Basics of Trees

EECS 233

Flashback: Phone Book

- Operations:
  - *search* for an item (and possibly delete it)
  - *insert* a new item
- If we use a list (in previous lectures) to implement a data dictionary, efficiency = O(n).

data structure	***searching for an item***	***inserting an item***
a list implemented using an array (sorted)	O(log n) using binary search	O(n) because we need to shift items over
a list implemented using a linked list	O(n) using linear search	O(1)

- Various tree structures allow for a more efficient phone book.

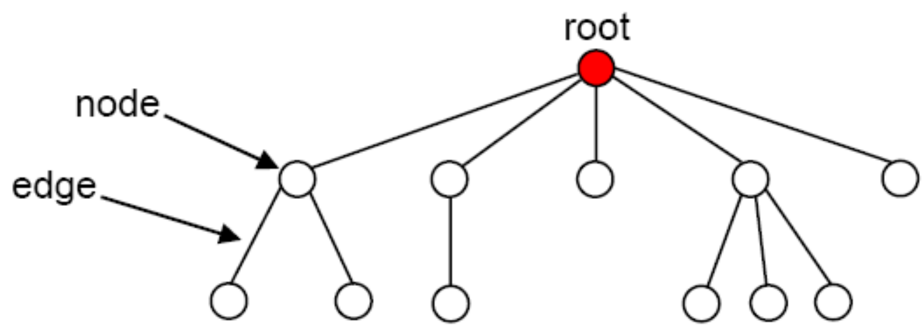
Hierarchies Everywhere

- Your local directory structure
- Governmental and organizational structures
- General approach of dealing with large-scale computer systems
  - P2P networks
    - Kazaa P2P network (peers and superpeers)
    - Gnutella 2
  - Hostnames
  - IP addresses
  - Internet

- Trees are natural representation of hierarchies

What Is A Tree?

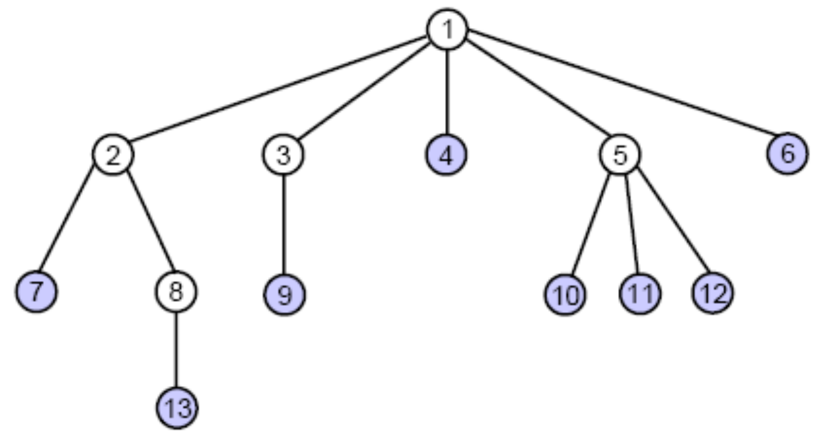
- A tree consists of:
  - a set of *nodes,* with one of them distinguished as a *root*
  - a set of *edges*, each of which connects a pair of nodes
  - **no cycles**



- Each node may have an associated *data item* ("payload").
  - consists of one or more fields
  - ***key field*** = the field used when searching for a data item
- The node at the "top" of the tree is called the *root* of the tree.

