Treaps

Max Haslbeck, Manuel Eberl, Tobias Nipkow

March 17, 2025

Abstract

A Treap [2] is a binary tree whose nodes contain pairs consisting of some payload and an associated priority. It must have the search-tree property w. r. t. the payloads and the heap property w. r. t. the priorities. Treaps are an interesting data structure that is related to binary search trees (BSTs) in the following way: if one forgets all the priorities of a treap, the resulting BST is exactly the same as if one had inserted the elements into an empty BST in order of ascending priority. This means that a treap behaves like a BST where we can pretend the elements were inserted in a different order from the one in which they were actually inserted.

In particular, by choosing these priorities at random upon insertion of an element, we can pretend that we inserted the elements in *random order*, so that the shape of the resulting tree is that of a random BST no matter in what order we insert the elements. This is the main result of this formalisation. [1]

Contents

1 Auxiliary material

theory *Probability-Misc*
imports *HOL−Probability.Probability*
begin

lemma *measure-eqI-countable-AE′*:
assumes [*simp*]: *sets M* = *Pow B sets N* = *Pow B* **and** *subset*: Ω ⊆ *B*
assumes *ae*: *AE x in M*. *x* ∈ Ω *AE x in N*. *x* ∈ Ω **and** [*simp*]: *countable* Ω
assumes *eq*: ⋀*x*. *x* ∈ Ω ⟹ *emeasure M* {*x*} = *emeasure N* {*x*}
shows *M* = *N*
proof (*rule measure-eqI*)
fix *A* **assume** *A*: *A* ∈ *sets M*
have *emeasure N A* = *emeasure N* {*x*∈Ω. *x* ∈ *A*}
using *ae subset A* **by** (*intro emeasure-eq-AE*) *auto*
also have ... = (∫ +*x*. *emeasure N* {*x*} ∂*count-space* {*x*∈Ω. *x* ∈ *A*})
using *A subset* **by** (*intro emeasure-countable-singleton*) *auto*
also have ... = (∫ +*x*. *emeasure M* {*x*} ∂*count-space* {*x*∈Ω. *x* ∈ *A*})
by (*intro nn-integral-cong eq*[*symmetric*]) *auto*
also have ... = *emeasure M* {*x*∈Ω. *x* ∈ *A*}
using *A subset* **by** (*intro emeasure-countable-singleton*[*symmetric*]) *auto*
also have ... = *emeasure M A*
using *ae A subset* **by** (*intro emeasure-eq-AE*) *auto*
finally show *emeasure M A* = *emeasure N A* **..**
qed *simp*

lemma *measurable-le*[*measurable* (*raw*)]:
fixes *f* :: *′a* ⇒ *′b*::{*second-countable-topology*, *linorder-topology*}
assumes *f* ∈ *borel-measurable M g* ∈ *borel-measurable M*
shows *Measurable.pred M* (λ*x*. *f x* ≤ *g x*)
unfolding *pred-def* **by** (*intro borel-measurable-le assms*)

lemma *measurable-eq*[*measurable* (*raw*)]:
fixes *f* :: *′a* ⇒ *′b*::{*second-countable-topology*, *linorder-topology*}
assumes *f* ∈ *borel-measurable M g* ∈ *borel-measurable M*
shows *Measurable.pred M* (λ*x*. *f x* = *g x*)
unfolding *pred-def* **by** (*intro borel-measurable-eq assms*)

context
fixes *M* :: *′a measure*
assumes *singleton-null-set*: *x* ∈ *space M* ⟹ {*x*} ∈ *null-sets M*
begin

lemma *countable-null-set*:
assumes *countable A A* ⊆ *space M*
shows *A* ∈ *null-sets M*
proof −
have (⋃*x*∈*A*. {*x*}) ∈ *null-sets M* **using** *assms*
by (*intro null-sets-UN′ assms singleton-null-set*) *auto*

also have $(\bigcup x{\in}A.\ \{x\}) = A$ **by** *simp*
finally show *?thesis* **.**
qed

lemma *finite-null-set*:
assumes *finite A* $A \subseteq$ *space M*
shows $A \in$ *null-sets M*
using *countable-finite*[*OF assms*(*1*)] *countable-null-set*[*OF - assms*(*2*)] **by** *simp*

end

lemma *measurable-inj-on-finite*:
assumes *fin* [*measurable*]: *finite I*
assumes [*measurable*]: $\bigwedge i\ j.$ *Measurable.pred* $(M\ i \bigotimes_M M\ j)\ (\lambda(x,y).\ x = y)$
shows *Measurable.pred* $(Pi_M\ I\ M)\ (\lambda x.$ *inj-on x I*$)$ **unfolding** *inj-on-def*
by *measurable*

lemma *almost-everywhere-not-in-countable-set*:
assumes *countable A*
assumes [*measurable*]: *Measurable.pred* $(M \bigotimes_M M)\ (\lambda(x,y).\ x = y)$
assumes *null*: $\bigwedge x.\ x \in$ *space* $M \Longrightarrow \{x\} \in$ *null-sets M*
shows *AE x in M*. $x \notin A$
proof −
have $A \cap$ *space* $M \in$ *null-sets M*
by (*rule countable-null-set*) (*insert assms*(*1*), *auto intro*: *null*)
hence *AE x in M*. $\forall y{\in}A.\ x \neq y$ **by** (*rule AE-I'*) *auto*
also have *?this* $\longleftrightarrow$ *?thesis* **by** (*intro AE-cong*) *auto*
finally show *?thesis* **.**
qed

lemma *almost-everywhere-inj-on-PiM*:
assumes *fin*: *finite I* **and** *prob-space*: $\bigwedge i.\ i \in I \Longrightarrow$ *prob-space* $(M\ i)$
assumes [*measurable*]: $\bigwedge i\ j.$ *Measurable.pred* $(M\ i \bigotimes_M M\ j)\ (\lambda(x,y).\ x = y)$
assumes *null*: $\bigwedge i\ x.\ i \in I \Longrightarrow x \in$ *space* $(M\ i) \Longrightarrow \{x\} \in$ *null-sets* $(M\ i)$
shows *AE f in* $(\Pi_M\ i{\in}I.\ M\ i)$. *inj-on f I*
proof −
note [*measurable*] = *measurable-inj-on-finite*
define I' **where** $I' = I$
hence $I \subseteq I'$ **by** *simp*
from *fin* **and** *this* **show** *?thesis*
proof (*induction I rule*: *finite-induct*)
case (*insert i I*)
interpret *pair-sigma-finite M i PiM I M*
unfolding *pair-sigma-finite-def* **using** *insert.prems*
by (*auto intro*!: *prob-space-imp-sigma-finite prob-space prob-space-PiM simp*: *I'-def*)
from *insert.hyps* **have** [*measurable*]: *finite* (*insert i I*) **by** *simp*
have *PiM* (*insert i I*) *M* = *distr* (*M i* $\bigotimes_M$ Pi_M *I M*) (Pi_M (*insert i I*) *M*) ($\lambda(x,\ X).\ X(i := x)$)

```
    using insert.prems
    by (intro distr-pair-PiM-eq-PiM [symmetric] prob-space) (auto simp: I'-def)
  also have (AE f in .... inj-on f (insert i I)) ⟷
          (AE x in M i ⨂M Pi_M I M. inj-on ((snd x)(i := fst x)) (insert i I))
    by (subst AE-distr-iff; measurable) (simp add: case-prod-unfold)?
  also have ... = (AE x in M i. AE y in Pi_M I M. inj-on (y(i := x)) (insert i
I))
    by (rule AE-pair-iff [symmetric]) measurable
  also have ... ⟷ (AE x in M i. AE y in Pi_M I M. inj-on (y(i := x)) I) ∧
                  (AE x in M i. AE y in Pi_M I M. x ∉ y(i := x) ‘ (I − {i})) by
simp
  also have ...
  proof (rule conjI, goal-cases)
    case 1
    from insert.prems have AE f in Pi_M I M. inj-on f I by (intro insert.IH)
auto
    hence AE f in Pi_M I M. inj-on (f(i := x)) I for x
      by eventually-elim (insert insert.hyps, auto simp: inj-on-def)
    thus ?case by blast
  next
    note [measurable] = ‹finite I›
    {
      fix f
      have f ‘ I ∩ space (M i) ∈ null-sets (M i)
        by (rule finite-null-set)
           (insert insert.hyps insert.prems, auto intro!: null simp: I'-def)
      hence AE x in M i. x ∉ f(i := x) ‘ I
        by (rule AE-I') (insert insert.hyps, auto split: if-splits)
      also have (AE x in M i. x ∉ f(i := x) ‘ I) ⟷ (AE x in M i. ∀ y∈I. f y
≠ x)
        using insert.hyps by (intro AE-cong) (auto split: if-splits)
      finally have ... .
    }
    hence AE f in Pi_M I M. AE x in M i. ∀ y∈I. f y ≠ x by blast
    hence AE x in M i. AE f in Pi_M I M. ∀ y∈I. f y ≠ x
      by (subst AE-commute) simp-all
    also have ?this ⟷ (AE x in M i. AE y in Pi_M I M. x ∉ y(i := x) ‘ (I −
{i}))
      using insert.hyps by (intro AE-cong) (auto split: if-splits)
    finally show ... .
  qed
  finally show ?case .
 qed auto
qed


lemma null-sets-uniform-measure:
 assumes A ∈ sets M emeasure M A ≠ ∞
 shows   null-sets (uniform-measure M A) = (λB. A ∩ B) −‘ null-sets M ∩ sets
```

M
using *assms* **by** (*auto simp*: *null-sets-def*)

lemma *almost-everywhere-avoid-finite*:
assumes *fin*: *finite I*
shows *AE f in* (Π_M *i*∈*I*. *uniform-measure lborel* {(*0*::*real*)..*1*}). *inj-on f I*
proof (*intro almost-everywhere-inj-on-PiM fin prob-space-uniform-measure*)
fix *x* :: *real*
show {*x*} ∈ *null-sets* (*uniform-measure lborel* {*0*..*1*})
by (*cases x* ∈ {*0*..*1*}) (*auto simp*: *null-sets-uniform-measure*)
qed *auto*

lemma *almost-everywhere-avoid-countable*:
assumes *countable A*
shows *AE x in uniform-measure lborel* {(*0*::*real*)..*1*}. *x* ∉ *A*
proof (*intro almost-everywhere-not-in-countable-set assms prob-space-uniform-measure*)
fix *x* :: *real*
show {*x*} ∈ *null-sets* (*uniform-measure lborel* {*0*..*1*})
by (*cases x* ∈ {*0*..*1*}) (*auto simp*: *null-sets-uniform-measure*)
qed *auto*

lemma *measure-pmf-of-set*:
assumes *A* ≠ {} **and** *finite A*
shows *measure-pmf* (*pmf-of-set A*) = *uniform-measure* (*count-space UNIV*) *A*
using *assms*
by (*intro measure-eqI*)
(*auto simp*: *emeasure-pmf-of-set divide-ennreal* [*symmetric*] *card-gt-0-iff*
ennreal-of-nat-eq-real-of-nat)

lemma *emeasure-distr-restrict*:
assumes *f* ∈ *M* →$_M$ *N f* ∈ *M'* →$_M$ *N' A* ∈ *sets N' sets M'* ⊆ *sets M sets N'* ⊆ *sets N*
assumes ⋀*X*. *X* ∈ *sets M'* ⟹ *emeasure M X* = *emeasure M' X*
assumes ⋀*X*. *X* ∈ *sets M* ⟹ *X* ⊆ *space M* − *space M'* ⟹ *emeasure M X* = *0*
shows *emeasure* (*distr M N f*) *A*= *emeasure* (*distr M' N' f*) *A*
proof −
have *space-subset*: *space M'* ⊆ *space M*
using ‹*sets M'* ⊆ *sets M*› **by** (*simp add*: *sets-le-imp-space-le*)
have *emeasure* (*distr M N f*) *A* = *emeasure M* (*f* −‘ *A* ∩ *space M*)
using *assms* **by** (*subst emeasure-distr*) *auto*
also have *f* −‘ *A* ∩ *space M* = *f* −‘ *A* ∩ *space M'* ∪ *f* −‘ *A* ∩ (*space M* − *space M'*)
using *space-subset* **by** *blast*
also have *emeasure M* ... = *emeasure M* (*f* −‘ *A* ∩ *space M'*)
proof (*intro emeasure-Un-null-set*)
show *f* −‘ *A* ∩ *space M'* ∈ *sets M*
using *assms* **by** *auto*
have *f* −‘ *A* ∩ (*space M* − *space M'*) ∈ *sets M*

```
    using assms by (metis Int-Diff measurable-sets sets.Diff sets.top subsetCE)
  moreover from this have emeasure M (f -` A ∩ (space M - space M')) = 0
    by (intro assms) auto
  ultimately show f -` A ∩ (space M - space M') ∈ null-sets M
    unfolding null-sets-def by blast
 qed
 also have ... = emeasure M' (f -` A ∩ space M')
  using assms by (intro assms) auto
 also have ... = emeasure (distr M' N' f) A
  using assms by (subst emeasure-distr) auto
 finally show ?thesis .
qed

lemma distr-uniform-measure-count-space-inj:
 assumes inj-on f A' A' ⊆ A f ` A ⊆ B finite A'
 shows   distr (uniform-measure (count-space A) A') (count-space B) f =
         uniform-measure (count-space B) (f ` A') (is ?lhs = ?rhs)
proof (rule measure-eqI, goal-cases)
 case (2 X)
 hence X-subset: X ⊆ B by simp
 from assms have eq: f ` A' ∩ X = f ` (A' ∩ (f -` X ∩ A))
  by auto
 from assms have [measurable]: f ∈ count-space A →M count-space B
  by (subst measurable-count-space-eq1) auto
 from X-subset have emeasure ?lhs X =
                    emeasure (uniform-measure (count-space A) A') (f -` X ∩ A)
  by (subst emeasure-distr) auto
 also from assms X-subset
    have ... = emeasure (count-space A) (A' ∩ (f -` X ∩ A)) / emeasure
(count-space A) A'
  by (intro emeasure-uniform-measure) auto
 also from assms have ... = of-nat (card (A' ∩ (f -` X ∩ A))) / of-nat (card
A')
  by (subst (1 2) emeasure-count-space) auto
 also have card (A' ∩ (f -` X ∩ A)) = card (f ` (A' ∩ (f -` X ∩ A)))
  using assms by (intro card-image [symmetric]) (auto simp: inj-on-def)
 also have f ` (A' ∩ (f -` X ∩ A)) = f ` A' ∩ X
  using assms by auto
 also have of-nat (card A') = of-nat (card (f ` A'))
  using assms by (subst card-image) auto
 also have of-nat (card (f ` A' ∩ X)) / ... =
           emeasure (count-space B) (f ` A' ∩ X) / emeasure (count-space B) (f `
A')
  using assms by (subst (1 2) emeasure-count-space) auto
 also from assms X-subset have ... = emeasure ?rhs X
  by (intro emeasure-uniform-measure [symmetric]) auto
 finally show ?case .
qed simp-all
```

