Expected Shape of Random Binary Search Trees

Manuel Eberl

September 13, 2023

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 ≤ (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]: [] ∈ permutations-of-set A ←→ A = {}
by (auto simp: permutations-of-set-def)

lemma max-power-distrib-right:
  fixes a :: 'a :: linordered-semidom
  shows a > 1 =⇒ max (a ^ b) (a ^ c) = a ^ max b c
by (auto simp: max-def)

lemma set-tree-empty-iff [simp]: set-tree t = {} ←→ t = Leaf
by (cases t) auto

lemma card-set-tree-bst: bst t =⇒ card (set-tree t) = size t
proof (induction t)
case (Node l x r)
have set-tree (l, x, r) = insert x (set-tree l ∪ set-tree r) by simp
also from Node.prems have card ... = Suc (card (set-tree l ∪ set-tree r))
  by (intro card-insert-disjoint) auto
also from Node have card (set-tree l ∪ set-tree r) = size l + size r
  by (subst card-Un-disjoint) force+
finally show ?case by simp
qed simp-all

lemma pair-pmf-cong:
  p = p' =⇒ q = q' =⇒ pair-pmf p q = pair-pmf p' q'
by simp

lemma expectation-add-pair-pmf:
  fixes f :: 'a ⇒ 'c::{banach, second-countable-topology}
  assumes finite (set-pmf p) and finite (set-pmf q)
  shows measure-pmf.expectation (pair-pmf p q) (λ(x,y). f x + g y) =
measure-pmf.expectation p f + measure-pmf.expectation q g

proof

- have measure-pmf.expectation (pair-pmf p q) (λ(x,y). f x + g y) =
  measure-pmf.expectation (pair-pmf p q) (λz. f (fst z) + g (snd z))
  by (simp add: case_prod_unfold)

- also have . . . = measure-pmf.expectation (pair-pmf p q) (λz. f (fst z)) +
  measure-pmf.expectation (pair-pmf p q) (λz. g (snd z))
  by (intro Bochner-Integration.integral_add integrable_measure-pmf_finite) (auto intro: assms)

- then also have measure-pmf.expectation (pair-pmf p q) (λz. f (fst z)) =
  measure-pmf.expectation (map_pmf fst (pair-pmf p q)) g
  by simp

- also have . . . = measure-pmf.expectation (map_pmf snd (pair-pmf p q)) g
  by simp

- 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 ⇒ '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 ↔ xs = []
  by (induction xs)

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)

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

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

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

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

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

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

**lemma bst-of-list-map**: 
\(\text{fixes } f :: 'a :: \text{linorder } \Rightarrow 'b :: \text{linorder}\)
\(\text{assumes strict-mono-on } A \ f \text{ set } xs \subseteq A\)
\(\text{shows } \text{bst-of-list } (\text{map } f \text{ } xs) = \text{map-tree } f \text{ } (\text{bst-of-list } xs)\)
using \(\text{assms}\)

**proof (induction } xs \text{ rule: bst-of-list.induct)**
\(\text{case } (2 \text{ } xs)\)
\(\text{have } [xa<xs . f xa < f x] = [xa<xs . xa < x] \text{ and } [xa<xs . f xa > f x] = [xa<xs . xa > x]\)
\(\text{using 2.prems by (auto simp: strict-mono-on-imp-less-iff intro: filter-cong)}\)
\(\text{with 2 show } \text{case by (auto simp: filter-map o-def)}\)
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 :: linorder set \Rightarrow 'a tree pmf where**
\(\text{random-bst } A = \)
\(\text{(if } \neg \text{finite } A \lor A = \{\} \text{ then return-pmf Leaf)}\)
\(\text{else do }\)

4
\[ x \leftarrow \text{pmf-of-set } A; \]
\[ l \leftarrow \text{random-bst } \set{y \in A. \; y < x}; \]
\[ r \leftarrow \text{random-bst } \set{y \in A. \; y > x}; \]
\[ \text{return-pmf } (\text{Node } l \; x \; r) \]
\}

by auto
termination by (relation finite-psubset) auto

declare random-bst.simps [simp def]

