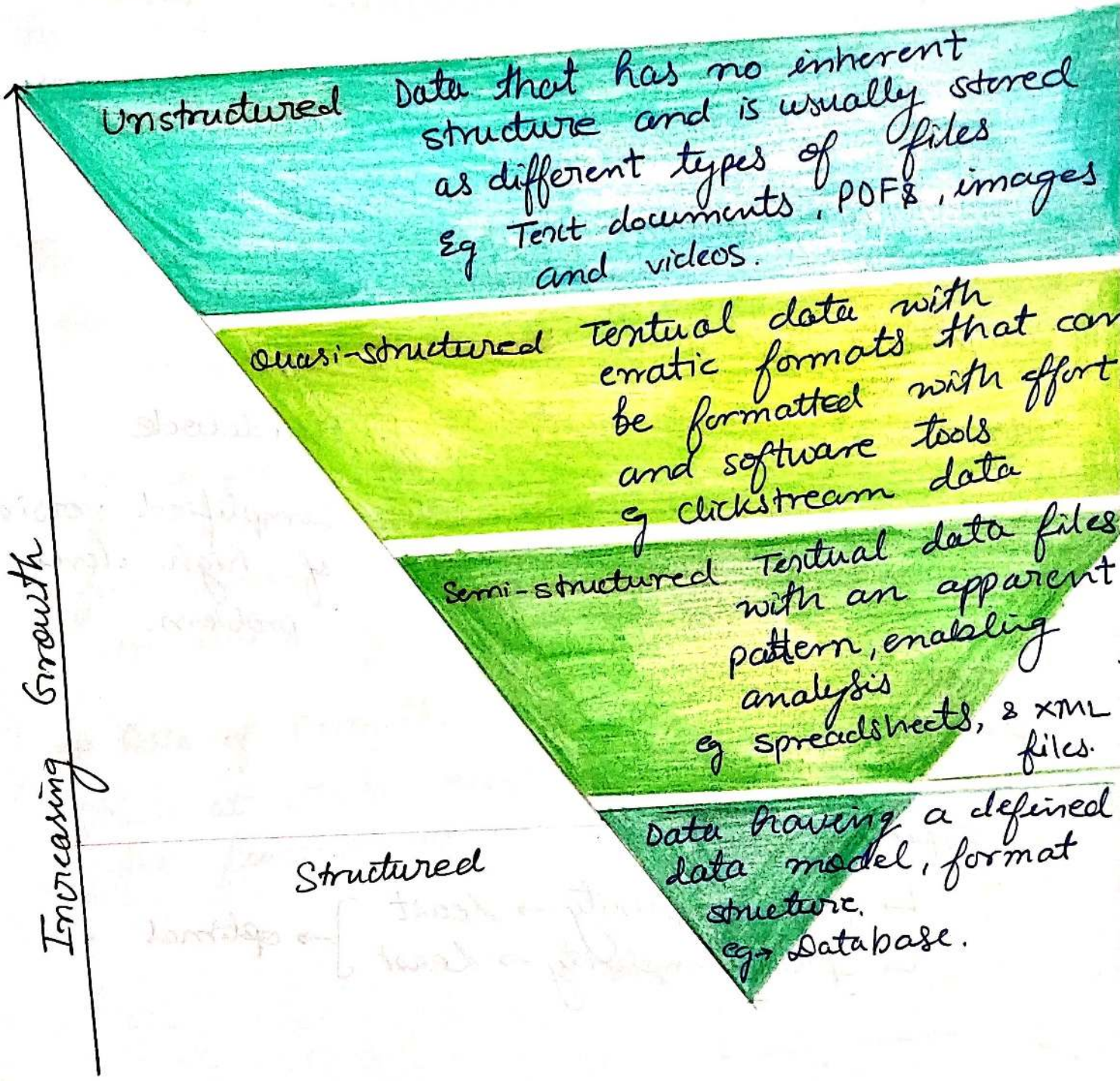
Data Structures

Data → Raw facts and figures

Information → Processed data

Knowledge → Analysis performed on processed data.



Data structure → It is the organisation of data in a proper structured manner in order to manage it.

Insertion
Selection
Searching
Sorting
Deletion
Modification/Updation
Traverse

} operation of Data.

Complexity of Algorithm

Algorithm

step by step procedure to solve a particular problem.

Flowchart

Pictorial representation of steps.

Pseudocode

simplified version of high level problem.

Efficiency ?

↳ Time complexity → Least
↳ Space complexity → Least

] → optimal

If N is an algorithm and suppose n is the size of input data then time and space used by N are two main measures of efficiency of N.

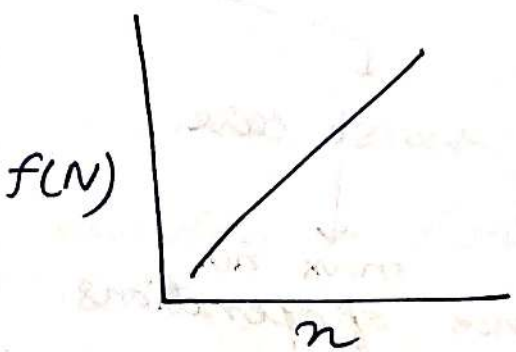
The time is measured by counting the no. of key operations in sorting and searching algorithms.

The space is measured by counting the ~~no. of key operations in sorting~~ maximum of memory needed by algorithm.

The complexity of a algorithm N is the function f(N) which gives the running time and/ for storage space required of algorithm in terms of size n. of the input data.

Rate of Growth

Rate at which running time increases as the function of input



Ex- Time complexity

```
int fib(n)
{
  int A[n+1]; ---- 1
    A[0]=0 ;  ---- 1
    A[1]=0;   ---- 1

  for (int i=0 , i<n < i++)
    {
      A[i] = A[i-1] + A[i-2] → n
    }
  return A[n] - 1
}
```

space complexity → n

Time complexity → 3n+3
(3n+c)

$T_c = n$
↳ O(n)
↓
avoid constant.

There are three cases that are employed in determining a complexity of an algorithm

- worst case
  - Take in accordance for sorting algorithm.
  - max no. of operations
- Avg case in b/w
- Best case
  - descending best
  - ascending best
  - random best

There are 3 main asymptotic notations

① Big oh (O) → considered for Tc since it is the worst case scenario.

② omega (Ω) → Best case

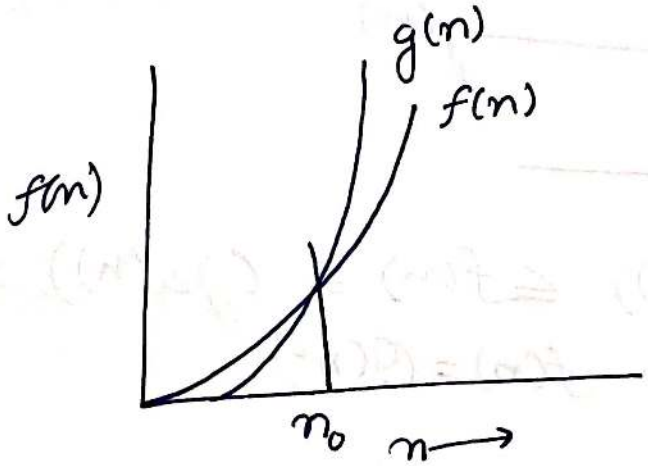
③ Theta (Θ) → Avg. case

The above mentioned are tightly bounded notations

$f(n) = n^2 + 3n + 1$ For Big oh ('O')

$g(n) = 2n^2$

$n=1$	$n=2$	$n=3$	$n=4$
$f(1) = 5$	$f(2) = 11$	$f(3) = 19$	$f(4) = 29$
$g(1) = 2$	$g(2) = 8$	$g(3) = 18$	$g(4) = 32$



$0 \leq f(n) \leq c.g(n)$

for $n \geq n_0$

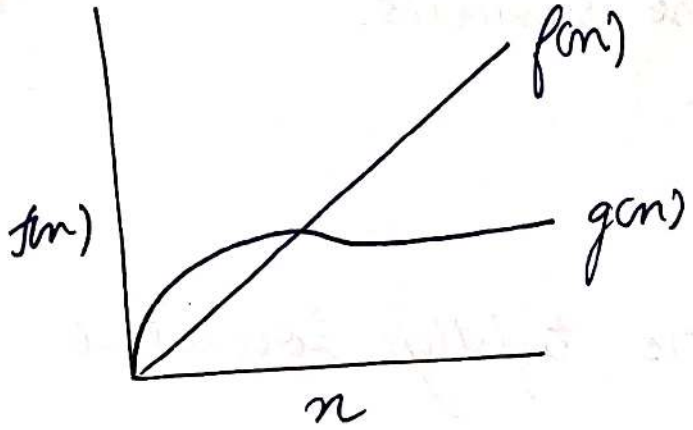
This determines $g(n)$ is upper bounded to $f(n)$ and can be written as

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

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

2) Ω - Bounded

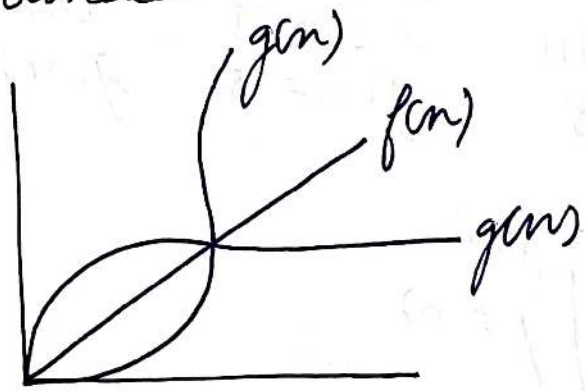
This states that g(n) lower bounds the f(n) function.



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

$f(n) = \Omega(g(n)) = \Omega(n^2)$

3) Θ - Bounded



$c(g_1(n)) \leq f(n) \leq (g_2(n))$

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

Q for $i \in 2$ to $n-1$ $\rightarrow n-2$

A1: if i divides n
n is not prime

A2: for $i \in 2$ to $\sqrt{n}$ $\rightarrow \sqrt{n}-1$
if i divides n
n is not prime

	A_1		A_2	
$n=1$	$(n-2)$		$\sqrt{n}-1$	
$n=10$	8	$<$	2	As more efficient ↓ Always for large no (efficiency)
$n=100$	98	$<$	9	

Q) Tc

1) if { } → O(1)
else { }

2) for(n)
{ } → O(n)

3) for(n)
{
for(n) → $O(n^2)$
}

4) for(n) → $O(n^2 \log n)$
for(n)
for(n/2)
↳ log n

5) for(n/2)
{ } → O(log n)

Asymptotic Notations (Loosely Bound)

i) little 'o'

$0 < f(n) < c\,(g(n))$

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

upper loosely bounded

ii) little 'w'

$cg(n) \leq f(n)$

lower loosely bounded

sorting Algorithms :-

classification on the basis of data

↳ Internal Sorting

↳ External sorting

Internal sorting → the data that needs to be sorted is present in main memory.

External sorting → The data that needs may not be present in main memory.

"sorting can't occur in secondary memory" (Rom)

1 stable sorting

10 (A) 5 6 10 (B)

i) Stable sorting 5 6 10 (A) 10 (B)

ii) Unstable sorting 5 6 10 (B) 10 (A)

Definition - ⇒ One of the sorting which keeps record in the same key value in same relative order that they were before sorting.

ii) It maintains the relative order of the records whose key values are same before and after sorting.

Inplace Sorting →
Sorting algorithm whose extra space requirement is constant i.e inputs are over-written by output as algorithm proceeds

Bubble Sort → stable sort -①

SC → O(1) → inplace

n = 30, 52, 29, 87, 63, 27, 18, 54

Passes = n-1 → Always

P_1 = 30, 52, 29, 63, 27, 18, 54 | 87

P_2 = 30, 29
= 29, 30, 52, 63, 27, 18, 54 | 87
= 29, 30, 52, 27, 63, 18, 54 | 87
= 29 30 52 27 18 63 54 | 87
= 29 30 52 27 18 54 63 | 87
= 29 30 52 27 18 54 | 63 87

P_3 = 29 30 52 27
= 29 30 27 52

P_4 = 29 30 52
= 27 29 30 52 18 | 54 63 87

P_5 = 18 27 29 30 52 54 63 87

Even though it was sorted in P_6, P_7 should run.

Algorithm for Bubble sort → Ascending order.

optimised version

```
{
int i, j, temp;                          → 1
for (i=0, i<n-1, i++)                    → 1, n, n
{
for (j=0; j<n-1-i, j++)                  → 1, n, n
{
if (a[j] > a[j+1])                       → n
{
temp = a[j];                             → n
a[j] = a[j+1];                           → n
a[j+1] = temp;                           → n
}
}
```

Inplace sorting

stable sorting

$(3+2n)(n+1)$ → time complexity exact.

$O(n^2)$ → Avg, best, worst

$O(1)$ → space complexity

Selection Sort ⓪

n=	0	1	2	3	4	5	6	7
	30,	52,	29,	87,	63,	27,	18,	54

Passes = n-1

P_1 = 18 52 29 87 63 27 30 54

P_2 = 18 27 29 | 87 63 52 30 54

P_3 = 18 27 29 30 | 63 52 87 54

P_4 = 18 27 29 30 52 | 63 87 54

P_5 = 18 27 29 30 52 54 | 63 87

P_6 = 18 27 29 30 52 54 87 63

P_7 = 18 27 29 30 52 54 87 63

unstable

space complexity = O(1)

Algorithm

```
void selection_sort (int a[], int n)
{
  int i, min, loc, j, temp;
  for (i=0, i<n, i++)
  {
    min [a[i]; loc=i)
    for (j= i+1; j<n ; j++)
    {
      if (a[J] < min)
      {
        min = a [J];
        loc = j;
      }
    }
  if (loc != i)
  {
    temp = a[loc];
    a[loc] = a[i];
    a[i] = temp;
  }
}
}
```

Insertion sorting (Card Placing) - (8)

n = 30, 52, 29, 87, 63, 27, 18, 54

By Default; Ascending Sort

Passes: n-1 | n considering 1 element
↓
not considering

If considered then (n)
else (n-1)

~~Passes: n~~ P_1 : 30 # Two sorts are going ~~to~~ i.e internal sorting occurs, 2 loops are working,

P_2 : 30, 52
30 < 52

P_3 : 29 30 52

P_4 : 29 30 52 87

P_5 : 29 30 52 87 63 → swap
87 ⩽ 63

P_6 : 29 30 52 63 87

P_6 : 29 30 52 63 87 27 # If all elements are sorted, then for T_c to be minimum INSERTION SORT is opted
↳ if not then

Counting sort

$O(n^2) = T_c$ } avg & worst case

$O(n)$ → best case (sorted elements)

Algorithm

```
void insertion_sort (int a[], int n)
    for (i=1, i<n, i++)
    {
        temp = a[i];
        J = i-1;
    while (a[J] > temp && J >= 0)
    {
        a[J+1] = a[J]; // element moving to right
        J = J-1;
        a[J+1] = temp; // Keeping new elements at its correct position.
    }
    }
```

Space complexity → o(1)
→ Inplace and Stable.

Divide and Conquer approach (DAC)

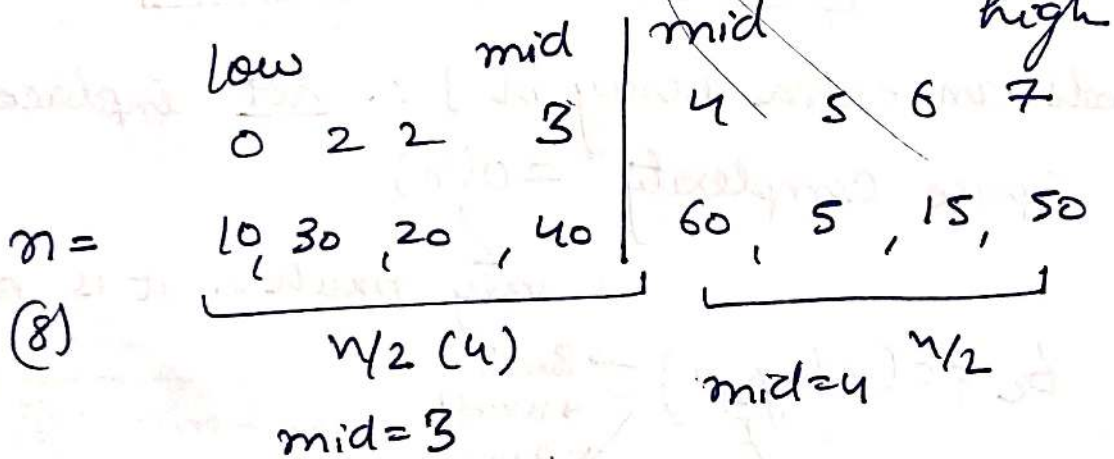
General Algorithm

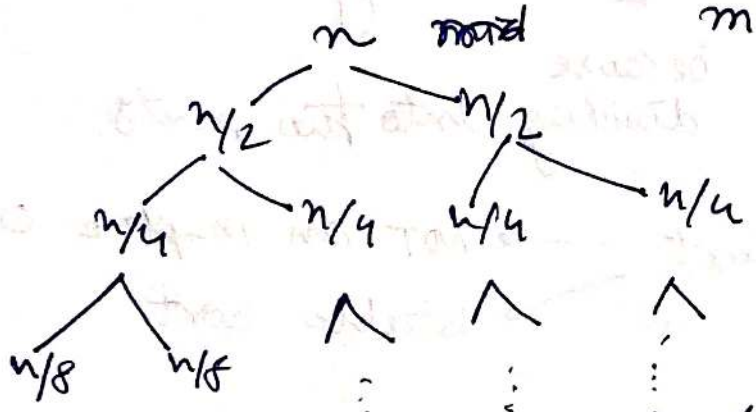
```
DAC(P)
{
  if (P is small & whose solution is known)
      return Solution(P)

  else
      decompose P into (P1, P2 ... Pn

}
DAC(P1) ∪ DAC(P2) ∪ ..... ∪ DAC(Pn)
```

Merge Sort is based on DAC approach. we are dividing array into 2 equal arrays untill we get the solution.