lemma (**in** *pair-prob-space*) *pair-measure-bind*:
assumes [*measurable*]: $f \in M1 \bigotimes_M M2 \to_M$ *subprob-algebra N*
shows $(M1 \bigotimes_M M2) \gg\!= f = do\ \{x \leftarrow M1;\ y \leftarrow M2;\ f\ (x,\ y)\}$
proof −
note $M1 = M1.$*prob-space-axioms* **and** $M2 = M2.$*prob-space-axioms*
have [*measurable*]: $M1 \in$ *space* (*subprob-algebra M1*)
by (*rule M1.M-in-subprob*)
have [*measurable*]: $M2 \in$ *space* (*subprob-algebra M2*)
by (*rule M2.M-in-subprob*)
have $(M1 \bigotimes_M M2) = M1 \gg\!= (\lambda x.\ M2 \gg\!= (\lambda y.\ return\ (M1 \bigotimes_M M2)\ (x,\ y)))$
by (*subst pair-measure-eq-bind*) *simp-all*
also have $\ldots \gg\!= f = M1 \gg\!= (\lambda x.\ (M2 \gg\!= (\lambda y.\ return\ (M1 \bigotimes_M M2)\ (x,\ y))) \gg\!= f)$
by (*rule bind-assoc*) *measurable*
also have $\ldots = M1 \gg\!= (\lambda x.\ M2 \gg\!= (\lambda xa.\ return\ (M1 \bigotimes_M M2)\ (x,\ xa)) \gg\!= f))$
by (*intro bind-cong refl bind-assoc*) *measurable*
also have $\ldots = do\ \{x \leftarrow M1;\ y \leftarrow M2;\ f\ (x,\ y)\}$
by (*intro bind-cong refl bind-return*)
(*measurable*, *simp-all add*: *space-pair-measure*)
finally show *?thesis* **.**
qed

lemma *count-space-singleton-conv-return*:
count-space $\{x\} =$ *return* (*count-space* $\{x\}$) x
proof (*rule measure-eqI*)
fix A **assume** $A \in$ *sets* (*count-space* $\{x\}$)
hence $A \subseteq \{x\}$ **by** *auto*
hence $A = \{\} \vee A = \{x\}$ **by** (*cases* $x \in A$) *auto*
thus *emeasure* (*count-space* $\{x\}$) $A =$ *emeasure* (*return* (*count-space* $\{x\}$) x) A
by *auto*
qed *auto*

lemma *distr-count-space-singleton* [*simp*]:
$f\ x \in$ *space* $M \Longrightarrow$ *distr* (*count-space* $\{x\}$) $M\ f =$ *return* M $(f\ x)$
by (*subst count-space-singleton-conv-return*, *subst distr-return*) *simp-all*

lemma *uniform-measure-count-space-singleton* [*simp*]:
assumes $\{x\} \in$ *sets* M *emeasure* M $\{x\} \neq 0$ *emeasure* M $\{x\} < \infty$
shows *uniform-measure* M $\{x\} =$ *return* M x
proof (*rule measure-eqI*)
fix A **assume** A: $A \in$ *sets* (*uniform-measure* M $\{x\}$)
show *emeasure* (*uniform-measure* M $\{x\}$) $A =$ *emeasure* (*return* M x) A
by (*cases* $x \in A$) (*insert assms A*, *auto*)
qed *simp-all*

lemma *PiM-uniform-measure-permute*:
fixes $a\ b$ :: *real*
assumes g *permutes* A $a < b$

shows *distr* (*PiM A* (λ-. *uniform-measure lborel* {*a*..*b*})) (*PiM A* (λ-. *lborel*)) (λ*f*. *f* ∘ *g*) =
PiM A (λ-. *uniform-measure lborel* {*a*..*b*})
proof −
have *distr* (*PiM A* (λ-. *uniform-measure lborel* {*a*..*b*})) (*PiM A* (λ-. *lborel*)) (λ*f*. *f* ∘ *g*) =
distr (*PiM A* (λ-. *uniform-measure lborel* {*a*..*b*}))
(*PiM A* (λ-. *uniform-measure lborel* {*a*..*b*})) (λ*f*. λ*x*∈*A*. *f* (*g x*)) **using** *assms*
by (*intro distr-cong sets-PiM-cong refl*)
(*auto simp*: *fun-eq-iff space-PiM PiE-def extensional-def permutes-in-image*[*of g A*])
also from *assms* **have** ... = Pi_M *A* (λ*i*. *uniform-measure lborel* {*a*..*b*})
by (*intro distr-PiM-reindex*)
(*auto simp*: *permutes-inj-on permutes-in-image*[*of g A*] *intro*!: *prob-space-uniform-measure*)
finally show *?thesis* **.**
qed

lemma *ennreal-fact* [*simp*]: *ennreal* (*fact n*) = *fact n*
by (*induction n*) (*auto simp*: *algebra-simps ennreal-mult' ennreal-of-nat-eq-real-of-nat*)

lemma *inverse-ennreal-unique*:
assumes *a* ∗ (*b* :: *ennreal*) = *1*
shows *b* = *inverse a*
using *assms*
by (*metis divide-ennreal-def ennreal-inverse-1 ennreal-top-eq-mult-iff mult.comm-neutral*
mult-divide-eq-ennreal mult-eq-0-iff semiring-normalization-rules(*7*))

end

2 Treaps

theory *Treap*
imports
HOL−*Library.Tree*
begin

definition *treap* :: ('*k*::*linorder* ∗ '*p*::*linorder*) *tree* ⇒ *bool* **where**
treap t = (*bst* (*map-tree fst t*) ∧ *heap* (*map-tree snd t*))

abbreviation *keys t* ≡ *set-tree* (*map-tree fst t*)
abbreviation *prios t* ≡ *set-tree* (*map-tree snd t*)

function *treap-of* :: ('*k*::*linorder* ∗ '*p*::*linorder*) *set* ⇒ ('*k* ∗ '*p*) *tree* **where**
treap-of KP = (*if infinite KP* ∨ *KP* = {} *then Leaf else*
let m = *arg-min-on snd KP*;
L = {*p* ∈ *KP*. *fst p* < *fst m*};
R = {*p* ∈ *KP*. *fst p* > *fst m*}

```
  in Node (treap-of L) m (treap-of R))
by pat-completeness auto
termination
proof (relation measure card)
  show wf (measure card)  by simp
next
  fix KP :: ('a × 'b) set and m L
  assume KP: ¬ (infinite KP ∨ KP = {})
  and m: m = arg-min-on snd KP
  and L: L = {p ∈ KP. fst p < fst m}
  have m ∈ KP using KP arg-min-if-finite(1) m by blast
  thus  (L, KP) ∈ measure card using KP L by(auto intro!: psubset-card-mono)
next
  fix KP :: ('a × 'b) set and m R
  assume KP: ¬ (infinite KP ∨ KP = {})
  and m: m = arg-min-on snd KP
  and R: R = {p ∈ KP. fst m < fst p}
  have m ∈ KP using KP arg-min-if-finite(1) m by blast
  thus  (R, KP) ∈ measure card using KP R by(auto intro!: psubset-card-mono)
qed

declare treap-of.simps [simp del]

lemma treap-of-unique:
  ⟦ treap t;  inj-on snd (set-tree t) ⟧
  ⟹ treap-of (set-tree t) = t
proof(induction set-tree t arbitrary: t rule: treap-of.induct)
  case (1 t)
  show ?case
  proof (cases infinite (set-tree t) ∨ set-tree t = {})
    case True
    thus ?thesis by(simp add: treap-of.simps)
  next
    case False
    let ?m = arg-min-on snd (set-tree t)
    let ?L = {p ∈ set-tree t. fst p < fst ?m}
    let ?R = {p ∈ set-tree t. fst p > fst ?m}
    obtain l a r where t: t = Node l a r
      using False by (cases t) auto
    have ∀ kp ∈ set-tree t. snd a ≤ snd kp
      using 1.prems(1)
      by(auto simp add: t treap-def ball-Un)
        (metis image-eqI snd-conv tree.set-map)+
    hence a = ?m
     by (metis 1.prems(2) False arg-min-if-finite(1) arg-min-if-finite(2) inj-on-def

         le-neq-trans t tree.set-intros(2))
    have treap l treap r using 1.prems(1) by(auto simp: treap-def t)
    have l: set-tree l = {p ∈ set-tree t. fst p < fst a}
```

```
    using 1.prems(1) by(auto simp add: treap-def t ball-Un tree.set-map)
  have r: set-tree r = {p ∈ set-tree t. fst p > fst a}
    using 1.prems(1) by(auto simp add: treap-def t ball-Un tree.set-map)
  have l = treap-of ?L
    using 1.hyps(1)[OF False ‹a = ?m› l r ‹treap l›]
      l ‹a = ?m› 1.prems(2)
    by (fastforce simp add: inj-on-def)
  have r = treap-of ?R
    using 1.hyps(2)[OF False ‹a = ?m› l r ‹treap r›]
      r ‹a = ?m› 1.prems(2)
    by (fastforce simp add: inj-on-def)
  have t = Node (treap-of ?L) ?m (treap-of ?R)
    using ‹a = ?m› ‹l = treap-of ?L› ‹r = treap-of ?R› by(subst t) simp
  thus ?thesis using False
    by (subst treap-of.simps) simp
 qed
qed

lemma treap-unique:
 ⟦ treap t1; treap t2; set-tree t1 = set-tree t2; inj-on snd (set-tree t1) ⟧
 ⟹ t1 = t2
 for t1 t2 :: ('k::linorder * 'p::linorder) tree
by (metis treap-of-unique)

fun ins :: 'k::linorder ⇒ 'p::linorder ⇒ ('k × 'p) tree ⇒ ('k × 'p) tree where
ins k p Leaf = ⟨Leaf, (k,p), Leaf⟩ |
ins k p ⟨l, (k1,p1), r⟩ =
 (if k < k1 then
    (case ins k p l of
      ⟨l2, (k2,p2), r2⟩ ⇒
        if p1 ≤ p2 then ⟨⟨l2, (k2,p2), r2⟩, (k1,p1), r⟩
        else ⟨l2, (k2,p2), ⟨r2, (k1,p1), r⟩⟩)
  else
  if k > k1 then
    (case ins k p r of
      ⟨l2, (k2,p2), r2⟩ ⇒
        if p1 ≤ p2 then ⟨l, (k1,p1), ⟨l2, (k2,p2), r2⟩⟩
        else ⟨⟨l, (k1,p1), l2⟩, (k2,p2), r2⟩)
  else ⟨l, (k1,p1), r⟩)

lemma ins-neq-Leaf: ins k p t ≠ ⟨⟩
 by (induction t rule: ins.induct) (auto split: tree.split)

lemma keys-ins: keys (ins k p t) = Set.insert k (keys t)
proof (induction t rule: ins.induct)
 case 2
 then show ?case
   by (simp add: ins-neq-Leaf split: tree.split prod.split) (safe; fastforce)
qed (simp)
```

lemma *prios-ins*: *prios* (*ins k p t*) ⊆ {*p*} ∪ *prios t*
apply(*induction t rule*: *ins.induct*)
apply *simp*
apply (*simp add*: *ins-neq-Leaf split*: *tree.split prod.split*)
by (*safe*; *fastforce*)

lemma *prios-ins'*: *k* ∉ *keys t* ⟹ *prios* (*ins k p t*) = {*p*} ∪ *prios t*
apply(*induction t rule*: *ins.induct*)
apply *simp*
apply (*simp add*: *ins-neq-Leaf split*: *tree.split prod.split*)
by (*safe*; *fastforce*)

lemma *set-tree-ins*: *set-tree* (*ins k p t*) ⊆ {(*k*,*p*)} ∪ *set-tree t*
by (*induction t rule*: *ins.induct*) (*auto simp add*: *ins-neq-Leaf split*: *tree.split prod.split*)

lemma *set-tree-ins'*: *k* ∉ *keys t* ⟹ {(*k*,*p*)} ∪ *set-tree t* ⊆ *set-tree* (*ins k p t*)
by (*induction t rule*: *ins.induct*) (*auto simp add*: *ins-neq-Leaf split*: *tree.split prod.split*)

lemma *set-tree-ins-eq*: *k* ∉ *keys t* ⟹ *set-tree* (*ins k p t*) = {(*k*,*p*)} ∪ *set-tree t*
using *set-tree-ins set-tree-ins'* **by** *blast*

lemma *prios-ins-special*:
⟦ *ins k p t* = *Node l* (*k'*,*p'*) *r*; *p'* = *p*; *p* ∈ *prios r* ∪ *prios l* ⟧
⟹ *p* ∈ *prios t*
by (*induction k p t arbitrary*: *l k' p' r rule*: *ins.induct*)
(*fastforce simp add*: *ins-neq-Leaf split*: *tree.splits prod.splits if-splits*)+

lemma *treap-NodeI*:
⟦ *treap l*; *treap r*;
∀ *k'* ∈ *keys l*. *k'* < *k*; ∀ *k'* ∈ *keys r*. *k* < *k'*;
∀ *p'* ∈ *prios l*. *p* ≤ *p'*; ∀ *p'* ∈ *prios r*. *p* ≤ *p'* ⟧
⟹ *treap* (*Node l* (*k*,*p*) *r*)
by (*auto simp*: *treap-def*)

