Semester III,
B. E. Computer Science & Engineering
(Artificial Intelligence and Machine Learning)
Course Code : CAT201
Course : Data Structure
L: 3Hrs, T: 1Hr, P: 0Hr, Per Week Total Credits : 04

Course Objectives

1. To impart to students the basic concepts of data structures and algorithms.
2. To familiarize students on different searching and sorting techniques.
3. To prepare students to use linear (stacks, queues, linked lists) and non-linear (trees, graphs) data structures.
4. To enable students to devise algorithms for solving real-world problems.

Course Outcomes

On completion of the course the student will be able to

1. Recognize different ADTs and their operations and specify their complexities.
2. Design and realize linear data structures (stacks, queues, linked lists) and analyze their computation complexity.
3. Devise different sorting (comparison based, divide-and-conquer, distributive, and tree- based) and searching (linear, binary) methods and analyze their time and space requirements.
4. Design traversal and path finding algorithms for Trees and Graphs.

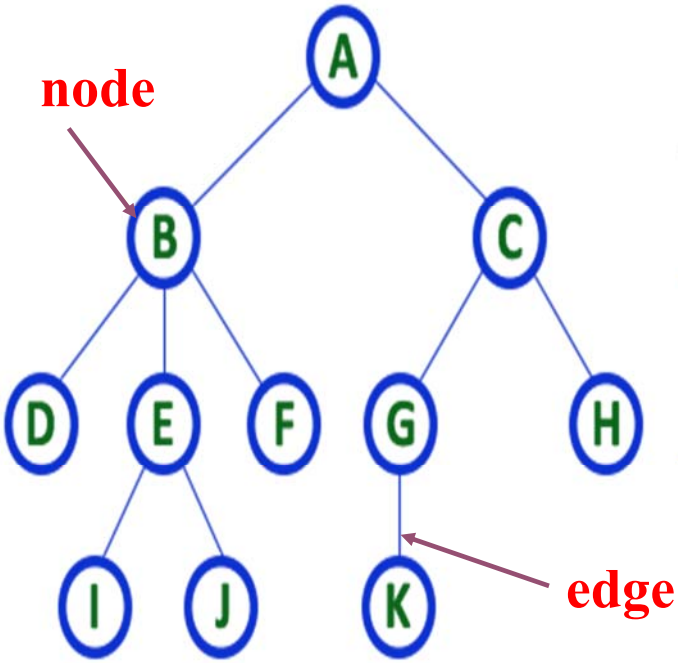
Trees

Trees in real life

Trees in Data Structure

Tree Definition

Tree is a non-linear data structure which organizes data in hierarchical structure.

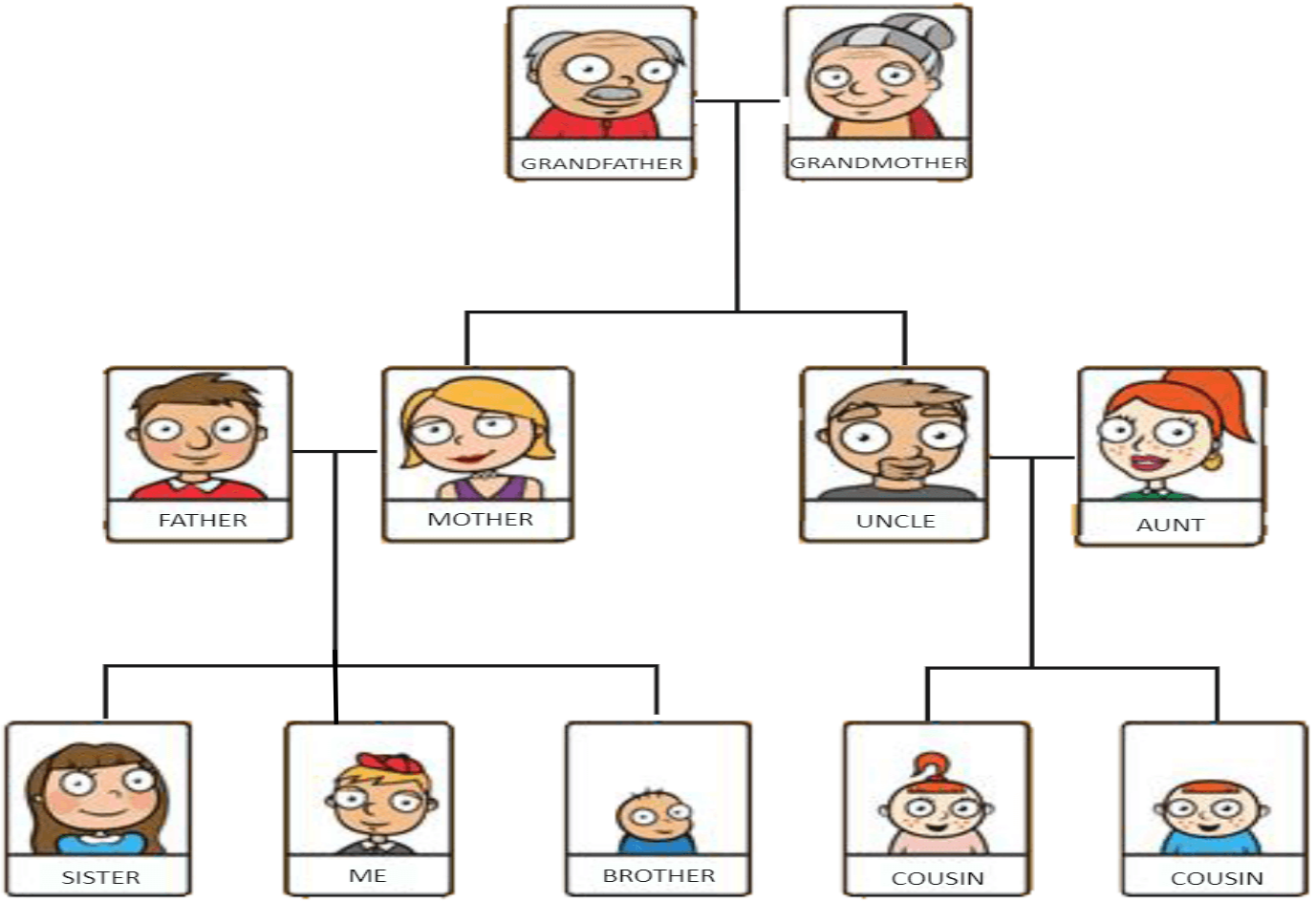


TREE with 11 nodes and 10 edges

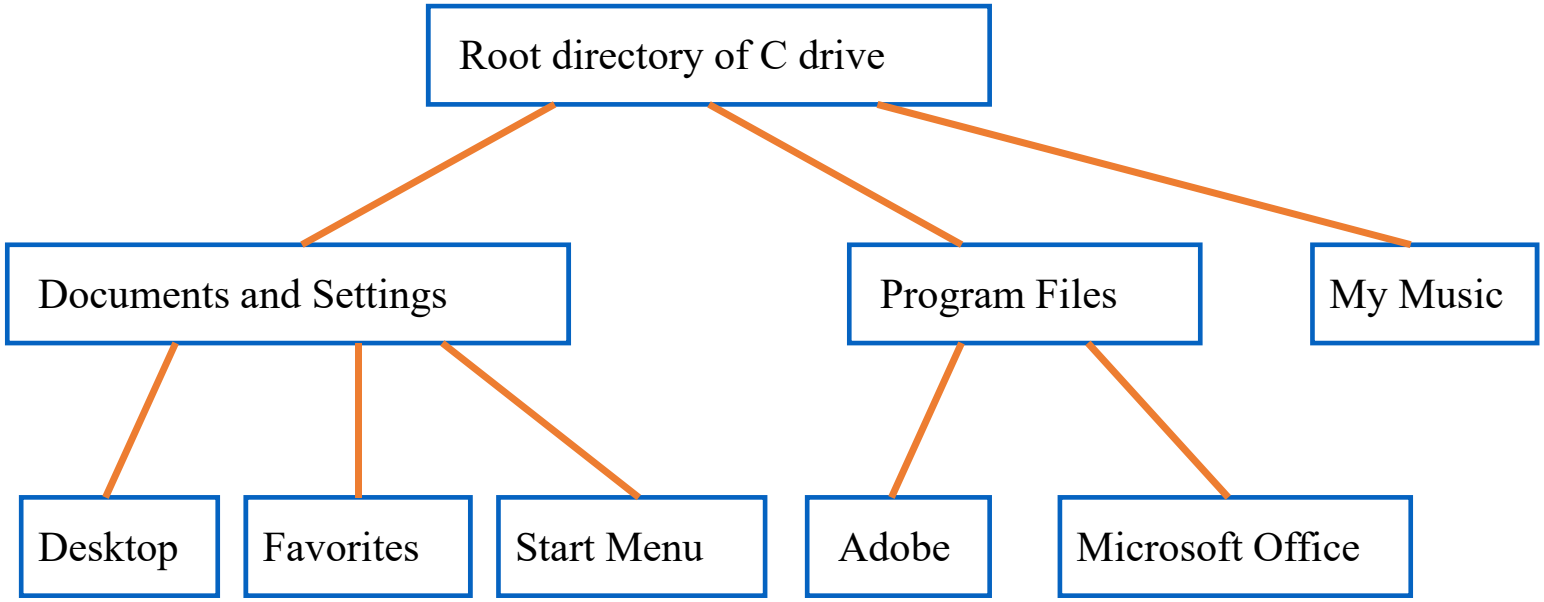
- In any tree with 'N' nodes there will be maximum of 'N-1' edges
- In a tree every individual element is called as 'NODE'

- Trees are used to represent hierarchical relationship
- Each tree consists of nodes and edges
- Each node represents an object
- Each edge represents the relationship between two nodes.

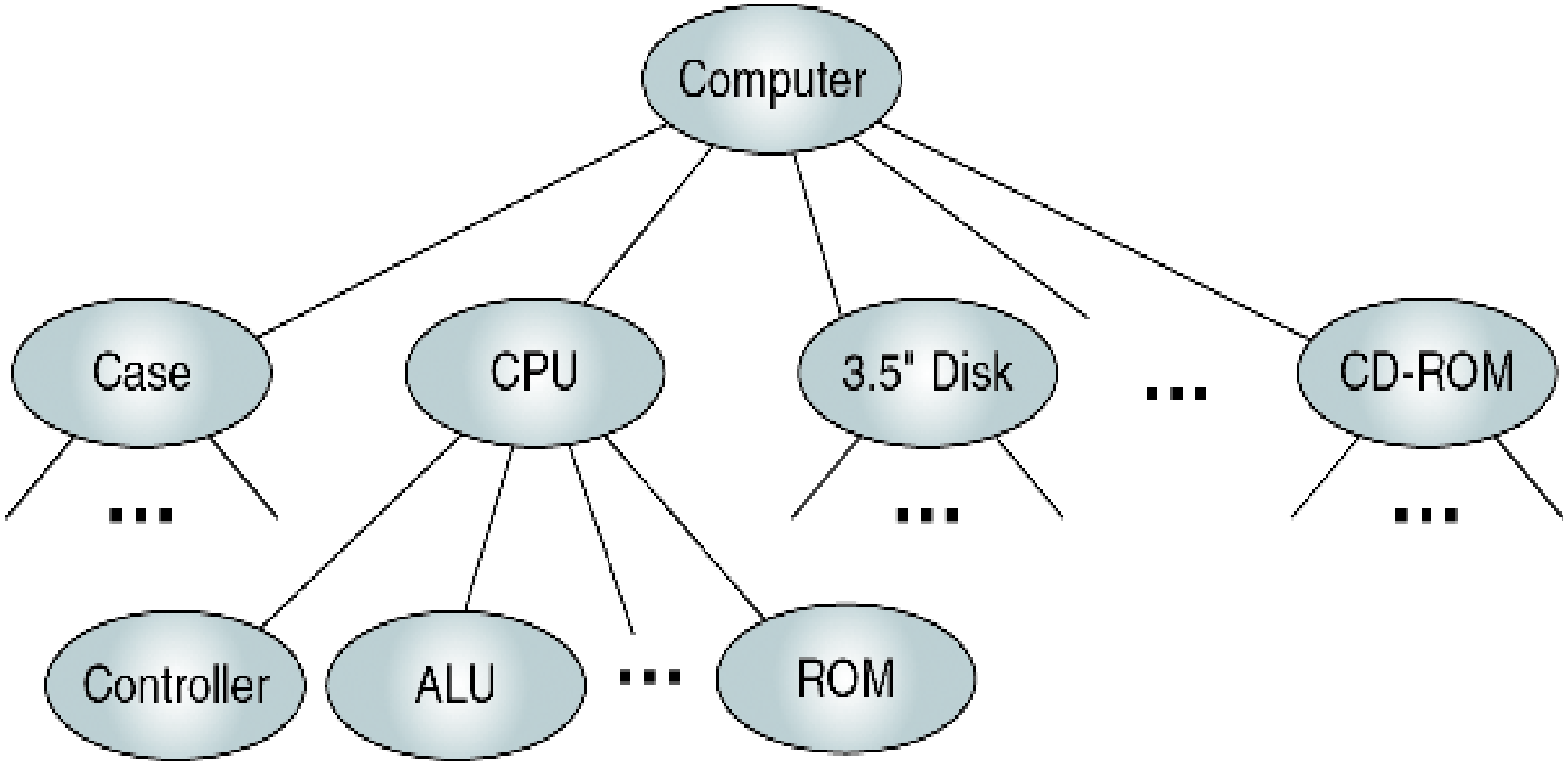
Example: Descendant Tree



Example: Computer File System

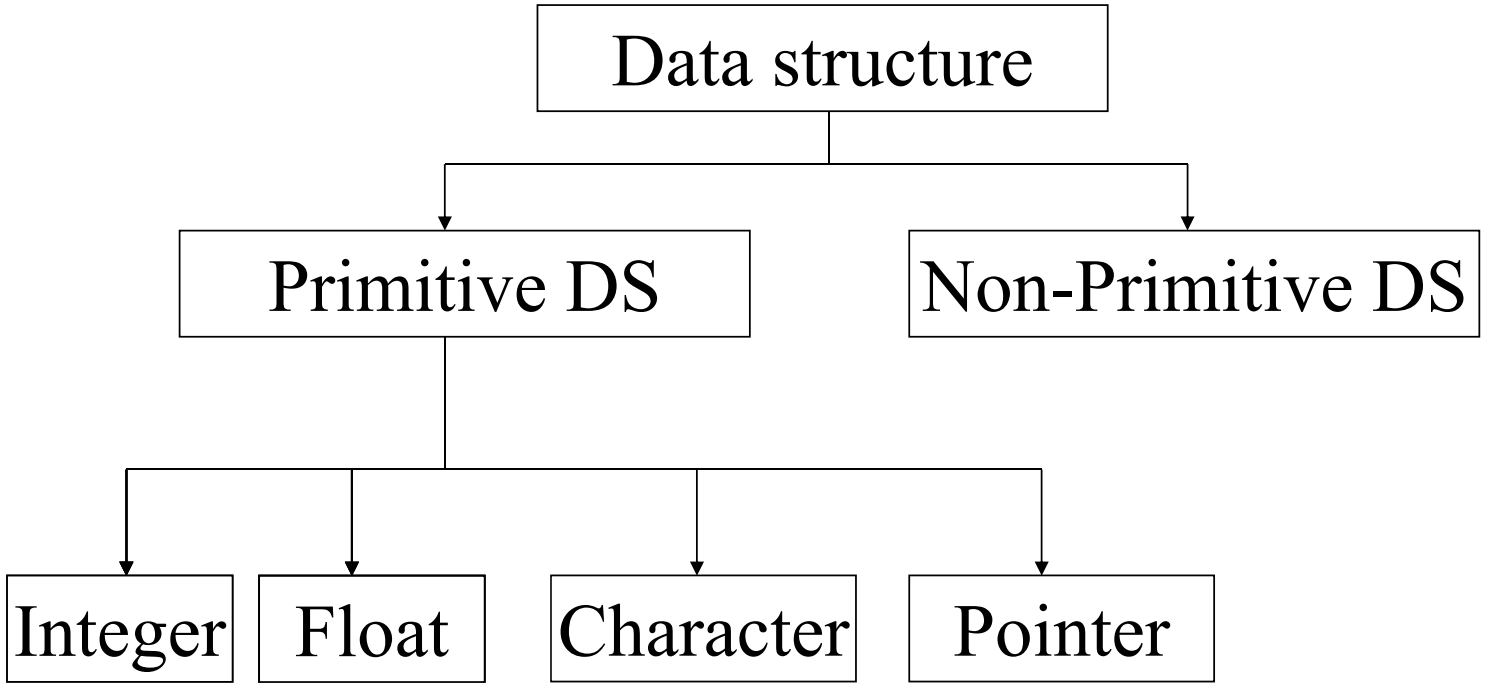


Example: Computer Parts

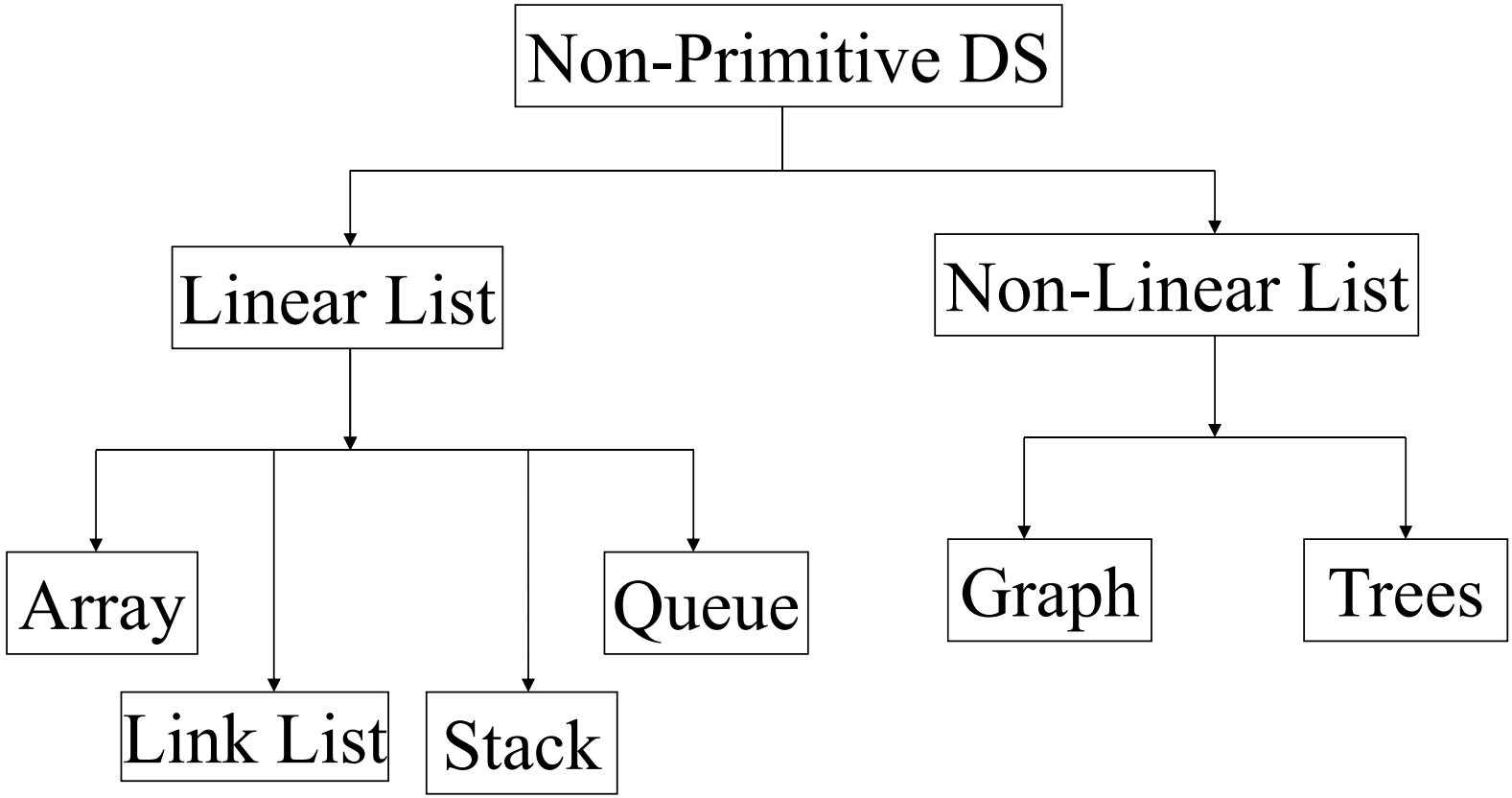


Computer Parts List as a General Tree

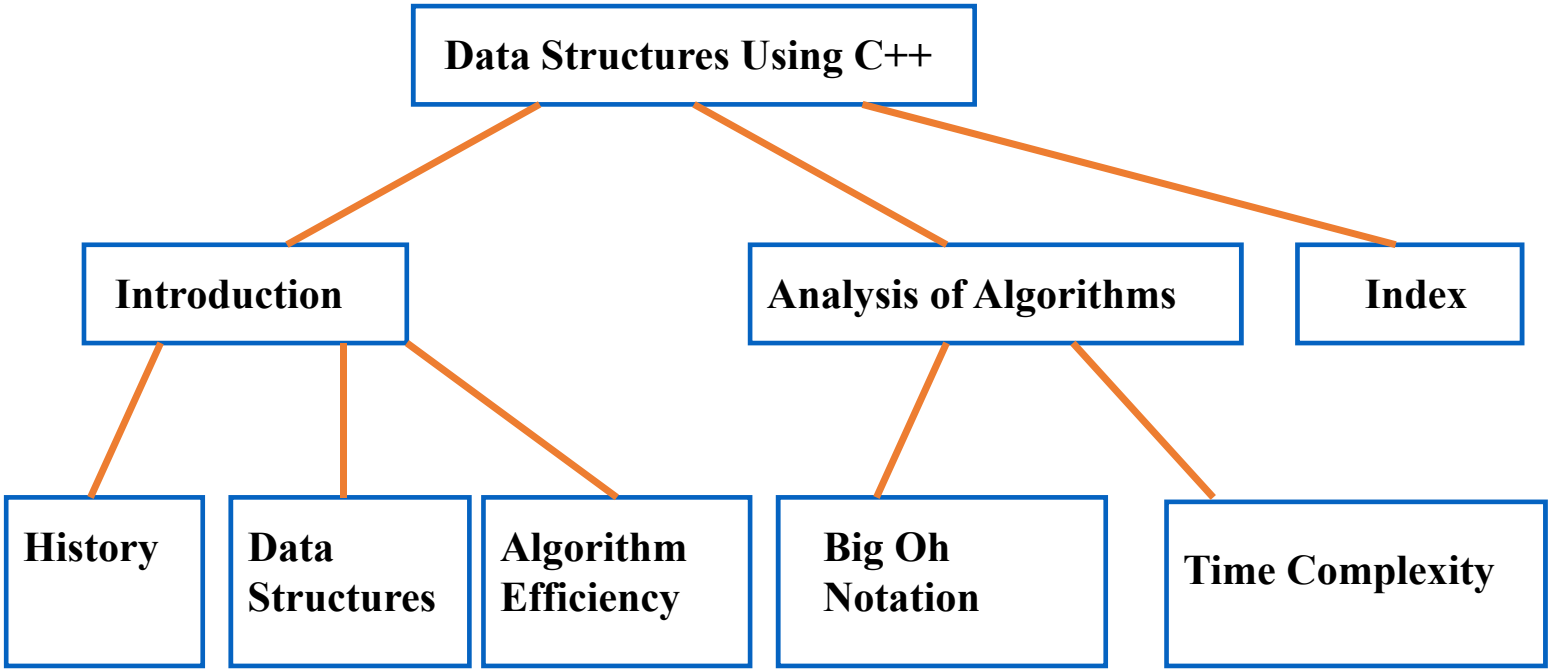
Example: Classification of Data Structure



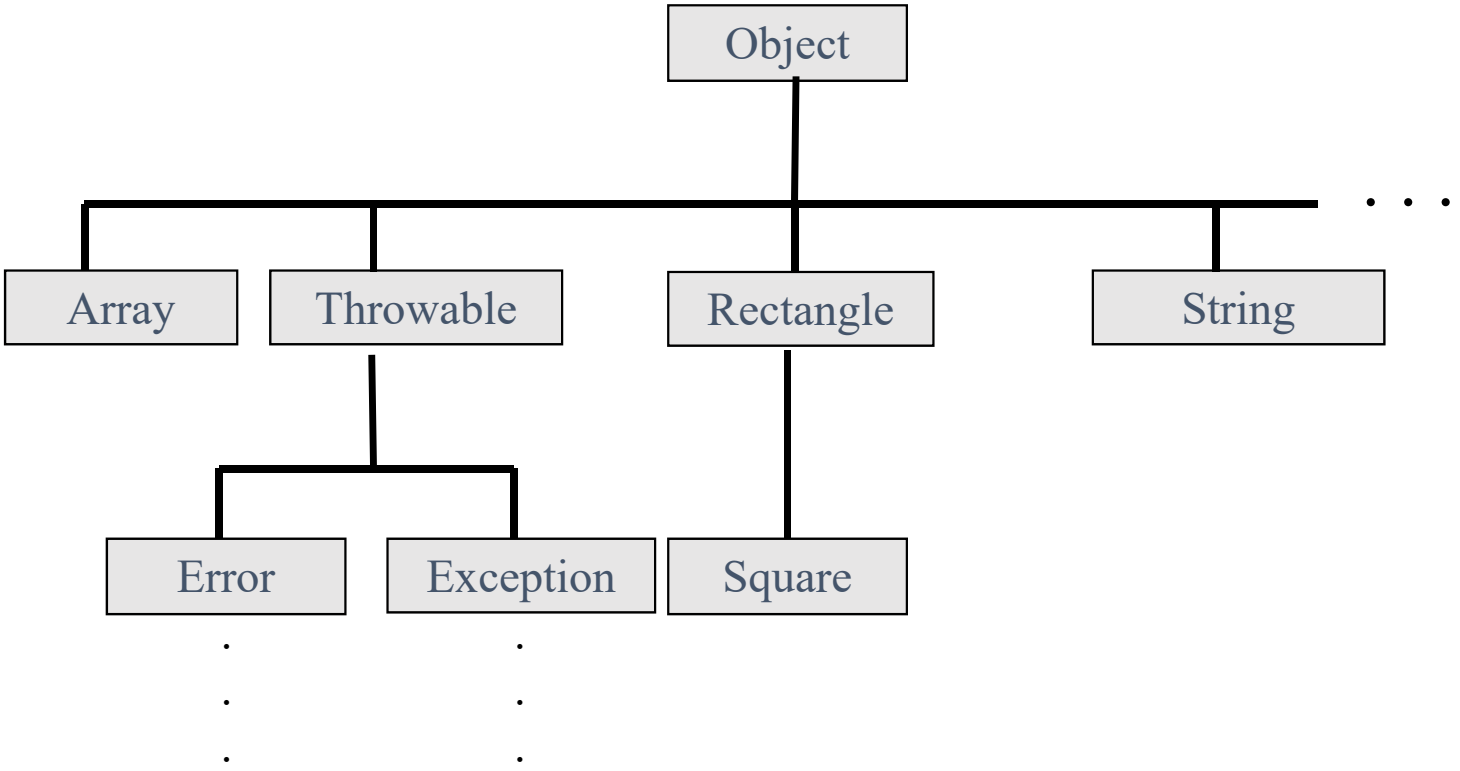
Example: Classification of Data Structure



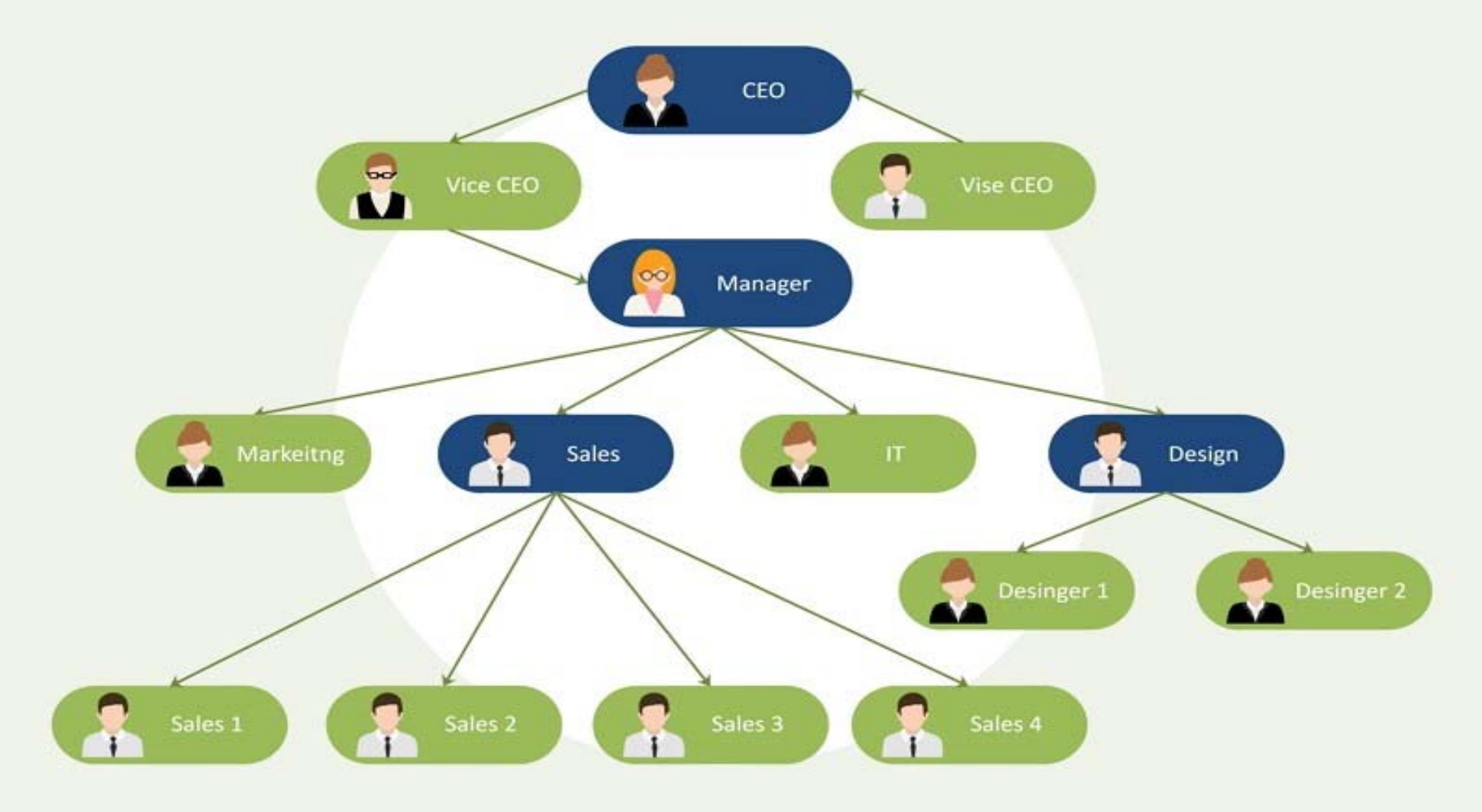
Example: Table of Contents



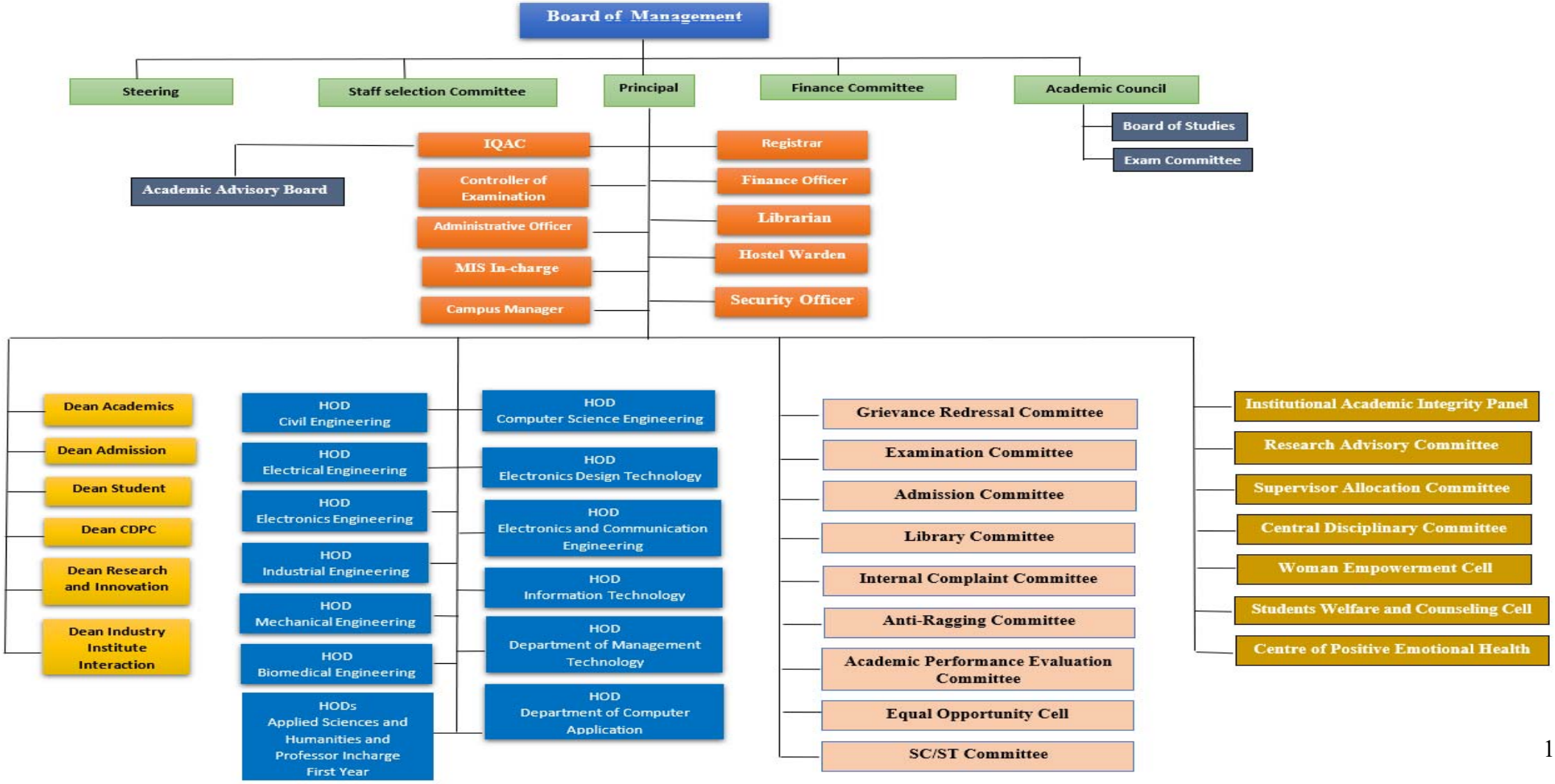
Example: Java's Class Hierarchy



Example: Business Company Hierarchy

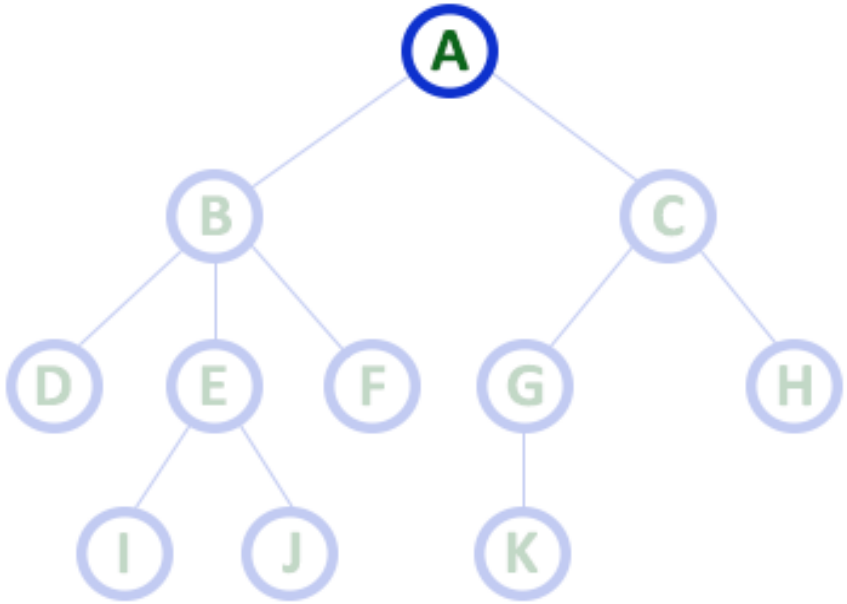


Example: Organization Structure at RCOEM



Tree terminology...

- 1.Root:
  - The first node is called as Root Node.
  - Every tree must have root node, there must be only one root node.
  - Root node doesn't have any parent.

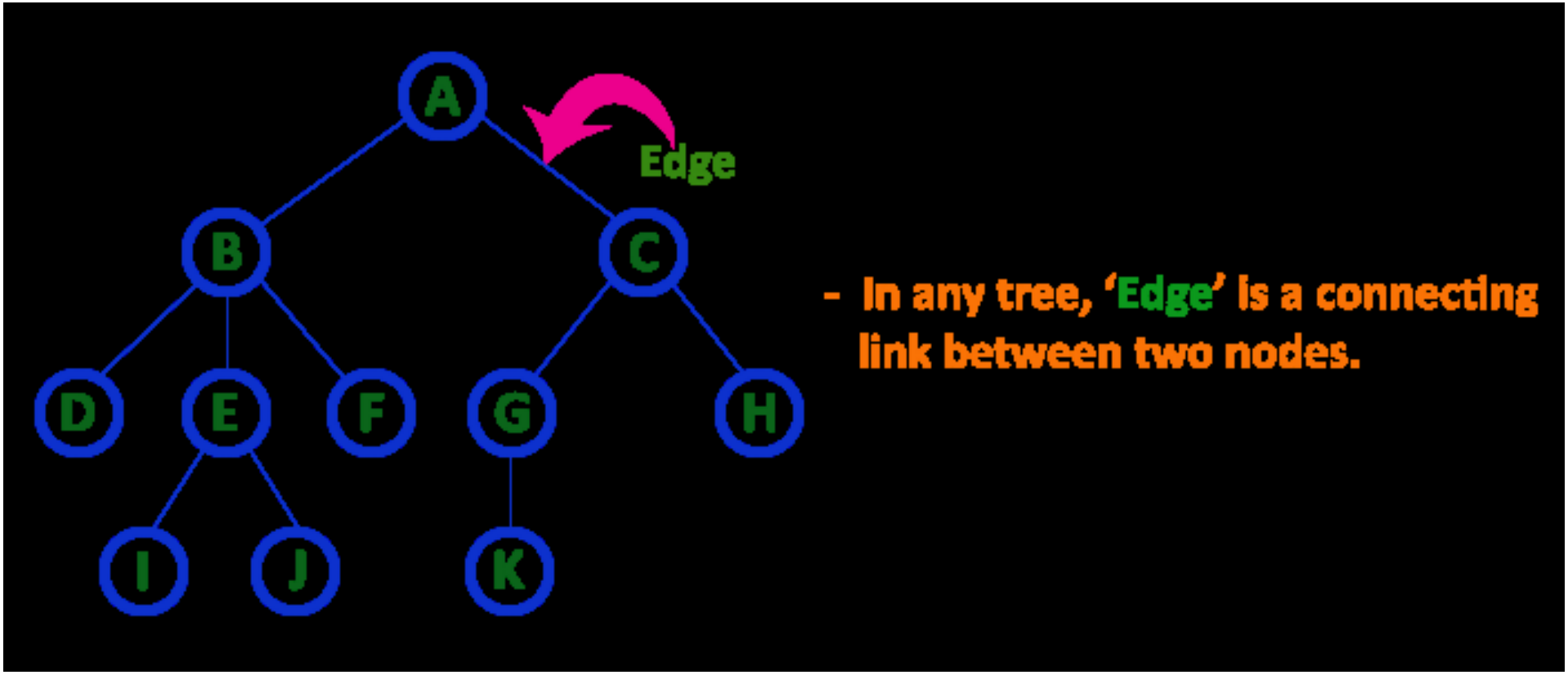


Here 'A' is the 'root' node

- In any tree the first node is called as ROOT node

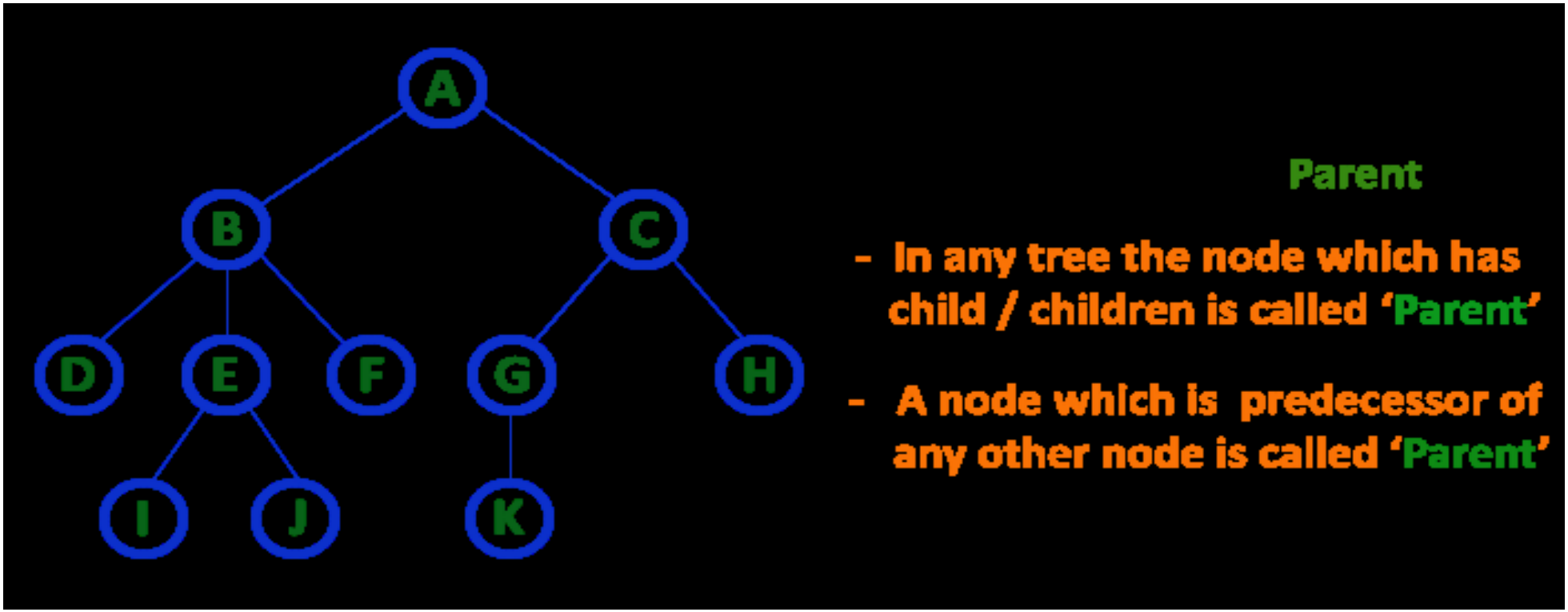
2. Edge

- In a tree data structure, the connecting link between any two nodes is called as EDGE.
- In a tree with 'N' number of nodes there will be a maximum of 'N-1' number of edges.



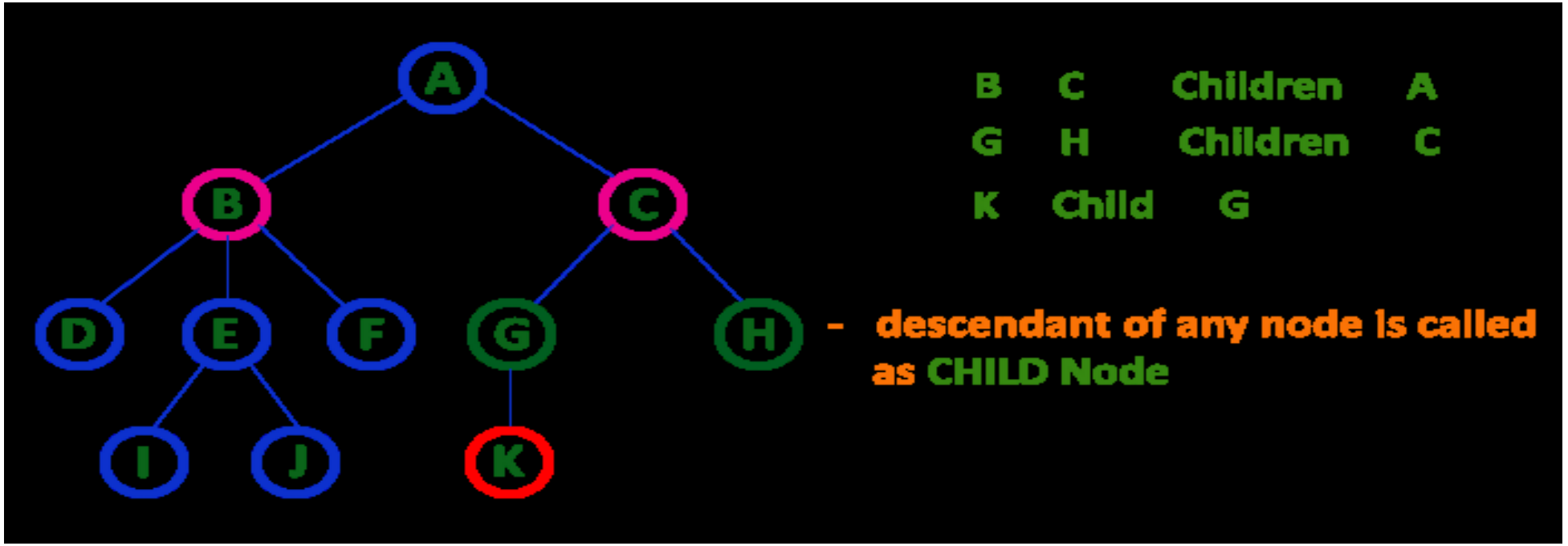
3. Parent

- In a tree data structure, the node which is predecessor of any node is called as PARENT NODE.
- In simple words, the node which has branch from it to any other node is called as parent node.
- Parent node can also be defined as "The node which has child / children".



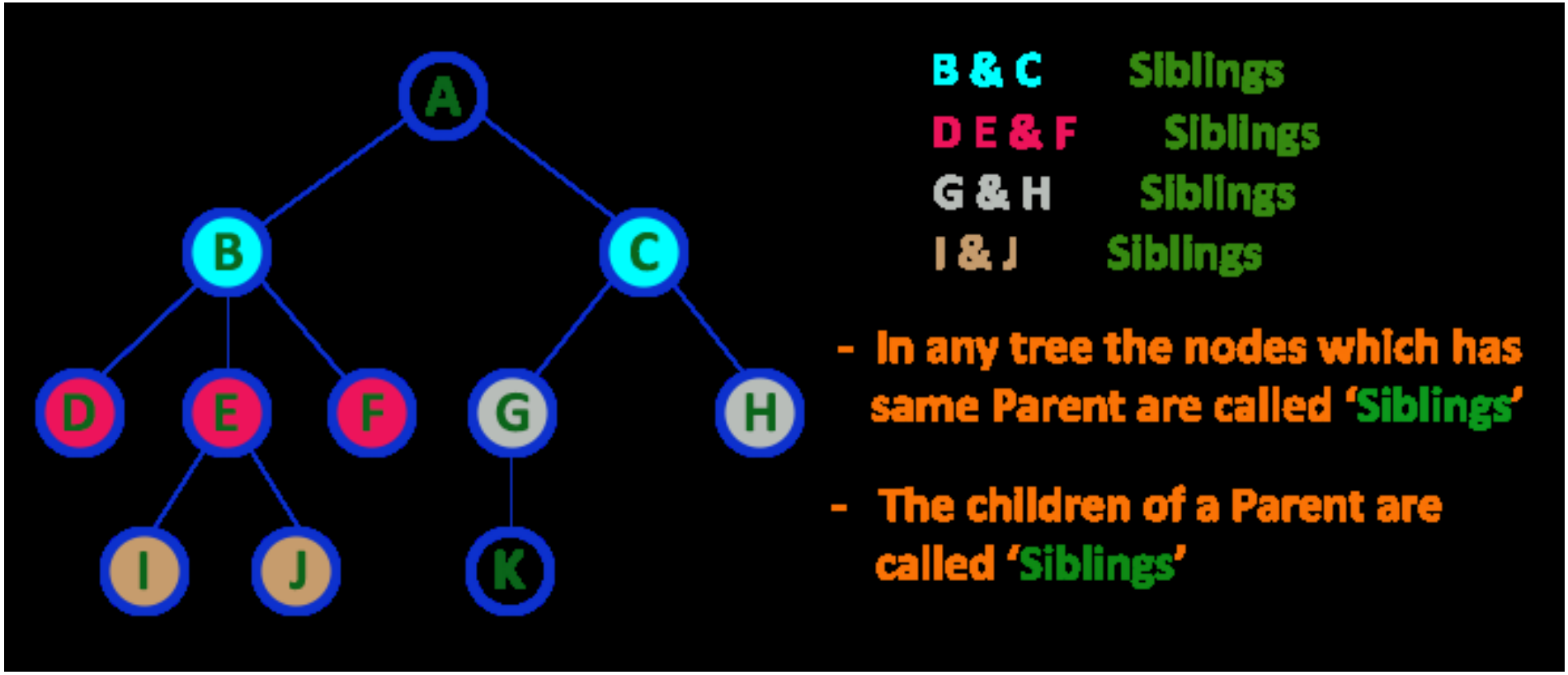
4. Child

- The node which has a link from its parent node is called as child node.
- In a tree, any parent node can have any number of child nodes.
- In a tree, all the nodes except root are child nodes.



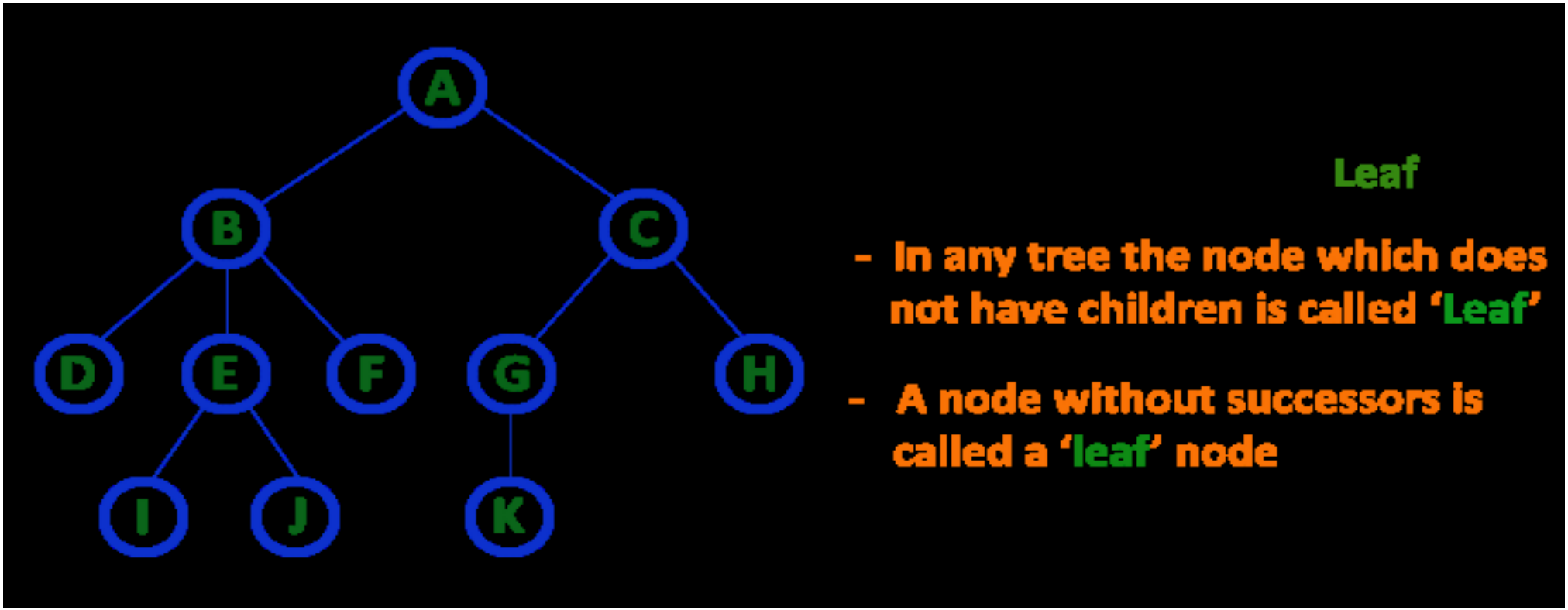
5. Siblings

- The nodes with same parent are called as Sibling nodes.



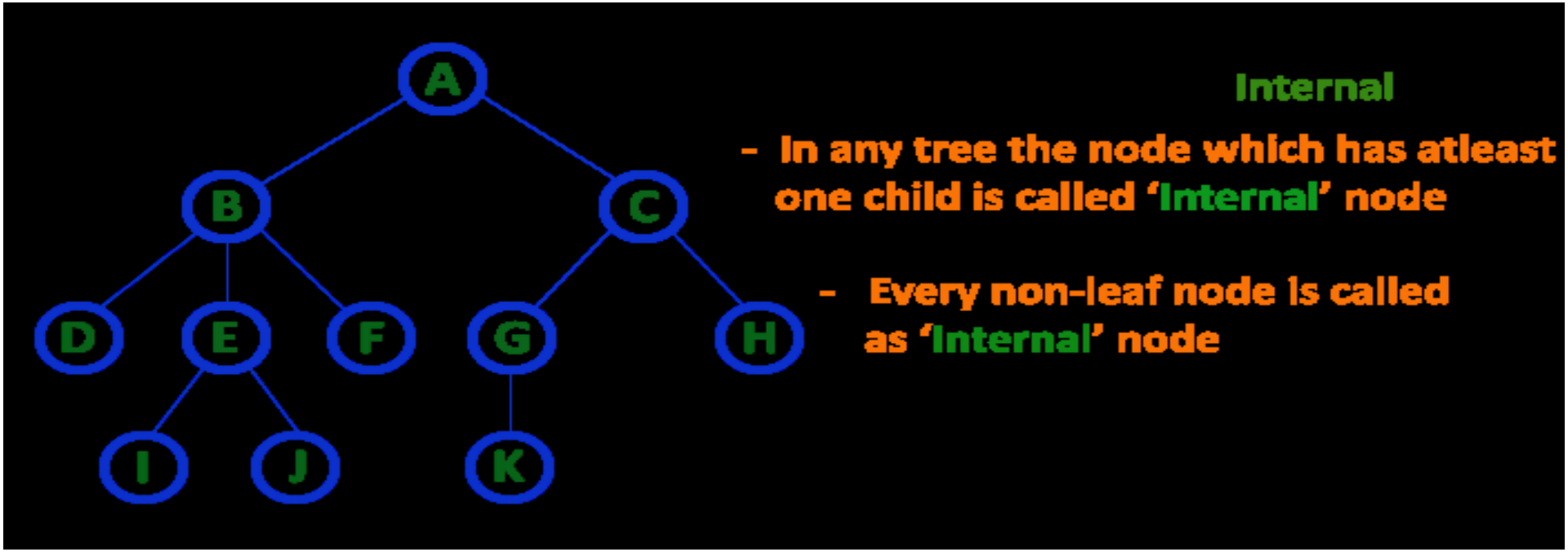
6. Leaf Node

- The node which does not have a child is called as LEAF Node.
- The leaf nodes are also called as External Nodes or 'Terminal' node.



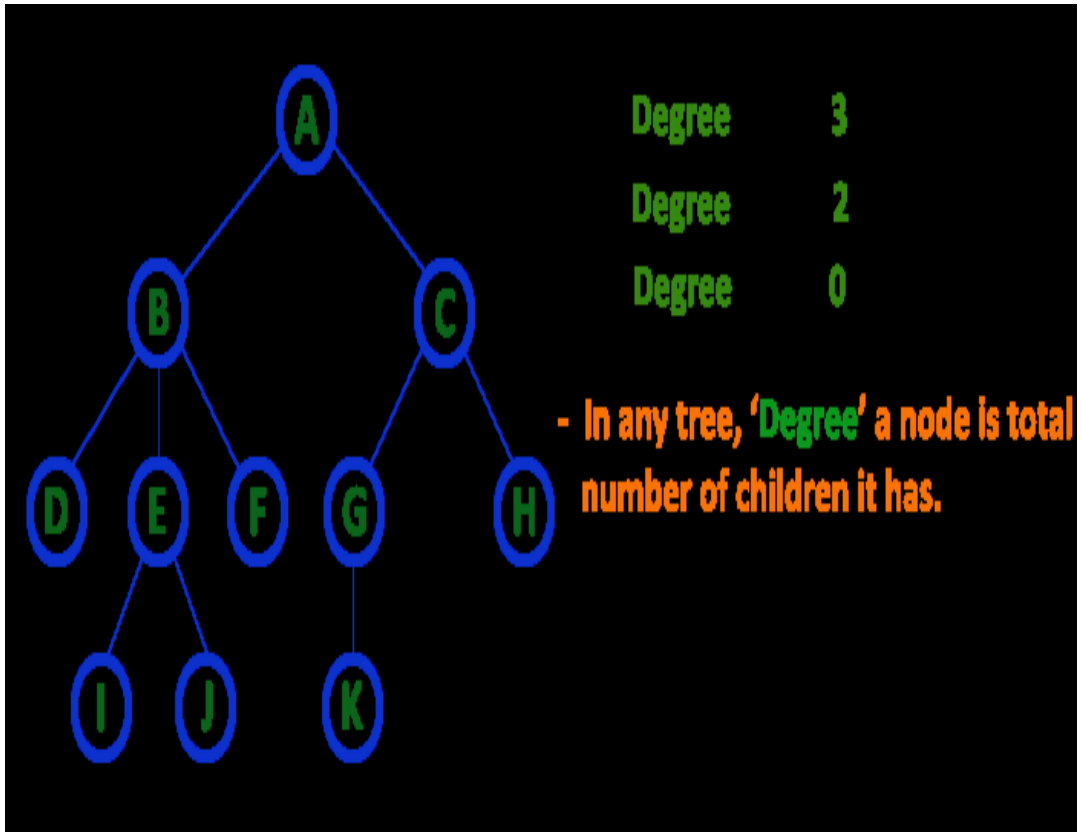
7. Internal Nodes

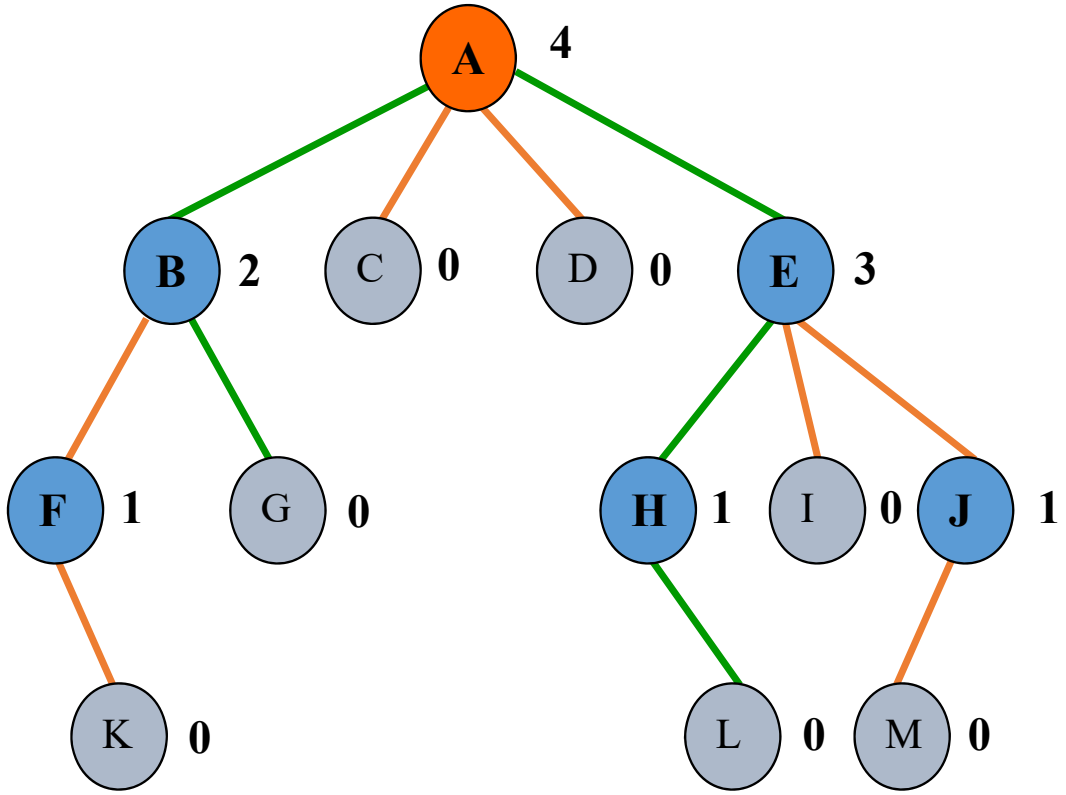
- An internal node is a node with at least one child.
- Nodes other than leaf nodes are called as Internal Nodes.
- The root node is also said to be Internal Node if the tree has more than one node.
- Internal nodes are also called as 'Non-Terminal' nodes.