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

lemma set-pmf-random-permutation [simp]:
finite A \implies set-pmf (\text{pmf-of-set (permutations-of-set } A)) = \{xs. \text{ distinct } xs \land \text{ 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 = (\lambda x. \set{y \in A. \; y < x}) and R = (\lambda x. \set{y \in A. \; y > x})
{ fix x assume x: x \in A
hence *: L \; x \subseteq A \; R \; x \subseteq 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 \neq \{\})
have random-bst A =
do {
x \leftarrow \text{pmf-of-set } A;
(l, r) \leftarrow \text{pair-pmf } (\text{random-bst } (L \; x)) \; (\text{random-bst } (R \; x));
\text{return-pmf } (\text{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 \ldots = do {
x \leftarrow \text{pmf-of-set } A;
(l, r) \leftarrow \text{pair-pmf }
(map-pmf bst-of-list (\text{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 \ldots = do {
  \begin{align*}
  &x \leftarrow \text{pmf-of-set } A; \\
  &(ls, rs) \leftarrow \text{pair-pmf (pmf-of-set (permutations-of-set (L x))}) \\
  &\quad \text{(pmf-of-set (permutations-of-set (R x)))}; \\
  &\text{return-pmf (Node (bst-of-list ls) x (bst-of-list rs))}
  \end{align*}
}

unfolding map-pair [symmetric] by (simp add: map-pmf-def case-prod-unfold bind-return-pmf bind-assoc-pmf)
also have \ldots = do {
  \begin{align*}
  &x \leftarrow \text{pmf-of-set } A; \\
  &\quad (ls, rs) \leftarrow \text{pair-pmf (pmf-of-set } \{ y \in A \setminus \{x\}. y \leq x \} \text{)}; \\
  &\quad (\text{pmf-of-set (permutations-of-set } \{ y \in A \setminus \{x\}. \neg y \leq x \})); \\
  &\text{return-pmf (Node (bst-of-list ls) x (bst-of-list rs))}
  \end{align*}
}

using \langle \text{finite } A \rangle by (intro bind-pmf-cong refl partition-random-permutations [symmetric]) auto
also have \ldots = do {
  \begin{align*}
  &x \leftarrow \text{pmf-of-set } A; \\
  &\quad (ls, rs) \leftarrow \text{map-pmf (partition } (\lambda y. y \leq x) \\
  &\quad \text{(pmf-of-set (permutations-of-set } (A \setminus \{x\})))); \\
  &\text{return-pmf (Node (bst-of-list ls) x (bst-of-list rs))}
  \end{align*}
}

also have \ldots = map-pmf bst-of-list (pmf-of-set (permutations-of-set A))
using A by (subt random-permutation-of-set[of A])
  (auto simp: map-pmf-def bind-return-pmf o-def bind-assoc-pmf not-le)
finally show \text{thesis}.
  qed (simp-all add: pmf-of-set-singleton)
qed

lemma finite-set-random-bst [simp, intro]:
  \text{finite } A \implies \text{finite } (\text{set-pmf } (\text{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 ∈ set-pmf (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))
  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 ⇒ nat where
\[ eheight t = (if t = Leaf then 0 else 2 ^ (\text{height } t - 1)) \]

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

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

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

**fun** eheight-rbst :: nat ⇒ nat pmf where
\[ \text{eheight-rbst 0 = return-pmf 0} \]
\[ \text{eheight-rbst (Suc 0) = return-pmf 1} \]
\[ \text{eheight-rbst (Suc } n) = \]
\[ \text{do } \{
\text{k ← pmf-of-set } \{..n\};
\text{h1 ← eheight-rbst k;}
\text{h2 ← eheight-rbst } (\text{n } - \text{k});
\text{return-pmf } (2 \times \text{max h1 h2})\}\]