lemma *treap-rotate1*:
assumes *treap l2 treap r2 treap r* ¬ *p1* ≤ *p2 k* < *k1* **and**
ins: *ins k p l* = *Node l2* (*k2*,*p2*) *r2* **and** *treap-ins*: *treap* (*ins k p l*)
and *treap*: *treap* ⟨*l*, (*k1*, *p1*), *r*⟩
shows *treap* (*Node l2* (*k2*,*p2*) (*Node r2* (*k1*,*p1*) *r*))
proof(*rule treap-NodeI*[*OF* ‹*treap l2*› *treap-NodeI*[*OF* ‹*treap r2*› ‹*treap r*›]])
from *keys-ins*[*of k p l*] **have** *1*: *keys r2* ⊆ {*k*} ∪ *keys l* **by**(*auto simp*: *ins*)
from *treap* **have** *2*: ∀ *k'*∈*keys l*. *k'* < *k1* **by** (*simp add*: *treap-def*)
show ∀ *k'*∈*keys r2*. *k'* < *k1* **using** *1 2* ‹*k* < *k1*› **by** *blast*
next
from *treap* **have** *2*: ∀ *p'*∈*prios l*. *p1* ≤ *p'* **by** (*simp add*: *treap-def*)
show ∀ *p'*∈*prios r2*. *p1* ≤ *p'*

```
  proof
    fix p' assume p' ∈ prios r2
    hence p' = p ∨ p' ∈ prios l using prios-ins[of k p l] ins by auto
    thus p1 ≤ p'
    proof
      assume [simp]: p' = p
      have p2 = p ∨ p2 ∈ prios l using prios-ins[of k p l] ins by simp
      thus p1 ≤ p'
      proof
        assume p2 = p
        thus p1 ≤ p'
          using prios-ins-special[OF ins] ‹p' ∈ prios r2› 2 by auto
      next
        assume p2 ∈ prios l
        thus p1 ≤ p' using 2 ‹¬ p1 ≤ p2› by blast
      qed
    next
      assume p' ∈ prios l
      thus p1 ≤ p' using 2 by blast
    qed
  qed
next
  have k2 = k ∨ k2 ∈ keys l using keys-ins[of k p l] ins by (auto)
  hence 1: k2 < k1
  proof
    assume k2 = k thus k2 < k1 using ‹k < k1› by simp
  next
    assume k2 ∈ keys l
    thus k2 < k1 using treap by (auto simp: treap-def)
  qed
  have 2: ∀k'∈keys r2. k2 < k'
    using treap-ins by(simp add: ins treap-def)
  have 3: ∀k'∈keys r. k2 < k'
    using 1 treap by(auto simp: treap-def)
  show ∀k'∈keys ⟨r2, (k1, p1), r⟩. k2 < k' using 1 2 3 by auto
next
  show ∀p'∈prios ⟨r2, (k1, p1), r⟩. p2 ≤ p'
    using ins treap-ins treap ‹¬ p1 ≤ p2› by (auto simp: treap-def ball-Un)
qed (use ins treap-ins treap in ‹auto simp: treap-def ball-Un›)


lemma treap-rotate2:
  assumes treap l treap l2 treap r2 ¬ p1 ≤ p2 k1 < k and
  ins: ins k p r = Node l2 (k2,p2) r2 and treap-ins: treap (ins k p r)
  and treap: treap ⟨l, (k1, p1), r⟩
  shows treap (Node (Node l (k1,p1) l2) (k2,p2) r2)
proof(rule treap-NodeI[OF treap-NodeI[OF ‹treap l› ‹treap l2›] ‹treap r2›])
  from keys-ins[of k p r] have 1: keys l2 ⊆ {k} ∪ keys r by(auto simp: ins)
  from treap have 2: ∀k'∈keys r. k1 < k' by (simp add: treap-def)
```

```
  show ∀k'∈keys l2. k1 < k' using 1 2 ‹k1 < k› by blast
next
  from treap have 2: ∀p'∈prios r. p1 ≤ p' by (simp add: treap-def)
  show ∀p'∈prios l2. p1 ≤ p'
  proof
    fix p' assume p' ∈ prios l2
    hence p' = p ∨ p' ∈ prios r using prios-ins[of k p r] ins by auto
    thus p1 ≤ p'
    proof
      assume [simp]: p' = p
      have p2 = p ∨ p2 ∈ prios r using prios-ins[of k p r] ins by simp
      thus p1 ≤ p'
      proof
        assume p2 = p
        thus p1 ≤ p'
          using prios-ins-special[OF ins] ‹p' ∈ prios l2› 2 by auto
      next
        assume p2 ∈ prios r
        thus p1 ≤ p' using 2 ‹¬ p1 ≤ p2› by blast
      qed
    next
      assume p' ∈ prios r
      thus p1 ≤ p' using 2 by blast
    qed
  qed
next
  have k2 = k ∨ k2 ∈ keys r using keys-ins[of k p r] ins by (auto)
  hence 1: k1 < k2
  proof
    assume k2 = k thus k1 < k2 using ‹k1 < k› by simp
  next
    assume k2 ∈ keys r
    thus k1 < k2 using treap by (auto simp: treap-def)
  qed
  have 2: ∀k'∈keys l. k' < k2 using 1 treap by(auto simp: treap-def)
  have 3: ∀k'∈keys l2. k' < k2
    using treap-ins by(auto simp: ins treap-def)
  show ∀k'∈keys ⟨l, (k1, p1), l2⟩. k' < k2 using 1 2 3 by auto
next
  show ∀p'∈prios ⟨l, (k1, p1), l2⟩. p2 ≤ p'
    using ins treap-ins treap ‹¬ p1 ≤ p2› by (auto simp: treap-def ball-Un)
qed (use ins treap-ins treap in ‹auto simp: treap-def ball-Un›)

lemma treap-ins: treap t ⟹ treap (ins k p t)
proof(induction t rule: ins.induct)
  case 1 thus ?case by (simp add: treap-def)
next
  case (2 k p l k1 p1 r)
  have treap l treap r
```

```
  using 2.prems by(auto simp: treap-def tree.set-map)
show ?case
proof cases
  assume k < k1
  obtain l2 k2 p2 r2 where ins: ins k p l = Node l2 (k2,p2) r2
    by (metis ins-neq-Leaf neq-Leaf-iff prod.collapse)
  note treap-ins = 2.IH(1)[OF ‹k < k1› ‹treap l›]
  hence treap l2 treap r2 using ins by (auto simp add: treap-def)
  show ?thesis
  proof cases
    assume p1 ≤ p2
    have treap (Node (Node l2 (k2,p2) r2) (k1,p1) r)
      apply(rule treap-NodeI[OF treap-ins[unfolded ins] ‹treap r›])
      using ins treap-ins ‹k < k1› 2.prems keys-ins[of k p l]
      by (auto simp add: treap-def ball-Un order-trans[OF ‹p1 ≤ p2›])
    thus ?thesis using ‹k < k1› ins ‹p1 ≤ p2› by simp
  next
    assume ¬ p1 ≤ p2
    have treap (Node l2 (k2,p2) (Node r2 (k1,p1) r))
      by(rule treap-rotate1[OF ‹treap l2› ‹treap r2›  ‹treap r› ‹¬ p1 ≤ p2›
          ‹k < k1› ins treap-ins 2.prems])
    thus ?thesis using ‹k < k1› ins ‹¬ p1 ≤ p2› by simp
  qed
next
  assume ¬ k < k1
  show ?thesis
  proof cases
  assume k > k1
  obtain l2 k2 p2 r2 where ins: ins k p r = Node l2 (k2,p2) r2
    by (metis ins-neq-Leaf neq-Leaf-iff prod.collapse)
  note treap-ins = 2.IH(2)[OF ‹¬ k < k1› ‹k > k1› ‹treap r›]
  hence treap l2 treap r2 using ins by (auto simp add: treap-def)
  have fst: ∀ k' ∈ set-tree (map-tree fst (ins k p r)).
              k' = k ∨ k' ∈ set-tree (map-tree fst r)
    by(simp add: keys-ins)
  show ?thesis
  proof cases
    assume p1 ≤ p2
    have treap (Node l (k1,p1) (ins k p r))
      apply(rule treap-NodeI[OF ‹treap l› treap-ins])
      using ins treap-ins ‹k > k1› 2.prems keys-ins[of k p r]
      by (auto simp: treap-def ball-Un order-trans[OF ‹p1 ≤ p2›])
    thus ?thesis using ‹¬ k < k1› ‹k > k1› ins ‹p1 ≤ p2› by simp
  next
    assume ¬ p1 ≤ p2
    have treap (Node (Node l (k1,p1) l2) (k2,p2) r2)
      by(rule treap-rotate2[OF ‹treap l› ‹treap l2› ‹treap r2› ‹¬ p1 ≤ p2›
          ‹k1 < k› ins treap-ins 2.prems])
    thus ?thesis using ‹¬ k < k1›  ‹k > k1› ins ‹¬ p1 ≤ p2› by simp
```

```
    qed
    next
      assume ¬ k > k1
      hence k = k1 using ‹¬ k < k1› by auto
      thus ?thesis using 2.prems by(simp)
    qed
  qed
qed

lemma treap-of-set-tree-unique:
  ⟦ finite A; inj-on fst A; inj-on snd A ⟧
  ⟹ set-tree (treap-of A) = A
proof(induction A rule: treap-of.induct)
  case (1 A)
  note IH = 1
  show ?case
  proof (cases infinite A ∨ A = {})
    assume infinite A ∨ A = {}
    with IH show ?thesis by (simp add: treap-of.simps)
  next
    assume not-inf-or-empty: ¬ (infinite A ∨ A = {})
    let ?m = arg-min-on snd A
    let ?L = {p ∈ A. fst p < fst ?m}
    let ?R = {p ∈ A. fst p > fst ?m}
    obtain l a r where t: treap-of A = Node l a r
      using not-inf-or-empty
      by (cases treap-of A) (auto simp: Let-def elim!: treap-of.elims split: if-splits)
    have [simp]: inj-on fst {p ∈ A. fst p < fst (arg-min-on snd A)}
                 inj-on snd {p ∈ A. fst p < fst (arg-min-on snd A)}
                 inj-on fst {p ∈ A. fst (arg-min-on snd A) < fst p}
                 inj-on snd {p ∈ A. fst (arg-min-on snd A) < fst p}
      using IH by (auto intro: inj-on-subset)
    have lr: l = treap-of ?L r = treap-of ?R
      using t by (auto simp: Let-def elim: treap-of.elims split: if-splits)
    then have l: set-tree l = ?L
      using not-inf-or-empty IH by auto
     have r = treap-of ?R
      using t by (auto simp: Let-def elim: treap-of.elims split: if-splits)
    then have r: set-tree r = ?R
      using not-inf-or-empty IH(2) by (auto)
    have a: a = ?m
      using t by (elim treap-of.elims) (simp add: Let-def split: if-splits)
    have a ≠ fst (arg-min-on snd A) if (a,b) ∈ A (a, b) ≠ arg-min-on snd A for
a b
       using IH(4,5) that not-inf-or-empty arg-min-if-finite(1) inj-on-eq-iff by
fastforce
    then have a < fst (arg-min-on snd A)
      if (a,b) ∈ A (a, b) ≠ arg-min-on snd A fst (arg-min-on snd A) ≥ a for a b
      using le-neq-trans that by auto
```

```
    moreover have arg-min-on snd A ∈ A
      using not-inf-or-empty arg-min-if-finite by auto
    ultimately have A: A = {?m} ∪ ?L ∪ ?R
      by auto
    show ?thesis using l r a A t by force
  qed
qed

lemma treap-of-subset: set-tree (treap-of A) ⊆ A
proof(induction A rule: treap-of.induct)
  case (1 A)
  note IH = 1
  show ?case
  proof (cases infinite A ∨ A = {})
    assume infinite A ∨ A = {}
    with IH show ?thesis by (simp add: treap-of.simps)
  next
    assume not-inf-or-empty: ¬ (infinite A ∨ A = {})
    let ?m = arg-min-on snd A
    let ?L = {p ∈ A. fst p < fst ?m}
    let ?R = {p ∈ A. fst p > fst ?m}
    obtain l a r where t: treap-of A = Node l a r
      using not-inf-or-empty by (cases treap-of A)
                        (auto simp add: Let-def  elim!: treap-of.elims split: if-splits)
    have l = treap-of ?L r = treap-of ?R
      using t by (auto simp: Let-def elim: treap-of.elims split: if-splits)
    have set-tree l ⊆ ?L set-tree r ⊆ ?R
    proof −
      have set-tree (treap-of {p ∈ A. fst p < fst (arg-min-on snd A)})
          ⊆ {p ∈ A. fst p < fst (arg-min-on snd A)}
        by (rule IH) (use not-inf-or-empty in auto)
      then show set-tree l ⊆ ?L
        using ‹l = treap-of ?L› by auto
    next
      have set-tree (treap-of {p ∈ A. fst (arg-min-on snd A) < fst p})
          ⊆ {p ∈ A. fst (arg-min-on snd A) < fst p}
        by (rule IH) (use not-inf-or-empty in auto)
      then show set-tree r ⊆ ?R
        using ‹r = treap-of ?R› by auto
    qed
    moreover have a = ?m
      using t by (auto elim!: treap-of.elims simp add: Let-def split: if-splits)
    moreover have {?m} ∪ ?L ∪ ?R ⊆ A
      using not-inf-or-empty arg-min-if-finite by auto
    ultimately show ?thesis by (auto simp add: t)
  qed
qed

lemma treap-treap-of:
```

```
  treap (treap-of A)
proof(induction A rule: treap-of.induct)
  case (1 A)
  show ?case
  proof (cases infinite A ∨ A = {})
    case True
    with 1 show ?thesis by (simp add: treap-of.simps treap-def)
  next
    case False
    let ?m = arg-min-on snd A
    let ?L = {p ∈ A. fst p < fst ?m}
    let ?R = {p ∈ A. fst p > fst ?m}
    obtain l a r where t: treap-of A = Node l a r
      using False by (cases treap-of A) (auto simp: Let-def elim!: treap-of.elims
split: if-splits)
    have l: l = treap-of ?L
      using t by (auto simp: Let-def elim: treap-of.elims split: if-splits)
    then have treap-l: treap l
      using False by (auto intro: 1)
    from l have keys-l: keys l ⊆ fst ‘ ?L
      by (auto simp add: tree.set-map intro!: image-mono treap-of-subset)
    have r: r = treap-of ?R
      using t by (auto simp: Let-def elim: treap-of.elims split: if-splits)
    then have treap-r: treap r
      using False by (auto intro: 1)
    from r have keys-r: keys r ⊆ fst ‘ ?R
      by (auto simp add: tree.set-map intro!: image-mono treap-of-subset)
    have prios: prios l ⊆ snd ‘ A prios r ⊆ snd ‘ A
      using l r treap-of-subset image-mono by (auto simp add: tree.set-map)
    have a: a = ?m
      using t by(auto simp: Let-def elim: treap-of.elims split: if-splits)
    have prios-l: ⋀x. x ∈ prios l ⟹ snd a ≤ x
      by (drule rev-subsetD[OF - prios(1)]) (use arg-min-least a False in fast)
    have prios-r: ⋀x. x ∈ prios r ⟹ snd a ≤ x
      by (drule rev-subsetD[OF - prios(2)]) (use arg-min-least a False in fast)
    show ?thesis
      using prios-r prios-l treap-l treap-r keys-l keys-r a
      by (auto simp add: t treap-def dest: rev-subsetD[OF - keys-l] rev-subsetD[OF
- keys-r])
    qed
qed

lemma treap-Leaf: treap ⟨⟩
  by (simp add: treap-def)

lemma foldl-ins-treap: treap t ⟹ treap (foldl (λt′ (x, p). ins x p t′) t xs)
  using treap-ins by (induction xs arbitrary: t) auto

lemma foldl-ins-set-tree:
```

assumes *inj-on fst* (*set ys*) *inj-on snd* (*set ys*) *distinct ys fst* ‘ (*set ys*) ∩ *keys t* = {}
shows *set-tree* (*foldl* (λ*t′* (*x*, *p*). *ins x p t′*) *t ys*) = *set ys* ∪ *set-tree t*
using *assms*
by (*induction ys arbitrary*: *t*) (*auto simp add*: *case-prod-beta′ set-tree-ins-eq keys-ins*)

lemma *foldl-ins-treap-of*:
assumes *distinct ys inj-on fst* (*set ys*) *inj-on snd* (*set ys*)
shows (*foldl* (λ*t′* (*x*, *p*). *ins x p t′*) *Leaf ys*) = *treap-of* (*set ys*)
using *assms* **by** (*intro treap-unique*) (*auto simp*: *treap-Leaf foldl-ins-treap foldl-ins-set-tree treap-treap-of treap-of-set-tree-unique*)

end

3 Randomly-permuted lists

theory *Random-List-Permutation*
imports
Probability-Misc
Comparison-Sort-Lower-Bound.Linorder-Relations
begin

