An Agda implementation of deletion in Left-leaning Red-Black trees

Julien Oster, Ludwig-Maximilians-Universität München

December 20, 2010

We present an implementation of deletion in Left-leaning Red-Black-trees in Literate Agda, based on a modified data structure originally by **?**, and explain in detail how Agda ensures correctness by definition of the algorithm.

1 Introduction and Related Work

Red-Black trees have become an ubiquitous data structure to implement ordered sets and maps, yet they are also known for leading to unwieldy and complex implementations that are prone to bugs, especially for deletion. **?** realizes this and presents a new variant of the Red-Black tree, the Left-leaning Red-Black tree, that aims at keeping the known complexity of implementations down by reducing the number of possible node configurations.

? showed how some invariants of Red-black trees can be part of the data type, which ensures correctness (wrt. the invariants) of any code written using these data types. **?** presented an implementation in Agda, using the full power of a dependent type system to implement all invariants as part of the data type, and implemented some simple operations and insertion using that data type.

This project thesis formalizes an algorithm for deletion in Left-leaning Red-Black trees in Agda. We revisit and modify the data structure by **?**, that provides correctness by definition for algorithms working on LLRBs. The so established internal logic ensures that our deletion algorithm does not violate any invariants pertaining to Left-leaning Red-Black trees.

2 Left-Leaning Red-Black Trees

Left-Leaning Red-Black trees were introduced by **?** as Red-Black trees with an additional invariant: Only left children of black nodes can have a red color, all right children are required to be black. Hence, they do not only remove the possibility of 4-nodes in a tree (represented by a black node with two red children), but also the ambiguous representation of 3-nodes by a black node with either a left or a right red child. This greatly reduces the number of possible cases, and we will make use of that in an algorithm that heavily relies on pattern matching of tree segments.

For reference, Left-Leaning Red-Black trees can be summarized as binary trees with the following invariants:

1. The *Left-Leaning Red-Black invariant*, which specifies that every left child node of a black node in the tree can be red or black, and that all other nodes, including the root, must be black. Leafs (which contain no data) are always black.

2. The *Black Height invariant*, which assigns a black height to each node as the number of black nodes on every downward path from that node to a descendant leaf, and consequently requires that the black heights of both children of every node are equal. Leafs are defined to have a black height of zero.

3. The *Search Tree invariant*, which allows only nodes with smaller keys left and only nodes with greater keys right of any node in the tree, thus keeping the tree ordered.

We will show in detail how each of these invariants is kept.

3 Imports

The following code, hereby presented in Literate Agda, uses the following imports:

```
open import Data.Bool using (Bool; true; false; if_then_else_)
open import Data.List
open import Data.Nat hiding (_⩽_; _<_; _≟_; compare)
open import Data.Product
open import Data.Unit hiding (_≟_)

open import Level

open import Relation.Binary hiding (_⇒_)
open import Relation.Nullary
```

4 The data type for LLRBs

The data type used for representing Left-leaning Red-Black trees uses correctness by definition as a technique to embed all invariants that make up a valid Left-leaning Red-Black tree in the data type and its constructors itself. Thus, every function that creates or transforms LLRB trees using this data type will only type check if every invariant is proven as part of its definition.

While we present the tree data type in full detail, we will also look at the various differences to the original data type as specified by **?**.

First, let us declare the module in which the data type and all code is contained.

```
module llrb-delete-julien-oster
  (order : StrictTotalOrder Level.zero Level.zero Level.zero) where
```

To understand the module's parameterization, let us first look at what we would need to allow the tree's key type to be fully specified by a user of the module, and not fixed to e.g. an integer type:

- a type (the key type),
- an order and an equivalence relation onto the key type,
- a function for comparing values,
- another function for applying the transitivity of the order relation.

This allows any type to fill out the role as a key, as long as we can also provide everything else listed above. To achieve this, we could make every item an explicit parameter to the module (as done by **?**).

However, it is very often the case that a satisfying data type brings its common order relation with it, together with a function to compare and means to apply transitivity. For example, on the type of natural numbers this would be its common strict total order relation $<$, which orders natural numbers from smaller to greater.

We therefore choose to use a single `StrictTotalOrder` (as defined in the Standard Library) as a parameter instead. This particular record type fits well because it contains all of the above as fields of itself. The added benefit is that the module now more neatly fits into the Standard Library.

By subsequently opening the record, all relevant fields (such as `_<_`, `_`$\overset{?}{=}$`_`, `trans`, `compare`) are available to us in the namespace of our module, without having to specify the name `order`:

```
open module sto = StrictTotalOrder order
A = StrictTotalOrder.Carrier order
```

The second line allows us to also refer to the `StrictTotalOrder`'s carrier, which is the type used to store the node's keys, simply as `A` in our namespace.

Next, we define what bounds are:

```
LowerBounds = List A
UpperBounds = List A
```

Bounds are just simple lists of values (of our key type, `A`). A value being within those bounds means that it is greater (`LowerBounds`) or smaller (`UpperBounds`) than all elements in the respective list. However, this simple list type doesn't reflect that yet. To achieve this, we want to get types for proofs out of these bounds, whose inhabitants we can later require as part of our nodes.

Requiring the proofs is what keeps our trees consistent with the search tree invariant: The tree type will be parameterized with each of both bounds lists. Then, any node constructor which contains a key (i.e. any constructor except for the leaf constructor) must contain proofs that its key is within `LowerBounds` and `UpperBounds`. Moreover, every non-leaf constructor which creates a node also specifies a left and a right subtree as children, but the bounds list in the type of those children (`UpperBounds` for the left child, `LowerBounds` for the right child) is the parent's bounds list, prepended with the key of this node. Conclusively, every subtree must now not only satisfy the original bounds of its parent node, but additionally all keys within it must now also be smaller (if it is a left subtree) or greater (if it is a right subtree) than its parent's key!

The *types* of these proofs are gained with the following operators:

```
infix 5 _isleftof_
_isleftof_ : A → UpperBounds → Set
z isleftof []     = ⊤
z isleftof b :: β  = z < b × z isleftof β

infix 5 _isrightof_
_isrightof_ : A → LowerBounds → Set
z isrightof []     = ⊤
z isrightof b :: γ  = b < z × z isrightof γ
```

Looking just at `UpperBounds`, proving that a value is within particular bounds means supplying a proof that the value is smaller than the head of its bounds list and, inductively, that it is within the rest of the bounds list. A value is always within empty bounds (thus the type is the trivially provable $\top$ in that case).

The same applies to `LowerBounds`, except that the list's head and the given value are now reversed in the relation (thus, the value must be greater).

A key difference between the bounds definition by **?** and this one is that there are now two bounds lists instead of just one. (**?**) defined the single bounds type to contain either constraints in any order, by not keeping simple lists of values, but having different constructors for bound values that a value has to be "left" or "right" of. Subsequently, the tree type was only parameterized with the one bounds list and each inner node constructor only needed to specifies one proof instead of two, the proof that its key value is within the given bounds.

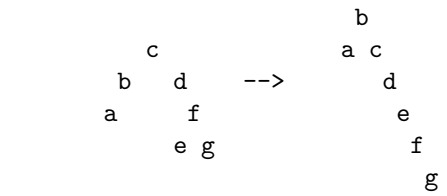
The reason why we choose to abandon this approach in favor of two explicit lists of bounds will get clear after we look at the reasons for handling transformations of them.

For that, we need data structures that represent proofs of how one list of bounds may imply another. Before we further elaborate on this subject, let us state that the interpretations of these data structures are the following: *Given proof that* $\beta \Rightarrow^l \beta'$ *and that* x isleftof β*, we conclude that* x isleftof β'*.* and *given proof that* $\beta \Rightarrow {}^r\beta'$ *and that* x isrightof β*, we conclude that* x isrightof β'*.*

```
infix 5 _⇒ʳ_
data _⇒ʳ_ : UpperBounds → UpperBounds → Set where
  ▪      : ∀ {γ} → γ ⇒ʳ γ
  keep_  : ∀ {γ γ' b} → γ ⇒ʳ γ' → b :: γ ⇒ʳ b :: γ'
  skip_  : ∀ {γ γ' b} → γ ⇒ʳ γ' → b :: γ ⇒ʳ γ'
  cover_,_  : ∀ {β β' x y} → x < y → x :: y :: β ⇒ʳ β'
         → x :: β ⇒ʳ β'

infix 5 _⇒ˡ_
data _⇒ˡ_ : LowerBounds → LowerBounds → Set where
  ▪      : ∀ {β} → β ⇒ˡ β
  keep_  : ∀ {β β' b} → β ⇒ˡ β' → b :: β ⇒ˡ b :: β'
  skip_  : ∀ {β β' b} → β ⇒ˡ β' → b :: β ⇒ˡ β'
  cover_,_  : ∀ {γ γ' x y} → x < y → y :: x :: γ ⇒ˡ γ'
         → y :: γ ⇒ˡ γ'
```

Transforming bounds into other (implied, non-contradictory) bounds is needed, because when reconfiguring a tree into another, the bounds of individual nodes may change, while the overall relation between them stays intact (otherwise, we would break the search invariant).

Let's look at a valid transformation of a simple tree:



It is clear that while the shapes of the trees are vastly different, they still retain the same order relations between their elements and thus the search tree invariant has been kept. They are both valid representations of the same data.

However, let's look at the bounds list of node `e`'s tree type. We hereby assume that the bounds for the root element of the left tree, `c`, are contained in the unknown lists β and γ for min and max bounds, respectively. In the left tree, the bounds lists for `e` look like that:

`lower: c :: d ::` β
`upper: f ::` γ

With every level down, the parent's key is prepended to the respective bounds list (depending on whether the child is left or right). Thus, according to the resulting lists, `e` must be smaller than `f`, but greather than `c` and `d`. `a`, `b` and `g` are not appearing because they are not on the path between the root and `e`, and thus never handed down. `e` must also be within β and γ, the bounds for the whole tree.

On the right tree, the bounds lists are:

`lower: b :: c :: d ::` β
`upper:` γ

Node `f`, formerly a bound, is now a child of `e`, and thus fell out of its bounds (the roles have been reversed: `f` must now satisfy `e`'s bounds, reflected in `f`'s type instead). In fact, along the path from the root to `e`, `e` is now the node with the largest key, and the upper bounds list has simply become γ, which must still be satisfied knowntheless.

On the other hand, due to the rotation of `b` into the path as the new root node, `b` now appears in the min bounds list, where it formerly didn't at all.

It is clear that we cannot just take the old bounds as parts of the new tree.

To allow operations like insertion or deletion of elements (which may change the shape of subtrees, or even the whole tree) without pulling apart and reconstructing everything, we need a way to show that the original relations (manifested by the original bounds) are suitable, or *imply*, the relations/bounds in our new shape.

The rules defined above exist for that purpose. They are a *series of rules* that are stringed together to form a proof that an original list of bounds satisfies a new one, which is manifested by the node's type at its place in the new tree (recall that the parent node essentially dictates the type of its children).

For ease of explanation however, we discard the notion that the aforementioned data type represents proofs of the old bounds implying the new bounds, and regard them as operations to transform the original list into the new one instead. We describe a constructor as an operation on a *current* bound, which is initially the head of the list and may be advanced to the next one by the operation itself. This next bound may be the original next one or a new one, inserted by a previous step.

The operations are as follows:

keep Keep the current bound and move on to the next.

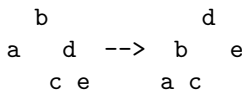
skip Skip the current bound, because it is not needed anymore (possibly because its node has been moved out of the path, or completely removed). The current bound is now the formerly following bound.

cover Insert a new bound after the current bound, because a new node appeared along the path to the root (similarly to above, it may have been moved in or is a new node). This is allowed in the case that the current bound is already a *better* bound than the new one: Let a be any lower bound for x. If $b < a$, then $b < x$ due to transitivity, and b is also a lower bound for x. Analog with upper bounds. This operation requires us to specify a proof of $b < a$. The current bound is still the former, better bound, so that it may be involved in another step.

▪ Keep the rest of the bounds, starting from the current one, as it is. (Actually, this constructor is read as "q.e.d." because it marks the end of the proof.)

Going back to the example above, we can transform the upper bounds with `skip ▪` and the lower bounds with `cover b<c , keep keep ▪`, or simply `cover b<c , ▪`, and we know that the required proof `b<c` is actually extractable from former tree, from the simple fact that `b` is a left child of `c`.

Another example lets us now illustrate the decision for representing the bounds with two lists instead of one. Given the following tree transformation:



Despite the transformation, the bounds for node `c` in both trees are:

`lower: b ::` β
`upper: d ::` γ

Whereas the representation used by **?**, which effectively denotes the path back to the root, would yield to the differing lists `leftOf d :: rightOf b ::` β for the left tree and `rightOf b :: leftOf d ::` β for the right tree.

This necessitated the need for another rule, the `swap` rule, which was used to prove that the position of bounds inside the list was irrelevant by allowing to swap the position of two consecutive bounds. However, in more complex transformations that change the shape of a tree significantly, parent nodes can change their position by more than one tree level (while still staying in a node's path to the root), resulting in a lot of applications of `swap`.

