DS

#) Hashing: (O(1) Search time EC)

*) Direct Address Table:
- key itself is address
- Not efficient, as if key = 2^{100}, we need 2^{100} slots, so use HF

*) Hash function:

1. Division modulo: $h(x) = x \bmod m$

Ex:1) m = 1000 (0-999)
123789456 % m = 456

Ex:2) m = 8 (0-7) $\Rightarrow 2^3$
10100101 % 2^3 ⇒ LSB 3 bits
⇒ 101 = 5

Note:) • Don't pick m value as exactly powers of 2, cuz · HF(key) = LSB k-bits always.
• Pick m value as prime which is not near to a power of 2

2. Mid square: i) $(key)^2$
ii) Take it's middle

Ex) m = 1000 (0-999) ⇒ Majority indexes are of 3 digits [100 - 999]
key = 87431
$(87431)^2$
= 7644179761 = 417 or 179
1 2 3 4 5 6 7 8 9 10

3. Digit Extraction (worst method) (or Truncation)

Ex:) m = 1000 (0-999)
key = 45 61 23 78 91
HF(key) = 639

4. Folding:
i) Fold Shift | ii) Fold boundary

m = 1000 (0-999)
key = 456 789 123

456	136	456
789	008	123
123	144	579
1368		

HF(key) = 144 | HF(key) = 579

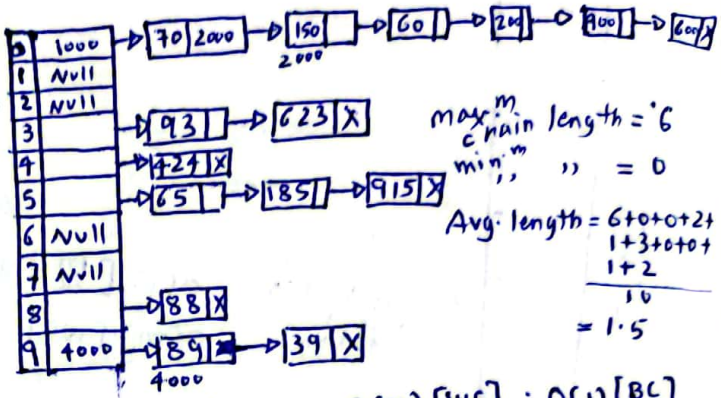
*) Collision Resolution techniques:
a) chaining (store outside)
b) Open addressing (store inside)
- Linear Probing
- Quadratic ''
- Double hashing

Note:) α = loading factor = $\frac{\text{Keys}}{\text{Slots}}$; $t \le \frac{1}{1-\alpha}$

1. Chaining:

Ex:) m = 10 (0-9) HF(K) = K mod m

K = 70, 84, 65, 150, 39, 88, 60, 200, 93, 185, 623, 915, 900, 424, 600



- Search time = O(n) [WC] ; O(1) [BC AC]
- Insertion = O(1)
- Deletion = O(n) [WC]; O(1) [BC AC]
- Insert-x if not exist = O(n) [WC], O(1) [BC AC]

2. Linear Probing:

- h(x) = x % m
- h'(x) = (h(x)+i) % m
  i ∈ [0, ∞) = Probe no.

Ex: m = 10 (0-9)
Keys = 75, 93, 46, 55, 79, 23, 89, 65.

0	89 ⇒ 1 collision
1	
2	
3	93
4	23 ⇒ 2 collision
5	75
6	46
7	55 ⇒ 2 collisions
8	65 ⇒ 3 collisions
9	79

Total collisions = 7

- Search Time = O(1) [BC]
  Insertion '' = O(m) [WC AC]
  Deletion '' = O(m) [WC AC]
  (Bec. of Primary clustering)

3. Quadratic Probing:

h(x) = x % m
h'(x) = (h(x)+i^2) % m

Ex:) ... same
... 23, 89, 65, 95

0	79
1	
2	
3	93
4	
5	75
6	46
7	
8	
9	55

for 55
$5+0^2 = 5$
$1^2 = 6$] 2 coll.
$2^2 = 9$

for 79
$9+0^2 = 9$] 1 coll.
$1^2 = 0$

Similarly for rest

95 not inserted anywhere

Total collisions = 21

- Search Time = O(1) [BC]
  Insertion '' = O(m) [WC AC]
  Deletion ''
- Deletion handled by special symbol $

4. Double Hashing: (Best method)

$h_1(x) = x \% m$
★ $h_2(x) = 1 + (x \% (m-2))$
★ $h'(x) = (h_1(x) + i \cdot h_2(x)) \% m$

$i \in [0, m-1]$

Ex:) Key = 75, 93, 46, 55, 79, 23, 89, 65
m = 10

0	
1	55
2	
3	93
4	
5	75
6	46
7	23
8	
9	79

for 55
$= 5 + 0 \cdot h_2(55)$
$h_2(55) = 1 + (55 \% 8) = 8$
5 ✗
$= 5 + 1 \cdot 8 = 3$ ✗ — 2 coll.
$= 5 + 2 \cdot 8 = 1$ ✓
Similarly rest
89, 65 not inserted

- Search time = O(1) [BC] [AC]
- Insert " = O(m) [WC]
- Delete " = O(m) [WC]

Note:) Ex:) m = 10 (0–9) use LP
k = 20, 54, 32, 61, 33, 70, 30, 40

0	20
1	61
2	32
3	33
4	54
5	70
6	30
7	40
8	
9	

Here, Till 33 insertion, no collisions, so those 5 no.s can be arranged in 5! ways
Primary clustering (a problem in LP)
[i.e. clustering in linear manner]

Note:) ...same... use QP

0	20
1	61
2	32
3	33
4	54
5	
6	30
7	
8	
9	70

Less harmful than primary clustering
secondary clustering (a problem in QP)
[i.e. clustering in quadratic manner]

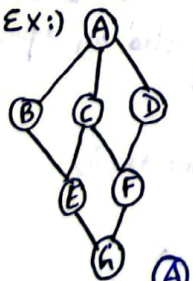
Graph Traversal:

Time Complexity:

- when graph stored as adj. matrix = $O(V^2)$
- " " adj. list = $O(e+v)$

1. BFT (or level order traversal):

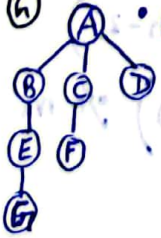
- Me followed by my adjacent all
- uses Queve
- Ex:)



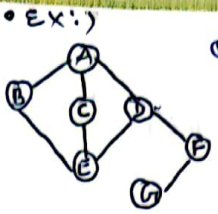
Queve

A	BCD	E	F	G		

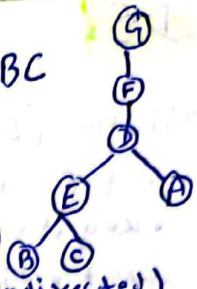
A B C D E F G
- BCD can be arranged 3! ways
- SC = V + V [flag + Queve] = O(V)
- BFT tree (spanning tree)



- Ex:)



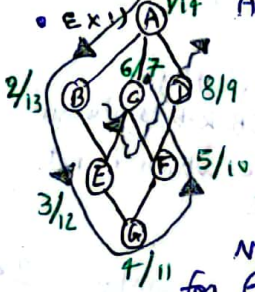
Queve: G, F, D, E A, B C



- App^n's of BFT:— (For undirected graph)
1 → Find spanning tree
2 → Find cycle in graph (Directed/undirected)
4 → Find no. of connected components
5 → Find single source shortest path (For unweighted graph)
6 → Find graph is bipartite or not.
3 → Find graph is connected or not

2. DFT:

- Me followed by my adjacent one
- uses stack
- Ex:) A B E G F C D



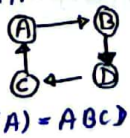
min.m Stack size = 6
max.m stack size = 7

Numbering (push-pop):—
- for A B E G F C D
1/14 2/13 3/12 4/11 5/10 6/7 8/9
- for E C A B D F G
1/14 2/13 3/12 4/5 6/11 7/10 8/9

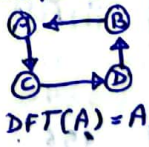
- App^n's of DFT:—
→ 1 to 4 = same
→ Single source shortest path we can't compute by DFT
→ 5: Find no. of strongly connected components in directed graph
→ 6: Find no. of articulation points in graph. (but BFT can't do it)

Note:) To find out strongly connected components:— Apply 2 times DFT on same vertex, one time with given edges one time with reversed edges.

Ex:)



DFT(A) = A B C D



DFT(A) = A B C D

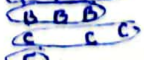
⇒ strongly connected

Classification of edges:—

1. DFT
- i) Directed: Tree, Back, Forward, Cross — 4
- ii) Un-Dir: Tree, Back — 2

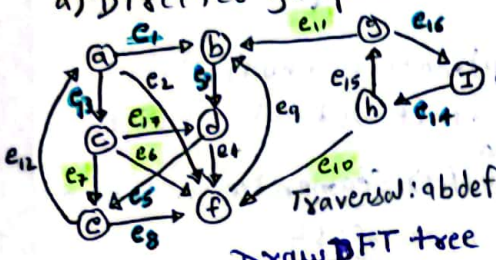
2. BFT
- i) Dir.: Tree, Back, Cross — BFT is bad as to remember it, so BFT doesn't give a Fwd i.e. Forward — 3
- ii) Un-Dir: Tree, Cross — 2

to remember:—



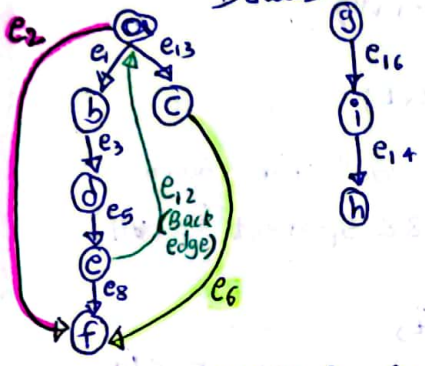
I) DFT:

a) Directed graph:



Traversal: abdefc gih

Draw DFT tree



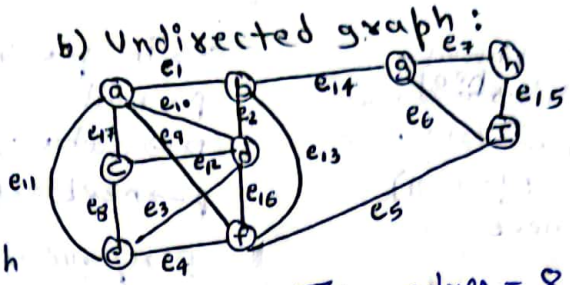
Tree edges = 7; e_1 e_{13} e_3 e_5 e_8 e_{16} e_{14}

