# Verified Analysis of Random Trees

Manuel Eberl, Maximilian Haslbeck, Tobias Nipkow

Technische Universität München, 85748 Garching bei München, Germany

**Abstract** This work is a case study of the formal verification and complexity analysis of some famous probabilistic algorithms and data structures in the proof assistant Isabelle/HOL: the expected number of comparisons in randomised quicksort, the relationship between randomised quicksort and average-case deterministic quicksort, the expected shape of an unbalanced random Binary Search Tree, and the expected shape of a Treap. The last two have, to our knowledge, not been analysed in a theorem prover before and the last one is of particular interest because it involves continuous distributions.

## 1 Introduction

This paper is about the verified analysis of a number of classic probabilistic algorithms and data structures related to binary search trees. It continues a research programme to formalise classic data structure analyses [1,2,3], especially for binary search trees, by adding randomisation. The key novel contributions of the paper are readable (but see conclusion) formalised analyses of

- the *precise* expected number of comparisons in randomised quicksort
- the relationship between the average-case behaviour of deterministic quicksort and the distribution of randomized quicksort
- the expected path length and height of a random binary search tree
- the expected shape of a treap, which involves *continuous* distributions.

Algorithms are shallowly embedded and expressed in the Giry monad, which allows for a natural and high-level presentation. The verification has been carried out in Isabelle/HOL [4,5].

After an introduction to probability theory in Isabelle/HOL, the core of the paper consists of three sections that analyse quicksort, random binary search trees and treaps. The corresponding formalisations can be found in the *Archive of Formal Proof* [6,7,8].

## 2 Probability Theory in Isabelle/HOL

### 2.1 Measures and PMFs

The basics for measure theory (and thereby probability theory) in Isabelle/HOL were laid by Hölzl [9]. It is highly general and flexible, allowing also measures with

uncountably infinite support (e. g. the normal distributions on the reals), and has been used for a number of large formalisation projects related to randomisation, e. g. Ergodic theory [10], compiling functional programs to densities [11], Markov chains and decision processes [12], and cryptographic algorithms [13].

Initially we shall only consider probability distributions over *countable* sets. These are captured in Isabelle as *probability mass functions* (PMFs). A PMF is simply the function that assigns a probability to each element, with the property that the probabilities are non-negative and sum up to 1. For any HOL type $\alpha$, the type $\alpha$ pmf denotes the type of all probability distributions over values of type $\alpha$ with countable support.

Working with PMFs is quite pleasant, since we do not have to worry about measurability of sets and functions. Since everything is countable, we can always choose the power set as the measurable space, so everything is always trivially measurable.

However, later, we will also need continuous distributions. For this, there exists a type $\alpha$ measure, which describes a measure-theoretic measure over elements of type $\alpha$. Such a measure is formally a triple of a carrier set $\Omega$, a $\sigma$-algebra on $\Omega$ (which we call the set of measurable sets), and a measure function $\mu : \alpha \to$ ennreal, where ennreal is the type of non-negative real numbers $\mathbb{R}_{\geq 0} \cup \{\infty\}$. Of course, since we only consider probability measures here, our measures will always return values between 0 and 1.

For these general measures (which are only relevant for Section 5), there is the problem that we often need to annotate the corresponding $\sigma$-algebras and prove that everything we do with a distribution is in fact measurable. These details are unavoidable on a formal level, but typically very uninteresting to a human: There is usually a 'natural' choice for these $\sigma$-algebras and any set or operation that can be written down is typically measurable in some adequate sense. Therefore, for the sake of readability, we will drop everything related to measurability in this presentation.

For an overview of the notation that we use for PMFs and general measures, see Table 2.1. We allow ourselves some more notational freedoms in this paper, but these are only of syntactical nature to make the presentation easier without introducing additional notation.

### 2.2 The Giry Monad

Specifying probabilistic algorithms compositionally requires a way to express sequential composition of randomised choice. The standard way to do this is the *Giry monad* [14]. A detailed explanation of this (especially in the context of Isabelle/HOL) can be found in an earlier paper by Eberl *et al.* [11]. For the purpose of this paper, let us only say that the Giry monad provides functions

$$\text{return} :: \alpha \to \alpha \text{ pmf} \qquad \text{bind} :: \alpha \text{ pmf} \to (\alpha \to \beta \text{ pmf}) \to \beta \text{ pmf}$$

(and analogously for measure) where return $x$ gives us a singleton distribution where $x$ is chosen with probability 1 and bind $p$ $f$ composes two distributions

| PMFs | Measures | Meaning |
|---|---|---|
| pmf $p$ $x$ | | probability of $x$ in distribution $p$ |
| set_pmf $p$ | | support of $p$, i. e. $\{x \mid p(x) > 0\}$ |
| measure_pmf.prob $p$ $X$ | emeasure $M$ $X$ | probability of set $X$ |
| measure_pmf.expectation $p$ $f$ | expectation $M$ $f$ | expectation of $f :: \alpha \to \mathbb{R}$ |
| map_pmf $g$ $p$ | distr $g$ $M$ | image measure under $g :: \alpha \to \beta$ |
| pmf_of_set $A$ | uniform_measure $A$ | uniform distribution over $A$ |
| pair_pmf $p$ $q$ | $M \otimes N$ | binary product measure |
| | $\bigotimes_{x \in A} M(x)$ | indexed product measure |

**Figure 1.** Basic operations on PMFs and general measures. The variables $p :: \alpha$ pmf and $M :: \alpha$ measure denote an arbitrary PMF (resp. measure).

in the intuitive sense of 'Choose $x$ from the distribution $p$, then return a value chosen from $f(x)$'.

For better readability, Isabelle supports a Haskell-like do-notation as syntactic sugar for bind operations where e. g.

$$\text{bind } A\ (\lambda x.\ \text{bind } B\ (\lambda y.\ \text{return } (x + y)))$$

can be written succinctly as

$$\textbf{do } \{x \leftarrow A;\ y \leftarrow B;\ \text{return } (x + y)\}\ .$$

## 3 Quicksort

We now show how to define and analyse quicksort [15,16] (in its functional representation) within this framework. Since all randomisation is discrete, we can restrict ourselves to PMFs for now.

