Algorithm : Del_bin_tree (INFO, LEFT, RIGHT, ROOT, AVAIL, ITEM)

The algorithm delete ITEM from tree T (binary search tree)

1. [Find the location of ITEM & its parent, using procedure]

   Call Find_bin (INFO, LEFT, RIGHT, ROOT, ITEM, LOC, PAR)

2. [ITEM in tree ?]

   If LOC = NULL, then:

      Write : ITEM not in tree, & Exit

3. [Delete node containing ITEM]

   If RIGHT[LOC] ≠ NULL & LEFT[LOC] ≠ NULL, then:

      Call CASE B (INFO, LEFT, RIGHT, ROOT, LOC, PAR)

   Else

      Call CASE A (INFO, LEFT, RIGHT, ROOT, LOC, PAR)

   [End of if statement]

4. [Return deleted node to the AVAIL list]

      Set LEFT[LOC] := AVAIL & AVAIL := LOC

5. EXIT

Procedure : CASEB (INFO, LEFT, RIGHT, ROOT, LOC, PAR)
This procedure will delete N at loc LOC, where N has two children. PAR (pointer) is location of parent of N, SUC (pointer) is the morder successor of N, PAR-SUC (pointer) gives the location of parent of morder successor.

1> [Find SUC & PAR-SUC]
   (a) Set PTR := RIGHT[LOC] & SAVE := LOC
   (b) Repeat while LEFT[PTR] ≠ NULL
         Set SAVE := PTR & PTR := LEFT[PTR]
       [End of loop]
   (c) Set SUC := PTR & PAR-SUC := SAVE

2> [Delete inorder successor, using CASEA()]
   Call CASEA (INFO, LEFT, RIGHT, ROOT, SUC, PAR-SUC)

3> [Replace N by its in-order successor]
   (a) If PAR ≠ NULL, then:
         If LOC = LEFT[PAR], then:
            Set LEFT[PAR] := SUC
         Else
            set RIGHT[PAR] := SUC
       [End of if structure]
   Else
      set ROOT := SUC
   [End of if structure]
   (b) Set LEFT[SUC] := LEFT[LOC]
       & RIGHT[SUC] := RIGHT[LOC]
   ] updating pointers

Replace N by SUC

4> Return

Now Case I & Case II are covered in procedure A whereas Case III ~~or~~ will be covered in prodedure B

Procedure : CASEA (INFO, LEFT, RIGHT, ROOT, LOC, PAR)

This Procedure delete node N at location LOC, when N doesn't have two child. PAR (pointer) give the location of the parent of N. CHILD (pointer) gives the location of the only child of N.

1. [Initialized CHILD]

If LEFT [LOC] = NULL & RIGHT [LOC] = NULL, then:

Set CHILD := NULL

Else if LEFT [LOC] ≠ NULL, then:

Set CHILD := LEFT [LOC]

Else :

Set CHILD := RIGHT [LOC]

[End of If structure]

2. [Delete by (updating) assigning CHILD]

If PAR ≠ NULL, then

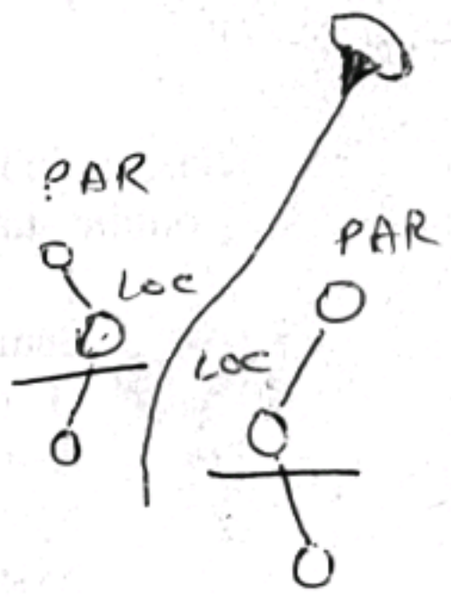
If LOC = LEFT [PAR], then:

Set LEFT [PAR] := CHILD

Else

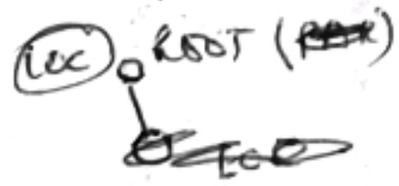
Set RIGHT [PAR] := CHILD

[End of if structure]



Else

Set ROOT := CHILD [where CHILD = NULL in this case]



[End of If structure]

3. Return

DELETING IN A BINARY SEARCH TREE

Deletion of algorithm uses procedure FIND() to find the location of node N (to be deleted) & also the location of the parent node P(N).

- Deletion of node depends upon three cases:

Case I : N has no children.

Action : simply replace N by null pointer.

Case II : N has exactly one child.

Action : Replacing N ~~in P(N)~~ by the location of the only child of N

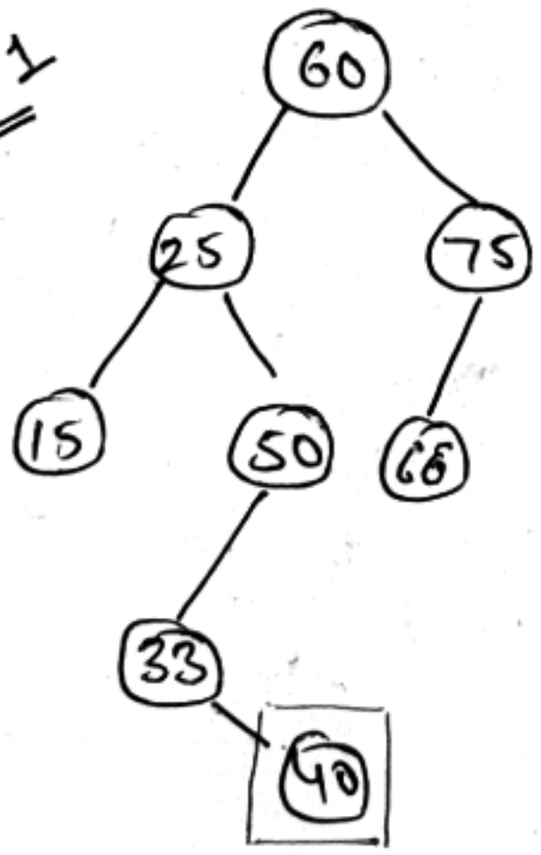
Case III : N has two children.

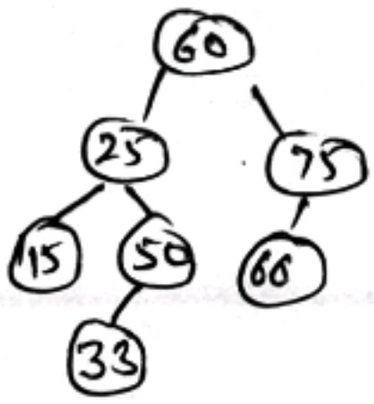
Assumption Action : Let S(N) be inorder successor of N (has no left child)

- Verify S(N) doesn't have a left child

Action : Replacing N by S(N)