Moreover, in the case that only lower or only upper bounds changed, superfluous `keep` applications were necessary to disregard bounds of the other kind.

With separate bounds, the invariant ordering of the lists renders `swap` futile, and we may look at only one kind of bounds and keep the other ones as they are, a situation which frequently arises.

The futility also results in the proof system's linearity, which greatly increases the chances for Agda's term searching solver, Agsy, to find the proofs.

Finally, we define the intepretation of our bounds implication proof data structure, which changes the proof type obtained by `_isleftof_` or `_isrightof_` according to the new bounds lists:

```
[[_]]ʳ : ∀ {β β'} → β ⇒ʳ β' → (x : A) → x isleftof β → x isleftof β'
[[ ▪          ]]ʳ z p              = p
[[ keep h     ]]ʳ z (p₁ , p₂)      = p₁ , [[ h ]]ʳ z p₂
[[ skip h     ]]ʳ z (_  , p)       = [[ h ]]ʳ z p
[[ cover  q , h ]]ʳ z (p₁ , p)       = [[ h ]]ʳ z (p₁ , trans p₁ q , p)
```

```
[[_]]ˡ : ∀ {γ γ'} → γ ⇒ˡ γ' → (x : A) → x isrightof γ → x isrightof γ'
[[ ▪          ]]ˡ z p              = p
[[ keep h     ]]ˡ z (p₁ , p₂)      = p₁ , [[ h ]]ˡ z p₂
[[ skip h     ]]ˡ z (_  , p)       = [[ h ]]ˡ z p
[[ cover  q , h ]]ˡ z (p₁ , p)       = [[ h ]]ˡ z (p₁ , trans q p₁ , p)
```

Given a value `x` and a proof that the old bounds imply the new ones (or, w.r.t. the terminology above, operations to transform the list) and a proof that `x` lies within the old bounds, we obtain a proof that `x` lies within the new bounds.

As with any function in Agda, we can read the types for the functions above as proposition. The first function for example can be read as, roughly: *Given proof that* $\beta \Rightarrow^l \beta'$ *and that* x isleftof β*, we conclude that* x isleftof β'–and this is exactly what we stated as the interpretation for our proof data structure above!

The final step towards our tree data structure is the simple definition of the data type for a tree node's color:

```
data Color : Set where
  red   : Color
  black : Color
```

So that now have everything together for the tree data type itself. The head is thus:

```
data Tree' (β : UpperBounds) (γ : LowerBounds) : Color → ℕ → Set where
```

The tree is parameterized by the two respective bounds lists, which are directly the bounds for its root node and by extension part of all descending node's bounds.

Furthermore, we index each node by its color and its black height. These two values on which a tree node depends on are used to maintain the Left-leaning Red-Black invariant and the black height invariant of an LLRB tree.

In the data type's body, we start with the definition of a leaf node, as it is simplest:

```
  lf : Tree' β γ black 0
```

The leaf node constructor has no arguments, as it has no children, its black height is always 0 and (because it contains no key and is always trivially within bounds) needs no proof to satisfy the search three invariant. Its color is, by definition, always black.

A red node is more interesting:

```
nr : ∀ {n}(a : A) → a isleftof β → a isrightof γ
   → Tree' (a :: β) γ black n → Tree' β (a :: γ) black n
   → Tree' β γ red n
```

It contains the following data:

- The black height `n`, which is implicit to the constructor and can be inferred by the children's black height. A red node doesn't add to a tree's black height, so a red node's black height is equal to that of its children (which must both have the same black height `n`).
- A key value `a` of type `A`.
- A proof that the key value is within upper bounds (remember that `a isleftof` β gives us the type for a proof that `a` is within the upper bounds given by β), as well as a proof that it is within the lower bounds (γ).
- A left child, whose upper bounds are prepended by this node's key value. Consequently, all nodes within this node's left child tree must not only satisfy this node's original bounds, but also be smaller than the this node's key, `a`. Note that the left bounds remain untouched. Because a red node is only allowed to contain black nodes, the left child is forced to be a black node.
- A right child, for which the same applies, except that `a` is prepended to the lower bounds instead. It must also be a black node, though not only because its parent is red, but because in the LLRB tree variant, right children in general must be black.

Black (non-leaf) nodes are similar:

```
nb : ∀ {leftSonColor n}(a : A) → a isleftof β → a isrightof γ
   → Tree' (a :: β) γ leftSonColor n → Tree' β (a :: γ) black n
   → Tree' β γ black (suc n)
```

Because the constructor is quite similar to those of red nodes, we will only look at the differences.

First, while we still implicitly gather the black height from its children (which must still be equal among them), the resulting black height for a black node seen as a tree

is now `suc n`, that is, the children's black height plus 1, resulting simply because the black node itself adds to it.

Second, while the right child's color is still constrained to be black, the left child may now be red or black. The implicit argument `leftSonColor` may be inferred by the left child's type. [1]

We now have a full data type for generic Left-leaning Red-Black trees that internally enforces all required invariants:

1. The Left-Leaning Red-Black invariant is enforced because the tree is indexed with its color, which is specified as black for all right child nodes in general and all left child nodes if they are children of a red node.

2. The Black Height invariant is maintained by having the tree type indexed with it, and specifying it as the same as both the children's black height in case of a red node or one more in case of a black node.

3. The Search Tree invariant must be kept satisfied by maintaining bounds along all tree nodes and the requirement to specify proofs that all respective keys are within those bounds.

Finally, we would like to have operators which take a subtree and transform its bounds by providing a proof that derives the new bounds, so that it may be used as part of a differently configured tree:

```
infixl 3 _<,_
_<,_ : ∀ {β β' γ c n} → Tree' β γ c n → β ⇒ʳ β' → Tree' β' γ c n
lf          <, φ = lf
nr x pxl pxr l r <, φ =
  nr x ([[ φ ]]ʳ x pxl) pxr (l <, keep φ) (r <, φ)
nb x pxl pxr l r <, φ =
  nb x ([[ φ ]]ʳ x pxl) pxr (l <, keep φ) (r <, φ)

infixl 3 _<,,_
_<,,_ : ∀ {β γ γ' c n} → Tree' β γ c n → γ ⇒ˡ γ' → Tree' β γ' c n
lf          <,, φ = lf
nr x pxl pxr l r <,, φ =
  nr x pxl ([[ φ ]]ˡ x pxr) (l <,, φ) (r <,, keep φ)
nb x pxl pxr l r <,, φ =
  nb x pxl ([[ φ ]]ˡ x pxr) (l <,, φ) (r <,, keep φ)
```

[1] As with any implicit argument, however, it may also be explicitly specified instead. In this case, this may faciliate pattern matching where we would later like to match against a black left child. By explicitly specifying the `leftSonColor`, we avoid the need for two similar patterns, one which matches a leaf child (`lf`) and one which matches a non-leaf child (`nb`).

Note that until now, we have been exclusively dealing with *subtrees*. Standalone trees must follow the same constraints, but with an additional invariant: the root node must always be black. To implement that invariant, we define a separate data type for standalone trees:

```
data Tree : Set where
  tree : ∀ {n} → Tree' [] [] black n → Tree
```

Because standalone trees are by definition never part of any other tree that could impose further bounds on it, the root's bounds lists are empty.

Exporting only this data type and all top level functions that use it effectively hides our implementation of the invariants from users of the module.

5 Deletion

There already is an algorithm for deletion available, which **?** presented alongside his introduction to Left-Leaning Red-Black trees. This algorithm, however, while being pretty concise, follows a very imperative approach, containing a lot of nested if-else-blocks and breaking several invariants during its run by design, later fixing everything up using several auxiliary functions.

Translating this algorithm into Agda code is possible, but doing so would require immediate data structures to hold various tree structures with broken invariants, along with tedious proofs for those data structures and auxiliary fix-up functions.

Instead, we follow an approach which is closer to what **?** did with his seminal, purely functional implementation of insertion in Red-Black trees. That is, our algorithm relies purely on pattern matching and simple recursive decent. Moreover, of the three functions it consists of, the two which implement the algorithm's bulk are completely self-contained, i.e. do not rely on auxiliary functions.

During recursion we match for certain *segments*, which are just tree portions encompassing a specific black height, and apply a recursive call on a smaller subtree which will be part of the remaining path. Depending on the result, we reconfigure the visited segment so as to incorporate the result without breaking any invariants.

The immediate downside of this approach is the sheer number of cases pertaining to differently shaped tree segments that have to be taken care of, making the algorithm seem long and complex at first. This, however, is merely superficial: The algorithm only follows a very simple pattern of recursive descent and Agda's coverage checking

ensures that none of the possible case have been forgotten, which in themselves aren't hard. Thanks to the linearity of the domain specific proof language for bound coercion, most of the proofs can be found by using Agda's term search solver, Agsy.

While we do not include any formal proof for one property of the algorithm, namely that it always does delete exactly the node with the given key in a tree that contains one (a proof as part of the Agda source code remains future work for now), this can easily be checked informally due to an additional property of the algorithm: The only time a node is really *removed* from a tree is when a base case has been reached, in every other case we follow a single tree path until we reach such a base case–possibly taking a matching node along as part of the reconfigurations that happen during descent.

Thus, checking that the node with the right key is removed means just checking if the simple base cases omit said node when building their result, and checking that no other node is affected means just checking whether all non-matching nodes are included in the result of every case.

(Also currently missing is a formal proof for the algorithm's termination. This would be possible with just minimal changes to the source by using *Sized Types* and is only due to a bug in Agda, present at the time this thesis was written. In the meantime, we just have to check that every recursive call happens on a *smaller* subtree.)

We will now take a detailed look at the algorithm itself, starting with its two helper functions, which implement the bulk of the algorithm.

6 Extracting the minimum node

Our first look is at `extractMinR`, a function that shall *extract* the minimum key from a red tree. It returns the minimum key and a modified tree, in which the node which carried this key has been deleted.[2]

This function shares its premise and a similar strategy with `deleteR`, which we will cover next. Both functions are needed as part of the overall algorithm, but because `extractMinR` contains much fewer and simpler cases than `deleteR`, we will discuss it first.

The premise shared with `deleteR` is that we will only handle deletion on *red* subtrees, that is, trees with a red root. The reason we have this constraint is that in such a tree, we can always remove a node *without affecting black height*.

[2] It will in fact return more terms which purely exist for type checking, but we will look at that in time.

This property is similar to one important detail when inserting nodes into a valid Red-Black tree: black height will only increase (by one) if after insertion and reconfiguration of the tree, the root node has become red and must thus be recolored black, which only happens at the very end of the algorithm. The recursive part of insertion doesn't affect black height at all.

We can mirror this for the removal of nodes: it is always possible to remove a node from a tree with a red root, without decreasing black height. There are a few base cases at black height 0 and 1, while the rest is a matter of always reconfiguring the tree in such a way that the recursive call of `extractMinR` on a smaller, red-rooted tree is possible.

Without this property, we would not only have to handle the different cases of a resulting tree with a red or black root[3], in other calling functions as well as in the recursive call of `extractMinR` itself, but also distinguish on whether the black height changed or remained the same, to avoid violating the black height invariant of Red-Black trees.

Thus, the overall strategy of `extractMinR` looks like this:

1. Pattern match against the top of the currently looked at subtree.
2. If we have reached the minimum (i.e., a base case), return a subtree without that node.
3. Otherwise, perform a recursive call with a smaller red subtree.
4. Finally, rebuild a valid tree from the result and all nodes that weren't part of the recursive call and return it.

Note that it may not be possible to just take an existing subtree in step 3. Because we need a red subtree, we may have to build a differently shaped one with existing nodes. Moreover, the outcome of step 4 may cause some reconfiguration within the parent step of recursion to incorporate the returned tree, possibly also distinguishing between a red or black result.

Let us now specify `extractMinR`'s type. Quick reasoning may suggest the following rather straightforward type at first:

```
extractMinR : ∀ {n β γ} → Tree' β γ red n
              → ∃ λ c → A × Tree' β γ c n
```

[3]Please note that the result of `extractMinR` is *not* always a red tree. This would inductively mean that a given black height would never decrease until no nodes are left, which is clearly nonsense.

This function would take a red tree and return the minimum as a value of key type `A`, as well as a new tree, supposedly without the minimum, which lies within the same bounds as the original one and which may now be either red or black.

While this is indeed what we want from an operational point of view, it throws away some important information regarding the relationship between minimum and resulting tree, and which we will actually need when type checking the overall algorithm. Namely, if the tree itself was within the bounds given by β and γ, then the minimum key value itself will be, too! Additionally, the new tree as a whole is now not only standing right of its original bounds, but of course also right of its former minimum.

