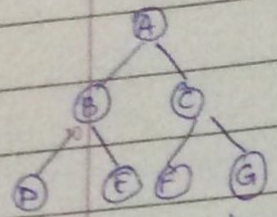
Binary Tree Traversal:

The process of visiting each node in the tree exactly once.

a) Breadthfirst / Level order:

We will visit all nodes at same depth before visiting nodes at next level. (L → R or R → L)



A, B, C, D, E, F, G

.) Using linked list implementation

if we reach G, we can't go to E.

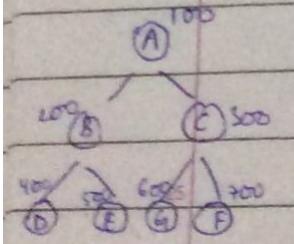
We will use an auxiliary structure Queue

Steps:

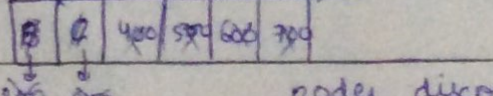
1.) When we visit a node, we can keep reference / add of all its children in queue so that we can visit them later

2) A node in a queue is called discover node whose add is known to us but we've not visited it yet.

3) The node which is discovered first is dequeued first. (visited first)



take an empty queue:



nodes discovered : B, C
visited : A (already processed)

Final answer:

A B C D E G F

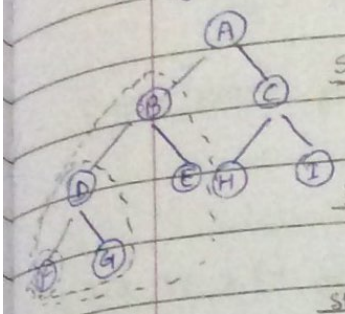
100 will be discovered first and its childrens' address will be put in queue.

→ level order

Camlin Page
Date / /

b.) Depth first :

we first visit the whole subtree of a child before going to another child.



6 type of orderings are possible :

Std → 1.) Root first - L.S. - R.S. (Right sub-tree)

2.) Root first - R.S. - L.S.

Std → 3.) L.S. - Root - R.S.

4.) R.S. - Root - L.S.

Std → 5.) L.S. - R.S. - Root

6.) R.S - L.S. - Root

1.) & 2.) : Pre-order (Root first)

3.) & 4.) : In-order (Root in b/w)

5.) & 6.) : Post-order (Root at last)

→ Pre-order :

1.) Visit the root

2.) Traverse the L.S. in pre-order

3.) " R.S. " "

→ In-order :

1.) Traverse L.S. in in-order

2.) Visit the root

3.) Traverse R.S. in in-order

→ Post-order :

1.) Traverse L.S. in post-order

2.) " R.S. "

3.) Visit the root

* can't go in reverse dir^ directly.

1.) In Pre-order :

FDGBE(A)HC I

root

L.S.

R.S.

2.) Pre-order :

A B D F G E C H I

→ Can you construct a binary tree using above two?

3.) Post-order :

F G D E B H I C A

→ Can you construct a binary tree using 1.) & 3), 2.) & 3)?

* In order to implement this, we use stack

inorder

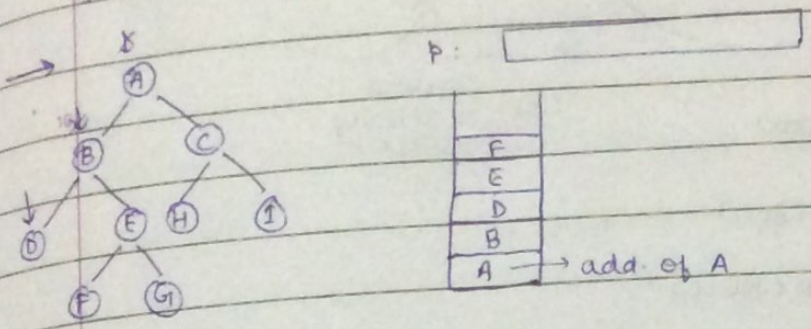
1.) Iterative soln :

```
struct Bnode
{
struct Bnode * left ;
struct Bnode * right ;
    int data    ;
}

struct Bnode *S[100];   // here, S will act as stack (array of pointers)

                        // tree → add of root
main ( )
{
struct Bnode * tree = NULL ;

void intrav (struct Bnode * tree)
{
    struct Bnode *p = tree ;
do
{
  while (p != NULL)     // we have traversed the left branches as far as possible saving ptrs to nodes which have passed //.
  {
    push ( s, p) ;
     p = p → left ;
  }
```

1) In comp memory, special section : stack or implicit stack.
** 2) Func" calls in prgrm are implemented using a stack.

Camlin | Page
Date / /

```
if (! isempty (s))
{
   p = pop (s) ;
   pf ("%d", p→ data) ;
    p = p→ right ;
}
} while (! isempty (s) || p! = NULL)
{
```



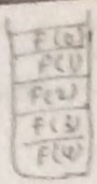
O/p : D B F E . . . .

Imp. * In iterative sol" of in-order, we have to use the stack explicity.

2.) Recursive sol" :

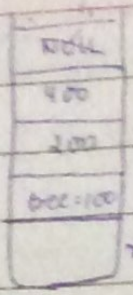
```
void intravR ( structnode * tree)
{
   if ( tree ! = NULL)
   {
      intravR ( tree → left) ;
      pf ("%d", tree → data) ;
      intravR ( tree → right) ;
   }
}
```

→ Post (4)



Firstly, F(0) will be executed

Camlin Page
Date / /



→ compared, popout

intrav (100)

O/p : D B F E .....

main()
↓
intravR(100)
↓
intravR(200) → intravR(400) → intravR(NULL)

P8 will be executed (S#1 is already executed)

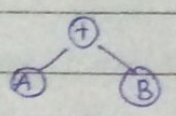
↓
intravR(500)
↓
intravR(800) → NULL
↓
intravR(NULL)

21-02-17

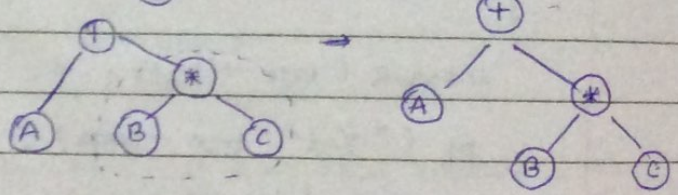
Expression Tree (App^{n}s of B.T.)

An exp^n of operands and operators can be represented by by a strict B.T. The root of strict B.T. contains an operator that is to be applied to the results of evaluating the exp^{n}s represented by left & right subtrees. A node representing an operator is an internal node whereas a node representing an operand is a leaf node.

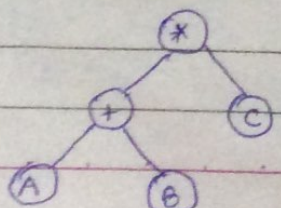
Eg. 1) A + B



2) A + B * C



3) (A + B) * C

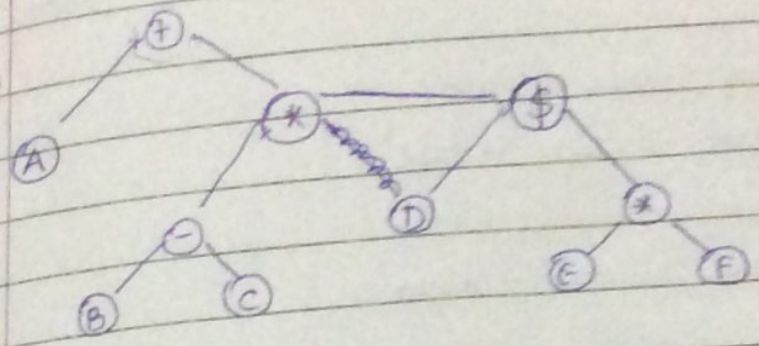


Camlin Page
Date / /

4) A+(B-C) * D $ (E * F)

$(A \$ B) = A^B$

Precedence : $
* /
+ -



→ We will get B strict B.T. only if binary operators are applied.

1) Pre-order

+ A * - B C $ D * E F ≡ Prefix

Prefix :

≡ +A * -BC $ D * EF

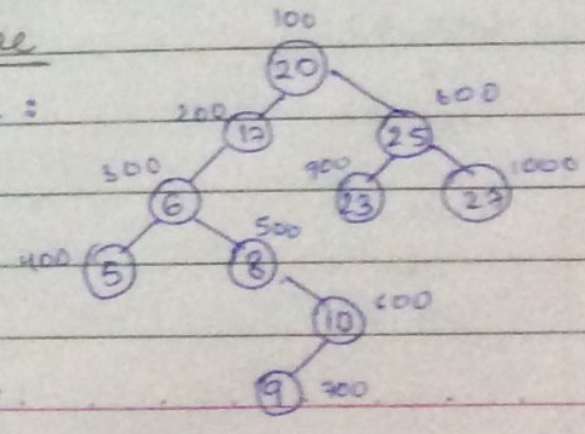
→ Postfix : ≡ → Post-order

A BC - D EF * $ * +

→ In-order ≡ Infix exp^n (without parentheses)

Binary Search Tree

1) Searching of a node :



→ Search 8

Compare 8 & 20

8 < 20 → Go to L.S.

(Make comparison)

Algorithm :

a) struct Bnode *search_BST1 (struct Bnode *tree, int key)

Iteration

{

while (tree != NULL && key != tree → data)

{

if (key < tree → data)

tree = tree → left ;

else

tree = tree → right ;

}

return tree ;

}

→ tree d.n.e. or item d.n.e. ⇒ it will return NULL

Recursion

b) struct Bnode *search_BST2 (struct Bnode *tree, int key)

{

if (tree == NULL || key == tree → data)

return tree ;

if (key < tree → data)

return search_BST2 (tree → left, key) ;

else

return search_BST2 (tree → right, key) ;

⇒ Trace this func^n using stack or

(100, 6)

↓

(200, 6)

↓

(300, 8)

↓

(500, 8)

max^m comparisons 6 (here) n = 10

searching : O(h) → height : Worst case

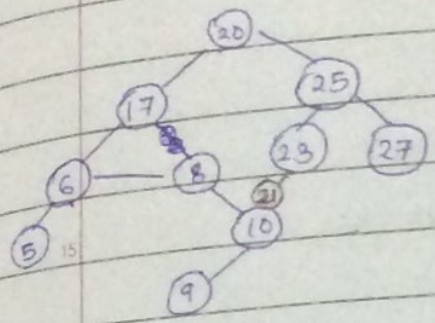
worst case

* In array :
  Worst case : O(n)
* In L.L. : O(n)
* In BST : O(h)

2) Inserting a node : O(h)

→ 20, 17, 6, 8, 25, 10, 23, 5, 27, 9

17 < 20 → L.S.



6 < 17 , 6 < 20

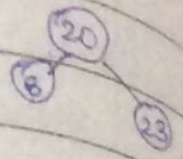
insert 21

Algorithm :

```
void BST_insert1 (struct Bnode * tree, int key)
{
    struct Bnode *y, *x, *z;
    z = (    ) malloc ...
    z → data = key;
    z → left = NULL;
    z → right = NULL;
    while (x != NULL)
    {
        y = x;
        if (key < x → data)
            x = x → left;
        else
            x = x → right;
```

x = tree;
y = NULL;
↓
y ko store kr rhe hain x ka value & x ko traverse kra rhe h.

```
if (y == NULL)          // tree is empty
    tree = z;
else if (key < y → data)
    y → left = z;
else
    y → right = z;
```



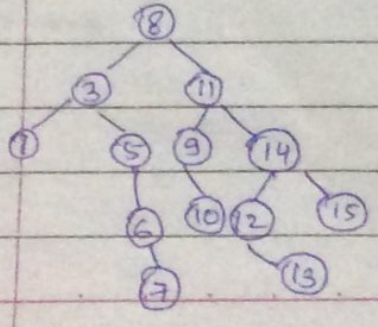

```
void BST_insert2 (struct Bnode * tree, int key)
{
    struct Bnode * z;
    z = ___ malloc ( , );
    if (tree == NULL)
        tree = z;
    else if (key < tree → data)
        BST_insert2 (tree → left, key);
    else
        BST_insert2 (tree → right, key);
}
```

Deletion of a node : O(h)

* After deletion, the obtained tree must also be a BST. In order to delete a node, we can consider three cases:

i) If node to be deleted has no children (leaf node), it may be deleted without further adjustment to tree.

eg.



