

Randomised Binary Search Trees

Manuel Eberl

March 17, 2025

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	2
1.1	Auxiliary facts	2
1.2	Partitioning a BST	7
1.3	Joining	11
1.4	Pushdown	16
1.5	Intersection and Difference	21
1.6	Union	29
1.7	Insertion and Deletion	39

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)
by simp
```

```
context
begin
```

```
interpretation pmf-as-function .
```

```
lemma True-in-set-bernoulli-pmf-iff [simp]:
True ∈ set-pmf (bernoulli-pmf p) ↔ p > 0
by transfer auto
```

```
lemma False-in-set-bernoulli-pmf-iff [simp]:
False ∈ set-pmf (bernoulli-pmf p) ↔ p < 1
by transfer auto
```

```
end
```

```
lemma in-set-pmf-of-setD: x ∈ set-pmf (pmf-of-set A) ==> finite A ==> A ≠ {}
==> x ∈ A
by (subst (asm) set-pmf-of-set) auto
```

```
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⟩}
by (subst random-bst.simps) auto
```

```
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
using assms by (simp add: pmf-bind)
```

```
lemma vimage-bool-pair:
f - ` A = (⋃ x ∈ {True, False}. ⋃ y ∈ {True, False}. iff f (x, y) ∈ A then {(x, y)})
else {})
(is ?lhs = ?rhs) unfolding set-eq-iff
proof
```

```

fix x :: bool × bool
obtain a b where [simp]: x = (a, b) by (cases x)
show x ∈ ?lhs ↔ x ∈ ?rhs
  by (cases a; cases b) auto
qed

lemma Leaf-in-set-random-bst-iff [simp]:
  Leaf ∈ set-pmf (random-bst A) ↔ A = {} ∨ ¬finite A
  by (subst random-bst.simps) auto

lemma bst-insert [intro]: bst t ==> bst (Tree-Set.insert x t)
  by (simp add: bst-iff-sorted-wrt-less inorder-insert sorted-ins-list)

lemma bst-bst-of-list [intro]: bst (bst-of-list xs)
proof -
  have bst (fold Tree-Set.insert xs t) if bst t for t
    using that
  proof (induction xs arbitrary: t)
    case (Cons y xs)
    show ?case by (auto intro!: Cons bst-insert)
  qed auto
  thus ?thesis by (simp add: bst-of-list-altdef)
qed

lemma bst-random-bst:
  assumes t ∈ set-pmf (random-bst A)
  shows bst t
proof (cases finite A)
  case True
  have random-bst A = map-pmf bst-of-list (pmf-of-set (permutations-of-set A))
    by (rule random-bst-altdef) fact+
  also have set-pmf ... = bst-of-list ` permutations-of-set A
    using True by auto
  finally show ?thesis using assms by auto
next
  case False
  hence random-bst A = return-pmf ⟨⟩
    by (simp add: random-bst.simps)
  with assms show ?thesis by simp
qed

lemma set-random-bst:
  assumes t ∈ set-pmf (random-bst A) finite A
  shows set-tree t = A
proof -
  have random-bst A = map-pmf bst-of-list (pmf-of-set (permutations-of-set A))
    by (rule random-bst-altdef) fact+
  also have set-pmf ... = bst-of-list ` permutations-of-set A
    using assms by auto

```

```

finally show ?thesis using assms
  by (auto simp: permutations-of-setD)
qed

lemma isin-bst:
  assumes bst t
  shows isin t x  $\longleftrightarrow$  x  $\in$  set-tree t
  using assms
  by (subst isin-set) (auto simp: bst-iff-sorted-wrt-less)

lemma isin-random-bst:
  assumes finite A t  $\in$  set-pmf (random-bst A)
  shows isin t x  $\longleftrightarrow$  x  $\in$  A
proof -
  from assms have bst t by (auto dest: bst-random-bst)
  with assms show ?thesis by (simp add: isin-bst set-random-bst)
qed

lemma card-3way-split:
  assumes x  $\in$  (A :: 'a :: linorder set) finite A
  shows card A = card {y  $\in$  A. y < x} + card {y  $\in$  A. y > x} + 1
proof -
  from assms have A = insert x ({y  $\in$  A. y < x}  $\cup$  {y  $\in$  A. y > x})
    by auto
  also have card ... = card {y  $\in$  A. y < x} + card {y  $\in$  A. y > x} + 1
    using assms by (subst card-insert-disjoint) (auto intro: card-Un-disjoint)
  finally show ?thesis .
qed

```

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 choosing an element from it uniformly at random.

```

lemma pmf-of-set-union-split:
  assumes finite A finite B A  $\cap$  B = {} A  $\cup$  B  $\neq$  {}
  assumes p = card A / (card A + card B)
  shows do {b  $\leftarrow$  bernoulli-pmf p; if b then pmf-of-set A else pmf-of-set B} =
    pmf-of-set (A  $\cup$  B)
    (is ?lhs = ?rhs)
proof (rule pmf-eqI)
  fix x :: 'a
  from assms have p: p  $\in$  {0..1}
    by (auto simp: divide-simps assms(5) split: if-splits)

  have pmf ?lhs x = pmf (pmf-of-set A) x * p + pmf (pmf-of-set B) x * (1 - p)
    unfolding pmf-bind using p by (subst integral-bernoulli-pmf) auto
  also consider x  $\in$  A B  $\neq$  {} | x  $\in$  B A  $\neq$  {} | x  $\in$  A B = {} | x  $\in$  B A = {} |
    x  $\notin$  A x  $\notin$  B
    using assms by auto
  hence pmf (pmf-of-set A) x * p + pmf (pmf-of-set B) x * (1 - p) = pmf ?rhs x

```

```

proof cases
  assume  $x \notin A$   $x \notin B$ 
  thus ?thesis using assms by (cases  $A = \{\}$ ; cases  $B = \{\}$ ) auto
next
  assume  $x \in A$  and [simp]:  $B \neq \{\}$ 
  have pmf (pmf-of-set  $A$ )  $x * p + \text{pmf}(\text{pmf-of-set } B) x * (1 - p) = p$  / real
  ( $\text{card } A$ )
  using ⟨ $x \in A$ ⟩ assms(1–4) by (subst (1 2) pmf-of-set) (auto simp: indicator-def)
  also have ... = pmf ?rhs  $x$ 
  using assms ⟨ $x \in A$ ⟩ by (subst pmf-of-set) (auto simp: card-Un-disjoint)
  finally show ?thesis .
next
  assume  $x \in B$  and [simp]:  $A \neq \{\}$ 
  from assms have *:  $\text{card}(A \cup B) > 0$  by (subst card-gt-0-iff) auto
  have pmf (pmf-of-set  $A$ )  $x * p + \text{pmf}(\text{pmf-of-set } B) x * (1 - p) = (1 - p)$ 
  / real ( $\text{card } B$ )
  using ⟨ $x \in B$ ⟩ assms(1–4) by (subst (1 2) pmf-of-set) (auto simp: indicator-def)
  also have ... = pmf ?rhs  $x$ 
  using assms ⟨ $x \in B$ ⟩ *
  by (subst pmf-of-set) (auto simp: card-Un-disjoint assms(5) divide-simps)
  finally show ?thesis .
qed (insert assms(1–4), auto simp: assms(5))
finally show pmf ?lhs  $x = \text{pmf}(\text{pmf-of-set } B) x$  .
qed

```

```

lemma pmf-of-set-split-inter-diff:
assumes finite  $A$  finite  $B$   $A \neq \{\}$   $B \neq \{\}$ 
assumes  $p = \text{card}(A \cap B) / \text{card } B$ 
shows do { $b \leftarrow \text{bernoulli-pmf } p$ ; if  $b$  then pmf-of-set  $(A \cap B)$  else pmf-of-set
( $B - A$ )} =
  pmf-of-set  $B$  (is ?lhs = ?rhs)
proof –
  have eq:  $B = (A \cap B) \cup (B - A)$  by auto
  have card-eq:  $\text{card } B = \text{card}(A \cap B) + \text{card}(B - A)$ 
  using assms by (subst eq, subst card-Un-disjoint) auto
  have ?lhs = pmf-of-set  $((A \cap B) \cup (B - A))$ 
  using assms by (intro pmf-of-set-union-split) (auto simp: card-eq)
  with eq show ?thesis by simp
qed

```

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  $f g h :: 'a \Rightarrow 'b$  pmf
assumes finite  $A$   $A \neq \{\}$   $A1 \cap A2 = \{\}$   $A1 \cap A3 = \{\}$   $A2 \cap A3 = \{\}$   $A1 \cup$ 

```

```

 $A_2 \cup A_3 = A$ 
shows do { $x \leftarrow \text{pmf-of-set } A$ ; if  $x \in A_1$  then  $f x$  else if  $x \in A_2$  then  $g x$  else  $h x$ } =
    do { $i \leftarrow \text{pmf-of-set }\{\dots < \text{card } A\}$ ;
        if  $i < \text{card } A_1$  then  $\text{pmf-of-set } A_1 \geqslant f$ 
        else if  $i < \text{card } A_1 + \text{card } A_2$  then  $\text{pmf-of-set } A_2 \geqslant g$ 
        else  $\text{pmf-of-set } A_3 \geqslant h$ } (is ?lhs = ?rhs)
proof (intro pmf-eqI)
  fix  $x :: 'b$ 
  define  $m n l$  where  $m = \text{card } A_1$  and  $n = \text{card } A_2$  and  $l = \text{card } A_3$ 
  have [simp]:  $\text{finite } A_1 \text{ finite } A_2 \text{ finite } A_3$ 
    by (rule finite-subset[of - A]; use assms in force)+
  from assms have card-pos:  $\text{card } A > 0$  by auto
  have A-eq:  $A = A_1 \cup A_2 \cup A_3$  using assms by simp
  have card-A-eq:  $\text{card } A = \text{card } A_1 + \text{card } A_2 + \text{card } A_3$ 
    using assms unfolding A-eq by (subst card-Un-disjoint, simp, simp, force)+
  auto
  have card-A-eq':  $\{\dots < \text{card } A\} = \{\dots < m\} \cup \{m\dots < m+n\} \cup \{m+n\dots < \text{card } A\}$ 
    by (auto simp: m-def n-def card-A-eq)
  let ?M =  $\lambda i$ . if  $i < m$  then  $\text{pmf-of-set } A_1 \geqslant f$  else if  $i < m+n$  then
     $\text{pmf-of-set } A_2 \geqslant g$  else  $\text{pmf-of-set } A_3 \geqslant h$ 

  have card-times-pmf-of-set-bind:
     $\text{card } X * \text{pmf} (\text{pmf-of-set } X \geqslant f) x = (\sum_{y \in X} \text{pmf} (f y) x)$ 
    if finite X for X :: 'a set and f :: 'a  $\Rightarrow$  'b pmf
    using that by (cases X = {}) (auto simp: pmf-bind-pmf-of-set)

  have pmf ?rhs x =  $(\sum_{i < \text{card } A} \text{pmf} (?M i) x) / \text{card } A$ 
    is - = ?S / - using assms card-pos unfolding m-def n-def
    by (subst pmf-bind-pmf-of-set) auto
  also have ?S =  $(\text{real } m * \text{pmf} (\text{pmf-of-set } A_1 \geqslant f) x +$ 
     $\text{real } n * \text{pmf} (\text{pmf-of-set } A_2 \geqslant g) x +$ 
     $\text{real } l * \text{pmf} (\text{pmf-of-set } A_3 \geqslant h) x)$  unfolding card-A-eq'
    by (subst sum.union-disjoint, simp, simp, force)+ (auto simp: card-A-eq m-def
  n-def l-def)
  also have ... =  $(\sum_{y \in A_1} \text{pmf} (f y) x) + (\sum_{y \in A_2} \text{pmf} (g y) x) + (\sum_{y \in A_3} \text{pmf} (h y) x)$ 
    unfolding m-def n-def l-def by (subst (1 2 3) card-times-pmf-of-set-bind) auto
  also have ... =  $(\sum_{y \in A_1 \cup A_2 \cup A_3} \text{pmf} (\text{if } y \in A_1 \text{ then } f y \text{ else if } y \in A_2 \text{ then } g y \text{ else } h y) x)$ 
    using assms(1-5)
    by (subst sum.union-disjoint, simp, simp, force)+
      (intro arg-cong2[of - - - (+)] sum.cong, auto)
  also have ... / card A = pmf ?lhs x
    using assms by (simp add: pmf-bind-pmf-of-set)
  finally show pmf ?lhs x = pmf ?rhs x
    unfolding m-def n-def l-def card-A-eq ..
qed

