Probing → finding a slot for key in hash table

Date
Page No.

gate
Gate-1997 → Good concept to Generalise and formulas to be cosite in short notes.

Open Addressing (Close Hashing)

Elements are not allowed to insert outside the table so

load factor $= \frac{n}{m} = \alpha$

But $n \leq m$ so

$0 \leq \alpha \leq 1$

7:31

If 'm' slot are present in hash table then in worst case time complexity is O(m).

$h(K_1,0) = 0$ → Probing sequence. If 4 slots are present in hash table and add are full then we always get a miss and a probing sequence obtained.
$h(K_1,1) = 2$
$h(K_1,2) = 3$
$h(K_1,3) = 1$

Ex

$h(K_1,0) = 2$	0	N
$h(K_1,1) = 6$	1	N
$h(K_1,2) = 3$	2	a
$h(K_1,3) = 4$	3	c
$h(K_1,4) = 5$	4	d
$h(K_1,5) = 1$	5	K_1
$h(K_1,6) = 7$	6	b
$h(K_1,7) = 0$	7	N

If we delete C and mark it as Null then if we search for K_1 then $h(K_1,2) = 3$ will hit a Null and Ns suppose that since there is a Null so K_1 can't follow the further and it must be present there but since there is NULL so K_1 is absent but K_1 is present as (5). So we mark other than Null by any symbol. Such that it shows that the memory is deleted and you have continue to next probe sequence.

Linear Probing

$h : U \longrightarrow \{0, \ldots m-1\}$

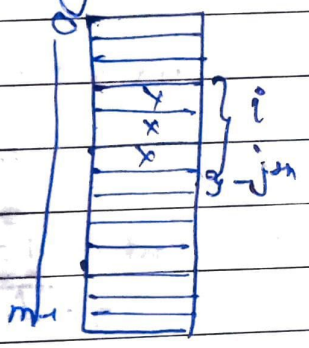
Take a key from U and mapped it with {0 – m-1}

No. of probe sequences possible → m

Suffers

↳ Secondary Clustering → If two elements starts to probe with the same value of index in hash table then both will follow the same probe sequence.

↳ Primary Clustering



If i slots before jth slot is filled then probability of occupy jth slot is

$$\left(\frac{i+1}{m}\right) = \frac{i}{m} + \frac{1}{m}$$

$\left(\frac{1}{m}\right)$ because in uniform probing each slot has equal probability. It may possible that the h(function) hits the slot directly.

$\left(\frac{i}{m}\right)$ because jth slot may be filled by the hash function if they ~~formed~~ found filled jth slot.

अगर $h(k) = i$ filled hai तो $h(k,1)$ पर shift होता है और Linear probing से chance है कि search करते हुए Empty slot jth मिल जाए तो उसे fill कर देगा।।

Date
Page No.

Seach time increases due to primary clustering.
But on an avg it is O(c).

It has two hash functions—
① for hashing
② for probing.

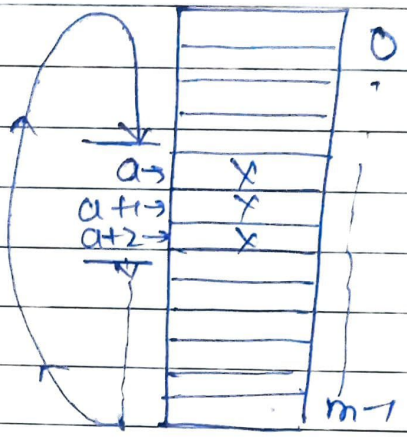
Ex $h(k) = a$

$h'(k,i) = (h(k)+i) \bmod m$

where $i \rightarrow$ no. of collision
$m \rightarrow$ size of table

$h'(k,1) = (a+1) \bmod m = a+1$

$h'(k,2) = (a+2) \bmod m = a+2$



There are 'm' possibilities of m sequence

Gate 2008 If they didn't mention anything then start the indexing from 0.

$h'(k,i) = (h(k)+i) \bmod 10$

Probability of collision $= \frac{n}{m}$

$n \rightarrow$ no. of slots filled
$m \rightarrow$ total no. of slots.

Gate-2010 → Good concept to remember.
(Start video from 9:20)

Quadratic probing

$h'(k,i) = (h(k) + c_1 i + c_2 i^2) \bmod m$

It reduces to negligible the consequences of primary clustering.

Date :
Page No.

If we have m slots in hash table then on probing from $h'(k_1, 0)$ — $h'(k_1, m-1)$ we have to visit all the different indices (0→m-1) only once.
If any index or slot repeated, means probing is inappropriate.

But in quadratic probing it is very difficult to choose the value of c_1, c_2 according to situation.

It may possible we may not visit all the (m-1) probes. So linear probing is better.

Let us say, $c_1 = c_2 > 1$ and $key = k_1$
then,
probe sequence will be (1, 3, 6, 1, 8, 7, 8 —)
and if
$key = k_2$ also have
probe sequence (1, 3, 6, 1, 8, 7, 8 —)
called secondary clustering.

Each probe sequence depends on initial value of probe sequence, any there may be 'm' different values at initial position. So no. of different probe sequence possible is → m.

first position is depend on key and all the positions are independent of key in probe sequence.
It is better than linear probing but it can't be run efficiently.

Date :
Page No.

Double Hashing

$h'(k,i) = (h_1(k) + i h_2(k)) \bmod m$

→ NO primary Clustering and secondry clustering.
→ Next position is decided by $i h_2(k)$.
for same starting probe position there may be different probe sequence.

→ No. of probe sequence → m^2

Ex- If $(h_1(k_1), h_2(k_1))$
$(h_1(k_2), h_2(k_2))$
and $h_1(k_1) = h_1(k_2)$ —— m sequence
then, $h_2(k_2) \neq h_2(k_1)$ So, —— m sequence

Total m^2 sequence

It is best probing technique till now even to increase efficiency you can modify to triple hashing and so, on.

Choose the last hashfunction ($h_2()$ here) and m carefully. So that you might not end up without visiting any index

i) Make sure that $h_2(k)$ and m are relatively prime.
To make that

$h_2(k) \rightarrow$ odd	or	$h_2(k) < m$	
$m \rightarrow 2^k$		and	$m \rightarrow$ prime no.

Avg time complexity = $O(k)$; $k \cong 1$
If table is full $k \approx 2.5$
—— is half $k \approx 1.5$

Trees (Module 7)

Date :
Page No.

Introduction to Tree Traversal :-

Information inside every node

Binary Tree → Tree have atmost 2 children.

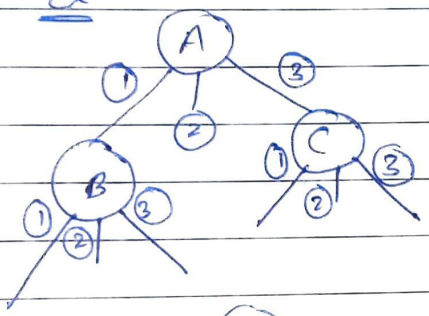
INORDER, PREORDER, POSTORDER — mainly proposed for binary tree but the extended version are applicable to trees with atmost 3, 4 .. so on nodes.

INORDER → L Root R → visit the node IInd time

PREORDER → Root L R → visit the node Ist time

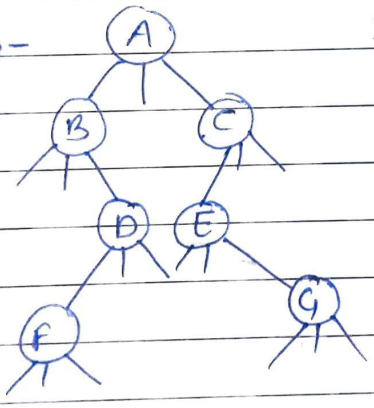
POSTORDER → L R Root → visit the node IIIrd time.

