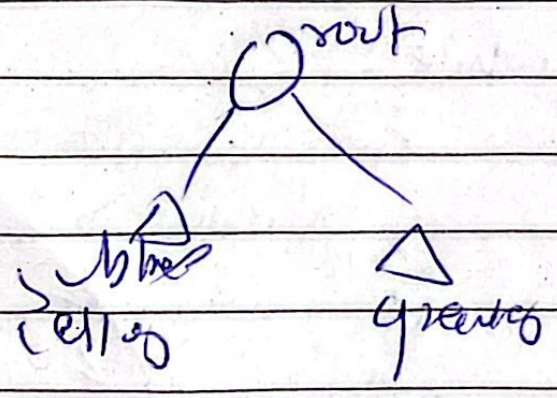
So I see that used Left & Right pointer a lot.
& he spent a lot of time, I just gave like no min & looked up answer, & everyone recommends that as well.

Binary Search Tree

search, Insert, delete
Balanced O(log n)

a binary ~~in can~~ tree, in each node, value in left is lesser & right has high



for search

$\hat{\ }$

$n/2 \to n/4$

$\log_2 n$ steps

$n/2^k = 2$

$2^k = n$

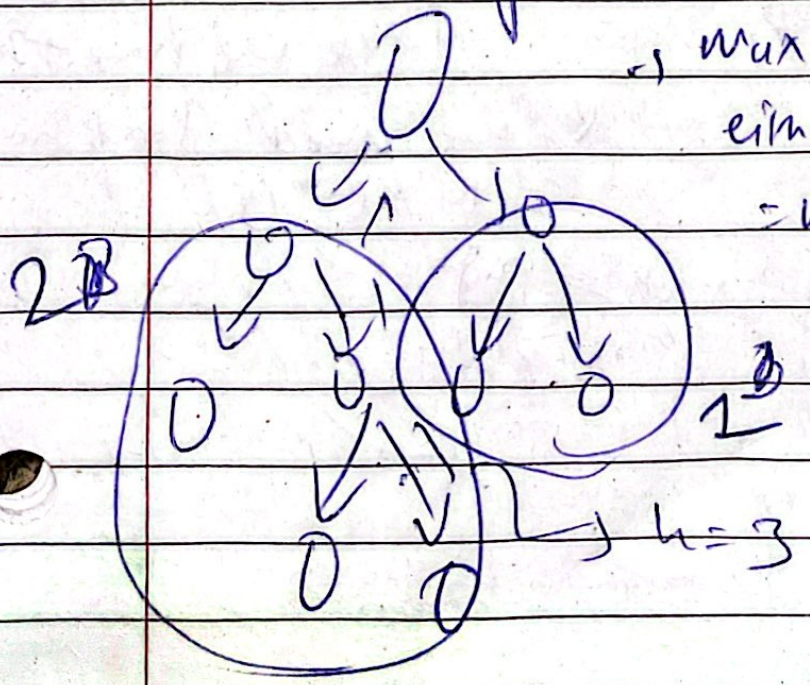
$k = \log_2 n$

find min & max

min, go to left as long as it possible

max, go to right as long as possible

find height of a binary tree



→ max height of either subtree + 1

$= \max(2,1) + 1$

Depth

- no of edges from root to that node

Find height -1

```
findHeight(root)
    left ← findHeight(root.left)
    height ← find height
                (root.right)

    return (max(left, right) +1)

    if root == null
        return -1
}
```

Leave we have +2
& we need leaf node
to be 0

Scanned with CamScanner

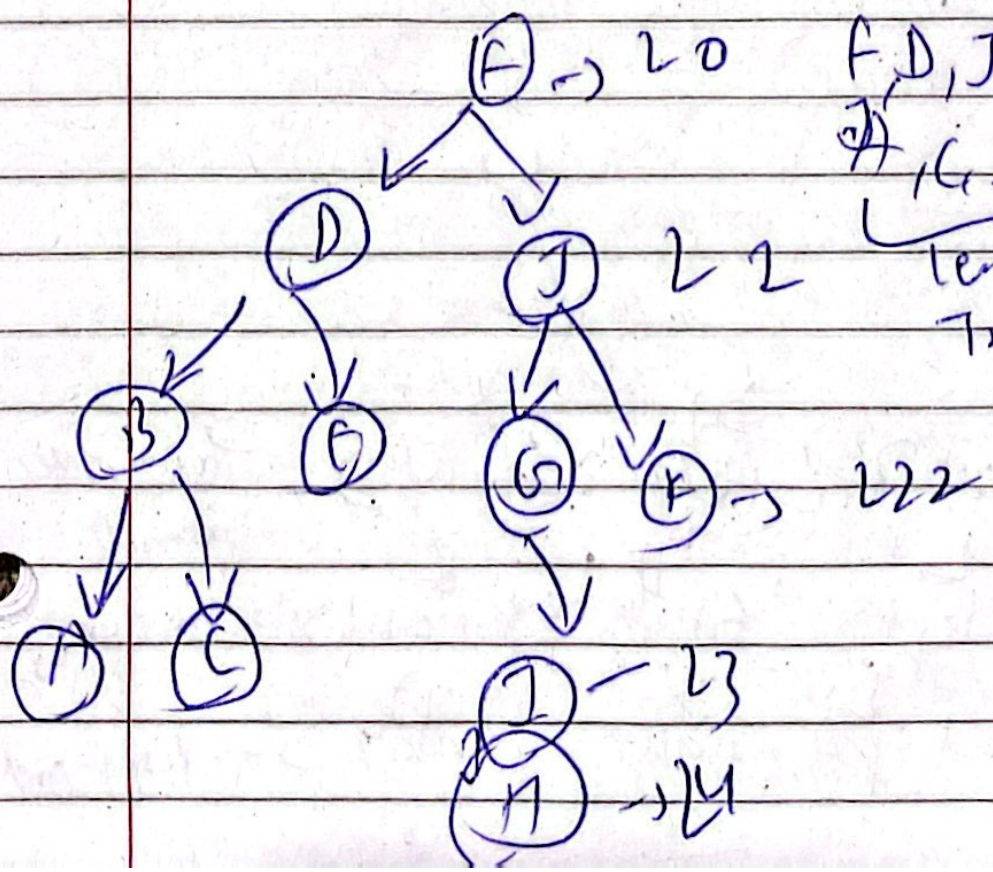
~~insert~~ insert-node

(Node ,)

if in ~~some~~ t
if n.value is not in
Node(u)
return ~~[illegible]~~ node

Binary Tree Traversal

Breadth-first -> visit all nodes at ~~for~~ some lo



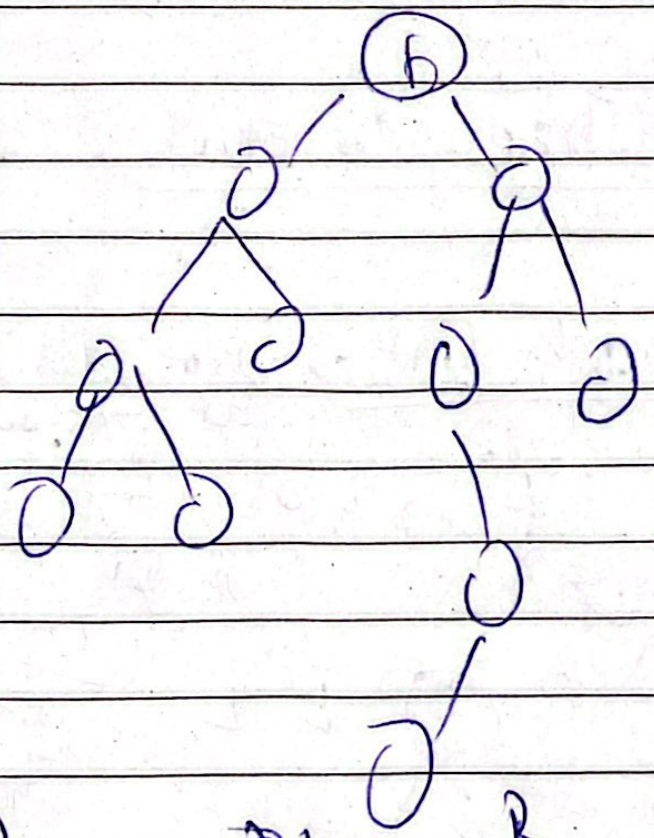
F, D, J, B, G, K, A, C, I, M

level-order Traversal

Breadth-first
- level order
- visit all children
before we visit its grandchildren

<u>Depth-first</u>

- finish all grandchildren



D L R
(root) (left) (right) - Pree-order traversal

L D R
(left) (root) (right) - in-order

L R D
(left) (right) (root) - Post order

Scanned with CamScanner

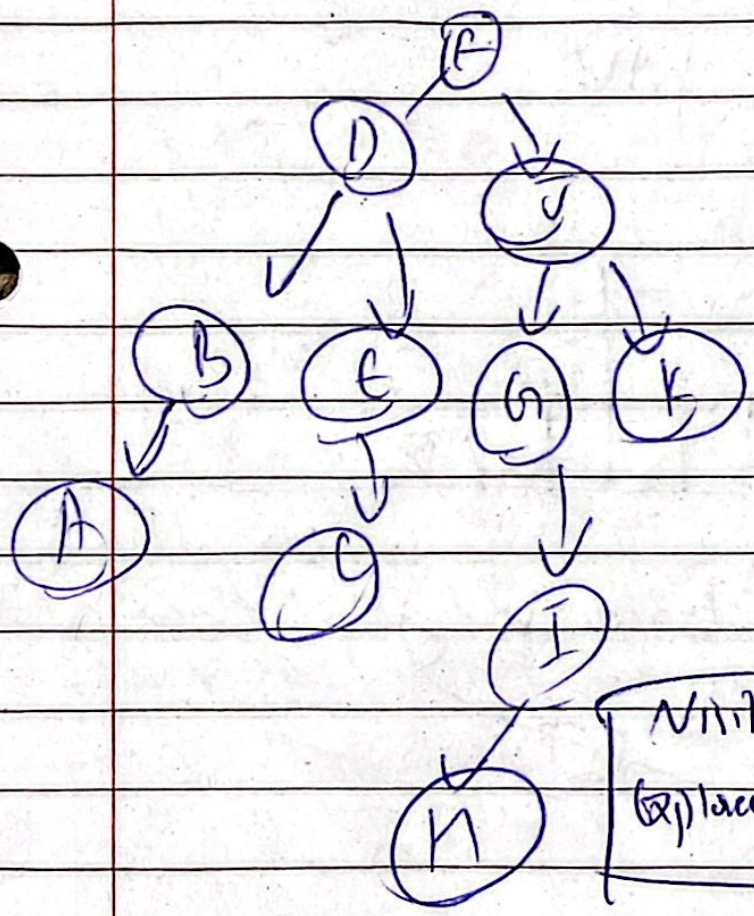
Preorder (DLR)

F, D, B, A, C, E, J, G, I, H, K

Recursion will make it easier

level-order Traversal (BFS)

Queue (FIFO)



