Verified Analysis of Random Binary Tree Structures

Manuel Eberl[0000−0002−4263−6571], Max W. Haslbeck[0000−0002−9900−5746], and Tobias Nipkow[0000−0003−0730−515X]

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. In particular, we consider 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 using a theorem prover before and the last one is of particular interest because it involves continuous distributions.

1 Introduction

This paper conducts verified analyses of a number of classic probabilistic algorithms and data structures related to binary search trees. It is part of a continuing 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 (with one caveat, discussed in the 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 randomised quicksort
- the expected path length and height of a random binary search tree
- the expected shape of a treap, which involves *continuous* distributions.

The above algorithms are shallowly embedded and expressed using the Giry monad, which allows for a natural and high-level presentation. All verifications were carried out in Isabelle/HOL [4,5].

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

2 Probability Theory in Isabelle/HOL

2.1 Measures and Probability Mass Functions

The foundation for measure theory (and thereby probability theory) in Isabelle/HOL was laid by Hölzl [9]. This approach is highly general and flexible, allowing also measures with uncountably infinite support (e. g. 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. In Isabelle, these are captured as *probability mass functions* (PMFs). A PMF is simply a 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 α, the type α *pmf* denotes the type of all probability distributions over values of type α with countable support.

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

Later, however, we will also need continuous distributions. For these, there exists a type α *measure*, which describes a measure-theoretic measure over elements of type α. Such a measure is formally defined as a triple consisting of a carrier set Ω, a σ-algebra on Ω (which we call the set of measurable sets), and a measure function $\mu : \alpha \to$ *ennreal*, where *ennreal* is the type of extended 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.

One problem with these general measures (which are only relevant for Section 5), is that we often need to annotate the corresponding σ-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 σ-algebras and any set or operation that can be written down is typically measurable in some adequate sense. For the sake of readability, we will therefore omit everything related to measurability in this presentation.

Table 2.1 gives an overview of the notation that we use for PMFs and general measures. Although we allow ourselves some more notational freedoms in this paper, these are purely syntactical changes designed 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

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

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

of Isabelle/HOL) can be found in an earlier paper by Eberl *et al.* [11]. For the purpose of this paper, we only need to know 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 α *measure*) where *return* x gives us the singleton distribution where x is chosen with probability 1 and *bind* p f composes two distributions in the intuitive sense of randomly choosing a value x according to the distribution p and then returning a value randomly chosen according to the distribution $f(x)$.

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 of the randomisation is discrete in this case, we can restrict ourselves to PMFs for the moment.

For the sake of simplicity (and because it relates to binary search trees, which we will treat later), we shall only treat the case of sorting lists without repeated elements. (See the end of this section for further discussion of this point.)

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

3.1 Randomised Quicksort

Intuitively, the good performance of randomised quicksort can be explained by the fact that a random pivot will usually not be among the most extreme values of the list, but somewhere in the middle, which means that, on average, the size of the lists is reduced significantly in every recursion step.

To make this more rigorous, let us 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 $0 \leq i < |xs|$. The *delete_index* function removes the i-th element of a list, and the parameter R is a linear ordering represented as a set of pairs.

It is easy to prove that all of the lists that can be returned by the algorithm are sorted w. r. t. R. To analyse its running time, its actual Isabelle definition was extended to also count the number of element comparisons made, i. e. to return an (α *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 numbers of comparisons made by the recursive calls. This could easily be encapsulated in a resource monad (as we have done elsewhere [3] for more complex code), but it is not worth the effort in this case.

For an element x and some list xs, we call the number of elements of xs that are smaller than x the *rank* of x w. r. t. xs. In lists with distinct elements, each element can clearly be uniquely identified by either its index in the list or its rank w. r. t. that list, so choosing an element uniformly at random, choosing its index uniformly at random, or choosing a rank uniformly at random are all interchangeable.

In the above algorithm, the length of ls 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 means that the length of ls is also distributed uniformly between 0 and $|xs| - 1$. From this, we can see 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 and a linear ordering R, we can easily show the equation

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

i. e. projecting out the number of comparisons from our cost-aware randomised quicksort yields the distribution given by *rqs_ cost*.

