Data Structures:

storage or organization of data in such a way that it is easy to access, retrieve, operate on it efficiently.

Classified in two types.

1) Linear Data structure

An element can have "at most" two neighbours

i) Arrays → Address related ques.

ii) Linked Lists → code should be understood.

iii) Stacks

iv) Queues

2) Non-linear data structure.

An element can have more than two neighbours or, no neighbours at all. (Any no. of neighbours possible)

i) Tree — trees
binary tree
binary search tree
Heap
AVL tree

ii) Graph

3rd Topic - Hashing

{ (Sometimes a Non-linear data structure might BEHAVE like a Linear data structure. e.g. A tree where each node has only right child & does not have left child, then it behaves like a linked list.)

Arrays :→ Collection of similar type of elements.

→ Index starts from zero. (precisely).

→ Elements are stored one after another.

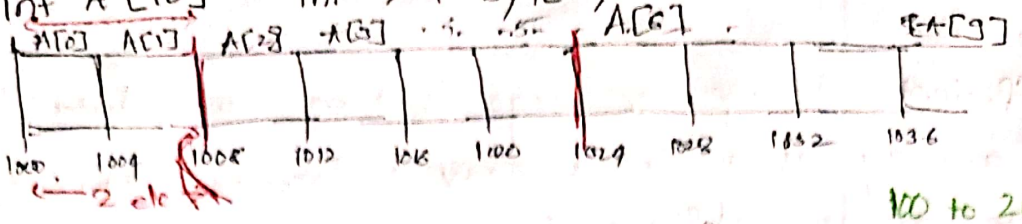
→ Relative Addressing is the major purpose.

→ Random Access (Advantage of Array) (using Index)
Access the element in constant time.

→ Cache Friendliness.

Static DS

→ int A[10] int → 4 bytes b.A. → 1000



Address of A[0] = 1000 (If the b.A. is 1000)

⟶ A[2] = ? ∴ 1000 + 8 = 1008

100 to 200 (including both)
⇒ 200 − 100 + 1 = 101
Last − First + 1

① How many elements are already filled before A[2] = 2

② size of each element = 4 bytes

③ memory filled already before A[2] = 2 × 4 = 8 bytes

Address [6] ⟵ Before A[6] = index 0 to 5 = 5 − 0 + 1 = 6 = last − first + 1

Before A[6] = 6 × 4 = 24 bytes

Addr (A[6]) = 1000 + 24 = 1024
b.A. + mem. Filled

In Theory ⇒ Index can start from any no.

e.g. A[−5 .... 5]
- lowest index
- Largest index

# of elements = last − first + 1
= 5 − (−5) + 1
= 11 elements

Q. A[−5 .... 5]
w = 4 bytes (size of each ele)
Base Address = 1000
Addd (A[1]) = ?

Ans: How many ele filled bf A[1]
= index : −5 to 0
= last − first + 1
= 0 − (−5) + 1 = 6 elements.

mem. filled before A[1] = 4 × 6
= 24 bytes

Addr. of A[1] = b.A. + mem. filled before A[1]
= 1000 + 24
= 1024 bytes //

★ Q. A[−20 .... 10], w = 2byte, B.A. = 1000, calculate (A[−5]) = ?

Elements already Filled before A[−5] = −20 to −6 (Not including −5 itself)
= −6 − (−20) + 1
= 15 //

size of each element = 2 bytes

mem. Already filled bf A[−5] = Elemnts filled * size = 15 × 2 = 30.

Addr. (A[−5]) = b.A. + m-filled
= 1000 + 30 = 1030

Elements Already filled before A[-5]

= -20 to -6

= -6 - (-20) + 1

= 15 elements

~~Negative~~ Negative Indexes are possible in Data structures. (Theoretically)

1024
p → pointer.

1023
p - 1 = Address backwarding

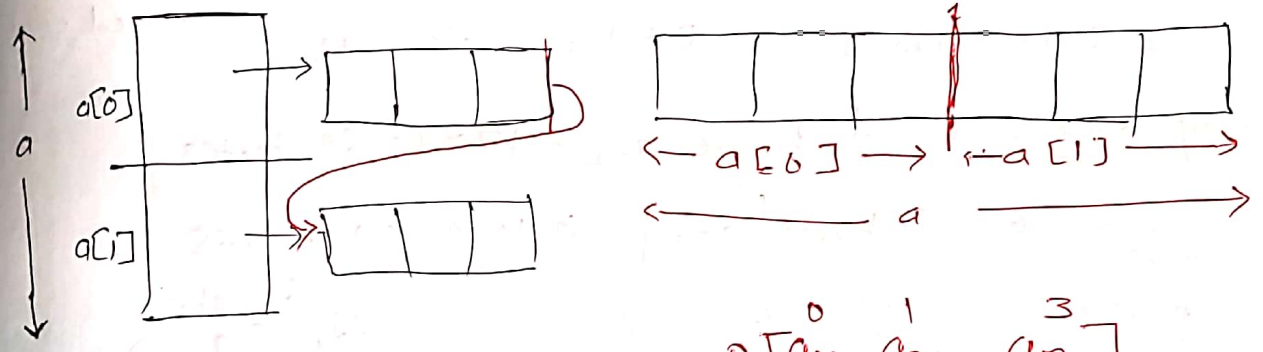
*(p-1) = Value decrement

p[-i] = *(p-1)

2D-Array:

int a[2][3]; — 0 } - Each index/no. in this dim represent 3 elements
1 }

a[0] = 3, a[1] = 3



	0	1	3
0	a_{00}	a_{01}	a_{02}
1	a_{10}	a_{11}	a_{12}

row a[0] ⇒ 3 elements

—11— [1] ⇒ —11—

2D-Array

- Row-wise store — Row-major order (RMO)
- Column-wise store — column-major order.

int a[3][4]

Each index = 4 elements

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

RMO

← row with index=0 → | row with index=1 → | row with index=2 →

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

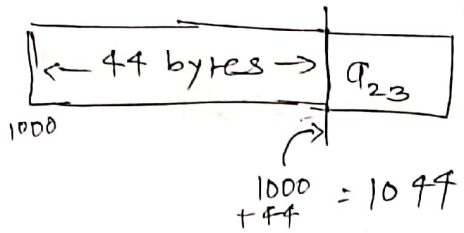
Add(a_{23}) = ?

① How many rows already filled before row with index 2 = 2

② No. of elements already filled before a_{23} in row with index 2 = 3 elements

2 rows ⇒ 2×4 elements.
3 elements + 3
11 elements already b4 a_{23}

w = 4 bytes, Elements before a_{23} = 11 elements.
Memory already filled before a_{23} = 11 × 4 = 44 bytes



∴ Addr. (a[2][3]) = 1044
-11-(a_{23})

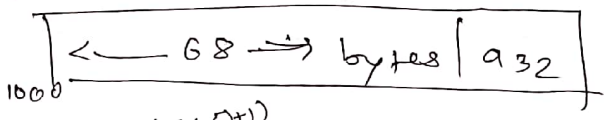
Q. int a[4][5], w = 4 bytes, B.A = 1000
add (a_{32}) (row index, column index)

→ How many rows filled b4 row with index 3 = 0 to 2
= 2 - 0 + 1 = 3 rows

Within row whose index is 3, the no. of elem. already Filled before a_{32}
= col with index 0 to 1
= 1 - 0 + 1 = 2 elements

Before a_{32} = 3 rows & 2 ele already filled
= 3 × 5 + 2 = 17 elements already filled.

∴ Memory already filled before a_{32} = 17 × 4 = 68 bytes.

∴ Addr ⇒ 

⇒ 1000 + 68 = 1068 is the addr. (a_{32})

Q. a[-5....5] [-3....3]
11 indexes (5-(-5)+1) 7 indexes (3-(-3)+1)
w = 2 bytes, B.A = 1000, add (a[i][1])

→ How many rows filled b4 row index 1 = -5 to 0
= 0 - (-5) + 1
= 6 rows.

Within row with index 1, # of elements filled before a_{11}
= col with index -3 to 0
= 0 + (+3) + 1 = 4 elements.

Before a_{11} = 6 rows & 4 elements already filled.
= 6 × 7 + 4 = 42 + 4 = 46 elements
(7 = # of elements/col. in each row)

Mem. already filled before a_{11} = 46 × 2 = 92 bytes. (2 = size)

∴ add (a[1][1]) = B.A. + Mem. filled before a_{11}
= 1000 + 92 = 1092

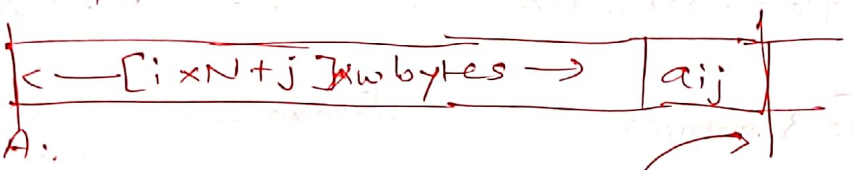
a[M][N] – 0 to N-1 add (aij) (Valid for index from 0 only)
0 to M-1

① rows already filled before row with index i = 0 to i-1
= i-1-0+1
= i Rows
↓
i × N elements

② within ith row, elements already filled before aij = col with index 0 to j-1
= j-1-0+1 = j elements

Total elements already filled = [i × N + j]

size of each ck = w bytes.
mem. already filled = [i × N + j] × w bytes.



add (aij) = BA + (i × N + j) × w

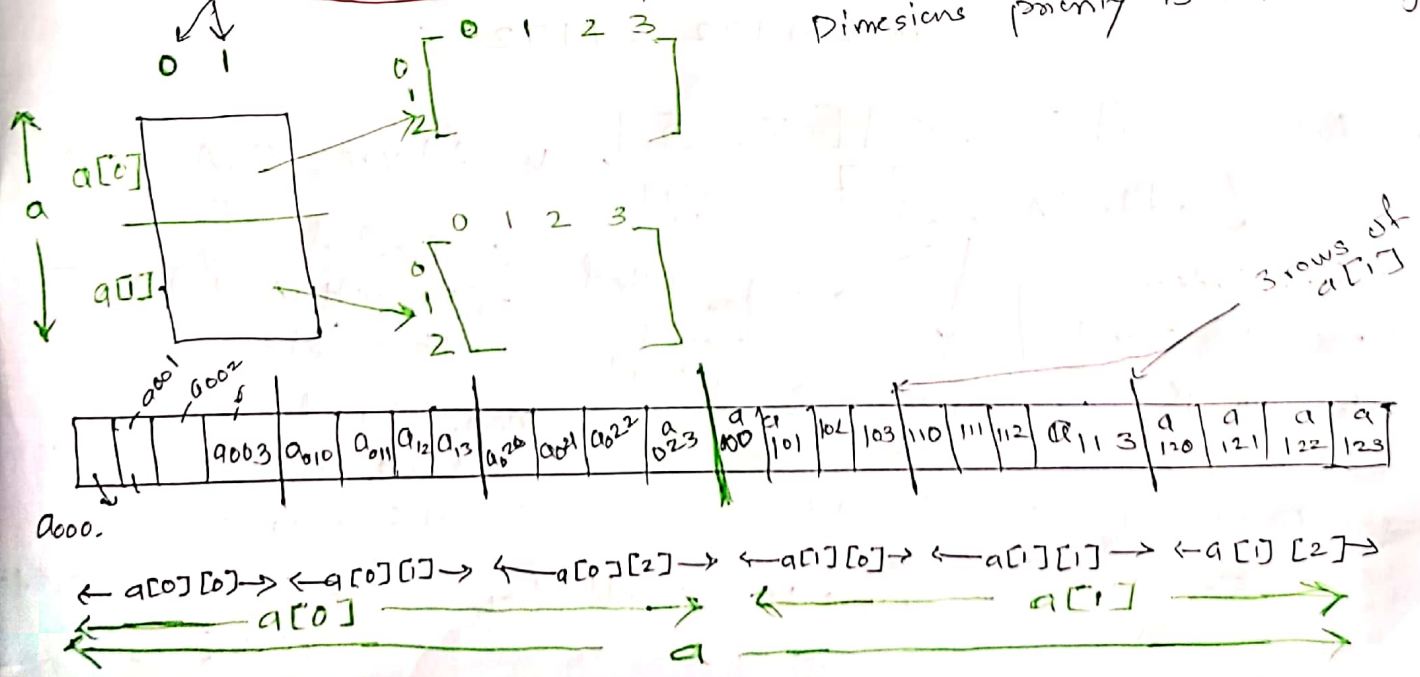
for index 1 ; i becomes i-1
— 11 — 2 ; i " i-2

Similarly for j

3-D Array

int a[2][3][4] 2D Array

a[2][3][4] Row-Major Order.
Dimesions priority is left to Right



int a[3][4][5]

Every index in this dimension represent = 4×5 elements

Every index/No. in this dimension represent = 5 elements

This dimension represent one element

∴ size of this dimension is the product of it's next dimension (i.e. 4×5)

Q. RMO, w = 4 bytes, B.A. = 1000

a[3][3][5]

Add (a223)

Soln: → How many matrices filled before index 2 in 1st Dim.

= 0 to 1

= 1 − 0 + 1 = 2 matrices

Size of each matrix = 3×5 ⇒ 2×3×5 elements total filled

How many elements filled in 2nd Dimension

= 0 to 1

= 1 − 0 + 1 = 2×5 elements.

In 3-D

= 0 to 2 = 2 − 0 + 1

= 3 elements.

Total ele = $\underbrace{(2\times3\times5)}_{1\text{-D}} + \underbrace{(2\times5)}_{2\text{D}} + 3 = 43$ elements.

Total mem. filled = 43 × 4 = 172 bytes.

Addr (a223) = B.A. + Total memory filled before a223

= 1000 + 172 = 1172.

Q. A[−5...5][−3...3][−5...5], w = 2 byte, B.A = 1000

addr ($A_{0,0,0}$) 3 − (−3) + 1 = 7, 5 − (−5) + 1 = 11

∴ Every index represent = 7×11

index filled = −5 to −1

= −1 − (−5) + 1

= 5 ⇒ 5×7×11 elements

index filled = −3 to −1

= −1 − (−3) + 1

= 3 ⇒ 3×11 elements

index filled = −5 to −1

= −1 − (−5) + 1

= 5

∴ Total ele = (5×7×11) + (3×11) + 5

= 423 ele.

Mem. filled = 423 × 2 (size)

= 846 bytes.

∴ addr($A_{0,0,0}$) = B.A. + 846 = 1000 + 846

= 1846

Q. A[-5...5][-10...10][-3...3][-5...5], w = 2 bytes
21 7 11
BA = 1000
add(0213)

Each index represent 21 3D matrices
→ In 21 3D matrices, each represent 7 2D matrices
→ In 7 2D " , each row represent 11 ele.

add(A0213)

-5 to -1 : -1-(-5)+1 = 5
-10 to 1 : 1-(-10)+1 = 12
-3 to 0 : 0-(-3)+1 = 4
-5 to 2 : 2-(-5)+1 = 8

Total ele =
5×21×7×11
+ 12×7×11
+ 4×11
+ 8
= 9061 ele

mem. filled Total = 9061×2
= 18122 bytes.

add(A0213) = 1000 + 18122
= 19122

RMO → 2-D

2D Array – CMO

int a[2][3]

	0	1	2
0	a_{00}	a_{01}	a_{02}
1	a_{10}	a_{11}	a_{12}

Within column, elements are filled row-wise.

col. with index = 0		col. with index = 1		col. with index = 2	
a_{00}	a_{10}	a_{01}	a_{11}	a_{02}	a_{12}

column-wise

Q. int a[3][4], add(a_{23}), w = 4 bytes.

How many columns already filled before col. with index 3.
= 3 (0 to 2)
(2-0+1)

After 3 columns, 2 elements, a_{23} is stored.

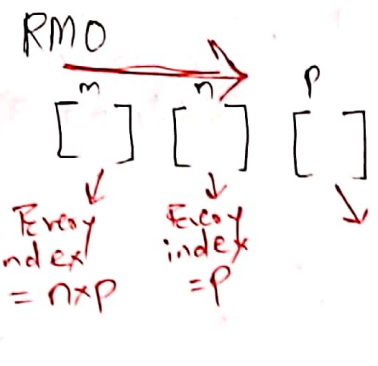
Every column ⇒ 3 elements/rows

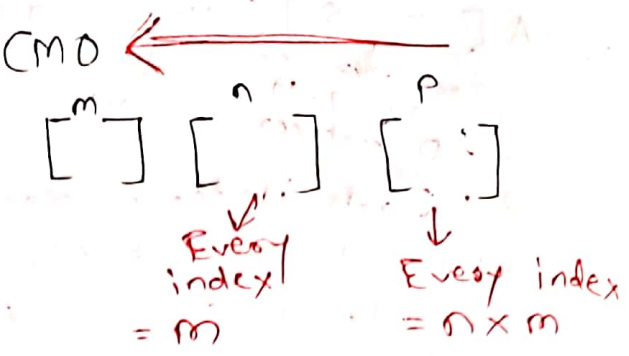
Total ele = (3×3)+2
= 9+2
= 11 elements

mem. Already filled
= 11×4
= 44 bytes.

add(a_{23})
= B.A. + M.F.
= 1000+44
= 1044 //

Within col. index 3
How many elements (rows) already filled before a_{23}
= 0 to 1 = 1-0+1 = (2)


RMO
m
n
p
Every index = n×p
Every index = p


CMO
m
n
p
Every index = m
Every index = n × m

Sparse Matrix

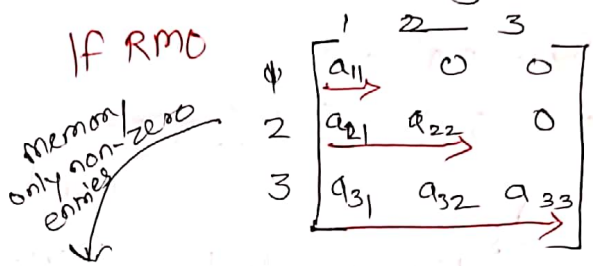
Matrix with more no. of zero's.

i) Lower Triangular matrix
ii) Upper — " —
iii) tri-diagonal matrix
iv) Identity matrix
v) scalar matrix.

i) Lower Triangular matrix — LTM is a square matrix ($i=j$) where the entries above the main diagonal are zero.

$$A_{ij} = 0, \quad i < j$$

If RMO



a_{11}	a_{21}	a_{22}	a_{31}	a_{32}	a_{33}

Q. a[5][5], add (a_{43}), w = 4 bytes.

How many rows already filled before row with index 4
= 1 to 3
= 3 ~~rows~~
= 1, 2, 3 index
1 + 2 + 3 = 6 elements.

Within row with index 4, columns/elements filled before
a_{43} = 1 to 2
= 2 − 1 + 1 = 2 elements

Total = 6 + 2 = 8 elements already filled.

mem. filled
= 8 × 4
= 32 bytes.

add (a_{43})
= B.A. + no. of
= 1000 + 32
= 1032

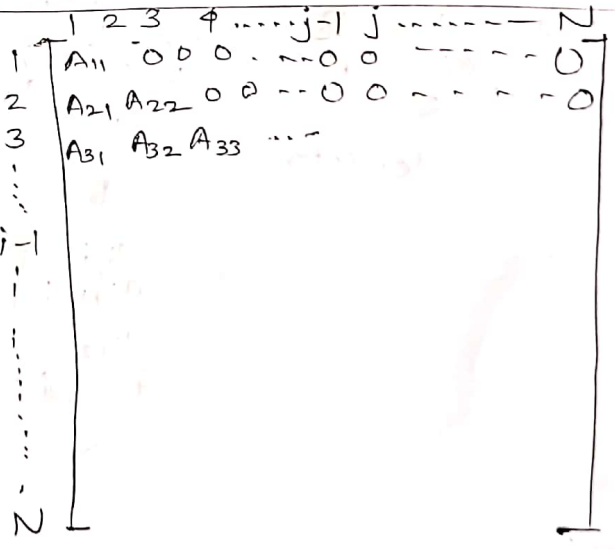
LTM ⇒ N × N

add (A_{ij})

Rows already filled
= index 1 to i−1
index 1, 2, 3, ... i−1
1 + 2 + 3 ... + (i−1) = S_n

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

$S_N = \frac{N(N+1)}{2}$



add (A i j) → within ith row, elements already filled before A ij

= col. with index 1 to j−1

= j−1−1+1 = (j−1) elements.

Total ele. already filled = $\frac{i(i-1)}{2} + (j-1)$. size = w bytes.

mem. already filled before A ij = $\left[\frac{i(i-1)}{2} + (j-1)\right]$ w bytes.

addr. (A i j) = B.A. + Memory filled before A ij

3. LTM, w = 2 byte, BA = 1000

RMO

A[−5....5] [−7...3]

add (A 1, −2) → within row index 1,

Rows filled
= −5 to 0
= 6 rows
1+2... 6
= $\frac{6\times7}{2}$ = 21 ele

col. filled / ele:
= −7 to −3
= 5 ele

size = 2 bytes

Total ele = 21 + 5 = 26 ele.

Mem. filled = 26 × 2 bytes
= 52 bytes.

add (A 1, −2) = B.A. + M f
= 1000 + 52
= 1052

LTM in CMO.

← 1st col	← 2nd →	← 3rd →	4th
a_{11} a_{21} a_{31} a_{41}	a_{22} a_{32} a_{42}	a_{33} a_{43}	a_{44}

	1	2	3	4
1	a_{11}	0	0	0
2	a_{21}	a_{22}	0	0
3	a_{31}	a_{32}	a_{33}	0
4	a_{41}	a_{42}	a_{43}	a_{44}

add (a_{43})

col. already filled
= index 1 to 3
= 4 + 3
= 7 elements

Within 3rd col, or intended col, ele. already filled before a_{43}
= 4 − 3 = 1 element

$a_{i,i}$
$a_{i+1,i}$
$a_{i+2,i}$
$a_{i+3,i}$ ← intended

In CMO, row − col gives the no. of elements filled before intended element in intended col
row − col = i + 3 − i = 3

Total el = 7 + 1 = 8
size = 2,
M.f. = 8 × 2 = 16 bytes
add (a_{43}) = B.A. + Mf
= 1000 + 16
= 1016

N x N

add (a;j)

→ col. already filled

index 1, 2, 3 .... j-1

index 1 → N = N-(1-1)

— " — 2 → N-1 = N-(2-1)

3 → N-2 = N-(3-1)

4 → N-3 = N-(4-1)

⋮

Only when indx starts from 1.

j-1 → N-(j-2) = N-j+2

Sn

$[N + (N-1) + (N-2) + \ldots + (N-j+2)]$

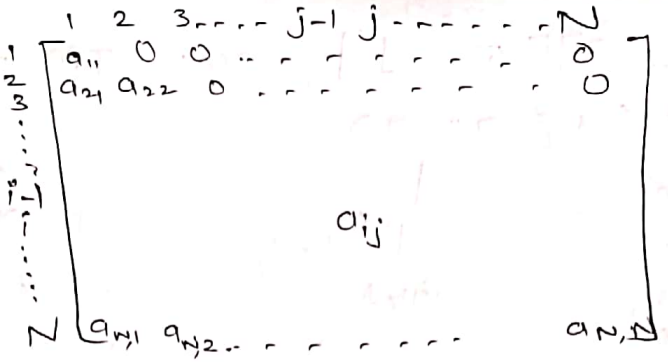
first term

last term

Sum of n terms of A.P. = $\frac{N}{2}$ [first term + last term]

Here N = j-1

$\therefore = \frac{j-1}{2}[N + (N-j+2)]$



→ Within col. index j, elements already filled before aij

= row-col^m = (i-j) (valid only when i & j starts from same # index)

→ Total elements already filled = $\frac{j-1}{2}[N + (N-(j-2))] + (i-j)$

$= \frac{j-1}{2}[2N-(j-2)] + (i-j)$

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

Mem. filled = $\left[N(j-1) - \frac{(j-1)(j-2)}{2} + (i-j)\right] w$

add(a;j) = B.A. + Mf

main diagonal \, cross diagonal /

7×7 — 7×7

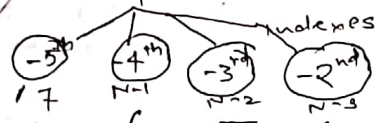
Q. A[-3....3][-5....1], CMO, LTM, w = 1 byte, BA = 1000

add (A[3][-1])

Within col^mn with index -1, how many rows already filled
= i - j = 4 - (-3 - (-1))
= 4

But row & col^mn indexes do not start from same number. Row starts from -3. whereas col^mn starts from -5
diff. bet^n two is 2. Subtract this, we get 4 - 2 = 2 elements filled.

col^mn already filled
= -5 to -2
= -2 - (-5) + 1
= 4 col^mns



N rows filled i.e. 7 + 6 + 5 + 4 elements

$\frac{N}{2}[7+4] = \frac{4}{2}[11]$

= 22 elements

(As -2nd indexed col^mn has 4 elements, -1^st has 3)

	-5	-4	-3	-2	-1	0	1
-3	×	o	o	o	o	o	o
-2	×	×	o	o	o	o	o
-1	×	×	×	o	o	o	o
0	×	×	×	×	o	o	o
1	×	×	×	×	×	o	o
2	×	×	×	×	×	×	o
3	×	×	×	×	X	×	×

↑ Addr. to be found

Total ele filled before $A_{3,-1}$ = 22 + 2 = 24 ele.

Total memory filled before $A_{[3][-1]}$ = 24 bytes

Addr(A[3][-1])
= B.A + MF
= 1000 + 24 = 1024 //

(Here row index starts from -3, col^mn index from -5.
i.e row starting index > col^mn starting index by 2 units
∴ We subtract 2. If row starting index < col^mn starting index, we add the difference.) In LTM, CMO.

Upper-triangular Matrix

Square matrix where entries/elements below main diagonal are zero.

$$a_{ij} = 0, \quad i > j$$

$$\begin{matrix} & 1 & 2 & 3 \\ 1 & a_{11} & a_{12} & a_{13} \\ 2 & 0 & a_{22} & a_{23} \\ 3 & 0 & 0 & a_{33} \end{matrix} \rightarrow \text{RMO}$$

1st row			2nd row		3rd row
a_{11}	a_{12}	a_{13}	a_{22}	a_{23}	a_{33}

Q. 5×5, add (a_{34}), RMO

Rows filled = 1 - 2 = 2 rows

$(5+4)$ ele = 9

Within intended row 3, ele before a_{34} = 4 - 3 $(j-i)$ = 1 elements

Total ele = 10

MF = 10 × w

add (a_{34}) = B A + MF

UTM in RMO N×N

add (a_{ij})

Rows filled = index 1, 2, 3, ... $(i-1)$

N, N-1, N-2, ... N-(i-2)

1st ⇒	N
2nd ⇒	N-1
3rd ⇒	N-2
⋮	
i-1 ⇒	N-(i-2)

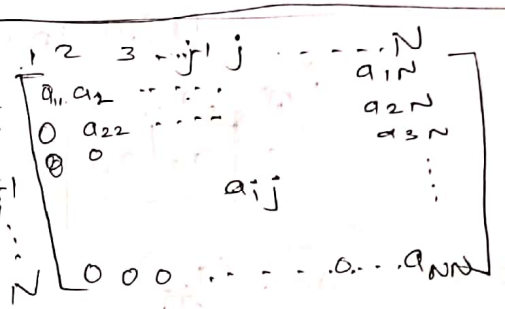
$N + (N-1) + \dots (N-i+2)$

$i-1$ entries (n)

$\frac{n}{2}[\text{first} + \text{last}]$

$\frac{i-1}{2}[N + (N-i+2)]$

Row already filled: $i-1$



Within ith row, elements already filled a_{ij} = $(j-i)$

(Works only when i & j starts from same index)

Total ele already filled = $\frac{i-1}{2}[2N-(i-2)] + (j-i)$

$= (i-1).N - \frac{(i-1)(i-2)}{2} + (j-i)$

M.F. = Total ele × w

add (a_{ij}) = B.A. + MF.

Q. A [-12 ... 12] [-12 ... 12], w = 2 bytes, BA = 1000.

25 elements, 25 elements

add UTM, RMO.

Here both indexes start from same.

∴ add $(A_{0,3})$

add ($A_{0,3}$)

Rows filled
= -12 to -1
= -1 - (-12) + 1
= 12 rows

1st row → 25 → 26
2nd " → 24 → 26
3rd " → 23 → 26
⋮
12th " → 14 → 26

$\Rightarrow \frac{12}{2}[25+14]$
$= 6 \times 39$
$= 234$

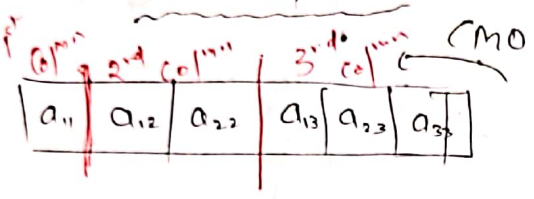
In row 0, elements before $A_{0,3}$
$= 3 - 0 = 3$

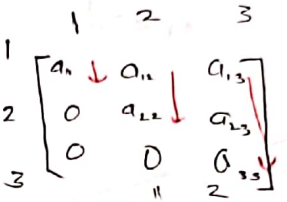
Total ele = 234 + 3
= 237

Mem. filled = 237 × 2 = 474 bytes

addr. ($A_{0,3}$) = 1000 + 474 = 1474 //

UTM in CMO

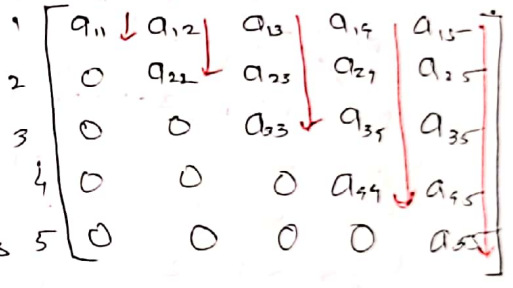




Q. 5×5, add (a_{45})

Within col 5, ele before a_{45}
= row index 1 to 3
= 3 - 1 + 1
= 3 elements

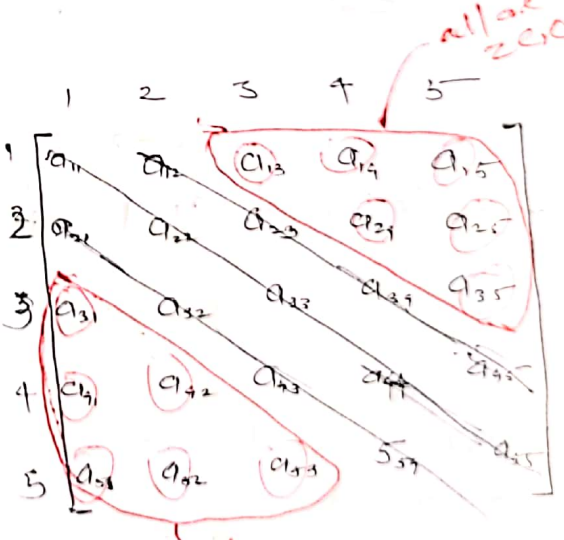
col^ms already filled
= index
= 1, 2, 3, 4
= 1 + 2 + 3 + 4 = 10 elements



Tri-diagonal matrix

- A square matrix
  - ⇒ Main diagonal
  - ⇒ diagonal just above main diagonal
  - ⇒ D just below main diagonal

} except them, all entries are 0



of elements in n×n tridiagonal matrix

in first row ⇒ 2
in last " ⇒ 2
in remo " ⇒ 3
remaining rows = (n-2)

Total elem = 2 + 2 + (n-2).3
(2: 1st row, 2: last row, (n-2): remaining rows, 3: elements in each remaining row)

= 2 + 2 + 3n - 6
= 3n - 2

∴ In 5×5 matrix, 3n−2 = (3×5)−2 = 13 elements

a_{11}	a_{12}	a_{21}	a_{22}	a_{23}	a_{32}	a_{33}	a_{34}	a_{43}	a_{44}	a_{45}	a_{54}	a_{55}

Q. add ($a_{4\ 5}$)

rows already filled = 3

1st ⇒ 2
2nd = 3
3rd = 3
8 ele.

Within 4th row, ele/cols before a_{45} = (5 − 4) + 1 = 2 elements

Total ele = 8 + 2 = 10

Generalize

N×N - midiagonal, RMO, add (a i j)

Within ith row, elements filled before a i j = j − i + 1

Rows filled = 1 to i−1 = (i−1) rows

Out of these (i−1) rows:
1st row = 2 el
rem. (i−2) rows = 3 el.

Total ⇒ 2 + 3(i−2) ⇒ 2 + 3i − 6 ⇒ 3i − 4

Total ele before a i j = 3i − 4 + j − i + 1 = 2i + j − 3

Disadvantages of array — 1) Pre allocate size

2) Insertion/deletion in array is expensive.

No space ✓

n = 10, size = 10

10	20	30	40	50					
0	1	2	3	4	5	....			9

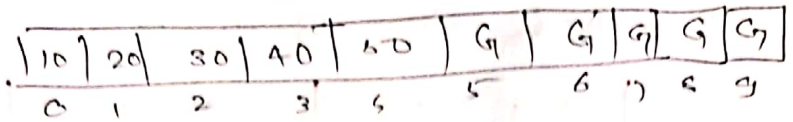
Q. Insert 100 at A[1]

Ans
A[5] = A[4]
A[4] = A[3]
A[3] = A[2]
A[2] = A[1]
A[1] = 100

```
For(i=n-1; i>=index; i--)
    A[i+1] = A[i]
A[index] = element
```

Time complexity = no. of operation = n worst case
∴ O(n)

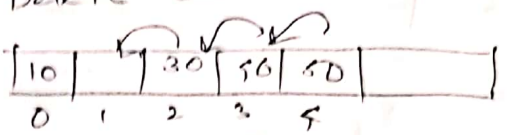
n = 5



O(n)

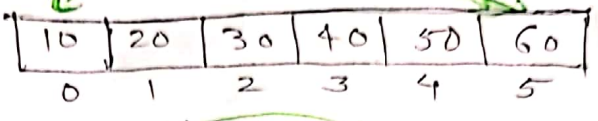
Delete last element
↳ No problem (OR)
n = n - 1;

Delete ele at index=1



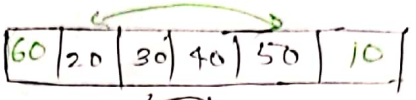
~~Delete~~

Reverse an Array



step:

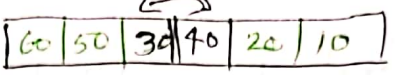
swap(A[0], A[5])



2 ele

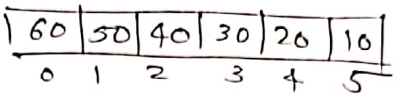
n/2 swaps

swap(A[1], A[4])



2 ele

swap(A[2], A[3])



2 el

In every swap, 2 elements take position, i.e. Here 6 elements were present, so we needed 3 swaps.

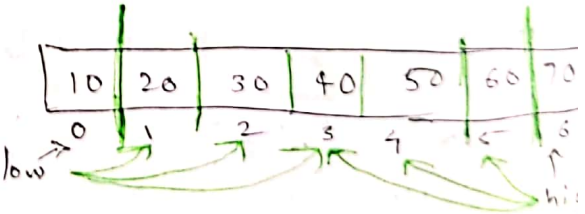
∴ For n elements, in n/2 swaps, the array will be reversed

In e swap(A[i], A[j]) i + j = n-1 ∴ j = n-i-1

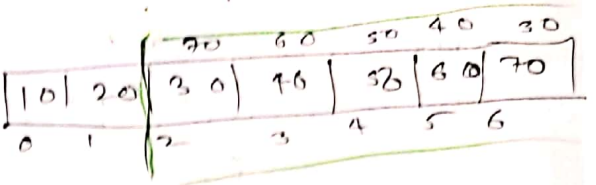
i: 0, 1, 2
j: 5, 4, 3

Code:
```
for (i=0; i< n/2 ; i++)
    swap(A[i], A[n-i-1]);
```

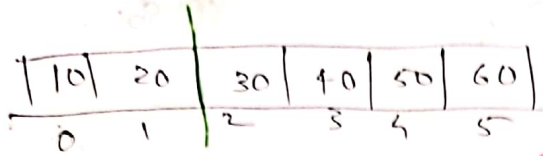
```
Void reverse (int arr[], int low, int high)
{
  While (low < high)
  {
    swap(A[low], A[high])
    low++;
    high--
  }
}
```



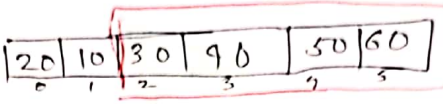
```
void main()
{
    int a[] = {10,20,30,40,50,60,70};
    reverse(a, 2, 6);
}
```



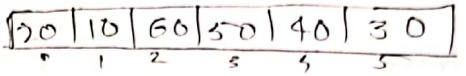
a. a



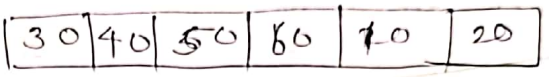
reverse(a, 0, 1); →



" (a, 2, 5); ⇒



" (a, 0, 5); ⇒



Array is rotated 2 times
d = 2

---

reverse(a, 0, d-1); (First reverse "d" elements)
reverse(a, d, n-1); (reverse remaining ")
reverse(a, 0, n-1); (reverse entire array)

Rotate D (=d) times

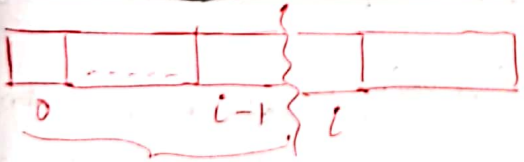
Largest, Second Largest ele. in array.

