implemented using pointers

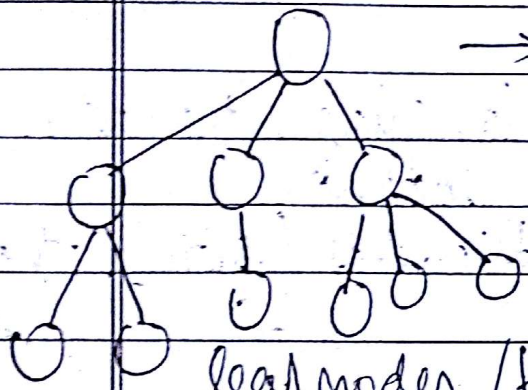
Page No.
Date

Ch-6 (Drozdek)

BINARY TREES (Binary Search Trees ↓ spl case of BST).

- hierarchial ~~tree~~ structure

Empty tree (root == NULL)

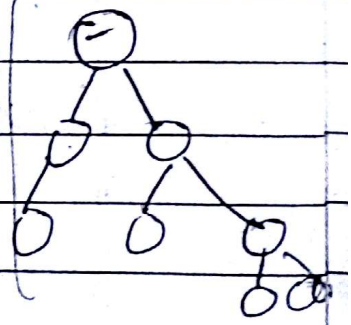


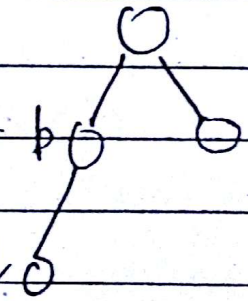
→ root

(Normal tree)

leaf nodes / terminal nodes.

```
template <class T>
class node
{public: T data;
    node * left;
    node * right;
};
```





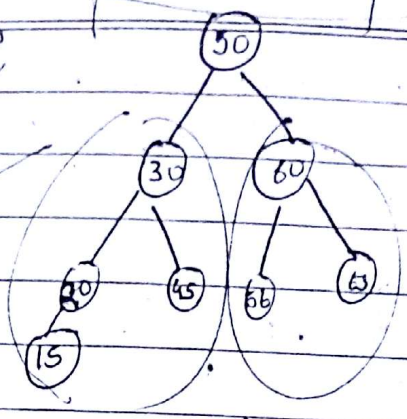

```
p->left = q;
p->right = NULL;
```

→ Root does not have any parent.

→ leaf node has no child

Condition for Binary Search Tree

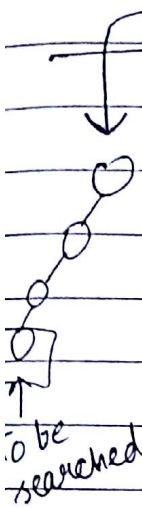
Binary Search Tree



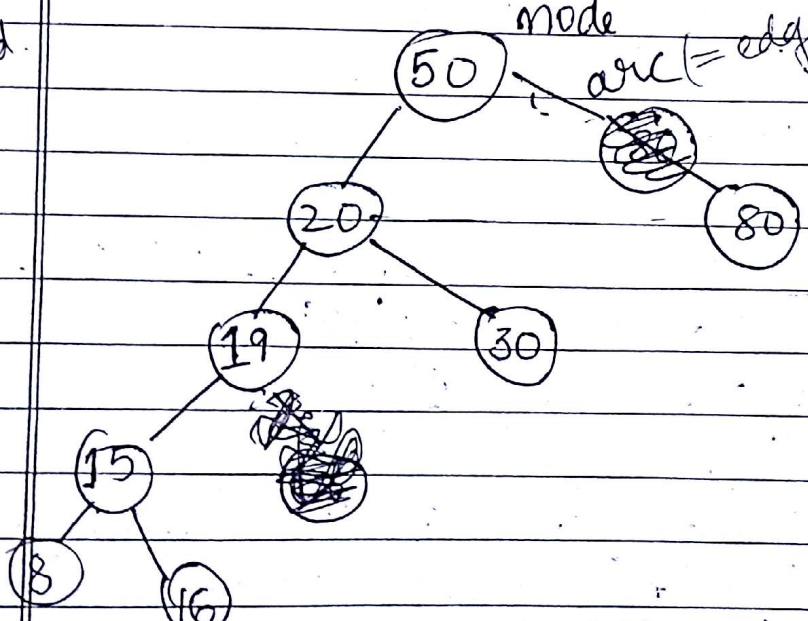
[Root's left should be less than 50

[Root's right sub tree should be less

worst case time can be of order n (unbalanced Tree).
avg time is better (if tree is balanced)



Q. Create a BST out of given data.
50, 20, 30, 80, 19, 15, 16, 8



There is a unique path to every node & no. of arcs we encounter is path of the length

Tree (Recursive defn)

→ An empty structure is an empty Tree

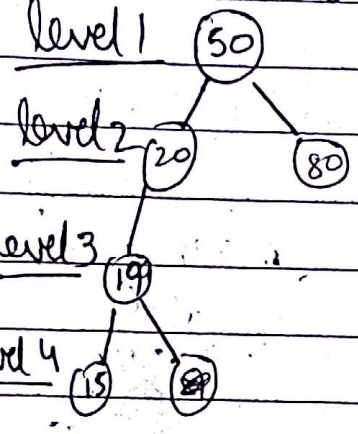
→ If $T_1, T_2, \ldots, T_k$ are disjoint trees, then the structure whose root has k children as the roots of $T_1, T_2, \ldots, T_k$ is also a tree.

→ Only structures generated by Rule ① & Rule ② are Trees / nothing else is a Tree

→ Path - each node has to be reachable from the root through a unique sequence of arcs called a path.

→ The no. of arcs in the path is called the length of the path

→ The level of a node is the length of the path from the root to the node + 1



19 - path length = 2

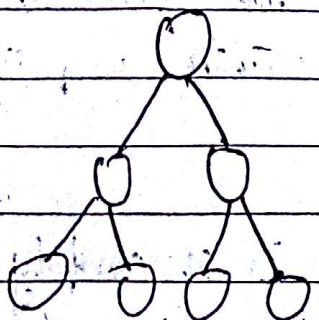
level of 19 = 2+1 = 3.

height = 4

ht
w

→ Height of a non-empty tree is the max. level of a node in the tree.
empty tree (ht. = 0), single node (ht. = 1)

→ In all binary trees, there are atmost 2^i nodes at level $i+1$ (or 2^{i-1} nodes at level i)



$2^0 = 2^{i-1}$ Level 1 $i=1$

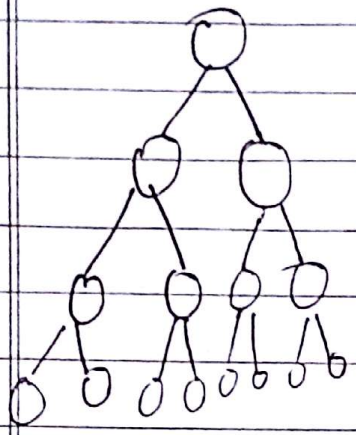
$2^1 = 2^{i-1}$ Level 2 $i=2$

$2^2 = 2^{i-1}$ Level 3 $i=3$

$2^{(i+1)-1} = 2^i$ $i=i+1$

Case for complete binary tree.

The tree having if all the nodes at all the levels except the last has 2 children, then that would be a complete binary tree.

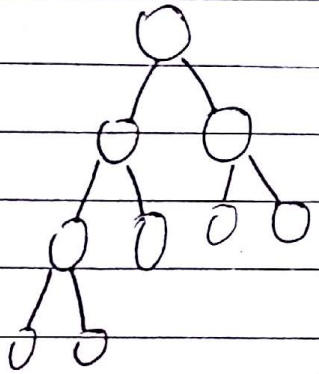


full m-ary

Complete BinTree

complete Binary tree

In ~~a~~ all non-terminal nodes have both their children. and all leaves are at same level.



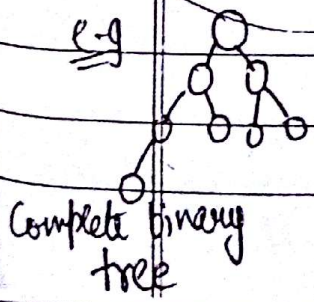
(NOT a complete binary tree)

neither full nor complete

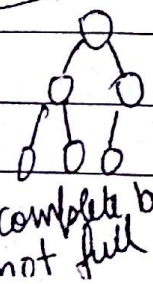
⇒ Full Binary tree ⇒ is a tree in which every node other than the leaf has 2 children. Also called Proper binary tree.

⇒ Complete binary tree ⇒ is a "binary tree in which every level except possibly the last level is completely filled and all nodes are as far as left possible. / last level has all nodes to left side

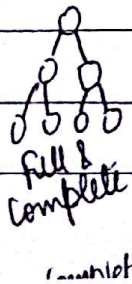
e.g



Complete binary tree

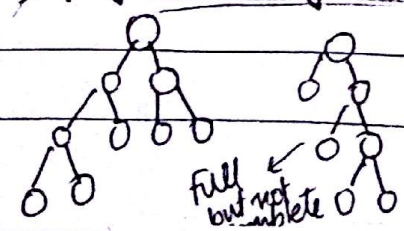


complete but not full



full & complete

e.g of full binary tree



full but not complete

15/9/16

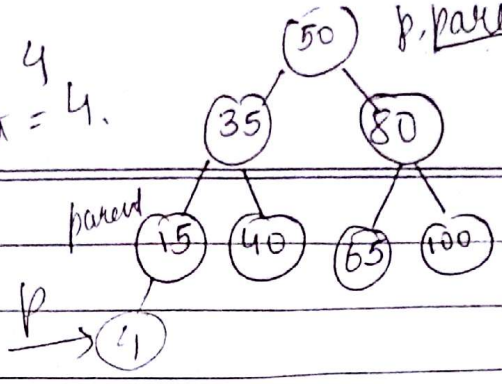
Page No.
Date

Binary Search Trees

```
template <class T>
class node{
    public: T data;
            node * left, right;
    node(){
            left = right = NULL;
    }
    node(T num)
    { data = num;
      left = right = NULL;
    }
};
template <class T>
class tree
{ ~~pr~~   node<T> * root;
    tree(){ root = NULL; }

    bool search (T num)
{   node<T> *p = root;
    if(p == NULL)
      { return false; }

    while (p != NULL)
{ if(p->data == num)
        return true;
    else if (p->data > num)
        p = p->left;
    else
        p = p->right;
}
    return false;
}
```




```
void insert (T num)
{ node<T> *temp = new node<T>;
  temp->data = num;
  node<T> *p = root, *parent = NULL;

  while (p != NULL)
  {
    if (num == root->data)
    { cout << "In Invalid";
      exit(0); return;
    }
  else
  { parent =
  while (p != NULL)
  { parent = p;
    if (num < p->data)
        p = p->left;
    else
        p = p->right;
  }
    if (root == NULL)
      root = temp;
  else if (num < parent->data)
      parent->left = temp;
  else parent->right = temp;
}
```

case where num = root

This part will only take us to the point where we want to insert // empty tree case

