Priority Search Trees

Peter Lammich   Tobias Nipkow

March 17, 2025

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

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* :: $'m \Rightarrow 'a \Rightarrow 'b$::*linorder option* +
  fixes *is-empty* :: $'m \Rightarrow$ *bool*
  fixes *getmin* :: $'m \Rightarrow 'a \times 'b$
  assumes *map-is-empty*: *invar m* $\Longrightarrow$ *is-empty m* $\longleftrightarrow$ *lookup m* = *Map.empty*
  and *map-getmin*: *getmin m* = (k,p) $\Longrightarrow$ *invar m* $\Longrightarrow$ *lookup m* $\neq$ *Map.empty*
  $\Longrightarrow$ *lookup m k* = *Some p* $\wedge$ ($\forall p' \in$ *ran* (*lookup m*). $p \le p'$)
begin

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

lemma *map-getminE*:
  assumes *getmin m* = (k,p) *invar m lookup m* $\neq$ *Map.empty*
  obtains *lookup m k* = *Some p* $\forall k'\ p'$. *lookup m k'* = *Some p'* $\longrightarrow p \le p'$
⟨*proof*⟩

end

definition *is-min2* :: $(\text{-}\times'a::linorder) \Rightarrow (\text{-}\times'a)$ *set* $\Rightarrow$ *bool* **where**
is-min2 x xs $\equiv$ $x \in xs \land (\forall y \in xs.\ snd\ x \leq snd\ y)$

1.2 Inorder-Based Specification

locale *PrioMap-by-Ordered* = *Map-by-Ordered*
  where *lookup=lookup* **for** *lookup* :: $'t \Rightarrow 'a::linorder \Rightarrow 'b::linorder$ *option* +
  fixes *is-empty* :: $'t \Rightarrow$ *bool*
  fixes *getmin* :: $'t \Rightarrow 'a\times'b$
  assumes *inorder-isempty'*: $[\![$*inv t*; *sorted1* (*inorder t*)$]\!]$
    $\Longrightarrow$ *is-empty t* $\longleftrightarrow$ *inorder t* = []
  and *inorder-getmin'*:
    $[\![$*inv t*; *sorted1* (*inorder t*); *inorder t* $\neq$ []; *getmin t* = $(a,b)]\!]$
      $\Longrightarrow$ *is-min2* (a,b) (*set* (*inorder t*))
begin

lemma
  inorder-isempty: *invar t* $\Longrightarrow$ *is-empty t* $\longleftrightarrow$ *inorder t* = []
  and *inorder-getmin*: $[\![$*invar t*; *inorder t* $\neq$ []; *getmin t* = $(a,b)]\!]$
      $\Longrightarrow$ *is-min2* (a,b) (*set* (*inorder t*))
⟨*proof*⟩

lemma *inorder-lookup-empty-iff*:
  invar m $\Longrightarrow$ *lookup m* = *Map.empty* $\longleftrightarrow$ *inorder m* = []
⟨*proof*⟩

lemma *inorder-lookup-ran-eq*:
  $[\![$*inv m*; *sorted1* (*inorder m*)$]\!]$ $\Longrightarrow$ *ran* (*lookup m*) = *snd* ‘ *set* (*inorder m*)
⟨*proof*⟩

sublocale *PrioMap empty update delete invar lookup is-empty getmin*
⟨*proof*⟩

end

end

2 General Priority Search Trees

theory *PST-General*
imports
  HOL−Data-Structures.Tree2
  Prio-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 $('k,'p,'c)\ pstree = (('k \times 'p) \times ('c \times ('k \times 'p)))\ tree$

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

fun *invpst* :: $('k,'p{::}linorder,'c)\ pstree \Rightarrow bool$ **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*$\neq$*Leaf* $\Longrightarrow$ *is-min2* (*pst-getmin t*) (*set-tree t*)
$\langle$*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* $\equiv \lambda(k,p)\ (k',p').$ *if* $p \le p'$ *then* (k,p) *else* (k',p')

