1 Intro

1.1 The problem

We want to keep a collection of rooted trees under the following operations:

1. root(v): given a node v, find the root of the tree that contains v
2. link(u, v): make the tree rooted at u the child of node v (assumes that u is a root and that v is in a different tree from u)
3. cut(v): cut the subtree rooted at v off from v’s parent
4. depth(v): return the depth of v

Has applications to max flow (reduces time for finding blocking flows in Dinic’s algorithm), and LCA in dynamic trees (i.e. trees that can change)

1.2 A naive algorithm

For each node v, store v’s parent.

What’s the running time? Link and cut are O(1), root and depth are O(n)

1.3 A note on binary search trees

If we use a BBST like a red-black tree, we have insertion, deletion, and lookup in O(log n) time. With some work, we can also do the following two things in O(log n) time:

1. join(T₁, T₂): join the trees T₁, T₂
2. split(T₁, v): split T₁ into a tree of elements less than v and a tree of elements greater than v

2 Dynamic Trees

2.1 Preferred edges

With our naive algorithm, we are sad when the trees have large height. What are some things we can do to deal with trees with large height?
Heavy-light decomposition! Except it doesn’t work exactly here, because the trees can change, so the heavy and light edges can also change.

We use the notion of preferred and unpreferred edges, which are analogous to heavy and light edges from earlier. We call an edge \((u, v)\) from \(u\) to its parent \(v\) preferred if \(u\) is the most recently accessed child of \(v\). As with heavy-light edges, this partitions each tree into preferred paths.

2.2 Access

To implement each of the four operations, we first create an operation `access(v)`, which simply traces the path from \(v\) up to the root, making each unpreferred edge along the way preferred. This might cause other incident edges to the path to become unpreferred.

2.3 How to implement access?

We can’t simply store each preferred path as a line because it can change.

Key insight: Store each preferred path as an auxiliary binary tree, keyed by height in the original tree.

Now we can implement access through tree operations (on board)

Using access, we can implement each of the four operations.

2.4 Running time analysis

As stated earlier, tree operations take \(O(\log n)\) time.

We do a constant number of tree operations each time we hit an unpreferred edge on the way up, so we want to bound this number.

We can analyze this using heavy-light decomposition. The number of light unpreferred edges that we can hit on the way up is at most \(\log n\). For heavy edges, we amortize over all operations (say there are \(m\) of them): At the end, there can be at most \(n - 1\) heavy preferred edges. Also, once we hit this bound, each time a heavy edge becomes preferred, another heavy edge must become unpreferred. This can only happen through an access operation, so during this, a light edge becomes preferred. This is bounded by \(\log n\), so the total cost over \(m\) operations is \(O(m \log n + n)\). Assuming \(m\) is large, this is \(O(\log n)\) amortized per operation, or a \(O(\log^2 n)\) total cost.

Using splay trees, we can make this \(O(\log n)\) amortized, which is pretty crazy.