

Expected Shape of Random Binary Search Trees

Manuel Eberl

April 14, 2026

Abstract

This entry contains proofs for the textbook results about the distributions of the height and internal path length of random binary search trees (BSTs), i. e. BSTs that are formed by taking an empty BST and inserting elements from a fixed set in random order.

In particular, we prove a logarithmic upper bound on the expected height and the $\Theta(n \log n)$ closed-form solution for the expected internal path length in terms of the harmonic numbers. We also show how the internal path length relates to the average-case cost of a lookup in a BST.

Contents

1	Expected shape of random Binary Search Trees	2
1.1	Auxiliary lemmas	2
1.2	Creating a BST from a list	3
1.3	Random BSTs	4
1.4	Expected height	8
1.5	Lookup costs	15
1.6	Average Path Length	16

1 Expected shape of random Binary Search Trees

theory *Random-BSTs*

imports

Complex-Main

HOL-Probability.Random-Permutations

HOL-Data-Structures.Tree-Set

Quick-Sort-Cost.Quick-Sort-Average-Case

begin

hide-const (**open**) *Tree-Set.insert*

1.1 Auxiliary lemmas

lemma *linorder-on-linorder-class* [intro]:

linorder-on UNIV $\{(x, y). x \leq (y :: 'a :: linorder)\}$

by (*auto simp: linorder-on-def refl-on-def antisym-def trans-def total-on-def*)

lemma *Nil-in-permutations-of-set-iff* [simp]: $[\] \in \text{permutations-of-set } A \longleftrightarrow A = \{\}$

by (*auto simp: permutations-of-set-def*)

lemma *max-power-distrib-right*:

fixes $a :: 'a :: linordered-semidom$

shows $a > 1 \implies \max (a \hat{\ } b) (a \hat{\ } c) = a \hat{\ } \max b c$

by (*auto simp: max-def*)

lemma *set-tree-empty-iff* [simp]: *set-tree* $t = \{\}$ $\longleftrightarrow t = \text{Leaf}$

by (*cases t*) *auto*

lemma *card-set-tree-bst*: *bst* $t \implies \text{card } (\text{set-tree } t) = \text{size } t$

proof (*induction t*)

case (*Node l x r*)

have *set-tree* $\langle l, x, r \rangle = \text{insert } x (\text{set-tree } l \cup \text{set-tree } r)$ **by** *simp*

also from *Node.premis* **have** $\text{card } \dots = \text{Suc } (\text{card } (\text{set-tree } l \cup \text{set-tree } r))$

by (*intro card-insert-disjoint*) *auto*

also from *Node* **have** $\text{card } (\text{set-tree } l \cup \text{set-tree } r) = \text{size } l + \text{size } r$

by (*subst card-Un-disjoint*) *force+*

finally show *?case* **by** *simp*

qed *simp-all*

lemma *pair-pmf-cong*:

$p = p' \implies q = q' \implies \text{pair-pmf } p \ q = \text{pair-pmf } p' \ q'$

by *simp*

lemma *expectation-add-pair-pmf*:

fixes $f :: 'a \Rightarrow 'c :: \{\text{banach, second-countable-topology}\}$

assumes *finite* (*set-pmf* p) **and** *finite* (*set-pmf* q)

shows *measure-pmf.expectation* (*pair-pmf* $p \ q$) $(\lambda(x,y). f \ x + g \ y) =$

$measure\text{-}pmf.expectation\ p\ f + measure\text{-}pmf.expectation\ q\ g$

proof –

have $measure\text{-}pmf.expectation\ (pair\text{-}pmf\ p\ q)\ (\lambda(x,y). f\ x + g\ y) =$
 $measure\text{-}pmf.expectation\ (pair\text{-}pmf\ p\ q)\ (\lambda z. f\ (fst\ z) + g\ (snd\ z))$
by (*simp add: case-prod-unfold*)

also have $\dots = measure\text{-}pmf.expectation\ (pair\text{-}pmf\ p\ q)\ (\lambda z. f\ (fst\ z)) +$
 $measure\text{-}pmf.expectation\ (pair\text{-}pmf\ p\ q)\ (\lambda z. g\ (snd\ z))$

by (*intro Bochner-Integration.integral-add integrable-measure-pmf-finite*) (*auto intro: assms*)

also have $measure\text{-}pmf.expectation\ (pair\text{-}pmf\ p\ q)\ (\lambda z. f\ (fst\ z)) =$
 $measure\text{-}pmf.expectation\ (map\text{-}pmf\ fst\ (pair\text{-}pmf\ p\ q))\ f$ **by** *simp*

also have $map\text{-}pmf\ fst\ (pair\text{-}pmf\ p\ q) = p$ **by** (*rule map-fst-pair-pmf*)

also have $measure\text{-}pmf.expectation\ (pair\text{-}pmf\ p\ q)\ (\lambda z. g\ (snd\ z)) =$
 $measure\text{-}pmf.expectation\ (map\text{-}pmf\ snd\ (pair\text{-}pmf\ p\ q))\ g$ **by** *simp*

also have $map\text{-}pmf\ snd\ (pair\text{-}pmf\ p\ q) = q$ **by** (*rule map-snd-pair-pmf*)

finally show *?thesis* .

qed

1.2 Creating a BST from a list

The following recursive function creates a binary search tree from a given list of elements by inserting them into an initially empty BST from left to right. We will prove that this is the case later, but the recursive definition has the advantage of giving us a useful induction rule, so we chose that definition and prove the alternative definitions later.