8. Degree

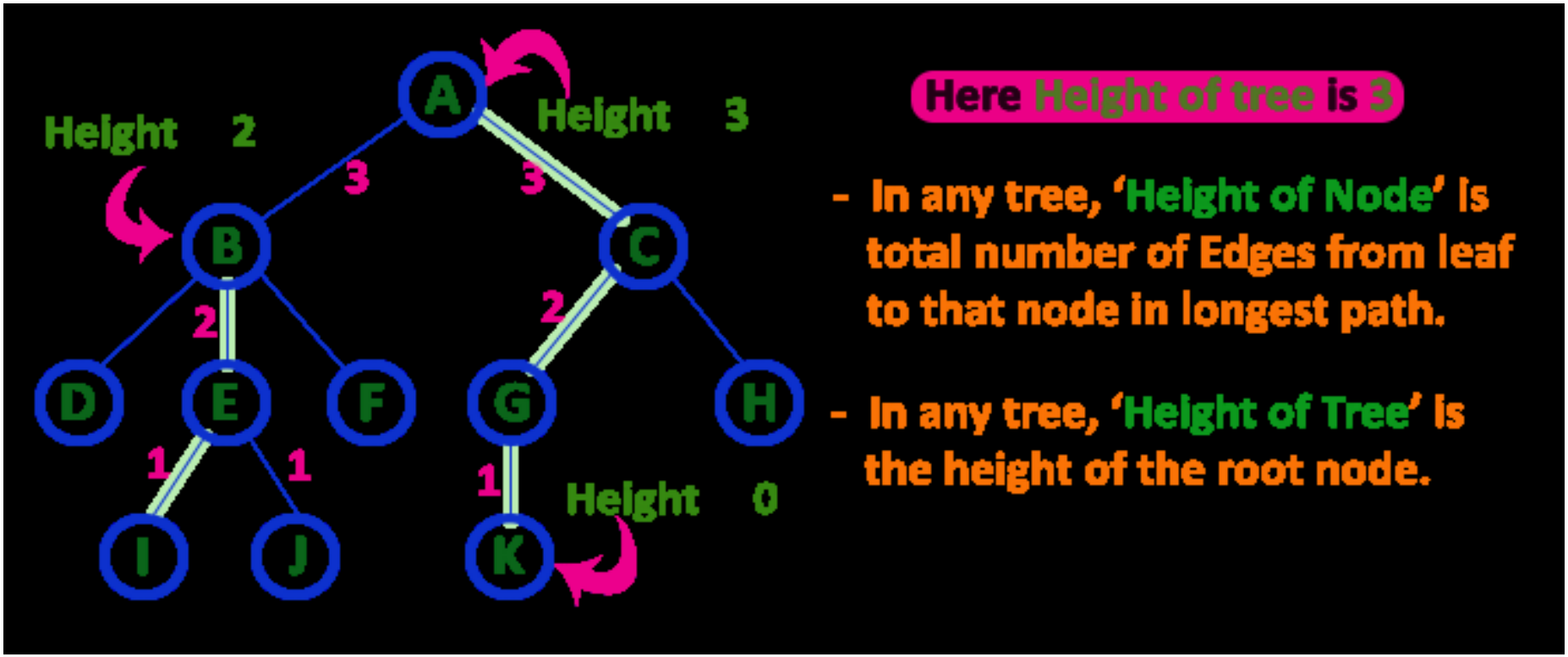
- In a tree data structure, the total number of children of a node is called as DEGREE of that Node.





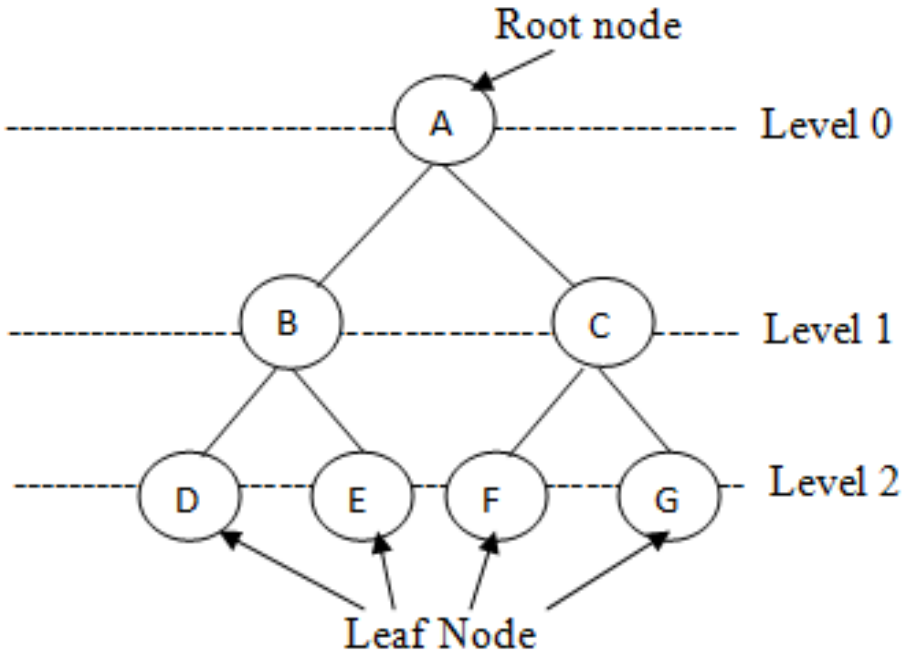
9. Height

- The total number of edges from leaf node to a particular node in the longest path is called as HEIGHT of that Node.
- In a tree, height of the root node is said to be height of the tree.

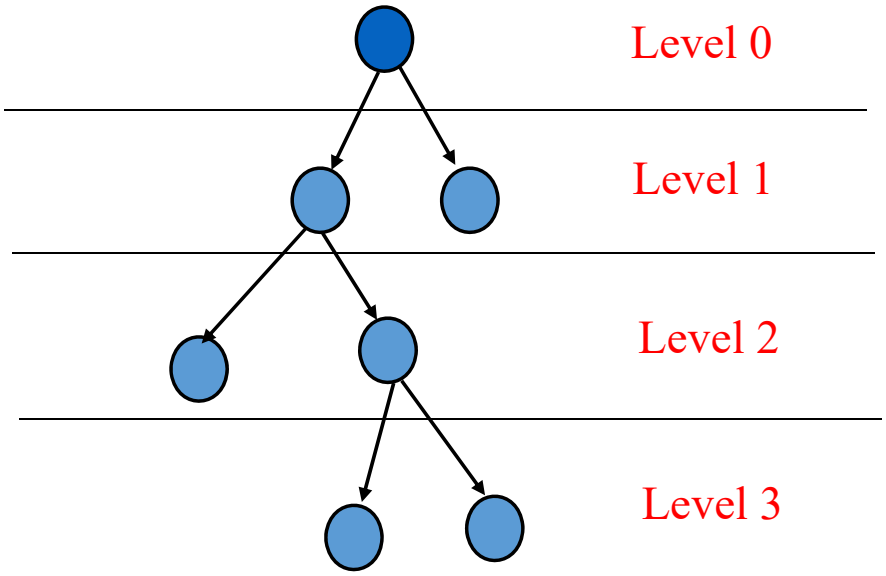


10. Level

- In a tree data structure, the root node is said to be at Level 0 and the children of root node are at Level 1 and the children of the nodes which are at Level 1 will be at Level 2 and so on...
- In a tree each step from top to bottom is called as a Level and the Level count starts with '0' and incremented by one at each level (Step).

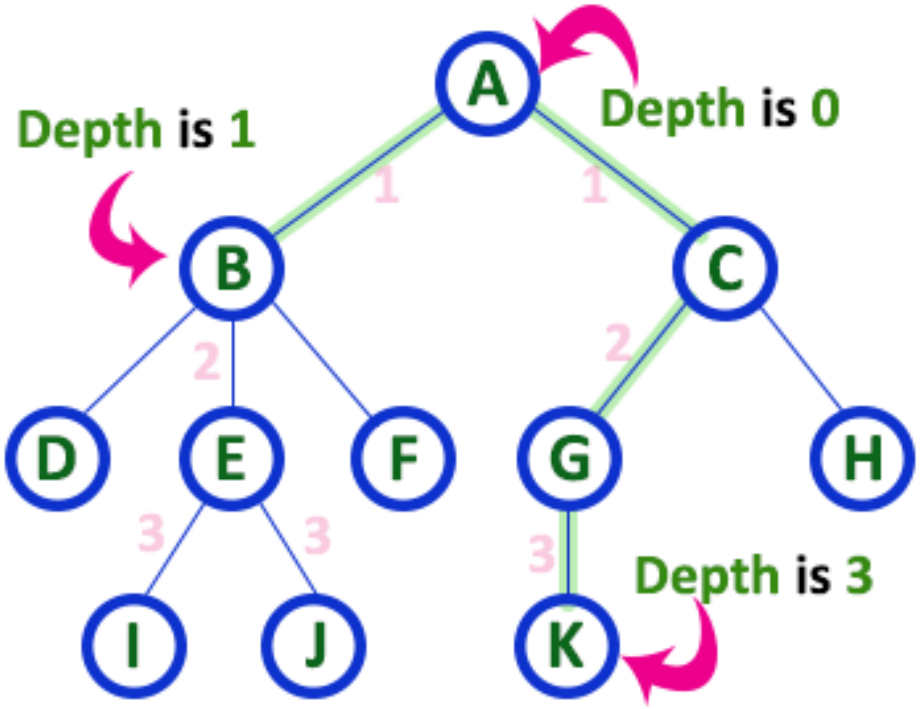


height=3



11. Depth

- In a tree data structure, the total number of edges from root node to a particular node is called as DEPTH of that Node.



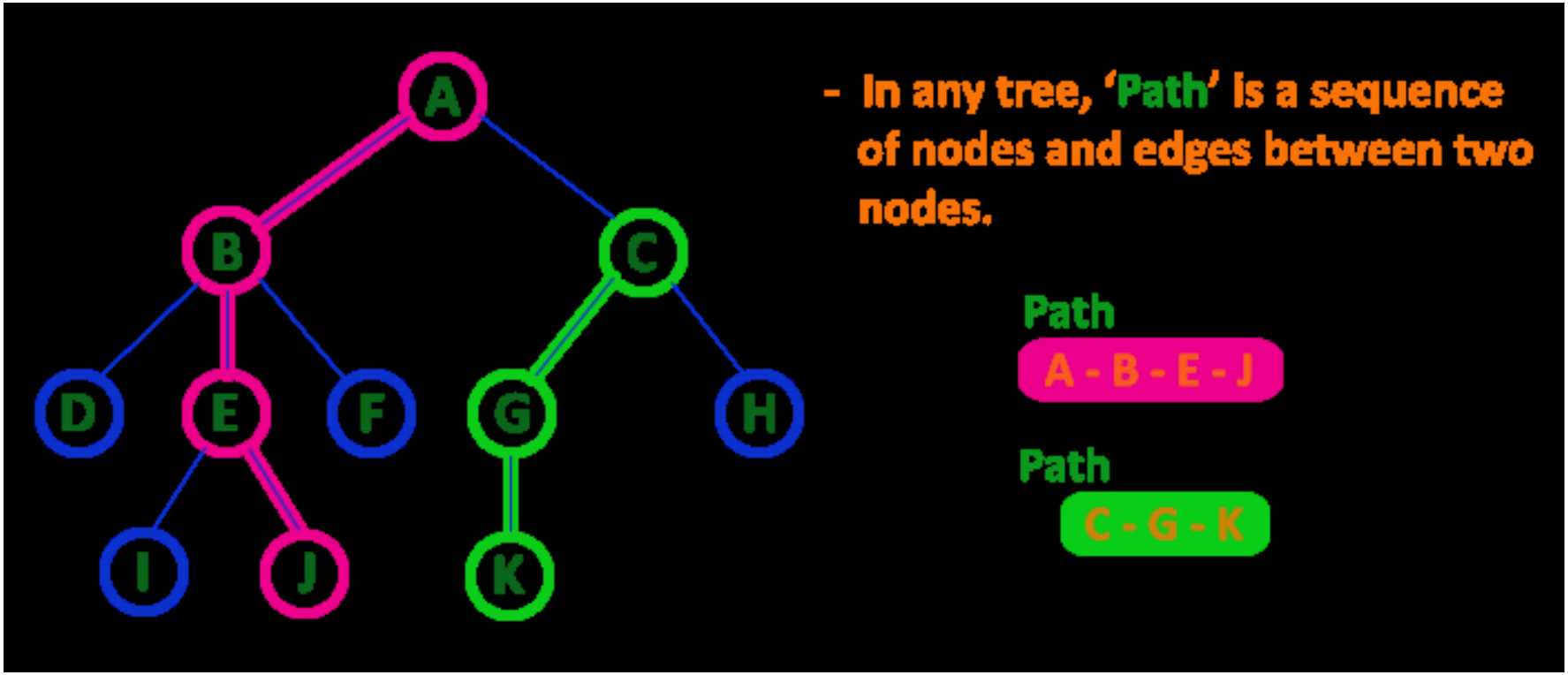
Here Depth of tree is 3

- In any tree, 'Depth of Node' is total number of Edges from root to that node.

- In any tree, 'Depth of Tree' is total number of edges from root to leaf in the longest path.

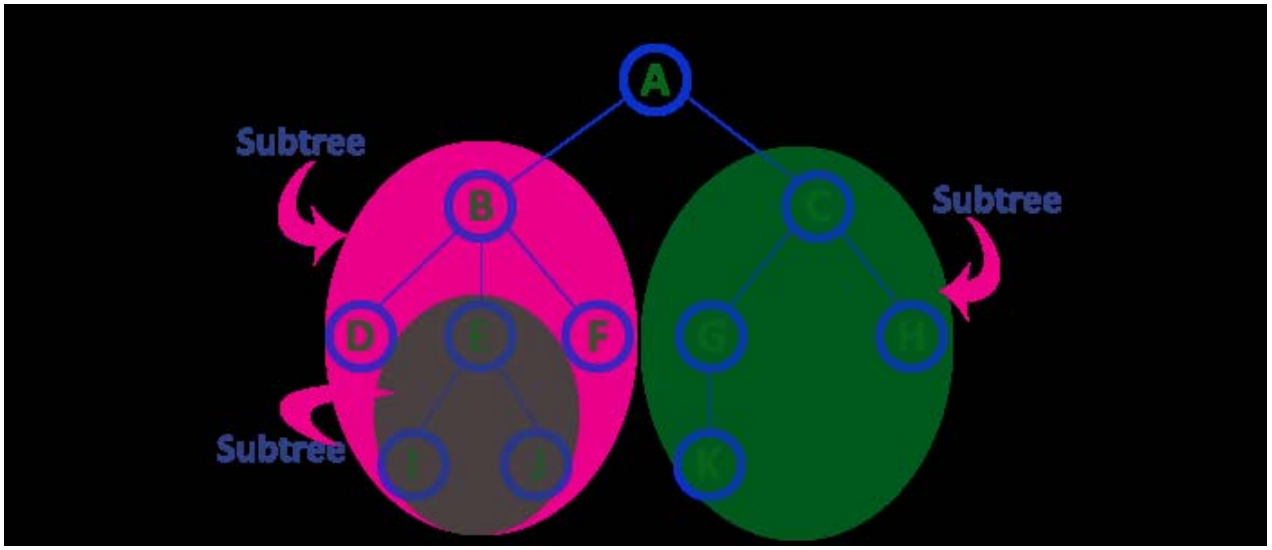
12. Path

- In a tree data structure, the sequence of Nodes and Edges from one node to another node is called as PATH between that two Nodes.
- Length of a Path is total number of nodes in that path. In below example the path A - B - E - J has length 4.



13. Sub Tree

- Every child node will form a subtree on its parent node.

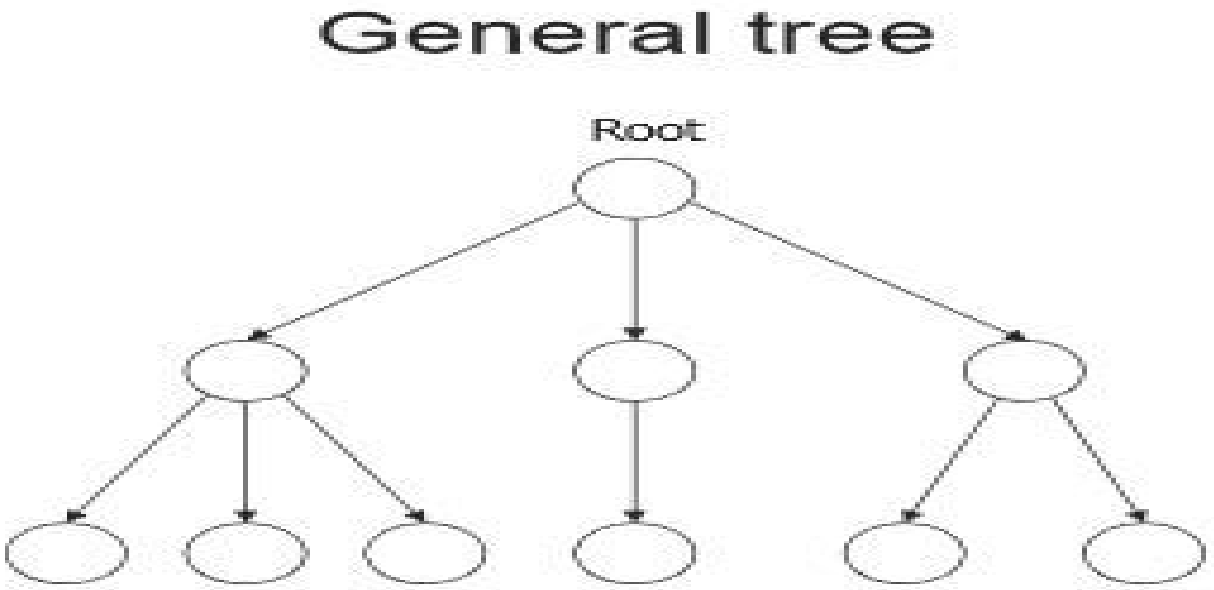


Type of Trees

- General tree
- Binary tree
- Binary Search Tree

General Tree

- A general tree is a **data structure** in that each node can have infinite number of children .
- In general tree, root has **in-degree 0** and maximum **out-degree n**.
- **Height** of a general tree is the length of longest path from root to the leaf of tree.
- Height(T) = {**max**(height(child1) , height(child2) , … height(child-n)) +1}



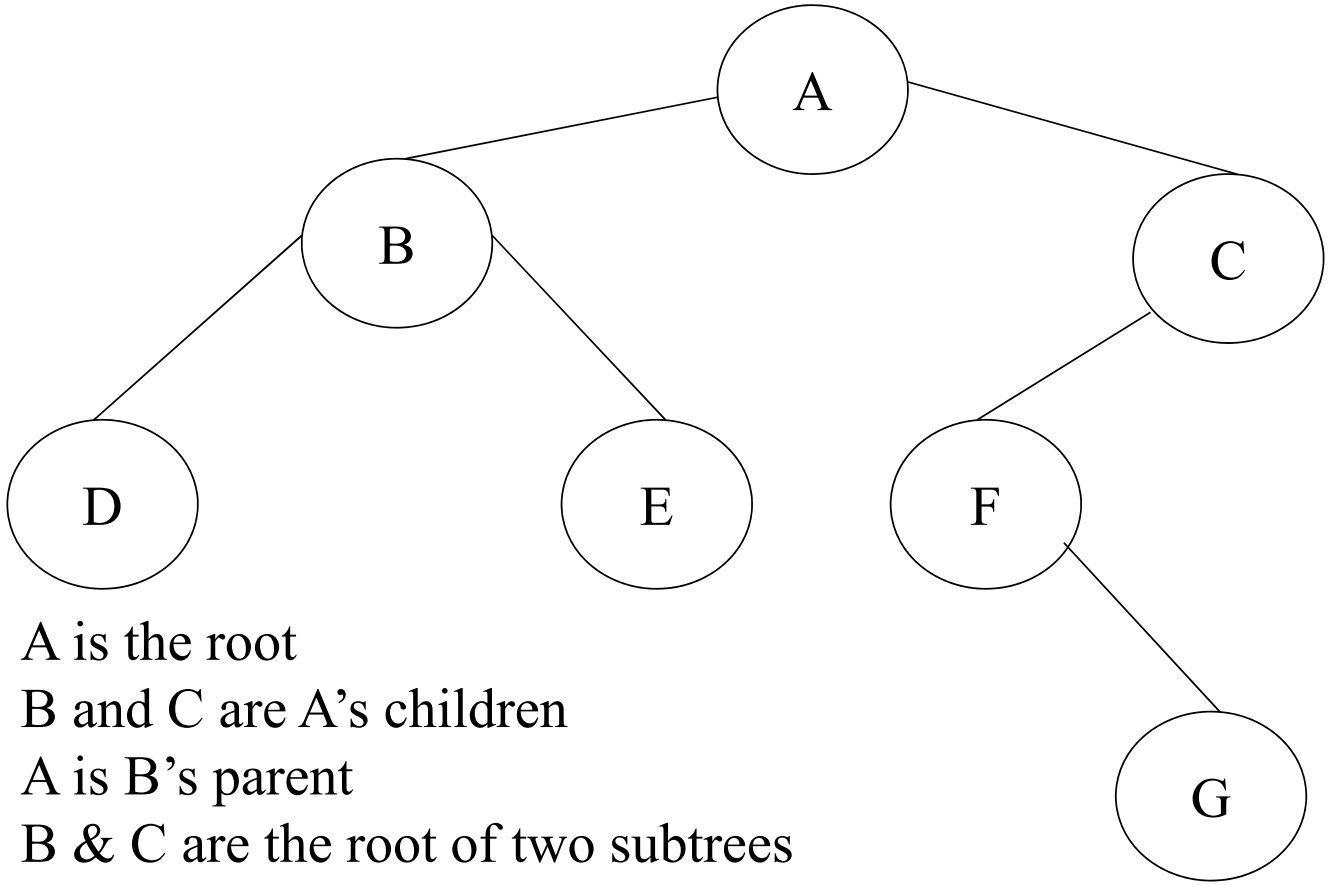
What is a Binary Tree

- A ***binary tree*** is a collection of nodes that consists of the ***root*** and other subsets to the root points, which are called the ***left*** and ***right subtrees***.

- Every ***parent*** node on a binary tree can have up to two ***child*** nodes (roots of the two subtrees); any more children and it becomes a general tree.

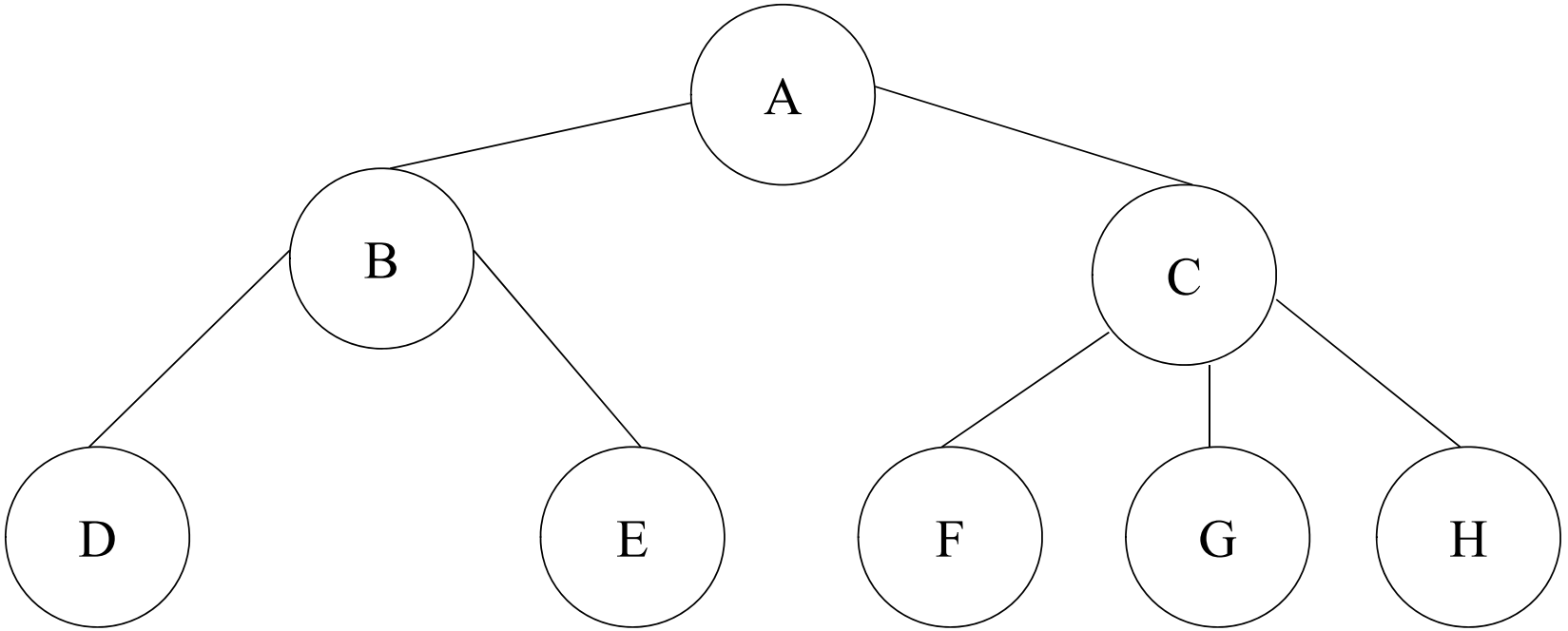
- A node that has no children is called a ***leaf*** node.

A Few Terms Regarding Binary Trees



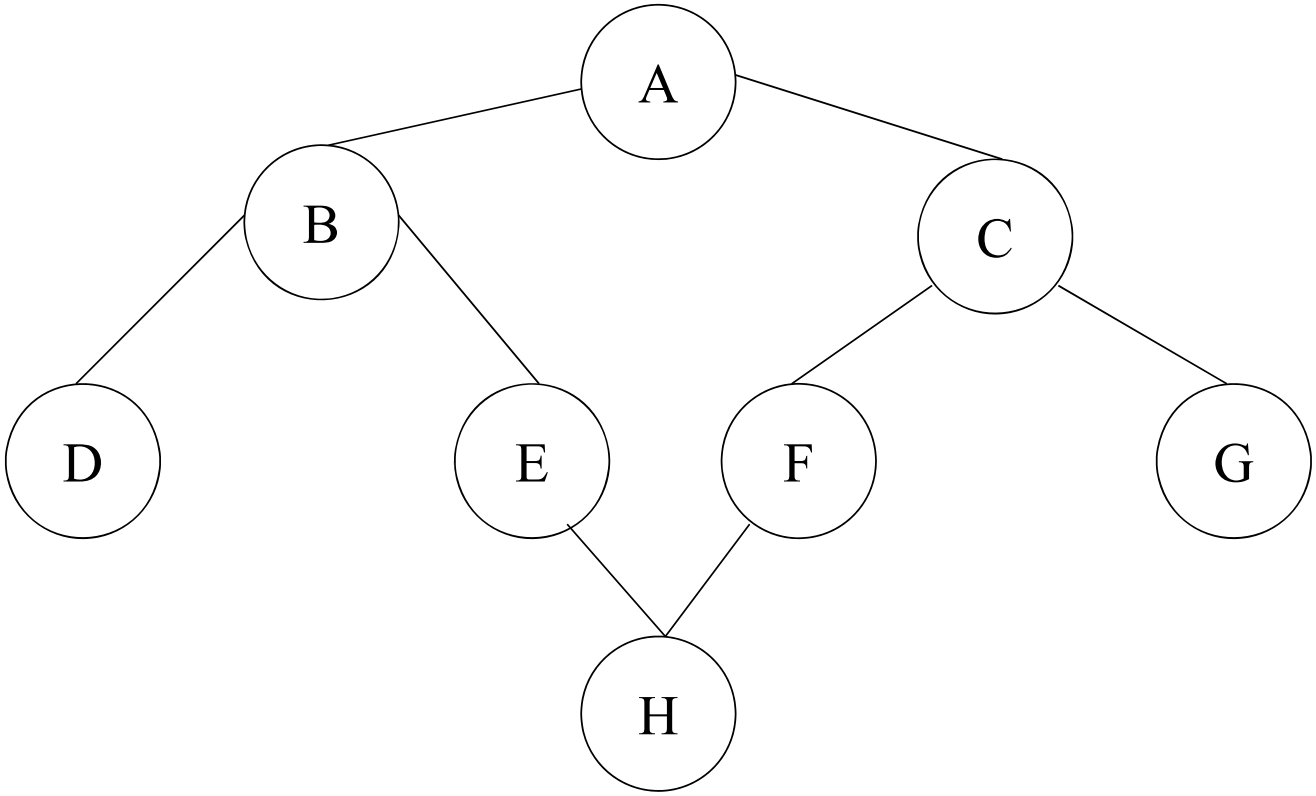
A is the root
B and C are A's children
A is B's parent
B & C are the root of two subtrees
D, E, and G are leaves

This is ***NOT*** A Binary Tree



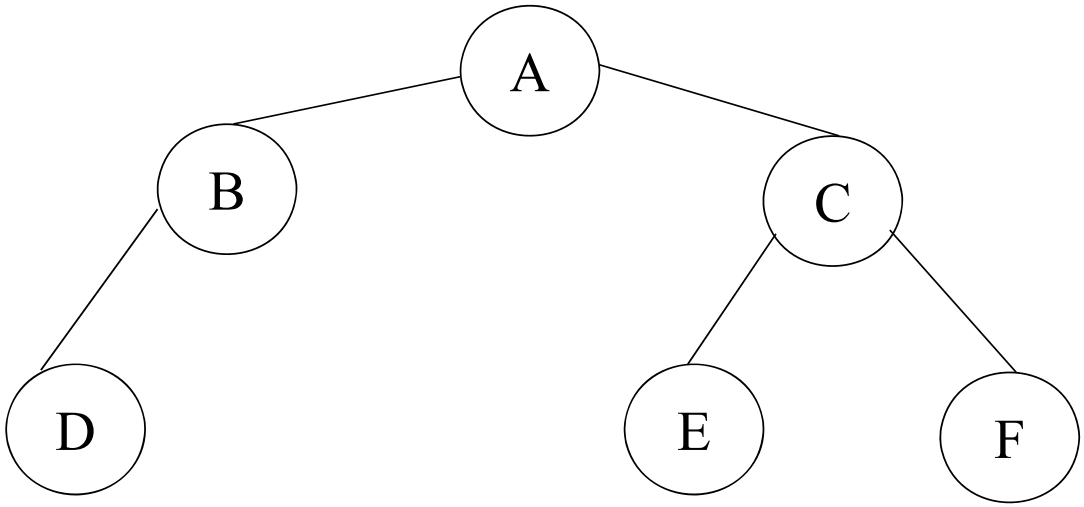
This is a ***general tree*** because C has three child nodes

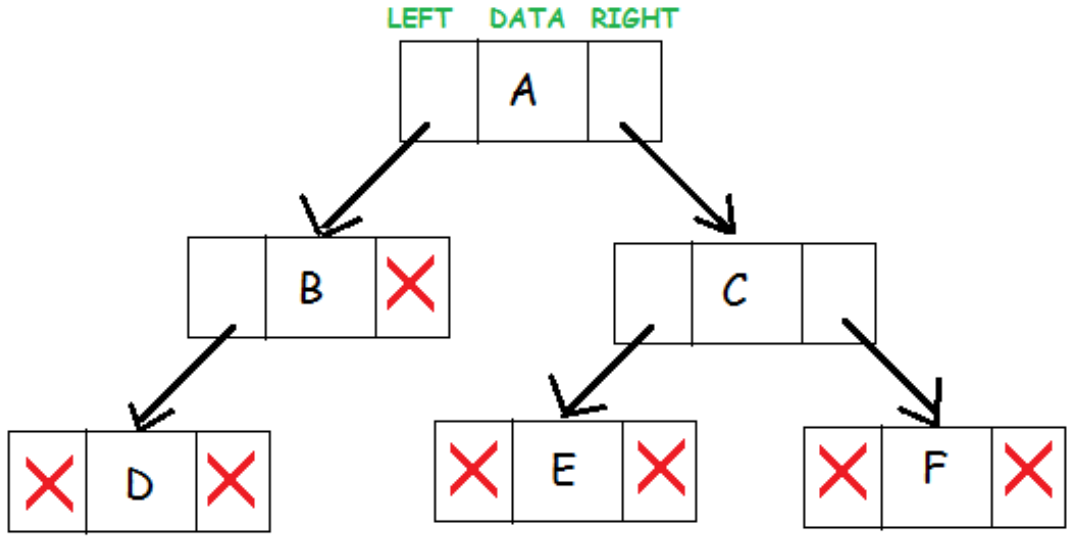
This is ***NOT*** A Binary Tree



This is a ***graph***
because A, B, E, H, F and C form a circuit

Memory Tree Representation (Linked List)






```
struct node {
      int data;
      struct node *Left_Child;
      struct node * Right_Child;
            };
struct node *tree;
```

```
struct Node* createNode(int value) {
    struct Node* newNode = (struct Node*)malloc(sizeof(struct
Node));
    newNode->data = value;
    newNode->left = NULL;
    newNode->right = NULL;
    return newNode;
```

Algorithm for Creation/Insertion of Binary Tree

Step 1: [check whether the tree is empty]
IF Root = NULL
Root = Create a Node
Left_Child [Root] = NULL
Right_Child [Root] = NULL
Step 2: [Test for the left child]
If data < data [Root]
Left_Child [Root] = Call Create_Tree
(Left_Child [Root], data)
Else
Right_Child [Root] = Call Create_Tree
(Right_Child [Root], data)
Step 3: Return (Root)
Step 4: EXIT

Tree Traversal

- There are three common ways to traverse a tree:
  - **Preorder**: Visit the root, traverse the left subtree (preorder) and then traverse the right subtree (preorder)
  - **Postorder**: Traverse the left subtree (postorder), traverse the right subtree (postorder) and then visit the root.
  - **Inorder**: Traverse the left subtree (in order), visit the root and the traverse the right subtree (in order).

Tree Traversal Algorithm

```
Step 1: Repeat Steps 2 to 4 while TREE != NULL
Step 2:            Write TREE -> DATA
Step 3:            PREORDER(TREE -> LEFT)
Step 4:            PREORDER(TREE -> RIGHT)
       [END OF LOOP]
Step 5: END
```

PRE-ORDER TRAVERSAL

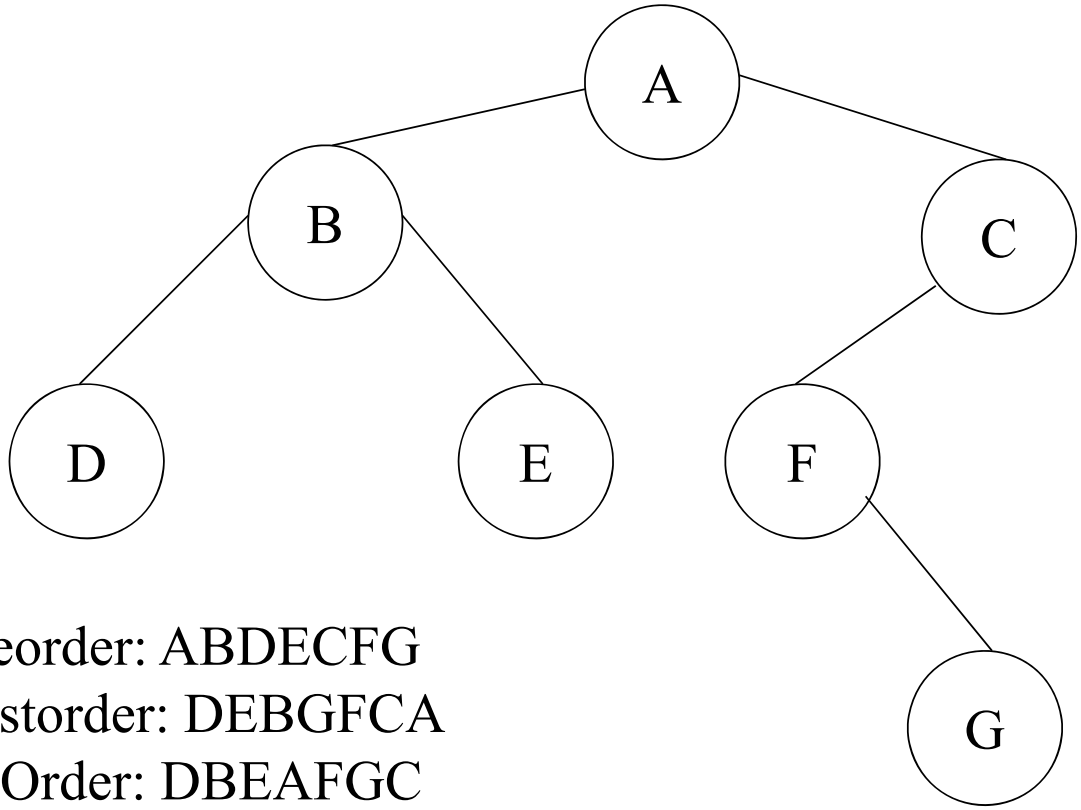
```
Step 1: Repeat Steps 2 to 4 while TREE != NULL
Step 2:            INORDER(TREE -> LEFT)
Step 3:            Write TREE -> DATA
Step 4:            INORDER(TREE -> RIGHT)
       [END OF LOOP]
Step 5: END
```

IN-ORDER TRAVERSAL

```
Step 1: Repeat Steps 2 to 4 while TREE != NULL
Step 2:            POSTORDER(TREE -> LEFT)
Step 3:            POSTORDER(TREE -> RIGHT)
Step 4:            Write TREE -> DATA
       [END OF LOOP]
Step 5: END
```

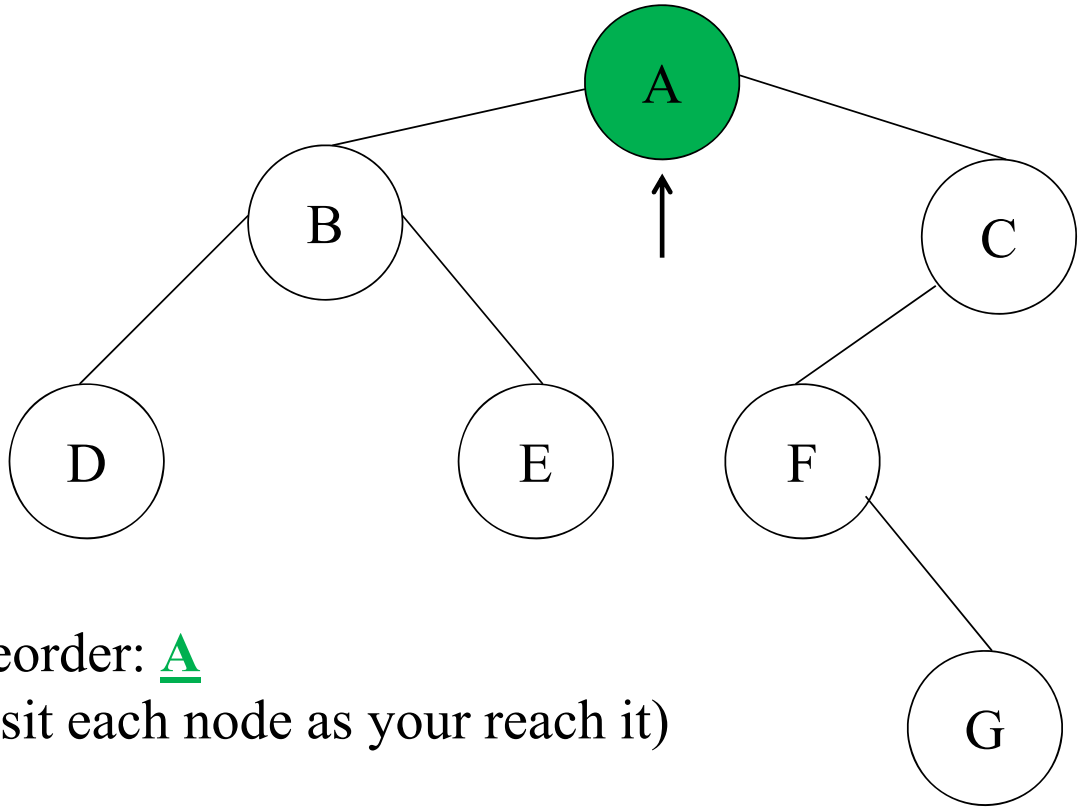
POST-ORDER TRAVERSAL

Tree Traversals: An Example



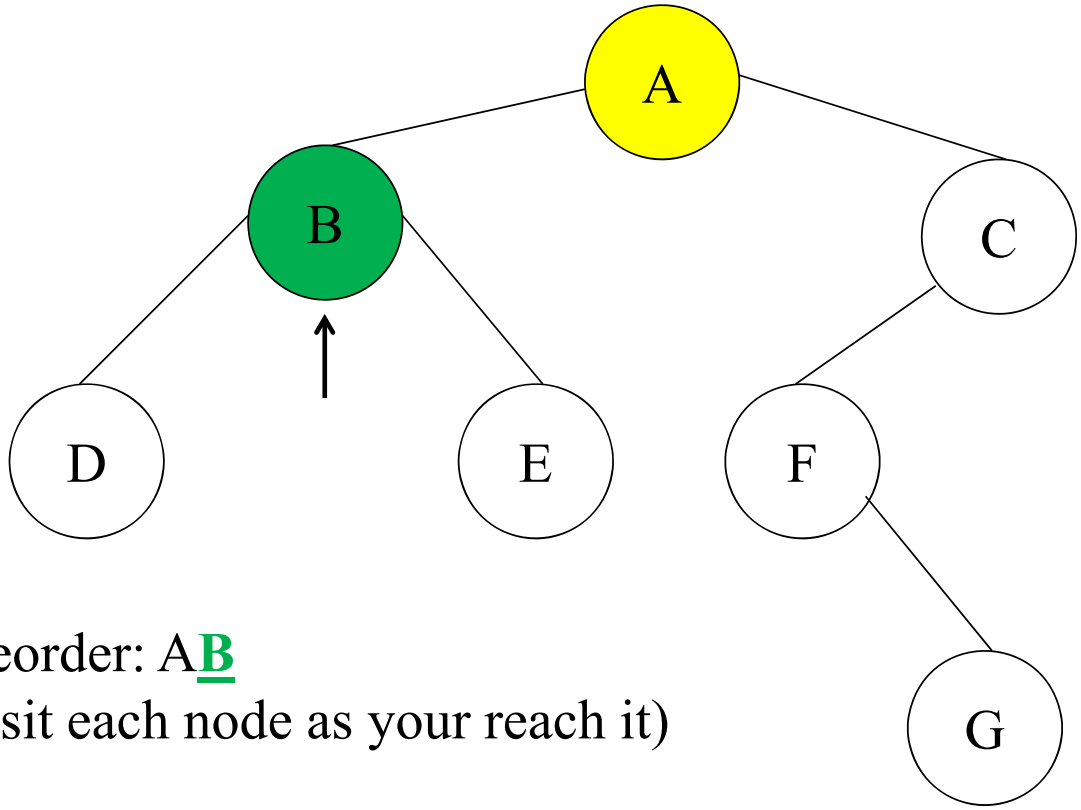
Preorder: ABDECFG
Postorder: DEBGFCA
In Order: DBEAFGC

Tree Traversals: An Example



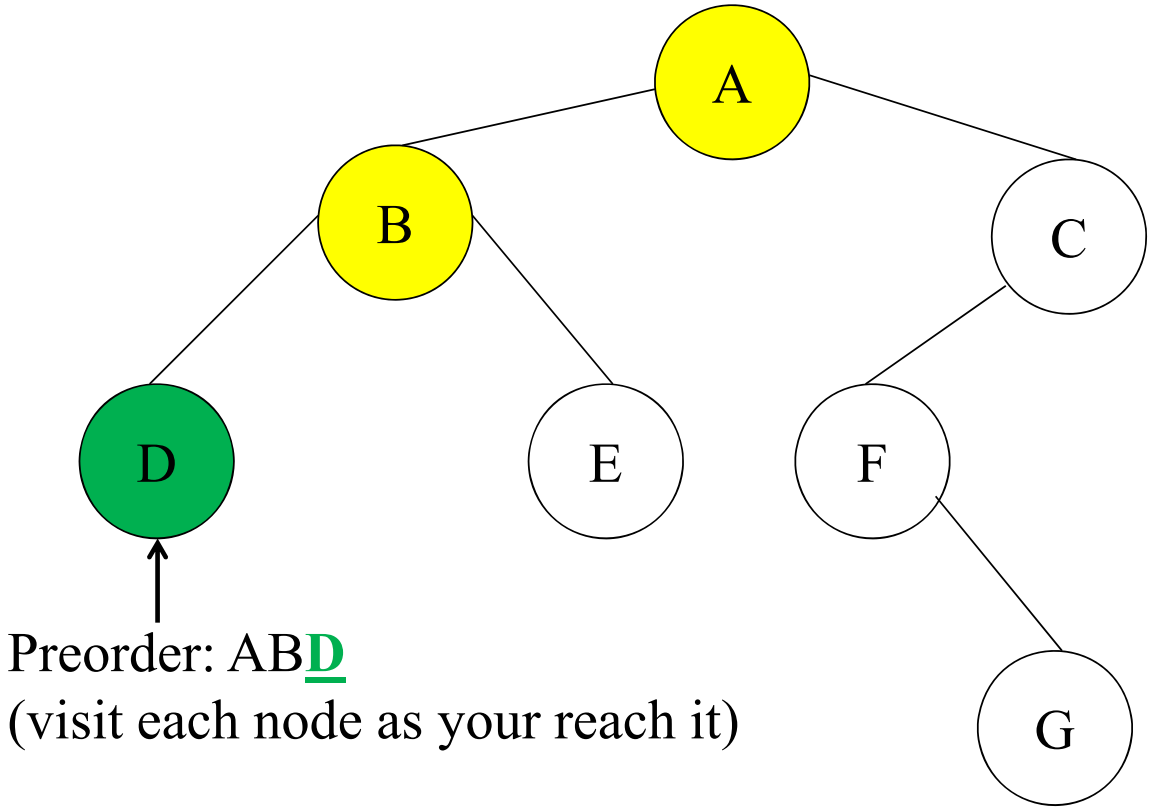
Preorder: A
(visit each node as your reach it)

Tree Traversals: An Example



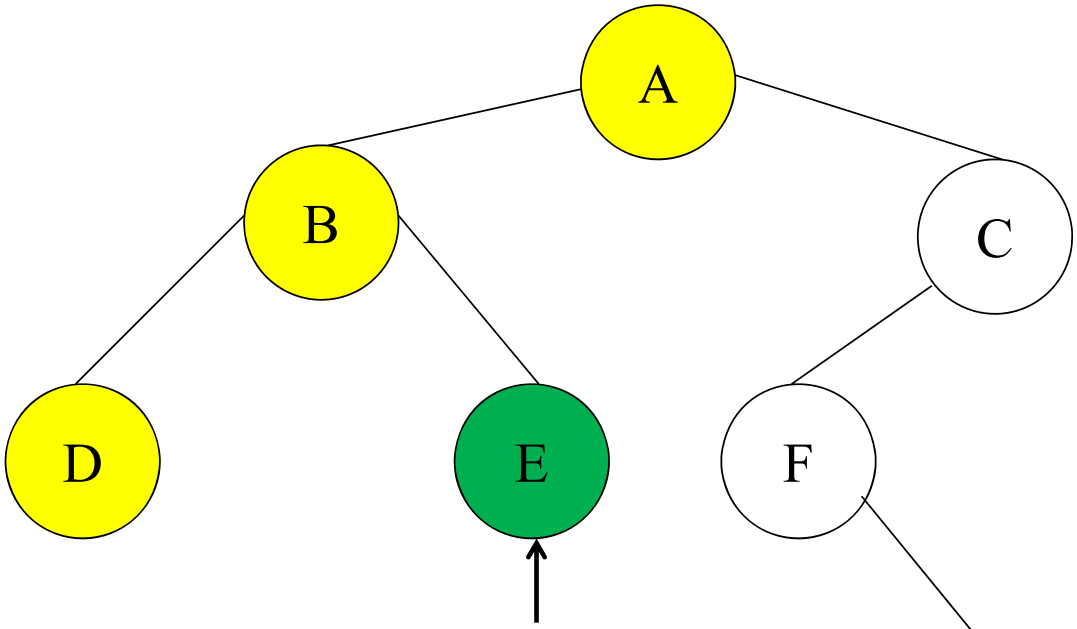
Preorder: A**B**
(visit each node as your reach it)

Tree Traversals: An Example



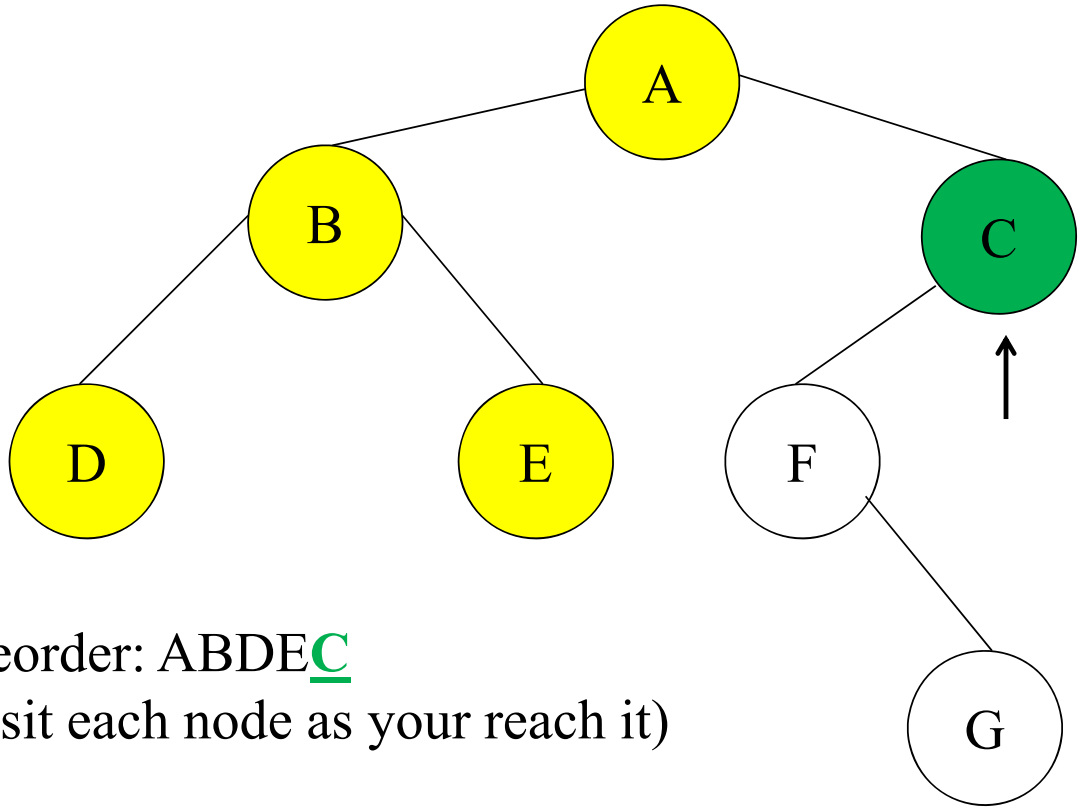
Preorder: AB**D**
(visit each node as your reach it)

Tree Traversals: An Example



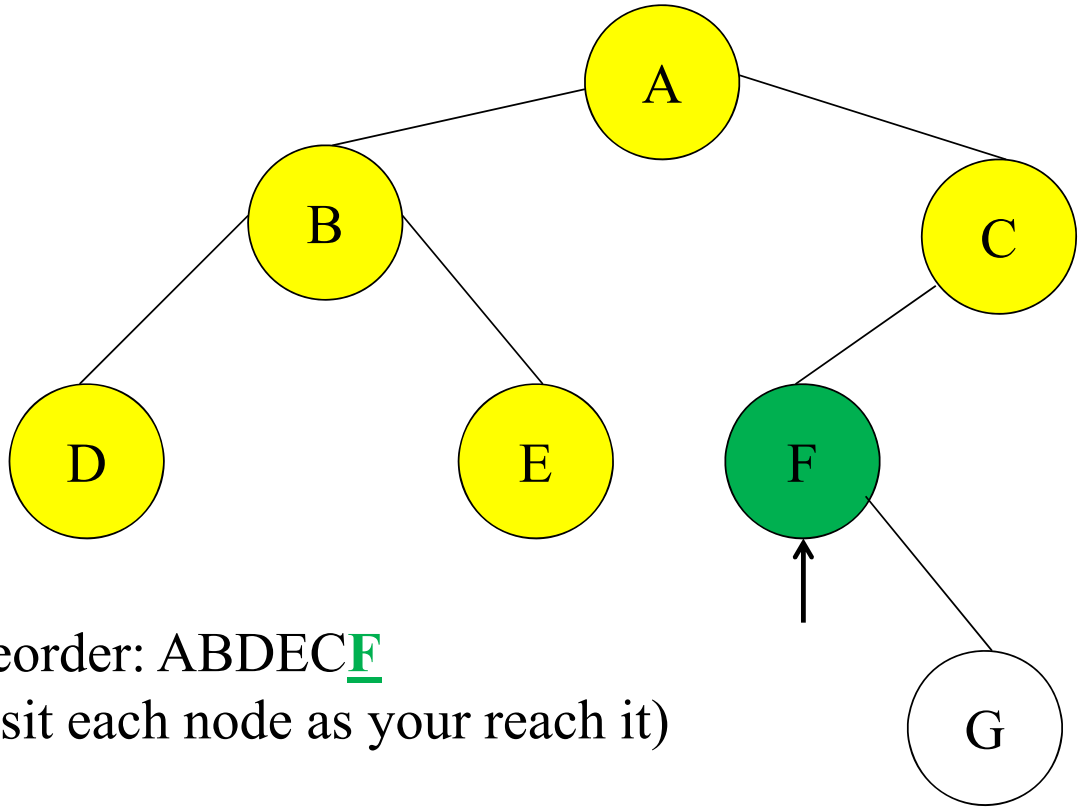
Preorder: ABDE
(visit each node as your reach it)

Tree Traversals: An Example



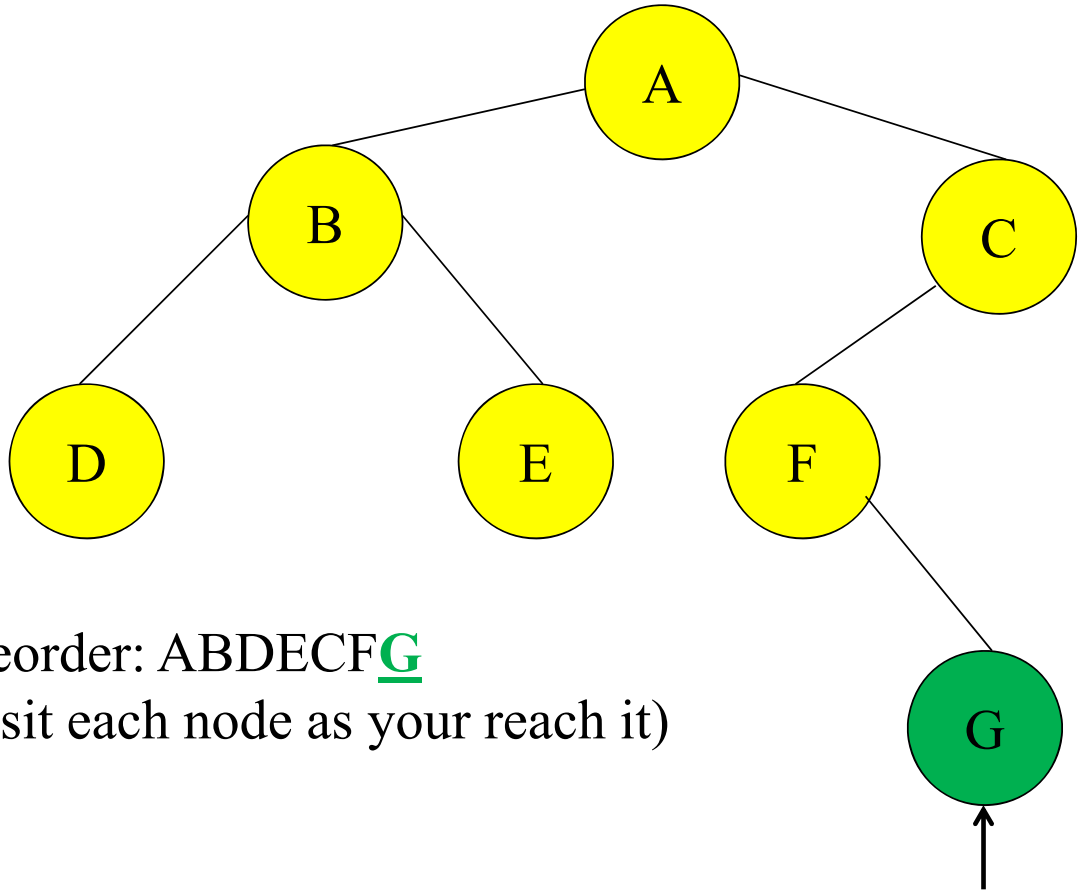
Preorder: ABDEC
(visit each node as your reach it)

Tree Traversals: An Example



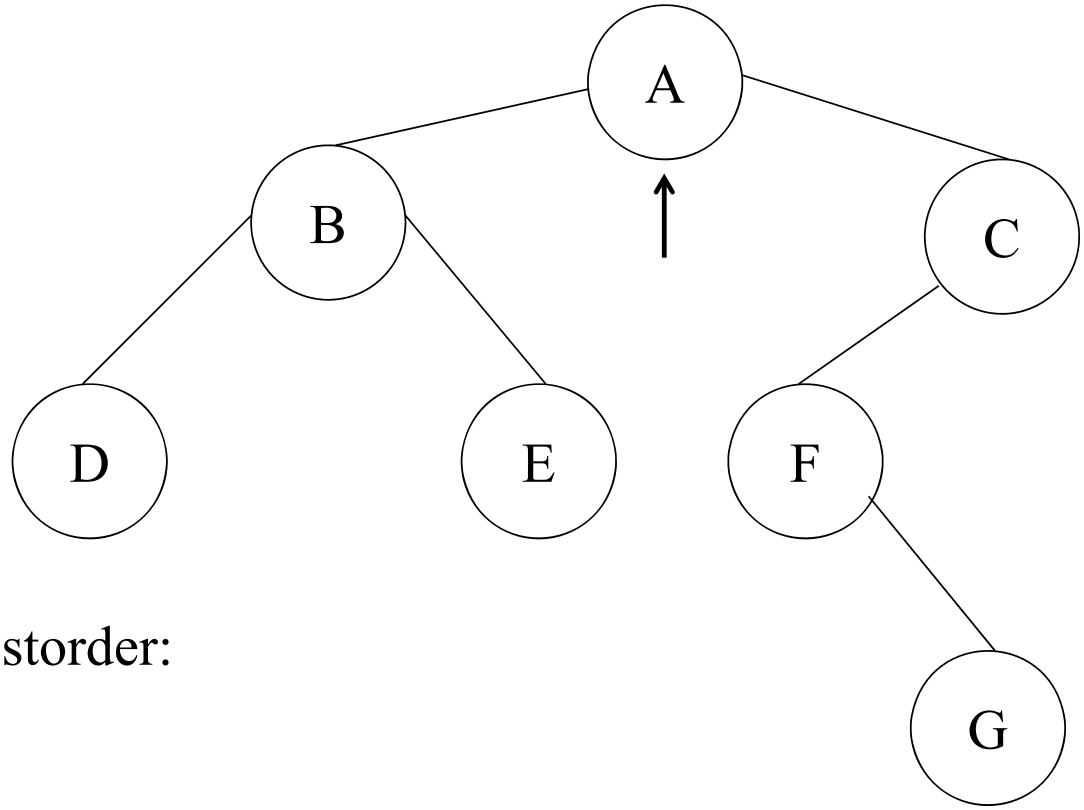
Preorder: ABDECF
(visit each node as your reach it)

Tree Traversals: An Example



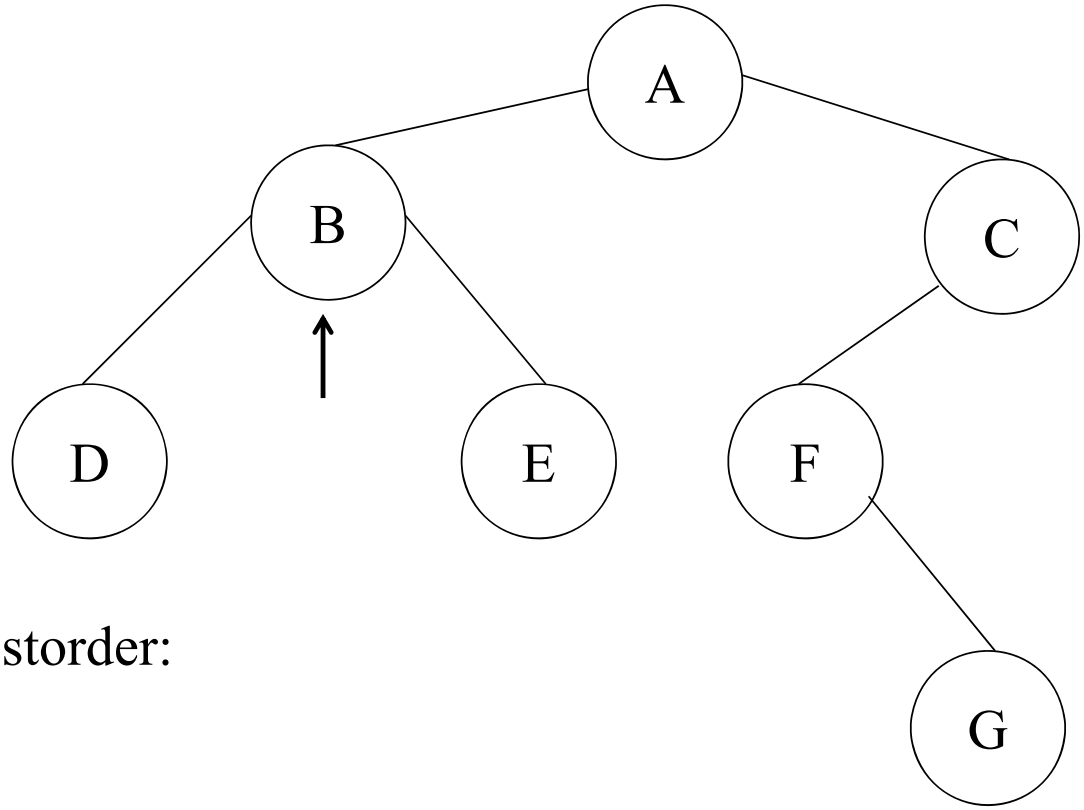
Preorder: ABDECFG
(visit each node as your reach it)

