Selected Solutions for Chapter 13: Red-Black Trees

Solution to Exercise 13.1-4

After absorbing each red node into its black parent, the degree of each node black node is

- 2, if both children were already black,
- 3, if one child was black and one was red, or
- 4, if both children were red.

All leaves of the resulting tree have the same depth.

Solution to Exercise 13.1-5

In the longest path, at least every other node is black. In the shortest path, at most every node is black. Since the two paths contain equal numbers of black nodes, the length of the longest path is at most twice the length of the shortest path.

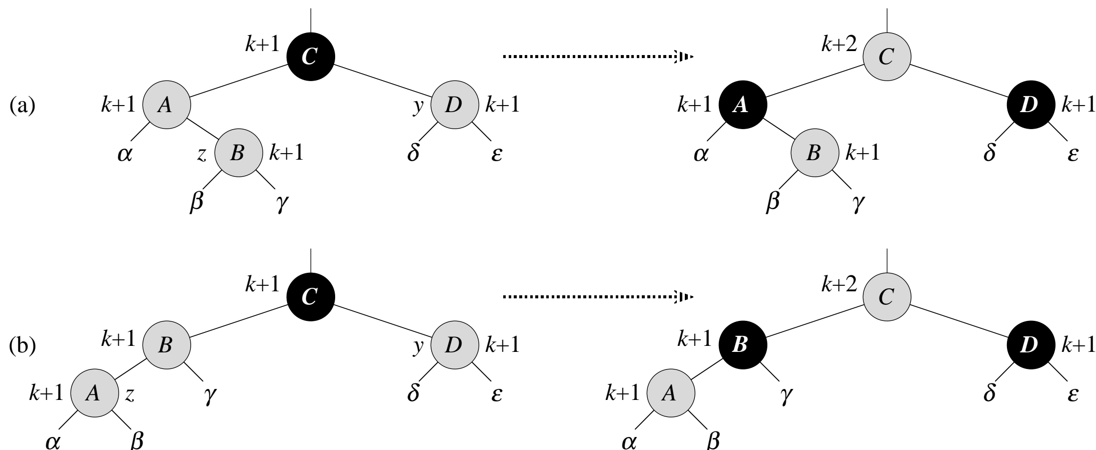
We can say this more precisely, as follows:

Since every path contains $\mathrm{bh}(x)$ black nodes, even the shortest path from x to a descendant leaf has length at least $\mathrm{bh}(x)$. By definition, the longest path from x to a descendant leaf has length $\mathrm{height}(x)$. Since the longest path has $\mathrm{bh}(x)$ black nodes and at least half the nodes on the longest path are black (by property 4), $\mathrm{bh}(x) \geq \mathrm{height}(x)/2$, so

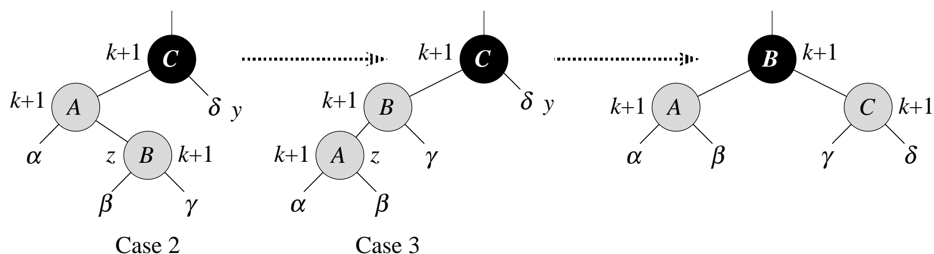
$$\text{length of longest path} = \mathrm{height}(x) \leq 2 \cdot \mathrm{bh}(x) \leq \text{ twice length of shortest path .}$$

Solution to Exercise 13.3-3

In Figure 13.5, nodes A, B, and D have black-height $k + 1$ in all cases, because each of their subtrees has black-height k and a black root. Node C has black-height $k + 1$ on the left (because its red children have black-height $k + 1$) and black-height $k + 2$ on the right (because its black children have black-height $k + 1$).



In Figure 13.6, nodes A, B, and C have black-height $k + 1$ in all cases. At left and in the middle, each of A's and B's subtrees has black-height k and a black root, while C has one such subtree and a red child with black-height $k + 1$. At the right, each of A's and C's subtrees has black-height k and a black root, while B's red children each have black-height $k + 1$.



Property 5 is preserved by the transformations. We have shown above that the black-height is well-defined within the subtrees pictured, so property 5 is preserved within those subtrees. Property 5 is preserved for the tree containing the subtrees pictured, because every path through these subtrees to a leaf contributes $k+2$ black nodes.

Solution to Problem 13-1

a. When inserting key k, all nodes on the path from the root to the added node (a new leaf) must change, since the need for a new child pointer propagates up from the new node to all of its ancestors.

When deleting a node, let y be the node actually removed and z be the node given to the delete procedure.

- If z has at most one child, it will be spliced out, so that all ancestors of z will be changed. (As with insertion, the need for a new child pointer propagates up from the removed node.)
- If z has two children, then its successor y will be spliced out and moved to z's position. Therefore all ancestors of both z and y must be changed.

Because z is an ancestor of y, we can just say that all ancestors of y must be changed.

In either case, y's children (if any) are unchanged, because we have assumed that there is no parent attribute.

b. We assume that we can call two procedures:

- MAKE-NEW-NODE(k) creates a new node whose *key* attribute has value k and with *left* and *right* attributes NIL, and it returns a pointer to the new node.
- COPY-NODE(x) creates a new node whose *key*, *left*, and *right* attributes have the same values as those of node x, and it returns a pointer to the new node.

