Program Development

Data Structures
(Organizing the data)
or
Representation (methods) of data
↓
linked list, tree, arrays.

(Algorithms)
↓
Step by step procedure for solving a problem or performing a task.

choose a data structure ← when we apply algorithms.

when using DS we have to develop efficient algo for performing various operations on these data structures
↓
Algorithms → Searching, Sorting.

Problem
- Soln Algo
- Soln Algo
- Soln Algo
- Soln Algo

So there can be many algo. to solve a particular problem.
Various Sorting Algo.

↓
Best suited for our req. (most efficient).

Comparison of Algorithms

Efficiency
- Running time of algo } less running time
- memory occupied. } less memory.

N/w Usage, Disk Usage;
mainly concerned about the running time.

Measuring Running Time
- Experimental / ~~Analytical~~ Empirical Method.
- Analytical.

Exp. → Implement an algo in a lang.
Run it on diff. i/p's.
Record exact running time.
Not feasible & has certain limitations.

Processor Pentium → 10ms I3 – 7ms I7 – 2ms.

Prog. lang – C++ more faster than C.

(1) Particular lang. & particular computer so the running time becomes dependent on s/w & h/w used for implementing the algo. This can lead to bad results.
Bad algo might take less time than a good algo. if it run on faster m/c (computer) or implemented in an efficient lang. Run both of them using exactly same s/w & h/w environment which might not be possible always.

(2) Limited no. of I/P's. → miss I/P's in which it behaves diff.

(3) Consumes a lot of time some algo takes lot of time to execute so not feasible.

Uniform Value नहीं मानेंगे।

⇒ Theoretical method is used which analyse the running time of algo. based on input size. Asymptotic analysis. In this we don't calculate the exact running time of algo. & therefore no need to run the program.

→ Independent of s/w & h/w. & allow us consider all possible inputs. (approximate values based on time & space Comp.)

Size of I/P (n)	⟷	Running time.
Less size	——	Less " "
More "	——	More " "
↑	——	↑.
Double	——	Double, quadruple, 20 times etc.

Analysis an algo → how the running time of an algo. increases with the increase in I/P size.

If running time increases rapidly with the ↑ in input size then that algo is not considered as an efficient algo.

To determine efficiency of an algo → see how the algo. behaves when the I/P size is increased.

how the running time ↑es.

Shortest path algo.
Shopping → Search algo — Recommend Products.
Everywhere using algo.

Def — Steps — finite
finite set of instructions used to solve a computational problem.
" Sequence of steps / instructions — — —
set X. same type of multiple values.
we cannot repeat a step again & again
Read a ——
One step will be one at a time.

Need of Algo.
Building → Map → Construction. without
Program → Update → Planning — designing. — Destroy.
Wastage of money / time.
Actual Const — Map — Algo. (Architect).
To design a program algorithms are reqd.
Design — Implementation.
Algorithm / Once finalized. Program.

Algo	Program
→ At Design Phase.	→ at implementation phase.
→ Natural language simple english lang, no need to follow any syntax or rules but to some extent there is some format to write an algo.	→ Written in any prog lang & req. to follow Syntax of that particular lang.
→ The person who has the domain Knowledge about the subject (domain) ~~जिसको डोमेन program लिखना है।~~	→ Programmer writes a program.
→ We analyze algo.	→ Testing of program. (Run & Compile)
→ Paper. (Independent of H/w & H/w)	→ IDE (Dependent)

Properties.

Finiteness & Unambignous, Clear.

Correctness — Tea steps follow Getting Proper O/P. nothing else.

Analysis of Algorithms.

Priori → Dependent of no. of times statements are executed.

Characteristics of an algo:-

→ Input → 0 or more input. well defined.

print ("Hello world").

→ O/P → atleast 1 or more output.

→ Unambignous / Definiteness → Every inst. must be clear and no ambiguity. (One meaning)

→ Read a

→ Read b.

→ Finiteness → steps must be countable.

```
while (1)   always true
{
}
```

Infinite loop. finite no. of steps & every inst. in that algo must take finite amount of time for execution.

→ Effectiveness → It should perform that task for which you have written that algorithm. Should not contain any unnecessary steps. statements.

```
start
Read a
Read b
sum = a+b
Print & sum
end.
stop.
```

Read cX.

Every statement should be feasible not imaginary.

Asymptotic Notations ⇒ Mathematical way of representing the time complexity.

1) Big - Oh (O)

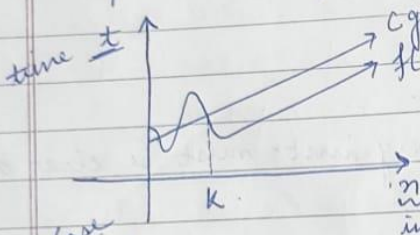
no. of times function अथवा - आप को call कर रहा है, " " " कोई statement execute हो रही है called iteration / frequency.

represented by some notations.

Big O represents → Upperbound (Atmost) (least, closest value)

किसी भी काम को करने के लिए आपको ज्यादा से ज्यादा कितना time लगेगा।

Constant ⇒ c



Worst case

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

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

$c > 0$

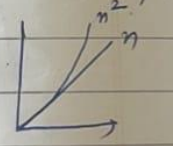
$n \geq K$

$K \geq 0$

$f(n) = 2n^2 + n$

$f(n) = O(n^2)$

f(n) { Problem को solve करने के लिए time लगा }



$2n^2 + n \leq c \cdot g(n^2)$

Quadratic Linear

Linear

$2n^2 + n \leq 2 \cdot n^2$

⇓

3.

$2(5)^2 + 5 \leq 3(5^2)$

$55 \leq 75$

$2n^2 + n \leq 3n^2$

$n \leq n^2$ Divide both sides by n.

$1 \leq n$

or $n \geq 1$

c = 3

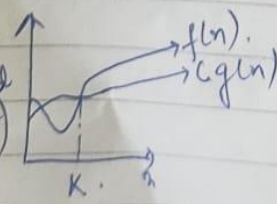
$f(n) = O(n^2)$

2) Big Omega (Ω) → Lower bound.

कम से कम कितना time लगेगा।

Best case

greatest Lower Bound (Atleast)



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

$f(n) \geq cg(n)$

$2n^2 + n \geq c(n^2)$

⇓

$c = 2$

n^2

n

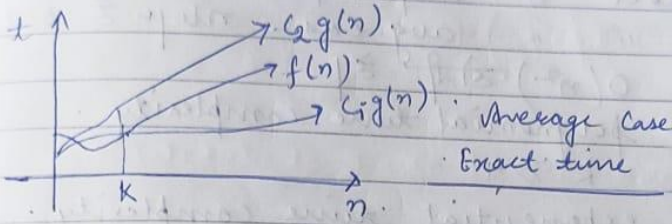
$\log n$

$\log(\log n)$

$2n^2 + n \geq 2 \cdot n^2$

$n \geq 0$

(8) Theta (Θ)



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

$2n^2 \leq 2n^2 + n \leq 3n^2$

Big (O)

Generally we measure worst case because it covers best & average case also.

Small (o) $\Rightarrow$ $f(n) < c g(n)$

Small (ω) $\Rightarrow$ $f(n) > c \cdot g(n)$.

Comparison of various time complexities :→

$O(c) < O(\log\log n) < O(\log n) < O(n^{1/2}) < O(n) < O(n \log n) < O(n^2)$

(O(c) = constant time; O(1) = constant time)

$< O(n^3) < O(n^k) < O(2^n) < O(n^n) < O(2^{2^n})$

Order → Atmost, Maximum, Upperbound.

O → Big O → max. time बताता है (worst case)

$O(1)$ $O(10^3)$ $O(1)$ $O(10^4)$ → no. fixed so $O(1)$ fixed $O(c)$

no one is bigger

Binary search → $O(\log n)$

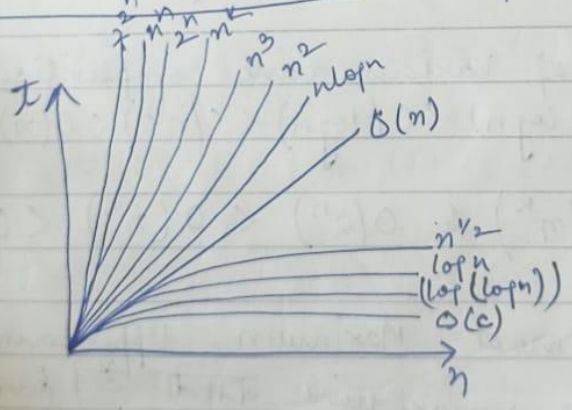
$O(n)$ → linear time complexity. (linear search → Best case)

$\log_2 1000$

$4 < 10 < 33 < 1000 < 10000$.

In sorting algo. mostly ~~worst~~ average case complexity is nlogn. and in some worst case also.
Merge sort # nlogn है।
Quick " " avg. case nlogn है but worst case # $O(n^2)$ होती है।
n^k → polynomial time complexity.
$k \geq 4$.
2^n → exponential time complexity.
↳ Dynamic problems →
↳ Matrix, cubic multiplication, Travelling salesman, sum of subset problems.
$O(n^n)$ → factorial time complexity.
$O(2^{2^n})$ → Doubles exponential time.

$O(2^n) \longrightarrow O(2^{n^2})$ — Exponential.



$f_1(n) = n^2 \log n$. $f_2(n) = n(\log n)^{10}$

$n = 16$

$f_1(n) = O f_2(n)$
$f_1(n) = \Omega f_2(n)$ $(16)^2 \log_2 16$ ⇒ $16(\log_2 16)^{10}$

$(16)^2 \times 4$ ⇒ $16 \times (4)^{10}$.