1st approach → 2 traverse — can we do it in 1 traversal

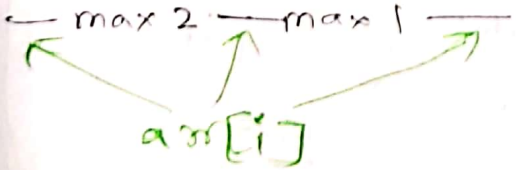
max = -∞;

```
① for (i=0; i<n; i++)
  {
    if (arr[i] > max)
        max = arr[i];
  }
```

```
② for (i=0; i<n; i++)
  {
    if (arr[i] > max2
        &&
        arr[i] != max)
        max2 = arr[i];
  }
```



We know max 1 & max 2



arr[i] → 1) If it is larger than max. till now.

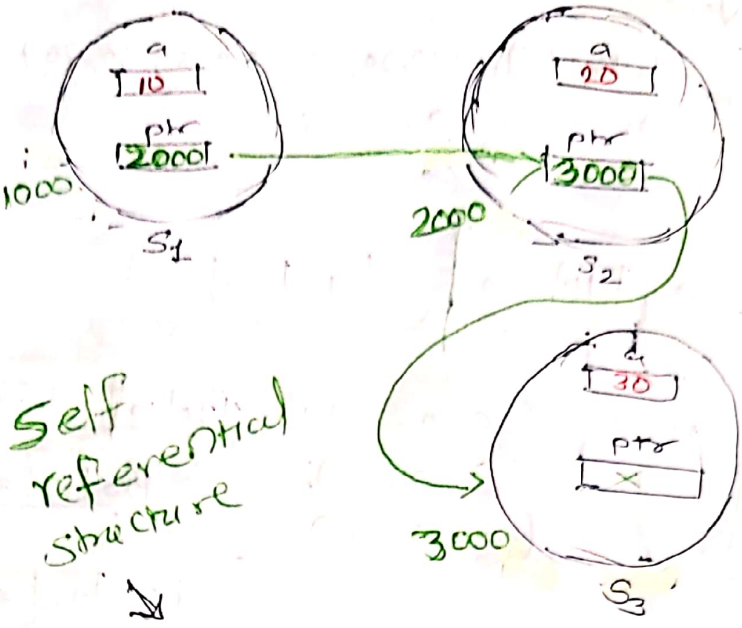
```
If (arr[i] > max1)
    max2 = max1;
    max1 = arr[i];
```

2) else if (arr[i] > max2) then

```
    max2 = arr[i];
```

Linked Lists

```
struct Student {
    int a;
    struct student *ptr;
};
void main()
{
    struct student s1, s2, s3;
    s1.a = 10;
    s2.a = 20;
    s3.a = 30;
    s1.ptr = "Address of
              struct student
              type variable"
```



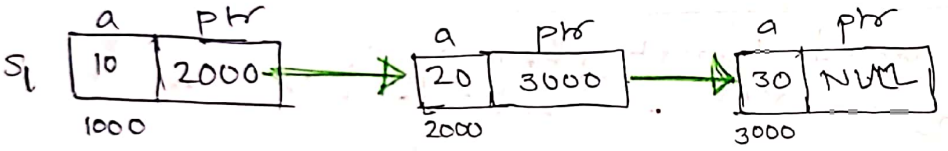
Self referential structure

A structure where a member is a pointer to same type of structure variable

```
s1.ptr = &s2;
s2.ptr = &s3;
s3.ptr = NULL;
```

$S_1.ptr = \&S_2$;
$S_2.ptr = \&S_3$

Advantage: Increase/decrease size at Runtime.



Linked List

It is a linear data structure which is a collection of elements called nodes, in which every node contains:

i) data ii) address of next node.

∴ Node, itself is a collection of diff. types of elements.

```
struct Node{
    int data;
    struct Node *next;
};
Void main() {
    struct Node s;
    Insert(100);
}
```

static, local variable (compile time)

```
Void Insert(int key)
{
    struct Node s1;
    s1.data = 10;
}
```

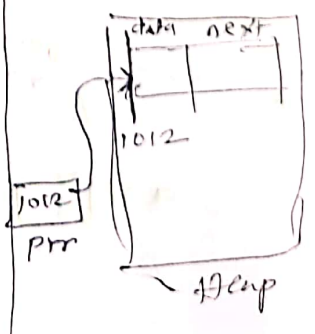
local

After this s1 is vanished. problem

∴ Node → malloc help → throughout program

Dynamic Allocation

```
struct Node *ptr;
ptr = malloc(sizeof(struct Node));
```



struct Node.

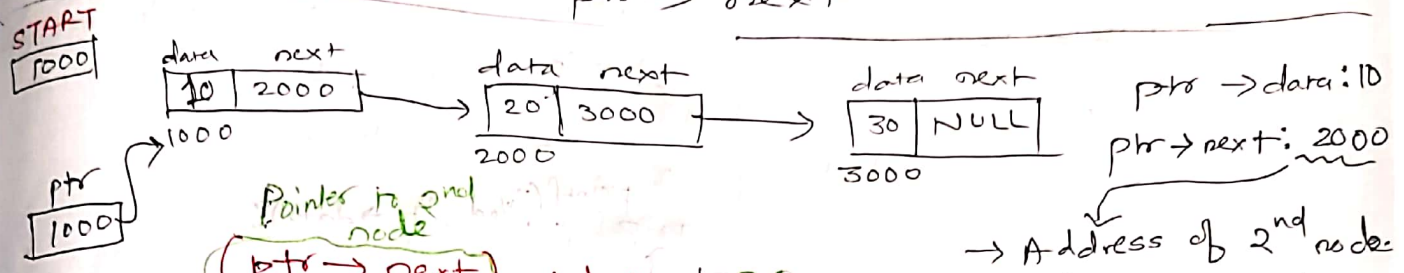
```
struct Node *ptr;
ptr = malloc (sizeof (struct Node));
```

Ptr : Pointer to structure variable.

Access members from it ? How

(*ptr). data OR ptr → data

—||— . next ptr → next



ptr → data : 10

ptr → next : 2000

→ Address of 2nd node.

→ pointer to node.

→ ptr —||— 2nd —||—

→ —||— struct variable

Pointer to 2nd node

(ptr → next) → data : 20

(ptr → next) → next

Next field of 2nd node / 3rd row address

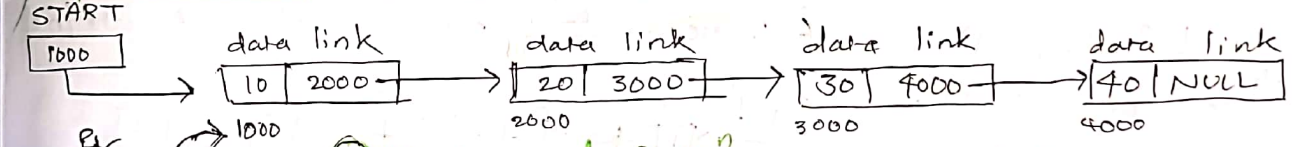
How to access next node? ptr = ptr → next

struct Node *START → Global var.

START Addr. of 1st Node

Empty L.L ⇒ No first Node ⇒ START cannot have any valid address, i.e. START = NULL

Operations ① Traversal



① Traversal Oper^n

```
struct Node {
    int data;
    struct Node *link;
} *START = NULL;
Void main() {
    Traversal();
}
Void Traversal() {
    struct Node *ptr;
    ptr = START;
    while (Ptr != NULL)
    {
        pf ("%d", ptr → data);
        ptr = ptr → link;
    }
}
```

START is Global

```
struct Node {
    int data;
    struct Node *link; }
Void main() {
    struct Node *START = NULL;
    {≡
    Traversal (START);
    }
Void Traversal (struct Node *Ptr) {
    While (Ptr != NULL)
    {
        pf ("%d", Ptr → data);
        Ptr = ptr → link;
    }}
```

START is local

That is why, START is passed as an argument to Traversal

While (Ptr) = While (Ptr != NULL)

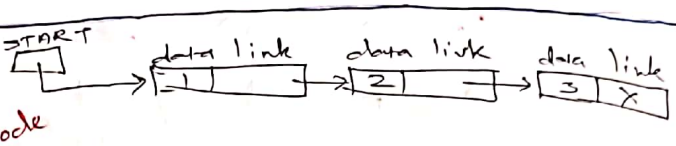
② Count the # of nodes in LL.

```
int count_node(){
    int count = 0;
    struct Node *ptr;
    ptr = START;
    while(Ptr != NULL)
    {
        count++;
        Ptr = Ptr -> link;
    }
    return count;
}
```

③ Given a LL, print the data of last node

```
void printlast()
{   struct Node *Ptr;
    ptr = START;
    if(START == NULL)   // No Node        To ensure List is not empty
        return;         // No last Node
    while(Ptr -> link != NULL)
    {
        Ptr = ptr -> link;
    }
    printf("%d", Ptr -> data);
}
```

④ Print 2nd last node data in LL.



0 Node 1 Node

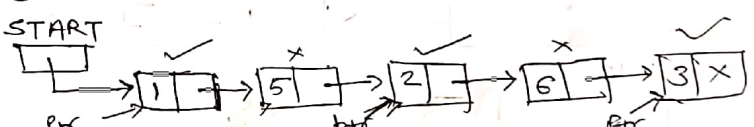
```
if(START == NULL || START -> link == NULL)
    return;
ptr = START;
while(Ptr -> link -> link != NULL)
    ptr = ptr -> link;
printf("%d", ptr -> data);
```

Ensure that atleast 2 nodes exist

⑤ Given a LL & a key, find whether the key is present in LL or not.

```
void search(int key)
{   struct Node *ptr = START;
    while(Ptr != NULL)
    {
        if(ptr -> data == key)
        {
            printf("Yes");
            return;
        }
        ptr = ptr -> link;
    }
    printf("NO")
}
```

⑥ Write a code to print alternate node data in LL



NULL -> link is error

should be valid address ptr = (ptr -> link) -> link

```
struct Node *Ptr = START;
while(Ptr != NULL && ptr -> link != NULL)
{
    printf("%d", Ptr -> data);
    ptr = ptr -> link -> link;
}
if(Ptr == NULL)
    return;
if(Ptr -> link == NULL)
    pf("%d", Ptr -> data);
```

Additional code works for 1 node. For add nodes to work

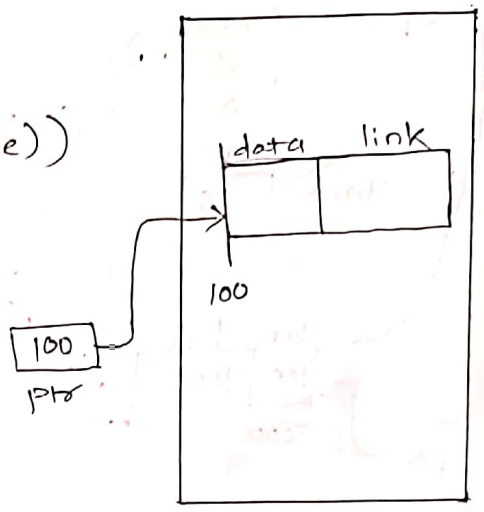
Insertion

① Memory allocate:

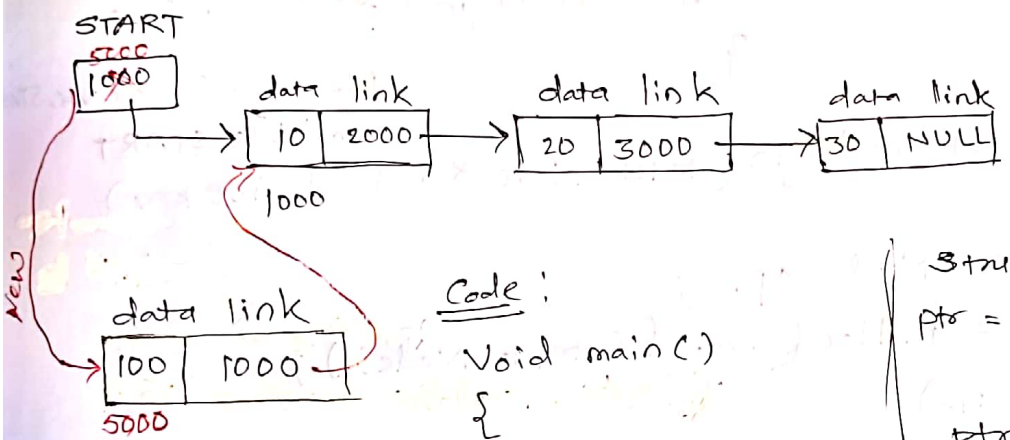
```
struct Node *ptr
ptr = malloc(sizeof(struct Node))
```

② Insert data/key in this new node.

```
ptr -> data = 100;
ptr -> data = key;
```



③ Where to place/insert this node.



Insert 100 a> beginning

Code:

```
Void main()
{
   ==========
   Insert_at_beg(key);
   ==========
}
```

```
Struct Node *ptr;
ptr = malloc(sizeof
        (struct Node));
ptr->data = key;
ptr->link = START;
START = ptr;
```

```
Void Insert_at_beg(int key)
{
  struct Node *ptr;
  ptr = malloc(sizeof(struct Node));
  if (Ptr != NULL)          // memory is available
  {
    ptr -> data = key;
    ptr -> link = START;
    START = Ptr;
  }
```

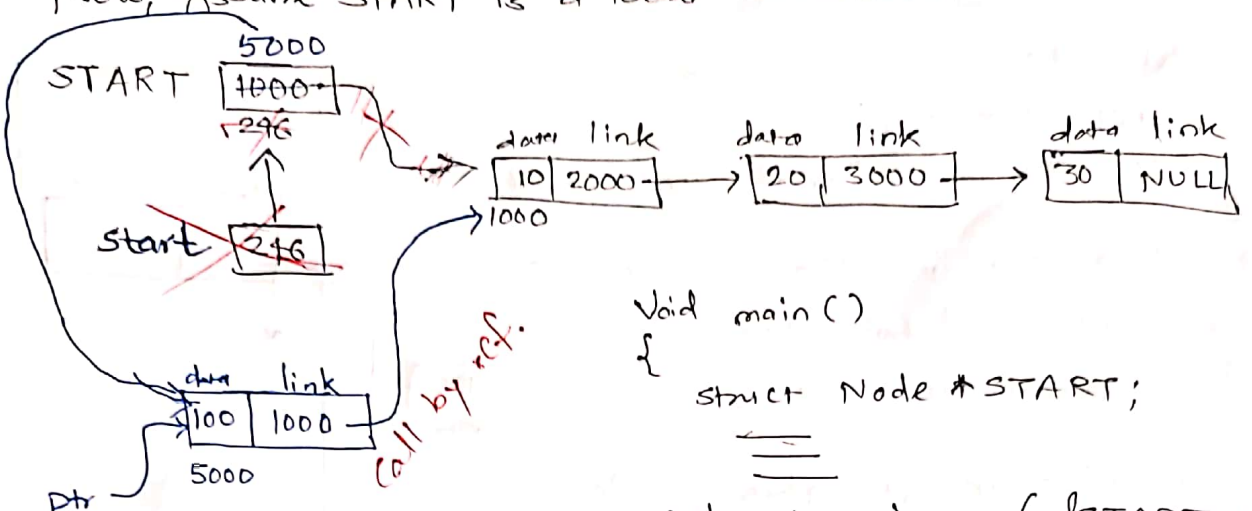
When START is Global

Another way :-

```
Void main()
{
  struct ----
  ========
  START = Insert()
```

```
struct Node * Insert_at_begin(struct Node *start, int key)
{ struct Node *Ptr;
  ptr = malloc( ---- );
  ========
  return (Ptr);
}
```

Now, Assume START is a local variable of main.




```
Void main()
{
   struct Node *START;
   ==
   ==
   Insert_at_begin(&START, 100);
}
```

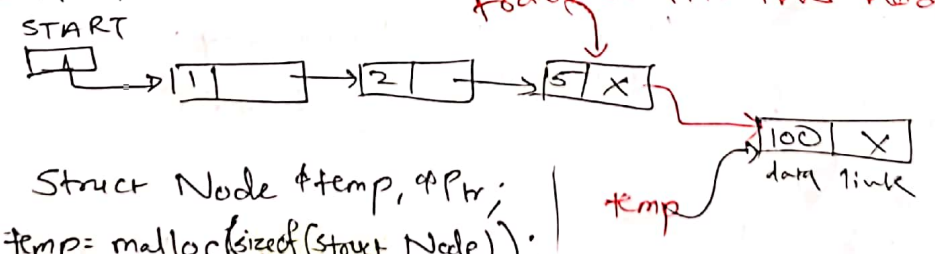
Addr. of a Pointer var. START

```
Void Insert_at_begin(struct Node **start, int key)
{
   struct Node *ptr;
   ptr = malloc(sizeof(struct Node));
   if(Ptr)
   {
      ptr->data = key;
      ptr->link = *start;   (*246 = 1000)
      *start = ptr;         (Value at start "1000" = 500)
   }
}
```

(Red) crosses Vanishes after this code)

*start
Addr. of first node. pointing to it

b) Insert 100 at last

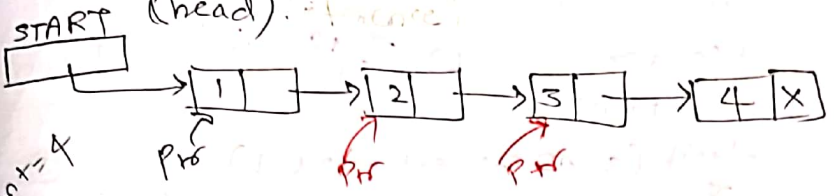



```
Struct Node *temp, *Ptr;
temp = malloc(sizeof(Struct Node));
if(temp)
{ temp->data = key;
  temp->link = NULL;
  if(START == NULL)
  {
     START = temp;
     return
  }
  ptr = START
```

At least 1 node present

```
While(Ptr->link != NULL)
{
   Ptr = Ptr->link;
}
ptr->link = temp;
```

Finding middle element in a L.L. (If there are two middle nodes, in case, when N is even, print the second middle element.)
Write a function getMiddle() which takes a head reference as the only argument and should return the data at the middle node of the linked list.

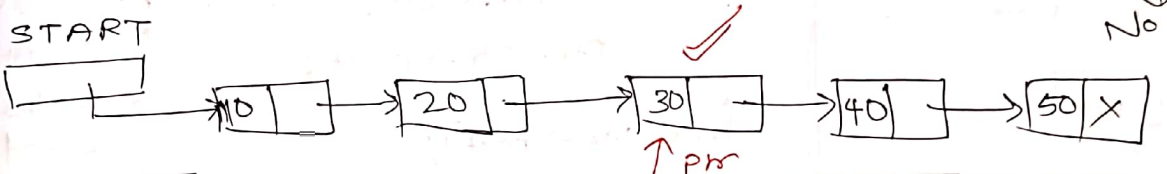


count = count / 2;

count = 2 → 2 times ptr = ptr → link

Works for Even no. of nodes

if count = 0 → No middle node.

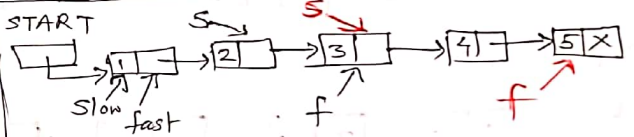


count = 5
count = count/2
= 2 times ptr = ptr → link

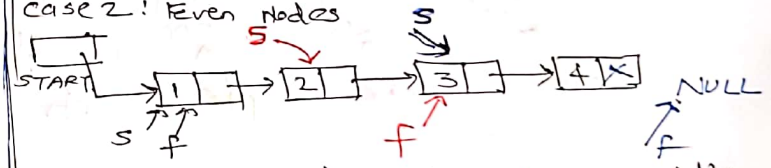
```
int getMiddle(Node *START)
{
    int count = 0;
    struct Node* ptr;
    ptr = START;
    while (Ptr != NULL)
    {
        count ++;
        ptr = ptr -> link;
    }
    count = count / 2
    int i;
    for(i = 0, i < count, i++)
    {
        ptr = ptr -> link;
    }
    return ptr -> data;
}
```

Another Approach

Case 1: If the nodes are odd



Case 2: Even Nodes



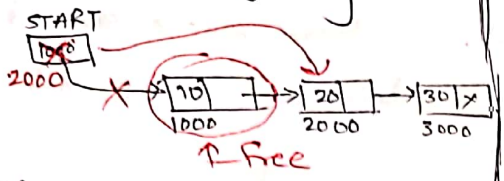
In both cases, slow ptr points middle node.

```
int getMiddle(Node *head)
{
    struct Node* slow, *fast;
    slow = head;
    fast = head;
    if (head == NULL)        // If empty LL, returns -1.
        return -1;
    while (fast != NULL && fast->next != NULL)
    {
        slow = slow -> next;
        fast = fast -> next -> next;
    }
    return slow->data;
}
```

(Here link = next, START = head)

Deletion

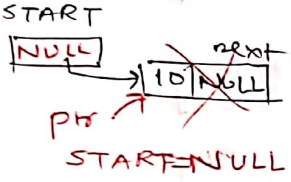
a) Deletion at beginning



After deletion of 1st node, START ⇒ must point 2nd ".

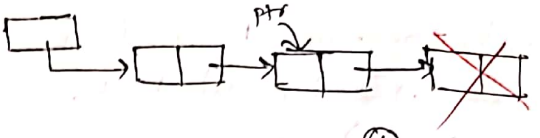
```
if(START == NULL)
    return;
ptr = START;
START = START->next;
free(ptr)
```

case 1: If LL has one node.



b) Deletion from End

Traverse till 2nd last node & Insert NULL to its next field.



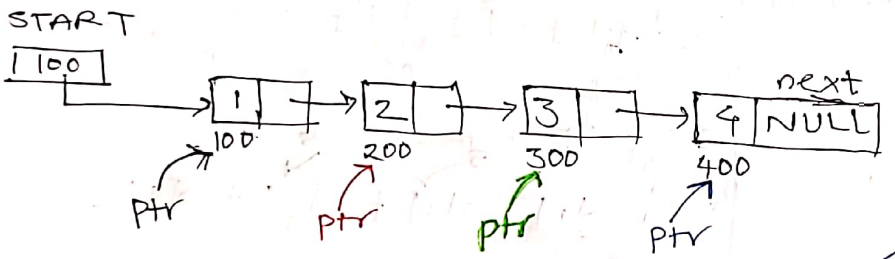

```
Ptr = START;
if (START == NULL)
    return;
```
case ① If L.L is empty

```
elseif (START->next == NULL)
{
    START = NULL;
    free (ptr);
    return;
}
```
case ② If L.L has Only one Node

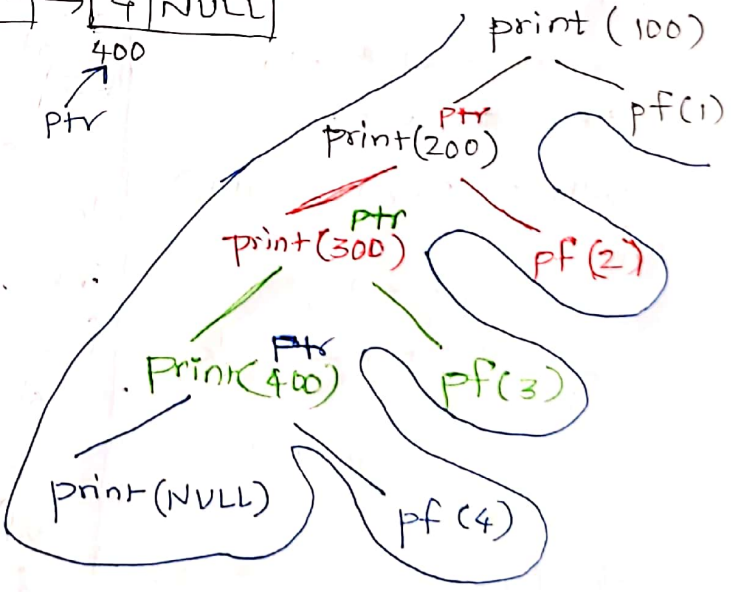
```
While (Ptr->next->next != NULL)
{
    Ptr = Ptr->next;
}
free(ptr->next);
ptr->next = NULL;
```
case ① & ② fails, Atleast 2 nodes are ensured.

Reversal

a) print values/data of L.L. in Reverse Order.



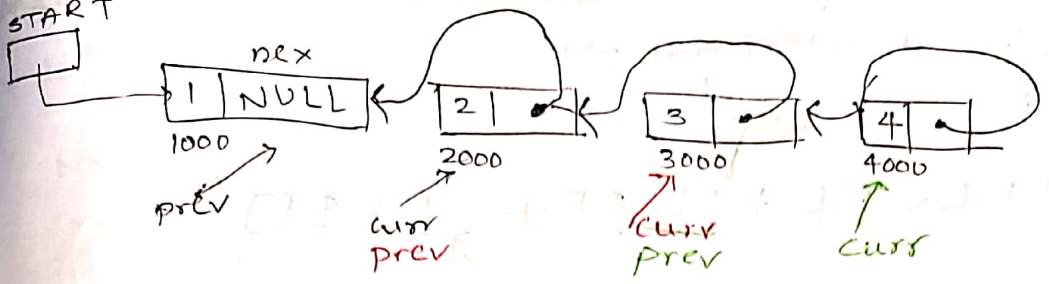

```
Void Print(Struct Node *ptr)
{
  if (ptr)
  {
    print (Ptr->next);
    pf ("%d", ptr->data);
  }
}
```



Output 4 3 2 1

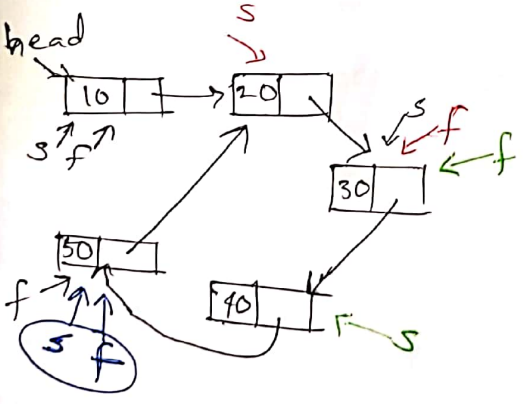
b) Complete fun" reverseList() with head ref. as the only argument should return new head after reversing the list.

Actual Reversal of a L.L




```
Struct Node* reverseList (struct Node *head)
{
    Struct Node *curr, *prev, *nex;
    if (head == NULL || head -> next == NULL)
        return head;        // 0 node or 1 node => do nothing
    prev = NULL;
    curr = head;
    while (curr != NULL)
    {
        nex = curr->next; //save next node
        curr-> next = prev; //reverse concept
        prev = curr;
        curr = nex;
    }
    return prev;
};
```

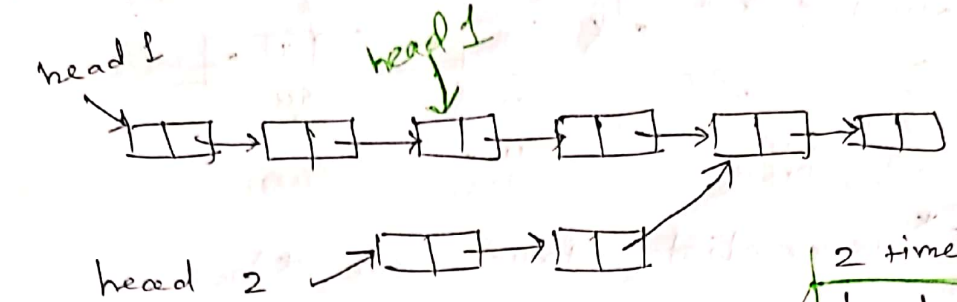
Q. Find whether there is a loop in the L.L or not.




```
bool detectLoop (Node* head)
{
    struct Node *slow, *fast;
    slow = fast = head;
    if (head == NULL)
        return false;
    While (fast != NULL &&
           fast -> next != NULL)
    {
        slow = slow->next;
        fast = fast->next->next;
        if (slow == fast)
            return true;
    }
    return false;
}
```

Q. Find the intersection point (if exists) bet^n 2 Linked lists & return the value at intersection pt.

Assume LL-1



$l_1 = 6$

$l_2 = 4$

2 times (diff time)
head 1 = head 1 -> next

diff = abs($l_1 - l_2$)
= 2

2 times LL_1 has to traverse in order to meet LL_2 at intersection point.

```
int intersectPoint (Node* head1, Node* head2)
{
    int diff; c1=0, c2=0, i;
    Node *ptr1=head1, *ptr2=head2;
    while (ptr1)
    {
        c1++;
        ptr1 = ptr1 -> next;
    }                                  } Find length of 1st LL
    while (ptr2)
    {
        c2++;                          // find length of 2nd LL
        ptr2 = ptr2 -> next;
    }
    diff = abs (c1 - c2);
    ptr1 = head1;
    ptr2 = head2;
    if (c1 > c2)  // 1st ll is bigger
    {
        for(i=1; i<=diff; i++)
            ptr1 = ptr1 -> next;
    }
    elseif (c2 > c1)
        for (i=1; i<= diff; i++)
            ptr2 = ptr2 -> next;
    }
    while (ptr1 != ptr2)
    {
        ptr1 = ptr1 -> next;
        ptr2 = ptr2 -> next
    }
```

```
if (ptr1 != NULL)
    return ptr1 -> data;
return -1;
```

5. Given two sorted Linked Lists, merge them & return its head.

```
Node* sortedMerge (Node* a, Node* b)
{
  Node *head = NULL;
  Node * last = NULL;
  if (a == NULL)
     return b; //If 1st LL empty.
  if (b == NULL)
     return a; // If 2nd LL empty
  if (a->data <= b->data)
  {
    head = last = a;
     a = a-> next;
   }
   else
   {
     head = last = b;
     b = b->next;
   }
   While (a != NULL && b != NULL)
   {
      if (a->data <= b->data)
      {
        last->next = a;
         last = a;
         a = a->next;
       }
       else
       {
         last->next = b;
         last = b;
         b = b->next;
        }
    }
    if (a == NULL)
       last -> next = b;
    else
       last-> next = a;
    return head;
}
```

The following C function takes a S.L.L. P as an argument.

```
int f(struct item *p)
{
  return ((p == NULL) || (p->next == NULL)) ||
         ((p->data <= p->next->data) && f(p->next));
}
```

(0) (1)

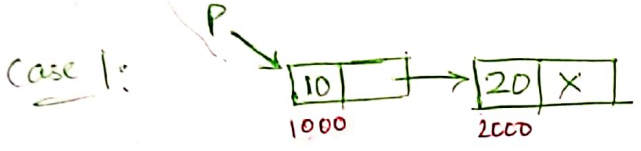
The func. returns 1 if & only if

A] The list is empty or exactly 1 ele.

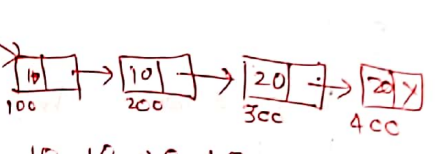
B] The ele. in list are sorted in non-decreasing order.

C] The " " " " " " " -increasing " .

D] Not all the elements in the list have same data.



case 1:



case 2

10, 10, 20, 20

Non-decreasing order

Not Ascending ✗

return 0||0||10<=10

① && f(200)

0||0||1 && 1 = returns 1

∴ 0||0||1 && (0||0||1 && f(300))

1 && (0||1||1)

2) In worst case, the no. of comparisons needed to search a S.L.L. of length n for a given element is :-

A] log n

B] n/2

C] log n − 1

D] n

In worst case, ele. will be in the last node where we have to traverse a L.L. n times.

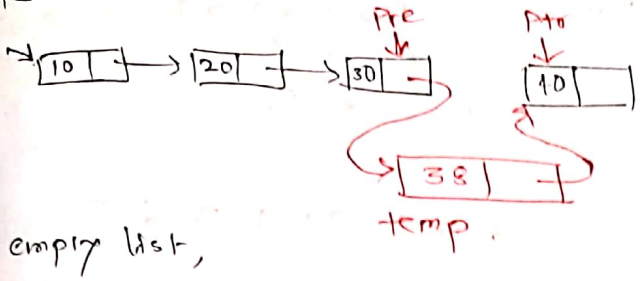
3) What is the worst case time complexity to reverse a S.L.L. in O(1) space ? O(n)

O(1) → constant memory → 1, 2, 3 → limited no. of variables

No. of comparisons is equal & dependent on no. of nodes.

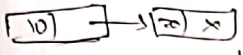
What is the worst case T.C. of inserting n elements into an empty L.L, if the L.L. needs to be maintained in sorted order?

Let's assume we have a sorted L.L & 38 needs to be inserted.



In the wrost, case the 38 could be inserted in the last node so, $O(n)$

In empty list,



If it is 1st ele $\Rightarrow$ 0 comparisons needed.

2nd " $\Rightarrow$ 1 — " —

— " — 3rd " $\Rightarrow$ 2 — " —

nth " $\Rightarrow$ (n-1) — " —

$\therefore O(n^2)$

$\therefore 1 + 2 + 3 + \dots (n-1) = \frac{(n-1)(n)}{2} = \frac{n^2}{2} - \frac{n}{2}$

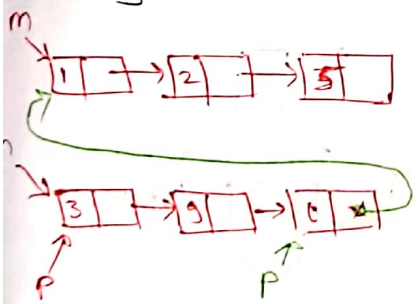
We need to traverse the list for each ele we want to insert

$n \times n = n^2$

to find correct position in sorted list.

---

4)
```
void join (node *m, node *n)
{
    node *p = n;
    while(p->next != NULL)
    {
        p = p->next;
    }
    p->next = m;
}
```



Assuming that m & n points to valid NULL terminated L.L. Invocation of join will-

A] Append list m to the end of list n for all i/ps.

B] Either cause a null pointer dereference or append list m to the end of list n.

C] Cause a null pointer dereference.

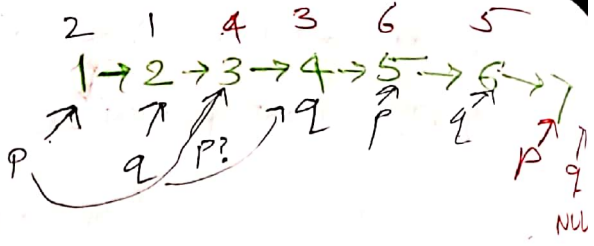
d] Append list m to the end of list n.

---

5) The following C function takes a S.L.L. of integer as a parameter & rearrange the elements of list. The function is called with the list containing the integer. What will be the contents of the list after the function completes it's execution?

```
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;
        p->value = q->value;        } swap
        q->value = temp;
        p=q->next;
        q=(p ? p->next : 0);
    }
}
```

True when non-zero



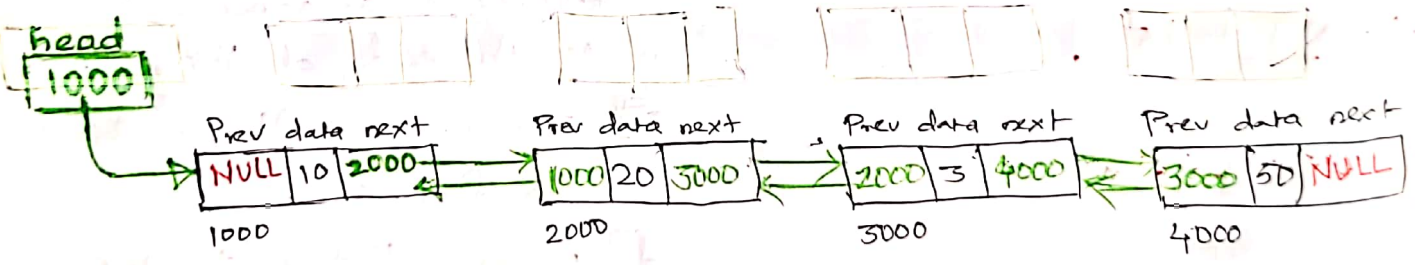
output 2 1 4 3 6 5

Types of Linked List.