Here are two ways to write PERSISTENT-TREE-INSERT. The first is a version of TREE-INSERT, modified to create new nodes along the path to where the new node will go, and to not use parent attributes. It returns the root of the new tree.

```
PERSISTENT-TREE-INSERT(T, k)
z = MAKE-NEW-NODE(k)
new-root = COPY-NODE(T.root)
y = NIL
x = new-root
while x ≠ NIL
    y = x
    if z.key < x.key
        x = COPY-NODE(x.left)
        y.left = x
    else x = COPY-NODE(x.right)
        y.right = x
if y == NIL
    new-root = z
elseif z.key < y.key
    y.left = z
else y.right = z
return new-root
```

The second is a rather elegant recursive procedure. The initial call should have $T.root$ as its first argument. It returns the root of the new tree.

```
PERSISTENT-TREE-INSERT(r, k)
if r == NIL
    x = MAKE-NEW-NODE(k)
else x = COPY-NODE(r)
    if k < r.key
        x.left = PERSISTENT-TREE-INSERT(r.left, k)
    else x.right = PERSISTENT-TREE-INSERT(r.right, k)
return x
```

c. Like TREE-INSERT, PERSISTENT-TREE-INSERT does a constant amount of work at each node along the path from the root to the new node. Since the length of the path is at most h, it takes $O(h)$ time.

Since it allocates a new node (a constant amount of space) for each ancestor of the inserted node, it also needs $O(h)$ space.

d. If there were parent attributes, then because of the new root, every node of the tree would have to be copied when a new node is inserted. To see why, observe that the children of the root would change to point to the new root, then their children would change to point to them, and so on. Since there are n nodes, this change would cause insertion to create $\Omega(n)$ new nodes and to take $\Omega(n)$ time.

e. From parts (a) and (c), we know that insertion into a persistent binary search tree of height h, like insertion into an ordinary binary search tree, takes worst-case time $O(h)$. A red-black tree has $h = O(\lg n)$, so insertion into an ordinary red-black tree takes $O(\lg n)$ time. We need to show that if the red-black tree is persistent, insertion can still be done in $O(\lg n)$ time. To do this, we will need to show two things:

- How to still find the parent pointers we need in $O(1)$ time without using a parent attribute. We cannot use a parent attribute because a persistent tree with parent attributes uses $\Omega(n)$ time for insertion (by part (d)).
- That the additional node changes made during red-black tree operations (by rotation and recoloring) don't cause more than $O(\lg n)$ additional nodes to change.

Each parent pointer needed during insertion can be found in $O(1)$ time without having a parent attribute as follows:

To insert into a red-black tree, we call RB-INSERT, which in turn calls RB-INSERT-FIXUP. Make the same changes to RB-INSERT as we made to TREE-INSERT for persistence. Additionally, as RB-INSERT walks down the tree to find the place to insert the new node, have it build a stack of the nodes it traverses and pass this stack to RB-INSERT-FIXUP. RB-INSERT-FIXUP needs parent pointers to walk back up the same path, and at any given time it needs parent pointers only to find the parent and grandparent of the node it is working on. As RB-INSERT-FIXUP moves up the stack of parents, it needs only parent pointers that are at known locations a constant distance away in the stack. Thus, the parent information can be found in $O(1)$ time, just as if it were stored in a parent attribute.

Rotation and recoloring change nodes as follows:

- RB-INSERT-FIXUP performs at most 2 rotations, and each rotation changes the child pointers in 3 nodes (the node around which we rotate, that node's parent, and one of the children of the node around which we rotate). Thus, at most 6 nodes are directly modified by rotation during RB-INSERT-FIXUP. In a persistent tree, all ancestors of a changed node are copied, so RB-INSERT-FIXUP's rotations take $O(\lg n)$ time to change nodes due to rotation. (Actually, the changed nodes in this case share a single $O(\lg n)$-length path of ancestors.)

- RB-INSERT-FIXUP recolors some of the inserted node's ancestors, which are being changed anyway in persistent insertion, and some children of ancestors (the "uncles" referred to in the algorithm description). There are at most $O(\lg n)$ ancestors, hence at most $O(\lg n)$ color changes of uncles. Recoloring uncles doesn't cause any additional node changes due to persistence, because the ancestors of the uncles are the same nodes (ancestors of the inserted node) that are being changed anyway due to persistence. Thus, recoloring does not affect the $O(\lg n)$ running time, even with persistence.

We could show similarly that deletion in a persistent tree also takes worst-case time $O(h)$.

- We already saw in part (a) that $O(h)$ nodes change.
- We could write a persistent RB-DELETE procedure that runs in $O(h)$ time, analogous to the changes we made for persistence in insertion. But to do so without using parent pointers we need to walk down the tree to the node to be deleted, to build up a stack of parents as discussed above for insertion. This is a little tricky if the set's keys are not distinct, because in order to find the path to the node to delete—a particular node with a given key—we have to make some changes to how we store things in the tree, so that duplicate keys can be distinguished. The easiest way is to have each key take a second part that is unique, and to use this second part as a tiebreaker when comparing keys.

Then the problem of showing that deletion needs only $O(\lg n)$ time in a persistent red-black tree is the same as for insertion.

- As for insertion, we can show that the parents needed by RB-DELETE-FIXUP can be found in $O(1)$ time (using the same technique as for insertion).
- Also, RB-DELETE-FIXUP performs at most 3 rotations, which as discussed above for insertion requires $O(\lg n)$ time to change nodes due to persistence. It also does $O(\lg n)$ color changes, which (as for insertion) take only $O(\lg n)$ time to change ancestors due to persistence, because the number of copied nodes is $O(\lg n)$.