$n = 10^9$.

⇒ $(10^9)^2 \log_{10} 10^9$ ⇒ $10^9 (\log_{10} 10^9)^{10}$

$\Rightarrow 10^9 \times 10^9 \times 9$ $\Rightarrow 10^9 \times (9)^{10}$

$\Rightarrow 9 \times 10^9$ $\Rightarrow 9 \times (9)^9$

$\Rightarrow (10)^9 > \Rightarrow (9)^9$

OR

$n^2 \log n$ $n(\log n)^{10}$

$n \cdot n \log n$ $n \log n (\log n)^9$

n $(\log n)^9$

$\log n$ $\log (\log n)^9$

$\log n$ $9 \cdot (\log (\log n))$

Constant can be ignored.

$\log n > \log(\log n)$

So. $f_2(n) = f_1(n)$

$f_2(n) \leq c \cdot f_1(n)$.

Recurrence Relations :→

अपने आप को बार - 2 call करना (function).

10 20 30 40 50 60 70

i mid j

1 2 x/2 3 x/2

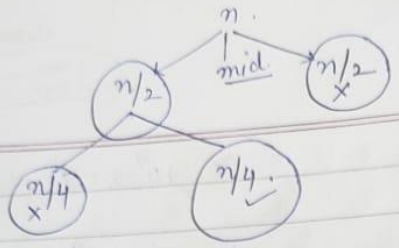
10 20 30 50 60 70

{ element को हमे search करना है but array sorted होना चाहिए }

```
BS (a, i, j, x)
    mid = (i+j)/2
40  if (a[mid] == x)
        return mid;
    else
40>30  if (a[mid] > x)
        BS(a, i, mid-1, x) else BS(a, mid+1, j, x).
```

30. $\frac{1+7}{2} = 4$

O(c) Constant time.



$n \to n/2 \to n/4 \to n/8$.

एक ही subproblem में interested हैं।

for finding middle, or comparison

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

Substitution method or Back Substitution method.

$$T(n) = \begin{cases} T(n/2) + c & \text{if } n > 1 \\ 1 & \text{if } n = 1 \end{cases}$$ — Recursive eqn.

If we have one element in array

$$T(n) = T(n/2) + c \quad — (1)$$

$$T(n/2) = T(n/4) + c \quad — (2)$$

$$T(n/4) = T(n/8) + c \quad — (3)$$

Back substitute (2) in (1).

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

$$= T\left(\frac{n}{2^2}\right) + 2c \quad \text{——}$$

$$T(n) = T(n/8) + 3c + 2c$$

Iterations

$$\Rightarrow T\left(\frac{n}{2^3}\right) + 3c \quad \text{——}$$

$$= T\left(\frac{n}{2^4}\right) + 4c \quad \text{——}$$

$$= T\left(\frac{n}{2^5}\right) + 5c \quad \text{——}$$

'k' times.

$\Rightarrow T\left(\frac{n}{2^k}\right) + Kc$

$T = 1$ लाने के लिए $n = 2^k$.

$\log n = \log 2^k$.

$\log n = k \log 2$

$\Rightarrow T\left(\frac{n}{n}\right) + k \cdot c$

$\Rightarrow T(1) + k \cdot c$

$\Rightarrow 1 + k \cdot c$.

$\Rightarrow 1 + \log n \cdot c$

$= O(\log_2 n) \Rightarrow$ { Time Complexity of this recurrence relation or BS }.

Substitution Method :-

जितनी भी recurrence relation की problems हैं can be solved by substitution method which is not posible in master theorem bcoz it can solve only some specific kinds of problems.

This always gives the correct soln.

Disad → Mathematical calculations ज्यादा हैं उसकी वजह से ये process as compared to others is somewhat slow.

$$T(n) = \begin{cases} 1 & \text{if } n = 1 \\ n * T(n-1) & \text{if } n > 1 \end{cases}$$

Base Condition / Anchor cond / Termination Cond. मतलब हम कहाँ पर stop करना है।

$T(n) = n * T(n-1)$ — (1)

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

$= (n-1) * T(n-2)$ — (2)

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

$= (n-2) * T(n-3)$ — (3)

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

$= n * n-1 * n-2 * T(n-3)$.

(n-1) steps.

Calculating iterations.

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

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

$\Rightarrow n * (n-1) * (n-2) * (n-3) - - - * 1$

$= n * (n-1) * (n-2) * (n-3) \text{ ------ } * 3 * 2 * 1$

$\Rightarrow n * n\left(1-\frac{1}{n}\right) * n\left(1-\frac{2}{n}\right) * \left(1-\frac{3}{n}\right) \cdots * n\left(\frac{3}{n}\right) * n\left(\frac{2}{n}\right) * n\left(\frac{1}{n}\right)$

$= O(n^n)$ factorial of n.

Q. $T(n) = \begin{cases} 1 & \text{if } n=1 \\ 2T(n/2) + n & \text{otherwise.} \end{cases}$

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

$T(n/2) = 2T(n/4) + \frac{n}{2} \quad - (2)$

$T(n/4) = 2T(n/8) + \frac{n}{4} \quad - (3)$

$T(n) = 2\left[2 \cdot T\left(\frac{n}{4}\right) + \frac{n}{2}\right] + n.$

$\Rightarrow 2^2 T\left(\frac{n}{2^2}\right) + 2n.$

$\Rightarrow 2^2\left[2T\left(\frac{n}{8}\right) + \frac{n}{4}\right] + 2n.$

$\Rightarrow 2^3 T\left(\frac{n}{2^3}\right) + 3n.$

$2^4 T\left(\frac{n}{2^4}\right) + 4n.$

!

k times.

$\Rightarrow 2^k T\left(\frac{n}{2^k}\right) + k \cdot n$

$\frac{n}{2^k} = 1$ or $n = 2^k.$

$\log n = \log 2^k$

$\log n = k \log 2$

$\boxed{k = \log n}$

$\Rightarrow 2^k T(n/n) + k \cdot n$

$\Rightarrow 2^k T(1) + k \cdot n$

$\Rightarrow 2^k + k \cdot n \Rightarrow n + n\log n$

$\Rightarrow O(n \log n)$ Dominating term.

$$T(n) = \begin{cases} 1 & \text{if } n = 1 \\ T(n-1) + \log n & \text{if } n > 1 \end{cases}$$

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

$T(n-1) = T(n-2) + \log(n-1) \quad -(2)$

$T(n-2) = T(n-3) + \log(n-2) \quad -(3)$

$T(n) = T(n-2) + \log(n-1) + \log n.$

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

k times.

$= T(n-k) + \log(n-(k-1)) + \log(n-(k-2)) + \cdots + \log n.$

$n - k = 1.$

$n = k.$

$T(n) = T(1) + \log(n-(n-1)) + \log(n-(n-2)) + \cdots + \log n$

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

$= 1 + \log(1) + \log(2) + \cdots + \log n.$

$= 1 + \log(1 + 2 + \cdots + n)$

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

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

$= 1 + n \log n.$

$= O(n \log n).$

$n! \Rightarrow \underbrace{n \times (n-1) \times (n-2)!}_{n \times n \times n \cdots} \Rightarrow n^n$

In worst case also $(n-1)(n-2)$ infinite times भी चलेगा तो n ही रहेगा।

data collection

facebook, youtube, instagram, google, amazon, ...

ata structure → Data organization, management & storage format.

कोई भी चीज़ जो आपके computer में है (Digital form) में song, dio file, video file, Excel file, Word file, Pdf file, Image वो data है ata को storage provide करना है ताकि हम future में उसे fficiently use कर पाएं, access in minimum time and can do odification.

acebook → structure → accordingly photos add, delete, access.

s → way to store data.

brary → Books (file) → data

Sectioning (acc. to subject).

Appl", Sfw, Website.

Google – Search – Thousands of People – Base DS.

Amazon —

Facebook — .

ray → C, C++, Java.

↳ Stack & Queues को implement करने के लिए भी use होता है।

finite ordered set of homogeneous elements.

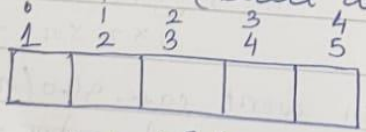
no. of elements fixed. similar data type.

element in sequence denoted by some index no.

Contigious memory location → लगातार

साथ में memory लेंगें (कोई कहीं - ऐसा नहीं होगा).

Random access → One by one elements को cross नहीं करना पड़ेगा (Direct access कर सकते हैं).



(location in memory).

int A[5]; → Total no. of elements.

array no.

element position उसके index no. से पता चलेगी।

जब कोई element define हो जाता है तो उसको एक space मिलता है अलग memory के अंदर । m/c dependent होता है कि उसको कितना size मिलेगा।

अगर 1 byte हो add उसमें acc होगा। 2 byte है। छोड़कर add मिलेगा।

Address

1000	1002	1004	1006	1008
0	1	2	3	4
1	2	3	4	5

integer – 2 byte. Computer.

Size of array = UB – LB + 1

= 5 – 1 + 1 = 5

or 4 – 0 + 1 = 5

Types of array :→ One dimensional.

→ Multidimensional. (array of arrays)

int a[5];

a				
0	1	2	3	4

a[1] ✓ accessing 2nd element.

2-D array → Row / Column. (Excel file, Screen, {Matrix; Tabular form में data enter होता है।

int a[3][4];

Row Column.

