Data Structure

Tree

Instructors:
Md Nazrul Islam Mondal
Department of Computer Science & Engineering
Rajshahi University of Engineering & Technology Rajshahi-6204

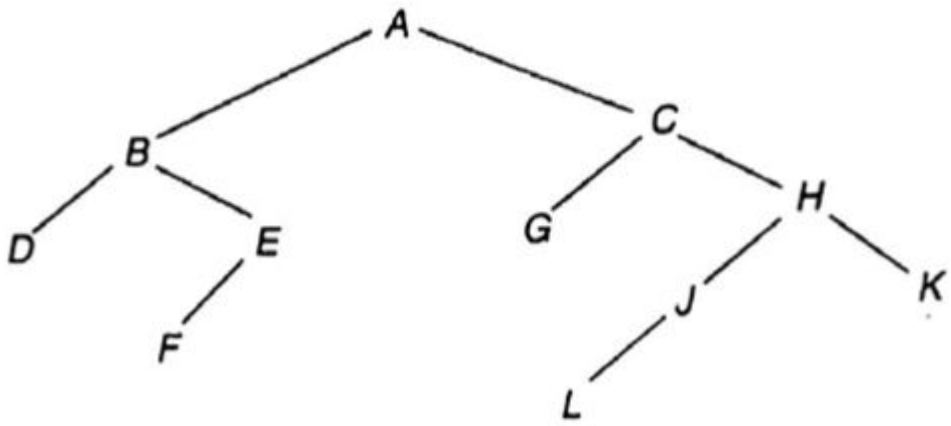
Outline

- **Binary Tree**
- **Representing Binary Trees in Memory**
- **Traversing Binary Trees**
- **Traversal Algorithm using Stacks**
- Header Nodes: Threads
- Binary Search Trees
- Searching and Inserting in Binary Search Trees
- Deleting in Binary Search Tree
- AVL Search Trees
- Insertion in an AVL Search Tree
- Deletion in an AVL Search Tree
- m-way Search Trees
- Searching, Insertion and Deletion in an m-way Search Tree
- B Trees
- Searching, Insertion and Deletion in a B-tree

Binary Tree

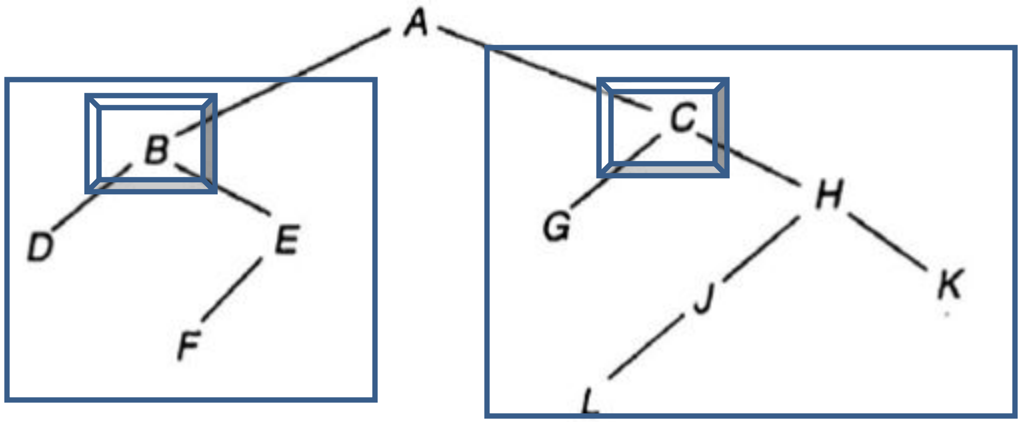
Binary Tree (Definition)

- A binary tree T is defined as **a finite set of elements**, called nodes, such that:
  a) T is empty (called the null tree or empty tree) **or**
  b) T contains a **distinguished node** R, called the **root** of T, and the remaining nodes of T form an ordered pair of **disjoint** binary trees **T_1 and T_2**



Binary Tree (Terminology)

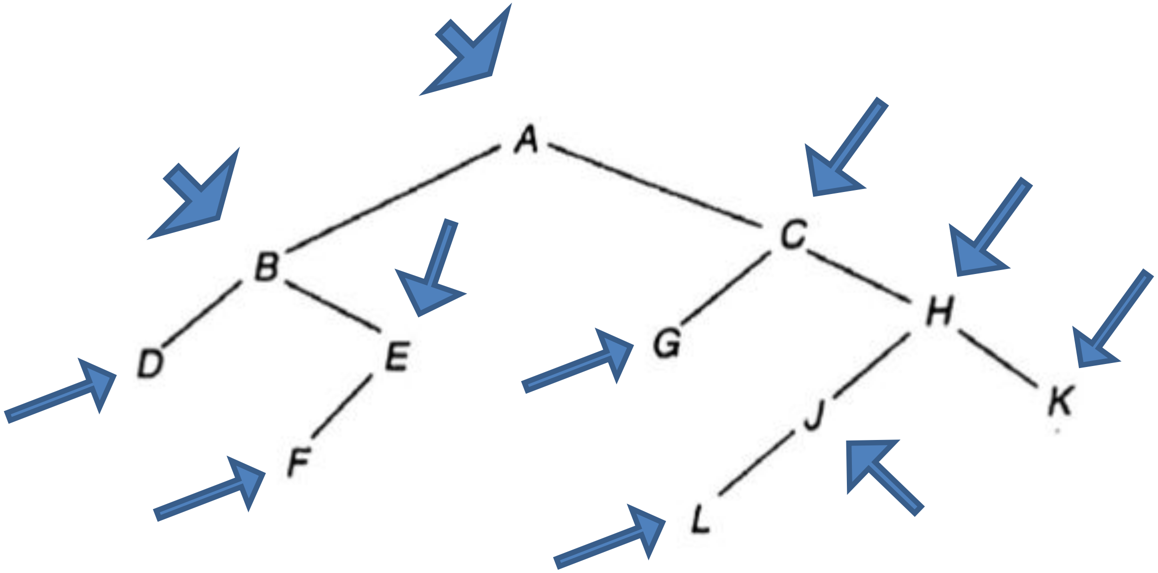
Root



- Left subtree of A
- **Root of left subtree**: B
- **Left Successor of A**: B

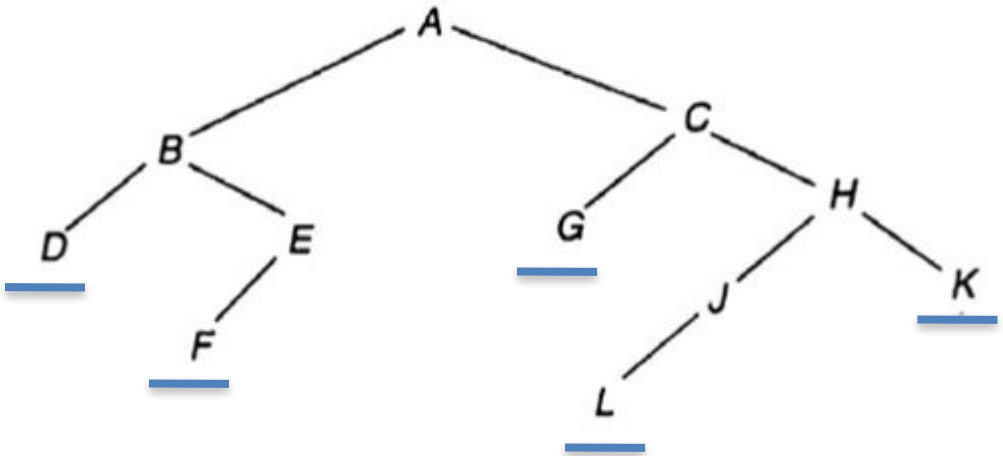
- Right subtree of A
- **Root of Right subtree**: C
- **Right Successor of A**: C

Binary Tree (Terminology)



- A, B, C, H have two successor
- E and J have One successor
- D, F, G, L and K have no successor

Binary Tree (Terminology)



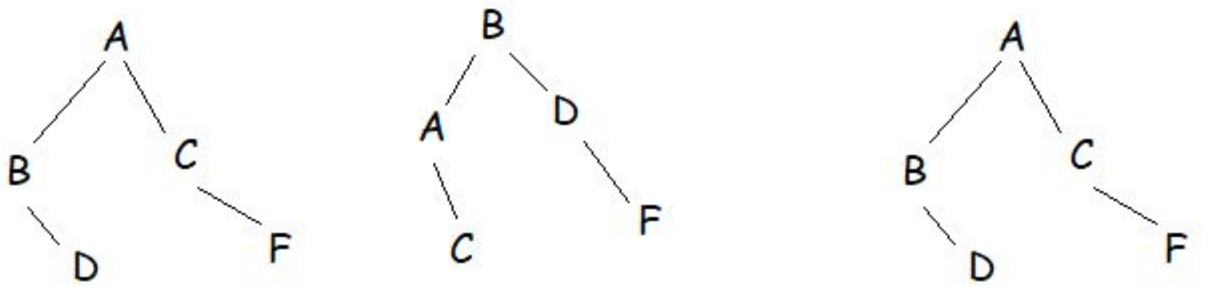
- The nodes with no successor are called **terminal nodes**

Binary Tree (Terminology)

- Binary Tree T and T' are said to be **similar** if they have the same structure.
- The trees are said to be **copies**, if they are **similar** and they have the **same contents** at corresponding nodes.

Binary Tree (Terminology)

- Binary Tree T and T’ are said to be **similar** if they have the same structure.
- The trees are said to be **copies**, if they are **similar** and they have the same **same contents** at corresponding nodes.



Similar

Copy

Binary Tree (Terminology)

$E = (a - b) / ((c * d) + e)$

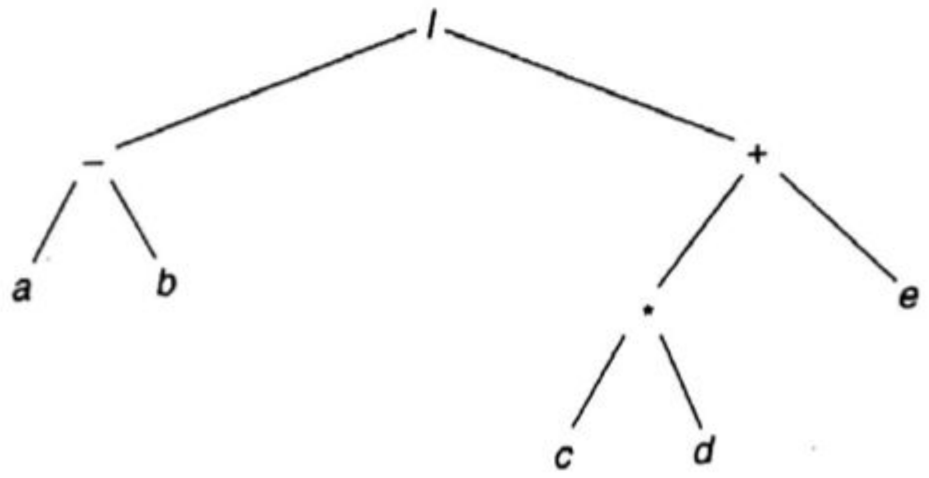
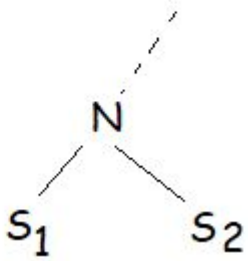


Fig. 7.3 $E = (a - b)/((c * d) + e)$

Binary Tree (Terminology)

- N is not root and any node in a binary tree T.
- Left successor of N: S_1
- Right Successor of N: S_2

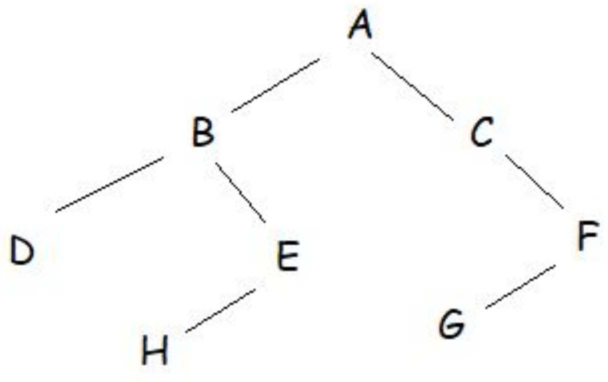


- N is called the **parent** (or father of S_1 and S_2
- S_1 Left child (or son) of N
- S_2 Right child (or son) of N
- S_1 and S_2 are said to be siblings (or brothers)

- **Predecessor** of S_1 is N (**parent**)
- **Predecessor** of S_2 is N (**parent**)

Binary Tree (Terminology)

- A node L is called a **descendant** of a node N if there is a **succession** of children from N to L.