1) Singly Linked List ✓
2) Doubly " "
3) Circular " "
4) Header " "

2) Doubly Linked List:

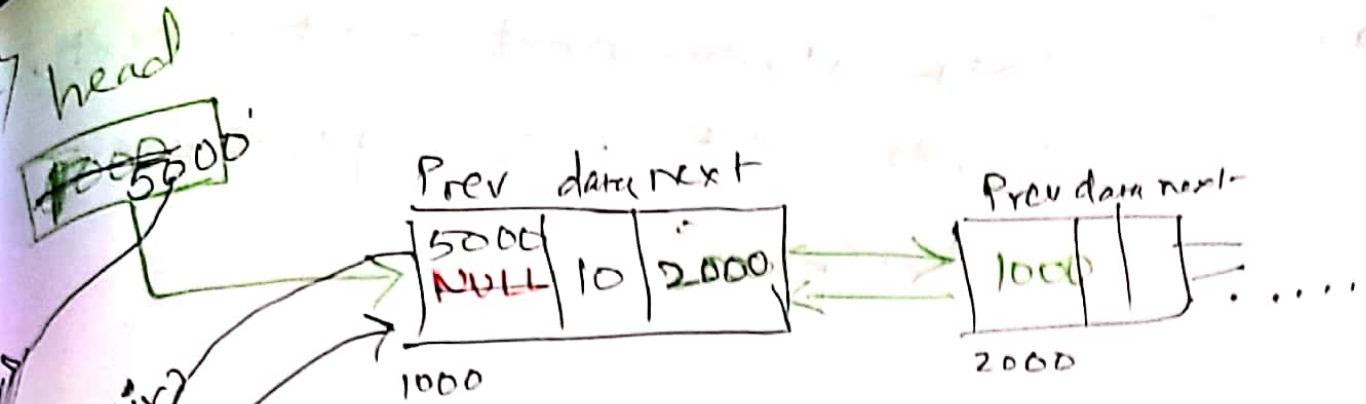
In each node, we have two pointers, pointer to previous as well next node.




```
struct Node
{
    struct Node *Prev;
    int data;
    struct Node *next;
}
```

Traversal → same as SLL

a) Insertion at beginning




```
struct Node *temp = malloc (sizeof (struct Node));
```

i) temp → data = key;

ii) temp → next = head;

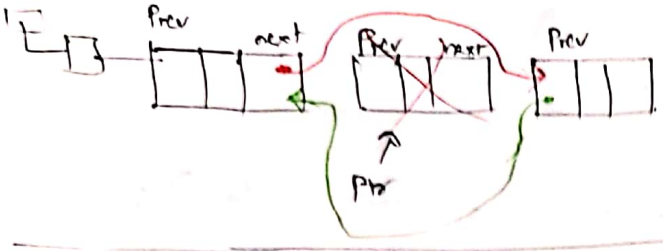
iii) temp → prev = NULL;

iv) head → prev = temp

v) head = temp

```
if (head == NULL)
{ temp → next = NULL;
  temp → prev = NULL;
  head = temp;
}
else
{ From point ii)
}
```

Q. Given a pointer to a node, delete that node.

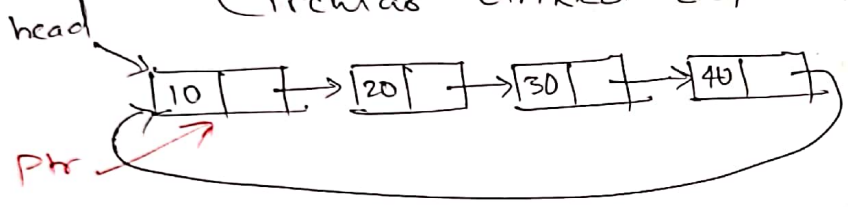



```
i) ptr->prev->next = ptr->next;
ii) ptr->next->prev = ptr->prev;
    free(ptr);
```

Q. Given a DLL, search for an element, return 1 if present.

```
int SearchNode(struct node *head, int key)
{
  struct node *temp = head;
  while(temp != NULL)
  {
    if(temp->data == key)
      return 1;
    temp = temp->next;
  }
  return -1;
}
```

Circular Linked List



Traversal code:

```
if(head == NULL)
    return;
do{
pf("%d", ptr->data);
ptr = ptr->next;
}while(ptr != head)
```

works for single node too.

We use do while as it runs for at least one time.
If we had used while (ptr != head), it wouldn't have run as ptr is head.

Q. Consider the following ANSI C program : GATE 2021 2 Marks

```
#include <stdio.h>
#include <stdlib.h>
struct Node {
    int value;
    struct Node *next; };
int main () {
    struct Node *boxE, *head, *boxN; int index = 0;
    boxE = head = (struct Node *) malloc (sizeof (struct Node));
    head -> value = index;
    for (index = 1; index <= 3; index++) {
        boxN = (struct Node*) malloc (size of (struct Node));
        boxE -> next = boxN;
        boxN -> value = index;
        boxE = boxN; }
    for (index = 0; index <= 3; index ++) {
        printf ("Value at index %d is %d \n", index, head -> value);
        head = head -> next;
        printf ("Value at index %d is %d \n", index+1, head -> value); } }
```

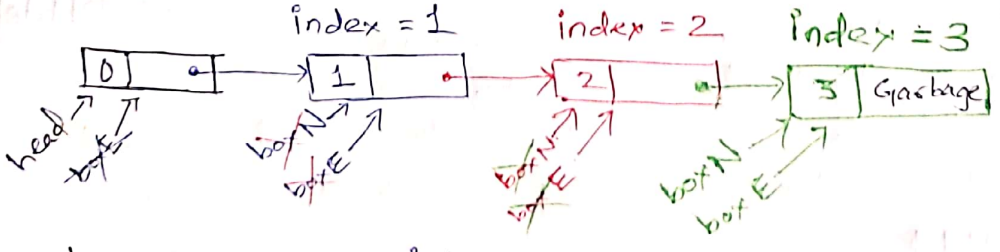
Which one of the statements below is correct about the program?

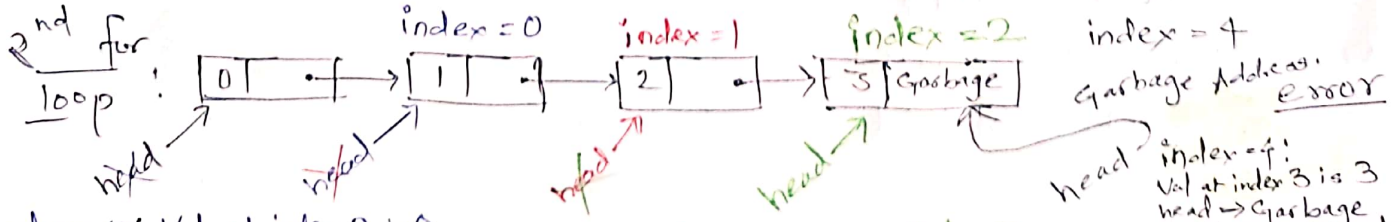
A) Upon execution, the program creates a linked-list of five nodes.

B) Upon execution, the program goes into an infinite loop.

C) It has a missing return which will be reported as an error by the compiler.

D) It dereferences an uninitialized pointer that may result in a run-time error.

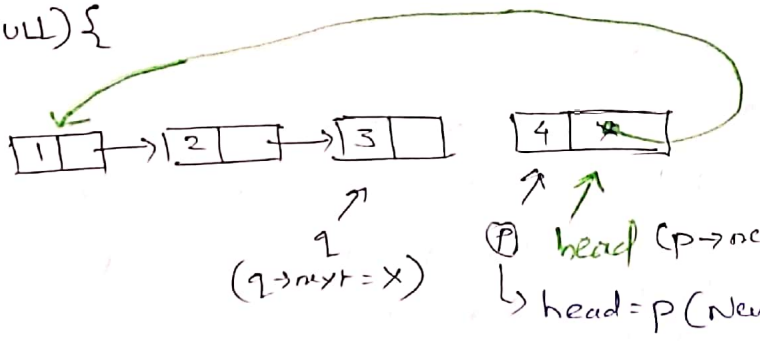




Q. The foll. C func. takes a S.L.L. as i/p argument. It modified the list by moving the last ele. to the front of the list & returns the modified list. Some part of the code is left blank.

GATE-2010 2 Marks

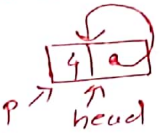
```
typedef struct node {
    int value;
    struct node * next;
} Node;

Node * move_to_front(Node * head) {
    Node *p, *q;
    if ((head == NULL) || (head->next == NULL))
        return head;
    q = NULL;
    p = head;
    while (p->next != NULL) {
        q = p;
        p = p->next;
    }
    ------------------
    return head;
}
```



Choose the correct alternative to replace the blank line.

A) q = NULL; p->next = head; head = p; ✗ It becomes a circular L.L.

B) q->next = NULL; head = p; p->next = head; ✗



(No way, we can access head if head = p)

C) head = p; p->next = q; q->next = NULL

✓ D) q->next = NULL; p->next = head; head = p;

Q. Let SLLdel be a function that deletes a node in a SLL given a pointer to the node & a pointer to the head of the list. Similarly let DLLdel be another function that deletes a node in a DLL given a pointer to the node & a pointer to the head of the list.

Let n denote the number of nodes in each of the linked lists.

Which one of the following choices is TRUE about the worst-case T.C. of SLLdel and DLLdel?

A) SLLdel is O(1) & DLLdel is O(n)

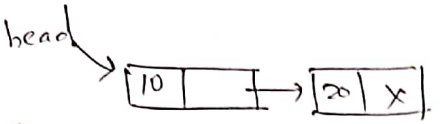
B) Both SLLdel & DLLdel are O(log(n))

C) ——— " ——— O(1)

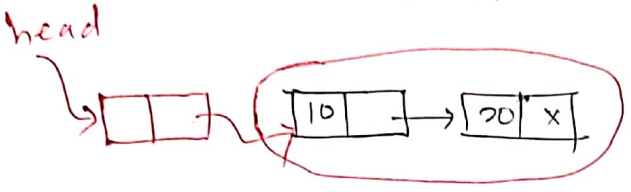
✓ D) SLLdel is O(n) & DLLdel is O(1)

Header Linked list

A Header node is a special node that is found at the beginning of the list. A list that contains this type of node, is called the header-linked list.



Normal/simple L.L.



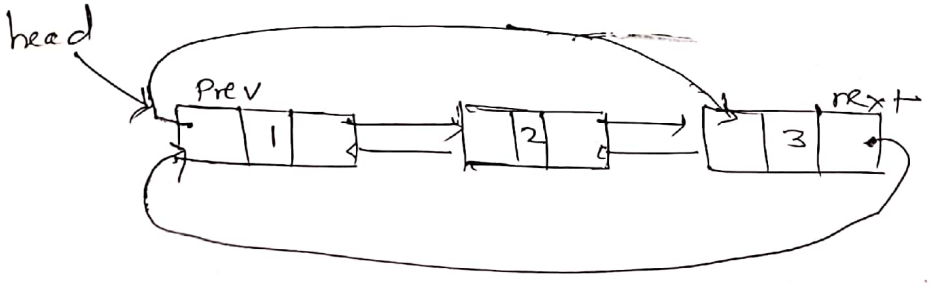
head points to the header node instead of the 1st node of the list. Header node does not represent an item in the LL. Actual LL starts from 2nd node.

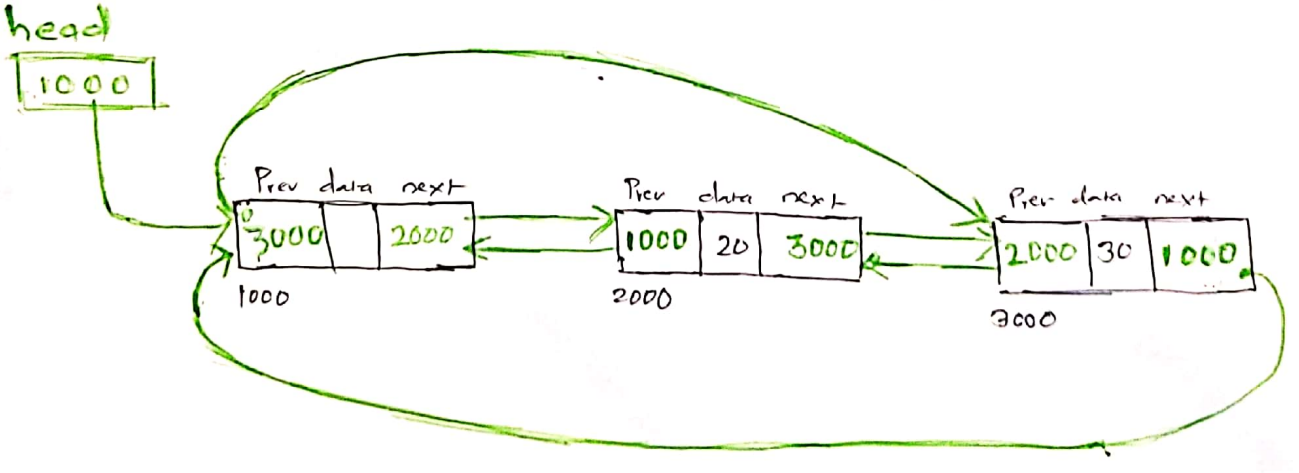
This header node can be used store information other than that found in every other node like no. of nodes in a list.

Doubly Circular Linked List

Doubly Circular Linked List is a circular linked list (i.e. node of the list contains the address of the first node of the list) in which a node contain pointers to its previous node as well as the next node.

Unlike simple Doubly Linked List, DCLL does not contain NULL in any of the node.





Stacks

* Linear D.S.
* LIFO / FILO
* Restricted D.S.
* Order of deletion: reverse order of insertion
* Both insertion and deletion are performed only at one end called as TOP of stack.
TOP: element added most recently.

Stack as ADT

Insert → Push
Delete → Pop
IsEmpty() → True if empty
→ false if not

Sole purpose of stack is to with or to delay to postpone decisions.

Applications:
1) Recursion / Function
2) TOH
3) Infix to prefix / postfix
4) Pre/Post fix Evaluation
5) Balanced parenthesis check.

Implementation using Array

Inserting an element:

with Array/stack Declaration
Global:- STACK[SIZE];
- TOP = -1;

O(1) constant time

```
Void Push(int x)
{
  if (TOP == SIZE-1)
    return;      // If stack is full.
  TOP++;
  STACK[TOP] = x;
}
```

Deletion:

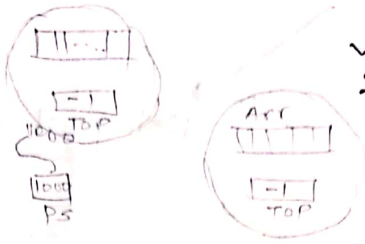
STACK[SIZE];
TOP = -1;

constant time
O(1)

```
int pop()
{
  int temp;
  if(TOP == -1)
    return INT_MIN; // If stack is empty
  temp = STACK[TOP]  // Element to be deleted
  TOP--
  return temp;       // we return the element to be deleted.
}
```

Stack: Collection of two things: An array & a variable TOP

∴ Collection of diff./Heterogenous Datatypes.




```
struct STACK
{
  int Arr[SIZE];
  int TOP;
}

Void main()
{
  struct STACK S1, S2;
   S1.TOP = S2.TOP = -1;
  PUSH (&S1, 10)
```

```
void Push(struct STACK* PS, int x)
{
  if (PS->TOP == SIZE-1)
      return;
  PS->TOP = PS->TOP + 1;
  PS->Arr[PS->TOP] = x;
}
```

Stack Permutation

Order of insertion of given elements is fixed. What could be possible order of pop? (stack permutation).

Let n = 3 1, 2, 3.

Possible permutation = 3!

Let, Order of insertion be 1, 2, 3 →

Pop can be performed any time.

1, 3, 2

a) push(1)
b) pop() 1
c) push(2)
d) " (3)
e) pop() 3
f) pop() 2

Valid stack permutation.

2, 3, 1

Push(1)
—|—(2)
pop() 2
push(3)
pop() 3
pop() 1

Valid

3, 1, 2

Push(1)
—|—(2)
—|—(3)
pop() 3

can we pop 1 before 2?
NO.

Not valid

Here, out of 3! or 6 permutations, 5 are valid.

n! Permutation	⟹	$\frac{2nC_n}{n+1}$	stack permutations

{Priority / Associativity}

Infix to Postfix

Without using stack

Ex. 1:

infix: 2 + 3 × 5

× + ↓

2 + [35×]

op1 op2

postfix: 2 3 5 × +

Ex. 2: 2 + 3 × 4 / 6 ↑ 2

Associativity (×/) from L-R

2 + 3 × 4 / [6 2 ↑]

2 + [3 4 ×] / [6 2 ↑]

op1 op2

2 + [3 4 × 6 2 ↑ /]

op1 op2

postfix:- 2 3 4 × 6 2 ↑ / +

Ex. 3

infix: (a+b) × c/d − e↑f↑g/h

[ab+] × c/d − e↑f↑g/h ①

[ab+] × c/d − e↑[fg↑]/h

[ab+] × c/d − [efg↑↑]/h

[ab+cx]/d − [efg↑↑]/h

[ab+cxd/] − [efg↑↑]/h

[ab+cxd/] − [efg↑↑h/]

Postfix:- ab+cxd/efg↑↑h/−

Parenthesis be solved on my Hime.

$2^{2^3} \Rightarrow 2^8 \Rightarrow 4^3$

↑ R to L

×, / L to R

+ − L to R

Using stack

infix: 2 + 3 × 5

Prefix: 2 + [× 3 5]

op1 op1

Prefix: + 2 × 3 5

Postfix: 2 3 5 × +

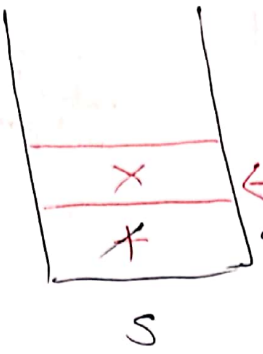
(Here, order of operands does not change)

Infix to postfix using stack.

infix: 2+3×4 ← End

postfix: 2 3 4 × +

Scan infix from L to R.
Add operands to postfix.



+ is encountered
⇒ stack is empty
Push it onto stack

② If operator is encountered, push it onto stack if no higher priority operator is waiting.

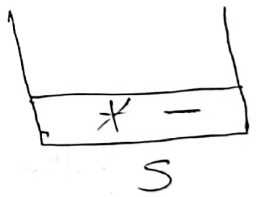
TOP operator scanned
+ < ×
× is encountered, push it onto stack, '+' won't be evaluated before × as it has lower priority

③ If higher priority operator is waiting, evaluate it & push current operator onto stack.

④ If Equal Priority operator is waiting, evaluate it & push current operator onto stack.

⑤ If string ends, pop all operators one by one.

Ex. Infix: a+b−c ← string ends
(pop '+' first)

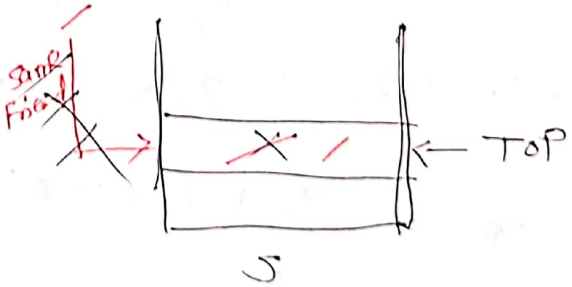


Postfix: ab+c−

×, / } L to R
+

Ex. Infix: a+b×c/d

Postfix abcxd/+

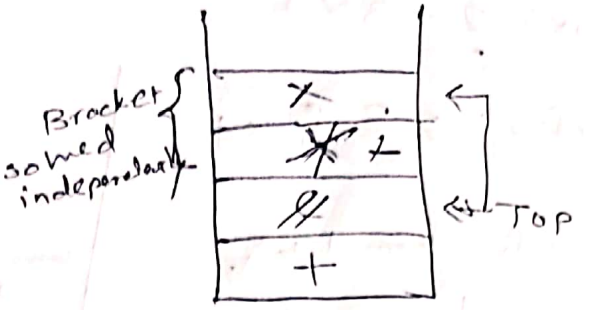


Verification:- a+b×c/d
a+[bcx]/d
a+[bcxd/]
Postfix: abcxd/+

Ex: Infix : 2 + (3 × 4 − 6 / 2) ← End

(⇒ Push

postfix: 2 3 4 × 6 2 / − +



When "(" is encountered; push onto stack. Push other operators as per rules until ")" is encountered

When ")" is encountered, pop all operators till "(" & discard "(".

"(" : start of an expression.

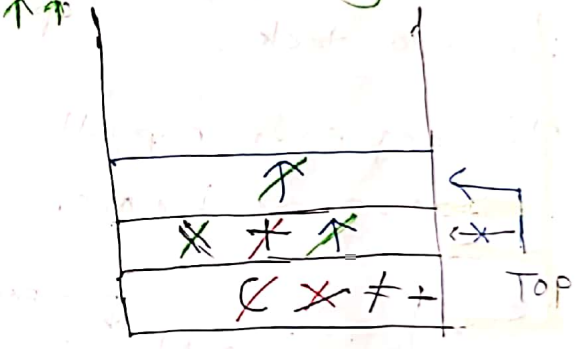
")" : end of an expression.

Ex. infix: (a+b)×c/d − e ↑ f ↑ g / h ← End of string

Higher priority than left ↑ as its R to L associative

postfix: a b + c × d / e f g ↑ ↑ h / −

∴ a b + c × d / e f g ↑ ↑ h / −



Infix to prefix

Without using stack.

Using Stack

Ex 1. Infix : 2 + 3 × 5

op1 2 + [×35] op2

+ 2 × 3 5

Ex. 2 infix : $(a+b) \times c/d - e \uparrow f \uparrow g / h$

$[+ab] \times c/d - e \uparrow f \uparrow g / h$

$[+ab] \times c/d - e \uparrow [\uparrow fg] / h$

op1 op2

$[+ab] \times c/d - [\uparrow e \uparrow fg] / h$

$[\times +abc]/d - [\uparrow e \uparrow fg]/h$

$[/ \times +abcd] - [\uparrow e \uparrow fg]/h$

$[/ \times +abcd] - [/ \uparrow e \uparrow fgh]$

$- / \times +abcd / \uparrow e \uparrow fgh$

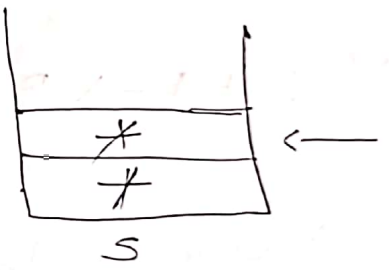
Using stack

Infix : $a+b-c$

reverse infix : $c-b+a$

o/p : $cba+-$

reverse o/p : $-+abc$



Though '+' arrives after '-' in reverse infix. originally, it comes before '-'. So, '+' has higher priority.

verification : $a+b-c$

$[+ab]-c$

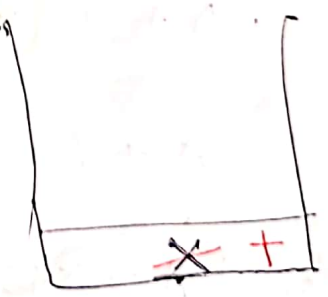
$-+abc$

Infix : $2+3 \times 4$

reverse infix : $4 \times 3 + 2$ ← Rand

o/p : $43 \times 2+$

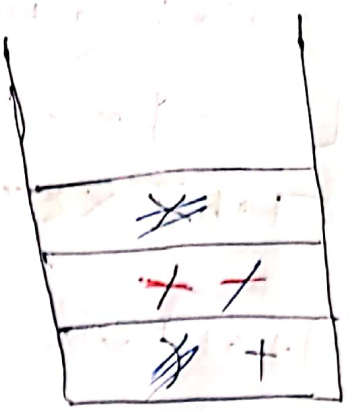
Reverse o/p : $+2 \times 34$



Ex-4 :

Infix: a+(bxc-d/e)

reverse
infix :)e/d-cxbC+a(Ends.

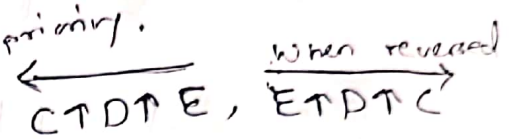


o/p : ed/cbx-a+

reverse o/p : +a-xbc/de

Ex-5

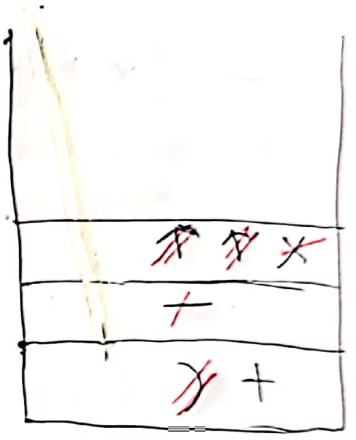
Arrows indicate priority.



Infix: A+(BxC↑D↑E-F)

Reverse
Infix :)F-E↑D↑CxB(+A← Ends

↑ Higher priority ↑ Lower priority



o/p : FED↑C↑BX-A+

reverse o/p : +A-XB↑C↑DEF

Verification Infix: A+(BxC↑D↑E-F)

A + (BxC↑[↑DE]-F)

op1 op2

A + (Bx[↑C↑DE]-F)

op1 op2

A + ([XB↑C↑DE]-F)

op1 op2

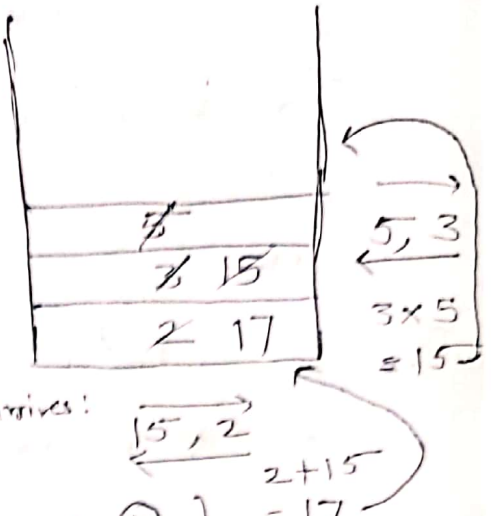
A + (-XB↑C↑DEF)

∴ +A-XB↑C↑DEF

Postfix Evaluation

infix: 2 + 3 x 5

postfix: 2 3 5 x +



Using stack

→ operators are already arranged. Push operands onto stack.

→ When operator is encountered (operator ⇒ ⊛) :

a) Pop 1st ele ⇒ A

b) " 2nd " ⇒ B

B ⊛ A

→ Push result onto stack

→ Repeat this process until all operations are evaluated. Final answer is the last ele on stack.

Ex. 2 infix: 2 + 3 x 4 - 6 / 2

postfix: 2 + [3 4 x] - 6 / 2

2 + [3 4 x] - [6 2 /]

[2 3 4 x +] - [6 2 /]

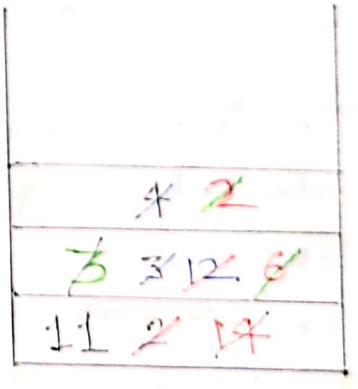
postfix: - 2 3 4 x + 6 2 / -

i) x
pop order
4, 3
push order
3 x 4
PUSH 12

ii) +
12, 2
2 + 12 = 14
PUSH 14

iii) /
2, 6
6 / 2
3
PUSH 3

iv) -
3, 14
14 - 3
11
PUSH 11



2 3 4 x + 6 2 / −

scan from L to R

2 12 + 6 2 / −

14 3 −

11

} Postfix Evaluation.

Q. The result of evaluating the postfix expression

10 5 + 60 6 / * 8 −

direct ✓ using stack X

GATE 2015

15 60 6 / x 8 −

15 10 x 8 −

150 8 − = 142

Q. Which of the foll. is essential to convert an infix expr. to post-fix exp. efficiently?

~~A~~ Operator stack

Q. Postfix expression of infix: a+b*c-d^e^f

a+b*c-d^[ef^]

a+b*c-[def^^]

a+[bc*]-[def^^]

[abc*+]-[def^^] = abc*+def^^-

Q. The foll. postfix exp. with single digit operand is evaluated using a stack: 823^/23*+51*−

Note that ^ is the exponentiation operator. The top two elements of the stack after the first * is evaluated are:

8 2 3 ^ / 2 3 x + 5 | x -

i) ^ → 3, 2 ← $2^3 = 8$

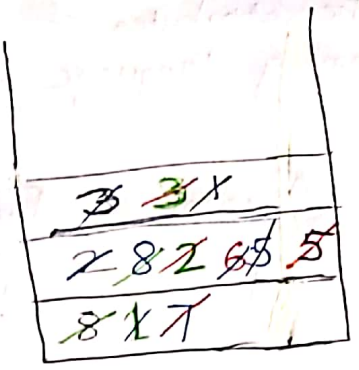
ii) / → 8, 8 ← 8/8 = 1

iii) x → 3, 2 ← 2×3 = 6

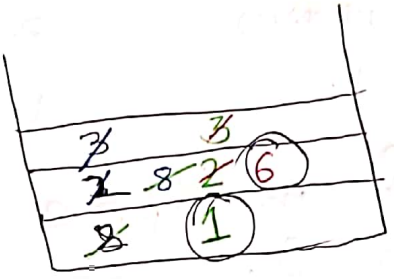
iv) + → 6, 1 ← 6+1 = 7

v) x → 1, 5 ← 5×1 = 5

vi) - → 5, 7 ← 7-5 = 2



First x



} After evaluating first x,

After evaluating first x, top two elements of the stack are 6, 1

Q. Consider the foll. C program:

```
#include <stdio.h>
#define EOF -1
void push(int); /* push the argument on the stack */
int pop(void); /* pop the top of the stack */
void flagError();
int main()
{
    int c, m, n, r
    while ((c = getchar()) != EOF)
    {
        if (isdigit(c)) push(c);
        if ((c == '+') || (c == '*'))
        {
            m = pop();
            n = pop();
            r = (c == '+') ? n+m : n*m;
            push(r);
        }
        elseif (c != ' ')
            flagError();
    }
    printf("%c", pop());
}
```

What is the o/p of the program for the foll.?

5 2 x 3 3 2 + x +

5 2 x 3 3 2 + x +

10 3 3 2 + x +

10 3 5 x +

10 15 +

25 //

isdigit() is a function that returns true if the argument is a digit

Q. The attributes of three arithmetic operators in some programming language are given below.

Operator	Precedence	Associativity	Arity
+	High	Left	Binary
−	Medium	Right	"
*	Low	Left	"

The value of the exp. $2-5+1-7*3$ in this language is 9.

→ (Priority tells where to use parentheses first & not who to evaluate first)

$$\left[2-\left(\overleftarrow{(5-}+1)-7\right)\right]\times 3 \qquad \text{R to L}$$

$$(2-(6-7))\times 3$$

$$[2-(-1)]\times 3$$

$$3\times 3$$

Q. A function f defined on stacks of integers satisfies the foll. properties.

$f[\emptyset]=0$ and $f[\text{push}[S,i]] = \max[f[S],0]+i$ for all stacks S and integer i.

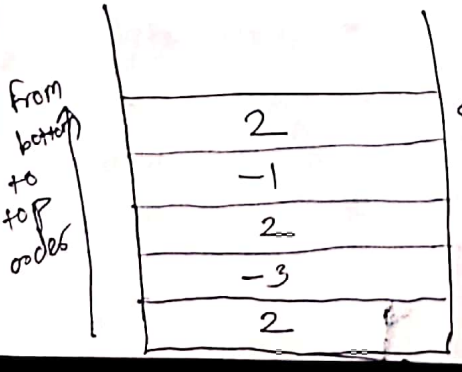
If a stack S contains the integers 2, −3, 2, −1, 2 in order from bottom to top, what is f[S]? (3)

→

f(stack is empty) = 0.

f(push(s,i)) = max(f(s),0)+i

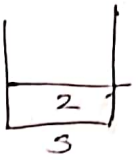
f(.. , i) = max(f(old),0)+i ← ele which is to be pushed.



In this stack, 2 is inserted first, then next element inserted is -3 and so on

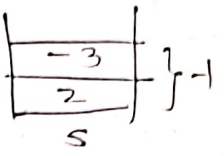
$\therefore f(push(s,2)) = \max(f(s), 0) + 2$ ← old

$= \max(0,0) + 2 = 2$



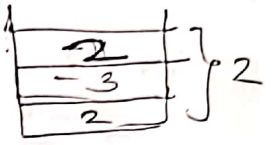
$f(Push(s,-3)) = \max(f(s), 0) + (-3)$ ← old value

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



$f(push(s,2)) = \max(f(s), 0) + 2$

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

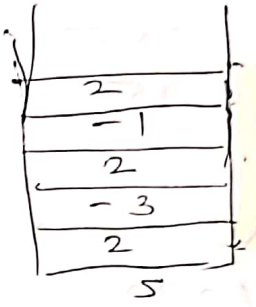


$f(push(s,-1)) = \max(f(s), 0) + (-1)$

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

$f(push(s,2)) = \max(f(s), 0) + 2$

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



Q. The best D.S. to check whether an arithmetic exp. has balanced parenthesis is

A) queue C) tree

B) stack D) list

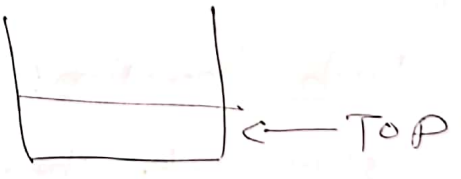
for every Right/left parenthesis, there must be a left/Right parenthesis

((())) most recently occured parenthesis

TOP in stack keeps the mostly recently added element.

The string eventually ends, and what we get is an empty stack which assures we had balanced parenthesis

i/p:) ((



Invalid

Prefix Evaluation

infix : 2 + 3 × 5

prefix : + 2 × 3 5

Reverse

prefix : 5 3 × 2 +

i) × → 3, 5

3 × 5

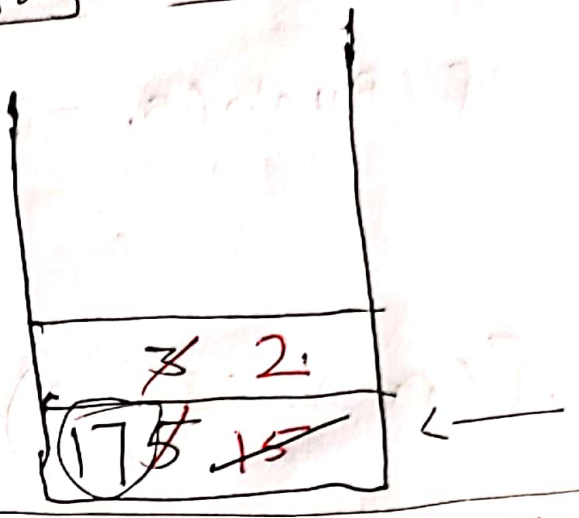
15

ii) + → 2, 15

2 + 15

= 17

using stack



When ⊛ operator is encountered,

pop 1st ele = A, pop 2nd = B

A ⊛ B

Applications of Stack:

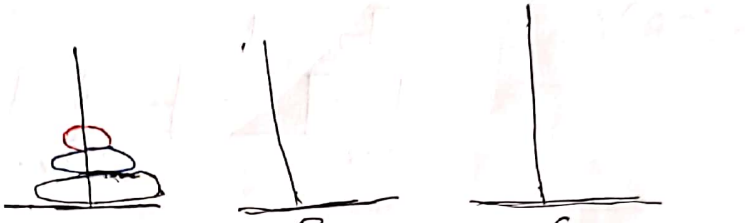
Tower of Hanoi Problem: Given 3 pegs/pillars & n discs each of distinct size

Move discs from first pillar to third, using second.

e.g. n = 3

i) We can not put a larger sized disc above smaller sized disc.

ii) Move only one disc at a time



Target: Move all discs from source to dest^n pillar

(For the largest disc to be at the bottom of dest^n pillar, then all the discs above the largest should be moved to auxillary in order to move the largest disc of the source to empty pillar of destination)

∴ The Function will look something like this:

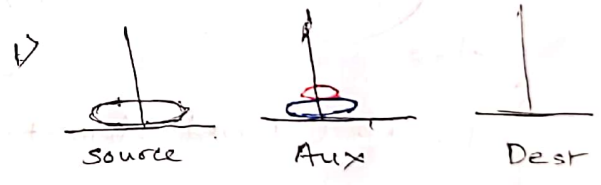
TOH (n, Source, Dest, Aux)

Algorithm:

Let us consider n = 3

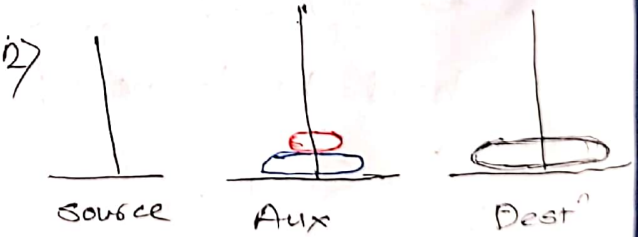
1) TOH (n-1, Source, Aux, Dest^n)

to ↗ ↖ using



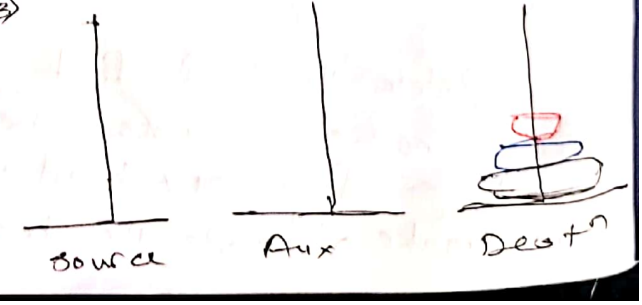
2) Move Source → Dest^n

(Simply move the largest disc to empty pillar of dest^n)



3) TOH (n-1, Aux, Dest^n, Source)

using →



Program :