F

Print C

Then

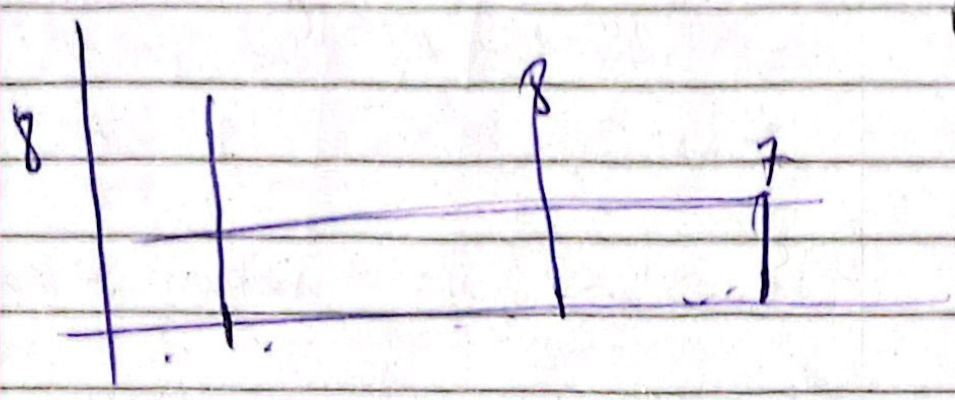
Queue

| D | J |

Visiting a vertex
1. work on node
2. Iteration: look at adjacen

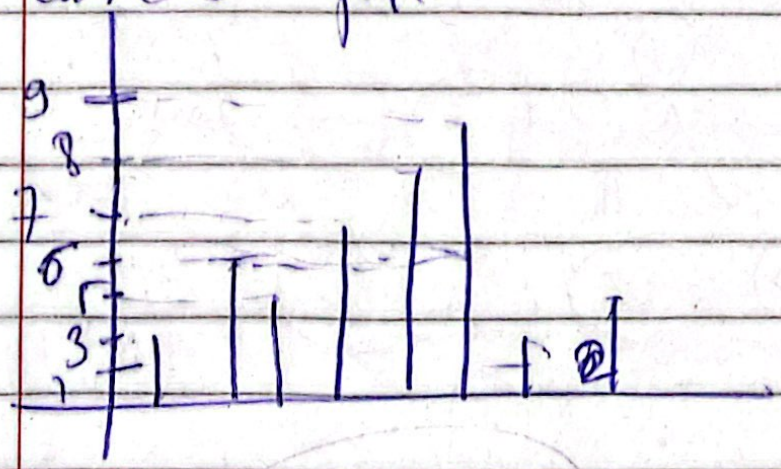
Traverse one the visited, then enqueue its children (adds) keep on going

[1, 8, 6, 2, 5, 4, 8, 3, 7]



7x7 = 49? why not 8?

case is full?



Brute force max value

[8

for i in L

for i - - -

for, k - - -

max product Brute

[1, 4, 6 ... 7]

L R

L=8 L=8

48

L R = 8

=1 8

1 6 L=4 R

= 6

=4 2 = (2)

Now, L=8 R=2 = 16

n = distance n

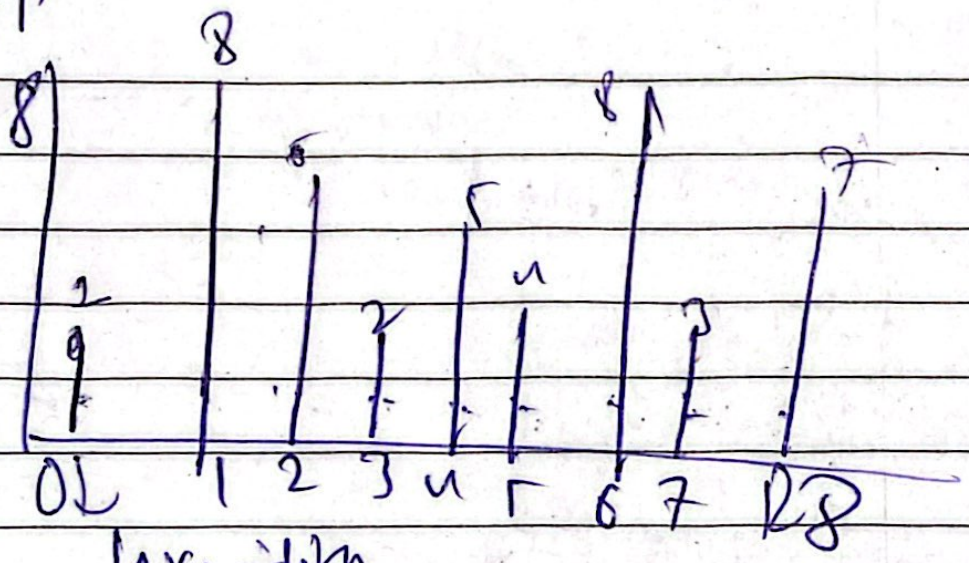
L=4 (R) 6×2=12

L=8 R=4

8

Scanned with CamScanner

Q1) final soln



Lx width
→ no of distance

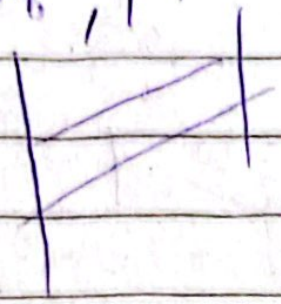
1st ~~[illegible]~~ =
max height is 1, else it
overflows
$1 \times 8 = 8$
since R > L, update L

Lt 8, 7

~~max~~ min = 17
distance = $L_i - R_i$
= 7
Lx v = 19

keep on going

19, 17



Depth first

Preorder <root> <left> <right>
Inorder <left> <root> <right>
Post <left> <right> <root>

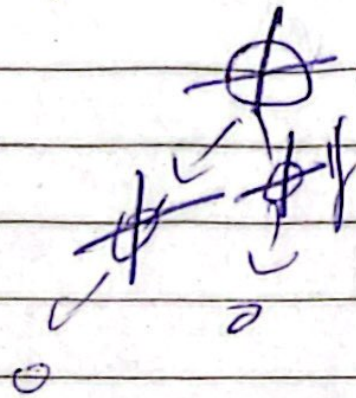
Preorder:
① Visit Node ② left subtree ③ right

```
Preorder( node *root)
if (root == NULL) return;
        print n (root.data);
        Preorder (root->left);
        ,  ,  (  -right);

Inorder (root->left) ->
print (  root->data);
Inorder (root->right)
```

for BST print in sorted

do In order



if false

check(node, prev value)

if node.value > prev value

return false

or we could do In order, & check for array

Check if a tree is binary search tree

IsBinarysearchtree(Node *root)

{ if (root == NULL) return true;

if (IsSubtreeLesser(root->left, root->value)

&& IsSubtreeGreater(root->right, root->data)

$(O(n^2))$ && IsBinarysearchtree(root->left), (root->right)

return true;

else return false;

Scanned with CamScanner

ih very expensive
simpler

bool IsBinarySearchTree(Node* root,
int minvalue,
int maxvalue)

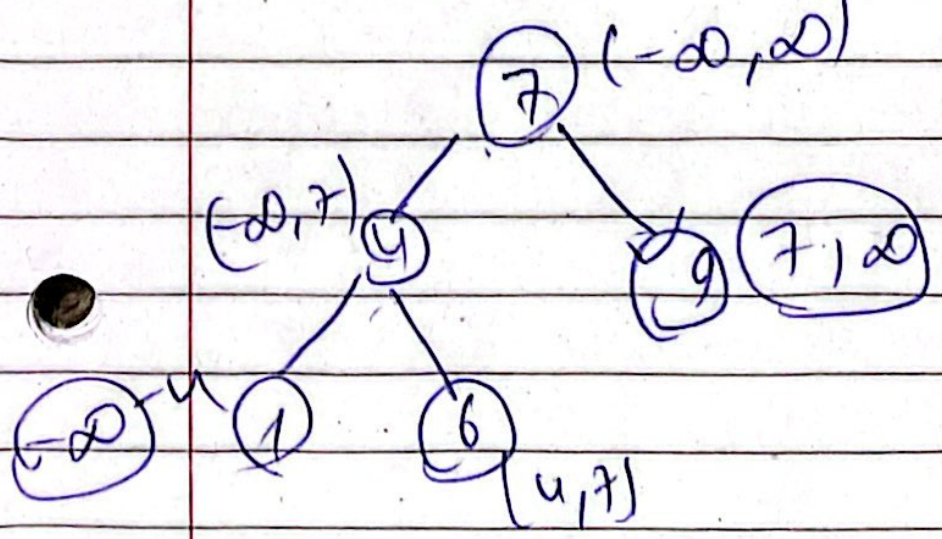
{
if root == nil { return true;

if (root->data > min
< max

&& IsBinarySearchTree(
root->left, min, root->data)

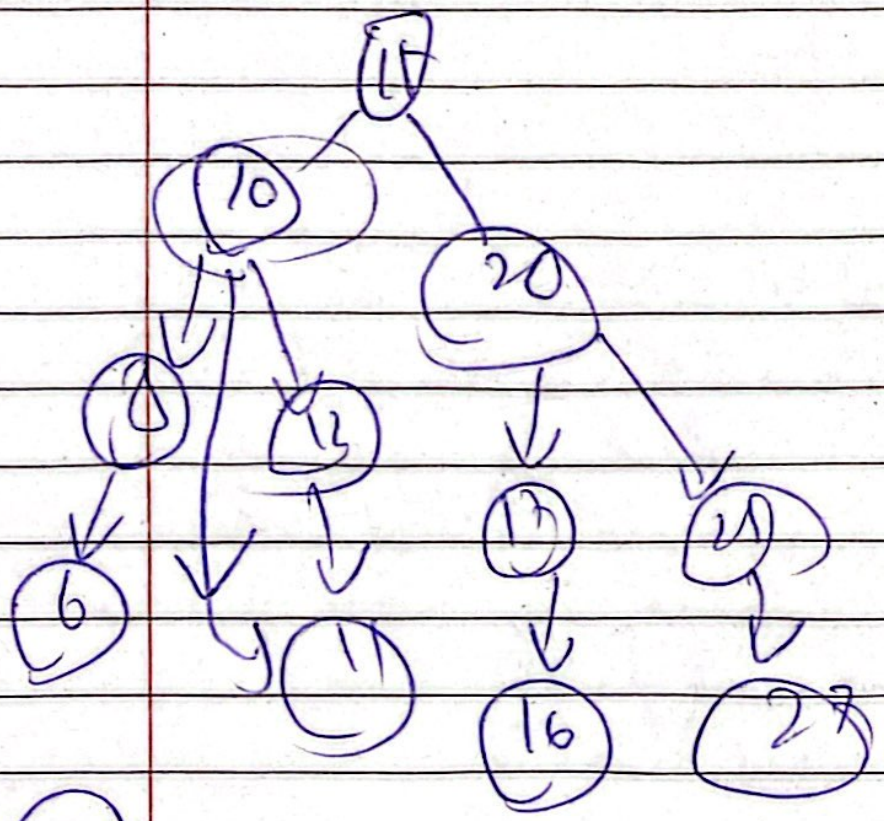
&& (root->right, root->data, max
value)

else false



(1) Inorder search Tree

- Find inorder successor in inorder traversal
- left, root, right



(i) case 1 | right subtree exists

inorder succ would be the left most nod in right subtree

case 2 i) no right subtree

(i) ~~it goes~~ if on left, it goes back to parent

(ii) if on right, in inorder, left already visited, so it will be in grandparent.
i.e. deepest ancestor

Need a node design, with reference to its parent as well

find successor (Node* root, int data)

Returns pointer to node that holds that data.

