# Project 5: Phileas Fogg Has a High-Stakes Conversation

## Problem

Expected Duration: 8 hours

You have probably heard of the famous story "Around the World in 80 Days" by Jules Verne, even if you've never actually read it. In our story, Mr. Fogg, has the conversation that launches him off on his perilous journey to circumnavigate the globe. In analyzing transcripts of conversations like this, intelligence and security agencies might do several kinds of pattern analyses. Scotland Yard did theirs and believing mistakenly that Fogg had robbed the Bank of England, sent detective Mr. Fix to apprehend and arrest him.

One basic analysis is letter frequency count, using a binary search tree to store our data. In CS 1400, you may recall, we learned how to do word counts using a dictionary—this problem is similar, only we are counting letters, not words, and we are building our own key, value data structure, not using the built-in dictionary type.

*Note: Yes, there are several ways in Python to do this exercise that don't require us to write much code of our own—but a lot of high-powered databases and search engines use some form of tree structure to store and retrieve dynamic data fast, and here we get a peek under the hood. This is, after all why you're in this class.*

Your job is to read the contents of the file and store character counts in the tree as follows. Read from the file character by character. If a character is not in the tree, insert it in its proper place using ordinal value and set its associated count value to 1. If it is in the tree, then increment the count by 1. Ignore whitespace and punctuation characters but count all others. Uppercase and lowercase letters are distinct. Organize the tree so that left characters are less than right characters.

In a real-world scenario, we would ask various analytical problem-related questions after building the tree, and several questions about the tree and its structure—but here, we do it with test code to emphasize correctness and automated checking. Each required operation will be tested at least once.

You will implement a binary search tree ADT using the specification below. The internal implementation details are left up to you, just make sure all operations are implemented and that your tree does the right thing and is complete.

The only requirements on main.py are:

1. Your main code must be in function main() with conditional execution of main.
2. main.py must implement a function make_tree, which takes no parameters but returns a tree constructed from the input file. Apart from being convenient for the application, the test code depends on having that function available from the main module. Don't change this!

We give you some test code so that you know how the BST is supposed to behave, and so that you know how you are doing before we grade. `test_bst.py`.

A few things to consider for implementation:

1. You can simplify the remove algorithm by always deleting a leaf node. If the root node or an interior node is deleted, copy the information from the appropriate leaf node, then delete the leaf.
2. The data at each node needs to account for both a letter and its count. You may use the Pair class provided in the starter code or create your own way.
3. You may NOT use a built-in dictionary as the core implementation of the tree.
4. You are welcome to implement any other operations you want that simplify the required operations.
5. Some tree algorithms are "self-balancing"—they rebalance the tree after every insert or delete, or if the left and right sibling height differ by more than 1. Some modern applications amortize the rebalancing operation by checking only periodically and rebalancing as needed, or when some other condition is true. This is what you will implement for this project, using the algorithm below.

## Rebalance Algorithm

The rebalancing algorithm you will implement is as follows:

1. do an in order traversal of the tree and write the node values out to a list. If you wish you can use a generator to easily create this list.
2. take the middle value as root
3. split the list in left and right halves, excluding the middle value
4. recursively rebuild the tree, using steps 2 and 3 until done.

When finished, the tree will be balanced.

### What to Submit

Submit in Canvas as `project5.zip`:

1. `main.py` → your main code that solves the problem above. It will not be tested.
2. `bst.py` → your BST implementation that WILL be tested by the test code.

### Binary Search Tree ADT (bst.py)

You will implement a BST that supports the following operations. Note that ALL operations return a meaningful value, and so could be chained where it makes sense.

- **is_empty**: Return True if empty, False otherwise.
- **size()**: Return the number of items in the tree.
- **height**: Return the height of the tree, which is the length of the path from the root to its deepest leaf.
- **add(item)**: Add item to its proper place in the tree. Return the modified tree.

- **remove(item)**: Remove item from the tree. Return the modified tree.
- **find(item)**: Return the matched item. If item is not in the tree, raise a ValueError.
- **inorder:** Return a list with the data items in order of in-order traversal.
- **preorder:** Return a list with the data items in order of pre-order traversal.
- **postorder:** Return a list with the data items in order of post-order traversal.
- **rebalance:** rebalance the tree. Return the modified tree.

**Grading (100 points)** pylint will be run on bst.py. Expected minimum score is 8.5.

Score is the sum of:

- Percentage of 11 test cases passed x 80 points
- min(Coding style score/8.5, 1) x 20 points
- Possible adjustment due to physical inspection of the code, to make sure you implemented things.