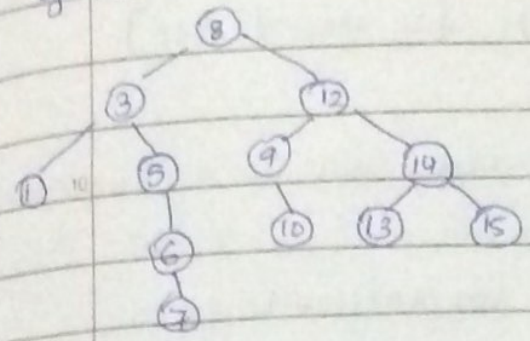
To delete 15

ii) If node to be deleted has only 1 sub-tree, then its only child can be moved up to take its place.

Eg. delete 5 in prev. tree.

iii) node has both subtrees : it is replaced by the left-most child of its right subtree (min^m element in right subtree)

Eg. delete 11 : left-most : 12



→ The inorder traversal of a BST results in an ascending order sequence

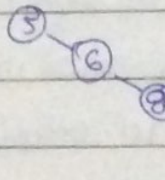
1 3 5 6 7 8 9 10 12 13 14 15

→ The max^m element in a BST can be found by traversing the root in right dir^n all along until a node with no right link is found.

Time Complexity :

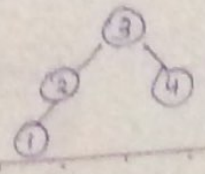
① 20, 17, 6, 8, 25, 10, 23, 5, 27, 9 h = 5

② 5, 6, 8, 9, 10, 17, 20, 23, 25, 27 h = 10

② →  : Skewed BST (all either in left or right subtree)

∴ → Right skewed BST

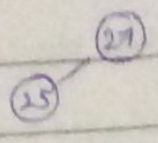
* BST will be like array or L.L if it is skewed BST



$O(\log_2 n)$

Almost Binary Tree

② 27, 25 ..... 6, 5 $h = 10$

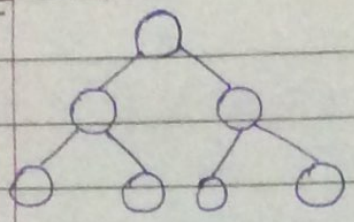


: Left - skewed BST

* All op^{n}s on BST completely depend on the shape of BST.

→ The worst case : $O(n)$ [if skewed BST]

23-2-17 Complete BST : all leaf nodes will be at same level



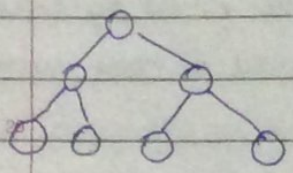
$h = \log_2(n+1) - 1$

Time complexity : $O(\log_2 n)$

Mid-term

In skew BST, $h = n-1$ → $O(n)$

If complete tree :



$h = \log_2(n+1) - 1$ → This is better

→ $O(\log_2 n)$

Binary Height balanced Tree or AVL Trees

25 A B.T. is Height balanced if diff of the heights of the left and right subtree of each node are in the range of {-1, 0, 1}. In other words, it is a B.T. in which heights of 2 subtrees of every node never differ by more than 1.

→ For this, we need to calculate Balance Factor of every node in the Binary Tree.

Balance factor (BF)

The B.F should lie in the range {-1, 0, 1}

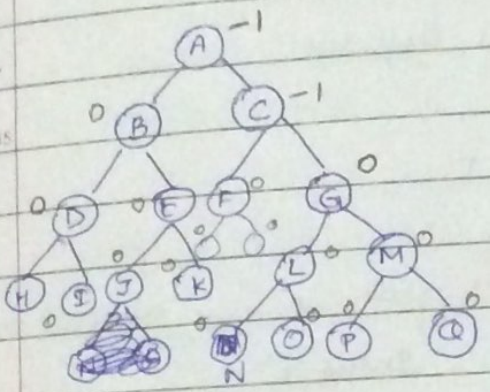
B.F of a node

(BF) = height of left subtree − height of right subtree

→ For a leaf node, B.F. = 0

→ The B.F. -1, 1 depend on whether height of L.S. is greater than or less than height of its R.S.

eg.



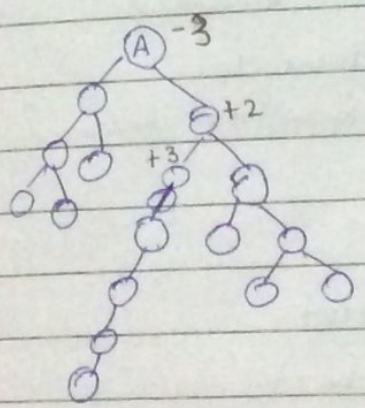
	(h) L.S.	B.F.	(h) R.S.
A →	3	-1	4 ✓
B →	2	0	2 ✓
D →	1	0	1 ✓
C →	2	-1	3 ✓

→ B.F : {-1, 0, 1}

⇓

Height Balanced Binary Tree

→



→ Not a height balanced Binary Tree.

→ AVL Trees : Adelson, Velskie & Landis

These scientists gave algo to such trees

⇓

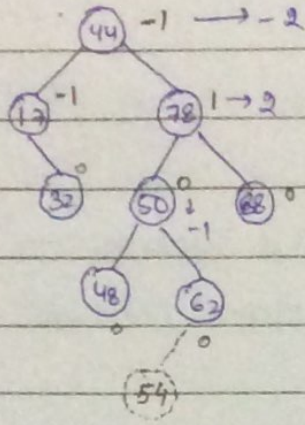
AVL Search Tree

AVL Search Trees.

In a given balanced B T, if we insert a new node, the resultant tree may / may not be balanced

Eg of a BST

a)



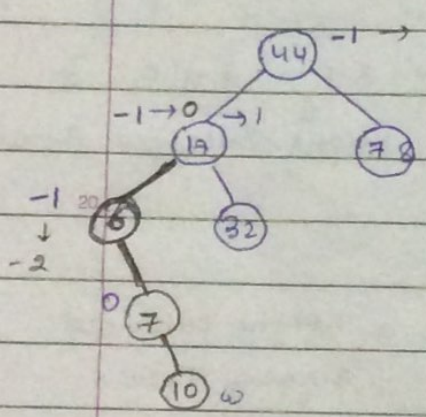
Insert node (54)

→ After Insertion, it doesn't remain Height balanced.

Imp.

* Whenever we insert a node, then all the nodes in the path from that node to root node will have changed B.F
* Ancestors B.F will get affected.

b) Insert 6 in above I



→ Still, it is a balanced tree

Insert 7

→ still balanced

Insert 10

→ Now, not balanced

Steps to follow for Insertion

1. Let newly inserted node be w.
2. Perform std BST, insertion for w.
3. Starting from w, travel up and find the 1st unbalanced node.

All Rot^n's will be done wrt middle key

4. Let z be the 1st unbalanced node.
5. Let y be the child of z (consider that child which is coming in path)
& x be the grandchild z that also comes on path from z to w

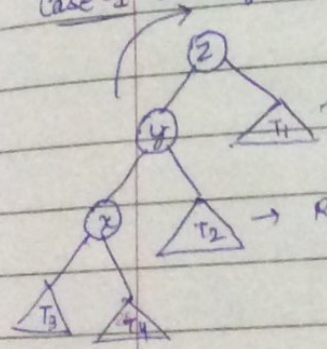
z : 78
y : 50 (In eg a)
x : 62
w : 54

6. Rebalance the tree by performing app. rotations on the subtree rooted with z. There can be 4 possible cases that needs to be handled as x, y & z can be arranged in 4 ways.

i.) y is left child of z, x is left child of y (LL case)
ii.) y → left child x → right child (LR case)
iii.) y → right child x → left child (RL case)
iv.) y → right child x → right child (RR case)

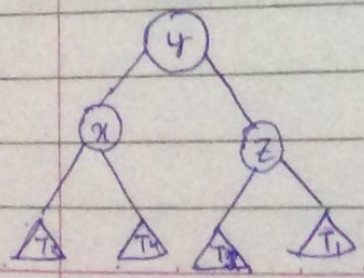
Rotation : It is a way of locally reorganising (only in some subtree) a BST.

Case-I : Left-left case : Rotation wrt (y, z) in right dir^n (right rot^n)



{y is in middle}
x < y < z

Make y → root
z → right child



Now, z will automatically become balanced

* After appⁿ of Rotation, the tree must remain BST.

Camlin | Page
Date 28 / 02 / 17

height of subtree T_3 : h (before insertion)
insertion occurs at T_3 (node W)
height after insertion : h+1

T_4 : possible heights : h ✓, h+1 ✓, h+2 ✗

T_3 : h → h+1

Before inser^n (for h+2) : B.f. = (h+2) - h = 2 ✗
(Before, it was AVL tree)

* After insertion, height of x, y and z ↑se definitely.
If height doesn't change, z won't become imbalanced.

T_3	T_4	diff
h → ~~h+1~~	h	1
~~h → h+1~~	h+1	0 → ~~if you consider this case~~, height of x, y, z remain same → this case is not possible

→ height of T_4 : h

→ x : h+1 → h+2

→ of T_3 : h ✓ (height will remain same) | h+1 ✗ (imbalanced before) | h+2 ✗ (afterwards, imbalanced) | h+3 ✗

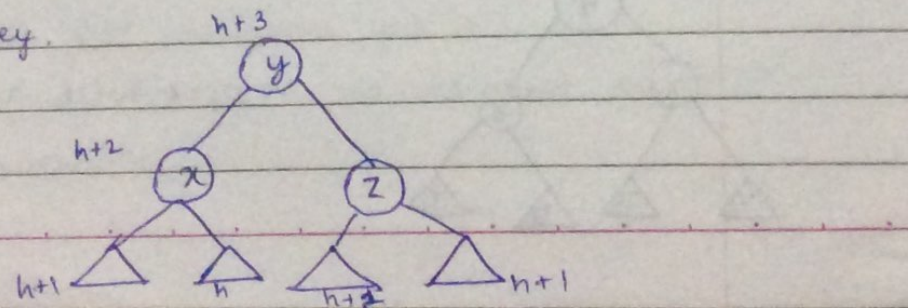
→ y : h+2 → h+3

→ T_4 : h-1 | h ✗ | h+1 ✓ | h+2 ✗ | h+3 ✗ | h+4 ✗

z : imbalanced after insertion.

z : (h+3 → h+4)

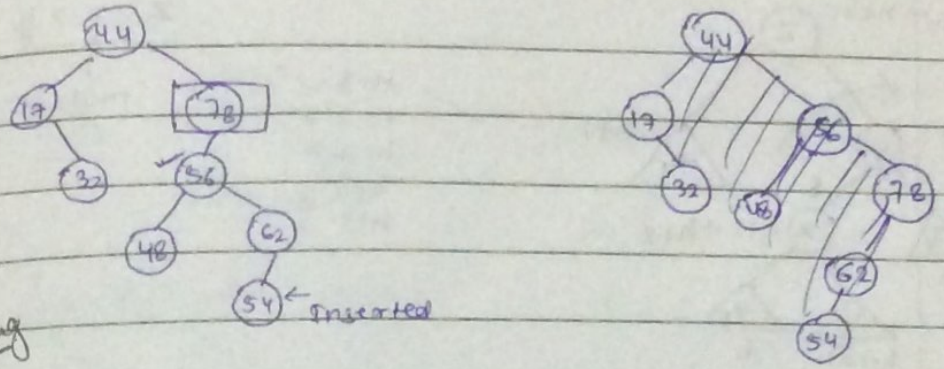
* In left - left case, we apply right rotation along the middle key.