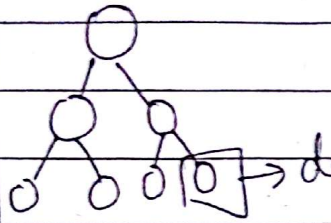
Page No.
Date

Concept

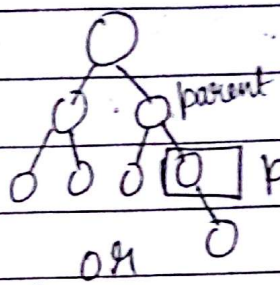
Deletion (3 cases)

1. Node to be deleted has no child
2. Node to be deleted has one child.
3. Node to be deleted has two children.

(1)



(2)

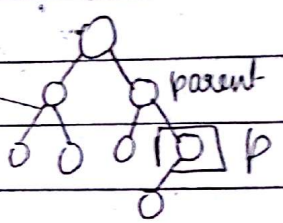


parent → left = p → left

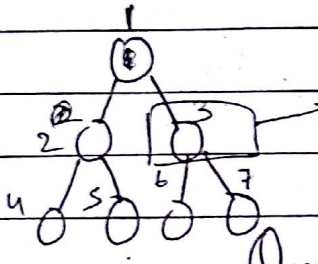
or

parent → right = p → right.

or



(3)

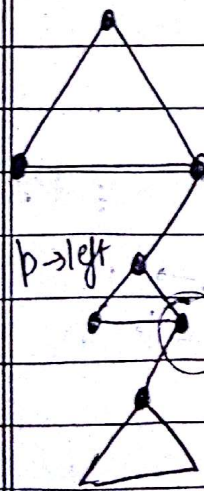


if this is to be deleted
⇒ problem

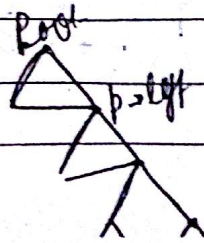
Q.. I will point to 6 or 7 ?
if 3 is deleted

Solution → (1) Deletion by merging
(2) Deletion by copying

search very next (immediate) predecessor of node to be deleted



After Deletion (delete(p))

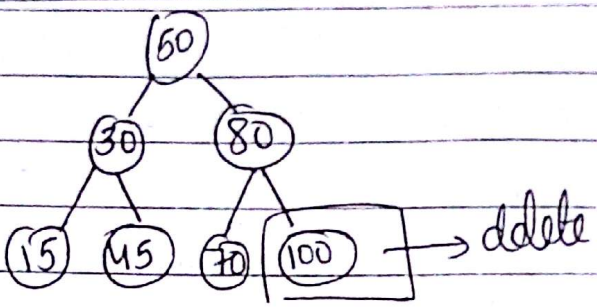


Since this is just smaller than (p=50) so the right subtree of (p=50) will be a subtree of predecessor found in previous

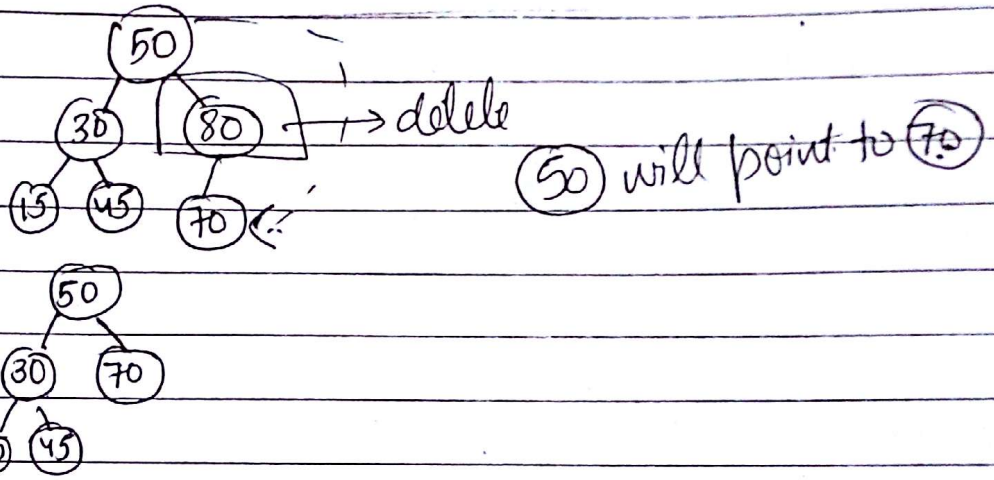
Page No.
Date

→ (p=50) will be replaced by (p→left)

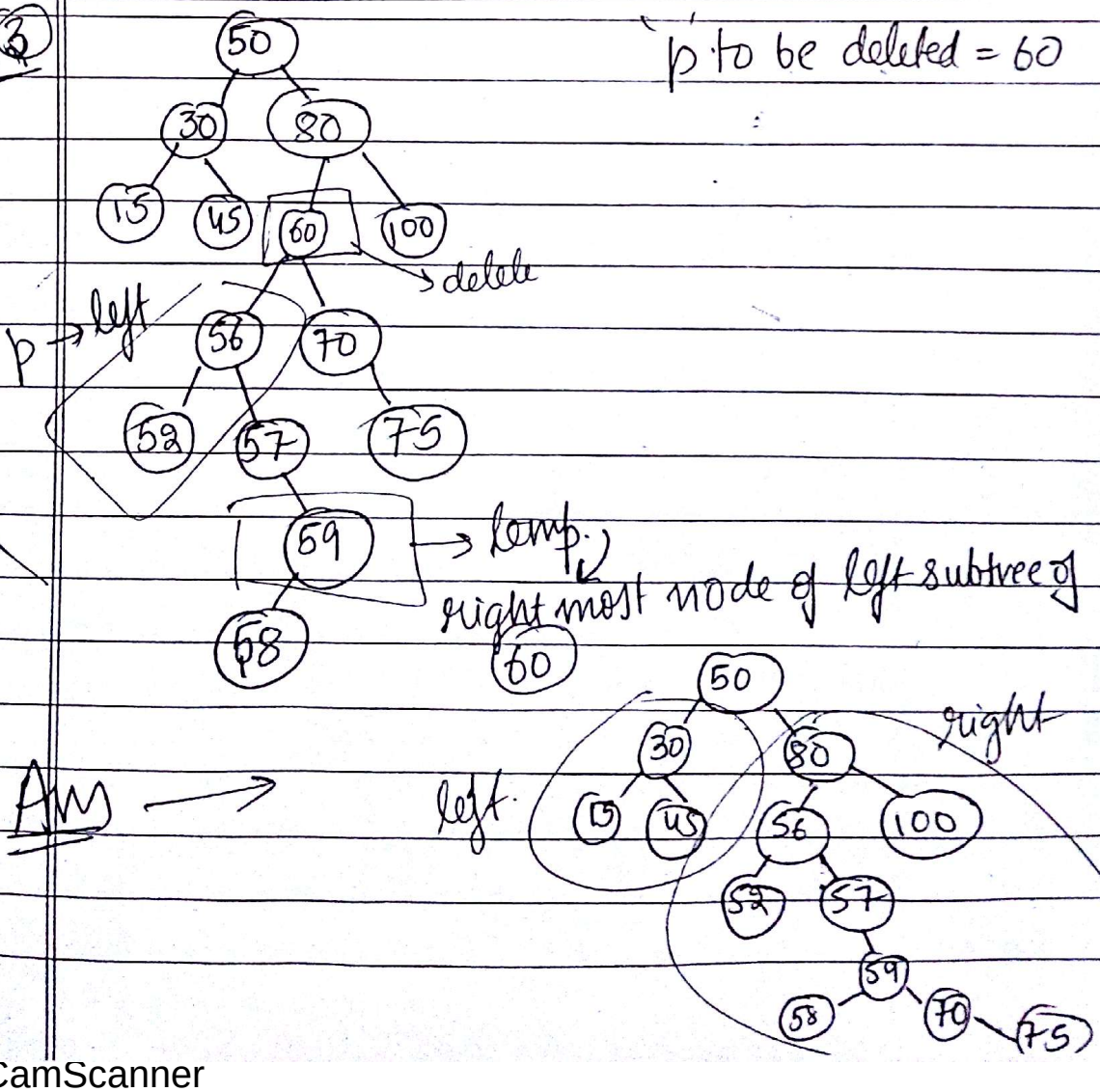
case 1



case 2

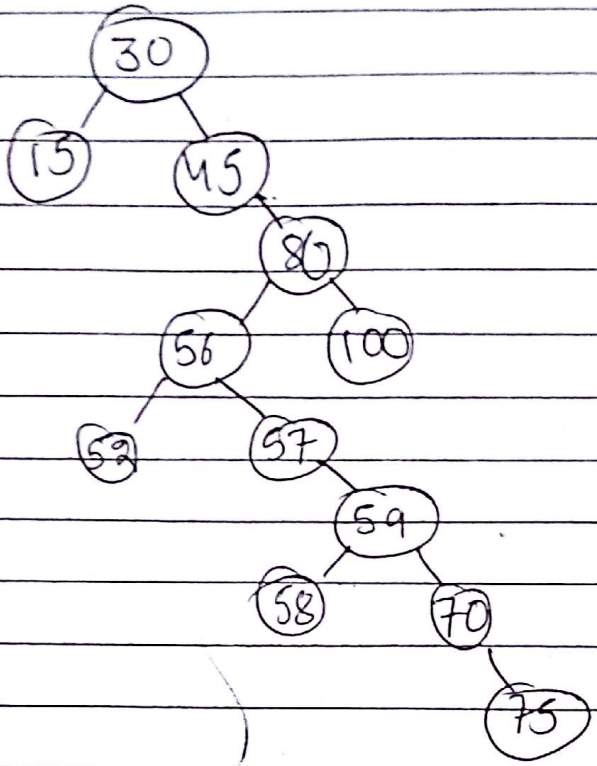


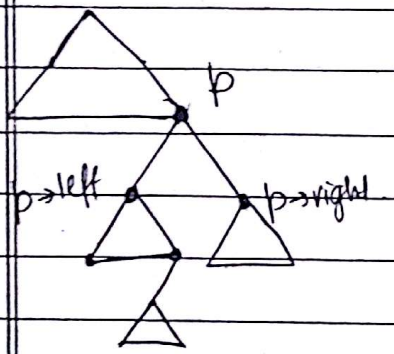
case 3



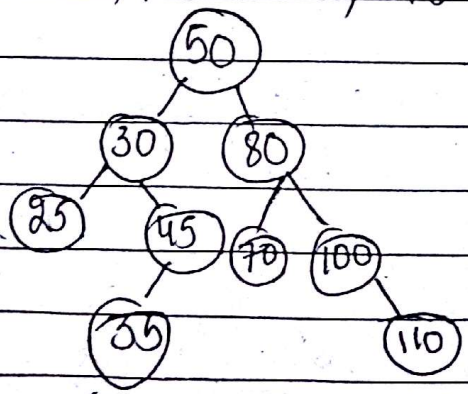
If root is to be deleted from prev Ans

Ans.