3.1 General facts about linear orderings

We define the set of all linear orderings on a given set and show some properties about it.

definition *linorders-on* :: *′a set* ⇒ (*′a* × *′a*) *set set* **where**
linorders-on A = {*R*. *linorder-on A R*}

lemma *linorders-on-empty* [*simp*]: *linorders-on* {} = {{}}
by (*auto simp*: *linorders-on-def linorder-on-def refl-on-def*)

lemma *linorders-finite-nonempty*:
assumes *finite A*
shows *linorders-on A* ≠ {}
proof −
from *finite-distinct-list*[*OF assms*] **obtain** *xs* **where** *set xs* = *A distinct xs* **by** *blast*
hence *linorder-on A* (*linorder-of-list xs*) **by** *auto*
thus *?thesis* **by** (*auto simp*: *linorders-on-def*)
qed

There is an obvious bijection between permutations of a set (i. e. lists with all elements from that set without repetition) and linear orderings on it.

lemma *bij-betw-linorders-on*:
assumes *finite A*

shows *bij-betw linorder-of-list* (*permutations-of-set A*) (*linorders-on A*)
using *bij-betw-linorder-of-list*[*of A*] *assms* **unfolding** *linorders-on-def* **by** *simp*

lemma *sorted-wrt-list-of-set-linorder-of-list* [*simp*]:
assumes *distinct xs*
shows *sorted-wrt-list-of-set* (*linorder-of-list xs*) (*set xs*) = *xs*
by (*rule sorted-wrt-list-of-set-eqI*[*of set xs*]) (*insert assms*, *auto*)

lemma *linorder-of-list-sorted-wrt-list-of-set* [*simp*]:
assumes *linorder-on A R finite A*
shows *linorder-of-list* (*sorted-wrt-list-of-set R A*) = *R*
proof −
from *assms*(*1*) **have** *subset*: $R \subseteq A \times A$ **by** (*auto simp*: *linorder-on-def refl-on-def*)
from *assms* **and** *subset* **show** *?thesis*
by (*auto simp*: *linorder-of-list-def linorder-sorted-wrt-list-of-set sorted-wrt-linorder-index-le-iff*)
qed

lemma *bij-betw-linorders-on′*:
assumes *finite A*
shows *bij-betw* (λ*R*. *sorted-wrt-list-of-set R A*) (*linorders-on A*) (*permutations-of-set A*)
by (*rule bij-betw-byWitness*[**where** *f′* = *linorder-of-list*])
(*insert assms*, *auto simp*: *linorders-on-def permutations-of-set-def linorder-sorted-wrt-list-of-set*)

lemma *finite-linorders-on* [*intro*]:
assumes *finite A*
shows *finite* (*linorders-on A*)
proof −
have *finite* (*permutations-of-set A*) **by** *simp*
also have *?this* ⟷ *finite* (*linorders-on A*)
using *assms* **by** (*rule bij-betw-finite* [*OF bij-betw-linorders-on*])
finally show *?thesis* **.**
qed

Next, we look at the ordering defined by a list that is permuted with some permutation function. For this, we first define the composition of a relation with a function.

definition *map-relation* :: *'a set* ⇒ (*'a* ⇒ *'b*) ⇒ (*'b* × *'b*) *set* ⇒ (*'a* × *'a*) *set*
where
map-relation A f R = {(*x*,*y*)∈*A*×*A*. (*f x*, *f y*) ∈ *R*}

lemma *index-distinct-eqI*:
assumes *distinct xs i* < *length xs xs* ! *i* = *x*
shows *index xs x* = *i*
using *assms* **by** (*induction xs arbitrary*: *i*) (*auto simp*: *nth-Cons split*: *nat.splits*)

lemma *index-permute-list*:
assumes π *permutes* {..<*length xs*} *distinct xs x* ∈ *set xs*

shows *index* (*permute-list* π *xs*) *x* = *inv* π (*index xs x*)
proof −
have ∗: *inv* π *permutes* {..<*length xs*} **by** (*rule permutes-inv*) *fact*
from *assms* **show** *?thesis*
using *assms permutes-in-image*[*OF* ∗]
by (*intro index-distinct-eqI*) (*simp-all add*: *permute-list-nth permutes-inverses*)
qed

lemma *linorder-of-list-permute*:
assumes π *permutes* {..<*length xs*} *distinct xs*
shows *linorder-of-list* (*permute-list* π *xs*) =
map-relation (*set xs*) ((!) *xs* ∘ *inv* π ∘ *index xs*) (*linorder-of-list xs*)
proof −
note ∗ = *permutes-inv*[*OF assms*(*1*)]
have *less*: *inv* π *i* < *length xs* **if** *i* < *length xs* **for** *i*
using *permutes-in-image*[*OF* ∗] **and** *that* **by** *simp*
from *assms* **and** ∗ **show** *?thesis*
by (*auto simp*: *linorder-of-list-def map-relation-def index-nth-id index-permute-list less*)
qed

lemma *inj-on-conv-Ex1*: *inj-on f A* ⟷ (∀ *y*∈*f*‘*A*. ∃!*x*∈*A*. *y* = *f x*)
by (*auto simp*: *inj-on-def*)

lemma *bij-betw-conv-Ex1*: *bij-betw f A B* ⟷ (∀ *y*∈*B*. ∃!*x*∈*A*. *f x* = *y*) ∧ *B* = *f* ‘ *A*
unfolding *bij-betw-def inj-on-conv-Ex1* **by** (*auto simp*: *eq-commute*)

lemma *permutesI*:
assumes *bij-betw f A A* ∀ *x*. *x* ∉ *A* ⟶ *f x* = *x*
shows *f permutes A*
unfolding *permutes-def*
proof (*intro conjI allI impI*)
fix *y*
from *assms* **have** [*simp*]: *f x* ∈ *A* ⟷ *x* ∈ *A* **for** *x*
by (*auto simp*: *bij-betw-def*)
show ∃!*x*. *f x* = *y*
proof (*cases y* ∈ *A*)
case *True*
also from *assms* **have** *A* = *f* ‘ *A* **by** (*auto simp*: *bij-betw-def*)
finally obtain *x* **where** *x* ∈ *A y* = *f x* **by** *auto*
with *assms* **and** ‹*y* ∈ *A*› **show** *?thesis*
by (*intro ex1I*[*of* - *x*]) (*auto simp*: *bij-betw-def dest*: *inj-onD*)
qed (*insert assms*, *auto*)
qed (*insert assms*, *auto*)

We now show the important lemma that any two linear orderings on a finite set can be mapped onto each other by a permutation.

lemma *linorder-permutation-exists*:
assumes *finite A linorder-on A R linorder-on A R′*
obtains *π* **where** *π permutes A R′ = map-relation A π R*
proof −
define *xs* **where** *xs = sorted-wrt-list-of-set R A*
define *ys* **where** *ys = sorted-wrt-list-of-set R′ A*
have *xs-ys*: *distinct xs distinct ys set xs = A set ys = A*
using *assms* **by** (*simp-all add*: *linorder-sorted-wrt-list-of-set xs-def ys-def*)

from *xs-ys* **have** *mset ys = mset xs* **by** (*simp add*: *set-eq-iff-mset-eq-distinct* [*symmetric*])
then obtain *π* **where** *π*: *π permutes* {..<*length xs*} *permute-list π xs = ys*
by (*rule mset-eq-permutation*)
define *π′* **where** *π′ = (λx. if x ∉ A then x else xs ! inv π (index xs x))*
have *π′*: *π′ permutes A*
proof (*rule permutesI*)
have *bij-betw* ((!) *xs* ∘ *inv π*) {..<*length xs*} *A*
by (*rule bij-betw-trans permutes-imp-bij permutes-inv π bij-betw-nth*)+ (*simp-all add*: *xs-ys*)
hence *bij-betw* ((!) *xs* ∘ *inv π* ∘ *index xs*) *A A*
by (*rule bij-betw-trans* [*rotated*] *bij-betw-index*)+
(*insert bij-betw-index*[*of xs A length xs*], *simp-all add*: *xs-ys atLeast0LessThan*)
also have *bij-betw* ((!) *xs* ∘ *inv π* ∘ *index xs*) *A A* ⟷ *bij-betw π′ A A*
by (*rule bij-betw-cong*) (*auto simp*: *π′-def*)
finally show ... **.**
qed (*simp-all add*: *π′-def*)

from *assms* **have** *R′ = linorder-of-list ys* **by** (*simp add*: *ys-def*)
also from *π* **have** *ys = permute-list π xs* **by** *simp*
also have *linorder-of-list* (*permute-list π xs*) =
map-relation A ((!) *xs* ∘ *inv π* ∘ *index xs*) (*linorder-of-list xs*)
using *π* **by** (*subst linorder-of-list-permute*) (*simp-all add*: *xs-ys*)
also from *assms* **have** *linorder-of-list xs = R* **by** (*simp add*: *xs-def*)
finally have *R′ = map-relation A* ((!) *xs* ∘ *inv π* ∘ *index xs*) *R* **.**
also have ... = *map-relation A π′ R* **by** (*auto simp*: *map-relation-def π′-def*)
finally show *?thesis* **using** *π′* **and** *that*[*of π′*] **by** *simp*
qed

We now define the linear ordering defined by some priority function, i. e. a function that injectively associates priorities to every element such that elements with lower priority are smaller in the resulting ordering.

definition *linorder-from-keys* :: *'a set* ⇒ (*'a* ⇒ *'b* :: *linorder*) ⇒ (*'a* × *'a*) *set*
where
linorder-from-keys A f = {(x,y)∈A×A. f x ≤ f y}

lemma *linorder-from-keys-permute*:
assumes *g permutes A*
shows *linorder-from-keys A* (*f* ∘ *g*) = *map-relation A g* (*linorder-from-keys A f*)

using *permutes-in-image*[*OF assms*] **by** (*auto simp*: *map-relation-def linorder-from-keys-def*)

lemma *linorder-on-linorder-from-keys* [*intro*]:
assumes *inj-on f A*
shows *linorder-on A* (*linorder-from-keys A f*)
using *assms* **by** (*auto simp*: *linorder-on-def refl-on-def antisym-def linorder-from-keys-def trans-def total-on-def dest*: *inj-onD*)

lemma *linorder-from-keys-empty* [*simp*]: *linorder-from-keys* {} = (λ-. {})
by (*simp add*: *linorder-from-keys-def fun-eq-iff*)

We now show another important fact, namely that when we draw n values i. i. d. uniformly from a non-trivial real interval, we almost surely get distinct values.

lemma *emeasure-PiM-diagonal*:
fixes *a b* :: *real*
assumes $x \in A$ $y \in A$ $x \neq y$
assumes $a < b$ *finite A*
defines $M \equiv$ *uniform-measure lborel* {*a*..*b*}
shows *emeasure* (*PiM A* (λ-. *M*)) {*h*∈*A* →$_E$ *UNIV*. *h x* = *h y*} = *0*
proof −
from *assms* **have** *M*: *prob-space M* **unfolding** *M-def*
by (*intro prob-space-uniform-measure*) *auto*
then interpret *product-prob-space* λ-. *M A*
unfolding *product-prob-space-def product-prob-space-axioms-def product-sigma-finite-def*
by (*auto simp*: *prob-space-imp-sigma-finite*)
from *M* **interpret** *pair-sigma-finite M M* **by** *unfold-locales*
have [*measurable*]: {*h*∈*extensional* {*x*, *y*}. *h x* = *h y*} ∈ *sets* (*Pi*$_M$ {*x*, *y*} (λ*i*. *lborel*))
proof −
have {*h*∈*extensional* {*x*,*y*}. *h x* = *h y*} = {*h* ∈ *space* (*Pi*$_M$ {*x*, *y*} (λ*i*. *lborel*)). *h x* = *h y*}
by (*auto simp*: *extensional-def space-PiM*)
also have ... ∈ *sets* (*Pi*$_M$ {*x*, *y*} (λ*i*. *lborel*))
by *measurable*
finally show *?thesis* **.**
qed
have [*simp*]: *sets* (*PiM A* (λ-. *M*)) = *sets* (*PiM A* (λ-. *lborel*)) **for** *A* :: *'a set*
by (*intro sets-PiM-cong refl*) (*simp-all add*: *M-def*)
have *sets-M-M*: *sets* (*M* ⨂$_M$ *M*) = *sets* (*borel* ⨂$_M$ *borel*)
by (*intro sets-pair-measure-cong*) (*auto simp*: *M-def*)
have [*measurable*]: (λ(*b*, *a*). *if b* = *a then 1 else 0*) ∈ *borel-measurable* (*M* ⨂$_M$ *M*)
unfolding *measurable-split-conv*
by (*subst measurable-cong-sets*[*OF sets-M-M refl*])
(*auto intro*!: *measurable-If measurable-const measurable-equality-set*)

have {*h*∈*A* →$_E$ *UNIV*. *h x* = *h y*} =
(λ*h*. *restrict h* {*x*, *y*}) −‘ {*h*∈*extensional* {*x*, *y*}. *h x* = *h y*} ∩ *space* (*PiM*

