Treaps

Max Haslbeck, Manuel Eberl, Tobias Nipkow

March 17, 2025

Abstract

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

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

Contents

1 Auxiliary material

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

lemma *measure-eqI-countable-AE′*:
  assumes [*simp*]: *sets M* = *Pow B sets N* = *Pow B* **and** *subset*: $\Omega \subseteq B$
  assumes *ae*: *AE x in M*. $x \in \Omega$ *AE x in N*. $x \in \Omega$ **and** [*simp*]: *countable* Ω
  assumes *eq*: $\bigwedge x.\ x \in \Omega \Longrightarrow$ *emeasure M* $\{x\}$ = *emeasure N* $\{x\}$
  shows $M = N$
⟨*proof*⟩

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

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

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

lemma *countable-null-set*:
  assumes *countable A A* ⊆ *space M*
  shows *A* ∈ *null-sets M*
⟨*proof*⟩

lemma *finite-null-set*:
  assumes *finite A A* ⊆ *space M*
  shows *A* ∈ *null-sets M*
  ⟨*proof*⟩

end

lemma *measurable-inj-on-finite*:
  assumes *fin* [*measurable*]: *finite I*
  assumes [*measurable*]: $\bigwedge i\ j.$ *Measurable.pred* $(M\ i \bigotimes_M M\ j)$ (λ(*x*,*y*). *x* = *y*)
  shows *Measurable.pred* $(Pi_M\ I\ M)$ (λ*x*. *inj-on x I*) ⟨*proof*⟩

lemma *almost-everywhere-not-in-countable-set*:
  assumes *countable A*

assumes [*measurable*]: *Measurable.pred* $(M \bigotimes_M M)$ $(\lambda(x,y).\ x = y)$
assumes *null*: $\bigwedge x.\ x \in$ *space* $M \Longrightarrow \{x\} \in$ *null-sets* M
shows *AE x in M*. $x \notin A$
⟨*proof*⟩

lemma *almost-everywhere-inj-on-PiM*:
assumes *fin*: *finite* I **and** *prob-space*: $\bigwedge i.\ i \in I \Longrightarrow$ *prob-space* $(M\ i)$
assumes [*measurable*]: $\bigwedge i\ j.$ *Measurable.pred* $(M\ i \bigotimes_M M\ j)$ $(\lambda(x,y).\ x = y)$
assumes *null*: $\bigwedge i\ x.\ i \in I \Longrightarrow x \in$ *space* $(M\ i) \Longrightarrow \{x\} \in$ *null-sets* $(M\ i)$
shows *AE f in* $(\Pi_M\ i{\in}I.\ M\ i)$. *inj-on f I*
⟨*proof*⟩

lemma *null-sets-uniform-measure*:
assumes $A \in$ *sets M emeasure M* $A \neq \infty$
shows *null-sets* (*uniform-measure M A*) $= (\lambda B.\ A \cap B)$ −' *null-sets M* $\cap$ *sets M*
⟨*proof*⟩

lemma *almost-everywhere-avoid-finite*:
assumes *fin*: *finite I*
shows *AE f in* $(\Pi_M\ i{\in}I.$ *uniform-measure lborel* $\{(0::real)..1\})$. *inj-on f I*
⟨*proof*⟩

lemma *almost-everywhere-avoid-countable*:
assumes *countable A*
shows *AE x in uniform-measure lborel* $\{(0::real)..1\}$. $x \notin A$
⟨*proof*⟩

lemma *measure-pmf-of-set*:
assumes $A \neq \{\}$ **and** *finite A*
shows *measure-pmf* (*pmf-of-set A*) = *uniform-measure* (*count-space UNIV*) *A*
⟨*proof*⟩

lemma *emeasure-distr-restrict*:
assumes $f \in M \to_M N$ $f \in M' \to_M N'$ $A \in$ *sets* N' *sets* $M' \subseteq$ *sets M sets* $N' \subseteq$ *sets N*
assumes $\bigwedge X.\ X \in$ *sets* $M' \Longrightarrow$ *emeasure M X* = *emeasure* M' *X*
assumes $\bigwedge X.\ X \in$ *sets* $M \Longrightarrow X \subseteq$ *space* M − *space* $M' \Longrightarrow$ *emeasure M X* = *0*
shows *emeasure* (*distr M N f*) *A*= *emeasure* (*distr* M' N' *f*) *A*
⟨*proof*⟩

