Data Structure

INTRODUCTION TO DATA STRUCTURE

UNIT - 2

ANKIT VERMA

ANKIT.VERMA@IITMJP.AC.IN
WWW.ANKITVERMA.CO.IN

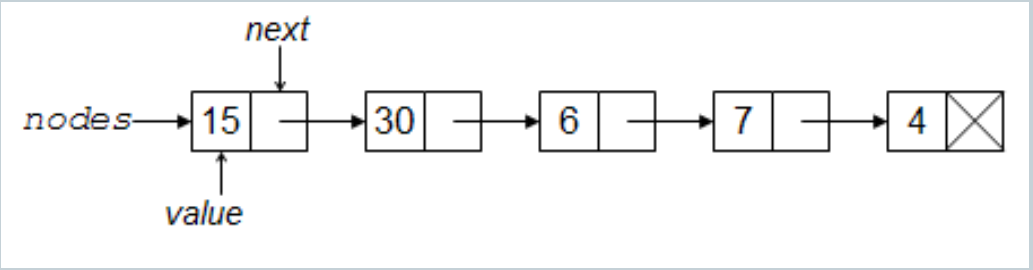
DEPARTMENT OF INFORMATION TECHNOLOGY

Which Book To Follow ?

- Text Book
  - Data Structures, Schaum's Outlines, Seymour Lipschutz
- Reference Book
  - Data Structure Using C, Udit Aggarwal

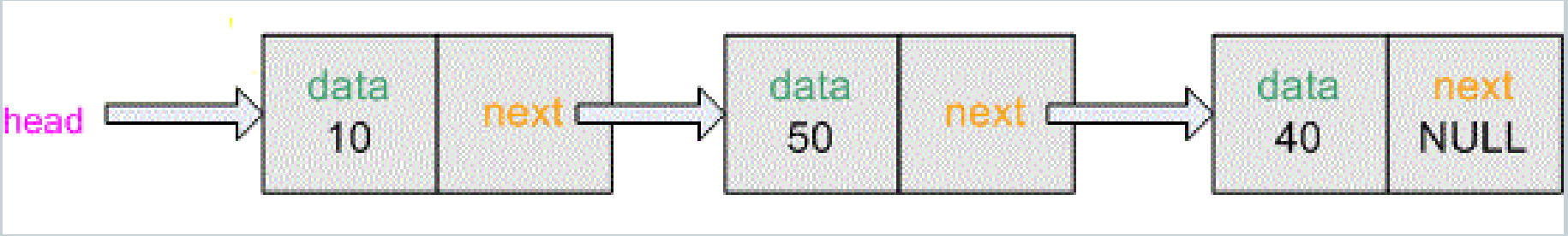
Linked Lists

- Linked Lists
  - Also called One-Way List
  - Linked List is Linear collection of Data Elements called **Nodes**
  - Linear order is given by Pointers
  - Each Node divided into 2 Parts:
    - Information
      - Contain Information of Element
    - Link Field / Next Pointer Field
      - Contain Address of Next Node in List



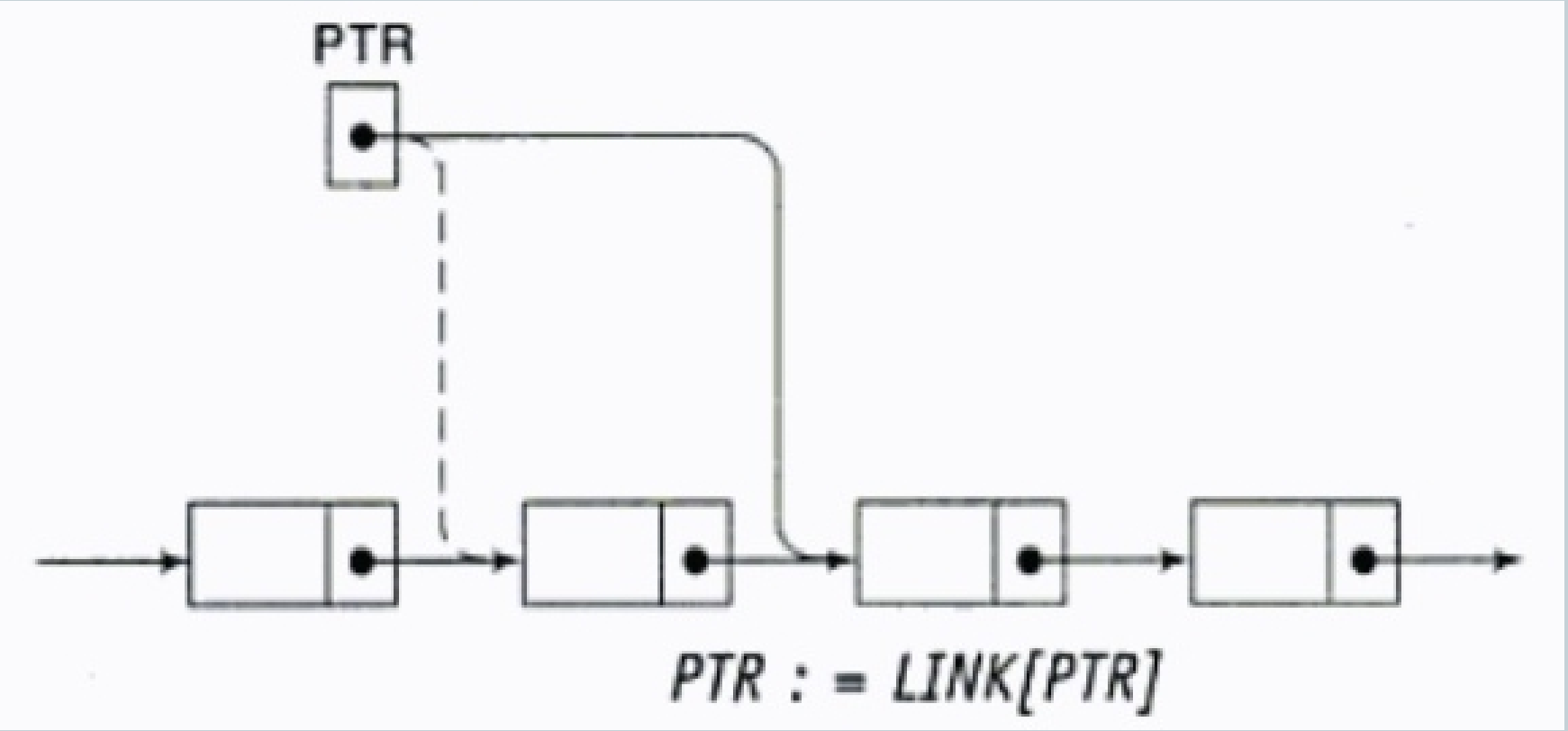
Linked Lists

- Null Pointer
  - Denoted by X in diagram & Signals the End of List
- Start / Name
  - It is List Pointer Variable which contain Address of First Node in Linked Lists
- Null List / Empty List
  - Special case in which List has No Node
  - It is Denoted by Null Pointer in Start



Traversing Linked List

- Traversing Linked List
  - PTR := LINK [PTR] moves Pointer to the Next Node in the List



Traversing Linked List

- ALGORITHM: Traversing Linked List
  - Let LIST be a Linked List in Memory. This algorithm traverse LIST, applying an operation PROCESS to each element of LIST. The variable PTR points to node currently being processed.
    1. Set PTR := START.
    2. Repeat Steps 3 and 4 while PTR != NULL.
    3. Apply PROCESS to INFO[PTR].
    4. Set PTR := LINK[PTR].
    5. Exit.

Searching Linked List

- Searching Linked List
  - ITEM can be Searched in Linked List in following 2 Conditions:
    - LIST is Unsorted
    - LIST is Sorted

Searching In Unsorted Linked List

- ALGORITHM: SEARCH (INFO, LINK, START, ITEM, LOC)
  - LIST is the linked list in memory. The algorithm finds the location LOC of the node where ITEM first appears in LIST, or sets LOC=NULL.