```

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  $\Rightarrow$  'a tree  $\Rightarrow$  'a tree  $\times$  'a tree where
  split-bst -  $\langle\rangle$  =  $\langle\langle\rangle, \langle\rangle\rangle$ 
  | split-bst x  $\langle l, y, r\rangle$  =
    (if  $y < x$  then
      case split-bst x r of (t1, t2)  $\Rightarrow$   $\langle\langle l, y, t1\rangle, t2\rangle$ 
      else if  $y > x$  then
        case split-bst x l of (t1, t2)  $\Rightarrow$   $\langle t1, \langle t2, y, r\rangle\rangle$ 
      else
         $\langle l, r\rangle$ )
    
```



```

fun split-bst' :: 'a :: linorder  $\Rightarrow$  'a tree  $\Rightarrow$  bool  $\times$  'a tree  $\times$  'a tree where
  split-bst' -  $\langle\rangle$  = (False,  $\langle\rangle, \langle\rangle\rangle$ )
  | split-bst' x  $\langle l, y, r\rangle$  =
    (if  $y < x$  then
      case split-bst' x r of (b, t1, t2)  $\Rightarrow$  (b,  $\langle l, y, t1\rangle, t2\rangle$ )
      else if  $y > x$  then
        case split-bst' x l of (b, t1, t2)  $\Rightarrow$  (b,  $t1, \langle t2, y, r\rangle$ )
      else
        (True,  $\langle l, r\rangle$ ))
    
```



```

lemma split-bst'-altdef: split-bst' x t = (isin t x, split-bst x t)
  by (induction x t rule: split-bst.induct) (auto simp: case-prod-unfold)

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
  by (simp-all add: split-bst'-altdef)

lemma size-fst-split-bst [termination-simp]: size (fst (split-bst x t))  $\leq$  size t
  by (induction t) (auto simp: case-prod-unfold)

lemma size-snd-split-bst [termination-simp]: size (snd (split-bst x t))  $\leq$  size t
  by (induction t) (auto simp: case-prod-unfold)

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

lemma set-split-bst1: bst t  $\Longrightarrow$  set-tree (fst (split-bst x t)) = {y  $\in$  set-tree t. y < x}
  by (induction t) (auto split: prod.splits)

lemma set-split-bst2: bst t  $\Longrightarrow$  set-tree (snd (split-bst x t)) = {y  $\in$  set-tree t. y > x}
  by (induction t) (auto split: prod.splits)

```

```
lemma bst-split-bst1 [intro]: bst t  $\implies$  bst (fst (split-bst x t))
by (induction t) (auto simp: case Prod-unfold set-split-bst1)
```

```
lemma bst-split-bst2 [intro]: bst t  $\implies$  bst (snd (split-bst x t))
by (induction t) (auto simp: case Prod-unfold set-split-bst2)
```

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  $\in$  A. y < x}) (random-bst {y  $\in$  A. y > x})
using assms
proof (induction A rule: random-bst.induct)
case (1 A)
define A1 A2 where A1 = {y  $\in$  A. y < x} and A2 = {y  $\in$  A. y > x}
have [simp]:  $\neg x \in A_2$  if x  $\in A_1$  for x using that by (auto simp: A1-def A2-def)
from ⟨finite A⟩ have [simp]: finite A1 finite A2 by (auto simp: A1-def A2-def)
include monad-normalisation

show ?case
proof (cases A = {})
case True
thus ?thesis by (auto simp: pair-return-pmf1)
next
case False

have map-pmf (split-bst x) (random-bst A) =
do {y  $\leftarrow$  pmf-of-set A;
if y < x then
do {
l  $\leftarrow$  random-bst {z  $\in$  A. z < y};
(t1, t2)  $\leftarrow$  map-pmf (split-bst x) (random-bst {z  $\in$  A. z > y});
return-pmf ((l, y, t1), t2)
}
else if y > x then
do {
(t1, t2)  $\leftarrow$  map-pmf (split-bst x) (random-bst {z  $\in$  A. z < y});
r  $\leftarrow$  random-bst {z  $\in$  A. z > y};
return-pmf (t1, ((t2, y, r)))
}
else
do {
l  $\leftarrow$  random-bst {z  $\in$  A. z < y};
r  $\leftarrow$  random-bst {z  $\in$  A. z > y};
return-pmf (l, r)
}
}
using 1.prems False
```

```

by (subst random-bst.simps)
  (simp add: map-bind-pmf bind-map-pmf return-pmf-if case-prod-unfold cong:
if-cong)
also have ... = do {y ← pmf-of-set A;
  if y < x then
    do {
      l ← random-bst {z∈A. z < y};
      (t1, t2) ← pair-pmf (random-bst {z∈{z∈A. z > y}. z < x})
        (random-bst {z∈{z∈A. z > y}. z > x});
      return-pmf ((l, y, t1), t2)
    }
  else if y > x then
    do {
      (t1, t2) ← pair-pmf (random-bst {z∈{z∈A. z < y}. z < x})
        (random-bst {z∈{z∈A. z < y}. z > x});
      r ← random-bst {z∈A. z > y};
      return-pmf (t1, (⟨t2, y, r⟩))
    }
  else
    do {
      l ← random-bst {z∈A. z < y};
      r ← random-bst {z∈A. z > y};
      return-pmf (l, r)
    }
}
using ⟨finite A⟩ and ⟨A ≠ {}⟩ thm 1.IH
by (intro bind-pmf-cong if-cong refl 1.IH) auto
also have ... = do {y ← pmf-of-set A;
  if y < x then
    do {
      l ← random-bst {z∈A. z < y};
      t1 ← random-bst {z∈{z∈A. z > y}. z < x};
      t2 ← random-bst {z∈{z∈A. z > y}. z > x};
      return-pmf ((l, y, t1), t2)
    }
  else if y > x then
    do {
      t1 ← random-bst {z∈{z∈A. z < y}. z < x};
      t2 ← random-bst {z∈{z∈A. z < y}. z > x};
      r ← random-bst {z∈A. z > y};
      return-pmf (t1, (⟨t2, y, r⟩))
    }
  else
    do {
      l ← random-bst {z∈A. z < y};
      r ← random-bst {z∈A. z > y};
      return-pmf (l, r)
    }
}

```

```

by (simp add: pair-pmf-def cong: if-cong)
also have ... = do {y ← pmf-of-set A;
  if y ∈ A1 then
    do {
      l ← random-bst {z ∈ A1. z < y};
      t1 ← random-bst {z ∈ A1. z > y};
      t2 ← random-bst A2;
      return-pmf ((l, y, t1), t2)
    }
  else if y ∈ A2 then
    do {
      t1 ← random-bst A1;
      t2 ← random-bst {z ∈ A2. z < y};
      r ← random-bst {z ∈ A2. z > y};
      return-pmf (t1, (t2, y, r))
    }
  else
    pair-pmf (random-bst A1) (random-bst A2)
}
using ‹finite A› ‹A ≠ {}›
by (intro bind-pmf-cong refl if-cong arg-cong[of -- random-bst])
  (auto simp: A1-def A2-def pair-pmf-def)
also have ... = do {i ← pmf-of-set {.. < card A};
  if i < card A1 then
    do {
      y ← pmf-of-set A1;
      l ← random-bst {z ∈ A1. z < y};
      t1 ← random-bst {z ∈ A1. z > y};
      t2 ← random-bst A2;
      return-pmf ((l, y, t1), t2)
    }
  else if i < card A1 + card A2 then
    do {
      y ← pmf-of-set A2;
      t1 ← random-bst A1;
      t2 ← random-bst {z ∈ A2. z < y};
      r ← random-bst {z ∈ A2. z > y};
      return-pmf (t1, (t2, y, r))
    }
  else do {
    y ← pmf-of-set (if x ∈ A then {x} else {});
    pair-pmf (random-bst A1) (random-bst A2)
  }
}
using ‹finite A› ‹A ≠ {}›
by (intro pmf-of-set-3way-split) (auto simp: A1-def A2-def not-less-iff-gr-or-eq)
also have ... = do {i ← pmf-of-set {.. < card A};
  if i < card A1 then
    pair-pmf (random-bst A1) (random-bst A2)
  else if i < card A1 + card A2 then

```

```

pair-pmf (random-bst A1) (random-bst A2)
else
pair-pmf (random-bst A1) (random-bst A2)
}
using ‹finite A› ‹A ≠ {}›
proof (intro bind-pmf-cong refl if-cong, goal-cases)
case (1 i)
hence A1 ≠ {} by auto
thus ?case using ‹finite A› by (simp add: pair-pmf-def random-bst-reduce)
next
case (2 i)
hence A2 ≠ {} by auto
thus ?case using ‹finite A› by (simp add: pair-pmf-def random-bst-reduce)
qed auto
also have ... = pair-pmf (random-bst A1) (random-bst A2)
by (simp cong: if-cong)
finally show ?thesis by (simp add: A1-def A2-def)
qed
qed

