Red-black Trees of smlnj
Studienarbeit

Angelika Kimmig

January 26, 2004

Contents

1 Introduction

Red-black trees guarantee logarithmic height of binary trees using only one additional bit per node to store balancing informations and are thus often used to implement finite sets of items efficiently. A prominent example is the set functor RedBlackSetFn provided by the library of Standard ML of New Jersey [1]. An analysis of parts of this implementation with the theorem prover Isabelle [2] revealed a major error in the library code. The trees produced by the given insertion and deletion functions often violate red-black invariants. The presented Isabelle/HOL theory includes insertion and deletion of elements, although deletion is not considered in the proofs. A corrected version of the insertion function is provided. The complete theory can be found in the appendix, as well as the corrected functor code.

The code of this set functor is based on Chris Okasaki's implementation of red-black trees [3], the delete function on the description by Cormen, Leiserson and Rivest [4].

According to the latter ones, a binary search tree is a red-black tree if it satisfies the following red-black properties:

(P1) Every node is either red or black.

(P2) The root is black.

(P3) Every leaf is black.

(P4) If a node is red, then both its children are black.

(P5) For each node, all paths from the node to descendant leaves contain the same number of black nodes.

The informal specification of red-black trees in redblack-set-fn.sml formulates this as

```
A red-black tree should satisfy the following two
invariants:
Red Invariant:  each red node has a black parent.
Black Condition:  each path from the root to an empty
node has the same number of black nodes (the tree's
black height).
The Red condition implies that the root is always
black.
```

The Isabelle/HOL theory RBT consists of two parts, the first one modelling datatypes and functions extracted from the set functor, the second one defining predicates for proofs of invariants. Among the difficulties of converting the original code to an Isabelle theory are syntactic transformations to meet restrictions imposed by Isabelle as well as providing well-founded relations such that termination can be proven automatically. For deletion, errors in case of elements not being present have to be modelled. Predicates are defined to test for presence of elements in trees and for orderedness of trees. Two red invariants are formulated. The weak one corresponds to invariant (P4) and thus accepts a red root if no red node has a red child. The strong one combines this with invariant (P2) and thus agrees with the informal specification. However, it has to be said that trees built by the insertion function do not meet the strong red invariant. Therefore, an alternative function which corrects this is given. Definitions of black height and black invariant complete the theory.

Based on this theory, four main theorems are proven for the original insertion function.

1. Inserting adds the element to be inserted to the tree and keeps all elements of the original tree.
2. Inserting an element into an ordered tree results in an ordered tree.
3. Inserting preserves the weak red invariant, i. e. property (P4).
4. Inserting preserves the black invariant (property (P5)) provided that the weak red one (property (P4)) holds before insertion.

This theorems are used to prove the same facts for the alternative insertion function. In addition, it is shown that this one preserves the strong red invariant, i. e. each red node has a black parent (properties (P4) and (P2)).

This paper proceeds as follows: Section 2 presents the theory, starting from the original code fragments and explaining the underlying principles as well as difficulties encountered during the development. Section 3 gives an overview of the formal analysis. First, some general schemes are introduced. The core part of this section consists of the presentation of the four main proofs, together with some lemmata which are used there. Finally, section 3.3 sketches the corresponding proofs for the new insertion function and shows that this one satisfies all required invariants.

2 Development of the Isabelle/HOL Theory RBT

2.1 Translation of Datatypes and Functions into HOL

In order to prove properties of the implementation with Isabelle, the datastructure and the corresponding algorithms are isolated from the set functor and converted to an Isabelle/HOL theory called RBT.thy.

The original ML-code defines datatypes at the beginning of the functor as follows:

```
functor RedBlackSetFn (K : ORD_KEY) :> ORD_SET
 where Key = K =
  struct

    structure Key = K

    type item = K.ord_key

    datatype color = R | B

    datatype tree
      = E
      | T of (color * tree * item * tree)
```

In ML, `ORD_KEY` denotes abstract linearly ordered keys of type `ord_key` together with a comparison function `compare` of type `ord_key * ord_key -> order`. Thus, the datatype `ml_order` as well as axiomatic type classes `ord_key` and `LINORDER` are introduced in the Isabelle theory and `item`, `color` and `tree` are defined. `LINORDER` allows to inherit properties of linear orders as e. g. transitivity from Isabelle's `linorder` by linking results of the `compare` function to ordinary comparisons by < and =. This results in the following part of RBT.thy:

```
datatype ml_order = LESS | EQUAL | GREATER

axclass ord_key < type

consts
  compare :: "'a::ord_key => 'a => ml_order "

axclass LINORDER < linorder, ord_key
  LINORDER_less "((compare x y) = LESS) = (x < y)"
  LINORDER_equal "((compare x y) = EQUAL) = (x = y)"
  LINORDER_greater "((compare x y) = GREATER) = (y < x)"

types 'a item = 'a::ord_key

datatype color = R | B

datatype 'a tree = E | T color ('a tree) ('a item) ('a tree)
```

The insertion function `ins` for red-black trees is taken from the following ML function `add` for sets, where `m` is the tree storing the set and `x` is the element to be added.

```
fun add (SET(nItems, m), x) = let
  val nItems' = ref nItems
  fun ins E = (nItems' := nItems+1; T(R, E, x, E))
    | ins (s as T(color, a, y, b)) = (case K.compare(x, y)
         of LESS => (case a
            of T(R, c, z, d) => (case K.compare(x, z)
                  of LESS => (case ins c
                        of T(R, e, w, f) =>
```

```
                                   T(R, T(B,e,w,f), z, T(B,d,y,b))
                                 | c => T(B, T(R,c,z,d), y, b)
                               (* end case *))
                           | EQUAL => T(color, T(R, c, x, d), y, b)
                           | GREATER => (case ins d
                                 of T(R, e, w, f) =>
                                     T(R, T(B,c,z,e), w, T(B,f,y,b))
                                   | d => T(B, T(R,c,z,d), y, b)
                                 (* end case *))
                         (* end case *))
                     | _ => T(B, ins a, y, b)
                   (* end case *))
                 | EQUAL => T(color, a, x, b)
                 | GREATER => (case b
                   of T(R, c, z, d) => (case K.compare(x, z)
                          of LESS => (case ins c
                                 of T(R, e, w, f) =>
                                     T(R, T(B,a,y,e), w, T(B,f,z,d))
                                   | c => T(B, a, y, T(R,c,z,d))
                                 (* end case *))
                           | EQUAL => T(color, a, y, T(R, c, x, d))
                           | GREATER => (case ins d
                                 of T(R, e, w, f) =>
                                     T(R, T(B,a,y,c), z, T(B,e,w,f))
                                   | d => T(B, a, y, T(R,c,z,d))
                                 (* end case *))
                         (* end case *))
                     | _ => T(B, a, y, ins b)
                   (* end case *))
               (* end case *))
  val m = ins m
  in
    SET(!nItems', m)
  end
```

Inserting into the empty tree `E` simply creates a red node containing the new element, both of its children being leaves. If an element equal to the new one is already present in the tree, it is just replaced. New elements are inserted into the corresponding subtree. Subtrees which are not red are simply replaced by the result of the recursive call, their father becoming black. For red subtrees, recursive calls are performed on their corresponding subtree. In order to guarantee that no red node has a red child, we have to distinguish between results with red and black root. To achieve this, different actions are chosen depending on whether the recursive call uses one of the outermost subtrees or one of the inner ones. Fig. 1 represents the first group, fig. 2 the second one. Black results are just linked to the corresponding parent node as shown on the left of each figure. In the case of red results, this would lead to a red-red conflict, which can't be corrected by simply recoloring one of the nodes. This would change black heights and thus violate the black invariant. Therefore, recolorations are combined with

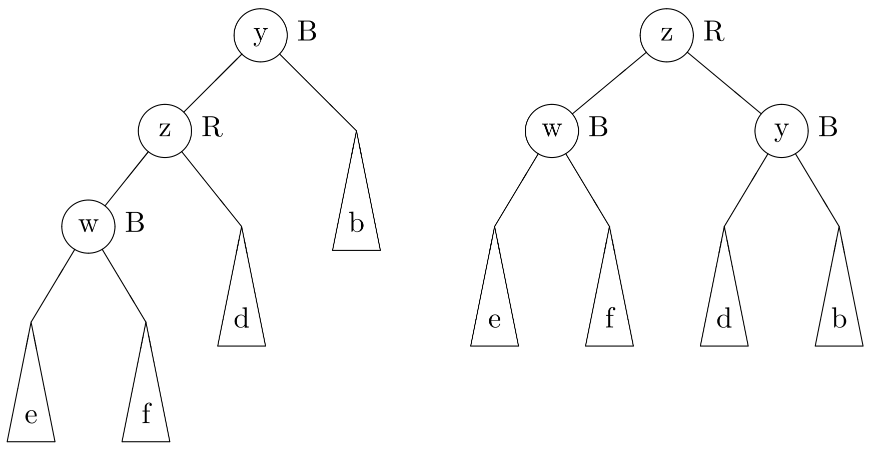

Figure 1: Inserting x<z into (T B (T R c z d) y b) with ins(x,c)=(T C e w f). For C=B, w is linked to z (left tree). For C=R, this would violate the red invariant. A single rotation to the right is performed and C is changed to B (right tree).

rotations to preserve both invariants. In the case of inserting into one of the outermost subtrees, a single rotation is performed which lifts the red father to be the new root, and the red child is colored black (fig. 1, right tree). The remaining case requires a double rotation. Again, the red child of the new red root is blackened, as can be seen on the right in figure 2. Details on this will be provided in sections 3.2.3 and 3.2.4 which present the proofs of this invariants.

