# CS 310: Trees (Part II)

Connor Baker

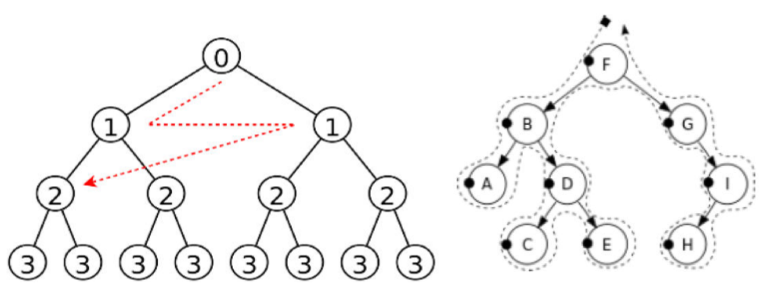
February 21, 2019

## Common Tree Operations

- Searching for an item
- Adding items
- Deleting items
- Balancing
- Iteration (either through all of the tree, or a sub-tree)

## Tree Traversals

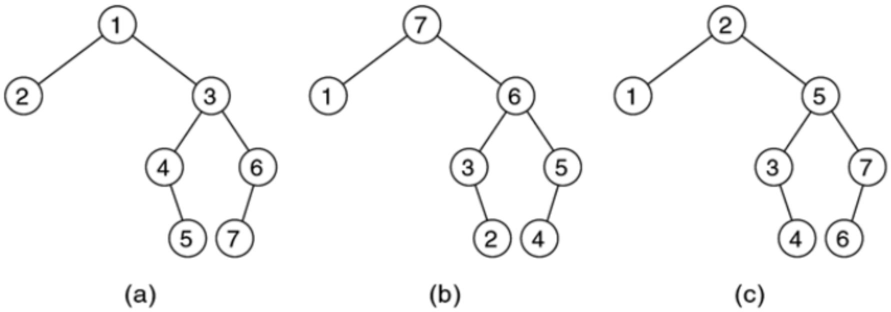
- Check Weiss 18.4: traversal implementation in an iterator
- Two common types:



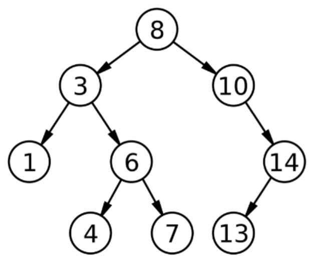

  - (Left) *Breadth first search*: a search which proceeds by processing all nodes level by level, with those closest to the root processed first
  - (Right) *Depth first search*: a search which proceeds by following a path all the way to a leaf and then backtracking

## Depth First Traversal

- The walking of a tree starts with the root and goes from parent to child
- There are three different orders we can “process” the nodes in:
  (a) Pre-order traversal (parent, left, right)
  (b) Post-order traversal (left, right, parent)
  (c) In-order traversal (left, parent, right)




## Traversal Examples




- For the given tree, show
  - Pre-order traversal
    * $8, 3, 1, 6, 4, 7, 10, 14, 13$
  - Post-order traversal
    * $1, 4, 7, 6, 3, 13, 14, 10$
  - In-order traversal
    * $1, 3, 6, 7, 4, 10, 13, 14$
- Which one sorts the nodes according to their data?
  - That would be a special feature of binary search trees

# Implementing Traversal for Binary Trees

```
class Node<T> {
    T data;
    Node<T> left, right;
}

inOrder(Node t) {
    if (t == null) {
        return;
    }

    inOrder(t.left);
    print(t.data);
    inOrder(t.right);
}

// to use
inOrder(this.root);
```

```
preOrder(Node t) {
    if (t == null) {
        return;
    }

    print(t.data);
    preOrder(t.left);
    preOrder(t.right);
}

postOrder(Node t) {
    if (t == null) {
        return;
    }

    postOrder(t.left);
    postOrder(t.right);
    print(t.data);
}
```

## Notes on the Tree Traversal Implementation

- The recursive approach is easy to code
- The time and space complexity isn't great, however
- We can perform various operations on each node following different orders

## Traversal Example

- Binary tree `size()` method, which counts how many nodes a tree contains

```
// Recursive method to count the number of nodes
public static <T> int size(Node<T> t) {
    if (t == null) {
        return 0;
    }

    int sL = size(t.left);
    int sR = size(t.right);

    return 1 + sL + sR;
}
```

- In what order are we processing the nodes? Why?
  - Post-order:- **is it because it allows lends itself to recursion more easily than the other methods?**

