ఓంసాయిరాం

ఓంశ్రీవేంకటేశాయ నమః

INDEX

12 BIT0077

NAME: T. Prashanth Reddy STD.: ______ SEC.: ______ ROLL NO.: ______ SUB.: ______

S. No.	Date	Title	Page No.	Teacher's Sign / Remarks

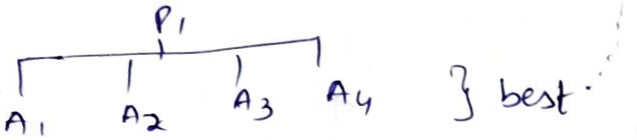
Algorithms & Data Structures

Problem

↓

Program in 'C'

i) Time
ii) Memory



→ Design & Analysis of algorithms :-

Asymptotic Notation :-

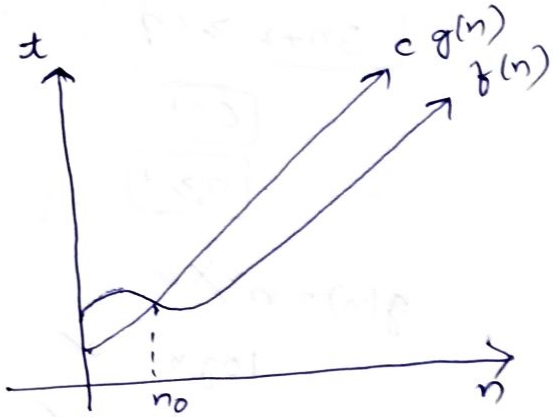
Big (oh) :-

$$f(n) \leq c \cdot g(n)$$

$n > n_0$

$c > 0$; $n_0 \geq 1$

$$\boxed{f(n) = O(g(n))}$$



$c \cdot g(n)$ is the tightest upper bound of $f(n)$.

Ex :- $f(n) = 3n+2$; $g(n) = n$

$$f(n) = O(g(n))$$

$f(n) \leq c\,g(n)$, $c > 0$, $n_0 \geq 1$

$3n+2 \leq cn$

if $\boxed{c = 4}$

$3n+2 \leq 4n$

$\boxed{n \geq 2}$

$g(n) = n^2$
n^3
n^4
$\vdots$
n^n
2^n

but, $g(n) = n$, consider tighest upper bound

Big Omega (Ω) :-

$f(n) = 3n+2$, $g(n) = n$

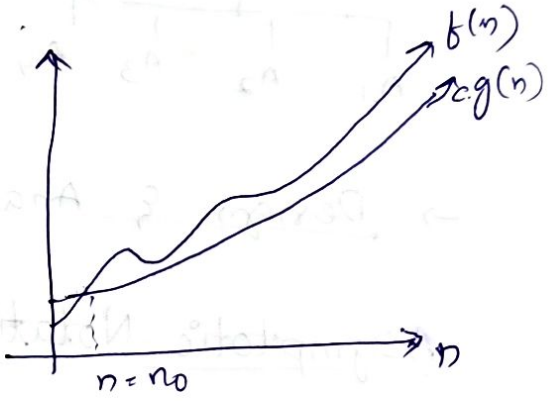
$f(n) = \Omega(g(n))$

$f(n) \geq c \cdot g(n)$

$3n+2 \geq cn$

$c = 1$

$n_0 \geq 1$



$f(n) \geq c \cdot g(n)$:

$n \geq n_0$

$c > 0$; $n_0 \geq 1$

$g(n) = n^2$ ✗

$= \log n$ ✓

$= \log \log n$ ✓

$g(n) = n$

↳ tighest lower bound

Big Theta (Θ) :-

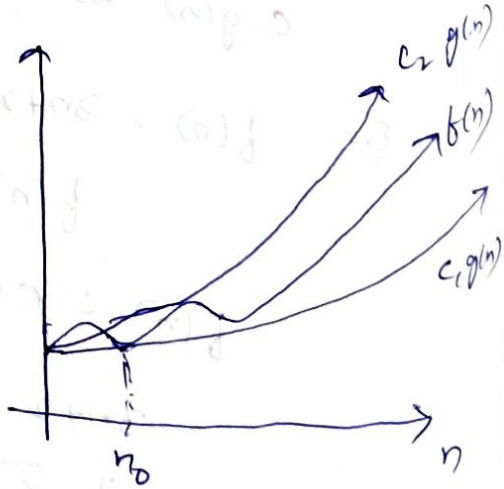
$f(n) = \Theta(g(n))$

$c_1 g(n) \leq f(n) \leq c_2 \cdot g(n)$

$c_1, c_2 > 0$

$n \geq n_0$

$n_0 \geq 1$



$f(n) = 3n+2 \quad g(n) = n$

$f(n) \leq c_2 g(n)$

$3n+2 \leq 4(n) \quad ; \underline{n_0 \geq 1}$

$f(n) \geq c_1 g(n)$

$3n+2 \geq n \Rightarrow \underline{n_0 \geq 1}$

$\boxed{c_1 = 1} \quad \boxed{c_2 = 4}$

→ O, Ω, Θ → used when both best & worst cases are same

O — worst case; Ω — best case; Θ — Average case

→ Amortised analysis - Average analysis

Complexity Analysis of Iterative Programs:

Algorithms

- Iterative
- Recursive

Iterative

```
A()
{
for i=0 to n;
   max(a,b);
}
```

power

$\Leftrightarrow$ Interchangable

Recursive

```
A(n)
{
if()
A(n/2)
}
```

Ex:- 1)

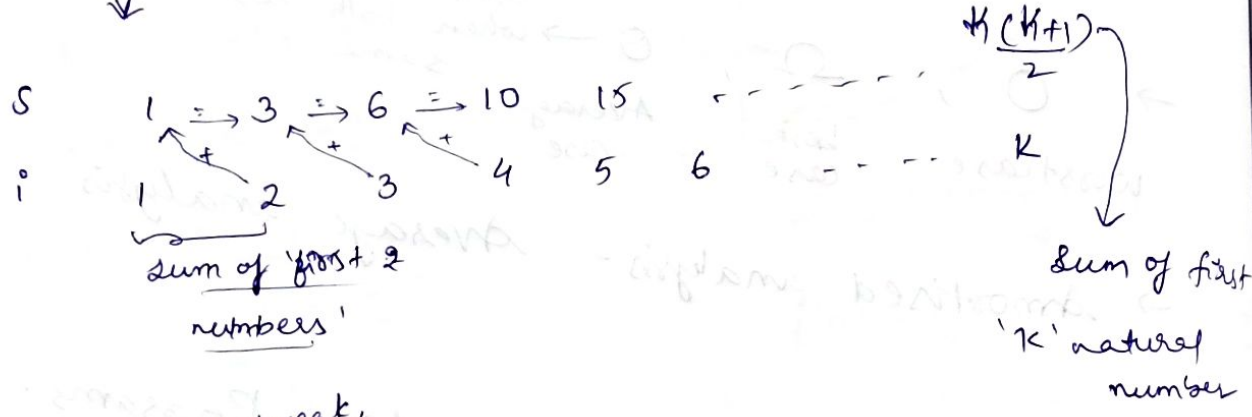
```
A()
{
  for (i=1 to n)
    Pf("Ravi");
}
```

↳ 'n' times it gets printed

$O(n)$

2)

```
A()
{
  i=1 ; s=1;
  while (s<=n)
  {
    i++;
    s=s+i;
    Pf("Ravi");
  }
}
```



for loop to break,

$$\frac{k(k+1)}{2} \gg n$$

$$\frac{k^2+k}{2} > n$$

$$k = O(\sqrt{n})$$

∴ time complexity, in the order of $\sqrt{n}$

3)

```
A()
{
  for (i=1 ; i*i<=n ; i++)
    Pf("Ravi");
}
```

$\Rightarrow$ $i = \sqrt{n}$

$O(\sqrt{n})$

A() {

4. for (i=1; i<=n; i++) {

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

for (k=1; k<=100; k++) {

pf("Ravi");

}

}

}

i=1	i=2	i=3	i=4	...	i=n
j=1	j=2	j=3	j=4	...	j=n
k=100*1	k=100*2	k=100*3	k=4*100	...	k=n*100
~~printf 100~~	2*100				

$100 + 200 + 300 + 400 + \cdots 100n$

$= 100(1+2+3+\cdots n) = 100\left(\frac{n(n+1)}{2}\right)$

$= 100\left(\frac{n^2+n}{2}\right)$

$\rightarrow O(n^2)$

5) A() {

for (i=0; i<=n; i++) {

for (j=0; j<=i^2; j++) {

for (k=1; k<=n/2; k++) {

pf("Ravi");

}

}

}

i=1	i=2	i=3	i=4	...	i=n
j=1	j=2^2	j=3^2	j=4^2		j=n^2
k=n/2	k=2^2*n/2	k=3^2*n/2	k=4^2*n/2		k=n^2*n/2

$$\frac{n}{2}*1 + \frac{n}{2}*2^2 + \frac{n}{2}*3^2 + \dots \frac{n}{2}*n^2$$

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

$$= \frac{n}{2}\left(\frac{n(n+1)(2n+1)}{6}\right) = O(n^4)$$

Ex:

→ While finding time complexity, concentrate on the incrementing relation value, i and input size, n.
(Take this variable as 'k' when calculating to avoid confusion)

→ If the function has multi-loop iterations, then unroll the loops & check for time complexity. and dependency b/w loops

Ex:

```
A() {
for(i=0 ; i<n ; i=i*2) {
    Pf("Ravi");
}
```

i = 1, 2, 2^2, 2^3 ... 2^k

A(): 2^0, 2^1, 2^2, 2^3 ... 2^k

$2^k = n$ $\quad k = \log_2 n$

NOTE:-

if "i" is incrementing by the ratio of '2' then,

$k = \log_2 n$

if i.e inc. in terms of $k = \log_3 n$

$i = i * 4$ $\quad k = \log_4 n$

$\vdots$

$i = i * n$ $\quad k = \log_n n$

Ex: A()

{ for (i=n/2 ; i<=n ; i++) { $\longrightarrow n/2$

for (j=1 ; j<=n/2 ; j++) { $\longrightarrow n/2$

for (k=1; k<=n; k=k*2) { $\longrightarrow \log n$

Pf ("Ravi");

} } }

$\Rightarrow \frac{n}{2} * \frac{n}{2} * \log n$

$= O(n^2 \log n)$

Ex:- A() {

for(i=n/2; i<=n; i++) { $\longrightarrow n/2$

for(j=1; j<=n; j=j*2) { $\longrightarrow \log_2 n$

for(k=1; k<n; k=k*2) { $\longrightarrow \log_2 n$

Pf("Ravi");

} } }

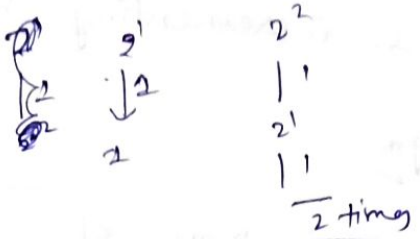
$\frac{n}{2} * \log_2 n * \log_2 n$

$\rightarrow O(n(\log_2 n)^2)$

Ex:-

assume $n \geq 2$

```
A() {
while(n>1) {
  n = n/2 ; }
}
```



$n = 2^k$

$k = \log_2 n$

→ It decrements in power of '2' then it has logarithmic time complexity

Ex:-

```
A() {
for(i=1;i<=n;i++) {
    for(j=1;j<=n;j++) {
      Pf("Ravi");
}}
```

i=1	i=2	i=3	...	i=k	...	i=n
j=1 to n	j=1 to n	j=1 to n		j=1 to n		j=1 to n
n times	n/2	n/3		n/k		n/n

$$n + \frac{n}{2} + \frac{n}{3} + \cdots \frac{n}{k} \cdots n/n$$

$$= n\left(1 + \frac{1}{2} + \frac{1}{3} + \cdots \frac{1}{n}\right)$$

$$= n(\log n)$$

$$O(n \log n)$$

Q

```
A() {
int n = 2^(2^k);
for (i=1; i<=n; i++) {
    j=2;
    while (j <= n) {
        j = j^2;
        Pf("Ravi"); } }
}
```

k=1	k=2	k=3
n=4	n=16	~~k=3~~ $n = 2^{2^3}$
j = 2, 4	j = 2, 4, 16	$j = 2, 2^2, 2^4, 2^8$
n*2 times	n*3 times	n*4 times

~~n*(k+1)~~

$n * (k+1) \rightarrow (1)$

$n = 2^{2^k} \Rightarrow \log_2 n = 2^k$

$\log \log n = k$

Sub in (1)

$\Rightarrow n(\log \log n + 1)$

$\Rightarrow O(n \log(\log n))$

NOTE:-

if incrementing by powers of '2' ⇒ $\log_2 n$

if incrementing by powers of (2^{2^k}) ⇒ $\log\log_2 n$

Time Analysis of Recursive Program :-

Recursive:

```
A(n) {
if ( ) → constant time for "if" condn
return ( A(n/2) + A(n/2) );
}
```

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

Ex :-

```
A(n) {
  if (n==1) {
    return 1; }
  else {
    return A(n-1);
  }
}
```

→ Anchor condition

$T(n) = 1 + T(n-1) ; n>1$

$= 1 ; n=1$

Recursive Equation:

$$T(n) = 1 + T(n-1) \quad \rightarrow (1)$$

Back Substitution:

~~T(n)~~ =

$T(n-1) = 1 + T(n-2) \rightarrow (2)$

$T(n-2) = 1 + T(n-3) \rightarrow (3)$

$T(n-3) = 1 + T(n-4)$

:

$T(n) = 1 + T(n-1) \rightarrow$ [sub (2) in (1)]

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

$= 2 + T(n-2)$ [sub (3) in]

$= 2 + 1 + T(n-3)$

$= 3 + T(n-3)$

:

$= k + T(n-k)$

:

$= (n-1) + T(n-(n-1))$

$= (n-1) + T(1)$

$= n - 1 + 1$

$T(n) = n$

$O(n)$

[~~for~~ If According to anchor condn, recursion stops at $n=1$; $\Rightarrow T(1)$

$\Rightarrow n - k = 1$

$k = n-1$

[If $T(1)$ return = 1;

Ex2:- Given,

$$T(n) = n + T(n-1) \quad ; \quad n > 1$$

$$1 \quad ; \quad n = 1$$

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

$$T(n-2) = (n-2) + T(n-3)$$

$$T(n-3) = (n-3) + T(n-4)$$

$$\vdots$$

so on.

Apply Back substitution:-

$$T(n) = n + T(n-1)$$

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

$$= n + (n-1) + (n-2) + T(n-3)$$

$$= n + (n-1) + (n-2) + \dots + (n-k) + T(n-(k+1))$$

for, Anchor condition,

$$= n-(k+1) = 1$$

$$= n-k-1 = 1$$

$$= \boxed{k = n-2}$$

$$\Rightarrow n + (n-1) + (n-2) + \dots + (n-(n-2)) + T(1)$$

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

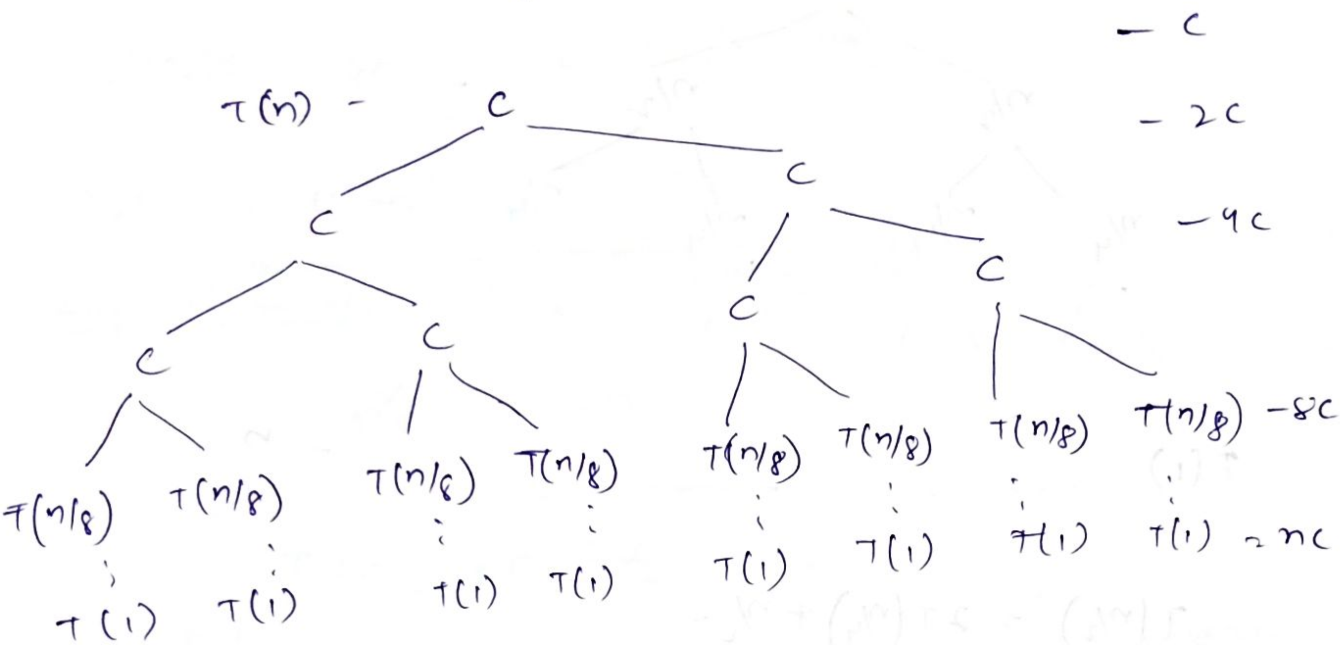
$$= \frac{n(n+1)}{2}$$

$$\rightarrow \underline{O(n^2)}$$

Recursion Tree Method :-

1) $T(n) = 2T(n/2) + c \quad ; n > 1$

$\quad\quad\quad\quad 1 \quad\quad ; n = 1$



$T(1) = T(n/n)$ $\quad\quad$ assume $n = 2^k$

$c + 2c + 4c + \dots + nc$

$c(1 + 2 + 4 + \dots + 2^k)$

$= c\left(\frac{1(2^{k+1} - 1)}{(2-1)}\right) = c(2^{k+1} - 1)$

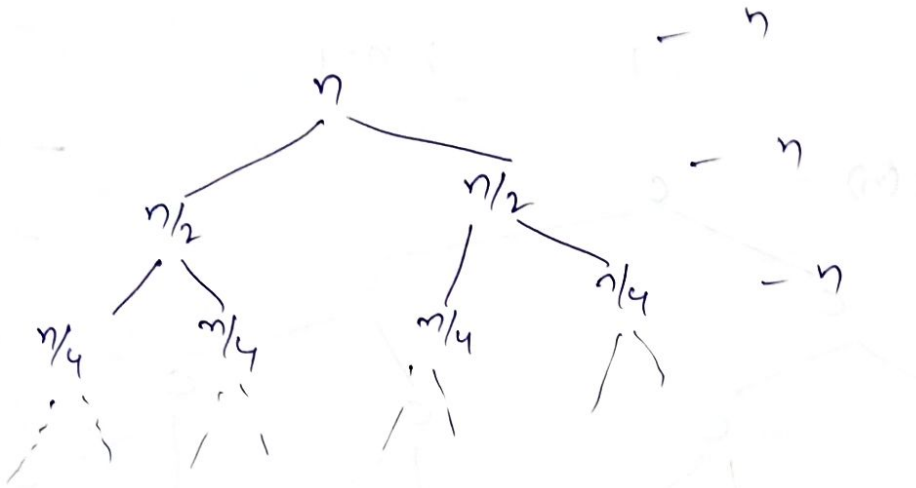
$= c(2n - 1)$

$= \boxed{O(n)}$

2)