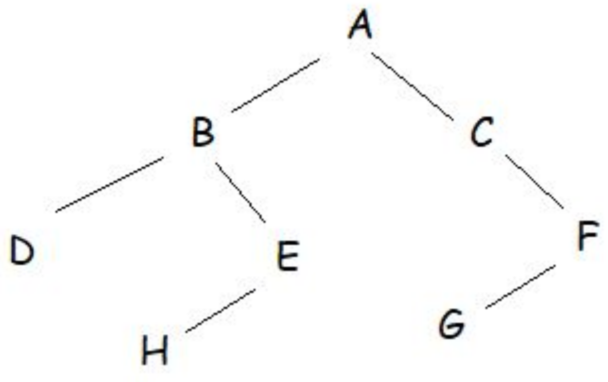


- All nodes are the A's descendant
- B has three descendants i.e. D, E, H
- C has two descendants i.e. F, G
- D has no descendant
- E has one descendant i.e. H
- F has one descendant i.e. G
- **H and G have no descendant**

- D is descendant of A or B (other word)

Binary Tree (Terminology)

- A node N is called a **ancestor** of a node L if there is a **succession** of children from N to L.

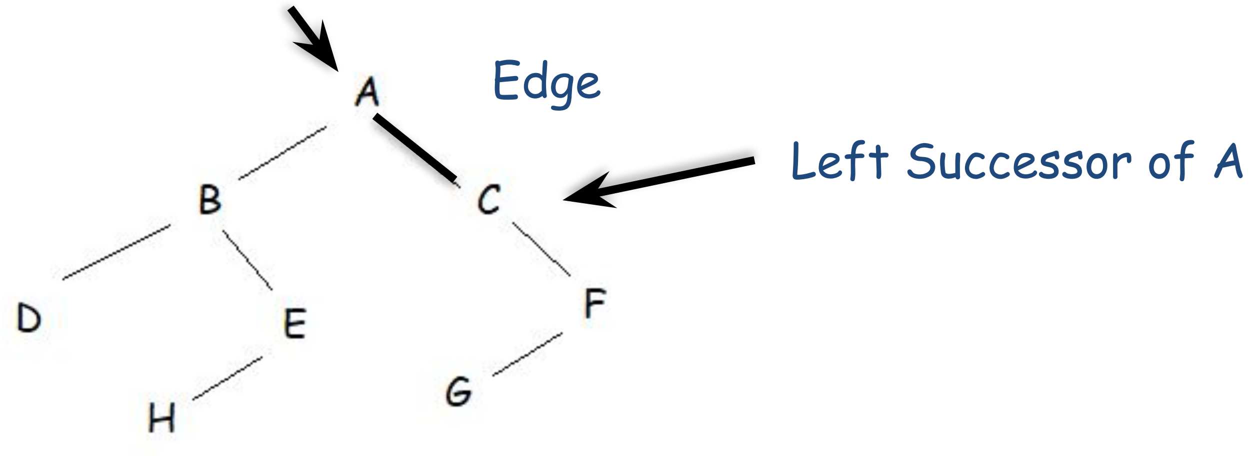


- ***A has no ancestor***
- B has one ancestor i.e. A
- C has one ancestor i.e. A
- D has two ancestors i.e. A, B
- E has two ancestors i.e. A, B
- F has two ancestors i.e. A, C
- H has three ancestors i.e. A, B, E
- G has three ancestors i.e. A, C, F

- B is an ancestor for D,E and B (other word)

Binary Tree (Terminology)

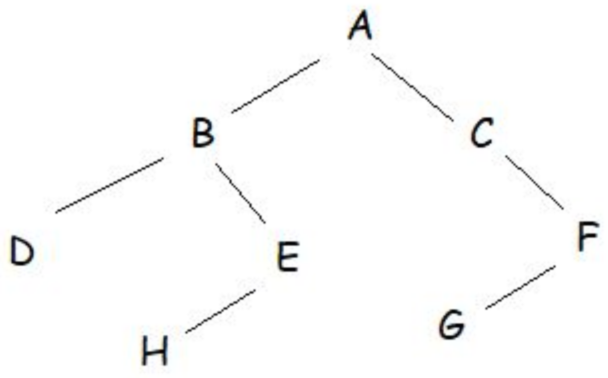
- The line draw from a node N of T to a successor is called an **edge**.


Edge

Left Successor of A

Binary Tree (Terminology)

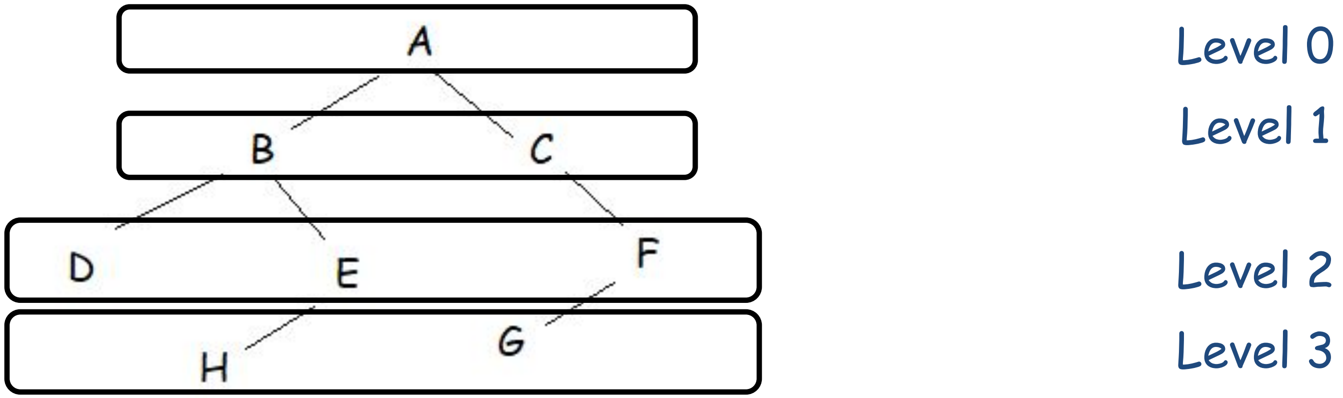
- A sequence of consecutive edges is called a **path**
- A terminal node is called a **leaf**.
- A path ending in a leaf is called a **branch**



- A path from A to E is **A-B-E**
- D, H, and G are **leaf** node
- A **branch** : A – C – F-G

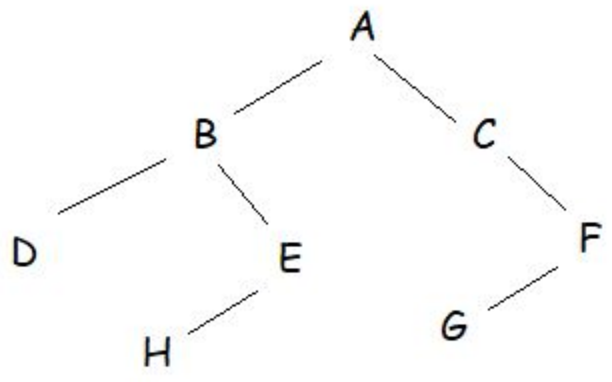
Binary Tree (Terminology)

- Level Number:



Binary Tree (Terminology)

- The **depth** (or **height**) of a tree T is the **maximum number of nodes** in a branch of T.



- Largest Branch is ACFG or ABEH
- Maximum Number of Nodes = 4
- **The depth (or height)** of this tree = 4

Binary Tree (Complete Binary Tree)

- The Tree T is said to be **complete**
  - if all its levels, except possibly the last, have the maximum number of possible nodes, and
  - if all the nodes at the last level appear as far left as possible.

- **Remember**: level r of T can have at most 2^r nodes

Binary Tree (Complete Binary Tree)

- Left child of K node is 2K i.e. 4 is the left child of 2
- Right Child of K node is 2K+1 i.e. 5 is the right child of 2

- The depth d_n of the complete tree T_n with n nodes is given by

$$D_n = \lfloor \log_2 n + 1 \rfloor$$

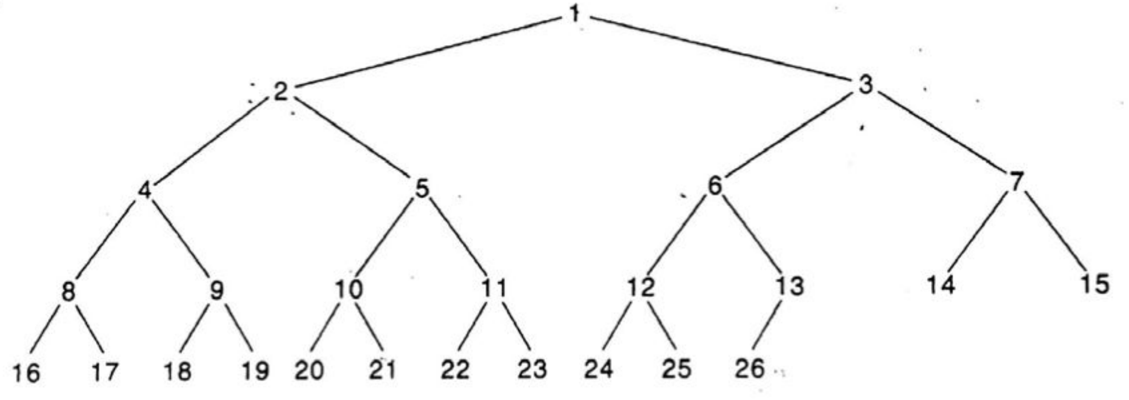
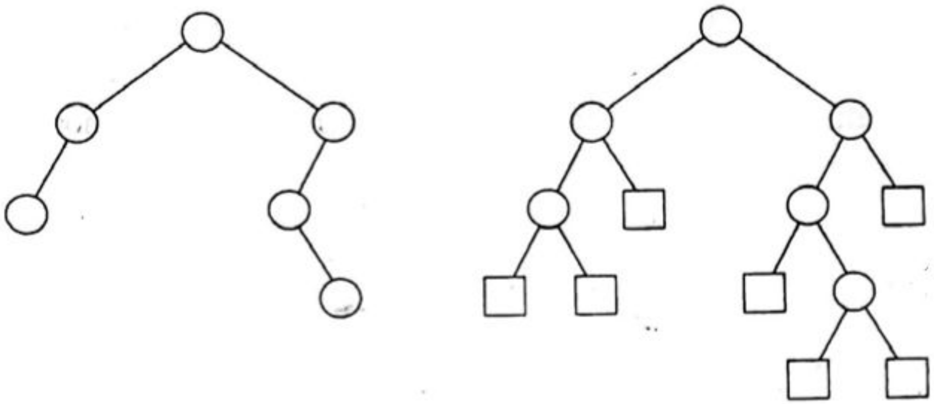


Fig. 7.4 *Complete Tree T_{26}*

Binary Tree (Extended Binary Tree: 2-Trees)

- A binary tree T is said to be a 2-tree or an extended binary tree if **each node N has either 0 or 2 children**.

- The nodes with 2 children are called **internal nodes**.
- The nodes with 0 children are called **external nodes**.



