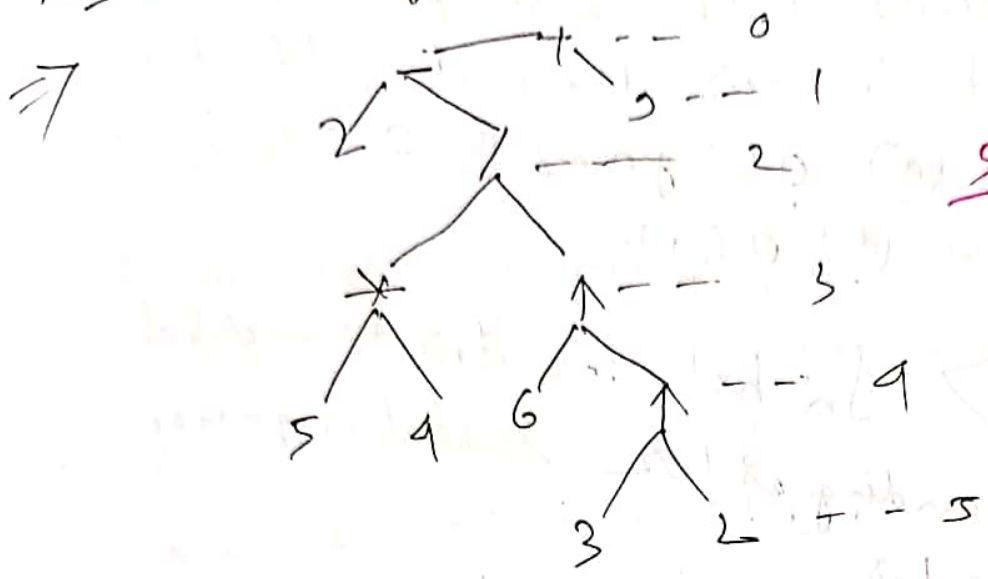
⇒ Obtain first appearance of element k at index i with the help of modified binary search (as array is sorted) then we will check if (A[i+n/4-1]==k) then return true

→ (b) O(log n)

16. Draw the expression tree & calculate height

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

node = height of tree with single node 0



5 Ans

17. Consider a singly linked list of n elements and P be a pointer pointing to a node of the list. The run time complexity to insert a new node before the node pointed by P in the list is

(a) $O(1)$ (b) $O(\log n)$ (c) $O(n)$ (d) none

But we do not know its address

So we have to traverse the whole list almost to know its address so that we can update its link value.

(c) $O(n)$ ✓

18) Consider an array of n elements holding the preorder traversal of a height balanced bst with n nodes. The worst time complexity of the best possible algo to find inorder traversal of the tree is

(a) $O(\log n)$ (b) $O(n)$ (c) $O(n \log n)$ (d) $O(n^2)$

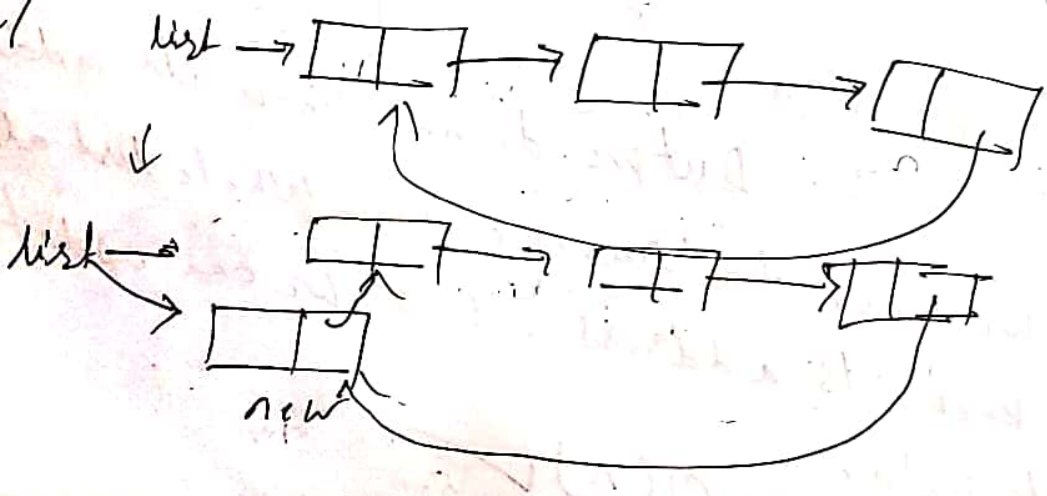
=> In bst inorder traversal = ascending order sorted sequence of preorder