For the sake of simplicity (and because it relates to binary search trees, which we will treat later), we restrict ourselves to the case of sorting a list that has no repeated elements. (See the end of this section for more detail)

As is well known, quicksort has quadratic worst-case performance if the pivot is chosen poorly. Picking the true median as a pivot would solve this, but is unpractical. A simple alternative is to choose the pivot randomly, which is the variant that we shall analyse here first.

### 3.1 Randomised Quicksort

Intuitively, the justification for the good performance of randomised quicksort is that a random pivot will usually not be among the most extreme values of the list, but somewhere in the middle, so, on average, we reduce the size of the lists significantly in every recursion step.

To make this more rigorous, let us first look at the definition of the algorithm in Isabelle:

**Definition 1.** *Randomised quicksort*

```
rquicksort R xs =
  if xs = [] then
    return []
  else do {
    i ← pmf_of_set {0 ... |xs| − 1}
    let x = xs ! i
    let xs' = delete_index i xs
    ls ← rquicksort R [y | y ← xs', (y, x) ∈ R]
    rs ← rquicksort R [y | y ← xs', (y, x) ∉ R]
    return (ls @ [x] @ rs)
  }
```

Here, @ denotes list concatenation and $xs\,!\,i$ denotes the $i$-th element of the list $xs$, where $0 \leq i < |xs|$. Function delete_index removes the $i$-th element of a list. Parameter $R$ is a linear order represented as a set of pairs.

It is easy to prove that all the lists that can be returned by the algorithm are sorted w. r. t. $R$. To analyse the running time of the algorithm, the actual Isabelle definition of the algorithm was extended to also count the number of element comparisons made, i. e. it returns a ($\alpha$ list $\times$ nat) pmf. The base case makes 0 comparisons and the recursive case makes $|xs| - 1 + n_1 + n_2$ comparisons, where $n_1$ and $n_2$ are the comparisons made by the recursive calls. It would be easy to encapsulate the counting in a resource monad (we have done this elsewhere [3] for more complex code), but it is hardly worth the effort here.

For an element $x$ and some list $xs$, we call the number of elements of $xs$ smaller than $x$ the *rank* of $x$ w. r. t. $xs$. In a list with distinct elements, it is clear that an element's index in the list or its rank w. r. t. that list uniquely identify it, so choosing an element uniformly at random, choosing its index uniformly at random, or choosing a rank uniformly at random are all interchangeable.

The length of $ls$ in the above algorithm is simply the rank $r$ of the pivot, and the length of $rs$ is simply $|xs| - 1 - r$, so choosing the pivot uniformly at random will mean that the length of $ls$ is also distributed uniformly between 0 and $|xs|-1$. It can then be seen that the distribution of the number of comparisons does not actually depend on the content of the list or the ordering $R$ at all, but only on the *length* of the list, and we can find the following recurrence for it:

**Definition 2.** *Cost of randomised quicksort*

```
rqs_cost 0 = return 0
rqs_cost (n + 1) =
  do {
    r ← pmf_of_set {0 ... n}
    a ← rqs_cost r
    b ← rqs_cost (n − r)
    return (n + a + b)
  }
```

For any list $xs$ with no repeated elements on which $R$ is a linear ordering, we can easily show the equation

$$\text{map\_pmf snd (rquicksort } R\ xs) \ = \ \text{rqs\_cost } |xs| \ ,$$

i. e. projecting out the number of comparisons from our cost-aware randomised quicksort yields the distribution specified by rqs_cost.

Due to the recursive definition of rqs_cost, we can easily show that its expected value—let us call it $Q(n)$—fulfils the characteristic recurrence

$$Q(n+1) = n + \frac{1}{n+1}\left(\sum_{i=0}^{n} Q(i) + Q(n-i)\right) \ ,$$

or, equivalently,

$$Q(n+1) = n + \frac{2}{n+1}\left(\sum_{i=0}^{n} Q(i)\right) \ .$$

This is often called the *quicksort recurrence*. Following a simple proof by Cichoń [17], we can turn this into a linear recurrence

$$\frac{Q(n+1)}{n+2} = \frac{2n}{(n+1)(n+2)} + \frac{Q(n)}{n+1}$$

which gives us (by telescoping)

$$\frac{Q(n)}{n+1} = 2\sum_{k=1}^{n} \frac{k-1}{k(k+1)} = 4H_{n+1} - 2H_n - 4$$

and thereby the closed-form solution

$$Q(n) = 2(n+1)H_n - 4n \ ,$$

where $H_n$ is the $n$-th harmonic number. From the Isabelle library we can use the well-known asymptotics $H_n \sim \ln n + \gamma$ (where $\gamma \approx 0.5772$ is the Euler–Mascheroni constant), we then have $Q(n) \sim 2n \ln n$. We have therefore shown that the expected number of comparisons is logarithmic.

Remember, however, that we only considered lists with no repeated elements. If there are repeated elements, the above algorithm can deteriorate to quadratic time. This can be fixed easily by using a three-way partitioning function instead, which makes things slightly more complicated since the number of comparisons made now depends on the *contents* of the list and not just its *length*. The only real difference in the cost analysis is that the lists in the recursive call no longer simply have lengths $r$ and $n - r - 1$, but can also be shorter if the pivot is contained in the list more than once. One can still show that the expected number of comparisons is at most $Q(n)$ in much the same way as before (and our entry [6] in the *Archive of Formal Proofs* does contain that proof), but we shall not go into more detail here.

Comparing our proof to the literature, note that both Cormen *et al.* [18] and Knuth [19] also restrict their analysis to distinct elements. Cormen *et al.* use a non-compositional approach with indicator variables and only derive the logarithmic upper bound; Knuth's analysis counts the detailed number of different operations made by a particular implementation of the algorithm in MIX and his general approach is very similar to the one presented here.