Ex



INORDER – BAC
PREORDER – ABC
POSTORDER – BCA

Ex-



PRE - ABDFCEG
IN - BFDAEGC
POST - FDBGECA

Implementation of Traversals, space and time complexity :-

```
Struct node{
    char data;
    struct node * left, * right;
};
```

```
void Inorder (struct node *t){
    if (t) {                    # (t != NULL)
  Inorder (t -> left);
   Printf ("%C", t -> data);
   Inorder (t -> right);
 }
}
```

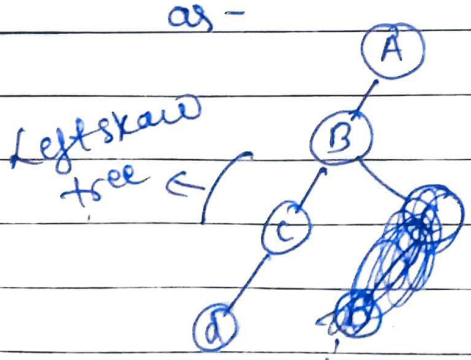
Program of recursion always solved using stack concept as learned earlier.

Implementation of tree should be done using Linked list.

Time Complexity → We are visiting 'n' nodes 3 times and every time we spent a constant amount of time, so-

$T(n) = 3 * n * C = \boxed{O(n)}$ | same for all orders

Space Complexity → Depends on level. In worst case it may possible that n nodes have level as -



Leftskew tree ←

→ Called as Skew-tree.

$= \boxed{O(n)}$ | same for all orders

for preorder :-

```
printf (t -> data);
Preorder (t -> left);
Preorder (t -> right);
```

for postorder

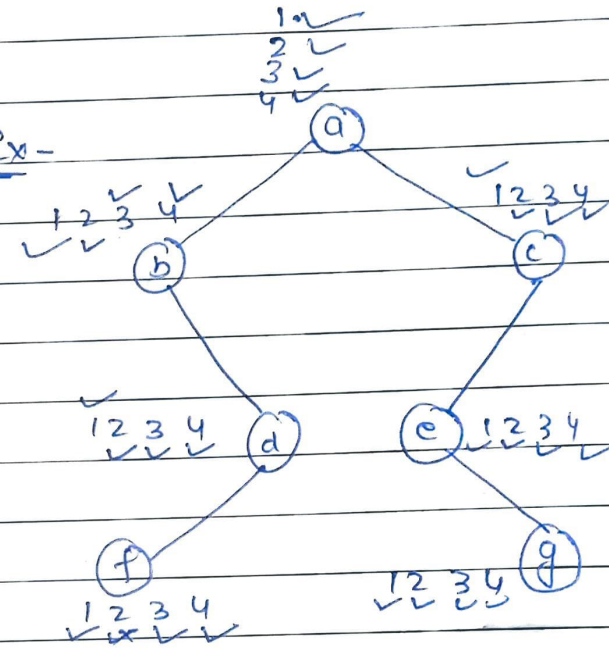
```
Postorder (t -> left);
Postorder (t -> right);
Printf (t -> data);
```

Date :
Page No.

Double Order Traversal # दो बार Printf use है।

```
Double Order (struct node *t) {
    if (t)
1.  pf (t → data);
2.  DO (t → left);
3.  pf (t → data);
4.  DO (t → right); }
```

Ex-

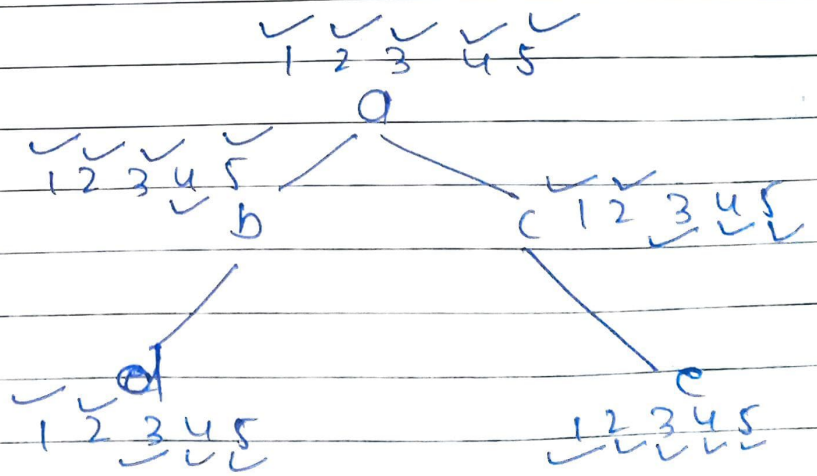


o/p | a b b d f f d a c e e g g c |

जितनी भी लाइन ऑफ कोड function में Given है। सबको हर Node पर Mark करो और जिस Number पर जो Line of Code है उसे execute करते हुए आगे बढ़ो।

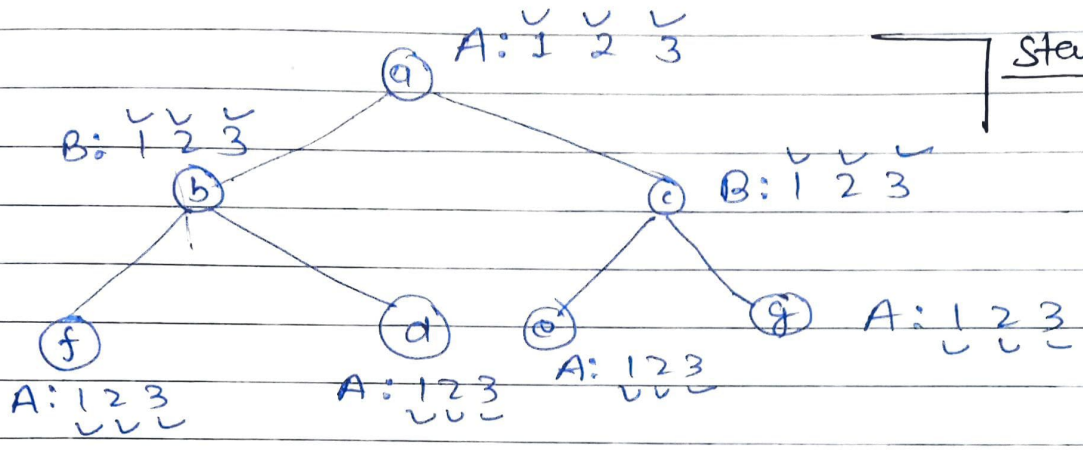
Triple Order Traversal # Printing value of each node 3 times.

```
    if (t) {
1.  pf (t → data)
2.  TO (t → LC)
3.  pf (t → data)
4.  TO (t → RC)
5.  pf (t → data)
    }
```



o/p | a b d d d b b a c c e e e c a |

Indirect recursion on Trees



Starting with A

o/p → afbdecg

Code

```
Void A (struct node *t){
  if (t)
  Pf (t->data);
  B(t->left);
  B(t->right); }}
```