Tree Traversals: An Example



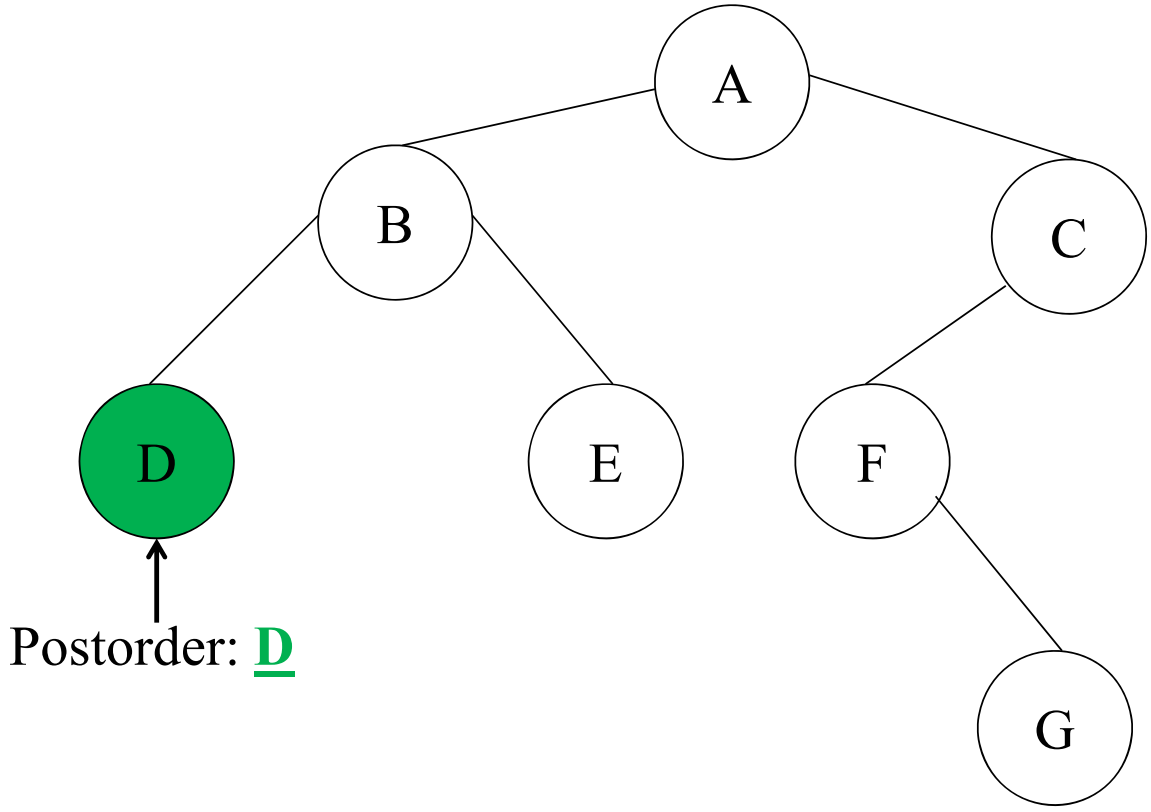
Postorder:

Tree Traversals: An Example

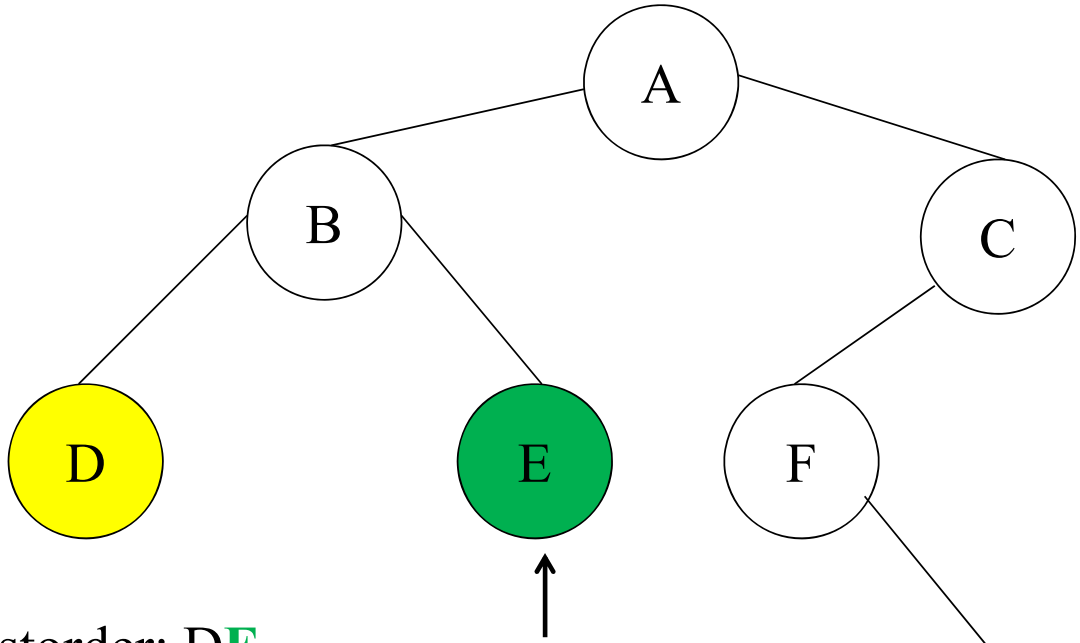


Postorder:

Tree Traversals: An Example

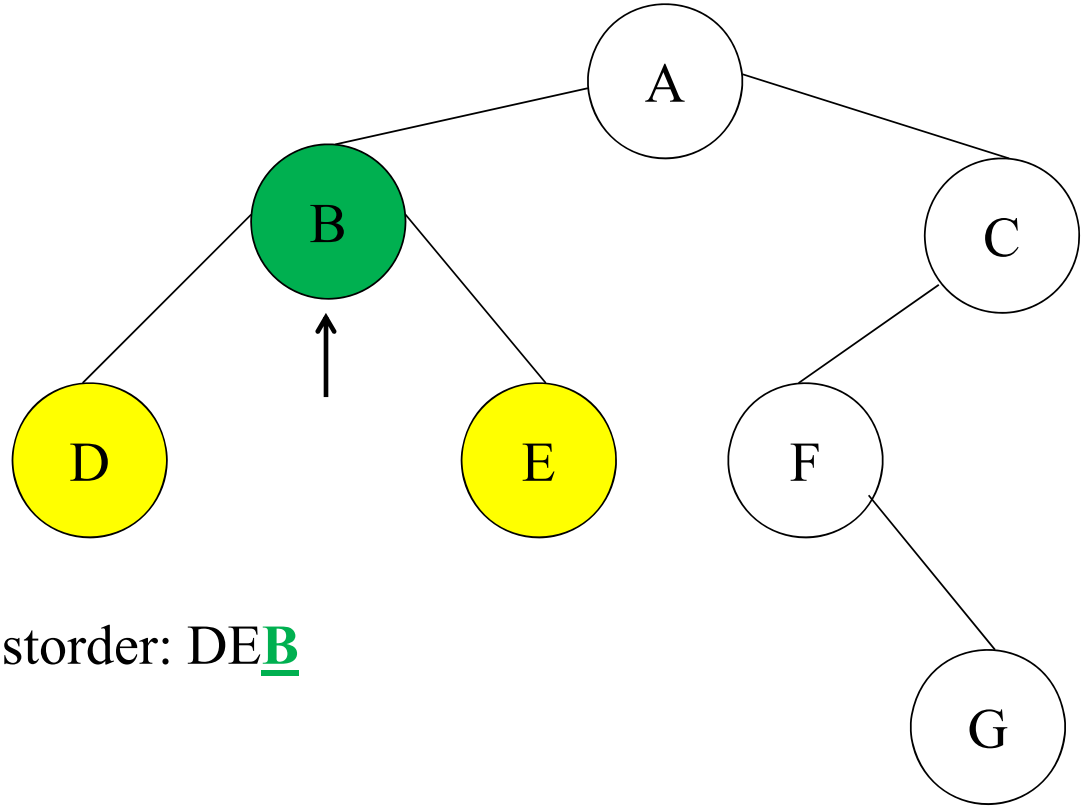


Tree Traversals: An Example



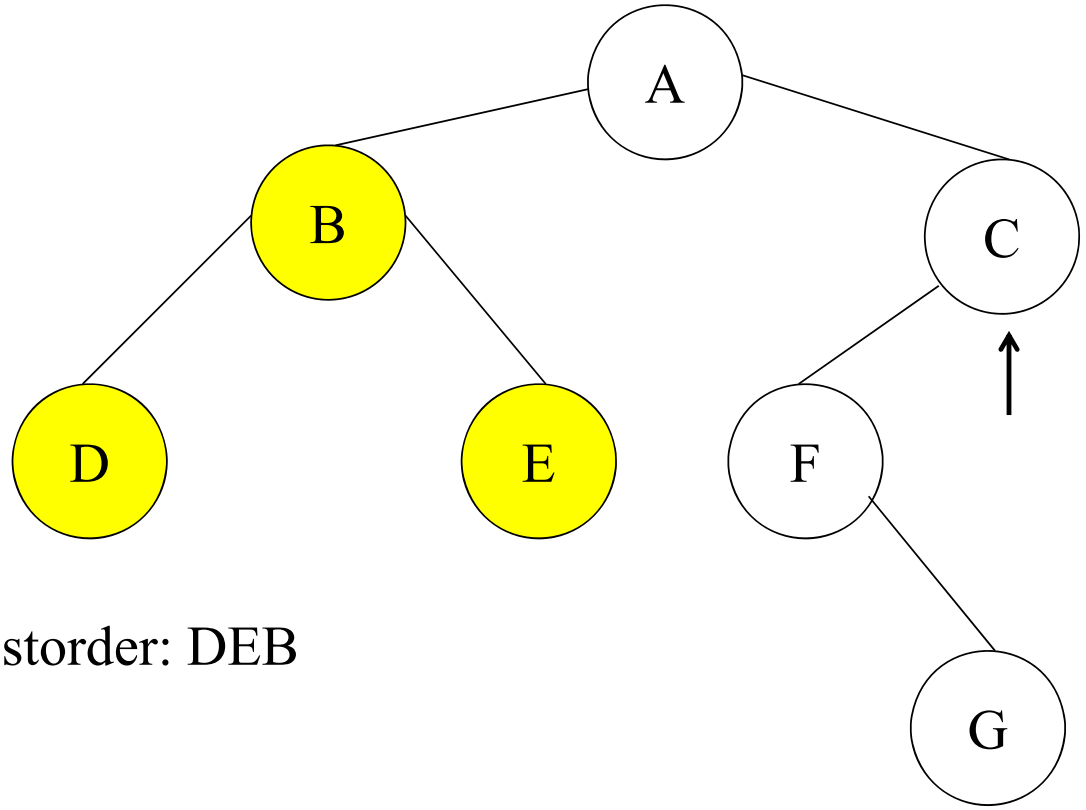
Postorder: D**E**

Tree Traversals: An Example



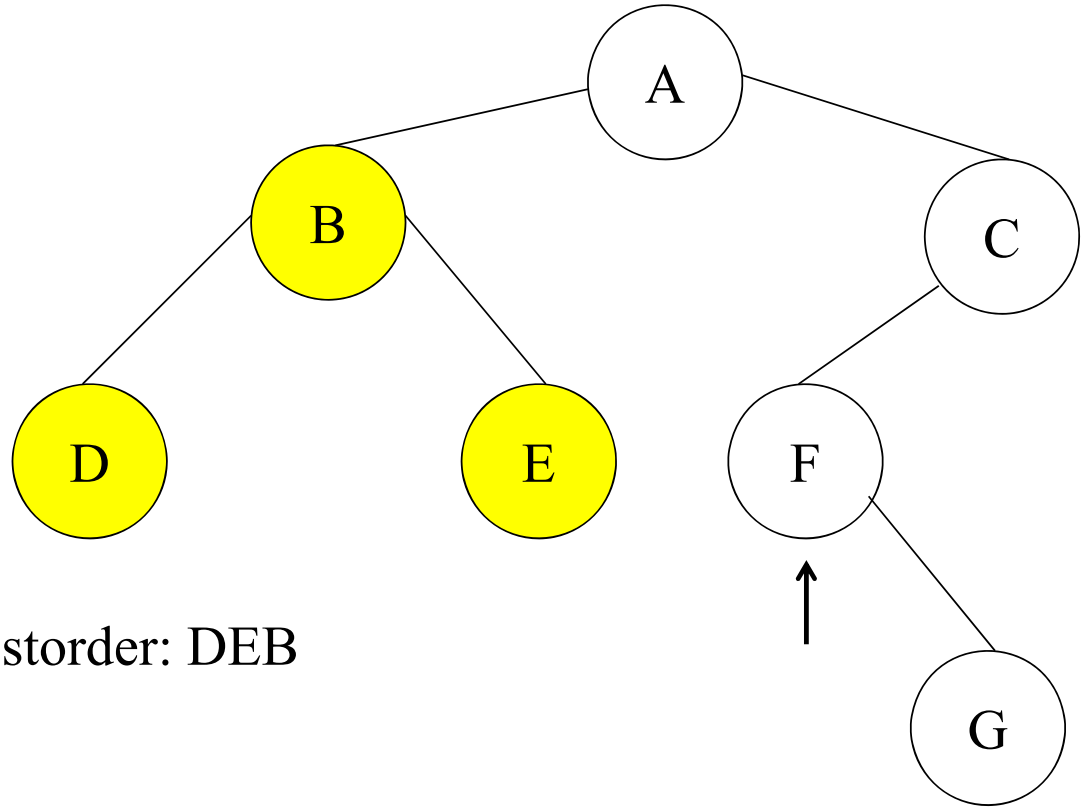
Postorder: DE**B**

Tree Traversals: An Example



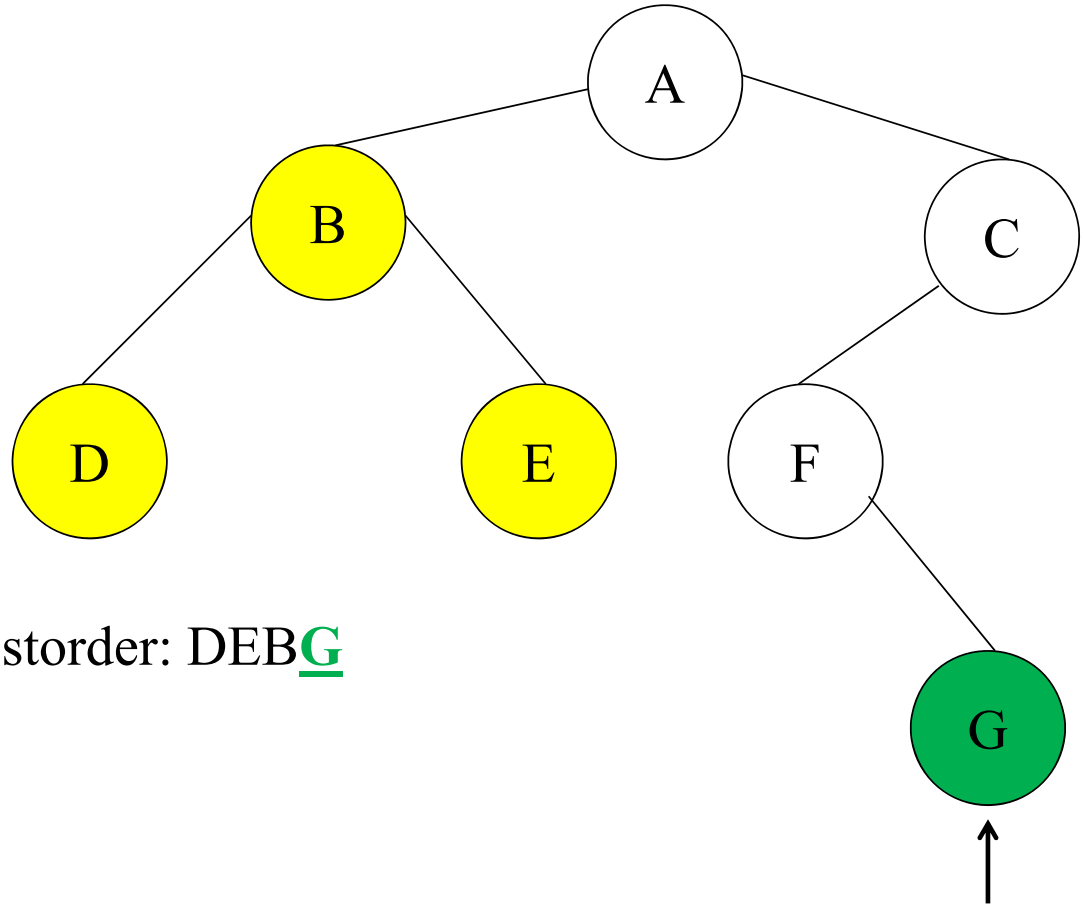
Postorder: DEB

Tree Traversals: An Example



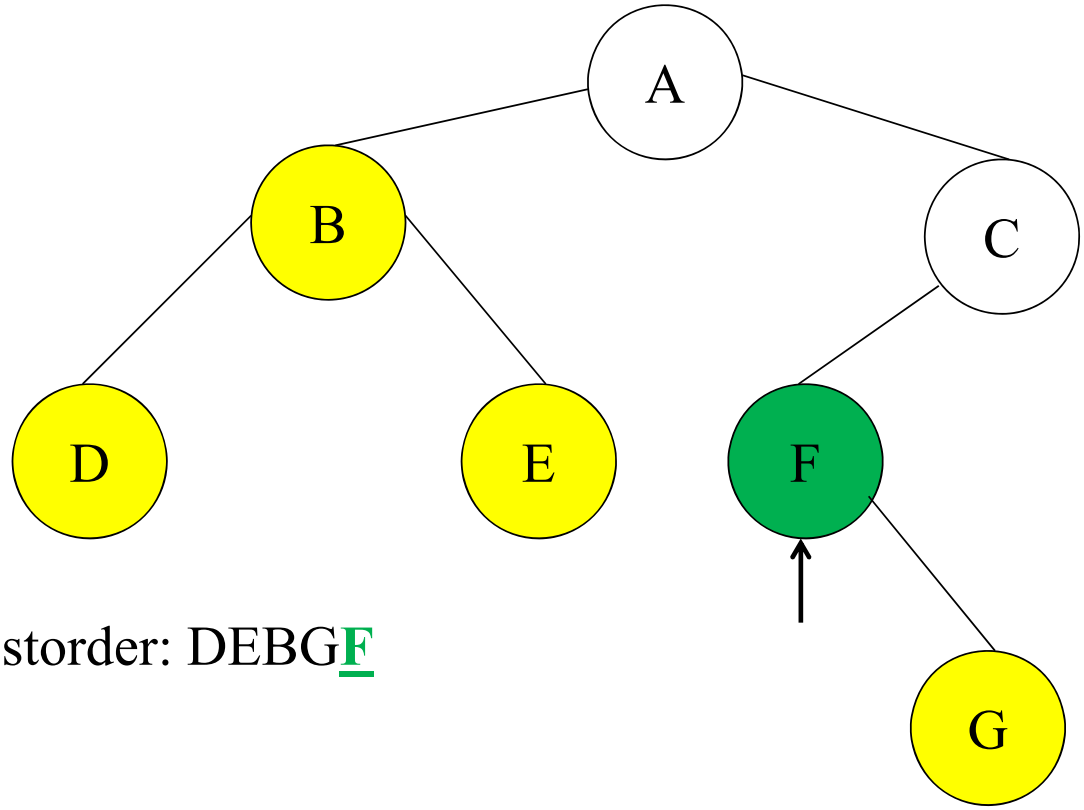
Postorder: DEB

Tree Traversals: An Example



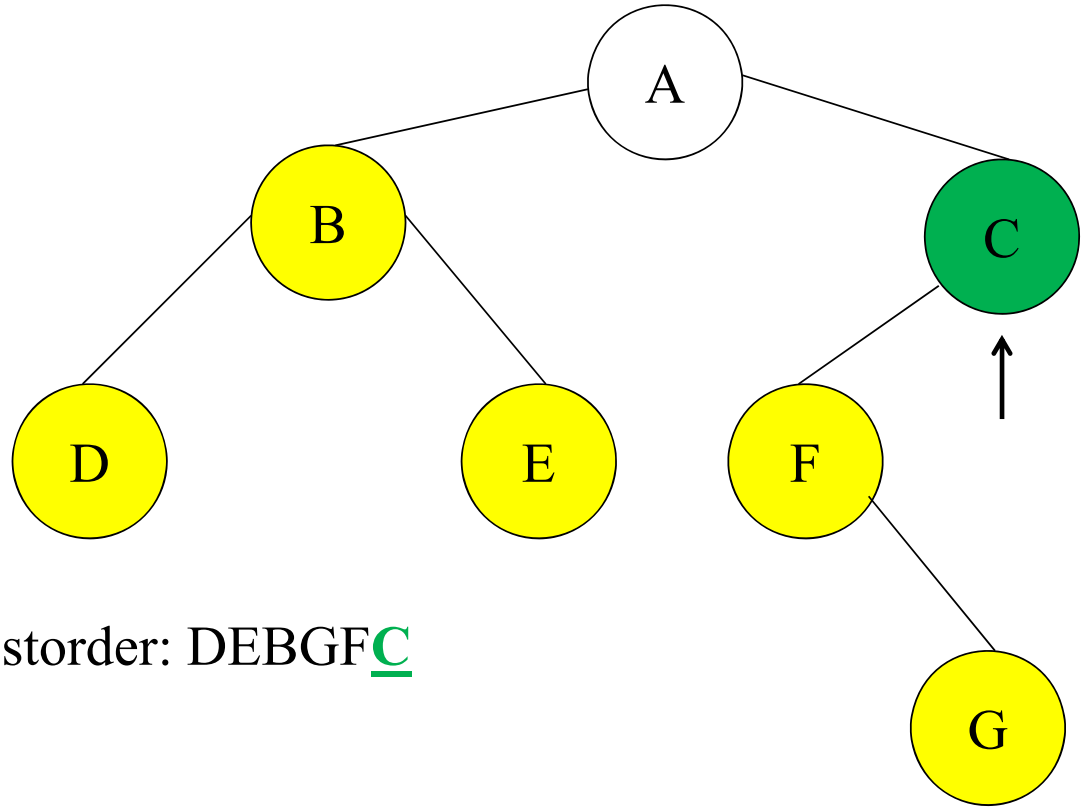
Postorder: DEBG

Tree Traversals: An Example



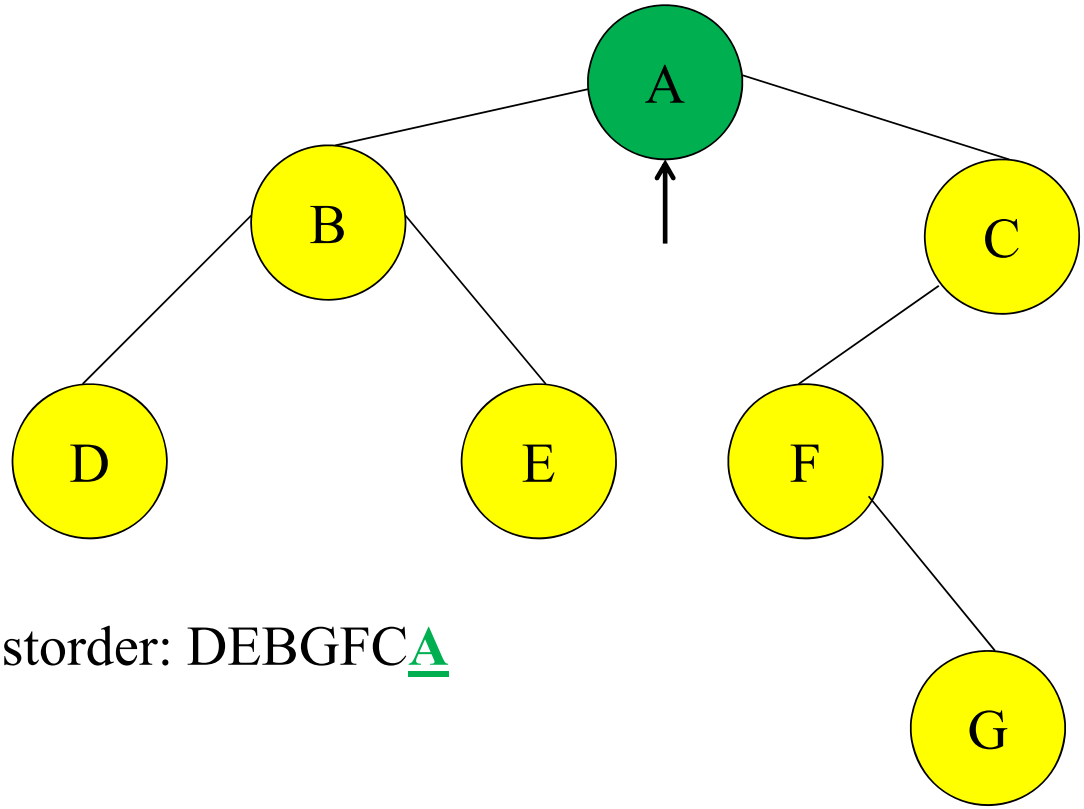
Postorder: DEBGF

Tree Traversals: An Example



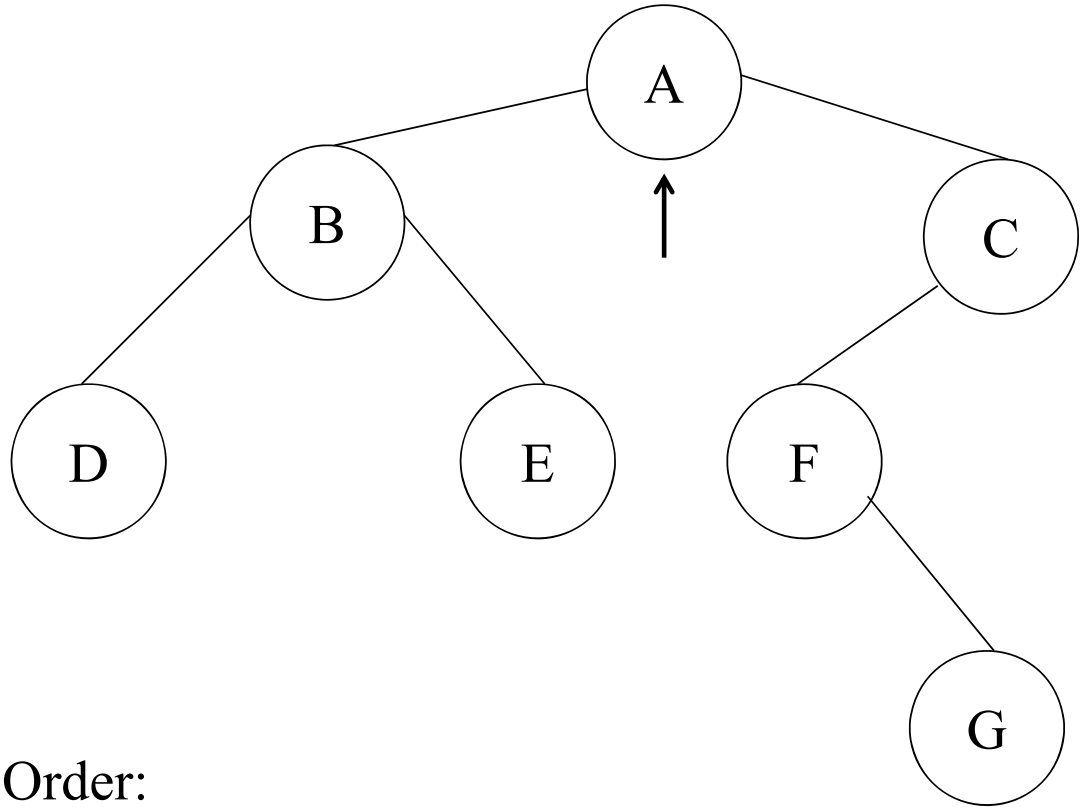
Postorder: DEBGFC

Tree Traversals: An Example



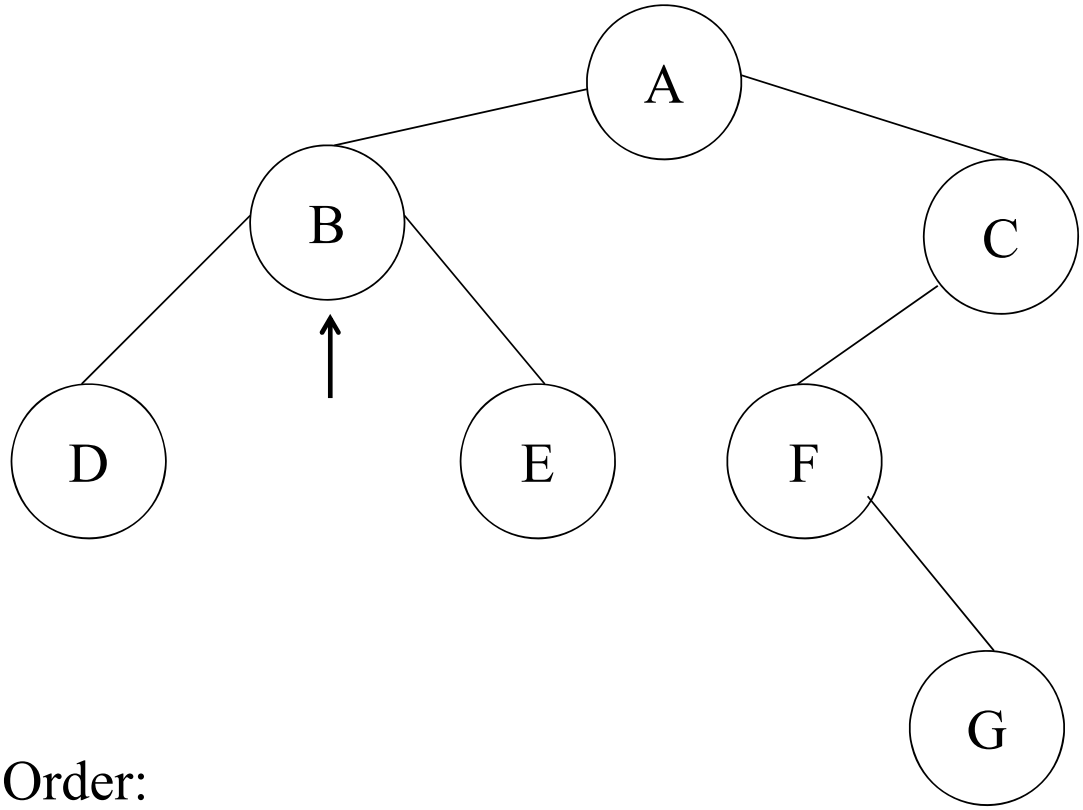
Postorder: DEBGFC**A**

Tree Traversals: An Example



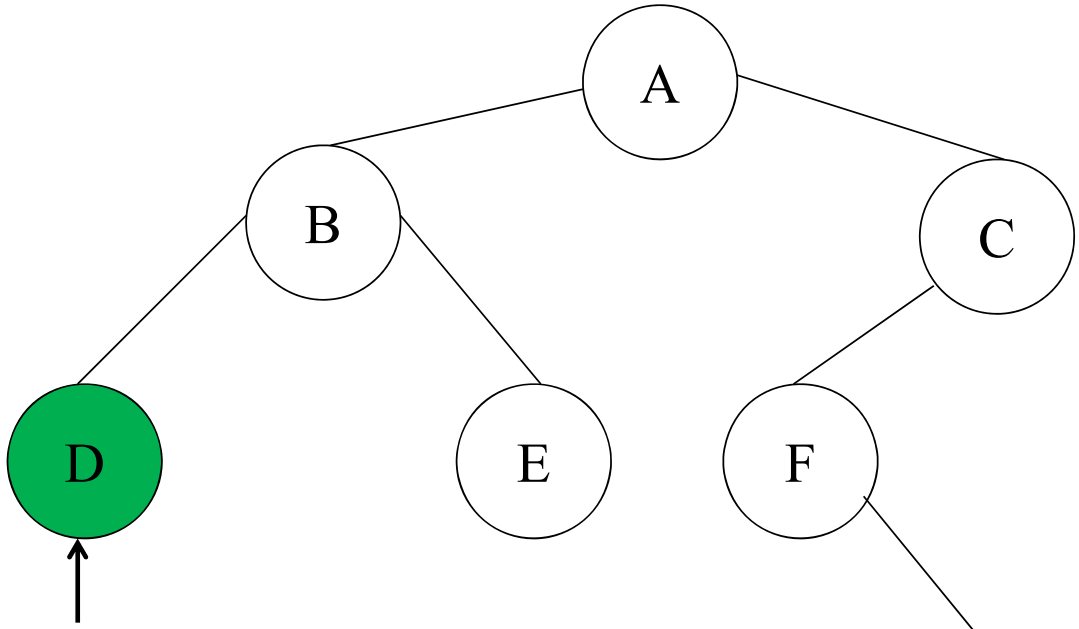
In Order:

Tree Traversals: An Example



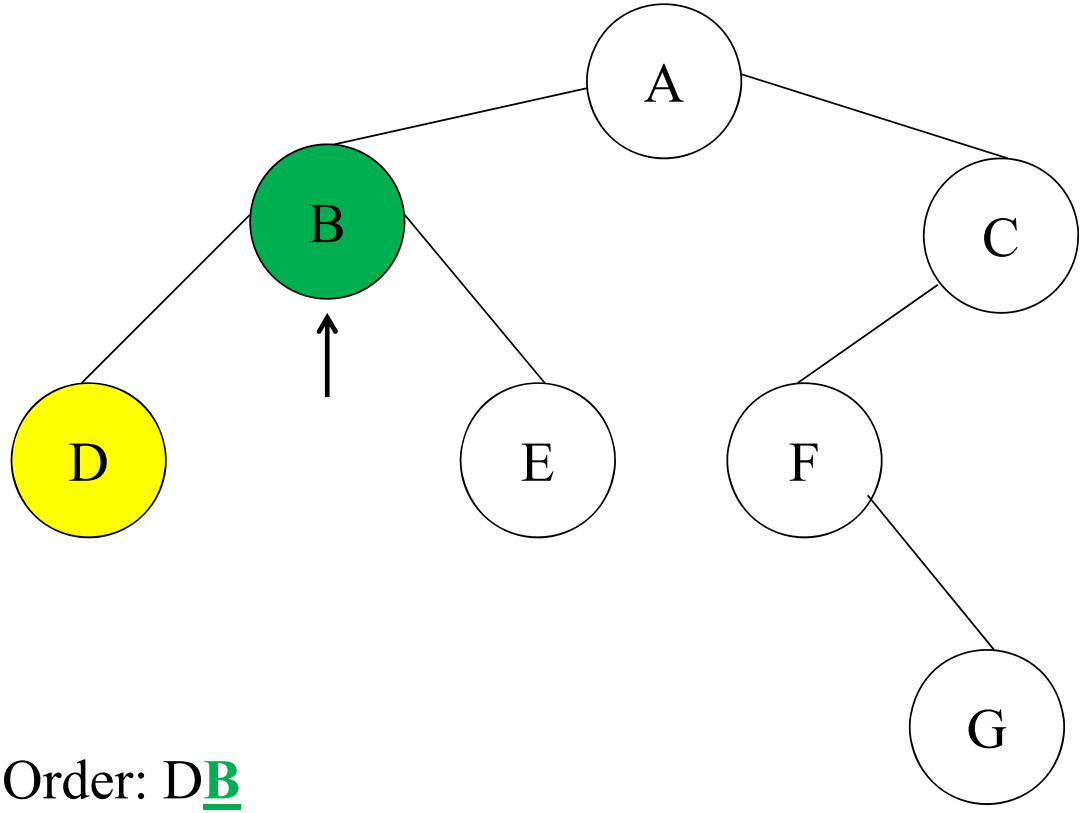
In Order:

Tree Traversals: An Example



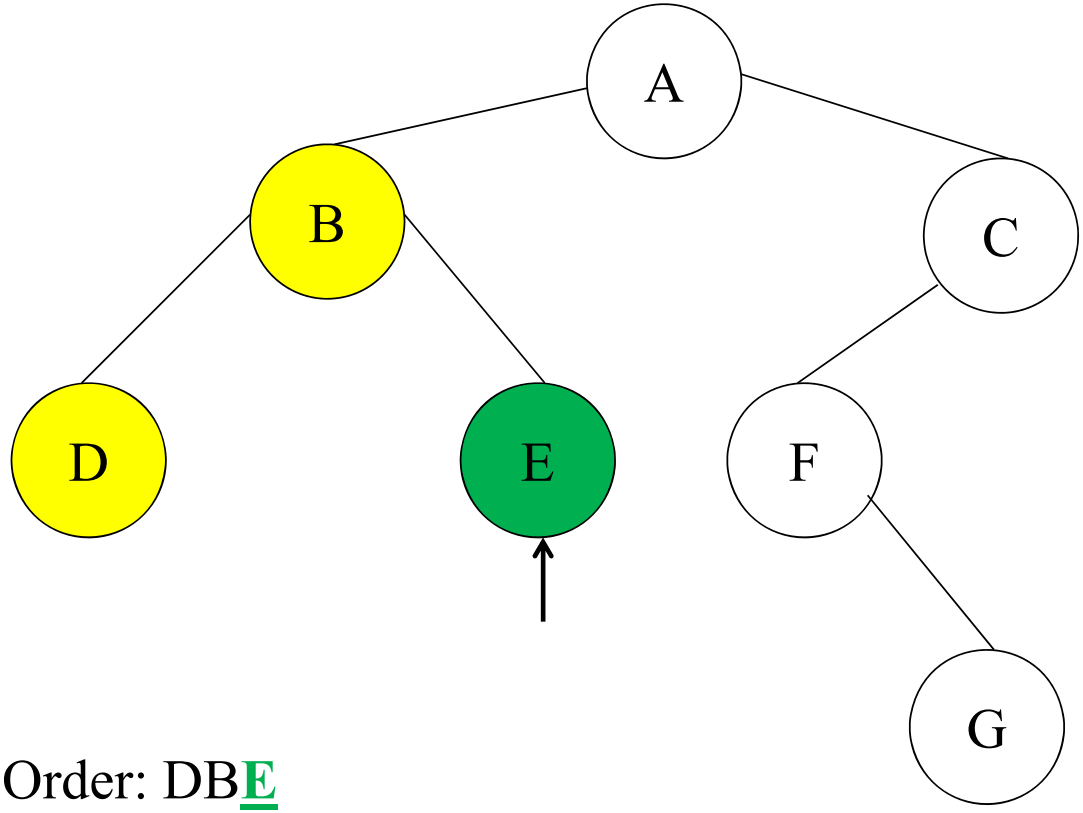
In Order: **D**

Tree Traversals: An Example



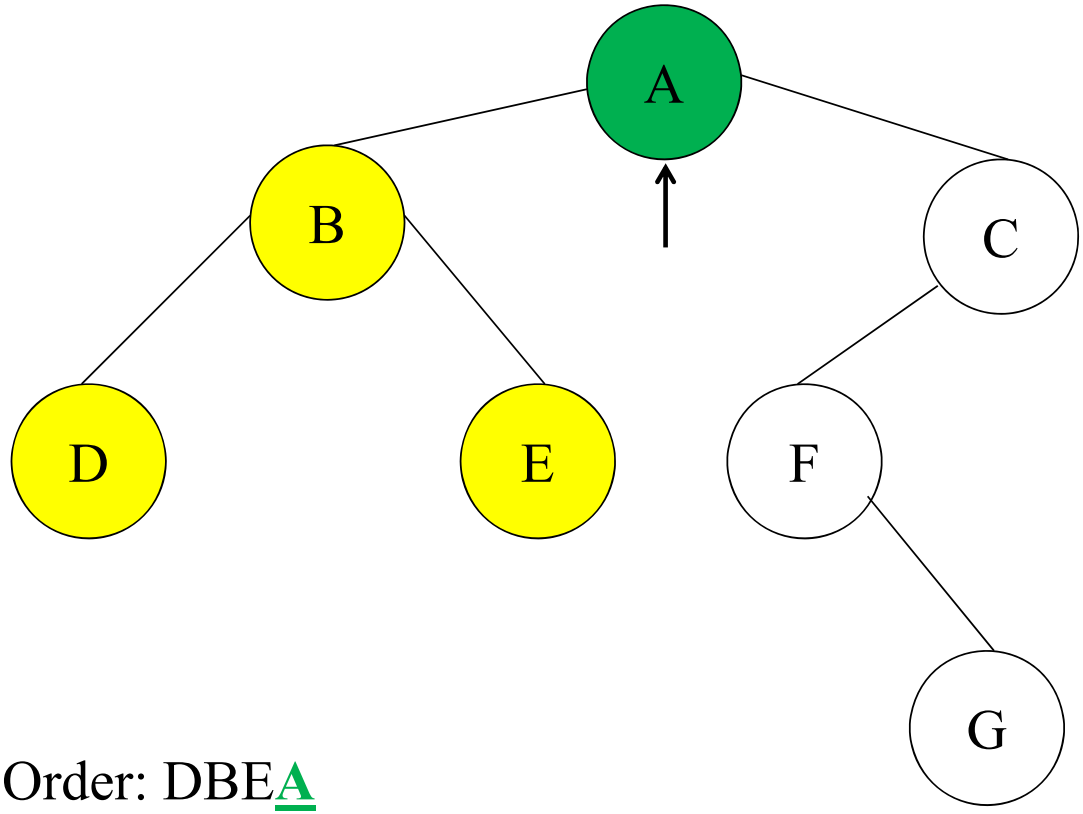
In Order: D**B**

Tree Traversals: An Example



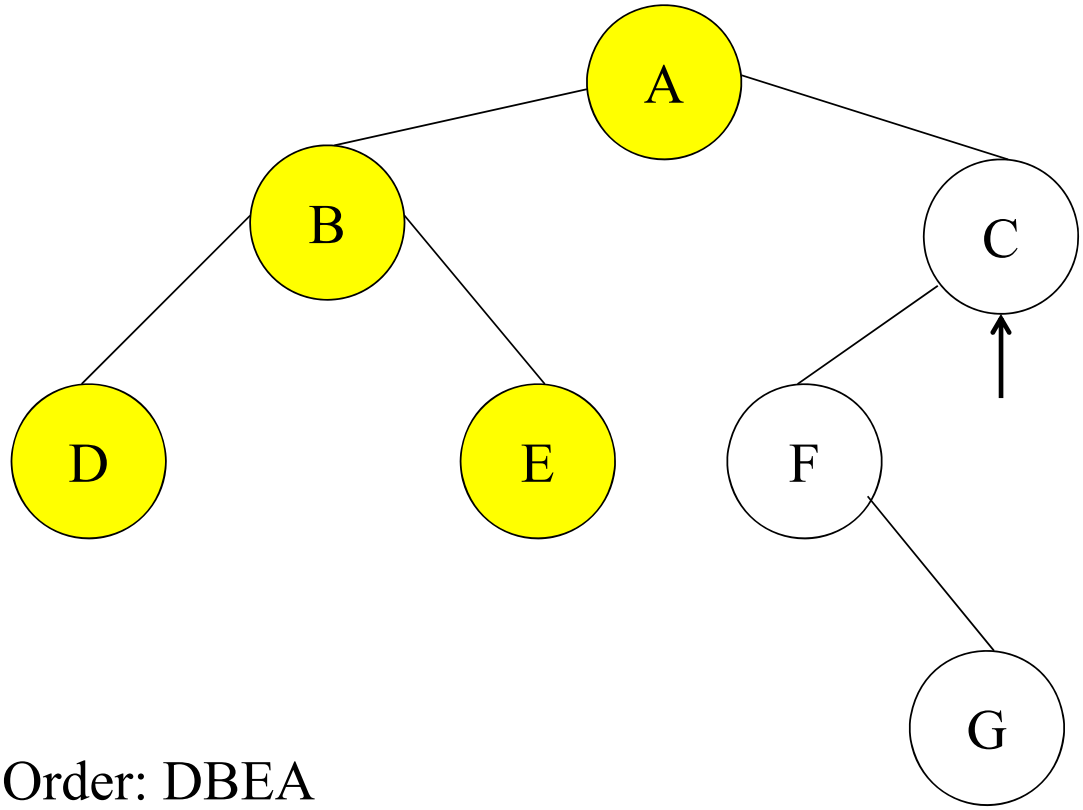
In Order: DB**E**

Tree Traversals: An Example



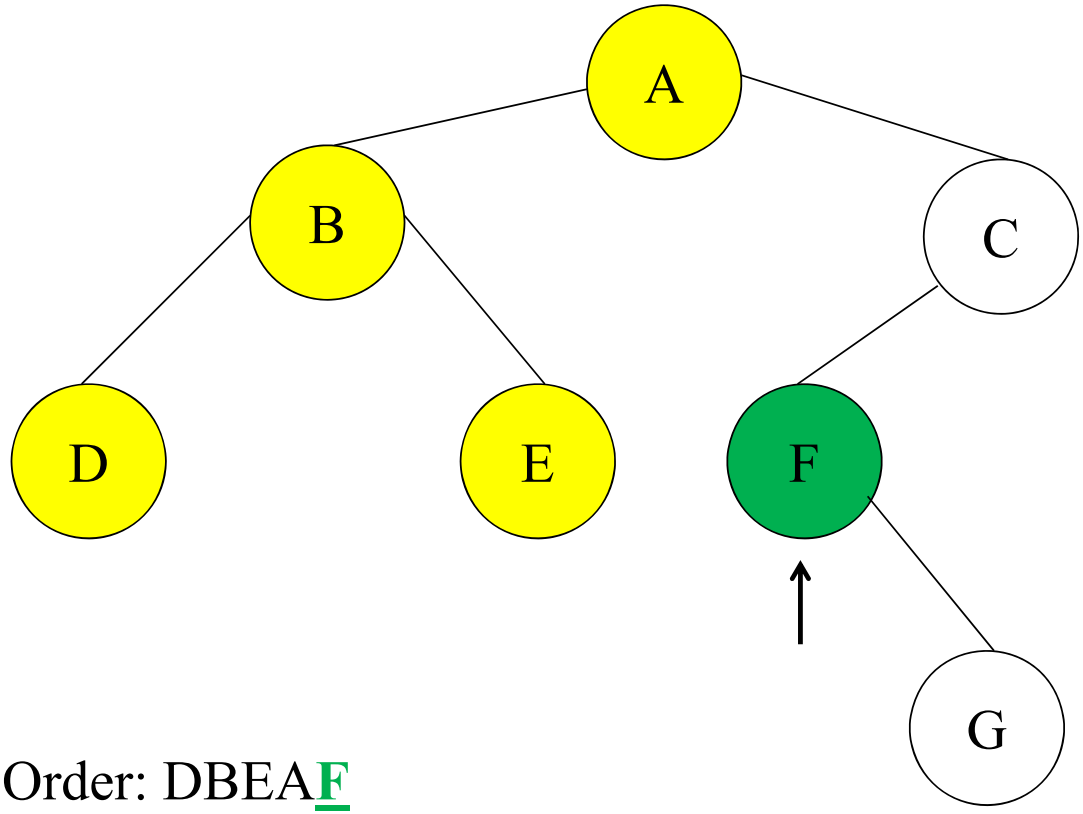
In Order: DBE**A**

Tree Traversals: An Example



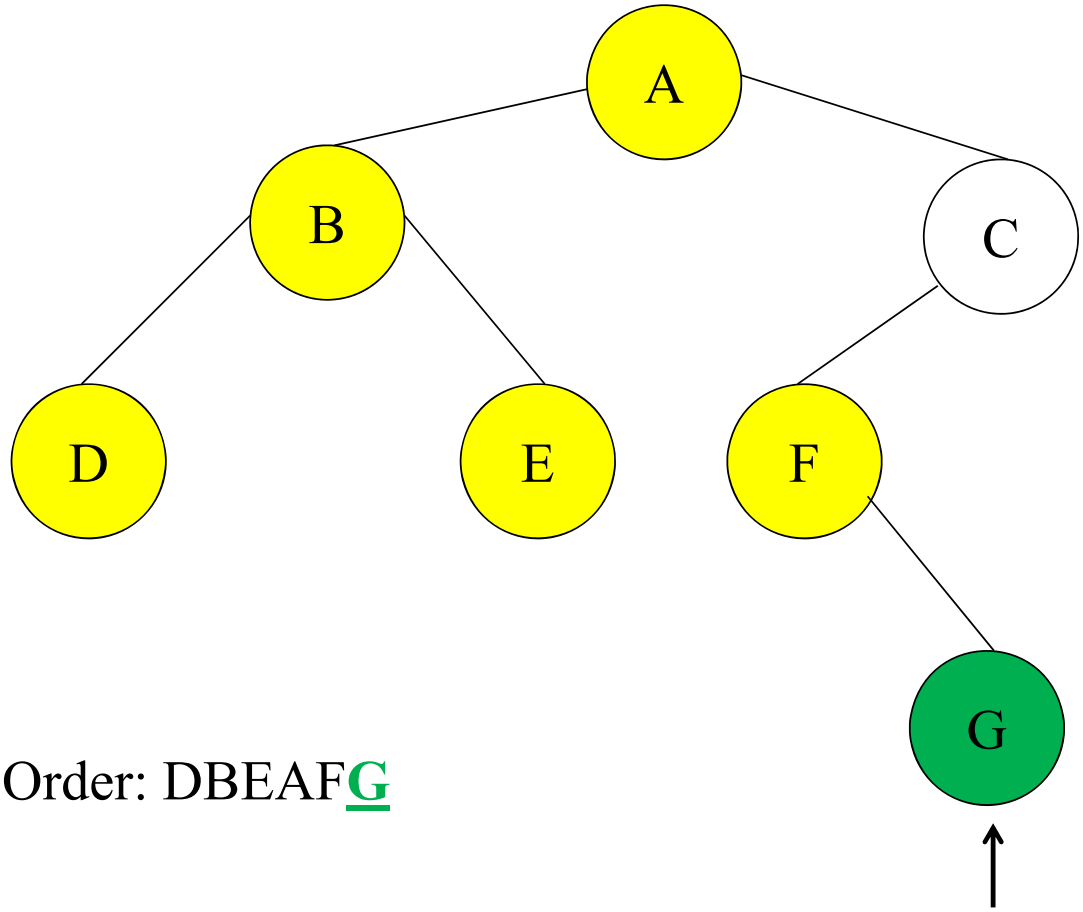
In Order: DBEA

Tree Traversals: An Example



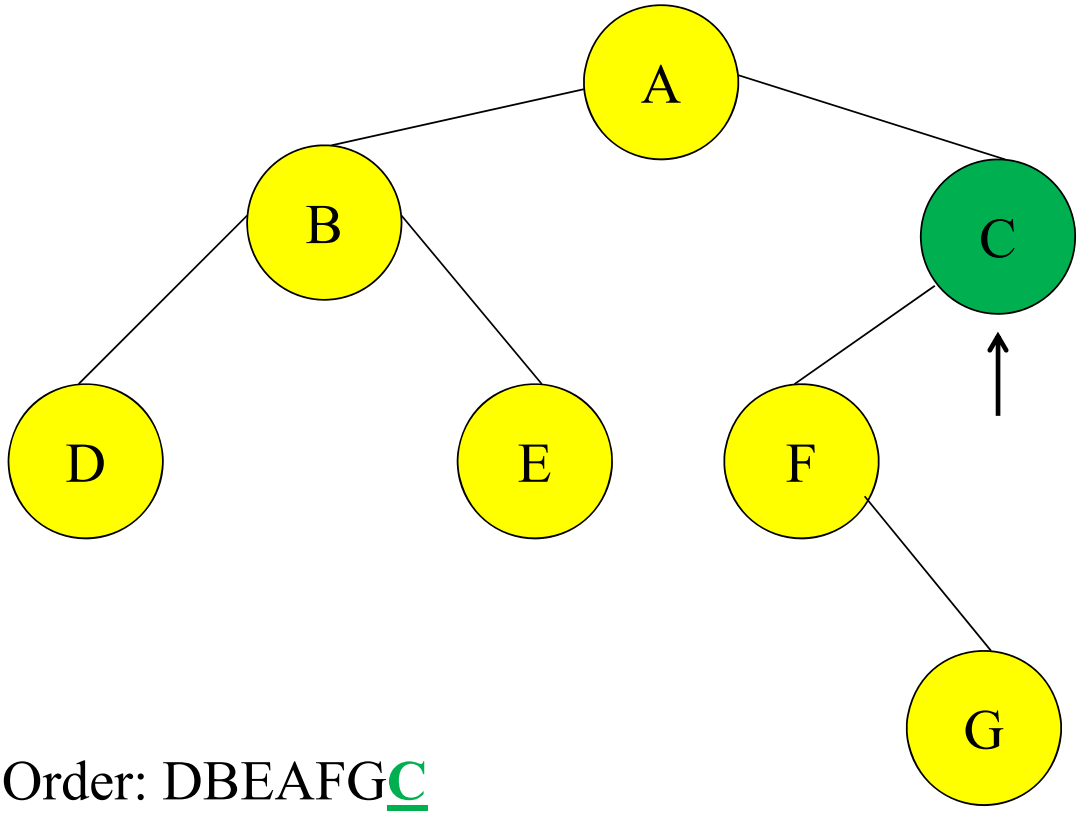
In Order: DBEA**F**

Tree Traversals: An Example



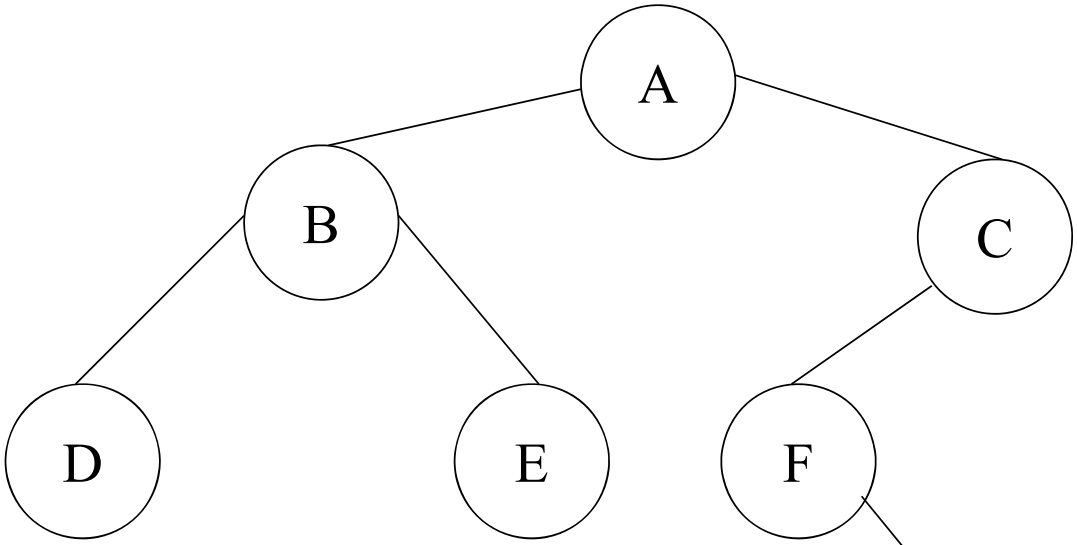
In Order: DBEAF**G**

Tree Traversals: An Example



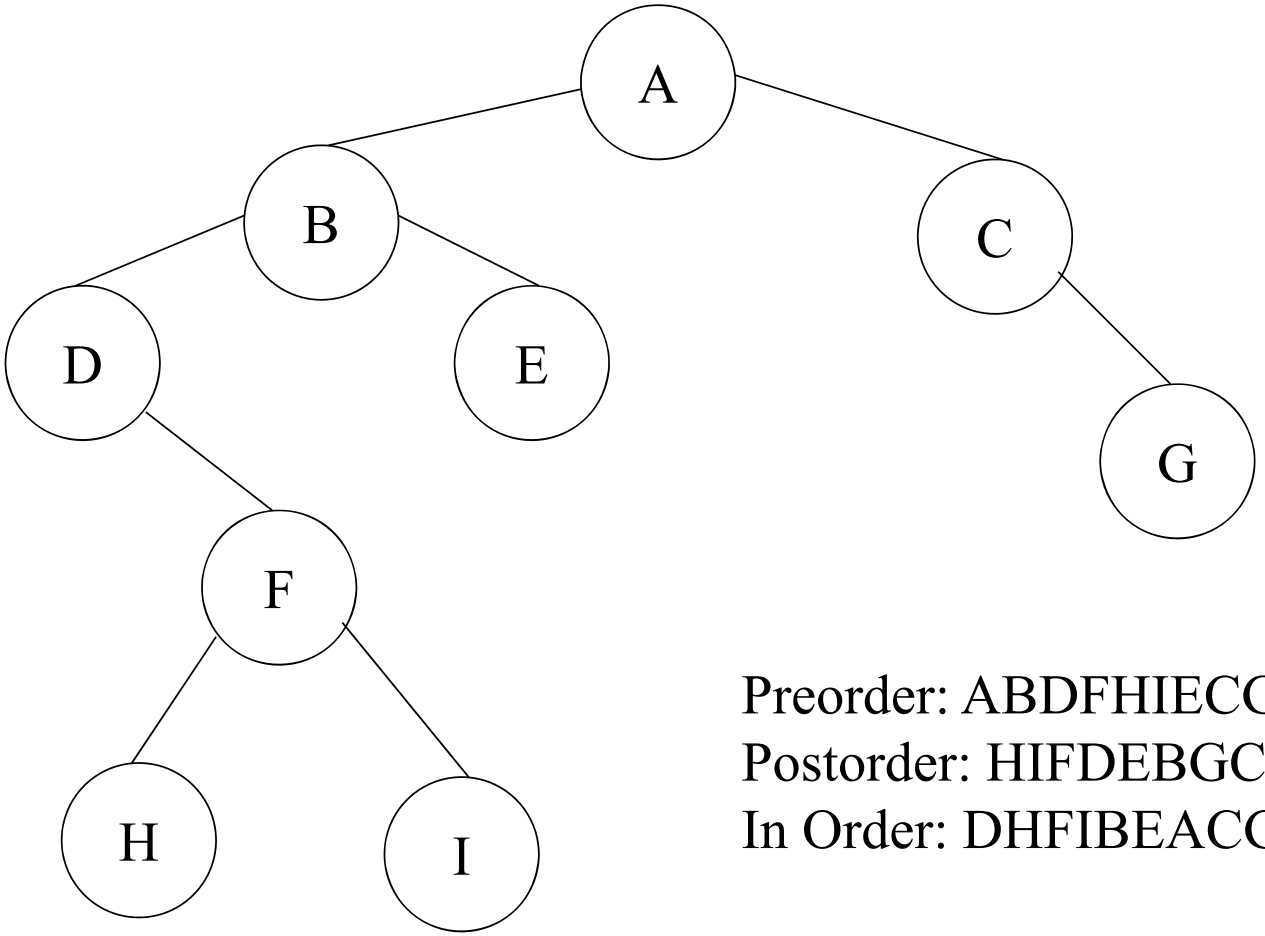
In Order: DBEAFGC

Tree Traversals: An Example



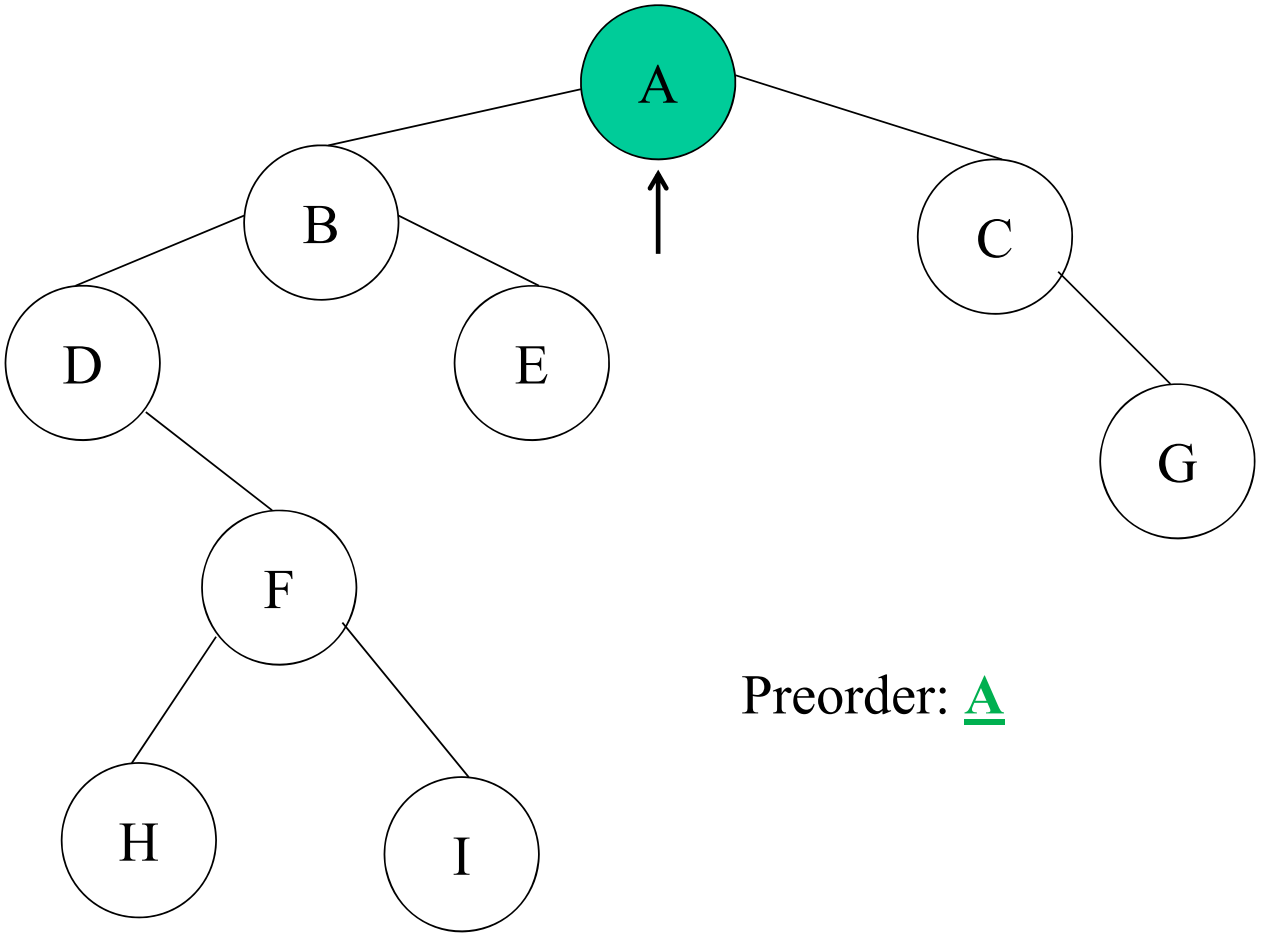
Preorder: ABDECFG
Postorder: DEBGFCA
In Order: DBEAFGC