A (λ-. *M* :: *real measure*))
by (*auto simp*: *space-PiM PiE-def extensional-def M-def*)
also have *emeasure* (*PiM A* (λ-. *M*)) ... =
emeasure (*distr* (*PiM A* (λ-. *M*)) (*PiM* {*x*,*y*} (λ-. *lborel* :: *real measure*))
(λ*h*. *restrict h* {*x*,*y*})) {*h*∈*extensional* {*x*, *y*}. *h x* = *h y*}
proof (*rule emeasure-distr* [*symmetric*])
have (λ*h*. *restrict h* {*x*, *y*}) ∈ *Pi*$_M$ *A* (λ-. *lborel*) →$_M$ *Pi*$_M$ {*x*, *y*} (λ-. *lborel*)
using *assms* **by** (*intro measurable-restrict-subset*) *auto*
also have ... = *Pi*$_M$ *A* (λ-. *M*) →$_M$ *Pi*$_M$ {*x*, *y*} (λ-. *lborel*)
by (*intro sets-PiM-cong measurable-cong-sets refl*) (*simp-all add*: *M-def*)
finally show (λ*h*. *restrict h* {*x*, *y*}) ∈ ... **.**
next
show {*h*∈*extensional* {*x*, *y*}. *h x* = *h y*} ∈ *sets* (*Pi*$_M$ {*x*, *y*} (λ-. *lborel*)) **by** *simp*
qed
also have *distr* (*PiM A* (λ-. *M*)) (*PiM* {*x*,*y*} (λ-. *lborel* :: *real measure*)) (λ*h*. *restrict h* {*x*,*y*}) =
distr (*PiM A* (λ-. *M*)) (*PiM* {*x*,*y*} (λ-. *M*)) (λ*h*. *restrict h* {*x*,*y*})
by (*intro distr-cong refl sets-PiM-cong*) (*simp-all add*: *M-def*)
also from *assms* **have** ... = *Pi*$_M$ {*x*, *y*} (λ*i*. *M*) **by** (*intro distr-restrict* [*symmetric*]) *auto*
also have *emeasure* ... {*h*∈*extensional* {*x*, *y*}. *h x* = *h y*} =
nn-integral ... (λ*h*. *indicator* {*h*∈*extensional* {*x*, *y*}. *h x* = *h y*} *h*)
by (*intro nn-integral-indicator* [*symmetric*]) *simp-all*
also have ... = *nn-integral* (*Pi*$_M$ {*x*, *y*} (λ*i*. *M*)) (λ*h*. *if h x* = *h y then 1 else 0*)
by (*intro nn-integral-cong*) (*auto simp add*: *indicator-def space-PiM PiE-def*)
also from ‹*x* ≠ *y*› **have** ... = (∫$^+$ *z*. (*if fst z* = *snd z then 1 else 0*) ∂(*M* ⨂$_M$ *M*))
by (*intro product-nn-integral-pair*) *auto*
also have ... = (∫$^+$ *x*. (∫$^+$*y*. (*if x* = *y then 1 else 0*) ∂*M*) ∂*M*)
by (*subst M*.*nn-integral-fst* [*symmetric*]) *simp-all*
also have ... = (∫$^+$ *x*. (∫$^+$*y*. *indicator* {*x*} *y* ∂*M*) ∂*M*)
by (*simp add*: *indicator-def of-bool-def eq-commute*)
also have ... = (∫$^+$ *x*. *emeasure M* {*x*} ∂*M*) **by** (*subst nn-integral-indicator*) (*simp-all add*: *M-def*)
also have ... = (∫$^+$ *x*. *0* ∂*M*) **unfolding** *M-def*
by (*intro nn-integral-cong-AE refl AE-uniform-measureI*) *auto*
also have ... = *0* **by** *simp*
finally show *?thesis* **.**
qed

lemma *measurable-linorder-from-keys-restrict*:
assumes *fin*: *finite A*
shows *linorder-from-keys A* ∈ *Pi*$_M$ *A* (λ-. *borel* :: *real measure*) →$_M$ *count-space* (*Pow* (*A* × *A*))
(**is** - : *?M* →$_M$ -)
apply (*subst measurable-count-space-eq2*)

apply (*auto simp add*: *fin linorder-from-keys-def*)
proof −
note *fin*[*simp*]
fix *R* **assume** $R \subseteq A \times A$
then have *linorder-from-keys A* −‘ {*R*} ∩ *space ?M* =
{*f* ∈ *space ?M*. ∀ *x*∈*A*. ∀ *y*∈*A*. (*x*, *y*) ∈ *R* ⟷ *f x* ≤ *f y*}
by (*auto simp add*: *linorder-from-keys-def set-eq-iff*)
also have ... ∈ *sets ?M*
by *measurable*
finally show *linorder-from-keys A* −‘ {*R*} ∩ *space ?M* ∈ *sets ?M* .
qed

lemma *measurable-count-space-extend*:
assumes *f* ∈ *measurable M* (*count-space A*) *A* ⊆ *B*
shows *f* ∈ *measurable M* (*count-space B*)
proof −
note *assms*(*1*)
also have *count-space A* = *restrict-space* (*count-space B*) *A*
using *assms*(*2*) **by** (*subst restrict-count-space*) (*simp-all add*: *Int-absorb2*)
finally show *?thesis* **by** (*simp add*: *measurable-restrict-space2-iff*)
qed

lemma *measurable-linorder-from-keys-restrict′*:
assumes *fin*: *finite A A* ⊆ *B*
shows *linorder-from-keys A* ∈ Pi_M *A* (λ-. *borel* :: *real measure*) →$_M$ *count-space* (*Pow* (*B* × *B*))
apply (*rule measurable-count-space-extend*)
apply (*rule measurable-linorder-from-keys-restrict*[*OF assms*(*1*)])
using *assms* **by** *auto*

context
fixes *a b* :: *real* **and** *A* :: ′*a set* **and** *M* **and** *B*
assumes *fin*: *finite A* **and** *ab*: *a* < *b* **and** *B*: *A* ⊆ *B*
defines *M* ≡ *distr* (*PiM A* (λ-. *uniform-measure lborel* {*a*..*b*}))
(*count-space* (*Pow* (*B* × *B*))) (*linorder-from-keys A*)
begin

lemma *measurable-linorder-from-keys* [*measurable*]:
linorder-from-keys A ∈ Pi_M *A* (λ-. *borel* :: *real measure*) →$_M$ *count-space* (*Pow* (*B* × *B*))
by (*rule measurable-linorder-from-keys-restrict′*) (*auto simp*: *fin B*)

The ordering defined by randomly-chosen priorities is almost surely linear:

theorem *almost-everywhere-linorder*: *AE R in M*. *linorder-on A R*
proof −
define *N* **where** *N* = *PiM A* (λ-. *uniform-measure lborel* {*a*..*b*})
have [*simp*]: *sets* (Pi_M *A* (λ-. *uniform-measure lborel* {*a*..*b*})) = *sets* (*PiM A* (λ-. *lborel*))

```
    by (intro sets-PiM-cong) simp-all
  let ?M-A = (Pi_M A (λ-. lborel :: real measure))
  have meas: {h ∈ A →_E UNIV. h i = h j} ∈ sets ?M-A
    if [simp]: i ∈ A j ∈ A for i j
  proof −
    have {h ∈ A →_E UNIV. h i = h j} = {h ∈ space ?M-A. h i = h j}
      by (auto simp: space-PiM)
    also have ... ∈ sets ?M-A
      by measurable
    finally show ?thesis .
  qed
  define X :: ('a ⇒ real) set where X = (⋃x∈A. ⋃y∈A−{x}. {h∈A →_E UNIV.
h x = h y})
  have AE f in N. inj-on f A
  proof (rule AE-I)
    show {f ∈ space N. ¬ inj-on f A} ⊆ X
      by (auto simp: inj-on-def X-def space-PiM N-def)
  next
    show X ∈ sets N unfolding X-def N-def
      using meas by (auto intro!: countable-finite fin sets.countable-UN')
  next
    have emeasure N X ≤ (∑ i∈A. emeasure N (⋃y∈A − {i}. {h ∈ A →_E UNIV.
h i = h y}))
      unfolding X-def N-def using fin meas
      by (intro emeasure-subadditive-finite)
        (auto simp: disjoint-family-on-def intro!: sets.countable-UN' countable-finite)
    also have ... ≤ (∑ i∈A. ∑ j∈A−{i}. emeasure N {h ∈ A →_E UNIV. h i =
h j})
      unfolding N-def using fin meas
      by (intro emeasure-subadditive-finite sum-mono)
        (auto simp: disjoint-family-on-def intro!: sets.countable-UN' countable-finite)
    also have ... = (∑ i∈A. ∑ j∈A−{i}. 0) unfolding N-def using fin ab
      by (intro sum.cong refl emeasure-PiM-diagonal) auto
    also have ... = 0 by simp
    finally show emeasure N X = 0 by simp
  qed
  hence AE f in N. linorder-on A (linorder-from-keys A f)
    by eventually-elim auto
  thus ?thesis unfolding M-def N-def
    by (subst AE-distr-iff) auto
qed
```

Furthermore, this is equivalent to choosing one of the $|A|!$ linear orderings uniformly at random.

```
theorem random-linorder-by-prios:
  M = uniform-measure (count-space (Pow (B × B))) (linorders-on A)
proof −
  from linorders-finite-nonempty[OF fin] obtain R where R: linorder-on A R
    by (auto simp: linorders-on-def)
```

have $*$: *emeasure M* $\{R\} \leq$ *emeasure M* $\{R'\}$ **if** *linorder-on A R linorder-on A R'* **for** *R R'*
proof −
define *N* **where** *N* = *PiM A* (λ-. *uniform-measure lborel* $\{a..b\}$)
from *linorder-permutation-exists*[*OF fin that*]
obtain π **where** π: π *permutes A R'* = *map-relation A* π *R*
by *blast*
have (λ*f*. *f* ∘ π) ∈ Pi_M *A* (λ-. *lborel* :: *real measure*) →$_M$ Pi_M *A* (λ-. *lborel* :: *real measure*)
by (*auto intro*!: *measurable-PiM-single' measurable-PiM-component-rev simp*: *comp-def permutes-in-image*[*OF* π(*1*)] *space-PiM PiE-def extensional-def*)
also have ... = *N* →$_M$ Pi_M *A* (λ-. *lborel*)
unfolding *N-def* **by** (*intro measurable-cong-sets refl sets-PiM-cong*) *simp-all*
finally have [*measurable*]: (λ*f*. *f* ∘ π) ∈ ... **.**

have [*simp*]: *measurable N X* = *measurable* (*PiM A* (λ-. *lborel*)) *X* **for** *X* :: (′*a* × ′*a*) *set measure*
unfolding *N-def* **by** (*intro measurable-cong-sets refl sets-PiM-cong*) *simp-all*
have [*simp*]: *measurable M X* = *measurable* (*count-space* (*Pow* (*B* × *B*))) *X* **for** *X* :: (′*a* × ′*a*) *set measure*
unfolding *M-def* **by** *simp*

have *M-eq*: *M* = *distr N* (*count-space* (*Pow* (*B* × *B*))) (*linorder-from-keys A*)
by (*simp only*: *M-def N-def*)
also have *N* = *distr N* (*PiM A* (λ-. *lborel*)) (λ*f*. *f* ∘ π)
unfolding *N-def* **by** (*rule PiM-uniform-measure-permute* [*symmetric*]) *fact*+
also have *distr* ... (*count-space* (*Pow* (*B* × *B*))) (*linorder-from-keys A*) = *distr N* (*count-space* (*Pow* (*B* × *B*))) (*linorder-from-keys A* ∘ (λ*f*. *f* ∘ π))
by (*intro distr-distr*) *simp-all*
also have ... = *distr N* (*count-space* (*Pow* (*B* × *B*))) (*map-relation A* π ∘ *linorder-from-keys A*)
by (*intro distr-cong refl*) (*auto simp*: *linorder-from-keys-permute*[*OF* π(*1*)])
also have ... = *distr M* (*count-space* (*Pow* (*B* × *B*))) (*map-relation A* π)
unfolding *M-eq* **using** *B*
by (*intro distr-distr* [*symmetric*]) (*auto simp*: *map-relation-def*)
finally have *M-eq'*: *distr M* (*count-space* (*Pow* (*B* × *B*))) (*map-relation A* π) = *M* **..**

from *that* **have** *subset'*: *R* ⊆ *B* × *B R'* ⊆ *B* × *B*
using *B* **by** (*auto simp*: *linorder-on-def refl-on-def*)
hence *emeasure M* $\{R\} \leq$ *emeasure M* (*map-relation A* π −‘ $\{R'\}$ ∩ *space M*)
using *subset'* **by** (*intro emeasure-mono*) (*auto simp*: *M-def* π)
also have ... = *emeasure* (*distr M* (*count-space* (*Pow* (*B* × *B*))) (*map-relation A* π)) $\{R'\}$
by (*rule emeasure-distr* [*symmetric*]) (*insert subset' B*, *auto simp*: *map-relation-def*)
also note *M-eq'*

finally show *?thesis* **.**
qed
have *same-prob*: *emeasure M* $\{R'\}$ = *emeasure M* $\{R\}$ **if** *linorder-on A R'* **for** *R'*
using $*[of\ R\ R']$ **and** $*[of\ R'\ R]$ **and** *R* **and** *that* **by** *simp*

from *ab* **have** *prob-space M*
unfolding *M-def*
by (*intro prob-space.prob-space-distr prob-space-PiM prob-space-uniform-measure*) *simp-all*
hence *1* = *emeasure M* (*Pow* (*B* × *B*))
using *prob-space.emeasure-space-1*[*OF* ‹*prob-space M*›] **by** (*simp add*: *M-def*)
also have (*Pow* (*B* × *B*)) = *linorders-on A* ∪ ((*Pow* (*B* × *B*))−*linorders-on A*)
using *B* **by** (*auto simp*: *linorders-on-def linorder-on-def refl-on-def*)
also have *emeasure M* ... = *emeasure M* (*linorders-on A*) + *emeasure M* (*Pow* (*B* × *B*)−*linorders-on A*)
using *B* **by** (*subst plus-emeasure*) (*auto simp*: *M-def linorders-on-def linorder-on-def refl-on-def*)
also have *emeasure M* (*Pow* (*B* × *B*)−*linorders-on A*) = *0* **using** *almost-everywhere-linorder*

by (*subst* (*asm*) *AE-iff-measurable*) (*auto simp*: *linorders-on-def M-def*)
also from *fin* **have** *emeasure M* (*linorders-on A*) = (∑ *R'*∈*linorders-on A*. *emeasure M* $\{R'\}$)
using *B* **by** (*intro emeasure-eq-sum-singleton*)
(*auto simp*: *M-def linorders-on-def linorder-on-def refl-on-def*)
also have ... = (∑ *R'*∈*linorders-on A*. *emeasure M* $\{R\}$)
by (*rule sum.cong*) (*simp-all add*: *linorders-on-def same-prob*)
also from *fin* **have** ... = *fact* (*card A*) ∗ *emeasure M* $\{R\}$
by (*simp add*: *linorders-on-def card-finite-linorders*)
finally have [*simp*]: *emeasure M* $\{R\}$ = *inverse* (*fact* (*card A*))
by (*simp add*: *inverse-ennreal-unique*)

show *?thesis*
proof (*rule measure-eqI-countable-AE'*)
show *sets M* = *Pow* (*Pow* (*B* × *B*))
by (*simp add*: *M-def*)
next
from ‹*finite A*› **show** *countable* (*linorders-on A*)
by (*blast intro*: *countable-finite*)
next
show *AE R in uniform-measure* (*count-space* (*Pow* (*B* × *B*))) (*linorders-on A*). *R* ∈ *linorders-on A*
by (*rule AE-uniform-measureI*)
(*insert B*, *auto simp*: *linorders-on-def linorder-on-def refl-on-def*)
next
fix *R'* **assume** *R'*: *R'* ∈ *linorders-on A*
have *subset*: *linorders-on A* ⊆ *Pow* (*B* × *B*)
using *B* **by** (*auto simp*: *linorders-on-def linorder-on-def refl-on-def*)
have *emeasure* (*uniform-measure* (*count-space* (*Pow* (*B* × *B*)))

(*linorders-on A*)) {*R'*} = *emeasure* (*count-space* (*Pow* (*B* × *B*))) (*linorders-on A* ∩ {*R'*}) /
emeasure (*count-space* (*Pow* (*B* × *B*))) (*linorders-on A*)
using *R' B* **by** (*subst emeasure-uniform-measure*) (*auto simp*: *linorders-on-def linorder-on-def refl-on-def*)
also have ... = *1* / *emeasure* (*count-space* (*Pow* (*B* × *B*))) (*linorders-on A*)
using *R' B* **by** (*subst emeasure-count-space*) (*auto simp*: *linorders-on-def linorder-on-def refl-on-def*)
also have ... = *1* / *fact* (*card A*)
using *fin finite-linorders-on*[*of A*]
by (*subst emeasure-count-space* [*OF subset*])
(*auto simp*: *divide-ennreal* [*symmetric*] *linorders-on-def card-finite-linorders*)
also have ... = *emeasure M* {*R*}
by (*simp add*: *field-simps divide-ennreal-def*)
also have ... = *emeasure M* {*R'*}
using *R'* **by** (*intro same-prob* [*symmetric*]) (*auto simp*: *linorders-on-def*)
finally show *emeasure M* {*R'*} = *emeasure* (*uniform-measure* (*count-space* (*Pow* (*B* × *B*)))
(*linorders-on A*)) {*R'*} **..**
next
show *linorders-on A* ⊆ *Pow* (*B* × *B*)
using *B* **by** (*auto simp*: *linorders-on-def linorder-on-def refl-on-def*)
qed (*auto simp*: *M-def linorders-on-def almost-everywhere-linorder*)
qed

end
end

4 Relationship between treaps and BSTs

theory *Treap-Sort-and-BSTs*
imports
Treap
Random-List-Permutation
Random-BSTs.Random-BSTs
begin

Here, we will show that if we "forget" the priorities of a treap, we essentially get a BST into which the elements have been inserted by ascending priority.