**definition** eheight-exp :: nat ⇒ real where
\[ \text{eheight-exp } n = \text{measure-pmf.expectation } (\text{eheight-rbst } n) \times \text{real} \]

**lemma** eheight-rbst-reduce:
\[ \text{assumes } n > 1 \]
\[ \text{shows } \text{eheight-rbst } n = \]
\[ \text{do } \{ \text{k ← pmf-of-set } \{..n\}; \text{h1 ← eheight-rbst k}; \text{h2 ← eheight-rbst } (\text{n } - \text{k } - \text{1}); \text{return-pmf } (2 \times \text{max h1 h2})\}\]
\[ \text{using } \text{assms by (cases } n \text{ rule: eheight-rbst.cases) (simp-all add: lessThan-Suc-atMost)} \]
lemma Leaf-in-set-random-bst-iff:
  assumes finite A
  shows Leaf ∈ set-pmf (random-bst A) ⟷ A = {}
proof
  assume Leaf ∈ set-pmf (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-rbst (card A) = map-pmf eheight (random-bst A)
using assms
proof (induction A rule: finite-psubset-induct)
case psubset A
  define rank where rank = linorder-rank {(x,y) . x ≤ y} A
  from finite A have A = {} ∨ is-singleton A ∨ card A > 1
    by (auto simp: not-less le-Suc-eq is-singleton-altdef)
  then consider A = {} | is-singleton A | card A > 1 by blast
  thus ?case
proof cases
  case 3
  hence nonempty: A ≠ {} by auto
  from 3 have ¬is-singleton A by (auto simp: is-singleton-def)
  hence exists-other: ∃y∈A. y ≠ x for x using A ≠ {} by (force simp: is-singleton-def)
  hence map-pmf eheight (random-bst A) =
    do { x ← pmf-of-set A;
         l ← random-bst {y ∈ A. y < x};
         r ← random-bst {y ∈ A. y > x};
         return-pmf (eheight (Node l x r))
      }
    using finite A by (subst random-bst.simps) (auto simp: map-bind-pmf)
  also have . . . = do { x ← pmf-of-set A;
                         l ← random-bst {y ∈ A. y < x};
                         r ← random-bst {y ∈ A. y > x};
                         return-pmf (2 * max (eheight l) (eheight r))
                     }
    using finite A · 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 . . . = do { x ← pmf-of-set A;
                         h1 ← map-pmf eheight (random-bst {y ∈ A. y < x});
                         h2 ← map-pmf eheight (random-bst {y ∈ A. y > x});
                         return-pmf (2 * max h1 h2)
                     }
by (simp add: bind-map-pmf)
also have \ldots = do { 
x \leftarrow pmf-of-set A;
  h1 \leftarrow eheight-rbst (card \{ y \in A. y < x\});
  h2 \leftarrow eheight-rbst (card \{ y \in A. y > x\});
  return-pmf (2 \ast \max h1 h2)
}
using \langle A \neq \{\} \rangle \langle \text{finite A} \rangle by (intro bind-pmf-cong psubset.IH [symmetric]
refl) auto
also have \ldots = do { 
k \leftarrow map-pmf rank (pmf-of-set A);
  h1 \leftarrow eheight-rbst k;
  h2 \leftarrow eheight-rbst (card A - k - 1);
  return-pmf (2 \ast \max h1 h2)
}
unfolding bind-map-pmf
proof (intro bind-pmf-cong refl, goal-cases)
case (1 \ x)
  have rank x = card \{ y \in A - \{x\}. y \leq x \} by (simp add: rank-def linorder-rank-def)
  also have \{ y \in A - \{x\}. y \leq x \} = \{ y \in A. y < x \} by auto
  finally show \?case by simp
next
case (2 \ x)
  have A - \{x\} = \{ y \in A - \{x\}. y \leq x \} \cup \{ y \in A. y \geq x \} by auto
  also have card \ldots = rank x + card \{ y \in A. y \geq x \}
  using \text{finite A} by (subst card-Un-disjoint) (auto simp: rank-def linorder-rank-def)
  finally have card \{ y \in A. y > x \} = card A - rank x - 1
  using 2 \langle \text{finite A} \rangle \langle A \neq \{\} \rangle by simp
  thus ?case by simp
qed
also have map-pmf rank (pmf-of-set A) = pmf-of-set \{..<\text{card A} \}
  using \langle A \neq \{\} \rangle \langle \text{finite A} \rangle unfolding rank-def
  by (intro map-pmf-of-set-bij-betw bij-betw-l inorder-rank-def[of UNIV]) auto
also have do { 
k \leftarrow pmf-of-set \{..<\text{card A} \};
  h1 \leftarrow eheight-rbst k;
  h2 \leftarrow eheight-rbst (card A - k - 1);
  return-pmf (2 \ast \max h1 h2)
} = eheight-rbst (card A)
by (rule eheight-rbst-reduce [symmetric]) fact+
finally show \?thesis ..
qed (auto simp: is-singleton-def)