(a) Binary tree *T* (b) Extended 2-tree

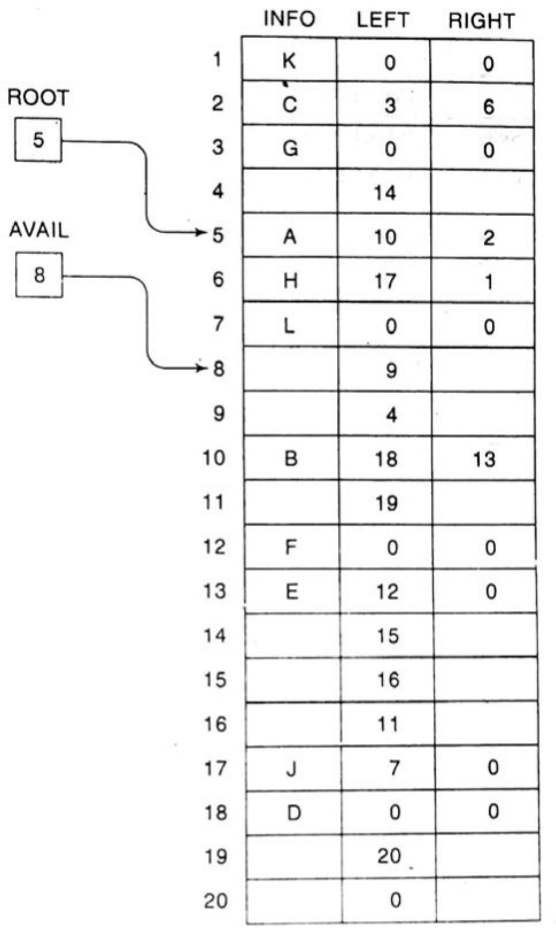
Fig. 7.5 *Converting a Binary Tree T into a 2-tree*

Representing Binary Trees in Memory

Representing Binary Trees in Memory (Linked Representation)

- Uses three parallel arrays: INFO, LEFT and RIGHT
- A pointer variable ROOT as follows – (each node N of T will correspond to a location K such that)
  1) INFO[K] contains the data at the node N
  2) LEFT[K] contains the location of the left child of node N
  3) RIGHT[K] contains the location of the right child of node N

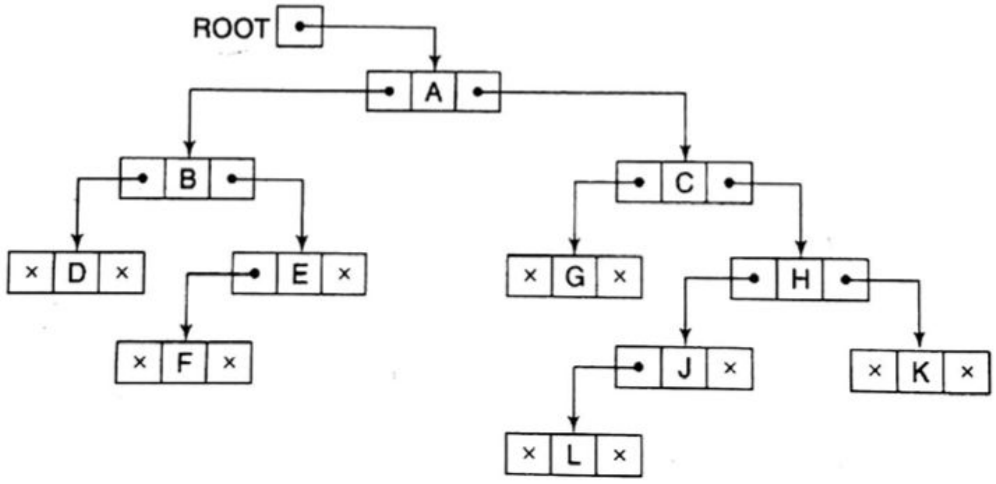
Representing Binary Trees in Memory (Linked Representation)



ROOT 5

AVAIL 8

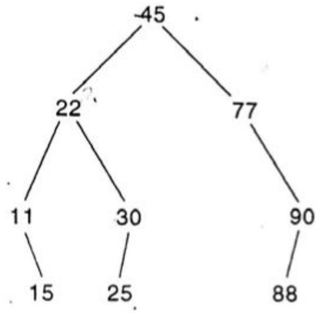
	INFO	LEFT	RIGHT
1	K	0	0
2	C	3	6
3	G	0	0
4		14	
5	A	10	2
6	H	17	1
7	L	0	0
8		9	
9		4	
10	B	18	13
11		19	
12	F	0	0
13	E	12	0
14		15	
15		16	
16		11	
17	J	7	0
18	D	0	0
19		20	
20		0	



Representing Binary Trees in Memory (Sequential Representation)

- Uses 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 is stored in TREE[2K] and its right child is stored in TREE[2K+1].

Representing Binary Trees in Memory (Sequential Representation)



(a)

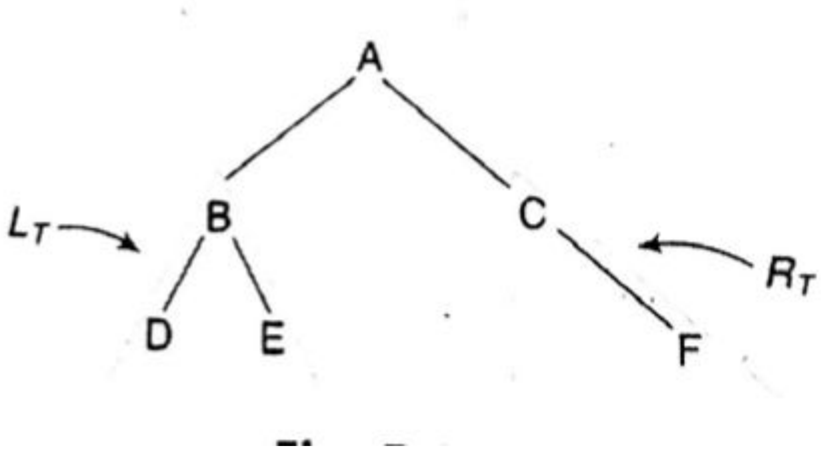
	TREE
1	45
2	22
3	77
4	11
5	30
6	
7	90
8	
9	15
10	25
11	
12	
13	
14	88
15	
16	
⋮	
29	

(b)

Traversing Binary Trees

Traversing Binary Trees

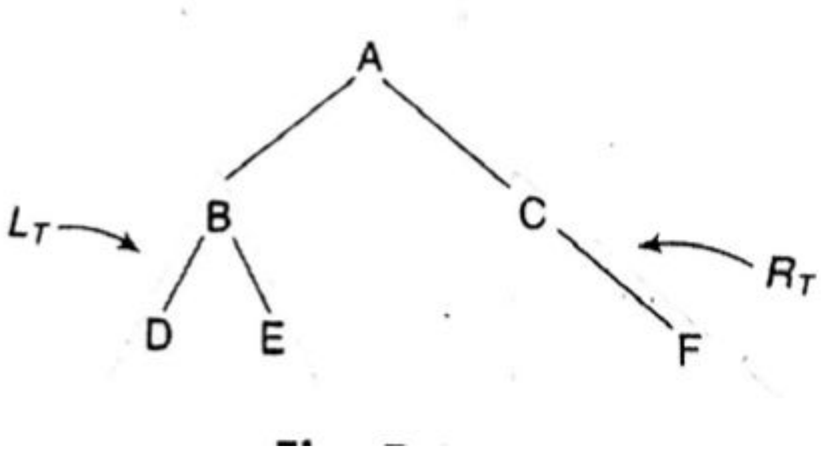
- Preorder –
  1) Process the root R
  2) Traverse the left subtree of R in preorder.
  3) Traverse the right subtree of R in preorder.



- A - (L_T) - (R_T)
- A- (B – (L_T) - (R_T)) – (R_T)
- A – (B – D – E) – (R_T)
- A – (B – D – E) – (C – (L_T) – (R_T))
- A – B – D – E – (C – NULL – F)
- A – B – D – E – C - F

Traversing Binary Trees

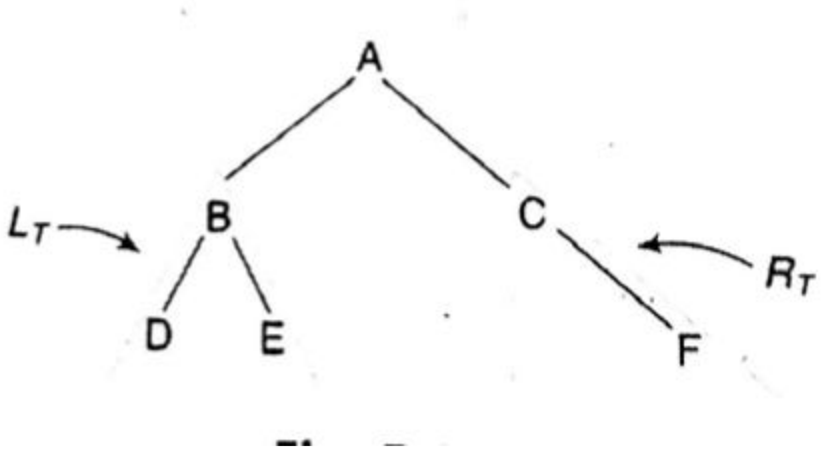
- Inorder –
  1) Traverse the left subtree of R in preorder.
  2) Process the root R
  3) Traverse the right subtree of R in preorder.



- (L_T) - A - (R_T)
- ((L_T) - B – (R_T)) – A- (R_T)
- (D – B – E) – A – (R_T)
- (D – B – E) – A – ((L_T) – C – (R_T))
- D – B – E – A – (NULL – C – F)
- D – B – E – A – C - F

Traversing Binary Trees

- Postorder –
  1) Traverse the left subtree of R in preorder.
  2) Traverse the right subtree of R in preorder.
  3) Process the root R



- (L_T) - (R_T) - A
- ((L_T) – (R_T) - B) - (R_T) – A
- (D – E – B)– (R_T) – A
- (D – E – B) – ((L_T) – (R_T) – C) – A
- D – E – B – (NULL – F – C) – A
- D – E – B – F – C - A

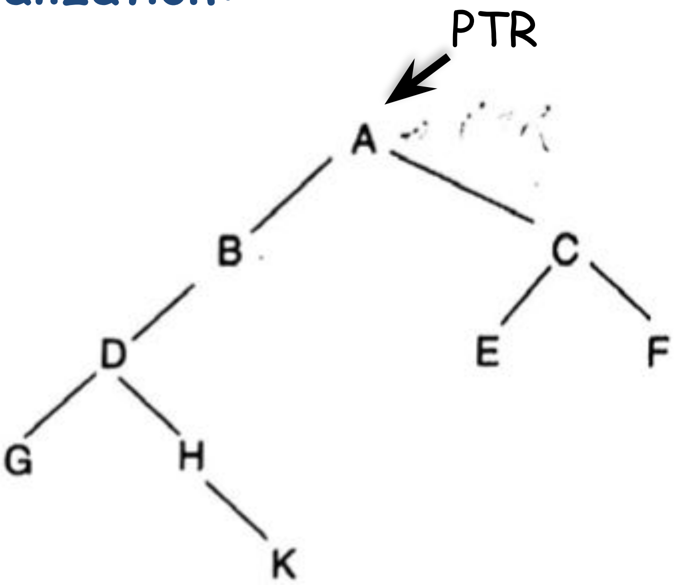
Traversing Binary Trees

Source Code

Traversal Algorithm using Stacks

Traversal Algorithm using Stacks (Preorder Traverse)

Initialization:

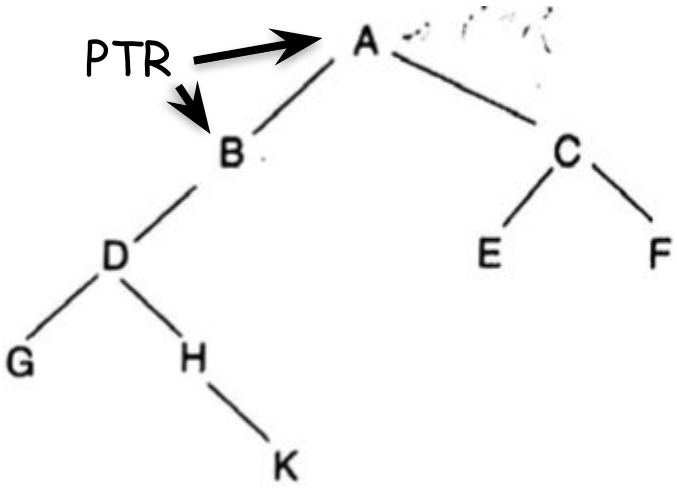


	7	
	6	
	5	
	4	
	3	
	2	
TOP	1	\0

STACK

PTR A, the root of T

Traversal Algorithm using Stacks (Preorder Traverse)



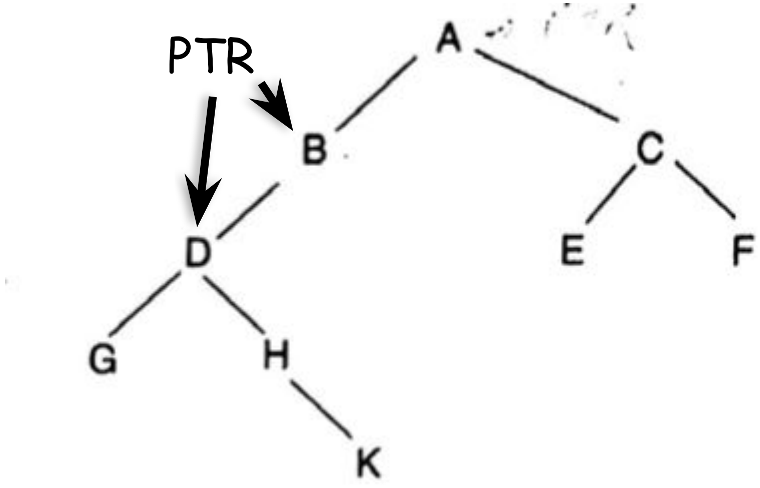
	7	
	6	
	5	
	4	
	3	
TOP	2	C
TOP	1	\0

STACK

A

- Process A
- Push C , Right Child of A
- PTR B, left child of A

Traversal Algorithm using Stacks (Preorder Traverse)