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

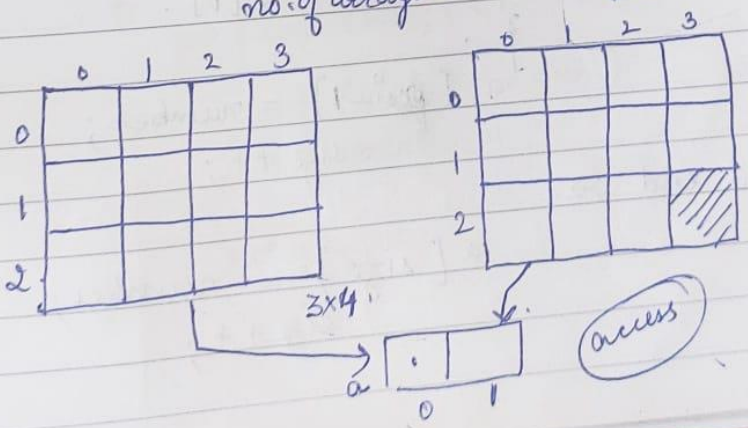
(3x4) Rows

a[0][0] . a[1][0] .

3-D. int a[2][3][4];

no. of arrays. Columns.

Total no. of elements. 2x3x4 ⇒ 24



a[1][2][3].

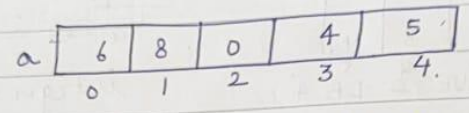
2nd array

memory के अंदर 1D array की form में ही store होते हैं।

Operations :→

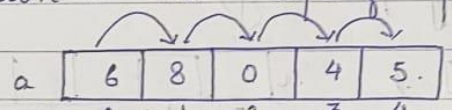
1) Traversal :→

size = 5.




```
for (i=0; i < size; i++).
    {
      print (a[i])
    }.
```

2) Insertion :→ . At a specific position.



no space.
↳ Overflow.
size → full.

```
if (pos <=0 || pos > size+1)
{ invalid position;
}.
else
    for (i = size-1 ; i >= pos-1 ; i--)
      {.
        a[i+1] = a[i] .
      }.
    a[pos-1] = number;
      size++;
```

In the beginning :→ .

```
for ( i = size-1; i >= 0 , i--)
    {
      a[i+1] = a[i].
    }.
  a[0] = number;
      size++;
```

Unsorted Array
O(1).
O(n).
O(n-p)
position

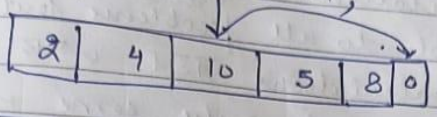
At the end :→ .

```
a[size] = number;
      size++;
```

Swapping →

a[size] = a[pos-1]. O(1)

a[pos-1] = num;



Traversal.

```
void traverse (int arr[], int n)
  {
    for (int i=0; i<n; i++)
      {
        print (arr[i], " ");
      }
  }.
```

O(n)

Access ⇒ arr[2] = 19. O(1).

arr[2] = 75 — Update (Random Access) O(1)

K^{th} element. arr[K-1] = new_value

(0 — K-1).

Search. :→

I/P :→ arr[] = {10, 30, 20, 90}.

num = 20.

O/P :→ (2)

no. not present (searching)

O/P → (-1).

```
int search (int arr[], int n, int num).
  {
    for (int i=0; i<n; i++).
      { if (arr[i] == num)
          return i;
      }
    return -1;
  }.
```

Worst Case. ⇒ O(n).

Best Case ⇒ O(1)

Imp

Master Method (Theorem)

↳ Uses recurrence relations to find their time complexity.

Substitution → applicable to all problems but it is slow because mathematical calculations बहुत ज्यादा होते हैं।

Master → faster method but only some recurrence relations can be solved by this method not all.

If $T(n) = aT(n/b) + f(n)$ } then this type of problem can be solved by this method.
$a \geq 1$ and $b > 1$

$T(n) = 1 \cdot T\left(\frac{n-1}{1}\right) + 1$ ✗

$T(n) = 8T\left(n/2\right) + n^2$ } ✓
$T(n) = T\left(n/2\right) + c$ }

Soln → $T(n) = n^{\log_b^a} [U(n)]$

$U(n)$ depends on $h(n)$

$h(n) = \frac{f(n)}{n^{\log_b^a}}$

Relation b/w $h(n)$ and $U(n)$ is ⇒ if

$h(n)$	$U(n)$
$n^\varepsilon, \varepsilon > 0$	$O(n^\varepsilon)$
$n^\gamma, \gamma < 0$	$O(1)$
$(\log n)^i, i \geq 0$	$\frac{(\log_2 n)^{i+1}}{i+1}$

$T(n) = 8T\left(n/2\right) + n^2$

$a = 8$, $b = 2$, $f(n) = n^2$.

b = base

$T(n) = n^{\log_b^a} U(n)$
$= n^{\log_2^8} \cdot U(n)$
$= n^3 \, U(n) \Rightarrow O(1)$
$= n^3 \cdot 1$
$= O(n^3)$

$T(n) = n^{\log_b^a}$
$= n^{\log_2^8}$
$= n^3$.

$O\left(n^{\log_b^a + \varepsilon}\right)$.

$O(n^3)$

Imp

Master Method (Theorem)

↳ Uses recurrence relations to find their time complexity.

Substitution → applicable to all recurrence relations problems but it is slow because mathematical calculations बहुत ज्यादा है।

Master → faster method but only some recurrence relations can be solved by this method not all.

If $T(n) = aT(n/b) + f(n)$ } then this type of problem can be solved by this method.
$a \geq 1$ and $b > 1$

$T(n) = 1 \cdot T\left(\frac{n-1}{1}\right) + 1$ ✗

$T(n) = 8T\left(n/2\right) + n^2$ } ✓
$T(n) = T\left(n/2\right) + c$ }

Soln → $T(n) = n^{\log_b^a} [U(n)]$

$U(n)$ depends on $h(n)$

$h(n) = \frac{f(n)}{n^{\log_b^a}}$

Relation b/w $h(n)$ and $U(n)$ is ⇒ if

$h(n)$	$U(n)$
n^x, $x>0$	$O(n^x)$
n^x, $x<0$	$O(1)$
$(\log n)^i$ $i \geq 0$	$\frac{(\log_2 n)^{i+1}}{i+1}$

$T(n) = 8T\left(n/2\right) + n^2$

$a = 8$, $b = 2$, $f(n) = n^2$.

$T(n) = n^{\log_b^a} U(n)$
$= n^{\log_2^8} \cdot U(n)$
$= n^3 \, U(n) \Rightarrow O(1)$
$= n^3 \cdot 1$
$= O(n^3)$

$T(n) = n^{\log_b^a}$
$\Rightarrow n^{\log_2^8}$
$\Rightarrow n^3$
$= \Theta(n^{\log_b^{a+0}})$
$= \Theta(n^3)$

$h(n) = \frac{n^2}{\log_2^8} = \frac{n^2}{n^3} = \frac{1}{n} = n^{-1}$ (2nd case)

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

$a = 1, \quad b = 2, \quad f(n) = c.$

$T(n) = n^{\log_b^a} U(n)$

$= n^{\log_2^1} U(n).$ $\quad \log 1 = 0$

$= n^0 U(n)$

$= 1 \cdot U(n) = U(n) \Rightarrow c \Rightarrow O(\log_2^n)$

$h(n) = \frac{f(n)}{n^{\log_b^a}} \Rightarrow \frac{c}{n^{\log_2^1}} \Rightarrow \frac{c}{n^0} = \frac{c}{1} = c.$

~~$O(\log_2 n)$~~

3rd case $\Rightarrow$ $(\log n)^0 \cdot c$ } $h(n)$

$(\log_2^n)^0 \cdot c$

$1 \cdot c$

$U(n) = \frac{(\log_2 n)^{0+1}}{0+1} \Rightarrow (\log_2 n) \cdot c.$

$= \log_2 n \cdot c = \log_2 n$

$$T(n) = \begin{cases} T(\sqrt{n}) + \log n & \text{if } n \geq 2 \\ O(1) & \text{else.} \end{cases}$$

$T(n) = aT(n/b) + n^k \log^p n$

$T(n) = T(\sqrt{n}) + \log n.$ } cannot be solved using master method b/c of <>

$n = 2^m$ } Assume.

$T(2^m) = T\left((2^m)^{1/2}\right) + \log 2^m$

$= T\left(2^{m/2}\right) + m \log_2^2$

$1.$

Assume $T(2^m) = S(m).$

$S(m) = S(m/2) + m^0$ power of m.

$a = 1, \quad b = 2, \quad k = 1, \quad p = 0$

I^st case

$a < b^k$

$1 < 2^1$

$1 < 2$

$n^k \log^p n$

$n^1 \log^0 n$

n

$n = 2^m$

$\log n = \log 2^m$

$= m \log 2$

$\log n = m$

$m = \log n$

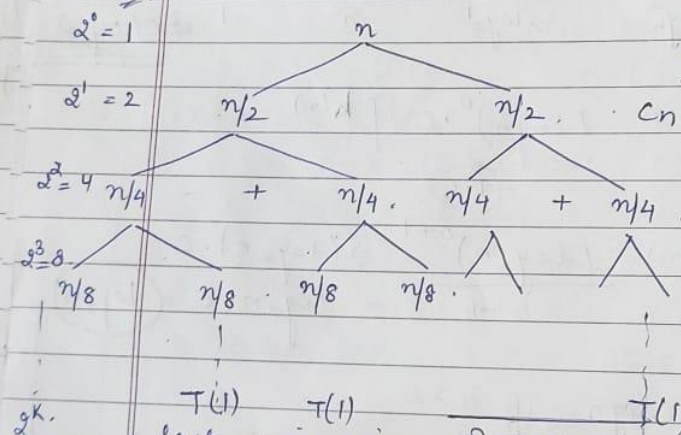
→ Divide & Conquer:

Problem → Divide → Subproblems → Solve → Combine – final Output.