best possible algo for sorting mergesort / heapsort $O(n \log n)$

(c) ✓

19) The time complexity to insert an element at the starting in a circular singly linked linked list is (a) $\theta(1)$ (b) $\theta(\log n)$ (c) $\theta(n)$ (d) $\theta(n \log n)$

=>



But We does not know the address of last node.
We have to do a traversal entirely to find that So it will take $\theta(n)$

→ (c) ✓

20] Consider an array A[-8:15] which is stored in memory from address 500 & each element take 4 location

(i) Total number of elements

(ii) What is address of A[9]

⇒ total number of elements = UB - LB + 1

$= 15 - (-8) + 1$

$= 24$

Address of $A[9] = Base + W * [i - LB_i]$

$= 500 + 4 \times [9 - (-8)]$

$= 568$

21] Which of the followings are true

(a) insertion in sorted array takes $O(n)$ time in worst case

(b) Deletion in sorted array takes $O(n)$ time in worst case

(c) Searchin in sorted array takes $O(\log n)$

time in worst case

(iv) Deletion in sorted array takes $O(\log n)$ in worst case

⇒ insertion → insert + shift others
$O(n)$ ✓

Deletion — Delete + shift
$O(n)$ ✓

Searching in sorted — $O(\log n)$ ✓
Binary

Deletion $O(\log n)$ ✗

---

22 Consider following postorder and inorder traversal of a binary tree

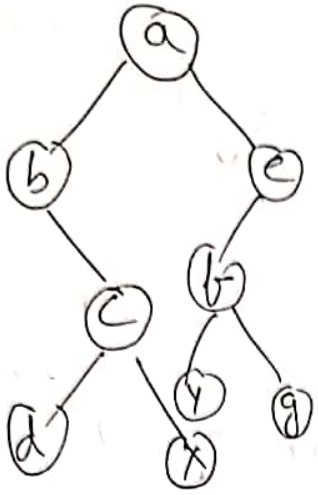
postorder – d x c b y g f e a

inorder – b d c x a y f g e

Which is preorder traversal

(a) a b c d e f y g x
(b) a b c d x e f g y
(c) a b c d x e f y g
(d) a b c d x f y e g

⇒ a is Root postorder - Root last
inorder - direction



d b c d x e y g

→ c ✓

23| How many different binary tree can be formulated with 4 distinct keys

Lets assume keys 1, 2, 3, 4

⇒

No of BT using n unlabelled nodes

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

using n distinct keys $= n! \times \frac{2nC_n}{n+1}$

$$= \frac{8C_4}{5} \times 4!$$

$$= \frac{8!}{4! \times 4!} \times 4!$$

$$= \frac{8 \times 7 \times 6 \times 5 \times 4!}{4! \times 4! \times 5} \times 4!$$

$$= 336$$

24| How many binary search tree can be formed with 9 distinct keys

⇒ $\frac{2nCn}{n+1} * 1$ ← bst has 1 possibility

$= 14$

25| Suppose an array contains n integer each of which is either 0 or 1 Then the array can be sorted in

(a) $O(n^2)$ (b) $O(n \log n)$ (c) $O(n)$ (d) $O(\log n)$

⇒ (i) count number of 0 & number of 1 in array in one traversal

then initially update 0's first then 1

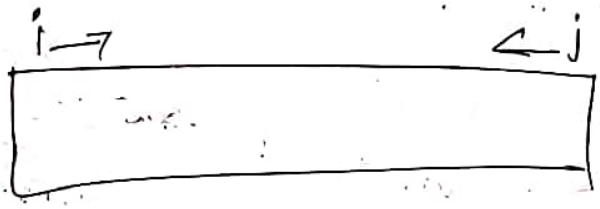
10000 11111

$O(n)$

one traversal

total 2 traversal

(ii)



