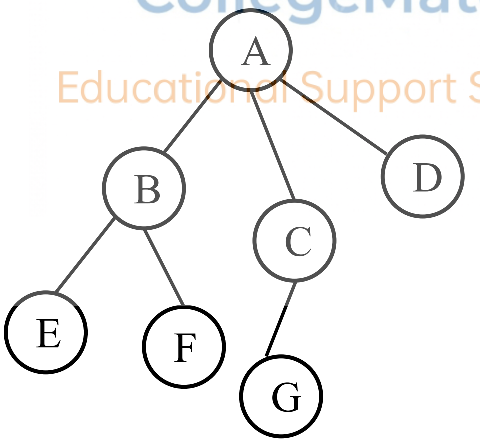
Trees

Dr. Jyoti
Professor
Department of Computer Science & Engineering
GJUST, Hisar

Tree

- It is a non-linear hierarchical data structure which shows hierarchical relationship between data elements.
  - It has a sequence of **Nodes**
  - The starting node is called **Root** of tree
  - Except root all other nodes have their **Parent** node
  - Node may have any number of **children**

For Example:-



- A is root of the tree.
- A has tree children B, C, D
- B is parent of E, F
- C has only one child G
- D has no child.

Tree Terminology

Root: A distinguish node with no parent

Level: Each node in tree T has assigned a level number. Root is assigned '0' level.

Sibling: Nodes belonging to same level number.

Generation: Nodes belong to same level are said to belong to same generation.

Ancestor: Means parents, grand-parents, great grand-parents etc.

Descendant: Means children, grand-children, great grand-children etc.

Leaf: Node which has no child or terminal node.

Edge: Line drawn from a node to its successor. A tree with n vertices has exactly (n-1) edges.

Path: A sequence of successive edges.

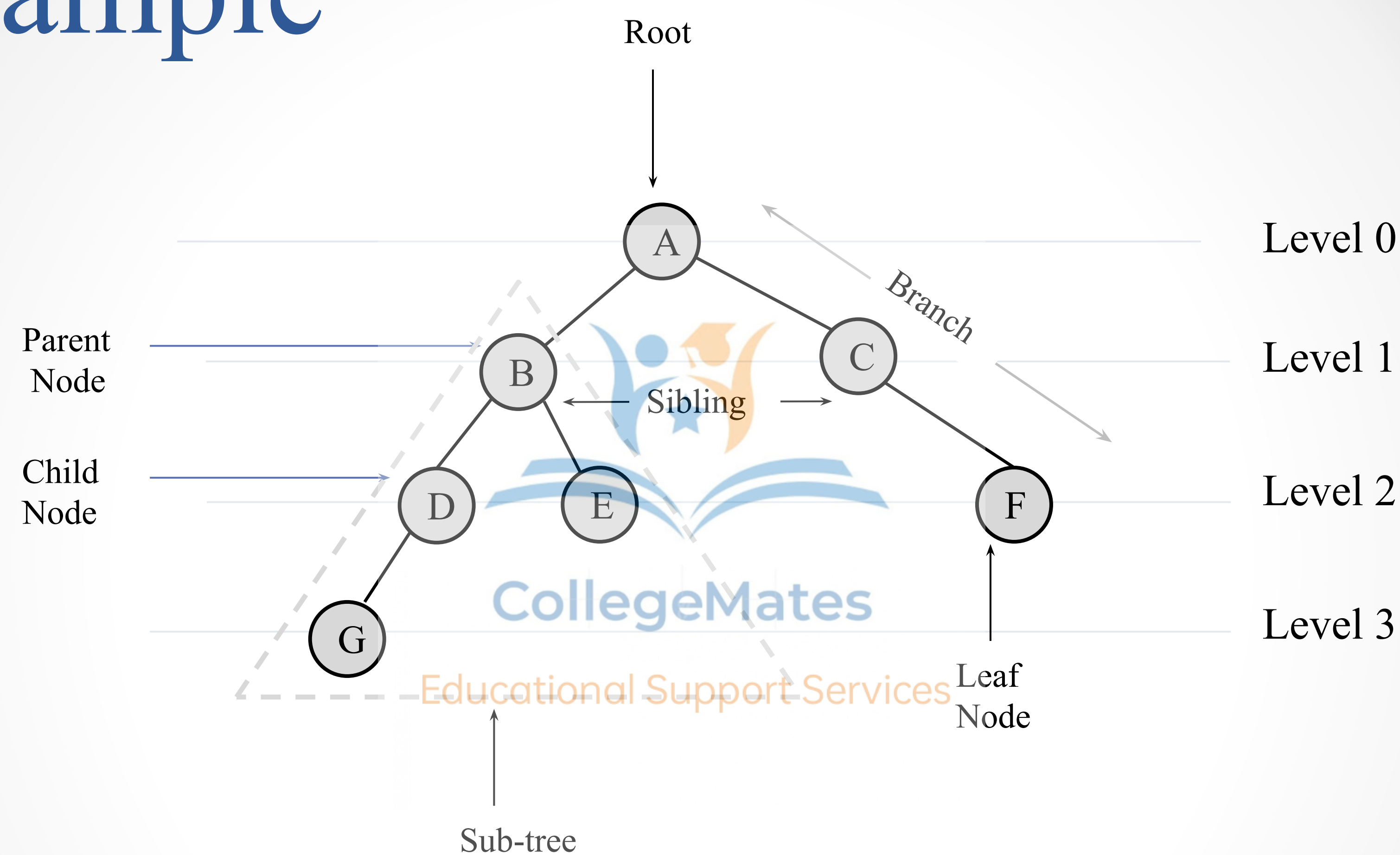
Branch: A path ending in a leaf node.

Depth/Height: Maximum number of nodes in a branch. It is 0 or more than 0.

Degree: Number of sub-trees of a node.

Key: Value represented at a node.

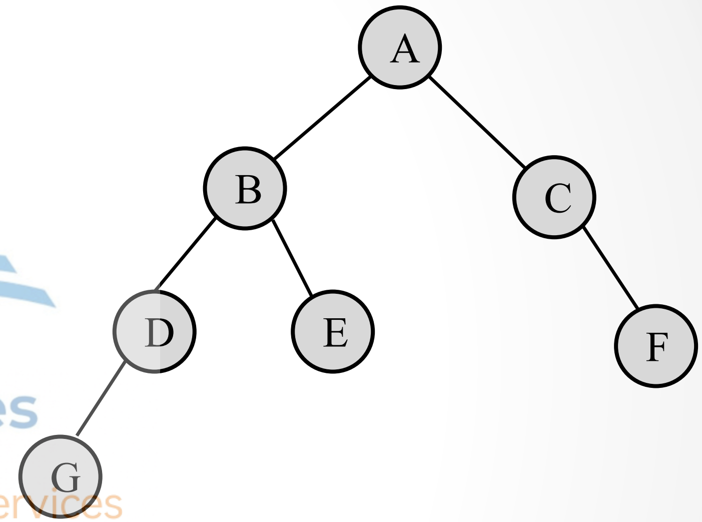
Example



- A is root of the tree.
- B, C are siblings
- G, E, F are leaves
- A-C, C-F are edges
- A-C-F is a branch

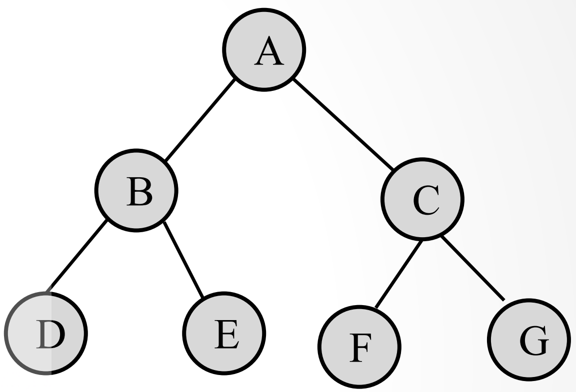
Binary Tree

- Binary Tree is a special kind of tree.
- It is defined as a finite set of elements called nodes.
- It has a distinguish node called root R.
- Rest of the nodes form an ordered pair of disjoint binary trees T1 and T2, where T1 and T2 are called left and right sub-trees.
- If T1 and T2 are non empty, then they are called left and right successor of R.
- In simple words, every node in a binary tree has either 0, 1 or 2 children.



