Sr. No.	Date	Subject	Page No.	Sign.

PAGE NO.
DATE / /

DSA

→ DS - Characteristics
- Correctnes
- Minimize time & space complexity

Need
- Search data
- Handle multiple requests
- Processor speed high

Execution Time Cases
Worst, Average, Best case

→ Algo - Characteristics
Unambiguous, well defined I/O, finite, feasible, independent

Analysis - A Priori - Theoretical analysis
A Posteriori - Empirical Analysis (Post Implementation)

Space Complexity :- Amt of memory space needed by algo.
Components - Fixed :- Independent of size of problem (constants & simple variables)
- Variable - Depends on size of problems (dynamic mem. allocation, recursion stack etc)

PAGE No.
DATE / /

Time Complexity :- Time needed by algo to run to completion

→ Asymptomatic Analysis :-
Define mathematical bounds of algo's runtime performance input bound

O Notation - Worst Case (Upper Bound)
Ω Notation - Best Case (Lower Bound)
Θ Notation - Avg Case

1. ARRAYS - Holds fixed no. of items of same type.
Can be 1D or ND

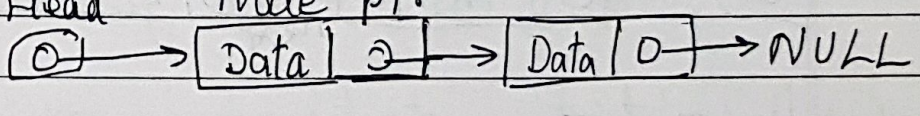
Traverse :- Simple for loop

Insert (At posn k)
Step1 Start
Step2 :- Repeat for i = n-1 to k
shift elt by one position down from i to i+1
Step3 :- set arr[k] = item
Step4 :- n ++
Step5 :- Exit

PAGE No.
DATE / /

Delete
1. Start
2. For i<k to n
   arr[i] = arr[i+1]
3. n--
4. Exit

2. LINKED LIST
linear collection of data elts (nodes)
Each node has one or more data fields and a pointer to next node
Null ptr denotes end of list
Extra space needed to store ptrs



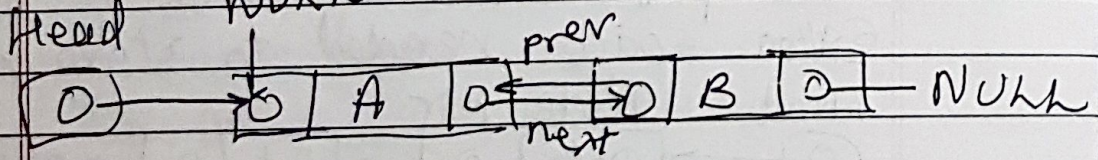
Operations :-
Insert :- Start
1. Link ptr of new node to next node
2. Link ptr of prev node to ~~next~~ new node

Delete
1. Link ptr of prev node to next node
(current node forgotten)

PAGE No.
DATE / /

Reverse Operation:-
1. Traverse to end of list, make pointer (currently null) to its previous node
2. Repeat for all nodes till first node.
3. Set ptr of first node to null
4. Make head pt to last node (from step 1)

→ Doubly linked list.
Each node has 2 pointers pointing to prev & next node



Insert:-
1. Set next ptr of current node to next node & prev of ptr to prev node
2. Make next of prev node & prev of next node point to this node

Handle empty LL, insert at start & at end accordingly

PAGE No.
DATE / /

Other operations :-
DeleteFirst, DeleteLast, Delete,
Display ~~Forward~~ Fwd, Display Backward

⇒ Circular Linked list
First element *prev points to last
& last elt (next) points to first

In single LL :- last node points to first node.

3. Stack :- LIFO Collection
Sequence of items, accessible at only one end of sequence

PUSH - Insert item into stack (top)
POP - Pop item from top of stack
TOP - current location of data in stack
PEEK - gets topmost elt (doesn't delet

PAGE No.
DATE / /

3.1 Polish Notations

Infix → a + b
Prefix → + ab
Postfix → ab +

Postfix must suitable for computers, universally accepted.
Any operation entered in a computer is converted to postfix, stored in stack and then calculated

Operator	Precedence
( )	4
exp {^}	3
* /	2
+ -	1

→ Infix to Postfix
Left paranthesis '(' → Push
Right paranthesis ')' → Pop and print till s[top] != '('
Then pop once more
Operand → Print
Operator → while(pre(in[i]) <= pre(s[top])
pop & print;
push operator

Note:- pre stands for precedence.
∴ While precedence of operator in infix expression <= Precedence of operator at top of stack

PAGE No.
DATE / /

Ex: A+(B*C-(D/E^F)*G)*H

Symbol	Postfix Str	Stack
A	A	
+	A	+
(	A	+ (
B	AB	+ (
*	AB	+ (*
C	ABC	+ (*
-	ABC*	+ (-
(	ABC*	+ (- (
D	ABC*D	+ (- (
/	ABC*D	+ (- (/
E	ABC*DE	+ (- (/
^	ABC*DE	+ (- (/ ^
F	ABC*DEF	+ (- (/ ^
)	ABC*DEF^/	+ (-
*	ABC*DEF^/	+ (- *
G	ABC*DEF^/G	+ (- *
)	ABC*DEF^/G*-	+
*	ABC*DEF^/G*-	+ *
H	ABC*DEF^/G*-H	

Pop full stack

ABC*DEF^/G*-H*+

~~Postfix to Infix~~

PAGE No.
DATE / /

→ Postfix Evaluation

If symbol is operand, push to stack
If operator:- op2 = pop
op1 = pop
Perform op1 operator op2 & push to stack

Ex 5 6 2 + * 12 4 / -

Symbol	Op1	Op2	Value	Stack
5				5
6				5 | 6
2				5 | 6 | 2
+	6	2	8	5 | 8
*	5	8	40	40
12				40 | 12
4				40 | 12 | 4
/	12	4	3	40 | 3
-	40	3	37	37

→ Infix to Prefix

(1) Reverse infix expression. (Note:- While reversing '(' will become ')' and vice versa)
(2) Obtain postfix expr.
(3) Reverse postfix expr.

Ex:- $\underbrace{((a+b)*(c-d))}$
$((d-c)*(b+a))$

Symbol	Prefix Str	Stack
(		(
(		((
d	d	((
−	d	((−
c	dc	((−
)	dc −	(
*	dc −	(*
(	dc −	(* (
b	dc − b	(* (
+	dc − b	(* (+
a	dc − b a	(* (+
)	dc − b a +	(
)	dc − b a + *	

Reverse
* + ab − cd

PAGE No.
DATE / /

4 RECURSION

Technique to break large problem to sub problems and call function recursively

→ Tail Recursion :- If no operations are pending when recursive function returns to caller.

Ex :- factorial :- (Tail)

```
int fact (int n) {
    return fact1 (n,1); }
int fact1 (int n, int res) {
    if (n == 1) return res;
    else
        return fact1 ((n-1), n*res); }
```

Ex Non Tail :-

```
int fact (int n)
{   if (x == 0 || x == 1) {
        return 1;
    else return (x * fact (x-1));
}
```

→ Non Tail Recursion :- Info about pending operation must be stored in stack. Hence, stack increases wt increase in calls.

∴ Tail recursion desirable.

→ Dired Reccursion:- If fnctn calls itself. Ex:- Prev fact fnctn

→ Indirect Reccursion:-

```
int fun(int n){ if (n == 0)
                    return n;
                else
                    return fun2(n); }
int fun2 (int x) {
                    return fun (x - 1);
    }
```

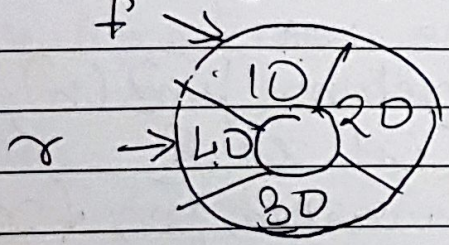
Calls to other function which ultimately calls it.

use of circular queue: Enables addition of elements in posn of deleted elts (prevents waste of space)

PAGE No.
DATE / /

5. QUEUE (FIFO)
Elt that gets in first, also gets out first. Elts added to rear & removed from front.

→ Circular Queue:



To increment front (f) or rear (r) we use
Delete: f = (f+1) % max len
Insert: r = (r+1) % max len
queue[r] = x

queue is full if f == (r+1) % max len
queue is empty if f == -1 and r == -1

→ Double Ended Queue (Dequeue)
Elts can be added and removed at either end
(Both ends must have f and r pointers) - called left and right
Dequeues can be input restricted (insert at one end only) or output restricted (remove from one end only.

PAGE No.
DATE / /

Implemented using circular array/ll

Note:- Stacks & queues can be implemented of arrays & or linked lists

6. ABSTRACT DATA TYPE (ADT)
Logically describing of how we view data and operations (data structures) without regared to how they will be implemented.

⇒ Priority Queue :- Each element is assigned a priority.
Priority determines which elt will be processed first.

ADT is like a blueprint.
It gives requirement & operations
Ex Array :-
Minimal Read Fnctionality :- Get elt at i & set an elt at pos^n & n
get(i) - get elt i
set(i, num) - set elt at pos^n n.

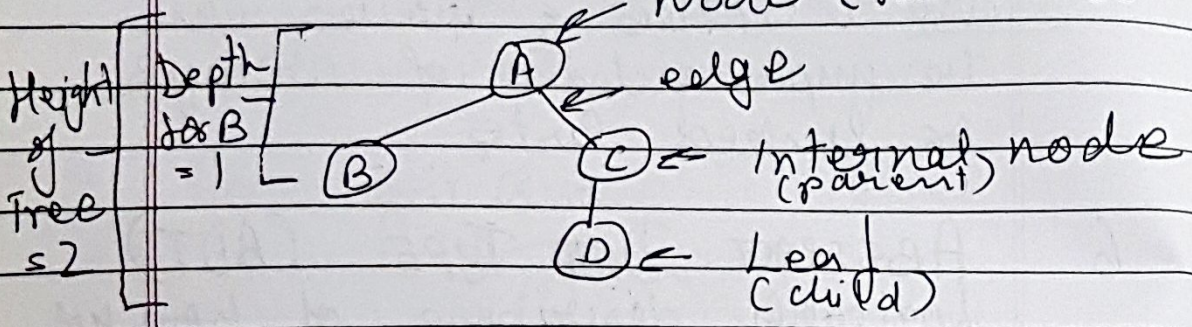
Operations :- max(), min(), search(num), insert(i, num), append(x)

Trees & Graphs →
Non Linear DS

PAGE No. | DATE / /

7. TREES

Non linear, Hierarchial DS

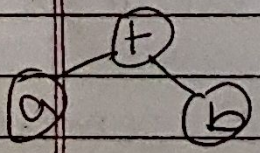


Height → Longest path to a leaf
Depth of a node : Length of path to root

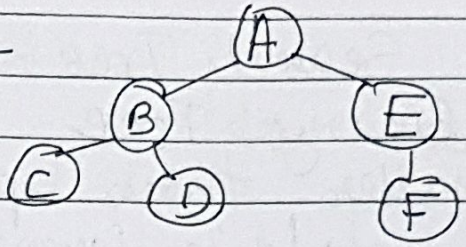
⇒ Binary Tree : Can have atmost two children

- Traversal of Binary Tree :
  L - Left Subtree
  R - Right Subtree
  D - Print Data

1. Inorder (LDR) ⇒ a + b
2. PreOrder (DLR) ⇒ + a b
3. PostOrder (LRD) ⇒ a b +



Ex :-

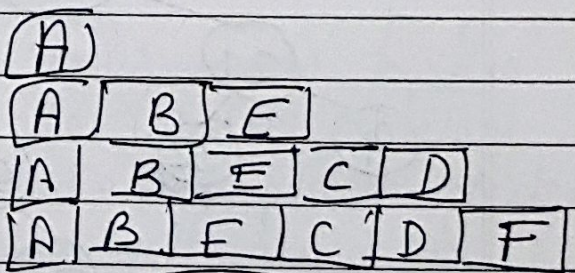


Inorder :- C B D A F E
PreOrder :- A B C D E F
PostOrder C D B F E A

⇒ PreOrder is same as Depth First Search (DFS)

→ Breadth First search (BFS) Traverses tree level by level (Use queue).
It would be like :-
A B E C D F