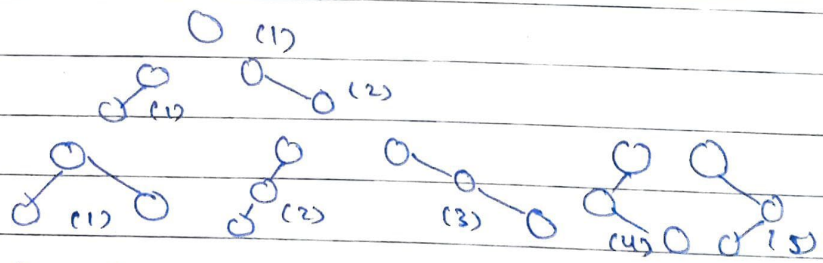
```
void B (struct node *t){
  if (t) {
  A(t->left);
  Pf(t->data);
  A(t->right); }}
```

Nodes are called in alternative levels

Number of binary trees possible!-

[Unlabelled trees]

With 1 node → 1
With 2 nodes → 2
With 3 nodes → 5



We are talking about numbes of structure b/coz no node is lebelled it may form more trees also for n nodes -

$$\frac{2n C_n}{(n+1)}$$

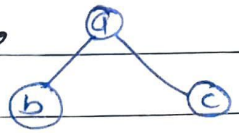
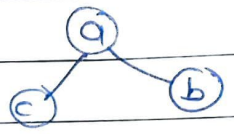
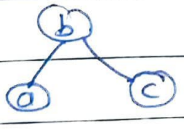
for $n = 3$ $6C_{3/4}$

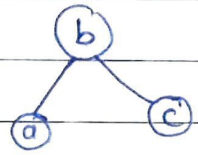
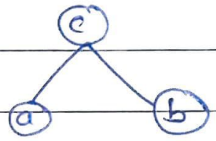
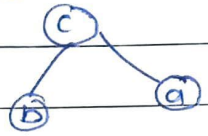
$$= \frac{6 \times 5 \times 4}{3 \times 2 \times 1 \times 4} = 5$$

Every structure of capable of generating labelled trees → $\boxed{n!}$

Date :
Page No.

for  if labelled then,

$= \underline{3!} = 6$

No. of binary tree possible with labelled n nodes

$$= \boxed{\frac{2nC_n}{(n+1)} * n!}$$

No. of binary tree possible labelled and unlabelled

All the binary trees with 3 nodes A, B and C, which have preorder "A B C".

Sol^n No. of binary trees possible with 3 nodes -

Leveled → $5 \times 6 = 30$

Unleveled → 5.

Since only a unique binary tree possible with given either preorder, postorder or Inorder.

No. of trees with partial pre-order, postorder or Inorder (only one) is → $\boxed{\frac{2nC_n}{(n+1)}}$

No. of tree satisfying any of the two conditions -

→ More than one tree possible

No of tree satisfying all the tree traversal orders is -

→ Only 1

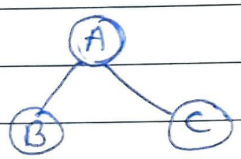
Date
Page No.

To generate a binary we need -
1. Inorder and Preorder
2. Inorder and Postorder.

] Inorder is necessary.

Construction of unique binary tree using In-order and pre-order.

Ex → PREORDER - ABC
INORDER - BAC

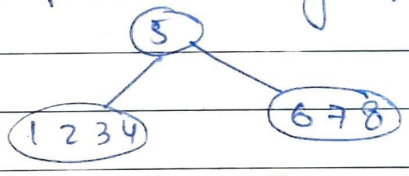


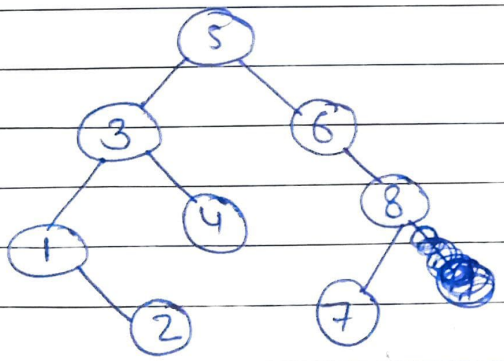
Ex-2 IN - 1 2 3 4 5 6 7 8
Pre - 5 3 1 2 4 6 8 7
Post - 1 2 1 4 3 7 8 6 5

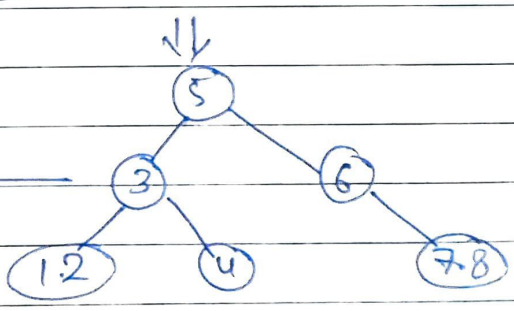
Case1 given in, pre order then,

] Start from beggining of pre-order

1 2 3 4 5 6 7 8 ⇒



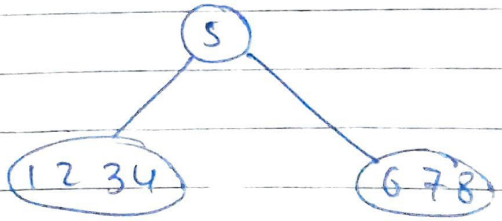




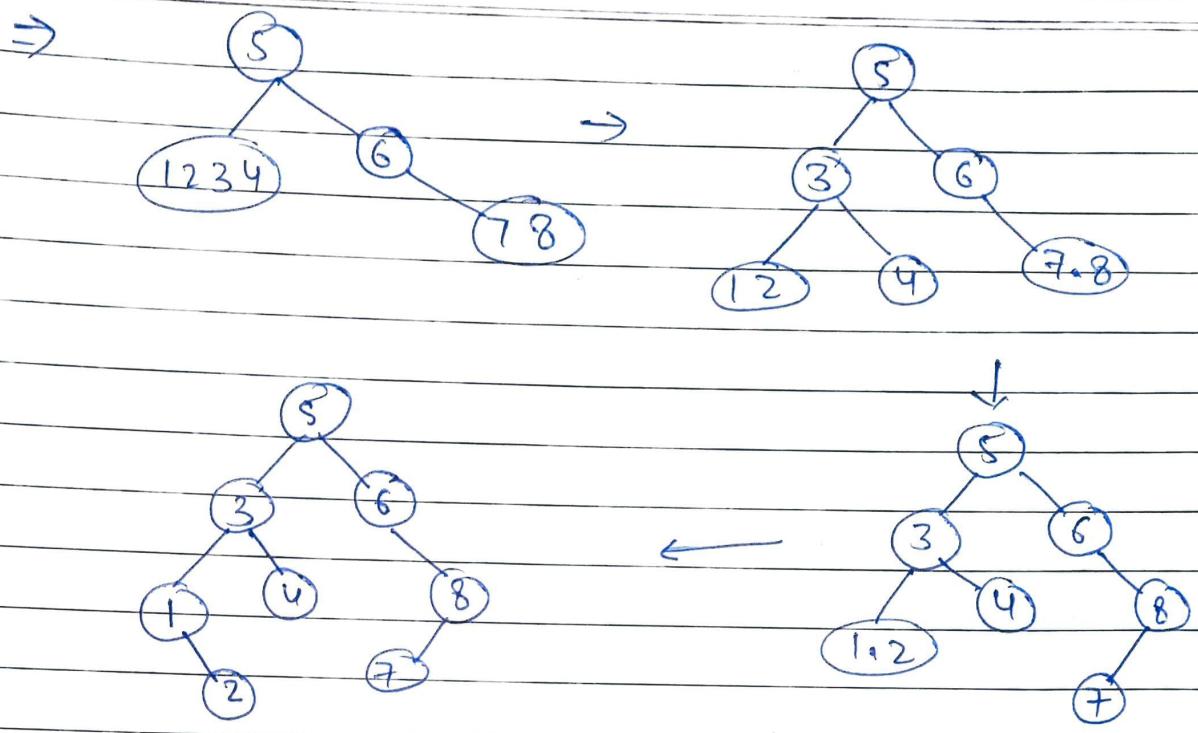
Case-2 Given in, post order

] Start from end of post order.