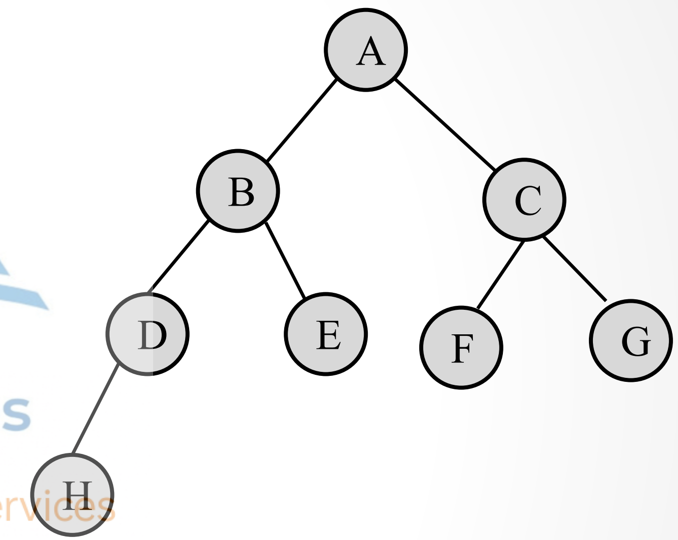
Strictly Binary Tree

- If every non-leaf node in a Binary Tree has only left and right sub-trees, then such a tree is called strictly binary tree.
- In simple words, every node in a strictly binary tree has either 0 or 2 children.
- A strictly binary tree with N leaf node always has 2*N-1 nodes.



Complete Binary Tree

- In Complete Binary Tree all the levels are completely filled except the last level and in the last level nodes appear in left.
- A complete binary tree has 2^d nodes at every depth d and 2^d-1 atleast non leaf nodes.



Binary Tree Representations

- <u>Static Representation</u>
  - For static representation of binary trees arrays are used. But it is not a flexible technique as the size of stack is fixed.
- <u>Dynamic Implementation</u>
  - For representation of binary trees linked lists are used for storing stacks in memory.

Array Representation

This representation uses only single 1-dimentional array TREE.

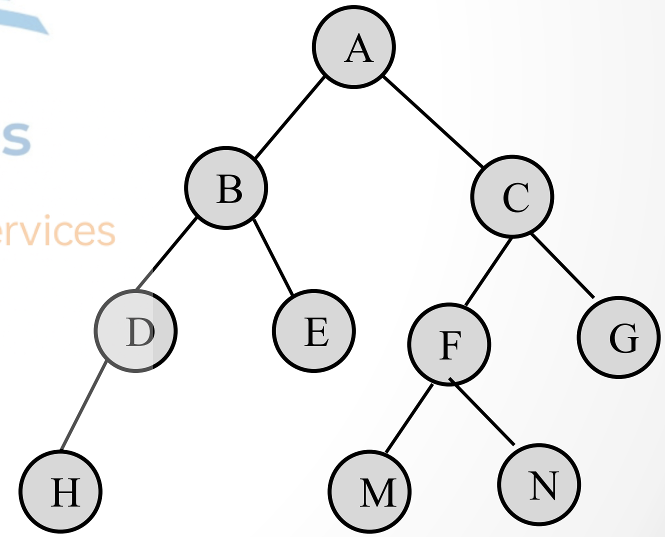
- Root R of T is stored at TREE[1]
- If a node N occupies location TREE[k], its left child is stored at TREE[2*k] location and right child is stored at TREE[2*k+1] location.
- If the depth of tree is ‘d’ then it requires and array of 2^{d+1}-1 elements.

Example:-

TREE[k]

TREE[2*k] TREE[2*k+1]

A	B	C	D	E	F	G	H				M	N		
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15



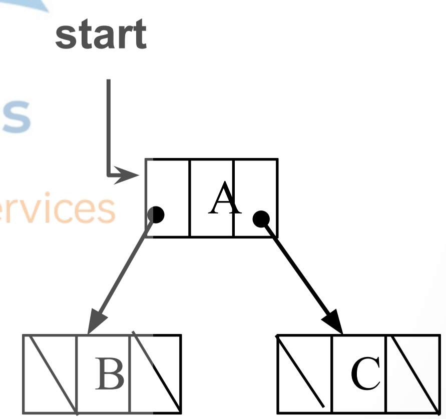
Depth of tree is d=3

Array size needed $2^{d+1}-1= 2^{3+1}-1=15$

Linked representation

- A binary tree is represented by links using linked lists
- Every node in the linked list has tree fields
  - ✔ Value of element
  - ✔ Address of left child
  - ✔ Address of right child

```
struct node
{
  char data;
  struct node *lchild;
  struct node *rchild;
}
typedef struct node btree;
```



Tree Traversals

Traversal is a process to visit all the nodes of the tree.

Binary tree traversal is of three types:

- Preorder Traversal
- Inorder Traversal
- Postorder Traversal

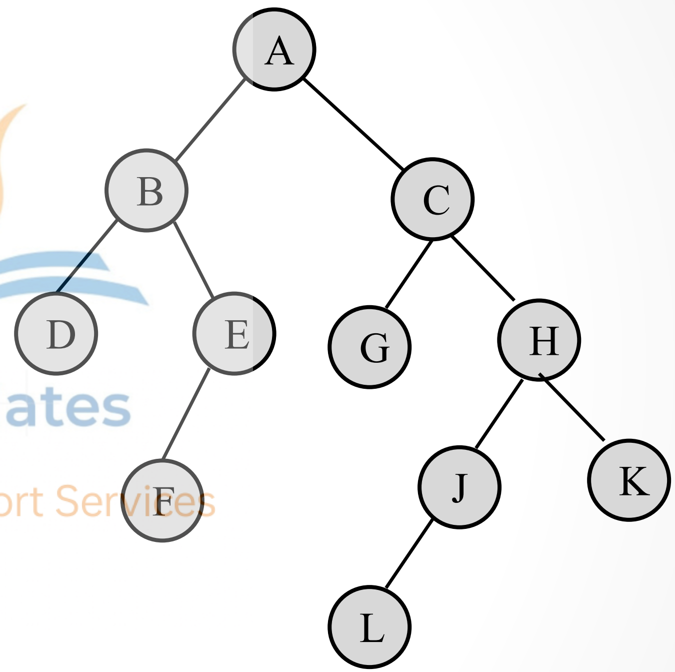
Preorder Traversals

In preorder traversal, a node is visited before its descendants.