$$T(n) = n + 2T(n/2) \quad ; n > 1$$

$$= 1 \quad ; n = 1$$



$$T(n/2) = 2T(n/4) + n/2$$

The tree is shrink at,

$$\frac{n}{2^0}, \frac{n}{2^1}, \frac{n}{2^2}, \frac{n}{2^3} \cdots \frac{n}{2^k} \quad \text{levels} \quad (1)$$

$$n = 2^k$$

$$\boxed{k = \log n}$$

$$n(\log n + 1)$$

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

Master's Theorem :-

To apply master's theorem we should know how to compare '2' functions. time complexities & decide which one is larger.

Ex: n^2 , n^3

Step I) Cancel the like terms first

$$n^2 \cdot = n^2 \times n$$

$$1 = n$$

Then compare

Another method

Step 2:

Using logarithms also we can compare time complexities very easily :-

i.e, Apply log on both the sides

$f(n)$	$g(n)$
1	2
3	2
4	1

↓

both the functions, asymptotically same.

if we have constants

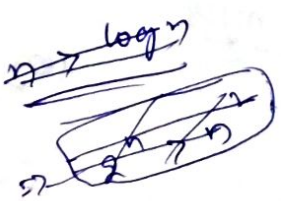
Ex 2:

2^n n^2

Apply log on B.S

$n \log 2 = 2 \log n$

$= n \cdot 2 \log 2$

$\Rightarrow$ 

Substitute large values for 'n' and then (more than 2, 3)

Conclude

$$n \log_2 2 = 2 \log_2 n$$

$$\therefore n = 2 \log_2 n$$

if $n = 128$

$$2^{128} = 2 \log_2 2^{128}$$

$$2^{128} > 256$$

$$\Rightarrow 2^n > n^2$$

Ex 3:-

$$3^n \; ; \; 2^n$$

Apply log on B.S

$$\log 3 = \log 2$$

$$\log 3 = \log 2 \Rightarrow 3^n > 2^n$$

Ex 4:

$$n^2 \; , \; n \log n$$

$$n \times n \quad n \log n \quad n > \log n$$

Ex 5:

$$n \; , \; (\log n)^{100}$$

Apply log on B.S

$$\log n = 100 \log \log n$$

if $n = 128$

$$\log 128 = 100 \cdot \log \log_2 2^{128}$$

$$= 100 \times 7$$

128

Here, ① $(\log n)^{100} > n$.

But, for 2^{2024}

$1024 = 100 \times 10$

$1024 = 1000$

② $n > (\log n)^{100}$

We have to consider for greatest values. So

$n > (\log n)^{100}$

Ex:1

$n^{\log n}$; $n \log n$

Apply log on B.S

$\log n \log n$; $\log n + \log \log n$

(substitute values & check)

$\Rightarrow n^{\log n} > n \log n$

Ex:2 $\sqrt{\log n}$; $\log \log n$

$\Rightarrow \sqrt{\log n} > \log \log n$

Ex:3 $n^{\sqrt{n}}$; $n^{\log n}$

$n^{\sqrt{n}} > n^{\log n}$

Ex:

$$f(n) = \begin{cases} n^3 & 0 < n < 10000 \\ n^2 & n \geq 10000 \end{cases}$$

$$g(n) : \begin{cases} n & 0 < n < 100 \\ n^3 & n > 100 \end{cases}$$

	0-99	100 - 9999	10000 -
$f(n)$	n^3	n^3	n^2
$g(n)$	n	n^3	n^3

for larger values, we have to consider, i.e,

$n > 10,000$

for $n > 10000$, $\boxed{f(n) = O(g(n))}$

$n_0 = 10,000$

Ex:

Compare following functions and order them in descending order of complexities

$f_1 = 2^n$; $f_2 = n^{3/2}$; $f_3 = n \log n$; $f_4 = n^{\log n}$

$$f_1 > f_4 > f_2 > f_3$$

$\log_{10}^{2} = 0.3010$ $\quad \log_{2}^{2} = 1$

$\Rightarrow \boxed{\log_{2}^{5} > \log_{10}^{5}}$

$\Rightarrow$ *) $\underline{\log^2 n \neq (\log n)^2}$

$\Rightarrow \underline{\log\log n \neq (\log n)(\log n)}$

Master's Theorem:-

$T(n) = a\,T(n/b) + \theta(n^k \log^p n)$

$a \geq 1$, $b > 1$, $k \geq 0$, p is any real number

1) if $a > b^k$, then $T(n) = \theta(n^{\log_b^a})$

2) if $a = b^k$, then

a) if $p > -1$, then, $T(n) = \theta(n^{\log_b^a} \log^{p+1} n)$

b) if $p = -1$, then $T(n) = \theta(n^{\log_b^a} \log\log n)$

c) if $p < -1$, then $T(n) = \theta(n^{\log_b^a})$

3) if $a < b^k$ then,

a) if $p \geq 0$, then $T(n) = \theta(n^k \log^p n)$

b) if $p < 0$, then $T(n) = O(n^k)$

Ex:

1) $T(n) = 3T(n/2) + n^2$

Sol:-

Here

$a = 3 \; ; \; b = 2 \; ; \; k = 2 \; ; \; p = 0$

$a \quad b^k$

$3 < 2^2$

$a < b^k$, $p \geq 0$ then $T(n) = \Theta(n^k \log^p n)$

$= \Theta(n^2 \log^0 n)$

$\underline{T(n) = \Theta(n^2)}$

2. $T(n) = 4T(n/2) + n^2$

3) $T(n) = T(n/2) + n^2$

4) $T(n) = (2^n) T(n/2) + n^2$ ✗

2^n → not a constant, hence master's theorem not applicable

"a"

5) $T(n) = 16T(n/4) + n$

6) $T(n) = 2T(n/2) + n\log n$

7) $T(n) = 2T(n/2) + n/\log n$

Sol:-

$T(n) = 2T(n/2) + n \cdot \log^{-1} n$

$a = 2 \; ; \; b = 2 \; ; \; k = 1 ; \; p = -1$

$a \quad b^k$

$2 = 2^1$, then $p = -1$, then

$$T(n) = \Theta(n^{\log_b^a} \cdot \log\log n)$$

$$= \Theta(n^{\log_2^2} \cdot \log\log n)$$

$$\underline{T(n) = \Theta(n \log\log n)}$$

8) $T(n) = 2T(n/4) + n^{0.51}$

$a = 2$; $b = 4$; $k = 0.51$; $p = 0$

$a \quad b^k$

$2 < 4^{0.51}$

$\Rightarrow$ $a < b^k$, $p = 0$, then $T(n) = \Theta(n^k \log^p n)$

$$T(n) = \Theta(n^{0.51} \log^0 n)$$

$$\boxed{T(n) = \Theta(n^{0.51})}$$

9) $T(n) = (0.5)\, T(n/2) + 1/n$

↳ $a = 0.5$ but $a \ngeq 1$. $\Rightarrow$ not valid

10) $T(n) = 6\, T(n/3) + n^2 \log n$

$a = 6$; $b = 3$; $k = 2$; $p = 1$

$a \quad b^k$

$6 < 3^2$

then $T(n) = \Theta(n^k \log^p n)$

$= \Theta(n^2 \log n)$

3) a)

11)

$T(n) = 64\,T(n/8) - n^2 \log n$

↳ negative ⇒ not valid

12)

$T(n) = 7\,T(n/3) + n^2$

$a = 7 ; \; b = 3 ; \; k = 2 ; \; p = 0$

$a \; b^k$

$7 < 3^2 , \; p = 0$

⇒ 3) a) $T(n) = \Theta(n^k \cdot \log^p n)$

$T(n) = \Theta(n^2 \log^0 n)$

$\boxed{T(n) = \Theta(n^2)}$

13)

$T(n) = 4\,T(n/2) + \log n$

$a = 4 ; \; b = 2 ; \; k = 0 ; \; p = 1$

$a \quad b^k$

$4 > 2^0$, then

$T(n) = \Theta\left(n^{\log_2^4}\right) = \Theta(n^2)$

14)

$T(n) = \sqrt{2}\,T(n/2) + \log n$

$a = \sqrt{2} ; \; b = 2 ; \; p = 1 ; \; k = 0$

$a > b^k$

$(\sqrt{2}) > (2)^0$, then

$T(n) = \Theta(n^{\log_b^a}) = \Theta(n^{\log_2^{\sqrt{2}}})$

$\Rightarrow \Theta(\sqrt{n}) = \Theta\left(\frac{1}{n}^{\log_2^2}\right)$

15)

$T(n) = 2T(n/2) + \sqrt{n}$

$a = 2 ; \; b = 2 ; \; k = 1/2 ; \; p = 0$

$a \quad b^k$

$2 > \sqrt{2}$, then, $T(n) = \Theta\left(n^{\log_b^a}\right)$

$T(n) = \Theta\left(n^{\log_2^2}\right) = \Theta(n)$.

16)

$T(n) = 3T(n/2) + n$

$a = 3 ; \; b = 2 ; \; k = 1 ; \; p = 0$,

$a \quad b^k$

$3 > 2^1$, then $T(n) = \Theta\left(n^{\log_b^a}\right)$

$T(n) = \Theta\left(n^{\log_2^3}\right)$

17)

$T(n) = 3T(n/3) + \sqrt{n}$

$a = 3 ; \quad b = 3 ; \quad k = 1/2 ; \; p = 0$

$a \quad b^k$

$3 > \sqrt{3}$, then, $T(n) = \Theta\left(n^{\log_b^a}\right)$

$T(n) = \Theta\left(n^{\log_3^3}\right) = \Theta(n)$

11) $T(n) = 4T(n/2) + cn$

$a = 4 ; \quad b = 2 ; \quad k = 1 ; \; p = 0$

$a \quad b^k$

$4 > 2^1$, then

$T(n) = \Theta\left(n^{\log_2^4}\right) = \underline{\Theta(n^2)}$

(d)

$T(n) = 3T(n/4) + n\log n$

$a = 3 ; \quad b = 4 ; \quad k = 1 ; \quad p = 1$

$a \quad b^k$

$3 < 4^1 , \quad p \geq 0 , \text{ then}$

$T(n) = \theta(n^k \log^p n)$

$T(n) = \theta(n^1 \log^1 n)$

$\underline{T(n) = \theta(n \log n)}$

Space Complexity

Space :- Given a program, how many memory cells are required in order to finish the algorithm (or) that program and this has to be analysed in terms of given input size

input size
↑
'n' — $O(n)$ → required extra cells

$O(1)$ → constant extra cells are required
ex:- 10, 20

$O(n^2)$

Algorithms

Recursive

Iterative

```
Algo(A, 1, n) {
    int i;
    for(i=1 to n)
        A[i] = 0;
}
```

→ For this program, the order of space complexity is $O(1)$, whatever, may be the no. of extra variables like i, j are added

Total space required

= (n) + (2) → i, j → two extra variables

So space complexity
→ $O(2)$ → constant
= $O(1)$

Consider because it has given has the input

The extra variables required apart from input, to solve a problem is called "Space Complexity"

* We should count only extra variables, added to pgm, because input is mandatory, we cannot increase (or) decrease it.

Ex2:

```
Algo (A, 1, n)
{
  int i;          ⟶ 1
  Create B[n];    ⟶ n
  for (i=1 to n)
    B[i] = A[i];
}
```

extra space ⇒ (1+n)
⇒ O(n)

extra space required grows along with the input given

3)

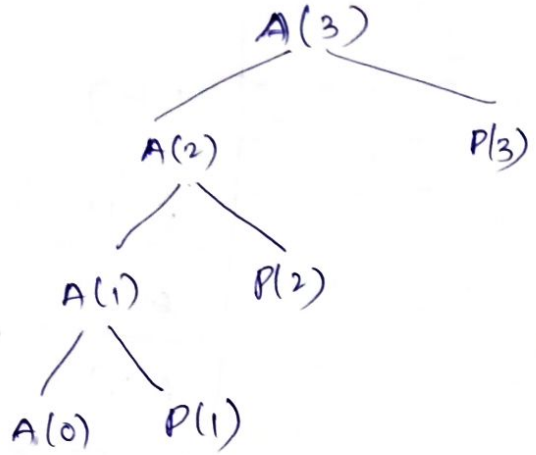
```
Algo( A, 1, n) {
Create B[n,n];   ⟶ n²
  int i, j;      ⟶ 2
for (i=1 to n)
   for (j=1 to n)
      B[i,j] = A[i];
}
```

extra space → (n²+2)
⇒ O(n²)

Ex: if '10' is 'n' then extra space $(10 \times 10)^{n^2}$

Recursion

```
A(n) {
  if (n>=1) {
    A(n-1);
    pf(n);
  }
}
```

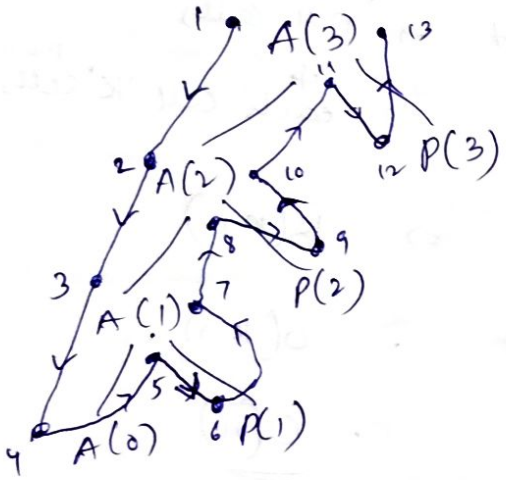


Tree formed according to recursive calls

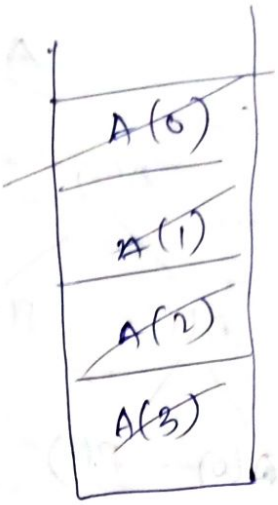
for A(3) ; we need 3+1 = 4 recursive calls

A(3) - 4
A(2) = 3
A(1) = 2

⇒ A(n) = n+1 for above example



→ You should traverse in tree from top to bottom and push the function calls on to the stack, and when you reach at last and (end of tree) visiting the node last time, pop the function out of the stack and start traversing from left to right.



