CIS 351-Data Structure-Tree
April 7, 2020

Dr. Farzana Rahman

Syracuse University

Drop grades

- 2 HW
- 2 labs
- 1 Quiz
- 3 in-class activities

Non-Linear Structure

- So far we have seen linear structures linear:
  - lists, vectors, arrays, stacks, queues, etc
- Non-linear structure:
  - trees
  - probably the most fundamental structure in computing
  - hierarchical structure
  - Terminology: from family trees (genealogy)

Application

- Applications of trees
  - class hierarchy in Java
  - file system
  - storing hierarchies in organizations

Tree

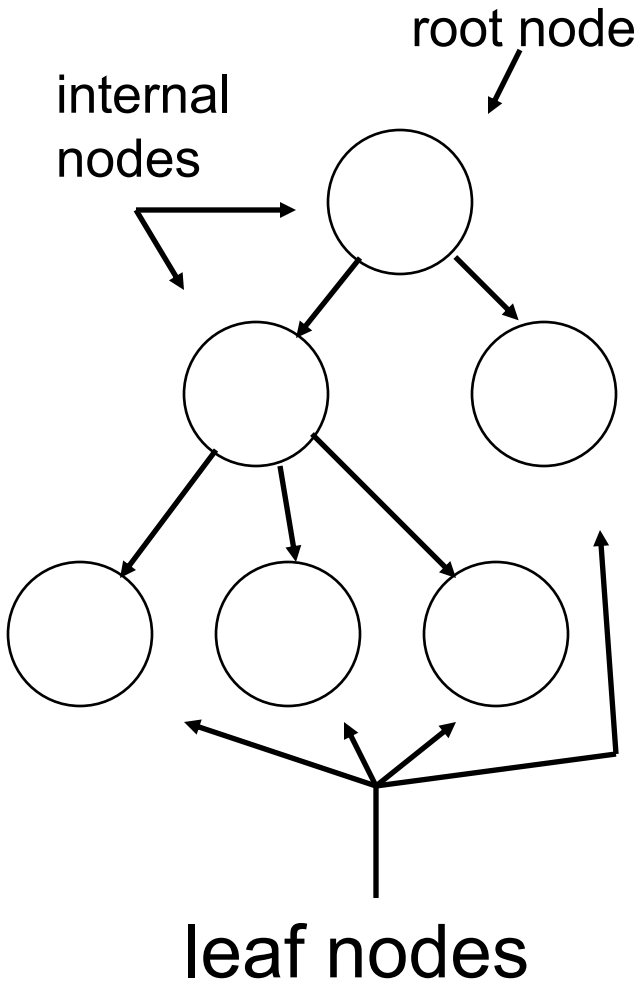
- Tree defined recursively
- A tree is a collection of nodes.
- The collection can be empty; otherwise, a tree consists of a distinguished node r, called the **root**, and zero or more non-empty (sub) trees $\mathbf{T_1}$, $\mathbf{T_2}$, …, $\mathbf{T_k}$ each of whose roots are connected by a directed edge from **r**.
- A tree is a collection of **N** nodes, one of which is the **root** and **N-1** edges.

Tree terminology

- The root of each subtree is said to be a **child** of **r** and r is said to be the **parent** of each subtree root.
- **Leaves**: nodes with no children (also known as external nodes)
- **Internal Nodes**: nodes with children
- **Siblings**: nodes with the same parent

Terminology

- The Tree ADT has
  - one entry point, the ***root***
  - Each node is either a ***leaf*** or an *internal node*
  - An internal node has 1 or more ***children***, nodes that can be reached directly from that internal node.
  - The internal node is said to be the ***parent*** of its child nodes

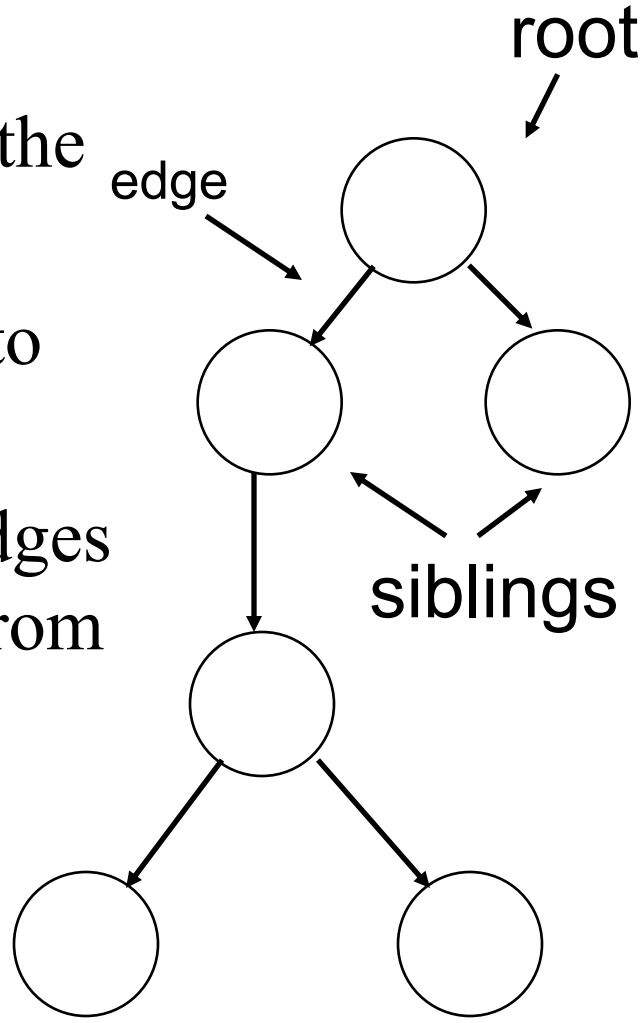


Tree terminology (continued)

- A **path** from node n_1 to n_k is defined as a sequence of nodes n_1, n_2, …, n_k such that n_i is the parent of n_{i+1} for $1 <= i <= k$.
- The length of this path is the number of edges on the path namely k-1.
- The length of the path from a node to itself is 0.
- There is exactly one path from the root to each node.