1. Visit the root
2. Traverse the left subtree
3. Traverse the right subtree

```
void preorder(btree *root)
{
  if (root !=NULL)
  {
    printf("%d",root->data);
    preorder(root->lchild);
    preorder(root->rchild);
  }
}
```

For example:-



Preorder Traversal

A B D E F C G H J L K

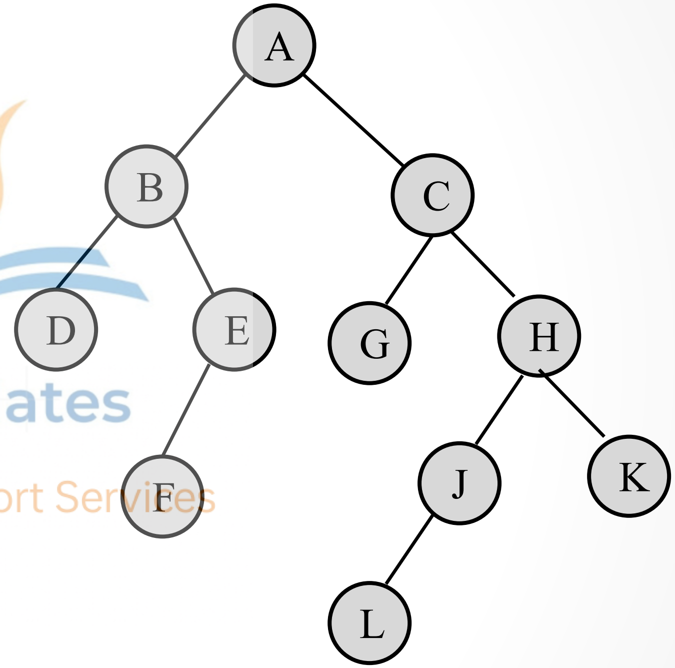
Inorder Traversals

In inorder traversal, a node is visited before its descendants.

1. Traverse the left subtree
2. Visit the root
3. Traverse the right subtree

```
void inorder(btree *root)
{
  if (root !=NULL)
  {
    inorder(root->lchild);
    printf("%d",root->data);
    inorder(root->rchild);
  }
}
```

For example:-



<u>Inorder Traversal</u>

D B F E A G C L J H K

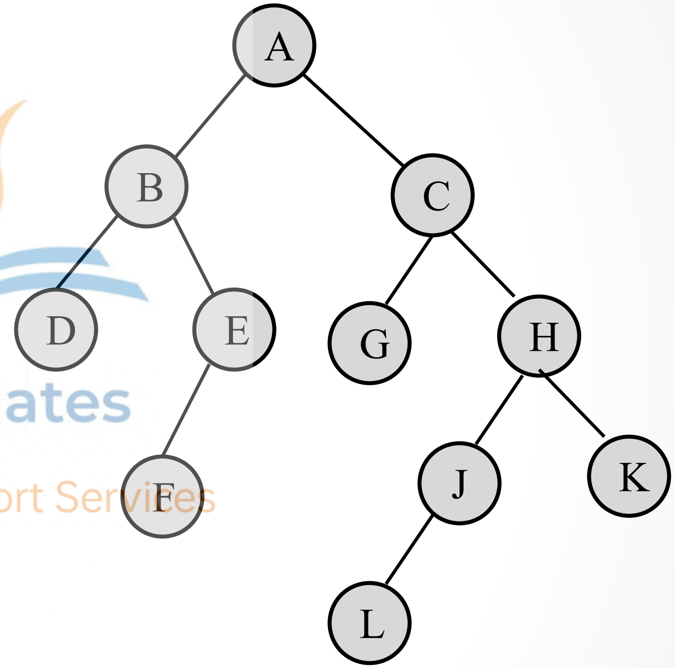
Postorder Traversals

In postorder traversal, a node is visited before its descendants.

1. Traverse the left subtree
2. Traverse the right subtree
3. Visit the root

```
void postorder(btree *root)
{
  if (root !=NULL)
  {
    postorder(root->lchild);
    postorder(root->rchild);
    printf("%d", root->data);
  }
}
```

For example:-



Postorder Traversal

D F E B G L J K H C A

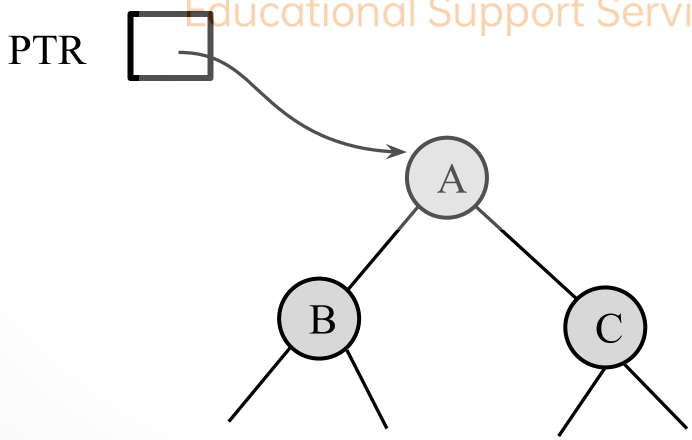
Non Recursive Traversal Algorithms (using Stacks)

Preorder Traversal

Push NULL onto STACK, set PTR=ROOT. Then repeat steps until PTR<>NULL.

(a) Proceed down the left most path rooted at PTR. Process each node in the path and pushing right child R(N), if any, on to the STACK. The traversing ends after a node with no left child L(N) is processed. (PTR is updated with PTR=LEFT[PTR] and traversing stops when LEFT[PTR]=NULL.

(b) [Backtracking] Pop and assign PTR to the top element on the STACK. If PTR<>NULL, then return to step (a); otherwise exit.



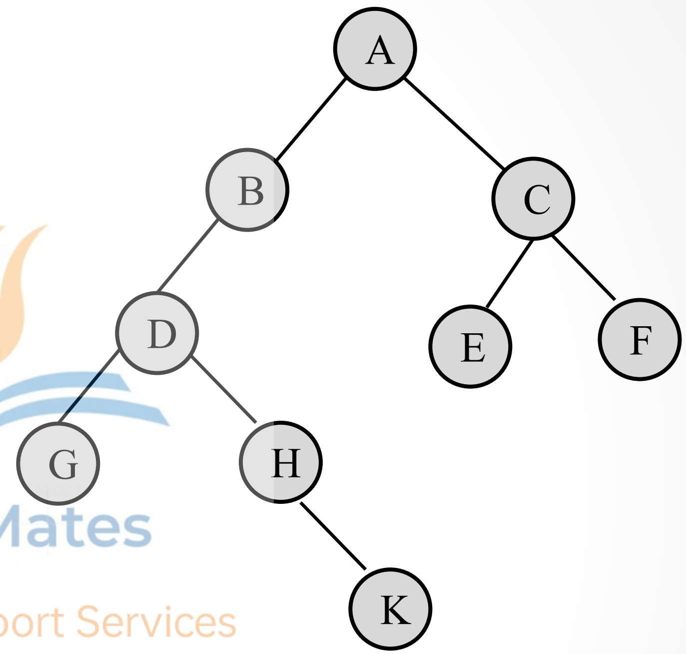
Non Recursive Preorder Algorithm

PREORD(INFO, LEFT, RIGHT, ROOT)

1. [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 to STACK]
   TOP=TOP+1 and STACK[TOP]=RIGHT[PTR]
5. [Left child?]
   If LEFT[PTR]<>NULL then
   Set PTR=LEFT[PTR]
   Else [Pop from STACK]
   LEFT[PTR] =STACK[TOP] and TOP=TOP-1
6. Exit.

