Randomised Binary Search Trees

Manuel Eberl

September 23, 2021

Abstract

This work is a formalisation of the Randomised Binary Search Trees introduced by Martínez and Roura [1], including definitions and correctness proofs. Like randomised treaps, they are a probabilistic data structure that behaves exactly as if elements were inserted into a non-balancing BST in random order. However, unlike treaps, they only use discrete probability distributions, but their use of randomness is more complicated.

Contents

1 Randomised Binary Search Trees

theory *Randomised-BSTs*
imports *Random-BSTs.Random-BSTs Monad-Normalisation.Monad-Normalisation*
begin

1.1 Auxiliary facts

First of all, we need some fairly simple auxiliary lemmas.

lemma *return-pmf-if*: *return-pmf* (*if P then a else b*) = (*if P then return-pmf a else return-pmf b*)
⟨*proof*⟩

context
begin

interpretation *pmf-as-function* ⟨*proof*⟩

lemma *True-in-set-bernoulli-pmf-iff* [*simp*]:
True ∈ *set-pmf* (*bernoulli-pmf p*) ⟷ *p* > *0*
⟨*proof*⟩

lemma *False-in-set-bernoulli-pmf-iff* [*simp*]:
False ∈ *set-pmf* (*bernoulli-pmf p*) ⟷ *p* < *1*
⟨*proof*⟩

end

lemma *in-set-pmf-of-setD*: *x* ∈ *set-pmf* (*pmf-of-set A*) ⟹ *finite A* ⟹ *A* ≠ {} ⟹ *x* ∈ *A*
⟨*proof*⟩

lemma *random-bst-reduce*:
finite A ⟹ *A* ≠ {} ⟹
random-bst A = *do* {*x* ← *pmf-of-set A*; *l* ← *random-bst* {*y*∈*A*. *y* < *x*};
r ← *random-bst* {*y*∈*A*. *y* > *x*}; *return-pmf* ⟨*l*, *x*, *r*⟩}
⟨*proof*⟩

lemma *pmf-bind-bernoulli*:
assumes *x* ∈ {*0..1*}
shows *pmf* (*bernoulli-pmf x* ⤜ *f*) *y* = *x* ∗ *pmf* (*f True*) *y* + (*1* − *x*) ∗ *pmf* (*f False*) *y*
⟨*proof*⟩

lemma *vimage-bool-pair*:
f −‘ *A* = (⋃*x*∈{*True*, *False*}. ⋃*y*∈{*True*, *False*}. *if f* (*x*, *y*) ∈ *A then* {(*x*, *y*)} *else* {})
(**is** *?lhs* = *?rhs*) ⟨*proof*⟩

lemma *Leaf-in-set-random-bst-iff* [*simp*]:
Leaf ∈ *set-pmf* (*random-bst A*) ⟷ *A* = {} ∨ ¬*finite A*
⟨*proof*⟩

lemma *bst-insert* [*intro*]: *bst t* ⟹ *bst* (*Tree-Set.insert x t*)
⟨*proof*⟩

lemma *bst-bst-of-list* [*intro*]: *bst* (*bst-of-list xs*)
⟨*proof*⟩

lemma *bst-random-bst*:
assumes *t* ∈ *set-pmf* (*random-bst A*)
shows *bst t*
⟨*proof*⟩

lemma *set-random-bst*:
assumes *t* ∈ *set-pmf* (*random-bst A*) *finite A*
shows *set-tree t* = *A*
⟨*proof*⟩

lemma *isin-bst*:
assumes *bst t*
shows *isin t x* ⟷ *x* ∈ *set-tree t*
⟨*proof*⟩

lemma *isin-random-bst*:
assumes *finite A t* ∈ *set-pmf* (*random-bst A*)
shows *isin t x* ⟷ *x* ∈ *A*
⟨*proof*⟩

lemma *card-3way-split*:
assumes *x* ∈ (*A* :: *'a* :: *linorder set*) *finite A*
shows *card A* = *card* {*y*∈*A*. *y* < *x*} + *card* {*y*∈*A*. *y* > *x*} + *1*
⟨*proof*⟩

The following theorem allows splitting a uniformly random choice from a union of two disjoint sets to first tossing a coin to decide on one of the constituent sets and then chooing an element from it uniformly at random.