As some of the recursive calls do not use immediate subtrees of the root, the function is not a primitive recursive one and can not be expressed by `primrec`. It is modeled by `recdef` using the size of the tree as measure function to prove termination. The syntactic restriction of Isabelle's case-construct forces some rearrangements of the code. It requires all constructors of a datatype to be present in exact order of definition and therefore allows neither immediate distinctions between trees with red or black root nor wildcards. As a consequence of that, all constructs of the form

```
case a of (T R c z d) => do−this
        | _ => do−that
```

must be translated to

```
case a of E => do−that
        | (T m c z d) => (case m of R => do−this
                                  | B => do−that)
```

duplicating the non-red case to meet both empty and black trees and resulting in the following definition:

consts

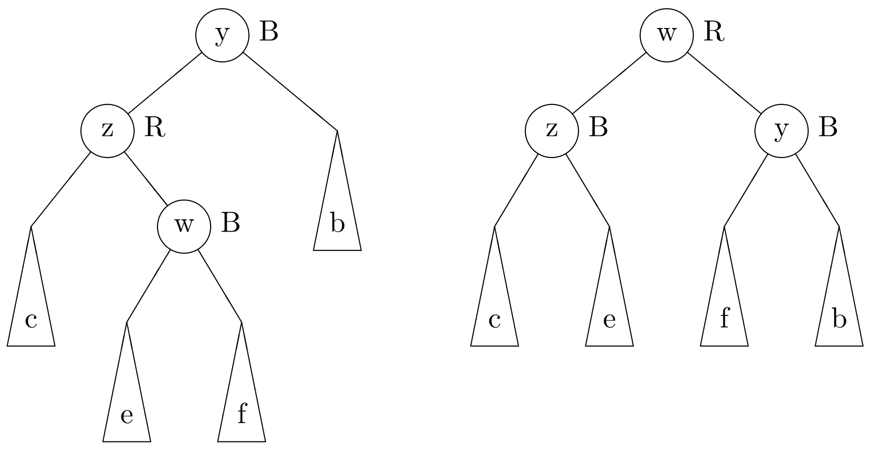

Figure 2: Inserting x, z<x<y, into (T B (T R c z d) y b) with ins(x,d)=(T C e w f). For C=B, w is linked to z (left tree). For C=R, this would violate the red invariant. A double rotation is performed and C is changed to B (right tree).

```
  ins :: "'a::LINORDER item * 'a tree => 'a tree"

recdef ins "measure (%(x,t).(size t))"
ins_empty  "ins (x, E) = T R E x E"
ins_branch
"ins (x, (T color a y b)) = (case (compare x y)
  of LESS => (case a
    of E => (T B (ins (x, a)) y b)
    | (T m c z d) => (case m
      of R => (case (compare x z)
        of LESS => (case (ins (x, c))
          of E => (T B (T R E z d) y b)
          | (T m e w f) => (case m
            of R => (T R (T B e w f) z (T B d y b))
            | B => (T B (T R (T B e w f) z d) y b)))
        | EQUAL => (T color  (T R c x d) y b)
        | GREATER => (case (ins (x, d))
          of E => (T B (T R c z E) y b)
          | (T m e w f) => (case m
            of R => (T R (T B c z e) w (T B f y b))
            | B => (T B (T R c z (T B e w f)) y b))
                    )
              )
      | B => (T B (ins (x, a)) y b))
             )
  | EQUAL => (T color a x b)
  | GREATER => (case b
    of E => (T B a y (ins (x, b)))
    | (T m c z d) => (case m
      of R =>(case (compare x z)
        of LESS => (case (ins (x, c))
          of E => (T B a y (T R E z d))
          | (T m e w f) => (case m
            of R => (T R (T B a y e) w (T B f z d))
            | B => (T B a y (T R (T B e w f) z d)))
                   )
        | EQUAL => (T color a y (T R c x d))
        | GREATER => (case (ins (x, d))
          of E => (T B a y (T R c z E))
          | (T m e w f) => (case m
            of R => (T R (T B a y c) z (T B e w f))
            | B => (T B a y (T R c z (T B e w f))))
                    )
             )
      | B => (T B a y (ins (x, b)))))
                )"
```

The `delete` function is composed of several interacting local functions. It raises an error if the element to be deleted is not present. Its structure is as follows (ommitted parts are presented below):

```
(* Remove an item.  Raises LibBase.NotFound if not found. *)
local
  datatype zipper
    = TOP
    | LEFT of (color * item * tree * zipper)
    | RIGHT of (color * tree * item * zipper)
in
fun delete (SET(nItems,t),k) = let
  fun zip [...]
  fun bbZip [...]
  fun delMin [...]
  fun join [...]
  fun del [...]
  in
    SET(nItems-1,del(t,TOP))
  end
end (* local *)
```

A `zipper` can be used to store parts of a tree which do not contain the element to be deleted in a way which allows to reconstruct the tree's original structure. An item is deleted by calling the local deletion function `del` with the empty zipper `TOP`.

```
fun del (E,z) = raise LibBase.NotFound
  | del (T(color,a,y,b),z) = (case K.compare(k,y)
        of LESS => del (a,LEFT(color,y,b,z))
         | EQUAL => join (color,a,b,z)
         | GREATER => del (b,RIGHT(color,a,y,z))
       (* end case *))
```

This function searchs for the element in the tree and simultaneously constructs the corresponding zipper. Descending to a subtree of the current node is done by adding a layer to the zipper which remembers the direction we chose (`LEFT` or `RIGHT`), the color of the current node, its element and its other subtree, the order of the latter two depending on the type of layer. If the element to be deleted is found, the function `join` is called.

```
fun join (R,E,E,z) = zip(z,E)
  | join (_,a,E,z) = #2(bbZip(z,a))    (* color = black *)
  | join (_,E,b,z) = #2(bbZip(z,b))    (* color = black *)
  | join (color,a,b,z) = let
      val (x,(needB,b')) = delMin(b,TOP)
      in
        if needB
          then #2(bbZip(z,T(color,a,x,b')))
          else zip(z,T(color,a,x,b'))
      end
```

Based on the color of the element's node, both subtrees of that node and the zipper containing all informations about the rest of the tree, `join` decides which function to call to rearrange the tree. If the deleted node was a red one with two empty children, we can just replace it by a leaf. This is done by function `zip`:

```
fun zip (TOP,t) = t
  | zip (LEFT(color,x,b,z),a) = zip(z,T(color,a,x,b))
  | zip (RIGHT(color,a,x,z),b) = zip(z,T(color,a,x,b))
```

This function inverts the construction of the zipper as done in `del` and thus reconstructs the tree stored in the zipper, linking the given tree into the reconstructed one at the position indicated by the outermost layer of the zipper.
If one subtree was empty (the node should then have been black because of the invariants), `join` calls function `bbZip` to link the other one to the node's father.

```
(* bbZip propagates a black deficit up the tree until either
 * the top is reached, or the deficit can be covered.  It
 * returns a boolean that is true if there is still a
 * deficit and the zipped tree.
 *)
  fun bbZip (TOP,t) = (true,t)
    | bbZip (LEFT(B,x,T(R,c,y,d),z),a) = (* case 1L *)
        bbZip (LEFT(R,x,c,LEFT(B,y,d,z)),a)
    | bbZip (LEFT(color,x,T(B,T(R,c,y,d),w,e),z),a) =
                                          (* case 3L *)
        bbZip (LEFT(color,x,T(B,c,y,T(R,d,w,e)),z),a)
    | bbZip (LEFT(color,x,T(B,c,y,T(R,d,w,e)),z),a) =
                                          (* case 4L *)
        (false,zip (z,T(color,T(B,a,x,c),y,T(B,d,w,e))))
    | bbZip (LEFT(R,x,T(B,c,y,d),z),a) = (* case 2L *)
        (false,zip (z,T(B,a,x,T(R,c,y,d))))
    | bbZip (LEFT(B,x,T(B,c,y,d),z),a) = (* case 2L *)
        bbZip (z,T(B,a,x,T(R,c,y,d)))
    | bbZip (RIGHT(color,T(R,c,y,d),x,z),b) = (* case 1R *)
        bbZip (RIGHT(R,d,x,RIGHT(B,c,y,z)),b)
    | bbZip (RIGHT(color,T(B,T(R,c,w,d),y,e),x,z),b) =
                                              (* case 3R *)
        bbZip (RIGHT(color,T(B,c,w,T(R,d,y,e)),x,z),b)
    | bbZip (RIGHT(color,T(B,c,y,T(R,d,w,e)),x,z),b) =
                                              (* case 4R *)
        (false,zip (z,T(color,c,y,T(B,T(R,d,w,e),x,b))))
    | bbZip (RIGHT(R,T(B,c,y,d),x,z),b) = (* case 2R *)
        (false,zip (z,T(B,T(R,c,y,d),x,b)))
    | bbZip (RIGHT(B,T(B,c,y,d),x,z),b) = (* case 2R *)
        bbZip (z,T(B,T(R,c,y,d),x,b))
    | bbZip (z,t) = (false,zip(z,t))
```

This function returns a boolean indicating whether reestablishing the black invariant has been successful and the resulting tree. Studying the details here would lead too far.

If none of the subtrees is empty, `join` needs to replace the node by its symmetric successor before reconstructing the tree. It is determined by function `delMin`:

```
fun delMin (T(R,E,y,b),z) = (y,(false,zip(z,b)))
  | delMin (T(B,E,y,b),z) = (y,bbZip(z,b))
  | delMin (T(color,a,y,b),z) = delMin(a,LEFT(color,y,b,z))
  | delMin (E,_) = raise Match
```

The result contains the symmetric successor, a boolean indicating whether there still is a black deficit to be covered from this part of the tree and the rest of the subtree. Depending on the value of the boolean, `join` calls either `zip` or `bbZip` to construct the entire tree. This completes the presentation of the original function.

