1. Describe how to modify the splay-tree-based rope data structure (described in the homework handout) to support a new operation Reverse(S), which replaces a string S with its reversal ***in $O(1)$ worst-case and amortized time***. The amortized times for all other operations should change by at most a small constant factor.

Solution: We add a single boolean flag $v.rev$ to every node v, which indicates whether the subtree rooted at v should be considered reversed. The Reverse algorithm is almost trivial: To reverse a string S, we call the following function on the root of the tree representing S.

```
REVERSE(v):
  if v ≠ NULL
      v.rev ← ¬v.rev
```

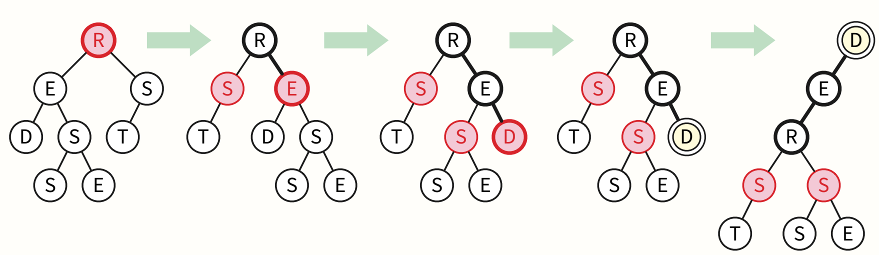
The simplest approach to modifying the other operations is to never let them see the reversal bits. In every operation, just before we read *any* field of *any* node v, we run the following algorithm. (To pronounce the function name "OkayFine" correctly, pretend that you are a petulant teenager whose parents have been nagging you for months to clean your room.)

```
OKAYFINE(v):
  if v.rev = TRUE
      v.rev ← FALSE
      swap v.left ⟷ v.right
      REVERSE(v.left)
      REVERSE(v.right)
```

(In C++, this code could be injected transparently by overloading the -> operator.) Calling OkayFine adds only $O(1)$ time to every node access, and therefore increases the cost of any other operation by only a constant factor.

The fact that we've implemented ropes using *splay* trees is utterly irrelevant. Precisely the same lazy-propagation strategy works for *any* balanced binary search tree that supports Split and Join in $O(\log n)$ (possibly amortized / expected) time.

In the following figure, a node is red if and only if its reversal bit is set to True. The tree on the left is a splay-rope for the string $S =$ STRESSED. The remaining steps show the execution of Lookup(S, 8), first performing the search and then splaying the target node up to the root.



■