Trees, Binary Search Trees, Lab 7, Project 2

Bryce Boe

2013/11/13

CS24, Fall 2013

Outline

- Stack/Queue Review
- Trees
- Binary Search Trees
- Lab 7 / Project 2

Stack / Queue Review

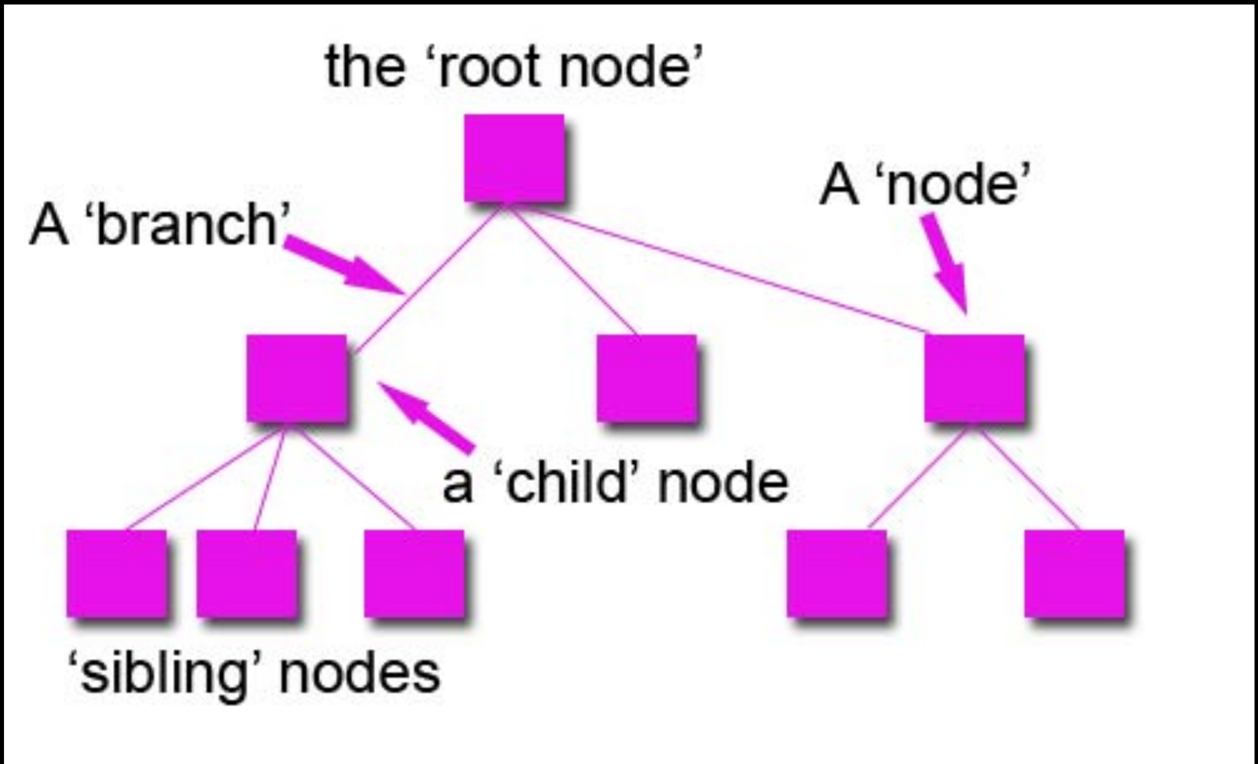
- Stack operations
  - push
  - pop
- Queue operations
  - enqueue
  - dequeue

TREES

Tree Explained

- Data structure composed of nodes (like a linked list)
- Each node in a tree can have one or more children (binary tree has at most two children)