### 3.2 Average-Case of Non-Randomised Quicksort

The above results carry over directly to the average-case analysis of non-randomised quicksort (again, we will only consider distinct elements). Here, the pivot is chosen deterministically; we always choose the first element for simplicity:

**Definition 3.** *Deterministic quicksort and its cost*

$$\begin{aligned}&\text{quicksort } R\ [] = []\\ &\text{quicksort } R\ (x \# xs) = \text{quicksort } R\ [y \mid y \leftarrow xs, (y,x) \in R]\ @\\ &\quad [x]\ @\ \text{quicksort } R\ [y \mid y \leftarrow xs, (y,x) \notin R]\end{aligned}$$

$$\begin{aligned}&\text{qs\_cost } R\ [] = 0\\ &\text{qs\_cost } R\ (x \# xs) = |xs| +\\ &\quad \text{qs\_cost } R\ [y \mid y \leftarrow xs, (y,x) \in R] + \text{qs\_cost } R\ [y \mid y \leftarrow xs, (y,x) \notin R]\end{aligned}$$

Interestingly, the number of comparisons made on a randomly-permuted input list has exactly the same distribution as the number of comparisons in randomised quicksort from before. The underlying idea is that randomly permuting an input, the randomness can be 'deferred' to the first place where an element is actually inspected, meaning that choosing the first element of a randomly-permuted list still makes the pivot essentially random.

The formal proof of this starts by noting that choosing a random permutation of a non-empty finite set $A$ is the same as first choosing the first list element $x \in A$ uniformly at random and then choosing a random permutation of $A \setminus \{x\}$ as the remaining list, which allows us to pull out the pivot selection. Then, we note that taking a random permutation of $A \setminus \{x\}$ and then partitioning it into elements that are smaller and bigger than $x$ is the same as first partitioning the set $A \setminus \{x\}$ into $\{y \in A \mid (y,x) \in R\}$ and $\{y \in A \mid (y,x) \notin R\}$ and choosing random permutations of these sets independently.

This last step of commuting partitioning and drawing a random permutation is probably the most crucial one and we will need it again later, so we print the corresponding lemma in full here. Let partition $P\ xs$ be the function that splits the list $xs$ into the pair of subsequences of sequences that fulfil (resp. do not fulfil) $P$. Then we have:

**Lemma 1.** *Partitioning a randomly permuted list*

$$\begin{aligned}&\textbf{assumes } \text{finite } A\\ &\textbf{shows } \text{map\_pmf } (\text{partition } P)\ (\text{pmf\_of\_set } (\text{permutations\_of\_set } A)) =\\ &\qquad \text{pair\_pmf } (\text{pmf\_of\_set } (\text{permutations\_of\_set } \{x \in A.\ P\ x\}))\\ &\qquad\qquad (\text{pmf\_of\_set } (\text{permutations\_of\_set } \{x \in A.\ \neg P\ x\}))\end{aligned}$$

This lemma is easily proven directly by extensionality, i. e. fixing permutations $xs$ of $\{x \in A.\ P\ x\}$ and $ys$ of $\{x \in A.\ \neg P\ x\}$ and computing their probability in the two distributions and noting that they are the same.

With this, the proof of the following theorem is just a straightforward induction on the recursive definition of rqs_cost: For every linear order $R$ on a finite set $A$, we have:

**Theorem 1.** *Cost distribution of randomised quicksort*

map_pmf (qs_cost $R$) (pmf_of_set (permutations_of_set $A$)) = rqs_cost $|A|$

Therefore, the cost distribution of deterministic quicksort on a randomly-permuted list is the same as that of randomised quicksort. In particular, the results about the logarithmic expectation of rqs_cost carry over directly.

## 4 Random Binary Search Trees

### 4.1 Preliminaries

We now turn to another average-case complexity problem that is somewhat related to quicksort, but not in an obvious way. We consider node-labelled binary trees, which are defined by the algebraic datatype

$$\textbf{datatype}\ \alpha\ \text{tree} = \text{Leaf} \mid \text{Node}\ (\alpha\ \text{tree})\ \alpha\ (\alpha\ \text{tree})\ .$$

We denote Leaf by $\langle\rangle$ and Node $l\ x\ r$ by $\langle l, x, r\rangle$. When the values of the tree come from some linear order, we say that the tree is a *binary search tree* (BST) if, for every node with some element $x$, all elements in the left sub-tree are smaller than $x$ and all elements to the right are larger than $x$.

Inserting elements can be done by performing a search and, if the element is not already in the tree, adding a node at the Leaf at which the search ends. We denote this operation by bst_insert. Note that these are simple, plain BSTs with no balancing, and our analysis will focus on what happens when elements are inserted into them in *random order*. We call the result of adding elements from a set $A$ to an initially empty BST in random order a *random BST*. This can also be seen as a kind of 'average-case' analysis of BSTs.

To analyse random BSTs, let us first examine what happens when we have a list of elements and insert them into an empty BST from left to right; formally:

**Definition 4.** *Inserting a list of elements into a BST*

$$\text{bst\_of\_list}\ xs\ =\ \text{fold bst\_insert}\ xs\ \langle\rangle$$

where bst_insert inserts a single element into a BST.

The first element $x$ will become the root of the tree and will never move again. Similarly, the next element will become either the left or right child of $x$ and will then also never move again and so on. It is also clear that no elements greater than $x$ will end up in the left subtree of $x$ in any point of the process, and no elements smaller in the right subtree. This leads to the following recurrence for bst_of_list:

**Lemma 2.** *A recurrence for bst_of_list*

$$\begin{aligned}&\text{bst\_of\_list } [] \;=\; \langle\rangle\\ &\text{bst\_of\_list } (x \mathbin{\#} xs) \;=\\ &\quad \langle \text{bst\_of\_list } [y \mid y \leftarrow xs, y < x],\; x,\; \text{bst\_of\_list } [y \mid y \leftarrow xs, y > x]\rangle\end{aligned}$$

We can now define our notion of 'random BST':

**Definition 5.** *Random BSTs*

$$\begin{aligned}&\text{random\_bst } A \;=\\ &\quad \text{map\_pmf bst\_of\_list (pmf\_of\_set (permutations\_of\_set } A))\end{aligned}$$

By re-using Lemma 1, we easily get the following recurrence:

**Lemma 3.** *A recurrence for random BSTs*

$$\begin{aligned}&\text{random\_bst } A \;=\\ &\quad \textbf{if } A = \{\} \textbf{ then return } \langle\rangle \textbf{ else do } \{\\ &\qquad x \leftarrow \text{pmf\_of\_set } A\\ &\qquad l \leftarrow \text{random\_bst } \{y \in A \mid y < x\}\\ &\qquad r \leftarrow \text{random\_bst } \{y \in A \mid y > x\}\\ &\qquad \text{return } \langle l, x, r\rangle\\ &\quad \}\end{aligned}$$

We can now analyse some of the properties of such a random BST. In particular, we will look at the expected height and the expected internal path length, and we will start with the latter since it is easier.