Node find (root, data);
current

O(h)

// if right subtree

if current.right != nil {

temp = current.right

while (temp.left != nil)

temp = temp.left

return temp

}

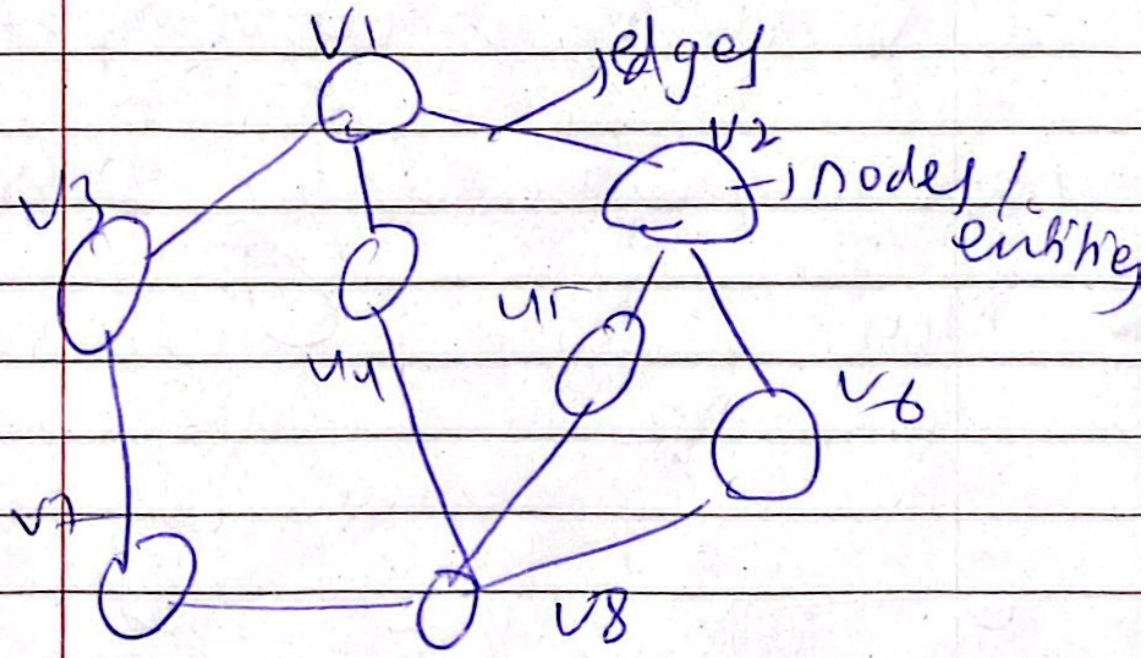
Scanned with CamScanner

anagram
nagaram

sMap[a] = 3 — tMap[n] = 2
[n] = 2 — [a] = 3
[m] = 2 — [n] = 2
[g] 1

Anthing Sort it, so both will be equal,

Graph



-) No rules for connections
-) (Discrete math)

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

$G = (V, E)$

In ordered pair

$(a, b) \neq (b, a)$ if $a \neq b$

Unordered pair

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

Order not important

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

Edges is represented by 2 vertices

types

$v \longrightarrow v$

$O \longrightarrow O$ directed

one way

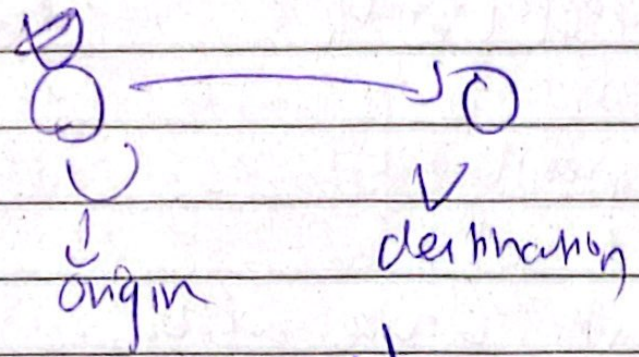
$O \text{———} O$ undirected

↳ tree members in tree

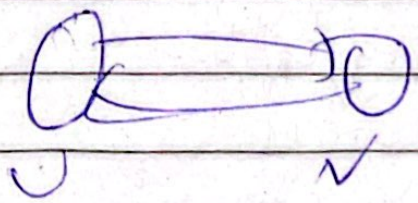
directed

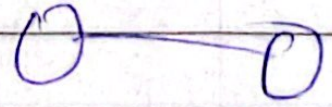
path from u to v, but not from v to u

one is destination & one is source



(u, v)





{u, u}

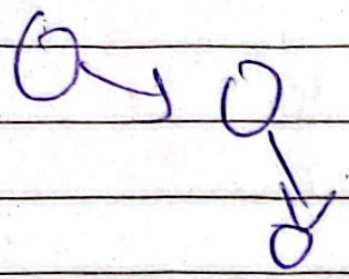
(u, v)
(v, u)

So

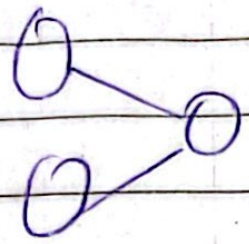
E = {(v1, v2), (v4, v3),

(v4, e)

(v6, v8)}



(Directed Graph
or
Digraph)



(Undirected graph)

- Represent data with pairwise relationship

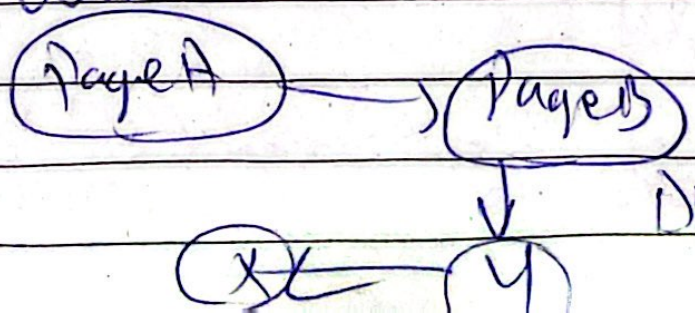
Ex:

1. Social network
   friend suggestion

(Standard graph problem

find all nodes having length to shortest ~~from~~ path from (nodes) equal to 2

2. WWW



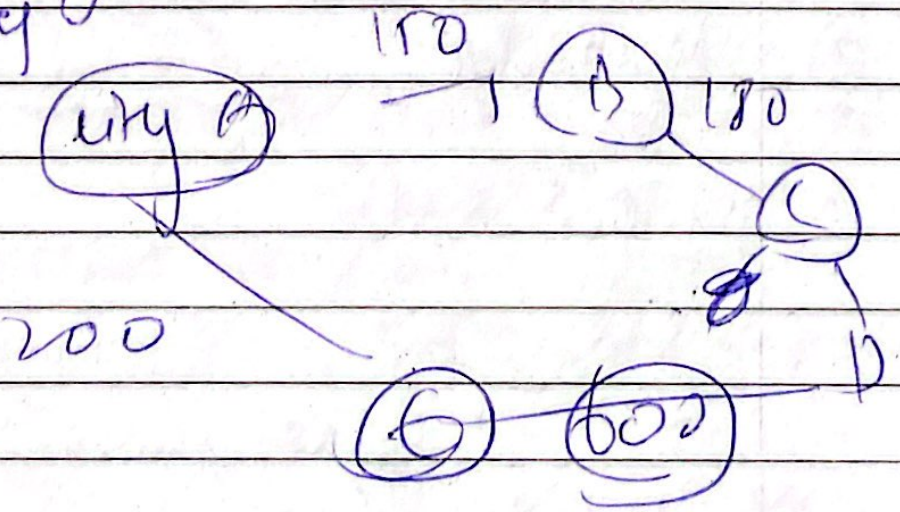
Directed graph

Scanned with CamScanner

web crawl (Graph traversal, act of visiting all nodes in a graph)

(5) weighted graph

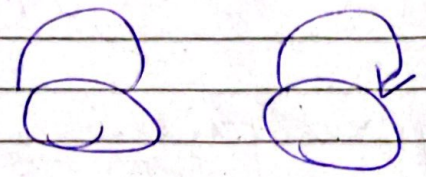
→ roadway betn cities.
→ adding weight/cost in connection, edges



Scanned with CamScanner

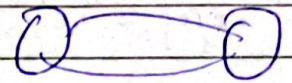
(*) Properties of graph

(1)

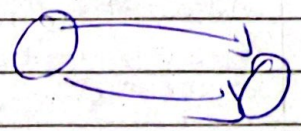


Self loop

(2) Multiedge



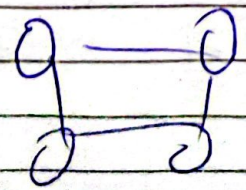
or parallel edge



flight connection
(Multiple edges to same node)
from same source

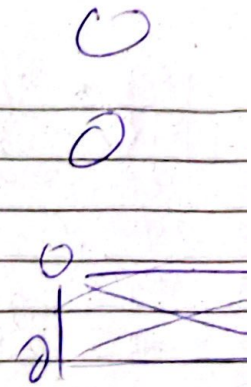
(3) The simple graph

No of edges



$|V| = 4$

if no edge
$E = \phi$



One can
if $|V| =$
0

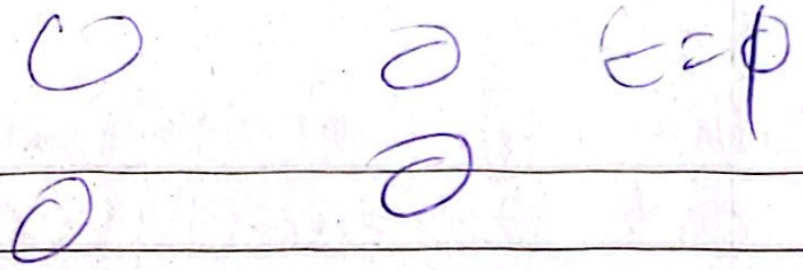
i

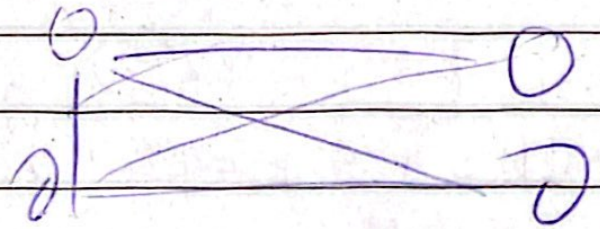
$0 \leq$

Dense →

Sparse →

(





one can have 3 connections

if $|V| = n$, then

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

if directed

$$0 \leq |E| \leq \frac{n(n-1)}{2}$$

if undirected

(adjacency matrix)

Dense → too many edges

Sparse → too few edges

(close to num of nodes/vertices)

(adjacency list)

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

Simple path: if vertices & edges not repeated
(a walk)

~~walk~~

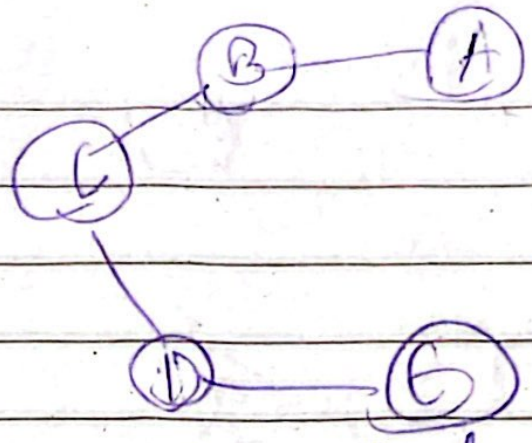
Trail = if vertices can be repeated but not edges

Strongly connected graph

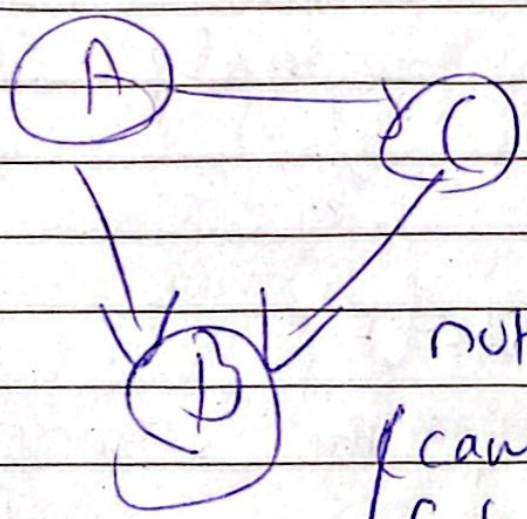
path from any vertex to any vertex

in undirected called
(connected graph)