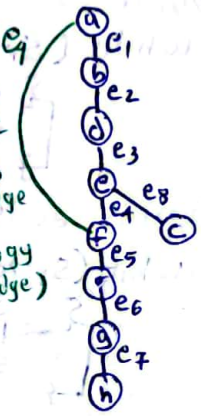
Back edges = 3; e_{12} e_9 e_{15}

Forward edges = 2; e_2 e_4

cross edges = 5; e_6 e_7 e_{17} e_{10} e_{11}

b) Undirected graph:





Tree edges = 8

e_1 e_2 e_3 e_4 e_5 e_6 e_7 e_8

Back edges = 9

e_9 e_{10} e_{11} e_{12} e_{13} e_{14} e_{15} e_{16} e_{17}

- Back edge present ⇒ cycle

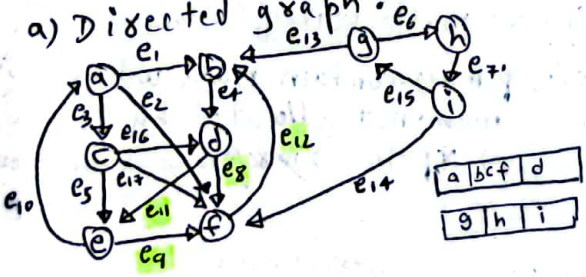
Note: In DFT, if 'k' edges are tree edges then, # connected components = n − k

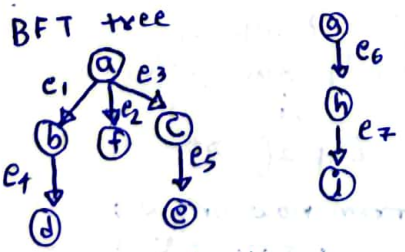
Note: In BFT, if T contains cycle then, G has strongly connected component. And, each cycle in T forms a SCC

Note: if T contains cycle then Topological sort not possible

II) BFT:

a) Directed graph:





Tree edges = 7

e_1 e_2 e_3 e_4 e_5 e_6 e_7

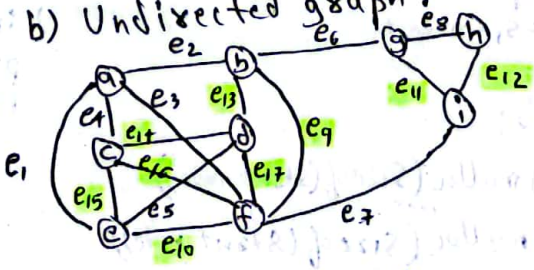
cross edges = 8

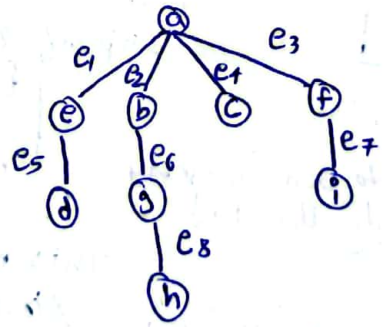
e_8 e_9 e_{11} e_{12} e_{13} e_{14} e_{16} e_{17}

Back edges = 2

e_{10}, e_{15}

b) Undirected graph:





Tree edges = 8

e_1 e_2 e_3 e_4 e_5 e_6 e_7 e_8

cross edges = 9

e_9 e_{10} e_{11} e_{12} e_{13} e_{14} e_{15} e_{16} e_{17}

Note:) Ascending order ⇒ p->data ≤ p->next->data

Increasing order ⇒ p->data < p->next->data

Note:) malloc ⇒ non-contiguous allocation

calloc ⇒ contiguous allocation initialized with 0

realloc(1000, 11) ⇒ 1000 onwards 11 B

↑ increase size or decrease size

Heap Area i.e. DMA

Linked List:

- Nodes need not be continuous
- LL is created inside ~~stack~~ heap area.
- WAP find address of 2nd last node

```
while(S->next->next != null)
        S = S->next
return (S)
```

(or)

```
while (S->next != null)
{
    P = S
    S = S->next
}
return(P)
```

TC: O(n)

- WAP insert data-x at end of SLL

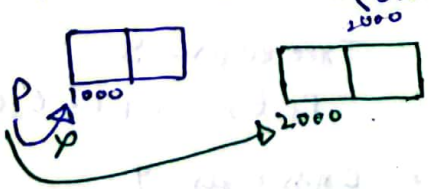
```
while(S != null)
{
    q = S;          O(n)
    S = S->next;
}
```

- WAP insert data-x before a node with data-y

```
while(S1->data != y && S1->next != null)
{
    P = S1
    S1 = S1->next
}
```

- Memory Leak:

```
P = (Struct node *) malloc (Sizeof(Struct node))
P = (Struct node *) malloc (Sizeof(Struct node))
```



- Dangling ptr: Pointing to some memory which is deallocated


```
free(P); // Allocated → Deallocated
pf(P); // 2000
pf(P->data); // gb
// Dangling ptr gives gb, not error
P = null;
pf(P->data); // Error: Dereference to null
```

↓ Segmentation Error

- Dereference to null: (Runtime Error) as in previous example

```
P = null
P->data ⇒ (*P).data ⇒ (*null).data
P->next ⇒ (*P).next ⇒ (*null).next
```

Dereference to null ⇒ Segmentation Error

- WAP delete last node of SLL

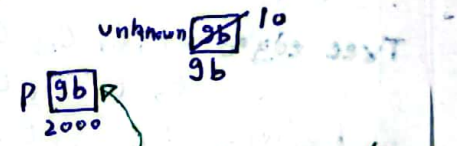
```
while (S1->next != null)
{
    P = S1;
    S1 = S1->next;
}
```

- WAP delete node containing data-x

```
while(S1->data != x && S1->next != null)
{
    P = S1;
    S1 = S1->next;
}
```

- Note:)

```
int* f(void)
{
    int *p;
    *p = 10;
    ret(p);
}
```



This gb can be null also
⇒ *p=10 will give Dereference to null ⇒ Error

- Note:)

```
int *p;
P = (int *) malloc(Sizeof(int));
*P = 10;
```

Here, p may contain null when mem not allocated by OS
⇒ *p=10 ⇒ Dereference to null ⇒ Error

- Note:)

```
typedef struct node
{
    int d;
    x *n;
} x;
```

OR

```
struct node
{
    int d;
    struct node *n;
};
typedef struct node x;
```

This means that from now onwards struct node का x बुलाया जाएगा

- WAP K^{th} node from last;

```
S1 = S
while(K != 1)
{
    S1 = S1->next;
    K--;
}
while(S1->next != null)
{
    S = S->next;
    S1 = S1->next;
}
return (S)
```

Link List for interviews :—

• WAP middle node address

```
mid(S){ S1=S, c=0;
        while(S1!=null)
            { S1=S1->next
              c++;
            }
        c = ⌈c/2⌉
        while(c!=1)
            { S=S->next
              c--;
            }
        return(S)
      }
```

O(n)

Alter

```
mid(S){ P=S;
        // run P with 2x speed & S with 1x speed
        while(P!=null && P->next!=null
              && P->n->n!=null)
            { P=P->n->n;
              S=S->next;
            }
        return(S)
      }
```

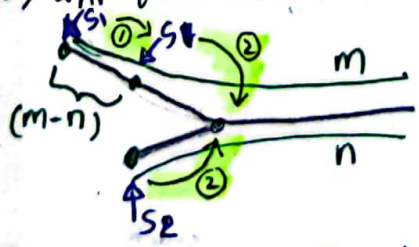
O(n)

• WAP Binary search on SLL

```
BS(S,x){
    m = mid(S)
    if(m->data==x) ret(m)
    else{
        if(x<m->data){m->n=null
                      BS(S,x);}
        else {BS(m->n, x);}
    }
}
```

O(n)

• WAP find 1st common node in 2 joint SLL (not for create)



Count nodes in S_1, store in C_1
Count nodes in S_2, store in C_2
C = Subtract big − small; $C_2 - C_1$ (say)
move S_1 to right 'c' times
now, while $(S_1 != S_2)\{S_1 = S_1 \to n;\ S_2 = S_2 \to n;\}$

• WAP to reverse SLL

```
P=q=null;
while(S!=null)
    { p=q
      q=S
      S=S->next
      q->next=p
    }
```

plan to move ↕ Reverse

• WAP to detect cycle in SLL.

DFS:

① Assume one extra field 'flag' is there in every node.

② Visit every node & check flag

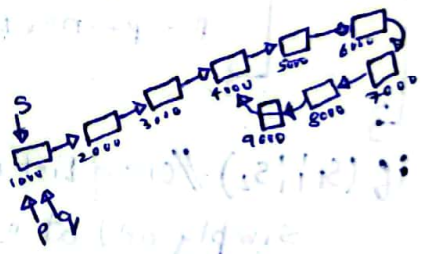
```
if(flag==0) flag=1;
else pf(cycle found)
```

TC: O(n) SC: O(n)

Alter

move P & q with different speeds

P	q
1000	1000
2000	3000
3000	5000
4000	7000
5000	9000
6000	5000
7000	7000



TC: O(n) SC: O(1)

• WAP Delete node where given pointer 'P' is pointing.

```
while(S->n!=P){S=S->n;}
S->n = P->n;
free(P)
P=null;
```

O(1) BC
O(n) WC, AC

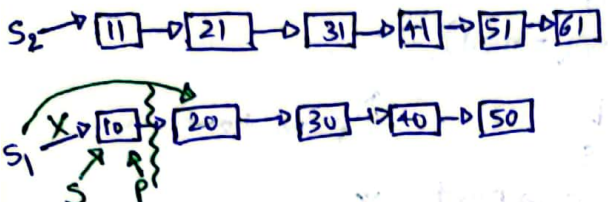
Alter

```
P->data = P->n->data;
q = P->next
P->next = q->next;
free(q)
q=null
```

O(1) EC

// But here, some address problem is there

•) Merging two sorted SLL




```
if(S1->data < S2->data)
{ S1 = S1->next;
  P = S;
  while(S1 != null || S2 != null)
  {
    if (S1->Data < S2->data)
        { P->next = S1
          S1 = S1->next
          P = P->next
        }
    else { P->next = S2
           S2 = S2->next
           P = P->next
         }
  }
  if (S1 || S2) //One of list completed
      simply add at end of S
        i.e. at P
}
```

TC: O[min.(m,n)] BC

O(m+n) AC, WC

•) Sort a SLL (by Mergesort):

```
mergeSort(S){ if(S->n == null) ret(S)
    m = mid(S) => n
    P = m->next
    m->next = null
  S1 = mergeSort(S) => T(n/2)
  S2 = mergeSort(P) => T(n/2)
  S = Merge(S1, S2) => n
  return(S) }
```

Merge(S1, S2){ compare S1->data, S2->data whichever smaller add at the end of resulting LL}

T(n) = 2T(n/2) + 2n

= O(n lgn) EC Inplace

•) Arrange SLL nodes such that 1st part contains even data, 2nd part contains odd data.