```

1.3 Joining

The “join” operation computes the union of two BSTs l and r where all the values in l are strictly 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
  by (simp add: fun-eq-iff)

lemma mrbst-join-Leaf-right [simp]: mrbst-join t ⟨⟩ = return-pmf t
  by (simp add: fun-eq-iff)

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 (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)))
}

```

```

        (case t2 of ⟨l, x, r⟩ ⇒ map-pmf (λl'. ⟨l', x, r⟩) (mrbst-join t1 l))
    }
by (subst mrbst-join.simps) auto

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 –
have bst t' ∧ set-tree t' = set-tree t1 ∪ set-tree t2
using assms
proof (induction size t1 + size t2 arbitrary: t1 t2 t' rule: less-induct)
case (less t1 t2 t')
show ?case
proof (cases t1 = ⟨⟩ ∨ t2 = ⟨⟩)
case False
hence t' ∈ set-pmf (case t1 of ⟨l, x, r⟩ ⇒ map-pmf (Node l x) (mrbst-join r t2)) ∨
t' ∈ set-pmf (case t2 of ⟨l, x, r⟩ ⇒ map-pmf (λl'. ⟨l', x, r⟩) (mrbst-join t1 l))
using less.preds by (subst (asm) mrbst-join-reduce) (auto split: if-splits)
thus ?thesis
proof
assume t' ∈ set-pmf (case t1 of ⟨l, x, r⟩ ⇒ map-pmf (Node l x) (mrbst-join r t2))
then obtain l x r r'
where *: t1 = ⟨l, x, r⟩ r' ∈ set-pmf (mrbst-join r t2) t' = ⟨l, x, r'⟩
using False by (auto split: tree.splits)
from * and less.preds have bst r' ∧ set-tree r' = set-tree r ∪ set-tree t2
by (intro less) auto
with * and less.preds show ?thesis by auto
next
assume t' ∈ set-pmf (case t2 of ⟨l, x, r⟩ ⇒ map-pmf (λl'. ⟨l', x, r⟩) (mrbst-join t1 l))
then obtain l x r l'
where *: t2 = ⟨l, x, r⟩ l' ∈ set-pmf (mrbst-join t1 l) t' = ⟨l', x, r⟩
using False by (auto split: tree.splits)
from * and less.preds have bst l' ∧ set-tree l' = set-tree t1 ∪ set-tree l
by (intro less) auto
with * and less.preds show ?thesis by auto
qed
qed (insert less.preds, auto)
qed
thus bst t' set-tree t' = set-tree t1 ∪ set-tree t2 by auto
qed

```

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  $\wedge \forall x y. x \in A \implies y \in B \implies x < y$ 
  shows do {t1  $\leftarrow$  random-bst A; t2  $\leftarrow$  random-bst B; mrbst-join t1 t2} =
    random-bst (A  $\cup$  B)
  proof -
    from assms have finite (A  $\cup$  B) by simp
    from this and assms show ?thesis
    proof (induction A  $\cup$  B arbitrary: A B rule: finite-psubset-induct)
      case (psubset A B)
      define m n where m = card A and n = card B
      define p where p = m / (m + n)

      include monad-normalisation
      show ?case
      proof (cases A = {}  $\vee$  B = {})
        case True
        thus ?thesis by auto
      next
        case False
        have AB: A  $\neq$  {} B  $\neq$  {} finite A finite B
          using False psubset.psms by auto
        have p-pos: A  $\neq$  {} if p > 0 using ‹finite A› that
          using AB by (auto simp: p-def m-def n-def)
        have p-lt1: B  $\neq$  {} if p < 1
          using AB by (auto simp: p-def m-def n-def)

        have do {t1  $\leftarrow$  random-bst A; t2  $\leftarrow$  random-bst B; mrbst-join t1 t2} =
          do {t1  $\leftarrow$  random-bst A;
              t2  $\leftarrow$  random-bst B;
              b  $\leftarrow$  bernoulli-pmf (size t1 / (size t1 + size t2));
              if b then
                case t1 of ⟨l, x, r⟩  $\Rightarrow$  map-pmf (λr'. ⟨l, x, r'⟩) (mrbst-join r t2)
                else
                  case t2 of ⟨l, x, r⟩  $\Rightarrow$  map-pmf (λl'. ⟨l', x, r⟩) (mrbst-join t1 l)
              }
            using AB
            by (intro bind-pmf-cong refl, subst mrbst-join-reduce) auto
        also have ... = do {t1  $\leftarrow$  random-bst A;
          t2  $\leftarrow$  random-bst B;
          b  $\leftarrow$  bernoulli-pmf p;
          if b then
            case t1 of ⟨l, x, r⟩  $\Rightarrow$  map-pmf (λr'. ⟨l, x, r'⟩) (mrbst-join
            r t2)
            else
              case t2 of ⟨l, x, r⟩  $\Rightarrow$  map-pmf (λl'. ⟨l', x, r⟩) (mrbst-join
            t1 l)
          }
        
```

```

using AB by (intro bind-pmf-cong refl arg-cong[of - - bernoulli-pmf])
  (auto simp: p-def m-def n-def size-random-bst)
also have ... = do {
  b ← bernoulli-pmf p;
  if b then do {
    t1 ← random-bst A;
    t2 ← random-bst B;
    case t1 of ⟨l, x, r⟩ ⇒ map-pmf (λr'. ⟨l, x, r'⟩) (mrbst-join r
t2)
  } else do {
    t1 ← random-bst A;
    t2 ← random-bst B;
    case t2 of ⟨l, x, r⟩ ⇒ map-pmf (λl'. ⟨l', x, r⟩) (mrbst-join t1 l)
  }
}
by simp
also have ... = do {
  b ← bernoulli-pmf p;
  if b then do {
    x ← pmf-of-set A;
    l ← random-bst {y ∈ A ∪ B. y < x};
    r ← random-bst {y ∈ A ∪ B. y > x};
    return-pmf ⟨l, x, r⟩
  } else do {
    x ← pmf-of-set B;
    l ← random-bst {y ∈ A ∪ B. y < x};
    r ← random-bst {y ∈ A ∪ B. y > x};
    return-pmf ⟨l, x, r⟩
  }
}
proof (intro bind-pmf-cong refl if-cong, goal-cases)
  case (1 b)
  hence [simp]: A ≠ {} using p-pos by auto
  have do {t1 ← random-bst A; t2 ← random-bst B;
    case t1 of ⟨l, x, r⟩ ⇒ map-pmf (λr'. ⟨l, x, r'⟩) (mrbst-join r t2)} =
    do {
      x ← pmf-of-set A;
      l ← random-bst {y ∈ A. y < x};
      r ← do {r ← random-bst {y ∈ A. y > x}; t2 ← random-bst B; mrbst-join
r t2};
      return-pmf ⟨l, x, r⟩
    }
  using AB by (subst random-bst-reduce) (auto simp: map-pmf-def)
also have ... = do {
  x ← pmf-of-set A;
  l ← random-bst {y ∈ A. y < x};
  r ← random-bst ({y ∈ A. y > x} ∪ B);
  return-pmf ⟨l, x, r⟩
}

```

```

using AB psubset.prem
by (intro bind-pmf-cong refl psubset arg-cong[of -- random-bst]) auto
also have ... = do {
  x ← pmf-of-set A;
  l ← random-bst {y ∈ A ∪ B. y < x};
  r ← random-bst {y ∈ A ∪ B. y > x};
  return-pmf ⟨l, x, r⟩
}
using AB psubset.prem
by (intro bind-pmf-cong refl arg-cong[of -- random-bst]; force)
finally show ?case .
next
case (2 b)
hence [simp]: B ≠ {} using p-lt1 by auto
have do {t1 ← random-bst A; t2 ← random-bst B;
  case t2 of ⟨l, x, r⟩ ⇒ map-pmf (λl'. ⟨l', x, r⟩) (mrbst-join t1 l)} =
  do {
    x ← pmf-of-set B;
    l ← do {t1 ← random-bst A; l ← random-bst {y ∈ B. y < x}; mrbst-join
    t1 l};
    r ← random-bst {y ∈ B. y > x};
    return-pmf ⟨l, x, r⟩
}
using AB by (subst random-bst-reduce) (auto simp: map-pmf-def)
also have ... = do {
  x ← pmf-of-set B;
  l ← random-bst (A ∪ {y ∈ B. y < x});
  r ← random-bst {y ∈ B. y > x};
  return-pmf ⟨l, x, r⟩
}
using AB psubset.prem
by (intro bind-pmf-cong refl psubset arg-cong[of -- random-bst]) auto
also have ... = do {
  x ← pmf-of-set B;
  l ← random-bst {y ∈ A ∪ B. y < x};
  r ← random-bst {y ∈ A ∪ B. y > x};
  return-pmf ⟨l, x, r⟩
}
using AB psubset.prem
by (intro bind-pmf-cong refl arg-cong[of -- random-bst]; force)
finally show ?case .
qed
also have ... = do {
  b ← bernoulli-pmf p;
  x ← (if b then pmf-of-set A else pmf-of-set B);
  l ← random-bst {y ∈ A ∪ B. y < x};
  r ← random-bst {y ∈ A ∪ B. y > x};
  return-pmf ⟨l, x, r⟩
}

```

```

by (intro bind-pmf-cong) simp-all
also have ... = do {
    x ← do {b ← bernoulli-pmf p; if b then pmf-of-set A else
pmf-of-set B};
    l ← random-bst {y∈A ∪ B. y < x};
    r ← random-bst {y∈A ∪ B. y > x};
    return-pmf ⟨l, x, r⟩
}
by simp
also have do {b ← bernoulli-pmf p; if b then pmf-of-set A else pmf-of-set B} =
pmf-of-set (A ∪ B)
using AB psubset.prem by (intro pmf-of-set-union-split) (auto simp: p-def
m-def n-def)
also have do {
    x ← pmf-of-set (A ∪ B);
    l ← random-bst {y∈A ∪ B. y < x};
    r ← random-bst {y∈A ∪ B. y > x};
    return-pmf ⟨l, x, r⟩
} = random-bst (A ∪ B)
using AB by (intro random-bst-reduce [symmetric]) auto
finally show ?thesis .
qed
qed
qed

```

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 (λr'. ⟨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 (λl'. ⟨l', y, rr⟩) (mrbst-push-down l x rl)
      else
        return-pmf ⟨l, x, r⟩
    }

```

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