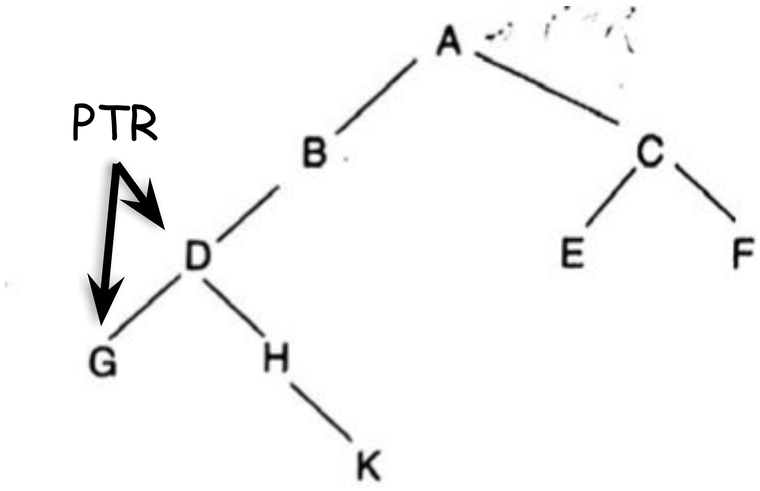
	7	
	6	
	5	
	4	
	3	
TOP	2	C
	1	\0

STACK

A	B

- Process B
- No Push Operation
- PTR □ D, left child of B

Traversal Algorithm using Stacks (Preorder Traverse)



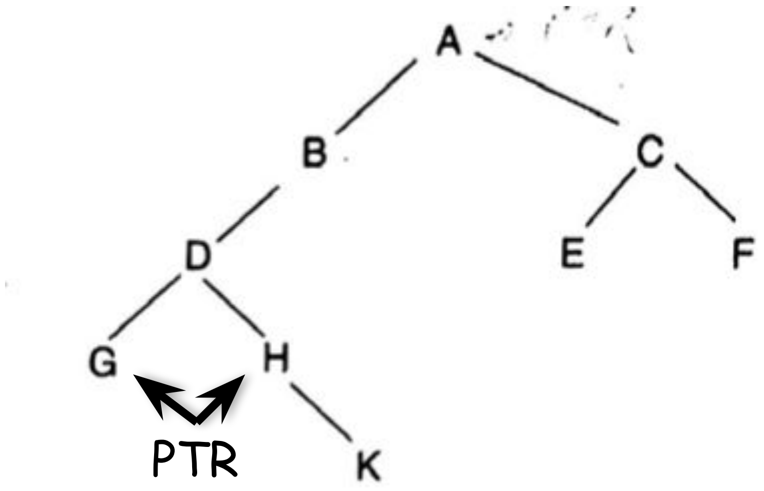
	7	
	6	
	5	
	4	
TOP	3	H
TOP	2	C
	1	\0

STACK

A	B	D

- Process D
- Push H, right child of D
- PTR □ G, left child of D

Traversal Algorithm using Stacks (Preorder Traverse)



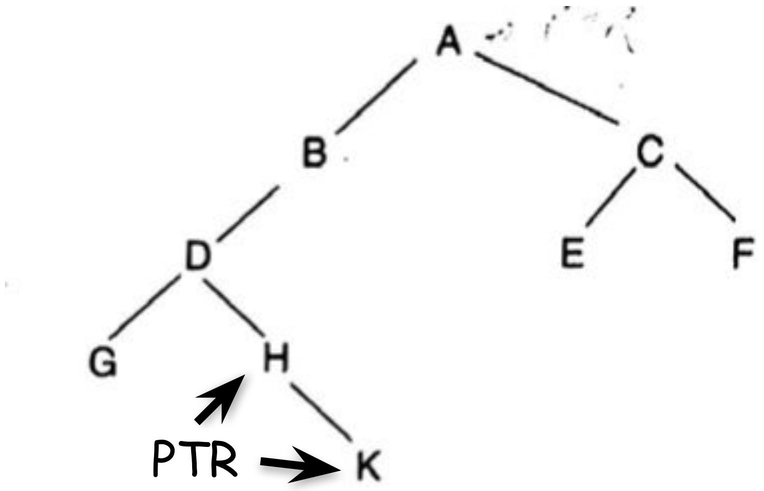
A	B	D	G

	7	
	6	
	5	
	4	
TOP	3	H
TOP	2	C
	1	\0

STACK

- Process G
- No Push, No right child of G
- Since left child of G is NULL, POP **[Backtracking]**
  - PTR H

Traversal Algorithm using Stacks (Preorder Traverse)



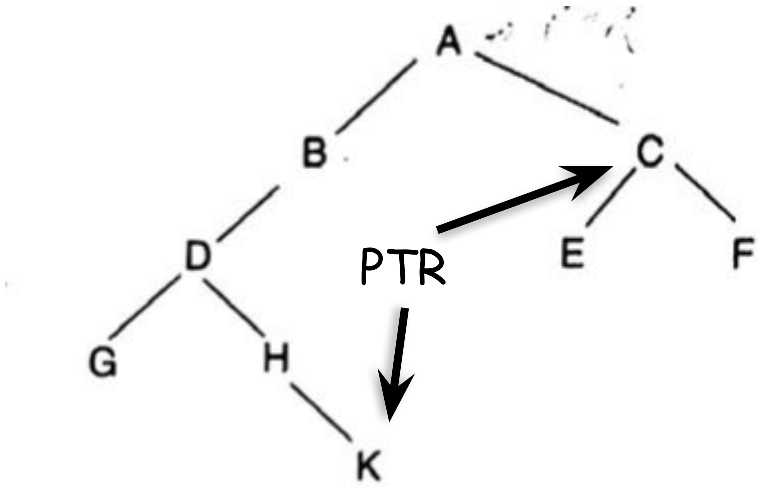
	7	
	6	
	5	
	4	
TOP	3	K
TOP	2	C
	1	\0

STACK

A	B	D	G	H

- Process H
- PUSH K, right child of H
- Since left child of H is NULL, POP**[Backtracking]**
  - PTR K

Traversal Algorithm using Stacks (Preorder Traverse)



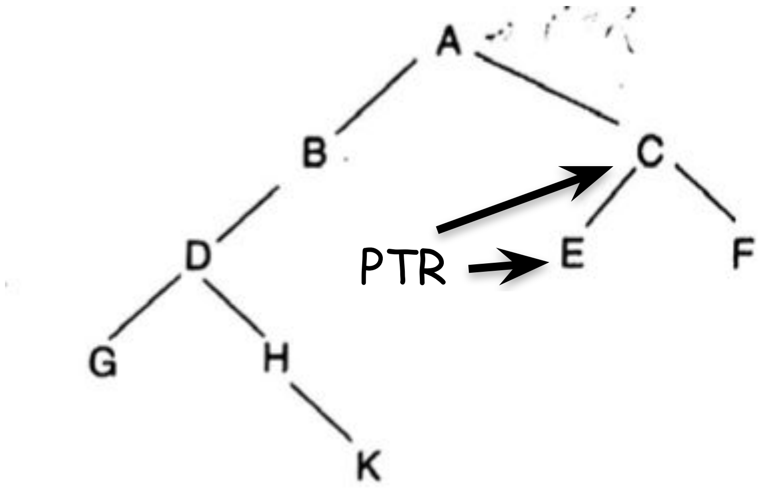
	7	
	6	
	5	
	4	
	3	
TOP	2	C
TOP	1	\0

STACK

A	B	D	G	H	K

- Process K
- No PUSH, No right child of K
- Since left child of K is NULL, POP**[Backtracking]**
    - PTR □ C

Traversal Algorithm using Stacks (Preorder Traverse)



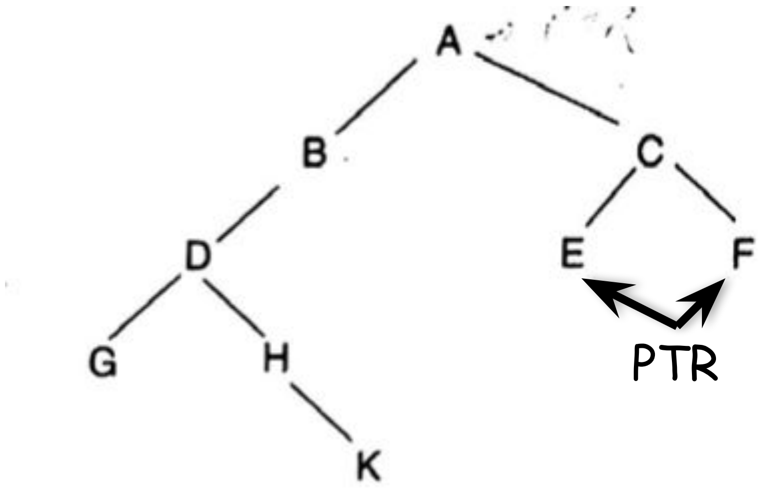
A	B	D	G	H	K	C

	7	
	6	
	5	
	4	
	3	
TOP	2	F
TOP	1	\0

STACK

- Process C
- PUSH F, right child of C
- PTR E, Left child of C

Traversal Algorithm using Stacks (Preorder Traverse)



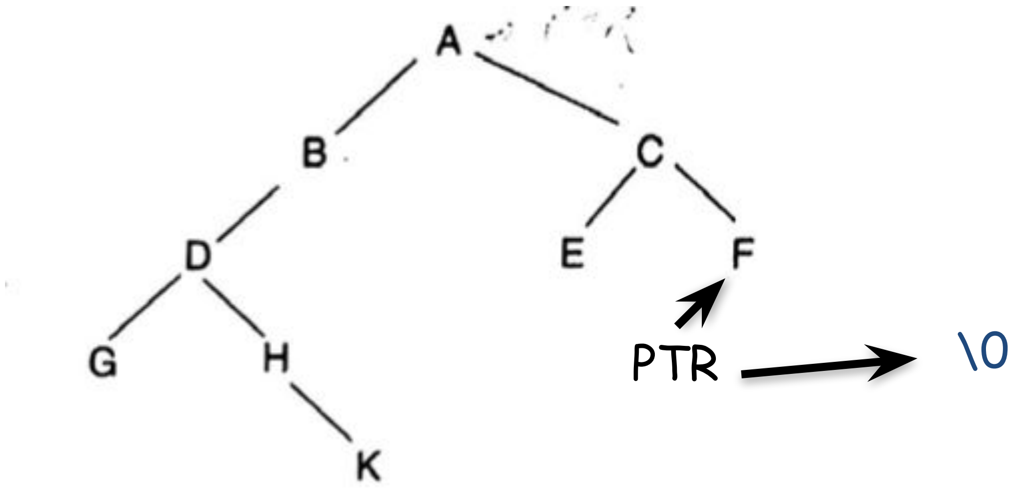
	7	
	6	
	5	
	4	
	3	
TOP	2	F
TOP	1	\0

STACK

A	B	D	G	H	K	C	E

- Process E
- No PUSH, No right child of E
- Left child of E is NULL, POP**[Backtracking]**
    - PTR F

Traversal Algorithm using Stacks (Preorder Traverse)



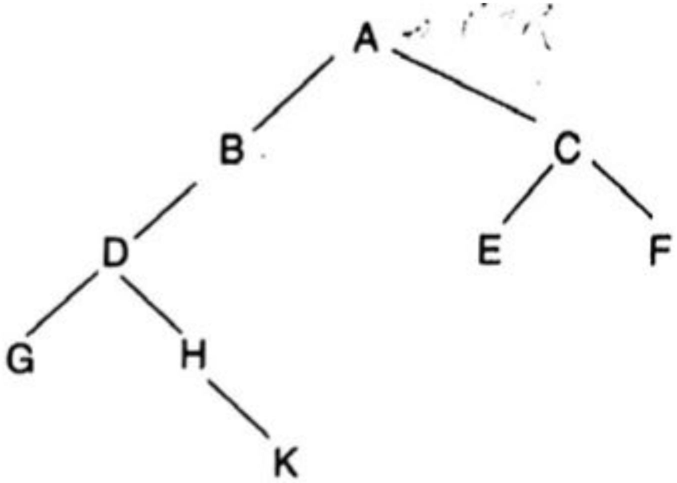
A	B	D	G	H	K	C	E	F

	7	
	6	
	5	
	4	
	3	
	2	
TOP	1	\0

STACK

- Process F
- No PUSH, No right child of F
- Left child of F is NULL, POP**[Backtracking]**
  - PTR← \0

Traversal Algorithm using Stacks (Preorder Traverse)