Scanned with CamScanner



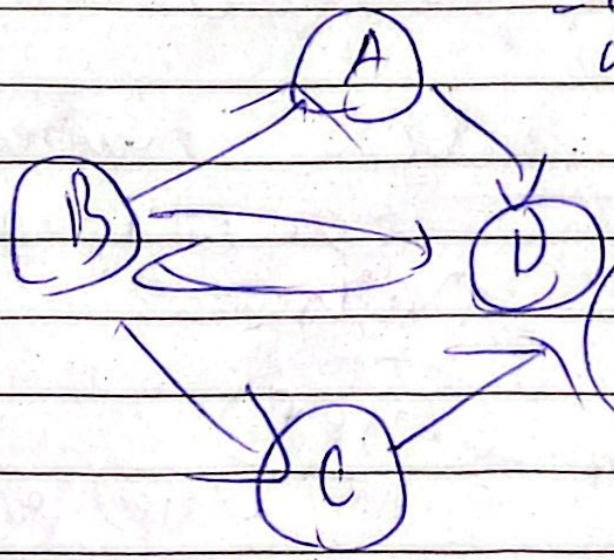
connected



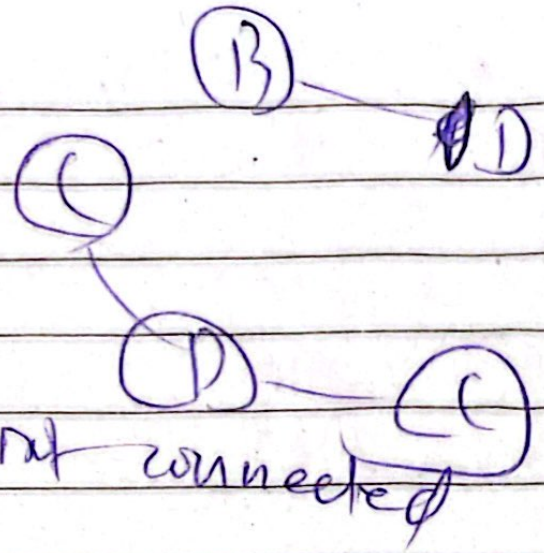
not connected

(cant go from C to A)

(weakly connected = connected if direction ignored)



strongly connected



→ disjoint graphs

Closed walk

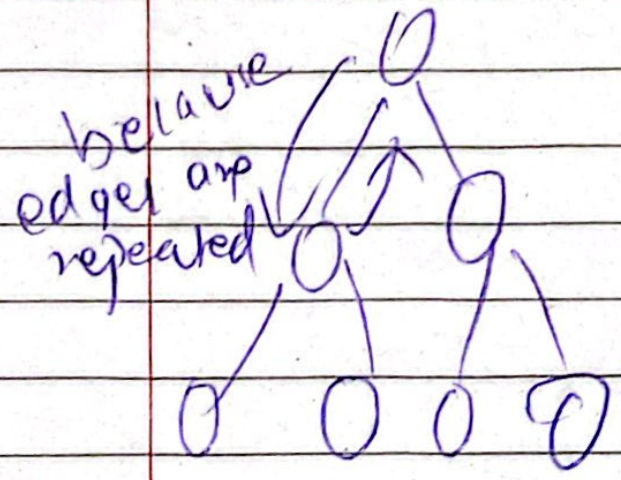
→ Starts & ends at same ~~vertext~~ vertex & length > 0

length of walk = number of edges in a path.

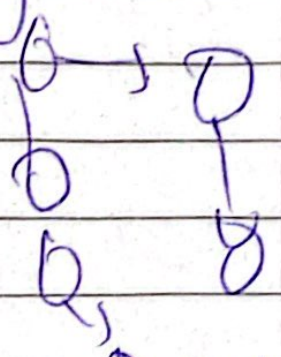
simple cycle = (cycle) a closed walk only start & end vertices is repeated, others are not.

Acyclic graph

↳ Line a graph with no cycle



Undirected Acyclic graph



DAG

(Directed Acyclic graph)

Scanned with CamScanner

map[int] map[string]int

for

map[int]

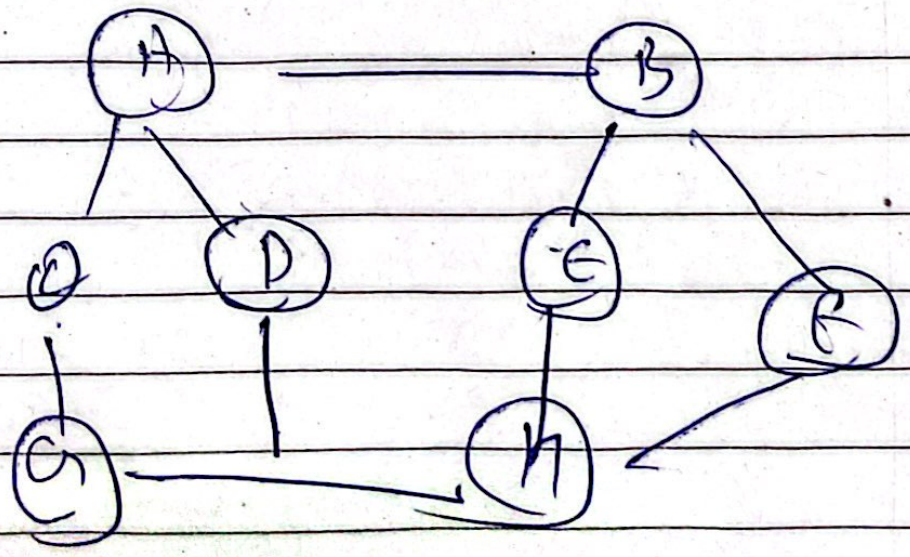
{ ch <- map[key]

Graph Representation

G(V, E)

Simplest

- a list (array, slice) to store in vertex
- another list to store edges



Scanned with CamScanner

8

vertex list

A
B
C
D
E
F
G
H

|V| rows

strings can be on diff length

~~struct~~ O(|V|)

struct edge
{
char * start vertex
char * end vertex
int ?
}

edge list

A	B
A	C
A	D
B	E
B	F
⋮	

instead have index of vertex list? or pointers for efficiency

O(|E|)

space complexity

space = O(|V| + |E|)

frequent operations

- find all nodes adjacent to a node (directly connected)

linear search

O(|E|)

Scanned with CamScanner

Check if edges are connected
$O(|E|)$

if $|V| = n$

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

$0 \leq |E| \leq \frac{n(n-1)}{2}$
if ~~that~~ undirected

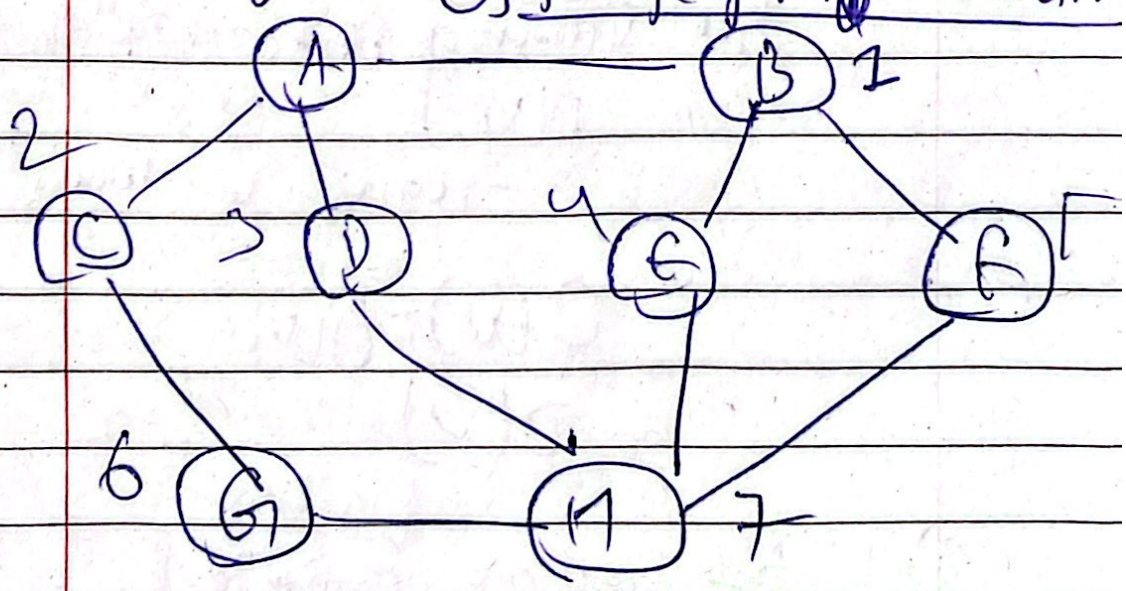
So $|E|$ can be ~~$O(V^2)$~~
$O(V^2)$
coz Hy

Efficient: store in 2D array or matrix. VXV array of edges

Adjacency matrix

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

storage of not connected



A_{ij} = 1 if edge from i to j
0 otherwise

Symmetric for undirected

$A_{ij} = A_{ji}$

so efficient

not for directed

Operation Time cost

① find adjacent nodes.

→ get index from vertex list ⇒ map?

→ Scan vertex $O(V)$ worst case

→ scanning row in matrix $O(V)$

→ scan V columns

$O(V) + O(V)$

$= O(V)$

② if two nodes are connected

→ look up value

if indices given $O(1)$

if not $O(V)$ → scan have to look up index

for weighted graph
let
A_{ij} to the cost/weight

Space complexity
$O(V^2)$: the matrix
we are using connection &
also not connection → 0 which
mean not connected

only good when graph is
dense: or $|V|$ is less
$|V| \approx |E|$

Scanned with CamScanner

Adjacency list

for A

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

{1, 2, 3}

↳ list of connection

↳ index in vertex list

[1, 2, 3] → array

(1 | 1) → [2 |] → [3]

linked list

index in vertex

tree

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

→ 1 Go

5 [7] int

not of it not shown

→ all index in vertex that have connection with

less memory

space = ~~O(1)~~ $O(|E|)$

Time:

① look if vertex are connected
- we have to scan through a row

| 0 | 1 | 2 |] | - linear search

- if sorted, can do binary search $O(\log)$

- worst case, v cells in row
$O(V)$

find all neighbors

$O(V)$, same as adjacency matrix

Append a new value

→ In adjacency matrix just flip a bit

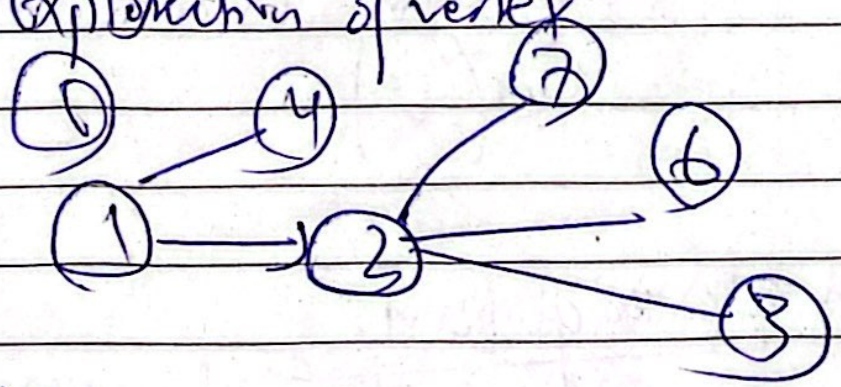
→ but in list, have to create new array, append & replace.