In the theory, the datatype `zipper` is defined first, followed by a list of functions and their types. In the cases of `delMin` and `join`, the results are of option types because searching for the minimal element in an empty tree raises an error or leads to result `None` respectively.

```
datatype 'a zipper
       = TOP
       | LEFT color ('a item) ('a tree) ('a zipper)
       | RIGHT color ('a tree) ('a item) ('a zipper)

consts
  zip :: "'a zipper => 'a tree => 'a tree"
  bbZip :: "'a zipper * 'a tree => bool * 'a tree"
  delMin :: "'a tree * 'a zipper
               => ('a item * (bool  * 'a tree)) option "
  join :: "color * 'a tree * 'a tree * 'a zipper
                                  => ('a tree) option"
  del :: "'a::LINORDER item => 'a tree => 'a zipper
                               => ('a tree) option"
```

The reconstruction function `zip` is the one to be modeled most easily by just using `primrec`.

```
primrec
zip_top "zip TOP t = t"
zip_left "zip (LEFT color x b z) a = zip z (T color a x b)"
zip_right "zip (RIGHT color a x z) b = zip z (T color a x b)"
```

Providing a well-founded relation such that Isabelle is able to prove termination for `bbZip` requires more complex constructions. Recursive calls of that function either decrease the depth of the zipper argument (cases 2), increase the depth of the zipper and simultaneously decrease the size of the tree component of the zipper's outermost layer (cases 1), or they just perform a rotation to the right on the zipper's outermost tree (cases 3). To cover all those possibilities, the size of a zipper is measured by the accumulated size of all its tree components, i. e. the number of nodes in those trees, where nodes occuring in left subtrees are counted twice to cover rotations.

```
consts
  treesize :: "'a tree => nat"
  zippersize :: "'a zipper => nat"

primrec
treesize_empty  "treesize E = 0"
treesize_branch  "treesize (T c a x b)
                 = 1 + treesize a + treesize a + treesize b"

primrec
zippersize_top  "zippersize TOP = 0"
zippersize_left
  "zippersize (LEFT c x t z) = treesize t + zippersize z"
zippersize_right
  "zippersize (RIGHT c t x z) = treesize t + zippersize z"

recdef bbZip "measure (%(z,t).(zippersize z))"

  "bbZip (TOP, t) = (True, t)"
  "bbZip ((LEFT B x (T R c y d) z), a) =
     bbZip ((LEFT R x c (LEFT B y d z)), a)"
  "bbZip ((LEFT color x (T B (T R c y d) w e) z), a) =
     bbZip ((LEFT color x (T B c y (T R d w e)) z), a)"
  "bbZip ((LEFT color x (T B c y (T R d w e)) z), a) =
     (False, zip z (T color (T B a x c) y (T B d w e)))"
  "bbZip ((LEFT R x (T B c y d) z), a) =
     (False, zip z (T B a x (T R c y d)))"
  "bbZip ((LEFT B x (T B c y d) z), a) =
     bbZip (z, (T B a x (T R c y d)))"
  "bbZip ((RIGHT color (T R c y d) x z), b) =
     bbZip ((RIGHT R d x (RIGHT B c y z)), b)"
  "bbZip ((RIGHT color (T B (T R c w d) y e) x z), b) =
     bbZip ((RIGHT color (T B c w (T R d y e)) x z), b)"
  "bbZip ((RIGHT color (T B c y (T R d w e)) x z), b) =
     (False, zip z (T color c y (T B (T R d w e) x b)))"
  "bbZip ((RIGHT R (T B c y d) x z), b) =
     (False, zip z (T B (T R c y d) x b))"
  "bbZip ((RIGHT B (T B c y d) x z), b) =
     bbZip (z, (T B (T R c y d) x b))"
  "bbZip (z, t) = (False, zip z t)"
```

`delMin` uses detailed pattern matching and thus needs general recursion, using the tree's size as measure to guarantee termination.

```
recdef delMin "measure (%(t,z).(size t))"
  "delMin ((T R E y b), z) = Some (y, (False, zip z b))"
  "delMin ((T B E y b), z) = Some (y, bbZip(z, b))"
  "delMin ((T color a y b), z)
                  = delMin(a, (LEFT color y b z))"
  "delMin (E, z) = None"
```

Although `join` itself is not recursive, `recdef` is used here because it allows extensive pattern matching. The fourth pattern depends on the result of `delMin`, thus the result of the function is of type `('a tree) option`. The nested assignment in the let-construct has to be split into a sequence of assignments for Isabelle.

```
recdef join "{}"
  "join (R, E, E, z) = Some (zip z E)"
  "join (c, a, E, z) = Some (snd(bbZip(z, a)))"
  "join (c, E, b, z) = Some (snd(bbZip(z, b)))"
  "join (color, a, b, z) = (case (delMin(b, TOP))
     of None => None
     | (Some r) =>
       (let
        x = fst (r); needB = fst(snd(r)); b' = snd(snd(r))
        in
          if needB
            then Some (snd(bbZip(z,(T color a x b'))))
            else Some (zip z (T color a x b'))
       ))"
```

The local deletion function `del` can be directly modeled by `primrec`.

```
primrec
del_empty "del k E z = None"
del_branch "del k (T color a y b) z = (case (compare k y)
                of LESS => (del k a (LEFT color y b z))
                 | EQUAL => (join (color, a, b, z))
                 | GREATER => (del k b (RIGHT color a y z)))"
```

This completes the definition of the local functions. Finally, we define the global deletion function `delete` which calls the local function `del` with the empty zipper `TOP`.

```
constdefs
  delete :: "'a::LINORDER item => 'a tree
                                   => ('a tree) option"
  "delete k t == del k t TOP"
```

2.2 Defining Test Predicates

The second part of the theory consists of definitions of the predicates needed for the theorems to be proven.
`isin` tests for the presence of a specified element without assuming orderedness, `isord` is true if and only if the tree is ordered, i. e. if it is either empty or both its subtrees are ordered and all elements of the left subtree are smaller than the one at the root and those of the right one greater.

```
consts
  isin  :: "'a::LINORDER item => 'a tree => bool"
  isord   :: "('a::LINORDER item) tree => bool"

primrec
isin_empty "isin x E = False"
isin_branch "isin x (T c a y b) = (((compare x y) = EQUAL)
                                   | (isin x a) | (isin x b))"

primrec
isord_empty "isord E = True"
isord_branch
"isord (T c a y b) = (isord a & isord b
             & (! x. isin x a --> ((compare x y) = LESS))
             & (! x. isin x b --> ((compare x y) = GREATER)))"
```

The weak red invariant `redinv` corresponds to property (P4), i. e. no red node has a red child. The strong red invariant `R_inv` combines the weak one with property (P2), which expects the root to be black. They are considered seperately because `R_inv` doesn't always hold after insertion as one can easily see by looking at `ins(x,E)` or the cases where the recursive call results in a red tree. All those cases lead to trees with red root. The alternative insertion function `insert` fixes this problem by recoloring the root to be black after completing the insertion as it is done by Chris Okasaki [3].

```
consts
  redinv :: "('a item) tree => bool"
  R_inv :: "('a item) tree => bool"

recdef redinv "measure (%t. (size t))"
  "redinv E = True"
  "redinv (T B a y b) = (redinv a & redinv b)"
  "redinv (T R (T R a x b) y c) = False"
  "redinv (T R a x (T R b y c)) = False"
  "redinv (T R a x b) = (redinv a & redinv b)"

recdef R_inv "{}"
  "R_inv E = True"
  "R_inv (T R a y b) = False"
  "R_inv (T B a y b) = (redinv a & redinv b)"

consts
  makeBlack :: "'a tree => 'a tree"
```

```
primrec
  "makeBlack E = E"
  "makeBlack (T color a x b) = (T B a x b)"

constdefs
  insert :: "'a::LINORDER item => 'a tree => 'a tree"
  "insert x t == makeBlack (ins(x,t))"
```

Finally, `max_B_height` is defined to be the maximal number of black nodes on a path from the root to a leaf. A tree satisfies property (P5), the black invariant `blackinv`, if and only if both its subtrees satisfy this invariant and their `max_B_height` is identical. The black invariant is thus not tested by looking at all the paths from the root to leaves but by recursively comparing black heights of neighbouring subtrees, starting at the empty ones.

```
consts
  blackinv :: "('a item) tree => bool"
  max_B_height :: "('a item) tree => nat"

recdef max_B_height "measure (%t. (size t))"
  "max_B_height E = 0"
  "max_B_height (T B a y b)
     = Suc(max (max_B_height a) (max_B_height b))"
  "max_B_height (T R a y b)
     = (max (max_B_height a) (max_B_height b))"

recdef blackinv "measure (%t. (size t))"
  "blackinv E = True"
  "blackinv (T color a y b) = ((blackinv a) & (blackinv b)
                & ((max_B_height a) = (max_B_height b)))"
```

This completes the development of the theory.

3 Formal Analysis

This section presents formal proofs that show that `ins(x,t)`—received by transcribing the ML-program to HOL—results in a correct red-black tree containing all elements of the original tree `t` plus `x` provided that `t` also was a correct red-black tree. *Correct* here means ordered and satisfying both the weak red invariant `redinv` and the black invariant `blackinv`, i. e. property (P2)—the root being black—is excluded.
We first explain some general principles. Section 3.2 consists of four sections corresponding to the four main theorems for `ins(x,t)`. Large parts of all proofs are just done by appropriate case distinctions and simplifications. The underlying ideas will be presented, especially those of the more complicated parts. There are often several symmetric cases which can be solved analogously. Section 3.3 then shows that the same properties also hold for

the corrected insertion function `insert`, and that this one additionaly preserves the strong red invariant `R_inv`, i. e. property (P2).

3.1 General Remarks

3.1.1 Induction Scheme

As the insertion function `ins` has been defined as a recursive function via `recdef`, Isabelle provides the induction scheme `RBT.ins.induct` for this function. It covers all recursive calls of `ins`, i. e. those on the immediate subtrees of the root provided these are black or empty and those on the subtrees of these subtrees otherwise. It will be used for all proofs and therefore is listed in figure 3 in its general form to give an idea of the meta level assumptions used later.

3.1.2 Lemma ins_not_E

As the translation of the case-syntax forced occurences of patterns like `"case (ins (x, c)) of E"`, the first lemma to be proven states the fact that the result of `ins(x,t)` is never empty. This allows to eliminate those cases by contradiction in further proofs. The proof is done by complete case distinction following the structure of `ins` and simplification using the assumptions. The resulting lemma

ins_not_E `"ins(x,t)` $\neq$ `E"`

will be used to remove subgoals with a premise of the form `ins(x,t)=E` by contradiction.

