Date: M T W T F S

(21) AVL — data, left, right nodes, height

∵ constructor

root -1 height

Null Null

— Height 1

— Height 2

— Height 3

Balance factor : Left sub − Right sub

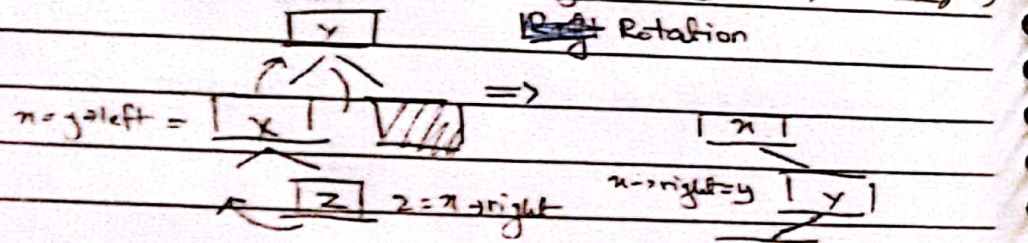
-1

2 — -2

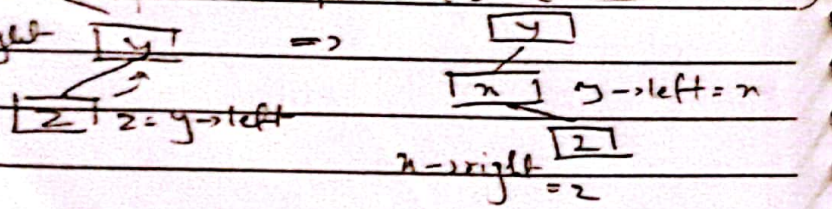
3

3 − 2 = 1 BF

Right ~~(left, subn right)~~ Rotation



Left Rotation ~~(left, subn left)~~

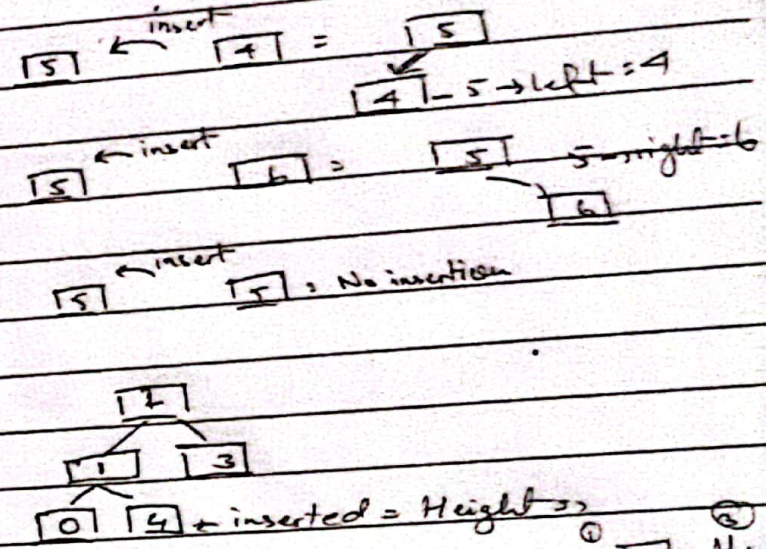


∴ Right Rotate = BF > 1, val < n→left.data

∴ Left Rotate : BF < -1, val > n→right.data

(22) Insert :-

Null root → make new using new constructor



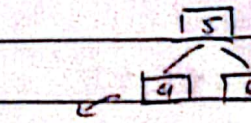
Inorder = Recursion → node→left // first

→ node.data

→ node→right

(25) BST data, left, right

-> constructor



Inserted at root -> left as 4<5

inserted at root -> right as 5<6

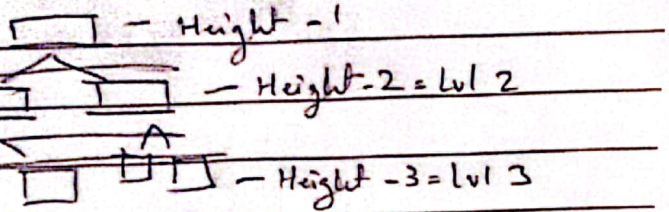
Can print at same level

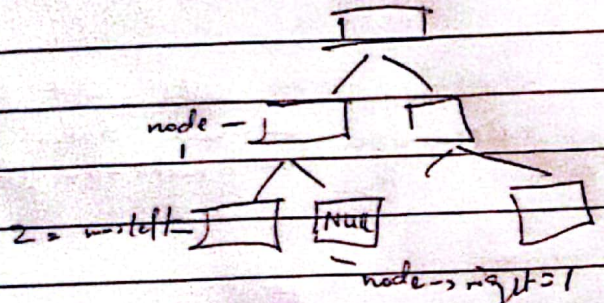
Recursion for (node->left, lvl-1)