lemma *distr-uniform-measure-count-space-inj*:
assumes *inj-on f* A' $A' \subseteq A$ *f* ' $A \subseteq B$ *finite* A'
shows *distr* (*uniform-measure* (*count-space A*) A') (*count-space B*) *f* =
uniform-measure (*count-space B*) (*f* ' A') (**is** *?lhs = ?rhs*)
⟨*proof*⟩

lemma (**in** *pair-prob-space*) *pair-measure-bind*:
assumes [*measurable*]: $f \in M1 \bigotimes_M M2 \rightarrow_M$ *subprob-algebra* N
shows $(M1 \bigotimes_M M2) \gg= f = do\ \{x \leftarrow M1;\ y \leftarrow M2;\ f\ (x,\ y)\}$
⟨*proof*⟩

lemma *count-space-singleton-conv-return*:
count-space $\{x\}$ = *return* (*count-space* $\{x\}$) x
⟨*proof*⟩

lemma *distr-count-space-singleton* [*simp*]:
$f\ x \in$ *space* $M \Longrightarrow$ *distr* (*count-space* $\{x\}$) $M\ f$ = *return* $M\ (f\ x)$
⟨*proof*⟩

lemma *uniform-measure-count-space-singleton* [*simp*]:
assumes $\{x\} \in$ *sets* M *emeasure* $M\ \{x\} \neq 0$ *emeasure* $M\ \{x\} < \infty$
shows *uniform-measure* $M\ \{x\}$ = *return* $M\ x$
⟨*proof*⟩

lemma *PiM-uniform-measure-permute*:
fixes $a\ b$ :: *real*
assumes g *permutes* $A\ a < b$
shows *distr* (*PiM* A (λ-. *uniform-measure lborel* $\{a..b\}$)) (*PiM* A (λ-. *lborel*)) ($\lambda f.\ f \circ g$) =
PiM A (λ-. *uniform-measure lborel* $\{a..b\}$)
⟨*proof*⟩

lemma *ennreal-fact* [*simp*]: *ennreal* (*fact* n) = *fact* n
⟨*proof*⟩

lemma *inverse-ennreal-unique*:
assumes $a * (b$ :: *ennreal*) = *1*
shows b = *inverse* a
⟨*proof*⟩

end

2 Treaps

theory *Treap*
imports
HOL−Library.Tree
begin

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

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

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

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*⟩

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*
⟨*proof*⟩

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* ≠ ⟨⟩
  ⟨*proof*⟩

lemma *keys-ins*: *keys* (*ins k p t*) = *Set.insert k* (*keys t*)
⟨*proof*⟩

lemma *prios-ins*: *prios* (*ins k p t*) ⊆ {*p*} ∪ *prios t*
⟨*proof*⟩

lemma *prios-ins'*: *k* ∉ *keys t* ⟹ *prios* (*ins k p t*) = {*p*} ∪ *prios t*

⟨*proof*⟩

lemma *set-tree-ins*: *set-tree* (*ins* k p t) $\subseteq$ $\{(k,p)\}$ $\cup$ *set-tree* t
⟨*proof*⟩

lemma *set-tree-ins'*: $k \notin$ *keys* $t \Longrightarrow$ $\{(k,p)\}$ $\cup$ *set-tree* $t \subseteq$ *set-tree* (*ins* k p t)
⟨*proof*⟩

lemma *set-tree-ins-eq*: $k \notin$ *keys* $t \Longrightarrow$ *set-tree* (*ins* k p t) $=$ $\{(k,p)\}$ $\cup$ *set-tree* t
⟨*proof*⟩

lemma *prios-ins-special*:
⟦ *ins* k p t = *Node* l (k',p') r; $p' = p$; $p \in$ *prios* r $\cup$ *prios* l ⟧
$\Longrightarrow p \in$ *prios* t
⟨*proof*⟩