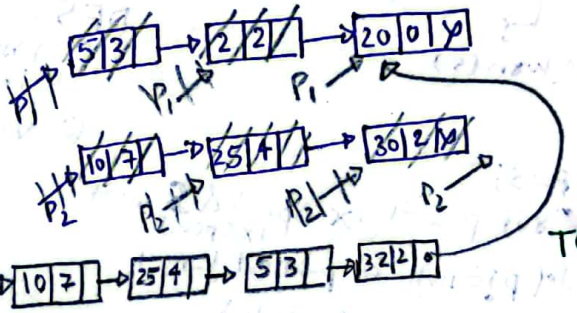
```
P = S
if(p->data % 2 == 0) add at beginning of LL
```

TC: O(n)

•) Polynomial addition (using SLL):

$5x^3 + 2x^2 + 20$

$10x^7 + 25x^4 + 30x^2$



TC: O(n) [WC/AC]

O(min(m,n)) [BC]

•) Polynomial mult.n (using SLL):

$P_1(5x^3 + 10x^2 + 7)\ P_2(10x^7 + 20x + 3)$

```
for( ; P1 != null; P1 = P1->next)
for( ; P2 != null; P2 = P2->next)
    { multiply coefficients;
      Add powers;
    }
```

TC: O(m·n)

##) Trees:—

*) Binary Tree:

•) WAP count leafs in BT

```
fun(struct BTnode *r)
{ 0. if (r == null) ret(0);
  1. if(r->lc == null && r->rc == null) ret(1);
     else{ 2. L = fun(r->lc) => T(n/2)
           3. R = fun(r->rc) => T(n/2)
           4. T = L+R => c
           5. ret(T);
         }
}
```

TC: O(n) [EC] (one scan)

BC/AC

T(n) = 2T(n/2) + c

$\lg_2 n$ stack space

= O(n)

WC

= T(n-1) + T(0) + c

n stack space

= O(n)

•> WAP count non-leaf nodes

1. ——— return(0);
4. T = 1+L+R

TC: O(n) [EC] stack: lgn [BC/AC] n [WC]

•> WAP count all nodes:

1. ——— return(1);
4. T = 1+L+R;

•> WAP count non-leaf nodes in left most path

1. ——— return(0);
4. T = 1+L;

Note:) Strict binary tree: exactly 0 or 2 child.

•> WAP to verify if given tree is strict BT

```
fun(r)
{ if(r==null) ret(1)
  if(r->lc==null && r->rc==null) ret(1)
  if(r->lc!=null && r->rc!=null)
      ret(fun(r->lc) && fun(r->rc));
  else ret(0)
```

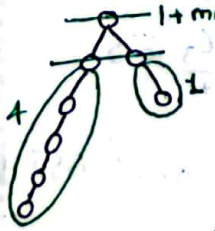
T(n/2) c T(n/2)

AC: $T(n) = 2T(n/2) + c$ → lgn → $= O(n)$

WC: $= T(n-1) + c$ → n → $= O(n)$

BC: $= O(1)$ [root & its 2 child]

•> WAP find height of BT



Levels = 6
Height = 5

```
fun(r)   TC: O(n) [EC] (one scan to total tree)
{ if(r==null) ret(0)
  if(r->lc==null && r->rc==null) ret(0)
  else{ LH = fun(r->lc) => T(n/2)
        RH = fun(r->rc) => T(n/2)
        H = 1+max(LH,RH) => C
        return(H)
```

$T(n) = 2T(n/2) + c = T(n-1) + T(0) + c$

•> WAP find levels of BT

if(r->lc==null && r->rc==null) ret(1)

•> WAP to verify two BT's equal or not

```
fun(r1, r2)
{ if(r1==null && r2==null) ret(1)
  if((r1==null && r2!=null) || (r1!=null && r2==null))
      ret(0)
  if(r1->data == r2->data)
      ret(fun(r1->lc, r2->lc) && fun(r1->rc, r2->rc))
  else
      ret(0)
```

TC: O(n) [AC/WC], O(1) [BC]
SC: n [WC], lgn [AC], 1 [BC]

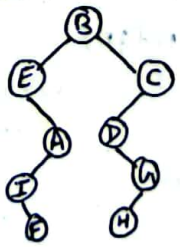
•> WAP Convert BT to mirror image

```
fun(r)
{ if(r==null) return(r)
  if(r->lc==null && r->rc==null) ret(r)
  else{ t = r->rc
        r->rc = fun(r->lc)
        r->lc = fun(t)
        return(r)
```

TC: O(n) [EC]

•> Tree traversal:

- Tree traversal is unique
- TC = O(e+v) = O(v+v-1) = O(V) [EC]
- SC = n [WC]
  lgn [BC, AC]
- $T(n) = 2T(n/2) + C = T(n-1) + T(0) + C$
- Ex:) Traversal



Pre: ls rs = BEAIFCDGH
In: ls r rs = EIFABDHGC
Post: ls rs r = FIAEHGDCB

- Ex:) Pre, In → Post

Pre: DEBCAFG
In: (D) B A G F C (E)

Post = ?

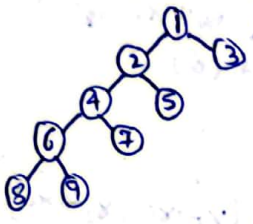
TC: $O(n^2)$ [n time linear search to find LST & RST]

Post = GFACBED

Post, In → Pre

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

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



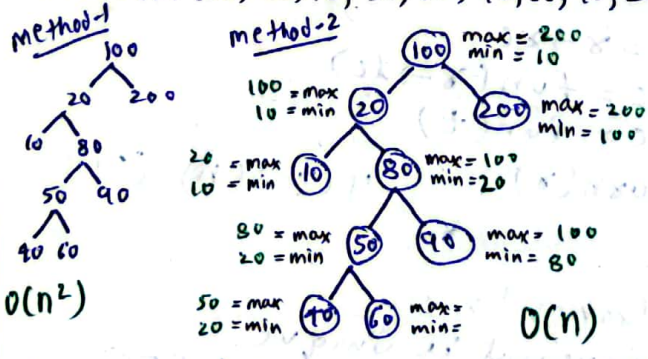
TC: $O(n^2)$

- For given Pre & post, unique BT not possible, but BT's possible
- For unique BT constr.^{n}, inorder needed.

*) Binary Search Tree (BST):

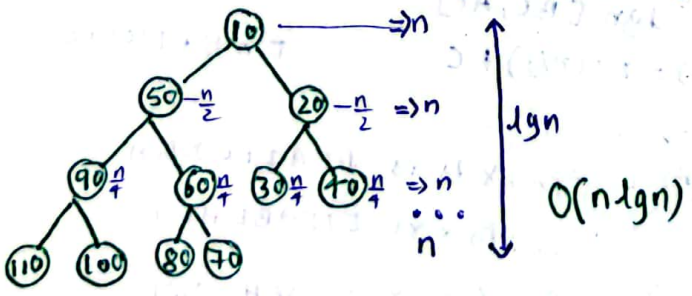
- LST ≤ root < RST
- Inorder of BST ⇒ Ascending (sorted) order ⇒ O(n)
- BST constr.^{n} from Pre (or) post ⇒ O(n)

Ex:) Pre: 100, 20, 10, 80, 50, 40, 60, 90, 200



Note:) Minheap constr.^{n} from Inorder ⇒ O(nlgn)

Ex: ((110 90 100) 50 (80 60 70)) 10 (30) 20 (40)



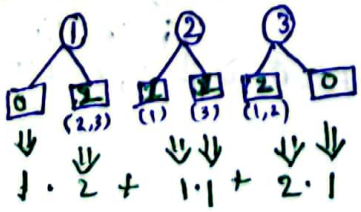
- No. of BST's possible questions:—

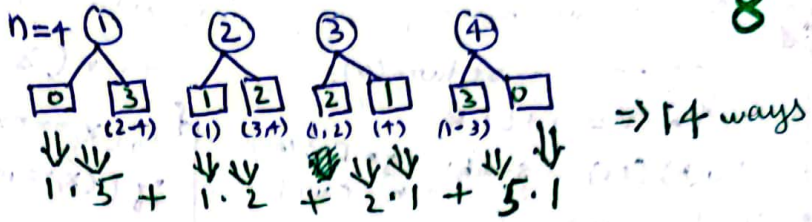
n = 0 (Empty) ⇒ 1 way

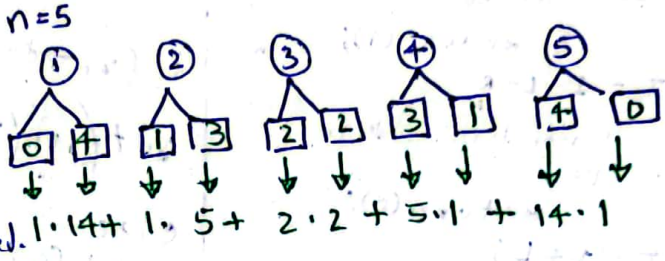
n = 1 ⇒ 1 way

n = 2 ⇒ 2 ways

n = 3 ⇒ 5 ways







$$BST(n) = \begin{cases} \sum_{L=0}^{n-1} \sum_{R=n-1}^{0} BST(L)\cdot BST(R); & n \neq 0 \text{ or } 1 \\ 1 & ; n = 0 \text{ or } 1 \end{cases}$$

L + R = n − 1

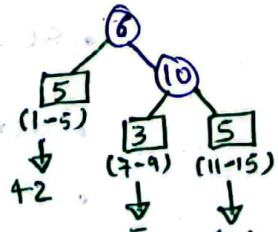
L → # nodes in the Left
R → # nodes in the Right