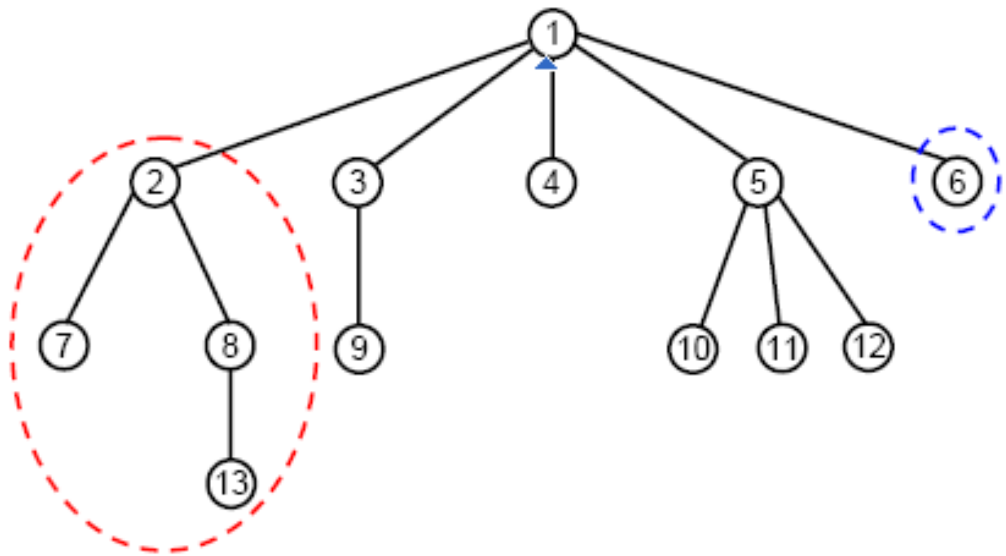
Relationships Between Nodes

- If a node N is connected to other nodes that are directly below it in the tree, N is referred to as their ***parent*** and the other nodes are referred to as its ***children***.
  - example: node 5 is the parent of nodes 10, 11, and 12



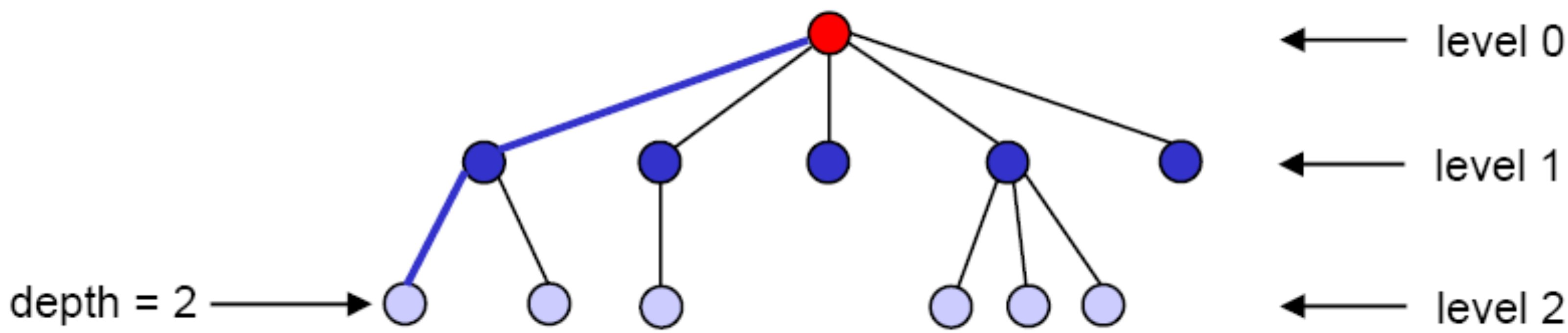
- Each node is the child of ***at most one*** **parent.**
- Other "family terms":
  - nodes with the same parent are ***siblings***
  - a node's ***ancestors*** are its parent, its parent's parent, etc.
  - a node's ***descendants*** are its children, their children, etc.
- A ***leaf*** *node* is a node without children.
- An ***interior*** *node* is a non-leaf node (but sometimes meant as a non-leaf and non-root).

A Tree Is A Recursive Data Structure



- Each node in the tree is the root of a smaller tree!
  - refer to such trees as ***subtrees*** to distinguish them from the tree as a whole
  - example: node 2 is the root of the subtree circled above
  - example: node 6 is the root of a subtree with only one node
- We'll see that tree algorithms often lend themselves to recursive implementations.
- Is an empty set of nodes a tree?