lemma *pmf-of-set-union-split*:
assumes *finite A finite B A* ∩ *B* = {} *A* ∪ *B* ≠ {}
assumes *p* = *card A* / (*card A* + *card B*)
shows *do* {*b* ← *bernoulli-pmf p*; *if b then pmf-of-set A else pmf-of-set B*} = *pmf-of-set* (*A* ∪ *B*)
(**is** *?lhs* = *?rhs*)
⟨*proof*⟩

lemma *pmf-of-set-split-inter-diff*:
assumes *finite A finite B A* ≠ {} *B* ≠ {}
assumes *p* = *card* (*A* ∩ *B*) / *card B*

shows *do* {$b \leftarrow$ *bernoulli-pmf p*; *if b then pmf-of-set* $(A \cap B)$ *else pmf-of-set* $(B - A)$} =
pmf-of-set B (**is** *?lhs* = *?rhs*)
⟨*proof*⟩

Similarly to the above rule, we can split up a uniformly random choice from the disjoint union of three sets. This could be done with two coin flips, but it is more convenient to choose a natural number uniformly at random instead and then do a case distinction on it.

lemma *pmf-of-set-3way-split*:
fixes *f g h* :: *'a* ⇒ *'b pmf*
assumes *finite A A* ≠ {} *A1* ∩ *A2* = {} *A1* ∩ *A3* = {} *A2* ∩ *A3* = {} *A1* ∪ *A2* ∪ *A3* = *A*
shows *do* {*x* ← *pmf-of-set A*; *if x* ∈ *A1 then f x else if x* ∈ *A2 then g x else h x*} =
do {*i* ← *pmf-of-set* {..<*card A*};
if i < *card A1 then pmf-of-set A1* ⤜ *f*
else if i < *card A1* + *card A2 then pmf-of-set A2* ⤜ *g*
else pmf-of-set A3 ⤜ *h*} (**is** *?lhs* = *?rhs*)
⟨*proof*⟩

1.2 Partitioning a BST

The split operation takes a search parameter x and partitions a BST into two BSTs containing all the values that are smaller than x and those that are greater than x, respectively. Note that x need not be an element of the tree.

fun *split-bst* :: *'a* :: *linorder* ⇒ *'a tree* ⇒ *'a tree* × *'a tree* **where**
split-bst - ⟨⟩ = (⟨⟩, ⟨⟩)
| *split-bst x* ⟨*l*, *y*, *r*⟩ =
(*if y* < *x then*
case split-bst x r of (*t1*, *t2*) ⇒ (⟨*l*, *y*, *t1*⟩, *t2*)
else if y > *x then*
case split-bst x l of (*t1*, *t2*) ⇒ (*t1*, ⟨*t2*, *y*, *r*⟩)
else
(*l*, *r*))

fun *split-bst'* :: *'a* :: *linorder* ⇒ *'a tree* ⇒ *bool* × *'a tree* × *'a tree* **where**
split-bst' - ⟨⟩ = (*False*, ⟨⟩, ⟨⟩)
| *split-bst' x* ⟨*l*, *y*, *r*⟩ =
(*if y* < *x then*
case split-bst' x r of (*b*, *t1*, *t2*) ⇒ (*b*, ⟨*l*, *y*, *t1*⟩, *t2*)
else if y > *x then*
case split-bst' x l of (*b*, *t1*, *t2*) ⇒ (*b*, *t1*, ⟨*t2*, *y*, *r*⟩)
else
(*True*, *l*, *r*))

lemma *split-bst'-altdef*: *split-bst' x t* = (*isin t x*, *split-bst x t*)

⟨*proof*⟩

lemma *fst-split-bst'* [*simp*]: *fst* (*split-bst'* *x* *t*) = *isin* *t* *x*
and *snd-split-bst'* [*simp*]: *snd* (*split-bst'* *x* *t*) = *split-bst* *x* *t*
⟨*proof*⟩

lemma *size-fst-split-bst* [*termination-simp*]: *size* (*fst* (*split-bst* *x* *t*)) ≤ *size* *t*
⟨*proof*⟩

lemma *size-snd-split-bst* [*termination-simp*]: *size* (*snd* (*split-bst* *x* *t*)) ≤ *size* *t*
⟨*proof*⟩

lemmas *size-split-bst* = *size-fst-split-bst size-snd-split-bst*

lemma *set-split-bst1*: *bst* *t* ⟹ *set-tree* (*fst* (*split-bst* *x* *t*)) = {*y* ∈ *set-tree* *t*. *y* < *x*}
⟨*proof*⟩

lemma *set-split-bst2*: *bst* *t* ⟹ *set-tree* (*snd* (*split-bst* *x* *t*)) = {*y* ∈ *set-tree* *t*. *y* > *x*}
⟨*proof*⟩

lemma *bst-split-bst1* [*intro*]: *bst* *t* ⟹ *bst* (*fst* (*split-bst* *x* *t*))
⟨*proof*⟩

lemma *bst-split-bst2* [*intro*]: *bst* *t* ⟹ *bst* (*snd* (*split-bst* *x* *t*))
⟨*proof*⟩

