

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

1	Priority Map Specifications	2
1.1	Abstract Data Type	2
1.2	Inorder-Based Specification	3
2	General Priority Search Trees	4
3	Priority Search Trees on top of RBTs	6
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	14
4	Related Work	14

1 Priority Map Specifications

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

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'
using map-getmin[OF assms] by (auto simp: ran-def)
```

```
end
```

definition *is-min2* :: $(-\times'a::\text{linorder}) \Rightarrow (-\times'a) \text{ set} \Rightarrow \text{bool}$ **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-by-Ordered* = *Map-by-Ordered*
where *lookup*=*lookup* **for** *lookup* :: $'t \Rightarrow 'a::\text{linorder} \Rightarrow 'b::\text{linorder} \text{ option} +$
fixes *is-empty* :: $'t \Rightarrow \text{bool}$
fixes *getmin* :: $'t \Rightarrow 'a \times 'b$
assumes *inorder-isempty'*: $\llbracket \text{inv } t; \text{sorted1 } (\text{inorder } t) \rrbracket$
 $\implies \text{is-empty } t \longleftrightarrow \text{inorder } t = []$
and *inorder-getmin'*:
 $\llbracket \text{inv } t; \text{sorted1 } (\text{inorder } t); \text{inorder } t \neq []; \text{getmin } t = (a,b) \rrbracket$
 $\implies \text{is-min2 } (a,b) (\text{set } (\text{inorder } t))$
begin

lemma

inorder-isempty: $\text{invar } t \implies \text{is-empty } t \longleftrightarrow \text{inorder } t = []$
and *inorder-getmin*: $\llbracket \text{invar } t; \text{inorder } t \neq []; \text{getmin } t = (a,b) \rrbracket$
 $\implies \text{is-min2 } (a,b) (\text{set } (\text{inorder } t))$
unfolding *invar-def* **by** (*auto simp: inorder-isempty' inorder-getmin'*)

lemma *inorder-lookup-empty-iff*:

invar m $\implies \text{lookup } m = \text{Map.empty} \longleftrightarrow \text{inorder } m = []$
using *inorder-lookup[of m]*
apply (*auto split: if-splits simp: invar-def*)
by (*metis map-of.elims option.discI*)

lemma *inorder-lookup-ran-eq*:

$\llbracket \text{inv } m; \text{sorted1 } (\text{inorder } m) \rrbracket \implies \text{ran } (\text{lookup } m) = \text{snd } ` \text{set } (\text{inorder } m)$
using *inorder-lookup[of m]* **unfolding** *ran-def*
by (*force simp: sorted-map-of-Some-eq*)

sublocale *PrioMap* *empty update delete invar lookup is-empty getmin*

apply *unfold-locales*

apply (*auto simp: inorder-isempty inorder-lookup-empty-iff*)

apply (*frule* (2) *inorder-getmin*)

apply (*auto simp: is-min2-def sorted-map-of-Some-eq invar-def inorder-lookup*) []

apply (*frule* (2) *inorder-getmin*)

apply (*force simp: is-min2-def sorted-map-of-Some-eq inorder-lookup-ran-eq*
eq-Min-iff invar-def inorder-lookup) []

done

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 × 'p) × ('c × ('k × '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 ⇒ bool where
  invpst Leaf = True
| invpst (Node l (x, -, mkp) r) ⇔ invpst l ∧ invpst r
  ∧ 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 ⇒ t ≠ Leaf ⇒ is-min2 (pst-getmin t) (set-tree t)
by (cases t rule: pst-getmin.cases) auto

```

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 $min-kp\ a\ l\ r \equiv \text{case } (l,r) \text{ of}$

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

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 $is-min2-Empty[simp]: \neg is-min2\ x\ \{\}$

by ($auto\ simp: is-min2-def$)

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

by ($auto\ simp: is-min2-def$)

lemma $is-min2-insert:$

$is-min2\ x\ (insert\ y\ ys)$

$\longleftrightarrow (y=x \wedge (\forall z \in ys. snd\ x \leq snd\ z)) \vee (snd\ x \leq snd\ y \wedge is-min2\ x\ ys)$

by ($auto\ simp: is-min2-def$)

lemma $is-min2-union:$

$is-min2\ x\ (ys \cup zs)$

$\longleftrightarrow (is-min2\ x\ ys \wedge (\forall z \in zs. snd\ x \leq snd\ z))$

$\vee ((\forall y \in ys. snd\ x \leq snd\ y) \wedge is-min2\ x\ zs)$

by ($auto\ simp: is-min2-def$)

lemma $is-min2-min2-insI: is-min2\ y\ ys \Longrightarrow is-min2\ (min2\ x\ y)\ (insert\ x\ ys)$

by ($auto\ simp: is-min2-def\ min2-def\ split: prod.split$)

lemma $is-min2-mergeI:$

$is-min2\ x\ xs \Longrightarrow is-min2\ y\ ys \Longrightarrow is-min2\ (min2\ x\ y)\ (xs \cup ys)$

by ($auto\ simp: is-min2-def\ min2-def\ split: prod.split$)

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

apply ($cases\ l\ \text{rule: } invpst.cases;$

$cases\ r\ \text{rule: } invpst.cases;$

```

      simp add: mkNode-def min-kp-def)
    subgoal using is-min2-min2-insI by blast
    subgoal by (auto intro!: is-min2-min2-insI simp: insert-commute)
  subgoal by (smt Un-insert-left Un-insert-right is-min2-mergeI is-min2-min2-insI
              sup-assoc)
done
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) |

```

$\text{baliR } t1 \ a \ t2 = \text{mkB } t1 \ a \ t2$

fun $\text{paint} :: \text{tcolor} \Rightarrow ('k, 'p :: \text{linorder}) \text{rbth} \Rightarrow ('k, 'p :: \text{linorder}) \text{rbth}$ **where**
 $\text{paint } c \ \text{Leaf} = \text{Leaf} \mid$
 $\text{paint } c \ (\text{Node } l \ (a, (-, \text{mkp})) \ r) = \text{Node } l \ (a, (c, \text{mkp})) \ r$

fun $\text{baldL} :: ('k, 'p :: \text{linorder}) \text{rbth} \Rightarrow 'k \times 'p \Rightarrow ('k, 'p :: \text{linorder}) \text{rbth}$
 $\Rightarrow ('k, 'p :: \text{linorder}) \text{rbth}$

where

$\text{baldL } (R - t1 \ x \ t2) \ y \ t3 = \text{mkR } (\text{mkB } t1 \ x \ t2) \ y \ t3 \mid$
 $\text{baldL } \text{bl } x \ (B - t1 \ y \ t2) = \text{baliR } \text{bl } x \ (\text{mkR } t1 \ y \ t2) \mid$
 $\text{baldL } \text{bl } x \ (R - (B - t1 \ y \ t2) \ z \ t3)$
 $= \text{mkR } (\text{mkB } \text{bl } x \ t1) \ y \ (\text{baliR } t2 \ z \ (\text{paint } \text{Red } t3)) \mid$
 $\text{baldL } t1 \ x \ t2 = \text{mkR } t1 \ x \ t2$

fun $\text{baldR} :: ('k, 'p :: \text{linorder}) \text{rbth} \Rightarrow 'k \times 'p \Rightarrow ('k, 'p :: \text{linorder}) \text{rbth}$
 $\Rightarrow ('k, 'p :: \text{linorder}) \text{rbth}$

where

$\text{baldR } t1 \ x \ (R - t2 \ y \ t3) = \text{mkR } t1 \ x \ (\text{mkB } t2 \ y \ t3) \mid$
 $\text{baldR } (B - t1 \ x \ t2) \ y \ t3 = \text{baliL } (\text{mkR } t1 \ x \ t2) \ y \ t3 \mid$
 $\text{baldR } (R - t1 \ x \ (B - t2 \ y \ t3)) \ z \ t4$
 $= \text{mkR } (\text{baliL } (\text{paint } \text{Red } t1) \ x \ t2) \ y \ (\text{mkB } t3 \ z \ t4) \mid$
 $\text{baldR } t1 \ x \ t2 = \text{mkR } t1 \ x \ t2$

fun $\text{combine} :: ('k, 'p :: \text{linorder}) \text{rbth} \Rightarrow ('k, 'p :: \text{linorder}) \text{rbth}$
 $\Rightarrow ('k, 'p :: \text{linorder}) \text{rbth}$

where

$\text{combine } \text{Leaf } t = t \mid$
 $\text{combine } t \ \text{Leaf} = t \mid$
 $\text{combine } (R - t1 \ a \ t2) \ (R - t3 \ c \ t4) =$
 $(\text{case } \text{combine } t2 \ t3 \ \text{of}$
 $\quad R - u2 \ b \ u3 \Rightarrow (\text{mkR } (\text{mkR } t1 \ a \ u2) \ b \ (\text{mkR } u3 \ c \ t4)) \mid$
 $\quad t23 \Rightarrow \text{mkR } t1 \ a \ (\text{mkR } t23 \ c \ t4)) \mid$
 $\text{combine } (B - t1 \ a \ t2) \ (B - t3 \ c \ t4) =$
 $(\text{case } \text{combine } t2 \ t3 \ \text{of}$
 $\quad R - t2' \ b \ t3' \Rightarrow \text{mkR } (\text{mkB } t1 \ a \ t2') \ b \ (\text{mkB } t3' \ c \ t4) \mid$
 $\quad t23 \Rightarrow \text{baldL } t1 \ a \ (\text{mkB } t23 \ c \ t4)) \mid$
 $\text{combine } t1 \ (R - t2 \ a \ t3) = \text{mkR } (\text{combine } t1 \ t2) \ a \ t3 \mid$
 $\text{combine } (R - t1 \ a \ t2) \ t3 = \text{mkR } t1 \ a \ (\text{combine } t2 \ t3)$

fun $\text{color} :: ('k, 'p) \text{rbth} \Rightarrow \text{tcolor}$ **where**

$\text{color } \text{Leaf} = \text{Black} \mid$
 $\text{color } (\text{Node } - \ (-, (c, -)) \ -) = c$

fun $\text{upd} :: 'a :: \text{linorder} \Rightarrow 'b :: \text{linorder} \Rightarrow ('a, 'b) \text{rbth} \Rightarrow ('a, 'b) \text{rbth}$ **where**

$\text{upd } x \ y \ \text{Leaf} = \text{mkR } \text{Leaf } (x, y) \ \text{Leaf} \mid$
 $\text{upd } x \ y \ (B - l \ (a, b) \ r) = (\text{case } \text{cmp } x \ a \ \text{of}$
 $\quad LT \Rightarrow \text{baliL } (\text{upd } x \ y \ l) \ (a, b) \ r \mid$

$GT \Rightarrow \text{balR } l \ (a,b) \ (\text{upd } x \ y \ r) \ |$
 $EQ \Rightarrow \text{mkB } l \ (x,y) \ r \ |$
 $\text{upd } x \ y \ (R - l \ (a,b) \ r) = (\text{case } \text{cmp } x \ a \ \text{of}$
 $LT \Rightarrow \text{mkR } (\text{upd } x \ y \ l) \ (a,b) \ r \ |$
 $GT \Rightarrow \text{mkR } l \ (a,b) \ (\text{upd } x \ y \ r) \ |$
 $EQ \Rightarrow \text{mkR } l \ (x,y) \ r)$

definition $\text{update} :: 'a::\text{linorder} \Rightarrow 'b::\text{linorder} \Rightarrow ('a,'b) \text{rbth} \Rightarrow ('a,'b) \text{rbth}$
where

$\text{update } x \ y \ t = \text{paint } \text{Black} \ (\text{upd } x \ y \ t)$

fun $\text{del} :: 'a::\text{linorder} \Rightarrow ('a,'b)::\text{linorder})\text{rbth} \Rightarrow ('a,'b)\text{rbth}$ **where**
 $\text{del } x \ \text{Leaf} = \text{Leaf} \ |$
 $\text{del } x \ (\text{Node } l \ ((a,b), (c,-)) \ r) = (\text{case } \text{cmp } x \ a \ \text{of}$
 $LT \Rightarrow \text{if } l \neq \text{Leaf} \wedge \text{color } l = \text{Black}$
 $\quad \text{then } \text{baldL} \ (\text{del } x \ l) \ (a,b) \ r \ \text{else } \text{mkR} \ (\text{del } x \ l) \ (a,b) \ r \ |$
 $GT \Rightarrow \text{if } r \neq \text{Leaf} \wedge \text{color } r = \text{Black}$
 $\quad \text{then } \text{baldR} \ l \ (a,b) \ (\text{del } x \ r) \ \text{else } \text{mkR} \ l \ (a,b) \ (\text{del } x \ r) \ |$
 $EQ \Rightarrow \text{combine } l \ r)$

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

3.1.2 Invariants

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

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

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

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

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

3.2 Functional Correctness

lemma $\text{inorder-paint}[\text{simp}]$: $\text{inorder}(\text{paint } c \ t) = \text{inorder } t$
by(cases t) (auto)

lemma *inorder-mkNode*[simp]:
 $inorder (mkNode\ c\ l\ a\ r) = inorder\ l\ @\ a\ \#\ inorder\ r$
by (*auto simp: mkNode-def*)

lemma *inorder-baliL*[simp]:
 $inorder(baliL\ l\ a\ r) = inorder\ l\ @\ a\ \#\ inorder\ r$
by(*cases (l,a,r) rule: baliL.cases*) (*auto*)

lemma *inorder-baliR*[simp]:
 $inorder(baliR\ l\ a\ r) = inorder\ l\ @\ a\ \#\ inorder\ r$
by(*cases (l,a,r) rule: baliR.cases*) (*auto*)

lemma *inorder-baldL*[simp]:
 $inorder(baldL\ l\ a\ r) = inorder\ l\ @\ a\ \#\ inorder\ r$
by (*cases (l,a,r) rule: baldL.cases*) *auto*

lemma *inorder-baldR*[simp]:
 $inorder(baldR\ l\ a\ r) = inorder\ l\ @\ a\ \#\ inorder\ r$
by(*cases (l,a,r) rule: baldR.cases*) *auto*

lemma *inorder-combine*[simp]:
 $inorder(combine\ l\ r) = inorder\ l\ @\ inorder\ r$
by (*induction l r rule: combine.induct*) (*auto split: tree.split tcolor.split*)

lemma *inorder-upd*:
 $sorted1(inorder\ t) \implies inorder(upd\ x\ y\ t) = upd-list\ x\ y\ (inorder\ t)$
by(*induction x y t rule: upd.induct*)
(auto simp: upd-list-simps)

lemma *inorder-update*:
 $sorted1(inorder\ t) \implies inorder(update\ x\ y\ t) = upd-list\ x\ y\ (inorder\ t)$
by(*simp add: update-def inorder-upd*)

lemma *inorder-del*:
 $sorted1(inorder\ t) \implies inorder(del\ x\ t) = del-list\ x\ (inorder\ t)$
by(*induction x t rule: del.induct*)
(auto simp: del-list-simps)

lemma *inorder-delete*:
 $sorted1(inorder\ t) \implies inorder(delete\ x\ t) = del-list\ x\ (inorder\ t)$
by(*simp add: delete-def inorder-del*)

3.3 Invariant Preservation

lemma *color-paint-Black*: $color\ (paint\ Black\ t) = Black$
by (*cases t*) *auto*

theorem *rbt-Leaf*: *rbt Leaf*

by (*simp add: rbt-def*)

lemma *invc2I*: $\text{invc } t \implies \text{invc2 } t$

by (*cases t rule: invc.cases*) *simp+*

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

by (*cases t*) *auto*

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

by (*cases t*) *auto*

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

by (*cases t*) *auto*

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

$\text{invc } (\text{mkR } l \ a \ r) \longleftrightarrow \text{invc } l \wedge \text{invc } r \wedge \text{color } l = \text{Black} \wedge \text{color } r = \text{Black}$

$\text{invc } (\text{mkB } l \ a \ r) \longleftrightarrow \text{invc } l \wedge \text{invc } r$

by (*simp-all add: mkNode-def*)

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

by (*simp-all add: mkNode-def*)

3.3.1 Update

lemma *invc-baliL*:

$\llbracket \text{invc2 } l; \text{invc } r \rrbracket \implies \text{invc } (\text{baliL } l \ a \ r)$

by (*induct l a r rule: baliL.induct*) *auto*

lemma *invc-baliR*:

$\llbracket \text{invc } l; \text{invc2 } r \rrbracket \implies \text{invc } (\text{baliR } l \ a \ r)$

by (*induct l a r rule: baliR.induct*) *auto*

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

$\text{bheight } (\text{mkR } l \ a \ r) = \text{bheight } l$

$\text{bheight } (\text{mkB } l \ a \ r) = \text{Suc } (\text{bheight } l)$

by (*simp-all add: mkNode-def*)

lemma *bheight-baliL*:

$\text{bheight } l = \text{bheight } r \implies \text{bheight } (\text{baliL } l \ a \ r) = \text{Suc } (\text{bheight } l)$

by (*induct l a r rule: baliL.induct*) *auto*

lemma *bheight-baliR*:

$\text{bheight } l = \text{bheight } r \implies \text{bheight } (\text{baliR } l \ a \ r) = \text{Suc } (\text{bheight } l)$

by (*induct l a r rule: baliR.induct*) *auto*

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

$\text{invh } (\text{mkNode } c \ l \ a \ r) \longleftrightarrow \text{invh } l \wedge \text{invh } r \wedge \text{bheight } l = \text{bheight } r$

by (*simp add: mkNode-def*)

lemma *invh-baliL*:

$\llbracket \text{invh } l; \text{invh } r; \text{bheight } l = \text{bheight } r \rrbracket \implies \text{invh } (\text{baliL } l \ a \ r)$
by (*induct l a r rule: baliL.induct*) *auto*

lemma *invh-baliR*:

$\llbracket \text{invh } l; \text{invh } r; \text{bheight } l = \text{bheight } r \rrbracket \implies \text{invh } (\text{baliR } l \ a \ r)$
by (*induct l a r rule: baliR.induct*) *auto*

lemma *invc-upd: assumes invc t*

shows $\text{color } t = \text{Black} \implies \text{invc } (\text{upd } x \ y \ t) \ \text{invc2 } (\text{upd } x \ y \ t)$
using *assms*
by (*induct x y t rule: upd.induct*)
(*auto simp: invc-baliL invc-baliR invc2I mkNode-def*)

lemma *invh-upd: assumes invh t*

shows $\text{invh } (\text{upd } x \ y \ t) \ \text{bheight } (\text{upd } x \ y \ t) = \text{bheight } t$
using *assms*
by(*induct x y t rule: upd.induct*)
(*auto simp: invh-baliL invh-baliR bheight-baliL bheight-baliR*)

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

by (*cases (c,t) rule: paint.cases*) *auto*

lemma *invpst-baliR: invpst l \implies invpst r \implies invpst (baliR l a r)*

by (*cases (l,a,r) rule: baliR.cases*) *auto*

lemma *invpst-baliL: invpst l \implies invpst r \implies invpst (baliL l a r)*

by (*cases (l,a,r) rule: baliL.cases*) *auto*

lemma *invpst-upd: invpst t \implies invpst (upd x y t)*

by (*induct x y t rule: upd.induct*) (*auto simp: invpst-baliR invpst-baliL*)

theorem *rbt-update: rbt t \implies rbt (update x y t)*

by (*simp add: invc-upd(2) invh-upd(1) color-paint-Black invc-paint-Black*
invh-paint rbt-def update-def invpst-upd)

3.3.2 Delete

lemma *bheight-paint-Red:*

$\text{color } t = \text{Black} \implies \text{bheight } (\text{paint } \text{Red } t) = \text{bheight } t - 1$
by (*cases t*) *auto*

lemma *invh-baldL-invc:*

$\llbracket \text{invh } l; \text{invh } r; \text{bheight } l + 1 = \text{bheight } r; \text{invc } r \rrbracket$

$\implies \text{invh}(\text{baldL } l \ a \ r) \wedge \text{bheight}(\text{baldL } l \ a \ r) = \text{bheight } l + 1$
by (*induct l a r rule: baldL.induct*)
(auto simp: invh-baliR invh-paint bheight-baliR bheight-paint-Red)

lemma *invh-baldL-Black*:

$\llbracket \text{invh } l; \text{invh } r; \text{bheight } l + 1 = \text{bheight } r; \text{color } r = \text{Black} \rrbracket$
 $\implies \text{invh}(\text{baldL } l \ a \ r) \wedge \text{bheight}(\text{baldL } l \ a \ r) = \text{bheight } r$
by (*induct l a r rule: baldL.induct*) (*auto simp add: invh-baliR bheight-baliR*)

lemma *invc-baldL*: $\llbracket \text{invc2 } l; \text{invc } r; \text{color } r = \text{Black} \rrbracket \implies \text{invc}(\text{baldL } l \ a \ r)$
by (*induct l a r rule: baldL.induct*) (*auto simp: invc-baliR invc2I mkNode-def*)

lemma *invc2-baldL*: $\llbracket \text{invc2 } l; \text{invc } r \rrbracket \implies \text{invc2}(\text{baldL } l \ a \ r)$
by (*induct l a r rule: baldL.induct*)
(auto simp: invc-baliR paint-invc2 invc2I mkNode-def)

lemma *invh-baldR-invc*:

$\llbracket \text{invh } l; \text{invh } r; \text{bheight } l = \text{bheight } r + 1; \text{invc } l \rrbracket$
 $\implies \text{invh}(\text{baldR } l \ a \ r) \wedge \text{bheight}(\text{baldR } l \ a \ r) = \text{bheight } l$
by(*induct l a r rule: baldR.induct*)
(auto simp: invh-baliL bheight-baliL invh-paint bheight-paint-Red)

lemma *invc-baldR*: $\llbracket \text{invc } a; \text{invc2 } b; \text{color } a = \text{Black} \rrbracket \implies \text{invc}(\text{baldR } a \ x \ b)$
by (*induct a x b rule: baldR.induct*) (*simp-all add: invc-baliL mkNode-def*)

lemma *invc2-baldR*: $\llbracket \text{invc } l; \text{invc2 } r \rrbracket \implies \text{invc2}(\text{baldR } l \ x \ r)$
by (*induct l x r rule: baldR.induct*)
(auto simp: invc-baliL paint-invc2 invc2I mkNode-def)

lemma *invh-combine*:

$\llbracket \text{invh } l; \text{invh } r; \text{bheight } l = \text{bheight } r \rrbracket$
 $\implies \text{invh}(\text{combine } l \ r) \wedge \text{bheight}(\text{combine } l \ r) = \text{bheight } l$
by (*induct l r rule: combine.induct*)
(auto simp: invh-baldL-Black split: tree.splits tcolor.splits)

lemma *invc-combine*:

assumes *invc l invc r*
shows $\text{color } l = \text{Black} \implies \text{color } r = \text{Black} \implies \text{invc}(\text{combine } l \ r)$
 $\text{invc2}(\text{combine } l \ r)$
using *assms*
by (*induct l r rule: combine.induct*)
(auto simp: invc-baldL invc2I mkNode-def split: tree.splits tcolor.splits)

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

lemma *del-invc-invh*: $\text{invh } t \implies \text{invc } t \implies \text{invh}(\text{del } x \ t) \wedge$
 $(\text{color } t = \text{Red} \wedge \text{bheight}(\text{del } x \ t) = \text{bheight } t \wedge \text{invc}(\text{del } x \ t) \vee$
 $\text{color } t = \text{Black} \wedge \text{bheight}(\text{del } x \ t) = \text{bheight } t - 1 \wedge \text{invc2}(\text{del } x \ t))$

```

proof (induct x t rule: del.induct)
case ( $2 x - y - c$ )
  have  $x = y \vee x < y \vee x > y$  by auto
  thus ?case proof (elim disjE)
    assume  $x = y$ 
    with 2 show ?thesis
    by (cases c) (simp-all add: invh-combine invc-combine)
  next
    assume  $x < y$ 
    with 2 show ?thesis
    by(cases c)
      (auto
        simp: invh-baldL-invc invc-baldL invc2-baldL mkNode-def
        dest: neq-LeafD)
  next
    assume  $y < x$ 
    with 2 show ?thesis
    by(cases c)
      (auto
        simp: invh-baldR-invc invc-baldR invc2-baldR mkNode-def
        dest: neq-LeafD)
  qed
qed auto

lemma invpst-baldR:  $invpst\ l \implies invpst\ r \implies invpst\ (baldR\ l\ a\ r)$ 
by (cases (l,a,r) rule: baldR.cases) (auto simp: invpst-baliL)

lemma invpst-baldL:  $invpst\ l \implies invpst\ r \implies invpst\ (baldL\ l\ a\ r)$ 
by (cases (l,a,r) rule: baldL.cases) (auto simp: invpst-baliR)

lemma invpst-combine:  $invpst\ l \implies invpst\ r \implies invpst\ (combine\ l\ r)$ 
by(induction l r rule: combine.induct)
  (auto split: tree.splits tcolor.splits simp: invpst-baldR invpst-baldL)

lemma invpst-del:  $invpst\ t \implies invpst\ (del\ x\ t)$ 
by(induct x t rule: del.induct)
  (auto simp: invpst-baldR invpst-baldL invpst-combine)

theorem rbt-delete:  $rbt\ t \implies rbt\ (delete\ k\ t)$ 
apply (clarsimp simp: delete-def rbt-def)
apply (frule (1) del-invc-invh[where x=k])
apply (auto simp: invc-paint-Black invh-paint color-paint-Black invpst-del)
done

lemma rbt-getmin-ismin:
   $rbt\ t \implies t \neq Leaf \implies is-min2\ (pst-getmin\ t)\ (set-tree\ t)$ 
unfolding rbt-def by (simp add: pst-getmin-ismin)

definition rbt-is-empty  $t \equiv t = Leaf$ 

```

lemma *rbt-is-empty*: $rbt-is-empty\ t \longleftrightarrow inorder\ t = []$
by (*cases t*) (*auto simp: rbt-is-empty-def*)

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*

apply *standard*

apply (*auto simp: lookup-map-of inorder-update inorder-delete rbt-update*

rbt-delete rbt-Leaf rbt-is-empty empty-def

dest: rbt-getmin-ismin)

done

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.