General Tree



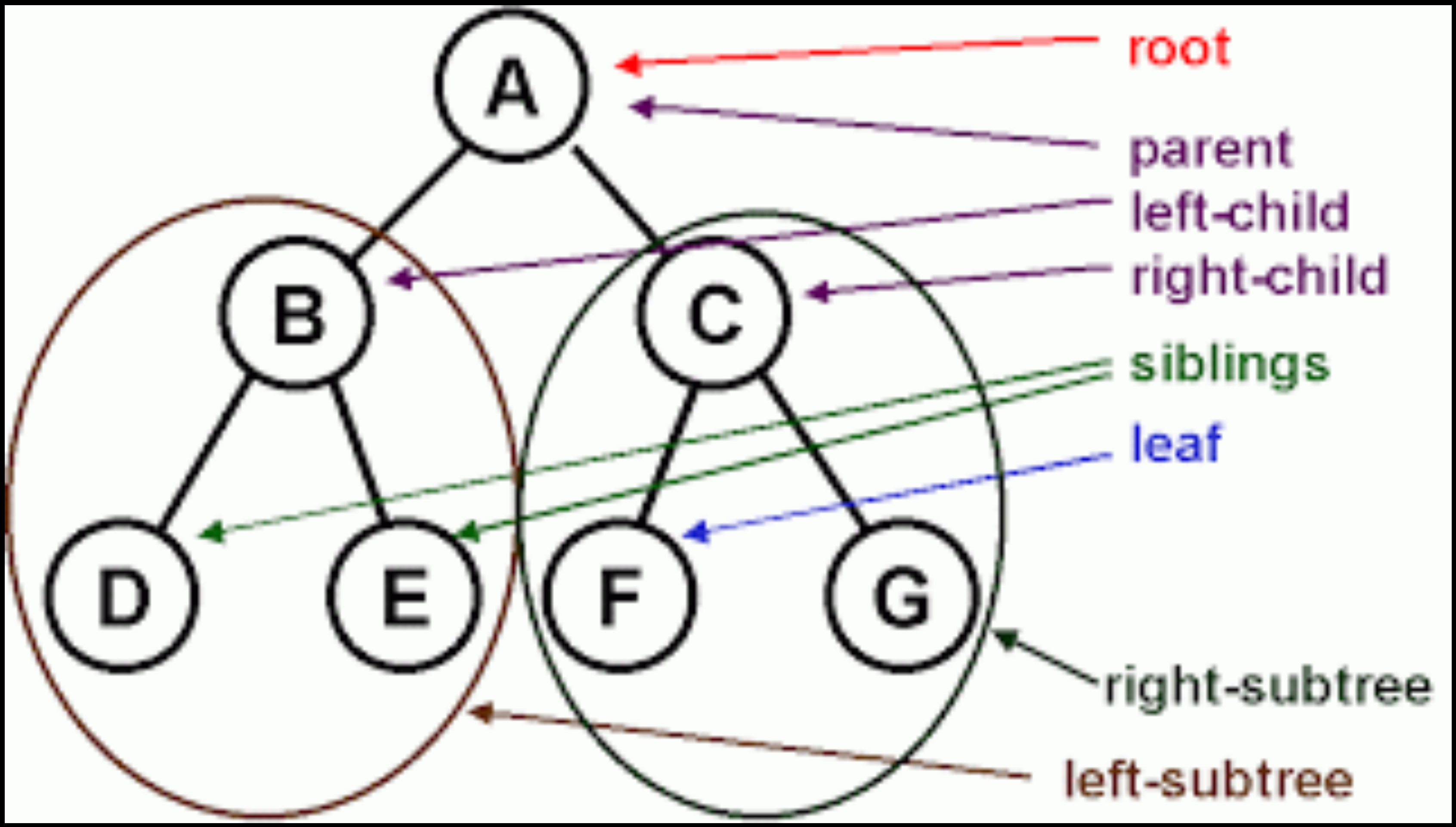
Tree Properties

- The **root** is the top-most node of the tree (has no parent)
- A node’s **parent** is the node immediately preceding it (closer to the root)
- A node can have at most two **children** or **child** nodes
- A **leaf** is a node with no children

More Properties

- A node's **ancestors** are all nodes preceding it
- A node's **descendants** all all nodes succeeding it
- A **subtree** is the complete tree starting with a given node and including its descendants