Due to the recursive definition of *rqs_ cost*, we can easily show that its expected value, which we denote by $Q(n)$, satisfies 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*. Cichoń [17] gave a simple way of solving this by turning it 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. We can use the well-known asymptotics $H_n \sim \ln n + \gamma$ (where $\gamma \approx 0.5772$ is the Euler–Mascheroni constant) from the Isabelle library and obtain $Q(n) \sim 2n \ln n$, which shows that the expected number of comparisons is logarithmic.

Remember, however, that we only considered lists with no repeated elements. If there are any repeated elements, the performance of the above algorithm can deteriorate to quadratic time. This can be fixed easily by using a three-way partitioning function instead, although this makes things slightly more complicated since the number of comparisons made now depends on the *content* 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. We 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 those in 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, whereas Knuth's analysis counts the detailed number of different operations made by a particular implementation of the algorithm in MIX. 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 lists with distinct elements). Here, the pivot is chosen deterministically; we always choose the first element for simplicity. This gives us the following definitions of quicksort and its cost:

Definition 3 (Deterministic quicksort and its cost).

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

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

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 when randomly permuting the input, the randomness can be 'deferred' to the first point where an element is actually inspected, which means 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 remainder of the list, allowing us to pull out the pivot selection. Then, we note that taking a random permutation of $A \setminus \{x\}$ and partitioning it into elements that are smaller and bigger than x is the same as first partitioning the

set $A \setminus \{x\}$ into $\{y \in A \setminus \{x\} \mid (y, x) \in R\}$ and $\{y \in A \setminus \{x\} \mid (y, x) \notin R\}$ and choosing random permutations of these sets independently.

This last step, which interchanges partitioning and drawing a random permutation, is probably the most crucial one and one that we will need again later, so we present the corresponding lemma in full here. Let *partition P xs* be the function that splits the list *xs* into the pair of sub-sequences that satisfy (resp. do not satisfy) the predicate P. Then, we have:

Lemma 1 (Partitioning a randomly permuted list).

assumes finite A
shows map_pmf (partition P) (pmf_of_set (permutations_of_set A)) =
pair_pmf (pmf_of_set (permutations_of_set $\{x \in A.\ P\ x\}$))
(pmf_of_set (permutations_of_set $\{x \in A.\ \neg P\ x\}$))

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 probabilities 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*:

Theorem 1 (Cost distribution of randomised quicksort). *For every linear order R on a finite set A, we have:*

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

Thus, the cost distribution of deterministic quicksort applied to 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, though not in an obvious way. We consider node-labelled binary trees, 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 have some linear ordering, we say that the tree is a *binary search tree* (BST) if, for every node with some element x, all of the values in the left sub-tree are smaller than x and all of the values in the right sub-tree 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, unbalanced BSTs and our analysis will focus on what happens when elements are inserted

into them in *random order*. We call the tree that results from adding elements of 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 insert a list of elements into an empty BST from left to right; formally:

Definition 4 (Inserting a list of elements into a BST).

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

Let x be the first element of the list. This element 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 sub-tree of x at any point in the process, and no elements smaller in the right sub-tree. This leads us to the following recurrence for *bst_of_list*:

Lemma 2 (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 formally define our notion of 'random BST':

Definition 5 (Random BSTs).

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

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 the sum of all the *level numbers* 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 equal to the number of that element's level, so if one chooses a random element in the tree, the average number of steps needed to access it is exactly the IPL divided by the size of the tree.

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

Theorem 2 (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 was also carried out by Ottman and Widmayer [20], who also noted its similarity to the analysis of quicksort.

4.3 Height

The height of a random BST 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 sub-trees, plus one. It is easy to show that the height distribution only depends on the *number* of elements and not their actual content, so let $H(n)$ denote the height of a random BST with n nodes.