```

lemma
  assumes  $t' \in \text{set-pmf}(\text{mrbst-push-down } t1 \ x \ t2)$   $\text{bst } t1 \ \text{bst } t2$ 
  assumes  $\bigwedge y. \ y \in \text{set-tree } t1 \implies y < x \ \bigwedge y. \ y \in \text{set-tree } t2 \implies y > x$ 
  shows  $\text{bst-mrbst-push-down}: \text{bst } t'$ 
    and  $\text{set-mrbst-push-down}: \text{set-tree } t' = \{x\} \cup \text{set-tree } t1 \cup \text{set-tree } t2$ 
proof -
  have  $\text{bst } t' \wedge \text{set-tree } t' = \{x\} \cup \text{set-tree } t1 \cup \text{set-tree } t2$ 
  using assms
  proof (induction size t1 + size t2 arbitrary: t1 t2 t' rule: less-induct)
    case (less t1 t2 t')
      have  $t1 \neq \langle \rangle \wedge t' \in \text{set-pmf}(\text{case t1 of } \langle l, y, r \rangle \Rightarrow$ 
         $\text{map-pmf}(\text{Node } l \ y) (\text{mrbst-push-down } r \ x \ t2)) \vee$ 
         $t2 \neq \langle \rangle \wedge t' \in \text{set-pmf}(\text{case t2 of } \langle l, y, r \rangle \Rightarrow$ 
         $\text{map-pmf}(\lambda l'. \langle l', y, r \rangle) (\text{mrbst-push-down } t1 \ x \ l)) \vee$ 
         $t' = \langle t1, x, t2 \rangle$ 
      using less.prems by (subst (asm) mrbst-push-down.simps) (auto split: if-splits)
      thus ?case
      proof (elim disjE, goal-cases)
        case 1
        then obtain  $l \ y \ r \ r'$ 
        where  $*: t1 = \langle l, y, r \rangle \ r' \in \text{set-pmf}(\text{mrbst-push-down } r \ x \ t2) \ t' = \langle l, y, r' \rangle$ 
          by (auto split: tree.splits)
        from  $*$  and less.prems have  $\text{bst } r' \wedge \text{set-tree } r' = \{x\} \cup \text{set-tree } r \cup \text{set-tree}$ 
 $t2$ 
        by (intro less) auto
        with  $*$  and less.prems show ?case by force
next
  case 2
  then obtain  $l \ y \ r \ l'$ 
  where  $*: t2 = \langle l, y, r \rangle \ l' \in \text{set-pmf}(\text{mrbst-push-down } t1 \ x \ l) \ t' = \langle l', y, r \rangle$ 
    by (auto split: tree.splits)
  from  $*$  and less.prems have  $\text{bst } l' \wedge \text{set-tree } l' = \{x\} \cup \text{set-tree } t1 \cup \text{set-tree}$ 
 $l$ 
  by (intro less) auto
  with  $*$  and less.prems show ?case by force
  qed (insert less.prems, auto)
qed
  thus  $\text{bst } t' \ \text{set-tree } t' = \{x\} \cup \text{set-tree } t1 \cup \text{set-tree } t2$  by auto
qed

theorem mrbst-push-down-correct:
  fixes  $A \ B :: 'a :: \text{linorder set}$ 
  assumes finite A finite B  $\bigwedge y. \ y \in A \implies y < x \ \bigwedge y. \ y \in B \implies x < y$ 
  shows  $\text{do } \{l \leftarrow \text{random-bst } A; r \leftarrow \text{random-bst } B; \text{mrbst-push-down } l \ x \ r\} =$ 
     $\text{random-bst } (\{x\} \cup A \cup B)$ 
proof -
  from assms have finite (A ∪ B) by simp
  from this and assms show ?thesis
  proof (induction A ∪ B arbitrary: A B rule: finite-psubset-induct)

```

```

case (psubset A B)
define m n where m = card A and n = card B
have A-ne: A ≠ {} if m > 0
    using that by (auto simp: m-def)
have B-ne: B ≠ {} if n > 0
    using that by (auto simp: n-def)

include monad-normalisation
have do {l ← random-bst A; r ← random-bst B; mrbst-push-down l x r} =
    do {l ← random-bst A;
        r ← random-bst B;
        k ← pmf-of-set {0..m + n};
        if k < m then
                case l of ⟨ll, y, lr⟩ ⇒ map-pmf (λr'. ⟨ll, y, r'⟩) (mrbst-push-down lr
x r)
            else if k < m + n then
                    case r of ⟨rl, y, rr⟩ ⇒ map-pmf (λl'. ⟨l', y, rr⟩) (mrbst-push-down l
x rl)
            else
                    return-pmf ⟨l, x, r⟩
    }
using psubset.prems
by (subst mrbst-push-down.simps, intro bind-pmf-cong refl)
    (auto simp: size-random-bst m-def n-def)
also have ... = do {k ← pmf-of-set {0..m + n};
        if k < m then do {
                l ← random-bst A;
                r ← random-bst B;
                case l of ⟨ll, y, lr⟩ ⇒ map-pmf (λr'. ⟨ll, y, r'⟩) (mrbst-push-down
lr x r)
            } else if k < m + n then do {
                    l ← random-bst A;
                    r ← random-bst B;
                    case r of ⟨rl, y, rr⟩ ⇒ map-pmf (λl'. ⟨l', y, rr⟩) (mrbst-push-down
l x rl)
            } else do {
                    l ← random-bst A;
                    r ← random-bst B;
                    return-pmf ⟨l, x, r⟩
            }
    }
by (simp cong: if-cong)
also have ... = do {k ← pmf-of-set {0..m + n};
        if k < m then do {
                y ← pmf-of-set A;
                ll ← random-bst {z ∈ A. z < y};
                r' ← do {lr ← random-bst {z ∈ A. z > y};
                        r ← random-bst B;
                        mrbst-push-down lr x r};
    }

```

```

        return-pmf ⟨ll, y, r⟩
    } else if  $k < m + n$  then do {
         $y \leftarrow \text{pmf-of-set } B;$ 
         $l' \leftarrow \text{do } \{l \leftarrow \text{random-bst } A;$ 
             $rl \leftarrow \text{random-bst } \{z \in B. z < y\};$ 
             $\text{mrbst-push-down } l \ x \ rl\};$ 
         $rr \leftarrow \text{random-bst } \{z \in B. z > y\};$ 
        return-pmf ⟨l', y, rr⟩
    } else do {
         $l \leftarrow \text{random-bst } A;$ 
         $r \leftarrow \text{random-bst } B;$ 
        return-pmf ⟨l, x, r⟩
    }
}

proof (intro bind-pmf-cong refl if-cong, goal-cases)
  case (1 k)
  hence  $A \neq \{\}$  by (auto simp: m-def)
  with ⟨finite A⟩ show ?case by (simp add: random-bst-reduce map-pmf-def)
next
  case (2 k)
  hence  $B \neq \{\}$  by (auto simp: m-def n-def)
  with ⟨finite B⟩ show ?case by (simp add: random-bst-reduce map-pmf-def)
qed
also have ... = do { $k \leftarrow \text{pmf-of-set } \{0..m + n\};$ 
   $\text{if } k < m \text{ then do } \{$ 
     $y \leftarrow \text{pmf-of-set } A;$ 
     $ll \leftarrow \text{random-bst } \{z \in A. z < y\};$ 
     $r' \leftarrow \text{random-bst } (\{x\} \cup \{z \in A. z > y\} \cup B);$ 
    return-pmf ⟨ll, y, r'⟩
  } else if  $k < m + n$  then do {
     $y \leftarrow \text{pmf-of-set } B;$ 
     $l' \leftarrow \text{random-bst } (\{x\} \cup A \cup \{z \in B. z < y\});$ 
     $rr \leftarrow \text{random-bst } \{z \in B. z > y\};$ 
    return-pmf ⟨l', y, rr⟩
  } else do {
     $l \leftarrow \text{random-bst } A;$ 
     $r \leftarrow \text{random-bst } B;$ 
    return-pmf ⟨l, x, r⟩
  }
}

using psubset.prem A-ne B-ne
proof (intro bind-pmf-cong refl if-cong psubset)
  fix k y assume  $k < m$   $y \in \text{set-pmf } (\text{pmf-of-set } A)$ 
  thus  $\{z \in A. z > y\} \cup B \subset A \cup B$ 
    using psubset.prem A-ne by (fastforce dest!: in-set-pmf-of-setD)
next
  fix k y assume  $\neg k < m$   $k < m + n$   $y \in \text{set-pmf } (\text{pmf-of-set } B)$ 
  thus  $A \cup \{z \in B. z < y\} \subset A \cup B$ 
    using psubset.prem B-ne by (fastforce dest!: in-set-pmf-of-setD)

```

```

qed auto
also have ... = do {k  $\leftarrow$  pmf-of-set {0..m + n};
  if k < m then do {
    y  $\leftarrow$  pmf-of-set A;
    ll  $\leftarrow$  random-bst {z $\in$ {x}  $\cup$  A  $\cup$  B. z < y};
    r'  $\leftarrow$  random-bst {z $\in$ {x}  $\cup$  A  $\cup$  B. z > y};
    return-pmf ⟨ll, y, r'⟩
  } else if k < m + n then do {
    y  $\leftarrow$  pmf-of-set B;
    l'  $\leftarrow$  random-bst {z $\in$ {x}  $\cup$  A  $\cup$  B. z < y};
    rr  $\leftarrow$  random-bst {z $\in$ {x}  $\cup$  A  $\cup$  B. z > y};
    return-pmf ⟨l', y, rr⟩
  } else do {
    l  $\leftarrow$  random-bst {z $\in$ {x}  $\cup$  A  $\cup$  B. z < x};
    r  $\leftarrow$  random-bst {z $\in$ {x}  $\cup$  A  $\cup$  B. z > x};
    return-pmf ⟨l, x, r⟩
  }
}

using psubset.prems A-ne B-ne
by (intro bind-pmf-cong if-cong refl arg-cong[of -- random-bst];
  force dest: psubset.prems(3,4))
also have ... = do {k  $\leftarrow$  pmf-of-set {0..m + n};
  if k < m then do {
    y  $\leftarrow$  pmf-of-set A;
    ll  $\leftarrow$  random-bst {z $\in$ {x}  $\cup$  A  $\cup$  B. z < y};
    r'  $\leftarrow$  random-bst {z $\in$ {x}  $\cup$  A  $\cup$  B. z > y};
    return-pmf ⟨ll, y, r'⟩
  } else if k < m + n then do {
    y  $\leftarrow$  pmf-of-set B;
    l'  $\leftarrow$  random-bst {z $\in$ {x}  $\cup$  A  $\cup$  B. z < y};
    rr  $\leftarrow$  random-bst {z $\in$ {x}  $\cup$  A  $\cup$  B. z > y};
    return-pmf ⟨l', y, rr⟩
  } else do {
    y  $\leftarrow$  pmf-of-set {x};
    l  $\leftarrow$  random-bst {z $\in$ {x}  $\cup$  A  $\cup$  B. z < y};
    r  $\leftarrow$  random-bst {z $\in$ {x}  $\cup$  A  $\cup$  B. z > y};
    return-pmf ⟨l, x, r⟩
  }
}
} (is - = ?X {0..m+n})
by (simp add: pmf-of-set-singleton cong: if-cong)
also have {0..m + n} = {.. $<$ card (A  $\cup$  B  $\cup$  {x})} using psubset.prems
by (subst card-Un-disjoint, simp, simp, force) +
  (auto simp: m-def n-def)
also have ?X ... = do {y  $\leftarrow$  pmf-of-set ({x}  $\cup$  A  $\cup$  B);
  l  $\leftarrow$  random-bst {z $\in$ {x}  $\cup$  A  $\cup$  B. z < y};
  r  $\leftarrow$  random-bst {z $\in$ {x}  $\cup$  A  $\cup$  B. z > y};
  return-pmf ⟨l, y, r⟩}
unfolding m-def n-def using psubset.prems
by (subst pmf-of-set-3way-split [symmetric])

```