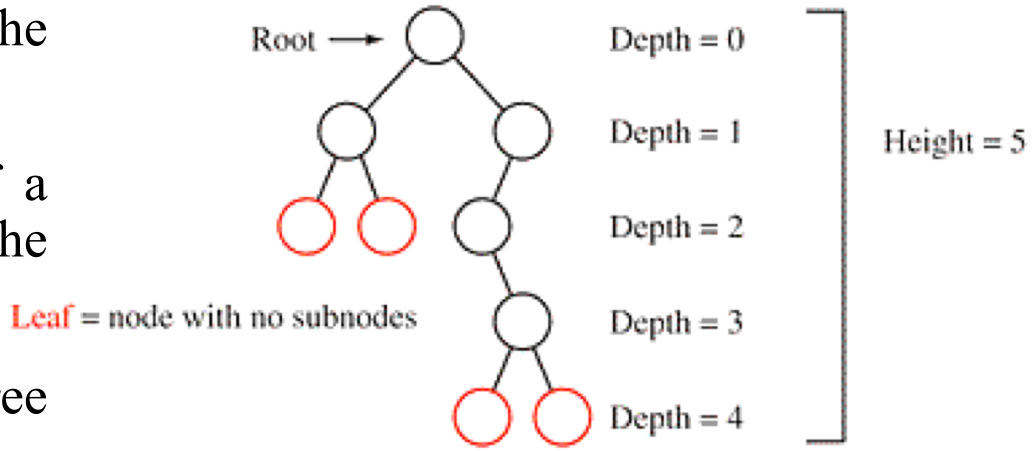
Definitions

- **siblings:** two nodes that have the same parent
- **edge:** the link from one node to another
- **path length:** the number of edges that must be traversed to get from one node to another



Tree terminology (continued)

- **Depth (of node):** The depth of a node is the number of edges from the root to the node.
- **Height (of node):** The height of a node is the number of edges from the node to the deepest leaf.
- **Height (of tree):** The height of a tree is a height of the root.



Depth of the tree = Height of the tree - 1

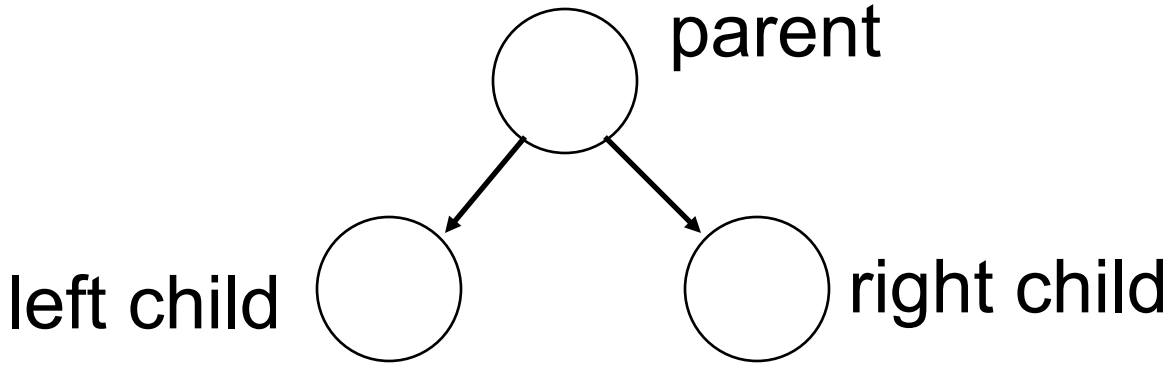
- **Descendants:** any nodes that can be reached via 1 or more edges from this node
- **Ancestors:** any nodes for which this node is a descendant

