Splay Tree

Tobias Nipkow

June 11, 2019

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

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 A x B*) = *Node A x B* |
$x<b \Longrightarrow$ *splay x* (*Node* (*Node A x B*) *b C*) = *Node A x* (*Node B b C*) |
$x<b \Longrightarrow$ *splay x* (*Node Leaf b A*) = *Node Leaf b A* |
$x<a \Longrightarrow x<b \Longrightarrow$ *splay x* (*Node* (*Node Leaf a A*) *b B*) = *Node Leaf a* (*Node A b B*) |
$x<b \Longrightarrow x<c \Longrightarrow AB \neq$ *Leaf* $\Longrightarrow$
 splay x (*Node* (*Node AB b C*) *c D*) =
 (*case splay x AB of Node A a B* $\Rightarrow$ *Node A a* (*Node B b* (*Node C c D*))) |
$a<x \Longrightarrow x<b \Longrightarrow$ *splay x* (*Node* (*Node A a Leaf*) *b B*) = *Node A a* (*Node Leaf b B*) |
$a<x \Longrightarrow x<c \Longrightarrow BC \neq$ *Leaf* $\Longrightarrow$
 splay x (*Node* (*Node A a BC*) *c D*) =
 (*case splay x BC of Node B b C* $\Rightarrow$ *Node* (*Node A a B*) *b* (*Node C c D*)) |
$a<x \Longrightarrow$ *splay x* (*Node A a* (*Node B x C*)) = *Node* (*Node A a B*) *x C* |
$a<x \Longrightarrow$ *splay x* (*Node A a Leaf*) = *Node A a Leaf* |
$a<x \Longrightarrow x<c \Longrightarrow BC \neq$ *Leaf* $\Longrightarrow$
 splay x (*Node A a* (*Node BC c D*)) =
 (*case splay x BC of Node B b C* $\Rightarrow$ *Node* (*Node A a B*) *b* (*Node C c D*)) |
$a<x \Longrightarrow x<b \Longrightarrow$ *splay x* (*Node A a* (*Node Leaf b C*)) = *Node* (*Node A a Leaf*) *b C* |
$a<x \Longrightarrow b<x \Longrightarrow$ *splay x* (*Node A a* (*Node B b Leaf*)) = *Node* (*Node A a B*) *b Leaf* |
$a<x \Longrightarrow b<x \Longrightarrow CD \neq$ *Leaf* $\Longrightarrow$
 splay x (*Node A a* (*Node B b CD*)) =
 (*case splay x CD of Node C c D* $\Rightarrow$ *Node* (*Node* (*Node A a B*) *b C*) *c D*)
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) =
  (if x=b
   then Node AB b CD
   else if x < b
        then case AB of
          Leaf ⇒ Node AB b CD |
          Node A a B ⇒
            (if x=a then Node A a (Node B b CD)
             else if x < a
                  then if A = Leaf then Node A a (Node B b CD)
                       else case splay x A of
                         Node A_1 a' A_2 ⇒ Node A_1 a' (Node A_2 a (Node B b CD))
                  else if B = Leaf then Node A a (Node B b CD)
                       else case splay x B of
                         Node B_1 b' B_2 ⇒ Node (Node A a B_1) b' (Node B_2 b CD))
        else case CD of
          Leaf ⇒ Node AB b CD |
          Node C c D ⇒
            (if x=c then Node (Node AB b C) c D
             else if x < c
                  then if C = Leaf then Node (Node AB b C) c D
                       else case splay x C of
                         Node C_1 c' C_2 ⇒ Node (Node AB b C_1) c' (Node C_2 c D)
                  else if D=Leaf then Node (Node AB b C) c D
                       else case splay x D of
                         Node D_1 d D_2 ⇒ Node (Node (Node AB b C) c D_1) d D_2))
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) |
```

$GT \Rightarrow Node\ (Node\ l\ a\ Leaf)\ x\ r)$

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 ⇒ *Node* (*Node* (*Node A a B*) *b C*) *c D*)

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

definition *delete* :: *'a*::*linorder* ⇒ *'a tree* ⇒ *'a tree* **where**
delete x t =
(*if t* = *Leaf then Leaf*
else case splay x t of Node l a r ⇒
if x = *a*
then if l = *Leaf then r else case splay-max l of Node l' m r'* ⇒ *Node l' m r*
else Node l a 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* ⟹ *sorted*(*inorder t*) ⟹
x ∈ *set* (*inorder t*) ⟷ *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*) ⟹ *isin t x* = (*x* ∈ *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*) $\Longrightarrow$ *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* $\Longrightarrow$ *sorted*(*inorder t*) $\Longrightarrow$