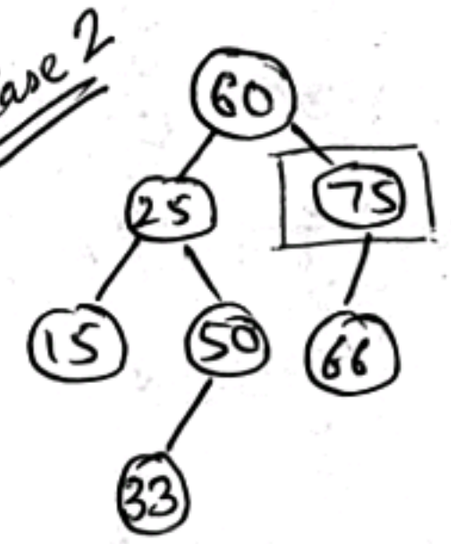
CASE 1

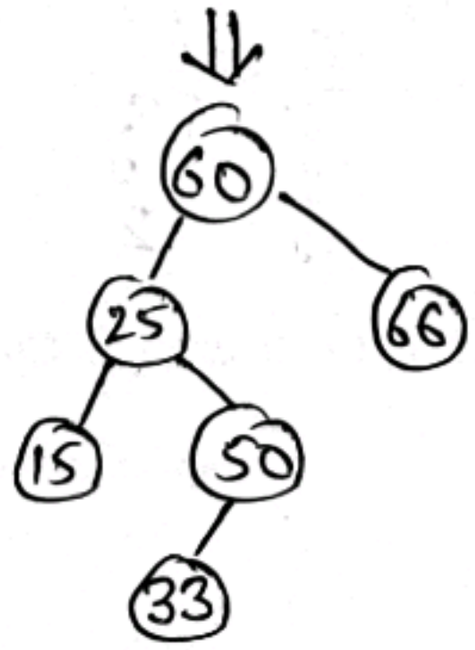




40 deleted

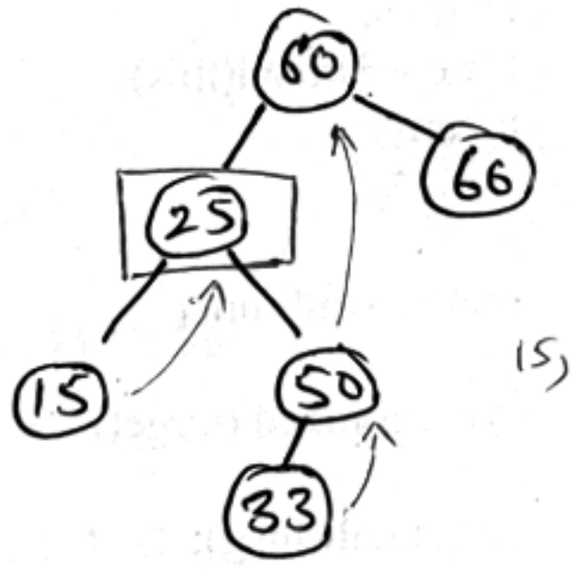
Case 2



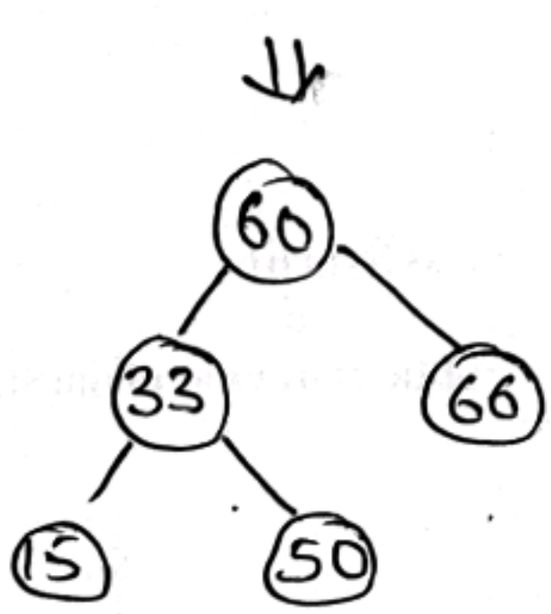


75 deleted

Case 3



15, 25, 33, 50, 60, 66



25 deleted.

Algorithm: Insert_bin (INFO, LEFT, RIGHT, ROOT, AVAIL, ITEM, LOC)

1) Call Find_bin (INFO, LEFT, RIGHT, ROOT, ITEM, LOC, PAR)

2) If LOC ≠ NULL, then Exit (item already exists)

3) (a) If AVAIL = NULL, then: Write: "OVERFLOW" & Exit

(b) Set NEW := AVAIL, AVAIL := RIGHT[AVAIL]

INFO[NEW] := ITEM

(c) Set LOC := NEW

LEFT[NEW] := NULL & RIGHT[NEW] = NULL

4) If PAR = NULL, then: [Tree is empty]

Set ROOT := NEW

Else if ITEM < INFO[PAR], then:

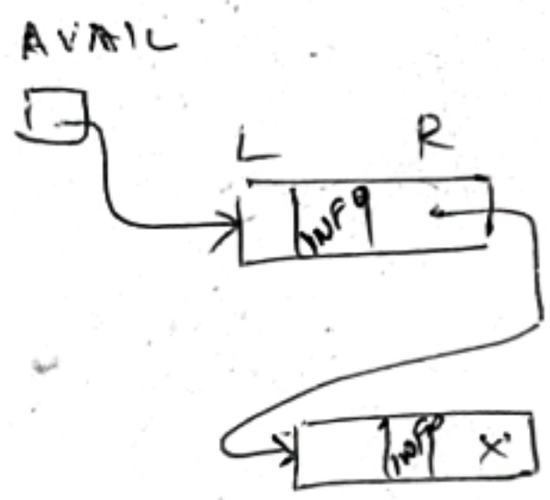
Set LEFT[PAR] := NEW

Else

Set RIGHT[PAR] := NEW

[End of if structure]

5) EXIT.



APPLICATION : → finding & deleting all duplicates in the collection. (complexity $O(n \log_2 n)$

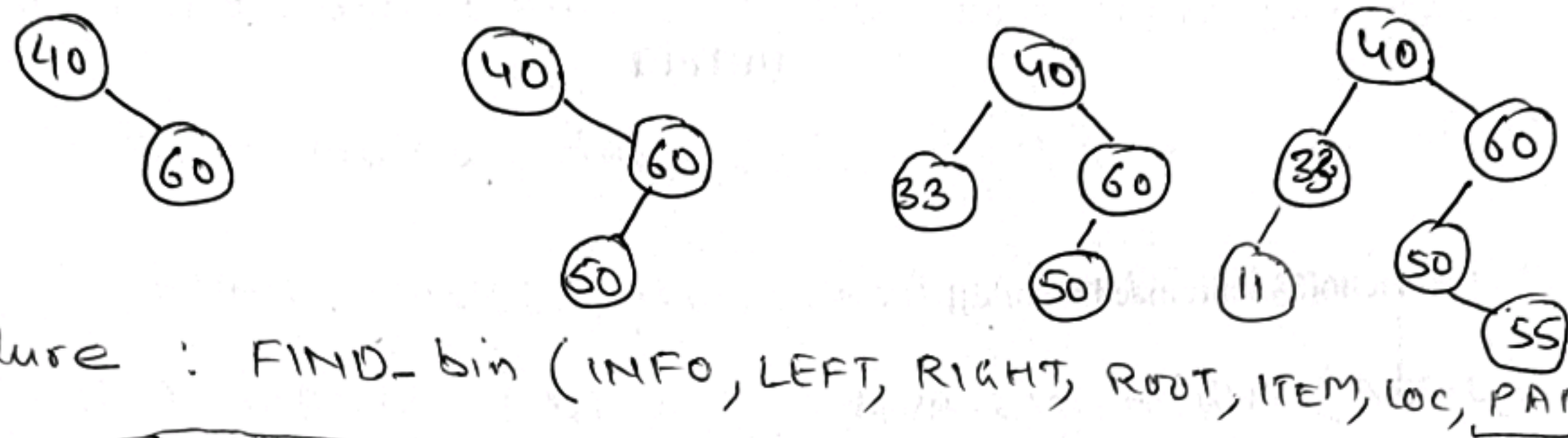
average length of such a branch of tree is approx

$(c \log_2 n)$ * no of elements n

Total no of comparison

Eg) make binary tree with elements

40, 60, 50, 33, 55, 11 (order matters)



Procedure : FIND_bin (INFO, LEFT, RIGHT, ROOT, ITEM, LOC, PAR)

Parent

Tree empty

<1> If ROOT = NULL, then Set LOC := NULL & PAR := NULL & return

Item is root

<2> If ITEM = INFO[ROOT], then: Set LOC := ROOT, PAR := NULL & return

Initialize the PTR & SAVE

<3> If ITEM < INFO[ROOT], then

Set PTR := LEFT[ROOT] & SAVE := ROOT

Else:

Set PTR := RIGHT[ROOT] & SAVE := ROOT

[End of if structure]

<4> Repeat step 5 & 6 while PTR ≠ NULL

ITEM found

<5> If ITEM = INFO[PTR], then: Set LOC := PTR & PAR := SAVE & return

<6> If ITEM < INFO[PTR], then: Set SAVE := PTR & PTR := LEFT[PTR]

Else Set SAVE := PTR & PTR := RIGHT[PTR]

[End of if structure]

[End of step 4 loop]

<7> Set LOC := NULL & PAR := SAVE [ITEM not found in tree]

<8> EXIT

BINARY SEARCH TREE

Pros / Cons

Sorted linear array : running time of searching is less ($O \log_2 n$)
but expensive to insert & delete elements.

Linked list : running time of searching is expensive ($O(n)$)
but insertion & deletion is easy

A Binary tree T is called binary search tree (or binary sorted tree) if each node N has following :

— The value of N .

* is greater than every value in left subtree of N
& * is less than every value in the right " " ".
(The inorder traversal of T will yield a sorted list)

Analogue def: same excepts N is less than or equal to every value in the right subtree of N.

SEARCHING & INSERTING IN BINARY TREE

Traversing in binary search tree is the same as traversing in any binary tree.

