6.009 Second Practice Quiz 2 -- Spring 2017

This quiz is **open-book and closed-internet**. Feel free to use any physical materials you have brought, but you may not access resources online (except web.mit.edu/6.009). Proctors will be available to answer administrative questions and clarify specifications of coding problems, but they should not be relied on for coding help.

Solutions that hardcode answers will receieve no credit. Here, hardcoding refers to building in special code for the particular inputs and outputs from our test suite. Many normal and perfectly acceptable programming patterns lead to functions having special cases for particular inputs. For instance, recursive functions usually include base cases that check for particular inputs. In contrast, we consider it unacceptable hardcoding for one of the problems if one of your classmates who solved that problem successfully would read your code and say "huh, that's a rather long and specialized value to test the input against verbatim!".

There are two problems worth 10 points each. Each problem has five test cases, each worth two points.

You *must* submit your modified `quiz.py` via web.mit.edu/6.009 before the deadline in order to receive credit. This quiz assumes you have Python 3.5 (or a later version) installed on your machine.

The `resources` directory contains the **Python library documentation** in PDF form. Please **do not import any Python modules** to use as part of your solutions -- quizzes with `import` statements (or their equivalents!) will be given grades of 0.

As in the labs, `test.py` can be used to test your code in `quiz.py`. Remember that you can run specific tests by listing the test numbers, like so

```
python test.py 1 3 5      # run tests #1, #3, #5
```

Your score will be computed from the number of tests you pass. If you pass all the tests for a problem, you'll receive full credit. If you pass 2 of the 5 tests, you'll receive 40% of the credit for the problem. Note that to pass some of the tests in the time allotted, your solution may have to be reasonably efficient.

Problem 1: Prairie Dog Housing Lottery (tests 1-5)

Please implement the function `lottery(prairie_dogs, capacities)`, which assigns prairie dogs to available burrows. Not all prairie dogs are willing to live in all burrows; they have idiosyncratic individual preferences. Furthermore, each borrow can only fit so many prairie dogs. The first input value is a list with one element per prairie dog, where each element is itself a list of numbers (in no particular order), each number standing for a burrow that this prairie dog will accept. The second input value is a list giving burrow capacities. Indices in this list correspond to numbers from the prairie-dog-preference lists. If an assignment exists from prairie dogs to burrows, satisfying everyone's preferences, then return that assignment, as a list of numbers, following the same order as the original list. If no satisfactory assignment exists, return `None`.

Warning: there are solutions to this problem at a variety of sophistication levels. The simplest solutions will be too slow for the largest test cases! You'll need to apply some of the ideas from class to finish all the tests in time.

Examples:

`lottery([[2], [1], [0]], [1, 1, 1])` should return `[2, 1, 0]`.

`lottery([[0, 1], [1, 0], [0, 1]], [1, 1])` should return `None`.

`lottery([[0, 1], [2, 3], [4, 5], [0], [2], [4]], [1, 1, 1, 1, 1, 1])` should return `[1, 3, 5, 0, 2, 4]`.

(Note that our test-case `.out` files list multiple correct answers, for those test cases whose answers aren't uniquely determined.)

Problem 2: Advanced Forestry (tests 6-10)

In class, we have seen two styles of linked structures. With each "modification" to a structure, we can return a new version without modifying the original, or we can modify the original in-place. In this question, we will stick to the second kind, using modification.

We will be working with a peculiar combination of binary search trees and linked lists. Every node in a structure belongs to *both* a binary search tree and a linked list.

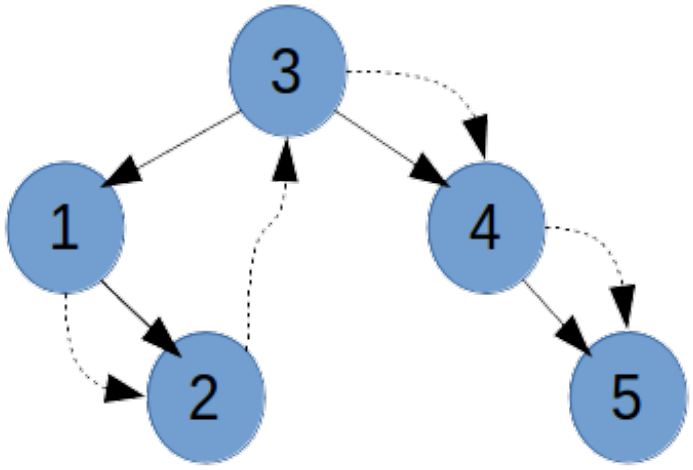
The binary-search-tree part is just as we saw in lecture. Recall that we may use these trees for representing sets of values. Some node is designated as the root, and each node has two children, left and right, either of which may be `None`. A node and all others reachable from it via left-child and right-child links is called a *subtree*. All data values in a node's left subtree must be *lower* than the node's own data value, and all data values in the right subtree must be *higher*.