Splitting a random BST produces two random BSTs:

theorem *split-random-bst*:
assumes *finite* *A*
shows *map-pmf* (*split-bst* *x*) (*random-bst* *A*) =
pair-pmf (*random-bst* {*y*∈*A*. *y* < *x*}) (*random-bst* {*y*∈*A*. *y* > *x*})
⟨*proof*⟩
include *monad-normalisation*

⟨*proof*⟩

1.3 Joining

The "join" operation computes the union of two BSTs l and r where all the values in l are stricly smaller than those in r.

fun *mrbst-join* :: *'a tree* ⇒ *'a tree* ⇒ *'a tree pmf* **where**
mrbst-join t1 t2 =
(*if t1* = ⟨⟩ *then return-pmf t2*
else if t2 = ⟨⟩ *then return-pmf t1*
else do {

```
    b ← bernoulli-pmf (size t1 / (size t1 + size t2));
    if b then
      (case t1 of ⟨l, x, r⟩ ⇒ map-pmf (λr'. ⟨l, x, r'⟩) (mrbst-join r t2))
    else
      (case t2 of ⟨l, x, r⟩ ⇒ map-pmf (λl'. ⟨l', x, r⟩) (mrbst-join t1 l))
  })
```

lemma *mrbst-join-Leaf-left* [*simp*]: *mrbst-join* ⟨⟩ = *return-pmf*
⟨*proof*⟩

lemma *mrbst-join-Leaf-right* [*simp*]: *mrbst-join t* ⟨⟩ = *return-pmf t*
⟨*proof*⟩

lemma *mrbst-join-reduce*:
t1 ≠ ⟨⟩ ⟹ *t2* ≠ ⟨⟩ ⟹ *mrbst-join t1 t2* =
do {
b ← *bernoulli-pmf* (*size t1* / (*size t1* + *size t2*));
if b then
(*case t1 of* ⟨*l*, *x*, *r*⟩ ⇒ *map-pmf* (λ*r'*. ⟨*l*, *x*, *r'*⟩) (*mrbst-join r t2*))
else
(*case t2 of* ⟨*l*, *x*, *r*⟩ ⇒ *map-pmf* (λ*l'*. ⟨*l'*, *x*, *r*⟩) (*mrbst-join t1 l*))
}
⟨*proof*⟩

lemmas [*simp del*] = *mrbst-join.simps*

lemma
assumes *t'* ∈ *set-pmf* (*mrbst-join t1 t2*) *bst t1 bst t2*
assumes ⋀*x y*. *x* ∈ *set-tree t1* ⟹ *y* ∈ *set-tree t2* ⟹ *x* < *y*
shows *bst-mrbst-join*: *bst t'*
and *set-mrbst-join*: *set-tree t'* = *set-tree t1* ∪ *set-tree t2*
⟨*proof*⟩

Joining two random BSTs that satisfy the necessary preconditions again yields a random BST.

theorem *mrbst-join-correct*:
fixes *A B* :: *'a* :: *linorder set*
assumes *finite A finite B* ⋀*x y*. *x* ∈ *A* ⟹ *y* ∈ *B* ⟹ *x* < *y*
shows *do* {*t1* ← *random-bst A*; *t2* ← *random-bst B*; *mrbst-join t1 t2*} = *random-bst* (*A* ∪ *B*)
⟨*proof*⟩

include *monad-normalisation*
⟨*proof*⟩

1.4 Pushdown

The "push down" operation "forgets" information about the root of a tree in the following sense: It takes a non-empty tree whose root is some known

fixed value and whose children are random BSTs and shuffles the root in such a way that the resulting tree is a random BST.