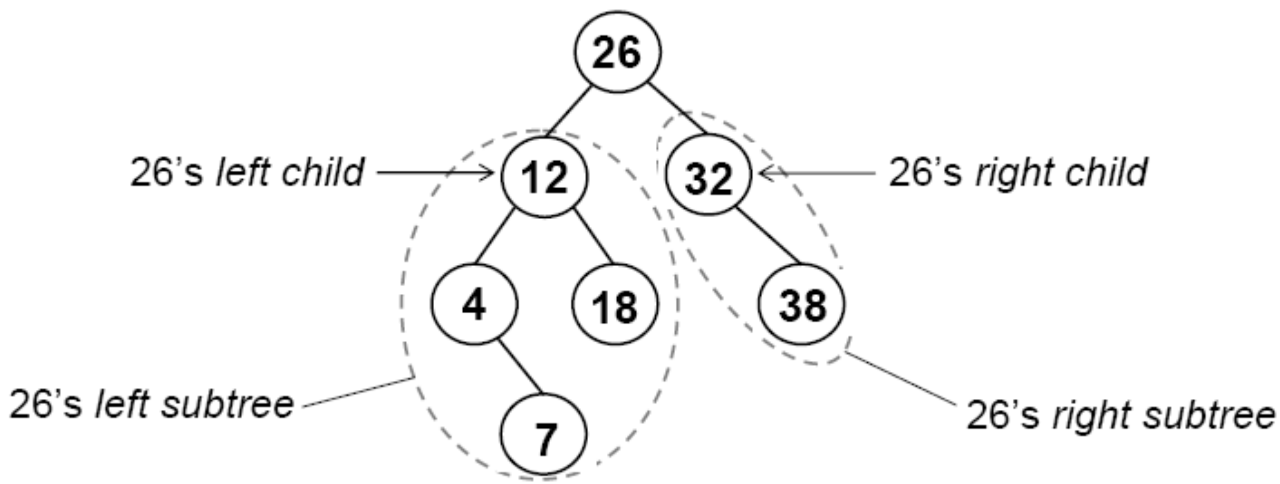
Path, Depth, Level, and Height



- There is exactly one *path* (one sequence of edges) connecting each node to the root.
- ***depth*** of a node = # of edges on the path from it to the root
- Nodes with the same depth form a ***level*** of the tree.
- The ***height*** of a tree is the maximum depth of its nodes.
  - example: the tree above has a height of 2

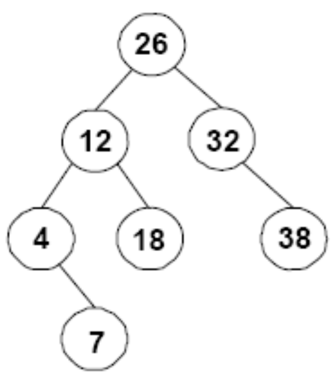
Binary Trees

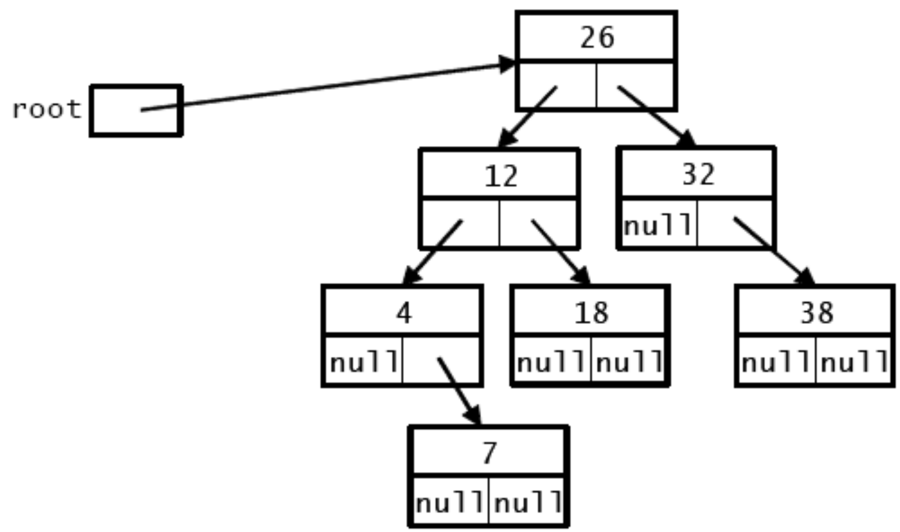
- In a *binary tree*, nodes have *at most two* children.
- Recursive definition: a binary tree is a collection of nodes that is either:
  - 1) empty, or
  - 2) contains a node R (the root of the tree) that has
    - a binary *left subtree*, whose root (if any) connects to R
    - a binary *right subtree*, whose root (if any) connects to R



