Lecture Notes for Chapter 13: Red-Black Trees

Chapter 13 overview

Red-black trees

- A variation of binary search trees.
- ***Balanced***: height is $O(\lg n)$, where n is the number of nodes.
- Operations will take $O(\lg n)$ time in the worst case.

[These notes are a bit simpler than the treatment in the book, to make them more amenable to a lecture situation. Our students first see red-black trees in a course that precedes our algorithms course. This set of lecture notes is intended as a refresher for the students, bearing in mind that some time may have passed since they last saw red-black trees.

The procedures in this chapter are rather long sequences of pseudocode. You might want to make arrangements to project them rather than spending time writing them on a board.]

Red-black trees

A ***red-black tree*** is a binary search tree + 1 bit per node: an attribute *color*, which is either red or black.

All leaves are empty (nil) and colored black.

- We use a single sentinel, $T.nil$, for all the leaves of red-black tree T.
- $T.nil.color$ is black.
- The root's parent is also $T.nil$.

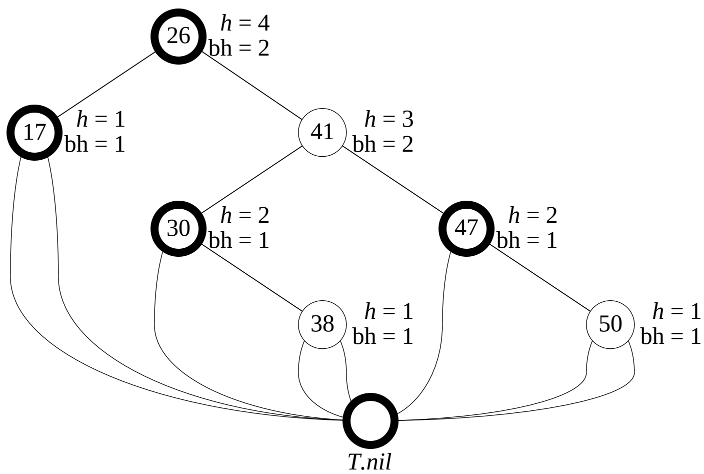
All other attributes of binary search trees are inherited by red-black trees (*key*, *left*, *right*, and p). We don't care about the key in $T.nil$.

Red-black properties

[Leave these up on the board.]

1. Every node is either red or black.
2. The root is black.
3. Every leaf ($T.nil$) is black.
4. If a node is red, then both its children are black. (Hence no two reds in a row on a simple path from the root to a leaf.)
5. For each node, all paths from the node to descendant leaves contain the same number of black nodes.

Example:



[Nodes with bold outline indicate black nodes. Don't add heights and black-heights yet. We won't bother with drawing $T.nil$ any more.]

Height of a red-black tree

- ***Height of a node*** is the number of edges in a longest path to a leaf.
- ***Black-height*** of a node x: $\text{bh}(x)$ is the number of black nodes (including $T.nil$) on the path from x to leaf, not counting x. By property 5, black-height is well defined.

[Now label the example tree with height h and bh *values.]*

Claim
Any node with height h has black-height $\geq h/2$.

Proof By property 4, $\leq h/2$ nodes on the path from the node to a leaf are red. Hence $\geq h/2$ are black. ■ (claim)

Claim
The subtree rooted at any node x contains $\geq 2^{\text{bh}(x)} - 1$ internal nodes.

Proof By induction on height of x.

Basis: Height of $x = 0 \Rightarrow x$ is a leaf $\Rightarrow \text{bh}(x) = 0$. The subtree rooted at x has 0 internal nodes. $2^0 - 1 = 0$.

Inductive step: Let the height of x be h and $\text{bh}(x) = b$. Any child of x has height $h - 1$ and black-height either b (if the child is red) or $b - 1$ (if the child is black). By the inductive hypothesis, each child has $\geq 2^{\text{bh}(x)-1} - 1$ internal nodes. Thus, the subtree rooted at x contains $\geq 2 \cdot (2^{\text{bh}(x)-1} - 1) + 1 = 2^{\text{bh}(x)} - 1$ internal nodes. (The $+1$ is for x itself.) ■ (claim)

Lemma
A red-black tree with n internal nodes has height $\leq 2\lg(n + 1)$.