We will maintain proofs for these propositions during our definition, and return them alongside the operational results of our function. The full type is thus:

```
extractMinR : ∀ {n β γ} → Tree' β γ red n
              → ∃₂ λ min c →
                   min isleftof β ×
                   min isrightof γ ×
                   Tree' β (min :: γ) c n
```

We see the additional inclusion of proofs of type `min` isleftof β and `min` isrightof γ, and that the lower bounds list of the returned tree has been prepended with `min`, the minimum value.

Let us look at the base cases. For black height 0, the only possible red tree is a red node with no children, which is the minimum and thus removed.

```
extractMinR (nr a pal par lf lf) = a , black , pal , par , lf
```

The removed element is returned as a minimum, and the proofs don't need to be manipulated.

The remaining two base cases are for black height 1. We handle those two cases directly:

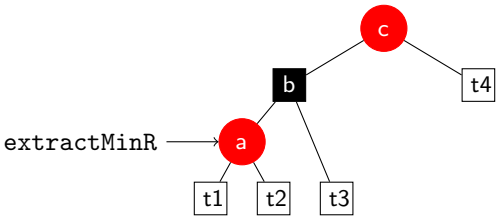
```
extractMinR (nr b pbl pbr
                  (nb a (a<b , pal) par lf lf)
                  (nb d pdl (b<d , pdr) lf lf)) =
   a , black , pal , par , nb d pdl (trans a<b b<d , pdr)
                                    (nr b (b<d , pbl) (a<b , pbr) lf lf) lf
```


```
extractMinR (nr b pbl pbr
                  (nb a (a<b , pal) par lf lf)
                  (nb d pdl (b<d , pdr)
                        (nr c (c<d , pcl) (b<c , pcr) lf lf) lf)) =
   a , red , pal , par , nr c pcl (trans a<b b<c , pcr)
                              (nb b (b<c , pbl) (a<b , pbr) lf lf)
                              (nb d pdl (c<d , trans a<b b<d , pdr) lf lf)
```

Notice how we have to use transitivity to arrive at the new proofs, which are slightly modified due to the tree's new shape, and due to the returned tree's modified type that includes the minimum in its bounds (in both cases `a`). Also note that since we built completely new trees and use no pre-existing subtrees anywhere, we don't transform any bounds yet (we effectively discard them, and the new nodes have bounds lists dictated by their parents).

We have now covered two of three cases for black height 1: no grandchildren and one red grandchild on the right. We could handle the remaining third case, which has a red grandchild on the left black node, but we omit it because it overlaps with the first recursive case, which we will take an in-depth look at.

```
extractMinR (nr c pcl pcr
                  (nb b pb pbr (nr a pal par t1 t2) t3) t4)
  with extractMinR (nr a pal par t1 t2)
... | x , c' , (x<b , x<c , pxl) , pxr , a' = x , red , pxl , pxr ,
  nr c pcl (x<c , pcr)
       (nb b pb (x<b , pbr)
             a'
             (t3 <,, cover x<b , ▪))
       (t4 <,, cover x<c , ▪)
```



If we reach this case, then the minimum must lay somewhere in the subtree `a` (which in its simplest case may be a single red node, covered in our first case, hence the omission of this as a base case).

In this case, we recursively call `extractMinR` on `a`. Because it is the left child of its black parent `b` (which allows its left child to be red or black) and according to `extractMinR`'s type the new tree `a'` has the same black height as before, we can plug in the result without any reconfiguration of the tree's shape.

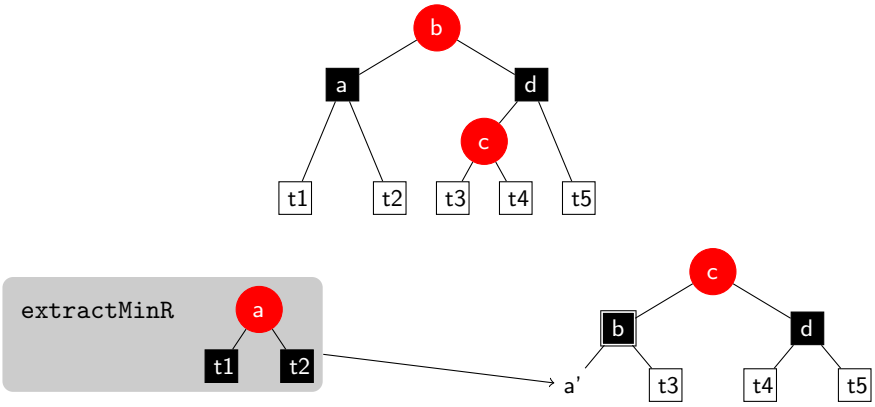
Yet we modify the proofs by prepending proof components of the form $x < \circ$, where x is the minimum returned. In turn we also have to transform the bounds of `t3` and `t4`, by prepending `x` on the lower bounds to them, to accomodate this change dictated by their parents `c` and `d`. Why, if we still deal with the same shape?

The answer lies in the return type of the tree as specified by `extractMinR`. Recall that we don't return a tree corresponding to the old bounds, but a tree of type `Tree' β (min :: γ) c n`, which means that we prepend the new minimum to its lower bounds. While the returned tree `a'`, coming from the function, already corresponds to that type, the untouched trees `t3` and `t4` do not. We need to prove that they can accomodate the new bounds contained in that type, which is done with a simple `cover`, because the bound specifying that a node has to be left of the minimum is not a better bound than any of the existing tighter bounds.

For the base cases handled before, this was more subtle, because the minimum is contained within the matched nodes and not returned by a recursive call.

```
extractMinR (nr b pbl pbr
                  (nb a (a<b , pal) par (nb x1 px1l px1r t1l t1r) t2)
                  (nb d pdl (b<d , pdr)
                    (nr c (c<d , pcl) (b<c , pcr) t3 t4)
                    t5))
  with extractMinR (nr a (a<b , pal) par (nb x1 px1l px1r t1l t1r) t2  <, cover b<c , ▪)
... | x , c' , (x<b , x<c , pxl) , pxr , a' = x , red , pxl , pxr ,
  nr c pcl (trans x<b b<c , pcr)
       (nb b (b<c , pbl) (x<b , pbr)
```

```
      a'
      (t3 <, keep skip ▪ <,, cover x<b , ▪))
(nb d pdl (c<d , trans x<b b<d , pdr)
      (t4 <,, keep cover x<b , skip ▪)
      (t5 <,, cover c<d , keep keep cover x<b , skip ▪))
```



Starting with this case, the nodes we are matching against aren't whole subtrees, but merely *segments*, which have other subtrees attached to their end nodes. End nodes are those nodes at the segment's bottom which are matched by variable (i.e. wildcard) and not by any node constructor. While the attached subtree's black height may be arbitrary, we only construct segments in shapes that accomodate subtrees of equal black height to each other. We can therefore say that a segment *encompasses* a certain black height, and the black height of all attached subtrees is the black height of the whole visited subtree (including the segment on top) minus the black height encompassed by the segment.

For this function, visiting segments which encompass black height 1 suffices.

In this case, as always, the minimum will be part of the subtree under the root `a` (possibly a itself if both of its children are leaf nodes), so we must continue there. However, in this case, `a` is black, so we cannot directly call `extractMinR` on it. Instead, we reconfigure the whole tree to a different shape in which `a` is now a red node.

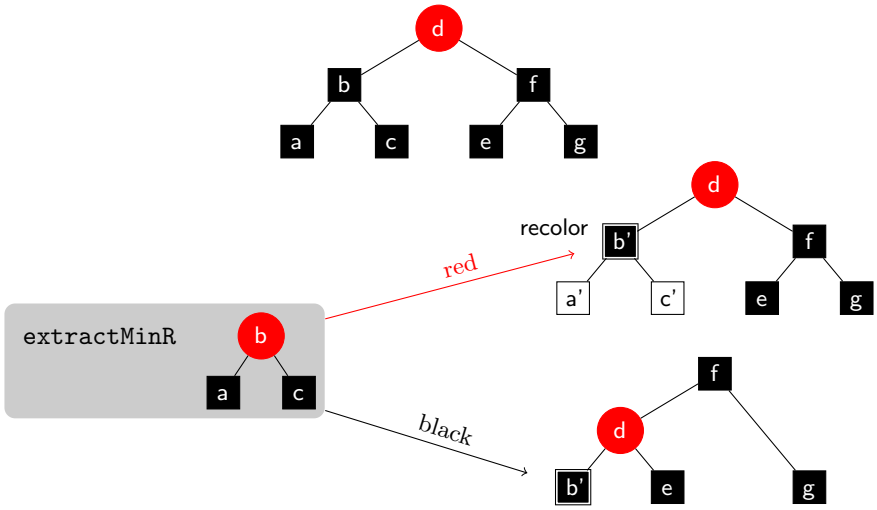
In such a reconfiguration, we must not only take care that the new shape still satisfies all invariants (paying attention to black heights in particular), but also that the subtree on which `extractMinR` is recursively applied remains a smaller subtree. Otherwise, the algorithm might not terminate.

To achieve the reconfiguration, we apply `extractMinR` on a new red subtree with root node `a` and *then* plug the result into the rest of the whole differently-shaped tree.

In addition to prepending the minimum to the tree bounds, some given subtrees' bounds need to be transformed to reflect their new position in the result tree.

For example, `t3`'s upper bounds previously stated that it was left of `c` and left of `d`, whereas in the new shape it is now only left of `c`. A `keep skip` on the upper bound fixes this. Similarly, `t5` was found right of `b` (and right of `d`, which remains true), but its new position puts it right of `c` instead. A `cover c<d` introduces the new lower bound `c` and a `skip` at the end discards the now defunct lower bound `b`.

Another subtree `t4` and even the subtree that gets passed to `extractMinR` are affected. Note that for the latter one, we could also have applied the transformation to the result instead.




```
extractMinR (nr d pdl pdr
                  (nb b (b<d , pbl) pbr
                        (nb a pal par al ar)
                        c)
                  (nb f pfl (d<f , pfr)
                        (nb e (e<f , pel) (d<e , per) el er)
                        g))
  with extractMinR (nr b (b<d , pbl) pbr (nb a pal par al ar) c)
... | x , red , (x<d , pxl) , pxr , nr b' pbl' pbr' a' c' =
  x , red , pxl , pxr ,
  nr d pdl (x<d , pdr)
       (nb b' pbl' pbr' a' c')
       (nb f pfl (d<f , pfr)
```

```
            (nb e (e<f , pel) (d<e , per) el er)
            g <,, cover x<d , ▪)
... | x , black , (x<d , pxl) , pxr , b' =
  x , black , pxl , pxr ,
  nb f pfl (trans x<d d<f , pfr)
       (nr d (d<f , pdl) (x<d , pdr)
             (b' <, cover d<f , ▪)
             (nb e (e<f , pel) (d<e , per) el er <,, cover x<d , ▪))
       (g <,, keep cover x<d , skip ▪)
```

We follow a different approach in this case. Once again, the subtree in which the minimum lies, `a`, is black instead of red. We can, however, not recolor it red, because we cannot make the assumption that its left child is black as well, which would be a precondition for turning `a` red.

But what we *do* know is that both `a` and `c` are black (the latter following directly from the invariant that right children are always black). Therefore, the recoloration of its parent `b` is possible, and we can apply `extractMinR` on the recolored subtree with root `b` because it is still a smaller tree than the overall tree.

Note, however, that this recoloration changes the black height of `b`: by turning it from a black node into a red node, we decremented its total black height by one and the result of `extractMinR`, whose black height is the same as for its input, as well!

We therefore need to take care of a result with a decremented black height. In the case that `extractMinR` returned a red tree, this is easy: We can always recolor a red node to a black node. This reincrements the black height to its former value and we can plug the subtree into its former place.

A black result is slightly less straightforward to handle. There is no immediate way to increase its black height[4], so we have to work with a result of differing black height. But because its black height is only off by one, a simple left rotation of the whole tree allows the incorporation of the result.

7 Deletion in red trees

The next part of the algorithm covers deletion of arbitrary nodes in red trees, another part of the overall deletion algorithm (which, from a purely external point of view,

[4] There would, by the way, also not be any immediate way to decrease its black height. Turning it red is not possible in general because we cannot make the assumption that a black root has two black children.

differs to this part only in the fact that it deletes nodes in black trees, i.e. proper LLRBs). The desired node will be selected by its key value. Because we are once again removing a single element from a subtree with a red root, everything that was said in the previous section with regard to maintaining tree properties applies. Specifically, we can remove the node without changing the black height of the tree, so that we do not have to handle black height differences of the result.

We call the function `deleteR`. As we want to pass it a key and a red tree in which to remove the matching node, returning the modified tree as a result, the type is easily specified as:

```
deleteR : ∀ {n β γ} → A → Tree' β γ red n → ∃ λ c' → Tree' β γ c' n
```

Once again the black height stays invariant (bound as `n` in the type), and the function may return a red or black tree as well.