Binary Tree Representation in Java

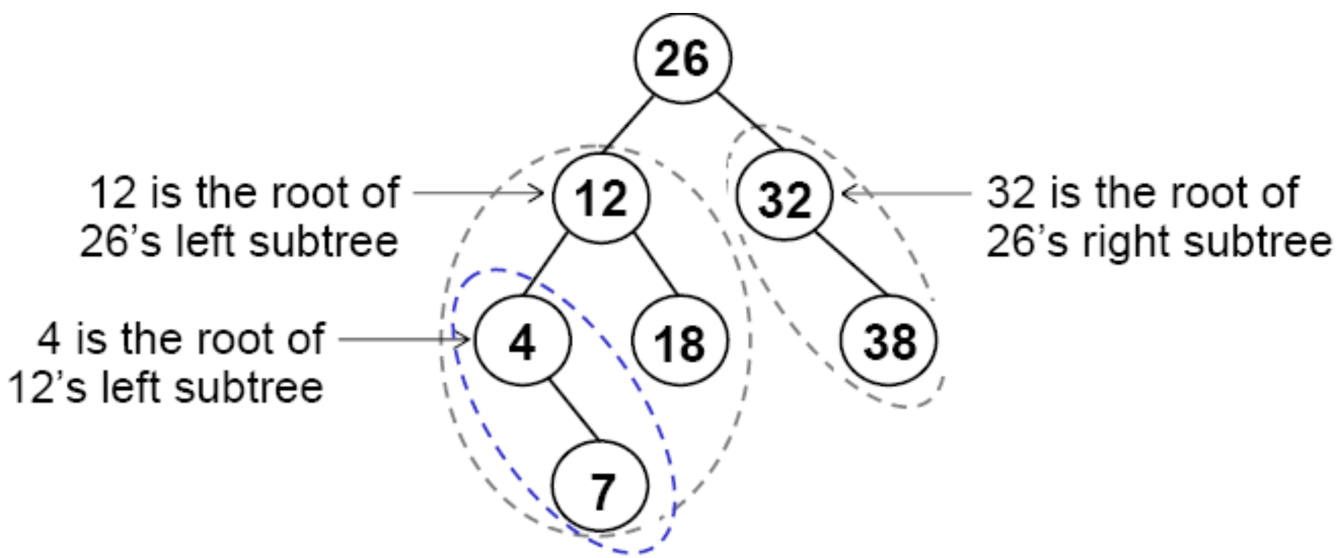
```
public class LinkedTree {
        private class Node {
                private int key;
                private String data;
                private Node left;          // reference to left child
                private Node right;         // reference to right child
                …
        }
        private Node root;
        …
}
```





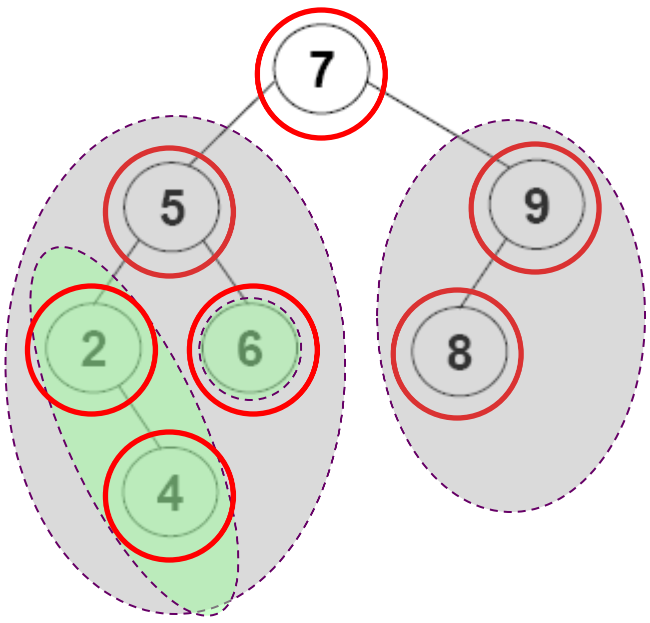
Traversing A Binary Tree

- Traversing a tree involves *visiting* all of the nodes in the tree.
  - visiting a node = processing its data in some way, e.g., print it
- We will look at four types of traversals.
  - Each visits the nodes in a different order.
- To understand traversals, keep in mind the recursive definition of a binary tree: every node is the root of a subtree.