### 4.2 Internal Path Length

The internal path length (IPL) is essentially the sum of the lengths of all the paths from the root of the tree to each node. Alternatively, one can think of it as summing all the *levels* of the nodes in the tree, where the root is on the 0-th level, its immediate children are on the first level etc.

One reason why this number is important is that it is related to the time it takes to access a random element in the tree: the number of steps required to access some particular element $x$ is the level of that element, so if one picks a random element in the tree, the average number of steps it takes to access it is exactly the IPL divided by the size of the tree.

The IPL can be defined recursively by noting that $\text{ipl } \langle\rangle = 0$ and $\text{ipl } \langle l, x, r\rangle = \text{ipl } l + \text{ipl } r + |l| + |r|$. With this, we can show the following theorem by a simple induction over the recurrence for random_bst:

**Theorem 2.** *The internal path length of a random BST*

$$\text{map\_pmf ipl (random\_bst } A) = \text{rqs\_cost } |A|$$

Thus, the IPL of a random BST has the exact same distribution as the number of comparisons in randomised quicksort, which we already analysed before. This analysis is also done by Ottman and Widmayer [20], who also note its similarity to the analysis of quicksort.

### 4.3 Height

The height is more difficult to analyse. By our definition, an empty tree (i. e. a leaf) has height 0, and the height of a non-empty tree is the maximum of the heights of its left and right subtree, plus one. Let then $H(n)$ denote the height of a random BST with $n$ nodes (it is easy to show that this only depends on the *number* of nodes and not their actual content).

The asymptotics of its expectation and variance were found by Reed [21], who showed that, if $H(n)$ is the height of a random BST with $n$ nodes, $\mathrm{E}[H(n)] = \alpha \ln n - \beta \ln \ln n + O(1)$ and $\mathrm{Var}[H(n)] \in O(1)$ where $\alpha \approx 4.311$ is the unique real solution of $\alpha \ln(2e/\alpha) = 1$ with $\alpha \geq 2$ and $\beta = 3\alpha/(2\alpha - 2) \approx 1.953$. The proof of this is quite intricate, so we will restrict ourselves to showing $\mathrm{E}[H(n)] \leq \frac{3}{\ln 2} \ln n \approx 4.328 \ln n$, which is enough to see that the expected height is logarithmic.

Let us now do some preliminary exploration of how we can analyse the expectation of $H(n)$ (we will make this more precise later). The base cases $H(0) = 0$ and $H(1) = 1$ are obvious. For any $n > 1$, the recursion formula for random_bst suggests:

$$\mathrm{E}[H(n)] = 1 + \frac{1}{n} \sum_{k=0}^{n-1} \mathrm{E}[\max(H(k), H(n-k-1))]$$

The term involving max is somewhat problematic, since the expectation of the maximum of two random variables is, in general, difficult to analyse. A relatively obvious bound is $\mathrm{E}[\max(A, B)] \leq \mathrm{E}[A] + \mathrm{E}[B]$, but that will only give us

$$\mathrm{E}[H(n)] \leq 1 + \frac{1}{n} \sum_{k=0}^{n-1} (\mathrm{E}[H(k)] + \mathrm{E}[H(n-k-1)])$$

and if we were to use this to derive an explicit upper bound on $\mathrm{E}[H(n)]$ by induction, we would only get the trivial upper bound $\mathrm{E}[H(n)] \leq n$.

A trick suggested e. g. by Cormen *et al.* [18] is to instead use the *exponential height* (which we shall denote by eheight) of the tree, which, in terms of our height, is defined as 0 for a leaf and $2^{\mathrm{height}(t)-1}$ for a non-empty tree. The advantage of this is that it decreases the relative error that we make when we bound $\mathrm{E}[\max(A, B)]$ with $\mathrm{E}[A] + \mathrm{E}[B]$: That error is precisely $\mathrm{E}[\min(A, B)]$, and when $A$ and $B$ are heights, these heights only differ by small amounts in many cases. However, even a difference of 1 in the height will lead to the relative error in the exponential height being at most $\frac{1}{2}$, and considerably less than that in many cases. This turns out to be enough to get a relatively precise upper bound.

Let $H'(n)$ be the exponential height of a random BST. Since $x \mapsto 2^x$ is convex, any upper bound on $H'(n)$ can be used to derive an upper bound on $H(n)$ by Jensen's inequality:

$$2^{\mathrm{E}[H(n)]} = 2 \cdot 2^{\mathrm{E}[H(n)-1]} \leq 2\mathrm{E}[2^{H(n)-1}] = 2\mathrm{E}[H'(n)]$$

Therefore, we have

$$\mathrm{E}[H(n)] \leq \log_2 \mathrm{E}[H'(n)] + 1 \ .$$

In particular, a polynomial upper bound for $\mathrm{E}[H'(n)]$ will directly imply a logarithmic upper bound for $\mathrm{E}[H(n)]$.