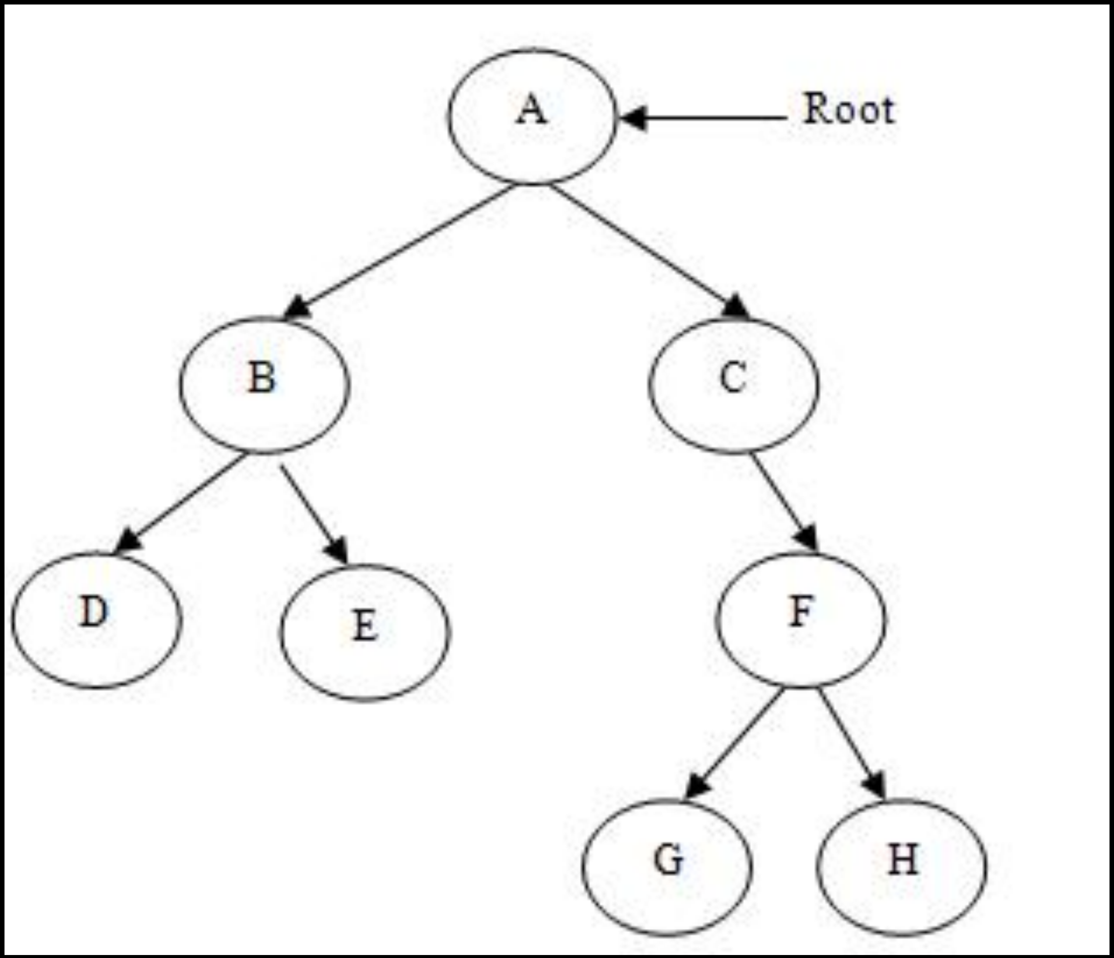
Tree properties



Binary Tree

- Each node can have at most two children

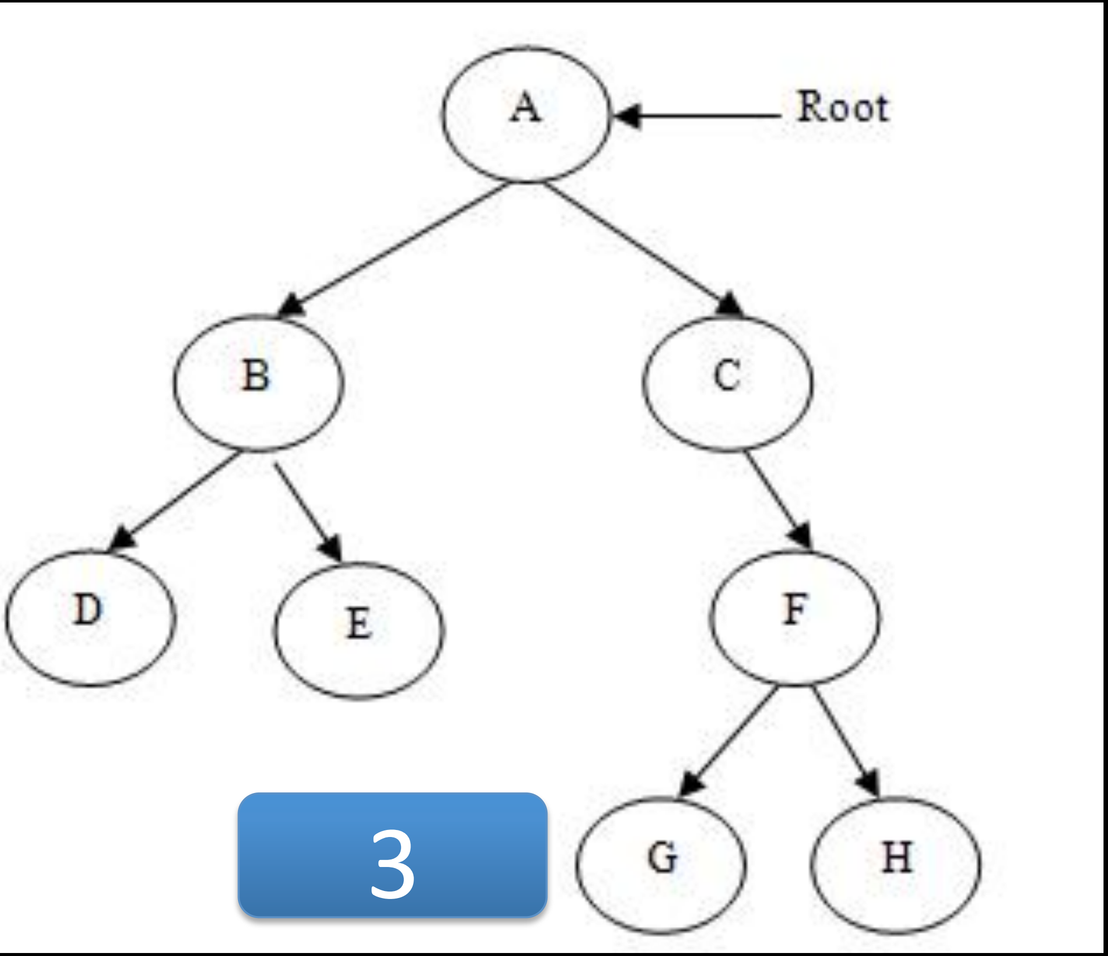
Binary Tree



More Properties

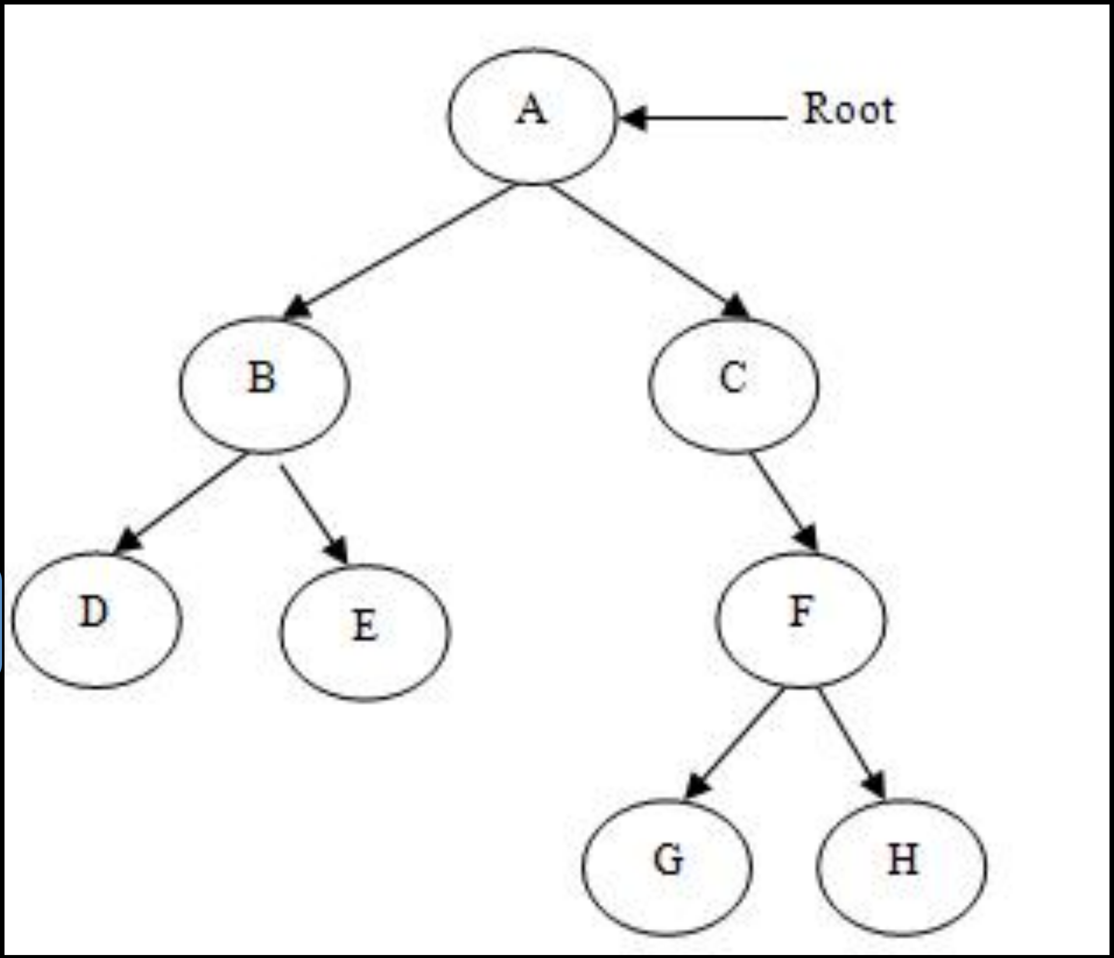
- The **depth** of a node is how far it is away from the root (the root is at depth 0)
- The **height** of a node is the maximum distance to one of its descendent leaf nodes (a leaf node is at height 0)
- The **height** of a tree is the height of the root node

What is the depth of G?