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*) $\Longrightarrow$ *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* = λ-. *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*)

1.3 Functional Correctness Proofs II

This subsection follows the traditional approach, is less automated and is retained more for historic reasons.

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* ⟹ *size t* = *size l* + *size r* + *1*
by (*metis One-nat-def size-splay tree.size*(*4*))

lemma *splay-not-Leaf*: *t* ≠ *Leaf* ⟹ ∃ *l x r*. *splay a t* = *Node l x r*
by (*metis neq-Leaf-iff splay-Leaf-iff*)

lemma *set-splay*: *set-tree*(*splay a t*) = *set-tree t*
proof(*induction a t rule*: *splay.induct*)
  case (*6 a*)
  with *splay-not-Leaf*[*OF 6*(*3*), *of a*] **show** *?case* **by**(*fastforce*)
next
  case (*8 - a*)
  with *splay-not-Leaf*[*OF 8*(*3*), *of a*] **show** *?case* **by**(*fastforce*)
next
  case (*11 - a*)
  with *splay-not-Leaf*[*OF 11*(*3*), *of a*] **show** *?case* **by**(*fastforce*)
next
  case (*14 - a*)
  with *splay-not-Leaf*[*OF 14*(*3*), *of a*] **show** *?case* **by**(*fastforce*)
qed *auto*

lemma *splay-bstL*: *bst t* ⟹ *splay a t* = *Node l e r* ⟹ *x* ∈ *set-tree l* ⟹ *x* < *a*
apply(*induction a t arbitrary*: *l x r rule*: *splay.induct*)
apply (*auto split*: *tree.splits*)
apply *auto*
done

lemma *splay-bstR*: *bst t* ⟹ *splay a t* = *Node l e r* ⟹ *x* ∈ *set-tree r* ⟹ *a* < *x*
apply(*induction a t arbitrary*: *l e x r rule*: *splay.induct*)
apply *auto*
apply (*fastforce split!*: *tree.splits*)+
done

lemma *bst-splay*: *bst t* ⟹ *bst*(*splay a t*)
proof(*induction a t rule*: *splay.induct*)
  case (*6 a - - ll*)
  with *splay-not-Leaf*[*OF 6*(*3*), *of a*] *set-splay*[*of a ll*,*symmetric*]
  show *?case* **by** (*fastforce*)
next
  case (*8 - a - t*)
  with *splay-not-Leaf*[*OF 8*(*3*), *of a*] *set-splay*[*of a t*,*symmetric*]
  show *?case* **by** *fastforce*
next
  case (*11 - a - t*)

  with *splay-not-Leaf*[*OF 11*(*3*), *of a*] *set-splay*[*of a t*,*symmetric*]
  show *?case* **by** *fastforce*
next
  case (*14* - *a* - *t*)
  with *splay-not-Leaf*[*OF 14*(*3*), *of a*] *set-splay*[*of a t*,*symmetric*]
  show *?case* **by** *fastforce*
qed *auto*

lemma *splay-to-root*: $\llbracket$ *bst t*; *splay a t* = *t'* $\rrbracket \Longrightarrow$
  $a \in$ *set-tree t* $\longleftrightarrow$ ($\exists$ *l r*. *t'* = *Node l a r*)
proof(*induction a t arbitrary*: *t' rule*: *splay.induct*)
  case (*6 a*)
  with *splay-not-Leaf*[*OF 6*(*3*), *of a*] **show** *?case* **by** *auto*
next
  case (*8* - *a*)
  with *splay-not-Leaf*[*OF 8*(*3*), *of a*] **show** *?case* **by** *auto*
next
  case (*11* - *a*)
  with *splay-not-Leaf*[*OF 11*(*3*), *of a*] **show** *?case* **by** *auto*
next
  case (*14* - *a*)
  with *splay-not-Leaf*[*OF 14*(*3*), *of a*] **show** *?case* **by** *auto*
qed *fastforce+*

1.3.1 Verification of Is-in Test

To test if an element *a* is in *t*, first perform *splay a t*, then check if the root is *a*. One could put this into one function that returns both a new tree and the test result.

lemma *is-root-splay*: *bst t* $\Longrightarrow$ *is-root a* (*splay a t*) $\longleftrightarrow$ *a* $\in$ *set-tree t*
by(*auto simp add*: *is-root-def splay-to-root split*: *tree.split*)

1.3.2 Verification of *insert*

lemma *set-insert*: *set-tree*(*insert a t*) = *Set.insert a* (*set-tree t*)
apply(*cases t*)
 apply *simp*
using *set-splay*[*of a t*]
by(*simp split*: *tree.split*) *fastforce*

lemma *bst-insert*: *bst t* $\Longrightarrow$ *bst*(*insert a t*)
apply(*cases t*)
 apply *simp*
using *bst-splay*[*of t a*] *splay-bstL*[*of t a*] *splay-bstR*[*of t a*]
