Expected Shape of Random Binary Search Trees

Manuel Eberl

March 17, 2025

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

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 $\{(x,\ y).\ x \le (y :: 'a :: linorder)\}$
⟨*proof*⟩

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

lemma *max-power-distrib-right*:
fixes $a :: 'a ::$ *linordered-semidom*
shows $a > 1 \Longrightarrow max\ (a \mathbin{\hat{}} b)\ (a \mathbin{\hat{}} c) = a \mathbin{\hat{}} max\ b\ c$
⟨*proof*⟩

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

lemma *card-set-tree-bst*: *bst* $t \Longrightarrow$ *card* (*set-tree* t) = *size* t
⟨*proof*⟩

lemma *pair-pmf-cong*:
$p = p' \Longrightarrow q = q' \Longrightarrow$ *pair-pmf* $p\ q$ = *pair-pmf* $p'\ q'$
⟨*proof*⟩

lemma *expectation-add-pair-pmf*:
fixes $f :: 'a \Rightarrow 'c::\{banach,\ second\text{-}countable\text{-}topology\}$
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-pmf.expectation $p\ f$ + *measure-pmf.expectation* $q\ g$
⟨*proof*⟩

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* $\leftarrow$ *xs*. *y* $<$ *x*]) *x* (*bst-of-list* [*y* $\leftarrow$ *xs*. *y* $>$ *x*])

lemma *bst-of-list-eq-Leaf-iff* [*simp*]: *bst-of-list xs* = *Leaf* $\longleftrightarrow$ *xs* = []
  ⟨*proof*⟩

lemma *bst-of-list-snoc* [*simp*]:
  bst-of-list (*xs* @ [*y*]) = *Tree-Set.insert y* (*bst-of-list xs*)
  ⟨*proof*⟩

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

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*: *bst-of-list xs* = *fold Tree-Set.insert xs Leaf*
  ⟨*proof*⟩

lemma *size-bst-insert*: *x* $\notin$ *set-tree t* $\Longrightarrow$ *size* (*Tree-Set.insert x t*) = *Suc* (*size t*)
  ⟨*proof*⟩

lemma *set-bst-insert* [*simp*]: *set-tree* (*Tree-Set.insert x t*) = *insert x* (*set-tree t*)
  ⟨*proof*⟩

lemma *set-bst-of-list* [*simp*]: *set-tree* (*bst-of-list xs*) = *set xs*
  ⟨*proof*⟩

lemma *size-bst-of-list-distinct* [*simp*]:
  assumes *distinct xs*
  shows *size* (*bst-of-list xs*) = *length xs*
  ⟨*proof*⟩

lemma *strict-mono-on-imp-less-iff*:
  assumes *strict-mono-on A f x* $\in$ *A y* $\in$ *A*
  shows *f x* $<$ (*f y* :: *'b* :: *linorder*) $\longleftrightarrow$ *x* $<$ (*y* :: *'a* :: *linorder*)
  ⟨*proof*⟩

lemma *bst-of-list-map*:
  fixes *f* :: *'a* :: *linorder* $\Rightarrow$ *'b* :: *linorder*
  assumes *strict-mono-on A f set xs* $\subseteq$ *A*

shows *bst-of-list* (*map f xs*) = *map-tree f* (*bst-of-list xs*)
⟨*proof*⟩

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* ⇒ *'a tree pmf* **where**
random-bst A =
(*if* ¬*finite A* ∨ *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*)
})
⟨*proof*⟩
termination ⟨*proof*⟩

declare *random-bst.simps* [*simp del*]

lemma *random-bst-empty* [*simp*]: *random-bst* {} = *return-pmf Leaf*
⟨*proof*⟩

lemma *set-pmf-random-permutation* [*simp*]:
finite A ⟹ *set-pmf* (*pmf-of-set* (*permutations-of-set A*)) = {*xs*. *distinct xs* ∧ *set xs* = *A*}
⟨*proof*⟩

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*))
⟨*proof*⟩

lemma *finite-set-random-bst* [*simp*, *intro*]:
finite A ⟹ *finite* (*set-pmf* (*random-bst A*))
⟨*proof*⟩

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

lemma *random-bst-singleton* [*simp*]: *random-bst* $\{x\}$ = *return-pmf* (*Node Leaf x Leaf*)
⟨*proof*⟩

lemma *size-random-bst*:
assumes $t \in$ *set-pmf* (*random-bst A*) *finite A*
shows *size t* = *card A*
⟨*proof*⟩