```

(auto dest!: psubset.prefs(3,4) cong: if-cong intro: bind-pmf-cong)
also have ... = random-bst ({x} ∪ A ∪ B)
  using psubset.prefs by (simp add: random-bst-reduce)
  finally show ?case .
qed
qed

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 -
have ?lhs = random-bst ({x} ∪ {y ∈ A. y < x} ∪ {y ∈ A. y > x})
  using assms by (intro mrbst-push-down-correct) auto
also have {x} ∪ {y ∈ A. y < x} ∪ {y ∈ A. y > x} = A
  using assms by auto
finally show ?thesis .
qed

```

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
})
by (rule ext) simp

```

lemma mrbst-inter-diff-Leaf-left [simp]:

```

mrbst-inter-diff b ⟨⟩ = (λ-. return-pmf ⟨⟩)

```

by (simp add: fun-eq-iff)

```

lemma mrbst-inter-diff-Leaf-right [simp]:
  mrbst-inter-diff b (t1 :: 'a :: linorder tree) () = return-pmf (if b then () else t1)
  by (induction t1) (auto simp: bind-return-pmf)

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 -
  write setop (infixl ⊲⊳ 80)
  have bst t' ∧ set-tree t' = set-tree t1 ⊲ set-tree t2
  using assms(2-)
  proof (induction t1 arbitrary: t2 t')
    case (Node l1 x r1 t2)
    note bst = ⟨bst (l1, x, r1), bst t2⟩
    define l2 r2 where "l2 = fst (split-bst x t2) and r2 = snd (split-bst x t2)"
    obtain l r
      where lr: l ∈ set-pmf (mrbst-inter-diff b l1 l2) r ∈ set-pmf (mrbst-inter-diff
        b r1 r2)
        and t': t' ∈ (if x ∈ set-tree t2 ⟷ b then {⟨l, x, r⟩} else set-pmf (mrbst-join
        l r))
      using Node.preds by (force simp: case-prod-unfold l2-def r2-def isin-bst split:
        if-splits)
    from lr have lr': bst l ∧ set-tree l = set-tree l1 ⊲ set-tree l2
      bst r ∧ set-tree r = set-tree r1 ⊲ set-tree r2
    using Node.preds by (intro Node.IH; force simp: l2-def r2-def) +
    have set-tree t' = set-tree l ∪ set-tree r ∪ (if x ∈ set-tree t2 ⟷ b then {x}
      else {})
    proof (cases x ∈ set-tree t2 ⟷ b)
      case False
      have x < y if x ∈ set-tree l y ∈ set-tree r for x y
        using that lr' bst by (force simp: setop-def split: if-splits)
      hence set-t': set-tree t' = set-tree l ∪ set-tree r
        using t' set-mrbst-join[of t' l r] False lr' by auto
      with False show ?thesis by simp
    qed (use t' in auto)
    also have ... = set-tree (l1, x, r1) ⊲ set-tree t2
    using lr' bst by (auto simp: setop-def l2-def r2-def set-split-bst1 set-split-bst2)
    finally have set-tree t' = set-tree (l1, x, r1) ⊲ set-tree t2 .
    moreover from lr' t' bst have bst t'
      by (force split: if-splits simp: setop-def intro!: bst-mrbst-join[of t' l r])
    ultimately show ?case by auto
  qed (auto simp: setop-def)
  thus bst t' and set-tree t' = set-tree t1 ⊲ set-tree t2 by auto

```

qed

```

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)
  using assms(2−)
  proof (induction A arbitrary: B rule: finite-psubset-induct)
    case (psubset A B)
    write setop (infixl ⟨⟩ 80)
    include monad-normalisation
    show ?case
    proof (cases A = {})
      case True
      thus ?thesis by (auto simp: setop-def)
    next
      case False
      define R1 R2 where R1 = (λx. random-bst {y ∈ A. y < x}) R2 = (λx.
      random-bst {y ∈ A. y > x})
      have A-eq: A = (A ∩ B) ∪ (A − B) by auto
      have card-A-eq: card A = card (A ∩ B) + card (A − B)
        using ⟨finite A⟩ ⟨finite B⟩ by (subst A-eq, subst card-Un-disjoint) auto
      have eq: pmf-of-set A =
        do {b ← bernoulli-pmf (card (A ∩ B) / card A);
            if b then pmf-of-set (A ∩ B) else pmf-of-set (A − B)}
      using psubset.prem False ⟨finite A⟩ A-eq card-A-eq
        by (subst A-eq, intro pmf-of-set-union-split [symmetric]) auto
      have card A > 0
        using ⟨finite A⟩ ⟨A ≠ {}⟩ by (subst card-gt-0-iff) auto
      have not-subset: ¬A ⊆ B if card (A ∩ B) < card A
      proof
        assume A ⊆ B
        hence A ∩ B = A by auto
        with that show False by simp
      qed

      have do {t1 ← random-bst A; t2 ← random-bst B; mrbst-inter-diff b t1 t2} =
        do {
          x ← pmf-of-set A;
          l1 ← random-bst {y ∈ A. y < x};
          r1 ← random-bst {y ∈ A. y > x};
          t2 ← random-bst B;
          let (l2, r2) = split-bst x t2;
          l ← mrbst-inter-diff b l1 l2;
          r ← mrbst-inter-diff b r1 r2;

```

```

        if isin t2 x = b then return-pmf ⟨l, x, r⟩ else mrbst-join l r
    }
using ⟨finite A⟩ ⟨A ≠ {}⟩
by (subst random-bst-reduce)
    (auto simp: mrbst-inter-diff-reduce map-pmf-def split-bst'-altdef)
also have ... = do {
    x ← pmf-of-set A;
    l1 ← random-bst {y∈A. y < x};
    r1 ← random-bst {y∈A. y > x};
    t2 ← random-bst B;
    let (l2, r2) = split-bst x t2;
    l ← mrbst-inter-diff b l1 l2;
    r ← mrbst-inter-diff b r1 r2;
    if x ∈ B = b then return-pmf ⟨l, x, r⟩ else mrbst-join l r
}
unfolding Let-def case-prod-unfold using ⟨finite B⟩
by (intro bind-pmf-cong refl) (auto simp: isin-random-bst)
also have ... = do {
    x ← pmf-of-set A;
    l1 ← random-bst {y∈A. y < x};
    r1 ← random-bst {y∈A. y > x};
    (l2, r2) ← map-pmf (split-bst x) (random-bst B);
    l ← mrbst-inter-diff b l1 l2;
    r ← mrbst-inter-diff b r1 r2;
    if x ∈ B = b then return-pmf ⟨l, x, r⟩ else mrbst-join l r
}
by (simp add: Let-def map-pmf-def)
also have ... = do {
    x ← pmf-of-set A;
    l1 ← random-bst {y∈A. y < x};
    r1 ← random-bst {y∈A. y > x};
    (l2, r2) ← pair-pmf (random-bst {y∈B. y < x}) (random-bst
{y∈B. y > x});
    l ← mrbst-inter-diff b l1 l2;
    r ← mrbst-inter-diff b r1 r2;
    if x ∈ B = b then return-pmf ⟨l, x, r⟩ else mrbst-join l r
}
by (intro bind-pmf-cong refl split-random-bst ⟨finite B⟩)
also have ... = do {
    x ← pmf-of-set A;
    l1 ← R1 x;
    r1 ← R2 x;
    l2 ← random-bst {y∈B. y < x};
    r2 ← random-bst {y∈B. y > x};
    l ← mrbst-inter-diff b l1 l2;
    r ← mrbst-inter-diff b r1 r2;
    if x ∈ B = b then return-pmf ⟨l, x, r⟩ else mrbst-join l r
}
unfolding pair-pmf-def bind-assoc-pmf R1-R2-def by simp

```

```

also have ... = do {
  x ← pmf-of-set A;
  l ← do {l1 ← R1 x; l2 ← random-bst {y∈B. y < x}; mrbst-inter-diff
  b l1 l2};
  r ← do {r1 ← R2 x; r2 ← random-bst {y∈B. y > x};
  mrbst-inter-diff b r1 r2};
  if x ∈ B = b then return-pmf ⟨l, x, r⟩ else mrbst-join l r
}
unfolding bind-assoc-pmf by (intro bind-pmf-cong[OF refl]) simp
also have ... = do {
  x ← pmf-of-set A;
  l ← random-bst ({y∈A. y < x} ▷ {y∈B. y < x});
  r ← random-bst ({y∈A. y > x} ▷ {y∈B. y > x});
  if x ∈ B = b then return-pmf ⟨l, x, r⟩ else mrbst-join l r
}
using ⟨finite A⟩ ⟨finite B⟩ ⟨A ≠ {}⟩ unfolding R1-R2-def
by (intro bind-pmf-cong refl psubset.IH) auto
also have ... = do {
  x ← pmf-of-set A;
  if x ∈ B = b then do {
    l ← random-bst ({y∈A. y < x} ▷ {y∈B. y < x});
    r ← random-bst ({y∈A. y > x} ▷ {y∈B. y > x});
    return-pmf ⟨l, x, r⟩
  } else do {
    l ← random-bst ({y∈A. y < x} ▷ {y∈B. y < x});
    r ← random-bst ({y∈A. y > x} ▷ {y∈B. y > x});
    mrbst-join l r
  }
}
by simp
also have ... = do {
  x ← pmf-of-set A;
  if x ∈ B = b then do {
    l ← random-bst ({y∈A. y < x} ▷ {y∈B. y < x});
    r ← random-bst ({y∈A. y > x} ▷ {y∈B. y > x});
    return-pmf ⟨l, x, r⟩
  } else do {
    random-bst ({y∈A. y < x} ▷ {y∈B. y < x} ∪ {y∈A. y > x} ▷
    {y∈B. y > x})
  }
}
using ⟨finite A⟩ ⟨finite B⟩
by (intro bind-pmf-cong refl mrbst-join-correct if-cong) (auto simp: setop-def)
also have ... = do {
  x ← pmf-of-set A;
  if x ∈ B = b then do {
    l ← random-bst ({y∈A ▷ B. y < x});
    r ← random-bst ({y∈A ▷ B. y > x});
    return-pmf ⟨l, x, r⟩
}