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

| (*Node* - (-, (-,*kpl*)) -,*Leaf*) ⇒ *min2 a kpl*
| (*Node* - (-, (-,*kpl*)) -,*Node* - (-, (-,*kpr*)) -) ⇒ *min2 a* (*min2 kpl kpr*)

definition *mkNode c l a r* ≡ *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*]: ¬*is-min2 x* {}
⟨*proof*⟩

lemma *is-min2-singleton*[*simp*]: *is-min2 a* {*b*} ⟷ *b*=*a*
⟨*proof*⟩

lemma *is-min2-insert*:
  is-min2 x (*insert y ys*)
  ⟷ (*y*=*x* ∧ (∀ *z*∈*ys*. *snd x* ≤ *snd z*)) ∨ (*snd x* ≤ *snd y* ∧ *is-min2 x ys*)
⟨*proof*⟩

lemma *is-min2-union*:
  is-min2 x (*ys* ∪ *zs*)
  ⟷ (*is-min2 x ys* ∧ (∀ *z*∈*zs*. *snd x* ≤ *snd z*))
    ∨ ((∀ *y*∈*ys*. *snd x* ≤ *snd y*) ∧ *is-min2 x zs*)
⟨*proof*⟩

lemma *is-min2-min2-insI*: *is-min2 y ys* ⟹ *is-min2* (*min2 x y*) (*insert x ys*)
⟨*proof*⟩

lemma *is-min2-mergeI*:
  is-min2 x xs ⟹ *is-min2 y ys* ⟹ *is-min2* (*min2 x y*) (*xs* ∪ *ys*)
⟨*proof*⟩

theorem *invpst-mkNode*[*simp*]: *invpst* (*mkNode c l a r*) ⟷ *invpst l* ∧ *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* ⇒ (*'a,'b::linorder*) *rbth* ⇒ (*'a,'b*) *rbth* **where**
delete x t = *paint Black* (*del x t*)

3.1.2 Invariants

fun *bheight* :: (*'k,'p*) *rbth* ⇒ *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* ⇒ *bool* **where**
invc Leaf = *True* |
invc (*Node l* (*a*, (*c*,-)) *r*) =
(*invc l* ∧ *invc r* ∧ (*c* = *Red* ⟶ *color l* = *Black* ∧ *color r* = *Black*))

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

fun *invh* :: (*'k,'p*) *rbth* ⇒ *bool* **where**
invh Leaf = *True* |
invh (*Node l* (*x*, -) *r*) = (*invh l* ∧ *invh r* ∧ *bheight l* = *bheight r*)

definition *rbt* :: (*'k,'p::linorder*) *rbth* ⇒ *bool* **where**
rbt t = (*invc t* ∧ *invh t* ∧ *invpst t* ∧ *color t* = *Black*)

3.2 Functional Correctness

lemma *inorder-paint*[*simp*]: *inorder*(*paint c t*) = *inorder t*
⟨*proof*⟩

lemma *inorder-mkNode*[*simp*]:
inorder (*mkNode c l a r*) = *inorder l* @ *a* # *inorder r*
⟨*proof*⟩

lemma *inorder-baliL*[*simp*]:
inorder(*baliL l a r*) = *inorder l* @ *a* # *inorder r*
⟨*proof*⟩

lemma *inorder-baliR*[*simp*]:
inorder(*baliR l a r*) = *inorder l* @ *a* # *inorder r*
⟨*proof*⟩

lemma *inorder-baldL*[*simp*]:
inorder(*baldL l a r*) = *inorder l* @ *a* # *inorder r*
⟨*proof*⟩

lemma *inorder-baldR*[*simp*]:
$inorder(baldR\ l\ a\ r) = inorder\ l\ @\ a\ \#\ inorder\ r$
⟨*proof*⟩

lemma *inorder-combine*[*simp*]:
$inorder(combine\ l\ r) = inorder\ l\ @\ inorder\ r$
⟨*proof*⟩