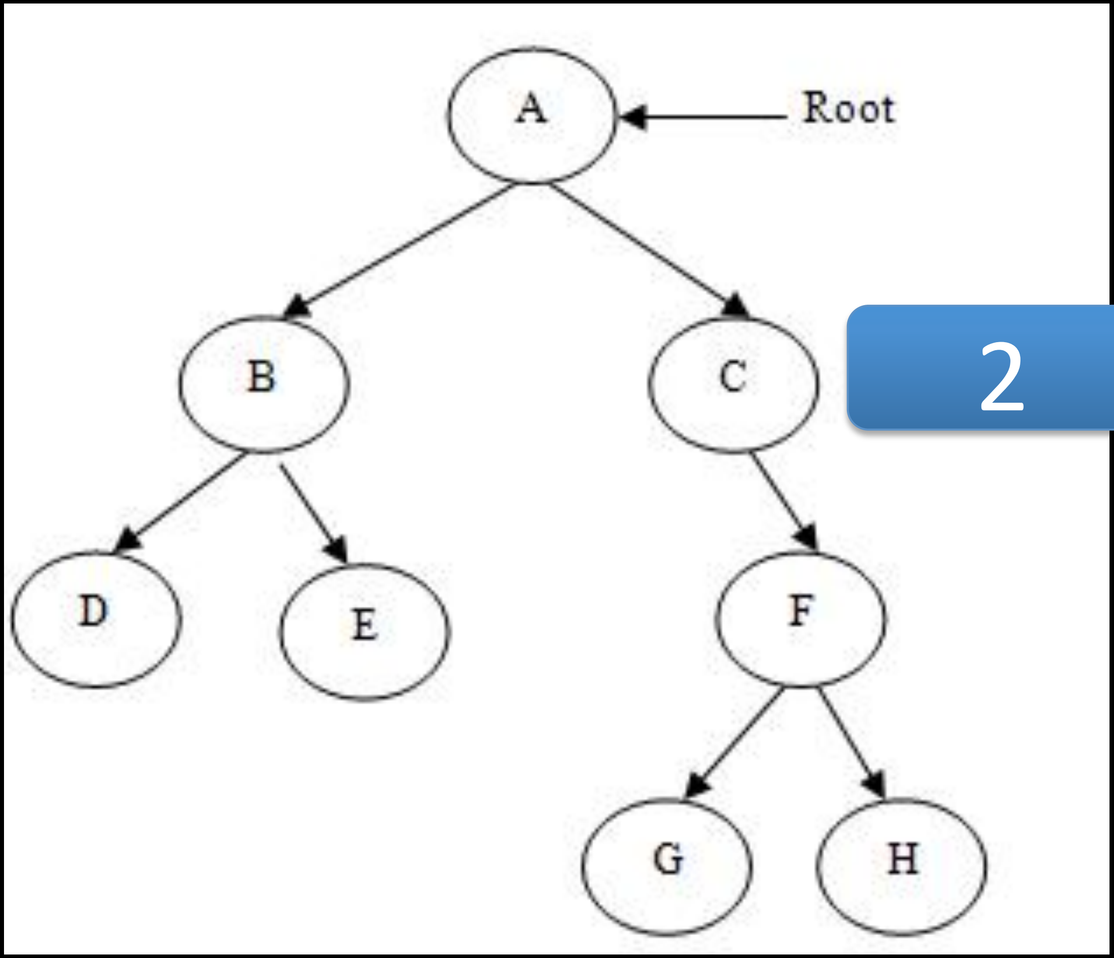
3

What is the depth of D?