This recursion, which already almost looks like QuickSort, will be key in analysing the shape distributions of random BSTs.

fun *bst-of-list* :: 'a :: *linorder* list \Rightarrow 'a tree **where**
bst-of-list [] = *Leaf*
| *bst-of-list* (x # xs) =
Node (*bst-of-list* [y ← xs. y < x]) x (*bst-of-list* [y ← xs. y > x])

lemma *bst-of-list-eq-Leaf-iff* [*simp*]: *bst-of-list* xs = *Leaf* \longleftrightarrow xs = []
by (*induction xs*) *auto*

lemma *bst-of-list-snoc* [*simp*]:
bst-of-list (xs @ [y]) = *Tree-Set.insert* y (*bst-of-list* xs)
by (*induction xs rule: bst-of-list.induct*) *auto*

lemma *bst-of-list-append*:
bst-of-list (xs @ ys) = *fold Tree-Set.insert* ys (*bst-of-list* xs)

proof (*induction ys arbitrary: xs*)
case (*Cons* y ys)
have *bst-of-list* (xs @ (y # ys)) = *bst-of-list* ((xs @ [y]) @ ys) **by** *simp*
also have $\dots = fold\ Tree\text{-}Set.insert\ ys\ (bst\text{-}of\text{-}list\ (xs\ @\ [y]))$
by (*rule Cons.IH*)
finally show *?case* **by** *simp*

qed *simp-all*

The following now shows that the recursive function indeed corresponds to the notion of inserting the elements from the list from left to right.

lemma *bst-of-list-altdef*: $\text{bst-of-list } xs = \text{fold } \text{Tree-Set.insert } xs \text{ Leaf}$
using *bst-of-list-append*[of [] *xs*] **by** *simp*

lemma *size-bst-insert*: $x \notin \text{set-tree } t \implies \text{size } (\text{Tree-Set.insert } x \ t) = \text{Suc } (\text{size } t)$
by (*induction t*) *auto*

lemma *set-bst-insert* [*simp*]: $\text{set-tree } (\text{Tree-Set.insert } x \ t) = \text{insert } x \ (\text{set-tree } t)$
by (*induction t*) *auto*

lemma *set-bst-of-list* [*simp*]: $\text{set-tree } (\text{bst-of-list } xs) = \text{set } xs$
by (*induction xs rule: rev-induct*) *simp-all*

lemma *size-bst-of-list-distinct* [*simp*]:
assumes *distinct xs*
shows $\text{size } (\text{bst-of-list } xs) = \text{length } xs$
using *assms* **by** (*induction xs rule: rev-induct*) (*auto simp: size-bst-insert*)

lemma *strict-mono-on-imp-less-iff*:
assumes *strict-mono-on A f x ∈ A y ∈ A*
shows $f \ x < (f \ y :: 'b :: \text{linorder}) \iff x < (y :: 'a :: \text{linorder})$
using *assms* **by** (*cases x y rule: linorder-cases; force simp: strict-mono-on-def*)⁺

lemma *bst-of-list-map*:
fixes $f :: 'a :: \text{linorder} \Rightarrow 'b :: \text{linorder}$
assumes *strict-mono-on A f set xs ⊆ A*
shows $\text{bst-of-list } (\text{map } f \ xs) = \text{map-tree } f \ (\text{bst-of-list } xs)$
using *assms*
proof (*induction xs rule: bst-of-list.induct*)
case ($2 \ x \ xs$)
have $[\text{xa} \leftarrow \text{xs} . f \ \text{xa} < f \ x] = [\text{xa} \leftarrow \text{xs} . \text{xa} < x]$ **and** $[\text{xa} \leftarrow \text{xs} . f \ \text{xa} > f \ x] = [\text{xa} \leftarrow \text{xs} . \text{xa} > x]$
using $2.\text{prems}$ **by** (*auto simp: strict-mono-on-imp-less-iff intro!: filter-cong*)
with 2 **show** $?case$ **by** (*auto simp: filter-map o-def*)
qed *auto*

1.3 Random BSTs

Analogously to the previous section, we can now view the concept of a random BST (i. e. a BST obtained by inserting a given set of elements in random order) in two different ways.

We again start with the recursive variant:

function *random-bst* :: $'a :: \text{linorder}$ *set* $\Rightarrow 'a$ *tree pmf* **where**
random-bst A =
(if $\neg \text{finite } A \vee A = \{\}$ *then*
return-pmf Leaf
else do {

```

    x ← pmf-of-set A;
    l ← random-bst {y ∈ A. y < x};
    r ← random-bst {y ∈ A. y > x};
    return-pmf (Node l x r)
  })
  by auto
termination by (relation finite-psubset) auto

```

```

declare random-bst.simps [simp del]

```

```

lemma random-bst-empty [simp]: random-bst {} = return-pmf Leaf
  by (simp add: random-bst.simps)

```

```

lemma set-pmf-random-permutation [simp]:
  finite A ⇒ set-pmf (pmf-of-set (permutations-of-set A)) = {xs. distinct xs ∧ set
xs = A}
  by (subst set-pmf-of-set) (auto dest: permutations-of-setD)

```

The alternative characterisation is the more intuitive one where we simply pick a random permutation of the set elements uniformly at random and insert them into an empty tree from left to right:

```

lemma random-bst-altdef:
  assumes finite A
  shows random-bst A = map-pmf bst-of-list (pmf-of-set (permutations-of-set A))
using assms
proof (induction A rule: finite-psubset-induct)
  case (psubset A)
  define L R where L = (λx. {y∈A. y < x}) and R = (λx. {y∈A. y > x})
  {
    fix x assume x: x ∈ A
    hence *: L x ⊂ A R x ⊂ A by (auto simp: L-def R-def)
    note this [THEN psubset.IH]
  } note IH = this

```

```

show ?case

```

```

proof (cases A = {})

```

```

  case False

```

```

    note A = ⟨finite A⟩ ⟨A ≠ {}⟩

```

```

    have random-bst A =

```

```

      do {

```

```

        x ← pmf-of-set A;

```

```

        (l, r) ← pair-pmf (random-bst (L x)) (random-bst (R x));

```

```

        return-pmf (Node l x r)

```

```

      } using A unfolding pair-pmf-def L-def R-def

```

```

    by (subst random-bst.simps) (simp add: bind-return-pmf bind-assoc-pmf)

```

```

  also have ... = do {

```

```

    x ← pmf-of-set A;

