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

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* ⇒ *'a tree* ⇒ *'a tree* **where**
splay x Leaf = *Leaf* |
splay x (*Node AB x CD*) = *Node AB x CD* |
x<*b* ⟹ *splay x* (*Node* (*Node A x B*) *b CD*) = *Node A x* (*Node B b CD*) |
x<*b* ⟹ *splay x* (*Node Leaf b CD*) = *Node Leaf b CD* |
x<*a* ⟹ *x*<*b* ⟹ *splay x* (*Node* (*Node Leaf a B*) *b CD*) = *Node Leaf a* (*Node B b CD*) |
x<*a* ⟹ *x*<*b* ⟹ *A* ≠ *Leaf* ⟹
splay x (*Node* (*Node A a B*) *b CD*) =
(*case splay x A of Node A1 a' A2* ⇒ *Node A1 a'* (*Node A2 a* (*Node B b CD*))) |
a<*x* ⟹ *x*<*b* ⟹ *splay x* (*Node* (*Node A a Leaf*) *b CD*) = *Node A a* (*Node Leaf b CD*) |
a<*x* ⟹ *x*<*b* ⟹ *B* ≠ *Leaf* ⟹
splay x (*Node* (*Node A a B*) *b CD*) =
(*case splay x B of Node B1 b' B2* ⇒ *Node* (*Node A a B1*) *b'* (*Node B2 b CD*)) |
b<*x* ⟹ *splay x* (*Node AB b* (*Node C x D*)) = *Node* (*Node AB b C*) *x D* |
b<*x* ⟹ *splay x* (*Node AB b Leaf*) = *Node AB b Leaf* |
b<*x* ⟹ *x*<*c* ⟹ *C* ≠ *Leaf* ⟹
splay x (*Node AB b* (*Node C c D*)) =
(*case splay x C of Node C1 c' C* ⇒ *Node* (*Node AB b C1*) *c'* (*Node C c D*)) |
b<*x* ⟹ *x*<*c* ⟹ *splay x* (*Node AB b* (*Node Leaf c D*)) = *Node* (*Node AB b Leaf*) *c D* |
b<*x* ⟹ *c*<*x* ⟹ *splay x* (*Node AB b* (*Node C c Leaf*)) = *Node* (*Node AB b C*) *c Leaf* |
a<*x* ⟹ *c*<*x* ⟹ *D* ≠ *Leaf* ⟹
splay x (*Node AB a* (*Node C c D*)) =
(*case splay x D of Node D1 d' D2* ⇒ *Node* (*Node* (*Node AB a C*) *c D1*) *d' D2*)
⟨*proof*⟩

termination *splay*
⟨*proof*⟩

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)))
⟨proof⟩

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 ⇒ 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)))
```
⟨*proof*⟩

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 Node l a r*
     else if l = *Leaf then r else case splay-max l of Node l' m r'* ⇒ *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*)
⟨*proof*⟩

lemma *splay-max-Leaf-iff*[*simp*]: (*splay-max t* = *Leaf*) = (*t* = *Leaf*)
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *isin-set*: *sorted*(*inorder t*) ⟹ *isin t x* = (*x* ∈ *set* (*inorder t*))
⟨*proof*⟩

1.2.2 Verification of *insert*

lemma *inorder-splay*: *inorder*(*splay x t*) = *inorder t*
⟨*proof*⟩

lemma *sorted-splay*:
  sorted(*inorder t*) ⟹ *splay x t* = *Node l a r* ⟹
  sorted(*inorder l* @ *x* # *inorder r*)
⟨*proof*⟩

lemma *inorder-insert*:
  sorted(*inorder t*) ⟹ *inorder*(*insert x t*) = *ins-list x* (*inorder t*)
⟨*proof*⟩

1.2.3 Verification of *delete*

lemma *inorder-splay-maxD*:

splay-max t = Node l a r ⟹ sorted(inorder t) ⟹
inorder l @ [a] = inorder t ∧ r = Leaf
⟨*proof*⟩

lemma *inorder-delete*:
sorted(inorder t) ⟹ inorder(delete x t) = del-list x (inorder t)
⟨*proof*⟩

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*⟩

Corollaries:

lemma *bst-splay*: *bst t ⟹ bst (splay x t)*
⟨*proof*⟩

lemma *bst-insert*: *bst t ⟹ bst(insert x t)*
⟨*proof*⟩

lemma *bst-delete*: *bst t ⟹ bst(delete x t)*
⟨*proof*⟩

lemma *splay-bstL*: *bst t ⟹ splay a t = Node l e r ⟹ x ∈ set-tree l ⟹ x < a*
⟨*proof*⟩

lemma *splay-bstR*: *bst t ⟹ splay a t = Node l e r ⟹ x ∈ set-tree r ⟹ a < x*
⟨*proof*⟩

1.2.5 Size lemmas

lemma *size-splay*[*simp*]: *size (splay a t) = size t*
⟨*proof*⟩

lemma *size-if-splay*: *splay a t = Node l u r ⟹ size t = size l + size r + 1*
⟨*proof*⟩

lemma *splay-not-Leaf*: *t ≠ Leaf ⟹ ∃ l x r. splay a t = Node l x r*
⟨*proof*⟩

lemma *size-splay-max*: *size(splay-max t) = size t*
⟨*proof*⟩

lemma *size-if-splay-max*: *splay-max t = Node l u r ⟹ size t = size l + size r + 1*
⟨*proof*⟩

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* ⟹ *splay x* (*Node t1 a t2*) = *Node t1 a t2* |
x = *fst a* ⟹ *x* < *fst b* ⟹ *splay x* (*Node* (*Node t1 a t2*) *b t3*) = *Node t1 a* (*Node t2 b t3*) |
x < *fst a* ⟹ *splay x* (*Node Leaf a t*) = *Node Leaf a t* |
x < *fst a* ⟹ *x* < *fst b* ⟹ *splay x* (*Node* (*Node Leaf a t1*) *b t2*) = *Node Leaf a* (*Node t1 b t2*) |
x < *fst a* ⟹ *x* < *fst b* ⟹ *t1* ≠ *Leaf* ⟹
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*)
⟨*proof*⟩

termination *splay*
⟨*proof*⟩

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))))
⟨proof⟩

definition lookup :: ('a*'b)tree ⇒ 'a::linorder ⇒ '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 ⇒ 'b ⇒ ('a*'b) tree ⇒ ('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 ⇒ ('a*'b) tree ⇒ ('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)
⟨proof⟩
```

