Splay Tree
Tobias Nipkow
September 19, 2015

Abstract
Splay trees are self-adjusting binary search trees which were invented by Sleator and Tarjan [1]. This entry provides executable and verified functional splay trees.

The amortized complexity of splay trees is analyzed in the AFP entry Amortized Complexity.

Contents

1 Splay Tree 1
1.1 Function splay ................................. 2
1.2 Is-in Test ........................................ 5
1.3 Function insert ................................. 5
1.4 Function splay-max ............................. 6
1.5 Function delete ................................. 7

theory Splay-Tree
imports ~/src/HOL/Library/Tree
begin

1 Splay Tree

Splay trees were invented by Sleator and Tarjan [1].

This compensates for an incompleteness of the partial order prover:

simproc-setup less-False ((x::'a::order) < y) = (∥ fn - => fn ctx => fn ct =>
let
  fun prp t thm = Thm.full-prop-of thm aconv t;

  val eq-False-if-not = @\{thm eq-False\} RS iffD2

  fun prove-less-False ((less as Const(_,T)) $ r $ s) =
  let val prems = Simplifier.prems-of ctxt;
    val le = Const (@\{const-name less-eq\}, T);
    val t = HOLogic.mk_Trueprop(le $ s $ r);
  in case find-first (prp t) prems of


NONE =>
  let val t = HOLogic.mk_Trueprop(less $ s $ r)
in case find-first (prp t) prems of
    NONE => NONE
    | SOME thm => SOME(mk-meta-eq((thm RS @\{thm less-not-sym\})) end
end
in prove-less-False (Thm.term_of ct) end
⟩⟩

1.1 Function splay

function splay :: 'a::linorder ⇒ 'a tree ⇒ 'a tree where
splay a Leaf = Leaf |
splay a (Node l a r) = Node l a r |
a<b ⇒ splay a (Node (Node ll a lr) b r) = Node ll a (Node lr b r) |
a<b ⇒ splay a (Node Leaf b r) = Node Leaf b r |
a<c ⇒ a<b ⇒ splay a (Node (Node Leaf b lr) c r) = Node Leaf b (Node lr c r) |
a<c ⇒ a<b ⇒ ll ≠ Leaf ⇒
splay a (Node (Node ll b lr) c r) =
  (case splay a ll of Node ll u llr ⇒ Node ll u (Node llr b (Node lr c r))) |
a<c ⇒ b<a ⇒ splay a (Node (Node ll b Leaf) c r) = Node ll b (Node Leaf c r) |
a<c ⇒ b<a ⇒ lr ≠ Leaf ⇒
splay a (Node (Node ll b lr) c r) =
  (case splay a lr of Node lrl u lrr ⇒ Node (Node ll b lrl) u (Node lrr b c r)) |
b<a ⇒ splay a (Node l b (Node rl a rr)) = Node (Node l b rl) a rr |
b<a ⇒ splay a (Node l b Leaf) = Node l b Leaf |
c<a ⇒ a<b ⇒ rl ≠ Leaf ⇒
splay a (Node l c (Node rl b rr)) =
  (case splay a rl of Node rl1 u rrl ⇒ Node (Node l c rl1) u (Node rrl b rr)) |
c<a ⇒ a<b ⇒ splay a (Node l c (Node Leaf b rr)) = Node (Node l c Leaf) b rr |
c<a ⇒ b<a ⇒ splay a (Node l c (Node rl b Leaf)) = Node (Node l c rl) b Leaf |
c<a ⇒ b<a ⇒ rr ≠ Leaf ⇒
splay a (Node l c (Node rl b rr)) =
  (case splay a rr of Node rrl u rrr ⇒ Node (Node l c rl) b rrl) u rrr)
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 a (Node cl c cr) =
  (if a=c
   then Node cl c cr
   else if a < c
     then case cl of
        Leaf ⇒ Node cl c cr |
        Node bl b br ⇒
          (if a=b then Node bl a (Node br c cr)
           else if a < b
             then if bl = Leaf then Node bl b (Node br c cr)
               else case splay a bl of
                 Node al a' ar ⇒ Node al a' (Node ar b (Node br c cr))
               else if br = Leaf then Node bl b (Node br c cr)
               else case splay a br of
                 Node al a' ar ⇒ Node (Node bl b al) a' (Node ar c cr))
     else case cr of
        Leaf ⇒ Node cl c cr |
        Node bl b br ⇒
          (if a=b then Node (Node cl c bl) a br
           else if a < b
             then if bl = Leaf then Node (Node cl c bl) b br
               else case splay a bl of
                 Node al a' ar ⇒ Node (Node cl c al) a' (Node ar b br)
               else if br=Leaf then Node (Node cl c bl) b br
               else case splay a br of
                 Node al a' ar ⇒ Node (Node (Node cl c bl) b al) a' ar))
  by (auto split: tree.split)

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

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)
next
  case (8 a - - t)
next
  case (11 - a - t)
next
  case (14 - a - t)
qed auto

lemma splay-to-root: [ bst t; splay a t = t’ ] =⇒ a ∈ set-tree t =⇒ (∃ l r. t’ = Node l a r)
proof(induction a t arbitrary: t’ rule: splay.induct)
case \((6 \ a)\)
with \texttt{splay-not-Leaf}\([OF \ 6(3), \ of \ a]\) show ?case by auto
next
\[\text{case } (8 \ a)\]
with \texttt{splay-not-Leaf}\([OF \ 8(3), \ of \ a]\) show ?case by auto
next
\[\text{case } (11 - a)\]
with \texttt{splay-not-Leaf}\([OF \ 11(3), \ of \ a]\) show ?case by auto
next
\[\text{case } (14 - a)\]
with \texttt{splay-not-Leaf}\([OF \ 14(3), \ of \ a]\) show ?case by auto
qed \texttt{fastforce+}

\subsection{1.2 \textit{Is-in} Test}

To test if an element \(a\) is in \(t\), first perform \texttt{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.