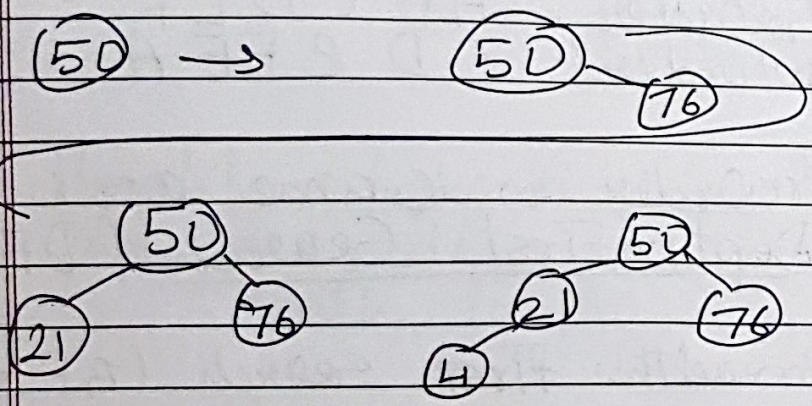
Queue :-



PAGE No.
DATE / /

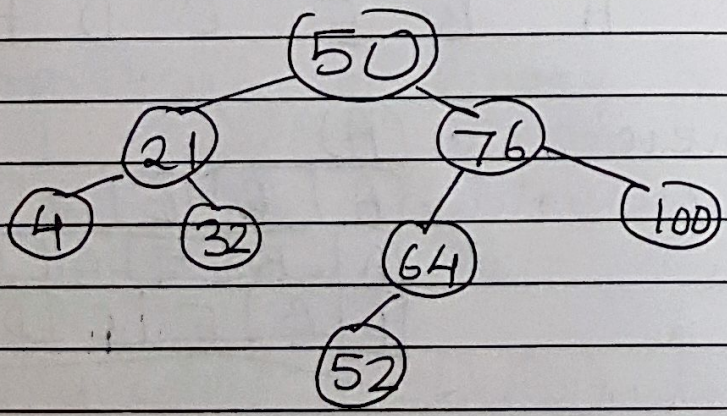
→ Binary Search Tree :-
Sorted Binary Tree. Left child is smaller than parent, right child is larger than parent

Ex :- 50, 76, 21, 4, 32, 100, 64, 52



...

Final Tree :-

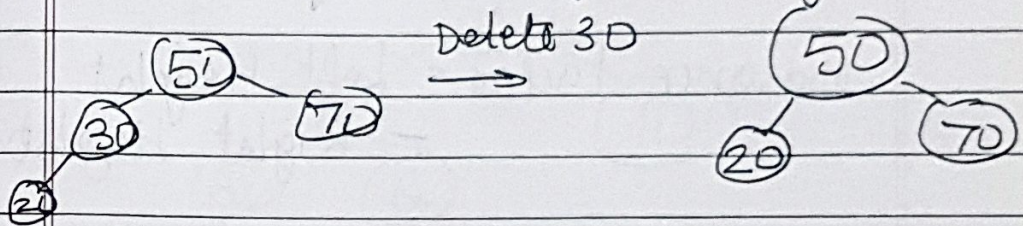


To find a value (x) :-
x == node → return true
x < node → traverse left
x > node → traverse right

Delete :-

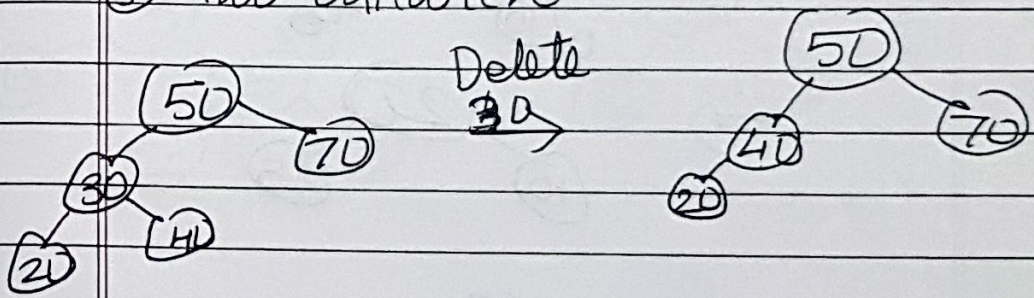
(1) No children (leaf node)
Just delete

(2) One child (Left or Right)



Make parent of node to be deleted point to child.

(3) Two children



Find the min value node to the right of node to be deleted. Add this node in place of deleted node & make left child of deleted node, left child of the new node and right child of deleted node, right child of new node (if applicable)

PAGE No.
DATE / /

→ AVL Tree

Height balanced BST.

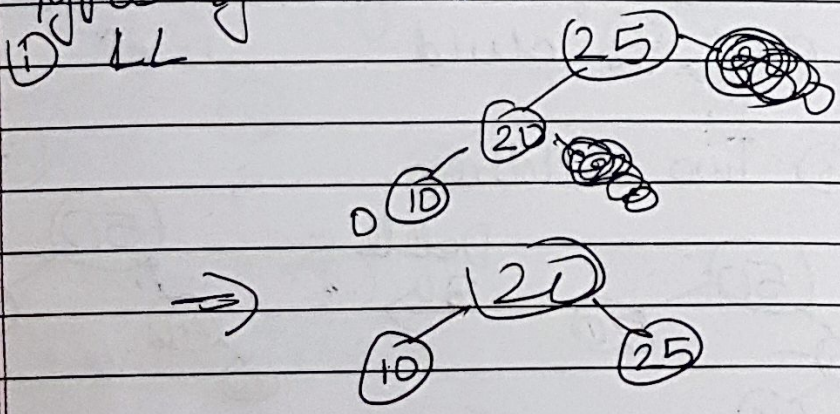
Height of the node kids is at most 1.

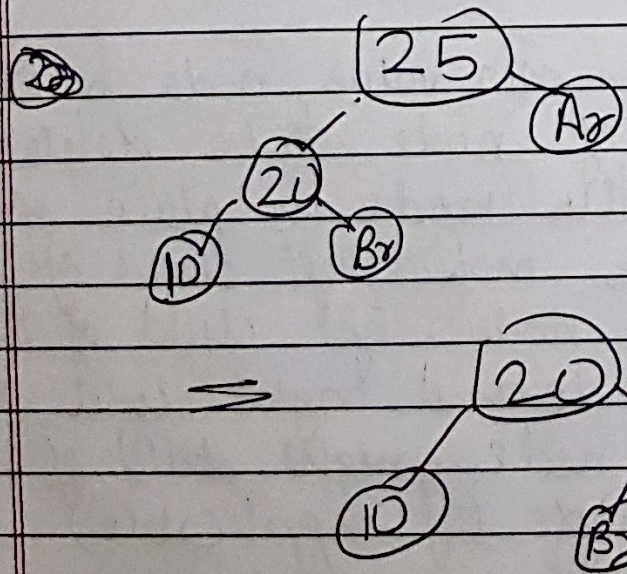
Correct balance factor is 1, 0, -1

Balance Factor = Left_Height − Right_Height

Types of Rotations :-

1) LL

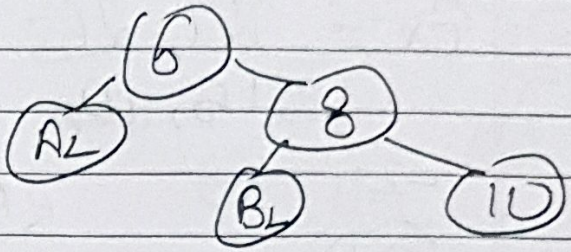




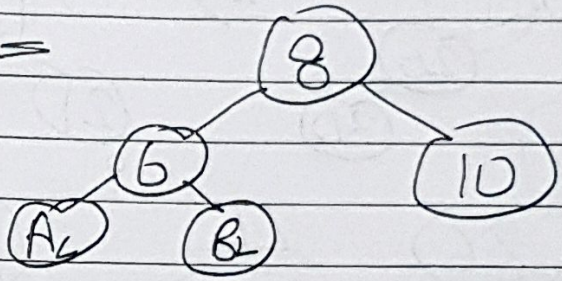
Viewing, Insertion, Deletion
takes O(log n)

