Lesson 16
RED-BLACK TREES

July 13, 2023

Last time, we iterating several times on implementing **generic dictionaries**, which are dictionaries which can have keys (and data) of arbitrary type.

We tried implementing them via passing an explicit comparison function, but found that this could lead to unsafety if comparison functions were mixed and matched.

We then used **functors**, which produce structures from other structures, to create a `MkDict` functor, which creates a library for dictionaries of a particular key type, from a given type and comparison function.

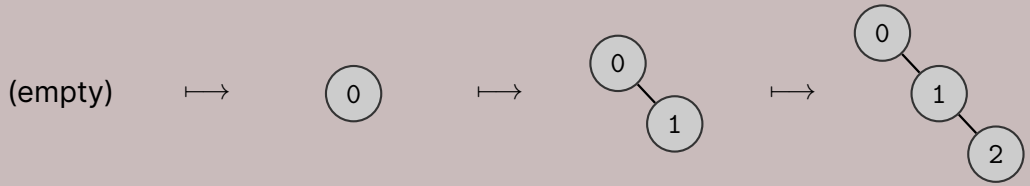
1 - Red-Black Trees

Recall the `POLY_DICT` signature we used last lecture:

```
signature POLY_DICT =
  sig
    structure Key : ORD

    (* mapping keys of type Key.t to values of 'a *)
    type 'a t

    val empty : 'a t
    val insert : Key.t * 'a -> 'a t -> 'a t
    val lookup : Key.t -> 'a t -> 'a option
  end
```

Recall that our definition of insertion was just via comparing keys to root keys, and going left or right depending on the result of that comparison, until eventually we found an `Empty` node.

This is not actually a good way to store data in our dictionaries, because it might not be $O(\log n)$ search time! In particular, consider the sequence of insertions of inserting ascending numbers, for a tree with nodes of integer keys.



This sequence of insertions is inefficient, because we need to keep traversing to the end of the spine to put the next element on. In essence, it's an $O(n)$ cost per insertion and lookup, in the worst case, so we aren't doing much better than a list!

We know the reason for this already, from our work and span calculations. This amounts to the fact that our binary search trees may not always be roughly balanced.

In this lecture, we will discuss and implement a new kind of data structure for self-balancing binary search trees, called **red-black trees**.

The definition of a red-black tree is as such:

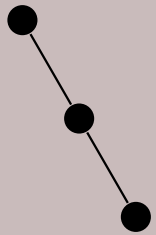
```
datatype 'a rbtree =
    Empty
  | Red of 'a rbtree * (Key.t * 'a) * 'a rbtree
  | Black of 'a rbtree * (Key.t * 'a) * 'a rbtree
```

Def A **red-black tree** is a kind of BST which has some nodes which are colored red, and some nodes which are colored black. It also has three important invariants:

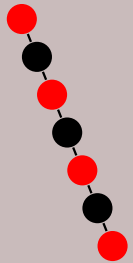
- It is a BST, meaning that its inorder traversal is sorted with respect to its keys.
- The children nodes of a `Red` node must be `Black`.
- Every path from the root to an `Empty` leaf node has the **same number** of `Black` nodes. This is also known as the tree's **black height**.

From the invariants, we can derive the fact that the tree should support efficient $O(\log n)$ insertion and lookup, in terms of the number of nodes in the tree n.

This is because the black height of any path is the same. In the worst case, the heights on different paths could be very different because of the number of red nodes, but because of the second property, the number of red nodes is, at maximum, the black height plus 1.



The best and worst case for black height relative to total height of the tree, respectively.



Key This means on a given path to the bottom, there can be at worst approximately as many red nodes as there are black nodes.

Equivalently, any path from the root to the bottom can be at worst twice the length of another.

This isn't a perfectly balanced tree, but it turns out this is close enough to guarantee asymptotic $O(\log n)$ traversal.

That's the theory. How do we ensure that we can maintain these invariants?

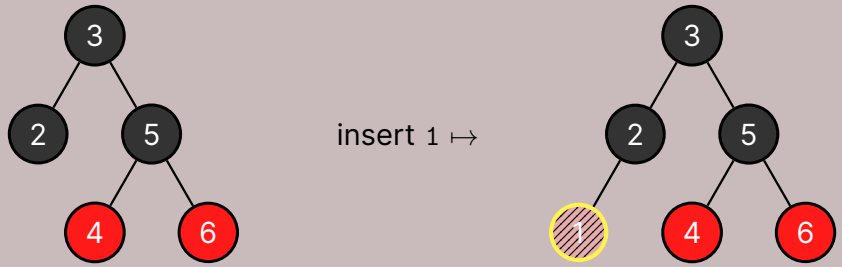
2 - Maintaining Invariants

The main operations we are concerned with are `insert` and `remove`. Since a red-black tree is a BST, we can look up in the same way, by just traversing down a path according to our comparison function.