so when lvl=1

Printing can start

then do same for right

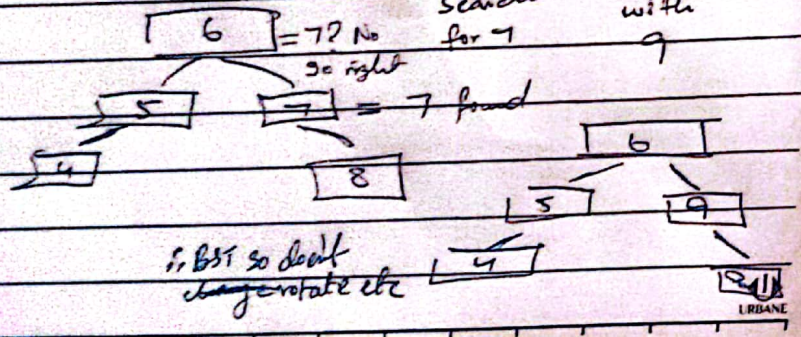




node->height = node->left = 2

as max

Update ->



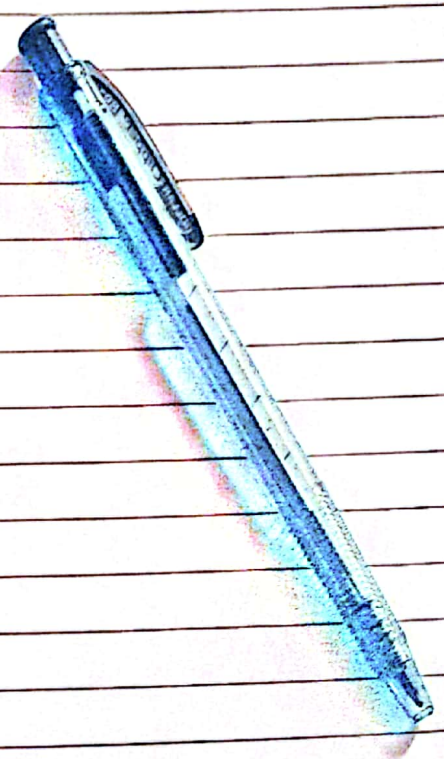
if BST so don't change/rotate etc

(27)

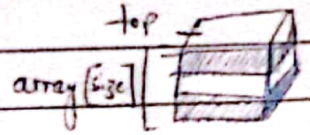
Preorder = Recursion
- current
- left
- right

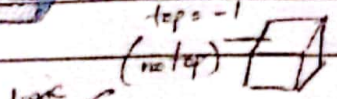
Inorder : Recursion
- left
- current
- right

Postorder : Recursion
- left
- right
- current



Q29) Stacks (array, linkedtop) ∴ can be used using arrays classes or linked lists





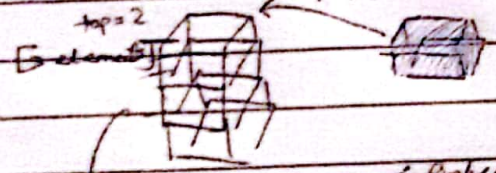
logic

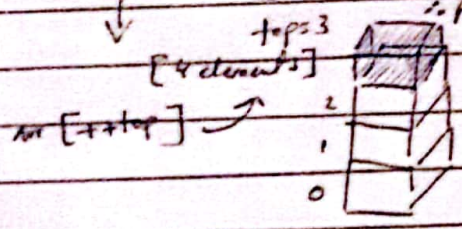
array[starts from 0] — no neg. val

array [0]
array [1]
array [n]

can be tops

Push



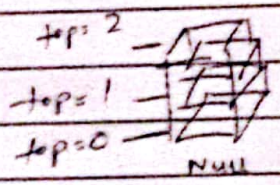


∴ pushed

(if top > max size)
3 > 5
no
if top was 5
so we wouldn't push

top != -1

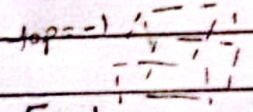
IsEmpty



Not empty

return false

if (top == -1)