A quick note about multiple nodes with the same key: Because we set bounds up as values which are on one side of the strict order relation $<$ as opposed to a non-strict relation like $\leqslant$, our tree type cannot accomodate more than one node with a given key value. Any existing key in a tree imposes its value as bounds to its node's children, i.e. all left descendants must be strictly smaller and all right descendants strictly greater. As one of the equal nodes would be a descendant of the other, it follows directly that no two nodes in a tree can contain the same key without violating the thusly maintained Search Tree Invariant. This, however, would be made possible by just using the corresponding non-strict relation for the tree's definition instead, by allowing nodes to have direct children with equal keys, and would render the tree an implementation of a multiset instead of a set.

The main difference between the algorithms of `extractMinR` and `deleteR` is that while the former allows us to just recurse down into the leftmost subtree until a base case is reached, the descending path of the latter is not as fixed, but rather determined by comparisons done within each recursive step.

For trees with black height 0 or 1, we can just select the target node by performing a few comparisons and return a tree with that node removed. We will handle such trees as the base cases for our recursion.

As already mentioned, there is only one red tree with black height 0, which contains one node. Handling it is just a matter of deciding whether to remove it:

```
deleteR .{0} x (nr a pal par lf lf) with x ≟ a
```

We remove the node if its key is equal to the given key `a`:

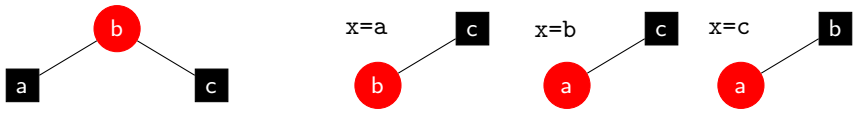
```
... | yes _ = , lf
```

And leave it if it's not:

```
... | no  _ = , nr a pal par lf lf
```

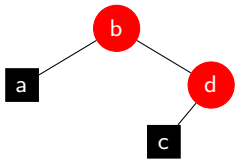
However, as we traverse the tree exactly once until a base case is reached, failure to find the desired node in such a base case means that the tree did not contain a node with the given key at all. We could easily return the information whether an actual deletion occurred together with the tree[5], but since it is not strictly necessary we omitted it in this code for clarity of type and definition.

At black height 1, more comparisons are necessary to single out the desired node:

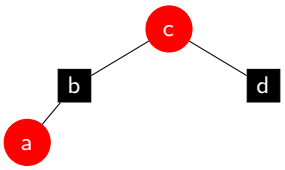



```
deleteR .{1} x (nr b pbl pbr
                    (nb a (a<b , pal) par lf lf)
                    (nb c pcl (b<c , pcr) lf lf)) with x ≟ a
... | yes _ = , nb c pcl pcr (nr b (b<c , pbl) pbr lf lf) lf
... | no  _ with x ≟ b
... | yes _ = , nb c pcl pcr (nr a (trans a<b b<c , pal) par lf lf) lf
... | no  _ with x ≟ c
... | yes _ = , nb b pbl pbr (nr a (a<b , pal) par lf lf) lf
... | no  _ = , nr b pbl pbr (nb a (a<b , pal) par lf lf)
                             (nb c pcl (b<c , pcr) lf lf)
```

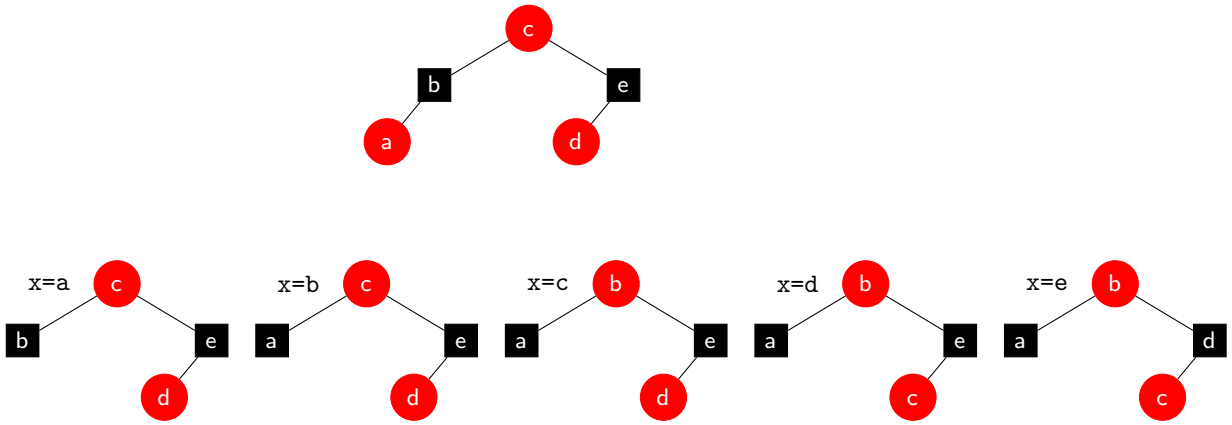
[5] Even if no deletion occurred, the tree is *not* necessarily untouched. While the tree remains equal in the sense that it represents the same set as before, it may (and in most random trees will) vastly change its shape due to the reconfigurations which happen while simply traversing it. To avoid this, one may search the tree for an appropriate node before attempting a deletion. While this just introduces a constant factor into the time complexity that inflicts no penalty on its asymptotical bound, the implications for an amortized analysis may differ due to the varying balance factor of a reshaped tree.




```
-- 1.5
deleteR .{1} x (nr b pbl pbr
                  (nb a (a<b , pal) par lf lf)
                  (nb d pdl (b<d , pdr)
                            (nr c (c<d , pcl) (b<c , pcr) lf lf) lf)) with x ≟ a
... | yes _ = , nr c pcl pcr
                  (nb b (b<c , pbl) pbr lf lf)
                  (nb d pdl (c<d , pdr) lf lf)
... | no  _ with x ≟ b
... | yes _ = , nr c pcl pcr
                  (nb a (trans a<b b<c , pal) par lf lf)
                  (nb d pdl (c<d , pdr) lf lf)
... | no  _ with x ≟ c
... | yes _ = , nr b pbl pbr
                  (nb a (a<b , pal) par lf lf)
                  (nb d pdl (trans b<c c<d , pdr) lf lf)
... | no  _ with x ≟ d
... | yes _ = , nr b pbl pbr
                  (nb a (a<b , pal) par lf lf)
                  (nb c pcl (b<c , pcr) lf lf)
... | no  _ = , nr b pbl pbr
                  (nb a (a<b , pal) par lf lf)
                  (nb d pdl (b<d , pdr) (nr c (c<d , pcl) (b<c , pcr) lf lf) lf)
```




```
-- 1.6
deleteR .{1} x (nr c pcl pcr
                  (nb b (b<c , pbl) pbr
                        (nr a (a<b , a<c , pal) par lf lf) lf)
                        (nb d pdl (c<d , pdr) lf lf)) with  x ≟ a
... | yes _ = , nr c pcl pcr
                  (nb b (b<c , pbl) pbr lf lf)
                  (nb d pdl (c<d , pdr) lf lf)
... | no  _ with x ≟ b
... | yes _ = , nr c pcl pcr
                  (nb a (trans a<b b<c , pal) par lf lf)
                  (nb d pdl (c<d , pdr) lf lf)
... | no  _ with x ≟ c
... | yes _ = , nr b pbl pbr
                  (nb a (a<b , pal) par lf lf)
                  (nb d pdl (trans b<c c<d , pdr) lf lf)
... | no  _ with x ≟ d
... | yes _ = , nr b pbl pbr
                  (nb a (a<b , pal) par lf lf)
                  (nb c pcl (b<c , pcr) lf lf)
... | no  _ = , nr c pcl pcr
                  (nb b (b<c , pbl) pbr
                        (nr a (a<b , a<c , pal) par lf lf) lf)
                  (nb d pdl (c<d , pdr) lf lf)
```



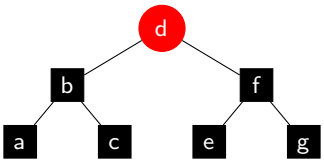

```
deleteR .{1} x (nr c pcl pcr
                    (nb b (b<c , pbl) pbr
                          (nr a (a<b , a<c , pal) par lf lf) lf)
                    (nb e pel (c<e , per)
                          (nr d (d<e , pdl) (c<d , pdr) lf lf) lf)) with  x ≟ a
... | yes _ = , nr c pcl pcr
                    (nb b (b<c , pbl) pbr lf lf)
                    (nb e pel (c<e , per)
                          (nr d (d<e , pdl) (c<d , pdr) lf lf) lf)
... | no  _ with x ≟ b
... | yes _ = , nr c pcl pcr
                    (nb a (trans a<b b<c , pal) par lf lf)
                    (nb e pel (c<e , per)
                          (nr d (d<e , pdl) (c<d , pdr) lf lf) lf)
... | no  _ with x ≟ c
... | yes _ = , nr b pbl pbr
                    (nb a (a<b , pal) par lf lf)
                    (nb e pel (trans b<c c<e , per)
                          (nr d (d<e , pdl) (trans b<c c<d , pdr) lf lf) lf)
... | no  _ with x ≟ d
... | yes _ = , nr b pbl pbr
                    (nb a (a<b , pal) par lf lf)
                    (nb e pel (trans b<c c<e , per)
                          (nr c (c<e , pcl) (b<c , pcr) lf lf) lf)
... | no  _ with x ≟ e
... | yes _ = , nr b pbl pbr
                    (nb a (a<b , pal) par lf lf)
                    (nb d pdl (trans b<c c<d , pdr)
```

```
                          (nr c (c<d , pcl) (b<c , pcr) lf lf) lf)
... | no  _ = , (nr c pcl pcr
                    (nb b (b<c , pbl) pbr
                          (nr a (a<b , a<c , pal) par lf lf) lf)
                    (nb e pel (c<e , per)
                          (nr d (d<e , pdl) (c<d , pdr) lf lf) lf))
```

For trees with black height 2 or up, we will visit tree *segments* of black height 2 and make comparisons to decide which path to take down in our recursion. As long as we always recurse into a smaller subtree, the algorithm will eventually reach a base case and terminate. But because we can only call `deleteR` on trees with red nodes, we may have to reconfigure the visited tree portion accordingly.

Because we only look at trees with a red root, both of the root's children are always black. The difference may lie in the root's left-left and right-left grandchild, both of which maybe red or black. This gives a total of four cases for black height 2.

Let's look at the simplest case of a red tree segment with black height 2, whose only red node is the root itself:

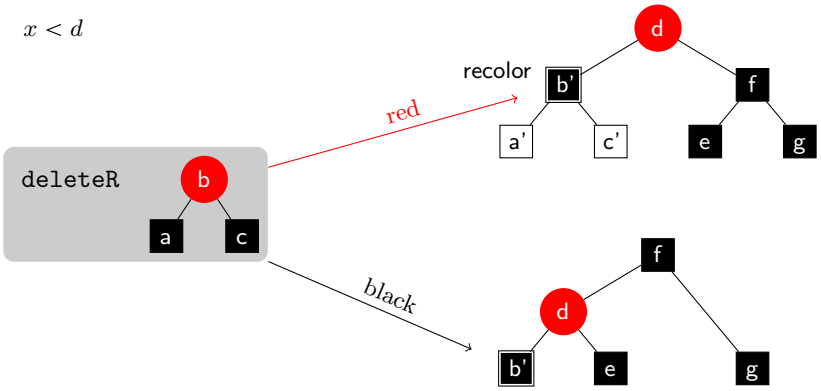



```
-- 2.4
deleteR x (nr d pdl pdr
                  (nb b pbl pbr
                      (nb a pal par al ar) (nb c pcl pcr cl cr))
                  (nb f pfl pfr
                      (nb e pef per el er) (nb g pgl pgr gl gr))) with compare x d
```

As mentioned, we have to compare the given key with the segment's nodes to decide how to continue. In this case, the possibilities are fairly simple. Besides the case that the given key may be equal to the root's key (which we will handle last), the path may continue in either of the two subtrees left and right of the node. They are both subtrees with a top that consists of three black nodes. Instead of inspecting them any further, we just turn the respective's subtree's root red and perform a recursive call of `deleteR` on it. Termination is once again guaranteed by its smaller size.

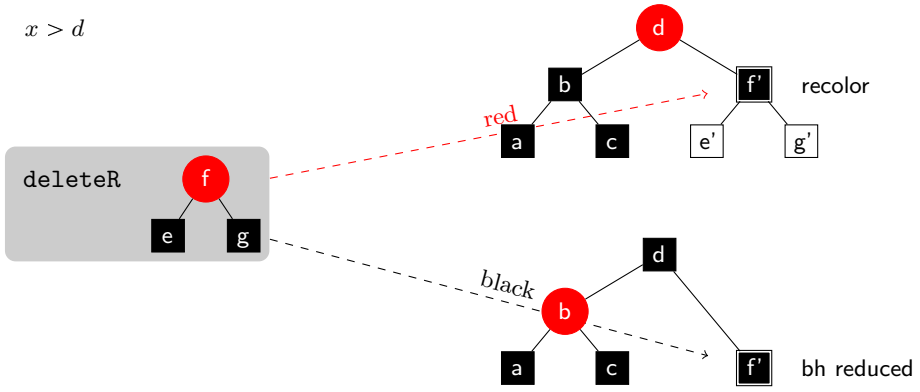
From a similar case in `extractMinR`, recall that turning a black subtree red decrements its black height. While discussing that case, we discovered that this is no problem if the function we applied afterwards returns a red result: we can turn the node back to its original black color and thus also reincrement the subtree's black height to its former value. In the case that a black result was returned, however, we still need to reconfigure the tree segment to a different shape.

