

# Treaps

Max Haslbeck, Manuel Eberl, Tobias Nipkow

May 26, 2024

## 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

<b>1</b>	<b>Auxiliary material</b>	<b>2</b>
<b>2</b>	<b>Treaps</b>	<b>8</b>
<b>3</b>	<b>Randomly-permuted lists</b>	<b>18</b>
3.1	General facts about linear orderings . . . . .	18
<b>4</b>	<b>Relationship between treaps and BSTs</b>	<b>28</b>
<b>5</b>	<b>Random treaps</b>	<b>32</b>
5.1	Measurability . . . . .	32
5.2	Main result . . . . .	36

# 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:  $\Omega \subseteq B$*

**assumes** *ae: AE x in M. x  $\in$   $\Omega$  AE x in N. x  $\in$   $\Omega$  and [simp]: countable  $\Omega$*

**assumes** *eq:  $\bigwedge x. x \in \Omega \implies \text{emeasure } M \{x\} = \text{emeasure } N \{x\}$*

**shows** *M = N*

**proof** (*rule measure-eqI*)

**fix** *A assume A: A  $\in$  sets M*

**have** *emeasure N A = emeasure N {x $\in$  $\Omega$ . x  $\in$  A}*

**using** *ae subset A by (intro emeasure-eq-AE) auto*

**also have** *... = ( $\int$   $^+x$ . emeasure N {x}  $\partial$ count-space {x $\in$  $\Omega$ . x  $\in$  A})*

**using** *A subset by (intro emeasure-countable-singleton) auto*

**also have** *... = ( $\int$   $^+x$ . emeasure M {x}  $\partial$ count-space {x $\in$  $\Omega$ . x  $\in$  A})*

**by** (*intro nn-integral-cong eq[symmetric] auto*)

**also have** *... = emeasure M {x $\in$  $\Omega$ . x  $\in$  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  $\Rightarrow$  'b::{second-countable-topology, linorder-topology}*

**assumes** *f  $\in$  borel-measurable M g  $\in$  borel-measurable M*

**shows** *Measurable.pred M ( $\lambda x$ . f x  $\leq$  g x)*

**unfolding** *pred-def by (intro borel-measurable-le assms)*

**lemma** *measurable-eq[measurable (raw)]*:

**fixes** *f :: 'a  $\Rightarrow$  'b::{second-countable-topology, linorder-topology}*

**assumes** *f  $\in$  borel-measurable M g  $\in$  borel-measurable M*

**shows** *Measurable.pred M ( $\lambda x$ . f x = g x)*

**unfolding** *pred-def by (intro borel-measurable-eq assms)*

**context**

**fixes** *M :: 'a measure*

**assumes** *singleton-null-set: x  $\in$  space M  $\implies$  {x}  $\in$  null-sets M*

**begin**

**lemma** *countable-null-set*:

**assumes** *countable A A  $\subseteq$  space M*

**shows** *A  $\in$  null-sets M*

**proof** –

**have** ( $\bigcup x \in A$ . {x})  $\in$  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 ⊆ space M*  
 shows *A ∈ 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]:  $\bigwedge i j. \text{Measurable.pred } (M i \otimes_M M j) (\lambda(x,y). x = y)$*   
 shows *Measurable.pred (Pi\_M I M) ( $\lambda x. \text{inj-on } x I$ )* **unfolding** *inj-on-def*  
 by *measurable*

**lemma** *almost-everywhere-not-in-countable-set*:  
 assumes *countable A*  
 assumes *[measurable]: Measurable.pred (M ⊗\_M M) ( $\lambda(x,y). x = y$ )*  
 assumes *null:  $\bigwedge x. x \in \text{space } M \implies \{x\} \in \text{null-sets } M$*   
 shows *AE x in M. x ∉ A*  
**proof** –  
 have *A ∩ space M ∈ 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:  $\bigwedge i. i \in I \implies \text{prob-space } (M i)$*   
 assumes *[measurable]:  $\bigwedge i j. \text{Measurable.pred } (M i \otimes_M M j) (\lambda(x,y). x = y)$*   
 assumes *null:  $\bigwedge i x. i \in I \implies x \in \text{space } (M i) \implies \{x\} \in \text{null-sets } (M i)$*   
 shows *AE f in (Pi\_M i ∈ I. M i). inj-on f I*  
**proof** –  
 note *[measurable] = measurable-inj-on-finite*  
 define *I' where I' = I*  
 hence *I ⊆ 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.prem*s  
 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 ⊗\_M Pi\_M I M) (Pi\_M (insert i I) M)*  
 ( $\lambda(x, X). X(i := x)$ )

```

    using insert.premis
    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))  $\longleftrightarrow$ 
    (AE x in M i  $\otimes$  PiM 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 PiM I M. inj-on (y(i := x)) (insert i
I))
    by (rule AE-pair-iff [symmetric]) measurable
  also have ...  $\longleftrightarrow$  (AE x in M i. AE y in PiM I M. inj-on (y(i := x)) I)  $\wedge$ 
    (AE x in M i. AE y in PiM I M. x  $\notin$  y(i := x) ' (I - {i})) by
simp
  also have ...
  proof (rule conjI, goal-cases)
    case 1
    from insert.premis have AE f in PiM I M. inj-on f I by (intro insert.IH)
  auto
  hence AE f in PiM 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  $\cap$  space (M i)  $\in$  null-sets (M i)
    by (rule finite-null-set)
    (insert insert.hyps insert.premis, auto intro!: null simp: I'-def)
  hence AE x in M i. x  $\notin$  f(i := x) ' I
    by (rule AE-I') (insert insert.hyps, auto split: if-splits)
  also have (AE x in M i. x  $\notin$  f(i := x) ' I)  $\longleftrightarrow$  (AE x in M i.  $\forall y \in I. f y$ 
 $\neq$  x)
    using insert.hyps by (intro AE-cong) (auto split: if-splits)
  finally have ... .
}
hence AE f in PiM I M. AE x in M i.  $\forall y \in I. f y \neq x$  by blast
hence AE x in M i. AE f in PiM I M.  $\forall y \in I. f y \neq x$ 
  by (subst AE-commute) simp-all
also have ?this  $\longleftrightarrow$  (AE x in M i. AE y in PiM I M. x  $\notin$  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**  $A \in \text{sets } M$  *emeasure*  $M A \neq \infty$

**shows**  $\text{null-sets (uniform-measure } M A) = (\lambda B. A \cap B) - \text{' null-sets } M \cap \text{sets}$

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

**lemma** *almost-everywhere-avoid-finite*:  
**assumes** *fin: finite I*  
**shows** *AE f in (Π<sub>M</sub> 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 →<sub>M</sub> N f ∈ M' →<sub>M</sub> N' A ∈ sets N' sets M' ⊆ sets M sets N' ⊆ sets N*  
**assumes**  $\bigwedge X. X \in \text{sets } M' \implies \text{emeasure } M X = \text{emeasure } M' X$   
**assumes**  $\bigwedge X. X \in \text{sets } M \implies X \subseteq \text{space } M - \text{space } M' \implies \text{emeasure } M X = 0$   
**shows** *emeasure (distr M N f) A = emeasure (distr M' N' f) A*  
**proof** –  
**have** *space-subset: space M' ⊆ space M*  
**using**  $\langle \text{sets } M' \subseteq \text{sets } M \rangle$  **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**  $\text{emeasure } M (f^{-1} A \cap (\text{space } M - \text{space } M')) = 0$   
**by** (*intro assms*) *auto*  
**ultimately show**  $f^{-1} A \cap (\text{space } M - \text{space } M') \in \text{null-sets } M$   
**unfolding** *null-sets-def* **by** *blast*  
**qed**  
**also have**  $\dots = \text{emeasure } M' (f^{-1} A \cap \text{space } M')$   
**using** *assms* **by** (*intro assms*) *auto*  
**also have**  $\dots = \text{emeasure } (\text{distr } M' N' f) A$   
**using** *assms* **by** (*subst emeasure-distr*) *auto*  
**finally show** *?thesis* .  
**qed**

**lemma** *distr-uniform-measure-count-space-inj*:  
**assumes** *inj-on f A' A'  $\subseteq$  A f^{-1} A  $\subseteq$  B finite A'*  
**shows**  $\text{distr } (\text{uniform-measure } (\text{count-space } A) A') (\text{count-space } B) f =$   
 $\text{uniform-measure } (\text{count-space } B) (f^{-1} A')$  (*is ?lhs = ?rhs*)  
**proof** (*rule measure-eqI, goal-cases*)  
**case** (*2 X*)  
**hence** *X-subset: X  $\subseteq$  B* **by** *simp*  
**from** *assms* **have**  $\text{eq: } f^{-1} A' \cap X = f^{-1} (A' \cap (f^{-1} X \cap A))$   
**by** *auto*  
**from** *assms* **have** [*measurable*]:  $f \in \text{count-space } A \rightarrow_M \text{count-space } B$   
**by** (*subst measurable-count-space-eq1*) *auto*  
**from** *X-subset* **have**  $\text{emeasure } ?lhs X =$   
 $\text{emeasure } (\text{uniform-measure } (\text{count-space } A) A') (f^{-1} X \cap A)$   
**by** (*subst emeasure-distr*) *auto*  
**also from** *assms* *X-subset*  
**have**  $\dots = \text{emeasure } (\text{count-space } A) (A' \cap (f^{-1} X \cap A)) / \text{emeasure}$   
 $(\text{count-space } A) A'$   
**by** (*intro emeasure-uniform-measure*) *auto*  
**also from** *assms* **have**  $\dots = \text{of-nat } (\text{card } (A' \cap (f^{-1} X \cap A))) / \text{of-nat } (\text{card}$   
 $A')$   
**by** (*subst (1 2) emeasure-count-space*) *auto*  
**also have**  $\text{card } (A' \cap (f^{-1} X \cap A)) = \text{card } (f^{-1} (A' \cap (f^{-1} X \cap A)))$   
**using** *assms* **by** (*intro card-image [symmetric]*) (*auto simp: inj-on-def*)  
**also have**  $f^{-1} (A' \cap (f^{-1} X \cap A)) = f^{-1} A' \cap X$   
**using** *assms* **by** *auto*  
**also have**  $\text{of-nat } (\text{card } A') = \text{of-nat } (\text{card } (f^{-1} A'))$   
**using** *assms* **by** (*subst card-image*) *auto*  
**also have**  $\text{of-nat } (\text{card } (f^{-1} A' \cap X)) / \dots =$   
 $\text{emeasure } (\text{count-space } B) (f^{-1} A' \cap X) / \text{emeasure } (\text{count-space } B) (f^{-1} A')$   
**using** *assms* **by** (*subst (1 2) emeasure-count-space*) *auto*  
**also from** *assms* *X-subset* **have**  $\dots = \text{emeasure } ?rhs X$   
**by** (*intro emeasure-uniform-measure [symmetric]*) *auto*  
**finally show** *?case* .  
**qed** *simp-all*

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

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

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

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

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

**shows**  $\text{distr } (PiM A (\lambda-. \text{uniform-measure lborel } \{a..b\})) (PiM A (\lambda-. \text{lborel}))$   
 $(\lambda f. f \circ g) =$   
 $PiM A (\lambda-. \text{uniform-measure lborel } \{a..b\})$   
**proof** –  
**have**  $\text{distr } (PiM A (\lambda-. \text{uniform-measure lborel } \{a..b\})) (PiM A (\lambda-. \text{lborel})) (\lambda f.$   
 $f \circ g) =$   
 $\text{distr } (PiM A (\lambda-. \text{uniform-measure lborel } \{a..b\}))$   
 $(PiM A (\lambda-. \text{uniform-measure lborel } \{a..b\})) (\lambda f. \lambda x \in A. f (g x))$  **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**  $\dots = PiM A (\lambda i. \text{uniform-measure lborel } \{a..b\})$   
**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**  $a * (b :: \text{ennreal}) = 1$   
**shows**  $b = \text{inverse } a$   
**using** *assms*  
**by** (*metis divide-ennreal-def ennreal-inverse-1 ennreal-top-eq-mult-iff mult.comm-neutral*  
 $\text{mult-divide-eq-ennreal mult-eq-0-iff semiring-normalization-rules(7)}$ )

**end**

## 2 Treaps

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

**definition** *treap*  $:: ('k::\text{linorder} * 'p::\text{linorder}) \text{tree} \Rightarrow \text{bool}$  **where**  
 $\text{treap } t = (\text{bst } (\text{map-tree fst } t) \wedge \text{heap } (\text{map-tree snd } t))$

**abbreviation** *keys*  $t \equiv \text{set-tree } (\text{map-tree fst } t)$   
**abbreviation** *prios*  $t \equiv \text{set-tree } (\text{map-tree snd } t)$

**function** *treap-of*  $:: ('k::\text{linorder} * 'p::\text{linorder}) \text{set} \Rightarrow ('k * 'p) \text{tree}$  **where**  
 $\text{treap-of } KP = (\text{if infinite } KP \vee KP = \{\} \text{ then Leaf else}$   
 $\text{let } m = \text{arg-min-on snd } KP;$   
 $L = \{p \in KP. \text{fst } p < \text{fst } m\};$   
 $R = \{p \in KP. \text{fst } p > \text{fst } m\}$