- Ex:) $BST(4) = BST(0)\cdot BST(3) + BST(1)\cdot BST(2) + BST(2)\cdot BST(1) + BST(3)\cdot BST(0)$
  $= 1\times5 + 1\times2 + 2\times1 + 5\times1 = 14$ BST's

- Direct formula

$$BST(n) = \frac{^{2n}C_n}{n+1}$$

- Ex:) BST's possible with 15 nodes (1–15) such that root is 6 & RST root is 10



$BST(5) = \frac{^{2\times5}C_5}{5+1} = 42$

$BST(3) = \frac{^{2\times3}C_3}{3+1} = 5$

⇒ 42 · 5 · 42 = 8820 BST's

- Note:) For 'n' nodes,
  No. of unlabelled BT's = No. of BST's

Ex:) n = 3;

No. of unlabelled BT's =  = 5 = No. of BST's

- Time complexity questions:—

Ex-1 To find min.^{m} element in BST

T(n) = 1 [BC] (root left child null)
n [WC] (root RC null)
lgn [AC] (Take leftmost path)

Ex-2 To find max.^{m} element

	BC	WC	AC
BST →	1	n	lgn [Same logic]
BT →	n	n	n [Always 1 scan needed cuz. unsorted]
AVL →	lgn	lgn	lgn [Tree balanced & sorted] data
Minheap →	n	n	n [1 scan needed, max.^{m} could b…

Ex-3 Find element-x which is present (i.e. Successful Search)

	BC	WC	AC
BST	1	n	lgn
BT	1	n	n
AVL	1	lgn	lgn
minheap	1	n	n

Ex-4 Find element-x which is not-present (i.e. unsuccessful Search)

	BC	WC	AC
BST	1	n	lgn
BT	n	n	n
AVL	lgn	lgn	lgn
minheap	n	n	n

Note: BST में AVL में ही lgn आ सकता है, मोरक्किमी में नहीं

• Insertion in BST:

① Create node with data-x ⇒ C

② Find x's correct place in BST ⇒ 1 [BC], n [WC], lgn [AC]

③ Link both of them ⇒ C

TC: O(1) BC
O(n) WC
O(lgn) AC

• Deletion in BST:

① Find node x ⇒ $f_1(n)$

② i) 0-child ⇒ Directly delete ⇒ C

ii) 1-child ⇒ Delete by connecting grandfather-grandchild ⇒ C

iii) 2-child ⇒ Delete x & replace by inorder predecessor/successor. ⇒ $f_2(n)$

TC: $f_1(n)+f_2(n)$ = O(1) BC
O(n) WC
O(lgn) AC

• Note:) To construct BST with n-nodes

TC = $O(n^2)$ WC
O(nlgn) AC, BC

*) AVL tree:

• BST where every node Bf = {0/+1/-1}

Tree	min.m Levels	max.m Levels
BST	lgn	n
AVL	lgn	lgn

• Left Rotation TC = Right rotation TC = O(1)

i.e.

Problem	Sol.n	TC
LL	R	O(1) for all
RR	L	
LR	LR	
RL	RL	

• AVL tree Insertion Algo:

① Insert element like in BST

② check if BF violated

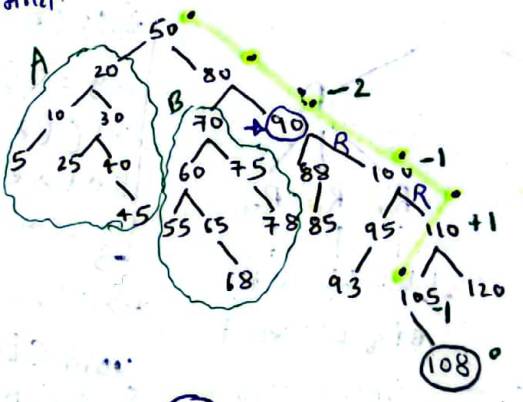
③ If violated, go two steps towards newly inserted node, identify problem (RR, RL, LR, LL) & rectify

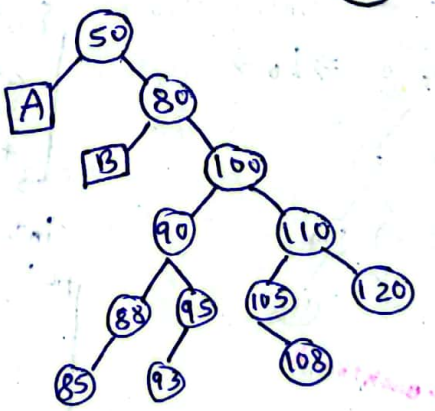
• AVL tree insertion TC:

TC = O(1) + O(lgn) + O(1) + O(lgn) + O(1)
(create node, Find position, inset, to check BF, for rectification if necessary)

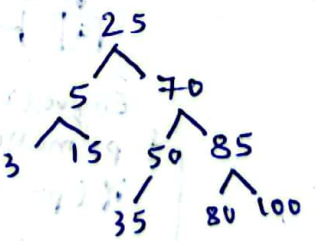
= O(lgn) [EC]

• Ex: Insert 108





• Ex: Construct AVL
{25, 50, 15, 70, 5, 80, 3, 35, 85, 100}



O(nlgn) [EC]

• In AVL tree after insertion while going back in that path max.m 1 problem can come (i.e. max.m 2 rotations) & min.m 0 problem can come (i.e. min.m 0 rotations)

• Min.m nodes for height 'h'

$N(h) = N(h-1) + N(h-2) + 1$

$N(1) = 2$
$N(2) = 4$

- Max.m height of AVL tree with n-nodes = $1.44 \lg_2 n$

[Apply this when bigger no. is given]

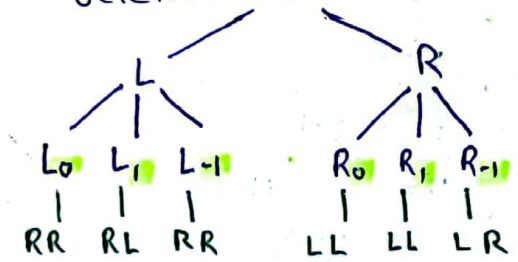
- Max.m nodes in tree of height 'h'

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

- Deletion in AVL:-

→ TC: $O(\lg n) + O(1) + O(\lg n)$
search Delete Backtrack
$= O(\lg n)$ [EC]

→ In terms of violated Bf node deletion has done



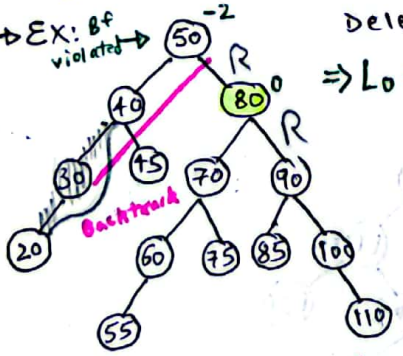
Bf of RST root

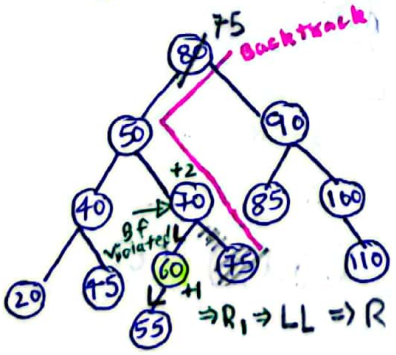
Bf of LST root

Delete: 30, 80

⇒ L_0 ⇒ RR ⇒ L

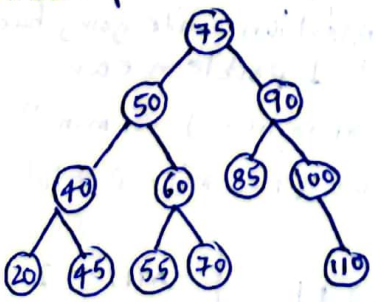
→ Ex: Bf violated





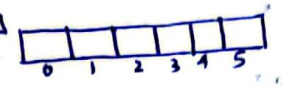
⇒ R_1 ⇒ LL ⇒ R

80's inorder predecessor = 75



#) Queue:

*) Linear Queue using array:




```
void Enque(int x)
{1. if (R+1==N) { pf(overflow)
                  return }
 2. if (R==-1) { F=R=0 }
 3. else       { R++; }
 4. Q[R] = x;
}
```

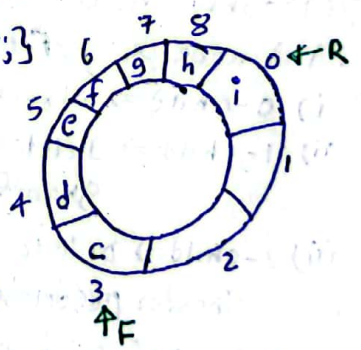
O(1) [EC]

```
int Deque(void)
{1. if (F==-1) { pf(underflow)
                 return }
 2. y = Q[F]
 3. if (F==R) { F=R=-1 }
 4. else      { F++; }
 5. return(y)
}
```

O(1) [EC]

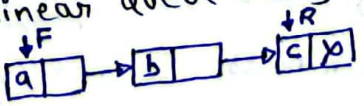
*) Circular Queue using array

```
void CEnque(int x)
{1. if((R+1) mod N == F){pf(overflow)
                         return }
 2. same
 3. else { R=(R+1) mod N; }
 4. same
}
```




```
int CDeque(void)
{1. same
 2.
 3.
 4. else { F=(F+1) mod N; }
 5. same
}
```

*) Linear Queue using LL:




```
Enque(x)
{ p=malloc()
  if (p==null) {pf(memoxy overflow)
                return }
  p->data = x
  p->next = null
  if (R==null) { F=R=P }
  else { R->next = P
         R = P }
}
```

heap

O(1) [EC]

```
Deque()
{ if(F==null){pf(underflow)
              return
  P=F
  if(F==R){ y=F->data
            free(F)
            F=R=null
            return(y)}
  F=F->next
  y=P->data
  free(P)
  P=null
  return(y)
}
```
O(1) [EC]

Note:) Alternative logic also you can do, i.e. make deque easier & enque complex. 11

Ex:) 5 enque, 5 deque are done on a queue implemented by stack. How many push & pop are made?

