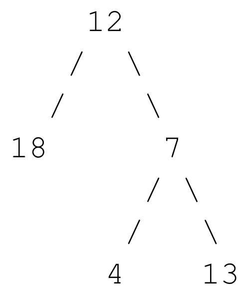
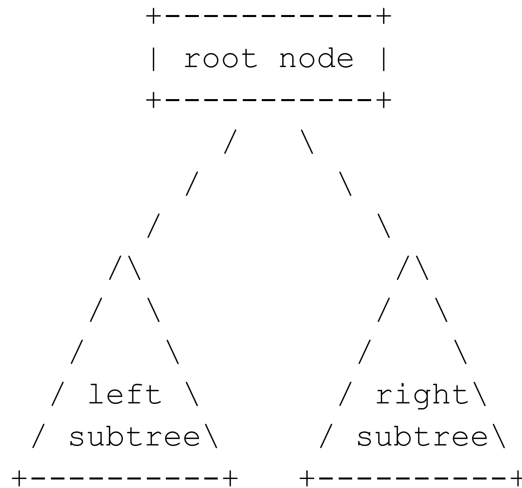
Lecture 17: Binary Trees (1)

- Midterm recap
  - Median: 66.5, Mean: 64.9
  - Harder than I expected, so +12 point curve
    - (won't be reflected on MyUW, but factored into course grades)
  - The answer key is posted
  - Regrade information will be posted soon
- Collaboration reminder
  - I have found a number of cheating cases, and I would like to remind everyone to review the collaboration policy
  - Gilligan's Island Rule: you can talk about it, but give 30 minutes after you stop talking before you write any code down
- Today we're introducing a new topic: binary trees
  - In computer science a tree looks kind of like this:



  - Notice that it's upside-down compared with a normal living tree
    - Like a family tree
  - We'll start with some terminology
  - We refer to the ROOT (the 12), LEAVES (the 18, 4, 13), and BRANCHES (the 12, 7)
    - The root is the base of the tree
    - Leaves have nothing beyond them
    - Branches are on the path from the root to the leaves (root is also a branch)
  - We also have terminology kind of like the family tree: PARENT and CHILD
    - Root is the ancestor of all other nodes
    - 12 is the parent of 18 and 7, 18 and 7 are 12's children
    - Each node has exactly one parent
    - We sometimes refer to 18 and 7 as SIBLINGS
  - Trees can be structured such that each root can have arbitrarily many children, but for this class, we will deal with BINARY TREES - each root has 2 children

- Now, for a more formal definition of a tree
    - We will define a tree recursively
        - What is the simplest possible type of tree you could have?
            - A one-element tree? A leaf?
            - Actually, a null tree! (empty)
        - And if it's not null, then it's a root node with left and right subtrees



    - This is very important to understand: each tree is composed of smaller trees
    - This will be very important when writing code to process trees
- Using this definition, what kind of trees can we draw?
    - Well, we can draw an empty tree
    - 1 node, two 2 node, 5 3 node...
- IntTreeNode class
    - It turns out that a tree is a lot like a linked list, but instead of one next, there are 2 nexts
    - The first constructor builds a leaf
    - The second constructor builds a branch with left and right subtrees
    - Just like in a linked list, we will terminate our branches with null - a leaf has null left and right subtrees
    - It's not well encapsulated, just like the ListNode - why was that ok?
        - Because we had a LinkedIntList class, and the client never got access to the internal nodes
    - So we'll do the same thing here - have an IntTree
        - Has one field - the overallRoot (to distinguish it from all the other roots)
        - Question: in one of our examples, how many TREES are there? How many roots?
- We'll get to building trees in a second, but first, how would we look at everything in a tree?
    - There's an easy way to look at everything in a linked list - just start at the beginning and go forward
    - But not so obvious for a tree
    - What ways could we go through all the values in a tree?
    - The idea is to TRAVERSE the tree, visiting each node only once
        - There's a fairly obvious way of doing this - look at the left subtree, then the right subtree
        - We have a left-right Western bias, so we always deal with the left first

    - But the question becomes: when do we deal with the root?
    - What are the choices?
        - Before we deal with the left side
        - In between left and right
        - After the right side
    - We call these preorder, inorder, and postorder, respectively
- Simulate these traversals on an example
    - You'll have a question like this on your final
    - The "sailboat" method: think of a sailboat sailing around the tree
        - Preorder: print when the sailboat gets to the left-hand side of the node
        - Inorder: print when the sailboat gets to the bottom of the node
        - Postorder: print when the sailboat gets to the right-hand side of the node
- Now let's write code for our IntTree class that will traverse the tree and print out the values
    - Write a method to print using a preorder traversal
    - You will almost ALWAYS have a public-private pair when writing these tree methods
        - Since trees are recursive structures, we need to write a recursive method
        - But that means that we need to know which subtree we're looking at
        - How do we tell the helper method where we are in the tree?
        - We pass it a "root" parameter
    - Start with the public method:

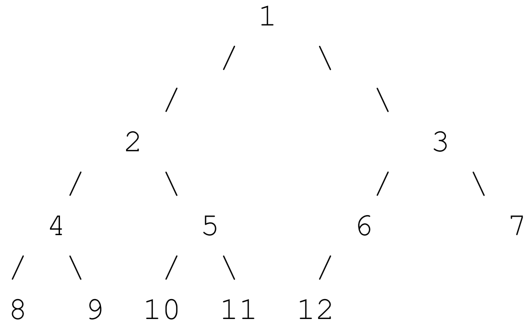
```
public void printPreorder() {
    System.out.print("preorder:");
    printPreorder(overallRoot);
    System.out.println();
}
```

    - For our private method, think recursively
        - What is the simplest type of tree to print?
            - The empty tree
        - How do we print the empty tree?
            - Do nothing
            - So actually, we reverse the condition
        - So if the tree is not empty, then we have a node with a left and a right and some data inside of it
        - We're doing a preorder traversal so we'll print out the root's data first (incl. space)
        - And what work do we have left to do? We need to print the left subtree and right subtree in a preorder manner
        - "if only I had a method..."

```
private void printPreorder(IntTreeNode root) {
    if (root != null) {
        System.out.println(" " + root.data);
        printPreorder(root.left);
        printPreorder(root.right);
    }
}
```

- And we're done!

- We can also do printInorder and printPostorder, just by changing the order of the statements in the private method
- Another task: write a constructor that builds up the following tree:



```
                  1
               /     \
            /           \
         2                 3
      /     \           /     \
     4       5         6       7
   /  \    /  \      /
  8    9  10   11   12
```

- Note that the children of a node have values 2*n and 2*n+1
- Code:

```
public IntTree2(int max) {
    if (max < 0) {
        throw new IllegalArgumentException("max: " + max);
    }
    overallRoot = buildTree(1, max);
}
private IntTreeNode buildTree(int n, int max) {
    if (n > max) {
        return null;
    } else {
        IntTreeNode left = buildTree(2 * n, max);
        IntTreeNode right = buildTree(2 * n + 1, max);
        return new IntTreeNode(n, left, right);
    }
}
```

- And finally, printSideways

```
public void printSideways() {
    printSideways(overallRoot, 0);
}


private void printSideways(IntTreeNode root, int level) {
    if (root != null) {
        printSideways(root.right, level + 1);
        for (int i = 0; i < level; i++) {
            System.out.print("    ");
        }
        System.out.println(root.data);
        printSideways(root.left, level + 1);
    }
}
```