→ Space complexity analysed, by the height of the stack or recursive calls made

→ for above ex:-

for 'n' input → 'n+1' calls

& each call 'k' cells memory it occupies

⇒ $k(n+1)$

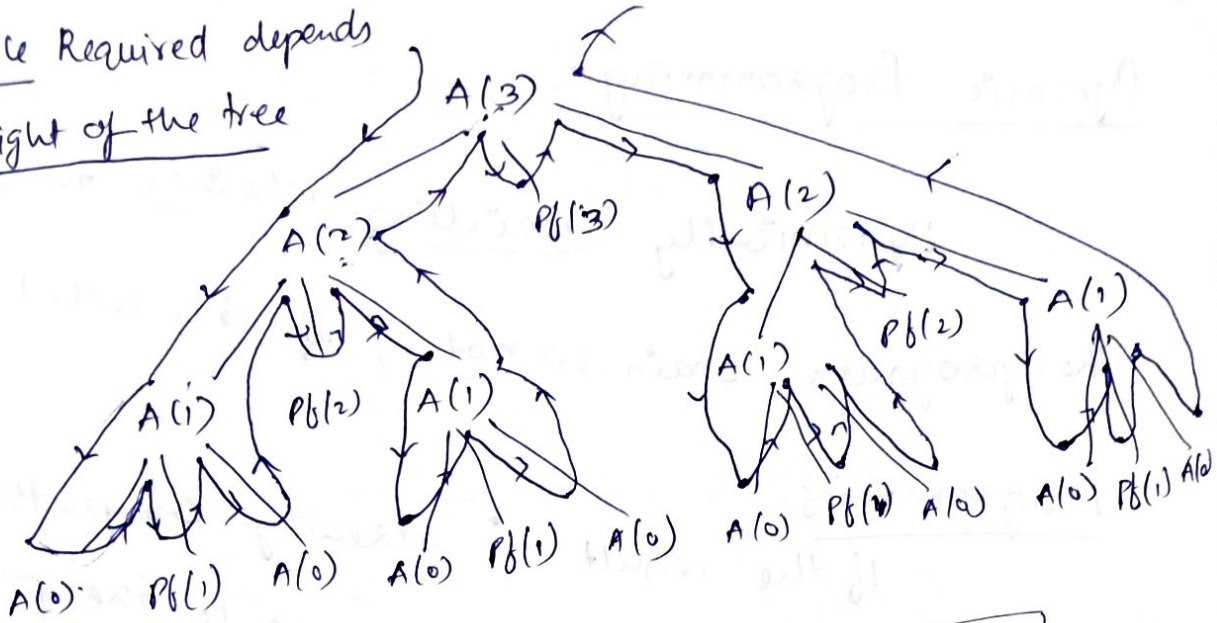
= $O(kn)$

= $O(n)$

ex2:-

```
A(n) {
if (n>=1) {
    A(n-1);
    Pf(n);
    A(n-1);
  }
}
```

* Space Required depends on height of the tree



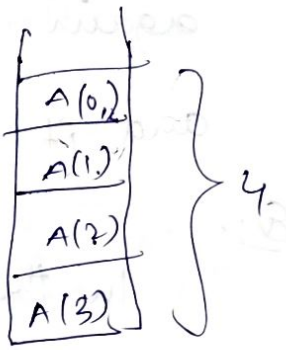
$A(3) = 15 = 2^{3+1} - 1$

$A(2) = 7 = 2^{2+1} - 1$

$A(1) = 3 = 2^{1+1} - 1$

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

→ The same cells are used by pushing & poping the recursive functions in the stack, so the space complexity is not equal to no. of recursive calls made



Space complexity ≠ no. of recursive calls

A(n) takes stack of size, (n+1)

space ⇒ (n+1) k

⇒ O(n)

output:

1 2 1 3 1 2 1

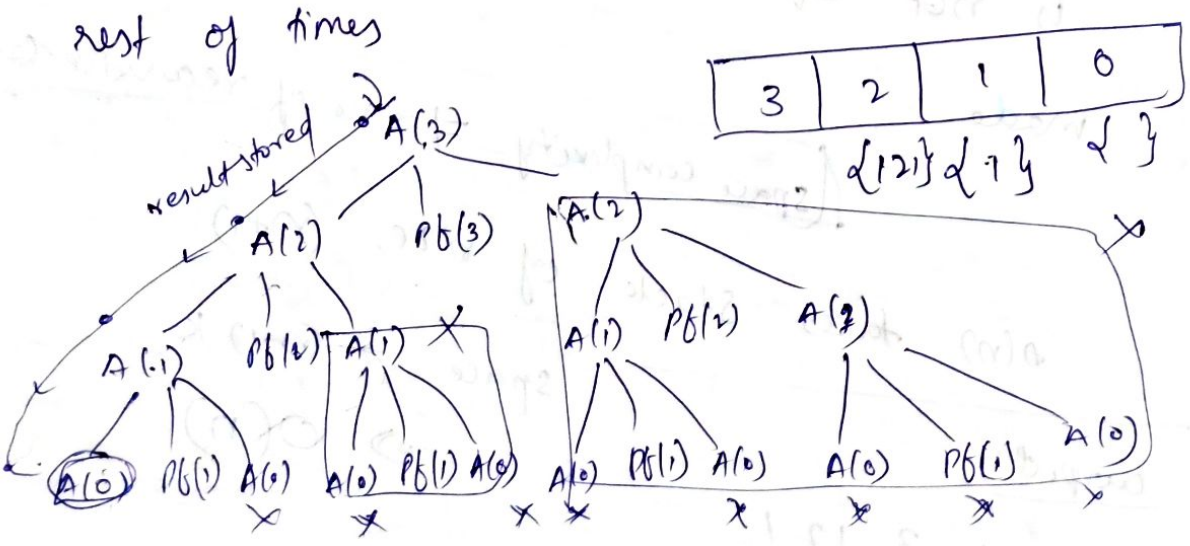
Dynamic Programming:

Dynamically deciding whether to execute the program again or not, it is called dynamic Programming.

If the result is already computed & stored, then we will not execute the program again, simply we will print the computed result and it is dynamically decided

Ex:-

In the above example, A(2) is computed twice, A(1) is computed four times, A(0), eight times. Store the result of above in table when they are computed first time, and use them for rest of times



Time complexity $= 2^{n+1}$

$= O(2^n)$

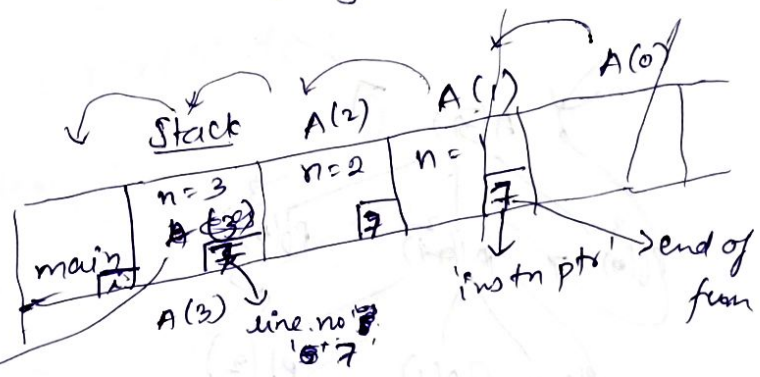
Recursion

two ways to trace

- Stack
- tree

```
1. A(n)
2. {
3.     if (n>0)
4.     { printf("%d", n-1);
5.
6.       A(n-1);
7.     }
8. }
```

```
main() {
    .
    .
    A(3)
    .
    .
}
```



* when a function is called "Activation Record" is maintained.

Instruction pointer:

Contains the ~~addr~~ next statement that is to be executed when again function returns

*Stored in Activation Record

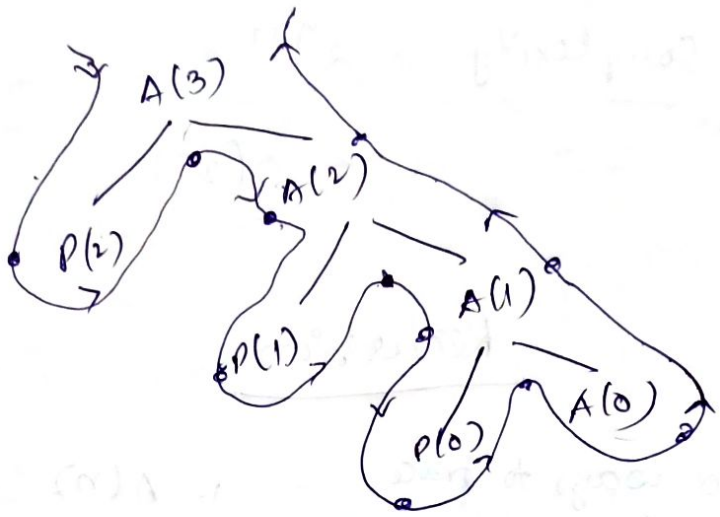
Output

2 1 0

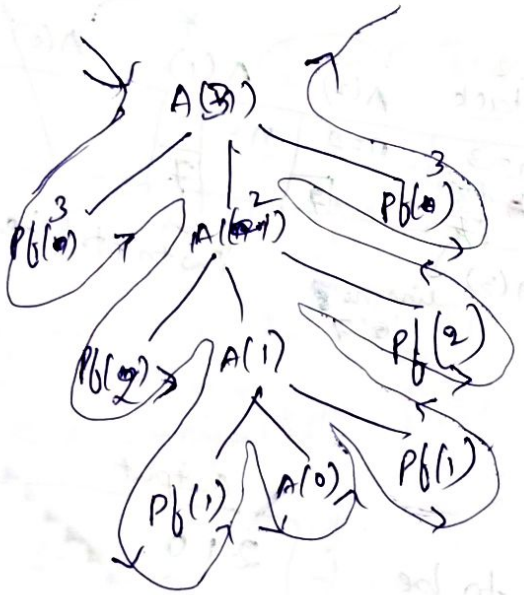
size of stack,

$A(n) = n+1$

$= O(n)$



→ Try printing fibonocci series, if you keep printf before recursion call you will get sequence, if you keep after it we will get reverse of sequence




```
A(n) {
if (n>0)
{   Pf(n);
    A(n-1);
    Pf(n);
}
}
```

3 2 1 0 1 2 3 will be printed

→ the stack size depends on the depth of the tree

Linked list

→ List-Reversal:

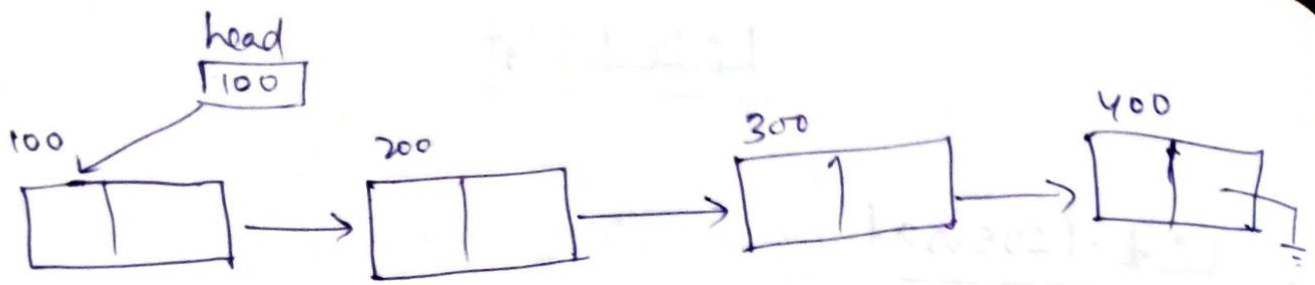
i) iterative

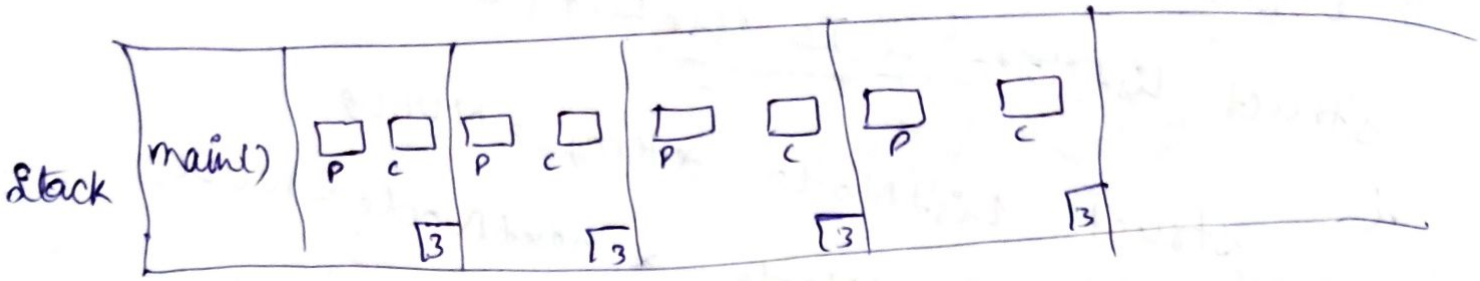
```
struct ListNode * ReverseList (struct ListNode *head)
{
    struct ListNode *temp = NULL;
    struct ListNode * nextNode = NULL;
    while(head) {
        nextNode = head -> next;
        head -> next = temp;
        temp = head;
        head = nextNode;
    }
    return temp;
}
```

2) Recursive

```
struct node *head;
void reverse(struct node * prev, struct node * cur) {
1   if (cur) {
2   reverse (cur, cur -> link);
3       cur -> link = prev;
    }
    else {
        head = prev; }
}
```

```
void main() {
    .
    .
reverse(NULL, head);
}
```


head
100
100
200
300
400


Stack
main()
p
c
3

TREES

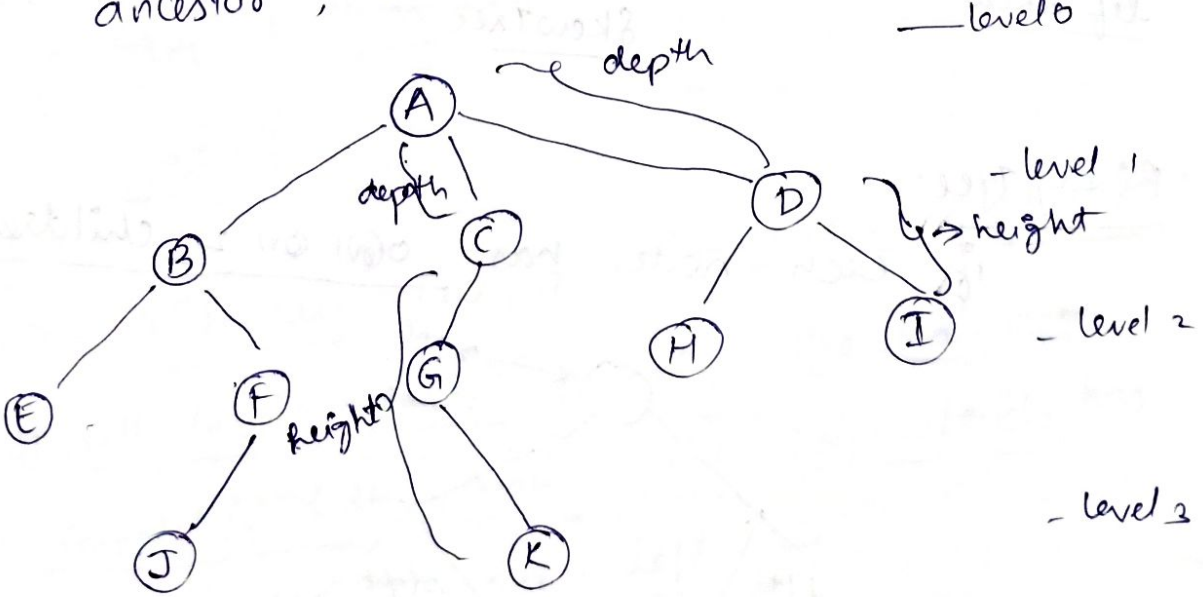
Tree:-

A data structure similar to linked list but instead of each node pointing simply to the next node in a linear fashion, each node points to a no of nodes

* → non-linear data structure
  → hierarchical representation
  → ADT

* **Important Notation:**

Root - no parents
edge - link b/w parent and child
leaf node - node with no children
siblings - children of same parent
ancestor, descendant



→ for a tree,

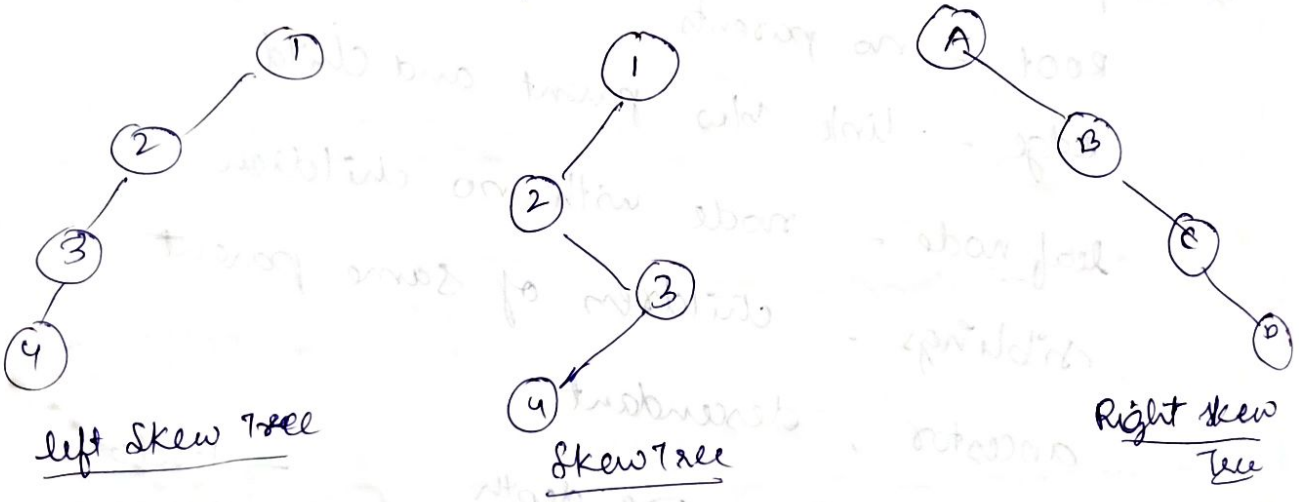
height: maximum height among all the nodes in the tree

depth: maximum depth among all the nodes in the tree

"height" & "depth" return same value for a tree

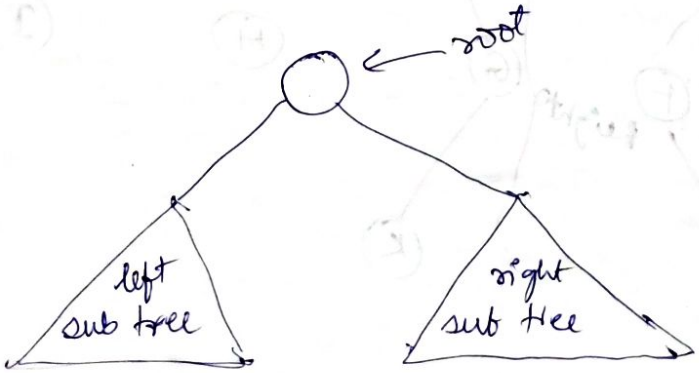
→ for individual nodes it will differ

Skewtrees: if every node has only one child (descendant) or none



Binary Tree:

if each node has 0 or 1 or 2 children

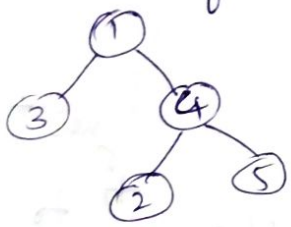


Types of B.T:

Strict B.T:

if each node has exactly two children (or) no children

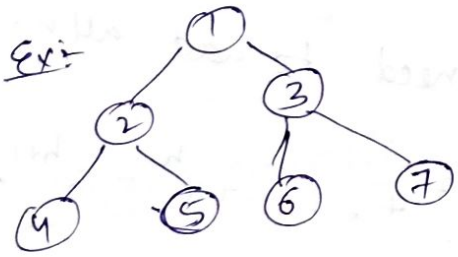
Ex:



Full B.T:

if each node has exactly two children and all leaf nodes are at same level

Ex:-



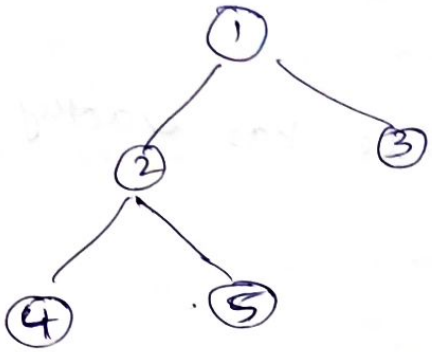
Complete B.T:-

1) A complete binary tree is a tree that is completely filled, with the possible exception of the bottom level

2) The bottom level is filled from left to right

→ A binary Tree, T with n levels is complete if all levels except possibly the last are completely full and the last levels has all its nodes to the left side

Ex:-



* No. of nodes in C.B.T is b/w 2^h to $2^{h+1}-1$ (h: height)

* no. of nodes in a full B.T is $2^{h+1}-1$. Since, there are 'h' levels we need to add all nodes at each level $[2^0 + 2^1 + 2^2 + \dots + 2^h = 2^{h+1}-1]$

Introduction to Tree Traversals

Traversing: To know what is present in the datastructure (Ex: tree, graph)

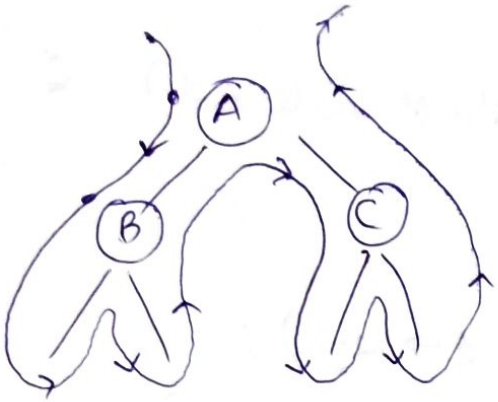
Searching: If we find an required element in datastructure then we stop

Traversal

In-order - LDR

Post-Order - LRD

Pre-Order - DLR



if we print nodes when we visit

1st time - Pre-order

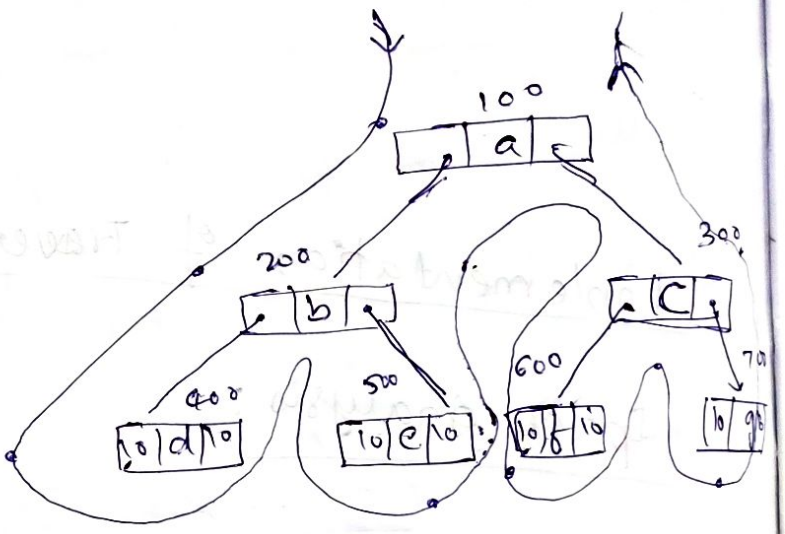
2nd time - In-order

3rd time - Post-order

Implementation of Traversals and Time and Space Analysis:

```
struct node {
    int data;
    struct node * left, right;
};
```

```
void Inorder ( struct node **)
{
    if (t) {
        1. Inorder ( t -> left);
        2. Printf ("%c", t -> data);
        3. Inorder ( t -> right);
        4. }
}
```

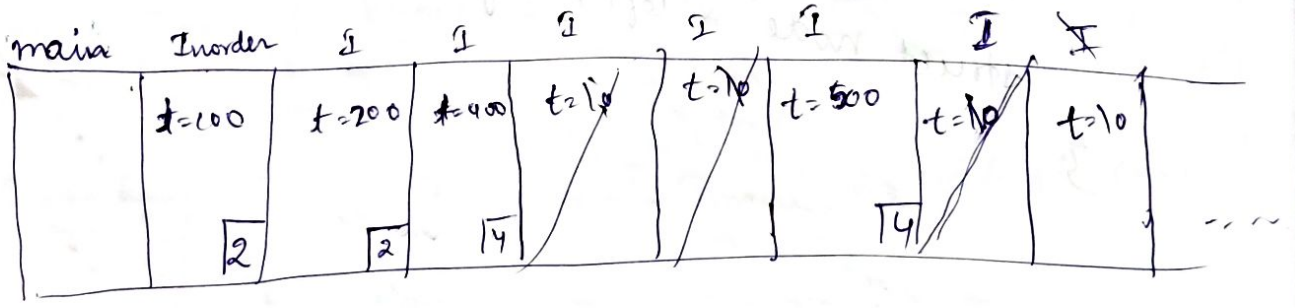


Preorder

2.
1.
3.

Post order

1.
3.
2.



Time Complexity:

n-nodes, each node visited '3' times & constant time spent at each node,

⇒ time complexity = 3 * n * Const time

= 3 * n * c

= O(n)

Space Complexity:

→ depends on no. of levels

→ worst case: Skew tree

→ O(n)

→ for all traversals,

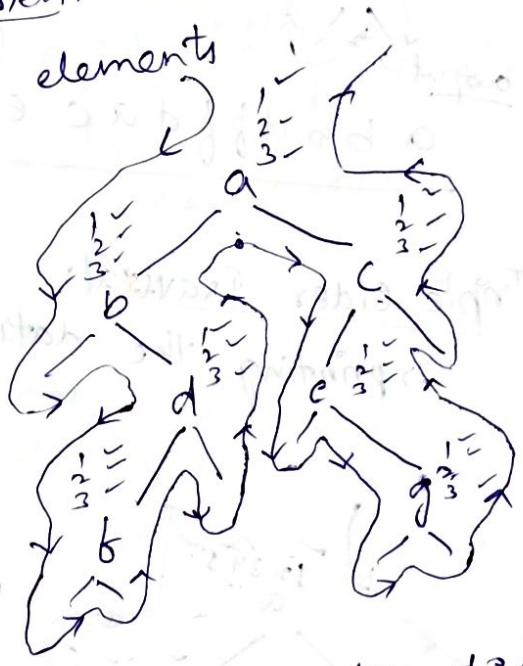
Space & Time Complexity = O(n)

Double-Order Traversal:-

↳ printing an element twice

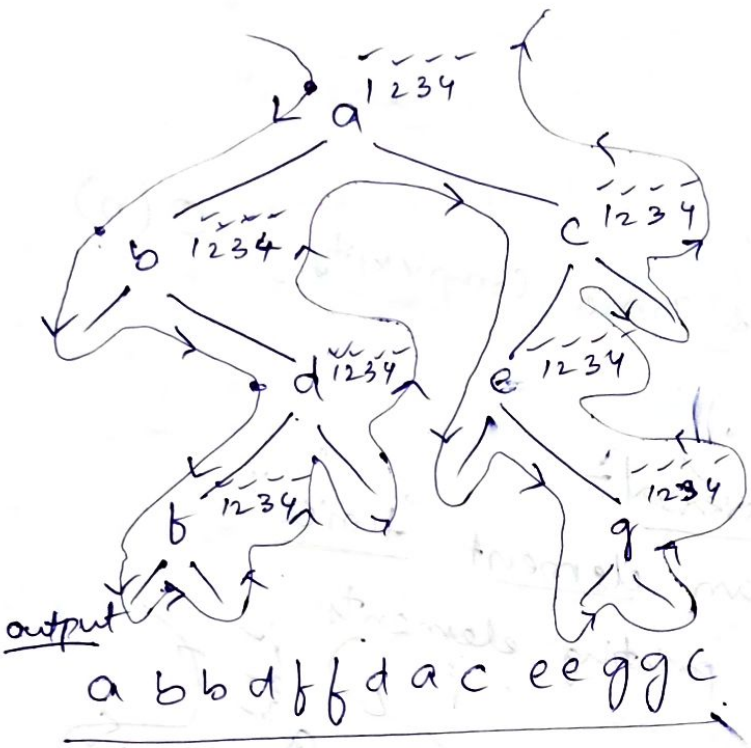
Ex: for Inorder, tracing printing elements

```
In(t) {
1. In(t→left);
2. Pf (t→data);
3. IN (t→right);
}
```



→ when you are traversing through the tree, tick each line executed, while writing code, it will help in tracing the order printed exactly

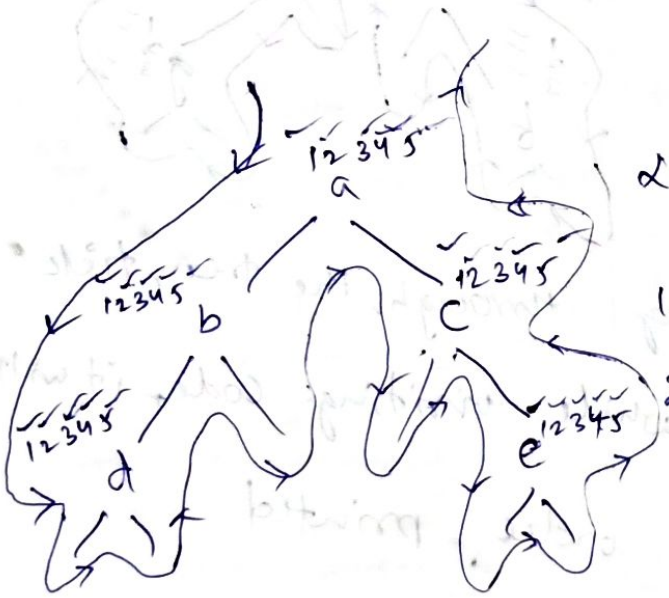
→ Mainly it shows, how to traverse during Recursion and what it prints




```
Double Order (t) {
if (t) {
1. pf(t->data);
2. Doub-Order(t->left)
3. pf(t->data);
4. Doub Order(t->right);
}
```

Triple order Traversal:

→ printing the data for three times

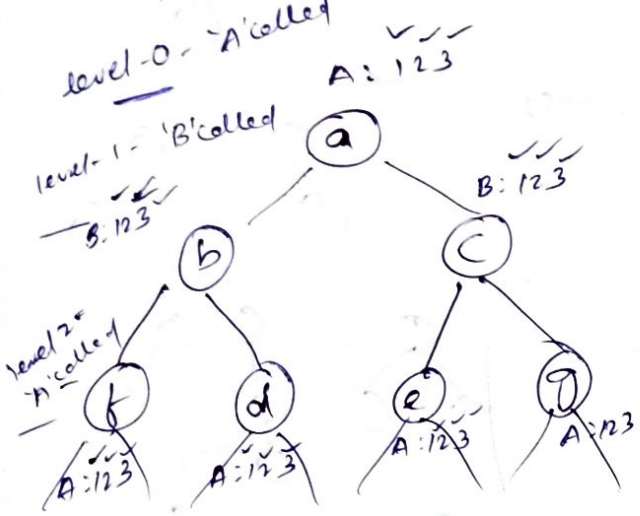



```
void TO (struct node *t)
{ if (t) {
1. Pf (t->data);
2. TO (t->LC);
3. Pf (t->data);
4 TO (t->RC)
5. Pf (t->data);
}
```

output:

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

Indirect Recursion on Trees



output:

a f b d e c g

→ two levels of Recursion

```
void A()
{
   B();
}
void B()
{
   C();
}
void C()
{
   A();
}
```

```
Void A (struct node *t)
{   if (t) {
1.    Pf ("%c", t->data);
2.    B(t->left);
3.    B(t->right);
   }
}
void B(struct node *t) {
    if (t) {
1.    A (t->left);
2.    Pf ("%c", t->data);
3.    A (t->right)
    }
}
void main()
{     A(root);
}
```

→ three (3) levels of recursion

A - level 0
B - " 1
C - " 2
A - " 3
B - " 4
C - " 5

Number of binary trees Possible

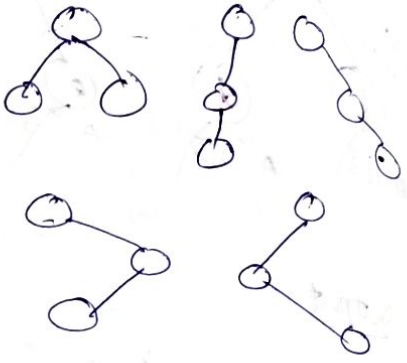
→ Unlabelled

$n \Rightarrow$ no of nodes

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

for labelled

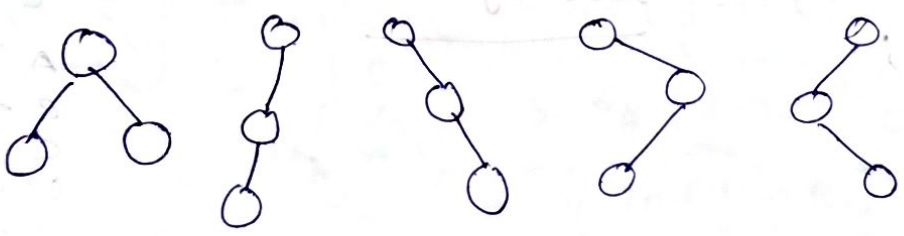
$$\frac{2n C_n}{(n+1)} * n!$$



each tree in unlabelled can be represented n! factorial ways

→ Q All the BT's with three nodes A, B, C which have pre-order 'ABC'

Sol:- no. of B-T's without label $= \frac{2n C_n}{(n+1)} = 5 \ (n=3)$

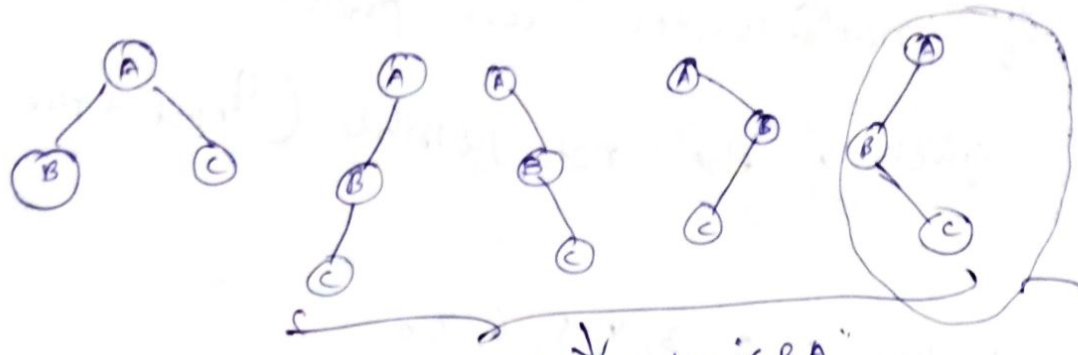


→ But for each tree, if labels are given, 3! trees can be formed (3!), but the given preorder 'ABC' can be formed by only one out of six

⇒ no. of B.T's with given preorder ABC

⇒ 5

⇒

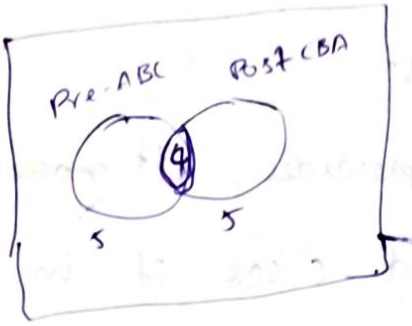


ii) Pre-order 'ABC' && Post Order 'CBA'

no. of trees = 30

⇒ '5' have Pre-order ABC

⇒ '4' have, both pre-order & post-order



iii) Pre-Order 'ABC' && Post Order 'CBA' && Inorder BCA

→ The above Condn is satisfied by only one 'tree' out of 30,

** Given, n nodes

Pre-order

$\frac{2^n C_n}{(n+1)}$ 'trees' are possible

Pre & Post more than '1' (or) equal to '1'

Pre & Post & In only '1' unique binary tree

→ Reconstruction of a binary tree is possible only if preorder & Inorder (or) post-order & In-order are given, else not possible (if post & pre are given)

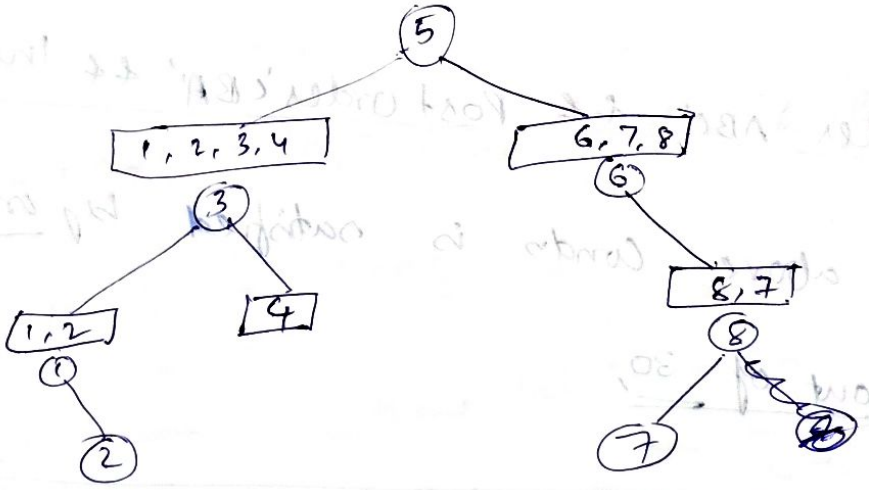
Q: Inorder: 1, 2, 3, 4, 5, 6, 7, 8

Preorder: 5, 3, 1, 2, 4, 6, 8, 7

Post: ?

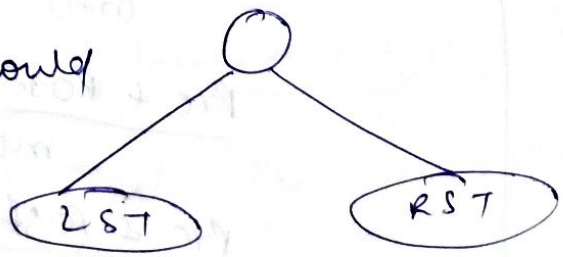
Sol: In -pre-order, 1st ~~node~~ element is root, i-e, root = 5

& and check it in order



Recursive Program to Count No. of Nodes :-

1. Write recursive equation before solving problem, then it could be easy for this type of problems



2 NN(T) = 1 + NN(LST) + NN(RST)

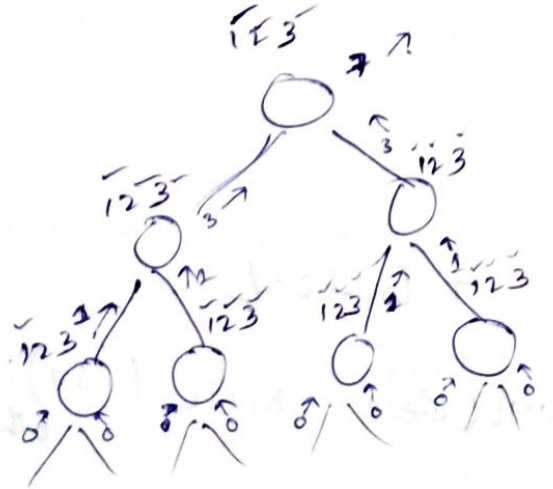
anchor cond^n. T == NULL;

⇒

NN: 1 + NN(LST) + NN(RST)
= 0 ; if T is NULL;

```
int NN (struct Node *t) {
if (t) {
return (1 + NN (t->left) + NN(t->right));
}
else
  return 0;
}
```

Elaborate manner



```
int NN( struct node *t)
{ if(t) {
    int l, r;
1.  l = NN(t->left);
2.  r = NN(t->right);
3.  return (1+l+r);
  }
  else
    return 0;
  }
```

Recursive Program to Count the no. of leaves and non leaves

i) Leaves

int NL(T) = 1 ; T is leaf

= NL(LST) + NL(RST) otherwise

```
int NL (struct node *t) {
    if (t==NULL)
        return 0;
    if (t->left==NULL && t->right==NULL)
        return 1;
    else
        return (NL(t->left) + NL (t->right));
}
```

ii) Non-Leaves:

NNL(T) = 0 ; T is leaf

= 1 + NNL(LST) + NNL (RST); otherwise

```
int NNL ( struct node *t) {
    if (t == NULL)
     return 0;
    if (t -> left == NULL && t->right == NULL)
       return 0;
    else {
       return (1+ NNL(t->left) + NNL (t->right));
}
```

Time Complexity:

$3 \times n \times c$ → constant amt of time spent at each node

3 → 3 times each node is visited

n → n nodes

$\Rightarrow O(n)$

Space Complexity:

$\Rightarrow O(n)$ → height of the stack → in worst case ⇒ Skew trees

Recursive Program to find Full Nodes:

$FN(T) = 0$; $T = NULL$

$= 0$; T is leaf

$= FN(LST) + FN(RST)$; if T has only one child.

$= 1 + FN(LST) + FN(RST)$: if 'T' is a full node

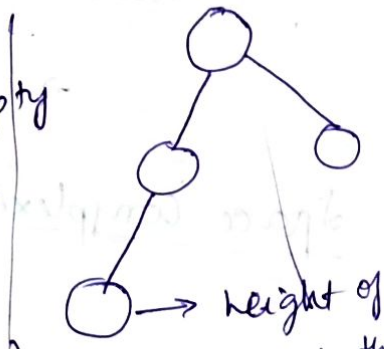
```
int FN (struct Node *t)
{
    if (!t) return 0;
    if (!t->left && !t->right)
        return 0;
    return (FN(t->left) + FN(t->right) +
                (t->left && t->right)?1:0);
}
```

Time & Space: O(n)

Recursive Program to find the height of a Tree

→ maximum of left & right subtrees should be returned

$$H(T) = \begin{cases} 0 & ; T \text{ is empty} \\ 0 & ; T \text{ is leaf} \\ 1 + \max(H(LST), H(RST)) & \end{cases}$$



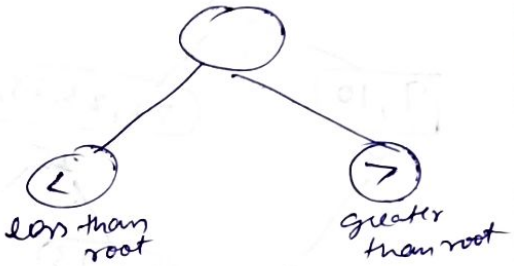
→ Height of leaf
in some books they call '0' & in some books '1'
Now considering "0"

```
int H(struct node *t)
{
    if (!t)
        return 0;
    if (!t->left && !t->right))
        return 0;
    return (1 + (H(t->left) > H(t->right) ? H(t->left) :
                                            H(t->right)));
}
```

Time & Space: O(n)

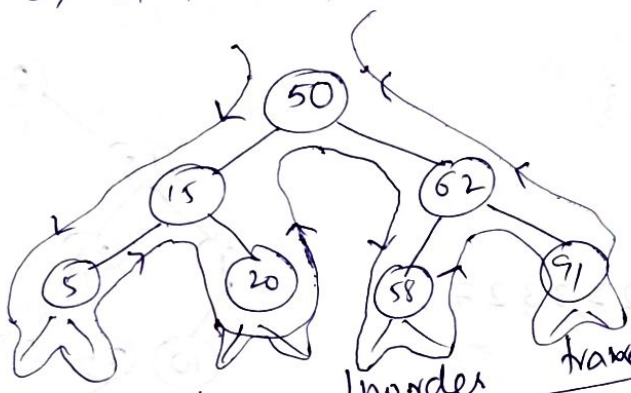
Binary Search Trees

→ Values are unique



Ex:-

50, 15, 62, 5, 20, 58, 91



Properties

→ If you perform Inorder traversal on a BST, we get nodes in increasing order

output

5, 15, 20, 50, 58, 62, 91

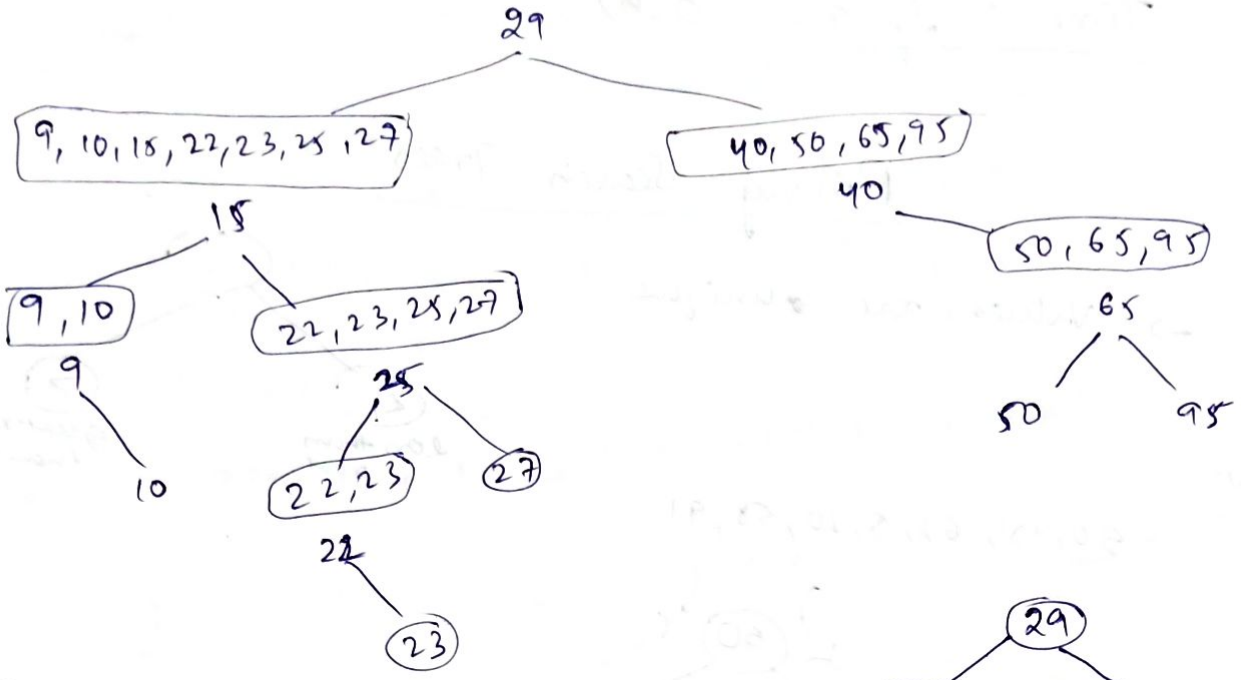
Q

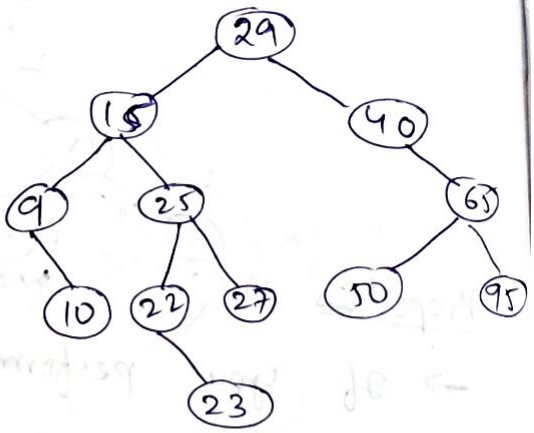
Q if post order is given. Then find Inorder & pre order of BST

Sol:- Post order: 10, 9, 23, 22, 27, 25, 15, 50, 95, 65, 40, 29

write in (sorted order) Inorder: 9, 10, 15, 22, 23, 25, 27, 29, 40, 50, 65, 95

Construct tree, from, In. order & post order.





Pre-order:

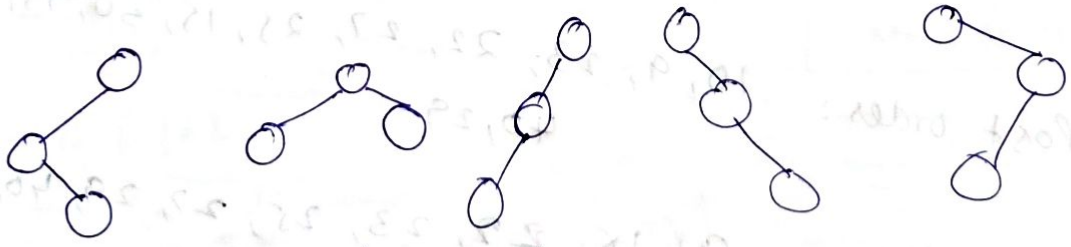
29, 15, 9, 10, 25, 22, 23, 27,
40, 65, 50, 95

2005 gate

How many BST's are possible with 4 distinct keys

Sol:

B.S.T – 3 distinct keys



no. of possible B.S.T's Trees of unlabelled ~~trees~~ type

$= \frac{2n_{C_n}}{(n+1)}$ ⇒ for 4 distinct keys

$\rightarrow \frac{8_{C_4}}{5} = 14$

⇒ 14 distinct trees

Q (1-100) in BST, search for 55. which of the following sequences cannot be the sequence of nodes examined?

a) {10, 75, 64, 43, 60, 57, 55}

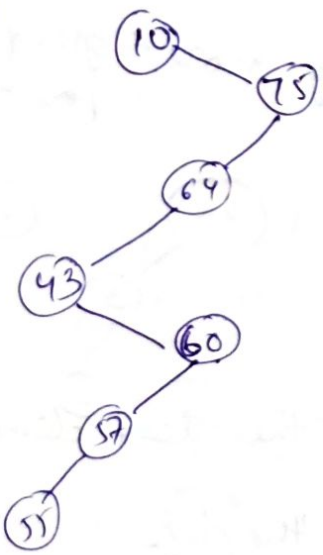
b) {90, 12, 68, 34, 62, 45, 55}

c) {9, 85, 47, 68, 43, 57, 55}

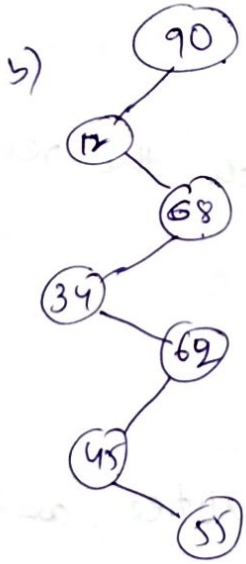
d) {79, 14, 72, 56, 16, 53, 55}

Ans Draw the tree & check consistency

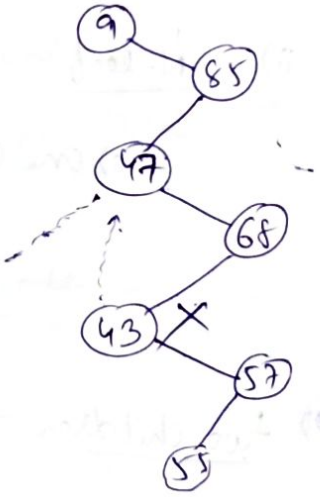
a)



b)



c)



$43 < 47$ → inconsistent

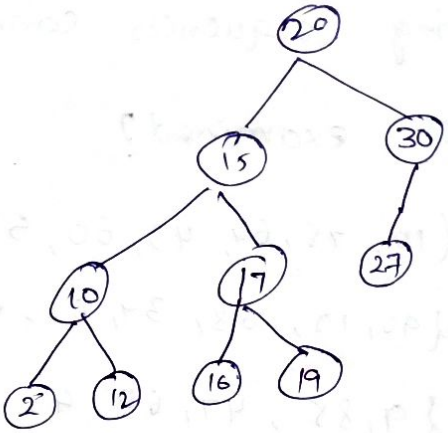
Q Identify pre, post, in-orders in given sequences

I : M B C A F H P Y K → Post order

II : K A M C B Y P F H → Pre order

III : M A B C K Y F P H → Inorder

Deleting a node from BST



i) leaf

ii) Non-leaf

a) one child

b) two children

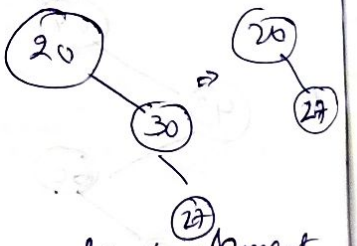
i) leaf

directly delete it

ii) N-leaf

a) one child

↳ attach the remaining node to grand parents



b) two children

a) go to right subtree, ~~and~~ take the least element in right subtree and place it in the place because (inorder successor)

(or)

b) go to left subtree, take the highest element and replace it there because (inorder predecessor)

In deletion In-order successor (or) Inorder predecessor will have only one child (or) no child

→ Finding max, min, insertion, traversal algorithms

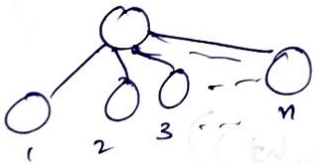
Gate 98 ⇒ A complete n-ary tree is one in which every node has '0' (or) 'n' sons. If 'x' is the no. of internal nodes of a complete n-ary tree, the no of leaves in it is given by.

Sol: n-ary tree ⇒ n-nodes for a parent

complete n-ary ⇒ either 0 (or) n-children

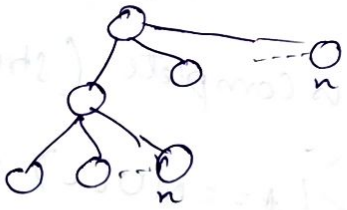
1 - internal node

n - leaves



n - leaves

2 - internal node



(n-1+n) - leaves

= (2n-1)

3 - int. nodes

(n-2+n+n)

= (2n-2) +n

= 3n-2

4 - int. nodes

(3n-3) +n

4n-3

if 'x' int. nodes

xn - (x-1)

x(n-1) +1

02 The no of leaf nodes in a rooted tree of 'n' nodes; with each node having 0 or 3 children is

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

Recursive Program on testing whether a tree is Complete B.T

Complete B.T (according to Siv):

A tree containing either '0' or '2' children for all nodes

(Empty tree also complete)

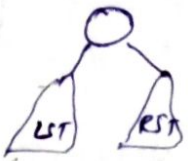
```
int isComplete (struct node *t) {
    if (t==NULL) return 1;
    if ( !(t->left) && !(t->right))
        return 1;
    else if (t->left && t->right)
    return ( isComplete (t->left) && isComplete(t->right))
    else
        return 0;
}
```

Introduction to AVL trees and balancing

→ Height of balanced B.T will be in order of 'logn' so search time also O(logn)

→ AVL tree is a balanced binary tree

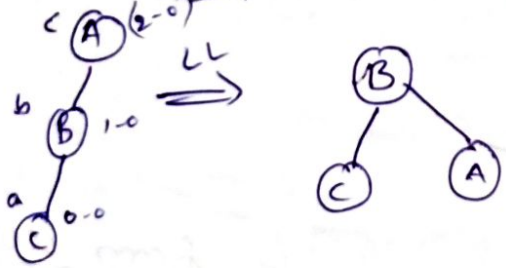
→ Based on 'Balance Factor' (B.F)



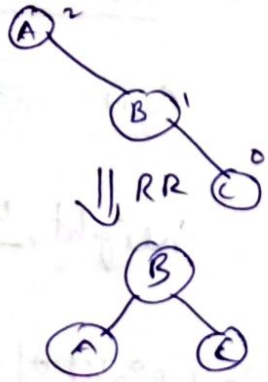
$$B.F = |Height(LST) - Height(RST)|$$

$-1 \leq BF \leq 1$ → should be satisfied

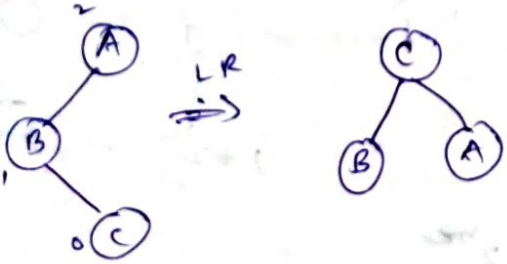
LL Rotation:



RR Rotation

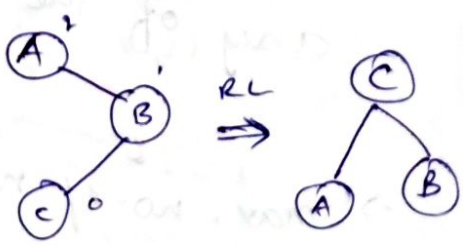


LR - Rotation



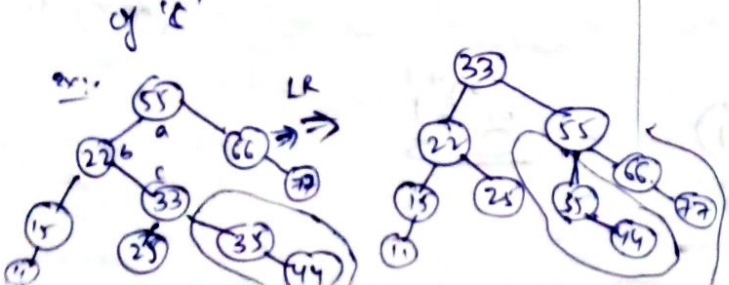
→ Take care of right sub-tree of 'c'

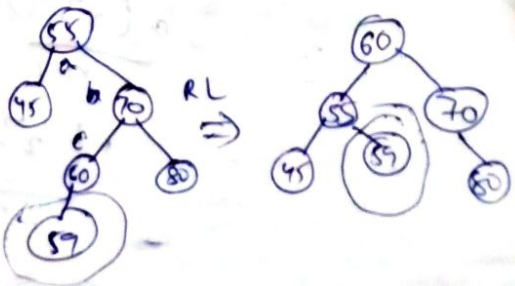
RL - Rotation



→ Take care of left subtree of 'c'

Ex:





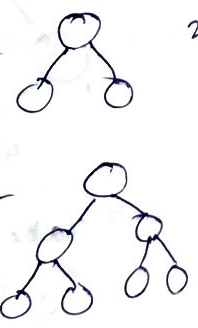
	BST	Balanced BST
Search	$O(n)$	$O(\log n)$
height	$O(n)$	$O(\log n)$
Insertion	$O(n)$	$O(\log n) + O(\log n) + c$ $\hookrightarrow O(\log n)$

Minimum and Maximum nodes in an AVL tree of height "h":

→ max. no. of nodes in an AVL tree of height, h (or) max. no of nodes in a binary tree of height, h are same

→ Height of tree:
length of the longest path from root to any of the leaf

max. no. of nodes	height
1	0
$1+2^1$	1
$1+2^1+2^2$	2



→ max. no. of nodes

$= 1 + 2^1 + 2^2 + 2^3 + \ldots 2^{height}$

$= \frac{a(r^{n+1}-1)}{r-1}$

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

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

For minimum no. of nodes:

take trees of height h-1

For AVL (add root & balance tree by adding a tree of height h-2)

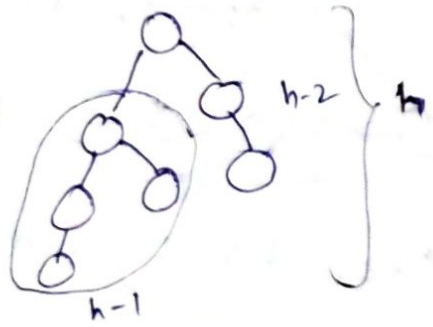
height

2 ○

2

3

4



No. of nodes for min. AVL Tree

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

$= 1 \quad ; h = 0$

$= 2 \quad ; h = 1$

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

$= 2 + 1 + 1$

$= 4$ nodes

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

$= 4 + 1 + 2$

$= 7$

for normal B.T:

$(h+1)$: min. nodes of height 'h'

Q95 A binary tree 'T' has 'n' leaf nodes. The no. of nodes of degree '2' in 'T' is

a) $\log_2 n$ c) n d) 2^n

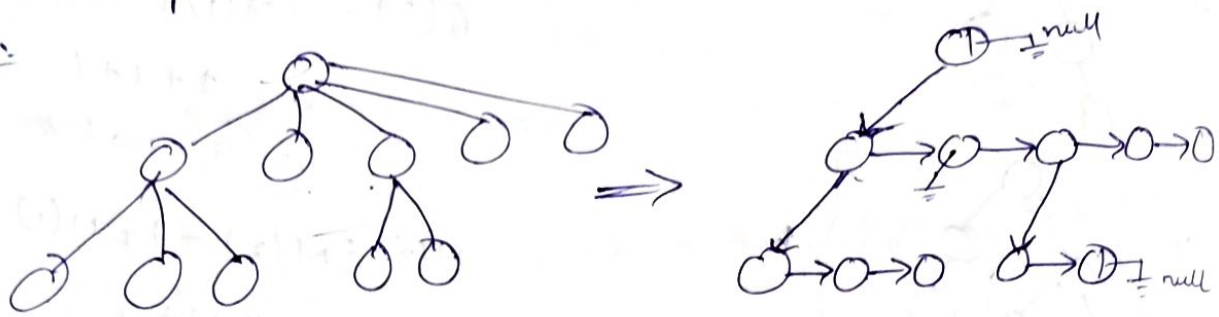
b) $n-1$

Various Tree Representations

1) if we have '10' children, then we have to have "10 pointers" in a structure & it is waste of memory. So, we use,

→ LCRS (Left most child Right Sibling) Representation

Ex:



→ Used to represent trees having more children like '10', '20'..

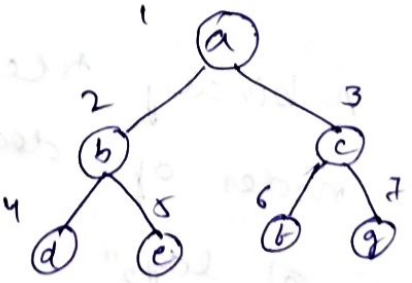
Array Representation

i - parent

Left child : $2i$

Right child : $2i+1$

$p(x) = x/2$



→ multiplying by '2' ⇒ left shifting binary number

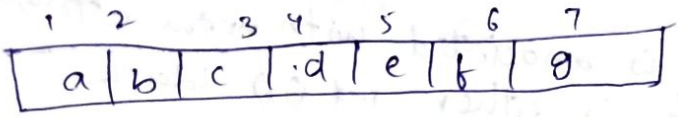
⇒ 10 (2) → left shift → 100 (4) → left shift

(or) add '1' → 101 → right shift

dividing by '2' ⇒ right shift ⇒ 100 (4) R.S → 10 (2)

101 (5) R.S → 10 (2)

→ parent

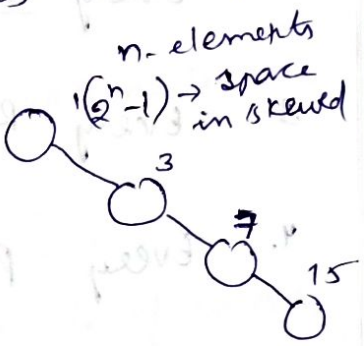


→ very easy for traversal from parent ⇄ child

⇒ *) widely used in "heaps"

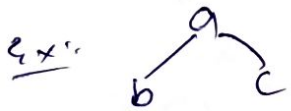
but if it "skewed" then leads a lot of wastage of memory

Ex:



Nested form:-

Ex:



(a bc)
R L R

Ex:

a (b de) ((fg)
R L R

Representations

i) Normal (pointers, structures)

ii) LCRS (n-ary $n \geq 2$)

iii) Arrays (heaps)

iv) Nested.

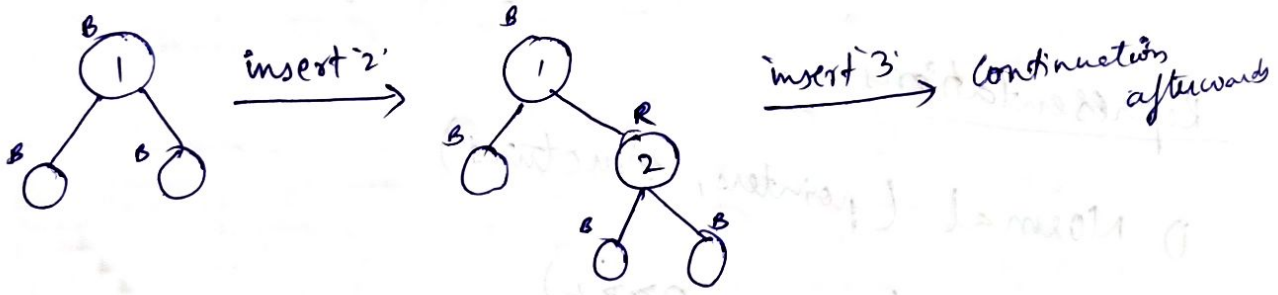
Red-black Trees

Each node is associated with extra attribute, the color which is either red (or) black.

Properties:

1. Every node is either red (or) black
2. Every nil node is BLACK
3. Every RED node has two BLACK child nodes
4. Every path from node "x" (any node) down to a leaf node has the same number of BLACK nodes
5. The root node is always BLACK

Ex:- Insertion of 1, 2, 3, 4, 5, 6

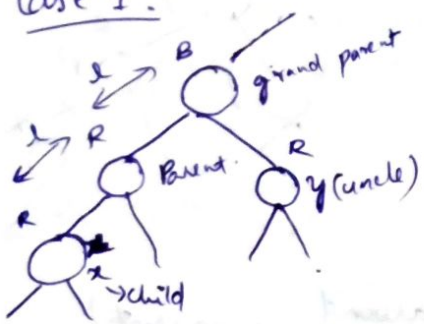


Why Red-Black Tree?:

BST operations (eg: search, max, min, delete, insert.. etc) take O(h) time, where 'h' is the height of the BST. The cost of these operations may become O(n) for a skewed binary Tree. But for R-B Tree, always height is "logn", so, it is guarantee that an upper bound of O(logn) for all these operations.

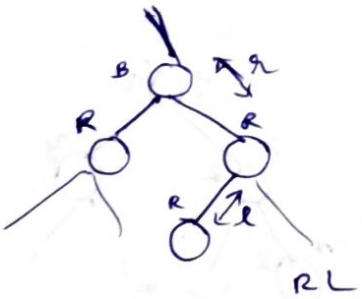
Cases of Violations for R-B tree:-

Case 1:



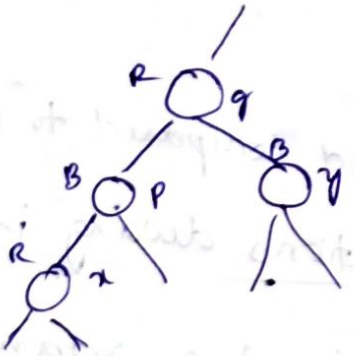
"LL"

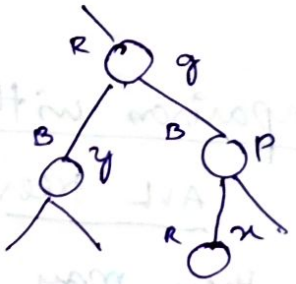
(i) Uncle is Red



i) This occurs when trying to insert a red node as a child of another red node

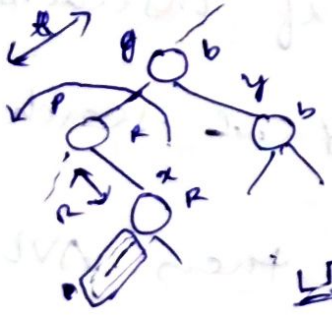
Fix:- change the color of parent, uncle, & grand parent

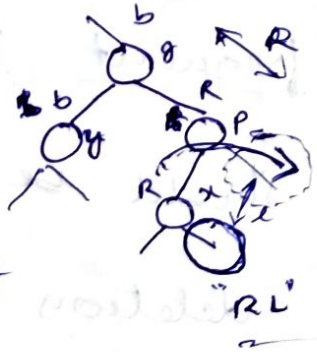




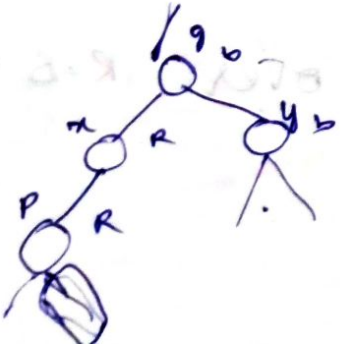
Case 2:

ii) Uncle is Black

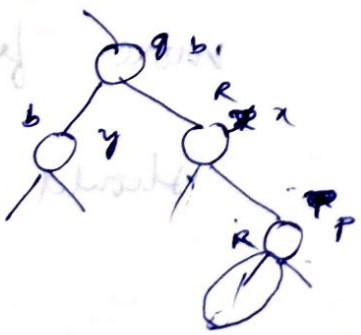




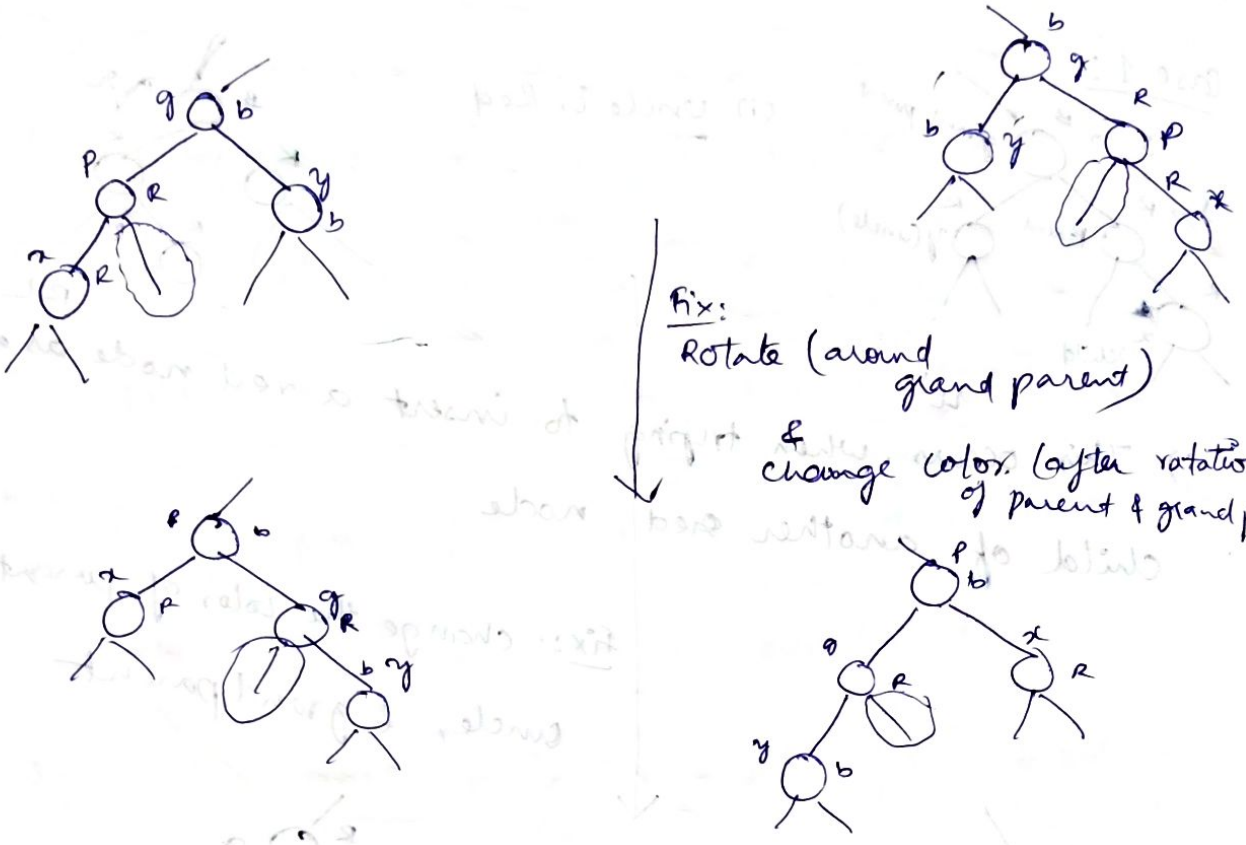
Fix:
Rotate parent



Case 3



Case 3 :- i) Uncle is Black



Comparison with AVL Tree :-

AVL trees are more balanced compared to R-B trees, but they may cause more rotations during insertion & deletion. So, if application involves many frequent insertions & deletions, the R-B trees should be preferred. And if insertions & deletions are less frequent & search is more frequent operation, then AVL tree should be preferred over R-B tree

Introduction to Graphs

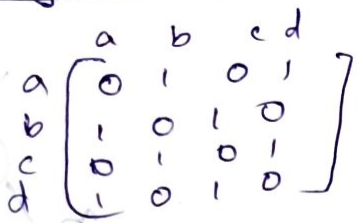
→ Traversal

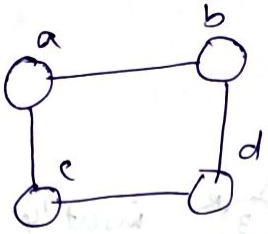
Breadth-first - checks in terms of levels like level 0 ; level 1 ...

