Notes | Appointment time

Trees

- ADT of tree

AbstractDataType Tree
{
instances: it is non empty set of nodes simulating hierarchical tree structure with root at top. A non empty set of linked nodes.
starting at root node, each node is data structure consisting of value together with list of references to nodes (i.e. childerns), no references are duplicated and none point to root.

operations:
- insert(value) insert a new node as a childern of some parent node but if tree ~~was~~ is empty then as a root node
- delete() - delete node & reattach its subtree to some node.
- traverse() - starting from root node to specific node or whole tree, display value of each node
- search() - search a specific value by traversing in tree.

}

time Appointment Notes

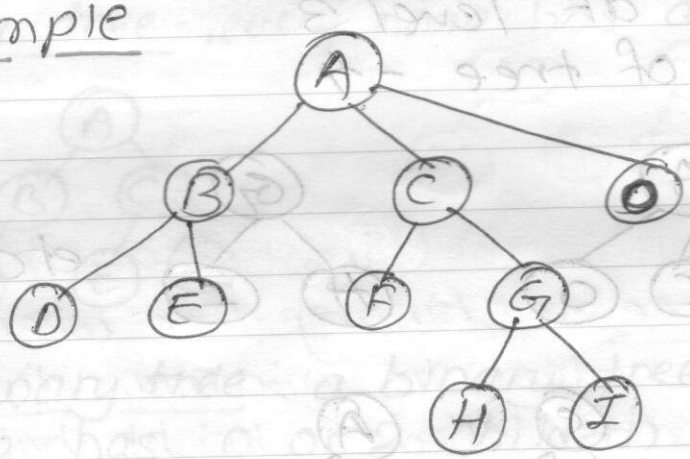
A tree is made up
- node - consist of value
- edges/~~vertices~~ - connects 2 nodes w/o creating cycle.
- parent node - node with outgoing edges
- child node - node with incoming edge
- root node - special node with no incoming edge, top of tree, start of tree
- siblings - nodes with same parent
- descendant - a node reachable by repeated traversal from parent to child
- ancestor - a node reachable by repeated traversal from child to parent

- leaf - node with no childern
- internal node - a node with atleast one child
- external node - a node with no childern
- indegree - no of incoming edges
- outdegree - no of outgoing edges
- degree of node - outdegree/no of childerns
- path - sequence of nodes & edges connecting two nodes
- degree of tree - max(degree of all nodes)
- level - starting with root at level 0 its childern at level 1 and so on

Notes | Appointment time

- height / depth of tree - max level + 1
- subtree - part of a tree
- forest - collection of disjoint trees

- example



node - A, B, C, ..., I
edges - AB, AC, BD, BE, CF, CG, GH, GI, AD
parent node - A, B, C, G
child node - B, C, D, E, F, G, H, I, D
root node - A
siblings - {B, C, D}, {D, E}, {F, G}, {H, I}
descendant - E is descendant of A
ancestor - A is ancestor of E
leaf nodes - D, E, H, I, F, D
internal nodes - A, B, C, G
external nodes - D, E, F, H, I, D
in degree - G has 1 as indegree
A has 0 as indegree
out degree - A has 3 as outdegree
G has 2 as outdegree
degree of node - G node has 2 as degree
degree of tree - 3

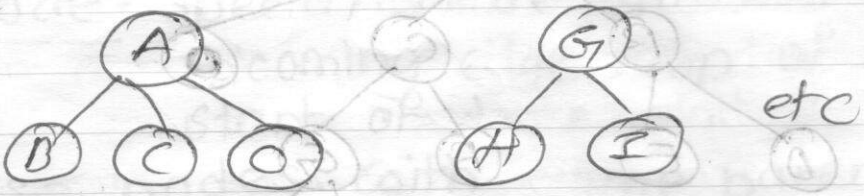
time | Appointment | Notes

path- A-B-E, A-C-G-I, C-G-H

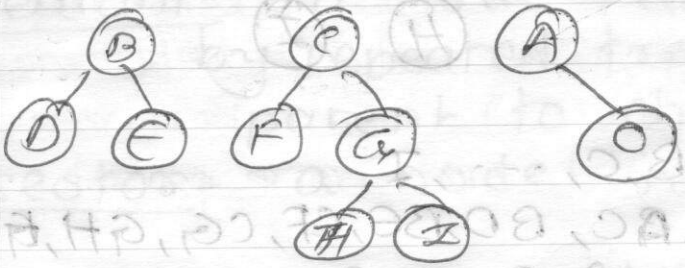
level - A at level 0
B, C, D at level 1
D, E, F, G as level 2
H, I at level 3

height/depth of tree - 4

subtree

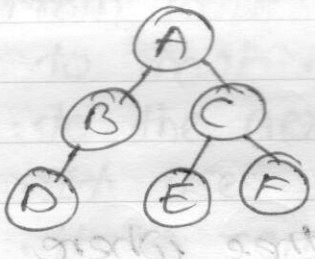


forest:

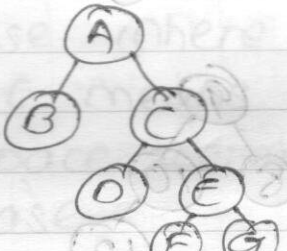


Notes | Appointment time

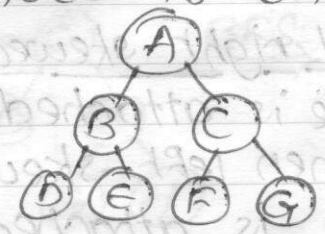
- Types of binary tree: perfect binary tree, full binary tree, almost perfect binary tree, binary search tree, balanced tree, expression tree
- Binary tree - a tree where every node has at most 2 children



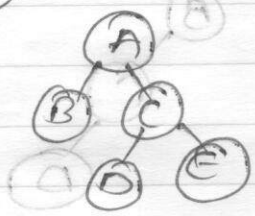
- full binary tree - a binary tree where every node has 0 or 2 children



- perfect binary tree - a full binary tree where every leaf node is on same depth level

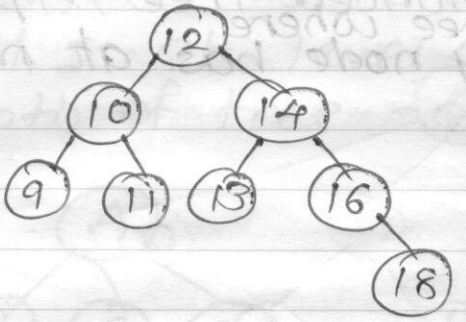


- almost perfect binary tree - a full binary tree where every leaf node is at same depth or (height - 1)

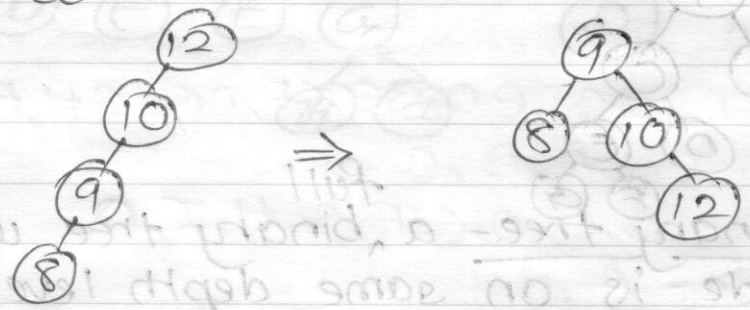


time | Appointment | Notes

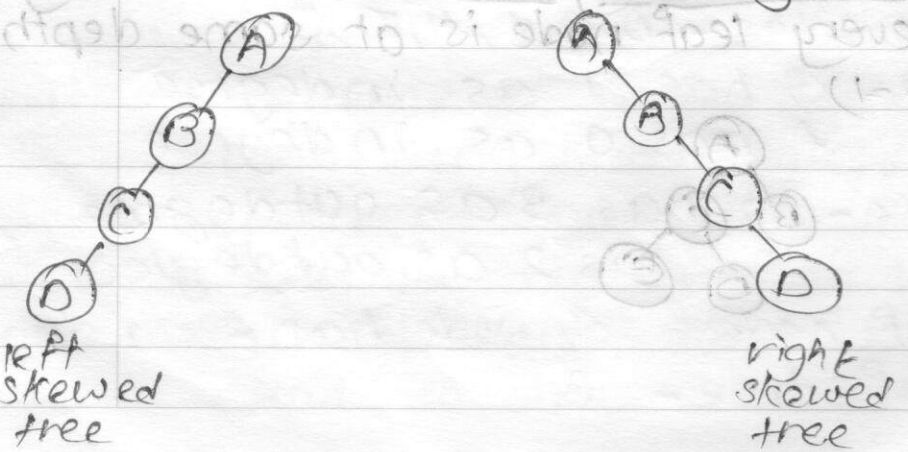
- binary search tree - a ~~[illegible]~~ binary tree where for every node
  left node ≤ parent node ≤ right node



- balanced tree - a binary tree where ~~[illegible]~~ leaf nodes are minimum depth possible, also called as height balanced tree



- ~~[illegible]~~ left and right skewed tree
the tree in which each node is attached as a left child of parent node then left skewed tree.
The tree in which each node is attached as a right child of parent node then right skewed tree.

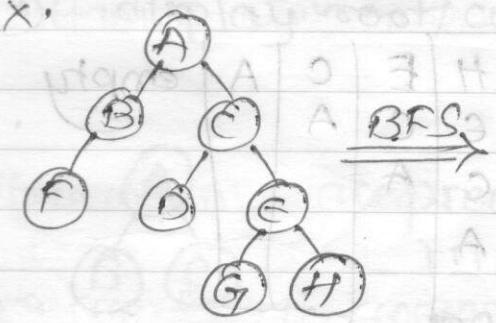


- Types of tree traversal / searching techniques

BFS : DFS

- **BFS** - Breadth first search
  - traversal in tree technique where it starts from root node, explores the node closer to root node (childs of root), then to the next level and so on.
  - It uses queue for traversal node storing
  - it is complete searching technique, means it will find a node in infinite tree
  - time complexity is $O(V+E)$ in worst case where V means total vertices & E means total edges.
  - space complexity is $O(V)$ in worst case

ex.



BFS

A-B-C-F-D-E-G-H
BFS traversal

queue

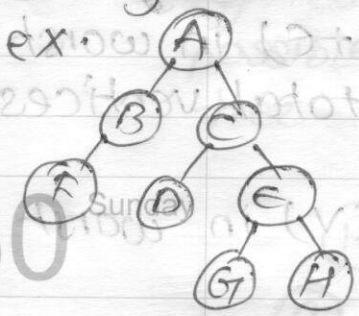
start
A
BC
CF
FDE
DE
E
GH
H
stop

time Appointment Notes

DFS - Depth first search

- it is a searching technique on tree.
- explores node farthest from root node first and so on by backtracking
- uses stack for storing intermediate nodes
- it is not complete for infinite tree where it can go on in corong part of tree (where goal node is not present)
- hence uses another variation iterative deepning DFS
- good ~~when~~ for limited depth of tree.

ex.



DFS ⇒

stack

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

DFS traversal

F-B-D-G-H-E-C-A

G	H	E	C	A	empty
H	E	C	A		
E	C	A			
C	A				
A					

Notes | Appointment time

- Types of binary tree traversal
  inorder, preorder, post order

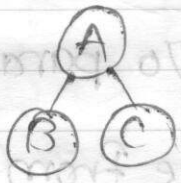
preorder → used to create expression tree, to create duplicate of tree

preorder current
1) display root element value
2) call preorder on left subtree recursively
3) call preorder on right subtree recursively

inorder → used in BST
1) call inorder on left subtree recursively
2) display root/current element value
3) call inorder on right subtree recursively

postorder → to delete node from tree
1) call postorder on left subtree recursively
2) call postorder on right subtree recursively
3) display root/current element value.

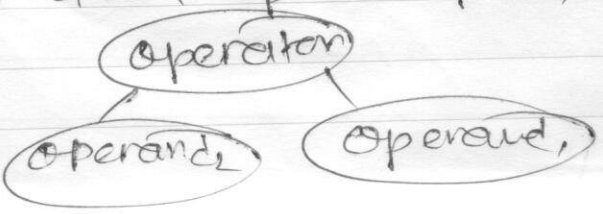
ex.



preorder → A-B-C
inorder — B-A-C
postorder → B-C-A

time Appointments / notes

- (Application of tree)
- Expression tree
  - a binary tree where internal node is operator and leaf node is operand.
  - it is used to evaluate an expression
  - it is also used to convert ~~between~~ among infix, postfix, prefix expression
    - inorder ~~infix~~ traversal)
      - + gives infix expression
      - + every subtree is infix expression
      - + put paranthesis around every expression
      - + opening paranthesis at start of each subtree, closing at end of processing all children
    - postorder ~~postfix~~ traversal)
      - gives postfix expression w/o paranthesis
    - preorder ~~prefix~~ traversal)
      - gives prefix expression w/o paranthesis
- How to construct expression tree from expression
  - take infix/prefix → convert to postfix expression
  - ~~if~~ ~~op~~ append · at end
  - if operand push
  - if operator then pop operand, $operand_2$ from stack top and push ~~as~~ subtree as