For insertion, we cannot avoid traversing the same path, as we cannot ever go to the left of a node we are `LESS` than, nor can we go right of a node we are `GREATER` than. However, *after* we insert, we have some options for rebalancing our tree.

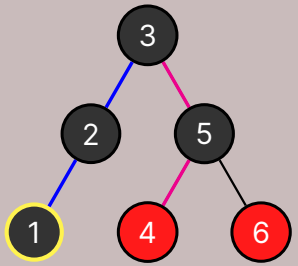
Def A **self-balancing tree** is a kind of tree data structure that can perform some balancing operations upon insertion or removal, to ensure the tree remains approximately balanced.

Let's consider the insertion case. If we're inserting into an arbitrary tree, what should we color the newly created node?



Clearly, we have two cases. We can either color the node red, or we can color the node black.

Consider the black height invariant. If every path in the tree has the same black height, then clearly we cannot color the node black, because the newly created path will have a greater black height!



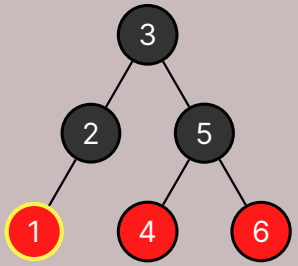
Black height invariant: ✗

Red children invariant: ✓

Black height (blue path): 3

Black height (magenta path): 2

What if we color the node red?



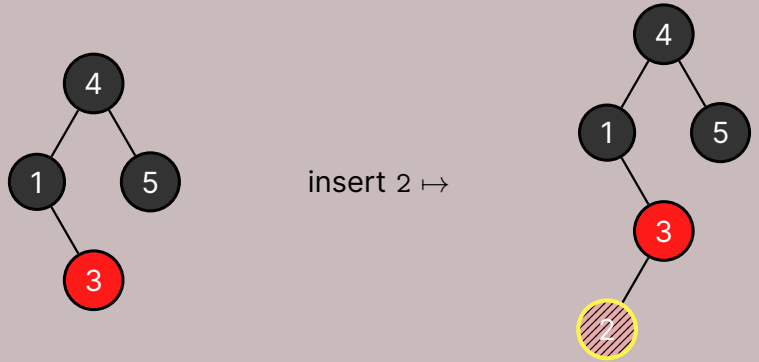
Black height invariant: ✓

Red children invariant: ✓

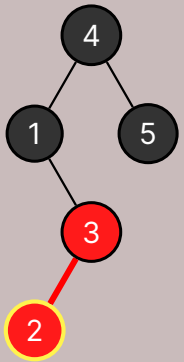
We see that coloring a node red will **always** end up satisfying the black height invariant, because the black height of every path will remain the same.

But, what about our other invariants? Let's see an example on a slightly different tree.

Let's insert again on a different tree. This time, suppose we're inserting with a key of 2.



As before, let's color the node red.



Black height invariant: ✓

Red children invariant: ✗

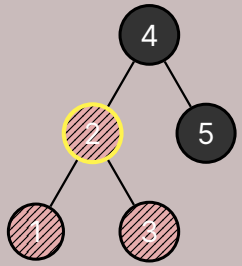
Uh oh, our heuristic of "always color new nodes red" didn't get us very far.

It only works in the case where our parent is black. Otherwise, we end up in a **red-red violation**.

We also probably don't want to produce a tree of this shape anyways, because it's not as balanced as it could be! How can we fix this?

If we had free rein to transform this tree, how would we ideally balance it?

We would like to produce this tree:

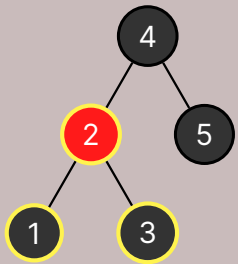


But what should we color the nodes of ①, ②, and ③ to ensure our invariants are respected?

We are going to choose to color the higher node red, which will force its children to be **black**, and see why this was the correct choice.

This will end up producing a valid red-black tree.

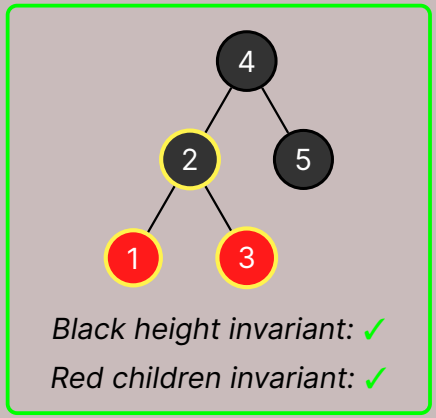
But why did we choose to do it this way? We could have also decided to color it the other way, with the root node being black, and the children being red.



