ECE2810J
Data Structures and Algorithms

Binary Tree Traversal

Learning Objectives:

- Know the effect and procedure of pre-order, post-order, and in-order depth-first traversal
- Know the effect and procedure of level-order traversal
- Application with traversal

Binary Tree Traversal

- Many binary tree operations are done by performing a **traversal** of the binary tree.
- In a traversal, each node of the binary tree is visited **exactly once**.
- During the visit of a node, all actions (making a clone, displaying, evaluating the operator, etc.) with respect to this node are taken.

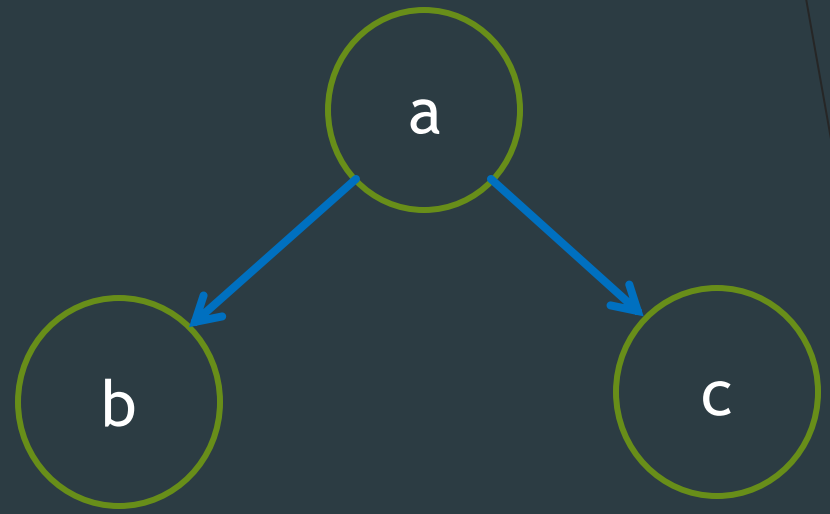
Binary Tree Traversal Methods

- Depth-first traversal
  - Pre-order
  - Post-order
  - In-order
- Level-order traversal

Pre-Order Depth-First Traversal
Procedure

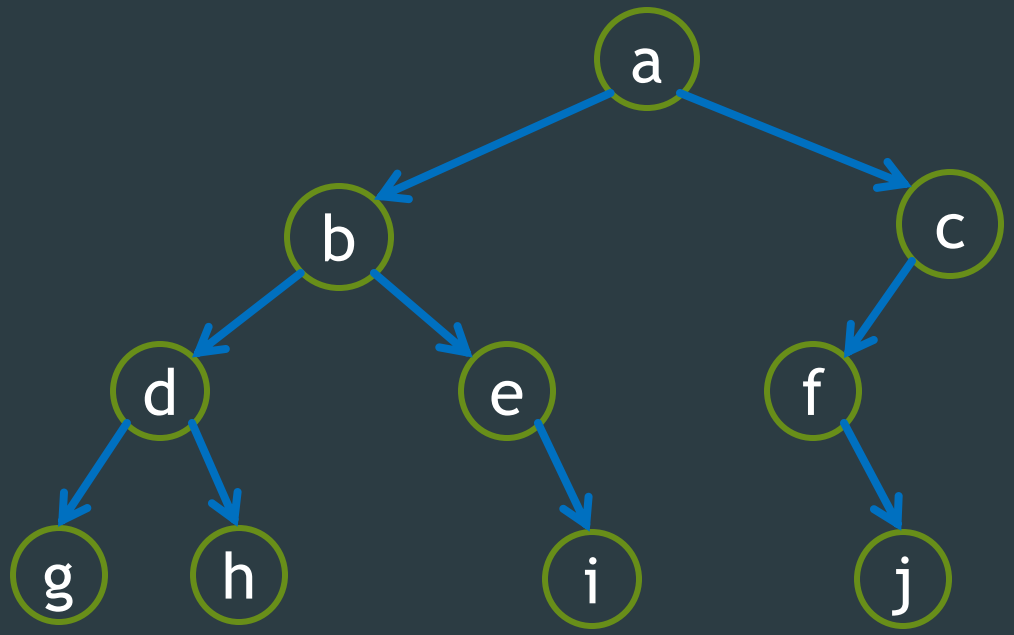
- Visit the node
- Visit its left subtree
- Visit its right subtree

```
void preOrder(node *n) {
  if(!n) return;
  visit(n);
  preOrder(n->left);
  preOrder(n->right);
}
```