Lets consider; Enque easy & Deque costly

5 enque ⇒ 5 push in S_1

1st Deque ⇒ 5 pop from S_1
5 push to S_2
1 pop from S_2

2nd Deque ⇒ 1 pop from S_2

3rd Deque ⇒ 1 pop " "

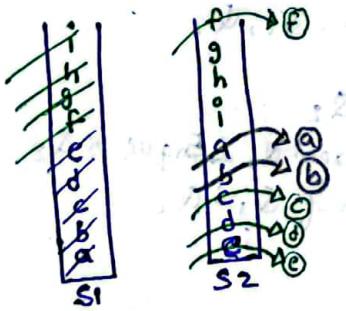
4th " ⇒ "

5th " ⇒ "

10 push & 10 pop

*) Linear Queue using Stack:

a	b	c	d	e	f	g	h	i



En(a)	push(a), S_1
En(b)	— b S_1
En(c)	— c —
En(d)	— d —
En(e)	— e —
De() ⇒ a	Pop all from S_1 & push into S_2
	Pop from S_2 ⇒ a
De() ⇒ b	Pop from S_2 ⇒ b
En(f)	push(f), S_1
En(g)	— g —
En(h)	— h —
En(i)	— i —
De() ⇒ e	Pop from S_2 ⇒ c
De() ⇒ d	⇒ d
De() ⇒ e	⇒ e
De() ⇒ f	Pop all from S_1 & push into S_2
	Pop from S_2 ⇒ f

```
Enque(x)
{ push(x,S1)
}
```
O(1) [Ec]

```
Deque()
{ if(S2 is not empty) ret[pop(S2)]
  else{ while(S1 is not empty)
          { x=pop(S1)
            push(x,S2)
          }
        ret[pop(S2)]
  }
```
O(1) [BC, AC]
O(n) [WC]

*) Priority Queue:

•) Types of priority queue:

1) Ascending priority queue 2) Descending priority queue

i/p: 50, 20, 80, 10, 70, 60, 90

EQ() | 50 | 20 | 80 | 10 | 70 | 60 | 90 |

DQ() 10, 20, 50, 60, 70, 80, 90

Smallest element ⇒ highest priority

EQ() same.

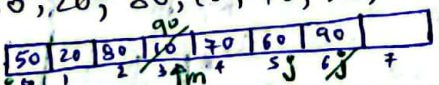
DQ() 90, 80, 70, 60, 50, 20, 10

Largest element ⇒ highest priority

•) Ascending priority queue implementation:

I) Using unsorted array

50, 20, 80, 10, 70, 60, 90

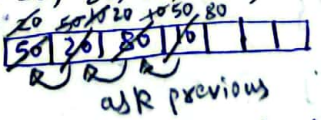


• Enqueue() {same like linear queue} O(1) [EC]

• Dequeue() { Find position of min^m element (say m)
x = a[m]
a[m] = a[j]
j--
ret(x).
}
O(n) [EC]

II) Using sorted array

50, 20, 80, 10, 70, 60, 90



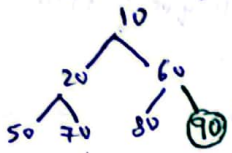

```
Enqueue(x){ 1. insert at a[i],
            2. ask previous iteratively until you find
               right position
          }
```
O(n) [WC]
O(1) [BC]

```
Dequeue(){ x=a[i]
           i++
         }
```
O(1) [EC]

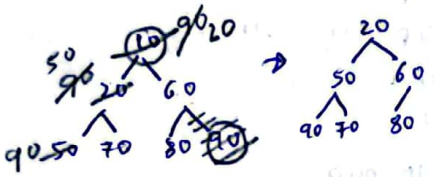
III) Using minheap (Best method):

50, 20, 80, 10, 70, 60, 90

Enque(){ 1. insert in minheap
2. minheapify()
}
O(lgn) [EC]



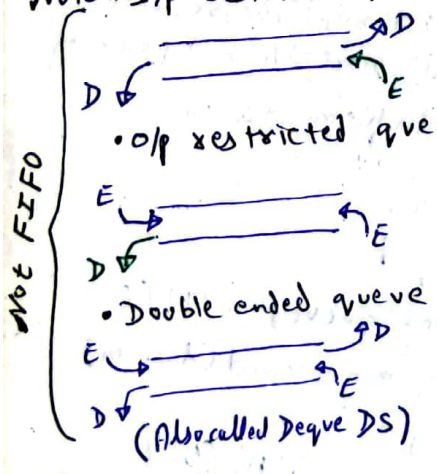
Deque() { 1. Delete root
2. heapify()
}
O(lgn) [EC]



Note:) Ascending priority queue ⇒ minheap
Descending " " ⇒ maxheap

•) Descending priority queue impl.?
Same approach

Note:) I/p restricted queue



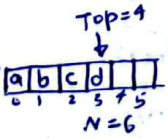
• o/p restricted queue

• Double ended queue

(Also called Deque DS)

Stack:

*) Stack using array




```
push(x)
{ if(Top+1 == N) { pf(overflow)
                   exit }
  Top++
  S[Top] = x        O(1) [EC]
}
```

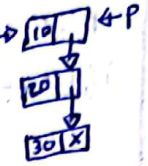
```
Pop()
{ if(Top == -1) { pf(underflow)
                  exit }
  y = S[top]
  Top--
  return(y)         O(1) [EC]
}
```

*) Stack using LL:

```
push(x){ p = malloc()
         if(p == null) { pf(overflow)
                         return }
         p->data = x
         p->next = Top
         Top = p;
}
```

i.e. always insert at beginning of LL

O(1) [EC]




```
pop(){ if(top == null) { pf(underflow)
                         return }
       y = Top->data
       p = Top
       Top = Top->next      O(1) [EC]
       free(p)
       p = null
       ret(y)
}
```

*) Stack using Queue:

Stack: S Queues: Q_1, Q_2

```
push(S,x){ Enque(x), Q2
           Deque() all from Q1 & Enque to Q2
           Swap names of Q1 & Q2
}
```

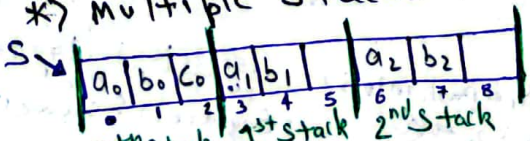
O(n)

```
Pop(S){ Dequeue(), Q1
}
```

O(1)

Note: You can make pop as costlier one also

*) Multiple Stacks in single array:-



N = 9 → Array size
m = 3 → No. of Stacks
n = N/m = 3 → size of one stack

$T_0 = -1$ $T_1 = 2$ $T_2 = 5$

•) For determining initial TOS:

Stack No.	I-TOS
0th	$0 \times \frac{9}{3} - 1 = -1$
1st	$1 \times \frac{9}{3} - 1 = 2$
...	...
ith	$i \times \frac{N}{m} - 1$ } → underflow condition

•)

Stack No.	overflow conditions
0th	if($T_0 == I\text{-}TOS\text{-}T_1$) i.e. $T_0 == (1 \times \frac{9}{3}) - 1$
1st	if($T_1 == I\text{-}TOS\text{-}T_2$) i.e. $T_1 == (2 \times \frac{9}{3}) - 1$
...	...
ith	if($T_i == I\text{-}TOS\text{-}T_{i+1}$) i.e. $T_i == (i+1) \times \frac{N}{m} - 1$ → overflow condition

•) push(x,i) // i is stack no.

```
{ if(Ti == (i+1)*N/m - 1) { pf(stack overflow); return; }
  else { Ti++; S[Ti] = x; }
}
```

```
Pop(i)
{ if(Ti == i*N/m - 1) { pf(stack underflow); return }
  else { y = S[Ti]; Ti--; return(y); }
}
```

•) Drawback: its not efficient bec. if one stack is full, its giving stack overflow, even though a lot of space available.

*) Multiple stacks in single array (Efficiently)

•) Two stacks in one array

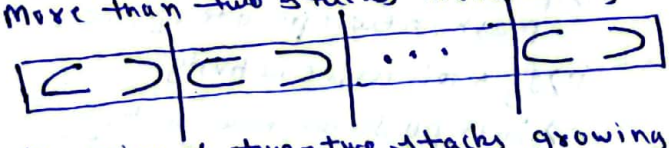
I-TOS: $T_L = -1$ $T_R = 9$

Push: T_L++ T_R--

Overflow: $T_L+1 == T_R$

Underflow: For Left stack $(T_L == -1)$ For right stack $(T_R == N)$

•) More than two stacks in one array



make pairs of two-two stacks growing in opposite direction.

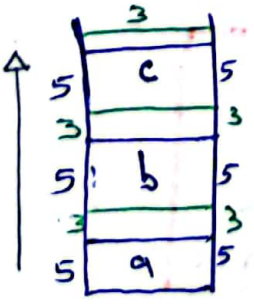
*) Avg. life of element in Stack:

Ex1:) n = 3 (a,b,c) [means continuously 3 push, 3 pop]

each stack operation time = x = 5 min.

elapsed time b/w 2 operations = y = 3 min.

Lifetime of :-



c = 3

b = (3,5),(3,5),3 = 19 (push c, pop c)

a = (3,5),(3,5),(3,5),(3,5),3 = 35 (push b, push c, pop c, pop b)

Avg. lifetime = $\frac{3+19+35}{3}$

= 19 min

Ex2:) n = 5 | x = 7 → push/pop, y = 5 → elapsed

c = 5

d = (5,7),(5,7),5 = 29

c = 4*(5,7),5 = 53

b = 6*(5,7),5 = 77

a = 8*(5,7),5 = 101

Avg. = $\frac{265}{5}$ = 53 min.

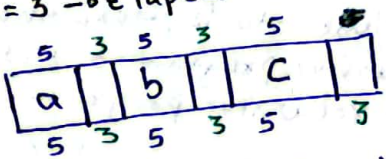
Note:) Avg. life = n(x+y) − x ; n = # of elements

*) Avg. life of element in Queue:

Ex: n = 3 (a,b,c) [continuous 3 Enq., continuous 3 Deq.

x = 5 → push/pop

y = 3 → elapsed



c = (3,5),(3,5),3 = 19

b = (3,5),(3,5),3 = 19

a = (3,5),(3,5),3 = 19

avg. = $\frac{3*19}{3}$ = 19 min.

*) Stack applications:

1) Recursion
2) Infix to postfix
3) Postfix evaluation
4) Prefix to postfix
5) Fibonacci series
6) Tower of hanoi

*) Recursion:

1) Tail recursion:

```
f(){ if{-}
     else{ ≡
           f();   // only 1 recursive call, which is at end
     }
}
```

Drawback: Wastes lot of stack space & time

Adv.: You can write equivalent non-Recursive program using loops to save stack space.