Notes | Appointment time

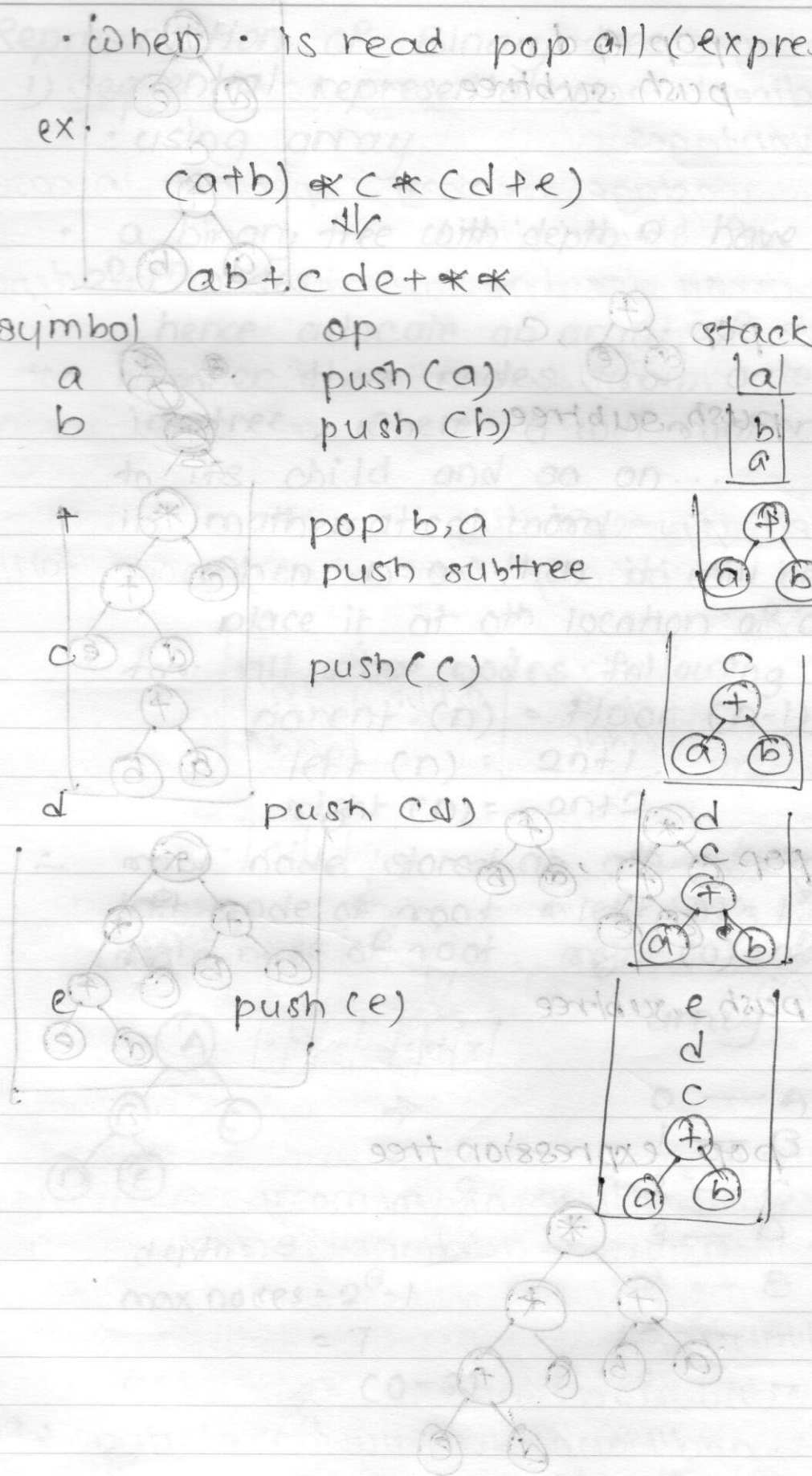
- when) is read pop all (expression tree)

ex.

(a+b) * c * (d+e)

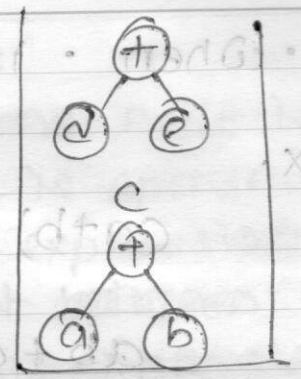
ab+c de+**

	symbol	op	stack
1)	a	push(a)	a
2)	b	push(b)	b, a
3)	+	pop b,a push subtree	+ (a, b)
4)	c	push(c)	c, + (a, b)
5)	d	push(d)	d, c, + (a, b)
6)	e	push(e)	e, d, c, + (a, b)

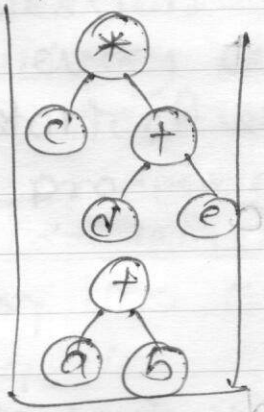


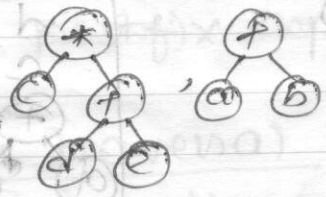
time Appointment Notes

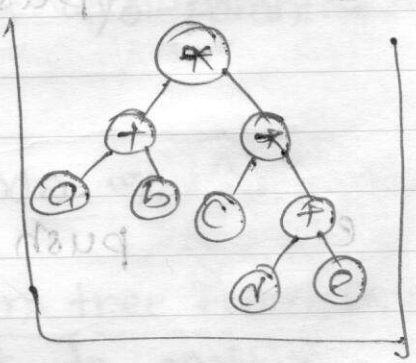
7) + pop e, d
push subtree



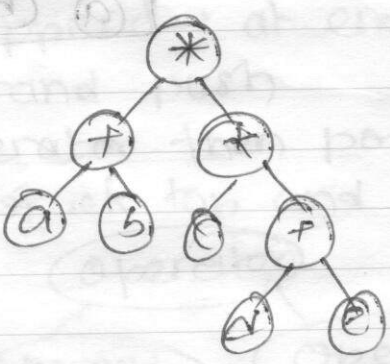
8) * pop  , c
push subtree



9) * pop 
push subtree



10) . pop expression tree



Notes | Appointment time

- Representation of Binary tree

1) sequential representation

- using array

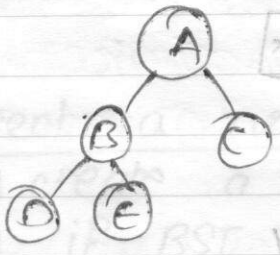
- a binary tree with depth n have maximum $2^n - 1$ nodes
- hence allocate an array of size $2^n - 1$
- number the nodes (from 0 to $(2^n - 1) - 1$) in tree, where 0 will root node, then to its child and so on...
- in mathematical word
  when n=0 then it will be root node
  place it at 0th location of an array
- for all nodes following is true

  parent (n) = floor (n-1)/2

  left (n) = 2n+1.

  right (n) = 2n+2

∴ root node stored at 0th location
left node of root left (0) = 1st location
right node of root right (0) = 2nd location



depth = 3

max nodes = $2^3 - 1$

= 7

= (0-6)

array

0 — A
1 — B
2 — C
3 — D
4 — E
5 —
6 —

time Appointment Notes

advantages

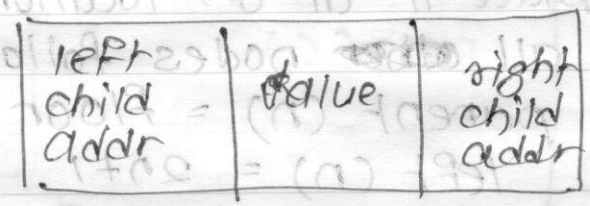
- direct access to any node,

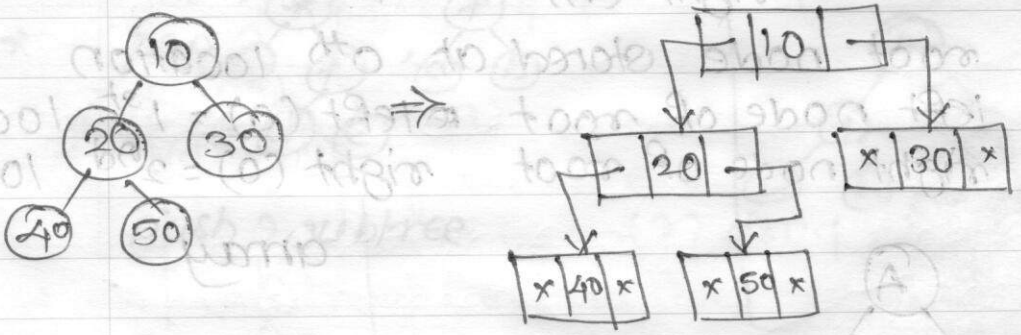
disadvantages

- wastage memory (specially in case of skewed tree)
- array size has to be decide first depending on depth
- insertion & deletion of node is not easy

2) linked representation

In a binary tree every node will have





advantages

- no wastage of memory
- insertion & deletion is easy

disadvantages

- no direct access
- additional memory for storing left node & right node addr & per node.

Notes | Appointment time

- Application of tree
- Binary search tree

Binary search algorithm

where left subtree < root node < right subtree
value

i.e. left node < parent node < right node

valid for every node

AbstractDataType BST

{

instances: it is a binary tree

where

left node < parent node < right node

operations:

insert() - add node in binary tree without breaking BST rule

delete() - delete a node and rearrange BST

search() - search as per binary search algo

display() - display tree node's values.

}

Insert a new node

1) create a 'new' node
2) if BST is empty then root = new
3) else
   i) 'current' node = root
   ii) if new < current
       then if left(current) != null
            then current = left(current)
                 go to step (ii)
       else left(current) = new

time Appointment Notes

else

iii) if new > current
then if right(current) != null
then current = right(current)
go to step (ii)
else right(current) = new

Delete a node

case 1) delete a leaf node
make left/right pointer of parent node as null

inorder successor (n) - perform inorder traversal on node n, first value outputed will be inorder successor

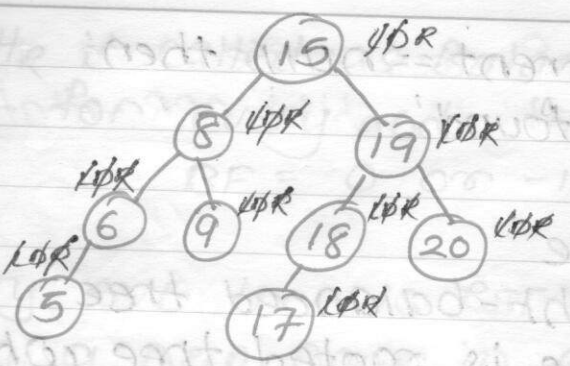
if a node have right subtree then inorder successor will leftmost node in right subtree (min value in right subtree)

if a node dont have right subtree then inorder successor will be nearest ancestor for which given node will be in left subtree

Notes

Appointment time

ex.



inorder traversal → 5-6-8-9-15-17-18-19-20

inorder traversal on BST gives sorted order list

inorder successor → next node in inorder traversal

inorder successor of node 8 will be 9

inorder successor of node 6 will be 8

inorder sucessor of node 15 will be 17

inorder successor of node 20 is null

delete a node

case 2) if node to ~~be~~ delete ~~have~~ has one child then that child will be replacing deleting node

case 3) if a node d to delete have 2 childerns then inorder successor will be replacing deleting node.

Search a node n

1) current node = 'root'

2) if n = = current
then show 'found'
else if n < current
then current = left (current), go to 2
else current = right (current), go to 2

time Appointment Notes

in step 2 current == null then
show 'not found'

Application of tree

Balanced tree / height-balanced tree

a balanced tree is rooted tree where each subtree of the root has equal number of nodes. The height of such binary tree is O(log n)

various types of Balanced trees are

1) AVL tree
2) RED-Black tree
3) B-tree

AVL Tree

- Adelsion Velski Lendis tree, 1962
- it is balance cort height of subtrees
- searching of node canbe done in O(log n) time

a tree T is

- AVL tree / height balanced tree if T is non-empty & T_L, T_R are left, right subtrees resp. and T is height balanced if
  1) T_L & T_R are height balanced.
  2) $-1 \leq (H_L - H_R) \leq 1$ where H_L & H_R are height of T_L & T_R resp.

Notes | Appointment time

$H_L - H_R$ is called as Balance factor (BF)