```

    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 = Node l a r
  using False by (cases t) auto
  have ∀ kp ∈ set-tree t. snd a ≤ snd kp
  using 1.prem(1)
  by(auto simp add: t treap-def ball-Un)
  (metis image-eqI snd-conv tree.set-map)+
  hence a = ?m
  by (metis 1.prem(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.prem(1) by(auto simp: treap-def t)
  have l: set-tree l = {p ∈ set-tree t. fst p < fst a}

```

```

    using 1.premis(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.premis(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.premis(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.premis(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*:  $\text{prios } (\text{ins } k \ p \ t) \subseteq \{p\} \cup \text{prios } t$   
**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'*:  $k \notin \text{keys } t \implies \text{prios } (\text{ins } k \ p \ t) = \{p\} \cup \text{prios } t$   
**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*:  $\text{set-tree } (\text{ins } k \ p \ t) \subseteq \{(k,p)\} \cup \text{set-tree } t$   
**by** (*induction t rule: ins.induct*) (*auto simp add: ins-neq-Leaf split: tree.split prod.split*)

**lemma** *set-tree-ins'*:  $k \notin \text{keys } t \implies \{(k,p)\} \cup \text{set-tree } t \subseteq \text{set-tree } (\text{ins } k \ p \ t)$   
**by** (*induction t rule: ins.induct*) (*auto simp add: ins-neq-Leaf split: tree.split prod.split*)

**lemma** *set-tree-ins-eq*:  $k \notin \text{keys } t \implies \text{set-tree } (\text{ins } k \ p \ t) = \{(k,p)\} \cup \text{set-tree } t$   
**using** *set-tree-ins set-tree-ins'* **by** *blast*

**lemma** *prios-ins-special*:  
 $\llbracket \text{ins } k \ p \ t = \text{Node } l \ (k',p') \ r; \ p' = p; \ p \in \text{prios } r \cup \text{prios } l \rrbracket$   
 $\implies p \in \text{prios } t$   
**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*:  
 $\llbracket \text{treap } l; \text{treap } r;$   
 $\forall k' \in \text{keys } l. k' < k; \forall k' \in \text{keys } r. k < k';$   
 $\forall p' \in \text{prios } l. p \leq p'; \forall p' \in \text{prios } r. p \leq p' \rrbracket$   
 $\implies \text{treap } (\text{Node } l \ (k,p) \ r)$   
**by** (*auto simp: treap-def*)

**lemma** *treap-rotate1*:  
**assumes** *treap l2 treap r2 treap r*  $\neg p1 \leq p2 \ k < k1$  **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  $\subseteq$  {k}  $\cup$  keys l* **by**(*auto simp: ins*)  
**from** *treap* **have** *2:  $\forall k' \in \text{keys } l. k' < k1$*  **by** (*simp add: treap-def*)  
**show**  $\forall k' \in \text{keys } r2. k' < k1$  **using** *1 2 (k < k1)* **by** *blast*  
**next**  
**from** *treap* **have** *2:  $\forall p' \in \text{prios } l. p1 \leq p'$*  **by** (*simp add: treap-def*)  
**show**  $\forall p' \in \text{prios } r2. p1 \leq p'$

```

proof
  fix  $p'$  assume  $p' \in \text{prios } r2$ 
  hence  $p' = p \vee p' \in \text{prios } l$  using  $\text{prios-ins}[of\ k\ p\ l]$  ins by  $\text{auto}$ 
  thus  $p1 \leq p'$ 
proof
  assume  $[simp]: p' = p$ 
  have  $p2 = p \vee p2 \in \text{prios } l$  using  $\text{prios-ins}[of\ k\ p\ l]$  ins by  $\text{simp}$ 
  thus  $p1 \leq p'$ 
proof
  assume  $p2 = p$ 
  thus  $p1 \leq p'$ 
  using  $\text{prios-ins-special}[OF\ ins]\ \langle p' \in \text{prios } r2 \rangle\ 2$  by  $\text{auto}$ 
next
  assume  $p2 \in \text{prios } l$ 
  thus  $p1 \leq p'$  using  $2\ \langle \neg\ p1 \leq p2 \rangle$  by  $\text{blast}$ 
qed
next
  assume  $p' \in \text{prios } l$ 
  thus  $p1 \leq p'$  using  $2$  by  $\text{blast}$ 
qed
qed
next
  have  $k2 = k \vee k2 \in \text{keys } l$  using  $\text{keys-ins}[of\ k\ p\ l]$  ins by  $(\text{auto})$ 
  hence  $1: k2 < k1$ 
proof
  assume  $k2 = k$  thus  $k2 < k1$  using  $\langle k < k1 \rangle$  by  $\text{simp}$ 
next
  assume  $k2 \in \text{keys } l$ 
  thus  $k2 < k1$  using  $\text{treap}$  by  $(\text{auto } \text{simp}: \text{treap-def})$ 
qed
  have  $2: \forall k' \in \text{keys } r2. k2 < k'$ 
  using  $\text{treap-ins}$  by  $(\text{simp } \text{add}: \text{ins } \text{treap-def})$ 
  have  $3: \forall k' \in \text{keys } r. k2 < k'$ 
  using  $1\ \text{treap}$  by  $(\text{auto } \text{simp}: \text{treap-def})$ 
  show  $\forall k' \in \text{keys } \langle r2, (k1, p1), r \rangle. k2 < k'$  using  $1\ 2\ 3$  by  $\text{auto}$ 
next
  show  $\forall p' \in \text{prios } \langle r2, (k1, p1), r \rangle. p2 \leq p'$ 
  using  $\text{ins } \text{treap-ins } \text{treap}\ \langle \neg\ p1 \leq p2 \rangle$  by  $(\text{auto } \text{simp}: \text{treap-def } \text{ball-Un})$ 
qed  $(\text{use } \text{ins } \text{treap-ins } \text{treap } \text{in } \langle \text{auto } \text{simp}: \text{treap-def } \text{ball-Un} \rangle)$ 