```

```

    (l, r) ← pair-pmf

```

```

      (map-pmf bst-of-list (pmf-of-set (permutations-of-set (L x))))

```

```

      (map-pmf bst-of-list (pmf-of-set (permutations-of-set (R x))));
      return-pmf (Node l x r)
    }
  using A by (intro bind-pmf-cong refl) (simp-all add: IH)
  also have ... = do {
    x ← pmf-of-set A;
    (ls, rs) ← pair-pmf (pmf-of-set (permutations-of-set (L x)))
      (pmf-of-set (permutations-of-set (R x)));
    return-pmf (Node (bst-of-list ls) x (bst-of-list rs))
  } unfolding map-pair [symmetric]
  by (simp add: map-pmf-def case-prod-unfold bind-return-pmf bind-assoc-pmf)
  also have L = (λx. {y ∈ A - {x}. y ≤ x}) by (auto simp: L-def)
  also have R = (λx. {y ∈ A - {x}. ¬y ≤ x}) by (auto simp: R-def)
  also have do {
    x ← pmf-of-set A;
    (ls, rs) ← pair-pmf (pmf-of-set (permutations-of-set {y ∈ A - {x}.
y ≤ x}))
      (pmf-of-set (permutations-of-set {y ∈ A - {x}. ¬y
≤ x}));
    return-pmf (Node (bst-of-list ls) x (bst-of-list rs))
  } =
  do {
    x ← pmf-of-set A;
    (ls, rs) ← map-pmf (partition (λy. y ≤ x))
      (pmf-of-set (permutations-of-set (A - {x})));
    return-pmf (Node (bst-of-list ls) x (bst-of-list rs))
  } using ⟨finite A⟩
  by (intro bind-pmf-cong refl partition-random-permutations [symmetric]) auto
  also have ... = do {
    x ← pmf-of-set A;
    (ls, rs) ← map-pmf (λxs. ([y ← xs. y < x], [y ← xs. y > x]))
      (pmf-of-set (permutations-of-set (A - {x})));
    return-pmf (Node (bst-of-list ls) x (bst-of-list rs))
  } using A
  by (intro bind-pmf-cong refl map-pmf-cong)
  (auto intro!: filter-cong dest: permutations-of-setD simp: order.strict-iff-order)
  also have ... = map-pmf bst-of-list (pmf-of-set (permutations-of-set A))
  using A by (subst random-permutation-of-set[of A])
  (auto simp: map-pmf-def bind-return-pmf o-def bind-assoc-pmf not-le)
  finally show ?thesis .
qed (simp-all add: pmf-of-set-singleton)
qed

lemma finite-set-random-bst [simp, intro]:
  finite A ⇒ finite (set-pmf (random-bst A))
  by (simp add: random-bst-altdef)

lemma random-bst-code [code]:
  random-bst (set xs) = map-pmf bst-of-list (pmf-of-set (permutations-of-set (set

```

xs)))
by (rule random-bst-altdef) simp-all

lemma random-bst-singleton [simp]: random-bst {x} = return-pmf (Node Leaf x Leaf)
by (simp add: random-bst-altdef pmf-of-set-singleton)

lemma size-random-bst:
assumes $t \in \text{set-pmf } (\text{random-bst } A)$ finite A
shows size t = card A
proof –
from *assms* **obtain** xs **where** distinct xs A = set xs t = bst-of-list xs
by (auto simp: random-bst-altdef dest: permutations-of-setD)
thus ?thesis **using** ⟨finite A⟩ **by** (simp add: distinct-card)
qed

lemma random-bst-image:
assumes finite A strict-mono-on A f
shows random-bst (f ‘ A) = map-pmf (map-tree f) (random-bst A)
proof –
from *assms*(2) **have** inj: inj-on f A **by** (rule strict-mono-on-imp-inj-on)
with *assms* **have** inj-on (map f) (permutations-of-set A)
by (intro inj-on-mapI) auto
with *assms* inj **have** random-bst (f ‘ A) =
map-pmf (λx. bst-of-list (map f x)) (pmf-of-set (permutations-of-set
A))
by (simp add: random-bst-altdef permutations-of-set-image-inj map-pmf-of-set-inj
[symmetric]
pmf.map-comp o-def)
also **have** ... = map-pmf (map-tree f) (random-bst A)
unfolding random-bst-altdef[OF ⟨finite A⟩] pmf.map-comp o-def **using** *assms*
by (intro map-pmf-cong refl bst-of-list-map[of A f]) (auto dest: permutations-of-setD)
finally **show** ?thesis .
qed

We can also re-phrase the non-recursive definition using the *fold-random-permutation* combinator from the HOL-Probability library, which folds over a given set in random order.

lemma random-bst-altdef':
assumes finite A
shows random-bst A = fold-random-permutation Tree-Set.insert Leaf A
proof –
have random-bst A = map-pmf bst-of-list (pmf-of-set (permutations-of-set A))
using *assms* **by** (simp add: random-bst-altdef)
also **have** ... = map-pmf (λxs. fold Tree-Set.insert xs Leaf) (pmf-of-set (permutations-of-set
A))
using *assms* **by** (intro map-pmf-cong refl) (auto simp: bst-of-list-altdef)
also **from** *assms* **have** ... = fold-random-permutation Tree-Set.insert Leaf A
by (simp add: fold-random-permutation-fold)

finally show *?thesis* .
qed

1.4 Expected height

For the purposes of the analysis of the expected height, we define the following notion of ‘expected height’, which is essentially two to the power of the height (as defined by Cormen *et al.*) with a special treatment for the empty tree, which has exponential height 0.