1 2 3 4 5 6 7 8 ⇒



Date :
Page No.

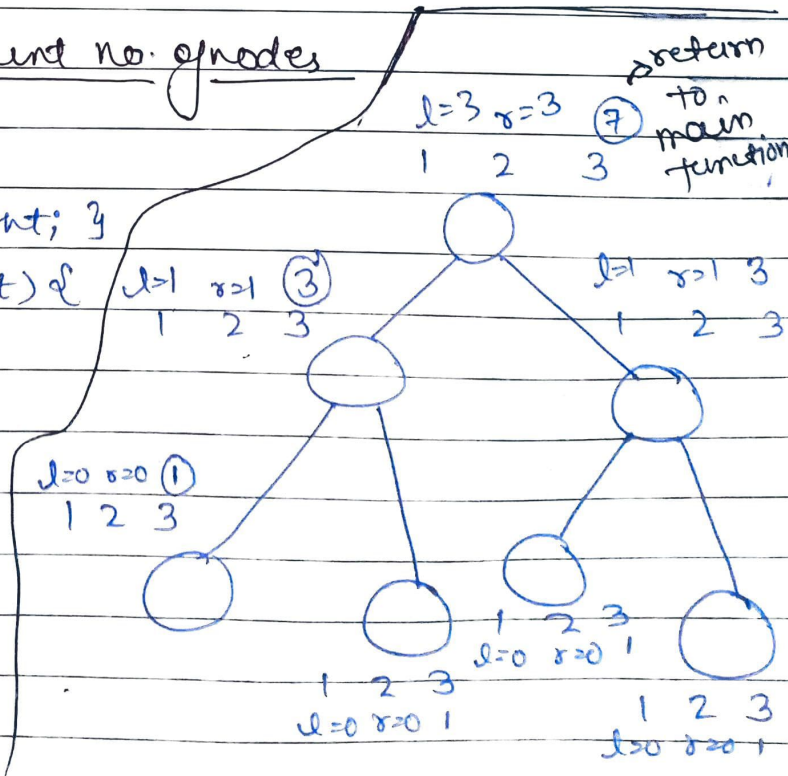


Only if one type order of traversal is given no unique binary tree can be constructed.
No of tree constructed = $\frac{2nCn}{(n+1)}$

No tree (uniques) can be formed by pre-order and post order.

Recursive program to count no. of nodes

```
Struct node {
    int i;
  Struct node * left, *right; }
int NN (struct node *t) {
    if (t) {
     int l, r;
1.  l = NN (t -> left);
2.  r = NN (t -> right);
3.  return (1+l+r);
    } else
      return 0;
}
```



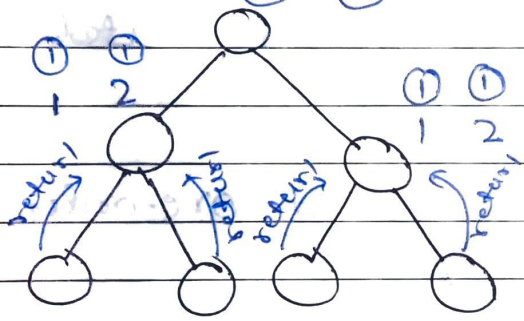
Date :
Page No.

Recursive program to count no. of leaves and non leaves :-

If T is leaf
NL(T) = 1;
otherwise
NL(T) = NL(LST) + NL(RST)

```
NL (struct node *t) { if(t == NULL) return 0;
  if (t -> left == NULL && t -> right == NULL)
      return 1;
  else
  return ( NL(t -> left) + NL(t -> right)); }
```

(2) ← (1) (2) – (2)



Main root t will return 4 to main function.

for non-leaf :-

```
if (t == NULL)
   return 0;
if (t -> left == NULL && t -> right == NULL)
   return 0;
else
return (1 + NL(t -> left) + NL(t -> right));
```

Same code as above with two changes only

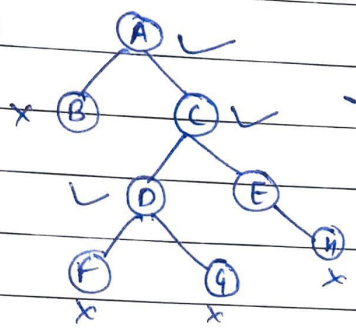
Time complexity in all the 3 Algorithm discuss previously = O(n) = space complexity.

Date :
Page No.

Recursive program to find full nodes

full node is a node having all children. In a binary tree node having 2 children called binary tree.

Ex →



→ 3 full nodes (V) (A, C, D)
→ 4 leaf (X) (B, F, G, H)
→ Non Leaf (A, C, D, E)

Full nodes always subset of Non Leaf.

FN ⊆ NL

FN(T) = 0 ; T = NULL || T is a leaf
= FN(T→LST) + FN(T→RST) ; if T has only 1 child
= FN(T→LST) + FN(T→RST) ; If T is full node

```
int FN (struct node *t){
 if (!t) return 0;
 if ( !t->left && !t->right) return 0;
  return ( FN(t->left) + FN(t->right) + (t->left && t->right)?
                    1:0); }
```

Recursive program to find the height of tree

$$H(T) = \begin{cases} 0 ; T \text{ is empty} \\ 0 ; T \text{ is leaf} \\ 1 + \max(H(LST), H(RST)) ; \text{otherwise} \end{cases}$$

Here we are assuming that height of a leaf is 0 but it may vary according to question.

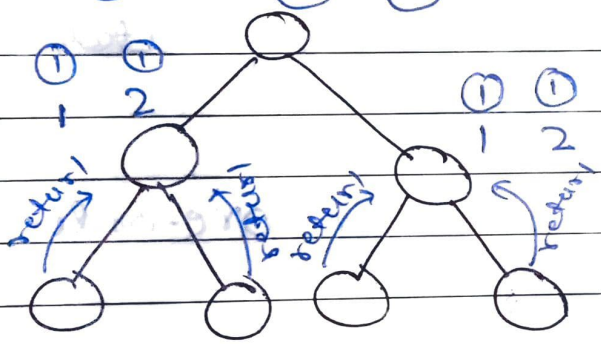
Date :
Page No.

Recursive program to count no. of leaves and non leaves :-

If T is leaf
NL(T) = 1;
otherwise
NL(T) = NL(LST) + NL(RST)

```
NL (struct node *t) { if(t == NULL) return 0;
  if (t -> left == NULL && t -> right == NULL)
      return 1;
  else
  return ( NL(t -> left) + NL(t -> right)); }
```

(1) (2)



main root t will return 4 to main function.

for non-leaf :-

```
if (t == NULL)
    return 0;
if (t -> left == NULL && t -> right = NULL)
    return 0;
else
return (1 + NL(t -> left) + NL(t -> right));
```

Same code as above with two changes only

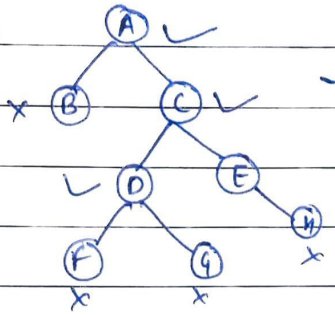
Time complexity in all the 3 Algorithm discuss previously = O(n) = space Complexity.

