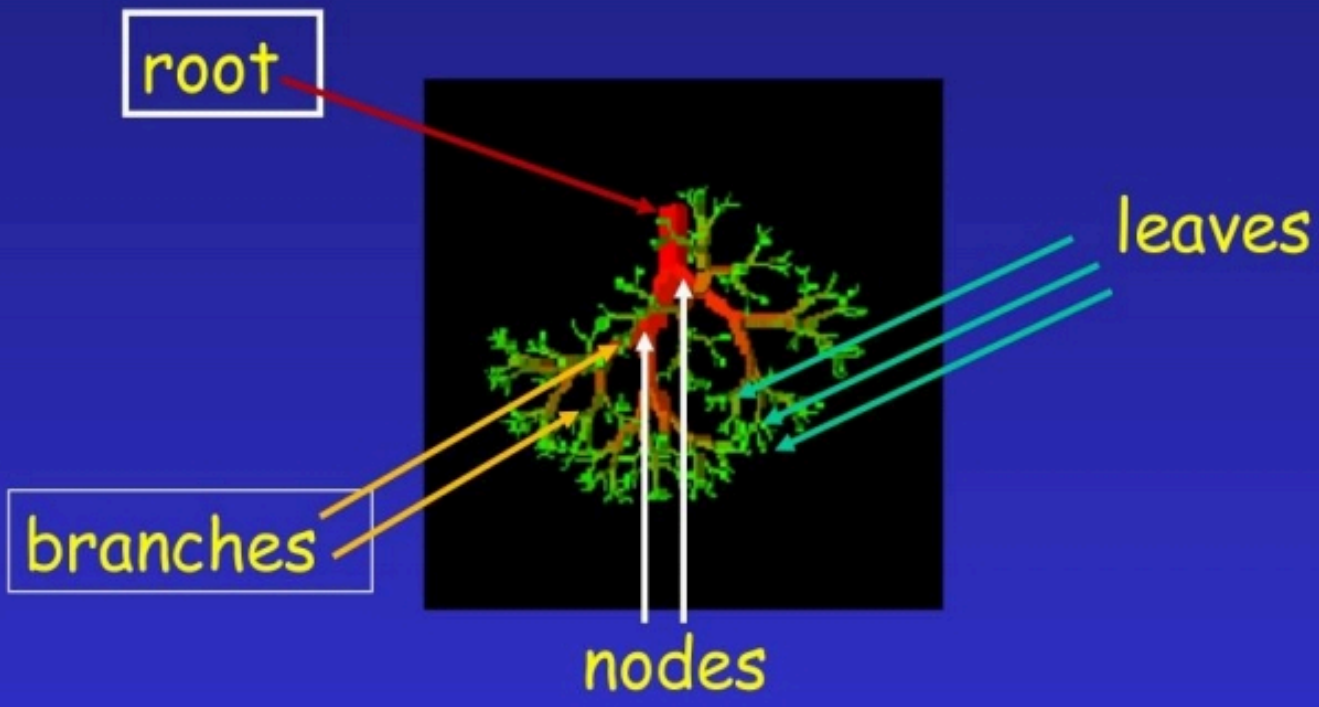
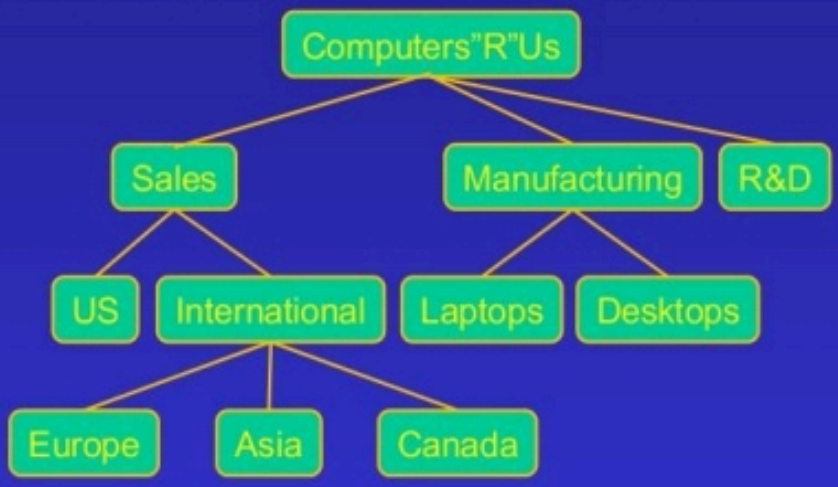
Before we can start some definition and properties of tree

Computer Scientist's View



Before we can start some definition and properties of tree

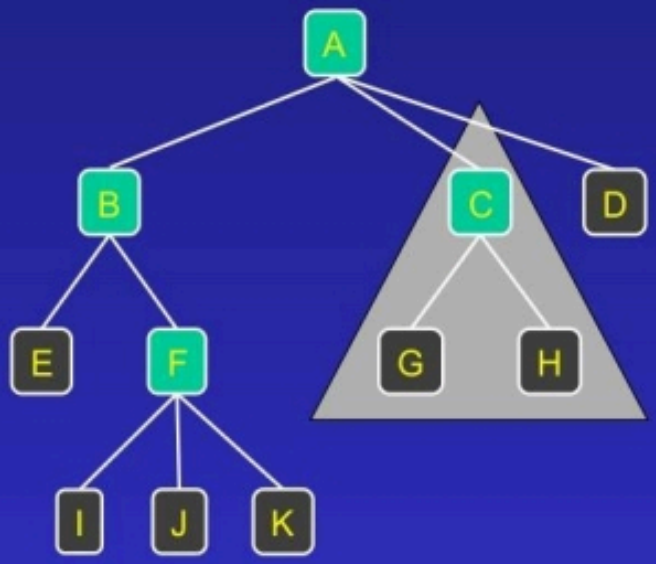
- **What is a Tree?**
- A tree is a finite nonempty set of elements.
- It is an abstract model of a hierarchical structure.
- Consists of nodes with a parent-child relation.



Tree Terminology

- **Root**: node without parent (A)
- **Siblings**: nodes share the same parent
- **Internal node**: node with at least one child (A, B, C, F)
- **External node (leaf)**: node without children (E, I, J, K, G, H, D)
- **Ancestors of a node**: parent, grandparent, grand-grandparent, etc.
- **Descendant** of a node: child, grandchild, grand-grandchild, etc.
- **Depth** of a node: number of ancestors
- **Height** of a tree: maximum depth of any node (3)
- **Degree** of a node: the number of its children
- **Degree** of a tree: the maximum number of its node.

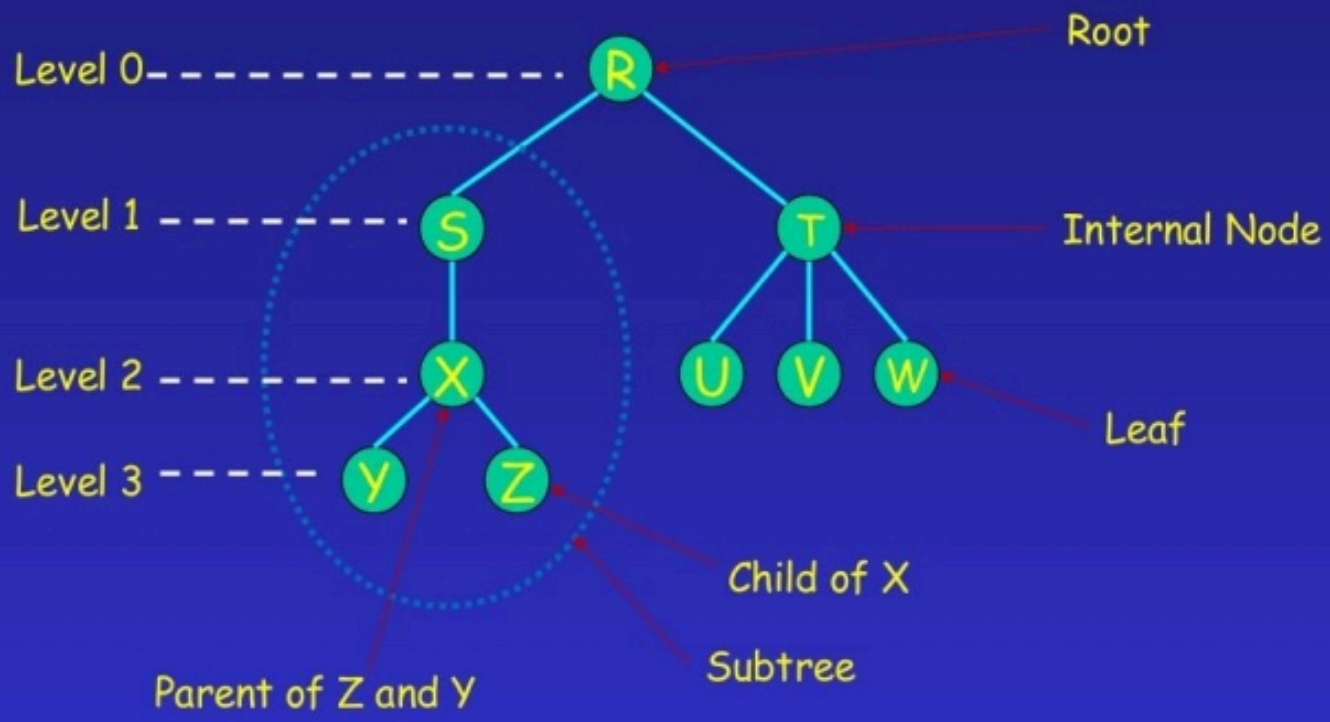
- **Subtree**: tree consisting of a node and its descendants



subtree

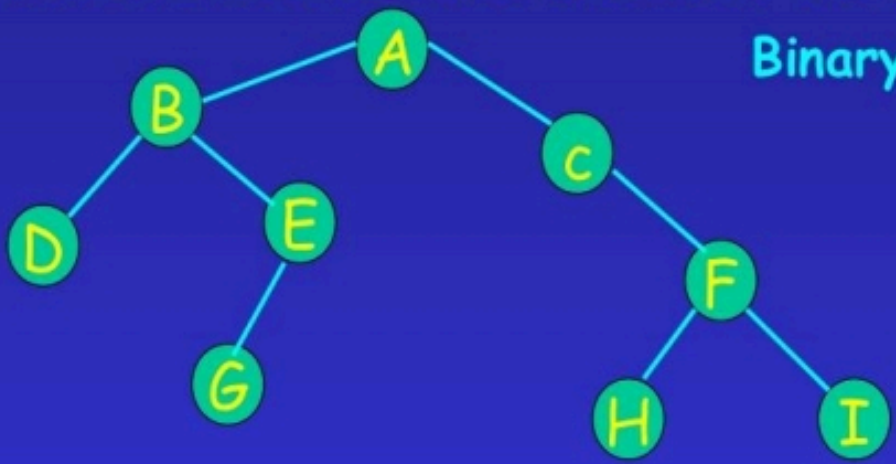
Tree Anatomy

- The children of a node are, themselves, trees, called subtrees.



Binary Tree

- A binary tree is a finite set of elements that is either empty or is partitioned into three disjoint subsets.
  - The first subset contains a single element called the root of the tree.
  - The other two subsets are themselves binary trees, called the left and right subtrees of the original tree. A left or right subtree can be empty.
- Each element of a tree is called a node of the tree.



Binary tree

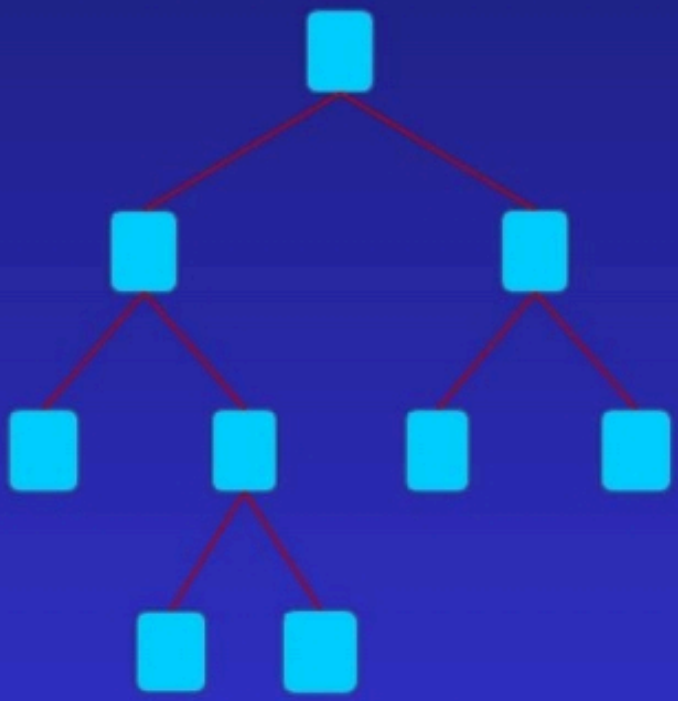
Left ≠ Right

The following two binary trees are *different:*

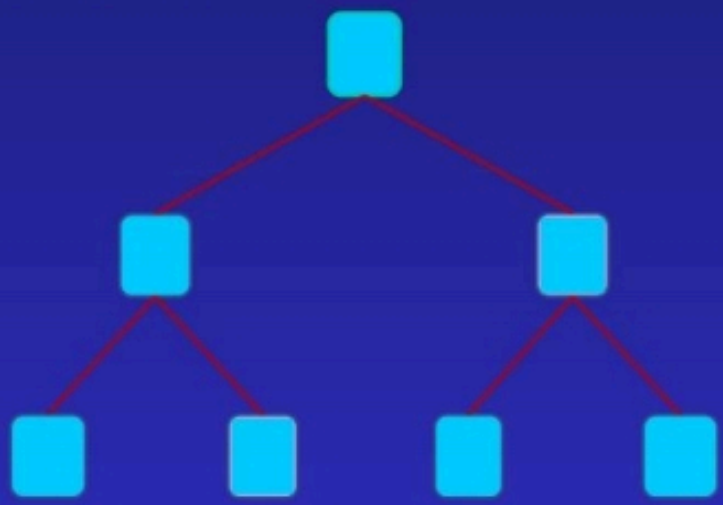
- In the first binary tree, node A has a left child but no right child; in the second, node A has a right child but no left child
- Put another way: Left and right are *not* relative terms

Perfection

- A binary tree is a **perfect binary tree** if every level is completely full



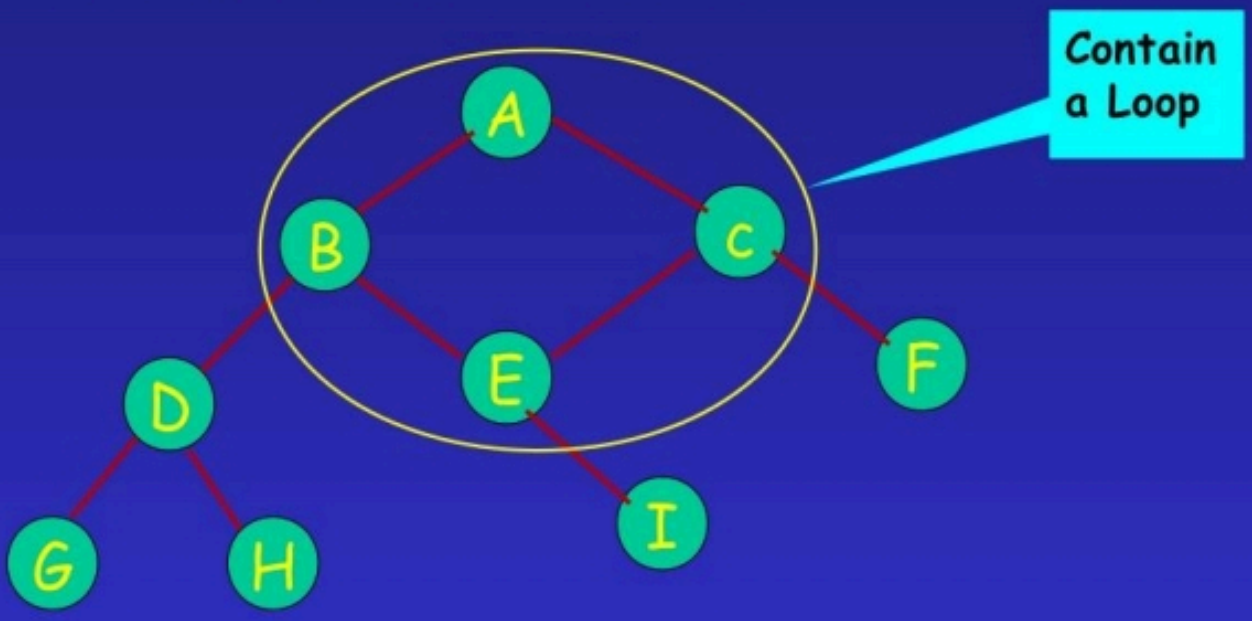
Not perfect



Perfect!

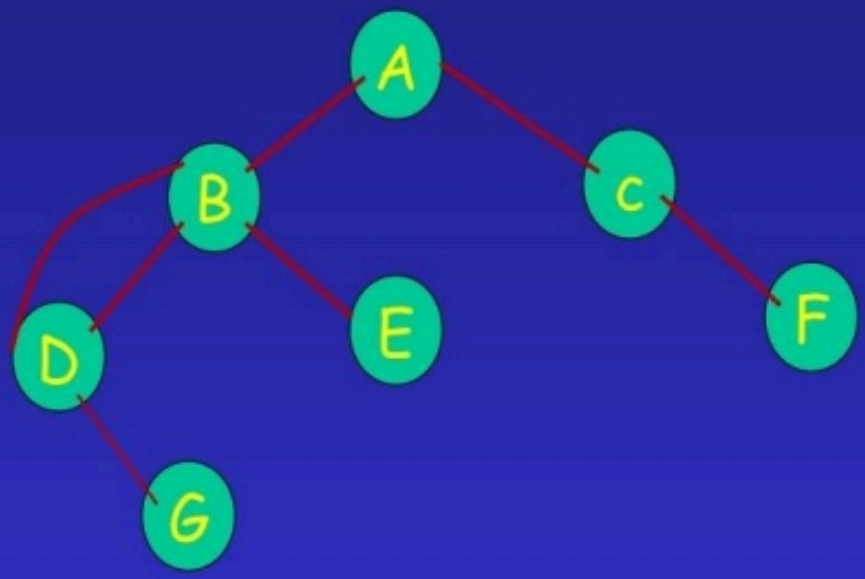
Binary Tree

Structures that are not binary trees



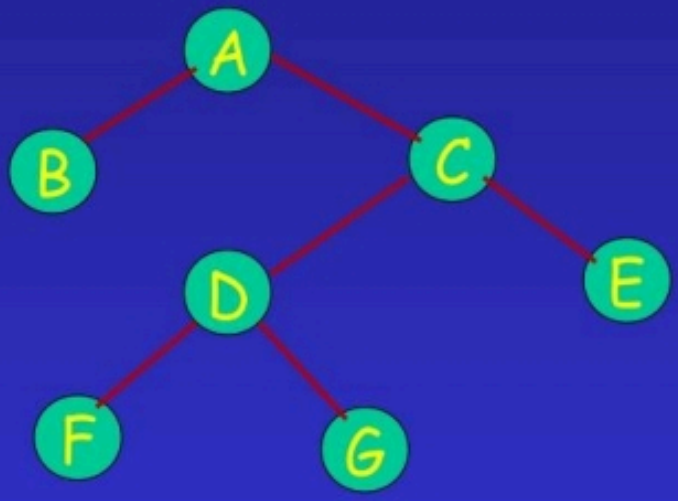
Binary Tree

Structures that are not binary trees



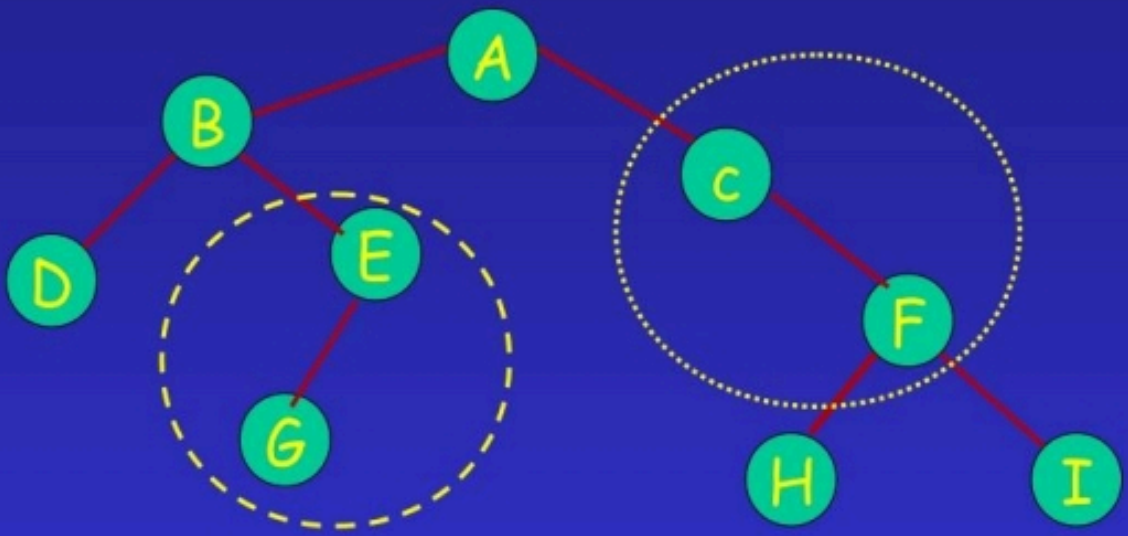
Strictly binary trees

- If every nonleaf node in a binary tree has nonempty left and
- right subtrees, the tree is called a strictly binary tree.
- A strictly binary tree with n leaves always contains 2n -1 nodes.



Strictly binary trees

- Structure that is not a strictly binary tree: because nodes C and E have one son each.



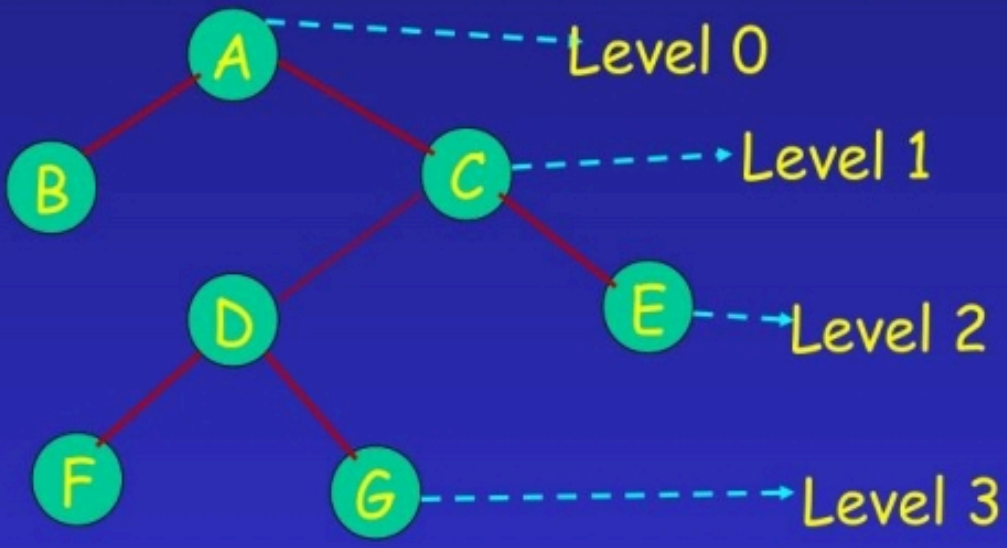
Level and depth of a binary Tree

Level of binary tree:

The root of the tree has level 0. And the level of any other node is one more than the level of its father.

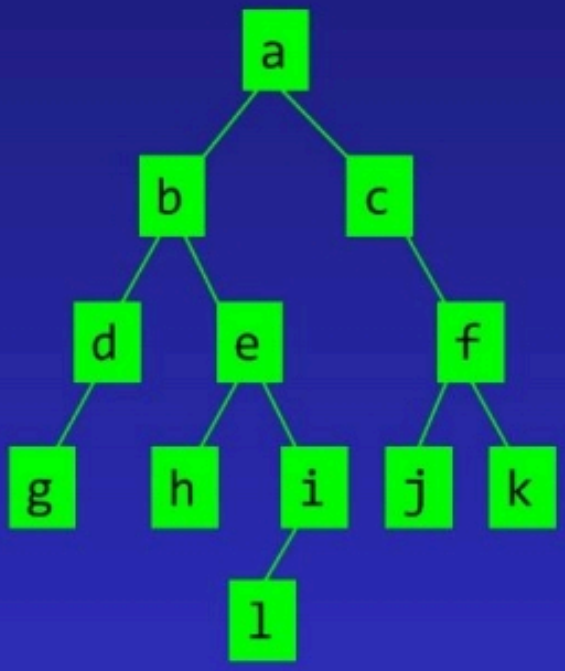
Depth of a binary tree:

The depth of a binary tree is the maximum level of any leaf in the tree.



Depth is 3.