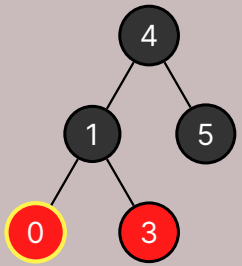
Black height invariant: ✓

Red children invariant: ✓

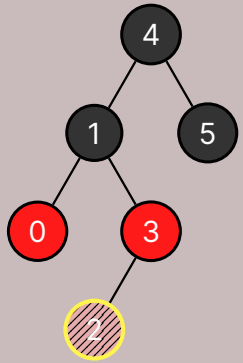
This is a valid red-black tree, but we will see that there are cases where we can't do this!



Let's insert again on a tree which looks very similar, but has an extra 0 node. As before, we insert the key 2:



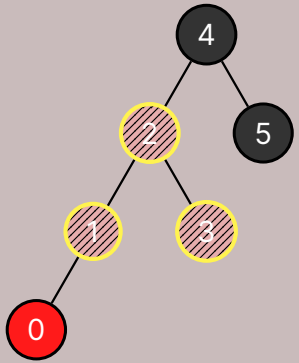
insert $2 \mapsto$

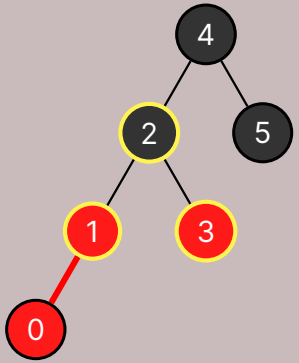


Well, if we want to rebalance this tree in the same way, we have to move the `2` up to `1`'s current position. But what should we do with the `0` node?

Because `0` is less than `1`, we have no choice but to keep it left of `1`! So we get:

Now, we have two choices over what we can color this triplet.



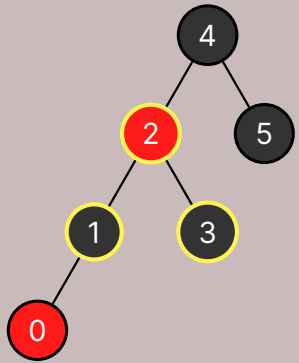


Black height invariant: ✓

Red children invariant: ✗

We see that only the triplet coloring with red at the root is valid!

This will form the basis for our canonical rebalance coloring scheme.



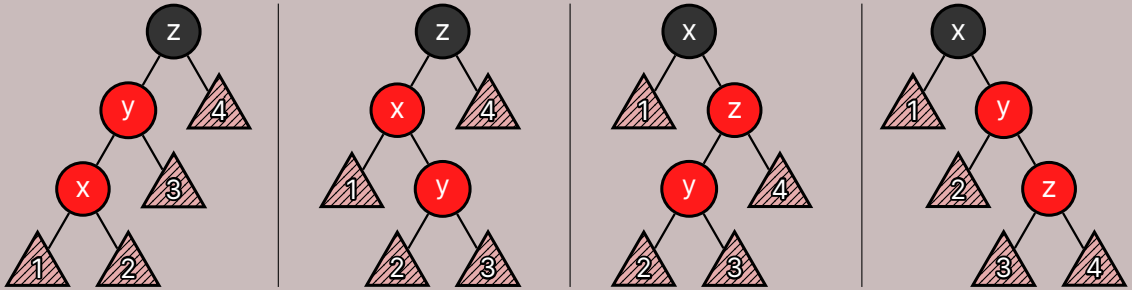
Black height invariant: ✓

Red children invariant: ✓

Now that we've seen a few examples, we're ready to more in-depth describe our scheme for inserting nodes into a red-black tree.

Our idea will be to insert a node, and always color it red, to maintain black height. If this induces a red-red violation, then we will simply rotate the tree, and recolor to prevent a red-red violation.

There are four cases we are interested in, for red-red violations.