so linked list would be better here

Traversal problems

BFS & DFS

① visiting a vertex
② exploration of vertex



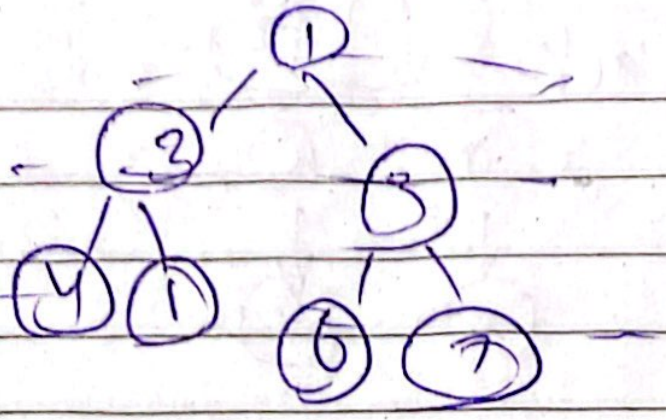
BFS: 1, 2, 4, 5, 7, 3, 6

→ look adjacent nodes first
all visiting a vertex first

DFS: look at each & explore them first

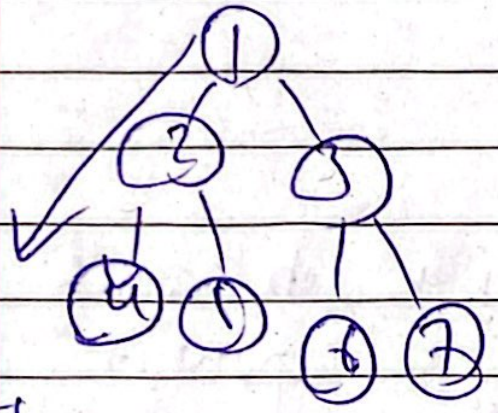
1, 2, 3, 6, 7, 4, 5

Scanned with CamScanner



BFS: ~~level order~~ level order

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

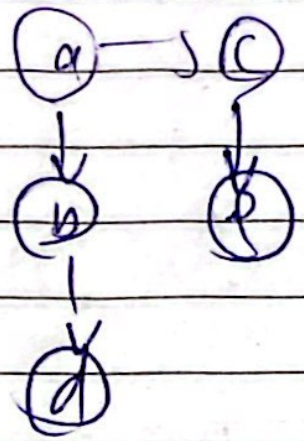


~~DFS~~

DFS: 1, 2, 4, 5, 3, 6, 7

Pre-Order

DFS = stack (LIFO)



stack

a

↳ pop off a, its current
consider its neighbors & push it
print it

b
c

← second
← pushed first

pop (b) → print it
(consider bs neighbors)

[d, c] → current
print pop off
(consider its neighbors

[d, c] → ///

no more

[c] → pop off

[c] → ///

Stack empty

BFS → Queue order FIFO

(9)

Queue → remove current
consider B, C (its neighbor

c b → remove (b) consider its neighbor

d c → ////

Scanned with CamScanner

c is neighbor

~~e d f~~

~~f e~~

~~f~~

for list

graph = {

a: ['b', 'c']

b: ['a']

c : ['e']

d : ['f']

f : [ ]

f : [ ]

}

~~[graph, source]~~

~~stack [ ]~~

stack ['a']

while len(stack) != 0 :

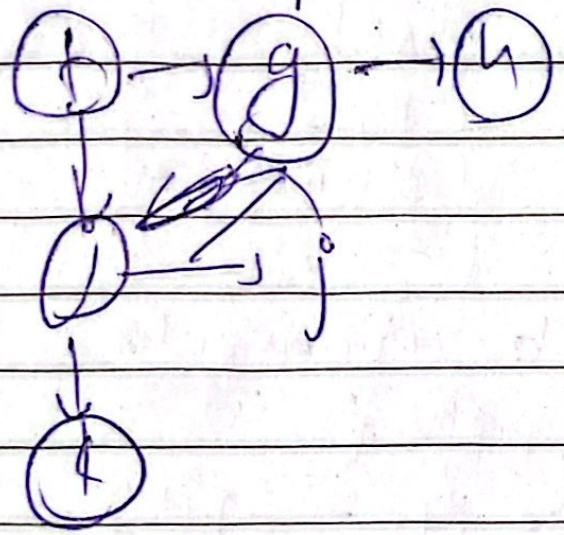
neighbors = graph[a]

stack.pop()

~~append~~ stack.push [neighbors]

hamiltonian path problem

directed path




```
{ f: [g, i]
  g: [h]
  h: [];
  i: [g, k]
  j: [i]
  k: [j];
}
```

src: f
dst: k

can path betn nodes?

DFS, or BFS

if current node == dstn
just return

n = # nodes
e = # edges $T = O(e)$
$S = O(n)$

both ways

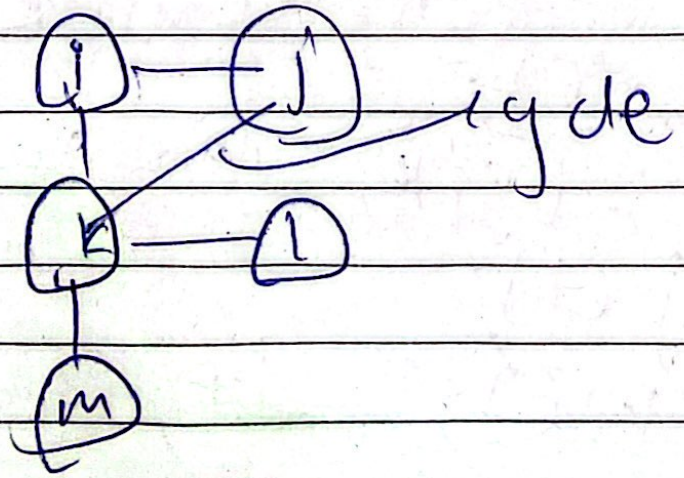
Undirected path problem:

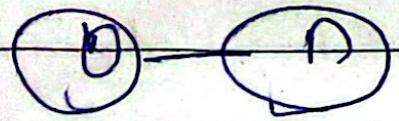
edges: [
[i, j]
[k, i]
[m, k]

① Convert to adjacency list

graph: {
i: [j, k]
⋮
j: [i]
}

inverse connection





⇄ cyclic behavior

Scanned with CamScanner

~~dfs~~ src = 0 , dst = 2

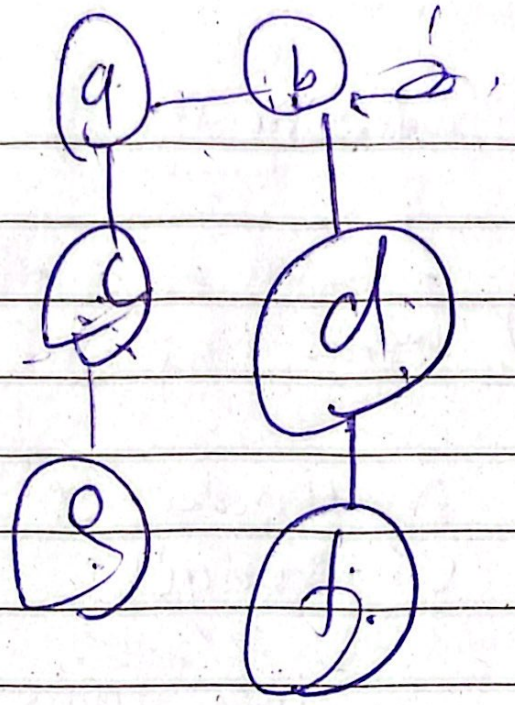
(3) - mark as visited, to avoid infinite loop

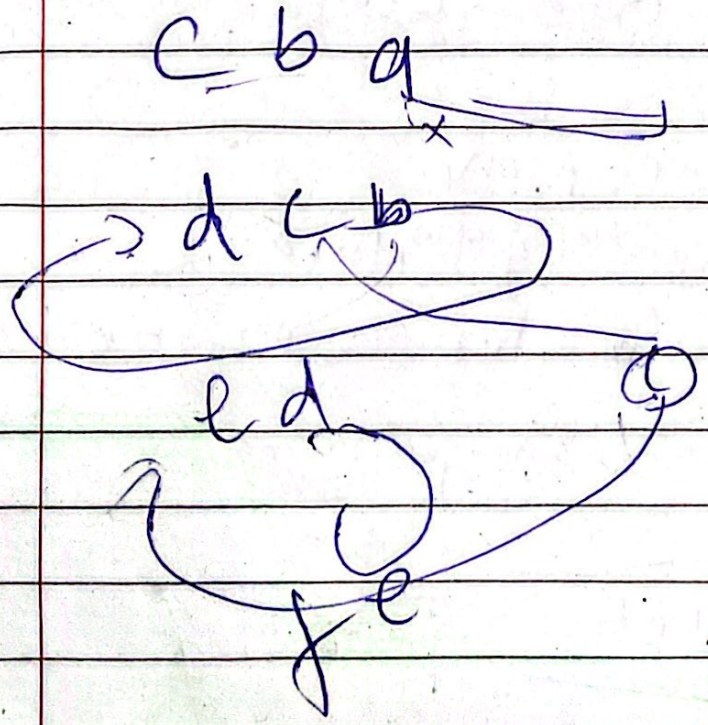
n = # nodes
e = # edges

Time O(e)
Space = O(n)

Recursive soln
bool path(graph, src, dst)
for let (n in graph[src])
bool path(...)
}