Size and depth

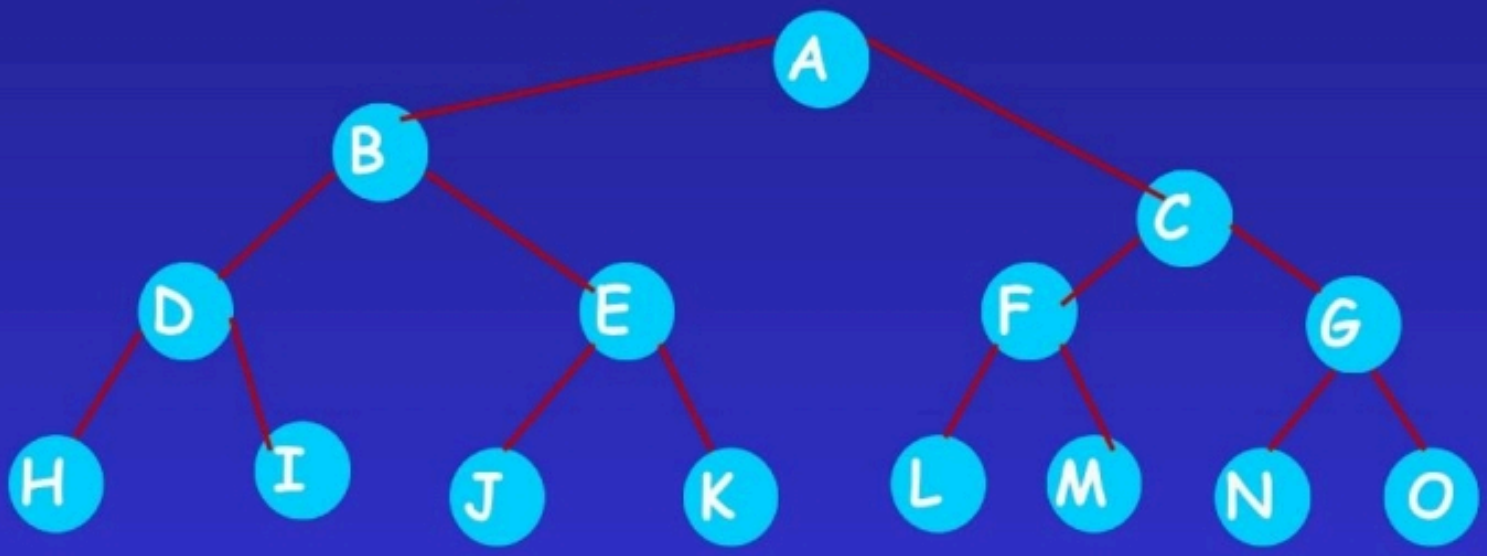


- The **size** of a binary tree is the number of nodes in it
  - This tree has size 12
- The **depth** of a node is its distance from the root
  - **a** is at depth zero
  - **e** is at depth 2
- The **depth** of a binary tree is the depth of its deepest node
  - This tree has depth 4

A complete binary tree

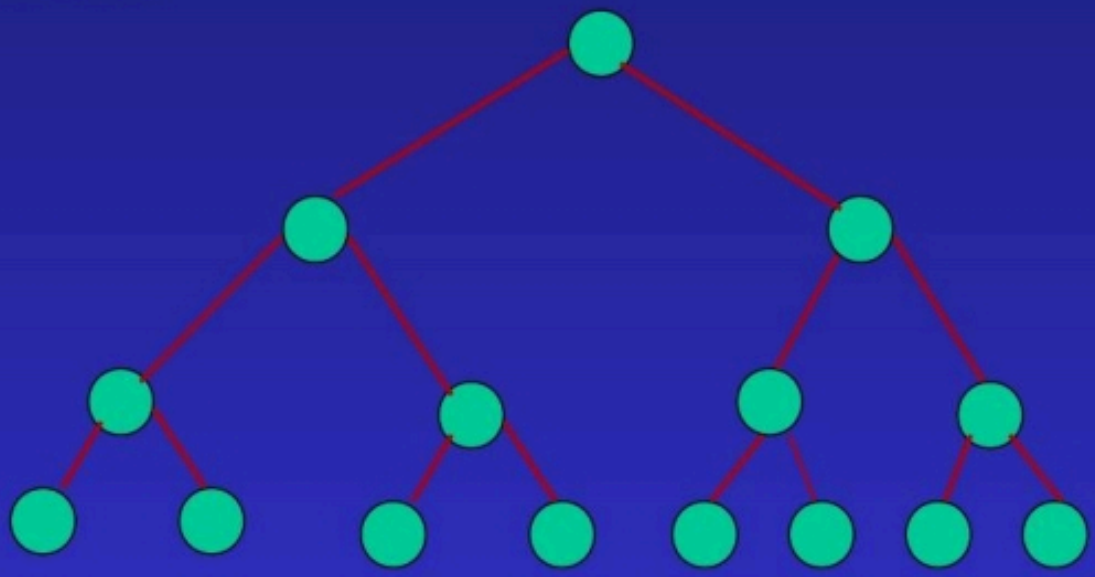
- Complete binary tree of depth d is the strictly binary tree all of whose leaves are at level d.
- A complete binary tree of depth d is the binary tree of depth d that contains exactly 2^{l} nodes at each level l between 0 and d.
- The total number of nodes = the sum of the number of nodes at each level between 0 and d.

$$= 2^{d+1} - 1$$



Full Binary Tree

A full binary tree of a given height **k** has $2^{k+1}-1$ nodes.



Height 3 full binary tree.

Tree Traversal

Preorder Traversal

- A traversal visits the nodes of a tree in a systematic manner
- In a preorder traversal, a node is visited before its descendants

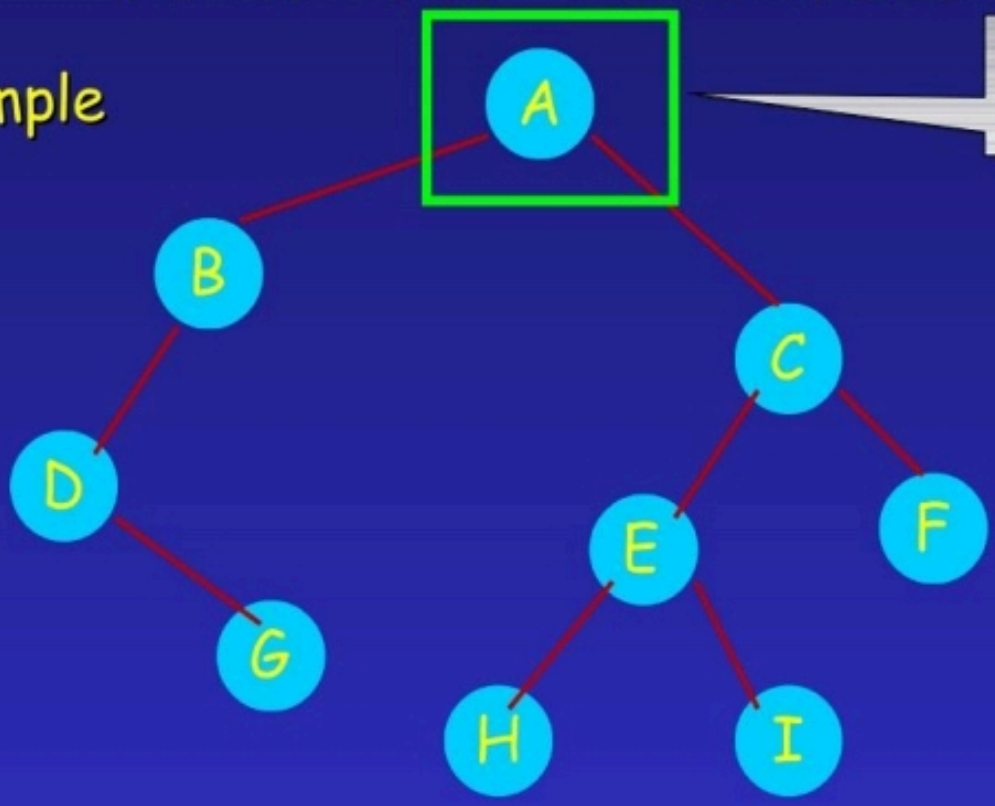
1. Visit the root
2. Traverse the left subtree in preorder
3. Traverse the right subtree in preorder

Preorder traversal

- node, left, right

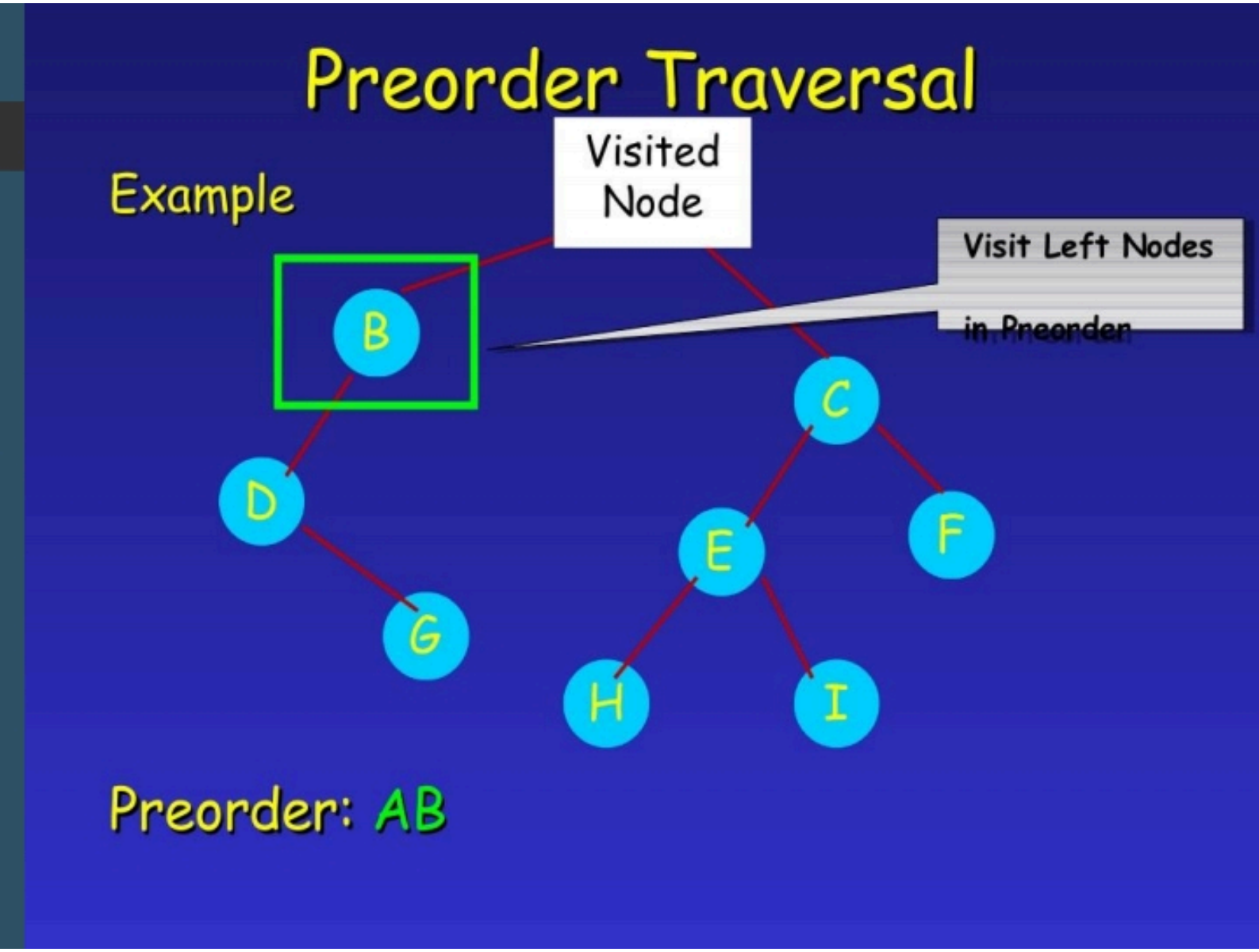
Preorder Traversal

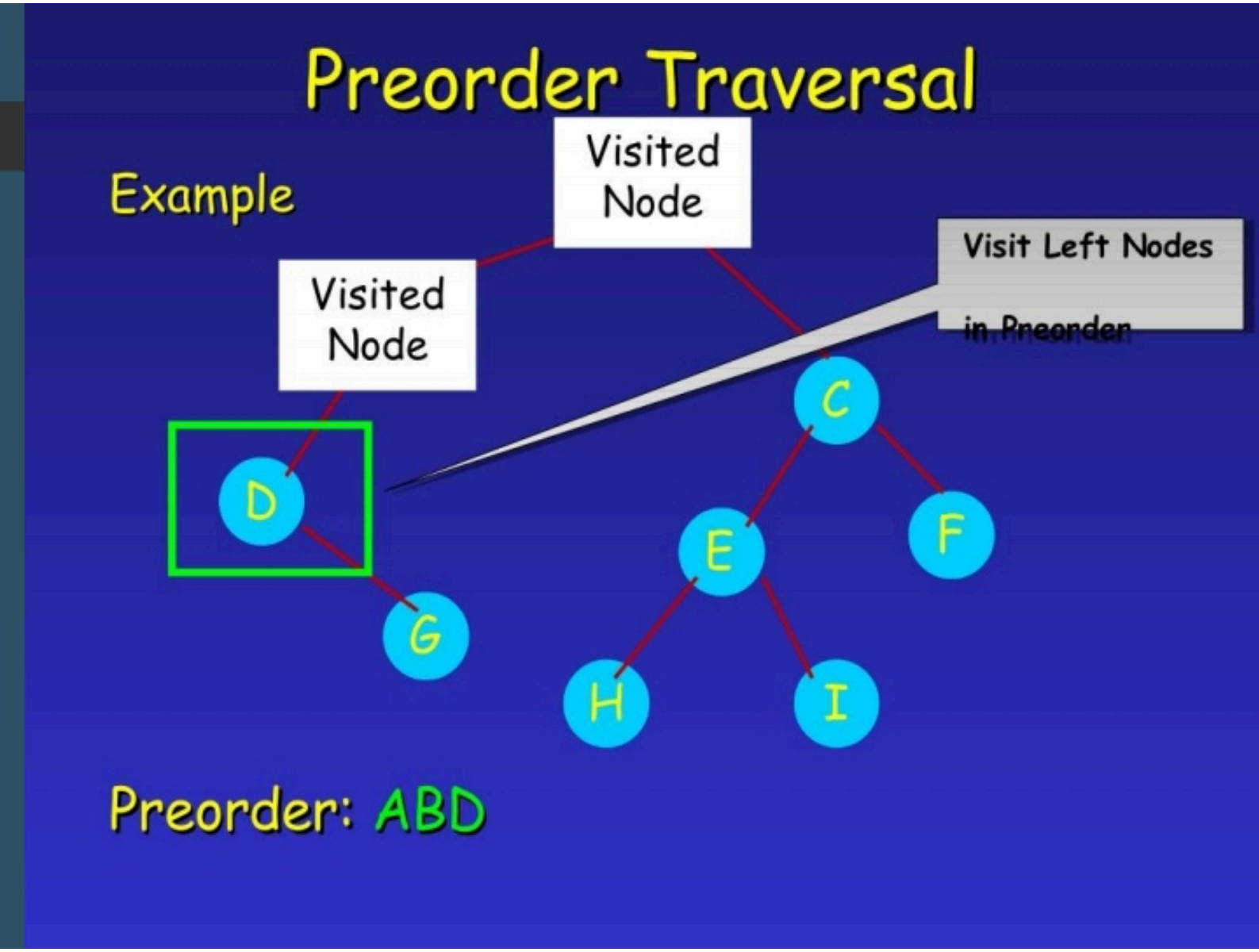
Example

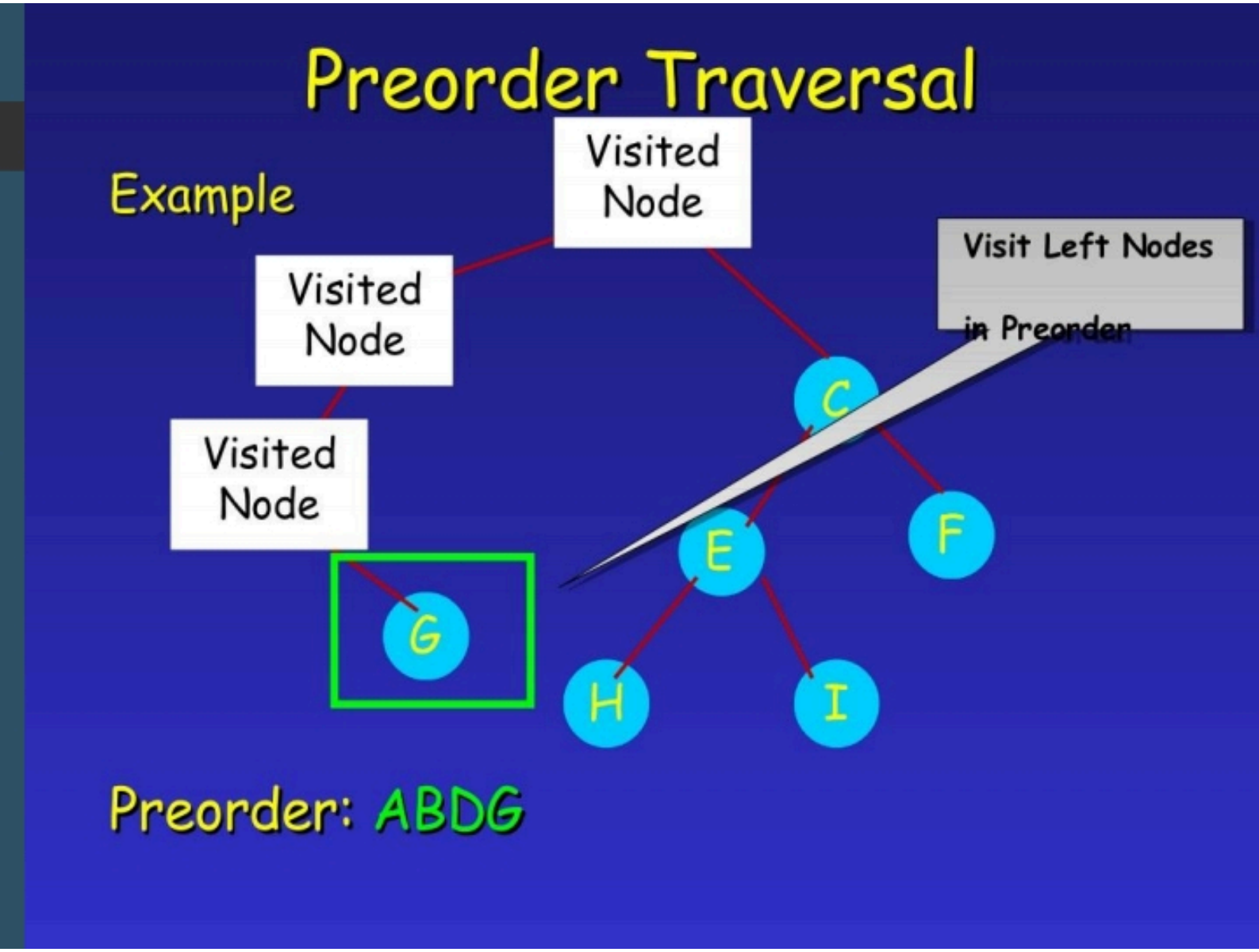


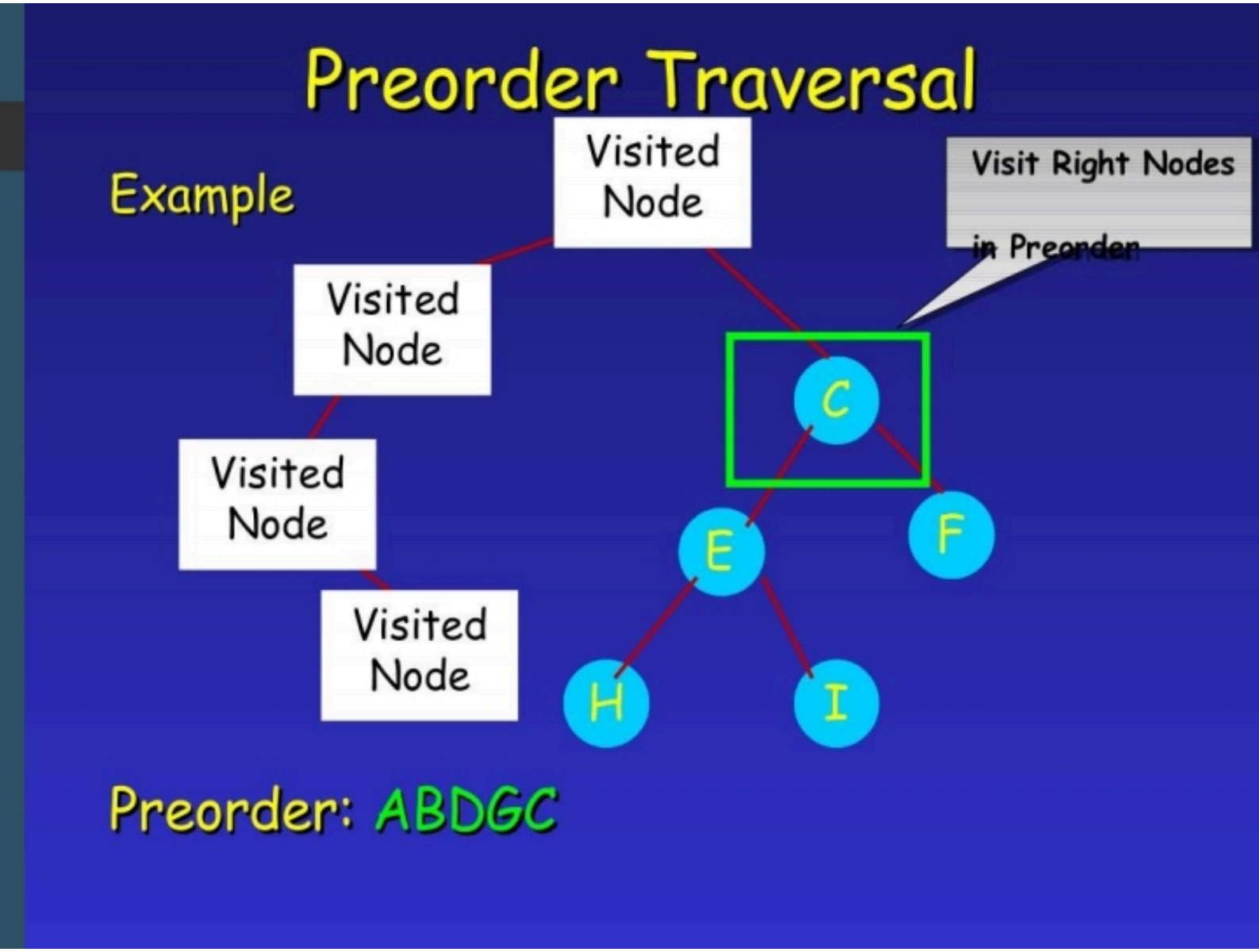
Visit Root Node First

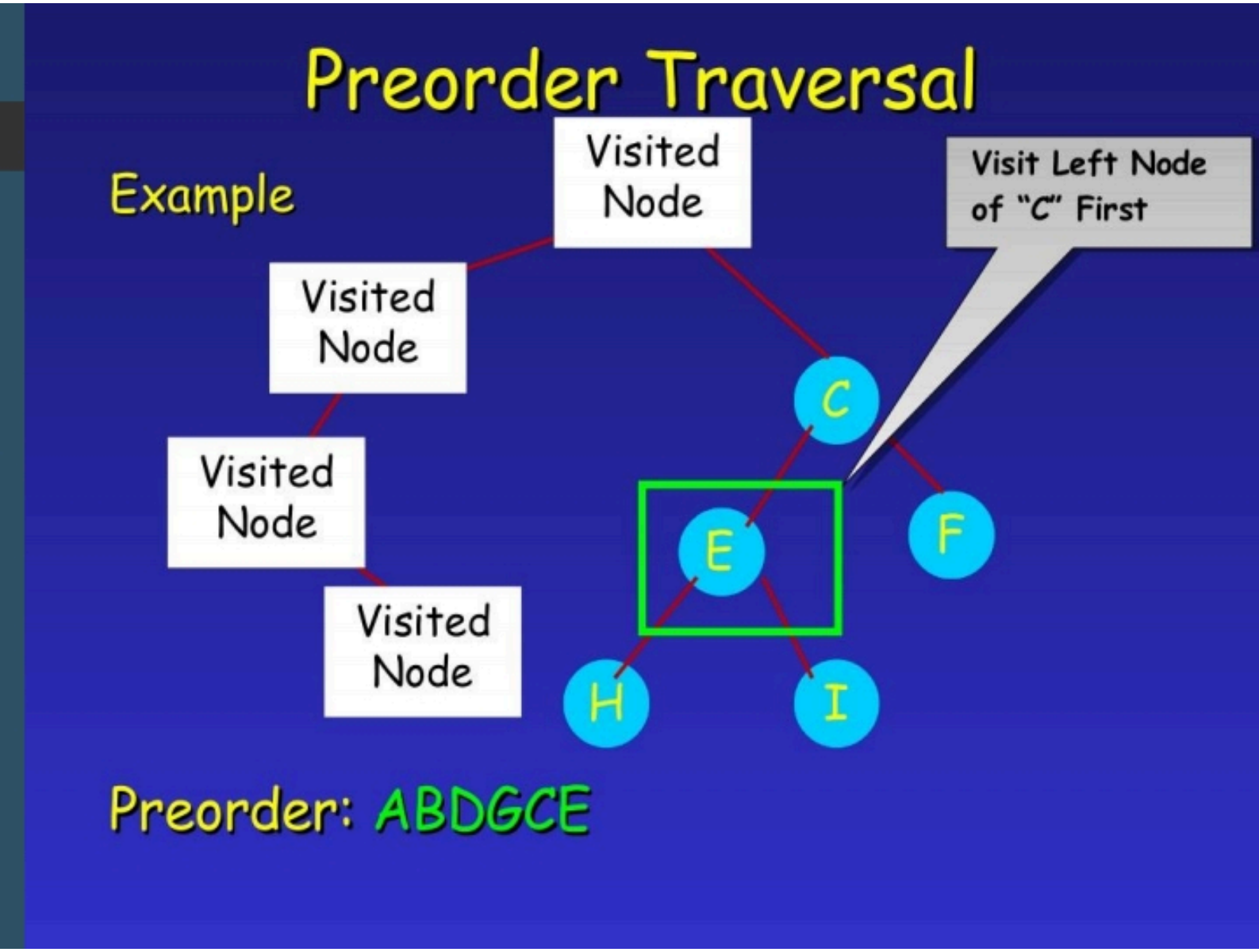
Preorder: A

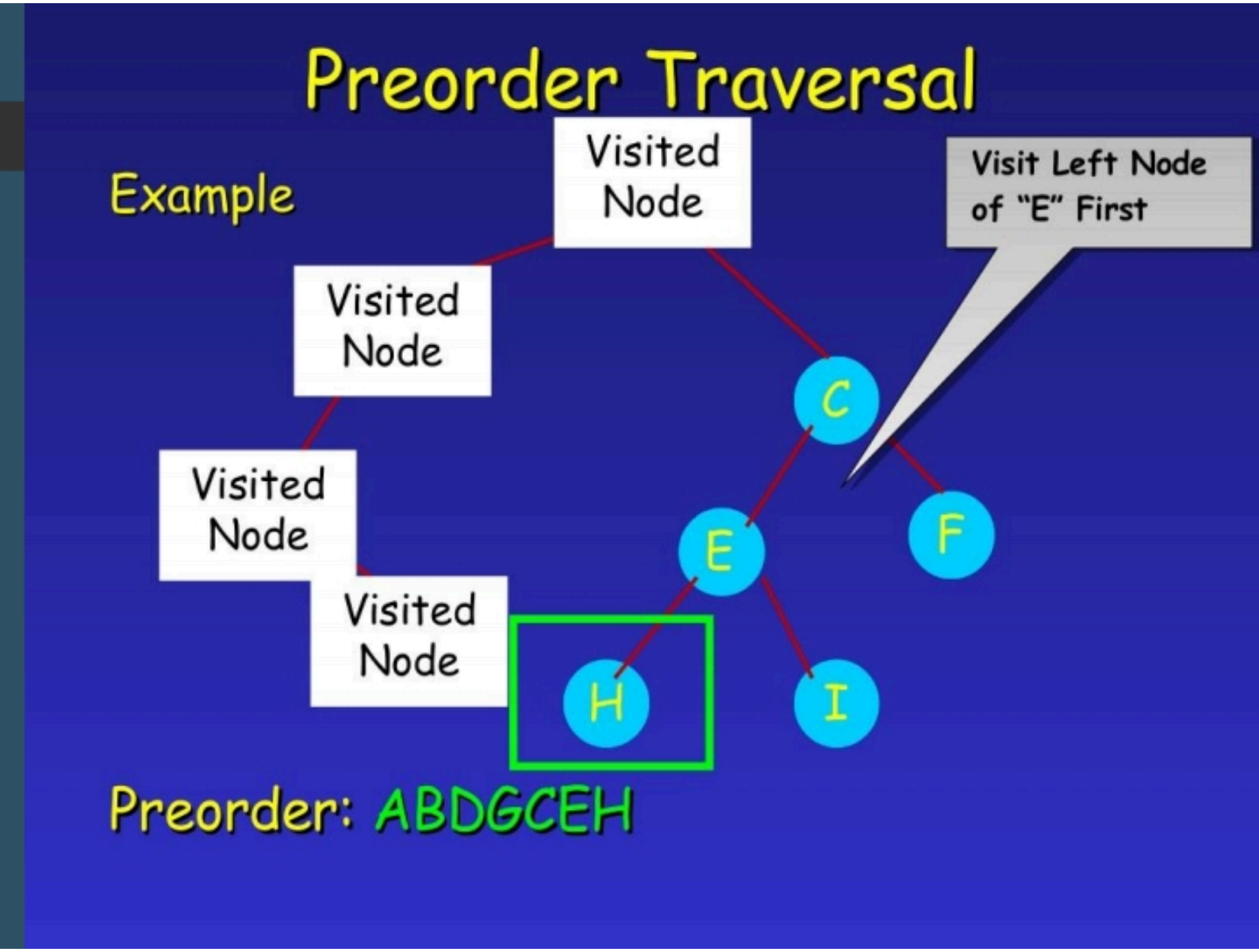

Preorder Traversal
Example
Visited Node
Visit Left Nodes in Preorder
B
D
G
C
E
F
H
I
Preorder: AB