Inorder, Pre Order, Post Order.



L = Left
R = Right
V = Visit

(VLR) Pre Order : (50) (30) (25) (45) (35) (80) (70) (100) (110)

(LVR) In Order : (25) (30) (35) (45) (50) (70) (80) (100) (110)

(LRV) Post Order : (25) (35) (45) (30) (70) (110) (100) (80) (50)

Note: InOrder gives the no. in ascending order

Search, deletion, merging → Code
PreOrder, PostOrder, InOrder → Traversal

16/9/16

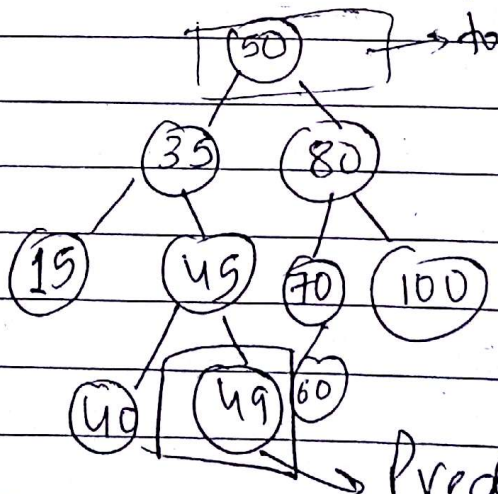
Deletion by copying

If node to be deleted has 2 children, the problem can be reduced to one of the 2 simple cases.

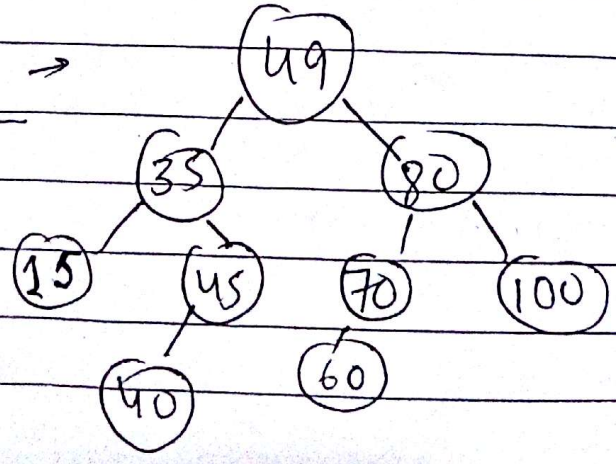
1) Node is a leaf // Node → Predecessor
2) Node have only one unknown child. // Node = Predecessor

This can be done by replacing the node being deleted with its immediate predecessor

case (1)



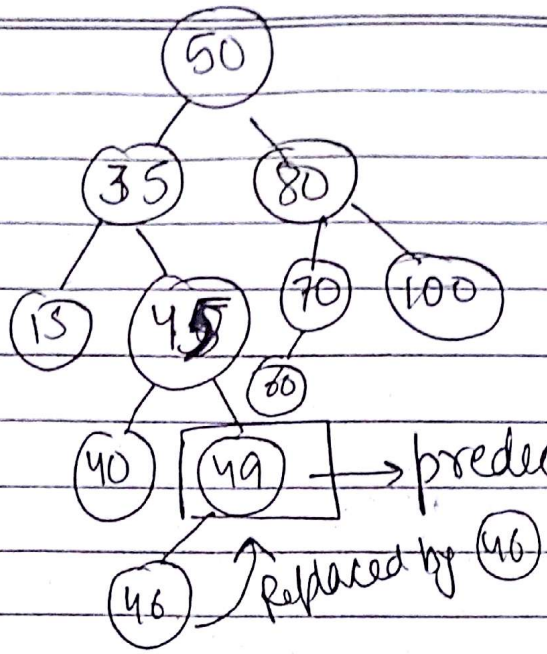
Ans →



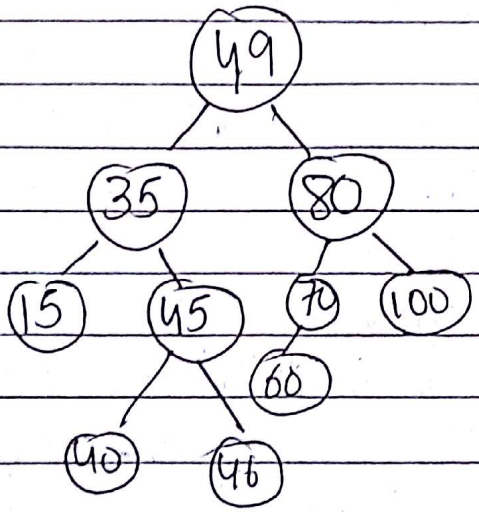
(Deletion by copying)

Page No.
Date

case (2)

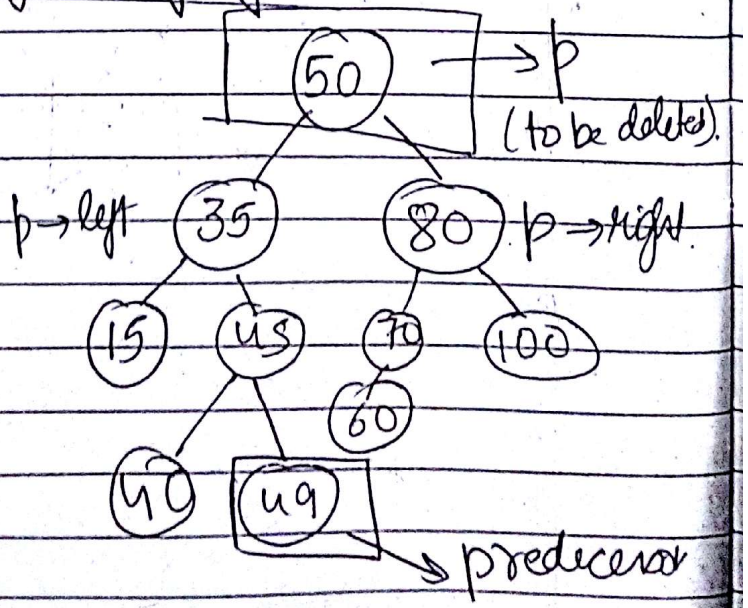


Ans



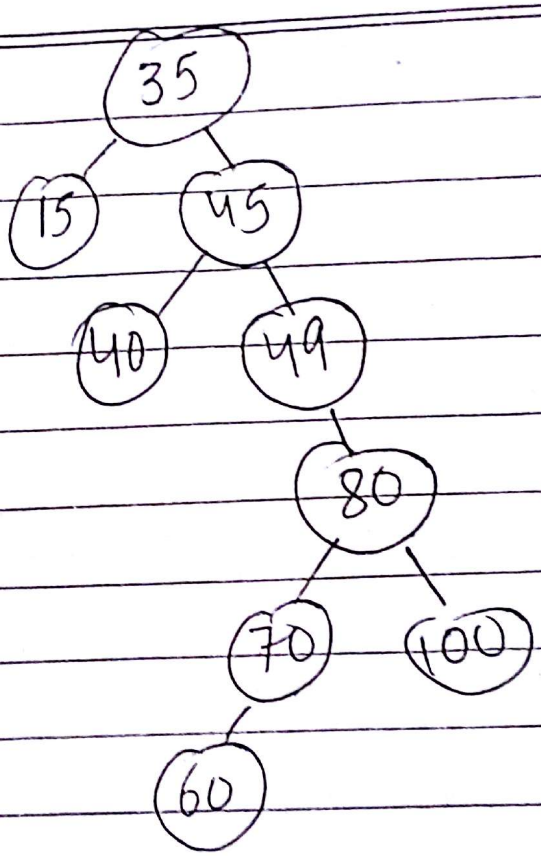
Deletion by Merging

delete (50)



Date

Ans



Read from Book

node to be deleted

call by reference

```
void deleteByMerging (node<T>*& p)
{   node<T> * temp = p;

    if ( p->right == NULL)
```

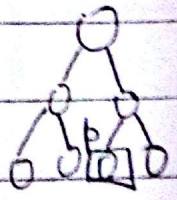


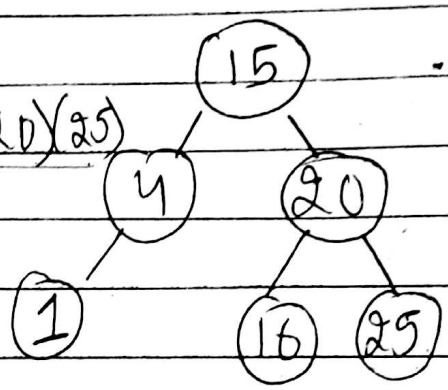
Fig 6-14 (pg 230)

LVR
left - root - right

```
void inOrder ( node<T> *p )
{
    if ( p != NULL )
    {
1.      inOrder ( p -> left );
2.      cout << p -> data;
3.      inOrder ( p -> right );
4.  }
}
```

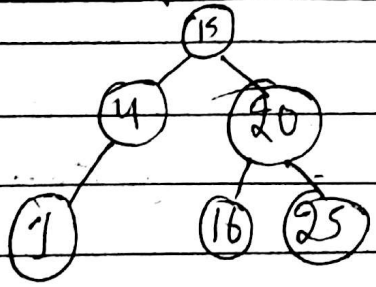
for every node 3 AR will be there 3 called.