Binary tree

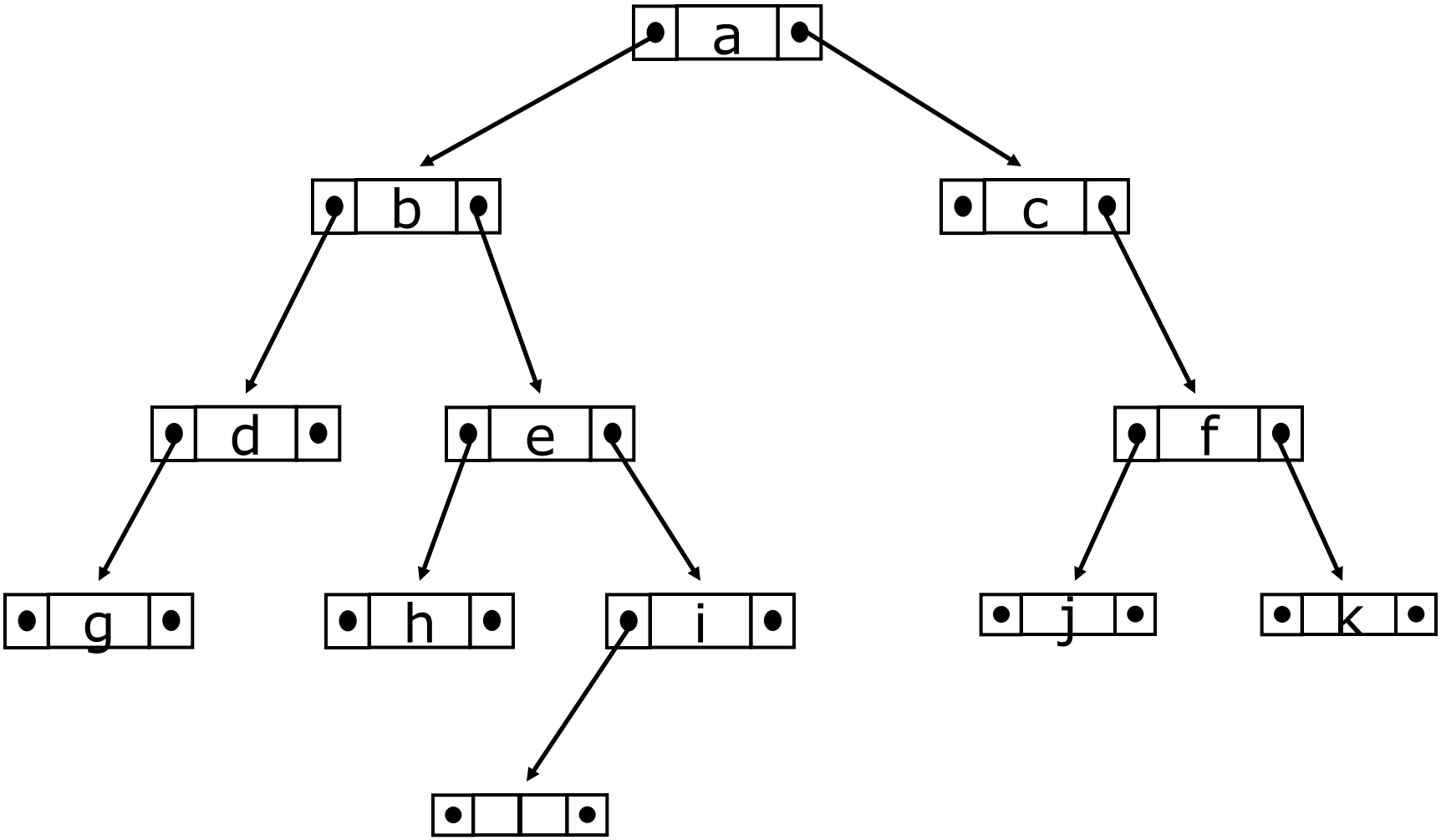
- A binary tree is composed of zero or more nodes
- Each node contains:
  - A value (some sort of data item)
  - A reference or pointer to a left child (may be null), and
  - A reference or pointer to a right child (may be null)
- A binary tree may be *empty* (contain no nodes)
- If not empty, a binary tree has a root node
  - Every node in the binary tree is reachable from the root node by a *unique* path
- A node with neither a left child nor a right child is called a leaf
  - In some binary trees, only the leaves contain a value

Binary Trees

- There are many variations on trees but we will start with **binary trees**
- **binary tree:** each node has at most **two** children
  - the possible children are usually referred to as the **left child** and the **right child**



Picture of a binary tree



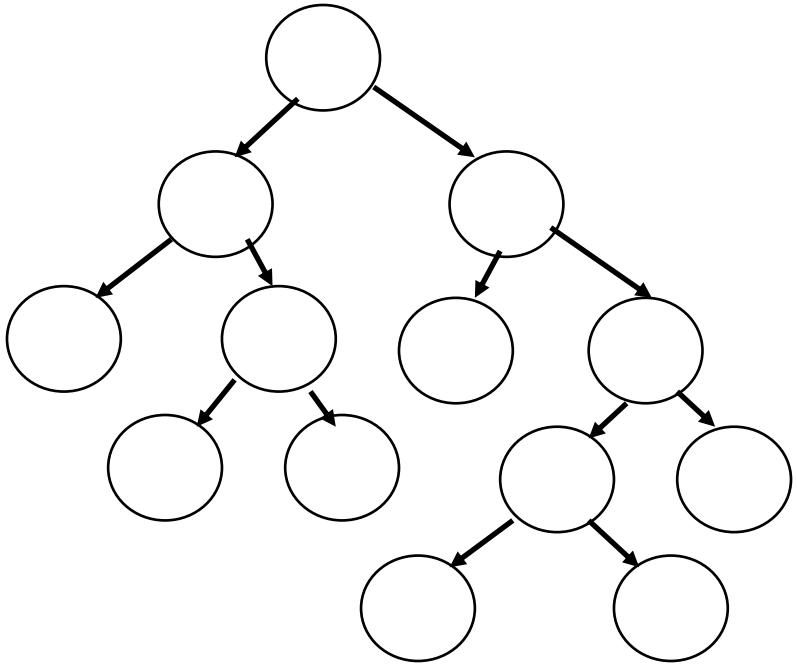
Recursive definition

Recursive definition:

- a binary tree is empty;
- or it consists of
    - a node (the root) that stores an element
    - a binary tree, called the left subtree of T
    - a binary tree, called the right subtree of T