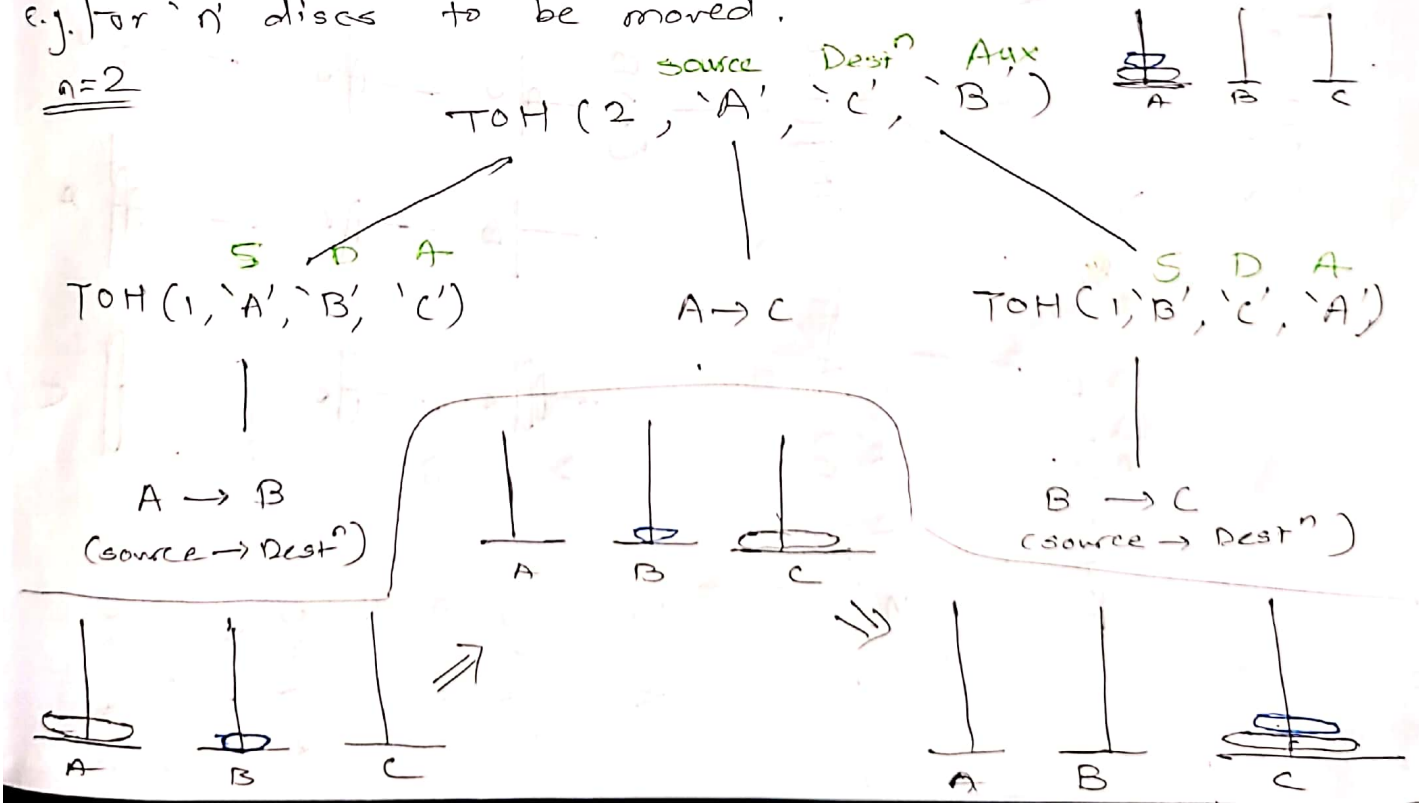
```
void main()
{
    int n;
    printf("Enter the value of n");
    scanf("%d", &n);
    TOH(n, 'A', 'C', 'B');
}

void TOH(int n, char source, char Dest, char Aux)
{
    if(n==1)
    {
        printf("%c -> %c", Source, Dest);
        return;
    }
    TOH(n-1, Source, Aux, Dest);
    printf("%c -> %c", Source, Dest);
    TOH(n-1, Aux, Dest, source);
}
```

S D A

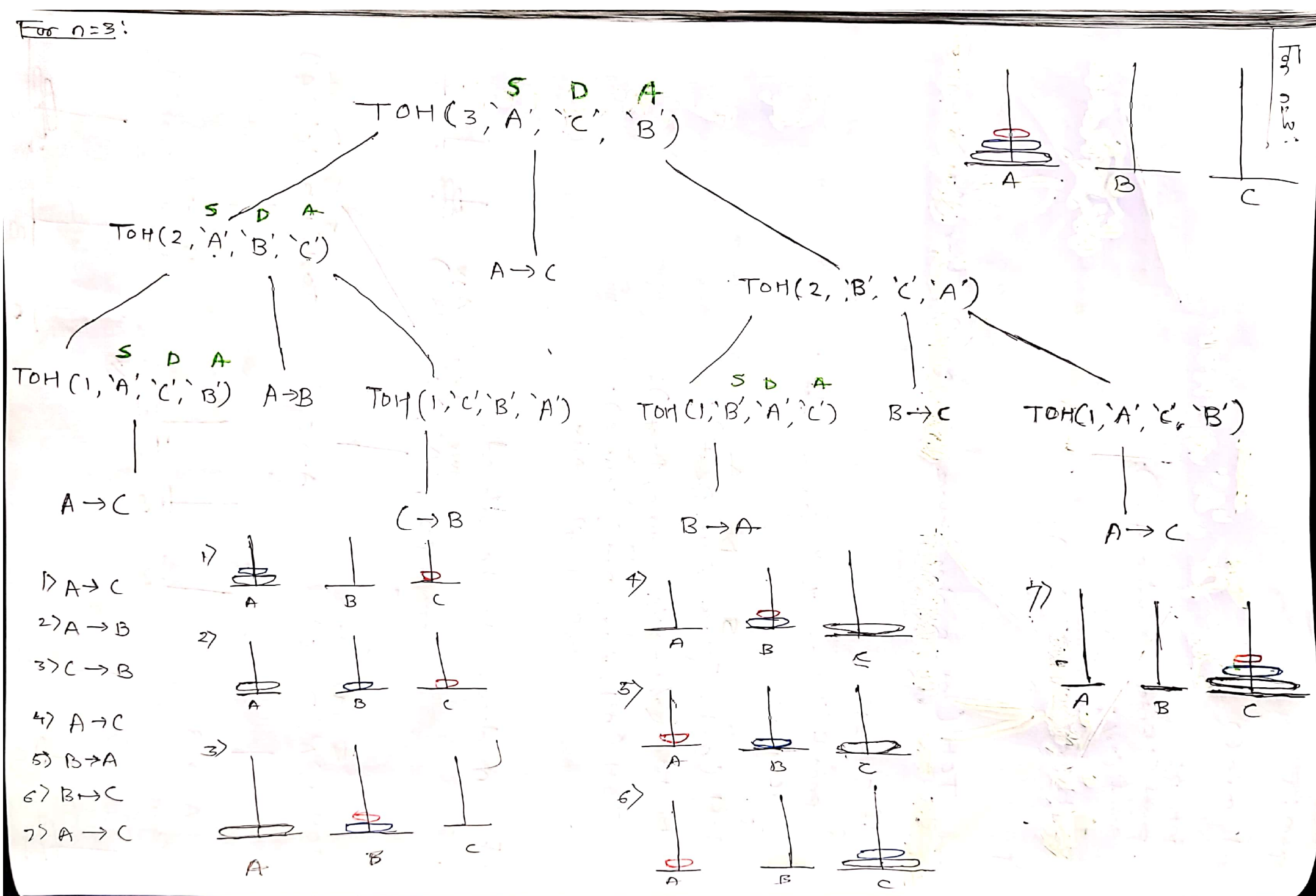
Source Dest Aux

e.g. For 'n' discs to be moved.

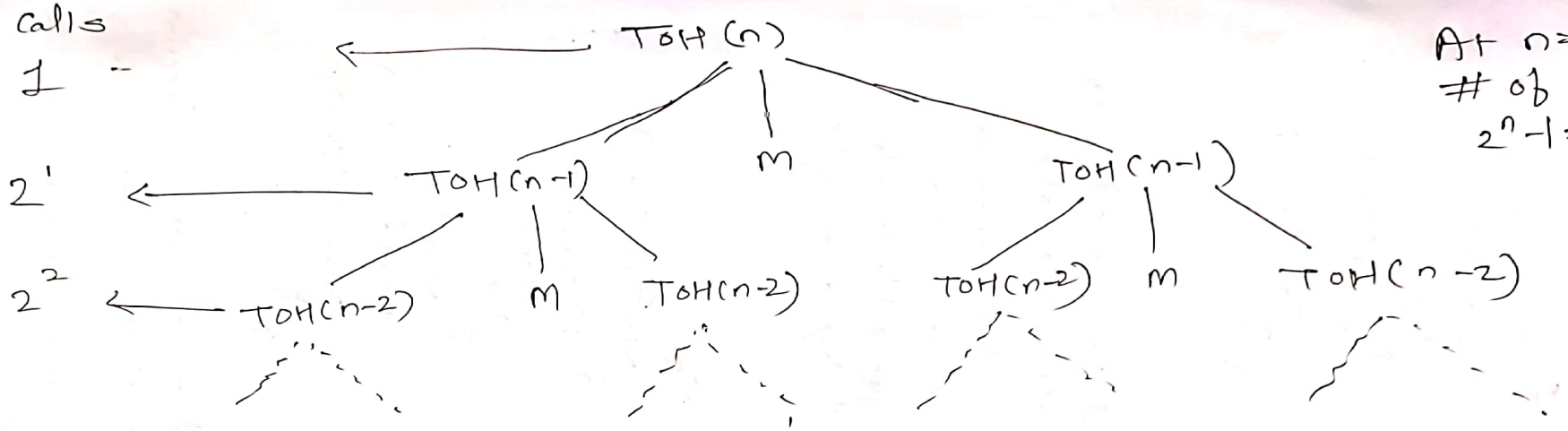
n=2



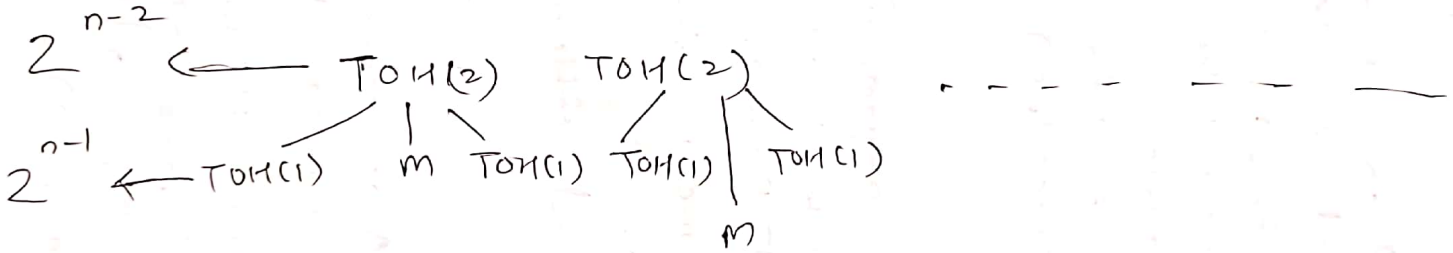
For n=3:


S D A
TOH(3,'A','C','B')
A→C
S D A
TOH(2,'A','B','C')
A→B
S D A
TOH(1,'A','C','B')
A→C
TOH(1,'C','B','A')
C→B
TOH(2,'B','C','A')
B→C
S D A
TOH(1,'B','A','C')
B→A
TOH(1,'A','C','B')
A→C
1) A→C
2) A→B
3) C→B
4) A→C
5) B→A
6) B→C
7) A→C
A
B
C

No. of calls



At n=3,
# of moves
$2^n - 1 = 2^3 - 1$
$= 8 - 1$
$= 7$



function calls $= 1 + 2^1 + 2^2 + \ldots + 2^{n-1}$ (GP n terms) $S_n = \frac{a(r^n - 1)}{r - 1}$

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

moves in TOH(n) $= 2^n - 1$

$\theta(2^n)$

Appl^n:

Fibbonacci Series.

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55
0 1 2 3 4 5 6 7 8 9 10

$Fib(n) = Fib(n-1) + Fib(n-2)$ $\quad n \geq 2$

$= 0 \quad n = 0$

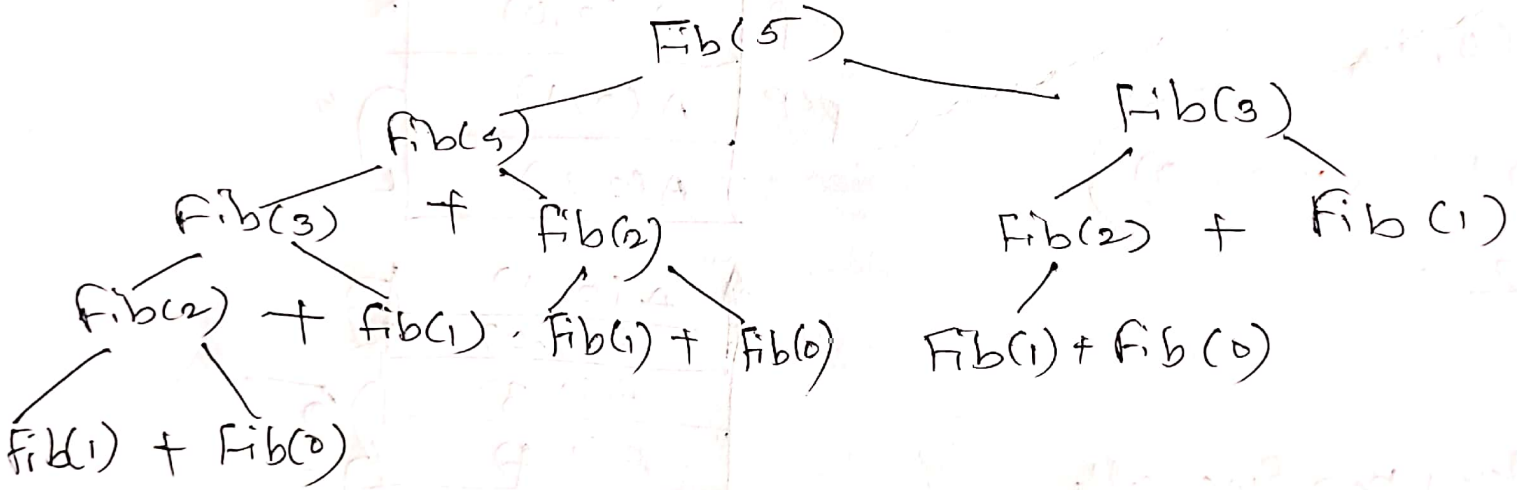
$= 1 \quad n = 1$

$Fib(0) = 0$

$Fib(1) = 1$

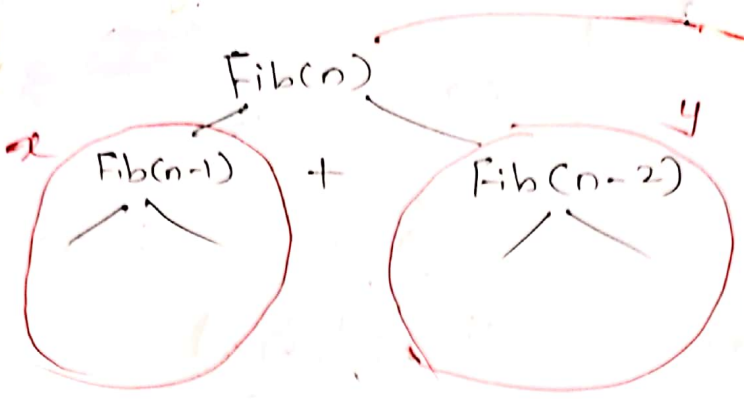
$Fib(2) = Fib(1) + Fib(0) = 1 + 0 = 1$

$Fib(3) = Fib(2) + Fib(1)$
$= 1 + 1 = 2$



$O(2^n)$

* Total No. of function calls in Fib(n)



Let H(n) be the # of func calls in Fib(n)

$H(n) = H(n-1) + H(n-2) + 1$

$H(0) = 1$
$H(1) = 1$

$H(n) = 2 \times Fib(n+1) - 1$

⇒ New Fibonacci series

$H(n) = H(n-1) + H(n-2) + 1$

$H(0) = 1$
$H(1) = 1$

$H(2) = H(1) + H(0) + 1 = 1 + 1 + 1 = 3$

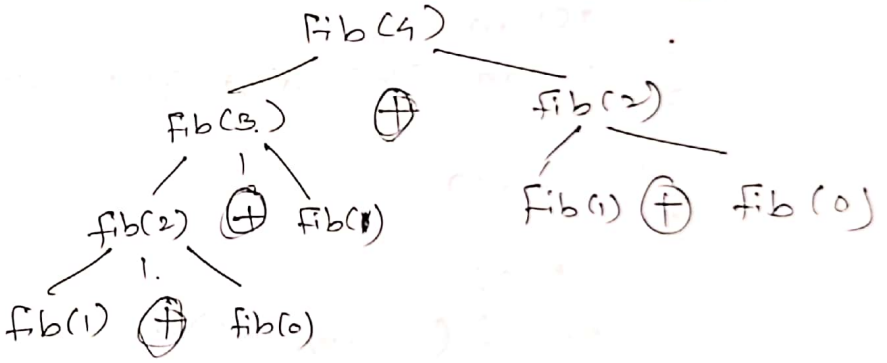
$H(3) = H(2) + H(1) + 1 = 3 + 1 + 1 = 5$

n	0	1	2	3	4	5	6
H(n)	1,	1	3	5	9	15	25

---

* Total no. of add^{ns} in calculating Fib(n) (Total # of Additions)

Fib(0) = 0
Fib(1) = 0
Fib(2) = 1
Fib(3) = 2
Fib(4) = 4
Fib(5) = 7



Let G(n) be the no. of addition in Fib(n)

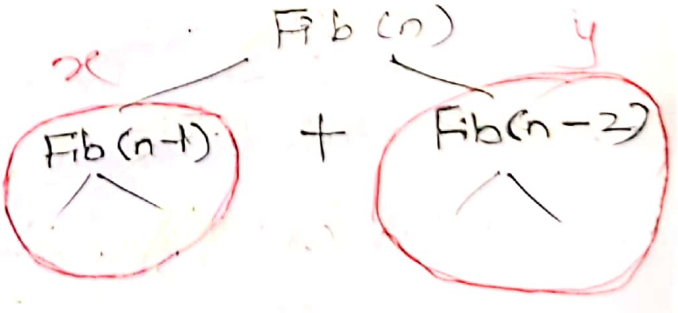
$G(0) = 0$
$G(1) = 0$

$G(n) = G(n-1) + G(n-2) + 1$

$G(0) = 0$

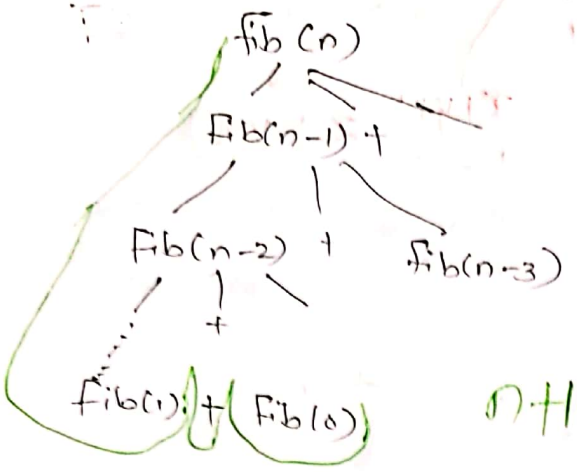
$G(1) = 0$

n	0	1	2	3	4	5	6
G(n)	0,	0,	1,	2,	4,	7,	12

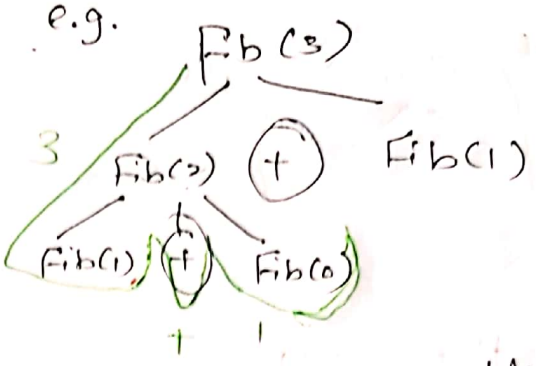


* No. of function calls before 1st addition take place in Fib(n)

invoked



e.g.



First addition will take place when both operands are present.

No. of Add^ns = 2 — No. of fun calls bf 1st add^n = 4
— 1st func calls = 5

Program:

```
long long int fib(int n)
{
   if (n==0 || n==1)
      return n;
else→ return fib(n-1) + Fib(n-2); //Binary Recursion
}
int main()
{
   int n;
   printf("enter n");
   Scanf("%d", &n);
   printf("%lld", fib(n));
}
```

No. of fun invocations for fib(n) = 2 * fib(n+1) - 1
— 1st additions for fib(n) = fib(n+1) - 1

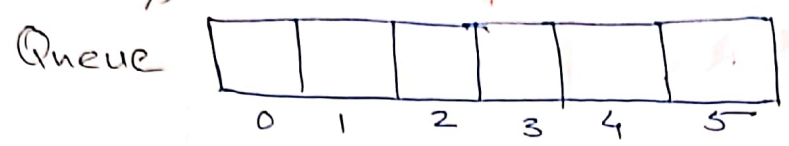
e.g. calculate: no. of invocations in fib(9) = 2 * fib(10) - 1
= 109

— 11 — additions in fib(8) = fib(9) - 1

Queue

* Linear D.S.
* First In First Out
* Last In Last Out
* Applications:
  CPU scheduling,
  semaphores, Memory Management, etc.
* Order of deletion is same as order of insertion.
* Insertion : Rear
  Deletion : Front

(Using Array)




```
#define SIZE 6
int Queue[SIZE];
int Rear;
int Front;
```

Front : Index from which an element can be deleted.

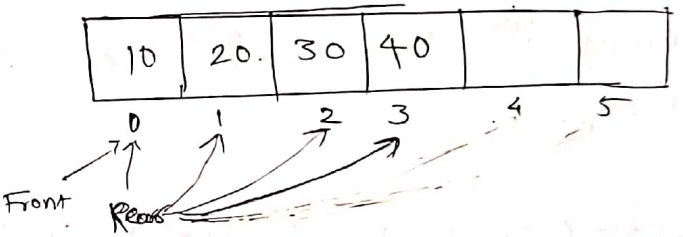
Rear : " of most recently added element.

{Front = -1, Rear = -1} Initially, when Queue is empty

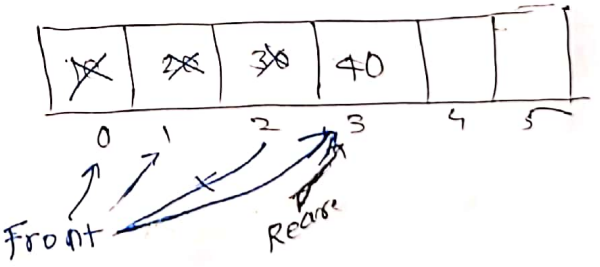
Case 1 : When we insert 1st element

Empty Queue ⇒ F = R = 0

	F	R
Initially	-1	-1
Insert (10)	0	0
Insert (20)	0	1
Insert (30)	0	2
Insert (40)	0	3
delete	1	3
delete	2	3
delete	3	3




```
Rear++
Queue[Rear] = x;
```



After deletion of 3 ele, Front == Rear

Case 1 : When Queue is Empty

Front == Rear = -1

Case 2 : When there is only 1 ele in Queue.

Front == Rear ≠ 1

* Insertion Operation (EnQueue)

EnQueue ⇒ O(1)

```
void EnQueue(int x)
{
  if (Rear == SIZE-1)
     return;                    // When the Queue is full
  else if (Front == -1)
     {
      Rear = Front = 0;         // When there's only one ele in queue.
      Queue [Rear] = x;
     }
  else
     {
       Rear ++                  // otherwise
       Queue [Rear] = x;
     }
}
```

* Deletion Operation (DeQueue)

DeQueue ⇒ O(1)

```
int DeQueue ()
{
  if (Front == -1)      // When the Queue is Empty.
     return INT_MIN;
  else if (Rear == Front)   // When there's only 1 ele. in queue.
     {
       temp = Queue [Front];
       Front = Rear = -1;
     } return temp;
  else {
       temp = Queue [Front];
       Front ++;
     } return temp;
}
```

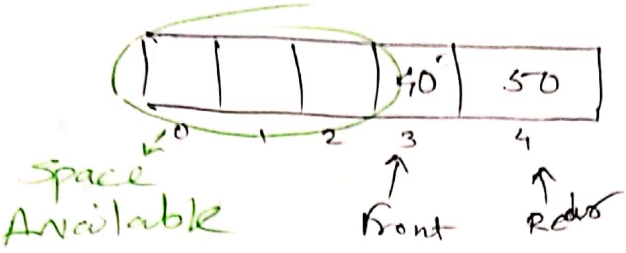
2 or more ele.

Front = Front + 1

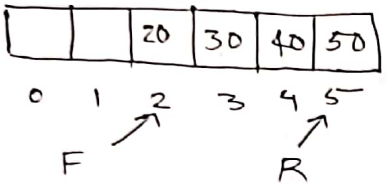
Rear remains unchanged.

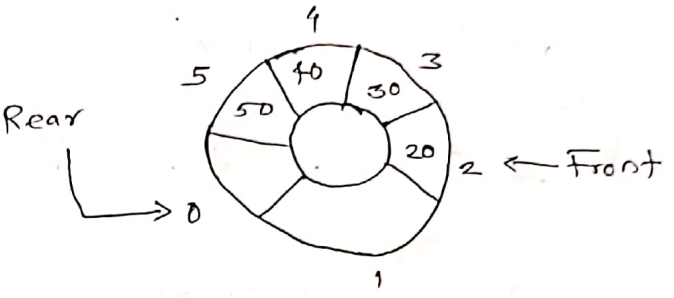
Just extract 1 ele

Drawback: In simple Queue, if Rear is pointing to last element, the Queue is Full, even if the Front is not pointing to the first & has performed some Enqueue operations.



Sol^n: Circular Queue.



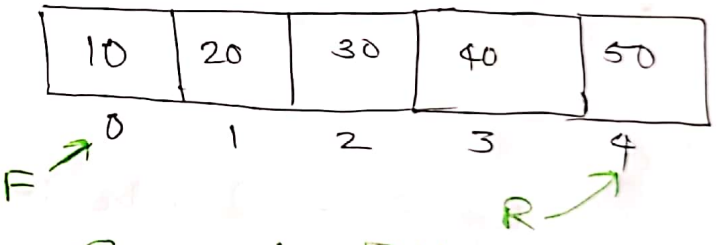


Insert (50) ⇒ EnQueue (50)

EnQueue (60) ⟶ Rear = 0

Queue[Rear] = x;

	F	R
Initially	-1	-1
EnQueue(10)	0	0
EnQueue(20)	0	1
EnQueue(30)	0	2
EnQueue(40)	0	3
EnQueue(50)	0	4



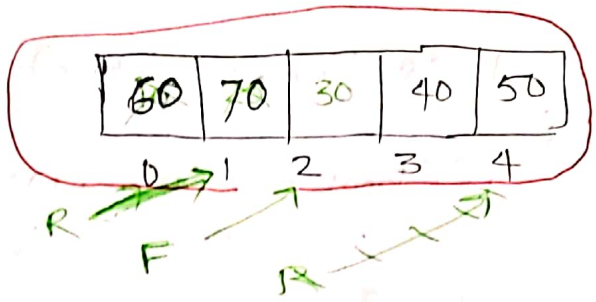
Queue is Full

Front == 0 && Rear == SIZE-1

This is Case 1.

Case 2: If F == (R+1) % SIZE, Q is full.

	F	R
Initially	-1	-1
Enqueue 10	0	0
Enqueue 20	0	1
Enqueue 30	0	2
Dequeue	1	2
Dequeue	2	2
EnQueue 40	2	3
EnQueue 50	2	4
EnQueue 60	2	0
EnQueue 70	2	1
Delete()	3	1
Delete ()	4	1
Delete ()	0	1



Now, the Queue is full.

∴ Rear = Front - 1

OR

Front = R + 1

Case 2: In Circular Queue,

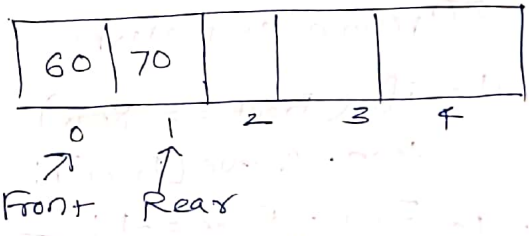
$$F == (R+1) \% \text{ SIZE}$$

In the above eg., F = 2, R = 1

∴ 2 == (1+1) % 5

2 == 2 ⟹ True

∴ Q is Full.



While Deletion, rear remains unchanged whereas

" Insertion, front —— " ——

Insertion order →

~~30~~, ~~40~~, ~~50~~, 60, 70

∴ First element to be deleted is the first that was added i.e. 30

```
Void CQ_Enqueue(int x)
{
  if (Front == (Rear+1)% SIZE) // When queue is full.
      return;
  else if (Rear == SIZE-1)  // When rear is pointing to last ele.
      Rear = 0;
  else if (Front == -1)  // When queue is empty
      Front = Rear = 0;
  else
      Rear++
      Queue[Rear] = x;
}
```

O(1)

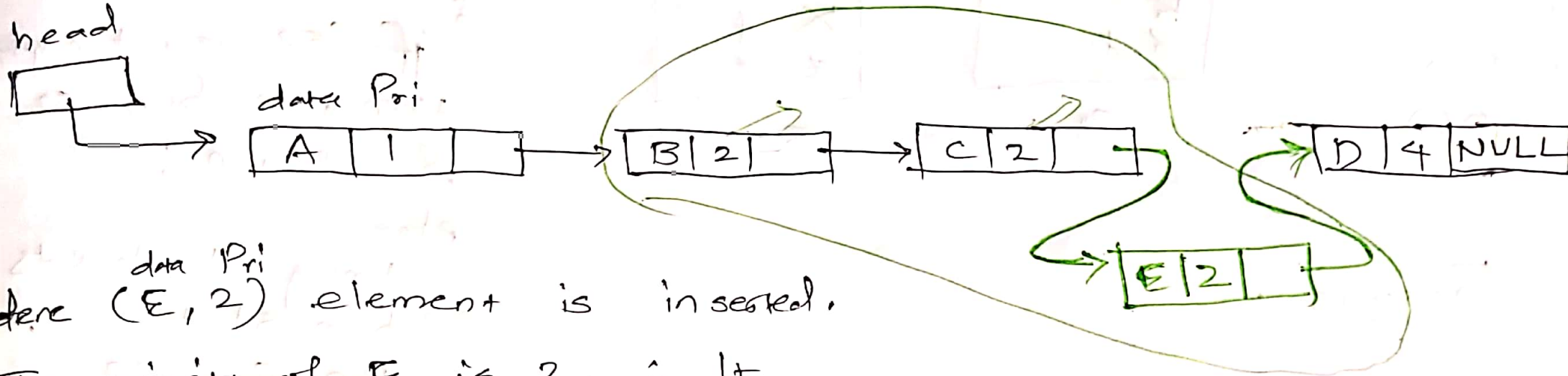
```
int  CQ_Delete()
{
  if (Front == -1) // When Queue is Empty
      pf("Underflow");
      return INT_MIN;
  else if (Front == Rear) // When there's only one ele in Q.
      temp = Queue[Front];
      Front = Rear = -1;
      return temp;
  else if (Front == SIZE-1) // When Rear is pointing to last ele
      temp = Queue[Front];
      Front = 0;
      return temp;
  else
      temp = Queue[Front];
      Front++;
      return temp;
}
```

Can be combined with

```
int temp = Queue[front];
else
 front = (front+1)% SIZE;
return temp;
```

Priority Queue

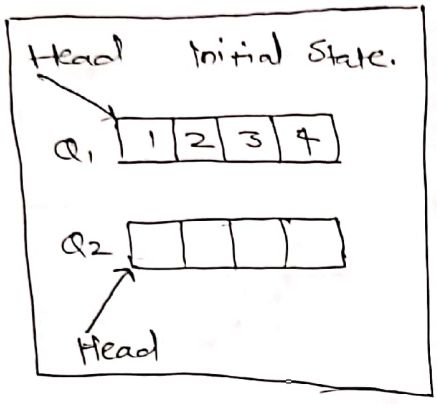
* A priority is associated with every element.
* Elements will be processed as per priority.

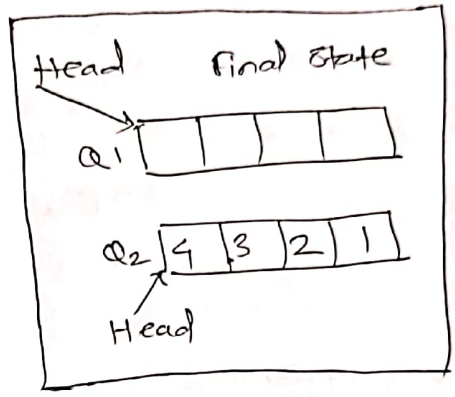


Here (E, 2) element is inserted.
data Pri

The priority of E is 2. ∴ It will be inserted before D that has priority 4. It will be inserted after B & C since they have arrived earlier.

Q. Consider the queues Q_1 containing four elements & Q_2 containing none (shown as the initial state in the fig). The only operations allowed on these two queues are Enqueue(Q, element) and Dequeue(Q). The min no. of Enqueue operations on Q_1 req^d to place the elements of Q_1 in Q_2 in reverse order (shown as the Final state in the figure) without using any additional storage is ___0___.





2 MARKS 2022

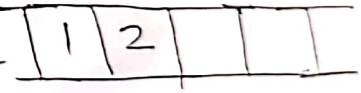
To place elements from Q_1 to Q_2, we should Dequeue from Q_1 & Enqueue it to Q_2.

Dequeue(Q_1) Q_1 

∴ We can write

Enqueue(Q_2, Dequeue(Q_1))

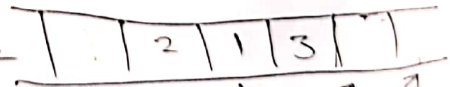
Enqueue(Q_2, Dequeue(Q_1))

Q_2 

Order

Now, To reverse the order of Q_2, Dequeue it & again enqueue it.

∴ Enqueue(Q_2, Dequeue(Q_2)) Q_2 

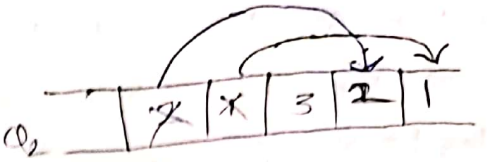
Enqueue(Q_2, Dequeue(Q_1)) Q_2 

Change the order

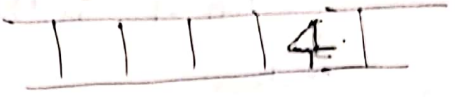
To change the order,

Enqueue (Q_2 (Dequeue (Q_2))

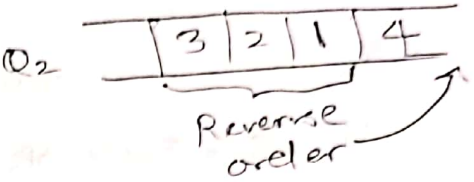
Enqueue (Q_2 (Dequeue (Q_2))



Now, Q_1 is

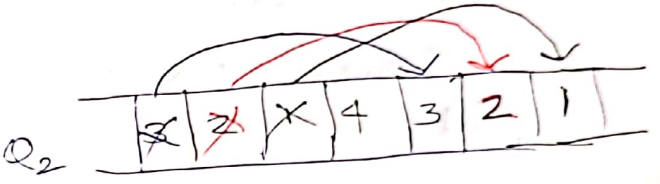


Enqueue (Q_2, Dequeue (Q_1))



∴ Dequeue 3 times & enqueue to itself.

Enqueue (Q_2 (Dequeue (Q_2)) ⇒ 3 times

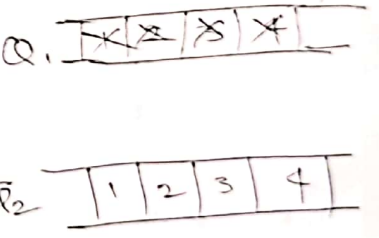


∴ Q_2 4 3 2 1

Note that we have not performed a single Enqueue operation on Q_1 to place all ele. of Q_1 in Q_2 in reverse order.

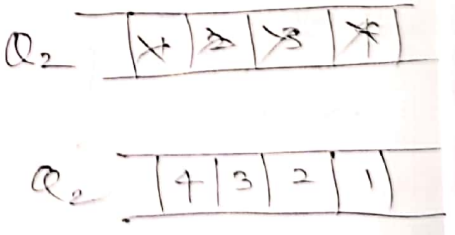
∴ Summary:

Enqueue (Q_2, Dequeue (Q_1)) } 4 times



Now Reverse :-

Enqueue (Q_2, Dequeue (Q_1)) } 4 times



Q. Let Q denote a queue containing sixteen numbers and S be an empty stack.

Head(Q) returns the ele. at the head of the queue Q w/o removing it from Q.

Top(S) ——— " ——— top of the S ——— " ——— S.

Consider the algo^m given below.

```
While Q is not empty do
    If S is Empty OR TOP(S) ≤ Head(Q) then
      x: Dequeue(Q);
      Push(S, x);
    Else
      x: Pop(S);
      Enqueue(Q, x);
End
End
```

∴ For 16 elements, $16^2 = 256$
∴ 256 iter^{ns} of while loop.

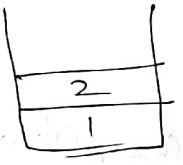
The max^m possible no. of iterations of the while loop in the algo^m?

⇒ Let us consider Q contains 3 elements

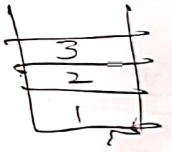
n = 3 | 1, 2, 3

While Q is not empty — If is true — 1st iteration

2nd iteration — While true — if "



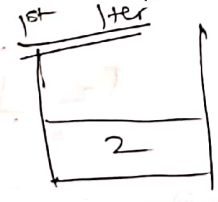
3rd



Let us change the order

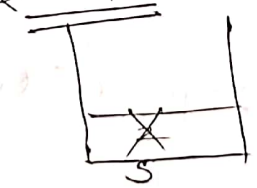
2, 1, 3

While Q is not empty — If cond^n true — 1st iter^n



Q' | 1, 3

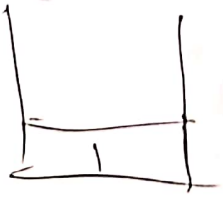
2nd iter^n



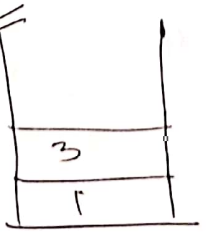
| 1, 3, 2 |
Q

If cond^n fails, goto else, true, pop & enqueue back

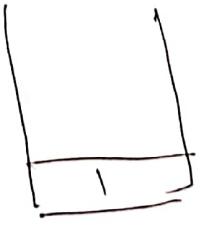
3rd iter^n — Q | 3 | 2 — If true



4th — Q | 2 — If true

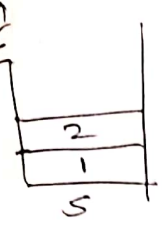


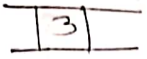
5th — If fails else true, pop & Enq



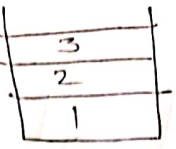
Q | 2 | 3

6th iter^n
If is true



Q 

7th iter^n
If true

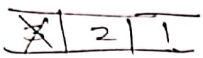


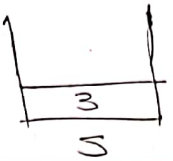
8th iter^n:
While false, control returns from while.