3.2 Theorems about ins

The following proofs all start in the same way. This common scheme is presented here. Initially, induction is performed using `RBT.ins.induct` (cf. fig. 3). The first subgoal, which states the theorem for the empty tree `E`, can be solved automatically, and if the theorem consists of a chain of implications, the premises are shifted to the meta level assumptions for the remaining one. The next step replaces occurences of `ins` by its definition. This is done by simplification, where definitions of the property currently of interest are removed from the simpset to avoid undesired unfolding. Then, the subgoal is split into eleven instances of the current theorem, where `x` is the element to be inserted into `(T color a y b)`:

1. x < y, a = E
2. x < y, a = T R tree1 aa tree2, x < aa

```
⟦⋀x. ?P x E;
      ⋀x color a y b.
         ⟦∀m c z d.
               m = B ∧
               b = T m c z d ∧
               compare x y = GREATER ⟶ ?P x b;
            ∀z m c d.
               compare x z = GREATER ∧
               m = R ∧
               b = T m c z d ∧
               compare x y = GREATER ⟶ ?P x d;
            ∀z m c d.
               compare x z = LESS ∧
               m = R ∧
               b = T m c z d ∧
               compare x y = GREATER ⟶ ?P x c;
            b = E ∧ compare x y = GREATER ⟶
            ?P x b;
            ∀m c z d.
               m = B ∧
               a = T m c z d ∧
               compare x y = LESS ⟶ ?P x a;
            ∀z m c d.
               compare x z = GREATER ∧
               m = R ∧
               a = T m c z d ∧
               compare x y = LESS ⟶ ?P x d;
            ∀z m c d.
               compare x z = LESS ∧
               m = R ∧
               a = T m c z d ∧
               compare x y = LESS ⟶ ?P x c;
            a = E ∧ compare x y = LESS ⟶
            ?P x a⟧
         ⟹ ?P x (T color a y b)⟧
   ⟹ ?P ?u ?v
```

Figure 3: Induction scheme `RBT.ins.induct`

3. x < y, a = T R tree1 aa tree2, x = aa
4. x < y, a = T R tree1 aa tree2, x > aa
5. x < y, a = T B tree1 aa tree2
6. x = y
7. x > y, b = E
8. x > y, b = T R tree1 aa tree2, x < aa
9. x > y, b = T R tree1 aa tree2, x = aa
10. x > y, b = T R tree1 aa tree2, x > aa
11. x > y, b = T B tree1 aa tree2

Inserting into an empty subtree (1/7) can normally be solved by simplifications, 6, 3 and 9 require adding properties of equality to the simpset to show that replacing an element by an equal one does not change any properties of the tree. Inserting into a black subtree (5/11) is sometimes slightly more complicated, and the remaining four cases, i. e. really inserting into red subtrees, have to be split once again according to the result of the recursive call. This results in four impossible cases solved immediately by contradiction (cf. 3.1.2) and eight cases which usually are the most interesting ones.

3.2.1 Elements

We first want to prove that inserting a new element manipulates the content of a tree in the expected way.

isin_ins `"(isin y (ins (x,t))) =`
```
     ((compare y x) = EQUAL ∨ (isin y t)) "
```

This theorem is quite obvious when looking at the code, and in fact it can be proven using `case_tac`, `Asm_full_simp_tac` (replacing `EQUAL` by = where necessary) and `force` without further manipulations.

3.2.2 Orderedness

During the proof of the fact that `ins` preserves orderedness, subgoals arise where it has to be proven that inserting an element into the corresponding subtree of an ordered tree does not influence relations between elements of that subtree and the root, i. e. if for example an element less than the one at the root is inserted into the left subtree, all elements of the resulting left subtree are smaller than the one at the root. This is stated as lemma for all six possible recursive calls. All those lemmata are proven by simplification using theorem `isin_ins` and properties of linear orders.

ins_left `"(∀ x. isin x t ⟹ compare x y = LESS)`
`∧ compare z y = LESS`
`⟹ (∀ x. isin x (ins(z,t)) ⟹ compare x y = LESS)"`

ins_left_left `"(∀ x. isin x (T c t1 w t2) ⟹ compare x y = LESS) ∧ compare z y = LESS ⟹ (∀ x. isin x (T c1 (ins(z,t1)) w t2) ⟹ compare x y = LESS)"`

ins_left_right `"(∀ x. isin x (T c t1 w t2) ⟹ compare x y = LESS) ∧ compare z y = LESS ⟹ (∀ x. isin x (T c1 t1 w (ins(z,t2))) ⟹ compare x y = LESS)"`

ins_right `"(∀ x. isin x t ⟹ compare x y = GREATER)`
`∧ compare z y = GREATER`
`⟹ (∀ x. isin x (ins (z,t)) ⟹ compare x y = GREATER)"`

ins_right_left `"(∀ x. isin x (T c t1 w t2) ⟹ compare x y = GREATER) ∧ compare z y = GREATER ⟹ (∀ x. isin x (T c1 (ins(z,t1)) w t2) ⟹ compare x y = GREATER)"`

ins_right_right `"(∀ x. isin x (T c t1 w t2) ⟹ compare x y = GREATER) ∧ compare z y = GREATER ⟹ (∀ x. isin x (T c1 t1 w (ins(z,t2))) ⟹ compare x y = GREATER)"`

Moreover, it turned out to be useful to state that the color of a tree does not influence its elements, which can be proven automatically.

isin_B_R `"isin x (T B a y b) = isin x (T R a y b)"`

Proving orderedness of `ins(x,t)` only requires `t` to be ordered, thus we have to show

isord_ins `"isord t ⟶ isord(ins (x,t))"`

The first interesting case considered here is the one where a new element is inserted into the left subtree which is black. This requires proving `isord(T B (ins (x, (T B tree1 aa tree2))) y b)`, which is simplified to

```
3. ⋀x a y b colora tree1 aa tree2.
      ⟦ isord (ins (x, T B tree1 aa tree2));
       isord tree1 ∧
       isord tree2 ∧
       (∀x. isin x tree1 ⟶ compare x aa = LESS) ∧
       (∀x. isin x tree2 ⟶ compare x aa = GREATER) ∧
       isord b ∧
       (∀x. (compare x aa = EQUAL ⟶ compare x y = LESS) ∧
            (isin x tree1 ⟶ compare x y = LESS) ∧
            (isin x tree2 ⟶ compare x y = LESS)) ∧
```

```
  (∀x. isin x b ⟶ compare x y = GREATER);
  compare x y = LESS; a = T B tree1 aa tree2;
  colora = B⟧
 ⟹ ∀xa. isin xa (ins (x, T B tree1 aa tree2))
          ⟶ compare xa y = LESS
```

i. e. it has to be shown that all elements `xa` of the new left subtree are smaller than `y`, the one at the root. This is done using `isin_ins` and properties of linear orders.
Now consider the case where the left subtree `a` has a red root and inserting into its left subtree results in a black tree which directly replaces the original subtree without changing the structure of the rest of the tree.

```
2. ⋀x color a y b colora tree1 aa tree2 colorb tree1a ab
    tree2a.
      ⟦isord tree1 ⟶ isord (T B tree1a ab tree2a);
       isord (T color (T R tree1 aa tree2) y b);
       compare x y = LESS; a = T R tree1 aa tree2;
       colora = R; compare x aa = LESS;
       ins (x, tree1) = T B tree1a ab tree2a; colorb = B⟧
      ⟹isord (T B (T R (T B tree1a ab tree2a) aa tree2) y b)
```

Parts of this can be solved by simplification using the assumptions. It remains to show

```
(∀x. isin x (T B tree1a ab tree2a) ⟶ compare x aa = LESS)
∧  (∀x. isin x (T R (T B tree1a ab tree2a) aa tree2)
⟶ compare x y = LESS)
```

This means proving that the modified leftmost subtree only contains elements less than its ancestors, which has already been shown in the lemmata presented at the beginning of this section. We thus rewrite `(T B tree1a ab tree2a)` to `ins (x, tree1)` and use the corresponding lemmata `ins_left` and `ins_left_left` to complete this part.

If the result of the recursive call is a red tree, the tree will be restructured to avoid a link between two red nodes. In the case of the left left subtree (and the right right one analogously) this is done by recoloring the root of the new tree to black and performing a single rotation resulting in the following tree:

```
1. ⋀x color a y b colora tree1 aa tree2 colorb tree1a ab
    tree2a.
      ⟦isord tree1 ⟶ isord (T R tree1a ab tree2a);
       isord (T color (T R tree1 aa tree2) y b);
       compare x y = LESS; a = T R tree1 aa tree2;
       colora = R; compare x aa = LESS;
       ins (x, tree1) = T R tree1a ab tree2a; colorb = R⟧
      ⟹ isord (T R (T B tree1a ab tree2a) aa (T B tree2 y b))
```

Simplification transforms the conclusion to

$$\begin{aligned}&(\forall \texttt{x. isin x tree2} \longrightarrow \texttt{compare x y = LESS})\\ \wedge\;\; &(\forall \texttt{x. isin x (T B tree1a ab tree2a)} \longrightarrow \texttt{compare x aa = LESS})\\ \wedge\;\; &(\forall \texttt{x. isin x (T B tree2 y b)} \longrightarrow \texttt{compare x aa = GREATER}).\end{aligned}$$

The first conjunct can be solved immediately, because the original tree was ordered. The third one is simplified to

$$\forall \texttt{x. isin x b} \;\longrightarrow\; \texttt{aa < x}$$

and finally solved exploiting transitivity of $<$ and the fact that `y` was between `aa` and `b` in the ordered initial tree. After recoloring the root to red, which does not change the order of elements, the remaining conjunct

$$(\forall \texttt{x. isin x (T B tree1a ab tree2a)} \;\longrightarrow\; \texttt{compare x aa = LESS})$$

can again be solved by rewriting the tree to `ins (x, tree1)` and using the appropriate lemma. The proof of that case is then completed.
In the remainig two subgoals the tree is restructured performing double rotations and recolorations which cause the changed subtree to be split.

```
1. ⋀x color a y b colora tree1 aa tree2 colorb tree1a ab
    tree2a.
      ⟦isord tree2 ⟶ isord (T R tree1a ab tree2a);
       isord (T color (T R tree1 aa tree2) y b);
       compare x y = LESS; a = T R tree1 aa tree2;
       colora = R; compare x aa = GREATER;
       ins (x, tree2) = T R tree1a ab tree2a; colorb = R⟧
      ⟹ isord (T R (T B tree1 aa tree1a) ab (T B tree2a y b))
```

Exploiting premises and properties of linear orders leads to conclusion

$$\begin{aligned}&(\forall \texttt{x. isin x tree1a} \longrightarrow \texttt{aa < x}) \quad \wedge\\ &(\forall \texttt{x. isin x tree2a} \longrightarrow \texttt{x < y}) \quad \wedge\\ &(\forall \texttt{x. (x = aa} \longrightarrow \texttt{aa < ab}) \quad \wedge \quad (\texttt{isin x tree1} \longrightarrow \texttt{x < ab})) \quad \wedge\\ &(\forall \texttt{x. (x = y} \longrightarrow \texttt{ab < y}) \quad \wedge \quad (\texttt{isin x b} \longrightarrow \texttt{ab < x})).\end{aligned}$$

The first conjunct can be removed by exploiting the facts that `tree1a` is part of `ins (x, tree2)` and that elements of `ins (x, tree2)` are either from `tree2` and therefore greater than `aa` or equal to `x` which is by assumption greater than `aa`. The second conjunct is removed analogously.

$$(\forall \texttt{x. (x = aa} \;\longrightarrow \texttt{aa < ab}) \quad \wedge \quad (\texttt{isin x tree1} \longrightarrow \texttt{x < ab}))$$

will be shown next.