```

**lemma**  $\text{treap-rotate2}$ :

```

assumes  $\text{treap } l\ \text{treap } l2\ \text{treap } r2\ \neg\ p1 \leq p2\ k1 < k$  and
   $\text{ins}: \text{ins } k\ p\ r = \text{Node } l2\ (k2, p2)\ r2$  and  $\text{treap-ins}: \text{treap } (\text{ins } k\ p\ r)$ 
and  $\text{treap}: \text{treap } \langle l, (k1, p1), r \rangle$ 
shows  $\text{treap } (\text{Node } (\text{Node } l\ (k1, p1)\ l2)\ (k2, p2)\ r2)$ 
proof  $(\text{rule } \text{treap-NodeI}[OF\ \text{treap-NodeI}[OF\ \langle \text{treap } l \rangle\ \langle \text{treap } l2 \rangle]\ \langle \text{treap } r2 \rangle])$ 
from  $\text{keys-ins}[of\ k\ p\ r]$  have  $1: \text{keys } l2 \subseteq \{k\} \cup \text{keys } r$  by  $(\text{auto } \text{simp}: \text{ins})$ 
from  $\text{treap}$  have  $2: \forall k' \in \text{keys } r. k1 < k'$  by  $(\text{simp } \text{add}: \text{treap-def})$ 

```

```

show  $\forall k' \in \text{keys } l2. k1 < k'$  using 1 2  $\langle k1 < k \rangle$  by blast
next
from treap have 2:  $\forall p' \in \text{prios } r. p1 \leq p'$  by (simp add: treap-def)
show  $\forall p' \in \text{prios } l2. p1 \leq p'$ 
proof
  fix p' assume p'  $\in$  prios l2
  hence p' = p  $\vee$  p'  $\in$  prios r using prios-ins[of k p r] ins by auto
  thus p1  $\leq$  p'
  proof
    assume [simp]: p' = p
    have p2 = p  $\vee$  p2  $\in$  prios r using prios-ins[of k p r] ins by simp
    thus p1  $\leq$  p'
    proof
      assume p2 = p
      thus p1  $\leq$  p'
      using prios-ins-special[OF ins]  $\langle p' \in \text{prios } l2 \rangle$  2 by auto
    next
      assume p2  $\in$  prios r
      thus p1  $\leq$  p' using 2  $\langle \neg p1 \leq p2 \rangle$  by blast
    qed
  next
    assume p'  $\in$  prios r
    thus p1  $\leq$  p' using 2 by blast
  qed
qed
next
have k2 = k  $\vee$  k2  $\in$  keys r using keys-ins[of k p r] ins by (auto)
hence 1: k1 < k2
proof
  assume k2 = k thus k1 < k2 using  $\langle k1 < k \rangle$  by simp
next
  assume k2  $\in$  keys r
  thus k1 < k2 using treap by (auto simp: treap-def)
qed
have 2:  $\forall k' \in \text{keys } l. k' < k2$  using 1 treap by (auto simp: treap-def)
have 3:  $\forall k' \in \text{keys } l2. k' < k2$ 
  using treap-ins by (auto simp: ins treap-def)
show  $\forall k' \in \text{keys } \langle l, (k1, p1), l2 \rangle. k' < k2$  using 1 2 3 by auto
next
show  $\forall p' \in \text{prios } \langle l, (k1, p1), l2 \rangle. p2 \leq p'$ 
  using ins treap-ins treap  $\langle \neg p1 \leq p2 \rangle$  by (auto simp: treap-def ball-Un)
qed (use ins treap-ins treap in  $\langle \text{auto simp: treap-def ball-Un} \rangle$ )

lemma treap-ins: treap t  $\implies$  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.premis 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.premis 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.premis])
  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.premis 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.premis])
  thus ?thesis using ⟨¬ k < k1⟩ ⟨k > k1⟩ ins ⟨¬ p1 ≤ p2⟩ by simp

```

```

qed
next
  assume  $\neg k > k1$ 
  hence  $k = k1$  using  $\langle \neg k < k1 \rangle$  by auto
  thus ?thesis using 2.prem by (simp)
qed
qed
qed

```

**lemma** *treap-of-set-tree-unique*:

```

[[ finite A; inj-on fst A; inj-on snd A ]]
 $\implies$  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  $\vee$  A = {})

```

```

  assume infinite A  $\vee$  A = {}

```

```

  with IH show ?thesis by (simp add: treap-of.simps)

```

```

next

```

```

  assume not-inf-or-empty:  $\neg$  (infinite A  $\vee$  A = {})

```

```

  let ?m = arg-min-on snd A

```

```

  let ?L = {p  $\in$  A. fst p < fst ?m}

```

```

  let ?R = {p  $\in$  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  $\in$  A. fst p < fst (arg-min-on snd A)}

```

```

    inj-on snd {p  $\in$  A. fst p < fst (arg-min-on snd A)}

```

```

    inj-on fst {p  $\in$  A. fst (arg-min-on snd A) < fst p}

```

```

    inj-on snd {p  $\in$  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  $\neq$  fst (arg-min-on snd A) if (a,b)  $\in$  A (a, b)  $\neq$  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)  $\in$  A (a, b)  $\neq$  arg-min-on snd A fst (arg-min-on snd A)  $\geq$  a for a b

```

```

    using le-neq-trans that by auto

```

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

**lemma** *treap-of-subset*: *set-tree* (*treap-of*  $A$ )  $\subseteq A$

**proof**(*induction A rule: treap-of.induct*)

**case** ( $1\ A$ )

**note**  $IH = 1$

**show** *?case*

**proof** (*cases infinite A  $\vee A = \{\}$* )

**assume** *infinite A  $\vee A = \{\}$*

**with**  $IH$  **show** *?thesis* **by** (*simp add: treap-of.simps*)

**next**

**assume** *not-inf-or-empty*:  $\neg$  (*infinite A  $\vee A = \{\}$* )

**let**  $?m = \text{arg-min-on snd } A$

**let**  $?L = \{p \in A. \text{fst } p < \text{fst } ?m\}$

**let**  $?R = \{p \in A. \text{fst } p > \text{fst } ?m\}$

**obtain**  $l\ a\ r$  **where**  $t: \text{treap-of } A = \text{Node } l\ a\ r$

**using** *not-inf-or-empty* **by** (*cases treap-of A*)

(*auto simp add: Let-def elim!: treap-of.elims split: if-splits*)

**have**  $l = \text{treap-of } ?L\ r = \text{treap-of } ?R$

**using**  $t$  **by** (*auto simp: Let-def elim: treap-of.elims split: if-splits*)

**have** *set-tree*  $l \subseteq ?L$  *set-tree*  $r \subseteq ?R$

**proof**  $-$

**have** *set-tree* (*treap-of*  $\{p \in A. \text{fst } p < \text{fst } (\text{arg-min-on snd } A)\}$ )

$\subseteq \{p \in A. \text{fst } p < \text{fst } (\text{arg-min-on snd } A)\}$

**by** (*rule IH*) (*use not-inf-or-empty in auto*)

**then show** *set-tree*  $l \subseteq ?L$

**using**  $\langle l = \text{treap-of } ?L \rangle$  **by** *auto*

**next**

**have** *set-tree* (*treap-of*  $\{p \in A. \text{fst } (\text{arg-min-on snd } A) < \text{fst } p\}$ )

$\subseteq \{p \in A. \text{fst } (\text{arg-min-on snd } A) < \text{fst } p\}$

**by** (*rule IH*) (*use not-inf-or-empty in auto*)

**then show** *set-tree*  $r \subseteq ?R$

**using**  $\langle r = \text{treap-of } ?R \rangle$  **by** *auto*

**qed**

**moreover have**  $a = ?m$

**using**  $t$  **by** (*auto elim!: treap-of.elims simp add: Let-def split: if-splits*)

**moreover have**  $\{?m\} \cup ?L \cup ?R \subseteq A$

**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  $\vee$  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  $\in$  A. fst p < fst ?m}
  let ?R = {p  $\in$  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  $\subseteq$  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  $\subseteq$  fst ' ?R
    by (auto simp add: tree.set-map intro!: image-mono treap-of-subset)
  have prios: prios l  $\subseteq$  snd ' A prios r  $\subseteq$  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:  $\bigwedge x. x \in$  prios l  $\implies$  snd a  $\leq$  x
    by (drule rev-subsetD[OF - prios(1)] (use arg-min-least a False in fast))
  have prios-r:  $\bigwedge x. x \in$  prios r  $\implies$  snd a  $\leq$  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  $\langle \rangle$ 
  by (simp add: treap-def)

```