a→b→c

Pre-Order Depth-First Traversal Example

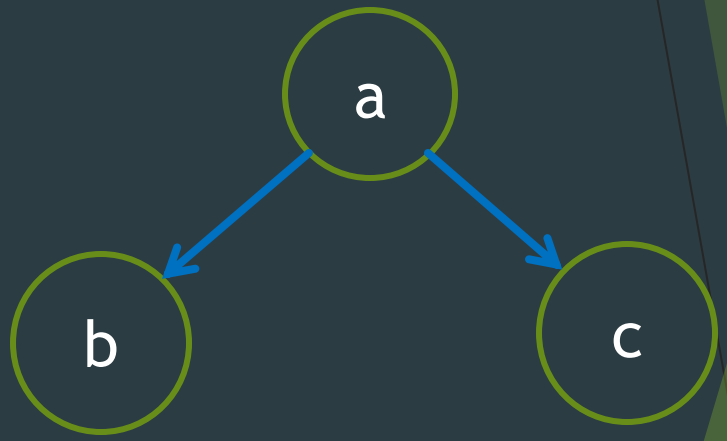


a→b→d→g→h→e→i→c→f→j

Post-Order Depth-First Traversal
Procedure

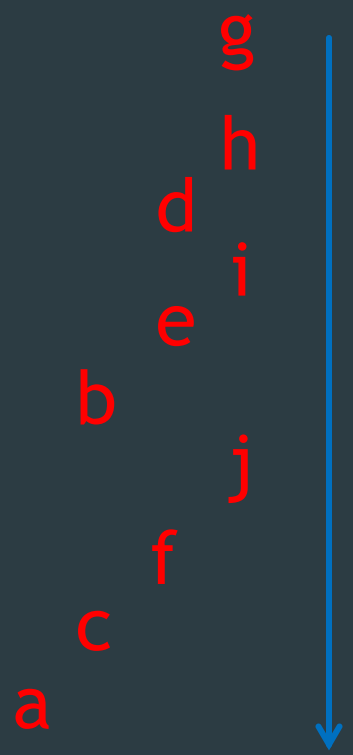
- Visit the left subtree
- Visit the right subtree
- Visit the node

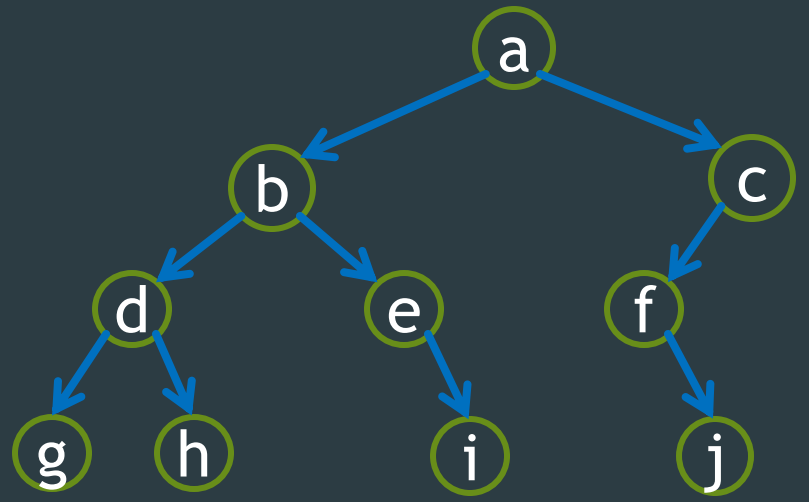
```
void postOrder(node *n) {
  if(!n) return;
  postOrder(n->left);
  postOrder(n->right);
  visit(n);
}
```