Tree Traversals: Another Example



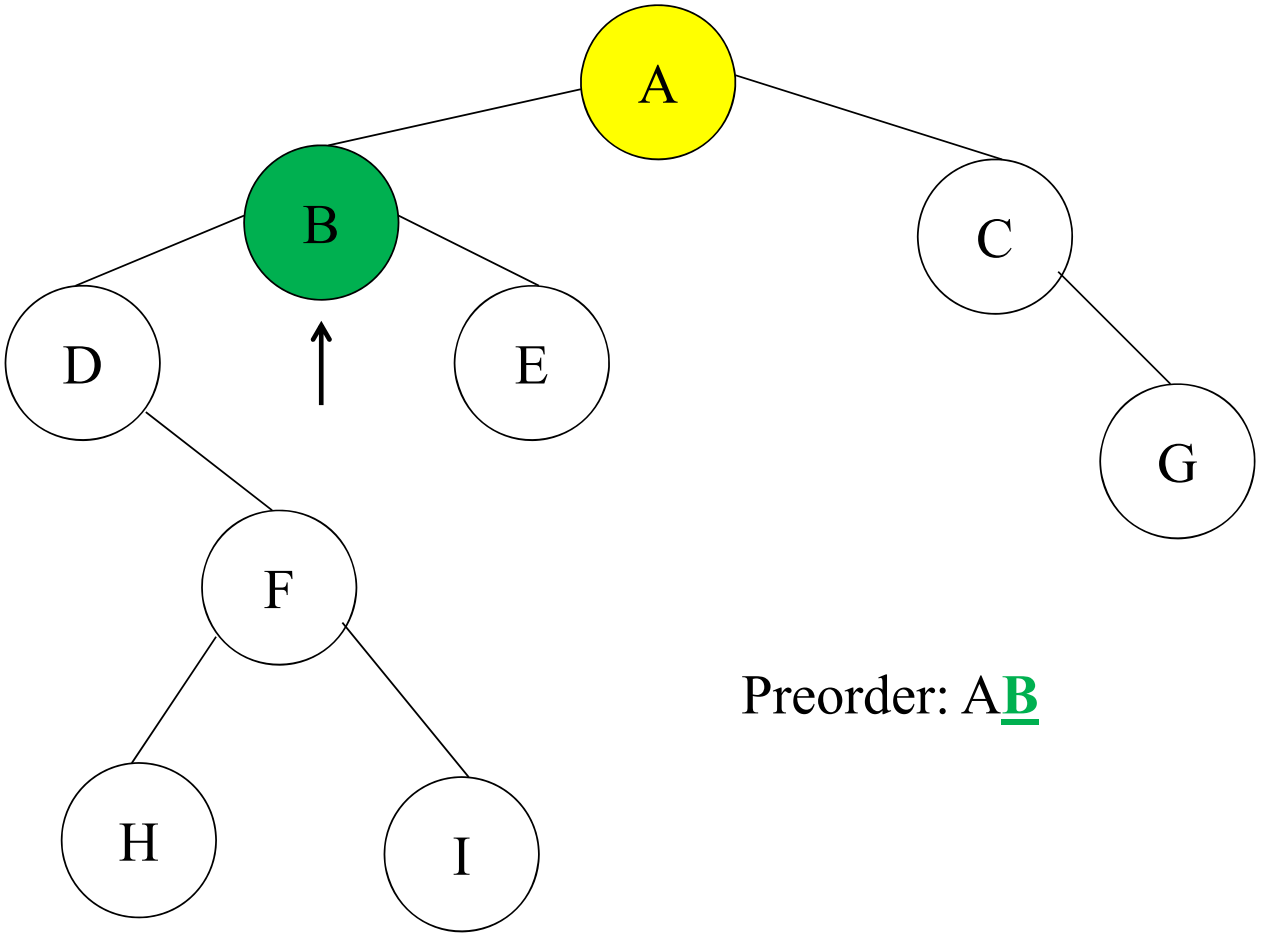
Preorder: ABDFHIECG
Postorder: HIFDEBGCA
In Order: DHFIBEACG

Tree Traversals: Another Example



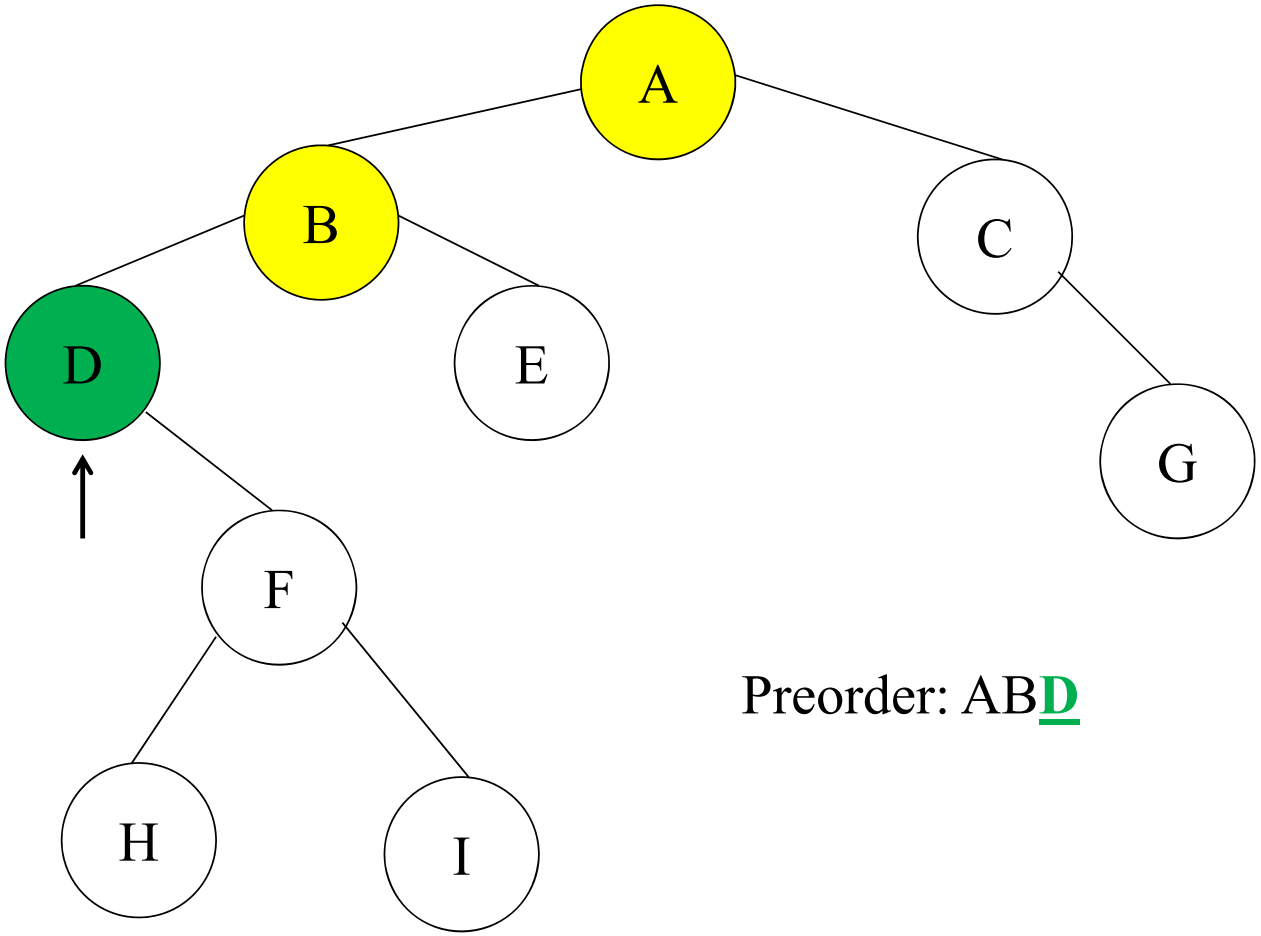
Preorder: A

Tree Traversals: Another Example



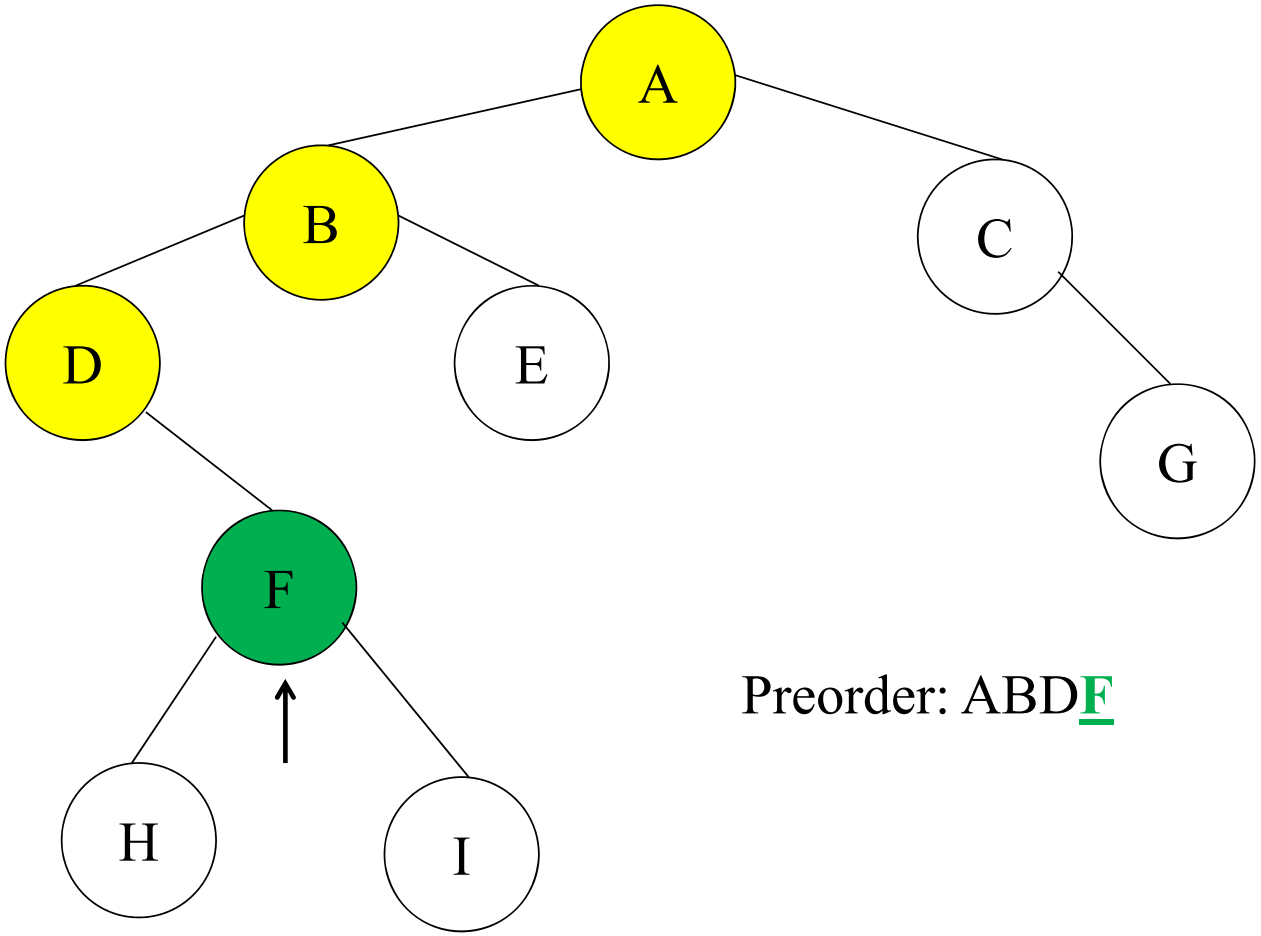
Preorder: A**B**

Tree Traversals: Another Example



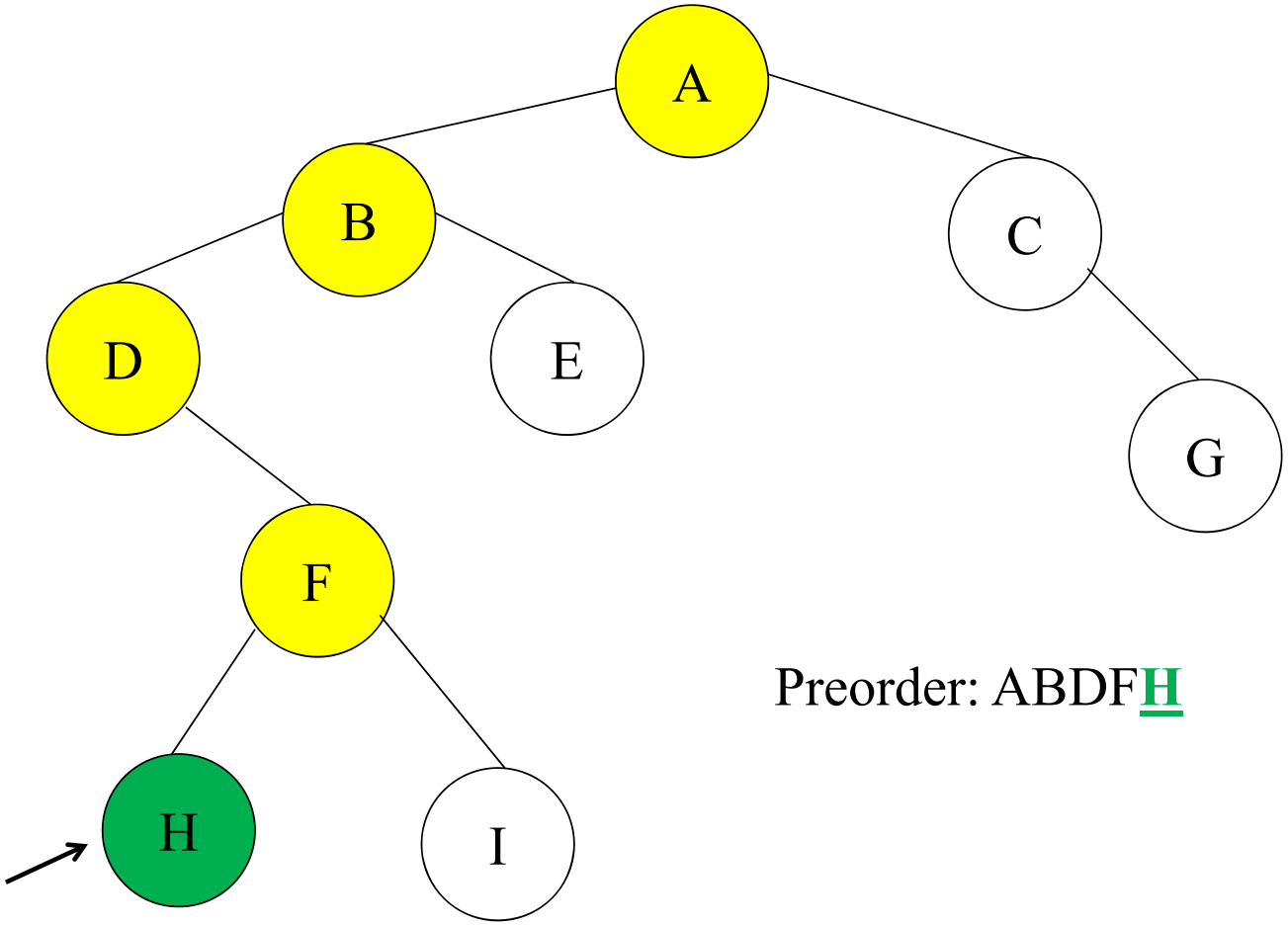
Preorder: AB**D**

Tree Traversals: Another Example



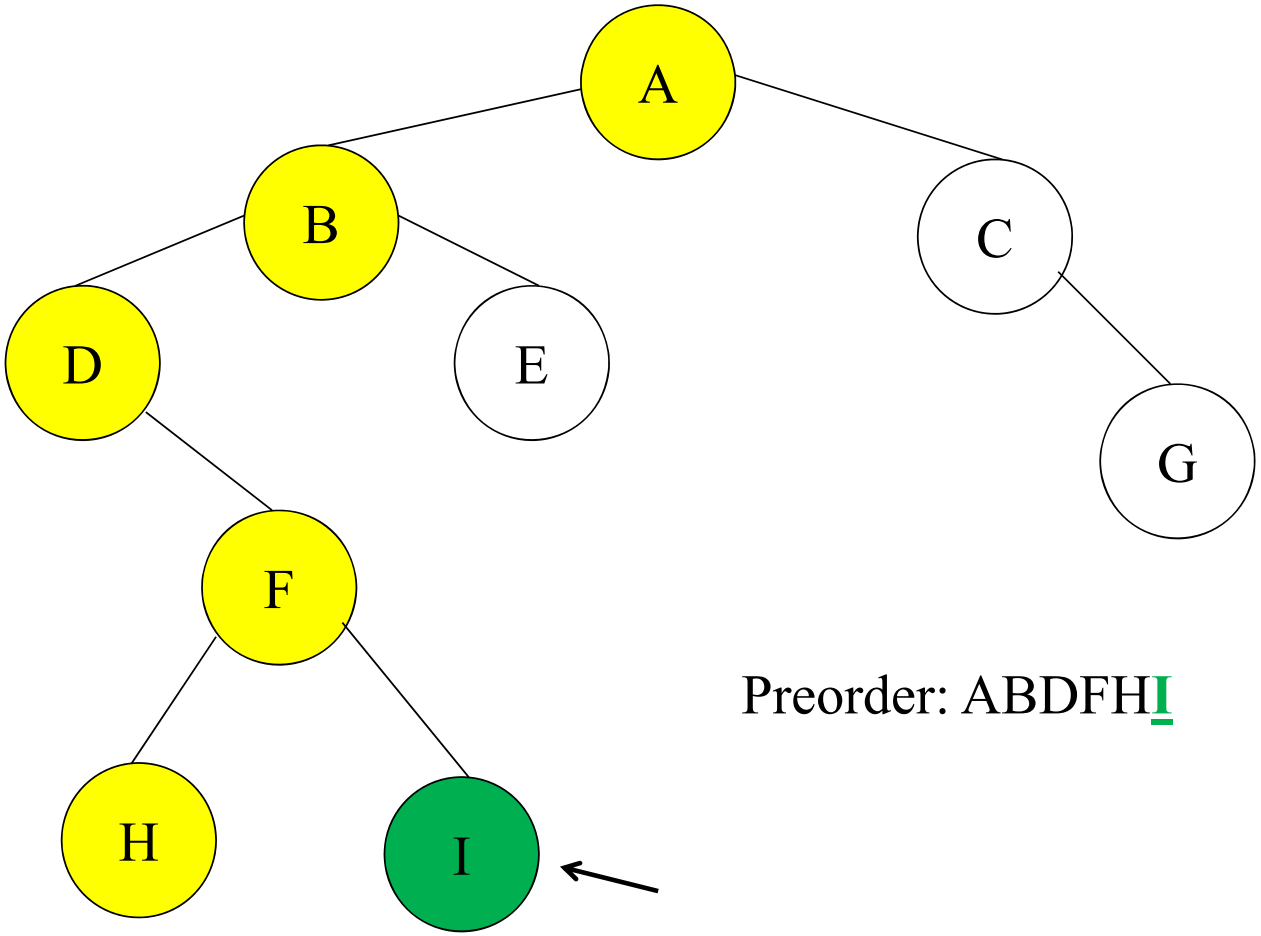
Preorder: ABD**F**

Tree Traversals: Another Example



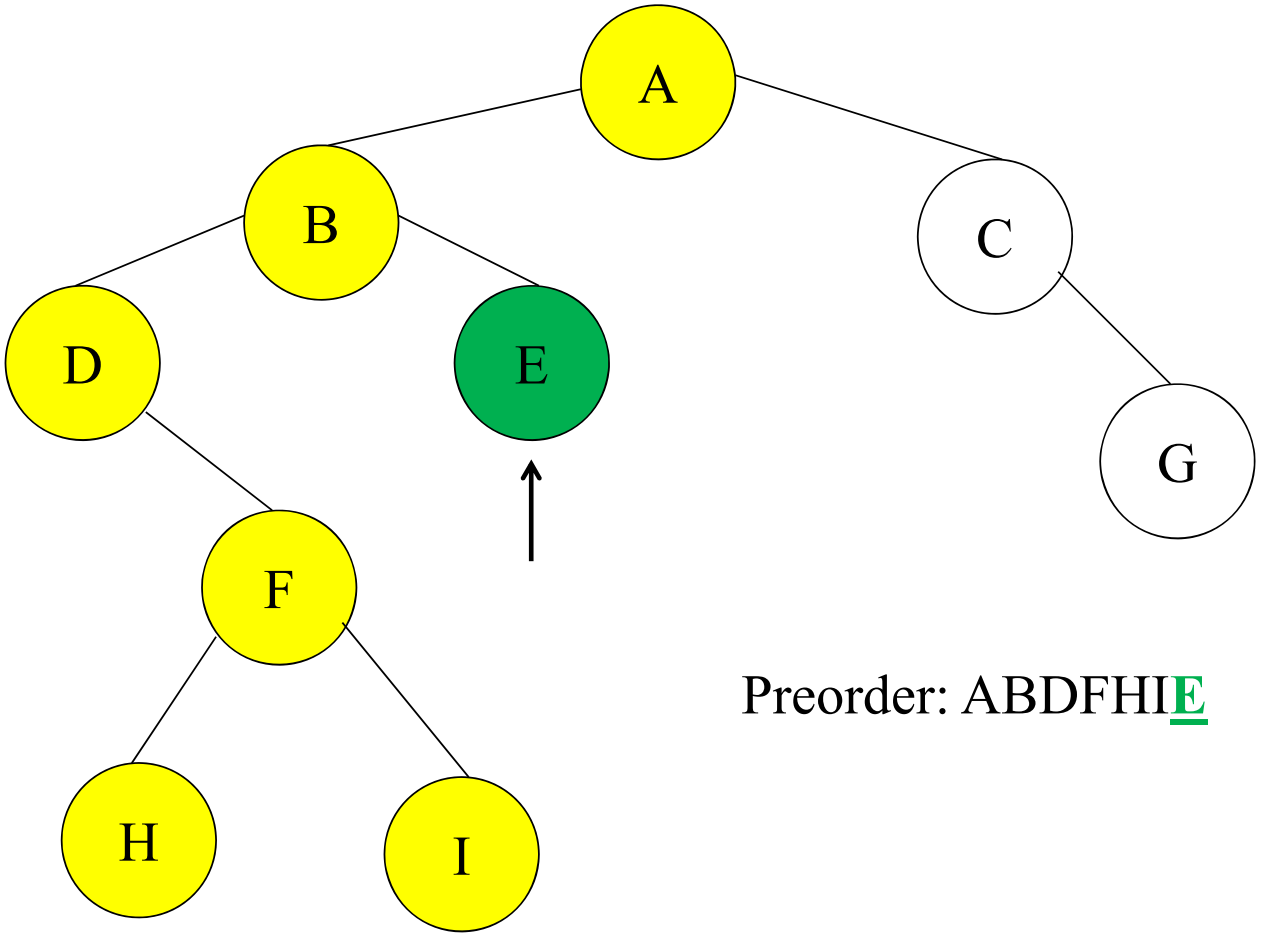
Preorder: ABDF**H**

Tree Traversals: Another Example



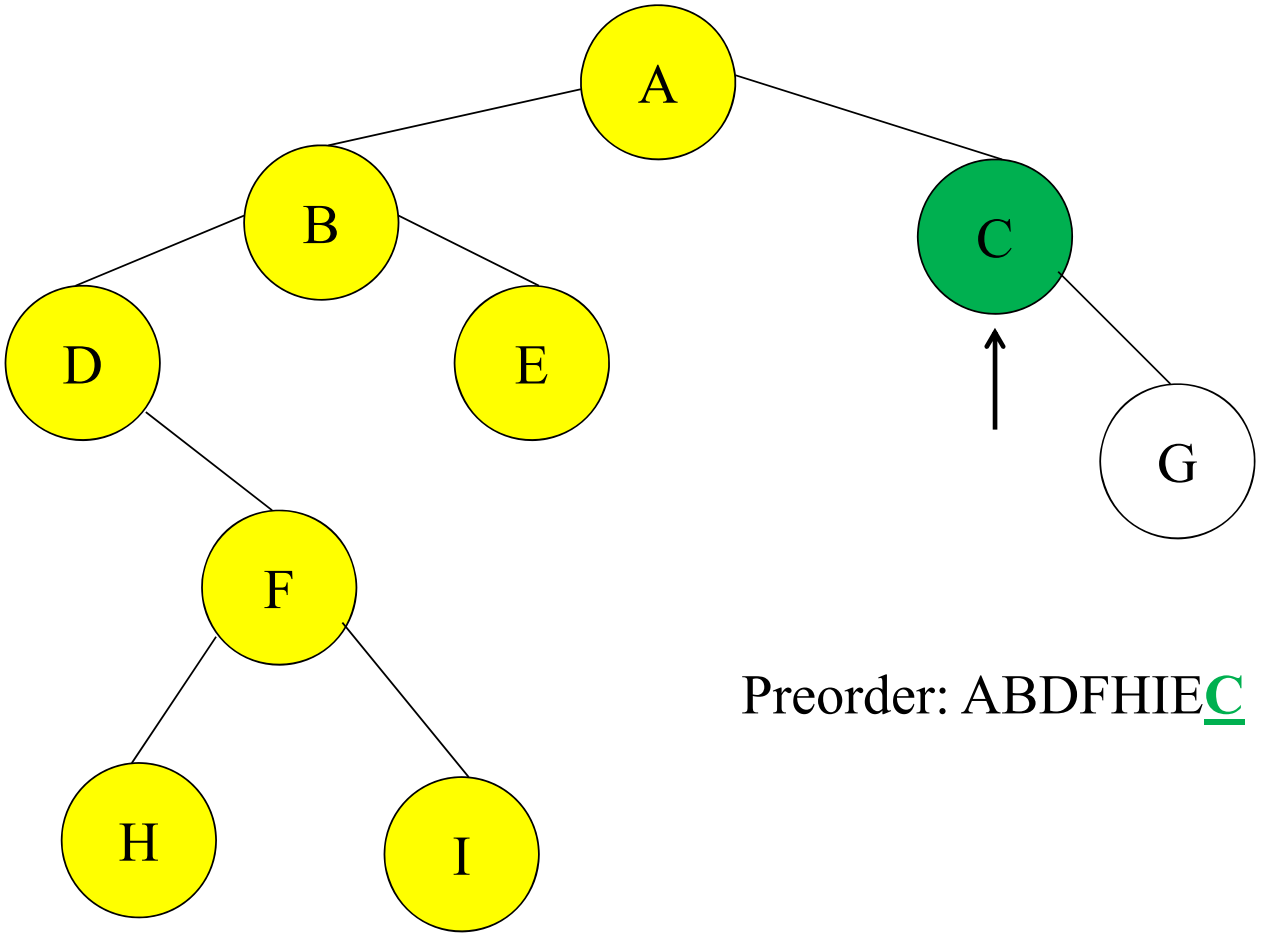
Preorder: ABDFHI

Tree Traversals: Another Example



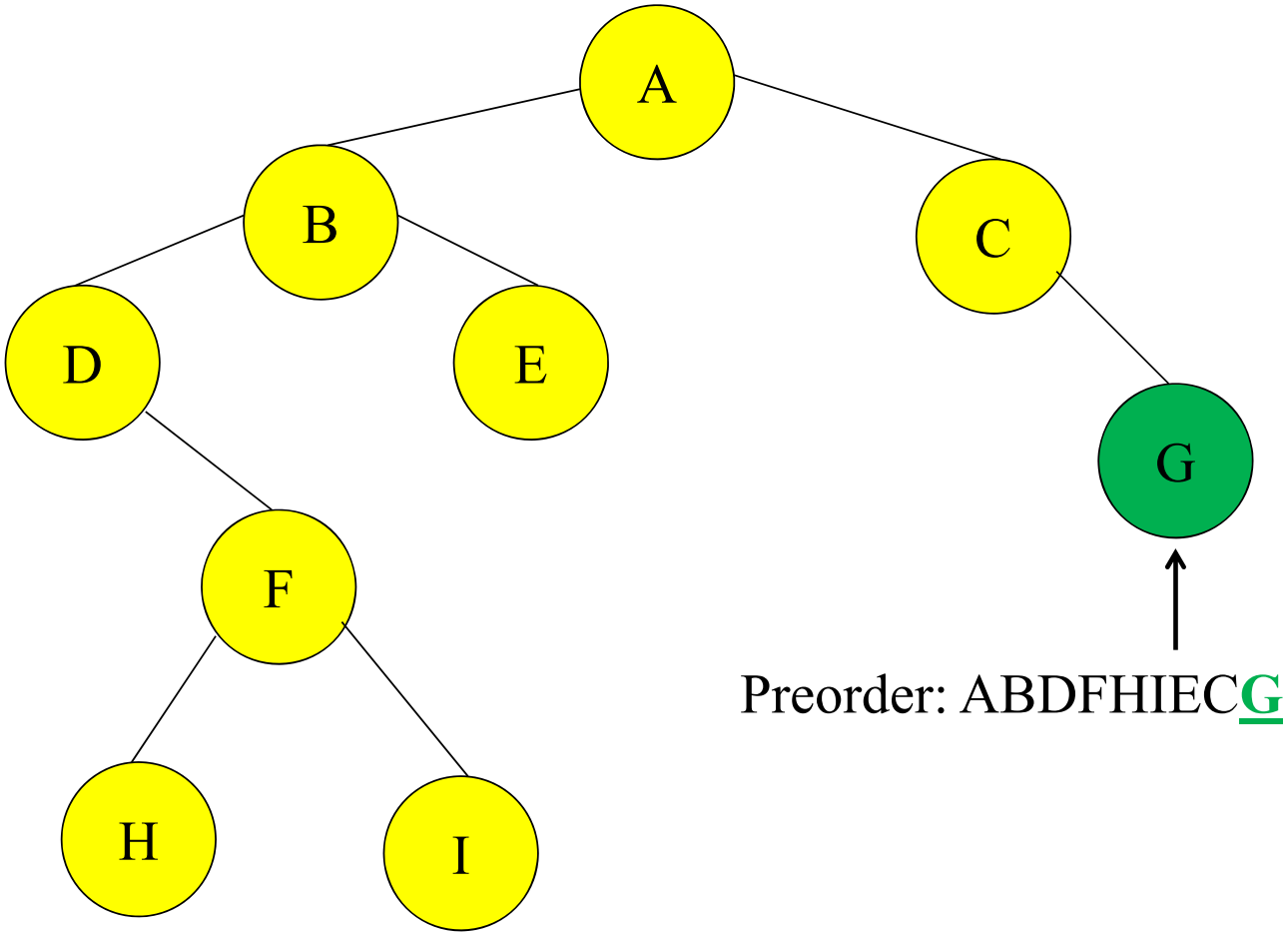
Preorder: ABDFHI**E**

Tree Traversals: Another Example

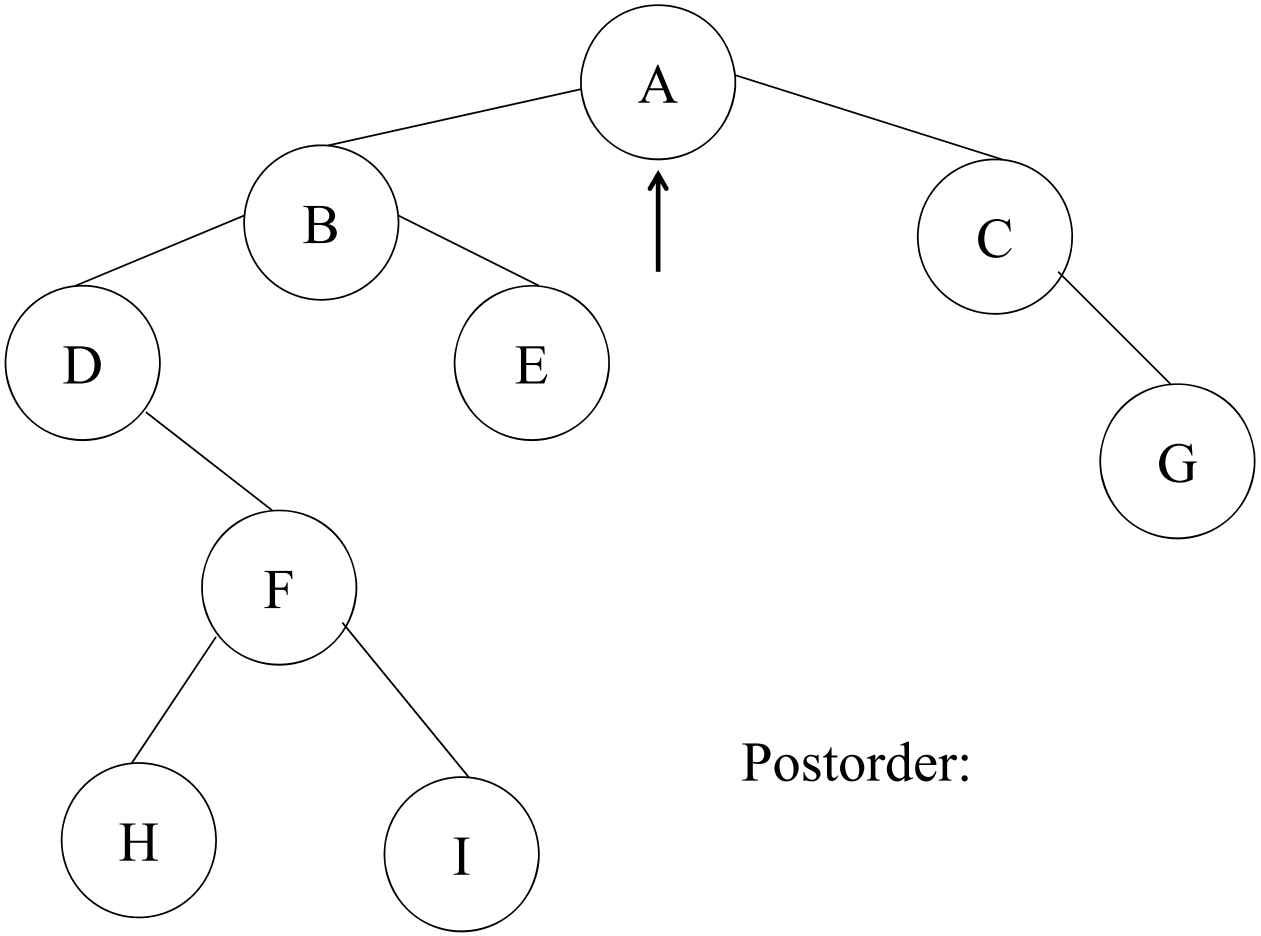


Preorder: ABDFHIE**C**

Tree Traversals: Another Example

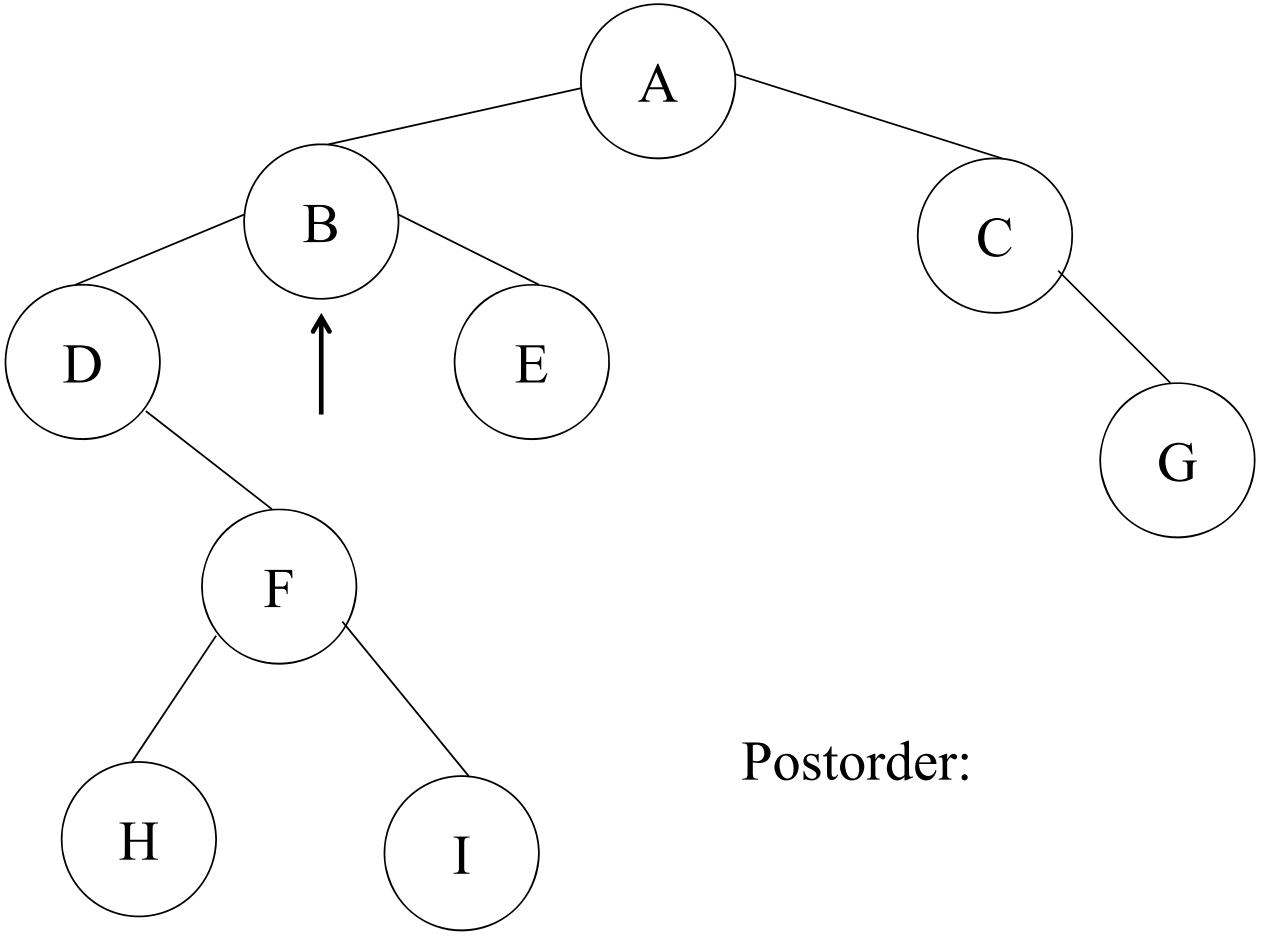


Tree Traversals: Another Example



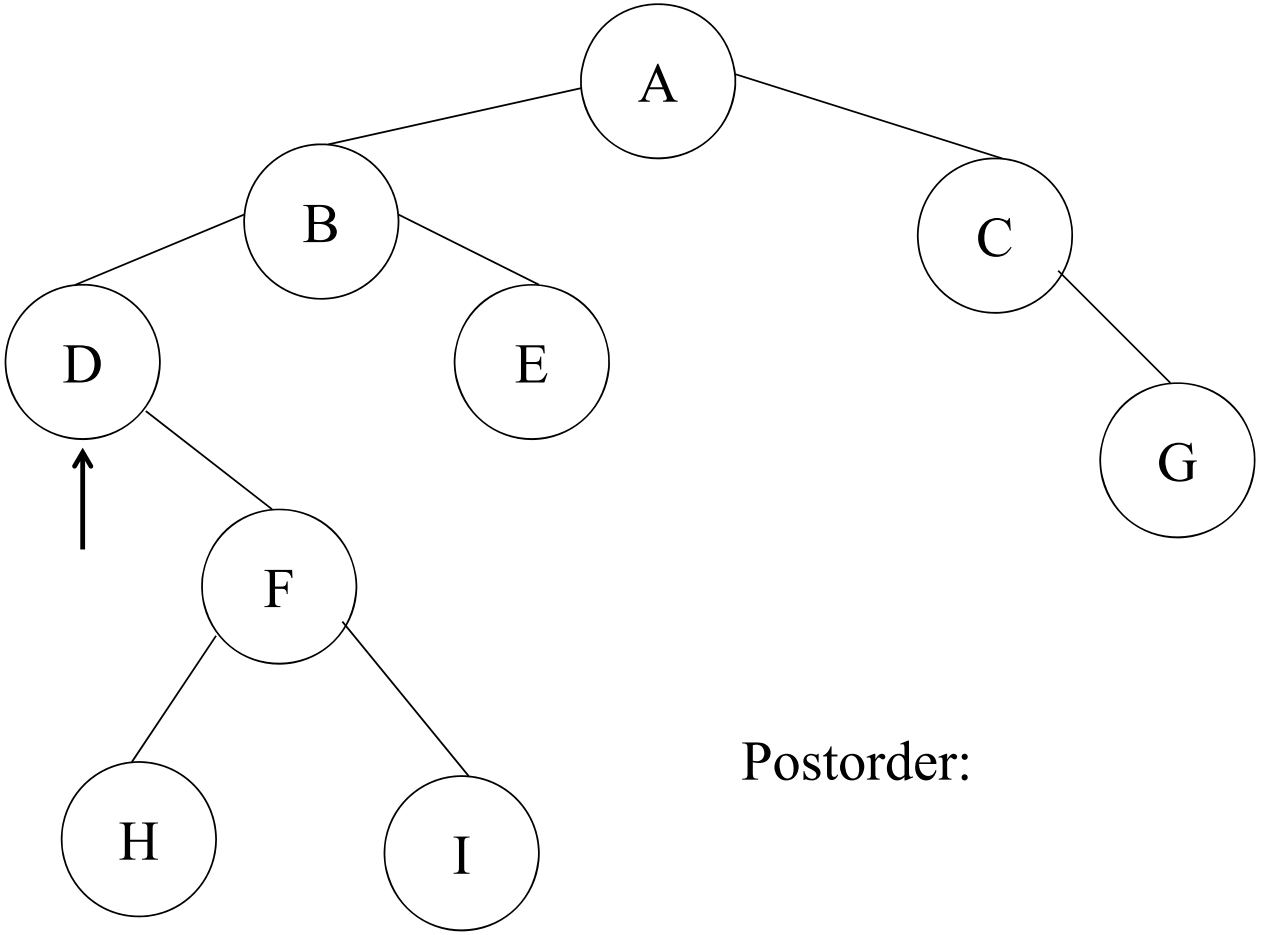
Postorder:

Tree Traversals: Another Example



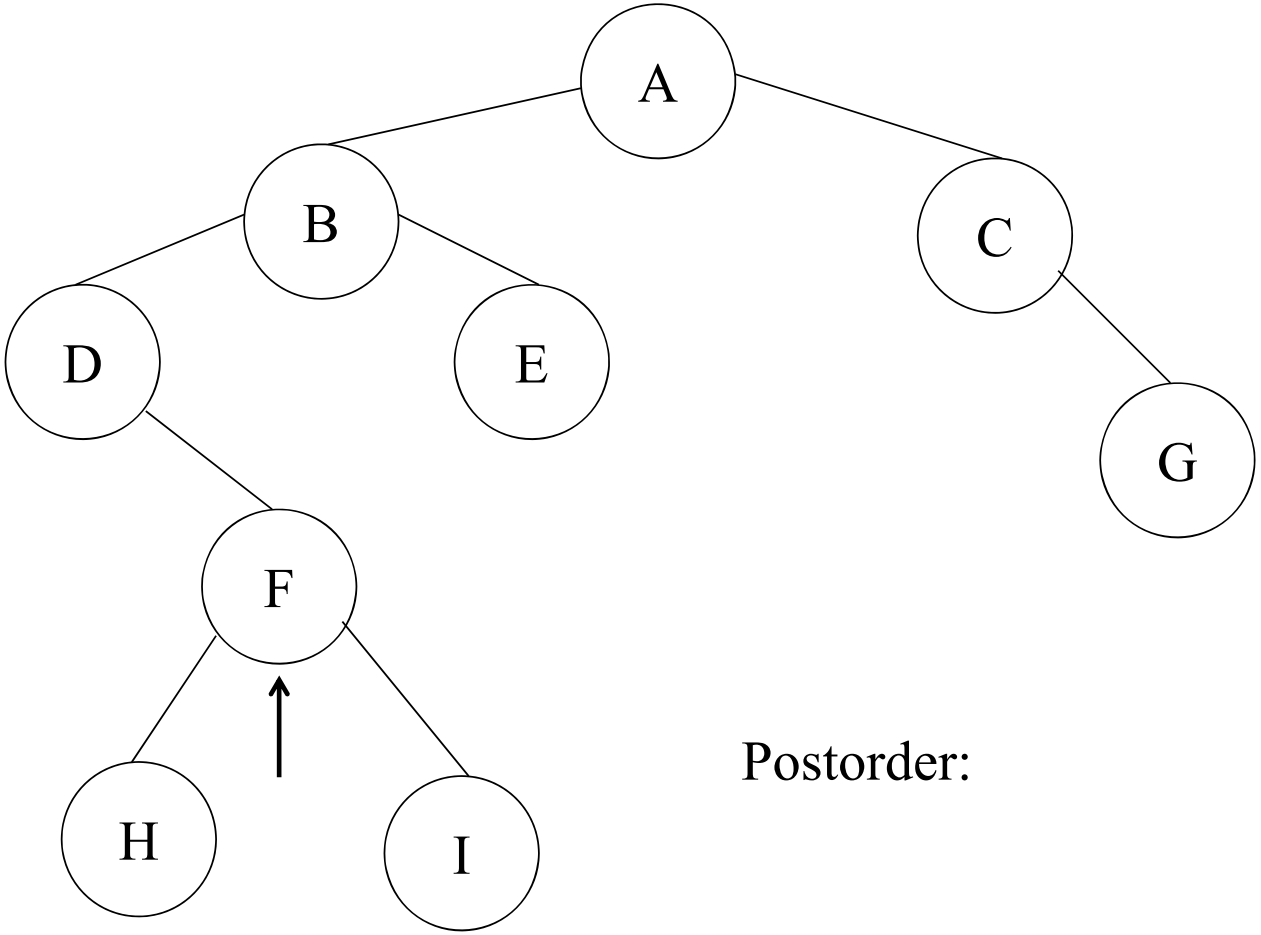
Postorder:

Tree Traversals: Another Example



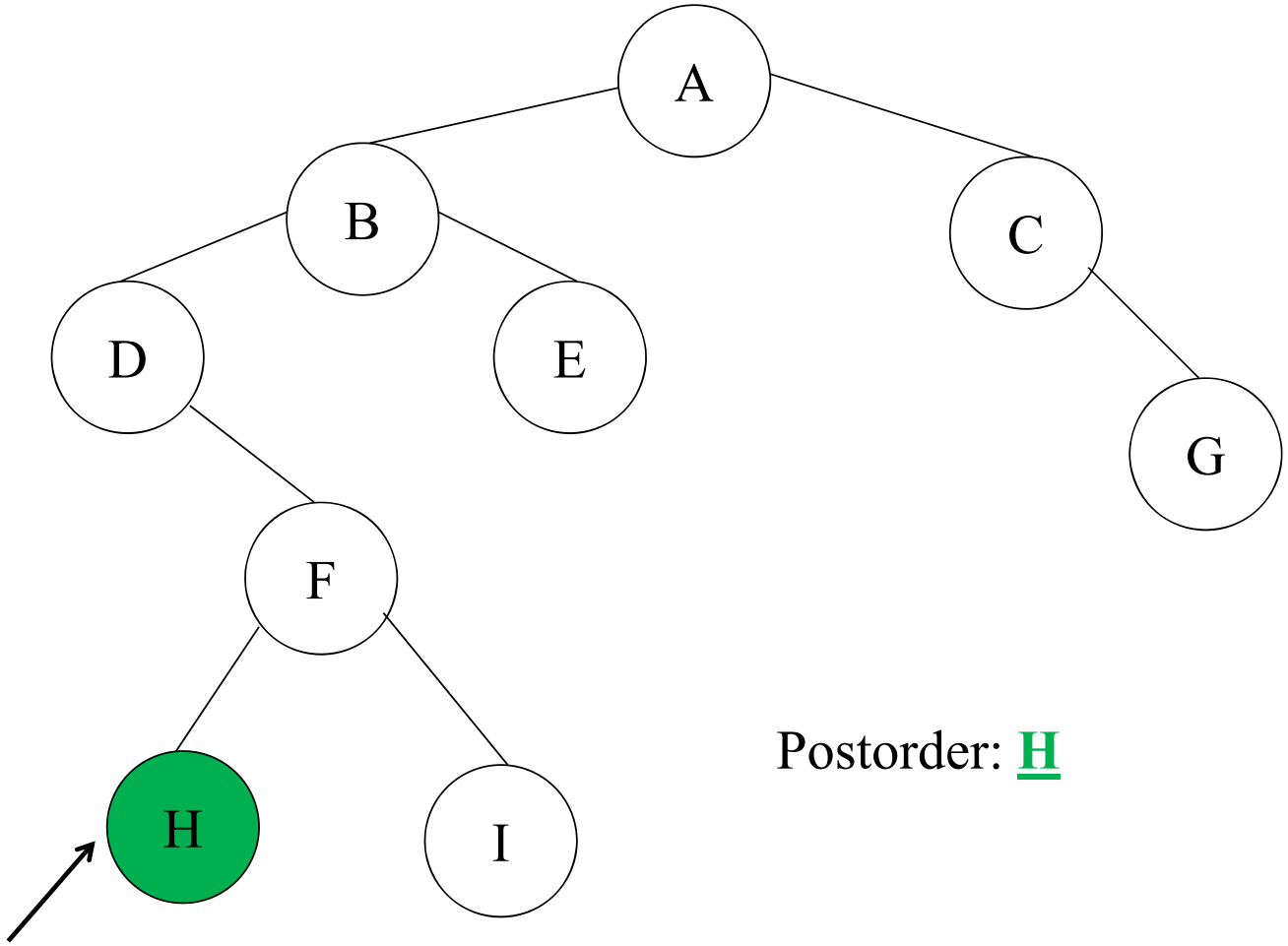
Postorder:

Tree Traversals: Another Example



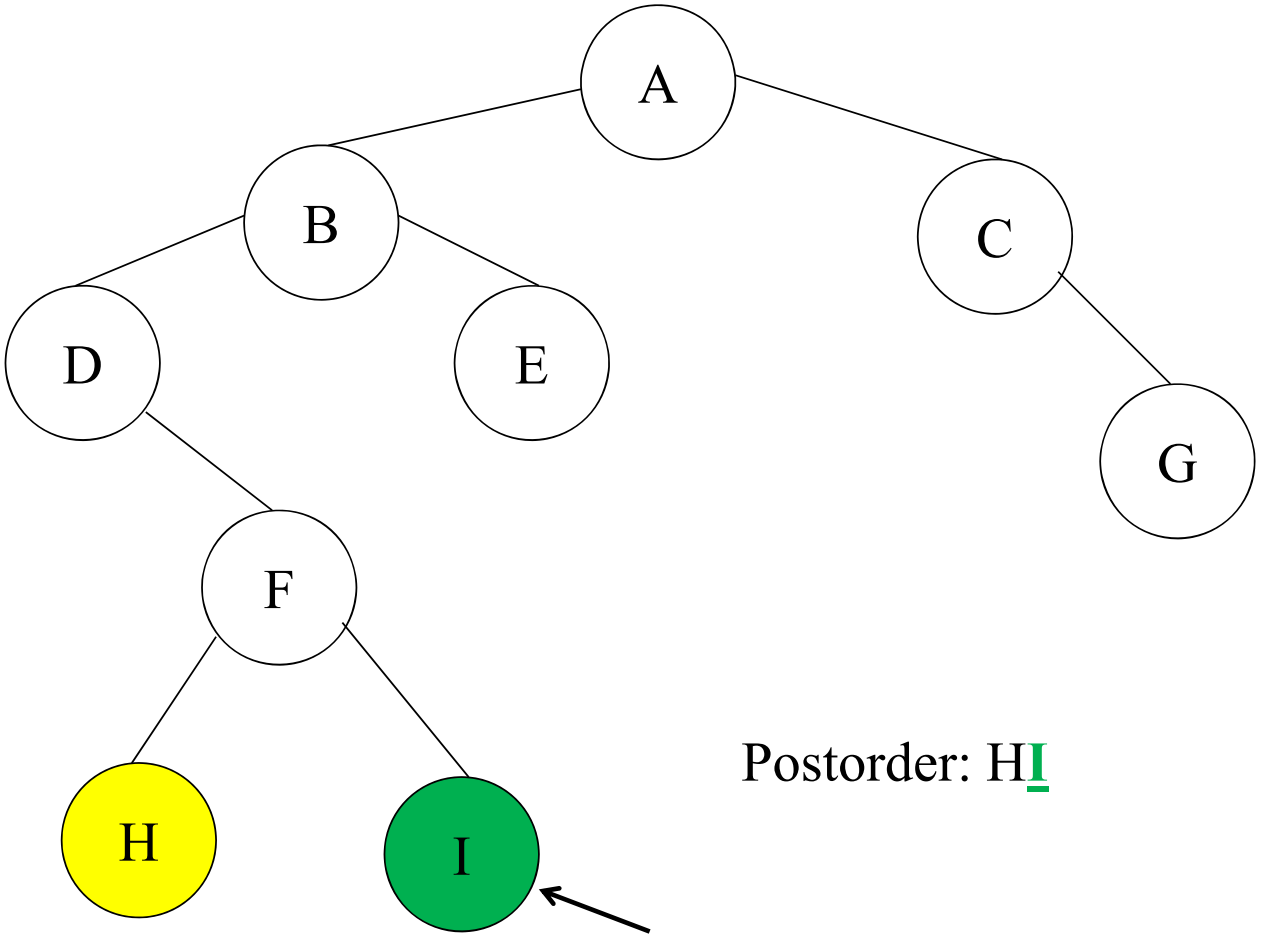
Postorder:

Tree Traversals: Another Example



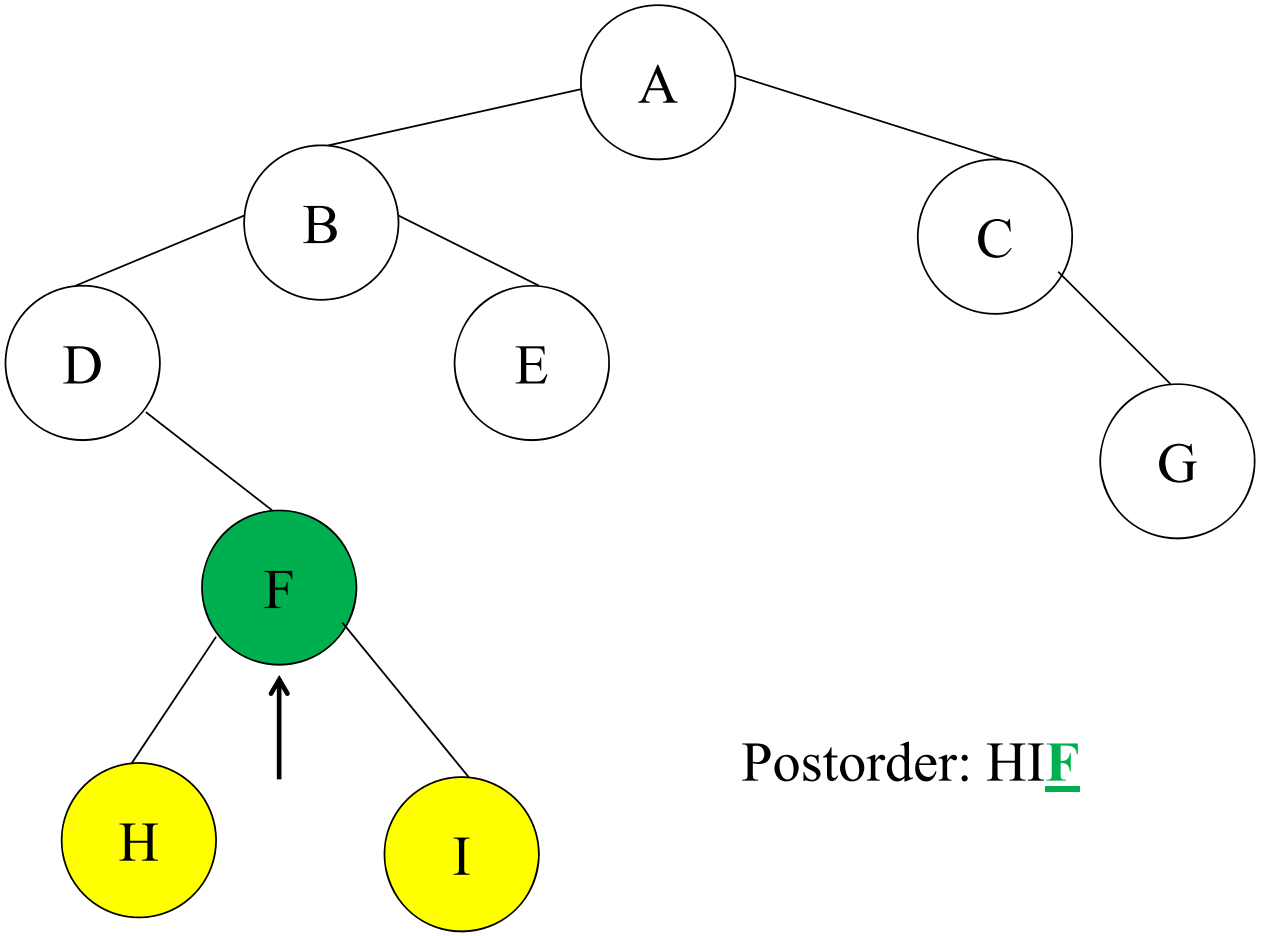
Postorder: **H**

Tree Traversals: Another Example



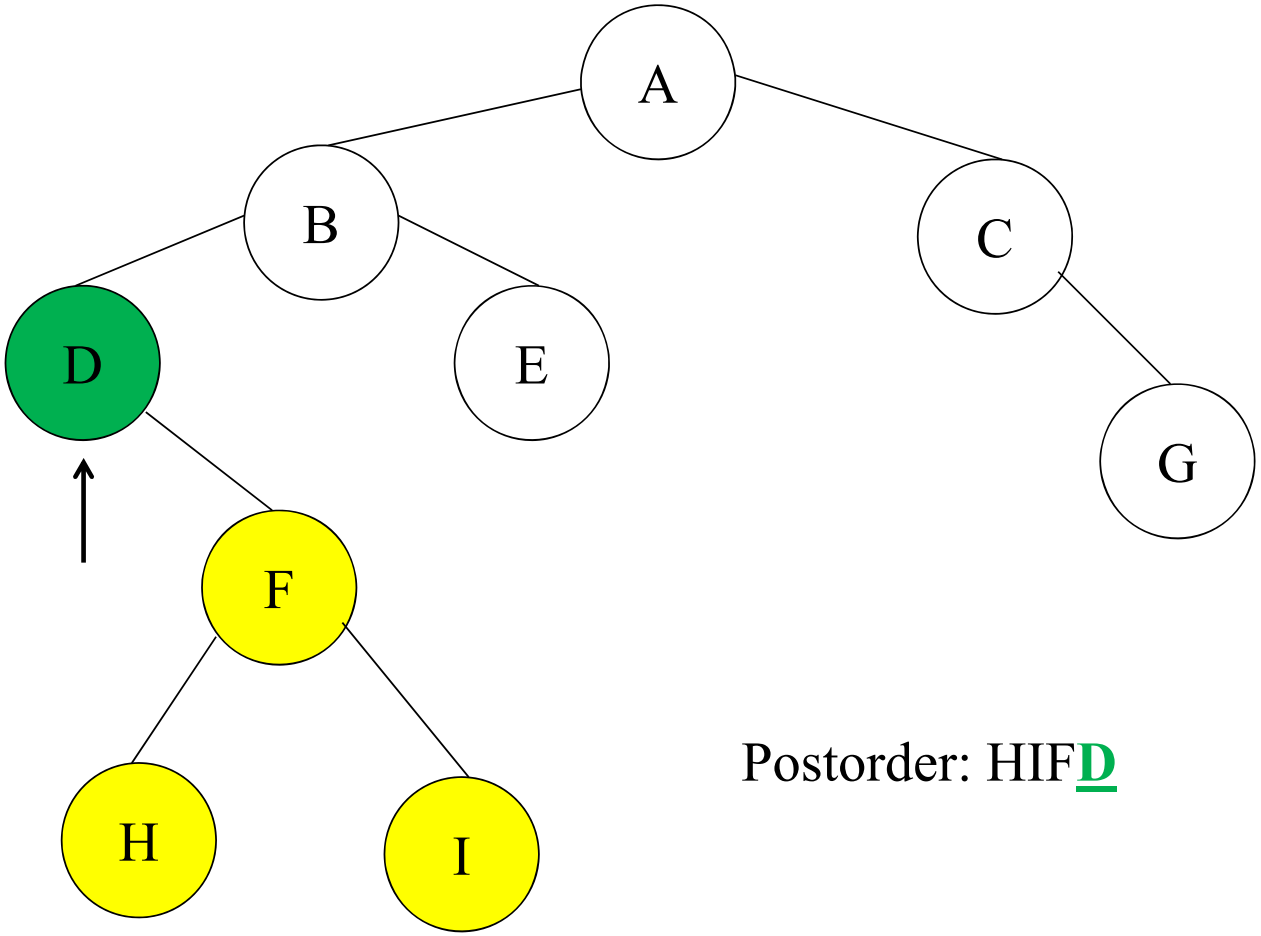
Postorder: HI

Tree Traversals: Another Example



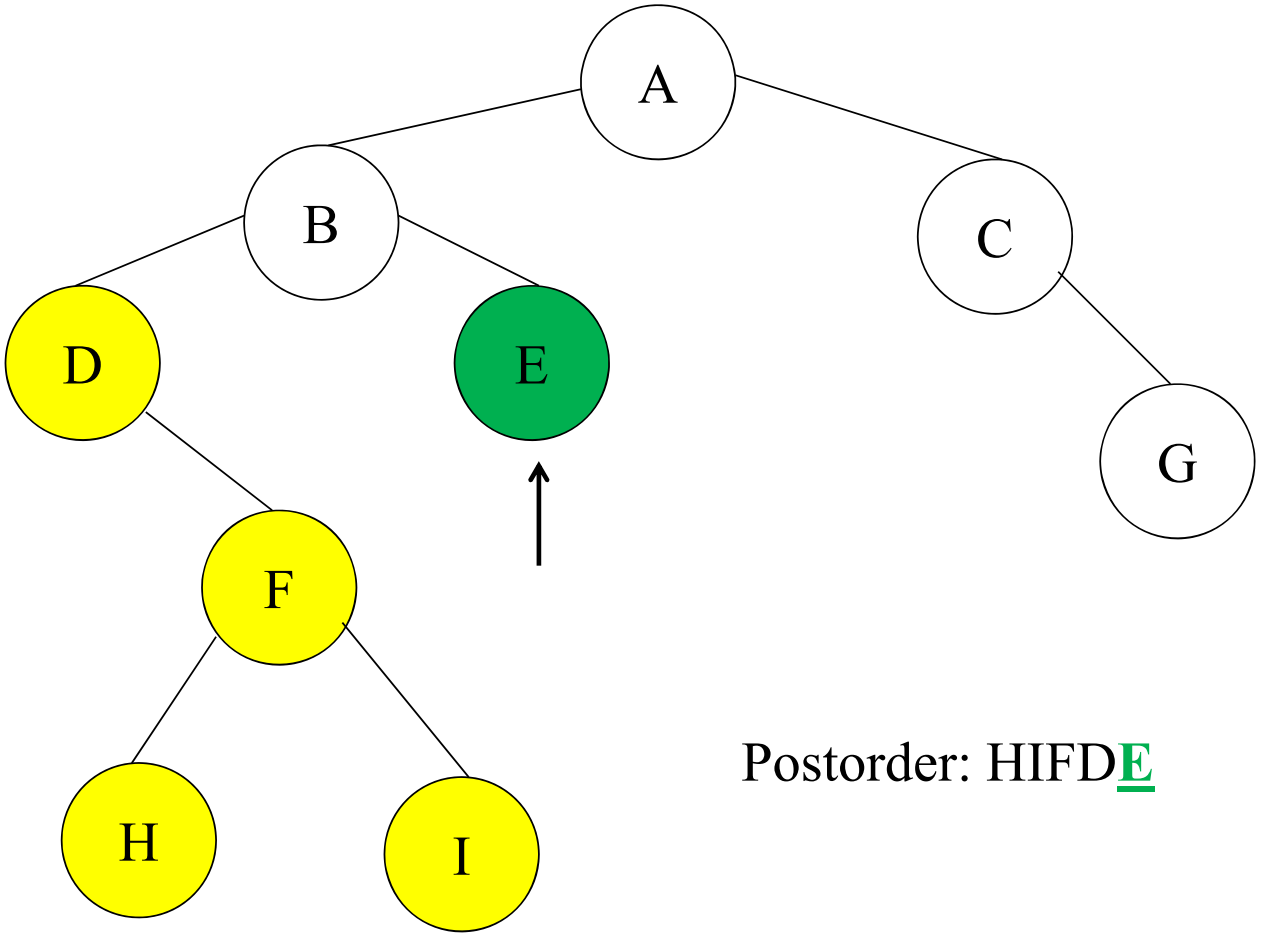
Postorder: HI**F**

Tree Traversals: Another Example



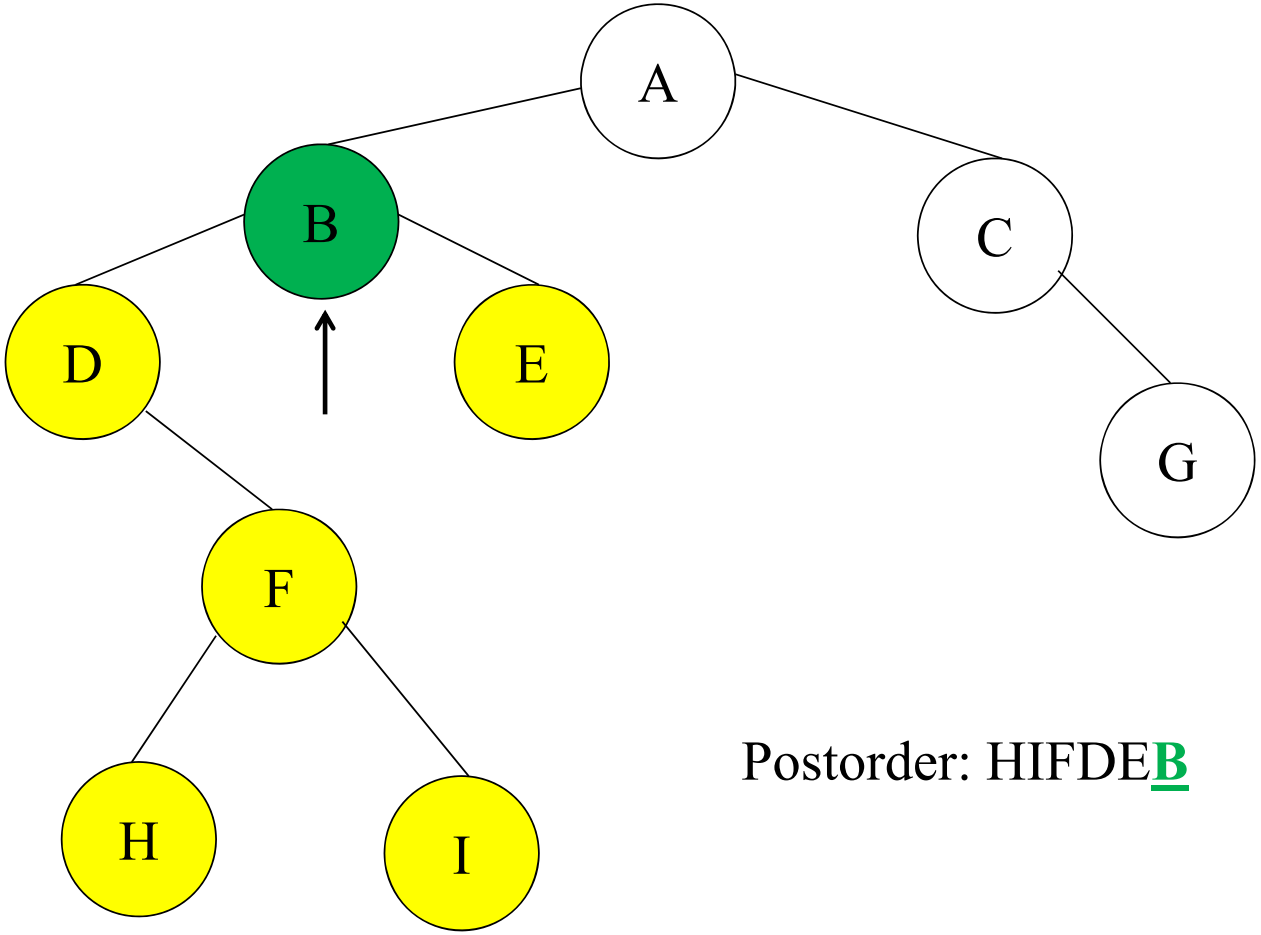
Postorder: HIF**D**

Tree Traversals: Another Example



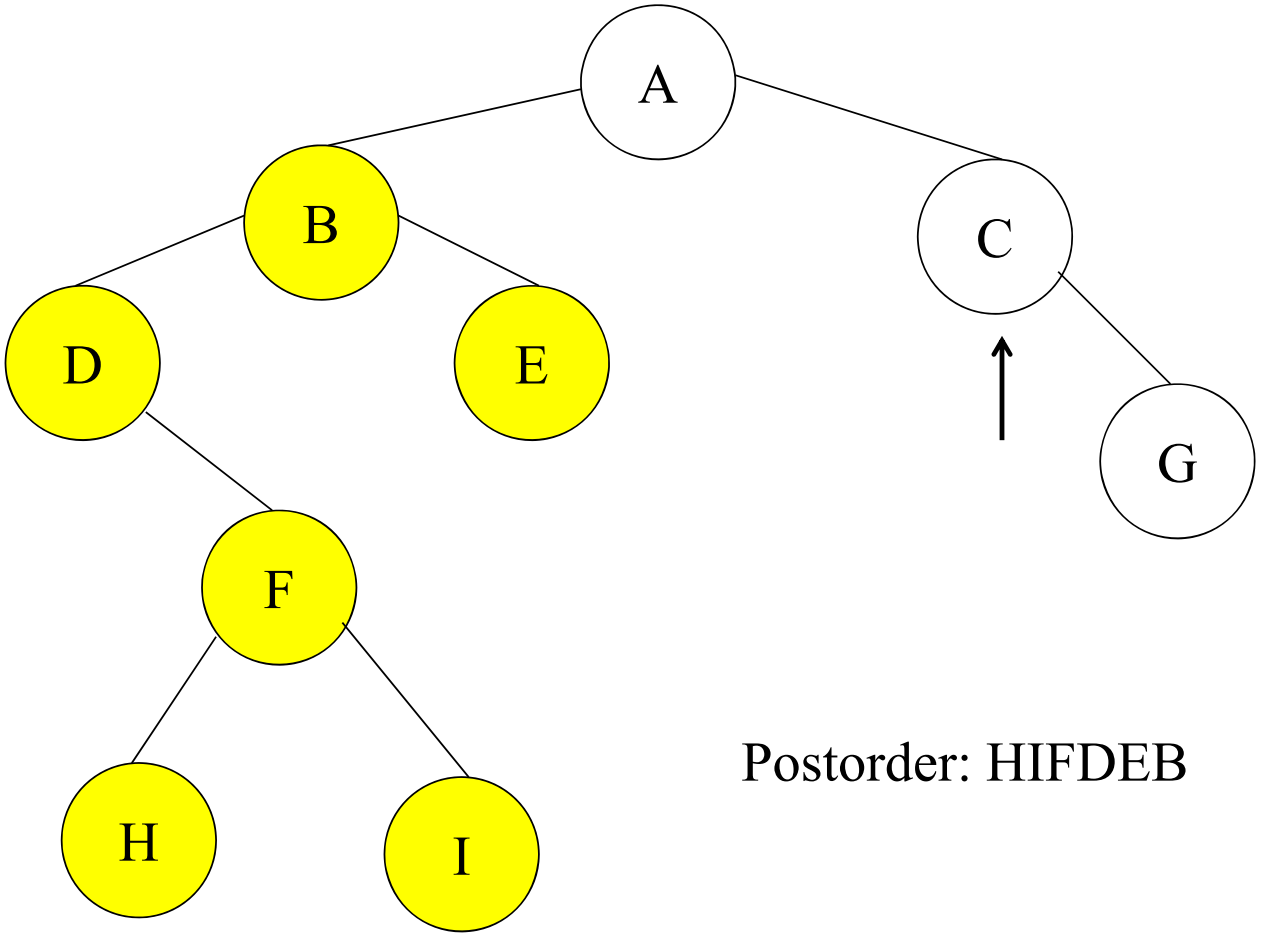
Postorder: HIFDE

Tree Traversals: Another Example



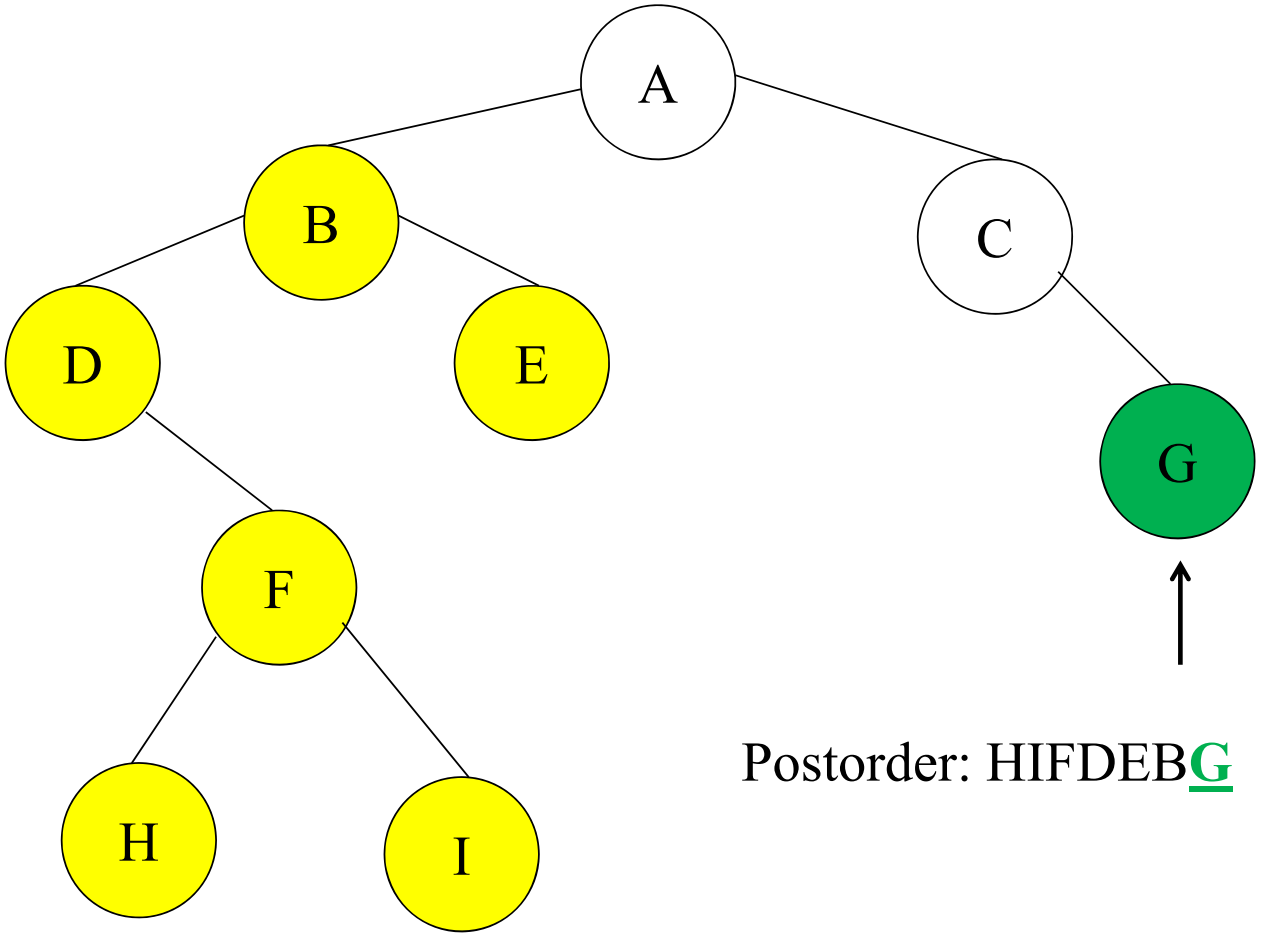
Postorder: HIFDE**B**

Tree Traversals: Another Example



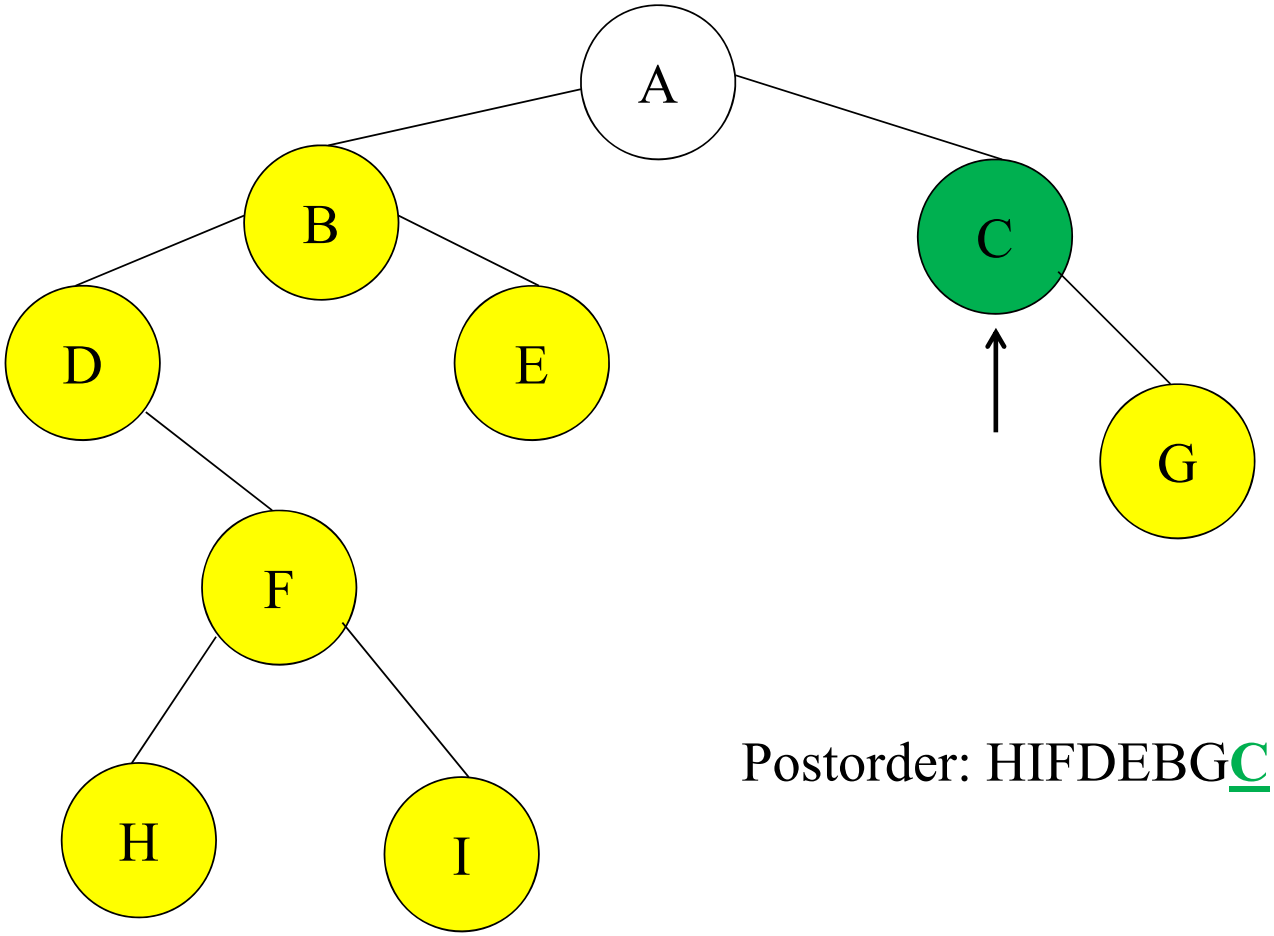
Postorder: HIFDEB

Tree Traversals: Another Example



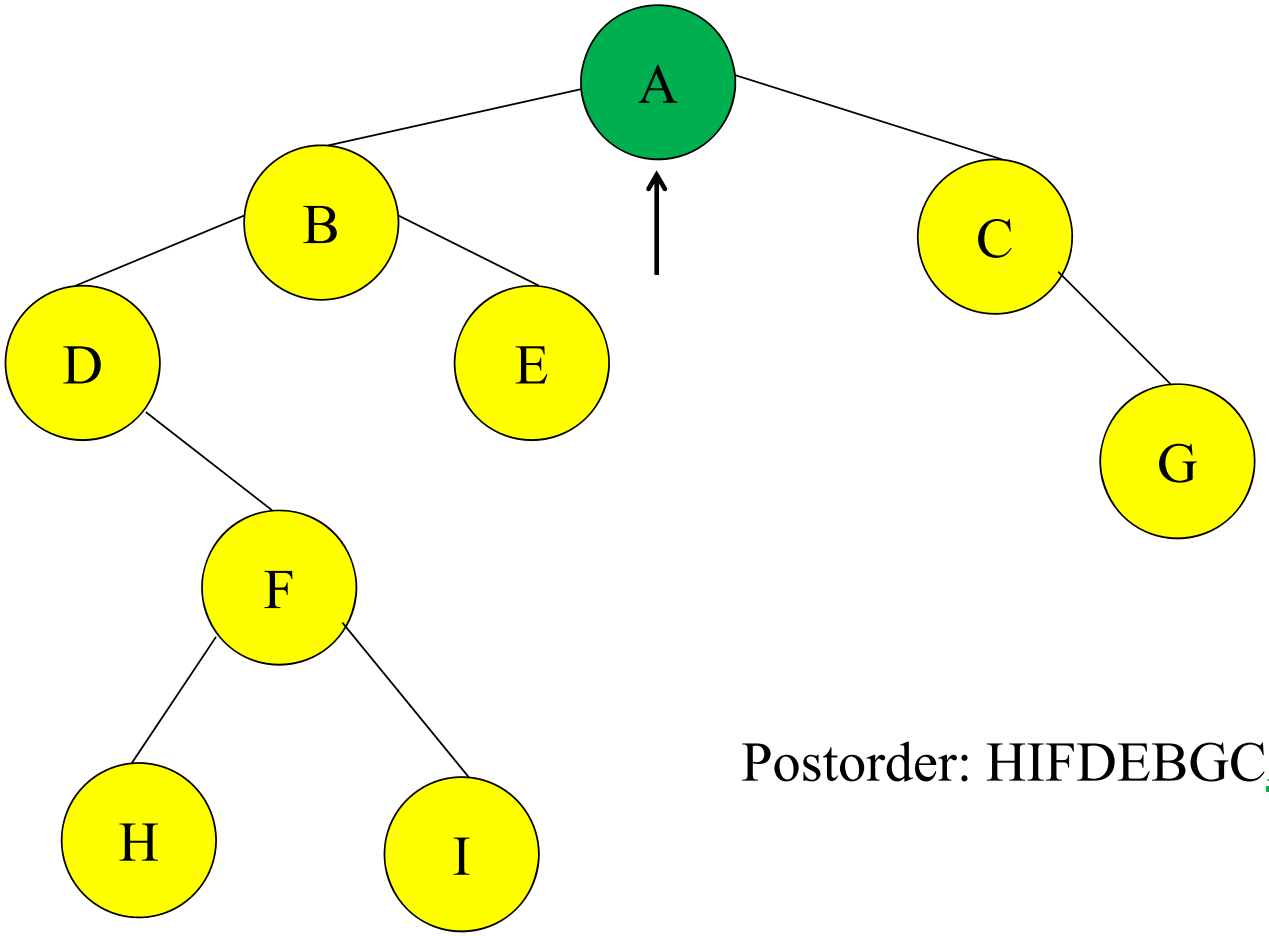
Postorder: HIFDEBG

Tree Traversals: Another Example



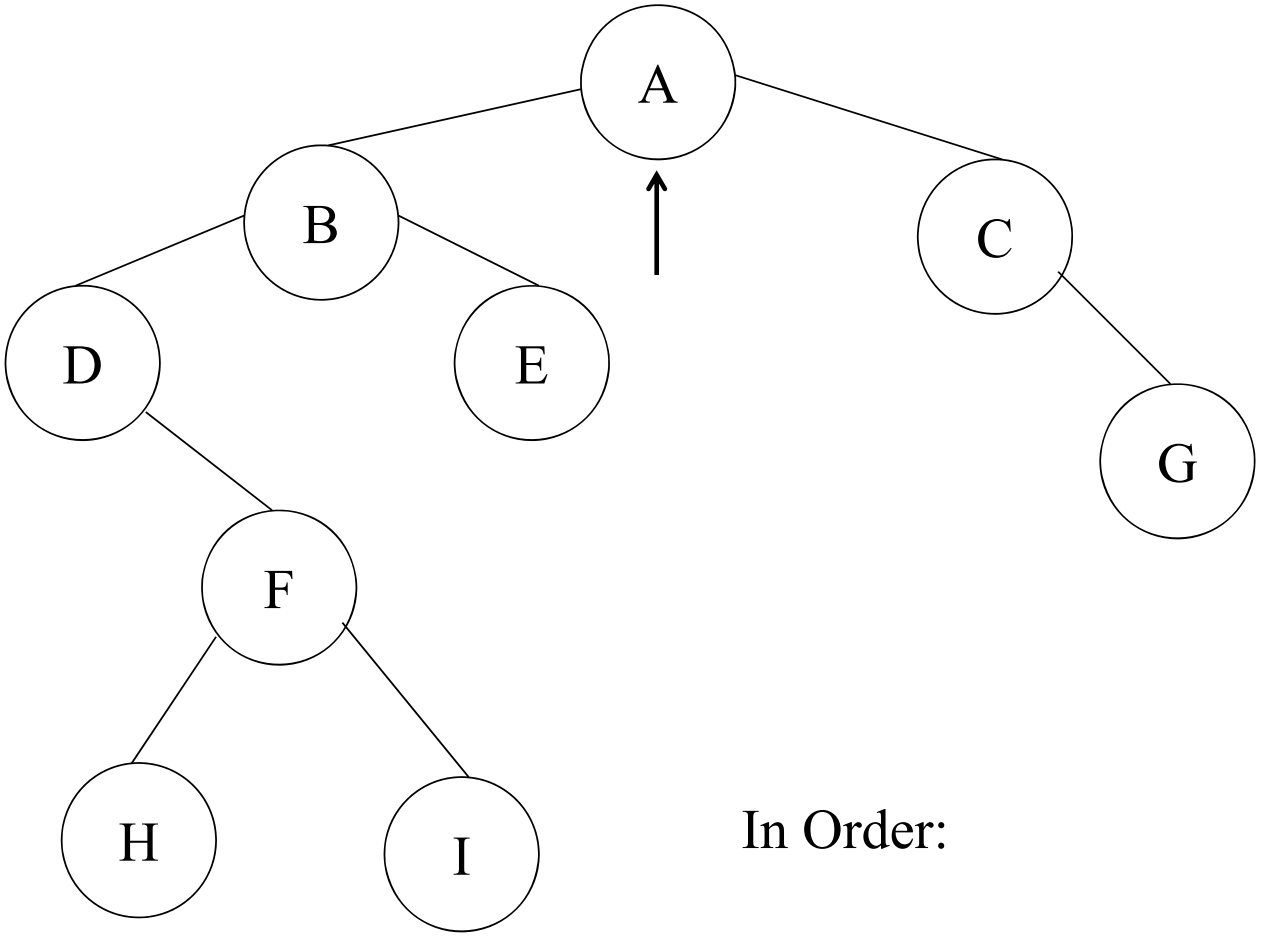
Postorder: HIFDEBG**C**

Tree Traversals: Another Example



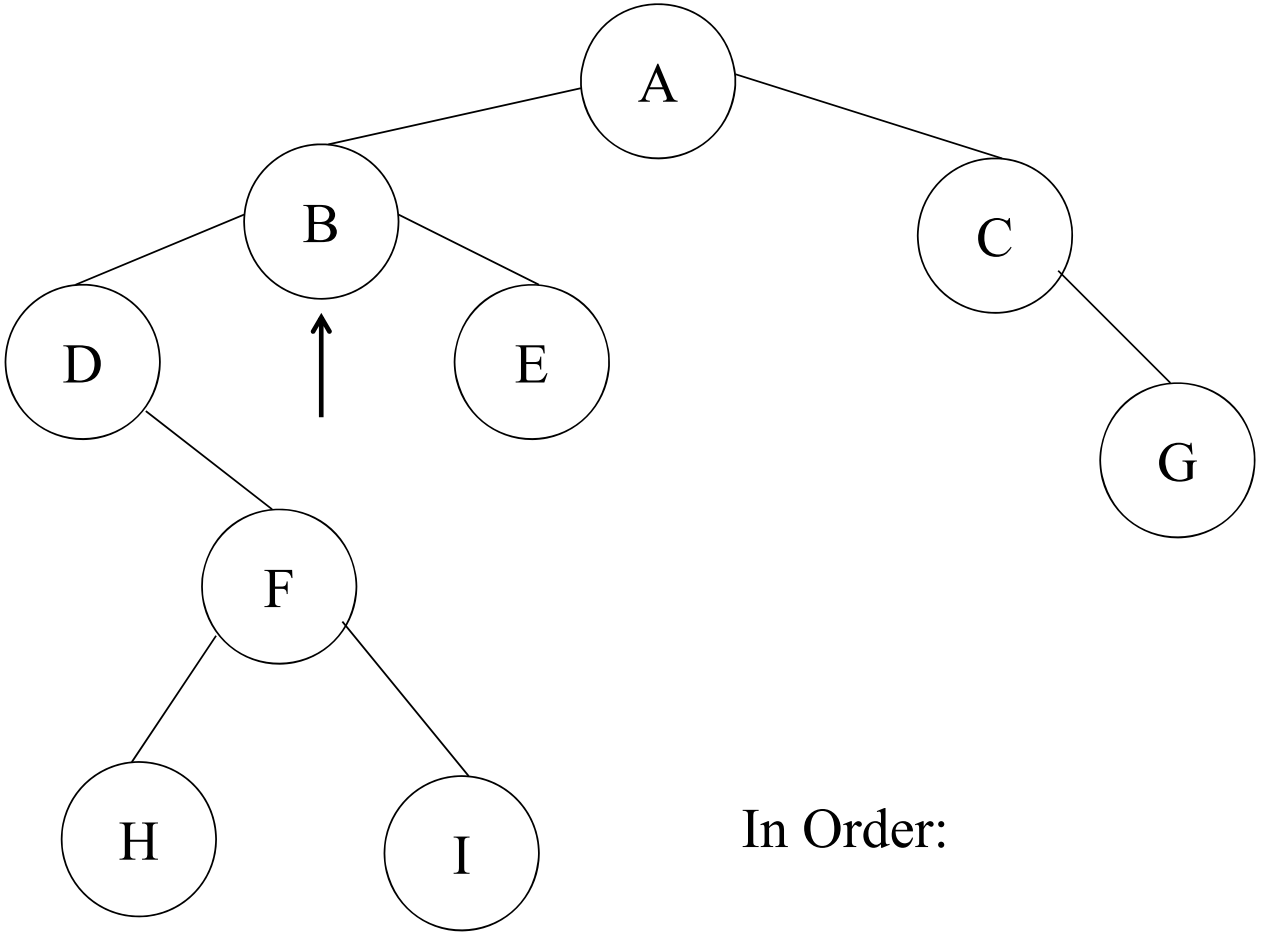
Postorder: HIFDEBGC**A**

Tree Traversals: Another Example



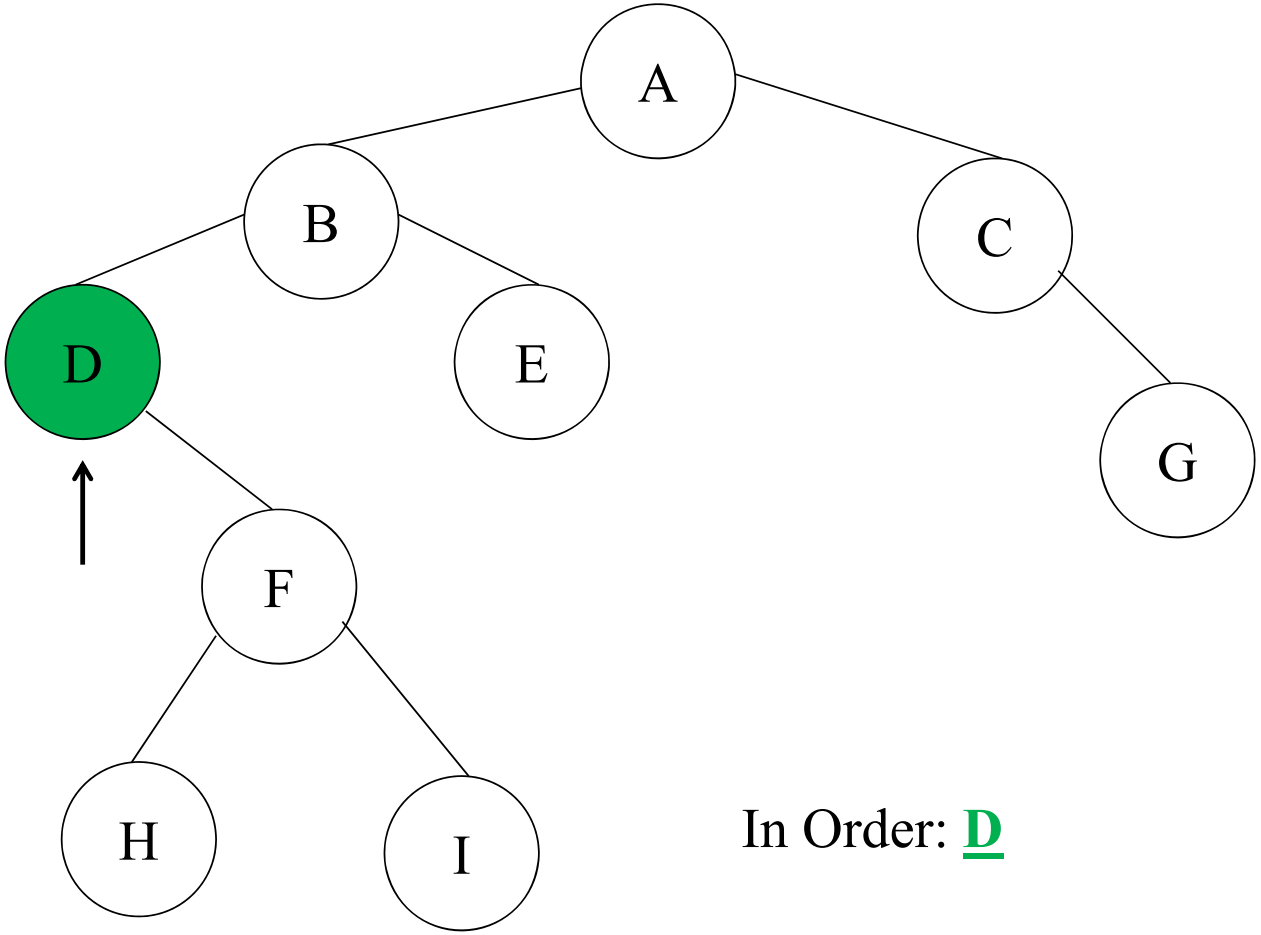
In Order:

Tree Traversals: Another Example



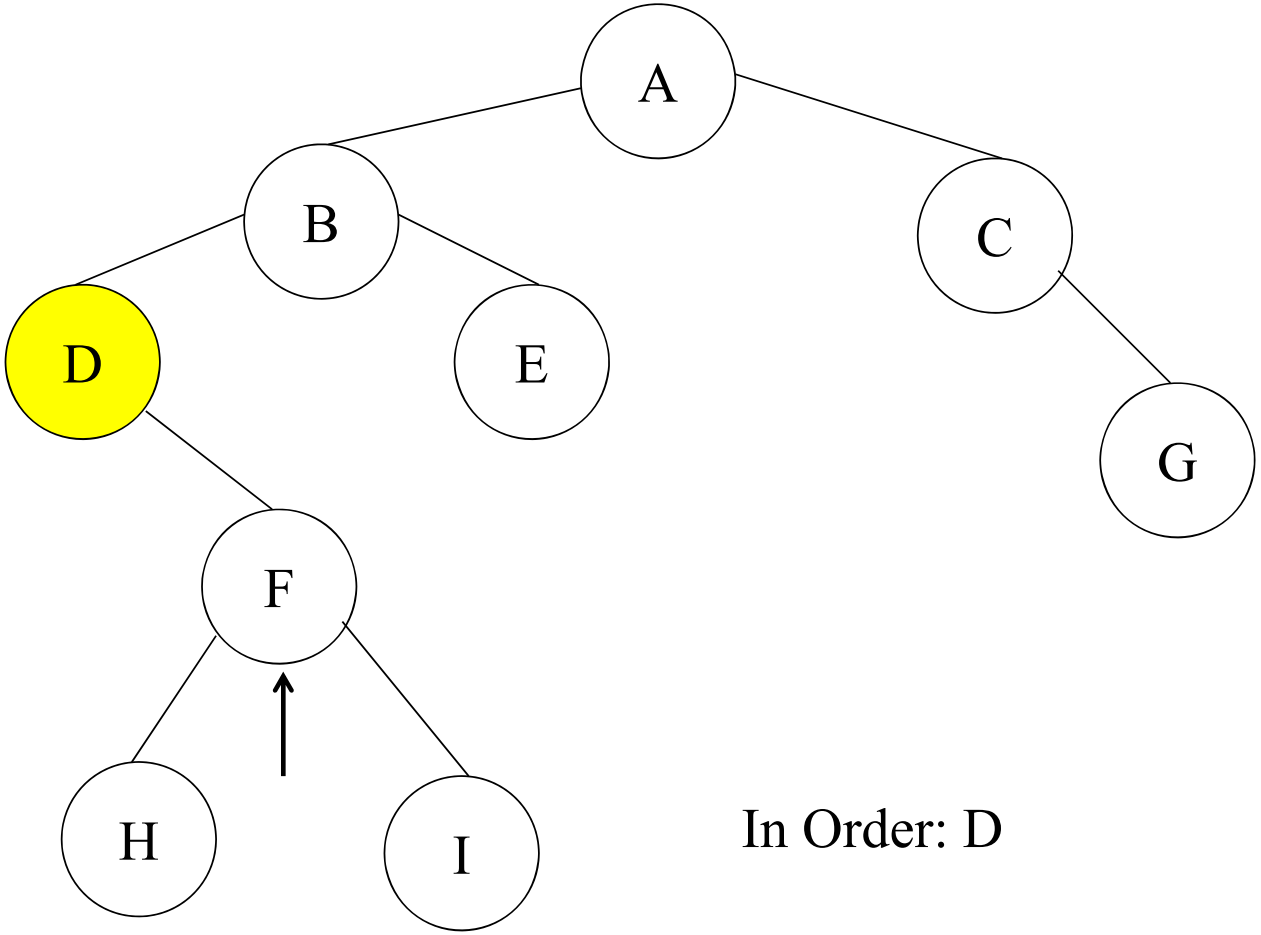
In Order:

Tree Traversals: Another Example



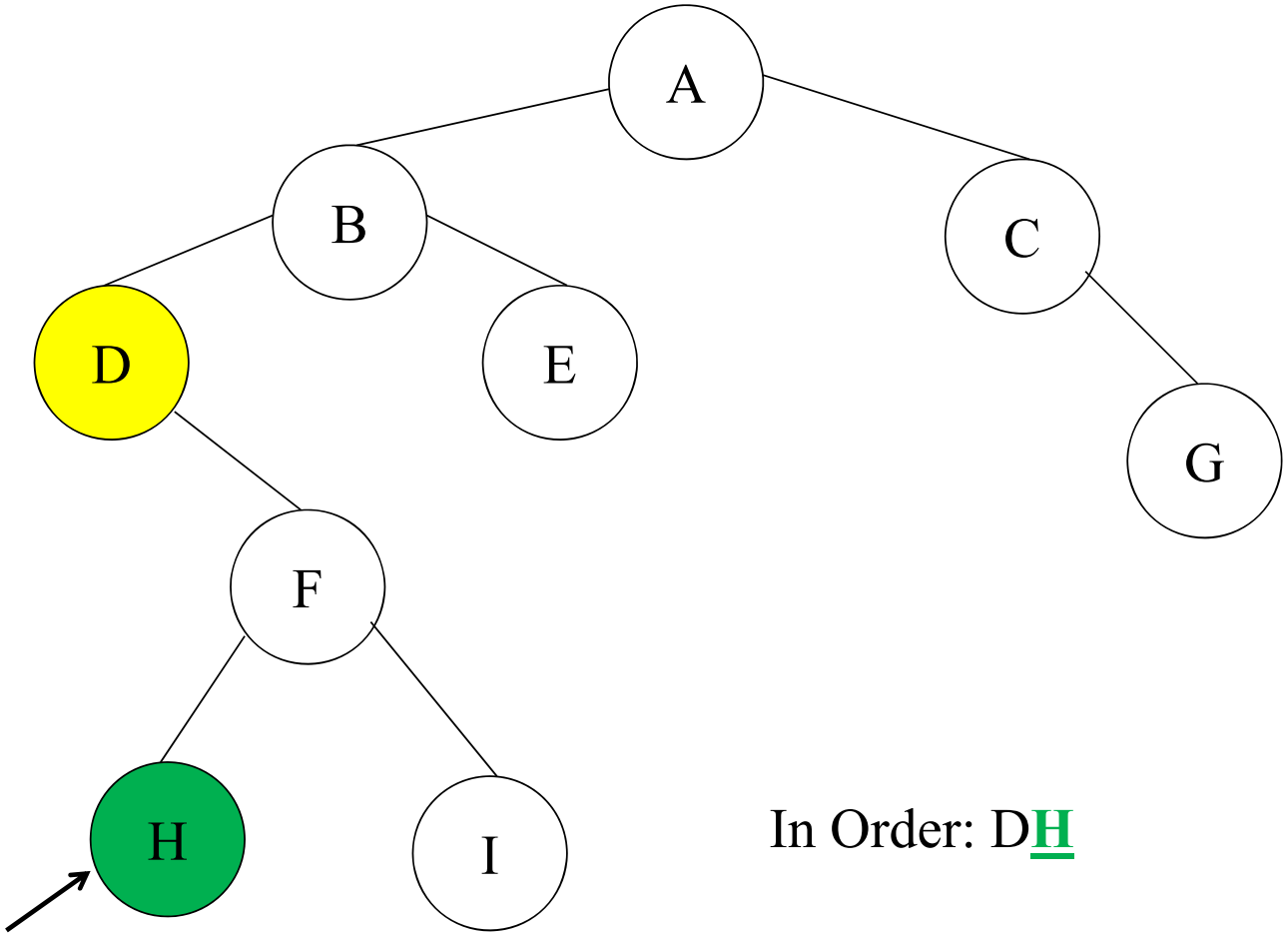
In Order: **D**

Tree Traversals: Another Example



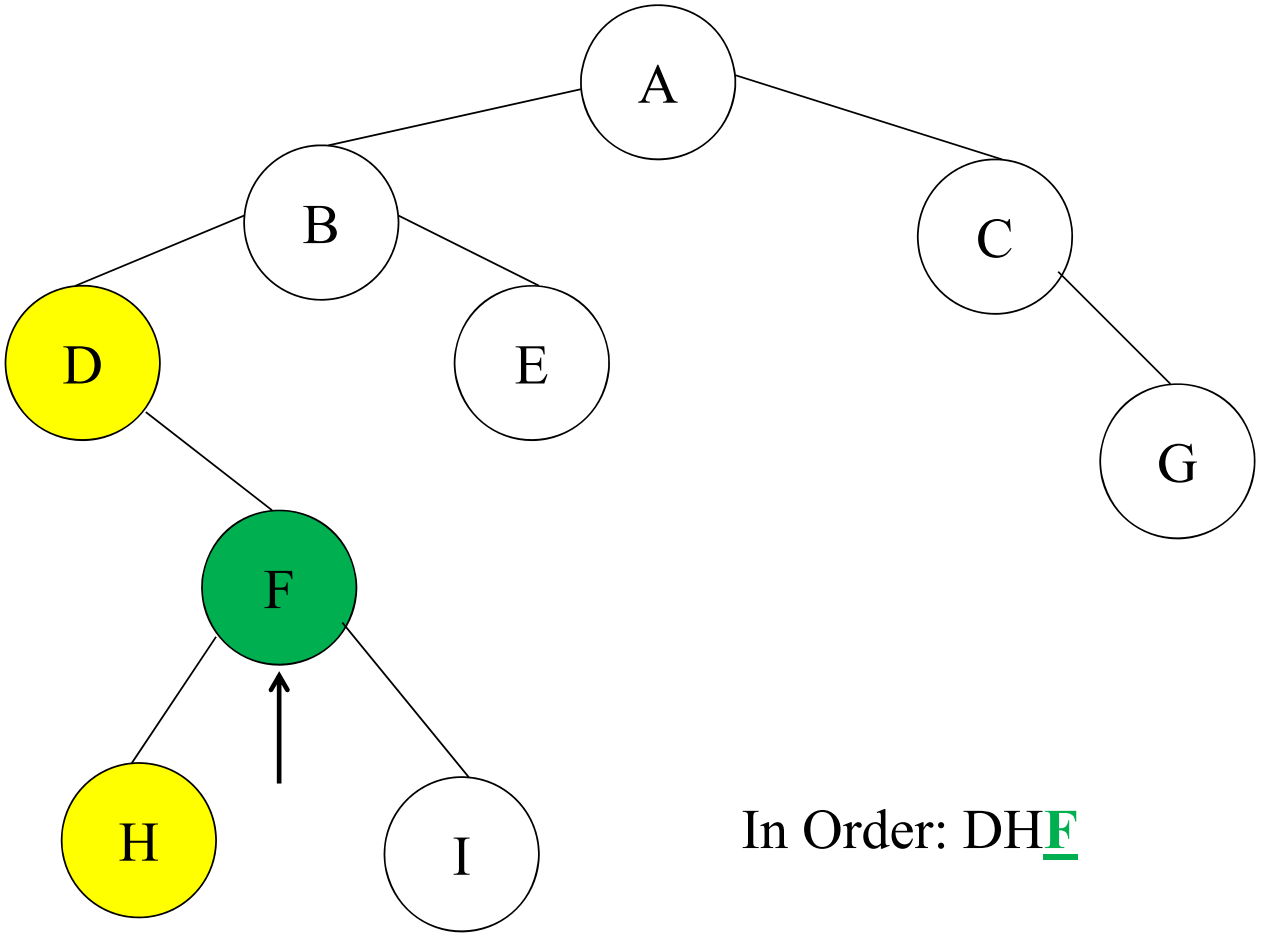
In Order: D

Tree Traversals: Another Example



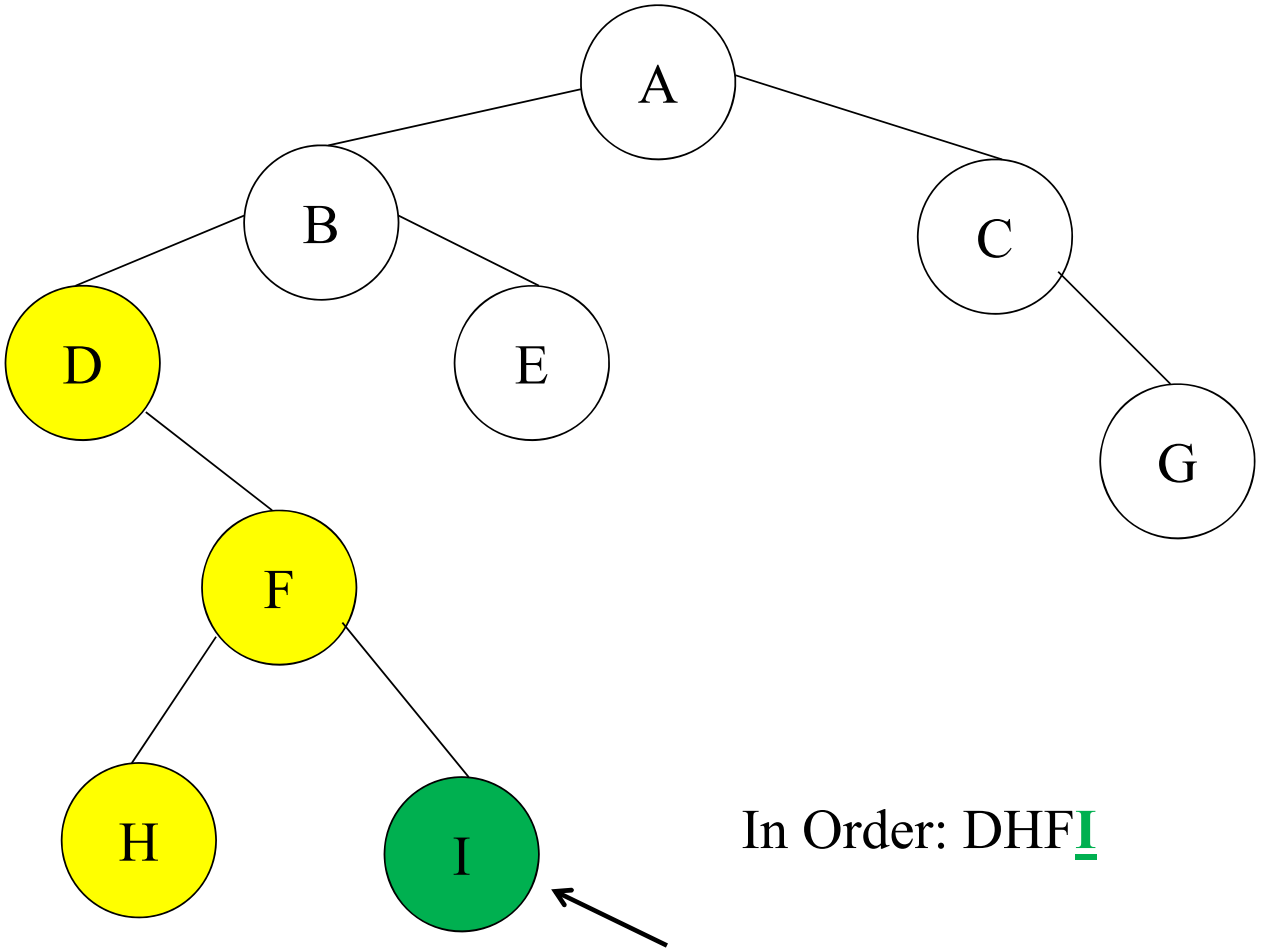
In Order: D**H**

Tree Traversals: Another Example



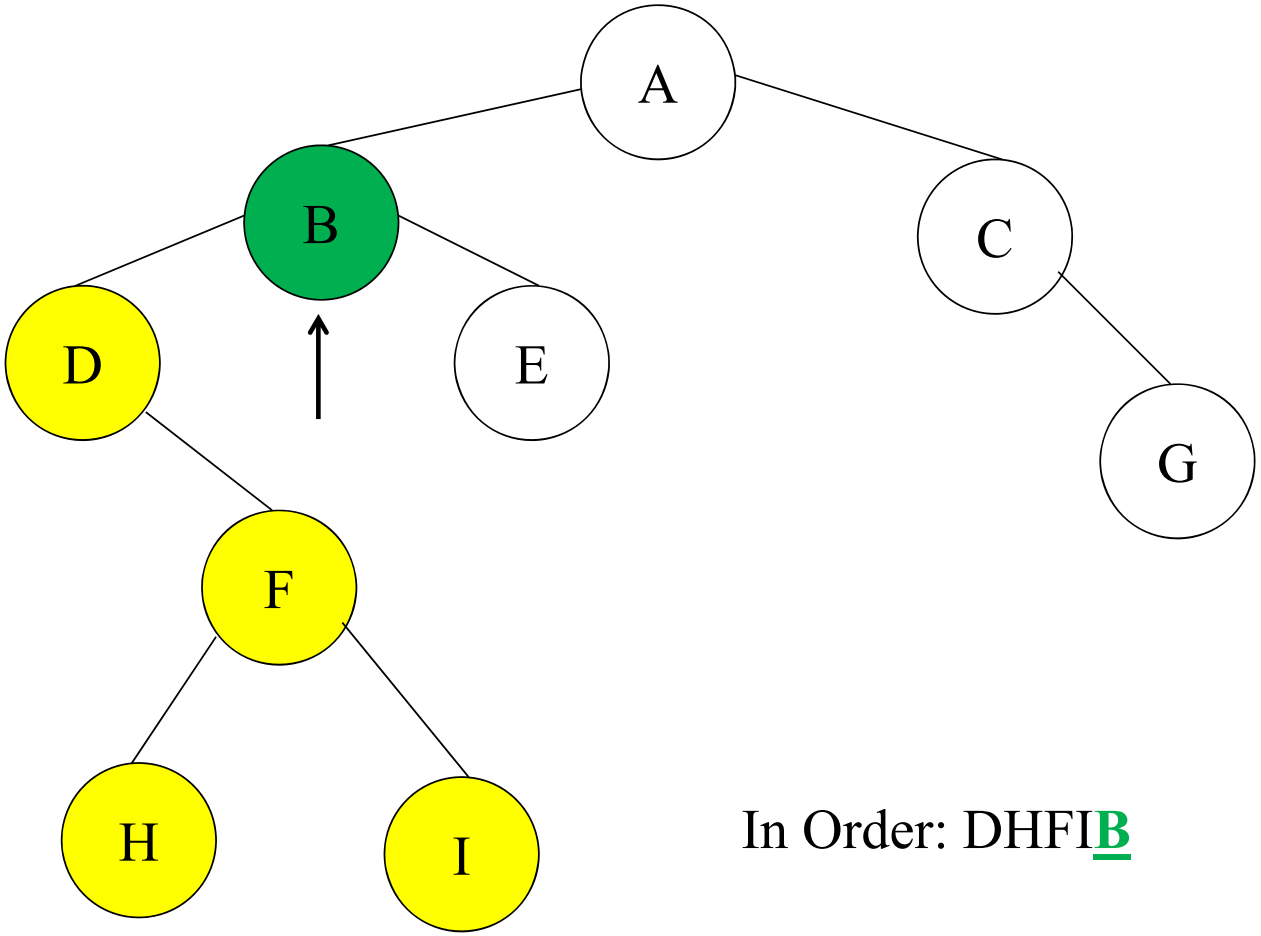
In Order: DHF

Tree Traversals: Another Example



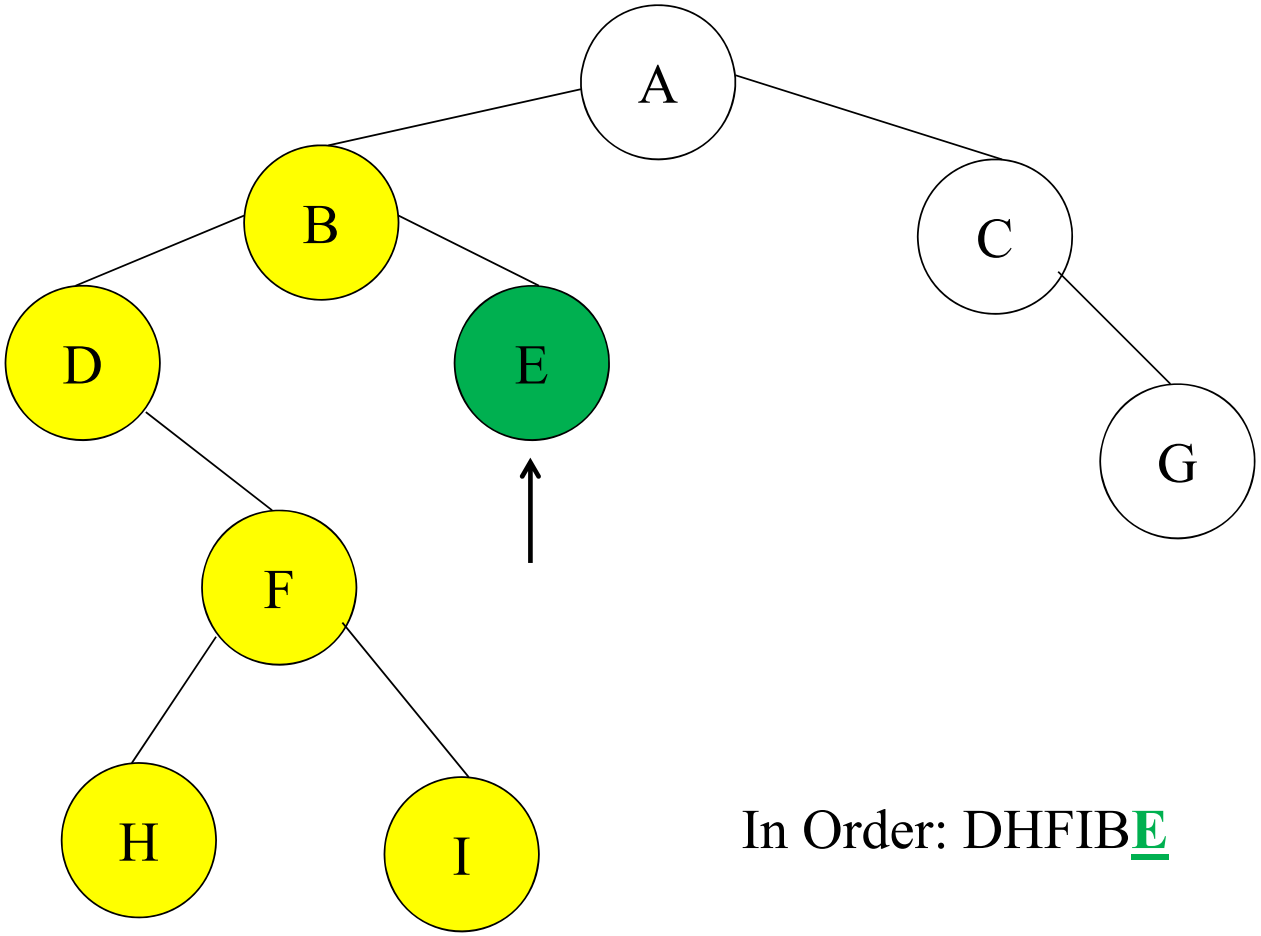
In Order: DHFI

Tree Traversals: Another Example



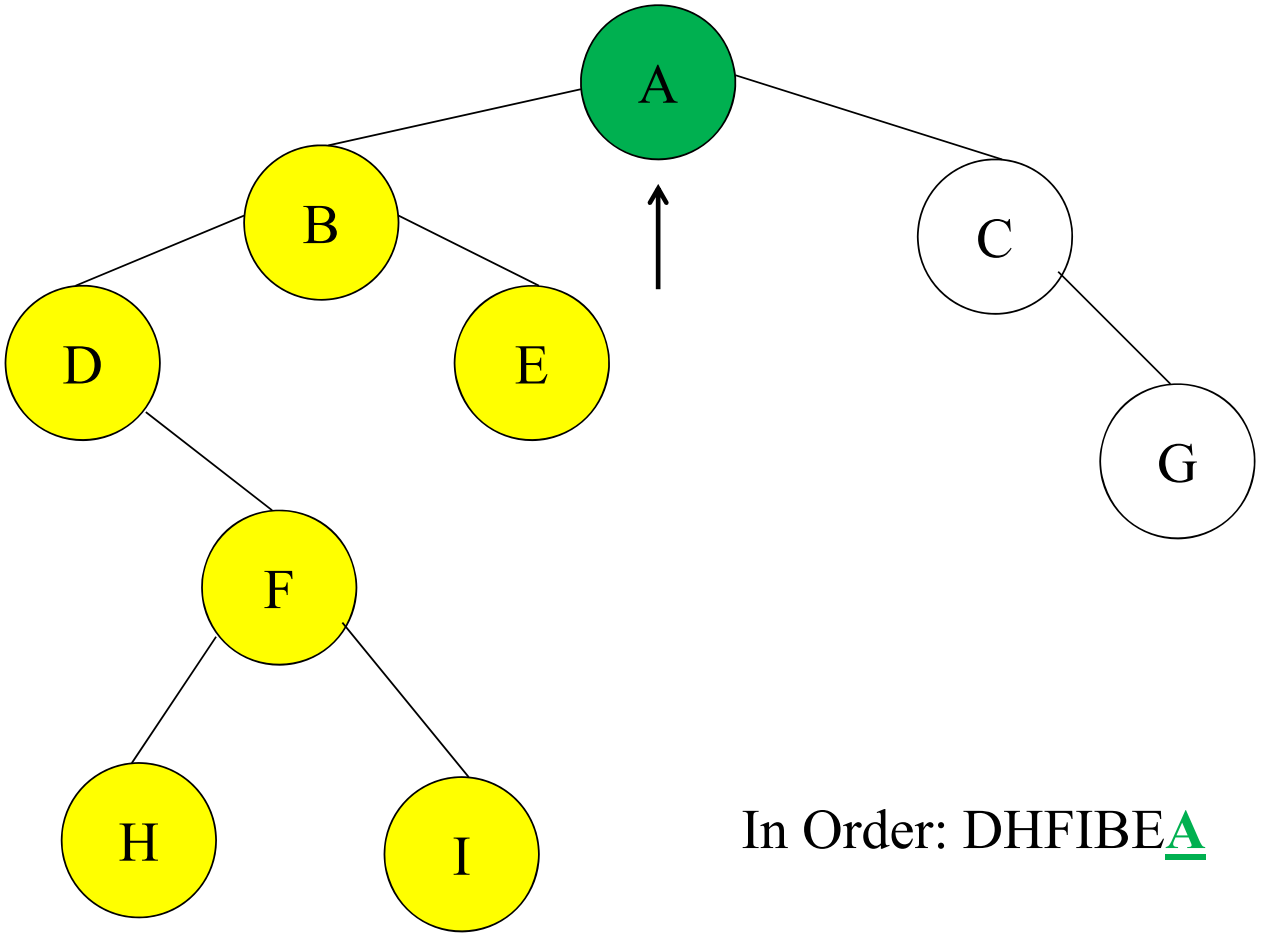
In Order: DHFI**B**

Tree Traversals: Another Example



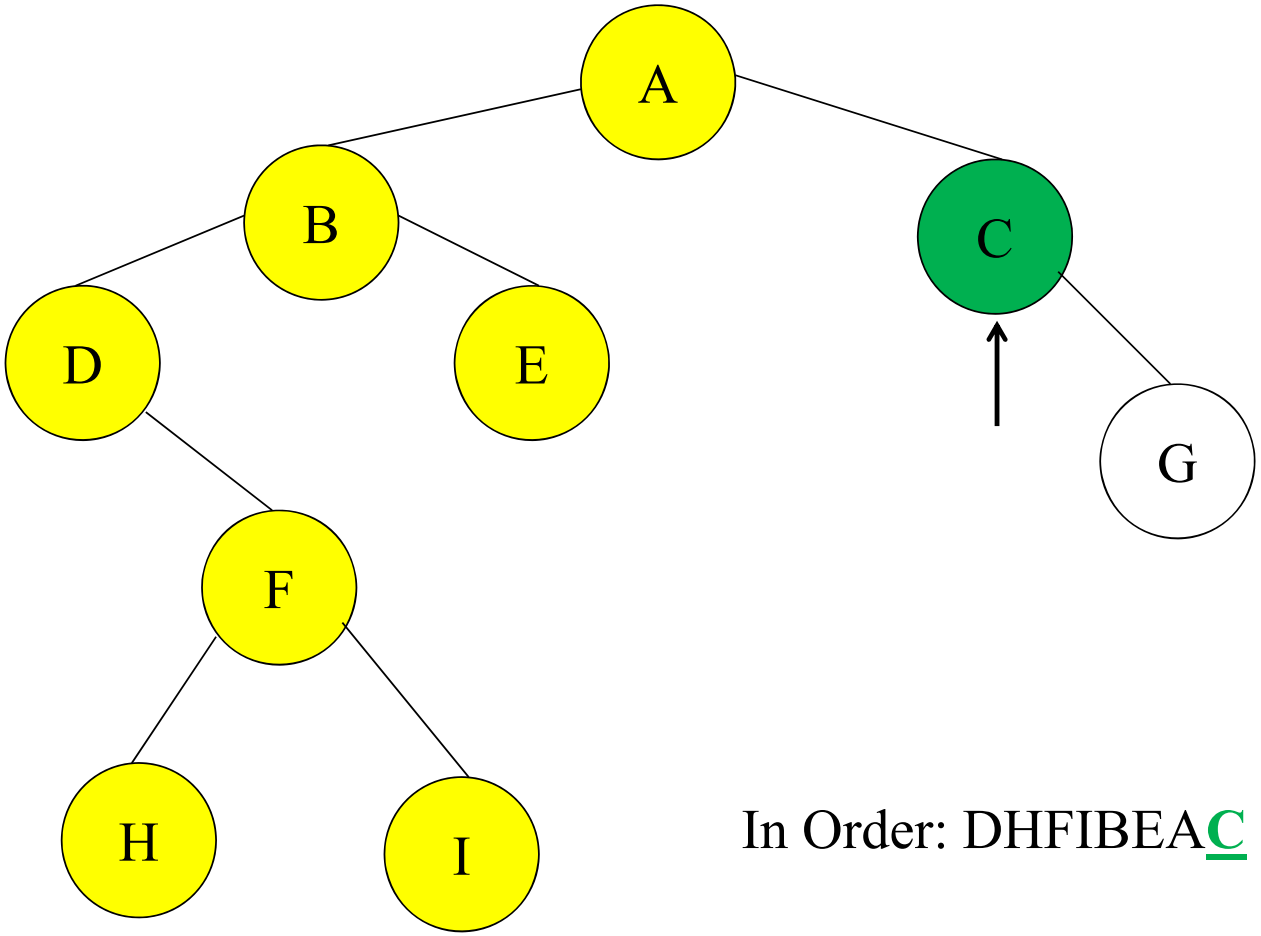
In Order: DHFIB**E**

Tree Traversals: Another Example



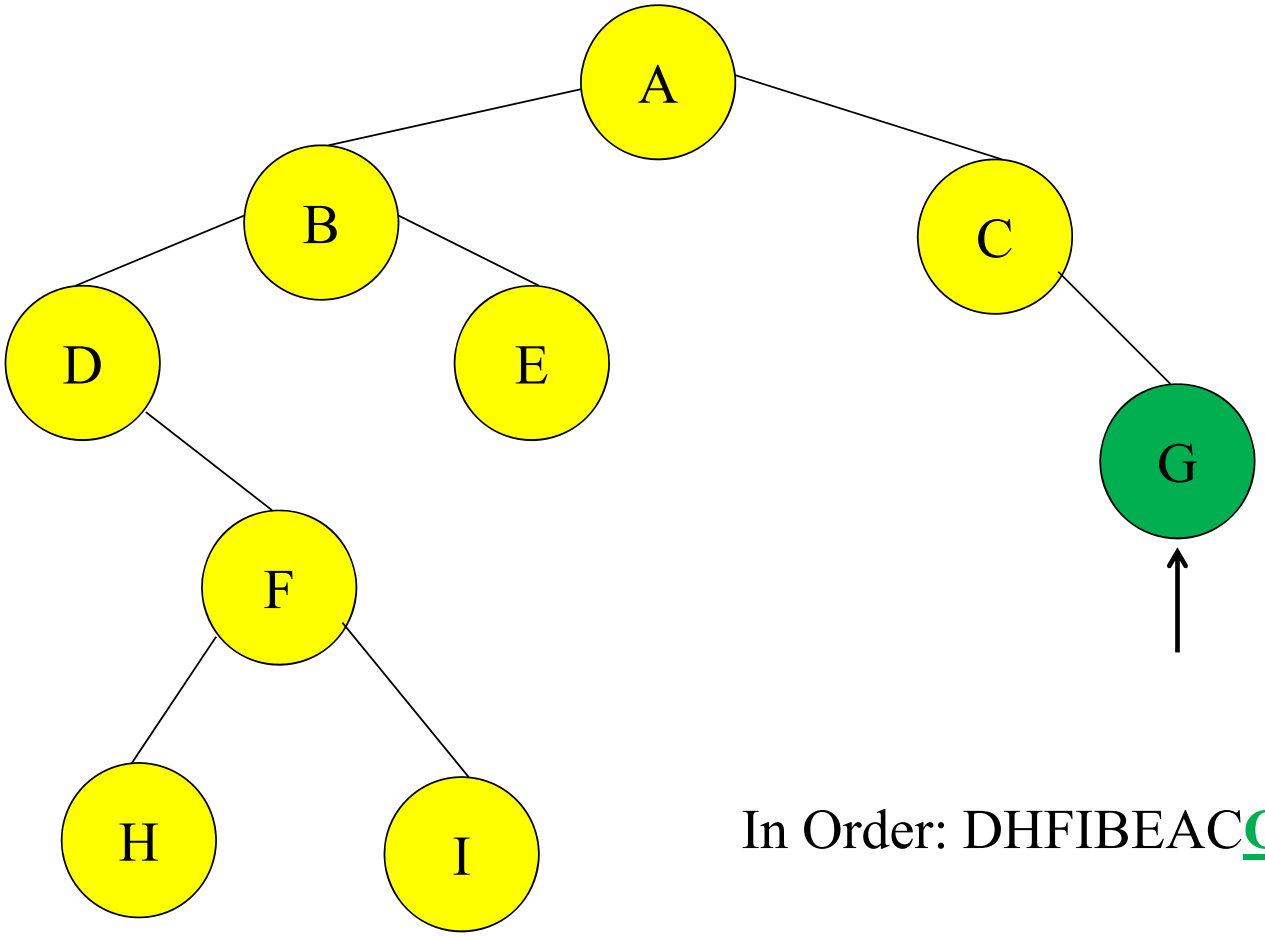
In Order: DHFIBE**A**

Tree Traversals: Another Example



In Order: DHFIBEAC

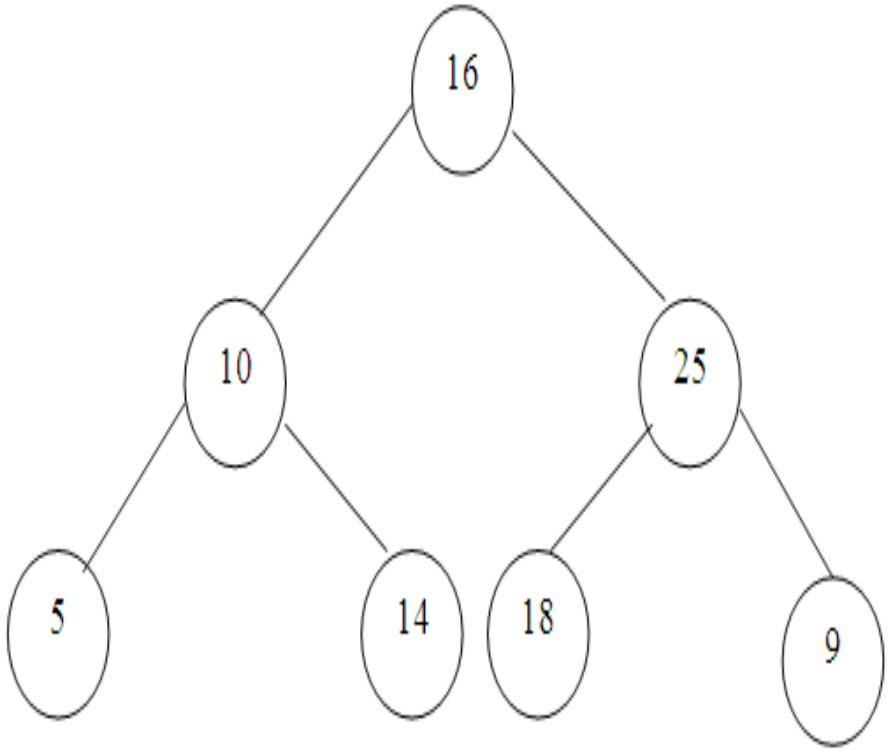
Tree Traversals: Another Example



In Order: DHFIBEACG

Tree Traversals: Another Example

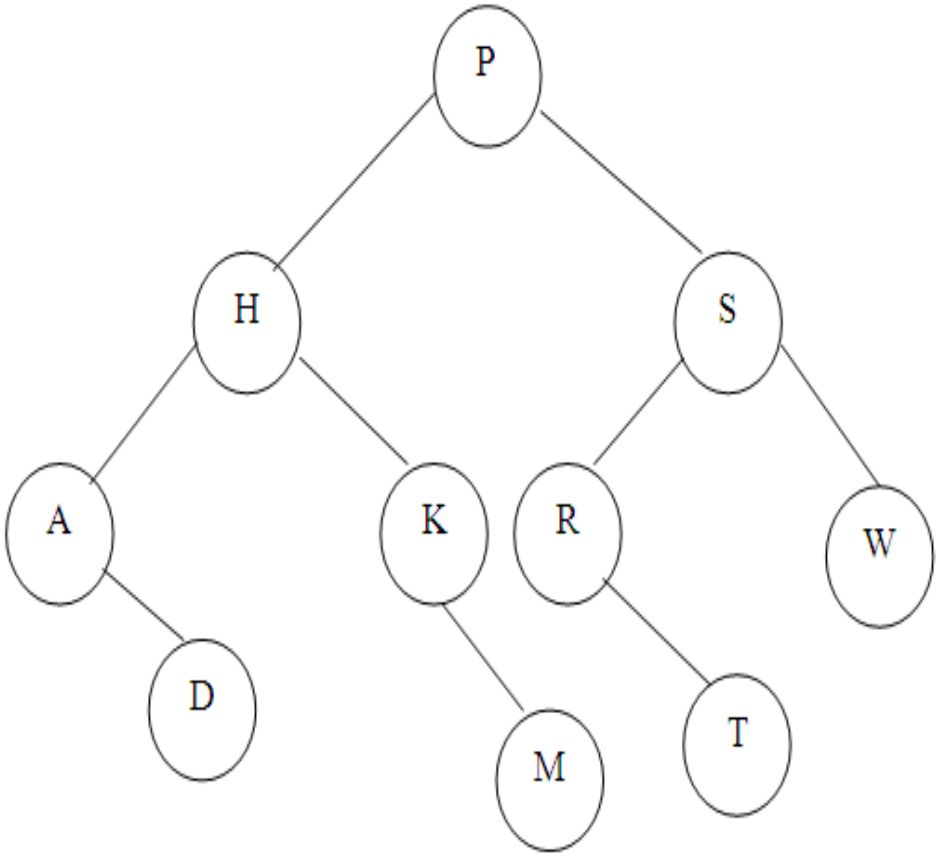
- Example 1: what are the traversal orders of bellow binary search tree?



- Preorder traversal (NLR)
  16,10,5,14,25,18,90
- Inorder traversal(LNR)
  5,10,14,16,18,25,90
- Postorder traversal(LRN)
  5,14,10,18,90,25,16

Tree Traversals: Another Example

Example 2: what are the traversal orders of bellow binary search tree?

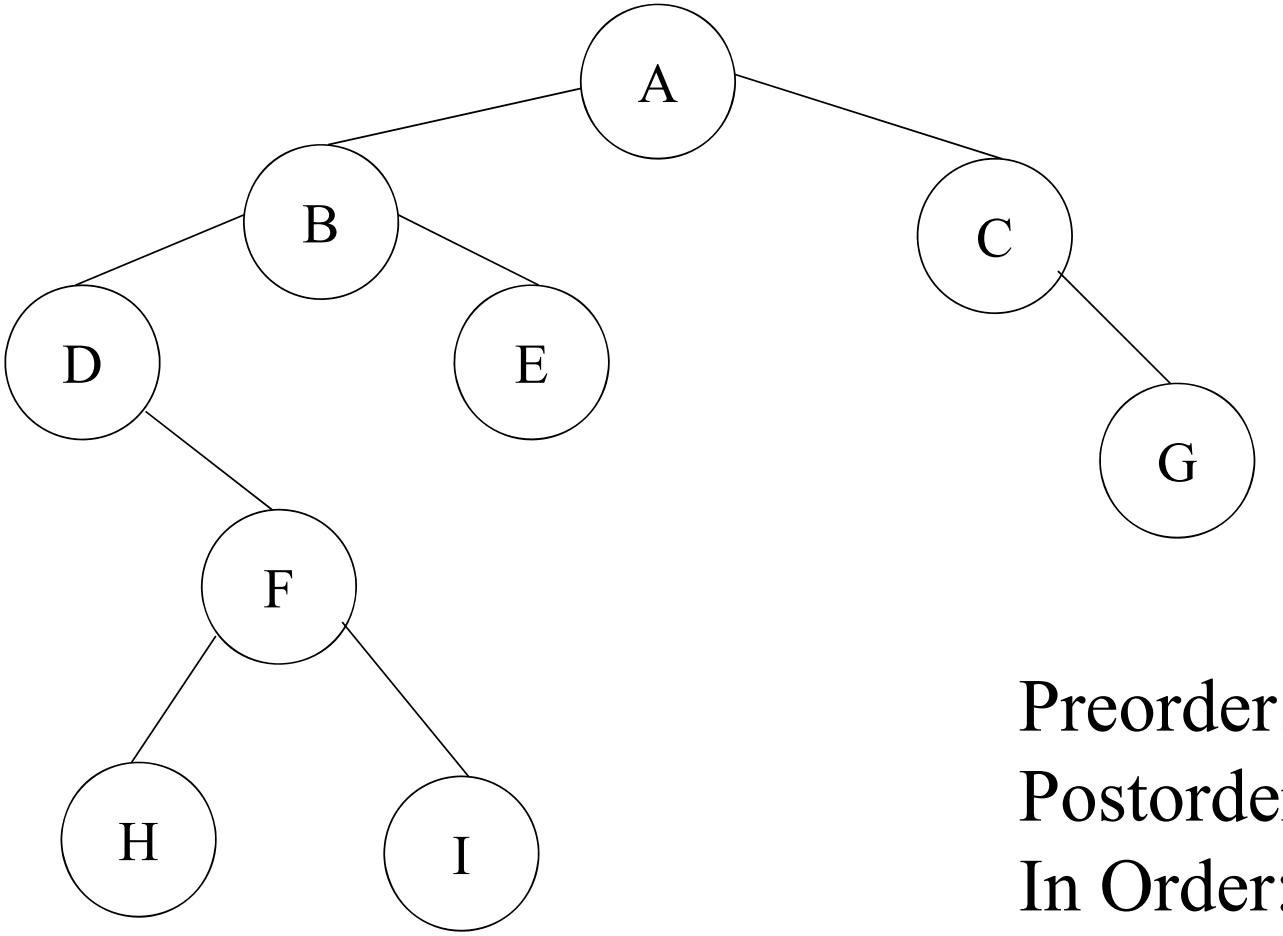


Preorder traversal (NLR)
P,H,A,D,K,M,S,R,T,W

Inorder traversal(LNR)
A,D,H,K,M,P,R,T,S,W

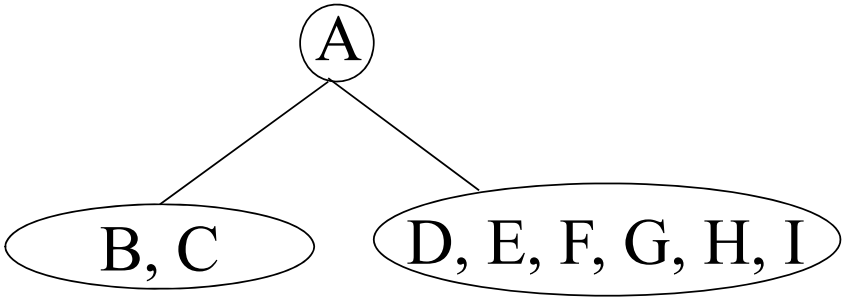
Postorder traversal(LRN)
D,A,M,K,H,T,R,W,S,P

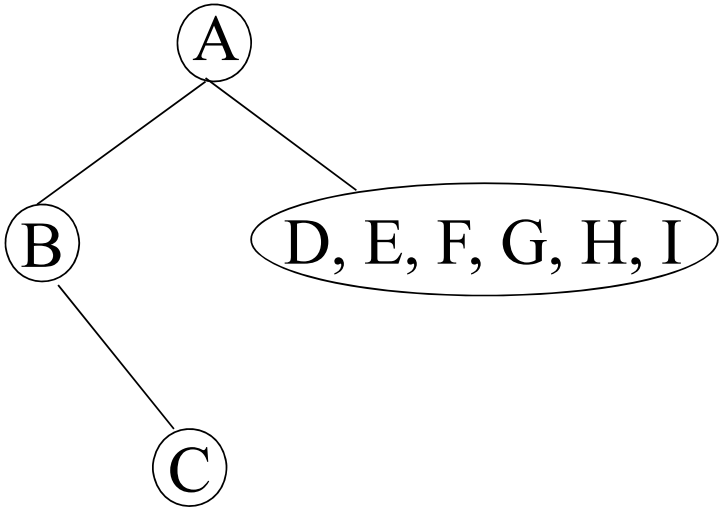
Tree Traversals: Another Example

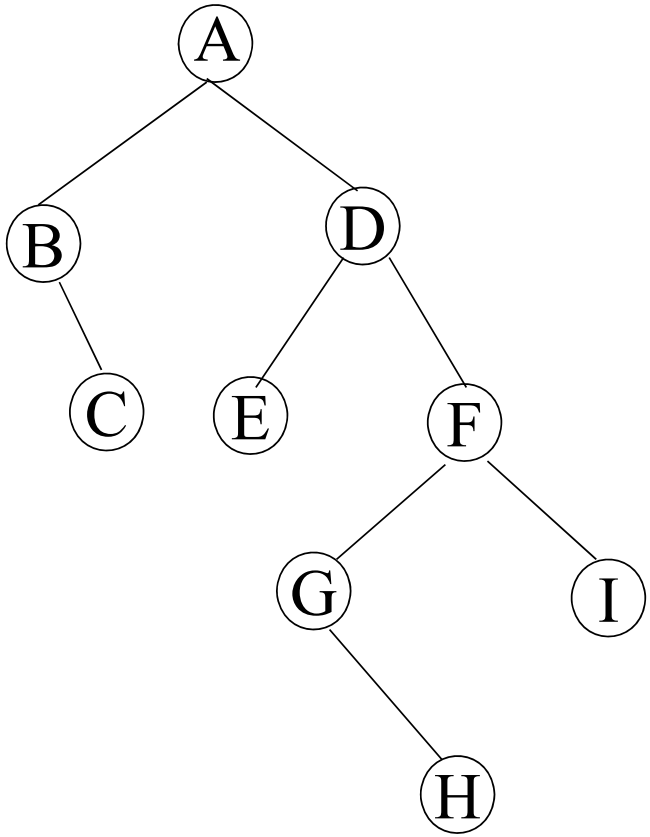


Preorder: ABDFHIECG
Postorder: HIFDEBGCA
In Order: DHFIBEACG

preorder: A B C D E F G H I
inorder: B C A E D G H F I







Implementations

```
void preorder (node *root)
{
    if (root != NULL)
    {
        printf ("%d ", root->element);
        preorder (root->left);
        preorder (root->right);
    }
}
```

```
void inorder (node *root)
{
    if (root != NULL)
    {
        inorder (root->left);
        printf ("%d ", root->element);
        inorder (root->right);
    }
}
```

```
void postorder (node *root)
{
    if (root != NULL)
    {
        postorder (root->left);
        postorder (root->right);
        printf ("%d ", root->element);
    }
}
```

Tree Algorithms

```
Insert (TREE, VAL)

Step 1: IF TREE = NULL
            Allocate memory for TREE
            SET TREE -> DATA = VAL
            SET TREE -> LEFT = TREE -> RIGHT = NULL
        ELSE
            IF VAL < TREE -> DATA
                  Insert(TREE -> LEFT, VAL)
            ELSE
                  Insert(TREE -> RIGHT, VAL)
            [END OF IF]
        [END OF IF]
Step 2: END
```

```
searchElement (TREE, VAL)

Step 1: IF TREE -> DATA = VAL OR TREE = NULL
          Return TREE
        ELSE
         IF VAL < TREE -> DATA
          Return searchElement(TREE -> LEFT, VAL)
         ELSE
          Return searchElement(TREE -> RIGHT, VAL)
         [END OF IF]
        [END OF IF]
Step 2: END
```

```
Delete (TREE, VAL)

Step 1: IF TREE = NULL
          Write "VAL not found in the tree"
        ELSE IF VAL < TREE -> DATA
          Delete(TREE->LEFT, VAL)
        ELSE IF VAL > TREE -> DATA
          Delete(TREE -> RIGHT, VAL)
        ELSE IF TREE -> LEFT AND TREE -> RIGHT
          SET TEMP = findLargestNode(TREE -> LEFT)
          SET TREE -> DATA = TEMP -> DATA
          Delete(TREE -> LEFT, TEMP -> DATA)
        ELSE
          SET TEMP = TREE
         IF TREE -> LEFT = NULL AND TREE -> RIGHT = NULL
              SET TREE = NULL
          ELSE IF TREE -> LEFT != NULL
               SET TREE = TREE -> LEFT
          ELSE
               SET TREE = TREE -> RIGHT
          [END OF IF]
          FREE TEMP
        [END OF IF]
Step 2: END
```

Tree Algorithms

```
Height (TREE)

Step 1: IF TREE = NULL
            Return 0
        ELSE
         SET LeftHeight = Height(TREE -> LEFT)
         SET RightHeight = Height(TREE -> RIGHT)
         IF LeftHeight > RightHeight
              Return LeftHeight + 1
         ELSE
              Return RightHeight + 1
         [END OF IF]
        [END OF IF]
Step 2: END
```

```
totalInternalNodes(TREE)

Step 1: IF TREE = NULL
             Return 0
        [END OF IF]
        IF TREE -> LEFT = NULL AND TREE -> RIGHT = NULL
             Return 0
        ELSE
            Return totalInternalNodes(TREE -> LEFT) +
             totalInternalNodes(TREE -> RIGHT) + 1
       [END OF IF]
Step 2: END
```

```
totalNodes(TREE)

Step 1: IF TREE = NULL
            Return 0
        ELSE
            Return totalNodes(TREE -> LEFT)
                + totalNodes(TREE -> RIGHT) + 1
        [END OF IF]
Step 2: END
```

```
totalExternalNodes(TREE)

Step 1: IF TREE = NULL
            Return 0
       ELSE IF TREE -> LEFT = NULL AND TREE -> RIGHT = NULL
            Return 1
        ELSE
            Return totalExternalNodes(TREE -> LEFT) +
            totalExternalNodes(TREE -> RIGHT)
        [END OF IF]
Step 2: END
```

Tree Algorithms

```
MirrorImage(TREE)

Step 1: IF TREE != NULL
            MirrorImage(TREE -> LEFT)
            MirrorImage(TREE -> RIGHT)
            SET TEMP = TREE -> LEFT
            SET TREE -> LEFT = TREE -> RIGHT
            SET TREE -> RIGHT = TEMP
        [END OF IF]
Step 2: END
```

```
deleteTree(TREE)

Step 1: IF TREE != NULL
            deleteTree (TREE -> LEFT)
            deleteTree (TREE -> RIGHT)
            Free (TREE)
        [END OF IF]
Step 2: END
```

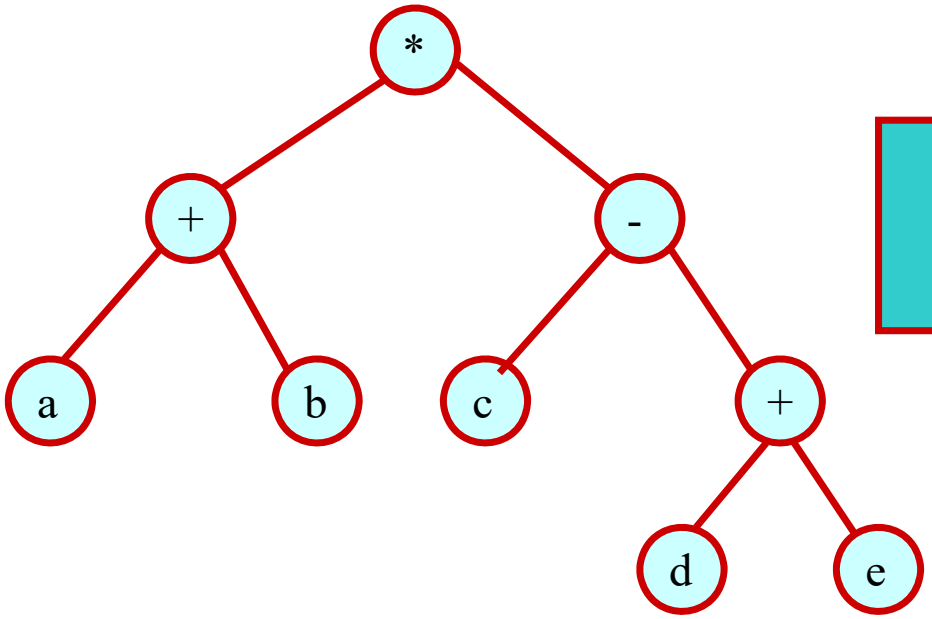
```
findSmallestElement(TREE)

Step 1: IF TREE = NULL OR TREE -> LEFT = NULL
            Returen TREE
        ELSE
            Return findSmallestElement(TREE -> LEFT)
        [END OF IF]
Step 2: END
```

```
findLargestElement(TREE)

Step 1: IF TREE = NULL OR TREE -> RIGHT = NULL
            Return TREE
        ELSE
            Return findLargestElement(TREE -> RIGHT)
        [END OF IF]
Step 2: END
```

A case study (Application) :: Expression Tree



Represents the expression:
(a + b) * (c – (d + e))

Preorder traversal :: * + a b - c + d e

Postorder traversal :: a b + c d e + - *

Threaded Binary Trees

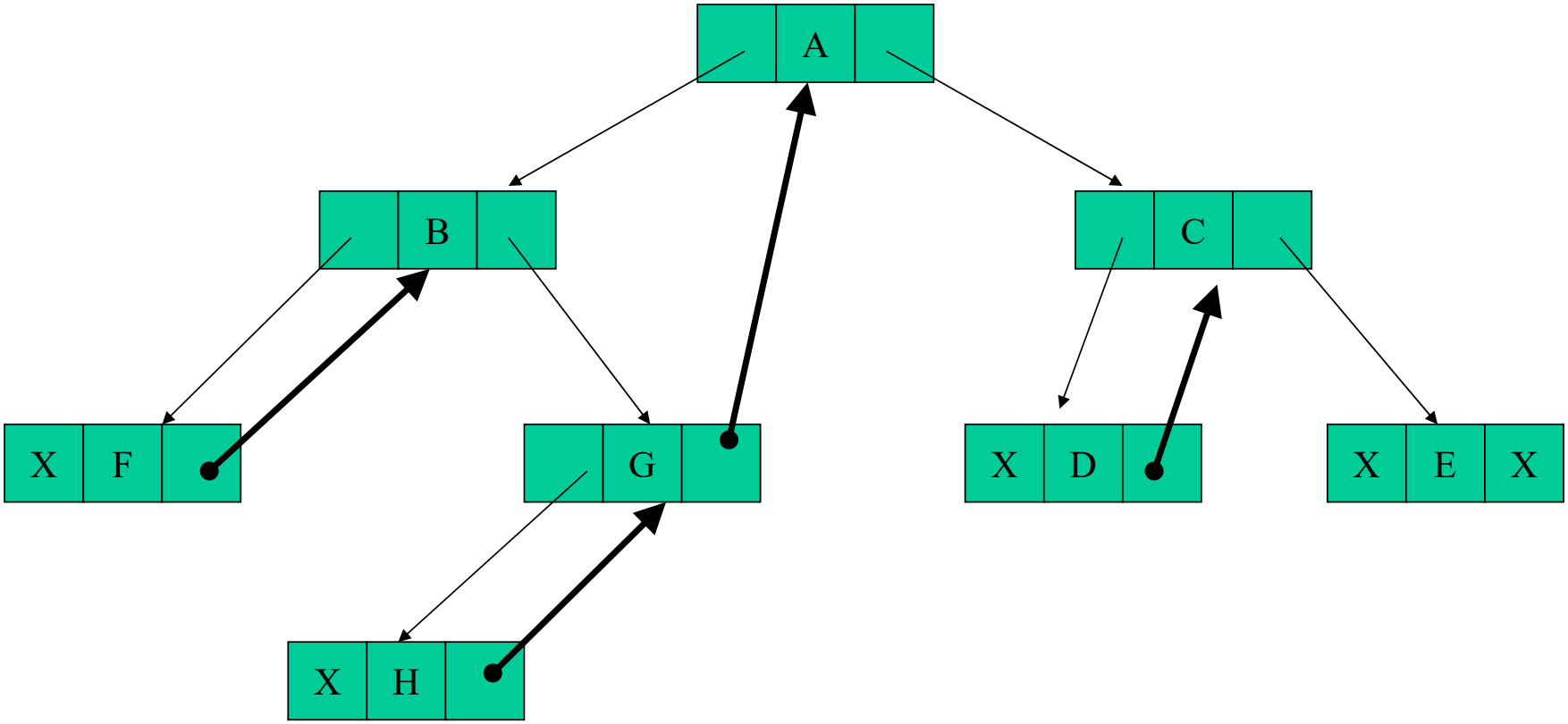
- On carefully examining the linked representation of a binary tree T , we will find that approximately half of the pointer fields contains NULL entries.
- The space occupied by these NULL entries can be utilized to store some kind of valuable information.
- One way to utilize this space is that we can store special pointer that points to nodes higher in the tree, i.e. ancestors.
- These special pointer are called **threads**, and the binary tree having such pointers is called a **threaded binary tree.**
- In computer memory, an extra field, called **tag or flag** is used to distinguish thread from a normal pointer.

Threaded Binary Trees

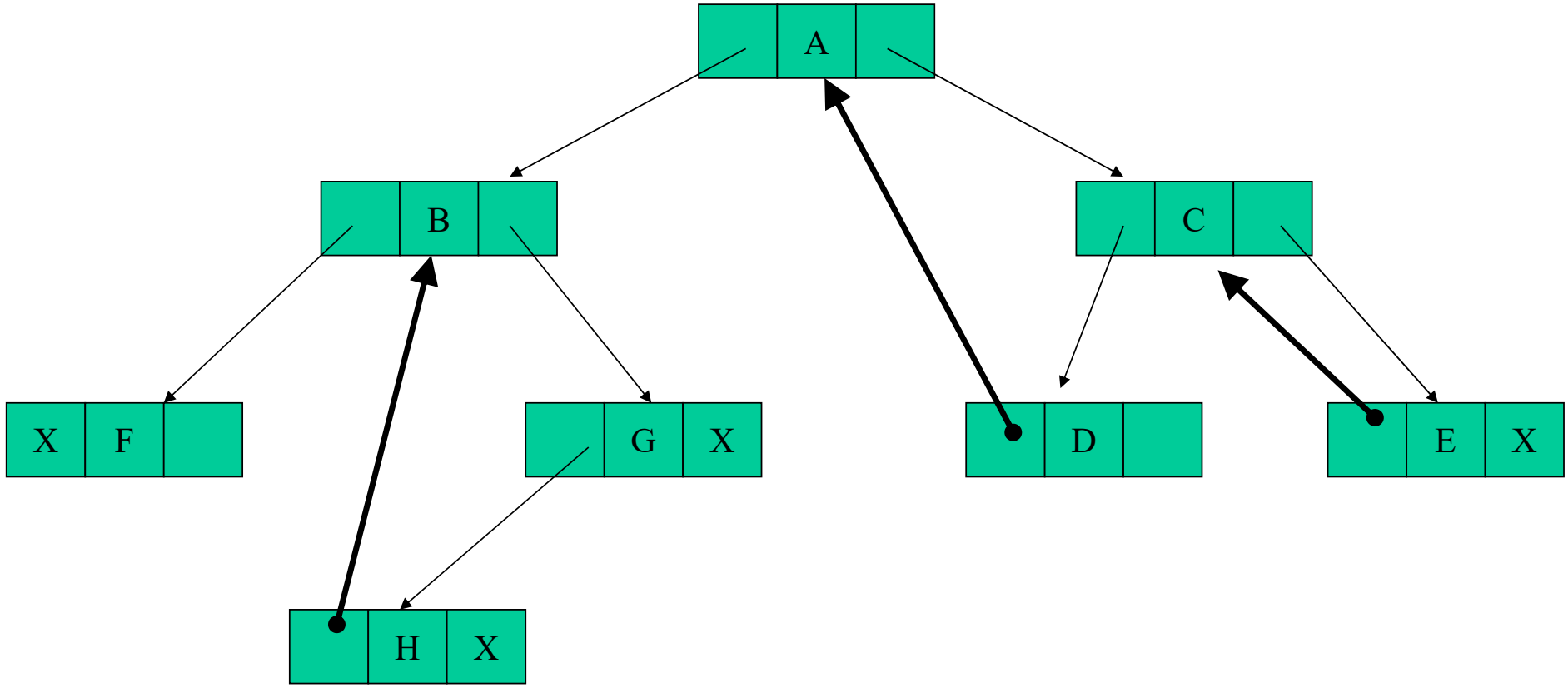
There are many ways to thread a binary tree:

- The right NULL pointer of each node can be replaced by a thread to the successor of the node under in-order traversal called a right thread, and the tree will called a **right threaded tree.**
- The left NULL pointer of each node can be replaced by a thread to the predecessor of the node under in-order traversal called a left thread, and the tree will called a **left threaded tree.**
- Both left and right NULL pointers can be used to point to predecessor and successor of that node respectively, under in-order traversal. Such a tree is called a **fully threaded tree.**
- **A threaded binary tree where only one thread is used is known as one way threaded tree and where both thread are used called two way threaded tree.**

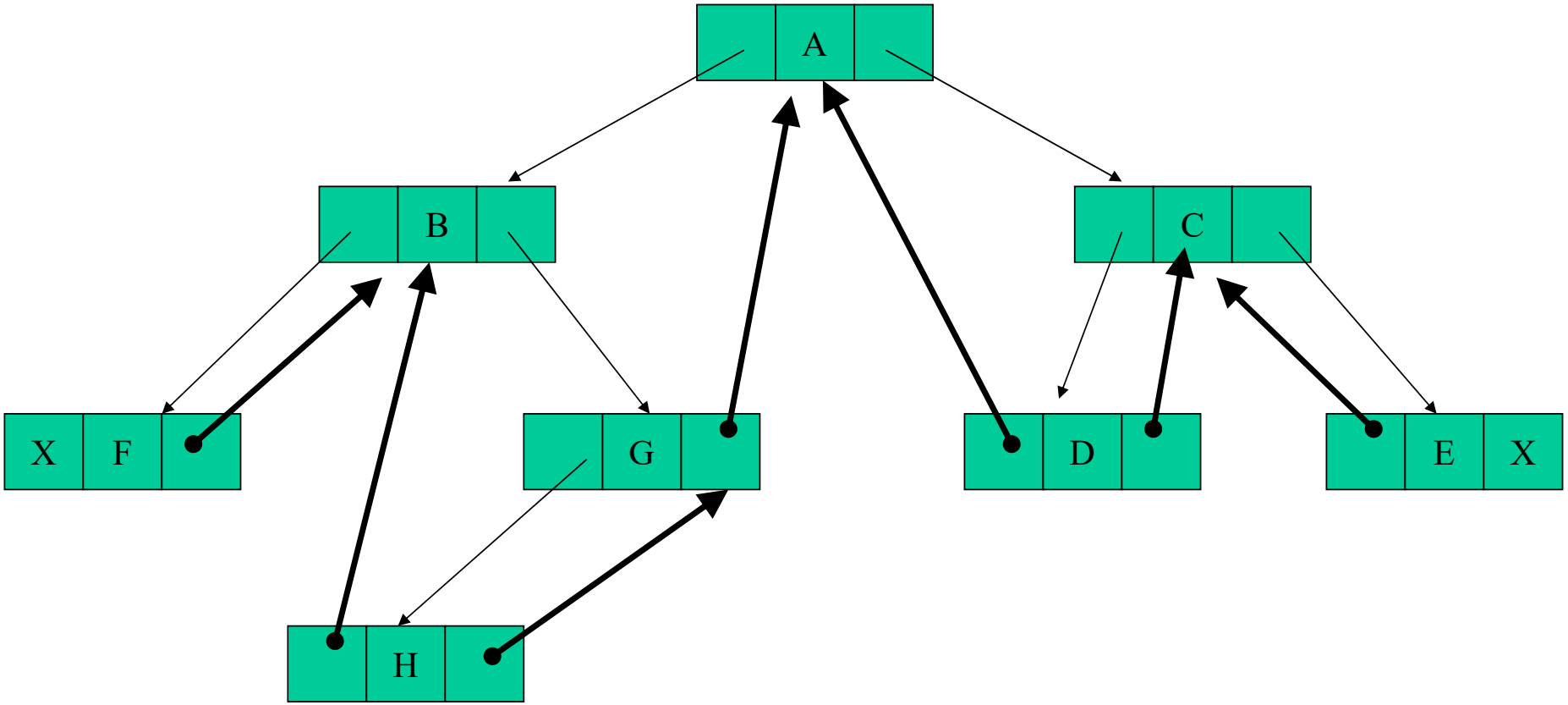
Right threaded binary tree (one way threading)



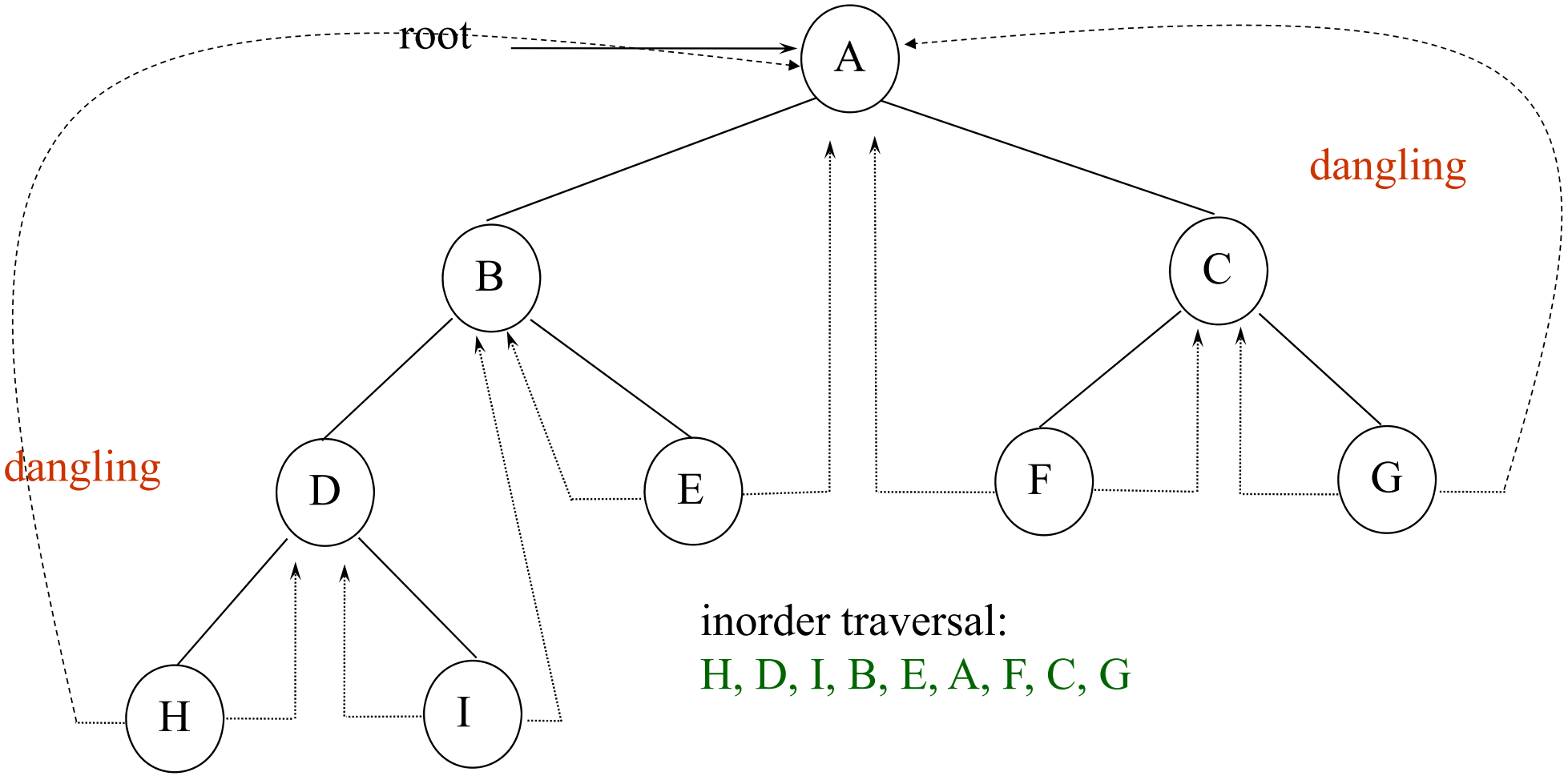
Left threaded binary tree (one way threading)



Fully threaded binary tree (two way threading)

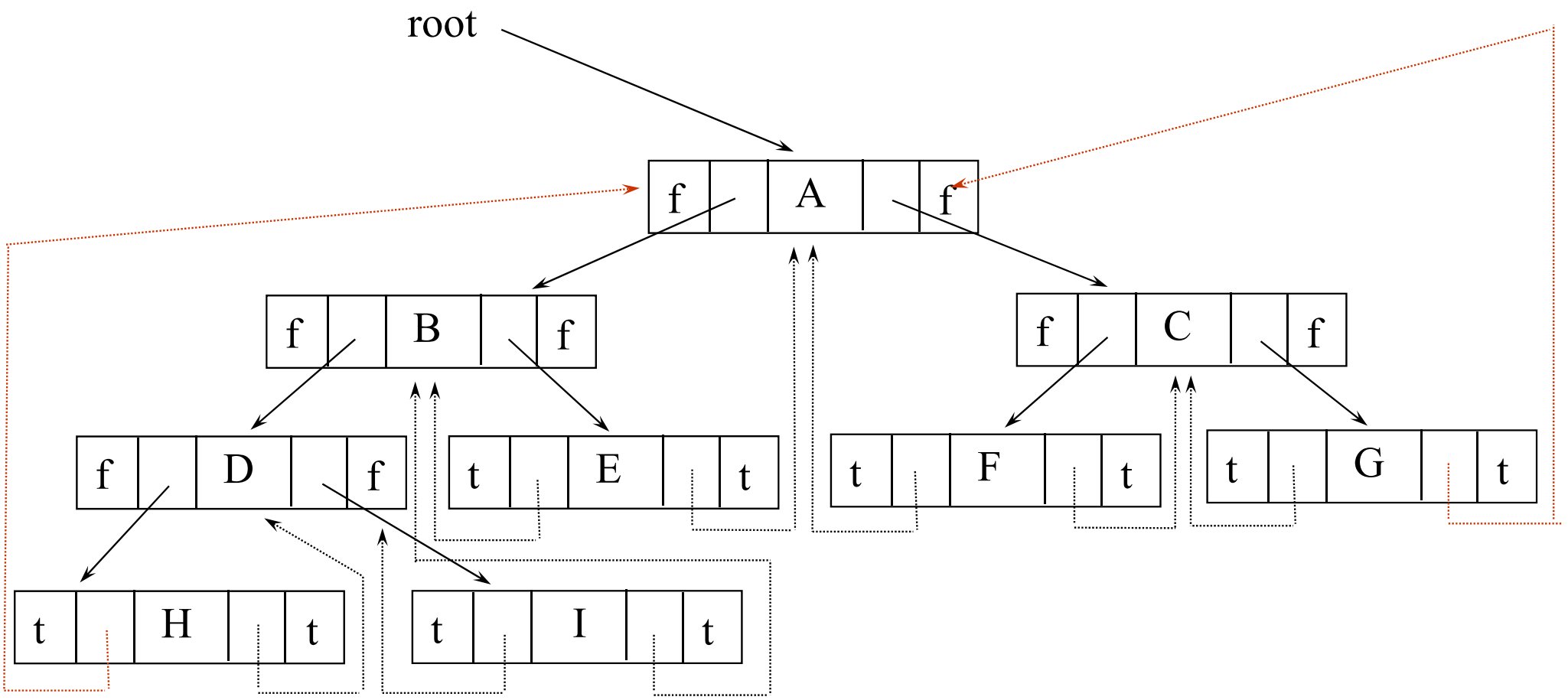


A Threaded Binary Tree



inorder traversal:
H, D, I, B, E, A, F, C, G

Memory Representation of A Threaded Binary Tree



Representation of threaded binary tree in memory

```
typedef struct nodetype
        {
                struct nodetype *left;
                int info;
                char thread;
                struct nodetype *right;
        }TBST;
TBST *root;
```

In this representation, we have used char field thread as a tag.

The character ‘0’ will used for normal right pointer and character ‘1’ will be used for thread.

Traversing a Threaded Binary Tree

Step 1: Check if the current node has a left child that has not been visited.

If a left child exists that has not been visited,

go to Step 2,

else go to Step 3.

Step 2: Add the left child in the list of visited nodes.

Make it as the current node and then go to Step 6.

Step 3: If the current node has a right child,

go to Step 4 else go to Step 5.

Step 4: Make that right child as current node and go to Step 6.

Step 5: Print the node and if there is a threaded node make it the current node.

Step 6: If all the nodes have visited then END else go to Step 1.

Heap Sort

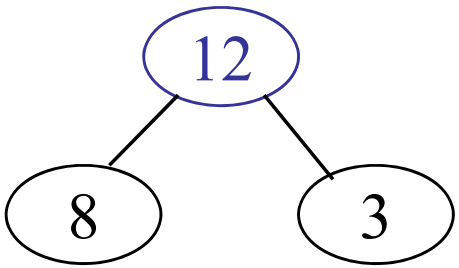
A max heap provides an efficient way to retrieve the maximum elements from an array.
The root of the heap is always the largest element since all child nodes must have smaller values.

STEPS OF A HEAP SORT ALGORITHM

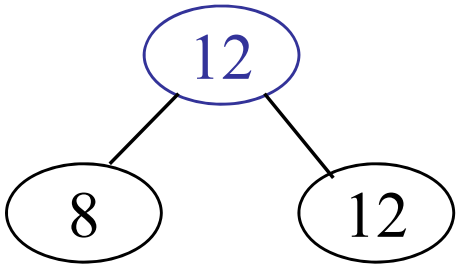
1. Transform the array into a binary tree by inserting each element as a node in a breadth-first manner.
2. Convert the binary tree into a max heap, ensuring that all parent nodes are greater than or equal to their child nodes.
3. Swap the root node — the largest element — with the last element in the heap.
4. Call the heapify() function to restore the max heap property.
5. Repeat steps 3 and 4 until the heap is sorted, and exclude the last element from the heap on each iteration.
6. After each swap and heapify() call, ensure that the max heap property is satisfied.

The heap property

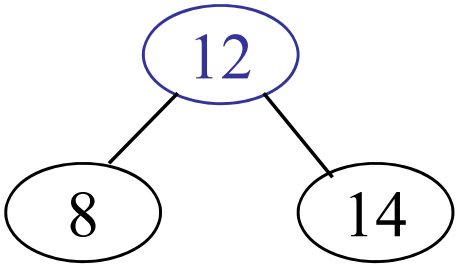
- A node has the heap property if the value in the node is as large as or larger than the values in its children



Blue node has heap property



Blue node has heap property

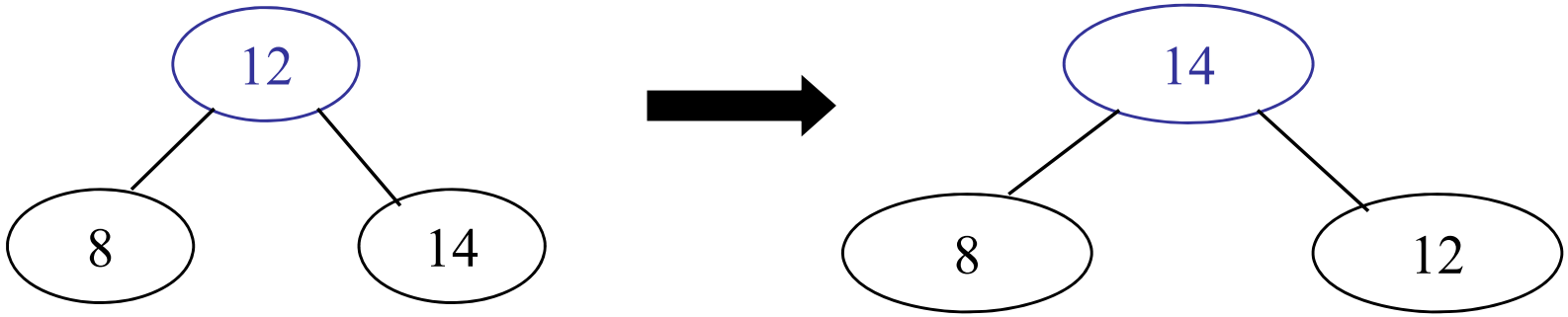


Blue node does not have heap property

- All leaf nodes automatically have the heap property
- A binary tree is a heap if *all* nodes in it have the heap property

The heap property: siftUp

- Given a node that does not have the heap property, you can give it the heap property by exchanging its value with the value of the larger child

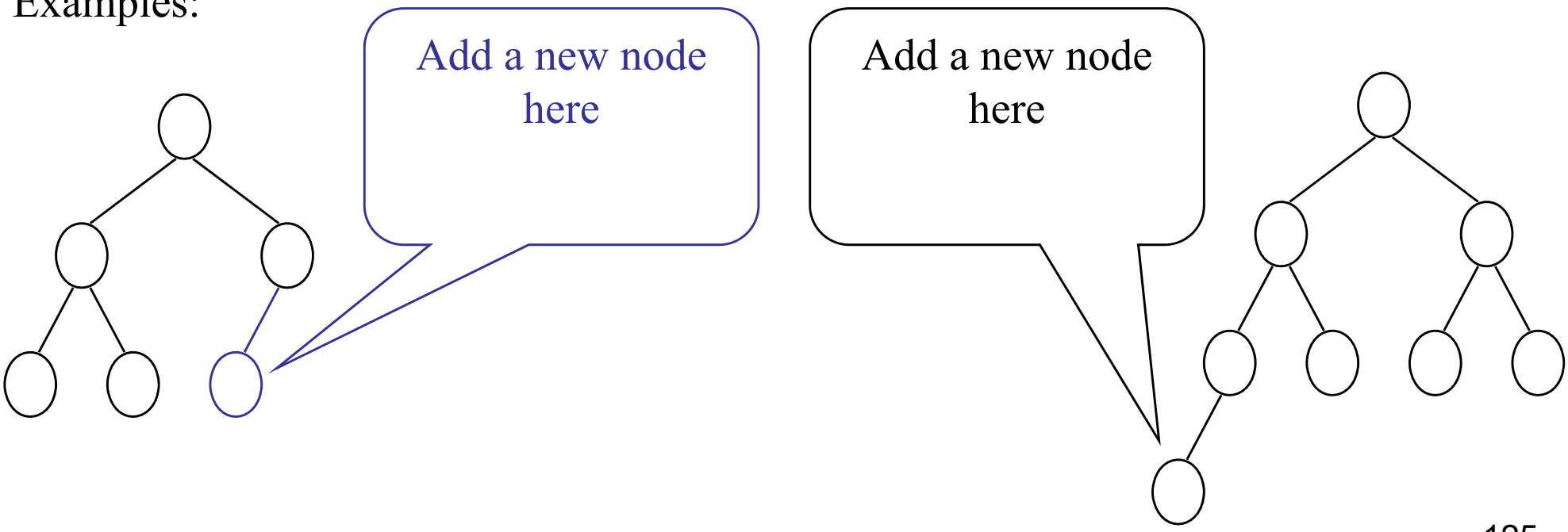


Blue node does not have heap property Blue node has heap property

- This is sometimes called sifting up

Constructing a heap tree

- A tree consisting of a single node is automatically a heap
- We construct a heap by adding nodes one at a time:
- Add the node just to the right of the rightmost node in the deepest level
- If the deepest level is full, start a new level
- Examples:



Constructing a heap tree

- Each time we add a node, we may destroy the heap property of its parent node
- To fix this, we sift up
- But each time we sift up, the value of the topmost node in the sift may increase, and this may destroy the heap property of its parent node
- We repeat the sifting up process, moving up in the tree, until either
- We reach nodes whose values don't need to be swapped (because the parent is larger than both children), or
- We reach the root

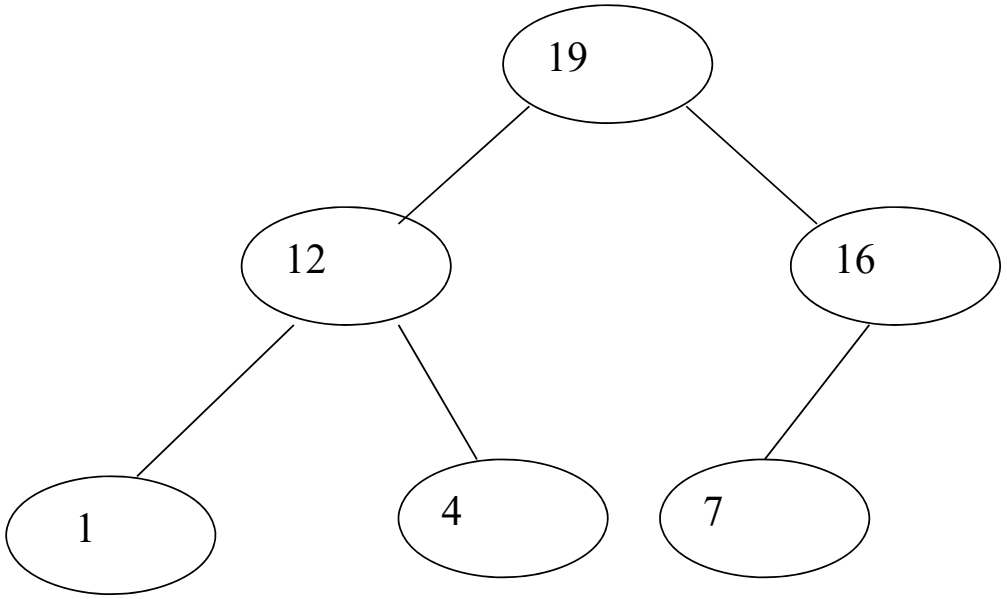
Definition

- Max Heap
  - Store data in ascending order
  - Has property of

    $A[Parent(i)] \geq A[i]$
- Min Heap
  - Store data in descending order
  - Has property of

    $A[Parent(i)] \leq A[i]$

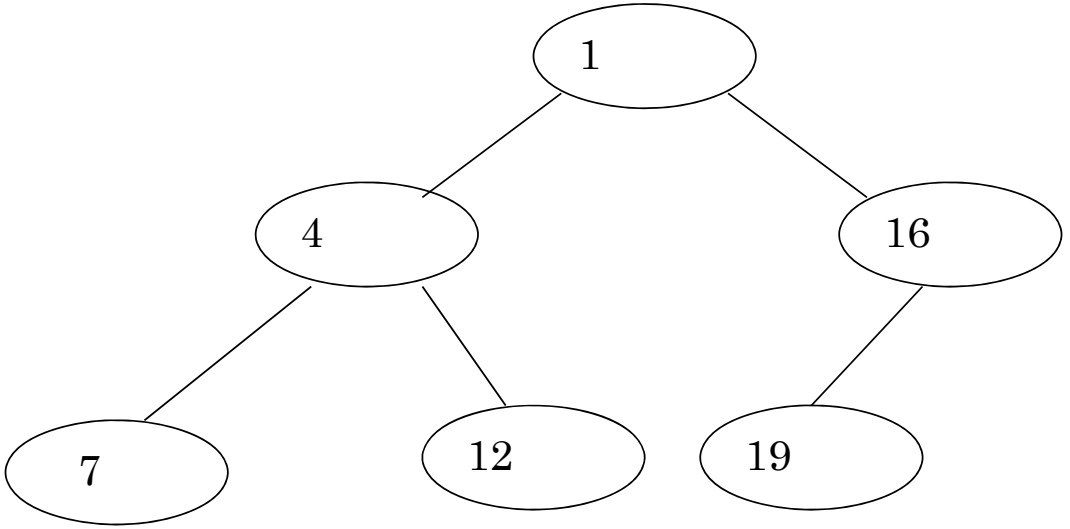
Max Heap Example



19	12	16	1	4	7

Array A

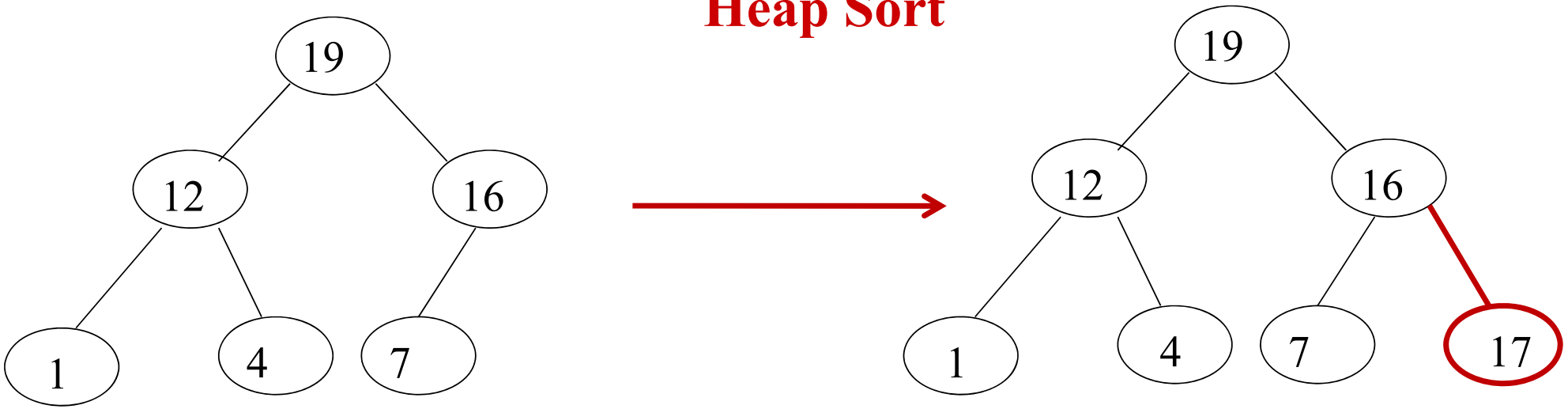
Min heap example

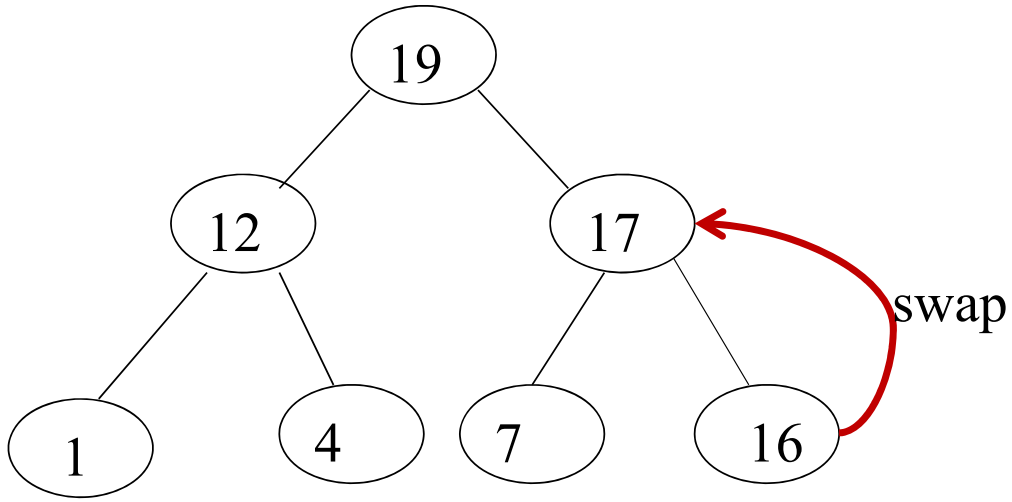


1	4	16	7	12	19

Array A

Heap Sort





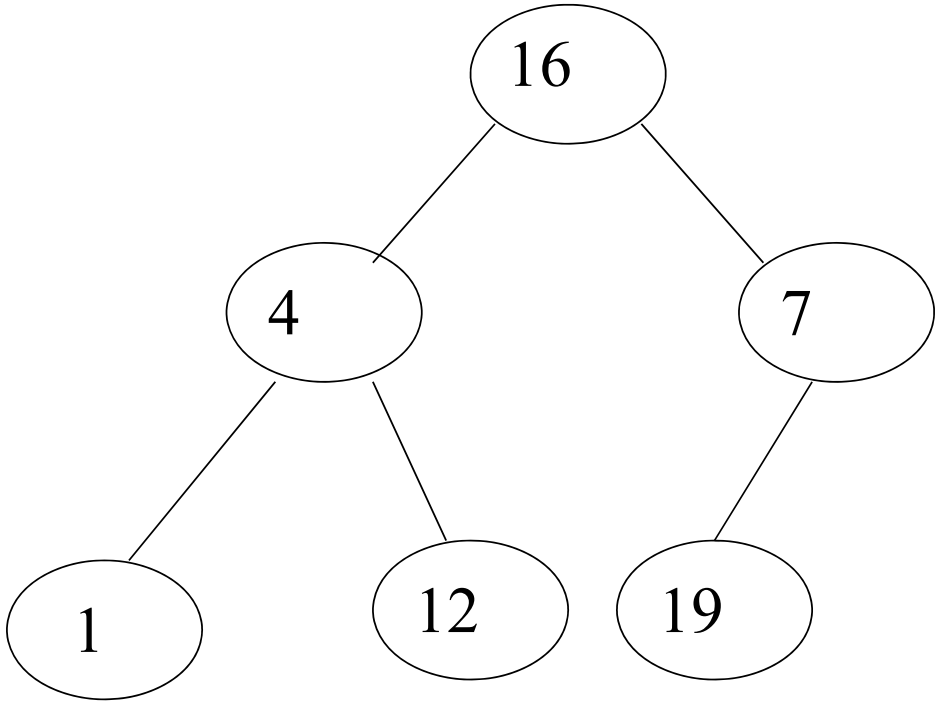
Percolate up to maintain the heap property