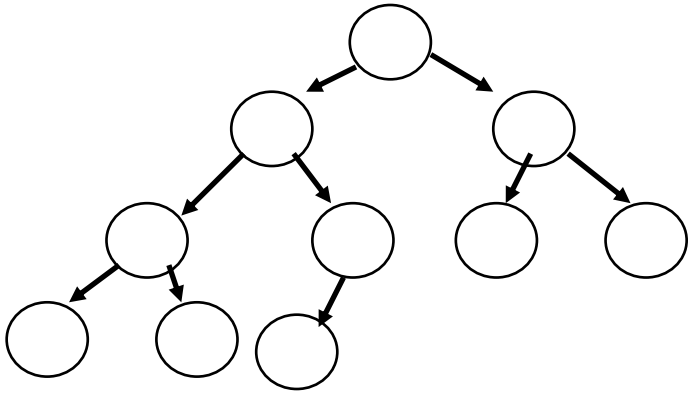
Binary tree property: Full

- **full binary tree:** a binary tree is which each node has exactly 2 or 0 children



Binary tree property: Complete

- **complete binary tree:** a binary tree T with n levels is complete if all levels except possibly the last are completely full, and the last level has all its nodes to the left side.



Where would the next node go
to maintain a complete tree?

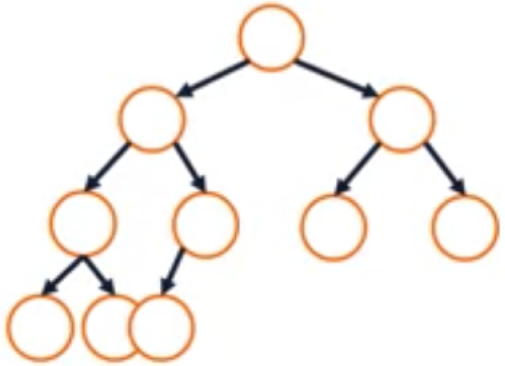
Binary tree property: Perfect

- **perfect binary tree:** a binary tree with all leaf nodes at the same depth. All internal nodes have exactly two children.
- a perfect binary tree has the maximum number of nodes for a given height
- a perfect binary tree has $(2^{(n+1)} - 1)$ nodes where n is the height of the tree
  - height = 0 -> 1 node
  - height = 1 -> 3 nodes
  - height = 2 -> 7 nodes
  - height = 3 -> 15 nodes

Puzzle:

Is a **full** tree **complete**?

Is a **complete** tree **full**?



Question

- What is the maximum height of a full binary tree with 11 nodes?

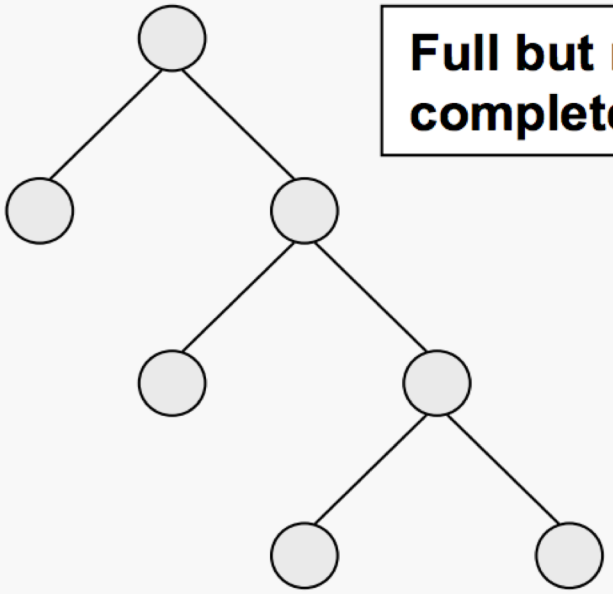
A. 3

B. 5

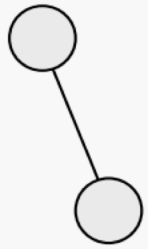
C. 7

D. 11

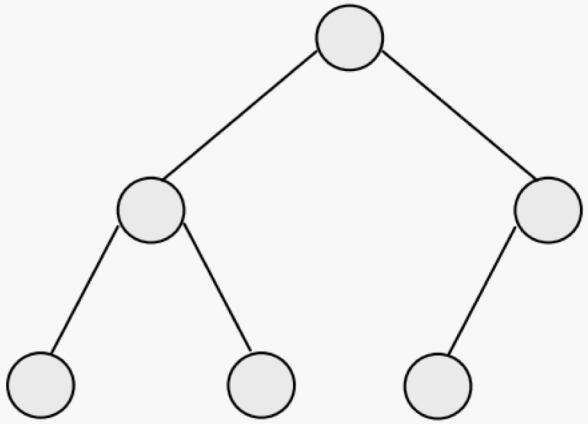
E. Not possible to have full binary tree with 11 nodes



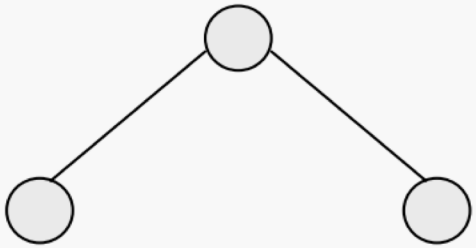
Full but not complete.



Neither complete nor full.