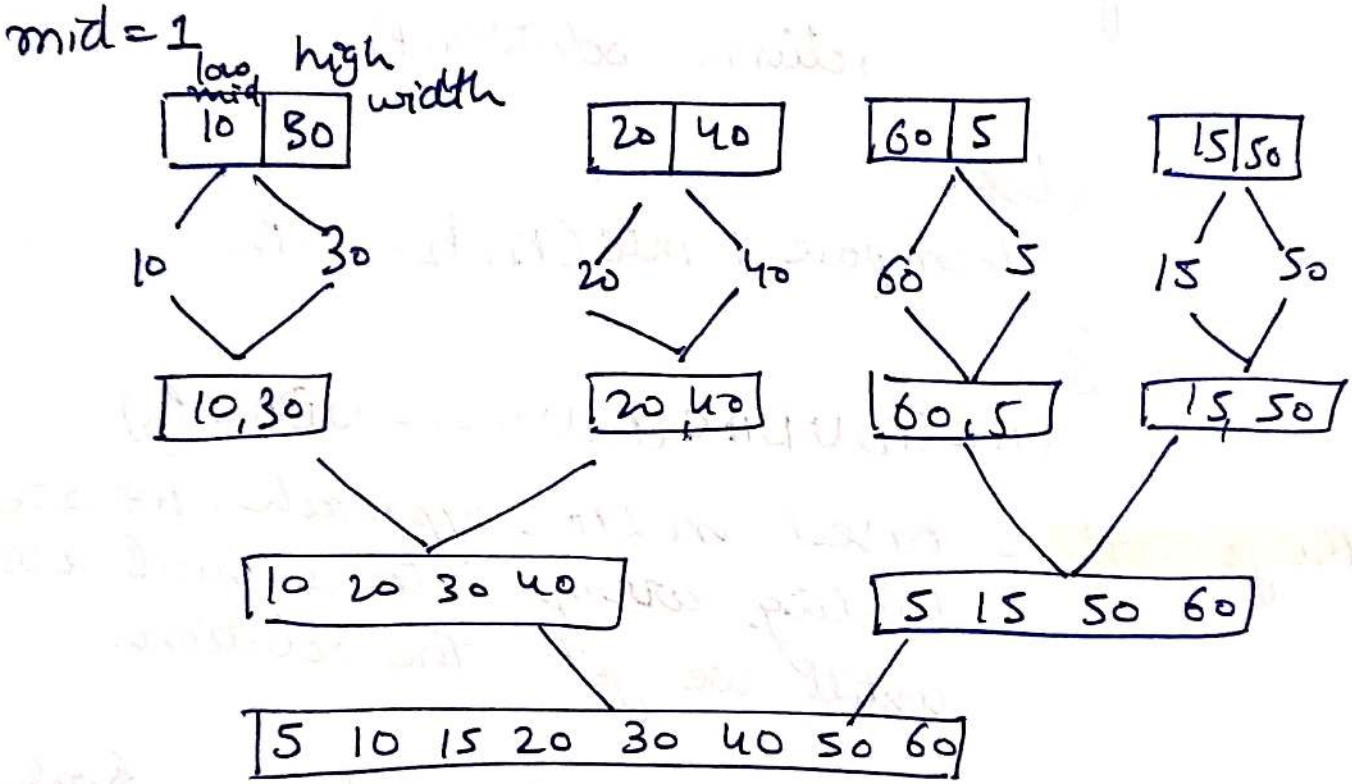


$mid = \frac{low + high}{2}$

Q) How to dec stable from n→ count

merge sort

low	mid	mid	high
10	30	20	40



Needs an extra array b[ ] ∴ not inplace sorting

Space complexity = O(n)

very much ∴ it is never used

$t_c = O(n \log_2 n)$ → Best, worst, Avg

because dividing into two parts.

Merge Sort → NOT an in-place O(N)

→ stable sort

```
void merge_sort (a, low, high)
                  ↓     ↓     ↓
                array  lower  higher
                       index  index
T(n)
for {n element}
        if (low ≠ high)
        {
            mid = (low + high) / 2 ;
```

front 1/2 $T(n/2)$ = merge_sort (a, low, mid);

Back 1/2 & $T(n/2)$ · merge_sort (a, mid +1, high);

$O(n)$ ← merge (a, low, mid, high);

}

}

merge is a function to produce sorted array from already two sorted array in linear time i.e $O(n)$.

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

$$= O(n \log_2 n)$$

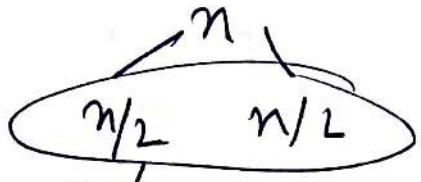
For random elements → Quick sort

Quick Sort :- Preferable for random elements

(4) ↳ Inplace and not stable

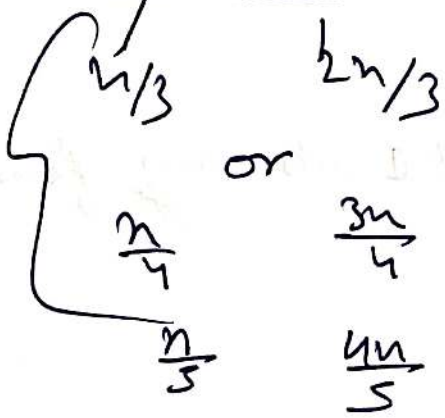
Quick sort is based on DAC approach. Here we are dividing n elements into 2 unequal sizes.

Best



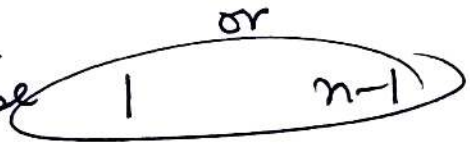
$$T(n) = 2T\left(\frac{n}{2}\right) + O(n)$$
$$= O(n \log_2 n)$$

Avg case



$$= T\left(\frac{n}{3}\right) + T\left(\frac{2n}{3}\right) + O(n)$$
$$= T\left(\frac{n}{4}\right) + T\left(\frac{3n}{4}\right) + O(n)$$

worst case



$$\Rightarrow T(n) = T(1) + T(n-1) + O(n)$$
$$= O(n^2)$$

⇒ It is based on a pivot element. which can be chosen from a specific position within the array.

Elements of an array can be partioned so as that pivot is placed into position "J" and following 2 condition hold.

1) Each element b/w $0 \to J-1 \leq$ pivot
2) Each element b/w $J+1 \to n-1 \geq$ pivot

Sorted elements → never prefer Quick sort as it will be worst case

$T(n) = O(n^2)$. [user insertion/ counting]

If pivot = median

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

$= n \log n$

#optimised version ⇒ Randomised Quick Sort

$T(n) = n \log n$ (every case)

⇒ The object of partition is to allow the pivot element to find its proper position w.r.t others in the array

Pivot = I element (default)

= last / median also

```
                    ┌ lower index
void quicksort (a, p, r) → higher index.
                 └ array
    if (p ≥ r) return;
O(n) ⇐ j = partition (a, p, r);
T(n/3) | T(n/2) ⇐ quicksort (a, p, j-1);
T(2n/3)| T(n/2) ⇐ quicksort (a, j+1, r);
  ↓    |   ↓
  n    |  n log n
```

```
int i, j, R, pivot;
pivot = a[p]; // 1st element as "pivot"
i = p; j = r;
```

Bucket Sort / Bin Sort → Better representation of Radix sort

⑥ ↳ not inplace

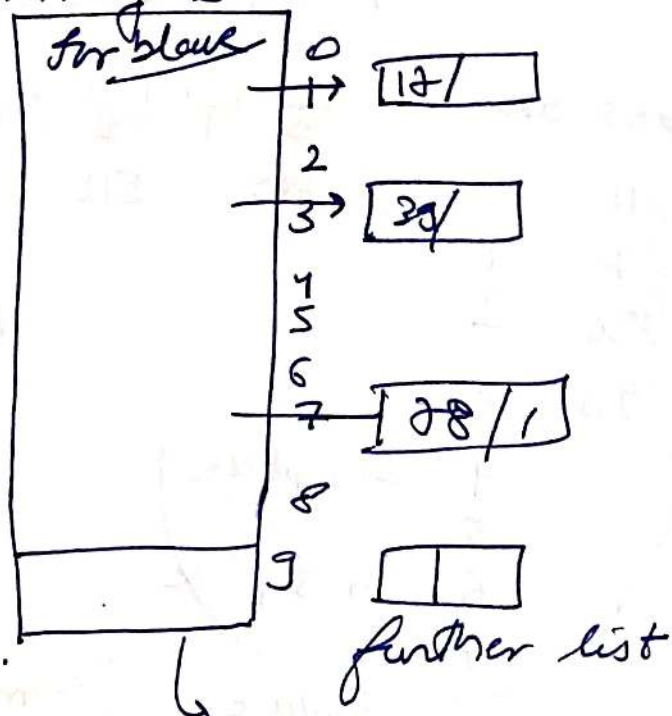
78, 17, 39, 26, 72, 94, 21, 12, 23, 68

Array A

78
17
39

$\lfloor n \cdot A[i] \rfloor$

Array B



S.C = O(n) → Not inplace.

Stable sort

Use insertion sort only (internal)

$O(n^2)$ → worst

$O(n)$ → Avg / Best

Algorithm

B.S(A)

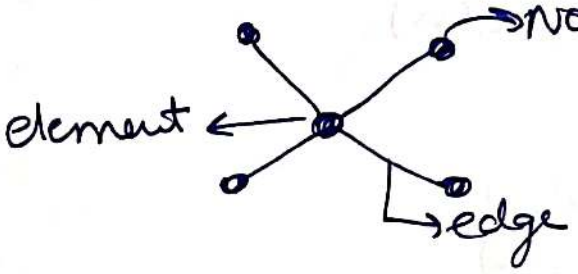
1) let B[0 ..... n−1] be a new array
2) n = A.length
3) for i = 0 to n−1
4) make B[i] empty list
5) for i = 1 to n
6) insert A[i] into B[⌊n.A[i]⌋] (flooring)
7) for i = 0 to n−1
8) Sort list B[i] with insertion sort

9) concatenate lists B[0], B[1], ...

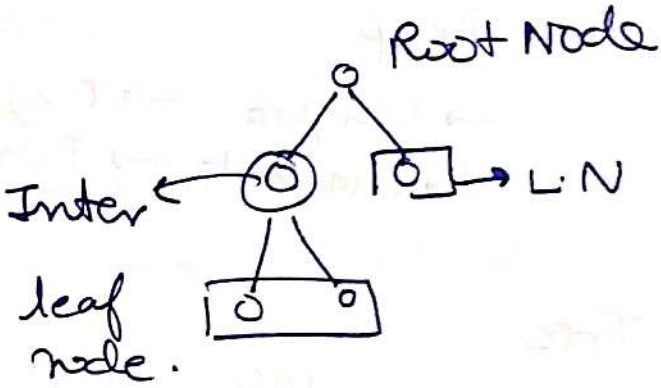
⑦ HEAP SORT → only for tree

Heap sort is based on almost complete binary tree

Tree

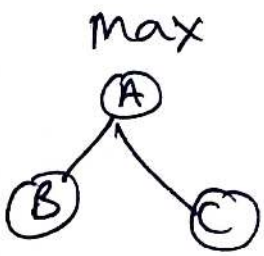


n nodes → n-1 edges

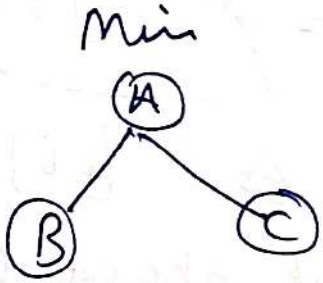


There are two types of heap

→ max Heap
→ min Heap



A > B
A > C



A < B
A < C

↓

Value at each node is bigger than the value of both child nodes

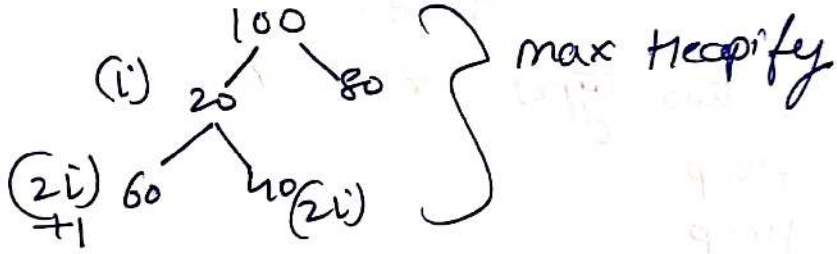
Root will provide you largest values going to leaf node from R → N ⇒ decreasing order

visiting node from root.

max heap
↳ Largest value → T(n) = O(1)
↳ smallest value → T(n) = O(n)

min heap
↳ Largest → T(n) = O(n)
↳ Smallest → T(n) = O(1)

Tree



Mem = 4b [i] < [2i]

Go [2i] < [i+1]

↳ if changed swap.

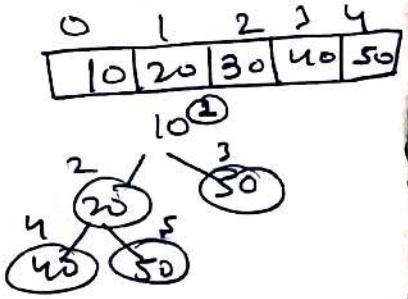
```
void max_heapify (a, i)
{
    l = 2i ; r = (2i+1);
    max = l;
    if (a[i] < a[l] and l ≤ n)
            ∴ max = l;
    if (a[max] < a[r] and r ≤ n)
            max = r
        if (max != i)
            a[i] ↔ a[max];
            max_heapify (a, max);
    }
}
```

$T(n) = \log_2 n \Leftrightarrow \log_2 n$

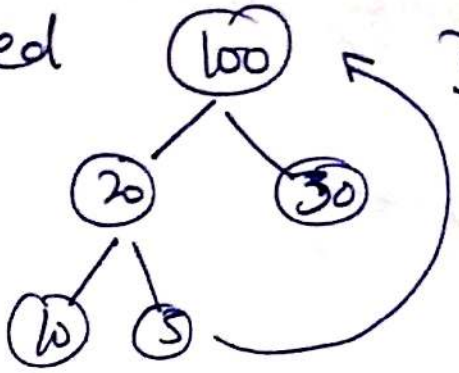
- $\log_2$ → working on 1/2 side
- 2 me → divide
- n → depends on height of tree

```
Build_max_heap (a, n)
{
    for (i = n/2; i ≤ 1; i--)
        max heapify (a, i);
}
```

$O(\log_2 n)$



deleted



} Always from top node. extraction / deletion.

```
int extract_max_heap (a)
{
    max = a[i];
    a[i] = a[n]
    n--; //elt deleted

    heapify (a, i)
    return max;
}.
```

```
void insert_into_heap (a, item)
{
    n++;
    i=n;
    while (item > a[i/2] and i>1)
    { a[i] = a[i/2];
      i = i/2;
    }
    a[i] = item;
}
```

```
void heap_sort (a,n)
{
  Build_max_heap (a,n);
  for (i=n; i>2; i--)
  {
    a[1] ← a[i]
    n--;
    heapify (a,i);
  }
}
```

Recurrence Relation (RR)

A Recurrence is an equation that describes a function in terms of its value an smaller inputs

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

There are 3 methods to solve RR

① Back substitution Method / Iterative / Inductive method

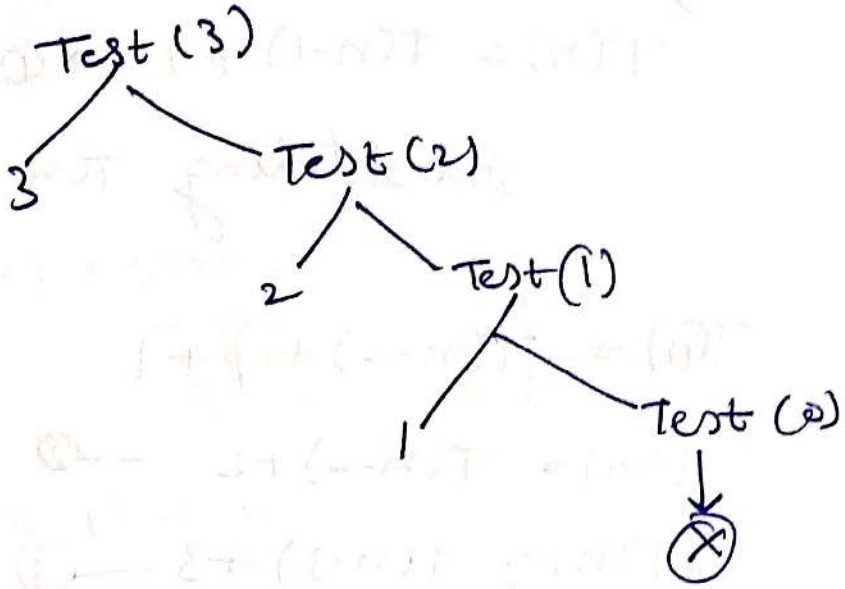
② Recursive Tree

③ Master's Theorem.

```
* void test (int n)  ──→ T(n)
  {
    if (n>0)  ──→ ①
    {
      printf ("%d", n);  ──→ ①
      Test (n-1);  ──→ T(n-1)
    }
  }
```

② Recursive Tree or Tracing Tree

$n=3$



By default $T(n) = 1$

printing and calling

Total $= 3+1$

↳ extra call

$T(3) = 3+1$

$T(n) = n+1$

$\therefore t_c \Rightarrow O(n)$

Recurrence Relation

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

$T(n) = T(n-1) + 1$ } ↳ for simplicity (roundoff to 1)

Ans. $T(n) = \begin{cases} 1 & , n=0 \\ T(n-1)+1 & , n>0 \end{cases}$

Using Substitution method:

$T(n) = T(n-1) + 1 \rightarrow ①$

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

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

Substituting $T(n-1)$

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

$T(n) = T(n-2) + 2 \rightarrow ②$

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

⋮

k steps

$$\boxed{T(n) = T(n-k) + k} \rightarrow ④$$

Assuming $T(n-k) = T(0)$

$n - k = 0$

$n = k$

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

$T(n) = T(0) + n$

$= 1 + n = O(n)$

*
```
void Test (int n)
{
  if (n>0) → ①
  {
    for (i=0; i<n; i++)
      → ② → ③ → ④
    {
      printf ("%d", n); → ⑤
    }
  }
```

```
    }
    Test (n-1);
  }
}
```

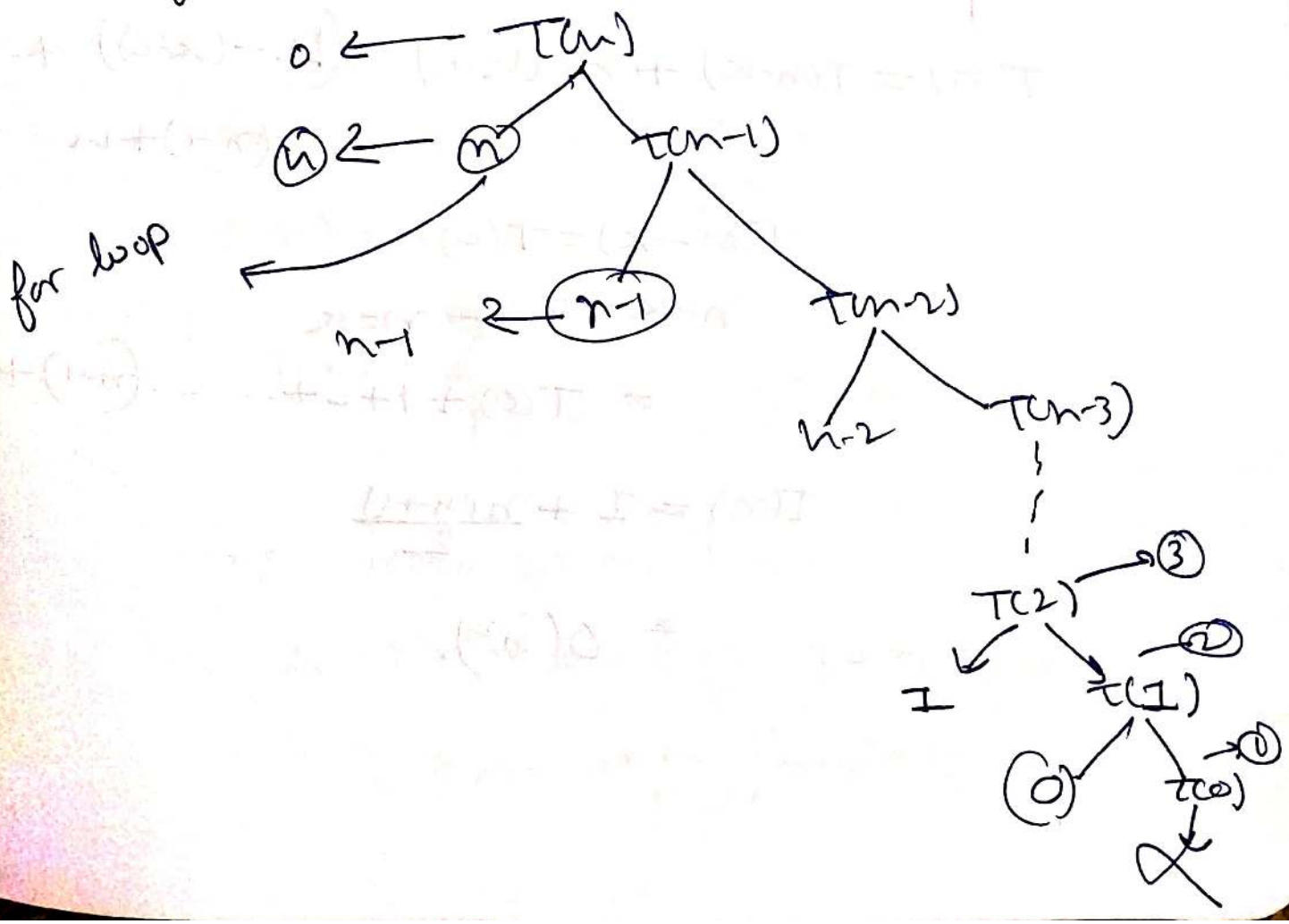
$T(n) = T(n-1) + 3n+2$

$3n+2$ ⇓ Round off to n

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

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

Using RT



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

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

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

Using Substitution

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

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

$$T(n) = T(n-2)+(n-1)+n \quad -①$$

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

$$\vdots$$

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

$$(n-1)+n$$

$$T(n-k) = T(0)$$

$$n-k=0 \Rightarrow n=k$$

$$= T(0)+1+2+\cdots+(n-1)+n$$

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

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

* void test (int n) → T(n)

```
{
  if(n>1) → (1)
  {
    for(i=0; i<n; i++)   (i → 1, n → n, i++ → n)
    {
      == ; (n)
    }
    Test (n/2); → T(n/2)
    Test (n/2); → T(n/2)
  }
}
```

$T(n) = 2T(n/2) + \underbrace{3n+2}_{\downarrow}$

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

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

$\because \quad T(n) = 2T(n/2) + n$

$\therefore \quad T(n) = 2\left[2T(n/4) + \frac{n}{2}\right] + n$

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

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

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

$$T(n) = 2^2 \left[2T\left(\frac{n}{2^3}\right) + \frac{n}{2^2} \right] + 2n$$

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

$\vdots$

k steps

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

$$T(1) = T\left(\frac{n}{2^k}\right)$$

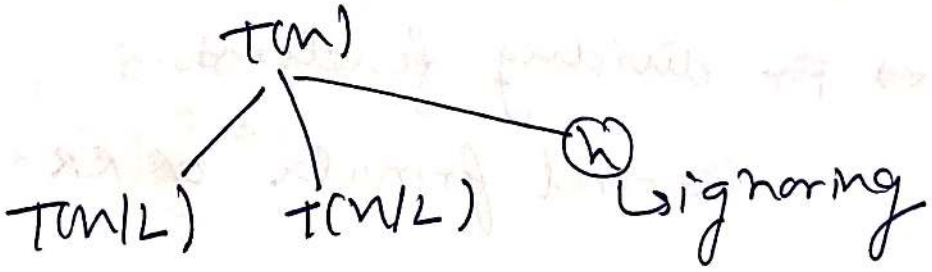
$$\boxed{n = 2^k}$$

$$T(n) = 2^k \cdot T(1) + n\log_2 n$$

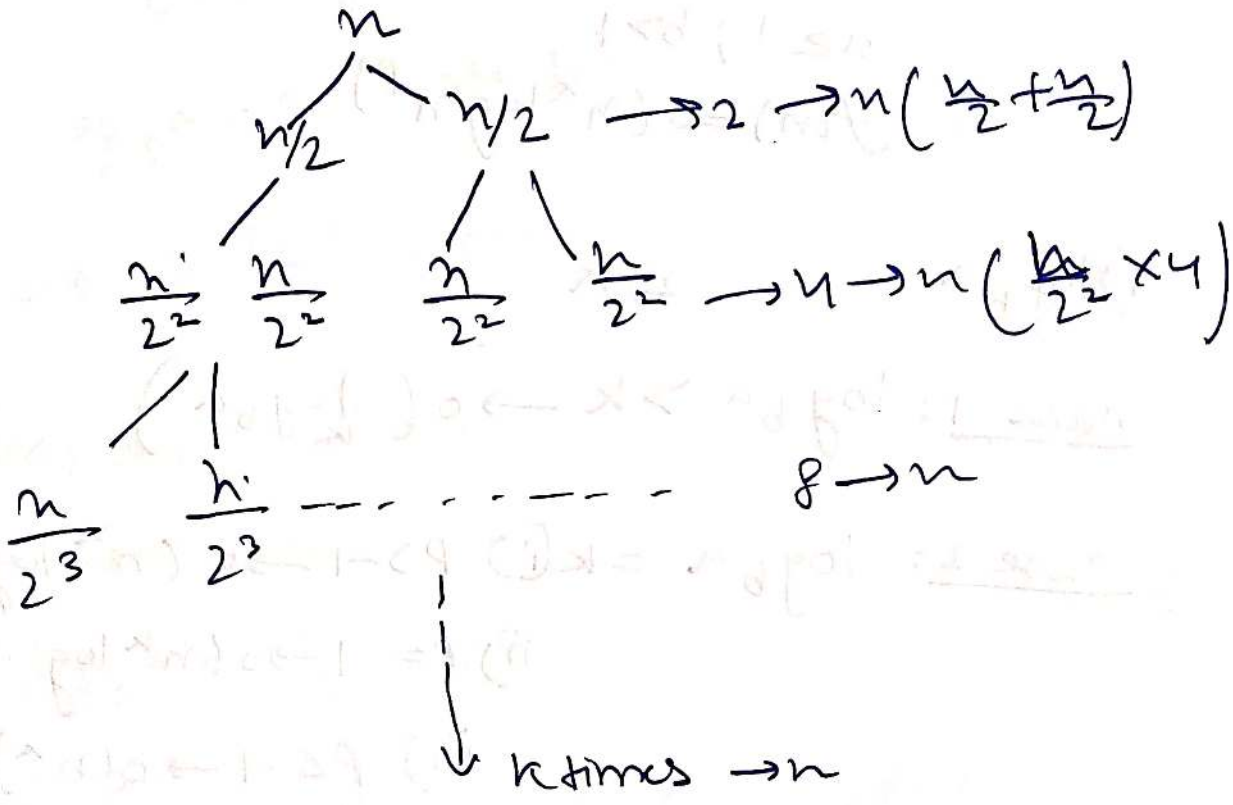
$$T(n) = n + n\log n$$

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

* using (RT)



or



$T\left(\frac{n}{2^k}\right) = T(1)$

$n = 2^k$

$\log n = k$

(3) Master's Theorem

⇒ For dividing functions

General formula of RR

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

$a \geq 1$; $b > 1$

$f(n) = o(n^k \log n^p)$

1) $\log_b a$ 2) k

case 1: $\log_b a > k \rightarrow o(n^{\log_b a})$

case 2: $\log_b a = k$ (i) $p > -1 \rightarrow o(n^k \log n^{p+1})$

ii) $p = -1 \rightarrow o(n^k \log \cdot \log n)$

iii) $p < -1 \rightarrow o(n^k)$

case 3: $\log_b a < k$ i) $p \geq 0 \rightarrow o(n^k \log n)$

ii) $p < 0 \rightarrow o(n^k)$

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

$a = 2, \quad b = 2$

$\log_a b = \log_2 2 = 1$

$f(n) = O(1)$

$= O(n^0 \log n) \quad k = 0, p = 0$

$\log_b a > k$ [case 1]

$\Rightarrow O_n(1) = O(n)$

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

$a = 4, b = 2$

$\log_b a = \log_2 4 = 2.$

$f(n) = O(n)$

$= O(n^1 \log n), k = 1, p = 0$

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

Q $T(n) = 8T\left(\frac{n}{2}\right) + n \rightarrow O(n^3)$

Q $T(n) = 8T\left(\frac{n}{2}\right) + n^2 \rightarrow O(n^3)$

Q $T(n) = 9T\left(\frac{n}{3}\right) + 1 \rightarrow O(n^2)$

Q _____ $+n \longrightarrow O(n^2)$

Q _____ $+n^2$

Decreasing Functions

1) $T(n) = T(n-1) + 3 \longrightarrow O(n)$

2) $T(n) = T(n-1) + n \longrightarrow O(n^2)$

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

4) $T(n) = 2T(n-1) + 1 \longrightarrow O(2^n)$

5) $T(n) = 3T(n-1) + 1 \rightarrow O(3^n)$

6) $T(n) = 3T(n-1) + n \Rightarrow O(n \cdot 3^n)$

General formula:

$T(n) = a \quad T(n-b) + f(n)$

$a \geq 0$

$b > 0$

$f(n) = O(n^k) \; ; \; k \geq 0$

case 1: $a = 1 \longrightarrow O(n^{k+1})$

$a > 1 \rightarrow O(n^k a^{n/b})$ $\quad [O(n^k a^{n/b})]$

$a < 1 \rightarrow O(n^k)$

Q $T(n) = 4T(n-3) + n \longrightarrow O(n \cdot 4^{n/3})$

Q $T(n) = 8T(n-1) + n^2 \log n \longrightarrow O(n^2 8^n)$

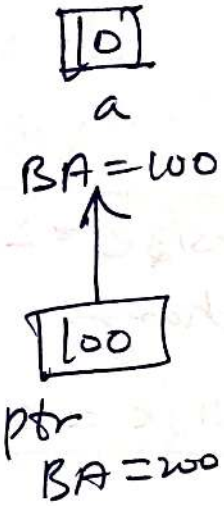
Q $T(n) = 10T(n-6) + n^3 \longrightarrow O(n^3 10^{n/6})$

DATA STRUCTURES

linear
- Array
- Linked list
- Stacks
- Queue

Non-linear
- Graph
- Tree

* Pointers :




```
int a;
int *ptr = &a
```
or
```
int a;
int *ptr;
ptr = &a;
```

\# Array :
It is the collection of homogenous elements stored in a continuous memory location

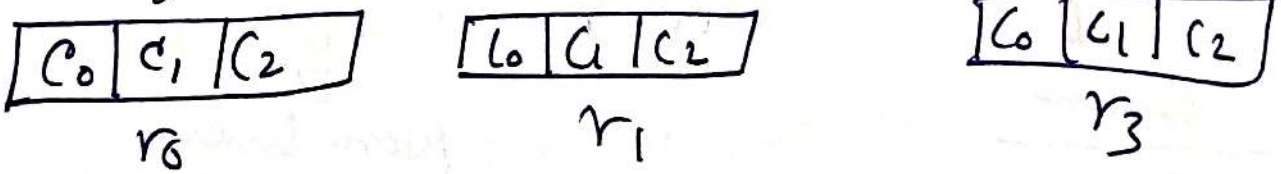
→ Continous memory location, consumes 2 bytes
→ pre-defined

a[i] /*(a+i)/ *(i+a)

* 1D
int a[10];

* 2D int a[3][5];
memory = 3 × 15 = 15
21
a[15]

Row Major



* Memory $a[m]_{row}[n]_{column}$.

Row Major

$\rightarrow BA + (i \times n + j) w$

column Major

$\Rightarrow BA + (j \times m + i) w$

BA ⇒ 100

	C_0	C_1	C_2	C_3
r_0	1	2	3	4
r_1	5	6	7	8
r_2	9	10	11	12

{ Pointer size = 2 bytes never changes
Int size = 2 bytes

a[3][4];
m n

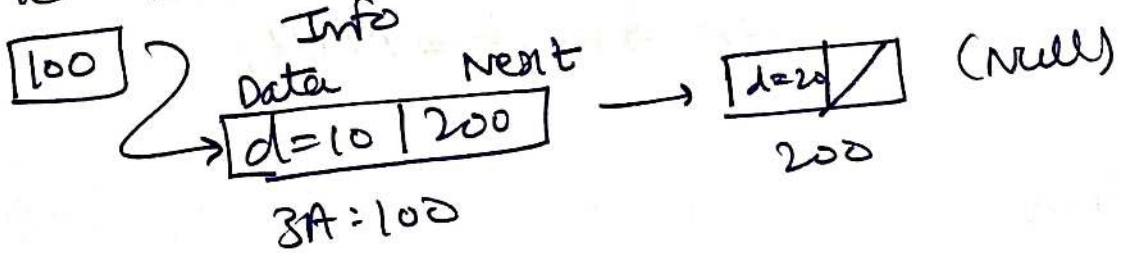
i=2 ; j=2

w=2 (int) {w - words per cell}

Linked list → singly
→ Doubly
→ Circularly doubly

* Singly linked list

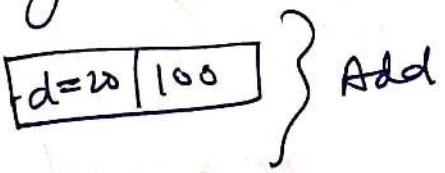
Head / start

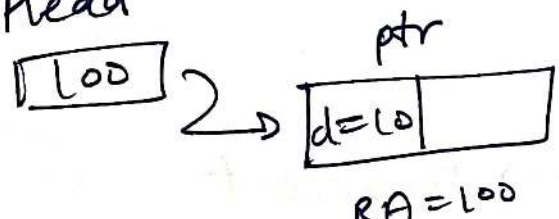


Levels

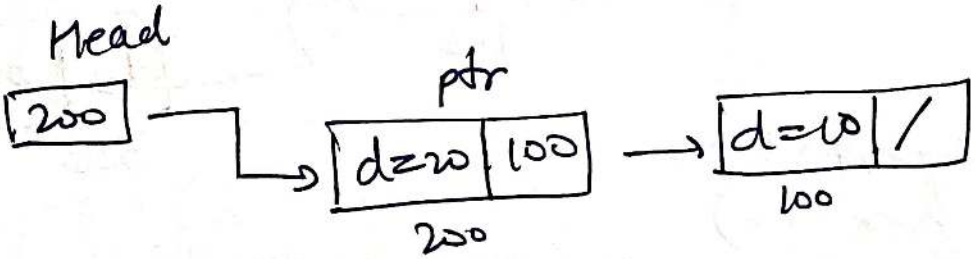
1) Beginning

Head = Null ; // empty d l





Head = pt:
ptr → next = Null;



ptr → next = head;
head = ptr;

2) End

Empty ll → head = ptr ; ptr → next = Null;