इन methods में हम recursive tree method का use कर सकते हैं recurrence relations को Solve करने के लिए।

→ Recursive tree method :→

$T(n) = 2T(n/2) + Cn$ } Merge sort Recurrence Relation



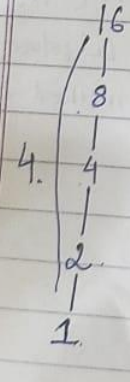
Divide + Combine करने में जो cost लगेगी ie $C(n)$ = Problem cost. $n = n$, $n/2 = n/2$

3Cn

↓

* Depends on the height of the tree.

last मेंn size कितना रह जाएगा ie (1) . (K.(n))



$\log_2 16 \Rightarrow (4)$

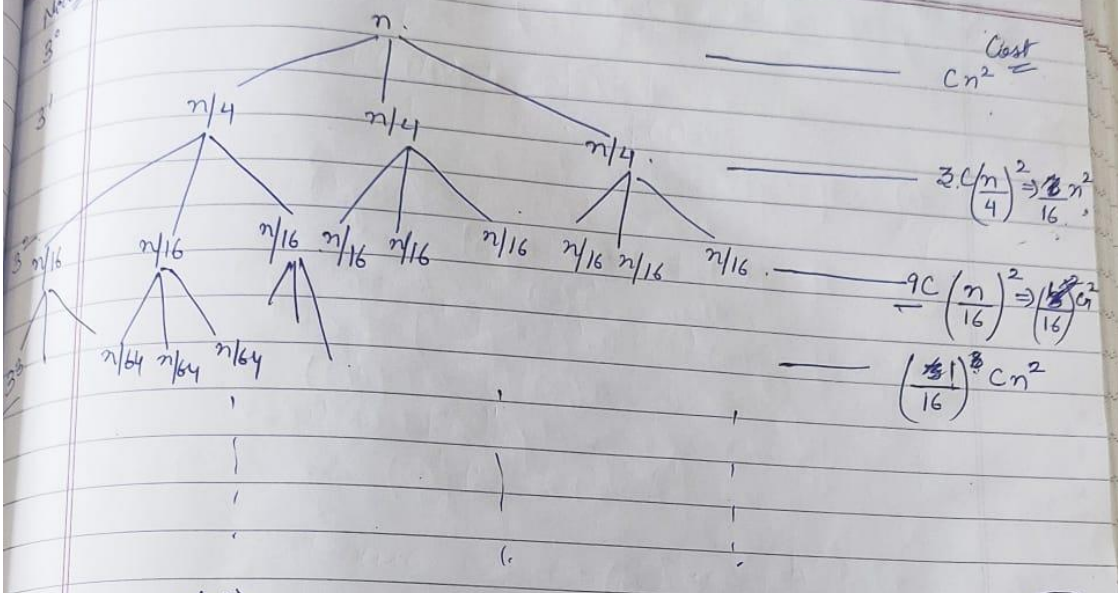
Height is $\log n$.

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

Constant

$T(n/2^k) = T(1)$

$T(n) = 3T(n/4) + cn^2$



T(1) T(1) T(1)

$3^k \cdot \frac{n^2}{4^k}$

16 / 4 / 1 — 2 steps / Height

$\log_4 16 \Rightarrow \log_4 n$

$$cn^2 + \frac{3}{16}cn^2 + \left(\frac{3}{16}\right)^2 cn^2 + \left(\frac{3}{16}\right)^3 cn^2 + \text{-----} + 1$$

$$cn^2\left[1 + \frac{3}{16} + \left(\frac{3}{16}\right)^2 + \left(\frac{3}{16}\right)^3 + \text{-----}\right]$$

↓,

GP series.

$\frac{1}{1-x}$ $x < 1$.

$$cn^2\left[\frac{1}{1-3/16}\right] \Rightarrow cn^2\left[\frac{16}{13}\right] \Rightarrow \underline{\theta(n^2)}.$$

Master Method (Theorem).

→ $T(n) = aT\left(n/b\right) + f(n)$.

where $a \geq 1$, $b \geq 1$.

$f(n) \geq 0$

① If $f(n) = O\left(n^{\log_b^{a-\epsilon}}\right)$, $\exists \epsilon > 0$

then $T(n) = \theta\left(n^{\log_b^a}\right)$

② If $f(n) = \theta\left(n^{\log_b^a}\right)$

then $T(n) = \theta\left(n^{\log_b^a} \log n\right)$

③ If $f(n) = \Omega\left(n^{\log_b^{a+\epsilon}}\right)$ $\exists \epsilon > 0$.

then $T(n) = \theta(f(n))$

iff the regularity condition holds.

$a \cdot f\left(n/b\right) \leq c f(n)$ for $c < 1$

Q. $T(n) = 4T\left(\frac{n}{2}\right) + n$.

$a = 4, b = 2, f(n) = n$.

$n^{\log_b^a} = n^{\log_2^4} = n^{\log_2 2^2} = n^2$.

Comparing $g(n) > f(n)$.

$f(n) = O\left(n^{\log_b^{a-\epsilon}}\right)$ $\epsilon = 2 - 1 = 1$.

1st Case

$T(n) = \theta\left(n^{\log_b^a}\right) = \underline{\underline{\theta(n^2)}}$.

Q. $T(n) = T\left(\frac{n}{2}\right) + 1$.

$a = 1, b = 2, f(n) = 1$

$n^{\log_b^a} = n^{\log_2^1} = n^0 = 1$

$f(n) = g(n)$.

$f(n) = \theta\left(n^{\log_b^a}\right)$.

Acc to 2^{nd} → $T(n) = \Theta(n^{\log_b a} \log n)$.
$T(n) = \Theta(\log n)$.

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

$a = 2$, $b = 2$, $f(n) = n^4$,
$n^{\log_2 2} = n$.

$f(n) = \Omega(n^{\log_b a + \epsilon})$, $\exists \epsilon > 0$.

Acc to 3^{rd}: $a \cdot f\left(\frac{n}{b}\right) \le c f(n)$.

~~$2 f\left(\frac{n}{2}\right)$~~ $2 \frac{n^4}{16\ 8} \le c n^4$.

$\frac{1}{8} \le c \le 1$.

$g(n) < f(n)$
$T(n) = \Theta(f(n)) = \Theta(n^4)$.

polynomial fnc → n की some power.
exponential → logarithmic होता है।

f(n) - polynomially बड़ा नहीं है logarithmically बड़ा है।

$T(n) = a T\left(\frac{n}{b}\right) + f(n)$, $a > 0$, $b > 0$.

If $f(n) = \Theta(n^{\log_b a} \log^{k} n)$

then $k \ge -1$, then $\Theta(n^{\log_b a} \log^{k+1} n)$.

$k = -1$, then $\Theta(n^{\log_b a} \log\log n)$

$k < -1$, then $\Theta(n^{\log_b a})$ ~~[illegible]~~.

Q. $T(n) = 2T\left(\frac{n}{2}\right) + n\log n$.

$a = 2, \; b = 2, \; f(n) = n\log n$.

$g(n) \Rightarrow n^{\log_b a} \Rightarrow n^{\log_2 2} = n$.

$f(n) = g(n)\log^k n$.

$f(n) = g(n)$ multiplied by some power of log.
then $f(n)$ is not polynomially larger than $n\log n$
it is logarithmically larger.

$f(\log n) = \log n$.

$\epsilon \Rightarrow n$ की power something.
जो यहाँ 0 से बड़ा है। it can be zero
$\epsilon > 0$

$f(n)$ में $\log^k n$. (k की value देखनी है).

$T(n) = \Theta\left(n^{\log_b a} (\log)^k n\right)$

$k = 1$.

$\Rightarrow \Theta\left(n^{\log_b a} \log^{k+1} n\right)$

$= \Theta\left(n^{\log_2 2} \log^{1+1} n\right)$.

$T(n) = \Theta\left(n\log^2 n\right)$

Q. $T(n) = 4T\left(\frac{n}{2}\right) + n^2 \log n$.

$a = 4, \; b = 2, \; f(n) = n^2 \log n$.

$g(n) = n^{\log_2 4}$

$\Rightarrow n^2$. $\quad f(n) \neq \Omega\left(n^{\log_b a+\epsilon}\right)$.

$T(n) = \Theta\left(n^2 \log^2 n\right)$

$T(n) = 2T\left(\frac{n}{2}\right) + \frac{n}{\log n}$.

$a = 2, \quad b = 2, \quad f(n) = \frac{n}{\log n}$.

$g(n) = n^{\log_b a}$.

$= n^{\log_2 2} = n$.

$k = -1$.

$T(n) = \Theta(n \log\log n)$

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

$a = 1, \quad b = 2, \quad f(n) = 2^n$.

$g(n) = n^{\log_b a} \Rightarrow n^{\log_2 1} \Rightarrow n^0 = 1$

$f(n) > g(n)$.

3rd Case.

$a \cdot f\left(\frac{n}{b}\right) \leq c f(n)$.

$1 \times 2^{n/2} \leq c\, 2^n$.

$2^{-n/2} \leq c \leq 1$.

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

$T(n) = 10T\left(\frac{n}{3}\right) + 17n^{1.2}$

$a = 10, \quad b = 3, \quad f(n) = 17n^{1.2}$.

$g(n) = n^{\log_b a}$.

$= n^{\log_3 10}$.

Comparing $f(n)$ & $g(n)$.

$n^{\log_3 9} = n^2$

$g(n) > f(n)$.