First, we show some facts about sorting that we will need.

The following two lemmas are only important for measurability later.

lemma *insort-key-conv-rec-list*:
insort-key f x xs =
rec-list [*x*] (λ*y ys zs*. *if f x* ≤ *f y then x* # *y* # *ys else y* # *zs*) *xs*
by (*induction xs*) *simp-all*

lemma *insort-key-conv-rec-list'*:
insort-key = (λ*f x*.
rec-list [*x*] (λ*y ys zs*. *if f x* ≤ *f y then x* # *y* # *ys else y* # *zs*))
by (*intro ext*) (*simp add*: *insort-key-conv-rec-list*)

lemma *bst-of-list-trees*:
assumes *set ys* ⊆ *A*
shows *bst-of-list ys* ∈ *trees A*
using *assms* **by** (*induction ys rule*: *bst-of-list.induct*) *auto*

lemma *insort-wrt-insort-key*:
a ∈ *A* ⟹
set xs ⊆ *A* ⟹
insert-wrt (*linorder-from-keys A f*) *a xs* = *insort-key f a xs*
unfolding *linorder-from-keys-def* **by** (*induction xs*) (*auto*)

lemma *insort-wrt-sort-key*:
assumes *set xs* ⊆ *A*
shows *insort-wrt* (*linorder-from-keys A f*) *xs* = *sort-key f xs*
using *assms* **by** (*induction xs*) (*auto simp add*: *insort-wrt-def insort-wrt-insort-key*)

The following is an important recurrence for *sort-key* that states that for distinct priorities, sorting a list w. r. t. those priorities can be seen as selection sort, i. e. we can first choose the (unique) element with minimum priority as the first element and then sort the rest of the list and append it.

lemma *sort-key-arg-min-on*:
assumes *zs* ≠ [] *inj-on p* (*set zs*)
shows *sort-key p* (*zs*::'*a*::*linorder list*) =
(*let z* = *arg-min-on p* (*set zs*) *in z* # *sort-key p* (*remove1 z zs*))
proof −
have *mset zs* = *mset* (*let z* = *arg-min-on p* (*set zs*) *in z* # *sort-key p* (*remove1 z zs*))
proof −
define *m* **where** *m* = *arg-min-on p* (*set zs*)
have *m* ∈ (*set zs*)
unfolding *m-def* **by** (*rule arg-min-if-finite*) (*use assms* **in** *auto*)
then show *?thesis*
by (*auto simp add*: *Let-def m-def*)
qed
moreover have *linorder-class.sorted*
(*map p* (*let z* = *arg-min-on p* (*set zs*) *in z* # *sort-key p* (*remove1 z zs*)))
proof −
have *set* (*map p* (*sort-key p* (*remove1* (*arg-min-on p* (*set zs*)) *zs*))) ⊆ *p* ‘ *set zs*
using *set-remove1-subset* **by** (*fastforce*)
moreover have ⋀*y*. *y* ∈ *p* ‘ *set zs* ⟹ *p* (*arg-min-on p* (*set zs*)) ≤ *y*
using *arg-min-least assms* **by** *force*
ultimately have *linorder-class.sorted*
(*p* (*arg-min-on p* (*set zs*)) # *map p* (*sort-key p* (*remove1* (*arg-min-on p* (*set*

```
zs)) zs)))
      by (auto)
    then show ?thesis
      by (simp add: Let-def)
  qed
  ultimately show ?thesis
    using sort-key-inj-key-eq assms by blast
qed

lemma arg-min-on-image-finite:
  fixes f :: 'b ⇒ 'c :: linorder
  assumes inj-on f (g ` B) finite B B ≠ {}
  shows arg-min-on f (g ` B) = g (arg-min-on (f ∘ g) B)
 by (smt (verit, best) antisym-conv3 arg-min-if-finite(1,2) assms(1,2,3) finite-imageI
image-iff image-is-empty o-apply
      the-inv-into-f-f)

lemma fst-snd-arg-min-on: fixes p::'a ⇒ 'b::linorder
  assumes finite B inj-on p B B ≠ {}
  shows   fst (arg-min-on snd ((λx. (x, p x)) ` B)) = arg-min-on p B
  by (subst arg-min-on-image-finite [OF inj-on-imageI])
     (auto simp: o-def assms)
```

The following is now the main result:

```
theorem treap-of-bst-of-list':
  assumes ys = map (λx. (x, p x)) xs inj-on p (set xs) xs' = sort-key p xs
  shows map-tree fst (treap-of (set ys)) = bst-of-list xs'
  using assms
proof(induction xs' arbitrary: xs ys rule: bst-of-list.induct)
  case 1
  from ‹[] = sort-key p xs›[symmetric] ‹ys = map (λx. (x, p x)) xs›
  have ys = []
    by (cases xs) (auto)
  then show ?case by (simp add: treap-of.simps)
next
  case (2 z zs)
  note IH = 2(1,2)
  note assms = 2(3,4,5)
  define m where m = arg-min-on snd (set ys)
  define ls where ls = map (λx. (x, p x)) [y←zs . y < z]
  define rs where rs = map (λx. (x, p x)) [y←zs . y > z]
  define L where L = {p ∈ (set ys). fst p < fst m}
  define R where R = {p ∈ (set ys). fst p > fst m}
  have h1: set (z#zs) = set xs
    using assms by simp
  then have h2: inj-on p {x ∈ set zs. x < z} inj-on p (set (filter ((<) z) zs))
    inj-on p (set zs)
    using ‹inj-on p (set xs)› by (auto intro!: inj-on-subset[of - set xs])
  have z # zs = (let z = arg-min-on p (set xs) in z # sort-key p (remove1 z xs))
```

```
  proof −
    have xs ≠ []
      using assms by force
    then show ?thesis
      by (auto simp add: assms intro!: sort-key-arg-min-on)
  qed
  then have h3: z = arg-min-on p (set xs) zs = sort-key p (remove1 z xs)
    unfolding Let-def by auto
  have h4: sort-key p zs = zs
  proof −
    have linorder-class.sorted (map p (z#zs))
      using assms by simp
    then have linorder-class.sorted (map p zs)
      by auto
    then show ?thesis
      using h1 h2 sort-key-inj-key-eq by blast
  qed
  note helpers = h1 h2 h3 h4
  have fst m = z
  proof −
    have fst m = arg-min-on p (set xs)
      unfolding m-def using assms by (auto intro!: fst-snd-arg-min-on)
    also have ... = z
      using helpers by auto
    finally show ?thesis .
  qed
  moreover have map-tree fst (treap-of L) = bst-of-list [y←zs . y < z]
  proof −
    have L = set ls
      unfolding L-def ls-def ‹fst m = z› using helpers assms by force
    moreover have map-tree fst (treap-of (set ls)) = bst-of-list [y←zs . y < z]
      unfolding ls-def using helpers
      by (intro IH(1)[of - [y←zs . y < z]]) (auto simp add: filter-sort[symmetric])
    ultimately show ?thesis
      by blast
  qed
  moreover have map-tree fst (treap-of R) = bst-of-list [y←zs . z < y]
  proof −
    have 0: R = set rs
      unfolding R-def rs-def ‹fst m = z› using helpers assms by force
    moreover have map-tree fst (treap-of (set rs)) = bst-of-list [y←zs . z < y]
      unfolding rs-def using helpers
      by (intro IH(2)[of - [y←zs . z < y]]) (auto simp add: filter-sort[symmetric])
    ultimately show ?thesis
      by blast
  qed
  moreover have treap-of (set ys) = ⟨treap-of L, m, treap-of R⟩
    unfolding L-def m-def R-def using assms by (auto simp add: treap-of.simps
Let-def)
```

ultimately show *?case* **by** *auto*
qed

corollary *treap-of-bst-of-list*: *inj-on p* (*set zs*) ⟹
map-tree fst (*treap-of* (*set* (*map* (λ*x*. (*x*, *p x*)) *zs*))) = *bst-of-list* (*sort-key p zs*)
using *treap-of-bst-of-list′* **by** *blast*

corollary *treap-of-bst-of-list″*: *inj-on p* (*set zs*) ⟹
map-tree fst (*treap-of* ((λ*x*. (*x*, *p x*)) ‘ *set zs*)) = *bst-of-list* (*sort-key p zs*)
using *treap-of-bst-of-list* **by** *auto*

corollary *fold-ins-bst-of-list*: *distinct zs* ⟹ *inj-on p* (*set zs*) ⟹
map-tree fst (*foldl* (λ*t* (*x*,*p*). *ins x p t*) ⟨⟩ (*map* (λ*x*. (*x*, *p x*)) *zs*)) = *bst-of-list* (*sort-key p zs*)
by (*auto simp add*: *foldl-ins-treap-of distinct-map inj-on-def inj-on-convol-ident treap-of-bst-of-list″*)

end

5 Random treaps

theory *Random-Treap*
imports
Probability-Misc
Treap-Sort-and-BSTs
begin

5.1 Measurability

The following lemmas are only relevant for measurability.

lemma *tree-sigma-cong*:
assumes *sets M* = *sets M′*
shows *tree-sigma M* = *tree-sigma M′*
using *sets-eq-imp-space-eq*[*OF assms*] **using** *assms* **by** (*simp add*: *tree-sigma-def*)

lemma *distr-restrict*:
assumes *sets N* = *sets L sets K* ⊆ *sets M*
⋀*X*. *X* ∈ *sets K* ⟹ *emeasure M X* = *emeasure K X*
⋀*X*. *X* ∈ *sets M* ⟹ *X* ⊆ *space M* − *space K* ⟹ *emeasure M X* = *0*
f ∈ *M* →$_M$ *N f* ∈ *K* →$_M$ *L*
shows *distr M N f* = *distr K L f*
proof (*rule measure-eqI*)
fix *X* **assume** *X* ∈ *sets* (*distr M N f*)
thus *emeasure* (*distr M N f*) *X* = *emeasure* (*distr K L f*) *X*
using *assms*(*1*) **by** (*intro emeasure-distr-restrict assms*) *simp-all*
qed (*use assms* **in** *auto*)

lemma *sets-tree-sigma-count-space*:
assumes *countable B*
shows *sets* (*tree-sigma* (*count-space B*)) = *Pow* (*trees B*)
proof (*intro equalityI subsetI*)
fix *X* **assume** *X*: *X* ∈ *Pow* (*trees B*)
have {*t*} ∈ *sets* (*tree-sigma* (*count-space B*)) **if** *t* ∈ *trees B* **for** *t*
using *that*
proof (*induction t*)
case (*2 l r x*)
hence {⟨*la*, *v*, *ra*⟩ |*la v ra*. (*v*, *la*, *ra*) ∈ {*x*} × {*l*} × {*r*}}
∈ *sets* (*tree-sigma* (*count-space B*))
by (*intro Node-in-tree-sigma pair-measureI*) *auto*
thus *?case* **by** *simp*
qed *simp-all*
with *X* **have** (⋃ *t*∈*X*. {*t*}) ∈ *sets* (*tree-sigma* (*count-space B*))
by (*intro sets.countable-UN′ countable-subset*[*OF - countable-trees*[*OF assms*]])
auto
also have (⋃ *t*∈*X*. {*t*}) = *X* **by** *blast*
finally show *X* ∈ *sets* (*tree-sigma* (*count-space B*)) **.**
next
fix *X* **assume** *X* ∈ *sets* (*tree-sigma* (*count-space B*))
from *sets.sets-into-space*[*OF this*] **show** *X* ∈ *Pow* (*trees B*)
by (*simp add*: *space-tree-sigma*)
qed

lemma *height-primrec*: *height* = *rec-tree 0* (λ- - - *a b*. *Suc* (*max a b*))
proof
fix *t* :: *′a tree*
show *height t* = *rec-tree 0* (λ- - - *a b*. *Suc* (*max a b*)) *t*
by (*induction t*) *simp-all*
qed

lemma *ipl-primrec*: *ipl* = *rec-tree 0* (λ*l* - *r a b*. *size l* + *size r* + *a* + *b*)
proof
fix *t* :: *′a tree*
show *ipl t* = *rec-tree 0* (λ*l* - *r a b*. *size l* + *size r* + *a* + *b*) *t*
by (*induction t*) *auto*
qed

lemma *size-primrec*: *size* = *rec-tree 0* (λ- - - *a b*. *1* + *a* + *b*)
proof
fix *t* :: *′a tree*
show *size t* = *rec-tree 0* (λ- - - *a b*. *1* + *a* + *b*) *t*
by (*induction t*) *auto*
qed

lemma *ipl-map-tree*[*simp*]: *ipl* (*map-tree f t*) = *ipl t*
by (*induction t*) *auto*

lemma *set-pmf-random-bst*: *finite A* $\Longrightarrow$ *set-pmf* (*random-bst A*) $\subseteq$ *trees A*
by (*subst random-bst-altdef*)
(*auto intro*!: *bst-of-list-trees simp add*: *bst-of-list-trees permutations-of-setD*)