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)
```

⟨*proof*⟩

lemma *lookup-eq*:
sorted1(*inorder t*) ⟹ *lookup t x* = *map-of* (*inorder t*) *x*
⟨*proof*⟩

2.1.2 Proofs for update

lemma *inorder-splay*: *inorder*(*splay x t*) = *inorder t*
⟨*proof*⟩

lemma *sorted-splay*:
sorted1(*inorder t*) ⟹ *splay x t* = *Node l a r* ⟹
sorted(*map fst* (*inorder l*) @ *x* # *map fst* (*inorder r*))
⟨*proof*⟩

lemma *inorder-update-splay*:
sorted1(*inorder t*) ⟹ *inorder*(*update x y t*) = *upd-list x y* (*inorder t*)
⟨*proof*⟩

2.1.3 Proofs for delete

lemma *inorder-splay-maxD*:
splay-max t = *Node l a r* ⟹ *sorted1*(*inorder t*) ⟹
inorder l @ [*a*] = *inorder t* ∧ *r* = *Leaf*
⟨*proof*⟩

lemma *inorder-delete-splay*:
sorted1(*inorder t*) ⟹ *inorder*(*delete x t*) = *del-list x* (*inorder t*)
⟨*proof*⟩

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*⟩

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* ⇒ *'a* **where**
get-min(*Node l m r*) = (*if l* = *Leaf then m else get-min l*)

fun *partition* :: *'a*::*linorder* ⇒ *'a tree* ⇒ *'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*
⟨*proof*⟩

lemma *get-min-min*:
⟦ *bst-wrt* (≤) *h*; *h* ≠ *Leaf* ⟧ ⟹ ∀ *x* ∈ *set-tree h*. *get-min h* ≤ *x*
⟨*proof*⟩

lemma *size-partition*: *partition p t* = (*l'*,*r'*) ⟹ *size t* = *size l'* + *size r'*
⟨*proof*⟩

lemma *mset-partition*: ⟦ *bst-wrt* (≤) *t*; *partition p t* = (*l'*,*r'*) ⟧
⟹ *mset-tree t* = *mset-tree l'* + *mset-tree r'*
⟨*proof*⟩

lemma *set-partition*: ⟦ *bst-wrt* (≤) *t*; *partition p t* = (*l'*,*r'*) ⟧
⟹ *set-tree t* = *set-tree l'* ∪ *set-tree r'*
⟨*proof*⟩

lemma *bst-partition*:
partition p t = (*l'*,*r'*) ⟹ *bst-wrt* (≤) *t* ⟹ *bst-wrt* (≤) (*Node l' p r'*)
⟨*proof*⟩

lemma *size-del-min*[*simp*]: *size*(*del-min t*) = *size t* − *1*
⟨*proof*⟩

lemma *mset-del-min*: *mset-tree* (*del-min h*) = *mset-tree h* − {# *get-min h* #}
⟨*proof*⟩

lemma *bst-del-min*: *bst-wrt* (≤) *t* ⟹ *bst-wrt* (≤) (*del-min t*)
⟨*proof*⟩

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.