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*⟩

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*⟩

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* ^ (*height t* − *1*))

lemma *eheight-Leaf* [*simp*]: *eheight Leaf* = *0*
⟨*proof*⟩

lemma *eheight-Node-singleton* [*simp*]: *eheight* (*Node Leaf x Leaf*) = *1*
⟨*proof*⟩

lemma *eheight-Node*:
$l \neq$ *Leaf* ∨ $r \neq$ *Leaf* ⟹ *eheight* (*Node l x r*) = *2* ∗ *max* (*eheight l*) (*eheight r*)
⟨*proof*⟩

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

definition *eheight-exp* :: *nat* ⇒ *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* ← *pmf-of-set* {..<*n*}; *h1* ← *eheight-rbst k*; *h2* ← *eheight-rbst* (*n* − *k* − *1*);
      return-pmf (*2* ∗ *max h1 h2*)}
  ⟨*proof*⟩

lemma *Leaf-in-set-random-bst-iff*:
  assumes *finite A*
  shows *Leaf* ∈ *set-pmf* (*random-bst A*) ⟷ *A* = {}
⟨*proof*⟩

lemma *eheight-rbst*:
  assumes *finite A*
  shows *eheight-rbst* (*card A*) = *map-pmf eheight* (*random-bst A*)
⟨*proof*⟩

lemma *finite-pmf-set-eheight-rbst* [*simp*, *intro*]: *finite* (*set-pmf* (*eheight-rbst n*))
⟨*proof*⟩

lemma *eheight-exp-0* [*simp*]: *eheight-exp 0* = *0*
  ⟨*proof*⟩

lemma *eheight-exp-1* [*simp*]: *eheight-exp* (*Suc 0*) = *1*
  ⟨*proof*⟩

lemma *eheight-exp-reduce-bound*:
  assumes *n* > *1*
  shows *eheight-exp n* ≤ *4* / *n* ∗ (∑*k*<*n*. *eheight-exp k*)
⟨*proof*⟩

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

lemma *eheight-exp-bound*: *eheight-exp n* ≤ *real* ((*n* + *3*) *choose 3*) / *4*
⟨*proof*⟩

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*⟩

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*: (λ*x*::*real*. *ln* (*x* + *c*)) ∈ *O*(*ln*)
⟨*proof*⟩

corollary *height-expectation-bigo*: *height-exp-approx* ∈ *O*(*ln*)
⟨*proof*⟩

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* ⇒ *'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*))

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 $(\sum x \in \mathit{set\text{-}tree}\ t.\ \mathit{lookup\text{-}cost}\ x\ t) = \mathit{ipl}\ t$
⟨*proof*⟩

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* $t \neq$ *Leaf*
  shows $\mathit{measure\text{-}pmf.expectation}\ (\mathit{pmf\text{-}of\text{-}set}\ (\mathit{set\text{-}tree}\ t))\ (\lambda x.\ \mathit{lookup\text{-}cost}\ x\ t) = \mathit{ipl}\ t\ /\ \mathit{size}\ t$
  ⟨*proof*⟩

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 $\mathit{ipl}\ \langle l, x, r\rangle = \mathit{ipl}\ l + \mathit{size}\ l + \mathit{ipl}\ r + \mathit{size}\ r$. 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 $\mathit{map\text{-}pmf}\ \mathit{ipl}\ (\mathit{random\text{-}bst}\ A) = \mathit{rqs\text{-}cost}\ (\mathit{card}\ A)$
⟨*proof*⟩

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

corollary *expected-ipl-random-bst-eq*:
  assumes *finite A*
  shows $\mathit{measure\text{-}pmf.expectation}\ (\mathit{random\text{-}bst}\ A)\ \mathit{ipl} = \mathit{rqs\text{-}cost\text{-}exp}\ (\mathit{card}\ A)$
⟨*proof*⟩

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

corollary *expected-ipl-random-bst-eq'*:
  assumes *finite A*
  shows $\mathit{measure\text{-}pmf.expectation}\ (\mathit{random\text{-}bst}\ A)\ \mathit{ipl} = 2 * \mathit{real}\ (\mathit{card}\ A + 1) * \mathit{harm}\ (\mathit{card}\ A) - 4 * \mathit{real}\ (\mathit{card}\ A)$
  ⟨*proof*⟩

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.