(1) (4) (15) (16) (20) (25)



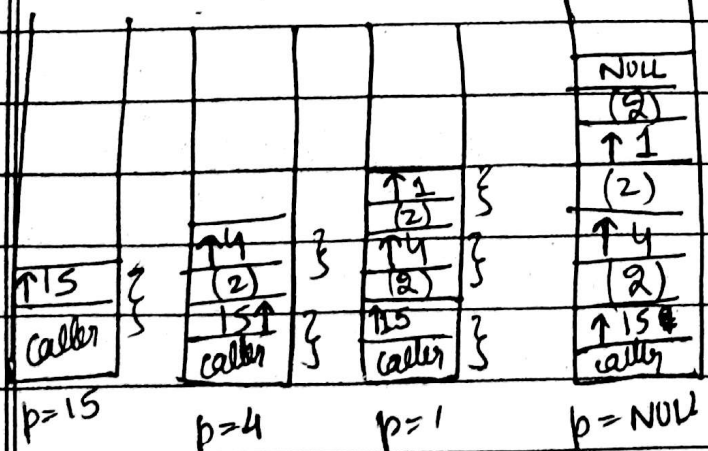
21/9/16

Stack Trace for inorder traversal

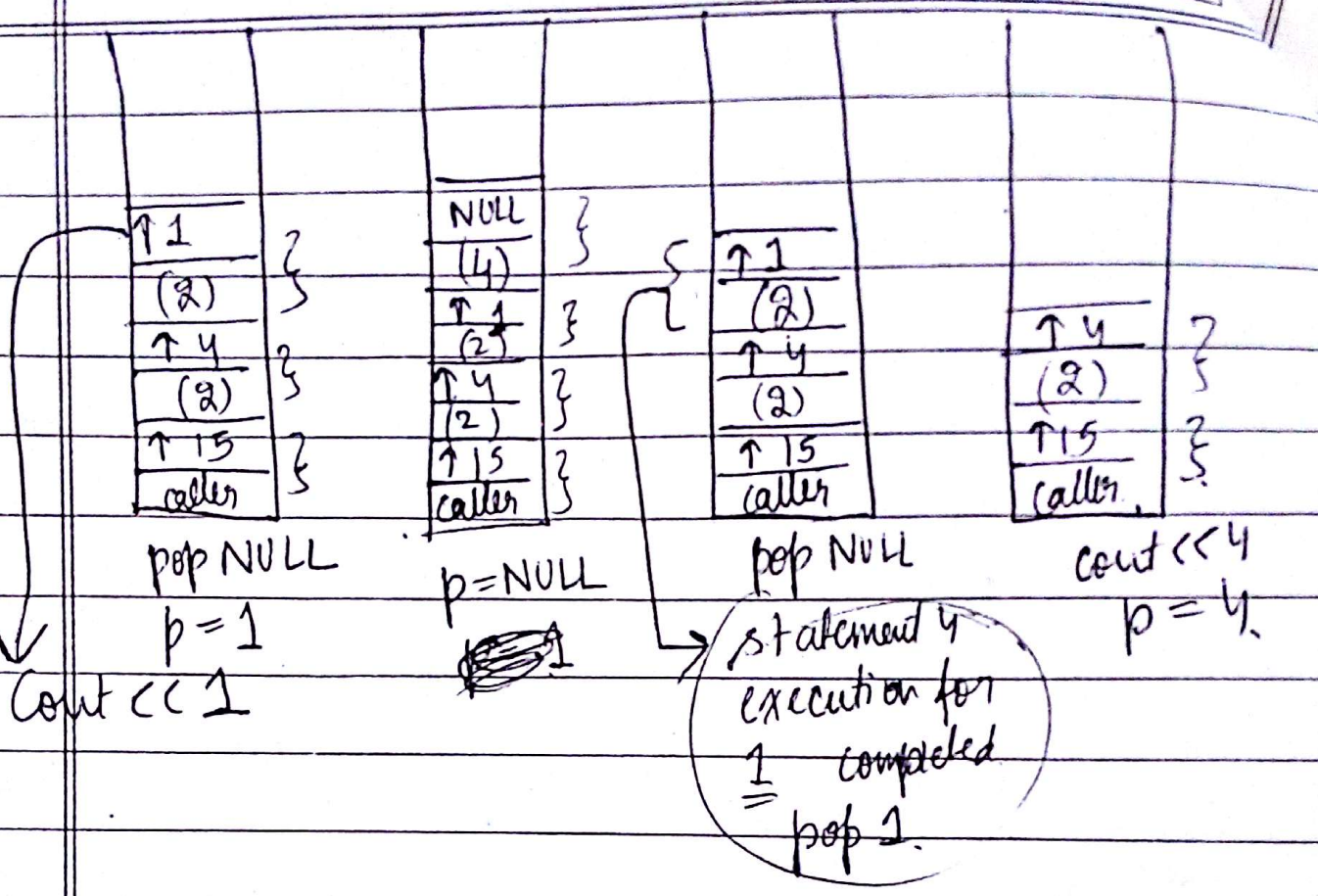


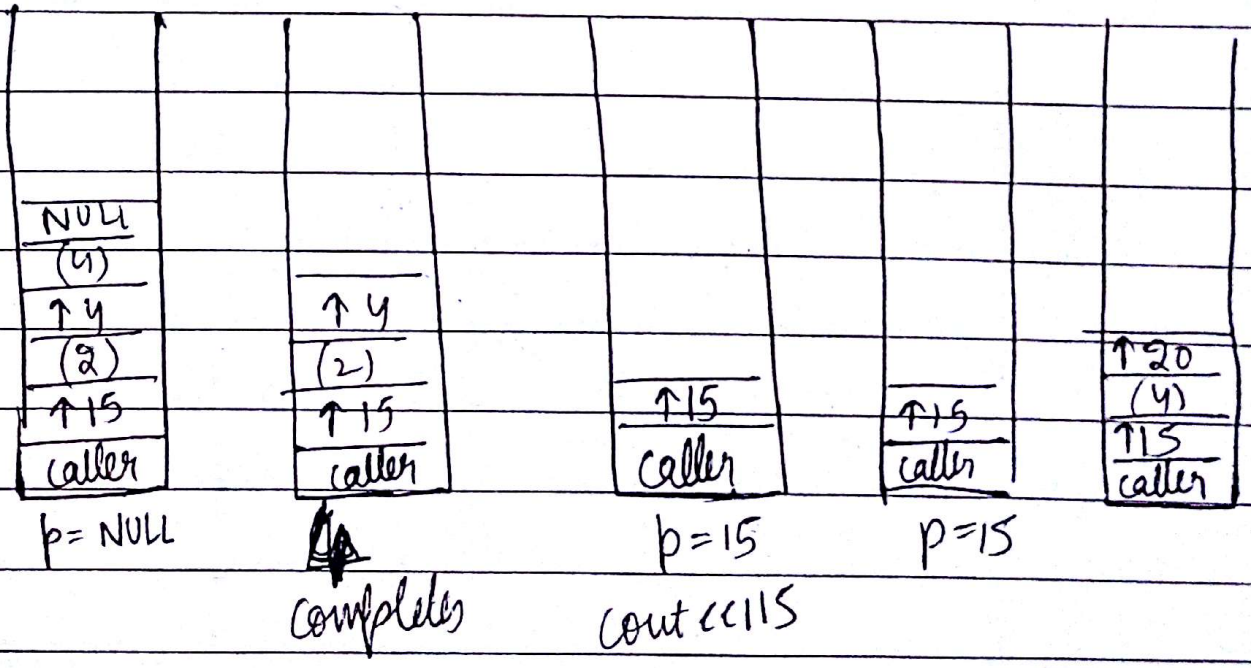
Notation: ↑4 (p->left)
(address of 4)
pointer to 4

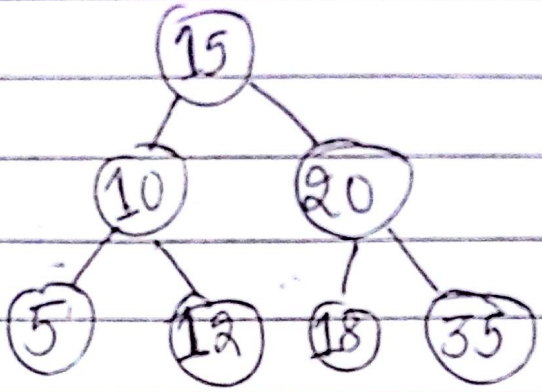
(2) = Address 2
(return address)



Date







Pre order : (15)(10)(5)(12)(20)(18)(35)

In Order : (5)(10)(12)(15)(18)(20)(35)

Post Order (5)(12)(10)(18)(35)(20)(15)

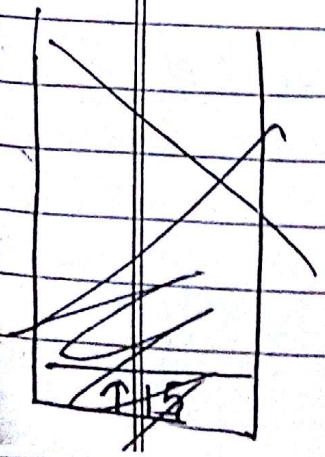
```
void preorder (node<T> * p)
{
    if(p != NULL)
    {
        cout << p -> data;
        preorder(p -> left);
        preorder(p -> right);
```

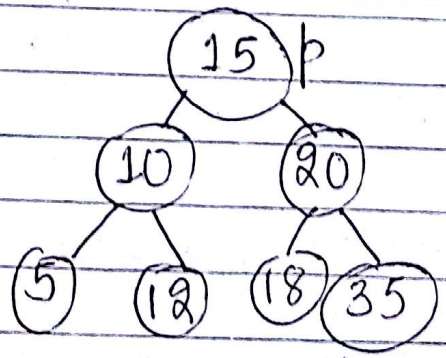
```
void postorder(node<T>* p)
{
  if(p != NULL)
  {
    postorder(p->left);
    postorder(p->right);
    cout << p->data;
  }
}

template <class T>
void iterativepreorder()   //we will use our
{                          //stack to push
Stack < node<T>* > s;      //node addresses.
  node<T> *p=root;
  if(p != NULL)
  {  s.push(p);
     while(!s.empty())
     {  p = s.pop();
        cout << p->data;
        if(p->right != NULL)
          s.push(p->right);
        if(p->left != NULL)
          s.push(p->left);
     }
  }
}
```