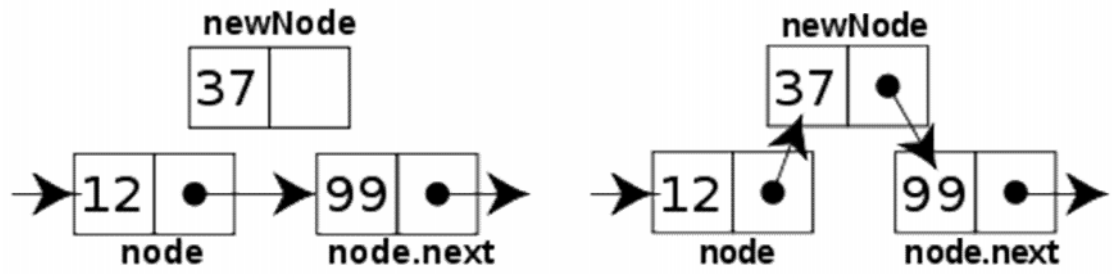
1. Set PTR := START.
2. Repeat Step 3 while PTR != NULL:
3. If ITEM = INFO[PTR], then:
   Set LOC := PTR, and Exit.
   Else:
   Set PTR := LINK[PTR].
4. Set LOC := NULL.
5. Exit

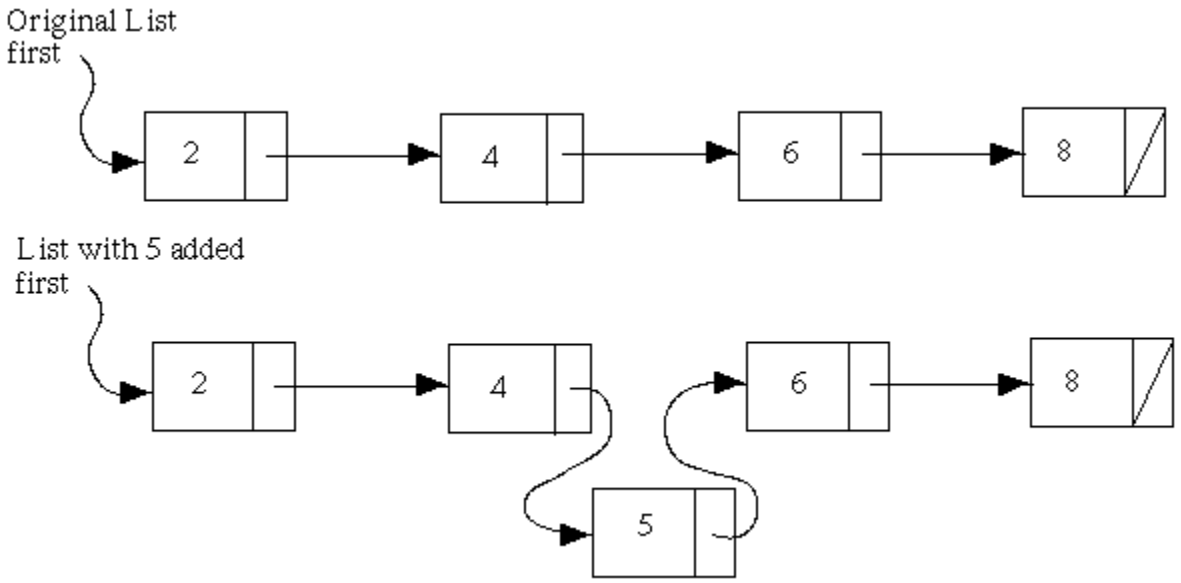
Searching In Sorted Linked List

- ALGORITHM: SEARCH (INFO, LINK, START, ITEM, LOC)
  - LIST is the linked list in memory. The algorithm finds the location LOC of the node where ITEM first appears in LIST, or sets LOC=NULL.

1. Set PTR := START
2. Repeat Step 3 while PTR != NULL:
3. If ITEM < INFO[PTR], then:
4. Set PTR := LINK[PTR].
5. Else if ITEM = INFO[PTR], then:
6. Set LOC := PTR, and Exit.
7. Else:
8. Set LOC := NULL, and Exit.
9. Set LOC := NULL
10. Exit.

Linked List Insertion





Insertion into Linked List

- Insertion into Linked List
  - Algorithm will include following Steps:
    - Check if Space is Available in AVAIL List or Not.
      - If AVAIL = NULL
        - Print Message OVERFLOW
    - Remove Node from Available List
      - NEW = AVAIL
      - AVAIL = LINK [AVAIL]
    - Copy Item into New Node
      - INFO[NEW] = ITEM

Insert At Beginning Of Linked List

- ALGORITHM: INSFIRST (INFO, LINK, START, AVAIL, ITEM)
  - This algorithm inserts ITEM as the first node in the List.
    1. If AVAIL = NULL, then: Write OVERFLOW and Exit.
    2. Set NEW := AVAIL and AVAIL := LINK[AVAIL].
    3. Set INFO[NEW] := ITEM.
    4. Set LINK[NEW] := START.
    5. Set START := NEW.
    6. Exit.

Insert After Given Node In Linked List

- ALGORITHM: INSLOC (INFO, LINK, START, AVAIL, LOC, ITEM)
  - This algorithm inserts ITEM so that ITEM follows the node with location LOC or inserts ITEM as the first node when LOC = NULL.
    1. If AVAIL = NULL, then: Write OVERFLOW and Exit.
    2. Set NEW := AVAIL and AVAIL := LINK[AVAIL].
    3. Set INFO[NEW] := ITEM.
    4. If LOC = NULL, then:
       Set LINK[NEW] := START and START := NEW.
       Else:
       Set LINK[NEW] := LINK[LOC] and LINK[LOC] := NEW.
    5. Exit.

Insert Into Sorted Linked List

- ALGORITHM 1: FINDA (INFO, LINK, START, ITEM, LOC)
  - This procedure finds the location LOC of the last node in a sorted list such that INFO[LOC] < ITEM, or sets LOC = NULL.
    1. If START = NULL, then: Set LOC := NULL and Return.
    2. If ITEM < INFO[START], then: Set LOC := NULL, and Return.
    3. Set SAVE := START and PTR := LINK[START].
    4. Repeat Steps 5 and 6 while PTR != NULL.
    5. If ITEM < INFO[PTR], then:
       Set LOC := SAVE, and Return.
    6. Set SAVE := PTR and PTR := LINK[PTR].
    7. Set LOC := SAVE.
    8. Return.

Insert Into Sorted Linked List

- ALGORITHM 2: INSERT (INFO, LINK, START, AVAIL, ITEM)
  - The algorithm inserts ITEM into a sorted linked list.
    1. Call FINDA (INFO, LINK, START, ITEM, LOC).
    2. Call INSLOC (INFO, LINK, START, AVAIL, LOC, ITEM).
    3. Exit.

Deleting Node Of Linked List

- ALGORITHM: DEL (INFO, LINK, START, AVAIL, LOC, LOCP)
  - This algorithm Deletes the node N with location LOC. LOCP is the location of node which precedes N or, when N is the first node, LOCP = NULL.

1. If LOCP = NULL, then:

   Set START := LINK[START].

   Else:

   Set LINK[LOCP] := LINK[LOC].
2. Set LINK[LOC] := AVAIL and AVAIL := LOC.
3. Exit.

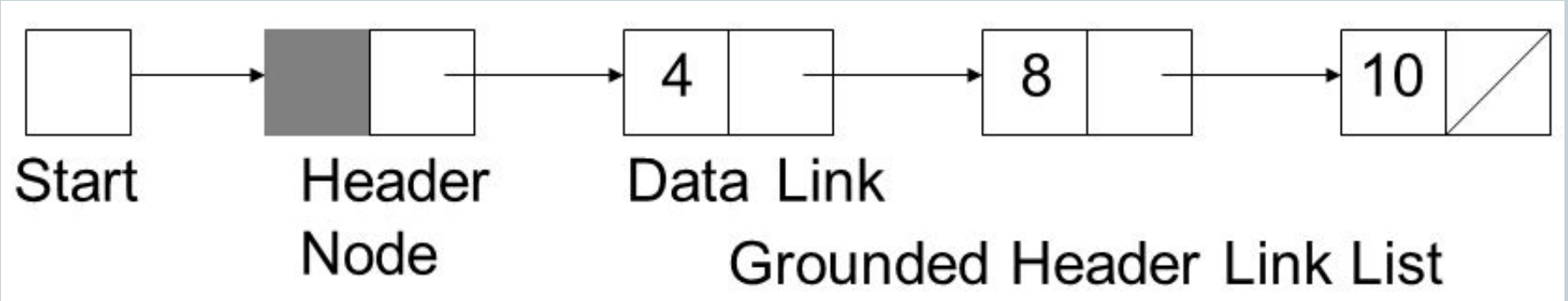
Header Linked List

Header Linked Lists

- Header Linked List is a Linked List which always contain a Special Node called **Header Node**, at beginning of List.
- Two kinds of Header Linked Lists:
  - Grounded Header List
  - Circular Header List

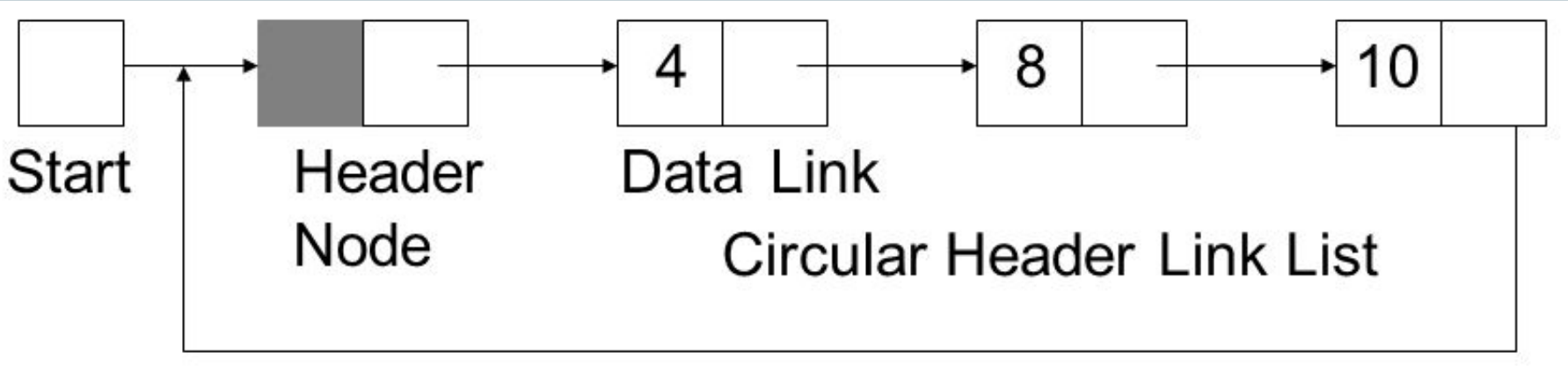
Header Linked Lists

- Grounded Header List
  - It is a Header List where Last Node contain Null Pinter.
  - Term Grounded indicate the Null Pointer.
  - START always Point to Header Node so LINK[START] = NULL means Grounded List is Empty.