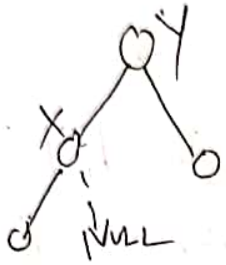
Stop I for 1
Stop j for 0
till $i \leq j$
} swap

only 1 traversal

$O(n)$

26| Consider a binary Search Tree T with distinct keys. The right subtree of a node x in T is empty and the node u has an inorder successor y then min no of ancestor x is having is _____
(every node is considered to be its own anchestor)

⇒



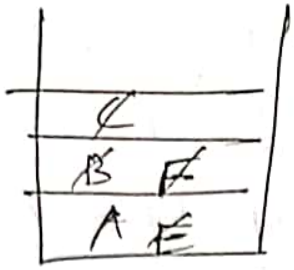
x anchestor is x ≠ y

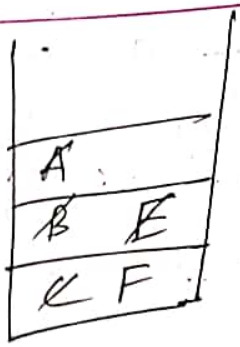
→ 2

x will be left side as it has inorder successor

27| A queue Q is implemented using 2 stack S1, S2 using best possible algorithms.
Consider following operation to be performed
Enqueue(A), Enqueue(B), Enqueue(C), Dequeue(). Enq(E) Enq(F), Deq(), Deq(), Deq()
To perform above operation minimum number of PUSH and POP operations required u and y respectively. value of x+y is ___ 2.

⇒





PUSH = 1+1+1+3+1+1+2 = 10

pop = 3+1+1+1+2+1 = 9

X*Y = 90

---

28) In an input restricted double ended queue following sequence of operations have been performed

Enq-rear(5), Enq-rear(7), Enq-front(9), Deq-rear(), Enq-rear(3), Enq-front(4), Enq-rear(8), Deq-front().

Assume that initially queue is empty. Restricted operation are having no operation impact on queue. The sum of elements at end ——— 2

⇒ input restricted double ended queue means

Enq — rear only

Deq — both end

5, 7, 3, 8

11

29. T or F

(a) Last symbol of inorder traversal of is not a leaf node

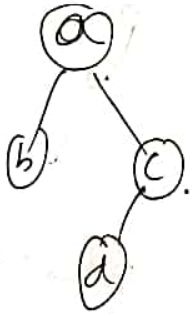
(b) First symbol of inorder traversal of binary tree is a leaf node

(c) Last symbol of a binary tree is a leaf node

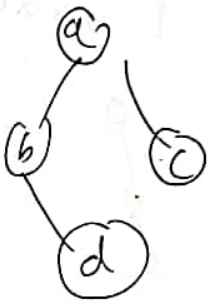
(d) First symbol of inorder traversal of binary tree is not a leaf node

⇒ All false because not always leaf node

In b a c



a b a d c



b d a c

30. Consider a binary tree which has X number of nodes with 1 child. Y number of nodes with 2 child. If total number of nodes in tree T is N then N = ?

(a) X+Y+1 (b) X+2Y+1 (c) 2X+Y+1 (d) 2X+2Y+1

⇒ total number of nodes in BT

$= I_2 + I_1 + \text{Leaf node}$

$= y + x + y + 1$

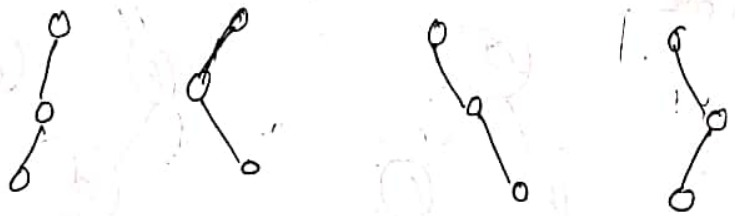
$= x + 2y + 1$

31] The total number of binary search tree can be constructed using 5 distinct keys So that height of the tree is always 4 are ____