by(*auto simp*: *ball-Un split*: *tree.split*)

1.3.3 Verification of *splay-max*

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* $\Longrightarrow$ *size t* = *size l* + *size r* + *1*
by (*metis One-nat-def size-splay-max tree.size*(*4*))

lemma *set-splay-max*: *set-tree*(*splay-max t*) = *set-tree t*
apply(*induction t rule*: *splay-max.induct*)
   apply(*simp*)
  apply(*simp*)
apply(*force split*: *tree.split*)
done

lemma *bst-splay-max*: *bst t* $\Longrightarrow$ *bst* (*splay-max t*)
proof(*induction t rule*: *splay-max.induct*)
  case (*3 l b rl c rr*)
  { fix *rrl′ d′ rrr′*
    have *splay-max rr* = *Node rrl′ d′ rrr′*
      $\Longrightarrow$ $\forall$ *x* $\in$ *set-tree*(*Node rrl′ d′ rrr′*). *c* < *x*
      using *3.prems set-splay-max*[*of rr*]
      by (*clarsimp split*: *tree.split simp*: *ball-Un*)
  }
  with *3* **show** *?case* **by** (*fastforce split*: *tree.split simp*: *ball-Un*)
qed *auto*

lemma *splay-max-Leaf*: *splay-max t* = *Node l a r* $\Longrightarrow$ *r* = *Leaf*
by(*induction t arbitrary*: *l rule*: *splay-max.induct*)
  (*auto split*: *tree.splits if-splits*)

For sanity purposes only:

lemma *splay-max-eq-splay*:
  bst t $\Longrightarrow$ $\forall$ *x* $\in$ *set-tree t*. *x* $\leq$ *a* $\Longrightarrow$ *splay-max t* = *splay a t*
proof(*induction a t rule*: *splay.induct*)
  case (*2 a l r*)
  show *?case*
  proof (*cases r*)
    case *Leaf* **with** *2* **show** *?thesis* **by** *simp*
  next
    case *Node* **with** *2* **show** *?thesis* **by**(*auto*)
  qed
qed (*auto simp*: *neq-Leaf-iff*)

lemma *splay-max-eq-splay-ex*: **assumes** *bst t* **shows** $\exists$ *a*. *splay-max t* = *splay a t*
proof(*cases t*)
  case *Leaf* **thus** *?thesis* **by** *simp*

next
  case *Node*
  hence *splay-max t* = *splay* (*Max*(*set-tree t*)) *t*
    using *assms* **by** (*auto simp*: *splay-max-eq-splay*)
  thus *?thesis* **by** *auto*
qed

1.3.4 Verification of *delete*

lemma *set-delete*: **assumes** *bst t*
shows *set-tree* (*delete a t*) = *set-tree t* − $\{a\}$
proof(*cases t*)
  case *Leaf* **thus** *?thesis* **by**(*simp add*: *delete-def*)
next
  case (*Node l x r*)
  obtain l' x' r' **where** *sp*[*simp*]: *splay a* (*Node l x r*) = *Node* l' x' r'
    by (*metis neq-Leaf-iff splay-Leaf-iff*)
  show *?thesis*
  proof *cases*
    assume [*simp*]: $x' = a$
    show *?thesis*
    proof *cases*
      assume l' = *Leaf*
      thus *?thesis*
        using *Node assms set-splay*[*of a Node l x r*] *bst-splay*[*of Node l x r a*]
        by(*simp add*: *delete-def split*: *tree.split prod.split*)(*fastforce*)
    next
      assume $l' \neq$ *Leaf*
      moreover then obtain l'' *m* r'' **where** *splay-max* l' = *Node* l'' *m* r''
        using *splay-max-Leaf-iff tree.exhaust* **by** *blast*
      moreover have $a \notin$ *set-tree* l'
        by (*metis* (*no-types*) *Node assms less-irrefl sp splay-bstL*)
      ultimately show *?thesis*
        using *Node assms set-splay*[*of a Node l x r*] *bst-splay*[*of Node l x r a*]
          splay-max-Leaf[*of* l' l'' *m* r''] *set-splay-max*[*of* l']
        by(*clarsimp simp*: *delete-def split*: *tree.split*) *auto*
    qed
  next
    assume $x' \neq a$
    thus *?thesis* **using** *Node assms set-splay*[*of a Node l x r*] *splay-to-root*[*OF - sp*]
      by (*simp add*: *delete-def*)
  qed
qed

lemma *bst-delete*: **assumes** *bst t* **shows** *bst* (*delete a t*)
proof(*cases t*)
  case *Leaf* **thus** *?thesis* **by**(*simp add*: *delete-def*)
next
  case (*Node l x r*)

obtain l' x' r' **where** *sp*[*simp*]: *splay a* (*Node l x r*) = *Node* l' x' r'
  by (*metis neq-Leaf-iff splay-Leaf-iff*)
show *?thesis*
proof *cases*
  assume [*simp*]: $x' = a$
  show *?thesis*
  proof *cases*