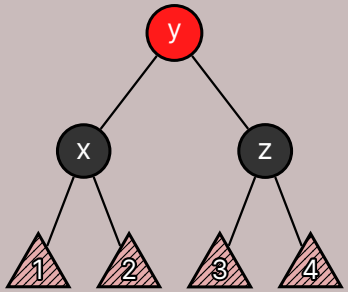


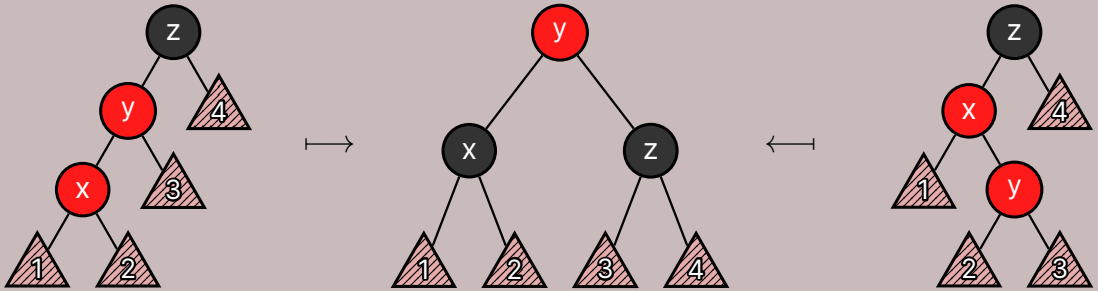
Given three nodes with arbitrary children, it turns out there is a prescribed way that we will rotate each of these cases. In fact, the resulting subtree is the same!

Given nodes `x`, `y`, and `z`, which are ordered in the same way as the letters we have used to denote them, there is only one tree which we can produce:

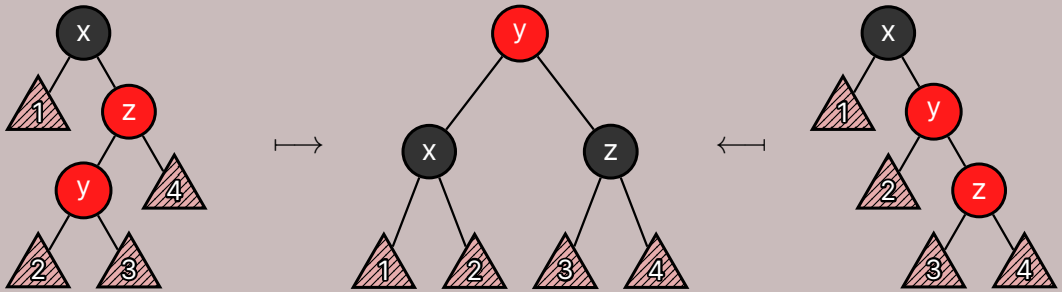
A key observation of this rebalancing scheme is that, because the bottom nodes are colored black, it is **impossible** to produce a red-red violation from the connections to the children trees `1`, `2`, `3`, and `4`, irrespective of color!

Given that we have also preserved black height, because each path to the children trees still encounters one black node, are we done?



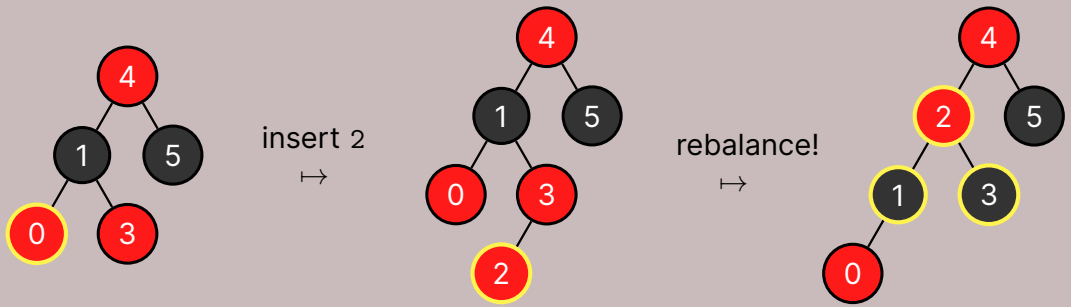


These two examples illustrate what is known as a **right rotation**. We will implement a function `restoreLeft` to do this, because it's restoring a path that goes left.

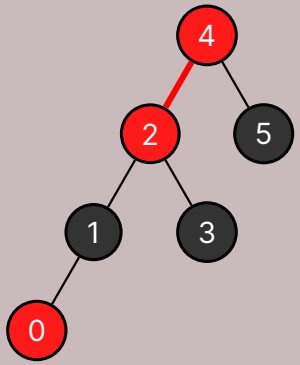


These two examples illustrate what is known as a **left rotation**. Similarly, we will implement a function called `restoreRight` that will fix this situation.

Let's see! Let's do another example, which is identical to the last one, except that the root is colored red. As before, we insert the key 2:



Uh oh...



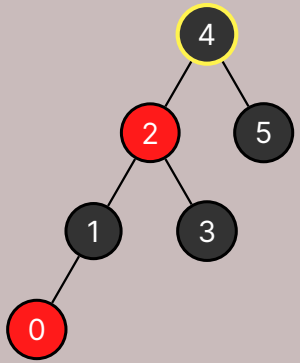
Black height invariant: ✓

Red children invariant: ✗

We see that we end up with a red-red violation.