Stack using Arrays

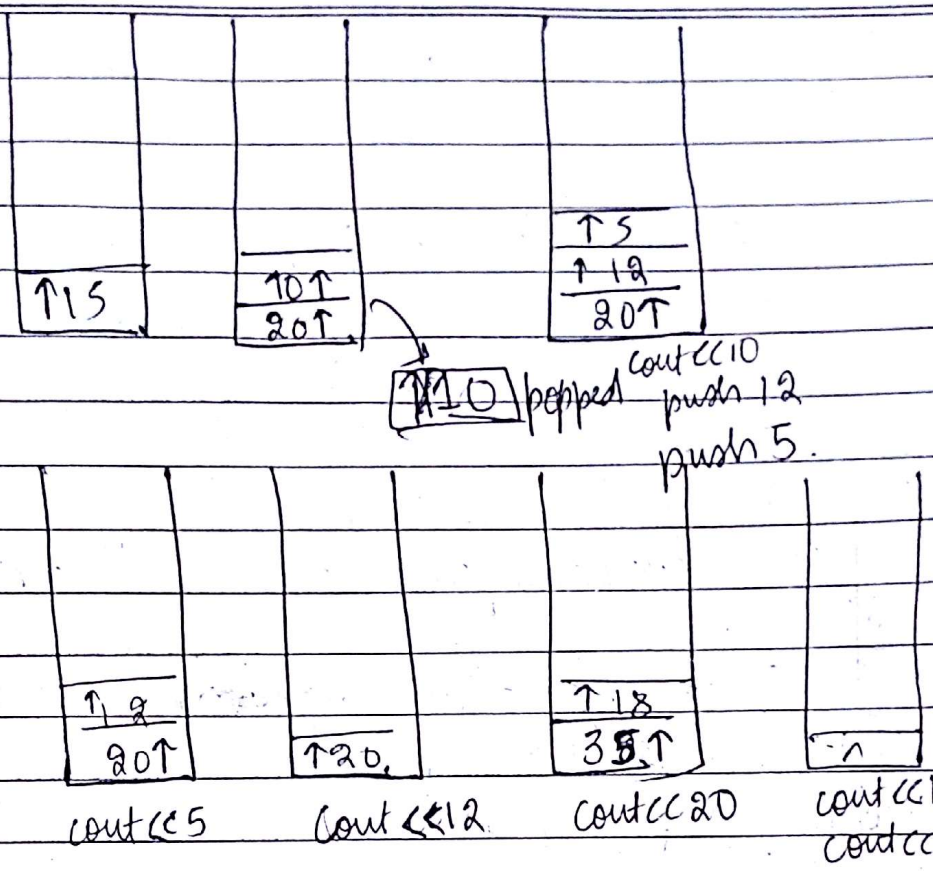
push right first then left so that left is popped first acc. to LIFO





15

p 15



5

ask Vipul sir

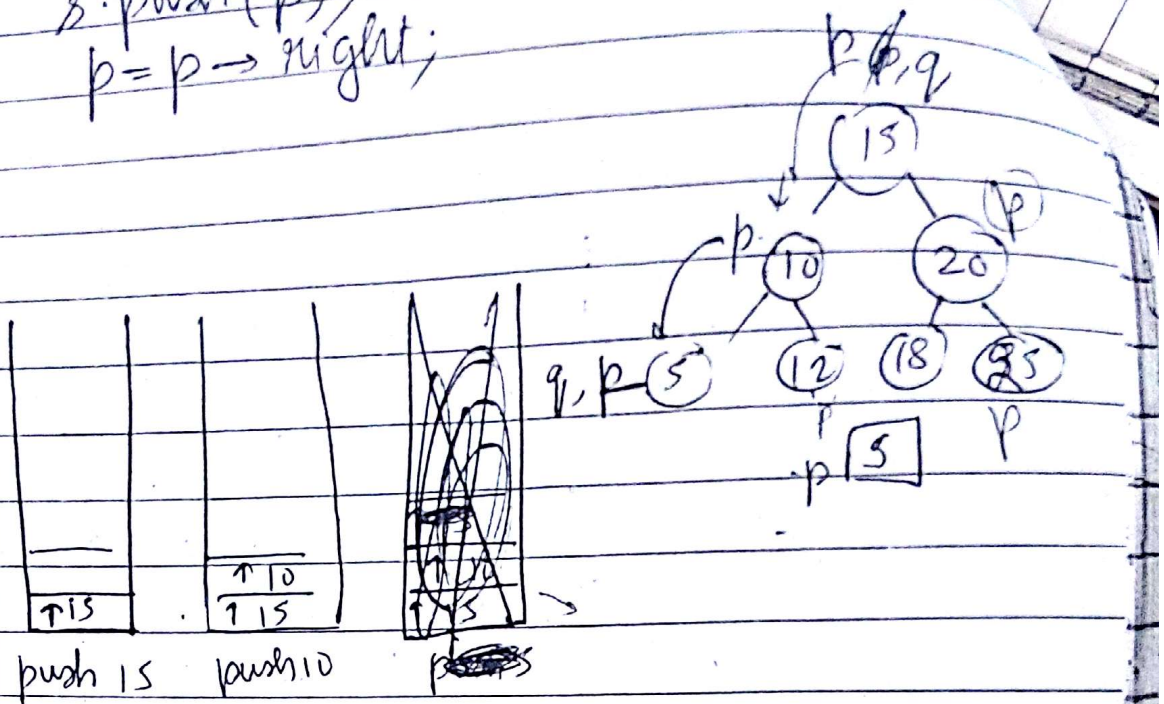
```
Void iterativePostOrder()
{   Stack<node<T>*> s;
    node<T> *p = root, *q = root;
    while(p != NULL)
    {
        for(; p->left != NULL; p = p->left)
            s.push(p);
        while(p != NULL && (p->right == NULL ||
                            p->right == q))
        {   cout << p->data;
            q = p;
            if(s.empty())
                return;
            p = s.pop();
        }
    }
```

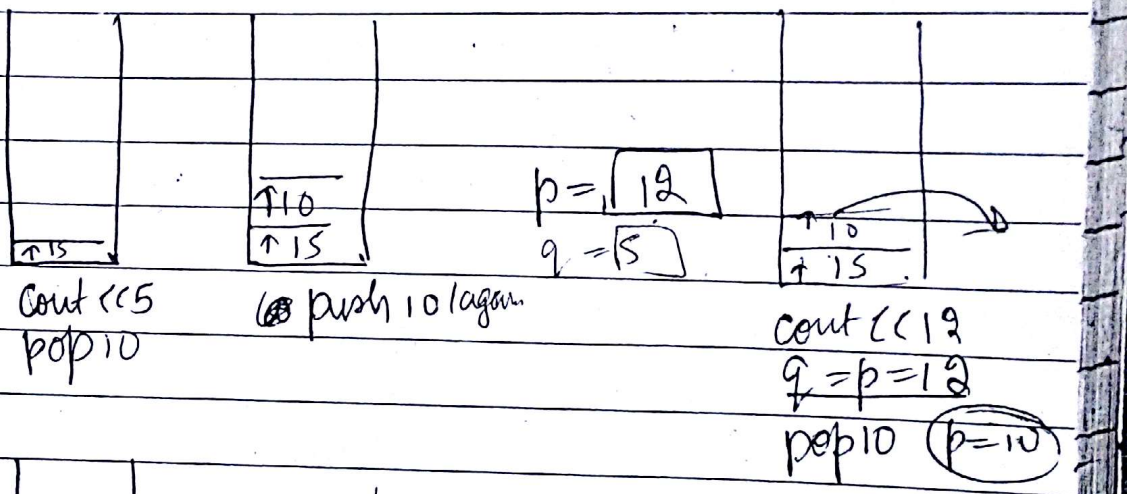
pushing left subtree except leaf elements (left subtree)

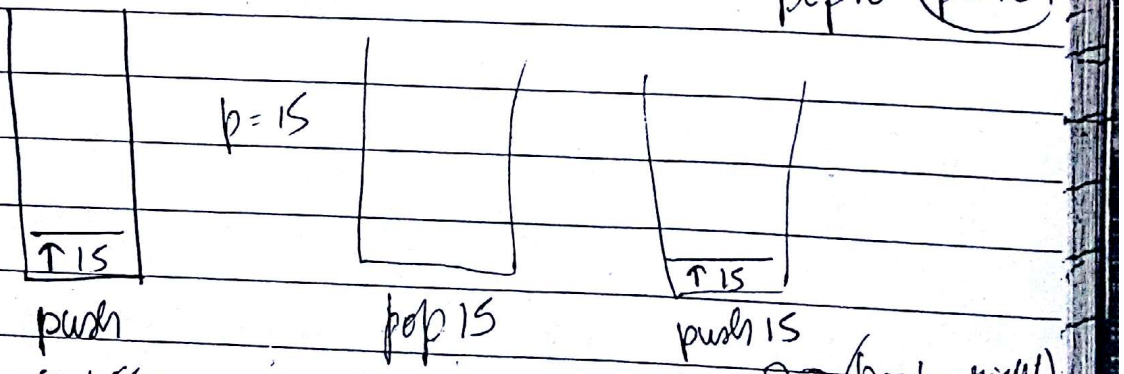
leaf node condition. right subtree handled or not (left subtree handled also first)

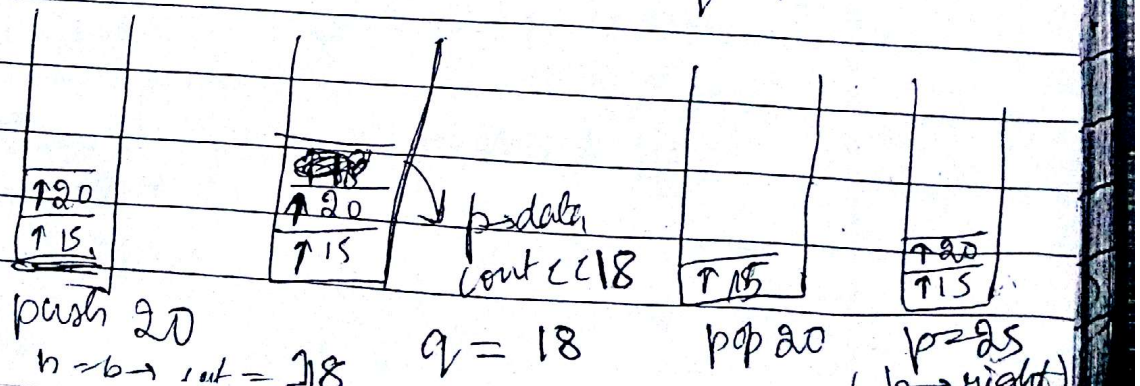
s.push(p);
p = p → right;