b→c→a

Post-Order Depth-First Traversal
Example



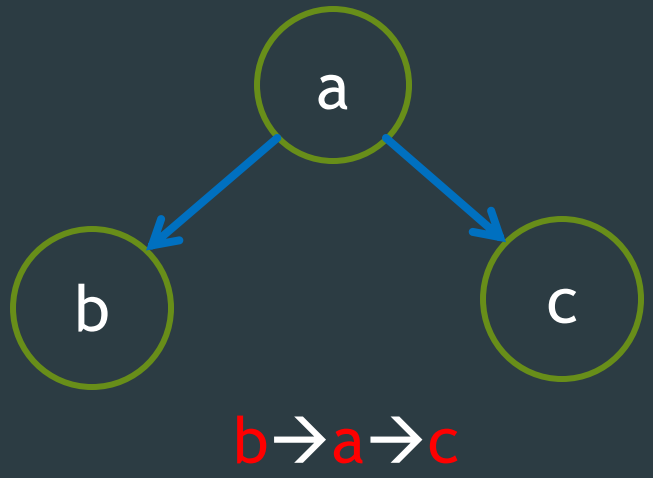


g→h→d→i→e→b→j→f→c→a

In-Order Depth-First Traversal
Procedure

- Visit the left subtree
- Visit the node
- Visit the right subtree

```
void inOrder(node *n) {
  if(!n) return;
  inOrder(n->left);
  visit(n);
  inOrder(n->right);
}
```



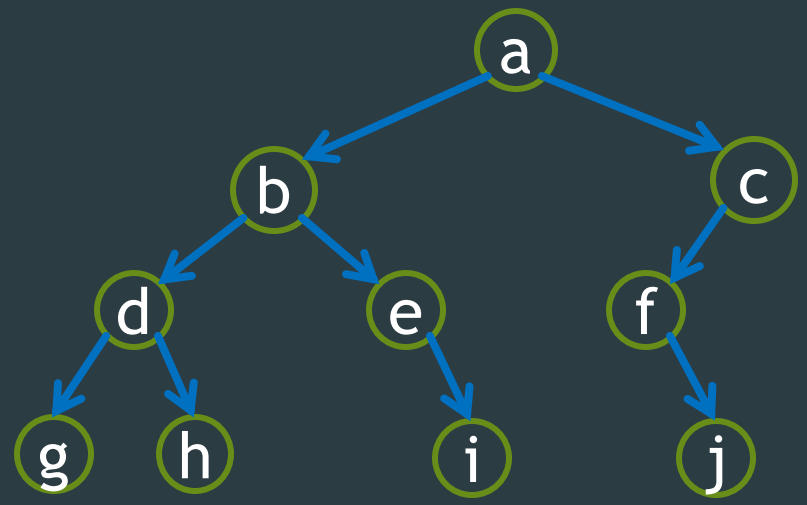
What Is the Result of In-Order Depth-First Traversal?

