Assignment 3: Euler Tour Trees and Amortization

This problem set is all about amortized analysis. You'll design an amortized-efficient data structure for dynamic connectivity on trees, see how to build double-ended queues out of stacks, and get a feel for how amortization works in practice.

Working in Pairs

You are welcome to work on the problem sets either individually or in pairs. If you work in pairs, you should jointly submit a single assignment, which will be graded out of 24 points. If you work individually, the problem set will be graded out of 22 points, but we will not award extra credit if you earn more than 22 points.

Due Wednesday, April 23 at 2:15PM at the start of lecture.

Problem One: Stacking the Deque (6 Points)

A *deque* (***d***ouble-***e***nded ***que***ue, pronounced "deck") is a data structure that acts as a hybrid between a stack and a queue. It represents a sequence of elements that supports the following four operations:

- *deque*.**add-front**(*x*), which adds *x* to the front of the sequence.
- *deque*.**add-back**(*x*), which adds *x* to the back of the sequence.
- *deque*.**remove-front**(), which removes and returns the front element of the sequence.
- *deque*.**remove-back**(), which removes and returns the last element of the sequence.

Typically, you would implement a deque as a doubly-linked list. In functional languages like Haskell, Scheme, or ML, however, this implementation is not possible. In those languages, you would instead implement a deque using three stacks.

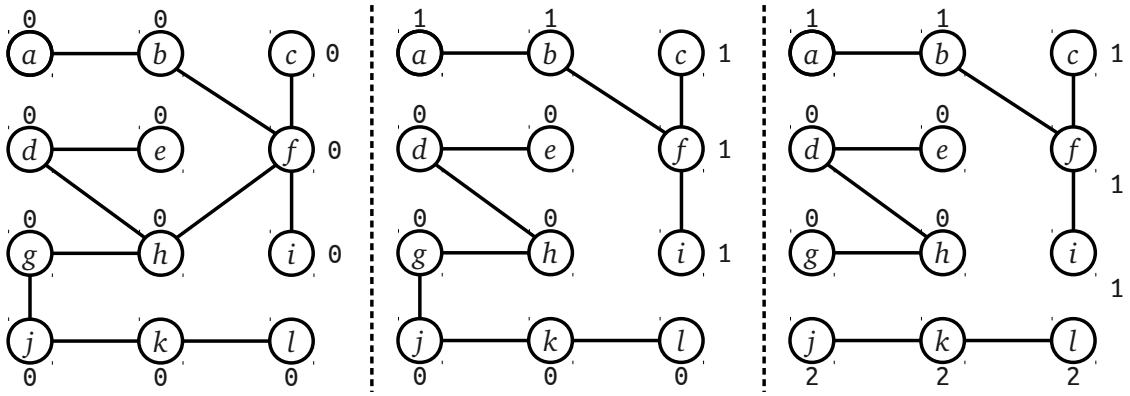
Design a deque that is implemented on top of three stacks. Each of the above operations should run in amortized time O(1). *(Hint: Store elements in two of the stacks and use the third stack as temporary storage space.)*

Problem Two: Decremental Connectivity in Trees (16 Points)

The *decremental connectivity problem on trees* is a variation on the dynamic connectivity problem on trees in which we are only allowed to *cut* edges from the tree and not *link* them. While we can solve this problem in time O(log n) per operation using Euler tour trees, there's a much simpler solution.

Suppose we are given as input a tree T represented as an adjacency list. We annotate each node with a *cluster ID*, which we'll use to identify which cluster each node belongs to. Initially, we'll set the cluster ID of each node to be 0, indicating that all nodes in the tree are a part of the same tree (since, initially, all nodes in T are connected). To implement *is-connected*(u, v), we just read off the cluster IDs of u and v and determine whether they're the same. This runs in time O(1).

The challenge is how to maintain cluster IDs efficiently as edges are deleted. We will implement *cut* as follows: to *cut* $\{u, v\}$, we delete $\{u, v\}$ from the forest. This will split some tree in the forest into two trees T_1 and T_2. We then choose a cluster ID that isn't already in use. Using either BFS or DFS, we visit all the nodes in the smaller of T_1 and T_2 and assign each node in the tree the new cluster ID. For example, here is a sample forest and the cluster IDs after executing *cut* $\{f, h\}$ and *cut* $\{g, j\}$:



i. **(3 Points)** Suppose that we *cut* $\{u, v\}$ from the forest, splitting some tree T into two trees T_1 and T_2, where $u \in T_1$ and $v \in T_2$. Design an algorithm that determines which of T_1 and T_2 is the smaller tree and does so in time O(min{$|T_1|$, $|T_2|$}).

Using the algorithm you developed in part (i), we can see that each *cut* operation might take time O(n) in the worst-case. However, using an amortized analysis, we can obtain a much better runtime.

ii. **(1 Point)** Prove that the cluster ID of each of the nodes in the graph change at most O(log n) times as *cut*s are performed.

iii. **(2 Points)** Show that this data structure can be constructed in O(n) amortized time such that each cut runs in amortized time O((n / k) log k), where k is the total number of cuts performed. This runtime nicely interpolates between O(n) and O(log n) as k increases.

iv. **(2 Points)** Show that this data structure can be constructed in amortized time O(n log n) such that each cut runs in amortized time O(1).

v. **(2 Points)** Explain why your results from (iii) and (iv) don't contradict one another.

vi. **(6 Points)** Using this new data structure for decremental connectivity, and without using Euler tour trees, show how to construct the Cartesian tree of a tree in time O(n log n). *(Hint: Start by sorting the edges and processing the edges in ascending order of weight. You might want to build up an auxiliary data structure associating some information with each cluster ID.)*

Problem Three: Course Feedback (2 Points)

We want this course to be as good as it can be and would really appreciate your feedback on how we're doing. For a free two points, please take a few minutes to answer the course feedback questions available at https://docs.google.com/forms/d/11DmGVSGDxdK10KCbnqPQW_Th_zg7cWS2L9wV4dZhaew/viewform. If you are submitting in a group, **please have each group member fill this out individually**.