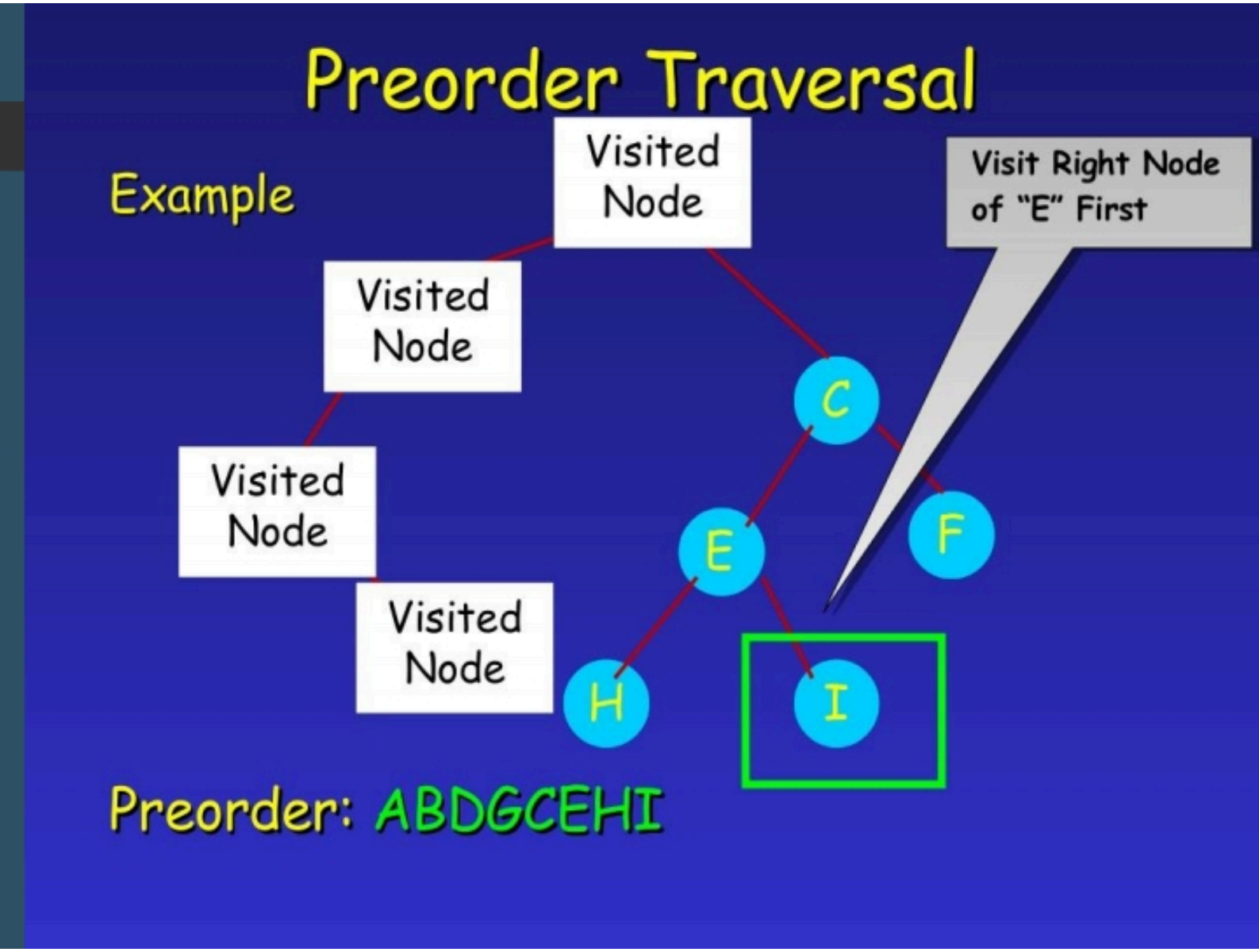

Preorder Traversal
Example
Visited Node
Visited Node
Visit Left Nodes in Preorder
D
G
C
E
F
H
I
Preorder: ABD

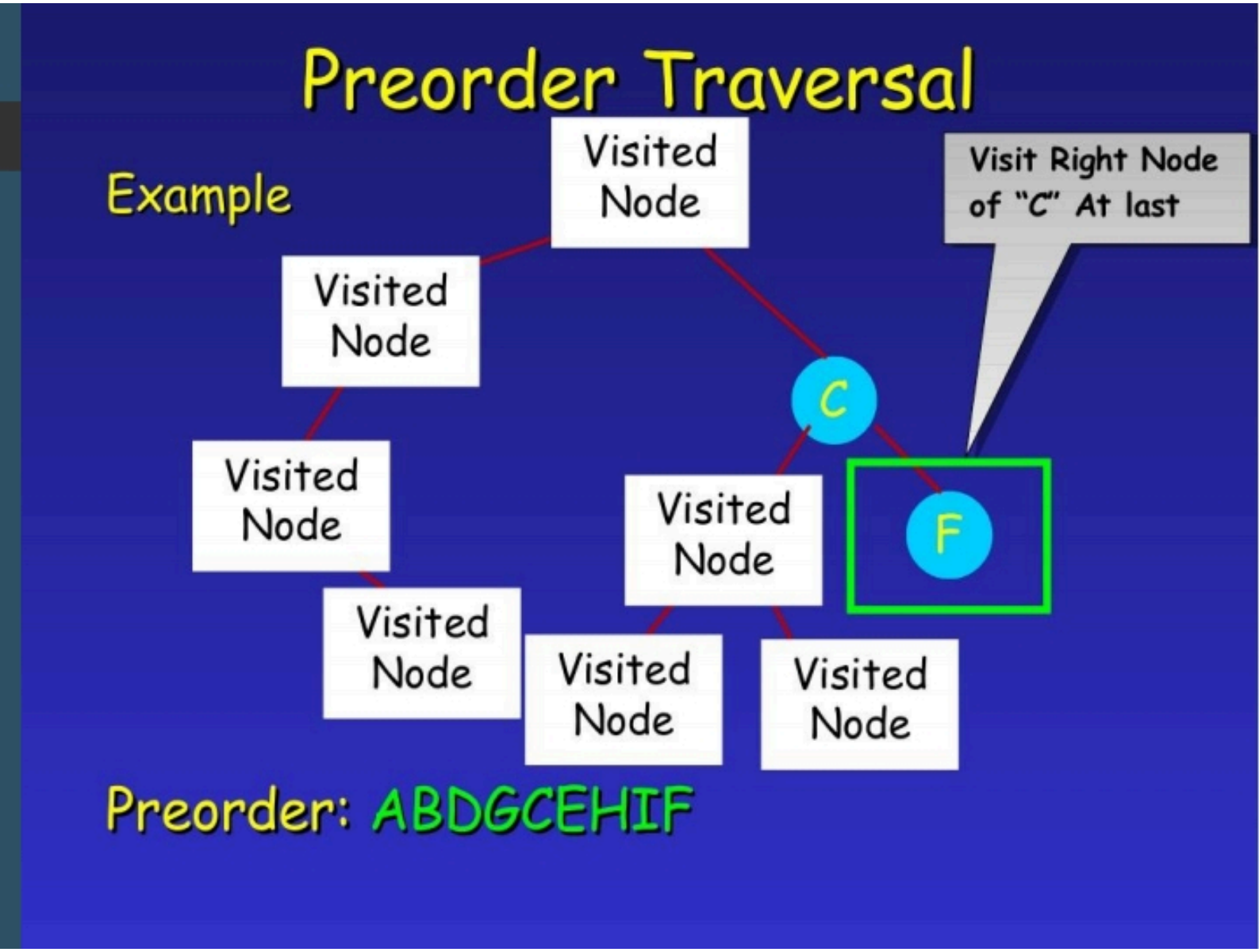

Preorder Traversal
Example
Visited Node
Visited Node
Visited Node
Visit Left Nodes in Preorder
A
B
C
D
E
F
G
H
I
Preorder: ABDG


Preorder Traversal
Example
Visited Node
Visited Node
Visited Node
Visited Node
Visit Right Nodes in Preorder
C
E
F
H
I
Preorder: ABDGC


Preorder Traversal
Example
Visited Node
Visited Node
Visited Node
Visited Node
Visit Left Node of "C" First
C
E
F
H
I
Preorder: ABDGCE

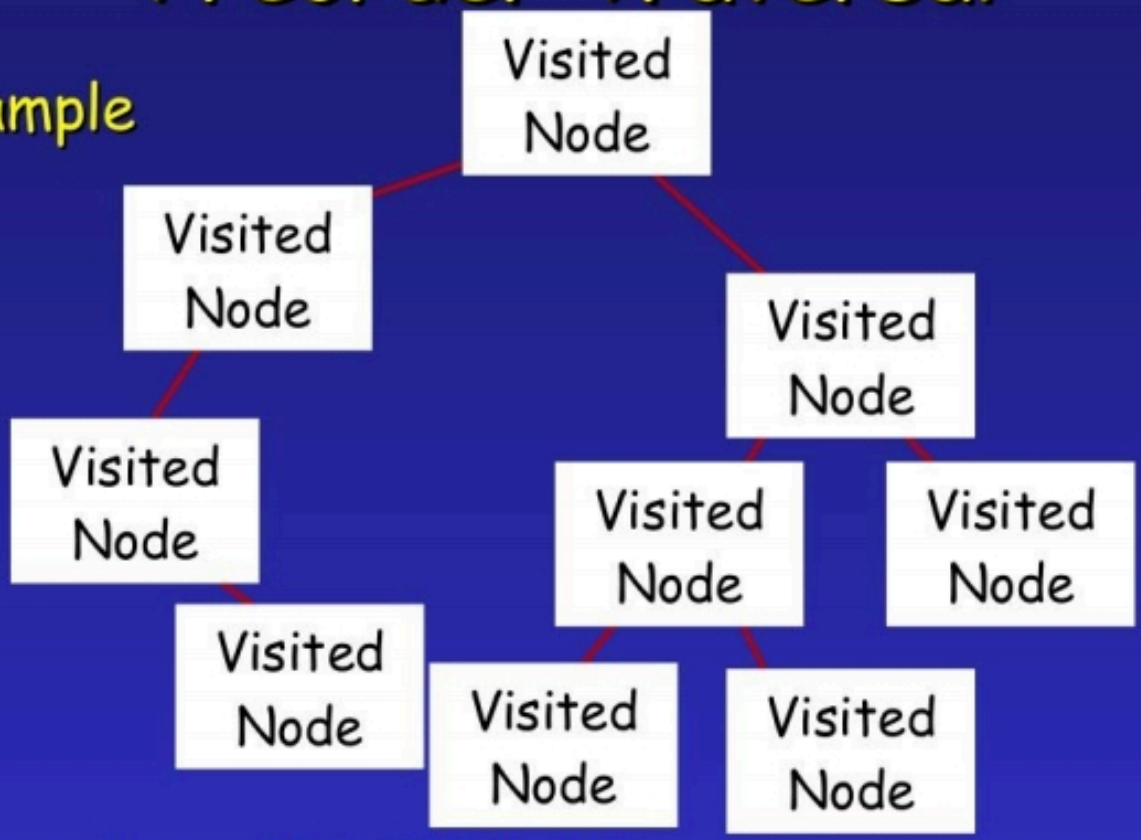

Preorder Traversal
Example
Visited Node
Visited Node
Visited Node
Visited Node
Visit Left Node of "E" First
C
E
F
H
I
Preorder: ABDGCEH


Preorder Traversal
Example
Visited Node
Visited Node
Visited Node
Visited Node
Visit Right Node of "E" First
C
E
F
H
I
Preorder: ABDGCEHI


Preorder Traversal
Example
Visited Node
Visited Node
Visited Node
Visited Node
Visited Node
Visited Node
Visited Node
C
F
Visit Right Node of "C" At last
Preorder: ABDGCEHIF

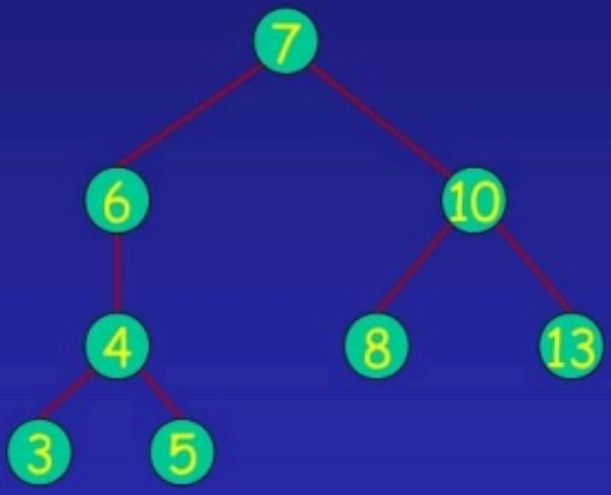
Preorder Traversal

Example



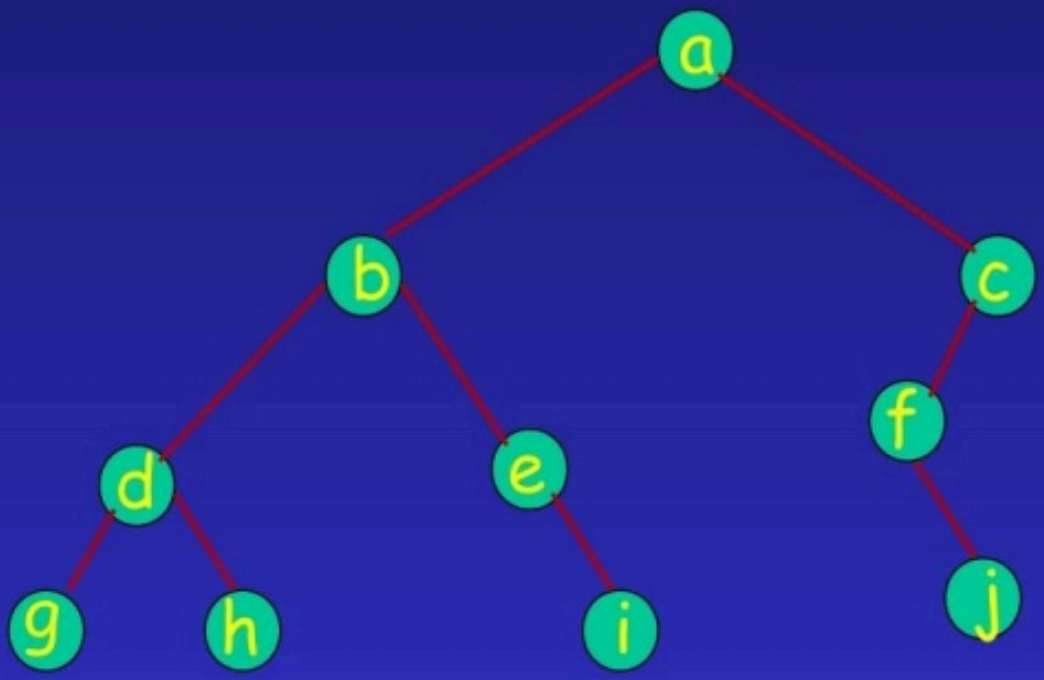
Preorder: ABDGCEHIF

Preorder Traversal



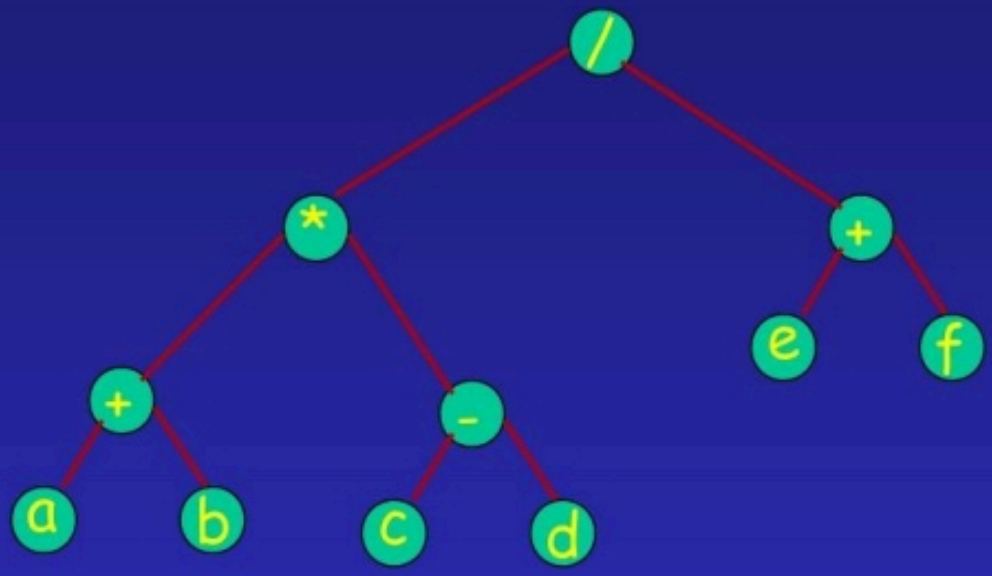
Preorder :7, 6, 4, 3, 5, 10, 8, 13

Preorder Traversal



Preorder: a,b,d,g,h,e,I,c,f,j

Preorder Of Expression Tree



/ * + a b - c d + e f

Gives prefix form of expression!

Algorithm of Preorder Traversal

```
Algorithm preOrder(v)
    visit(v)
    for each child w of v
        preorder (w)
```

Tree Traversal

Postorder traversal

- In a postorder traversal, a node is visited after its descendants

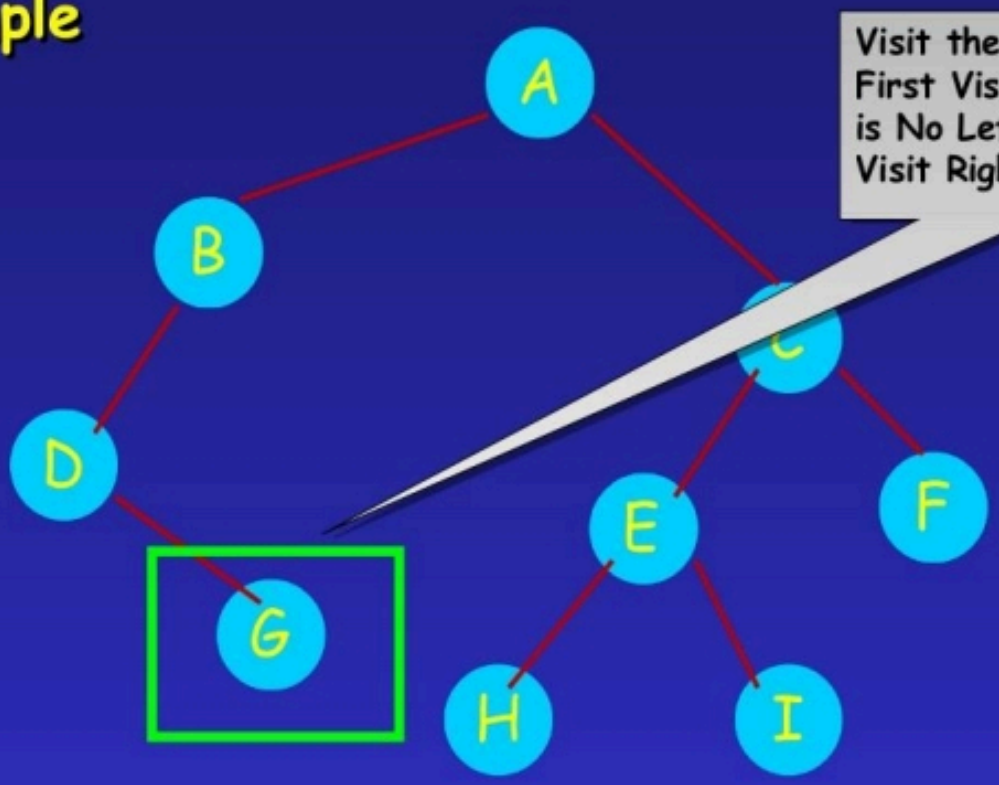
1. Traverse the left subtree in postorder
2. Traverse the right subtree in postorder
3. Visit the root

Postorder traversal

- left, right, node

Postorder traversal

Example

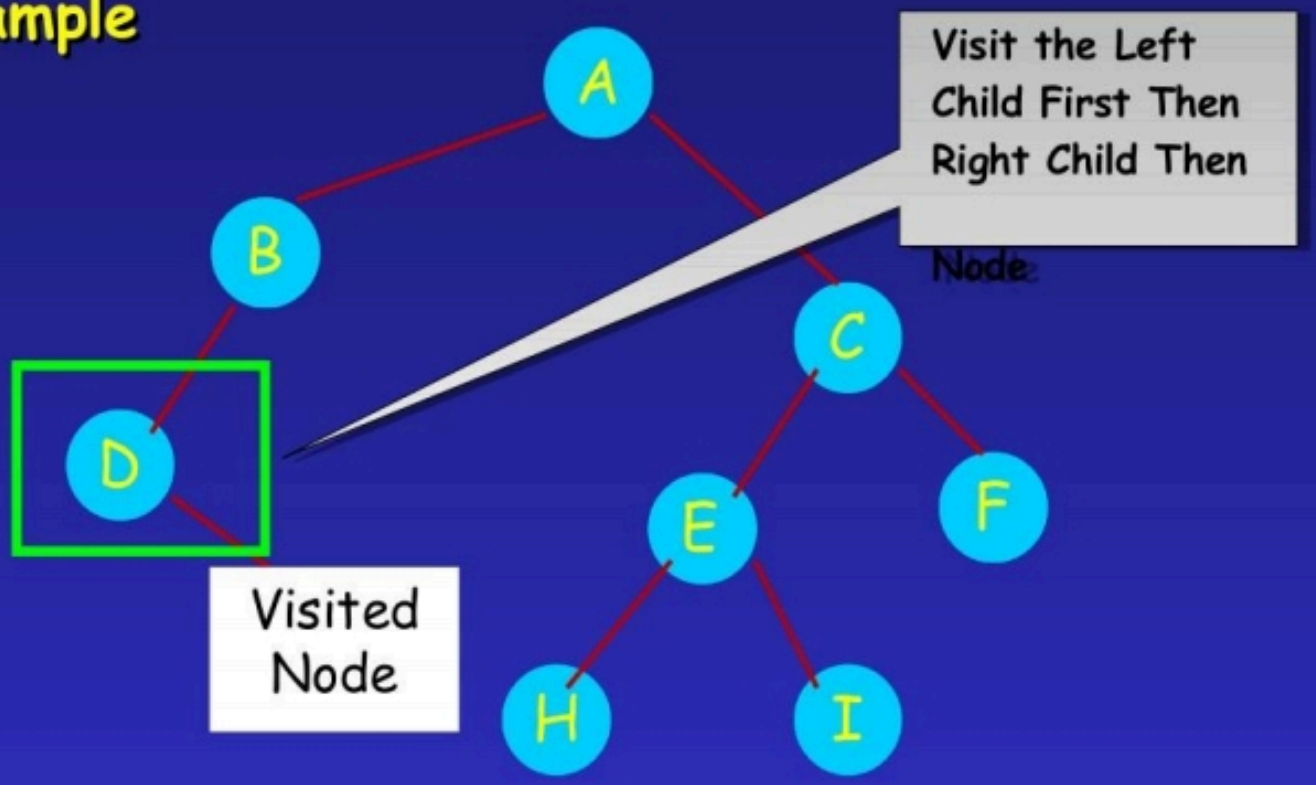


Visit the Left Subtree. First Visit "D" But there is No Left Child so we Visit Right Child of "D"