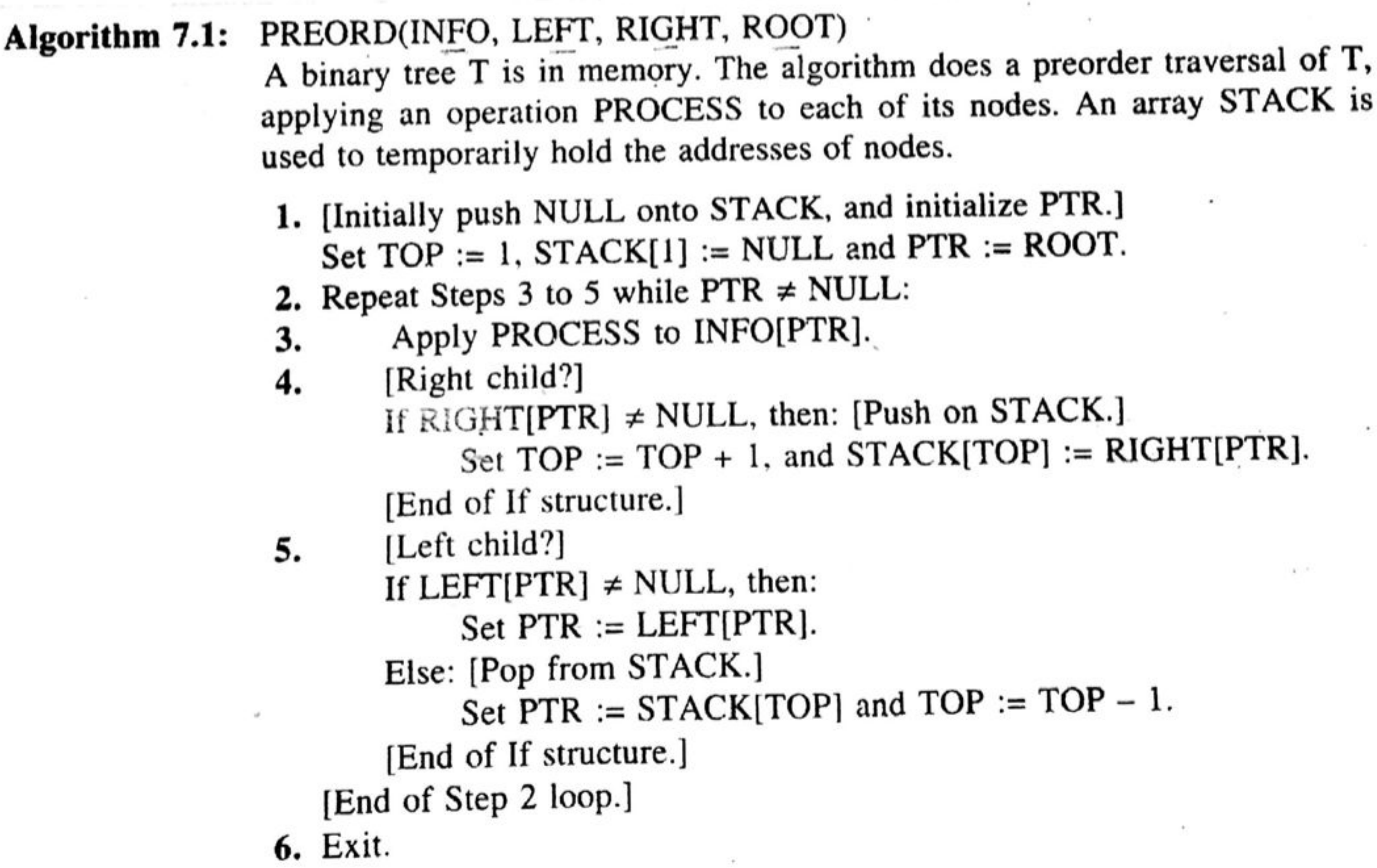
PTR → \0

A	B	D	G	H	K	C	E	F

PREORDER TRAVERSE

- PTR□ \0 **(BREAK CONDITION)**

Traversal Algorithm using Stacks (Preorder Traverse)



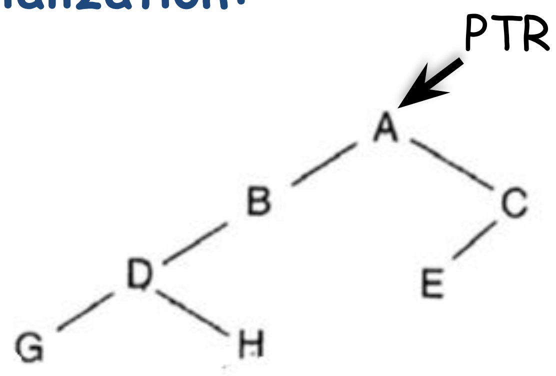
Algorithm 7.1: PREORD(INFO, LEFT, RIGHT, ROOT)

A binary tree T is in memory. The algorithm does a preorder traversal of T, applying an operation PROCESS to each of its nodes. An array STACK is used to temporarily hold the addresses of nodes.

```
1. [Initially push NULL onto STACK, and initialize PTR.]
   Set TOP := 1, STACK[1] := NULL and PTR := ROOT.
2. Repeat Steps 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, and 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] and TOP := TOP - 1.
        [End of If structure.]
   [End of Step 2 loop.]
6. Exit.
```

Traversal Algorithm using Stacks (Inorder Traverse)

Initialization:



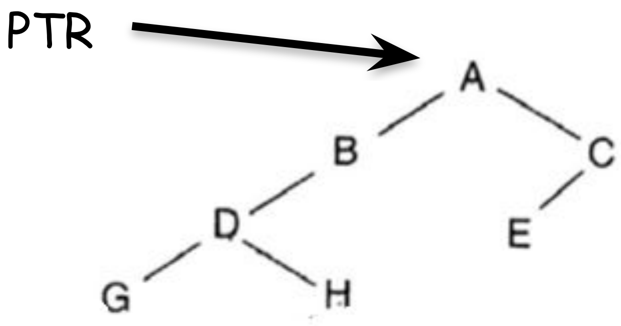
	7	
	6	
	5	
	4	
	3	
	2	
TOP	1	\0

STACK

PTR A, the root of T

Traversal Algorithm using Stacks (Inorder Traverse)

Initialization:



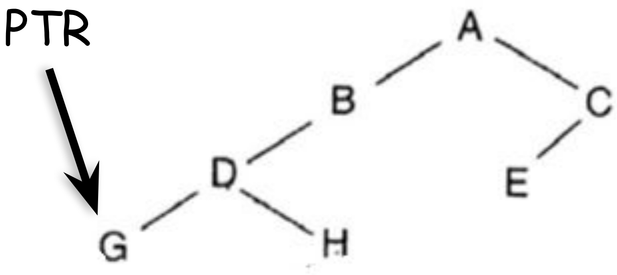
	7	
	6	
TOP	5	G
	4	D
	3	B
	2	A
TOP	1	\0

STACK

```
• Until, PTR ≠ NULL
    • PUSH PTR
    • PTR← LEFT[PTR]
```

Traversal Algorithm using Stacks (Inorder Traverse)

Initialization:



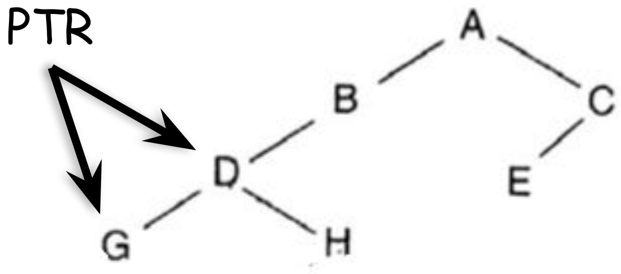
	7	
	6	
TOP	5	G
TOP	4	D
	3	B
	2	A
	1	\0

STACK

- PTR POP() = G **[BACKTRACKING)**

Traversal Algorithm using Stacks (Inorder Traverse)

Initialization:



	7	
	6	
	5	
TOP	4	D
TOP	3	B
	2	A
	1	\0

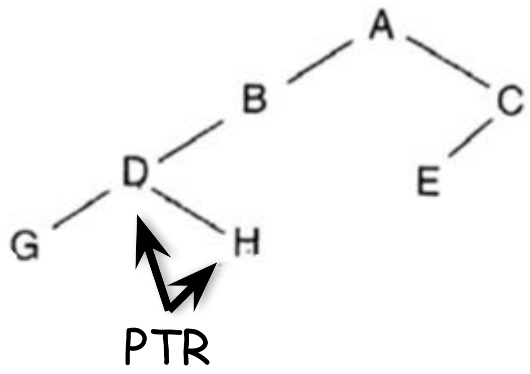
STACK

G

- Process G
- Since right child of G is NULL, No Push
- PTR POP() = D [backtracking]

Traversal Algorithm using Stacks (Inorder Traverse)

Initialization:



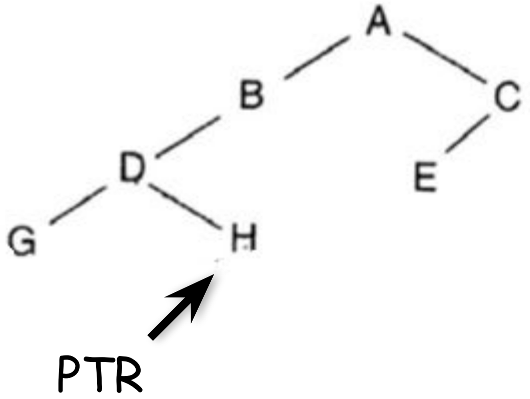
	7	
	6	
	5	
	4	
TOP	3	B
	2	A
	1	\0

STACK

G D

- Process D
- Since right child of D is not NULL,
    - PTR H , right child of D

Traversal Algorithm using Stacks (Inorder Traverse)



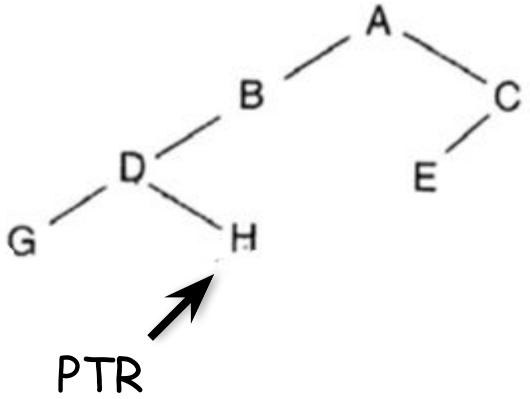
	7	
	6	
	5	
TOP	4	H
TOP	3	B
	2	A
	1	\0

STACK

G D

- Until, PTR ≠ NULL
  - PUSH PTR
  - PTR LEFT[PTR]

Traversal Algorithm using Stacks (Inorder Traverse)



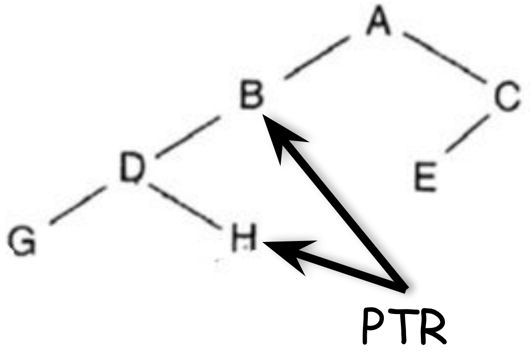
	7	
	6	
	5	
TOP	4	H
TOP	3	B
	2	A
	1	\0

STACK

G D

- PTR□ POP() =H [Backtracking]

Traversal Algorithm using Stacks (Inorder Traverse)



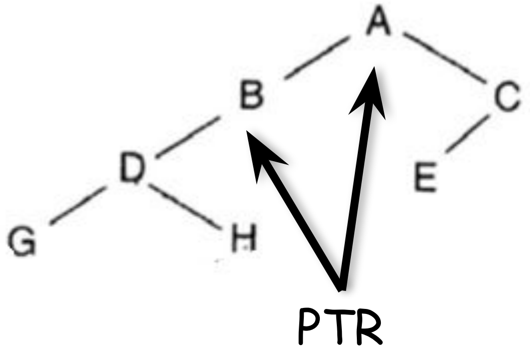
	7	
	6	
	5	
	4	
TOP	3	B
TOP	2	A
	1	\0

STACK

G D H

- Process H
- Since, right child of H is NULL, No PUSH
- PTR POP()= B [Backtracking]

Traversal Algorithm using Stacks (Inorder Traverse)



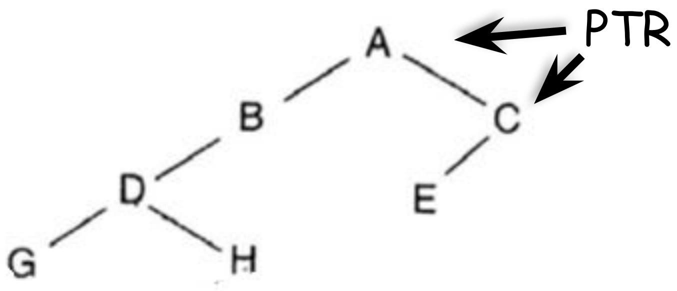
	7	
	6	
	5	
	4	
	3	
TOP	2	A
TOP	1	\0

STACK

G D H B

- Process B
- Since, right child of B is NULL, No PUSH
- PTR POP()=A[Backtracking]