∴ for every node in AVL tree

BF = 0 or −1 or +1

every AVL tree is BST tree

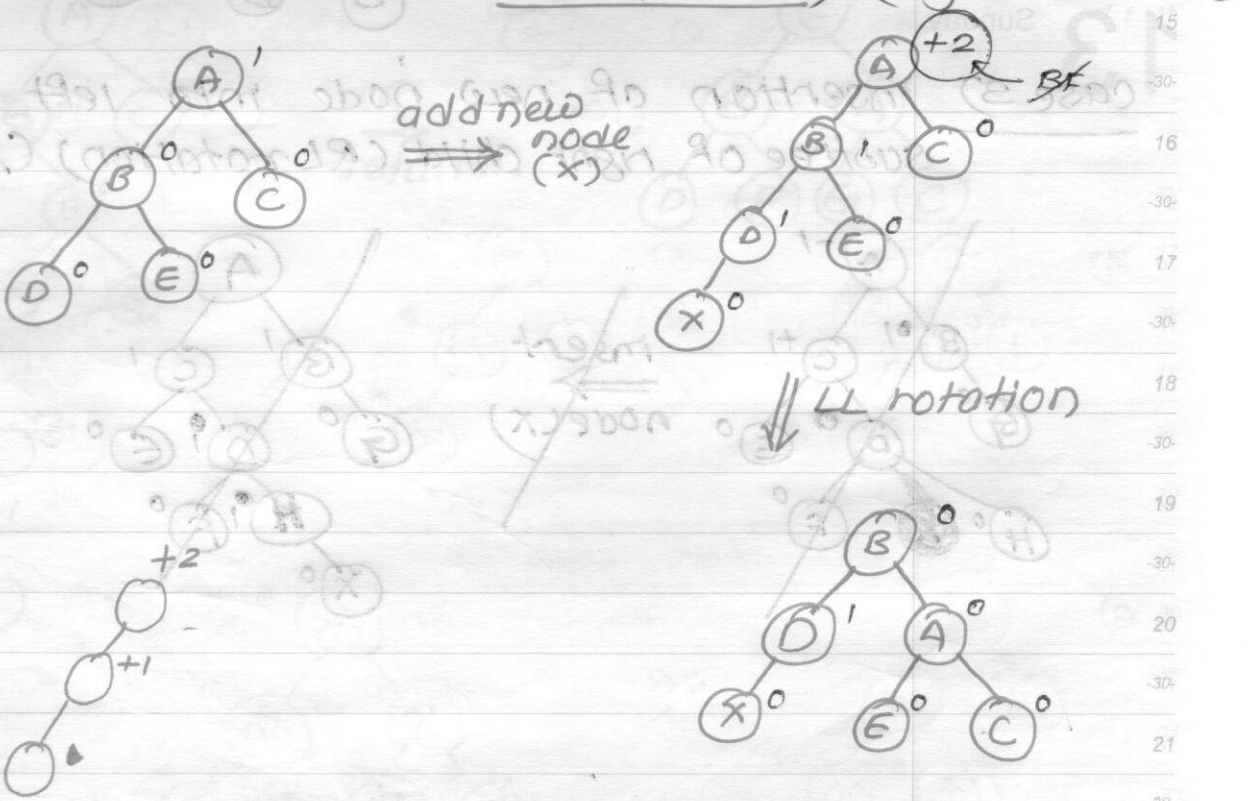
Inserting a new node in AVL tree

Insertion of a new node in AVL tree is same as BST tree

only after insertion rebalancing may req

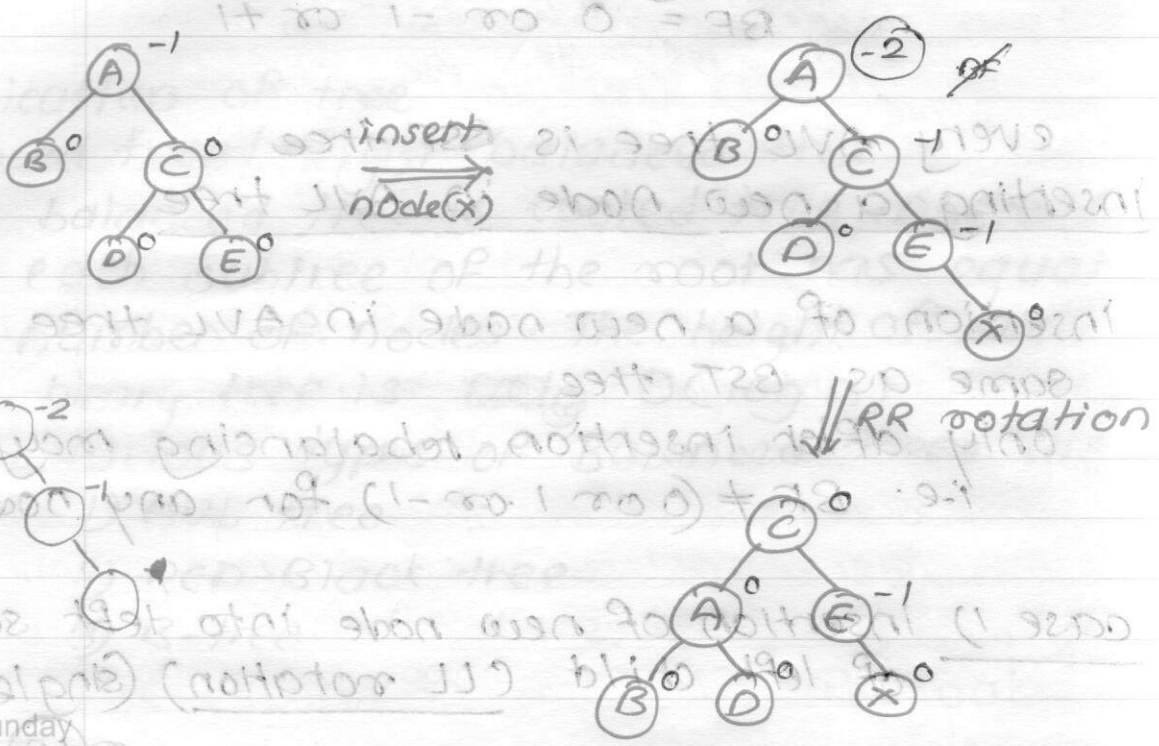
i.e. BF ≠ (0 or 1 or −1) for any node

case 1) insertion of new node into left subtree of left child (LL rotation) (single rotation)

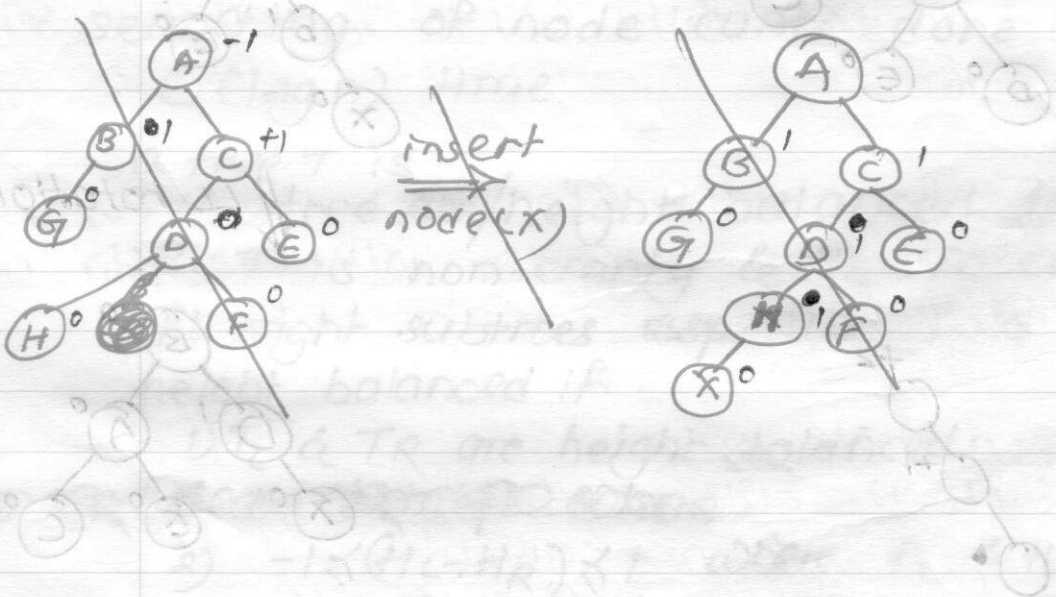


time | Appointment | Notes

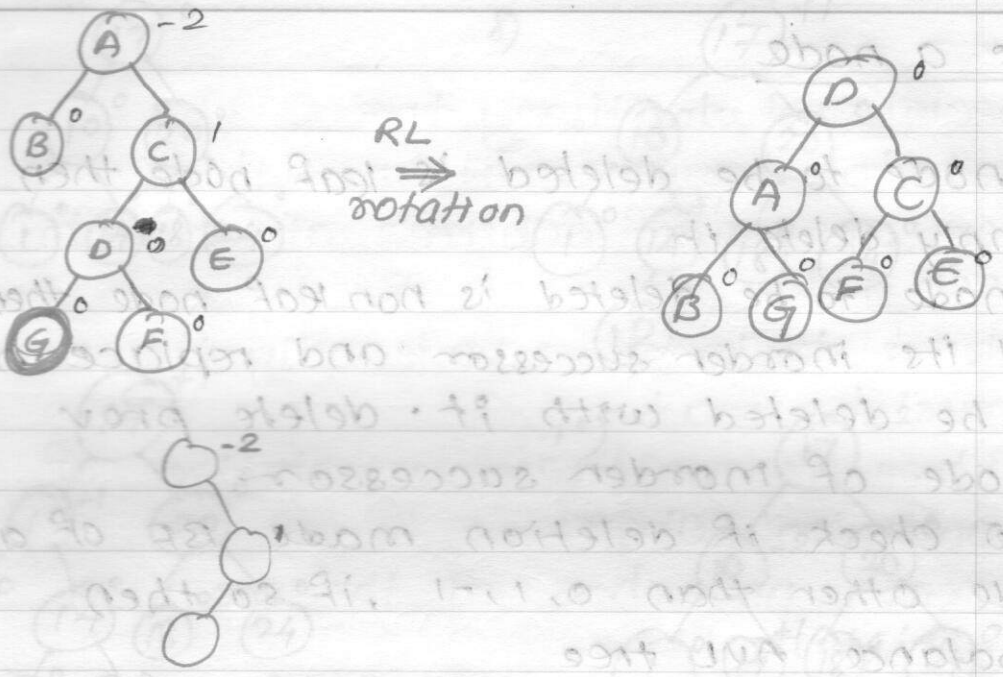
case 2) insertion of new node into right subtree of right child (RR rotation) (single rotation)



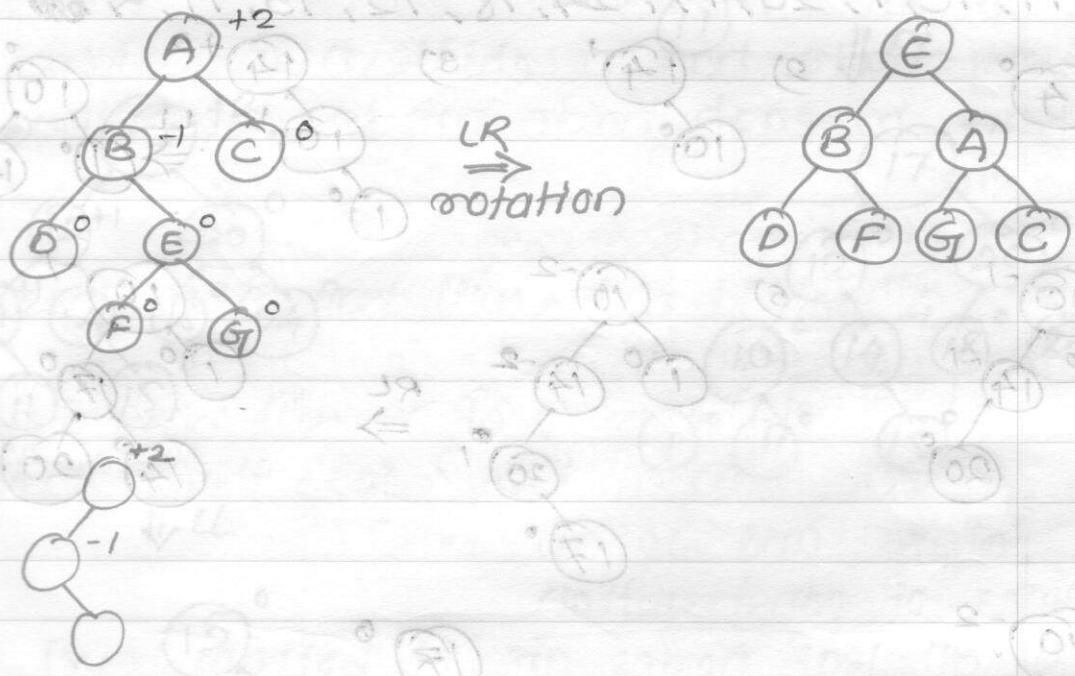
case 3) insertion of new node into left subtree of right child (RL rotation) (double rotation)



Notes | Appointment | time



Case 4) insertion of new node into right subtree of left child (LR rotation) (double rotation)



time | Appointment | Notes

delete a node

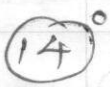
if node to be deleted is leaf node then simply delete it

if node to be deleted is non leaf node then find its inorder successor and replace node to be deleted with it. delete prev node of inorder successor

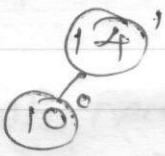
now check if deletion made BF of any node other than 0, 1, -1, if so then rebalance AVL tree

ex. construct AVL tree for
14, 10, 1, 20, 17, 24, 18, 12, 15, 11, 4

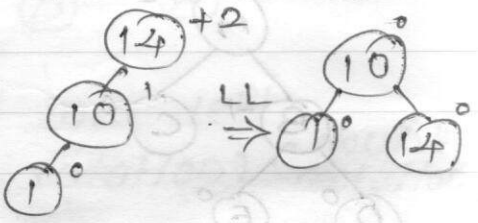
step 1)



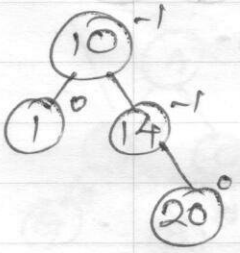
2)



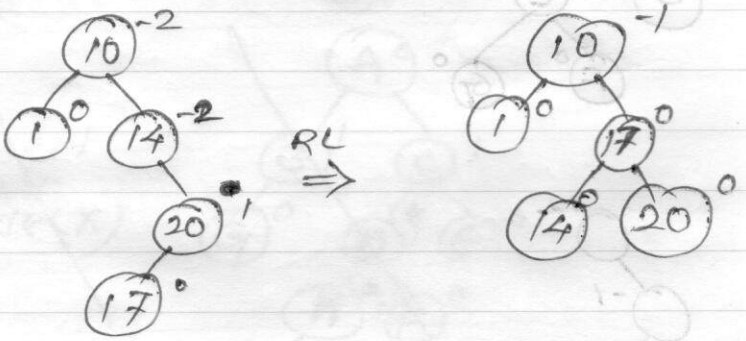
3)



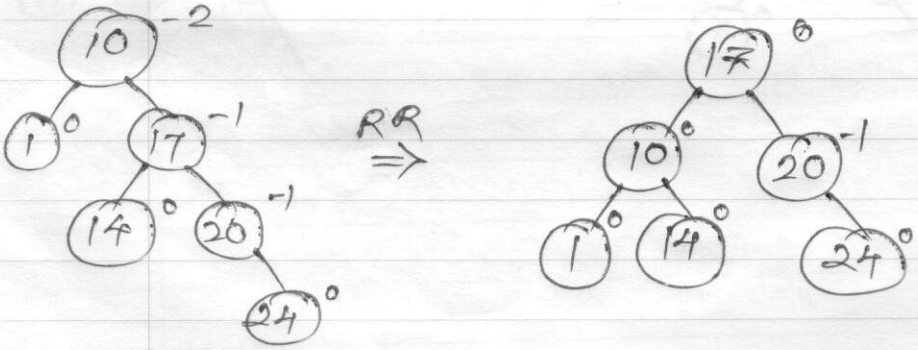
4)