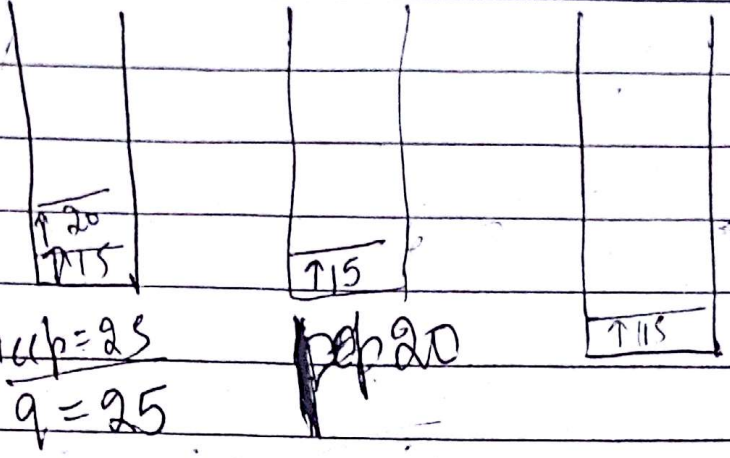
}
}











20

Curr cp = 25
q = 25

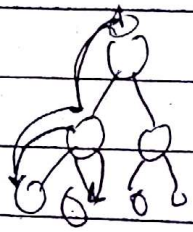
pop 20

Pg 231 (Book)

ask vipul sir

2 nested loops with exactly same condition

```
void iterative_inorder()
{
Stack <node<T> *> s;
 node<T> *p = root;
while (p != NULL)
{
if(p -> right != NULL)
      s.push (p -> right);
    s.push(p);
    p = p -> left;
}
p = s.pop();
while( !s.empty() && p->right == NULL)
{
    cout << p -> data;
     p = s.pop();
}
 cout << p -> data;
if( !s.empty())
     p = s.pop();
else
     p = NULL;
}
```



BST

23/9/16

Q Min → Ith minimum of tree, using recursion, iteration

Ans Maximum (p → right)

```
void Minimum()
{
    node<T> * min = root;
    while (min → left != NULL)
        min = min → left;
    cout << "Minimum" << min → data;
}
```

Iteration

```
void recMinimum (node<T> * p)
{
    if (p → left == NULL)
    {
        cout << p → data;
        return;
    }
    else
        recMinimum (p → left);
}
```

Recursion

For finding maximum element ESAP just traverse (p → right)

Q Count the no. of leaf node in a tree (recursive)

```
int count_leaf (node<T> *p)
{
    if (p == NULL)
        return 0;          // empty tree
    else if (p → left == NULL && p → right == NULL)
        return 1.          // root
    else return (count_leaf (p → left) + count_leaf (p → right));
}
```

Page
Date

Q Count the number of nodes recursively

```
int count (node<T> *p)
{
  if (p == NULL)
    return 0;
  else
    return ( 1 + count(p->left) + count(p->right));
}
```

Q. Calculate height of the tree (level)

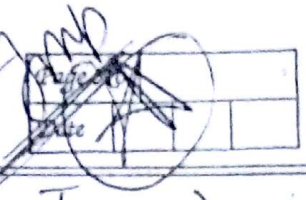
```
int height (node<T> *p)
{
  if (p == NULL)
    return 0;
  else
    return( 1 + max( height(p->left), height(p->right)) );
}
```

check

max(node<T>*) already a function to find max

```
int max (int a, int b)
{ return ( (a>b)? a : b);
```

Recursive insert

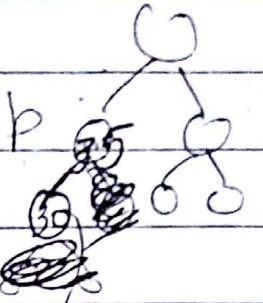


```
void insert (node<T> *p, T num)
{
if (p == NULL)          //empty tree
{
    node<T> * temp = new node<T>;
    temp->data = num;
root =temp;
}
else if( num< p->data)
    {
        if(p-> left == NULL)
        { node<T> *q= new node<T>;
          q->data = num;
          p->left = q;
        }
    else
        insert(p->left, num);
}
else if( num> p->data)
{ if (p-> right == NULL){
    node<T> *q = new node<T>;
      q->data = num;
      p->right =q;
    }
else
      insert (p->right, num);
}
else if (num == p->data)
        cout<<"Node already exist";
}
```

Q. IIIII

bool binarytree(node<T>* root1)
// write a function to check whether a given tree is BST or not.

```
bool is_BST( node<T> *p)
{
    if (p == NULL) //empty tree
        return true;
    else if ( p->left == NULL && p->right == NULL)
        return true;
    else if ( p->left != NULL && p->right == NULL)
    {
        if (p->data > (p->left)->data)
            return is_BST(p->left);
        else
            return false;
    }
    else if ( p->left == NULL && p->right != NULL)
    {
        if (p->data < (p->right)->data)
            return is_BST(p->right);
        else
            return false;
    }
    else
        return ( is_BST(p->left) && is_BST(p->right));
}
```



one node all case //

if data case

for checking further left BST

26/9/16

*& → Reference to a pointer

```
int p=10;
int *&q = p;
```

p, q → 10

& ⇒ same memory location को refer कर रहे हैं

```
int a = 10;
int *p = &a;
int *&q = p;
```

q → reference to an integer pointer

a → 10 (100)

q, p → 100

```
void pass_by_value (int *p)
{
    p = new int;
}
```

'*p' is local to pass_by_value() hence no changes are reflected back to main()

```
void pass_by_reference (int *&p)
{
    p = new int;
}
```

↳ changes reflected back to main

```
int main()
{
    int *p1 = NULL;
    int *p2 = NULL;
    pass_by_value (p1);      // p1 will still be NULL after this call
    pass_by_reference (p2);  // p2's value is changed to point to newly allocated memory.
    return 0;
}
```

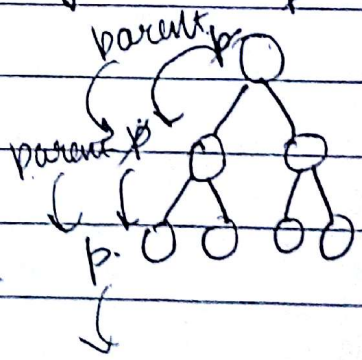
```
void fun ( node * &p)
{ p = new node;
  p -> data = 1;
}
// calling function
   node * q;
   fun(q);
```

changes reflected back to calling function.

≡ Recursive Insert ≡

Insertion by maintaining Parent node (root ka Parent = NULL pass karna)

```
void insert (node<T> * p, node<T> *parent, T num)
{ if(p == NULL)
  {
    node<T> * temp = new node<T>;
    temp -> data = num;   // root created only 1 node
    if(parent == NULL)
      root = temp;
    else
    { if (num < parent -> data)
        parent -> left = temp;
      else
        parent -> right = temp;
    }
  }
else if ( num < p -> data)
    insert ( p -> left, p, num);
else if ( num > p -> data)
    insert ( p -> right, p, num);
else if ( num == p -> data)
    { cout << "\n No. already exist"; return; }.
```



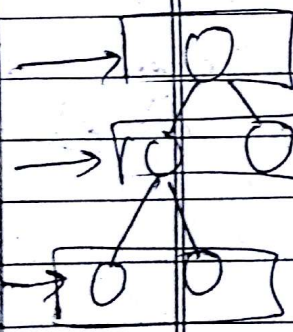
WAF

Q. Count no. of non-leaf nodes (internal nodes)

```
int count_nonleaf (node<T> * p)
{
if(p == NULL || (p->left == NULL && p->right == NULL))
        return 0;
else
return (1 + count_nonleaf(p->left) +
          count_nonleaf(p->right));
}
```

Counting no. of leaf node

⇒ Do level-by-level traversal & check for
(p->left == NULL && p->right == NULL)
↳ if true → increment the count of nodes



level-by-level - Breadth first search (BFS) Traversal.

INORDER, PREORDER, POSTORDER → DFS (Depth First Search)

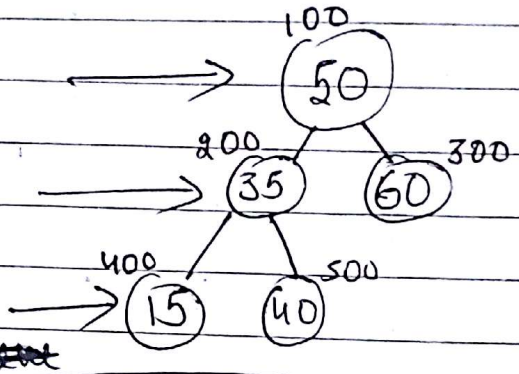
```
template <class T>
void BST<T>:: breadthFirst()
{ Queue< node<T> * > q;
    if(p != NULL)
    { q.enqueue(p);
```

(Level-by-Level Tree Traversal use of Queues)

empty() → function to check if queue is empty

```
while (!q.empty())
{   p = q.dequeue();
    cout << p->data;
    if (p->left != NULL)
        q.enqueue(p->left);
    if (p->right != NULL)
        q.enqueue(p->right);
} //end of while
} //end of if
} // end of function.
```