Example

1. STACK=Ø, PTR=A
2. Proceed left path
   (i) Process A, Push C onto STACK
   STACK=Ø, C
   (ii) Process B, Process D, Push H onto STACK
   STACK=Ø, C, H
   (iii) Process G. [No left child]
3. [Backtrack] Pop H, **STACK=Ø**, C
4. Proceed left path
   (i) Process H, Push K onto STACK
   STACK=Ø, C, K [No left child]
5. [Backtrack] Pop K, **STACK=Ø**, C
6. Proceed left path
   (i) Process K [No left child]
8. [Backtrack] Pop C, **STACK=Ø**
9. Proceed left path rooted at C
   (i) Process C, Push F onto STACK
   STACK=Ø, F
   (ii) Process E [No left child]
10. [Backtrack] Pop F, **STACK=Ø**
11. Proceed left path
    (i) Process F [No left child]
12. [Backtrack] Pop top element=NULL
    Algo completed as PTR=NULL



Preorder Traversal

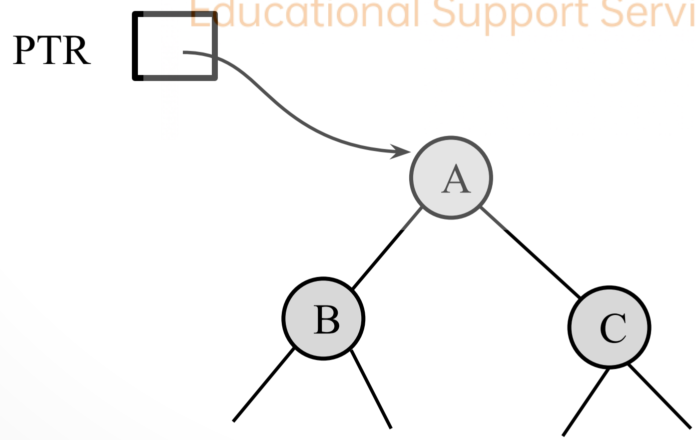
A B D G H K C E F

Inorder Traversal

Push NULL onto STACK, set PTR=ROOT. Then repeat steps until PTR<>NULL.

(a) Proceed down the left most path rooted at PTR. Push each node N onto STACK and stopping when a node N with no left child L(N) is pushed onto STACK.

(b) [Backtracking] Pop and process the nodes on STACK. If NULL is popped, then exit. If a node N with a right child R(N) is processed, set PTR=R(N) and return to step (a).



Non Recursive Inorder Algorithm

INORD(INFO, LEFT, RIGHT, ROOT)

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
   (b) Set PTR=LEFT[PTR]
3. Set PTR=STACK[TOP] and TOP=TOP-1
4. Repeat steps 5 to 7 while PTR<>NULL [Backtrack]
5. Apply Process to INFO[PTR]
6. [Right child?]
   If RIGHT[PTR]<>NULL then
   (a) Set PTR=RIGHT[PTR]
   (b) Go to step 2
7. Set PTR=STACK[TOP] and TOP=TOP-1 [Pops Node]
   [End of step 4 loop]
8. Exit.

Example

1. STACK=Ø, PTR=A
2. Proceed left path. Push A, B, D, G, K onto STACK. [K has no left child]

   STACK=Ø, A, B, D, G, K

3. [Backtrack] Pop K, G, D. D has right child, set PTR=H

   STACK=Ø, A, B

4. Proceed left path

   Push H, L onto STACK

   STACK=Ø, A, B, H, L [L has no left child]

5. [Backtrack] Pop L, H. H has right child, set PTR=M

   STACK=Ø, A, B

6. Proceed left path

   Push M onto STACK

   STACK=Ø, A, B, M [M has no left child]

7. [Backtrack] Pop M, B, A. A has right child, set PTR=C

   STACK=Ø

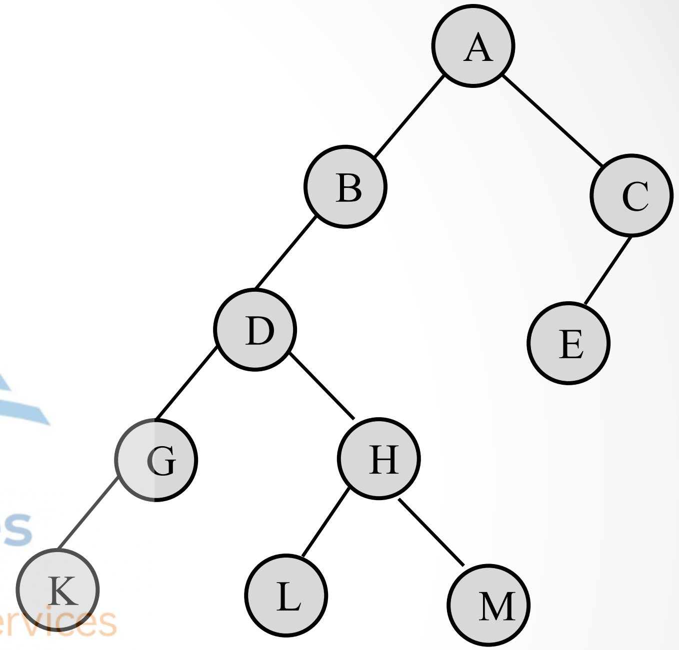
8. Proceed left path

   Pushing C, E onto STACK

   STACK=Ø, C, E [E has no left child]

9. [Backtrack] Pop E, C and process.

   Finally NULL popped out.



<u>Inorder Traversal</u>

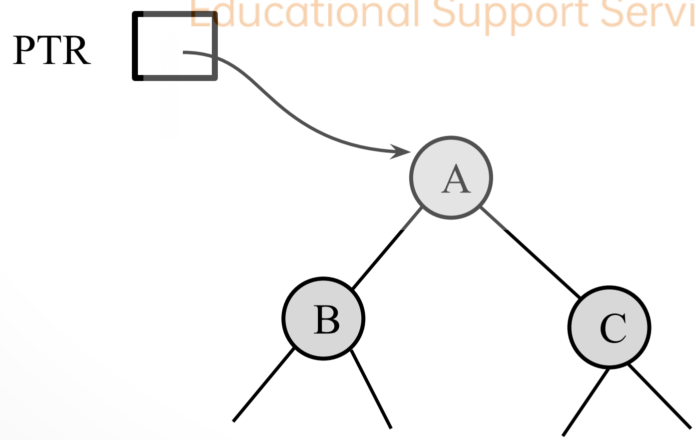
K G D L H M B A E C

Postorder Traversal

Push NULL onto STACK, set PTR=ROOT. Then repeat steps until PTR<>NULL.

(a) Proceed down the left most path rooted at PTR. At each node N of path, push N onto the STACK and if N has right child R(N), push –R(N) onto STACK.

(b) [Backtracking] Pop and process the positive nodes on STACK. If NULL is popped, then exit. If a negative node N is popped i.e. if PTR=-N for some node N, set PTR=N (by assigning PTR= - PTR) and return to step (a).



Non Recursive Postorder Algorithm

POSTORD(INFO, LEFT, RIGHT, ROOT)

1. [Push NULL onto STACK and initialize PTR]
   Set TOP=1, STACK[1]=NULL and PTR=ROOT
2. Repeat step 3 to 5 while PTR <> NULL [Push left-most path onto STACK]
3. Set TOP=TOP+1 and STACK[TOP]=PTR
4. If RIGHT[PTR]<>NULL then
   Set TOP=TOP+1 and STACK[TOP]= -RIGHT[PTR]
   [End of if structure]
5. Set PTR = LEFT[PTR] [Update pointer PTR]
6. Set PTR= STACK[TOP] and TOP=TOP-1 [Pops node from STACK]
7. Repeat while PTR>0
   Apply PROCESS TO INFO[PTR]
   Set PTR= STACK[TOP] and TOP= TOP-1 [Pops node from STACK]
8. If PTR<0 then
   Set PTR= -PTR, Go to step 2
9. Exit.