Height of tree with single node 0

⇒ Height 4 means every ~~tree~~ level only 1 node can be kept

$n = 3$



$2^n - 1$

2^4

So if height 4 then $n = 5$

$\therefore 2^4 = \underline{\underline{16}}$

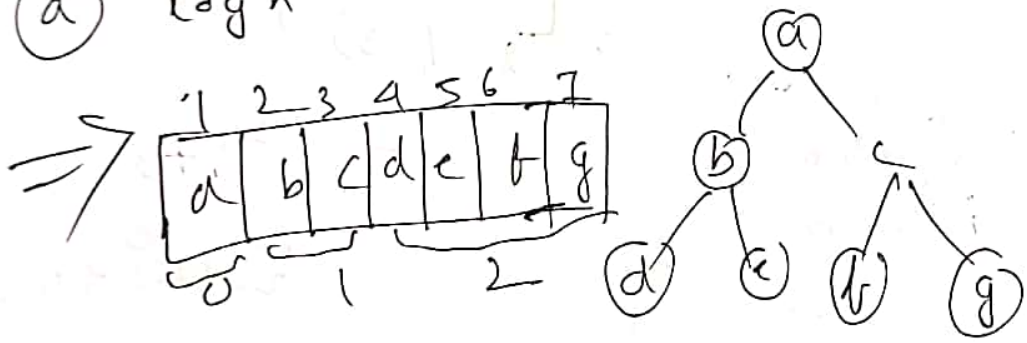
32. Consider an array representation of heap with n elements

- Root as index 1
- The left child of a node at index i is stored at index 2i
- The right child of a node at index i is stored at 2i+1.

If the root is at level 0 then the level of element A[i] is

(a) $\lfloor \log i \rfloor$ (b) $\log i + 1$ (c) $\lfloor \log (i+1) \rfloor$

(d) $\log n$



$\lfloor \log i \rfloor$ = floor value of $\log i$

$\log 7 = 2.$ something its floor value 2

→ (a) ✓

33] What is the value returned by the call of function(5)

```
int function (int n)
{
    if (n-1)
        return (2*function(n-1)+n);
    else
        return 0;
}
```

⇒ f(5)

└ 2 * f(4) + 5 = 41 (18)

└ 2 * f(3) + 4 (7)

└ 2 * f(2) + 3 (2)

└ 2 * f(1) + 2 (1)

2 * f(0) + 1

0

Sorting :=> A sorting algorithm is used to rearrange a given array or list elements according to a comparison operator on the elements.

6, 10, 7, 12, 9, → 6, 7, 9, 10, 12.

Types:-

- Bubble sort
- Selection sort
- Insertion sort
- Quick sort
- Merge sort
- Heap sort
- Radix sort
- Shell sort

Bubble sort: Bubble sort arrange N number of elements by placing the biggest element on proper position. It always arrange the data in ~~descending~~ ascending order.

7, 6, 3, 4, 1 → 1 3, 4, 6, 7

Procedure -

(1) 7, 6, 3, 4, 1

6 7 3 4 1

6 3 7 4 1

6 3 4 7 1

6 3 4 1 [7] ← sorted

(2) 6 3 4 1

3 6 4 1

3 4 6 1

3 4 1 [6] — sorted

(3) 3 4 1

3 4 1

3 1 [4] — sorted

(4) 3 1

1 [3] — sorted

For array of 5 element there will be 4 pass

(7) 6, 3 4 1

comparison will be (n-1) = 4 time

(6) 3 4

∴ Total comparison $n-1 + n-2 + n-3 + \dots + 1 = \frac{(n-1)(n-1+1)}{2} = \frac{n(n-1)}{2}$

∴ Complexity $= O(n^2)$

Selection Sort :=> Selection sort arrange N elements of array by placing the smallest element in proper position in case of ascending order arrangement.

Working –

- Set the first element as min
- compare min with every element and if any other is less then make it min
- After each iteration min is placed in the front of unsorted list. First element goes to the min's place
- For each iteration min value index incremented and rest repeated

20, 12, 10, 15, 2

(i) 20, 12, 10, 15, 2
20, 12, 10, 15, 2
20, 12, 10, 15, 2
20, 12, 10, 15, 2
20, 12, 10, 15, 2
total (n-1) comparis.
Swap 1 time.