Proof Let h and b be the height and black-height of the root, respectively. By the above two claims,

$$n \geq 2^b - 1 \geq 2^{h/2} - 1 .$$

Adding 1 to both sides and then taking logs gives $\lg(n + 1) \geq h/2$, which implies that $h \leq 2\lg(n + 1)$. ■ (theorem)

Operations on red-black trees

The non-modifying binary-search-tree operations MINIMUM, MAXIMUM, SUCCESSOR, PREDECESSOR, and SEARCH run in $O(\text{height})$ time. Thus, they take $O(\lg n)$ time on red-black trees.

Insertion and deletion are not so easy.

If we insert, what color to make the new node?

- Red? Might violate property 4.
- Black? Might violate property 5.

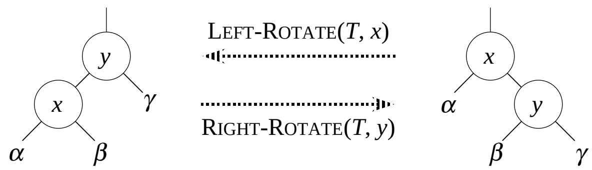
If we delete, thus removing a node, what color was the node that was removed?

- Red? OK, since we won't have changed any black-heights, nor will we have created two red nodes in a row. Also, cannot cause a violation of property 2, since if the removed node was red, it could not have been the root.
- Black? Could cause there to be two reds in a row (violating property 4), and can also cause a violation of property 5. Could also cause a violation of property 2, if the removed node was the root and its child—which becomes the new root—was red.

Rotations

- The basic tree-restructuring operation.
- Needed to maintain red-black trees as balanced binary search trees.
- Changes the local pointer structure. (Only pointers are changed.)

- Won't upset the binary-search-tree property.
- Have both left rotation and right rotation. They are inverses of each other.
- A rotation takes a red-black-tree and a node within the tree.



LEFT-ROTATE(T, x)

```
y = x.right                   // set y
x.right = y.left              // turn y's left subtree into x's right subtree
if y.left ≠ T.nil
    y.left.p = x
y.p = x.p                     // link x's parent to y
if x.p == T.nil
    T.root = y
elseif x == x.p.left
    x.p.left = y
else x.p.right = y
y.left = x                    // put x on y's left
x.p = y
```

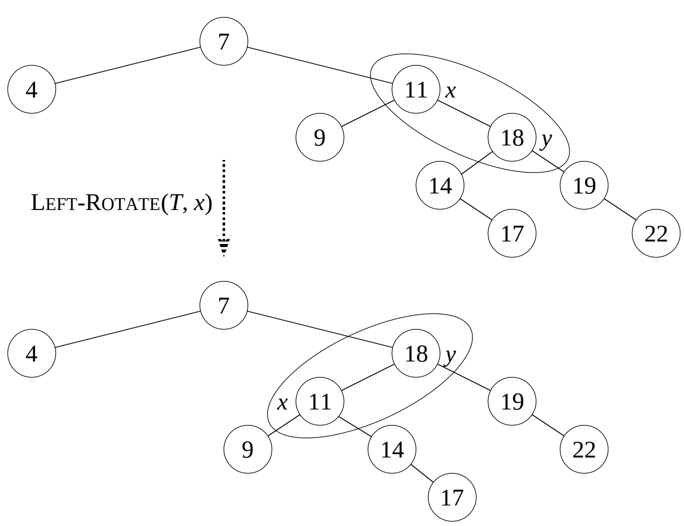
The pseudocode for LEFT-ROTATE assumes that

- $x.right \neq T.nil$, and
- root's parent is $T.nil$.

Pseudocode for RIGHT-ROTATE is symmetric: exchange *left* and *right* everywhere.

Example

[Use to demonstrate that rotation maintains inorder ordering of keys. Node colors omitted.]



- Before rotation: keys of x's left subtree $\leq 11 \leq$ keys of y's left subtree $\leq 18 \leq$ keys of y's right subtree.
- Rotation makes y's left subtree into x's right subtree.
- After rotation: keys of x's left subtree $\leq 11 \leq$ keys of x's right subtree $\leq 18 \leq$ keys of y's right subtree.