lemma finite-pmf-set-eheight-rbst [simp, intro]: finite (set-pmf (eheight-rbst n))
proof
  have eheight-rbst n = map-pmf eheight (random-bst \{..<n\})
    by (subst eheight-rbst [symmetric]) auto
  also have finite (set-pmf \ldots) by simp
We now define the following upper bound on the expected exponential height due to Cormen et al. [2]:

\[ eheight-exp \leq \frac{n+3}{4} \]
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 \( \text{eheight-exp} n \leq \frac{4}{n} \ast (\sum_{k<n.} \text{eheight-exp} k) \)
  by (rule eheight-exp-reduce-bound)
  also have \( (\sum_{k<n.} \text{eheight-exp} k) \leq (\sum_{k<n.} \text{real} ((k + 3) \text{ choose} 3) / 4) \)
  by (intro sum-mono less.IH) auto
  also have \( \ldots = \text{real} ((\sum_{k<n.} ((k + 3) \text{ choose} 3)) / 4 \)
  by (simp add: sum-divide-distrib)
  also have \( (\sum_{k<n.} ((k + 3) \text{ choose} 3)) = (\sum_{k\leq n - 1.} ((k + 3) \text{ choose} 3)) \)
  using \( n > 1 \) by (intro sum.cong) auto
  also have \( \ldots = ((n + 3) \text{ choose} 4) \)
  using choose-rising-sum[of 3 \( n - 1 \)] and \( \langle n > 1 \rangle \) by (simp add: add-ac Suc3-eq-add-3)
also have \( \frac{4}{\text{real} n} \ast (\ldots / 4) = \text{real} ((n + 3) \text{ choose} 3) / 4 \) using \( \text{assms} \)
by (auto simp: mult-left-mono divide-right-mono)
finally show \( \text{?thesis} \) using \( \langle n > 1 \rangle \) by (simp add: mult-left-mono divide-right-mono)
qed (auto simp: eval-nat-numeral)

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 :: \( \text{nat} \Rightarrow \text{real} \) where
height-exp-approx \( n = \log 2 \ast (\text{real} ((n + 3) \text{ choose} 3) / 4) + 1 \)