Empty return true

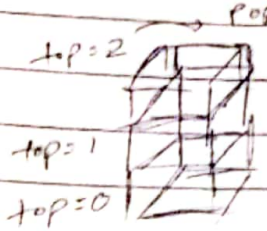
Ex 2.4 (Q1 - Q38)

Q1)

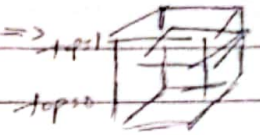
Pop

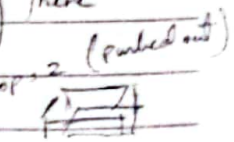
↳ true // empty array

if (IsEmpty())
∴ can't pop // can't pop out with if there's nothing there



top--





Peek

↳ true // empty array

if (IsEmpty())
∴ can't peek // can't look at with if there's nothing there

return [top] → return 5

Q3)

Graphs — Has vertices & maintain an adjacency list

adjacency list : array [vertices]

Adding edges

A → B . Directed .

adjlist [a][b] = 1

	A	B
A	0	1
B	0	0

A – B : Undirected :

adjlist [a][b] = 1

adjlist [b][a] = 1

	A	B
A	0	1
B	1	0

DFSUtil (v, visited)

visited array for all vertices made

vertice passed in function

visited [vertice] = true ← that vertice is visited

for (int i = 0; i < vertice; i++) ← from 1st vertice to this vertice

if (adjlist [v][i] && !visited[i])

DFSUtil (i, visited)

passed

Not 0 (there is edge btw v & i)

DFS (start)

DFSUtil (start, visited)

∴ visited array made in this scope

Q5)

BFS (start)

∴ queue made for all vertices ∴ visited array for all vertices

visited array at start index visited = true

while (front != rear)

front/rear

push start in queue (queue [rear++]) [start → start]

v = queue [front++] front = rear

for (int i = 0; i < V; i++) → if v & i have edge btw

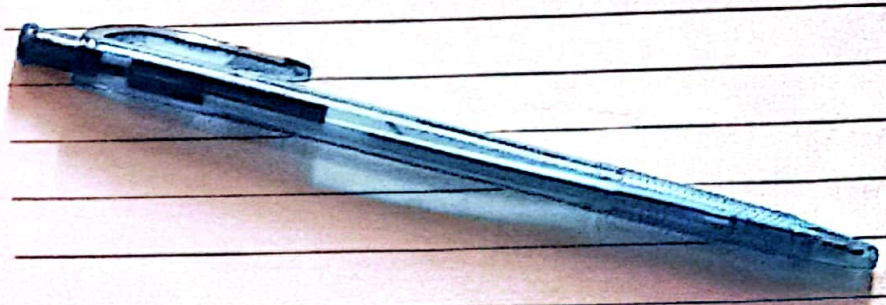
if (adjlist [v][i] && !visited) → not visited

visited [i] = true → visited now

queue [rear++] = i → queue pushed [i] current vertice

delete visited [for scope]

delete queue [for scope]



i replaces v

Date: M T W Y F S S

8) topologicalUtil (v, visited, stack, index)

for (int i = 0; i < v; i++)

if (adjlist[v][i] && !visited[i])

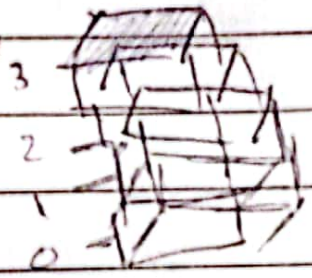
topologicalUtil (i, visited, stack, index);

∴ visited array at v = true

stack[index++] = v ;

3 ← pink concept

∴ stack used with array concept



topologicalSort ()

index = 0

for (int i = 0; i < v; i++)

if (!visited[i])

topologicalUtil (i, visited, stack, index)

∴ visited, stack, index made for this function scope

} stack now has sorted (topologically) values

← pop

print (stack[index--])