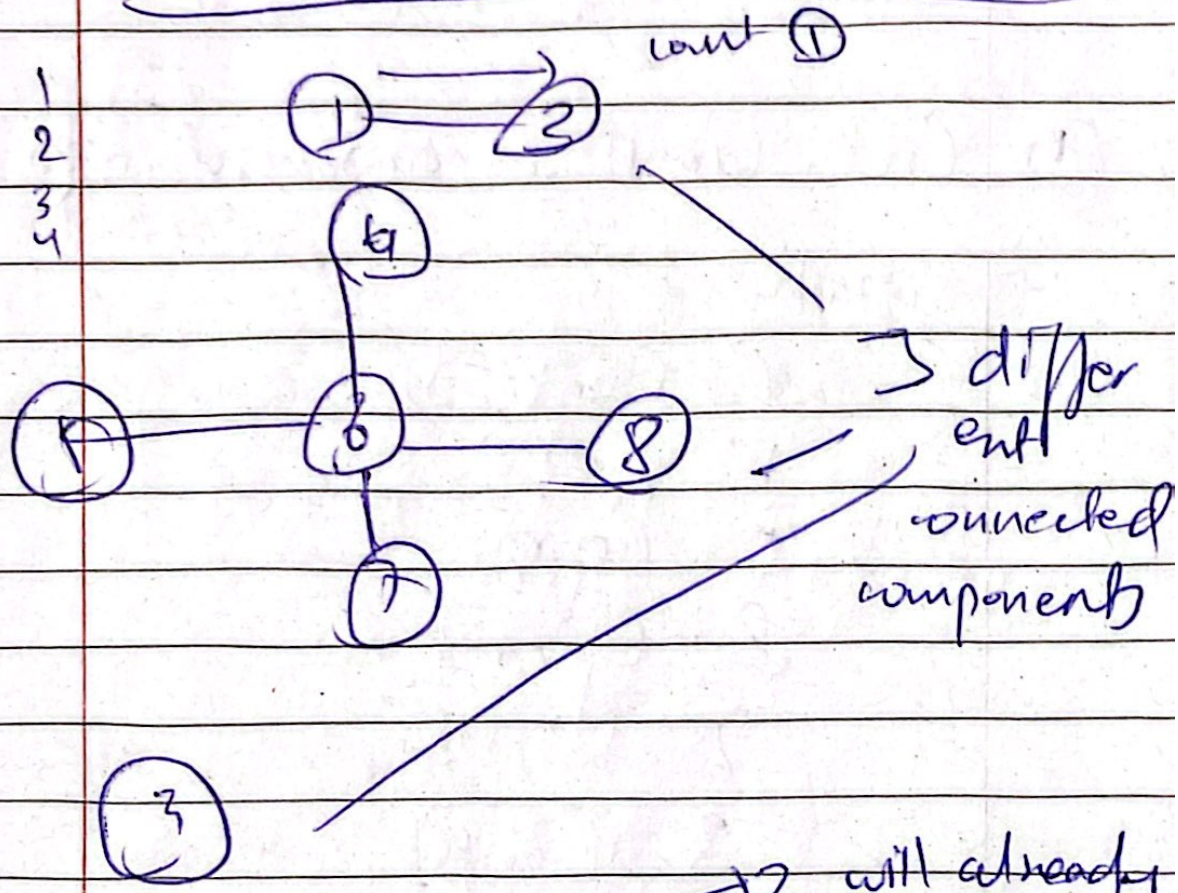
map or set for visited





Scanned with CamScanner

connected components count



2 will already be visited on

① List nodes

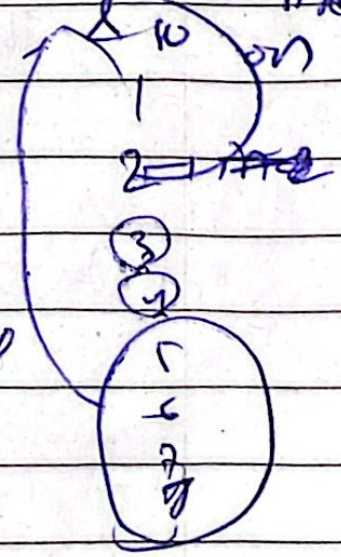
[1, 2, 3, 4, 5, 6, 7, 8]

count = 0

Traverse

for i in nodes

d, p, f, b, s



② if end reached increased count

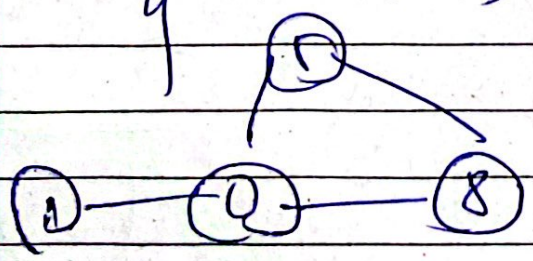
③ Mark visited

n = # nodes
e = # edges

Time = O(e)
Space = O(n)

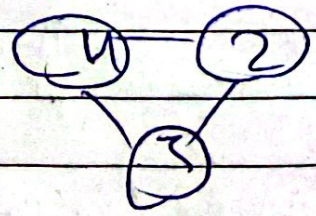
(8) Get largest component size

graph = {
0: [8, 1, 5],
1: [0],
5: [0, 8]
8: [0, 5]
2: [3, 4],
3: [2, 4]
4: [3, 2],
}



size = 4

no of components



big = 3

Nodes:
0
1
2
3
4
8

Trave

(*) Shorte
edge
[
[
[3,
[3,
[w,
}

sla
co

same as before, also count

Nodes:

0 Traverse (Mark nodes as visited)

1 ✓
2 ✓
3
4 ✓
8 ✓ visited

current largest

Traverse again & count it.

Time $O(e)$

Space $O(n)$

* Shortest path

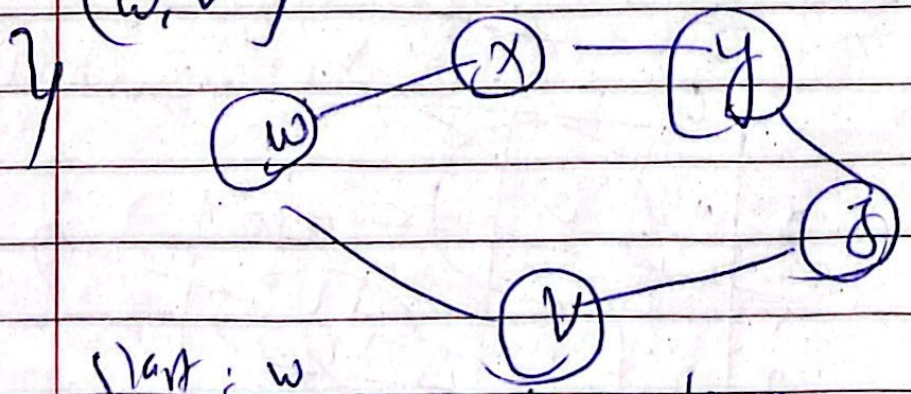
edges of

[w, x]
[x, y]
[z, y]
[z, v]
[w, v]

into adjacency list

{w: [x, y, z, v]
⋮
}

Path length: (No of edges)



start: w
end: z

Path length: 2

Track visited

BFS is gonna be more useful
cause we look at all

current path:

① (x,1) (w,0) → path
(v,1) queue current (w,0)

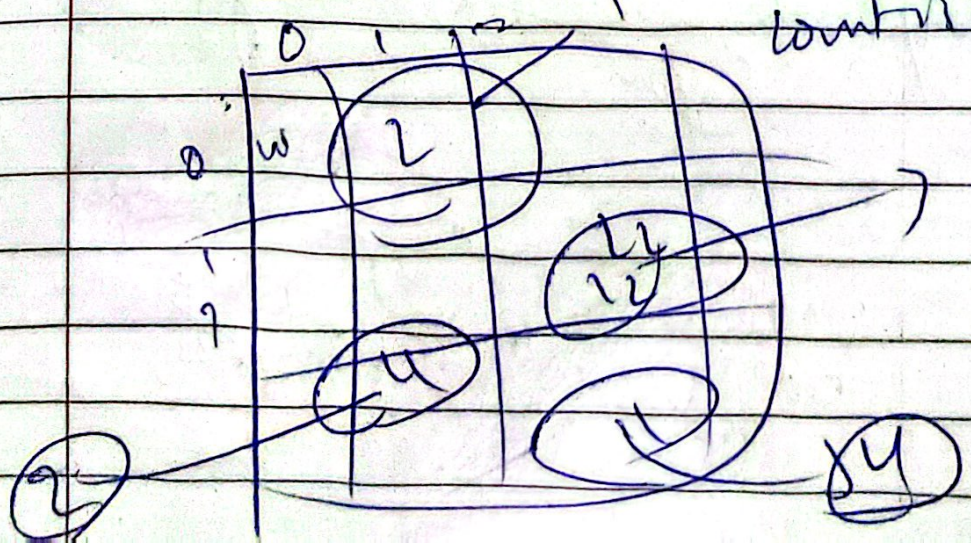
(y,2) (v,1) (x,1)
(z,2) queue current (x,1)

O(n)

[w,x]

Island Count

count island



① iterative loop

② got there do DFS, mark as visited

	0	1	2	3	4	5
0	W	L		W	L	W
1		LW		W	L	W
2	W	LW		W	W	W
3		L		W	W	
4			W	L	L	W
5		L	W	L	L	W
6				W	W	W
7						

③ Increased count keep on going since node is marked as visited ~~so~~ we don't go through it again.

r = rows O(rc)
c = cols space O(rc)

const grid = [
  ['W', 'L', 'W' . . .
  ]

visited = new Set() → position

for row = 0; row < grid.length . .

  for column . grid[0].length

    explore(grid, r, c)

}

explore(grid, r, c, visited) = s/

rowInbounds = 0 <= r && r <= grid.length

pos = [r, c] or "r, c" (string rep)

Here,

if visited.has(pos) return false
not a new island

visited.add(pos);

colInbound = 0 <= c && c < grid.getLy

if (!rowInbound && !colInbound) return false;

if grid[r][c] = "w" return false;

up

explore(grid, r-1, c, visited)

r+1
down

c-1
left
c+1 right

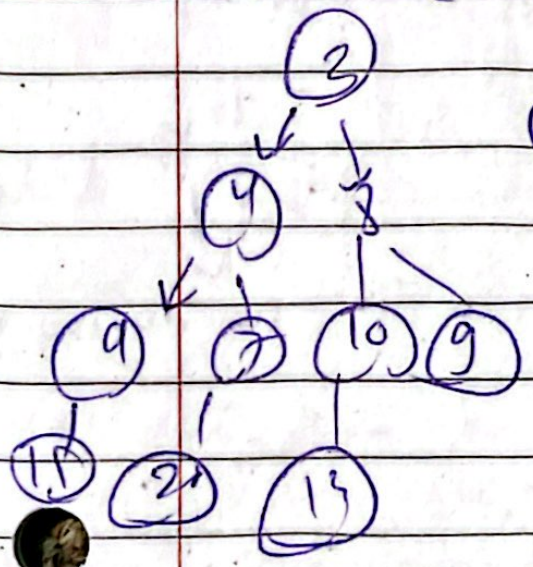
return true
// Explored brand new Island
// we have marked visited

(7) Heaps

(1) Min Heap

- its child are all to smaller than their children

root node is always smallest



(a) insertion

- at bottom, from left to right then keep checking till its in in right place

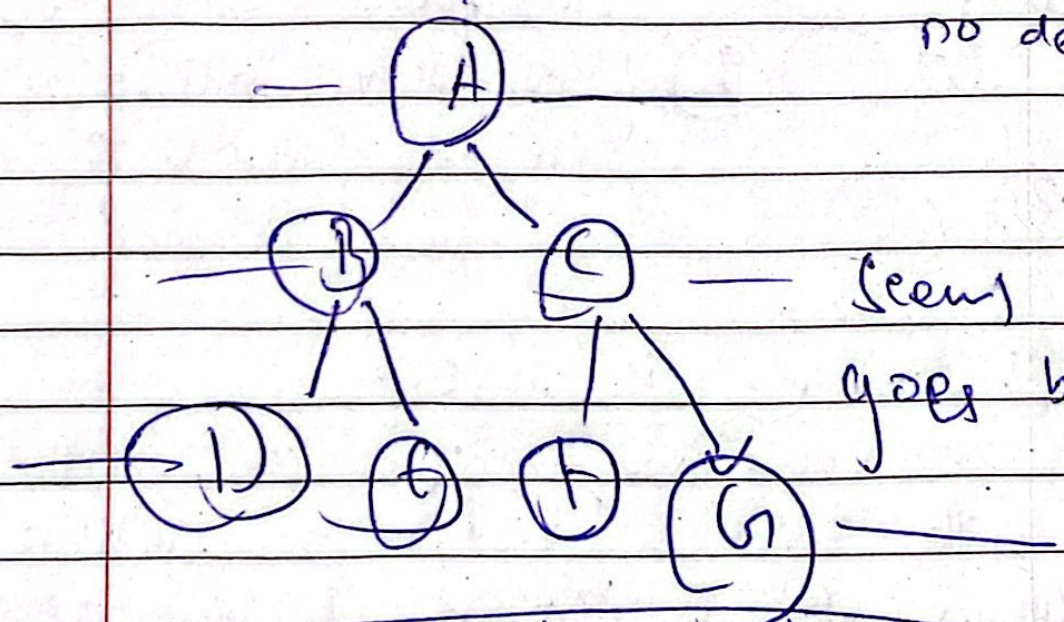
(b) Remove the smallest

- remove root, get the most bottom & put in root, then bubble down to put the right one at top.