    assume $l' = Leaf$
    thus *?thesis* **using** *Node assms bst-splay*[*of Node l x r a*]
      by(*simp add*: *delete-def split*: *tree.split prod.split*)
  next
    assume $l' \neq Leaf$
    thus *?thesis*
      using *Node assms set-splay*[*of a Node l x r*] *bst-splay*[*of Node l x r a*]
        bst-splay-max[*of l'*] *set-splay-max*[*of l'*]
      by(*clarsimp simp*: *delete-def split*: *tree.split*)
        (*metis* (*no-types*) *insertI1 less-trans*)
  qed
next
  assume $x' \neq a$
  thus *?thesis* **using** *Node assms bst-splay*[*of Node l x r a*]
    by(*auto simp*: *delete-def split*: *tree.split prod.split*)
qed
qed

end

2 Splay Tree Implementation of Maps

theory *Splay-Map*
imports
  Splay-Tree
  HOL−Data-Structures.Map-Specs
begin

function *splay* :: *'a::linorder* ⇒ (*'a*∗*'b*) *tree* ⇒ (*'a*∗*'b*) *tree* **where**
splay x Leaf = *Leaf* |
$x = fst\ a \Longrightarrow splay\ x\ (Node\ t1\ a\ t2) = Node\ t1\ a\ t2$ |
$x = fst\ a \Longrightarrow x < fst\ b \Longrightarrow splay\ x\ (Node\ (Node\ t1\ a\ t2)\ b\ t3) = Node\ t1\ a\ (Node\ t2\ b\ t3)$ |
$x < fst\ a \Longrightarrow splay\ x\ (Node\ Leaf\ a\ t) = Node\ Leaf\ a\ t$ |
$x < fst\ a \Longrightarrow x < fst\ b \Longrightarrow splay\ x\ (Node\ (Node\ Leaf\ a\ t1)\ b\ t2) = Node\ Leaf\ a\ (Node\ t1\ b\ t2)$ |
$x < fst\ a \Longrightarrow x < fst\ b \Longrightarrow t1 \neq Leaf \Longrightarrow$
 $splay\ x\ (Node\ (Node\ t1\ a\ t2)\ b\ t3) =$
 $(case\ splay\ x\ t1\ of\ Node\ t11\ y\ t12 \Rightarrow Node\ t11\ y\ (Node\ t12\ a\ (Node\ t2\ b\ t3)))$ |
$fst\ a < x \Longrightarrow x < fst\ b \Longrightarrow splay\ x\ (Node\ (Node\ t1\ a\ Leaf)\ b\ t2) = Node\ t1\ a\ (Node\ Leaf\ b\ t2)$ |
$fst\ a < x \Longrightarrow x < fst\ b \Longrightarrow t2 \neq Leaf \Longrightarrow$

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
                Node bll y blr ⇒ Node (Node al a bll) y (Node blr b br) |
        GT ⇒ if br=Leaf then Node (Node al a bl) b br
              else case splay x br of
                Node bll y blr ⇒ Node (Node (Node al a bl) b bll) y blr))))

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* ⇒ *None* | *Node* - (*a*,*b*) - ⇒ *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 ⇒ *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 ⇒
if x = *fst a*
then if l = *Leaf then r else case splay-max l of Node l′ m r′* ⇒ *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* ⟹ *sorted1*(*inorder t*) ⟹
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*) ⟹ *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*) ⟹ *splay x t* = *Node l a r* ⟹
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*) $\Longrightarrow$ *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* $\Longrightarrow$ *sorted1*(*inorder t*) $\Longrightarrow$
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*) $\Longrightarrow$ *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*)

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 ≤ p then
     case ar of
       Leaf ⇒ (Node al a ar, Leaf) |
       Node bl b br ⇒
         if b ≤ 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 ⇒ (Leaf, Node al a ar) |
       Node bl b br ⇒
         if b ≤ 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 ⇒ 'a tree ⇒ 'a tree where
insert x h = (let (l,r) = partition x h in Node l x r)

fun del-min :: 'a::linorder tree ⇒ '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 ≠ Leaf ⟹ get-min h ∈ set-tree h
by(induction h) auto

lemma get-min-min:
  ⟦ bst-wrt (≤) h; h ≠ Leaf ⟧ ⟹ ∀ x ∈ set-tree h. get-min h ≤ 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') ⟹ 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 (≤) t; partition p t = (l',r') ⟧
 ⟹ 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 ?case
  proof cases
    assume a ≤ p
```