lemma *treap-NodeI*:
⟦ *treap* l; *treap* r;
$\forall k' \in$ *keys* l. $k' < k$; $\forall k' \in$ *keys* r. $k < k'$;
$\forall p' \in$ *prios* l. $p \le p'$; $\forall p' \in$ *prios* r. $p \le p'$ ⟧
$\Longrightarrow$ *treap* (*Node* l (k,p) r)
⟨*proof*⟩

lemma *treap-rotate1*:
assumes *treap l2 treap r2 treap r* $\neg$ *p1* $\le$ *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*⟩

lemma *treap-rotate2*:
assumes *treap l treap l2 treap r2* $\neg$ *p1* $\le$ *p2 k1* $<$ *k* **and**
ins: *ins k p r* = *Node l2* (*k2*,*p2*) *r2* **and** *treap-ins*: *treap* (*ins k p r*)
and *treap*: *treap* ⟨*l*, (*k1*, *p1*), *r*⟩
shows *treap* (*Node* (*Node l* (*k1*,*p1*) *l2*) (*k2*,*p2*) *r2*)
⟨*proof*⟩

lemma *treap-ins*: *treap* $t \Longrightarrow$ *treap* (*ins* k p t)
⟨*proof*⟩

lemma *treap-of-set-tree-unique*:
⟦ *finite* A; *inj-on fst* A; *inj-on snd* A ⟧
$\Longrightarrow$ *set-tree* (*treap-of* A) $= A$
⟨*proof*⟩

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

lemma *treap-treap-of*:
treap (*treap-of A*)
⟨*proof*⟩

lemma *treap-Leaf*: *treap* ⟨⟩
⟨*proof*⟩

lemma *foldl-ins-treap*: *treap t* $\Longrightarrow$ *treap* (*foldl* ($\lambda t'$ (x, p). *ins x p t'*) *t xs*)
⟨*proof*⟩

lemma *foldl-ins-set-tree*:
assumes *inj-on fst* (*set ys*) *inj-on snd* (*set ys*) *distinct ys fst* ‘ (*set ys*) ∩ *keys t* = {}
shows *set-tree* (*foldl* ($\lambda t'$ (x, p). *ins x p t'*) *t ys*) = *set ys* ∪ *set-tree t*
⟨*proof*⟩

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*)
⟨*proof*⟩

end

3 Randomly-permuted lists

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

3.1 General facts about linear orderings

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

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

lemma *linorders-on-empty* [*simp*]: *linorders-on* {} = {{}}
⟨*proof*⟩

lemma *linorders-finite-nonempty*:
assumes *finite A*
shows *linorders-on A* ≠ {}
⟨*proof*⟩

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*)
⟨*proof*⟩

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*
⟨*proof*⟩

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*⟩

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*)
⟨*proof*⟩

lemma *finite-linorders-on* [*intro*]:
assumes *finite A*
shows *finite* (*linorders-on A*)
⟨*proof*⟩

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* $\Rightarrow$ $('a \Rightarrow 'b) \Rightarrow ('b \times 'b)$ *set* $\Rightarrow$ $('a \times 'a)$ *set*
where
map-relation A f R = $\{(x,y) \in A \times A.\ (f\ x,\ f\ y) \in R\}$

lemma *index-distinct-eqI*:
assumes *distinct xs* $i <$ *length xs xs* ! $i = x$
shows *index xs* $x = i$
⟨*proof*⟩

lemma *index-permute-list*:
assumes π *permutes* {..<*length xs*} *distinct xs* $x \in$ *set xs*
shows *index* (*permute-list* π *xs*) x = *inv* π (*index xs x*)
⟨*proof*⟩

lemma *linorder-of-list-permute*:
assumes π *permutes* {..<*length xs*} *distinct xs*
shows *linorder-of-list* (*permute-list* π *xs*) =
map-relation (*set xs*) ((!) *xs* $\circ$ *inv* π $\circ$ *index xs*) (*linorder-of-list xs*)
⟨*proof*⟩

lemma *inj-on-conv-Ex1*: *inj-on f A* $\longleftrightarrow$ ($\forall\, y \in f`A.\ \exists!x \in A.\ y = f\ x$)
⟨*proof*⟩

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
⟨*proof*⟩

lemma *permutesI*:
assumes *bij-betw f A A* $\forall\, x.\ x \notin A \longrightarrow f\ x = x$
shows *f permutes A*
⟨*proof*⟩