For 2, 1, 3 order 7th iter^ns reqd.

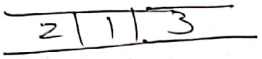
Worst case order where elements are in descending order
3, 2, 1

1st Iter^n: While is true, If is true (stack empty)
∴ Pe, ene & Push

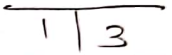
Q 

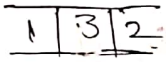


2nd Iter^n:
While true
Else true
∴ Pop & Enqueue back

Q 

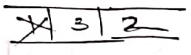
3rd Iter^n:
while T,
If T

Q 

4th Iter^n: Q 

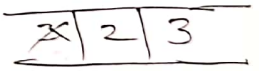
If fails
Else true

5th:
If T
True

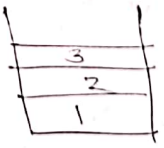
Q 

Now, 1 is the lowest no. in the queue, so it will never pop.

After 2 more Iter^ns: 3 is pushed (6th)
& popped back (7th)

Q 

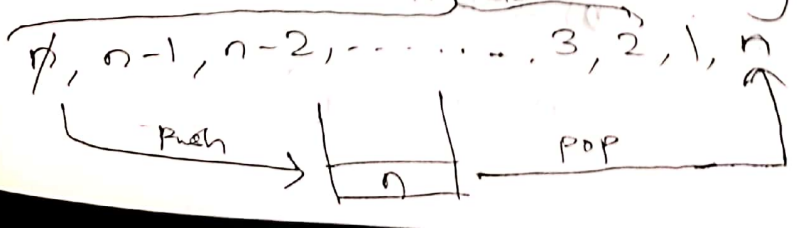
After 2 more Iter^ns: 2 is pushed (8th)
3 —"— (9th)



10th Iter^n:
While false.

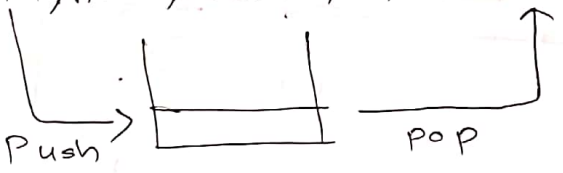
∴ For n=3, 9 max^m Iterations of while loop are possible.
i.e. For n ele, n^2 Iterations.

Proof: For n elements in descending order



After 2 iterations, n goes in the end
↑ Largest no.

Now, Q is $n-1, n-2, \ldots 3, 2, 1, n$



After 2 More Iterations Q is $n-2, n-3, \ldots 3, 2, 1, n, n-1$

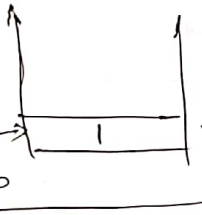
For all 1st $(n-1)$ elements, 2 iterations are req'd each.

$n, n-1, n-2, \ldots 3, 2, 1$

$n-1$ elements, 2 iter^{ns} each.

$\therefore$ $n, n-1, n-2, \ldots 3, 2, 1 \xrightarrow[\text{iterations}]{2(n-1)} 1, n, n-1, n-2 \ldots 3, 2$

$\xrightarrow[\text{iteration}]{1 \text{ more}}$



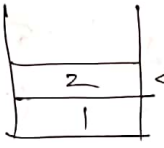
1 is pushed & it will never get popped as it is lowest.

$\therefore$ n sized Queue $\xrightarrow{2(n-1)+1}$ $n-1$ sized Queue

$n, n-1, n-2, \ldots 3, 2, 1$ $\quad$ $n, n-1, \ldots 3, 2$

$n-2$ elements, 2 iternd each.

$\therefore$ $n, n-1, n-2, \ldots 3, 2 \xrightarrow{2(n-2)+1}$ (1 of push)



$n-1$ sized Queue $\xrightarrow{2(n-2)+1}$ $n-2$ sized Queue.

$\therefore$ To reduce the Queue size i.e. to dequeue all the elements in order to ~~pop~~ make Stack full & end while loop, We have to perform this iteration "n" times.

i.e. $2(n-1)+1, 2(n-2)+1, 2(n-3)+1, \ldots 2(1)+1, \underbrace{2(0)+1}_{\text{push last ele.}}$

Sum $= 2(n-1+n-2+n-3, \ldots 1+1) + 1+1+1 \ldots n$ times.

i.e. $2(1+2+3+\ldots(n-1)) + n$ $\quad$ – We know, $\frac{N(N+1)}{2} = S_n$.

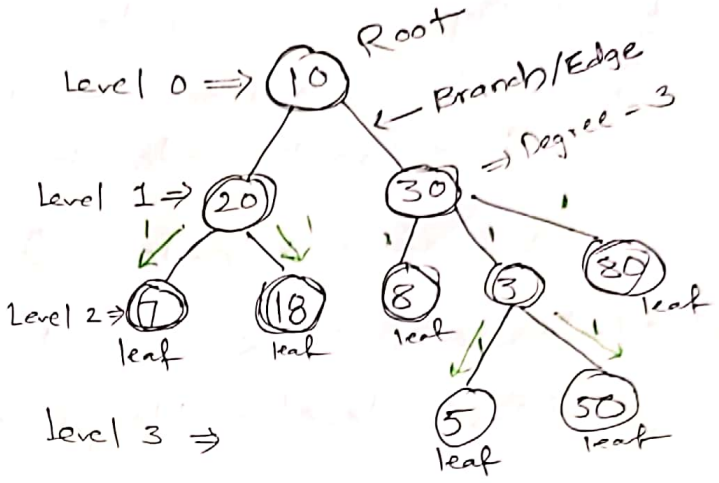
$\therefore S_{n-1} = \cancel{2}\ \frac{(n-1)(n)}{\cancel{2}} + n = n^2 - n + n = \underline{\underline{n^2}}$

Tree

Undirected, acyclic, connected graph.

Non-Linear D.S. used to store hierarchical data.

Used in
1) Directory structure
2) organization
3) HTML/XML
4) Parse tree
5) BST
6) Binary Heap
7) B-Tree, B+ Trees in DBMS.



tree with height = 3

1) Node: Each element in tree is represented by a node.

2) Child: Child is a node immediately below a node.
20 & 30 are children for node 10
7 is a child of 20.

3) Parent: Node immediately above is Parent.
30 is a parent of 8, 3, 80

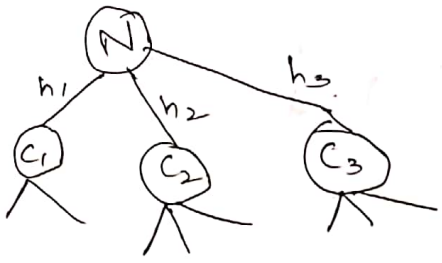
4) Root Node: Distinguishable Node.
Only node w/o any parent.
10 is the root node.
(Topmost node)

5) Leaf Node: Node w/o any child
7, 18, 8, 5, 50, 80

6) Internal Node: Node with at least 1 child. e.g. 10, 20, 3, 30
(Nodes except leaf nodes)

7) Degree of a node: It represents the no. of children of a node in a tree.
Degree of node with key 10 ⇒ 2
" " " 30 ⇒ 3

8) Height of a node:
It is the length of path from node "x" to the farthest leaf node.
Height of a leaf node = 0.
Ht. of 20 = 1
Ht. of 30 = 2 (1+1)



$h(N) = 1 + \max(h_1, h_2, h_3)$

Maxm height in B.T. = n-1

$\therefore H(10) = 1 + \max(h(20), h(30))$

$= 1 + 2 = 3$

No. of Edges from that node to the longest leaf is height of a node.

~~height of tree~~ = height of root node

9) Level/Depth of a Node:

level of a node x is the length of the path from root node to node x.

Level of root node = 0

No. of Edges from the root to that node is depth of a node

If Node Level is k, level of it's child is k+1.

~~Maxm~~ Maxm level = Height of tree

10) Ancestor of a node: if there's a path from node p to node q, then all the nodes in the path other than q are called as ancestors of q.

3, 30, 10 are ancestors of 50.

10 has no ancestor.

20 - 10	7 - 20, 10	8 - 30, 10	5 - 3, 30, 10
30 - 10	18 - 20, 10	80 - 30, 10	50 - 3, 30, 10

11) descendent of a node: 7 → 20, 10 (Ancestor)

7 is a descendent for 20

7 — 11 — 10

All nodes are descendants of 10. 10 is the Ancestor of all

12) Sibling: Nodes with same parent.

7, 18 are siblings

8, 3, 80 — 11 —

2, 30 — 11 —

Root Node can not have a sibling as it does not have a parent.

3) Generation: All Nodes at a particular level belongs to same Generation.

4) size of a node: No. of descendents of a node (including the node itself).

* size of leaf node ⇒ 1

size = descendents + 1 (itself)

size of node 3 ⇒ 3

size of node 30 ⇒ 6 (5+1)

size of node 20 ⇒ 3 (2+1)

size of node 10 ⇒ 3+6+1 = 10

∴ size of a node = size of it's descendents + 1

Binary tree

Every node can have at most 2-child.

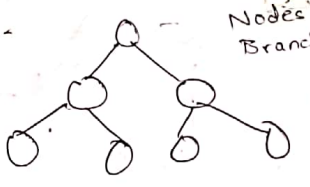
Node can have 0-child. (leaf node)
1- "
2- " [Internal node]

i. A binary tree, i.e. at each level, there're max^m no. of elements / nodes present is a Full Binary Tree / Complete or Perfect Binary Tree.

Max^m no. of nodes at level "l" of a B.T. is 2^{l-1} (Proof by Induction)

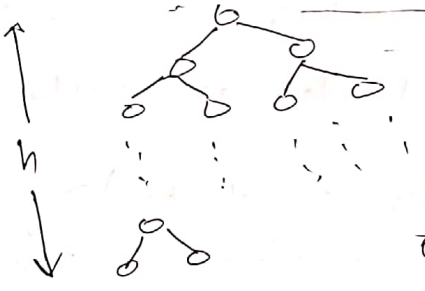
Max^m no. of nodes possible in a binary tree of height h?

For h=2



Level	# of Nodes	max^m # of Branches in BT
0	2^0	= N-1
1	2^1	= 7-1 = 6
2	2^2	
	7	

∴ for h =



level	# of Nodes
0	2^0
1	2^1
2	2^2
⋮	⋮
h	2^h

Total = $2^0 + 2^1 + 2^2 + \cdots + 2^h$

$= 1 + 2 + 2^2 \cdots\cdots + 2^h$ (G.P.)

$= \frac{2^{h+1} - 1}{2 - 1} = 2^{h+1} - 1$

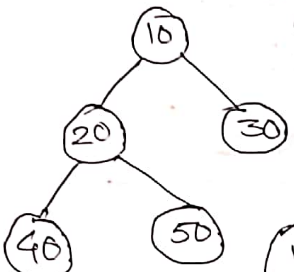
∴ $n_{max} = 2^{h+1} - 1$

Min^m no. of nodes in a binary tree of height h

i.e. $n_{min} = h+1$ — We add 1 as height count starts from zero.

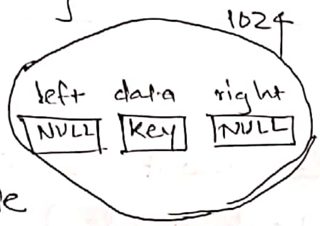
$\therefore$ for $h=2$, $n_{min}=3$.

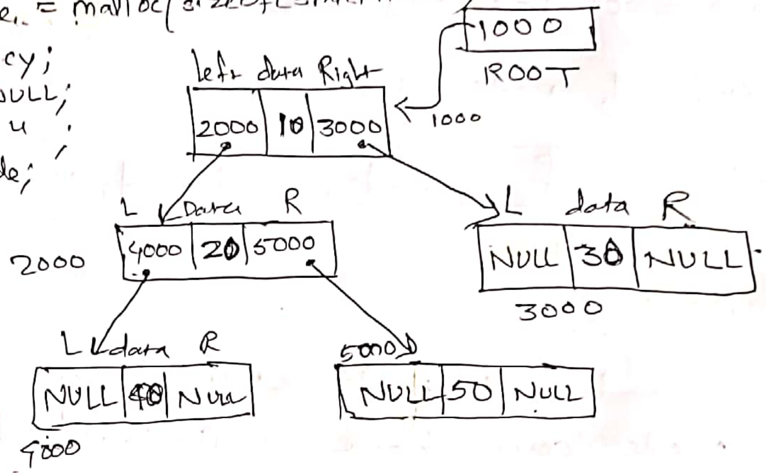
Root:



Node creation program

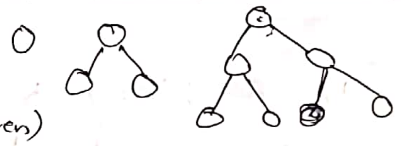
```
struct node *getnewNode(int key)
{
struct node *newNode = malloc(sizeof(struct node));
  newNode->data = key;
  newNode->left = NULL;
  —||— ->right = "  ;
  return newNode;
}
```



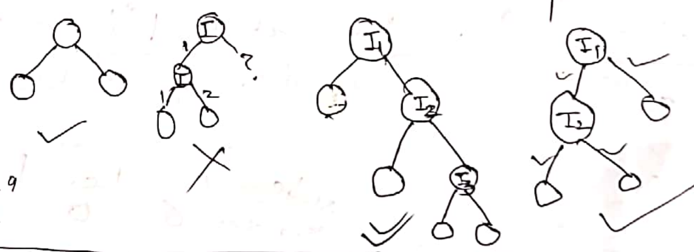



```
Struct Node
{
  struct Node *Left;
      int data;
  struct Node *Right;
};

Struct Node *ROOT;
ROOT=malloc(sizeof(struct node));
```

Full Binary Tree :
strict —||— —||—
(strictly 0 or 2 children)



2-ary tree :- A tree in which every Internal node has exactly 2-children



Every Full Binary Tree is a 2-ary tree. "Perfect B.T"

Total no. of nodes $\Rightarrow$ No. of Internal Nodes $\times$ 2 + 1

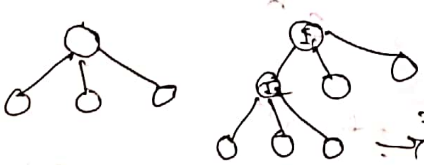
(2 → Each Internal Node has 2 childs; 1 → Root)

Total nodes = 3 Internal nodes $\times 2 + 1$
$= 6+1 = 7$

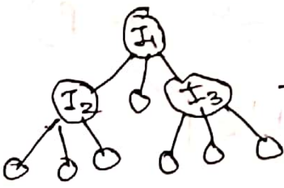
3-ary tree: A tree in which every internal node has exactly 3 childs.

Total No. of nodes = No. of Internal nodes $\times$ 3 + 1 (Root)

(3 → Each IN has 3 childs)



$\Rightarrow$ # of nodes $= 2\times3+1$
$= 6+1 = 7$



$\Rightarrow$ # of nodes = No. of Internal nodes $\times$ Nodes because of these IN + 1 (Root)

$= 3\times3+1 = 10$

k-ary tree: A tree in which every internal node has k-childs.

Let I be the no. of internal nodes in a k-ary tree.

∴ Total nodes = $I \times k + 1$ (root)

(No. of IN; Each IN has k-childs)

∴ $n = k \cdot I + 1$ — (1)

$n = k \cdot I + 1$

# of leaf nodes + # of internal nodes = $k \cdot I + 1$

$L + I = k \cdot I + 1$

∴ $L = k \cdot I - I + 1$

∴ $L = (k-1)I + 1$ — (2)

# of leaf nodes.

$n = f(L)$

We know,

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

∴ $I = \frac{L-1}{k-1}$

$n = k\left(\frac{L-1}{k-1}\right) + 1$ (from (1))

∴ $n = \frac{kL - k + k - 1}{k-1}$

$$n = \frac{k \cdot L - 1}{k-1}$$, where

K = No. of childs for each IN

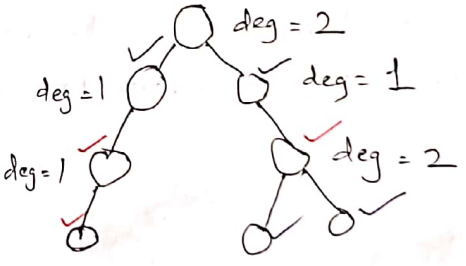
L = No. of leaf nodes

A binary tree with

2 nodes of degree 2

3 —"— 1,

Find the no. of leaf nodes.



$n = 2 \times 2 + 3 \times 1 + 1$ (root)

Total no. of nodes $n = 8 = L + IN$

$n = 8$

$IN + L = 8$

$5 + L = 8$

∴ $L = 3$

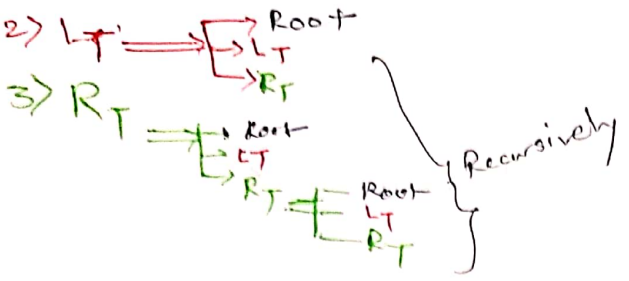
No. of leaf nodes = 3

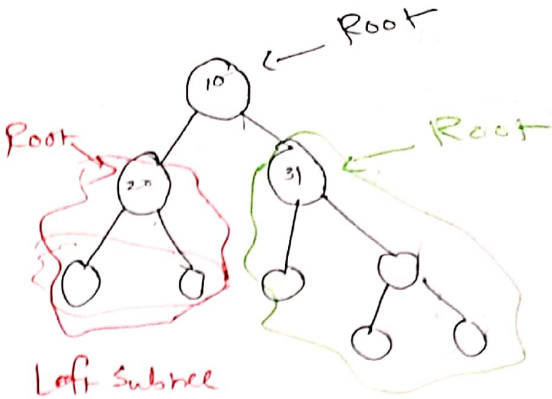
Tree Traversal

Tree consists of:

1) Root
2) L_T
3) R_T

Recursion



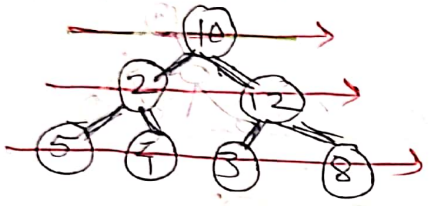


Left subtree of R.N (L_T)

Right subtree of root node

Traversal

Level order



10, 2, 13, 5, 4, 3, 8

Depth order

Root/ Node, L_T, R_T

$3! = 6$ ways.

i) Root, L_T, R_T
ii) L_T, Root, R_T
iii) L_T, R_T, Root

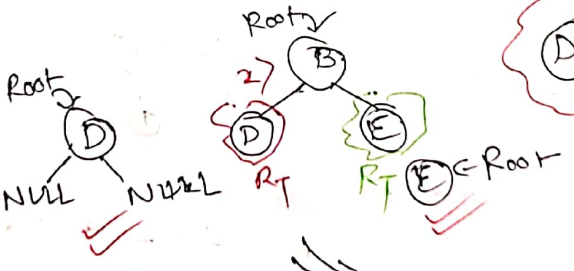
} most used in CS where left subtree is before right subtree ✡

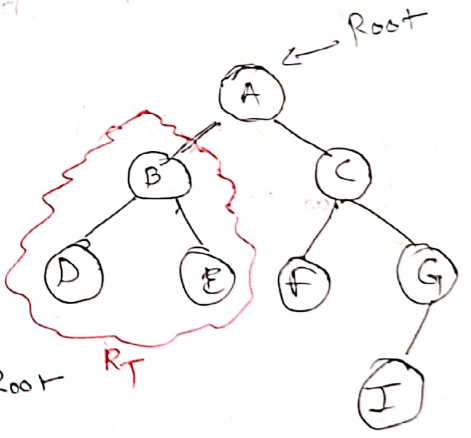
iv) R_T, L_T, Root
v) R_T, Root, L_T
vi) Root, R_T, L_T

* Pre-order Traversal

1) Visit/Print/Process the root node.
2) Traverse L_T of root node in pre-order.
3) — 4 — R_T —

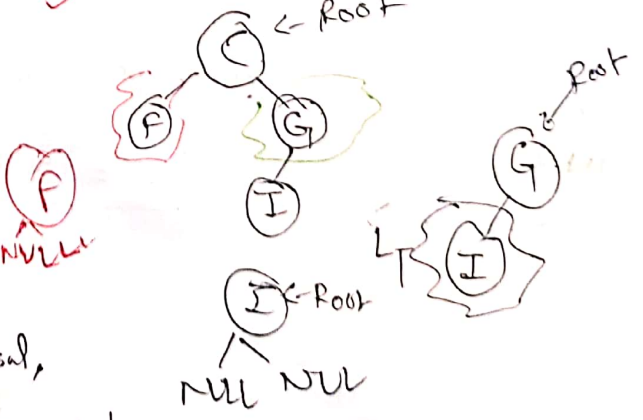
Root, L_T, R_T





Print: A B D E C F G I

Root L_T R_T



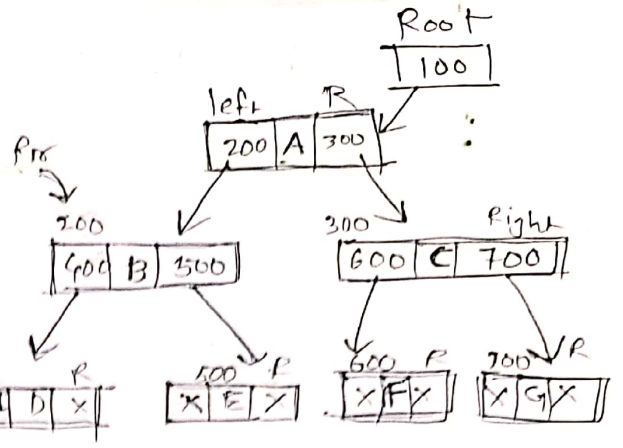
Program:

```
struct Node
{
    struct Node *left;
           int data;
    struct Node *Right;
};
```

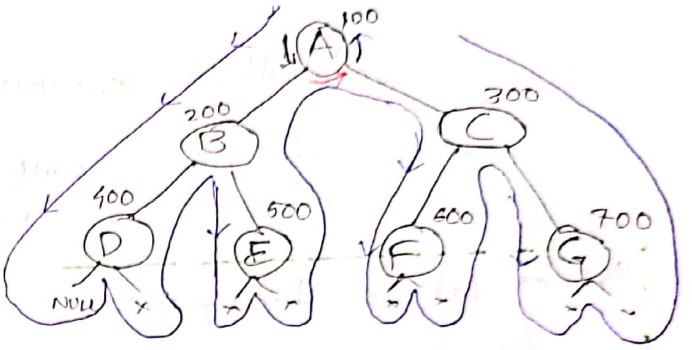
Every Node will be visited three times during traversal, preorder is the printing of node during it's first visit

Program:

```
void main()
{
   ========
   Preorder(ROOT)
   ========
}
```




```
Preorder(struct Node *Ptr)
{
   if( Ptr != NULL)
   {1> pf("%d", ptr->data);
    2> Preorder(Ptr->left);
     3> --||-- (Ptr->Right);
     4> return
     }
}
```



* In-order Traversal

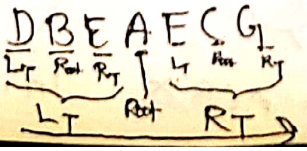
(L_T, Root, R_T) (In ber" Left & Right)

1) Traverse L_T of root node in In-order.
2) Print/Visit/Process root node.
3) Traverse R_T of root node in In-order.

(In-order is printing of noodle during it's 2nd visit)


```
void Inorder(struct Node *Ptr)
{
  if(Ptr) {
  1>Inorder(Ptr->left);
  2> pf("%d", ptr->data);
  3> Inorder(Ptr->right)
  }
}
```

Output: D B E A E C G



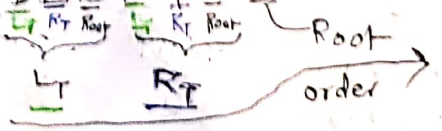
* Post-order Traversal (L_T, R_T, Root)

1) Traverse L_T of root node in Post-order.
2) --||-- R_T --||--
3) Print/Visit/Process root node.

(Post-order is printing node during its third or last visit)


```
Void Postorder(struct Node *Ptr)
{
   if(Ptr) {
     1> Postorder(Ptr->left);
     2> --||-- (Ptr->Right);
     3> printf("%d", ptr->data);
     }
}
```

Output: D E B E G C A



No. of unlabelled binary trees with n nodes = $\frac{^{2n}C_n}{n+1}$
(shape / structure / Geometry)

$n=1 \Rightarrow \frac{^2C_1}{1+1} = \frac{2}{2} = 1$

$n=2 \Rightarrow \frac{^4C_2}{3} = \frac{1}{3} \times \frac{4!}{2! \times 2!} = 2$

$n=3 \quad \frac{^6C_3}{4} = \frac{1}{4} \times \frac{6!}{3! \times 3!} = 5$

For each unlabelled structure $\Rightarrow n!$ possible ways to label.

$\therefore$ Labelled binary trees with n nodes = No. of structures / unlabelled binary trees with n nodes $\times n!$

No. of Labelled binary trees with n nodes = $\frac{^{2n}C_n}{n+1} \times n!$

Q. No. of binary trees with preorder ABC

Ans: No. of structures possible = $\frac{^{2n}C_n}{n+1}$, Possible ways to label = $n!$. Here $n=3$

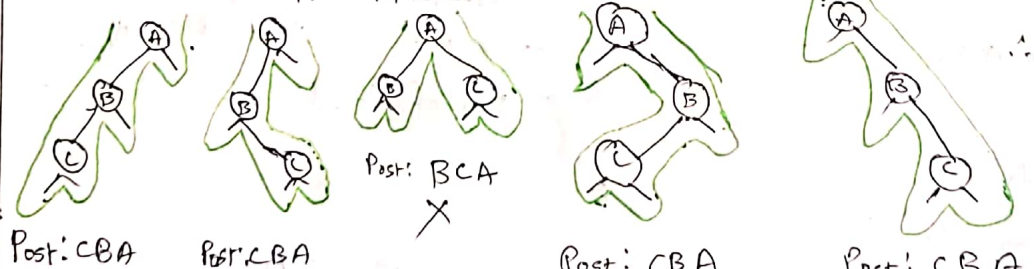
possible ways to label with preorder ABC = 1.

$\therefore$ No. of binary trees with preorder ABC = $\frac{^6C_3}{4} = 5$

$\therefore$ With a given In/Pre/Post order (n length), no. of B.Ts possible = $\frac{^{2n}C_n}{n+1}$
Exactly either one is given.

No. of trees possible with pre: ABC & post: CBA

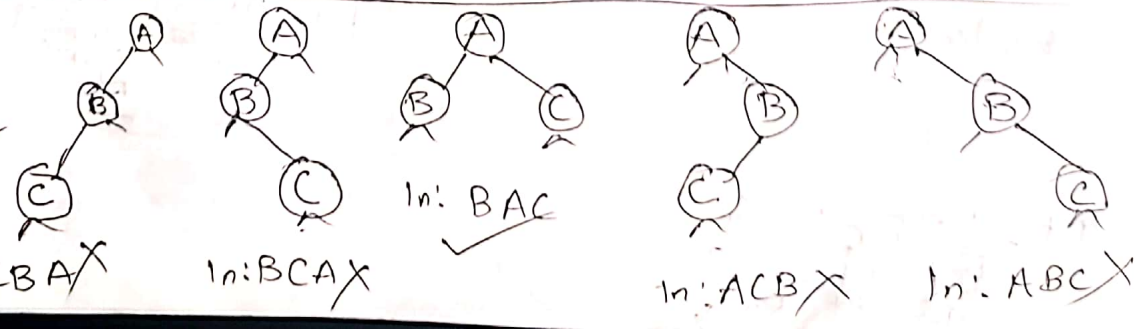
B.T. with Pre: ABC.



$\therefore$ 4 are possible.

No. of B.T. with Pre: ABC, In: BAC

$\therefore \approx 1$



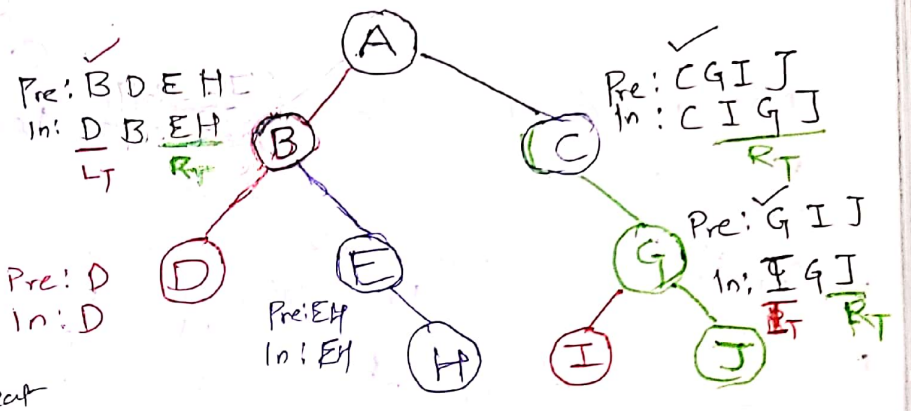
* No. of binary trees with given pre-order & post-order ⇒ many
* No. " " " pre-order & In-order ⇒ 1
* " " " post-order & In-order ⇒ 1

With a given pre-order/post & In-order, no. of B.T.s = Atmost 1

Pre: A B D E H C G I J
In: D B E A C I G J

Short trick:-
(i) Traverse in pre-order →
(ii) Mark root in Inorder, (iii) Repeat

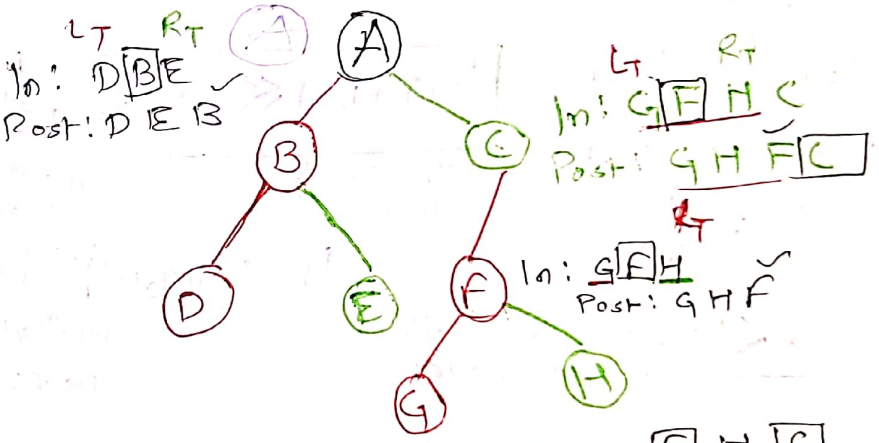
Pre: A B D E H C G I J
In:



In: D B E A G F H C
Post: D E B G H F C A

Short-trick :-
1) Traverse in post-order by opp. direction ←
2) Keep marking the nodes in In-order & keep dividing LT & RT till leaf/last node.

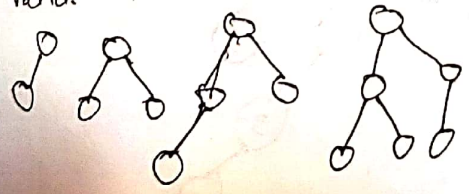
∴ Post: D E B G H F C A



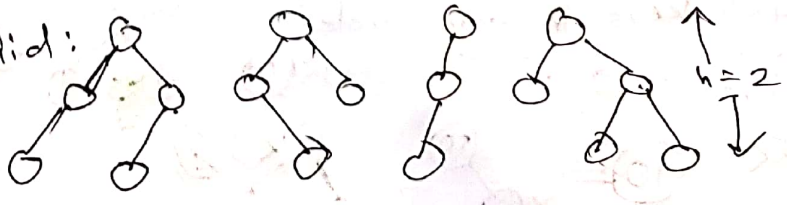
Complete Binary Tree

A CBT is a binary tree which is full upto second last level and nodes at last level are filled from left to right.

Valid:



Invalid:

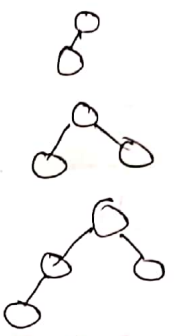


1> Structure of a CBT with 1 node ①

2> — " — 2 "

3> — " — 3 "

4> — " — 4 "



max no. of nodes in a CBT of height $h = 2^{h+1} - 1$

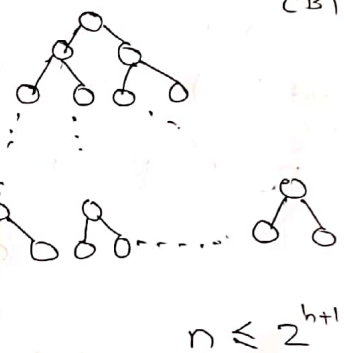
min — " — $= h+1$ ✗

$n = (1 + 2^1 + 2^2 + \ldots + 2^{h-1}) + 1$ (G.P.)

$n = \frac{2^h - 1}{2-1} + 1 = 2^h - 1 + 1 = 2^h$

$\therefore \boxed{2^h \leq n \leq 2^{h+1} - 1}$

# nodes	level
1	0
2^1	1
2^2	2
2^{h-2}	h-2
2^{h-1}	h-1
1	h



min^m no. of nodes in CBT of ht h

$2^h \leq n$

$\log 2^h \leq \log n$

$h \log 2 \leq \log n$

$h \leq \frac{\log n}{\log 2}$

$\boxed{h \leq \log_2 n}$ max^m # of nodes

$n \leq 2^{h+1} - 1$

$n+1 \leq 2^{h+1}$

$\log(n+1) \leq (h+1)\log 2$

$\frac{\log(n+1)}{\log 2} \leq h+1$

$h+1 \geqslant \log_2(n+1)$

$\boxed{h \geqslant \log_2(n+1) - 1}$

$\boxed{\log(n+1) - 1 \leqslant h \leqslant \log_2 n}$

Min^m No. of nodes — max^m Nodes

$\boxed{h = O(\log_2 n)}$ Time complexity

min^m # of nodes

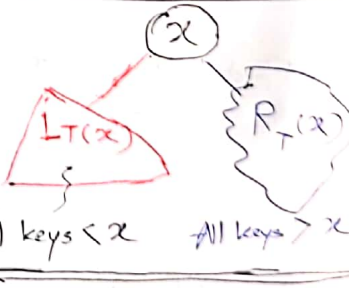
Binary Search Trees:

A BST is a B.T. that satisfies the property:

All the keys in the L_T of a node(x) are smaller than x.

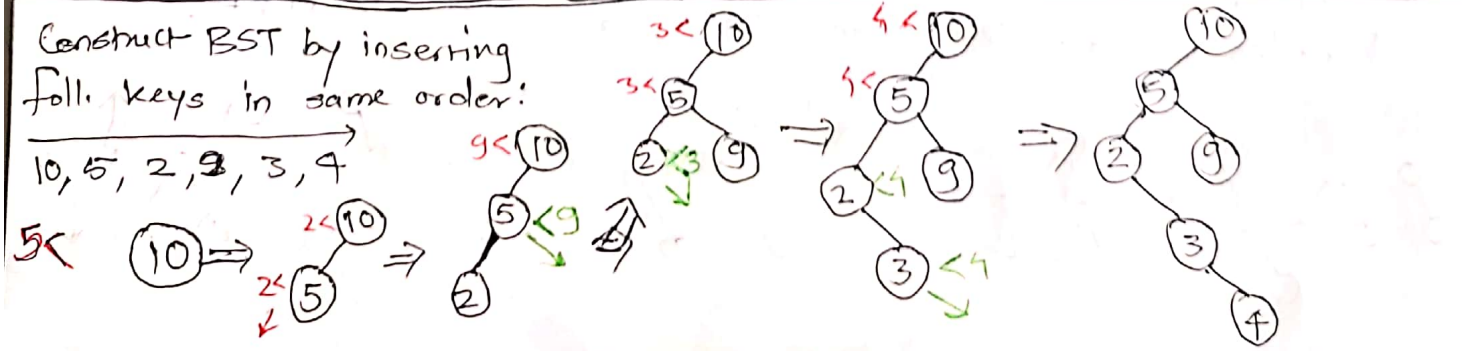
— " — R_T — " — greater — " —.

(For every node in BST)



Construct BST by inserting foll. keys in same order:

10, 5, 2, 9, 3, 4



No. of BSTs, when insertion order of keys is given = 1.

Q. Construct BST by inserting keys 10, 20, 30.

Insertion order can be

10, 20, 30 | 20, 10, 30 | 30, 10, 20
10, 30, 20 | 20, 30, 10 | 30, 20, 10

∴ 3 keys ⇒ 5 BSTs

> No. of BSTs with n keys = $\frac{2nC_n}{n+1}$

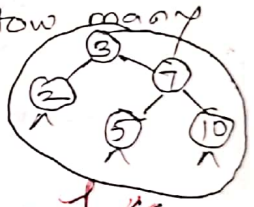
* In order traversal of a BST is always increasing order of keys. i.e. Given the keys of a BST, Inorder traversal can be i.e. In order is fixed for any BST.