* the original height (before inser^n) of a subtree rooted at z was "h+3". After inser^n and then the rot^n, the height of the new subtree rooted at y is also "h+3"

→ when h(z) = h+3 (earlier) ⇒ it was AVL Tree
when inserted : (h+3) → (h+4) → not AVL Tree
Now, again height of sub-tree : (h+3) ⇒ again becomes AVL Tree

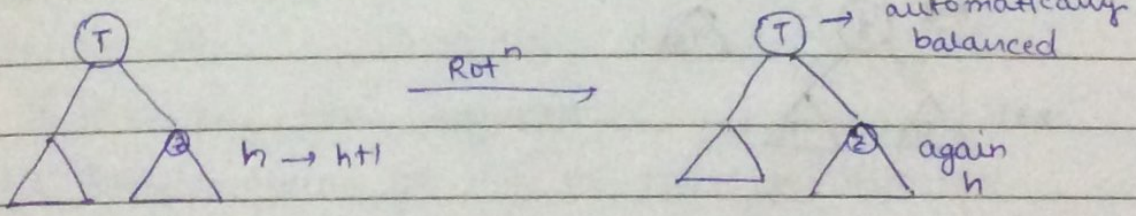
In earlier eg :



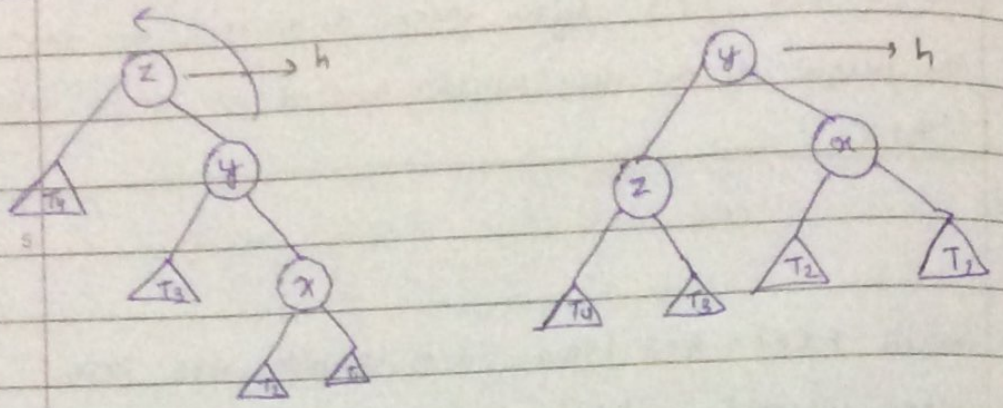
while inserting

* We only need to rebalance the sub-tree rooted with z and complete tree becomes balanced as height of subtree after app. rot^{n}s rooted with z becomes same as it was before insertion.

→

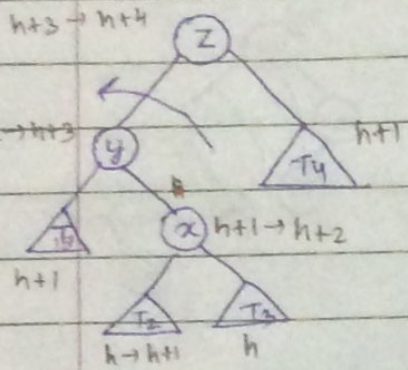


Case - II : Right - Right case Left Rot^n wrt middle key
$z < y < x$



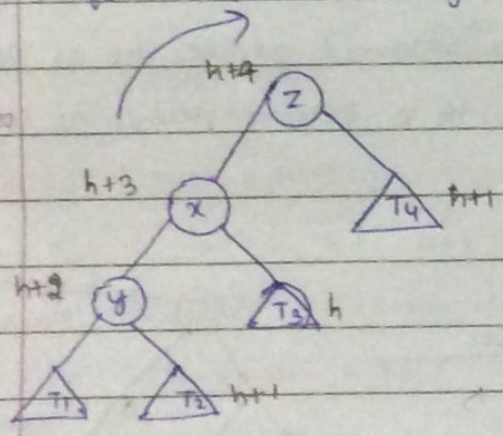
Case - III : Left - Right case Inser^n : either at T_2 or T_3
(assume at T_2)

$z > x > y$
middle key



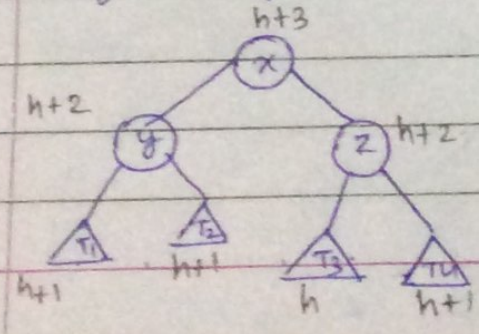
h+1 ✓
h+2 ✗
h+3 ✗
h+4 ✗
h+5 ✗

Step 1. Perform left rot^n along the middle key (x, y)



→ again, it is unbalanced

Step 2 : Perform right rot^n along middle key (x, z)



middle key

→ h

T_2 or T_3

$x > y$

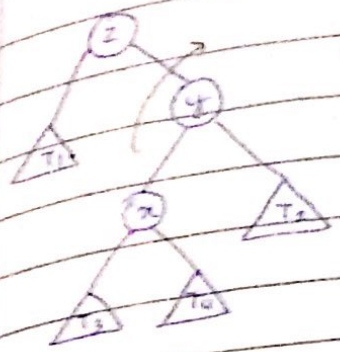
middle key

x, y)

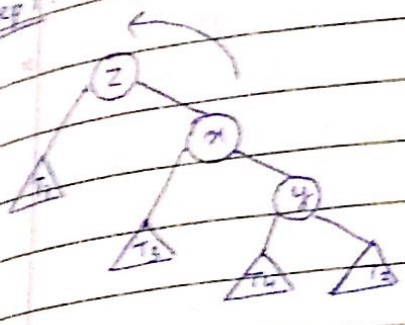
unbalanced

z)

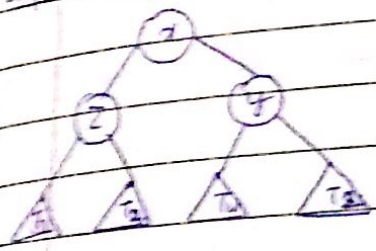
Ex-III Right - Left case :



Step 1 : Right rotⁿ wrt (x, y)



Step 2 : Left rotⁿ wrt (x, z)



→ Time Complexity :

In case - III : $x \to T_3 \Rightarrow x \xrightarrow[1]{\checkmark} z \xrightarrow[2]{\checkmark} T_3$

2 links you have adjusted.

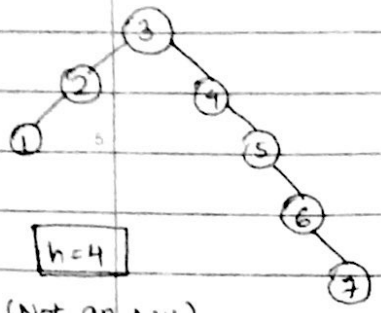
∴ Rotⁿ doesn't depend on no. of nodes in sub-tree.

∴ Time Complexity of Rotation : $O(1)$

Camlin Page
Date / /

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

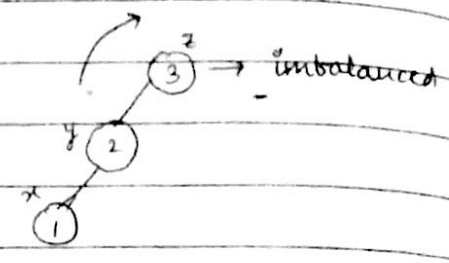
BST:

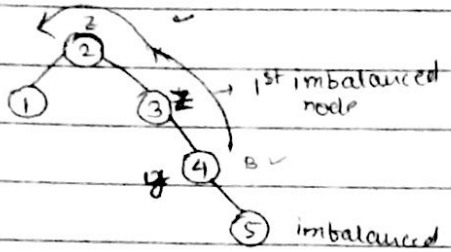


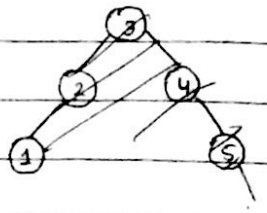
h=4

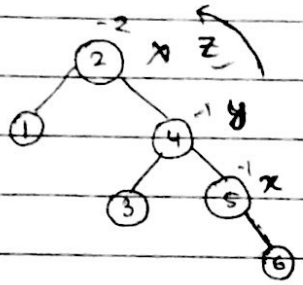
(Not an AVL)

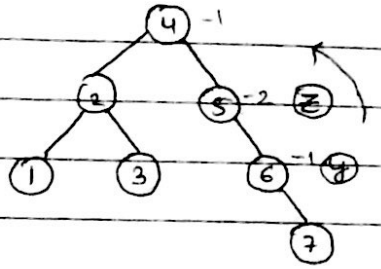
AVL search tree:

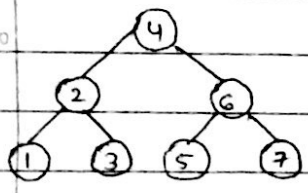






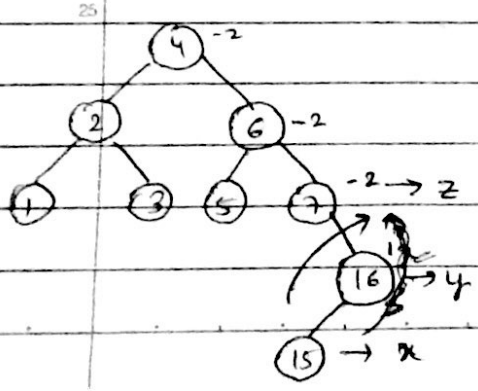


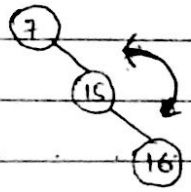


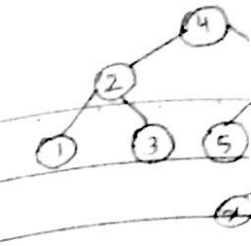


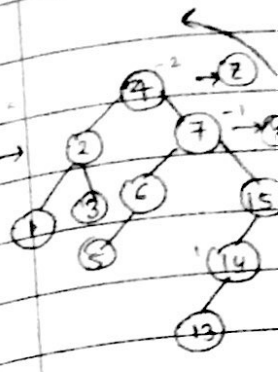
eg: 3, 2, 1, 4, 5, 6, 7 || 16, 15, 14, 13, 12, 11, 10

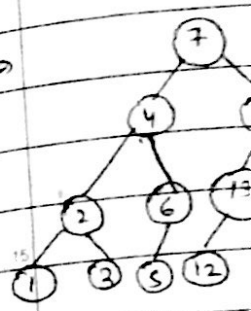
Right-left











2-3-17

Proposition

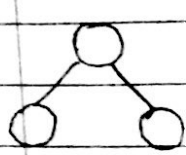
The height

is always

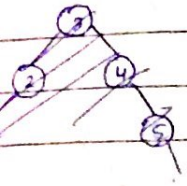
Perfectly

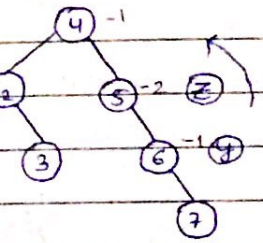
right su

or B.F.

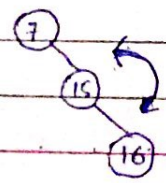


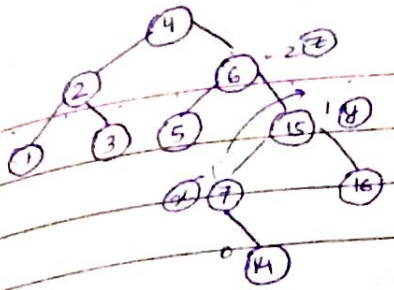
AVL search tree :

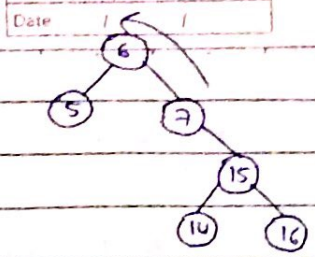


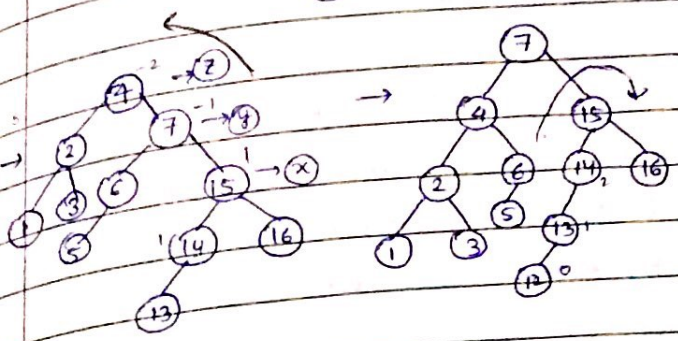


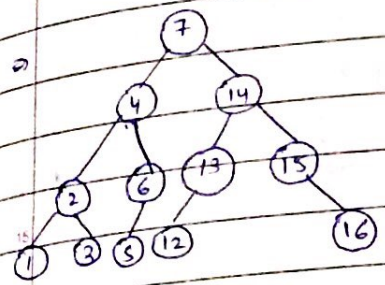
12, 11, 10











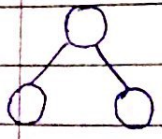
1-3-17

Proposition :

The height of an AVL tree storing 'n' elements (nodes) is always $O(\log_2 n)$

Perfectly height balanced tree : in which left and right subtree of each node are at same height.

or B.F. of each node = 0. (also complete binary tree) or full binary tree



$h = \log_2(n+1) - 1$

$h : O(\log_2 n)$

$h \propto \log_2 n$

→ AVL search tree $h \propto \log_2 n$ $O(\log_2 n)$

→ BST (skewed) $h \propto n$ $O(n)$

→ BST_search : $O(h)$ complete binary : $O(\log_2 n)$ skewed : $O(n)$

→ AVL - search : $O(h)$ → worst case : $O(\log_2 n)$

* Searching, deletion, insertion would be of same order in AVL & BST Trees. Only diff. b/w both is that AVL is height balanced.

→ Insertion

BST_insertion : $O(h)$ —worst case→ $O(n)$

AVL_insertion : $O(h) + O(1) = O(h)$

($O(h)$: Before Rotⁿ; $O(1)$: for Rotⁿ)

worst case → $O(\log_2 n)$ (always balanced before insertⁿ)

→ AVL deletion :-

It is similar to BST deletion.

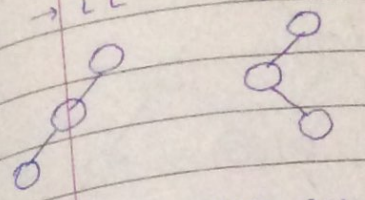
To make sure that given tree remains AVL tree after every deletion, we must arguement the std. BST delete opⁿ to perform some rebalancing.

Let w be the node to be deleted.

- Perform std. BST_delete for w.
- Starting from w, travel up & find 1st unbalanced node.
- Mark it as z.
- let y be larger height child of z. x is child of y with larger height.
- Rebalance the tree by performing app rotⁿs on subtree rooted with z.
- There can be 4 cases that need to be handled as x, y & z can be arranged in foll. 4 cases:

→ 4 both subtree

→ LL → LR



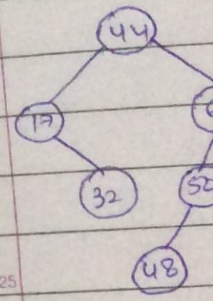
These cases are

* Unlike insertion AVL tree Aft ancestors of z trace the path

→ Time complexit worst cast :

→ Insertion

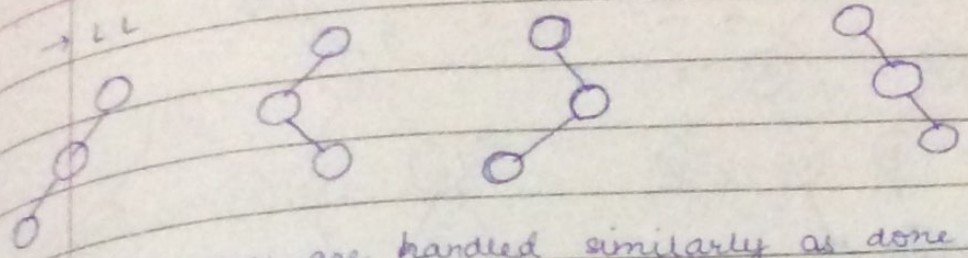
eg.



① del : 3

→ if both subtree have same height : we can choose anyone

→ LL → LR → RL → RR



These cases are handled similarly as done in insertion

* Unlike insertion, fixing node z will not fix the whole AVL tree. After fixing z, we may have to fix ancestors of z as well. Thus, we must continue to trace the path until we reach the root.

→ Time Complexity

worst cast : all above z are unbalanced

⇒ $O(\log_2 n)$ + $O(1)$ + $O(\log n)$

$O(\log_2 n)$ ⇓ : you have to rotate all above z

$O(1)$ ↑ Rot^n

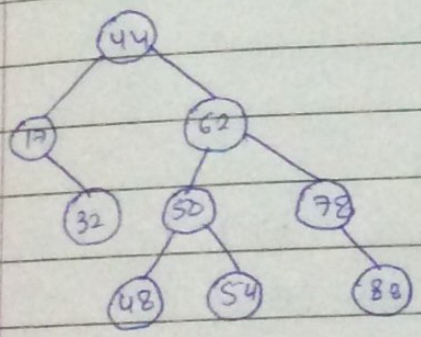
$O(\log n)$ ⇓ : To delete $O(h)$

$= O(\log_2 n)$

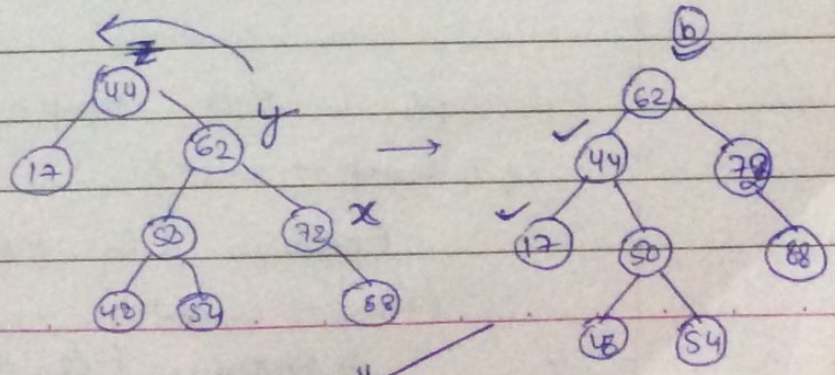
→ Insertion : $O(1) + O(\log_2 n) = O(\log_2 n)$

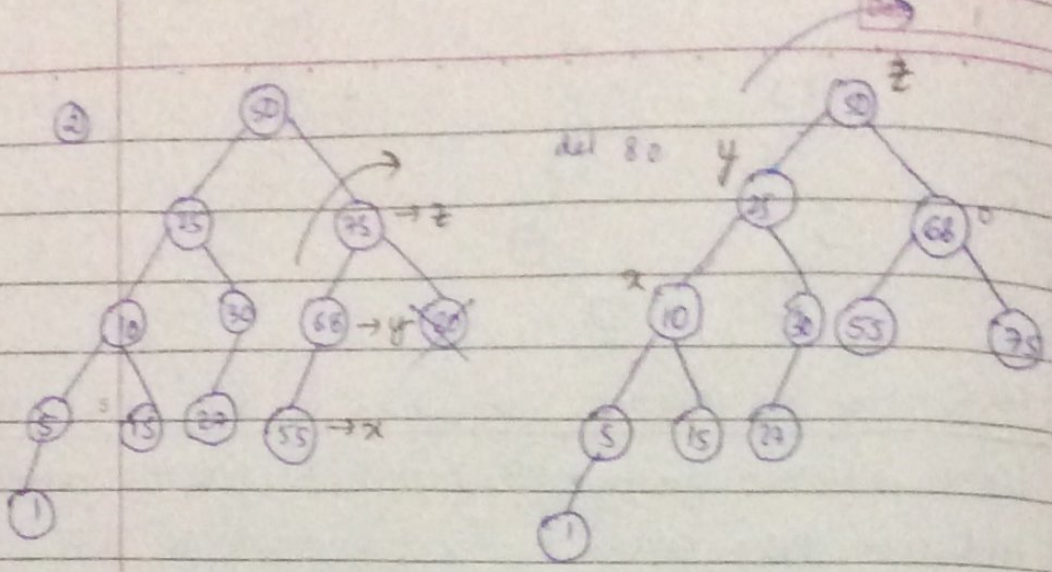
$O(1)$ ↓ for rot^n ; $O(\log_2 n)$ ↓ for ins^n

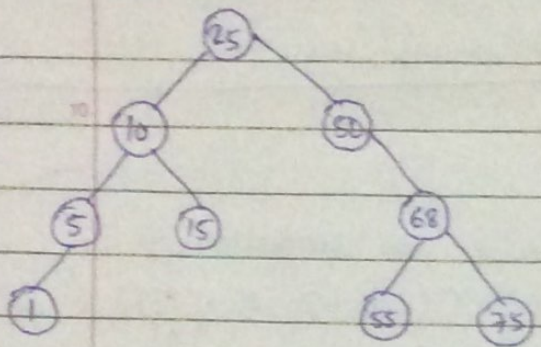
Eg



① del : 32







→ Level Order Traversal (BST) :-

```
void level_order ( struct Bnode * tree)
{
  struct Bnode * temp ;
  struct Queue * Q = createQueue ( ) ;
  if ( ! tree)
      return ;
  Enqueue ( Q, tree ) ;
  while ( ! IsemptyQueue (Q) ) {
      temp = Dequeue (Q) ;

    pf ( "%d", temp → data ) ;
  if ( temp → left)
      EnQueue ( Q, temp → left ) ;
  if (temp → right)
      EnQueue ( Q, temp → right ) ;
```

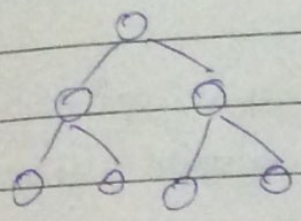
DeleteQueue (Q);
}

→ Height of Binary Tree :

Recursion :-

```
int height (struct Bnode * root)
{ int leftheight, rightheight;
  if (root == NULL)
      return 0;
  else {
      leftheight = height (root → left);
      rightheight = height (root → right);

  if ( leftheight > rightheight )
      return ( leftheight + 1);
  if ( rightheight > leftheight )
      return ( rightheight + 1);
```

Part - C



$N = 2^n - 1$

Worst case : $2^n - 1$

* Every leaf node has a sibling

2-3 Tree

Def": A 2-3 tree is a tree in which each internal node (non-leaf) has either 2 or 3 children and all leaves are at same level

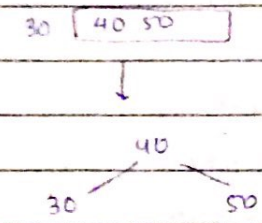
Different Cases in Insertion of a key-value into a 2-3 Tree :

→ It starts with leaf node

30 | 50 60 : 70 will go up (middle one)

Case-I: Insertion starts with appropriate leaf node

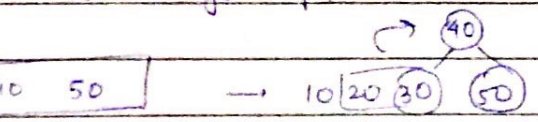
50, 40, 30, 20, 10, 15, 25, 35, 45

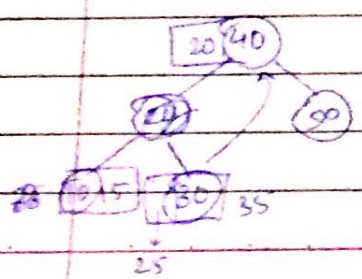


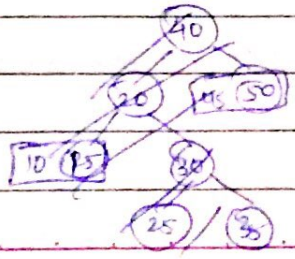
Case-II: If leaf node has "space" (single valued) then add the new key-value

40 50 Put 40

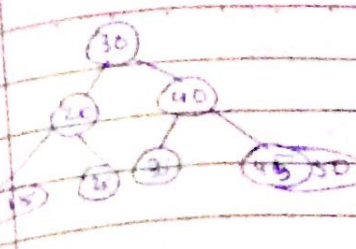
Case-III: No space (double valued), split op" should be performed this leads to propogation upwards occurs Middle value goes up







* We insert a key-val



Deletion of a k

* First, search the k we delete op"

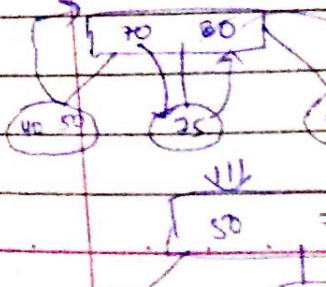
* a) key-value is

i) leaf node is delete the key valued

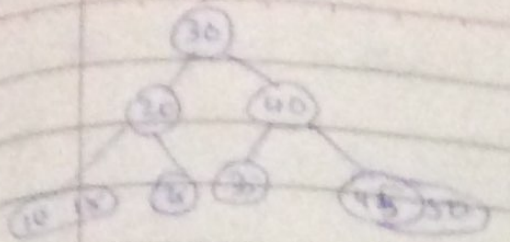
ii) leaf node is A sibling is (if we delete (if 35

→ We can't dire have att Use eith from p

B. Siblin value



* We insert a key-value, not a node



Deletion of a key-value from a 2-3 Tree :

* First, search the key-value. If present, then perform the delete op^n. Else, return (-1) and give error msg

* a) key-value is found in a leaf-node :

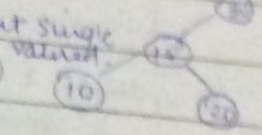
i) leaf node is double valued

Delete the key-value and make the node single valued

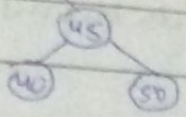
ii) leaf node is single valued

A. sibling is double valued, parent single valued

(If we delete 25, it will become :)



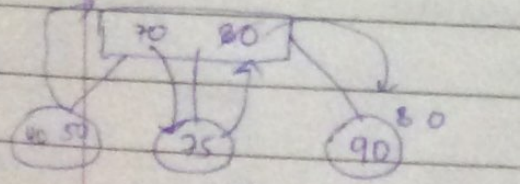
(If 35 :

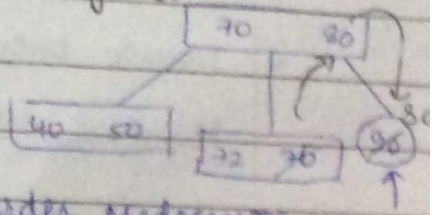


→ we can't directly delete node bcoz parent needs to have atl 2 or 3 children

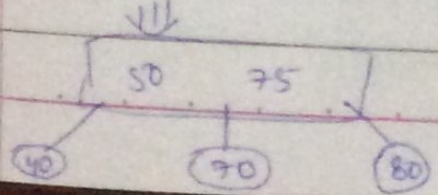
Use either In-order successor (immediate) or predecessor from parent and sibling. (Parent is single valued)

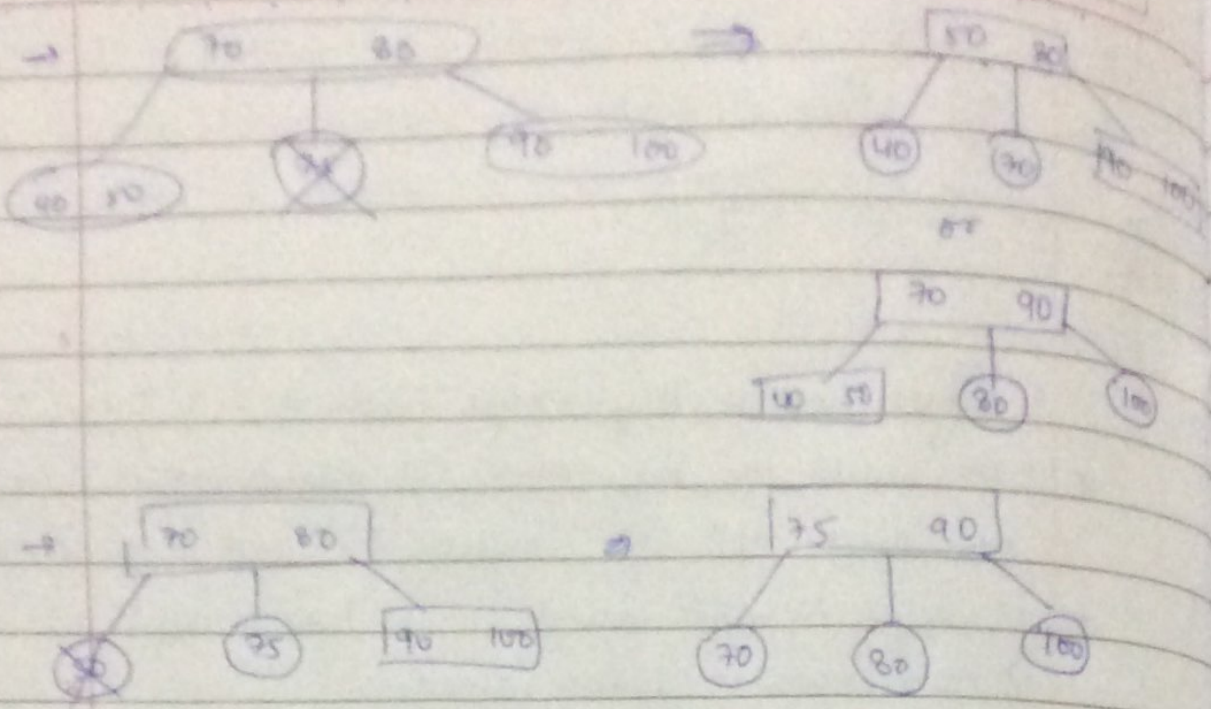
B. sibling is double valued, Parent is double valued : → have two sibling nodes





Use inorder predecessor or successor



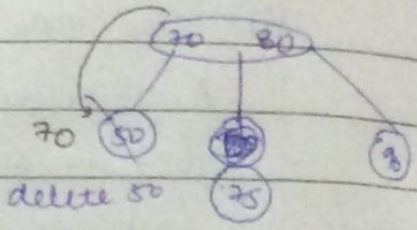


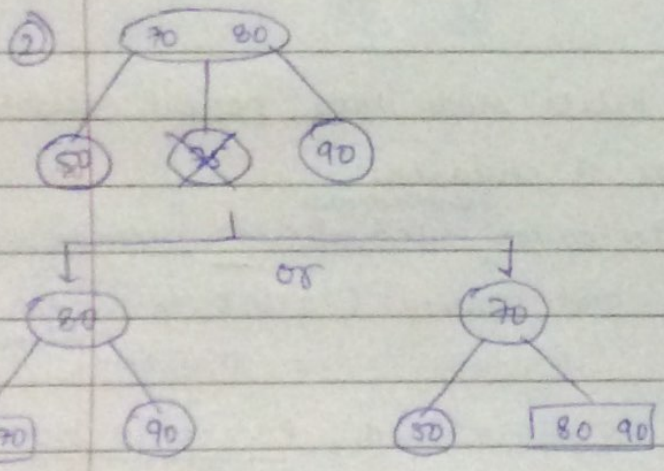
C. No sibling is double valued :

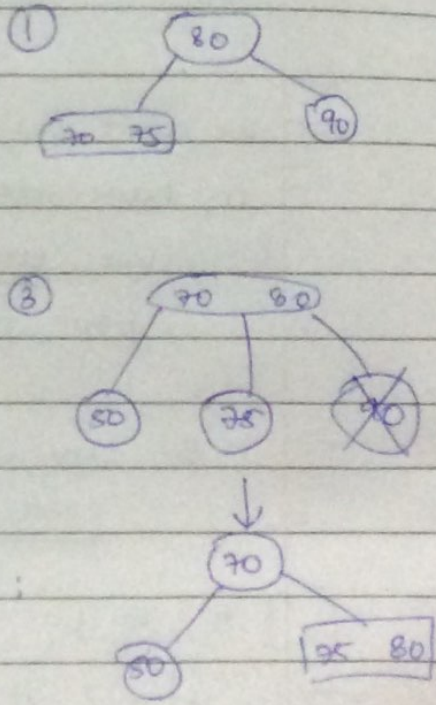
1) Parent is double valued

delete 1 of 4 nodes : can't be double valued parent

⇒ parent will become single







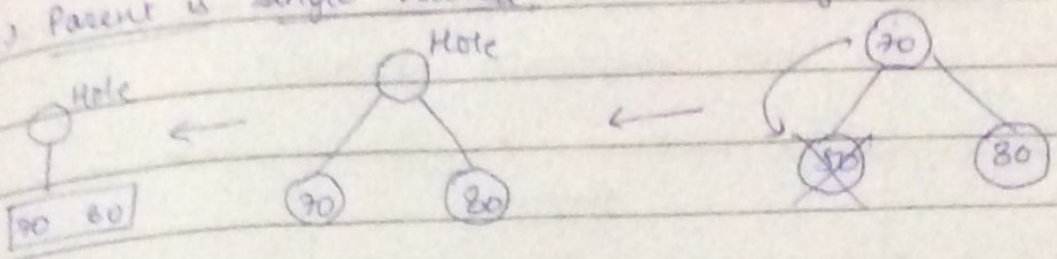
⇒ Parent becomes single-valued ;

Merging takes place

* Hole is always in the parent
* 'Hole' is temporary, so it can have single chain

Camlin Page
Date / /

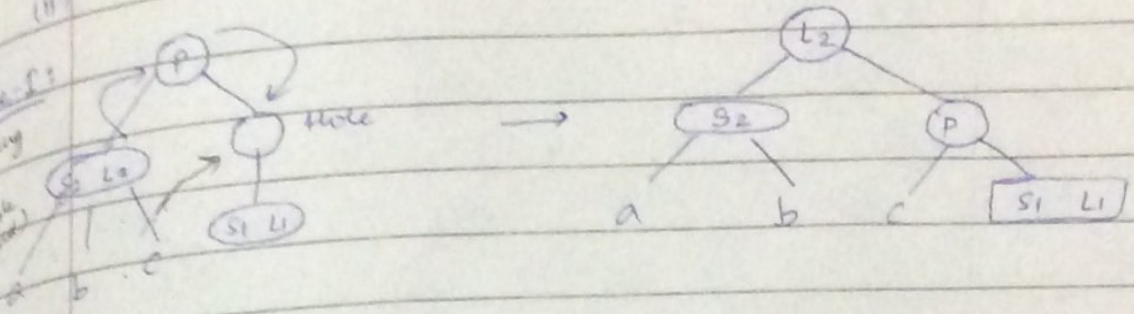
2) Parent is single valued ⇒ 1 sibling



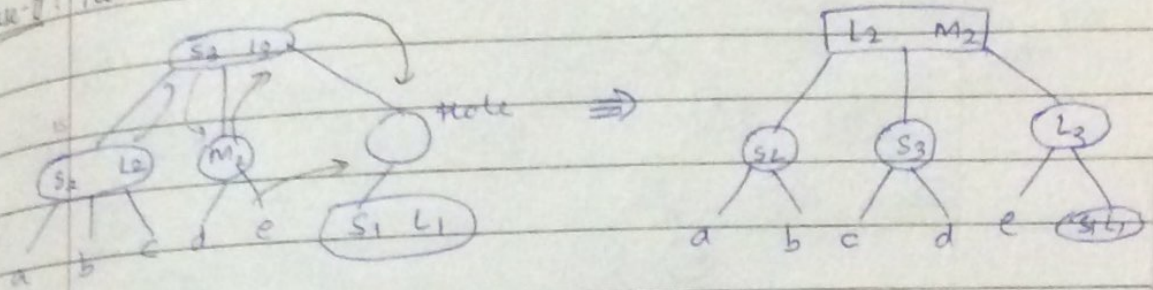
(i) Parent becomes 'Hole'

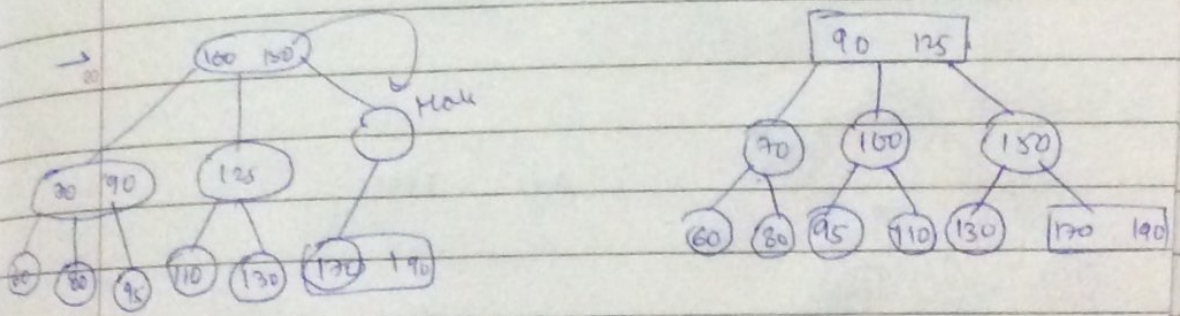
(ii) A double value is the child ⇒ Hole disappears

Case-I: sibling (double valued)



Case-II: Hole has 2 siblings





Case-I: Parent : single valued Sibling : Double valued
⇒ Hole disappears

Case-II : Parent : double valued Sibling : atleast one is double valued.
⇒ Hole disappears (In-order traversal)

Case-III : Parent : double valued sibling : both single valued
⇒ Hole disappears (Merging takes place)

Case IV Parent : single valued Sibling single valued

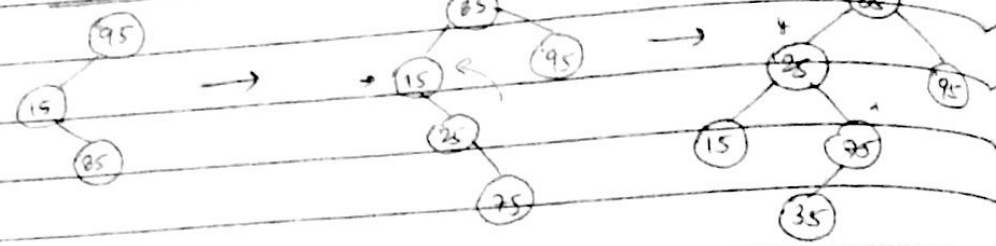
* hole propagates upwards

Revision

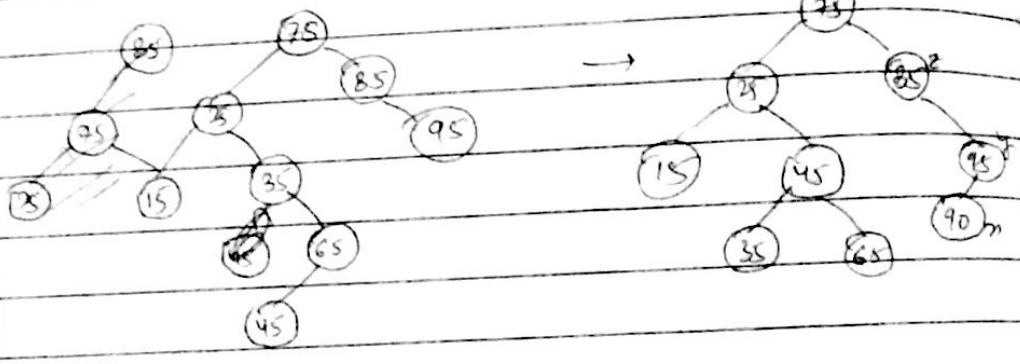
Given a sequence : 95, 15, 85, 25, 75, 35, 65, 45, 90, 10, 80, 20, 70, 30, 60, 40

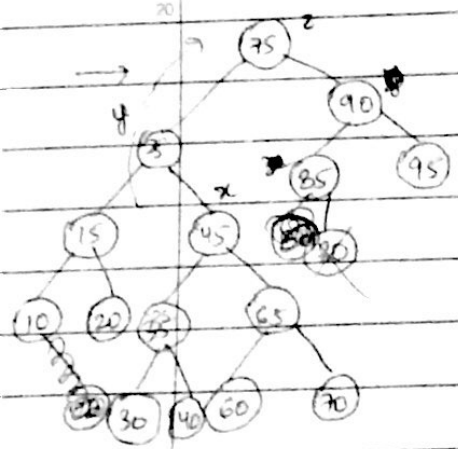
construct an AVL Tree and a 2-3 tree

AVL Tree :-



D.R. →

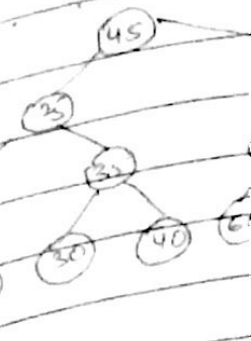




AVL Tree

Q If we delete 80, what will be the rotn ?

Left Right rotn

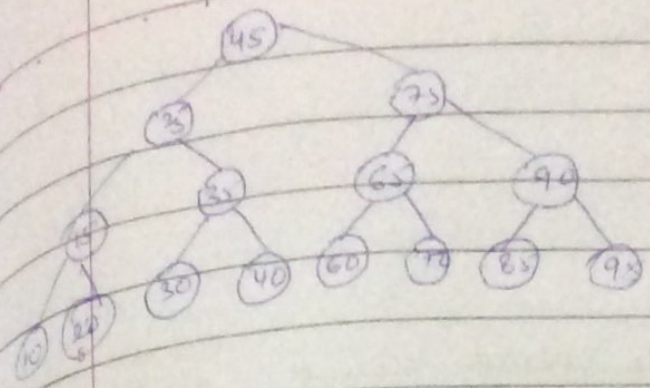


2-3 tree

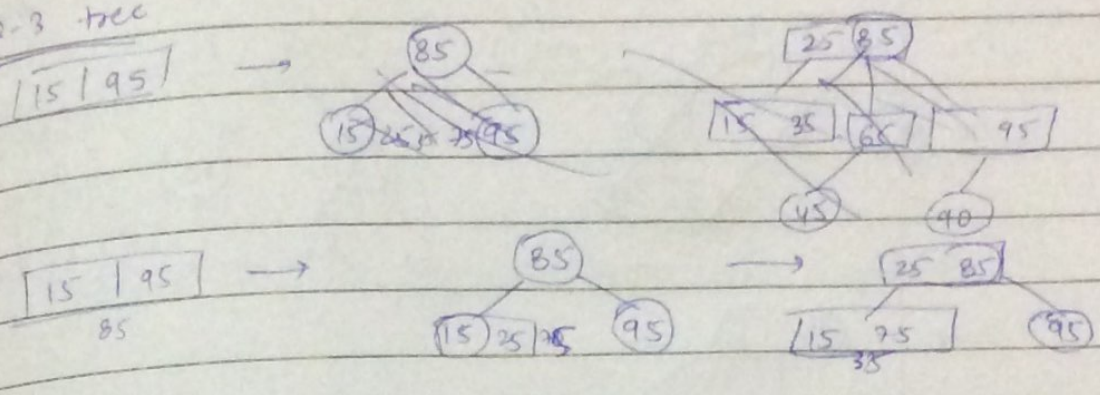
Time complexity : → depth

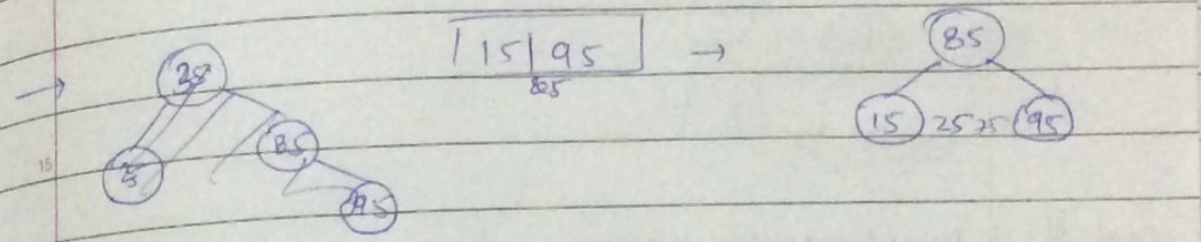
$\log_3 n \leq D \leq \log_2 n$ → depth

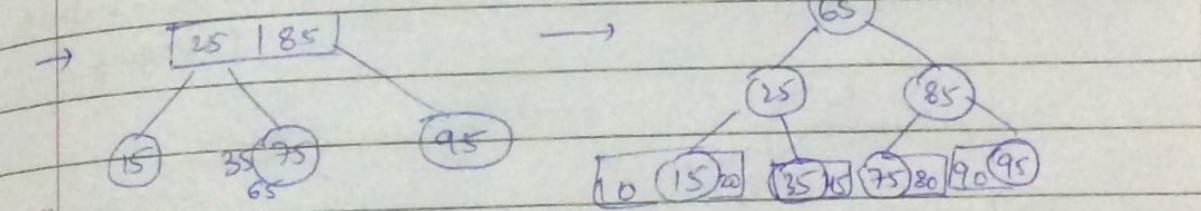
Camlin Page
Date / /

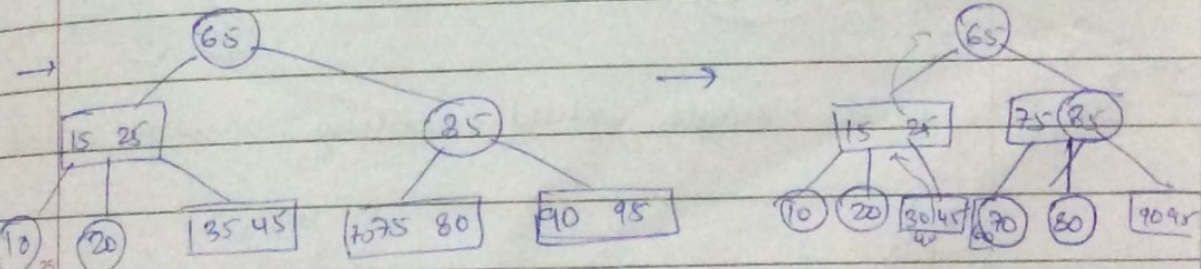


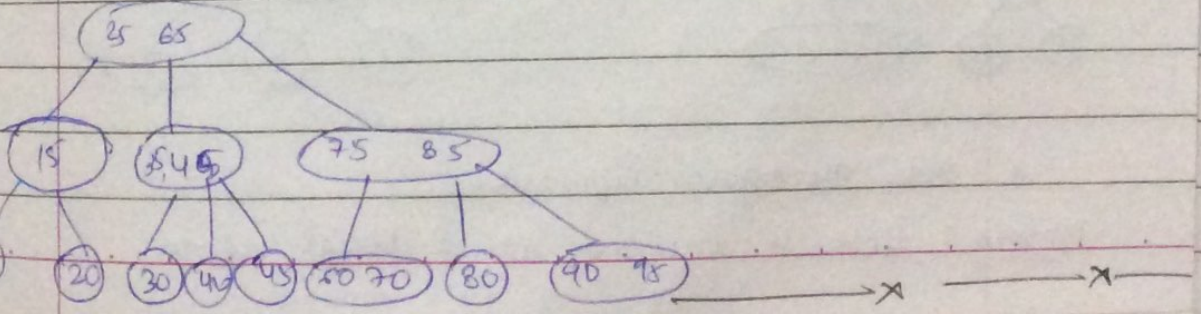
2-3 tree











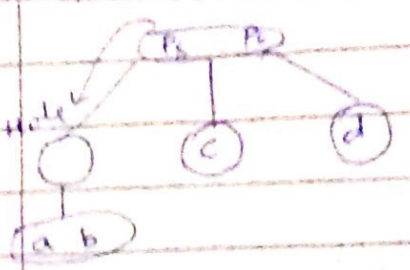
first hole is created at a level higher than leaf node.