Time

$O(1)$ for both LEFT-ROTATE and RIGHT-ROTATE, since a constant number of pointers are modified.

Notes

- Rotation is a very basic operation, also used in AVL trees and splay trees.
- Some books talk of rotating on an edge rather than on a node.

Insertion

Start by doing regular binary-search-tree insertion:

```
RB-INSERT(T, z)
  y = T.nil
  x = T.root
  while x ≠ T.nil
      y = x
      if z.key < x.key
          x = x.left
      else x = x.right
  z.p = y
  if y == T.nil
      T.root = z
  elseif z.key < y.key
      y.left = z
  else y.right = z
  z.left = T.nil
  z.right = T.nil
  z.color = RED
  RB-INSERT-FIXUP(T, z)
```

- RB-INSERT ends by coloring the new node z red.
- Then it calls RB-INSERT-FIXUP because we could have violated a red-black property.

Which property might be violated?

1. OK.

2. If z is the root, then there's a violation. Otherwise, OK.
3. OK.
4. If $z.p$ is red, there's a violation: both z and $z.p$ are red.
5. OK.

Remove the violation by calling RB-INSERT-FIXUP:

```
RB-INSERT-FIXUP(T, z)
 while z.p.color == RED
     if z.p == z.p.p.left
         y = z.p.p.right
         if y.color == RED
             z.p.color = BLACK                           // case 1
             y.color = BLACK                             // case 1
             z.p.p.color = RED                           // case 1
             z = z.p.p                                   // case 1
         else if z == z.p.right
                 z = z.p                                 // case 2
                 LEFT-ROTATE(T, z)                       // case 2
             z.p.color = BLACK                           // case 3
             z.p.p.color = RED                           // case 3
             RIGHT-ROTATE(T, z.p.p)                      // case 3
     else (same as then clause with "right" and "left" exchanged)
 T.root.color = BLACK
```

Loop invariant:

At the start of each iteration of the **while** loop,

a. z is red.
b. There is at most one red-black violation:
   - Property 2: z is a red root, or
   - Property 4: z and $z.p$ are both red.

[The book has a third part of the loop invariant, but we omit it for lecture.]

Initialization: We've already seen why the loop invariant holds initially.

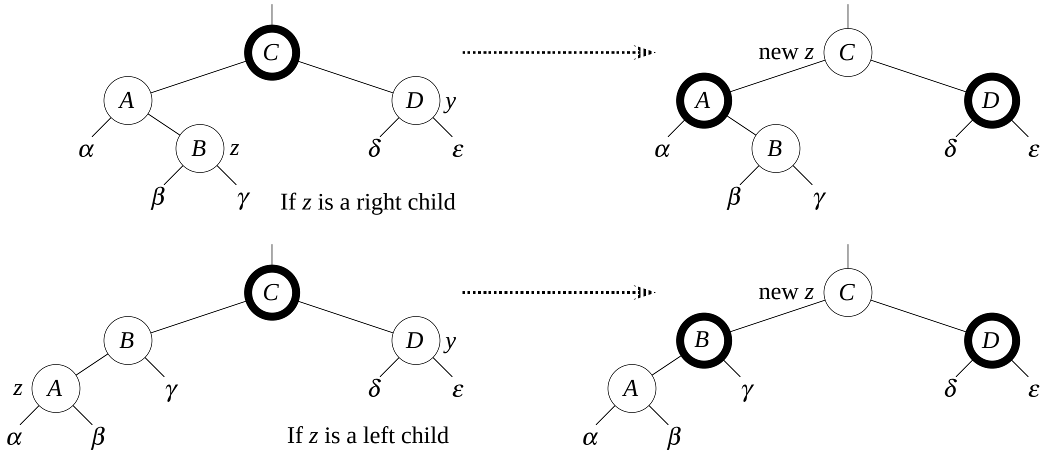
Termination: The loop terminates because $z.p$ is black. Hence, property 4 is OK. Only property 2 might be violated, and the last line fixes it.

Maintenance: We drop out when z is the root (since then $z.p$ is the sentinel $T.nil$, which is black). When we start the loop body, the only violation is of property 4.