Postorder: G

Postorder traversal

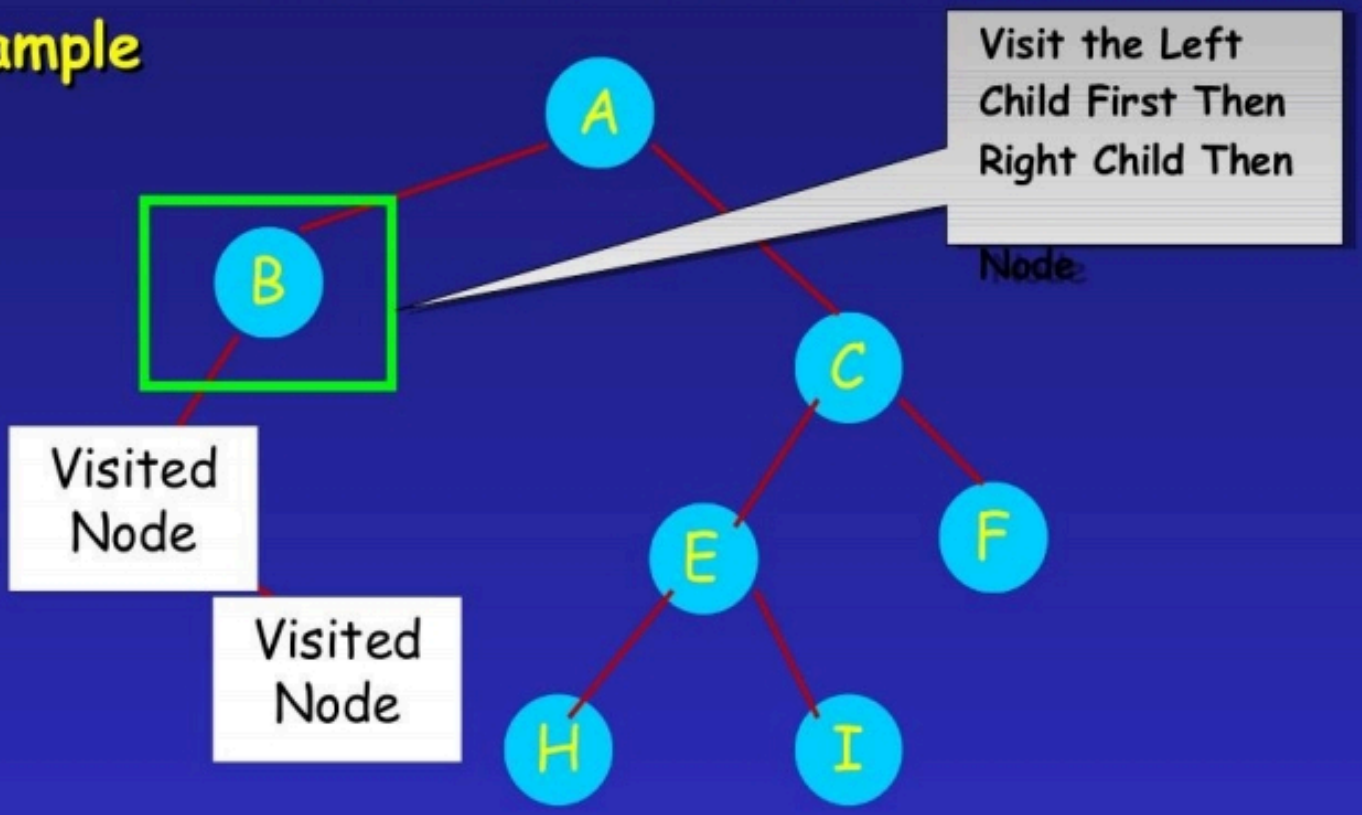
Example



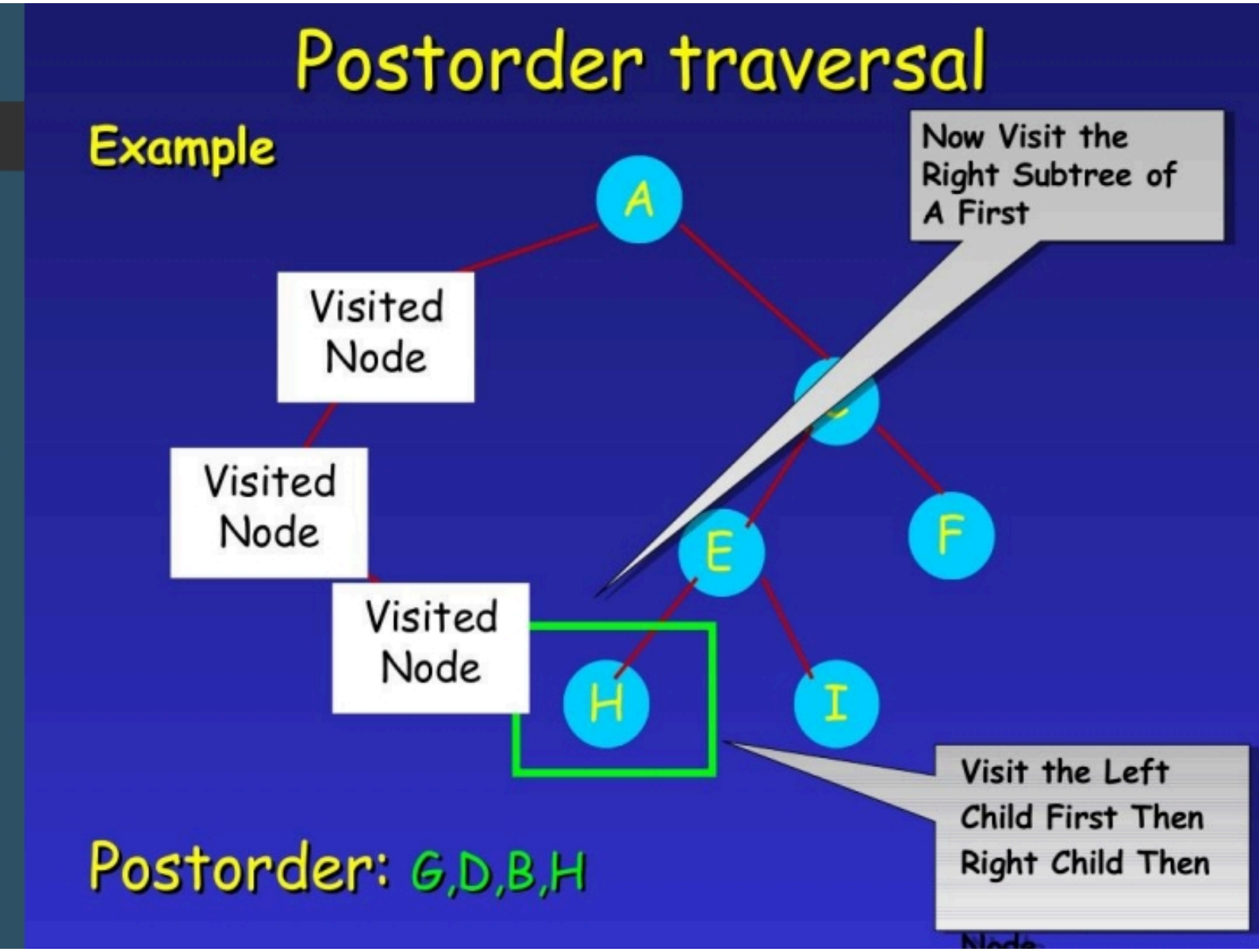
Postorder: G,D

Postorder traversal

Example

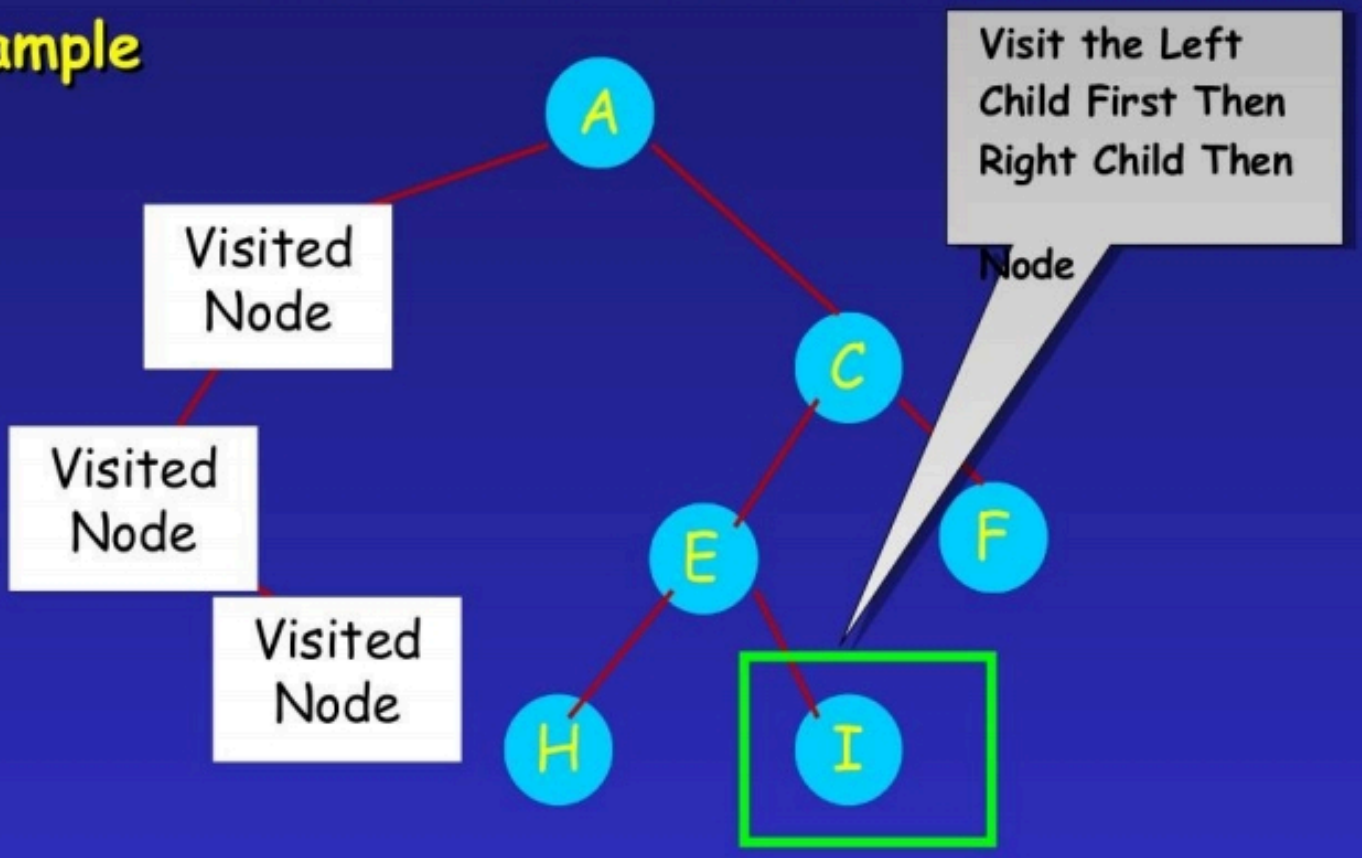


Postorder: G,D,B


Postorder traversal
Example
Now Visit the Right Subtree of A First
A
Visited Node
Visited Node
Visited Node
E
F
H
I
Visit the Left Child First Then Right Child Then
Postorder: G,D,B,H

Postorder traversal

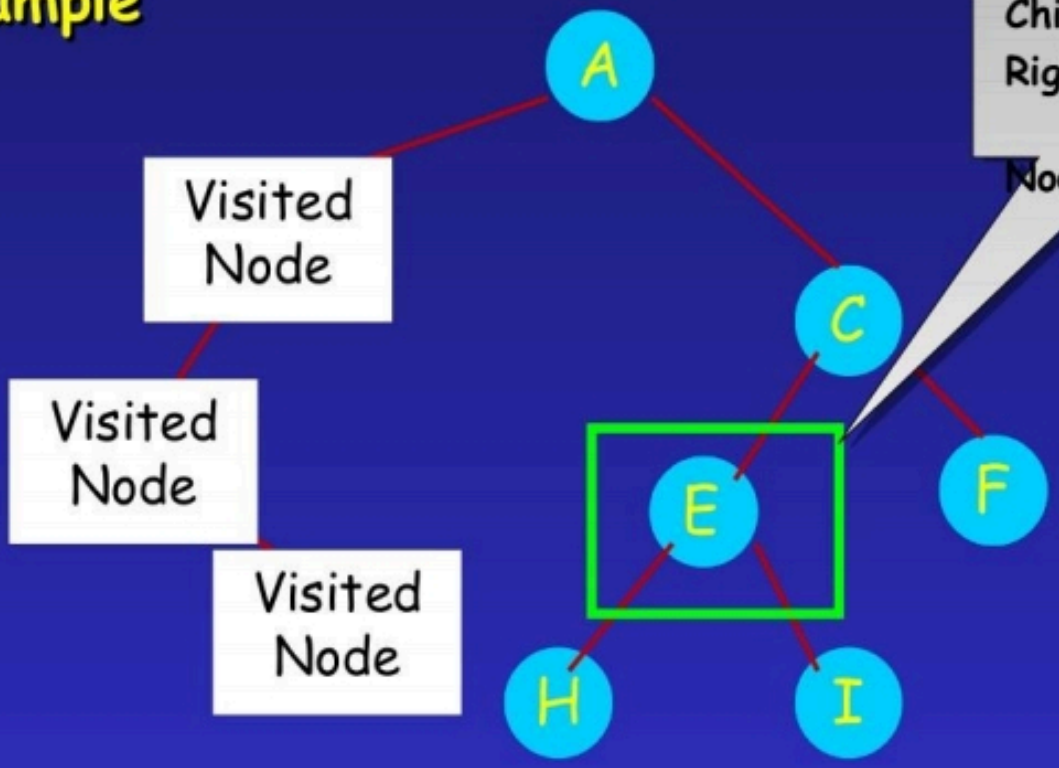
Example



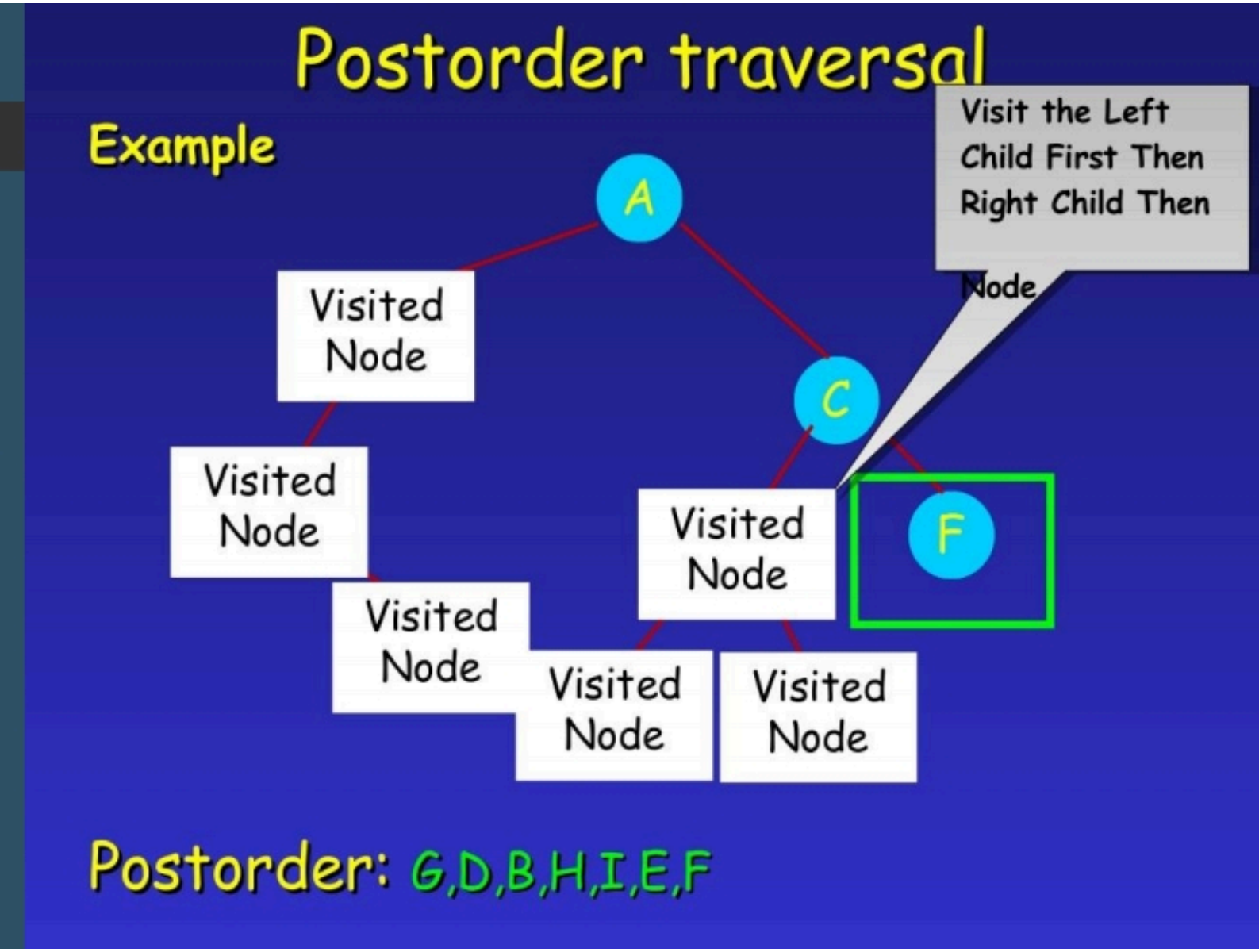
Postorder: G,D,B,H,I

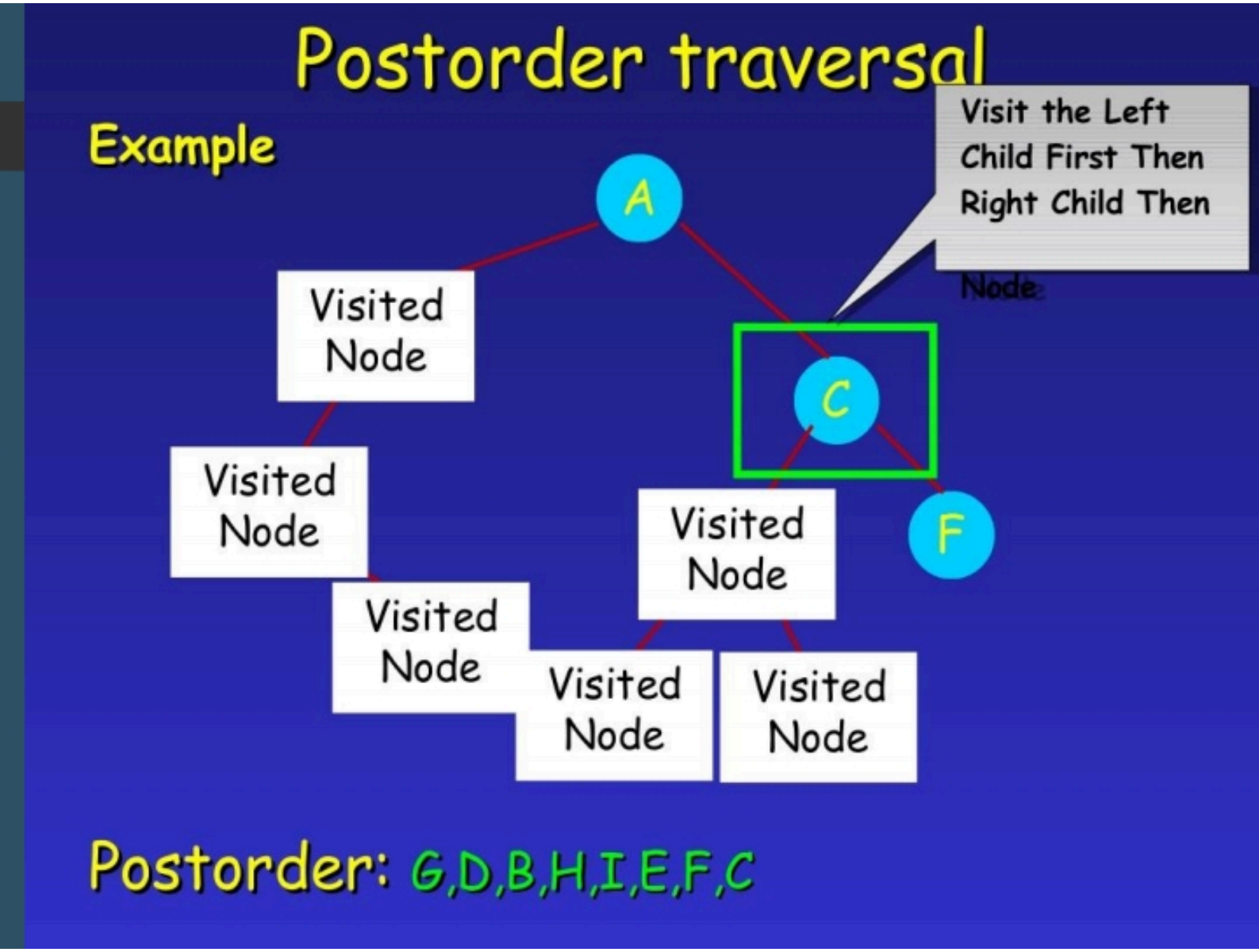
Postorder traversal

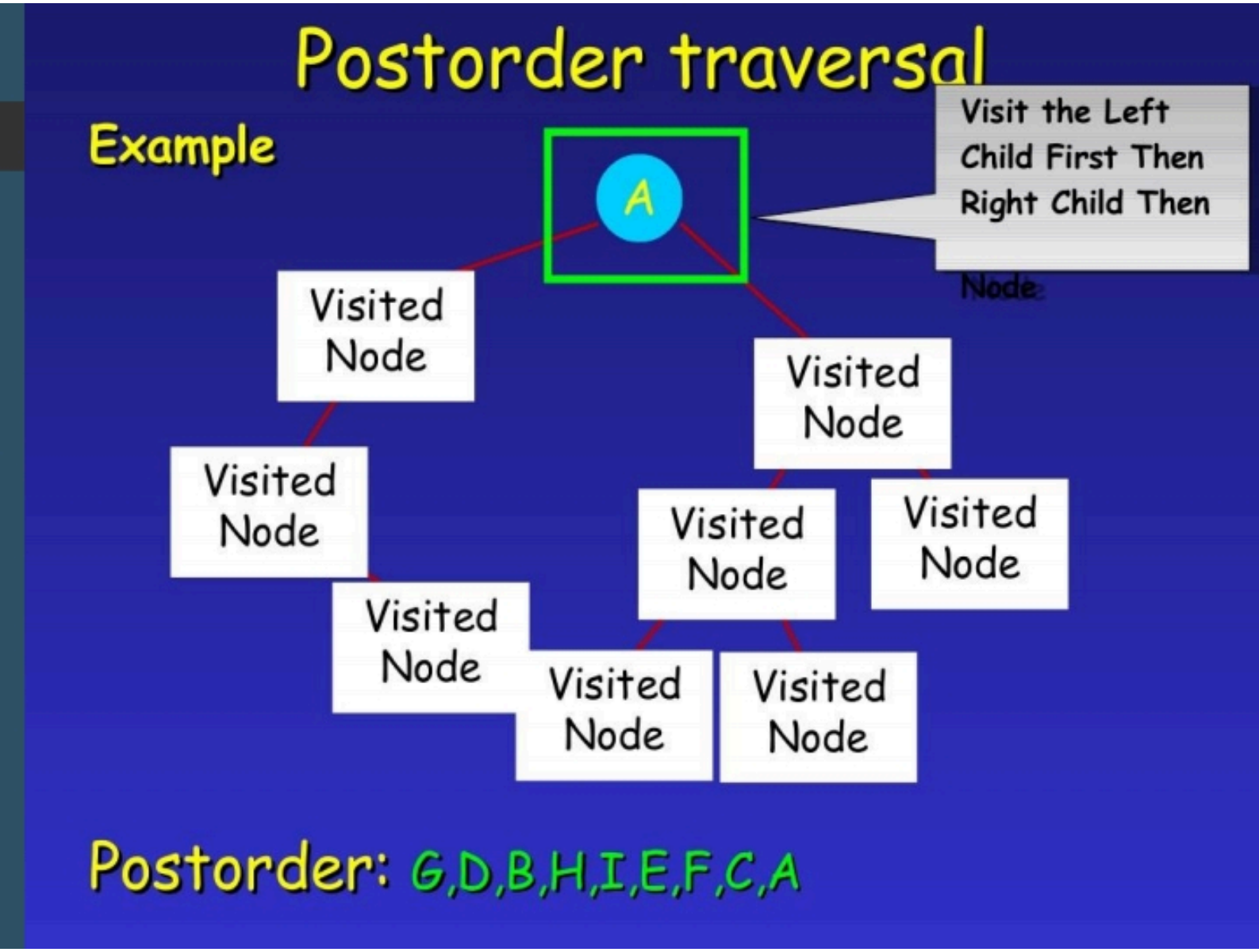
Example



Postorder: G,D,B,H,I,E

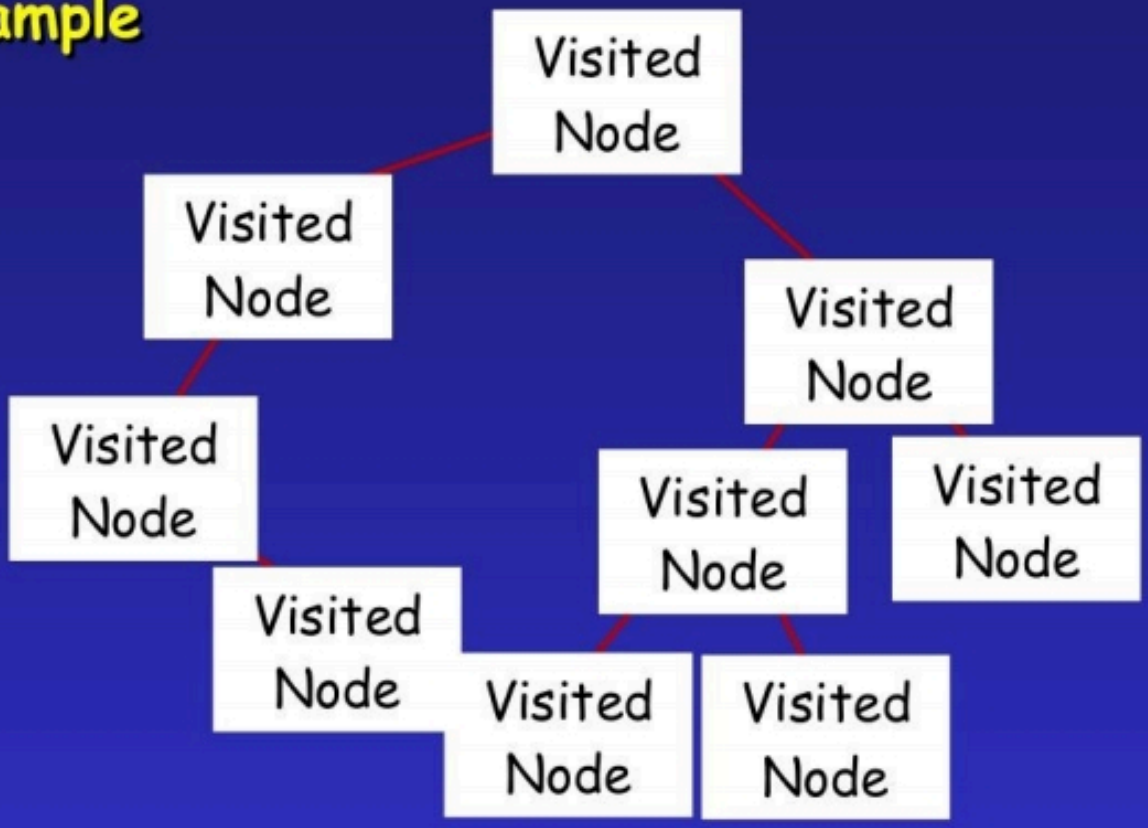

Postorder traversal
Example
Visit the Left Child First Then Right Child Then Node
A
C
F
Visited Node
Visited Node
Visited Node
Visited Node
Visited Node
Visited Node
Postorder: G,D,B,H,I,E,F