**theorem** height-expectation-bound:
assumes finite A A \( \neq \{\} \)
sows measure-pmf.expectation (random-bst A) height 
\( \leq \text{height-exp-approx} (\text{card} A) \)
proof 
  have convex-on UNIV ((\text{powr} 2)
  by (intro convex-on-realI[where \( f' = \lambda x. \ln 2 \ast 2 \text{ powr} x \)])
  (auto intro!: derivative-eq-intros DERIV-powr simp: powr-def [abs_def])
hence \( \text{powr} \text{measure-pmf.expectation} (\text{random-bst} A) (\lambda t. \text{real} (\text{height} t - 1)) \)
\( \leq \text{measure-pmf.expectation} (\text{random-bst} A) (\lambda t. 2 \text{ powr} \text{real} (\text{height} t - 1)) \)
using \( \text{assms} \)
by (intro measure-pmf.jensens-inequality[where \( I = \text{UNIV} \)]
  (auto intro!: integrable-measure-pmf-finite)
also have \( (\lambda t. 2 \text{ powr} \text{real} (\text{height} t - 1)) = (\lambda t. \text{2} \prec (\text{height} t - 1)) \)
by (simp add: powr-realpow)
also have \( \text{measure-pmf.expectation } (\text{random-bst } A) (\lambda t. 2 ^t (\text{height } t - 1)) = \text{measure-pmf.expectation } (\text{random-bst } A) (\lambda t. \text{real } (\text{height } t)) \)

\[ \text{using } \text{assms} \]

\[ \text{by } \text{intro integral-cong-AE} \]
\[ (\text{auto simp: AE-measure-pmf-iff } \text{random-bst-altdef } \text{height-def}) \]

also have \( \ldots = \text{measure-pmf.expectation } (\text{map-pmf } \text{eheight } (\text{random-bst } A)) \text{ real} \)

\[ \text{by } \text{simp} \]

also have \( \text{map-pmf } \text{eheight } (\text{random-bst } A) = \text{eheight-rbst } (\text{card } A) \)

\[ \text{by } (\text{rule eheight-rbst [symmetric]} \text{ fact+}) \]

also have \( \text{measure-pmf.expectation } \ldots \text{ real } = \text{eheight-exp } (\text{card } A) \)

\[ \text{by } (\text{simp add: eheight-exp-def}) \]

also have \( \ldots \leq \text{real } ((\text{card } A + 3) \text{ choose } 3) / 4 \text{ by } (\text{rule eheight-exp-bound}) \)

also have \( \text{measure-pmf.expectation } (\text{random-bst } A) (\lambda t. \text{real } (\text{height } t - 1)) = \text{measure-pmf.expectation } (\text{random-bst } A) (\lambda t. \text{real } (\text{height } t) - 1) \)

\[ \text{proof } \text{intro integral-cong-AE } \text{AE-pmfI, goal-cases} \]

\[ \text{case } (3 \, t) \]

\[ \text{with } \{ A \neq \{ \} \} \text{ and } \text{assms show } ?\text{case} \]

\[ \text{by } (\text{subst of-nat-diff}) \text{ (auto simp: Suc-le-eq } \text{random-bst-altdef}) \]

\[ \text{qed auto} \]

finally have \( 2 \text{ powr measure-pmf.expectation } (\text{random-bst } A) (\lambda t. \text{real } (\text{height } t) - 1) \)

\[ \leq \text{real } ((\text{card } A + 3) \text{ choose } 3) / 4 , \]

hence \( \log 2 (2 \text{ powr measure-pmf.expectation } (\text{random-bst } A) (\lambda t. \text{real } (\text{height } t) - 1)) \leq \)

\( \log 2 (\text{real } ((\text{card } A + 3) \text{ choose } 3) / 4) \text{ is } ?\text{lhs} \leq ?\text{rhs} \)

\[ \text{by } (\text{subst \log-le-cancel-iff}) \text{ auto} \]

also have \( ?\text{lhs} = \text{measure-pmf.expectation } (\text{random-bst } A) (\lambda t. \text{real } (\text{height } t) - 1) \)

\[ \text{by } \text{simp} \]

also have \( \ldots = \text{measure-pmf.expectation } (\text{random-bst } A) (\lambda t. \text{real } (\text{height } t)) - 1 \)

\[ \text{using } \text{assms} \]

\[ \text{by } (\text{subst Bochner-Integration.integral-diff}) \text{ (auto intro!: integrable-measure-pmf-finite}) \]

finally show \( ?\text{thesis} \text{ by } (\text{simp add: height-exp-approx-def}) \]

\[ \text{qed} \]