The first half, namely `aa < ab`, is solved by the fact that `ab` is the root of the result of the recursive call in the right subtree of `aa`. Remains to show

$$\texttt{(isin xa tree1} \longrightarrow \texttt{xa < ab).}$$

The premise is shifted to the meta assumptions and again we use the fact that `ab` is the root of `ins(x,tree2)` and thus either equal to `x` or in `tree2` as stated in `isin_ins`.

$$\texttt{(ab = x} \longrightarrow \texttt{xa < x)} \land \texttt{(isin ab tree2} \longrightarrow \texttt{xa < ab)}$$

`xa < ab` holds because `xa` is in `tree1` which originally was the left subtree of `aa`, thus `xa < aa`. This implies `xa < ab`. We still have to show the second conjunct

$$\texttt{(isin ab tree2} \longrightarrow \texttt{xa < ab)}$$

which is again solved using transitivity of $<$ with `aa` as intermediate element. We now show the last conjunct of the original conclusion, i. e.

$$(\forall\texttt{x. (x = y} \longrightarrow \texttt{ab < y)} \land \texttt{(isin x b} \longrightarrow \texttt{ab < x)),}$$

the root of `ins(x,tree2)` is less than the overall root and less than all elements in the right subtree.

The first half is solved by the fact that `isin ab (ins(x,tree2))`, which is true as `ab` is the root of that tree, implies `ab < y`. The same fact is used to solve the second half: `isin ab (ins(x,tree2))` implies `ab < y` and `y < xa` because of `isin xa b`. Proving the symmetric case analogously completes the proof of the theorem. We thus have shown that `ins` preserves search tree properties.

3.2.3 Weak Red Invariant

In order to proof that `ins` preserves the weak red invariant `redinv` (cf. p.12), we first prove lemma

redinv_R_B `"redinv (T R a y b)` $\longrightarrow$ `redinv (T B a y b)"`

by case distinctions and `Asm_full_simp_tac`. Case distinctions are necessary because `redinv (T R a y b)` can only be simplified using information about the subtrees.

redinv_ins `"redinv t` $\longrightarrow$ `redinv (ins(x,t))"`

is the theorem to be shown.

For inserting into a black subtree this leads to the obligation of showing that both subtrees of the result are `redinv`. In the case of `(T B (ins (x, (T B tree1 aa tree2))) y b)`, this means proving

```
3. ⋀x color a y b colora tree1 aa tree2.
     ⟦redinv tree1 ∧ redinv tree2
      ⟶ redinv (ins (x, T B tree1 aa tree2));
      redinv (T color (T B tree1 aa tree2) y b);
      compare x y = LESS; a = T B tree1 aa tree2; colora = B⟧
    ⟹  redinv (ins (x, T B tree1 aa tree2)) ∧ redinv b
```

The first half is reduced to `redinv tree1 ∧ redinv tree2` using the implication in the assumptions. We then have to instantiate `color` and `b` by case distinction such that `redinv (T color (T B tree1 aa tree2) y b)` can be simplified. This results in either a contradiction in the premises or in the fact that all subtrees satisfy the invariant and thus proves our current claim.

The rest of the proof is done similarly. In the cases where the result of the recursive call is red, an appropriate instance of the lemma is added to the premises to be able to do so.

3.2.4 Black Invariant

As we have already proven that trees built by `ins` starting at the empty tree are `redinv`, we add `redinv t` to the assumptions for showing that insertion preserves the black invariant `blackinv` (cf. p.12). This reduces the number of cases to be considered and ensures that recursive calls on subtrees of red nodes are calls on either empty or black trees. Insertion into empty subtrees can be handled by simplification. Calls on black subtrees allow using the following lemma which states that inserting an element into a tree with black root which is both `redinv` and `blackinv` does not change the tree's black height.

max_B_height_ins_B `"(EX a y b.  t=(T B a y b))`
```
    ⟶ redinv t ⟶ blackinv t
    ⟶ max_B_height (ins (x,t)) = max_B_height t"
```

Apart from inserting fresh elements into red subtrees, everything can be proven just by simplifications. The remaining cases can be completly reduced to equations about connections between black heights of different subtrees. In the case of the recursive call on the left left subtree with red result, which leads to `T R (T B tree1a ab tree2a) aa (T B tree2 y b)`, this means showing

```
1. ⋀x color a y b colora tree1 aa tree2 colorb tree1a ab
   tree2a.
     ⟦(∃a y b. tree1 = T B a y b) ⟶
      redinv tree1 ⟶
      max (max_B_height tree1a) (max_B_height tree2a) =
   max_B_height tree2;
   color = B; compare x y = LESS; a = T R tree1 aa tree2;
   colora = R; compare x aa = LESS;
   ins (x, tree1) = T R tree1a ab tree2a;
   colorb = R; redinv (T R tree1 aa tree2); redinv b;
   blackinv tree1; blackinv tree2;
   max_B_height tree1 = max_B_height tree2; blackinv b;
   max (max_B_height tree1) (max_B_height tree2)
   = max_B_height b⟧
  ⟹ max (max (max_B_height tree1a) (max_B_height tree2a))
     (max (max_B_height tree2) (max_B_height b)) =
    max (max_B_height tree2) (max_B_height b)
```

`(max (max_B_height tree1a) (max_B_height tree2a))` can be replaced by `max_B_height tree2` provided that `redinv tree1` holds. In this case, the above equation becomes trivial. We thus exploit the red invariant given in the assumptions by case distinctions on `tree1` and `tree2` and simplifications. Proofs for the second and third subsubtree in some cases require to show a chain of implications between inequalities which can be solved by another case distinction on an appropriate inequality. In the case of the right subtree of `a`, both subtrees of `a` being black, this is

$$
\begin{array}{l}
(\texttt{max\_B\_height tree2a} \leq \texttt{max\_B\_height b} \\
\longrightarrow \texttt{max\_B\_height b} \leq \texttt{max\_B\_height tree1a} \\
\longrightarrow \texttt{max\_B\_height tree1a} \leq \texttt{max\_B\_height b}) \\
\wedge \quad \texttt{max\_B\_height tree2a} \leq \texttt{max\_B\_height b}
\end{array}
$$

which is solved by case distinction on

$$\texttt{max\_B\_height tree1a} \leq \texttt{max\_B\_height tree2a}.$$

This allows for establishing the theorem

blackinv_ins `"(redinv t ∧ blackinv t) ⟶ blackinv (ins(x,t))"`

Inserting into the left black subtree requires proving `blackinv (T B (ins (x, T B tree1 aa tree2)) y b)` which reduces to

```
5. ⋀x color a y b colora tree1 aa tree2.
      ⟦redinv tree1 ∧ redinv tree2 ⟶
       blackinv (ins (x, T B tree1 aa tree2));
       redinv (T color (T B tree1 aa tree2) y b);
       blackinv tree1 ∧
       blackinv tree2 ∧
       max_B_height tree1 = max_B_height tree2 ∧
```

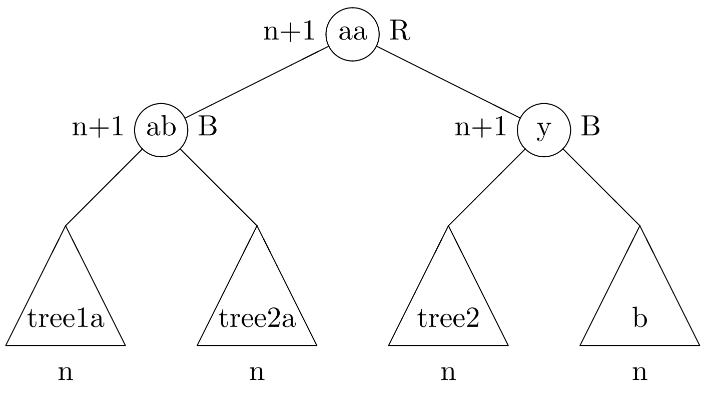

Figure 4: `ins(x, (T B (T R tree1 aa tree2) y b))` with black heights of subtrees, `ins(x, tree1)` resulting in `(T R tree1a ab tree2a)`

```
      blackinv b ∧
      Suc (max (max_B_height tree1) (max_B_height tree2)) =
      max_B_height b;
      compare x y = LESS; a = T B tree1 aa tree2; colora = B⟧
     ⟹ blackinv (ins (x, T B tree1 aa tree2)) ∧
       max_B_height (ins (x, T B tree1 aa tree2))
       = max_B_height b
```

We thus have to show that the result of the recursive call is `blackinv`, which is implied by the original subtrees being `redinv`, and that it has the same black height as the original subtree, which can be shown using the above lemma. In order to use `redinv (T color (T B tree1 aa tree2) y b)`, we have to do case distinctions on `color` and `b`. Most cases can then be solved by simplification.