* Given a pre-order traversal of a BST, we can find the Inorder as it is fixed for any BST (i.e. in increasing order). Thus, we know with a given a pre-order & Inorder, we can construct BST with short trick. ∴ post-order traversal can be found too.

Q. Given a BT structure with n nodes & n keys, How many BSTs are possible? 10, 2, 5, 7, 3

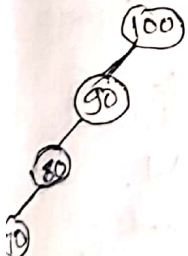
No. of BSTs with n keys = $\frac{2nC_n}{n+1}$,

No. of " " & a structure = 1



---

* Search in a BST

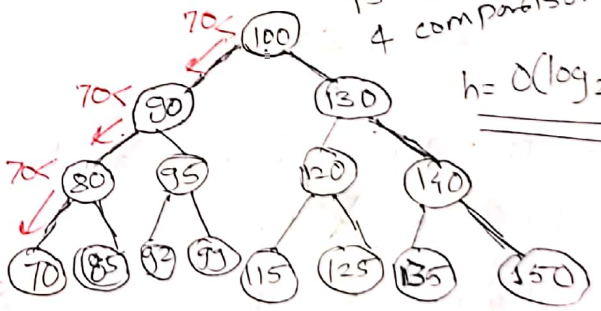


O(n) { 4 elements, 4 comparisons } 4 comparisons needed

Skewed tree

h = n − 1 (max^m height) O(n)

search 70



15 elements, 4 comparisons

h = O(log₂n)

70 = =

∴ No. of comp = h + 1 = O(h) h = O(log₂n)

∴ Time complexity of searching a node in BST is O(n) in the worst case assuming the tree is skewed (i.e. Unbalanced / skewed on only 1 side)

Otherwise, T.C. is O(log₂n) Min^m height of BST = log₂(n+1)

For n nodes, total searchable size will continue to reduce

n : n/2⁰
n/2 : n/2¹
n/4 : n/2²
n/8 : n/2³
⋮
1

∴ $\frac{n}{2^k} = 1$, k = 0, 1, 2 ⋯

∴ $n = 2^k \Rightarrow \log n = k \log_2 2$

∴ $k = \log_2 n$

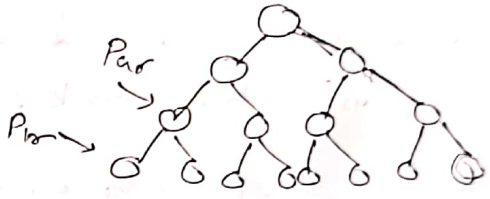
If we have, 8 elements, $k = \log_2 2^3 = 3$, 3 iterns/ comparisons needed.

Deletion

Case I : Deletion of a node having 0-child (leaf node)

" II : —— " —— 1 "

" III : —— " —— 2 "



Assign two pointers Parent & ptr

Case I: We need to identify the parent pointer (of node to be deleted) which is pointing to node to be deleted

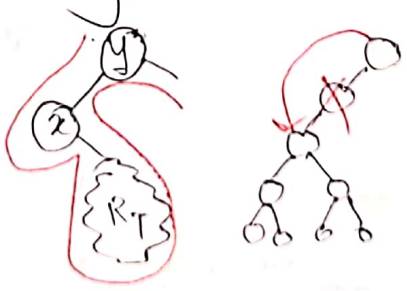
⟹ Make this pointer NULL

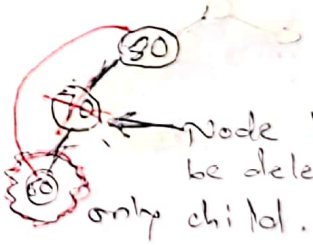
```
if (Ptr -> data < Par -> data)
  { par -> left = NULL
    free(ptr); }
else {
  par -> right = NULL;
   free(ptr);
}
```

Case II: Deletion of a node having one child.

In order: X, R_T, y

Deletion: R_T, y

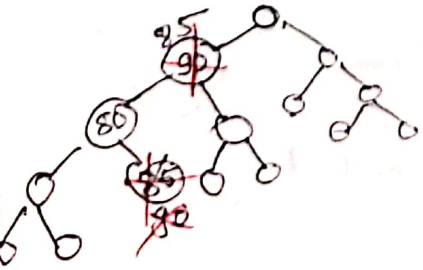




It could be a node with left or right any child.

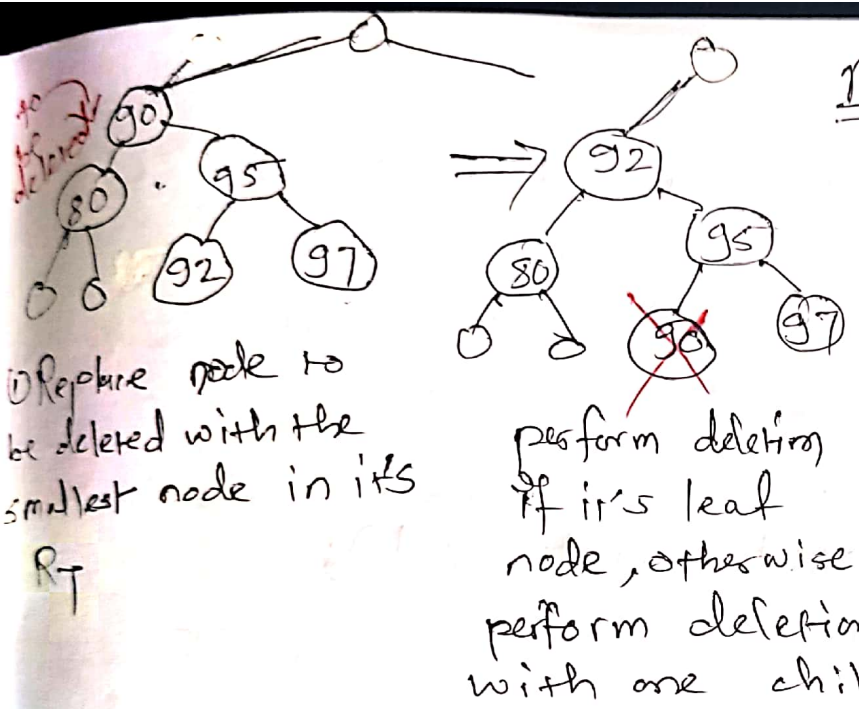
Identify the parent ptr (of node to be deleted) which points to node to be deleted, Make this pointer point to the child of node to be deleted.

Case III: Deletion of node with two children.



Deletion of 90.

Method 1: Replace the node to be deleted with the largest node in the left subtree of the node & perform deletion ensuring no element is larger in the left subtree of newly updated node.



① Replace node to be deleted with the smallest node in its RT

perform deletion if it's leaf node, otherwise perform deletion with one child.

Method 2: Replace the node to be deleted with the smallest node in the right subtree of the node & perform deletion ensuring no element/node in the right subtree of the newly updated node is smaller than the newly updated node.

* Every deletion of node with two children can be reduced to deletion with either one or no children

If the node to be replaced has a child, perform deletion with one child. (A node to be replaced in L_T cannot have 2 children as we are replacing the largest in L_T).

(Note: A node to be replaced in the right subtree cannot be having two children as we are replacing the smallest node with the node to be deleted, If it has left node as well, then that becomes the smallest node to be replaced.

AVL - Tree

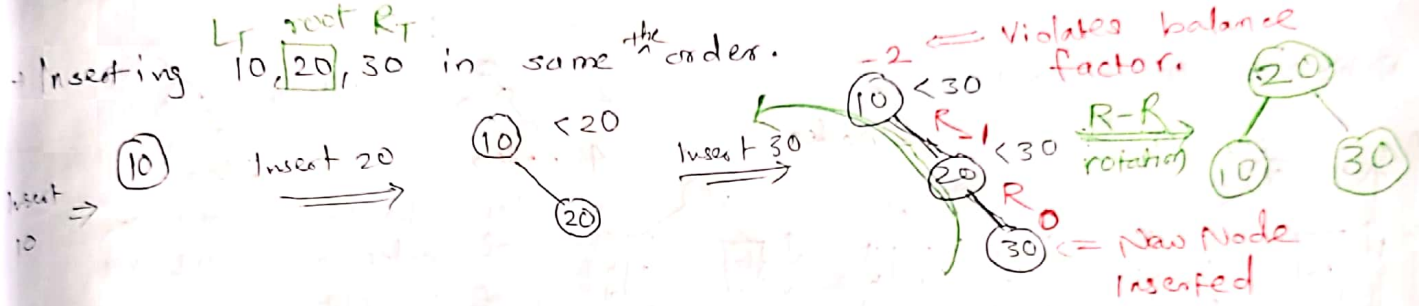
In AVL tree, every node satisfies BST property (keys of L_T < root & all the keys of R_T > root).

* AVL Tree property: The balancing factor of a node can either be 0, 1, -1. Balancing factor of a node is a difference bet^n the height of its left subtree and the height of its right subtree. Whenever the diff/balance is more than one, tree should be balanced using operations called AVL rotations.

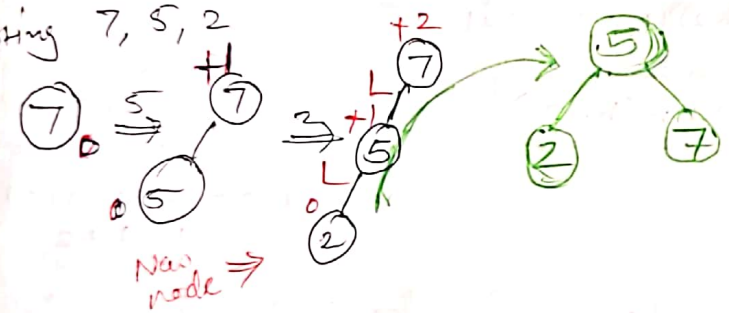
∴ AVL trees are height balancing BSTs.

Insertion of keys may cause the bal. factor of some node other than 0, -1, +1. Tree becomes unbalanced due to this.

- Inserting 10, 20, 30 in some order. (L_T, root, R_T; the ^ order)



Inserting 7, 5, 2



Arranging keys in ascending order 2, 5, 7 ← R_T
L_T ↑ make root

Q. Construct AVL tree by inserting keys in given order.

50, 20, 60, 10, 8, 15, 32, 46, 11, 38

Similarly Insert 32, 46, 11 one by one

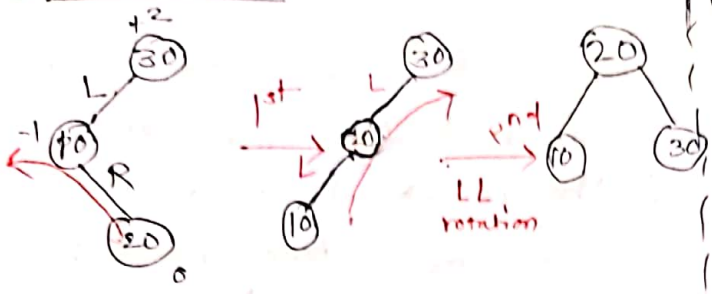
Now, let's Insert 38

32, 38, 46 ⇒

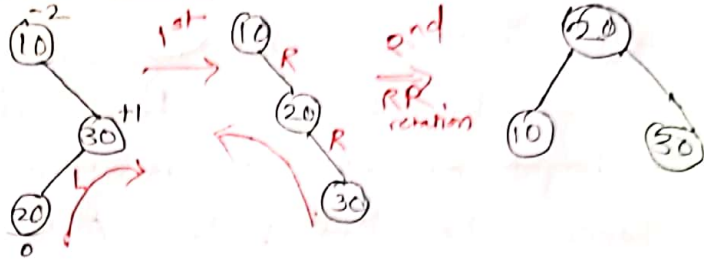
L_T root R_T

30, 10, 20

① LR rotation:



② RL rotation:



LL Rotation, RR —11—] single rotation

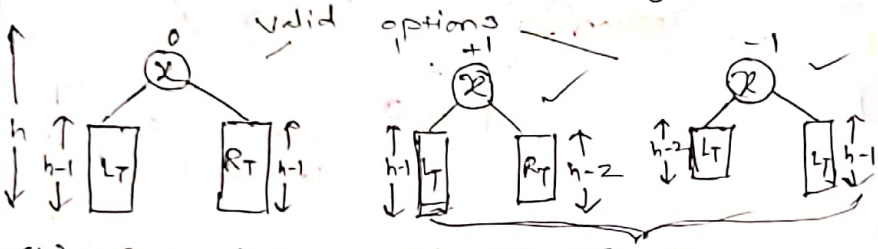
LR Rotation, RL —11—] double rotation.

Maxm no. of nodes in an AVL-tree of height $h = 2^{h+1} - 1$

Minm no. of nodes in —11— :

1> h = 0 ○ 2> h = 1 , 3> h = 2 4> h = 4

Valid options



$n(h) = n(h-1) + n(h-1) + 1$ ✗ $\quad n(h) = n(h-1) + n(h-2) + 1$ ✓

Let n(h) be the min. no. of nodes in an AVL-tree of h height.

$\therefore \boxed{n(h) = 1 + n(h-1) + n(h-2)}$

$n(0) = 1$

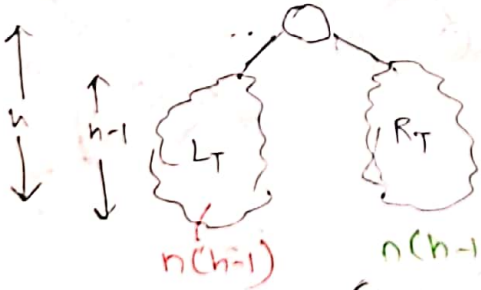
$n(1) = 2$

$n(2) = 1 + n(1) + n(0)$
$= 1 + 2 + 1 = 4$

$n(3) = 1 + n(2) + n(1)$
$= 1 + 4 + 2 = 7$

Q. ∀ B.T. where the diff b/w the no. of nodes in L_T & no. of nodes in R_T is atmost 1 for each node, find the min. no. of nodes in such a tree of height 5.

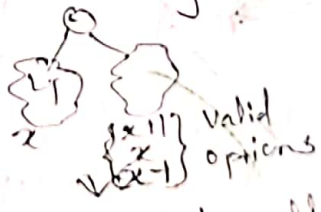
Node Balancing



(As diff. bet n of nodes in L_T & R_T is atmost 1.)

$\therefore n(h) = 1 + n(h-1) + n(h-1) - 1$

$\therefore \boxed{n(h) = 2n(h-1)}$



x-1 would be the best option for minm no. of nodes.

$\therefore n(h) = 1 + n(h-1) + n(h-1) - 1$

$n(0) = 1$ ○

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

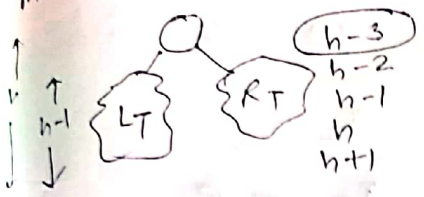
$n(1) = 2 \cdot n(0) = 2^1$

$n(2) = 2 \cdot n(1) = 2^2$

$\vdots$

$n(5) = 2^5 = 32$

Q. for every node: diff of height of L_T & height of R_T is at most 2. min^m no. of nodes in such a B.T. of $h = 4$



← would be the best choice for min nodes as R_T of height "h-3" would contain least nodes.

$$\therefore \quad n(h) = 1 + n(h-1) + n(h-3)$$

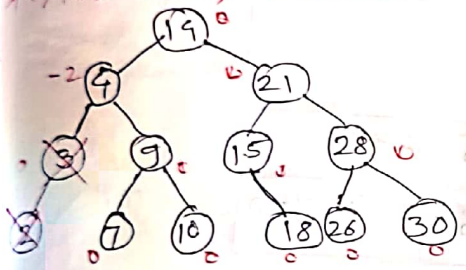
*AVL tree Insertion : i) Insert using BST process.

ii) Update the height from new node to the root i.e. Balance factor. $O(\lg n)$

iii) Check for imbalance, if imbalanced, balance it using rotations. $O(\lg n)$

$O(\lg n)$ Insert → const. number of rotations.

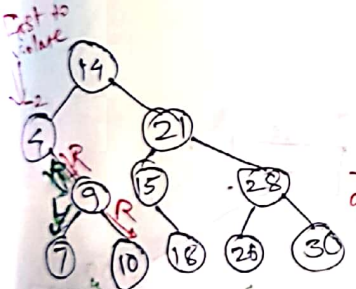
*AVL tree, Deletion of node.



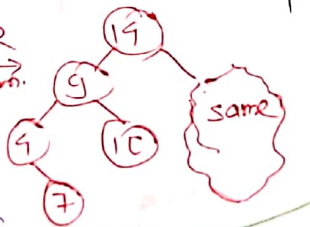
Delete 2. (No. Imbalance)

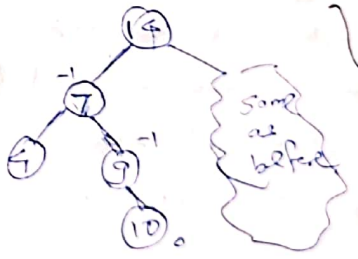
Subsequently perform deletion of 3.

Now, here Balance factor of 4 becomes −2. ∴ 4 is the first node to violate the B.F. (i.e. property of AVL tree)



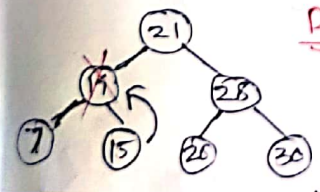
We can either perform RR or RL operation.





Out of these, RR is preferred as it is a single rotation operation.

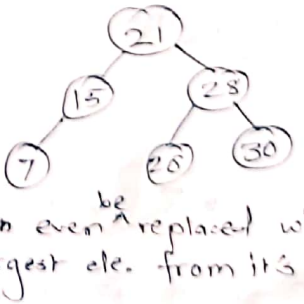
*Deletion of I.N.



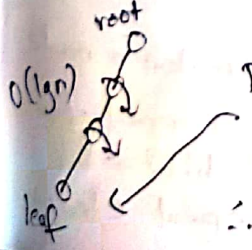
Delete 14

Whenever I.N. is to be deleted, delete it & replace it with it's inorder successor.

We can replace it with the smallest node of the right subtree of the node to be deleted.



→ It can even be replaced with the largest ele. from it's L_T



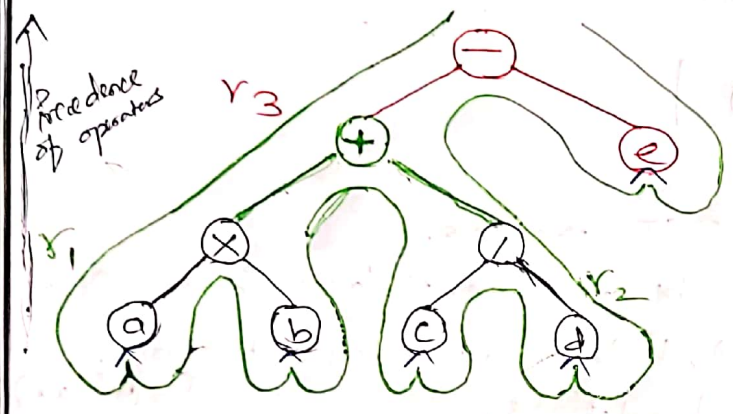
Deletion can propagate it's impact till the root node.

∴ Deletion → $O(\log n)$ no. of rotations in the worst case.

Expression tree

Operands: leaf nodes

Operators: Internal nodes.



Infix: a×b + c/d − e

r_1 r_2

$r_1 + r_2 - e$

$r_3 - e$

The inorder traversal of this tree is infix itself.

Inorder: a×b + c/d − e

Root ⇒ Least priority (last operator)

* Postfix to Expression tree *

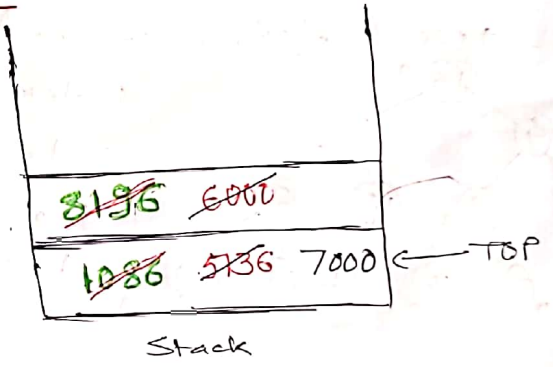
Infix : 4 × 3 + 2

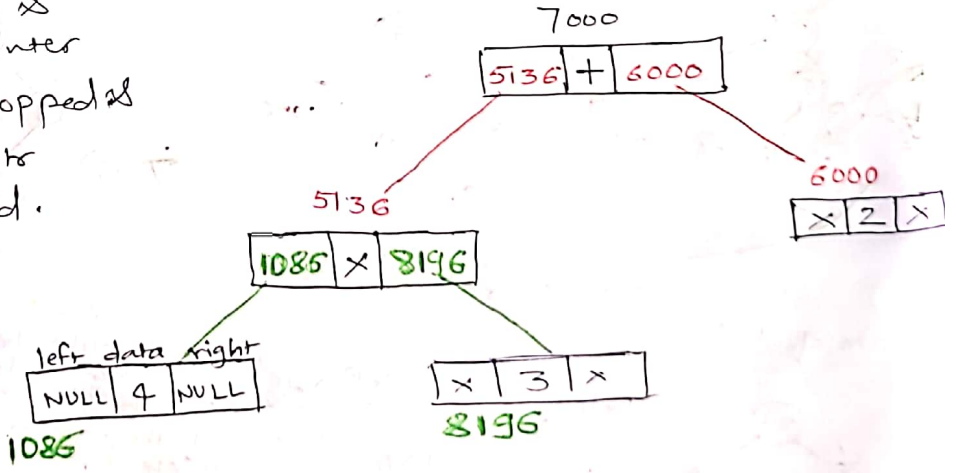
Postfix : 4 3 × 2 + ← End

① Create nodes of postfix operands until an operator encounters.

② When operator is encountered, pop an ele. from stack & make operator's right pointer point to the element popped & pop again, make left ptr point to the ele. popped.

Continue until evaluation of expression.





* Prefix to Expression tree *

Infix: 2 + 3 × 4

Prefix: + 2 × 3 4

Reverse: 4 3 × 2 +

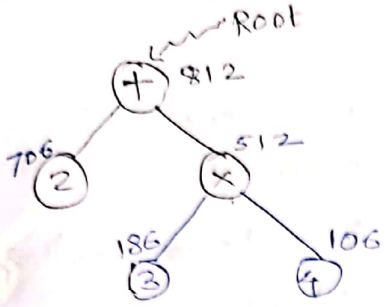
① Create nodes of prefix operands until operator encounters.

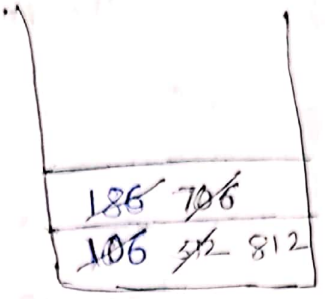
② When operator is encountered, make a node for it first, the first node gets popped from the stack & becomes the left child of the operator node. The 2nd poppeal element

becomes the right child of the operator node.

⑤ The Address of this operator node is then pushed onto stack.

⑥ Continue until evaluation.

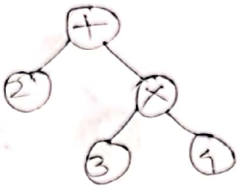




Inorder : Infix

Preorder : Prefix

Postorder: Postfix

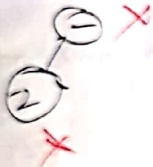


Inorder: 2 + 3 x 4

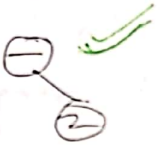
Prefix: + 2 x 3 4

Postfix: 2 3 4 x +

Exp: Unary operator -2



Inorder : 2 −



Inorder: − 2

Priority of Unary Operator > Binary operator

Q. Find postfix of $3 \times \log(x+1)$ GATE QUESTION

log → value / unary operator.

$3 \times \log([x1+])$ — postfix; $[x1+]$ — operand

$3 \times [x1+\log]$

op1 — op2 — operator

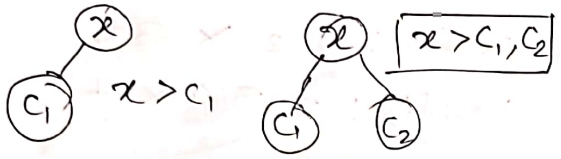
$\therefore$ $3\ x\ 1\ +\ \log\ \times$

Heap

Max Heap

A CBT in which every node satisfies property:

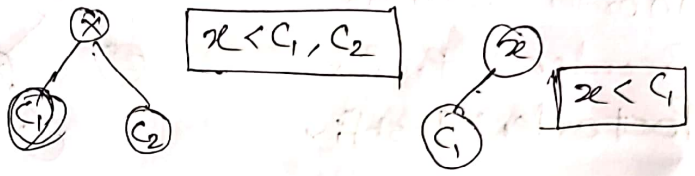
The value of node is greater than its children.



Min-Heap.

A CBT in which every node satisfies the property:

The value of node is smaller than its children.

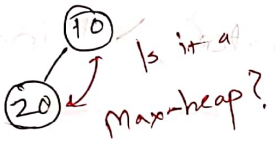


Construction of heap by inserting keys one after another in a given order. $\Rightarrow n \log n$ as Insertion in heap with n nodes $\rightarrow O(\log n)$

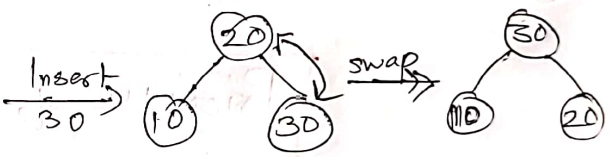
Construct a Max-heap by inserting 10, 20, 30, 40, 50, 60, 70 in same order

Insert 10

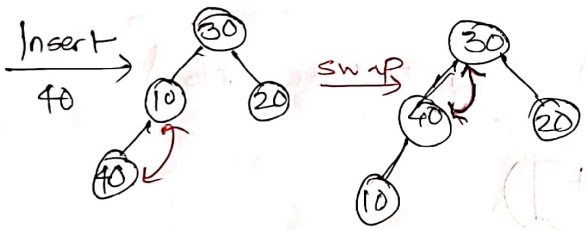
Insert 20



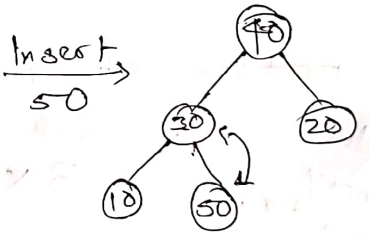
Insert 30

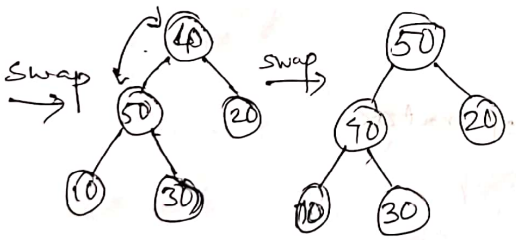


Insert 40

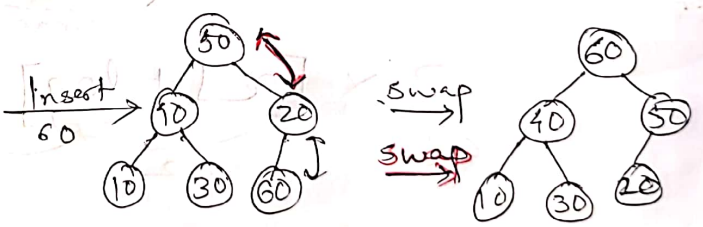


Insert 50

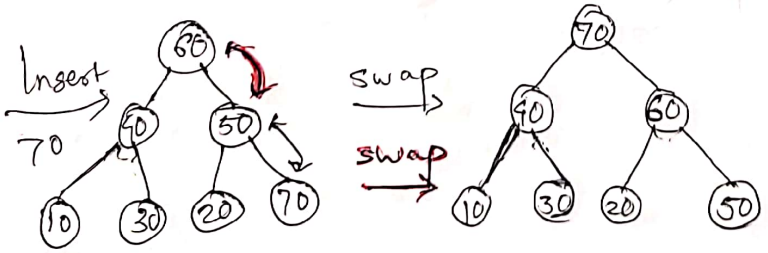




Insert 60

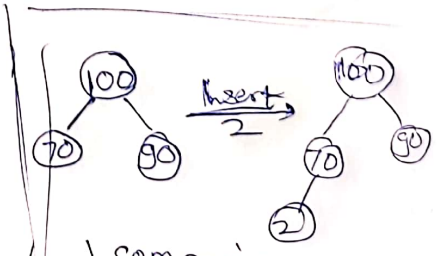


Insert 70

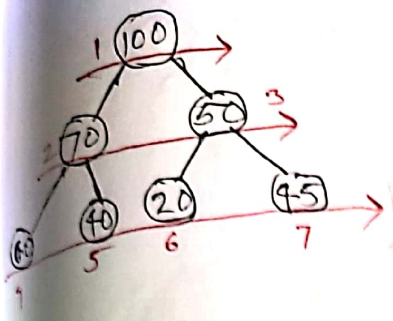


Insertion in heap with n nodes $\longrightarrow O(\log_2 n)$

∴ Construct

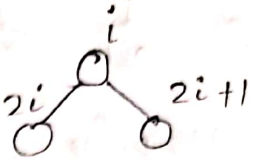


CBT ⟹ Array representation.

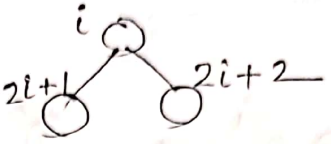


100	70	50	60	40	20	4
1	2	3	4	5	6	7

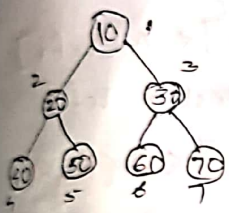
Index of node 100 ⇒ 1
— " — 70 ⇒ 2



very Imp.
Index of node 100 ⇒ 0
— " — 70 ⇒ 1



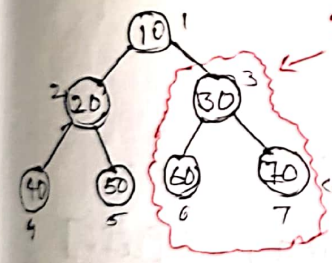
Q. Given an array rep. of a CBT as 10, 20, 30, 40, 50, 60, 70 ? Is it a max-heap ? If No, Convert it to max heap.

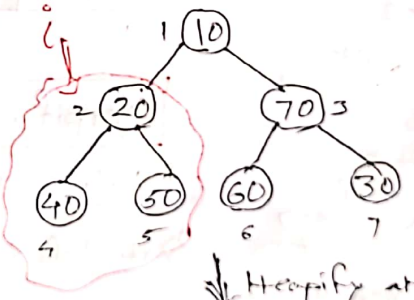


10	20	30	40	50	60	70
1	2	3	4	5	6	7
0	1	2	3	4	5	6 (Practically)

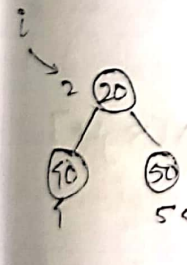
Every leaf node always satisfies max-heap property. → 4, 5, 6, 7 ✓

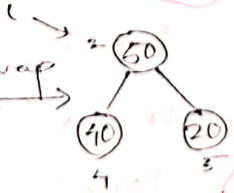
Index of Internal nodes = 1, 2, 3 — 1 to $\lfloor \frac{n}{2} \rfloor$

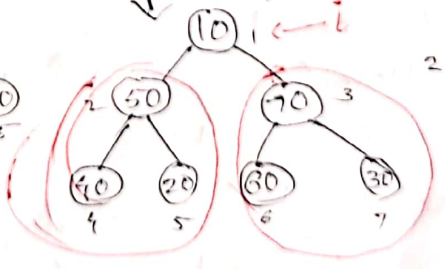




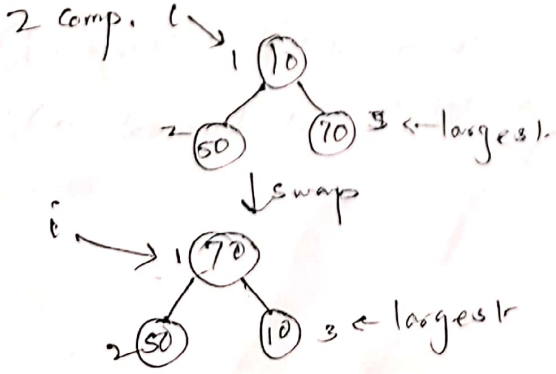
Heapify at index 2

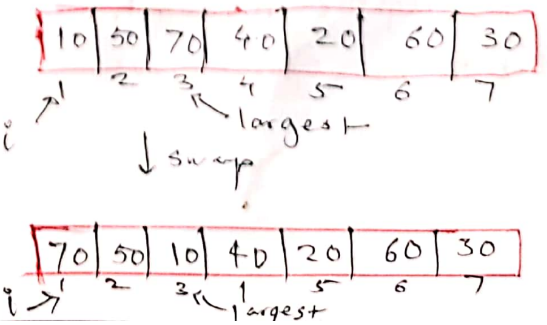


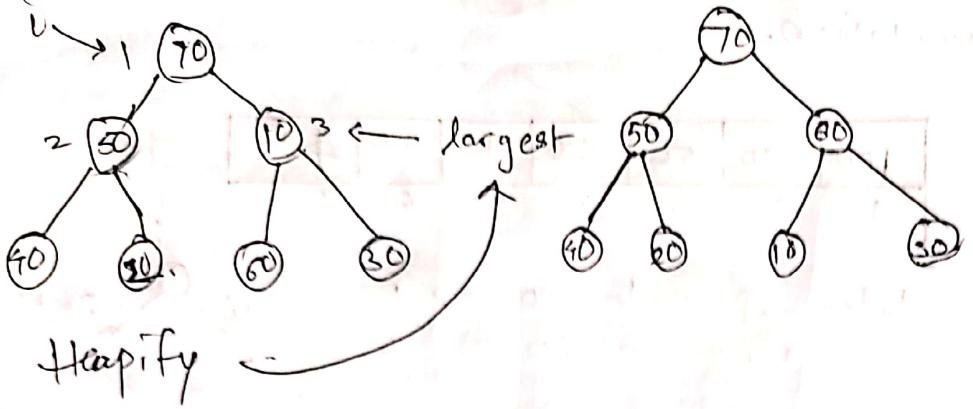




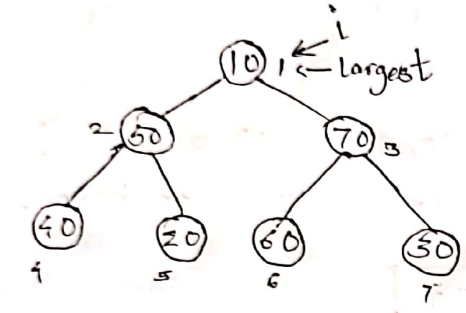
Heapify at 1



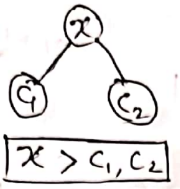




Algorithm:



10	50	70	40	20	60	30
1	2	3	4	5	6	7



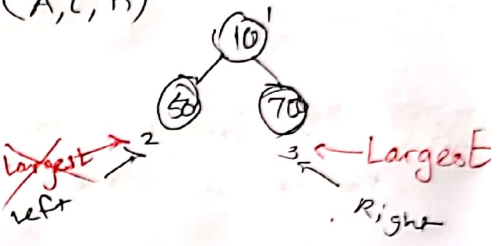
$x > c_1, c_2$

$n = 7$

1) left = 2 * i
Right = 2 * i + 1
Largest = i

17 ← Initially

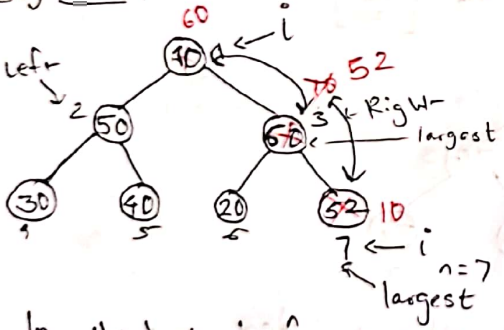
Heapify (A, i, n)



A[largest] < A[Left]
Largest = Left

A[largest] < A[Right]
Largest = Right

Algorithm:



In the last iter^, when the right of 3 i.e. 7 is passed as an i to Heapify, it's left becomes 2i = 2×7 = 14 & right becomes 2i+1 = 15

Now, the condition is checked if Left & right <= n or not. As, they are not control comes out

A	10	50	60	30	40	20	52
	1	2	3	4	5	6	7

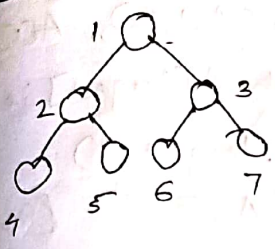
Heapify (A, i, n)

1) Left = 2i; Right = 2i+1; largest = i

2) if Left <= n && A[largest] < A[Left]
Largest = Left;

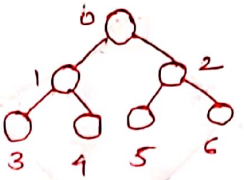
3) if Right <= n && A[largest] < A[Right]
Largest = Right;

4) if (i != Largest)
{
swap (A[i], A[Largest]);
⇒ Heapify (A, Largest, n);
}



Node $\Rightarrow i$ (index)

Parent $\Rightarrow \left\lfloor \frac{i}{2} \right\rfloor$

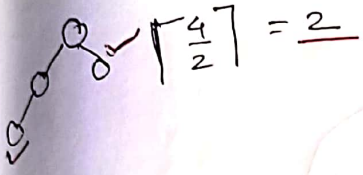


Node $\Rightarrow$ i (index)