(level-by-level traversal)



q: | 100 | 200 | 300 | 400 | 500 |

p = 100, 200, 300, 400, 500

Output

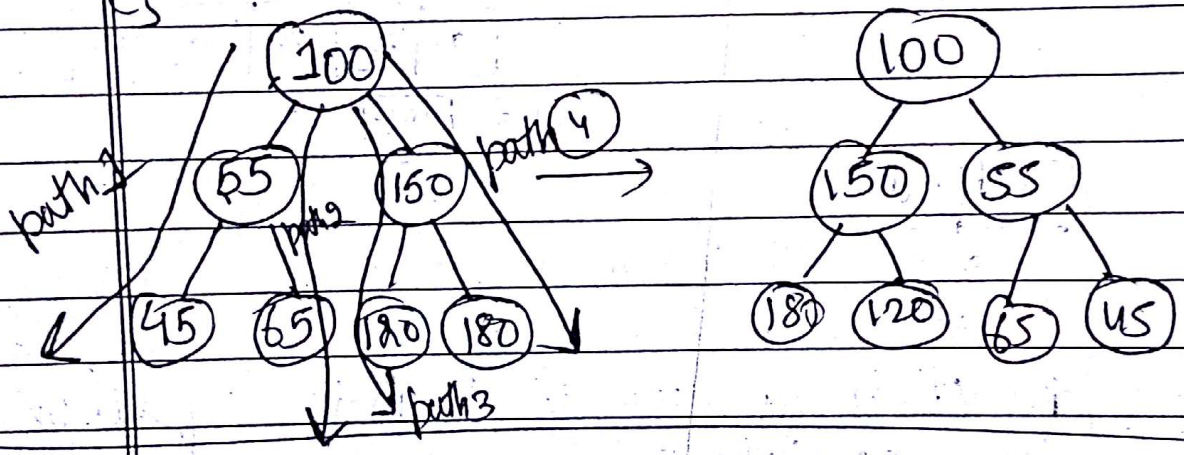
↳ 50 35 60 15 40

Q. function to check if 2 BST equal or not.

```
bool identicalBST(node<T>* p, node<T>* q)
{   if (p == NULL && q == NULL)
        return true;
    if (p != NULL && q != NULL)
    {
    return ((p->data == q->data) &&
        identicalBST(p->left, q->left) && identicalBST(p->right, q->right));
    }
    return false;
}
```

WAF to find mirror image of BST

```
void mirror (node<T> *p)
{   if(p == NULL)
        return;
    else{
        mirror (p->left);
        mirror (p->right);
        node<T> * temp;
          temp = p->left;
          p->left = p->right;
          p->right = temp;
}
}
```



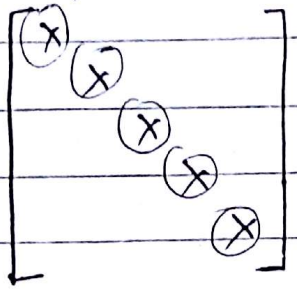
Q. Given sum of paths, find whether a path with same sum exist in BST or not?

```
bool path_sum (node<T>* p, T sum)
{
  if(p == NULL && sum == 0)
        return true;
  else if(p == NULL && sum != 0)
        return false;
```

```
else {
        sum = sum - (p->data);
return ( path_sum (p->left, sum) ||
         path_sum (p->right, sum));
}
}
```

27/9/13. Square Matrices

a



a_{11}	a_{22}	a_{33}	a_{44}	a_{55}
0	1	2	3	4

n=5

5 x 4 → for 1-D Array implementation of sparse matrix

25x4 → for 2-D Array representation of sparse matrix

Sparse matrix.

- Diagonal
- Upper triangle
- Lower triangle
- tri-diagonal.

get() function :→ returns the value at (i,j) location.

set() function :→ non zero value at a(i,j) location at i=j; then 0 at all i≠j.

ADT

overload operators

+, -, *, /, ()

```
template <class T>
class DiagonalMatrix {                    → private : int n;
public : DiagonalMatrix (int size = 10)     T *d;  → 1-D array
         { n = size; d = new T[n];
~DiagonalMatrix () { delete d[]; }
T get(int, int);
void set(int, int, const T&);
};
```

for diagonal element

for diagonal element

not altering the value getting from matrix

```
template <class T>
T diagonalMatrix<T>::get(int i, int j) const
{
  if(i<1 || j<1 || i>n || j>n)
  { cout << "\n Index Out of Bounds"; return;}
  if(i==j)
     return element[i-1];
  else
     return 0;
}
```

1-D array for storing 2-D array elements

a11 ↔ element[0]
a22 ↔ element[1]
aii ↔ element[i-1]

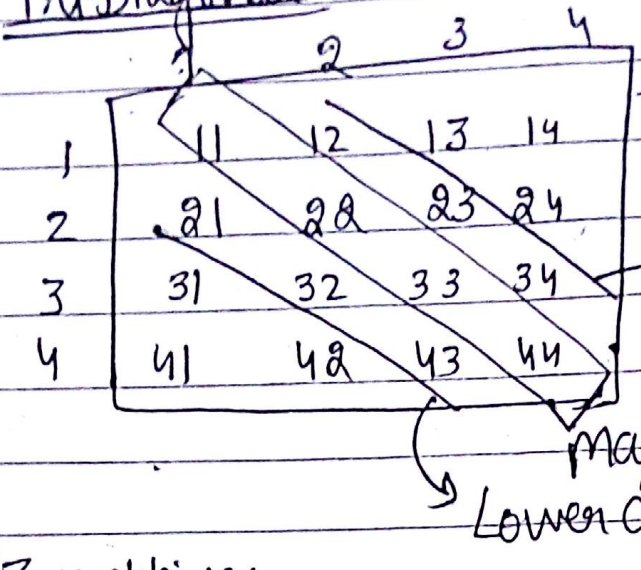
```
template <class T>
void diagonalMatrix<T>::set(int i, int j,
                            const T& newValue)
{
  if(i<1 || j<1 || i>n || j>n)
  { cout << "\n Index Out of Bounds";
    return;
  }
  if(i==j)
    element[i-1] = newValue;
  else
  {
    if(newValue != 0)
      cout << "\n Non-diagonal elements
              must be zero";
  }
}
```

a

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 2 & 0 & 0 \\ 0 & 0 & 3 & 0 \\ 0 & 0 & 0 & 4 \end{bmatrix}_{4\times4}$$

1	2	3	4
0	1	2	3

Tri-Diagonal



3 mappings
- diagonal (→ in book only this)
- Row major
- Column major

min size of 1D Array to represent tri diagonal

$3*n-2$

$n+(n-1)+(n-1)$

Diagonal Mapping

element [0:9]

Lower diagonal – Main – Diagonal – Upper Diagonal.

21 32 43 11 22 33 44 12 23 34

index → 0 1 2 3 4 5 6 7 8 9

i → array
j → mapping

21	32	43	11	22	33	44	12	23	34
0	1	2	3	4	5	6	7	8	9

$(i-2)$

$(n-1)+(i-1) = n+i-2$

$(n-1)+n+(i-1) = 2n+i-2$

* Lower diagonal → $i = j+1$
  Upper diagonal → $i = j-1$
  main diagonal → $i = j$

$(n-1)$ positions already occupied.

```
T get (int i, int j) const
{ if(i<1 || j<1 || i>n || j>n)
  { cout<<"\n Out of Bounds"; return; }
  switch (i-j)
  {
    case 1: // lower diagonal
            return element[i-2];
    case 0: // main diagonal
            return element[n+i-2];
    case -1: // upper diagonal
            return element[2*n+i-2];
    default: return 0;
  }
}
void
  set (int i, int j, const T & newValue)
{
  if (i<1 || j<1 || i>n || j>n)
  { cout<<"\n Out of Bounds";
  } return;
  switch (i-j)
{ case 1: // lower diagonal
          element[i-2] = newValue; break;
  case 0: // main diagonal
          element[n+i-2] = newValue; break;
  case -1: // upper diagonal
          element[2*n+i-2] = newValue; break;
  default: if(newValue != 0)
           { cout<<"\n Element can't be nonzero";
           }
```

Lower Triangular

	1	2	3	4
1	11	12	13	14
2	21	22	23	24
3	31	32	33	34
4	41	42	43	44

(i - 1)

$(i >= j)$

non-zero.

Row Major: 1-D Array size $\rightarrow \frac{n(n+1)}{2}$ (for Lower triangular)

element [0:9]

index →	0	1	2	3	4	5	6	7	8	9
	a_{11}	a_{21}	a_{22}	a_{31}	a_{32}	a_{33}	a_{41}	a_{42}	a_{43}	a_{44}

For Row major, if we are looking in i^{th} row upto then $(i-1)$ elements have ~~been~~ occupied space $\rightarrow 1 + 2 + \dots + (i-1)$

$$\Rightarrow \frac{(i-1)(i-1+1)}{2}$$

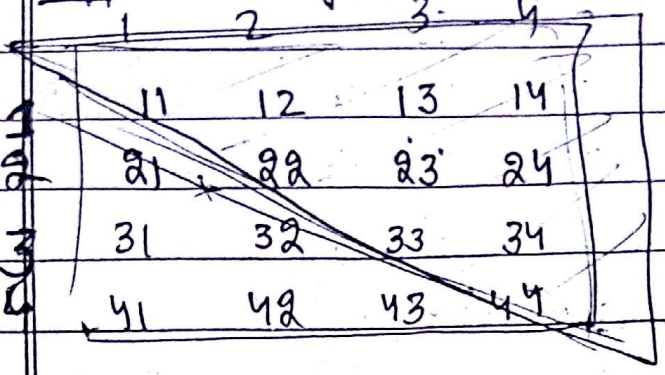
$$= \frac{i(i-1)}{2}$$

Put $\frac{n(n+1)}{2} = i - 1$

n is array location we store 1 behind.

$$[(i*(i-1))/2 + (j-1)] = \text{new Value}$$

Upper Triangular



Size of 1-D Array reqd $= \frac{n(n+1)}{2}$

element [0-9]

Row-major

row index = 1 col. index = j.

$j-1$

row index = 2 $N+(j-2)$ $N+(N-1)+(j-3)$

11	12	13	14	22	23	24	33	34	44
0	1	2	3	4	5	6	7	8	9

$N+N-1$

row 1 $0+j-1$

2 $N+(j-2)$

3 $N+(N-1)+(j-3)$

i

$$N+(N-1)+(N-2)+\dots+(N-(i-2))+(j-i)$$

$$N*(i-1)-[1+2+3+\dots+(i-2)]+j-i$$

$$N*(i-1)-\left[\frac{(i-2)(i-1)}{2}+(j-i)\right]$$

2-D Array
Row major mapping.

$u_2 \rightarrow$ no. of columns in the array.

$$map(i_1, i_2) = i_1 * u_2 + i_2$$

where row goes from $0, \ldots, i_1 - 1$

$$map(i_1, i_2) = 3i_1 + i_2$$

$map(0, 0) = 0$
$map(0, 1) = 1$

4 x 3

00	01	02
10	11	12
20	21	22
30	31	32

00	01	02	10	11	12	20	21	22	30	31	32
0	1	2	3	4	5	6	7	8	9	10	11

3-D Array

a[3][2][4].