```

lemma foldl-ins-treap: treap t  $\implies$  treap (foldl ( $\lambda t' (x, p). ins x p t')$  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*  $(set\ ys) \cap keys\ t = \{\}$   
**shows** *set-tree (foldl ( $\lambda t' (x, p). ins\ x\ p\ t'$ ) t ys) = set ys  $\cup$  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 ( $\lambda t' (x, p). ins\ x\ p\ t'$ ) 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*  $:: 'a\ set \Rightarrow ('a \times 'a)\ set\ set$  **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  $\neq$  {}*

**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 ⊆ A × 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 (λR. 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  $\pi$  xs) x = inv  $\pi$  (index xs x)
proof -
  have *: inv  $\pi$  permutes {.. $\text{length}$  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  $\pi$  permutes {.. $\text{length}$  xs} distinct xs
  shows linorder-of-list (permute-list  $\pi$  xs) =
    map-relation (set xs) (!) xs  $\circ$  inv  $\pi$   $\circ$  index xs (linorder-of-list xs)
proof -
  note * = permutes-inv[OF assms(1)]
  have less: inv  $\pi$  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  $\longleftrightarrow$  ( $\forall y \in f'A. \exists !x \in A. y = f x$ )
  by (auto simp: inj-on-def)

lemma bij-betw-conv-Ex1: bij-betw f A B  $\longleftrightarrow$  ( $\forall y \in B. \exists !x \in A. f x = y$ )  $\wedge$  B = f
  ' A
  unfolding bij-betw-def inj-on-conv-Ex1 by (auto simp: eq-commute)

lemma permutesI:
  assumes bij-betw f A A  $\forall x. x \notin A \longrightarrow f x = x$ 
  shows f permutes A
  unfolding permutes-def
proof (intro conjI allI impI)
  fix y
  from assms have [simp]: f x  $\in$  A  $\longleftrightarrow$  x  $\in$  A for x
    by (auto simp: bij-betw-def)
  show  $\exists !x. f x = y$ 
  proof (cases y  $\in$  A)
    case True
    also from assms have A = f ' A by (auto simp: bij-betw-def)
    finally obtain x where x  $\in$  A y = f x by auto
    with assms and  $\langle y \in A \rangle$  show ?thesis
    by (intro exI[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**  $\pi$  **where**  $\pi$  *permutes A R' = map-relation A  $\pi$  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**  $\pi$  **where**  $\pi$ :  $\pi$  *permutes  $\{..<length\ xs\}$  permute-list  $\pi$  xs = ys*  
**by** *(rule mset-eq-permutation)*  
**define**  $\pi'$  **where**  $\pi' = (\lambda x. \text{if } x \notin A \text{ then } x \text{ else } xs ! \text{inv } \pi \text{ (index } xs \ x))$   
**have**  $\pi'$ :  $\pi'$  *permutes A*  
**proof** *(rule permutesI)*  
**have** *bij-betw  $(!!)$  xs  $\circ$  inv  $\pi$   $\{..<length\ xs\}$  A*  
**by** *(rule bij-betw-trans permutes-imp-bij permutes-inv  $\pi$  bij-betw-nth)+ (simp-all add: xs-ys)*  
**hence** *bij-betw  $(!!)$  xs  $\circ$  inv  $\pi$   $\circ$  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  $\circ$  inv  $\pi$   $\circ$  index xs) A A  $\longleftrightarrow$  bij-betw  $\pi'$  A A*  
**by** *(rule bij-betw-cong) (auto simp:  $\pi'$ -def)*  
**finally show**  $\dots$  .  
**qed** *(simp-all add:  $\pi'$ -def)*

**from** *assms have R' = linorder-of-list ys by (simp add: ys-def)*  
**also from**  $\pi$  **have** *ys = permute-list  $\pi$  xs by simp*  
**also have** *linorder-of-list (permute-list  $\pi$  xs) =*  
*map-relation A  $(!!)$  xs  $\circ$  inv  $\pi$   $\circ$  index xs) (linorder-of-list xs)*  
**using**  $\pi$  **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  $\circ$  inv  $\pi$   $\circ$  index xs) R .*  
**also have**  $\dots = \text{map-relation A } \pi' \text{ R by (auto simp: map-relation-def } \pi' \text{-def)}$   
**finally show** *?thesis using  $\pi'$  and that[of  $\pi'$ ] 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* ::  $'a \text{ set} \Rightarrow ('a \Rightarrow 'b :: \text{linorder}) \Rightarrow ('a \times 'a) \text{ set}$   
**where**

$$\text{linorder-from-keys } A \ f = \{(x,y) \in A \times A. f \ x \leq f \ y\}$$

**lemma** *linorder-from-keys-permute*:

**assumes** *g permutes A*

**shows** *linorder-from-keys A (f  $\circ$  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 ∈ A y ∈ A x ≠ y*

**assumes** *a < b finite A*

**defines** *M ≡ uniform-measure lborel {a..b}*

**shows** *emeasure (PiM A (λ-. M)) {h ∈ A →<sub>E</sub> 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 (PiM {x, y} (λi. lborel))*

**proof** –

**have** *{h ∈ extensional {x, y}. h x = h y} = {h ∈ space (PiM {x, y} (λi. lborel)). h x = h y}*

**by** (*auto simp: extensional-def space-PiM*)

**also have** *... ∈ sets (PiM {x, y} (λi. 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 ⊗<sub>M</sub> M) = sets (borel ⊗<sub>M</sub> 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 ⊗<sub>M</sub> 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 →<sub>E</sub> UNIV. h x = h y} =*

*(λh. restrict h {x, y}) - ' {h ∈ extensional {x, y}. h x = h y} ∩ space (PiM*

$A (\lambda-. M :: \text{real measure})$   
**by** (*auto simp: space-PiM PiE-def extensional-def M-def*)  
**also have**  $\text{emeasure } (PiM A (\lambda-. M)) \dots =$   
 $\text{emeasure } (\text{distr } (PiM A (\lambda-. M)) (PiM \{x,y\} (\lambda-. \text{lborel} :: \text{real measure})))$   
 $(\lambda h. \text{restrict } h \{x,y\}) \{h \in \text{extensional } \{x, y\}. h x = h y\}$   
**proof** (*rule emeasure-distr [symmetric]*)  
**have**  $(\lambda h. \text{restrict } h \{x, y\}) \in PiM A (\lambda-. \text{lborel}) \rightarrow_M PiM \{x, y\} (\lambda-. \text{lborel})$   
**using** *assms by (intro measurable-restrict-subset) auto*  
**also have**  $\dots = PiM A (\lambda-. M) \rightarrow_M PiM \{x, y\} (\lambda-. \text{lborel})$   
**by** (*intro sets-PiM-cong measurable-cong-sets refl*) (*simp-all add: M-def*)  
**finally show**  $(\lambda h. \text{restrict } h \{x, y\}) \in \dots$   
**next**  
**show**  $\{h \in \text{extensional } \{x, y\}. h x = h y\} \in \text{sets } (PiM \{x, y\} (\lambda-. \text{lborel}))$  **by**  
*simp*  
**qed**  
**also have**  $\text{distr } (PiM A (\lambda-. M)) (PiM \{x,y\} (\lambda-. \text{lborel} :: \text{real measure})) (\lambda h.$   
 $\text{restrict } h \{x,y\}) =$   
 $\text{distr } (PiM A (\lambda-. M)) (PiM \{x,y\} (\lambda-. M)) (\lambda h. \text{restrict } h \{x,y\})$   
**by** (*intro distr-cong refl sets-PiM-cong*) (*simp-all add: M-def*)  
**also from** *assms have*  $\dots = PiM \{x, y\} (\lambda i. M)$  **by** (*intro distr-restrict [symmetric]*)  
*auto*  
**also have**  $\text{emeasure } \dots \{h \in \text{extensional } \{x, y\}. h x = h y\} =$   
 $\text{nn-integral } \dots (\lambda h. \text{indicator } \{h \in \text{extensional } \{x, y\}. h x = h y\} h)$   
**by** (*intro nn-integral-indicator [symmetric]*) *simp-all*  
**also have**  $\dots = \text{nn-integral } (PiM \{x, y\} (\lambda i. M)) (\lambda h. \text{if } h x = h y \text{ then } 1 \text{ else } 0)$   
**by** (*intro nn-integral-cong*) (*auto simp add: indicator-def space-PiM PiE-def*)  
**also from**  $\langle x \neq y \rangle$  **have**  $\dots = (\int^+ z. (\text{if } \text{fst } z = \text{snd } z \text{ then } 1 \text{ else } 0) \partial(M \otimes_M M))$   
**by** (*intro product-nn-integral-pair*) *auto*  
**also have**  $\dots = (\int^+ x. (\int^+ y. (\text{if } x = y \text{ then } 1 \text{ else } 0) \partial M) \partial M)$   
**by** (*subst M.nn-integral-fst [symmetric]*) *simp-all*  
**also have**  $\dots = (\int^+ x. (\int^+ y. \text{indicator } \{x\} y \partial M) \partial M)$   
**by** (*simp add: indicator-def of-bool-def eq-commute*)  
**also have**  $\dots = (\int^+ x. \text{emeasure } M \{x\} \partial M)$  **by** (*subst nn-integral-indicator*)  
*(simp-all add: M-def)*  
**also have**  $\dots = (\int^+ x. 0 \partial M)$  **unfolding** *M-def*  
**by** (*intro nn-integral-cong-AE refl AE-uniform-measureI*) *auto*  
**also have**  $\dots = 0$  **by** *simp*  
**finally show** *?thesis* .  
**qed**