Note that the height defined by Cormen *et al.* differs from the *height* function here in Isabelle in that for them, the height of the empty tree is undefined and the height of a singleton tree is 0 etc., whereas in Isabelle, the height of the empty tree is 0 and the height of a singleton tree is 1.

definition *eheight* :: ‘a tree \Rightarrow nat **where**
eheight t = (if t = Leaf then 0 else 2 $^$ (height t - 1))

lemma *eheight-Leaf* [*simp*]: *eheight* Leaf = 0
by (*simp add: eheight-def*)

lemma *eheight-Node-singleton* [*simp*]: *eheight* (Node Leaf x Leaf) = 1
by (*simp add: eheight-def*)

lemma *eheight-Node*:
 $l \neq \text{Leaf} \vee r \neq \text{Leaf} \implies \text{eheight} (\text{Node } l \ x \ r) = 2 * \max (\text{eheight } l) (\text{eheight } r)$
by (*cases l; cases r*) (*simp-all add: eheight-def max-power-distrib-right*)

fun *eheight-rbst* :: nat \Rightarrow nat pmf **where**
eheight-rbst 0 = return-pmf 0
| *eheight-rbst* (Suc 0) = return-pmf 1
| *eheight-rbst* (Suc n) =
do {
k \leftarrow pmf-of-set {..*n*};
h1 \leftarrow *eheight-rbst* *k*;
h2 \leftarrow *eheight-rbst* (n - *k*);
return-pmf (2 * max *h1* *h2*)}

definition *eheight-exp* :: nat \Rightarrow real **where**
eheight-exp n = measure-pmf.expectation (*eheight-rbst* n) real

lemma *eheight-rbst-reduce*:
assumes $n > 1$
shows *eheight-rbst* n =
do {*k* \leftarrow pmf-of-set {..*n*}; *h1* \leftarrow *eheight-rbst* *k*; *h2* \leftarrow *eheight-rbst* (n - *k* - 1);
return-pmf (2 * max *h1* *h2*)}
using *assms* **by** (*cases n rule: eheight-rbst.cases*) (*simp-all add: lessThan-Suc-atMost*)

```

lemma Leaf-in-set-random-bst-iff:
  assumes finite A
  shows  $Leaf \in \text{set-pmf } (\text{random-bst } A) \iff A = \{\}$ 
proof
  assume  $Leaf \in \text{set-pmf } (\text{random-bst } A)$ 
  from size-random-bst[OF this] and assms show  $A = \{\}$  by auto
qed auto

lemma eheight-rbst:
  assumes finite A
  shows  $eheight\text{-rbst } (\text{card } A) = \text{map-pmf } eheight (\text{random-bst } A)$ 
using assms
proof (induction A rule: finite-psubset-induct)
  case (psubset A)
  define rank where  $rank = \text{linorder-rank } \{(x,y). x \leq y\} A$ 
  from  $\langle \text{finite } A \rangle$  have  $A = \{\} \vee \text{is-singleton } A \vee \text{card } A > 1$ 
  by (auto simp: not-less le-Suc-eq is-singleton-altdef)
  then consider  $A = \{\} \mid \text{is-singleton } A \mid \text{card } A > 1$  by blast
  thus ?case
  proof cases
    case 3
    hence nonempty:  $A \neq \{\}$  by auto
    from 3 have  $\neg \text{is-singleton } A$  by (auto simp: is-singleton-def)
    hence exists-other:  $\exists y \in A. y \neq x$  for  $x$  using  $\langle A \neq \{\} \rangle$  by (force simp: is-singleton-def)

  hence  $\text{map-pmf } eheight (\text{random-bst } A) =$ 
    do {
       $x \leftarrow \text{pmf-of-set } A;$ 
       $l \leftarrow \text{random-bst } \{y \in A. y < x\};$ 
       $r \leftarrow \text{random-bst } \{y \in A. y > x\};$ 
       $\text{return-pmf } (eheight (\text{Node } l x r))$ 
    }
  using  $\langle \text{finite } A \rangle$  by (subst random-bst.simps) (auto simp: map-bind-pmf)
  also have  $\dots = \text{do}$  {
     $x \leftarrow \text{pmf-of-set } A;$ 
     $l \leftarrow \text{random-bst } \{y \in A. y < x\};$ 
     $r \leftarrow \text{random-bst } \{y \in A. y > x\};$ 
     $\text{return-pmf } (2 * \max (eheight l) (eheight r))$ 
  }
  using 3  $\langle \text{finite } A \rangle$  exists-other
  by (intro bind-pmf-cong refl, subst eheight-Node)
  (force simp: Leaf-in-set-random-bst-iff not-less nonempty eheight-Node)+
  also have  $\dots = \text{do}$  {
     $x \leftarrow \text{pmf-of-set } A;$ 
     $h1 \leftarrow \text{map-pmf } eheight (\text{random-bst } \{y \in A. y < x\});$ 
     $h2 \leftarrow \text{map-pmf } eheight (\text{random-bst } \{y \in A. y > x\});$ 
     $\text{return-pmf } (2 * \max h1 h2)$ 
  }

```

```

    by (simp add: bind-map-pmf)
  also have ... = do {
    x ← pmf-of-set A;
    h1 ← eheight-rbst (card {y ∈ A. y < x});
    h2 ← eheight-rbst (card {y ∈ A. y > x});
    return-pmf (2 * max h1 h2)
  }
  using ⟨A ≠ {}⟩ ⟨finite A⟩ by (intro bind-pmf-cong psubset.IH [symmetric]
refl) auto
  also have ... = do {
    k ← map-pmf rank (pmf-of-set A);
    h1 ← eheight-rbst k;
    h2 ← eheight-rbst (card A - k - 1);
    return-pmf (2 * max h1 h2)
  }
  unfolding bind-map-pmf
proof (intro bind-pmf-cong refl, goal-cases)
  case (1 x)
  have rank x = card {y ∈ A - {x}. y ≤ x} by (simp add: rank-def linorder-rank-def)
  also have {y ∈ A - {x}. y ≤ x} = {y ∈ A. y < x} by auto
  finally show ?case by simp
next
  case (2 x)
  have A - {x} = {y ∈ A - {x}. y ≤ x} ∪ {y ∈ A. y > x} by auto
  also have card ... = rank x + card {y ∈ A. y > x}
  using ⟨finite A⟩ by (subst card-Un-disjoint) (auto simp: rank-def linorder-rank-def)
  finally have card {y ∈ A. y > x} = card A - rank x - 1
  using 2 ⟨finite A⟩ ⟨A ≠ {}⟩ by simp
  thus ?case by simp
qed
also have map-pmf rank (pmf-of-set A) = pmf-of-set {..