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

lemma *linorder-permutation-exists*:
assumes *finite A linorder-on A R linorder-on A R'*
obtains π **where** π *permutes A* R' = *map-relation A* π *R*
⟨*proof*⟩

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

definition *linorder-from-keys* :: *'a set* $\Rightarrow$ (*'a* $\Rightarrow$ *'b* :: *linorder*) $\Rightarrow$ (*'a* $\times$ *'a*) *set*
where
linorder-from-keys A f = $\{(x,y) \in A \times A.\ f\ x \le f\ y\}$

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*)
⟨*proof*⟩

lemma *linorder-on-linorder-from-keys* [*intro*]:
assumes *inj-on f A*
shows *linorder-on A* (*linorder-from-keys A f*)
⟨*proof*⟩

lemma *linorder-from-keys-empty* [*simp*]: *linorder-from-keys* {} = (λ-. {})
⟨*proof*⟩

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

lemma *emeasure-PiM-diagonal*:
fixes *a b* :: *real*
assumes $x \in A$ $y \in A$ $x \neq y$
assumes $a < b$ *finite A*

defines $M \equiv$ *uniform-measure lborel* $\{a..b\}$
shows *emeasure* (*PiM A* (λ-. *M*)) $\{h{\in}A \rightarrow_E UNIV.\ h\ x = h\ y\} = 0$
⟨*proof*⟩

lemma *measurable-linorder-from-keys-restrict*:
assumes *fin*: *finite A*
shows *linorder-from-keys* $A \in Pi_M$ A (λ-. *borel* :: *real measure*) $\rightarrow_M$ *count-space* (*Pow* ($A \times A$))
(**is** - : *?M* $\rightarrow_M$ -)
⟨*proof*⟩

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*⟩

lemma *measurable-linorder-from-keys-restrict'*:
assumes *fin*: *finite A* $A \subseteq B$
shows *linorder-from-keys* $A \in Pi_M$ A (λ-. *borel* :: *real measure*) $\rightarrow_M$ *count-space* (*Pow* ($B \times B$))
⟨*proof*⟩

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* (λ-. *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 (λ-. *borel* :: *real measure*) $\rightarrow_M$ *count-space* (*Pow* ($B \times B$))
⟨*proof*⟩

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

theorem *almost-everywhere-linorder*: *AE R in M*. *linorder-on A R*
⟨*proof*⟩

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

theorem *random-linorder-by-prios*:
$M =$ *uniform-measure* (*count-space* (*Pow* ($B \times B$))) (*linorders-on A*)
⟨*proof*⟩

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*:
$insort\text{-}key\ f\ x\ xs =$
$rec\text{-}list\ [x]\ (\lambda y\ ys\ zs.\ if\ f\ x \le f\ y\ then\ x \# y \# ys\ else\ y \# zs)\ xs$
⟨*proof*⟩

lemma *insort-key-conv-rec-list'*:
$insort\text{-}key = (\lambda f\ x.$
$rec\text{-}list\ [x]\ (\lambda y\ ys\ zs.\ if\ f\ x \le f\ y\ then\ x \# y \# ys\ else\ y \# zs))$
⟨*proof*⟩

lemma *bst-of-list-trees*:
assumes $set\ ys \subseteq A$
shows $bst\text{-}of\text{-}list\ ys \in trees\ A$
⟨*proof*⟩

lemma *insort-wrt-insort-key*:
$a \in A \Longrightarrow$
$set\ xs \subseteq A \Longrightarrow$
$insert\text{-}wrt\ (linorder\text{-}from\text{-}keys\ A\ f)\ a\ xs = insort\text{-}key\ f\ a\ xs$
⟨*proof*⟩

lemma *insort-wrt-sort-key*:
assumes $set\ xs \subseteq A$
shows $insort\text{-}wrt\ (linorder\text{-}from\text{-}keys\ A\ f)\ xs = sort\text{-}key\ f\ xs$
⟨*proof*⟩

