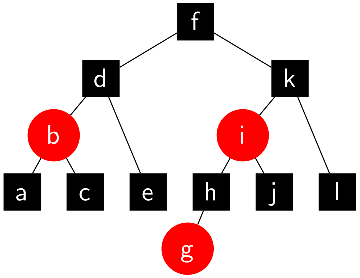
An Agda implementation of deletion in Left-leaning Red-Black trees

Julien Oster, Ludwig-Maximilians-Universität München

December 10th, 2010

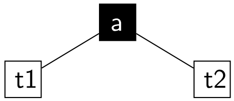
Left-Leaning Red-Black trees

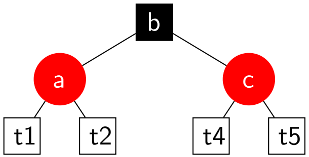


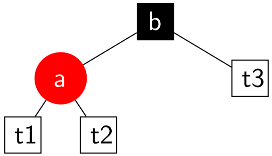
Invariants of LLRBs

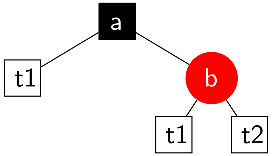
1. Left-Leaning Red-Black invariant
    - every node is either red or black
    - **right children are always black**
    - the root must be black
    - leafs are black by definition
2. Black Height invariant
    - each node is assigned a black height, the number of black nodes on every downward path from that node to a descendant leaf
    - consequently, for every node, black heights of both children must be equal
    - leafs have black height zero
3. Search Tree invariant
    - all left descendant keys of a node have smaller keys
    - all right descendant keys of a node have greater keys

Node types in **non**-Left-Leaning Red-Black trees

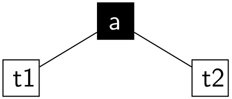


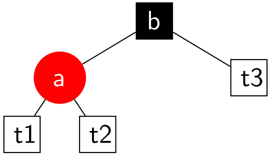






Node types in Left-Leaning Red-Black trees





Less patterns, less cases.

How Agda is going to help us

- **the data structure** will enforce all invariants: *correctness by definition*
- **coverage checking** will ensure that we do not forget any cases
- **termination checking** will make sure that the algorithm terminates, here using *Sized Types*[1]

[1] Unfortunately, they are buggy right now.

Module Parameterization

```
module llrb
  (order : StrictTotalOrder Level.zero Level.zero Level.zero)

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

This includes what we need, which is:

- 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.

Bounds

```
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.

Proof Types for Bounds

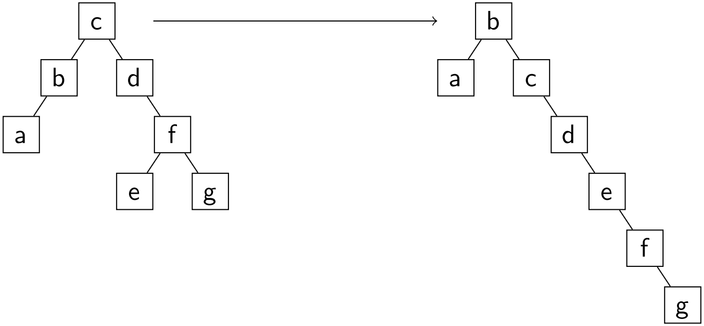
```
infix 5 _isleftof_
_isleftof_ : A → UpperBounds → Set
z isleftof []       = ⊤
z isleftof b :: β   = z < b × z isleftof β

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

Example

```
  a isleftof [b,c,d,e,f]   ==   a < b × a < c × a < d × ...
```

Bounds Change



for node e:

```
lower: c :: d :: β              lower: b :: c :: d :: β
upper: f :: γ                   upper: γ
```

Incompatible if we want to take a subtree from one tree and plug it into the other!

Bounds Implication