5)

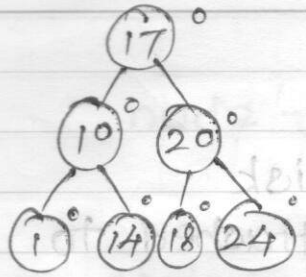


6)

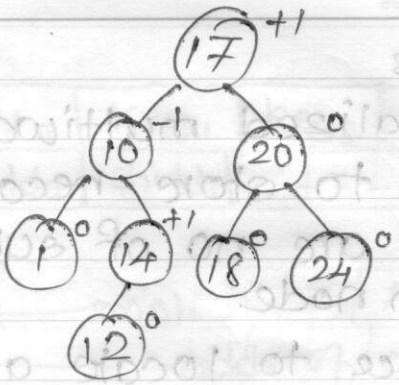


Notes | Appointment | time

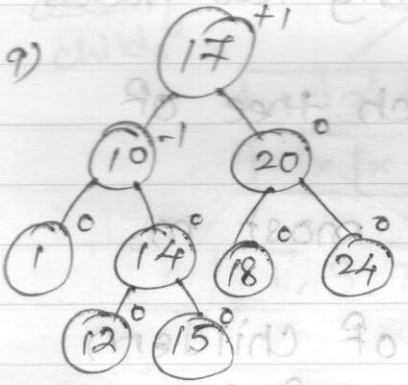
7)



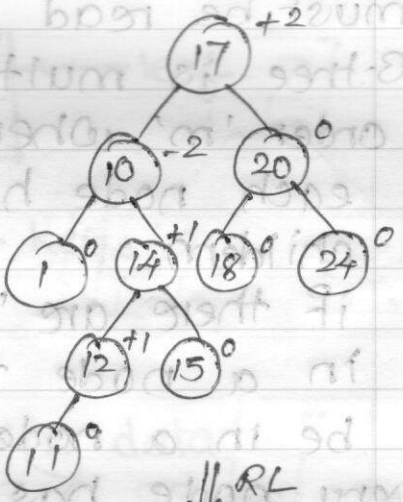
8)



9)

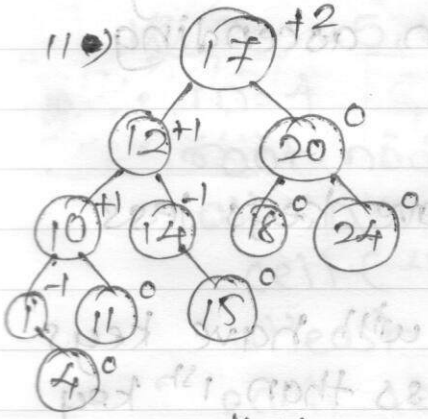


10)

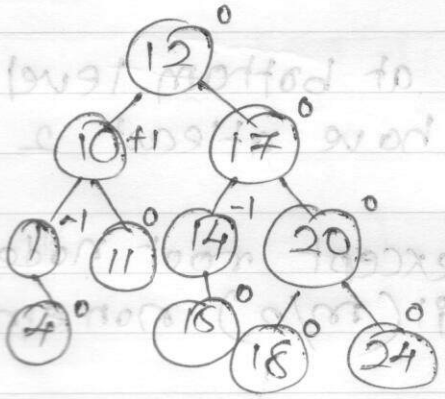


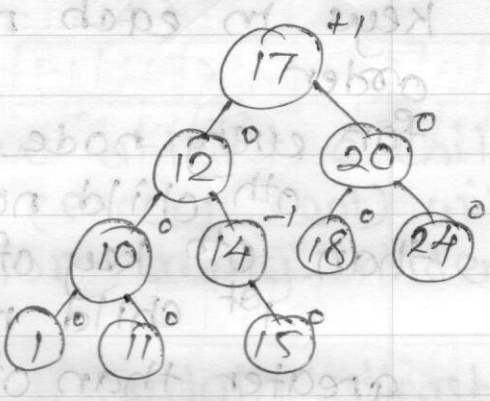
RL

11)



LL





time Appointment Notes

8 B-trees

-30- • specialized multiway tree

9 • used to store record on disk

-30- • there are no of subtrees attached to each node

10

-30- • Hence to locate a node only few nodes must be read

11

-30- • B-tree is multiway search tree of order 'm' where

12 each node has at the most m childern

-30- if there are 'n' ~~node~~ no of childern

13 in a node then (n-1) no of keys will be in a node.

-30-

14 • Every node has n childern, (n-1) keys

15 • keys in each node are in ascending order

-30-

16 • for every node x of x

-30- 0^{th} child node will have keys less

17 than 0^{th} key of x,

-30- 1^{st} child node of x will have keys

18 greater than 0^{th} key & less than 1^{st} key

-30- of x and so on

19 • Rules of construction

-30- 1) all leaf nodes are at bottom level

20 2) root node should have atleast 2 childern

-30-

21 3) all internal nodes except root node

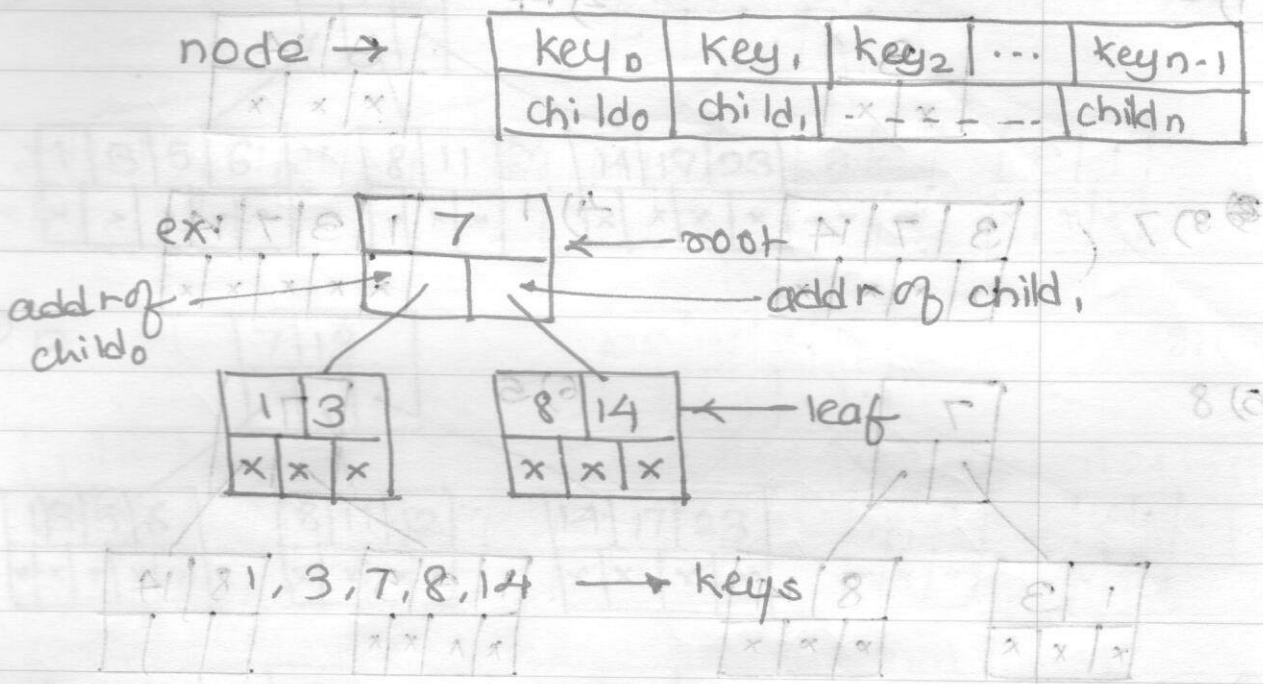
-30- have at least ceil(m/2) non empty

22 childern

-30-

4) each leaf node must contain at ...

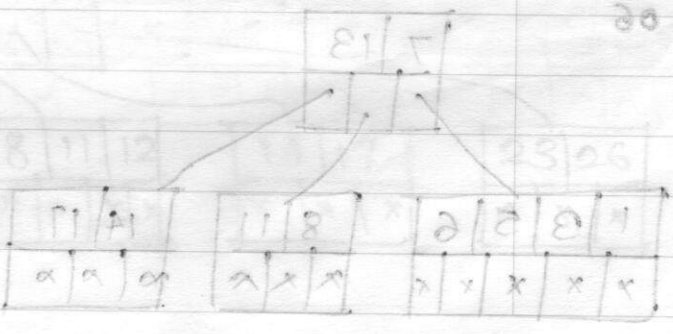
ceil (m/2) - 1 ...



Ex.

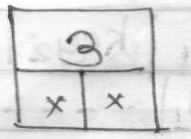
create B-tree for following data
3, 14, 7, 1, 8, 5, 11, 17, 13, 6, 23, 20, 26, 4, 16, 18, 24, 25, 19 (order = 4)

- m = 4
- root node will have atleast 2 childern
- all internal nodes (except root) will have ceil(4/2) = 2 non empty childern atleast
- each leaf node will contain atleast ceil(m/2) − 1 = 2 − 1 = 1 key

