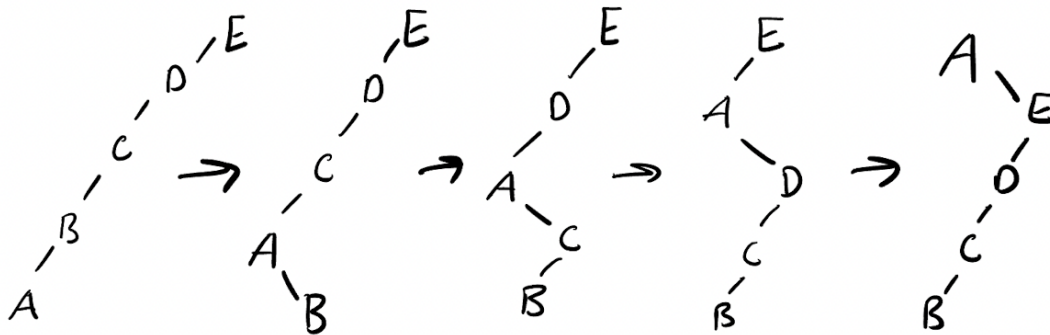


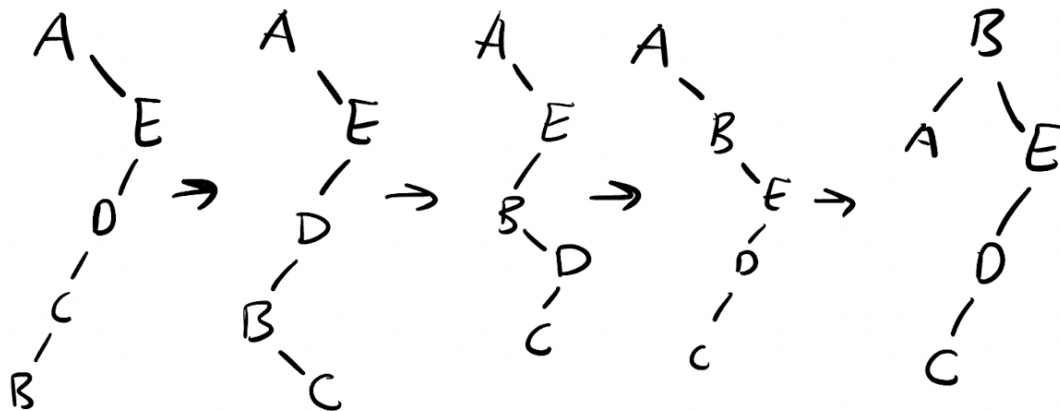
Problem 1. In order to efficiently answer LCA queries, we want to store the binary ancestors of each given node k , that is the 1st ancestor (the parent), the 2nd ancestor (the grandparent), the 4th ancestor, the 8th ancestor, and so on until there are no more binary ancestors. There are at most $O(\log n)$ such binary ancestors for a given node, so the total time for preprocessing takes $O(n \log n)$.

For the query step, suppose that we have two nodes x, y which are at different depths. Assume without loss of generality that x is deeper than y ; we can use the pre-processed binary ancestors of x to jump up the chain of ancestors until the depth of x 's ancestor is equal to y . We can accomplish this in $O(\log n)$ time by consistently jumping as far up the chain of ancestors without overshooting (that is, initializing the jump height to $k = \log n$ and decrementing if it surpasses y 's depth). If we consistently maximize the jump height without overshooting the depth of y , then there will be at most $\log n$ jumps before y 's depth is reached. Once the depths of x 's ancestor and y are equal, we then proceed to jump both x and y 's ancestors at the same time until we reach the least common ancestor via maximizing the jump height as before without overshooting. Overshooting can be determined by comparing the k -th ancestors of both nodes; if they are the same node, the jump height k must be decremented. Thus, the total time to determine the least common ancestor of x and y is $O(\log n)$.

Problem 2. (a) After splaying the leaf to the root, the old root becomes the right child of the leaf, which is now the new root. For $n = 5$, the process of splaying A to the root looks like this:

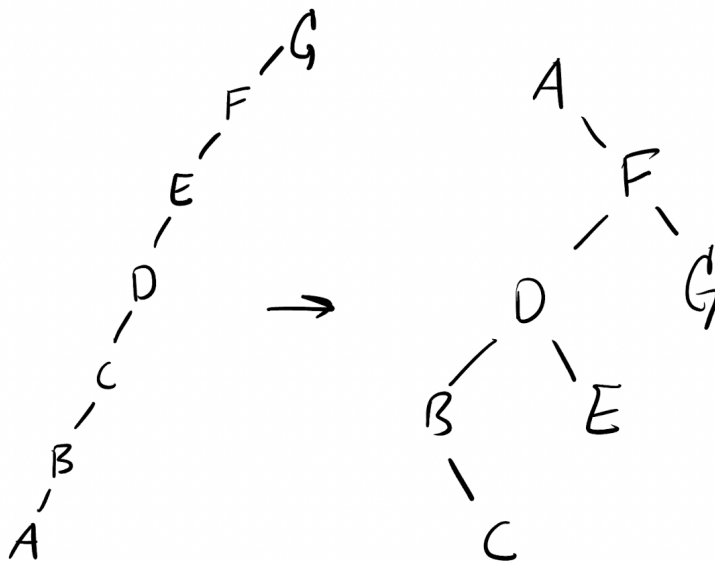


If we continue by splaying B to the root, we get:



In other words, each subsequent splay decreases the height of the tree by 1, so the next splay will take 1 less work. That is, A takes n work, B takes $n - 1$ work, C takes $n - 2$ work, and so on until $(n/2)$ -th node, which takes $n/2$ work (after this point, the original subtree no longer occupies the deepest levels of the tree, so the work to splay the leaf node is less than $n/2$). Thus, for an n -node degenerate tree there is a sequence of $n/2$ splays, where each splay takes at least $n/2$ work.

- (b) After splaying A to the root in a 7-node tree, we get:

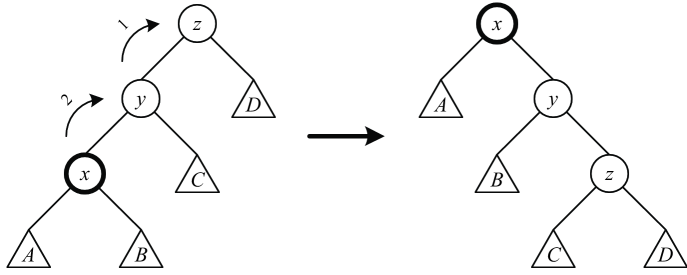
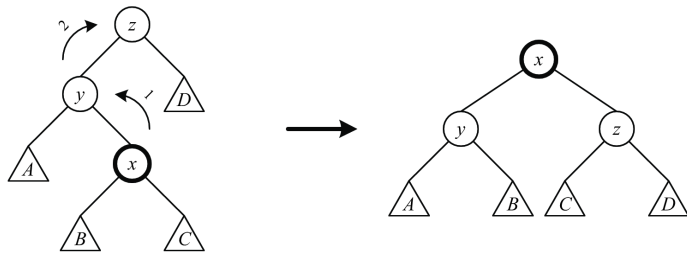


In other words, for each zig-zig operation, the right child of x becomes the left child of y , and x , y , and z swap orders. This leads to a chain of y - z pairs, with the last one attached to the root. Already, we notice that the state of the tree is in much better shape than in part (a), in which each node besides the leaf became deeper after the single rotations. In this case, the overall depth of the tree is much shallower due to the zig-zig operations bringing nodes closer to the root.

- (c) To make a specified node into a leaf, we can first degenerate the tree into a linked list where each node only has a left child and no right child (this can be accomplished by splaying the nodes in ascending order). Then we perform a zig-zig rotation on $\{x, y, z\}$ where z is the node we would like to turn into a leaf. Since neither y nor z have right children, the zig-zig rotation leaves z as a leaf node. If z does not have a grandchild, then we can instead degenerate the tree into a linked list with only right children (by splaying the nodes in descending order) and perform a zag-zag rotation on z to turn it into a leaf node using the same logic in reverse. For the base case where the tree consists of three or less nodes (i.e. z may not have grandchildren in either degenerate representation), z can be turned into a leaf node in two splays or less trivially.

Next, we will show that it is possible to restructure a tree with n nodes into another with splays via induction. For the base cases $n = 1$ and $n = 2$ it is clear that a tree can be turned into any other tree via at most 0 or 1 splays respectively. For the inductive step, we assume that it is possible to restructure a tree with n nodes into any other tree with n nodes, and try to show that an $n + 1$ node transformation is also feasible. Since a leaf is not affected by splays not on itself, we first turn a random node k into a leaf, and perform the transformation for the remaining n nodes. Since k is a leaf, the splay operations used in the transformation do not change the leaf status of k (so long as k is not the target of a splay - we can see this by noting that the structure of the subtrees A , B , C , and D do not change in either zig-zig

or zig-zag rotations). Thus we are left with two $n + 1$ size trees, where the only difference is the leaf node k . Let us denote the mismatched node in the tree we wish to imitate as k' . If k' is also a leaf node, then the process of turning k into a leaf would result in k naturally being in the same position as k' after the transformation splays. If k' is not a leaf node, we can perform the same hypothetical process of turning k' into a leaf, and repeat these steps in reverse for k so that k attains the position of k' before leafification. Thus, in either case the mismatched node problem is dealt with, closing the induction. Therefore, it is possible to restructure a tree with n nodes into another through a sequence of splays.