For `color = R` and `b = T B tree1a ab tree2a` we have to show

```
max_B_height (ins (x, T B tree1 aa tree2))
  = Suc (max_B_height tree2a).
```

`tree2a` is the right subtree of the black tree `b` and therefore has black height one less than `b` which itself has same black height as `(T B tree1 aa tree2)` because of the black invariant holding for the original tree. Using the lemma about black heights completes this part of the proof.

Inserting into the left subtree of the red tree `a` with red result leads to `(T R (T B tree1a ab tree2a) aa (T B tree2 y b))` (fig. 4). We then have to show that

```
1. ⋀x color a y b colora tree1 aa tree2 colorb tree1a ab
    tree2a.
      ⟦redinv tree1 ⟶
    blackinv tree1a ∧ blackinv tree2a
    ∧ max_B_height tree1a = max_B_height tree2a;
    redinv (T color (T R tree1 aa tree2) y b);
    compare x y = LESS; a = T R tree1 aa tree2; colora = R;
    compare x aa = LESS;
    ins (x, tree1) = T R tree1a ab tree2a; colorb = R;
    blackinv tree1; blackinv tree2;
    max_B_height tree1 = max_B_height tree2; blackinv b;
    max (max_B_height tree1) (max_B_height tree2)
    = max_B_height b⟧
  ⟹ blackinv tree1a ∧
      blackinv tree2a ∧
      max_B_height tree1a = max_B_height tree2a ∧
      max_B_height tree2 = max_B_height b ∧
      max (max_B_height tree1a) (max_B_height tree2a) =
      max (max_B_height tree2) (max_B_height b)
```

i. e. the result of the recursive call is `blackinv` (first three conjuncts) and its subtrees have the same black height as the second subtree of the root and the second subtree of the left subtree. Being `blackinv` is due to `tree1` being `redinv` (as the whole tree is `redinv`). `tree2` and `b` have the same black height because `tree1` and `tree2` have the same black height and `b`'s black height is the maximum of those two. As the whole tree is `redinv` and `tree1` has a red father, `tree1` has to be either empty or black. The first case can be done by case distinctions and simplifications. In the second one, we can again use the lemma to show the last equality. Case distinctions on the components of the original tree are necessary to simplify and use the red invariant.

For `b = T R tree1b ac tree2b` and `tree1 = T B tree1c ad tree2c`, we have to show

$$\texttt{max\_B\_height tree2a} = \texttt{max\_B\_height tree2b}$$

which is done via

$$\begin{aligned}\text{max\_B\_height tree2a} &= \text{max\_B\_height (T R tree1a ab tree2a)}\\ &= \text{max\_B\_height (ins (x, T B tree1c ad tree2c))}\\ &= \text{max\_B\_height (T B tree1c ad tree2c)}\\ &= \text{max\_B\_height tree2b.}\end{aligned}$$

The first step is obvious, the second one just changes the representation of the tree, the third one uses the lemma and the last one is done automatically using the assumptions. The rest of the first subgoal is solved the same way.

For the new element being inserted in `a`'s right subtree with red result—leading to `(T R (T B tree1 aa tree1a) ab (T B tree2a y b))`, figure

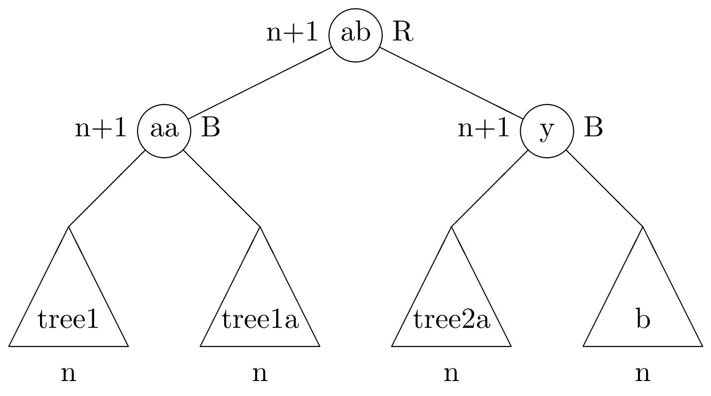

Figure 5: `ins(x, (T B (T R tree1 aa tree2) y b))` with black heights of subtrees, `ins(x, tree2)` resulting in `(T R tree1a ab tree2a)`

5—the conclusion to be shown is

```
blackinv tree1a  ∧  max_B_height tree2 = max_B_height tree1a
∧  blackinv tree2a  ∧  max_B_height tree2a = max_B_height b
∧  max (max_B_height tree2) (max_B_height tree1a)
= max (max_B_height tree2a) (max_B_height b)
```

This is what remains of `blackinv(T R (T B tree1 aa tree1a) ab (T B tree2a y b))` after simplification, where `max_B_height tree1` has been automatically replaced by `max_B_height tree2`. We thus must prove that all four subtrees have the same black height, and that the two new ones satisfy the black invariant. Most of this can again be proven out of the assumptions using similar case distinctions as before.
For `b = T R tree1b ac tree2b` and `tree2 = T B tree1c ad tree2c` it simplifies to

```
max_B_height tree2b = max_B_height tree2a
∧  max_B_height tree2a = max_B_height tree2b
∧  max (max_B_height tree2b) (max_B_height tree2a)
= max (max_B_height tree2a) (max_B_height tree2b).
```

It is sufficient to prove the first conjunct

```
max_B_height tree2b = max_B_height tree2a,
```

which implies the rest of the goal and can be shown as above.
The remaining cases of this goal as well as the other goals with red result are done similar.

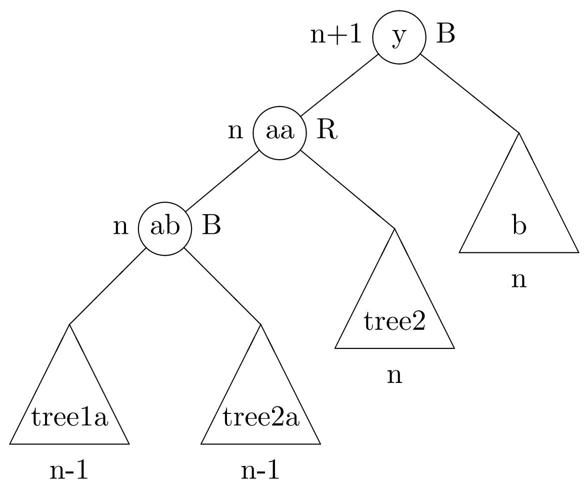

Figure 6: `ins(x, (T B (T R tree1 aa tree2) y b))` with black heights of subtrees, `ins(x,tree1)` resulting in `(T B tree1a ab tree2a)`

If inserting into the left subtree of the left subtree results in a black tree (cf. figure 6), the initial conclusion is

```
blackinv tree1a   ∧   blackinv tree2a   ∧
max_B_height tree1a = max_B_height tree2a   ∧
Suc (max (max_B_height tree1a) (max_B_height tree2a))
= max_B_height tree2   ∧
max (Suc (max (max_B_height tree1a) (max_B_height tree2a)))
(max_B_height tree2) = max_B_height b.
```

The first three conjuncts express that the resulting tree must satisfy the black invariant, the other two that `(T B tree1a ab tree2a)`, i. e. the result of `ins(x,tree1)`, has same black height as `tree2` and `b`.
For `tree2 = T B tree1d ae tree2d` and `b = T R tree1b ac tree2b` this reduces to

```
max_B_height tree2a = max_B_height tree2d   ∧
Suc (max (max_B_height tree2a) (max_B_height tree2d))
= max_B_height tree2b
```

which is implied by

```
Suc (max_B_height tree2a) = Suc (max_B_height tree2d).
```

This is true because of

Suc (max_B_height tree2a)

$$\begin{aligned}
&= \text{max\_B\_height (T B tree1a ab tree2a)}\\
&= \text{max\_B\_height (ins (x, T B tree1c ad tree2c))}\\
&= \text{max\_B\_height (T B tree1c ad tree2c)}\\
&= \text{Suc (max\_B\_height tree2d)}
\end{aligned}$$

Similar procedures are used to complete the proof. It has thus been shown that trees built by `ins` are both `redinv` and `blackinv`.

3.3 Theorems about insert

As mentioned before, the problem of the implementation, that the invariant (P2) is violated, can be fixed by making the root black after insertion, which is done by function `insert` in the theory. The following section adapts the above proofs of the main theorems to that function, the next one proves that `insert` preserves the strong red invariant.

3.3.1 Main Theorems for insert

The four main theorems concerning `ins` are used to prove the following corresponding ones for `insert`:

isin_insert `"(isin y (insert x t))`
`= ((compare y x) = EQUAL ∨ (isin y t)) "`

isord_insert `"isord t ⟶ isord(insert x t)"`

redinv_insert `"redinv t ⟶ redinv (insert x t)"`

blackinv_insert `"(redinv t ∧ blackinv t)`
`⟶ blackinv (insert x t)"`

Each time, `(insert x t)` is first replaced by its definition `(makeBlack (ins(x,t)))`. We then perform case distinction on `ins(x,t)`, removing the case of the empty tree by contradiction and solving the other two by using the corresponding theorem about `ins`. In the case of `isin_insert`, for instance, we have to show

```
1. ⋀color tree1 a tree2.
      ins (x, t) = T color tree1 a tree2 ⟹
      isin y (T B tree1 a tree2)
      = (compare y x = EQUAL ∨ isin y t)
```

which, after case distinction on `color`, can be solved by

$$\begin{aligned}
\texttt{isin y (T B tree1 a tree2)} &= \texttt{isin y (ins(x,t))}\\
&= \texttt{(compare y x = EQUAL} \lor \texttt{isin y t).}
\end{aligned}$$

The second step corresponds to `isin_ins`, the first one is done reusing the lemma about colors and contents in the case of the red root and by mere simplification otherwise.

The remaining theorems are proven similarly, using a chain of implications of the form P(t)⟶P(ins(x,t))⟶P(insert x t) to reduce them to the ones about `ins`.

3.3.2 Strong Red Invariant

Finally, we will show that trees built by `insert` in fact satisfy the strong red invariant `R_inv` (cf. p.12).