Preorder Traversal

- Preorder traversal of the tree whose root is N:
  - visit the root, N
  - recursively perform a preorder traversal of N's left subtree
  - recursively perform a preorder traversal of N's right subtree



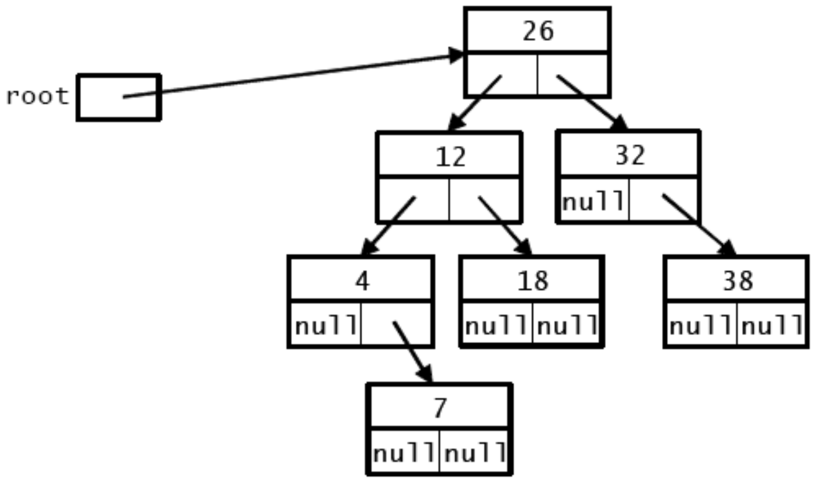
Result: 7 5 2 4 6 9 8

Implementing Preorder Traversal in Java

```
public class LinkedTree {
    …
    private Node root;
    public void preorderPrint() {
        if (root != null)
            myPreorderPrint(root);
    }
    private void myPreorderPrint(Node root) {
        System.out.print(root.key + " ");
        if (root.left != null)
            ?
        if (root.right != null)
            ?
    }
}
```

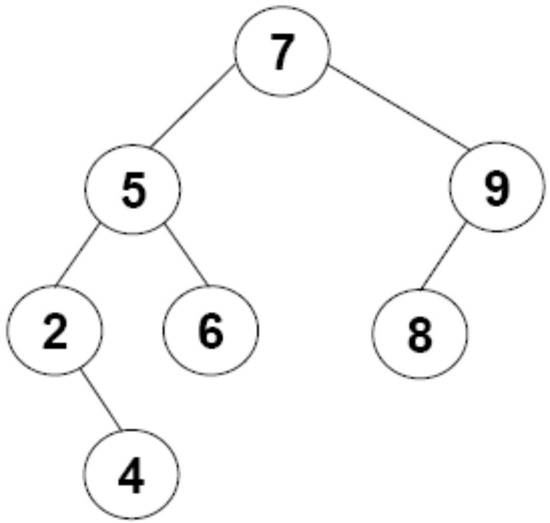
```
private class Node {
    private int key;
    private String data;
    private Node left;

    pivate Node right;
    …
}
```



Postorder Traversal

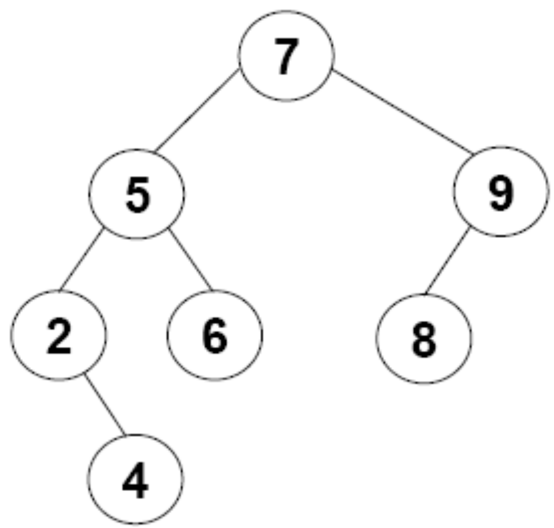
- postorder traversal of the tree whose root is N:
  - recursively perform a postorder traversal of N's left subtree
  - recursively perform a postorder traversal of N's right subtree
  - visit the root, N