$T(n) = \Theta(n \log_3 10)$

8. $T(n) = 2T\left(\frac{n}{4}\right) + \sqrt{n}$.

$a = 2, \quad b = 4, \quad f(n) = \sqrt{n}$ or $n^{1/2}$.

$g(n) = n\log_b^a$.

$\Rightarrow n\log_4^2 \Rightarrow n^{1/2}$.

base perfect square that is why power 1/2

$f(n) = g(n)$.

$T(n) = \Theta(n\log_b^a \log n)$.

$= \Theta(n^{1/2} \log n)$ or $\Theta(\sqrt{n} \log n)$.

$\log_4^2 = x$.

$\log_b^a = y \quad (b^y = a)$

$4^x = 2$.

$x = 1/2$

9. $T(n) = 2T\left(\frac{n}{2}\right) + \frac{n}{\log^2 n}$.

$a = 2, \quad b = 2, \quad f(n) = n\log^{-2} n$.

$g(n) = n\log_2^2 \Rightarrow n$.

$k < -1$

$\Theta(n\log_b^a) = \underline{\Theta(n)}$

10. $T(n) = T\left(\frac{9n}{10}\right) + n$.

$a = 1, \quad b = \frac{10}{9}, \quad f(n) = n$.

$f(n) = -ve$, a = fraction } master method fail.

$g(n) = n^{\log_b a} = n^{\log 1/4} \Rightarrow n^0 = 1$

$f(n) > g(n)$.

~~$T(n) =$~~

$a f\left(\frac{n}{b}\right) \leq c f(n)$.

$1 \cdot \frac{n}{10} \leq c \cdot n$

$\frac{9}{10} \leq c \leq 1$

$T(n) = \Theta(f(n)) = \Theta(n)$.

Arrays :→

Insert-

I/P → arr[ ] = {10, 30, 40}.

n = 3.

num = 20.

pos = 2. index = 1. (10, 20, 30, 40)

O/P → n = 4;

{10, 30, 40, ___, ___}

array में कितने max elements आ सकते हैं।

```
int insert (int arr[ ], int n, int num, int pos,
            int max_limit)
{ if (n == max_limit)
      return n;
  int index = pos-1;
  for (i = n-1; i >= index; i--)
      arr[i+1] = arr[i];
  arr[index] = num;
  return n+1;
}
```

Worst case :→ O(n).

Delete :-

I/P :- arr[] = {10, 20, 30, 40}.
n = 4
num = 20.
O/P → 3.

index पता होना चाहिए

10, 30, 40

15 not present

```
int delete (int arr[], int n, int num)
{
    int i;
    for (i=0; i<n; i++)
    {if (arr[i] == num)
        break;
    }
    if (i == n)
        return n;
    for (int j = i; j < n-1; j++)
    {
        arr[j] = arr[j+1];
    }
    return (n-1);
}
```

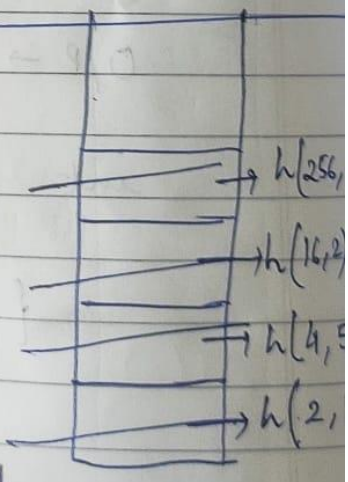
10, 20, 30, 40

O(n)

```
int myPow(double x, int n).
```

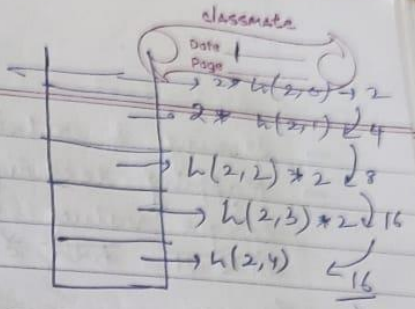
Lab :- power (double x, int n)

```
helper (double x, int n).
{
    if (n==0) return 1.;
    if (n==1) " x;
    if (n%2 == 0)
        return power (x*x, n/2).
    return x * power (x*x, n/2);
}
}
```



(2^4)

```
power (double x, int n).
{ if (n <= 0)
    { return 1;
    }
  return x * power (x, n-1);
}.
```




```
ans = 1
for (i = 1 ; i <= n ; i++)
    ans = ans x x
```
} O(n).

$5^{10} = 5^5 \times 5^5$. if n == even.

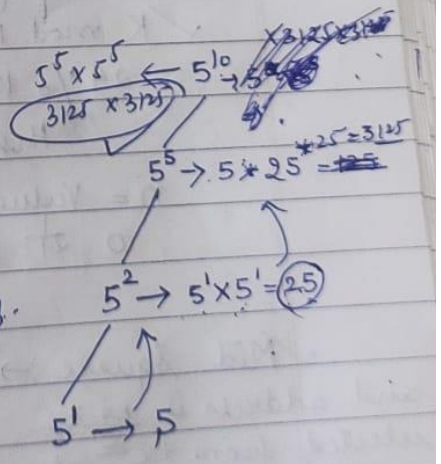
$5^{11} = 5^5 \times 5^5 \times 5$.

↑ temp

-ve

if n < 0.
1/power (x, n)

```
power (x, n)
  if (n == 1)
      return x;
  if (n == 0)
      return 1;
temp = power (x, n/2)
   if (n % 2 == 0)
      return temp * temp.
   else
      return x * temp * temp.
```



∴

5 X 5 X 5 X 5 X 5 X 5 X 5 X 5

625. 625 X25.

Hashing

→ Storing & retrieving data in a database in O(1) time.

→ Mapping technique.
Larger values को smaller values में map करने की कोशिश करते हैं by using hashing.

Terms:
Search key → In database, data is searched using some key. eg:- Student database, RollNo, Reg. No. — key.

→ Stored in hash table.
Record → बहुत सारी चीजे होती हैं जो कि hash table में नहीं डाल सकते that is why we use only search key to be stored into the hash table.

→ Hash Table → DS for storing the data in a proper way. Similar to array, indexing is there but insertion, searching & deletion can be done in O(1) time (no need of scanning) using hash function.

→ Hash fnc → (K mod 10, K mod n, Mid square, folding method).

Search key (24, 52, 91, 67, 48, 83).

✓ K mod 10.

24 % 10 → 4.

Hash Value = 4

n = Value. Given होगी.
0 से लेकर (n-1) का table बनालो।

	0
91	1
52	2
83.	3
24.	4
	5
	6
67	7
48	8
	9

Hash Table.

✓ Mid Square :→ Take square of ~~middle~~ Key. address is ~~125~~ ~~value & store it in~~ ed from the ~~=~~ ~~4~~. ~~the table~~. le of the square no.

✓ Folding Method :→ Value store करनी है 123456.

Table → 0 to 999

123
456
579. Store at this location.

inter point to the database. then you can check the record
re there.

Radix Sort :→

It is a non comparison based algorithm. Used for sorting.

Comparison algo (Selection, Insertion, Bubble, Merge sort, Quicksort).

① 904, 046, 005, 074, 062, 001 { Append zero so that all nos. become equal to the largest digit no.)

Radix means base.

means we will focus on least significant bit ie unit place. (arrange)

②	③	④
001	001	001
062	904	005
904	005	046
074	046	062
005	062	074
046	074	904

(position change नहीं करनी) 904, 074 } जब भी same element होंगे।

Array sorted.

Time Complexity :- $O(d * (n+k))$ (d = no. of digits)

```
radixsort (a, n)
{
    int max = getmax (A, n)
    for (pos = 1 ; max/pos > 0 ; pos * 10)
    {
        countsort (A, n, pos)
    }
}
Countsort (int A[ ], int n, int pos).
{   int count[10] = {0};
```

max ⇒ 677

pos ⇒ 1 10 100

67, 6.

1000 = 0. } fail

Counting Sort :→

→ Non comparison based algorithm :-
given input size is 'n' and range is 'K'. (1-5).
means elements that you have to sort is in a given range
इससे बाहर नहीं होंगे। (Constraint or restriction).
→ Take size equal to range or the highest element value in the array.
→ Counting means you have to count the occurrence of that element.

'n' → 2 1 2 3 1 2 4. (Value)

1	✗2
2	✗✗3
3	1
4	1
5	0

} occurrence.

Array को traverse करो और जो जितनी बार occur हो रहा है उसे लिख दो।

1 1 2 2 2 3 4 →.

Time Complexity → depends upon input size 'n' and range 'K' ie. O(n+K)
Space Complexity → Array (Keys जिन्हें sort करना है) extra space is taken by range 'K' ie. O(n+K).
Linear time में work कर रहा है पर disadvantage कि range से बाहर नहीं जा सकते।
ie. 2 20000 3 6 7
↓
extra space लेना पड़ेगा इस no. की वजह से range 20,000 तक लेनी होगी।

When range is defined then counting sort is best algorithm.

Date
Page

~~Bucket Sort~~ :-

i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	
a	2	1	1	0	2	5	4	0	2	8	7	7	9	2	0	1	9	n=17, k=9

(Index represents element)

Index →	0	1	2	3	4	5	6	7	8	9
Count	3	3	4	0	1	1	0	2	1	2

occurrences of key values.

```
for (i = 0 ; i < n ; i++)
{
    count [a[i]] = count[a[i]] + 1;
}
```

(Index represents Element)

Index →	0	1	2	3	4	5	6	7	8	9
Count	3	6	10	10	11	12	12	14	15	17

(position till where 1 will occur)

```
for ( i = 1 ; i <= k ; i++)
{
    count[i] = count[i] + count [i-1] ;
}
```

Another array b we are taking which is of same size as that of a and it will be the sorted array & we are going to start from the back. just to maintain the stability of the sort.

