Communications of the ACM
“Learning to Learn” Denning & Flores
https://cacm.acm.org/magazines/2016/12/210369-learning-to-learn/fulltext
CS 2230
CS II: Data structures
Meeting 22: tree traversal
Brandon Myers
University of Iowa
Today’s learning objectives

• Identify various **Depth-first traversals** of a tree, specifically pre-order, in-order, and post-order

• Identify the **Breadth-first traversal** of a tree

• Interpret recursive code for tree traversals

• Describe how depth-first and breadth-first tree traversals can be done using iteration (i.e., loop)

• Interpret code that uses higher order functions on trees
Summary: visiting nodes of tree in different orders

- **breadth-first**
- **depth-first**
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
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
Depth-first traversal orders of binary tree

**Pre order**
Root, Left, Right
FABDCEGIH

**In order**
Left, Root, Right
ABCDEFGHI

**Post order**
Left, Right, Root
ACEDBHIGF

we can use recursion to perform any of these orders
Identify various **Depth-first traversals** of a tree, specifically pre-order, in-order, and post-order.

1. post order?
2. pre order?
3. in order?

a) 10, 6, 33, 2, 4, 20, 31, 3, 12
b) 2, 33, 4, 6, 10, 31, 20, 12, 3
c) 33, 31, 20, 12, 10, 6, 4, 3, 2
d) 10, 6, 20, 33, 31, 3, 2, 4, 12
e) 2, 3, 4, 6, 10, 12, 20, 31, 33
f) 2, 4, 33, 6, 31, 12, 3, 20, 10

https://b.socrative.com/login/student/
room CS2230X ids 1000-2999
room CS2230Y ids 3000+
Breadth-first traversal

This traversal is known as **breadth-first**
visit all nodes in a given depth before going to the next depth
Identify the **Breadth-first traversal** of a tree

a) 10, 6, 33, 2, 4, 20, 31, 3, 12
b) 2, 33, 4, 6, 10, 31, 20, 12, 3
c) 2, 4, 12, 33, 31, 3, 6, 20, 10
d) 10, 6, 20, 33, 31, 3, 2, 4, 12
e) 2, 3, 4, 6, 10, 12, 20, 31, 33
f) 2, 4, 33, 6, 31, 12, 3, 20, 10

https://b.socrative.com/login/student/
room CS2230X ids 1000-2999
room CS2230Y ids 3000+
Interpret recursive code for tree traversals

```java
void printTree1() {
    System.out.print("","+this.data);
    if (this.left != null) this.left.printTree1();
    if (this.right != null) this.right.printTree1();
}

void printTree2() {
    if (this.left != null) this.left.printTree2();
    System.out.print("","+this.data);
    if (this.right != null) this.right.printTree2();
}

void printTree3() {
    if (this.left != null) this.left.printTree3();
    if (this.right != null) this.right.printTree3();
    System.out.print("","+this.data);
}
```

A. Depth-first pre-order
B. Depth-first post-order
C. Depth-first in-order
D. Breadth-first
Can we use **iteration** (a loop) for a depth-first traversal?

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

**pre order**

<table>
<thead>
<tr>
<th>iteration</th>
<th>look at</th>
<th>frontier</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>--------</td>
<td>F</td>
</tr>
<tr>
<td>1</td>
<td>F</td>
<td>B,G</td>
</tr>
<tr>
<td>2</td>
<td>B</td>
<td>A,D,G</td>
</tr>
<tr>
<td>3</td>
<td>A</td>
<td>D,G</td>
</tr>
<tr>
<td>4</td>
<td>D</td>
<td>C,E,G</td>
</tr>
<tr>
<td>5</td>
<td>C</td>
<td>E,G</td>
</tr>
<tr>
<td>6</td>
<td>E</td>
<td>G</td>
</tr>
<tr>
<td>7</td>
<td>G</td>
<td>I</td>
</tr>
<tr>
<td>8</td>
<td>I</td>
<td>H</td>
</tr>
<tr>
<td>9</td>
<td>H</td>
<td></td>
</tr>
</tbody>
</table>

aside question: what data structure would be useful for the frontier?
Describe how depth-first and breadth-first tree traversals can be done using iteration (i.e., loop)

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>1</td>
<td>A</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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

[fill out the table](https://b.socrative.com/login/student/
room CS2230X ids 1000-2999
room CS2230Y ids 3000+)
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
interface TriFunction <T> {
    public T apply(T x, T y, T z);
    public T zero();
}

class BinaryTreeNode<
T> {
    void convert(TriFunction<T> f) {
        T left = f.zero();
        T right = f.zero();
        if (this.left != null) {
            this.left.convert(f);
            left = this.left.data;
        }
        if (this.right != null) {
            this.right.convert(f);
            right = this.right.data;
        }
        this.data = f.apply(this.data, left, right);
    }
}

class A implements TriFunction<String> {
    public String apply(String x, String y, String z) {
        return x + "," + y + "," + z;
    }
    public T zero() {
        return "";
    }
}
Today’s learning objectives

• Identify various **Depth-first traversals** of a tree, specifically pre-order, in-order, and post-order

• Identify the **Breadth-first traversal** of a tree

• Interpret recursive code for tree traversals

• Describe how depth-first and breadth-first tree traversals can be done using iteration (i.e., loop)

• Interpret code that uses higher order functions on trees