Example

1. STACK=Ø, PTR=A
2. Proceed left path. Push A onto STACK. A has a right child, so push –C onto STACK

   STACK=Ø, A, -C

   Proceeding left path push B, D, -H, G, K

   STACK=Ø, A, -C, B, D, -H, G, K [K has no left child]

3. [Backtrack] Pop K, G, -H. [node –H is popped] set PTR=-(-H)

   STACK=Ø, A, -C, B, D

4. Proceed left path

   Push H, -M, L onto STACK

   STACK=Ø, A, -C, B, D, H, -M, L [L has no left child]

5. [Backtrack] Pop L, -M. [–M is popped] set PTR=-(-M)

   STACK=Ø, A, -C, B, D, H

6. Proceed left path

   Push M onto STACK

   STACK=Ø, A, -C, B, D, H, M [M has no left child]

7. [Backtrack] Pop M, H, D, B, -C. [–C is popped] set PTR=-(-C)

   STACK=Ø, A

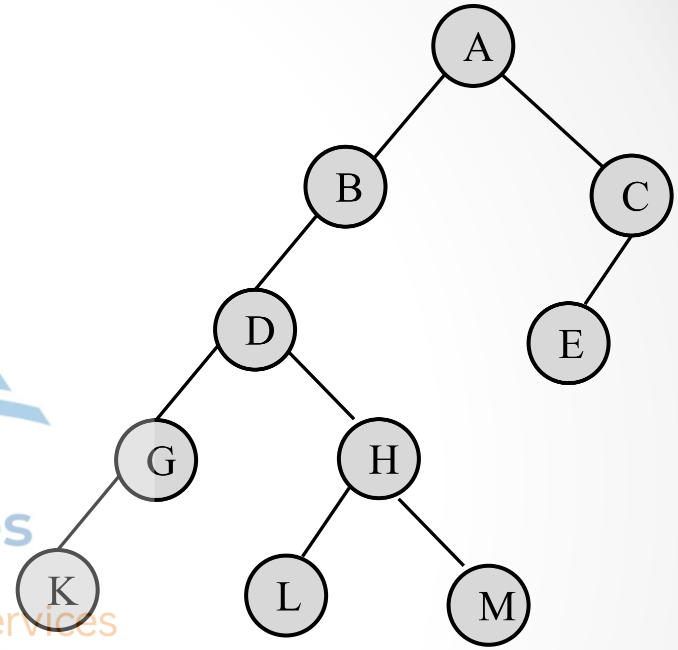
8. Proceed left path

   Pushing C, E onto STACK

   STACK=Ø, A, C, E [E has no left child]

9. [Backtrack] Pop E, C, A and process.

   Finally NULL popped out.

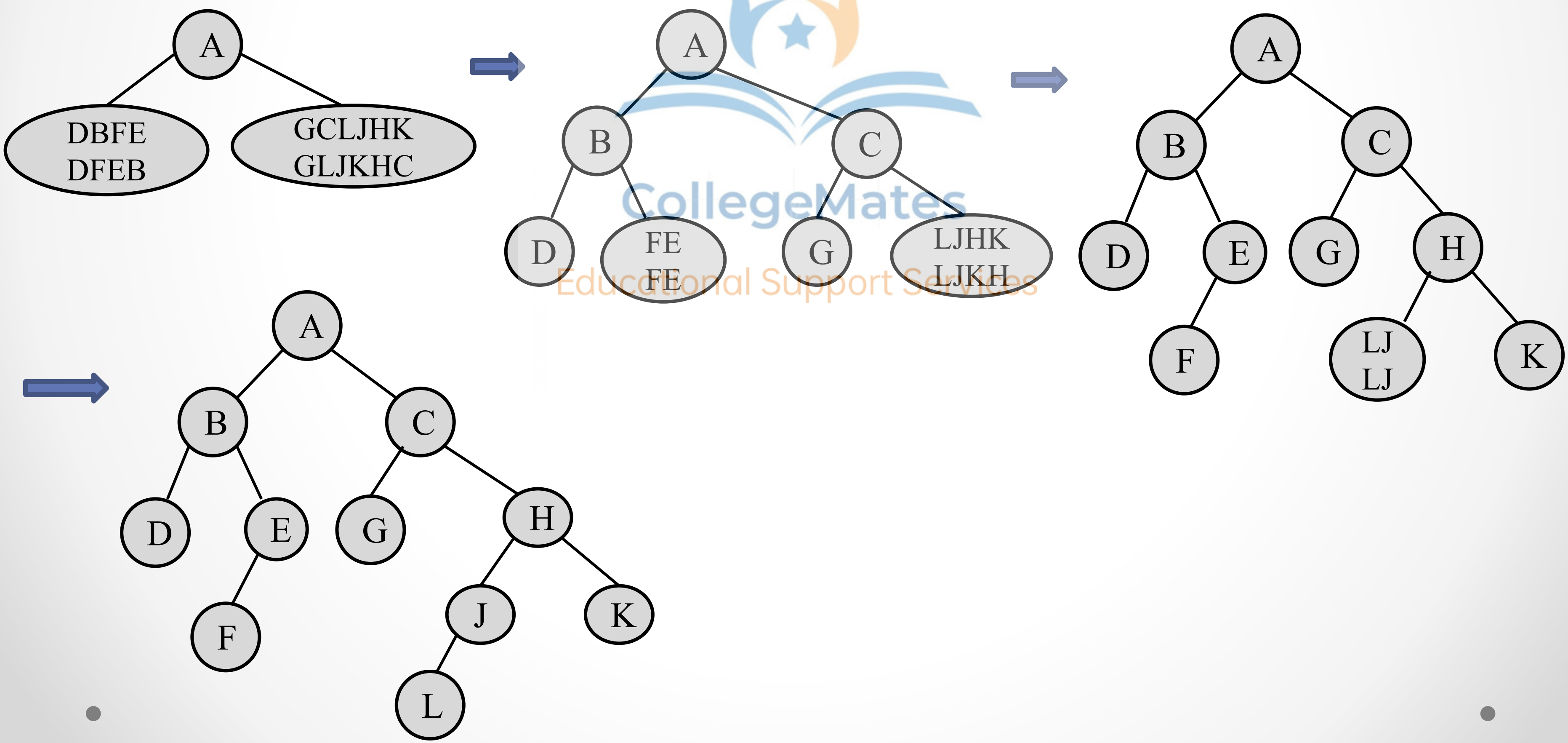


<u>Postorder Traversal</u>

K G L M H D B E C A

Build a Binary Tree

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



Build a Binary Tree

In:	4	7	2	8	5	1	6	9	3
Pre:	1	2	4	7	5	8	3	6	9

Build a Binary Tree

In:	E	A	C	K	F	H	D	B	G
Pre:	F	A	E	K	C	D	H	G	B

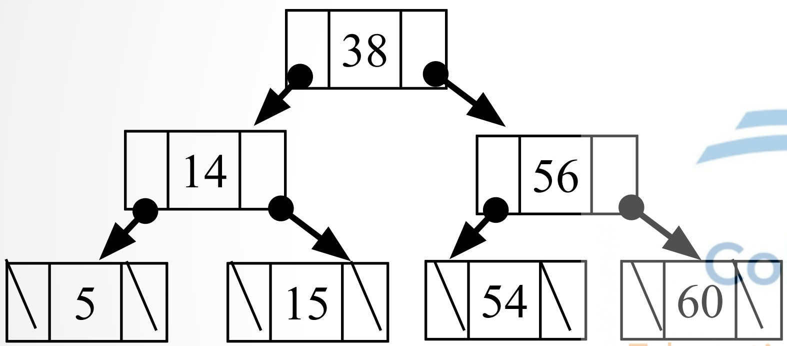
Build a Binary Tree

In:	1	5	7	10	15	20	25	30	32	35	40
Pre:	20	10	5	1	7	15	30	25	35	32	40

Threaded Binary Tree

In a linked representation of a binary tree, the number of null links (null pointers) are actually more than non-null pointers.