2) Non-tail recursion:

```
f(){ if{—}
     else{ =
           f();
           = }
}
```

After recursive call some work is there

Drawback: Difficult to write equivalent non-Recursive program

Adv: Not wasting stack space

Here, in recursion
- → calling = Preorder
- → execution = Postorder
- → From given non-tail recursive program, we're using stack DS to remove recursion
- → From any recursive program to eliminate recursion if we don't use stack, then that recursive program is mostly tail recursive program.

4) Nested recursion:-

Ex: $A(m,n) = \begin{cases} n+1 & ; \text{if } m = 0 \\ A(m-1,n) & ; \text{if } n = 0 \\ A(m-1, A(m,n-1)) & ; \text{o/w} \end{cases}$

$A(1,5) = A(0, A(1,4))$ 5 → 6

$A(0, A(1,3))$ 4 → 5

$A(0, A(1,2))$ 3 → 4

$A(0, A(1,1))$ 2 → 3

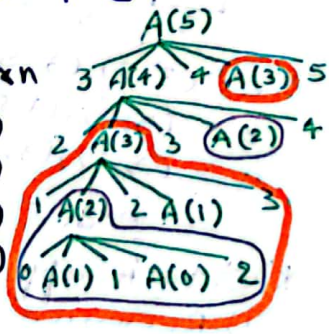
$A(0, A(1,0))$ 1 → 2

$A(0,0)$ → 1

1

Ex: A(n)

```
{ if(n ≤ 1) return
  else{ pf(n-2)
        A(n-1)
        pf(n-1)
        A(n-2)
        pf(n)
}
```



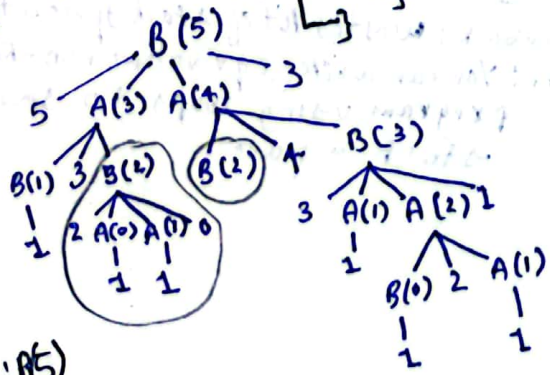
i/p: 5

o/p: 3 2 1 0 1 2 2 3 3 0 1 2 4 + + 1 0 1 2 2 3 5

3) Indirect recursion:

```
Ex: A(n)
{if(n≤1) ret. 1
 else{ B(n-2);
       pf(n);
       B(n-1);
}
```

```
B(n)
{;if(n≤1) ret. 1
 else{
       pf(n);
       A(n-2);
       A(n-1);
       pf(n-2);
}
```



i/p: B(5)

ip: 5 1 3 2 1 1 0 2 1 1 0 + 3 1 1 2 1 1 3

*) Infix to postfix:

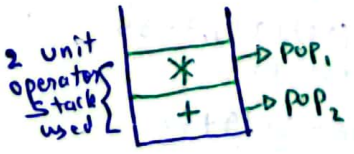
(Infix to prefix also here)

•) Points:
1. It uses operator stack
2. It takes O(n) [EC]

•) Rules:
1. Operand coming, direct print
2. Operator coming, it must go inside stack
3. If new operator have higher priority than TOS then direct push. (Ex: 1)
4. If new operator have equal priority than TOS then :-
   i) If L to R associativity
   ⇒ pop old operator first, then push new operator. [Ex: 2 (−,+)]
   ii) If R to L associativity
   ⇒ just push the new operator [Ex: 3 ($=_1$, $=_2$)]
5. If new operator have less priority than TOS then pop old operator first, then push new operator. [Ex: 3]
6. '(' will begin a new stack within old stack and all pops happen as we reach ')'.

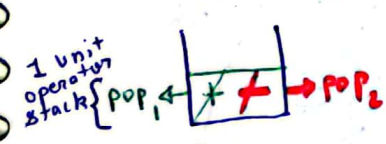
•) Examples:

1) IF: a * b * c

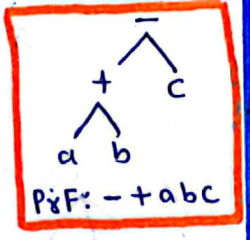


PoF: a b c * +

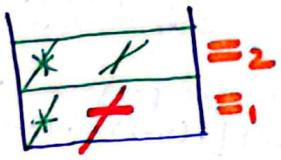
2) IF: a*b+c



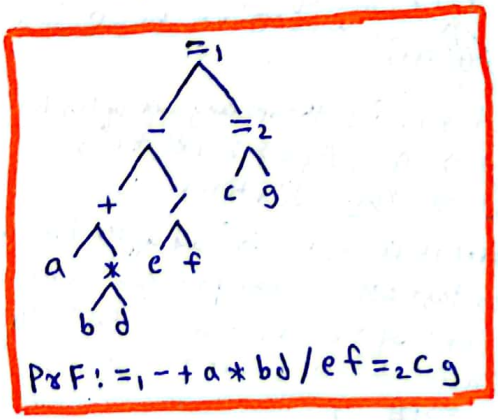
PoF: ab+c-



3) IF: a+b*d/e-f=1c=2g



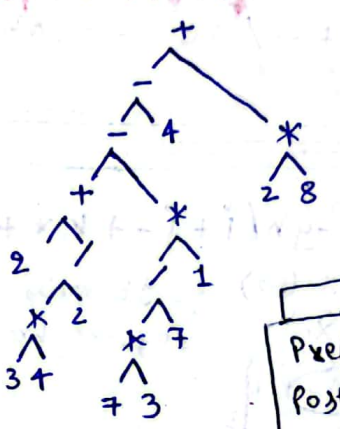
PoF: abd*+ef/-cg=2=1



•) Example:

IF: 2+3*4/2-7*3/7*1-4+2*8

↳ PoF: 2 3 4 * 2 / + 7 3 * 7 / 1 * - 4 + 2 8 * +



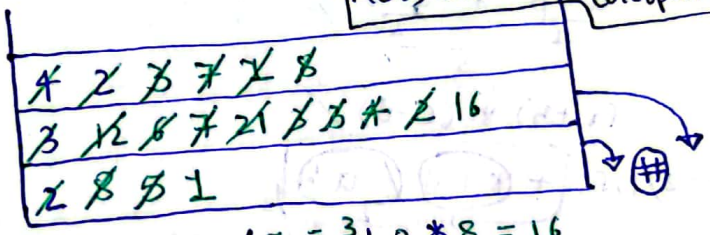
Note:

Prefix: <root><lc><rc>

Postfix: <lc><rc><root>

Infix: <lc><root><rc>

Here, root is operator & lc, rc are operands



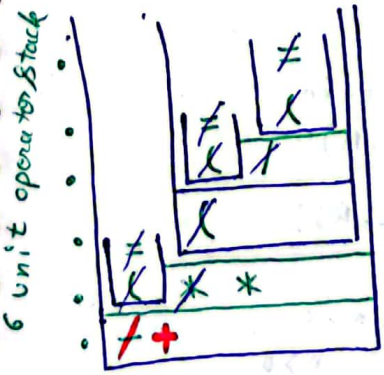
3 * 4 = 12
12 / 2 = 6
2 + 6 = 8
7 * 3 = 21

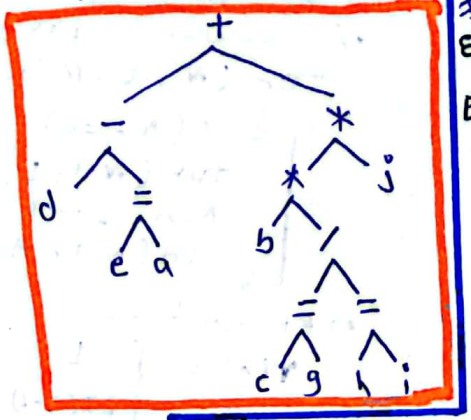
21 / 7 = 3
3 * 1 = 3
8 - 3 = 5
5 - 4 = 1

2 * 8 = 16
1 + 16 = 17

Ans: 17

4) IF: a+(c-a)*b*((c-a)+i)/i*j





PrF: de a = - bc g = hi = /*j*+

PoF: + -d=ea**b/=cg=hij

Read Rules once again

*) Postfix evaluation:

•) Points:

1. It uses operand stack
2. It takes O(n) [EC]

•) Rules:

1. if (operand) then push
2. if (operator) then

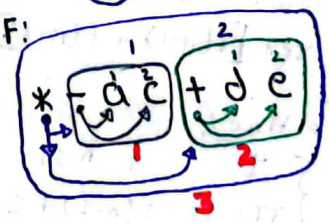
opnd2 = pop()
opnd1 = pop()
result = opnd1 (operator) opnd2
push(result)

*) Prefix to Postfix:

*Approach: Left to Right

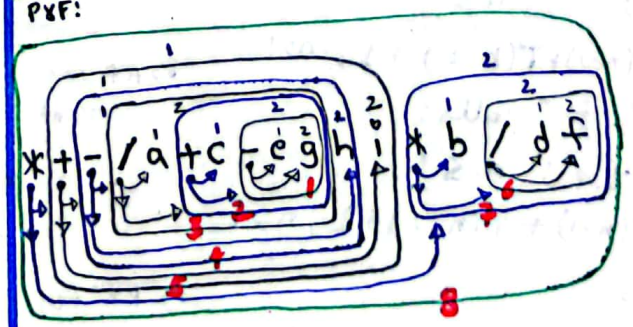
Ex-1) PrF: +ab PoF: ab+

Ex-2) PrF:



PoF: ac-de+*

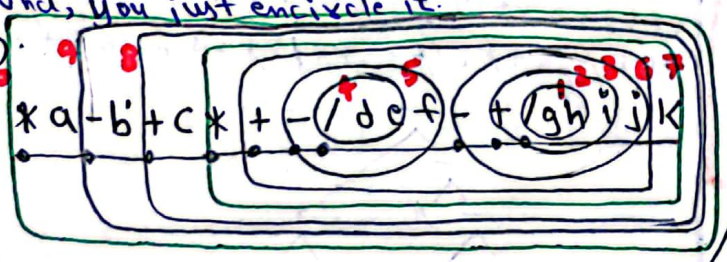
Ex-3) PrF:



PoF: aceg-+/h-i+ bdf/**

#Approach: Scan RtoL and wherever operator found, you just encircle it.