Heap is a complete binary tree

In array

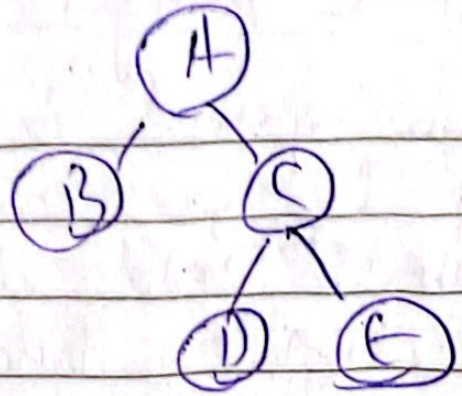
→ if a node at index X → i
left child at → 2×i
right child at → 2×i+1
parent at → $\left\lfloor \frac{i}{2} \right\rfloor$ floor value
↓
no decimal



seems goes breadth wise

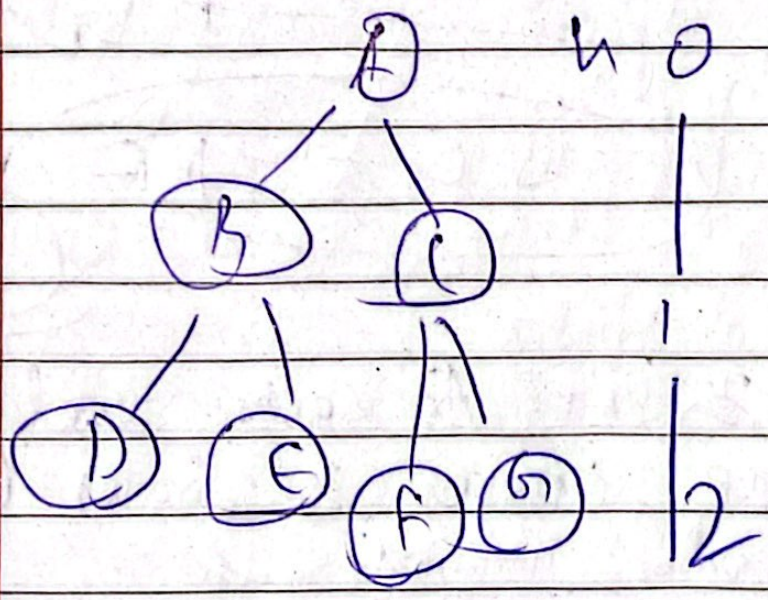
A	B	C	D	E	F	G

Breadth first traversal



A	B	C	-	-	D	E

if element is missing so fill blank



if nodes are full at all height, it is called full binary tree

$2^{h+1} - 1$ nodes

Complete Binary Tree

When you represent in binary tree in array, if there no missing value in both field it is called a ~~bin~~ complete binary tree.

A	B	C	D	E	F	G

✓

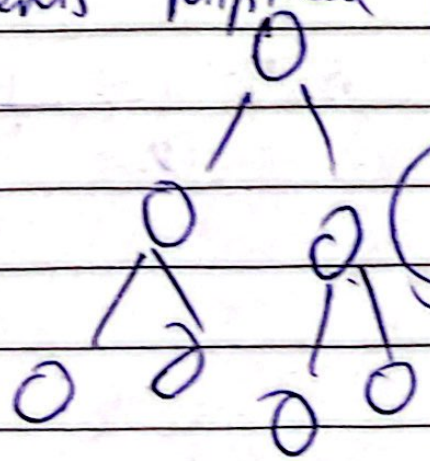
A	D	C	D	E	F

✓

A	B	C	-	-	D	E

✗

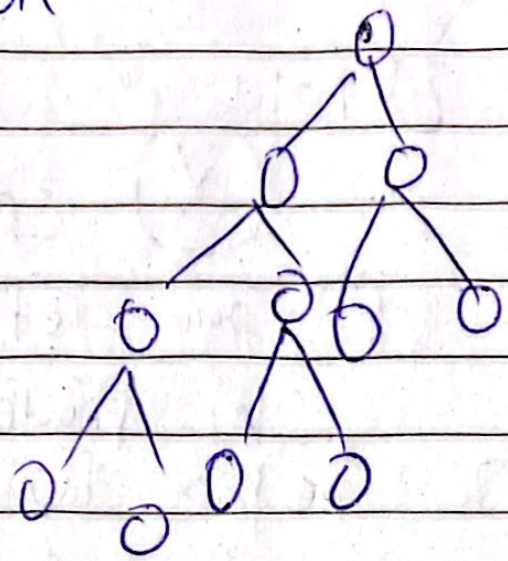
→ i.e. At each level in a tree there 2 nodes (no nodes missing.
(elements fillpilled from left to right)



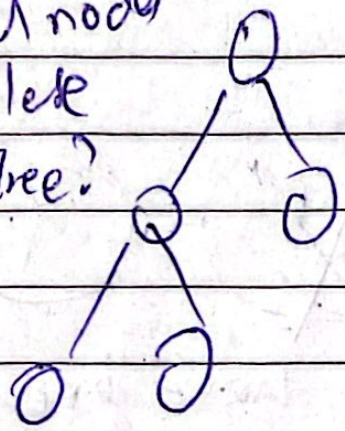
✓

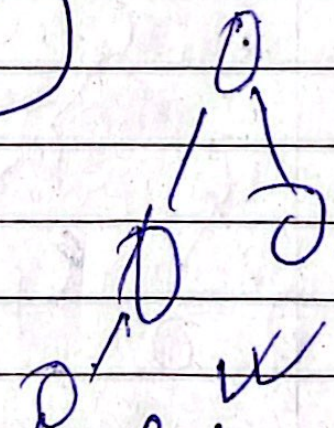
Scanned with CamScanner

~~1) complete BT of height h is a full~~

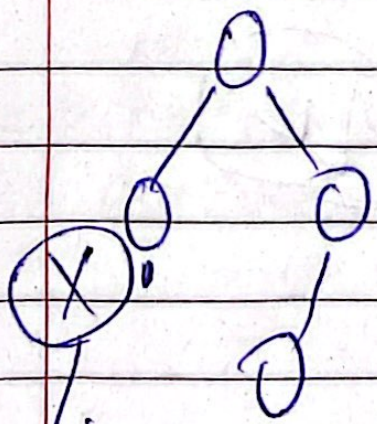


Which nodes are complete Binary Tree?

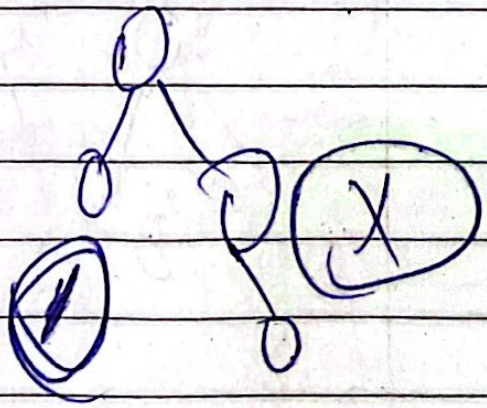


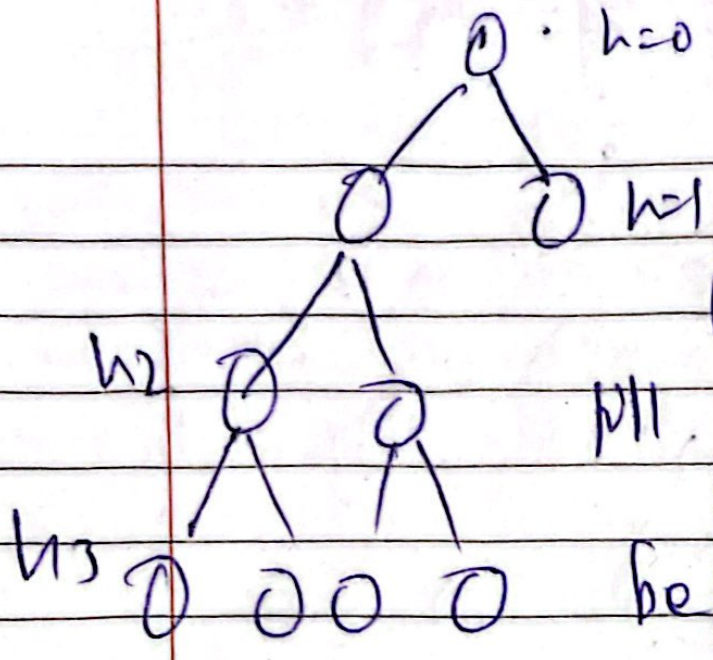


filled from left

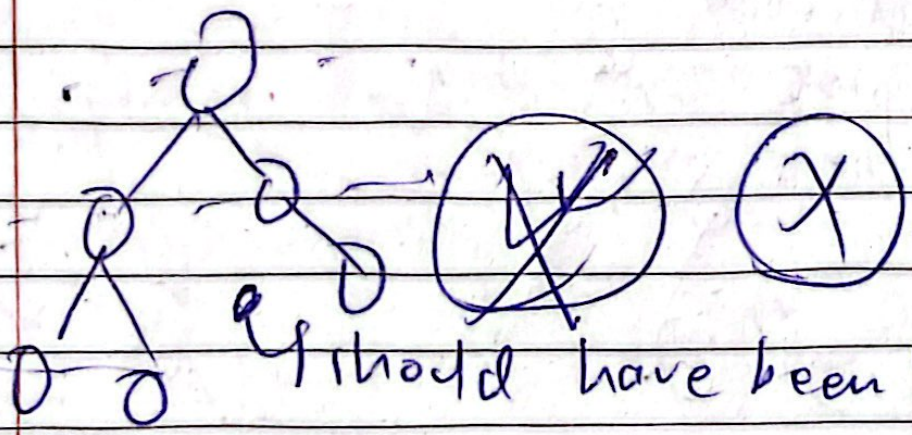


not filled from left



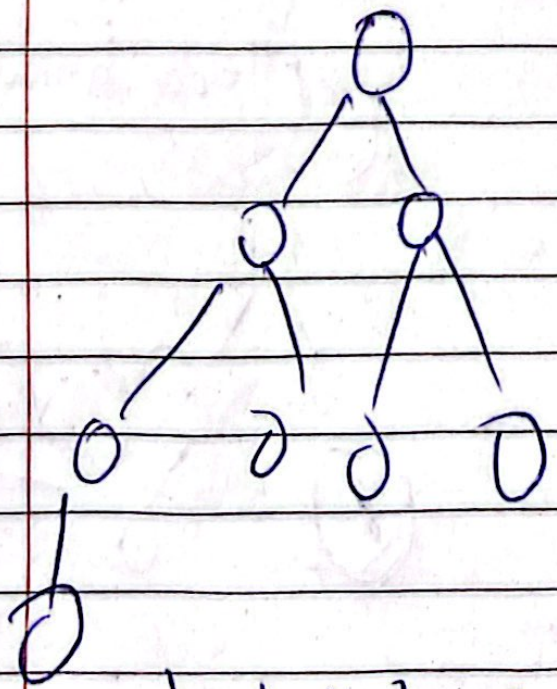


(X) cause
fill from left to right
h1 should be filled
before h2.