```
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'
```

Interpretation: *Given proof that* $B \Rightarrow^l B'$ *and that* x isleftof B, *we conclude that* x isleftof B'. and *given proof that* $B \Rightarrow {}^r B'$ *and that* x isrightof B, *we conclude that* x isrightof B'.

Seen as Transformations of Bounds Lists

Treat it as *a sequence of operations* on a bounds list.

- **keep** Keep the current bound and move on to the next.
- **skip** Skip the current bound (not needed anymore).
- **cover** Insert a new bound after the current bound (because a new node appeared along the path to the root).
  The current bound must be a *better* bound than the new one!
  Let a be any minimum bound for x. If $b < a$, then $b < x$ due to transitivity, and b is also a minimum bound for x. Analog with maximum bounds.
  This operation requires us to specify a proof of $b < a$.
- Keep the rest of the bounds.

Example for Bounds Implication/Transformation

Example from above:

```
lower: c :: d :: β              lower: b :: c :: d :: β
upper: f :: γ                   upper: γ
```

We can transform the lower bounds with:

```
cover b<c , keep keep ∎
```

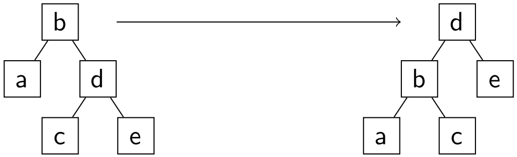
or simply:

```
cover b<c , ∎
```

And the upper bounds with:

```
skip ∎
```

Bounds Representation with Single List



With two lists, bounds lists for node c are:

```
lower: b :: β                  lower: b :: β
upper: d :: γ                  upper: d :: γ
```

But with just a single list:

```
leftOf d :: rightOf b :: β     rightOf b :: leftOf d :: β
```

Disadvantages of Single List Representation

- Ordering is not invariant
- Additional rule *swap*, was necessary, to change positions of bounds
- Proof system not linear, term searcher (Agsy) less likely to find proof
- Superfluous *keep* operations to jump over bounds

Interpretation of Bounds Implication Data Type

```
[[_]]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)
```

First function's type read as proposition:
Given proof that $B \Rightarrow^l B'$ and that x isleftof B, *we conclude that* x isleftof B'
Exactly what we stated as our desired interpretation!

Color

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

Could have used a boolean, but this is clearer.

The Data Type

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

Parameterized and indexed by:

- upper and lower bounds
- color of the tree (meaning its root)
- black height

Leafs

```
lf : Tree’ β γ black 0
```

Leafs are black and have black height 0.

Red Nodes

```
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
```

- black height *n*, children must have same height
- key value *a* of type *A*
- proof that key is within upper bounds
- proof that key is within lower bounds
- left child
  - must be black
  - upper bound prepended with key
- right child
  - must be black
  - lower bound prepended with key

Black Nodes

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

- black height *n*, **children's black height + 1**
- key value *a* of type *A*
- proof that key is within upper bounds
- proof that key is within lower bounds
- left child
    - **can be red or black**
    - upper bound prepended with key
- right child
    - must be black
    - lower bound prepended with key

How Invariants are Enforced

1. *Left-Leaning Red-Black Invariant*: tree indexed with color, specified accordingly in node constructors
2. *Black Height Invariant*: tree indexed with height, specified accordingly in node constructors
3. *Search Tree Invariant*: Bounds Lists maintained for all nodes

Transform a subtree's bounds

```
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)
```

We can now take subtrees and plug them into other trees, provided we can specify a proof of bounds implication.

Opaque Data Type

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

Hides implementation, suitable for exporting.

Overview of Algorithm

Some properties:

- Three functions.
- Two of the functions self-contained (in recursion pattern).
- Only using simple recursive descent.
- Only base cases actually remove nodes.

General approach:

- Traverse the tree down a path.
- Reconfigure while traversing it.
- If matching node is found during traversal, carry it down the path.
- When reaching a base case, removing matching node (if any).

extractMinR

extractMinR extracts the minimum node from a red tree. It returns minimum key and a modified tree.

extractMinR operates on red trees only!

This is because we can always remove nodes from red trees without affecting black height.

If black height would be affected, we would have to handle many more cases in every caller, including recursion!

Strategy of extractMinR

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 might not be possible to take existing subtree in step 3, we need a *red subtree*.
- The outcome of step 4 might cause some reconfiguration to merge the result back!

We are reconfiguring the tree's shape almost constantly during descent!

Type for extractMinR?

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

Operationally okay, but we are throwing some type information away...

Type for extractMinR!