Header Linked Lists

- Circular Header List
  - It is a Header List where Last Node points back to Header Node.
  - START always Point to Header Node so LINK[START] = START indicates Circular Header List is Empty.
  - Unless otherwise stated or implied, Header List always be Circular.
  - Header Node acts as a sentinel indicating the End of List.



Header Linked Lists

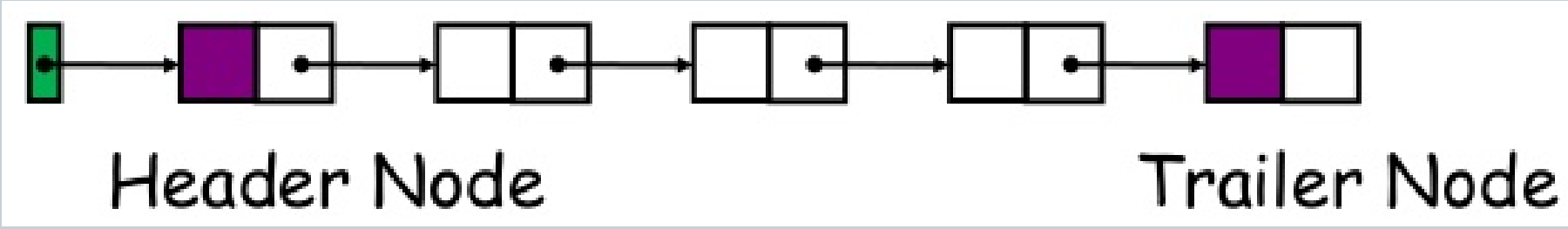
- Header List vs Ordinary List
  - Node refers to Ordinary Node, not the Header Node.
  - First Node in Header List is Node following Header Node.
  - Location of First Node is LINK[START], not START as Ordinary List.
  - Circular Header Lists are Easier to State & Implement so more frequently used instead of Ordinary List.
  - Circular Header Lists have properties over Ordinary Lists:
    - Null Pointer is Not Used, and hence All Pointers contain Valid Addresses.
    - Ordinary Node has a Predecessor, so First Node may not require a Special Case.

Traversing Circular Header List

- ALGORITHM: Traversing Circular Header List
  - Let LIST be a Circular Header List in memory. The Algorithm traverses LIST, applying an operation PROCESS to each Node of LIST.
    1. Set PTR := LINK[START].
    2. Repeat Steps 3 and 4 while PTR != NULL.
    3. Apply PROCESS to INFO[PTR].
    4. Set PTR := LINK[PTR].
    5. Exit.

Header Linked Lists

- Special Type of Header List
  - Linked List with Header & Trailer Node
    - A Linked List which contain both a Special Header Node at Beginning of List & Special Trailer Node at End of List.



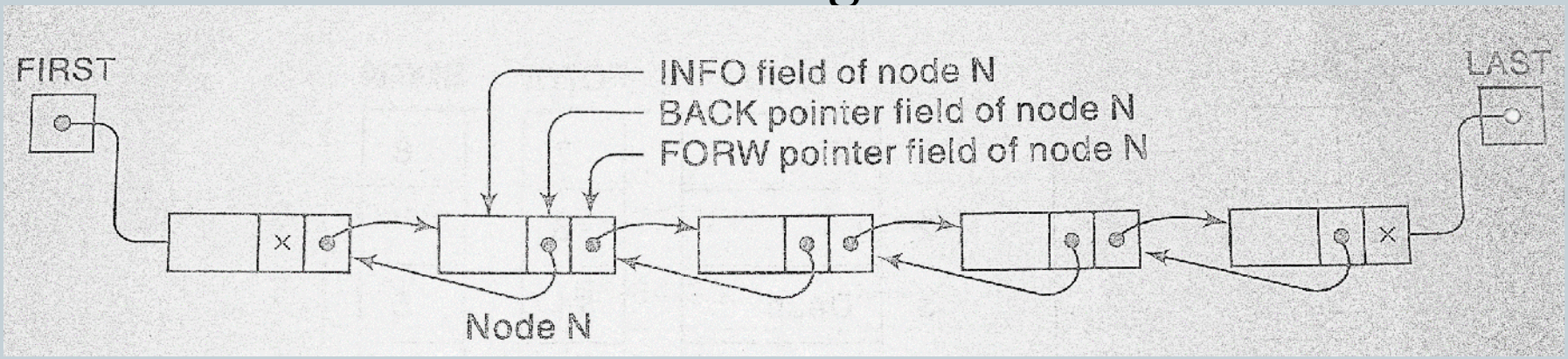
Header Linked Lists

- Important About Header List
  - Although our Data may be Maintained by Header List in Memory, the AVAIL List will always be maintained as an Ordinary Linked List.

Two-Way List

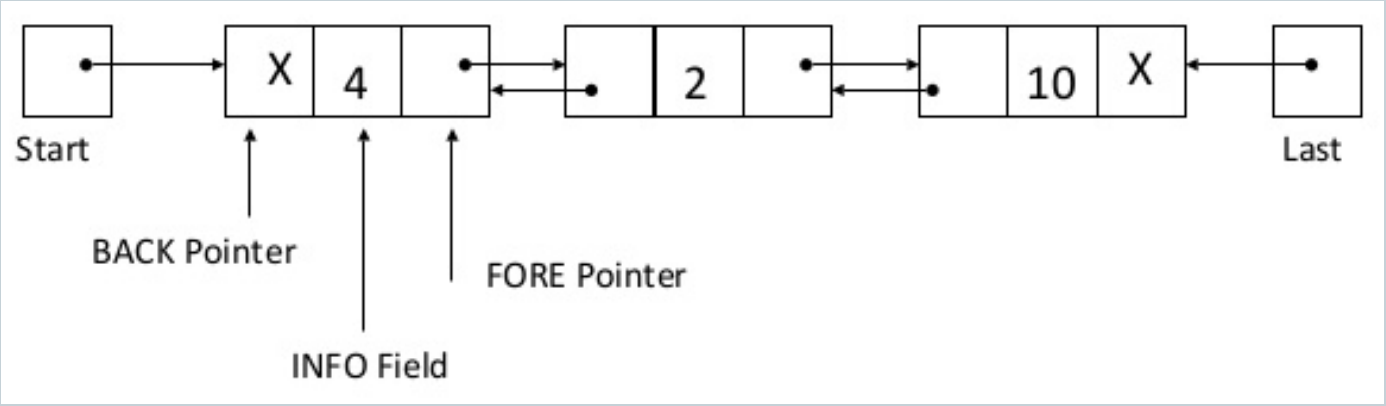
Two-Way Lists

- Two-Way List can be Traversed in Two Directions:
  - Forward Direction
    - From Beginning of List to End
  - Backward Direction
    - From End of List to Beginning
- Given Location of Node has immediate Access to both Next Node & Preceding Node in List.