**lemma** *measurable-linorder-from-keys-restrict:*

**assumes** *fin: finite A*  
**shows** *linorder-from-keys*  $A \in PiM A (\lambda-. \text{borel} :: \text{real measure}) \rightarrow_M \text{count-space}$   
 $(Pow (A \times A))$   
*(is - : ?M  $\rightarrow_M$  -)*  
**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}  $\cap$  space ?M =
    {f  $\in$  space ?M.  $\forall x \in A. \forall y \in A. (x, y) \in R \iff f x \leq f y$ }
    by (auto simp add: linorder-from-keys-def set-eq-iff)
  also have ...  $\in$  sets ?M
    by measurable
  finally show linorder-from-keys A - ' {R}  $\cap$  space ?M  $\in$  sets ?M .
qed

```

```

lemma measurable-count-space-extend:
  assumes f  $\in$  measurable M (count-space A)  $A \subseteq B$ 
  shows f  $\in$  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  $A \subseteq B$ 
  shows linorder-from-keys A  $\in$   $Pi_M$  A ( $\lambda$ -. borel :: real measure)  $\rightarrow_M$  count-space
  (Pow (B  $\times$  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:  $A \subseteq B$ 
  defines M  $\equiv$  distr (PiM A ( $\lambda$ -. uniform-measure lborel {a..b}))
    (count-space (Pow (B  $\times$  B))) (linorder-from-keys A)
begin

```

```

lemma measurable-linorder-from-keys [measurable]:
  linorder-from-keys A  $\in$   $Pi_M$  A ( $\lambda$ -. borel :: real measure)  $\rightarrow_M$  count-space (Pow
  (B  $\times$  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 ( $\lambda$ -. uniform-measure lborel {a..b})
  have [simp]: sets (PiM A ( $\lambda$ -. uniform-measure lborel {a..b})) = sets (PiM A
  ( $\lambda$ -. 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:*

$M = \text{uniform-measure} (\text{count-space} (\text{Pow} (B \times B))) (\text{linorders-on } A)$

**proof** –

**from** *linorders-finite-nonempty*[*OF fin*] **obtain**  $R$  **where**  $R$ : *linorder-on A R*  
**by** (*auto simp: linorders-on-def*)

**have** \*:  $\text{emeasure } M \{R\} \leq \text{emeasure } M \{R'\}$  **if**  $\text{linorder-on } A \ R \ \text{linorder-on } A \ R'$  **for**  $R \ R'$   
**proof** –  
**define**  $N$  **where**  $N = \text{PiM } A \ (\lambda\cdot. \text{uniform-measure } \text{lborel } \{a..b\})$   
**from**  $\text{linorder-permutation-exists}[OF \ \text{fin } \text{that}]$   
**obtain**  $\pi$  **where**  $\pi: \pi \ \text{permutes } A \ R' = \text{map-relation } A \ \pi \ R$   
**by**  $\text{blast}$   
**have**  $(\lambda f. f \circ \pi) \in \text{PiM } A \ (\lambda\cdot. \text{lborel} :: \text{real measure}) \rightarrow_M \text{PiM } A \ (\lambda\cdot. \text{lborel} :: \text{real measure})$   
**by**  $(\text{auto } \text{intro!}: \text{measurable-PiM-single}' \ \text{measurable-PiM-component-rev} \ \text{simp}: \text{comp-def } \text{permutes-in-image}[OF \ \pi(1)] \ \text{space-PiM } \text{PiE-def } \text{extensional-def})$   
**also have**  $\dots = N \rightarrow_M \text{PiM } A \ (\lambda\cdot. \text{lborel})$   
**unfolding**  $N\text{-def}$  **by**  $(\text{intro } \text{measurable-cong-sets } \text{refl } \text{sets-PiM-cong}) \ \text{simp-all}$   
**finally have**  $[\text{measurable}]: (\lambda f. f \circ \pi) \in \dots$  .  
  
**have**  $[\text{simp}]: \text{measurable } N \ X = \text{measurable } (\text{PiM } A \ (\lambda\cdot. \text{lborel})) \ X$  **for**  $X :: ('a \times 'a) \ \text{set measure}$   
**unfolding**  $N\text{-def}$  **by**  $(\text{intro } \text{measurable-cong-sets } \text{refl } \text{sets-PiM-cong}) \ \text{simp-all}$   
**have**  $[\text{simp}]: \text{measurable } M \ X = \text{measurable } (\text{count-space } (\text{Pow } (B \times B))) \ X$   
**for**  $X :: ('a \times 'a) \ \text{set measure}$   
**unfolding**  $M\text{-def}$  **by**  $\text{simp}$   
  
**have**  $M\text{-eq}: M = \text{distr } N \ (\text{count-space } (\text{Pow } (B \times B))) \ (\text{linorder-from-keys } A)$   
**by**  $(\text{simp } \text{only}: M\text{-def } N\text{-def})$   
**also have**  $N = \text{distr } N \ (\text{PiM } A \ (\lambda\cdot. \text{lborel})) \ (\lambda f. f \circ \pi)$   
**unfolding**  $N\text{-def}$  **by**  $(\text{rule } \text{PiM-uniform-measure-permute } [\text{symmetric}]) \ \text{fact+}$   
**also have**  $\text{distr } \dots \ (\text{count-space } (\text{Pow } (B \times B))) \ (\text{linorder-from-keys } A) = \text{distr } N \ (\text{count-space } (\text{Pow } (B \times B))) \ (\text{linorder-from-keys } A \circ (\lambda f. f \circ \pi))$   
**by**  $(\text{intro } \text{distr-distr}) \ \text{simp-all}$   
**also have**  $\dots = \text{distr } N \ (\text{count-space } (\text{Pow } (B \times B))) \ (\text{map-relation } A \ \pi \circ \text{linorder-from-keys } A)$   
**by**  $(\text{intro } \text{distr-cong } \text{refl}) \ (\text{auto } \text{simp}: \text{linorder-from-keys-permute}[OF \ \pi(1)])$   
**also have**  $\dots = \text{distr } M \ (\text{count-space } (\text{Pow } (B \times B))) \ (\text{map-relation } A \ \pi)$   
**unfolding**  $M\text{-eq}$  **using**  $B$   
**by**  $(\text{intro } \text{distr-distr } [\text{symmetric}]) \ (\text{auto } \text{simp}: \text{map-relation-def})$   
**finally have**  $M\text{-eq}' : \text{distr } M \ (\text{count-space } (\text{Pow } (B \times B))) \ (\text{map-relation } A \ \pi) = M \dots$   
  
**from that have**  $\text{subset}' : R \subseteq B \times B \ R' \subseteq B \times B$   
**using**  $B$  **by**  $(\text{auto } \text{simp}: \text{linorder-on-def } \text{refl-on-def})$   
**hence**  $\text{emeasure } M \{R\} \leq \text{emeasure } M \ (\text{map-relation } A \ \pi - \{R'\} \cap \text{space } M)$   
**using**  $\text{subset}'$  **by**  $(\text{intro } \text{emeasure-mono}) \ (\text{auto } \text{simp}: M\text{-def } \pi)$   
**also have**  $\dots = \text{emeasure } (\text{distr } M \ (\text{count-space } (\text{Pow } (B \times B)))) \ (\text{map-relation } A \ \pi) \ \{R'\}$   
**by**  $(\text{rule } \text{emeasure-distr } [\text{symmetric}]) \ (\text{insert } \text{subset}' \ B, \ \text{auto } \text{simp}: \text{map-relation-def})$   
**also note**  $M\text{-eq}'$