There are 6 cases, 3 of which are symmetric to the other 3. The cases are not mutually exclusive. We'll consider cases in which $z.p$ is a left child.

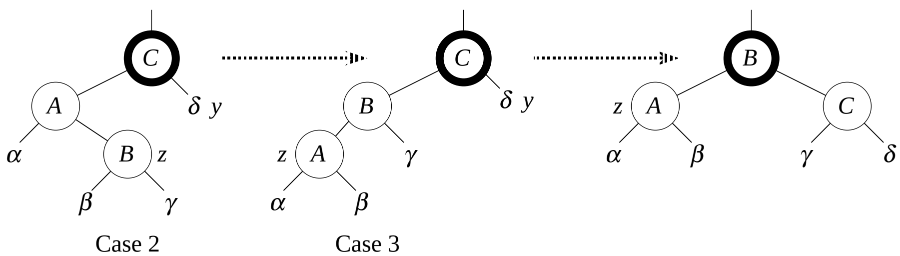
Let y be z's uncle ($z.p$'s sibling).

Case 1: y is red



- $z.p.p$ (z's grandparent) must be black, since z and $z.p$ are both red and there are no other violations of property 4.
- Make $z.p$ and y black $\Rightarrow$ now z and $z.p$ are not both red. But property 5 might now be violated.
- Make $z.p.p$ red $\Rightarrow$ restores property 5.
- The next iteration has $z.p.p$ as the new z (i.e., z moves up 2 levels).

Case 2: y is black, z is a right child



- Left rotate around $z.p$ $\Rightarrow$ now z is a left child, and both z and $z.p$ are red.
- Takes us immediately to case 3.

Case 3: y is black, z is a left child

- Make $z.p$ black and $z.p.p$ red.
- Then right rotate on $z.p.p$.
- No longer have 2 reds in a row.
- $z.p$ is now black $\Rightarrow$ no more iterations.

Analysis

$O(\lg n)$ time to get through RB-INSERT up to the call of RB-INSERT-FIXUP.

Within RB-INSERT-FIXUP:

- Each iteration takes $O(1)$ time.
- Each iteration is either the last one or it moves z up 2 levels.
- $O(\lg n)$ levels $\Rightarrow$ $O(\lg n)$ time.
- Also note that there are at most 2 rotations overall.

Thus, insertion into a red-black tree takes $O(\lg n)$ time.

Deletion

[Because deletion from a binary search tree changed in the third edition, so did deletion from a red-black tree. As with deletion from a binary search tree, the node z deleted from a red-black tree is always the node z passed to the deletion procedure.]

Based on the TREE-DELETE procedure for binary search trees:

```
RB-DELETE(T, z)
 y = z
 y-original-color = y.color
 if z.left == T.nil
     x = z.right
     RB-TRANSPLANT(T, z, z.right)
 elseif z.right == T.nil
     x = z.left
     RB-TRANSPLANT(T, z, z.left)
 else y = TREE-MINIMUM(z.right)
     y-original-color = y.color
     x = y.right
     if y.p == z
         x.p = y
     else RB-TRANSPLANT(T, y, y.right)
         y.right = z.right
         y.right.p = y
     RB-TRANSPLANT(T, z, y)
     y.left = z.left
     y.left.p = y
     y.color = z.color
 if y-original-color == BLACK
     RB-DELETE-FIXUP(T, x)
```

RB-DELETE calls a special version of TRANSPLANT (used in deletion from binary search trees), customized for red-black trees:

RB-TRANSPLANT(T, u, v)

```
if u.p == T.nil
    T.root = v
elseif u == u.p.left
    u.p.left = v
else u.p.right = v
v.p = u.p
```

Differences between RB-TRANSPLANT and TRANSPLANT:

- RB-TRANSPLANT references the sentinel $T.nil$ instead of NIL.
- Assignment to $v.p$ occurs even if v points to the sentinel. In fact, we exploit the ability to assign to $v.p$ when v points to the sentinel.

RB-DELETE has almost twice as many lines as TREE-DELETE, but you can find each line of TREE-DELETE within RB-DELETE (with NIL replaced by $T.nil$ and calls to TRANSPLANT replaced by calls to RB-TRANSPLANT).