```
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
```

Additionally includes:

- proof that minimum is within tree’s bounds
- bounds of resulting tree indicate that tree is right of minimum

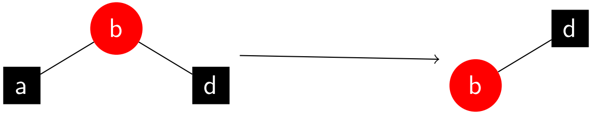
And we do need that stuff later on.

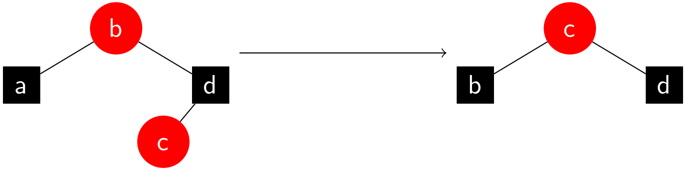
Base Cases

Tree with one node becomes empty tree:

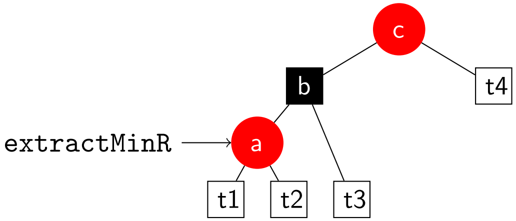
```
extractMinR (nr a pal par lf lf) = a , black , pal , par , lf
```

Further base cases:

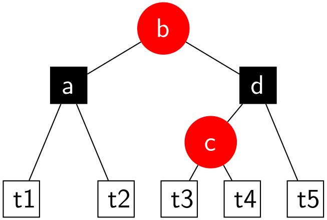


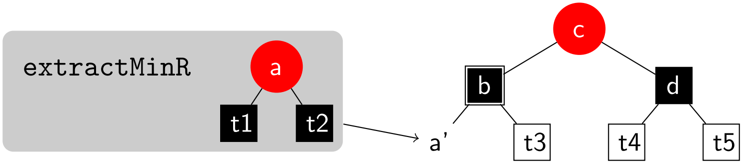


First Recursive Case

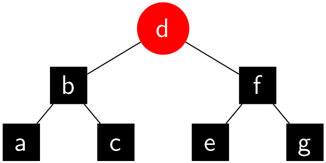


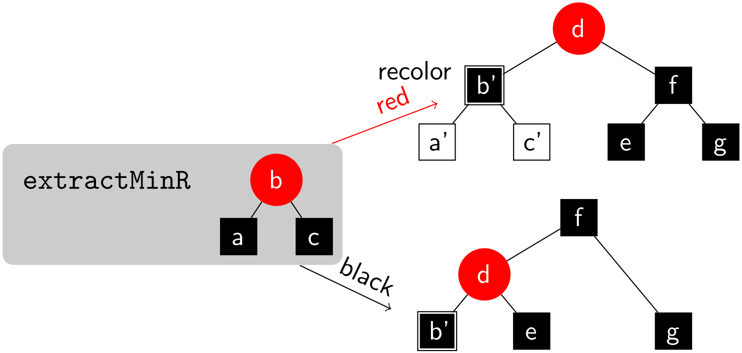
Second Recursive Case





And another.





Overview of deleteR

Shares overall approach with *extractMinR*:

- operates on red trees only (no black height reduction)
- reconfigures tree shape during traversal
- removes nodes only in base cases (carries matching node through tree in recursive cases)

The main difference is that we don't just follow the leftmost path, but have to make comparisons to decide which path to take.

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