**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] (λy ys zs. if f x ≤ f y then x # y # zs else y # zs) xs
  by (induction xs) simp-all

```

**lemma** *insort-key-conv-rec-list'*:  
*insort-key* = ( $\lambda f x.$   
*rec-list* [x] ( $\lambda y ys zs.$  if  $f x \leq f y$  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*  $\subseteq A$   
**shows** *bst-of-list ys*  $\in$  *trees A*  
**using** *assms* **by** (*induction ys rule: bst-of-list.induct*) *auto*

**lemma** *insort-wrt-insort-key*:  
 $a \in A \implies$   
 $set\ xs \subseteq A \implies$   
*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**  $set\ xs \subseteq A$   
**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**  $zs \neq []$  *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**  $m \in$  (*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*)))  $\subseteq$   $p \text{ ` } set\ zs$

**using** *set-remove1-subset* **by** (*fastforce*)

**moreover have**  $\bigwedge y. y \in p \text{ ` } set\ zs \implies p$  (*arg-min-on p* (*set zs*))  $\leq y$

**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)
proof -
  note * = arg-min-if-finite[OF ⟨finite B⟩ ⟨B ≠ {}⟩, of ⟨f ∘ g⟩]
  show ?thesis
    using assms * arg-min-inj-eq
    by (smt arg-min-if-finite(1) arg-min-least
        comp-apply finite-imageI imageE image-eqI image-is-empty inj-onD less-le)
qed

```

```

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**  $\text{treap-of } (\text{set } ys) = \langle \text{treap-of } L, m, \text{treap-of } R \rangle$   
**unfolding**  $L\text{-def } m\text{-def } R\text{-def}$  **using**  $assms$  **by**  $(\text{auto simp add: treap-of.simps } Let\text{-def})$   
**ultimately show**  $?case$  **by**  $auto$   
**qed**

**corollary**  $\text{treap-of-bst-of-list: inj-on } p (\text{set } zs) \implies$   
 $\text{map-tree fst } (\text{treap-of } (\text{set } (\text{map } (\lambda x. (x, p x)) zs))) = \text{bst-of-list } (\text{sort-key } p zs)$   
**using**  $\text{treap-of-bst-of-list'}$  **by**  $blast$

**corollary**  $\text{treap-of-bst-of-list'': inj-on } p (\text{set } zs) \implies$   
 $\text{map-tree fst } (\text{treap-of } ((\lambda x. (x, p x)) ` \text{set } zs)) = \text{bst-of-list } (\text{sort-key } p zs)$   
**using**  $\text{treap-of-bst-of-list}$  **by**  $auto$

**corollary**  $\text{fold-ins-bst-of-list: distinct } zs \implies \text{inj-on } p (\text{set } zs) \implies$   
 $\text{map-tree fst } (\text{foldl } (\lambda t (x,p). \text{ins } x p t) \langle \rangle (\text{map } (\lambda x. (x, p x)) zs)) = \text{bst-of-list } (\text{sort-key } p zs)$   
**by**  $(\text{auto simp add: foldl-ins-treap-of distinct-map inj-on-def inj-on-convol-ident } \text{treap-of-bst-of-list''})$

**end**

## 5 Random treaps

**theory**  $Random\text{-Treap}$   
**imports**  
 $Probability\text{-Misc}$   
 $Treap\text{-Sort-and-BSTs}$   
**begin**

### 5.1 Measurability

The following lemmas are only relevant for measurability.

**lemma**  $\text{tree-sigma-cong:}$   
**assumes**  $\text{sets } M = \text{sets } M'$   
**shows**  $\text{tree-sigma } M = \text{tree-sigma } M'$   
**using**  $\text{sets-eq-imp-space-eq}[OF assms]$  **using**  $assms$  **by**  $(\text{simp add: tree-sigma-def})$

**lemma**  $\text{distr-restrict:}$   
**assumes**  $\text{sets } N = \text{sets } L \text{ sets } K \subseteq \text{sets } M$   
 $\bigwedge X. X \in \text{sets } K \implies \text{emeasure } M X = \text{emeasure } K X$   
 $\bigwedge X. X \in \text{sets } M \implies X \subseteq \text{space } M - \text{space } K \implies \text{emeasure } M X = 0$   
 $f \in M \rightarrow_M N f \in K \rightarrow_M L$   
**shows**  $\text{distr } M N f = \text{distr } K L f$   
**proof**  $(\text{rule measure-eqI})$   
**fix**  $X$  **assume**  $X \in \text{sets } (\text{distr } M N f)$   
**thus**  $\text{emeasure } (\text{distr } M N f) X = \text{emeasure } (\text{distr } K L f) X$



**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**  $\text{sets } (\text{tree-sigma } (\text{count-space } B)) = \text{Pow } (\text{trees } B)$   
**proof** (*intro equalityI subsetI*)  
**fix** *X* **assume** *X: X ∈ Pow (trees B)*  
**have**  $\{t\} \in \text{sets } (\text{tree-sigma } (\text{count-space } B))$  **if** *t ∈ trees B* **for** *t*  
**using** *that*  
**proof** (*induction t*)  
**case** (*2 l r x*)  
**hence**  $\{\langle la, v, ra \rangle \mid la \ v \ ra. (v, la, ra) \in \{x\} \times \{l\} \times \{r\}\}$   
 $\in \text{sets } (\text{tree-sigma } (\text{count-space } B))$   
**by** (*intro Node-in-tree-sigma pair-measureI*) *auto*  
**thus** *?case* **by** *simp*  
**qed** *simp-all*  
**with** *X* **have**  $(\bigcup t \in X. \{t\}) \in \text{sets } (\text{tree-sigma } (\text{count-space } B))$   
**by** (*intro sets.countable-UN' countable-subset[OF - countable-trees[OF assms]]*)  
*auto*  
**also** **have**  $(\bigcup t \in X. \{t\}) = X$  **by** *blast*  
**finally** **show**  $X \in \text{sets } (\text{tree-sigma } (\text{count-space } B))$  .  
**next**  
**fix** *X* **assume**  $X \in \text{sets } (\text{tree-sigma } (\text{count-space } B))$   
**from** *sets.sets-into-space[OF this]* **show**  $X \in \text{Pow } (\text{trees } B)$   
**by** (*simp add: space-tree-sigma*)  
**qed**

**lemma** *height-primrec*:  $\text{height} = \text{rec-tree } 0 \ (\lambda - - a \ b. \text{Suc } (\text{max } a \ b))$

**proof**  
**fix** *t :: 'a tree*  
**show**  $\text{height } t = \text{rec-tree } 0 \ (\lambda - - a \ b. \text{Suc } (\text{max } a \ b)) \ t$   
**by** (*induction t*) *simp-all*  
**qed**

**lemma** *ipl-primrec*:  $\text{ipl} = \text{rec-tree } 0 \ (\lambda l - r \ a \ b. \text{size } l + \text{size } r + a + b)$

**proof**  
**fix** *t :: 'a tree*  
**show**  $\text{ipl } t = \text{rec-tree } 0 \ (\lambda l - r \ a \ b. \text{size } l + \text{size } r + a + b) \ t$   
**by** (*induction t*) *auto*  
**qed**

**lemma** *size-primrec*:  $\text{size} = \text{rec-tree } 0 \ (\lambda - - a \ b. 1 + a + b)$

**proof**  
**fix** *t :: 'a tree*  
**show**  $\text{size } t = \text{rec-tree } 0 \ (\lambda - - a \ b. 1 + a + b) \ t$   
**by** (*induction t*) *auto*  
**qed**

**lemma** *ipl-map-tree[simp]*:  $ipl (\text{map-tree } f \ t) = ipl \ t$   
**by** (*induction t*) *auto*

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

**lemma** *trees-mono*:  $A \subseteq B \implies \text{trees } A \subseteq \text{trees } B$

**proof**  
**fix**  $t$   
**assume**  $A \subseteq B \ t \in \text{trees } A$   
**then show**  $t \in \text{trees } B$   
**by** (*induction t*) *auto*  
**qed**

**lemma** *ins-primrec*:

$ins \ k \ (p::\text{real}) \ t = \text{rec-tree}$   
 $(\text{Node } \text{Leaf } (k,p) \ \text{Leaf})$   
 $(\lambda l \ z \ r \ l' \ r'. \text{case } z \ \text{of } (k1, p1) \Rightarrow$   
 $\text{if } k < k1 \ \text{then}$   
 $\text{(case } l' \ \text{of}$   
 $\text{Leaf } \Rightarrow \text{Leaf}$   
 $| \text{Node } l2 \ (k2,p2) \ r2 \Rightarrow$   
 $\text{if } 0 \leq p2 - p1 \ \text{then } \text{Node } (\text{Node } l2 \ (k2,p2) \ r2) \ (k1,p1) \ r$   
 $\text{else } \text{Node } l2 \ (k2,p2) \ (\text{Node } r2 \ (k1,p1) \ r))$   
 $\text{else if } k > k1 \ \text{then}$   
 $\text{(case } r' \ \text{of}$   
 $\text{Leaf } \Rightarrow \text{Leaf}$   
 $| \text{Node } l2 \ (k2,p2) \ r2 \Rightarrow$   
 $\text{if } 0 \leq p2 - p1 \ \text{then } \text{Node } l \ (k1,p1) \ (\text{Node } l2 \ (k2,p2) \ r2)$   
 $\text{else } \text{Node } (\text{Node } l \ (k1,p1) \ l2) \ (k2,p2) \ r2)$   
 $\text{else } \text{Node } l \ (k1,p1) \ r$   
 $) \ t$

**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*]:  $a \in B \rightarrow_M \text{count-space } A$   
**assumes** [*measurable*]:  $b \in B \rightarrow_M \text{count-space } A$   
**shows**  $\text{Measurable.pred } B \ (\lambda x. a \ x < b \ x)$   
**proof** –  
**have**  $\text{Measurable.pred } (\text{count-space } (A \times A)) \ (\lambda x. \text{fst } x < \text{snd } x)$  **by** *simp*  
**also have**  $\text{count-space } (A \times A) = \text{count-space } A \otimes_M \text{count-space } A$

**using** *assms*(1) **by** (*simp add: pair-measure-countable*)  
**finally have** *Measurable.pred* B (( $\lambda x. \text{fst } x < \text{snd } x$ )  $\circ$  ( $\lambda x. (a \ x, b \ x)$ ))  
**by** *measurable*  
**thus** *?thesis* **by** (*simp add: o-def*)  
**qed**

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

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

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

**declare** *U-def*[*simp*]

**fun** *insR*:: 'a::linorder  $\Rightarrow$  ('a  $\times$  real) tree  $\Rightarrow$  'a set  $\Rightarrow$  ('a  $\times$  real) tree measure  
**where**  
*insR* x t A = *distr* (U 0 1) (*tree-sigma* (*count-space* A  $\otimes_M$  *lborel*)) ( $\lambda p. \text{ins } x \ p \ t$ )

**fun** *rinss* :: 'a::linorder list  $\Rightarrow$  ('a  $\times$  real) tree  $\Rightarrow$  'a set  $\Rightarrow$  ('a  $\times$  real) tree measure  
**where**  
*rinss* [] t A = *return* (*tree-sigma* (*count-space* A  $\otimes_M$  *lborel*)) t |  
*rinss* (x#xs) t A = *insR* x t A  $\gg$  ( $\lambda t. \text{rinss } xs \ t \ A$ )

**lemma** *sets-rinss'*:  
**assumes** *countable* B *set* ys  $\subseteq$  B  
**shows**  $t \in \text{trees } (B \times \text{UNIV}) \Longrightarrow \text{sets } (\text{rinss } ys \ t \ B) = \text{sets } (\text{tree-sigma } (\text{count-space } B \otimes_M \text{lborel}))$   
**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**  $f \in A \rightarrow_M B$  *set* xs  $\subseteq$  *space* C  
**assumes**  $\bigwedge c. c \in \text{set } xs \Longrightarrow (\lambda(a,b). g \ a \ b \ c) \in (A \otimes_M B) \rightarrow_M B$   
**shows** ( $\lambda x. \text{foldl } (g \ x) (f \ x) \ xs$ )  $\in A \rightarrow_M B$   
**using** *assms*  
**proof** (*induction xs arbitrary: f*)  
**case** *Nil*