The asymptotics of its expectation and variance were found by Reed [21], who showed that $\mathrm{E}[H(n)] = \alpha \ln n - \beta \ln \ln n + O(1)$ 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 that $\mathrm{E}[H(n)] \leq \frac{3}{\ln 2} \ln n \approx 4.328 \ln n$, which is enough to see that the expected height is logarithmic.

Before going into a precise discussion of the proof, let us first undertake a preliminary exploration of how we can analyse the expectation of $H(n)$. 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 *max* term 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] (which they attribute to Javed Aslam [22]) 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 $2^{height(t)-1}$ for a non-empty tree. The advantage of this is that it decreases the relative error that we make when we bound $\mathrm{E}[\max(A, B)]$ by $\mathrm{E}[A] + \mathrm{E}[B]$: this error is precisely $\mathrm{E}[\mathit{min}(A, B)]$, and if A and B are heights, these heights only differ by a small amount in many cases. However, even a height difference of 1 will lead to a relative error in the exponential height of at most $\frac{1}{2}$, and considerably less than that in many cases. This turns out to be enough to obtain 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 on $\mathrm{E}[H'(n)]$ directly implies a logarithmic upper bound on $\mathrm{E}[H(n)]$.

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

Definition 6 (The exponential height of a random BST).

$$\begin{aligned}
&\mathrm{eheight_rbst}\ 0 = \mathrm{return}\ 0\\
&\mathrm{eheight_rbst}\ 1 = \mathrm{return}\ 1\\
n > 1 \Longrightarrow\ &\mathrm{eheight_rbst}\ n =\\
&\quad \mathbf{do}\ \{\\
&\qquad k \leftarrow \mathrm{pmf_of_set}\ \{0 \ldots n-1\}\\
&\qquad h_1 \leftarrow \mathrm{eheight_rbst}\ k\\
&\qquad h_2 \leftarrow \mathrm{eheight_rbst}\ (n-k-1)\\
&\qquad \mathrm{return}\ (2 \cdot \max\ h_1\ h_2)\\
&\quad \}
\end{aligned}$$

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

Lemma 4 (Correctness of *eheight_rbst*).

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

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

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

However, we still have to find a suitable polynomial upper bound to complete the induction argument. If we had some polynomial $P(n)$ that fulfils $0 \leq P(0)$, $1 \leq P(1)$, 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 $\mathrm{E}[H'(n)] \leq P(n)$ by induction. Cormen *et al.* give the following polynomial, which satisfies all these conditions and makes everything work out nicely:

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

Putting all of these together gives us the following theorem:

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

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

5 Treaps

As we have seen, BSTs 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, the elements are instead 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* instead, which we shall introduce and analyse now.

5.1 Definition

A treap is a binary tree in which every node contains both an element and an associated priority and which is a BST 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). This kind of structure was first described by Vuillemin [23], who called it a *cartesian tree*, and independently studied further by Seidel and Aragon [24], who noticed its relationship to random BSTs. 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 these treap conditions is that, as long as all of the priorities are distinct, the shape of a treap is uniquely determined by the set of its elements and their priorities. Since the sub-trees of a treap must also be treaps, this uniqueness property follows by induction and we can construct this unique treap for a given set using the following simple algorithm:

Lemma 5 (Constructing the unique treap for a set). *Let A be a set of pairs of type $\alpha \times \mathbb{R}$ where the second components are all distinct. Then there exists a unique treap* treap_of A *whose elements are precisely A, and it satisfies the recurrence*

$$\begin{aligned}&\text{treap\_of } A \;=\\ &\quad \textbf{if } A = \{\} \textbf{ then } \langle\rangle \textbf{ else}\\ &\qquad \textbf{let } x = \text{arg\_min\_on snd } A\\ &\qquad \textbf{in } \langle \text{treap\_of } \{y \in A \mid \text{fst } y < \text{fst } x\}, x, \text{treap\_of } \{y \in A \mid \text{fst } y > \text{fst } x\}\rangle\end{aligned}$$

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 and consider it as a simple BST, the resulting tree is exactly the same as if we had first sorted the keys by increasing priority and then inserted them into an empty BST in that order. Formally, we have the following lemma:

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*

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

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

Proof. By induction over $xs' := \mathit{sort\_key}\ p\ xs$, using the fact that sorting w. r. t. distinct priorities can be seen as a selection sort: The list xs' consists of the unique minimum-priority element x, followed by $\mathit{sort\_key}\ p\ (\mathit{remove1}\ x\ xs)$, where *remove1* deletes the first occurrence of an element from 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 regardless of the order in which they were actually inserted. In particular, we can assign each element a *random* priority upon its insertion, which turns our treap (a deterministic data structure for values of type

($\alpha \times \mathbb{R}$) *set*) into a randomised treap, which is a *randomised* data structure for values of type α that has the same distribution as a random BST with the same content.

One caveat is that for all the results so far, we assumed that no two distinct elements have the same priority, 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. For this reason, treaps are usually described in the literature as using a continuous distribution (e. g. uniformly-distributed real numbers between 0 and 1), even though this cannot be implemented faithfully on an actual computer. We shall do the same here, since it makes the analysis much easier.[1]

The argument goes as follows:

1. Choosing the priority of each element randomly when we insert it is the same as choosing all the priorities beforehand (i. i. d. at random) 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 priorities i. i. d. for all elements and then looking at the linear ordering defined by these priorities is the same as choosing one of the $n!$ possible linear orderings uniformly at random.
4. Thus, inserting a list of elements into a randomised treap is the same as inserting them into a BST in random order.

5.2 The Measurable Space of Trees

One complication when 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 with a priority that is uniformly distributed between 0 and 1, we get a distribution of trees with 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: 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 $\{\mathit{Leaf}\} \in \mathcal{T}(M)$ and *Node* must be $\mathcal{T}(M) \otimes \mathcal{M} \otimes \mathcal{T}(M)$–$\mathcal{T}(M)$-measurable

[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.

– the projection functions, i. e. selecting the value/left sub-tree/right sub-tree 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 can construct such a measurable space by taking the σ-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. insertion is now a randomised operation.

Definition 7 (Randomised insertion into a treap).

$$\begin{aligned}&\text{rins} :: \alpha \to \alpha\ \text{treap} \to \alpha\ \text{treap measure}\\&\text{rins}\ x\ t = \textbf{do}\ \{p \leftarrow \text{uniform\_measure}\ \{0\ldots1\};\ \text{return}\ (\text{ins}\ x\ p\ t)\}\end{aligned}$$

Since we would like to analyse what happens when we insert a large 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 again assume that all of the elements that 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, the uniqueness property of treaps means that 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 like this:

Theorem 4 (Connecting randomised treaps to 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 $\mathcal{U}^A$ denote a vector of i. i. d. distributions $\mathcal{U}$, indexed by A:

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

The first step is to 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 priorities) 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 nonetheless rather tedious, mostly because of the issue of having to prove measurability in every single step. The second equality follows from the uniqueness of treaps.

Next, we note that the priority function returned by $\mathcal{U}^{\,set\,xs}$ 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 the uniform distribution on all the $|A|!$ possible 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 π 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: Proving that 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 inserting them in the order given by a randomly-permuted list):

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

Here we use the fact that priorities chosen uniformly at random induce a uniformly random linear ordering, and that sorting a list with such an ordering produces permutations of that list uniformly at random. The proof of this involves 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*), which concludes the proof. □

6 Randomised BSTs

Another approach to use randomisation in order to obtain a tree data structure that behaves essentially like a random BST irrespectively of the order of insertion are the *Randomised BSTs* introduced by Martinez and Roura [25]. We will call them *MR trees* from now on. The key differences between randomized treaps and MR trees are:

The randomness in treaps lies in the priorities associated with each entry. This priority is chosen *once* for each key and never modified. The algorithms that operate on the tree itself (insertion, deletion, etc.) are completely deterministic; their randomness comes only from the random priorities that were chosen in advance.