- Problem 3.** (a) Consider the case in which there is a sequence of almost-unbalanced triples along the search path where for each triple $\{x_i, y_i, z_i\}$, $9/10$ (and no more) of z_i 's descendants are below the corresponding grandchild x_i . In this case, the height of tree is $O(\log_{9/10} n) = O(\frac{\log n}{\log 9/10}) = O(\log n)$, which is also equal to the maximum number of triples along a given search path. Since a more balanced tree will have a shorter height, the number of balanced triples along any given search path is therefore bounded by $O(\log n)$.
- (b) **Zig-zig:** The size of x after the rotation is equal to the entire subtree, i.e. n , whereas it was at least $\frac{9}{10}n$ before the rotation by definition. Thus, the increase in the rank of x is at most $\log \frac{n}{10}$. On the other hand, the size of z after the rotation is at most $\frac{1}{10}n$. We can see this by noting that z 's children after the rotation are C and D , which comprised at most $1/10$ of z 's children before the rotation (since the children of x , A and B , comprised at least $9/10$). In addition, the size of y also decreases by some amount due to losing A as a descendant. Therefore, the decrease in rank of y and z is at least $\log \frac{9n}{10}$, so the total change in potential is at least
- $$\log \frac{n}{10} - \log \frac{9n}{10} = \log \frac{1}{9} \approx -3.1699$$
- which is enough to pay for the cost of the splay rotations.
- 
- Figure 1: After the rotation, z 's children are C and D (image from scribe notes)
- Zig-zag:** The increase in $r(x)$ is the same as the zig-zig case, $\log \frac{n}{10}$. Before the rotation, $s(y)$ is at least $\frac{9}{10}n$, so $s(y) + s(z) \geq \frac{19}{10}n$. After the rotation, the total size of y and z is at most n , i.e. the entire subtree minus x . Thus, the decrease in rank of y and z is $\log \frac{9n}{10}$, so the change in potential is again
- $$\log \frac{n}{10} - \log \frac{9n}{10} = \log \frac{1}{9}$$
- 
- Figure 2: Before the rotation, the sizes of x and y are overlapping (image from scribe notes)

- (c) The reasoning is the same regardless of which rotation type. Recall that the change in potential is given by $\Delta\phi = r'(x) + r'(y) + r'(z) - r(x) - r(y) - r(z)$. The $r'(x)$ and $r(z)$ terms cancel out because the rank of the root does not change, and $r(y) > r(x)$ since y is x 's parent before the rotation. We also know that since $r(z)$ was originally the root, $r(z) > r'(y), r'(z)$. Thus, we have

$$\begin{aligned}\Delta\phi &= r'(x) + r'(y) + r'(z) - r(x) - r(y) - r(z) \\ &\leq r'(y) + r'(z) - 2r(x) \\ &\leq 2(r(z) - r(x))\end{aligned}$$

for both zig-zig and zig-zag rotation types.

- (d) From part (b), we know that the cost of biased rotations is sufficiently paid for by the loss of potential. This leaves the (amortized) cost of the balanced rotations, which is given by

$$\text{amortized cost}_i = \text{real cost}_i + \Delta\phi$$

for the i -th rotation. From part (c), we know that the change in potential of any balanced rotation is bounded by $2(r(z) - r(x))$, and we have shown in part (a) that there are at most $O(\log n)$ such rotations. Additionally, note that for each subsequent rotation, $r(x_{i+1}) = r(z_i)$ since each x_i takes the position of z_i as root before becoming the next x_{i+1} for the following rotation, so the sum of potential changes telescopes. Thus, the total amortized cost of a sequence of balanced rotations is

$$\text{amortized cost} = \sum_i^{\log n} \text{cost}(\text{rotation}_i) = 2(r(\text{root}) - r(x))$$

If we set $w(x) = 1$ for each node in the tree, we get $r(\text{root}) = O(\log n)$, which therefore bounds the amortized cost of balanced rotations.