```

```

lemma finite-pmf-set-eheight-rbst [simp, intro]: finite (set-pmf (eheight-rbst n))
proof -
  have eheight-rbst n = map-pmf eheight (random-bst {..

```

finally show *?thesis* .
qed

lemma *eheight-exp-0* [*simp*]: *eheight-exp 0 = 0*
by (*simp add: eheight-exp-def*)

lemma *eheight-exp-1* [*simp*]: *eheight-exp (Suc 0) = 1*
by (*simp add: eheight-exp-def lessThan-Suc*)

lemma *eheight-exp-reduce-bound*:

assumes $n > 1$

shows $eheight\text{-}exp\ n \leq 4 / n * (\sum k < n. eheight\text{-}exp\ k)$

proof –

have [*simp*]: $real\ (max\ a\ b) = max\ (real\ a)\ (real\ b)$ **for** $a\ b$

by (*simp add: max-def*)

let $?f = \lambda(h1, h2). max\ h1\ h2$

let $?p = \lambda k. pair\text{-}pmf\ (eheight\text{-}rbst\ k)\ (eheight\text{-}rbst\ (n - Suc\ k))$

have $eheight\text{-}exp\ n = measure\text{-}pmf.\text{expectation}\ (eheight\text{-}rbst\ n)\ real$

by (*simp add: eheight-exp-def*)

also have $\dots = 1 / real\ n * (\sum k < n. measure\text{-}pmf.\text{expectation}\ (map\text{-}pmf\ (\lambda(h1, h2). 2 * max\ h1\ h2)\ (?p\ k))\ real)$

(**is** $=$ $*$ $?S$) **unfolding** *pair-pmf-def map-bind-pmf*

by (*subst eheight-rbst-reduce [OF assms], subst pmf-expectation-bind-pmf-of-set (insert assms, auto simp: sum-divide-distrib divide-simps)*)

also have $?S = (\sum k < n. measure\text{-}pmf.\text{expectation}\ (map\text{-}pmf\ (\lambda x. 2 * x)\ (map\text{-}pmf\ ?f\ (?p\ k)))\ real)$

by (*simp only: pmf.map-comp o-def case-prod-unfold*)

also have $\dots = 2 * (\sum k < n. measure\text{-}pmf.\text{expectation}\ (map\text{-}pmf\ ?f\ (?p\ k))\ real)$ (**is** $=$ $*$ $?S'$)

by (*subst integral-map-pmf (simp add: sum-distrib-left)*)

also have $?S' = (\sum k < n. measure\text{-}pmf.\text{expectation}\ (?p\ k)\ (\lambda(h1, h2). max\ (real\ h1)\ (real\ h2)))$

by (*simp add: case-prod-unfold*)

also have $\dots \leq (\sum k < n. measure\text{-}pmf.\text{expectation}\ (?p\ k)\ (\lambda(h1, h2). real\ h1 + real\ h2))$

unfolding *integral-map-pmf case-prod-unfold*

by (*intro sum-mono Bochner-Integration.integral-mono integrable-measure-pmf-finite*)
auto

also have $\dots = (\sum k < n. eheight\text{-}exp\ k) + (\sum k < n. eheight\text{-}exp\ (n - Suc\ k))$

by (*subst expectation-add-pair-pmf (auto simp: sum.distrib eheight-exp-def)*)

also have $(\sum k < n. eheight\text{-}exp\ (n - Suc\ k)) = (\sum k < n. eheight\text{-}exp\ k)$

by (*intro sum.reindex-bij-witness[of - $\lambda k. n - Suc\ k$ $\lambda k. n - Suc\ k$] auto*)

also have $1 / real\ n * (2 * (\dots + \dots)) = 4 / real\ n * \dots$ **by** *simp*

finally show *?thesis using assms by (simp-all add: mult-left-mono divide-right-mono)*
qed

We now define the following upper bound on the expected exponential height due to Cormen *et al.* [2]:

lemma *eheight-exp-bound*: $eheight\text{-}exp\ n \leq real\ ((n + 3)\ choose\ 3) / 4$

```

proof (induction n rule: less-induct)
  case (less n)
  consider n = 0 | n = 1 | n > 1 by force
  thus ?case
  proof cases
    case 3
    hence eheight-exp n ≤ 4 / n * (∑ k<n. eheight-exp k)
      by (rule eheight-exp-reduce-bound)
    also have (∑ k<n. eheight-exp k) ≤ (∑ k<n. real ((k + 3) choose 3) / 4)
      by (intro sum-mono less.IH) auto
    also have ... = real (∑ k<n. ((k + 3) choose 3)) / 4
      by (simp add: sum-divide-distrib)
    also have (∑ k<n. ((k + 3) choose 3)) = (∑ k≤n - 1. ((k + 3) choose 3))
      using ⟨n > 1⟩ by (intro sum.cong) auto
    also have ... = ((n + 3) choose 4)
      using choose-rising-sum(1)[of 3 n - 1] and ⟨n > 1⟩ by (simp add: add-ac
Suc3-eq-add-3)
    also have 4 / real n * (... / 4) = real ((n + 3) choose 3) / 4 using ⟨n > 1⟩
      by (cases n) (simp-all add: binomial-fact fact-numeral divide-simps)
    finally show ?thesis using ⟨n > 1⟩ by (simp add: mult-left-mono divide-right-mono)
  qed (auto simp: eval-nat-numeral)
qed

```

We then show that this is indeed an upper bound on the expected exponential height by induction over the set of elements. This proof mostly follows that by Cormen *et al.* [2], and partially an answer on the Computer Science Stack Exchange [1].