fun *mrbst-push-down* :: *'a tree* ⇒ *'a* ⇒ *'a tree* ⇒ *'a tree pmf* **where**
mrbst-push-down l x r =
do {
k ← *pmf-of-set* {*0..size l* + *size r*};
if k < *size l then*
case l of
⟨*ll*, *y*, *lr*⟩ ⇒ *map-pmf* (λ*r'*. ⟨*ll*, *y*, *r'*⟩) (*mrbst-push-down lr x r*)
else if k < *size l* + *size r then*
case r of
⟨*rl*, *y*, *rr*⟩ ⇒ *map-pmf* (λ*l'*. ⟨*l'*, *y*, *rr*⟩) (*mrbst-push-down l x rl*)
else
return-pmf ⟨*l*, *x*, *r*⟩
}

lemmas [*simp del*] = *mrbst-push-down.simps*

lemma
assumes *t'* ∈ *set-pmf* (*mrbst-push-down t1 x t2*) *bst t1 bst t2*
assumes ⋀*y*. *y* ∈ *set-tree t1* ⟹ *y* < *x* ⋀*y*. *y* ∈ *set-tree t2* ⟹ *y* > *x*
shows *bst-mrbst-push-down*: *bst t'*
and *set-mrbst-push-down*: *set-tree t'* = {*x*} ∪ *set-tree t1* ∪ *set-tree t2*
⟨*proof*⟩

theorem *mrbst-push-down-correct*:
fixes *A B* :: *'a* :: *linorder set*
assumes *finite A finite B* ⋀*y*. *y* ∈ *A* ⟹ *y* < *x* ⋀*y*. *y* ∈ *B* ⟹ *x* < *y*
shows *do* {*l* ← *random-bst A*; *r* ← *random-bst B*; *mrbst-push-down l x r*} =
random-bst ({*x*} ∪ *A* ∪ *B*)
⟨*proof*⟩

include *monad-normalisation*
⟨*proof*⟩

lemma *mrbst-push-down-correct'*:
assumes *finite* (*A* :: *'a* :: *linorder set*) *x* ∈ *A*
shows *do* {*l* ← *random-bst* {*y*∈*A*. *y* < *x*}; *r* ← *random-bst* {*y*∈*A*. *y* > *x*};
mrbst-push-down l x r} =
random-bst A (**is** *?lhs* = *?rhs*)
⟨*proof*⟩

1.5 Intersection and Difference

The algorithms for intersection and difference of two trees are almost identical; the only difference is that the "if" statement at the end of the recursive case is flipped. We therefore introduce a generic intersection/difference operation first and prove its correctness to avoid duplication.

```
fun mrbst-inter-diff where
  mrbst-inter-diff - ⟨⟩ - = return-pmf ⟨⟩
| mrbst-inter-diff b ⟨l1, x, r1⟩ t2 =
    (case split-bst' x t2 of (sep, l2, r2) ⇒
      do {
        l ← mrbst-inter-diff b l1 l2;
        r ← mrbst-inter-diff b r1 r2;
        if sep = b then return-pmf ⟨l, x, r⟩ else mrbst-join l r
      })

lemma mrbst-inter-diff-reduce:
  mrbst-inter-diff b ⟨l1, x, r1⟩ =
    (λt2. case split-bst' x t2 of (sep, l2, r2) ⇒
      do {
        l ← mrbst-inter-diff b l1 l2;
        r ← mrbst-inter-diff b r1 r2;
        if sep = b then return-pmf ⟨l, x, r⟩ else mrbst-join l r
       })
  ⟨proof⟩

lemma mrbst-inter-diff-Leaf-left [simp]:
  mrbst-inter-diff b ⟨⟩ = (λ-. return-pmf ⟨⟩)
  ⟨proof⟩

lemma mrbst-inter-diff-Leaf-right [simp]:
  mrbst-inter-diff b (t1 :: 'a :: linorder tree) ⟨⟩ = return-pmf (if b then ⟨⟩ else t1)
  ⟨proof⟩

lemma
  fixes t1 t2 :: 'a :: linorder tree and b :: bool
  defines setop ≡ (if b then (∩) else (−) :: 'a set ⇒ -)
  assumes t' ∈ set-pmf (mrbst-inter-diff b t1 t2) bst t1 bst t2
  shows   bst-mrbst-inter-diff: bst t'
    and   set-mrbst-inter-diff: set-tree t' = setop (set-tree t1) (set-tree t2)
⟨proof⟩

theorem mrbst-inter-diff-correct:
  fixes A B :: 'a :: linorder set and b :: bool
  defines setop ≡ (if b then (∩) else (−) :: 'a set ⇒ -)
  assumes finite A finite B
  shows   do {t1 ← random-bst A; t2 ← random-bst B; mrbst-inter-diff b t1 t2}
=
          random-bst (setop A B)
  ⟨proof⟩
  include monad-normalisation
  ⟨proof⟩
```