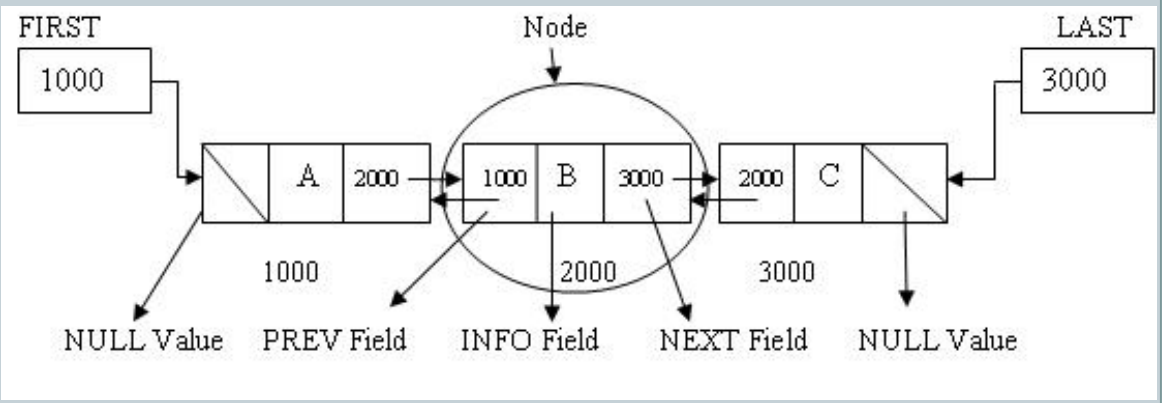
Two-Way Lists

- Two-Way List is Linear Collection of Data Elements called Nodes, where each Node is divided into 3 Parts:
  - Information Field INFO
    - Which contain Data
  - Pointer Field FORW
    - Which contain Location of Next Node
  - Pointer Field BACK
    - Which contain Location of Preceding Node



Two-Way Lists

- List Require Two Pointer Variable:
  - FIRST
    - Points to First Node of List
  - LAST
    - Points to Last Node of List
- Move Forward & Backward
  - If Node B follows Node A and LOCB & LOCA are their Locations then:
    - FORW[LOCA] = LOCB
    - BACK[LOCB] = LOCA

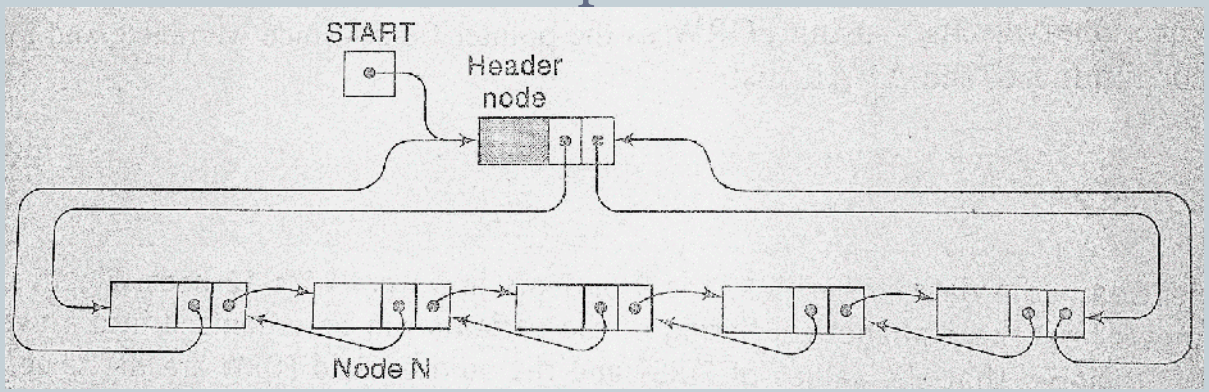


Two-Way Lists

- Disadvantage of Two-Way Lists
  - Storing Data requires Extra Space for Backward Pointers & Extra Time to change the Added Pointers.
- Important Things about Two-Way Lists
  - Two-Way List require Two Pointers, FORW & BACK, instead One Pointer LINK.
  - AVAIL List of Available Space still maintained as One-Way List, using FORW Pointer, so Addition & Deletion will be only at Beginning of AVAIL List.

Two-Way Header Lists

- Two-Way Header Lists
  - Two-Way Circular Header List combine Advantages of Two-Way List & Circular Header List.
  - List is Circular because Two End Nodes point back to Header Node.
  - Two-Way Header List require only one Pointer START, which points to Header Node.
  - Two Pointers in Header Node point to Two Ends of List.



Two-Way Header Lists

- Operations on Two-Way Lists
  - Traversing
  - Searching
  - Deleting
  - Inserting

Deleting Node in Two-Way List

- ALGORITHM: DELTWL (INFO, FORW, BACK, START, AVAIL, LOC)

1. Set FORW[BACK[LOC]] := FORW[LOC] and BACK[FORW[LOC]] := BACK[LOC].
2. Set FORW[LOC] := AVAIL and AVAIL := LOC.
3. Exit.

Inserting Node in Two-Way List

- ALGORITHM: INSTWL (INFO, FORW, BACK, START, AVAIL, LOCA, LOCB, ITEM)
  - LOCA & LOCB are adjacent locations of node A & node B, and ITEM will be inserted between Nodes A & B.
    1. If AVAIL = NULL, then Write: OVERFLOW, and Exit.
    2. Set NEW := AVAIL, AVAIL := FORW[AVAIL], INFO[NEW] := ITEM.
    3. Set FORW[LOCA] := NEW, FORW[NEW] := LOCB,
    4. BACK[LOCB] := NEW, BACK[NEW] := LOCA.
    5. Exit.

Trees

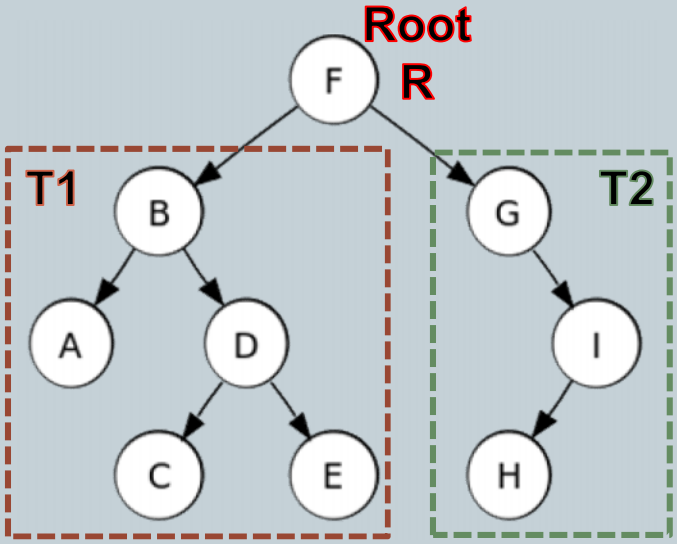
Trees

- Tree is Non-Linear Data Structure.
- Represent Data containing Hierarchical Relationship between Elements.
- Types of Tree
  - Binary Tree
    - Complete Binary Tree
    - Extended Binary Tree (2-Tree)
  - Binary Search Tree
  - AVL Search Tree
  - m-Way Tree
  - B-Tree

Binary Trees

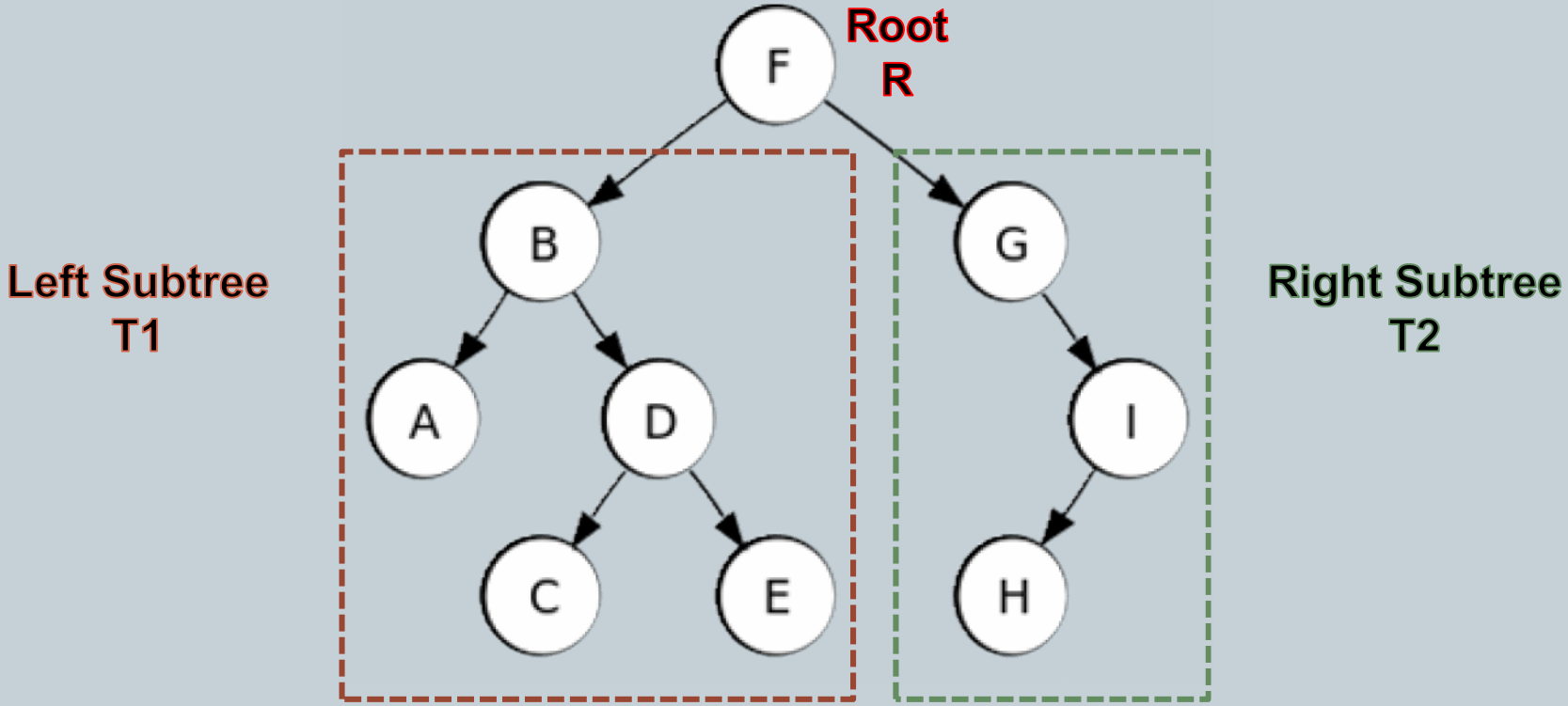
Binary Tree

- Binary Tree T is defined as a Finite Set of Elements called Nodes, such that:
- T is Empty, called **Null Tree / Empty Tree**, or
- T contains a distinguished Node R, called **Root** of T & remaining Nodes of T from an Ordered Pair of disjoint Binary Trees T1 & T2.