Traversal Algorithm using Stacks (Inorder Traverse)



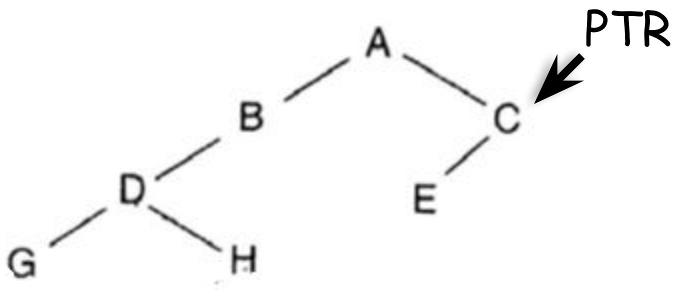
	7	
	6	
	5	
	4	
	3	
	2	
TOP	1	\0

STACK

G	D	H	B	A

- Process A
- Since, right child of A is not NULL,
  - PTR C, right child of A

Traversal Algorithm using Stacks (Inorder Traverse)



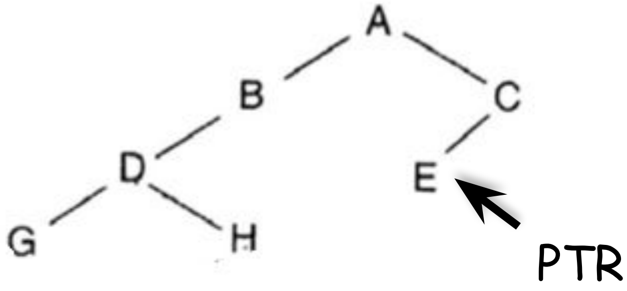
	7	
	6	
	5	
	4	
TOP	3	E
	2	C
TOP	1	\0

STACK

G D H B A

- Until PTR ≠ NULL
  - PUSH PTR
  - PTR□ LEFT[PTR]

Traversal Algorithm using Stacks (Inorder Traverse)



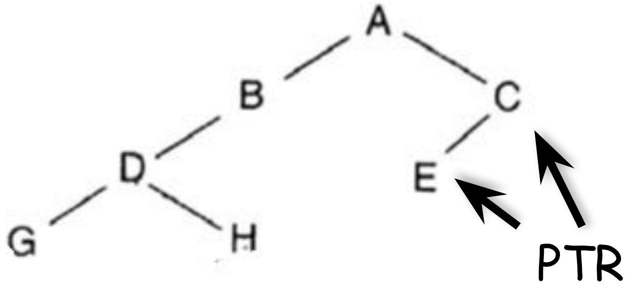
	7	
	6	
	5	
	4	
TOP	3	E
TOP	2	C
	1	\0

STACK

G D H B A

- PTR POP()=E [Backtarcking]

Traversal Algorithm using Stacks (Inorder Traverse)



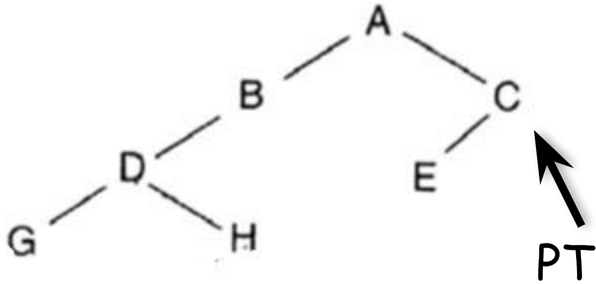
	7	
	6	
	5	
	4	
	3	
TOP	2	C
TOP	1	\0

STACK

G	D	H	B	A	E

- Process E
- Since, right child of E is NULL, No PUSH
- PTR□POP()=C [Backtracking]

Traversal Algorithm using Stacks (Inorder Traverse)



PTR□\0

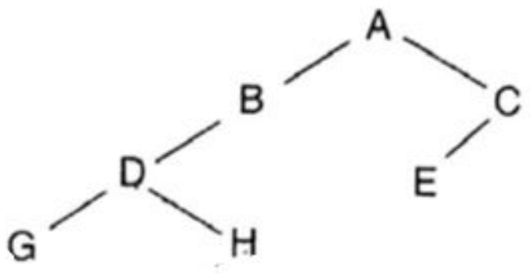
	7	
	6	
	5	
	4	
	3	
	2	
TOP	1	\0

STACK

G	D	H	B	A	E	C

- Process C
- Since, right child of C is NULL, No PUSH
- PTR□POP()=\0 [Backtracking, No more node]

Traversal Algorithm using Stacks (Inorder Traverse)



PTR□\0

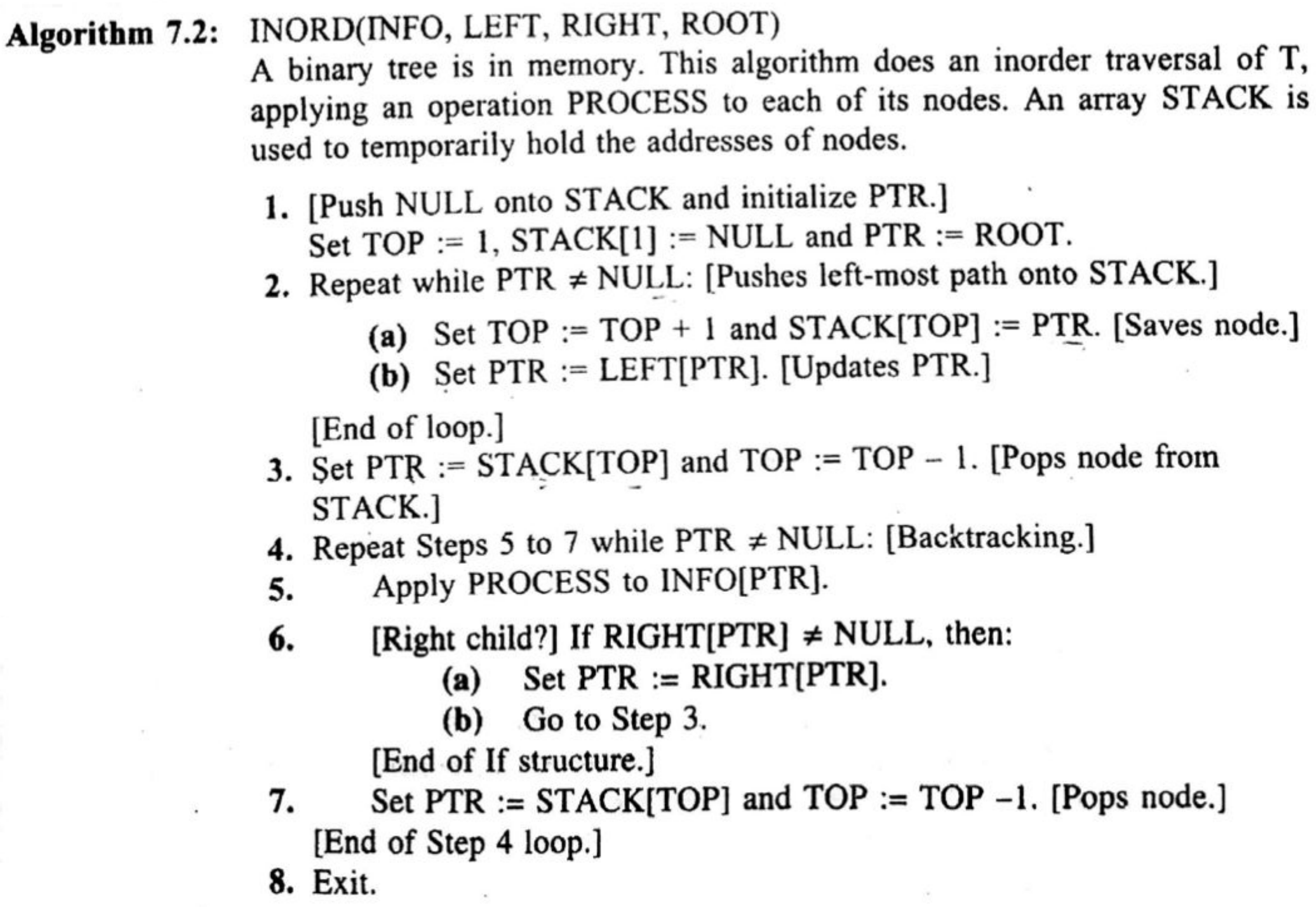
7	
6	
5	
4	
3	
2	
1	

STACK

G	D	H	B	A	E	C

- PTR□\0 (Terminate Algorithm)

Traversal Algorithm using Stacks (Inorder Traverse)

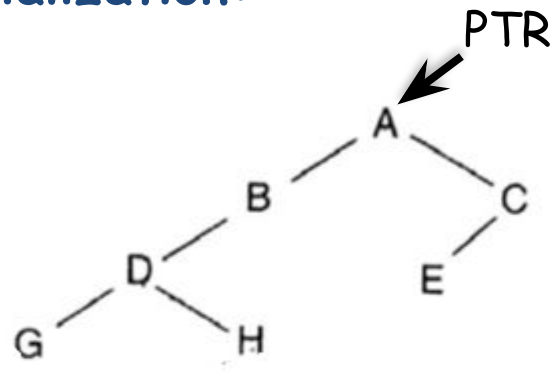


Algorithm 7.2: INORD(INFO, LEFT, RIGHT, ROOT)
A binary tree is in memory. This algorithm does an inorder traversal of T, applying an operation PROCESS to each of its nodes. An array STACK is used to temporarily hold the addresses of nodes.

```
1. [Push NULL onto STACK and initialize PTR.]
   Set TOP := 1, STACK[1] := NULL and PTR := ROOT.
2. Repeat while PTR ≠ NULL: [Pushes left-most path onto STACK.]
       (a) Set TOP := TOP + 1 and STACK[TOP] := PTR. [Saves node.]
       (b) Set PTR := LEFT[PTR]. [Updates PTR.]
   [End of loop.]
3. Set PTR := STACK[TOP] and TOP := TOP – 1. [Pops node from
   STACK.]
4. Repeat Steps 5 to 7 while PTR ≠ NULL: [Backtracking.]
5.     Apply PROCESS to INFO[PTR].
6.     [Right child?] If RIGHT[PTR] ≠ NULL, then:
           (a) Set PTR := RIGHT[PTR].
           (b) Go to Step 3.
       [End of If structure.]
7.     Set PTR := STACK[TOP] and TOP := TOP –1. [Pops node.]
   [End of Step 4 loop.]
8. Exit.
```

Traversal Algorithm using Stacks (Postorder Traverse)

Initialization:

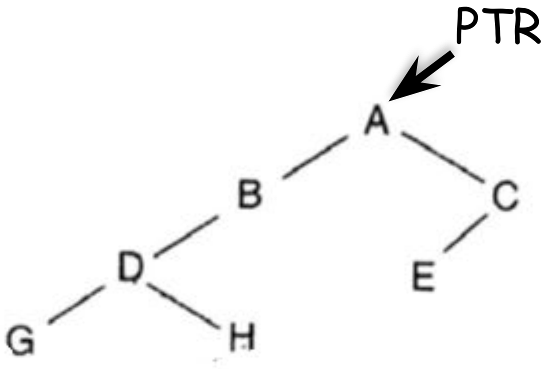


	7	
	6	
	5	
	4	
	3	
	2	
TOP	1	\0

STACK

PTR A, the root of T

Traversal Algorithm using Stacks (Postorder Traverse)



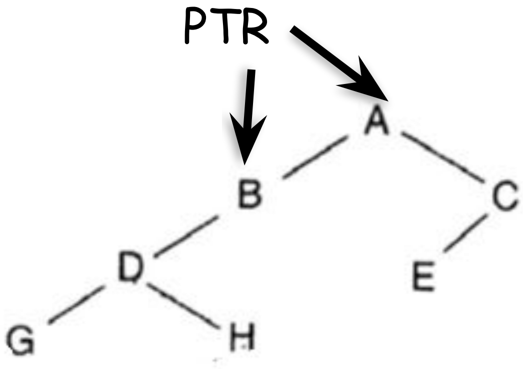
	7	
	6	
	5	
	4	
	3	
	2	
TOP	1	\0

STACK

Basic Step 1:

```
• Until PTR≠NULL
    • PUSH PTR
    • IF RIGHT[PTR]≠NULL
        • PUSH -RIGHT[PTR]
    • PTR□LEFT[PTR]
```

Traversal Algorithm using Stacks (Postorder Traverse)

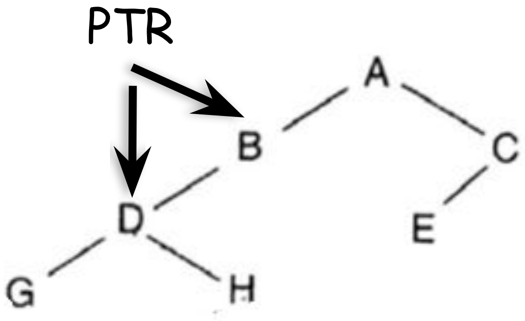


	7	
	6	
	5	
	4	
TOP	3	-C
	2	A
TOP	1	\0

STACK

- **Basic Step 1:**
  - PUSH A
  - PUSH -RIGHT[PTR]= -C
  - PTR□LEFT[PTR]
  - PTR ≠ NULL, Step 1 continue

Traversal Algorithm using Stacks (Postorder Traverse)

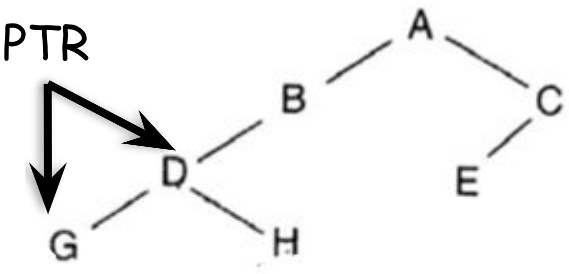


	7	
	6	
	5	
TOP	4	B
TOP	3	-C
	2	A
	1	\0

STACK

- **Basic Step 1:**
  - PUSH B
  - Since Right child of B is NULL, no PUSH
  - PTR←LEFT[PTR]
  - PTR ≠NULL, Step 1 continue

Traversal Algorithm using Stacks (Postorder Traverse)

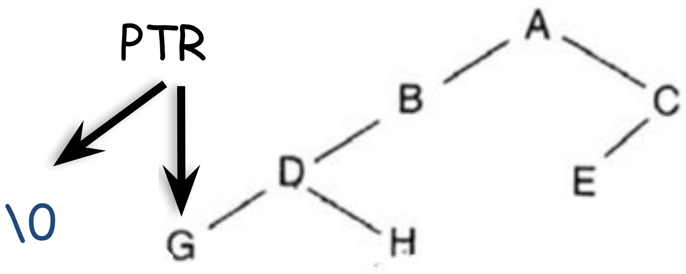


	7	
TOP	6	-H
	5	D
TOP	4	B
	3	-C
	2	A
	1	\0

STACK

- **Basic Step 1:**
  - PUSH D
  - PUSH -RIGHT[PTR]=-H
  - PTR□LEFT[PTR]
  - PTR ≠NULL, Step 1 continue

Traversal Algorithm using Stacks (Postorder Traverse)

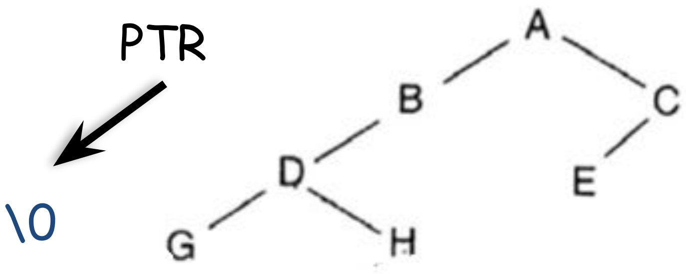


TOP	7	G
TOP	6	-H
	5	D
	4	B
	3	-C
	2	A
	1	\0

STACK

- **Basic Step 1:**
  - PUSH G
  - Since right child of G is NULL, No PUSH
  - PTR□LEFT[PTR]
  - PTR =NULL, Break

Traversal Algorithm using Stacks (Postorder Traverse)

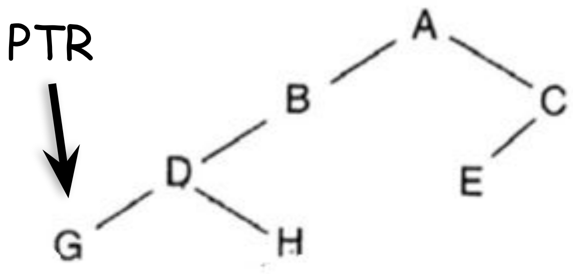


TOP	7	G
	6	-H
	5	D
	4	B
	3	-C
	2	A
	1	\0

STACK

- **Basic Step 2:**
  - PTR POP()

Traversal Algorithm using Stacks (Postorder Traverse)

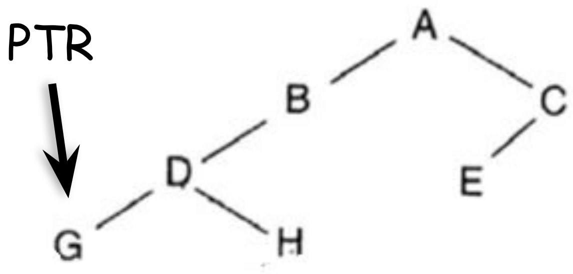


	7	
TOP	6	-H
	5	D
	4	B
	3	-C
	2	A
	1	\0

STACK

- **Basic Step 2:**
  - PTR POP()=G

Traversal Algorithm using Stacks (Postorder Traverse)



	7	
TOP	6	-H
	5	D
	4	B
	3	-C
	2	A
	1	\0

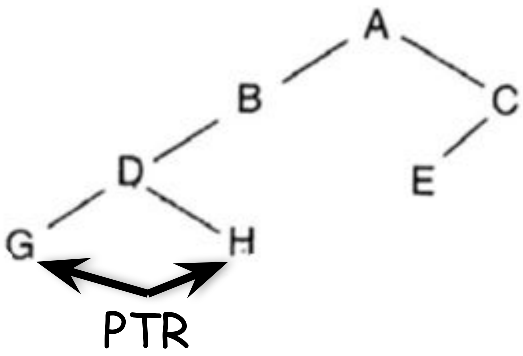
STACK

- **Basic Step 3:**
  - Until PTR>0
    a) Process PTR
    b) PTR□POP()

Traversal Algorithm using Stacks (Postorder Traverse)

	7	
TOP	6	-H
TOP	5	D
	4	B
	3	-C
	2	A
	1	\0

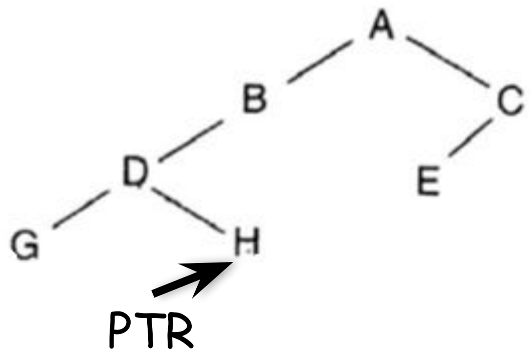
STACK



G

- **Basic Step 3:**
  - Process G
  - PTRPOP()=-H
  - Since PTR<0, Break Loop

Traversal Algorithm using Stacks (Postorder Traverse)



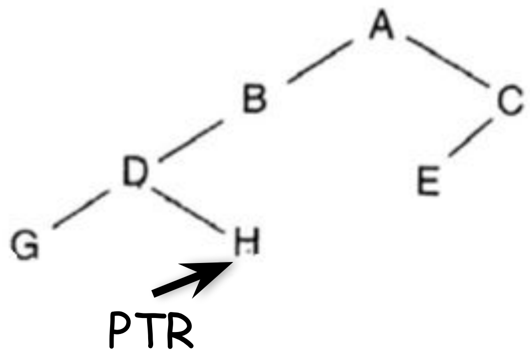
G

	7	
	6	
TOP	5	D
	4	B
	3	-C
	2	A
	1	\0

STACK

- **Basic Step 4:**
  - If PTR<0
    - PTR -PTR
    - Apply Basic Step 1

Traversal Algorithm using Stacks (Postorder Traverse)



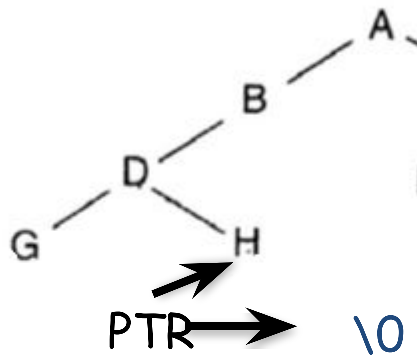
G

	7	
	6	
TOP	5	D
	4	B
	3	-C
	2	A
	1	\0

STACK

- **Basic Step 4:**
  - PTR← - (-H) = H

Traversal Algorithm using Stacks (Postorder Traverse)



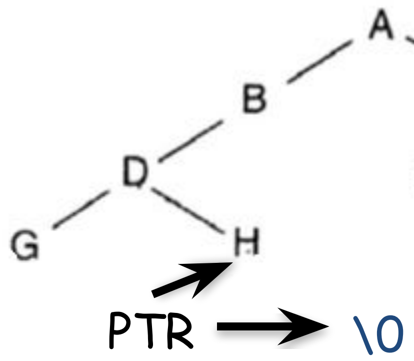
	7	
TOP	6	H
TOP	5	D
	4	B
	3	-C
	2	A
	1	\0

STACK

G

- **Basic Step 1:**
  - PUSH H
  - Since right child of H is NULL, No PUSH
  - PTR LEFT[PTR]=NULL (Break)

Traversal Algorithm using Stacks (Postorder Traverse)



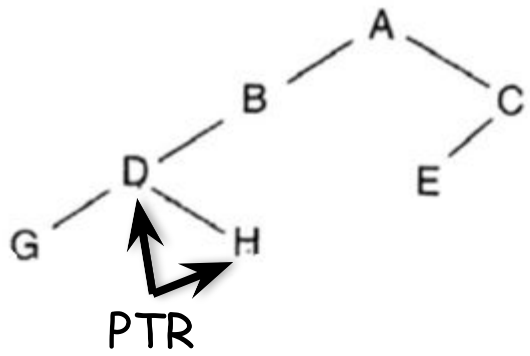
	7	
TOP	6	H
TOP	5	D
	4	B
	3	-C
	2	A
	1	\0

STACK

G

- **Basic Step 2:**
  - PTR POP()=H

Traversal Algorithm using Stacks (Postorder Traverse)



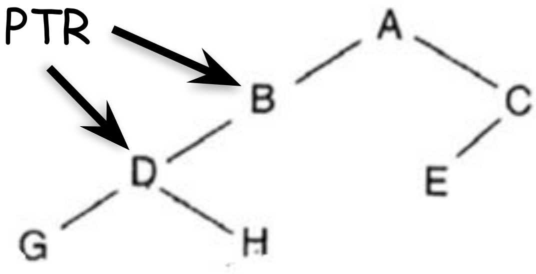
	7	
	6	
TOP	5	D
TOP	4	B
	3	-C
	2	A
	1	\0

STACK

G H

- **Basic Step 3:**
  - Process H
  - PTR←POP()=D
  - PTR > 0, continue

Traversal Algorithm using Stacks (Postorder Traverse)



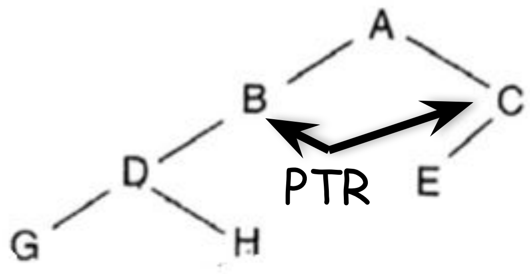
	7	
	6	
	5	
TOP	4	B
TOP	3	-C
	2	A
	1	\0

STACK

G H D

- **Basic Step 3:**
  - Process D
  - PTR□POP()=B
  - PTR > 0, continue

Traversal Algorithm using Stacks (Postorder Traverse)



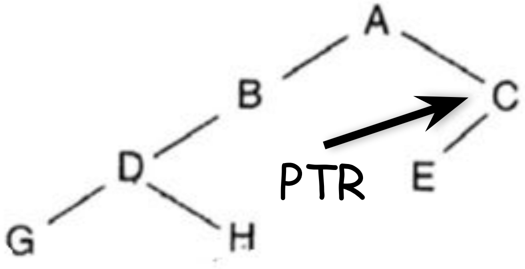
	7	
	6	
	5	
	4	
TOP	3	-C
TOP	2	A
	1	\0

STACK

G H D B

- **Basic Step 3:**
  - Process B
  - PTR□POP()=-C
  - PTR < 0, **BREAK**

Traversal Algorithm using Stacks (Postorder Traverse)