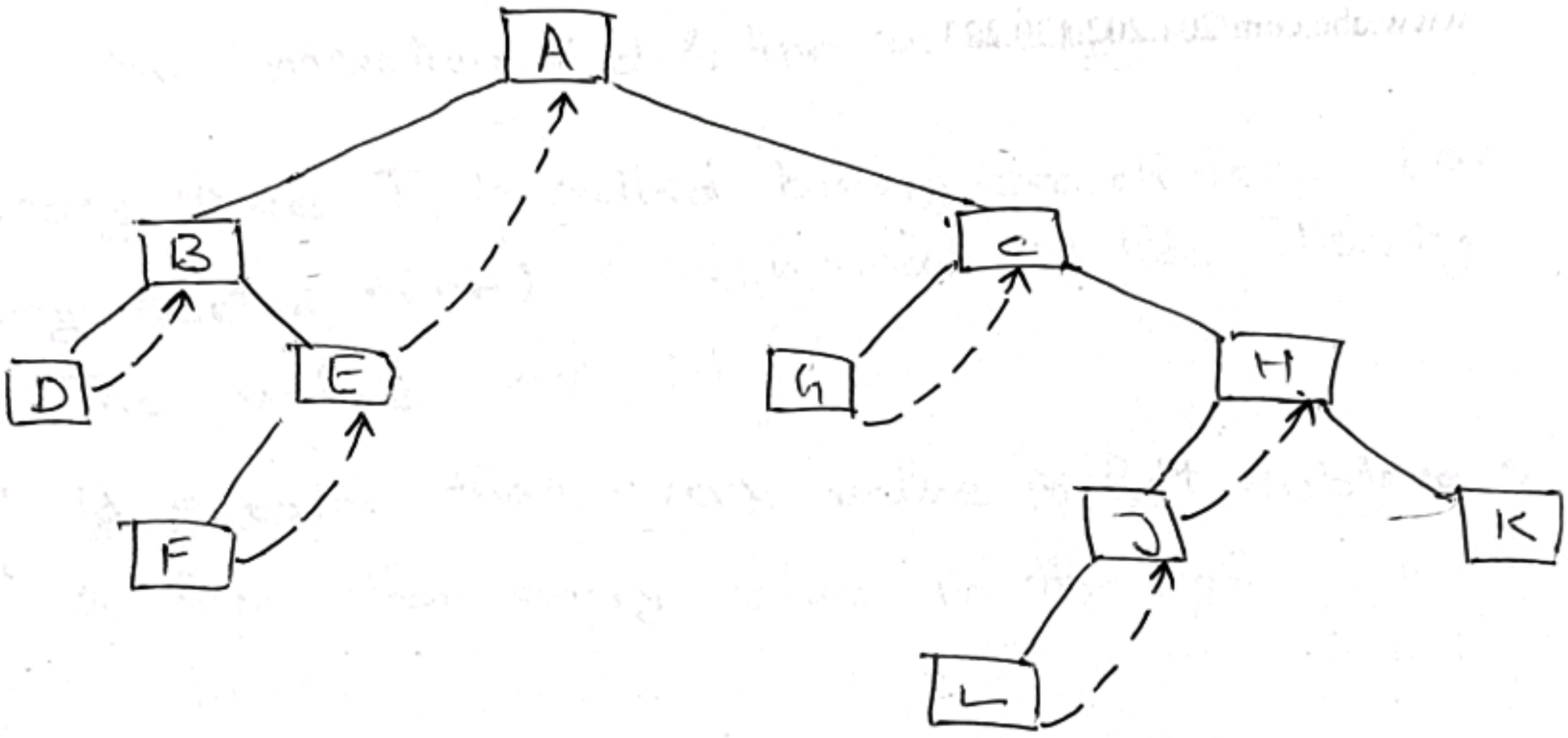
Inserting :

(a)(i) If ITEM < N, proceed to the left child of N.
(ii) If ITEM > N, proceed to the right child of N.

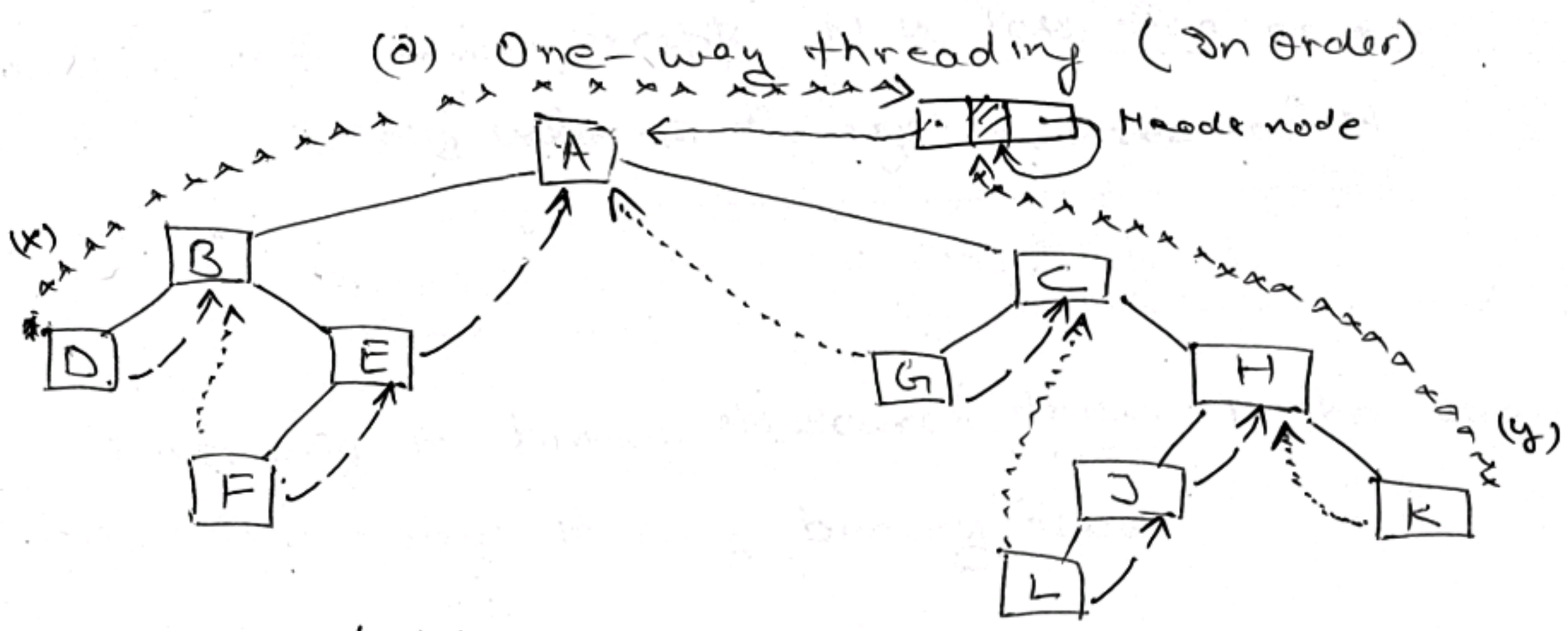
(b) Repeat step (a) until one of following occurs
(i) We have ITEM = N
(ii) We meet empty subtree (we place ITEM in place of empty subtree).

→ The left pointer field of first node & the right pointer field of last node ——— will contain the NULL values; when T doesn't have header node

→ If header node is present then both points to it.



(a) One-way threading (In order)



(b) Two way threading (in order)

(If header is absent the point x & y will not there

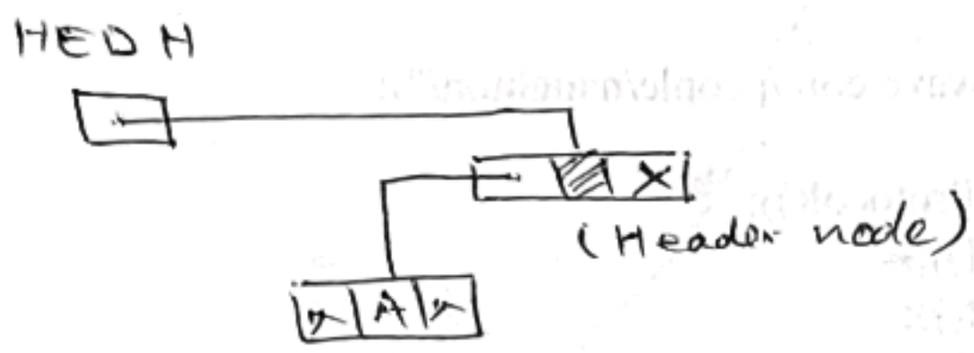
Eg:

HEAD 10

	1	2	3	4	5	6	7	8	9	10	11	12
INFO	A		B		E			F				D
LEFT	3		12		-1					1		
RIGHT			5		8					10		

Header nodes, Threads (of trees)

Header node is added to the beginning of T. The tree pointer variable which we call as HEAD (instead of ROOT) which points to the header node & left pointer of header node will point to the root of T.



Another variation of binary tree T is use the header node as a sentinel.

i.e The pointer field of header node/ subtree will contain a address of the header node instead of null values. i.e LEFT[HEAD] = HEAD ⇒ empty tree

THREADS (Inorder threading)

→ Approximately half of the entries in the pointer fields LEFT & RIGHT will contain null elements.

→ We replace certain NULL entries by special pointers which points to the node higher in the tree

→ These special pointers are called threads.

→ A binary tree with such pointers are called threaded tree

→ In computer memory an extra 1-bit TAG field may be used to distinguish threads from ordinary pointers. (or threads may be denoted by -ve integers)

Type of threading

* One - way threading or
* Two - way threading

Unless otherwise stated, our threading will correspond to the inorder ~~trans~~ traversal of T.