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 $zs \neq []$ $inj\text{-}on\ p\ (set\ zs)$
shows $sort\text{-}key\ p\ (zs::'a::linorder\ list) =$
$(let\ z = arg\text{-}min\text{-}on\ p\ (set\ zs)\ in\ z \# sort\text{-}key\ p\ (remove1\ z\ zs))$
⟨*proof*⟩

lemma *arg-min-on-image-finite*:
fixes f :: $'b \Rightarrow 'c$ :: *linorder*
assumes *inj-on* f (g ‘ B) *finite* B $B \neq \{\}$
shows *arg-min-on* f (g ‘ B) = g (*arg-min-on* ($f \circ g$) B)
⟨*proof*⟩

lemma *fst-snd-arg-min-on*: **fixes** p::$'a \Rightarrow 'b$::*linorder*
assumes *finite* B *inj-on* p B $B \neq \{\}$
shows *fst* (*arg-min-on snd* ((λx. (x, p x)) ‘ B)) = *arg-min-on* p B
⟨*proof*⟩

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′*
⟨*proof*⟩

corollary *treap-of-bst-of-list*: *inj-on* p (*set zs*) $\Longrightarrow$
map-tree fst (*treap-of* (*set* (*map* (λx. (x, p x)) *zs*))) = *bst-of-list* (*sort-key* p *zs*)
⟨*proof*⟩

corollary *treap-of-bst-of-list″*: *inj-on* p (*set zs*) $\Longrightarrow$
map-tree fst (*treap-of* ((λx. (x, p x)) ‘ *set zs*)) = *bst-of-list* (*sort-key* p *zs*)
⟨*proof*⟩

corollary *fold-ins-bst-of-list*: *distinct zs* $\Longrightarrow$ *inj-on* p (*set zs*) $\Longrightarrow$
map-tree fst (*foldl* (λt (x,p). *ins* x p t) ⟨⟩ (*map* (λx. (x, p x)) *zs*)) = *bst-of-list* (*sort-key* p *zs*)
⟨*proof*⟩

end

5 Random treaps

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

5.1 Measurability

The following lemmas are only relevant for measurability.

lemma *tree-sigma-cong*:
assumes *sets* M = *sets* M'
shows *tree-sigma* M = *tree-sigma* M'
⟨*proof*⟩

lemma *distr-restrict*:
  assumes *sets N = sets L sets K ⊆ sets M*
    ⋀X. X ∈ sets K ⟹ emeasure M X = emeasure K X
    ⋀X. X ∈ sets M ⟹ X ⊆ space M − space K ⟹ emeasure M X = 0
    $f \in M \to_M N$ $f \in K \to_M L$
  shows *distr M N f = distr K L f*
⟨*proof*⟩

lemma *sets-tree-sigma-count-space*:
  assumes *countable B*
  shows *sets (tree-sigma (count-space B)) = Pow (trees B)*
⟨*proof*⟩

lemma *height-primrec*: *height = rec-tree 0 (λ- - - a b. Suc (max a b))*
⟨*proof*⟩

lemma *ipl-primrec*: *ipl = rec-tree 0 (λl - r a b. size l + size r + a + b)*
⟨*proof*⟩

lemma *size-primrec*: *size = rec-tree 0 (λ- - - a b. 1 + a + b)*
⟨*proof*⟩

lemma *ipl-map-tree[simp]*: *ipl (map-tree f t) = ipl t*
⟨*proof*⟩

lemma *set-pmf-random-bst*: *finite A ⟹ set-pmf (random-bst A) ⊆ trees A*
⟨*proof*⟩

lemma *trees-mono*: *A ⊆ B ⟹ trees A ⊆ trees B*
⟨*proof*⟩

lemma *ins-primrec*:

```
ins k (p::real) t = rec-tree
  (Node Leaf (k,p) Leaf)
  (λl z r l' r'. case z of (k1, p1) ⇒
    if k < k1 then
      (case l' of
        Leaf ⇒ Leaf
      | Node l2 (k2,p2) r2 ⇒
          if 0 ≤ p2 − p1 then Node (Node l2 (k2,p2) r2) (k1,p1) r
          else Node l2 (k2,p2) (Node r2 (k1,p1) r))
    else if k > k1 then
      (case r' of
        Leaf ⇒ Leaf
      | Node l2 (k2,p2) r2 ⇒
          if 0 ≤ p2 − p1 then Node l (k1,p1) (Node l2 (k2,p2) r2)
          else Node (Node l (k1,p1) l2) (k2,p2) r2)
```

else Node l (k1,p1) r
) t
⟨*proof*⟩

lemma *measurable-less-count-space* [*measurable* (*raw*)]:
assumes *countable A*
assumes [*measurable*]: $a \in B \to_M$ *count-space A*
assumes [*measurable*]: $b \in B \to_M$ *count-space A*
shows *Measurable.pred B* ($\lambda x.\ a\ x < b\ x$)
⟨*proof*⟩