	0	1	2	3	4	5	6	7	8	9	
o/p	~~3~~	~~5~~ 6	~~8~~ 10	~~10~~	10	11	12	~~14~~	14	~~15~~	15

b ⇒	0	0	0	1	1	1	2	2	2	2	4	5	7	7	8	9	9
	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16

```
for ( i = n-1 ; i >= 0 ; i--)
{
    count [a[i]] = count [a[i]] - 1 ;
    b [count [a[i]]] = a[i]
}
```

```
for (i = 0 ; i < n ; i++)
{
    a[i] = b[i] ;
}
```

Drawback → a = 100.

Count: 0 K = 10000 ... 10000

K = O(n). no. of elements.

2n, 5n, 6n.

≠ n^2

cannot be applied to -ve or floating point values.

some modifications can be done and then applied.

(-5)(+5) -5, 1, 0, 6, 9 → 1, 0, -5.

Directly we cannot apply.

Diff. elements with diff. ranges (any range).

652, 0, 999, 1111, 10000, 7, 432.

It is better to apply radix sort in this case. Radix sort also uses counting sort but it is applied on digits. Then K values would be (0 to 9)

0	1	2	3	4	5	6	7
2	5	3	0	2	3	0	3

n = 8.

K = 0 to 5.

Index → (K+1)

C

0	1	2	3	4	5
2	0	2	3	0	1

C

0	1	2	3	4	5
2	2	4	7	7	8

C

0	1	2	3	4	5
0	2	4	5	7	8

B

0	1	2	3	4	5	6	7
0	0	2	2	3	3	3	5

Sorted array

Radix Sort continued :→

530/10 ⇒ 53%10 = 3
530/100 ⇒ 5%10 ⇒ 5

```
for (i=0; i<n; i++)
    count[(a[i]/pas)%10] = count[(a[i]/pas)%10] + 1;

for (i=1; i<=K; i++)
    count[i] = count[i] + count[i-1];

for (i=n-1; i>=0; i--)
    b[count[(a[i]/pas)%10] = a[i]
    count[(a[i]/pas)%10] = count[(a[i]/pas)%10] - 1;

for (i=0; i<n; i++)
    a[i] = b[i];
}
```

a :

0	1	2	3	4	5
904	046	005	074	062	001

Pass 1

Count

0	1	2	3	4	5	6
0	1	1	0	2	1	1

Count

0	1	2	3	4	5	6
0	1	2	2	4	5	6

b ⇒

0	1	2	3	4	5
001	062	904	074	005	046

Pass 2

Count

0	1	2	3	4	5	6	7
1	0	0	0	1	0	1	1

Count

0	1	2	3	4	5	6	7
0	3	3	3	4	4	5	6

b ⇒

0	1	2	3	4	5
001	904	005	046	062	074

Pass 3

Count -

0	1	2	3	4	5	6	7	8	9
5	0	0	0	0	0	0	0	0	1

Count ⇒

0	1	2	3	4	5	6	7	8	9
5	5	5	5	5	5	5	5	5	6

b ⇒

0	1	2	3	4	5
001	005	046	062	074	904

✓ sorted array

Bucket Sort :→

→ Non comparison based algorithm.
→ Works on floating point nos. between range 0.0 to 1.0
→ I/P's should be uniformly & independently distributed across [0,1] to get a running time of O(n).

0.79, 0.13, 0.64, 0.39, 0.20, 0.89, 0.53, 0.42, 0.06, 0.94

0.06, 0.13, 0.20, 0.39, 0.42, 0.53, 0.64, 0.79, 0.89, 0.94

0	→ 0.06
1	→ 0.13
2	→ 0.20.
3	→ 0.39
4	→ 0.42
5	→ 0.53
6	→ 0.64
7	→ 0.79
8	→ 0.89
9	→ 0.94.

Uniformly Distributed.
$O(1) \times n = O(n)$
{Best Case, Avg. Case}

0.79×10.
$\lfloor 7.9 \rfloor = 7$

```
Bucket-Sort (A)
1) Let B[0 ---- n-1] be a new array.
2) n = length[A]. or n = A.length
3) for i = 0 to n-1
4) make B[i] an empty list.
5) for i = 1 to n.
6) insert A[i] into list B[⌊nA[i]⌋]   B[7].
7) for i = 0 to n-1
8) sort list B[i] with insertion sort.
9) Concatenate lists B[0], B[1], --B[n-1]
   together in order.
```

If the elements are like this 0.79, 0.78, 0.795, 0.74...
(Not uniformly distributed).

0
1
2
3
4
5
6
7 → .74 → .78 → .79 → .795
8
9.

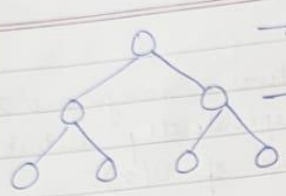
Worst case:
↑
$O(n^2)$

अगर सारे elements यहीं पे आ जाते n. in insertion sort and insert elements
⇒ n for $n \times n = O(n^2)$.

.78 → .79
.79 → .78
Compare & sort in insertion sort.

classmate
Date
Page

Trees → Non linear DS. (Recursive DS.)



→ Root - (Top most element) } hierarchical form. + multiple links (stored)
→ Internal node (Non leaf node) } storing some information + containing links to other nodes }
→ Leaf node. (External nodes)

→ It does not have any cycles or loops like graphs. (Graphs ≠ source and destination node) not root.

Trees can be defined as a collection of entities (nodes) linked together to simulate a hierarchy.

→ Parent → Immediate predecessor of that node.
→ Child :→ " successor " "
→ Path :→ Sequence of consecutive edges from source node to destination node.
→ Subtree :→ Containing a node with all its descendants.
→ Sibling :→ Children of same parent.
Degree → Degree of a node. No. of children of that node.
Degree of a tree → Max. degree ~~of any node in a tree~~. among all nodes. [Max. child of all nodes is not more than 3 so degree of tree = 3].
Depth of a node → Length of path from root to that node. (No. of edges). Depth of root is always zero (0)

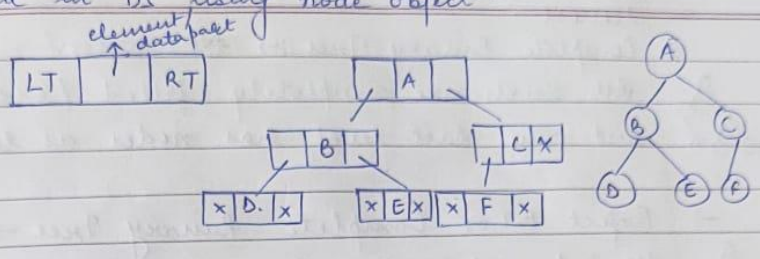
Height of node → No. of edges in the longest path from that node to the leaf.

Height of a node may or may not be equal to the depth of a node.

Height of a tree = Height of a root node = Level of a tree
Level of a node ⇒ Depth of a node.

n nodes = (n-1) edges

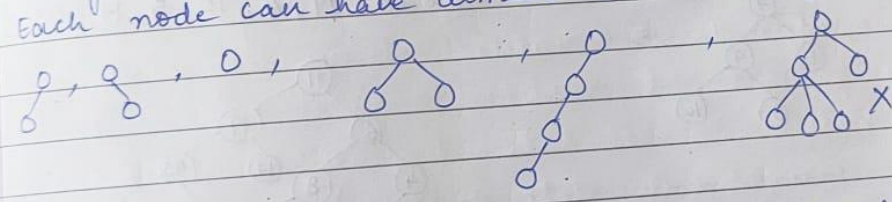
Tree represented in DS using node object.




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

Binary Tree :-

→ Each node can have atmost 2 children. {0, 1, 2}.

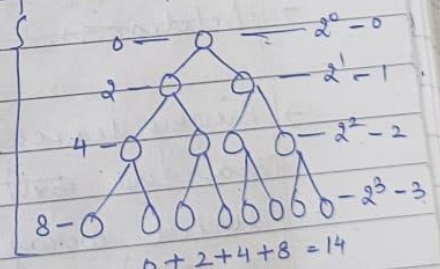


Property → At each level i, max no. of nodes possible is 2^i.

Max. no. of nodes of height h ⇒.

$2^0 + 2^1 + 2^2 + \dots + 2^h$

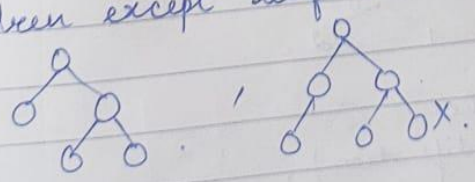
$\Rightarrow (2^{h+1} - 1)$



Min. no. of nodes of height h ⇒ h+1

Full / Strict / Perfect (BT.)

→ BT where each node have either 0 or 2 children or exactly 2 children except leaf node.

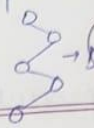


No. of leaf node = No. of internal nodes + 1.

Degenerate BT → All internal nodes have only one child.


→ Left skewed BT.


→ Right skewed BT.

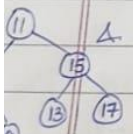

→ ~~Degenerate Tree~~ Skewed BT.

~~Almost~~.

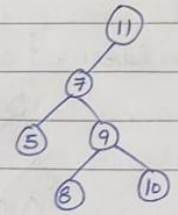
→ Complete Binary Tree :- ~~No holes.~~ All levels are completely filled (except the last level leaf node) and the last level has nodes as left as possible.