PAGE NO
DATE / /

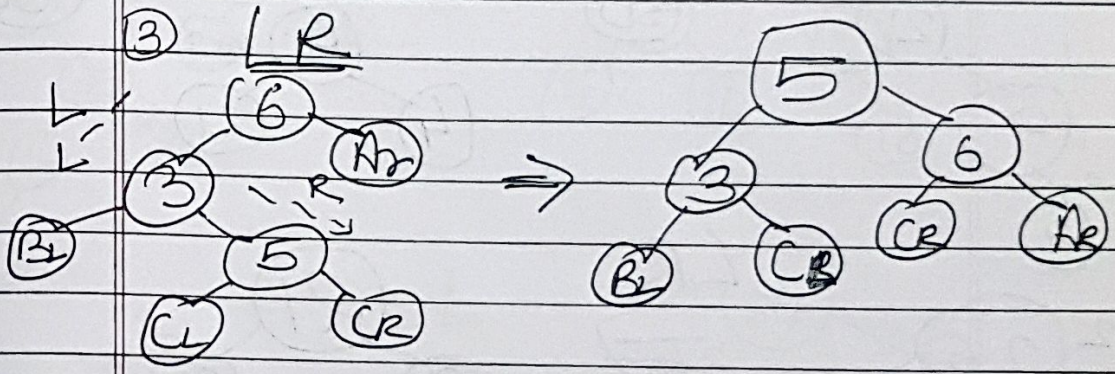
② RR



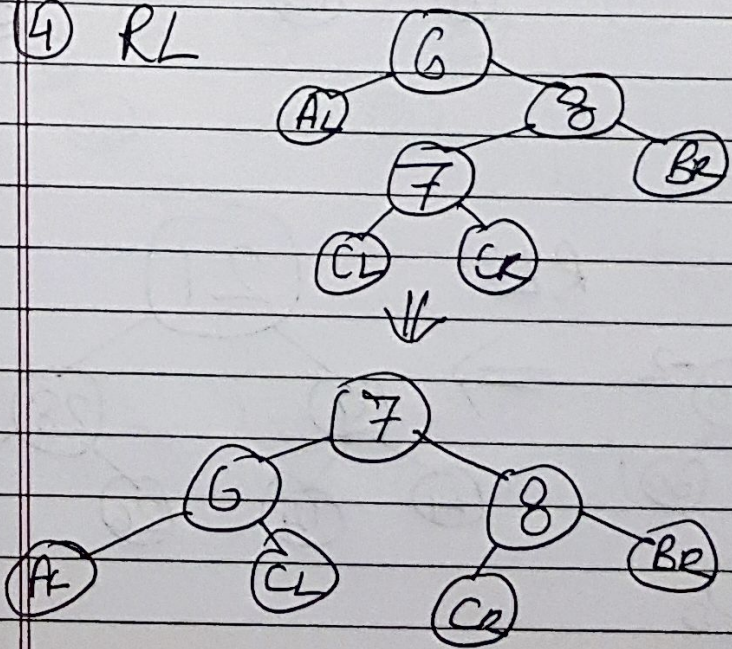
=



③ LR

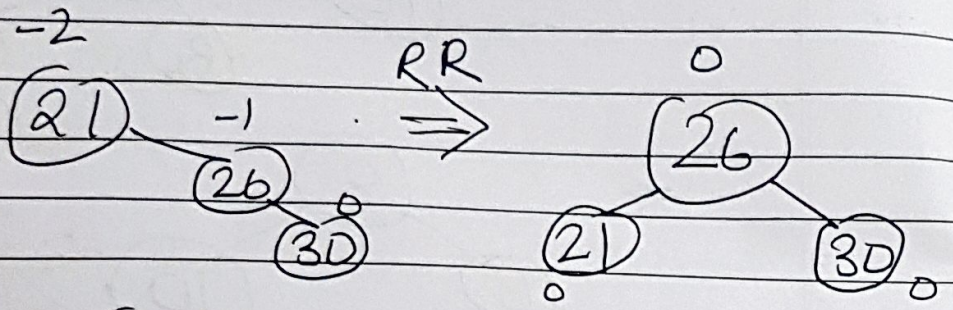


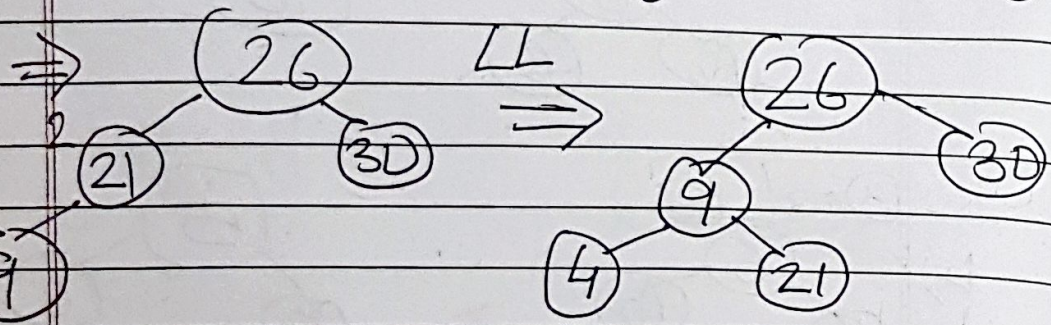
④ RL

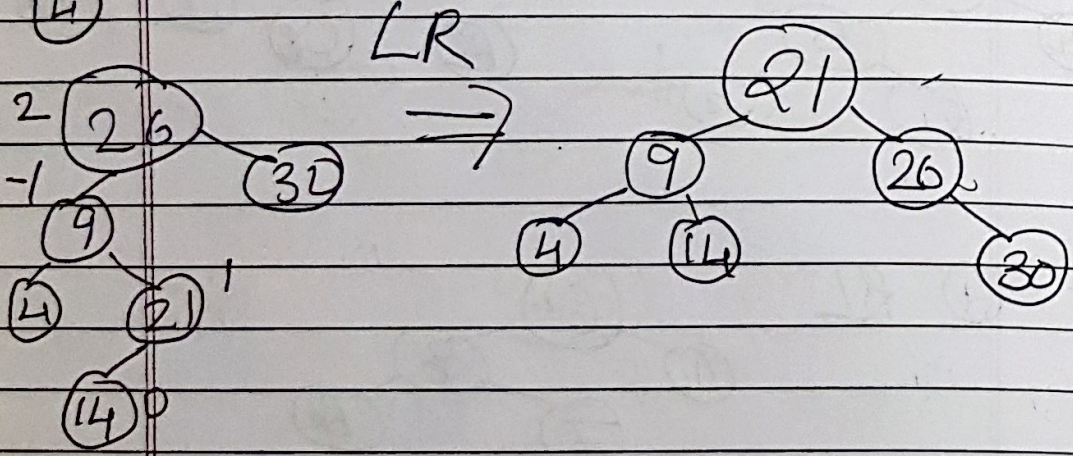


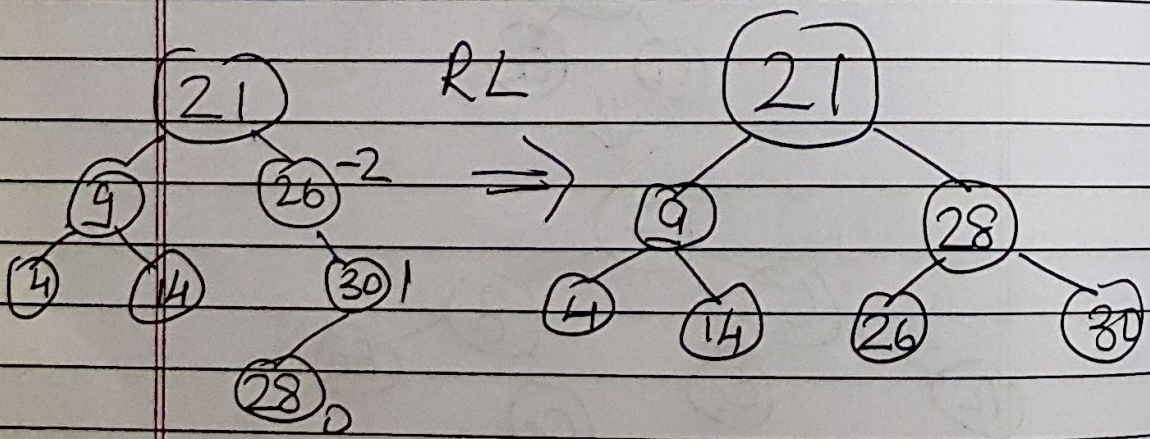
PAGE No.
DATE / /

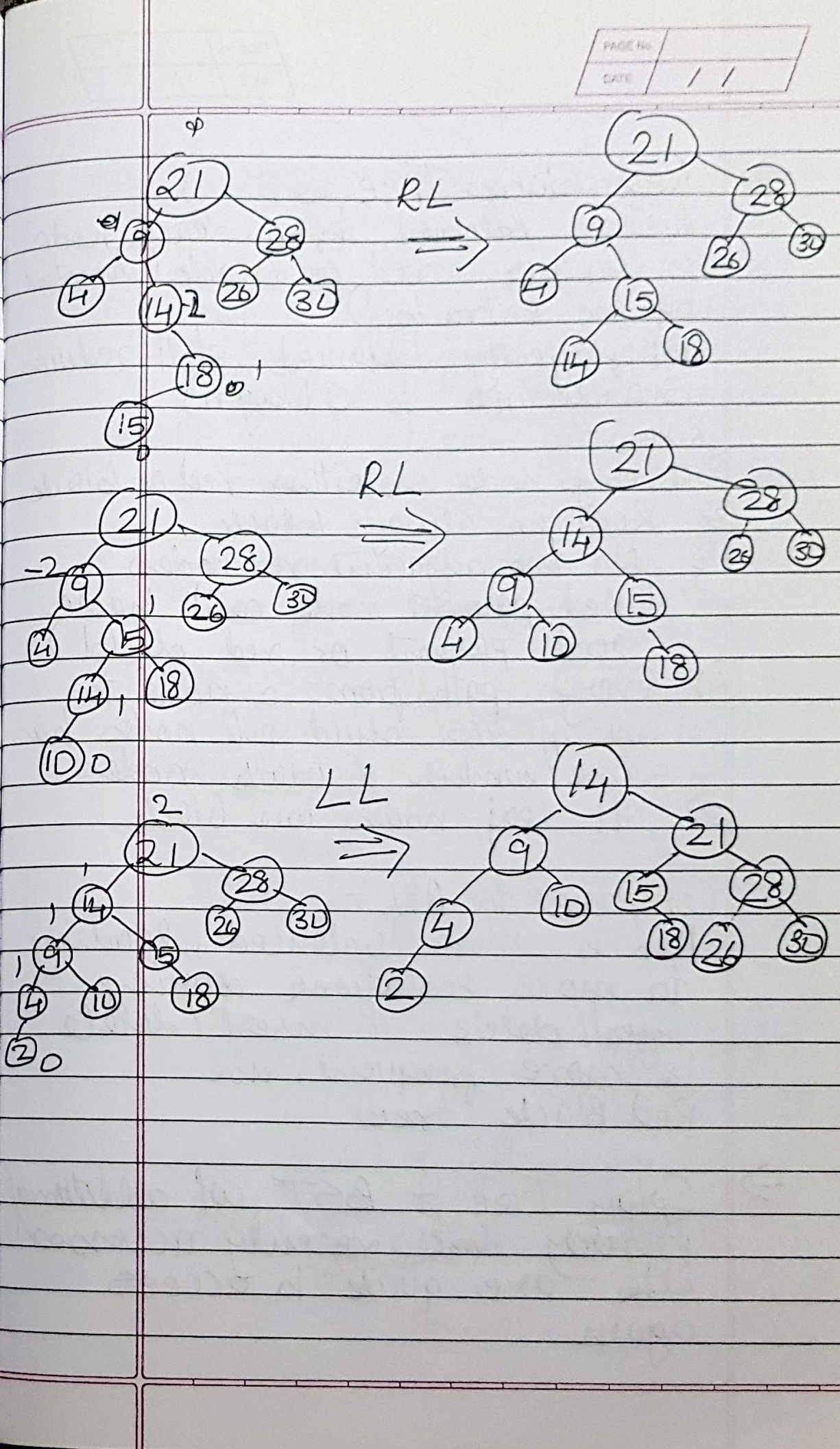
Ex:- 21, 26, 30, 9, 4, 14, 28, 18, 15, 10, 2,










RL
RL
LL

PAGE No.
DATE / /

⇒ Red Black Tree

Another balanced tree. Each node is either red or black (denoted by an extra bit)

Not perfectly balanced, still reduces search time to O(log n)

Rules :-

1. Every node is either red or black
2. Root is always black
3. No two adjacent red nodes (Red ~~parent~~ node can't have red parent or red child)
4. Every path from a node to any of its child null nodes has same number of black nodes
5. All leaf nodes are black

Compare to AVL :-

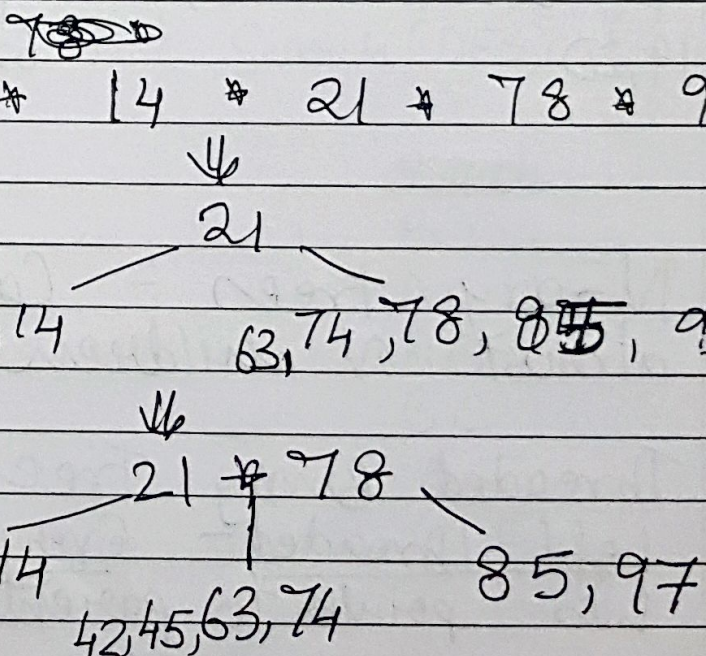
AVL is more balanced, leading to more rotations during insert/delete. If insert/delete is more frequent, use Red Black tree

⇒ Splay Tree :- BST w/ additional property that recently accessed elts are quick to access again

PAGE No.
DATE / /

⇒ B-Tree :- Self balancing tree.
When data is too large to be stored in main memory, it is read of from disk in chunks.
B-Tree designed to minimize disk access O(logn) complexity.
- All leaves at same level
- Each node can hold m-1 keys where m is disk size.
- Number of children = no. of key + 1
- Tree grows & shrinks at root

Ex:- 78, 21, 14, 11, 97, 85, 74, 63, 45, 42, 57, 20, 16, 19, 30 32, 30, 31 m = 5



Root will split

⇒ N-ary trees :- Can have atmost N childrens

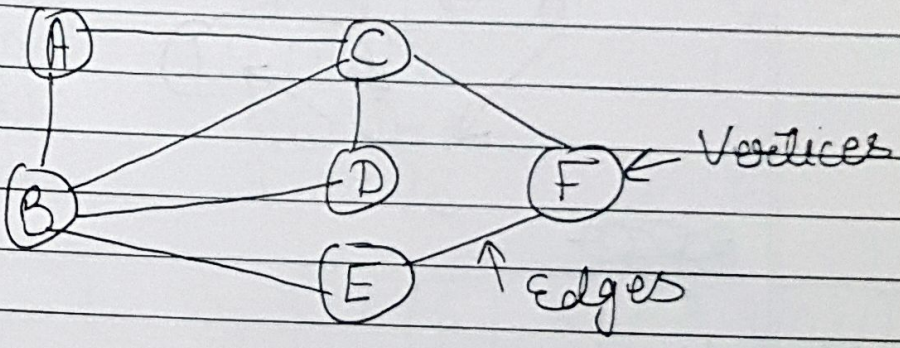
⇒ Threaded Binary Tree

Left Threaded :- Every right child has pointer to parent

Right Threaded :- Left child has pointers to parent

~~8.~~ ~~GRAPH~~
- Two way threaded - Both children point to parent.