It remains to analyse $H'(n)$ and find a polynomial upper bound for it. As the first step, note that if $l$ and $r$ are not both empty, the exponential height fulfils the recurrence eheight $\langle l, x, r\rangle = 2 \cdot \max$ (eheight $l$) (eheight $r$). Combining this with the recurrence for random_bst, the following recurrence suggests itself for $H'(n)$:

**Definition 6.** *The exponential height of a random BST*

$$
\begin{aligned}
&\text{eheight\_rbst } 0 = \text{return } 0\\
&\text{eheight\_rbst } 1 = \text{return } 1\\
n > 1 \Longrightarrow\ &\text{eheight\_rbst } n =\\
&\quad \textbf{do } \{\\
&\qquad k \leftarrow \text{pmf\_of\_set } \{0 \ldots n-1\}\\
&\qquad h_1 \leftarrow \text{eheight\_rbst } k\\
&\qquad h_2 \leftarrow \text{eheight\_rbst } (n-k-1)\\
&\qquad \text{return } (2 \cdot \max\ h_1\ h_2)\\
&\quad \}
\end{aligned}
$$

Showing that this definition is indeed the correct one can be done by a straightforward induction following the definition of random_bst:

**Lemma 4.** *Correctness of eheight_rbst*

$$\text{finite } A \Longrightarrow \text{eheight\_rbst } |A| = \text{map\_pmf eheight } (\text{random\_bst } A)$$

Using this, we note that for any $n > 1$:

$$
\begin{aligned}
\mathrm{E}[H'(n)] &= \frac{2}{n}\sum_{k=0}^{n-1} \mathrm{E}[\max(H'(k), H'(n-k-1))]\\
&\le \frac{2}{n}\sum_{k=0}^{n-1} \mathrm{E}[H'(k) + H'(n-k-1)]\\
&= \frac{2}{n}\left(\sum_{k=0}^{n-1} \mathrm{E}[H'(k)] + \sum_{k=0}^{n-1} \mathrm{E}[H'(n-k-1)]\right) = \frac{4}{n}\sum_{k=0}^{n-1} \mathrm{E}[H'(k)]
\end{aligned}
$$

It remains to find a suitable polynomial upper bound to make the induction argument work out. If we had some polynomial $P(n)$ that fulfils $0 \le P(0)$, $1 \le P(1)$, and the recurrence $P(n) = \frac{4}{n}\sum_{k=0}^{n-1} P(k)$ , the above recursive estimate for $\mathrm{E}[H'(n)]$ would directly imply $\mathrm{E}[H'(n)] \le P(n)$ by induction. Cormen *et al.* give the following polynomial that satisfies all these conditions and makes everything work out nicely:

$$P(n) = \frac{1}{4}\binom{n+3}{3} = \frac{1}{24}(n+1)(n+2)(n+3)$$

To conclude, we have shown:

**Theorem 3.** *Asymptotic expectation of* $H(n)$

$$\mathrm{E}[H(n)] \;\le\; \log_2 \mathrm{E}[H'(n)] + 1 \;\le\; \log_2 P(n) + 1 \;\sim\; \frac{3}{\ln 2} \ln n$$

## 5 Treaps

As we have seen, binary search trees have the nice property that even without any explicit balancing, they tend to be fairly balanced if elements are inserted into them in random order. However, if, for example, elements are inserted in ascending order, the tree degenerates into a list and no longer has logarithmic height. One interesting way to prevent this is to use a *treap* [22] instead, which we shall introduce and analyse now.

### 5.1 Definition

A treap is a binary tree in which every node contains an element and a priority associated to that element, and the tree is a binary search tree w. r. t. the elements and a heap w. r. t. the priorities (i. e. the root is always the node with the lowest priority). Due to space constraints, we shall not go into how insertion of elements (denoted by ins) works, but it is fairly easy to implement.

An interesting consequence of the treap conditions is that, as long as all priorities are distinct, the shape of the treap is uniquely determined by the elements and their priorities. Since the subtrees of a treap must again be treaps, uniqueness follows by induction and we have the following simple algorithm for constructing the unique treap for a set:

**Lemma 5.** *Constructing the unique treap for a set* $A :: (\alpha \times \mathbb{R})$ *set where the second components are all distinct:*

$$\begin{array}{l}\mathrm{treap\_of}\ A\ = \\ \quad \textbf{if}\ A = \{\}\ \textbf{then}\ \langle\rangle\ \textbf{else} \\ \quad\quad \textbf{let}\ x = \mathrm{arg\_min\_on\ snd}\ A \\ \quad\quad \textbf{in}\ \langle \mathrm{treap\_of}\ \{y \in A \mid y < x\}, x, \mathrm{treap\_of}\ \{y \in A \mid y > x\}\rangle\end{array}$$

where arg_min_on $f$ $A$ is some $a \in A$ such that $f(a)$ is minimal on $A$. In our case the choice of $a$ is unique by assumption.

This is very similar to the recurrence for bst_of_list that we saw earlier: in fact, it is easy to prove that if we forget about the priorities in the treap, the resulting tree is exactly the same as if we first sorted the keys by increasing priority and then inserted them into a binary search tree in that order. Formally, we have:

**Lemma 6.** *Connection between treaps and BSTs. Let* $p$ *be an injective function that associates a priority to each element of a list* $xs$*. Then:*

$$\mathrm{map\_tree\ fst}\ (\mathrm{treap\_of}\ \{(x, p(x)) \mid x \in \mathrm{set}\ xs\}) = \mathrm{bst\_of\_list}\ (\mathrm{sort\_key}\ p\ xs)$$

*where sort_key sorts a list in ascending order w. r. t. the given priority function.*

*Proof.* By induction over $xs' := \text{sort\_key}\ p\ xs$, using the fact that sorting w. r. t. distinct priorities can be seen as a selection sort: $xs'$ consists of the unique element $x$ with minimum priority as its first element, followed by sort_key (remove1 $x$ $xs$), where remove1 deletes the first occurrence of an element in a list.

With this and Lemma 2, the recursion structure of the right-hand side is exactly the same as that of the treap_of from Lemma 5. □

This essentially allows us to build a BST that behaves as if we inserted the elements by ascending priority no matter in what order we actually inserted them. In particular, we can assign every element a *random* priority upon its insertion, which turns our treaps (a deterministic data structure for values of type $(\alpha \times \mathbb{R})$ set) into randomised treaps, which are a *randomised* data structure for values of type $\alpha$ that has the same distribution as a random BST with the same content.