```

```

} else do {
  random-bst (A ◇ B)
}
} (is - = pmf-of-set A ≈? ?f)
using ⟨finite A⟩ ⟨A ≠ {}⟩
by (intro bind-pmf-cong refl if-cong arg-cong[of - - random-bst])
  (auto simp: order.strict-iff-order setop-def)
also have ... = do {
  b' ← bernoulli-pmf (card (A ∩ B) / card A);
  x ← (if b' then pmf-of-set (A ∩ B) else pmf-of-set (A - B));
  if b' = b then do {
    l ← random-bst ({y ∈ A ◇ B. y < x});
    r ← random-bst ({y ∈ A ◇ B. y > x});
    return-pmf ⟨l, x, r⟩
  } else do {
    random-bst (A ◇ B)
  }
}
unfolding bind-assoc-pmf eq using ⟨card A > 0⟩ ⟨finite A⟩ ⟨finite B⟩ not-subset
by (intro bind-pmf-cong refl if-cong)
  (auto intro: bind-pmf-cong split: if-splits simp: divide-simps card-gt-0-iff
dest!: in-set-pmf-of-setD)
also have ... = do {
  b' ← bernoulli-pmf (card (A ∩ B) / card A);
  if b' = b then do {
    x ← pmf-of-set (A ◇ B);
    l ← random-bst ({y ∈ A ◇ B. y < x});
    r ← random-bst ({y ∈ A ◇ B. y > x});
    return-pmf ⟨l, x, r⟩
  } else do {
    random-bst (A ◇ B)
  }
}
by (intro bind-pmf-cong) (auto simp: setop-def)
also have ... = do {
  b' ← bernoulli-pmf (card (A ∩ B) / card A);
  if b' = b then do {
    random-bst (A ◇ B)
  } else do {
    random-bst (A ◇ B)
  }
}
using ⟨finite A⟩ ⟨finite B⟩ ⟨A ≠ {}⟩ not-subset ⟨card A > 0⟩
by (intro bind-pmf-cong refl if-cong random-bst-reduce [symmetric])
  (auto simp: setop-def field-simps)
also have ... = random-bst (A ◇ B) by simp
finally show ?thesis .
qed
qed

```

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]:
 $\text{mrbst-inter } \langle \rangle = (\lambda _. \text{return-pmf } \langle \rangle)$
by (simp add: fun-eq-iff)

lemma mrbst-inter-Leaf-right [simp]:
 $\text{mrbst-inter } (t1 :: 'a :: \text{linorder tree}) \langle \rangle = \text{return-pmf } \langle \rangle$
by (induction t1) (auto simp: bind-return-pmf)

lemma mrbst-inter-reduce:
 $\text{mrbst-inter } \langle l1, x, r1 \rangle =$
 $(\lambda t2. \text{case split-bst' } x t2 \text{ of (sep, l2, r2)} \Rightarrow$
 $\text{do } \{$
 $l \leftarrow \text{mrbst-inter } l1 l2;$
 $r \leftarrow \text{mrbst-inter } r1 r2;$
 $\text{if sep then return-pmf } \langle l, x, r \rangle \text{ else mrbst-join } l r$
 $\})$
by (rule ext) simp

lemma mrbst-inter-altdef: $\text{mrbst-inter} = \text{mrbst-inter-diff} \text{ True}$
proof (intro ext)
 fix t1 t2 :: 'a tree
 show $\text{mrbst-inter } t1 t2 = \text{mrbst-inter-diff} \text{ True } t1 t2$
 by (induction t1 arbitrary: t2) auto
qed

corollary
 fixes t1 t2 :: 'a :: linorder tree
 assumes $t' \in \text{set-pmf} (\text{mrbst-inter } t1 t2) \text{ bst } t1 \text{ bst } t2$
 shows $\text{bst-mrbst-inter: bst } t'$
 and $\text{set-mrbst-inter: set-tree } t' = \text{set-tree } t1 \cap \text{set-tree } t2$
 using $\text{bst-mrbst-inter-diff}[of t' \text{ True } t1 t2] \text{ set-mrbst-inter-diff}[of t' \text{ True } t1 t2]$
 assms
by (simp-all add: mrbst-inter-altdef)

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)
  using assms unfolding mrbst-inter-altdef by (subst mrbst-inter-diff-correct)
  simp-all

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 ())
  by (simp add: fun-eq-iff)

lemma mrbst-diff-Leaf-right [simp]:
  mrbst-diff (t1 :: 'a :: linorder tree) () = return-pmf t1
  by (induction t1) (auto simp: bind-return-pmf)

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)
   })
  by (rule ext) simp

lemma If-not: (if ¬b then x else y) = (if b then y else x)
  by auto

lemma mrbst-diff-altdef: mrbst-diff = mrbst-inter-diff False
proof (intro ext)
  fix t1 t2 :: 'a tree
  show mrbst-diff t1 t2 = mrbst-inter-diff False t1 t2
    by (induction t1 arbitrary: t2) (auto simp: If-not)
qed

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
  using bst-mrbst-inter-diff[of t' False t1 t2] set-mrbst-inter-diff[of t' False t1 t2]

```

```

assms
by (simp-all add: mrbst-diff-altdef)

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)
using assms unfolding mrbst-diff-altdef by (subst mrbst-inter-diff-correct) simp-all

```

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
by (rule ext) simp

lemma mrbst-union-Leaf-right [simp]: mrbst-union t1 ⟨⟩ = return-pmf t1
by (cases t1) simp-all

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 -
  have bst t' ∧ set-tree t' = set-tree t1 ∪ set-tree t2
  using assms
  proof (induction size t1 + size t2 arbitrary: t1 t2 t' rule: less-induct)
    case (less t1 t2 t')
    show ?case
    proof (cases t1 = ⟨⟩ ∨ t2 = ⟨⟩)
      case False
      then obtain l1 x r1 l2 y r2 where t1: t1 = ⟨l1, x, r1⟩ and t2: t2 = ⟨l2, y,
      r2⟩
        by (cases t1; cases t2) auto
        from less.preds consider l r where
          l ∈ set-pmf (mrbst-union l1 (fst (split-bst x t2)))
          r ∈ set-pmf (mrbst-union r1 (snd (split-bst x t2)))
          t' = ⟨l, x, r⟩
        | l r where
          l ∈ set-pmf (mrbst-union (fst (split-bst y t1)) l2)
          r ∈ set-pmf (mrbst-union (snd (split-bst y t1)) r2)
          t' ∈ (if isin ⟨l1, x, r1⟩ y then set-pmf (mrbst-push-down l y r) else {⟨l, y,
          r⟩})
        by (auto simp: case-prod-unfold t1 t2 Let-def
              simp del: split-bst.simps split-bst'.simps isin.simps split: if-splits)
      thus ?thesis
      proof cases
        case 1
        hence lr: bst l ∧ set-tree l = set-tree l1 ∪ set-tree (fst (split-bst x t2))
              bst r ∧ set-tree r = set-tree r1 ∪ set-tree (snd (split-bst x t2))
        using less.preds size-split-bst[of x t2]
        by (intro less; force simp: t1)+
        thus ?thesis
          using 1 less.preds by (auto simp: t1 set-split-bst1 set-split-bst2)
      next
        case 2
        hence lr: bst l ∧ set-tree l = set-tree (fst (split-bst y t1)) ∪ set-tree l2
              bst r ∧ set-tree r = set-tree (snd (split-bst y t1)) ∪ set-tree r2
        using less.preds size-split-bst[of y t1]
        by (intro less; force simp: t2)+
        show ?thesis
        proof (cases isin ⟨l1, x, r1⟩ y)
          case False
          thus ?thesis using 2 less.preds lr
            by (auto simp del: isin.simps simp: t2 set-split-bst1 set-split-bst2)
      next
        case True
        have bst': ∀ z ∈ set-tree l. z < y ∀ z ∈ set-tree r. z > y
        using lr less.preds by (auto simp: set-split-bst1 set-split-bst2 t2)
        from True and 2 have t': t' ∈ set-pmf (mrbst-push-down l y r)
        by (auto simp del: isin.simps)
        from t' have bst t'

```

```

    by (rule bst-mrbst-push-down) (use lr bst' in auto)
  moreover from t' have set-tree t' = {y} ∪ set-tree l ∪ set-tree r
    by (rule set-mrbst-push-down) (use lr bst' in auto)
  ultimately show ?thesis using less.prems lr
    by (auto simp del: isin.simps simp: t2 set-split-bst1 set-split-bst2)
qed
qed
qed (use less.prems in auto)
qed
thus bst t' and set-tree t' = set-tree t1 ∪ set-tree t2 by auto
qed

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 -
  from assms have finite (A ∪ B) by simp
  thus ?thesis
    proof (induction A ∪ B arbitrary: A B rule: finite-psubset-induct)
      case (psubset A B)
      show ?case
        proof (cases A = {} ∨ B = {})
          case True
          thus ?thesis including monad-normalisation by auto
        next
          case False
            with psubset.hyps have AB: finite A finite B A ≠ {} B ≠ {} by auto
            define m n l where m = card A and n = card B and l = card (A ∩ B)
            define p q where p = m / (m + n) and q = l / n
            define r where r = p / (1 - (1 - p) * q)
            from AB have mn: m > 0 n > 0 by (auto simp: m-def n-def)
            have pq: p ∈ {0..1} q ∈ {0..1}
              using AB by (auto simp: p-def q-def m-def n-def l-def divide-simps intro:
                card-mono)
            moreover have p ≠ 0
            using AB by (auto simp: p-def m-def n-def divide-simps add-nonneg-eq-0-iff)
            ultimately have p > 0 by auto

            have B - A = B - (A ∩ B) by auto
            also have card ... = n - l
              using AB unfolding n-def l-def by (intro card-Diff-subset) auto
            finally have [simp]: card (B - A) = n - l .
            from AB have l ≤ n unfolding l-def n-def by (intro card-mono) auto

            have p ≤ 1 - (1 - p) * q
              using mn <l ≤ n> by (auto simp: p-def q-def divide-simps)
            hence r-aux: (1 - p) * q ∈ {0..1 - p}
              using pq by auto