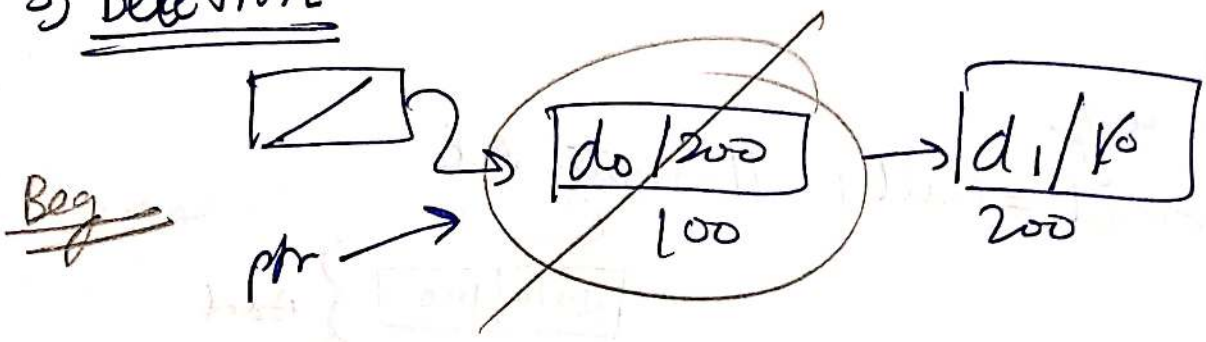
while - loop (loc ! = Null)

{

loc = loc → next = ptr;

ptr → next = Null;

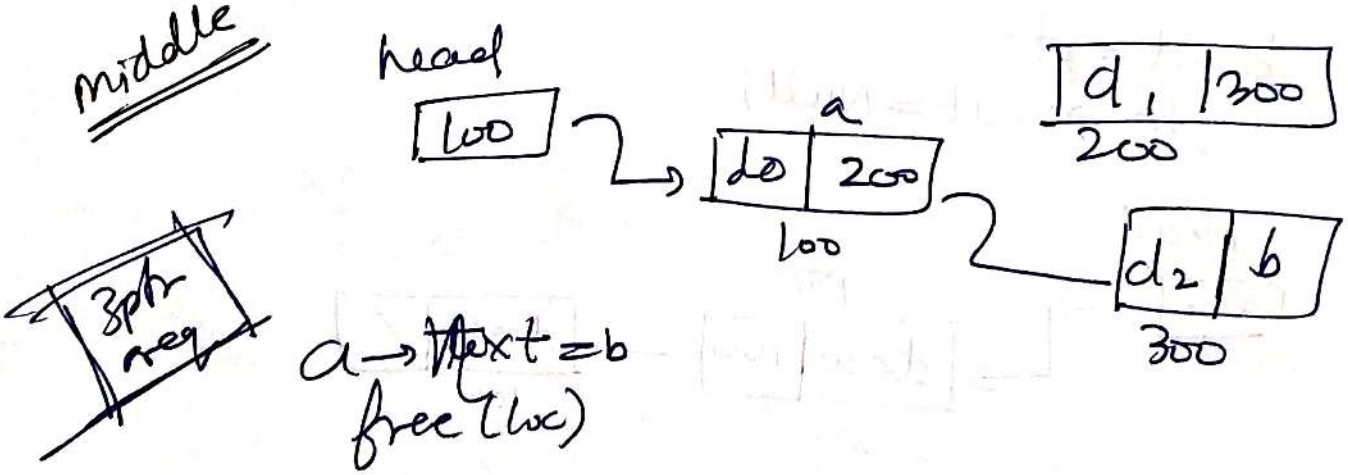
3) Deletion

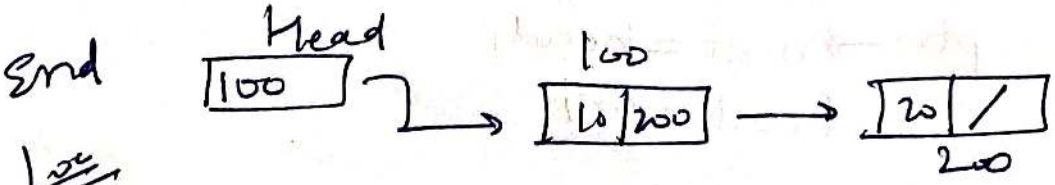


ptr = head

next = Null

free (ptr); → Important

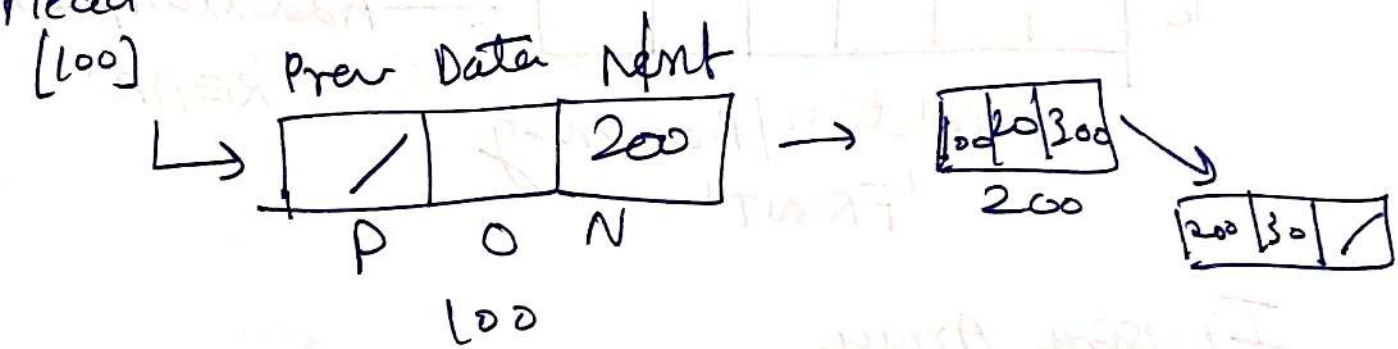




Doubly → O(1) → first
O(n) → last or particular } Insertion deletion

```
ptr = head;
while (ptr != NULL)
```

Doubly linked list (2-way traversing)
↳ [L→R; R→L]



Circular

O(1) — start [insertion & deletion]
O(1) — end
O(n) → particular

Null not allowed.

Q write a program to perform the following operations on an array

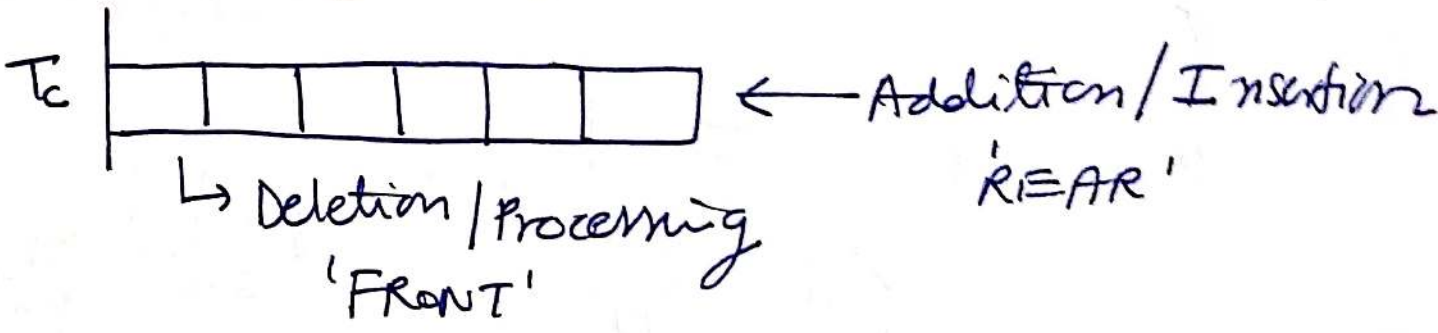
- Insertion
- Deletion
- Sorting
- Searching

Q write a ... on a singly linked list

- Insertion at beginning
- Insertion after particular loc.
- → end

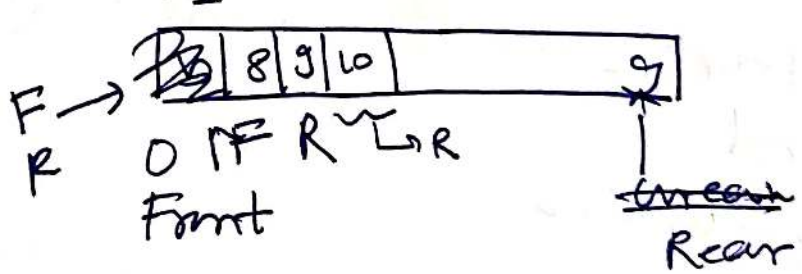
- Deletion (all 3)
- Searching
- Sorting
-

Queues



I) Using Array

```
int arr[10];
```



For empty

Front = Rear = -1
++ ++

empty → delete : underflow
complete → add : overflow

Enqueue (Insertion)

```
if (rear == -1)   // 0 is empty
{
  arr[++rear]=x;
     ++front
}
else
{
   if (rear == ±N) // overflow
{No spaces;}
else
  { arr[++rear]=x.
  }
```

* Dequeue (Deletion)

```
if (front == -1)    //underflow
    {
      No Deletion;}
      else if (front == rear)
        {
          front=rear=-1;}
        else
          {++ front;
        }
```

II Using Pointers

```
struct queue
{
int data;
struct queue * next;
}
struct queue * front * rear;
front = rear = Null;
void enqueue.
{
struct queue * ptr;
ptr = (struct queue *)
malloc (size of (struct queue))
if (front == Null)
{
front = rear = ptr;
ptr → next = Null;
}
else
{rear) → next = ptr;
rear = ptr;
}
```

```
void dequeue
{
  struct queue *ptr;
    if (front==NULL)
     {
       no element;
     }
  else if (front==rear)
     {
       ptr = front;
       front = rear = NULL;
        free (ptr);
     }
  else;
    { ptr=front;
      front = (front)->next;
        free (ptr);
    }
}
```

Applications

1) Ticket Counter
2) BFS (Breadth first search)
3) Priority Queue [Ques comes: short ques]
   Priority check → Delete

Very Common

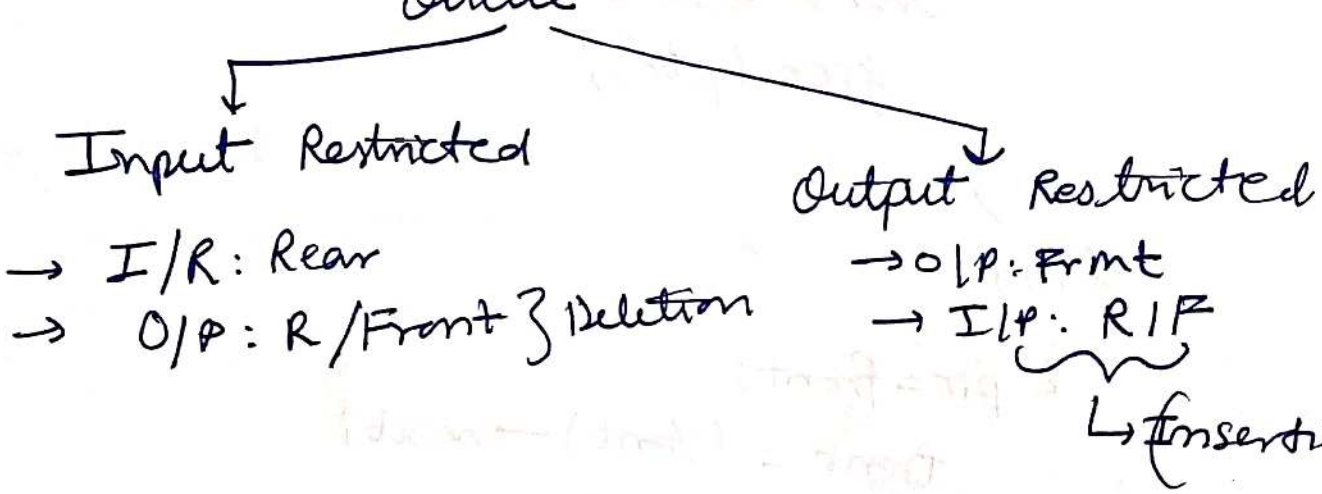
2) 2 Priorities → normal queue (first come)

3) Min Heap tree

↳ top vala process ho jayega

Linked list: | Prior | ID | Next |

Queue : FIFO / LILO



STACKS → [LIFO] / [FILO]

stack of plates → Addition / deletion from TOP

Add : Push
Delete : Pop

Add : 1, 2, 3, 4, 5 : Push

Delete : 5, 4, 3, 2, 1 : Pop

LIFO / FILO

OU) ordered list of similar elements.

OU) Push ( ) – Insertion at top
Pop ( ) – Deletion at TOP
Peek ( ) – Provide element at TOP – without deleting
Overflow ( ) – stack full (Is full)
Underflow ( ) – stack Empty (Is-Empty)

Using Array

```
int arr [10];
int top = -1, // Empty
void Push() // overflow stack
```

Empty = -1
elements
Full (N-1)

```
{ Pf("Enter value")
  sf("%d", and++top).
}
```

3

```
void pop()
{
pf("%d", arr[top])
--top;
}
```

② Using pointers

```
struct stack
    {
    int data;
      struct stack * next;
    }
  struct stack *top;
        top = Null;
void push()
    {
    struct stack *ptr;
pf(Enter data item)
sf( ____ ;&pf->data)
 if (top == Null)
    {
    top = ptr;
     ptr->Next = Null;
    }
  else
    {ptr->next = top
          top = ptr;
    }
```

```
void pop()
  { struct stack *temp;
        temp = ( )
       if (top == Null)
        {
          // stack Empty }

  else
      {
        temp = top;
           top = (top) -> next;
                 full (temp);
       }

  void peek ( )
    { if (top = Null)
          {// Empty)

    else
        { pf ( — ; & top -> data ); }

    }
```

Applications of stack

→ Expression conversion

→ Expression evaluate

Infix (a+b) op1 operant op2

Prefix (+ab) operant op1 op2

Postfix (ab+) op1 op2 operant

Infix to postfix

Infix to prefix
Postfix to Infix
prefix to Infix

Precedance:-

(), ($, ↑), /, *, +, -

Infix to Postfix

(A+B) * C-D

Symbol	scan	stack
(	(	
	((	
A	((	A
+	((+	
B	((+	AB
)	(	AB+
*	(*	AB+
C	(*C	AB+
-	C	AB+C*
D	C-	AB+C+D
)	Empty	AB+C*D-

(A+B) *C-D
L→R

2) A+B * C+D-E

Que → Linklist
S → Array

Symbol	Scan	Stack
	C	
A	C	
+	C+	A
B	C+	AB
*	C+*	AB
C	C+	ABC*+
+	C+	ABC*+D
D	C-	ABC*+D+
-	C-	ABC(*+D+E
E	Empty	ABC*+D+E-
)		

Infix to Prefix

(A+B*C+D-E) ⇒ (E-D+C*B+A)

Symbol	Scan	Stack
	C	
E	C	E
-	C-	E
D	C-	ED
+	C+-	ED
C	C+-	EDC
*	C+-*	EDC
B	C+-*	EDCB
+	C-+*	EDCB*+
A	C-+	EDCB*+A
)	Empty	

(EDCB*+A)

-- A+*BCDE

↳ check

A+B*C+D-E

A+*BC

30 September 2019 @ 1:11 pm

Non-Linear Data structure
↳ Graph
↳ Trees

Graph 'G'

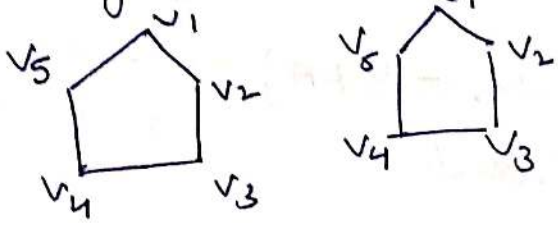
→ vertices, $V = \{ v_1 \ldots\ldots, v_n \}$

Edges, $E = \{ E_1, \ldots\ldots, E_n \}$

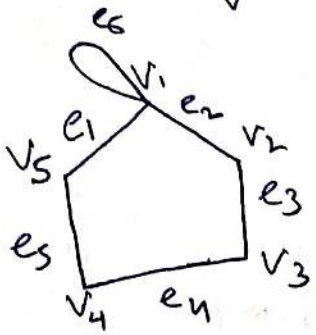
$G = (V, E)$

$E \subseteq V \times V$

cyclic or Acyclic



Degree of a vertex 'v' – The no. of edges that are directed to that particular vertex



$v_1 = 2(e_1, e_2)$

$v_2 = 2(e_2, e_3)$

$v_3 = 2(e_3, e_4)$

$v_4 = 2(e_4, e_5)$

$v_5 = 2(e_5, e_1)$

self loop ⇒ $v_1 = 4$ (e_1, e_2, e_6, e_6)

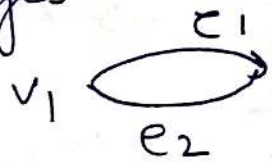
Degree = (4) v_1

source & final destination (e_6)

$v_3 = 6[e_3, e_4, e_7, e_8]$ (2) (2)

e_4	v_3	v_4
	1	1
e_7	v_3	v_3
e_8	1	1

Parallel Edges



e_1 $v_1 \longrightarrow v_2$

e_2 $v_1 \longrightarrow v_2$

$deg(v_1) = 2$

$deg(v_2) = 2$

Self loop

e_1 $v_1 \longrightarrow v_1$

$deg(v_1) = 2$

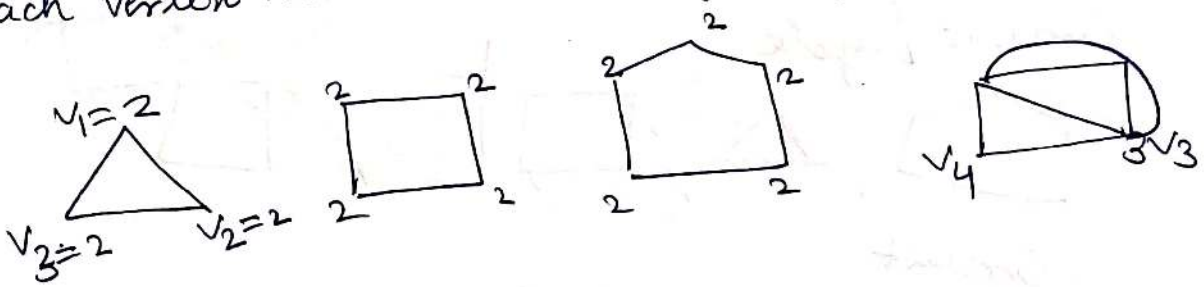
Simple Graph

→ No self loop

↳ No Parallel Edges

Regular graph

→ Each vertex has the same no. of neighbours (same degree)



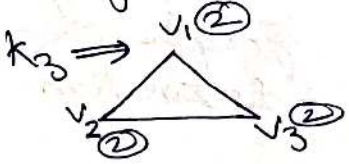
Complete Graph

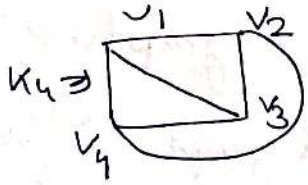
Each vertex (must be connected to each vertex)

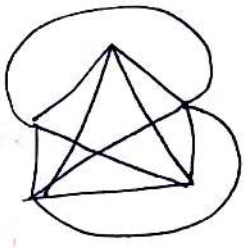
adjacent → edge in between

(All are neighbours)

→ Cyclic graph







K5 ⇒

RG ⇏ C.G

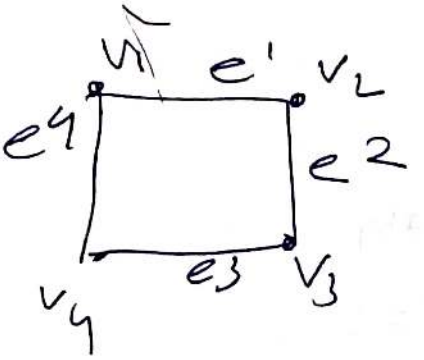
C.G ⟹ R.G

→ WALK → Path → Circuit / Cycle

↳ V repeat → VR ✗ $v_1 c_1 v_2$

↳ E repeat → ER ✗ $c_2 v_3 v_3$

$v_1 \to e_1 \to v_2 \to e_2 \to v_3 \to e_3 \to v_4 \to e_4 \to v_1 \to e_1 \to v_2$.