(ii) 2, 12, 10, 15, 20
:
2 10 12 15 20
n-2 comparis. + Swap 1 time.

iii) 2, 10, 12, 15, 20 — total comparis. (n-3)

⋮

2 10, 12, 15, 20

already in plac.

iv) 2, 10, 12, 15, 20 — (n-4) comparis.

2, 10, 12, 15, 20

Total (n-1) pass

Time complexity = $O(n^2)$

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

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

Insertion Sort :→ it same like playing cards in hands (N-1) pass required

- It insert particular element in particular place.

Working: 7 6 3 4 1

6 7 3 4 1

6 3 7 4 1

3 6 7 4 1

3 6 4 7 1

3 4 6 7 1

3 4 6 7 1

3 4 6 1 7

3 4 1 6 7

3 1 4 6 7

1 3 4 6 7

an outer for loop

an inner while loop for checking smaller

(x) For sorting small arrays it is useful

. Take first element already sorted

. Take second element & store it separately in key.

. Compare key with left side element & swap

. Now the first two elements are sorted Take third element & compare it with the left of it, placed it just behind the element smaller than it.

Complexity → Average $O(n^2)$

Best (if already sorted) $O(n)$

- Stable & adaptive
- incremental algo as its sort one by one element at a time.

Quick sort :=>

- It is a divide & conquer algo

35 50 15 25 80 20 ... ∞

p check ≥ V

q check ≤ V So, it will encounter V

So no need of +∞

(10) pivot

80, 30, 60, 30, 20.

all > 10

6, 3, 5, 4, 3, 2, 1 (9) pivot

all < 9

5, 4, 3, 2, (10) 12, 13, 20

all < 10 all > 10

it is in sorted position.

lo pivot 16 8 12 15 6 3 9 5 ∞

i j

interchange

10 5 8 12 15 6 3 9 16 ∞

i j

10 5 8 9 15 6 3 12 16 ∞

i j

10 5 8 9 3 6 15 12 16 ∞

Pivot

exchange

j i

j cross i so pivot with

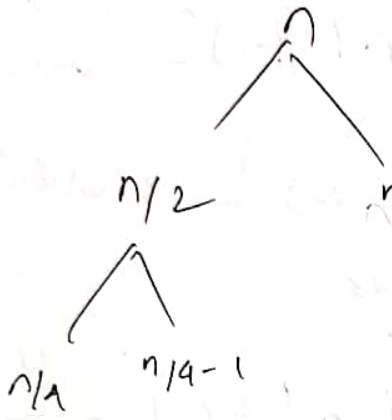
6 5 8 9 3 10 15 12 16 ∞

quicksort

Pivot

quicksort

recursively



Average complexity

$= n \log n$

for scanning recursive

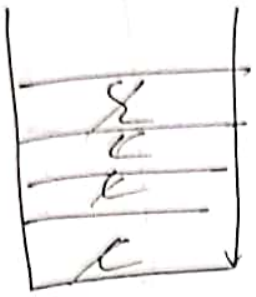
Worst case $= O(n^2)$

Time Complexity

- Binary Search – $O(\log n)$ – worst, average
  $O(1)$ – Best
- Linear Search – $O(n)$ – worst, average
  $O(1)$ – Best
- Quick sort – $O(n \log n)$ – Best, average
  sorted element – $O(n^2)$ – worst
  1, 2, 3, 4, 5
  pivot
- Merge sort – $O(n \log n)$ all
- Insertion sort – $O(n^2)$ – average, worst
  pre sorted Sorted element $O(n)$ – Best case
- Bubble sort – $O(n^2)$ – all
- Selection sort – $O(n^2)$ all
- Heap sort – $O(n \log n)$ – all