Consider the following binary tree:

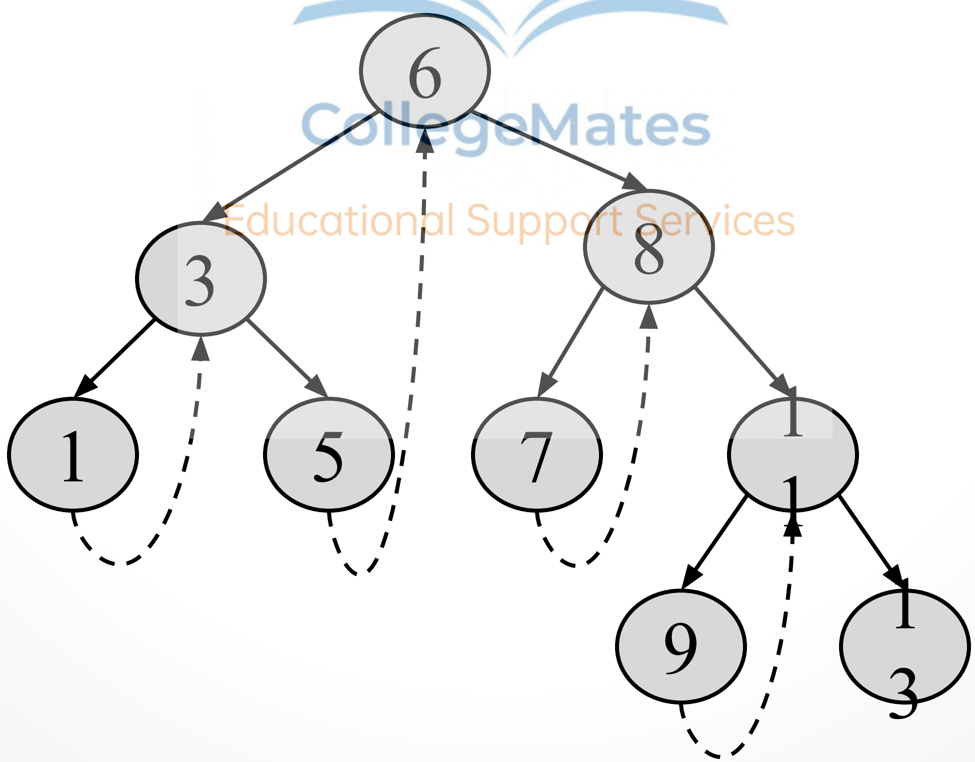


- In this binary tree, there are 14 pointers and out of them 8 are null pointers.
- We can generalize it that for any binary tree with n nodes there will be 2n total pointers and (n+1) null pointers.
- The objective here to make effective use of these null pointers.
- A. J. perils & C. Thornton jointly proposed idea to make effective use of these null pointers.
- According to this idea we are going to replace all the null pointers by the appropriate pointer values called threads and such trees are called **Threaded Binary Trees**.

Types of Threading

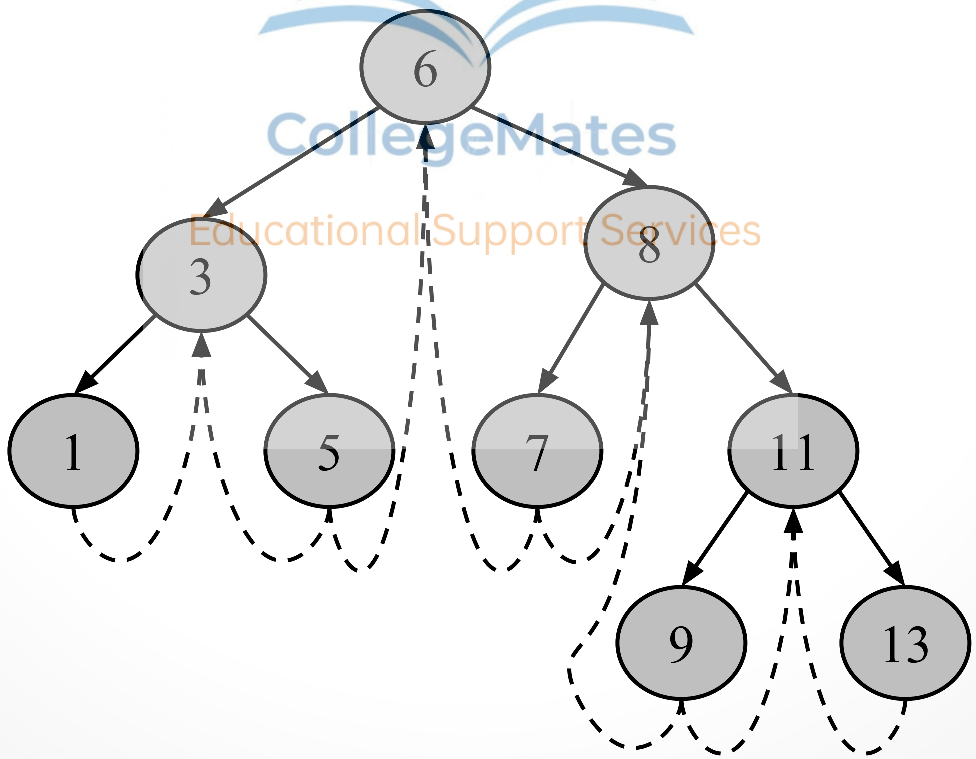
By default, threading corresponds to **inorder traversal**.

- One-way Threading
  - In this a thread appears in RIGHT child’s address field of a node and points to the **next node in the inorder traversal** of the tree
  - The RIGHT pointer of last node contains NULL.



Types of Threading

- Two-way Threading
  - In this, in addition to right field, a thread will also appear in the LEFT child's address field of a node and will point to the **preceding node in the inorder traversal** of the tree.
  - The LEFT pointer of the first node contains NULL.



- In the memory representation of a threaded binary tree, it is necessary to distinguish between a normal pointer and a thread.
- Therefore we have an alternate node representation for a threaded binary tree which contains five fields as show below:

lthread	lchild	data	rchild	rthread

- To differentiate between a child address pointer and a thread, two more Boolean fields are added in node structure. If lchild has a left child's address pointer, then it is '0' and if it is a thread it has value '1'. Same as with the right side.

Comparison of Threaded BT

Threaded Binary Trees

- In threaded binary trees, The null pointers are used as thread.
- We can use the null pointers which is a efficient way to use computers memory.
- Traversal is easy. Completed without using stack or reccursive function.
- Structure is complex.
- Insertion and deletion takes more time.

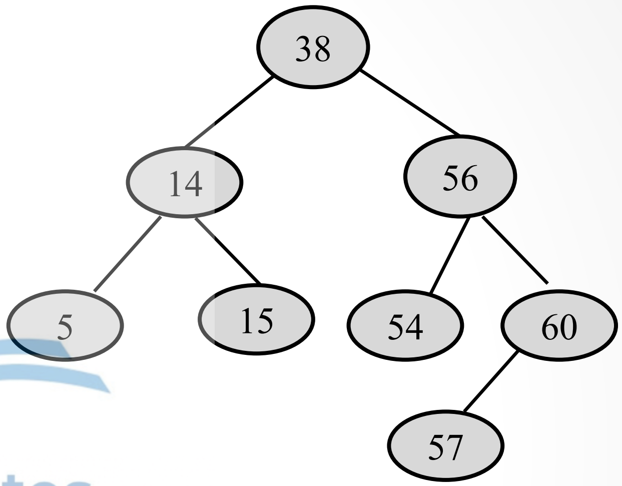
Normal Binary Trees

- In a normal binary trees, the null pointers remains null.
- We can't use null pointers so it is a wastage of memory.
- Traverse is not easy and not memory efficient.
- Less complex than Threaded binary tree.
- Less Time consuming than Threaded Binary tree.