Walk

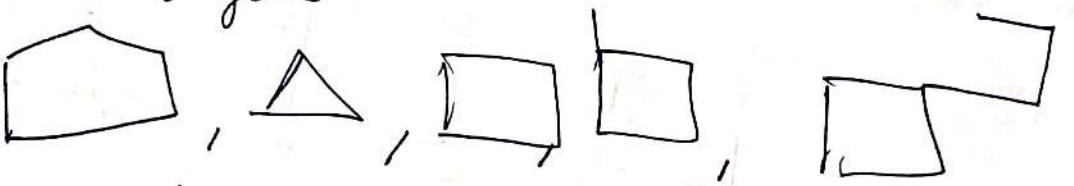
$v_1\ e_1\ v_2\ e_2\ v_3\ e_3\ v_4$

Closed walk

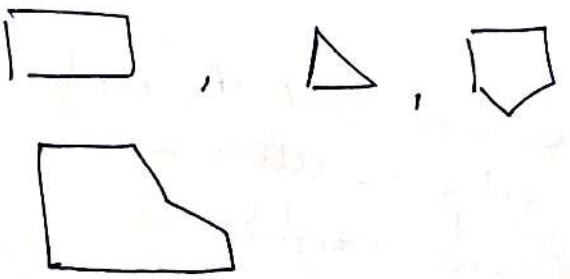
SV, DV ⇒ Same

$v_1 e_1 v_2\ e_2 v_3 e_3 v_4 e_4 v$

Circuit / Cycle



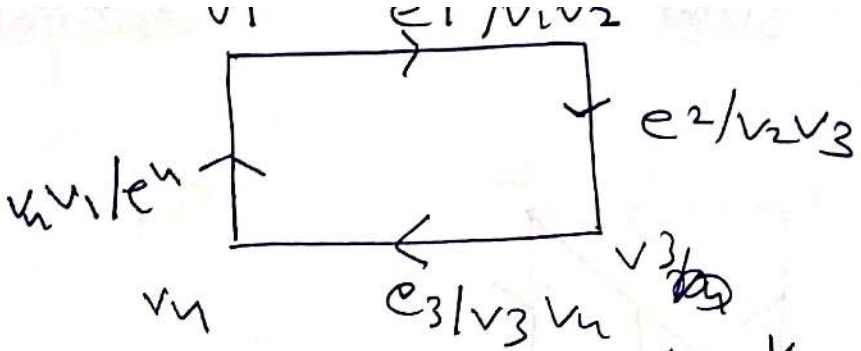
Circuit



Size of a graph → Total no. of edges of the graph.

Directed graph → Source & destination.

$e_1 : v_1 \longrightarrow v_2$



Indegree → Incoming edges to the vertex

Outdegree → Outgoing edges to the vertex

Source vertex: Indegree = 0
(SV) Outdegree = +ve

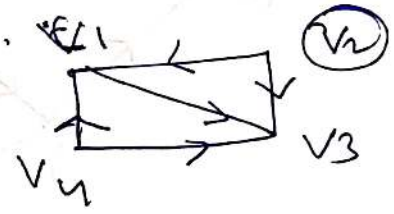
Destination vertex
(DV)

V_1 Indeg = 1 (e_4)
outdeg = 1 (e_1)

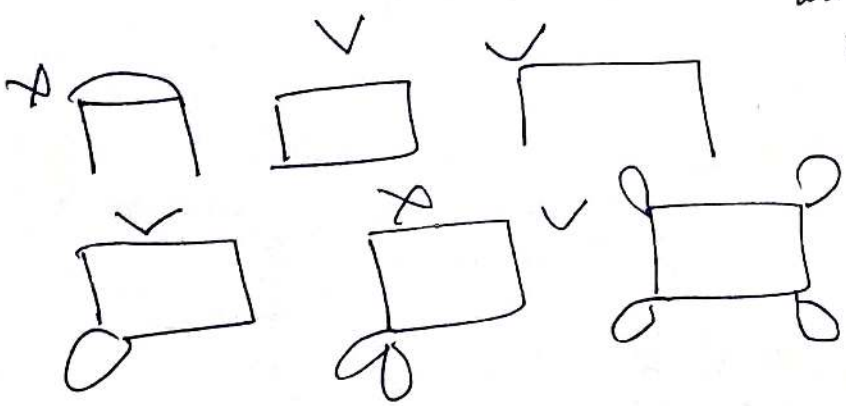
V_2 I = 1 (e_1)
O = 1 (e_2).

SV: I = 0 $V_2 V_4$
O = +ve

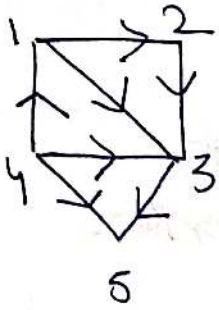
DV: I = +ve V_3
O = 0.

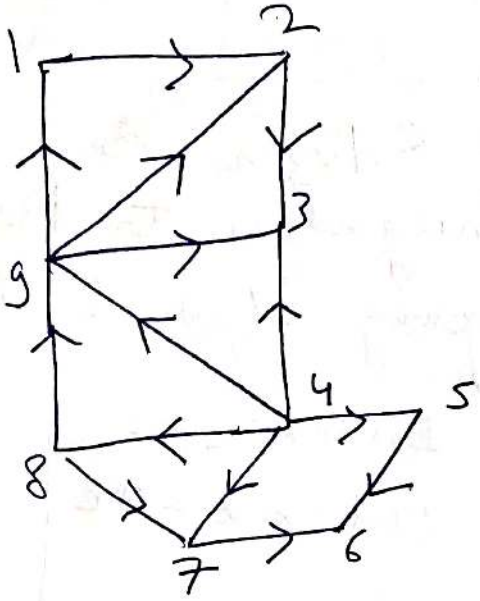


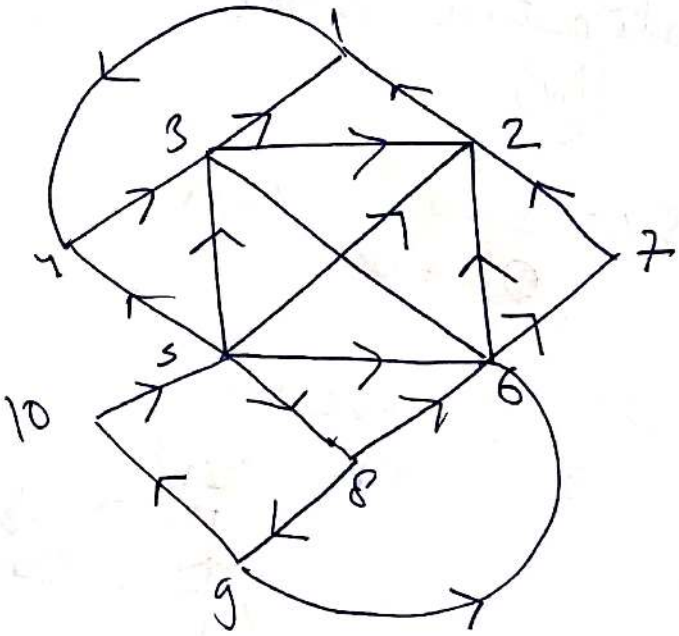
Simple directed graphs → No parallel edges.
→ 1 self loop at max allowed for each vertex.



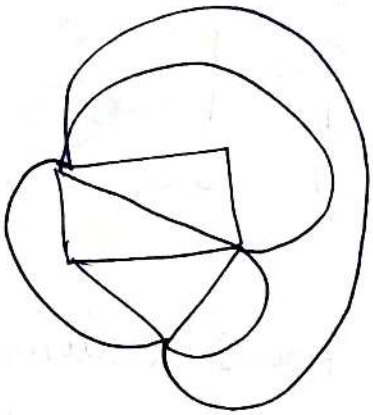
Q7 I=? O=?

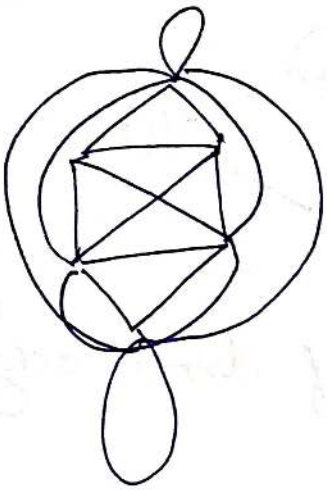






Regular ?
Simple ?
Complite ?

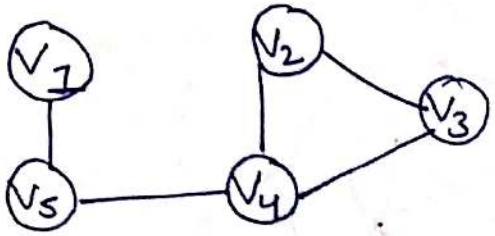




Representation

1 October 2019

1) Adjacent matrix R & C = V X V



	V_1	V_2	V_3	V_4	V_5
V_1	0	1	0	1	1
V_2	1	0	1	1	0
V_3	0	1	0	1	0
V_4	1	1	1	0	1
V_5	1	0	0	1	0

Presence of direct edge = 1

Absence = 0

Space complexity = $O(V^2)$

Time Complexity = $O(1)$ Direct Edge

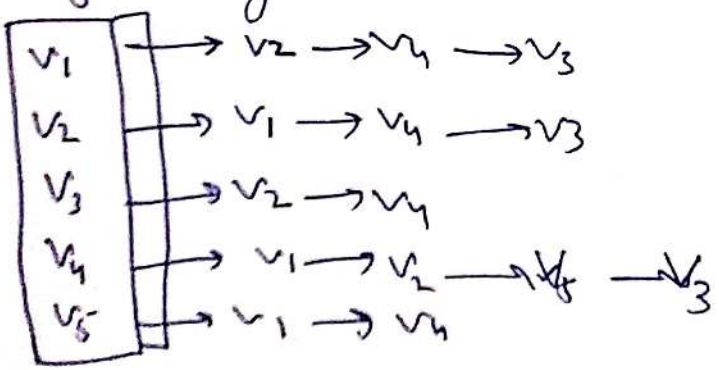
Rows & Columns R X C

V X V = V^2

Space Matrix : in case - not preferred

Referred = Comple Graph - Dense matrix.

2) Adjacency list



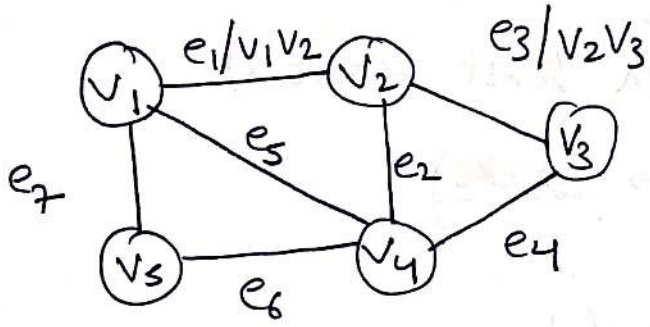
Space complexity = $O(V+E)$

Merit: Space matrix

Demerit: Dense matrix.

3.) Incidence Matrix

R = V
C = E



	e_1	e_2	.....	e_7
v_1	1	0		
v_2	1	1		
v_3	0	0		
v_4	0	1		
v_5	0	0		

SC = O(V X E)
= O(VE)

TC = O(1)

Undirected ($\sharp E$) $v_1 \xrightarrow{e_1} v_2$

Directed AL

v_1 v_2 v_1

v_L

A list is maintained for each vertex (v_i; $v_1 - v_5$) and the list contains all the vertices which has a direct edge from v_i.

- Neither BFS nor DFS traversal of a graph need to be unique; i.e a graph may have many BFS traversals & DFS traversals.

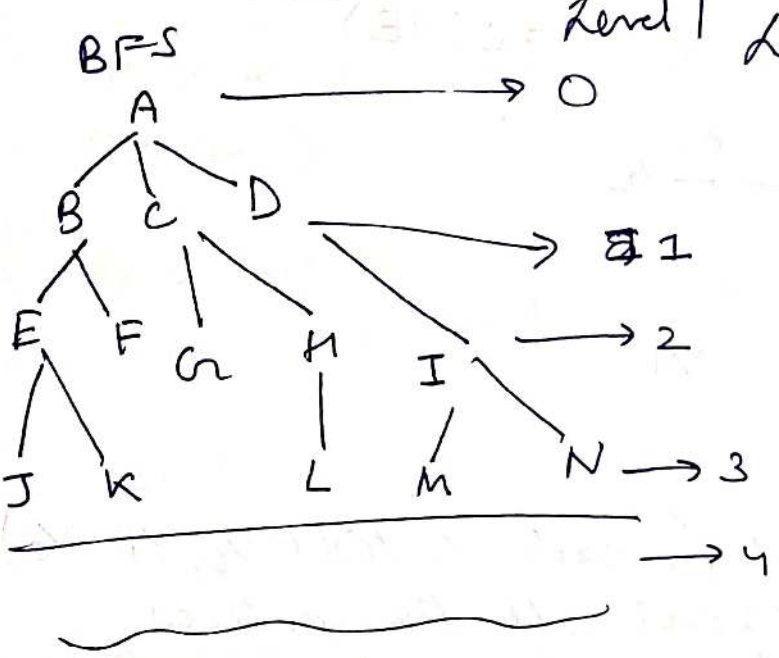
→ Level order traversal (BFS)
→ It discovers the vertices of a graph uniformly across the breadth of the graph.
It discovers all the vertices at distance k from the source before discovering any vertex at distance (k+1) from the source.

Graph Traversal

BFS (Breadth First search)
Processing from front
→ (Queues) FI, FO

DFS (Depth first search)
→ (Stacks)

BFS

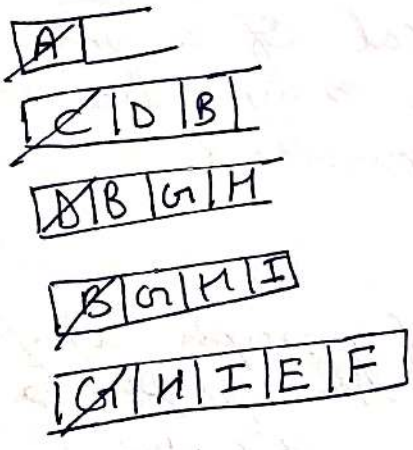


Level 1 Level 1 1 2 3 ...

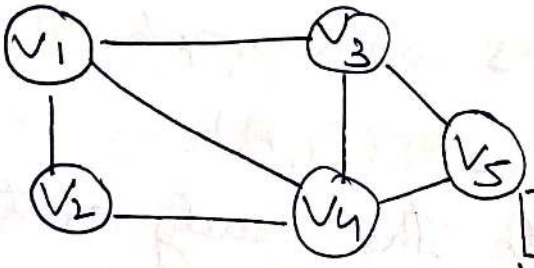
A, B, C, D, F, G H, E, I, J K L M N, O

Ready = 1
waiting = 2
Processed = 3

Yearwise events to Level.



A, C, D, B, G H I E F L M N K



V1	V2	V4	V3	V5

V1 V2 V3

or V1 V3 V2

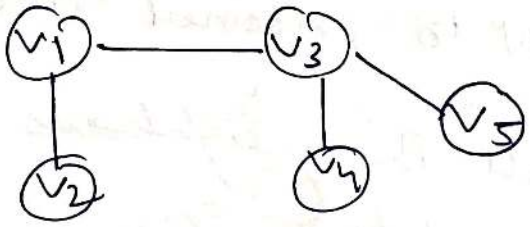
or V1 V2 V3 V5 V4

or V1 V3 V2 V4 V5

V1, V3, V4 V2, V5

Pruned doesn't exist

No copies



V1	V2	V3	V4	V5

V1, V2, V3, V4, V5

- No copies are <u>maintained</u> : Queues.