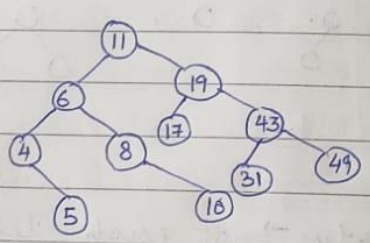
→ Perfect BT or Complete Binary Tree → no holes.
All internal nodes have 2 children and all leaves are at same level.

Binary Search Tree :-
→ Binary Tree — atmost 2 children.
→ Values of left subtree of any node should be less than that node
& " " right " " " more " " ".



Operations :- Insertion.
11, 6, 8, 19, 4, 10, 5, 17, 43, 49, 31.

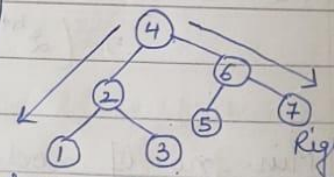




~~→ Deletion :- 3 options.~~

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

→ Binary Search Tree का आगर inorder traversal करते हैं तो वो एक sorted array देगा that means कि आपने BST सही draw किया है।

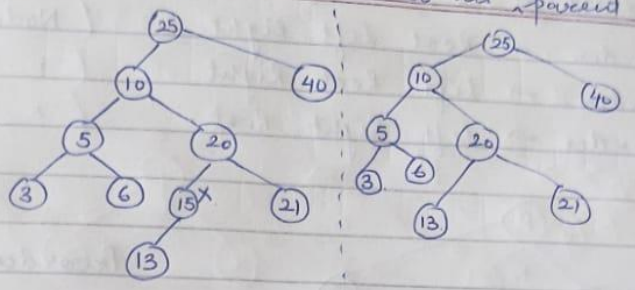


→ Deletion → 3 Cases :-

Leaf
↓
0 child
↓
simply delete it (no modification).

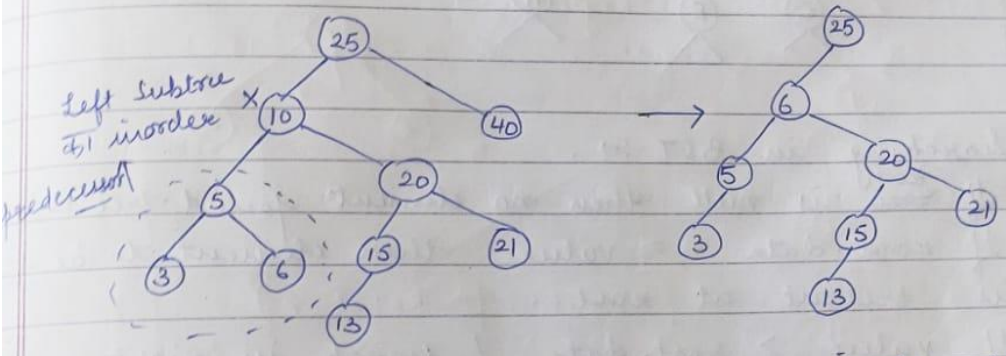
Non leaf.

1 Child → Delete & link it to its grandparent.

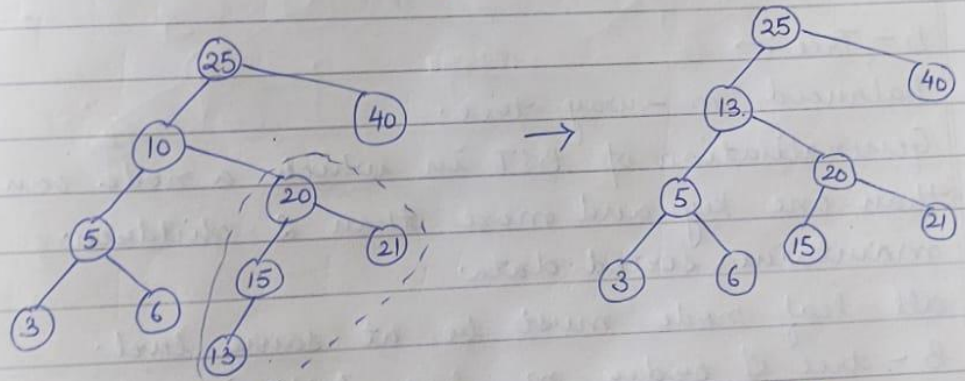


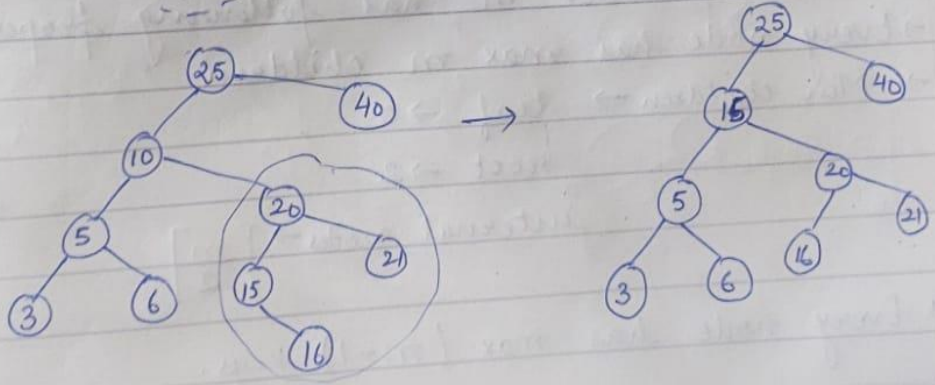
2 Child :→ 2 Cases. left subtree.

(1) Replace with the inorder predecessor.



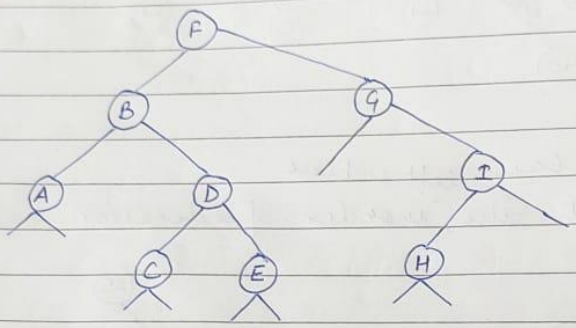
(2) Replace with right subtree inorder successor. smallest element.





Traversal :→ 3 techniques :-

Preorder → Root Left Right. (Node – 1st time)
Inorder → Left Root Right. (" – 2nd ")
Postorder → Left Right Root. (" – 3rd ")



Preorder → F B A D C E G I H
Inorder → A B C D E F G H I
Postorder → A C E D B H I G F

Searching in BST :→.

* If ~~root~~ root is null then no element found tree is empty.
* If root.data == value then element to be searched is present at root.
* If value > root.data (search in right subtree else search in left subtree).

B – Tree :→

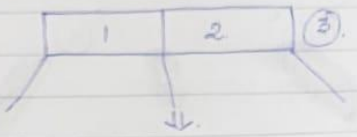
→ Balanced m – way tree.
→ Generalisation of BST in which a node can have more than one key and more than 2 children.
→ maintains sorted data.
→ all leaf node must be at same level.
→ B – tree of order m has following properties.
→ Every node has max m children.
→ Min children → leaf → 0
root → 2
internal nodes → $\lceil \frac{m}{2} \rceil$
→ Every node has max. (m – 1) keys.

→ Min keys :- root node → 1

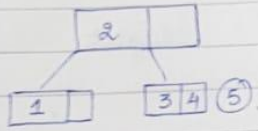
all other node → $\lceil \frac{m}{2} \rceil - 1$

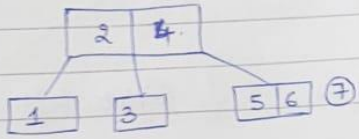
Insertion will always be in the leaf node.

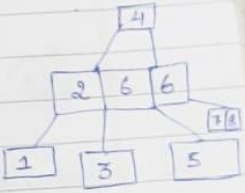
$m = 3$.

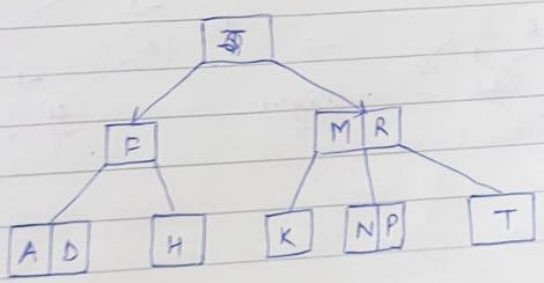


max key = (3-1) = 2









Searching, Insertion, Deletion, Traversing — $O(\log n)$ Average Case
similar to BST. Worst case — $O(n)$.

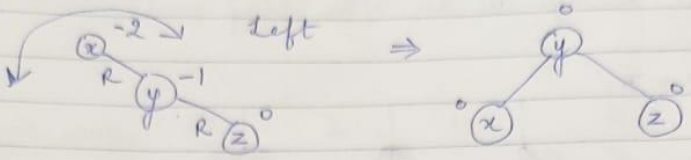
AVL :→ It is a BST. (Self balanced tree)

$|\text{height of left subtree} - \text{height of right subtree}| = \{0, 1, -1\}$

↑ Balance factor.

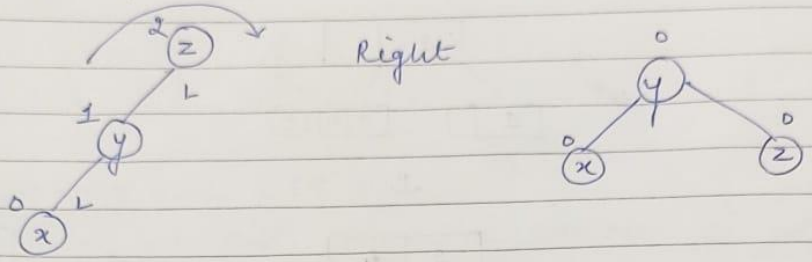
4 Rotations :

① x, y, z.