A. g, d, h, b, e, i, a, c, f, j

B. g, d, h, b, e, i, a, f, j, c

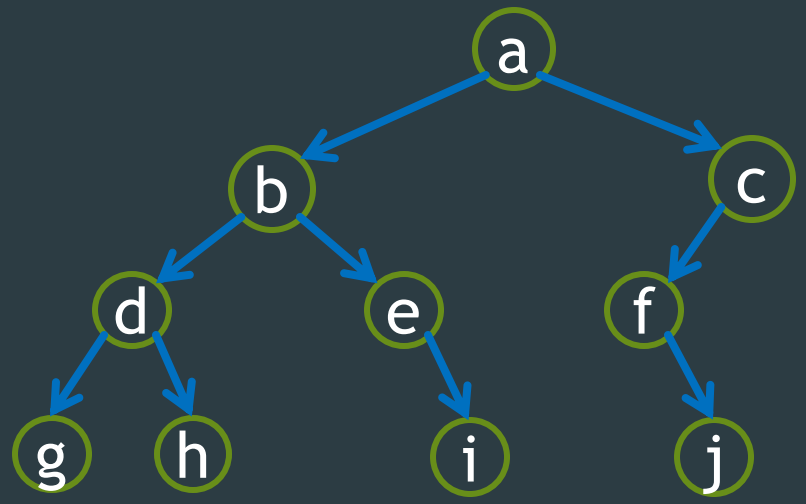
C. g, d, h, b, i, e, a, j, f, c

D. g, d, h, b, i, e, a, f, j, c



Level-Order Traversal

- We want to traverse the tree level by level from top to bottom.
- Within each level, traverse from left to right.



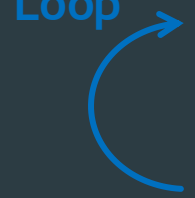
a→b→c→d→e→f→g→h→i→j

How can we implement this traversal?

Level-Order Traversal
Procedure

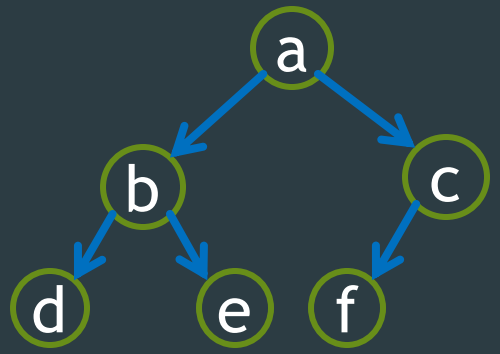
- Use a queue!

1. Enqueue the root node into an empty queue.
2. While the queue is not empty, dequeue a node from the front of the queue.
    1. Visit the node.
    2. Enqueue its left child (if exists) and right child (if exists) into the queue.



Level-Order Traversal
Code and Example

```
void levelOrder(node *root) {
  queue q; // Empty queue
  q.enqueue(root);
  while(!q.isEmpty()) {
    node *n = q.dequeue();
    visit(n);
    if(n->left) q.enqueue(n->left);
    if(n->right) q.enqueue(n->right);
  }
}
```