[0] [ ] 2 x 4

[1] [ ] 2 x 4

[2] [ ] 2 x 4

Page No.
Date

1st dimension → Break

→ a[3][2][4]

[0][0][0] [0][0][1] [0][0][2] [0][0][3] [0][1][0] [0][1][1] [0][1][2] [0][1][3]

[1][0][0] [1][0][1] [1][0][2] [1][0][3] [1][1][0] [1][1][1] [1][1][2] [1][1][3]

[2][0][0] [2][0][1] [2][0][2] [2][0][3] [2][1][0] [2][1][1] [2][1][2] [2][1][3].

$map(i_1, i_2, i_3) = i_1 u_2 u_3 + i_2 u_3 + i_3$

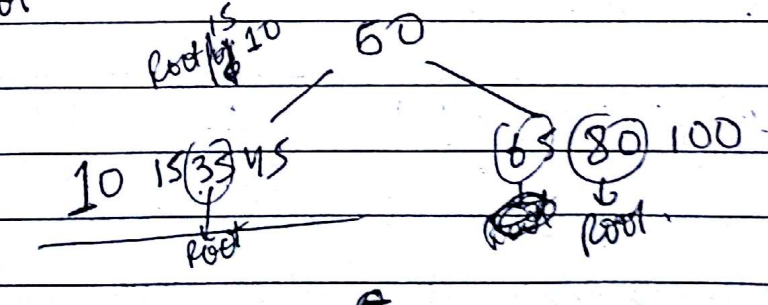
$i_1 u_2 u_3$ → for 8 elements in each row.

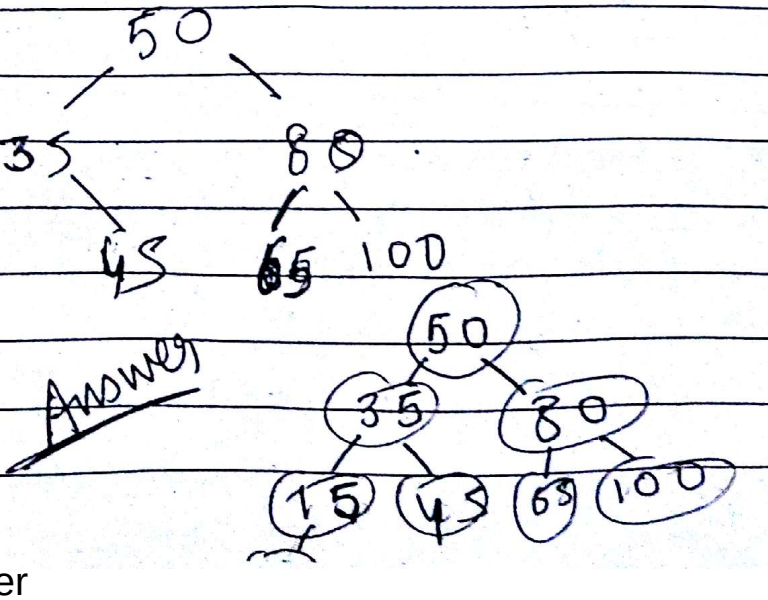
$i_2 u_3$ → for 2 →

Make a tree out of the given info

Inorder: 10 (left) 15 (35) 45 (50) 65 80 100
(Left Root Right)
R2 R1 Right subtree of R1

PreOrder: (50) 35 (15) 10 45 80 65 100
Root left right
root ↓ left subtree Right subtree





Left Right Root

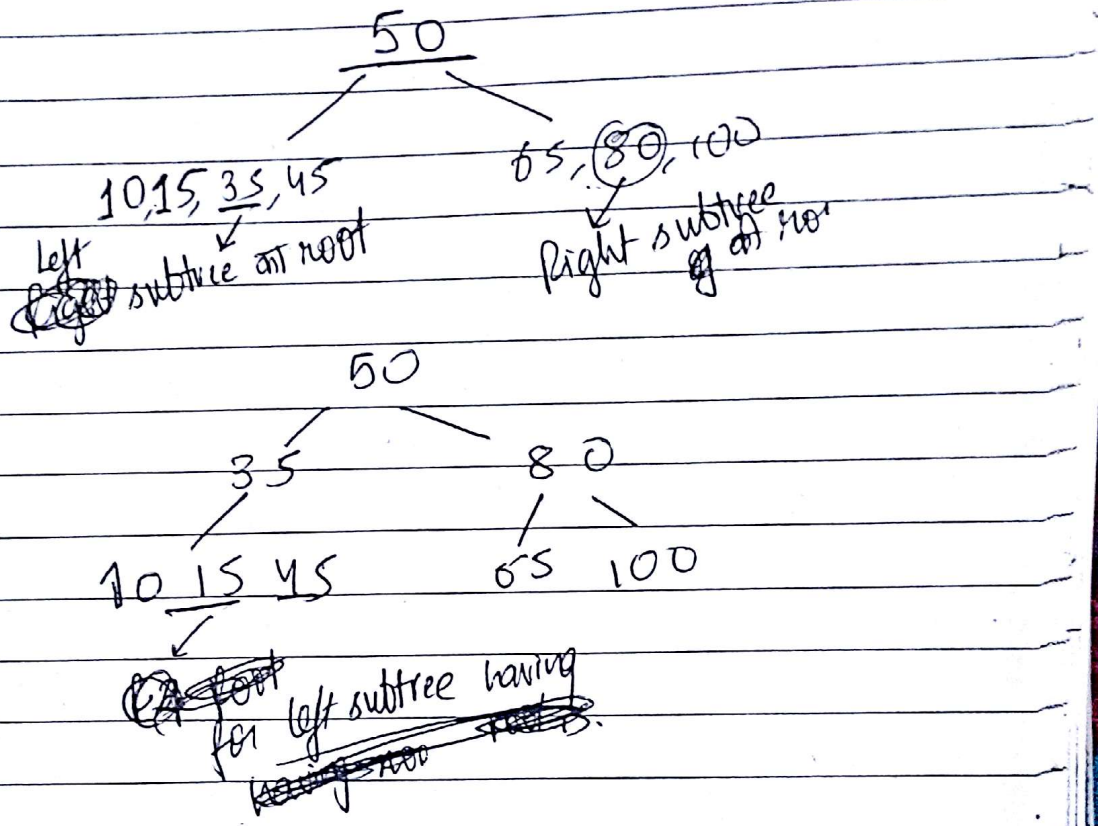
Page No.
Date

Last one is the root.

PostOrder: (10) (15) (45) (35) (65) (100) (80) (50)

InOrder: 10 15 35 45 50 65 80 100

Left subtree of (50) — Rt subtree of 50



root of tree

LevelOrder/BFS: 50 35 80 15 45 65 100 10

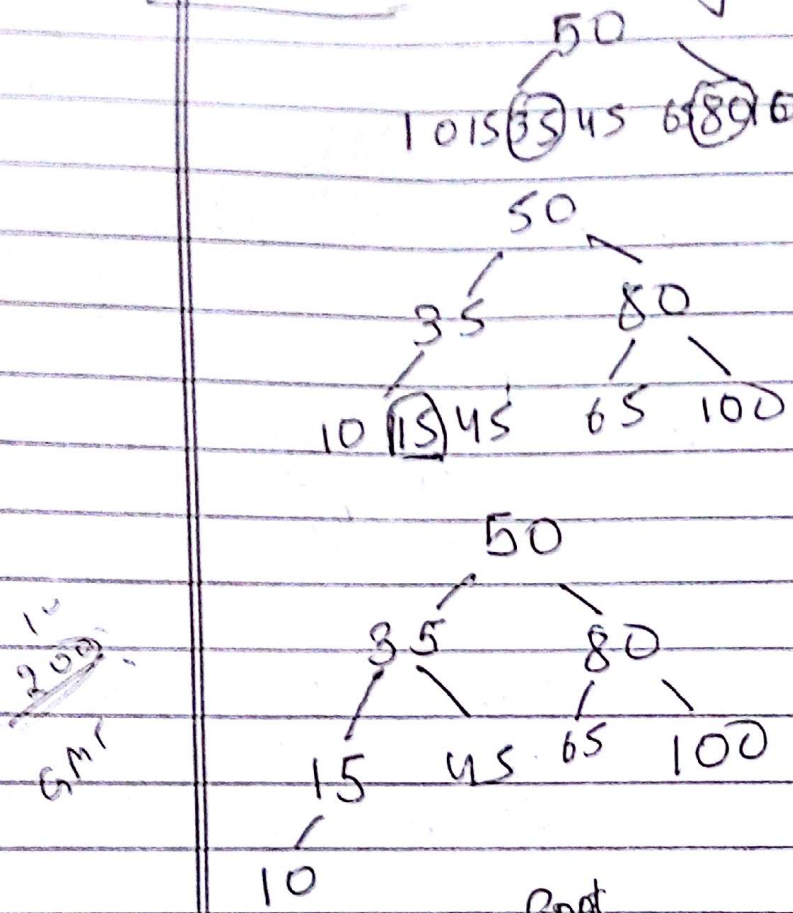
InOrder: 10 (15) (35) 45 (50) 65 (80) 100

L R1 R2 R L R3 R

Left subtree — Right subtree

Date

⇒ Level Order ⇒ Directly root comes first



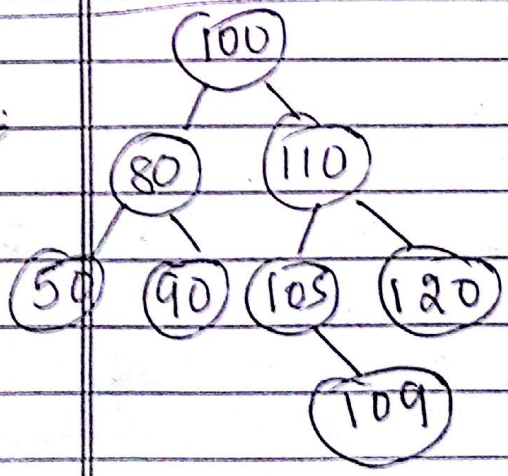
Root

Pre Order : 50 35 15 10 45 80 65 100

Post Order : 10 15 45 35 65 100 80 50

Root.

Q.



Root

Pre Order : 100 80 50 90 110 105 109 120

Post Order : 50 90 80 109 105 120 110 100

(50) (90) (80) (109) (105) (120) (110) (100)

Pre Order