Homework 6

Due Friday, March 8th at 5:00 PM

Hey now, hey nooowwwww... Lizzie McGuire has just graduated junior-high, and as a savvy 2000s icon, she's trying to get a head start in the tech world. But Lizzie doesn't have to look far to find data structures and algorithms in her every day life. This is what dreams are made of, baby!

Handing In

To hand in a homework, go to the directory where your work is saved and run `cs0160_handin hwX` where X is the number of the homework. Make sure that your written work is saved as a .pdf file, and any Python problems are completed in the same directory or a subdirectory. You can re-handin any work by running the handin script again. We'll only grade your most recent submission. To install stencil Python files for a homework, run `cs0160_install hwX`. **You will lose points if you do not hand in your written work as a .pdf file.**

1 Written Problems

Problem 6.1

Lizzie McGuire is going to meet up with her BFF Gordo, but GET THIS: he's on the other side of town. With her totally impractical (but totally fashionable) platform flip flops, Lizzie needs to figure out the most efficient route to Gordy's current location. Help Lizzie OUT!

The four algorithms below all solve a problem of size n. What are the run times, in big-O notation, of each of these algorithms? Which is fastest? Note: If we do not specify a runtime for any step of the problem, you can assume that step is done in constant time. You do not have to write a formal proof for any of these algorithms, simply *show your work*.

1. An algorithm that recursively solves a subproblem of size $n - 1$ and performs a linear amount of work on the solution.
2. An algorithm that divides the problem in linear time into four subproblems, each of size $n/7$, recursively solves each subproblem, and then combines the solutions in constant time.
3. An algorithm that recursively solves three subproblems of size $n - 1$ and then combines the solutions in constant time.
4. An algorithm that divides the problem in constant time into eight subproblems of size $n/2$, recursively solves each subproblem, and then combines the solutions in $O(n^3)$ time.

5. Which of these algorithms is fastest?

Problem 6.2

Delete *this*!

Oops! Lizzie was just texting Gordo about her crush, Italian pop sensation Paolo, when she realized she'd accidentally sent all the texts to Paolo himself! Ruh Roh! Help Lizzie delete the singly-linked list of texts from Paolo's phone before he sees it.

You've got a singly-linked list. This means each node n has a next field (`n.next`) that points to the next node in the linked list but does not have a field pointing to the previous node in the list. If a node's `next == null`, then it's the final node in the list. You can assume there are no cycles in the list. Each node n also has a data field (`n.data`) where that node's data is stored.

Inserting new node n somewhere into the list is straightforward: you create a new node, make its `next` be n's `next` and then reset n's `next` to be the newly created node. Assuming we already have a reference to n, this takes $O(1)$ operations, independent of the length of the list.

(a) Write brief pseudocode for `deleteAfter(n)`, which takes in a node n and deletes the node after node n. If n is the last element in the list, it does nothing. The function must be $O(1)$. You may assume that the input(s) are valid (present in the list and not null) for all of these questions.

(b) Using your `deleteAfter` function, write pseudocode for `delete(n, head)`, which takes in a node n and a node *head* and removes n from the list, where *head* is the first node in the list. You can assume that `head != n`. The function must be $O(k)$, where k is the number of nodes in the list. (You should actually call `deleteAfter` somewhere in your pseudocode.)

(c) Write pseudocode for `smartDelete(n)`, which removes a node n from the list in $O(1)$ time. You can assume that n is not the last node in the list. (Hint: What differentiates two nodes? Their `data` field is their only identifying feature...)

2 Python Problems

Problem 6.3

Increment that

Lizzie's trouble-maker little brother, Matt, is determined to prank Lizzie to her breaking point. Help Matt calculate how many more pranks he'll need to pull to get his sister TOTALLY over it.

Implement a *recursive* Python function called `increment(number)` that takes in a stack of 0's and 1's representing a binary number k, and returns another stack representing the binary number $k + 1$. You should use a Python list as your stack representation, but you are only allowed to use the functions `len()`, `pop()` and `append()` (Python's version of `push()`).

Your code should be as neat and simple as possible (our solution is about 9 lines). You may mutate the input list directly. Note: there are several tricky edge cases to consider, so make sure to hand simulate your algorithm thoroughly!

In case you're unsure of how to add binary numbers, here are some links to help you out:

Wikihow.com/Add-Binary-Numbers
Courses.cs.vt.edu/AddingTwoBinaryNumbers

Remember to do this recursively!

Examples

- `increment([1,0,0,0])` → `[1,0,0,1]`
- `increment([1])` → `[1,0]`
- `increment([0,0,1])` → `[0,1,0]`

Testing

Write tests in the provided file `increment_test.py` to make sure your algorithm works. To help you generate test cases, we've provided a helper function called `strToList(string)` which takes in a string of 0's and 1's and returns a list that you can input to your `increment` function. Note: you may assume that if your input list is not null or empty, then it contains only 0's and 1's. Don't forget to check for invalid lists (None and `[]`), though!

Example

- `strToList("1010")` → `[1,0,1,0]`

Problem 6.4

Binary Tree Traversals

Thanks to your recursive increment function, Matt is sure he's got one prank to go, and he's saved the best for last: the ol' tree-traversin' prank. Help Matt traverse the trees outside the McGuire home, throwing acorns at his big sister until she's seriously had it.

Implement a preorder, inorder, postorder, and breadth-first traversal of a binary tree in Python. The `bt` object passed in to the traversals is a TA implementation of a binary tree.

Requirements

Each function should make a list of the nodes of a binary tree in the order of the respective traversal. Feel free to add helper methods to complete this task. You will use the TA binary tree implementation, and you are free to use Python's built-in Queue.

Functions

- preorder(bt)
- inorder(bt)
- postorder(bt)
- breadthfirst(bt)

Testing

You know the drill. Write tests in the provided file `traversals_test.py` and make sure your stuff works! Don't forget (again) to raise an error for invalid inputs. You may assume, however, that if your input is not null, it is a tree.