1. Describe and analyze an algorithm that rebuilds an arbitrary n-node binary search tree T, some of whose nodes may be marked "dead", into a perfectly balanced binary search tree, in $O(n)$ time. The output tree should contain each of the unmarked nodes in T and nothing else.

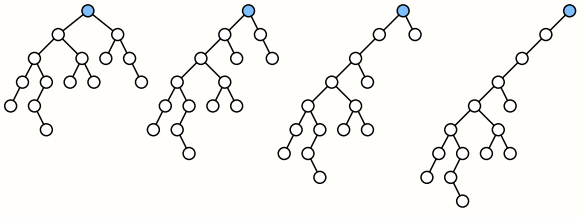
   For full credit, your algorithm should use only $O(\log n)$ space, *including the call stack* but not including the input tree T. *[Hint: Use rotations to reorganize the tree.]* For extra credit, describe an algorithm that uses only $O(1)$ additional space.

Solution: This can actually be done using only $O(1)$ additional space, by modifying an algorithm of Quentin Stout and Bette Warren [*Communications of the ACM* 29(9):902–908, 1986].(The original Stout/Warren algorithm did not consider "dead" nodes.) The algorithm works in four phases, called STRAIGHTEN, PURGE, FIXLOWEST, and COMPRESS. Each of these phases runs in $O(n)$ time, using only a constant number of additional pointers and integer variables.

```
REBALANCE(T):
  STRAIGHTEN(T.root)
  T.root ← PURGE(T.root)
  n ← size(T)
  ℓ ← n + 1 − 2^⌊lg n⌋
  if ℓ ≠ 0
      FIXLOWEST(T.root, n, ℓ)
      n ← n − ℓ
  while n > 1
      COMPRESS(T.root, n)
      n ← ⌊n/2⌋
```

STRAIGHTEN transforms the tree into a leftward path, in which nodes appear in reverse sorted order, by rotating any right child of a node on the left spine. Each rotation adds a node to the left spine of the tree (and decreases the number of nodes with right children), so line ($\star$) is executed at most n times. When the algorithm moves to the left, the previous node v is never touched again, so line (†) is executed at most n times. Thus, the overall running time of STRAIGHTEN is $O(n)$.

```
STRAIGHTEN(v):
  while v ≠ NULL
      while right(v) ≠ NULL
          v ← right(v)
          ROTATE(v)          (⋆)
      v ← left(v)            (†)
```



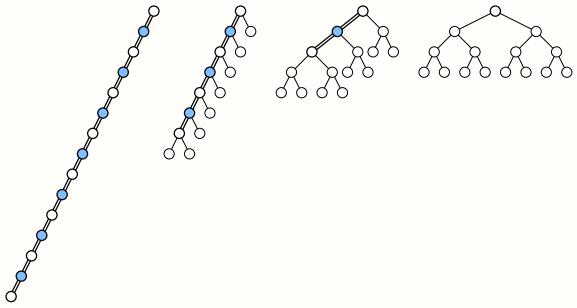
The first three iterations of STRAIGHTEN, with v at the root.

PURGE traverses the path produced by STRAIGHTEN, deleting all "dead" nodes, and returns a pointer to the new root. This clearly requires only $O(n)$ time and only $O(1)$ additional space.

```
PURGE(r):
  while r is "dead"
      v ← left(r); delete r; r ← v
  v ← r
  while left(v) ≠ NULL
      if left(v) is "dead"
          x ← left(v); left(v) ← left(x); delete x
      else
          v ← left(v)
  return r
```

COMPRESS compresses a left chain of $n = 2^k - 1$ nodes by roughly a factor of 2, by rotating every second node to the right. The results is a left chain of $\lfloor n/2 \rfloor = 2^{k-1} - 2$ nodes, each with a right child, plus the last node also has a left child. The main algorithm calls this subroutine $\lceil \lg n \rceil$ times; the overall running time of these calls is at most $\sum_{i=0}^{\lg n} O(n/2^i) = O(n)$.

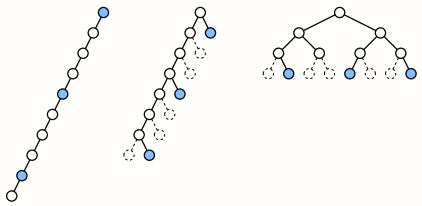
```
COMPRESS(v, n):
  for i ← 1 to ⌊n/2⌋
      v ← left(v)
      ROTATE(left(v))
      v ← left(v)
```



Three passes of COMPRESS perfectly balance a 15-node chain. Shaded nodes are rotated in each pass.

If the number of active nodes is one less than a power of 2, this is already enough to perfectly balance the tree. Otherwise, the last level of the perfectly balanced tree will have less than its full complement of nodes. Specifically, there must be $\ell = n - (2^{\lfloor \lg n \rfloor} - 1)$ nodes at the lowest level of the tree, at depth $\lfloor \lg n \rfloor$. FIXLOWEST locates these bottom-level leaves, spreading them out as evenly as possible, and rotates them into place. This algorithm clearly runs in $O(n)$ time.

```
FIXLOWEST(v, ℓ, n):
  sofar ← 0      ⟨⟨#bottom nodes so far⟩⟩
  next ← 0       ⟨⟨next bottom node⟩⟩
  for i ← 0 to n − ℓ − 1
      v ← left(v)
      if i = next
          ROTATE(v)
          v ← left(v)
          sofar ← sofar + 1
          next ← ⌊sofar · ℓ/(n − ℓ)⌋
```



FIXLOWEST fixes the three bottom nodes in a 10-node tree; two passes of COMPRESS then perfectly balance the tree.

■