Heap Sort

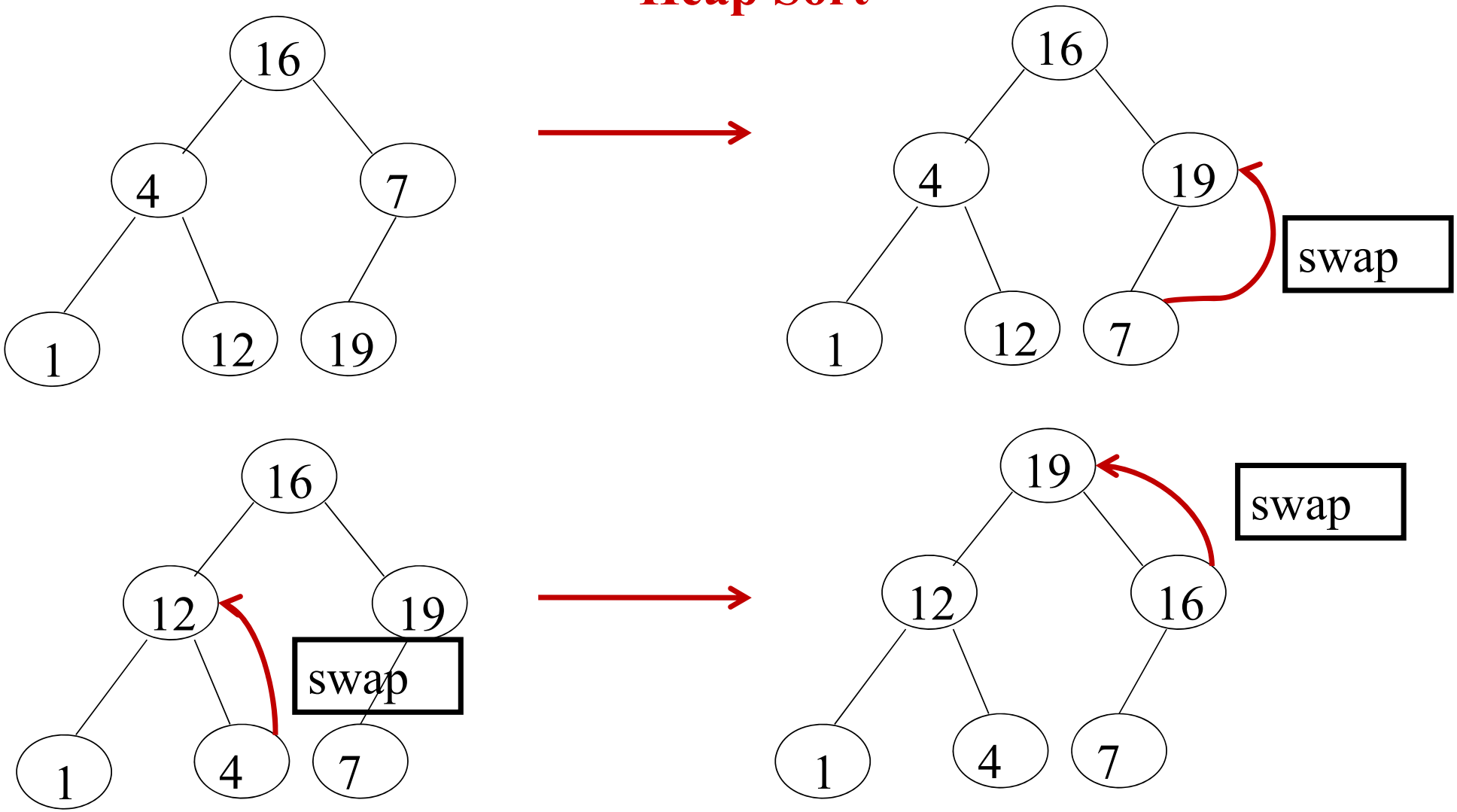
Example: Convert the following array to a heap

16	4	7	1	12	19

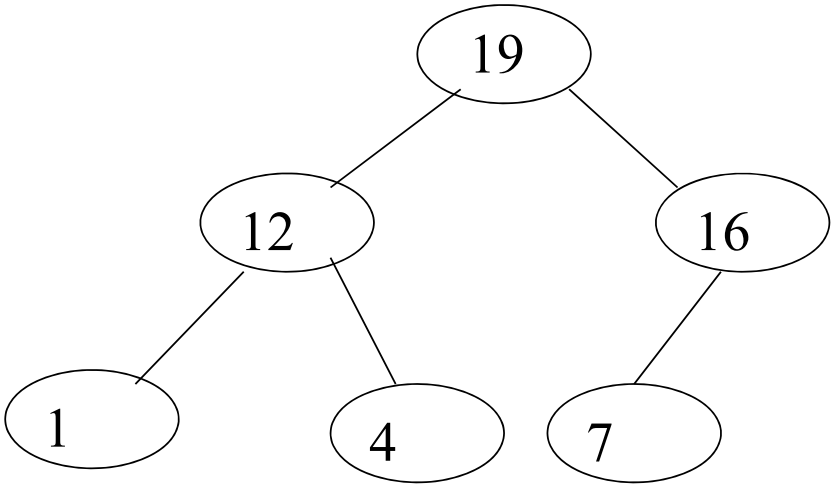
Picture **the array as a complete binary tree:**



Heap Sort



Example of Heap Sort



Array A

19	12	16	1	4	7

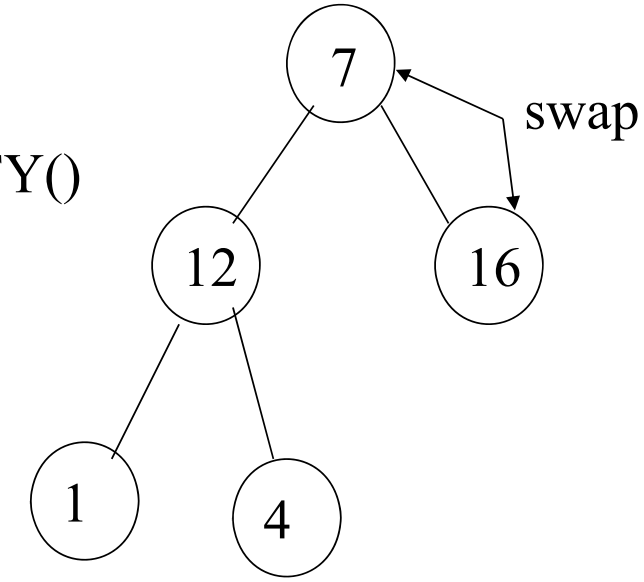
Sorted:

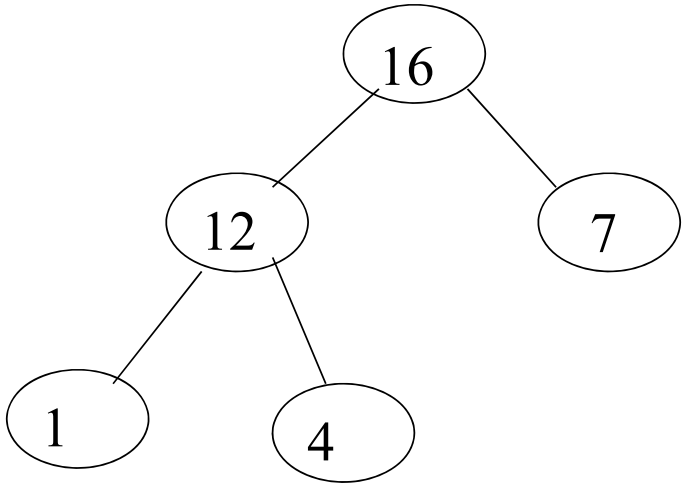
19

Example of Heap Sort

Array A | 7 | 12 | 16 | 1 | 4 |

HEAPIFY()





Sorted:

19

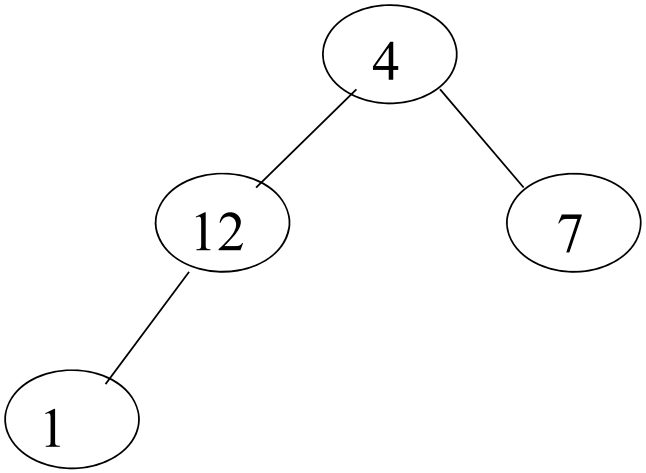
Array A

| 16 | 12 | 7 | 1 | 4 |

Sorted:

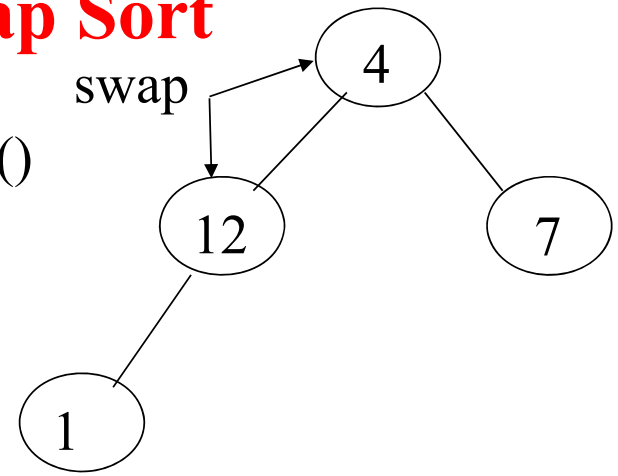
| 16 | 19 |

Example of Heap Sort



HEAPIFY()

swap



Array A

4	12	7	1

Sorted:

16	19

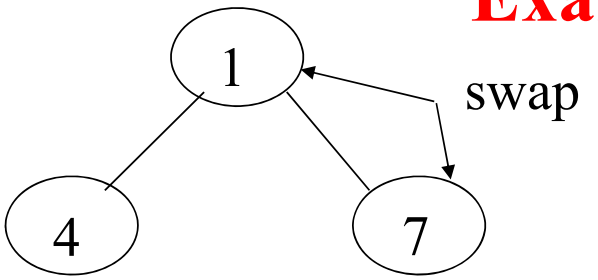
Array A

12	4	7	1

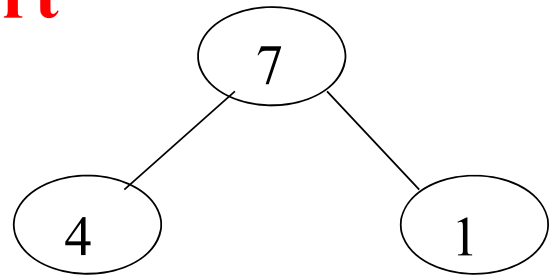
Sorted:

12	16	19

Example of Heap Sort



swap



Array A

1	4	7

Sorted:

12	16	19

Array A

7	4	1

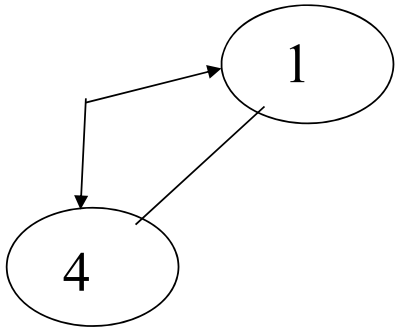
Sorted:

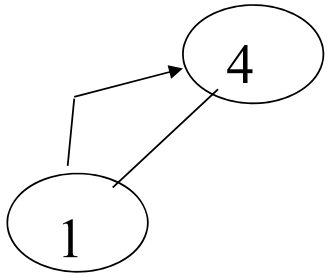
7	12	16	19

Example of Heap Sort

HEAPIFY()

swap





Array A

4	1

Sorted:

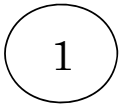
7	12	16	19

Array A

1

Sorted:

4	7	12	16	19



Sorted:

1	4	7	12	16	19

Heap Sort Complexity

- To reheap the root node, we have to follow *one path* from the root to a leaf node (and we might stop before we reach a leaf)
- The binary tree is perfectly balanced
- Therefore, this path is O(log n) long
  - And we only do O(1) operations at each node
  - Therefore, reheaping takes O(log n) times
- Since we reheap inside a while loop that we do n-1 times, the total time for the while loop is n*O(log n), or O(n log n)

- **Build Heap Algorithm will run in O(n) time**
- **Remove and re-heap** **O(log n)**
  - **Do this n-1 times** **O(n log n)**
- **Total time** **O(n log n) + O(n log n) = O(n log n)**

Introduction to B-Tree

- B-trees are balanced search tree.
- More than two children are possible.
- B-Tree, stores all information in the leaves and stores only keys and Child pointer.
- If an internal B-tree node x contains n[x] keys then x has n[x]+1 children.

Example of B-Tree

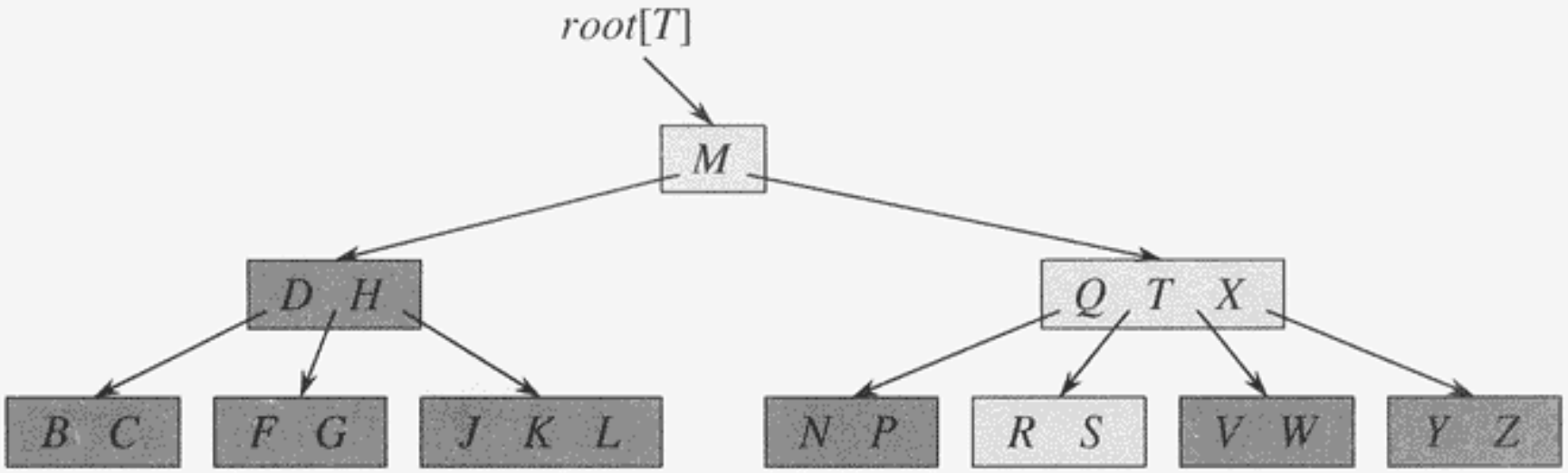
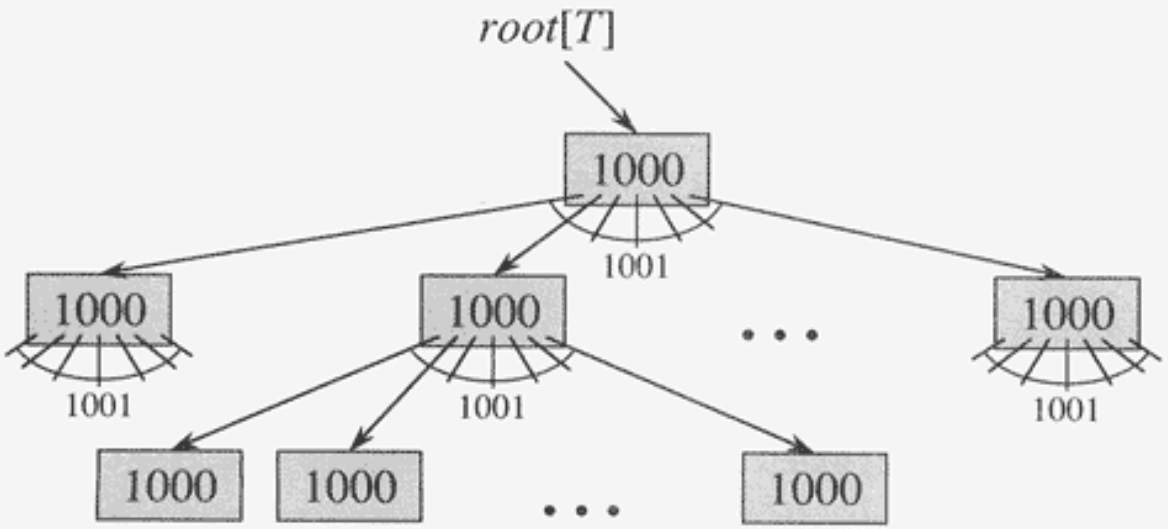


Figure 18.1 A B-tree whose keys are the consonants of English. An internal node x containing $n[x]$ keys has $n[x] + 1$ children. All leaves are at the same depth in the tree. The lightly shaded nodes are examined in a search for the letter R.

Another example of B-Tree



1 node,
1000 keys

1001 nodes,
1,001,000 keys

1,002,001 nodes,
1,002,001,000 keys

Figure 18.3 A B-tree of height 2 containing over one billion keys. Each internal node and leaf contains 1000 keys. There are 1001 nodes at depth 1 and over one million leaves at depth 2. Shown inside each node x is $n[x]$, the number of keys in x.

Application of B-Tree

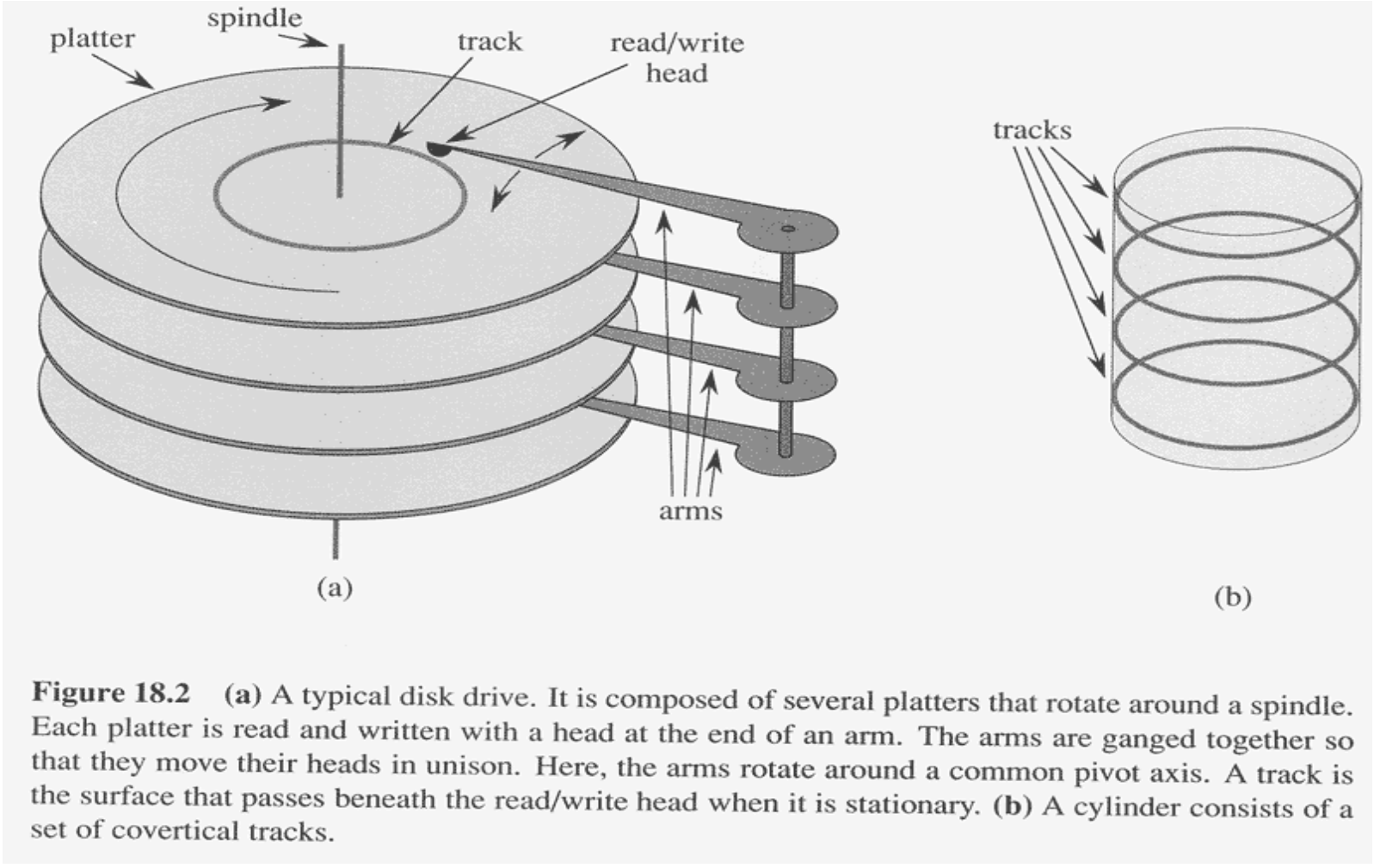


Figure 18.2 **(a)** A typical disk drive. It is composed of several platters that rotate around a spindle. Each platter is read and written with a head at the end of an arm. The arms are ganged together so that they move their heads in unison. Here, the arms rotate around a common pivot axis. A track is the surface that passes beneath the read/write head when it is stationary. **(b)** A cylinder consists of a set of covertical tracks.

It designed to work on magnetic disks or other (direct access) secondary storage devices.

Properties of B-Tree

- A B-Tree T is a rooted tree having the following properties:

1. Every node x has the following fields:
    1. n[x] the no. of keys currently stored in node x
    2. The n[x] keys themselves, stored in non-decreasing order, so that $key_1[x] \leq key_2[x] \ldots\ldots \leq key_{n-1}[x] \leq key_n[x]$.
    3. Leaf[x], a Boolean value that is TRUE if x is a leaf and FALSE if x is an internal node.
2. Each internal node x also contains n[x]+1 pointers (Childs) $c_1[x]$, $c_2[x]$,-------- $c_{n[x]+1}[x]$.
3. All leaves have the same depth, which is the tree's height h.
4. There are lower and upper bounds on the no. of keys a node can contains: these bounds can be expressed in terms of a fixed integer $t \geq 2$ called the minimum degree of B-Tree.

– Every node other than the root must have at least t-1 keys, then root has at least t children if the tree is non empty the root must have at least one key.

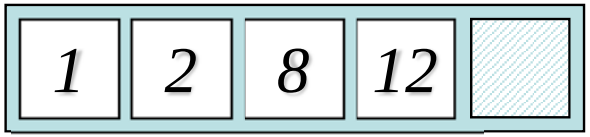
– Every node can contain at most 2t-1 keys. Therefore, an internal node can have at most 2t children we say that a node is full if it contains exactly 2t-1 keys.

B-Tree Insertion

1) B-tree starts with a single root node (which is also a leaf node) at level 0.
2) Once the root node is full with p – 1 search key values and when attempt to insert another entry in the tree, the root node splits into two nodes at level 1.
3) Only the middle value is kept in the root node, and the rest of the values are split evenly between the other two nodes.
4) When a non-root node is full and a new entry is inserted into it, that node is split into two nodes at the same level, and the middle entry is moved to the parent node along with two pointers to the new split nodes.
5) If the parent node is full, it is also split.
6) Splitting can propagate all the way to the root node, creating a new level if the root is split.

Constructing a B-tree

- Suppose we start with an empty B-tree and keys arrive in the following order: 1 12 8 2 25 6 14 28 17 7 52 16 48 68 3 26 29 53 55 45
- We want to construct a B-tree of degree 5
- The first four items go into the root:

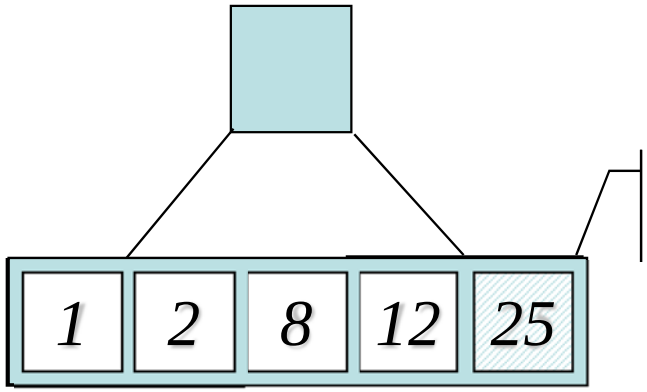


- To put the fifth item in the root would violate condition 5
- Therefore, when 25 arrives, pick the middle key to make a new root

1
12
8
2
25
6
14
28
17
7
52
16
48
68
3
26
29
53
55
45

Constructing a B-tree

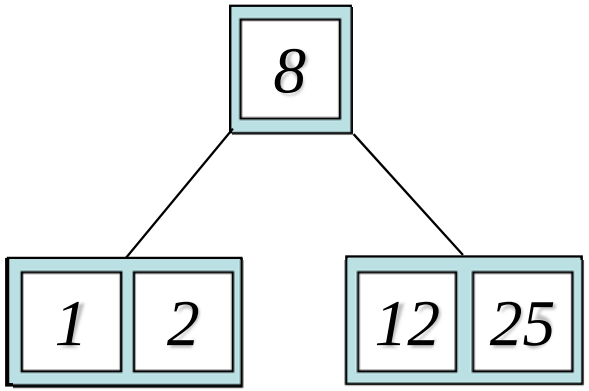
Add 6 to the tree



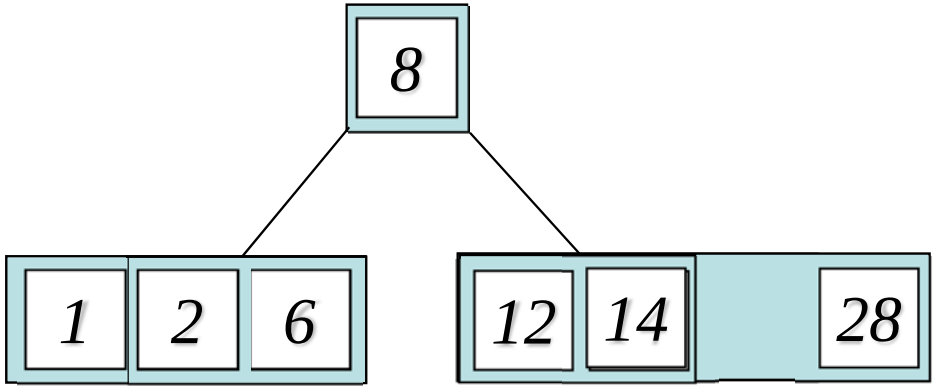
Exceeds Order. Promote middle and split.

1
12
8
2
25
6
14
28
17
7
52
16
48
68
3
26
29
53
55
45

Constructing a B-tree (contd.)



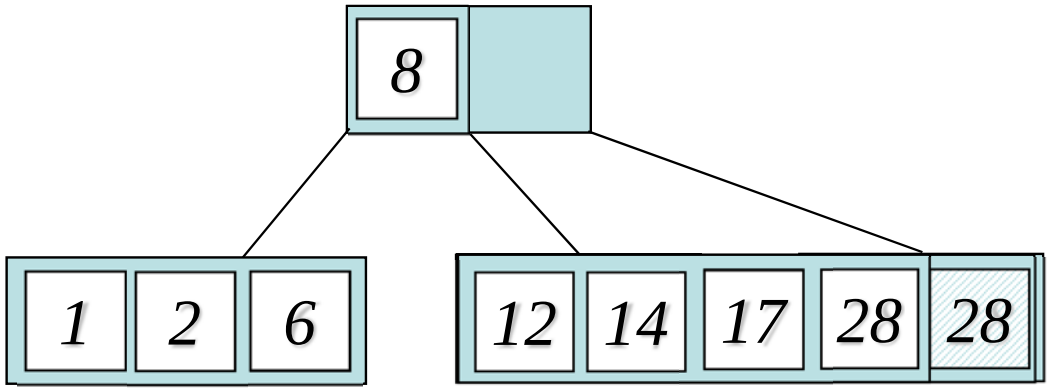
6, 14, 28 get added to the leaf nodes:



1
12
8
2
25
6
14
28
17
7
52
16
48
68
3
26
29
53
55
45

Constructing a B-tree (contd.)

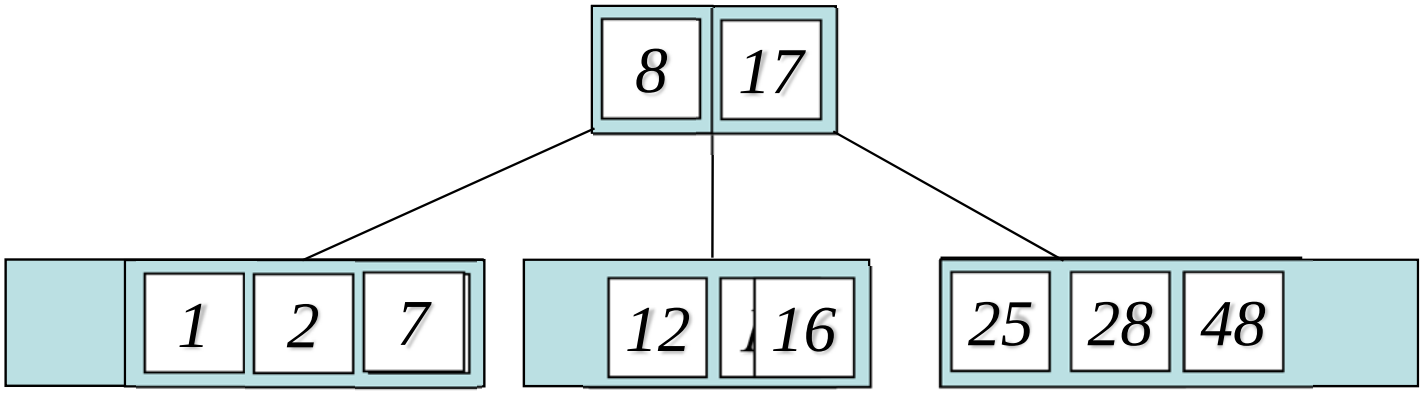
Adding 17 to the right leaf node would over-fill it, so we take the middle key, promote it (to the root) and split the leaf



1
12
8
2
25
6
14
28
17
7
52
16
48
68
3
26
29
53
55
45

Constructing a B-tree (contd.)

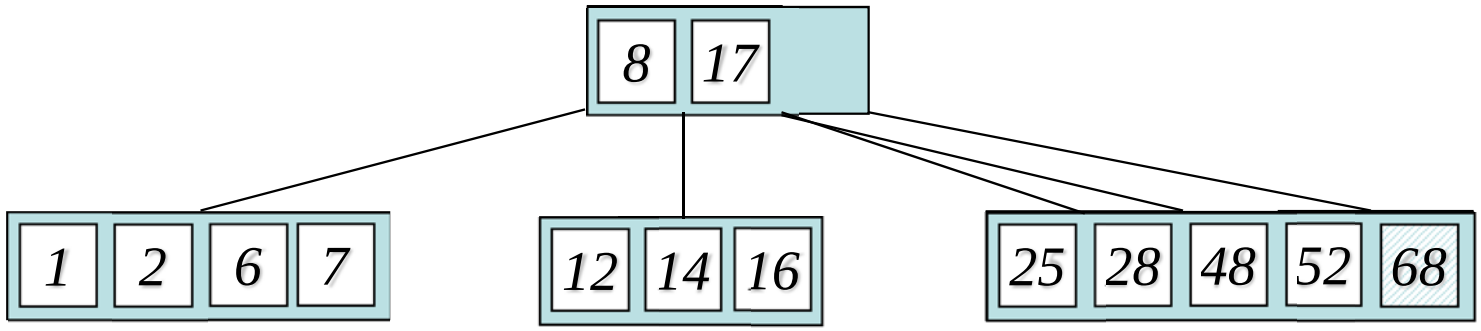
7, 52, 16, 48 get added to the leaf nodes



1
12
8
2
25
6
14
28
17
7
52
16
48
68
3
26
29
53
55
45

Constructing a B-tree (contd.)

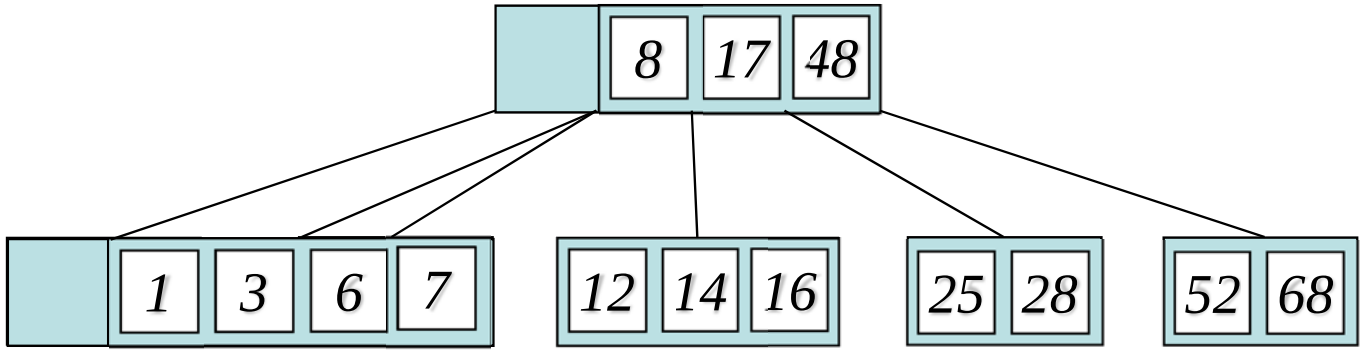
Adding 68 causes us to split the right most leaf, promoting 48 to the root



1
12
8
2
25
6
14
28
17
7
52
16
48
68
3
26
29
53
55
45

Constructing a B-tree (contd.)

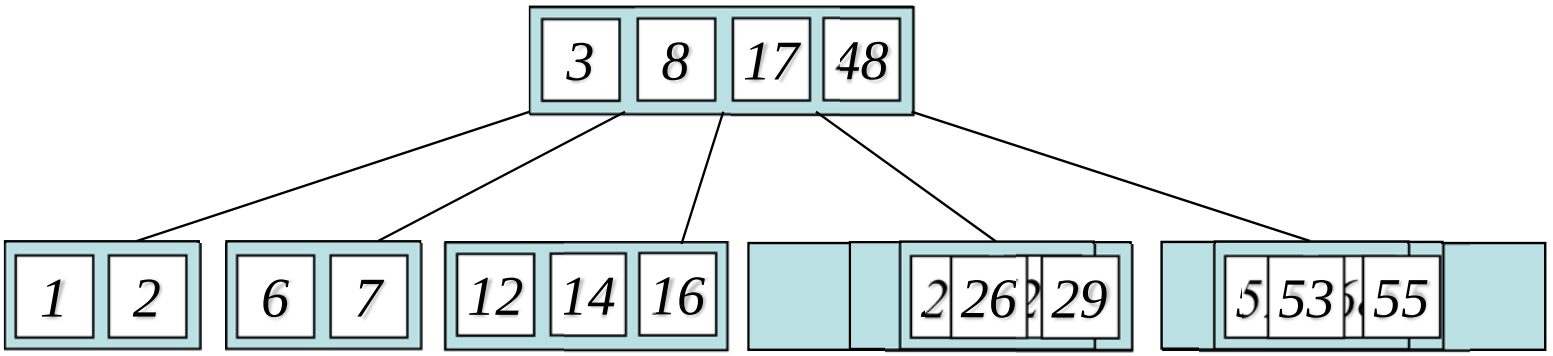
Adding 3 causes us to split the left most leaf



1
12
8
2
25
6
14
28
17
7
52
16
48
68
3
26
29
53
55
45

Constructing a B-tree (contd.)

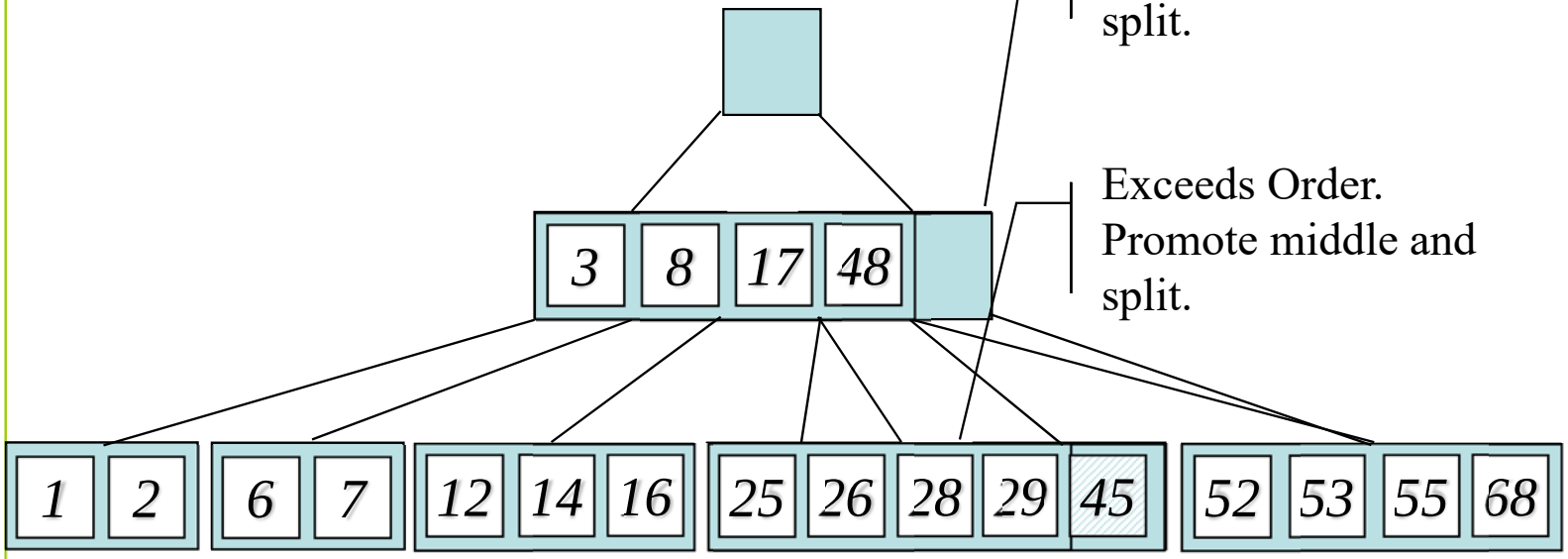
Add 26, 29, 53, 55 then go into the leaves



1
12
8
2
25
6
14
28
17
7
52
16
48
68
3
26
29
53
55
45

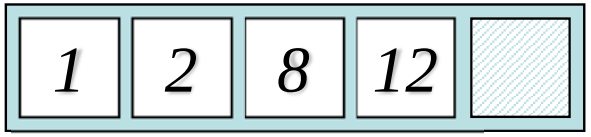
Constructing a B-tree (contd.)

Add 45 increases the trees level



Constructing a B+ Tree

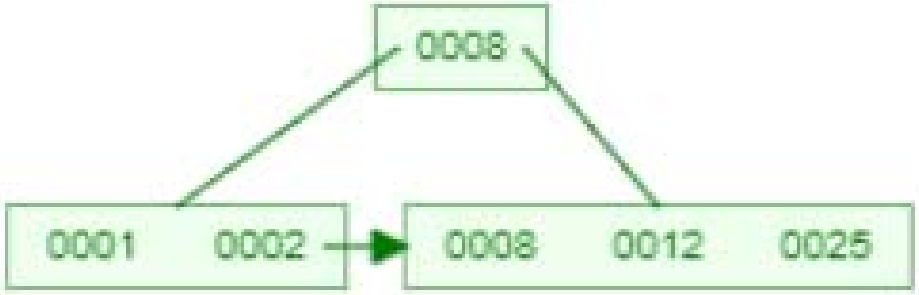
- Suppose we start with an empty B-tree and keys arrive in the following order: 1 12 8 2 25 6 14 28 17 7 52 16
- We want to construct a B+ tree of degree 5
- The first four items go into the root:

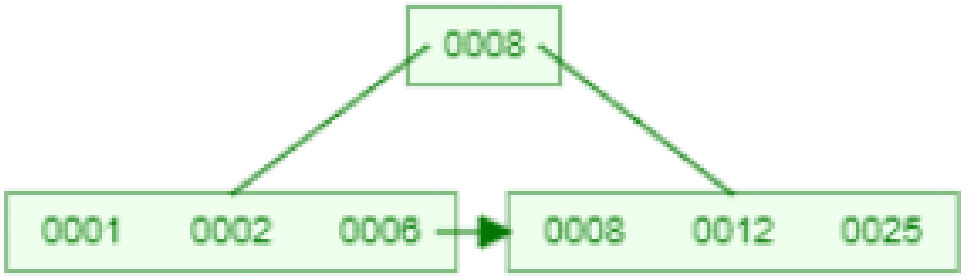


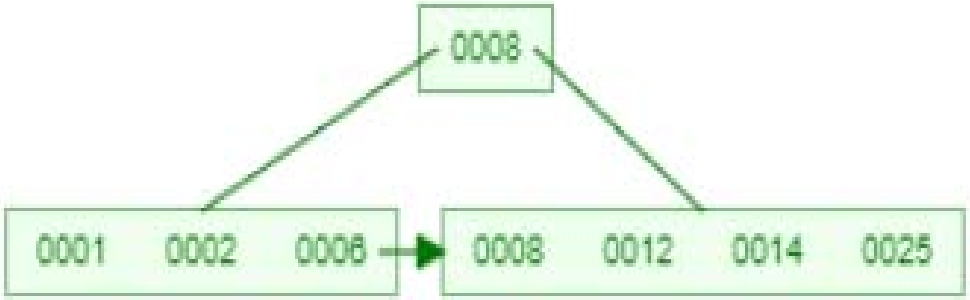
- To put the fifth item in the root would violate condition 5
- Therefore, when 25 arrives, pick the middle key to make a new root

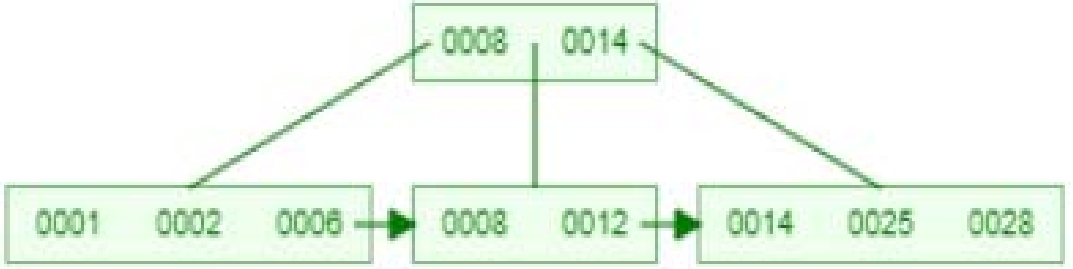
Constructing a B+ Tree

- Suppose we start with an empty B+ tree and keys arrive in the following order: 1 12 8 2 25 6 14 28 17 7 52 16



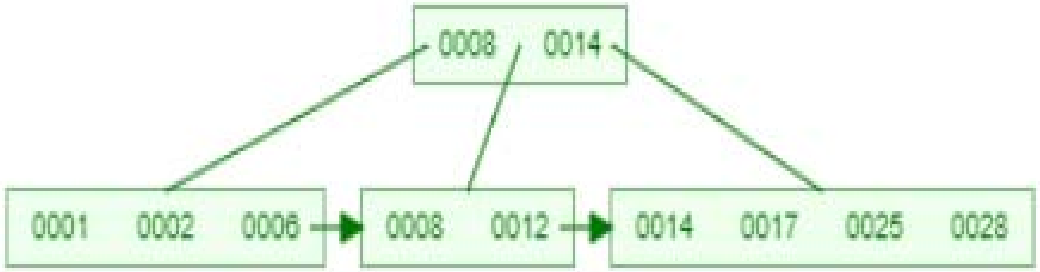


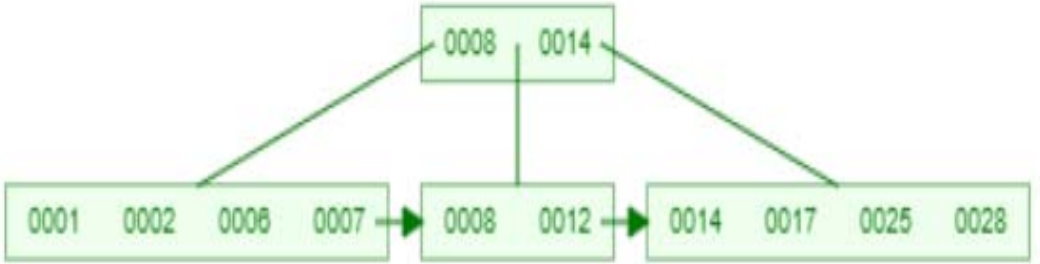


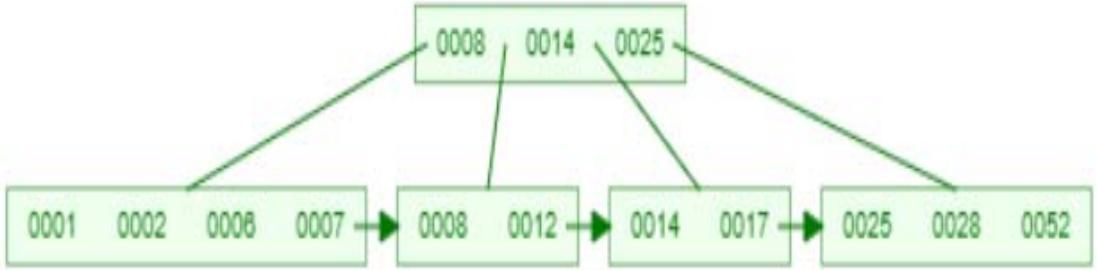


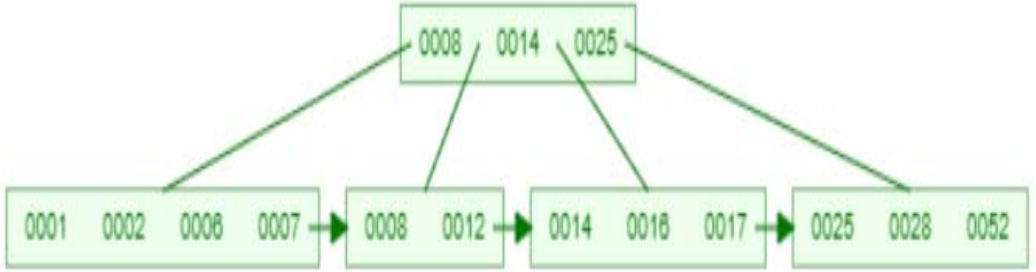
Constructing a B+ Tree

- Suppose we start with an empty B+ tree and keys arrive in the following order: 1 12 8 2 25 6 14 28 17 7 52 16







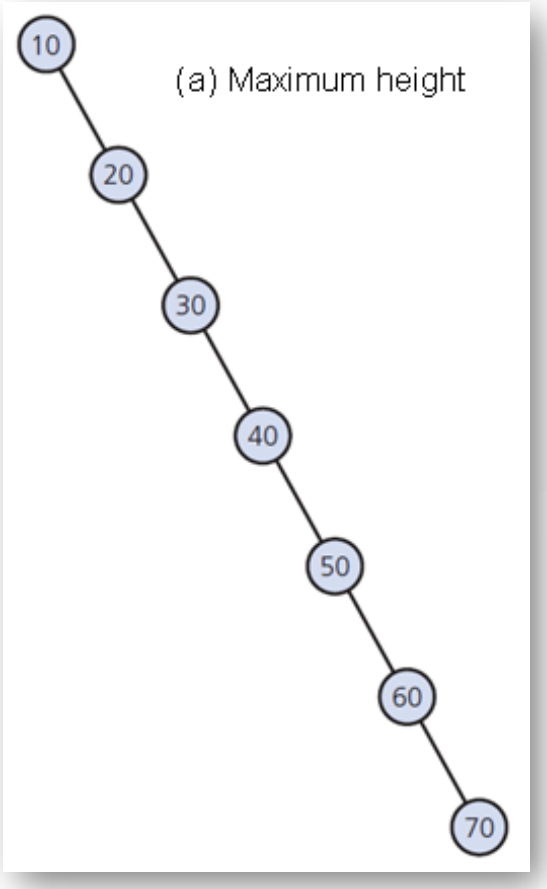


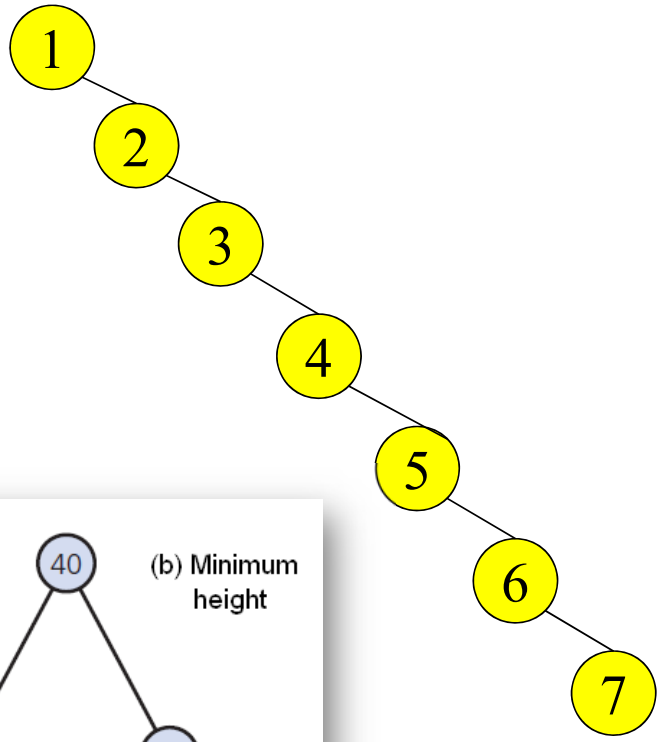
Exercise in Inserting a B+ Tree

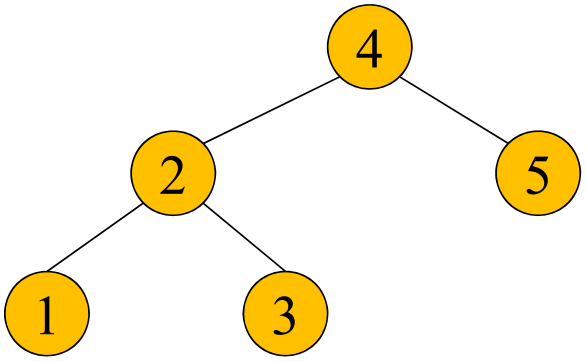
- Insert the following keys in B+ tree when t=3 :
- 3, 7, 9, 23, 45, 1, 5, 14, 25, 24, 13, 11, 8, 19, 4, 31, 35, 56

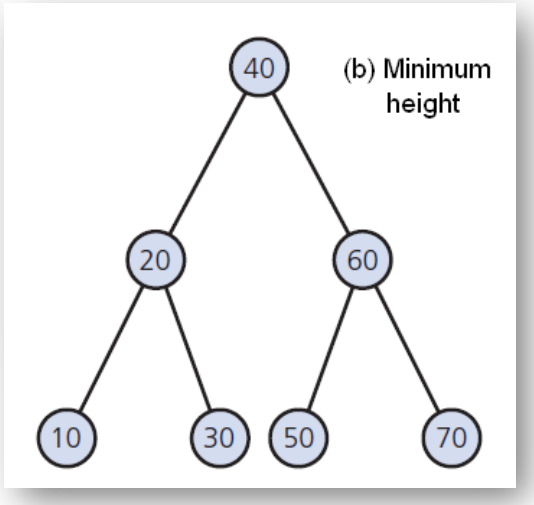
- Check your approach with a neighbour and discuss any differences.

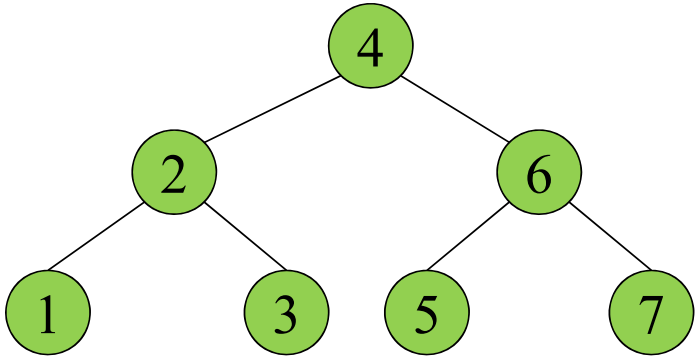
Balanced and unbalanced BST









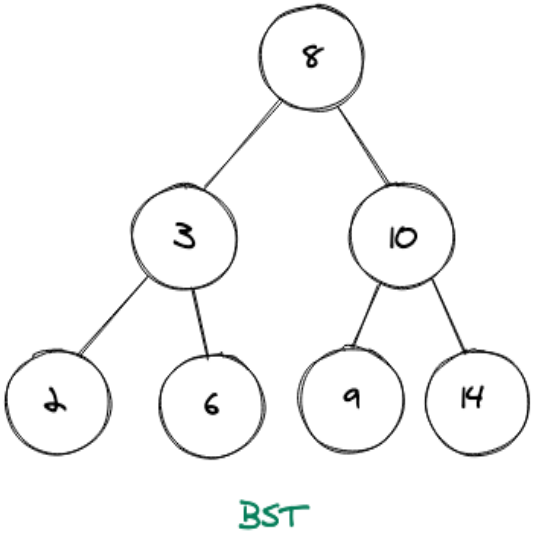


AVL Tree

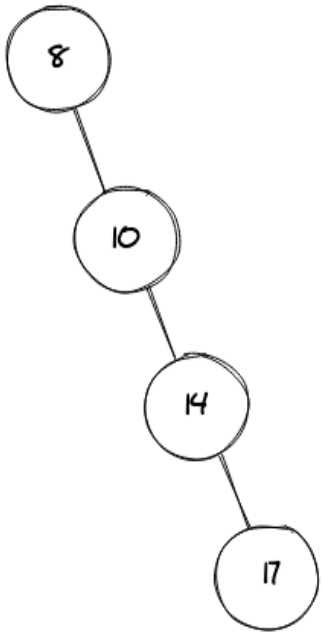
- AVL trees are nothing but self-balancing binary search trees.
- A self-balanced tree means that when we try to insert any element in this BST and if it violates the balancing factor of any node, then it dynamically rotates itself accordingly to make itself self-balanced.
- It is also well known that dynamically balanced trees.
- The height of the left and the right subtrees of every node differs by at most 1, which implies the difference between the heights will always be either 1,0 or -1.
- So, AVL trees are an optimized version of binary search trees named after its inventors G.M. Abelson-Velvety and E.M. Landis, in 1962.
- It was an attempt to improve the performance of Binary Search Trees.

Why AVL Tree

- When we already had a similar data structure (i.e. BST's), why would we need a complex version of it?.
- The answer lies in the fact that BST has some limitations in certain scenarios that make some operations like searching, inserting costly (as costly as O(n)).
- Consider the pictorial representation of two BST's below:



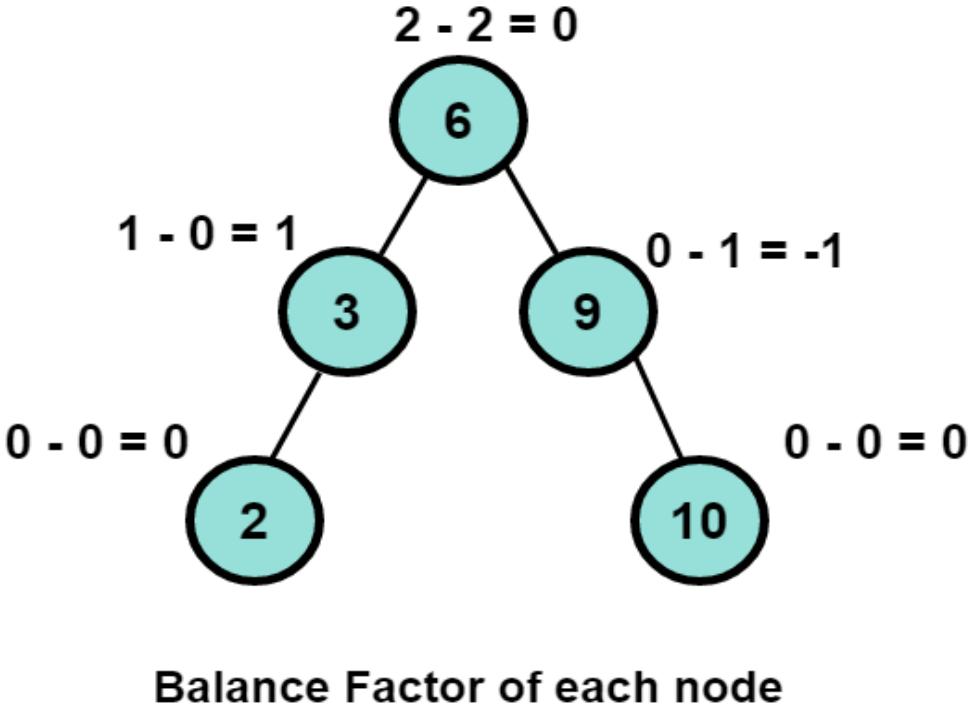
BST



Right Skewed BST

- The height of the left BST is O(log n) and the height of the right BST is O(n), thus the search operation time complexity for the left BST is O(log n) and for the right-skewed is O(n) which is its worst case too.
- Hence, if we have such a skewed tree then there's no benefit of using a BST, as it is just like a linked list when it comes to time and space complexity.
- That's why we needed a balanced BST's so that all the basic operations guarantee a time complexity of O(log n).

Balance Factor in AVL Trees



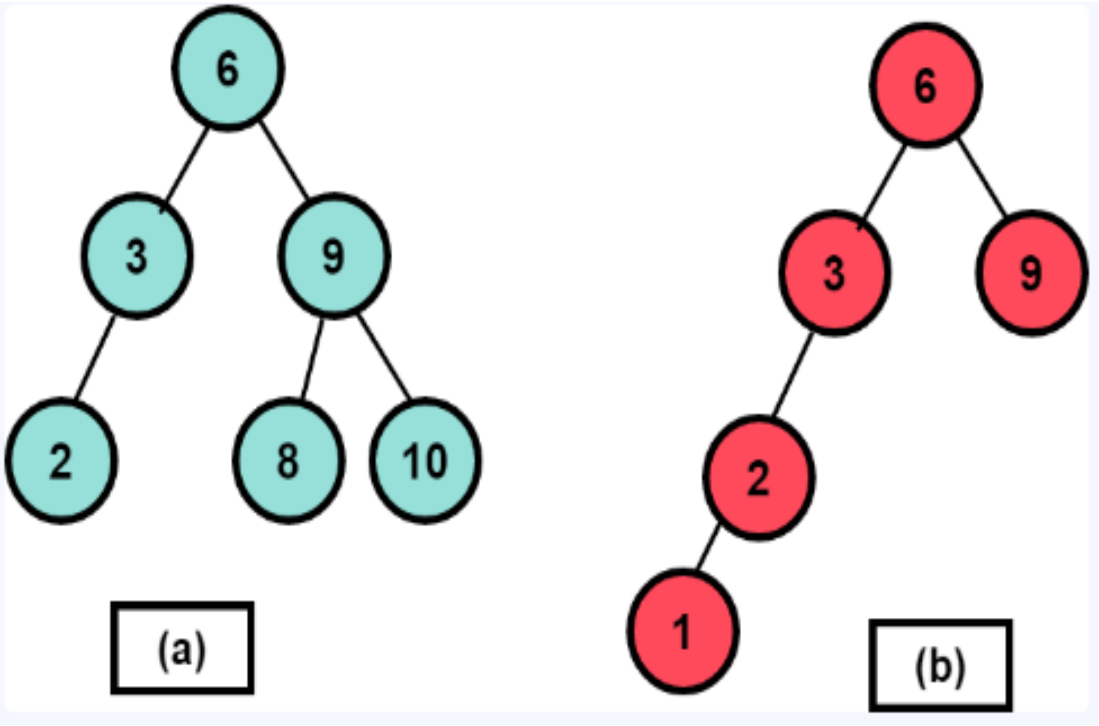
Balance Factor of each node

Balance Factor = height(left subtree) - height(right subtree)

Properties of Balance Factor

- If a subtree has a **balance factor>0**, then the subtree is called **left-heavy** as the height of the left subtree is greater than the height of its right subtree, so the left subtree has more nodes than the right subtree.
- If a subtree has a **balance factor<0**, then the subtree is called **right-heavy** as the height of the left subtree is smaller than the height of its right subtree, so the right subtree has more nodes than the left subtree.
- If the **balance factor=0,** the subtree is perfectly balanced, with both left and right subtrees having equal heights.

