Functional Programming
Practicals 2: Red-Black Tree

Shin-Cheng Mu

FLOLAC 2026

Download `RedBlack.zip` from the course website and unzip it. Try loading the file `RedBlack.hs` (which is our main file) into `ghci`.

The datatype definition RBTree and functions for inserting an element into a red-black tree (*insert* and *ins*) and rotation (*rotate*) are defined in `Base.hs`. Read the code and try to understand them. Note that *insert* is defined using *ins*.

Operations and Properties

1. The function below, defined in `Base.hs`, builds a tree from a list.

   $$\begin{aligned} &\mathit{build} :: \mathsf{List\ Int} \to \mathsf{RBTree\ Int} \\ &\mathit{build} = \mathit{foldr}\ \mathit{insert}\ \mathsf{E}\ . \end{aligned}$$

   Define $\mathit{search} :: \mathsf{Int} \to \mathsf{RBTree\ Int} \to \mathsf{Bool}$ such that $\mathit{search}\ x\ t$ determines whether x occurs in t.

   You may play around with *build* and *search* a bit before moving on.

2. The function *bheight* defines the *black height* of an RBTree. An RBTree is *balanced* iff for *every* $\mathsf{R}\ t\ x\ u$ or $\mathsf{B}\ t\ x\ u$ in the tree, t and u have the same black height. Finish the definition of function $\mathit{isBalanced} :: \mathsf{RBTree}\ a \to \mathsf{Bool}$, which determines whether a tree is balanced.

3. The functions $\mathit{maxRBT}, \mathit{minRBT} :: \mathsf{RBTree\ Int} \to \mathsf{Int}$ respectively returns the maximum and minimum value stored in the given tree. Finish their definitions. You may use the functions $(\downarrow), \mathit{max} :: \mathsf{Int} \to \mathsf{Int} \to \mathsf{Int}$.

4. An RBTree is sorted if for every $\mathsf{R}\ t\ x\ u$ (or $\mathsf{B}\ t\ x\ u$) in the tree, x is greater than every value in t and smaller than every value in u. Finish the definition of function $\mathit{isSorted} :: \mathsf{RBTree\ Int} \to \mathsf{Bool}$, which determines whether a tree is sorted.

5. An RBTree is *semi-red-black* if every red node has two black subtrees (there is no such restrictions for black nodes). A red-black tree is a semi-red-black tree whose root is black. Finish

the definition of function $isSemiRB :: \text{RBTree Int} \to \text{Bool}$, which determines whether a tree is semi-red-black.

Meanwhile, *isRB* is given by:

$$isRB :: \text{RBTree } a \to \text{Bool}$$
$$isRB\ t = color\ t == \text{Blk} \wedge isSemiRB\ t \quad .$$

Note: a tree being semi-red-black implies that there are no consecutive red nodes on every path of the tree. Therefore, in a *balanced* semi-red-black tree, the longest path is at most around twice as long as the shortest path. This is how we achieve $O(\log n)$-time searching in a red-black tree.

With the properties given above, being a red-black tree is defined by:

$$isRedBlackTree :: \text{RBTree Int} \to \text{Bool}$$
$$isRedBlackTree\ t = isSorted\ t \wedge isBalanced\ t \wedge isBlkRB\ t \quad .$$

Proving the Properties

To show that *insert* is correct, we wish to have that for all k and t,

$$isRedBlackTree\ t \Rightarrow isRedBlackTree\ (insert\ k\ t) \quad ,$$

which is equivalent (why?) to showing that for all k and t,

$$\begin{aligned} &isSorted\ t \Rightarrow isSorted\ (insert\ k\ t) \wedge \\ &isBalanced\ t \Rightarrow isBalanced\ (insert\ k\ t) \wedge \\ &isRB\ t \Rightarrow isRB\ (insert\ k\ t) \quad . \end{aligned}$$

But are these all true?

Properties regarding sortedness are similar to the last question in Practicals 0 and thus omitted. Consider balancing and the red-black structure.

1. To talk about balancing we need to talk about heights. Play around with *bheight*, *build*, and *insert* a bit. You might notice that *insert* sometimes increases the black height of a tree, sometimes not.

   It turns out, interestingly, that *ins* does *not* increase the black height of a tree! The height of a tree is always incremented by the *blacken* step in *insert*.

   Prove that $bheight\ (ins\ k\ t) = bheight\ t$, for all k and t. Find out what auxiliary properties you need and prove them too if they are not trivial. If it involves *rotate*, you may prove only one representative case.

2. Prove that $isBalanced\ t \Rightarrow isBalanced\ (insert\ k\ t)$.

3. To prove $isRB\ t \Rightarrow isRB\ (insert\ k\ t)$, we wish to have $isSemiRB\ t \Rightarrow isSemiRB\ (ins\ k\ t)$. But that is certainly not true — *ins* does sometimes generate trees with two consecutive red nodes! Still, *ins* definitely preserves some useful properties. What is it?

   We call a tree *infrared* (meaning that it is extra-red) if both subtrees are semi-red-black. The root must be red (thus E is not infrared), furthermore, we only demand *at least one* subtree to be black:

$$
\begin{aligned}
&isIRB :: \mathsf{RBTree}\ a \to \mathsf{Bool} \\
&isIRB\ (\mathsf{R}\ t\ x\ u) = (color\ t == \mathsf{Blk} \vee color\ u == \mathsf{Blk}) \wedge \\
&\quad isSemiRB\ t \wedge isSemiRB\ u \\
&isIRB\ \_ = \mathsf{False}\ .
\end{aligned}
$$

   Indeed, *ins* sometimes return trees that are IRB. But when? It turns out that for all k and t,

$$
\begin{aligned}
&isSemiRB\ t \wedge color\ t == \mathsf{Red} \Rightarrow isIRB\ (ins\ k\ t)\ , \\
&isSemiRB\ t \wedge color\ t == \mathsf{Blk} \Rightarrow isSemiRB\ (ins\ k\ t)\ .
\end{aligned}
$$

   Prove the properties above. Find out what properties you need about *rotate*.