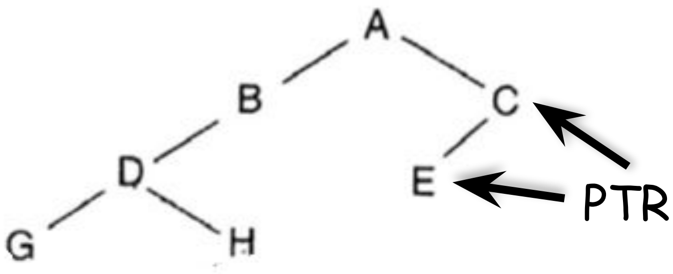
	7	
	6	
	5	
	4	
	3	
TOP	2	A
	1	\0

STACK

G H D B

- **Basic Step 4:**
  - PTR← - (-C) = C

Traversal Algorithm using Stacks (Postorder Traverse)



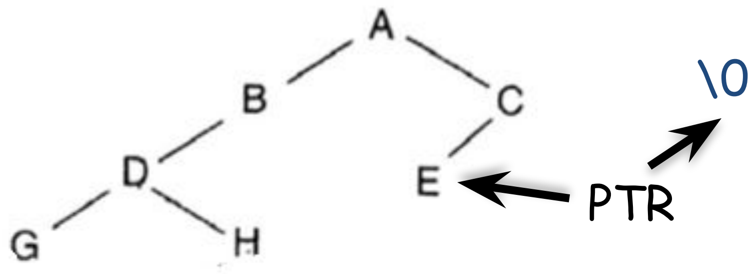
	7	
	6	
	5	
	4	
TOP	3	C
TOP	2	A
	1	\0

STACK

G H D B

- **Basic Step 1:**
  - PUSH C
  - Since right child of C is NULL, No PUSH
  - PTR□ LEFT[PTR]=E
  - PTR ≠NULL, continue

Traversal Algorithm using Stacks (Postorder Traverse)



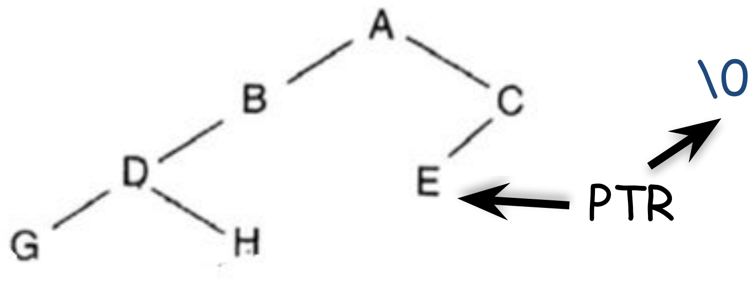
	7	
	6	
	5	
TOP	4	E
TOP	3	C
	2	A
	1	\0

STACK

G H D B

- **Basic Step 1:**
  - PUSH E
  - Since right child of H is NULL, No PUSH
  - PTR□ LEFT[PTR]=NULL
  - PTR =NULL, BREAK

Traversal Algorithm using Stacks (Postorder Traverse)



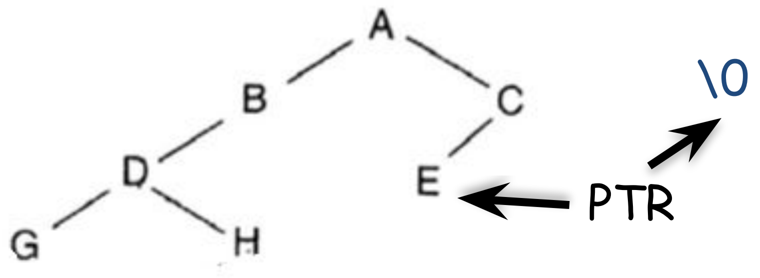
	7	
	6	
	5	
TOP	4	E
TOP	3	C
	2	A
	1	\0

STACK

G H D B

- **Basic Step 2:**
  - PTR← POP() = E

Traversal Algorithm using Stacks (Postorder Traverse)



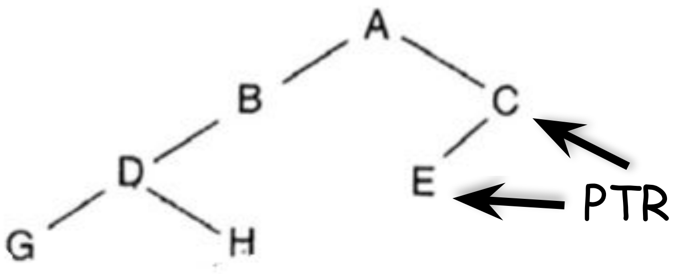
	7	
	6	
	5	
TOP	4	E
TOP	3	C
	2	A
	1	\0

STACK

G H D B

- **Basic Step 2:**
  - PTR POP() = E

Traversal Algorithm using Stacks (Postorder Traverse)



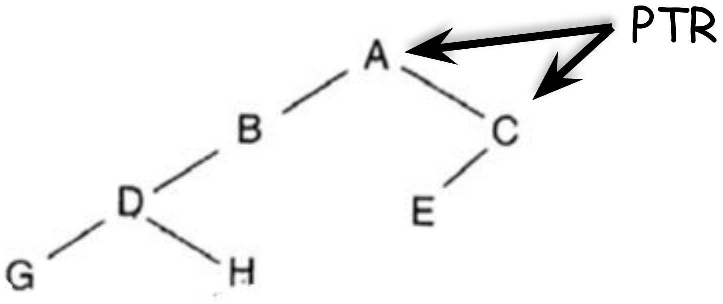
	7	
	6	
	5	
	4	
TOP	3	C
TOP	2	A
	1	\0

STACK

G H D B E

- **Basic Step 3:**
  - Process E
  - PTR□POP()=C
  - PTR > 0, continue

Traversal Algorithm using Stacks (Postorder Traverse)



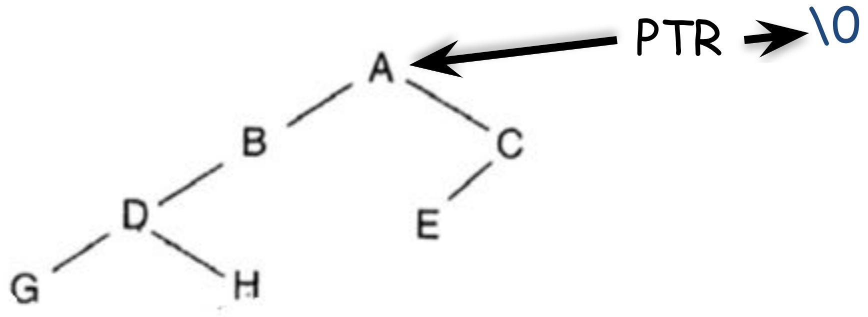
	7	
	6	
	5	
	4	
	3	
TOP	2	A
TOP	1	\0

STACK

G H D B E C

- **Basic Step 3:**
  - Process C
  - PTR□POP()=A
  - PTR > 0, continue

Traversal Algorithm using Stacks (Postorder Traverse)



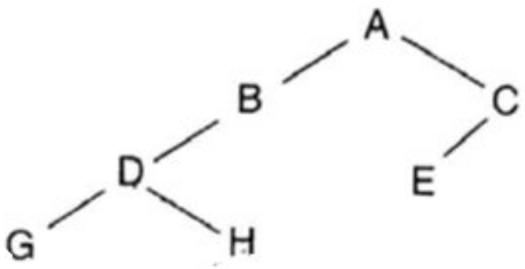
	7	
	6	
	5	
	4	
	3	
	2	
TOP	1	\0

STACK

G H D B E C A

- **Basic Step 3:**
  - Process A
  - PTR□POP()=\0
  - PTR = 0, BREAK

Traversal Algorithm using Stacks (Postorder Traverse)



PTR → \0

7	
6	
5	
4	
3	
2	
1	

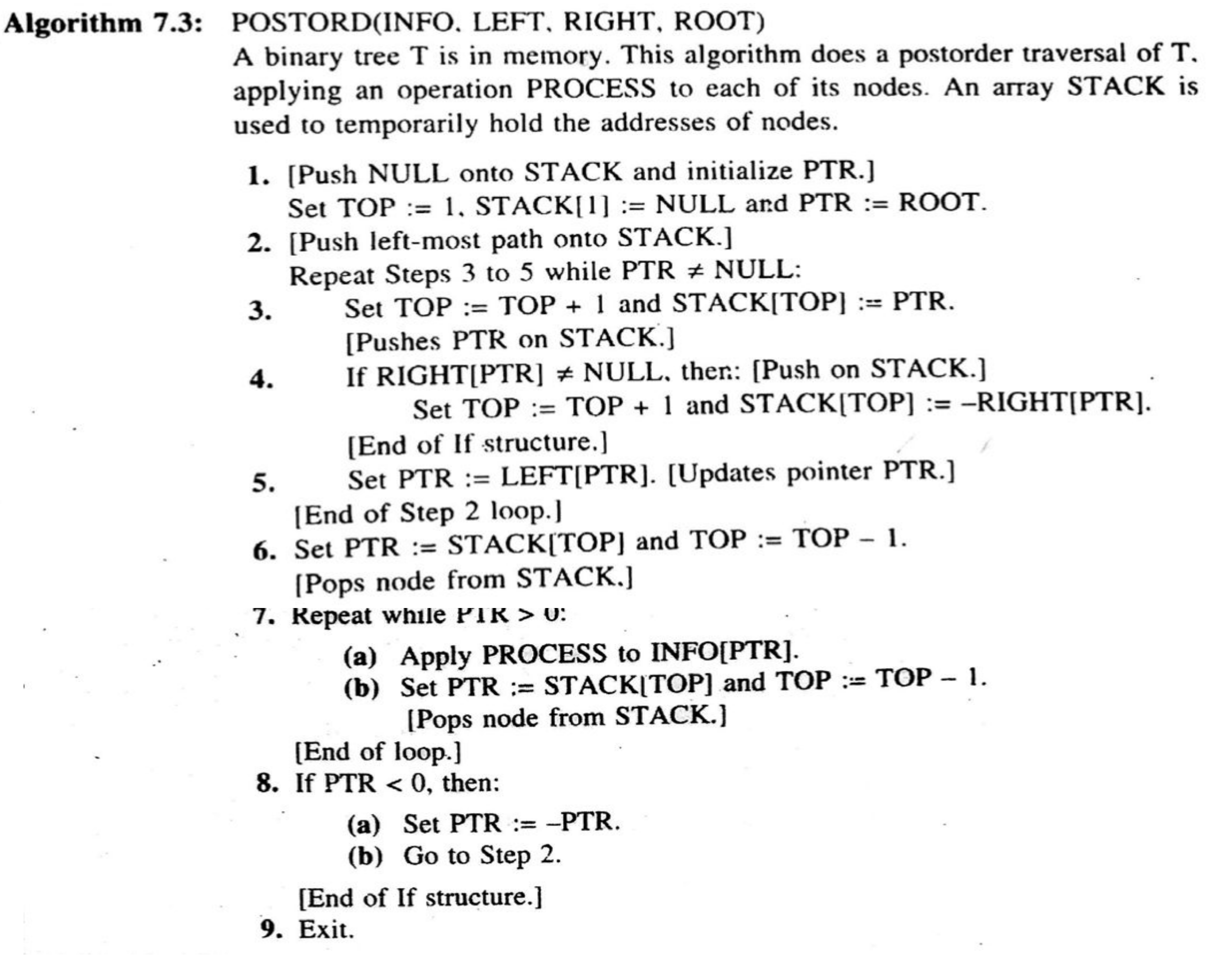
STACK

G H D B E C A

- **Basic Step 4:**
  - Not execute

- PTR\0 & STACK empty
  - (Terminate Algorithm)

Traversal Algorithm using Stacks (Postorder Traverse)



Algorithm 7.3: POSTORD(INFO, LEFT, RIGHT, ROOT)

A binary tree T is in memory. This algorithm does a postorder traversal of T, applying an operation PROCESS to each of its nodes. An array STACK is used to temporarily hold the addresses of nodes.

```
1. [Push NULL onto STACK and initialize PTR.]
   Set TOP := 1, STACK[1] := NULL and PTR := ROOT.
2. [Push left-most path onto STACK.]
   Repeat Steps 3 to 5 while PTR ≠ NULL:
3.       Set TOP := TOP + 1 and STACK[TOP] := PTR.
         [Pushes PTR on STACK.]
4.       If RIGHT[PTR] ≠ NULL, then: [Push on STACK.]
               Set TOP := TOP + 1 and STACK[TOP] := –RIGHT[PTR].
         [End of If structure.]
5.       Set PTR := LEFT[PTR]. [Updates pointer PTR.]
   [End of Step 2 loop.]
6. Set PTR := STACK[TOP] and TOP := TOP – 1.
   [Pops node from STACK.]
7. Repeat while PTR > 0:
       (a) Apply PROCESS to INFO[PTR].
       (b) Set PTR := STACK[TOP] and TOP := TOP – 1.
           [Pops node from STACK.]
   [End of loop.]
8. If PTR < 0, then:
       (a) Set PTR := –PTR.
       (b) Go to Step 2.
   [End of If structure.]
9. Exit.
```

Any Query?

Data Structure