Date :
Page No :

Recursive program to find full nodes

full node is a node having all children. In a binary tree node having 2 children called binary tree.

Ex →



→ 3 full nodes (V) (A, C, D)
→ 4 leaf (X) (B, F, G, H)
→ Non Leaf (A, C, D, E)

Full nodes always subset of Non Leaf.

$FN \subseteq NL$

FN(T) = 0 ; T = NULL || T is a leaf
= FN(T→LST) + FN(T→RST) ; if T has only 1 child
= FN(T→LST) + FN(T→RST) ; if T is full node

```
int FN (struct node *t){
 if (!t) return 0;
 if ( !t->left && !t->right) return 0;
  return ( FN(t->left) + FN(t->right) + (t->left && t->right)?
                    1:0); }
```

Recursive program to find the height of tree

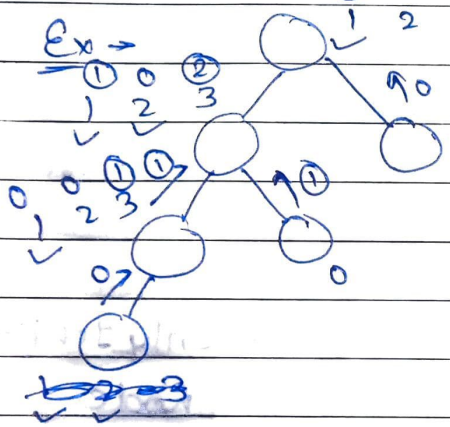
$$H(T) = \begin{cases} 0 ; T \text{ is empty} \\ 0 ; T \text{ is leaf} \\ 1 + \max(H(LST), H(RST)) ; \text{otherwise} \end{cases}$$

Here we are assuming that height of a leaf is 0 but it may vary according to question.

```
int H(struct node *t) {
  if (!t) return 0;
  if ( !t->left && !t->right) return 0;
    int l, r;
1.  l = H(t->left)
2.  r = H(t->right)
3.  return (1 + max(l,r)); }
```

max(l,r) = (l>r) ? l : r;

② 0 ③ → ③ will be return
1 2 3

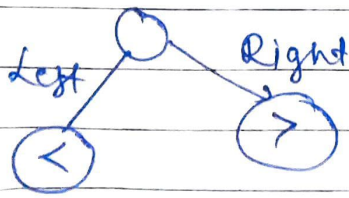


Time Complexity = O(n)
Space Complexity = O(n)

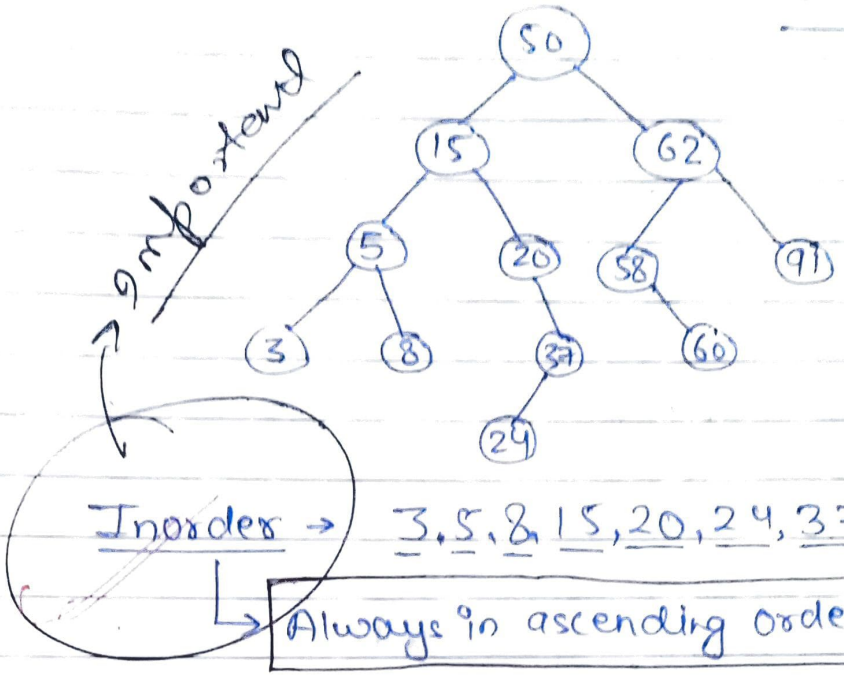
Introduction to BST

Special kind of Binary tree. In balanced BST time complexity may be O(log n).

Ex Gate(96) 50, 15, 62, 5, 20, 58, 91, 3, 8, 37, 60, 24



BST used to store the keys to access the data, and no two data have the same keys.



Every sub tree is also a BST

Process → starting from Root and move on accordingly by comparison.

Inorder → 3, 5, 8, 15, 20, 24, 37, 50, 58, 60, 62, 91

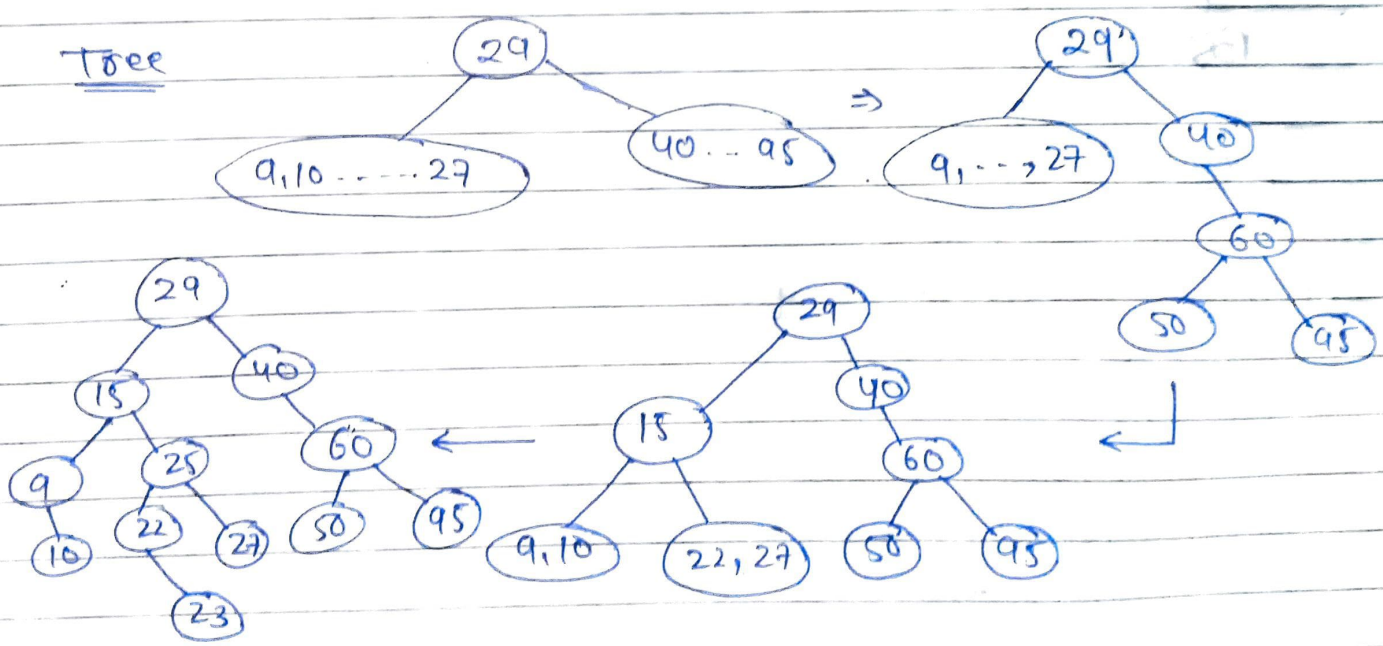
↳ Always in ascending order (sorted)

Gate - 05 (Question on BST)

Post Order :- 10, 9, 23, 22, 27, 25, 15, 50, 95, 60, 40, 29.