Algo:
1) This algorithm execute BFS on a graph G at a starting ~~node~~ node A BFS(G, A)

2) Initiatise all the nodes to the ready state (status=1)

Put the starting node A in Q and change its status to the waiting state (status=2)

Repeat step 4 & 5, untill Q is empty

4) Remove the front node on Ques. and change the status of N to processed state

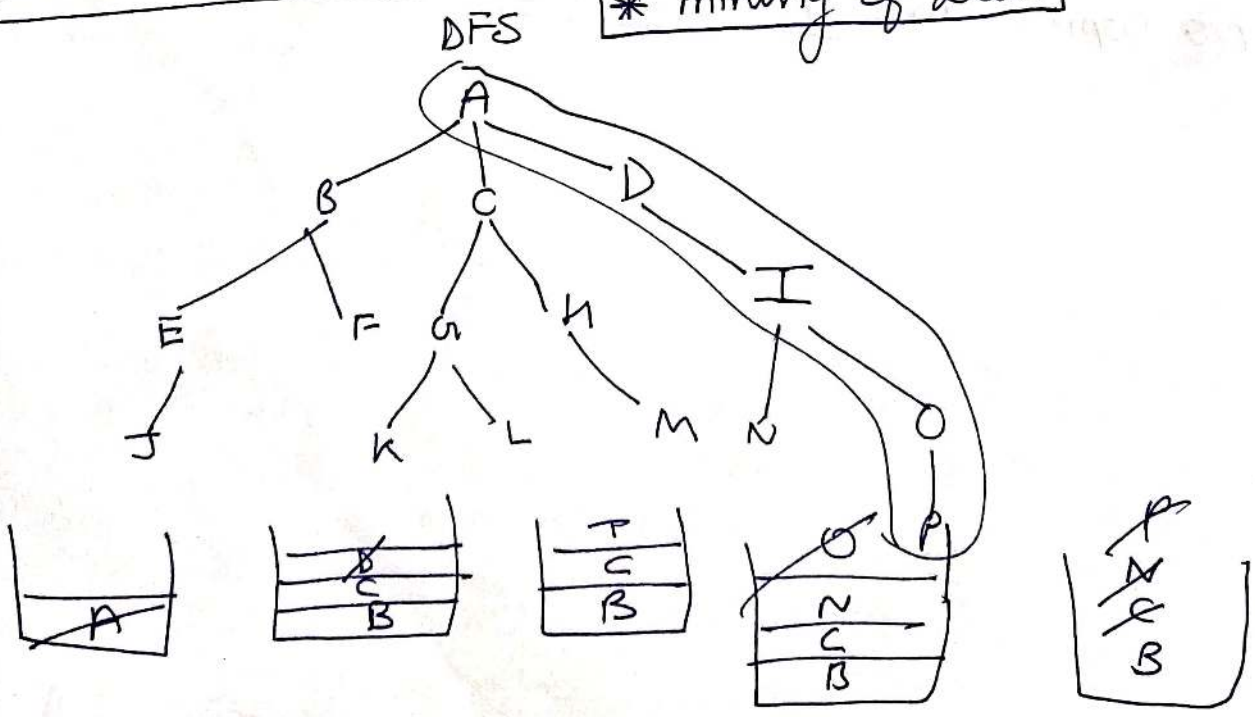
5) Add to rear of the Q, all the neighbours of N that are in the steady state. (1) =2).

6) And change their status in the waiting state.

7) Exit.

---

* mining of Data



A, D, I, O, P, N, C, H, M, G, L, K, B, F, E, J, ...

#1) It searches deeper in graph whenever possible

→ 2) A vertex can be kept multiple times in the stack if it is unvisited.
(Due to property of stack, copies maintained).
but after the copy is deleted, that
therefore discarded.

This Algo

1) Algorithm executes DFS on a Graph G, beginning at a starting node (A), intialise all the nodes

2) Push the starting node A onto stack and change its status to the waiting state.

3) Repeat Step 4 & 5

4) Pop the top node n, of stack. Process n and change its status (Delete)

5) Push onto the stack, all the neighbours of n, that are in the ready state.
and change their status to the neighbour state.

Spanning Tree

Original $G(V, E)$

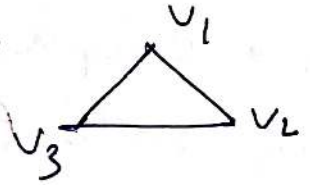
spanning $G'(V, E)$

$|V| = n$

$|E'| = n-1$

There will Exist only one path which mean any pair of vertices A Graph can have

1) G



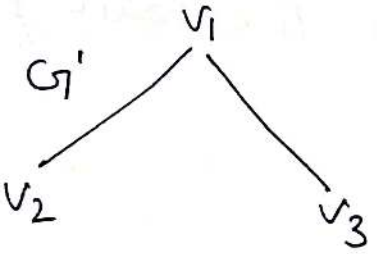
$|V| = 3$

$|E'| = -2$

$V = \{V_1, V_2, V_3\}$

$E = \{V_1V_2, V_2V_3, V_3V_1\}$

G'



Q for a complete graph containing n vertices the total no. of spanning tree (n^{n-2})

1) min Spanning tree aplicable for waited graph

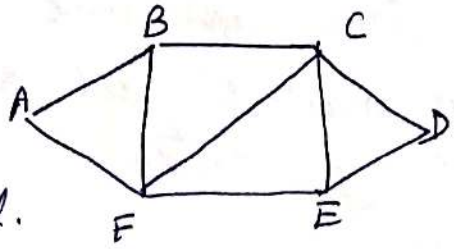
2) A graph non-waited all have same cost a sign to every verten

$MST \cong ST$

3) MST is defined as a spaning tree whose total cost is min. But its cost should be unique.

4) It represent the cheapest way to connect every verten of a Graph.

1) KRUSKAL



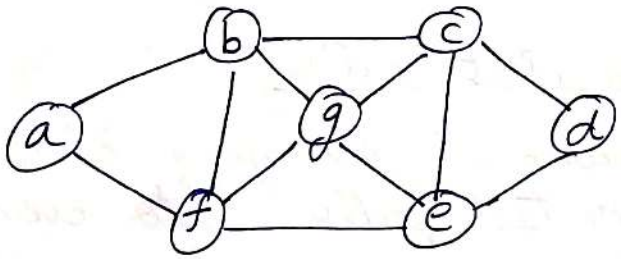
The nodes of the graph are eventually considered.

The edges are arranged in ascending order of weight [Cost

3) The edges are selected one by one & added to Spanning tree such that it should not form a cycle.

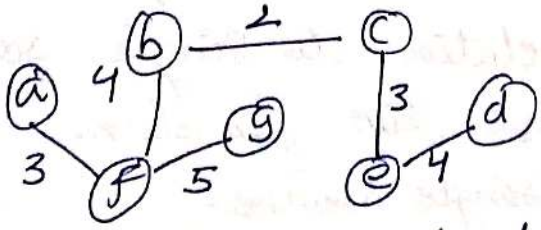
4) In above case, those edges should be selected if & only if both end shouldn't belong to the same tree.

3) PRIMS



$(bc)_2$, $(af)_3$, $(ce)_3$, $(ed)_4$, $(bf)_4$, $(ab)_5$, $(fg)_5$, $(cd)_5$

$(bg)_6$, $(gc)_6$, $(ge)_6$, $(fe)_6$



Unlike KRUSKAL it chooses an arbitary node as a root node (source) any moment it keeps all the vertices, of the graph, in

1) Min. spanning tree aplicable for waited graph
2) A graph non-waited all have same cost a sign to every vertex

MS
one compound.

5) If a node is addeed to the partial then all edges coming out is

6) In each iteration of an algorithm. an edge (U,V) is added to the partial tree, so than exactly one end belong to the set of vertices in orginal graph (G).

7) Out of all possible the edge having the least cost is added to the partial tree, so the tree, so that total weight will be min.

Single seme shortest path

1) from a given sourec (S) Belonging E(V) we want to find the shortest path to every vertex $v \in V$

2) Single source shortest path in a Graph from the tree which is called shortest path tree

3) Shortest path through Vertex need not be unique but cost must be unique.

4) All the Algorithm, related to single source shortest path cause two function.
   a) Initialise Single Source.
   b) Relax (for relaxing edges)

1) I-S---S(G,S)

```
{
  For Each Vertex ∈V
    {
  d[U]= ∞   // distance vector
  π[U]= nil ; // Predecessor vector
    }
  d[S]=0;
}
```

2) relax (U,V)

```
{
  if [d[V] > d[U]+w(U,V)]
      d[V]= d[U] +w(U,V);
          π[V]=4
}
```

The process of relaxing an edge (U,V) consist of testing, we can improve, the shortest to V, found so far, by going through U.

If it is possible then we update d&π (A) relaxation step may dec. the value of shortest path as remain.

And predecessor filled by V.

DIJKSTRA ALGORITHM

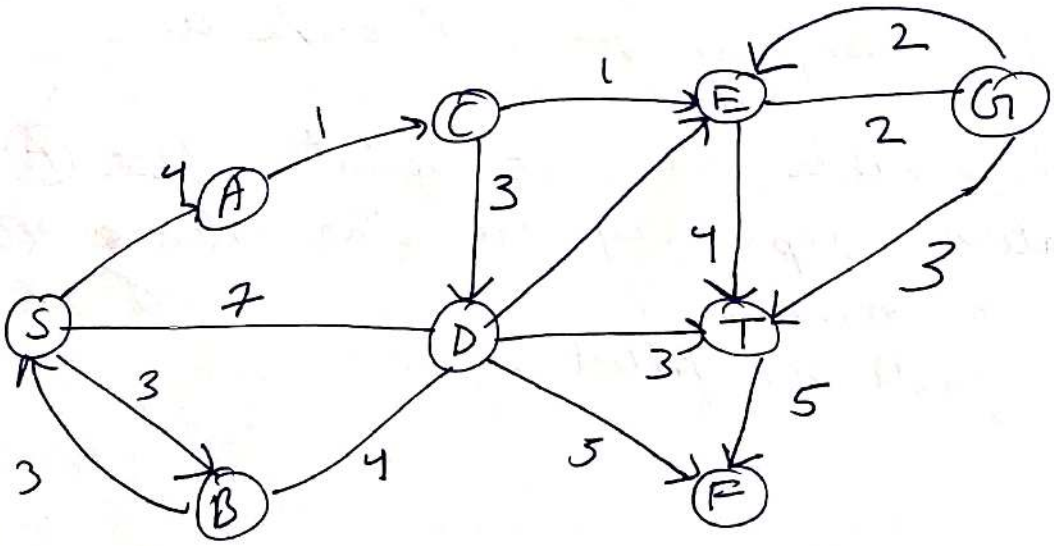
1) This algorithm solve single source shortest path algorithm. problem on a waited Graph (G) G(V, t) in which, all edges are non-negative (it doesn't) recognised an edge whose cost is negative.

2) It maintain a set of vertices whose final shortest path from source s, have already been determined.

3) the Algorithm, repeatedly select the verten 'u' which belong to (V-S) here V is set of vertices and S is visited vertices.

4) with the minimum shortest path estimate of and (u) to (s) & relax all edge leaving u.

Q



(SA) (SB) (SD)
4 3 7
shortest

V = All-S
S = {S}

G(All-S = V(G))
S = = S(G)

V = All{S}
S = {S, b}

Shortest — (SA)(SD)
4 7
(SBD)
3+4=7

step 3

V(G1) = All - {S, b, a}
S(G1) = {S, b, a}

SAG1 (4+1=5) SD (7) SBD (4+3)=7

step 4

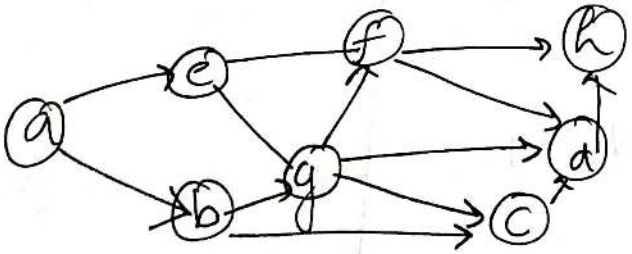
V = All = {S, b, a, c}

S = {S, b, a, c}

{SACt} 4+1+1=6 {SD} 7 {SBD} 4+3=7

SACD
{4+1+3=8}

9



a, b, e, c
S = {a, b, e, c}
S = {a, b, e, c, f}

'a'
S = {a}

acf, aeg, abg, abc
3 5 5 ~~x~~

a b e c
~~acf~~, aeg, abg, abcd
3 5 5 4

aeth, aetd, aeg, abg, abcd
5 6 5 5 4

$S = \{a, b, e, c, f, d\}$

aeth, aeg, abg
5 5 5

$S = \{a, b, e, c, f, d, g\}$

aeth, ab, cd, h

$S = \{a, b, e, c, f, d, h\}$

Algorithm (G, W, S)

1. Initialise – Single source (G, S)
2. $S = \phi$
3. $Q = (G, V)$
4. while $Q = \phi$
5. u = Expect - min(Q)
   (Extent the min cost)
6. $S = SU\{u\}$
7. for each vertex $V \in G$, Adj[u]
8. RELAX(u, V, w)

- PRIMS = $O(E \log V)$
- KRUSKAL = $O(E \log V)$
- Dij = $O(E \log V)$ → Adjacency cost (preferable).
- Dij = $O(V^2)$ Adjacent Matrix

Tree

→ Non linear data structure

→ A tree can be defined an acyclic graph in which there exist only path between any pair of vertices

→ G(V,E)

$|V| = n$

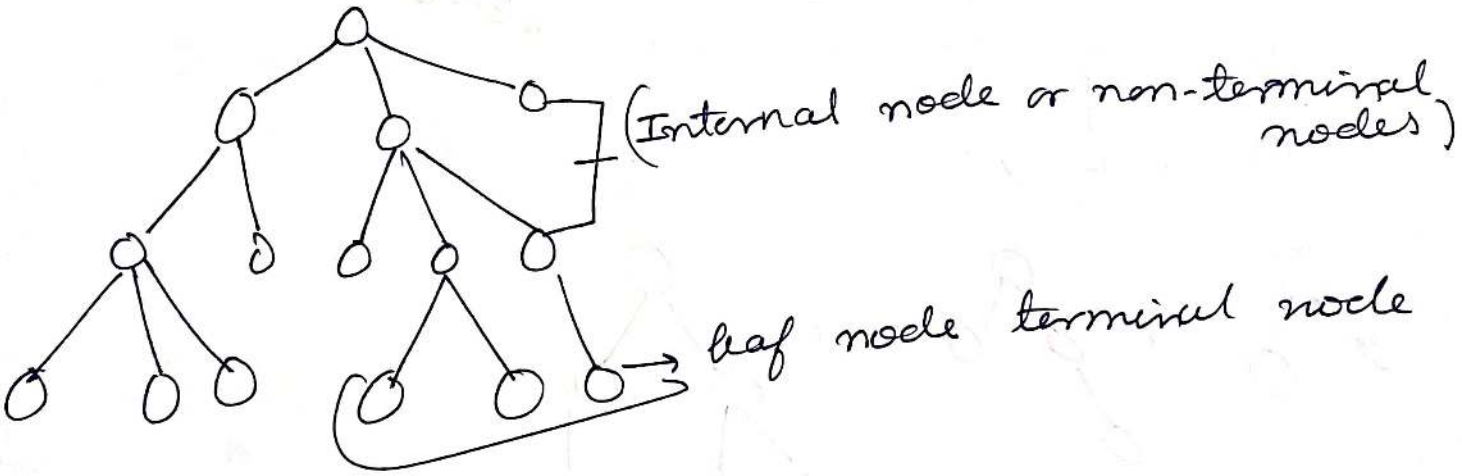
$|E| = m$

$n \leq m$

when

$V = n \quad E = n-1$ → condition for Tree

→ Minimal connected Graph where no. of vertices is n-1 (Root node Parent Node)



Degree of Node → eg (2) → (3)

Degree of a Tree Degree (4+5

The number of children than particular node.

Degree for Tree - The highest degree of node that particular tree.

Height of a Tree H(1) = 3 Node

a L

The no. of edges from a node to the deepest level present.

Depth of Tree:-

The no. of edges from the root to that particular node.

2020 (one year) → Root ↓ node or Root ↑ node

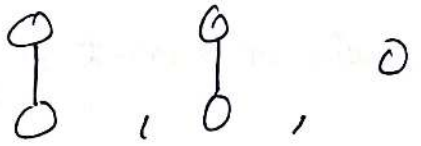
4/8 D(7) = 2

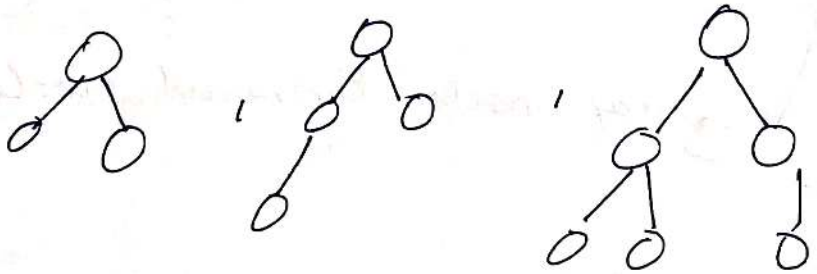
level of the node :-

Binary tree :-

A tree which has atmost two children.

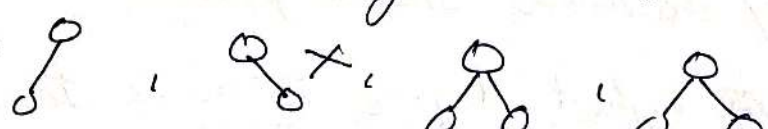
Atmost 2 – 0, 1, 2 (BTS)



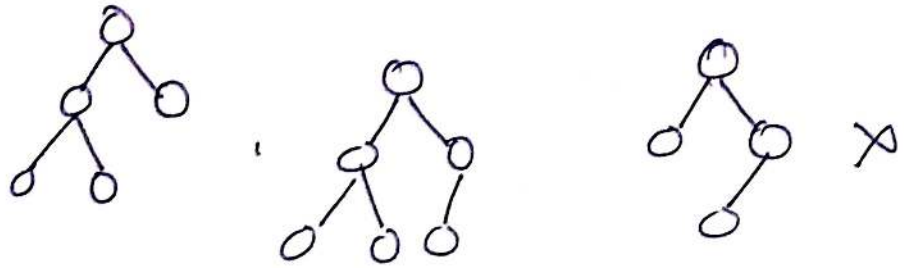


Complete Binary Tree :-

The nodes are being added from left side only.