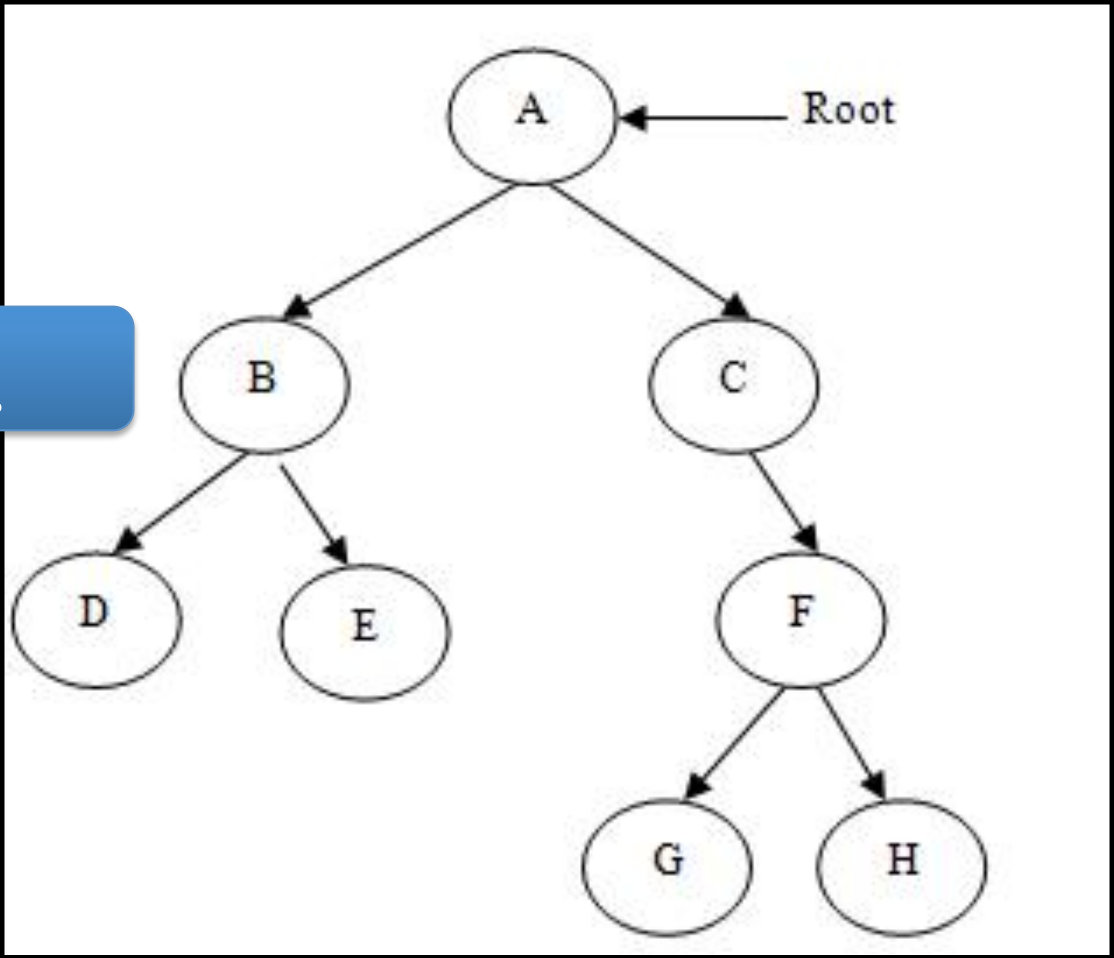
2

What is the height of C?



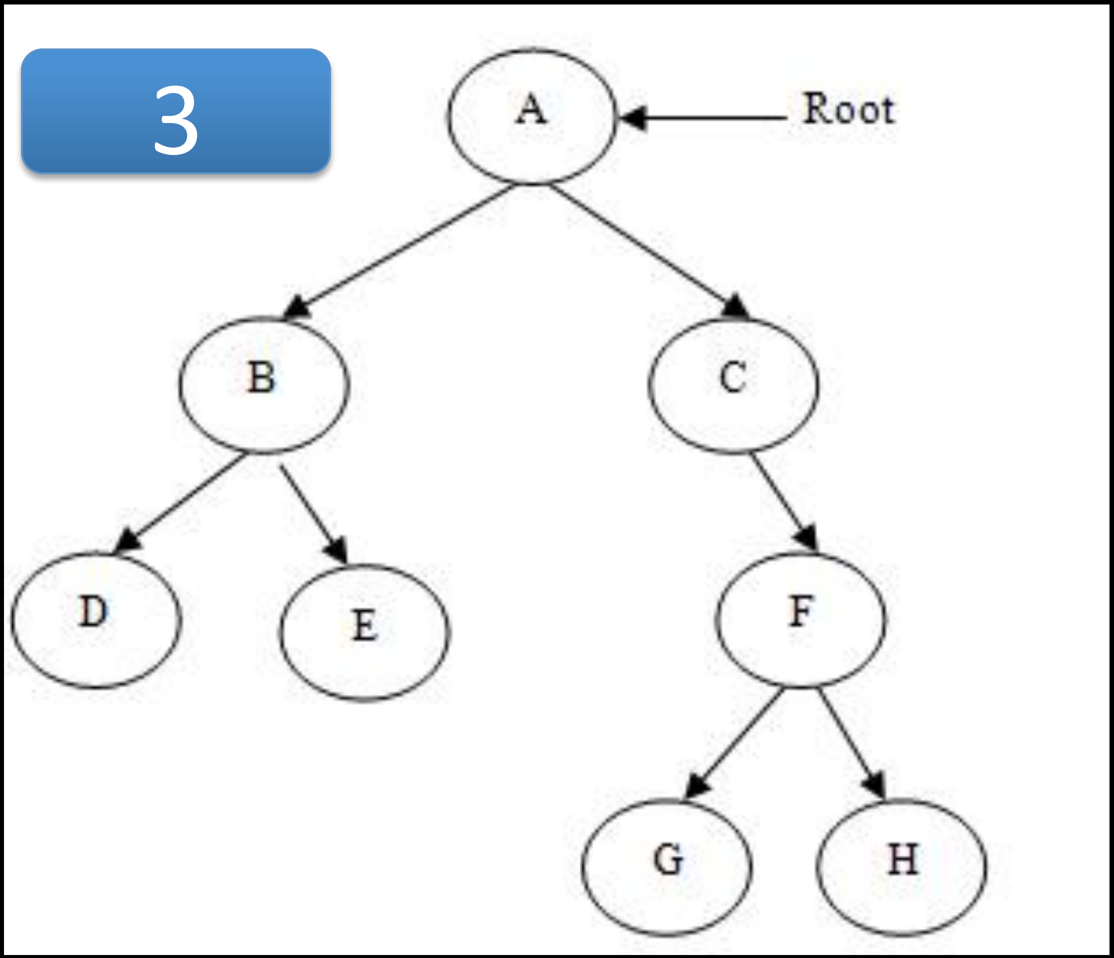
2

What is the height of B?

