

# Priority Search Trees

Peter Lammich      Tobias Nipkow

February 23, 2021

## Abstract

We present a new, purely functional, simple and efficient data structure combining a search tree and a priority queue, which we call a *priority search tree*. The salient feature of priority search trees is that they offer a decrease-key operation, something that is missing from other simple, purely functional priority queue implementations. Priority search trees can be implemented on top of any search tree. This entry does the implementation for red-black trees.

This entry formalizes the first part of our ITP-2019 proof pearl *Purely Functional, Simple and Efficient Priority Search Trees and Applications to Prim and Dijkstra* [2].

# Contents

<b>1</b>	<b>Priority Map Specifications</b>	<b>2</b>
1.1	Abstract Data Type . . . . .	2
1.2	Inorder-Based Specification . . . . .	3
<b>2</b>	<b>General Priority Search Trees</b>	<b>3</b>
<b>3</b>	<b>Priority Search Trees on top of RBTs</b>	<b>5</b>
3.1	Definitions . . . . .	6
3.1.1	The Code . . . . .	6
3.1.2	Invariants . . . . .	8
3.2	Functional Correctness . . . . .	8
3.3	Invariant Preservation . . . . .	9
3.3.1	Update . . . . .	10
3.3.2	Delete . . . . .	11
3.4	Overall Correctness . . . . .	12
<b>4</b>	<b>Related Work</b>	<b>13</b>

## 1 Priority Map Specifications

```
theory Prio-Map-Specs
imports HOL-Data-Structures.Map-Specs
begin
  <proof><proof>
```

### 1.1 Abstract Data Type

```
locale PrioMap = Map where lookup = lookup
  for lookup :: 'm ⇒ 'a ⇒ 'b::linorder option +
  fixes is-empty :: 'm ⇒ bool
  fixes getmin :: 'm ⇒ 'a × 'b
  assumes map-is-empty: invar m ⇒ is-empty m ⇔ lookup m = Map.empty
  and map-getmin: getmin m = (k,p) ⇒ invar m ⇒ lookup m ≠ Map.empty
  ⇒ lookup m k = Some p ∧ (∀ p' ∈ ran (lookup m). p ≤ p')
```

```
begin
```

```
lemmas prio-map-specs = map-specs map-is-empty
```

```
lemma map-getminE:
```

```
  assumes getmin m = (k,p) invar m lookup m ≠ Map.empty
  obtains lookup m k = Some p ∀ k' p'. lookup m k' = Some p' → p ≤ p'
  <proof>
```

```
end
```

**definition**  $is\_min2 :: (-\times 'a::linorder) \Rightarrow (-\times 'a) \text{ set} \Rightarrow \text{bool}$  **where**  
 $is\_min2\ x\ xs \equiv x \in xs \wedge (\forall y \in xs. \text{snd } x \leq \text{snd } y)$

## 1.2 Inorder-Based Specification

**locale**  $PrioMap\text{-by-Ordered} = Map\text{-by-Ordered}$   
**where**  $lookup = lookup$  **for**  $lookup :: 't \Rightarrow 'a::linorder \Rightarrow 'b::linorder\ option +$   
**fixes**  $is\_empty :: 't \Rightarrow \text{bool}$   
**fixes**  $getmin :: 't \Rightarrow 'a \times 'b$   
**assumes**  $inorder\text{-isempty}' : \llbracket inv\ t; sorted1\ (inorder\ t) \rrbracket$   
 $\implies is\_empty\ t \longleftrightarrow inorder\ t = []$   
**and**  $inorder\text{-getmin}' :$   
 $\llbracket inv\ t; sorted1\ (inorder\ t); inorder\ t \neq []; getmin\ t = (a,b) \rrbracket$   
 $\implies is\_min2\ (a,b)\ (set\ (inorder\ t))$   
**begin**

**lemma**  
 $inorder\text{-isempty} : inv\ t \implies is\_empty\ t \longleftrightarrow inorder\ t = []$   
**and**  $inorder\text{-getmin} : \llbracket inv\ t; inorder\ t \neq []; getmin\ t = (a,b) \rrbracket$   
 $\implies is\_min2\ (a,b)\ (set\ (inorder\ t))$   
 $\langle proof \rangle$

**lemma**  $inorder\text{-lookup-empty-iff} :$   
 $inv\ m \implies lookup\ m = Map.empty \longleftrightarrow inorder\ m = []$   
 $\langle proof \rangle$

**lemma**  $inorder\text{-lookup-ran-eq} :$   
 $\llbracket inv\ m; sorted1\ (inorder\ m) \rrbracket \implies ran\ (lookup\ m) = \text{snd } ' \text{ set } (inorder\ m)$   
 $\langle proof \rangle$

**sublocale**  $PrioMap\ empty\ update\ delete\ inv\ lookup\ is\_empty\ getmin$   
 $\langle proof \rangle$

**end**

**end**

## 2 General Priority Search Trees

**theory**  $PST\text{-General}$   
**imports**  
 $HOL\text{-Data-Structures.Tree2}$   
 $Prio\text{-Map-Specs}$   
**begin**

We show how to implement priority maps by augmented binary search trees. That is, the basic data structure is some arbitrary binary search tree, e.g. a red-black tree, implementing the map from  $'a$  to  $'b$  by storing pairs  $(k,p)$  in each node. At this point we need to assume that the keys are also linearly

ordered. To implement *getmin* efficiently we annotate/augment each node with another pair  $(k',p')$ , the intended result of *getmin* when applied to that subtree. The specification of *getmin* tells us that  $(k',p')$  must be in that subtree and that  $p'$  is the minimal priority in that subtree. Thus the annotation can be computed by passing the  $(k',p')$  with the minimal  $p'$  up the tree. We will now make this more precise for balanced binary trees in general.

We assume that our trees are either leaves of the form  $\langle \rangle$  or nodes of the form  $\langle l, (kp, b), r \rangle$  where  $l$  and  $r$  are subtrees,  $kp$  is the contents of the node (a key-priority pair) and  $b$  is some additional balance information (e.g. colour, height, size, ...). Augmented nodes are of the form  $\langle l, (kp, b, kp'), r \rangle$ .

**type-synonym**  $\langle 'k, 'p, 'c \rangle$  *pstree* =  $((('k \times 'p) \times ('c \times ('k \times 'p)))$  *tree*

The following invariant states that a node annotation is actually a minimal key-priority pair for the node's subtree.

**fun** *invpst* ::  $\langle 'k, 'p :: \text{linorder}, 'c \rangle$  *pstree*  $\Rightarrow$  *bool* **where**  
*invpst* *Leaf* = *True*  
| *invpst* (*Node*  $l$  ( $x$ ,  $-$ ,  $mkp$ )  $r$ )  $\longleftrightarrow$  *invpst*  $l \wedge$  *invpst*  $r$   
 $\wedge$  *is-min2*  $mkp$  (*set* (*inorder*  $l$  @  $x$  # *inorder*  $r$ ))

The implementation of *getmin* is trivial:

**fun** *pst-getmin* **where**  
*pst-getmin* (*Node*  $-$  ( $-$ ,  $-$ ,  $a$ )  $-$ ) =  $a$

**lemma** *pst-getmin-ismin*:

*invpst*  $t \Longrightarrow t \neq \text{Leaf} \Longrightarrow \text{is-min2}$  (*pst-getmin*  $t$ ) (*set-tree*  $t$ )  
 $\langle \text{proof} \rangle$

It remains to upgrade the existing map operations to work with augmented nodes. Therefore we now show how to transform any function definition on un-augmented trees into one on trees augmented with  $(k',p')$  pairs. A defining equation  $f$  *pats* =  $e$  for the original type of nodes is transformed into an equation  $f$  *pats'* =  $e'$  on the augmented type of nodes as follows:

- Every pattern  $\langle l, (kp, b), r \rangle$  in *pats* and  $e$  is replaced by  $\langle l, (kp, b, DUMMY), r \rangle$  to obtain *pats'* and  $e_2$ .
- To obtain  $e'$ , every expression  $\langle l, (kp, b), r \rangle$  in  $e_2$  is replaced by *mkNode*  $l$   $kp$   $b$   $r$  where:

**definition** *min2*  $\equiv \lambda(k,p) (k',p'). \text{if } p \leq p' \text{ then } (k,p) \text{ else } (k',p')$

**definition** *min-kp*  $a$   $l$   $r \equiv \text{case } (l,r) \text{ of}$   
 $(\text{Leaf}, \text{Leaf}) \Rightarrow a$   
|  $(\text{Leaf}, \text{Node } - (-, (-, kpr)) -) \Rightarrow \text{min2 } a$   $kpr$   
|  $(\text{Node } - (-, (-, kpl)) -, \text{Leaf}) \Rightarrow \text{min2 } a$   $kpl$

| (Node - (-, (-, kpl)) -, Node - (-, (-, kpr)) -)  $\Rightarrow$  min2 a (min2 kpl kpr)

**definition** mkNode c l a r  $\equiv$  Node l (a, (c, min-kp a l r)) r

Note that this transformation does not affect the asymptotic complexity of  $f$ . Therefore the priority search tree operations have the same complexity as the underlying search tree operations, i.e. typically logarithmic (*update*, *delete*, *lookup*) and constant time (*empty*, *is-empty*).

It is straightforward to show that *mkNode* preserves the invariant:

**lemma** *is-min2-Empty[simp]*:  $\neg$ is-min2 x {}  
 <proof>

**lemma** *is-min2-singleton[simp]*: is-min2 a {b}  $\longleftrightarrow$  b=a  
 <proof>

**lemma** *is-min2-insert*:  
 is-min2 x (insert y ys)  
 $\longleftrightarrow$  (y=x  $\wedge$  ( $\forall z \in$ ys. snd x  $\leq$  snd z))  $\vee$  (snd x  $\leq$  snd y  $\wedge$  is-min2 x ys)  
 <proof>

**lemma** *is-min2-union*:  
 is-min2 x (ys  $\cup$  zs)  
 $\longleftrightarrow$  (is-min2 x ys  $\wedge$  ( $\forall z \in$ zs. snd x  $\leq$  snd z))  
 $\vee$  (( $\forall y \in$ ys. snd x  $\leq$  snd y)  $\wedge$  is-min2 x zs)  
 <proof>

**lemma** *is-min2-min2-insI*: is-min2 y ys  $\Longrightarrow$  is-min2 (min2 x y) (insert x ys)  
 <proof>

**lemma** *is-min2-mergeI*:  
 is-min2 x xs  $\Longrightarrow$  is-min2 y ys  $\Longrightarrow$  is-min2 (min2 x y) (xs  $\cup$  ys)  
 <proof>

**theorem** *invpst-mkNode[simp]*: invpst (mkNode c l a r)  $\longleftrightarrow$  invpst l  $\wedge$  invpst r  
 <proof>

end

### 3 Priority Search Trees on top of RBTs

```
theory PST-RBT
imports
  HOL-Data-Structures.Cmp
  HOL-Data-Structures.Isin2
  HOL-Data-Structures.Lookup2
  PST-General
begin
```

We obtain a priority search map based on red-black trees via the general priority search tree augmentation.

This theory has been derived from the standard Isabelle implementation of red black trees in `HOL-Data_Structures`.

### 3.1 Definitions

#### 3.1.1 The Code

**datatype** *tcolor* = *Red* | *Black*

**type-synonym** ('k,'p) *rbth* = (('k×'p) × (tcolor × ('k × 'p))) *tree*

**abbreviation** *R* **where** *R mkp l a r* ≡ *Node l (a, Red, mkp) r*

**abbreviation** *B* **where** *B mkp l a r* ≡ *Node l (a, Black, mkp) r*

**abbreviation** *mkR* ≡ *mkNode Red*

**abbreviation** *mkB* ≡ *mkNode Black*

**fun** *baliL* :: ('k,'p::linorder) *rbth* ⇒ 'k×'p ⇒ ('k,'p) *rbth* ⇒ ('k,'p) *rbth*

**where**

*baliL* (*R* - (*R* - *t1 a1 t2*) *a2 t3*) *a3 t4* = *mkR* (*mkB* *t1 a1 t2*) *a2* (*mkB* *t3 a3 t4*) |  
 | *baliL* (*R* - *t1 a1* (*R* - *t2 a2 t3*)) *a3 t4* = *mkR* (*mkB* *t1 a1 t2*) *a2* (*mkB* *t3 a3 t4*) |  
 | *baliL* *t1 a t2* = *mkB* *t1 a t2*

**fun** *baliR* :: ('k,'p::linorder) *rbth* ⇒ 'k×'p ⇒ ('k,'p) *rbth* ⇒ ('k,'p) *rbth*

**where**

*baliR* *t1 a1* (*R* - (*R* - *t2 a2 t3*) *a3 t4*) = *mkR* (*mkB* *t1 a1 t2*) *a2* (*mkB* *t3 a3 t4*) |  
 | *baliR* *t1 a1* (*R* - *t2 a2* (*R* - *t3 a3 t4*)) = *mkR* (*mkB* *t1 a1 t2*) *a2* (*mkB* *t3 a3 t4*) |  
 | *baliR* *t1 a t2* = *mkB* *t1 a t2*

**fun** *paint* :: tcolor ⇒ ('k,'p::linorder) *rbth* ⇒ ('k,'p::linorder) *rbth* **where**

*paint* *c Leaf* = *Leaf* |

*paint* *c* (*Node l* (*a*, (-, *mkp*)) *r*) = *Node l* (*a*, (*c*, *mkp*)) *r*

**fun** *baldL* :: ('k,'p::linorder) *rbth* ⇒ 'k × 'p ⇒ ('k,'p::linorder) *rbth*

⇒ ('k,'p::linorder) *rbth*

**where**

*baldL* (*R* - *t1 x t2*) *y t3* = *mkR* (*mkB* *t1 x t2*) *y t3* |  
 | *baldL* *bl x* (*B* - *t1 y t2*) = *baliR* *bl x* (*mkR* *t1 y t2*) |  
 | *baldL* *bl x* (*R* - (*B* - *t1 y t2*) *z t3*)  
 = *mkR* (*mkB* *bl x t1*) *y* (*baliR* *t2 z* (*paint Red* *t3*)) |  
 | *baldL* *t1 x t2* = *mkR* *t1 x t2*

**fun** *baldR* :: ('k,'p::linorder) *rbth* ⇒ 'k × 'p ⇒ ('k,'p::linorder) *rbth*

⇒ ('k,'p::linorder) *rbth*

**where**

*baldR* *t1 x* (*R* - *t2 y t3*) = *mkR* *t1 x* (*mkB* *t2 y t3*) |  
 | *baldR* (*B* - *t1 x t2*) *y t3* = *baliL* (*mkR* *t1 x t2*) *y t3* |

```

baldR (R - t1 x (B - t2 y t3)) z t4
  = mkR (baliL (paint Red t1) x t2) y (mkB t3 z t4) |
baldR t1 x t2 = mkR t1 x t2

```

```

fun combine :: ('k,'p)::linorder) rbth => ('k,'p)::linorder) rbth
  => ('k,'p)::linorder) rbth

```

**where**

```

combine Leaf t = t |
combine t Leaf = t |
combine (R - t1 a t2) (R - t3 c t4) =
  (case combine t2 t3 of
    R - u2 b u3 => (mkR (mkR t1 a u2) b (mkR u3 c t4)) |
    t23 => mkR t1 a (mkR t23 c t4)) |
combine (B - t1 a t2) (B - t3 c t4) =
  (case combine t2 t3 of
    R - t2' b t3' => mkR (mkB t1 a t2') b (mkB t3' c t4) |
    t23 => baldL t1 a (mkB t23 c t4)) |
combine t1 (R - t2 a t3) = mkR (combine t1 t2) a t3 |
combine (R - t1 a t2) t3 = mkR t1 a (combine t2 t3)

```

```

fun color :: ('k,'p) rbth => tcolor where

```

```

color Leaf = Black |
color (Node - (-, (c,-)) -) = c

```

```

fun upd :: 'a::linorder => 'b::linorder => ('a,'b) rbth => ('a,'b) rbth where

```

```

upd x y Leaf = mkR Leaf (x,y) Leaf |
upd x y (B - l (a,b) r) = (case cmp x a of
  LT => baliL (upd x y l) (a,b) r |
  GT => baliR l (a,b) (upd x y r) |
  EQ => mkB l (x,y) r) |
upd x y (R - l (a,b) r) = (case cmp x a of
  LT => mkR (upd x y l) (a,b) r |
  GT => mkR l (a,b) (upd x y r) |
  EQ => mkR l (x,y) r)

```

```

definition update :: 'a::linorder => 'b::linorder => ('a,'b) rbth => ('a,'b) rbth

```

**where**

```

update x y t = paint Black (upd x y t)

```

```

fun del :: 'a::linorder => ('a,'b::linorder)rbth => ('a,'b)rbth where

```

```

del x Leaf = Leaf |
del x (Node l ((a,b), (c,-)) r) = (case cmp x a of
  LT => if l ≠ Leaf ∧ color l = Black
    then baldL (del x l) (a,b) r else mkR (del x l) (a,b) r |
  GT => if r ≠ Leaf ∧ color r = Black
    then baldR l (a,b) (del x r) else mkR l (a,b) (del x r) |
  EQ => combine l r)

```

**definition**  $delete :: 'a::linorder \Rightarrow ('a,'b::linorder) rbth \Rightarrow ('a,'b) rbth$  **where**  
 $delete\ x\ t = paint\ Black\ (del\ x\ t)$

### 3.1.2 Invariants

**fun**  $bheight :: ('k,'p) rbth \Rightarrow nat$  **where**  
 $bheight\ Leaf = 0$  |  
 $bheight\ (Node\ l\ (x,\ (c,-))\ r) = (if\ c = Black\ then\ bheight\ l + 1\ else\ bheight\ l)$

**fun**  $invc :: ('k,'p) rbth \Rightarrow bool$  **where**  
 $invc\ Leaf = True$  |  
 $invc\ (Node\ l\ (a,\ (c,-))\ r) =$   
 $(invc\ l \wedge invc\ r \wedge (c = Red \longrightarrow color\ l = Black \wedge color\ r = Black))$

**fun**  $invc2 :: ('k,'p) rbth \Rightarrow bool$  — Weaker version **where**  
 $invc2\ Leaf = True$  |  
 $invc2\ (Node\ l\ (a,\ -)\ r) = (invc\ l \wedge invc\ r)$

**fun**  $invh :: ('k,'p) rbth \Rightarrow bool$  **where**  
 $invh\ Leaf = True$  |  
 $invh\ (Node\ l\ (x,\ -)\ r) = (invh\ l \wedge invh\ r \wedge bheight\ l = bheight\ r)$

**definition**  $rbt :: ('k,'p::linorder) rbth \Rightarrow bool$  **where**  
 $rbt\ t = (invc\ t \wedge invh\ t \wedge invpst\ t \wedge color\ t = Black)$

## 3.2 Functional Correctness

**lemma**  $inorder-paint[simp]:\ inorder(paint\ c\ t) = inorder\ t$   
 $\langle proof \rangle$

**lemma**  $inorder-mkNode[simp]:$   
 $inorder\ (mkNode\ c\ l\ a\ r) = inorder\ l\ @\ a\ \#\ inorder\ r$   
 $\langle proof \rangle$

**lemma**  $inorder-baliL[simp]:$   
 $inorder(baliL\ l\ a\ r) = inorder\ l\ @\ a\ \#\ inorder\ r$   
 $\langle proof \rangle$

**lemma**  $inorder-baliR[simp]:$   
 $inorder(baliR\ l\ a\ r) = inorder\ l\ @\ a\ \#\ inorder\ r$   
 $\langle proof \rangle$

**lemma**  $inorder-baldL[simp]:$   
 $inorder(baldL\ l\ a\ r) = inorder\ l\ @\ a\ \#\ inorder\ r$   
 $\langle proof \rangle$

**lemma**  $inorder-baldR[simp]:$

$inorder(\text{baldR } l \ a \ r) = inorder \ l \ @ \ a \ \# \ inorder \ r$   
(proof)

**lemma** *inorder-combine*[simp]:  
 $inorder(\text{combine } l \ r) = inorder \ l \ @ \ inorder \ r$   
(proof)

**lemma** *inorder-upd*:  
 $sorted1(inorder \ t) \implies inorder(\text{upd } x \ y \ t) = \text{upd-list } x \ y \ (inorder \ t)$   
(proof)

**lemma** *inorder-update*:  
 $sorted1(inorder \ t) \implies inorder(\text{update } x \ y \ t) = \text{upd-list } x \ y \ (inorder \ t)$   
(proof)

**lemma** *inorder-del*:  
 $sorted1(inorder \ t) \implies inorder(\text{del } x \ t) = \text{del-list } x \ (inorder \ t)$   
(proof)

**lemma** *inorder-delete*:  
 $sorted1(inorder \ t) \implies inorder(\text{delete } x \ t) = \text{del-list } x \ (inorder \ t)$   
(proof)

### 3.3 Invariant Preservation

**lemma** *color-paint-Black*:  $color \ (\text{paint } Black \ t) = Black$   
(proof)

**theorem** *rbt-Leaf*:  $rbt \ Leaf$   
(proof)

**lemma** *invc2I*:  $invc \ t \implies invc2 \ t$   
(proof)

**lemma** *paint-invc2*:  $invc2 \ t \implies invc2 \ (\text{paint } c \ t)$   
(proof)

**lemma** *invc-paint-Black*:  $invc2 \ t \implies invc \ (\text{paint } Black \ t)$   
(proof)

**lemma** *invh-paint*:  $invh \ t \implies invh \ (\text{paint } c \ t)$   
(proof)

**lemma** *invc-mkRB*[simp]:  
 $invc \ (\text{mkR } l \ a \ r) \longleftrightarrow invc \ l \ \wedge \ invc \ r \ \wedge \ color \ l = Black \ \wedge \ color \ r = Black$   
 $invc \ (\text{mkB } l \ a \ r) \longleftrightarrow invc \ l \ \wedge \ invc \ r$   
(proof)

**lemma** *color-mkNode*[simp]:  $color \ (\text{mkNode } c \ l \ a \ r) = c$

$\langle proof \rangle$

### 3.3.1 Update

**lemma** *invc-baliL*:

$\llbracket invc2\ l; invc\ r \rrbracket \implies invc\ (baliL\ l\ a\ r)$   
 $\langle proof \rangle$

**lemma** *invc-baliR*:

$\llbracket invc\ l; invc2\ r \rrbracket \implies invc\ (baliR\ l\ a\ r)$   
 $\langle proof \rangle$

**lemma** *bheight-mkRB[simp]*:

$bheight\ (mkR\ l\ a\ r) = bheight\ l$   
 $bheight\ (mkB\ l\ a\ r) = Suc\ (bheight\ l)$   
 $\langle proof \rangle$

**lemma** *bheight-baliL*:

$bheight\ l = bheight\ r \implies bheight\ (baliL\ l\ a\ r) = Suc\ (bheight\ l)$   
 $\langle proof \rangle$

**lemma** *bheight-baliR*:

$bheight\ l = bheight\ r \implies bheight\ (baliR\ l\ a\ r) = Suc\ (bheight\ l)$   
 $\langle proof \rangle$

**lemma** *invh-mkNode[simp]*:

$invh\ (mkNode\ c\ l\ a\ r) \longleftrightarrow invh\ l \wedge invh\ r \wedge bheight\ l = bheight\ r$   
 $\langle proof \rangle$

**lemma** *invh-baliL*:

$\llbracket invh\ l; invh\ r; bheight\ l = bheight\ r \rrbracket \implies invh\ (baliL\ l\ a\ r)$   
 $\langle proof \rangle$

**lemma** *invh-baliR*:

$\llbracket invh\ l; invh\ r; bheight\ l = bheight\ r \rrbracket \implies invh\ (baliR\ l\ a\ r)$   
 $\langle proof \rangle$

**lemma** *invc-upd*: **assumes** *invc t*

**shows**  $color\ t = Black \implies invc\ (upd\ x\ y\ t)\ invc2\ (upd\ x\ y\ t)$   
 $\langle proof \rangle$

**lemma** *invh-upd*: **assumes** *invh t*

**shows**  $invh\ (upd\ x\ y\ t)\ bheight\ (upd\ x\ y\ t) = bheight\ t$   
 $\langle proof \rangle$

**lemma** *invpst-paint[simp]*:  $invpst\ (paint\ c\ t) = invpst\ t$

$\langle proof \rangle$

**lemma** *invpst-baliR*:  $\text{invpst } l \implies \text{invpst } r \implies \text{invpst } (\text{baliR } l \ a \ r)$   
 ⟨proof⟩

**lemma** *invpst-baliL*:  $\text{invpst } l \implies \text{invpst } r \implies \text{invpst } (\text{baliL } l \ a \ r)$   
 ⟨proof⟩

**lemma** *invpst-upd*:  $\text{invpst } t \implies \text{invpst } (\text{upd } x \ y \ t)$   
 ⟨proof⟩

**theorem** *rbt-update*:  $\text{rbt } t \implies \text{rbt } (\text{update } x \ y \ t)$   
 ⟨proof⟩

### 3.3.2 Delete

**lemma** *bheight-paint-Red*:  
 $\text{color } t = \text{Black} \implies \text{bheight } (\text{paint } \text{Red } t) = \text{bheight } t - 1$   
 ⟨proof⟩

**lemma** *invh-baldL-invc*:  
 $\llbracket \text{invh } l; \text{invh } r; \text{bheight } l + 1 = \text{bheight } r; \text{invc } r \rrbracket$   
 $\implies \text{invh } (\text{baldL } l \ a \ r) \wedge \text{bheight } (\text{baldL } l \ a \ r) = \text{bheight } l + 1$   
 ⟨proof⟩

**lemma** *invh-baldL-Black*:  
 $\llbracket \text{invh } l; \text{invh } r; \text{bheight } l + 1 = \text{bheight } r; \text{color } r = \text{Black} \rrbracket$   
 $\implies \text{invh } (\text{baldL } l \ a \ r) \wedge \text{bheight } (\text{baldL } l \ a \ r) = \text{bheight } r$   
 ⟨proof⟩

**lemma** *invc-baldL*:  $\llbracket \text{invc2 } l; \text{invc } r; \text{color } r = \text{Black} \rrbracket \implies \text{invc } (\text{baldL } l \ a \ r)$   
 ⟨proof⟩

**lemma** *invc2-baldL*:  $\llbracket \text{invc2 } l; \text{invc } r \rrbracket \implies \text{invc2 } (\text{baldL } l \ a \ r)$   
 ⟨proof⟩

**lemma** *invh-baldR-invc*:  
 $\llbracket \text{invh } l; \text{invh } r; \text{bheight } l = \text{bheight } r + 1; \text{invc } l \rrbracket$   
 $\implies \text{invh } (\text{baldR } l \ a \ r) \wedge \text{bheight } (\text{baldR } l \ a \ r) = \text{bheight } l$   
 ⟨proof⟩

**lemma** *invc-baldR*:  $\llbracket \text{invc } a; \text{invc2 } b; \text{color } a = \text{Black} \rrbracket \implies \text{invc } (\text{baldR } a \ x \ b)$   
 ⟨proof⟩

**lemma** *invc2-baldR*:  $\llbracket \text{invc } l; \text{invc2 } r \rrbracket \implies \text{invc2 } (\text{baldR } l \ x \ r)$   
 ⟨proof⟩

**lemma** *invh-combine*:  
 $\llbracket \text{invh } l; \text{invh } r; \text{bheight } l = \text{bheight } r \rrbracket$

$\implies \text{invh} (\text{combine } l \ r) \wedge \text{bheight} (\text{combine } l \ r) = \text{bheight } l$   
 $\langle \text{proof} \rangle$

**lemma** *invc-combine*:

**assumes** *invc l invc r*

**shows**  $\text{color } l = \text{Black} \implies \text{color } r = \text{Black} \implies \text{invc} (\text{combine } l \ r)$   
 $\text{invc2} (\text{combine } l \ r)$

$\langle \text{proof} \rangle$

**lemma** *neg-LeafD*:  $t \neq \text{Leaf} \implies \exists l \ x \ c \ r. t = \text{Node } l \ (x, c) \ r$

$\langle \text{proof} \rangle$

**lemma** *del-invc-invh*:  $\text{invh } t \implies \text{invc } t \implies \text{invh} (\text{del } x \ t) \wedge$

$(\text{color } t = \text{Red} \wedge \text{bheight} (\text{del } x \ t) = \text{bheight } t \wedge \text{invc} (\text{del } x \ t) \vee$

$\text{color } t = \text{Black} \wedge \text{bheight} (\text{del } x \ t) = \text{bheight } t - 1 \wedge \text{invc2} (\text{del } x \ t))$

$\langle \text{proof} \rangle$

**lemma** *invpst-baldR*:  $\text{invpst } l \implies \text{invpst } r \implies \text{invpst} (\text{baldR } l \ a \ r)$

$\langle \text{proof} \rangle$

**lemma** *invpst-baldL*:  $\text{invpst } l \implies \text{invpst } r \implies \text{invpst} (\text{baldL } l \ a \ r)$

$\langle \text{proof} \rangle$

**lemma** *invpst-combine*:  $\text{invpst } l \implies \text{invpst } r \implies \text{invpst} (\text{combine } l \ r)$

$\langle \text{proof} \rangle$

**lemma** *invpst-del*:  $\text{invpst } t \implies \text{invpst} (\text{del } x \ t)$

$\langle \text{proof} \rangle$

**theorem** *rbt-delete*:  $\text{rbt } t \implies \text{rbt} (\text{delete } k \ t)$

$\langle \text{proof} \rangle$

**lemma** *rbt-getmin-ismin*:

$\text{rbt } t \implies t \neq \text{Leaf} \implies \text{is-min2} (\text{pst-getmin } t) (\text{set-tree } t)$

$\langle \text{proof} \rangle$

**definition** *rbt-is-empty*  $t \equiv t = \text{Leaf}$

**lemma** *rbt-is-empty*:  $\text{rbt-is-empty } t \longleftrightarrow \text{inorder } t = []$

$\langle \text{proof} \rangle$

**definition** *empty where*  $\text{empty} = \text{Leaf}$

### 3.4 Overall Correctness

**interpretation** *PM*: *PrioMap-by-Ordered*

**where**  $\text{empty} = \text{empty}$  **and**  $\text{lookup} = \text{lookup}$  **and**  $\text{update} = \text{update}$  **and**  $\text{delete} = \text{delete}$

**and**  $\text{inorder} = \text{inorder}$  **and**  $\text{inv} = \text{rbt}$  **and**  $\text{is-empty} = \text{rbt-is-empty}$

```
and getmin = pst-getmin  
⟨proof⟩
```

```
end
```

## 4 Related Work

Our priority map ADT is close to Hinze’s [1] *priority search queue* interface, except that he also supports a few further operations that we could easily add but do not need for our applications. However, it is not clear if his implementation technique is the same as our priority search tree because his description employs a plethora of concepts, e.g. *priority search pennants*, *tournament trees*, *semi-heaps*, and multiple *views* of data types that obscure a direct comparison. We claim that at the very least our presentation is new because it is much simpler; we encourage the reader to compare the two.

As already observed by Hinze, McCreight’s [3] priority search trees support range queries more efficiently than our trees. However, we can support the same range queries as Hinze efficiently, but that is outside the scope of this entry.

## References

- [1] R. Hinze. A simple implementation technique for priority search queues. In B. C. Pierce, editor, *Proceedings of the Sixth ACM SIGPLAN International Conference on Functional Programming (ICFP ’01), Firenze (Florence), Italy, September 3-5, 2001.*, pages 110–121. ACM, 2001.
- [2] P. Lammich and T. Nipkow. Proof pearl: Purely functional, simple and efficient Priority Search Trees and applications to Prim and Dijkstra. In *Proc. of ITP*, 2019. to appear.
- [3] E. M. McCreight. Priority search trees. *SIAM J. Comput.*, 14(2):257–276, 1985.