8. GRAPH
Non linear DS with vertices/nodes and edges. G(V,E)



Breadth First Search (BFS) (Queue)
Depth First Search (DFS) (Stack)

BFS :-

	~~Stack~~
A	B C
B	A C D F
C	A B D F
D	B C
E	B F
F	C E

Visited

A	B	C	D	E	F
1	1	1	1	1	1

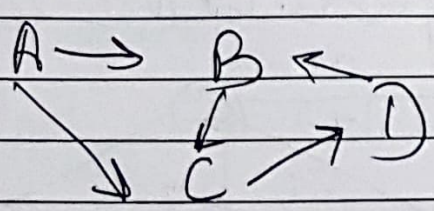
Queue
O/P

A	B	C	D	E	F
A	B	C	D	E	F

PAGE No.
DATE / /

Adjacency Matrix :- Better for dense graph
Adjacency List :- Better for sparse graph

→ Directed Graph :-
Only one direction edge

Ex 

BFS

9. HASHING
Storing data by dynamic DS called Hash table
Table organization and searching technique in which Insert/Delete/search operations done in O(1)
Uses HASH FUNCTION to put data in Hash table.
Enables to store & retrieve data quickly. Can be implemented as an array

Every entry in hash table is associated w/ some KEY.
Hash Function transforms key to a/an ~~for~~ table index.

Key (k) → [Hash Function] → Address h(k)

Ex:-

Name	key (k)	h(k) key%13
DLDA	985926	6
EDLC	970876	10
DSF	986047	11
COA	970428	5

0	1	2	3	4	5	6	7	8	9	10	11	12
					COA	D2DA				EDL	DSF	

Characteristics of good hash fnctn
→ Minimize collision i.e. distribute keys uniformly across the table
→ compute h(k) efficiently
→ Use all info in the key.

→ Collision:- Diff keys when hash to same locations.
Colliding keys are known as Synonyms.

Popular Hash Fnct^ns

- Division Remainder Method (Modulo Division Method)
- Mid Square Method
- Folding Method
- Multiplication Method

① Division Remainder Method :-

$h(k) = k \bmod n$ OR $k \bmod (n+1)$

Decide a n.

ID HUFFMAN ALGORITHM

Used for text compression

Lossy compression - Quality degraded

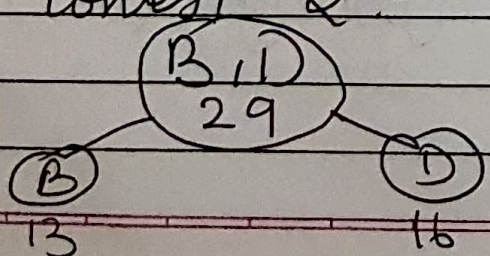
Ex:-

Character →	A	B	C	D	E
Frequency →	22	13	18	16	31

Sort ace to frequency

B	D	C	A	E
13	16	18	22	31

Combine lowest 2 :-

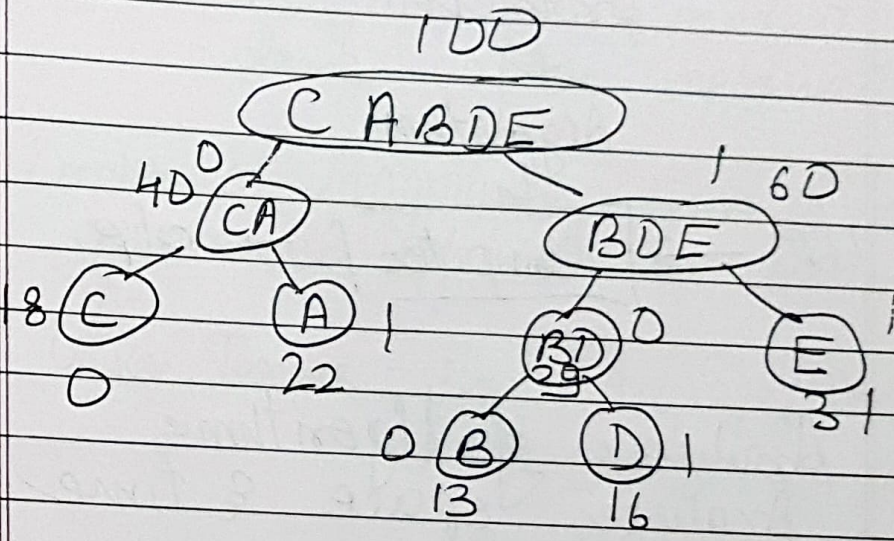


Reorder

C	A	B,D	E
18	22	29	31

B,D	E	C,A
29	31	40

CA	BDE
40	60



Code for each alphabet

A	0 1	22 x 2
B	1 0 0	13 x 3
C	0 0	18 x 2
D	1 0 1	16 x 3
E	1 1	31 x 2
		229 bits

Most frequently occuring character has lesser number of bits.

Before compression (100 + 40 + 60 + 29 + 31 + 18 + 22 + 31 + 13 + 16) = 8 000 bits

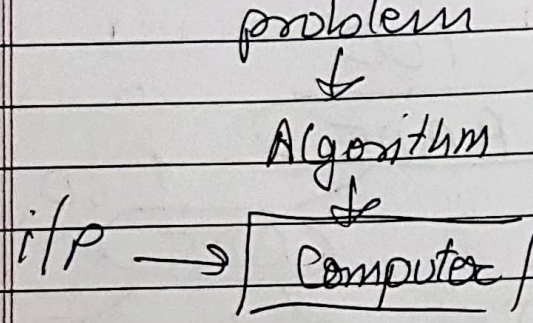
After compression = 229 bits

PAGE No.
DATE / /

Analysis of Algorithms

1 Introduction

Algorithm - Well defined computational procedure that takes some value or set of values as input & produces value(s) as output



→ Analysis of Algorithms
Analyze space & time complexity

Algorithms can be specified in natural language or pseudo code

Time Complexity :- Amt of computer time an algo needs to execute & get the result

PAGE No.
DATE / /

Space Complexity - Amt of memory needed to run an algo.

⇒ Priori Analysis
- Machine independent,
- done before implementing

⇒ Posteriori Analysis
- Target m/c dependent
- done after implementing

⇒ Order of Magnitude - Sum of no. of occurences of stmts contained in it.
Other Terms :- Best / Average / Worst case Running time

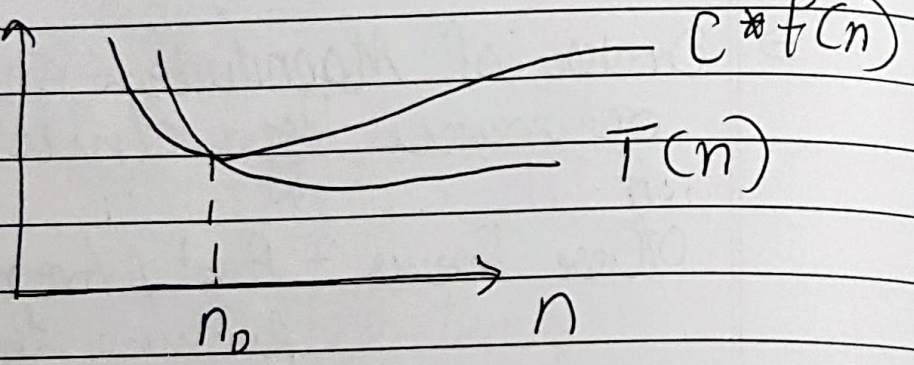
⇒ Asymptotic Notations - How running time of an algo increases w/ size of input (bounded)

i O Notation : Worst Case running time.

For x functions T(n) and f(n) non negative

$T(n) = O(f(n))$ if there are positive constants c, n_0 such that

$T(n) \leq c * f(n)$ for all n, $n \geq n_0$ ($c > 0$ $n_0 \geq 1$)



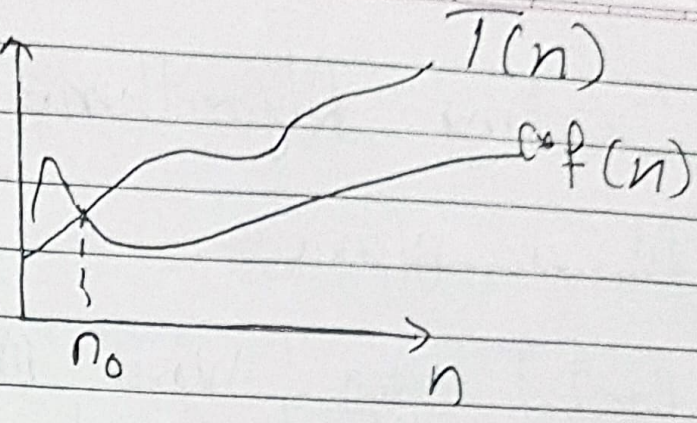
f(n) is the uppere bound

ii Ω Notation : Best Case

$T(n) = \Omega(f(n))$ if there is c and n_0 such that

$T(n) \geq c * f(n)$ for all n, $n \geq n_0$ ($c > 0$, $n_0 \geq 1$)

f(n) is the lower bound



(iii) Θ notation - Average Case
$T(n) = \Theta(g(n))$ if there is ~~some~~ c_1, c_2, n_0 such that

$$c_1 * g(n) \leq T(n) \leq c_2 * g(n)$$

for all n, $n \geq n_0$
$(c_1, c_2 > 0, n_0 \geq 1)$

→ Algorithm of two types
① Iterative - Count no. of times instructions are executed

② Recursive - Recurrence equation

2. Sorting Algorithms

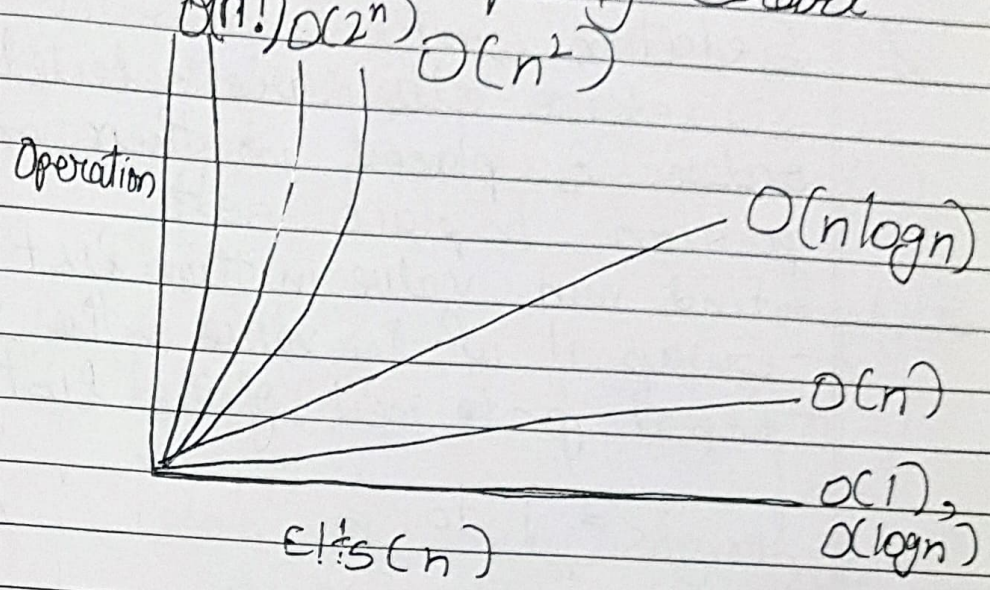
At a glance

Name (Sort)	Best Case	Avg Case	Worst Case	Memory	Method	Stable
Quick	$n \log n$	$n \log n$	n^2	$\log n$	Partition	N
Merge	$n \log n$	$n \log n$	$n \log n$	n	Merge	Y
Heap	$n \log n$	$n \log n$	$n \log n$	1	Selection	N
Insertion	n	n^2	n^2	1	Insertion	Y
Selection	n^2	n^2	n^2	1	Selection	N
Bubble	n	n^2	n^2	1	Exchange	Y
Shell	$n \log n$	$n^{4/3}$	$n^{3/2}$	1	Insertion	N
Tree	$n \log n$	$n \log n$	$n \log n$	n	Insertion	Y
Cycle	n^2	n^2	1.2^2	1	Selection	N
TimSort ↑ Used by Python	n	$n \log n$	$n \log n$	n	Merge+ Insertion	Y

TimSort is hybrid of merge + insertion sort

PAGE NO.
DATE / /

Big O Complexity Chart



1. Bubble Sort

Simplest algo. Repeatedly swaps adjacent elts if they are in wrong order.