Depth-first:
selects a node and searches for it in depth till end

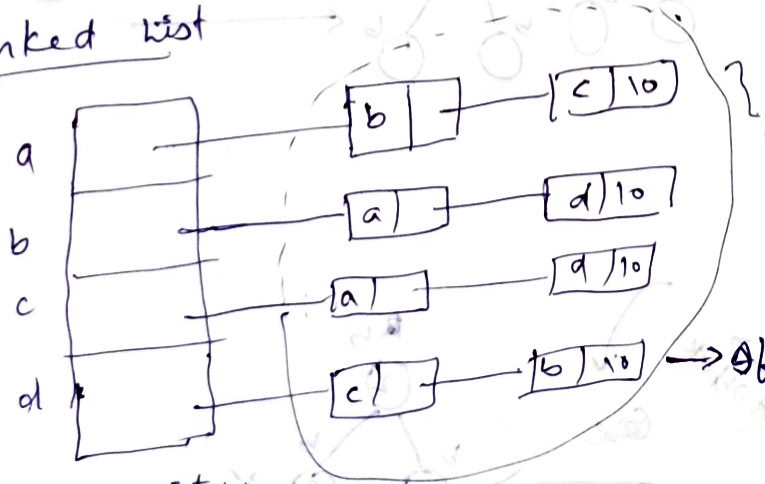
Representations of Graphs

i) Adjacency Matrix Representation





ii) Linked list



array of pointers

→ Of undirected - $2E$ (edges) elements

directed - E elements

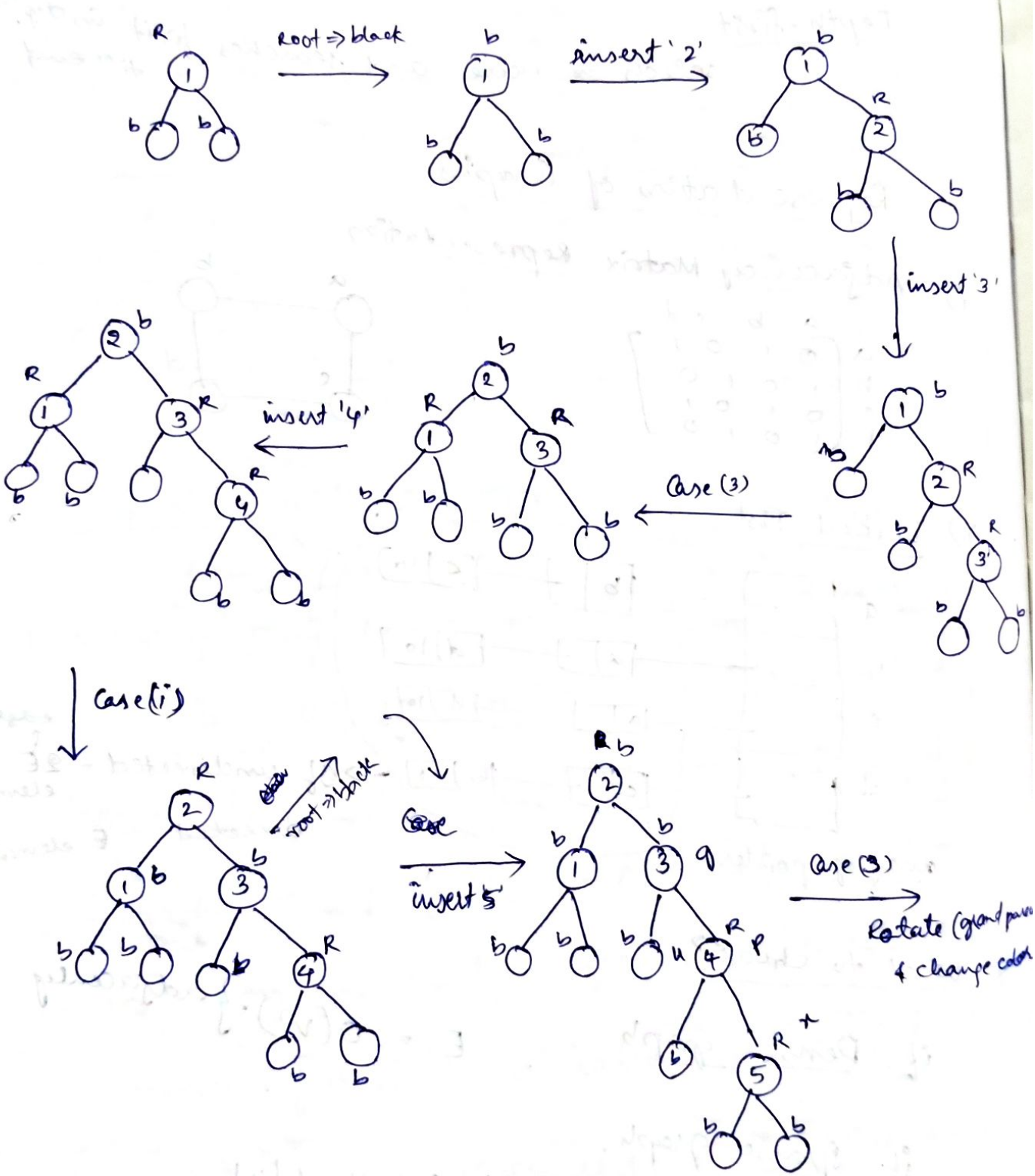
what to choose?

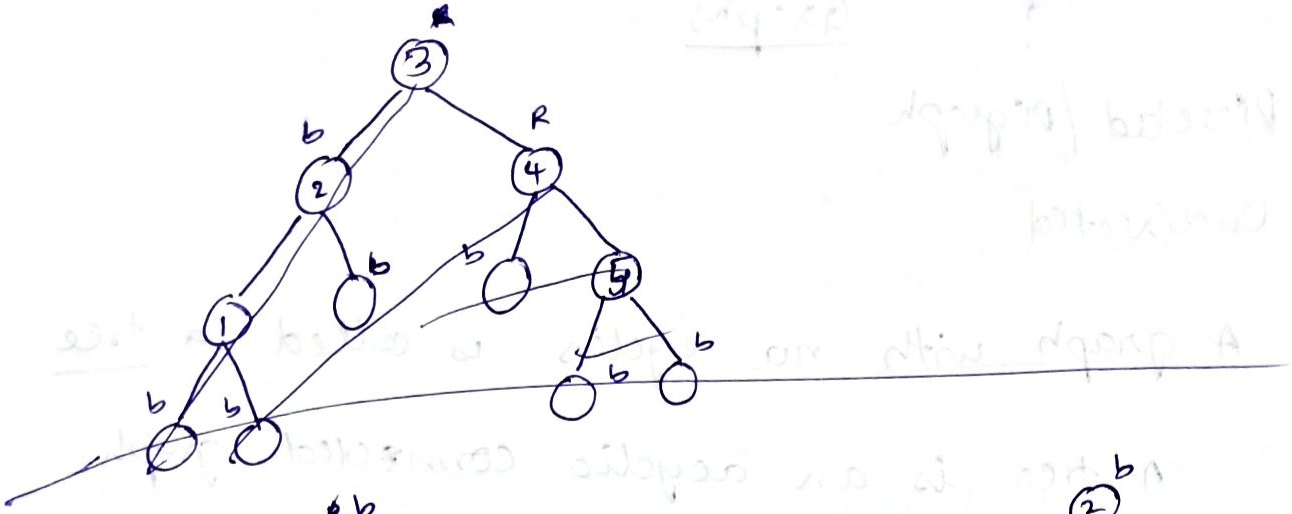
if Dense graph $E = O(V^2)$ } adjacency

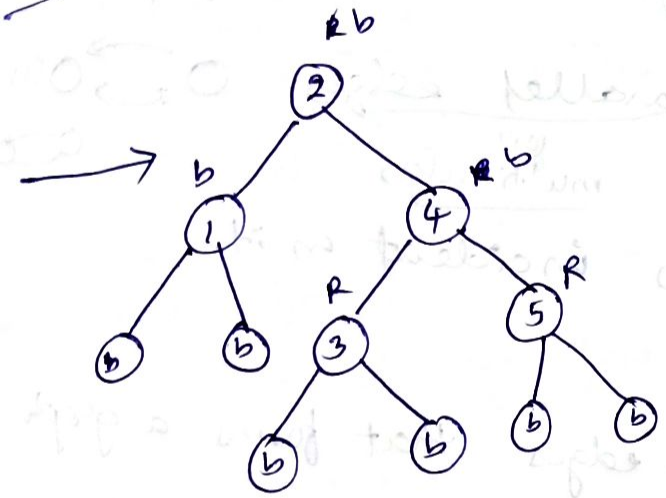
if Sparse graph. $E = O(V)$ } Linked list

Ex:- Red-black Tree (1, 2, 3, 4, 5, 6 elements insertion)

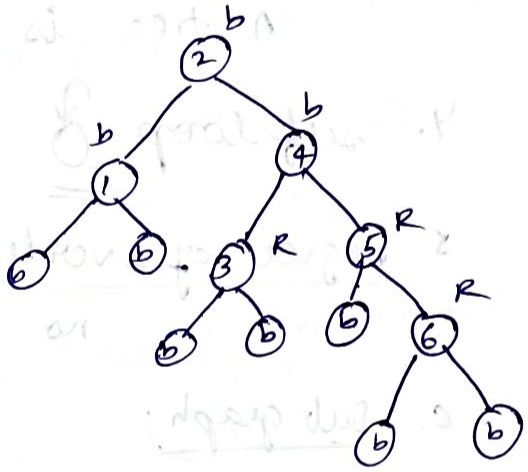
i) while inserting, you have insert a 'Red node' & then change tree without violating rules



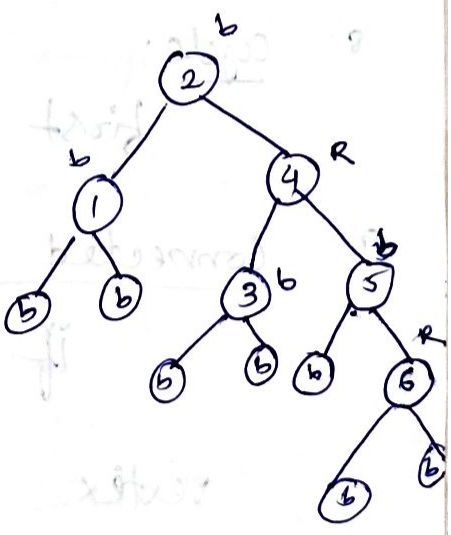




insert '6'



case (i)



∴ final R.B Tree ⇒

Graphs

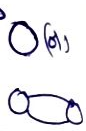
1) Directed / Digraph

2. Undirected

3. A graph with no cycles is called a tree

A tree is an acyclic connected graph

4. self loop

5) Parallel edges (or) multi-edges 

5. Degree of a vertex:
no. of edges incident on it.

6. Sub graph:
Subset of graph edges that forms a graph

7. Path.
sequence of adjacent vertices

8. cycle:
first & last node of path are same

9. Connected graph:
If every vertex is connected every other vertex in graph.

10. Directed Acyclic graph:
Directed graph with no cycles

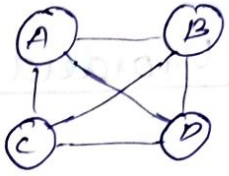
11. Forest:-
disjoint set of trees

Bipartite graph :-

It is graph whose vertices can be divided into two sets such that all edges connect a vertex in one set with a vertex in other set.

12. Complete graph :

Graphs with all edges present are called complete graphs



13. Sparse Graph:

with relatively few edges are called Sparse graphs

14 Dense graph:

with relatively few possible edges are missing

Applns:

1. representing relationship b/w components in electronic circuits
2. Transportation n/ws:
3. Computer n/ws:
4. Databases : Representing ER, dependency of tables

Graph:

A graph, G is an ordered pair of a set V of vertices and a set of edges, E

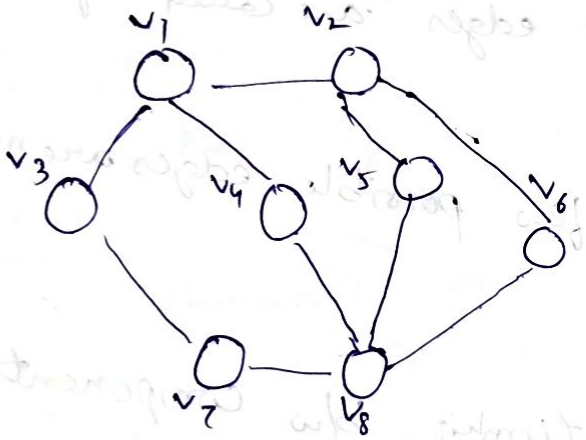
$$G = (V, E)$$

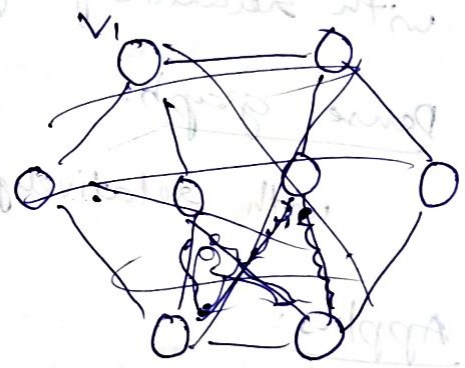
Ordered pair

$$(a, b) \neq (b, a) \quad \text{if } a \neq b$$

Unordered pair:

$$\{a, b\} = \{b, a\}$$





$V = \{ v_1, v_2, v_3, v_4, v_5, v_6, v_7, v_8 \}$

$E = \{ \{v_1, v_2\}, \{v_1, v_3\}, \{v_1, v_4\}, \{v_2, v_5\}, \{v_2, v_6\}, \{v_3, v_8\}, \{v_4, v_8\}, \{v_5, v_8\}, \{v_6, v_8\}, \{v_7, v_8\} \}$

Edges:-

Directed

undirected

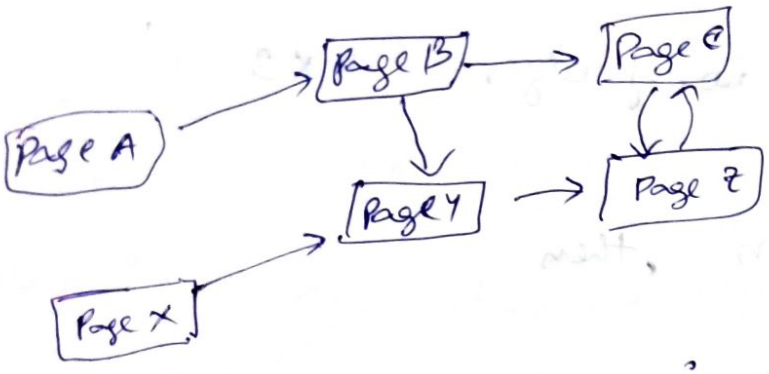
$\{u, v\}$

Ex:-

Q Find all nodes having length of shortest path from Rama equal to 2.