A caveat is that for all the results so far, we have assumed that the priorities of two distinct elements are never the same, and, of course, without that assumption, we lose all these nice properties. If the priorities in our randomised treap are chosen from some discrete probability distribution, there will always be some non-zero probability that they are not distinct. Therefore, the literature usually draws them from some continuous distribution (e. g. real numbers between 0 and 1), since it makes the analysis much easier. We shall do the same here.[1]

The argument works like this:

1. Choosing the priority of each element randomly when we insert it is the same as choosing all the i. i. d. probabilities beforehand and then inserting the elements into the treap deterministically.
2. By the theorems above, this is the same as choosing the priorities i. i. d. at random, sorting the elements by increasing priority, and then inserting them into a BST in that order.
3. By symmetry considerations, choosing i. i. d. priorities for all elements and then looking at the linear ordering defined by these priorities is the same as choosing one of the $n!$ linear orderings uniformly at random.
4. Thus, inserting a list of elements into a randomised treap is the same as inserting the elements into a BST in random order.

### 5.2 The Measurable Space of Trees

One complication in formalising treaps that is typically not addressed in pen-and-paper accounts is that since we will randomise over priorities, we need to talk about continuous distributions of trees, i. e. distributions of type $(\alpha \times \mathbb{R})$ tree measure. For example, if we insert the element $x$ into an empty treap

[1] In fact, any non-discrete probability distribution works, where by 'non-discrete' we mean that all singleton sets have probability 0. In the formalisation, however, we restricted ourselves to the case of a uniform distribution over a real interval.

with a priority that is uniformly distributed between 0 and 1, we get a distribution of trees of the shape $\langle\langle\rangle, (x, p), \langle\rangle\rangle$ where $p$ is uniformly distributed between 0 and 1.

In order to be able to express this formally, we need a way to lift some measurable space $M$ to a measurable space $\mathcal{T}(M)$ of trees with elements from $M$ attached to their nodes. Of course, we cannot just pick *any* measurable space—in order for our treap operations to be well-defined, all the basic tree operations need to be measurable w. r. t. $\mathcal{T}(M)$; in particular:

- the constructors Leaf and Node, i. e. we need $\{\text{Leaf}\} \in \mathcal{T}(M)$ and Node must be $\mathcal{T}(M) \otimes \mathcal{M} \otimes \mathcal{T}(M)$–$\mathcal{T}(M)$-measurable
- the projection functions, i. e. selecting the value/left subtree/right subtree of a node; e. g. selecting a node's value must be $(\mathcal{T}(M) \setminus \{\langle\rangle\})$–$M$-measurable
- primitively recursive functions involving only measurable operations must also be measurable; we will need this to define e. g. the insertion operation

We construct such a measurable space by taking the $\sigma$-algebra that is generated by certain *cylinder sets*: consider a tree whose nodes each have an $M$-measurable set attached to them. Then this tree can be 'flattened' into the set of trees of the same shape where each node has a single value from the corresponding set in $t$ attached to it. Then we define $\mathcal{T}(M)$ to be the measurable space generated by all these cylinder sets, and prove that all the above-mentioned operations are indeed measurable.

### 5.3 Randomisation

In order to achieve a good height distribution on average, the priorities of a treap need to be chosen randomly. Since we do not know how many elements will be inserted into the tree in advance, we need to draw the priority to assign to an element when we insert it, i. e. the insertion operation is now a randomised operation:

**Definition 7.** *Randomised insertion into a treap*

$$\begin{aligned}&\text{rins} :: \alpha \to \alpha\ \text{treap} \to \alpha\ \text{treap measure}\\&\text{rins}\ x\ t = \textbf{do}\ \{p \leftarrow \text{uniform\_measure}\ \{0 \ldots 1\};\ \text{return}\ (\text{ins}\ x\ p\ t)\}\end{aligned}$$

Since we would like to analyse what happens when we insert a larger number of elements into an initially empty treap, we also define the following 'bulk insert' operation that inserts a list of elements into the treap from left to right:

$$\begin{aligned}&\text{rinss} :: \alpha\ \text{list} \to \alpha\ \text{treap} \to \alpha\ \text{treap measure}\\&\text{rinss}\ []\ t = \text{return}\ t\\&\text{rinss}\ (x \# xs)\ t = \textbf{do}\ \{t' \leftarrow \text{rins}\ x\ t;\ \text{rinss}\ xs t'\}\end{aligned}$$

Note that, from now on, we will assume that all the elements we insert are *distinct*. This is not really a restriction, since inserting duplicate elements does not change the tree, so we can just drop any duplicates from the list without

changing the result. Similarly, due to the uniqueness property of treaps, after deleting an element, the resulting treap is exactly the same as if the element had never been inserted in the first place, so even though we only analyse the case of insertions without duplicates, this extends to any sequence of insertion and deletion operations (although we do not show this explicitly).

The main result, as sketched above, shall be that after inserting a certain number of distinct elements into the treap and then forgetting about their priorities, we get a BST that is distributed identically to a random BST with the same elements, i. e. the treap behaves as if we had inserted the elements in random order. Formally, this can be expressed as:

**Theorem 4.** *Main result on randomised treaps and random BSTs*

$$\text{distr } (\text{rinss } xs \langle\rangle) \ (\text{map\_tree fst}) = \text{random\_bst } (\text{set } xs)$$

*Proof* Let $\mathcal{U}$ denote the uniform distribution of real numbers between 0 and 1 and $\mathcal{U}^A$ a vector of i. i. d. distributions $\mathcal{U}$, indexed by $A$:

$$\mathcal{U} := \text{uniform\_measure } \{0 \dots 1\} \qquad \mathcal{U}^A := \bigotimes_A \mathcal{U}$$

For the first step, we show that our bulk-insertion operation rinss is equivalent to *first* choosing random priorities for *all* the elements at once and then inserting them all (with their respective priority) deterministically:

$$\begin{aligned} \text{rinss } xs \ t &= \text{distr } \mathcal{U}^{\text{set } xs} \ (\lambda p.\ \text{foldl } (\lambda t\ x.\ \text{ins } x\ (p(x))\ t)\ t\ xs) \\ &= \text{distr } \mathcal{U}^{\text{set } xs} \ (\lambda p.\ \text{treap\_of } [(x, p(x)) \mid x \leftarrow xs]) \end{aligned}$$

The first equality is proved by induction over $xs$, pulling out one insertion in the induction step and moving the choice of the priority to the front. This is intuitively obvious, but the formal proof is rather tedious nevertheless, mostly due to the issue of having to prove measurability in every single step. The second equality follows by uniqueness of treaps.