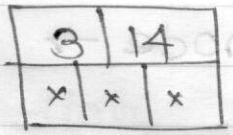


time Appointment Notes

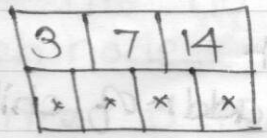
1) 3



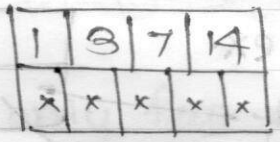
2) 14



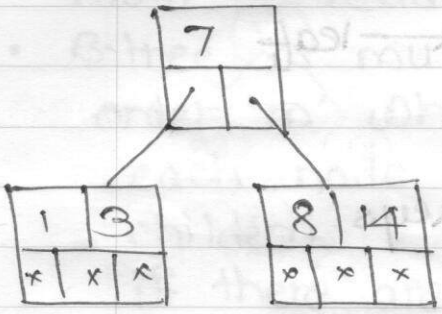
3) 7



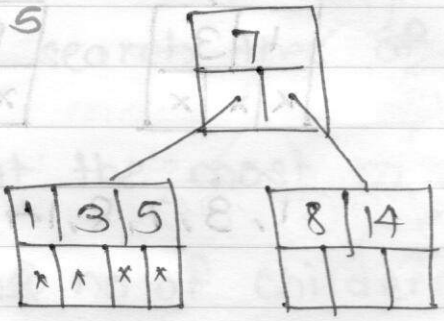
4) 1



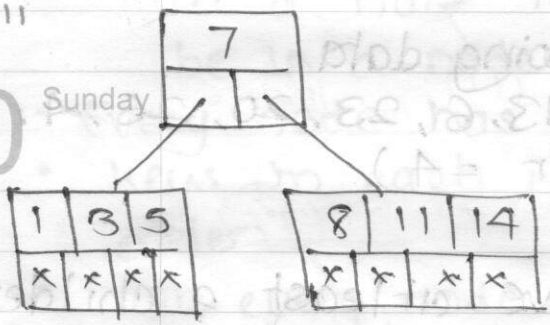
5) 8



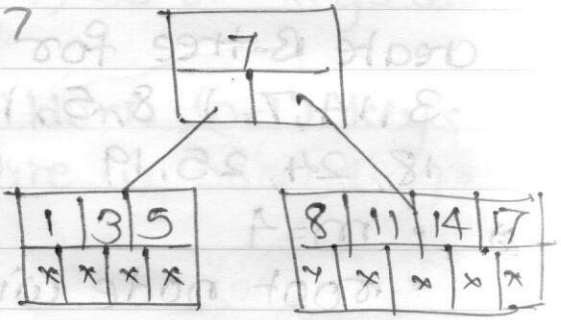
6) 5



7) 11

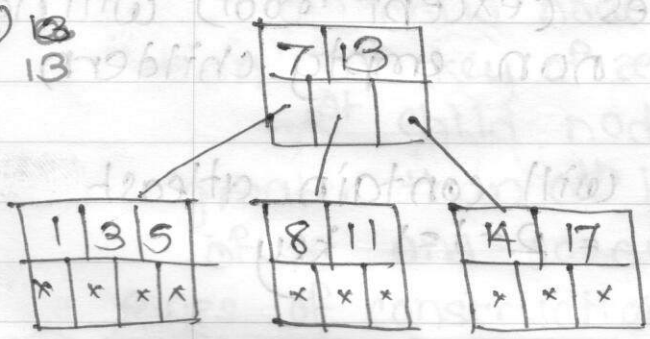


8) 17

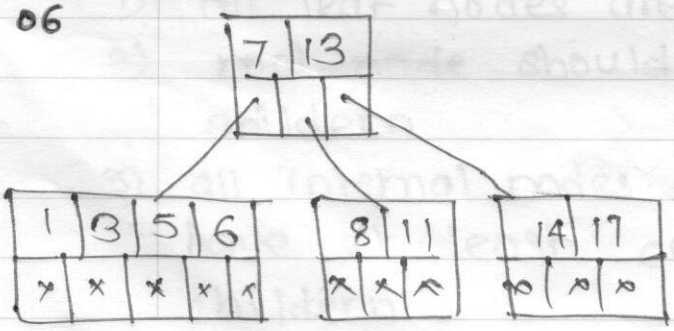


20 Sunday

9) ~~13~~ 13

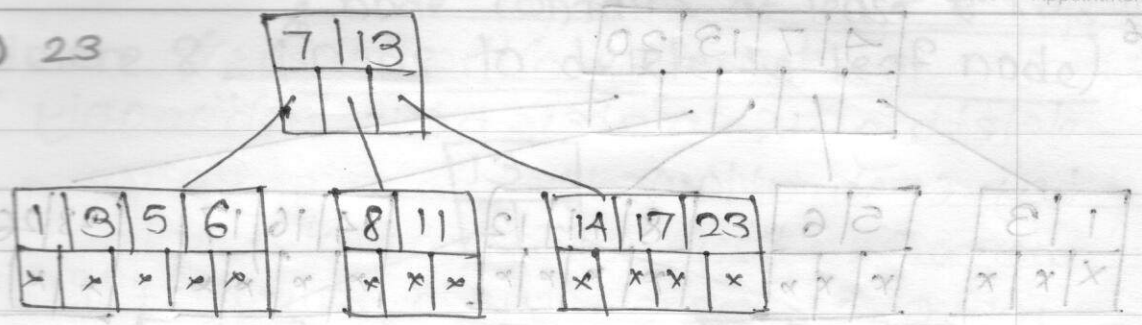


10) 06

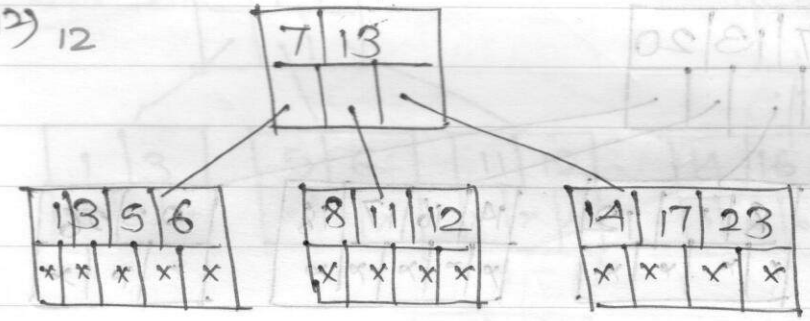


Notes | Appointment time

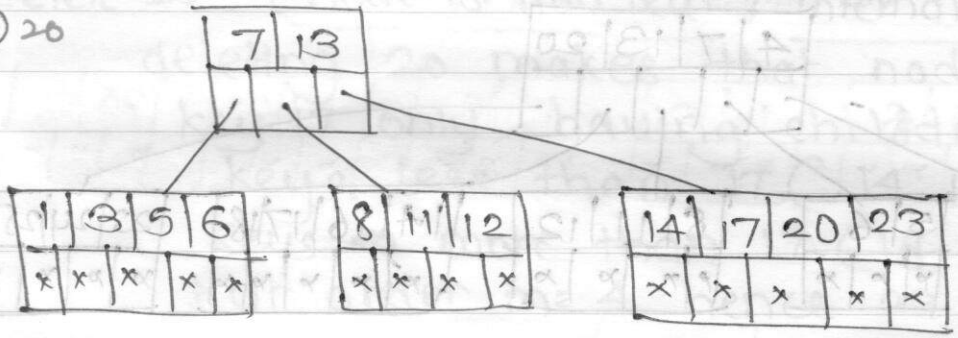
11) 23



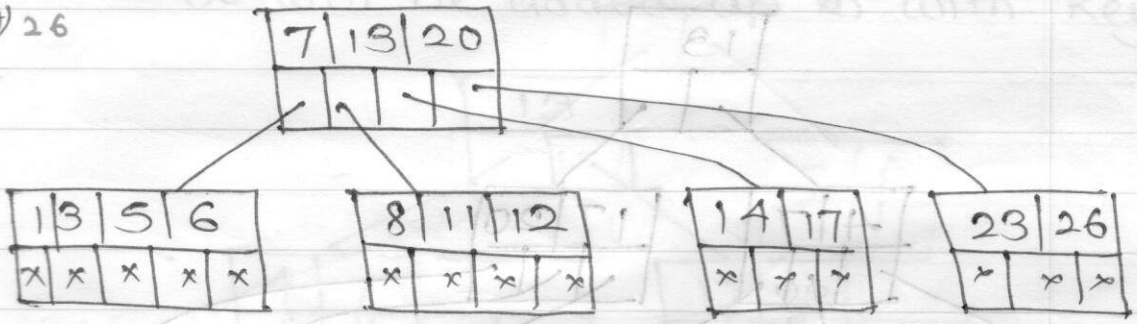
12) 12



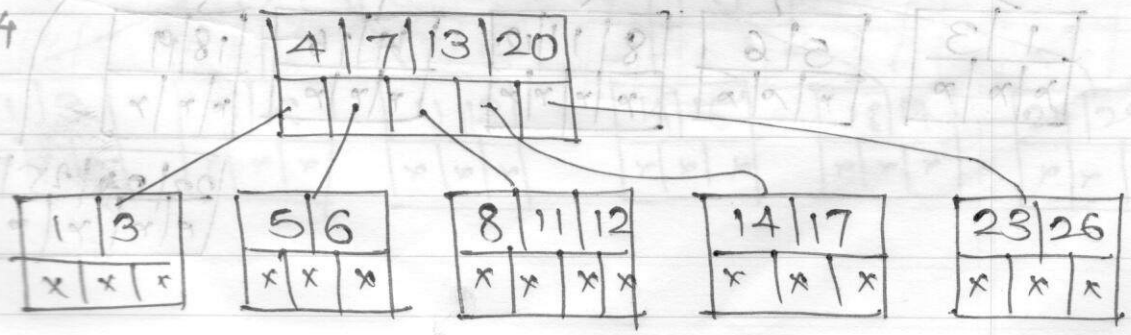
13) 20



14) 26

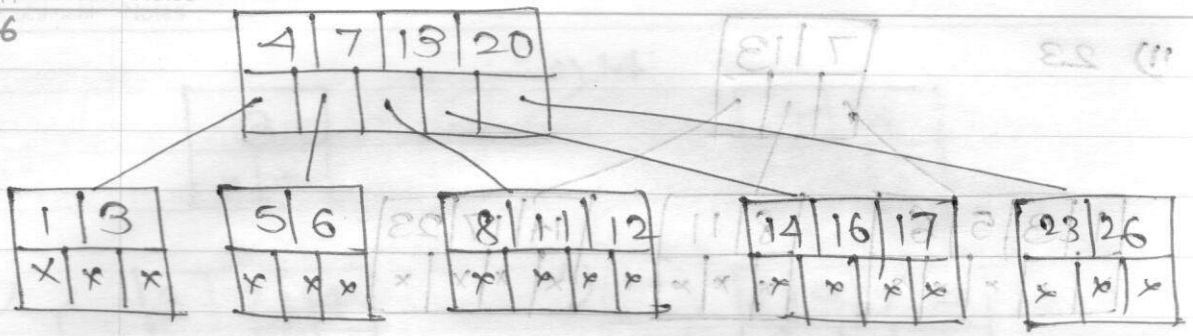


15) 04

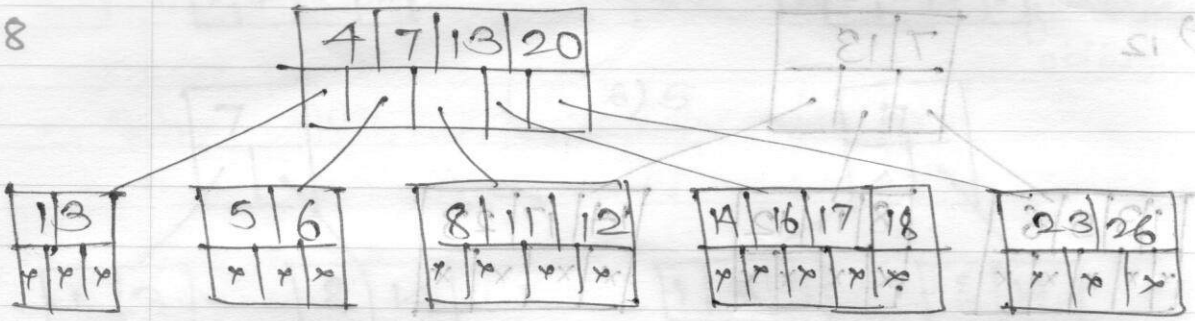


time Appointment Notes

16) 16

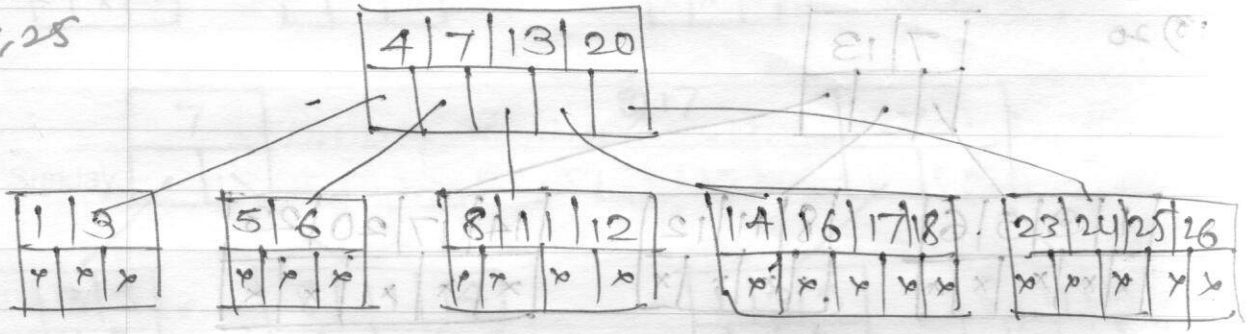


17) 18

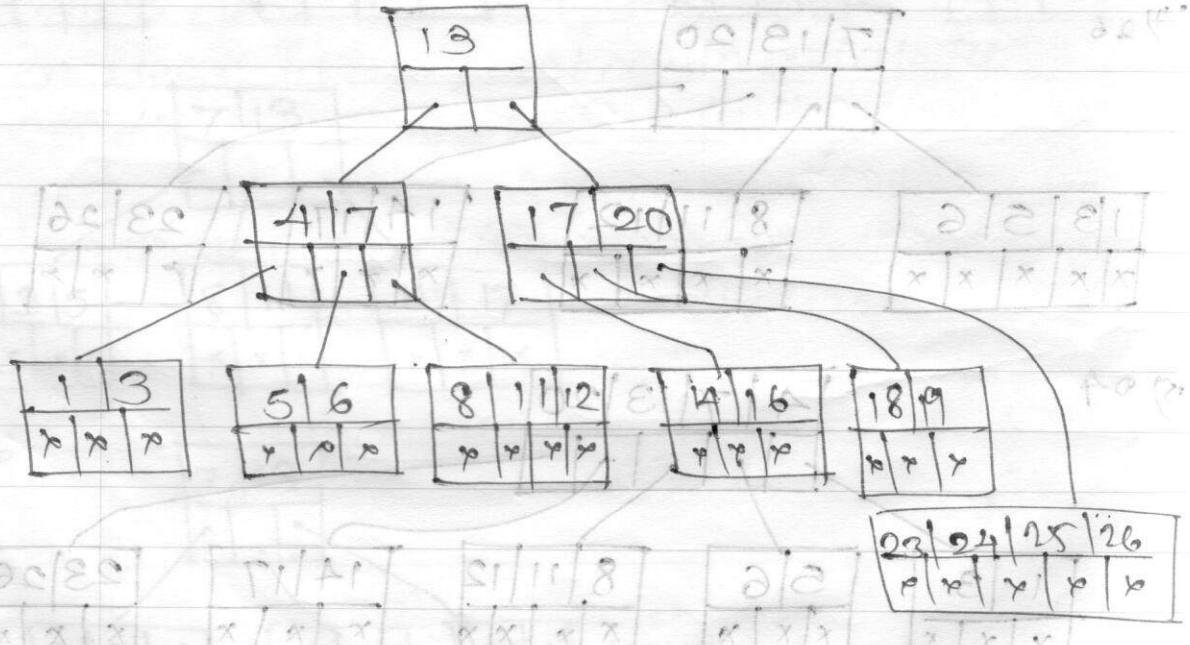


18) 24, 25

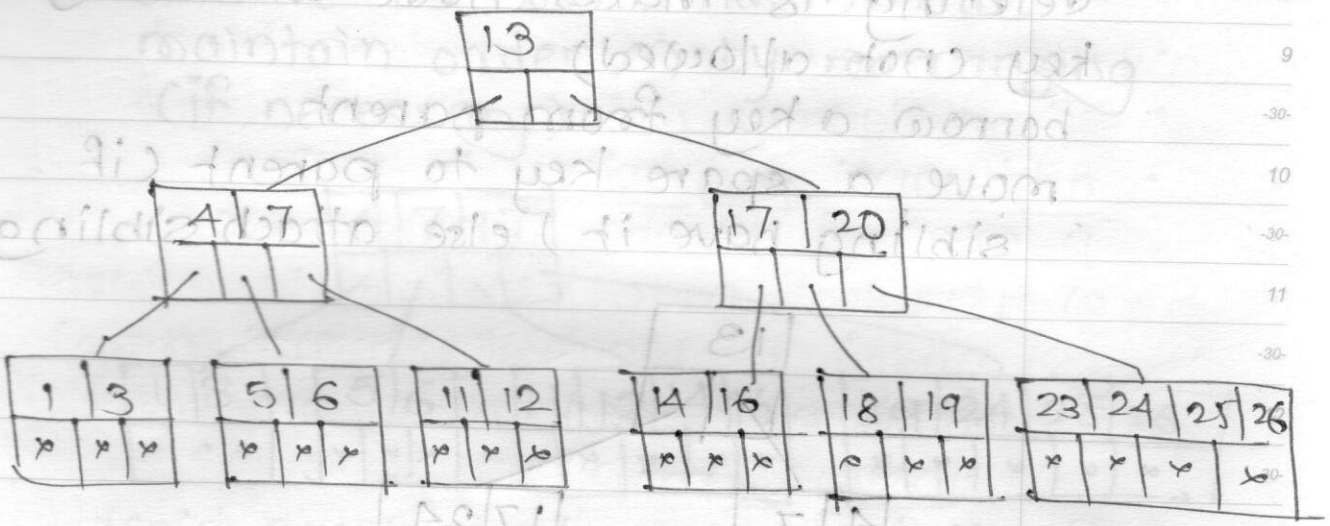
19)