Binary Search Tree (BST)

In Binary Search Tree each node has this property:

The value stored at a node N is greater than every value stored in its left subtree and less than the every value stored in its right subtree.



Inorder Traversal

5 14 15 38 54 56 57 60

If this BST is traversed in inorder, it produces a sorted list.

Searching an element is very easy as compared to linked lists.

Searching in a BST

Searching an element is very easy. Let ITEM is the element to be searched.

(a) Compare ITEM with root node N of tree

- If ITEM < N then proceed to left subtree
- If ITEM > N then proceed to right subtree

(b) Repeat (a) until one of the following occurs:

- We meet a node N such that ITEM=N, successful
- We meet empty subtree, unsuccessful.

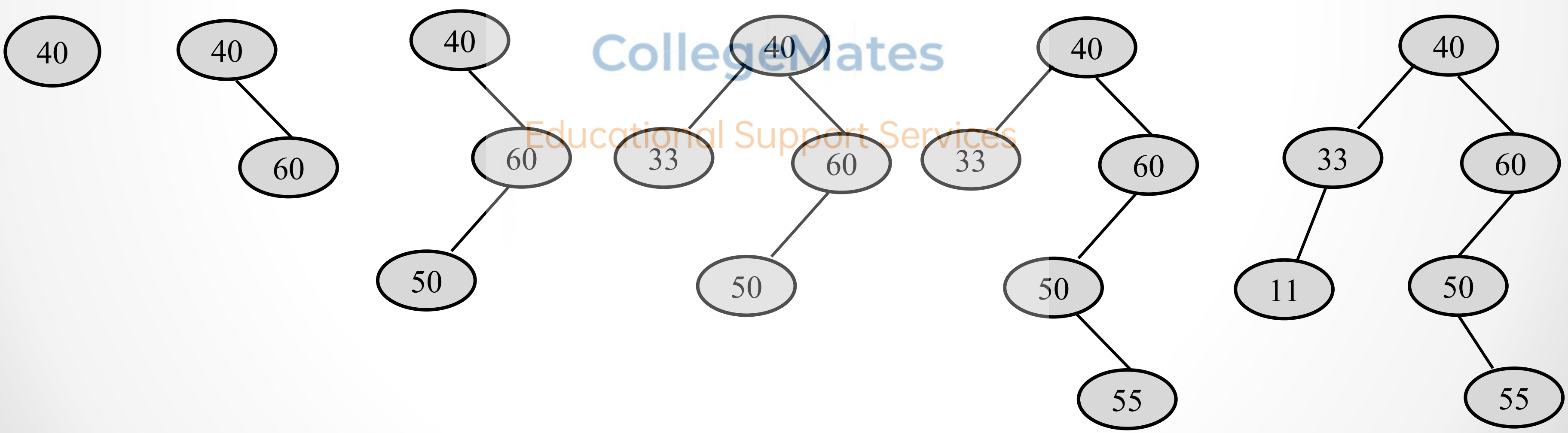
Building a BST

(a) Search the element ITEM on the BST.

(b) If we meet a node N such that ITEM=N, then ITEM already available on BST and return.

(c) Else insert ITEM at appropriate location

Array T: 40 60 50 33 55 11



(i) Insert 40 | (ii) Insert 60 | (iii) Insert 50 | (iv) Insert 33 | (v) Insert 55 | (vi) Insert 11

Algorithm for finding location of a given item

FIND(INFO, LEFT, RIGHT, ITEM, LOC, PAR)

1. If ROOT=NULL, Set LOC=NULL & PAR=NULL and Return

   [Item at ROOT?]

2. If ITEM=INFO[ROOT], then Set LOC=ROOT, PAR=NULL and return
3. If ITEM<INFO[ROOT] then

      Set PTR=LEFT[ROOT] and SAVE=ROOT

   Else

      Set PTR=RIGHT[ROOT] and SAVE=ROOT

4. Repeat 5 and 6 while PTR<>NULL
5. If ITEM=INFO[PTR], then Set LOC=PTR, PAR=SAVE and return
6. If ITEM<INFO[PTR] then

      Set SAVE=PTR AND PTR=LEFT[PTR]

   Else

      Set SAVE=PTR AND PTR=RIGHT[PTR]

7. Set LOC=NULL and PAR=SAVE (Search Unsuccessful)
8. Exit

Algorithm for inserting an item in BST

INSBST(INFO, LEFT, RIGHT, ROOT, AVAIL, ITEM, LOC)

1. Call FIND(INFO, LEFT, RIGHT, ITEM, LOC, PAR)
2. If LOC<>NULL, then exit.
3.(a) If AVAIL=NULL, write overflow and exit.
   (b) Set NEW=AVAIL, AVAIL=LEFT[AVAIL] and INFO[NEW]=ITEM.
   (c) Set LOC=NEW, LEFT[NEW]=NULL, RIGHT[NEW]=NULL.
4. If PAR=NULL then Set ROOT=NEW
   Else if ITEM<INFO[PAR] then Set LEFT[PAR]=NEW
   Else Set RIGHT[PAR]=NEW
5. Exit

Build BST

1. 45, 36, 76, 23, 89, 115, 98, 39, 41, 56, 69, 48

2. 44, 30, 50, 22, 60, 55, 77, 55

3. 10, 11, 23, 43, 55, 66, 89, 99, 201

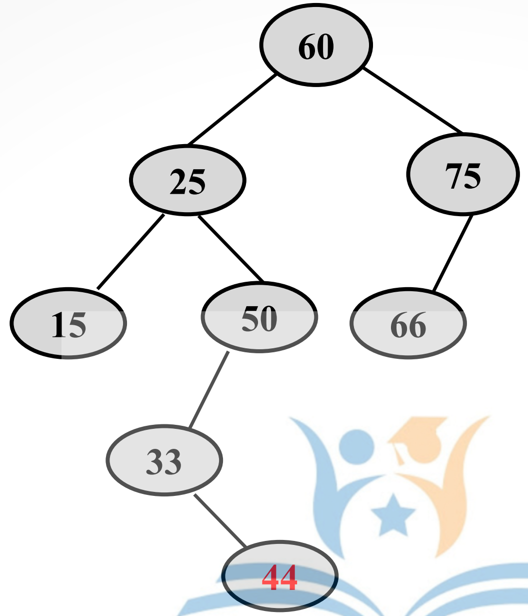
4. 99, 44, 42, 22, 13, 10, 9, 6, 2, 1

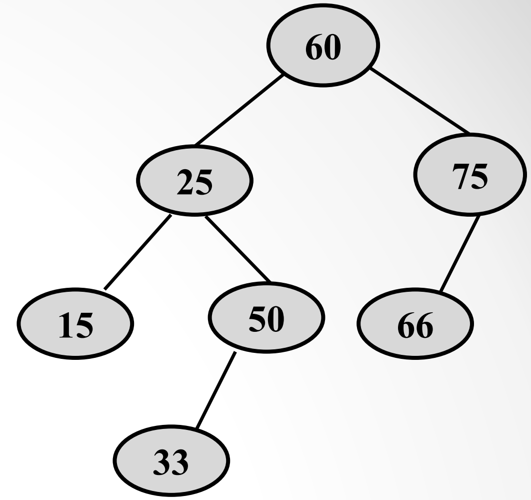
Deletion in Binary Search Tree

Three cases of deletion of a node N from a BST:-

- N has no child - N is deleted from the tree T by simply replacing location of N in its parent P(N) as NULL and free this space
- N has exactly one child - N is deleted from the tree T by simply replacing location of N in its parent P(N) by the location of only child of N
- N has two child - N, the node to be deleted from the tree T, is replaced by its inorder successor S(N) (S(N) does not have a left child)