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

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

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

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

3.3 Invariant Preservation

lemma *color-paint-Black*: $color\ (paint\ Black\ t) = Black$
⟨*proof*⟩

theorem *rbt-Leaf*: $rbt\ Leaf$
⟨*proof*⟩

lemma *invc2I*: $invc\ t \Longrightarrow invc2\ t$
⟨*proof*⟩

lemma *paint-invc2*: $invc2\ t \Longrightarrow invc2\ (paint\ c\ t)$
⟨*proof*⟩

lemma *invc-paint-Black*: $invc2\ t \Longrightarrow invc\ (paint\ Black\ t)$
⟨*proof*⟩

lemma *invh-paint*: $invh\ t \Longrightarrow invh\ (paint\ c\ t)$
⟨*proof*⟩

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

lemma *color-mkNode*[*simp*]: *color* (*mkNode c l a r*) = *c*
⟨*proof*⟩

3.3.1 Update

lemma *invc-baliL*:
⟦*invc2 l*; *invc r*⟧ ⟹ *invc* (*baliL l a r*)
⟨*proof*⟩

lemma *invc-baliR*:
⟦*invc l*; *invc2 r*⟧ ⟹ *invc* (*baliR l a r*)
⟨*proof*⟩

lemma *bheight-mkRB*[*simp*]:
bheight (*mkR l a r*) = *bheight l*
bheight (*mkB l a r*) = *Suc* (*bheight l*)
⟨*proof*⟩

lemma *bheight-baliL*:
bheight l = *bheight r* ⟹ *bheight* (*baliL l a r*) = *Suc* (*bheight l*)
⟨*proof*⟩

lemma *bheight-baliR*:
bheight l = *bheight r* ⟹ *bheight* (*baliR l a r*) = *Suc* (*bheight l*)
⟨*proof*⟩

lemma *invh-mkNode*[*simp*]:
invh (*mkNode c l a r*) ⟷ *invh l* ∧ *invh r* ∧ *bheight l* = *bheight r*
⟨*proof*⟩

lemma *invh-baliL*:
⟦ *invh l*; *invh r*; *bheight l* = *bheight r* ⟧ ⟹ *invh* (*baliL l a r*)
⟨*proof*⟩

lemma *invh-baliR*:
⟦ *invh l*; *invh r*; *bheight l* = *bheight r* ⟧ ⟹ *invh* (*baliR l a r*)
⟨*proof*⟩

lemma *invc-upd*: **assumes** *invc t*
shows *color t* = *Black* ⟹ *invc* (*upd x y t*) *invc2* (*upd x y t*)
⟨*proof*⟩

lemma *invh-upd*: **assumes** *invh t*
shows *invh* (*upd x y t*) *bheight* (*upd x y t*) = *bheight t*
⟨*proof*⟩

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

lemma *invpst-baliR*: $invpst\ l \Longrightarrow invpst\ r \Longrightarrow invpst\ (baliR\ l\ a\ r)$
⟨*proof*⟩

lemma *invpst-baliL*: $invpst\ l \Longrightarrow invpst\ r \Longrightarrow invpst\ (baliL\ l\ a\ r)$
⟨*proof*⟩

lemma *invpst-upd*: $invpst\ t \Longrightarrow invpst\ (upd\ x\ y\ t)$
⟨*proof*⟩

theorem *rbt-update*: $rbt\ t \Longrightarrow rbt\ (update\ x\ y\ t)$
⟨*proof*⟩

3.3.2 Delete

lemma *bheight-paint-Red*:
$color\ t = Black \Longrightarrow bheight\ (paint\ Red\ t) = bheight\ t - 1$
⟨*proof*⟩

lemma *invh-baldL-invc*:
$[\![\ invh\ l;\ \ invh\ r;\ \ bheight\ l + 1 = bheight\ r;\ \ invc\ r\ ]\!]$
$\Longrightarrow invh\ (baldL\ l\ a\ r) \land bheight\ (baldL\ l\ a\ r) = bheight\ l + 1$
⟨*proof*⟩

lemma *invh-baldL-Black*:
$[\![\ invh\ l;\ \ invh\ r;\ \ bheight\ l + 1 = bheight\ r;\ \ color\ r = Black\ ]\!]$
$\Longrightarrow invh\ (baldL\ l\ a\ r) \land bheight\ (baldL\ l\ a\ r) = bheight\ r$
⟨*proof*⟩