Algorithm: POST_ORD (INFO, LEFT, RIGHT, ROOT)

7.13

1. [Push NULL to stack & initialize PTR]

   Set TOP := 1 , STACK[1] := NULL & PTR := ROOT

2. [Push left-most path onto STACK]

   Repeat Steps 3 to 5 while PTR ≠ NULL

3. Set TOP := TOP + 1 & STACK[TOP] := PTR

4. If RIGHT[PTR] ≠ NULL, then:

      Set TOP := TOP + 1 & STACK[TOP] := -RIGHT[PTR]

   [End of if structure]

5. Set PTR := LEFT[PTR]

   [END of step 2 loop]

Pop the node

6. Set PTR := STACK[TOP] & TOP := TOP - 1

Pop node from STACK

7. Repeat while PTR > 0 :

      (a) Apply PROCESS to INFO[PTR]

      (b) Set PTR := STACK[TOP] & TOP := TOP - 1

8. If PTR < 0, then

      (a) Set PTR := -PTR

      (b) Go to Step 2.

   [End of if structure]

9. EXIT

POSTORDER TRAVERSAL

7.12

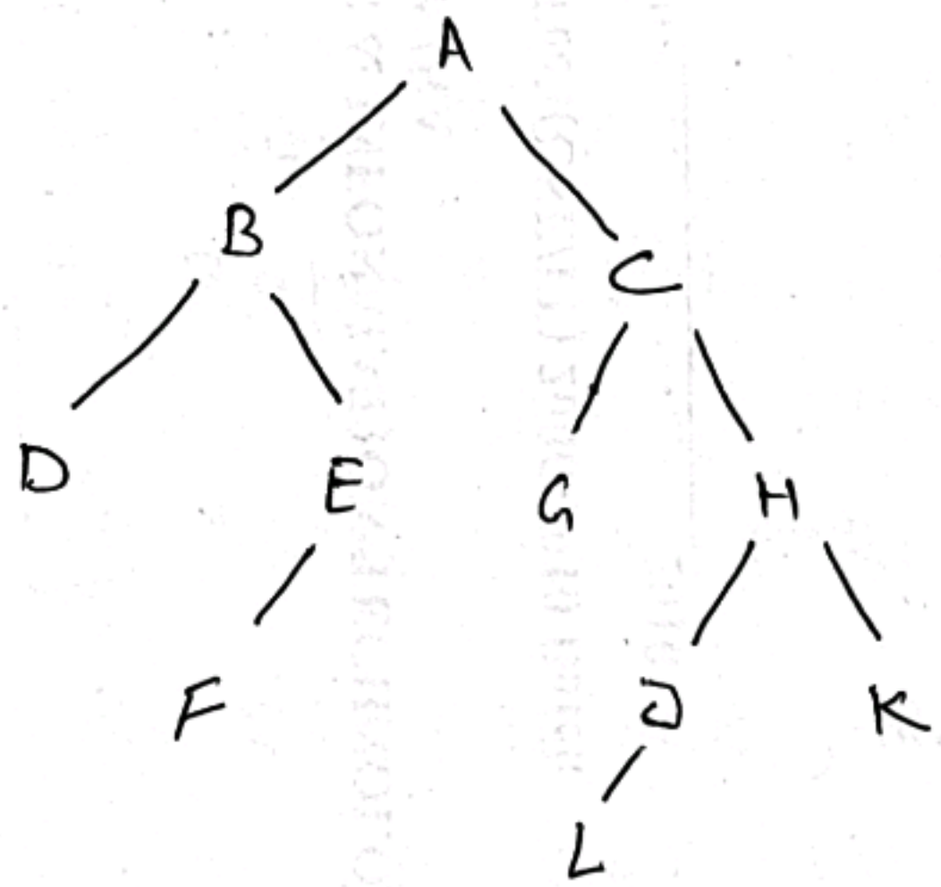
Method

- Initially push NULL to stack (as sentinel) & PTR := ROOT

Repeat following steps until NULL is POPED from STACK

(a) Proceed down Leftmost path rooted at PTR,
push N onto STACK.
if [N Has right child R(N)] then:
push - R(N) onto STACK

(b) POP & process +ve nodes onto STACK.
If (NULL is popped) : then EXIT
{PTR = -N} If (-ve node is "): set PTR = N {By assignment PTR := -PTR}
-(-N) = N & return to step (a)



STACK : ϕ

(a) STACK : ϕ A, -C, B, -E, D

(b) P : D, -(-E) ⇒ P : D, E

(a) STACK : ϕ A -C B E F

P : D F E B

STACK : ϕ A C -H G

P : D F E B G

STACK : ϕ A C H

STACK : ϕ A C H -K J L

P : D F E B G L J

STACK : ϕ A C H K

P : D F E B G L J K H C A

Required Post order traversing solution

Algorithm : IN-ORD (INFO, LEFT, RIGHT, ROOT) (7.11)

1. [Push NULL into STACK & initialize PTR]
   Set TOP := 1 , STACK[1] = NULL & PTR := ROOT
2. Repeat while PTR ≠ NULL [Push left most path into STACK]
   (a) Set TOP := TOP + 1 &
   STACK[TOP] := PTR [Save node]
   (b) Set PTR := LEFT[PTR] [Update PTR]
   [End of loop]
3. Set PTR := STACK[TOP] & TOP := TOP - 1 [Pop node from stack]
4. Repeat step 5 to 7 while PTR ≠ NULL
5. Apply PROCESS to INFO[PTR]
6. If RIGHT[PTR] ≠ NULL, then
   (a) Set PTR := RIGHT[PTR]
   (b) Go to Step 4
   [End of if structure]
7. Set PTR := STACK[TOP] & TOP := TOP - 1
   [End of Step 4 loop]
8. EXIT

(b) In-order Traversal

Initially push NULL to stack & PTR := ROOT

Repeat until NULL is poped

(a) Proceed ~~L(N)~~ left most path at PTR, ~~P~~

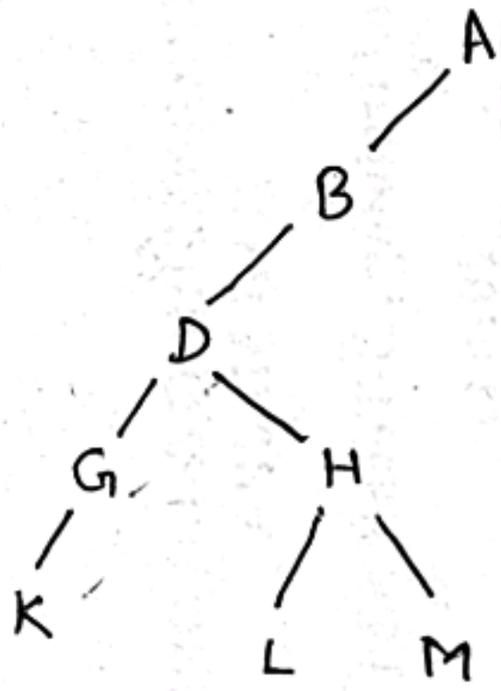
— PUSH each node N onto STACK.

— stop when a node N with no left child (is pushed onto STACK)

(b) [Backtracking] POP & process STACK.

· If NULL is popped, then EXIT.

If R(N) is processed, set PTR = R(N) & return to (a)



STACK : ϕ ; PTR : A

S : ϕ A B D G K ~~G P : K G D~~

$: ϕ A B ——— O/P : K G D

$: ϕ A B H L

$ ϕ A B ——— O/P : K G D L H

$ ϕ A B M : K G D L H

$ ϕ A B ——— K G D L H M

$ ϕ ——— : K G D L H M B A

$ ϕ C E

$ ϕ ——— K G D L H M B A E C

Algorithm : PREORD (INFO, LEFT, RIGHT, ROOT) (7.9)

1. [Push NULL to stack & initialize PTR]

   Set TOP := 1 , STACK [1] := NULL & PTR := ROOT

2. Repeat step 3 to 5 while PTR ≠ NULL

3. Apply PROCESS to INFO [PTR]

4. [Right child ?]

   If RIGHT [PTR] ≠ NULL , then : [Push on STACK]

   { Set TOP := TOP + 1 &

      STACK [TOP] := RIGHT [PTR]

   [End of If structure]

5. [Left child ?]

   If LEFT [PTR] ≠ NULL , then:

      set PTR := LEFT [PTR]

   Else : [Pop from STACK]

      set PTR := STACK [TOP] &

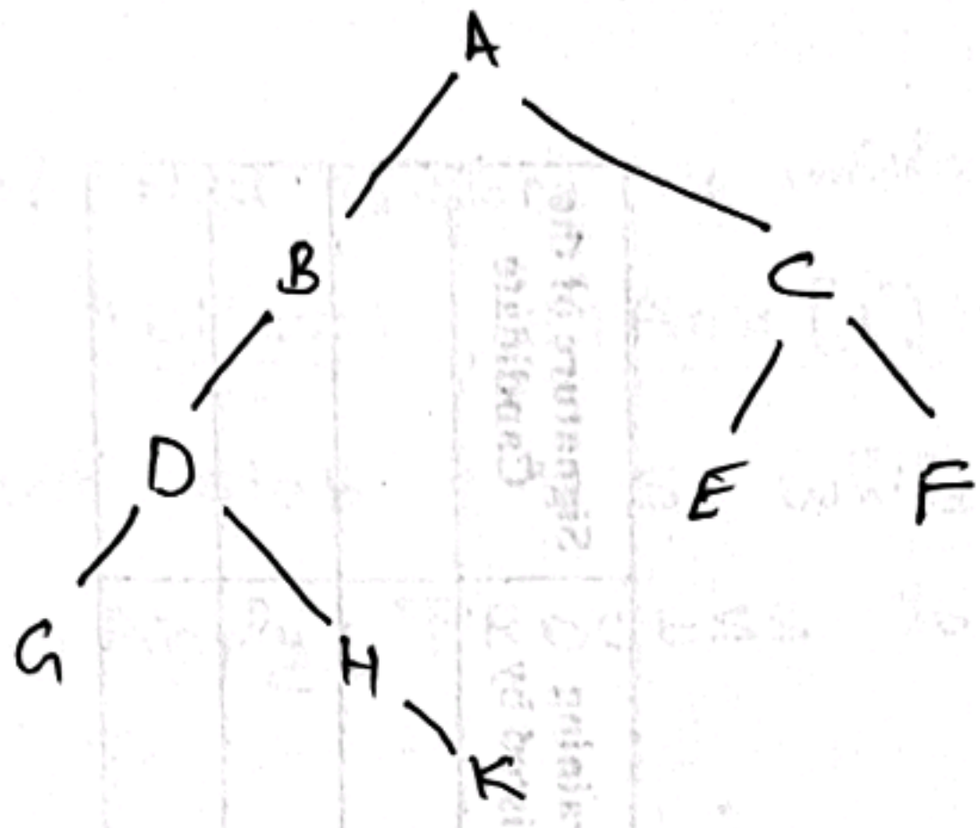
      TOP := TOP - 1

   [End of If structure]

[End of step 2 loop]

6. EXIT

7.8



< PRE-ORDER >

1.> Push NULL into STACK : STACK : ϕ

PTR := A (PTR := ROOT)

PROCESSED : A —— STACK : ϕ C

2> PROCESSED : A B D —— STACK : ϕ H C H

: A B D G —— ϕ H C K

: A B D G H K —— ϕ C

: A B D G H K C —— ϕ F

A B D G H K C E

A B D G H K C E F —— ϕ

↑

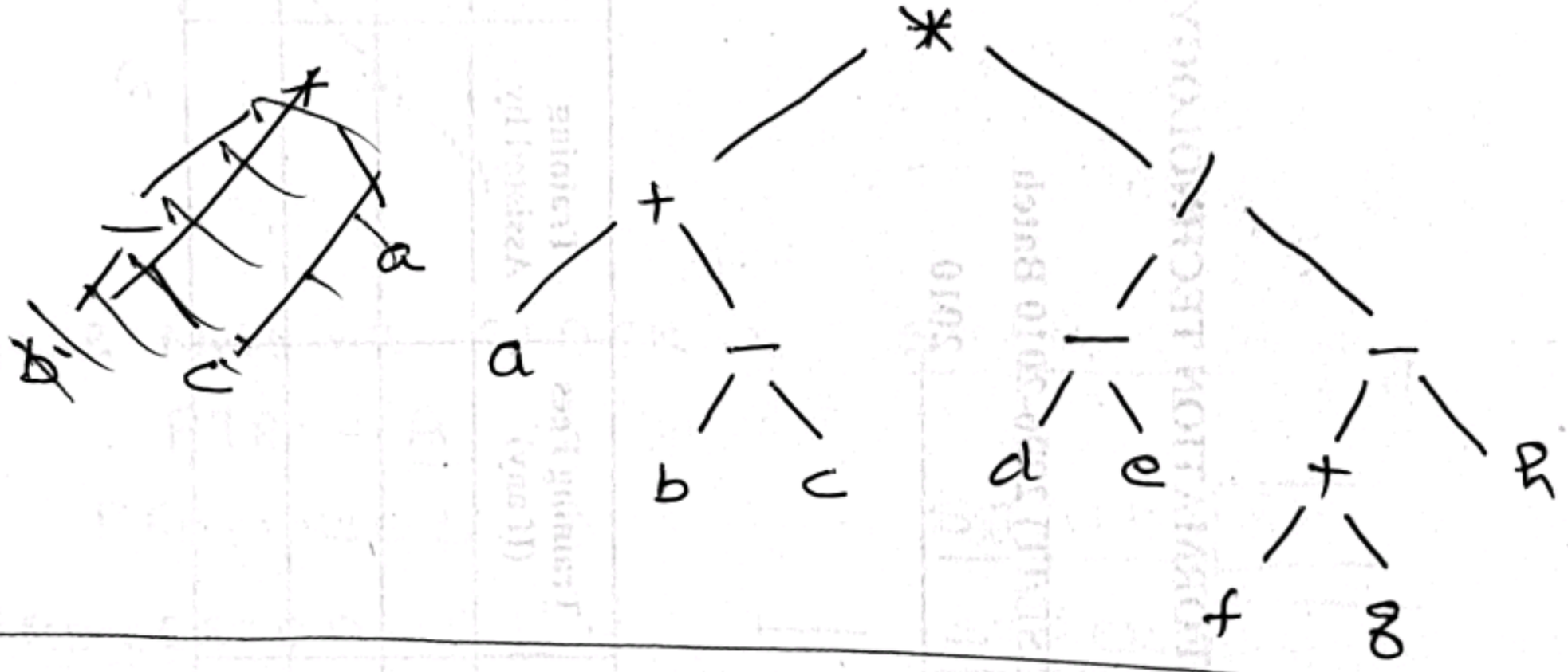
POP the element ϕ (NULL from STACK, set PTR := NULL

Since PTR := NULL, the algorithm is completed.

$$E = [a+(b-c)] * [(d-e)/(f+g-h)]$$

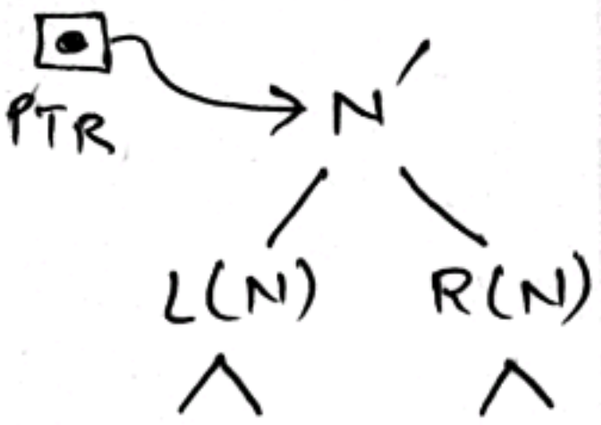
(Prefix) PREORDER: * + a - b c / - d e - + f g h

(Post fix) POST-ORDER: a b c - + d e - f g + h - / *



TRAVERSAL ALGORITHM USING STACK

(d) Pre-order Traversal :- The algorithm use PTR (pointer) which contain location of node N currently being scanned.



STEPS: PUSH NULL in STACK & PTR := ROOT

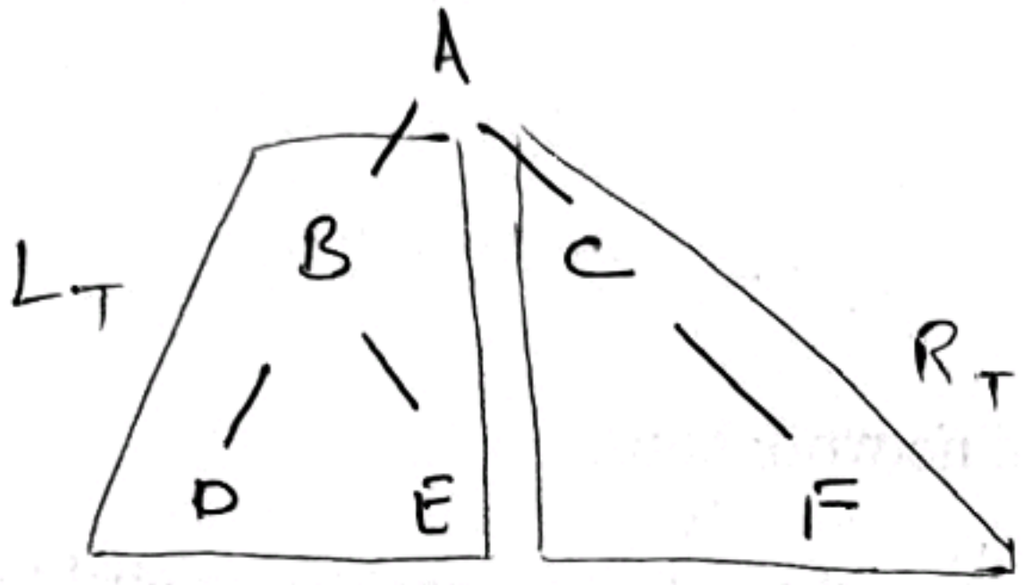
→(2) Proceeds left-most part processing each node N & PUSHING R(N) on STACK until not L(N) is processed)

→(b) POP & assign to PTR the TOP of STACK.

if PTR ≠ NULL, then return to Step(2)

otherwise EXIT

STACK[NULL] is used as sentinel.

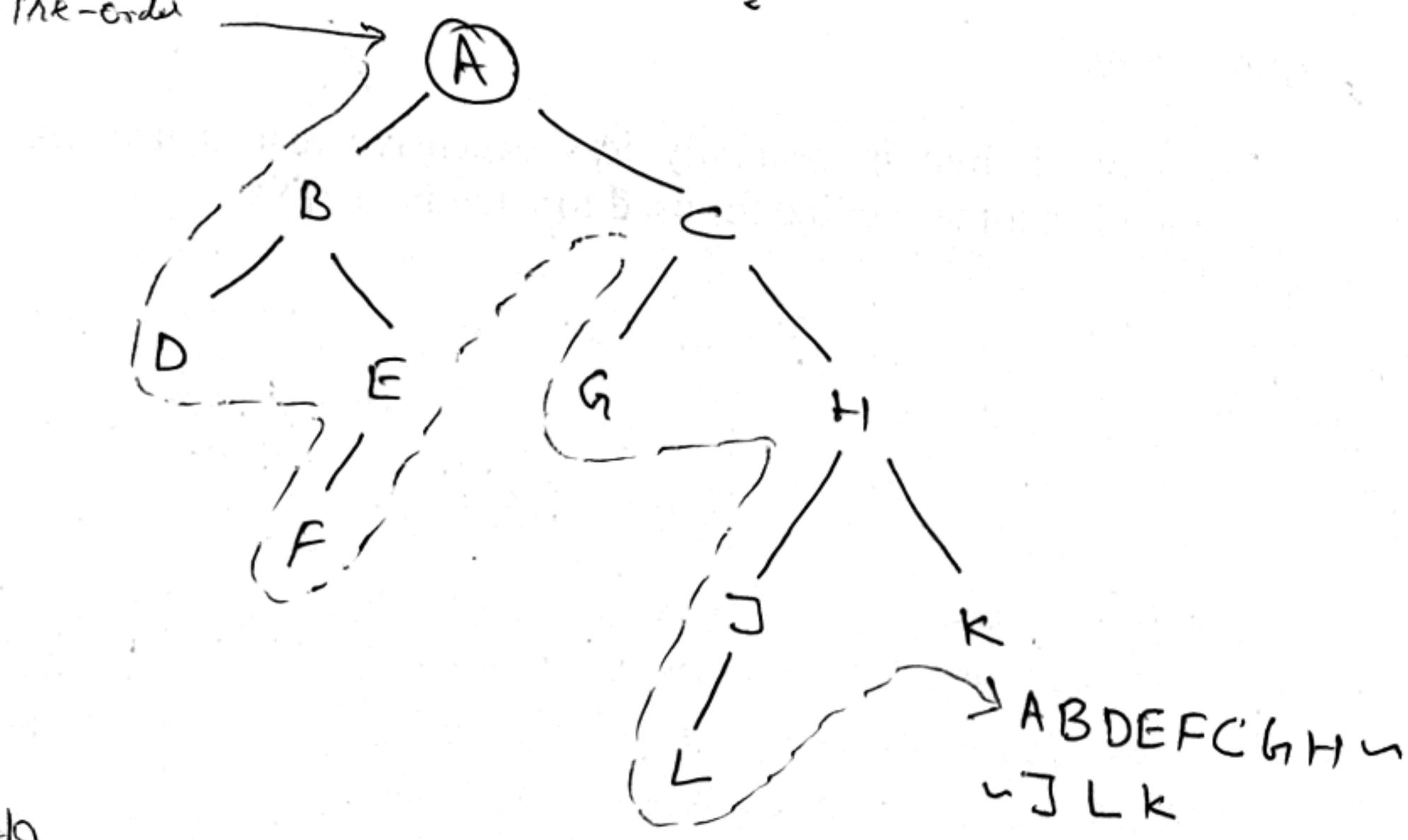


* It defines the processing of root

Pre-order: (A), L_T, R_T ⇒ A B̄ D E̲ C F
In-order: L_T, (A), R_T ⇒ D B̄ E̲ A C F
Post-order: L_T, R_T, (A) ⇒ D E̲ B̄ F C A

} * B is the root of L_T

———— :O: ————



→ (In order): D B F E A G C L J H K

→ (Post-order): D F E B G L J K H C A

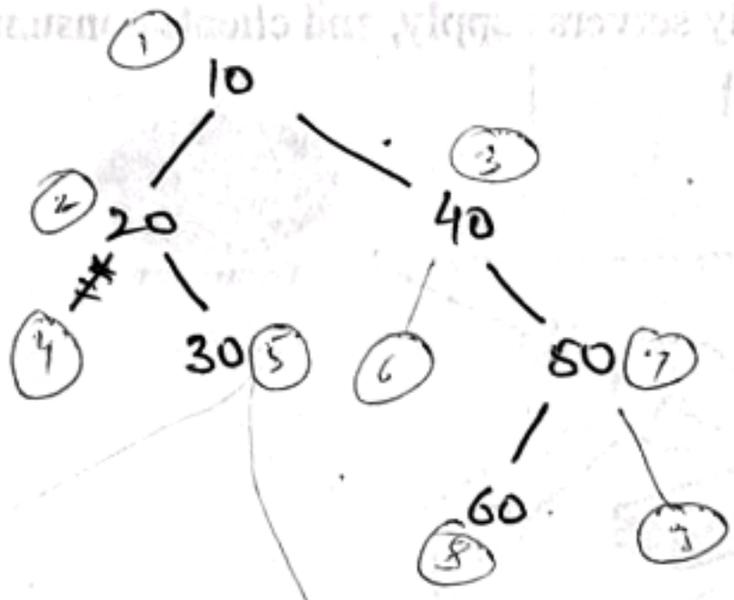
(b) Sequential Representation of Binary tree

This representation use only a single linear array. TREE[] as follows:

(a) The root R of T is stored in TREE[1]

(b) If a node N occupies TREE[k], then
its left child stored in TREE[2*k] &
its right child stored in TREE[(2*k)+1]

*NULL is used to indicate an empty sub-tree.



1	2	3	4	5	6	7	8	9
10	20	40		30		50	60	

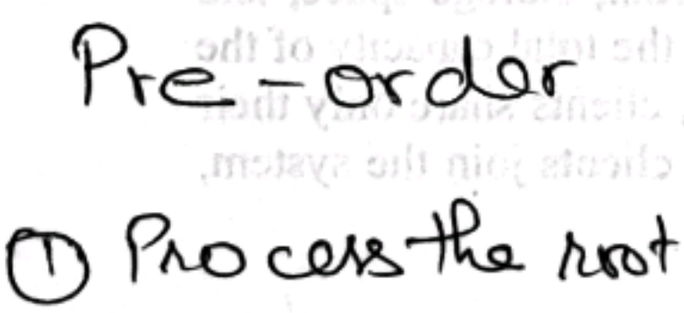
TREE[ ]

tree with depth d will require an array with approximately 2^{d+1}. (some time its ineffient if depth is greater but not complete Bin tree)

TRAVERSING BINARY TREES

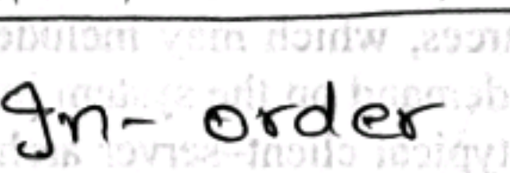
Pre-order

(1) Process the root R
(2) Traverse the left subtree of R in pre-order
(3) Tranverse the right subtree of R in pre order



In-order

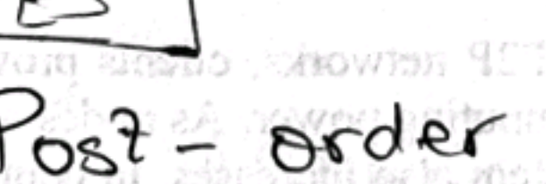
(1) left subtree in inorder
(2) root R
(3) Right sub-tree in in order



LNR

Post-order

(1) left subtree in post order
(2) Right subtree in post order
(3) Process root R

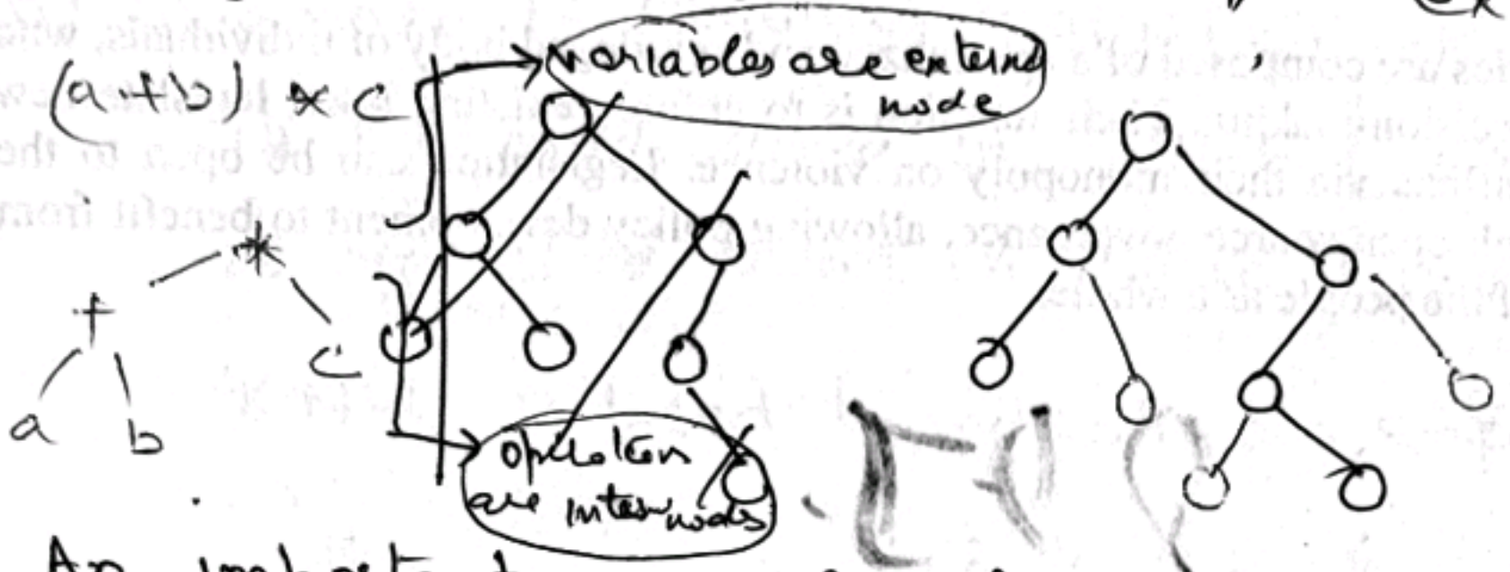


LRN

EXTENDED BINARY TREES : 2-TREES (7.4)

If each node N has either 0 (zero) or 2 children.

* Node with two children called internal nodes.
* Node with zero " " external nodes.



An important example of a 2-tree is the tree T corresponding to any algebraic expression E which uses only binary operations

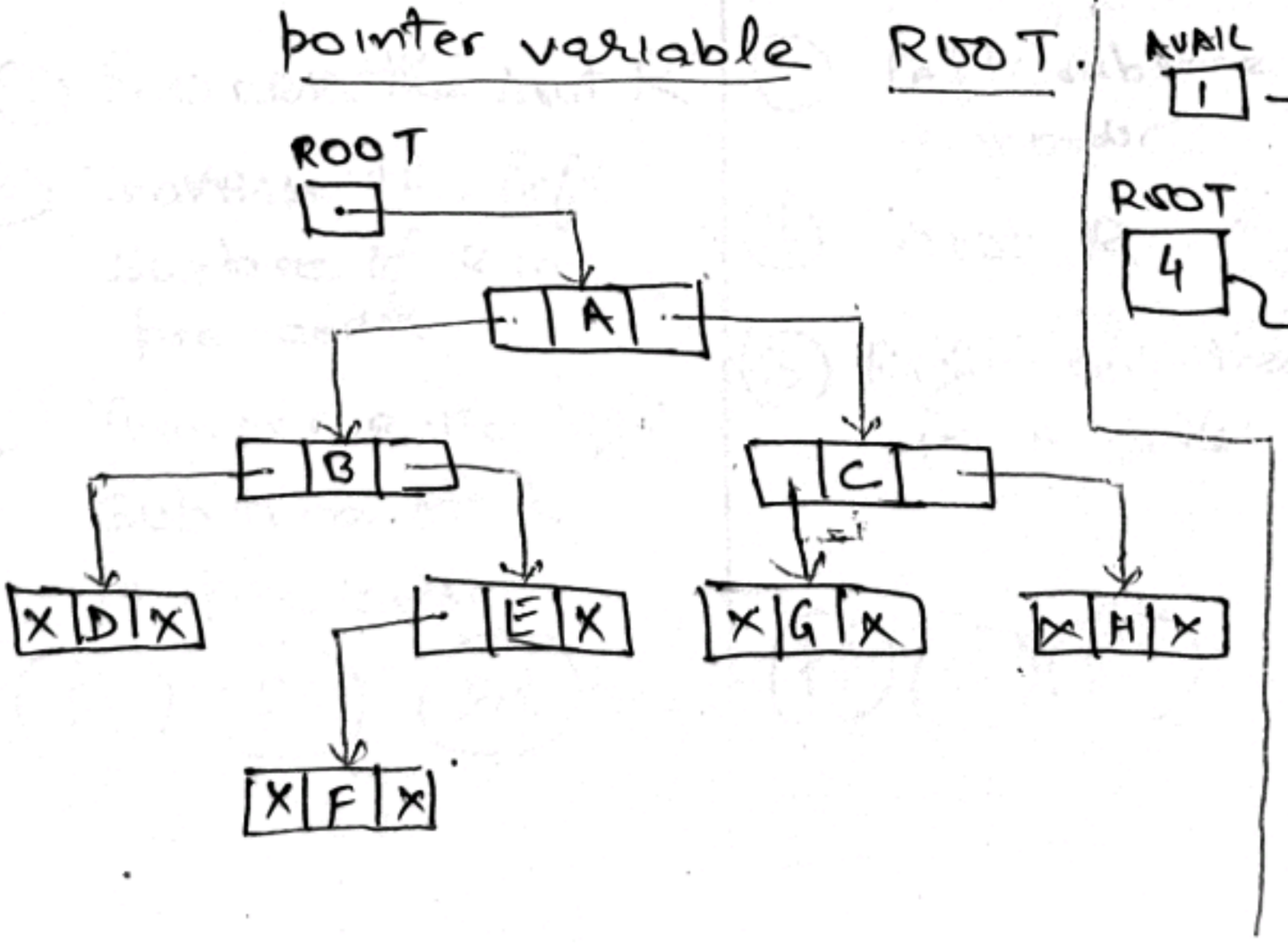
REPRESENTING BINARY TREES IN MEMORY

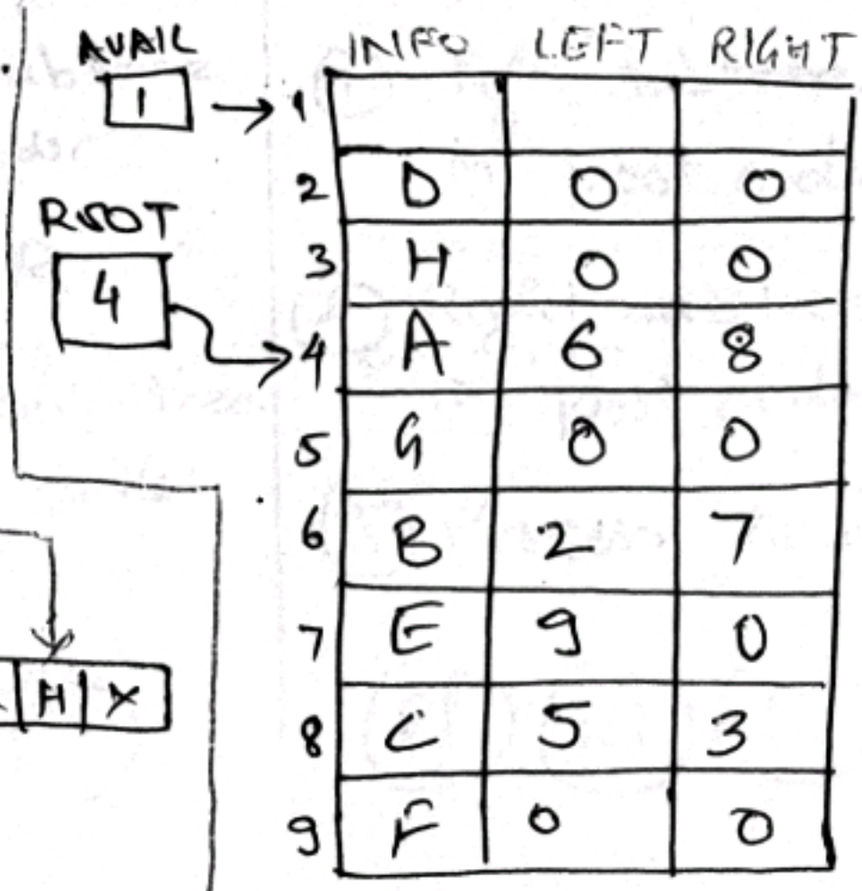
The main requirements:

(i) One should have direct access to the root R of T

(ii) One should have direct access to children of N

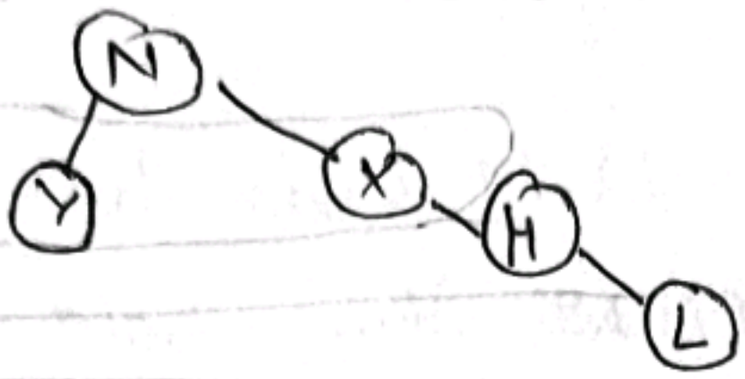
(a) Linked representation of Binary trees: by using three parallel arrays, INFO, LEFT & RIGHT & a pointer variable ROOT.