Ex-1)



PrF: * a - b + c * + - / d e f - + g h i j k

PoF: a b c d e / f - g h / i + j - + k * + - *

*) Postfix to infix (or) Prefix to infix :- It may require brackets

Ex-1) PoF: (a b +)$_1$ (c d -)$_2$ *$_3$

IF: a + b * c - d ✗ Brackets needed

(a + b) * (c - d) ✓

Ex-2) PrF: + * c d / a b

(c * d) + (a / b) ✓ Brackets not needed

*) Fibonacci Series:

•) Recursive program:

```
f(n) { if (n<=1) ret.(n)
       ret.[f(n-1) + f(n-2)]
```

•) RR for time:

$$T(n) = \begin{cases} 1 & ; n \le 1 \\ T(n-1) + T(n-2) + C & ; n > 1 \end{cases}$$

complexity: $O(2^n)$

•) RR for no. of additions:

$$T(n) = \begin{cases} 0 & ; n \le 1 \\ T(n-1) + T(n-2) + 1 & ; n > 1 \end{cases}$$

•) RR for no. of fx.n calls:

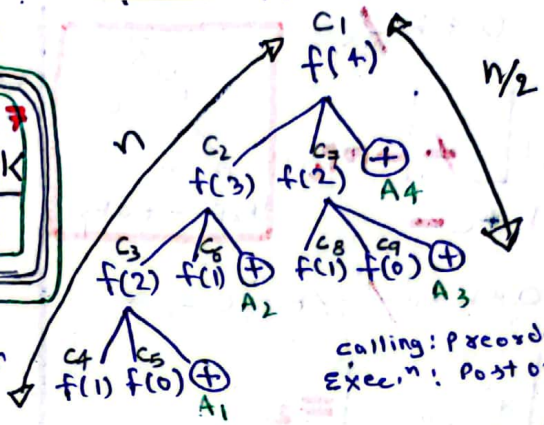
$$T(n) = \begin{cases} 1 & ; n \le 1 \\ T(n-1) + T(n-2) + 1 & ; n > 1 \end{cases}$$

•) Questions:

Q-1) In f(n) after how many fx.n calls first add.n took place?

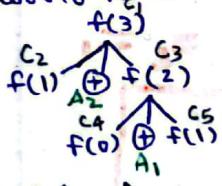
f(4) => 4+1

so f(n) => n+1



calling: Preorder
Exec.n: Post order

Note: If in recursive program, return would've been ret[f(n-2) + f(n-1)] then tree would've been different. And answer would've been 2n-1



Q:2) In f(n), is there any add.n after last fx.n call?

f(4) => A3, A4 => 2 additions

f(n) => n/2 additions

Q-3) In f(n) is there any fx.n call after last add.n?

No, after last add.n entire program is over.

Q-4) In f(4) after how many fx.n calls 4th add.n took place?

=> 9

*) Towers of Hanoi:

•) Recursive program:

```
TOH (N, L, M, R)
{ if (N==0) return;
  TOH(N-1, L, R, M);
  Move: L -> R
  TOH(N-1, M, L, R);
```

•) RR for time:

$$T(n) = \begin{cases} 1 & ; n = 0 \\ 2T(n-1) + C & ; n > 0 \end{cases}$$

complexity: $\theta(2^n)$

•) RR for moves:

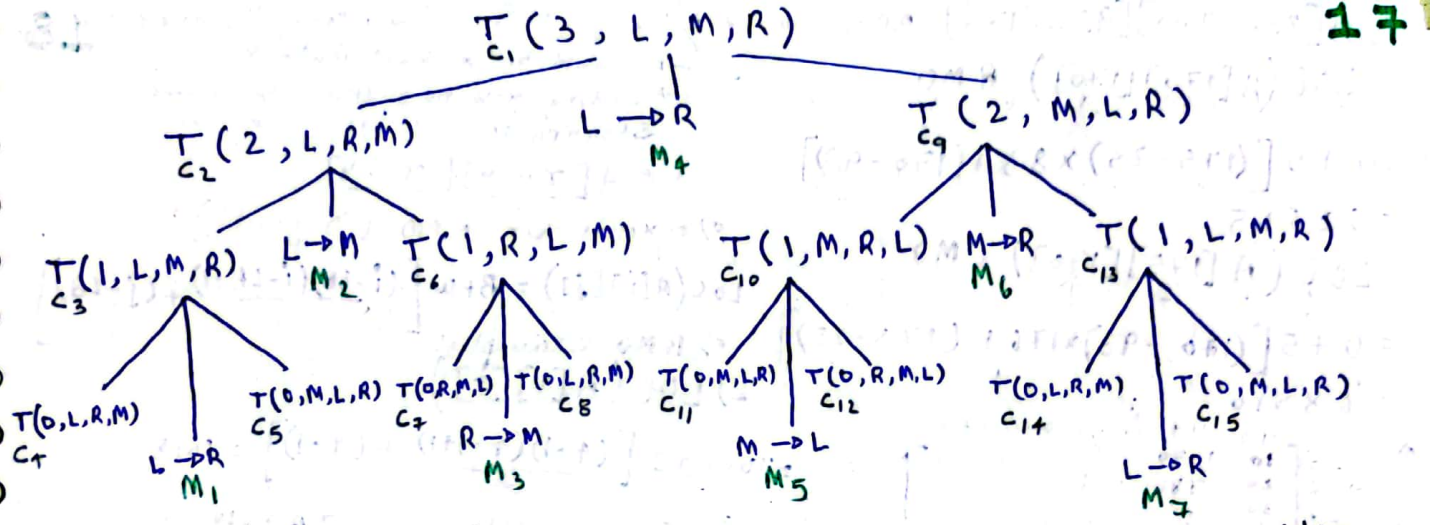
$$T(n) = \begin{cases} 0 & ; n = 0 \\ 1 & ; n = 1 \\ 2T(n-1) + 1 & ; n > 1 \end{cases}$$

$T(n) = 2^n - 1$

•) RR for no. of fx.n calls:

$$T(n) = \begin{cases} 1 & ; n = 0 \\ 2T(n-1) + 1 & ; n > 0 \end{cases}$$

$T(n) = 2^{n+1} - 1$



•) Questions:

Q-1) In TOH(n) after how many fx'n calls 1st move took place?
n+1 (After C_4 there's M_1)

Q-2) In TOH(n) is there any fx'n call after last move?
Yes (After M_7 there's C_{15})

Q-3) In TOH(n) is there any move after last fx'n call?
NO (After C_{15} no move while backtracking)

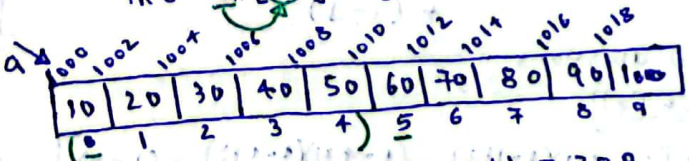
Q-4) In TOH(3) after how many fx'n calls there's a first move M→R?
12 (After C_{12} there's M_6)

#) Arrays:

*) 1D Array:

•) Declaration: a is an array which has 10 slots, where everyone is int

```
int a[10] = {10, 20, ... 100};
```



Array size = 1019 − 1000 + 1 = 208
Array indexes or elements = 9 − 0 + 1 = 10

a => base address
print(a) => 1000
*1000 => 10
*1018 => 100
& a[5] => 1010

* → Dereference
& → Reference

•) Examples:

1) LOC(a[5]) = 1000 + 2(5 − 0) = 1010
2) LOC(a[9]) = 1000 + 2(9 − 0) = 1018
3) LOC(a[0]) = 1000 + 2(0 − 0) = 1000
4) A[75...200] BA = 0 w = 10
   • LOC(A[150]) = 0 + 10(150 − 75) = 750
   • Array elements = 200 − 75 + 1 = 126
5) A[1...10] BA = 1000 w = 2
   LOC(A[9]) = 1000 + (9 − 1)2 = 1016

Note: By default array index starts from 0, bec. it enables us not to calculate the offset value (i.e. subtraction is always 0, so no need to subtract).

•) Formula: A[LB ... UB]
Array elements = UB − LB + 1
LOC(A[K]) = BA + w[K − LB]

*) 2D Array:

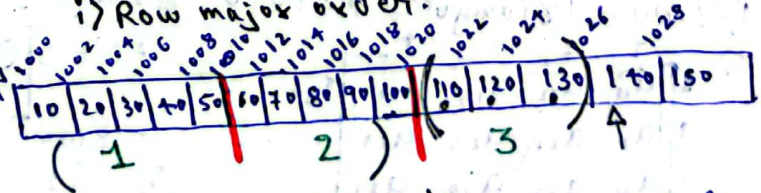
•) Declaration:
int a[1...3, 1...5] = {10, 20, ... 150};
Rows = 3 − 1 + 1 = 3 Col = 5 − 1 + 1 = 5

•) Representation for user:

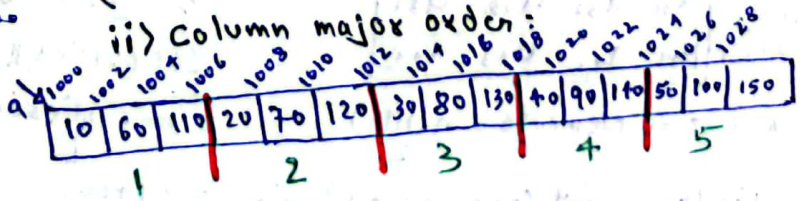
a	0	1	2	3	4
0	10	20	30	40	50
1	60	70	80	90	100
2	110	120	130	140	150

•) Representation for computer:

i) Row major order:



ii) Column major order:



•) Examples:

1) LOC(a[3][4]); RMO
= 1000 + 2[(3 − 1) × 5 + (4 − 1)] = 1026

2) A[25...200][93...175] BA=0 w=5B

LOC(A[175][170]) RMO

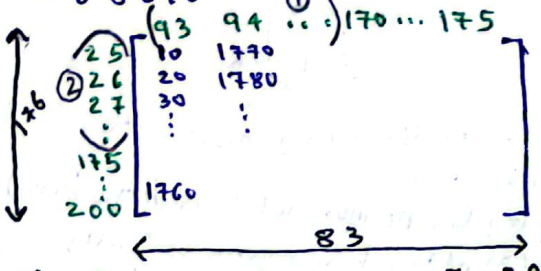
=0 + 5[(175-25)×83 + (170-93)]