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

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

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

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

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

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

lemma *sets-rinss'*:
assumes *countable B set ys* ⊆ *B*
shows *t* ∈ *trees* (*B* × *UNIV*) ⟹ *sets* (*rinss ys t B*) = *sets* (*tree-sigma* (*count-space B* $\bigotimes_M$ *lborel*))
⟨*proof*⟩

lemma *measurable-foldl* [*measurable*]:
assumes $f \in A \to_M B$ *set xs* ⊆ *space C*
assumes $\bigwedge c.\ c \in$ *set xs* ⟹ ($\lambda(a,b).\ g\ a\ b\ c$) ∈ ($A \bigotimes_M B$) $\to_M B$
shows ($\lambda x.$ *foldl* ($g\ x$) ($f\ x$) *xs*) $\in A \to_M B$
⟨*proof*⟩

lemma *ins-trees*: $t \in trees\ A \Longrightarrow (x,y) \in A \Longrightarrow ins\ x\ y\ t \in trees\ A$
⟨*proof*⟩

5.2 Main result

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

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

context
fixes $xs::'a::linorder\ list$ **and** $A::'a\ set$ **and** $random\text{-}perm :: 'a\ list \Rightarrow 'a\ list\ measure$
assumes *con-assms*: $countable\ A\ set\ xs \subseteq A\ distinct\ xs$
defines $random\text{-}perm \equiv (\lambda xs.\ uniform\text{-}measure\ (count\text{-}space\ (permutations\text{-}of\text{-}set\ (set\ xs)))$
$(permutations\text{-}of\text{-}set\ (set\ xs)))$
begin

Again, we first need some facts about measurability.

lemma *sets-rinss* [*simp*]:
assumes $t \in trees\ (A \times UNIV)$
shows $sets\ (rinss\ xs\ t\ A) = tree\text{-}sigma\ (count\text{-}space\ A \bigotimes_M borel)$
⟨*proof*⟩

lemma *bst-of-list-measurable* [*measurable*]:
$bst\text{-}of\text{-}list \in measurable\ (count\text{-}space\ (lists\ A))\ (tree\text{-}sigma\ (count\text{-}space\ A))$
⟨*proof*⟩

lemma *insort-wrt-measurable* [*measurable*]:
$(\lambda x.\ insort\text{-}wrt\ x\ xs) \in count\text{-}space\ (Pow\ (A \times A)) \rightarrow_M count\text{-}space\ (lists\ A)$
⟨*proof*⟩

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

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

corollary *random-treap-fold-Leaf*:
shows *rinss xs Leaf A* =
distr (Π_M *x*∈*set xs*. $\mathcal{U}$ *0 1*)
(*tree-sigma* (*count-space A* $\bigotimes_M$ *lborel*))
(λ*p*. *foldl* (λ*t x*. *ins x* (*p x*) *t*) *Leaf xs*)
⟨*proof*⟩

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

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

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*) (λ*x*. $\mathcal{U}$ *0 1*)) (*tree-sigma* (*count-space A*))
(λ*p*. *bst-of-list* (*sort-key p xs*)) =
distr (*random-perm xs*) (*tree-sigma* (*count-space A*)) *bst-of-list* (**is** *?lhs* = *?rhs*)
⟨*proof*⟩

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*
⟨*proof*⟩

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*⟩

We put everything together and obtain our main result:

theorem *rinss-random-bst*:
distr (*rinss xs* ⟨⟩ *A*) (*tree-sigma* (*count-space A*)) (*map-tree fst*) =
restrict-space (*measure-pmf* (*random-bst* (*set xs*))) (*trees A*)
⟨*proof*⟩

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.