Not suitable for large datasets.

```
for(i = 0; i < n-1; i++)
    for(j = 0; j < n-i-1; j++)
        if (arr[j] > arr[j+1])
            swap(arr[j], arr[j+1])
```

Useful for almost sorted datasets.

PAGE No.
DATE / /

2. Selection Sort

Successive elts are selected in order & placed in their propere position. In place sort.

- Find min. value in the list
- Swap it w/ the value in the 1st posn
- Repeat for rest of the list

	Step cost
for i = 1 to n	; n+1
j = i	;
for k = i+1 to n	; n(n+1)/2
if (A[k] < A[j]) then	; n(n+1)/2 - 1
j = k	; n(n+1)/2 - 1
~~swap (A[i], A[j])~~	;
swap (A[i], A[j])	; n

Use when list is small or memory is limited.

PAGE No.
DATE / /

3. Insertion Sort :- Places an unsorted element and place in its right position. (Like cards)

Code	Analysis
for $j = 2$ to n	n
key = A[j]	$n-1$
$i = j-1$	$n-1$
while $i > 0$ and $A[i] > key$	$\sum_{j=2}^{n} t_i$
$A[i+1] = A[i]$	$n-1$
$i = i-1$	$n-1$
$A[i+1] = key$	$n-1$

When list is sorted (nearly sorted.
Space complexity of O(1)

3. Merge Sort (Divide & Conquer)
Divides input array in half repeatedly till only one elt left, and merges the two sorted halves.

```
Merge_Sort (A, low, high)
  if (low < high)
      mid = (low + high)/2
      Merge_Sort (A, low, mid)
      Merge_Sort (A, mid+1, high)
      Merge (A, low, mid, high)
```

PAGE No.
DATE / /

```
Merge (A, low, mid, high)
  h = low, i = low, j = mid+1
  Auxiliary array B[]
  while (h ≤ mid & j ≤ high)
      if (A[h] ≤ A[j])
          B[i] = A[h]
          h = h+1
      else
          B[i] = A[j]
          j = j+1
      i = i+1
  if (h > mid)
      for k = j to high
          B[i] = A[k]
          i = i+1
  else
      for k = h to mid
          B[i] = A[k]
          i = i+1
  for k = low to high
      A[k] = B[k]
```

Uses:- When data structure doesn't allow random access. (linked list ex)
Used in external sorting.
Sequential access

5 Quicle Sort (Divide & Conquer)
Fastest known Sorting algo
Aka Partition Exchange Sort.
No merge step involved.

```
Quicksort (A, lb, ub)
  if (lb < ub)
      pivot = Partition (A, lb, ub)
      Quicksort (A, lb, pivot-1)
      Quicksort (A, pivot+1, ub)

Partition (A, lb, ub)
   x = A[lb]
   up = ub
   down = lb
   while (down < up)
       while (A[down] <= x & down < ub)
           down = down+1
       while( A[up] > x)
           up = up - 1
       if (down < up)
           Swap( A[down, A[up])
   A[lb] = A[up]
   A[up] = x
   Return up
```

x = 40

Ex:- 40 | 20 | 10 | 80 | 60

down ↓ up ↓

40 | 20 | 10 | 80 | 60

down up

40 | 20 | 10 | 80 | 60

down up

40 | 20 | 10 | 80 | 60

up up

After while partition :-

40 | 20 | 10 | 80 | 60

up down

A[lb] = A[up], A[up] = x

10	20	40	80	60

Partition will returns 3 as o/p

When to use :-

More effective when data fits in memory. It is an in-place sort.

Worst performance on sorted array.

Solution :- Randomized quicksort.

- Choose pivot randomly.
- Make random shuffling.

5. Searching

1. Linear Search

```
Linear-search (se3, A)
    for (i = i to n)
        if (A[i] = x)
            return i
    return null
```

Time Complexity :- O(n) [Avg, Worst]
Best :- O(1)

2. Binary Search (Divide & Conquer)

```
Bin_search (A, n, target)
    left = 1
    right = n
    while (left ≤ right)
        mid = floor((left + right)/2)
        if (target = A[mid])
            return mid
        else if (x > A[mid])
            low = mid + 1
        else
            high = mid - 1
```

Time Complexity Avg/Worse = O(logn)

PAGE No.
DATE / /

5. Approximating Time Complexity

⇒ Iterative :- Count number of times instructions are executed

Ex:- for i = 1 to n
  for j = 1 to n
    for k = 1 to n
      ...

Time Complexity = $O(n^3)$

for i = 1 to n n
...
for i = 1 to n n

Total = n + n = 2n
Time Complexity = O(n)

⇒ Recursive Algorithms :-
Three methods :-
(i) Recursion Relation / Substitution
Ex:- A(n)
  if (n>1)
    return A(n/2) + A(n/2)

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

If return was n-1, it would be
$$T(n) = c + 2T(n-1)$$

* Some Common Recurrences

Recurrence	Solution
$T(n)=T(n/2)+d$	$T(n)=O(\log_2 n)$
$T(n/2)+n$	$O(n)$
$2T(n/2)+d$	$O(n)$
$2T(n/2)+n$	$O(n\log n)$
$T(n-1)+d$	$O(n)$

(ii) Master's Theorem :- For divide & conquer algo;
if problem of size n is divided into 'a' problems of size n/b
Let cost of each stage to divide & conquer be $f(n)$
Then, according to Master's theorem
If $T(n) = aT(n/b) + f(n)$
$a \geq 1$ $b \geq 1$ then,

$$T(n) = \begin{cases} \Theta(n^{\log_b a}) & f(n) = O(n^{\log_b a - \epsilon}) \\ \Theta(n^{\log_b a} \log n) & f(n) = \Theta(n^{\log_b a}) \\ \Theta(f(n)) & f(n) = \Omega(n^{\log_b a + \epsilon}) \\ & \text{and } af(n/b) < cf(n) \end{cases}$$

$\epsilon > 0$, $c < 1$

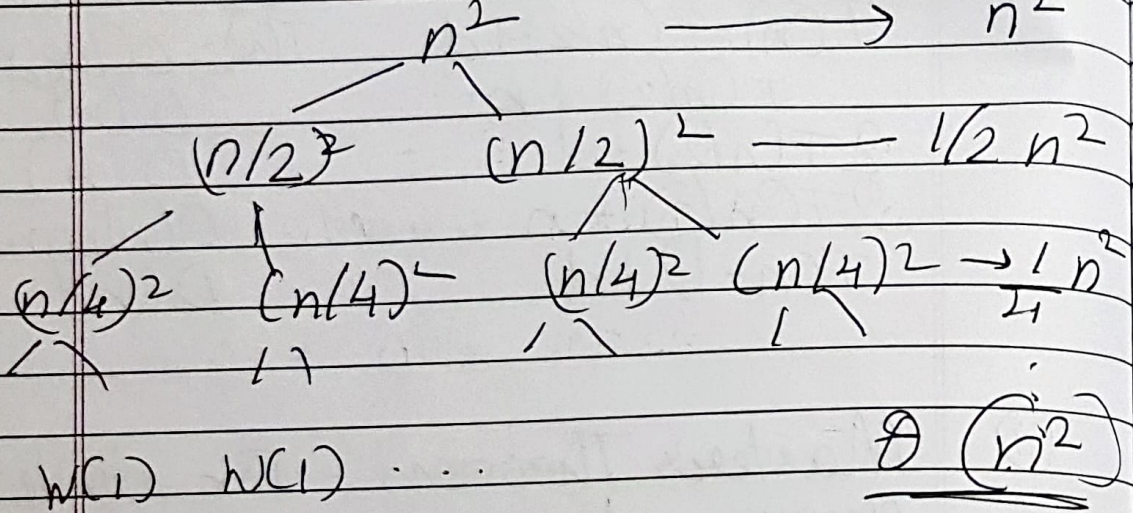
(iii) Recurrence Tree

Node - Cost at various levels

Sum up cost of all levels.

Ex :-

$W(n) = 2W(n/2) + n^2$



$\Theta(n^2)$

PAGE No.
DATE / /

5. DIVIDE & CONQUER

Divide problem to subproblems
Solve Subproblem - Conquer
Combine to get solution

⇒ Previous examples :-
① - Merge Sort
② - Quick Sort
③ - Binary Search

④ Strassen's Matrix Multiplication

Standard Matrix Multiplication :- $O(n^3)$
Strassens - " - : $O(n^{\log 7})$

where n is from $n \times n$ matrix

```
MatMul(A, B, C, n)
  if n = 1
      C = C + A * B
  else
      MatMul(A, B, C, n/4)
      MatMul(A, (B+n/4), (C+n/4), n/4)
      MatMul(A+2x(n/4), B, C+2xn/4, n/4)
      MatMul(A+2x(n/4), B+n/4, C+3xn/4, n/4)
      MatMul(A+(n/4), B+2x(n/4), C, n/4)
      MatMul(A+(n/4), B+3x(n/4), C+(n/4), n/4)
      MatMul(A+3x(n/4), B+2x(n/4), C+2x(n/4), n/4)
      MatMul(A+3x(n/4), B+3x(n/4), C+3x(n/4), n/4)
```

There are 18 $n/2 \times n/2$ matrix add^n / sub^n $O(n^2)$
7 $n/2 \times n/2$ multiplications
$T(n) = 7T(n/2) + O(n^2)$
$= O(n^{2.81})$
$T(n) = O(n^{\log 7})$

(5) Max & Min in a list
- Divide problems into $n/2$ subproblems & find min and max recursively.
Time Complexity = $O(n)$

```
MaxMin(i, y, max, min)
if i = y
   max = min = A[i]
else if (i = y - 1)
     if (A[i] < A[y])
        min = A[i]; max = A[y]
     else
        min = A[y]; max = A[i]
else
   mid = floor((i+y)/2)
   MaxMin(i, mid, max, min)
   MaxMin(mid+1, y, max1, min1)
   if (max < max1)
        max = max1
   if (min > min1)
        min = min1
```

PAGE No.
DATE / /

6. DYNAMIC PROGRAMMING

Method used when problem to be solved is a sequence of decisions. Final solution by combining solution to sequence of subproblems.

Steps :-
- Characterize structure of optimal sol^n
- Recursively define value of optimal sol^n
- Compute value of optimal sol^n, typically in a bottom-up fashion.
- Construct optimal sol^n from computed info.

(1) All Pairs Shortest Path - Floyd-Warshall Algorithm

A graph G(V, E), find shortest path between every pair of vertices

w(i,j) = 0 if i=j
w(i,j) = ∞ if no connecting edge
w(i,j) = wt of edge.