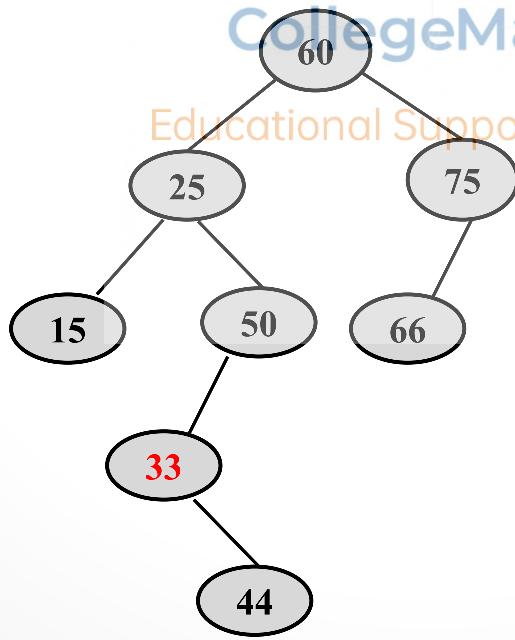
Deletion of a node N with no child

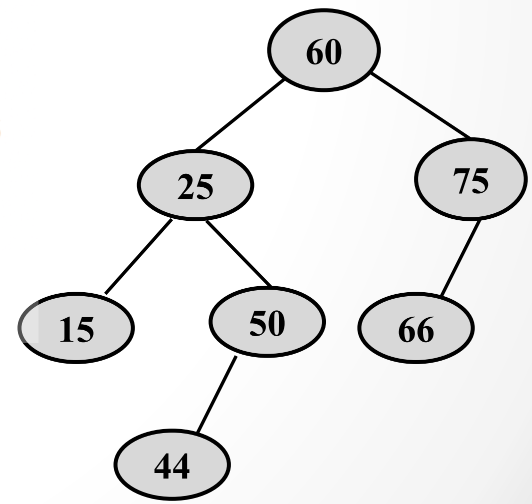




(After deletion of 44)

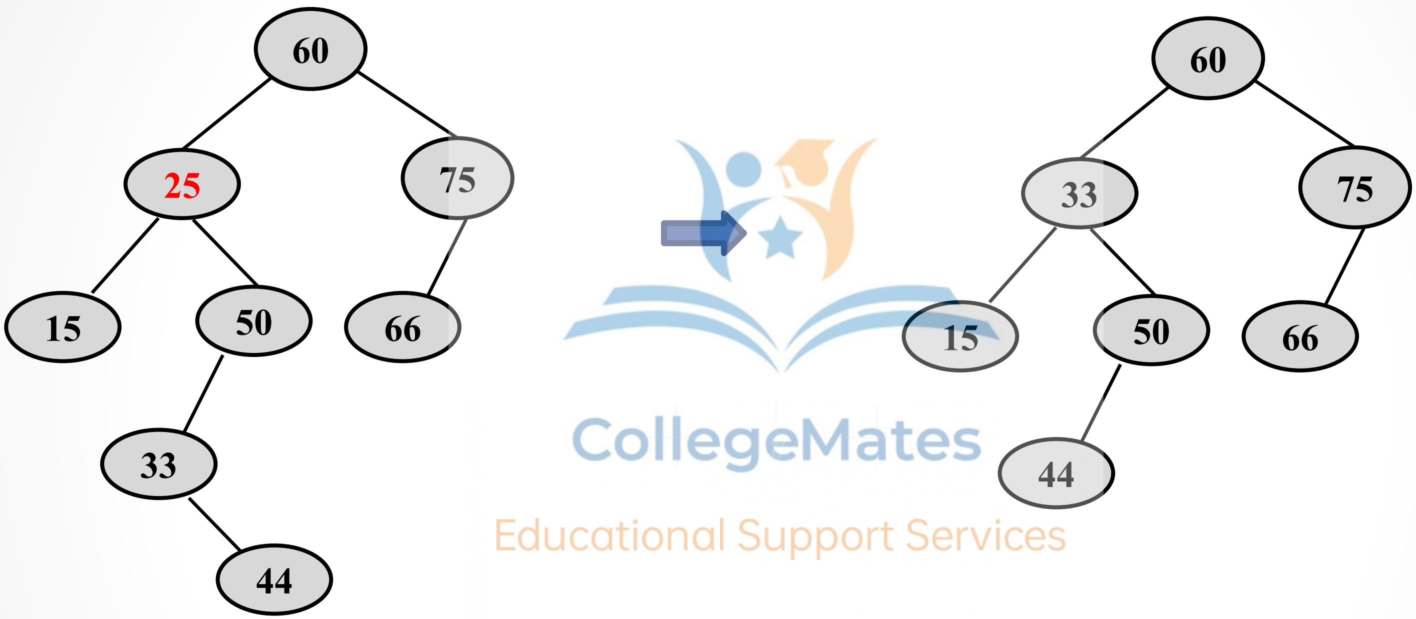
Deletion of a node N with single child





(After deletion of 33)

Deletion of a node N with two children



(After deletion of 25)

Inorder Traversal: 15, 25, 33, 44, 50, 60, 66, 75

Algorithm to delete an element `ITEM'

DEL(INFO, LEFT, RIGHT, ROOT, AVAIL, ITEM)

1. [Find the location of ITEM and its parent]
   Call FIND(INFO, LEFT, RIGHT, ITEM, LOC, PAR)
2. [ITEM not in tree]
   If LOC=NULL, then ITEM not in the tree, Exit.
3. [Delete node containing ITEM]
   If RIGHT[LOC]<>NULL and LEFT[LOC]<>NULL, then
   Call CaseB(INFO, LEFT, RIGHT, ROOT, LOC, PAR)
   Else
       Call CaseA(INFO, LEFT, RIGHT, ROOT, LOC, PAR)
4. [Return deleted node to AVAIL list]
   Set LEFT[LOC]=AVAIL and AVAIL=LOC
5. EXIT

Algorithm for CaseA

CaseA(INFO, LEFT, RIGHT, ROOT, LOC, PAR)

1. [Initialize CHILD]
   If LEFT[LOC] = NULL and RIGHT[LOC] = NULL, then
       Set CHILD = NULL
   Elseif LEFT[LOC] <> NULL then CHILD = LEFT[LOC]
           else CHILD = RIGHT[LOC]
2. [Delete LOC and move the CHILD at the place of LOC
   If PAR<>NULL then
     If LOC=LEFT[PAR] then Set LEFT[PAR]=CHILD
           else Set RIGHT[PAR]=CHILD
   else SET ROOT=CHILD
3. Return.

Algorithm for CaseB

CaseB(INFO, LEFT, RIGHT, ROOT, LOC, PAR)

1. [Find SUC and PARSUC]
   (a) SET PTR= RIGHT[LOC] and SAVE=LOC
   (b) Repeat while LEFT[PTR]<>NULL
       Set SAVE=PTR and PTR=LEFT[PTR]
   (c) Set SUC=PTR and PARSUC=SAVE
2. [Delete inorder successor]
   Call CaseA(INFO, LEFT, RIGHT, ROOT, SUC, PARSUC)
3. [Replace node N by its inorder successor]
   (a) If PAR<> NULL
       If LOC=LEFT[PAR] then Set LEFT[PAR]= SUC
           else Set RIGHT[PAR]= SUC
     Else
     ROOT=SUC
   (b) Set LEFT[SUC]=LEFT[LOC] and
       RIGHT[SUC]=RIGHT[LOC]
4. Return.

SUC – location of inorder successor
PARSUC – location of parent of inorder successor
PTR is used to traverse the tree and SAVE is used to store the PTR