In Order :- 9, 10, 15, 22, 23, 25, 27, 29, 40, 50, 60, 95

Tree



Pre-Order → 29, 15, 9, 10, 25, 22, 23, 27, 40, 60, 50, 95

Date :
Page No.

Gate-05 (Number of BST possible)

How many BST are possible with 4 keys?

No. of BST with n distinct keys $= \frac{2nC_n}{(n+1)}$

Because.

Total no. of structure unlebbeled tree possible
$= \frac{2nC_n}{(n+1)}$

Each BST have sorted Inorder.

Each Structure have n! lebelled BST but only 1 have sorted Inorder.

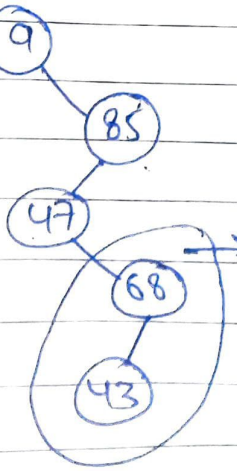
So, $\frac{2nC_n}{(n+1)}$ trees will have $\frac{2nC_n}{(n+1)} \times 1$ trees.

So, unlebelled trees with n nodes and BST with n distint keys are same.

Ans $\boxed{\frac{8C_4}{5}}$.

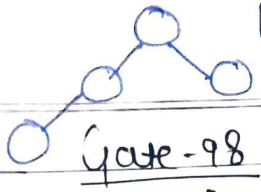
Gate (05) → Search sequence

(c)



→ All the nodes on right side of 47 must be greater than 47 but 43 is lesser, so sequence is not valid.

Heap → It is a almost complete binary tree but not complete binary tree

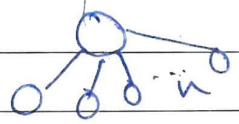


Date
Page No.

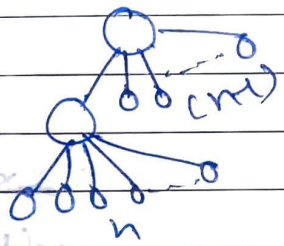
Gate-98 · A complete n-ary tree is one ---- given by-
(a) $x(n-1)+1$ (b) $xn-1$ (c) $xn+1$ (d) $x(n+1)$

N-ary tree (Complete) → A complete n-ary tree is one in which every node has '0' or 'n' children.

Soln for a n-ary tree with I internal node



1 internal node → n leaves



for 2 internal nodes = $n+(n-1)$
$= 2n-1$

for 3 internal nodes = $(2n-1)-1+n = 3n-2$
— 4 — = $(3n-2)-1+n = 4n-3$
|
— x — = $(xn)-(x-1) = x(n-1)+1$

Gate-2002 The number of leaf ---- 0 or 3 is.
Let L be the number of leafs and I be no. of Internal nodes then,

$L = I(n-1)+1$ [substitute $I \to x$ in above eqn]
$L+I = n$

Solving both

$I = \frac{(n-1)}{3}$, $\boxed{L = \frac{(2n+1)}{3}}$ Ans//

Total no. of nodes = Leaf node + Internal nodes.

Max. height of AVL tree with n nodes $= \lceil \log_2 n \rceil$

Recursive program on testing whether the tree is complete binary tree or not.

1 → True
0 → False

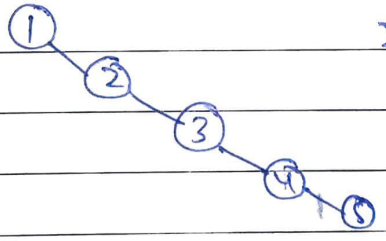
```
int iscomplete (struct node *t)
{
    if (t==NULL) return 1;
    if ( !t->left && !t->right) return 1;
    else if (t->left && t->right)
        return ( iscomplete (t->left) && iscomplete (t->right));
    else
        return 0;
}
```

Complete means either 0 or 2 child

Introduction to AVL trees and balancing:

BST introduced to make search easier or O(log n) but sometimes BST also have search as O(n)

Ex 1 2 3 4 5



In this case there is no use of tree bcoz it will also acts as a linked list and it is unbalanced.

To deal with these kind of unbalanced tree the concept of AVL tree introduced.

AVL is a balanced binary search tree but the height is O(log n)

Other balanced BST are (R-B), (2-3) tree.

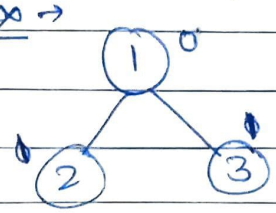
AVL is based on balanced factor BF (-1, 0, 1)

(-2, 2) → Unbalanced

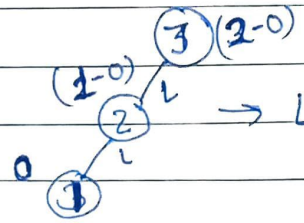
Date :
Page No.

Balanced factor = Height (LST) − Height (RST)

Ex →

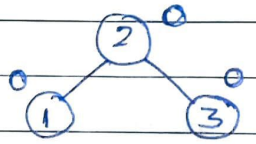


(Valid)

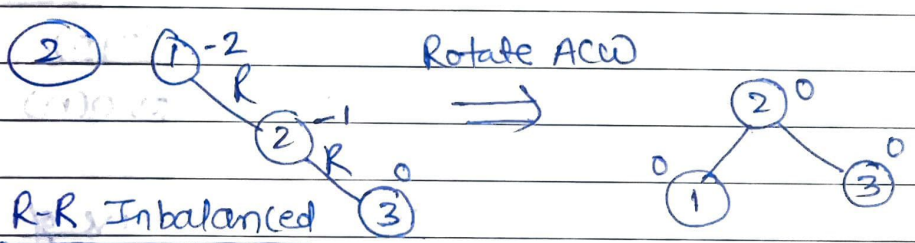


→ L-L Inbalanced

(1) In L-L Inbalanced rotate clockwise

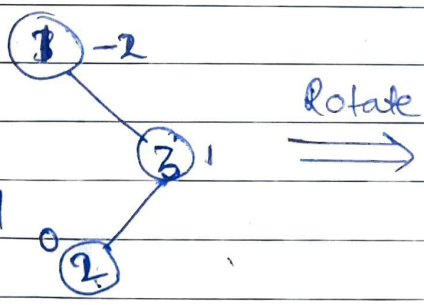


After Insertion we check every node should be balanced if it is inbalanced then balanced it.