## Tree Traversal Examples

- Task: mark the depth of every node
  - What order should we use?
  - **Practice**: write the recursive code

- Task: check whether the (binary) tree is balanced
  - Order?
  - **Practice**: write the recursive code

## Iterative Tree Traversal

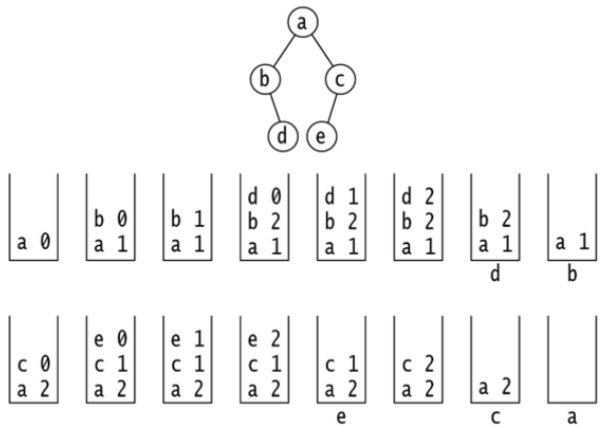
- Can we implement tree traversal without recursion?
  - Well... yes, but we need to maintain a stack (which is what recursion does for us)
- We can check the post-order traversal. Every node needs to go through three steps in order:
  - Process the node's left subtree
  - Process the node's right subtree
  - Process the node itself

## Iterative Implementation of Traversal

```
// Pseudo-code for post-order printing
void postOrder(root) {
    Stack s = new Stack();
    s.push({root, DOLEFT});
    while (!s.isEmpty()) {
        {tree, action} = s.popTop();
        if (tree == null) {
            // Do nothing
        } else if (action == DOLEFT) {
            s.push({tree, DORIGHT});
            s.push({tree.left, DOLEFT});
        } else if (action == DORIGHT) {
            s.push({tree, DOTHIS});
            s.push({tree.right, DOLEFT});
        } else if (action == DOTHIS) {
            print(tree.data);
        } else {
            throw new YouScrewedUpException();
        }
    }
}
```

- Use an explicit stack
  - Push the node onto the stack three times, each with a different task
- Auxiliary data action
- `DOLEFT` works on the left sub-tree
- `DORIGHT` works on the right sub-tree
- `DOTHIS` processes data for the current node

## Iterative Post-Order Traversal



- Stack status:
  - 0: DOLEFT
  - 1: DORIGHT
  - 3: DOSELF

## Iterative Tree Traversal

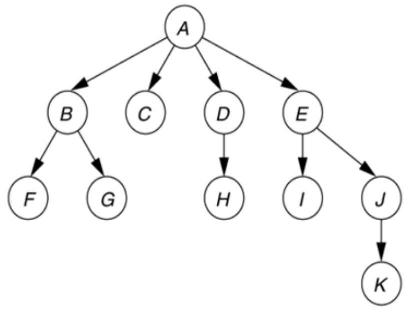
- Recursive implementations are easier to code but generally cost more memory
- Iterative methods are possible and save memory at the expense of tricky code
  - Pre-order and in-order follow the same idea as post-order
  - We can augment tree noes to have a reference to their parent
    - `class Node<T> {T data; Node left, right, parent;}`
  - This enables stack-less, iterative traversals with great cleverness

## Weiss' Traversals

```
BinaryTree<Integer> t = new BinaryTree<>();
// fill the tree
TreeIterator<T> itr = new PreOrder<Integer>(t);
for (itr.first(); itr.isValid(); itr.advance()) {
    System.out.println(" " + itr.retrieve());
}
```

- Weiss' traversals are implemented as iterators
  - More complex to understand, but generally easier to use
  - Play with some of these if you want more practice

## Breadth First Traversal




| Node | Height | Depth |
|---|---|---|
| *A* | 3 | 0 |
| *B* | 1 | 1 |
| *C* | 0 | 1 |
| *D* | 1 | 1 |
| *E* | 2 | 1 |
| *F* | 0 | 2 |
| *G* | 0 | 2 |
| *H* | 0 | 2 |
| *I* | 0 | 2 |
| *J* | 1 | 2 |
| *K* | 0 | 3 |

- Level-order traversal: visit the nodes from top to bottom, left to right
  - $A \to B \to C \to D \to E \to F \to G \to H \to I \to J \to K$
- If you're using an array, you can walk the array
- If you're using a linked list, you can use a queue

## Next Lecture

- Summary: tree basis
  - Definitions
  - Operations
  - Recursion review
- Next lecture: binary search trees
  - Reading: Chapter 19