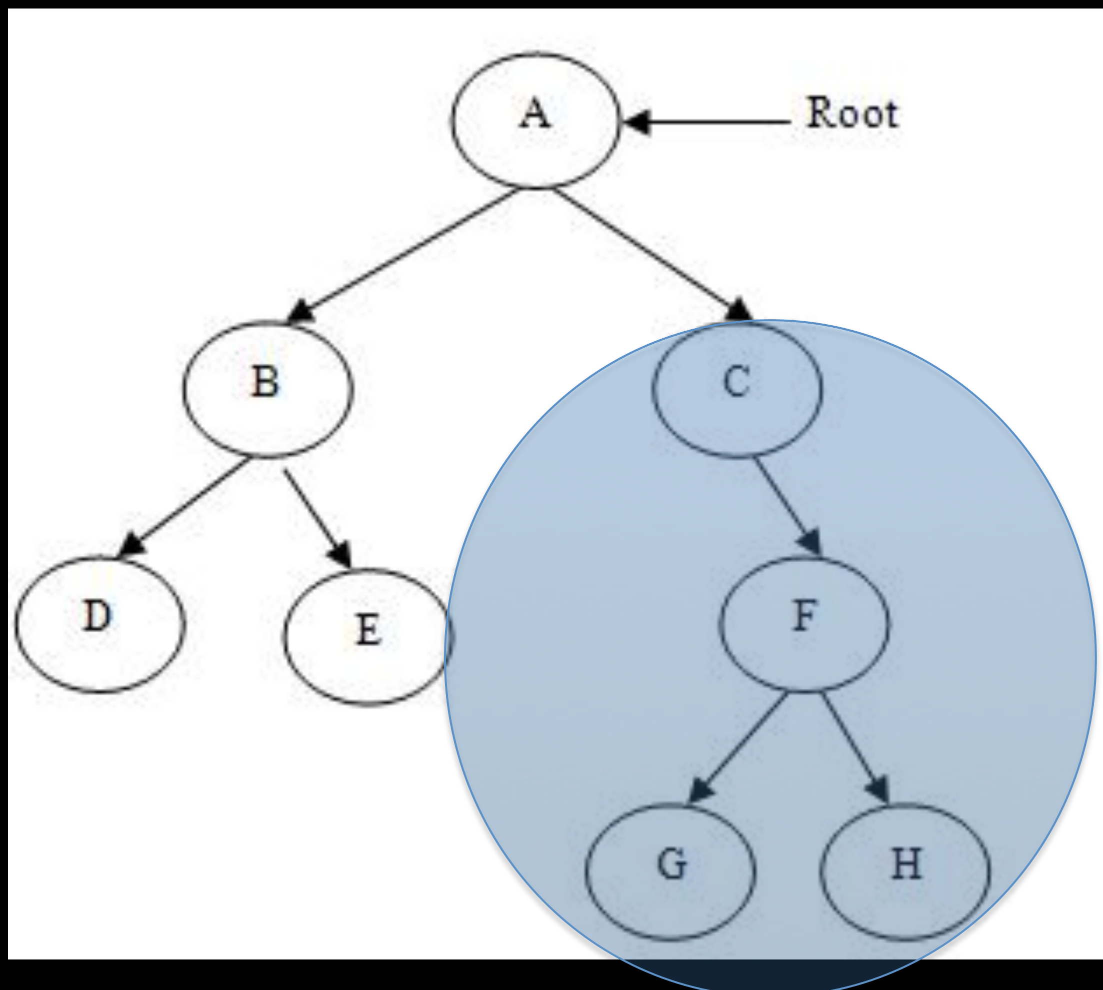


1

What is the height of the tree?



What nodes make up A’s right subtree?