R_inv_insert `"R_inv t ⟶ R_inv (insert x t)"`

If inserting into a black tree results in a red one, we have to show

```
1. ⋀color tree1 a tree2 colora tree1a aa tree2a.
      ⟦ins (x, T B tree1a aa tree2a) = T R tree1 a tree2;
         color = R; redinv tree1a ∧ redinv tree2a;
         t = T B tree1a aa tree2a; colora = B⟧
      ⟹ redinv tree1 ∧ redinv tree2
```

The assumption `R_inv t` has been simplified to `redinv tree1a ∧ redinv tree2a`, which is equal to `redinv t`. This implies `redinv(ins(x,t))`, as has been proven in theorem `redinv_ins`. We thus have to show

```
redinv (T R tree1 a tree2) ⟶ redinv tree1 ∧ redinv tree2
```

This can be done by case distinctions on both subtrees. If at least one of them is red, the premise will be false, otherwise, it will be simplified to the conclusion. The rest of the proof is done similarly. The alternative insertion function `insert` thus preserves all considered invariants.

4 Conclusion

This work examines formal properties of red-black trees used to implement sets in the library of Standard ML of New Jersey. Red-black trees as well as the insertion and deletion functions are modelled in an Isabelle/HOL theory.

SML implementation of both functions revealed some major errors.

The delete function is able to produce trees which violate up to three essential properties of red-black trees. Those false trees have a red root, red nodes with red children and different numbers of black nodes on paths from the root to a leaf. Correction of this will be a lot more difficult than in the case of insertion. Therefore, the presented theory just models the original version.

The insertion function violates one property of red-black trees, namely the one stating that the root of a red-black tree is always black. This is corrected by modifying the function such that after completing the insertion with the original function, the color of the root is set to black. The SML implementation of this corrected version can be found in appendix B.
The HOL theory is used to formally prove that the original insertion function preserves all other invariants, i. e. it maintains search tree properties and it guarantees that there are neither red nodes with red children nor different numbers of black nodes on different paths from the root to some leaf. Afterwards, it is shown that this also holds for the corrected function. The formal analysis ends with the proof of the fact that the corrected function ensures the root being black. It thus has been verified that this extended version of the insertion function maintains all properties of red-black trees.

References

[1] redblack-set-fn.sml. Standard ML of New Jersey, version 110.0.7. http://www.smlnj.org

[2] http://isabelle.in.tum.de

[3] Chris Okasaki. Functional Pearls: Red-Black trees in a functional setting. Journal of functional programming 9(4), 1999

[4] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest. Introduction to algorithms, 2nd ed. Cambridge, Mass.: MIT Press, 2001

[5] Lawrence C. Paulson. The Isabelle Reference Manual. 2003

[6] Tobias Nipkow, Lawrence C. Paulson, Markus Wenzel. Isabelle's Logics: HOL. 2003

[7] Riccardo Pucella. Notes on Programming Standard ML of New Jersey. Cornell University, 2001. http://www.cs.cornell.edu/riccardo/smlnj.html

[8] Lawrence C. Paulson. ML for the working programmer. Cambridge University Press, 1996

A RBT

```
(* Title:     RBT.thy
   Author:    Angelika Kimmig

RedBlackTrees as used for implementing sets in smlnj redblack-set-fn.sml, smlnj
110.0.7

*)

RBT = Main +

(* mimicing ml's datatype order *)

datatype ml-order = LESS | EQUAL | GREATER

(* signature ORD-KEY: type ord-key and val compare : ord-key * ord-key ->
order *)

axclass ord-key < type

consts
  compare :: 'a::ord-key => 'a => ml-order

axclass LINORDER < linorder, ord-key
  LINORDER-less ((compare x y) = LESS) = (x < y)
  LINORDER-equal ((compare x y) = EQUAL) = (x = y)
  LINORDER-greater ((compare x y) = GREATER) = (y < x)

(* defining redblacktrees *)

types 'a item = 'a::ord-key

datatype color = R | B

datatype 'a tree = E | T color ('a tree) ('a item) ('a tree)

(* ins as used by add, element to be inserted given as first argument, case-distinction
expanded to meet Isabelle's syntactic constraints *)

consts
  ins :: 'a::LINORDER item * 'a tree => 'a tree

recdef ins measure (%(x,t).(size t))
ins-empty  ins (x, E) = T R E x E
ins-branch
ins (x, (T color a y b)) = (case (compare x y)
  of LESS => (case a
    of E => (T B (ins (x, a)) y b)
    | (T m c z d) => (case m
      of R => (case (compare x z)
        of LESS => (case (ins (x, c))
          of E => (T B (T R E z d) y b)
          | (T m e w f) => (case m
            of R => (T R (T B e w f) z (T B d y b))
            | B => (T B (T R (T B e w f) z d) y b)))
        | EQUAL => (T color  (T R c x d) y b)
        | GREATER => (case (ins (x, d))
          of E => (T B (T R c z E) y b)
          | (T m e w f) => (case m
            of R => (T R (T B c z e) w (T B f y b))
            | B => (T B (T R c z (T B e w f)) y b))
                  )
              )
      | B => (T B (ins (x, a)) y b))
            )
  | EQUAL => (T color a x b)
  | GREATER => (case b
    of E => (T B a y (ins (x, b)))
    | (T m c z d) => (case m
      of R =>(case (compare x z)
        of LESS => (case (ins (x, c))
          of E => (T B a y (T R E z d))
          | (T m e w f) => (case m
            of R => (T R (T B a y e) w (T B f z d))
            | B => (T B a y (T R (T B e w f) z d)))
                )
        | EQUAL => (T color a y (T R c x d))
        | GREATER => (case (ins (x, d))
          of E => (T B a y (T R c z E))
          | (T m e w f) => (case m
            of R => (T R (T B a y c) z (T B e w f))
            | B => (T B a y (T R c z (T B e w f))))
                  )
            )
      | B => (T B a y (ins (x, b)))))
              )

(* deletion *)
(* originally local datatype, used to memorize information about structure and con-
tent of tree *)
datatype 'a zipper
       = TOP
       | LEFT color ('a item) ('a tree) ('a zipper)
       | RIGHT color ('a tree) ('a item) ('a zipper)

(* functions used by delete *)
consts
```

```
  zip :: 'a zipper => 'a tree => 'a tree
  bbZip :: 'a zipper * 'a tree => bool * 'a tree
  delMin :: 'a tree * 'a zipper => ('a item * (bool  * 'a tree)) option
  join :: color * 'a tree * 'a tree * 'a zipper => ('a tree) option
  del :: 'a::LINORDER item => 'a tree => 'a zipper => ('a tree) option

(* reconstruction of the tree, second argument is changed subtree *)
primrec
zip-top  zip TOP t = t
zip-left  zip (LEFT color x b z) a = zip z (T color a x b)
zip-right  zip (RIGHT color a x z) b = zip z (T color a x b)

(* needed to construct wfo for bbZip *)
consts
  treesize :: 'a tree => nat
  zippersize :: 'a zipper => nat

(* left subtrees have to be weighted stronger in order to cope with rotations *)
primrec
treesize-empty  treesize E = 0
treesize-branch  treesize (T c a x b) = 1 + treesize a + treesize a + treesize b

(* sum up weights of trees in the zipper *)
primrec
zippersize-top  zippersize TOP = 0
zippersize-left  zippersize (LEFT c x t z) = treesize t + zippersize z
zippersize-right  zippersize (RIGHT c t x z) = treesize t + zippersize z

(* reconstructing trees, the new subtree having a black deficit *)
recdef bbZip measure (%(z,t).(zippersize z))
  bbZip (TOP, t) = (True, t)
  bbZip ((LEFT B x (T R c y d) z), a) =
              bbZip ((LEFT R x c (LEFT B y d z)), a)  (* case 1L *)
  bbZip ((LEFT color x (T B (T R c y d) w e) z), a) =
              bbZip ((LEFT color x (T B c y (T R d w e)) z), a) (* case 3L *)
  bbZip ((LEFT color x (T B c y (T R d w e)) z), a) =
              (False, zip z (T color (T B a x c) y (T B d w e))) (* case 4L *)
  bbZip ((LEFT R x (T B c y d) z), a) =
              (False, zip z (T B a x (T R c y d))) (* case 2L *)
  bbZip ((LEFT B x (T B c y d) z), a) =
              bbZip (z, (T B a x (T R c y d))) (* case 2L *)
  bbZip ((RIGHT color (T R c y d) x z), b) =
              bbZip ((RIGHT R d x (RIGHT B c y z)), b) (* case 1R *)
  bbZip ((RIGHT color (T B (T R c w d) y e) x z), b) =
              bbZip ((RIGHT color (T B c w (T R d y e)) x z), b) (* case 3R *)
  bbZip ((RIGHT color (T B c y (T R d w e)) x z), b) =
              (False, zip z (T color c y (T B (T R d w e) x b))) (* case 4R *)
  bbZip ((RIGHT R (T B c y d) x z), b) =
              (False, zip z (T B (T R c y d) x b)) (* case 2R *)
  bbZip ((RIGHT B (T B c y d) x z), b) =
              bbZip (z, (T B (T R c y d) x b)) (* case 2R *)
  bbZip (z, t) = (False, zip z t)

(* getting symmetric successor, information about deficit and resulting subtree *)
recdef delMin measure (%(t,z).(size t))
  delMin ((T R E y b), z) = Some (y, (False, zip z b))
  delMin ((T B E y b), z) = Some (y, bbZip(z, b))
  delMin ((T color a y b), z) = delMin(a, (LEFT color y b z))
  delMin (E, z) = None  (* raise Match *)

(* replaced - by c, let (x, (needB, b')) = delMin(b, TOP) by sequence of assigne-
ments *)
(* choose appropriate function for reconstruction *)
recdef join {}
  join (R, E, E, z) = Some (zip z E)
  join (c, a, E, z) = Some (snd(bbZip(z, a))) (* color = black *)
  join (c, E, b, z) = Some (snd(bbZip(z, b))) (* color = black *)
  join (color, a, b, z) = (case (delMin(b, TOP)) of None => None
                         | (Some r) =>
              (let
             x = fst (r); needB = fst(snd(r)); b' = snd(snd(r))
             in
               if needB
                 then Some (snd(bbZip(z,(T color a x b'))))
                 else Some (zip z (T color a x b'))
             ))

(* delete k *)
primrec
del-empty del k E z = None (* raise LibBase.NotFound    *)
del-branch del k (T color a y b) z = (case (compare k y)
              of LESS => (del k a (LEFT color y b z))
               | EQUAL => (join (color, a, b, z))
               | GREATER => (del k b (RIGHT color a y z)))

(* delete k in t using del with empty zipper *)
constdefs
  delete :: 'a::LINORDER item => 'a tree => ('a tree) option
  delete k t == del k t TOP

(* end of translation *)

(* extended version of inserting as in Okasaki's implementation, coloring the root
black after finishing insertion *)

consts
  makeBlack :: 'a tree => 'a tree
```

```
primrec
  makeBlack E = E
  makeBlack (T color a x b) = (T B a x b)

constdefs
  insert :: 'a::LINORDER item => 'a tree => 'a tree
  insert x t == makeBlack (ins(x,t))

(∗ invariants ∗)

consts
  isin :: 'a::LINORDER item => 'a tree => bool
  isord  :: ('a::LINORDER item) tree => bool
  redinv :: ('a item) tree => bool
  R-inv :: ('a item) tree => bool
  blackinv :: ('a item) tree => bool
  max-B-height :: ('a item) tree => nat

(∗ test for elements ∗)
primrec
isin-empty isin x E = False
isin-branch isin x (T c a y b) = (((compare x y) = EQUAL) | (isin x a) | (isin x
b))

(∗ test for order of elements ∗)
primrec
isord-empty isord E = True
isord-branch isord (T c a y b) = (isord a & isord b & (! x. isin x a −−> ((compare
x y) = LESS)) & (! x. isin x b −−> ((compare x y) = GREATER)))

(∗ weak red invariant: no red node has a red child ∗)
recdef redinv measure (%t. (size t))
  redinv E = True
  redinv (T B a y b) = (redinv a & redinv b)
  redinv (T R (T R a x b) y c) = False
  redinv (T R a x (T R b y c)) = False
  redinv (T R a x b) = (redinv a & redinv b)

(∗ strong red invariant: every red node has an immediate black ancestor, i.e. the
root is black and the weak red invariant holds ∗)
recdef R-inv {}
  R-inv E = True
  R-inv (T R a y b) = False
  R-inv (T B a y b) = (redinv a & redinv b)

(∗ calculating maximal number of black nodes on any path from root to leaf ∗)
recdef max-B-height measure (%t. (size t))
  max-B-height E = 0
  max-B-height (T B a y b) = Suc(max (max-B-height a) (max-B-height b))
```