lemma *trees-mono*: $A \subseteq B \Longrightarrow$ *trees A* $\subseteq$ *trees B*
proof
fix *t*
assume $A \subseteq B$ $t \in$ *trees A*
then show $t \in$ *trees B*
by (*induction t*) *auto*
qed

lemma *ins-primrec*:
ins k (*p*::*real*) *t* = *rec-tree*
(*Node Leaf* (*k*,*p*) *Leaf*)
(λ*l z r l*′ *r*′. *case z of* (*k1*, *p1*) ⇒
if k < *k1 then*
(*case l*′ *of*
Leaf ⇒ *Leaf*
| *Node l2* (*k2*,*p2*) *r2* ⇒
if 0 ≤ *p2* − *p1 then Node* (*Node l2* (*k2*,*p2*) *r2*) (*k1*,*p1*) *r*
else Node l2 (*k2*,*p2*) (*Node r2* (*k1*,*p1*) *r*))
else if k > *k1 then*
(*case r*′ *of*
Leaf ⇒ *Leaf*
| *Node l2* (*k2*,*p2*) *r2* ⇒
if 0 ≤ *p2* − *p1 then Node l* (*k1*,*p1*) (*Node l2* (*k2*,*p2*) *r2*)
else Node (*Node l* (*k1*,*p1*) *l2*) (*k2*,*p2*) *r2*)
else Node l (*k1*,*p1*) *r*
) *t*
proof (*induction k p t rule*: *ins.induct*)
case (*2 k p l k1 p1 r*)
thus *?case*
by (*cases k* < *k1*) (*auto simp add*: *case-prod-beta ins-neq-Leaf split*: *tree.splits if-splits*)
qed *auto*

lemma *measurable-less-count-space* [*measurable* (*raw*)]:
assumes *countable A*
assumes [*measurable*]: *a* ∈ *B* →$_M$ *count-space A*
assumes [*measurable*]: *b* ∈ *B* →$_M$ *count-space A*
shows *Measurable.pred B* (λ*x*. *a x* < *b x*)
proof −
have *Measurable.pred* (*count-space* (*A* × *A*)) (λ*x*. *fst x* < *snd x*) **by** *simp*
also have *count-space* (*A* × *A*) = *count-space A* ⨂$_M$ *count-space A*
using *assms*(*1*) **by** (*simp add*: *pair-measure-countable*)
finally have *Measurable.pred B* ((λ*x*. *fst x* < *snd x*) ∘ (λ*x*. (*a x*, *b x*)))
by *measurable*
thus *?thesis* **by** (*simp add*: *o-def*)

qed

lemma *measurable-ins* [*measurable* (*raw*)]:
assumes [*measurable*]: *countable A*
assumes [*measurable*]: $k \in B \to_M$ *count-space A*
assumes [*measurable*]: $x \in B \to_M$ (*lborel* :: *real measure*)
assumes [*measurable*]: $t \in B \to_M$ *tree-sigma* (*count-space A* $\bigotimes_M$ *lborel*)
shows $(\lambda y.\ ins\ (k\ y)\ (x\ y)\ (t\ y)) \in B \to_M$ *tree-sigma* (*count-space A* $\bigotimes_M$ *lborel*)
unfolding *ins-primrec* **by** *measurable*

lemma *map-tree-primrec*: *map-tree f t* = *rec-tree* ⟨⟩ $(\lambda l\ a\ r\ l'\ r'.\ \langle l', f\ a, r'\rangle)\ t$
by (*induction t*) *auto*

definition $\mathcal{U}$ **where** $\mathcal{U} = (\lambda a\ b{::}real.\ uniform\text{-}measure\ lborel\ \{a..b\})$

declare $\mathcal{U}$*-def*[*simp*]

fun *insR*:: *'a*::*linorder* ⇒ (*'a* × *real*) *tree* ⇒ *'a set* ⇒ (*'a* × *real*) *tree measure*
where
insR x t A = *distr* ($\mathcal{U}$ *0 1*) (*tree-sigma* (*count-space A* $\bigotimes_M$ *lborel*)) (λ*p*. *ins x p t*)

fun *rinss* :: *'a*::*linorder list* ⇒ (*'a* × *real*) *tree* ⇒ *'a set* ⇒ (*'a* × *real*) *tree measure*
where
rinss [] *t A* = *return* (*tree-sigma* (*count-space A* $\bigotimes_M$ *lborel*)) *t* |
rinss (*x*#*xs*) *t A* = *insR x t A* ⤜ (λ*t*. *rinss xs t A*)

lemma *sets-rinss'*:
assumes *countable B set ys* ⊆ *B*
shows *t* ∈ *trees* (*B* × *UNIV*) ⟹ *sets* (*rinss ys t B*) = *sets* (*tree-sigma* (*count-space B* $\bigotimes_M$ *lborel*))
using *assms* **proof**(*induction ys arbitrary*: *t*)
case (*Cons y ys*)
then show *?case*
by (*subst rinss.simps*, *subst sets-bind*) (*auto simp add*: *space-tree-sigma space-pair-measure*)
qed *auto*

lemma *measurable-foldl* [*measurable*]:
assumes *f* ∈ *A* →$_M$ *B set xs* ⊆ *space C*
assumes ⋀*c*. *c* ∈ *set xs* ⟹ (λ(*a*,*b*). *g a b c*) ∈ (*A* $\bigotimes_M$ *B*) →$_M$ *B*
shows (λ*x*. *foldl* (*g x*) (*f x*) *xs*) ∈ *A* →$_M$ *B*
using *assms*
proof (*induction xs arbitrary*: *f*)
case *Nil*
thus *?case* **by** *simp*
next
case (*Cons x xs*)
note [*measurable*] = *Cons.prems*(*1*)

from *Cons.prems* **have** [*measurable*]: $x \in space\ C$ **by** *simp*
have $(\lambda a.\ (a,\ f\ a)) \in A \rightarrow_M A \bigotimes_M B$
by *measurable*
hence $(\lambda(a,b).\ g\ a\ b\ x) \circ (\lambda a.\ (a,\ f\ a)) \in A \rightarrow_M B$
by (*rule measurable-comp*) (*rule Cons.prems*, *auto*)
hence $(\lambda a.\ g\ a\ (f\ a)\ x) \in A \rightarrow_M B$ **by** (*simp add*: *o-def*)
hence $(\lambda xa.\ foldl\ (g\ xa)\ (g\ xa\ (f\ xa)\ x)\ xs) \in A \rightarrow_M B$
by (*rule Cons.IH*) (*use Cons.prems* **in** *auto*)
thus *?case* **by** *simp*
qed

lemma *ins-trees*: $t \in trees\ A \Longrightarrow (x,y) \in A \Longrightarrow ins\ x\ y\ t \in trees\ A$
by (*induction x y t rule*: *ins.induct*)
(*auto split*: *tree.splits simp*: *ins-neq-Leaf*)

5.2 Main result

In our setting, we have some countable set of values that may appear in the input and a concrete list consisting only of those elements with no repeated elements.

We further define an abbreviation for the uniform distribution of permutations of that lists.

context
fixes *xs*::*'a*::*linorder list* **and** *A*::*'a set* **and** *random-perm* :: *'a list* ⇒ *'a list measure*
assumes *con-assms*: *countable A set xs* ⊆ *A distinct xs*
defines *random-perm* ≡ (λ*xs*. *uniform-measure* (*count-space* (*permutations-of-set* (*set xs*)))
(*permutations-of-set* (*set xs*)))
begin

Again, we first need some facts about measurability.

lemma *sets-rinss* [*simp*]:
assumes $t \in trees\ (A \times UNIV)$
shows $sets\ (rinss\ xs\ t\ A) = tree\text{-}sigma\ (count\text{-}space\ A \bigotimes_M borel)$
proof −
have $tree\text{-}sigma\ (count\text{-}space\ A \bigotimes_M (lborel{::}real\ measure)) = tree\text{-}sigma\ (count\text{-}space\ A \bigotimes_M borel)$
by (*intro tree-sigma-cong sets-pair-measure-cong*) *auto*
then show *?thesis*
using *assms con-assms* **by** (*subst sets-rinss'*) *auto*
qed

lemma *bst-of-list-measurable* [*measurable*]:
$bst\text{-}of\text{-}list \in measurable\ (count\text{-}space\ (lists\ A))\ (tree\text{-}sigma\ (count\text{-}space\ A))$
by (*subst measurable-count-space-eq1*)
(*auto simp*: *space-tree-sigma intro*!: *bst-of-list-trees*)

lemma *insort-wrt-measurable* [*measurable*]:
(λ*x*. *insort-wrt x xs*) ∈ *count-space* (*Pow* (*A* × *A*)) →$_M$ *count-space* (*lists A*)
using *con-assms* **by** *auto*

lemma *bst-of-list-sort-meaurable* [*measurable*]:
(λ*x*. *bst-of-list* (*sort-key x xs*)) ∈
Pi$_M$ (*set xs*) (λ*i*. *borel*::*real measure*) →$_M$ *tree-sigma* (*count-space A*)
proof −
note *measurable-linorder-from-keys-restrict'*[*measurable*]
have (*0*::*real*) < *1*
by *auto*
then have [*measurable*]: (λ*x*. *bst-of-list* (*insort-wrt* (*linorder-from-keys* (*set xs*) *x*) *xs*))
∈ *Pi*$_M$ (*set xs*) (λ*i*. *borel* :: *real measure*) →$_M$ *tree-sigma* (*count-space A*)
using *con-assms* **by** *measurable*
show *?thesis*
by (*subst insort-wrt-sort-key*[*symmetric*]) (*measurable*, *auto*)
qed

In a first step, we convert the bulk insertion operation to first choosing the priorities i. i. d. ahead of time and then inserting all the elements deterministically with their associated priority.

lemma *random-treap-fold*:
assumes *t* ∈ *space* (*tree-sigma* (*count-space A* ⨂$_M$ *lborel*))
shows *rinss xs t A* = *distr* (Π$_M$ *x*∈*set xs*. *𝒰 0 1*)
(*tree-sigma* (*count-space A* ⨂$_M$ *lborel*))
(λ*p*. *foldl* (λ*t x*. *ins x* (*p x*) *t*) *t xs*)
proof −
let *?U* = *uniform-measure lborel* {*0*::*real*..*1*}
have *set xs* ⊆ *space* (*count-space A*) *c* ∈ *set xs* ⟹ *c* ∈ *space* (*count-space A*)
for *c*
using *con-assms* **by** *auto*
then have ∗[*intro*]: (λ*p*. *foldl* (λ*t x*. *ins x* (*p x*) *t*) *t xs*) ∈
Pi$_M$ (*set xs*) (λ*x*. *?U*) →$_M$ *tree-sigma* (*count-space A* ⨂$_M$ *lborel*)
if *t* ∈ *space* (*tree-sigma* (*count-space A* ⨂$_M$ *lborel*)) **for** *t*
using *that con-assms* **by** *measurable*
have *insR'*:
insR x t A = *?U* ⤜ (λ*u*. *return* (*tree-sigma* (*count-space A* ⨂$_M$ *lborel*)) (*ins x u t*))
if *x* ∈ *A t* ∈ *space* (*tree-sigma* (*count-space A* ⨂$_M$ *lborel*)) **for** *t x*
using *con-assms assms that* **by** (*auto simp add*: *bind-return-distr' 𝒰-def*)
have *rinss xs t A* = (Π$_M$ *x*∈*set xs*. *?U*) ⤜
(λ*p*. *return* (*tree-sigma* (*count-space A* ⨂$_M$ *lborel*)) (*foldl* (λ*t x*. *ins x* (*p x*) *t*) *t xs*))
using *con-assms*(*2*,*3*) *assms* **proof** (*induction xs arbitrary*: *t*)
case *Nil*
then show *?case*
by (*intro measure-eqI*) (*auto simp add*: *space-PiM-empty emeasure-distr bind-return-distr'*)

next
case (*Cons x xs*)
note *insR.simps*[*simp del*]
let *?treap-sigma* = *tree-sigma* (*count-space A* $\bigotimes_M$ *lborel*)
have [*measurable*]: *set xs* $\subseteq$ *space* (*count-space A*) *x* $\in$ *A*
$c \in A \Longrightarrow c \in$ *space* (*count-space A*) **for** *c*
using *Cons* **by** *auto*
have [*intro*!]: *ins k p t* $\in$ *space ?treap-sigma* **if** *t* $\in$ *space ?treap-sigma k* $\in$ *A*
for *k t* **and** *p*::*real*
using *that*
by (*auto intro*!: *ins-trees simp add*: *space-tree-sigma space-pair-measure*)
have [*measurable*]: Pi_M (*set xs*) (*λx. ?U*) $\in$ *space* (*prob-algebra* (Pi_M (*set xs*)
(*λi. ?U*)))
unfolding *space-prob-algebra* **by** (*auto intro*!: *prob-space-uniform-measure prob-space-PiM*)
have [*measurable*]: Pi_M (*set xs*) (*λx. ?U*) $\in$ *space* (*subprob-algebra* (Pi_M (*set xs*)
(*λi. ?U*)))
unfolding *space-subprob-algebra*
by (*auto intro*!: *prob-space-imp-subprob-space prob-space-uniform-measure prob-space-PiM*)
have [*measurable*]: (*λx. x*) $\in$ (*?treap-sigma* $\bigotimes_M$ Pi_M (*set xs*) (*λi. ?U*)) $\bigotimes_M$
?treap-sigma $\rightarrow_M$
(*?treap-sigma* $\bigotimes_M$ Pi_M (*set xs*) (*λi. borel*)) $\bigotimes_M$ *?treap-sigma*
by (*auto intro*!: *measurable-ident-sets sets-pair-measure-cong sets-PiM-cong simp
add*: $\mathcal{U}$*-def*)
have [*simp*]: (*λw.* Pi_M (*set xs*) (*λx. ?U*) $\gg\!=$
(*λp. return ?treap-sigma* (*foldl* (*λt x. ins x* (*p x*) *t*) *w xs*)))
$\in$ *?treap-sigma* $\rightarrow_M$ *subprob-algebra ?treap-sigma*
proof −
have [*measurable*]: *c* $\in$ *set xs* $\Longrightarrow$ *c* $\in$ *A* **for** *c*
using *Cons* **by** *auto*
show *?thesis*
using *con-assms* **by** *measurable*
qed
have [*measurable*]: *?U* $\in$ *space* (*prob-algebra* (*?U*))
by (*simp add*: *prob-space-uniform-measure space-prob-algebra*)
have [*measurable*, *intro*]: (*λt. rinss xs t A*) $\in$ *?treap-sigma* $\rightarrow_M$ *subprob-algebra
?treap-sigma*
if *set xs* $\subseteq$ *A* **for** *xs*
using *that* **proof** (*induction xs*)
case (*Cons x xs*)
then have [*measurable*]: *x* $\in$ *A set xs* $\subseteq$ *A*
by *auto*
have [*measurable*]: (*λy. x*) $\in$ *tree-sigma* (*count-space A* $\bigotimes_M$ *lborel*) $\bigotimes_M$ *?U*
$\rightarrow_M$ *count-space A*
using *Cons* **by** *measurable*
have [*measurable*]: (*λx. x*) $\in$ *?treap-sigma* $\bigotimes_M$ *?U* $\rightarrow_M$ *?treap-sigma* $\bigotimes_M$
borel
unfolding $\mathcal{U}$*-def* **by** *auto*
have [*measurable*]: (*λt. distr* (*?U*) (*tree-sigma* (*count-space A* $\bigotimes_M$ *lborel*)) (*λp.
ins x p t*))