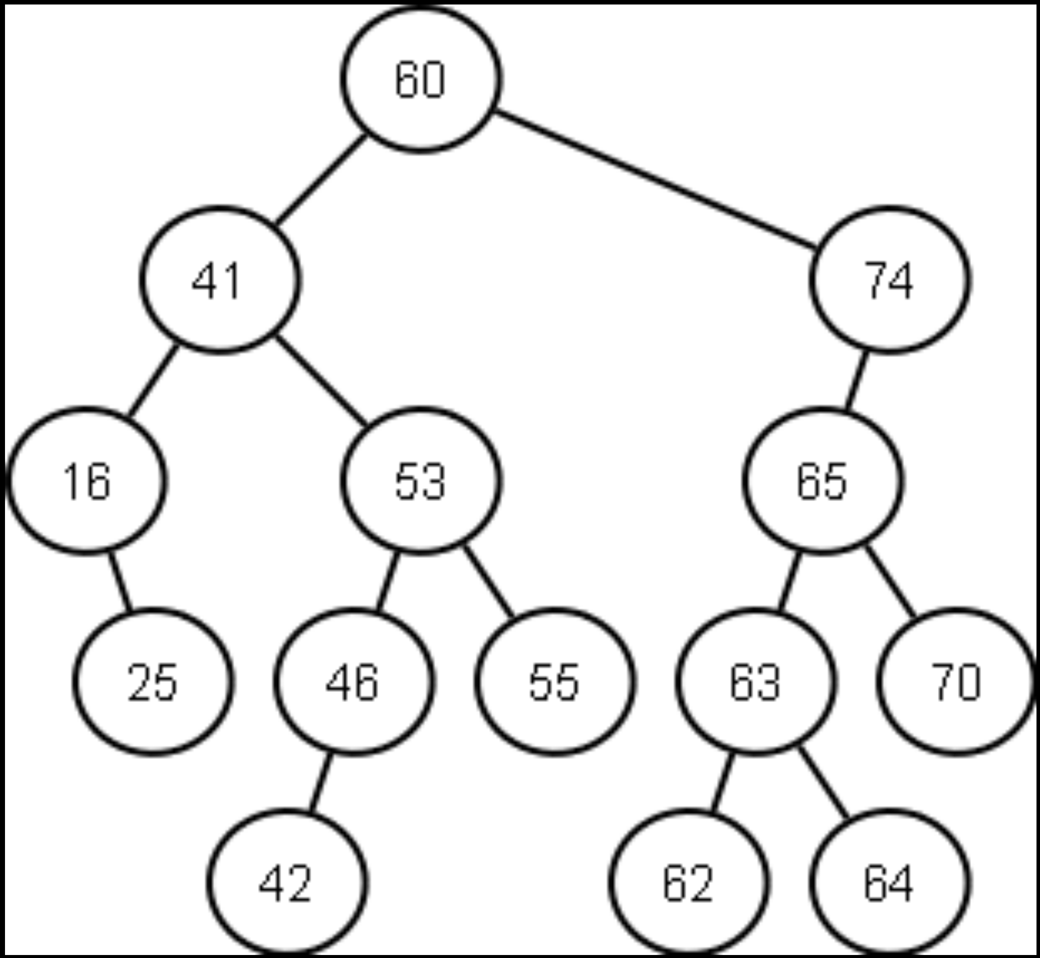


BINARY SEARCH TREES

Binary Search Trees

- A tree with the property that the value of all descendants of a node’s left subtree are smaller, and the value of all descendants of a node’s right subtree are larger

BST Example



BST Operations

- insert(item)
  - Add an item to the BST
- remove(item)
  - Remove an item from the BST
- contains(item)
  - Test whether or not the item is in the tree

What are the running times?

Balanced Tree

- A tree is considered balanced if
  - The height of the left and right subtrees differ by at most 1
  - The left and right subtrees are balanced

BST Running Times

- All operations are O(n) in the worst case
  - Why?
- Assuming a balanced tree (CS130A material):
  - insert: O(log(n))
  - delete: O(log(n))
  - contains: O(log(n))

BST Insert

- If empty insert at the root
- If smaller than the current node
  - If no node on left: insert on the left
  - Otherwise: set the current node to the lhs (repeat)
- If larger than the current node
  - If no node on the right: insert on the right
  - Otherwise: set the current node to the rhs (repeat)
- Otherwise fail the insert (attempt to insert a duplicate node)

BST Contains

- If the value is equal SUCCESS!
- If the value is smaller, continue down the left subtree
- If the value is larger, continue down the right subtree
- If the node is a leaf and the value does not match, FAILURE!

BST iterative traversal

```
ADT items;
items.add(root);  // Seed the ADT with the root
while(items.has_stuff() {
        Node *cur = items.random_remove();
        do_something(cur);
        items.add(cur.get_lhs()); // might fail
        items.add(cur.get_rhs()); // might fail
}
```

A look at lab 7 and project 2

- Lab 7 requires you to write *insert*, *queue_output* and a destructor for a BST
- The first part of project 2 requires you to utilize this code to implement a virtual tree

BST Remove

- If the node has no children simply remove it
- If the node has a single child, update its parent pointer to point to its child and remove the node

Removing a node with two children

- Replace the value of the node with the largest value in its left-subtree (right-most descendant on the left hand side)
- Then repeat the remove procedure to remove the node whose value was used in the replacement

Removing a node with two children