CS 2230
CS II: Data structures

Meeting 22: tree traversal
Ryan Wedoff
University of Iowa
Today’s big ideas

- **Depth-first traversal** finishes a branch before moving to another and can be done in pre-order, in-order, and post-order.

- **Breadth-first traversal** visits every node at the current depth before visiting the next depth.

- We can use higher order functions to separate the code that computes something interesting from the code that does the traversal.
Summary: visiting nodes of tree in different orders

- Depth-first: nodes are visited in order of depth, starting from the root node and going as deep as possible before backtracking.
- Breadth-first: nodes are visited in order of their level in the tree, starting from the root node and moving outward to the leaves.

The diagrams illustrate the difference between depth-first and breadth-first traversal of a tree.
Peer instruction

Consider BinaryTree.sum() from last time

```java
// return sum of all data in the tree
// only works for a BinaryTreeNode<Integer>
public int sum() {
    int total = 0;
    // check for base case on left
    if (this.left != null) {
        // the inductive step
        total += this.left.sum();
    }
    // check for base case on right
    if (this.right != null) {
        // the inductive step
        total += this.right.sum();
    }
    total += (Integer) this.data;
    return total;
}
```

What is the running time?

a) O(1)
b) O(logN)
c) O(N)
d) O(NlogN)
e) O(N^2)

https://b.socrative.com/login/student/
CS2230A ids 1000-4999
CS2230B ids 5000+
sum() does a **traversal**

```
// return sum of all data in the tree
// only works for a BinaryTreeNode<Integer>
public int sum() {
    int total = 0;
    // check for base case on left
    if (this.left != null) {
        // the inductive step
        total += this.left.sum();
    }
    // check for base case on right
    if (this.right != null) {
        // the inductive step
        total += this.right.sum();
    }
    total += (Integer) this.data;
    return total;
}
```

dotted line: traversal path through the tree
solid dots: indicate where on the traversal the data field is looked at

sum() visits the data field of each BinaryTreeNode in the following order
This traversal is known as **depth-first, post-order**.

**Depth first** means finish exploring entire subtree before looking at another child.

**Post-order** means visit the root of a subtree *after* visiting its children.
Peer instruction

void printTree() {
    if (this.left != null) this.left.printTree();
    System.out.print(this.data);
    System.out.print(",");
    if (this.right != null) this.right.printTree();
}

What does this program print if printTree() is called on the following tree?

a) 1,2,4,5,3,6,7,8
b) 2,4,6,8,1,3,5,7,
c) 2,4,5,1,3,6,7,8,
d) 4,5,2,1,6,3,8,7,
e) 1,2,3,4,5,6,7,8,

https://b.socrative.com/login/student/
CS2230A ids 1000-4999
CS2230B ids 5000+
Depth-first traversal orders of binary tree

- **Preorder**: FABDCEGIH
- **Inorder**: ABCDEFGHI
- **Postorder**: ACEDBHIGF

We can use recursion to perform any of these orders.
Can we use a **loop** for a depth-first traversal?

*frontier*: the collection of nodes we know about but haven’t looked at yet

You’ll write the code in discussion this week!
Breadth-first traversal

This traversal is known as **breadth-first**
visit all nodes in a given depth before going to the next depth
Peer instruction

frontier: the collection of nodes we know about but haven’t looked at yet

Let’s use the frontier idea for *breadth-first* traversal

<table>
<thead>
<tr>
<th>iteration</th>
<th>look at</th>
<th>frontier</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td>A</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

In the iteration when we “look at” node C, what will be in the frontier?

a) A, B, E
b) F, G
c) F, G, D, H
d) D
e) D, E
# Depth vs Breadth

<table>
<thead>
<tr>
<th>Example data</th>
<th>Depth first</th>
<th>Breadth first</th>
</tr>
</thead>
<tbody>
<tr>
<td>file system</td>
<td>see files buried in many layers of folders before seeing <em>all</em> the top level folders</td>
<td>look at all the folders at a depth before looking deeply inside of one of them</td>
</tr>
<tr>
<td>classification of life</td>
<td>see a specific species before looking at all the classes</td>
<td>look at all the species at the end</td>
</tr>
<tr>
<td>management</td>
<td>talk to a low-level employee before talking to all the managers</td>
<td>talk to all the managers first before talking to any of their employees</td>
</tr>
</tbody>
</table>
You’ve seen higher order functions used with Lists and Iterators. Here’s an example with BinaryTreeNode.

```java
interface ApplyFunction<InT, OutT> {
    public OutT apply(InT x);
}

class BinaryTreeNode<T> {
    // replace every element e in the tree with f(e)
    void transformElements(ApplyFunction<T, T> f) {
        // transform the data by applying f
        this.data = f.apply(this.data);
        // transform each of the children if they exist
        if (this.left != null) this.left.transformElements(f);
        if (this.right != null) this.right.transformElements(f);
    }
}
```

Separation of concerns
- write the traversal code once in `transformElements`
- whoever wants to transform elements in the tree in a specific way just has to define a class that implements the `ApplyFunction` interface
Today’s big ideas

• **Depth-first traversal** finishes a branch before moving to another and can be done in pre-order, in-order, and post-order

• **Breadth-first traversal** visits every node at the current depth before visiting the next depth

• we can use higher order functions to separate the code that computes something interesting from the code that does the traversal