$\in$ *?treap-sigma* $\rightarrow_M$ *subprob-algebra ?treap-sigma*
using *con-assms* **by** (*intro measurable-prob-algebraD*) *measurable*
from *Cons* **show** *?case*
unfolding *rinss.simps insR.simps* $\mathcal{U}$*-def* **by** *measurable*
qed *auto*
have [*intro*]: (*λu. return ?treap-sigma* (*ins x u t*)) $\in$ *?U* $\rightarrow_M$ *subprob-algebra ?treap-sigma*
using *con-assms Cons* **by** *measurable*
have [*simp*]: *space* (*?U* $\bigotimes_M$ Pi_M (*set xs*) (*λx. ?U*)) $\neq$ {}
by (*simp add*: *prob-space.not-empty prob-space-PiM prob-space-pair prob-space-uniform-measure*)
from *Cons* **have** *rinss* (*x # xs*) *t A* = (*?U* $\gg\!=$
(*λu. return ?treap-sigma* (*ins x u t*))) $\gg\!=$
(*λt. rinss xs t A*)
by (*simp add*: *insR′*)
also have ... = *?U* $\gg\!=$ (*λu. return ?treap-sigma* (*ins x u t*) $\gg\!=$ (*λt. rinss xs t A*))
using *con-assms Cons* **by** (*subst bind-assoc*) *auto*
also have ... = *?U* $\gg\!=$ (*λu. rinss xs* (*ins x u t*) *A*)
using *con-assms Cons* **by** (*subst bind-return*) *auto*
also have ... = *?U* $\gg\!=$
(*λu.* Pi_M (*set xs*) (*λx. ?U*) $\gg\!=$
(*λp. return ?treap-sigma* (*foldl* (*λt x. ins x* (*p x*) *t*) (*ins x u t*) *xs*)))
using *Cons* **by** (*subst Cons*) (*auto simp add*: *treap-ins keys-ins*)
also have ... = *?U* $\bigotimes_M$ Pi_M (*set xs*) (*λx. ?U*) $\gg\!=$
(*λ*(*u*,*p*). *return ?treap-sigma* (*foldl* (*λt x. ins x* (*p x*) *t*) (*ins x u t*) *xs*))
proof −
have [*measurable*]: *pair-prob-space* (*?U*) (Pi_M (*set xs*) (*λx. ?U*))
by (*simp add*: $\mathcal{U}$*-def pair-prob-space-def pair-sigma-finite.intro prob-space-PiM*
prob-space-imp-sigma-finite prob-space-uniform-measure)
note *this*[*unfolded* $\mathcal{U}$*-def*, *measurable*]
have [*measurable*]: *c* $\in$ *set xs* $\Longrightarrow$ *c* $\in$ *A* **for** *c*
using *Cons* **by** *auto*
show *?thesis*
using *con-assms Cons* **by** (*subst pair-prob-space.pair-measure-bind*) *measurable*
qed
also have ... = *distr* (*?U* $\bigotimes_M$ Pi_M (*set xs*) (*λx. ?U*)) (*tree-sigma* (*count-space A* $\bigotimes_M$ *lborel*))
(*λ*(*u*, *f*). *foldl* (*λt x. ins x* (*f x*) *t*) (*ins x u t*) *xs*)
proof −
have [*simp*]: *c* $\in$ *set xs* $\Longrightarrow$ *c* $\in$ *A* **for** *c*
using *Cons* **by** *auto*
have (*λxa. foldl* (*λt x. ins x* (*snd xa x*) *t*) (*ins x* (*fst xa*) *t*) *xs*) =
(*λ*(*u*, *f*). *foldl* (*λt x. ins x* (*f x*) *t*) (*ins x u t*) *xs*)
by (*auto simp add*: *case-prod-beta′*)
then show *?thesis*
using *con-assms Cons* **by** (*subst case-prod-beta′*, *subst bind-return-distr′*) *measurable*

```
  qed
  also have
    ... = distr (?U ⨂M Pi_M (set xs) (λi. ?U)) ?treap-sigma
        (λf. foldl (λt y. ins y (if y = x then fst f else snd f y) t) (ins x (fst f) t) xs)
  proof −
    have foldl (λt y. ins y (snd f y) t) (ins x (fst f) t) xs =
        foldl (λt y. ins y (if y = x then fst f else snd f y) t) (ins x (fst f) t) xs for f
      using Cons by (intro foldl-cong) auto
    then show ?thesis
      by (auto simp add: case-prod-beta')
  qed
  also have ... = distr (?U ⨂M Pi_M (set xs) (λi. ?U)) (Pi_M (insert x (set xs))
(λi. ?U))
                        (λ(r, f). f(x := r)) ⤜
                        (λp. return ?treap-sigma (foldl (λt x. ins x (p x) t) (ins x (p
x) t) xs))
    using con-assms  Cons
    by (subst bind-distr-return) (measurable, auto simp add: case-prod-beta')
  also have ... = Pi_M (insert x (set xs)) (λx. ?U) ⤜
                (λp. return ?treap-sigma (foldl (λt x. ins x (p x) t) (ins x (p x) t)
xs))
    by (subst distr-pair-PiM-eq-PiM) (auto simp add: prob-space-uniform-measure)
  finally show ?case
    by (simp)
qed
  then show ?thesis
   using assms by (subst bind-return-distr'[symmetric]) (auto simp add: bind-return-distr')
qed

corollary random-treap-fold-Leaf:
  shows rinss xs Leaf A =
        distr (Π_M x∈set xs. 𝒰 0 1)
              (tree-sigma (count-space A ⨂M lborel))
              (λp. foldl (λt x. ins x (p x) t) Leaf xs)
  by (auto simp add: random-treap-fold)
```

Next, we show that additionally forgetting the priorities in the end will yield the same distribution as inserting the elements into a BST by ascending priority.

```
lemma rinss-bst-of-list:
      distr (rinss xs Leaf A) (tree-sigma (count-space A)) (map-tree fst) =
       distr (Pi_M (set xs) (λx. 𝒰 0 1)) (tree-sigma (count-space A))
            (λp. bst-of-list (sort-key p xs)) (is ?lhs = ?rhs)
proof −
  have [measurable]: set xs ⊆ space (count-space A)
    c ∈ set xs ⟹ c ∈ space (count-space A) for c
    using con-assms by auto
  have [simp]: map-tree fst ∘ (λp. foldl (λt x. ins x (p x) t) ⟨⟩ xs)
              ∈ Pi_M (set xs) (λx. uniform-measure lborel {0::real..1}) →M
```

```
            tree-sigma (count-space A)
    unfolding 𝒰-def map-tree-primrec using con-assms by measurable
  have AE f in Pi_M (set xs) (λi. 𝒰 0 1). inj-on f (set xs)
    unfolding 𝒰-def by (rule almost-everywhere-avoid-finite) auto
  then have AE f in Pi_M (set xs) (λx. 𝒰 0 1).
          map-tree fst (foldl (λt (k,p). ins k p t) ⟨⟩ (map (λx. (x, f x)) xs)) =
          bst-of-list (sort-key f xs)
    by (eventually-elim) (use con-assms in ‹auto simp add: fold-ins-bst-of-list›)
  then have [simp]: AE f in Pi_M (set xs) (λx. 𝒰 0 1).
          map-tree fst (foldl (λt k. ins k (f k) t) ⟨⟩ xs) = bst-of-list (sort-key f xs)
    by (simp add: foldl-map)
  have ?lhs = distr (Pi_M (set xs) (λx. 𝒰 0 1)) (tree-sigma (count-space A))
                (map-tree fst ∘ (λp. foldl (λt x. ins x (p x) t) ⟨⟩ xs))
    unfolding random-treap-fold-Leaf 𝒰-def map-tree-primrec using con-assms
    by (subst distr-distr) measurable
  also have ... = ?rhs
    by (intro distr-cong-AE) (auto simp add: 𝒰-def)
  finally show ?thesis .
qed
```

This in turn is the same as choosing a random permutation of the input list and inserting the elements into a BST in that order.

```
lemma lborel-permutations-of-set-bst-of-list:
  shows distr (Pi_M (set xs) (λx. 𝒰 0 1)) (tree-sigma (count-space A))
            (λp. bst-of-list (sort-key p xs)) =
        distr (random-perm xs) (tree-sigma (count-space A)) bst-of-list (is ?lhs =
?rhs)
proof −
  have [measurable]: (0::real) < 1
    by auto
  have insort-wrt R xs = insort-wrt R (remdups xs) for R
    using con-assms distinct-remdups-id by metis
  then have ∗: insort-wrt R xs = sorted-wrt-list-of-set R (set xs)
    if linorder-on (set xs) R for R
    using that by (subst sorted-wrt-list-set) auto
   have [measurable]: (λx. x) ∈ count-space (permutations-of-set (set xs)) →_M
count-space (lists A)
    using con-assms permutations-of-setD by fastforce
  have [measurable]: (λR. insort-wrt R xs) ∈
                  count-space (Pow (A × A)) →_M count-space (permutations-of-set
(set xs))
    using con-assms by (simp add: permutations-of-setI)
  have ?lhs
   = distr (Pi_M (set xs) (λx. 𝒰 0 1)) (tree-sigma (count-space A))
         (λp. bst-of-list (insort-wrt (linorder-from-keys (set xs) p) xs))
    unfolding Let-def by (simp add: insort-wrt-sort-key)
 also have ... =
  distr (distr (Pi_M (set xs) (λx. uniform-measure lborel {0::real..1}))
    (count-space (Pow (A × A))) (linorder-from-keys (set xs)))
```

```
  (tree-sigma (count-space A)) (λR. bst-of-list (insort-wrt R xs))
    unfolding 𝒰-def using con-assms by (subst distr-distr) (measurable, metis
comp-apply)
  also have ... =
  distr (uniform-measure (count-space (Pow (A × A))) (linorders-on (set xs)))
        (tree-sigma (count-space A)) (λR. bst-of-list (insort-wrt R xs))
    using con-assms by (subst random-linorder-by-prios) auto
  also have ... = distr (distr (uniform-measure (count-space (Pow (A × A)))
(linorders-on (set xs)))
                               (count-space (permutations-of-set (set xs))) (λR. insort-wrt
R xs))
                        (tree-sigma (count-space A)) bst-of-list
    by (subst distr-distr) (measurable, metis comp-apply)
  also have ... = distr (uniform-measure (count-space (permutations-of-set (set
xs)))
                          ((λR. insort-wrt R xs) ‘ linorders-on (set xs)))
                    (tree-sigma (count-space A)) bst-of-list
  proof −
    have bij-betw (λR. insort-wrt R xs) (linorders-on (set xs)) (permutations-of-set
(set xs))
      by (subst bij-betw-cong, fastforce simp add: ∗ linorders-on-def bij-betw-cong)
        (use bij-betw-linorders-on′ in blast)
    then have inj-on (λR. insort-wrt R xs) (linorders-on (set xs))
      by (rule bij-betw-imp-inj-on)
    then have distr (uniform-measure (count-space (Pow (A × A))) (linorders-on
(set xs)))
                    (count-space (permutations-of-set (set xs))) (λR. insort-wrt R xs)
              = uniform-measure (count-space (permutations-of-set (set xs)))
                                ((λR. insort-wrt R xs) ‘ linorders-on (set xs))
      using con-assms by (intro distr-uniform-measure-count-space-inj)
        (auto simp add: linorders-on-def linorder-on-def refl-on-def)
    then show ?thesis by auto
  qed
  also have ... = distr (random-perm xs) (tree-sigma (count-space A)) bst-of-list
  proof −
    have ((λR. insort-wrt R xs) ‘ linorders-on (set xs)) = permutations-of-set (set
xs)
      by (intro bij-betw-imp-surj-on, subst bij-betw-cong, rule ∗)
        (fastforce simp add: linorders-on-def, use bij-betw-linorders-on′ in blast)
    then show ?thesis by (simp add: random-perm-def)
  qed
  finally show ?thesis .
qed

lemma distr-bst-of-list-tree-sigma-count-space:
  distr (random-perm xs) (tree-sigma (count-space A)) bst-of-list =
    distr (random-perm xs) (count-space (trees A)) bst-of-list
  using con-assms by (intro distr-cong) (auto intro!: sets-tree-sigma-count-space)
```

This is the same as a *random BST*.

lemma *distr-bst-of-list-random-bst*:
  distr (*random-perm xs*) (*count-space* (*trees A*)) *bst-of-list* =
    restrict-space (*random-bst* (*set xs*)) (*trees A*) (**is** *?lhs* = *?rhs*)
proof −
  have *?rhs* = *restrict-space* (*distr* (*uniform-measure* (*count-space UNIV*)
            (*permutations-of-set* (*set xs*))) (*count-space UNIV*) *bst-of-list*) (*trees A*)
    by (*auto simp*: *random-bst-altdef measure-pmf-of-set map-pmf-rep-eq*)
  also have *distr* (*uniform-measure* (*count-space UNIV*) (*permutations-of-set* (*set xs*)))
              (*count-space UNIV*) *bst-of-list* =
            distr (*random-perm xs*) (*count-space UNIV*) *bst-of-list*
    by (*intro distr-restrict*) (*auto simp*: *random-perm-def*)
  also have *restrict-space* ... (*trees A*) =
            distr (*random-perm xs*) (*count-space* (*trees A*)) *bst-of-list*
    using *con-assms*
    by (*subst restrict-distr*)
       (*auto simp*: *random-perm-def bst-of-list-trees restrict-count-space permutations-of-setD*)
  finally show *?thesis* **..**
qed

We put everything together and obtain our main result:

theorem *rinss-random-bst*:
  distr (*rinss xs* ⟨⟩ *A*) (*tree-sigma* (*count-space A*)) (*map-tree fst*) =
    restrict-space (*measure-pmf* (*random-bst* (*set xs*))) (*trees A*)
  by (*simp only*: *rinss-bst-of-list lborel-permutations-of-set-bst-of-list*
            distr-bst-of-list-tree-sigma-count-space distr-bst-of-list-random-bst)

end
end

References

[1] M. Eberl, M. Haslbeck, and T. Nipkow. Verified analysis of random trees, 2018 (forthcoming).

[2] R. Seidel and C. R. Aragon. Randomized search trees. *Algorithmica*, 16(4):464–497, Oct 1996.