```

thus ?case by simp
next
  case (Cons x xs)
  note [measurable] = Cons.premis(1)
  from Cons.premis have [measurable]:  $x \in \text{space } C$  by simp
  have  $(\lambda a. (a, f a)) \in A \rightarrow_M A \otimes_M B$ 
    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.premis, auto)
  hence  $(\lambda a. g a (f a) x) \in A \rightarrow_M B$  by (simp add: o-def)
  hence  $(\lambda xa. \text{foldl } (g xa) (g xa (f xa) x) xs) \in A \rightarrow_M B$ 
    by (rule Cons.IH) (use Cons.premis in auto)
  thus ?case by simp
qed

```

**lemma** *ins-trees*:  $t \in \text{trees } A \implies (x,y) \in A \implies \text{ins } x y t \in \text{trees } A$   
**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  $xs::'a::\text{linorder list}$  and  $A::'a \text{ set}$  and  $\text{random-perm} :: 'a \text{ list} \Rightarrow 'a \text{ list}$ 
  measure
  assumes con-assms:  $\text{countable } A$   $\text{set } xs \subseteq A$  distinct  $xs$ 
  defines  $\text{random-perm} \equiv (\lambda xs. \text{uniform-measure } (\text{count-space } (\text{permutations-of-set } (\text{set } xs))))$ 
  (permutations-of-set (set xs))

```

**begin**

Again, we first need some facts about measurability.

```

lemma sets-rinss [simp]:
  assumes  $t \in \text{trees } (A \times \text{UNIV})$ 
  shows  $\text{sets } (\text{rinss } xs t A) = \text{tree-sigma } (\text{count-space } A \otimes_M \text{borel})$ 
proof –
  have  $\text{tree-sigma } (\text{count-space } A \otimes_M (\text{lborel}::\text{real measure})) = \text{tree-sigma } (\text{count-space } A \otimes_M \text{borel})$ 
    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*]:

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

**lemma** *insort-wrt-measurable* [*measurable*]:  
 $(\lambda x. \text{insort-wrt } x \text{ } xs) \in \text{count-space } (\text{Pow } (A \times A)) \rightarrow_M \text{count-space } (\text{lists } A)$   
**using** *con-assms by auto*

**lemma** *bst-of-list-sort-measurable* [*measurable*]:  
 $(\lambda x. \text{bst-of-list } (\text{sort-key } x \text{ } xs)) \in$   
 $Pi_M (\text{set } xs) (\lambda i. \text{borel}::\text{real measure}) \rightarrow_M \text{tree-sigma } (\text{count-space } A)$

**proof** –

**note** *measurable-linorder-from-keys-restrict'* [*measurable*]

**have**  $(0::\text{real}) < 1$

**by** *auto*

**then have** [*measurable*]:  $(\lambda x. \text{bst-of-list } (\text{insort-wrt } (\text{linorder-from-keys } (\text{set } xs)$   
 $x) \text{ } xs))$

$\in Pi_M (\text{set } xs) (\lambda i. \text{borel} :: \text{real measure}) \rightarrow_M \text{tree-sigma}$

$(\text{count-space } A)$

**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**  $t \in \text{space } (\text{tree-sigma } (\text{count-space } A \otimes_M \text{lborel}))$

**shows**  $\text{rinss } xs \ t \ A = \text{distr } (\Pi_M x \in \text{set } xs. \mathcal{U} \ 0 \ 1)$   
 $(\text{tree-sigma } (\text{count-space } A \otimes_M \text{lborel}))$   
 $(\lambda p. \text{foldl } (\lambda t \ x. \text{ins } x \ (p \ x) \ t) \ t \ xs)$

**proof** –

**let**  $?U = \text{uniform-measure } \text{lborel } \{0::\text{real}..1\}$

**have**  $\text{set } xs \subseteq \text{space } (\text{count-space } A) \ c \in \text{set } xs \implies c \in \text{space } (\text{count-space } A)$

**for**  $c$

**using** *con-assms by auto*

**then have**  $*[\text{intro}]$ :  $(\lambda p. \text{foldl } (\lambda t \ x. \text{ins } x \ (p \ x) \ t) \ t \ xs) \in$

$Pi_M (\text{set } xs) (\lambda x. ?U) \rightarrow_M \text{tree-sigma } (\text{count-space } A \otimes_M \text{lborel})$

**if**  $t \in \text{space } (\text{tree-sigma } (\text{count-space } A \otimes_M \text{lborel}))$  **for**  $t$

**using** *that con-assms by measurable*

**have** *insR'*:

$\text{insR } x \ t \ A = ?U \gg (\lambda u. \text{return } (\text{tree-sigma } (\text{count-space } A \otimes_M \text{lborel})) (\text{ins}$   
 $x \ u \ t))$

**if**  $x \in A \ t \in \text{space } (\text{tree-sigma } (\text{count-space } A \otimes_M \text{lborel}))$  **for**  $t \ x$

**using** *con-assms assms that by (auto simp add: bind-return-distr' U-def)*

**have**  $\text{rinss } xs \ t \ A = (\Pi_M x \in \text{set } xs. ?U) \gg$

$(\lambda p. \text{return } (\text{tree-sigma } (\text{count-space } A \otimes_M \text{lborel})) (\text{foldl } (\lambda t \ x. \text{ins } x \ (p \ x)$   
 $t) \ t \ xs))$