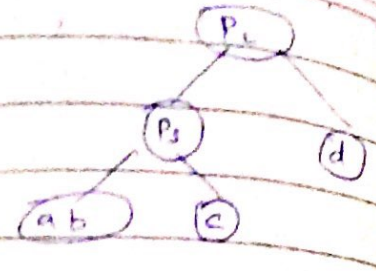
Deletion continues...

Hole:

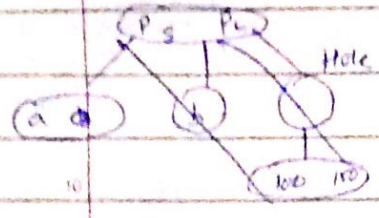
eg.

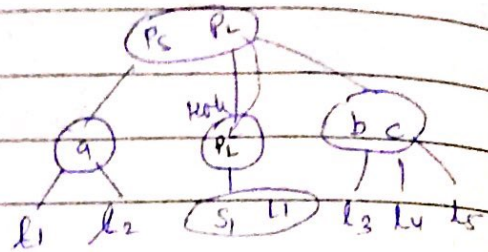
parent single valued

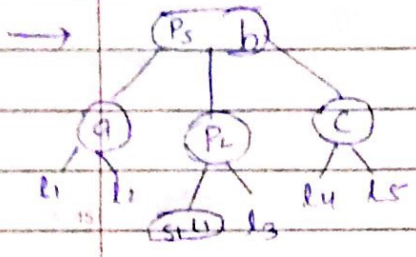




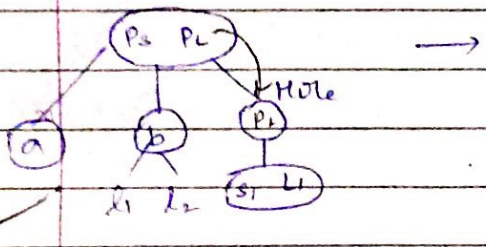
Case-II parent: single double valued siblings: one is single-valued double

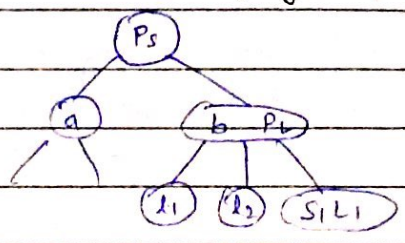






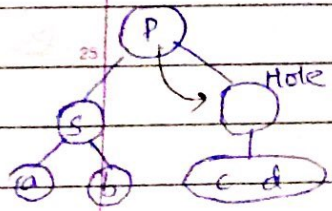
Case-III: parent: double valued siblings: single-valued. (Merging)

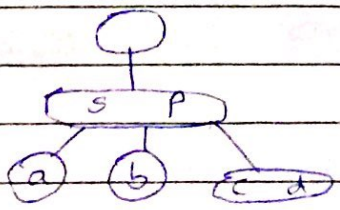




SB 3-12

Case-IV: parent: single valued sibling: single valued.





→ hole propogates upwards

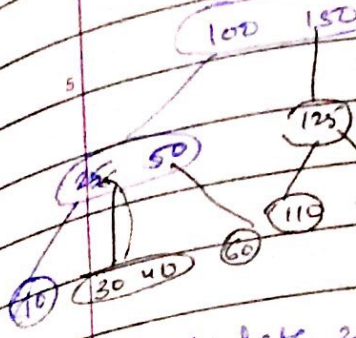
Terminal: hole is created at a level above.

* Eventually, in case

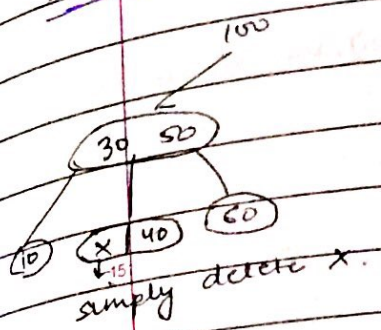
Deletion of I

1) Using in ord

2) Delete acc. t



X = 125 (delete 2

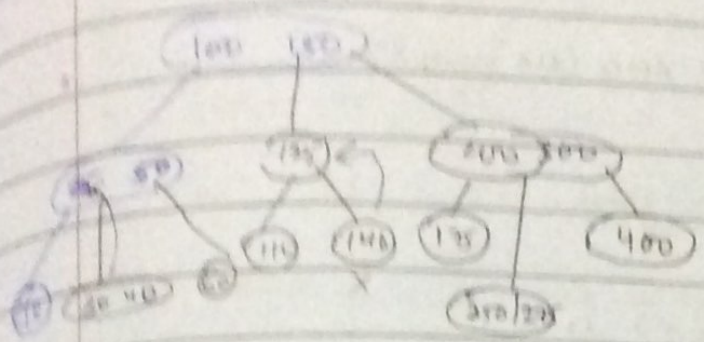


simply delete X.

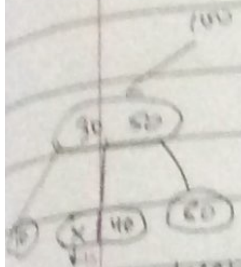
Eventually, in case IV, hole propagates to root

Deletion of Internal node key :

1) Using in order successor, propagate key to leaf node
2) Delete acc. to leaf node



X = 25 (delete 25)



simply delete X

X = 125

Hole gets created

X = 100

inorder successor = ~~100~~ 110

(immediate successor / predecessor)

(Hole gets created)

If double node : ~~leave as it is~~

it will still remain 2-3 tree.

Bubble Sorting

```
sorted list[] = Bubble_Sort (USL[], N)
{
  int J = N-1 ; boolean did_swap = true ;

  while ( (J>0) AND (did_swap) )
  {
    did_swap = false ;

    for (K=1 to J) do
    {
      if (USL[K] > USL[K+1])
      { swap (USL[K], USL[K+1]) ;
        did_swap = true ; }
    }
    J = J-1 ;
  }
  return (USL[]) ;
```

⇒ Worst case : Array is in reversed order.

Best case : Array is already sorted.
↳ no. of iterations = 1
no. of comparisons : n-1

* Worst case Time Complexity : $O(n^2)$

Best case Time Complexity : $O(n)$

Insertion

35 45

comparison & come do

35 45

(1)

5 35

5 25

1st is placed

Best case :

→ tot

Worst case

→ Time

35 75

→ Divide

→ Apply

5

→ NOW

→ It will

till

Then

Camlin Page
Date / /

Insertion Sorting

35 45 5 95 25

Put 1st ele as it is.
Compare the next one & put it in the array.
The array formed at every step is sorted.

35 45 5 95 25
↓ comparison & comes down
35 45 5 95 25
① ② ③
5 35 45 95 25
5 25 35 45 95 ✓ sorted

1st is placed as it is next (n-1) elements

1) Best Case : Sorted, each next ele is compared 1 time (with last one)
→ total (n-1) comparisons ⇒ O(n)

2) Worst case :
Total elements : (n-1)
Avg comparisons : n/2 { n-1, n-2, ... }

→ Time complexity : $O(n^2)$

Merge Sorting

35 75 55 5 | 95 65 45 25

→ Divide it into 2 arrays : left and right array
→ Apply merge sort on both sides

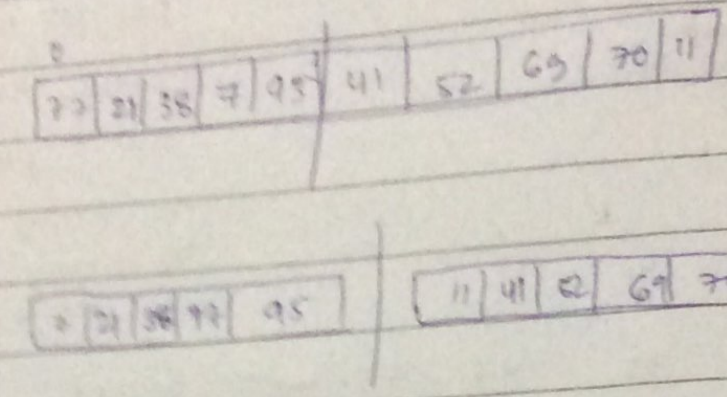
5 35 55 75 | 25 45 65 95

→ Now merge both sorted arrays.

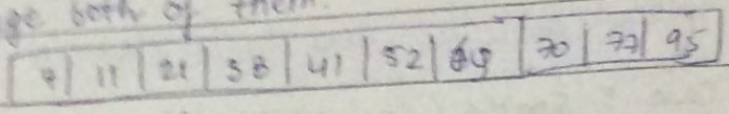
Merge Sort
→ It will perform recursion calling merge sorting again till only 2 elements are remaining
Then start merging

Time complexity for both best & worst case: O(n log n)

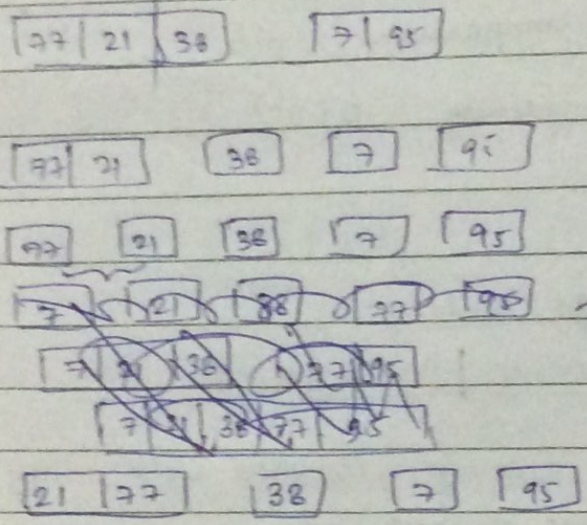
Eg.



→ Merge both of them:



→ Divide each divided segment again & perform merge sorting.



→ Merge now

I/P : unsorted array of $n \geq 1$ integers

O/P : sorted array

→ Base case:

single element ($n = 1$), return

Array size (1 to n)

Algo:

```
(1)  Merge_sort (A,n)
(2)  if n=1 return (A) ;
(3)  else {
(4)  If (N is an odd no.)
(5)  then { n1 = (n-1)/2 ; n2 = n-n1 ; }
(6)  else { n1 = n/2 ; n2 = n/2 ; }
(7)  create arrays L[n1] ; R[n2]
(8)  for (i=1 to n1) do L[i] = A[i];
(9)  for (j=1 to n2) do R[j] = A[j+n1] ;
(10) L[ ] = Merge_sort (L, n1) ;     : already taking n1 as an arguement, so size already defined.
(11) R[ ] = Merge_sort (R, n2) ;
(12) A[ ] = Merge (A, L, n1, R, n2) ;
(13) return (A) ;
```

At the end of while loop either I = n1+1 or J = n2+1 (both can't be possible as only one will be incremented in last while loop)

```
A[ ] = Merge (A, L, n1, R, n2) ;
{
int I=1 ; J=1 ; K=1 ;
while ((I ≤ n1) && (J ≤ n2)) do
{
  if (L[I] <= R[J])
    then { A[K] = L[I] ;
           K = K+1 ;
           I = I+1 ;
         }
    else {
           A[K] = R[J]
           K = K+1 ;
           J = J+1 ;
         }
}
```

April 15

```
if (J ≥ n1)
{ while (J ≤ n2) do
    { A[K] = R[J];
      K = K+1;
      J = J+1;
    }
}
else while (I ≤ n1) do
    {
      A[K] = L[I];
      K = K+1;
      I = I+1;
```

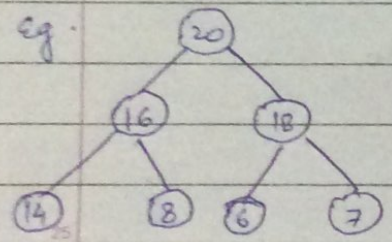
6-4-17 **Heap Sort**

Heap: A large area of memory from which a programm can allocate blocks as needed & deallocate them when no longer needed

or A balanced, left-justified binary tree in which no node has a value greater than the value in its parent

→ Heap sort uses 2nd definition.

