# CSC148, Lab #6

This document contains the instructions for lab number 6 in CSC148H. To earn your lab mark, you must actively participate in the lab. We mark you in order to ensure a serious attempt at learning, NOT to make careful critical judgments on the results of your work.

## General rules

We will use the same general rules as for the first lab (including pair programming).

## Overview

In this lab, you'll solve a neat problem using binary trees. It's from a high school programming competition. Dan's pretty sure — and his co-instructors agree — that he didn't know how to do this in high school! If you have time, there are other tree exercises that you can work on as well.

## Solving the Problem

### Understand the Problem

Read the specification of the Trick or Tree'ing problem that you'll solve.

Before doing any programming, we'd like you to demonstrate that you understand what is being asked. On paper, give three sample binary trees along with the output that should be produced for each tree. Now give the string that corresponds to each tree; these are the kinds of strings that you will end up reading from the `candy.txt` file. Show your work to your TA before you continue.

### Write Helper Functions

One reason that we like this problem is because it uses some natural helper functions. Download the `trick.py` file from the Labs page of the course website. There are several helper functions for you to write. We suggest this order:

- Begin by reading through the `BTNode` class that you will use. It's a nodes and references representation, but It's slightly different from the one in lecture. It also has a `__repr__` method so that we can print trees to the Python shell.
- After writing each helper function, run the program. The helper functions contain tests that will be run by `doctest` when the program is run.
- Switch driver and navigator after each function.
- Write the `tree_sum` helper function to calculate the total amount of candy in the leaves of the tree. Think recursively!
- Write the `tree_height` function to calculate the height of the tree. Again, a recursive solution is most appropriate.
- Write `tree_paths`. This calculates the total number of paths (roads) that are required to go from the root, to every house, and back to the root. This is almost, but not quite, the minimum number of paths required by the question.
- OK. We are now ready to solve an instance of the tree: given a binary tree `t`, we can use the helper functions to return the number of paths required and the total candy in the tree. Write the `solve` function to do this. Note that we still haven't done anything with input or output files, so we are not done. What we have accomplished so far is to solve the problem **given** a tree.
- Next, write `read_bt`. It is designed to take a line representing a tree, and return a `BTNode` object for that tree. These lines will ultimately come from `candy.txt`. This, again, will be a recursive function. **Hint**: you might consider writing a recursive helper function that returns not only the tree, but also the part of the string that remains.
- And finally, write `process`. Here, for the first time, we read directly from a file like `candy.txt` and thereby solve the original problem.

## Refactoring

If you have time: notice that `tree_height` and `tree_paths` are structurally quite similar. You can combine those into a single function that returns **both** the height and number of paths!

Also: if you're still not convinced that recursion makes life easier, take a look at `iterative.py`. That's what happens if you try to implement one of the functions without recursion!

## Trees, in General

Recall that not all trees are binary trees. We can have trees where nodes have 3, 4, or more children. The most general case is that we allow a tree node to have an arbitrary number of children.

Take a look at `remove.py`. Each node in these trees has an item and zero or more children. Notice how `left` and `right` attributes are now **not** sufficient to describe the tree. The constructor for `tree` therefore takes a **list** of children of a node.

Carefully read the `remove_equal` function and spend time interpreting the examples in the docstring. When you are ready, implement this function with your partner. **Hint**: a postorder-like traversal is likely easiest here. That is, consider first applying the function recursively on the children of a node before considering what to do with the node and its children.