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

lemma set-partition: ⟦ bst-wrt (≤) t; partition p t = (l′,r′) ⟧
 ⟹ set-tree t = set-tree l′ ∪ set-tree r′
by (metis mset-partition set-mset-tree set-mset-union)

lemma bst-partition:
  partition p t = (l′,r′) ⟹ bst-wrt (≤) t ⟹ bst-wrt (≤) (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 ≤ p
    show ?thesis
    proof (cases r)
      case Leaf thus ?thesis using ‹a ≤ p› 2.prems by fastforce
    next
      case (Node rl b rr)
      show ?thesis
      proof cases
        assume b ≤ p
        thus ?thesis
          using Node ‹a ≤ p› 2.prems 2.IH(1)[OF - Node] set-partition[of rr]
          by (fastforce split: prod.splits)
      next
        assume ¬ b ≤ p
        thus ?thesis
          using Node ‹a ≤ p› 2.prems 2.IH(2)[OF - Node] set-partition[of rl]
          by (fastforce split: prod.splits)
      qed
    qed
  next
    assume ¬ a ≤ p
    show ?thesis
    proof (cases l)
      case Leaf thus ?thesis using ‹¬ a ≤ p› 2.prems by fastforce
    next
      case (Node ll b lr)
      show ?thesis
      proof cases
        assume b ≤ p
        thus ?thesis
          using Node ‹¬ a ≤ p› 2.prems 2.IH(3)[OF - Node] set-partition[of lr]
          by (fastforce split: prod.splits)
      next
        assume ¬ b ≤ p
        thus ?thesis
          using Node ‹¬ a ≤ p› 2.prems 2.IH(4)[OF - Node] set-partition[of ll]
          by (fastforce split: prod.splits)
      qed
    qed
  qed
qed

lemma size-del-min[simp]: size(del-min t) = 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 (3 ll)
  show ?case
```

```
  proof cases
    assume ll = Leaf thus ?thesis using 3 by (simp add: ac-simps)
  next
    assume ll ≠ Leaf
    hence get-min ll ∈# mset-tree ll
      by (simp add: get-min-in)
    then obtain A where mset-tree ll = add-mset (get-min ll) A
      by (blast dest: multi-member-split)
    then show ?thesis using 3 by auto
  qed
qed auto

lemma bst-del-min: bst-wrt (≤) t ⟹ bst-wrt (≤) (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.