No of iterations = no of vertices

Algorithm
for
$n \leftarrow$ rows(W) ← No of vertices
$D(0) \leftarrow W$ # Input
for $k \leftarrow 1$ to n
  for $i \leftarrow 1$ to n
    for $j \leftarrow 1$ to n
      $D_{ij}^{k} \leftarrow \min(D_{ij}^{k-1}, D_{ik}^{k-1} + D_{kj}^{k-1})$

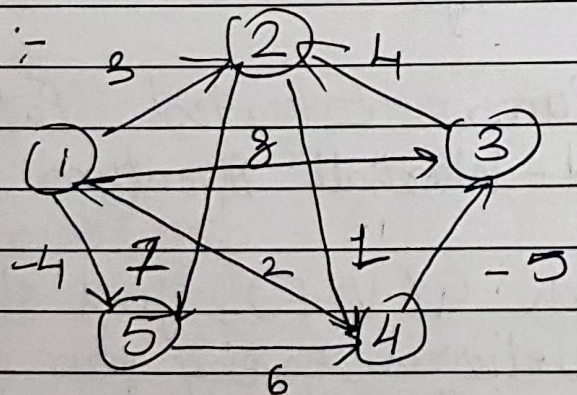
return D

Ex:-

$D_0 = W_0$

Time complexity $= O(n^3)$

Ex:-



Input- $D_0 = W_0$

	1	2	3	4	5
1	0	3	8	∞	-4
2	∞	0	∞	1	7
3	∞	4	0	∞	∞
4	2	∞	-5	0	∞
5	∞	∞	∞	6	0

PAGE No.
DATE 1 1

Iteration 1 :- D_1 → Distance from to shortest all nodes via node 1

$$D^{(1)} = \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \end{array} \begin{bmatrix} 1 & 2 & 3 & 4 & 5 \\ 0 & 3 & 8 & \infty & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & \infty & \infty \\ 2 & 5 & -5 & 0 & -2 \\ \infty & \infty & \infty & 6 & 0 \end{bmatrix}$$

Expl:-

i = 4, j = 2 To go from node 4 to 2 via 1, total cost
= min(∞, 2 + 3 = 5) = 5

Similarly i = 4, j = 3
Wt = min(−5, 2+8=10) = −5

Iteration k = 2

$$D^{(2)} = \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \end{array} \begin{bmatrix} 1 & 2 & 3 & 4 & 5 \\ 0 & 3 & 8 & 4 & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & 5 & 11 \\ 2 & 5 & -5 & 0 & -2 \\ \infty & \infty & \infty & 6 & 0 \end{bmatrix}$$

Algorithm

for n ← rows(W) ← No of vertices

D(0) ← W # Input

for k ← 1 to n

  for i ← 1 to n

    for j ← 1 to n

      $D_{ij}^{k} \leftarrow \min(D_{ij}^{k-1}, D_{ik}^{k-1} + D_{kj}^{k-1})$

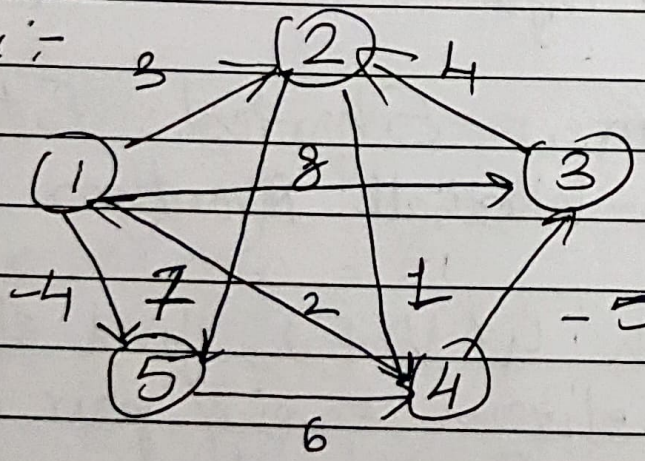
return D

Ex:-

$D_0 = W_0$

Time complexity = $O(n^3)$

Ex:-



Input- $D_0 = W_0$

	1	2	3	4	5
1	0	3	8	∞	-4
2	∞	0	∞	1	7
3	∞	4	0	∞	∞
4	2	∞	-5	0	∞
5	∞	∞	∞	6	0

Shortest

Iteration 1 :- D_1 → Distance from to all nodes via node 1

$D^{(1)}$ =

	1	2	3	4	5
1	0	3	8	∞	-4
2	∞	0	∞	1	7
3	∞	4	0	∞	∞
4	2	5	-5	0	-2
5	∞	∞	∞	6	0

Expl:-

i = 4, j = 2 To go from node 4 to 2 via 1, total cost
= min(∞, 2 + 3 = 5) = 5

Wardy i = 4, j = 3
Wt = min(-5, 2+8=10) = -5

Iteration k=2

$D^{(2)}$ =

	1	2	3	4	5
1	0	3	8	4	-4
2	∞	0	∞	1	7
3	∞	4	0	5	11
4	2	5	-5	0	-2
5	∞	∞	∞	6	0

k = 3

$D_3 =$

	1	2	3	4	5
1	0	3	8	4	-4
2	∞	0	∞	1	7
3	∞	4	0	5	11
4	2	-1	-5	0	-2
5	∞	∞	∞	6	0

k = 4

$D_4 =$

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

k = 5

Output = D_5

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

(2) Longest Common Subsequence

Substring v/s subsequence

Substring :- Substring of a string S is another string S' that occur in S and all the letters are contiguous in S

Eg S :- Helloworld

S' :- Woold.

Subsequence :- Subsequence of string S is another string S' that occurs in S and all letters need not be contiguous

Eg S :- Helloworld ; S' = Herd

Longest Common Subsequence :-

Given two strings A of length x and B of length y, find the longest common subsequence between both strings from left to right

Brute Force Method :-

S1 :- Find all subsequences of A, (2^x)

S2 :- Find each subsequence find whether it is subsequence of B, $(yx2^x)$

S3 :- Choose the longest common subsequence

Time complexity: $yx2^x$ or $O(n2^m)$

Dynamic Programming Method :-

Two Strings:

$X = \{ x_1, x_2, x_3 \dots x_m \}$

$Y = \{ y_1, y_2, y_3 \dots y_n \}$

First compare x_m and y_n, if matched, find subsequence in remaining string & append x_m.

If $x_m \neq y_n$,

Remove x_m from X & find LCS from x_1 to x_{m-1} and y_1 to y_n

Remove y_n from Y & find LCS from x_1 to x_m and y_1 to y_{n-1}

$c[i,y]$ = length of LCS of x_i & y_y

$$c[i,y] = \begin{cases} 0 & \text{if } i=0 \text{ or } y=0 \\ c[i-1, y-1]+1 & \text{if } i,y>0 \text{ \& } x_i = y_i \\ \max(c[i,y-1], c[i-1,y]) & \text{if } i,y>0 \text{ \& } x_i \neq y_i \end{cases}$$

```
LCS-Length(X, Y)
m ← length(X)
n ← length(Y)
for i ← 1 to m
    do c[i,0] ← 0
for y ← 0 to n
    do c[0,y] ← 0
for i ← 1 to m
    for J ← 1 to n
        if xi = yJ
            c[i,y] ← c[i-1, y-1] + 1
        else if b[i,y] ← "↖"
        else if c[i-1, y] >= c[i, y-1]
            c[i,y] ← c[i-1, y]
            b[i,y] ← "↑"
        else
            c[i,y] ← c[i, y-1]
            b[i,y] ← "←"
return c & b
```

Initial call i, j is x.length, y.len

```
PRINT_LCS (b, X, i, y)
if i=0 & y=0
    return
if b[i,y] = "↖"
    PRINT_LCS (b, X, i-1, y-1)
    print xi
else if b[i,y] = "↑"
    PRINT_LCS(b, X, i-1, y)
else PRINT_LCS(b, X, i, y-1)
```

PAGE No.
DATE / /

Total Runtime = O(mn)

Ex:- LCS of ACBCF & ABCDAF

			A	C	B	C	F
	A	0	0	0	0	0	0
A	B	0	1↖	1←	1←	1←	1←
B	C	0	1↑	1↑	2↖	2←	2←
C	D	0	1↑	2↖	2↑	3↖	3←
D	A	0	1↑	2↑	2↑	3↑	3↑
A	F	0	1↖	2↑	2↑	3↑	3↑
F		0	1↑	2↑	2↑	3↑	4↖

Follow as per arrows, print on (↖)

F C B A → Reverse ⇒ ABCF

PAGE NO.
DATE / /

(3) Single Source Shortest Path

For graph $G = (V, E)$, to find the shortest path from given source vertex $s \in V$ to every other vertex $v \in V$

→ Bellman-Ford Algorithm

Given a weighted, directed graph $G = (V, E)$ with source s, the algo returns a boolean value indicating whether or not there is a -ve weight cycle reachable from the source.

```
Bellman-Ford (G, w, s)
INITIALIZE-SINGLE-SOURCE (G, s)
for i ← 1 to |V[G]| - 1
    for each edge (u, v) ∈ E[G]
        RELAX (u, v, w)
for each edge (u, v) ∈ E[G]
        d[v] > d[u] + w(u, v)
            return FALSE
return TRUE
```

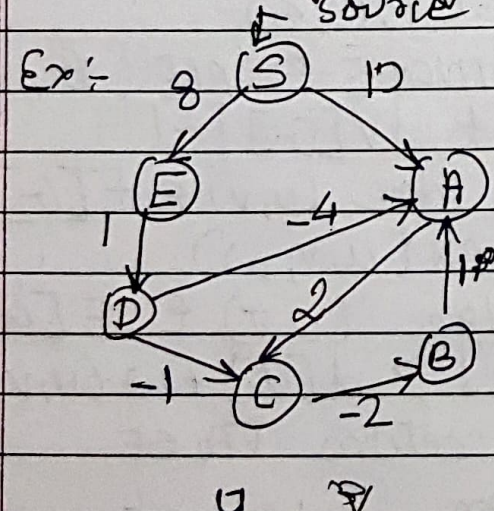
PAGE No.
DATE / /

INITIALIZE-SINGLE-SOURCE (G, S)
for each vertex $v \in V[G]$
do $d[v] \leftarrow \infty$
$\pi[v] \leftarrow NIL$
$d[s] \leftarrow 0$

RELAX (u, v, w)
if $d[v] > d[u] + w(u, v)$
$d[v] \leftarrow d[u] + w(u, v)$
$\pi[v] \leftarrow u$

Time complexity :- For V vertices & E ~~edge~~ edge $O(VE)$

Ex :-



u	v	w
S	A	10
S	E	8
A	C	2
E	D	1
B	A	1
D	A	-4
D	C	-1
C	B	-2

Initialization

	S	A	B	C	D	E
d	0	∞	∞	∞	∞	∞
π	N	N	N	N	N	N

PAGE No.
DATE / /

I st Iteration. (Go through u, v, w list)

	S	A	B	C	D	E
d	0	10 5	6	8 9 10 12 12	8+1=9	8
Π	N	S D	C	A D	E	S

Keep looping through u, v, w list till n-1 or two iterations give same result.
While creating u, v, w list always start with S & go to next connected vertex.

II

	S	A	B	C	D	E
d	0	5	5	7	9	8
Π	N	D	C	A	E	S

III

	S	A	B	C	D	E
d	0	5	5	7	9	8
Π	N	D	C	A	E	S

Output	
S → A	S $\xrightarrow{8}$ E $\xrightarrow{1}$ D $\xrightarrow{-4}$ A
S → B	S $\xrightarrow{8}$ E $\xrightarrow{1}$ D $\xrightarrow{-4}$ A $\xrightarrow{2}$ C $\xrightarrow{-1}$ B
S → C	S → E → D → A → C
S → D	S → E → D
S → E	S → E

PAGE No.
DATE / /

(4) Multistage graph
Can use Single source shortest path.

(5) 0/1 Knapsack Problem

```
DP_Knapsack (V, p, w, M, n)
for i ← 1 to n
    V[i, 0] ← 0
for y ← 0 to M
    V[0, y] ← 0
for i ← 1 to n
    for y ← 1 to M
        if w[i] ≤ y
            V[i, y] ← max{V[i-1, y],
                          p[i] + V[i-1, y - w[i]]}
        else
            V[i, y] ← V[i-1, y]
```

Complexity = O(n × M)
n ← No. of elts
M ← Capacity of knapsack
wi - Weight of item
Pi - Price of items

Maximize price within given capacity

Ex :- $M = 7$, $n = 4$

$W = \{1, 3, 4, 5\}$, $P = \{1, 4, 5, 7\}$

Item		1	2	3	4	5	6	7
	0	0	0	0	0	0	0	0
$W_1 = 1, P_1 = 1$	0	1	1	1	1	1	1	1
$W_2 = 3, P_2 = 4$	0	1	1	[4]	5	5	5	5
$W_3 = 4, P_3 = 5$	0	1	1	4	5	6	6	[9]
$W_4 = 5, P_4 = 7$	0	1	1	4	5	7	8	[9]

Trace_Knapsack (W, V, M)

```
SW ← ϕ
SP ← ϕ
i ← n      j ← M
while (j > 0)
    if (V[i,j] == V[i-1,j])
        i ← i - 1
    else
        SW ← SW + W[i]
        SP ← SP + P[i]
        j ← j - W[i]
        i ← i - 1
```

For previous problem

SW	SP
—	—
4	5
3	4

$j = 7 - 4 = 3$

PAGE No.
DATE / /

7. Greedy Method

Make choice that looks best at that moment.

Sometimes works, sometimes doesn't

Properties :-

- Globally optimal soln is obtained from locally optimal soln
- Best choice in current situation selected.
- Optimal Substructure :- Problem has optimal substructure if an optimal soln to the problems is composed of optimal solns to subproblems.

(1) Single Source Shortest path
- Dijkstra's algorithm

All weights must be non-negative

S :- Set of vertices whose final shortest path weights have already been determined.