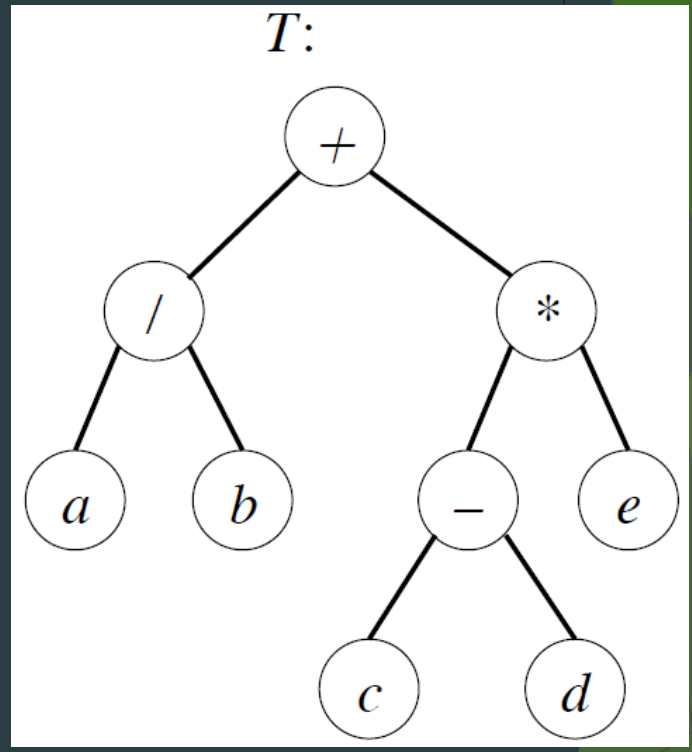


Queue: a b c d e f

Output: a b c d e f

Binary Tree Traversal Application

- The expression $a/b + (c - d)e$ has been encoded as a tree T.
  - The leaves are operands.
  - The internal nodes are operators.
- How would you traverse the tree T to print out the expression (ignoring parentheses)?
  - In-order depth-first traversal.
- What is the expression printed out by post-order depth-first traversal?
  - $ab/cd - e * +$
  - Reverse Polish Notation



In Class Exercise 1 (15 mins)

94. Binary Tree Inorder Traversal

Easy | Topics | Companies

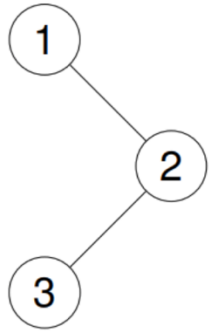
Given the `root` of a binary tree, return *the inorder traversal of its nodes' values*.

Example 1:

Input: `root = [1,null,2,3]`

Output: `[1,3,2]`

Explanation:



Example 2:

Input: `root = [1,2,3,4,5,null,8,null,null,6,7,9]`

Output: `[4,2,6,5,7,1,3,9,8]`

Explanation:

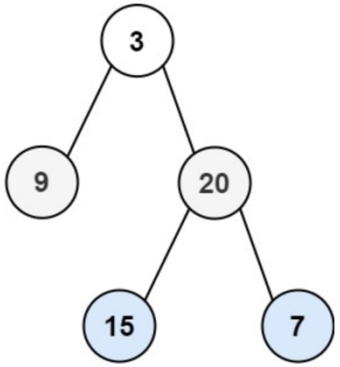
In Class Exercise 2 (15 mins)

107. Binary Tree Level Order Traversal II

Medium | Topics | Companies

Given the `root` of a binary tree, return *the bottom-up level order traversal of its nodes' values*. (i.e., from left to right, level by level from leaf to root).

Example 1:




```
Input: root = [3,9,20,null,null,15,7]
Output: [[15,7],[9,20],[3]]
```

Example 2:

```
Input: root = [1]
Output: [[1]]
```

Example 3:

```
Input: root = []
Output: []
```

Constraints:

- The number of nodes in the tree is in the range `[0, 2000]`.
- `-1000 <= Node.val <= 1000`

In Class Exercise 3 (15 mins)

1993. Operations on Tree

Medium | Topics | Companies | Hint

You are given a tree with `n` nodes numbered from `0` to `n - 1` in the form of a parent array `parent` where `parent[i]` is the parent of the i^{th} node. The root of the tree is node `0`, so `parent[0] = -1` since it has no parent. You want to design a data structure that allows users to lock, unlock, and upgrade nodes in the tree.

The data structure should support the following functions:

- **Lock: Locks** the given node for the given user and prevents other users from locking the same node. You may only lock a node using this function if the node is unlocked.
- **Unlock: Unlocks** the given node for the given user. You may only unlock a node using this function if it is currently locked by the same user.
- **Upgrade: Locks** the given node for the given user and **unlocks** all of its descendants **regardless** of who locked it. You may only upgrade a node if **all** 3 conditions are true:
  - The node is unlocked,
  - It has at least one locked descendant (by **any** user), and
  - It does not have any locked ancestors.

Implement the `LockingTree` class:

`rent)` initializes the data structure with the parent array.

`er)` returns `true` if it is possible for the user with id `user` to lock the node `num`, or `false` otherwise. If it is will become **locked** by the user with id `user`.

`user)` returns `true` if it is possible for the user with id `user` to unlock the node `num`, or `false` otherwise. If it is will become **unlocked**.

`user)` returns `true` if it is possible for the user with id `user` to upgrade the node `num`, or `false` otherwise. If `num` will be **upgraded**.