Our wrinkle is to reuse the same nodes to form a *linked list*, taking us through all the nodes in *sorted order*. What added benefit does this wrinkle bring? It lets us answer *range queries* very efficiently: given a tree and data values `u` and `v`, we can quickly return a sorted list of all data values `w` in the tree such that `u <= w <= v`.

Every node of the tree is a **dictionary**. Here are the keys that every dictionary includes. (We augment this explanation with a picture afterward.)

1. `"data"` stores the data value that this node tells us is in the set.
2. `"left"` points to a subtree with data values *less than* `"data"`, or contains `None` if that subtree is empty.
3. `"right"` points to a subtree with data values *greater than* `"data"`, or contains `None` if that subtree is empty.
4. `"prev"` points to the tree node whose data value comes *immediately before* this one in sorted order, or contains `None` if this is the first element of the sorted list.
5. `"next"` points to the tree node whose data value comes *immediately after* this one in sorted order, or contains `None` if this is the last element of the sorted list.

Here is an example of one of these trees.



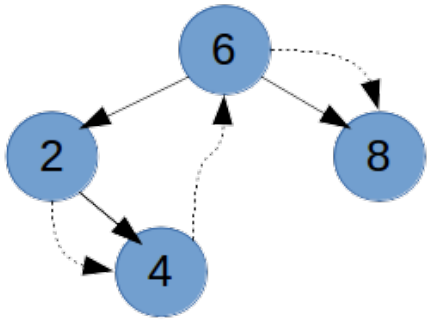
Nodes are circles with their data values inside. When number `n` appears inside node `A`, we have `A["data"] == n`. The root, with data value 3, appears highest. A node connects to its left child with a solid arrow going down and to the left, and solid rightward arrows connect to right children. So, node `A` with a solid left arrow to `B` and a solid right arrow to `C` has `A["left"] == B` and `A["right"] == C`. A dashed arrow from node `A` to node `B` indicates that `A["next"] == B`. Furthermore, we also record the information in the other direction: `B["prev"] == A`. Our tests will enforce exactly this convention for encoding nodes with dictionaries, so please follow it.

Your task is to write function `insert(tree, data)` implementing insertion into `tree` of a new node with data value `data`. Just as we learned in lecture, find the unique node `nd` of `tree` that currently has no left or right child, such that it is legal to insert the new node to the left or right of `nd`, respectively; and then go ahead and perform that insertion. It may also be necessary to fix up the `"next"` and `"prev"` pointers not just in the new node but also in some others from the original tree, to keep this linked list in sorted order. Note that this function doesn't return anything; we run it just for the modifications it makes to `tree`.

The test-case files (`.in` and `.out`) for this problem follow an unusually hard-to-read textual format, so we now include pictures of all the input trees. In the starter code, we also provide a function `print_tree` that generates a nicer rendering of a tree.

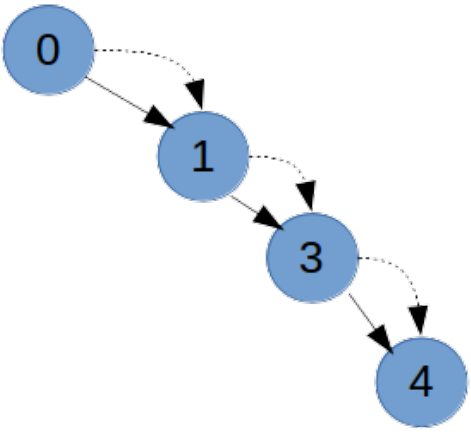
When `t` is the tree shown for Cases 7-8 and `u` is the tree given as an example output for Case 8 at the bottom, we have `insert(t, 7) == u`.

Cases 7-8

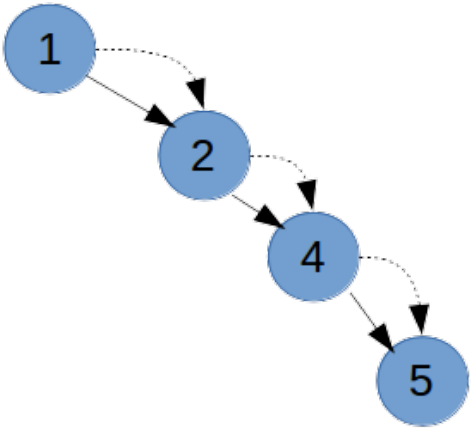


Case 6

Case 9



Case 10



Example output for Case 8