Q :- A min priority queue keyed by their distance values.

Repeatedly select vertex $u \in V - S$ (kept in Q, with minimum shortest

path and add u to S, & relax all edges leaving u.

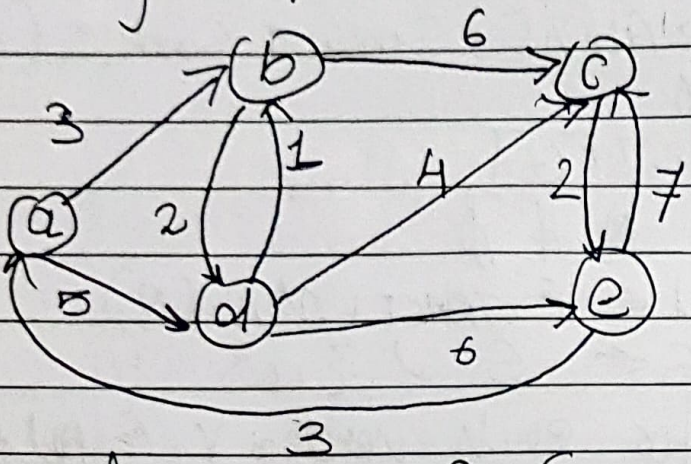
```
Dijkstra(G, w, s)
INITIALIZE_SINGLE_SOURCE(G, s)
S ← ∅
Q ← V[G]
while Q ≠ ∅
    u ← EXTRACT-MIN(Q)
    S ← S ∪ {u}
    for each vertex v ∈ Adj[u]
        RELAX(u, v, w)

INITIALIZE_SINGLE_SOURCE(G, s)
for each vertex v ∈ V[G]
    d[v] ← ∞
    π[v] ← NIL
d[s] ← 0

RELAX(u, v, w)
if d[v] > d[u] + w(u, v)
    d[v] ← d[u] + w(u, v)
    π[v] ← u
```

Time complexity $O(V^2)$

Priority Queue Impl.	Time Complexity
Array	$O(V^2)$
Priority Queue	$O(V \log V)$
Binary Heap	$O(E \log V)$



$S = \phi$ $\quad Q = \{a, b, c, d, e\}$

	a	b	c	d	e
d	0	∞	∞	∞	∞
π	N	N	N	N	N

$u = a$ $\quad Q = \{b, c, d, e\}$ $\quad S = \{a\}$ $\quad V = b, d$

	a	b	c	d	e
d	0	3	∞	5	∞
π	N	a	N	a	N

$u = b$ $\quad Q = \{c, d, e\}$ $\quad S = \{a, b\}$ $\quad V = \{c, d\}$

	a	b	c	d	e
d	0	3	9	5	∞
π	N	a	b	a	N

$u = d$; $Q = \{c, e\}$, $S = \{a, b, d\}$; $V = \{b, c, e\}$

	a	b	c	d	e
d	0	3	9	5	11
π	N	a	b	a	d

$u = c$, $Q = \{e\}$, $S = \{a, b, c, d\}$; $V = \{e\}$

	a	b	c	d	e
d	0	3	9	5	11
π	N	a	b	a	d

$u = e$, $Q = \phi$, $S = \{a, b, c, d, e\}$

PAGE No.
DATE / /

② Knapsack (Fractional)

- Sort items in decreasing c_i / w_i
- Add items till no more items or sack is full (exceeds W)
- If sack is not full, fill with fraction of next unselected item.

```
Knapsack(C, W, M, X, n)
for i ← 1 to n
    X[i] ← 0
RC ← M
for i ← 1 to n
    if W[i] > RC
        break
    X[i] ← 1
    RC ← RC - W[i]

if i ≤ n then
    X[i] ← RC / W[i]
```

M - Knapsack capacity
W[i] - weight of item i
X[i] - 0/1 array telling if item i is selected or not.

Time Complexity :- O(nlogn)

PAGE NO.
DATE / /

(3) Spanning Tree

- For an connected undirected graph, spanning tree of the graph is subgraph which is a tree and connects all vertices.

for $G(V, E)$, $T = (V, E')$ is a spanning tree iff T is a tree.

Minimum Spanning Tree (MST) is a spanning tree with weight less than or equal to every other spanning tree of that graph.

Algorithms

3.1 Kruskal's Algorithm

- Set A is ~~post~~
- Edge A is added to A if it is always a least weighted edge in the graph that connects two distinct components.

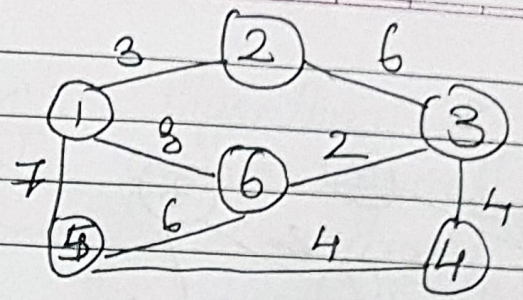
MST_Kruskal (G, w)
A ← ϕ
for each vertex v ∈ V[G]
    Make_Set (v)
Sort edges of E by increasing wt w
for each edge (u,v) ∈ E
    if Find_Set(u) ≠ Find_Set(v)
        A ← A ∪ {(u,v)}
        Union (u,v)
Return A

Make_Set (x) :- Creates set of single elt x

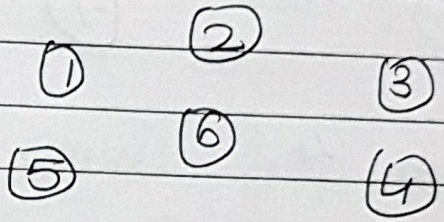
Find_Set (x) - determine if two vertices u & v belong to same tree by checking if Find_Set (u) ≠ Find_Set (v)

Union (x,y) - Unites the sets that contain x & y say Sx & Sy into a new set that is union of the two sets.

Analysis :- Avg Case :- O(E log E)



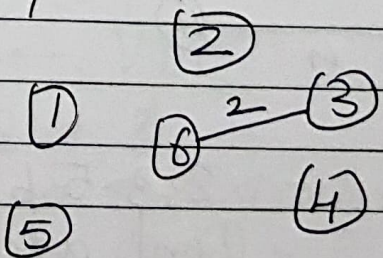
Disjoint Forest



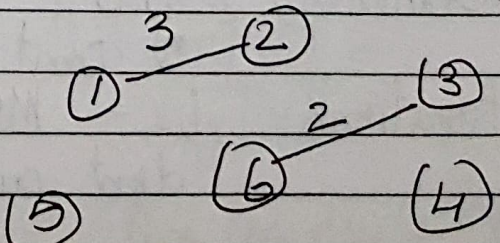
Sort Edges

Edge	wt
(3,6)	2
1,2	3
3,4	4
4,5	4
2,3	6
5,6	6
1,5	7
1,6	8

Iteration 1

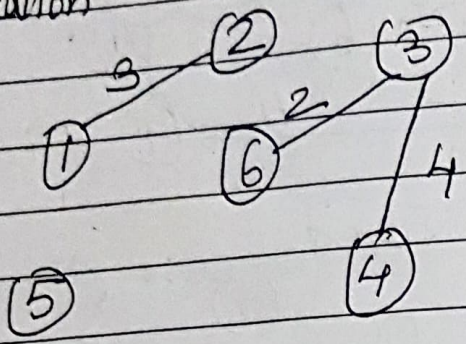


Iteration 2

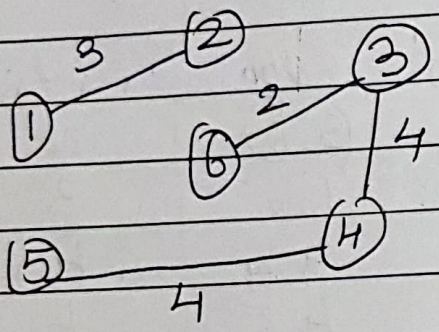


DATE / /

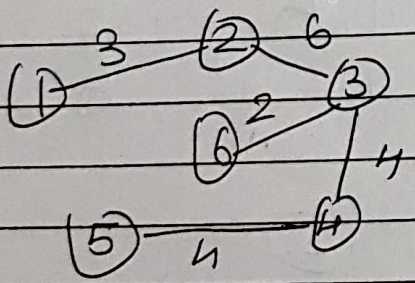
Iteration 3



Iteration 4



Iteration 5 :- (2,3) forms cycle? No, add



Iteration 6 :- 2,3 - 5,6 makes cycle? Yes
Dont add

Iteration 7 :- 1,5 - Ye makes cycle? Yes
Dont add

Iteration 8 1,6 - Makes cycle, ? Yes
dont add.

3.2 Prim's Algorithm

- MST grows "naturally" starting from an arbitrary root.
- Has the property that the edges in the set always form a single tree

Logic

- Tree starts from arbitrary root vertex r.
- Grow the tree until it spans all vertices in set V

Data Structure

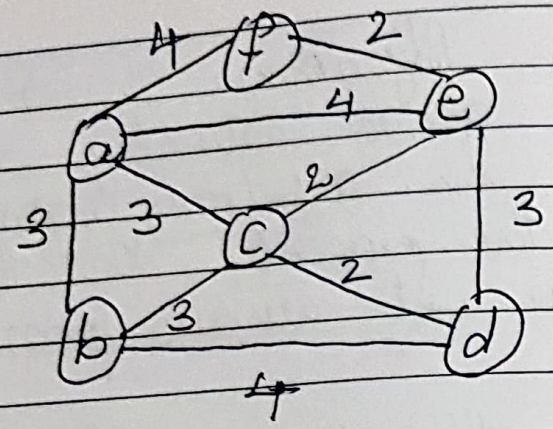
- A min. priority queue keyed by edge values.

```
                          ↓ - root vertex
MST_PRIM(G, w, r)
for each u ∈ V[G]
        key[u] ← ∞
        π[u] ← NIL
key[r] ← 0
Q      ← V[G]
while Q ≠ ∅
    do u ← Extract-Min(Q)
       for each v ∈ Adj[u]
           if v ∈ Q and w(u,v) < key[v]
                π[v] ← u
                key[v] ← w(u,v)
```

Analysis :- Avg complexity $O(E \log V)$

DATE / /

Ex



	a	b	c	d	e	f
key	0	∞	∞	∞	∞	∞
π	N	N	N	N	N	N

① $u = a$ $Q = \{b, c, d, e, f\}$

$v = b, c, e, f$

	a	b	c	d	e	f
key	0	3	3	∞	4	4
π	N	a	a	N	a	a

② $u = b$ $Q = \{c, d, e, f\}$ $v = c, d, e$ (nothing)

	a	b	c	d	e	f
key	0	3	3	4	4	4
π	N	a	a	b	a	a

③ $u = c$ $Q = \{d, e, f\}$ $v = a, b, e, d$

	a	b	c	d	e	f
key	0	3	3	2	2	4
π	N	a	a	c	c	a

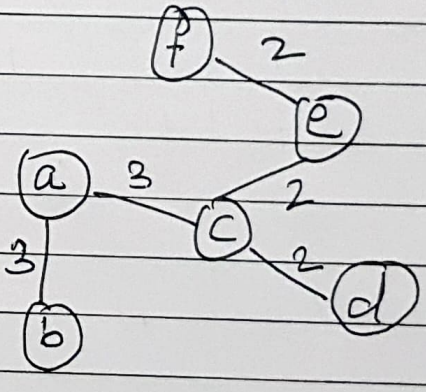
④ $u = d \quad Q = \{e, f\} \quad V = b, e$

	a	b	c	d	e	f
key	∞	3	3	2	2	4
π	N	a	a	c	c	a

⑤ $u = e \quad Q = f \quad V = a, f$

	a	b	c	d	e	f
key	∞	3	3	2	2	2
π	N	a	a	c	c	e

⑥ $u = f \quad Q = \phi \quad V = a, e$



PAGE No.
DATE / /

(4) Job Sequencing using deadlines
- List of n jobs with each job i has a deadline $d_i \geq 0$ and profit $p_i > 0$ earned only if job completed

- Find a subset J s.t. each job of this subset can be completed within deadline; maximizing profit

① Sort p_i into decreasing order
② Add next job i to the soln set J if i can be completed by deadline.
- Assign i to time slot $[r-1, r]$, where r is $1 \leq r \leq d_i$ and slot $[r-1, r]$ is free.
* Schedule in latest possible free slot for optimal soln
③ Repeat step 2 till slot full or jobs examined.

