1. A ***rope*** is a data structure that stores a string (that is, a sequence of characters) and that supports the following operations:

   - NewString(a) creates a new string of length 1 containing only the character a and returns a pointer to that string.
   - Concat(S, T) replaces the strings S and T (given by pointers) with the concatenated string ST, and returns a pointer to the new string.
   - Split(S, k) replaces the string S (given by a pointer) with the prefix $S[1..k]$ and the suffix $S[k+1..length(S)]$, and returns pointers to those two new strings. You can safely assume that $1 \le k \le length(S) - 1$.
   - Lookup(S, k) returns a copy of the kth character in string S (given by a pointer), or Null if the length of S is less than k.

   For example, we can build the strings `SPLAYTREE` and `UNIONFIND` with 18 calls to NewString and 16 calls to Concat. Further operations modify our collection of strings as follows:

operation	result	stored strings
Split(`SPLAYTREE`, 5)	`SPLAY`, `TREE`	**`SPLAY`**, **`TREE`**, `UNIONFIND`
Split(`UNIONFIND`, 3)	`UNI`, `ONFIND`	`SPLAY`, `TREE`, **`UNI`**, **`ONFIND`**
Concat(`UNI`, `SPLAY`)	`UNISPLAY`	**`UNISPLAY`**, `TREE`, `ONFIND`
Split(`UNISPLAY`, 5)	`UNISP`, `LAY`	**`UNISP`**, **`LAY`**, `TREE`, `ONFIND`
NewString(Lookup(`UNISP`, 5))	`P`	**`P`**, `UNISP`, `LAY`, `TREE`, `ONFIND`

   Except for NewString and Lookup, these operations are destructive; at the end of the sequence above, the string `UNISPLAY` is no longer stored anywhere in memory.

   One standard implementation of ropes stores each string in a splay tree, implicitly using the character positions as the search keys. Each node v stores two values, in addition to its left and right child pointers: $v.char$ is the corresponding character, and $v.size$ is the size of the subtree rooted at v. The rope for a single string of length n uses $O(n)$ space.

   - NewString: $O(1)$ worst-case and amortized time.
   - Lookup(S, k): Run a Select operation to find the target node, splay that node to the root, and return the root's character. This takes $O(\log|S|)$ amortized time.
   - Concat(S, T): Run Lookup(T, 1) (which splays the first symbol in T to the root of its splay tree), set $T.left \leftarrow S$, and return T. This takes $O(\log(|S| + |T|))$ amortized time.
   - Split(S, k): Run Lookup(S, k), set $S_1 \leftarrow S$ and $S_2 \leftarrow S.right$, set $S.right \leftarrow$ Null, and return S_1 and S_2. This takes $O(\log|S|)$ amortized time.

   Describe how to modify this data structure to support a new operation Reverse(S), which replaces a string S with its reversal ***in $O(1)$ worst-case and amortized time***. For example, Reverse(`ONFIND`) = `DNIFNO`. Achieving this time bound will require modifying some of the other rope update algorithms; carefully describe these modifications. The amortized times for all other operations should change by at most a small constant factor.

The remaining problems are for you play with on your own.
Discussion in office hours or on Discord is welcome, but don't submit solutions!

---

2. Define a *left spine* to be a binary tree in which no vertex has a right child. Let T be an arbitrary binary search tree with n vertices, with keys $1, 2, \ldots, n$, for some $n \geq 4$.

   (a) Prove that splaying the nodes of T in increasing order from 1 to n transforms T into a left spine. *[Hint: What does T looks like after the first i splays?]*

   (b) Show that it is possible to transform T into *any* other binary tree with n vertices using splay operations. How many splays do you need in the worst case? *[Hint: Find a sequence of splays that turn a particular node into a leaf, and then recurse.]*

   (c) Why did we require $n \geq 4$?

3. Let T be a binary tree with n vertices.

   (a) Prove that $\sum_v depth(v) = \Omega(n \log n)$.

   (b) Suppose that $\sum_v depth(v) = O(n \log n)$. Prove that $\max_v depth(v) = O(\sqrt{n \log n})$.

   (c) Show that the analysis in part (b) is tight; that is, for any integer n describe a binary search tree with n vertices such that $\sum_v depth(v) = \Theta(n \log n)$ and $\max_v depth(v) = \Theta(\sqrt{n \log n})$

4. In last Monday's lecture, we saw how to efficiently implement Split and Join operations on splay trees in $O(\log n)$ amortized time. This question asks you how to support the same operations in AVL trees in $O(\log n)$ *worst-case* time.

   (a) Suppose we are given two AVL trees $T_<$ and $T_>$, with a total of n vertices, such that every search key in $T_<$ is smaller than every search key in $T_>$. Describe how to join $T_<$ and $T_>$ into a new AVL tree containing all n vertices, in $O(\log n)$ time. Crucially, the two input trees $T_<$ and $T_>$ may have very different sizes.

   (b) Describe how to split an AVL tree T at a given key value k into two AVL trees, one tree $T_{<k}$ containing all nodes in T with search keys less than k, and the other tree $T_{>k}$ containing all nodes in T with search keys greater than k. You can assume no node has search key equal to k. *[Hint: Split T along the search path to k and then use part (a) to assemble $T_{<k}$ and $T_{>k}$. The analysis is the hard part.]*