However, a neat fact about red-black trees is that the root can always be colored black, no matter what!

Check your understanding Why does this preserve the invariants?



Black height invariant: ✓

Red children invariant: ✓

So by coloring the root node black for free, we end up getting a valid red-black tree.

A question remains – did we get lucky? What happens if we aren't at the root?

Our issue was that rebalancing our subtree to get rid of a red-red violation might introduce another red-red violation, slightly above it. Our scheme will be to **continuously rebalance**, and push the red-red violation further upwards, until we potentially reach the root, or no longer have a violation.

3 - Implementing Red-Black Trees

We can formalize the notion of our algorithm by defining a new kind of tree, which we will call an **almost red-black tree**.[1]

Def An **almost red-black tree** shares the same properties as a red-black tree, but its root node and one of the root node's children may both be red.

In other words, an almost red-black tree is allowed to break the red children invariant, once, and only at the root.

[1] Sort of in the same way that I am an *almost professor*, which is to say, I am not one.

Here is a specification for our rotation functions, which we will call `restoreLeft` and `restoreRight`.

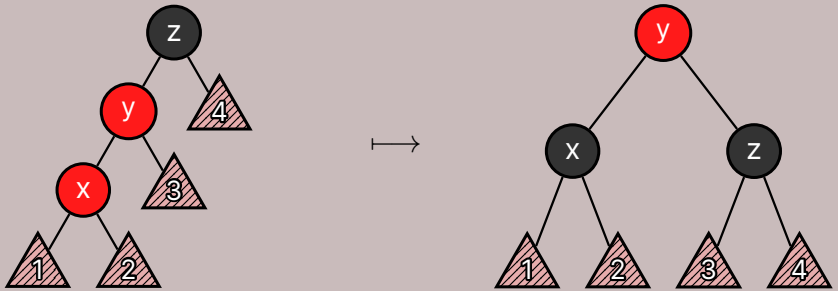
`restoreLeft : 'a rbtree -> 'a rbtree`
REQUIRES: `T` is an RBT, or `T` is a `Black` tree, with a **left** child that is an ARBT, and a right child that is an RBT
ENSURES: `restoreLeft T` is an RBT with the same entries as `T`

`restoreRight : 'a rbtree -> 'a rbtree`
REQUIRES: `T` is an RBT, or `T` is a `Black` tree, with a **right** child that is an ARBT, and a left child that is an RBT
ENSURES: `restoreRight T` is an RBT with the same entries as `T`

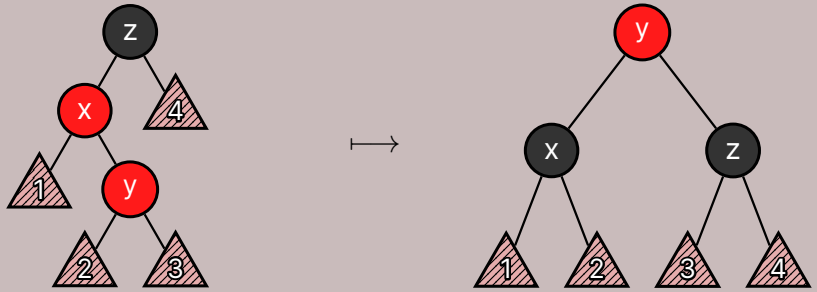
Recall that functions which do not appear in the signature of a module are not visible to users of the library. This means that, after we write `restoreLeft` and `restoreRight`, they are only for internal use, in helping us implement `insert`!

Note also that although `restoreLeft` and `restoreRight` produce true RBTs, it may produce an ARBT in the node right above it, if it is colored red. We will percolate these fixes up as we insert, however.

```
fun restoreLeft (Black(Red(Red(t1,x,t2),y,t3),z,t4)) =
  Red(Black(t1,x,t2), y, Black(t3,z,t4))
```



```
fun restoreLeft (Black(Red(Red(t1,x,t2),y,t3),z,t4)) =
      Red(Black(t1,x,t2), y, Black(t3,z,t4))
  | restoreLeft (Black(Red(t1,x,Red(t2,y,t3)),z,t4)) =
      Red(Black(t1,x,t2), y, Black(t3,z,t4))
```