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

\[ \text{proof } (\text{rule bigoI-tendsto}) \]

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

\[ \text{by } \text{eventually-elim simp-all} \]

next

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

qed

corollary height-expectation-bigo: height-exp-approx ∈ O(ln)
proof —
  let \(?T = \lambda x::real. \log 2 (x + 1) + \log 2 (x + 2) + \log 2 (x + 3) + (1 - \log 2 24)\)
  have eventually \((\lambda n. \ln (\ln (x + c) / \ln 2))\) (at top)
    using eventually-gt-at-top[of 0::nat]
  proof eventually-elim
    case (elim n)
    have \((\lambda n. \ln (\ln (x + c) / \ln 2))\) (at top)
      by (simp add: height-exp-approx-def log-divide)
    also have \((\ln (\ln (x + c) / \ln 2))\) (at top)
      by (simp add: binomial-binomial)
    also have \((\ln (\ln (x + c) / \ln 2))\) (at top)
      by (simp add: gbinomial-pochhammer numeral-3_eq_3 pochhammer-Suc add-ac)
    also have \((\ln (\ln (x + c) / \ln 2))\) (at top)
      by (simp add: log-divide log-mult)
    finally show \(?T n\) (at top) by simp
  qed
  hence \((\lambda n. \ln (\ln (x + c) / \ln 2))\) (at top)
  also have \((\ln (\ln (x + c) / \ln 2))\) (at top)
    by (auto intro!: ln-sum-bigo-ln)
  have \(?T ∈ O(\ln n\cdot \ln (\ln n))\)
    unfolding log-def
    by (intro bigo-real-nat-transfer sum-in-bigo ln-sum-bigo-ln \(*\)) simp-all
  finally show \(\Theta(\ln (\ln (x + c) / \ln 2))\) by (rule bigthetaI-cong)

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.

\begin{verbatim}
primrec lookup-cost :: 'a :: linorder ⇒ 'a tree ⇒ 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))
\end{verbatim}

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.

\begin{verbatim}
theorem sum-lookup-costs:
  fixes t :: 'a :: linorder tree
  assumes bst t
  shows (∑x ∈ set-tree t. lookup-cost x t) = ipl t
using assms
proof (induction t)
case (Node l x r)
  from Node.prems
  have disj: x ∉ set-tree l x ∉ set-tree r set-tree l ∩ set-tree r = {} by force
  have set-tree (Node l x r) = insert x (set-tree l ∪ set-tree r) by simp
  also have (∑y ∈ set-tree l. lookup-cost y (Node l x r)) = lookup-cost x (l, x, r) +
    (∑y ∈ set-tree l. lookup-cost y (l, x, r)) + (∑y ∈ set-tree r. lookup-cost y (l, x, r))
    using disj by (simp add: sum.union-disjoint)
  also have (∑y ∈ set-tree l. lookup-cost y (l, x, r)) = (∑y ∈ set-tree l. l +
    lookup-cost y l)
    using disj and Node by (intro sum.cong refl) auto
  also have ... = size l + ipl l using Node
    by (subst sum.distrib) (simp-all add: card-set-tree-bst)
  also have (∑y ∈ set-tree r. lookup-cost y (l, x, r)) = (∑y ∈ set-tree r. r +
    lookup-cost y r)
    using disj and Node by (intro sum.cong refl) auto
  also have ... = size r + ipl r using Node
    by (subst sum.distrib) (simp-all add: card-set-tree-bst)
  finally show ?case by simp
\end{verbatim}
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:**

\[ \text{assumes } \text{bst } t \quad \text{shows } \text{measure-pmf} \cdot \text{expectation} (\text{pmf-of-set} (\text{set-tree } t)) (\lambda x. \text{lookup-cost } x \ t) = \frac{\text{ipl } t}{\text{size } t} \]

\text{using} \text{assms by} (\text{subst integral-pmf-of-set})

