Red-Black Trees

Part II: Details of the insertion algorithm
Inserting a new element

• For a red-black tree of \( n \) nodes, inserting a new node can be done in \( O(\log_2 n) \) time.

• First, we insert a new node \( x \) exactly as if it were an ordinary binary search tree.

• The new node \( x \) is then colored red.

• To guarantee the red-black tree rules are preserved, the new tree undergoes some node recoloring and tree rotations.

• Most of the actual (pseudo)code handles various cases that may arise in this process.
Inserting a new element

• The only rule that can get violated is that the new node is a red child of a red node.

• Our goal is to move this violation up along the tree while maintaining the second rule
  • Every path from a given node to one of its non-branching descendants contains the same number of black nodes.

\[
\text{RBInsert}(T, x) \\
\text{Insert } x \text{ into } T \text{ like a binary search tree; } \\
x.\text{color} = \text{red; } \\
\text{while } (T.\text{root} \neq x \&\& x.\text{parent}.\text{color} == \text{red}) \\
\quad \text{move the violation up along the tree}
\]

If this rule is broken, fixing it is quite complex! So we always take care to maintain it.

There are two possibilities in each iteration of the while loop:
  1. The node \( x \) moves up the tree
  2. Some rotations are done and the loop terminates.
Inserting a new element

• There are 6 possible cases to consider
  • 3 cases for when \( x \)’s parent is a left child of \( x \)’s grandparent
  • 3 cases for when \( x \)’s parent is a right child of \( x \)’s grandparent

• These are symmetric, so we will study only the code for when \( x \).parent is a left child.

```java
while (T.root != x && x.parent.color == red) {
    if (x.parent is a left child) {
        Case 1: x.parent and y are both red
        else { // i.e. y is black
            Case 2: x is a right child
            Case 3: x is a left child
        }
    }
}
```

We will need to do different things depending on the color of \( x \)’s parent’s sibling (a.k.a. the “uncle” node). So let’s just call it \( y \).
Case 1: Parent and “uncle” are colored red

• The tree was a valid red-black tree before insertion.
• That means \texttt{x.parent.parent.color = black;}
• So
  • We set both \texttt{x.parent.color} and \texttt{y.color} as black. This fixes the red-red problem between \texttt{x} and its parent.
  • And set \texttt{x.parent.parent.color = red;}
• Now, the only problem that may arise is that \texttt{x.parent.parent} may have a red parent!
  • So we set \texttt{x.parent.parent} as the new \texttt{x}!
  • And go to the next iteration of the while loop (... this is how we move the violation up along the tree)
Case 2 & 3: Parent is colored red, but “uncle” isn’t!

**Case 2: \( x \) is a right child**

- What happens if we do a left rotation on the node \( x \)?
- Well ...
  - The node \( x \) takes its parent’s place, and
  - The parent becomes the left child of \( x \).
  - Now, we have a tree where the two consecutive red-red nodes are parent and left-child.
- But this is just case 3!
  - So ... we just convert case 2 to case 3 😊

**Case 3: \( x \) is a left child**
Case 2 & 3: Parent is colored red, but “uncle” isn’t!

**Case 2: $x$ is a right child**
- What happens if we do a left rotation on the node $x$?
  - Well …
    - The node $x$ takes its parent’s place, and
    - The parent becomes the left child of $x$.
    - Now, we have a tree where the two consecutive red-red nodes are parent and left-child.
  - But this is just case 3!
    - So … we just convert case 2 to case 3 😊

**Case 3: $x$ is a left child**
- We make some color changes and perform a right-rotation
  - Set $x$.parent.color = black;
  - Set $x$.parent.parent.color = red;
  - rightRotate(T, $x$.parent.parent);
  - Now there are no longer any two red-red parent-child nodes, so we stop.

**WHY?**
Case 1 example

The red-red problem is pushed “up”. If C’s parent is red, then we set C as the new $x$ and go to the next iteration of the while loop.
Case 2 and 3 example

Left-rotation on B and then a right-rotation it’s grandparent, C.

Node that just got inserted or became red via a recoloring lower in the tree