Next, we note that the priority function returned by $\mathcal{U}^{\text{set } xs}$ is almost surely injective, so we can apply Lemma 6 and get:

$$\begin{aligned} &\text{distr } (\text{rinss } xs \langle\rangle) \ (\text{map\_tree fst}) = \\ &\qquad \text{distr } \mathcal{U}^{\text{set } xs} \ (\lambda p.\ \text{bst\_of\_list } (\text{sort\_key } p\ xs) \end{aligned}$$

The next key lemma is the following, which holds for any finite set $A$:

$$\text{distr } \mathcal{U}^A \ (\text{linorder\_from\_keys } A) = \text{uniform\_measure } (\text{linorders\_on } A)$$

This essentially says that choosing priorities for all elements of $A$ and then looking at the ordering on $A$ that these priorities induce will give us a uniform distribution on all the $|A|!$ linear ordering relations on $A$. In particular, this means that that relation will be linear with probability 1, i. e. the priorities will almost surely be injective. The proof of this is a simple symmetry argument:

given any two linear orderings $R$ and $R'$ of $A$, we can find some permutation $\pi$ of $A$ that maps $R'$ to $R$. However, $\mathcal{U}^A$ is stable under permutation. Therefore, $R$ and $R'$ have the same probability, and since this holds for all $R$, $R'$, the distribution must be the uniform distribution.

This brings us to the last step: Sorting our list of elements by random priorities and then inserting them to a BST is the same as inserting them in random order (in the sense of choosing a randomly-permuted list and inserting them in that order):

$$\begin{aligned}&\text{distr } \mathcal{U}^{\,\text{set } xs}\ (\lambda p.\ \text{bst\_of\_list } (\text{sort\_key } p\ xs) = \\ &\qquad \text{distr } (\text{uniform\_measure } (\text{permutations\_of\_set } (\text{set } xs))) \text{ bst\_of\_list}\end{aligned}$$

Here we use the fact that priorities chosen uniformly at random induce a uniformly random linear order, and that sorting a list with such an order produces permutations of that list uniformly at random. The proof of this is little more than rearranging and using some obvious lemmas on sort_key etc. Now the right-hand side is exactly the definition of a random BST (up to a conversion between pmf and measure), so that concludes the proof. □

## 6 Related Work

The earliest analysis of randomised algorithms in a theorem prover was probably by Hurd [23] in the HOL system, who modelled randomised algorithms by assuming the existence of an infinite sequence of random bits which programs can consume. He used this to formalise the Miller–Rabin primality test.

Audebaud and Paulin-Mohring [24] created a shallowly-embedded formalisation of (discrete) randomised algorithms in Coq and demonstrate its usage on two examples. Barthe *et al.* [25] use this framework to implement the *CertiCrypt* system to write machine-checked cryptographic proofs for a deeply embedded imperative language. Petcher and Morrisett [26] developed a similar framework but based on a monadic embedding. A similar framework was developed for Isabelle/HOL by Lochbihler [27].

The expected running time of randomised quicksort (with possibly repeated elements) was first analysed in a theorem prover by van der Weegen and McKinna [28] using Coq. They prove the upper bound $2n\lceil\log_2 n\rceil$, whereas we actually proved the closed-form $2(n+1)H_n - 4n$ and its precise asymptotics. Although their paper's title mentions 'average-case complexity', they only treat the expected running time of the randomised algorithm in their paper. They did, however, later add a separate proof of an upper bound for the average-case of deterministic quicksort to their GitHub repository. Unlike we, they allow lists to have repeated elements even in the average case, but they prove the expectation bounds separately and independently, while we assumed no repeated elements, but showed something stronger, namely that the distributions are exactly the same, allowing us to reuse the results from the randomised case.

Kaminski *et al.* [29] presented a Hoare-style calculus for analysing the expected running time of imperative programs and use it to analyse a 1D random

walk and the Coupon Collector's problem. Hölzl [30] formalised this approach in Isabelle and found a mistake in their proof of the random walk in the process.

Outside theorem provers, other approaches exist to automate the analysis of such algorithms: Probabilistic model checkers like PRISM [31] can check safety properties and compute expectation bounds. The $\Lambda\gamma^{\Omega}$ system by Flajolet *et al.* [32] performs fully automatic analysis of average-case running time for a restricted variety of (deterministic) programs. Chatterjee *et al.* [33] developed a method for deriving expected running time bounds of the shape $O(\ln n)$, $O(n)$, or $O(n \ln n)$ automatically and applied it to a number of interesting examples, including quicksort.

## 7 Future Work

We have closed a number of important gaps in the formalisation of classic algorithms around binary search trees, including the thorny treap that requires measure theory. Up to that point we claim that these formalisations are readable (the definitions thanks to the Giry monad, the proofs thanks to Isar [34]), but for treaps this becomes debatable: the issue of measurability makes proofs and definitions significantly more cumbersome and less readable. Existing automation for measurability is already very helpful, but there is still room for improvement. This concerns also the construction of the measurable space of trees, which generalises to other algebraic data types and could be automated.

So far all our work has been at the functional level. It would be desirable to refine it to an imperative level in a modular way. The development of the necessary theory and infrastructure is future work.

**Acknowledgement.** This work was funded by DFG grant NI 491/16-1. We thank Johannes Hölzl and Andreas Lochbihler for helpful discussions on probability theory in Isabelle, Johannes Hölzl for his help with the construction of the tree space $\mathcal{T}(M)$, and Bohua Zhan and Maximilian Paul Louis Haslbeck for comments on a draft.