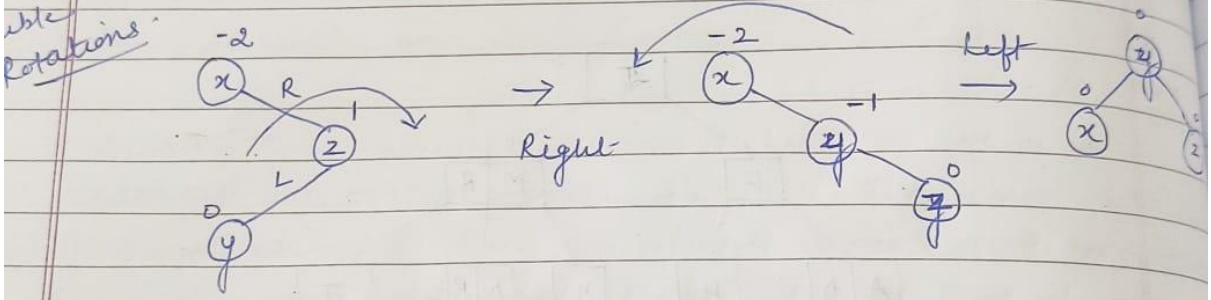


Single Rotation

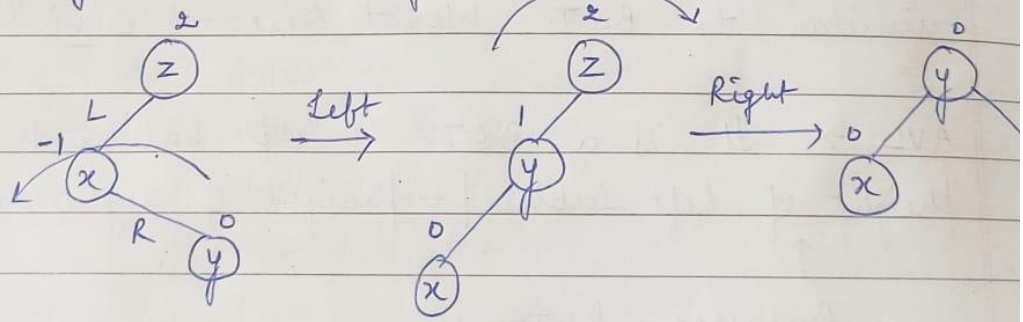
② z, y, x.



③ x, z, y. (Right Left Rotation)



z, x, y (Left Right Rotation)



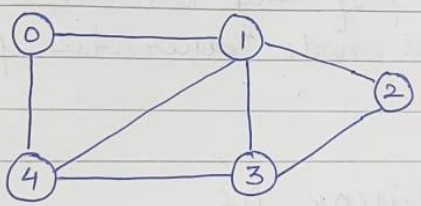
Red - Black Trees :-

→ It is a self balancing BST.
→ Every node is either Black or red.
→ Root is always black.
→ Every leaf which is nil is black.
→ If node is red then its children are always black.
→ Every path from node to any of its descendant nil node has same no. of black nodes.

In AVL - Searching faster.
↳ Strictly height balanced tree.
but no. of rotations are more.

In Red Black Trees → Roughly Self balanced trees.
Insertion & deletion easy.
max. 2 rotations or recolouring is required.

Graphs :→
A graph consists of a finite set of vertices (or nodes) & set of edges which connect with a pair of nodes.

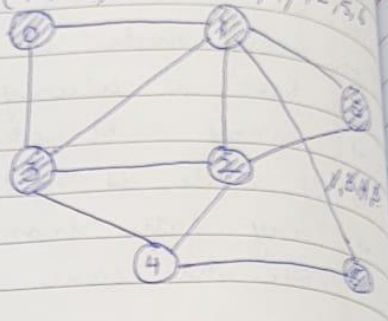


Representation of Graphs :→ Removing & Searching takes O(1) time.
Adjacency Matrix — Adv :→ Representation is easier to implement. Disad. →. Consumes more space, Adding a vertex is $O(V^2)$ time.
Adjacency List —
Adv :→ Saves space $O(V+E)$, Adding a vertex is easy.
Disad :→ Searching takes times $O(V)$.

Graph Traversals :-

BFS (Breadth first search).
Queue data structure is used (FIFO)

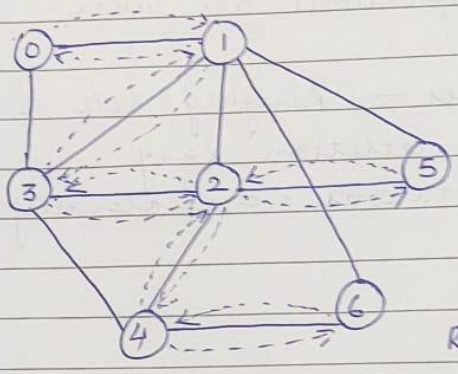
Queue :- | 0 | 1 | 3 | 2 | 5 | 6 | 4 |

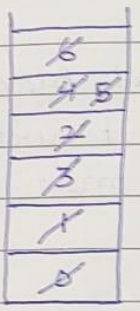


Result :- 0 1 3 2 5 6 4

(Check adjacent nodes)

DFS (Depth first search) {also known as level order traversal}
Stack data structure is used (LIFO).



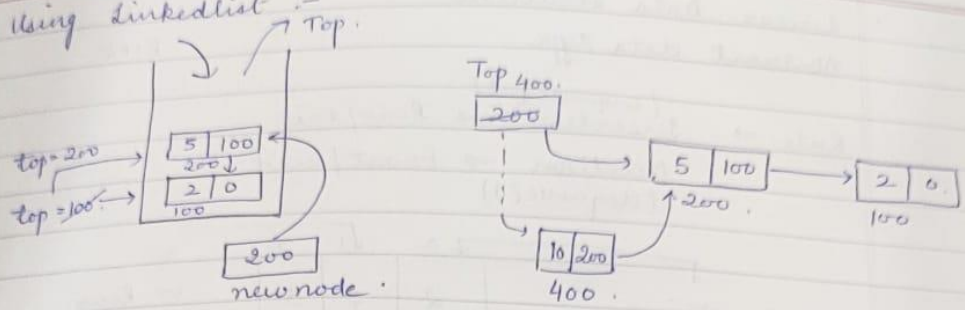


Result :- 0, 1, 3, 2, 4, 6, 5.

After 6 we cannot go anywhere so we will backtrack and for that we will pop out the elements from the stack. (If any unvisited node is there then go to that node otherwise pop it out from the stack).

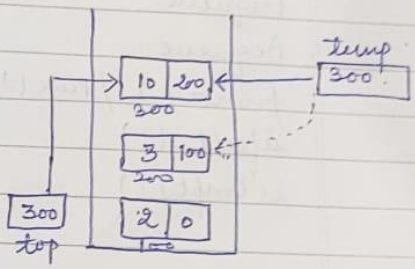
Stacks :- Linear DS.
LIFO or FILO.
Implemented using Stacks & Linkedlist.
Operations → Push :- Insert.
Pop :- Delete.
top / peek → Return topmost element without removing it.
isempty, isfull.

Using Linkedlist :-



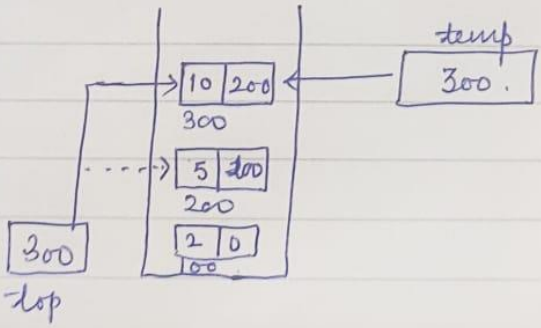
Display :-

```
display ()
{
  struct node * temp ;
      temp = top ;
  if ( top == 0)
    {
      ____
    }
  else
    {
      while(temp != 0)
        {
          printf ("%d", temp -> data);
            temp = temp -> link ;
        }
    }
}
```



10 3 2

pop :-




```
void pop()
{
struct node * temp;
  temp = top ;
  if (top == 0)
    { ____ }
else
  { printf("%d", top->data);
    top = top -> next ;
      free (temp);
} }.
```

Queue :→

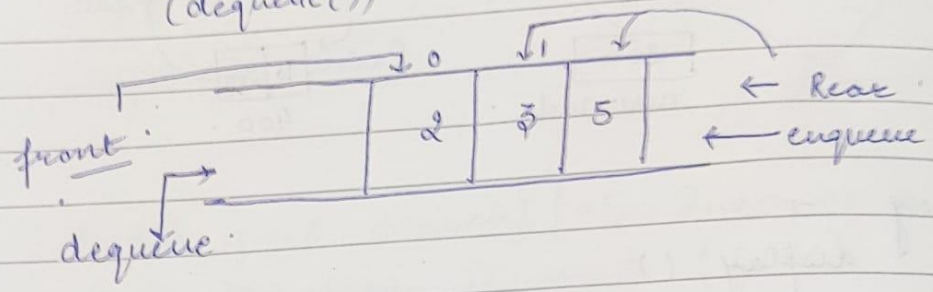
Linear Data Structure.

Abstract data type.

FIFO.

Rule → Insertion (enqueue()) → Rear/tail

Deletion (dequeue()) → Front/head.



Operations :-

Enqueue

Dequeue

front() / Peek()

isfull()

isEmpty()