```

```

include monad-normalisation
define RA1 RA2 RB1 RB2
  where RA1 = ( $\lambda x. \text{random-bst} \{z \in A. z < x\}$ ) and RA2 = ( $\lambda x. \text{random-bst} \{z \in A. z > x\}$ )
    and RB1 = ( $\lambda x. \text{random-bst} \{z \in B. z < x\}$ ) and RB2 = ( $\lambda x. \text{random-bst} \{z \in B. z > x\}$ )

have do { $t_1 \leftarrow \text{random-bst } A; t_2 \leftarrow \text{random-bst } B; \text{mrbst-union } t_1 t_2\}$ } =
  do {
     $x \leftarrow \text{pmf-of-set } A;$ 
     $l_1 \leftarrow \text{random-bst} \{z \in A. z < x\};$ 
     $r_1 \leftarrow \text{random-bst} \{z \in A. z > x\};$ 
     $y \leftarrow \text{pmf-of-set } B;$ 
     $l_2 \leftarrow \text{random-bst} \{z \in B. z < y\};$ 
     $r_2 \leftarrow \text{random-bst} \{z \in B. z > y\};$ 
    let  $m = \text{size } \langle l_1, x, r_1 \rangle$ ;
    let  $n = \text{size } \langle l_2, y, r_2 \rangle$ ;
     $b \leftarrow \text{bernoulli-pmf} (m / (m + n));$ 
    if  $b$  then do {
       $l \leftarrow \text{mrbst-union } l_1 (\text{fst} (\text{split-bst } x \langle l_2, y, r_2 \rangle));$ 
       $r \leftarrow \text{mrbst-union } r_1 (\text{snd} (\text{split-bst } x \langle l_2, y, r_2 \rangle));$ 
      return-pmf  $\langle l, x, r \rangle$ 
    } else do {
       $l \leftarrow \text{mrbst-union} (\text{fst} (\text{split-bst } y \langle l_1, x, r_1 \rangle)) l_2;$ 
       $r \leftarrow \text{mrbst-union} (\text{snd} (\text{split-bst } y \langle l_1, x, r_1 \rangle)) r_2;$ 
      if  $\text{isin} \langle l_1, x, r_1 \rangle y$  then
        mrbst-push-down  $l y r$ 
      else
        return-pmf  $\langle l, y, r \rangle$ 
    }
  } using AB
  by (simp add: random-bst-reduce split-bst'-altdef Let-def case-prod-unfold
cong: if-cong)
also have ... = do {
   $x \leftarrow \text{pmf-of-set } A;$ 
   $l_1 \leftarrow \text{random-bst} \{z \in A. z < x\};$ 
   $r_1 \leftarrow \text{random-bst} \{z \in A. z > x\};$ 
   $y \leftarrow \text{pmf-of-set } B;$ 
   $l_2 \leftarrow \text{random-bst} \{z \in B. z < y\};$ 
   $r_2 \leftarrow \text{random-bst} \{z \in B. z > y\};$ 
   $b \leftarrow \text{bernoulli-pmf } p;$ 
  if  $b$  then do {
     $l \leftarrow \text{mrbst-union } l_1 (\text{fst} (\text{split-bst } x \langle l_2, y, r_2 \rangle));$ 
     $r \leftarrow \text{mrbst-union } r_1 (\text{snd} (\text{split-bst } x \langle l_2, y, r_2 \rangle));$ 
    return-pmf  $\langle l, x, r \rangle$ 
  } else do {
     $l \leftarrow \text{mrbst-union} (\text{fst} (\text{split-bst } y \langle l_1, x, r_1 \rangle)) l_2;$ 
     $r \leftarrow \text{mrbst-union} (\text{snd} (\text{split-bst } y \langle l_1, x, r_1 \rangle)) r_2;$ 
  }
}

```

```

if  $y \in A$  then
  mrbst-push-down  $l\ y\ r$ 
else
  return-pmf  $\langle l, y, r \rangle$ 
}
}

unfolding Let-def
proof (intro bind-pmf-cong refl if-cong)
  fix  $l1\ x\ r1\ y$ 
  assume  $l1 \in \text{set-pmf}(\text{random-bst}\{z \in A. z < x\})$   $r1 \in \text{set-pmf}(\text{random-bst}\{z \in A. z > x\})$ 
   $x \in \text{set-pmf}(\text{pmf-of-set } A)$ 
  thus  $\text{isin}\langle l1, x, r1 \rangle\ y \longleftrightarrow (y \in A)$ 
  using AB by (subst isin-bst) (auto simp: bst-random-bst set-random-bst)
qed (insert AB,
  auto simp: size-random-bst m-def n-def p-def isin-random-bst dest!
  card-3way-split)
also have ... = do {
   $b \leftarrow \text{bernoulli-pmf } p;$ 
  if  $b$  then do {
     $x \leftarrow \text{pmf-of-set } A;$ 
     $(l1, r1) \leftarrow \text{pair-pmf}(\text{random-bst}\{z \in A. z < x\}) (\text{random-bst}\{z \in A. z > x\});$ 
     $(l2, r2) \leftarrow \text{map-pmf}(\text{split-bst } x) (\text{random-bst } B);$ 
     $l \leftarrow \text{mrbst-union } l1\ l2;$ 
     $r \leftarrow \text{mrbst-union } r1\ r2;$ 
    return-pmf  $\langle l, x, r \rangle$ 
  } else do {
     $y \leftarrow \text{pmf-of-set } B;$ 
     $(l1, r1) \leftarrow \text{map-pmf}(\text{split-bst } y) (\text{random-bst } A);$ 
     $(l2, r2) \leftarrow \text{pair-pmf}(\text{random-bst}\{z \in B. z < y\}) (\text{random-bst}\{z \in B. z > y\});$ 
     $l \leftarrow \text{mrbst-union } l1\ l2;$ 
     $r \leftarrow \text{mrbst-union } r1\ r2;$ 
    if  $y \in A$  then
      mrbst-push-down  $l\ y\ r$ 
    else
      return-pmf  $\langle l, y, r \rangle$ 
  }
} using AB
by (simp add: random-bst-reduce map-pmf-def case-prod-unfold pair-pmf-def
cong: if-cong)
also have ... = do {
   $b \leftarrow \text{bernoulli-pmf } p;$ 
  if  $b$  then do {
     $x \leftarrow \text{pmf-of-set } A;$ 
     $(l1, r1) \leftarrow \text{pair-pmf}(RA1\ x)\ (RA2\ x);$ 
     $(l2, r2) \leftarrow \text{pair-pmf}(RB1\ x)\ (RB2\ x);$ 
     $l \leftarrow \text{mrbst-union } l1\ l2;$ 
  }
}

```

```

     $r \leftarrow mrbst\text{-union } r1\ r2;$ 
     $\text{return-pmf } \langle l, x, r \rangle$ 
} else do {
     $y \leftarrow pmf\text{-of-set } B;$ 
     $(l1, r1) \leftarrow pair\text{-pmf } (RA1\ y) (RA2\ y);$ 
     $(l2, r2) \leftarrow pair\text{-pmf } (RB1\ y) (RB2\ y);$ 
     $l \leftarrow mrbst\text{-union } l1\ l2;$ 
     $r \leftarrow mrbst\text{-union } r1\ r2;$ 
    if  $y \in A$  then
         $mrbst\text{-push-down } l\ y\ r$ 
    else
         $\text{return-pmf } \langle l, y, r \rangle$ 
}
}

unfolding case-prod-unfold RA1-def RA2-def RB1-def RB2-def
by (intro bind-pmf-cong refl if-cong split-random-bst AB)
also have ... = do {
     $b \leftarrow bernoulli\text{-pmf } p;$ 
    if  $b$  then do {
         $x \leftarrow pmf\text{-of-set } A;$ 
         $l \leftarrow do \{l1 \leftarrow RA1\ x; l2 \leftarrow RB1\ x; mrbst\text{-union } l1\ l2\};$ 
         $r \leftarrow do \{r1 \leftarrow RA2\ x; r2 \leftarrow RB2\ x; mrbst\text{-union } r1\ r2\};$ 
         $\text{return-pmf } \langle l, x, r \rangle$ 
    } else do {
         $y \leftarrow pmf\text{-of-set } B;$ 
         $l \leftarrow do \{l1 \leftarrow RA1\ y; l2 \leftarrow RB1\ y; mrbst\text{-union } l1\ l2\};$ 
         $r \leftarrow do \{r1 \leftarrow RA2\ y; r2 \leftarrow RB2\ y; mrbst\text{-union } r1\ r2\};$ 
        if  $y \in A$  then
             $mrbst\text{-push-down } l\ y\ r$ 
        else
             $\text{return-pmf } \langle l, y, r \rangle$ 
    }
}
by (simp add: pair-pmf-def cong: if-cong)
also have ... = do {
     $b \leftarrow bernoulli\text{-pmf } p;$ 
    if  $b$  then do {
         $x \leftarrow pmf\text{-of-set } A;$ 
         $l \leftarrow random\text{-bst } (\{z \in A. z < x\} \cup \{z \in B. z < x\});$ 
         $r \leftarrow random\text{-bst } (\{z \in A. z > x\} \cup \{z \in B. z > x\});$ 
         $\text{return-pmf } \langle l, x, r \rangle$ 
    } else do {
         $y \leftarrow pmf\text{-of-set } B;$ 
         $l \leftarrow random\text{-bst } (\{z \in A. z < y\} \cup \{z \in B. z < y\});$ 
         $r \leftarrow random\text{-bst } (\{z \in A. z > y\} \cup \{z \in B. z > y\});$ 
        if  $y \in A$  then
             $mrbst\text{-push-down } l\ y\ r$ 
        else
             $\text{return-pmf } \langle l, y, r \rangle$ 
    }
}