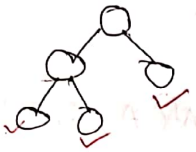
Par $\Rightarrow \left\lfloor \frac{(i-1)}{2} \right\rfloor$

No. of leaf nodes in a heap with n nodes $= \lceil \frac{n}{2} \rceil$

$\lceil \frac{7}{2} \rceil = 4$

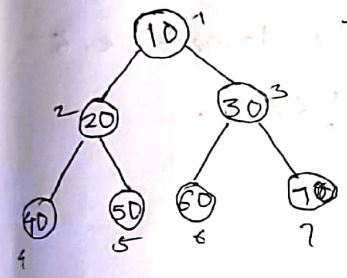


$\lceil \frac{4}{2} \rceil = 2$



No of Nodes = 5 $\therefore \lceil \frac{5}{2} \rceil = \lceil 2.5 \rceil = 3$

Q. Given an array repr. of CBT 10, 20, 30, 40, 50, 60, 70 $\Rightarrow$ Convert to max-heap.



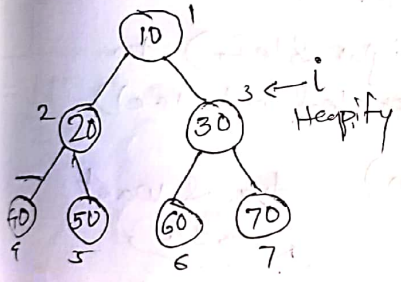
Here, # of leaf nodes $= \lceil \frac{7}{2} \rceil = 4$

$\therefore$ # of Internal Nodes 1 to $\lfloor \frac{n}{2} \rfloor$

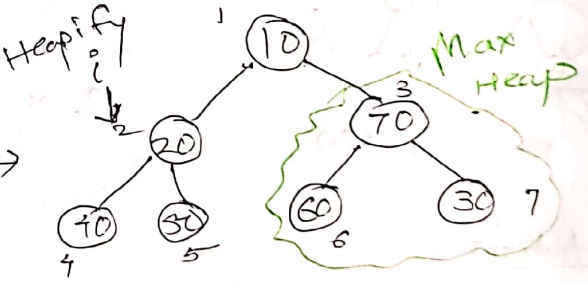
1 to 3

1, 2, 3

```
Build-Heap (A, n)
{
    for (i = ⌊n/2⌋; i >= 1; i--)
        Heapify (A, i, n);
}
```



Heapify (A, 3, 7) $\longrightarrow$

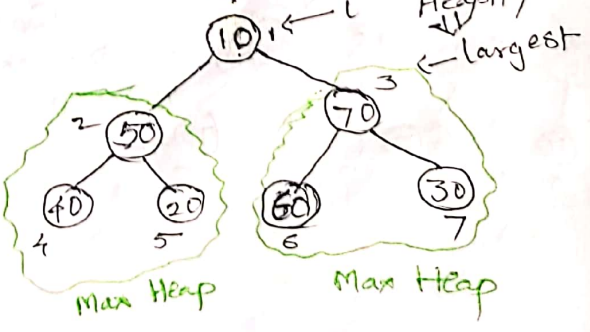


For the 1st iteration of for loop, $i = \lfloor \frac{n}{2} \rfloor$ i.e. 3

Thus, Heapify will be evaluated on index 3 (Heapify (A, 3, 7))

It'll set i i.e. 3 equal to largest & check its right & left for the largest. If i is not equal to largest, it will swap A[i] with A[largest]

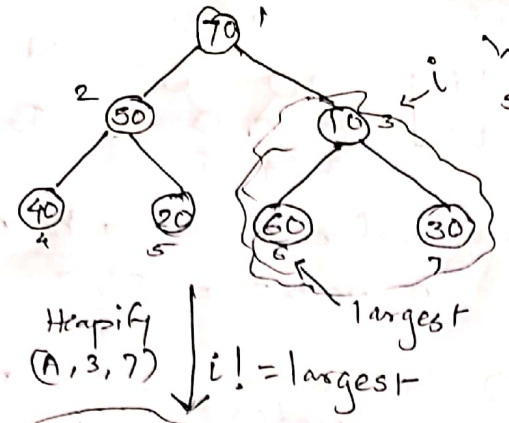
2nd iteration of for loop, i = 2 $\Rightarrow$ Heapify (A, 2, 7)



i != largest

i) swap
ii) Heapify on largest

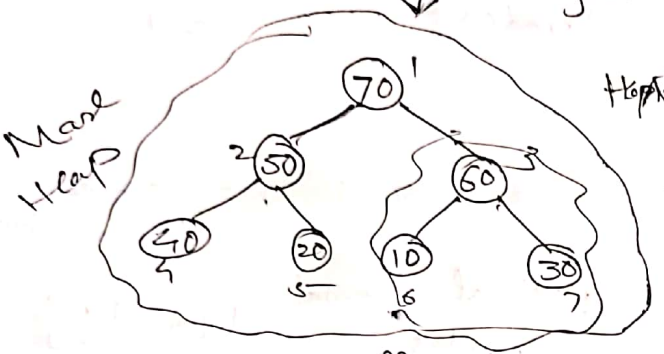
A(A, 1, 7)



While Heapify (A,1,7), A[1] swapped with A[largest] i.e. A[3] & Heapify (A, largest, 7) i.e. Heapify (A, 3, 7) (Recursive call)

Heapify (A, 3, 7)

i != largest



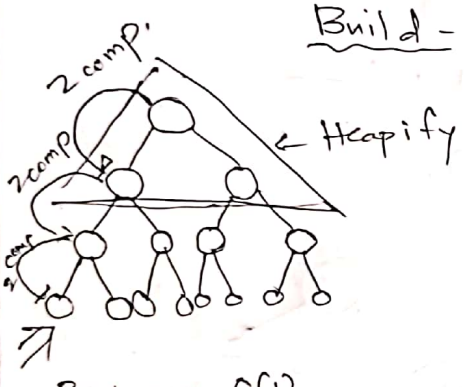
Heapify (A, 6, 7)

i = largest

STOP

Heapify returns as the left & right of the largest i.e. 6 is NULL i.e. ! <= n.

Build - Heap



Best case O(1)

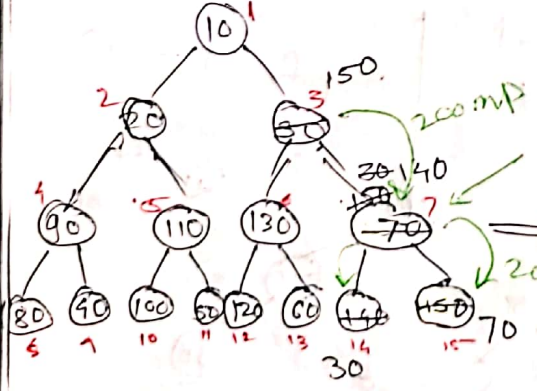
At every level, Heapify func has to do 2 comparisons (i.e. to its right & left) to find out the largest. (except leaf node level)

$2 \times \log_2 n \Rightarrow O(\log_2 n)$

Avg. & worst case.

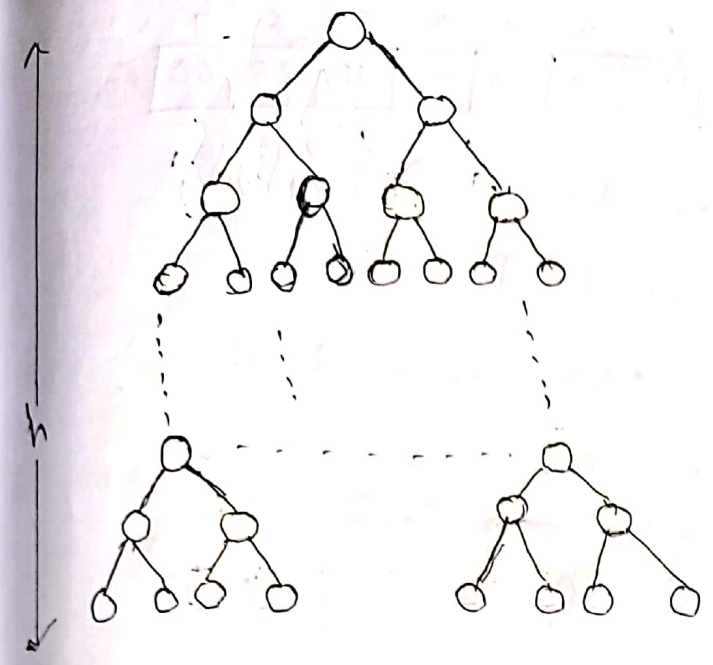
When applied to leaf node.

Analysis:



For every node at this level = 2.2

for every node at this level $\Rightarrow$ 2 comp.



Level	# of comp. for each node	# of nodes at this level
0	$2(h)$	2^0
1	$2(h-1)$	2^1
2	$2(h-2)$	2^2
3	$2(h-3)$	2^3
$h-2$	$2(2)$	2^{h-2}
$h-1$	$2 \cdot 1$	2^{h-1}
h	0	2^h

$$S = 2^0 \times 2(h) + 2^1 \times 2(h-1) + 2^2 \times 2(h-2) + 2^3 \times 2(h-3) + \dots + 2^{h-2} \times 2(2) + 2^{h-1} \times 2(1)$$

$$S = 2\left[2^0(h) + 2^1(h-1) + 2^2(h-2) + 2^3(h-3) + \dots + 2^{h-2}(2) + 2^{h-1}(1)\right]$$

$$\frac{S}{2} = 2^0(h) + 2^1(h-1) + 2^2(h-2) + 2^3(h-3) + \dots + 2^{h-2}(2) + 2^{h-1}(1)$$

$$-S = \quad -2^1(h) \pm 2^2(h-1) \pm 2^2(h-2) \pm \dots \dots \dots \pm 2^{h-1}(2) \pm 2^h(1)$$

$$\frac{-S}{2} = 2^0(h) + 2^1(h-1-h) + 2^2(h-2\underbrace{-h+1}_{-(h-1)}) + 2^3(h-3-h+2) + \dots + 2^{h-1}(1-2) - 2^h$$

$$\frac{-S}{2} = h - 2^1 - 2^2 - 2^3 \dots \dots 2^h$$

$$\frac{-S}{2} = h - \underbrace{(2^1 + 2^2 + \dots 2^h)}_{\text{(G.P.)}}$$

$$\frac{-S}{2} = h - \frac{2^1(2^h - 1)}{2-1} \qquad \frac{a(r^n-1)}{r-1}$$

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

$$\boxed{\frac{S}{2} = 2^{h+1} - 2 - h} \qquad h = \log_2 n$$

We know, Max^m no. of nodes in a tree

$$\boxed{n_{max} = 2^{h+1} - 1}$$

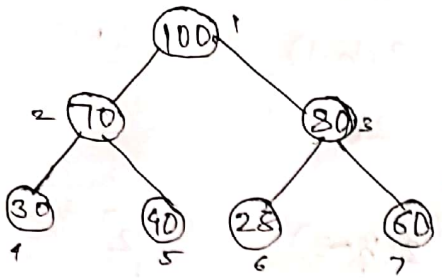
$$\therefore \frac{S}{2} = n_{max} + 1 - 2 - \log_2 n$$

$$\therefore \frac{S}{2} = n - 1 - \log_2 n$$

$$\therefore S = 2n - 2 - 2\log_2 n$$

$$\boxed{S = O(n)}$$

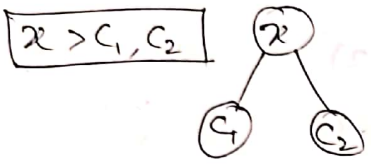
Max-Heap



1	2	3	4	5	6	7
100	70	80	30	40	28	60

```
Find_Max (A, n)
{
    return A[1]
}
```

O(1) constant time.



Minimum can be among these 3 c_1 or c_2

Find_Min
→ Can be some leaf node.
No. of leaf node = $\lceil \frac{n}{2} \rceil$ = O(n)

Max-Heap

Find_Min ⇒ O(n)
Find_Max ⇒ O(1)

comp.
10 ele ⇒ 9 2 ele ⇒ 1
n ele ⇒ n-1
$\lceil \frac{n}{2} \rceil$ ele ⇒ $\lceil \frac{n}{2} \rceil - 1$ ⇒ O(n)

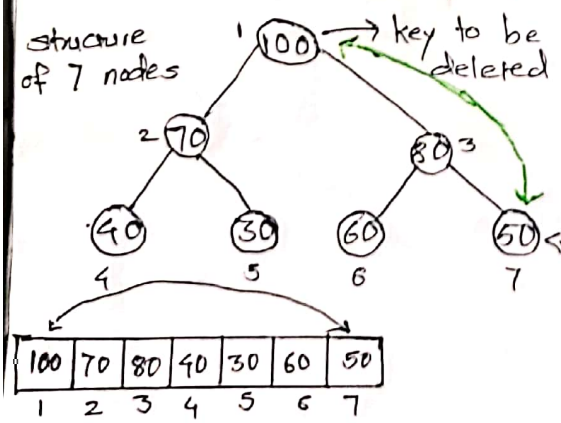
Min Heap

Find_Min ⇒ return A[1]
↳ O(1)
Find_Max ⇒ O(n)

Search in heap : worst case : O(n)
Insert : O($\log_2 n$)

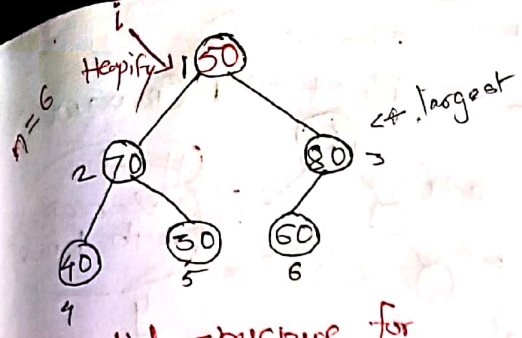
* The structure of a heap with any no. of nodes is fixed, being a CBT

In the structure of n nodes, only the n^{th} node (i.e. last node in the last level among leaf nodes) can be deleted to maintain the property of CBT in order for tree to remain a Heap.



← Only node that can be deleted in a Heap.

That means, before deleting the last i.e. n^{th} node (In this case, 7^{th} node), the key to be deleted must be replaced with the key at node to delete.
That way, the key that was supposed to be deleted will be deleted by getting replaced at node to be deleted.



only valid structure for 6 nodes.

The key to be deleted : A[1]
The node that can be deleted : A[n]

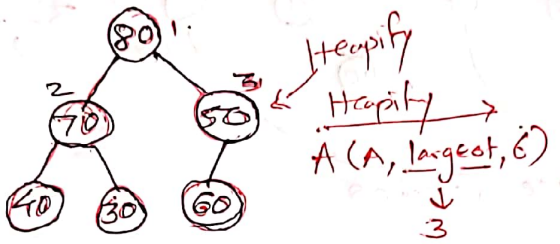
swap keys, A[1] ↔ A[n]

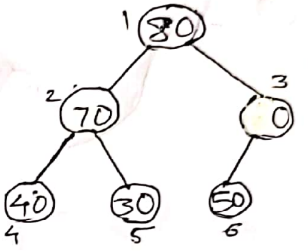
50	70	80	40	30	60	100
1	2	3	4	5	6	7

A[1] ↔ A[n] → const. time
n = n-1 → const. time
Heapify (A, 1, n) → $\log_2 n$

Lastly, Heapify should be implemented on the index at which the key from the node to deleted is present (i.e. 1st index, key-50)
This way, our CBT becomes a Heap again.

∴ Heapify (A, 1, 6)
swap A[1] ↔ A[3]





Max-Heap

Deletion : $O(\log_2 n)$
Insertion : $O(\log_2 n)$
Search : $O(n)$
Find_Max : $O(1)$
Find_Min : $O(n)$
Extract_Max : $O(\log_2 n)$
Extract_Min : $O(1)$

To Extract is to find & delete.
As deletion is performed, Heapify should be implemented.
We know, T.C. of Heapify - log n

Q. How many max-heaps can be possible with 3 distinct keys 10, 20, 30 ?

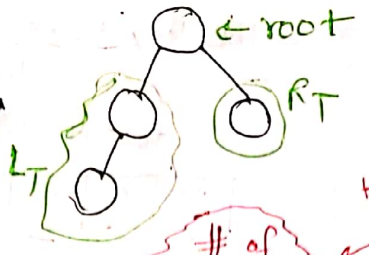
Poss. keys = 10, 20

$^2C_1$

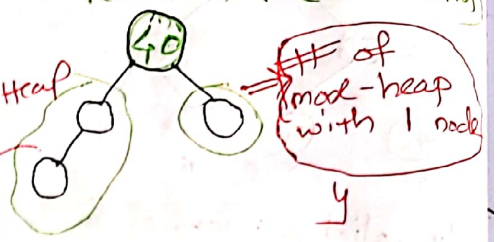
∴ n = 3 → # of Max-Heap → (2)

n = 4 10, 20, 30, 40

4 node structure ⟹



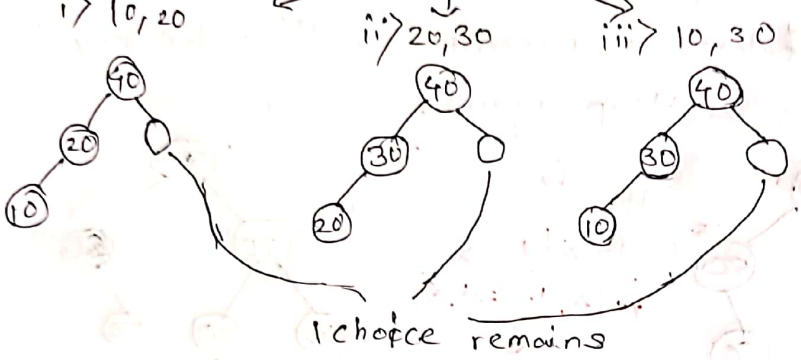
choice for Root ⇒ 1 (maximum)



# of max-heap with 2 nodes
x

Remaining keys 10, 20, 30

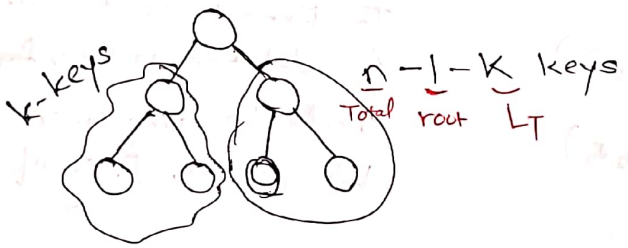
Out of 3 ⇒ select any 2 keys for left sub tree. ${}^3C_2 = 3$



${}^3C_2 \times$ (# of max heaps with 2 dist. keys) $\times$ (# of max heaps with 1 key)

${}^3C_2 \times x \times y$

T(n) : No. of max-heaps with n-distinct keys



$T(n) = 1 \times {}^{n-1}C_k \times T(k) \times T(n-k-1)$

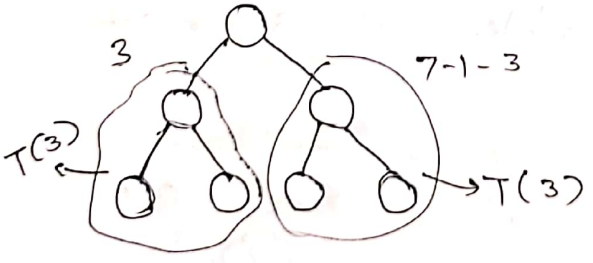
↓ selecting k keys from (n-1) for L_T

E.g. n = 7 (7 distinct keys)

$T(7) = 1 \times {}^6C_3 \times T(3) \times T(3)$

$= 1 \times {}^6C_3 \times 2 \times 2$ (Heaps with 3 dist. keys)

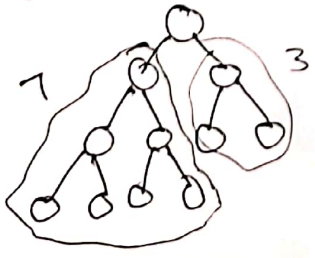
$= \frac{6!}{3!3!} \times 4 = 80$



Eg. n = 11 (11 distinct keys)

$T(11) = 1 \times {}^{10}C_7 \times T(7) \times T(3)$

$\boxed{T(11) = {}^{10}C_7 \times 80 \times 2}$



Q. Which one of the foll. seq. when stored in an array at loc. A[1] to A[20] result a max-heap. GATE - 2023

1) 23, 14, 19, 1, 10, 13, 16, 12, 7, 5

2) 23, 17, 14, 7, 13, 10, 1, 5, 6, 12

3) 23, 17, 10, 6, 13, 14, 1, 5, 9, 12

4) 23, 17, 14, 6, 13, 10, 1, 5, 7, 15

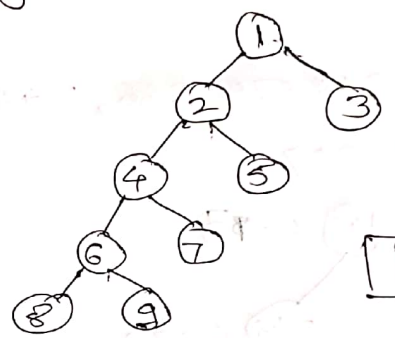
1 2 3 4 5 6 7 8 9 10

Q. The postorder traversal of binary tree is 8, 9, 6, 7, 4, 5, 2, 3, 1. The inorder traversal of the same tree is 8, 6, 9, 4, 7, 2, 5, 1, 3. The height of a tree is the length of the longest path from root to any leaf. What is the height of the above binary tree?

Post: 8, 9, 6, 7, 4, 5, 2, 3, 1

In: 8, 6, 9, 4, 7, 2, 5, 1, 3

LT RT LT RT LT RT Root RT LT



h = 4

Q. We are given a set of n distinct elements & an unlabelled binary tree with n nodes. In how many ways can we populate the tree with given set so that it becomes a BST.

A] 0

B] 1

C] n!

D] $\frac{2nC_n}{n+1}$

No. of BSTs with n keys = $\frac{2nC_n}{n+1}$

Here, the structure is given.

∴ No. of BSTs with n keys/elements & an unlabelled binary tree structure is 1.

Q. A BST stores values in the range 37 to 573. Consider the foll. sequence of keys.

I] 81, 537, 102, 439, 285, 376, 305

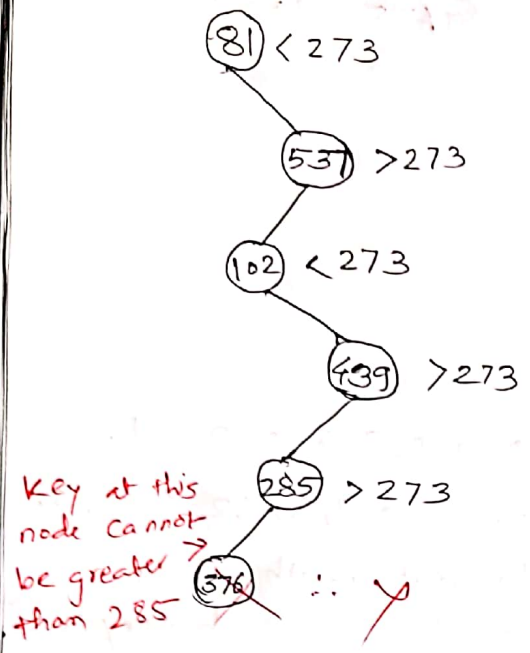
II] 52, 97, 121, 195, 242, 381, 472

III] 142, 248, 520, 386, 345, 270, 307

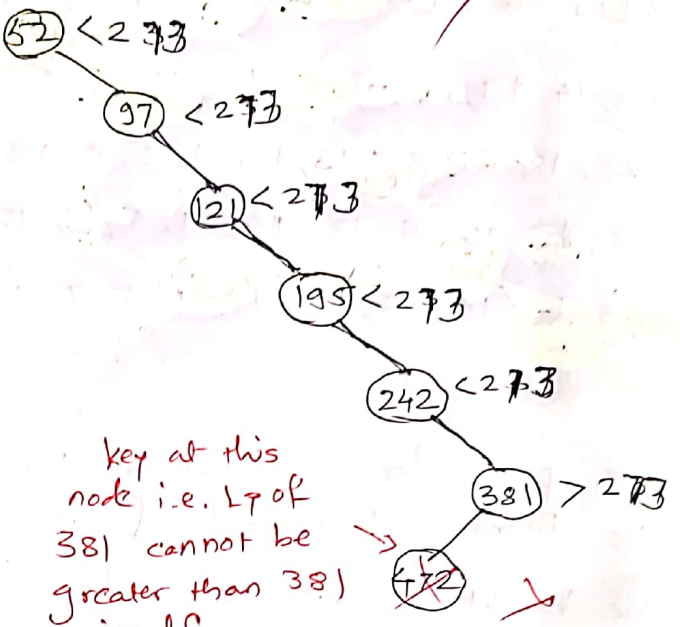
IV] 580, 149, 507, 395, 463, 402, 270

Suppose the BST has been successfully searched for key 273. Which all of above seq. list nodes in the order in which we could have encountered them in search.

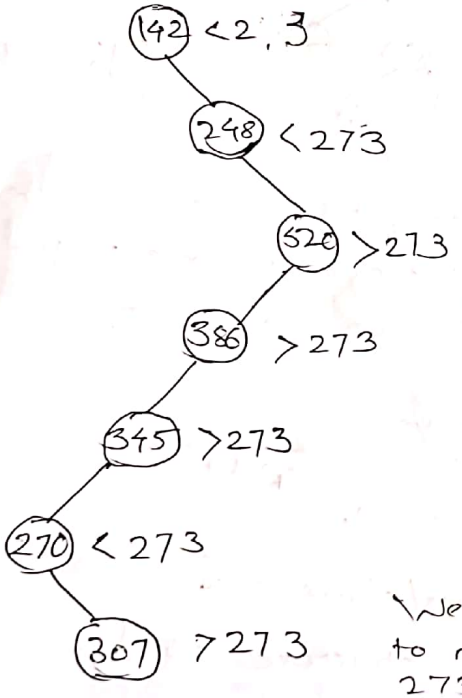
I] 81, 537, 102, 439, 285, 376, 305 ✗



II] 62, 97, 121, 195, 242, 381, 472 ✓

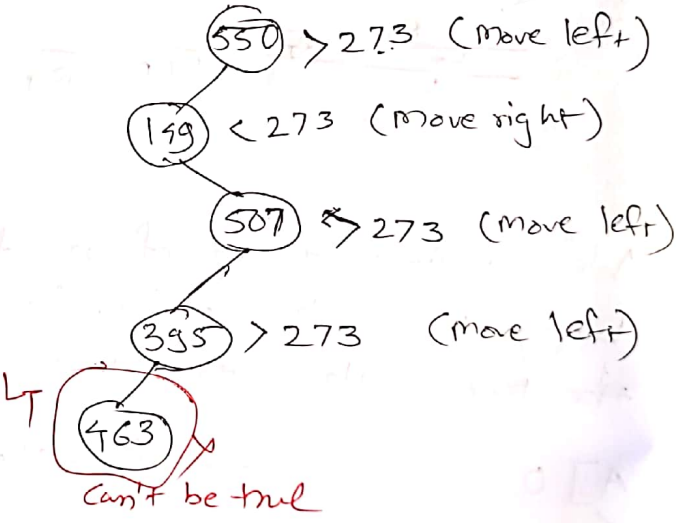


III] 142, 248, 520, 386, 345, 270, 307



We still have a chance to move left of 307 for 273.

IV] 550, 149, 507, 395, 463, 402, 270



Q. When searching for the key 60 in a BST, nodes containing keys 10, 20, 40, 50, 70, 80, 90 are traversed, not necessarily in this given order. How many diff. orders are possible in which these key values can occur on the search path from root to the node containing key 60?

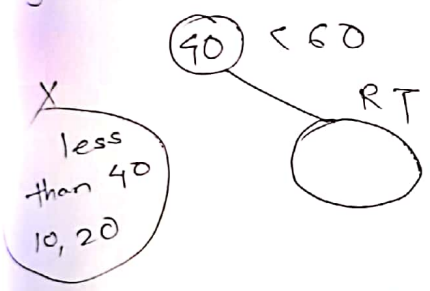
A] 35 ✓ C] 128

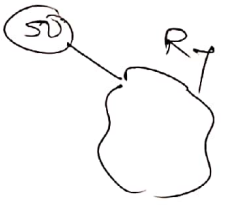
B] 64 D] 5040

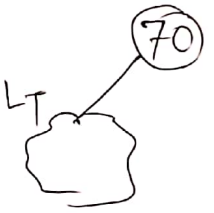
Say, If the first node traversed is 40, then to get to 60, we would not traverse keys less than 40 i.e. 10, 20,

Similarly, If the 1st node traversed is 50, keys < 50 are not traversed.

On the other hand, If the 1st node traversed is 70, keys greater than 70 will not be traversed to get to 60.







80, 90 will never be traversed

10, 20, 40, 50 → 60

these are traversed only in ascending order.

70, 80, 90 ←

traversed in descending order only

90, 80, 70 →

10 90 20 80 70 40 50

90 10 20 80 70 40 50

10 20 90 40 50 80 70

For these 3, 3 places are available & only one choice is valid as it is descending.

10, 20, 40, 50 → 90, 80, 70 →

— — — — — — —

Out of these 7 spaces, Select any 4 places ⟹ 1 choice for placing as it is ascending order

$^7C_4 \times 1$

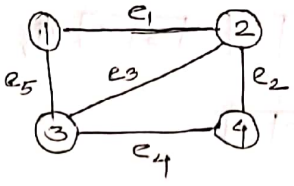
$^7C_4 = 35$

Graph

→ Non-linear data structure.

$G = (V, E)$

- V → set of nodes/vertices
- E → set of Edges



$e_1 = (1, 2)$
$e_2 = (2, 4)$
$e_3 = (2, 3)$
$e_4 = (3, 4)$
$e_5 = (1, 3)$

Order does not matter.
∴ No. of edges = 10

$V = \{1, 2, 3, 4\}$

$E = \{e_1, e_2, e_3, e_4, e_5\}$

$= \{(1,2), (2,4), (2,3), (3,4), (1,3)\}$

Graph Representation

1> Adjacency Matrix 2> Adjacency list.

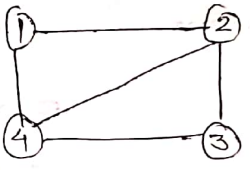
Adjacency Matrix: A n×n square matrix in which every entry is either 0 or 1.

$A_{ij} = 1$ if node i is adjacent to node j.
$= 0$ otherwise

n ⇒ no. of nodes.

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

n = 4
∴ n×n
i.e. 4×4 sq. matrix.



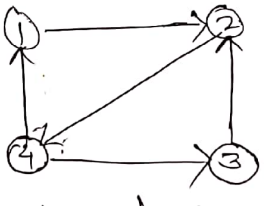
Undirected Graph

No. of entries for Undirected → $2|E|$
No. of edges

Insertion of new node : O(k) or O(v²)

No. of vertices = No. of rows/columns

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



Directed Graph

No. of entries → $|E|$
No. of edges

① Insertion : O(1) For edge
② search : O(1)
③ Deletion : O(1)

2> Adjacency List :

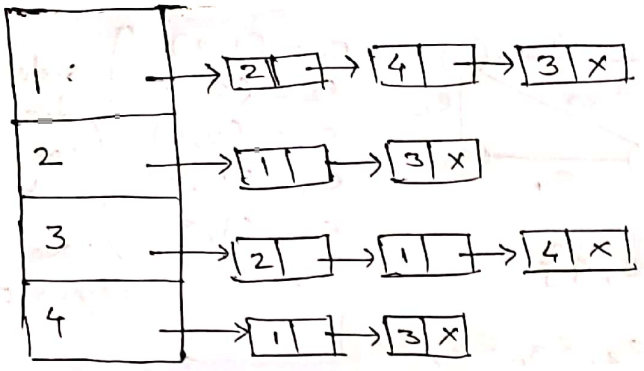
It is an array of linked lists in case of C.

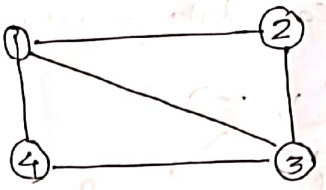
— " — vectors — " — C++.

Size of an array is the number of vertices in Graph.

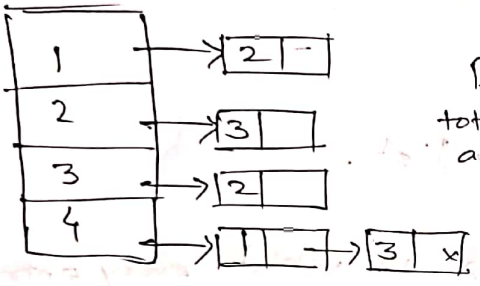
Each list contains neighbours of a node.

Directed Graph



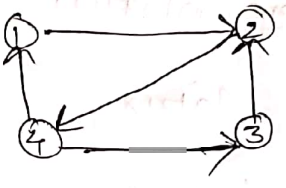


The Order does not matter in both.

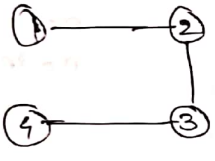


Degree is the total of nodes adjacent in a list.

Undirected Graph



Insertion: O(1)
Search: O(k)
Deletion: O(k)
Add new vertice: O(1)

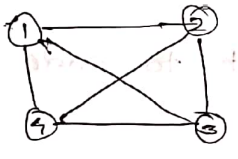


Sparse Graph

$E = |V|$

Space complexity: $O(|V|+|E|)$

Adjacency list used for sparse graph for effective utilization.



Dense Graph

$E \approx |V|^2$

space complexity: $O(V^2)$

Sparse Graph: No. of Edges are close to minimal no. of Edges.

Dense Graph: No. of Edges are close to maximal no. of edges. Every node is connected to each other.

Adjacency matrix is used for dense graph as all the entries will be used by edges.

GRAPH TRAVERSALS

① Depth First Search :

→ Exploring each node first (i.e. visiting it's neighbours) before moving onto next node.

It is a recursive algorithm which traverses a graph in a depthward motion, entails conducting exhaustive searches of all nodes by moving forward if possible and backtracking, if necessary.

This algorithm explores as far as possible along each branch before backtracking.

It uses a stack to remember to get the next vertex to start a search, when a dead end occurs in any iteration.

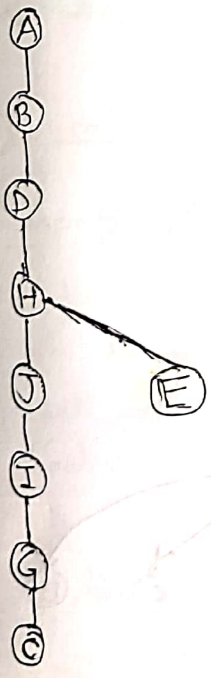
In DFS, We visit the adjacent unvisited vertex, then mark it as visited, display it & push it in a stack.

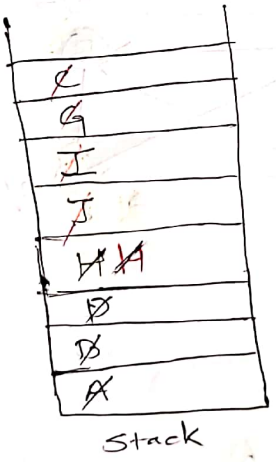
If no adjacent vertex is found, pop a vertex from the stack.

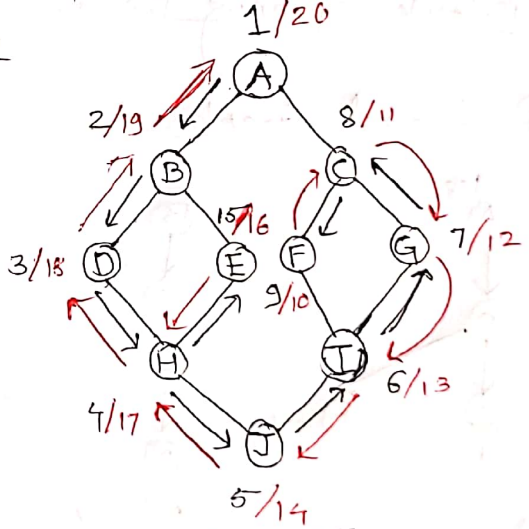
Repeat this until the stack is empty.

Discovery time → First visit time

Finish time → last time visit







1) DFS: A B D H J I G C F E

2)

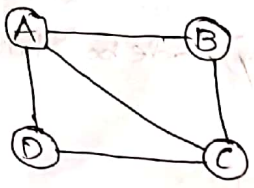
OR

	A	B	C	D	E	F	G	H	J	J
visits	0,1	1	1	1	1	1	1	0,1	1	1

3)

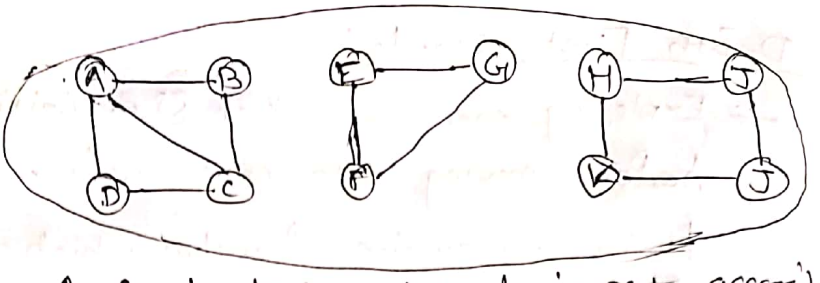
O/p: Spanning tree, In case of DFS of Connected Simple Undirected Graph.