Differences between RB-DELETE and TREE-DELETE:

- y is the node either removed from the tree (when z has fewer than 2 children) or moved within the tree (when z has 2 children).
- Need to save y's original color (in *y-original-color*) to test it at the end, because if it's black, then removing or moving y could cause red-black properties to be violated.
- x is the node that moves into y's original position. It's either y's only child, or $T.nil$ if y has no children.
- Sets $x.p$ to point to the original position of y's parent, even if $x = T.nil$. $x.p$ is set in one of two ways:
  - If z is not y's original parent, $x.p$ is set in the last line of RB-TRANSPLANT.
  - If z is y's original parent, then y will move up to take z's position in the tree. The assignment $x.p = y$ makes $x.p$ point to the original position of y's parent, even if x is $T.nil$.
- If y's original color was black, the changes to the tree structure might cause red-black properties to be violated, and we call RB-DELETE-FIXUP at the end to resolve the violations.

If y was originally black, what violations of red-black properties could arise?

1. No violation.
2. If y is the root and x is red, then the root has become red.
3. No violation.
4. Violation if $x.p$ and x are both red.
5. Any simple path containing y now has 1 fewer black node.
   - Correct by giving x an "extra black."
   - Add 1 to count of black nodes on paths containing x.
   - Now property 5 is OK, but property 1 is not.

- x is either ***doubly black*** (if $x.color = $ BLACK) or ***red & black*** (if $x.color = $ RED).
- The attribute $x.color$ is still either RED or BLACK. No new values for *color* attribute.
- In other words, the extra blackness on a node is by virtue of x pointing to the node.

Remove the violations by calling RB-DELETE-FIXUP:

```
RB-DELETE-FIXUP(T, x)
 while x ≠ T.root and x.color == BLACK
     if x == x.p.left
         w = x.p.right
         if w.color == RED
             w.color = BLACK                                   // case 1
             x.p.color = RED                                   // case 1
             LEFT-ROTATE(T, x.p)                               // case 1
             w = x.p.right                                     // case 1
         if w.left.color == BLACK and w.right.color == BLACK
             w.color = RED                                     // case 2
             x = x.p                                           // case 2
         else if w.right.color == BLACK
                 w.left.color = BLACK                          // case 3
                 w.color = RED                                 // case 3
                 RIGHT-ROTATE(T, w)                            // case 3
                 w = x.p.right                                 // case 3
             w.color = x.p.color                               // case 4
             x.p.color = BLACK                                 // case 4
             w.right.color = BLACK                             // case 4
             LEFT-ROTATE(T, x.p)                               // case 4
             x = T.root                                        // case 4
     else (same as then clause with "right" and "left" exchanged)
 x.color = BLACK
```

Idea

Move the extra black up the tree until

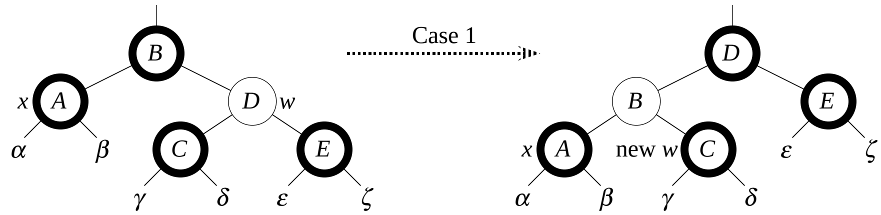
- x points to a red & black node $\Rightarrow$ turn it into a black node,
- x points to the root $\Rightarrow$ just remove the extra black, or
- we can do certain rotations and recolorings and finish.

Within the **while** loop:

- x always points to a nonroot doubly black node.
- w is x's sibling.
- w cannot be $T.nil$, since that would violate property 5 at $x.p$.

