ECS 32B – Introduction to Data Structures
Homework 07

Due: Sunday, December 13, 2020, 5:00pm PST

Important:

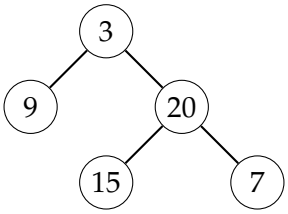
- The purpose of the homework is to practice **trees, graphs, and priority queues.** If you didn't meet the requirements as stated in the description, only half of the points will be given.
- For the purpose of testing, **the testing files contain two functions**: `serialize` and `deserialize`. The code and description is attached at the end of the document. You might need to understand what they do to be able to read the test cases.
- You may upload as many times as you want before the deadline. Each time the autograder will let you know how many of the test cases you have passed/failed. To prevent people from manipulating the autogader, the content of only half of the test cases are visible.
- The testing files contain the test cases for which Gradescope shows the content. You may use the testing files to test your code locally before submitting to Gradescope. However, it does not contain the rest of the test cases, for which Gradescope does not show the content. So passing all the test cases in the testing files does not necessarily mean you will get 100.
- Please **do not copy others' answers**. Autograder can detect similar code.

Problem 1: reconstructBT

Given the preorder and inorder traversals of a tree, construct the binary tree, assuming the values of all nodes in the tree are distinct. The preorder and inorder traversals are represented as Python lists, with the elements being the values of the nodes. The output is the root of the constructed binary tree.

Note: Assume the inputs are always valid, i.e., there is always a unique binary tree given the input preorder and inorder traversals.

Example:
Given
`preorder = [3, 9, 20, 15, 7]`
`inorder = [9, 3, 15, 20, 7],`
the constructed binary tree is



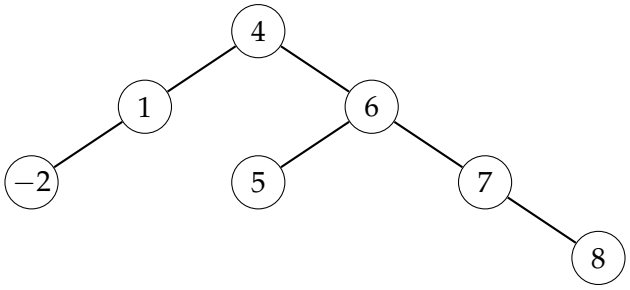
Problem 2: convertBSTtoGST

Given the root of a binary search tree (BST), convert it to a greater sum tree (GST) such that for each node in the BST, its value *val* is updated to the sum of all values greater than or equal to *val*.

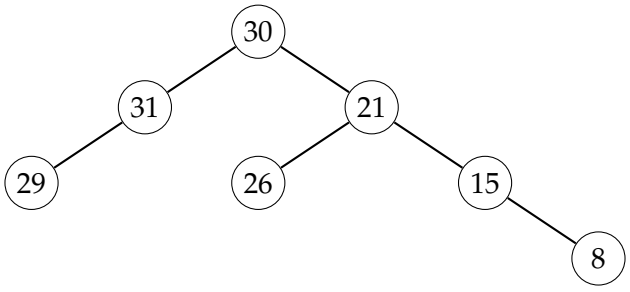
Note:

- Assume the values of all nodes are distinct.
- You may change the values of the input BST directly or may create a new tree as the output. Either way, you need to return the root of the GST.

Example:
Given the root of the following BST:



the GST is:



Follow up: not as a requirement for the homework, but the problem as a $O(n)$ solution (n is the number of nodes in the input BST). Can you figure it out?

Problem 3: kFrequent

Given a (possibly empty) Python list of integers and a non-negative integer k, find the k most frequent elements in the Python list using heap.

Note:

- k may not always be valid, i.e., the number of distinct elements in the Python list may be less than k.
- If two elements have the same frequency and they can not both be included, then only the smaller element is included.
- Return the elements in a Python list. The order of the elements does not matter.
- The runtime is $O(nlog\ min(k, n))$. When not using heap, the runtime is $O(nlogn)$.
- Feel free to use Python's heapq library. Documentation: `https://docs.python.org/3/library/heapq.html`

Example:
Given $nums = [5, 3, 9, 10, 10, 6, 6, 5]$ and $k = 2$, return $[6, 5]$ (or $[5, 6]$).

Problem 4: allPaths

A graph is **acyclic** if there is no cycle in it.

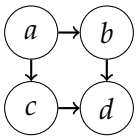
Given a **directed acyclic graph (DAG)** (with at least one edge) and two vertices v, u in the DAG, find all possible paths from v to u.

The input graph is represented as a Python list of all edges in the graph. Each edge is a pair (u, v) where u and v indicating there is an edge from vertex u to vertex v.

Note:

- A path is represented as an ordered Python list of vertices $[v_1, ..., v_n]$ such that for each $1 \leq i \leq n - 1$, there is an edge from v_i to v_{i+1}.
- The order of the possible paths in the output does not matter.
- The return type is a Python list of Python lists.
- This problem is to get you familiar with graphs. You do not have to use dynamic programming to improve its time complexity.

Example:
Given the following graph



all possible paths from a to d are

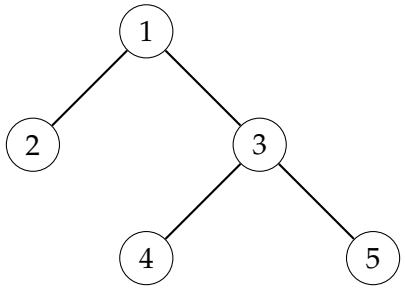
$$[a, b, d]$$
$$[a, c, d]$$

Serialization

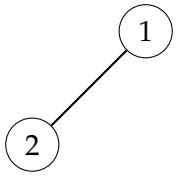
Serialization is the process of converting a data structure or object into a sequence of bits so that it can be stored in a file or memory buffer, or transmitted across a network connection link to be reconstructed, i.e., **deserialized**, later in the same or another computer environment.

The `serialize` and `deserialize` functions in the testing files are for the binary tree data structure. The `serialize` function encodes a binary tree to a list of strings; `deserialize` decodes such a string and reconstructs the encoded binary tree. The list of strings corresponds to the values of the nodes in the tree. The order is similar to what we get when we do a lever-order traversal on the tree, except that if a node is `None`, we write "n" instead of skipping it.

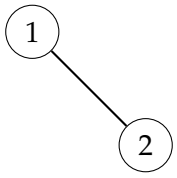
Example 1: The following tree is encoded as [“1”, “2”, “3”, “*n*”, “*n*”, “4”, “5”].



Example 2: The following tree is encoded as [“1”, “2”]



Example 3: The following tree is encoded as [“1”, “*n*”, “2”]



Note: The format of the encoding is similar to how Leetcode serializes a binary tree (`https://support.leetcode.com/hc/en-us/articles/360011883654-What-does-1-null-2-3-mean-in-binary-tree-representatio`