Since the function $\lambda x. 2^x$ is convex, we can then easily derive a bound on the actual height using Jensen's inequality:

definition *height-exp-approx* :: nat ⇒ real **where**
height-exp-approx n = log 2 (real ((n + 3) choose 3) / 4) + 1

theorem *height-expectation-bound*:

assumes finite A A ≠ {}
shows *measure-pmf.expectation* (random-bst A) *height*
≤ *height-exp-approx* (card A)

proof –

```

have convex-on UNIV ((powr) 2)
  by (intro convex-on-realI[where f' = λx. ln 2 * 2 powr x])
  (auto intro!: derivative-eq-intros DERIV-powr simp: powr-def [abs-def])
hence 2 powr measure-pmf.expectation (random-bst A) (λt. real (height t - 1))
≤
  measure-pmf.expectation (random-bst A) (λt. 2 powr real (height t - 1))
using assms
by (intro measure-pmf.jensens-inequality[where I = UNIV])
  (auto intro!: integrable-measure-pmf-finite)
also have (λt. 2 powr real (height t - 1)) = (λt. 2 ^ (height t - 1))
by (simp add: powr-realpow)

```

also have $\text{measure-pmf.expectation (random-bst A) } (\lambda t. 2^{\wedge} (\text{height } t - 1)) =$
 $\text{measure-pmf.expectation (random-bst A) } (\lambda t. \text{real (eheight } t))$
using *assms*
by (*intro integral-cong-AE*)
(auto simp: AE-measure-pmf-iff random-bst-altdef eheight-def)
also have $\dots = \text{measure-pmf.expectation (map-pmf eheight (random-bst A)) real}$
by *simp*
also have $\text{map-pmf eheight (random-bst A) = eheight-rbst (card A)}$
by (*rule eheight-rbst [symmetric]*) *fact+*
also have $\text{measure-pmf.expectation } \dots \text{ real = eheight-exp (card A)}$
by (*simp add: eheight-exp-def*)
also have $\dots \leq \text{real } ((\text{card } A + 3) \text{ choose } 3) / 4$ **by** (*rule eheight-exp-bound*)
also have $\text{measure-pmf.expectation (random-bst A) } (\lambda t. \text{real (height } t - 1)) =$
 $\text{measure-pmf.expectation (random-bst A) } (\lambda t. \text{real (height } t) - 1)$
proof (*intro integral-cong-AE AE-pmfI, goal-cases*)
case ($3\ t$)
with $\langle A \neq \{\} \rangle$ **and** *assms* **show** *?case*
by (*subst of-nat-diff*) (*auto simp: Suc-le-eq random-bst-altdef*)
qed *auto*
finally have $2 \text{ powr } \text{measure-pmf.expectation (random-bst A) } (\lambda t. \text{real (height } t)$
 $- 1)$
 $\leq \text{real } ((\text{card } A + 3) \text{ choose } 3) / 4 .$
hence $\log 2 (2 \text{ powr } \text{measure-pmf.expectation (random-bst A) } (\lambda t. \text{real (height } t)$
 $- 1)) \leq$
 $\log 2 (\text{real } ((\text{card } A + 3) \text{ choose } 3) / 4)$ (**is** *?lhs* \leq *?rhs*)
by (*subst log-le-cancel-iff*) *auto*
also have *?lhs* $= \text{measure-pmf.expectation (random-bst A) } (\lambda t. \text{real (height } t) -$
 $1)$
by *simp*
also have $\dots = \text{measure-pmf.expectation (random-bst A) } (\lambda t. \text{real (height } t)) -$
 1
using *assms*
by (*subst Bochner-Integration.integral-diff*) (*auto intro!: integrable-measure-pmf-finite*)
finally show *?thesis* **by** (*simp add: height-exp-approx-def*)
qed

This upper bound is asymptotically equivalent to $c \ln n$ with $c = \frac{3}{\ln 2} \approx 4.328$. This is actually a relatively tight upper bound, since the exact asymptotics of the expected height of a random BST is $c \ln n$ with $c \approx 4.311$. [3] However, the proof of these precise asymptotics is very intricate and we will therefore be content with the upper bound.

In particular, we can now show that the expected height is $O(\log n)$.

lemma *ln-sum-bigo-ln*: $(\lambda x::\text{real}. \ln (x + c)) \in O(\ln)$

proof (*rule bigoI-tendsto*)

from *eventually-gt-at-top*[*of 1::real*] **show** *eventually* $(\lambda x::\text{real}. \ln x \neq 0)$ *at-top*
by *eventually-elim simp-all*

next

show $((\lambda x. \ln (x + c) / \ln x) \longrightarrow 1)$ *at-top*

proof (rule *lhospital-at-top-at-top*)
show eventually $(\lambda x. ((\lambda x. \ln (x + c)) \text{ has-real-derivative inverse } (x + c)) \text{ (at } x)) \text{ at-top}$
using eventually-gt-at-top[*of -c*]
by eventually-elim (auto intro!: *derivative-eq-intros simp: field-simps*)
show eventually $(\lambda x. ((\lambda x. \ln x) \text{ has-real-derivative inverse } x) \text{ (at } x)) \text{ at-top}$
using eventually-gt-at-top[*of 0*]
by eventually-elim (auto intro!: *derivative-eq-intros simp: field-simps*)
show $((\lambda x. \text{inverse } (x + c) / \text{inverse } x) \longrightarrow 1) \text{ at-top}$
proof (rule *Lim-transform-eventually*)
show eventually $(\lambda x. \text{inverse } (1 + c / x) = \text{inverse } (x + c) / \text{inverse } x)$
at-top
using eventually-gt-at-top[*of 0::real*] eventually-gt-at-top[*of -c*]
by eventually-elim (*simp add: field-simps*)
have $((\lambda x. \text{inverse } (1 + c / x)) \longrightarrow \text{inverse } (1 + 0)) \text{ at-top}$
by (*intro tendsto-inverse tendsto-add tendsto-const*
real-tendsto-divide-at-top[OF tendsto-const] filterlim-ident) *simp-all*
thus $((\lambda x. \text{inverse } (1 + c / x)) \longrightarrow 1) \text{ at-top}$ **by** *simp*
qed
qed (auto *simp: ln-at-top eventually-at-top-not-equal*)
qed

corollary *height-expectation-bigo: height-exp-approx* $\in O(\ln)$

proof –

let $?T = \lambda x::\text{real}. \log 2 (x + 1) + \log 2 (x + 2) + \log 2 (x + 3) + (1 - \log 2 2^4)$

have eventually $(\lambda n. \text{height-exp-approx } n = \log 2 (\text{real } n + 1) + \log 2 (\text{real } n + 2) + \log 2 (\text{real } n + 3) + (1 - \log 2 2^4)) \text{ at-top}$