(X)

should have been here



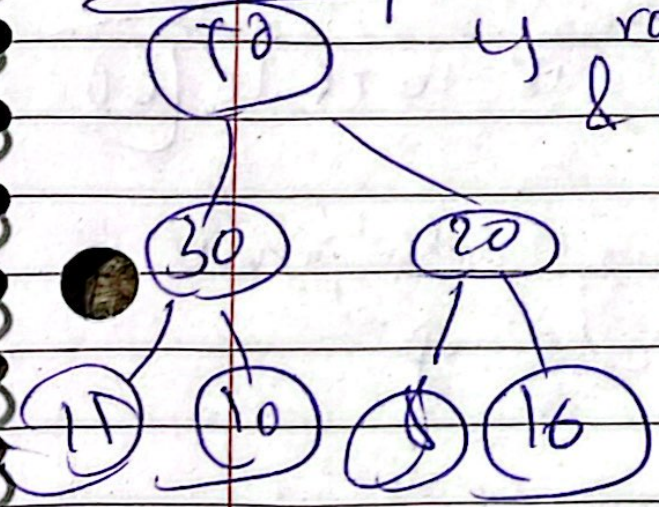
(✓)

last is here so complete
Binary tree

Height of complete binary tree is $\log n$ cause we won't go to next level until we go from left to right

Heap: lift complete Binary Tree

Max heap

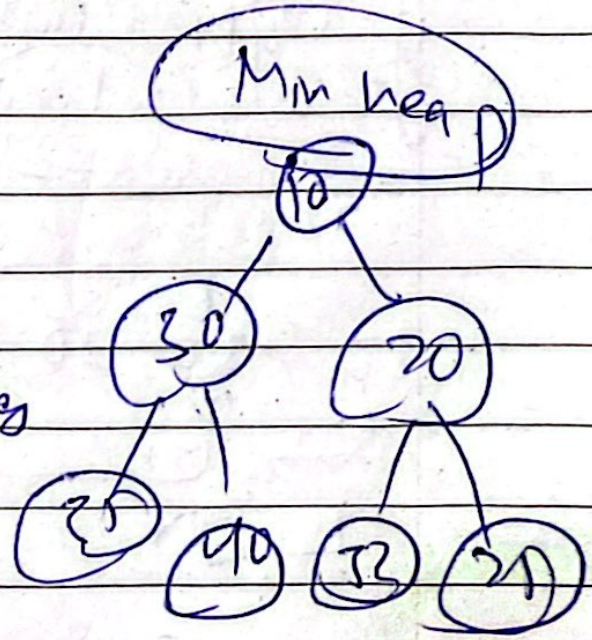


↳ root is largest, & parent is always greater

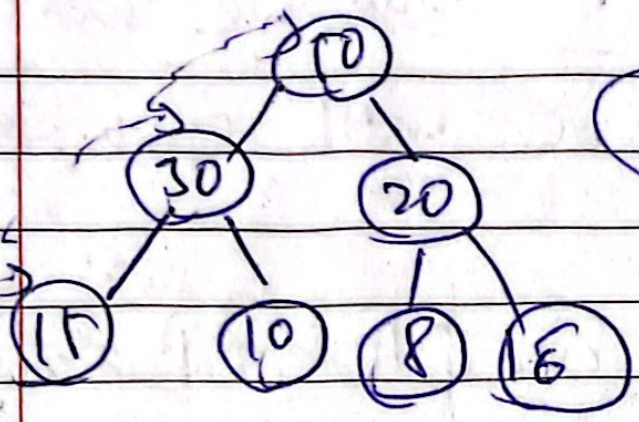
duplicates allowed

~~Min h~~

each node is smaller than descendant



Insertion Max Heap



Insert 60

60

1	50	30	20	15	10	18	16	

① Insert in last

50 30 20 15 10 18 16 60

② keep company with parent
if high, switch it up
(swap)

50 30 20 15 10 18 16 60

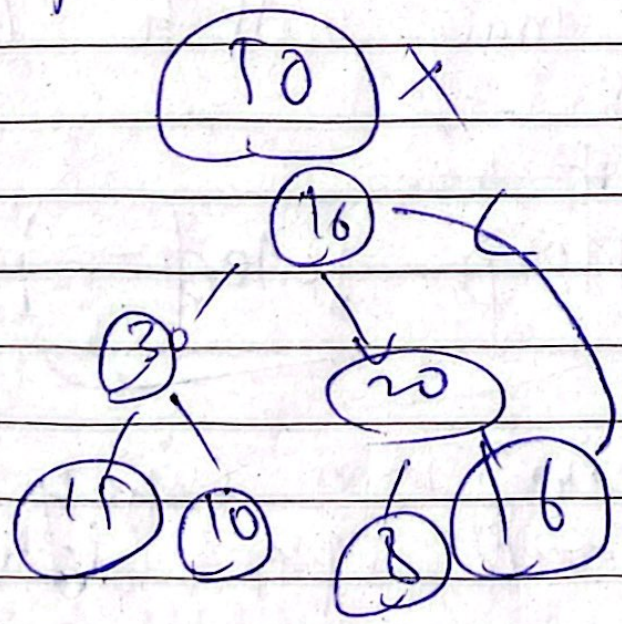
worst case $O(\log n)$

min $O(1)$

Delete

- only delete root, no other choice.
- can not adjust as we like to preserve the property

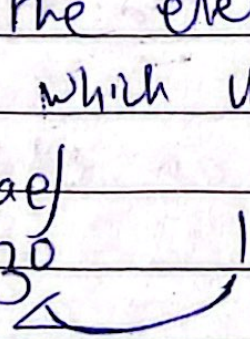
last element, on the right will go root



30 20 15 10 8 16

adjust the element, which child is greater, & replace

(the formated



30	10	20	15	10	8

Time taken $O(\log n)$

- Note! After deletion, we can use put values in array (last space) & ↓ last index

Heap sort

Since highest value will be in the end array will be sorted

~~It be~~

Heap sorted

→ Delete in an empty place obtained after deletion

Heap sort

1. from an element
   → Create a Heap

2. Delete an element

Scanned with CamScanner

10 20 15 30 25 40 5

1 10

3 10

20

Insertion

10 | 20

31

compare with ancestor

20

20

4

20 | 10

20 | 10 | 15

5

20 | 10 | 15 | 30

20

10

15

30

parent 15 is smaller to no need to changes

30 | 20 | 15 | 10

6

30

20

15

10

40

40

40 | 30 | 15 | 10 | 20

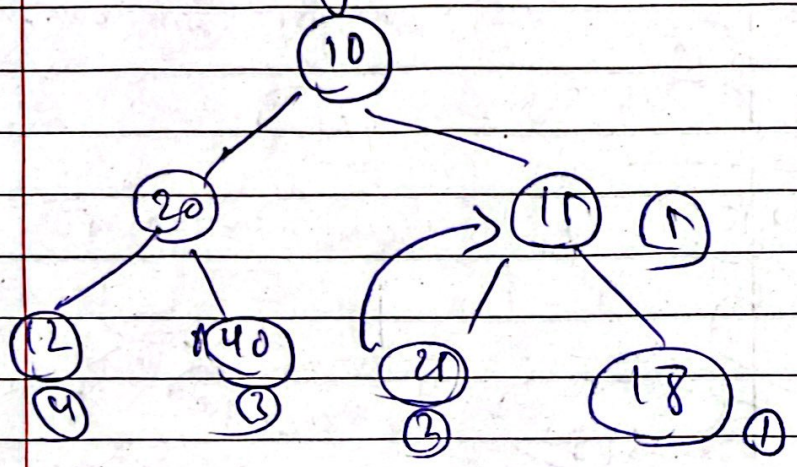
n elements

inserting depends

n log n

No start Deleting

Heapify → ~~to~~ change a random array to heap



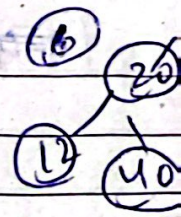
10	20	15	12	40	25	18

① check from last to front, one node are all heap no need to change

5 → 15 has children, compare 25 or 18, 25, so, 25 goes up

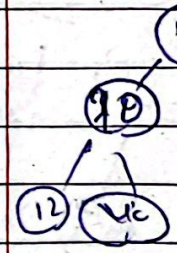
10 25 12 40 15 18

Below



so 40 is g

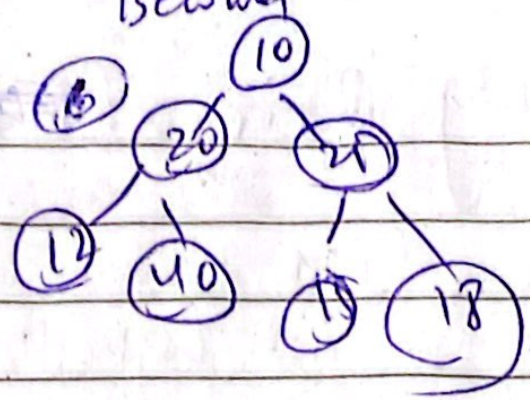
~~to~~ repeat



Again we need

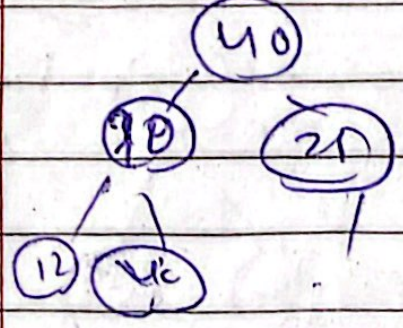
Time Take

Becomes



40 is greater 10 switch

~~to~~ repeat till last



Again check, 10 12 & 20

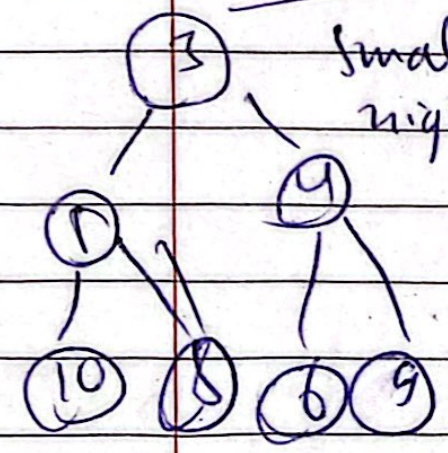
we need to check again & replace

Time Take O(n),

Priority Queue ~~FIFO~~

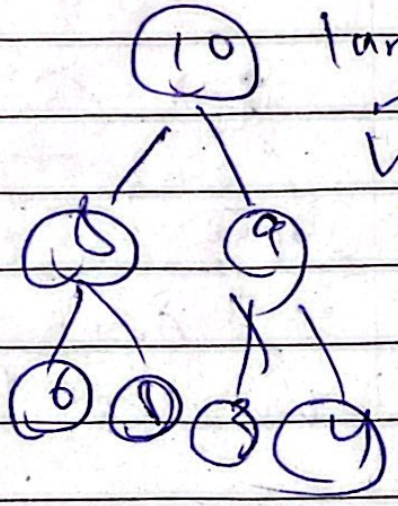
- Queue → FIFO
- → in PQ ~~th~~, get highest priority from Queue
  - there number itself has priority

min heap



small numbers higher priority

max heap



larger numbers higher priority

(log n) for deleting

Kadane Algorithm

→ similarity with 2 pointers & sliding window

Problem: | 4 | -1 | 2 | -7 | 3 | 4 |

we could brute force, but here's Kadane

| 4 | -1 | 2 | -7 | 3 | 4 |

window ← | 4-1=3 | 3 | +2=5 |

→ we just work on sum

| 5-7=-3 |

since last sum was less than 0, we ignore, reset current sum

| 3 | 4 |

3+4=7

```
def kadane(nums):
    maxsum = nums[0]
    cursum = 0
    for n in nums:
        cursum = max(cursum, 0)  # make sure it is not (-ve)
        cursum += n
        maxsum = max(maxsum, cursum)  # replace if larger
    return maxsum
```

Sliding window variation to get indices

```
def slide(nums):
    maxL = 0, maxR = 0
    L = 0
    for
        if curSum < 0:
            curSum = 0
            L = R + 1 reset
        curSum += nums[R]

        if curSum > maxSum:
            maxSum = curSum
            maxL, maxR = L, R

    return [maxL, maxR]
```

Scanned with CamScanner