```

```

        }
    }

unfolding RA1-def RA2-def RB1-def RB2-def using AB
    by (intro bind-pmf-cong if-cong refl psubset) auto
also have ... = do {
    b ← bernoulli-pmf p;
    if b then do {
        x ← pmf-of-set A;
        l ← random-bst {z∈A ∪ B. z < x};
        r ← random-bst {z∈A ∪ B. z > x};
        return-pmf ⟨l, x, r⟩
    } else do {
        y ← pmf-of-set B;
        l ← random-bst {z∈A ∪ B. z < y};
        r ← random-bst {z∈A ∪ B. z > y};
        if y ∈ A then
            mrbst-push-down l y r
        else
            return-pmf ⟨l, y, r⟩
    }
}

by (intro bind-pmf-cong if-cong refl arg-cong[of - - random-bst]) auto
also have ... = do {
    b ← bernoulli-pmf p;
    if b then do {
        x ← pmf-of-set A;
        l ← random-bst {z∈A ∪ B. z < x};
        r ← random-bst {z∈A ∪ B. z > x};
        return-pmf ⟨l, x, r⟩
    } else do {
        b' ← bernoulli-pmf q;
        if b' then do {
            y ← pmf-of-set (A ∩ B);
            random-bst (A ∪ B)
        } else do {
            y ← pmf-of-set (B - A);
            l ← random-bst {z∈A ∪ B. z < y};
            r ← random-bst {z∈A ∪ B. z > y};
            return-pmf ⟨l, y, r⟩
        }
    }
}

proof (intro bind-pmf-cong refl if-cong, goal-cases)
case (1 b)
have q-pos: A ∩ B ≠ {} if q > 0 using that by (auto simp: q-def l-def)
have q-lt1: B - A ≠ {} if q < 1
proof
    assume B - A = {}
    hence A ∩ B = B by auto

```

thus *False* using that AB by (auto simp: q-def l-def n-def)
qed

```

have eq: pmf-of-set  $B = \text{do } \{b' \leftarrow \text{bernoulli-pmf } q;$   

                    if  $b'$  then pmf-of-set ( $A \cap B$ ) else pmf-of-set ( $B - A$ ) $\}$   

using  $AB$  by (intro pmf-of-set-split-inter-diff [symmetric])  

                    (auto simp: q-def l-def n-def)
have do { $y \leftarrow \text{pmf-of-set } B;$   

             $l \leftarrow \text{random-bst } \{z \in A \cup B. z < y\};$   

             $r \leftarrow \text{random-bst } \{z \in A \cup B. z > y\};$   

            if  $y \in A$  then  

              mrbst-push-down  $l \ y \ r$   

            else  

              return-pmf  $\langle l, y, r \rangle$   

          } =  

        do {  

           $b' \leftarrow \text{bernoulli-pmf } q;$   

           $y \leftarrow (\text{if } b' \text{ then pmf-of-set } (A \cap B) \text{ else pmf-of-set } (B - A));$   

           $l \leftarrow \text{random-bst } \{z \in A \cup B. z < y\};$   

           $r \leftarrow \text{random-bst } \{z \in A \cup B. z > y\};$   

          if  $b'$  then  

            mrbst-push-down  $l \ y \ r$   

          else  

            return-pmf  $\langle l, y, r \rangle$   

        } unfolding eq bind-assoc-pmf using AB q-pos q-lt1  

        by (intro bind-pmf-cong refl if-cong) (auto split: if-splits)
also have ... = do {  

     $b' \leftarrow \text{bernoulli-pmf } q;$   

    if  $b'$  then do {  

       $y \leftarrow \text{pmf-of-set } (A \cap B);$   

      do { $l \leftarrow \text{random-bst } \{z \in A \cup B. z < y\};$   

             $r \leftarrow \text{random-bst } \{z \in A \cup B. z > y\};$   

            mrbst-push-down  $l \ y \ r$ }  

    } else do {  

       $y \leftarrow \text{pmf-of-set } (B - A);$   

       $l \leftarrow \text{random-bst } \{z \in A \cup B. z < y\};$   

       $r \leftarrow \text{random-bst } \{z \in A \cup B. z > y\};$   

      return-pmf  $\langle l, y, r \rangle$   

    }  

} by (simp cong: if-cong)
also have ... = do {  

     $b' \leftarrow \text{bernoulli-pmf } q;$   

    if  $b'$  then do {  

       $y \leftarrow \text{pmf-of-set } (A \cap B);$   

      random-bst ( $A \cup B$ )  

    } else do {  

       $y \leftarrow \text{pmf-of-set } (B - A);$   

       $l \leftarrow \text{random-bst } \{z \in A \cup B. z < y\};$   

       $r \leftarrow \text{random-bst } \{z \in A \cup B. z > y\};$ 

```

```

        return-pmf ⟨l, y, r⟩
    }
}
using AB q-pos by (intro bind-pmf-cong if-cong refl mrbst-push-down-correct')
auto
  finally show ?case .
qed
also have ... = do {
  b ← bernoulli-pmf p;
  b' ← bernoulli-pmf q;
  if b then do {
    x ← pmf-of-set A;
    l ← random-bst {z ∈ A ∪ B. z < x};
    r ← random-bst {z ∈ A ∪ B. z > x};
    return-pmf ⟨l, x, r⟩
  } else if b' then do {
    random-bst (A ∪ B)
  } else do {
    y ← pmf-of-set (B - A);
    l ← random-bst {z ∈ A ∪ B. z < y};
    r ← random-bst {z ∈ A ∪ B. z > y};
    return-pmf ⟨l, y, r⟩
  }
}
by (simp cong: if-cong)
also have ... = do {
  (b, b') ← pair-pmf (bernoulli-pmf p) (bernoulli-pmf q);
  if b ∨ ¬b' then do {
    x ← (if b then pmf-of-set A else pmf-of-set (B - A));
    l ← random-bst {z ∈ A ∪ B. z < x};
    r ← random-bst {z ∈ A ∪ B. z > x};
    return-pmf ⟨l, x, r⟩
  } else do {
    random-bst (A ∪ B)
  }
} unfolding pair-pmf-def bind-assoc-pmf
by (intro bind-pmf-cong) auto
also have ... = do {
  (b, b') ← map-pmf (λ(b, b'). (b ∨ ¬b', b))
  (pair-pmf (bernoulli-pmf p) (bernoulli-pmf q));
  if b then do {
    x ← (if b' then pmf-of-set A else pmf-of-set (B - A));
    l ← random-bst {z ∈ A ∪ B. z < x};
    r ← random-bst {z ∈ A ∪ B. z > x};
    return-pmf ⟨l, x, r⟩
  } else do {
    random-bst (A ∪ B)
  }
} (is - = bind-pmf - ?f)

```

```

by (simp add: bind-map-pmf case-prod-unfold cong: if-cong)
also have map-pmf ( $\lambda(b, b'). (b \vee \neg b', b)$ )
  (pair-pmf (bernoulli-pmf p) (bernoulli-pmf q)) =
    do {
       $b \leftarrow \text{bernoulli-pmf } (1 - (1 - p) * q);$ 
       $b' \leftarrow (\text{if } b \text{ then bernoulli-pmf } r \text{ else return-pmf } \text{False});$ 
      return-pmf ( $b, b'$ )
    } (is ?lhs = ?rhs)
proof (intro pmf-eqI)
  fix  $bb' :: \text{bool} \times \text{bool}$ 
  obtain  $b b'$  where [simp]:  $bb' = (b, b')$  by (cases bb')
  thus pmf ?lhs  $bb' = \text{pmf } ?rhs bb'$ 
    using pq r-aux  $\langle p > 0 \rangle$ 
    by (cases b; cases b')
    (auto simp: pmf-map pmf-bind-bernoulli measure-measure-pmf-finite
      vimage-bool-pair pmf-pair r-def field-simps)
qed
also have ...  $\gg= ?f = \text{do } \{$ 
   $b \leftarrow \text{bernoulli-pmf } (1 - (1 - p) * q);$ 
  if  $b$  then do {
     $x \leftarrow \text{do } \{ b' \leftarrow \text{bernoulli-pmf } r;$ 
    if  $b'$  then pmf-of-set A else pmf-of-set ( $B - A$ );
     $l \leftarrow \text{random-bst } \{z \in A \cup B. z < x\};$ 
     $r \leftarrow \text{random-bst } \{z \in A \cup B. z > x\};$ 
    return-pmf  $\langle l, x, r \rangle$ 
  } else do {
    random-bst ( $A \cup B$ )
  }
}
by (simp cong: if-cong)
also have ... = do {
   $b \leftarrow \text{bernoulli-pmf } (1 - (1 - p) * q);$ 
  if  $b$  then do {
     $x \leftarrow \text{pmf-of-set } (A \cup (B - A));$ 
     $l \leftarrow \text{random-bst } \{z \in A \cup B. z < x\};$ 
     $r \leftarrow \text{random-bst } \{z \in A \cup B. z > x\};$ 
    return-pmf  $\langle l, x, r \rangle$ 
  } else do {
    random-bst ( $A \cup B$ )
  }
} (is - = ?f ( $A \cup (B - A)$ ))
using AB pq  $\langle l \leq n \rangle mn$ 
by (intro bind-pmf-cong if-cong refl pmf-of-set-union-split)
  (auto simp: m-def [symmetric] n-def [symmetric] r-def p-def q-def divide-simps)
also have  $A \cup (B - A) = A \cup B$  by auto
also have ?f ... = random-bst ( $A \cup B$ )
  using AB by (simp add: random-bst-reduce cong: if-cong)
finally show ?thesis .

```

```

qed
qed
qed
```

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
by (induction x t rule: mrbst-insert.induct)
 (simp-all add: Let-def map-pmf-def bind-return-pmf case-prod-unfold cong:
 if-cong)

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)
using bst-mrbst-union[of t' ⟨⟨⟩, x, ⟨⟩⟩ t] set-mrbst-union[of t' ⟨⟨⟩, x, ⟨⟩⟩ t] assms
by (simp-all add: mrbst-insert-altdef)

corollary mrbst-insert-correct:
assumes finite A
shows random-bst A ≈ mrbst-insert x = random-bst (insert x A)
using mrbst-union-correct[of {x} A] assms
by (simp add: mrbst-insert-altdef[abs-def] bind-return-pmf)

```

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 ( $\lambda l'. \langle l', y, r \rangle$ ) (mrbst-delete x l)
else if  $x > y$  then
  map-pmf ( $\lambda r'. \langle l, y, r' \rangle$ ) (mrbst-delete x r)
else
  mrbst-join l r)

lemma mrbst-delete-altdef: mrbst-delete x t = mrbst-diff t ⟨⟨⟩, x, ⟩⟩
by (induction t) (auto simp: bind-return-pmf map-pmf-def)

corollary
fixes t :: 'a :: linorder tree
assumes t'  $\in$  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}
using bst-mrbst-diff[of t' t ⟨⟨⟩, x, ⟩⟩] set-mrbst-diff[of t' t ⟨⟨⟩, x, ⟩⟩] assms
by (simp-all add: mrbst-delete-altdef)

corollary mrbst-delete-correct:
  finite A  $\implies$  do {t  $\leftarrow$  random-bst A; mrbst-delete x t} = random-bst (A - {x})
using mrbst-diff-correct[of A {x}] by (simp add: mrbst-delete-altdef bind-return-pmf)

end

```

References

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