max-B-height (*T R a y b*) = (*max* (*max-B-height a*) (*max-B-height b*))

(∗ *black invariant*: *number of black nodes equal on all pathes from root to leaf* ∗)
recdef blackinv measure (%*t*. (*size t*))
blackinv E = *True*
blackinv (*T color a y b*) = ((*blackinv a*) & (*blackinv b*) & ((*max-B-height a*) = (*max-B-height b*)))

end

B Corrected Extract from redblack-set-fn.sml

```
(* Extract from redblack-set-fn.sml
 *
 * COPYRIGHT (c) 1999 Bell Labs, Lucent Technologies.
 *
 * modified by Angelika Kimmig:
 * - corrected function add using an additional local function
 *   makeBlack to guarantee black root after insertion
 *
 * This code is based on Chris Okasaki's implementation of
 * red-black trees.  The linear-time tree construction code is
 * based on the paper "Constructing red-black trees" by Hinze,
 * and the delete function is based on the description in Cormen,
 * Leiserson, and Rivest.
 *
 * A red-black tree should satisfy the following two invariants:
 *
 *   Red Invariant: each red node has a black parent.
 *
 *   Black Condition: each path from the root to an empty node has the
 *     same number of black nodes (the tree's black height).
 *
 * The Red condition implies that the root is always black and the Black
 * condition implies that any node with only one child will be black and
 * its child will be a red leaf.
 *)

functor RedBlackSetFn (K : ORD_KEY) :> ORD_SET where Key = K =
 struct

  structure Key = K

  type item = K.ord_key

  datatype color = R | B

  datatype tree
```

```
    = E
    | T of (color * tree * item * tree)

datatype set = SET of (int * tree)

[...]

fun add (SET(nItems, m), x) = let
      val nItems' = ref nItems
      fun ins E = (nItems' := nItems+1; T(R, E, x, E))
        | ins (s as T(color, a, y, b)) = (case K.compare(x, y)
             of LESS => (case a
                   of T(R, c, z, d) => (case K.compare(x, z)
                         of LESS => (case ins c
                               of T(R, e, w, f) =>
                                    T(R, T(B,e,w,f), z, T(B,d,y,b))
                                | c => T(B, T(R,c,z,d), y, b)
                              (* end case *))
                          | EQUAL => T(color, T(R, c, x, d), y, b)
                          | GREATER => (case ins d
                               of T(R, e, w, f) =>
                                    T(R, T(B,c,z,e), w, T(B,f,y,b))
                                | d => T(B, T(R,c,z,d), y, b)
                              (* end case *))
                        (* end case *))
                    | _ => T(B, ins a, y, b)
                  (* end case *))
              | EQUAL => T(color, a, x, b)
              | GREATER => (case b
                   of T(R, c, z, d) => (case K.compare(x, z)
                         of LESS => (case ins c
                               of T(R, e, w, f) =>
                                    T(R, T(B,a,y,e), w, T(B,f,z,d))
                                | c => T(B, a, y, T(R,c,z,d))
                              (* end case *))
                          | EQUAL => T(color, a, y, T(R, c, x, d))
                          | GREATER => (case ins d
                               of T(R, e, w, f) =>
                                    T(R, T(B,a,y,c), z, T(B,e,w,f))
                                | d => T(B, a, y, T(R,c,z,d))
                              (* end case *))
                        (* end case *))
                    | _ => T(B, a, y, ins b)
                  (* end case *))
            (* end case *))
      fun makeBlack E = E
        | makeBlack (T(color, a, x, b)) = T(B, a, x, b)
      val m = makeBlack (ins m)                          (* corrected *)
      in
```

```
          SET(!nItems', m)
        end

    [...]

    (* Remove an item.  Raises LibBase.NotFound if not found. *)
      local
        datatype zipper
          = TOP
          | LEFT of (color * item * tree * zipper)
          | RIGHT of (color * tree * item * zipper)
      in
      fun delete (SET(nItems, t), k) = let
            fun zip (TOP, t) = t
              | zip (LEFT(color, x, b, z), a) = zip(z, T(color, a, x, b))
              | zip (RIGHT(color, a, x, z), b) = zip(z, T(color, a, x, b))
          (* bbZip propagates a black deficit up the tree until either the top
           * is reached, or the deficit can be covered.  It returns a boolean
           * that is true if there is still a deficit and the zipped tree.
           *)
            fun bbZip (TOP, t) = (true, t)
              | bbZip (LEFT(B, x, T(R, c, y, d), z), a) = (* case 1L *)
                  bbZip (LEFT(R, x, c, LEFT(B, y, d, z)), a)
              | bbZip (LEFT(color, x, T(B, T(R, c, y, d), w, e), z), a) =
                                                                  (* case 3L *)
                  bbZip (LEFT(color, x, T(B, c, y, T(R, d, w, e)), z), a)
              | bbZip (LEFT(color, x, T(B, c, y, T(R, d, w, e)), z), a) =
                                                                  (* case 4L *)
                  (false, zip (z, T(color, T(B, a, x, c), y, T(B, d, w, e))))
              | bbZip (LEFT(R, x, T(B, c, y, d), z), a) = (* case 2L *)
                  (false, zip (z, T(B, a, x, T(R, c, y, d))))
              | bbZip (LEFT(B, x, T(B, c, y, d), z), a) = (* case 2L *)
                  bbZip (z, T(B, a, x, T(R, c, y, d)))
              | bbZip (RIGHT(color, T(R, c, y, d), x, z), b) = (* case 1R *)
                  bbZip (RIGHT(R, d, x, RIGHT(B, c, y, z)), b)
              | bbZip (RIGHT(color, T(B, T(R, c, w, d), y, e), x, z), b) =
                                                                  (* case 3R *)
                  bbZip (RIGHT(color, T(B, c, w, T(R, d, y, e)), x, z), b)
              | bbZip (RIGHT(color, T(B, c, y, T(R, d, w, e)), x, z), b) =
                                                          (* case 4R *)
                  (false, zip (z, T(color, c, y, T(B, T(R, d, w, e), x, b))))
              | bbZip (RIGHT(R, T(B, c, y, d), x, z), b) = (* case 2R *)
                  (false, zip (z, T(B, T(R, c, y, d), x, b)))
              | bbZip (RIGHT(B, T(B, c, y, d), x, z), b) = (* case 2R *)
                  bbZip (z, T(B, T(R, c, y, d), x, b))
              | bbZip (z, t) = (false, zip(z, t))
            fun delMin (T(R, E, y, b), z) = (y, (false, zip(z, b)))
              | delMin (T(B, E, y, b), z) = (y, bbZip(z, b))
              | delMin (T(color, a, y, b), z) = delMin(a, LEFT(color, y, b, z))
        | delMin (E, _) = raise Match
      fun join (R, E, E, z) = zip(z, E)
        | join (_, a, E, z) = #2(bbZip(z, a))        (* color = black *)
        | join (_, E, b, z) = #2(bbZip(z, b))        (* color = black *)
        | join (color, a, b, z) = let
            val (x, (needB, b')) = delMin(b, TOP)
            in
              if needB
                then #2(bbZip(z, T(color, a, x, b')))
                else zip(z, T(color, a, x, b'))
            end
      fun del (E, z) = raise LibBase.NotFound
        | del (T(color, a, y, b), z) = (case K.compare(k, y)
             of LESS => del (a, LEFT(color, y, b, z))
              | EQUAL => join (color, a, b, z)
              | GREATER => del (b, RIGHT(color, a, y, z))
            (* end case *))
      in
        SET(nItems-1, del(t, TOP))
      end
  end (* local *)

 [...]

end;
```