	INFO	LEFT	RIGHT
1			
2	D	0	0
3	H	0	0
4	A	6	8
5	G	0	0
6	B	2	7
7	E	9	0
8	C	5	3
9	F	0	0

The term descendent and ancestor have their usual meaning



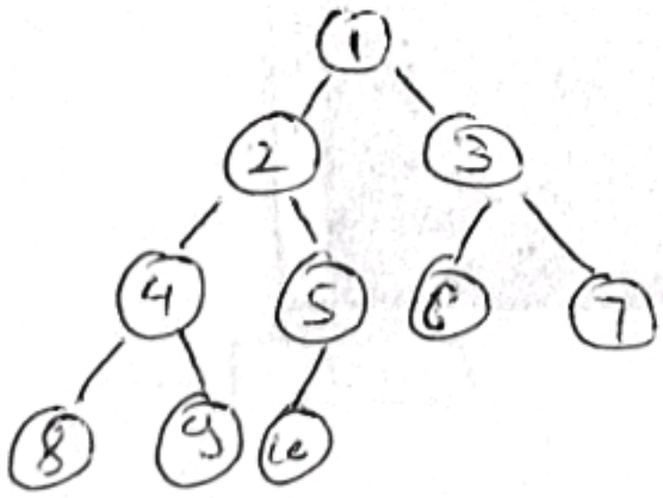
N is ancestor of L &
L is descendent of N (Left/Right)

- The line drawn from node N to successor called edge. (N-X)
- Sequence of coseative edges is called a path. (N-X-H)
- A terminal node is called a leaf. (L)
- Path ending in a leaf is called a branch (N-X-H-L)
- Level number: The root R of a tree T is assigned the level number '0' or (zero), and every node is assigned a level number which is 1 more than the level number of its parent
- Same generation: Nodes with same level number are said to belong to same generation. (X, Y)
- Depth (or height): of tree T is the maximum number of nodes in the branch of T. (4)

COMPLETE BINARY TREE

The tree T is said to be complete if all its levels, except possible the last, have the maximum number of possible nodes; & if all the nodes at the last level appear as far left as possible.

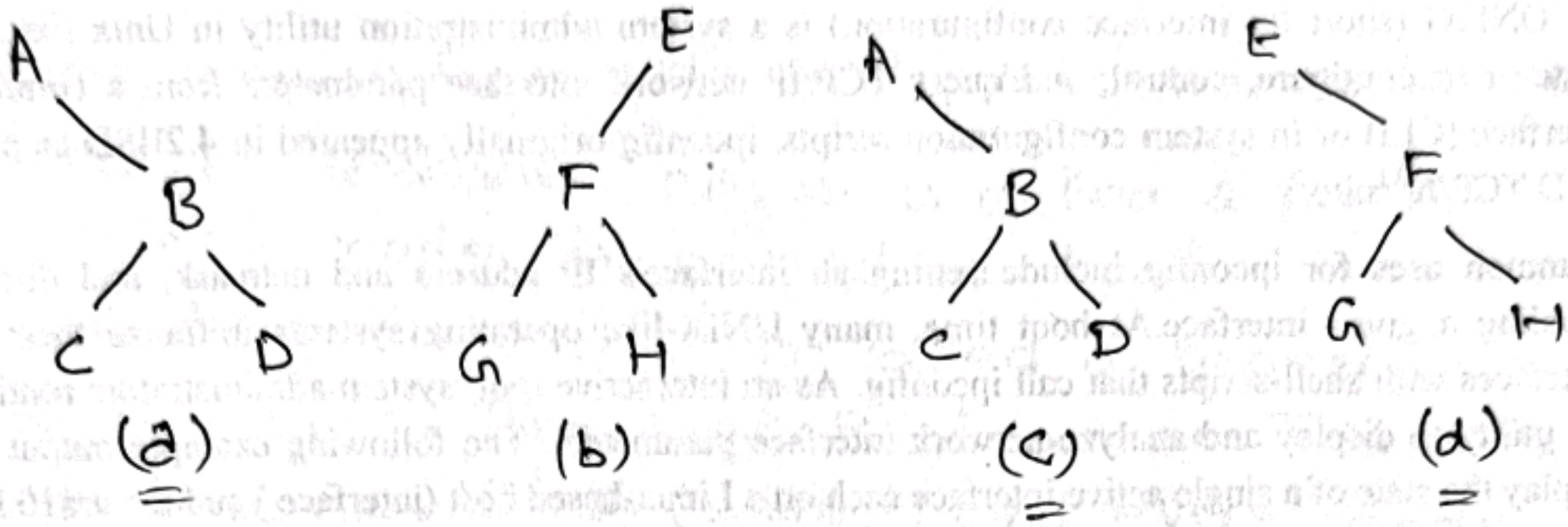
→ Level k of T can have at most 2^k nodes



* The left & right children of node K are:
(2 * K) & (2*K)+1

* Parent of K := $\lceil K/2 \rceil$ (lower bound)

Binary tree T & T' are said to be similar if they have the same structure or in other words if they have the same shape.

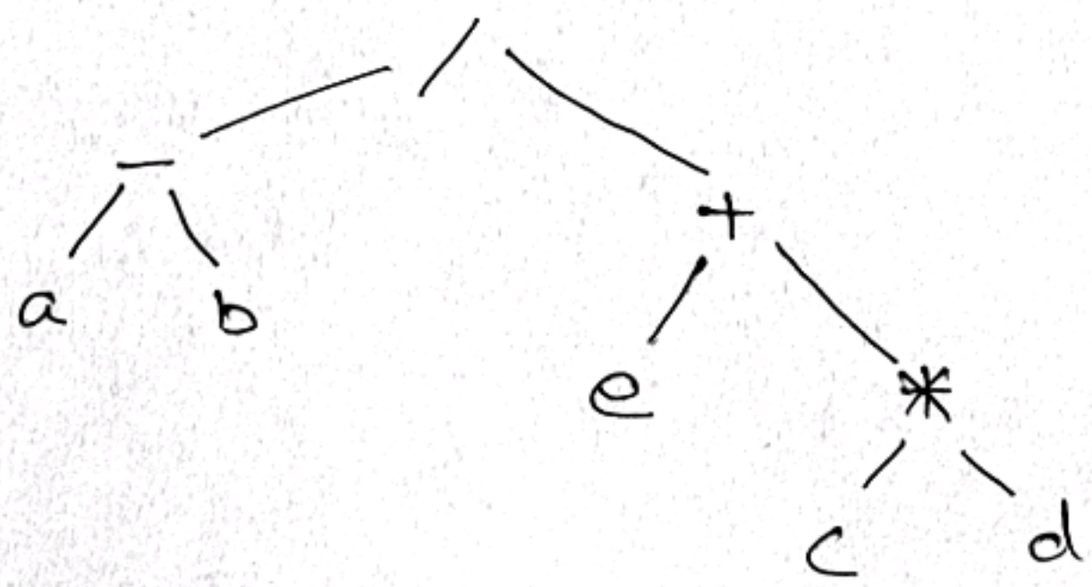


→ Tree (a), (c) & (d) are similar

→ Tree (a) & (c) are copies (Because they have same data)

→ Tree (b) & (d) there is difference between left & right successor

Eg: E = (a-b) / ((c*a) + e)



TERMINOLOGY

Suppose node N is a node in T with the left successor S1 and right successor S2. Then

N is called Parent (or father) of S1 & S2

S1 is left child & S2 is the right child (son)

S1 & S2 are said to be siblings (or brothers)

Except root every node (N) has a unique parent called predecessor of N

"If you don't have enemies, you don't have character" - Paul Newman

TREES

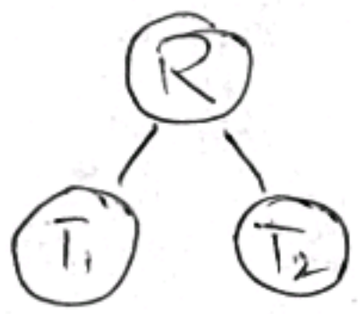
Tree is non-linear data structure, use to represent the data ~~structure~~ containing a hierarchical relationship between elements.
i.e. family tree etc.

BINARY TREE

A binary tree T is defined as finite set of elements, called nodes, such that:

(a) T is empty (called null or empty tree)

(b) T contains a distinguished node R called root of T, and the remaining nodes of T ~~from~~ form an order pair of disjoint binary trees T_1 and T_2



R → Root node
T_1 → Left subtree of R
T_2 → Right subtree of R

If T_1 is nonempty: then its root is called the left successor of R.

~ If T_2 is non-empty: then its root is called the right successor of R.

Any node N in binary tree T has either 0, 1 or 2 successor. The node with no successor are called terminal nodes.

The above definition of the binary tree T is recursive since T is defined in terms of the binary subtrees T_1 & T_2