20) 19

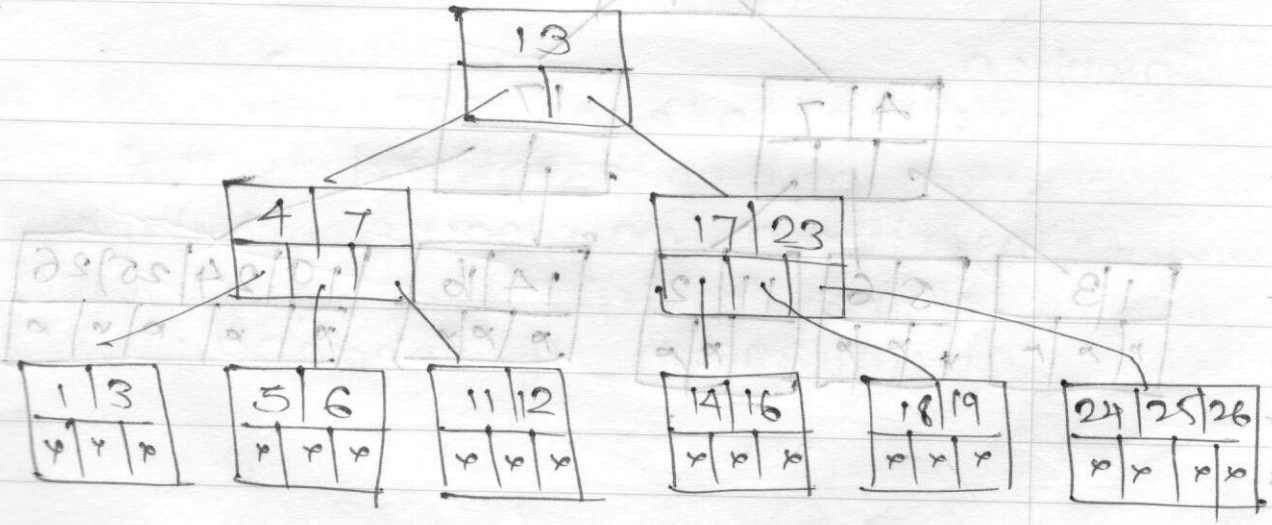


delete 8 (node contains at least 3 keys & node to delete is leaf node)



delete 20 (node is non leaf / internal node)

deleting 20 makes that node have key 17 only, having children with keys less than 17 (14, 16) and keys more than 17 (18, 19, 23, 24, 25, 26) but order m=4 hence 23 is median so will be added up with key 17



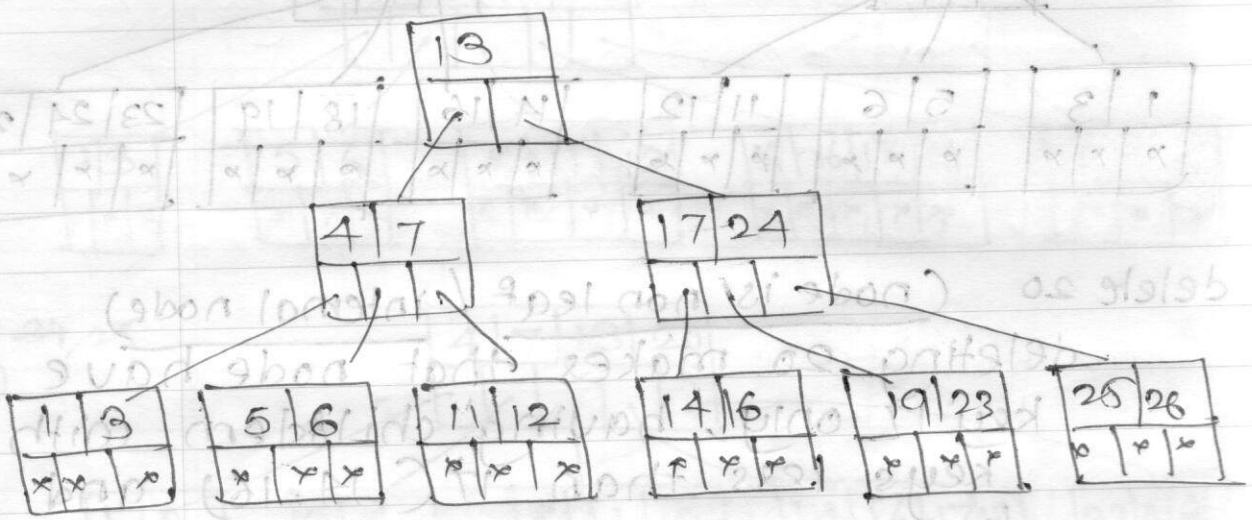
time Appointment Notes

delete 18 (leaf node with 2 keys)

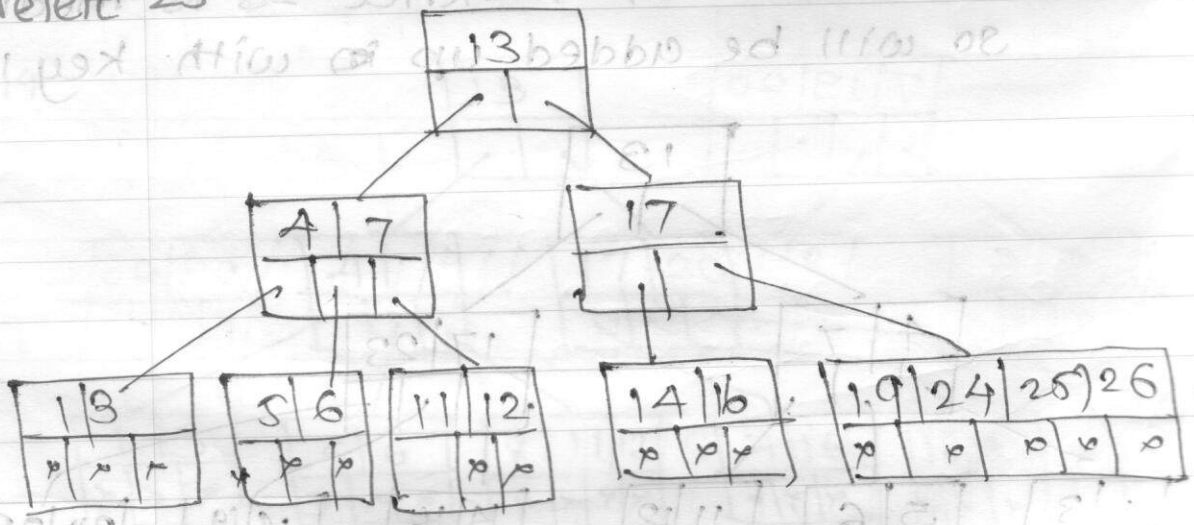
deleting 18 makes node with only 1 key (not allowed).

borrow a key from parent

move a spare key to parent (if sibling have it) else attach sibling

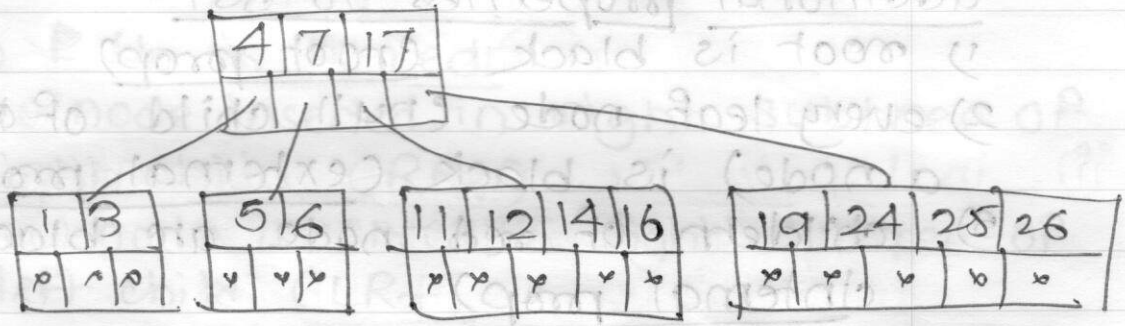


delete 23



Notes | Appointment time

delete 13 (internal/root node, have only 1 key)
combine its both children node
maintain order m by rearranging
(if necessary)



Height of B-tree : $O(\log_m n)$

time Appointment Notes

Red Black tree

- it is BST
- where every node is red or black
- self balancing BST
- additional properties to BST
  1) root is black (root prop)
  2) every leaf node (null child of a node) is black (external prop)
  3) childern of red node are black (internal prop)
  4) all the leaves have same black depth level
  5) path property - every simple path from root to descendant leaf node contains same no of black nodes.
  6) from root to external nodes there wont be two consecutive red nodes in path

Insertion

- insert a node as BST
- every new node inserted is red.
- insertion may cause imbalance, remove it depending upon type/ configuration of imbalance.

Pu = black creates no color imbalance
gu ≠ red, else tree is color imbalance before insertion itself.

Monday November 28

Notes | Appointment | time

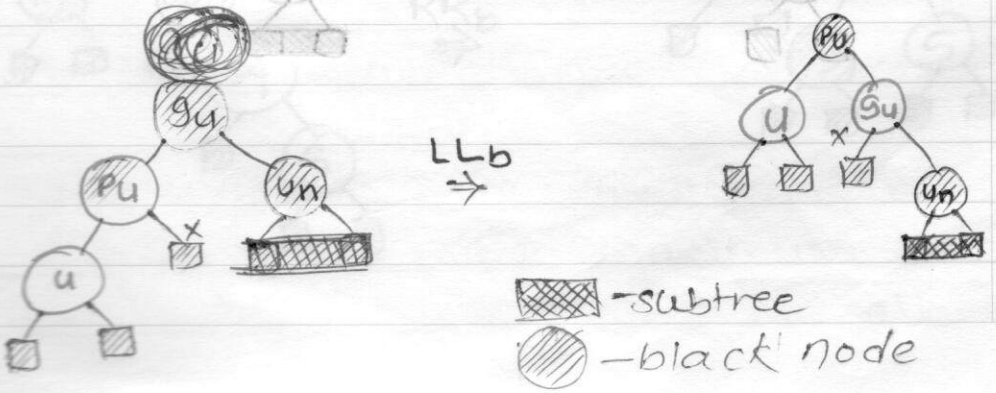
if u is newly inserted node
Pu is parent of u (red)
gu is grandparent of u, parent of Pu
Un is uncle node

A) then Un is red
i) new node inserted in right subtree of right child (RR_r)
ii) new node inserted in right subtree of left child (LR_r)
iii) new node inserted in left subtree of right child (RL_r)
iv) new node inserted in left subtree of left child (LL_r)

↓ how to solve

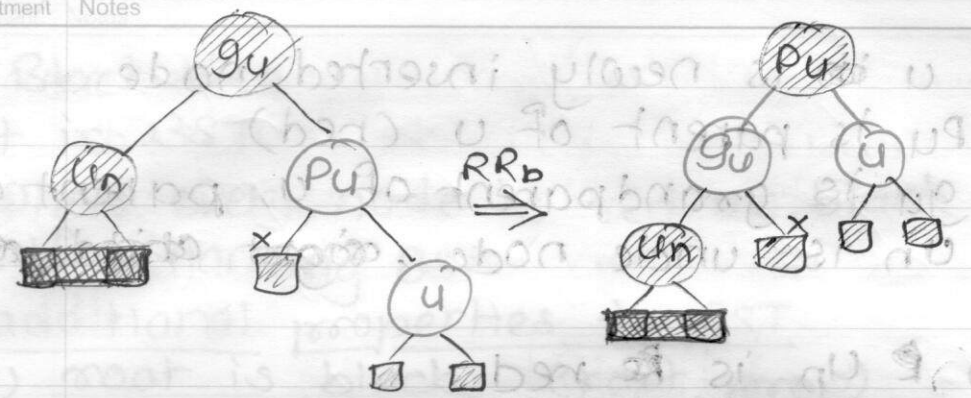
1) change color of Pu from red to black
2) change color of Un from red to black
3) change color of gu from black to red iff gu ≠ root

B) if Un is black
i) in LL_b & RR_b case
∴ apply single rotation of Pu about gu
∴ recolor pu to black, gu to red

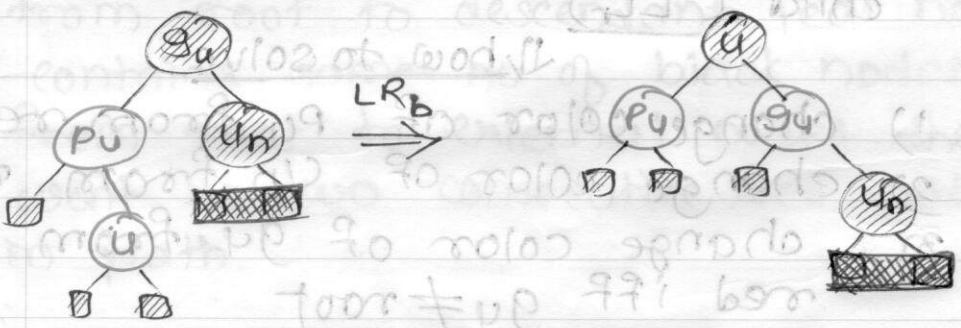


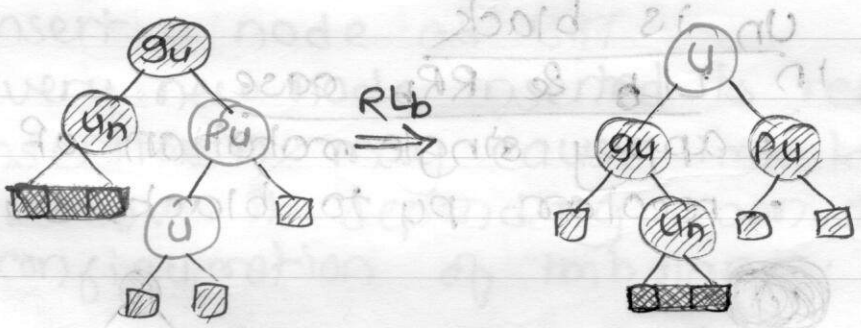
subtree
– black node
– red node

time | Appointment | Notes



ii) in LR_b & RL_b case of imbalance
- apply double rotation of u about Pu followed by u about gu
- for LR_b recolor u → black
  Pu, gu → red
- for RL_b recolor pu → black