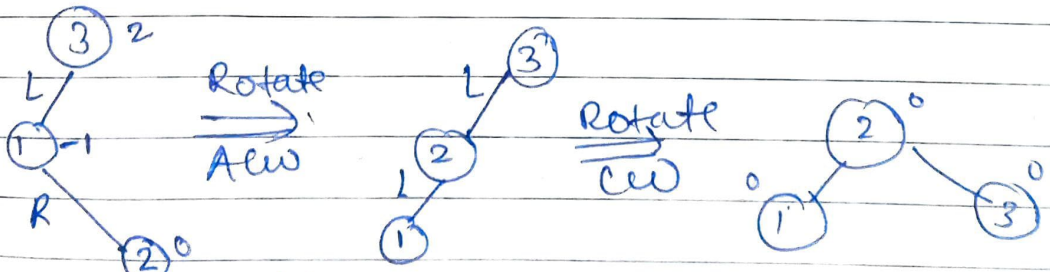


R-R Inbalanced

Ex



(4)

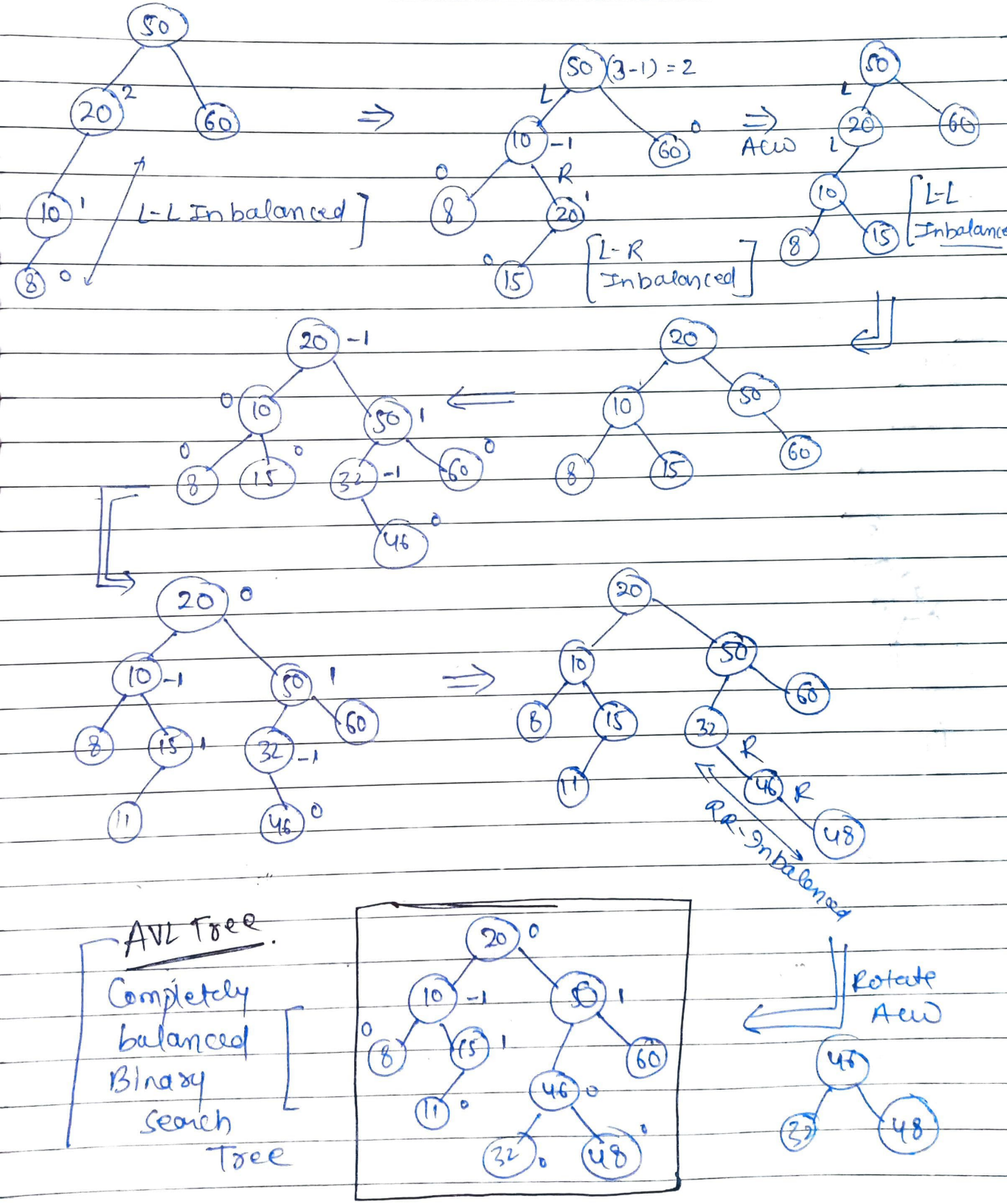


(L-R Inbalanced)

Date :
Page No.

Constructing AVL tree and time complexity Analysis

50, 20, 60, 10, 8, 15, 32, 46, 11, 48 (Insert element in BST)



AVL Tree.