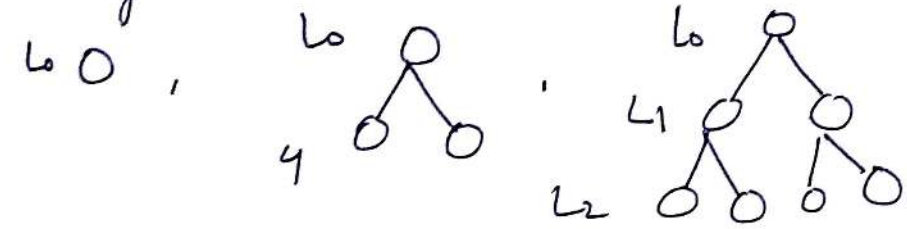
first finalled left side.



Full Binary Tree:-

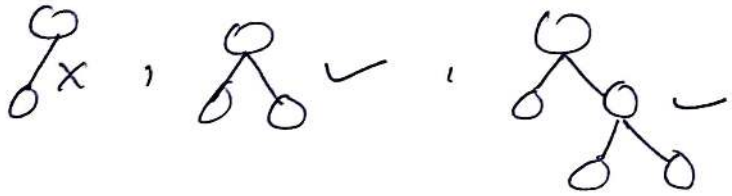
Every node should have two children -

-Every leaf should be complete



Strictly Binary tree

0, or two children are allowed.

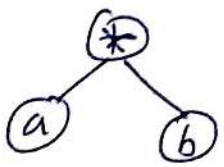


Expression Binary Tree

a*b+c

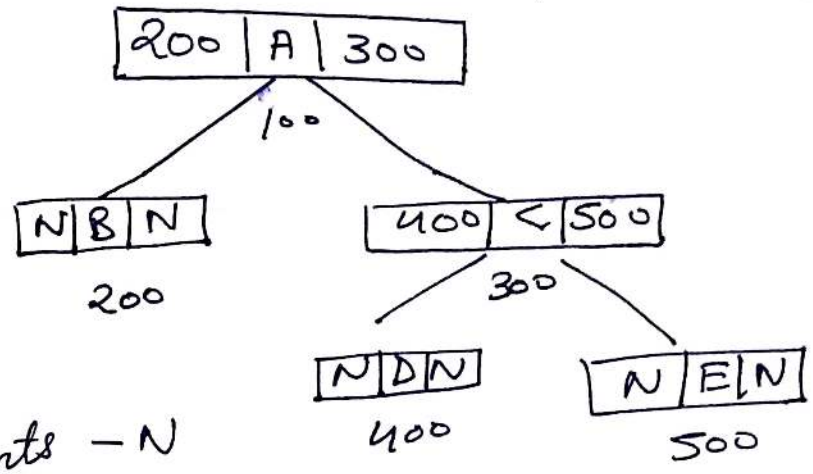
* > + precedence.

a * b



Representation of Trees via Linked list

15 Oct @ 2:06

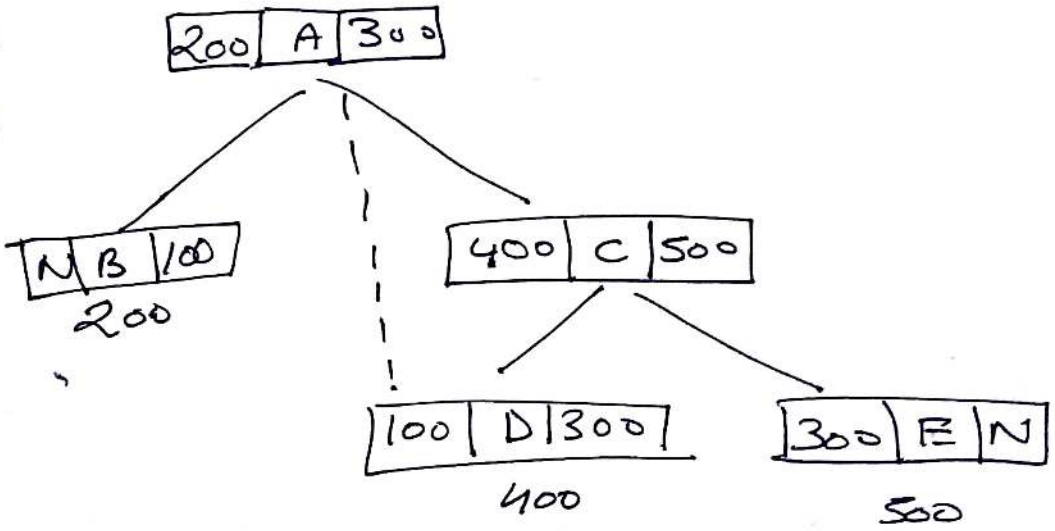


Elements — N

Null — N+1

Thread Binary Tree

6 Null
↓
2 Null



Traversal is in inorder then left null pointer of a node may have the adress of its inorder predessor. and

Right null pointer may have the address of its inorder sucessor.

Inorder to distinguish between the pointer of the child and the pointer of P_s or

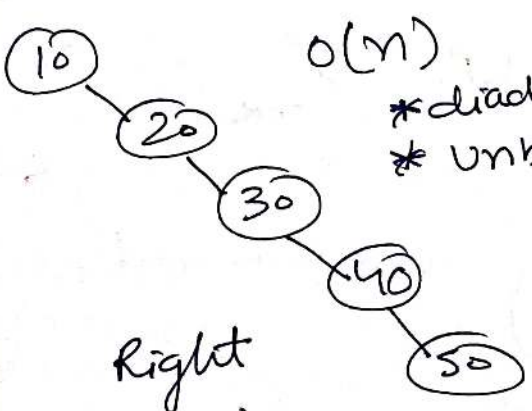
we may need to raise a flag.

The left most node will have no predecessor.

The right most node will have no Successor.

Binary Search Tree

Eg 10, 20, 30, 40, 50

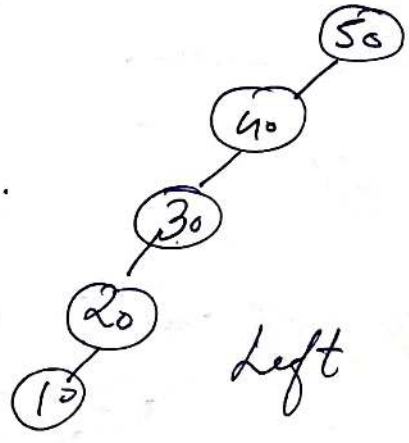


O(n)

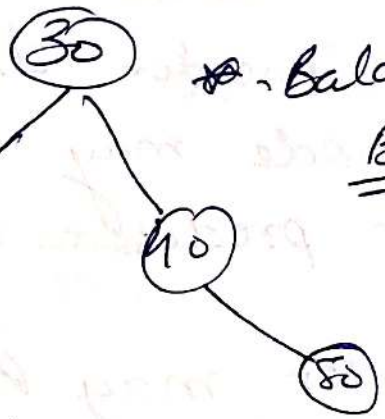
* disadvantage.
* Unbalanced.

Right skew Tree

50, 40, 30, 20, 10



Left skew Tree.



Balanced BST

Best Avg $O(\log n)$.

Overcome disadvantage → (AVL Trees) = BST + High Balancing factor.

GM Adelson - Velski

EM Landis

To tackle problem of skew Trees

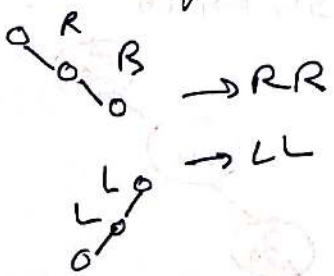
$H = H - H$ skew Tree
subTree

$= 0, -1, +1$

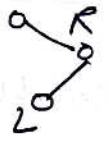
Rotations

Single R

Double R



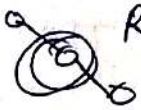
→ RL



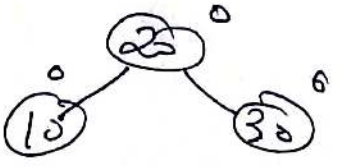
→ LR

Eg.1) 10 $= (-2)$ 20 $= (-1)$ 30 $= 0$ unbalanced

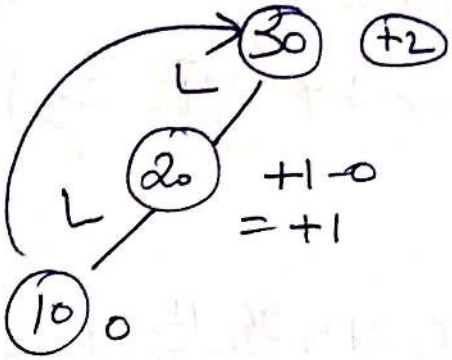
Root



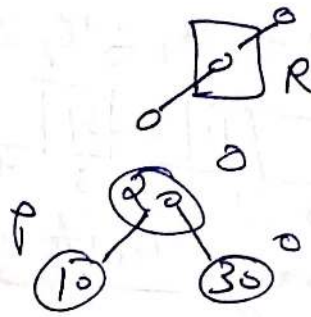
$\Rightarrow$



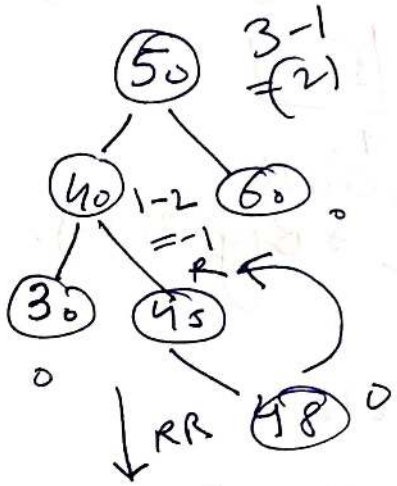
2)



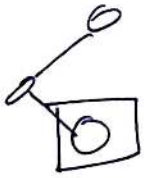
LL Rotation

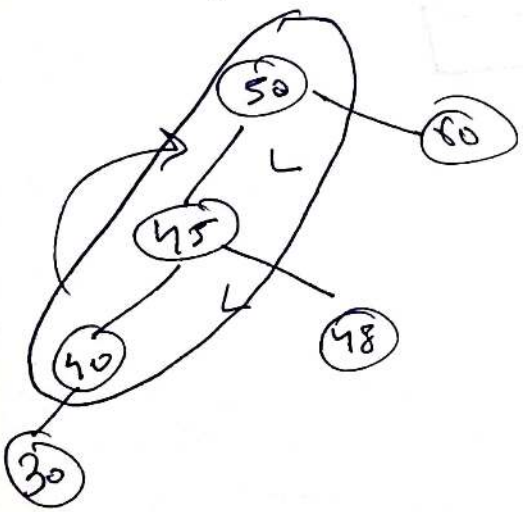


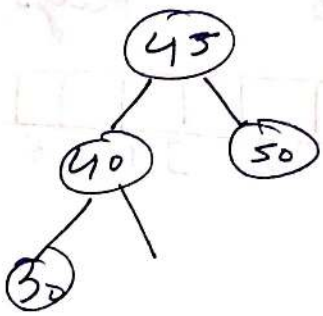
3)



LR Rotation







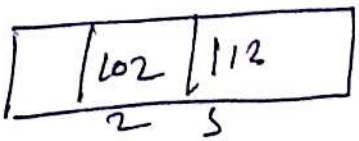
HASHING → Big(O)(1)

COLLISION RESOLVING → Open Addressing
                    → chaining (or separate chaining)

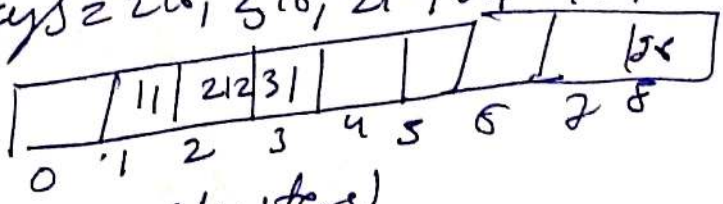
1) open Addressing
2) Linear Probing

m=10

Key = loc



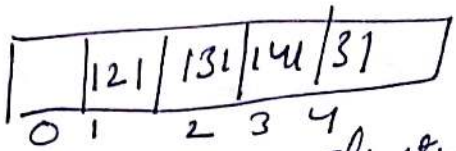
keys = 216, 316, 212, 96, 11, 31



112

collision (primary clusters).

→ Division
→ LP

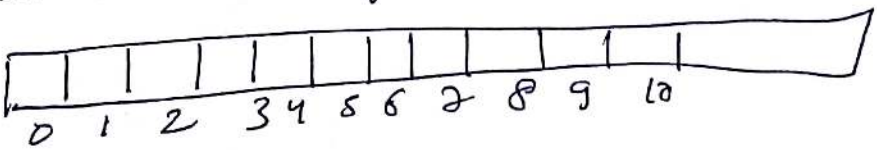


① primary clusters
② → searching → Big O(n).

Quadratic Probing

m=10

keys = 92, 216, 212, 98, 818



Double hashing (* Division | folding | mid-tree)

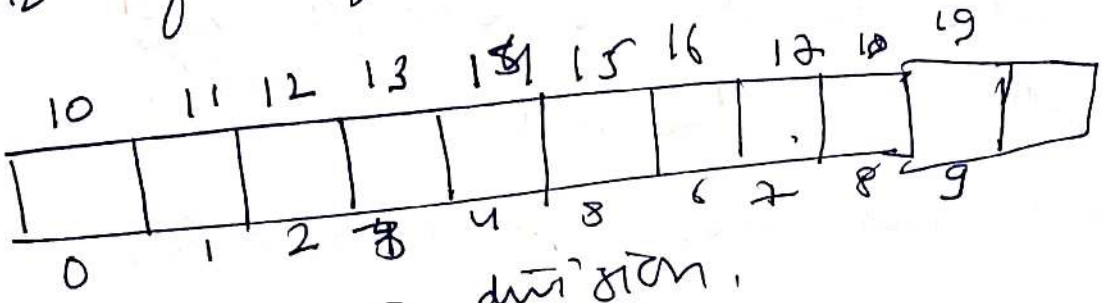
1st Hashing $f \rightarrow H(x)$

2nd Hashing $f \rightarrow H_1(x) * H_2(x)$

Q $m = 10$; key = 212, 111, 312, 36, 916

$H_1(x)$ = Division

$H_2(x)$ = folding



$H_1(x)$ = 2 division.

$H_2(x)$ = 6 (folding)

Red Black Trees.

It is a self Balancing BST, where every node follow certain rules.

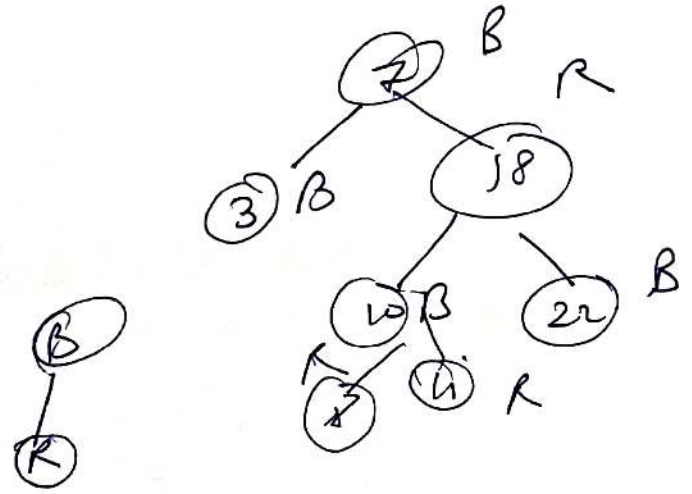
① Every node has either color Red/Black.

② Root of tree is always black.

③ There are no two adjacent Red node

④ (Red node cannot have a Red/Parent child.

④ Every path from the node null node has same no. of blacknode.



Why does Redblack Tree came into existence.

In BST operation (search, max, min, delete) etc

The time complexity is log(n) → no. of nodes

BigO(log(n)) → no. of nodes.

of that particular tree.

The time complexity (h) → h

Therefore RBT is.

Height of the tree is BigO logn)

AVL Tree
more Balanced in comparison to Red Black Trees.
But may cause more rotations during insertion and deletion.

So if our application requires many frequent insertions & deletions, then RB Trees are preferred

But if I/D are less frequent & search is more frequent operation.

Then AV

Black height → root Depth
no of black nodes on a path from root to ~~leaf~~ nodes → Black notes.

From property (3) & (4), we can drive that a (RB) tree of height $\geq \frac{h}{2}$

Dynamic Programming

6 NOV 2019
@ 11:04 AM

Greedy approach & dynamic programming

① Both are used for solving optimisation

②

In dynamic programing, we try to find out the

This approach is bit different and time consuming in comparision to greedy method.

Dynamic P are solved by recurrie formulas. Not recursion of programy, but the formula are indeed. recursive.

Although recursion can also be used,

Dynamic Programming follows ; principle of optimality

A problem can be solved by taking sequence of decisions to get the optimal solution.

4) In greedy method, we need to make our decision only once.

Smethod → sdecision dynmic.

But in DP decision to find the optimal solution is several times.

~~Fibo~~ Fiboracci series.

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

```
int fib(int n)
{ if(n<=1)
      return n;

  return (fib(n-2)+fib(n-1));
}
```

Recursive formula is used.

Dynamic Programming follows ; principle of optimality

A problem can be solved by taking sequence of decisions to get the optimal solution.

4) In greedy method, we need to make our decision only once.

| Smethod → sdecision | dynmic.