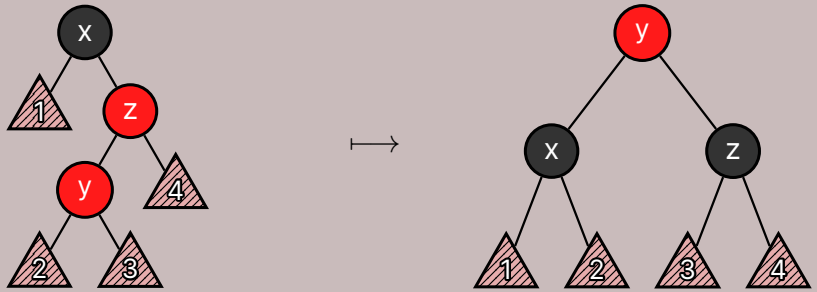


```
fun restoreLeft (Black(Red(Red(t1,x,t2),y,t3),z,t4)) =
      Red(Black(t1,x,t2), y, Black(t3,z,t4))
  | restoreLeft (Black(Red(t1,x,Red(t2,y,t3)),z,t4)) =
      Red(Black(t1,x,t2), y, Black(t3,z,t4))
  | restoreLeft other = other
```

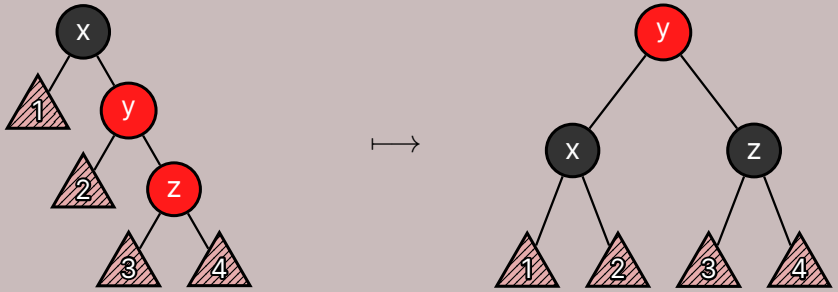
And finally, any other input doesn't need to be restored from the left, so we omit it here.

Now we can proceed to the right case.

```
fun restoreRight (Black(t1,x,Red(Red(t2,y,t3),z,t4))) =
      Red(Black(t1,x,t2), y, Black(t3,z,t4))
```



```
fun restoreRight (Black(t1,x,Red(Red(t2,y,t3),z,t4))) =
      Red(Black(t1,x,t2), y, Black(t3,z,t4))
  | restoreRight (Black(t1,x,Red(t2,y,Red(t3,z,t4)))) =
      Red(Black(t1,x,t2), y, Black(t3,z,t4))
```



```
fun restoreRight (Black(t1,x,Red(Red(t2,y,t3),z,t4))) =
      Red(Black(t1,x,t2), y, Black(t3,z,t4))
  | restoreRight (Black(t1,x,Red(t2,y,Red(t3,z,t4)))) =
      Red(Black(t1,x,t2), y, Black(t3,z,t4))
  | restoreRight other = other
```

And finally, as before, `restoreRight` also acts as the identity function on any input which is not one of the two rotation cases.

Now that we've written our balancing functions, let's handle insertion.

We will achieve this via two functions. One will carry out the insertion algorithm we described previously, and one will be a wrapper function that simply takes care of the rectifying any red-red violations at the root.

`insert : (Key.t * 'a) -> 'a rbtree -> 'a rbtree`
REQUIRES: `T` is an RBT
ENSURES: `insert (k, v) T` is an RBT with all the entries of `T`, plus the key-value pair `(k, v)`. If an entry already exists, it is replaced.

We will define the `ins` function locally to the definition of `insert`, meaning that it does not need to take the key-value pair as an argument, since it is already in the environment.

`ins : 'a rbtree -> 'a rbtree`
REQUIRES: `T` is an RBT
ENSURES: `ins (k, v) T` has the same specification as `insert`, but `ins (k, v) (Black v)` is an RBT, and `ins (k, v) (Red v)` is an ARBT.

```
fun insert (k, v) T =
  let
    fun ins T = (* ... *)
  in
    case ins (k, v) T of
      Red v => Black v
    | other => other
  end
```

If there is ever a red root, we can safely recolor it to black. We only want to do this at the true root of the tree, however, hence why we have the outer `insert`, which is not recursive.

Now, we must define the `ins` function, which will do the real work of the insertion algorithm.