Postorder traversal
Example
Visit the Left Child First Then Right Child Then Node
A
C
F
Visited Node
Visited Node
Visited Node
Visited Node
Visited Node
Visited Node
Postorder: G,D,B,H,I,E,F,C


Postorder traversal
Example
A
Visit the Left Child First Then Right Child Then Node
Visited Node
Visited Node
Visited Node
Visited Node
Visited Node
Visited Node
Visited Node
Visited Node
Visited Node
Postorder: G,D,B,H,I,E,F,C,A

Postorder traversal

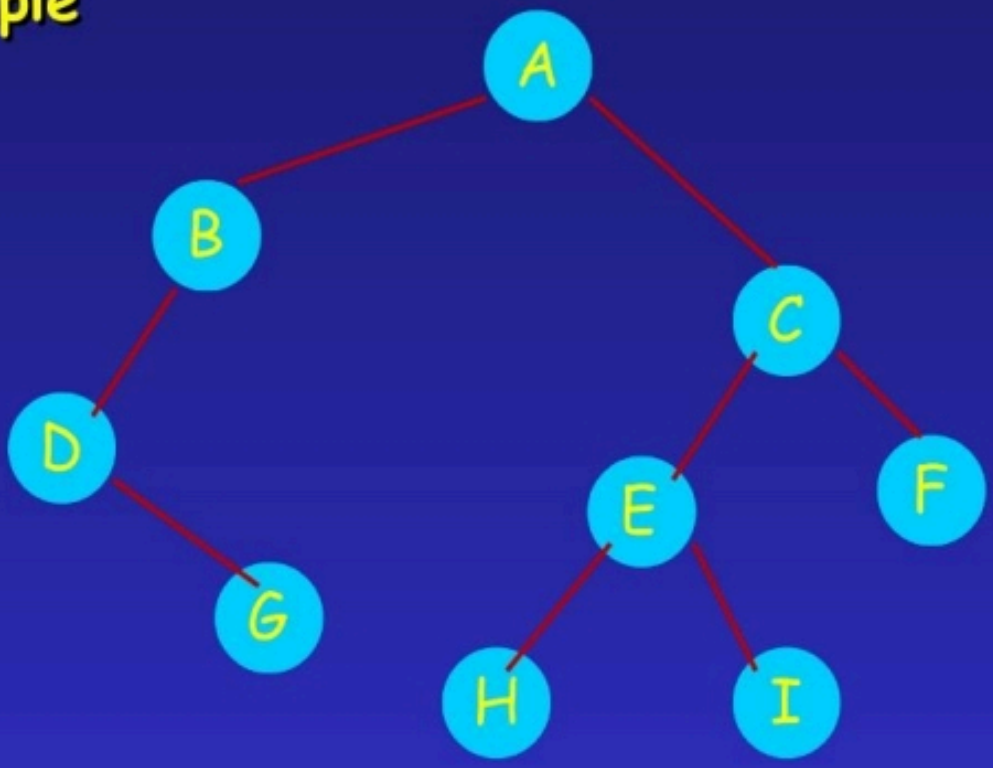
Example



Postorder: G,D,B,H,I,E,F,C,A

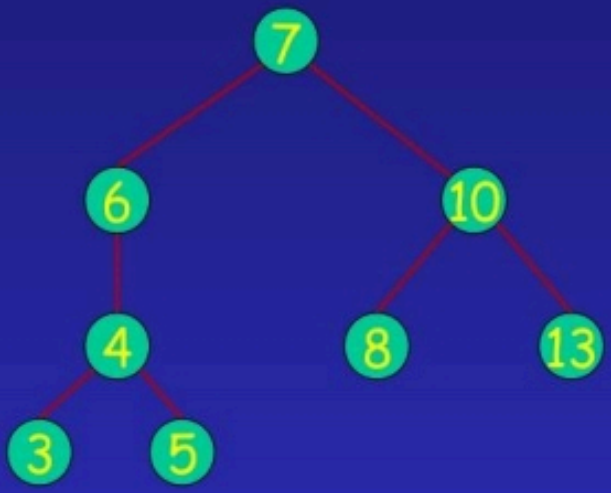
Postorder traversal

Example



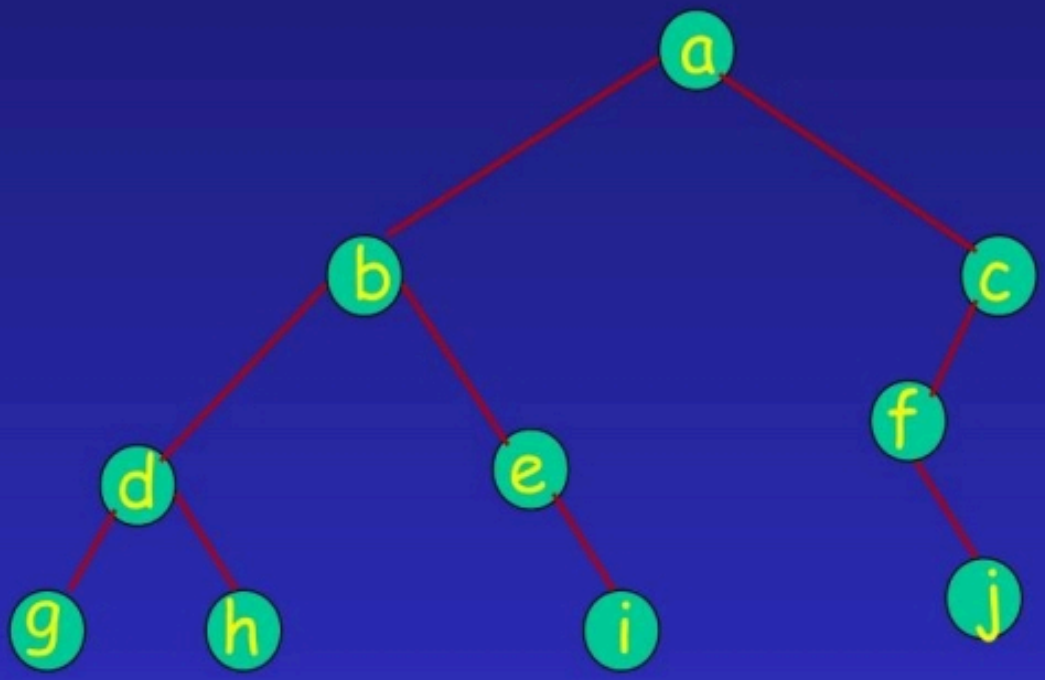
Postorder: G,D,B,H,I,E,F,C,A

Postorder traversal



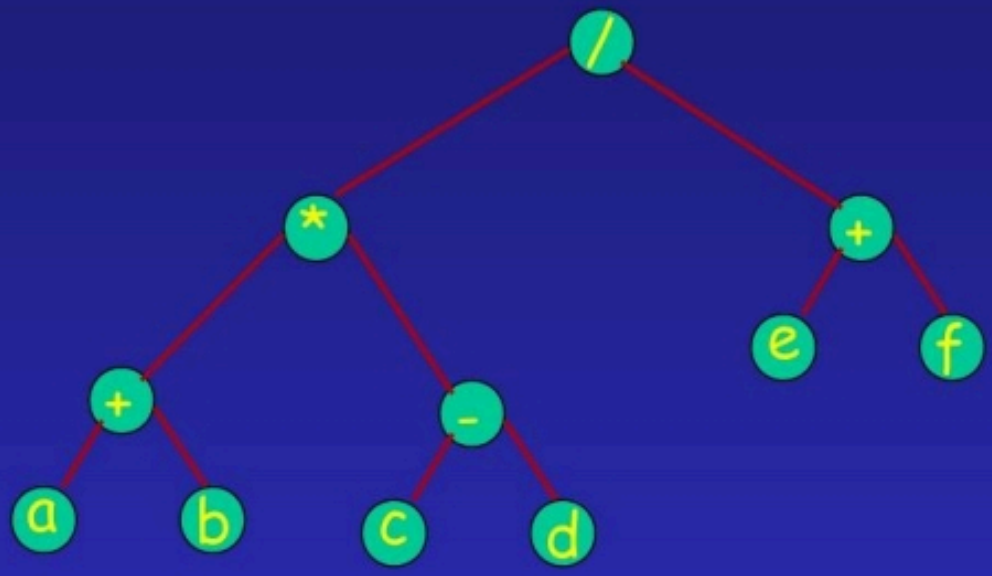
Postorder: 3, 5, 4, 6, 8, 13, 10, 7

Postorder traversal



Postorder: g,h,d,ie,b,j,f,c,a

Postorder Of Expression Tree



a b + c d - * e f + /

Gives postfix form of expression!

Algorithm of Postorder Traversal

Algorithm ***postOrder(v)***

```
for each child w of v
    postOrder (w)
visit(v)
```

Tree Traversal

Inorder traversal

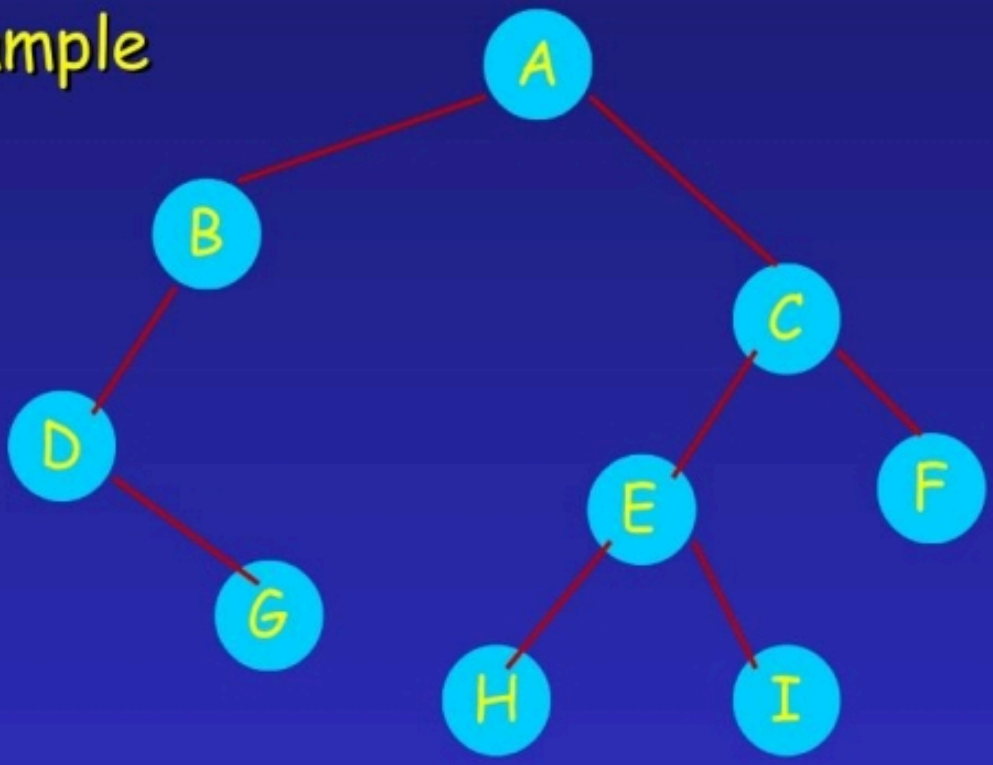
1. Traverse the left subtree in inorder
2. Visit the root
3. Traverse the right subtree in inorder

Inorder traversal

- left, node, right

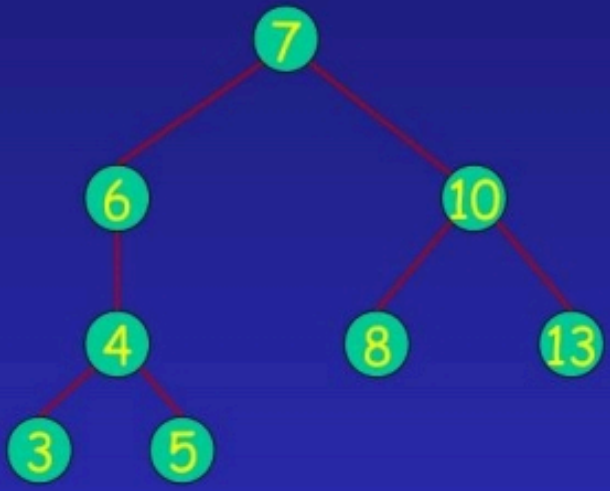
Inorder traversal

Example



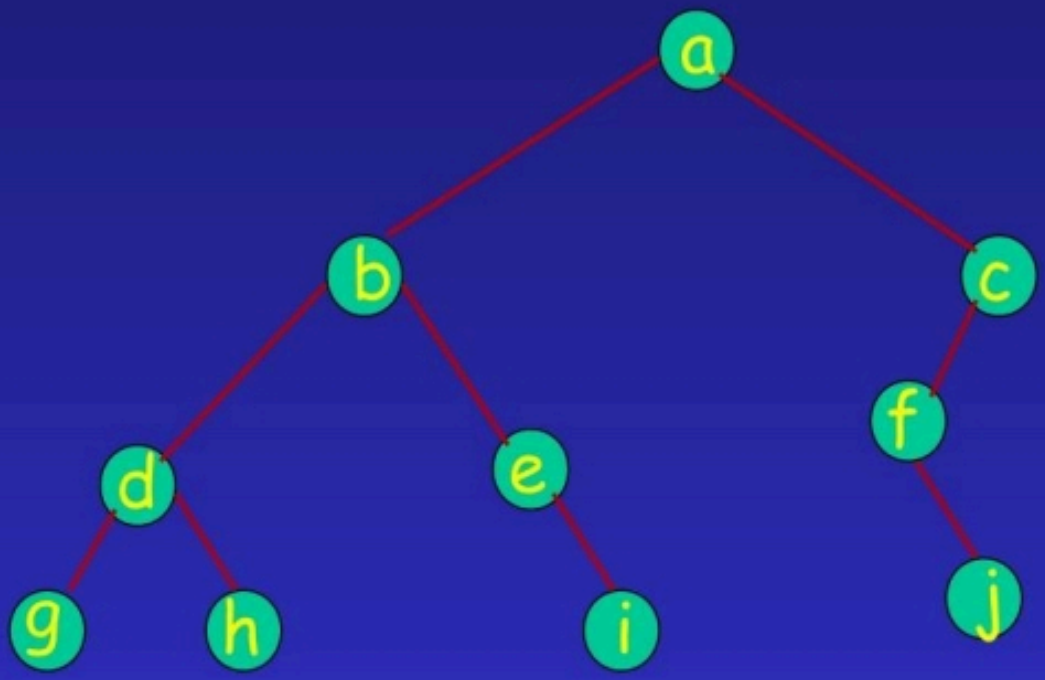
Inorder: D,G,B,A,H,E,I,C,F

Inorder traversal



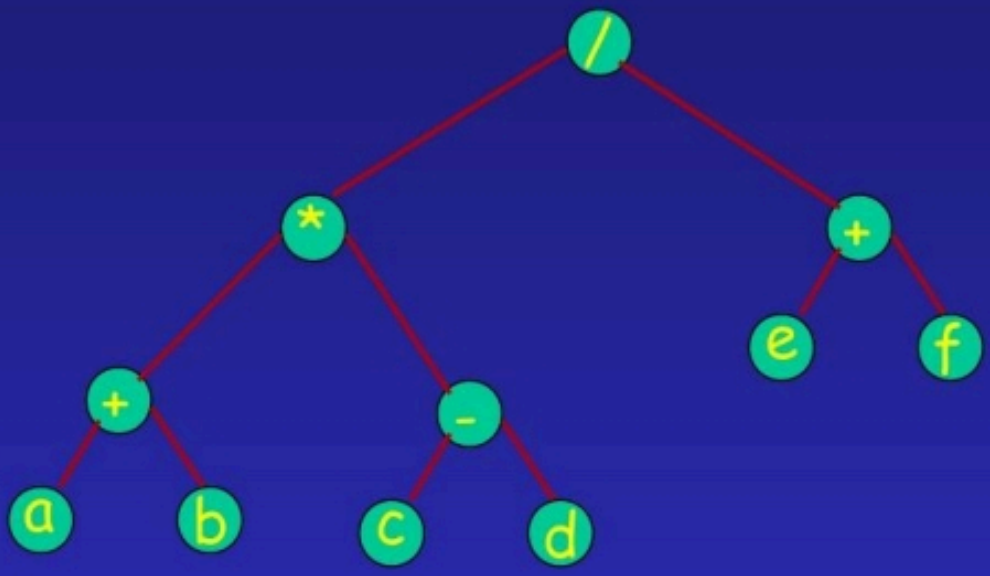
Inorder: 3, 4, 5, 6, 7, 8, 10, 13

Inorder traversal



Inorder :

Inorder Of Expression Tree



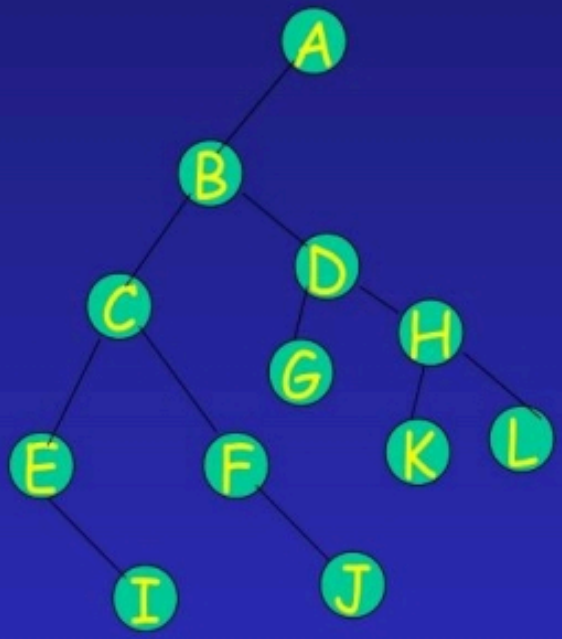
a + b * c - d / e + f

Gives infix form of expression!

Algorithm of Inorder Traversal

```
Algorithm inOrder(v)
    if hasLeft (v)
        inOrder (left (v))
    visit(v)
    if hasRight (v)
        inOrder (right (v))
```

Your task

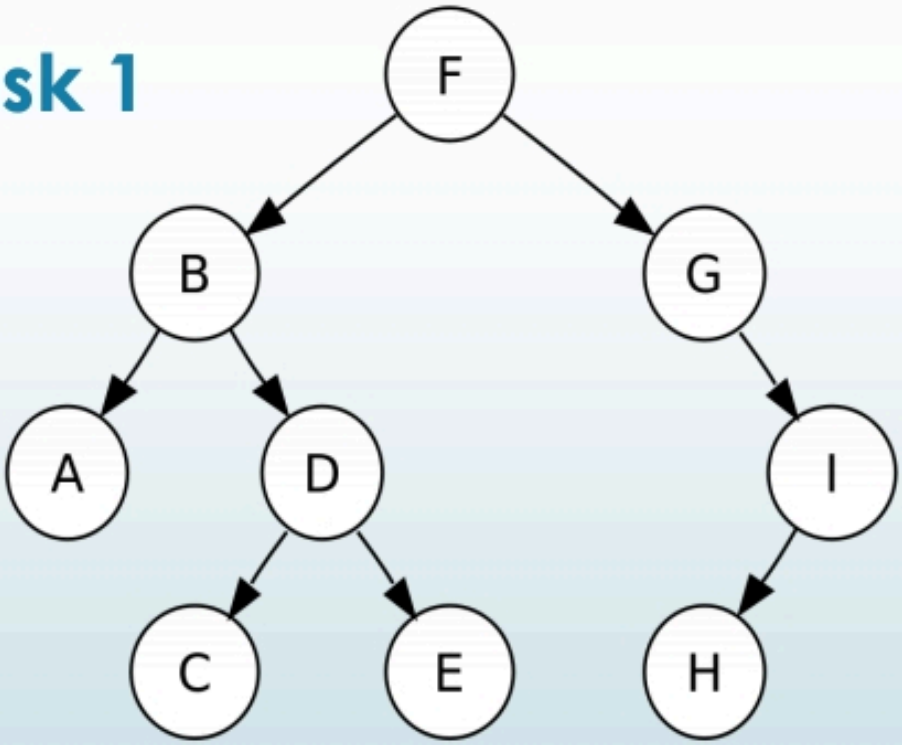


Preorder: ABCEIFJDGHKL

Postorder: IEJFCGKLHDBA

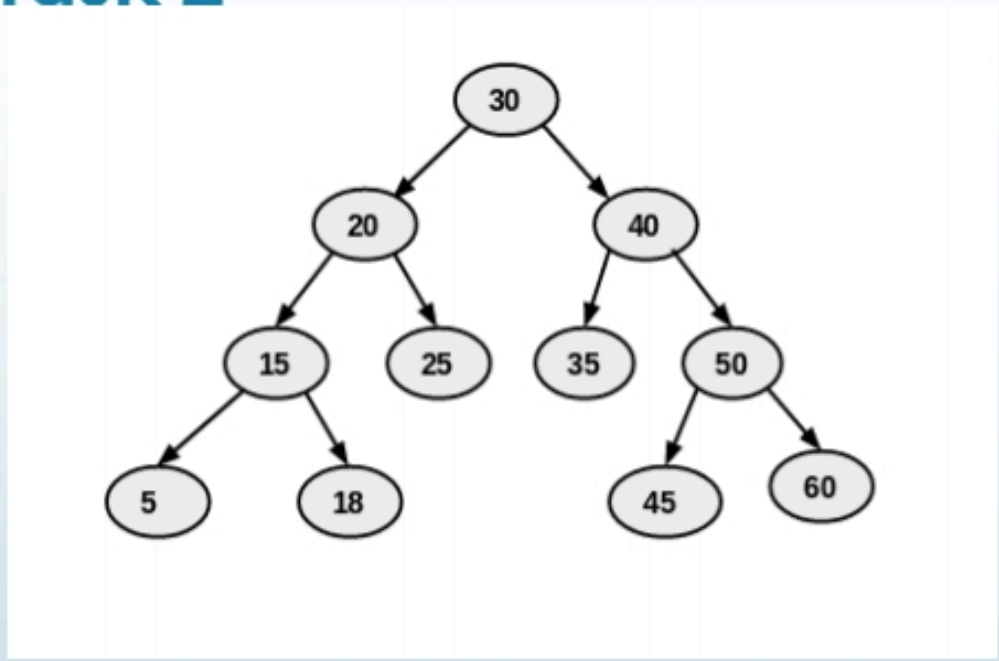
Inorder: EICFJBGDKHLA

Task 1



- **Pre-order -** F, B, A, D, C, E, G, I, H
- **Post-order -** A, C, E, D, B, H, I, G, F
- **In-order -** A, B, C, D, E, F, G, H, I

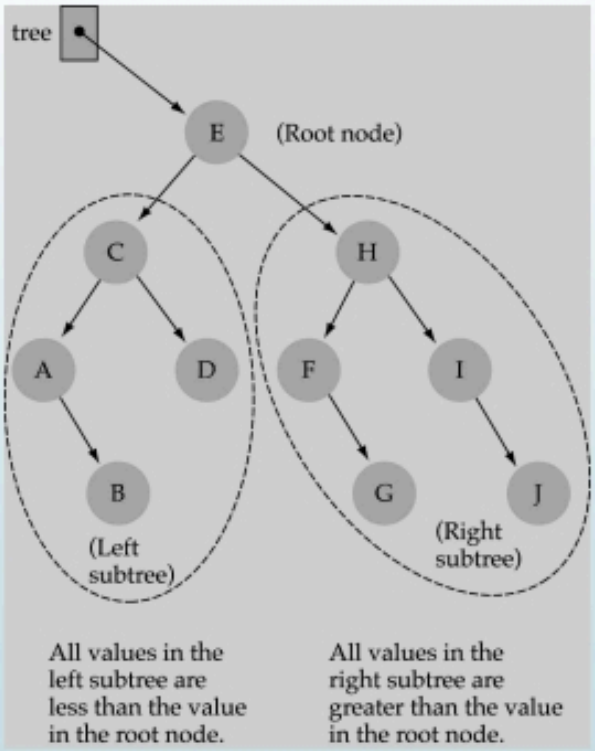
Task 2



- **Pre-order -** 30 , 20 , 15 , 5 , 18 , 25 , 40 , 35 , 50 , 45 , 60
- **Post-order -** 5 , 18 , 15 , 25 , 20 , 35 , 45 , 60 , 50 , 40 , 30
- **In-order -** 5 , 15 , 18 , 20 , 25 , 30 , 35 , 40 , 45 , 50 , 60

Binary Search Trees (BSTs)

In a BST, the value stored at the root of a subtree is *greater* than any value in its left subtree and *less* than any value in its right subtree!



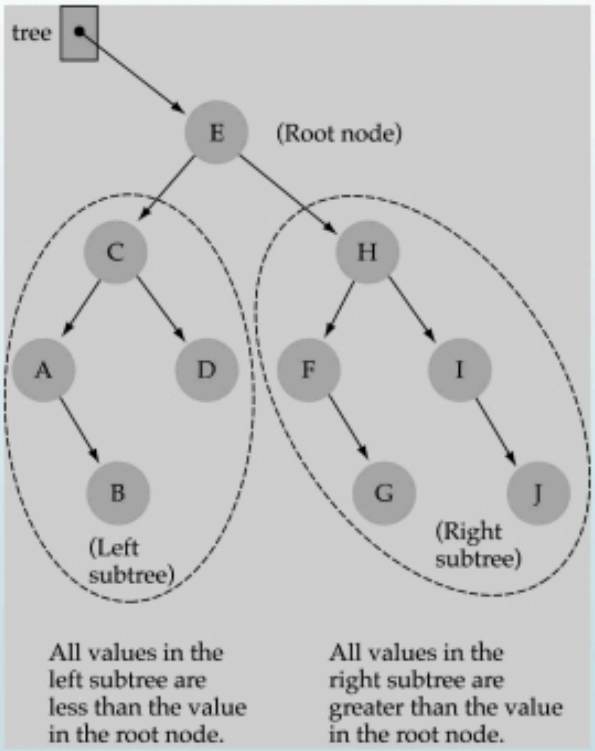
Binary Search Trees (BSTs)

Where is the smallest element?