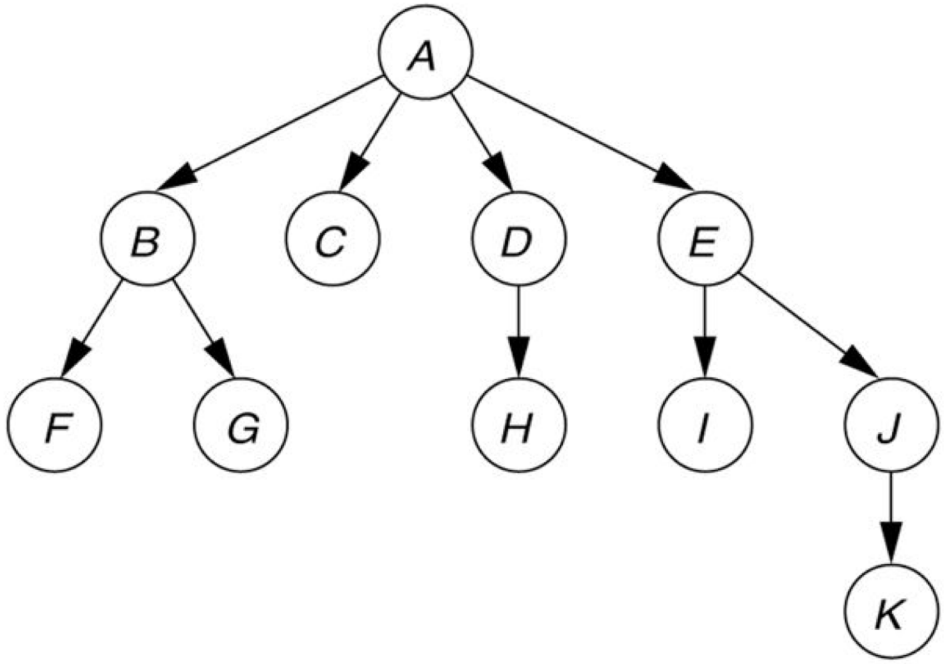


Complete but not full.



Full and complete.

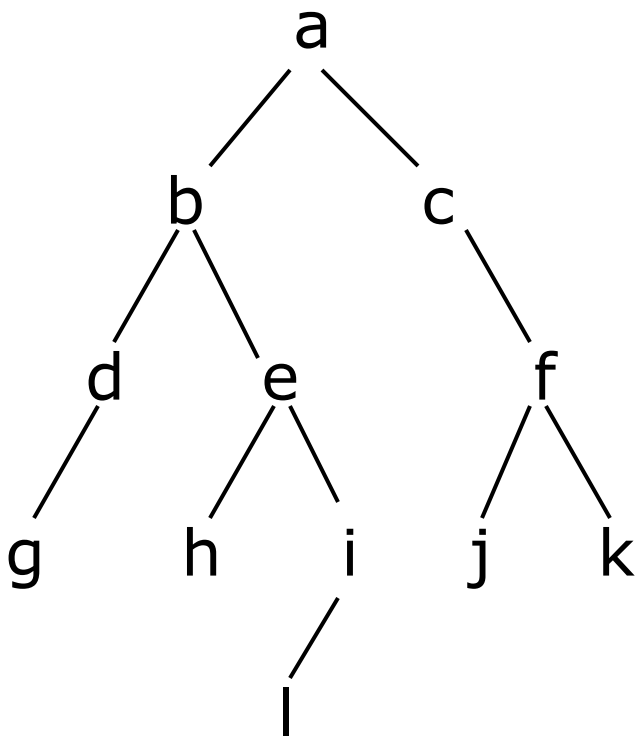
Figure 1
A tree, with height and depth information



Node	Height	Depth
A	3	0
B	1	1
C	0	1
D	1	1
E	2	1
F	0	2
G	0	2
H	0	2
I	0	2
J	1	2
K	0	3

1. The depth of a **node** is the number of edges from the root to the **node**.
2. The **height of a node** is the number of edges from the **node** to the deepest leaf.
3. The **height** of a **tree** is a **height** of the root.

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

ADT

- Whatever the implementation of a tree is, its interface is the following
  - root()
  - size()
  - isEmpty()
  - parent(v)
  - children(v)
  - isInternal(v)
  - isExternal(v)
  - isRoot()

A Binary Node class

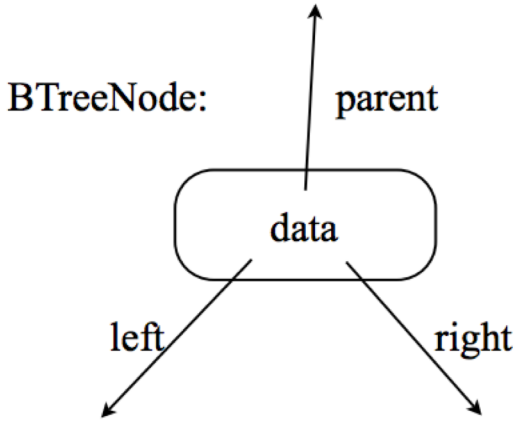
```
public class Bnode <E> {
      private E myData;
      private Bnode<E> myLeft;
      private Bnode<E> myRight;

      public BNode();
      public BNode(E data, Bnode<E> left, Bnode<E> right)
      public E getData()
      public Bnode<E> getLeft()
      public Bnode<E> getRight()

      public void setData(E data)
      public void setLeft(Bnode<E> left)
      public void setRight(Bnode<E> right)
}
```

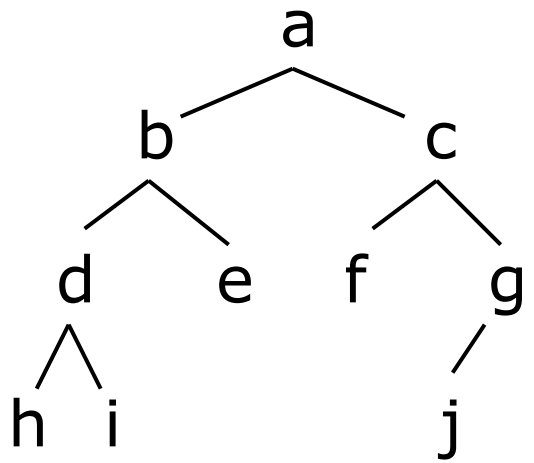
Binary tree implementation

- use a linked-list structure; each node points to its left and right children ; the tree class stores the root node and the size of the tree
- implement the following functions:
  - left(v)
  - right(v)
  - hasLeft(v)
  - hasRight(v)
  - isInternal(v)
  - is External(v)
  - isRoot(v)
  - size()
  - isEmpty()