Problem 4. (a) Suppose we are searching for an element i that is the f_i -th most frequent element. We know that i is first encountered in some data structure S_k , where $2^{2^{k-1}} < f_i \leq 2^{2^k}$. This implies that $k - 1 < \log \log f_i \leq k$. Then, in order to find i we must search through $S_0, S_1, S_2, \dots, S_k$, where searching through S_j takes at most $O(\log 2^{2^j}) = O(2^j)$ time and the number of trees to search through is at most $k = \log \log f_i$. Thus, the total search time for element i is $\sum_{j=0}^{\log \log f_i} O(2^j) = O(2^{\log \log f_i})$, and since element i is accessed $p_i m$ times, the total access time for all elements is given by

$$\sum_{i=1}^m p_i m O(2^{\log \log f_i}) = m \sum_{i=1}^m p_i O(\log f_i)$$

Finally, note that if i is the f_i -th most frequent element, then there can be at most $1/p_i$ elements that are more frequent than or equally frequent to i (including i itself). This can be seen since if each element has a frequency of at least p_i , and there are more than $1/p_i$ such elements, the total sum of their frequencies would exceed 1. Thus, $f_i < 1/p_i$, so the total access time becomes

$$m \sum p_i O(\log 1/p_i)$$

- (b) For each S_j , we'll keep track of the access frequencies of each element unique to S_j (i.e. $\{i | i \in S_j, i \notin S_{j-1}\}$) in an auxiliary binary search tree T_j . On insert, given the access frequency p_i of the new element i , we will iterate through the data structures starting from S_0 and insert i into any S_j for which the minimum search frequency in T_j is less than p_i (and update T_j as well). During this process, some of the S_j may overflow (i.e. $\text{size}(S_j) > 2^{2^j}$), in which case we delete the minimum frequency element from T_j and the corresponding element from S_j . If i is inserted into the last S_k and it overflows, then create a new pair of trees S_{k+1} and T_{k+1} and copy over all the elements from S_k into S_{k+1} before deleting the minimum frequency element from S_k and T_k and inserting it into T_{k+1} .

Since the operations for constructing T_j are always matched by equivalent operations for S_j , the overall time for the construction of the auxiliary trees is $O(\log n)$ for each element. The cost of insertion is $O(\log n)$ for elements that do not overflow S_k , and $O(n \log n)$ for elements that do cause S_k to overflow and therefore require copying a new pair of trees. However, we note that once S_k overflows, it will not overflow for another $n - \sqrt{n}$ elements (since the previous tree contains $2^{2^{k-1}} = \sqrt{n}$ unique elements). Each of these non-overflow element insertions will cost $O(\log n)$ in the future when being inserted into the new tree, and therefore all of them together pay for the $O(n \log n)$ cost of the new tree. Thus, the amortized cost of insertion is $O(\log n)$, and the total cost of searches remains the same as in part (a) since the order of access frequency is preserved.

- (c) Rather than keeping track of access frequencies, we can keep track of access *counts*, so that when a new element is inserted, we initialize it with a count of 1, and when an element is accessed, we increase its count by 1. Insertion is still done as before in part (b), and accessing is done by updating the access count and possibly moving an element up (towards S_0) by inserting it to S_{j-1} and cascading the overflow similar to inserts. Thus, the cost of insert and access is $O(\log n)$ in both cases, maintaining the time costs of the previous part and thus the static optimality condition.
- (d) The working set theorem states that for m accesses to n elements, the total time is given by

$$O(n \log n + \sum_{j=1}^m \log t_j)$$

(where the j -th access is for element x_j and t_j denotes the number of distinct elements accessed since the previous access to x_j). We can accomplish this by keeping track of time since the last access for each element in the auxiliary trees T_j rather than the access frequencies, and inverting the order so that elements with more recent access times are higher in priority than elements with lower time since the previous access. The operations laid out in part (b) are then unchanged, and insert, access, and delete are done in $O(\log n)$ time, so populating the empty data structure with n items takes $O(n \log n)$ time. Next, we note that after an element i is accessed or inserted, it moves to the very start of the data structure at S_0 and slowly propagates backward with each subsequent access not to itself, so that by t_j accesses to other elements, element i is in $S_{\log \log t_j}$. Recalling that searching through S_k takes $O(\log 2^{2^k}) = O(2^k)$ time, the total access time is given by

$$\sum_{j=1}^m \sum_{k=1}^{\log \log t_j} O(2^k) = \sum_{j=1}^m O(2^{\log \log t_j}) = \sum_{j=1}^m O(\log t_j)$$

Thus, the total time for populating n items and accessing m times is $O(n \log n + \sum_{j=1}^m \log t_j)$, which satisfies the working set theorem.