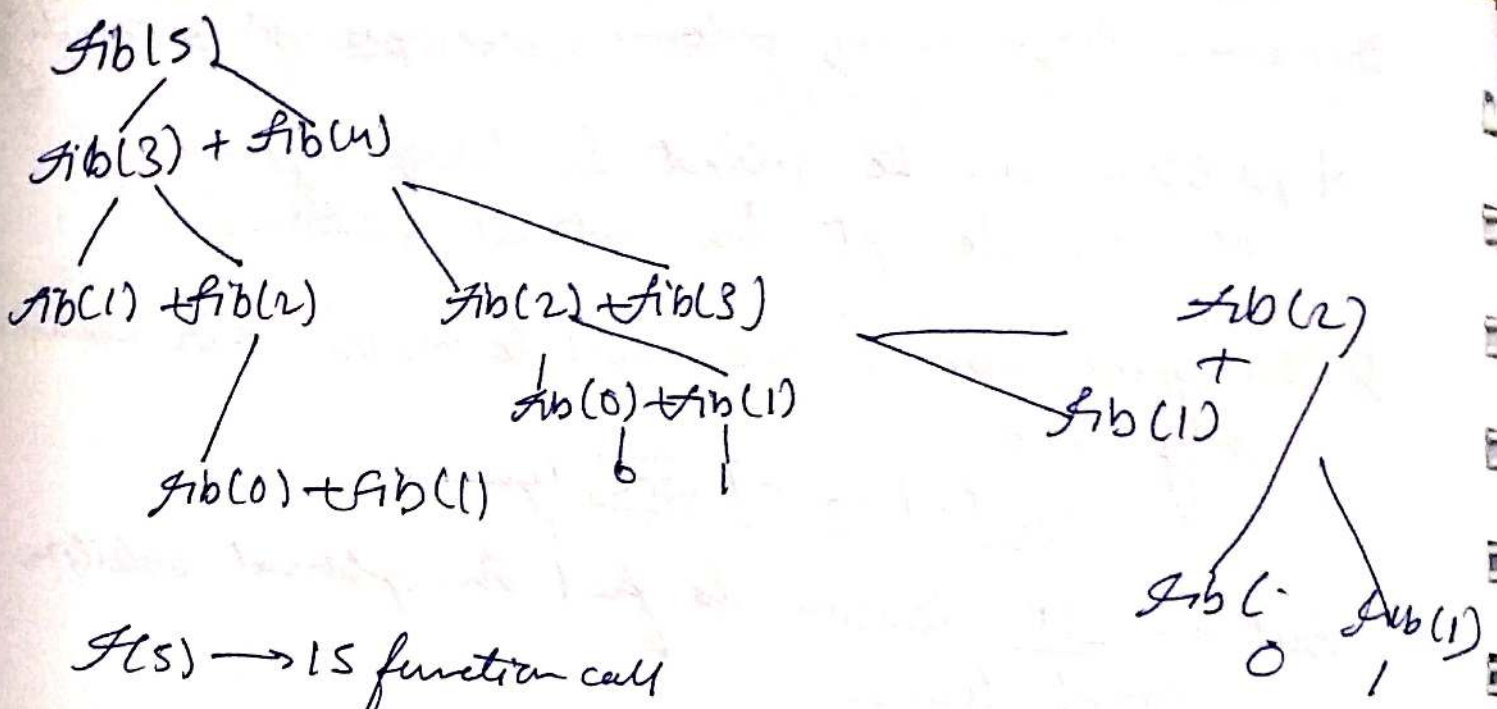
But in DP decision to find the optimal solution is several times.

~~Fibo~~ Fibonacci series.

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

```
int fib(int n)
{ if(n<=1)
      return n;
  return (fib(n-2)+fib(n-1));
}
```

Recursive formula is used.



$F(5) \longrightarrow 15$ function call

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

RR : $T(n) = 2\,T(n-1) + 1$

$O(2^n)$ Master's

Memoization method

Global Array

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

15 function calls $\longrightarrow$ 6 function calls.

$T_C = O(2^n)$ — Exponential.

fib(5) $\longrightarrow$ 6

$n+1$

$\rightarrow O(n)$ Linear.

By storing the result of function; we are avoiding the calls of same function again

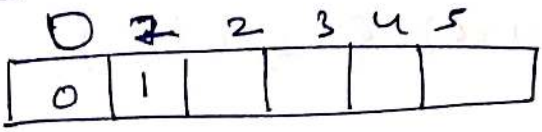
the calls made are conditional calls now & a global array is used to store the result.

Topdown approach.

Memoization method can be used in Dynamic programming But we prefer Tabulation method b/c of its using Iterative property.

Bottom up approach

3) Tabulation.



fib(5).

```
int fb (int n)
{
  if (n<=1)
    return n;
  F(0)=0, F(1)= 1;
    for (int i=2; i<=n, i++);
      f(i)=f(i-2)+f[i-1];
}
return f(n);
```

m CM matrix chain multiplication

$A_1 \cdot A_2 \cdot A_3 \cdot A_4$

(5x4) (4x6) (6x2) (2x3)

A B

(nxm) (mxp) = ↓

BA

(mxp) (nxm)

AXB = C

(5x4)(4x3) (5X3)=15

$\begin{bmatrix} + & + & + & + \end{bmatrix}_{5\times4} \begin{bmatrix} \vdots \\ \vdots \\ \vdots \\ \vdots \end{bmatrix}_{4\times3} = \begin{bmatrix} \cdot & & \end{bmatrix}$ Total no. of elements

$5 \times 3 \times 4 = 60$

Total no. of multiplication / cost.

$A_1 \cdot A_2 \cdot A_3 \cdot A_4$

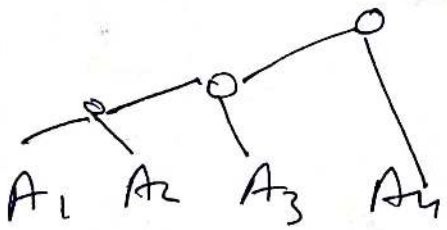
(5x4) (4x6) (6x2) (2x7)

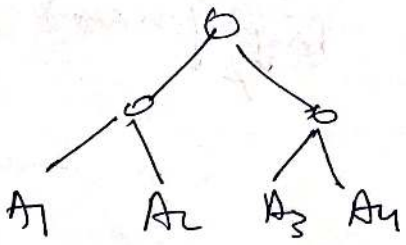
1st $(A_1 \cdot A_2 \cdot A_3) \cdot A_4$

2nd $(A_1 A_2) \cdot (A_3 \cdot A_4)$

We do not need to multiply just provide the ans. But we need to choose the write pair form the parenthesis in order to decrease the total no. of multiplication or the cost of computation.

1st





$$\frac{{}^{8}C_4}{5} = \frac{\frac{8!}{4!\,4!}}{5}$$

$$\frac{\cancel{8} \times 7 \times \cancel{6}^{2} \times \cancel{5} \times \cancel{4} \times \cancel{3} \times \cancel{2}}{\cancel{4} \times \cancel{3} \times \cancel{2} \times \cancel{4} \times \cancel{3} \times \cancel{2}}$$

$$\frac{}{5}$$

Tabulation method

$A_1 \cdot A_2 \cdot A_3 \cdot A_4$
$(5\times4)\ (4\times6)\ (6\times2)\ (2\times7)$

	1	2	3	4
1	0	120	88	158
2		0	48	104
3			0	84
4				0

(cost of Table)

	1	2	3	4
1		1	1	3
2			2	3
3				3
4				

(5)

parenthesis.

① matrix formation

$m[1,1] = A_1 = 0$

$m[2,2] A_2 = 0$

$m[3,3] A_3 = 0$

$m[4,4] A_4 = 0$

$S(1,2) = A_1 | A_2$

$S(2,3) = A_2 | A_3$

$S(3,4) = A_3 | A_4$

$S(1,3) = A_1 | (A_2 A_3)_S$

② matrix formation

$m[1,2] = A_1 A_2 = 5\times4\times6 = 120$

$m[2,3] = A_2 \times A_3 = 4\times6\times2 = 48$

$m[3,4] = A_3 A_4 = 6\times2\times7 = 84$

3 matrix formation

$m[1,3] = A_1 A_2 A_3$

$A_1(A_2A_3)$ or $(A_1A_2)A_3$

$m[1,1] + m[2,3] + 5\times4\times2$ | $m[1,2] + m[3,3] + 5\times6\times2$

$0 + 48 + 40$ | $120 + 0 + 60$

$A_1 | A_2A_3 = 88$ | 180

$A_2 \rightarrow A_4$

$m[2,4] = A_2 A_3 A_4$

$A_2(A_3A_4)$ or $(A_2A_3)A_4$

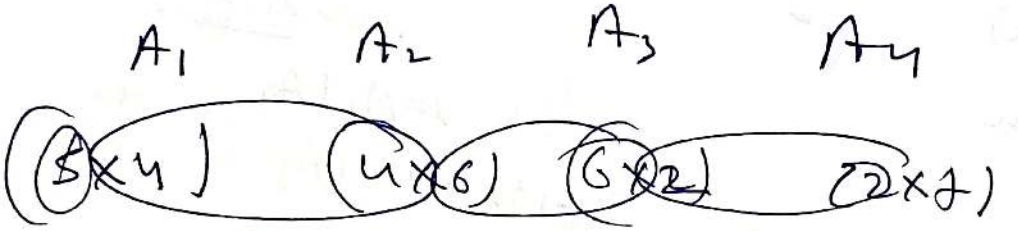
$m[2,2] + m[3,4] + 4\times6\times7$ (252)

$m[2,3] + m[4,4] + 4\times2\times7$ (104)

$m[1,4] = A_1(A_2\ A_3\ A_4)$

$= \min\begin{cases} m[1,1] + m[2,4] + 5\times4\times7 = 0 + 104 + 140 \\ \text{or } m[1,2] + m[3,4] + 5\times6\times7 = 120 + 84 + 210 \\ \text{or } m[1,3] + m[4,4] + 5\times2\times7 = 88 + 0 + 70 = 158 \end{cases}$

$(A_1A_2)\ A_3$



$m[i,j] = \min\{m[i,k] + m[k+1,j] + d_{i-1}\, d_k\, d_j$

min cost = 158

Quick Sort

- Divide & Conquer concept

Dividing element into two unequal sizes.

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

- if dividing into two equal parts = Best case.
- worst case is dividing 1 and n-1.

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

- Based on pivot element which can be chosen from a specific position within the array.

Pivot as j; & following conditions hold.

Each element b/w 0 to (j-1) is less than pivot
Each element b/w j+1 & n-1 will be > pivot element

- for sorted elements

Quick sort : worst case
(Divides in 1 & n-1)

for median calculation = $2T(n/2) + O(n) + O(n)$
$= n \log n$

Algorithm	Time complexity	
	Best	
Heapsort	$\Omega(n \log(n))$	$\Theta(n \log(n))$
Bubble sort	$\Omega(n)$	$\Theta(n^2)$
Insertion sort	$\Omega(n)$	$\Theta(n^2)$

1) Bubble : space complexity $O(1)$
Time complexity $O(n^2)$

2) Insertion ⇒ Time complexity $O(n^2)$ Avg/worst
= Best for (Ie) Best $O(n)$
2nd : (Counting sort is preferred after that)

3) Quick : space complexity = $O(1)$

4) Radix → Not Inplace
→ Stable sort
Time complexity = $O(wn)$
Space complexity = $O(w+n)$

5) Bucket/(Generalised Radix sort)
Bin spacke complexity = $O(n)$
↳ extra array
• stable sort

Heap sort

Recurrence Relation.

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

Smaller inputs.

→ Back substitution / Iterative / Inductive

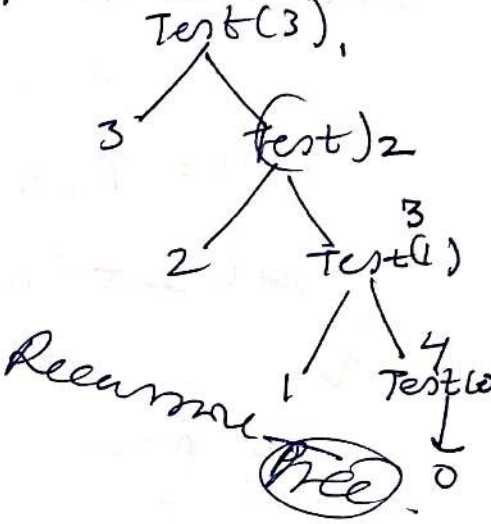
→ Recursive Tree

→ Master's Theorem.

(1) Back substitution / Iterative / Inductive.

```
T(n) → void list (int n)
       {
1 →    if (n>0)
       {
         print ("%d", n);
         test (n-1)
T(n-1) →
       }
}
```

for n = 3



i) Tree → print + calling

Total call = 3 + 1 = 4
↳ calling

Test(n) = n+1 = O(n)

$$T(n) = \begin{cases} 1 & n=0 \\ T(n-1)+1 & n>0 \end{cases}$$ Recurrence Relation.

Back Substitution

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

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

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

$\vdots$

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

$T(0) = 1$

Assume $T(n-k) = T(0)$

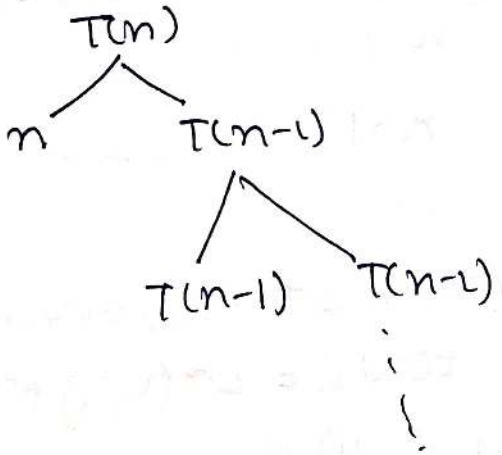
$n - k = 0 \Rightarrow n = k$

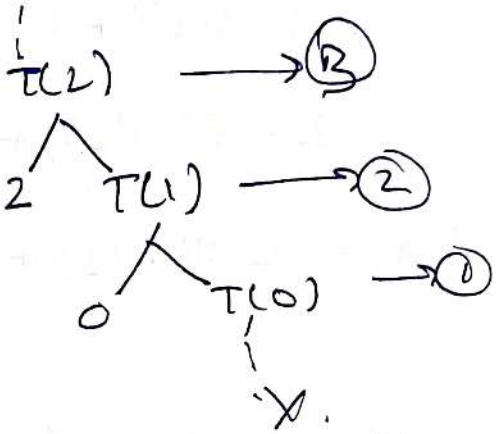
$T(n) = T(n-n) + n = T(0) + n = 1 + n$

```
void test (int n)
{
  if (n>0)
    {for (i=0; i<n ; i++)
      {
       printf ("%d", n);
      }
                          test (n-1);
```

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

Using Recursion Tree





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

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

Back substitution

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

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

$T(n-k) = T(0)$

$n - k = 0 \Rightarrow n = k$

$$= T(0) + \frac{n(n+1)}{2}$$

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

T(n)

```
void test (int n)
{
    if (n > 1)                    // 1
    {
        for (i=0; i<n; i++)       // 1  n  n+1
        {
            ...                   // n
        }
    }
    else
        return;
}
```

$T(n/2) \rightarrow$ test (n/2);

$T(n/2) \rightarrow$ test (n/2);

Back substitution

$$T(n) = 2(T(n/2)) + 3n + 2 \qquad n=1$$

$$\{T(n) = 2(T(n/2)) + n \qquad n>1$$

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

$$\vdots$$

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

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

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

$$= 2^2\left[2T\left(\frac{n}{2^3}\right) + \frac{n}{2^2}\right] + 2n$$

Back substitution

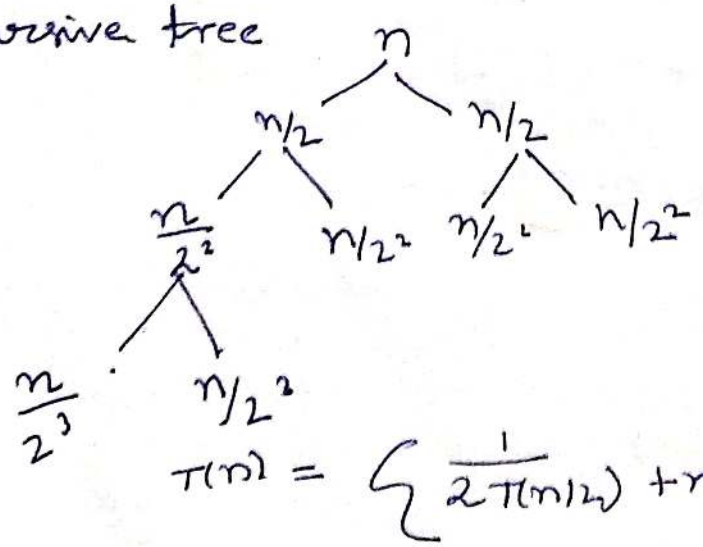
\# $$T_n = 2^k T\left(\frac{n}{2^k}\right) + kn$$

Assume $T\left(\frac{n}{2^k}\right) = T(1)$

$$\frac{n}{2^k} = 1 \Rightarrow n = 2^k \Rightarrow \log n = k$$

$$T(n) = n \times 1 + n\log n = n + n\log n = O(n\log n)$$

Recursive tree



$$T(n) = \begin{cases} 1 & n/2^k. \\ 2T(n/2) + n & \end{cases}$$

master's Theorem (for Dividing functions).

General form: $T(n) = aT\left(\frac{n}{b}\right) + f(n)$

$a \geq 1$

$b \geq 1$

$f(n) = O(n^k \log^p n)$

1) $\log_b a$

2) k.

Case ① $\log_b a$

$\log_b a > k \rightarrow O(_2 \log_b a)$

case ② $\log_b a = k$ i) $p > -1$ $O(n^k \log^{p+1} n$

ii) $p = -1$ $O(n^k / \log \log n)$

iii) $p < -1$ $O(n^k)$

case ③ $\log_b a < k$ i) $p \geq 0$ $(n^k \log^p n)$

ii) $p < 0$ $O(n^k)$

Q $T(n) = 2T(n/2) + 1$

$a = 2$

$b = 2$

$\log_b a = \log_2 2 = 1$

$f(n) = O(1)$

$= O(n^0 \log n)$ $k = 0$

$p = 0$

Case