Lecture 21
Trees III

- Exploring Binary Trees
  - Challenge 2
  - Challenge 3
  - Challenge 4
- structure Package
- Huffman Codes
Exploring Binary Trees
Challenge: Exploring Binary Trees (Part 2)

How can we determine the following value in a binary tree?

- The height of the tree.

Hints:
- Consider a recursive algorithm.
- Remember that a parent and child are not at the same level.
// Return the height of the tree rooted at node.  
// Note: A tree with one node has height 0.  
height(node)  
  // todo

// Main method: Run the algorithm from the tree’s root.  
answer = height(root)

Determining the height of a binary tree.
// Return the height of the tree rooted at node.
// Note: A tree with one node has height 0.
height(node)
   // Base case: the root node is null
   if node is null then
      return 0

   // Base case: the tree consists only of the root
   if node.left is null and node.right is null then
      return 0

   // Determine the height of the two subtrees.
   heightLeft = height(node.left)
   heightRight = height(node.right)

   // Return the maximum plus one.
   return max(heightLeft, heightRight) + 1

// Main method: Run the algorithm from the tree’s root.
answer = height(root)

Right: Determining the height of a binary tree. Left: The return values shown in each node.
Challenge: Exploring Binary Trees (Part 3)

How can we determine the following values in a binary tree?

- The total number of nodes.
- The smallest level that has a leaf.
- The number of left links that are used.

Think about this for 1 minute. Then discuss it with your neighbor for 4 minutes.

This binary tree has 8 total nodes. The smallest level of a leaf is 2. It has 5 left links in total.

What other quantities could we try to count?

Determining the number of left links in a binary tree.

// Return the number of left links in a binary tree that is // rooted at a given node.
left(node)
  // todo

// Main method: Run the algorithm from the tree’s root.
answer = left(root)
Determining the number of left links in a binary tree.

// Return the number of left links in a binary tree that is // rooted at a given node.
left(node)
    // todo

// Main method: Run the algorithm from the tree’s root.
answer = left(root)
Challenge: Exploring Binary Trees (Part 4)

How could we print out a nice text representation of a binary tree?

Think about this for 1 minute.

Questions:
- What do you interpret *nice* to mean?
- What values would you want to compute?
structure Package
Are you surprised by anything?

- Everything is a tree!
- There is not a separate class for nodes (as was the case with structure's linked lists).

```java
data package structure5;
import java.util.Iterator;

public class BinaryTree<E> {

    // The value associated with this node
    protected E val; // value associated with node

    // The parent of this node
    protected BinaryTree<E> parent; // parent of node

    // The left child of this node, or an "empty" node
    protected BinaryTree<E> left, right; // children of node

    // A one-time constructor, for constructing empty trees.
    public BinaryTree() {
        val = null;
        parent = null; left = right = this;
    }

    // Constructs a tree node with no children. Value of the node
    // and subtrees are provided by the user
    public BinaryTree(E value) {
        Assert.pre(value != null, "Tree values must be non-null.");
        val = value;
        right = left = new BinaryTree<E>();
        setLeft(left);
        setRight(right);
    }

    // Constructs a tree node with two children. Value of the node
    // and subtrees are provided by the user.
    public BinaryTree(E value, BinaryTree<E> left, BinaryTree<E> right) {
        Assert.pre(value != null, "Tree values must be non-null.");
        val = value;
        if (left == null) { left = new BinaryTree<E>(); }
        setLeft(left);
        if (right == null) { right = new BinaryTree<E>(); }
        setRight(right);
    }
}
```
Implementations for \texttt{size} (i.e. number of nodes) and \texttt{height} and more.

```java
// Computes the depth of a node. The depth is the path length
// from node to root
public int depth()
{
    if (parent() == null) return 0;
    return 1 + parent().depth();
}

// Returns true if tree is full. A tree is full if adding a node
// to tree would necessarily increase its height
public boolean isFull()
{
    if (isEmpty()) return true;
    if (left().height() != right().height()) return false;
    return left().isFull() && right().isFull();
}

// Returns true if tree is empty.
public boolean isEmpty()
{
    return val == null;
}
```
Huffman Codes
Encoding an Image

How can we encode an image in binary (i.e., in a file)?

- Assign a code word for each color.
- Write the code words for each pixel’s color in row major order (i.e., from left-to-right starting at the top row).

How should we assign the code words? Several options below.

1. Use 00000001 for yellow, 00000010 for red, etc.
   This works, but it is wasteful.

2. Use 0 for yellow, 1 for red, 10 for green, 11 for teal, etc.
   This is compact, but it results in a prefix problem.
   - Suppose that the file starts with 11.
     That could indicate two red pixels or one blue pixel.
     The problem is that code 1 is prefix of code 11.

3. Use binary strings of the same length. Length: ⌈log(n)⌉
   Use 000 for yellow, 001 for red, etc.
Example: Huffman Codes

Let's try to improve upon the 3rd encoding scheme from the prefix slide.
We want to represent each color using as few bits as possible.
- Frequent colors should use fewer bits.
Repeatedly merge the two lowest frequency nodes under a parent node.

- The left / right edges have label 0 / 1.
- The parent’s node is the sum of the frequencies.
Repeatedly merge the two lowest frequency nodes under a parent node.

- The left / right edges have label 0 / 1.
- The parent’s node is the sum of the frequencies.
Repeatedly merge the two lowest frequency nodes under a parent node.

- The left / right edges have label 0 / 1.
- The parent’s node is the sum of the frequencies.
Repeatedly merge the two lowest frequency nodes under a parent node.

- The left / right edges have label 0 / 1.
- The parent’s node is the sum of the frequencies.
Repeatedly merge the two lowest frequency nodes under a parent node.

- The left / right edges have label 0 / 1.
- The parent’s node is the sum of the frequencies.
Repeatedly merge the two lowest frequency nodes under a parent node.

- The left / right edges have label 0 / 1.
- The parent’s node is the sum of the frequencies.
Repeatedly merge the two lowest frequency nodes under a parent node.

- The left / right edges have label 0 / 1.
- The parent’s node is the sum of the frequencies.
The path to each color gives its encoding. e.g. green is 010.
The resulting image can then be encoded as below. (Bit saving occurs for the black and peach colors.)

1101100110110110110110110111111111111110...

Notes:
- We also need to store the codes and the image dimensions. Otherwise, this stream of colors could be interpreted as a 10-by-10 or 5-by-20 image, since $10 \cdot 10 = 5 \cdot 20 = 100$.
- This is an example of a greedy algorithm. You’ll see many more of these in CSCI 256.
How can we be sure that the bit stream is uniquely unencodable?

The code words satisfy the *prefix property* (i.e., no code word is the prefix of another code word). This is due to the fact that every color is stored in a leaf in this tree. For example, the codeword for yellow is $110$. Since it is in a leaf, there cannot be another code word starting with $110$. 

$$110\, 110\, 011\, 011\, 011\, 011\, 011\, 111\, 111\, 111\, 110\ldots$$