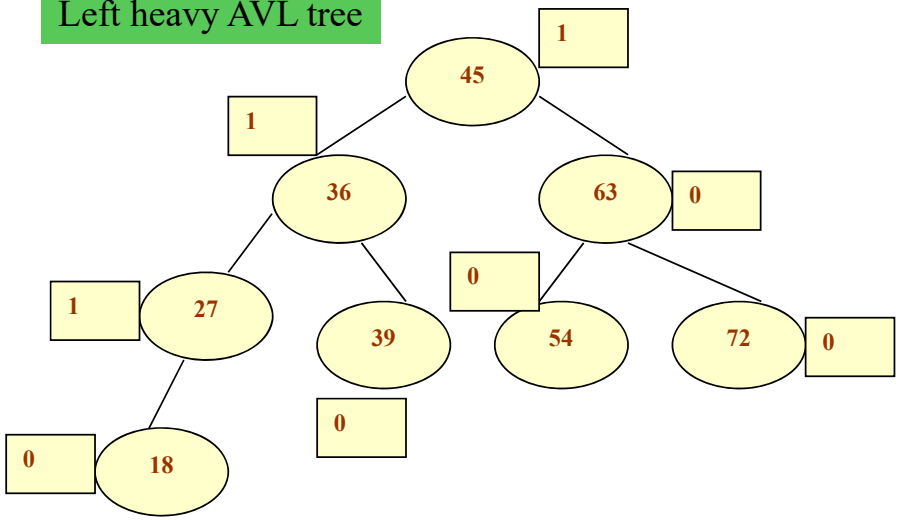
AVL Tree



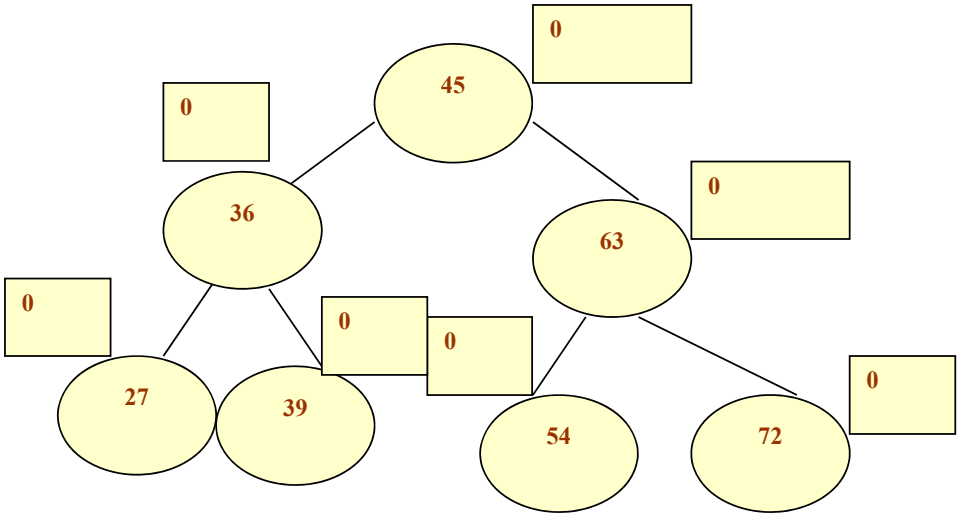
- The tree in figure (a) is an AVL tree. As you can see, for every node, the height difference between its left subtree and right subtree does not exceed mod(1).
- While in figure (b), if you check the node with value 3, its left subtree has height=2 while the right subtree's height=0. So, the difference is mod(2-0) = 2.
- Hence, the AVL property is not satisfied, and it is not an AVL tree.

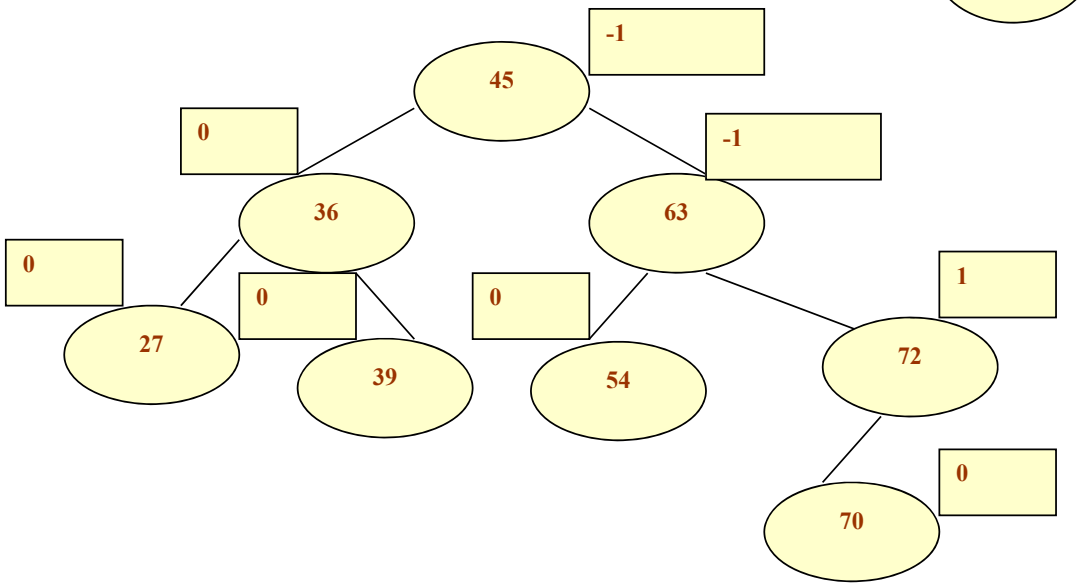
Balance Factor in AVL Trees

Left heavy AVL tree



Balanced AVL tree



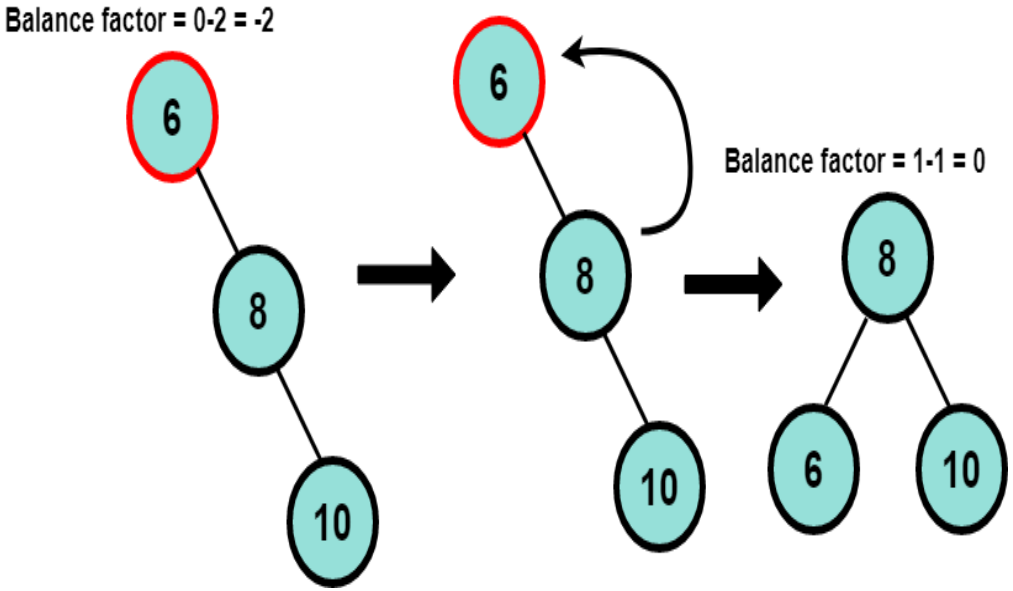


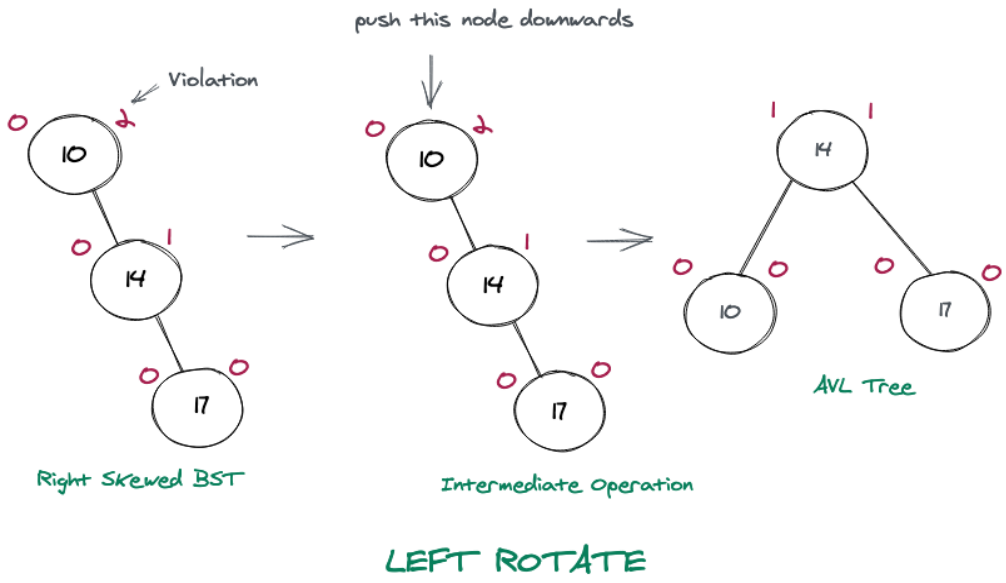
Right heavy AVL tree

AVL Trees Rotations

Left Rotation (LL - Single Rotation)

- Left rotation is performed when the imbalance is created due to the insertion of a new node in the right subtree of a right subtree or when a subtree becomes right heavy.





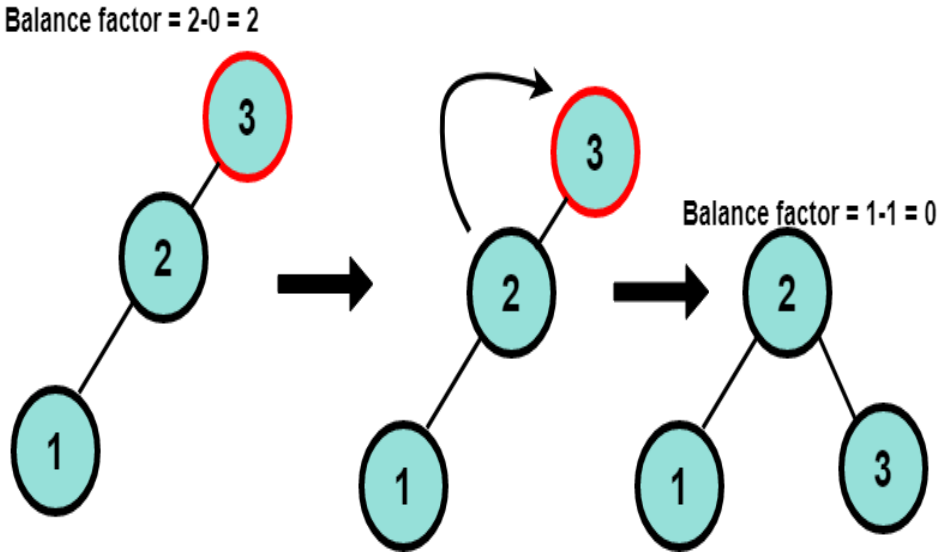
Node 6 has a balance factor of -2 after the insertion of node 10. After a single rotation towards the left, node 8 becomes the root and node 6 becomes its left child.

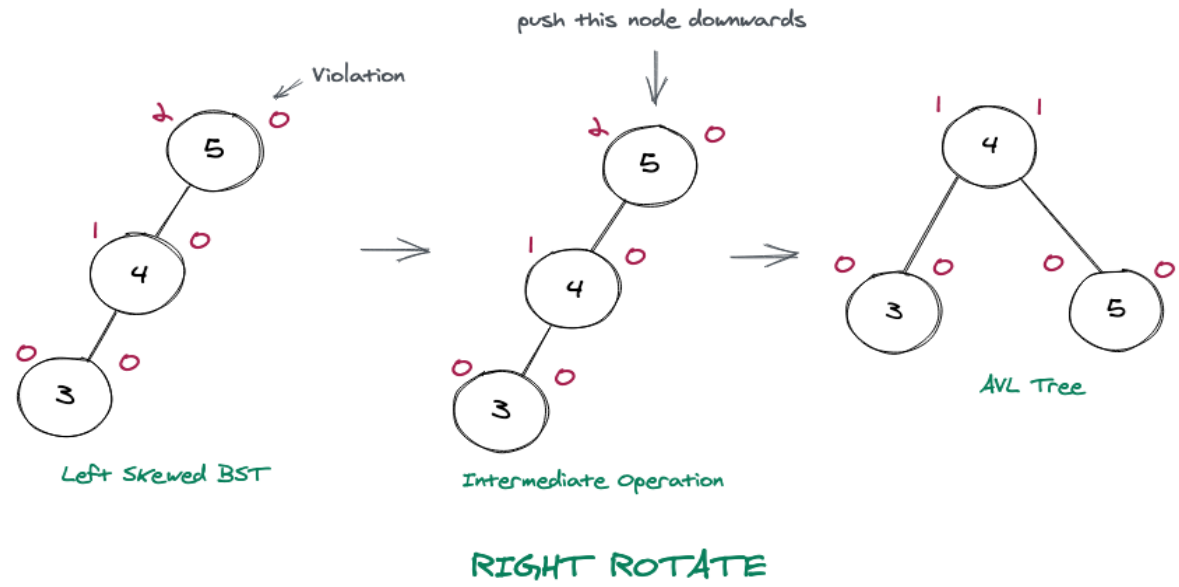
By doing this, the BST property also remains intact, and the tree becomes height-balanced.

AVL Trees Rotations

Right Rotation (RR - Single Rotation)

Right rotation is needed when the imbalance is created due to the insertion of a node in the left subtree of a left subtree or when a subtree becomes left heavy.





Node 3 has a balance factor of 2, so it is unbalanced. After a single rotation towards the right, node 2 becomes the root and node 3 becomes its right child.

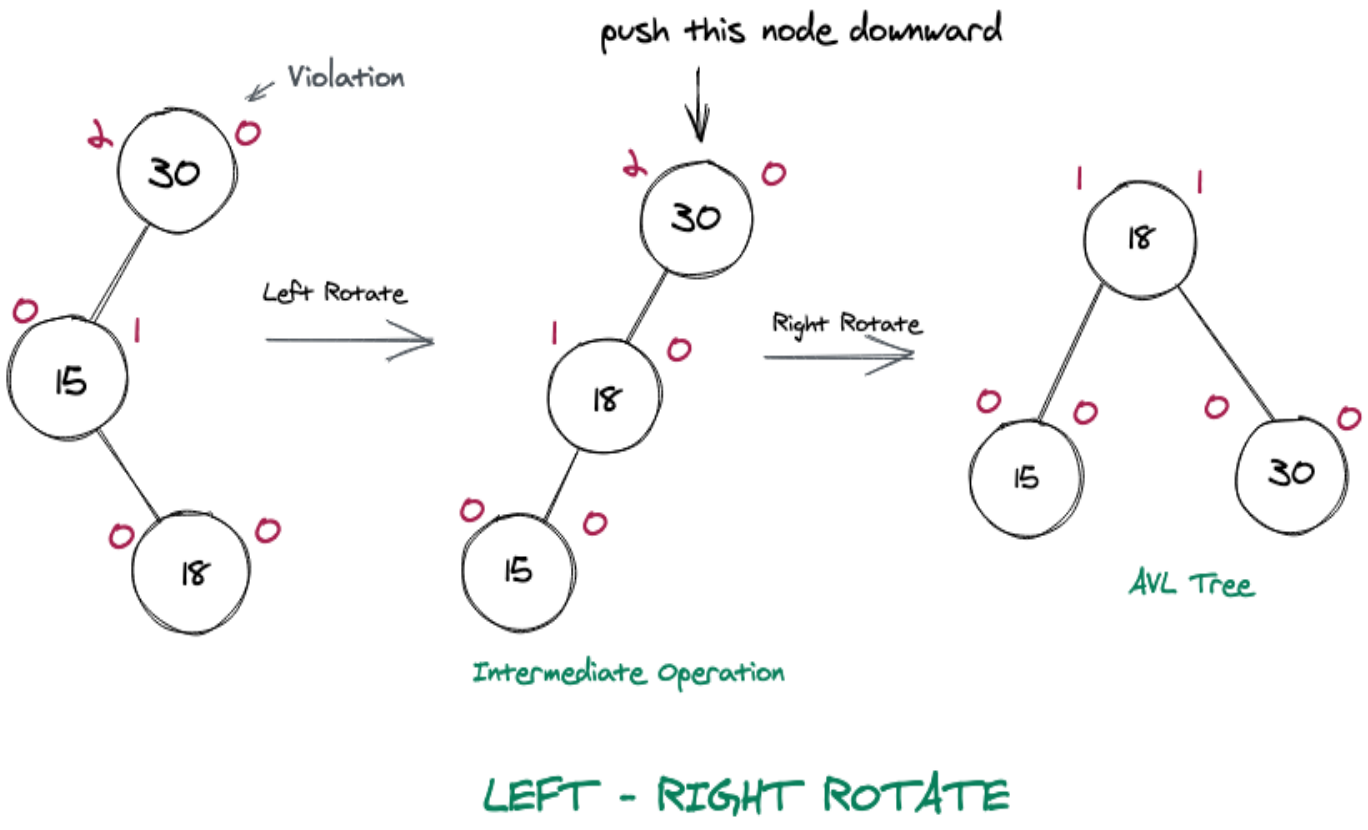
And the new root, i.e., node 2, has a balance factor of 0. Hence, the tree becomes balanced.

AVL Trees Rotations

Left-Right Rotation (LR - Double Rotation)

It is a double rotation in which a right rotation follows a left rotation.

It is needed when a new node is inserted in the left subtree of the right subtree of a node.



Node 1 is inserted as the left child of node 2, which is the right child of node 5.

See that the balance factor of node 5 becomes 2 after insertion of node 1.

Firstly, we do a left rotation, but the tree is still not balanced. If you see carefully, the tree is left heavy after this step, so we need a right rotation.

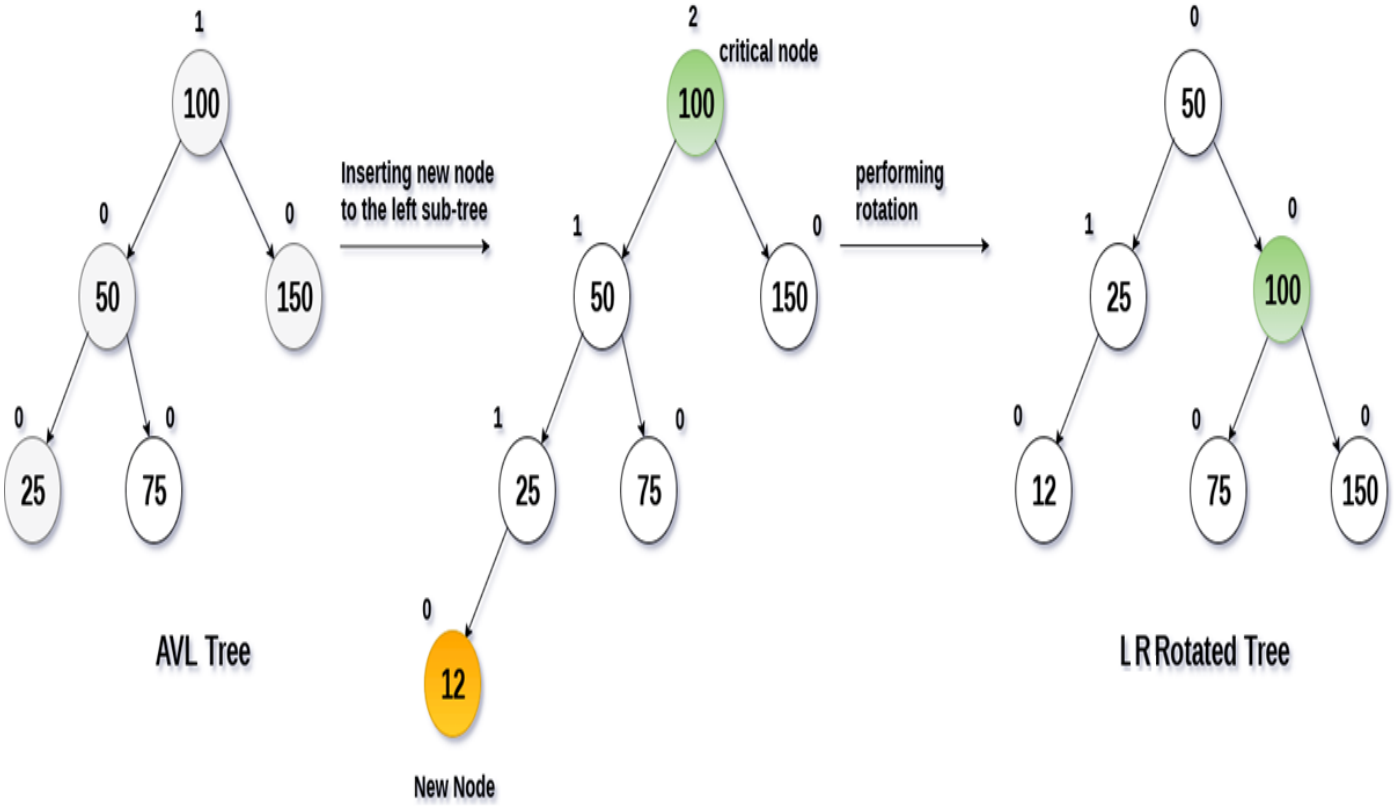
After which node 1, i.e. the newly inserted node, becomes the root.

AVL Trees Rotations

Left-Right Rotation (LR - Double Rotation)

It is a double rotation in which a right rotation takes place.

It is needed when a new node is inserted in the left subtree of the left subtree of a node.



Node 12 inserted to the left of 25 and therefore, it disturbs the Balance Factor of the tree.

The tree needs to be rebalanced by rotating it through LR rotation.

Here, the critical node 100 will be moved to its right, and the root of its left sub-tree.

The right sub-tree of with root node 75 will be place to the left of Node with value 100.

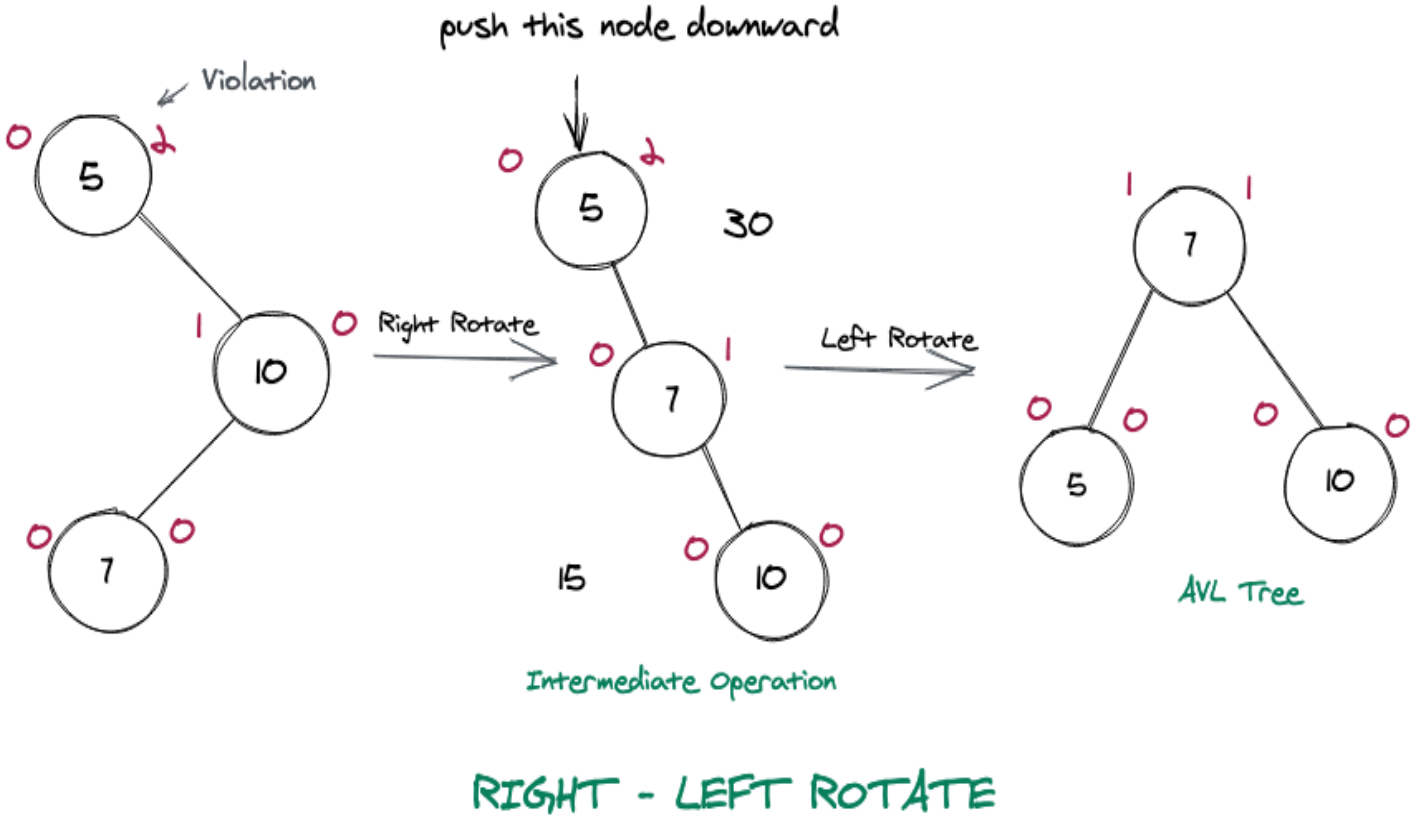
Recall term predecessor and successor.

AVL Trees Rotations

Right-Left Rotation (RL - Double Rotation)

In this, a right rotation is followed by a left rotation.

It is needed when a node is inserted in the left subtree of the right subtree of a node.



The insertion of node 6 creates an imbalance.

Firstly, we perform a right rotation which makes node 6 a parent of node 8 and node 8 becomes the right child of node 6.

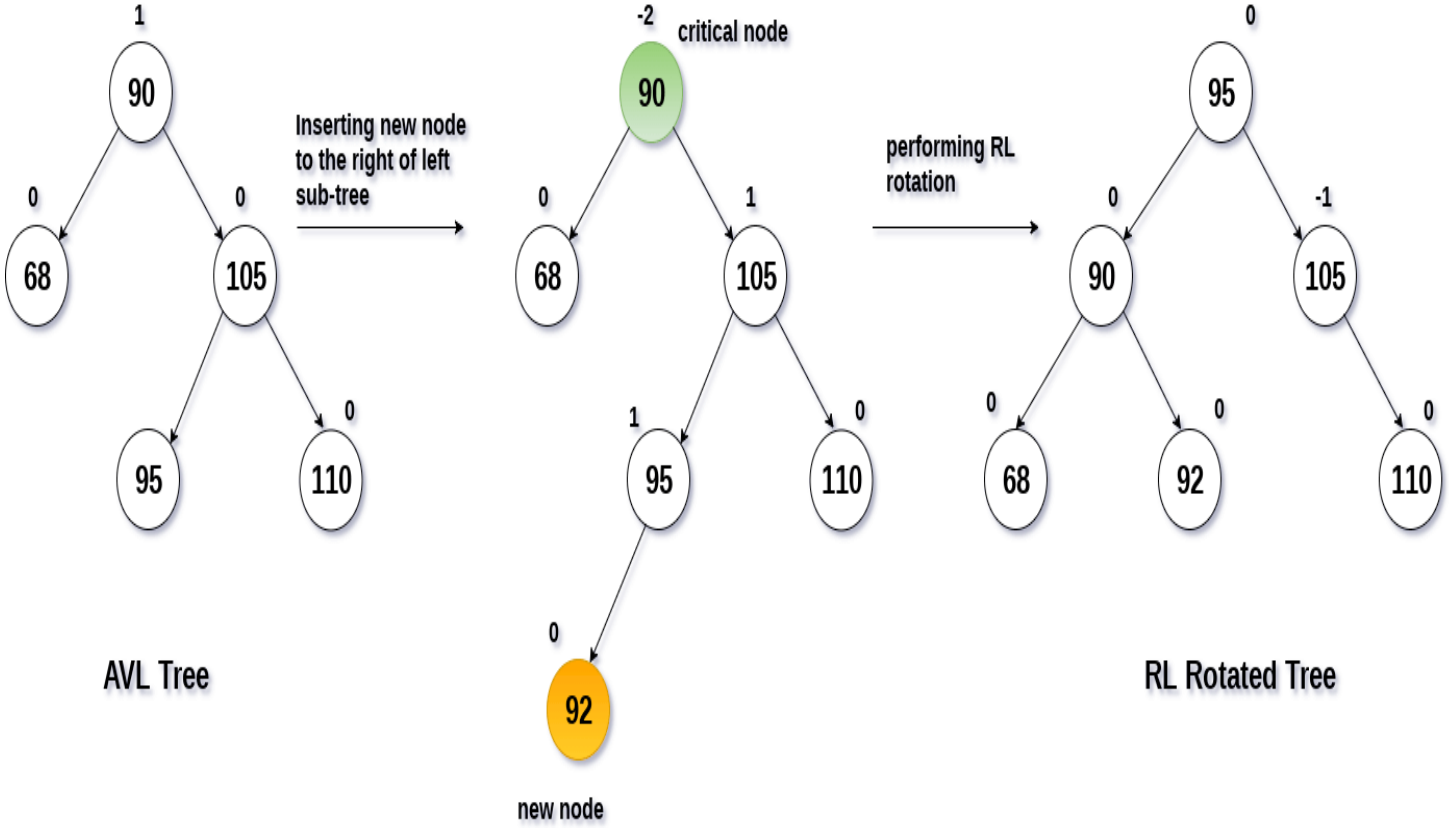
Next, we perform left rotation on the resulting right-heavy tree and finally, the tree becomes balanced.

AVL Trees Rotations

Right-Left Rotation (RL - Double Rotation)

In this, a right rotation is followed by a left rotation.

It is needed when a node is inserted in the left subtree of the right subtree of a node.

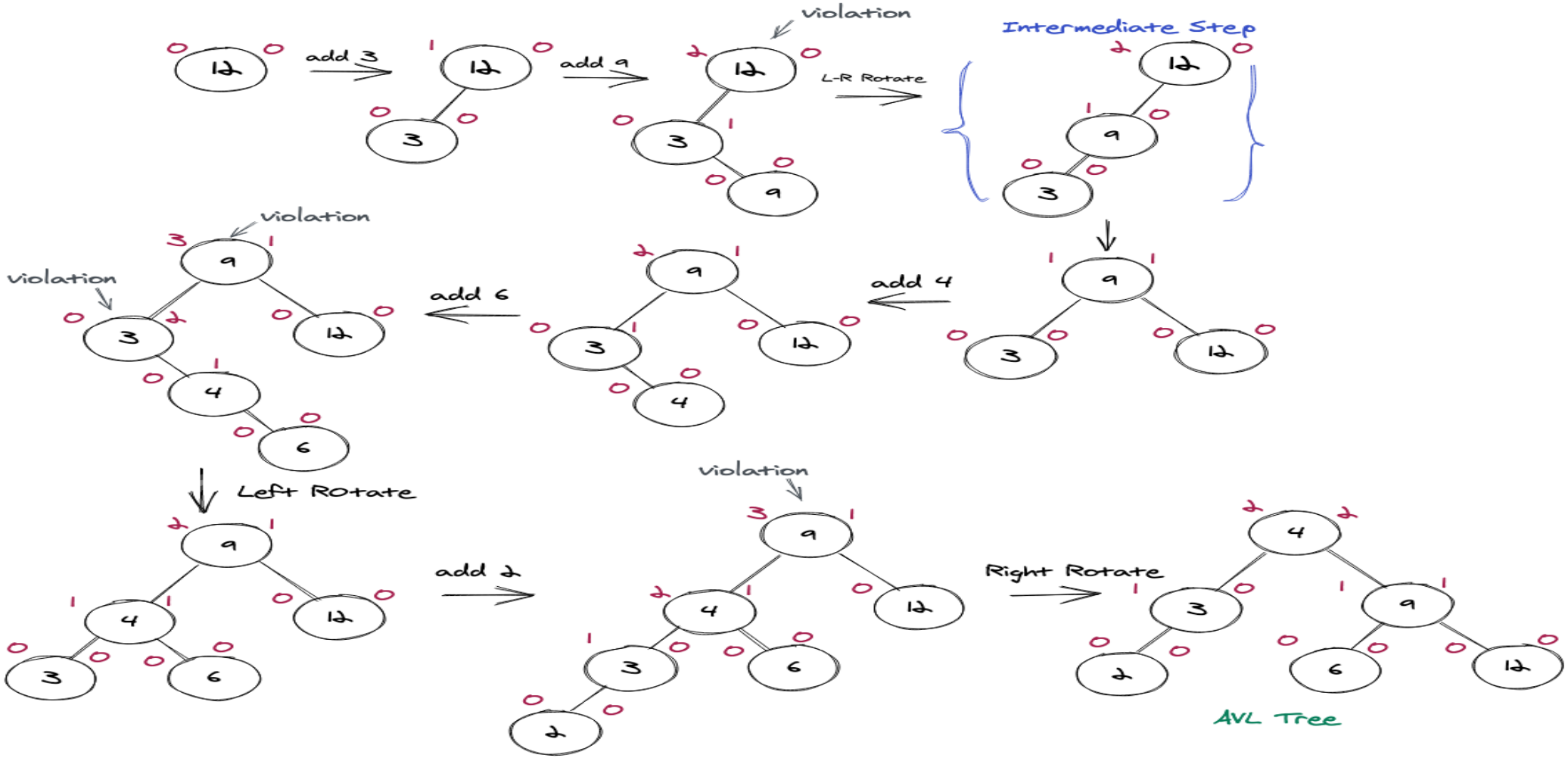


Inserting 92 disturbs the balance factor of the node 92 and it becomes the critical node with 105 as the node and 95.

In RL rotation, 95 becomes the root of the tree with node 90 and 105 as its left and right children respectively.

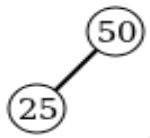
Recall term predecessor and successor.

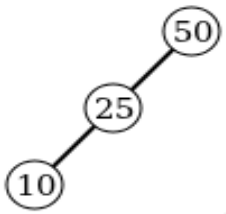
Insertion in AVL Trees

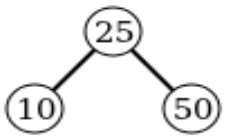


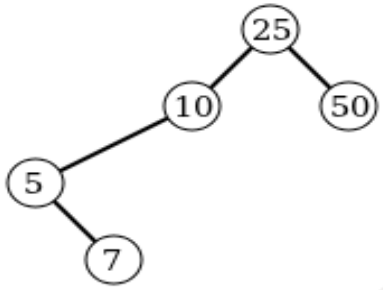
Extended Example of Insertion in AVL Trees

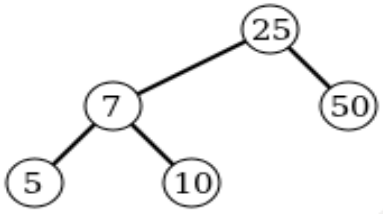
Insert 50, 25, 10, 5, 7, 3, 30, 20, 8, 15 into AVL tree

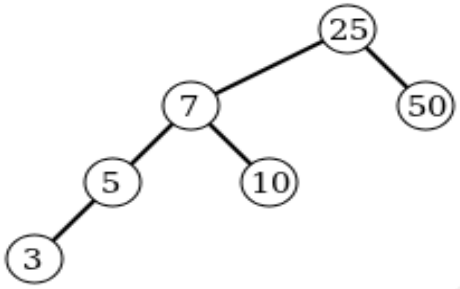


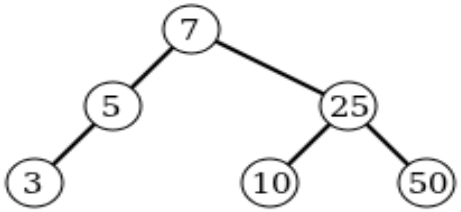


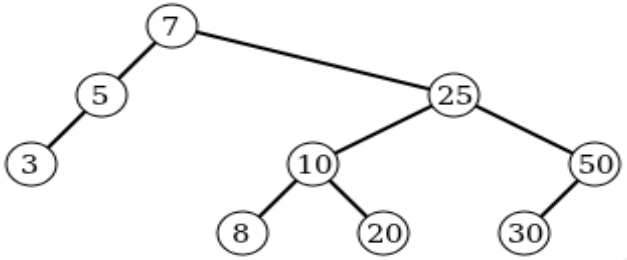


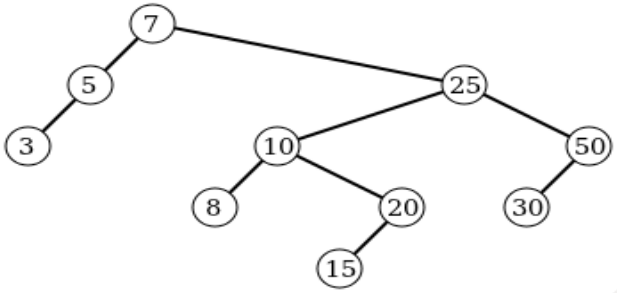


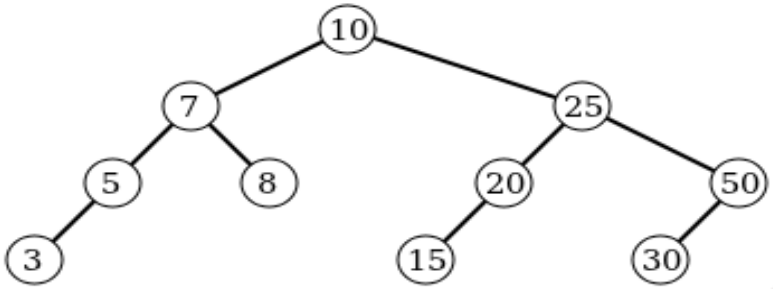












Critical Node is 7, so make in-order successor node as root and arrange node accordingly

Extended Example of Insertion in AVL Trees

Insert 3, 2, 1, 4, 5, 6, 7, 16, 15, 14

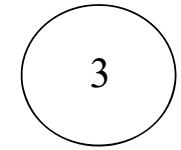


Fig 1

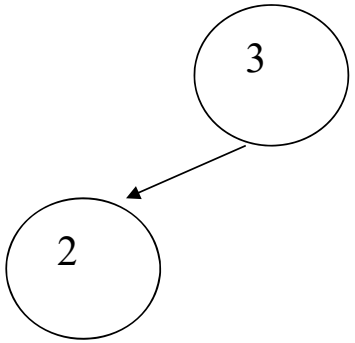


Fig 2

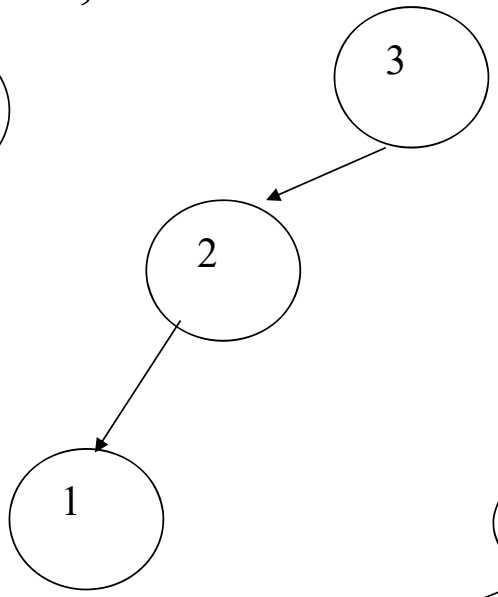


Fig 3

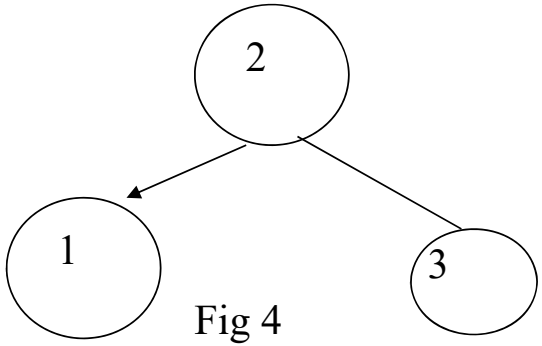


Fig 4

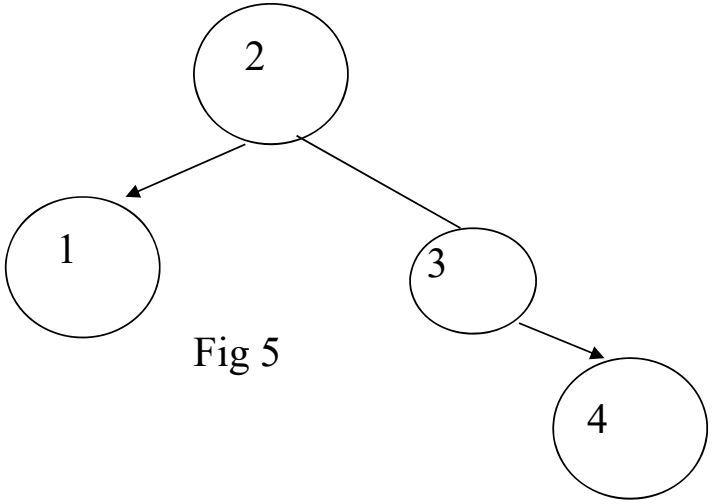


Fig 5

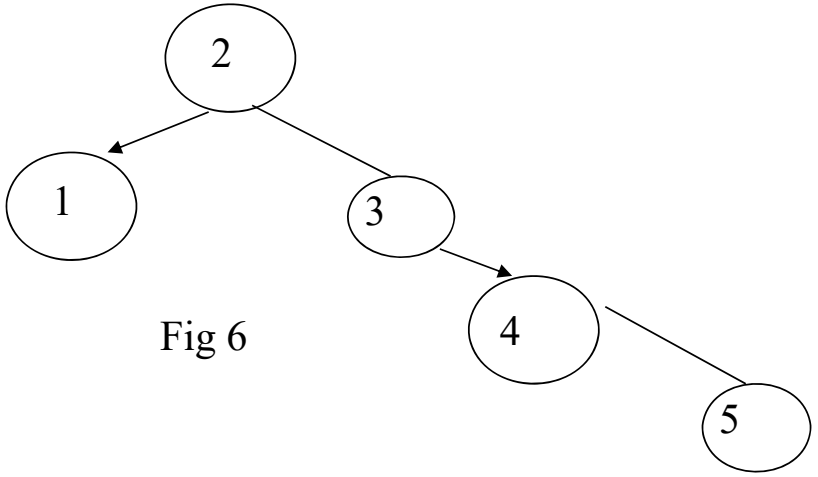


Fig 6

Insert 3, 2, 1, 4, 5, 6, 7, 16, 15, 14

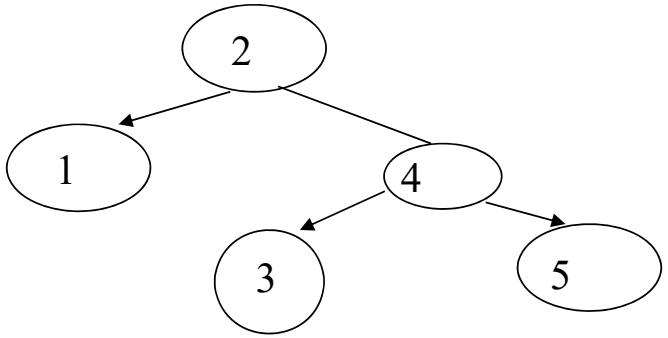


Fig 7

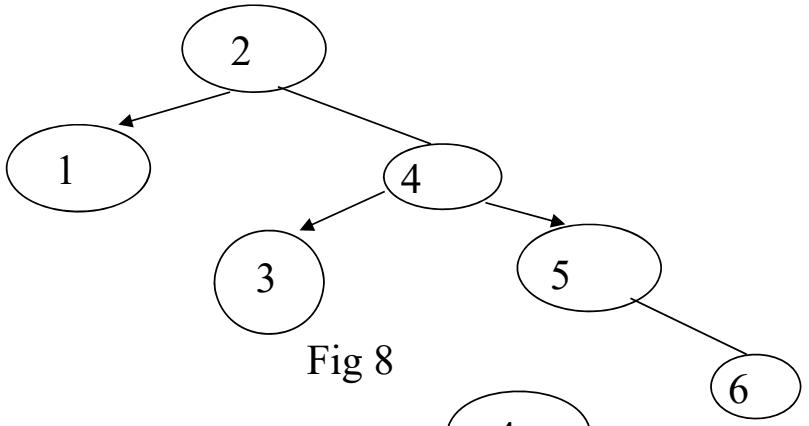


Fig 8

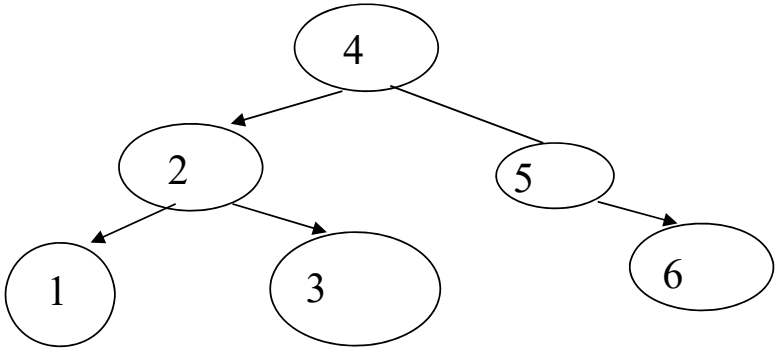


Fig 9

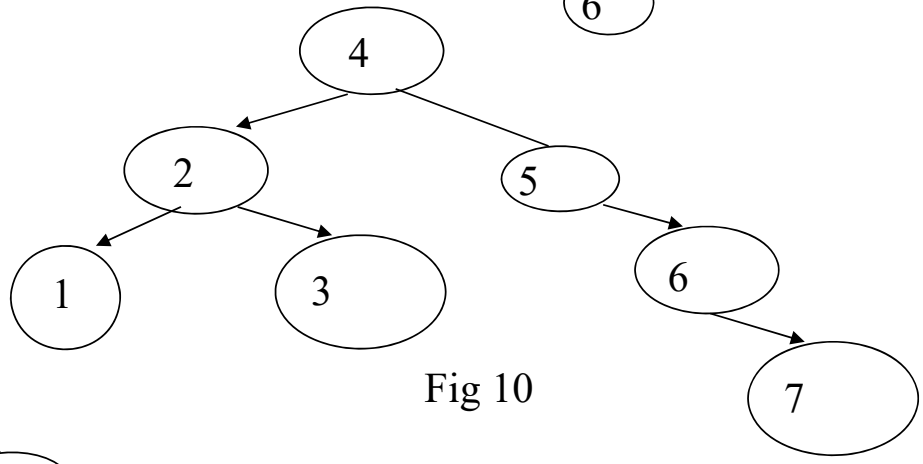


Fig 10

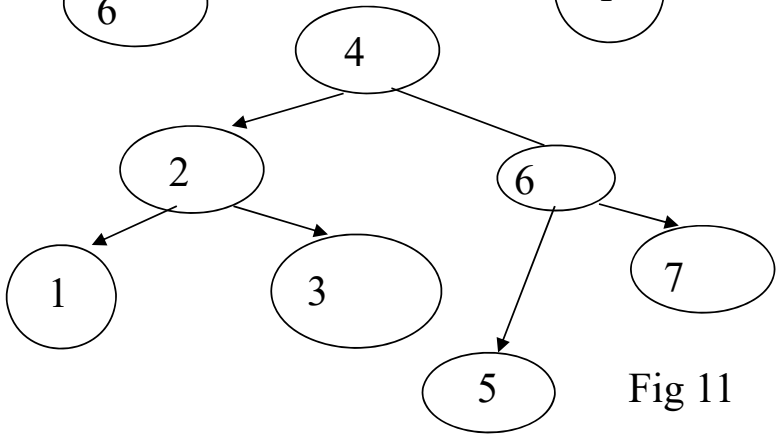


Fig 11

Insert 3, 2, 1, 4, 5, 6, 7, 16, 15, 14

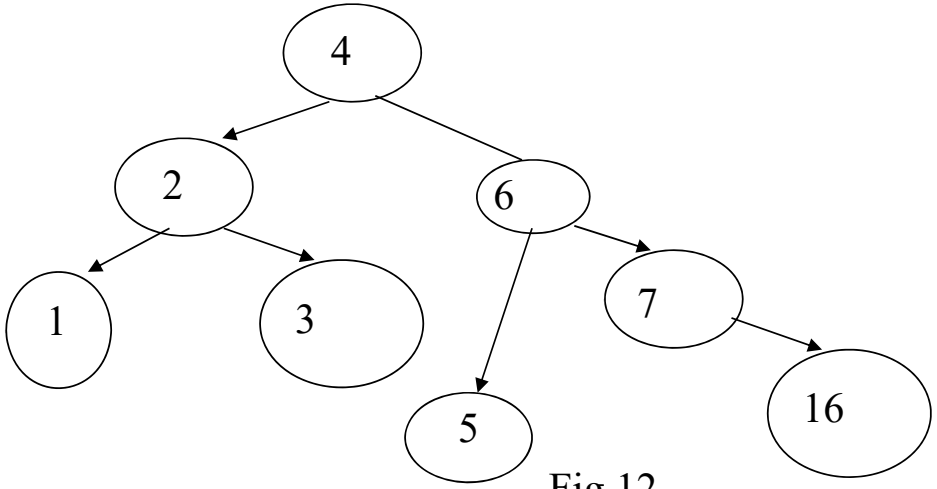


Fig 12

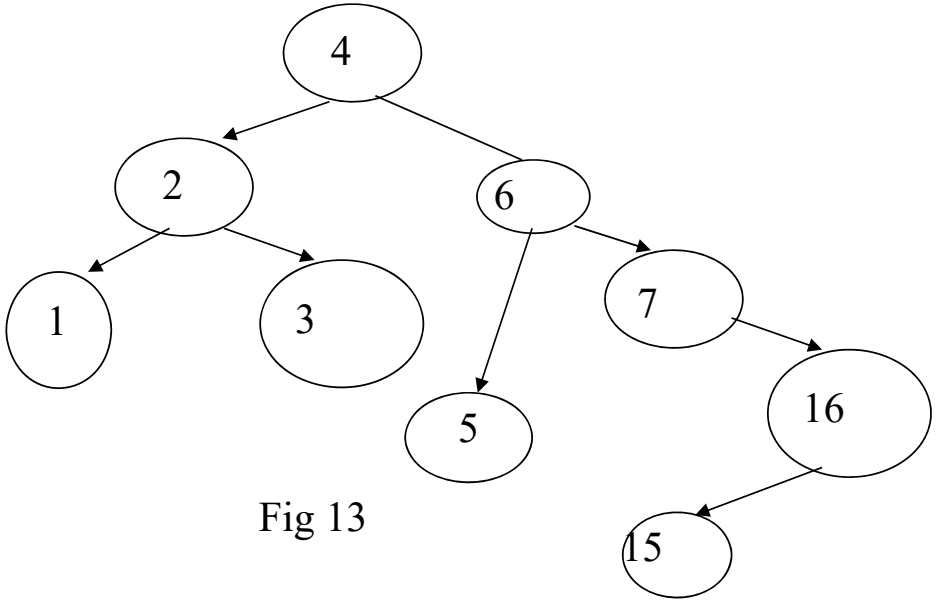


Fig 13

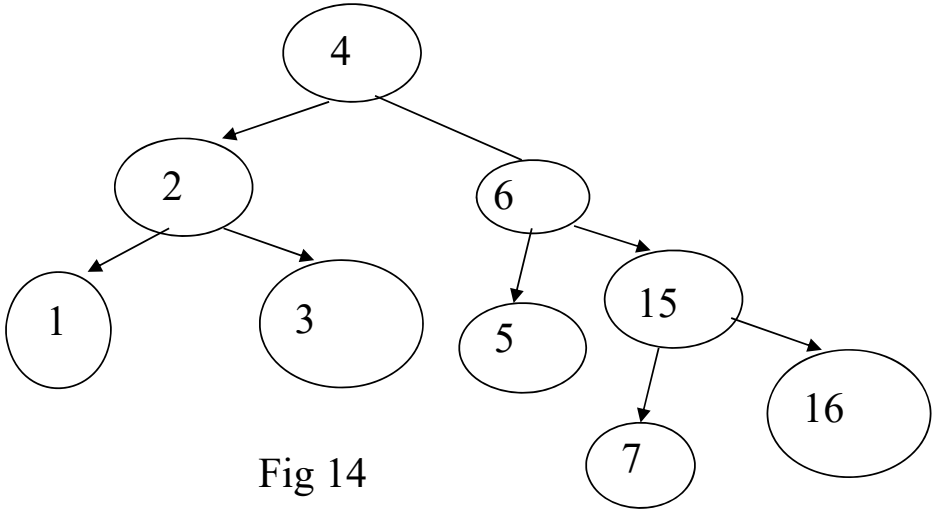


Fig 14

Insert 3, 2, 1, 4, 5, 6, 7, 16, 15, 14

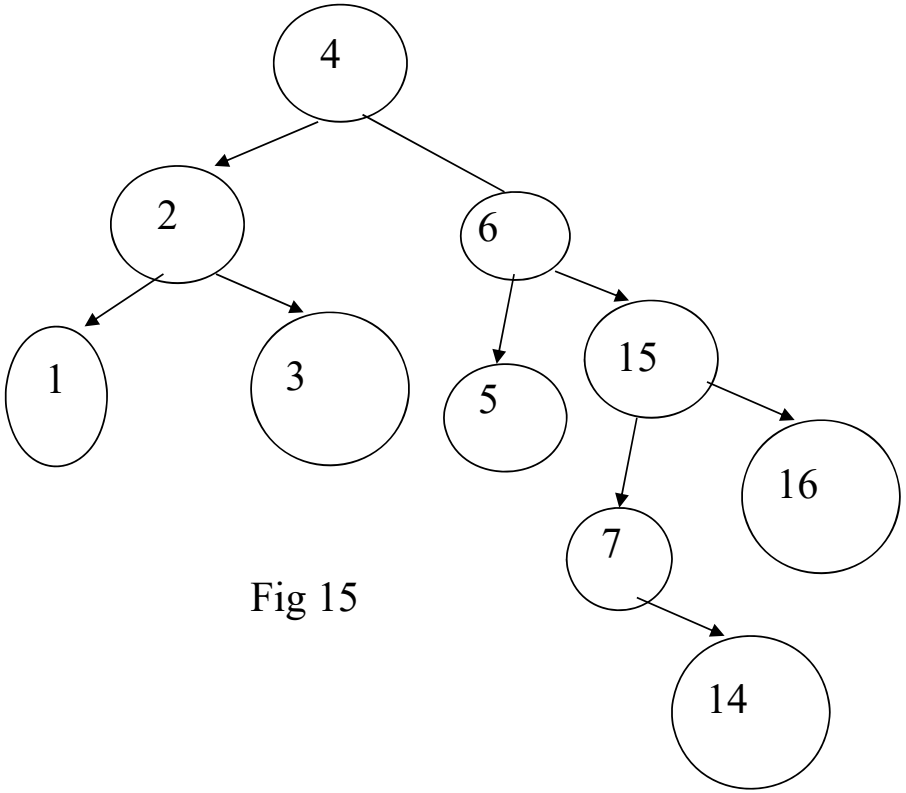


Fig 15

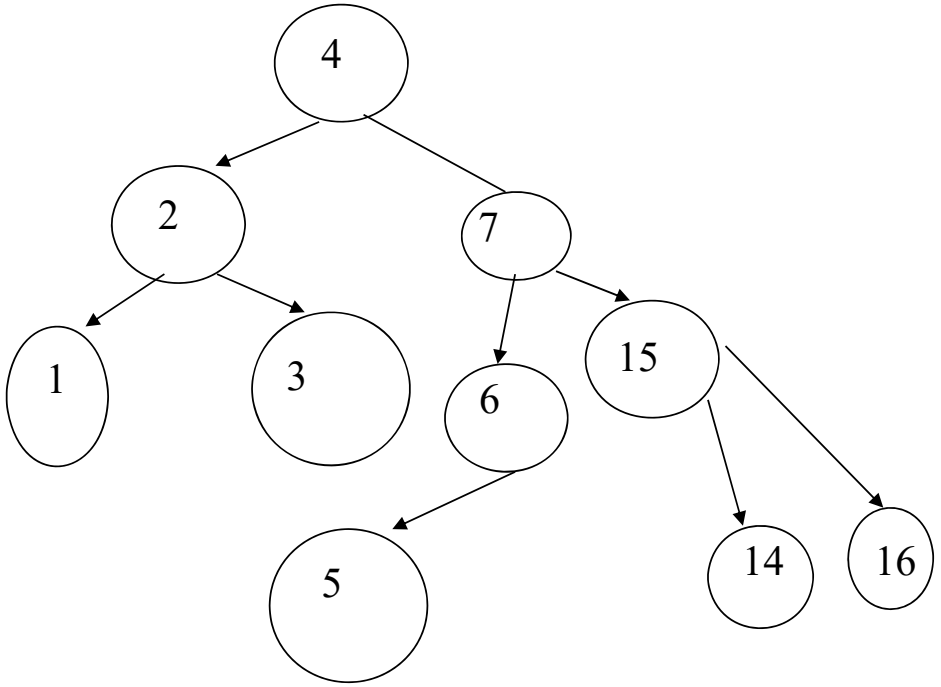
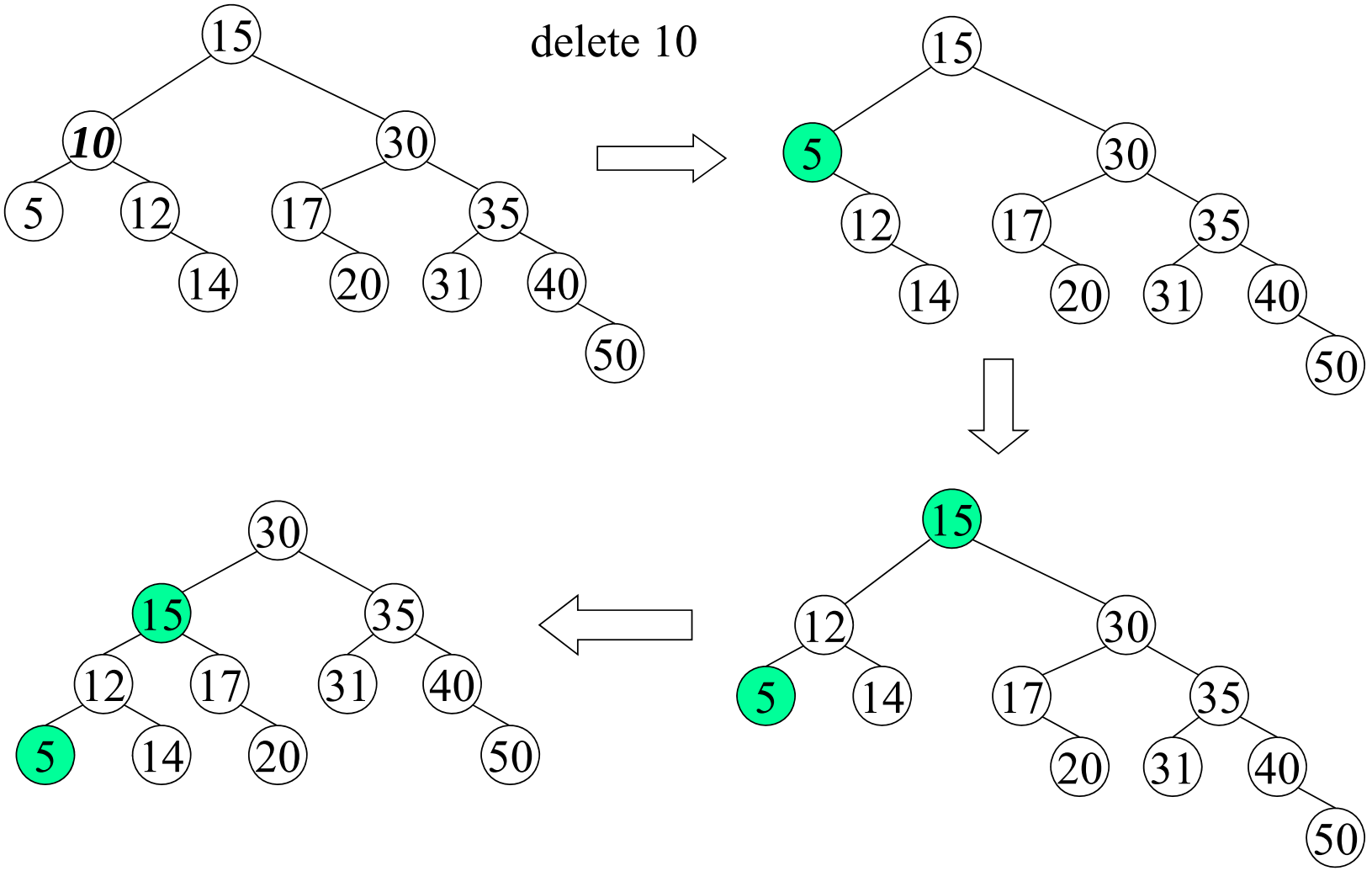


Fig 16

Deletions can be done with similar rotations

Deletion Example



Advantages of AVL Trees

- The AVL tree is always height-balanced, and its height is always O(log N), where N is the total number of nodes in the tree.
- The time complexities of all the operations are better than BSTs as the AVL tree has the worst-case time complexity of all operations as O(log N), while in BST, it is O(N).
- AVL trees have self-balancing properties.
- AVL trees are not skewed.
- AVL trees perform better searching operations than Red-Black Trees
- It shows better search time complexity.

Application of AVL Tree

- Its application is in in-memory and dictionaries to store data.
- Database systems frequently use AVL trees as indexes to allow quick data searching and recovery of data functions.
- AVL trees operate in networking routing algorithms.
- AVL trees encounter applications in interactive storytelling games.

Exercise in Inserting in a BST, B-Tree, B+ Tree, AVL Tree, Heap Sort, Draw TBT for BST

- Insert the following keys in B-tree when Order = 3, 4 and 5
  - 42, 22, 12, 82, 62, 52, 9, 32, 102
  - 46, 2, 83, 41, 102, 5, 17, 31, 64, 49, 18
  - 21, 16, 36, 51, 67, 75, 27, 32, 72, 14, 12
  - 90, 66, 57, 33, 87, 82, 13, 76, 14, 45, 38
  - 3, 7, 9, 23, 45, 1, 5, 14, 25, 24, 13, 11, 8, 19, 4, 31, 35, 56
  - 17, 9, 34, 56, 11, 4, 71, 86, 55, 10, 39, 49, 52, 82, 31, 13, 22, 35, 44, 20, 60, 28

Check your approach with a neighbour and discuss any differences.

DS Online Learning Resources & Simulators

- https://www.cs.usfca.edu/~galles/visualization/Algorithms.html
- https://yongdanielliang.github.io/animation/animation.html
- https://www.calcont.in/Conversion/infix_to_postfix
- https://csvistool.com/
- https://visualgo.net/en
- https://runestone.academy/ns/books/published/pythonds/index.html
- https://github.com/Lugriz/typescript-algorithms/tree/master
- https://lnogueir.github.io/expression-tree-gen/
- https://cse.iitkgp.ac.in/~rkumar/pds-vlab/index1.html
- https://cse01-iiith.vlabs.ac.in/List%20of%20experiments.html

Question?

- A good question deserve a good grade…

Dr. Amit Pimpalkar

+91-988-171-3450

pimpalkarap@rknec.edu