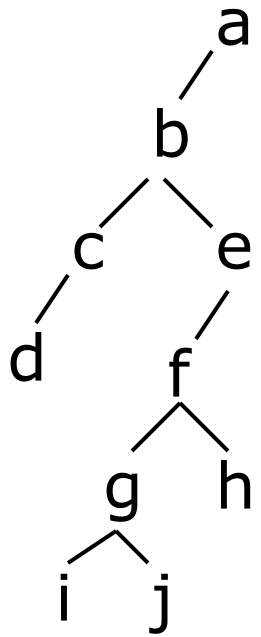


- also • insertLeft(v,e) , insertRight(v,e) , remove(e) , addRoot(e)

Balance



A **balanced** binary tree



An **unbalanced** binary tree

- A binary tree is balanced if every level above the lowest is "full" (contains 2^n nodes)
- In most applications, a reasonably balanced binary tree is desirable

Binary Tree traversals

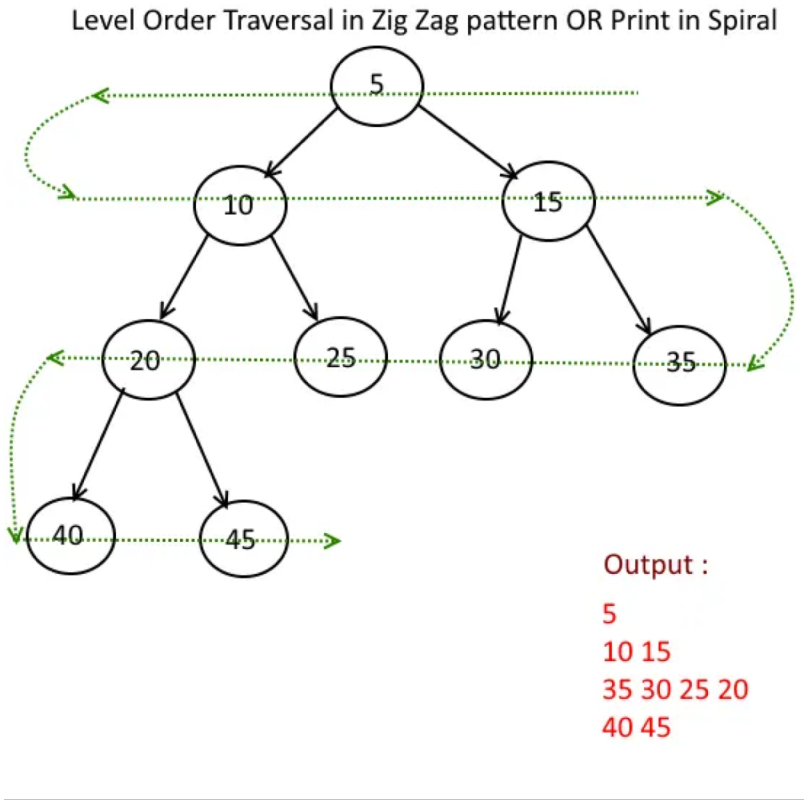
- A binary tree is defined recursively: it consists of a root, a left subtree, and a right subtree
- To traverse (or walk) the binary tree is to visit each node in the binary tree exactly once
- Tree traversals are naturally **recursive**
- Since a binary tree has three "parts," there are six possible ways to traverse the binary tree:
  - root, left, right
  - left, root, right
  - left, right, root
  - root, right, left
  - right, root, left
  - right, left, root

Traversal

- A traversal is a process that visits all the nodes in the tree.
- Since a tree is nonlinear structure, there is no unique traversal.
- We will consider several traversal algorithms with we group in the following two kinds
  - **Depth-first traversal**
  - **Breadth-first traversal**

Breadth First Traversals

- There is only one kind of **breadth-first traversal**
  - **The level order traversal**.
- **level order traversal:** starting from the root of a tree, process all nodes at the same depth from left to right, then proceed to the nodes at the next depth.