Connected Graph



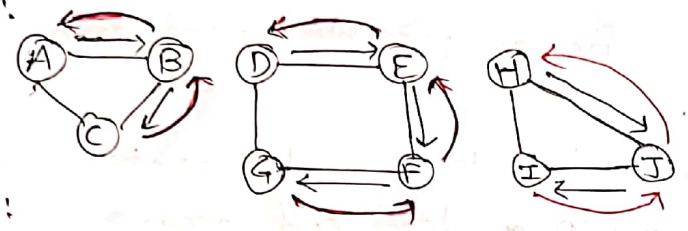
A Graph in which each node is accessible to every other node in the graph.

Disconnected Graph



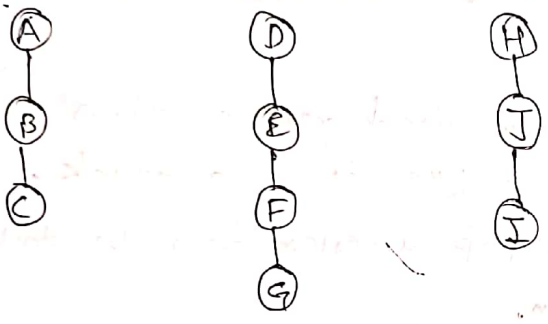
A Graph where each node is not accessible to every other node. There is no edge betⁿ A & E or G & H.

DFS:



	A	B	C	D	E	F	G	H	I	J
vis	1	1	1	1	1	1	1	1	1	1

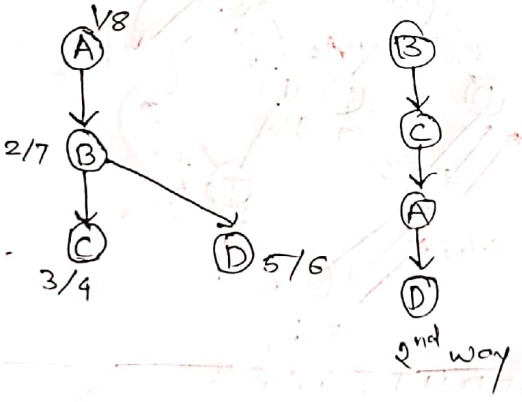
DFS O/P:

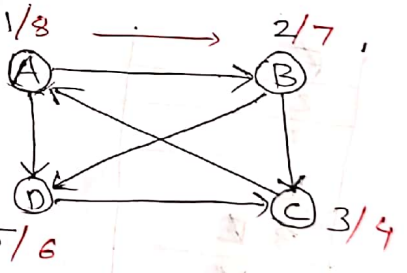


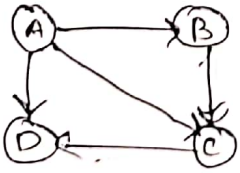
ST Forest

Multiple Spanning Trees

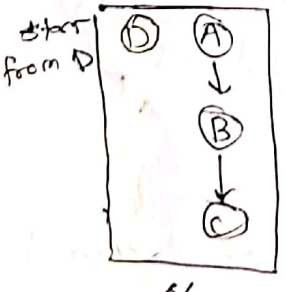
*DFS of directed Graph:



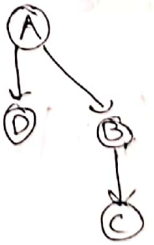




DFS trees possible:



O/P

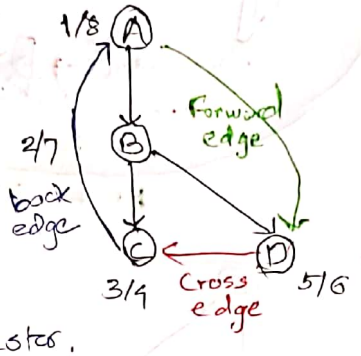


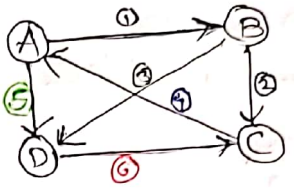
DFS

1. connected simple Graph (Undirected) → ST
2. disconnected simple graph (Undirected) → ST Forest
3. directed → ST Forest

Edges:

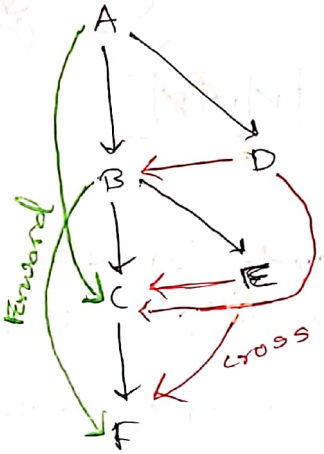
i) Tree Edges:
- A → B
- B → C
- B → D

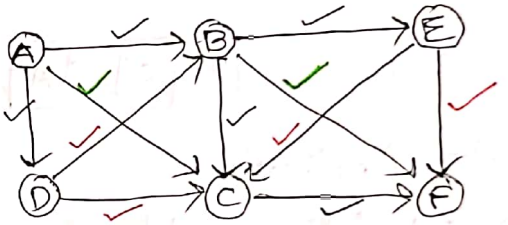




ii) Back edge:
Node to an ancestor.

iii) Forward edge:
Node to non-child descendent

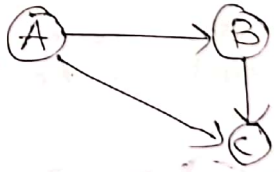




iv) Cross edge:
Edges that do not have any ancestor & a descendant relationship betⁿ them.

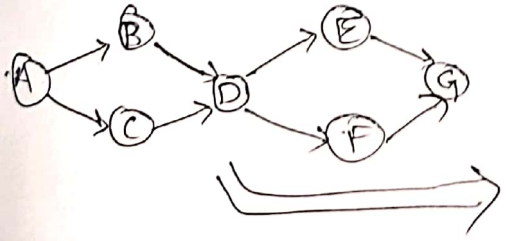
#Applications:

1. Topological sorting: This algorithm takes a directed graph & returns an array of the nodes where each node appears before all the nodes it points to.



A — B — C

> This works only on Directed Acyclic Graph (DAG)



A
- B — C — D — F — E — G
- B — C — D — E — F — G
- C — B — D — E — F — G
- C — B — D — F — E — G

② Breadth First Search (BFS):

The BFS algorithm is used to search a graph D.S. for a node. It starts at the root of the graph or any source node and visits all nodes at the current depth level before moving on to the nodes at the next depth level. BFS algorithm visits every neighbour or adjacent node first before exploring any one of them till dead end, unlike DFS.

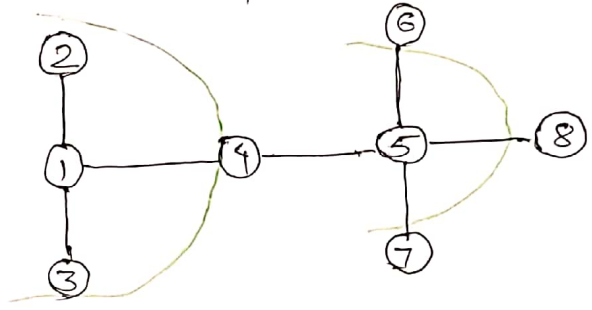
It uses a queue to remember to get the next vertex to start a search, when a dead end occurs in any iteration or when all the neighbour nodes are successfully visited.

Queue

1	2	3	4	5	6	7	8

visited

1	2	3	4	5	6	7	8
✓	✓	✓	✓				



Bfs: 1, 2, 3, 4, 5, 6, 7, 8.

1, 2, 3, 4 — level 1 from source node

5 — 2

6, 7, 8 — At level 3 from source node

Other valid BFS:

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

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

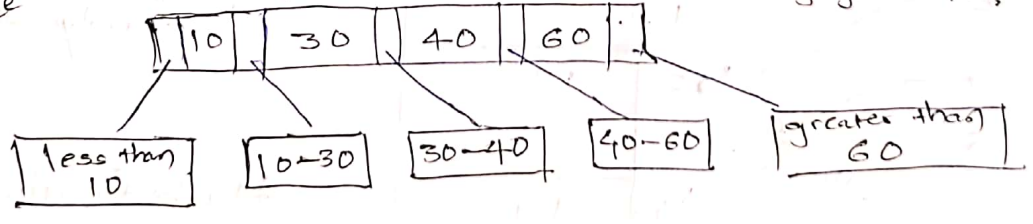
1, 3, 2, 4, 5, 6, 7, 8.

```
BFS (u)
{
  1. Mark u as visited & Enqueue (u)
  2. While (q is not empty)
       x = Dequeue()
       Find all unvisited neighbors of x, mark them
       visited & Enqueue them.
}
```

- Hashing

- Say, we want to store an element in a Data structure that has $n = 2^{20}$ elements i.e. $2^{10} \times 2^{10} = 1024 \times 1024 = 10^6$.
No. of comp req^d here is 2^{20} i.e. n. $\therefore O(n)$
- In BST, this # comp reduced down to $O(\log_2 n)$.
In the worst case, it is still $O(n)$.
- Then AVL tree is introduced which is a height balanced search tree. In AVL tree, # of comp is $O(\log_2 n)$
In this case, $\log_2 2^{20} = 20$ comparisons.
- In Database systems, B & B+ trees are used.
As we have seen above in the case of AVL tree, # of comparisons depend on the height of the tree.
Each node had 2 children & thus the height was $\log_2 n$.
- Multiway Search Tree, commonly known as a M-way tree can yield a max^m of m children where $m > 2$. Due to their structure, they've used mainly in external searching i.e. in situations where data is to be retrieved from secondary storage like disk files, minimizing the no. of file accesses. Lesser the height, more efficient is the external search.
- B-tree is a special type of M-way tree which balances itself. Eases searches due to their balanced nature to manage gigantic datasets.

B-tree/
B+ tree

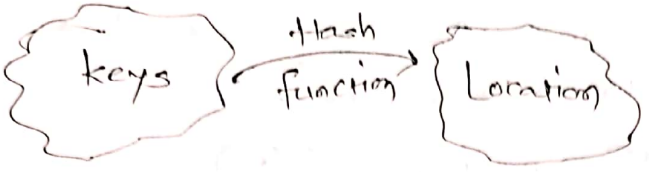


B-tree
$m = 8$
$O(\log_m n) = \log_8 2^{20}$
$= \log_{2^3} 2^{20} = \frac{20}{3} \log_2 2$

Order of a node: Represents the max^m no. of children a tree's node could have.

≈ 7 comparisons.

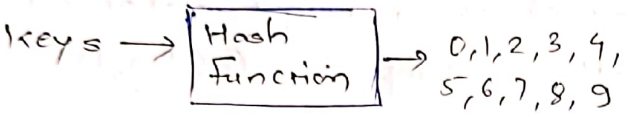
The goal of Hashing is to try to make it O(1)



Hash Function takes keys as inputs & outputs the suitable location for efficient access.

Say $m = 10$

keys: 13, 22, 15, 78, 86, 91, 107



$h(k) = k \bmod m$

$h(13) = 13 \bmod 10 = 3$

$h(22) = 22 \bmod 10 = 2$

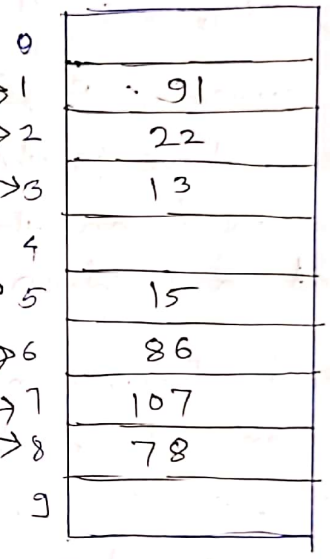
$h(15) = 15 \bmod 10 = 5$

$h(78) = 78 \bmod 10 = 8$

$h(86) = 86 \bmod 10 = 6$

$h(91) = 91 \bmod 10 = 1$

$h(107) = 107 \bmod 10 = 7$



Now, let's assume the keys : 12, 14, 16, 22, 38, 47

$h(k) = k \bmod m$

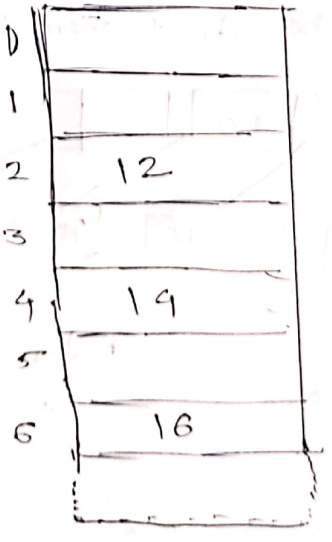
$h(12) = 2$

$h(14) = 4$

$h(16) = 6$

$h(22) = 2$

collision



Good Hash function

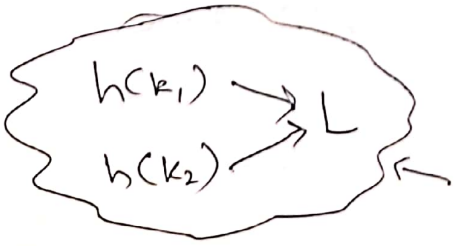
i) Easy to compute.

ii) Uniformly distributed

Hash functions

$h(k) = k \bmod m$
$0, 1, \ldots, m-1$

$h(k) = (k \bmod m) + 1$
$1, \ldots, m$



Collision

To avoid collision, there're collision Resolution Techn

Collision Resolution Techniques.

i) Linear Probing
ii) Quadratic Probing
iii) Double Hashing
iv) Separable chaining

LINEAR PROBING

Let $h(k) = k \bmod m$

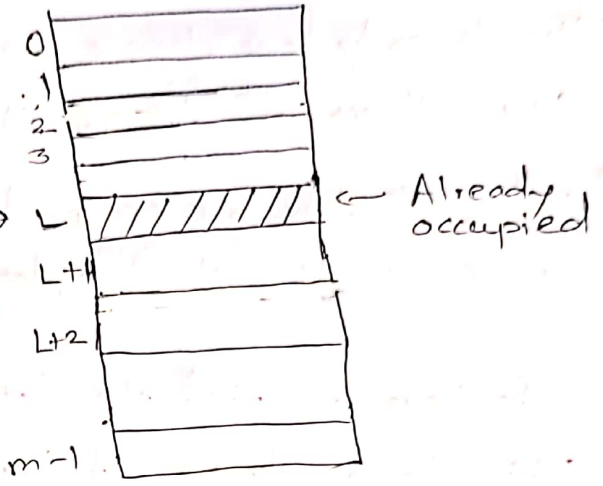
$h(k,i) = L$] collision occur

$H(k,i) = (h(k)+i) \bmod m$

key, collision no.

$H(k,1) = (h(k)+1) \bmod m$

$= (L+1) \bmod m$



e.g.

$h(k) = k \bmod m$

keys: 31, 26, 43, 27, 34, 46, 14, 58, 13

$m = 12$

i) $h(31) = 31 \bmod 12 = 7$

ii) $h(26) = 26 \bmod 12 = 2$

iii) $h(43) = 43 \bmod 12 = 7$ collision

$i = 1$ (1st collision)

$H(k,i) = (h(k)+i) \bmod m$

$H(43,1) = (h(43)+1) \bmod 12$

$= (7+1) \bmod 12 = 8$ ✓

iv) $h(27) = 27 \bmod 12 = 3$

v) $h(34) = 34 \bmod 12 = 10$

vi) $h(46) = 46 \bmod 12 = 10$

$H(46,1) = (h(46)+1) \bmod 12 = 11$ ✓ (1st collision)

vii) $h(14) = 14 \bmod 12 = 2$

$H(14,1) = (h(14)+1) \bmod 12$ (1st collision)

$= 3$ (2nd collision)

$H(14,2) = (h(14)+2) \bmod 12 = 4$ ✓

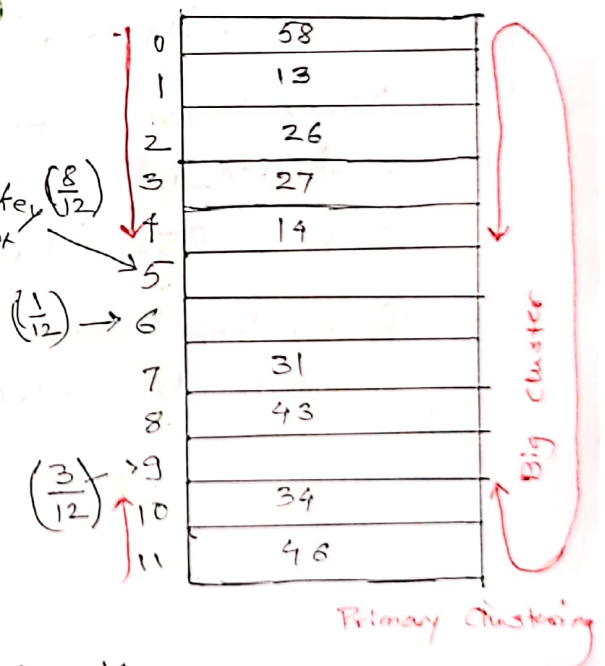
viii) $h(58) = 58 \bmod 12 = 10$ (1st collision)

$H(58,1) = (h(58)+1) \bmod 12 = 11$ (2nd collision)

$H(58,2) = (h(58)+2) \bmod 12 = 0$ ✓

ix) $h(13) = 13 \bmod 12 = 1$

Prob. that new key will get this slot: $\frac{8}{12}$, $\frac{1}{12}$, $\frac{3}{12}$



∴ Probability that new key will get 5^{th} slot is $\left(\frac{8}{12}\right)$ as out of 12, 7 slots are occupied before 5^{th} slot, 7 slots + 5^{th} slot = $\frac{8}{12}$, as the key will get 5^{th} slot if it gets location of any of these 7 slots or 5^{th} slot.

There's only one chance out of 12 outcomes that the new key will get 6^{th} slot i.e. when it is assigned 6^{th} slot.

7^{th} & 8^{th} slots are occupied. Thus, probability of getting a new node at 9^{th} slot is $2+1 = \left(\frac{3}{12}\right)$.

∴ The keys occupied the slots from 10 to 4. (10, 11, 0, 1, 2, 3, 4)
This is a cluster of length 7.

The probability of getting a new key at 5^{th} slot is the highest. Therefore, the cluster can get bigger.
This results in a Primary Clustering problem.

Many keys occupied successive slots/buckets forming a cluster and consequently, it will consume time to find a free slot or to search for an element.
If any key hashes into the cluster will require several attempts to resolve the collision & then it will get added.

QUADRATIC PROBING

$h(k) = k \bmod m = L_1$ } collision

$H(k,i) = (h(k) + i^2) \bmod m$

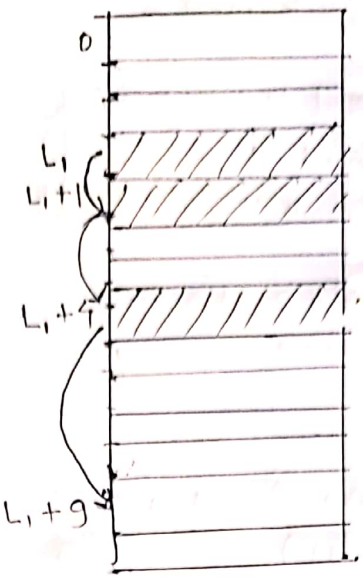
$H(k,1) = (h(k) + 1^2) \bmod m = L_1 + 1$

$H(k,2) = (h(k) + 2^2) \bmod m = L_1 + 4$

$H(k,3) = (h(k) + 3^2) \bmod m$

$= L_1 + 9$ ✓

Free from primary clustering problem



eg: 24, 17, 32, 2, 13, 50, 30, 61

$m = 11$

$h(k) = k \bmod m$

i) $h(24) = 24 \bmod 11 = 2$

ii) $h(17) = 17 \bmod 11 = 6$

iii) $h(32) = 32 \bmod 11 = 10$

iv) $h(2) = 2 \bmod 11 = 2$ collision

$H(2,1) = (h(2) + 1^2) \bmod 11$

$= 3 \bmod 11 = 3$ ✓

v) $h(13) = 13 \bmod 11 = 2$ 1st collision

$H(13,1) = (h(13) + 1^2) \bmod 11$

$= 3$ 2nd collision

$H(13,2) = (h(13) + 2^2) \bmod 11$

$= 6$ 3rd collision

$H(13,3) = (h(13) + 3^2) \bmod 11$

$= 0$ ✓

vi) $h(50) = 50 \bmod 11 = 6$ collision

$H(50,1) = (h(50) + 1^2) \bmod 11 = 7$ ✓

vii) $h(30) = 30 \bmod 11 = 8$

viii) $h(61) = 61 \bmod 11 = 6$ collision

$H(61,1) = (h(61) + 1^2) \bmod 11 = 7$ 2nd collision

0	13
1	
2	24
3	2
4	61
5	
6	17
7	50
8	30
9	
10	32

$H(61,2) = (h(k) + 2^2) \bmod 11$

$= 10$ 3rd collision

$H(61,3) = (h(k) + 3^2) \bmod 11$

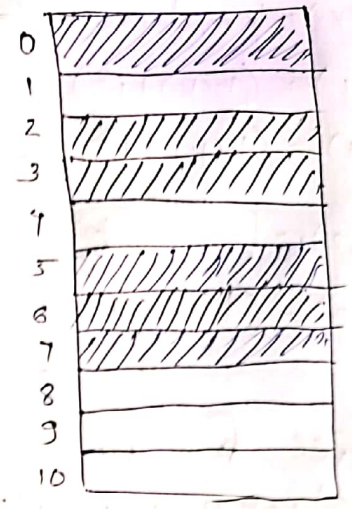
$= 4$ ✓

Secondary Clustering Problem

$m = 11$, $h(k) = k \bmod m$

$h(24) = 24 \bmod 11 = 2$
$h(2) = 2 \bmod 11 = 2$
$h(13) = 13 \bmod 11 = 2$

In this case, 24, 2, 13 are hashed to same memory location.



i = 1

$H(24,1) = (h(24) + 1^2) \bmod 11 = 3$
$H(2,1) = (h(2) + 1^2) \bmod 11 = 3$
$H(13,1) = (h(13) + 1^2) \bmod 11 = 3$

i = 2

$H(24,2) = (h(24) + 2^2) \bmod 11 = 6$
$H(2,2) = (h(2) + 2^2) \bmod 11 = 6$
$H(13,2) = (h(13) + 2^2) \bmod 11 = 6$

i = 3

$H(24,3) = (h(24) + 3^2) \bmod 11 = 0$
$H(24,3) = (h(2) + 3^2) \bmod 11 = 0$
$H(13,3) = (h(13) + 3^2) \bmod 11 = 0$

i = 4

$H(24,4) = (h(24) + 4^2) \bmod 11 = 7$
$H(2,4) = (h(2) + 4^2) \bmod 11 = 7$
$H(13,4) = (h(13) + 4^2) \bmod 11 = 7$

i = 5

$H(24,5) = (h(24) + 5^2) \bmod 11 = 5$
$H(2,5) = (h(2) + 5^2) \bmod 11 = 5$
$H(13,5) = (h(13) + 5^2) \bmod 11 = 5$

2, 3, 6, 0, 7, 5 → same resolution path for all 3 keys

i = 6

$H(24,6) = (h(24) + 6^2) \bmod 11 = 5$
$H(2,6) = 5$
$H(13,6) = 5$

2, 3, 6, 0, 7, 5, 7, 0, 6, 3, 2 → repeat

i = 7 : 7 , i = 8 : 0

∴ We are not effectively utilizing the size of the table despite having free space availability, as keys that are hashed to same memory location always follows same resolution path.

DOUBLE HASHING BCRT

Let $h(k)$ is the hash we are using

$h(k) = k \bmod m \Rightarrow$ collision occurs

$H(k,i) = (h(k) + i) \bmod m$ .....Linear Probing

$H(k,i) = (h(k) + i^2) \bmod m$ .....Quadratic Probing

$H(k,i) = (h(k) + i\, h'(k)) \bmod m$ .... Double Hashing

- $h(k)$ → Primary hash function
- $h'(k)$ → Secondary hash function

Conditions: $h'(k) \neq k \bmod m$

Secondary hash function cannot be same as the primary " " as it will not resolve the collision occured.

$h'(k) = k \bmod m + 1$ ✓ can be accepted.

Secondary hash function ($h'(k)$) can never be 0.

$H(k,i) = (h(k) + i \times 0) \bmod m$

$= h(k) \bmod m$ = same

$\Rightarrow$ collision.

Q.

keys: 13, 17, 21, 2, 57, 28, 30, 27

$h(x) = x \bmod 11 \Rightarrow (m = 11)$

$h'(x) = 7 - (x \bmod 7)$ ← 0 to 6

∴ Cannot be zero

i) $h(13) = 13 \bmod 11 = 2$

ii) $h(17) = 17 \bmod 11 = 6$

iii) $h(21) = 21 \bmod 11 = 10$

iv) $h(2) = 2 \bmod 11 = 2$ Collision

$H(2,1) = (h(k) + 1 \cdot h'(k)) \bmod 11$

$h'(2) = 7 - 2 \bmod 7 = 5$

$\therefore H(2,1) = (2 + 1 \cdot 5) \bmod 11 = 7$ ✓

v) $h(57) = 57 \bmod 11 = 2$ Collision → $7 - 57 \bmod 7$

$H(57,1) = (h(57) + 1 \cdot h'(57)) \bmod 11$

$= (2 + 6) \bmod 11 = 8$ ✓

vi) $h(28) = 28 \bmod 11 = 6$ Collision

$H(28,1) = (h(28) + 1 \cdot h'(28)) \bmod 11$ 2nd

$= (6 + 7) \bmod 11 = 2$ Collision

$H(28,2) = (h(28) + 2 \cdot h'(28)) \bmod 11$

$= (6 + 2 \cdot 7) \bmod 11 = 9$ ✓

vii) $h(30) = 30 \bmod 11 = 8$ Collision

$H(30,1) = (h(30) + 1 \cdot h'(30)) \bmod 11$

$= (8 + 5) \bmod 11 = 2$ 2nd Collision

$H(30,2) = (h(30) + 2 \cdot h'(30)) \bmod 11$

$= (8 + 2 \cdot 5) \bmod 11 = 7$ 3rd collision

$H(30,3) = (h(30) + 3 \cdot h'(30)) \bmod 11$

$= (8 + 3 \cdot 5) \bmod 11 = 1$ ✓

viii) $h(27) = 27 \bmod 11 = 5$

D. Hashing

$h(2) = 2, 7$

Double Hashing

$h(57) = 2, 8$

Same resolution path is not followed.

Overhead

↳ Computing two hash functions

Hence more time complexity.

In all the above discussed CRTs, if we want to perform deletion, we need to Re-Hash all remaining keys after deletion.

e.g.

keys: 31, 26, 43, 27, 34, 12, 46, 14, 58

m = 12

i) delete 26

ii) search 14

$h(14) = 14 \bmod 12 = 2$

point to a free slot

Re-hash all remaining keys

0	12
1	58
2	Free ~~26~~
3	27
4	14
5	
6	
7	31
8	43
9	
10	34
11	46

To resolve this problem, there's another CRT called Separate chaining is used.

* Load Factor (λ):

$\lambda = \frac{n}{m}$ → no of keys; → table size

$n = 20$, $m = 40$, $\lambda = \frac{20}{40} = \frac{1}{2}$

$n = 30$, $m = 40$, $\lambda = \frac{30}{40} = \frac{3}{4}$

$n = 60$, $m = 40$, $\lambda = \frac{60}{40}$

$\lambda \leq 1$

Linear Probing, Quad. Probing, Double Hashing

$\lambda = \frac{n}{m}$ → No. of keys; → Table size

SEPARATE CHAINING

When the load factor $\lambda > 1$ i.e. # of keys are more than the table size, collision is resolved using separate chaining technique which lists the elements and then hashed to locations/ buckets in table.

Each position may be just a link to the list (direct chaining) or may be an item and a link, essentially, the head of a list.

e.g. m=10

keys: 400, 300, 125, 625, 36, 96, 106, 500

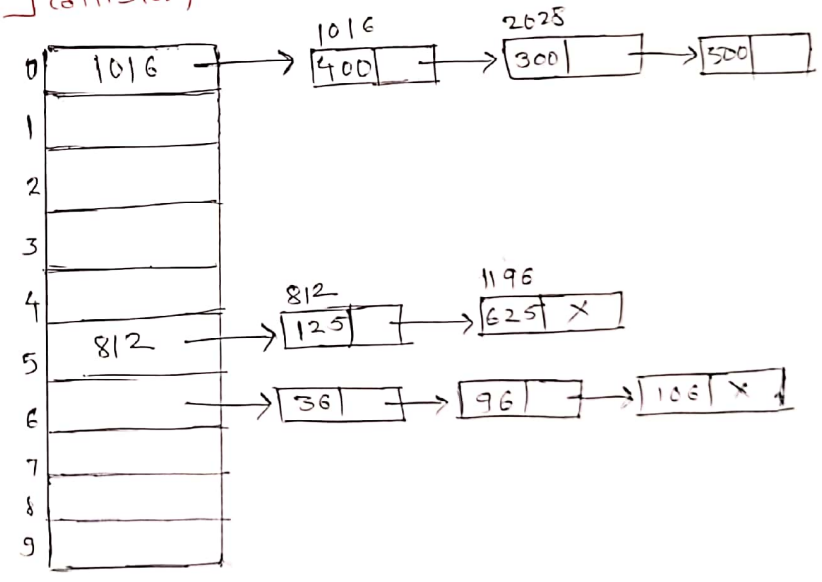
h(400) = 400 mod 10 = 0

h(300) = 300 mod 10 = 0] collision

h(125) = 125 mod 10 = 5

h(625) = 625 mod 10 = 5

h(36) = 36 mod 10 = 6



Worst Case → All keys will map to same bucket (e.g. 700, 800, 900, 600, 70)

In the worst case, time complexity will be an issue.

Instead of lists, AVL trees can be used by creating nodes & then mapping its root's address to a bucket.

B-tree can also be used. That way, time complexity will be reduced. ($O(\log n)$)

Q. Consider a double-hashing, in which the primary hash func. $h(k) = k \bmod 23$ & sec. hash func. $h_2(k) = 1 + (k \bmod 19)$. Then the addr. returned by probe 1 in the seq. (assume probe sequence begins at probe 0) for key $k = 90$ is 13

$m = 23$

$$H(k,i) = (h_1(k) + i \cdot h_2(k)) \bmod m$$

$i = 1$, $m = 23$

$h_1(k) = k \bmod 23$

$h_2(k) = 1 + (k \bmod 19)$

$k = 90$

$h_1(90) = 90 \bmod 23 = 21$

$h_2(90) = 1 + 90 \bmod 19 = 15$

$\therefore H(K,1) = (21 + 1.15) \bmod 23 = 36 \bmod 23 = 13$

Q. Given a Hash table T with 25 slots that store 2000 elements, the load factor, α, is 80

$\alpha = \frac{2000}{25} = 80$

GATE 2015

Q. Which one of the foll. hash func. on integers will distribute keys most uniformly over 10 buckets number 0 to 9 for i ranging from 0 to 2020?

a) $h(i) = i^2 \bmod 10$

b) $h(i) = i^3 \bmod 10$

c) $h(i) = (11 \times i^2) \bmod 10$

d) $h(i) = (12 \times i^2) \bmod 10$

	a	b	c	d
0	0	0	same	odd
1	1	1	as	numbered
2	4	8	option	buckets
3	9	7		↓
4	6	4		Empty.
5	5	5	a)	
6	6	6		
7	9	3		
8	4	2		
9	1	9		

Q. Consider a hash table with 100 slots. Collisions are resolved using Chaining. Assuming simple uniform hashing, What is the probability that the first 3 slots are unfilled after the first 3 insertions.

A) $(97 \times 97 \times 97) / 100^3$

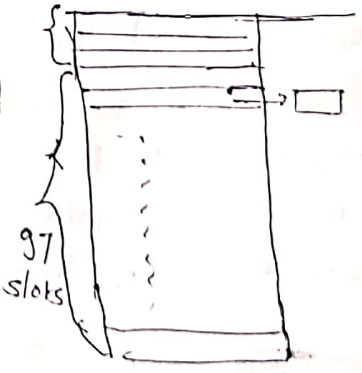
B) $(99 \times 98 \times 97) / 100^3$

C) $(97 \times 96 \times 95) / 100^3$

D) $(97 \times 96 \times 95) / (3! \times 100^3)$

Probability of 1st key not getting inserted at first 3 slots

$= \frac{97}{100}$ (P_1)



2nd insertion

↳ $\frac{97}{100}$ & not $\frac{96}{100}$

as it is chaining, Hash table never fills up

3rd insertion

probability of

↳ $\frac{97}{100}$

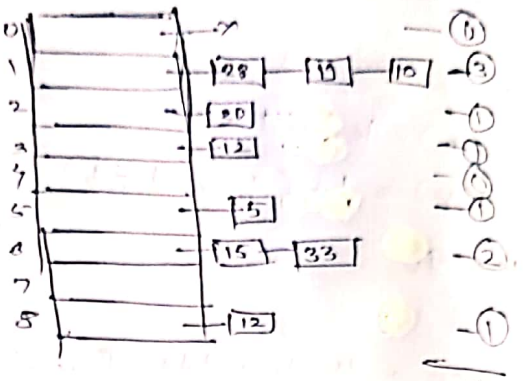
$\therefore$ Probability that first 3 slots are unfilled after the 1st 3 insertions.

$= P_1 \times P_2 \times P_3 = \frac{97}{100} \times \frac{97}{100} \times \frac{97}{100}$

Q. Consider a hash table of 9 slots. The hash function is $h(k) = k \bmod 9$. The collisions are resolved by chaining. 9 keys that are inserted:
5, 28, 19, 15, 20, 33, 12, 17, 10. The max^m, min & average chain lengths in hash table respectively are:

a) 3, 0, and 1
b) 3, 3 and 3
c) 4, 0 and 1
d) 3, 0 and 2

$h(5) = 5\%9 = 5$
$h(28) = 1$
$h(19) = 1$
$h(15) = 6$
$h(20) = 2$
$h(33) = 6$
$h(12) = 3$
$h(17) = 8$
$h(10) = 1$



GATE 2014

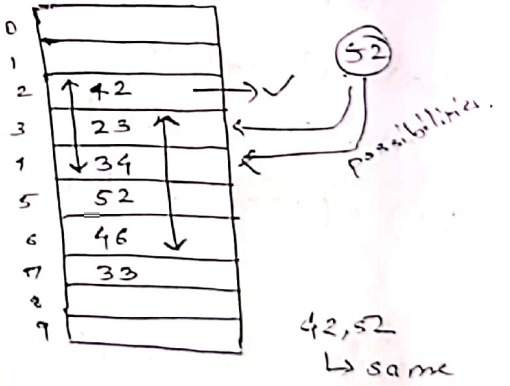
Max^m: 3, Min: 0

$$\text{Avg} = \frac{0+3+1+1+0+1+2+0+1}{9} = \frac{9}{9} = 1$$

Q. A hash table of length 10 uses open addressing with $h(k) = k \bmod 10$ & linear probing. After inserting 6 values into an empty hash table, table is shown below. Which one of foll. choices give a possible order in which the key values could have been inserted in table?

a) 46, 42, 34, 52, 23, 33 → 1st cond^n fails, 23 comes after 52.
b) 34, 42, 23, 52, 33, 46 → fails 2nd cond^n
c) 46, 34, 42, 23, 52, 33 → Both cond^ns satisfied
d) 42, 46, 33, 23, 34, 52 → fails 2nd cond^n

Ans: C



42 mod 10 = 2
∴ 42 should be assigned slot 2.

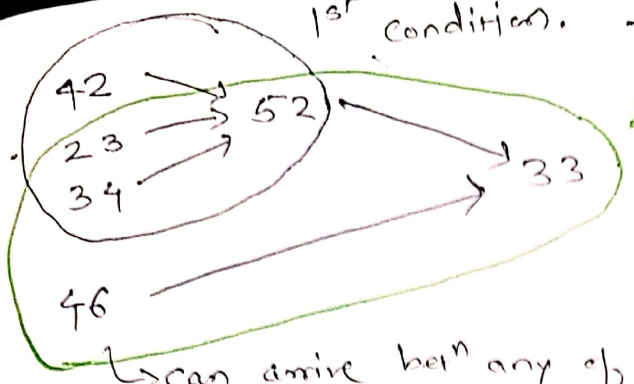
(1) 52 mod 10 = (2) collision. ∴ 52 should be assigned slot 3 since it's linear probing. But, 52 was assigned to slot 5. That means keys 42, 23, 34 must've been arrived before 52.

(2) 23 mod 10 = 3 ✓ assigned slot 3 for 23. 33 mod 10 = (3) collision. By the method of linear probing 33 should be assigned slot 4. If collision occurs again, then slot 5. But, 33 was assigned slot 7. That means 23, 34, 52, 46 must've been arrived before 33.

Q. How many diff. insertion sequences of the key values using same h(k) & linear probing will result in the hash table.

a) 10
b) 20
c) 30
d) 40

Refer Hash Table from the example above.



1st condition. ⟶ 6 possibilities - 3!

2nd Condn

× 42, × 23, × 34, × 52 ×

42, 34, 23 52

34, 42, 23 52

34, 23, 42 52

23, 42, 34 52

× 23, × 34, × 42 × 52 ×

↳ can arrive bet^n any of the two numbers among these 6 possibilities

∴ 5 × 3! = 5 × 6 = 30 //

∴ There are 5 ways to arrive among any of two no.s from above possibilities

1. Consider a hash table of size 11 that uses open addressing with linear probing. Let h(k) = k mod 11 be the hash func. used. A sequence of records with keys 43, 36, 92, 87, 11, 4, 71, 13, 14 is inserted into initially empty table, the bins of which are indexed from 0 to 10. What is the index of the bin into which the last record is inserted?

a) 2
b) 4
c) 6
d) 7 ✓

43 % 11 = 10
36 = 3
92 = 4
87 = ~~10~~ 0
11 = 0 1
4 = 5
71 = 6
13 = 2
14 = ~~3 4 5 6~~ 7 //