Our first full subcase within this shape occurs when the node we look for lies somewhere in the left subtree:




```
-- 2.4.1
deleteR x (nr d pdl pdr
                  (nb b pbl pbr
                      (nb a pal par al ar)
                      (nb c pcl pcr cl cr))
                  (nb f pfl (d<f , pfr)
                      (nb e pel per el er)
                      (nb g pgl pgr gl gr)))
    | tri< x<d _ _ with deleteR x (nr b pbl pbr
                                                (nb a pal par al ar)
                                                (nb c pcl pcr cl cr))
... | red   , (nr b' pbl' pbr' a' c') = , nr d pdl pdr
                                          (nb b' pbl' pbr' a' c')
                                          (nb f pfl (d<f , pfr)
                                              (nb e pel per el er)
                                              (nb g pgl pgr gl gr))
... | black , b' = ,
  let b'' = b' <, cover d<f , ▪
      d' = nr d (d<f , pdl) pdr b'' (nb e pel per el er)
      g' = nb g pgl pgr gl gr <,, keep skip ▪
  in nb f pfl pfr d' g'
```

The subcase occuring when we have to continue in the right tree is similar, but the reconfiguration in the case of a black result looks simpler, since we can just swap the color of root and left child (in the previous subcase, the Left Leaning invariant prevented this).




```
-- 2.4.3
deleteR x (nr d pdl pdr
                  (nb b pbl pbr
                      (nb a pal par al ar)
                      (nb c pcl pcr cl cr))
                  (nb f pfl (d<f , pfr)
                      (nb e pel per el er)
                      (nb g pgl pgr gl gr)))
    | tri> _ _ x>d with deleteR x (nr f pfl (d<f , pfr)
                                       (nb e pel per el er)
                                       (nb g pgl pgr gl gr))
... | red   , nr f' pfl' pfr' e' g' = , nr d pdl pdr
                                        (nb b pbl pbr
                                            (nb a pal par al ar)
                                            (nb c pcl pcr cl cr))
                                        (nb f' pfl' pfr' e' g')
... | black , nb f' pfl' pfr' e' g' = , nb d pdl pdr
                                        (nr b pbl pbr
                                            (nb a pal par al ar)
                                            (nb c pcl pcr cl cr))
                                        (nb f' pfl' pfr' e' g')
```

The only missing subcase is the one where the root's key is equal to the one we are looking for. Naively, we might just attempt to remove that node completely and

reassemble the remaining nodes into a new tree segment. This, however, is not possible for the following reason:

The segment we visit in this case has four nodes on its lowest level. Attached to each of these nodes are two subtrees, which results in 8 subtrees attached to the whole segment, all with equal black height. We cannot change these black heights, as the only means for doing so would be by recoloration, and even though we can assume a black color for half of the subtree's roots (because they are right children), we cannot assume that their children are black, too, which would be a prerequisite for recoloring the subtrees red. This means that we need to create a shape that continues to accomodate exactly 8 subtrees with equal black height. However, in LLRBs, there are no shapes with 6 nodes that fulfill this condition.

Because it is not possible to generally remove nodes from segments without affecting the subtrees attached to the segment, the imperative algorithm described by **?** extracts the minimum of the subtree right of the matching node (in our case, this subtree starts at `f`), deleting the minimum node in the process, and replaces the key in the matching node with the obtained minimum.

We choose to follow a simpler approach, that fits well into our functional, pattern-matching algorithm: instead of deleting the minimum, we simply reapply `deleteR` on a *smaller* tree that *still contains the node that is to be removed.* As this is a smaller tree, the algorithm will eventually reach a base case and remove the matching node–which in this case has been handed all the way down from the tree segment in which it first occurred.

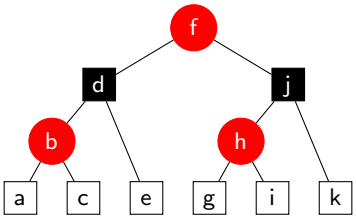
In effect, instead of explicitly deleting the minimum (reshaping the tree in the process) and replacing the key of the matching node, we continuously reshape the tree on our way down until the matching node becomes part of a base case which will remove it.

In addition to making `deleteR` completely self-contained, this approach spares us from proving that the right minimum can take the place in the node which contained the matching key. It also makes any case in which the given key is found anywhere as part of the visited segment similar to the others and in no way special.

```
-- 2.4.2
deleteR x (nr d pdl pdr
                  (nb b (b<d , pbl) pbr
                      (nb a pal par al ar)
                      (nb c (c<d , pcl) (b<c , pcr) cl cr))
                  (nb f pfl (d<f , pfr)
                      (nb e (e<f , pel) (d<e , per) el er)
                      (nb g pgl pgr gl gr)))
    | tri≈ _ x≈d _ with deleteR x (nr d (d<f , pdl) (b<d , pdr)
```

```
                                        (nb c (c<d , pcl) (b<c , pcr) cl cr <, cover d<f , ▪)
                                        (nb e (e<f , pel) (d<e , per) el er <,, cover b<d , ▪))
... | red   , nr d' (d<f' , pdl') (b<d' , pdr') c' e' = ,
  let a' = nb a pal par al ar <, cover b<d' , keep keep skip ▪
      c'' = c' <, keep skip ▪
      b' = nb b (b<d' , pbl) pbr a' c''
      e'' = e' <,, keep skip ▪
      g' = nb g pgl pgr gl gr <,, cover d<f' ,  keep keep skip ▪
      f' = nb f pfl (d<f' , pfr) e'' g'
  in nr d' pdl' pdr' b' f'
... | black , d' = ,
  let a' = nb a pal par al ar <, keep cover d<f , skip ▪
      b' = nr b (trans b<d d<f , pbl) pbr a' d'
      g' = nb g pgl pgr gl gr <,, keep skip ▪
  in nb f pfl pfr b' g'
```

After handling the case in which both grandchildren are black, we can take a look at the opposite case, in which both grandchildren are red:

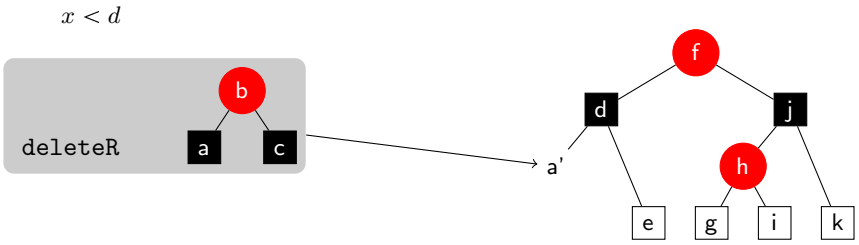



```
-- 2.3
deleteR x (nr f pfl pfr
                  (nb d pdl pdr
                      (nr b pbl pbr a c)
                      e)
                  (nb j pjl pjr
                      (nr h phl phr g i)
                      k)) with compare x d
```

Because, as we will see, there are more subcases to handle, more comparisons are made to find the right subcase. We start by comparing the given key with node `d`. It turns out that a really simple subcase emerges if the given key is smaller than `d`: the subtree left of `d` is already a red subtree, so we can just call `deleteR` on it. Moreover, for the fact alone that it is a red subtree in its present position, the segment's shape is able to

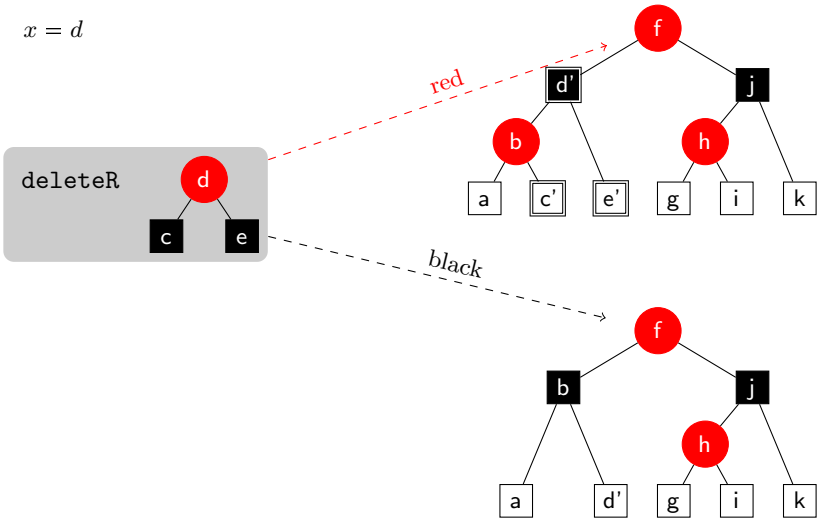
accomodate a red or a black subtree at this position, so we can directly plug the result subtree without any change of shape!

It's immediately visible that this also applies to subtree `h`, which is in a similar position, but due to our chaining of comparisons we will handle that later.




```
-- 2.3.1
deleteR x (nr f pfl pfr
                  (nb d pdl pdr
                      (nr b pbl pbr a c)
                      e)
                  (nb j pjl pjr
                      (nr h phl phr g i)
                      k))
    | tri< x<d _ _ with deleteR x (nr b pbl pbr a c)
... | _ , b' = , nr f pfl pfr (nb d pdl pdr b' e) (nb j pjl pjr (nr h phl phr g i) k)
```

In the case that `d` is the matching node:

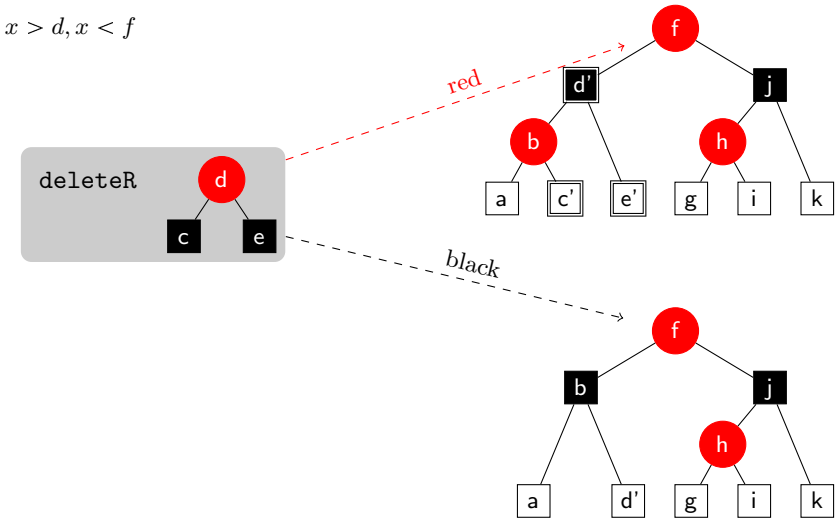


```
-- 2.3.2a
deleteR x (nr f pfl pfr
                  (nb d (d<f , pdl) pdr
                        (nr b (b<d , b<f , pbl) pbr a c)
                        e)
                  (nb j pjl pjr
                        (nr h phl phr g i)
                        k))
    | tri≈ _ x≈d _ with deleteR x (nr d (d<f , pdl) (b<d , pdr) c ( e <,, cover b<d , ▪ ))
... | red    , nr d' (d<f' , pdl') (b<d' , pdr') c' e' = ,
  let e'' = e' <,, keep skip ▪
      a' = a <, cover b<d' , keep keep skip ▪
      b' = nr b (b<d' , b<f , pbl) pbr a' c'
      d'' = nb d' (d<f' , pdl') pdr' b' e''
  in nr f pfl pfr d'' (nb j pjl pjr (nr h phl phr g i) k)
... | black  , d' = ,
  let a' = a <, keep skip ▪
      b' =  nb b (trans b<d d<f , pbl) pbr a' d'
  in nr f pfl pfr b' (nb j pjl pjr (nr h phl phr g i) k)
```

And if the given key is greater than the value in `d`, we start comparing with the root `f` for some further subcases:

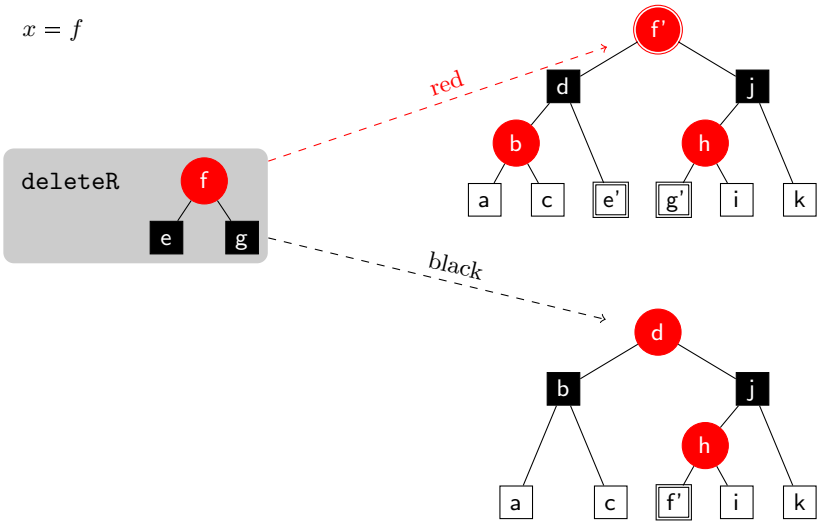
```
deleteR x (nr f pfl pfr
                  (nb d pdl pdr
                        (nr b pbl pbr a c)
                        e)
                  (nb j pjl pjr
                        (nr h phl phr g i)
                        k))
    | tri> _ _ x>d with compare x f
```

Let our given key be greater than `d`, but still less than `f`. A matching node must be somewhere in the subtree whose root is `e`. We can just recycle the previous subcase, because the tree on which the recursive call is applied includes `e`:




```
-- 2.3.2b
deleteR x (nr f pfl pfr
                  (nb d (d<f , pdl) pdr
                        (nr b (b<d , b<f , pbl) pbr a c)
                        e)
                  (nb j pjl pjr
                        (nr h phl phr g i)
                        k))
    | tri> _ _ x>d | tri< x<f _ _ with deleteR x (nr d (d<f , pdl) (b<d , pdr)
                                                      c
                                                      (e <,, cover b<d , ▪))
... | red    , nr d' (d<f' , pdl') (b<d' , pdr') c' e' = ,
  let e'' = e' <,, keep skip ▪
      a' = a <, cover b<d' , keep keep skip ▪
      b' = nr b (b<d' , b<f , pbl) pbr a' c'
      d'' = nb d' (d<f' , pdl') pdr' b' e''
  in nr f pfl pfr d'' (nb j pjl pjr (nr h phl phr g i) k)
... | black  , d' = ,
  let a' = a <, keep skip ▪
      b' =  nb b (trans b<d d<f , pbl) pbr a' d'
  in nr f pfl pfr b' (nb j pjl pjr (nr h phl phr g i) k)
```

A matching root is handled as follows. Again, we just pass the key down until we reach a base case and incorporate the result:




```
-- 2.3.3
deleteR x (nr f pfl pfr
                    (nb d (d<f , pdl) pdr
                          (nr b (b<d , b<f , pbl) pbr a c)
                          e)
                    (nb j pjl (f<j , pjr)
                        (nr h (h<j , phl) (f<h , phr) g i)
                        k))
    | tri> _ _ x>d | tri≈ _ x≈f _ with deleteR x (nr f (f<h , f<j , pfl) (d<f , pfr)
                                                       (e <, cover f<j , cover f<h , ▪)
                                                       (g <,, cover d<f , ▪))
... | red   , nr f' (f<h' , f<j' , pfl') (d<f' , pfr') e' g' = ,
  let e'' = e' <, keep skip skip ▪
      g'' = g' <,, keep skip ▪
      k'  = k <,, cover f<j' , keep keep skip ▪
      i'  = i <,, cover f<h' , keep keep skip ▪
      h'  = nr h (h<j , phl) (f<h' , phr) g'' i'
      j'  = nb j pjl (f<j' , pjr) h' k'
      a'  = a <, keep cover d<f' , keep keep skip ▪
      c'  = c <, cover d<f' , keep keep skip ▪
      b'  = nr b (b<d , trans b<d d<f' , pbl) pbr a' c'
      d'  = nb d (d<f' , pdl) pdr b' e''
  in nr f' pfl' pfr' d' j'
... | black , f' = ,
  let k' = k <,, keep cover d<f , skip ▪
      i' = i <,, keep cover d<f , skip ▪
      h' = nr h (h<j , phl) (trans d<f f<h , phr) f' i'
```

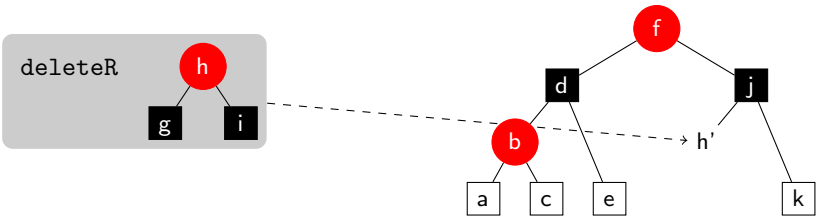
```
      j' = nb j pjl (trans d<f f<j , pjr) h' k'
      b' = nb b (b<d , b<f , pbl) pbr a c <, keep skip ▪
  in nr d pdl pdr b' j'
```

As we go further to the right of the segment, we now need another comparison:

```
deleteR x (nr f pfl pfr
                  (nb d pdl pdr
                        (nr b pbl pbr a c)
                        e)
                  (nb j pjl pjr
                        (nr h phl phr g i)
                        k))
    | tri> _ _ x>d | tri> _ _ x>f with compare x j
```

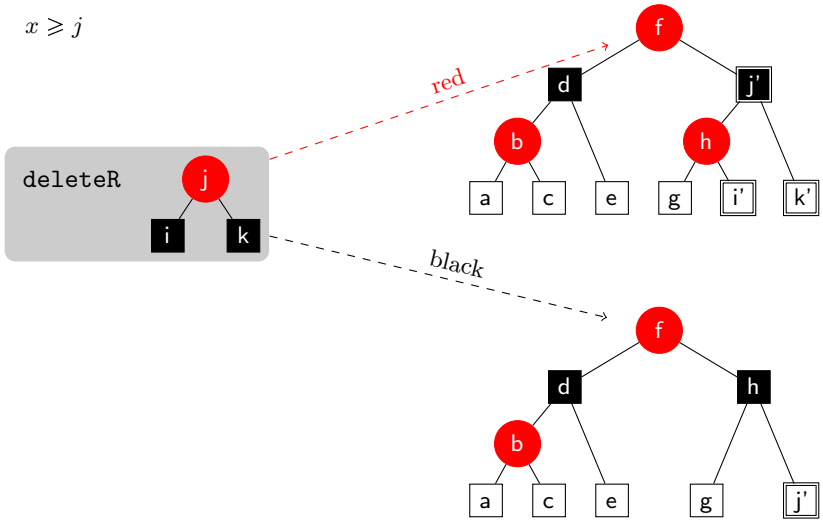
If the node lies somewhere in `h`'s subtree, the case is very similar to `b`'s subtree, we can just call `deleteR` on `h` and plug the result in:

$x > f, x < j$




```
-- 2.3.4
deleteR x (nr f pfl pfr
                  (nb d pdl pdr
                        (nr b pbl pbr a c)
                        e)
                  (nb j pjl pjr
                        (nr h phl phr g i)
                        k))
    | tri> _ _ x>d | tri> _ _ x>f | tri< x<j _ _ with deleteR x (nr h phl phr g i)
... | _ , h' = , nr f pfl pfr (nb d pdl pdr (nr b pbl pbr a c) e) (nb j pjl pjr h' k)
```

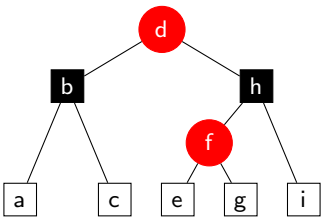
The remaining case is when we have to continue right, at `j` or `k`.



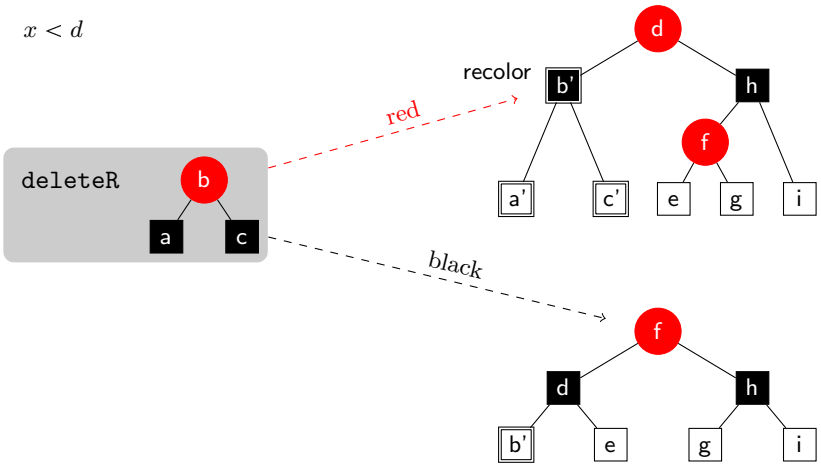

```
-- 2.3.5a
deleteR x (nr f pfl pfr
                  (nb d pdl pdr
                        (nr b pbl pbr a c)
                        e)
                  (nb j pjl (f<j , pjr)
                        (nr h (h<j , phl) (f<h , phr) g i)
                        k))
    | tri> _ _ x>d | tri> _ _ x>f | _ with deleteR x (nr j pjl (h<j , f<j , pjr)
                                                              i
                                                              (k <,, cover h<j , ▪))

... | red   , nr j' pjl' (h<j' , f<j' , pjr') i' k' = ,
  let k'' = k' <,, keep skip ▪
      g'  = g <, cover h<j' , keep keep skip ▪
      h'  = nr h (h<r , phl) (f<h , phr) g' i'
      j''  = nb j' pjl' (f<j' , pjr') h' k''
  in nr f pfl pfr (nb d pdl pdr (nr b pbl pbr a c) e) j''
... | black , j' = ,
  let g' = g <, keep skip ▪
      h' = nb h phl (f<h , phr) g' j'
  in nr f pfl pfr (nb d pdl pdr (nr b pbl pbr a c) e) h'
```

Remaining are the cases where either but not both of the root's grand children are read. They are handled in a similar manner.

7.1 Red Grandchild Right

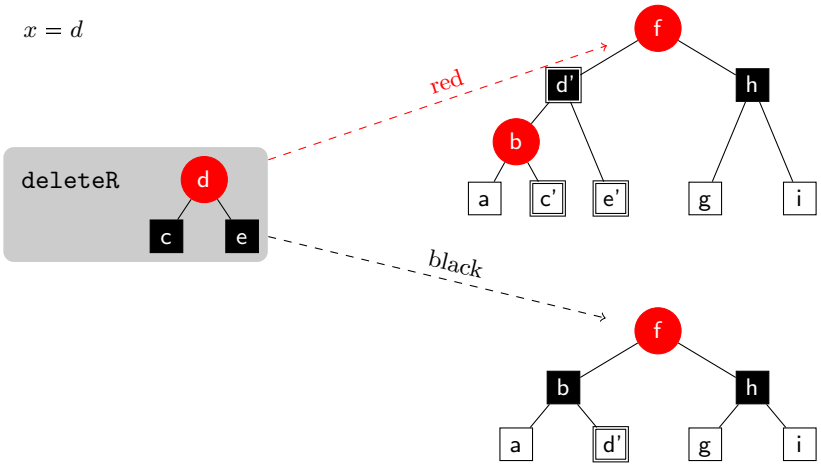



```
-- 2.2
deleteR x (nr d pdl pdr
                  (nb b pbl pbr
                      (nb a pal par al ar)
                      c)
                  (nb h phl phr
                      (nr f pfl pfr e g) i)) with compare x d
```



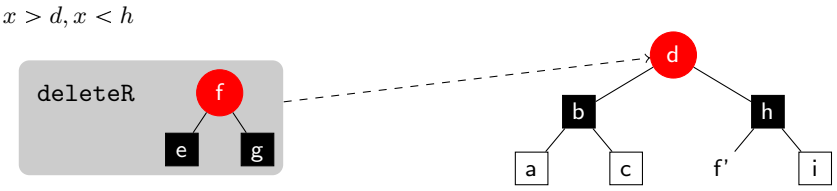

```
-- 2.2.1
deleteR x (nr d pdl pdr
                  (nb b pbl pbr
                      (nb a pal par al ar)
                      c)
                  (nb h phl (d<h , phr)
                      (nr f (f<h , pfl) (d<f , pfr) e g)
```

```
                    i))
    | tri< x<d _ _ with deleteR x (nr b pbl pbr (nb a pal par al ar) c <, cover d<f , ▪)
... | red   , nr b' pbl' pbr' a' c' = ,
  let b'' = nb b' pbl' pbr' a' c' <, keep skip ▪
  in nr d pdl pdr b'' (nb h phl (d<h , phr) (nr f (f<h , pfl) (d<f , pfr) e g) i)
... | black , b' = ,
  let e' = e <, keep skip ▪
      d' = nb d (d<f , pdl) pdr b' e'
      g' = g <,, keep skip ▪
      i' = i <,, cover f<h , keep keep skip ▪
      h' = nb h phl (f<h , phr) g' i'
  in nr f pfl pfr d' h'
```