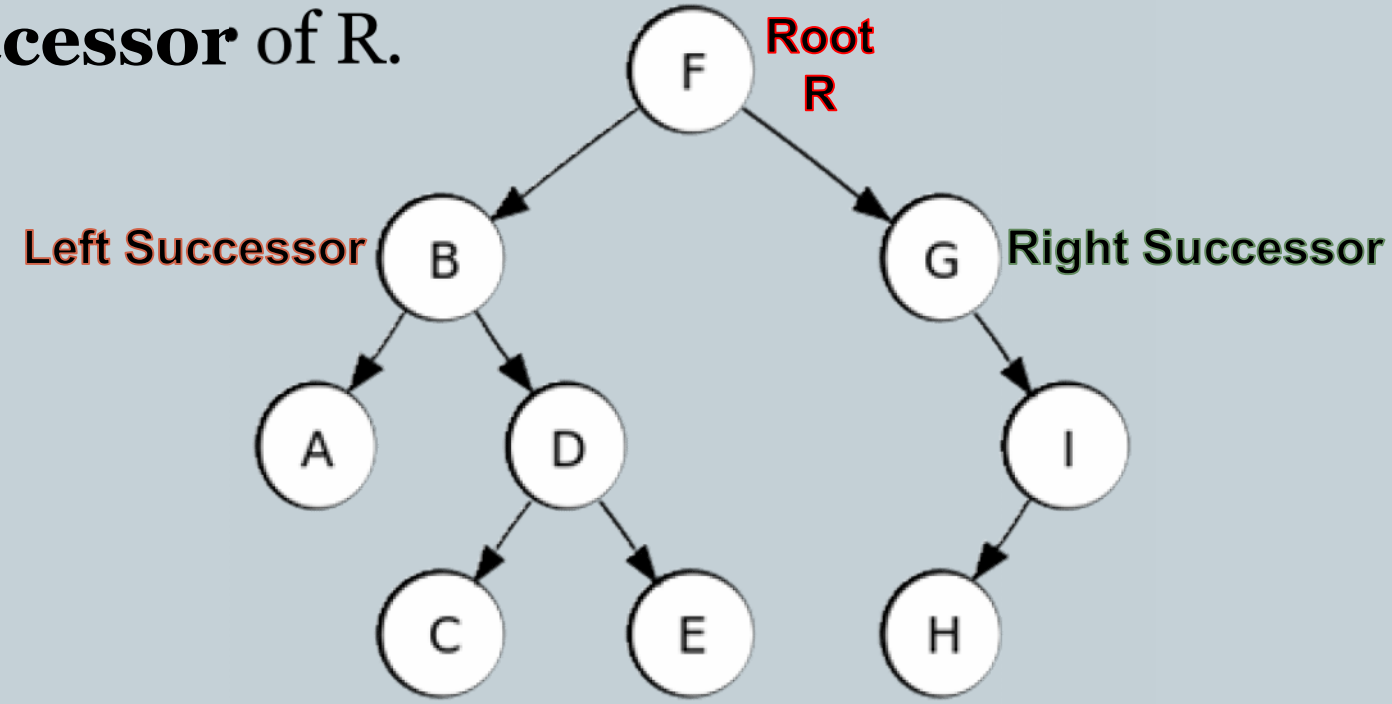
Binary Tree

- If T does contain a Root R, then two Trees **T1** & **T2** are called, respectively, **Left** & **Right Subtrees** of R.



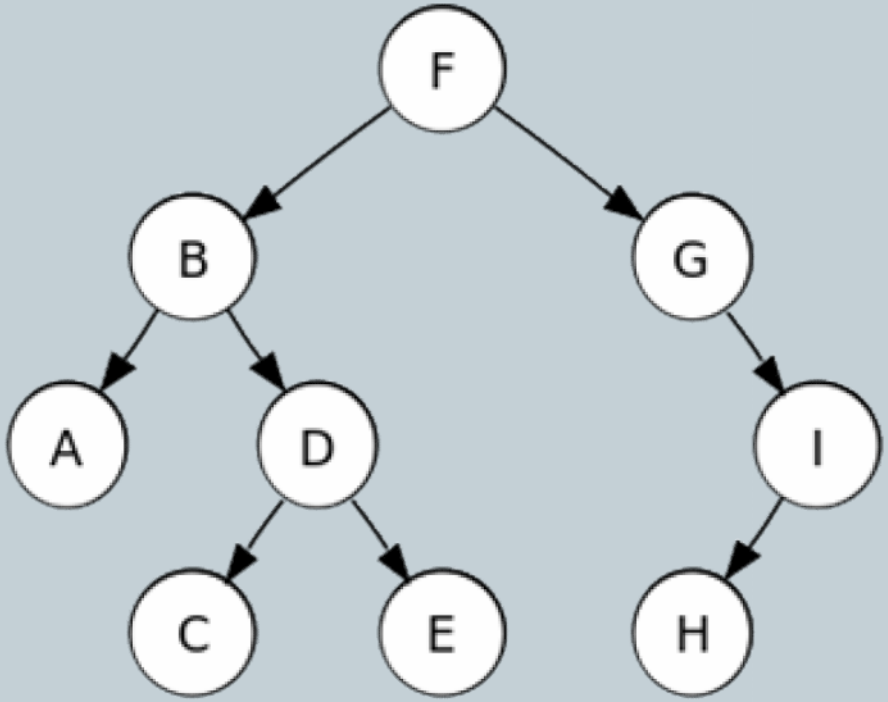
Binary Tree

- If T1 is Non Empty, then its Root is called **Left Successor** of R;
- If T2 is Non Empty, then its Root is called **Right Successor** of R.



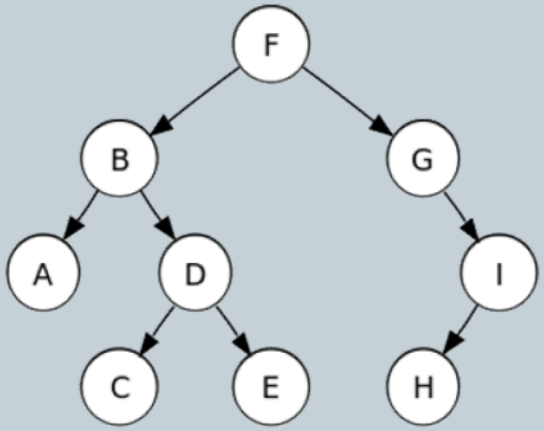
Binary Tree

- B is Left Successor & G is Right Successor of Node F.
- Left Subtree of Root F consists of Node B, A, D, C & E.
- Right Subtree of Root F consists of Node G, I & H.