We now derive the intersection and difference from the generic operation:

```
fun mrbst-inter where
```

```
  mrbst-inter ⟨⟩ - = return-pmf ⟨⟩
| mrbst-inter ⟨l1, x, r1⟩ t2 =
    (case split-bst' x t2 of (sep, l2, r2) ⇒
      do {
        l ← mrbst-inter l1 l2;
        r ← mrbst-inter r1 r2;
        if sep then return-pmf ⟨l, x, r⟩ else mrbst-join l r
      })

lemma mrbst-inter-Leaf-left [simp]:
  mrbst-inter ⟨⟩ = (λ-. return-pmf ⟨⟩)
  ⟨proof⟩

lemma mrbst-inter-Leaf-right [simp]:
  mrbst-inter (t1 :: 'a :: linorder tree) ⟨⟩ = return-pmf ⟨⟩
  ⟨proof⟩

lemma mrbst-inter-reduce:
  mrbst-inter ⟨l1, x, r1⟩ =
    (λt2. case split-bst' x t2 of (sep, l2, r2) ⇒
      do {
        l ← mrbst-inter l1 l2;
        r ← mrbst-inter r1 r2;
        if sep then return-pmf ⟨l, x, r⟩ else mrbst-join l r
       })
  ⟨proof⟩

lemma mrbst-inter-altdef: mrbst-inter = mrbst-inter-diff True
⟨proof⟩

corollary
  fixes t1 t2 :: 'a :: linorder tree
  assumes t' ∈ set-pmf (mrbst-inter t1 t2) bst t1 bst t2
  shows   bst-mrbst-inter: bst t'
    and   set-mrbst-inter: set-tree t' = set-tree t1 ∩ set-tree t2
  ⟨proof⟩

corollary mrbst-inter-correct:
  fixes A B :: 'a :: linorder set
  assumes finite A finite B
  shows   do {t1 ← random-bst A; t2 ← random-bst B; mrbst-inter t1 t2} =
random-bst (A ∩ B)
  ⟨proof⟩


fun mrbst-diff where
  mrbst-diff ⟨⟩ - = return-pmf ⟨⟩
| mrbst-diff ⟨l1, x, r1⟩ t2 =
    (case split-bst' x t2 of (sep, l2, r2) ⇒
```

```
      do {
        l ← mrbst-diff l1 l2;
        r ← mrbst-diff r1 r2;
        if sep then mrbst-join l r else return-pmf ⟨l, x, r⟩
      })

lemma mrbst-diff-Leaf-left [simp]:
  mrbst-diff ⟨⟩ = (λ-. return-pmf ⟨⟩)
  ⟨proof⟩

lemma mrbst-diff-Leaf-right [simp]:
  mrbst-diff (t1 :: 'a :: linorder tree) ⟨⟩ = return-pmf t1
  ⟨proof⟩

lemma mrbst-diff-reduce:
  mrbst-diff ⟨l1, x, r1⟩ =
    (λt2. case split-bst' x t2 of (sep, l2, r2) ⇒
      do {
        l ← mrbst-diff l1 l2;
        r ← mrbst-diff r1 r2;
        if sep then mrbst-join l r else return-pmf ⟨l, x, r⟩
      })
  ⟨proof⟩

lemma If-not: (if ¬b then x else y) = (if b then y else x)
  ⟨proof⟩

lemma mrbst-diff-altdef: mrbst-diff = mrbst-inter-diff False
⟨proof⟩

corollary
  fixes t1 t2 :: 'a :: linorder tree
  assumes t' ∈ set-pmf (mrbst-diff t1 t2) bst t1 bst t2
  shows   bst-mrbst-diff: bst t'
    and   set-mrbst-diff: set-tree t' = set-tree t1 − set-tree t2
  ⟨proof⟩

corollary mrbst-diff-correct:
  fixes A B :: 'a :: linorder set
  assumes finite A finite B
   shows    do {t1 ← random-bst A; t2 ← random-bst B; mrbst-diff t1 t2} =
random-bst (A − B)
  ⟨proof⟩
```