Parenthesis Matching

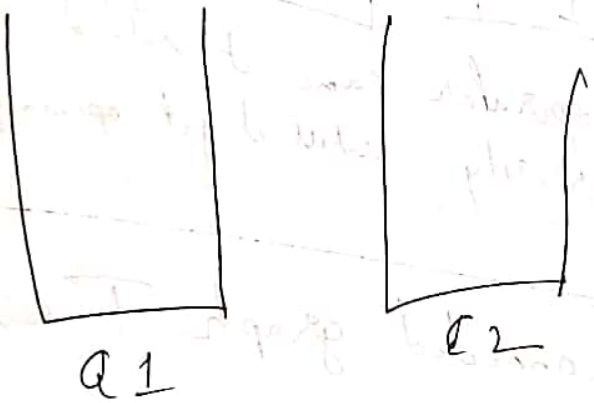
((({ 3)))



We push when we get diff parenthesis then we pop if similar right parenthesis can

- Stack is used for checking balanced parenthesis.

Stack using Queue



push(n)

⇒ • Add n to Q2
• Q1 → Q2 (element by element from bottom to top)
• Swap (Q1 ← Q2)

Time - O(n)

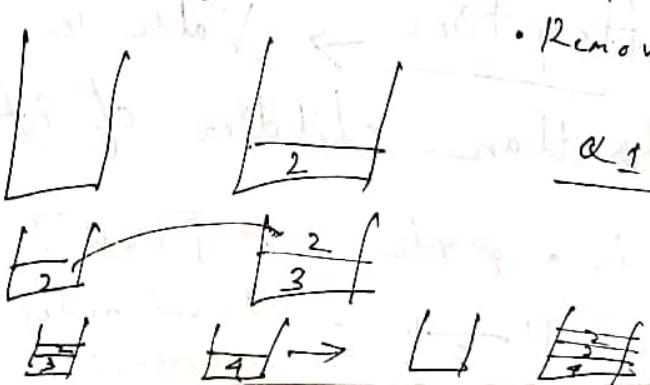
2 Stack needed

pop ( )

• Remove the top

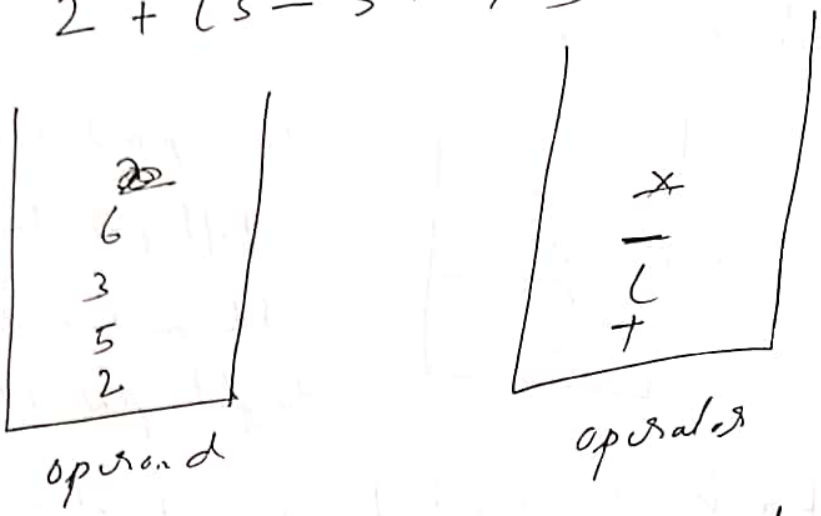
Q1

P(2)
P(3)
P(4)
Pop( )



Infix Evaluation using stack

2 + (5 − 3 * 6 / 2)



When divide came 3 & 6 poped and 3/6 = 18

When a operator came it solved higher priority pop
or same priority untill it get opening bracket.

(X) A connected graph T without any cycle is called tree, free tree, tree graph.

(X) Heap tree → Value in a node is greater than children of it

(X) in a graph $e = [U, V]$ mean $U \overset{e}{\text{---}} V$ — adjacent nodes

- $e = (u, v)$ 
  ↓
  e begins at u & end at v
  u is predecessor & v is successor.

(*) time factor for efficiency = counting number of key operation.

(*) Space Complexity — max memory needed by algorithm

- Complexity of average case = much more complicated to analyze than that of worst case.
- Single unit of values = Elementary item
- Entity has certain attribute & properties
- Abstract data type = Set of data values + associative operations

---

Equivalent tree = Same structure + Same value
Similar tree = Same structure + different value.