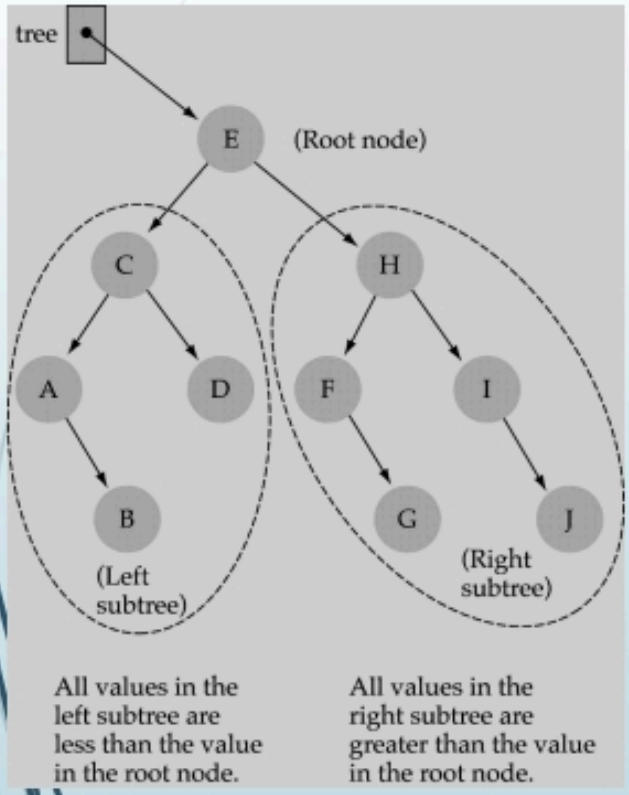
Ans: leftmost element

Where is the largest element?

Ans: rightmost element



How to search a binary search tree?

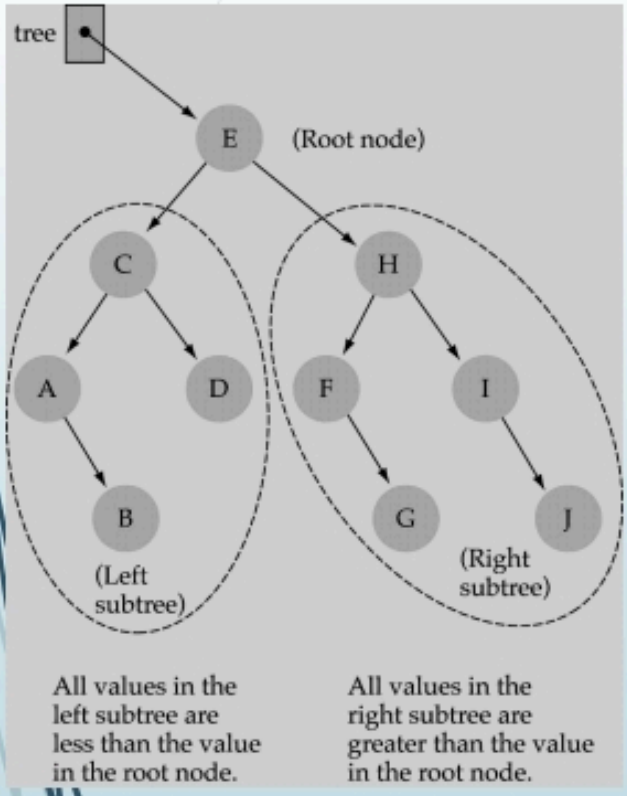


(1) Start at the root

(2) Compare the value of the item you are searching for with the value stored at the root

(3) If the values are equal, then *item found*; otherwise, if it is a leaf node, then *not found*

How to search a binary search tree?

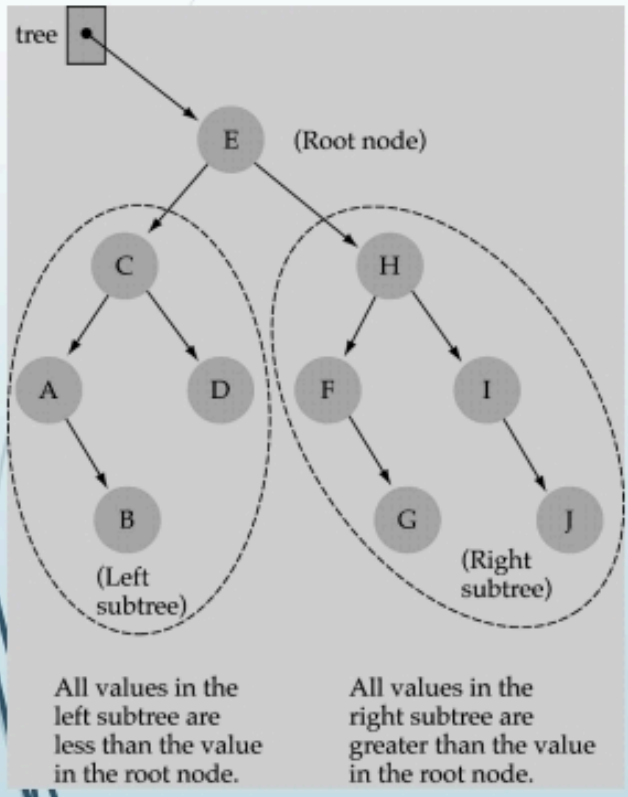


(4) If it is less than the value stored at the root, then search the left subtree

(5) If it is greater than the value stored at the root, then search the right subtree

(6) Repeat steps 2-6 for the root of the subtree chosen in the previous step 4 or 5

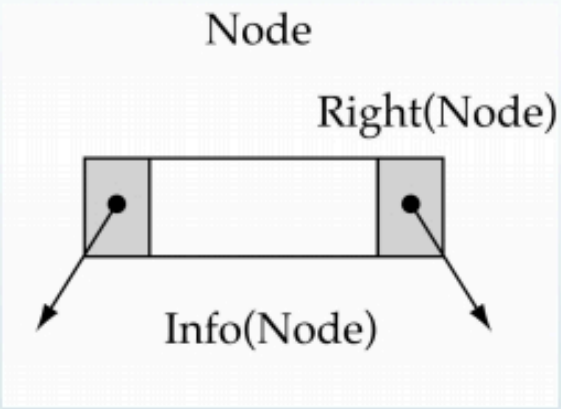
How to search a binary search tree?



Is this better than searching a linked list?

Yes !! ---> O(logN)

Tree node structure

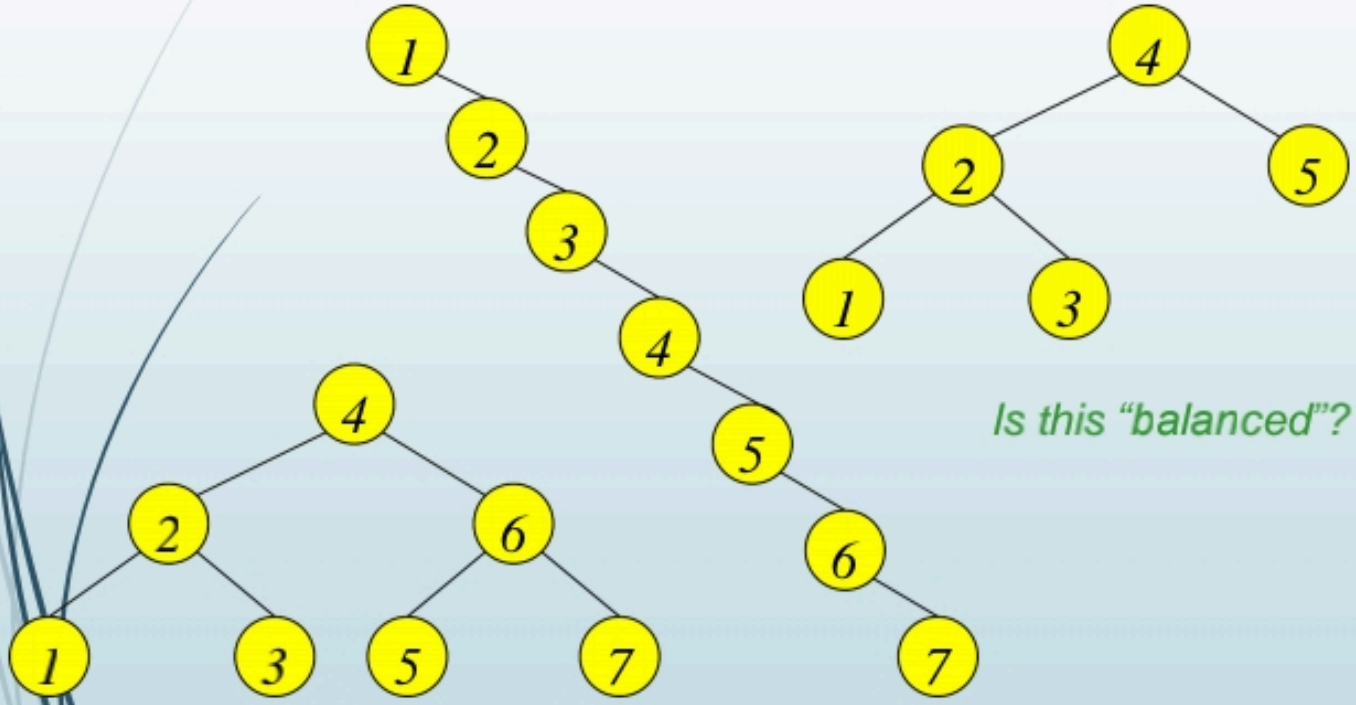



```
template<class ItemType>
    struct TreeNode<ItemType> {
ItemType info;
            TreeNode<ItemType>* left;
             TreeNode<ItemType>* right;
                 };
```

Balanced binary tree

- The disadvantage of a binary search tree is that its height can be as large as N-1
- This means that the time needed to perform insertion and deletion and many other operations can be O(N) in the worst case
- We want a tree with small height
- A binary tree with N node has height at least (Θlog N)
- Thus, our goal is to keep the height of a binary search tree O(log N)
- Such trees are called balanced binary search trees.
  Examples are AVL tree, red-black tree.

Balanced and unbalanced BST



Is this "balanced"?

Approaches to balancing trees

- Don't balance
  - May end up with some nodes very deep
- Strict balance
  - The tree must always be balanced perfectly
- Pretty good balance
  - Only allow a little out of balance
- Adjust on access
  - Self-adjusting

Balancing Binary Search Trees

- Many algorithms exist for keeping binary search trees balanced
  - Adelson-Velskii and Landis (AVL) trees (height-balanced trees)
  - Splay trees and other self-adjusting trees
  - B-trees and other multiway search trees

AVL Tree is…

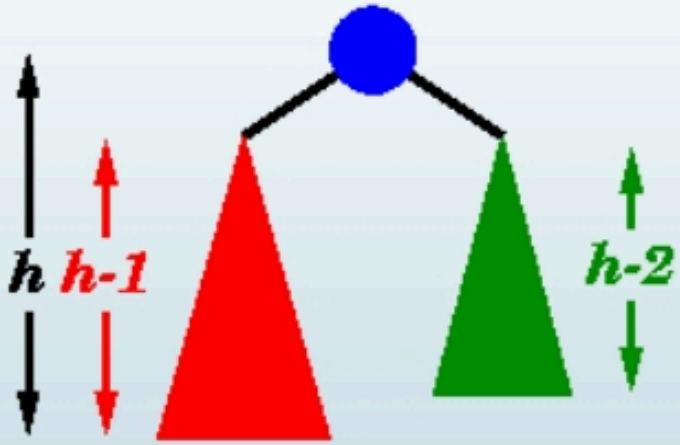
- Named after **A**delson-**V**elskii and **L**andis
- the first dynamically balanced trees to be propose
- Binary search tree with **balance condition** in which the sub-trees of each node can differ by at most 1 in their height

Definition of a balanced tree

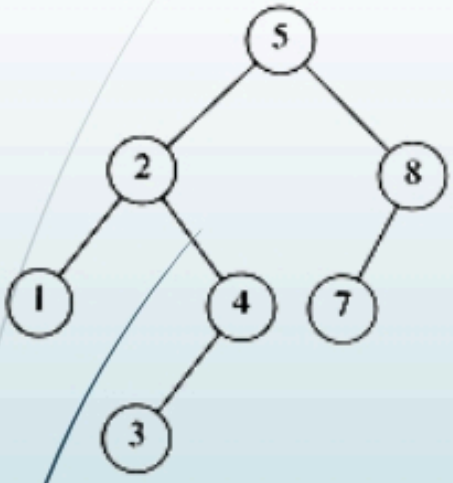
- Ensure the depth = O(log N)
- Take O(log N) time for searching, insertion, and deletion
- Every node must have left & right sub-trees of the same height

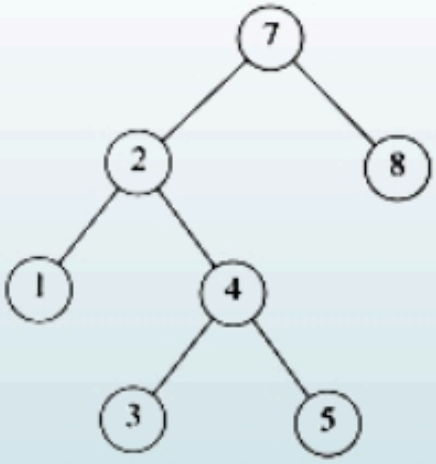
An AVL tree has the following properties:

1. Sub-trees of each node can differ by at most 1 in their height
2. Every sub-trees is an AVL tree



AVL tree?





YES

Each left sub-tree has height 1 greater than each right sub-tree

NO

Left sub-tree has height 3, but right sub-tree has height 1

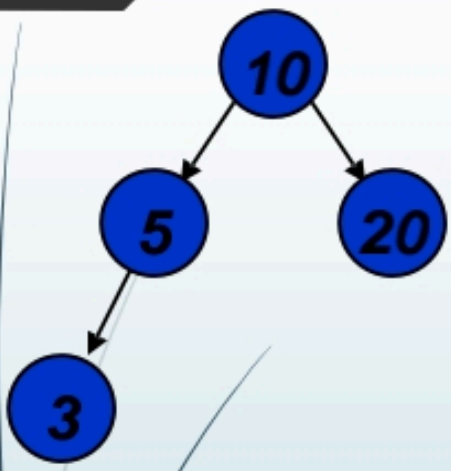
AVL tree

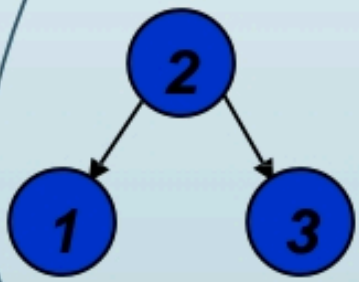
Height of a node

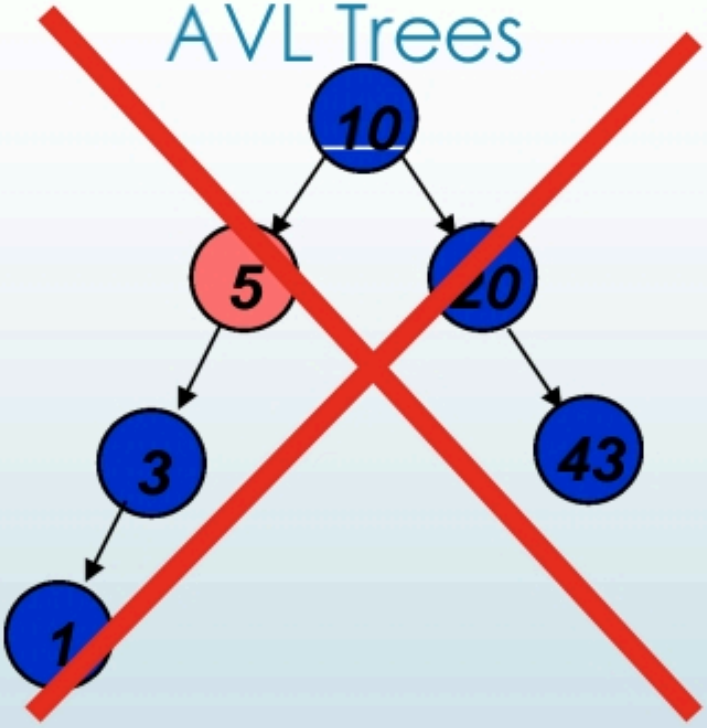
- The height of a leaf is 1.The height of a null pointer is zero.
- The height of an internal node is the maximum height of its children plus 1

Note that this definition of height is different from the one we defined previously (we defined the height of a leaf as zero previously).

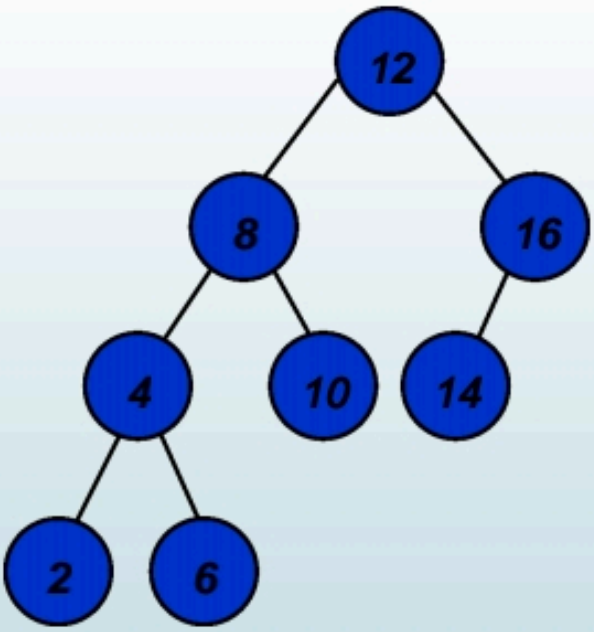
AVL Trees



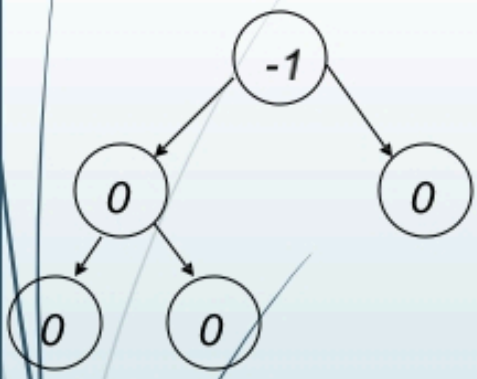




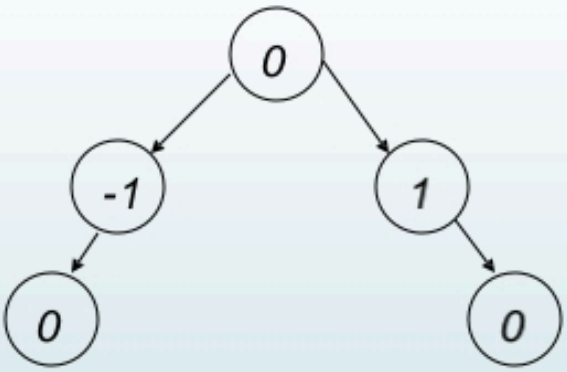
AVL Trees



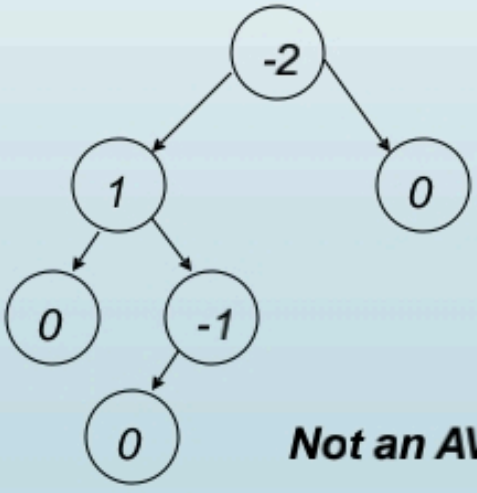
AVL Tree



AVL Tree



AVL Tree

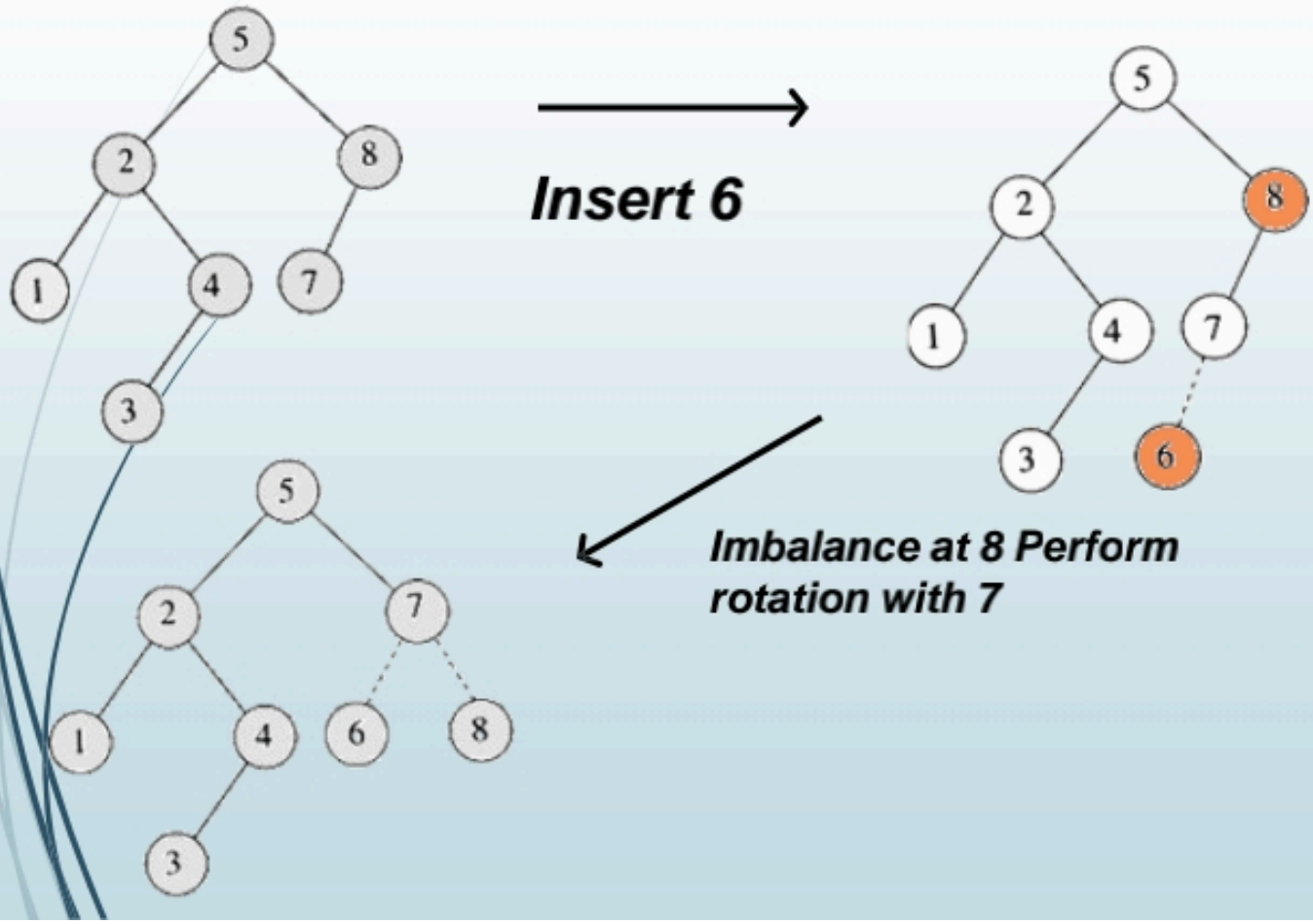


Not an AVL Tree

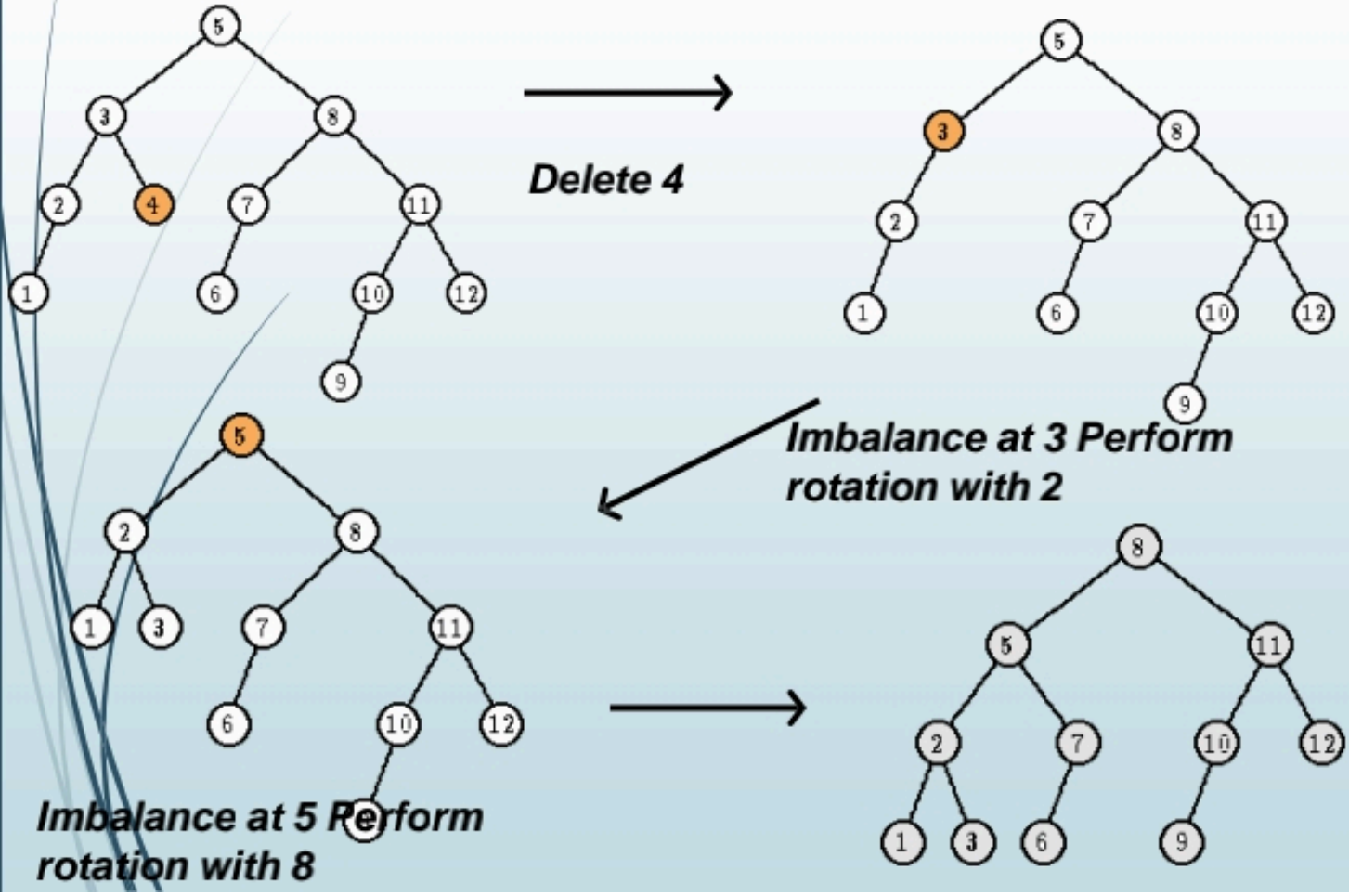
AVL - Good but not Perfect Balance

- AVL trees are height-balanced binary search trees
- Balance factor of a node
  - height(left subtree) - height(right subtree)
- An AVL tree has balance factor calculated at every node
  - For every node, heights of left and right subtree can differ by no more than 1
  - Store current heights in each node

Insertion


5
2
8
1
4
7
3
Insert 6
5
2
8
1
4
7
3
6
Imbalance at 8 Perform rotation with 7
5
2
7
1
4
6
8
3

Deletion


5
3
8
2
4
7
11
1
6
10
12
9
Delete 4
5
3
8
2
7
11
1
6
10
12
9
Imbalance at 3 Perform rotation with 2
5
2
8
1
3
7
11
6
10
12
Imbalance at 5 Perform rotation with 8
8
5
11
2
7
10
12
1
3
6
9

Graphs

- A data structure that consists of a set of nodes (*vertices*) and a set of edges that relate the nodes to each other
- The set of edges describes relationships among the vertices .
- A graph *G* is defined as follows:

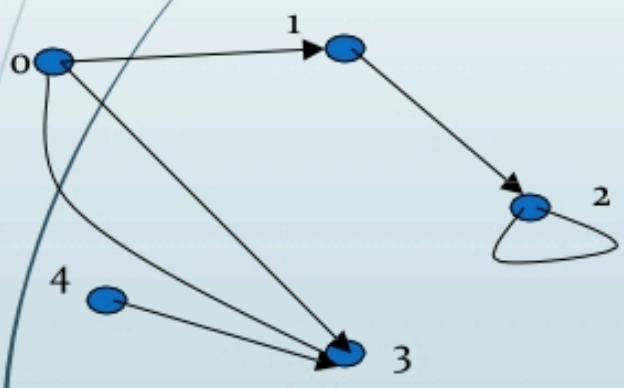
$$G=(V,E)$$

V(G): a finite, nonempty set of vertices

E(G): a set of edges (pairs of vertices)

Examples of Graphs

- V={0,1,2,3,4}
- E={(0,1), (1,2), (0,3), (3,0), (2,2), (4,3)}



When (x,y) is an edge,
we say that x is *adjacent to* y, and y is *adjacent from* x.