Q9) Hashing

class HashNode — key, nextNode

class HashTable — bucketArray, numOfBuckets

↑ HashNode type

hashFunction (k -> key)

return (k % numOfBuckets) // can change the question

insert (k -> key)

index = hashFunction(k)

Node = new HashNode(k) ← constructor parameterized

if (bucketArray[index] == NULL)

{ bucketArray[index] = Node } // space for the Node

↑ inserted

else // no space

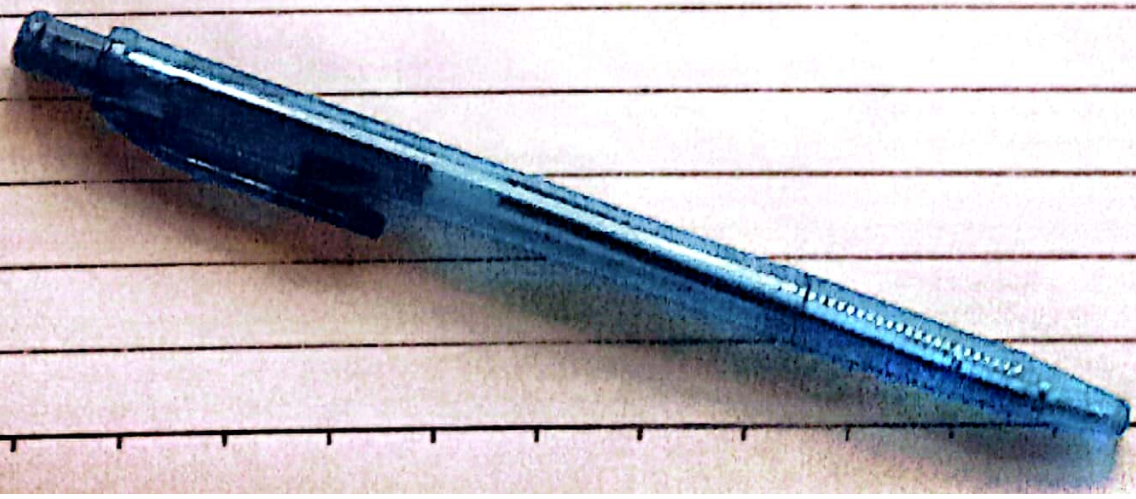
// traversal to find Space // LinkedList style

tempNode = bucketArray[index]

while (tempNode -> nextNode != NULL) // until Space found

{ tempNode -> nextNode // traversed } out of loop

tempNode -> nextNode = Node ← inserted



Date
M|T|W|T|F|S|S

So) ← val to find

search (k)

index = hashfunction(k)

Hashnode * current = bucketarray[index]

// Traversal while (current.key != NULL)

{ if (current.key == k)

{{ found }}

{{ → out of loop

Not found

delete (~~val~~ k) ← val to delete

index = hashfunction(k)

Hashnode * current = bucketarray[index]

// Traversal

// same as search while (current != NULL)

{ if (current.key == k)

{ // deletion linked list

Points at current ← temp = current

current → next // current now at next node

delete temp

{ ← word not found if out of loop

Delete Unsuccessful

8(17)

2) 1) estacks node mein node = customer name
array class = arrayorder

2) equeues main orders push
when procesed, dequeue

3) search in stack by popping & pushing

```
minHeapify(arr, size, i)
large = i
left = 2*i+1
right = 2*i+2
if (left < large && arr[left] < arr[large])
    large = left arr[left]
```

left

right

same but right replace left

```
if (large != i)
    swap(arr[i], arr[large])
    minHeapify(arr, size, large)
```

Q(2)

constructMax (arr, size)

int nonleaf = (size/2) - 1 // n/2 - 1

for (i = n/2 - 1; i >= 0; i--)

for (i = nonleaf; i >= 0; i--)

maxHeapify (arr, size, i)

constructMin (arr, size)

nonleaf = size/2 + 1 // n/2 + 1

for (i = nonleaf; i >= 0; i--)

minHeapify (arr, size, i)

maxDelete (arr, size, num) ← target

{ i = 0,

for (i = 0; i < size; i++) ← Linear Search

if (arr[i] == num) break

swap (arr[i], arr[size - 1])

same for min except minHeapify

Send to last index

size = size - 1 // n

maxHeapify (arr, size, 0) ← start again