(*is eventually* $(\lambda n. - = ?T n) \text{ at-top}$) **using** eventually-gt-at-top[*of 0::nat*]

proof eventually-elim

case (*elim n*)

have $\text{height-exp-approx } n = \log 2 (\text{real } (n + 3 \text{ choose } 3) / 4) + 1$

by (*simp add: height-exp-approx-def log-divide*)

also have $\text{real } ((n + 3) \text{ choose } 3) = \text{real } (n + 3) \text{ gchoose } 3$

by (*simp add: binomial-gbinomial*)

also have $\dots / 4 = (\text{real } n + 1) * (\text{real } n + 2) * (\text{real } n + 3) / 24$

by (*simp add: gbinomial-pochhammer' numeral-3-eq-3 pochhammer-Suc add-ac*)

also have $\log 2 \dots = \log 2 (\text{real } n + 1) + \log 2 (\text{real } n + 2) + \log 2 (\text{real } n + 3) - \log 2 2^4$

by (*simp add: log-divide log-mult*)

finally show *?case* **by** *simp*

qed

hence *height-exp-approx* $\in \Theta(?T)$ **by** (rule *bigthetaI-cong*)

also have $*$: $(\lambda x. \ln (x + c) / \ln 2) \in O(\ln)$ **for** $c :: \text{real}$

by (*subst landau-o.big.cdiv-in-iff'*) (auto intro!: *ln-sum-bigo-ln*)

have $?T \in O(\lambda n. \ln (\text{real } n))$ **unfolding** *log-def*

by (*intro bigo-real-nat-transfer sum-in-bigo ln-sum-bigo-ln **) *simp-all*

finally show *?thesis* .

qed

1.5 Lookup costs

The following function describes the cost incurred when looking up a specific element in a specific BST. The cost corresponds to the number of edges traversed in the lookup.

```
primrec lookup-cost :: 'a :: linorder  $\Rightarrow$  'a tree  $\Rightarrow$  nat where
  lookup-cost x Leaf = 0
| lookup-cost x (Node l y r) =
  (if x = y then 0
   else if x < y then Suc (lookup-cost x l)
   else Suc (lookup-cost x r))
```

Some of the literature defines these costs as 1 in the case that the current node is the correct one, i. e. their costs are our costs plus 1. These alternative costs are exactly the number of comparisons performed in the lookup. Our cost function has the advantage of precisely summing up to the internal path length and therefore gives us slightly nicer results, and since the difference is only a + 1 in the end, this variant seemed more reasonable.

It can be shown with a simple induction that The sum of all lookup costs in a tree is the internal path length of the tree.

theorem sum-lookup-costs:

fixes t :: 'a :: linorder tree

assumes bst t

shows $(\sum_{x \in \text{set-tree } t} \text{lookup-cost } x \ t) = \text{ipl } t$

using assms

proof (induction t)

case (Node l x r)

from Node.premis

have disj: $x \notin \text{set-tree } l \ x \notin \text{set-tree } r \ \text{set-tree } l \cap \text{set-tree } r = \{\}$ **by** force+

have set-tree (Node l x r) = insert x (set-tree l \cup set-tree r) **by** simp

also have $(\sum_{y \in \dots} \text{lookup-cost } y \ (\text{Node } l \ x \ r)) = \text{lookup-cost } x \ \langle l, x, r \rangle +$

$(\sum_{y \in \text{set-tree } l} \text{lookup-cost } y \ \langle l, x, r \rangle) + (\sum_{y \in \text{set-tree } r} \text{lookup-cost } y \ \langle l, x, r \rangle)$

using disj **by** (simp add: sum.union-disjoint)

also have $(\sum_{y \in \text{set-tree } l} \text{lookup-cost } y \ \langle l, x, r \rangle) = (\sum_{y \in \text{set-tree } l} 1 + \text{lookup-cost } y \ l)$

using disj **and** Node **by** (intro sum.cong refl) auto

also have $\dots = \text{size } l + \text{ipl } l$ **using** Node

by (subst sum.distrib) (simp-all add: card-set-tree-bst)

also have $(\sum_{y \in \text{set-tree } r} \text{lookup-cost } y \ \langle l, x, r \rangle) = (\sum_{y \in \text{set-tree } r} 1 + \text{lookup-cost } y \ r)$

using disj **and** Node **by** (intro sum.cong refl) auto

also have $\dots = \text{size } r + \text{ipl } r$ **using** Node

by (subst sum.distrib) (simp-all add: card-set-tree-bst)

finally show ?case **by** simp

qed *simp-all*

This allows us to easily show that the expected cost of looking up a random element in a fixed tree is the internal path length divided by the number of elements.

theorem *expected-lookup-cost*:

assumes *bst t t ≠ Leaf*

shows *measure-pmf.expectation (pmf-of-set (set-tree t)) (λx. lookup-cost x t) = ipl t / size t*

using *assms by (subst integral-pmf-of-set)*

(simp-all add: sum-lookup-costs of-nat-sum [symmetric] card-set-tree-bst)

Therefore, we will now turn to analysing the internal path length of a random BST. This then clearly related to the expected lookup costs of a random element in a random BST by the above result.