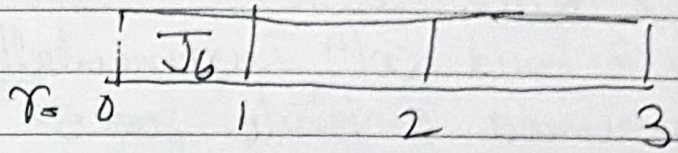
Complexity :- $O(n^2)$

$p = \{p_1, p_2, p_3 \ldots p_7\} = [5, 4, 8, 7, 6, 9, 3]$
$d = \{d_1, d_2, d_3 \ldots, d_7\} = [3, 2, 1, 3, 2, 1, 2]$

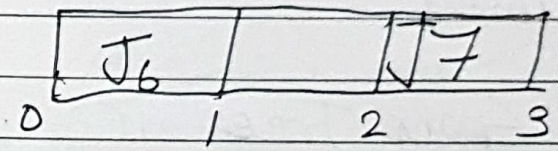
Sort in descending order

i	6	3	4	5	1	2	7
p	9	8	7	6	5	4	3
d	1	1	3	2	3	2	2

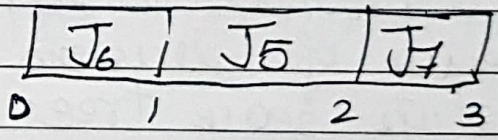
Job scheduling



Early loop:-
J_3 - Rejected
J_4 - Accepted



J_5 - Accept



→ Optional :- ① Optimal Storage on tapes
② Huffmann Coding

PAGE No.
DATE / /

8. BACKTRACKING

Try to build soln incrementally
Refinement of brute force.
Systematically search set of feasible soln "solution space" to solve given problem.
From n solutions $(x_1, x_2 \dots x_n)$, x_i is chosen.

⇒ State Space Tree

State Space :- Set of states a system can exist in
Soln to problem is obtained by sequence of decisions resulting in state space tree.
Root :- State before any decision

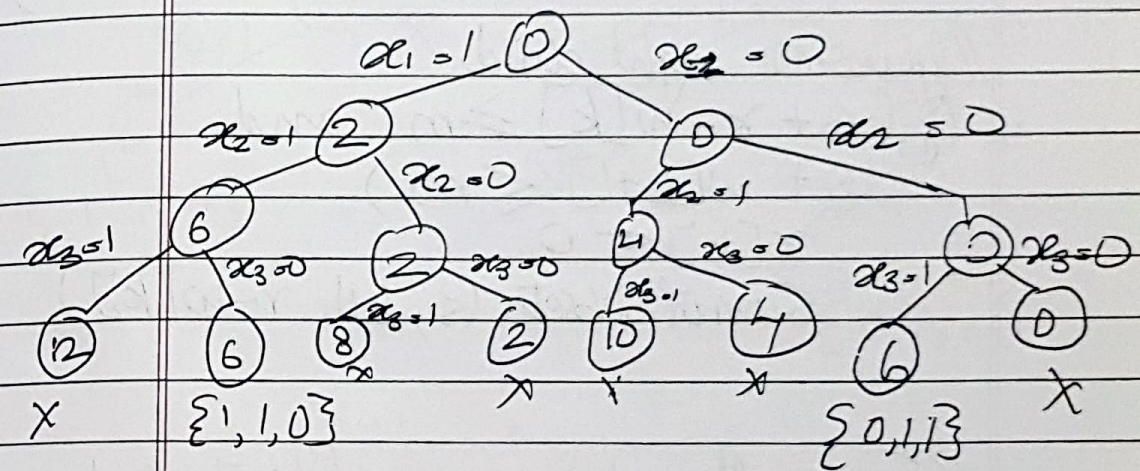
① Sum of Subsets

Given n positive integers find subsets of $w_1 \dots w_n$ that sum to S.
Soln vector $\{x_1, x_2 \dots x_n\}$ x_i is 1 if included, else 0.

Binary State Space Tree :-
Node at depth 1 is for item1 (yes/no) at depth 2 for item2, etc.
left ieft branch include w_i;
right branch excludes w_i.

Nodes contain sums of weights so far
At every stage, we check whether node is valid or not. If invalid, we backtrack to parent.

Ex
$w_1 = 2,\ w_2 = 4,\ w_3 = 6$ & $S = 6$ ← o/p



Initial call :- (0, 1, r)
s :- current sum; initialised to 0
k :- current elt to consider
r :- sum of all elts w[1 : n]

PAGE No.
DATE / /

```
SumofSubsets(s, k, r)
// Generate left child
x[k] ← 1
if (s+w[k] = m)
    write x[1:k] // one subset found
else if (s+w[k]+w[k+1] ≤ m)
    SumofSubsets(s+w[k], k+1, r-w[k])

// Generate right child
if (s+r-w[k] ≥ m and
    s+w[k+1] ≤ m)
    x[k] ← 0
    SumofSubsets(s, k+1, r-w[k])
```

// Find all subsets of w[1:n] that sum to m

$$s = \sum_{j=1}^{k-1} w[j] * x[j]$$

$$r = \sum_{j=k}^{n} w[j]$$

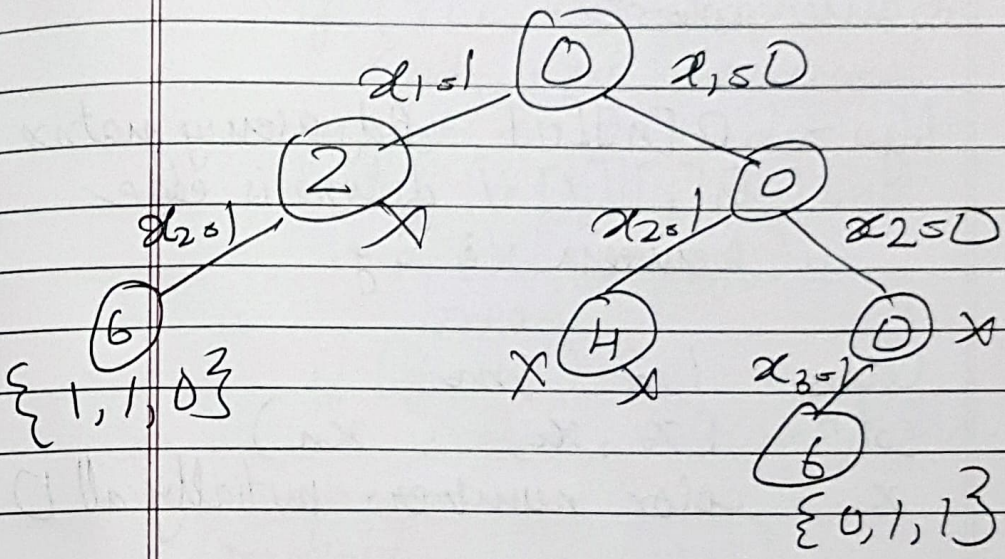
w[j] are in ascending order (sorted array)

∴ it is assumed that $w[i] \leq m$ & $\sum_{i=1}^{n} w[i] \geq m$

left &

Abv generation of right child results in a pruned tree.

$w_1 \leq 2$, $w_2 \leq 4$, $w_3 \leq 6$ $S \leq 6$



Complexity = $O(2^n)$

(2) Graph Coloring

G - Graph, m - No. of colors (positive int)

No two adjacent nodes can have same color.

Algo:- G[n][n] - Adjacency matrix
G[i][j] = 1 if there is edge between i & j.

Colors = 1, 2 ..., m

Soln $(x_1, x_2, \ldots x_n)$

x_i = color number. initially all 0

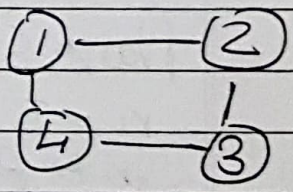
↓ Node (Initial k = 1)

```
Graph Coloring (k)
while (true)
NextValue (k)
if (x[k] = 0)
    return // No new color possible
if (k = n)
    write(x[1..n])
else
    GraphColoring (k+1)
```

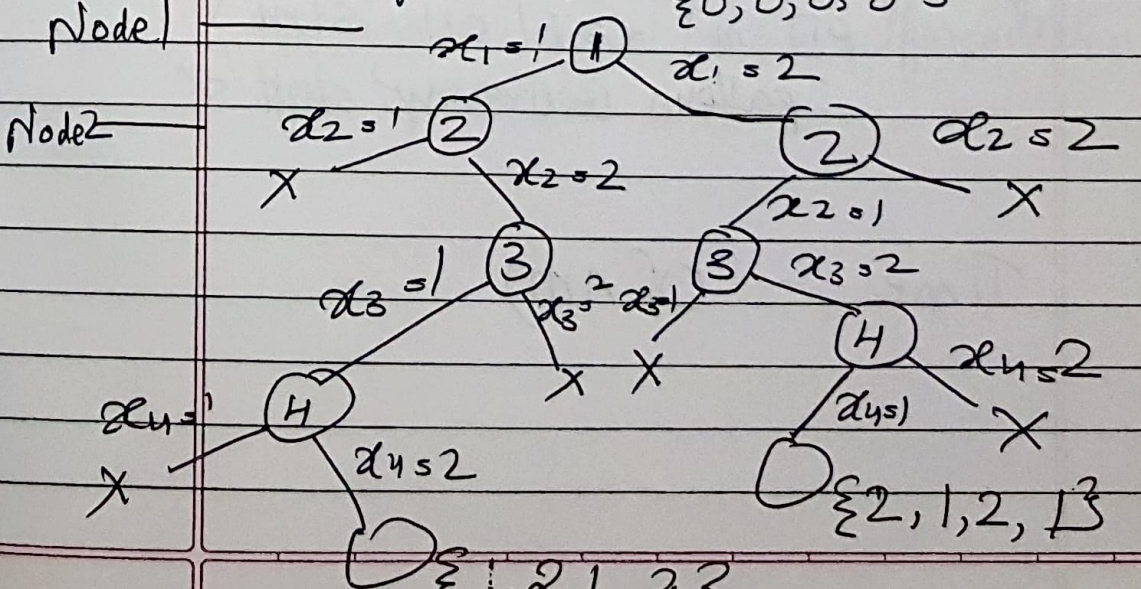
```
NextValue (k)
while true
    x[k] = (x[k]+1) mod(m+1) // Next
                  // Highest color
    if (x[k] = 0) return
      // All colors are used
    for j ← 1 to n do
        if (G[k][j] ≠ 0) and
           (x[k] = x[j])
          break
    if (j = n+1)
          return // Color found.
```

complexity = $O(mn)$

Ex: 4 nodes, 2 colors



State Space tree- {0, 0, 0, 0}



PAGE No.
DATE / /

9. STRING AND PATTERN MATCHING

→ PATTERN MATCHING
Given Text (T) & Pattern (P), pattern matching finds substring of T equal to P
Algorithms:

① Naive String Matching Algo :-
Brute force approach.
Compares first char of P wt T
if matched, increment both ptrs
else, shift pointer of T only & reset P.
Repeat till end of text or pattern found

```
Naive String Match (T, P)
n = T.length
m = P.length
for s=0 to (n-m)
    if P[1..m] == T[s+1..s+m]
        "pattern occurs w/ shift s"
```

Time :- O(nm)

(2) RABIN KARP

Generalizes well to related problems like 2-D pattern matching.

Idea

Compare a string's hash value.

Algo slides pattern one by one matching hash value of P w/ hash of current substr of T.

If value matches, check if P matches substring.

Converting substr & P to hash & comparing nos is easier.

Input :- Text T, Pattern P, Radix d (UNICODE, ASCII(256), etc), prime no q.

Radix $|\Sigma|$ - set of possible characters in a string.

For ascii, 256 characters are possible.

```
RABIN-KARP (T, P, d, q)
n = T.length
m = P.length
h = d^(m-1) mod q
p = 0
t0 = 0
for i = 1 to m          // Preprocess
    p = (dp + P[i]) mod q
    t0 = (d t0 + T[i]) mod q

for s = 0 to n-m        // Match
    if p == ts
        if P[1..m] == T[s+1 ... s+m]
            print "Pattern at shift" s
    if s < n-m
        ts+1 = (d(ts - T[s+1].h)
                + T[s+m+1]) mod q
```

Time = $O((n-m+1)m)$

Optimal Time $O(m+n)$