## References


1. Nipkow, T.: Amortized complexity verified. In Urban, C., Zhang, X., eds.: Interactive Theorem Proving (ITP 2015). Volume 9236 of LNCS., Springer (2015) 310–324
2. Nipkow, T.: Automatic functional correctness proofs for functional search trees. In Blanchette, J., Merz, S., eds.: Interactive Theorem Proving (ITP 2016). Volume 9807 of LNCS., Springer (2016) 307–322
3. Nipkow, T.: Verified root-balanced trees. In Chang, B.Y.E., ed.: Asian Symposium on Programming Languages and Systems, APLAS 2017. Volume 10695 of LNCS., Springer (2017) 255–272
4. Nipkow, T., Paulson, L., Wenzel, M.: Isabelle/HOL — A Proof Assistant for Higher-Order Logic. Volume 2283 of LNCS. Springer (2002)
5. Nipkow, T., Klein, G.: Concrete Semantics with Isabelle/HOL. Springer (2014) `http://concrete-semantics.org`.
6. Eberl, M.: The number of comparisons in QuickSort. Archive of Formal Proofs (March 2017) `http://isa-afp.org/entries/Quick_Sort_Cost.html`, Formal proof development.
7. Eberl, M.: Expected shape of random binary search trees. Archive of Formal Proofs (April 2017) `http://isa-afp.org/entries/Random_BSTs.html`, Formal proof development.
8. Haslbeck, M., Eberl, M., Nipkow, T.: Treaps. Archive of Formal Proofs (March 2018) `http://isa-afp.org/entries/Treaps.html`, Formal proof development.
9. Hölzl, J., Heller, A.: Three chapters of measure theory in Isabelle/HOL. In: Interactive Theorem Proving - Second International Conference, ITP 2011, Berg en Dal, The Netherlands, August 22–25, 2011. Proceedings. (2011) 135–151
10. Gouëzel, S.: Ergodic theory. Archive of Formal Proofs (December 2015) `http://isa-afp.org/entries/Ergodic_Theory.html`, Formal proof development.
11. Eberl, M., Hölzl, J., Nipkow, T.: A verified compiler for probability density functions. In Vitek, J., ed.: Proceedings of the 24th European Symposium on Programming, Springer Berlin Heidelberg (2015) 80–104
12. Hölzl, J.: Markov chains and markov decision processes in Isabelle/HOL. Journal of Automated Reasoning (2017)
13. Basin, D.A., Lochbihler, A., Sefidgar, S.R.: Crypthol: Game-based proofs in higher-order logic. Cryptology ePrint Archive, Report 2017/753 (2017) `https://eprint.iacr.org/2017/753`.
14. Giry, M.: A categorical approach to probability theory. In: Categorical Aspects of Topology and Analysis. Volume 915 of Lecture Notes in Mathematics. Springer Berlin Heidelberg (1982) 68–85
15. Hoare, C.A.R.: Quicksort. The Computer Journal **5**(1) (1962) 10
16. Sedgewick, R.: The analysis of Quicksort programs. Acta Inf. **7**(4) (December 1977) 327–355
17. Cichoń, J.: Quick Sort – average complexity. `http://cs.pwr.edu.pl/cichon/Math/QSortAvg.pdf`
18. Cormen, T.H., Stein, C., Rivest, R.L., Leiserson, C.E.: Introduction to Algorithms. 2nd edn. McGraw-Hill Higher Education (2001)
19. Knuth, D.E.: The Art of Computer Programming, Volume 3: Sorting and Searching. Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA (1998)

20. Ottmann, T., Widmayer, P.: Algorithmen und Datenstrukturen, 5. Auflage. Spektrum Akademischer Verlag (2012)
21. Reed, B.: The height of a random binary search tree. J. ACM **50**(3) (May 2003) 306–332
22. Seidel, R., Aragon, C.R.: Randomized search trees. Algorithmica **16**(4) (Oct 1996) 464–497
23. Hurd, J.: Formal Verification of Probabilistic Algorithms. PhD thesis, University of Cambridge (2002)
24. Audebaud, P., Paulin-Mohring, C.: Proofs of randomized algorithms in Coq. Sci. Comput. Program. **74**(8) (2009) 568–589
25. Barthe, G., Grégoire, B., Béguelin, S.Z.: Formal certification of code-based cryptographic proofs. In: Proceedings of the 36th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2009. (2009) 90–101
26. Petcher, A., Morrisett, G.: The foundational cryptography framework. In Focardi, R., Myers, A.C., eds.: Principles of Security and Trust - 4th International Conference, POST 2015. Volume 9036 of Lecture Notes in Computer Science., Springer (2015) 53–72
27. Lochbihler, A.: Probabilistic functions and cryptographic oracles in higher order logic. In Thiemann, P., ed.: Programming Languages and Systems (ESOP 2016). Volume 9632 of LNCS., Springer (2016) 503–531
28. van der Weegen, E., McKinna, J. In: A Machine-Checked Proof of the Average-Case Complexity of Quicksort in Coq. Springer Berlin Heidelberg, Berlin, Heidelberg (2009) 256–271
29. Kaminski, B.L., Katoen, J.P., Matheja, C., Olmedo, F.: Weakest precondition reasoning for expected run—times of probabilistic programs. In: Proceedings of the 25th European Symposium on Programming Languages and Systems - Volume 9632, New York, NY, USA, Springer-Verlag New York, Inc. (2016) 364–389
30. Hölzl, J.: Formalising semantics for expected running time of probabilistic programs. In Blanchette, J.C., Merz, S., eds.: Interactive Theorem Proving (ITP 2016), Springer (2016) 475–482
31. Kwiatkowska, M.Z., Norman, G., Parker, D.: Quantitative analysis with the probabilistic model checker PRISM. Electr. Notes Theor. Comput. Sci. **153**(2) (2006) 5–31
32. Flajolet, P., Salvy, B., Zimmermann, P.: Lambda - upsilon - omega: An assistant algorithms analyzer. In: Applied Algebra, Algebraic Algorithms and Error-Correcting Codes, 6th International Conference, AAECC-6, Rome, Italy, July 4-8, 1988, Proceedings. (1988) 201–212
33. Chatterjee, K., Fu, H., Murhekar, A.: Automated recurrence analysis for almost-linear expected-runtime bounds. In: Computer Aided Verification - 29th International Conference, CAV 2017. (2017) 118–139
34. Wenzel, M.: Isabelle/Isar — A Versatile Environment for Human-Readable Formal Proof Documents. PhD thesis, Institut für Informatik, Technische Universität München (2002) `https://mediatum.ub.tum.de/node?id=601724`.