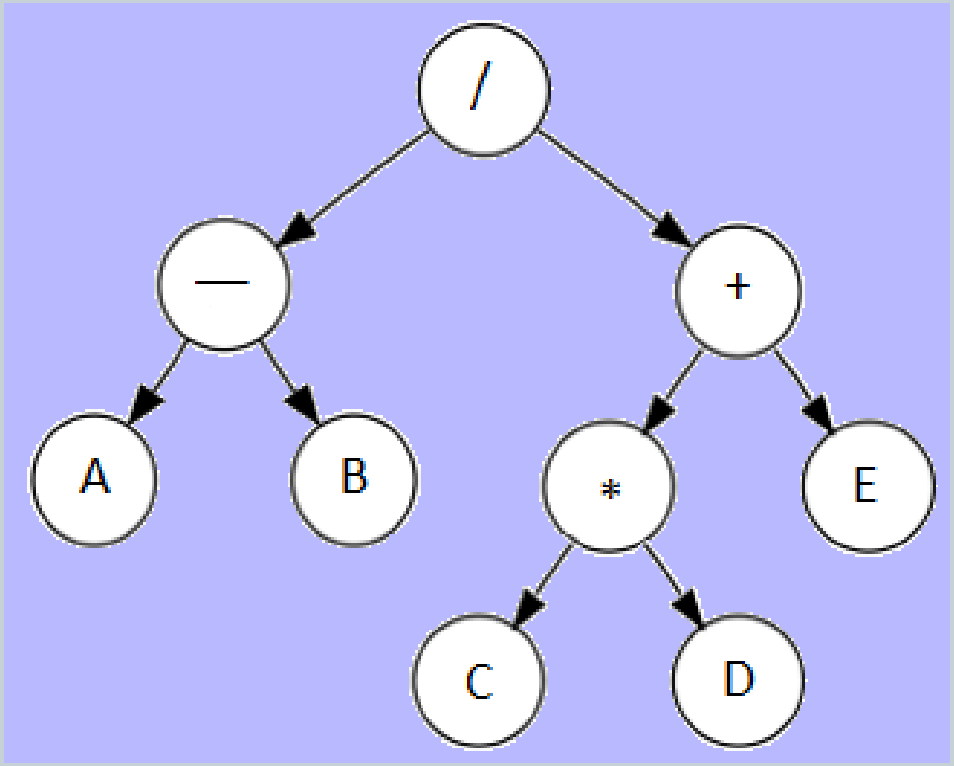
Binary Tree

- Any Node in Binary Tree T has either 0, 1 or 2 Successors.
- Nodes F, B, & D have 2 Successors.
- Nodes G & I have only 1 Successor.
- Nodes A, C, E & H have No Successor
- Node with No Successor is called **Terminal Node**.



Binary Tree

- Arithmetic Expression
  - E = (A – B) / ((C * D) + E)



Binary Tree Terminology

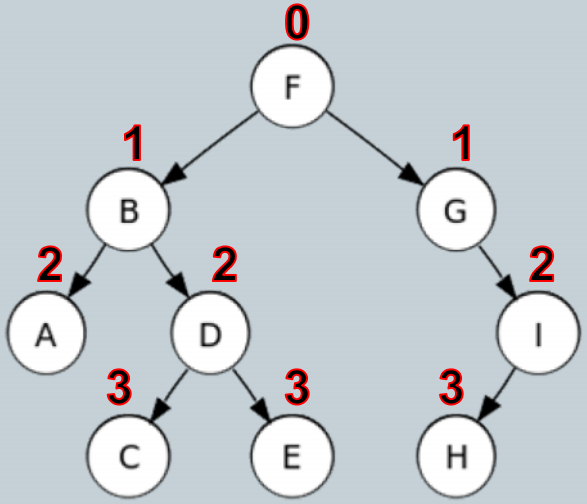
- Terminology describes Family Relationships used to describe Relationships between Nodes of Tree T.
- Parent, Child & Siblings
  - Suppose N is a Node in T with Left Successor S1 & Right Successor S2, then
    - N is called Parent (or Father) of S1 & S2.
    - S1 is called Left Child (or Son) of N & S2 is called Right Child (or Son) of N.
    - S1 & S2 are said to be Siblings (or Brothers).
- Edge
  - Line drawn from Node N to T to a Successor is called an Edge.

Binary Tree Terminology

- Path
  - Sequence of consecutive Edges is called Path.
- Leaf
  - Terminal Node is called Leaf.
- Branch
  - Path ending in a leaf is called Branch.
- Predecessor & Ancestor
  - Every Node N in Binary Tree T, except Root, has unique Parent called Predecessor of N.
  - If L is called Descendant of Node N, then N is called Ancestor of L.

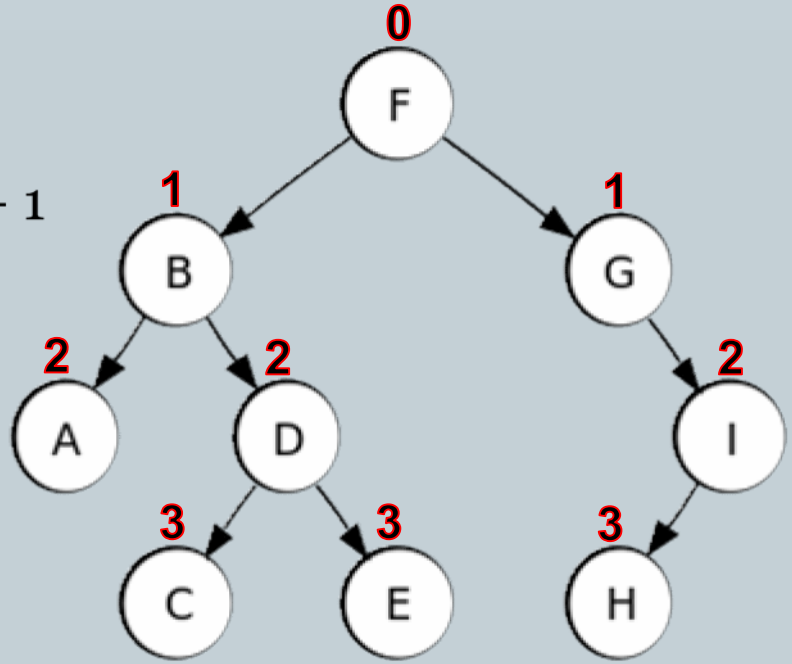
Binary Tree Terminology

- Level Number
  - Each Node in Binary Tree T is assigned a Level Number.
  - Root R of Tree T is assigned Level Number 0.
  - Every other Node is assigned a Level Number which is 1 more than the Level Number of it's Parent.
  - Same Level Number are said to belong to Same Generation.



Binary Tree Terminology

- Depth
  - Depth (or Height) of Tree T is maximum number of Nodes in a Branch of T.
  - This turns out to be 1 more than the largest Level Number of T.
  - Example Figure:
    - Largest Level Number = 3
    - Depth = Largest Level Number + 1
      = 3 + 1
      = 4

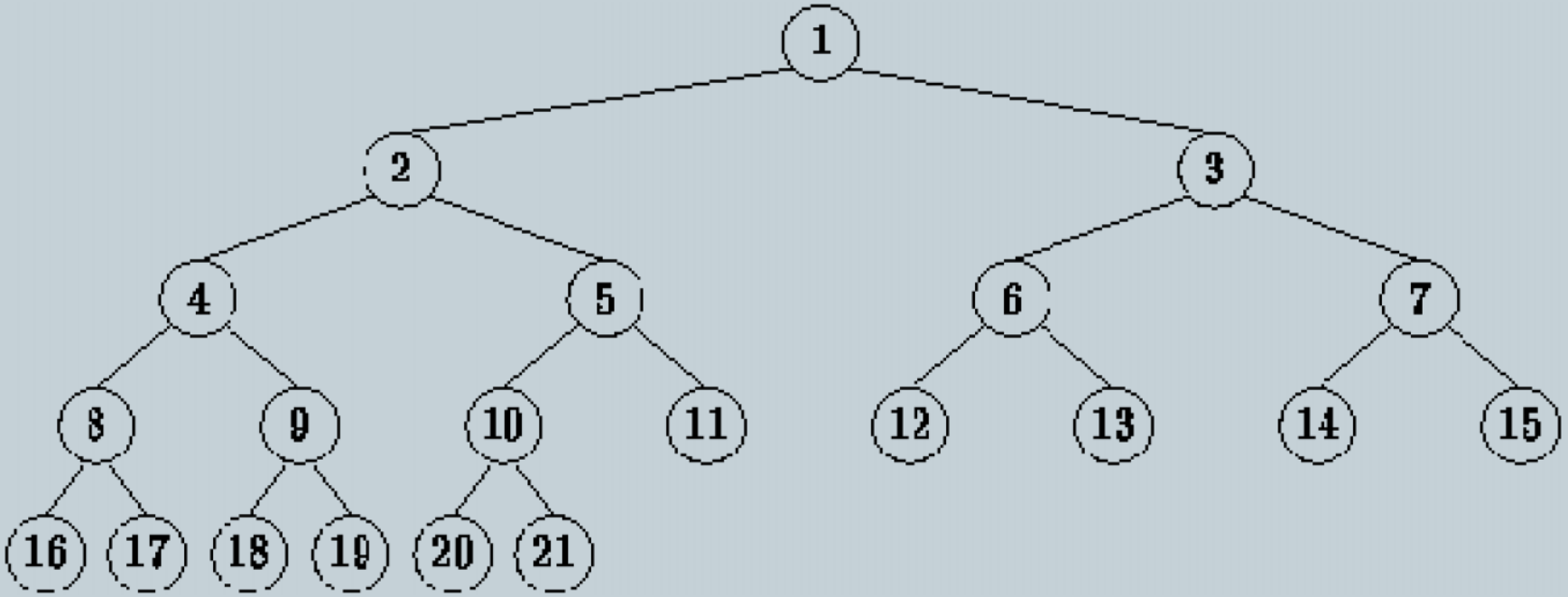


Complete Binary Tree

- Binary Tree
  - Each Node of T can have at most 2 Children.
  - Tree T of Level r can have at most $\mathbf{2^r}$ Nodes.
- Complete Binary Tree
  - Tree T is said to be Complete if all its Levels, except possibly the Last, have maximum number of possible Nodes, and if all the Nodes at the Last Level appear as far Left as possible.
  - It is unique Complete Tree $\mathbf{T_n}$ with exactly n Nodes.
    - E.g. Complete Tree T_{21} contain 21 Nodes.