0 is adjacent to 1.
1is not adjacent to 0.
2is adjacent from 1.

Directed vs. Undirected Graphs

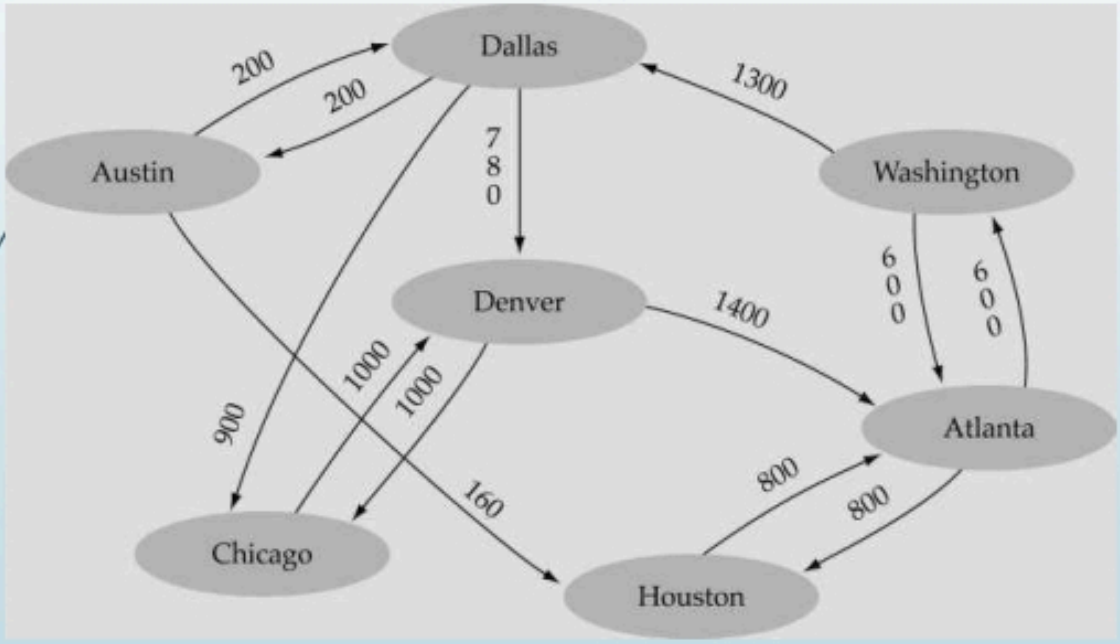
- Undirected edge has no orientation (no arrow head)
- Directed edge has an orientation (has an arrow head)
- Undirected graph – all edges are undirected
- Directed graph – all edges are directed

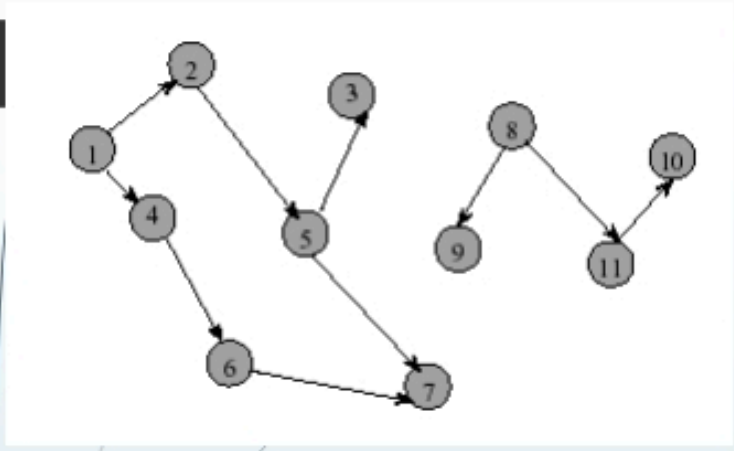
undirected edge

directed edge

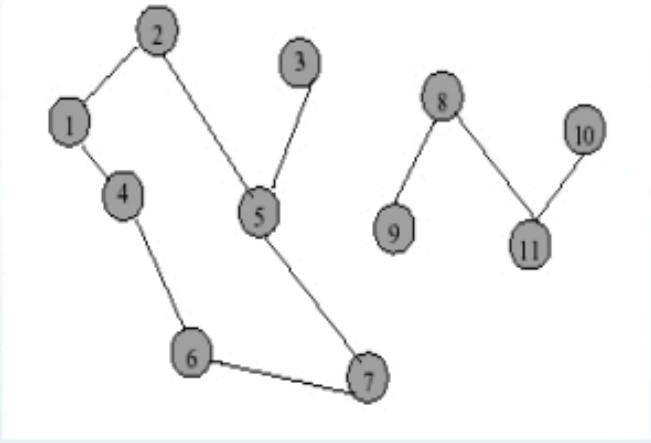
Weighted graph:

-a graph in which each edge carries a value





Directed graph



Undirected graph

Directed Graph

- Directed edge (i, j) is incident to vertex j and incident from vertex i
- Vertex i is adjacent to vertex j, and vertex j is adjacent from vertex i

Graph terminology

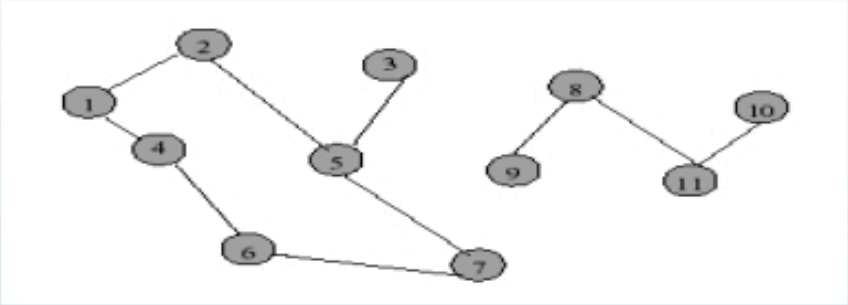
- Adjacent nodes: two nodes are adjacent if they are connected by an edge

- Path: a sequence of vertices th at connect two nodes in a graph

- A simple path is a path in which all vertices, except possibly in the first and last, are different.

- Complete graph: a graph in which every vertex is directly connected to every other vertex

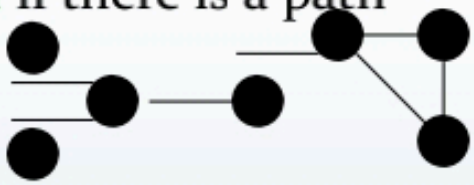
- A cycle is a simple path with the same start and end vertex.
- The **degree** of vertex *i* is the no. of edges incident on vertex *i*.



e.g., degree(2) = 2, degree(5) = 3, degree(3) = 1

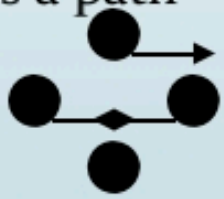
Continued…

Undirected graphs are *connected* if there is a path between any two vertices



Directed graphs are *strongly connected* if there is a path from any one vertex to any other

Directed graphs are *weakly connected* if there is a path between any two vertices, *ignoring direction*



A *complete* graph has an edge between every pair of vertices

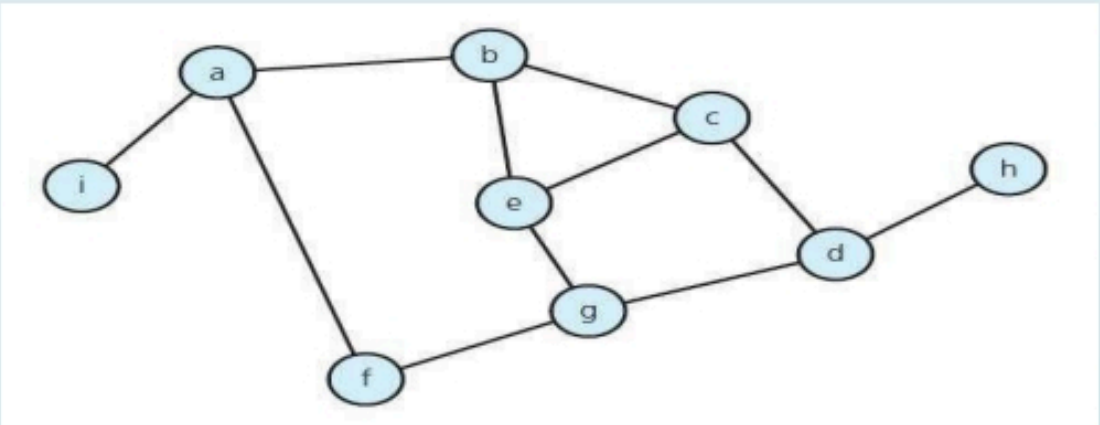
Continued…

- *Loops*: edges that connect a vertex to itself
- *Paths*: sequences of vertices p0, p1, … pm such that each adjacent pair of vertices are connected by an edge
- A simple path is a path in which all vertices, except possibly in the first and last, are different.
- *Multiple Edges*: two nodes may be connected by >1 edge
- *Simple Graphs*: have no loops and no multiple edges

Graph Properties

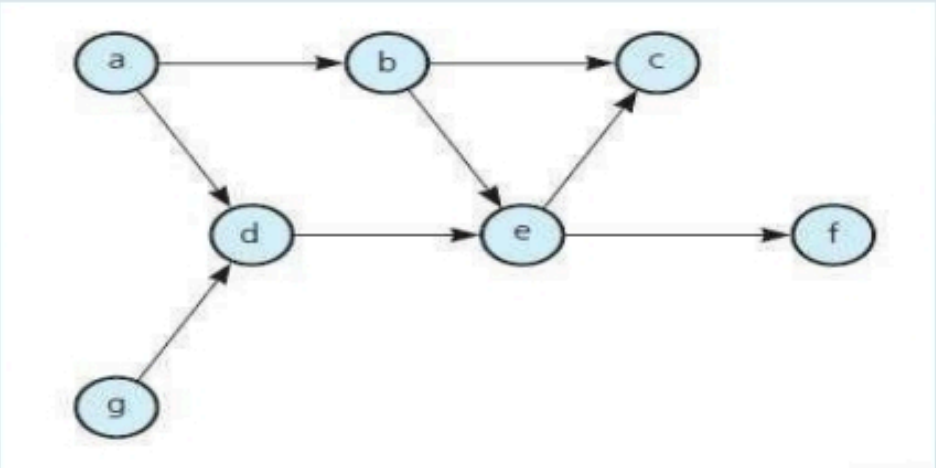
Number of Edges – Undirected Graph

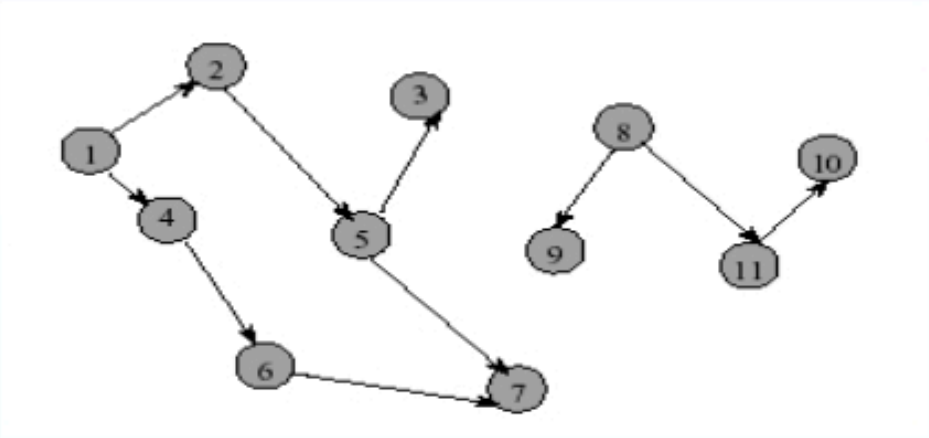
- The no. of possible pairs in an n vertex graph is n*(n-1)
- Since edge *(u,v)* is **the same** as edge *(v,u)*, the number of edges in an undirected graph is n*(n-1)/2.



Number of Edges - Directed Graph

- The no. of possible pairs in an n vertex graph is $n*(n-1)$
- Since edge *(u,v)* is **not the same** as edge *(v,u)*, the number of edges in a directed graph is $n*(n-1)$
- Thus, the number of edges in a directed graph is $\leq n*(n-1)$





- In-degree of vertex *i* is the number of edges incident to *i* (i.e., the number of incoming edges).

e.g., indegree(2) = 1, indegree(8) = 0

- Out-degree of vertex *i* is the number of edges incident from *i*

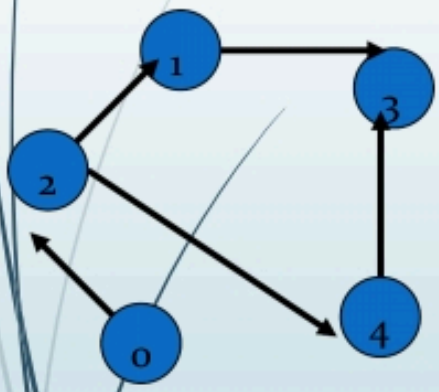
(i.e., the number of outgoing edges). e.g., outdegree(2) = 1, outdegree(8) = 2

Graph Representation

- For graphs to be computationally useful, they have to be conveniently represented in programs
- There are two computer representations of graphs:
  - Adjacency matrix representation
  - Adjacency lists representation

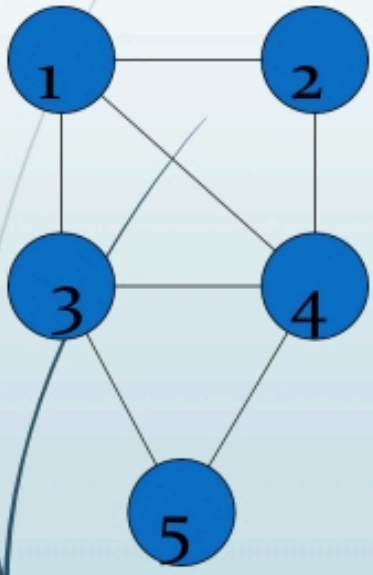
- *Adjacency Matrix*
  - A square grid of boolean values
  - If the graph contains N vertices, then the grid contains N rows and N columns
  - For two vertices numbered I and J, the element at row I and column J is true if there is an edge from I to J, otherwise false

Adjacency Matrix



	0	1	2	3	4
0	false	false	true	false	false
1	false	false	false	true	false
2	false	true	false	false	true
3	false	false	false	false	false
4	false	false	false	true	false

Adjacency Matrix



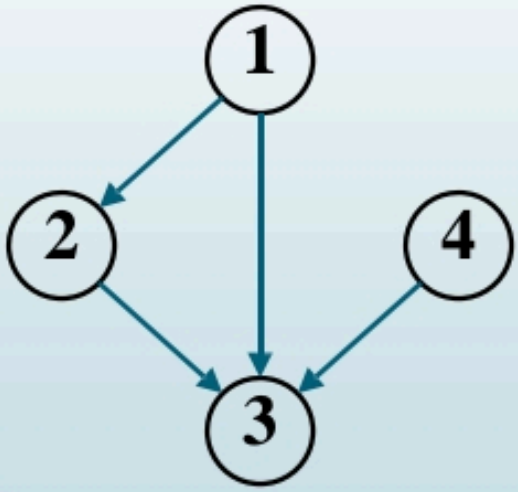
	1	2	3	4	5
1	0	1	1	1	0
2	1	0	0	1	0
3	1	0	0	1	1
4	1	1	1	0	1
5	0	0	1	1	0

Adjacency Lists Representation

- A graph of n nodes is represented by a one-dimensional array L of linked lists, where
  - L[i] is the linked list containing all the nodes adjacent from node i.
  - The nodes in the list L[i] are in no particular order

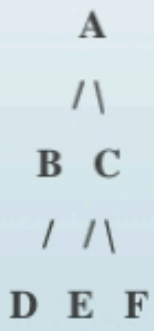
Graphs: Adjacency List

- Adjacency list: for each vertex $v \in V$, store a list of vertices adjacent to v
- Example:
  - Adj[1] = {2,3}
  - Adj[2] = {3}
  - Adj[3] = {}
  - Adj[4] = {3}
- Variation: can also keep a list of edges coming *into* vertex



Breadth First Search

- **Breadth First Search-** BFS stands for Breadth First Search is a vertex based technique for finding a shortest path in graph. It uses a Queue data structure which follows first in first out. In BFS, one vertex is selected at a time when it is visited and marked then its adjacent are visited and stored in the queue. It is slower than DFS.
- Ex-



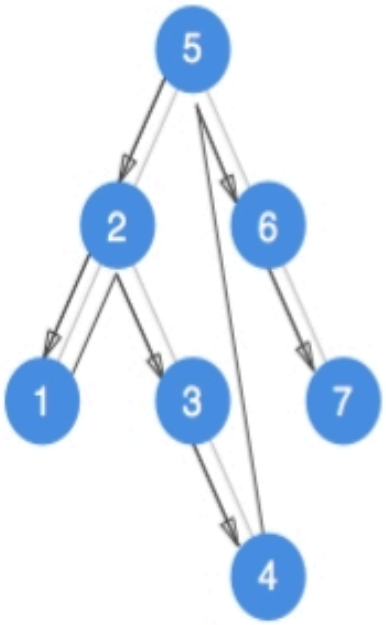
- Output is: A, B, C, D, E, F

Depth First Search

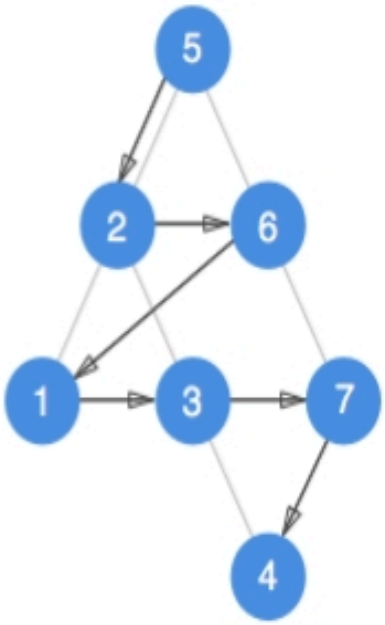
- **Depth First Search -** DFS stands for Depth First Search is a edge based technique. It uses the Stack data structure, performs two stages, first visited vertices are pushed into stack and second if there is no vertices then visited vertices are popped.
- Ex-

- Output is: A, B, D, C, E, F

Example



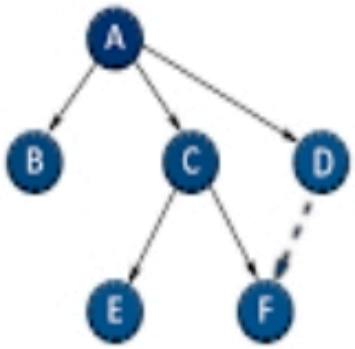
Depth-first traversal



Breadth-first traversal

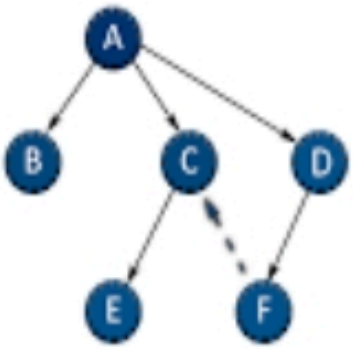
Example

BFS



A B C D E F

DFS



A D F C E B

Searching Algorithms

Searching Algorithm

- **Searching Algorithm** is an algorithm made up of a series of instructions that retrieves information stored within some data structure, or calculated in the search space of a problem domain.
- There are many sorting algorithms, such as:
  - **Linear Search**
  - **Binary Search**

Linear Search

- **Linear Search** is a method for finding a target value within a list. It sequentially checks each element of the list for the target value until a match is found or until all the elements have been searched.

Linear Search

Algorithm:

- **Step1:** Start from theleftmost element of array and one by one compare x with each element of array.
- **Step2:** If x matches with an element, return the index.
- **Step3:** If x doesn't match with any of elements, return -1.

Linear Search

- Assume the following Array:
- Search for 9

8	12	5	9	2

Linear Search

- Compare
- X= 9

Linear Search

- Compare
- X= 9

Linear Search

- Compare
- X= 9

Linear Search

- Compare
- X= 9

Linear Search

- Found at index = 3

Binary Search

- **Binary Search** is the most popular Search algorithm. It is efficient and also one of the most commonly used techniques that is used to solve problems. Binary search use sorted array by repeatedly dividing the search interval in half.

Binary Search

Algorithm:

- **Step1:** Compare x with the middle element.
- **Step2:** If x matches with middle element, we return the mid index.
- **Step3:** Else If x is greater than the mid element, search on right half.
- **Step4:** Else If x is smaller than the mid element. search on left half.

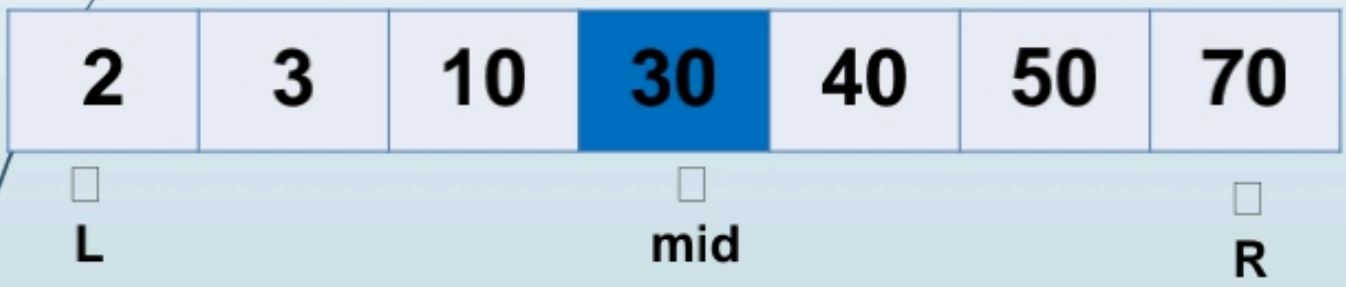
Binary Search

- Assume the following Array:
- Search for 40

2	3	10	30	40	50	70

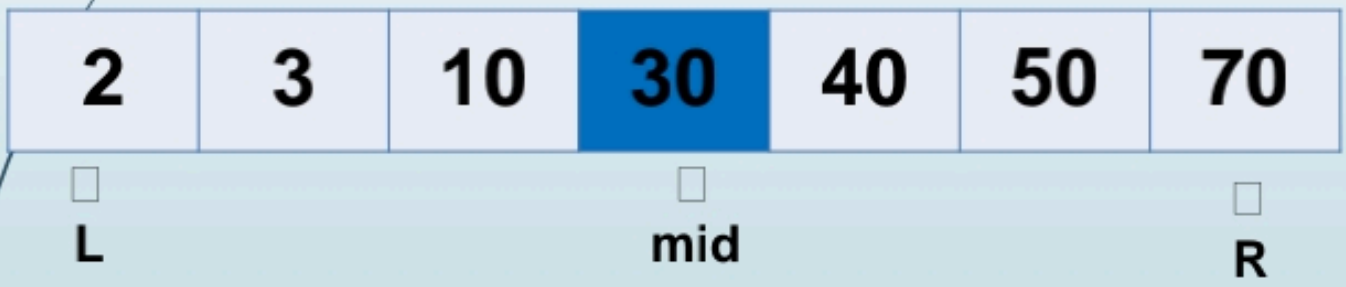
Binary Search

- Compare
- X= 40



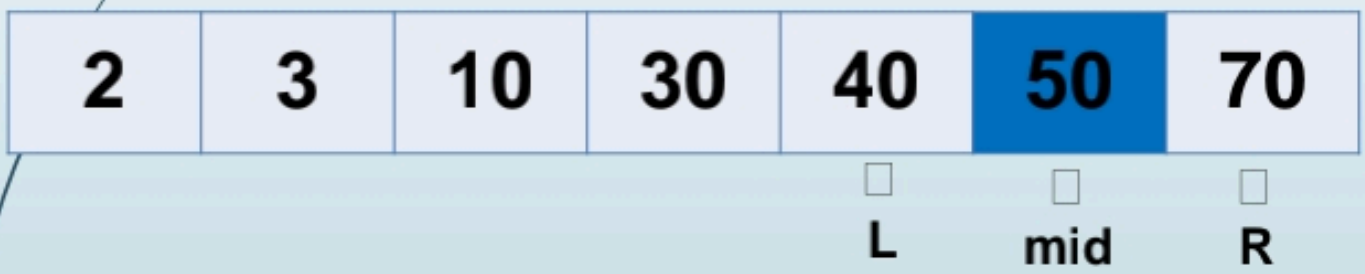
Binary Search

- Compare
- X= 40



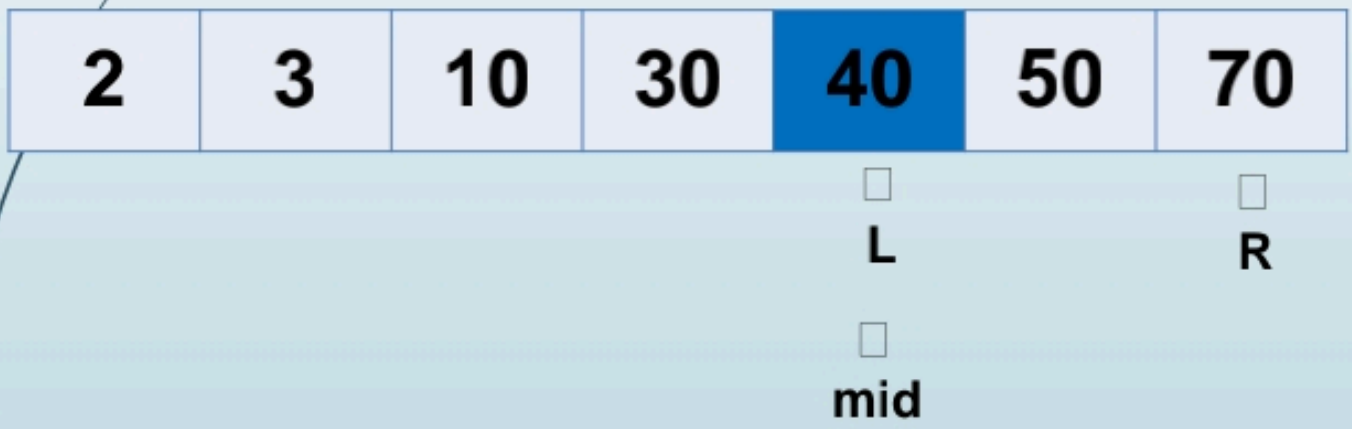
Binary Search

- Compare
- X= 40



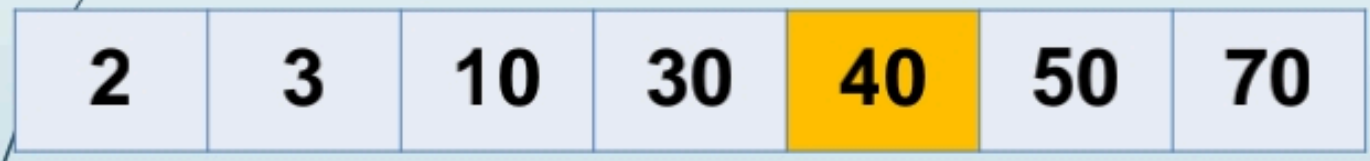
Binary Search

- Compare
- X= 40



Binary Search

- X= 40
- Found at index = 4



Sorting

- The arrangement of data in a preferred order is called sorting in the data structure. By sorting data, it is easier to search through it quickly and easily. The simplest example of sorting is a dictionary. Before the era of the Internet, when you wanted to look up a word in a dictionary, you would do so in alphabetical order. This made it easy.