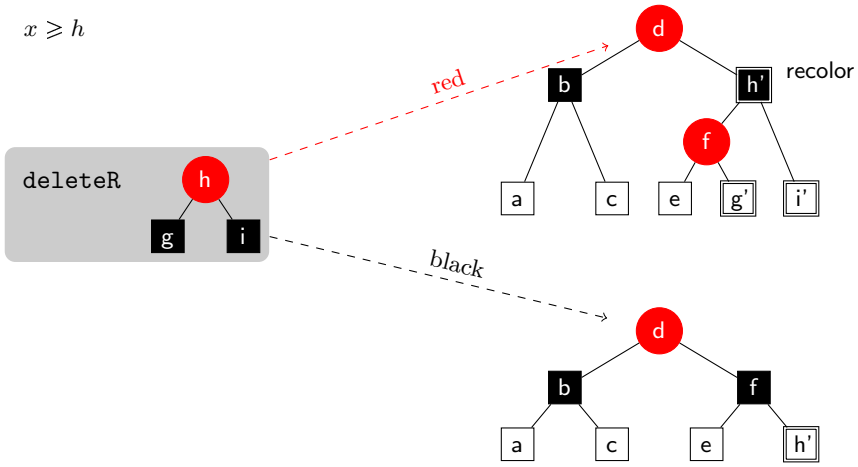



```
-- 2.2.2
deleteR x (nr d pdl pdr
                  (nb b (b<d , pbl) pbr
                      (nb a pal par al ar)
                      c)
                  (nb h phl (d<h , phr)
                      (nr f (f<h , pfl) (d<f , pfr) e g)
                      i))
    | tri≈ _ x≈d _ with deleteR x (nr d (d<f , pdl) (b<d , pdr)
                                       (c <, cover d<f , ▪)
                                       (e <, keep skip ▪ <,, cover b<d , ▪))
... | red   , nr d' pdl' (b<d' , pdr') c' e' = ,
  let a'  = nb a pal par al ar <, cover b<d' , keep keep cover d<f , skip ▪
      b'  = nr b (b<d' , trans b<d d<f , pbl) pbr a' c'
      e'' = e' <,, keep skip ▪
      d''  = nb d' pdl' pdr' b' e''
      g'  = g <,, keep skip ▪
      i'  = i <,, cover f<h , keep keep skip ▪
      h'  = nb h phl (f<h , phr) g' i'
  in nr f pfl pfr c'' h'
... | black , d' = ,
  let a' = nb a pal par al ar <, keep cover d<f , skip ▪
      b' = nb b (trans b<d d<f , pbl) pbr a' d'
      g' = g <,, keep skip ▪
      i' = i <,, cover f<h , keep keep skip ▪
      h' = nb h phl (f<h , phr) g' i'
  in nr f pfl pfr b' h'
```

```
deleteR x (nr d pdl pdr
                  (nb b pbl pbr
                      (nb a pal par al ar)
                      c)
                  (nb h phl phr
                      (nr f pfl pfr e g) i))
    | tri> _ _ x>d with compare x h
```

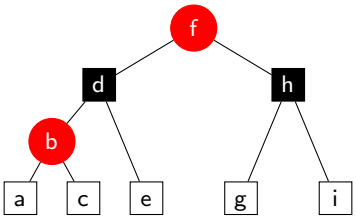



```
-- 2.2.3
deleteR x (nr d pdl pdr
                  (nb b pbl pbr
                      (nb a pal par al ar)
                      c)
                  (nb h phl phr
                      (nr f pfl pfr e g) i))
    | tri> _ _ x>d | tri< x<h _ _ with deleteR x (nr f pfl pfr e g)
... | _ , f' = , nr d pdl pdr (nb b pbl pbr (nb a pal par al ar) c) (nb h phl phr f' i)
```

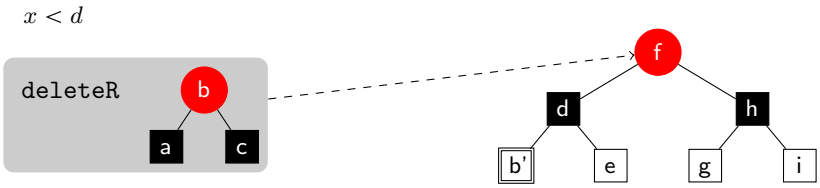



```
-- 2.2.4
deleteR x (nr d pdl pdr
                  (nb b pbl pbr
                      (nb a pal par al ar)
                      c)
                  (nb h phl phr
                      (nr f (f<h , pfl) (d<f , pfr) e g) i))
    | tri> _ _ x>d | _ with deleteR x (nr h phl (f<h , phr)
                                                    g
                                                    (i <,, cover f<h , ▪))
... | red   , nr h' phl' (f<h' , d<h' , phr') g' i' = ,
  let e'  = e <, cover f<h' , keep keep skip ▪
      f'  = nr f (f<r , pfl) (d<f , pfr) e' g'
      i'' = i' <,, keep skip ▪
      h''  = nb h' phl' (d<r , phr') f' i''
  in nr d pdl pdr (nb b pbl pbr (nb a pal par al ar) c) h''
... | black , h' = ,
  let e' = e <, keep skip ▪
      f' = nb f pfl (d<f , pfr) e' h'
  in nr d pdl pdr (nb b pbl pbr (nb a pal par al ar) c) f'
```

7.2 Red Grandchild Left

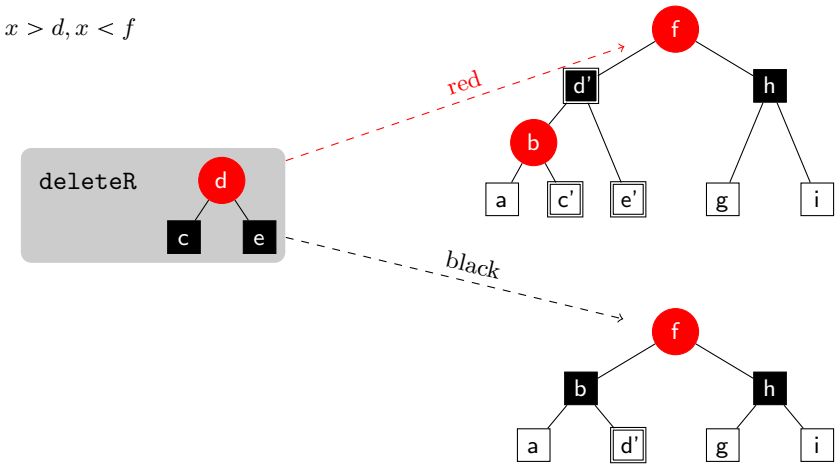



```
-- 2.1
deleteR x (nr f pfl pfr
                   (nb d pdl pdr
                       (nr b pbl pbr a c)
                       e)
                   (nb h phl phr
                       (nb g pgl pgr gl gr)
                       i)) with compare x d
```



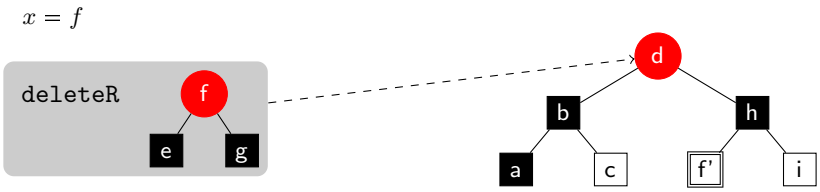

```
-- 2.1.1
deleteR x (nr f pfl pfr
                   (nb d pdl pdr
                       (nr b pbl pbr a c)
                       e)
                   (nb h phl phr
                       (nb g pgl pgr gl gr)
                       i))
    | tri< x<d _ _ with deleteR x (nr b pbl pbr a c)
... | _ , b' = , nr f pfl pfr (nb d pdl pdr b' e) (nb h phl phr (nb g pgl pgr gl gr) i)
```

```
deleteR x (nr f pfl pfr
                  (nb d pdl pdr
                      (nr b pbl pbr a c)
                      e)
                  (nb h phl phr
                      (nb g pgl pgr gl gr)
                      i))
    | tri> _ _ x>d with compare x f
```

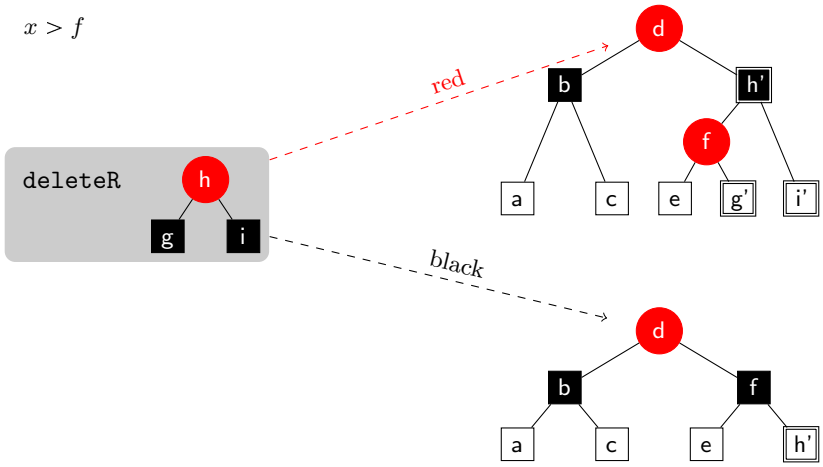



```
-- 2.1.2
deleteR x (nr f pfl pfr
                  (nb d pdl pdr
                      (nr b (b<d , b<f , pbl) pbr a c)
                      e)
                  (nb h phl phr
                      (nb g pgl pgr gl gr)
                      i))
    | tri> _ _ x>d | tri< x<f _ _ with deleteR x (nr d pdl (b<d , pdr)
                                                        c
                                                        (e <,, cover b<d , ▪))
... | red , nr d' (d<f' , pdl') (b<d' , pdr') c' e' = ,
  let e'' = e' <,, keep skip ▪
      a'  = a <, cover b<d' , keep keep skip ▪
      b'  = nr b (b<d' , b<f , pbl) pbr a' c'
      d''  = nb d' (d<f' , pdl') pdr' b' e''
  in nr f pfl pfr d'' (nb h phl phr (nb g pgl pgr gl gr) i)
```

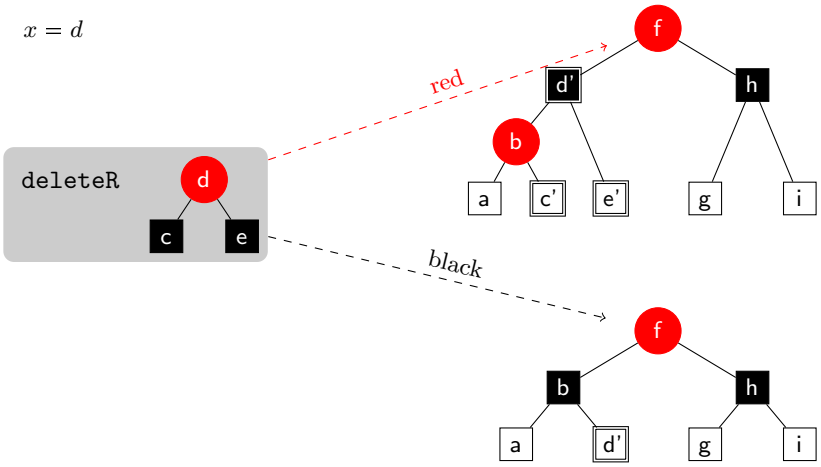
```
... | black , d' = ,
  let a' = a <, cover b<f , keep skip skip ▪
      b' = nb b (b<f , pbl) pbr a' d'
  in nr f pfl pfr b' (nb h phl phr (nb g pgl pgr gl gr) i)
```




```
-- 2.1.3
deleteR x (nr f pfl pfr
                  (nb d (d<f , pdl) pdr
                      (nr b (b<d , b<f , pbl) pbr a c)
                      e)
                  (nb h phl (f<h , phr)
                      (nb g pgl pgr gl gr)
                      i))
    | tri> _ _ x>d | tri≈ _ x≈f _ with deleteR x (nr f (f<h , pfl) (d<f , pfr)
                                                        (e <, cover f<h , ▪)
                                                        (nb g pgl pgr gl gr <,, cover d<f , ▪)
... | _ , f' = ,
  let b' = nb b (b<d , b<f , pbl) pbr a c <, keep skip ▪
      i' = i <,, keep cover d<f , skip ▪
      h' = nb h phl (trans d<f f<h , phr) f' i'
  in nr d pdl pdr b' h'
```




```
 -- 2.1.4
deleteR x (nr f pfl pfr
                  (nb d (d<f , pdl) pdr
                      (nr b (b<d , b<f , pbl) pbr a c)
                      e)
                  (nb h phl (f<h , phr)
                      (nb g pgl pgr gl gr)
                      i))
    | tri> _ _ x>d | tri> _ _ x>f with deleteR x (nr h phl (f<h , phr)
                                                      (nb g pgl pgr gl gr)
                                                      i
                                                <,, cover d<f , ▪)
... | red , nr h' phl' (f<h' , d<h' , phr') g' i' = ,
  let b'  = nb b (b<d , b<f , pbl) pbr a c <, keep skip ▪
      e'  = e <, cover f<h' , ▪
      f'  = nr f (f<r , pfl) (d<f , pfr) e' g'
      i'' = i' <,, keep skip ▪
      h''  = nb h' prl (d<h' , phr') f' i''
  in nr d pdl pdr b' h''
... | black , h' = ,
  let b' = nb b (b<d , b<f , pbl) pbr a c <, keep skip ▪
      f' = nb f pfl (d<f , pfr) e h'
  in nr d pdl pdr b' f'
```




```
deleteR x (nr f pfl pfr
                  (nb d pdl pdr
                      (nr b (b<d , b<f , pbl) pbr a c)
                      e)
                  (nb h phl phr
                      (nb g pgl pgr gl gr)
                      i))
    | tri≈ _ x≈d _ with deleteR x (nr d pdl (b<d , pdr)
                                        c
                                        (e <,, cover b<d , ▪))
... | red , nr d' (d<f' , pdl') (b<d' , pdr') c' e' = ,
  let e'' = e' <,, keep skip ▪
      a'  = a <, cover b<r , keep keep skip ▪
      b'  = nr b (b<d' , b<f , pbl) pbr a' c'
      d''  = nb d' (d<f' , pdl') pdr' b' e''
  in nr f pfl pfr d'' (nb h phl phr (nb g pgl pgr gl gr) i)
... | black , d' = ,
  let a' = a <, cover b<f , keep skip skip ▪
      b' = nb b (b<f , pbl) pbr a' d'
  in nr f pfl pfr b' (nb h phl phr (nb g pgl pgr gl gr) i)
```