1.6 Average Path Length

The internal path length satisfies the recursive equation $ipl \langle l, x, r \rangle = ipl \ l + size \ l + ipl \ r + size \ r$. This is quite similar to the number of comparisons performed by QuickSort, and indeed, we can reduce the internal path length of a random BST to the number of comparisons performed by QuickSort on a randomly-ordered list relatively easily:

theorem *map-pmf-random-bst-eq-rqs-cost*:

assumes *finite A*

shows *map-pmf ipl (random-bst A) = rqs-cost (card A)*

using *assms*

proof *(induction A rule: finite-psubset-induct)*

case *(psubset A)*

show *?case*

proof *(cases A = {})*

case *False*

note *A = ⟨finite A⟩ ⟨A ≠ {}⟩*

define *n where n = card A - 1*

define *rank :: 'a ⇒ nat where rank = linorder-rank {(x,y). x ≤ y} A*

from *A have card: card A = Suc n by (cases card A) (auto simp: n-def)*

from *A have map-pmf ipl (random-bst A) =*

do {

x ← pmf-of-set A;

(l,r) ← pair-pmf (random-bst {y ∈ A. y < x}) (random-bst {y ∈

A. y > x});

return-pmf (ipl (Node l x r))

}

by *(subst random-bst.simps)*

(simp-all add: pair-pmf-def card map-pmf-def bind-assoc-pmf bind-return-pmf)

also have *... = do {*

x ← pmf-of-set A;

```

    (l,r) ← pair-pmf (random-bst {y ∈ A. y < x}) (random-bst {y
∈ A. y > x});
    return-pmf (n + ipl l + ipl r)
  }
proof (intro bind-pmf-cong refl, clarify, goal-cases)
  case (1 x l r)
  from 1 and A have n = card (A - {x}) by (simp add: n-def)
  also have A - {x} = {y ∈ A. y < x} ∪ {y ∈ A. y > x} by auto
  also have card ... = card {y ∈ A. y < x} + card {y ∈ A. y > x}
    using ⟨finite A⟩ by (intro card-Un-disjoint) auto
  also from 1 and A have card {y ∈ A. y < x} = size l by (auto dest:
size-random-bst)
  also from 1 and A have card {y ∈ A. y > x} = size r by (auto dest:
size-random-bst)
  finally show ?case by simp
qed
  also have ... = do {
    x ← pmf-of-set A;
    (l,r) ← pair-pmf (map-pmf ipl (random-bst {y ∈ A. y < x}))
      (map-pmf ipl (random-bst {y ∈ A. y > x}));
    return-pmf (n + l + r)
  } by (simp add: map-pair [symmetric] case-prod-unfold bind-map-pmf)
  also have ... = do {
    i ← map-pmf rank (pmf-of-set A);
    (l,r) ← pair-pmf (rqs-cost i) (rqs-cost (n - i));
    return-pmf (n + l + r)
  } (is - = bind-pmf - ?f) unfolding bind-map-pmf
proof (intro bind-pmf-cong refl pair-pmf-cong, goal-cases)
  case (1 x)
  have map-pmf ipl (random-bst {y ∈ A. y < x}) = rqs-cost (card {y ∈ A. y
< x})
    using 1 and A by (intro psubset.IH) auto
  also have {y ∈ A. y < x} = {y ∈ A - {x}. y ≤ x} by auto
  hence card {y ∈ A. y < x} = rank x by (simp add: rank-def linorder-rank-def)
  finally show ?case .
next
  case (2 x)
  have map-pmf ipl (random-bst {y ∈ A. y > x}) = rqs-cost (card {y ∈ A. y
> x})
    using 2 and A by (intro psubset.IH) auto
  also have {y ∈ A. y > x} = A - {x} - {y ∈ A - {x}. y ≤ x} by auto
  hence card {y ∈ A. y > x} = card ... by (simp only:)
  also from 2 and A have ... = n - rank x
    by (subst card-Diff-subset) (auto simp: rank-def linorder-rank-def n-def)
  finally show ?case .
qed
  also from A have map-pmf rank (pmf-of-set A) = pmf-of-set {..unfolding rank-def by (intro map-pmf-of-set-bij-betw bij-betw-linorder-rank[of
UNIV]) auto

```

```

also have {..card A} = {..n} by (auto simp: card)
also have pmf-of-set ...  $\gg$  ?f = rqs-cost (card A)
  by (simp add: pair-pmf-def bind-assoc-pmf bind-return-pmf card)
finally show ?thesis .
qed simp-all
qed

```

In particular, this means that the expected values are the same:

```

corollary expected-ipl-random-bst-eq:
  assumes finite A
  shows measure-pmf.expectation (random-bst A) ipl = rqs-cost-exp (card A)
proof -
  have measure-pmf.expectation (random-bst A) ipl =
    measure-pmf.expectation (map-pmf ipl (random-bst A)) real by simp
  also from assms have map-pmf ipl (random-bst A) = rqs-cost (card A)
    by (rule map-pmf-random-bst-eq-rqs-cost)
  also have measure-pmf.expectation ... real = rqs-cost-exp (card A)
    by (rule expectation-rqs-cost)
  finally show ?thesis .
qed

```

Therefore, the results about the expected number of comparisons of Quick-Sort carry over to the expected internal path length:

```

corollary expected-ipl-random-bst-eq':
  assumes finite A
  shows measure-pmf.expectation (random-bst A) ipl =
    2 * real (card A + 1) * harm (card A) - 4 * real (card A)
  by (simp add: expected-ipl-random-bst-eq rqs-cost-exp-eq assms)
end

```

References

- [1] Proof that a randomly built binary search tree has logarithmic height. Computer Science Stack Exchange. URL: <http://cs.stackexchange.com/q/6356>.
- [2] T. H. Cormen, C. Stein, R. L. Rivest, and C. E. Leiserson. *Introduction to Algorithms*. McGraw-Hill Higher Education, 2nd edition, 2001.
- [3] B. Reed. The height of a random binary search tree. *J. ACM*, 50(3):306–332, May 2003.