Eg:



: Max heap

Min heap : children have greater value than parent

→ A binary tree of depth n is balanced if all the nodes at depths 0, 1, 2, through (n-2) have 2 children.

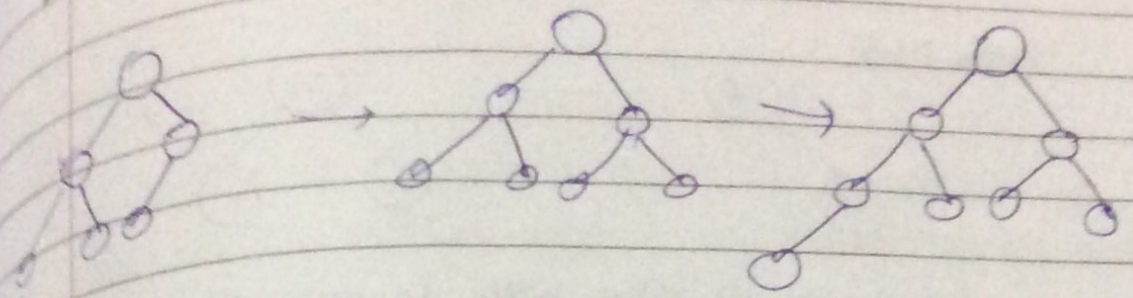
→ at (n-1) : every node has 2 children : Complete B.T.

→ at (n-2) : every node has 2 children : Heap

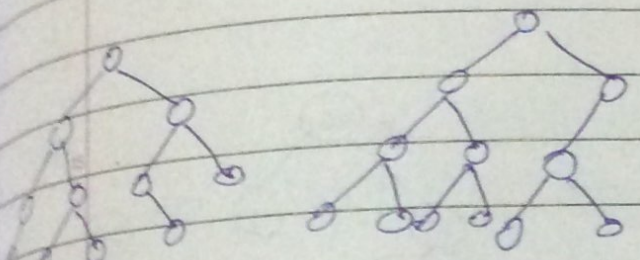
Camlin Page
Date / /

left justified.

* Any time you add a node, it has to be leftmost.
* left justified is mandatory cond^n.

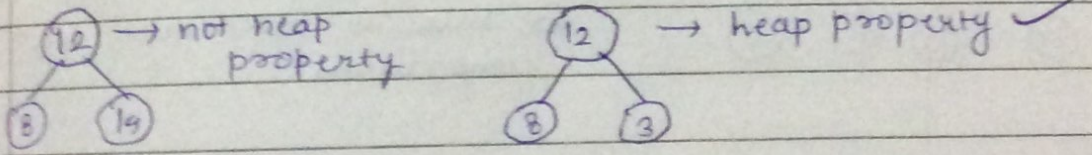


* Before depth ↑es, it should become complete binary tree.
* Delete will also take place for latest node added



left justified, not balanced

* A node has heap property if node value is larger than value in its children.

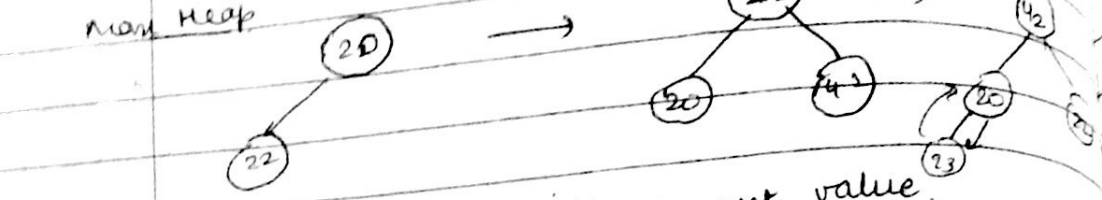


* All leaf nodes automatically have heap property
* A B T is a heap if all nodes have " "

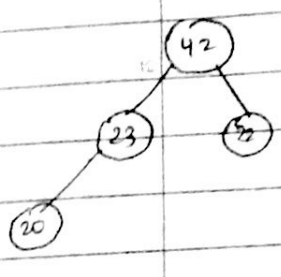
Sift Up

To convert a node that doesn't have heap property into heapified node

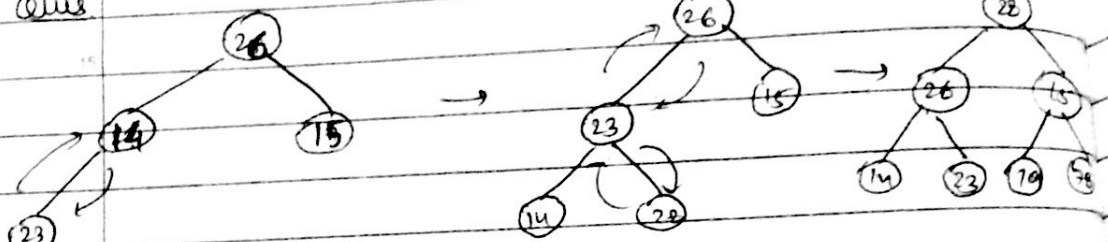
Max Heap

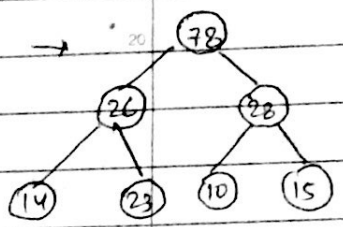


* exchange its value with largest value
* Other children are not affected



Ques



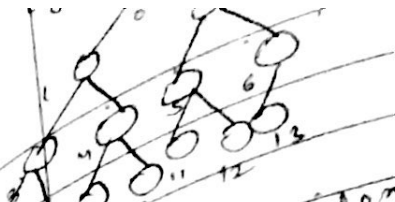


* If we add, max. time needed to reach its position (root) = $\log_2 n$ (max)

We have to make max. $\log_2 n$ comparisons

* Heapified doesn't mean sorted
* This can be also used as Priority Queue

(If we remove root & reheapify, we will get next max. value)



parent index
children ind
↓
In case it is
each node's

$\log_2 10^6$ =

→ always inde

eg. [22 | 71 | 10]
0 1

Algo to
B[] :=
{
int I
if (N
B[0] =
for (I
{
B[
}
retu

b) For ab

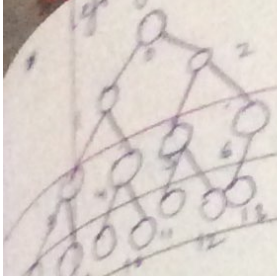
22	71
0	1

Why justified, balanced B.T. :-

indexes will be continuously occupied.

⇓

packed array.



parent index = i

children index = 2i+1 & 2i+2

↓

In case it is heapified, along with packed array, each node's children have lower value than the node

$\log_2 10^6 = 6 \log_2 10 = 6 \times 3.2 = 20$

→ always index 0 will have largest value

22	71	19	65	7	83	24	45
0	1	2	3	4	5	6	7

N = 8

Algo to get Heapified array :

```
B[] := Build ( A[], N, B[])
{
  int I = 1
  if (N ≤ 1) return (A[]);
  B[0] = A[0];
for (I = 1 to (N-1); I++) do
  {
    B[] := Insert_new-node ( A[I], I, B[I])
  }
  return (B[]);
```

For above eg :-

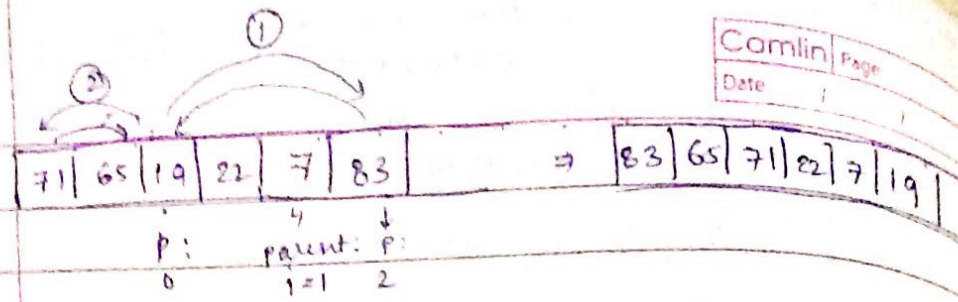
22	71
0	1

⇒

71	22	19	65
1		2	3

parent: 0

Parent: i=1

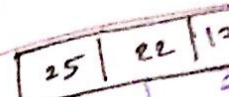


Algo to insert new node

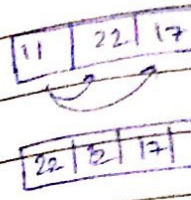
```
B[ ] = Insert_new_node (Ele, I, B[])
{
  int parent_inden, boolean Propogation_required := true
  B[ I ]: = ele
  if (I is odd) parent_inden = (I-1)/2
   else.  parent_inden = (I-2)/2

while((parent → inden) >=0 && Propogation_required) do
  {
    if  B[I] > B[parent_inden]
      { swap ( B[I], B[parent_inden])
          I := parent inden;
      }

  if  (I = =0)  Propogation_required = false
  else  (I = odd )  parent_inden = I-1/2
        else        "    "    = I-2/2
    }
    else propogation_required = false;
}
  return B [];
```

12-4-19

* Adding
not left-just
* complexit



Sorting :
→ Rep Replace
than array
(Max heap
→ Perform
index. &



* sifting u
sifting d
*15 Again

→ each el
→ total

① Remove
② Reheap

Heap
B = H
{
int
in
B[

* Adding a node can only destroy heap property, not left-justified, height balanced
* Complexity of Insertion $O(\log_2 n)$

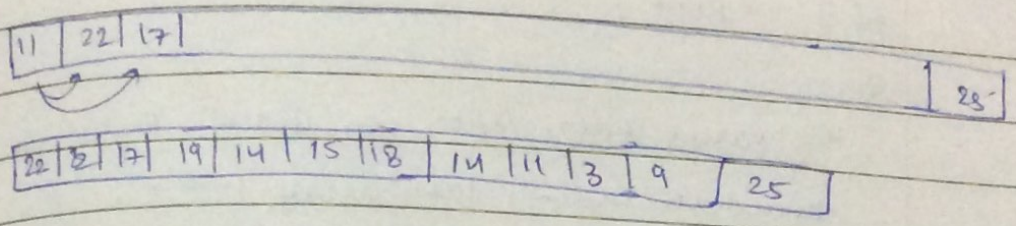
25	22	17	19	22	14	15	18	14	21	3	9	11
0	1	2	3	4	5	6	7	8	9	10	11	12

Sorting :

→ Rep- Replace 1st & last element data

than array is only unsorted upto $(n-2)^{th}$ index. (Max heapified root)

→ perform sift down and bring max. value to 0 index & it again becomes heapified binary tree.



* sifting up : heapification → $O(\log_2 n)$

sifting down : re-heapification → $O(\log_2 n)$ anytime → each element

* Again, replace 1st & last 2nd element.

→ each element take $\log_2 n$ time

→ total time complexity = $O(n \log_2 n)$

① Removing & replacing the root.

② Reheap & repeat

Heap Sort Algo

```
B = Heap_Sort (A[], N);
{
  int array B [0: N-1];
  int I, Array_size;
  B[] = Build_Max_HLB_Binary_Tree (A[], N,
                                   B[])
```

Ques. Write heap sort in your own language.

```
{ for (I=(N-1) to 2 in steps of (-1)) do
  { swap{B[0], B[I]};
    Array_size = (I-1);
    B[] = Root_Downward_Propogation (B[], Array_size)
  }
  return (B[]);
}

B[] := Root - - -
{
  int parent_index, left_son_index, Right_son_index,
  Candidate_index, last_array_index;
  boolean
```

* In heap sort, O(complexity) may be less than $O(n \log_2 n)$, unlike mixed sort.