\texttt{definition is-root :: 'a ⇒ 'a tree ⇒ bool where}
\[\texttt{is-root a t} = (\texttt{case t of Leaf ⇒ False | Node - x - ⇒ x = a})\]

\texttt{lemma is-root-splay: bst t ⇒ is-root a (splay a t) ↔ a ∈ set-tree t}
\[\texttt{by(auto simp add: is-root-def splay-to-root split: tree.split)}\]

\subsection{1.3 Function \textit{insert}}

\texttt{context begin}

\texttt{qualified fun insert :: 'a::linorder ⇒ 'a tree ⇒ 'a tree where}
\[\texttt{insert a t} = (\texttt{if t = Leaf then Node Leaf a Leaf}
\texttt{else case splay a t of}
\texttt{Node l a' r ⇒ if a=a' then Node l a r}
\texttt{else if a<a' then Node l a (Node Leaf a' r) else Node (Node l a' Leaf) a r})\]

\texttt{end}

\texttt{lemma set-insert: set-tree(Splay-Tree.insert a t) = insert a (set-tree t)}
\[\texttt{apply(cases t)}\]
\[\texttt{apply simp}\]
\[\texttt{using set-splay[of a t]}\]
\[\texttt{by(simp split: tree.split) fastforce}\]

\texttt{lemma bst-insert: bst t ⇒ bst(Splay-Tree.insert a t)}
\[\texttt{apply(cases t)}\]
\[\texttt{apply simp}\]
\[\texttt{using bst-splay[of t a] splay-bstL[of t a] splay-bstR[of t a]}\]
\[\texttt{by(auto simp: ball-Un split: tree.split)}\]
1.4 Function splay-max

fun splay-max :: 'a::linorder tree ⇒ 'a tree where
splay-max Leaf = Leaf |
splay-max (Node l b Leaf) = Node l b Leaf |
splay-max (Node l b (Node rl c rr)) =
  (if rr = Leaf then Node (Node l b rl) c Leaf
   else case splay-max rr of
      Node rrl m rrr ⇒ Node (Node (Node l b rl) c rrl) m rrr)
lemma splay-max-Leaf-iff [simp]: (splay-max t = Leaf) = (t = Leaf)
apply (induction t rule: splay-max.induct)
apply (auto split: tree.splits)
done

lemma splay-max-code: splay-max t = (case t of
  Leaf ⇒ t |
  Node l b r ⇒ (case r of
    Leaf ⇒ t |
    Node rl c rr ⇒
      (if rr = Leaf then Node (Node l b rl) c rr
       else case splay-max rr of
            Node rrl u rrr ⇒ Node (Node (Node l b rl) c rrl) u rrr)))))
by (auto simp: neq-Leaf-iff split: tree.split)

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 ⇒ 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 ⇒ 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'
      ⇒ !x : 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 ⇒ 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 ⇒ ∀ x ∈ set-tree t. x ≤ a ⇒ 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 ∃ 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

1.5 Function delete

context
begin

qualified definition delete :: 'a::linorder ⇒ 'a tree ⇒ 'a tree where
delete a t = (if t=Leaf then Leaf
  else case splay a t of Node l a' r ⇒
    if a=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)

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 \(sp[\text{simp}]: \text{splay} a \ (\text{Node} \ l \ x \ r) = \text{Node} \ l' \ x' \ r'\)
by (metis neq-Leaf-iff splay-Leaf-iff)
show ?thesis
proof cases
assume [simp]: \(x' = a\)
show ?thesis
proof cases
assume \(l' = \text{Leaf}\)
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 \(l' \neq \text{Leaf}\)
moreover then obtain \(l'' \ m \ r''\) where \(\text{splay-max} \ l' = \text{Node} \ l'' \ m \ r''\)
using splay-max-Leaf-iff tree.exhaust by blast
moreover have \(a \notin \text{set-tree} \ l'\)
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]
\(\text{splay-max-Leaf of } l''\)
\(\text{set-splay-max of } l''\)
by(clarsimp simp: delete-def split: tree.split) auto
qed
next
assume \(x' \neq a\)
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 \(sp[\text{simp}]: \text{splay} a \ (\text{Node} \ l \ x \ r) = \text{Node} \ l' \ x' \ r'\)
by (metis neq-Leaf-iff splay-Leaf-iff)
show ?thesis
proof cases
assume [simp]: \(x' = a\)
show ?thesis
proof cases
assume \(l' = \text{Leaf}\)
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 \(l' \neq \text{Leaf}\)
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 \(\text{thesis using Node assms bst-splay[of Node \ l \ x \ r \ a]}\)
  by (auto simp: delete-def split: tree.split prod.split)
qed

References