Result: 4 2 6 5 8 9 7

Inorder Traversal

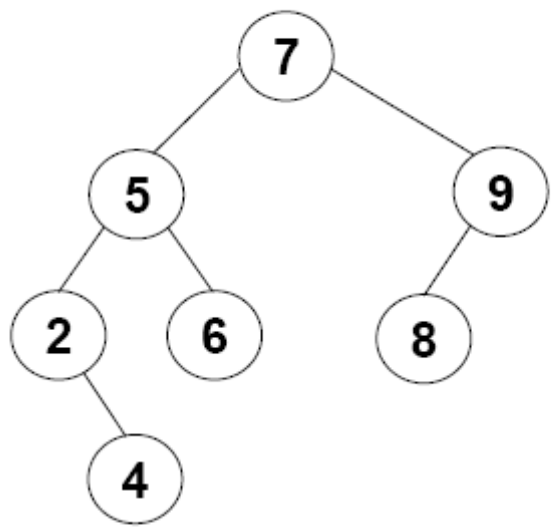
- Inorder traversal of the tree whose root is N:
  - recursively perform an inorder traversal of N’s left subtree
  - visit the root, N
  - recursively perform an inorder traversal of N’s right subtree



Result: 2 4 5 6 7 8 9

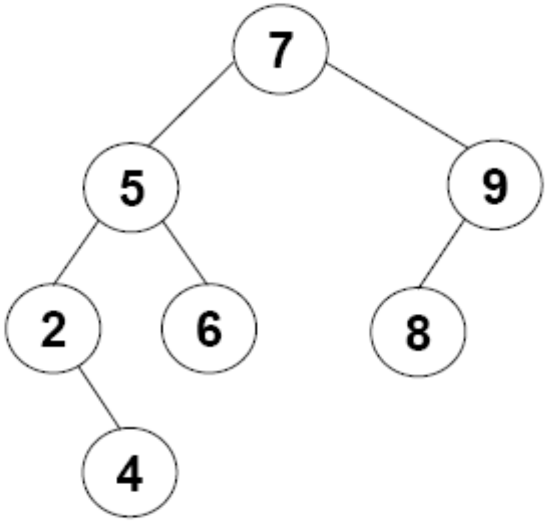
Level-Order Traversal

- Visit the nodes one level at a time, from top to bottom and left to right
- AKA “breadth-first traversal”



- Result: 7 5 9 2 6 8 4

Level-Order Traversal - Implementation



- Result: 7 5 9 2 6 8 4

Level-Order Traversal using Queue

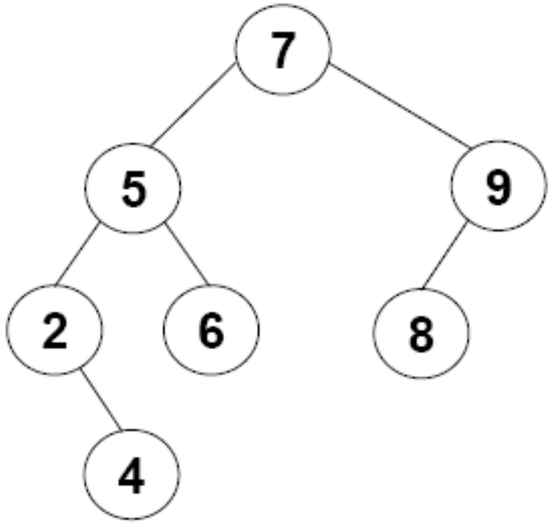
Initialization: insert the root in the queue

while queue is not empty

Remove a node from queue

Print the node

Insert each child into the queue



- Result: 7 5 9 2 6 8 4

Tree-Traversal Summary

- preorder: root, left subtree, right subtree
- postorder: left subtree, right subtree, root
- inorder: left subtree, root, right subtree
- level-order: top to bottom, left to right

- Perform each type of traversal on the tree below: