

Splay Tree

Tobias Nipkow

October 27, 2022

Abstract

Splay trees are self-adjusting binary search trees which were invented by Sleator and Tarjan [3]. This entry provides executable and verified functional splay trees as well as the related splay heaps due to Okasaki [2].

The amortized complexity of splay trees and heaps is analyzed in the AFP entry [Amortized Complexity](#).

Contents

1	Splay Tree	1
1.1	Function <i>splay</i>	2
1.2	Functional Correctness Proofs I	4
1.2.1	Verification of <i>isin</i>	4
1.2.2	Verification of <i>insert</i>	4
1.2.3	Verification of <i>delete</i>	5
1.2.4	Overall Correctness	5
1.2.5	Size lemmas	6
2	Splay Tree Implementation of Maps	6
2.1	Functional Correctness Proofs	8
2.1.1	Proofs for lookup	8
2.1.2	Proofs for update	8
2.1.3	Proofs for delete	9
2.1.4	Overall Correctness	9
3	Splay Heap	9

1 Splay Tree

```
theory Splay-Tree
imports
  HOL-Library.Tree
```

HOL-Data-Structures.Set-Specs
HOL-Data-Structures.Cmp

begin

declare *sorted-wrt.simps(2)[simp del]*

Splay trees were invented by Sleator and Tarjan [3].

1.1 Function *splay*

function *splay* :: 'a::linorder \Rightarrow 'a tree \Rightarrow 'a tree **where**

splay *x* *Leaf* = *Leaf* |
splay *x* (*Node* *AB* *x* *CD*) = *Node* *AB* *x* *CD* |
x < *b* \Rightarrow *splay* *x* (*Node* (*Node* *A* *x* *B*) *b* *CD*) = *Node* *A* *x* (*Node* *B* *b* *CD*) |
x < *b* \Rightarrow *splay* *x* (*Node* *Leaf* *b* *CD*) = *Node* *Leaf* *b* *CD* |
x < *a* \Rightarrow *x* < *b* \Rightarrow *splay* *x* (*Node* (*Node* *Leaf* *a* *B*) *b* *CD*) = *Node* *Leaf* *a* (*Node* *B* *b* *CD*) |
x < *a* \Rightarrow *x* < *b* \Rightarrow *A* \neq *Leaf* \Rightarrow
splay *x* (*Node* (*Node* *A* *a* *B*) *b* *CD*) =
(case *splay* *x* *A* of *Node* *A1* *a'* *A2* \Rightarrow *Node* *A1* *a'* (*Node* *A2* *a* (*Node* *B* *b* *CD*))) |
a < *x* \Rightarrow *x* < *b* \Rightarrow *splay* *x* (*Node* (*Node* *A* *a* *Leaf*) *b* *CD*) = *Node* *A* *a* (*Node* *Leaf* *b* *CD*) |
a < *x* \Rightarrow *x* < *b* \Rightarrow *B* \neq *Leaf* \Rightarrow
splay *x* (*Node* (*Node* *A* *a* *B*) *b* *CD*) =
(case *splay* *x* *B* of *Node* *B1* *b'* *B2* \Rightarrow *Node* (*Node* *A* *a* *B1*) *b'* (*Node* *B2* *b* *CD*)) |
b < *x* \Rightarrow *splay* *x* (*Node* *AB* *b* (*Node* *C* *x* *D*)) = *Node* (*Node* *AB* *b* *C*) *x* *D* |
b < *x* \Rightarrow *splay* *x* (*Node* *AB* *b* *Leaf*) = *Node* *AB* *b* *Leaf* |
b < *x* \Rightarrow *x* < *c* \Rightarrow *C* \neq *Leaf* \Rightarrow
splay *x* (*Node* *AB* *b* (*Node* *C* *c* *D*)) =
(case *splay* *x* *C* of *Node* *C1* *c'* *C* \Rightarrow *Node* (*Node* *AB* *b* *C1*) *c'* (*Node* *C* *c* *D*)) |
b < *x* \Rightarrow *x* < *c* \Rightarrow *splay* *x* (*Node* *AB* *b* (*Node* *Leaf* *c* *D*)) = *Node* (*Node* *AB* *b* *Leaf*) *c* *D* |
b < *x* \Rightarrow *c* < *x* \Rightarrow *splay* *x* (*Node* *AB* *b* (*Node* *C* *c* *Leaf*)) = *Node* (*Node* *AB* *b* *C*) *c* *Leaf* |
a < *x* \Rightarrow *c* < *x* \Rightarrow *D* \neq *Leaf* \Rightarrow
splay *x* (*Node* *AB* *a* (*Node* *C* *c* *D*)) =
(case *splay* *x* *D* of *Node* *D1* *d'* *D2* \Rightarrow *Node* (*Node* (*Node* *AB* *a* *C*) *c* *D1*) *d'* *D2*)

apply(*atomize-elim*)

apply(*auto*)

apply (*subst* (*asm*) *neq-Leaf-iff*)

apply(*auto*)

apply (*metis tree.exhaust le-less-linear less-linear*)+

done

termination *splay*

by *lexicographic-order*

lemma *splay-code*: *splay* *x* (*Node* *AB* *b* *CD*) =

(case *cmp* *x* *b* of

```

EQ ⇒ Node AB b CD |
LT ⇒ (case AB of
  Leaf ⇒ Node AB b CD |
  Node A a B ⇒
    (case cmp x a of EQ ⇒ Node A a (Node B b CD) |
      LT ⇒ if A = Leaf then Node A a (Node B b CD)
        else case splay x A of
          Node A1 a' A2 ⇒ Node A1 a' (Node A2 a (Node B b CD)) |
      GT ⇒ if B = Leaf then Node A a (Node B b CD)
        else case splay x B of
          Node B1 b' B2 ⇒ Node (Node A a B1) b' (Node B2 b CD))) |
GT ⇒ (case CD of
  Leaf ⇒ Node AB b CD |
  Node C c D ⇒
    (case cmp x c of EQ ⇒ Node (Node AB b C) c D |
      LT ⇒ if C = Leaf then Node (Node AB b C) c D
        else case splay x C of
          Node C1 c' C2 ⇒ Node (Node AB b C1) c' (Node C2 c D) |
      GT ⇒ if D = Leaf then Node (Node AB b C) c D
        else case splay x D of
          Node D1 d D2 ⇒ Node (Node (Node AB b C) c D1) d D2)))
by(auto split!: tree.split)

```

definition *is-root* :: 'a ⇒ 'a tree ⇒ bool **where**
is-root x t = (case t of Leaf ⇒ False | Node l a r ⇒ x = a)

definition *isin* t x = *is-root* x (splay x t)

definition *empty* :: 'a tree **where**
empty = Leaf

hide-const (open) *insert*

fun *insert* :: 'a::linorder ⇒ 'a tree ⇒ 'a tree **where**
insert x t =
 (if t = Leaf then Node Leaf x Leaf
 else case splay x t of
 Node l a r ⇒
 case cmp x a of
 EQ ⇒ Node l a r |
 LT ⇒ Node l x (Node Leaf a r) |
 GT ⇒ Node (Node l a Leaf) x r)

fun *splay-max* :: 'a tree ⇒ 'a tree **where**
splay-max Leaf = Leaf |
splay-max (Node A a Leaf) = Node A a Leaf |
splay-max (Node A a (Node B b CD)) =
 (if CD = Leaf then Node (Node A a B) b Leaf

else case *splay-max* *CD* of
 Node *C c D* \Rightarrow Node (Node (Node *A a B*) *b C*) *c D*)

lemma *splay-max-code*: *splay-max t* = (case *t* of
 Leaf \Rightarrow *t* |
 Node *la a ra* \Rightarrow (case *ra* of
 Leaf \Rightarrow *t* |
 Node *lb b rb* \Rightarrow
 (if *rb=Leaf* then Node (Node *la a lb*) *b rb*
 else case *splay-max rb* of
 Node *lc c rc* \Rightarrow Node (Node (Node *la a lb*) *b lc*) *c rc*)))
by(*auto simp: neq-Leaf-iff split: tree.split*)

definition *delete* :: '*a*::*linorder* \Rightarrow '*a* tree \Rightarrow '*a* tree **where**
delete x t =
 (if *t = Leaf* then *Leaf*
 else case *splay x t* of Node *l a r* \Rightarrow
 if *x \neq a* then Node *l a r*
 else if *l = Leaf* then *r* else case *splay-max l* of Node *l' m r'* \Rightarrow Node *l' m r*)

1.2 Functional Correctness Proofs I

This subsection follows the automated method by Nipkow [1].

lemma *splay-Leaf-iff[simp]*: (*splay a t = Leaf*) = (*t = Leaf*)
by(*induction a t rule: splay.induct*) (*auto split: tree.splits*)

lemma *splay-max-Leaf-iff[simp]*: (*splay-max t = Leaf*) = (*t = Leaf*)
by(*induction t rule: splay-max.induct*)(*auto split: tree.splits*)

1.2.1 Verification of *isin*

lemma *splay-elemsD*:
splay x t = Node l a r \Longrightarrow *sorted(inorder t)* \Longrightarrow
x \in set (inorder t) \longleftrightarrow *x=a*
by(*induction x t arbitrary: l a r rule: splay.induct*)
 (*auto simp: isin-simps ball-Un split: tree.splits*)

lemma *isin-set*: *sorted(inorder t)* \Longrightarrow *isin t x = (x \in set (inorder t))*
by (*auto simp: isin-def is-root-def dest: splay-elemsD split: tree.splits*)

1.2.2 Verification of *insert*

lemma *inorder-splay*: *inorder(splay x t) = inorder t*
by(*induction x t rule: splay.induct*)
 (*auto simp: neq-Leaf-iff split: tree.split*)

lemma *sorted-splay*:
sorted(inorder t) \Longrightarrow *splay x t = Node l a r* \Longrightarrow
sorted(inorder l @ x # inorder r)

unfolding *inorder-splay*[of $x t$, *symmetric*]
by(*induction* $x t$ *arbitrary*: $l a r$ *rule*: *splay.induct*)
(*auto simp*: *sorted-lems sorted-Cons-le sorted-snoc-le split: tree.splits*)

lemma *inorder-insert*:
 $sorted(inorder t) \implies inorder(insert x t) = ins-list x (inorder t)$
using *inorder-splay*[of $x t$, *symmetric*] *sorted-splay*[of $t x$]
by(*auto simp*: *ins-list-simps ins-list-Cons ins-list-snoc neq-Leaf-iff split: tree.split*)

1.2.3 Verification of delete

lemma *inorder-splay-maxD*:
 $splay-max t = Node l a r \implies sorted(inorder t) \implies$
 $inorder l @ [a] = inorder t \wedge r = Leaf$
by(*induction* t *arbitrary*: $l a r$ *rule*: *splay-max.induct*)
(*auto simp*: *sorted-lems split: tree.splits if-splits*)

lemma *inorder-delete*:
 $sorted(inorder t) \implies inorder(delete x t) = del-list x (inorder t)$
using *inorder-splay*[of $x t$, *symmetric*] *sorted-splay*[of $t x$]
by (*auto simp*: *del-list-simps del-list-sorted-app delete-def*
del-list-notin-Cons inorder-splay-maxD split: tree.splits)

1.2.4 Overall Correctness

interpretation *splay*: *Set-by-Ordered*
where *empty* = *empty* **and** *isin* = *isin* **and** *insert* = *insert*
and *delete* = *delete* **and** *inorder* = *inorder* **and** *inv* = $\lambda\cdot$. *True*
proof (*standard*, *goal-cases*)
case 2 **thus** ?*case* **by**(*simp add*: *isin-set*)
next
case 3 **thus** ?*case* **by**(*simp add*: *inorder-insert del: insert.simps*)
next
case 4 **thus** ?*case* **by**(*simp add*: *inorder-delete*)
qed (*auto simp*: *empty-def*)

Corollaries:

lemma *bst-splay*: $bst t \implies bst (splay x t)$
by (*simp add*: *bst-iff-sorted-wrt-less inorder-splay*)

lemma *bst-insert*: $bst t \implies bst(insert x t)$
using *splay.invar-insert*[of $t x$] **by** (*simp add*: *bst-iff-sorted-wrt-less splay.invar-def*)

lemma *bst-delete*: $bst t \implies bst(delete x t)$
using *splay.invar-delete*[of $t x$] **by** (*simp add*: *bst-iff-sorted-wrt-less splay.invar-def*)

lemma *splay-bstL*: $bst t \implies splay a t = Node l e r \implies x \in set-tree l \implies x < a$
by (*metis* *bst-iff-sorted-wrt-less list.set-intros*(1) *set-inorder sorted-splay sorted-wrt-append*)

lemma *splay-bstR*: $bst t \implies splay a t = Node l e r \implies x \in set-tree r \implies a < x$

by (metis bst-iff-sorted-wrt-less sorted-Cons-iff set-inorder sorted-splay sorted-wrt-append)

1.2.5 Size lemmas

```
lemma size-splay[simp]: size (splay a t) = size t
apply(induction a t rule: splay.induct)
apply auto
  apply(force split: tree.split)+
done
```

```
lemma size-if-splay: splay a t = Node l u r  $\implies$  size t = size l + size r + 1
by (metis One-nat-def size-splay tree.size(4))
```

```
lemma splay-not-Leaf: t  $\neq$  Leaf  $\implies$   $\exists$  l x r. splay a t = Node l x r
by (metis neq-Leaf-iff splay-Leaf-iff)
```

```
lemma size-splay-max: size(splay-max t) = size t
apply(induction t rule: splay-max.induct)
  apply(simp)
  apply(simp)
apply(clarsimp split: tree.split)
done
```

```
lemma size-if-splay-max: splay-max t = Node l u r  $\implies$  size t = size l + size r + 1
by (metis One-nat-def size-splay-max tree.size(4))
```

end

2 Splay Tree Implementation of Maps

theory Splay-Map

imports

Splay-Tree

HOL-Data-Structures.Map-Specs

begin

```
function splay :: 'a::linorder  $\Rightarrow$  ('a*'b) tree  $\Rightarrow$  ('a*'b) tree where
splay x Leaf = Leaf |
x = fst a  $\implies$  splay x (Node t1 a t2) = Node t1 a t2 |
x = fst a  $\implies$  x < fst b  $\implies$  splay x (Node (Node t1 a t2) b t3) = Node t1 a (Node
t2 b t3) |
x < fst a  $\implies$  splay x (Node Leaf a t) = Node Leaf a t |
x < fst a  $\implies$  x < fst b  $\implies$  splay x (Node (Node Leaf a t1) b t2) = Node Leaf a
(Node t1 b t2) |
x < fst a  $\implies$  x < fst b  $\implies$  t1  $\neq$  Leaf  $\implies$ 
splay x (Node (Node t1 a t2) b t3) =
```

```

(case splay x t1 of Node t11 y t12 ⇒ Node t11 y (Node t12 a (Node t2 b t3))) |
fst a < x ⇒ x < fst b ⇒ splay x (Node (Node t1 a Leaf) b t2) = Node t1 a
(Node Leaf b t2) |
fst a < x ⇒ x < fst b ⇒ t2 ≠ Leaf ⇒
splay x (Node (Node t1 a t2) b t3) =
(case splay x t2 of Node t21 y t22 ⇒ Node (Node t1 a t21) y (Node t22 b t3)) |
fst a < x ⇒ x = fst b ⇒ splay x (Node t1 a (Node t2 b t3)) = Node (Node t1
a t2) b t3 |
fst a < x ⇒ splay x (Node t a Leaf) = Node t a Leaf |
fst a < x ⇒ x < fst b ⇒ t2 ≠ Leaf ⇒
splay x (Node t1 a (Node t2 b t3)) =
(case splay x t2 of Node t21 y t22 ⇒ Node (Node t1 a t21) y (Node t22 b t3)) |
fst a < x ⇒ x < fst b ⇒ splay x (Node t1 a (Node Leaf b t2)) = Node (Node t1
a Leaf) b t2 |
fst a < x ⇒ fst b < x ⇒ splay x (Node t1 a (Node t2 b Leaf)) = Node (Node
t1 a t2) b Leaf |
fst a < x ⇒ fst b < x ⇒ t3 ≠ Leaf ⇒
splay x (Node t1 a (Node t2 b t3)) =
(case splay x t3 of Node t31 y t32 ⇒ Node (Node (Node t1 a t2) b t31) y t32)
apply(atomize-elim)
apply(auto)

```

```

apply (subst (asm) neq-Leaf-iff)
apply(auto)
apply (metis tree.exhaust surj-pair less-linear)+
done

```

```

termination splay
by lexicographic-order

```

```

lemma splay-code: splay (x:::linorder) t = (case t of Leaf ⇒ Leaf |
Node al a ar ⇒ (case cmp x (fst a) of
EQ ⇒ t |
LT ⇒ (case al of
Leaf ⇒ t |
Node bl b br ⇒ (case cmp x (fst b) of
EQ ⇒ Node bl b (Node br a ar) |
LT ⇒ if bl = Leaf then Node bl b (Node br a ar)
else case splay x bl of
Node bll y blr ⇒ Node bll y (Node blr b (Node br a ar)) |
GT ⇒ if br = Leaf then Node bl b (Node br a ar)
else case splay x br of
Node brl y brr ⇒ Node (Node bl b brl) y (Node brr a ar))) |
GT ⇒ (case ar of
Leaf ⇒ t |
Node bl b br ⇒ (case cmp x (fst b) of
EQ ⇒ Node (Node al a bl) b br |
LT ⇒ if bl = Leaf then Node (Node al a bl) b br
else case splay x bl of

```

$$\begin{aligned} & \text{Node } bll \ y \ blr \Rightarrow \text{Node } (\text{Node } al \ a \ bll) \ y \ (\text{Node } blr \ b \ br) \mid \\ GT \Rightarrow & \text{if } br = \text{Leaf} \text{ then } \text{Node } (\text{Node } al \ a \ bl) \ b \ br \\ & \text{else case splay } x \ br \ \text{of} \\ & \text{Node } bll \ y \ blr \Rightarrow \text{Node } (\text{Node } (\text{Node } al \ a \ bl) \ b \ bll) \ y \ blr \end{aligned}$$
by(*auto split!*: *tree.split*)

definition *lookup* :: ('a*'b)tree \Rightarrow 'a::linorder \Rightarrow 'b option **where** *lookup* t x =
 (case splay x t of Leaf \Rightarrow None | Node - (a,b) - \Rightarrow if x=a then Some b else None)

hide-const (open) *insert*

fun *update* :: 'a::linorder \Rightarrow 'b \Rightarrow ('a*'b) tree \Rightarrow ('a*'b) tree **where**
update x y t = (if t = Leaf then Node Leaf (x,y) Leaf
 else case splay x t of
 Node l a r \Rightarrow if x = fst a then Node l (x,y) r
 else if x < fst a then Node l (x,y) (Node Leaf a r) else Node (Node l a Leaf)
 (x,y) r)

definition *delete* :: 'a::linorder \Rightarrow ('a*'b) tree \Rightarrow ('a*'b) tree **where**
delete x t = (if t = Leaf then Leaf
 else case splay x t of Node l a r \Rightarrow
 if x = fst a
 then if l = Leaf then r else case splay-max l of Node l' m r' \Rightarrow Node l' m r
 else Node l a r)

2.1 Functional Correctness Proofs

lemma *splay-Leaf-iff*: (splay x t = Leaf) = (t = Leaf)
by(*induction* x t rule: *splay.induct*) (*auto split*: *tree.splits*)

2.1.1 Proofs for lookup

lemma *splay-map-of-inorder*:
 splay x t = Node l a r \implies sorted1 (inorder t) \implies
 map-of (inorder t) x = (if x = fst a then Some(snd a) else None)
by(*induction* x t arbitrary: l a r rule: *splay.induct*)
 (*auto simp*: *map-of-simps splay-Leaf-iff split*: *tree.splits*)

lemma *lookup-eq*:
 sorted1 (inorder t) \implies lookup t x = map-of (inorder t) x
by(*auto simp*: *lookup-def splay-Leaf-iff splay-map-of-inorder split*: *tree.split*)

2.1.2 Proofs for update

lemma *inorder-splay*: inorder(splay x t) = inorder t
by(*induction* x t rule: *splay.induct*)
 (*auto simp*: *neq-Leaf-iff split*: *tree.split*)

lemma *sorted-splay*:
 sorted1 (inorder t) \implies splay x t = Node l a r \implies


```

    sorted(map fst (inorder l) @ x # map fst (inorder r))
unfolding inorder-splay[of x t, symmetric]
by(induction x t arbitrary: l a r rule: splay.induct)
    (auto simp: sorted-lems sorted-Cons-le sorted-snoc-le splay-Leaf-iff split: tree.splits)

```

lemma *inorder-update-splay*:

```

    sorted1 (inorder t)  $\implies$  inorder(update x y t) = upd-list x y (inorder t)
using inorder-splay[of x t, symmetric] sorted-splay[of t x]
by(auto simp: upd-list-simps upd-list-Cons upd-list-snoc neq-Leaf-iff split: tree.split)

```

2.1.3 Proofs for delete

lemma *inorder-splay-maxD*:

```

    splay-max t = Node l a r  $\implies$  sorted1 (inorder t)  $\implies$ 
    inorder l @ [a] = inorder t  $\wedge$  r = Leaf
by(induction t arbitrary: l a r rule: splay-max.induct)
    (auto simp: sorted-lems split: tree.splits if-splits)

```

lemma *inorder-delete-splay*:

```

    sorted1 (inorder t)  $\implies$  inorder(delete x t) = del-list x (inorder t)
using inorder-splay[of x t, symmetric] sorted-splay[of t x]
by (auto simp: del-list-simps del-list-sorted-app delete-def del-list-notin-Cons in-
order-splay-maxD
    split: tree.splits)

```

2.1.4 Overall Correctness

interpretation *Map-by-Ordered*

where *empty* = *empty* **and** *lookup* = *lookup* **and** *update* = *update*
and *delete* = *delete* **and** *inorder* = *inorder* **and** *inv* = λ -. *True*

proof (*standard*, *goal-cases*)

```

    case 2 thus ?case by(simp add: lookup-eq)
next
    case 3 thus ?case by(simp add: inorder-update-splay del: update.simps)
next
    case 4 thus ?case by(simp add: inorder-delete-splay)
qed (auto simp: empty-def)

```

end

3 Splay Heap

theory *Splay-Heap*

imports

HOL-Library.Tree-Multiset

begin

Splay heaps were invented by Okasaki [2]. They represent priority queues by splay trees, not by heaps!

```

fun get-min :: ('a::linorder) tree  $\Rightarrow$  'a where
get-min(Node l m r) = (if l = Leaf then m else get-min l)

fun partition :: 'a::linorder  $\Rightarrow$  'a tree  $\Rightarrow$  'a tree * 'a tree where
partition p Leaf = (Leaf,Leaf) |
partition p (Node al a ar) =
  (if a  $\leq$  p then
    case ar of
      Leaf  $\Rightarrow$  (Node al a ar, Leaf) |
      Node bl b br  $\Rightarrow$ 
        if b  $\leq$  p
        then let (pl,pr) = partition p br in (Node (Node al a bl) b pl, pr)
        else let (pl,pr) = partition p bl in (Node al a pl, Node pr b br)
    else case al of
      Leaf  $\Rightarrow$  (Leaf, Node al a ar) |
      Node bl b br  $\Rightarrow$ 
        if b  $\leq$  p
        then let (pl,pr) = partition p br in (Node bl b pl, Node pr a ar)
        else let (pl,pr) = partition p bl in (pl, Node pr b (Node br a ar)))

```

```

definition insert :: 'a::linorder  $\Rightarrow$  'a tree  $\Rightarrow$  'a tree where
insert x h = (let (l,r) = partition x h in Node l x r)

```

```

fun del-min :: 'a::linorder tree  $\Rightarrow$  'a tree where
del-min Leaf = Leaf |
del-min (Node Leaf - r) = r |
del-min (Node (Node ll a lr) b r) =
  (if ll = Leaf then Node lr b r else Node (del-min ll) a (Node lr b r))

```

```

lemma get-min-in:
  h  $\neq$  Leaf  $\implies$  get-min h  $\in$  set-tree h
by(induction h) auto

```

```

lemma get-min-min:
  [ bst-wrt ( $\leq$ ) h; h  $\neq$  Leaf ]  $\implies$   $\forall x \in$  set-tree h. get-min h  $\leq$  x
proof(induction h)
  case (Node l x r) thus ?case using get-min-in[of l] get-min-in[of r]
  by auto (blast intro: order-trans)
qed simp

```

```

lemma size-partition: partition p t = (l',r')  $\implies$  size t = size l' + size r'
by (induction p t arbitrary: l' r' rule: partition.induct)
  (auto split: if-splits tree.splits prod.splits)

```

```

lemma mset-partition: [ bst-wrt ( $\leq$ ) t; partition p t = (l',r') ]
 $\implies$  mset-tree t = mset-tree l' + mset-tree r'
proof(induction p t arbitrary: l' r' rule: partition.induct)
  case 1 thus ?case by simp

```

```

next
  case (2 p l a r)
  show ?thesis
  proof cases
    assume a ≤ p
    show ?thesis
    proof (cases r)
      case Leaf thus ?thesis using ⟨a ≤ p⟩ 2.prem1 by auto
    next
      case (Node rl b rr)
      show ?thesis
      proof cases
        assume b ≤ p
        thus ?thesis using Node ⟨a ≤ p⟩ 2.prem1 2.IH(1)[OF - Node]
          by (auto simp: ac-simps split: prod.splits)
      next
        assume ¬ b ≤ p
        thus ?thesis using Node ⟨a ≤ p⟩ 2.prem1 2.IH(2)[OF - Node]
          by (auto simp: ac-simps split: prod.splits)
      qed
    qed
  qed
next
  assume ¬ a ≤ p
  show ?thesis
  proof (cases l)
    case Leaf thus ?thesis using ⟨¬ a ≤ p⟩ 2.prem1 by auto
  next
    case (Node ll b lr)
    show ?thesis
    proof cases
      assume b ≤ p
      thus ?thesis using Node ⟨¬ a ≤ p⟩ 2.prem1 2.IH(3)[OF - Node]
        by (auto simp: ac-simps split: prod.splits)
    next
      assume ¬ b ≤ p
      thus ?thesis using Node ⟨¬ a ≤ p⟩ 2.prem1 2.IH(4)[OF - Node]
        by (auto simp: ac-simps split: prod.splits)
    qed
  qed
qed
qed
qed

```

lemma *set-partition*: $\llbracket \text{bst-wrt } (\leq) t; \text{partition } p t = (l', r') \rrbracket$
 $\implies \text{set-tree } t = \text{set-tree } l' \cup \text{set-tree } r'$
by (*metis mset-partition set-mset-tree set-mset-union*)

lemma *bst-partition*:
 $\text{partition } p t = (l', r') \implies \text{bst-wrt } (\leq) t \implies \text{bst-wrt } (\leq) (\text{Node } l' p r')$
proof(*induction p t arbitrary: l' r' rule: partition.induct*)

```

case 1 thus ?case by simp
next
case (2 p l a r)
show ?case
proof cases
  assume  $a \leq p$ 
  show ?thesis
  proof (cases r)
    case Leaf thus ?thesis using  $\langle a \leq p \rangle$  2.prem1 by fastforce
  next
    case (Node rl b rr)
    show ?thesis
    proof cases
      assume  $b \leq p$ 
      thus ?thesis
      using Node  $\langle a \leq p \rangle$  2.prem1 2.IH(1)[OF - Node] set-partition[of rr]
      by (fastforce split: prod.splits)
    next
      assume  $\neg b \leq p$ 
      thus ?thesis
      using Node  $\langle a \leq p \rangle$  2.prem1 2.IH(2)[OF - Node] set-partition[of rl]
      by (fastforce split: prod.splits)
    qed
  qed
next
  assume  $\neg a \leq p$ 
  show ?thesis
  proof (cases l)
    case Leaf thus ?thesis using  $\langle \neg a \leq p \rangle$  2.prem1 by fastforce
  next
    case (Node ll b lr)
    show ?thesis
    proof cases
      assume  $b \leq p$ 
      thus ?thesis
      using Node  $\langle \neg a \leq p \rangle$  2.prem1 2.IH(3)[OF - Node] set-partition[of lr]
      by (fastforce split: prod.splits)
    next
      assume  $\neg b \leq p$ 
      thus ?thesis
      using Node  $\langle \neg a \leq p \rangle$  2.prem1 2.IH(4)[OF - Node] set-partition[of ll]
      by (fastforce split: prod.splits)
    qed
  qed
qed
qed

```

lemma size-del-min[simp]: $\text{size}(\text{del-min } t) = \text{size } t - 1$
by(induction t rule: del-min.induct) (auto simp: neq-Leaf-iff)

```

lemma mset-del-min: mset-tree (del-min h) = mset-tree h - {# get-min h #}
proof(induction h rule: del-min.induct)
  case ( $\exists ll$ )
  show ?case
  proof cases
    assume  $ll = \text{Leaf}$  thus ?thesis using  $\exists$  by (simp add: ac-simps)
  next
    assume  $ll \neq \text{Leaf}$ 
    hence  $\text{get-min } ll \in \# \text{ mset-tree } ll$ 
      by (simp add: get-min-in)
    then obtain  $A$  where  $\text{mset-tree } ll = \text{add-mset (get-min } ll) A$ 
      by (blast dest: multi-member-split)
    then show ?thesis using  $\exists$  by auto
  qed
qed auto

lemma bst-del-min:  $\text{bst-wrt } (\leq) t \implies \text{bst-wrt } (\leq) (\text{del-min } t)$ 
apply(induction t rule: del-min.induct)
  apply simp
  apply simp
apply auto
by (metis Multiset.diff-subset-eq-self subsetD set-mset-mono set-mset-tree mset-del-min)

end

```

References

- [1] T. Nipkow. Automatic functional correctness proofs for functional search trees. In J. Blanchette and S. Merz, editors, *Interactive Theorem Proving (ITP 2016)*, volume 9807 of *LNCS*, pages 307–322. Springer, 2016.
- [2] C. Okasaki. *Purely Functional Data Structures*. Cambridge University Press, 1998.
- [3] D. D. Sleator and R. E. Tarjan. Self-adjusting binary search trees. *J. ACM*, 32(3):652–686, 1985.