Base Case for Black Height 0

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

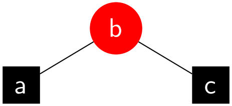
We remove the node if its key is equal to the given key a:

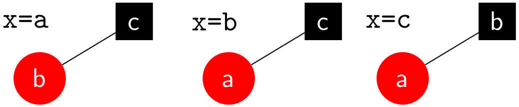
```
... | yes _ = , lf
```

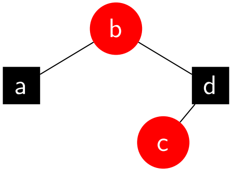
And leave it if it's not:

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

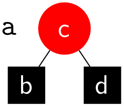
Base Cases for Black Height 1



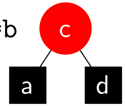




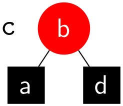
x=a



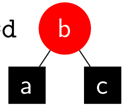
x=b

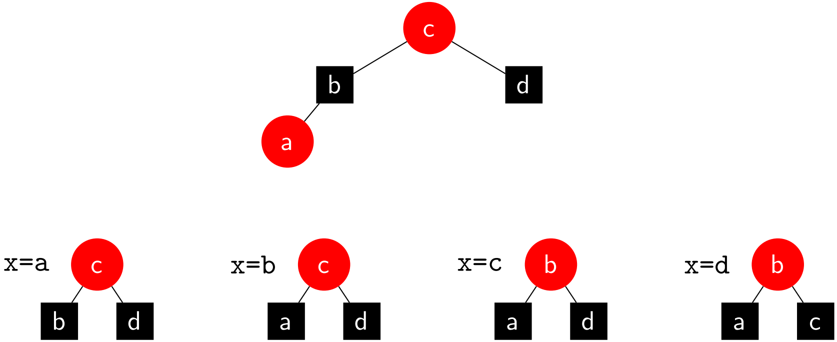


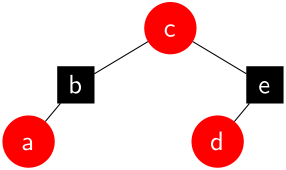
x=c

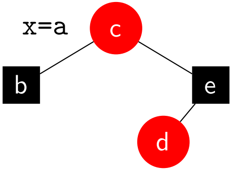


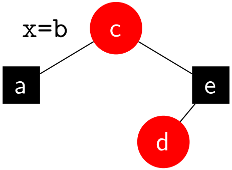
x=d

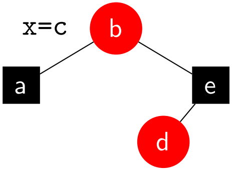


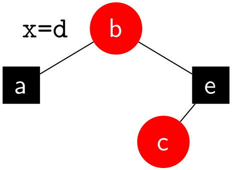

c
b
d
a
x=a
c
b
d
x=b
c
a
d
x=c
b
a
d
x=d
b
a
c


c
b
e
a
d

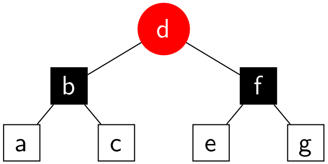

x=a
c
b
e
d


x=b
c
a
e
d

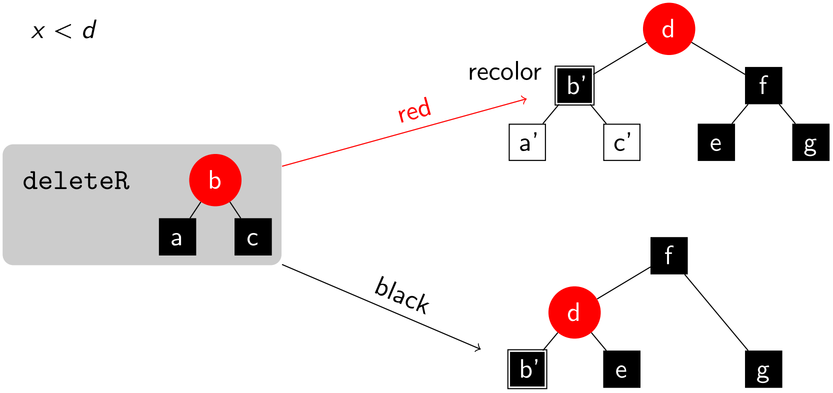

x=c
b
a
e
d


x=d
b
a
e
c

Recursive Cases



$x < d$


deleteR
b
a
c
red
recolor
d
b'
f
a'
c'
e
g
black
f
d
b'
e
g