Complete Binary Tree

- Complete Binary Tree T_{21} contain 21 Nodes.

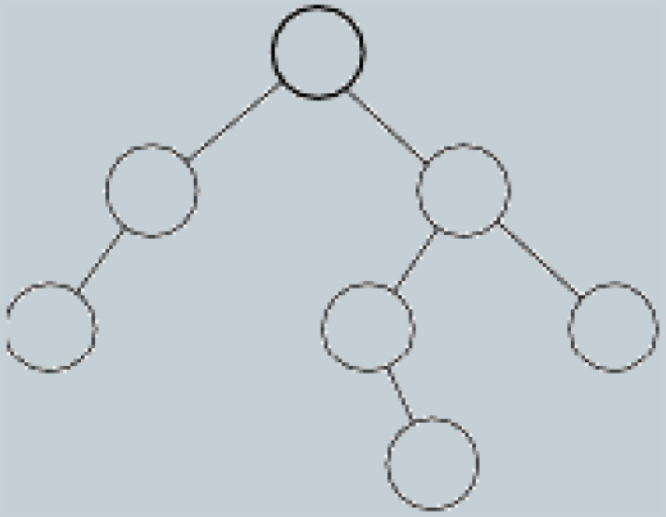


Extended Binary Tree: 2-Tree

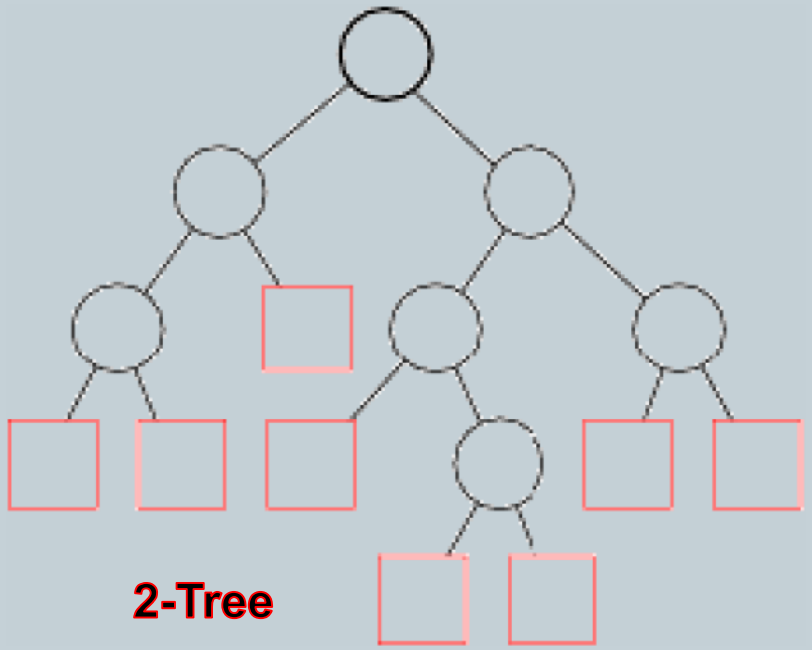
- Extended Binary Tree
  - A Binary Tree T is said to be a 2-Tree or an Extended Binary Tree if each Node N has either 0 or 2 Children.
  - Internal Nodes
    - Node with 2 Children are called Internal Nodes.
    - Sometimes to distinguish, they are represented by Circles.
  - External Nodes
    - Node with 0 Children are called External Nodes.
    - Sometimes to distinguish, they are represented by Squares.

Extended Binary Tree: 2-Tree

- Converting Binary Tree into 2-Tree
  - Replace each empty Subtree by a new Node as given in diagram.
  - Nodes in Original Tree T are Internal Nodes in Extended Tree.
  - New Nodes are External Nodes in Extended Tree.



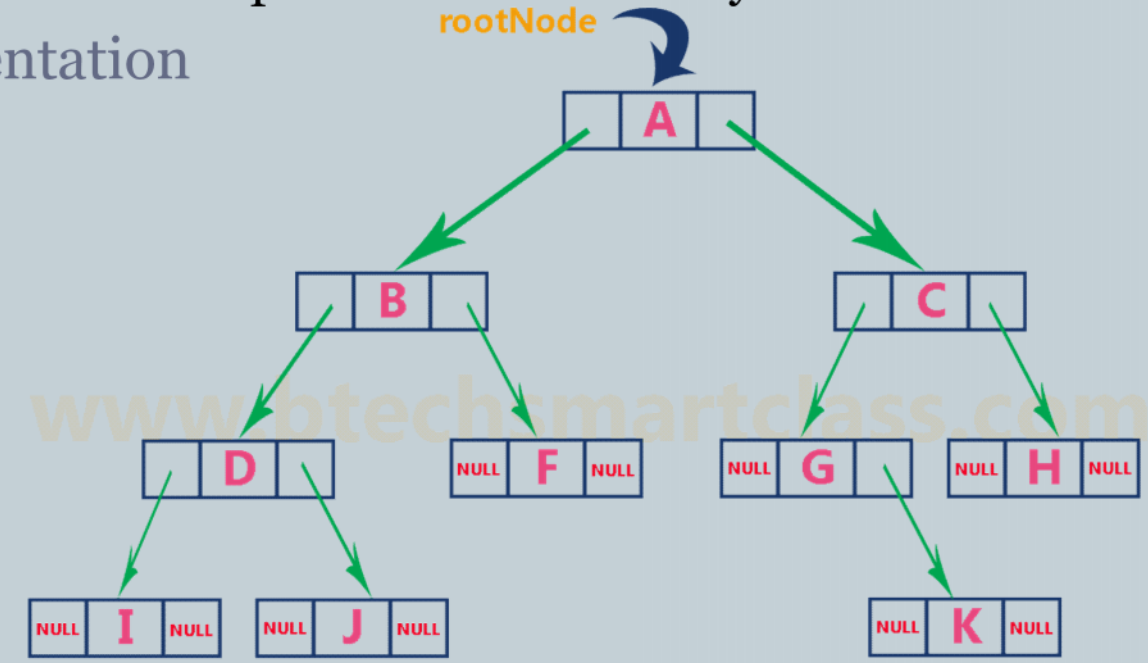
Binary Tree



2-Tree

Representing Binary Tree in Memory

- Two ways of representing Binary Tree T in Memory:
  - Linked List Representation
    - Use Linked List
    - Analogous to way Lists are represented in Memory
  - Sequential Representation
    - Uses Single Array