Completely balanced Binary Search Tree

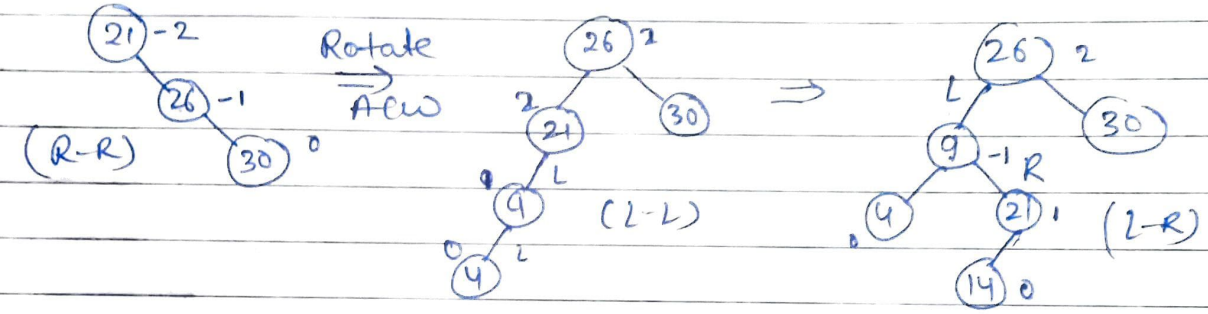
L-R → Convert it into L-L

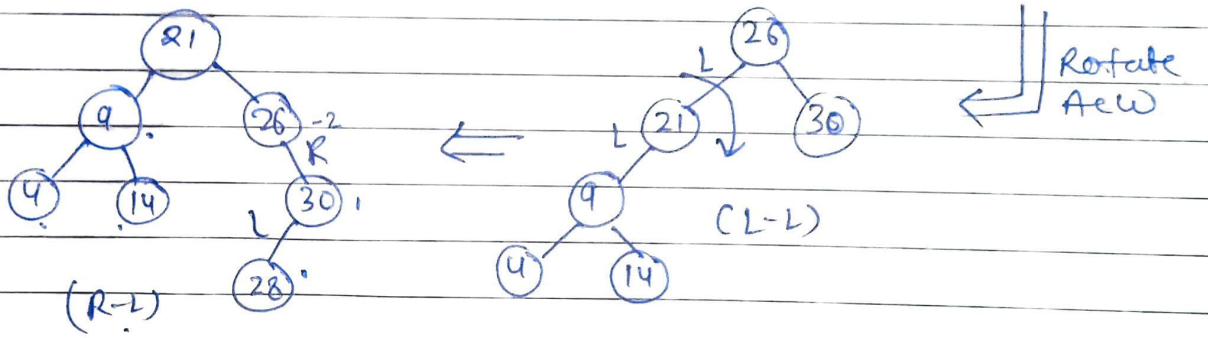
R-L → Convert it into R-R

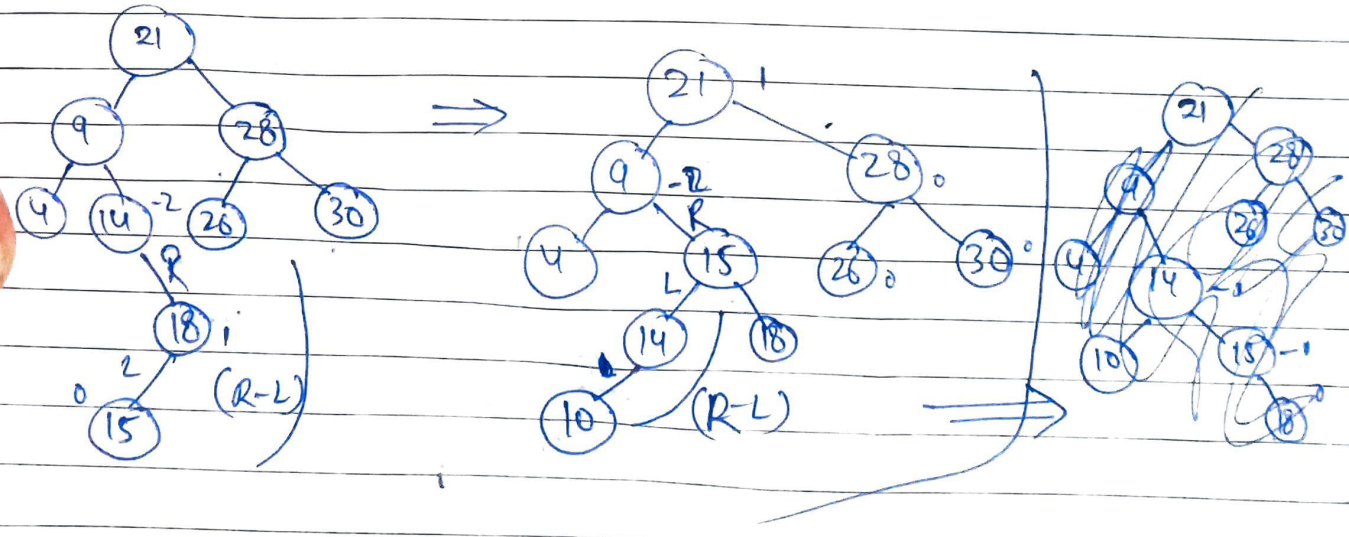
	BST	BBST	
Search	O(n)	O(logn)	→ Seach time depends on height
Height	O(n)	O(logn)	→ Worst Case
Insertion	O(n)	O(logn) + O(logn)	→ may be at the end in BST but in BBST O(logn) for insertion and O(logn) for balancing.

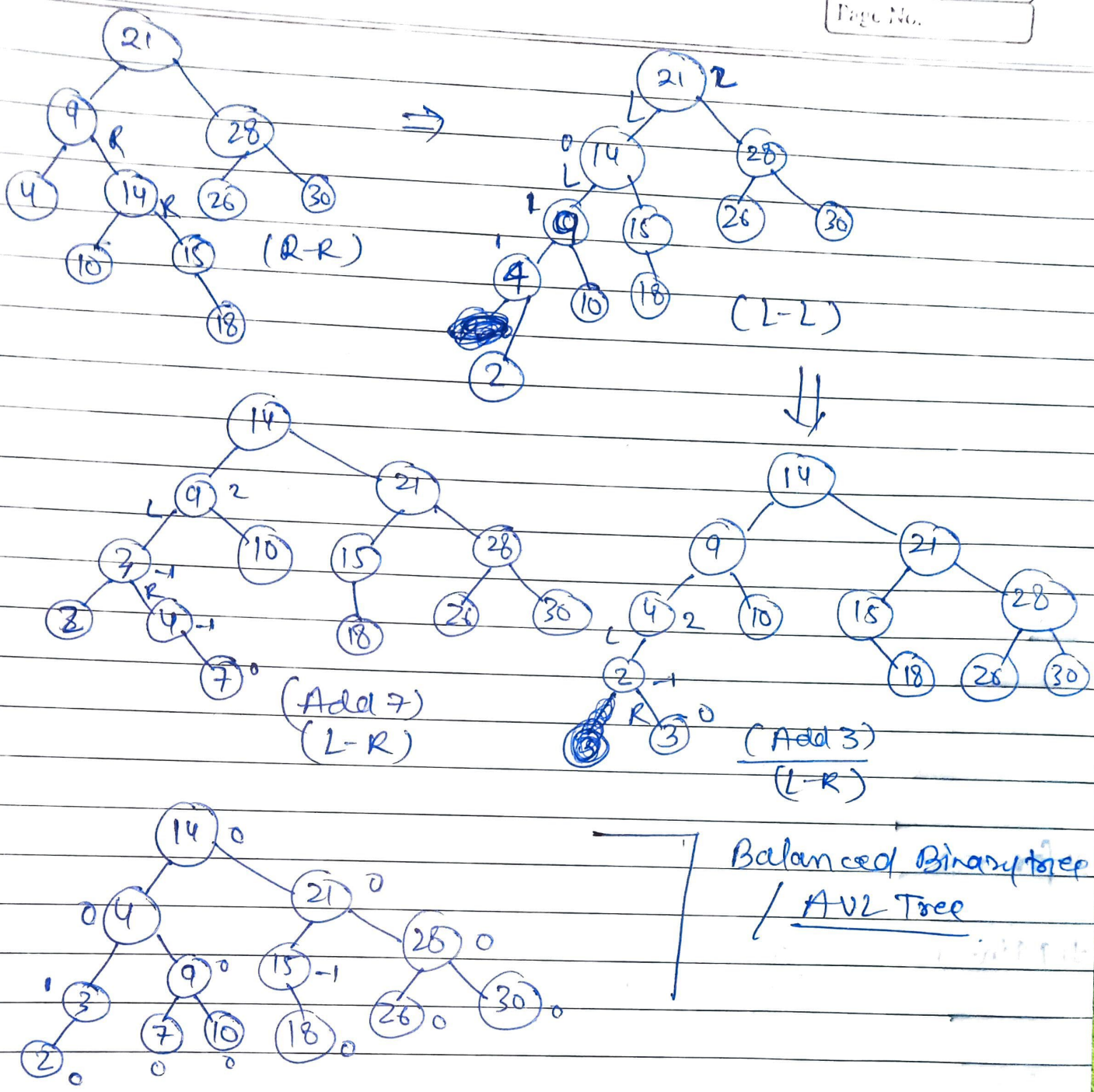
Construction of AVL tree

21, 26, 30, 9, 4, 14, 28, 18, 15, 10, 2, 3, 7.



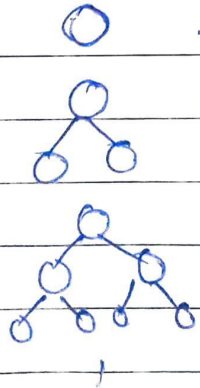






Balanced Binary tree / AVL Tree

Min and max nodes in AVL tree of height h



$\rightarrow 1$

$\rightarrow 1+2^1$

$\rightarrow 1+2^1+2^2$

for height h $= 1+2^1+2^2+\cdots+2^h$

max. no. of nodes for height h $= \frac{1(2^{h+1}-1)}{2-1}$

$= \frac{2^{h+1}-1}{1}$

↓ for binary tree

9 is a gp so