\text{(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

\[ \text{ipl } (l, x, r) = \text{ipl } l + \text{size } l + \text{ipl } r + \text{size } r. \]

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:**

\[ \text{assumes } \text{finite } A \quad \text{shows } \text{map-pmf} \text{ ipl } (\text{random-bst } A) = \text{rqs-cost } (\text{card } A) \]

\text{using} \text{assms}

\text{proof} (\text{induction } A \text{ rule: finite-psubset-induct})

\text{case } \text{False}

\text{proof} (\text{cases } A = \{\})

\text{case } False

\text{note } A = \langle \text{finite } A \rangle \langle A \neq \{\} \rangle;

\text{define } n \text{ where } n = \text{card } A - 1

\text{define} \text{ rank } :: 'a \Rightarrow \text{nat} \quad \text{where} \text{ rank } = \text{linorder-rank } \langle (x,y). \ x \leq y \rangle \ A

\text{from } A \text{ have} \text{ card: } \text{card } A = \text{Suc } n \text{ by} (\text{cases card } A) (\text{auto simp: n-def})

\text{from } A \text{ have } \text{map-pmf} \text{ ipl } (\text{random-bst } A) =

\text{do } \{\text{x } \leftarrow \text{pmf-of-set A};\}

\text{(l,r) } \leftarrow \text{pair-pmf } (\text{random-bst } \langle y \in A. \ y < x \rangle) \ (\text{random-bst } \langle y \in A. \ y > x \rangle));

\text{return-pmf } (\text{ipl } (\text{Node } l \ x \ r))

\text{by} (\text{subst random-bst.simps})

\text{(simp-all add: pair-pmf-def card map-pmf-def bind-assoc-pmf bind-return-pmf)}

\text{also have } \ldots = \text{do } \{\text{x } \leftarrow \text{pmf-of-set A};\}
\[(l, r) \leftarrow \text{pair-pmf} \ (\text{random-bst} \ \{ y \in A. \ y < x \}) \ (\text{random-bst} \ \{ y \in A. \ y > x \})\]
\[
\text{return-pmf} \ (n + \text{ipl} \ l + \text{ipl} \ r)
\]

\textbf{proof (intro bind-pmf-cong refl, clarify, goal-cases)}

\textbf{case (1 \ x \ l \ r)}

\textbf{from 1 and A have n = card} \ \{ A - \{ x \} \} \ \textbf{by (simp add: n-def)}

\textbf{also have} \ A - \{ x \} = \{ y \in A. \ y < x \} \cup \{ y \in A. \ y > x \} \ \textbf{by auto}

\textbf{also have} \ \text{card} \ \ldots = \text{card} \ \{ y \in A. \ y < x \} + \text{card} \ \{ y \in A. \ y > x \}

\textbf{using \{finite A\} by (intro card-Un-disjoint) auto}

\textbf{also from 1 and A have} \ \text{card} \ \{ y \in A. \ y < x \} = \text{size} \ l \ \textbf{by (auto dest: size-random-bst)}

\textbf{also from 1 and A have} \ \text{card} \ \{ y \in A. \ y > x \} = \text{size} \ r \ \textbf{by (auto dest: size-random-bst)}

\textbf{finally show} \ ?\text{case by simp}

\textbf{qed}

\textbf{also have} \ \ldots = \text{do} \ \{ \ 
\textbf{x} \leftarrow \text{pmf-of-set} \ A; 
(l, r) \leftarrow \text{pair-pmf} \ (\text{map-pmf} \ \text{ipl} \ (\text{random-bst} \ \{ y \in A. \ y < x \})) 
\text{ (map-pmf} \ \text{ipl} \ (\text{random-bst} \ \{ y \in A. \ y > x \}))\); 
\text{return-pmf} \ (n + l + r)
\} \ \textbf{by (simp add: map-pair \{symmetric\} case-prod-unfold bind-map-pmf)