(friends of friends in fb)

② world wide web (directed graph)



google → web-crawling is graph traversal

(or) act of visiting all nodes in graph

Weighted vs Unweighted:

Ex:- Intercity n/w → weighted

social n/w → unweighted.

can draw a undirected graph → directed graph

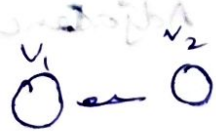
but cannot draw from directed → undirected

$|V|$ = no. of vertices

$|E|$ = no. of edges

Simple graph:

A graph with no self loops & parallel edges



No. of Edges:-

$V = \{v_1, v_2, v_3, v_4\}$

$|V| = 4$

minimum edges = ϕ

max. no. of edges = 4×3

Directed:

if $|V| = n$, then

$0 \leq |E| \leq n(n-1)$; directed

$0 \leq |E| \leq \frac{n(n-1)}{2}$; undirected (because edges in two directions not possible)

↳ (Assuming no self-loop (or) multi-edges are present)

Ex:

if $|V| = 10 \Rightarrow |E| \leq 90$

$|V| = 100 \Rightarrow |E| \leq 9900$

$|E|$ is very very large compared to $|V|$

$\Rightarrow \underline{|E| \propto |V|^2}$

Dense graph ⇒ too many edges, $|E| \approx |V|^2$ → Adjacency matrix

Sparse graph ⇒ too few edges, $|E| \propto |V|$ → Adjacency list

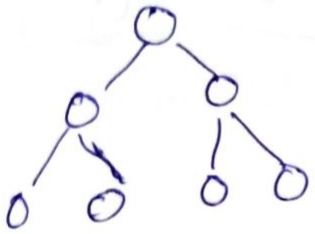
Path: A sequence of vertices where each adjacent pair is connected by an edge

<A, B, F, H>

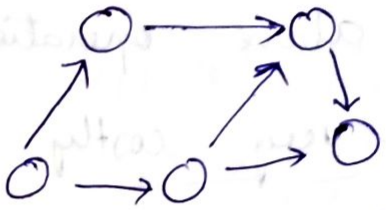
Simple path: no vertices are repeated in path

Trail: A walk in which no edges are repeated

Acyclic graph:



Undirected acyclic graph



Directed acyclic graph

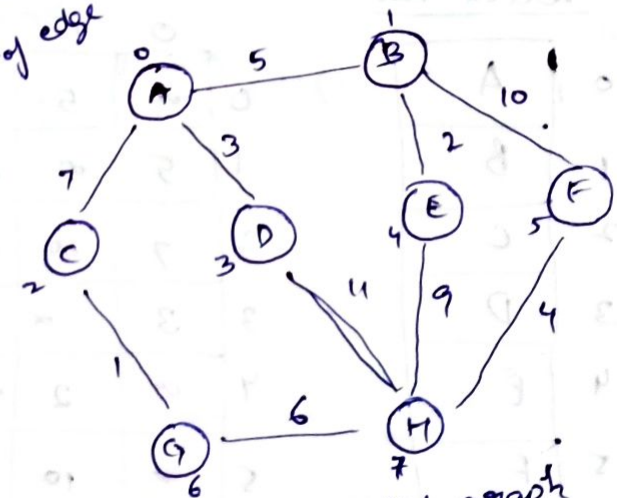
GRAPH REPRESENTATION

Vertex list

0	A
1	B
2	C
3	D
4	E
5	F
6	G
7	H

Edge list (starting of edge, ending of edge, weight of edge)

0	1	5
0	2	7
0	3	3
1	4	2
1	5	10
2	6	1
3	7	11
4	7	9
5	7	4



a weighted graph

→ Not very efficient for frequently performed operations like

operation	Running time
1. finding adjacent nodes	O(|E|)
2. Check if given nodes are connected	O(|E|)

→ we have to scan entire edge list to perform above operations .i.e, O(|E|) (or) O(|V| * |V|) very costly

Try for O(|V|)

Array Representation (Adjacency Matrix Representation)

vertex list

0	A
1	B
2	C
3	D
4	E
5	F
6	G
7	H

"A"

	0	1	2	3	4	5	6	7
0	∞	5	7	3	∞	∞	∞	∞
1	5	∞	∞	∞	2	10	∞	∞
2	7	∞	∞	∞	∞	∞	1	∞
3	3	∞	∞	∞	∞	∞	∞	11
4	∞	2	∞	∞	∞	∞	∞	9
5	∞	10	∞	∞	∞	∞	∞	4
6	∞	∞	1	∞	∞	∞	∞	6
7	∞	∞	∞	6	11	9	4	∞

M = V

→ (V * V) → array size

$A_{ij} = \begin{cases} 1 & : \text{if } \exists \text{ edge from } i \text{ to } j \\ 0 & : \text{otherwise} \end{cases}$

Undirected graph: matrix Symmetric

directed graph: not symmetric

Operations — Time cost

1. finding adjacent nodes — $O(V) + O(V) \Rightarrow O(V)$
   (↓ scanning vertex, ↓ scanning row in matrix)
2. finding two nodes are connected — $O(1)$ → if indices are given
   $O(V)$ → if names are given

$\boxed{O(V^2)}$ → Space (big trade off of Space)

Dense → good

Sparse → very bad

Good if graph is dense (or) V^2 is too less to matter

Hashing

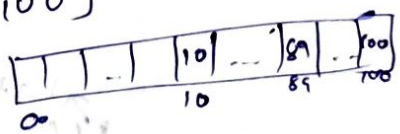
Data structure ⇒ Insertion
Searching
Deletion

Hashing - solution for minimising the search time
↳ O(1) is possible

Before hashing,

Direct Address Table (DAT):
→ very early (starting)
→ Here key values are stored in corresponding indexes

Ex: Suppose, if we want to store key values b/w 1 & 100 an array of '100' locations is created and values are inserted in corresponding indices {1, 2, 3, 4 ... 100}



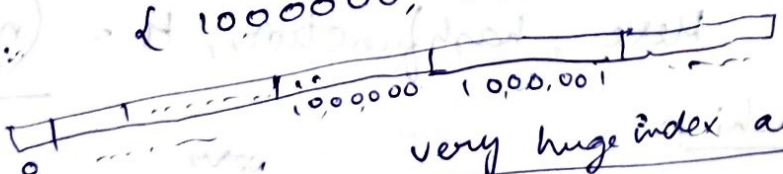
Searching

1. Unsorted array - O(n)
2. Sorted array - O(log n)
3. Linked list - O(n)
4. Binary Tree - O(n)
5. B.S.T - O(~~log~~ n)
6. Balanced B.S.T - O(log n)
7. Priority (min & max heaps) - O(n)

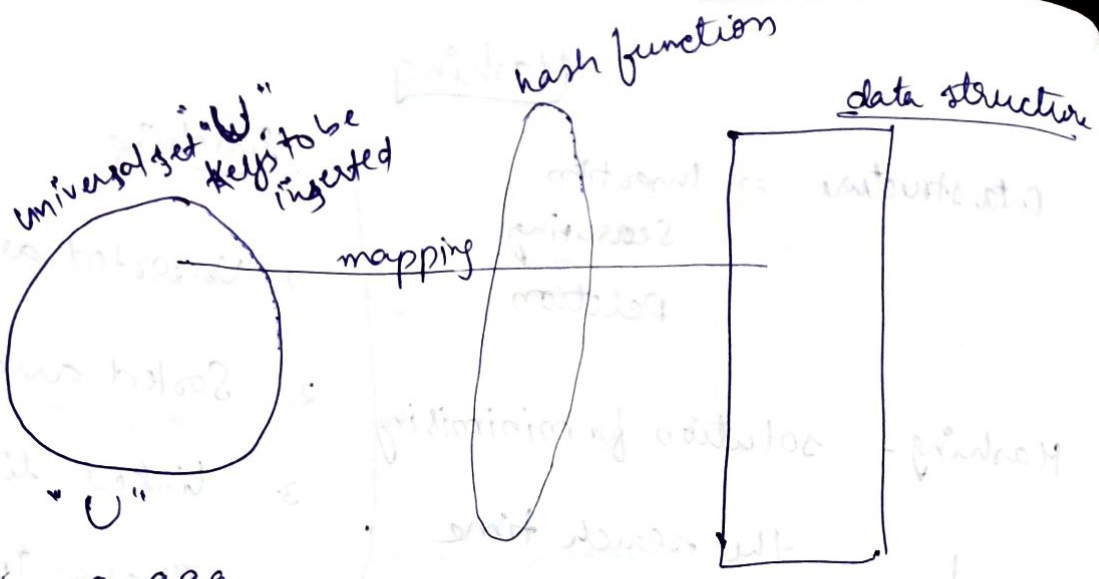
minimum - O(log n)

Disadv

Ex: {1000000, 1000001, 1000002, ...}



very huge index array



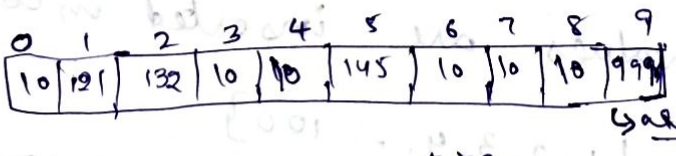
Ex: 0-999

consider max '10' elements & declare an array of size '10'

→ while inserting an element, apply function on that element & obtain address and store, while retrieving the element, apply function on that element & retrieve

Ex: (0-999) range

values
121, 145, 132, 999



hash function
H(0-999) → values (0-9)

Here, hash function, H = (mod 10)

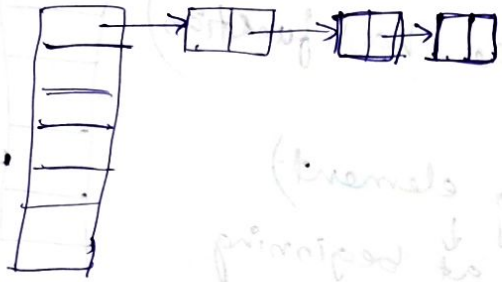
→ Hashing:
Taking a set of very large no's & then mapping it to a set of small numbers, corresponding to the indices of hash table.

Problem → Collision

Solution (Collision Resolution Techniques):

1. Better hashing functions (i.e, minimise collisions)

ii) chaining



iii) Open. Adressing

⇒ Recompute the hash function, until you find empty space

→ Probing - Searching for empty space

a) Linear Probing

b) Quadratic Probing

c) Double hashing

Diff b/w chaining & Open. Add:

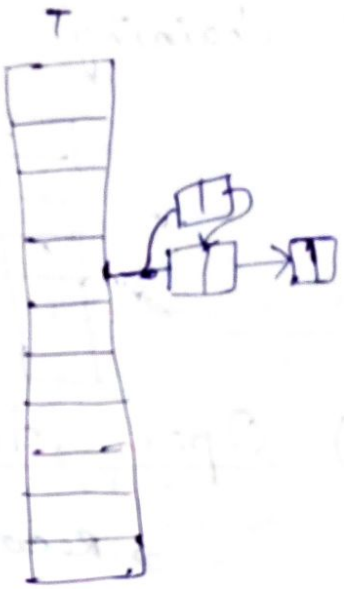
chaining: 1. Values are stored outside the table,
2. wastage of memory for ptrs
3. Better for Insertion, Deletion, Searching
4. size can exceed hashtable size

Open. Add: 1. Values inside the table
2. Increase in table size, better utilisation of space
3. Better for Insertion, Searching
4. Size cannot exceed hash table

Chaining

element with, K in Table 'T' using hashfunction,

⇒ $T[h(K)]$



(Time taken for insertion) = (Compute hash function) + (inserting element at beginning)

⇒ $O(1)$

Search:

worst case → all keys → one index

⇒ $O(n)$

Deletion

worst case → $O(n)$

Average Case:-

$\alpha = \frac{n \text{ of elements}}{m \text{ slots}}$ → equal share for all slots in

average case

load factor = $\left(\frac{\text{Total no. of elements}}{\text{slots available}}\right)$

$O(1+n/m)$

$\rightarrow O(1+\alpha)$

$n = km$

$\Rightarrow \alpha = k$

$\Rightarrow O(1+\alpha) = O(1)$

Deletion : $O(1)$

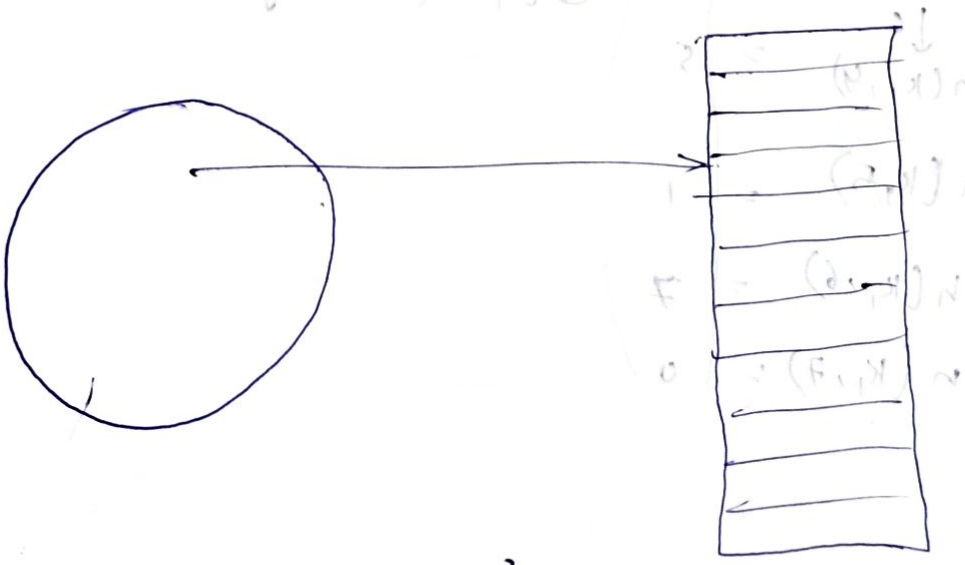
Disadv:-

- extra space for pointers (data outside table)

Open Addressing
(or) (closed hashing)

- all elements inside the table
- load factor, $\alpha = \frac{n}{m} = \frac{\text{(no. of elements)}}{\text{(no. of slots available)}}$

$0 \le \alpha \le 1$



if collision → probe again with changes

$h : U \to \{0, 1, 2, 3 \ldots m-1\}$

$h : (U \times \{0, 1, 2, \ldots m-1\}) \to \{0, 1, 2, 3 \ldots m-1\}$

Take (key, & probe no)

Searching:

$O(m)$ → all slots to be probed

$h(k_1) = 1 \Rightarrow$ free $\Rightarrow$ insert

$h(k_2) = 2 \Rightarrow$ collision $\Rightarrow$ again probe

$h(k_2, 1) = 6 \Rightarrow$ collision $\Rightarrow$ again probe

$h(k_2, 2) = 7 \Rightarrow$ free insert

$h(k, 0) = 2$
↓ collision
$h(k, 1) = 6$
↓ c
$h(k, 2) = 3$
↓ c
$h(k, 3) = 4$
↓ c
$h(k, 4) = 5$
$h(k, 5) = 1$
$h(k, 6) = 7$
$h(k, 7) = 0$

⇒ Hash function should be in such a way, that before declaring that element cannot be inserted (or) table is full, we should probe each & every cell

Sorting Techniques

1. Insertion Sort (Playing Cards in hand)

```
void Insertion_Sort (A)
{
  for j = 2 to A.length
      key = A[j]
      // insert a[j] into sorted sequence A[1..j-1]
      i = j-1;
      while ( i > 0 and a[i] > key)
          a[i+1] = a[i]
          i = i-1
      a[i+1] = key;
}
```

(or)

```
void insertionSort ( int a[], int n)
{
    int i, key, j;
    for (i=1; i<n; i++)
    {
        key = a[i];
        j = j-1;
        /* move elements of a[0,1,... i-1], that are
        greater than key, to one position ahead of
        their current position */
        while ( j >= 0 && a[j] > key)
        {
            a[j+1] = a[j]; j = j-1;
        }
        a[j+1] = key;
    }
}
```

Time Complexity : $O(n^2)$

Auxiliary Space : $O(1)$

Boundary Cases : max time - sorted in reverse order
min time - if in sorted order

Algorithmic Paradigm : Incremental Approach

Sorting In Place : yes → If we don't use extra space for sorting then it is inplace

Stable : yes

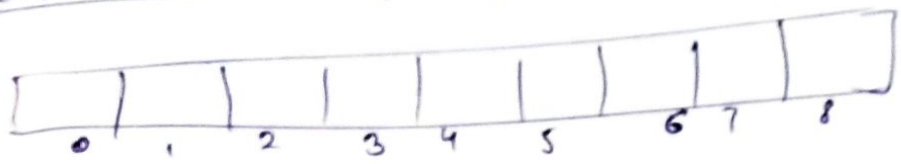
Online : yes

Online Algorithms :-

It is a algo which does not require the data to be processed, at the starting of processing and the data may be of streaming kind.

Ex:- Insertion sorting technique can work even all the numbers to be sorted are not present in the memory at a time, but others cannot

Worst Case Time Complexity Analysis



- $i=1 = 1+1 = 2 = 2(1)$
  $i=2 = 2+2 = 4 = 2(2)$
  $i=3 = 3+3 = 6 = 2(3)$
  ⋮
  $i=n = n+n = 2n = 2(n)$

$2(n) \Rightarrow 2(1+2+3+\dots n)$

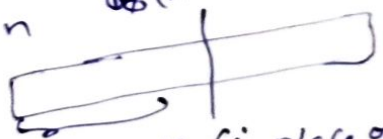
$2(1) + 2(2) + 2(3) + \dots$

$= 2\left(\frac{n(n+1)}{2}\right) = O(n(n-1))$

$= O(n^2)$

→

	Comparisons	movements	for n² total elements
Binary search	$O(\log n)$	n	n
Doubly linked list	Linear search $O(n)$	$O(1)$	$n(n-1)$ n^2



Binary search (in place of linear search)