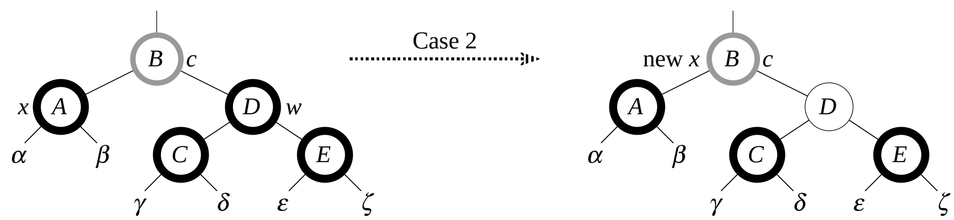
There are 8 cases, 4 of which are symmetric to the other 4. As with insertion, the cases are not mutually exclusive. We'll look at cases in which x is a left child.

Case 1: w is red



- w must have black children.
- Make w black and $x.p$ red.
- Then left rotate on $x.p$.
- New sibling of x was a child of w before rotation $\Rightarrow$ must be black.
- Go immediately to case 2, 3, or 4.

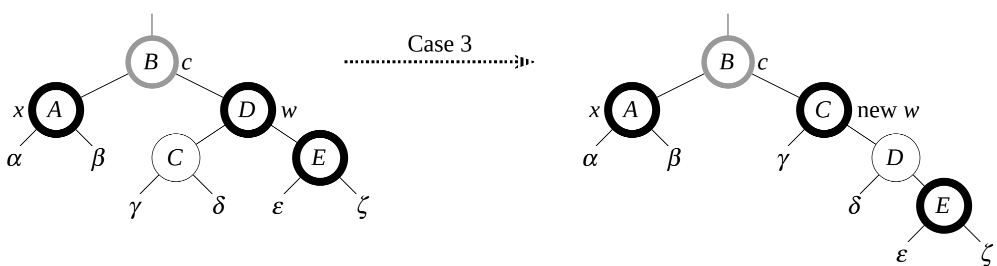
Case 2: w is black and both of w's children are black



[Node with gray outline is of unknown color, denoted by c.]

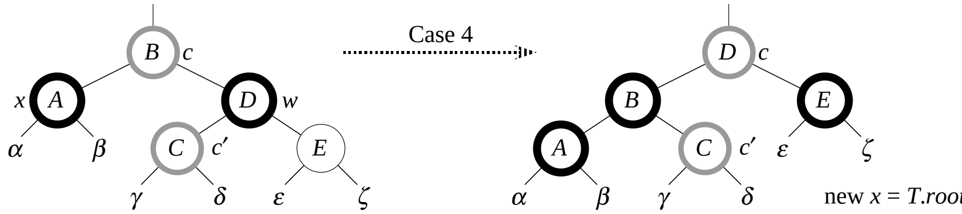
- Take 1 black off x ($\Rightarrow$ singly black) and off w ($\Rightarrow$ red).
- Move that black to $x.p$.
- Do the next iteration with $x.p$ as the new x.
- If entered this case from case 1, then $x.p$ was red $\Rightarrow$ new x is red & black $\Rightarrow$ color attribute of new x is RED $\Rightarrow$ loop terminates. Then new x is made black in the last line.

Case 3: w is black, w's left child is red, and w's right child is black



- Make w red and w's left child black.
- Then right rotate on w.
- New sibling w of x is black with a red right child $\Rightarrow$ case 4.

Case 4: w is black, w's left child is black, and w's right child is red



[Now there are two nodes of unknown colors, denoted by c and c'.]

- Make w be $x.p$'s color (c).
- Make $x.p$ black and w's right child black.
- Then left rotate on $x.p$.
- Remove the extra black on x ($\Rightarrow x$ is now singly black) without violating any red-black properties.
- All done. Setting x to root causes the loop to terminate.

Analysis

$O(\lg n)$ time to get through RB-DELETE up to the call of RB-DELETE-FIXUP.

Within RB-DELETE-FIXUP:

- Case 2 is the only case in which more iterations occur.
  - x moves up 1 level.
  - Hence, $O(\lg n)$ iterations.
- Each of cases 1, 3, and 4 has 1 rotation $\Rightarrow \leq 3$ rotations in all.
- Hence, $O(\lg n)$ time.

[In Chapter 14, we'll see a theorem that relies on red-black tree operations causing at most a constant number of rotations. This is where red-black trees enjoy an advantage over AVL trees: in the worst case, an operation on an n-node AVL tree causes $\Omega(\lg n)$ rotations.]