Note that this function is being written in the body of `insert`, so we obtain the values of `k` and `v` as the argument, the key-value pair we are trying to insert.

```
fun ins Empty = Red (Empty, e, Empty)
  | ins (Black (l, (k', v'), r)) =
      (case Key.compare (k, k') of
        EQUAL   => Black (l, (k, v), r))
      | LESS    => restoreLeft (Black (ins l, (k', v'), r))
      | GREATER => restoreRight (Black (l, (k', v'), ins r))
  | ins (Red (l, (k', v'), r)) =
      (case Key.compare (k, k') of
        EQUAL   => Red (l, (k, v), r)
      | LESS    => Red (ins l, (k', v'), r)
      | GREATER => Red (l, (k', v'), ins r))
```

```
fun ins Empty = Red (Empty, e, Empty)
  | ins (Black (l, (k', v'), r)) =
      (case Key.compare (k, k') of
        EQUAL   => Black (l, (k, v), r))
      | LESS    => restoreLeft (Black (ins l, (k', v'), r))
      | GREATER => restoreRight (Black (l, (k', v'), ins r))
  | ins (Red (l, (k', v'), r)) =
      (case Key.compare (k, k') of
        EQUAL   => Red (l, (k, v), r)
      | LESS    => Red (ins l, (k', v'), r)
      | GREATER => Red (l, (k', v'), ins r))
```

Note that we do the `restoreLeft` and `restoreRight` calls only when inserting into the left and right trees, respectively.

We also don't need to restore when inserting into a `Red` tree – why is that?

Lookup for a red-black tree looks similar to the regular BST case:

```
fun lookup k Empty = NONE
  | lookup k ( Red (l, (k', v'), r)
             | Black (l, (k', v'), r)) =
      case Key.compare (k, k') of
        EQUAL   => SOME v'
      | LESS    => lookup k l
      | GREATER => lookup k r
```

Here, we make use of something called an **or-pattern**, to reduce some boilerplate. This is a pattern `(<pat1> | <pat2>)` that is matched if either `<pat1>` or `<pat2>` is matched, but is only valid so long as both patterns introduce the exact same bindings, with the same types.

Now we can assemble the final parts of the puzzle, by putting everything we just wrote in a functor[2], so that we can achieve full generality!

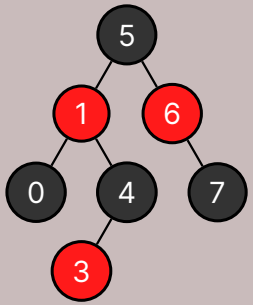
```
functor MkRedBlackDict (Key : ORD) :> POLY_DICT =
  struct
    structure Key = Key

    datatype 'a rbtree =
        Empty
      | Red of 'a rbtree * (Key.t * 'a) * 'a rbtree
      | Black of 'a rbtree * (Key.t * 'a) * 'a rbtree

    (* ... *)
  end
```

[2]Remember those? This didn't stop being a lecture on modules, or anything.

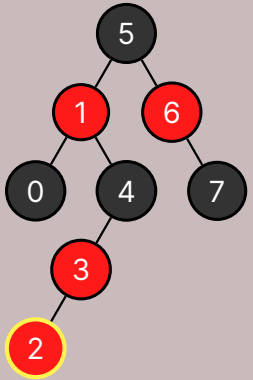
```
functor MkRedBlackDict (Key : ORD) :> POLY_DICT =
  struct
    structure Key = Key

    datatype 'a rbtree =
        Empty
      | Red of 'a rbtree * (Key.t * 'a) * 'a rbtree
      | Black of 'a rbtree * (Key.t * 'a) * 'a rbtree
    type 'a t = 'a rbtree

    val empty = Empty

    fun restoreLeft (Black(Red(Red(t1,x,t2),y,t3),z,t4)) =
          Red(Black(t1,x,t2), y, Black(t3,z,t4))
      | restoreLeft (Black(Red(t1,x,Red(t2,y,t3)),z,t4)) =
          Red(Black(t1,x,t2), y, Black(t3,z,t4))
      | restoreLeft other = other

    fun restoreRight (Black(t1,x,Red(Red(t2,y,t3),z,t4))) =
          Red(Black(t1,x,t2), y, Black(t3,z,t4))
      | restoreRight (Black(t1,x,Red(t2,y,Red(t3,z,t4)))) =
          Red(Black(t1,x,t2), y, Black(t3,z,t4))
      | restoreRight other = other

    (* ... *)
```

```
(* ... *)

    fun insert (k, v) T =
      let
        fun ins Empty = Red (Empty, e, Empty)
          | ins (Black (l, (k', v'), r)) =
              (case Key.compare (k, k') of
                EQUAL   => Black (l, (k, v), r))
              | LESS    => restoreLeft (Black (ins l, (k', v'), r))
              | GREATER => restoreRight (Black (l, (k', v'), ins r))
          | ins (Red (l, (k', v'), r)) =
              (case Key.compare (k, k') of
                EQUAL   => Red (l, (k, v), r)
              | LESS    => Red (ins l, (k', v'), r)
              | GREATER => Red (l, (k', v'), ins r))
      in
        case ins (k, v) T of
          Red v => Black v
        | other => other
      end

    fun lookup k Empty = NONE
      | lookup k ( Red (l, (k', v'), r)
                 | Black (l, (k', v'), r)) =
          case Key.compare (k, k') of
            EQUAL   => SOME v'
          | LESS    => lookup k l
          | GREATER => lookup k r
  end
```