```

    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  $\otimes_M$  lborel)
have [measurable]: set xs  $\subseteq$  space (count-space A) x  $\in$  A
    c  $\in$  A  $\implies$  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) ( $\lambda x. ?U$ )  $\in$  space (prob-algebra (Pi_M (set xs)
( $\lambda i. ?U$ )))
unfolding space-prob-algebra by (auto intro!: prob-space-uniform-measure prob-space-PiM)
have [measurable]: Pi_M (set xs) ( $\lambda x. ?U$ )  $\in$  space (subprob-algebra (Pi_M (set xs)
( $\lambda i. ?U$ )))
unfolding space-subprob-algebra
by (auto intro!: prob-space-imp-subprob-space prob-space-uniform-measure prob-space-PiM)
have [measurable]: ( $\lambda x. x$ )  $\in$  (?treap-sigma  $\otimes_M$  Pi_M (set xs) ( $\lambda i. ?U$ ))  $\otimes_M$ 
?treap-sigma  $\rightarrow_M$ 
    (?treap-sigma  $\otimes_M$  Pi_M (set xs) ( $\lambda i. \text{borel}$ ))  $\otimes_M$  ?treap-sigma
by (auto intro!: measurable-ident-sets sets-pair-measure-cong sets-PiM-cong simp
add: U-def)
have [simp]: ( $\lambda w. \text{Pi}_M$  (set xs) ( $\lambda x. ?U$ ))  $\gg$ 
    ( $\lambda p. \text{return } ?treap\text{-sigma (foldl } (\lambda t x. \text{ins } x (p x) t) w xs)$ )
     $\in$  ?treap-sigma  $\rightarrow_M$  subprob-algebra ?treap-sigma
proof -
  have [measurable]: c  $\in$  set xs  $\implies$  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]: ( $\lambda t. \text{rinss } xs t A$ )  $\in$  ?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]: ( $\lambda y. x$ )  $\in$  tree-sigma (count-space A  $\otimes_M$  lborel)  $\otimes_M$  ?U
 $\rightarrow_M$  count-space A
using Cons by measurable
have [measurable]: ( $\lambda x. x$ )  $\in$  ?treap-sigma  $\otimes_M$  ?U  $\rightarrow_M$  ?treap-sigma  $\otimes_M$ 

```

```

borel
  unfolding  $\mathcal{U}$ -def by auto
  have [measurable]:  $(\lambda t. \text{distr } (?U) (\text{tree-sigma } (\text{count-space } A \otimes_M \text{lborel}))) (\lambda p. \text{ins } x \ p \ t)$ 
     $\in ?\text{treap-sigma} \rightarrow_M \text{subprob-algebra } ?\text{treap-sigma}$ 
  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]:  $(\lambda u. \text{return } ?\text{treap-sigma } (\text{ins } x \ u \ t)) \in ?U \rightarrow_M \text{subprob-algebra } ?\text{treap-sigma}$ 
  using con-assms Cons by measurable
  have [simp]:  $\text{space } (?U \otimes_M \text{Pi}_M (\text{set } xs) (\lambda x. ?U)) \neq \{\}$ 
  by (simp add: prob-space.not-empty prob-space-PiM prob-space-pair prob-space-uniform-measure)
  from Cons have rinss  $(x \# \ xs) \ t \ A = (?U \ggg (\lambda u. \text{return } ?\text{treap-sigma } (\text{ins } x \ u \ t))) \ggg (\lambda t. \text{rinss } xs \ t \ A)$ 
  by (simp add: insR')
  also have  $\dots = ?U \ggg (\lambda u. \text{return } ?\text{treap-sigma } (\text{ins } x \ u \ t)) \ggg (\lambda t. \text{rinss } xs \ t \ A)$ 
  using con-assms Cons by (subst bind-assoc) auto
  also have  $\dots = ?U \ggg (\lambda u. \text{rinss } xs \ (\text{ins } x \ u \ t) \ A)$ 
  using con-assms Cons by (subst bind-return) auto
  also have  $\dots = ?U \ggg (\lambda u. \text{Pi}_M (\text{set } xs) (\lambda x. ?U) \ggg (\lambda p. \text{return } ?\text{treap-sigma } (\text{foldl } (\lambda t \ x. \text{ins } x \ (p \ x) \ t) (\text{ins } x \ u \ t) \ xs)))$ 
  using Cons by (subst Cons) (auto simp add: treap-ins keys-ins)
  also have  $\dots = ?U \otimes_M \text{Pi}_M (\text{set } xs) (\lambda x. ?U) \ggg (\lambda (u,p). \text{return } ?\text{treap-sigma } (\text{foldl } (\lambda t \ x. \text{ins } x \ (p \ x) \ t) (\text{ins } x \ u \ t) \ xs))$ 
  proof -
  have [measurable]:  $\text{pair-prob-space } (?U) (\text{Pi}_M (\text{set } xs) (\lambda x. ?U))$ 
  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]:  $c \in \text{set } xs \implies c \in A$  for  $c$ 
  using Cons by auto
  show ?thesis
  using con-assms Cons by (subst pair-prob-space.pair-measure-bind) measurable
qed
  also have  $\dots = \text{distr } (?U \otimes_M \text{Pi}_M (\text{set } xs) (\lambda x. ?U)) (\text{tree-sigma } (\text{count-space } A \otimes_M \text{lborel})) (\lambda (u, f). \text{foldl } (\lambda t \ x. \text{ins } x \ (f \ x) \ t) (\text{ins } x \ u \ t) \ xs)$ 
  proof -
  have [simp]:  $c \in \text{set } xs \implies c \in A$  for  $c$ 
  using Cons by auto
  have  $(\lambda xa. \text{foldl } (\lambda t \ x. \text{ins } x \ (\text{snd } xa \ x) \ t) (\text{ins } x \ (\text{fst } xa) \ t) \ xs) = (\lambda (u, f). \text{foldl } (\lambda t \ x. \text{ins } x \ (f \ x) \ t) (\text{ins } x \ u \ t) \ xs)$ 

```

**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**  
 $\dots = \text{distr } (?U \otimes_M Pi_M (\text{set } xs) (\lambda i. ?U)) ?treap\text{-sigma}$   
 $(\lambda f. \text{foldl } (\lambda t y. \text{ins } y \text{ (if } y = x \text{ then fst } f \text{ else snd } f y) t) (\text{ins } x \text{ (fst } f) t) xs)$   
**proof** –  
**have**  $\text{foldl } (\lambda t y. \text{ins } y \text{ (snd } f y) t) (\text{ins } x \text{ (fst } f) t) xs =$   
 $\text{foldl } (\lambda t y. \text{ins } y \text{ (if } y = x \text{ then fst } f \text{ else snd } f y) t) (\text{ins } x \text{ (fst } f) t) xs$  **for** *f*  
**using** *Cons* **by** (*intro foldl-cong*) *auto*  
**then show** *?thesis*  
**by** (*auto simp add: case-prod-beta'*)  
**qed**  
**also have**  $\dots = \text{distr } (?U \otimes_M Pi_M (\text{set } xs) (\lambda i. ?U)) (Pi_M (\text{insert } x (\text{set } xs))$   
 $(\lambda i. ?U))$   
 $(\lambda(r, f). f(x := r)) \gg=$   
 $(\lambda p. \text{return } ?treap\text{-sigma } (\text{foldl } (\lambda t x. \text{ins } x (p x) t) (\text{ins } x (p$   
 $x) t) xs))$   
**using** *con-assms Cons*  
**by** (*subst bind-distr-return*) (*measurable, auto simp add: case-prod-beta'*)  
**also have**  $\dots = Pi_M (\text{insert } x (\text{set } xs)) (\lambda x. ?U) \gg=$   
 $(\lambda p. \text{return } ?treap\text{-sigma } (\text{foldl } (\lambda t x. \text{ins } x (p x) t) (\text{ins } x (p x) t)$   
 $xs))$   
**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**  $\text{rinss } xs \text{ Leaf } A =$   
 $\text{distr } (\prod_M x \in \text{set } xs. \mathcal{U} 0 1)$   
 $(\text{tree-sigma } (\text{count-space } A \otimes_M \text{lborel}))$   
 $(\lambda p. \text{foldl } (\lambda t x. \text{ins } x (p x) t) \text{ Leaf } xs)$   
**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*:

$\text{distr } (\text{rinss } xs \text{ Leaf } A) (\text{tree-sigma } (\text{count-space } A)) (\text{map-tree } \text{fst}) =$   
 $\text{distr } (Pi_M (\text{set } xs) (\lambda x. \mathcal{U} 0 1)) (\text{tree-sigma } (\text{count-space } A))$   
 $(\lambda p. \text{bst-of-list } (\text{sort-key } p xs))$  (**is** *?lhs = ?rhs*)

**proof** –

**have** [*measurable*]:  $\text{set } xs \subseteq \text{space } (\text{count-space } A)$



```

  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 insert-wrt R xs = insert-wrt R (remdups xs) for R
  using con-assms distinct-remdups-id by metis
  then have *: insert-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. insert-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 (insert-wrt (linorder-from-keys (set xs) p) xs))

```

**unfolding** *Let-def* **by** (*simp add: insort-wrt-sort-key*)  
**also have** ... =  
*distr (distr (Pi<sub>M</sub> (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 permuta-
tions-of-setD)
  finally show ?thesis ..
qed

```

We put everything together and obtain our main result:

**theorem** *rins-random-bst*:

```

    distr (rins xs ⟨⟩ A) (tree-sigma (count-space A)) (map-tree fst) =
      restrict-space (measure-pmf (random-bst (set xs))) (trees A)
  by (simp only: rins-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.