Merge Sort

- **Merge Sort** is one of the most popular sorting algorithms that is based on the principle of Divide and Conquer Algorithm.

- Here, a problem is divided into multiple sub-problems. Each sub-problem is solved individually. Finally, sub-problems are combined to form the final solution.

Sorting Categories

- There are two different categories in sorting:
- **Internal sorting:** If the input data is such that it can be adjusted in the main memory at once, it is called internal sorting.
- **External sorting:** If the input data is such that it cannot be adjusted in the memory entirely at once, it needs to be stored in a hard disk, floppy disk, or any other storage device. This is called external sorting.

Divide and Conquer Strategy

- Using the Divide and Conquer technique, we divide a problem into sub problems. When the solution to each sub problem is ready, we 'combine' the results from the sub problems to solve the main problem.
- Suppose we had to sort an array A. A sub problem would be to sort a sub-section of this array starting at index p and ending at index r, denoted as A[p..r].

Divide and Conquer Strategy

- **Divide -** If q is the half-way point between p and r, then we can split the subarray A[p..r] into two arrays A[p..q] and A[q+1, r].

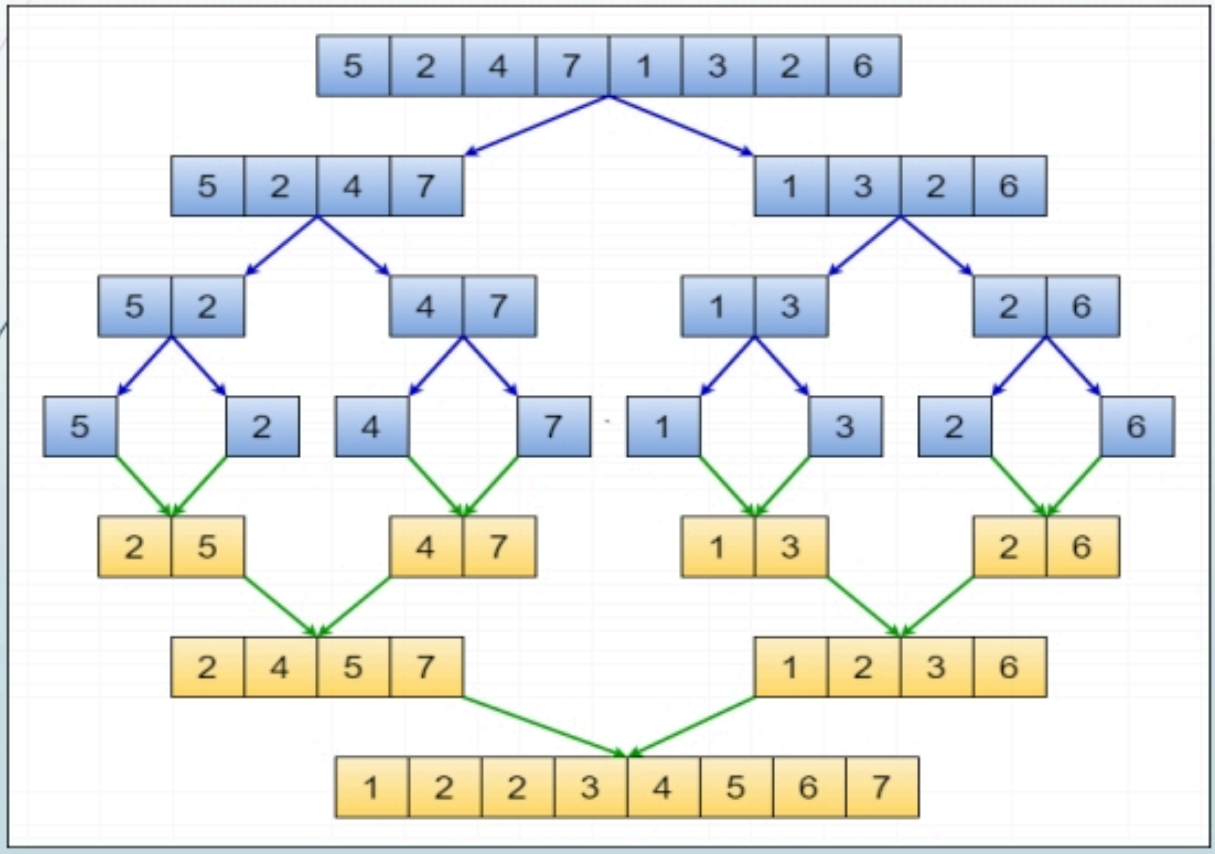
- **Conquer -** In the conquer step, we try to sort both the subarrays A[p..q] and A[q+1, r]. If we haven't yet reached the base case, we again divide both these subarrays and try to sort them.

- **Combine -** When the conquer step reaches the base step and we get two sorted subarrays A[p..q] and A[q+1, r] for array A[p..r], we combine the results by creating a sorted array A[p..r] from two sorted subarrays A[p..q] and A[q+1, r].

Merge Sort

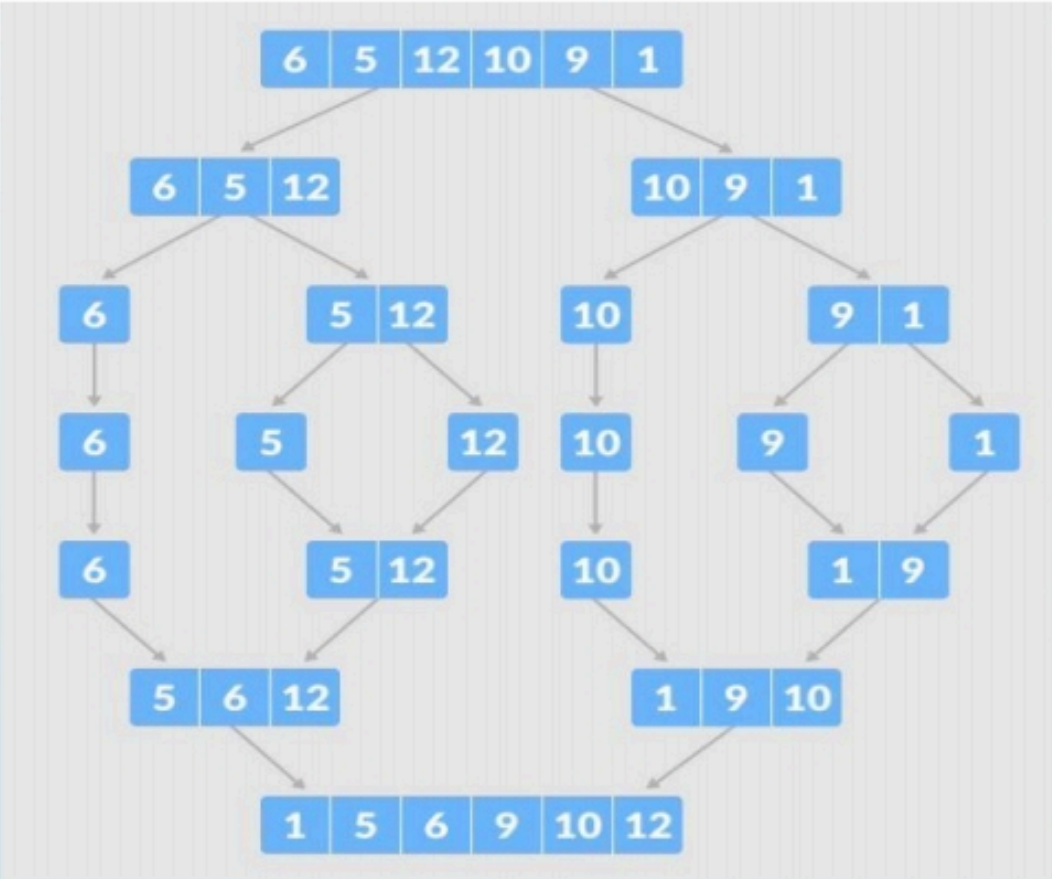
6 5 3 1 8 7 2 4

Merge Sort

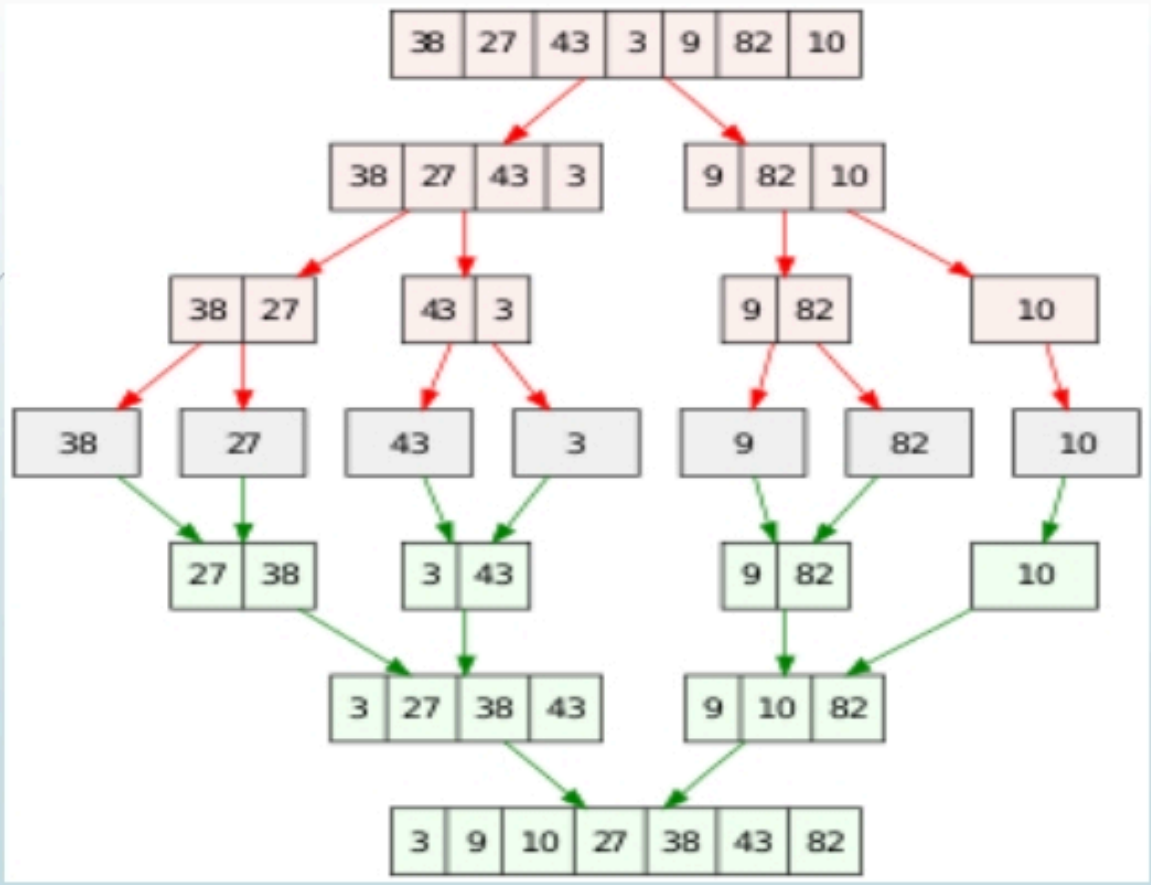

5 2 4 7 1 3 2 6
5 2 4 7
1 3 2 6
5 2
4 7
1 3
2 6
5
2
4
7
1
3
2
6
2 5
4 7
1 3
2 6
2 4 5 7
1 2 3 6
1 2 2 3 4 5 6 7

Questions

Merge Sort


6 5 12 10 9 1
6 5 12
10 9 1
6
5 12
10
9 1
6
5
12
10
9
1
6
5 12
10
1 9
5 6 12
1 9 10
1 5 6 9 10 12

Merge Sort

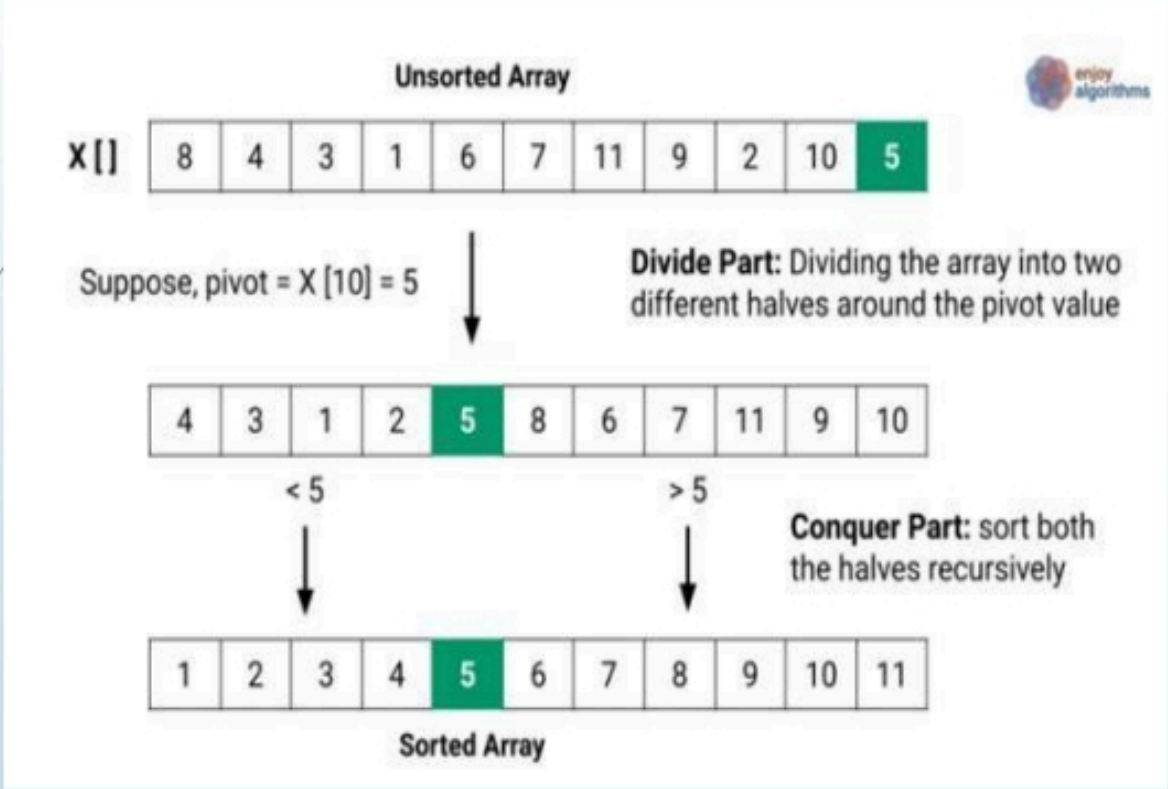

38 27 43 3 9 82 10
38 27 43 3
9 82 10
38 27
43 3
9 82
10
38
27
43
3
9
82
10
27 38
3 43
9 82
10
3 27 38 43
9 10 82
3 9 10 27 38 43 82

Quicksort

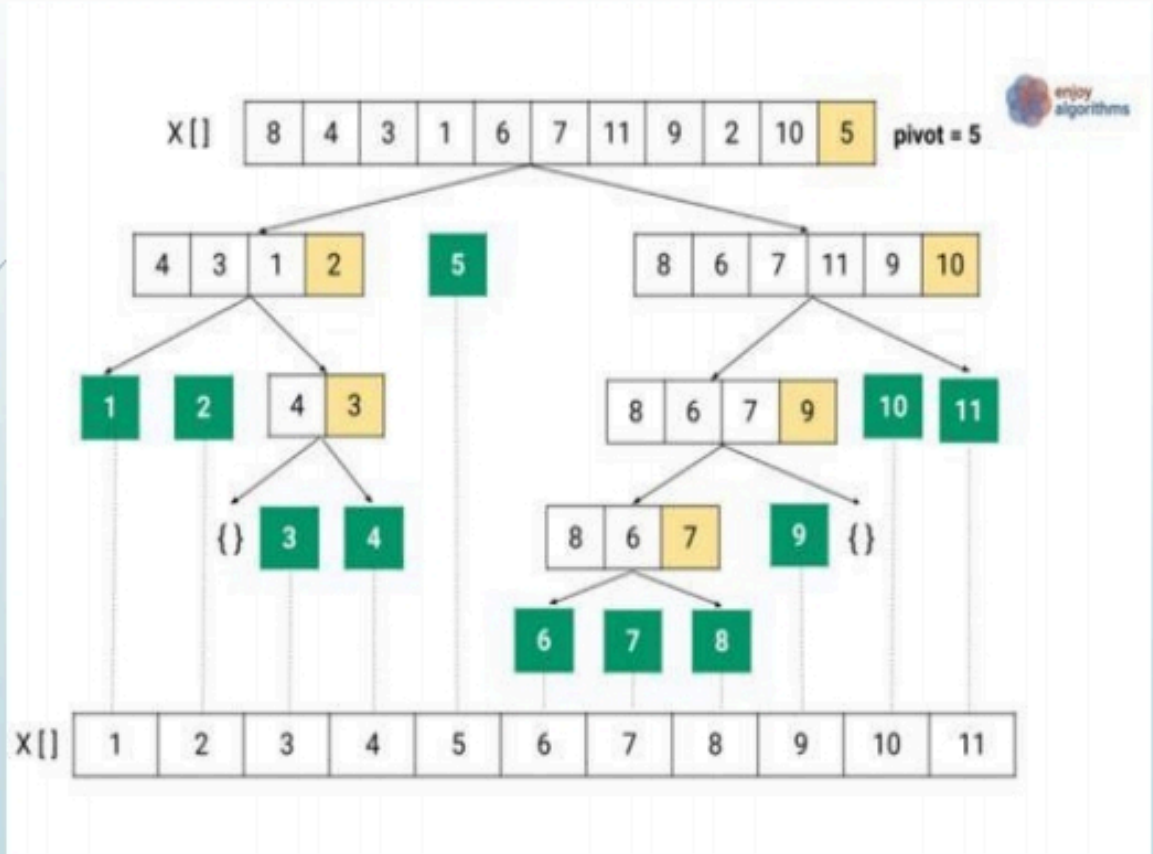
- **Quicksort** is a divide-and-conquer algorithm. It works by selecting a 'pivot' element from the array and partitioning the other elements into two sub-arrays, according to whether they are less than or greater than the pivot.

- For this reason, it is sometimes called partition-exchange sort.The sub-arrays are then sorted recursively. This can be done in-place, requiring small additional amounts of memory to perform the sorting.

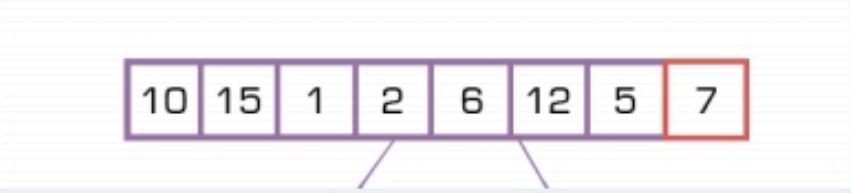
Quicksort

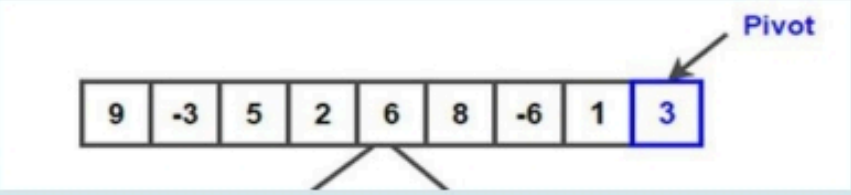

Unsorted Array
X[]
8 4 3 1 6 7 11 9 2 10 5
Suppose, pivot = X [10] = 5
Divide Part: Dividing the array into two different halves around the pivot value
4 3 1 2 5 8 6 7 11 9 10
< 5
> 5
Conquer Part: sort both the halves recursively
1 2 3 4 5 6 7 8 9 10 11
Sorted Array

Quicksort

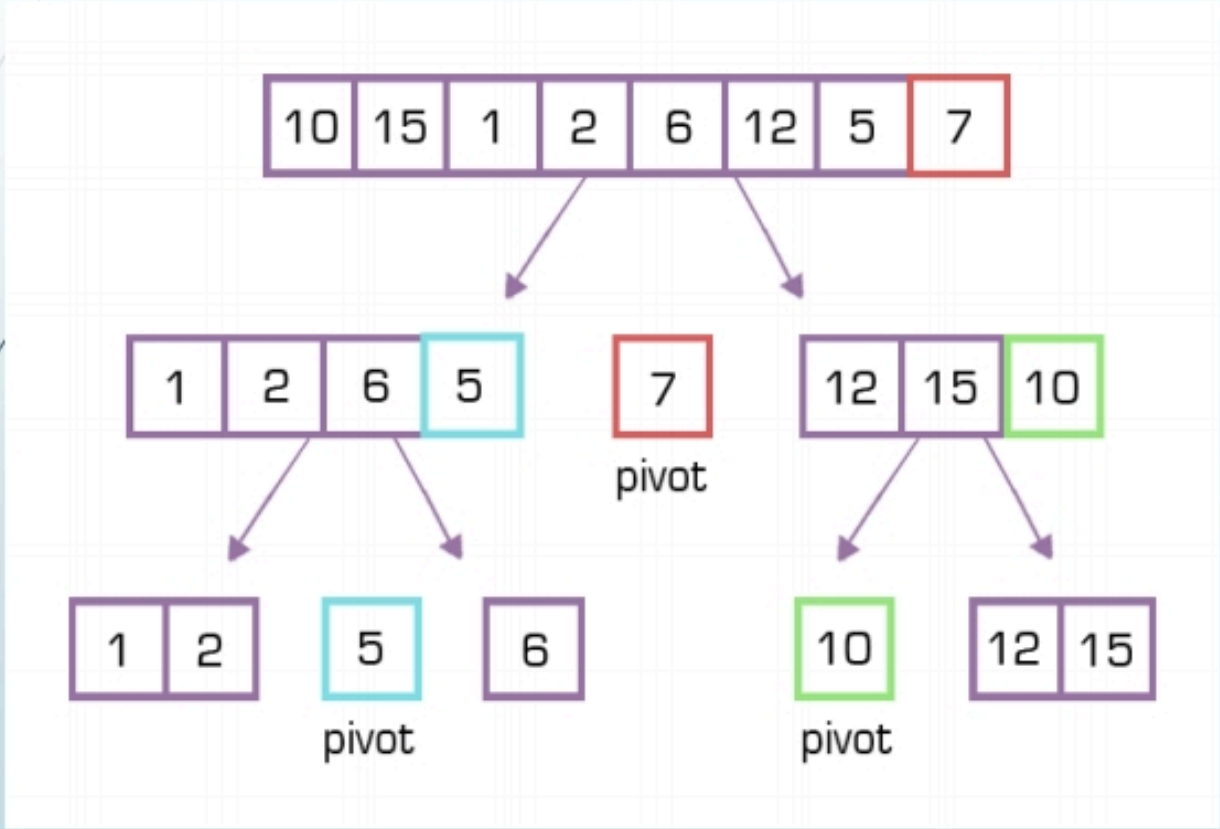

X[]
8 4 3 1 6 7 11 9 2 10 5
pivot = 5
4 3 1 2
5
8 6 7 11 9 10
1
2
4 3
8 6 7 9
10
11
{}
3
4
8 6 7
9
{}
6
7
8
X[]
1 2 3 4 5 6 7 8 9 10 11

Questions





Quicksort



Quicksort