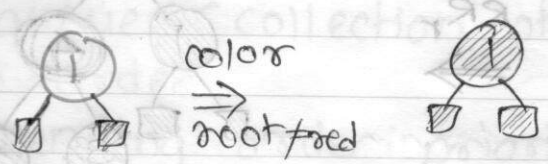




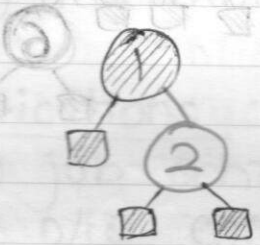
Notes | Appointment time

example insert 1,2,3,4,5,6

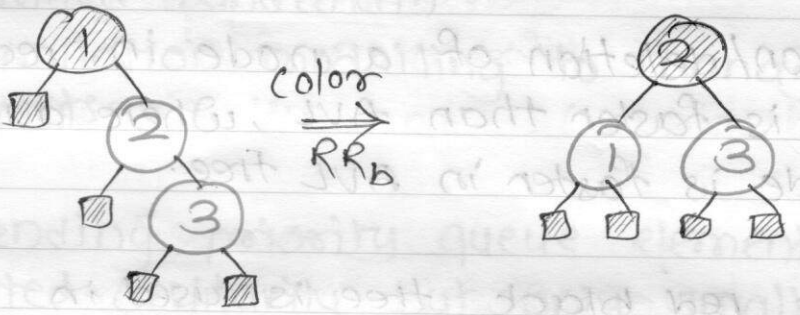
i) 1



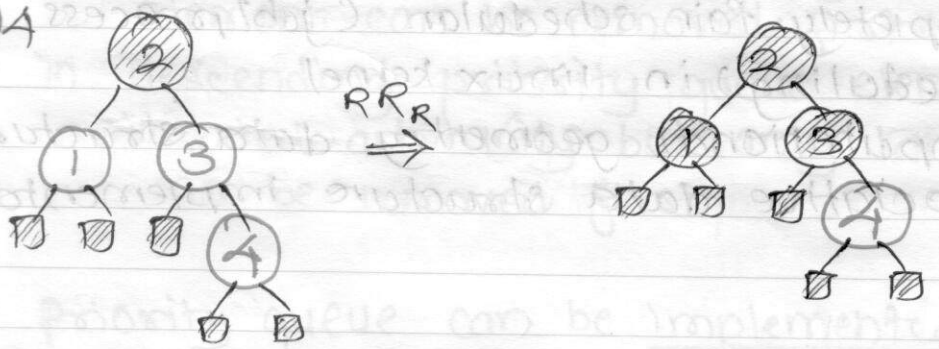
ii) 2



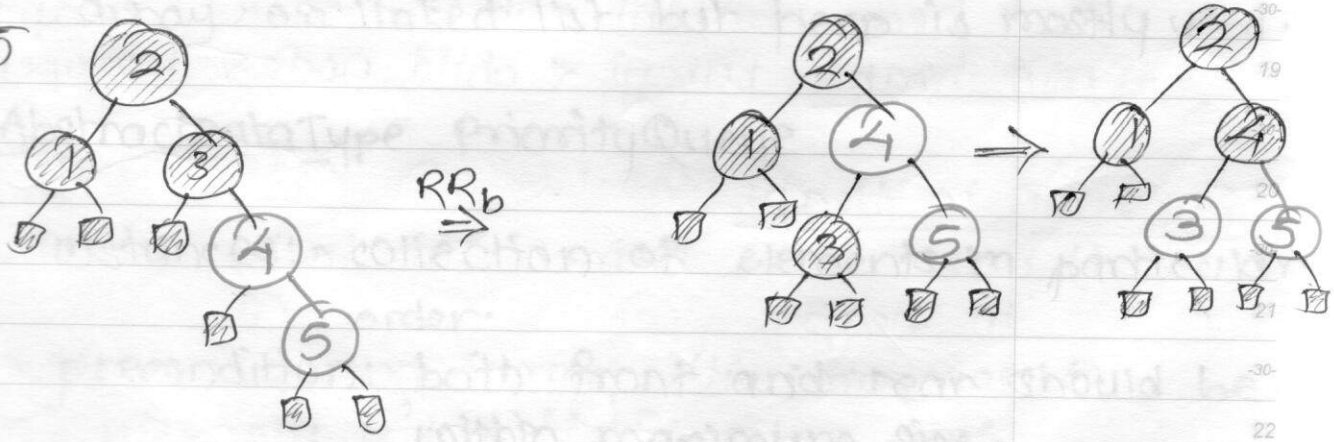
iii) 3



iv) 4

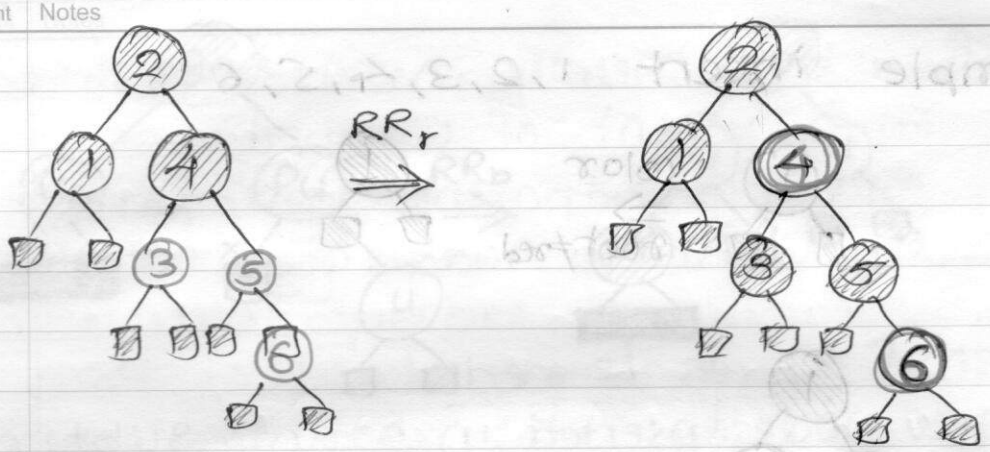


v) 5



time | Appointment | Notes

vi) 6



Use of red-black tree

insertion/deletion of a node in red-black tree is faster than AVL, where as searching a node is faster in AVL tree.

Hence red black tree is used in

1. completely fair scheduler (job/process scheduling) in linux kernel
2. computational geometry data structures
3. associative data structure implementation

Notes | Appointment time

Priority queue

a queue (collection of elements) in specific order

i) ascending order priority queue
ii) descending order priority queue

applications

i) job scheduling in operating system
ii) n/w communication while managing limited bandwidth
iii) simulation modelling to manage discrete events.

In Ascending priority queue elements are inserted arbitarily but only smallest element can be removed first.

In descending priority queue elements are inserted arbitarily but only largest element can be removed first.

Priority queue can be implemented using array or linked list but heap is mostly used.

AbstractDataType PriorityQueue
{
instances: collection of elements in particular order.
precondition: both front and rear should be within maximum size
insertion is not allowed if queue is full

time Appointment Notes

deletion is not allowed if queue is empty

operations:

create( ) :- create an empty queue.
insert( ) : insert new element at next available position
delete ( ) : remove & show smallest element (ascending priority queue) or largest element (descending priority queue) from available elements.
display ( ) : Show all elements in queue.

}

Heap: Is a complete binary tree or almost complete binary tree where every parent node is greater than (max heap) or lesser than (min heap) child nodes.

Types of heap

1) max heap : parent > child nodes } Heapify
2) min heap : parent < child nodes } property

Insertion in Heap:

1) insert new element at last position in heap
2) compare with parent node, swap the two nodes if Heapify property is not followed

3) continue comparing new node with upper parent nodes till Heapify property is satisfied.
4) always fill Heap from left to right

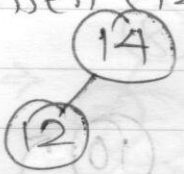
delete an element from heap

1) delete always root node (as root node will be the maximum/minimum element depending upon type of heap)
2) exchange the root with last leaf (last level, last node from left to right) delete that leaf node.
3) continue checking Heapify property from root to leaf (last level) until Heapify property is satisfied, if not satisfied then exchange parent, child node

example insert 14, 12, 9, 8, 7, 10, 18 & delete them all after

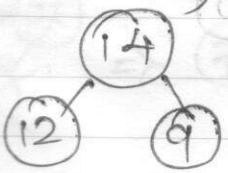
i) insert(14)

ii) insert(12)

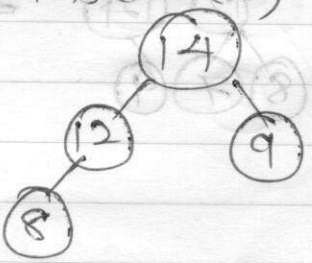


MAX Heap

iii) insert(9)

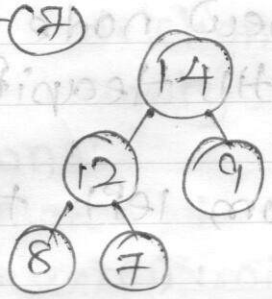


iv) insert(8)

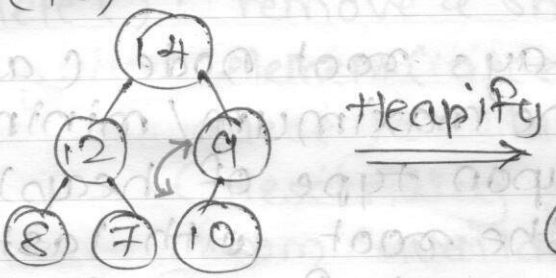


time | Appointment | Notes

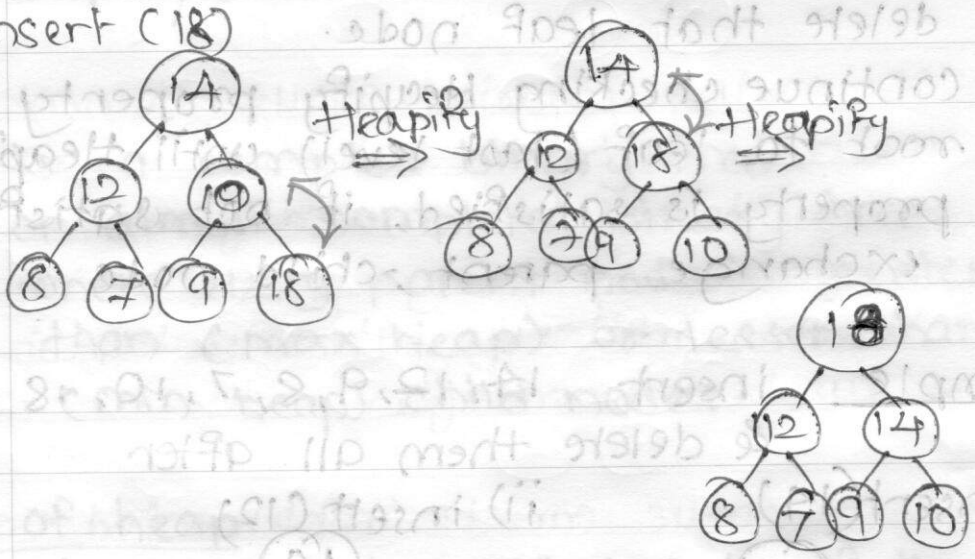
v) insert(7)



vi) insert(10)

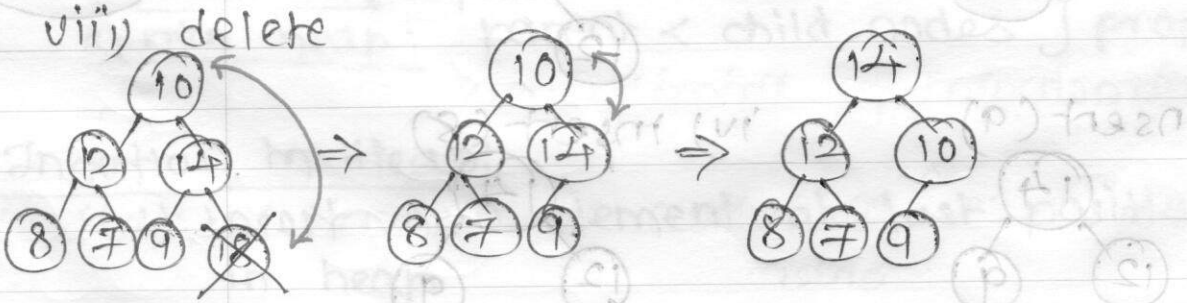


vii) insert(18)



