

CS 310: Trees (Part II)

Connor Baker

February 21, 2019

Recursion Example

- Binary tree `.size()` counts the number of nodes in a tree
- Recursive helper: `.size(Node n)` that counts the number of nodes in a sub-tree rooted at the node including itself
 - Base case: the size of `null` is zero
 - Recursive case: Size of tree = `size(node.left) + size(node.right) + 1`

- We could implement this as

```
// Recursive method to count
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;
}
```

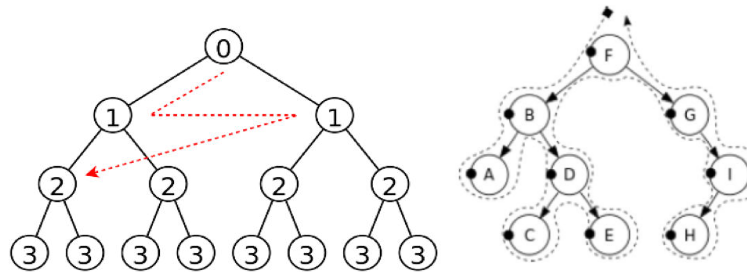
- Exercise: define a recursive function `.height(Node n)` that returns the height of a tree node
 - Length of the longest path from a node to a leaf
 - Leaf node height is zero

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

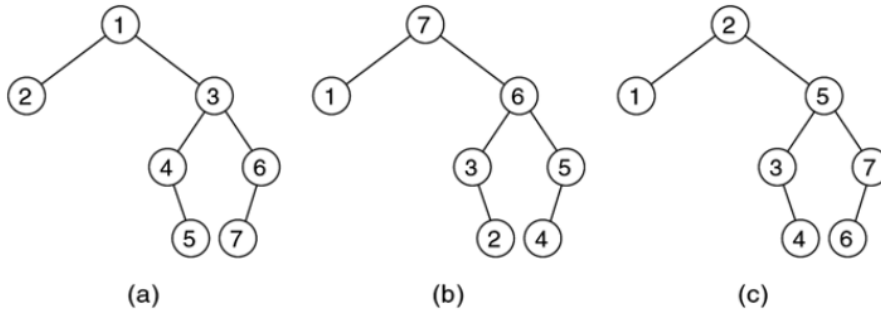
- Check Weiss 18.4: traversal implementation in an iterator
- Two common types:



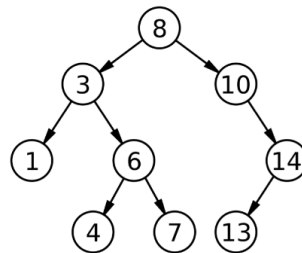
- (Left) *Breadth first search*: a search which proceeds by processing all nodes level by level, with those closest to the root processed first
- (Right) *Depth first search*: a search which proceeds by following a path all the way to a leaf and then backtracking

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 under an in-order traversal

Implementing Traversal for Binary Trees

```

1  class Node<T> {
2      T data;
3      Node<T> left, right;
4  }
5
6  inOrder(Node t) {
7      if (t == null) {
8          return;
9      }
10
11     inOrder(t.left);
12     print(t.data);
13     inOrder(t.right);
14 }
15
16 // to use
17 inOrder(this.root);
18
19 preOrder(Node t) {
20     if (t == null) {
21         return;
22     }
23     print(t.data);
24     preOrder(t.left);
25     preOrder(t.right);
26 }
27
28 postOrder(Node t) {
29     if (t == null) {
30         return;
31     }
32
33     postOrder(t.left);
34     postOrder(t.right);
35     print(t.data);
36 }

```

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: if we have a sorted binary tree, it gives us the elements in order

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

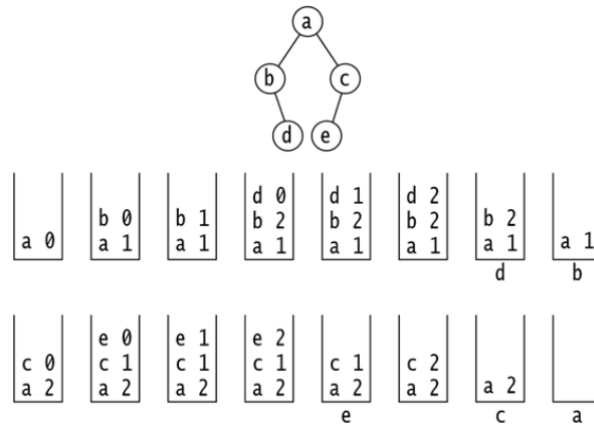
- Can we implement tree traversal without recursion?
 - Well... yes, but we need to maintain a stack (which is what recursion does for us)
- We can check the post-order traversal. Every node needs to go through three steps in order:
 - Process the node's left subtree
 - Process the node's right subtree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

- 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
 - 2: 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 nodes 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

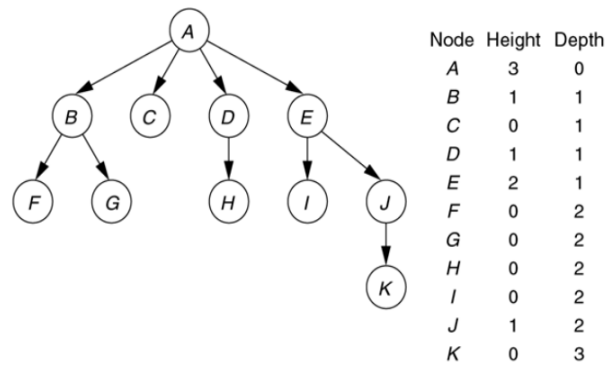
```

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



- Level-order traversal: visit the nodes from top to bottom, left to right
 - $A \rightarrow B \rightarrow C \rightarrow D \rightarrow E \rightarrow F \rightarrow G \rightarrow H \rightarrow I \rightarrow J \rightarrow K$
- 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: Hashing
 - Reading: Chapter 20