\textbf{also have} \ \ldots = \text{do} \ \{ 
\textbf{i} \leftarrow \text{map-pmf} \ \text{rank} \ (\text{pmf-of-set} \ A); 
(l, r) \leftarrow \text{pair-pmf} \ (\text{rqs-cost} \ i) \ (\text{rqs-cost} \ (n - i)); 
\text{return-pmf} \ (n + l + r)
\} \ (\text{is - = bind-pmf - ?f}) \ \textbf{unfolding bind-map-pmf}

\textbf{proof (intro bind-pmf-cong refl pair-pmf-cong, goal-cases)}

\textbf{case (1 \ x)}

\textbf{have} \ \text{map-pmf} \ \text{ipl} \ (\text{random-bst} \ \{ y \in A. \ y < x \}) = \text{rqs-cost} \ (\text{card} \ \{ y \in A. \ y < x \})

\textbf{using 1 and A by (intro psubset.IH) auto}

\textbf{also have} \ \{ y \in A. \ y < x \} = \{ y \in A - \{ x \}. \ y \leq x \} \ \textbf{by auto}

\textbf{hence} \ \text{card} \ \{ y \in A. \ y < x \} = \text{rank} \ x \ \textbf{by (simp add: rank-def linorder-rank-def)}

\textbf{finally show} \ ?\text{case .}

\textbf{next}

\textbf{case (2 \ x)}

\textbf{have} \ \text{map-pmf} \ \text{ipl} \ (\text{random-bst} \ \{ y \in A. \ y > x \}) = \text{rqs-cost} \ (\text{card} \ \{ y \in A. \ y > x \})

\textbf{using 2 and A by (intro psubset.IH) auto}

\textbf{also have} \ \{ y \in A. \ y > x \} = A - \{ x \} - \{ y \in A - \{ x \}. \ y \leq x \} \ \textbf{by auto}

\textbf{hence} \ \text{card} \ \{ y \in A. \ y > x \} = \text{card} \ \ldots \ \textbf{by (simp only:)}

\textbf{also from 2 and A have} \ \ldots = n - \text{rank} \ x 

\textbf{by (subst card-Diff-subset) (auto simp: rank-def linorder-rank-def n-def)}

\textbf{finally show} \ ?\text{case .}

\textbf{qed}

\textbf{also from A have} \ \text{map-pmf} \ \text{rank} \ (\text{pmf-of-set} \ A) = \text{pmf-of-set} \ \{ ..<\text{card} \ A \}

\textbf{unfolding \text{rank-def} by (intro map-pmf-of-set-bij-betw bij-betw-linorder-rank[of UNIV]) auto}
also have \{..<\text{card } A\} = \{..n\} by (auto simp: card)  
also have \text{pmf-of-set} \ldots \Rightarrow \exists f = \text{rqs-cost} (\text{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 \text{expected-ipl-random-bst-eq}:  
  assumes \text{finite } A  
  shows \text{measure-pmf.expectation} (\text{random-bst } A) \text{ ipl} = \text{rqs-cost-exp} (\text{card } A)  
proof –  
  have \text{measure-pmf.expectation} (\text{random-bst } A) \text{ ipl} = 
    \text{measure-pmf.expectation} (\text{map-pmf ipl} (\text{random-bst } A)) \text{ real} by simp  
  also from \text{assms} have \text{map-pmf ipl} (\text{random-bst } A) = \text{rqs-cost} (\text{card } A)  
    by (rule \text{map-pmf-random-bst-eq-rqs-cost})  
  also have \text{measure-pmf.expectation} \ldots \text{ real} = \text{rqs-cost-exp} (\text{card } A)  
    by (rule \text{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 \text{expected-ipl-random-bst-eq}':  
  assumes \text{finite } A  
  shows \text{measure-pmf.expectation} (\text{random-bst } A) \text{ ipl} = 
    2 \ast \text{real} (\text{card } A + 1) \ast \text{harm} (\text{card } A) - 4 \ast \text{real} (\text{card } A)  
  by (simp add: \text{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.