8 Putting it all together

We now have two important parts of our algorithm, removal of the minimal node in a red subtree and deletion of arbitrary nodes in a red subtree. But since our goal is to artibtrarily delete nodes in a correct LLRB, which is first and foremost a tree with a black root, we need to put this two parts together in another function, and handle some additional cases emerging from it.

This function will delete arbitary keys in black subtrees (we will introduce a simple wrapper function that operates on the exported, simple `Tree` datatype instead of subtrees of type `Tree'` later on). A first attempt to write its type may lead to the following signature:

```
deleteB : ∀ {n β γ} → A → Tree' β γ black n → Tree' β γ black ?
```

It takes a key of type `A` and a *black* subtree and returns a possibly modified tree. We can constrain the returned tree to be black, because a red tree would just need a simple recoloration to be turned into a black one. We haven't filled out its resulting black height, however. It can't be fixed, because as shown earlier, it is not generally possible to remove a node from a black subtree without potentially decrementing its black height.

To take this into account, we include an additional bit that indicates whether the subtree's black height was decremented or not. To allow a result with such a decremented black height, we need to specify the given tree's blackheight as `suc n` instead of n, so that the returned tree can be either of black height `suc n` or `n`, depending on the bit `z`. This also implies that trees given to `deleteB` must have a black height of at least 1. The only valid (black-rooted) LLRB for which this isn't true is the empty one, but we can handle this very simple special case later, in the outer wrapper function.

The full type is thus:

```
-- the returned bit z indicates whether the tree's black height has shrunk
deleteB : ∀ {n β γ} → A → Tree' β γ black (suc n) → ∃ λ z
          → Tree' β γ black (if z then n else (suc n))
```

While this type seems unwieldy for general use, recall that the exported `Tree` data type contains no indication of any black height, hence leading us to discard the unnecessary bit and black height in the outer wrapper, which yields to a simple and intuitive type.

As always, we will first look at the simple base cases (just one in this function), before dealing with the more complex recursive cases.

The simple base case is a tree with just one black node, which may be removed:

```
-- case terminal node
deleteB x (nb a pal par lf lf) with x ≟ a
... | yes _ = true , lf                    -- shrunk (black height reduced)
... | no  _ = false , nb a pal par lf lf   -- not shrunk (black height preserved)
```

On to bigger trees, we must now look at the root's left child, which is either red or black. A subtree with a black child is easily handled, as we can just recolor its root red and call `deleteR` on it, which does all the work! Whether we have to decrement the black height or not depends on the color of `deleteR`'s result. If it is red, we can immediately recolor it black and black height has been preserved. An already black result instead leads to a decreased black height.

```
-- case 2-node: color red and call deleteR
deleteB x (nb b pbl pbr (nb a pal par al ar) br)
  with deleteR x (nr b pbl pbr (nb a pal par al ar) br)
... | red   , nr r prl prr rl rr = false , nb r prl prr rl rr -- red --> black
... | black , nb r prl prr rl rr = true  , nb r prl prr rl rr -- already black ==> shrunk
```

If the root's left child is red, a little more work is required, as we cannot recolor the root without violating the Left-Leaning Red-Black invariant. We start by checking which path we must follow:

```
-- case 3-node
deleteB x (nb b pbl pbr (nr a pal par al ar) br) with compare x b
```

Continuing left is the easiest case. The left subtree is red, so we can again just call `deleteR` on it. Because, this time, we don't care about the color of the result–we can plug it in either way–no corrections are necessary.

```
-- delete in left (red) subtree
deleteB x (nb b pbl pbr (nr a pal par al ar) br) | tri< x<b _ _
  with (deleteR x (nr a pal par al ar))
... | _ , bl' = false , nb b pbl pbr bl' br -- whatever comes back, no shrinking
```

The right subtree, however, is forcefully black, so this again requires special attention. But first, we handle the case of a *missing* right subtree, i.e. we would have to continue there, but there is only a leaf. This obviously means that the key isn't in the tree and nothing happens:

```
-- would delete in right (black) subtree, but it is a leaf
deleteB x (nb b pbl pbr (nr a pal par al ar) lf) | tri> _ _ x>b =
  false , (nb b pbl pbr (nr a pal par al ar) lf)
```

On to a "true" black subtree. We can recursively call `deleteB` on it. Note that unlike `extractMinR` and `deleteR`, this function may return a tree of a differing black height, which creates more cases.

```
-- delete in right (black) subtree
deleteB x (nb h phl phr (nr b pbl pbr bl br) (nb i pil pir il ir)) | tri> x<h x≈h x>h
  with (deleteB x (nb i pil pir il ir))
```

Reassembling is simple, without any needed corrections, if the returned tree retained its former black height:

```
-- no shrinkage, just reassemble
deleteB x (nb h phl phr (nr b pbl pbr bl br) (nb i pil pir il ir)) | tri> x<h x≈h x>h | fals
  false , nb h phl phr (nr b pbl pbr bl br) r
```

Otherwise, we need to reconfigure the tree accordingly:

```
-- if there was shrinkage, we need to merge with right brother or parts of it
{- case: right brother f is a 2-node
             [h]              [b]
   (b)                     [a]         [h]
[a]     [f]           --->         (f)
     [d]   [g]   [r]             [d] [g]  [r]
-}
deleteB x (nb h phl phr (nr b (b<h , pbl) pbr a (nb {leftSonColor = black} f pfl pfr d g)) (r
  false , nb b pbl pbr
             (a <, keep skip ▪)
             (nb h phl (b<h , phr)
                 (nr f pfl pfr d g <, ▪)
                 (r <,, cover b<h , ▪))

{- case: right brother f is a 3-node
             [h]                    [f]
   (b)                         (b)
[a]     [f]           ---> [a]  [d]      [h]
    (d)
```

```
  [c] [e] [g]   [r]           [c] [e]  [g] [r]
-}
deleteB x (nb h phl phr (nr b (b<h , pbl) pbr a (nb f (f<h , pfl) (b<f , pfr) (nr d pdl pdr c
  false , nb f pfl pfr
             (nr b (b<f , pbl) pbr
                 (a <, cover b<f , keep keep skip ▪)
                 (nb d pdl pdr c e <, keep skip ▪))
             (nb h phl (f<h , phr)
                 (g <,, keep skip ▪)
                 (r <,, cover f<h , ▪))
```

This concludes what may happen if deletion continued in the right subtree, and leaves only the case where the given key was the root itself. Our strategy is simple: we extract the minimum node from the *right* subtree (`extractMinR` comes into play), replace the root's key with it and merge the new right subtree back into the tree, depending on its color and the shape of the left subtree:

```
-- delete root

{- case root is a terminal 3-node -}
deleteB x (nb d pdl pdr (nr b (b<d , pbl) pbr lf lf) lf) | tri≈ _ x≈d _ =
  false , nb b pbl pbr lf lf

{- case right son is a 2-node, left-right grandchild is a 2-node
         [d]
  (b)
[a]  [c]      [h]     call extractMinR (h)
  [cl] [cr] [f] [i]                  [f] [i]
-}
deleteB x (nb d pdl pdr (nr b (b<d , pbl) pbr a (nb {leftSonColor = black} c pcl pcr cl cr))
... | min , black , pminl , (d<min , pminr) , r  =
  false , let a' = a <, keep skip ▪
              c' =  nr c pcl pcr cl cr <, cover d<min , skip ▪
              r'' = r <,, keep cover b<d , skip ▪
              d' = nb min pminl (trans b<d d<min , pminr) c' r''
          in nb b pbl pbr a' d'
... | min , red   , pminl , (d<min , pminr) , nr r prl prr rl rr  =
  false , let r' = nb r prl prr rl rr <,, keep skip ▪
              b' = nr b (b<d , pbl) pbr a (nb c pcl pcr cl cr) <, cover d<min , skip ▪
          in nb min pminl pminr b' r'

{- case right son is a 2-node, left-right grandchild is a 3-node
            [d]
```

```
    (b)
[a]     [c]       [h]     call extractMinR (h)
    (cl)   [cr] [f] [i]                  [f] [i]
[clr]  [crr]
-}
deleteB x (nb d pdl pdr (nr b (b<d , pbl) pbr a (nb c (c<d , pcl) (b<c , pcr) (nr cl pcll pcl
... | min , black , pminl , (d<min , pminr) , r  =
  false , let a' = a <, cover b<c , keep keep skip ▪
              cl' = nb cl pcll pclr cll clr <, keep skip ▪
              b' =  nr b (b<c , pbl) pbr a' cl'
              cr' = cr <,, keep skip ▪ <, cover d<min , skip ▪
              r' = r <,, keep cover c<d , skip ▪
              d' =  nb min pminl (trans c<d d<min , pminr) cr' r'
          in nb c pcl pcr b' d'
... | min , red   , pminl , (d<min , pminr) , nr r prl prr rl rr  =
  false , let r' = nb r prl prr rl rr <,, keep skip ▪
              b' = nr b (b<d , pbl) pbr a (nb c (c<d , pcl) (b<c , pcr) (nr cl pcll pclr cll
          in nb min pminl pminr b' r'

{- case right son is a 3-node
       [d]
  (b)
[a] [c]      [h]
          (f)        call extractMinR (f)
        [e] [g] [i]               [e] [g]
-}
deleteB x (nb d pdl pdr (nr b pbl pbr a c) (nb h phl (d<h , phr) (nr f (f<h , pfl) (d<f , pfr
... | result with extractMinR (nr f (f<h , pfl) (d<f , pfr) e g)
... | min , _ , (min<h , pminl) , (d<min , pminr) , r =
  false , let r' = r <,, keep skip  ▪
              i' = i <,, cover min<h , keep keep skip ▪
              h' =  nb h phl (min<h , phr) r' i'
              b' = nr b pbl pbr a c <, cover d<min , skip ▪
          in nb min pminl pminr b' h'
```

Finally, having everthing together, we can write the wrapper function, which takes an opaque `Tree` and applies `deleteB` to its inner `Tree'` value. Discarding all internal type information, it is suitable for exporting and we just call it `delete`:

```
delete : A → Tree → Tree
delete x (tree lf) = tree lf
delete x (tree (nb a pal par al ar)) with deleteB x (nb a pal par al ar)
... | _ , result = tree result
```

9 Conclusion

We have shown a purely functional algorithm for deletion, and have shown how a dependently typed language like Agda can ensure the validity and totality of all occurring cases and their subcases. To help us find proofs for bound implication in a mostly automatic manner, we have modified the domain specific language specified by **?** to contain a representation of bounds using two lists, for upper and lower bounds respectively, instead of one.

We ensured termination by only allowing smaller subtrees when recursing. Once *Sized Types* work as intended in Agda, additional work can be put into integrating them in this source code, to allow for automatic termination checking by Agda.

References

Linus Ek, Ola Holmström, and Stevan Andjelkovic. Formalizing arne andersson trees and left-leaning red-black trees in agda. Technical report, Chalmers Tekniska Högskola, 2009.

R. Sedgewick. Left-leaning red-black trees. In *Dagstuhl Workshop on Data Structures*. Citeseer, 2008.

Stefan Kahrs. Red-black trees with types. *J. Funct. Program.*, 11(4):425–432, 2001.

Chris Okasaki. Red-black trees in a functional setting. *J. Funct. Program.*, 9(4):471–477, 1999.