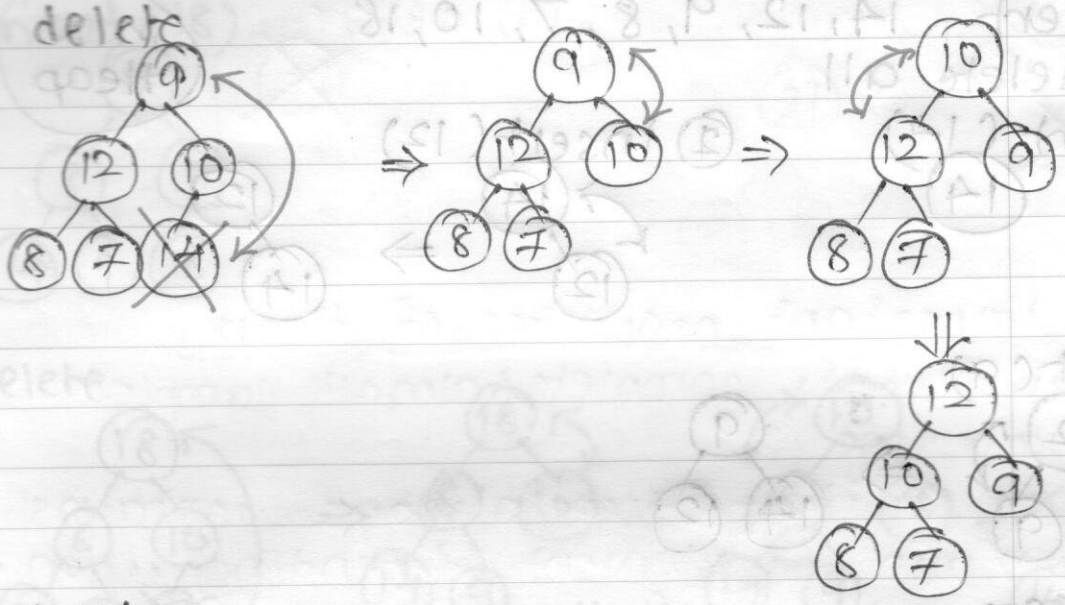
22 Sunday

viii) delete

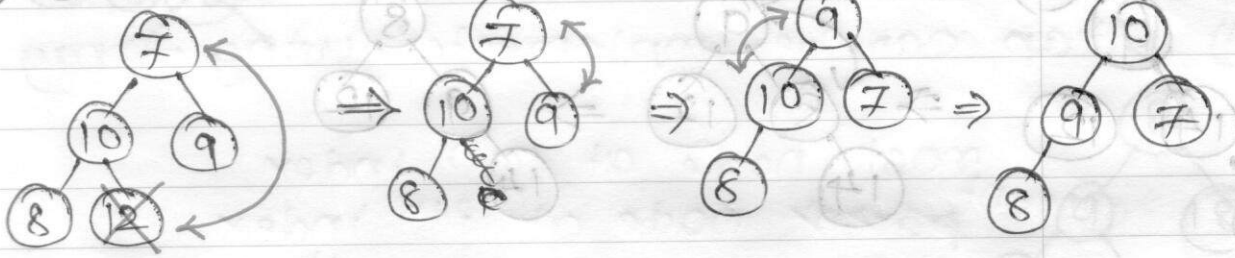


Notes | Appointment | time

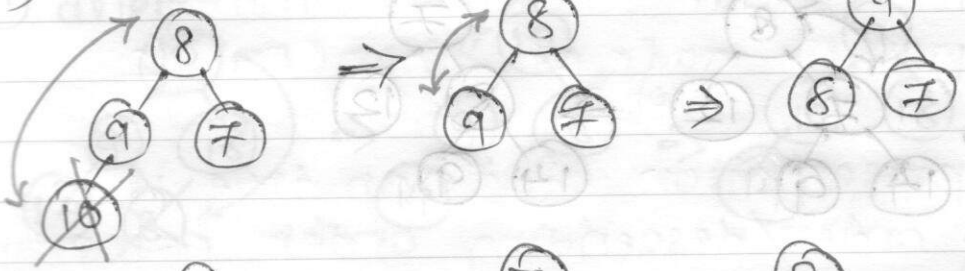
ix) delete



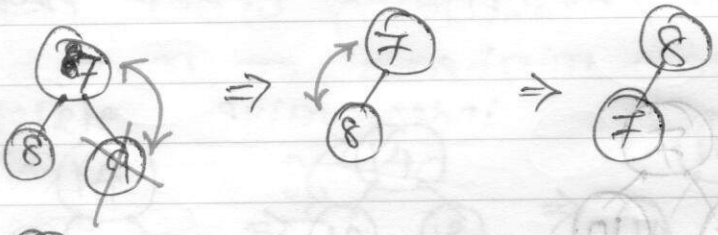
x) delete



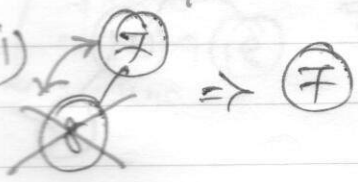
xi) delete



xii)



xiii)



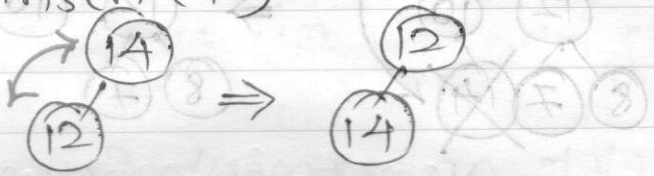
xiv)

insert 14, 12, 9, 8, 7, 10, 18
& delete all

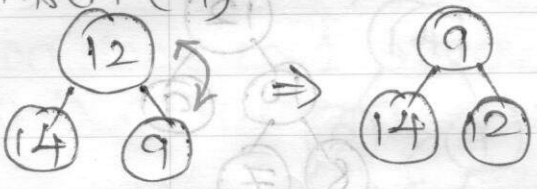
MIN Heap

① insert (14)

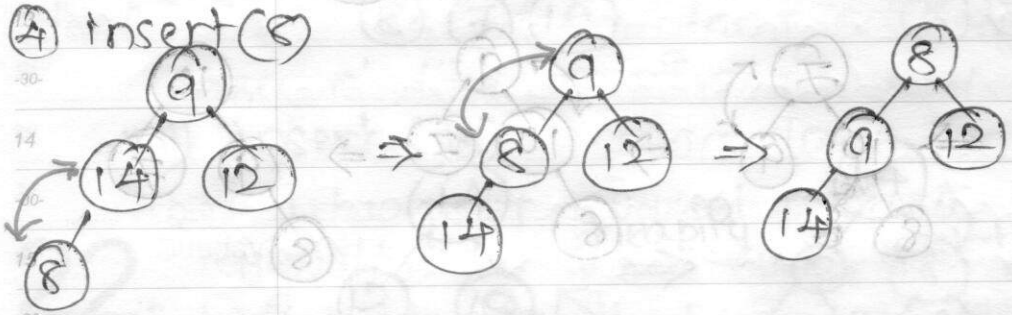
② insert (12)



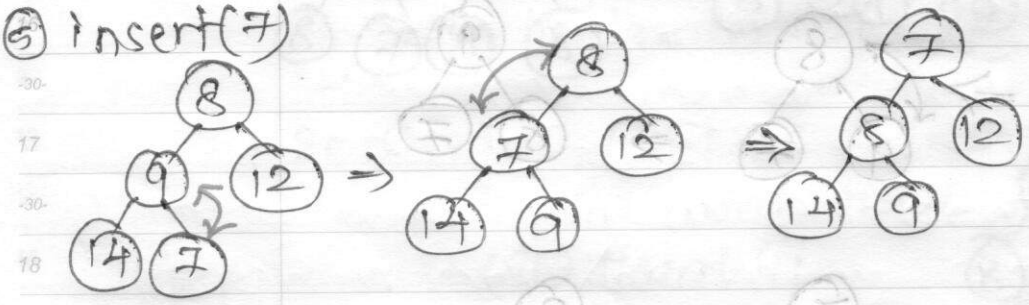
③ insert (9)



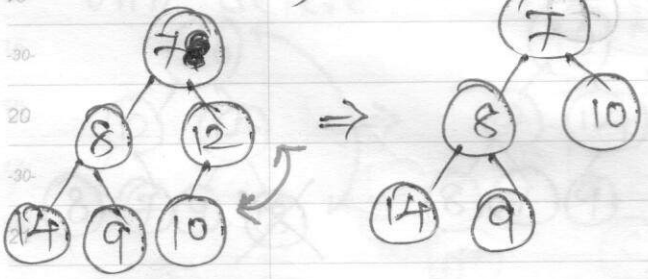
④ insert (8)



⑤ insert (7)

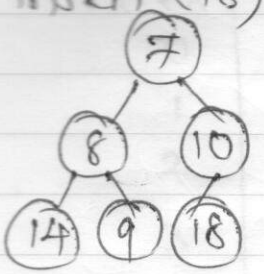


⑥ insert (10)

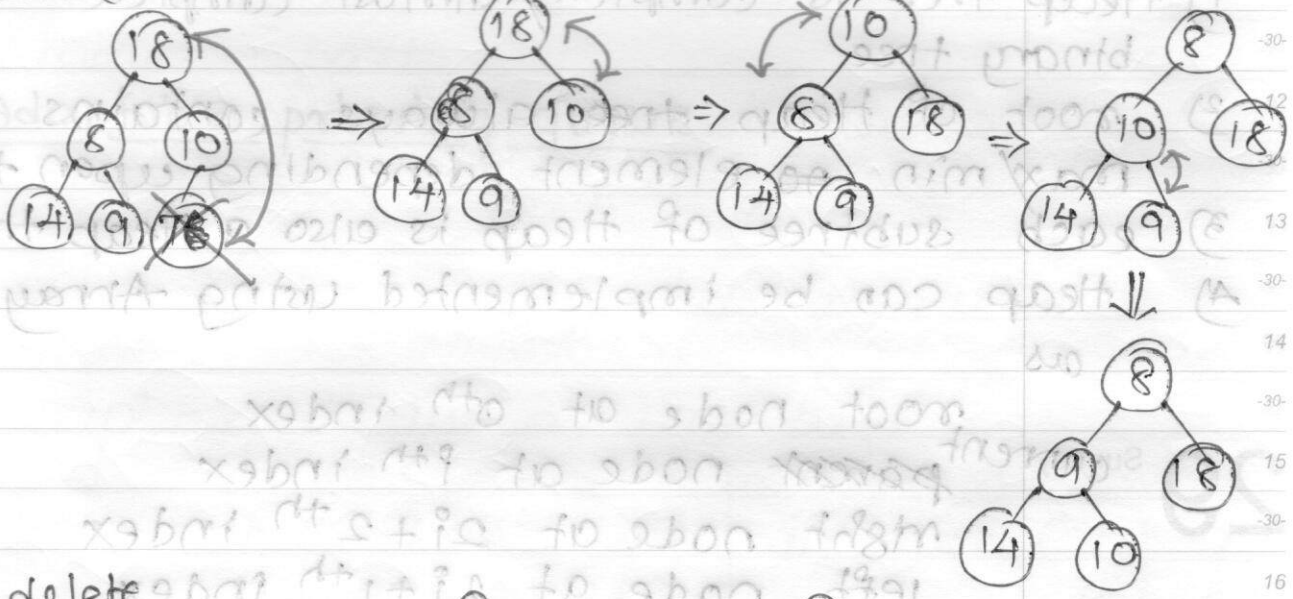


Notes | Appointment | time

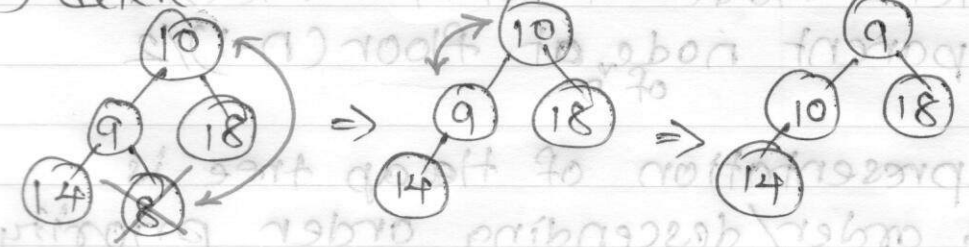
7) insert (18)



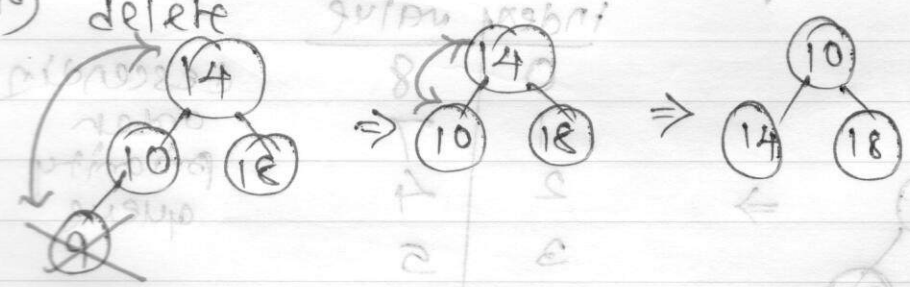
8) delete



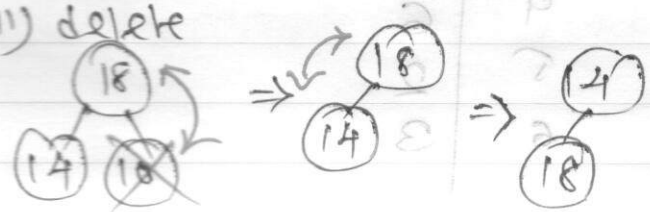
9) delete



10) delete

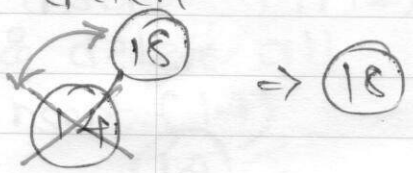


11) delete



time Appointment Notes

12) delete



13) delete

Some important properties of Heap

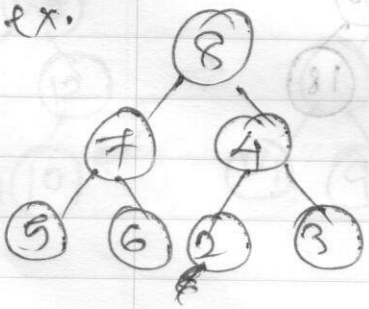
1) Heap tree is complete / almost complete binary tree
2) root of Heap tree always contains max/min ~~no~~ element depending upon type
3) each subtree of Heap is also a Heap tree
4) Heap can be implemented using Array as

current root node at 0^{th} index
~~parent~~ node at i^{th} index
right node at $2i+2^{th}$ index
left node at $2i+1^{th}$ index
parent node of n at floor $(n-1)/2$

5) Array representation of Heap tree is ascending order/descending order priority queue if Heap minheap or max heap resp.

ex.

max heap



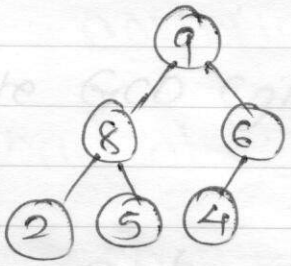
⇒

index	value
0	8
1	7
2	4
3	5
4	6
5	2
6	3

descending order priority queue

index	value
0	9
1	8
2	6
3	2
4	5
5	4

⇒



Adaptable priority queue