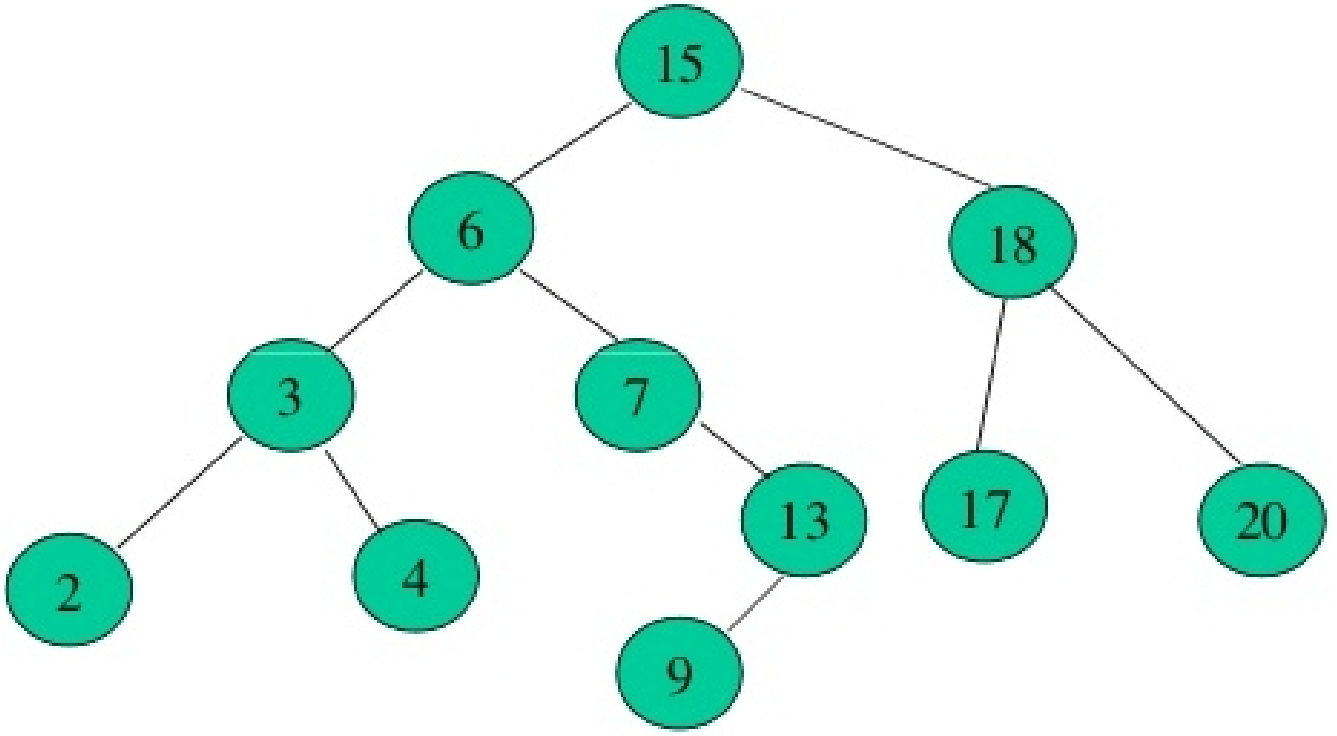
Representing Binary Tree in Memory

- Linked List Representation
  - Use three Parallel Arrays INFO, LEFT & RIGHT, and Pointer variable Root:
    - INFO[K]
      - Contains Data of Node N.
    - LEFT[K]
      - Contains Location of Left Child of Node N.
    - RIGHT[K]
      - Contains Location of Right Child of Node N.
    - ROOT
      - Contain Location of Root R of T.
      - If T is Empty then ROOT will contain NULL value.
      - If Subtree is Empty then Corresponding Pointer will contain NULL Value.

Traversing Binary Tree

- Preorder (Node-Left-Right) [NLR]
  - Process the Root R.
  - Traverse the Left Subtree of R in Preorder.
  - Traverse the Right Subtree of R in Preorder.
- Inorder (Left-Node-Right) [LNR]
  - Traverse the Left Subtree of R in Inorder.
  - Process the Root R.
  - Traverse the Right Subtree of R in Inorder.
- Postorder (Left-Right-Node) [LRN]
  - Traverse the Left Subtree of R in Preorder.
  - Traverse the Right Subtree of R in Preorder.
  - Process the Root R.

Traversing Binary Tree



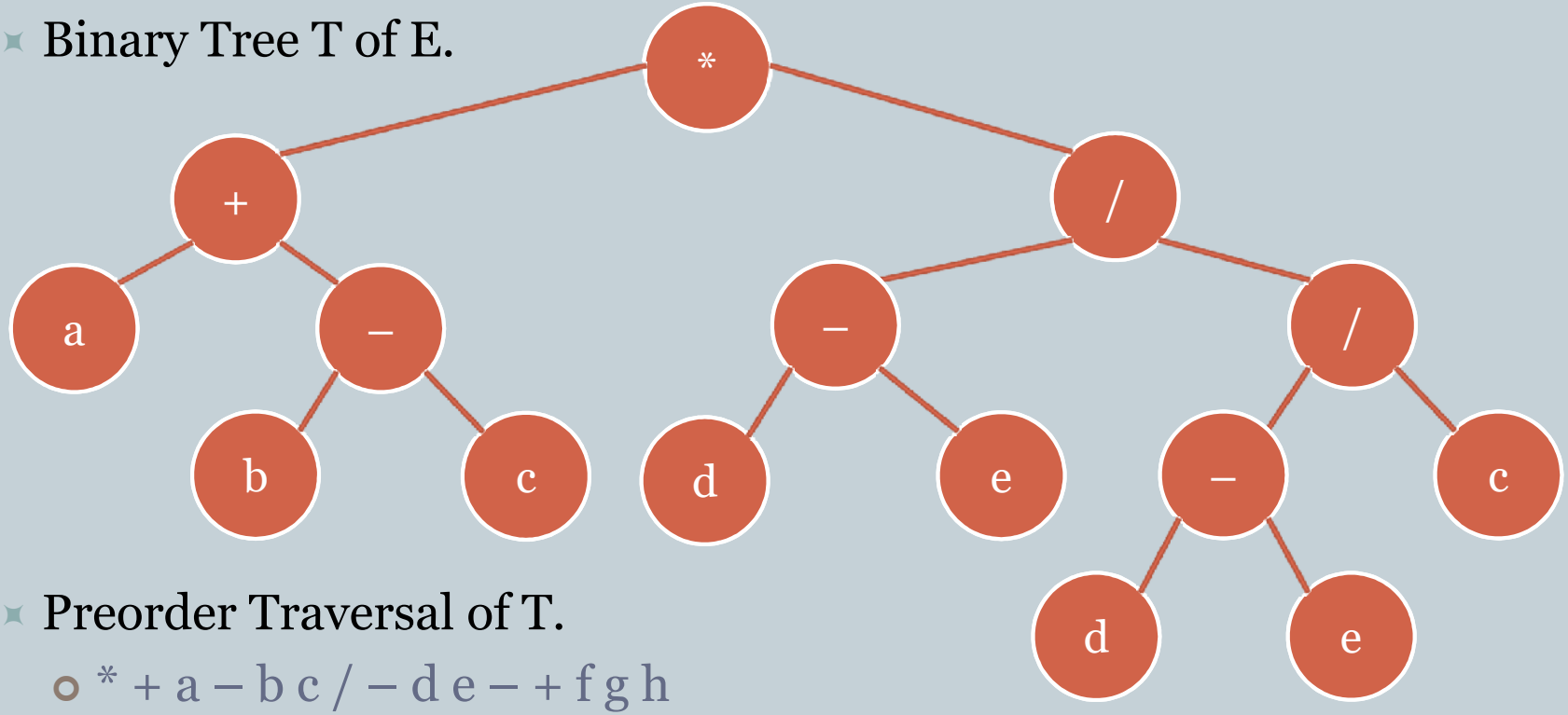
Preorder: 15, 6, 3, 2, 4, 7, 13, 9, 18, 17, 20
Inorder: 2, 3, 4, 6, 7, 9, 13, 15, 17, 18, 20
Postorder: 2, 4, 3, 9, 13, 7, 6, 17, 20, 18, 15

Binary Tree

- E denote following Arithmetic Expression:
  - E = [a + (b – c)] * [(d – e) / (f + g –h)]
    - Construct the Binary Tree T of E.
    - Calculate Preorder Traversal of T.
    - Calculate Postorder Traversal of T.

Binary Tree

- E = [a + (b – c)] * [(d – e) / (f + g –h)]
  - Binary Tree T of E.



  - Preorder Traversal of T.
    - * + a – b c / – d e – + f g h
  - Postorder Traversal of T.
    - a b c – + d e – f g + h – / *

Preorder Traversal

- ALGORITHM: PREORD (INFO, LEFT, RIGHT, ROOT)
  - The algorithm does a Preorder Traversal of Binary Tree T, applying operation PROCESS to each of Node. An array STACK is used to temporary hold the address of Nodes.
    1. Set TOP := 1, STACK[1] := NULL and PTR := ROOT.
    2. Repeat Steps 3 to 5 while PTR != NULL:
       Apply PROCESS to INFO[PTR].
    3. If RIGHT[PTR] != NULL, then:
       Set TOP := TOP + 1, and STACK[TOP] := RIGHT[PTR].
    4. If LEFT[PTR] != NULL, then:
       Set PTR := LEFT[PTR].
    5. Else:
       Set PTR := STACK[TOP] and TOP := TOP – 1.
    6. Exit

Inorder Traversal

- ALGORITHM: INORD (INFO, LEFT, RIGHT, ROOT)
  1. Set TOP := 1, STACK[1] := NULL and PTR := ROOT.
  2. Repeat while PTR != NULL:
     a) Set TOP := TOP + 1 and STACK[TOP] := PTR.
     b) Set PTR := LEFT[PTR].
  3. Set PTR := STACK[TOP] and TOP := TOP – 1.
  4. Repeat Steps 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 2.
  5. Set PTR := STACK[TOP] and TOP := TOP – 1.
  9. Exit

Postorder Traversal

- ALGORITHM: POSTORD (INFO, LEFT, RIGHT, ROOT)
  1. Set TOP := 1, STACK[1] := NULL and PTR := ROOT.
  2. Repeat Steps 3 to 5 while PTR != NULL:
  3. Set TOP := TOP + 1 and STACK[TOP] := PTR.
  4. If RIGHT[PTR] != NULL, then:
     Set TOP := TOP + 1, and STACK[TOP] := – RIGHT[PTR].
  5. Set PTR := LEFT[PTR].
  6. Set PTR := STACK[TOP] and TOP := TOP – 1.
  7. Repeat while PTR > 0:
     a) Apply PROCESS to INFO[PTR].
     b) Set PTR := STACK[TOP] and TOP := TOP – 1 .
  8. If PTR < 0, then:
     a) Set PTR := – PTR.
     b) Go to Step 2.
  9. Exit

THANKYOU

DOUBTS

PRESENTATION BY:
ANKIT VERMA
(IT DEPARTMENT)