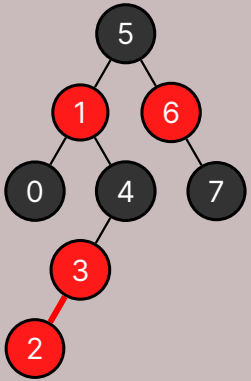
4 - A Final Trace

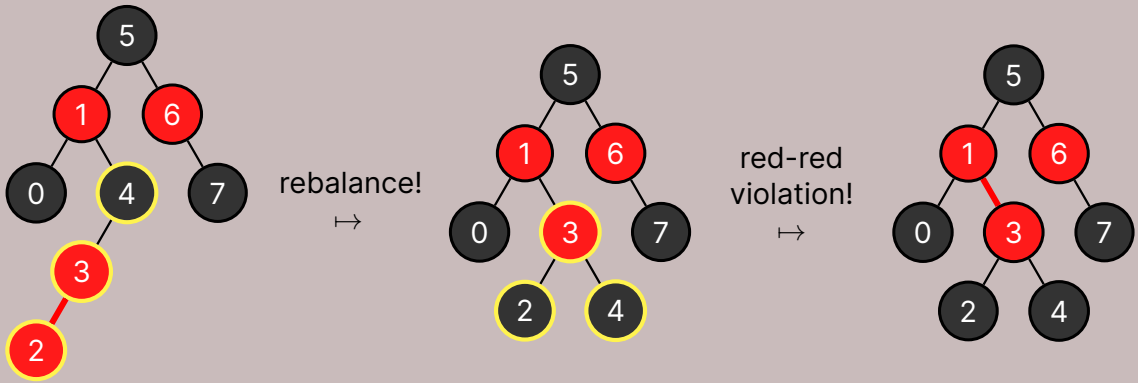


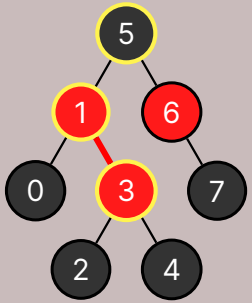
insert 2
↦



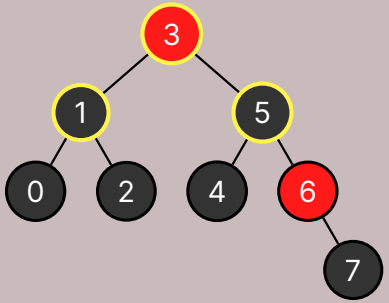
red-red
violation!
↦

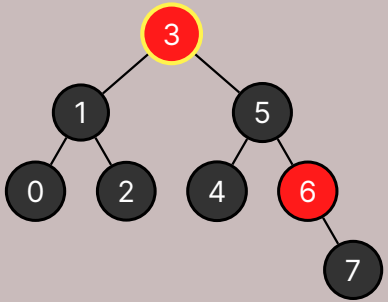



5
1
6
0
4
7
3
2
rebalance!
↦
5
1
6
0
3
7
2
4
red-red
violation!
↦
5
1
6
0
3
7
2
4

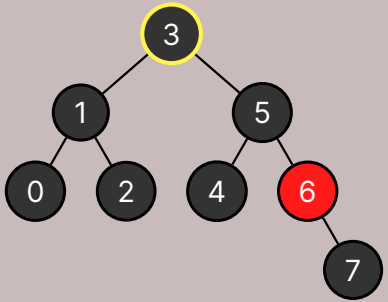


rebalance! ↦

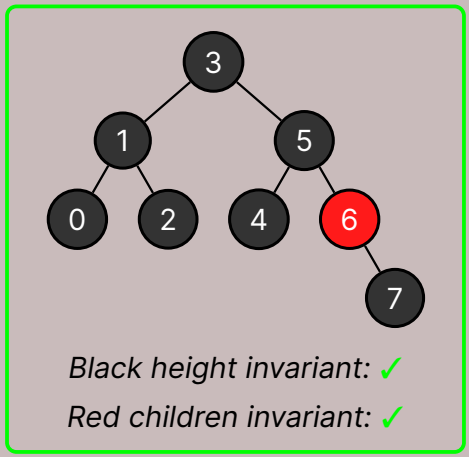




red root! ↦



Now, we obtain a full RBT!



Thank you!