lemma *invc-baldL*: $[\![invc2\ l;\ invc\ r;\ color\ r = Black]\!] \Longrightarrow invc\ (baldL\ l\ a\ r)$
⟨*proof*⟩

lemma *invc2-baldL*: $[\![\ invc2\ l;\ invc\ r\ ]\!] \Longrightarrow invc2\ (baldL\ l\ a\ r)$
⟨*proof*⟩

lemma *invh-baldR-invc*:
$[\![\ invh\ l;\ \ invh\ r;\ \ bheight\ l = bheight\ r + 1;\ \ invc\ l\ ]\!]$
$\Longrightarrow invh\ (baldR\ l\ a\ r) \land bheight\ (baldR\ l\ a\ r) = bheight\ l$
⟨*proof*⟩

lemma *invc-baldR*: $[\![invc\ a;\ invc2\ b;\ color\ a = Black]\!] \Longrightarrow invc\ (baldR\ a\ x\ b)$
⟨*proof*⟩

lemma *invc2-baldR*: $[\![\ invc\ l;\ invc2\ r\ ]\!] \Longrightarrow invc2\ (baldR\ l\ x\ r)$
⟨*proof*⟩

lemma *invh-combine*:
$[\![$ *invh l*; *invh r*; *bheight l* = *bheight r* $]\!]$
$\Longrightarrow$ *invh* (*combine l r*) $\wedge$ *bheight* (*combine l r*) = *bheight l*
⟨*proof*⟩

lemma *invc-combine*:
assumes *invc l invc r*
shows *color l* = *Black* $\Longrightarrow$ *color r* = *Black* $\Longrightarrow$ *invc* (*combine l r*)
invc2 (*combine l r*)
⟨*proof*⟩

lemma *neq-LeafD*: *t* ≠ *Leaf* $\Longrightarrow$ ∃ *l x c r*. *t* = *Node l* (*x*,*c*) *r*
⟨*proof*⟩

lemma *del-invc-invh*: *invh t* $\Longrightarrow$ *invc t* $\Longrightarrow$ *invh* (*del x t*) $\wedge$
(*color t* = *Red* $\wedge$ *bheight* (*del x t*) = *bheight t* $\wedge$ *invc* (*del x t*) $\vee$
color t = *Black* $\wedge$ *bheight* (*del x t*) = *bheight t* − *1* $\wedge$ *invc2* (*del x t*))
⟨*proof*⟩

lemma *invpst-baldR*: *invpst l* $\Longrightarrow$ *invpst r* $\Longrightarrow$ *invpst* (*baldR l a r*)
⟨*proof*⟩

lemma *invpst-baldL*: *invpst l* $\Longrightarrow$ *invpst r* $\Longrightarrow$ *invpst* (*baldL l a r*)
⟨*proof*⟩

lemma *invpst-combine*: *invpst l* $\Longrightarrow$ *invpst r* $\Longrightarrow$ *invpst* (*combine l r*)
⟨*proof*⟩

lemma *invpst-del*: *invpst t* $\Longrightarrow$ *invpst* (*del x t*)
⟨*proof*⟩

theorem *rbt-delete*: *rbt t* $\Longrightarrow$ *rbt* (*delete k t*)
⟨*proof*⟩

lemma *rbt-getmin-ismin*:
rbt t $\Longrightarrow$ *t*≠*Leaf* $\Longrightarrow$ *is-min2* (*pst-getmin t*) (*set-tree t*)
⟨*proof*⟩

definition *rbt-is-empty t* ≡ *t* = *Leaf*

lemma *rbt-is-empty*: *rbt-is-empty t* ⟷ *inorder t* = []
⟨*proof*⟩

definition *empty* **where** *empty* = *Leaf*

3.4 Overall Correctness

interpretation *PM*: *PrioMap-by-Ordered*
where *empty* = *empty* **and** *lookup* = *lookup* **and** *update* = *update* **and** *delete* =

delete
and *inorder* = *inorder* **and** *inv* = *rbt* **and** *is-empty* = *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.