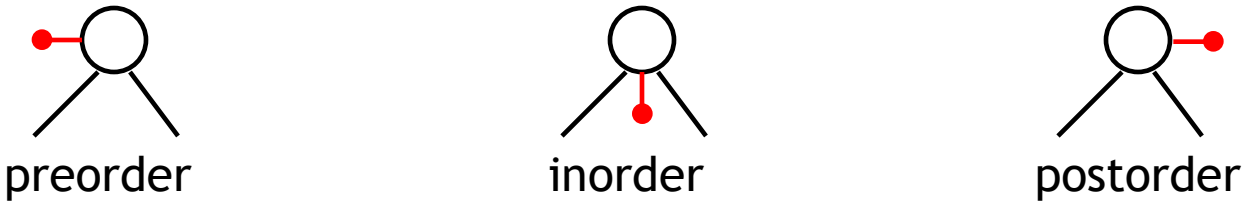


Depth First Traversals

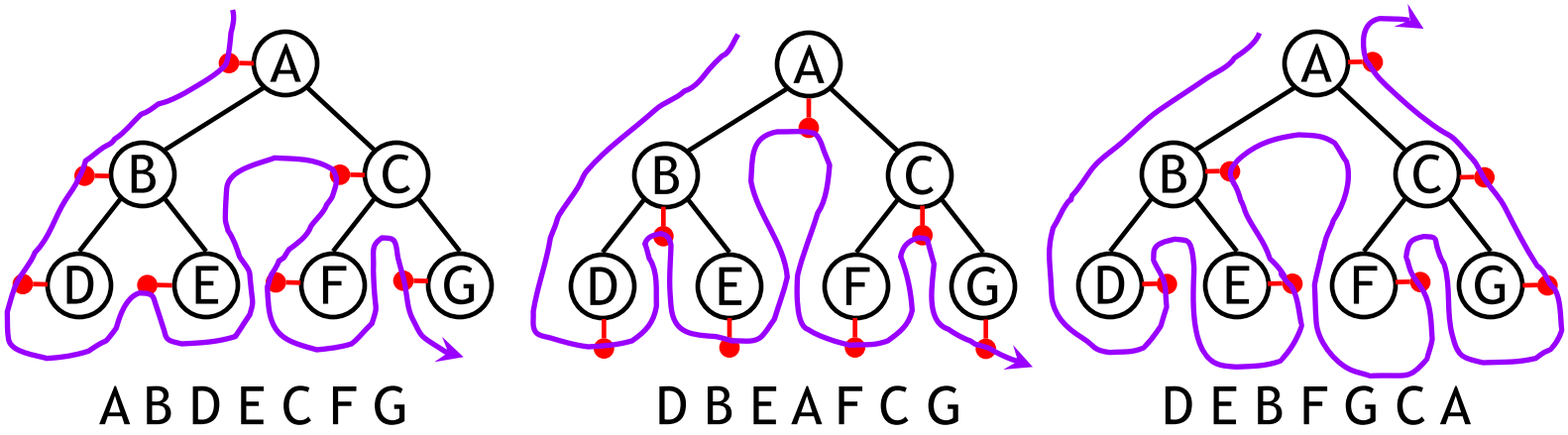
- There are **three** different types of depth-first traversals
  - **preorder traversal:** process the root, then process all sub trees (left to right)
  - **in order traversal:** process the left sub tree, process the root, process the right sub tree
  - **post order traversal:** process the left sub tree, process the right sub tree, then process the root

Tree traversals using “flags”

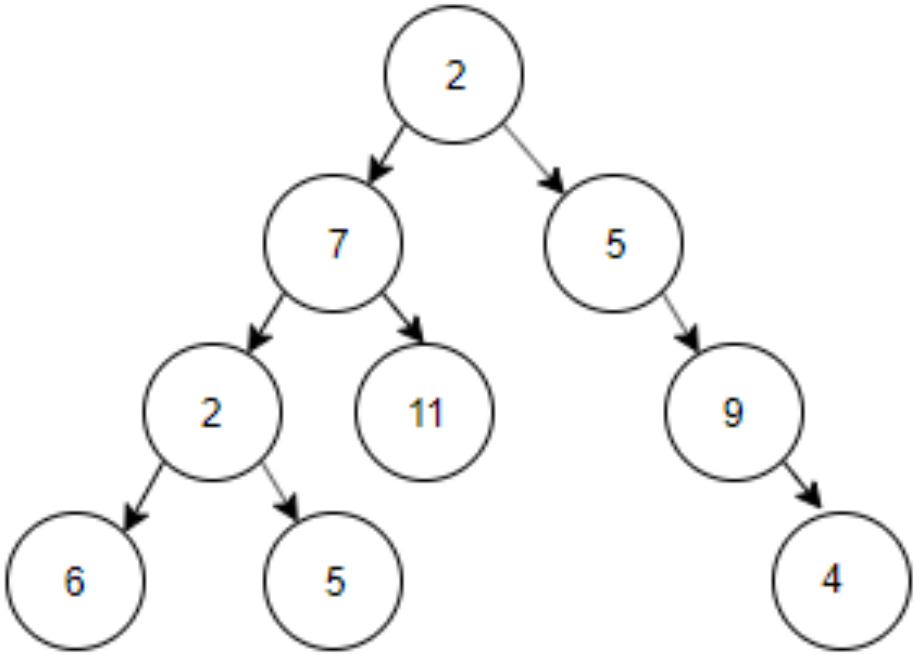
- The order in which the nodes are visited during a tree traversal can be easily determined by imagining there is a “flag” attached to each node, as follows:



- To traverse the tree, collect the flags:



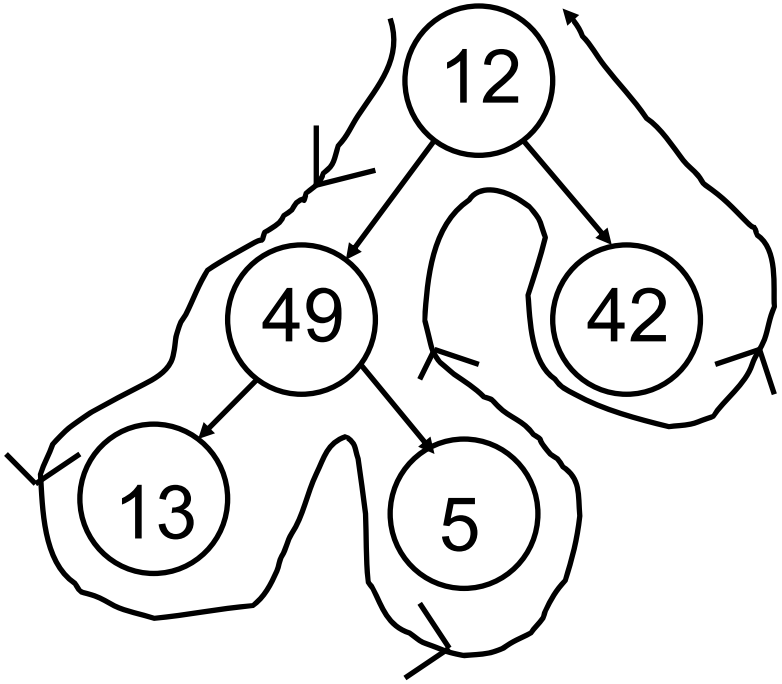
Answer this



- Inorder:
- Preorder:
- Postorder:

Results of Traversals

- To determine the results of a traversal on a given tree draw a path around the tree.
  - start on the **left side** of the root and trace around the tree. The path should stay close to the tree.

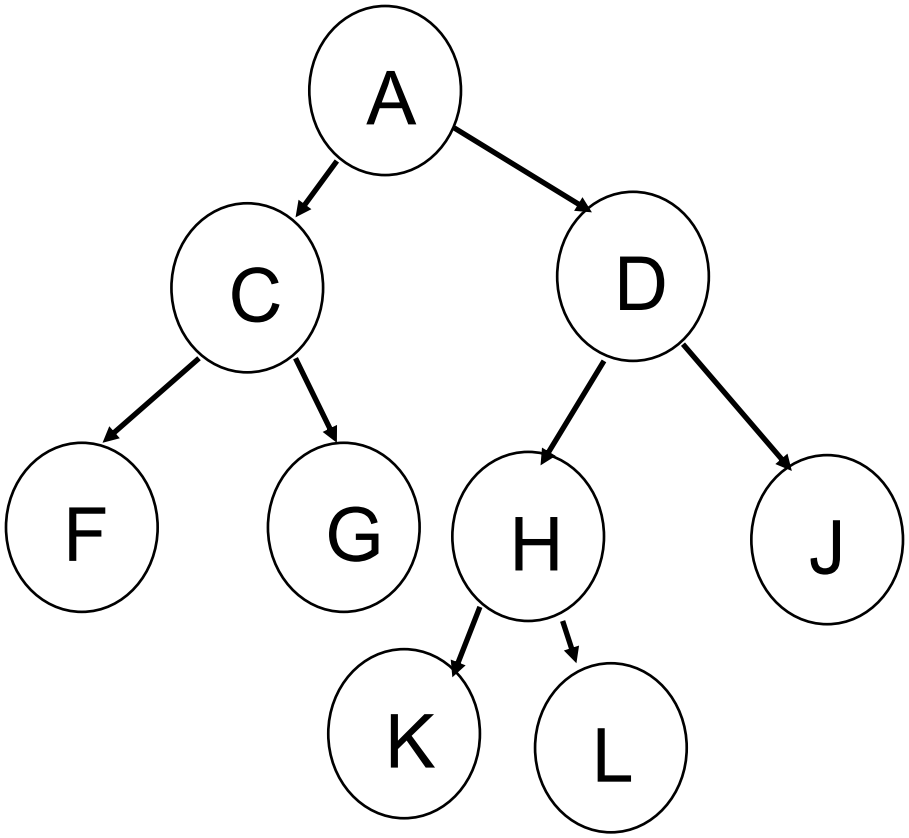


pre order: process when pass down left side of node
12 49 13 5 42

in order: process when pass underneath node
13 49 5 12 42

post order: process when pass up right side of node
13 5 49 42 12

Question: Tree Traversals



What is a the result of a **post order** traversal of the tree to the left?

A. F C G A K H L D J
B. F G C K L H J D A
C. A C F G D H K L J
D. A C D F G H J K L
E. L K J H G F D C A

Preorder traversal

- In preorder, the root is visited *first*
- Here's a preorder traversal to print out all the elements in the binary tree:

```
public void preorderPrint(BinaryTree bt) {
    if (bt == null) return;
    System.out.println(bt.value);
    preorderPrint(bt.leftChild);
    preorderPrint(bt.rightChild);
}
```

Inorder traversal

- In inorder, the root is visited *in the middle*
- Here's an inorder traversal to print out all the elements in the binary tree:

```
public void inorderPrint(BinaryTree bt) {
    if (bt == null) return;
    inorderPrint(bt.leftChild);
    System.out.println(bt.value);
    inorderPrint(bt.rightChild);
}
```

Postorder traversal

- In postorder, the root is visited *last*
- Here's a postorder traversal to print out all the elements in the binary tree:

```
public void postorderPrint(BinaryTree bt) {
    if (bt == null) return;
    postorderPrint(bt.leftChild);
    postorderPrint(bt.rightChild);
    System.out.println(bt.value);
}
```

Copying a binary tree

- In postorder, the root is visited *last*
- Here's a postorder traversal to make a complete copy of a given binary tree:

```
public BinaryTree copyTree(BinaryTree bt) {
    if (bt == null) return null;
    BinaryTree left = copyTree(bt.leftChild);
    BinaryTree right = copyTree(bt.rightChild);
    return new BinaryTree(bt.value, left, right);
}
```

Other traversals

- The other traversals are the reverse of these three standard ones
  - That is, the right subtree is traversed before the left subtree is traversed
- **Reverse preorder:** root, right subtree, left subtree
- **Reverse inorder:** right subtree, root, left subtree
- **Reverse postorder:** right subtree, left subtree, root