1.6 Union

The algorithm for the union of two trees is by far the most complicated one. It involves a

```
fun mrbst-union where
  mrbst-union ⟨⟩ t2 = return-pmf t2
| mrbst-union t1 ⟨⟩ = return-pmf t1
| mrbst-union ⟨l1, x, r1⟩ ⟨l2, y, r2⟩ =
    do {
      let m = size ⟨l1, x, r1⟩; let n = size ⟨l2, y, r2⟩;
      b ← bernoulli-pmf (m / (m + n));
      if b then do {
        let (l2', r2') = split-bst x ⟨l2, y, r2⟩;
        l ← mrbst-union l1 l2';
        r ← mrbst-union r1 r2';
        return-pmf ⟨l, x, r⟩
      } else do {
        let (sep, l1', r1') = split-bst' y ⟨l1, x, r1⟩;
        l ← mrbst-union l1' l2;
        r ← mrbst-union r1' r2;
        if sep then
          mrbst-push-down l y r
        else
          return-pmf ⟨l, y, r⟩
      }
    }

lemma mrbst-union-Leaf-left [simp]: mrbst-union ⟨⟩ = return-pmf
  ⟨proof⟩

lemma mrbst-union-Leaf-right [simp]: mrbst-union t1 ⟨⟩ = return-pmf t1
  ⟨proof⟩

lemma
  fixes t1 t2 :: 'a :: linorder tree and b :: bool
  assumes t' ∈ set-pmf (mrbst-union t1 t2) bst t1 bst t2
  shows   bst-mrbst-union: bst t'
    and   set-mrbst-union: set-tree t' = set-tree t1 ∪ set-tree t2
⟨proof⟩

theorem mrbst-union-correct:
  assumes finite A finite B
  shows   do {t1 ← random-bst A; t2 ← random-bst B; mrbst-union t1 t2} =
          random-bst (A ∪ B)
⟨proof⟩ including monad-normalisation ⟨proof⟩

    include monad-normalisation
    ⟨proof⟩
```

1.7 Insertion and Deletion

The insertion and deletion operations are simple special cases of the union and difference operations where one of the trees is a singleton tree.

```
fun mrbst-insert where
  mrbst-insert x ⟨⟩ = return-pmf ⟨⟨⟩, x, ⟨⟩⟩
| mrbst-insert x ⟨l, y, r⟩ =
    do {
      b ← bernoulli-pmf (1 / real (size l + size r + 2));
      if b then do {
        let (l', r') = split-bst x ⟨l, y, r⟩;
        return-pmf ⟨l', x, r'⟩
      } else if x < y then do {
        map-pmf (λl'. ⟨l', y, r⟩) (mrbst-insert x l)
      } else if x > y then do {
        map-pmf (λr'. ⟨l, y, r'⟩) (mrbst-insert x r)
      } else do {
        mrbst-push-down l y r
      }
    }

lemma mrbst-insert-altdef: mrbst-insert x t = mrbst-union ⟨⟨⟩, x, ⟨⟩⟩ t
  ⟨proof⟩

corollary
  fixes t :: 'a :: linorder tree
  assumes t' ∈ set-pmf (mrbst-insert x t) bst t
  shows   bst-mrbst-insert: bst t'
    and   set-mrbst-insert: set-tree t' = insert x (set-tree t)
  ⟨proof⟩

corollary mrbst-insert-correct:
  assumes finite A
  shows   random-bst A ⤜ mrbst-insert x = random-bst (insert x A)
  ⟨proof⟩


fun mrbst-delete :: 'a :: ord ⇒ 'a tree ⇒ 'a tree pmf where
  mrbst-delete x ⟨⟩ = return-pmf ⟨⟩
| mrbst-delete x ⟨l, y, r⟩ = (
    if x < y then
      map-pmf (λl'. ⟨l', y, r⟩) (mrbst-delete x l)
    else if x > y then
      map-pmf (λr'. ⟨l, y, r'⟩) (mrbst-delete x r)
    else
      mrbst-join l r)

lemma mrbst-delete-altdef: mrbst-delete x t = mrbst-diff t ⟨⟨⟩, x, ⟨⟩⟩
  ⟨proof⟩

corollary
  fixes t :: 'a :: linorder tree
  assumes t' ∈ set-pmf (mrbst-delete x t) bst t
```

shows *bst-mrbst-delete*: *bst* t'
and *set-mrbst-delete*: *set-tree* $t' =$ *set-tree* $t - \{x\}$
⟨*proof*⟩

corollary *mrbst-delete-correct*:
finite $A \Longrightarrow$ *do* $\{t \leftarrow$ *random-bst* A; *mrbst-delete* x $t\} =$ *random-bst* $(A - \{x\})$
⟨*proof*⟩

end

References

[1] C. Martínez and S. Roura. Randomized binary search trees. *Journal of the ACM*, 45, 1997.