= 62635

LOC(A[175][170]) CMO

= 0 + 5[(170-93)×176 + (175-25)]

= 68510



3) A[-75...75][-85...-25] BA=1000 w=10

151 61

LOC(A[73][-30])

i) RMO

= 1000 + 10[(73+75)×61 + (-30+85)]

= 91830

ii) CMO

= 1000 + 10[(-30+85)×151 + (73+75)]

= 85530

• Formula

A[L1...U1][L2...U2]

RMO:

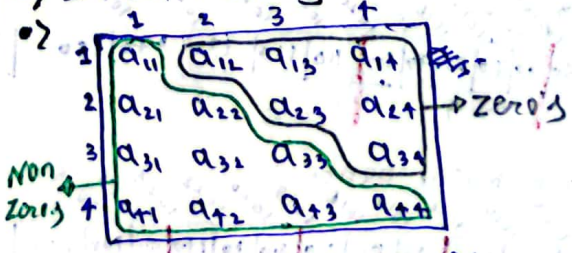
LOC[A[i][j]] = B + w[(U2-L2+1)×(i-L1) + (j-L2)]

CMO:

= B + w[(U1-L1+1)(j-L2) + (i-L1)]

*) Lower Triangular Matrix:

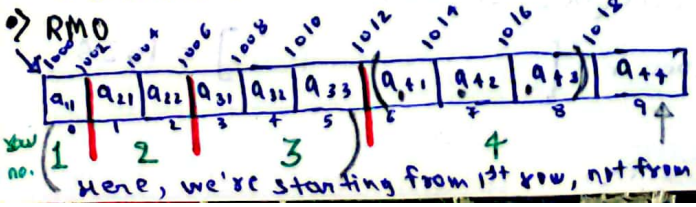
•)



Non-zero elements = $\frac{n(n+1)}{2}$

This lower part (non-zeros) only is stored in memory.

•) RMO



Here, we're starting from 1st row, not from 0th row, so LB = 1

•) Properties of lower triangular matrix:-

→ Should be square matrix

→ Ending row no.s should be same

Starting " " " " "

i.e. A[x...y][x...y]

•) Formula for RMO (Default):

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

•) RMO examples:

1) LOC(A[4][4])

$= 1000 + 2\left[\frac{(4-1)(4-1+1)}{2} + (4-1)\right] = 1018$

2) A[75...200][75...200] BA=1000 w=10

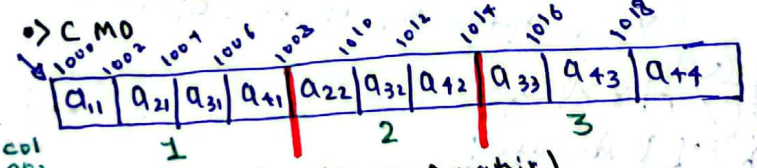
126 126

LOC(A[150][100]) ← possible ∵ i>j

$= 1000 + 10\left[\frac{(150-75)(150-75+1)}{2} + (100-75)\right]$

= 29750

•) CMO



•) CMO formula: (Lower ∆ matrix)

$Loc(A[i][j]) = B + w\left[\frac{(UB-LB+1)(UB-LB+2)}{2} - \frac{(UB-j+1)(UB-j+2)}{2} + (i-j)\right]$

•) CMO questions:

1) LOC(A[4][4])

$= 1000 + 2\left[\frac{(4-1+1)(4-1+2)}{2} - \frac{(4-4+1)(4-4+2)}{2} + (4-4)\right]$

= 1018

2) A[75...150][75...150] BA=0 w=5

76 76

LOC(A[120][100])

$= 0 + 5\left[\frac{(150-75+1)(150-75+2)}{2} - \frac{(150-100+1)(150-100+2)}{2} + (120-100)\right]$

= 8100

3) A[-25...50][-25...50] BA=1000 w=10

76 76

LOC(A[43][38])

$= 1000 + 10\left[\frac{(50+25+1)(50+25+2)}{2} - \frac{(50-38+1)(50-38+2)}{2} + (43-38)\right]$

= 29400

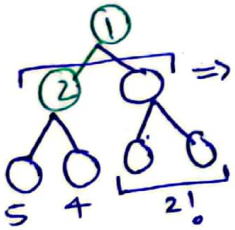
*) 3D Array:

#) Some important notes:

*) Binary Heap

Q:) No. of possible minheaps containing each value from [1,2,3,4,5,6,7] exactly once is?

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



$\Rightarrow 2!$

$\Rightarrow 80$

Q:) Binary minheap containing 1023 elements (array repr.^{n})
Min.^{m} comparisons required to find max.^{m}?
max.^{m} would be there at any of the leaf
so, 1. leaves $= \lceil n/2 \rceil = \lceil \frac{1023}{2} \rceil = 512$

2. comparisons $\Rightarrow (512-1) = 511$
[$\because$ n-1 comparisons for 'n' nodes]

*) BST:-

Q:) when searching for key 60 in BST with nodes 10, 20, 40, 50, 70, 80, 90
Different orders possible in which these key values can occur on the search path?

$$= \frac{\text{Permutation of 7 no.s}}{\text{permutations of no.s smaller than 60} \times \text{perm.}^n\text{s of no.s greater than 60}}$$

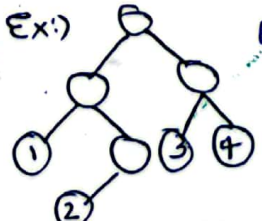
$$= \frac{7!}{4! \times 3!} = 35$$

*) Random:-

- For Binary Tree: max.^{m} height = n-1
  min.^{m} height = $\lfloor \lg_2 n \rfloor$

- For Binary Tree:
  → External path length = Σ P_{lw}
  where, P_{lw} is path length of external node (leaf) no. w

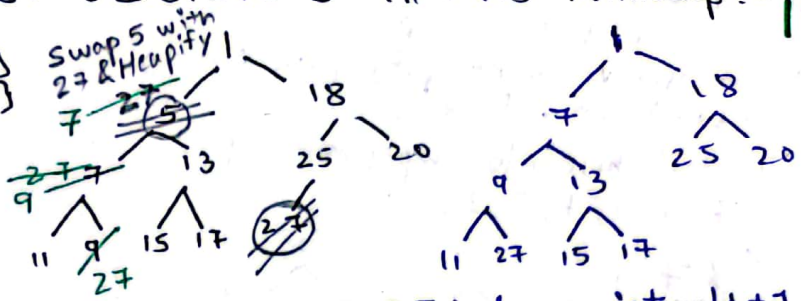
Ex:)



$EPL = \Sigma P_{lw} = 2+3+2+2$

→ Weighted EPL = $\Sigma P_{lw} * Weight_w$

- Delete 5 in this minheap:-



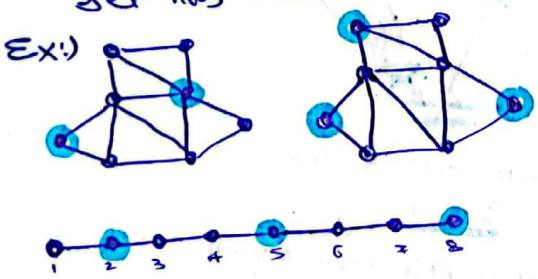
- In Strict BT & BT: leaves = internals + 1
  In CBT/ACBT: leaves $= \lceil \frac{n}{2} \rceil$; n = Total no. of nodes

- For AVL to attain height 'h'
  min.^{m} nodes $= 2^h$

- Polish = Postfix
  Reverse Polish = Prefix

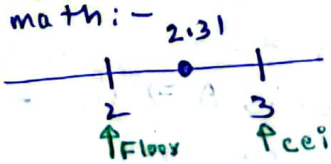
- For a graph: Maximal independent set is set S of vertices where no two vertices in S are neighbours.
  → S is maximal if it is impossible to add another vertex & stay independent
  → S is maximum if no other independent set has more vertices.

Ex:)



- For hash table:-
  Constructing hash table with linear probing $\Rightarrow O(n^2)$ [WC] bec. all elements hashed to same index,
  do no. of probes: 0, 1, 2, ... n-1 $\Rightarrow \frac{n(n-1)}{2}$ probes

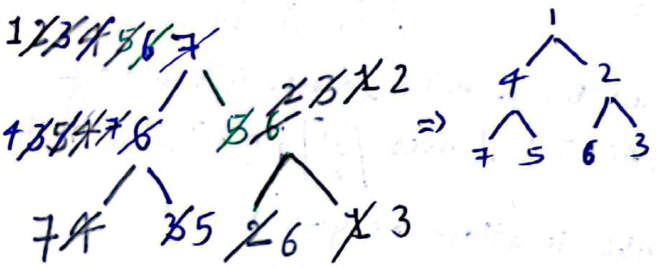
- In math:-



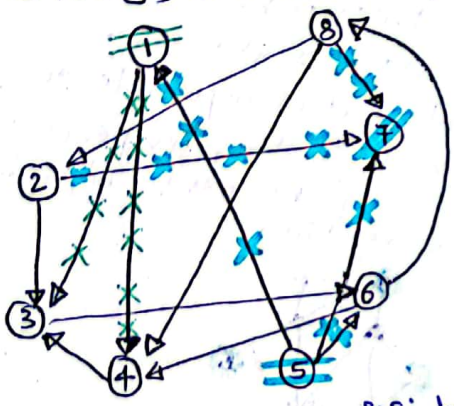
- In Stack:-
  Operator Stack $\Rightarrow$ Infix to postfix, Infix to prefix
  Operand Stack $\Rightarrow$ Postfix evaluation, Prefix evaluation
  Both operator & operand Stack $\Rightarrow$ Prefix to infix, Prefix to postfix, Postfix to infix, Postfix to prefix

- If asked to insert elements into an empty minheap :-
  Then you've to heapify after every insertion, like :-

Ex:) Insert these into empty minheap
7, 6, 5, 4, 3, 2, 1



- For n-ary tree :-
  $l = (n-1)I + 1$; $l + I = N$
- To find no. of nodes in maximal strongly connected component of G :-



Step-1 : Remove Source & Sink node
i.e. 5 & 7

Step-2 : Remove new source & sink
i.e. 1 & none

now, Remaining graph is maximal strongly connected.

- In G ; edges ↑ ⇒ # Topo. orders ↓
- In CBT ; # Leaves = 2^h
- If adjacency matrix is A ⇒ # paths of size 1
  then A^3 ⇒ # path of size 3