In MR trees, on the other hand, there is no extra information associated to the nodes. The randomness is introduced through coin flips in every recursive step of the tree operations (insertion, deletion, etc). Unlike with treaps, the results of these random choices are not stored in the tree explicitly (although they are, of course, implicitly present in the tree's structure). Another key difference is that, seeing as they are coin flips, the random choices that are made are *discrete*.

Note that while we said above that MR trees do not contain any additional *new* information, it is necessary to add cached information for an efficient implementation: the precise coin weights at each step depend on the total number of nodes in the tree; since this number takes linear time to compute and we are aiming for logarithmic time, it needs to be cached in the node itself. However, unlike with treaps, this, this caching can be ignored completely in the correctness analysis of the algorithm. It could easily be added in a refinement step later.

Let us now look at the implementation of the operations themselves. For a more didactic and informal introduction, we refer the reader to the original presentation by Martinez and Roura [25].

6.1 Auxiliary Operations

Splitting First, we need the following deterministic auxiliary function that splits a BST into two trees consisting of all the elements that are strictly smaller (resp. strictly greater) than some fixed element x.

$$\begin{aligned}&\text{split\_bst } \_\ \langle\rangle = (\langle\rangle, \langle\rangle)\\ &\text{split\_bst } x\ \langle l, y, r\rangle =\\ &\quad \textbf{if } y < x \textbf{ then}\end{aligned}$$

```
    case split_bst x r of (t1, t2) ⇒ (⟨l, y, t1⟩, t2)
else if y > x then
    case split_bst x l of (t1, t2) ⇒ (t1, ⟨t1, y, r⟩)
else
    (l, r)
```

This operation preserves the 'random BST' property in the following sense:

```
map_pmf (split_bst x) (random_bst A) =
    pair_pmf (random_bst {y ∈ A | y < x}) (random_bst {y ∈ A | y > x})
```

where

$$\text{pair\_pmf } P\ Q = \textbf{do } \{x \leftarrow P;\ y \leftarrow Q;\ \text{return } (x, y)\}$$

is the discrete product measure, analogous to $M \otimes N$ before.

For convenience, we additionally define a variant

$$\text{split\_bst' } x\ t = (x \in t,\ \text{split\_bst } x\ t)$$

that additionally tracks whether x itself was present in the original tree or not.

Joining Next, we define a kind of inverse operation for split_bst that computes the union of two BSTs t_1 and t_2 under the precondition that all values in t_1 are strictly smaller than those in t_2. The idea is essentially to build up the tree top-down, flipping a weighted coin in each step to determine whether to insert a branch from t_1 or from t_2 as the next element:

```
mrbst_join ⟨⟩ t2 = t2
mrbst_join t1 ⟨⟩ = t1
mrbst_join t1 t2 =
  do {
    b ← bernoulli_pmf (|t1| / (|t1| + |t2|))
    if b then
      case t1 of ⟨l, x, r⟩ ⇒ do {r' ← mrbst_join r t2; return ⟨l, x, r'⟩}
    else
      case t2 of ⟨l, x, r⟩ ⇒ do {l' ← mrbst_join t1 l; return ⟨l', x, r⟩}
  }
```

This operation fulfils the following correctness theorem:

Theorem 5. *If t_1 and t_2 are BSTs such that every element of t_1 is smaller than any element in t_2, then the result of mrbst_join t_1 t_2 is also a BST and its elements are the unions of those of t_1 and t_2. In Isabelle notation:*

```
theorem
  assumes t' ∈ set_pmf (mrbst_join t1 t2) and bst t1 and bst t2
  assumes ∀x∈set_tree t1. ∀y∈set_tree t2. x < y
```

shows bst t' **and** set_tree $t' =$ set_tree $t_1 \cup$ set_tree t_2

Moreover, if t_1 and t_2 are random BSTs, then mrbst_join t_1 t_2 is as well, i. e.:

theorem
assumes finite A **and** finite B **and** $\forall x{\in}A.\ \forall y{\in}B.\ x < y$
shows **do** $\{t_1 \leftarrow$ random_bst A; $t_2 \leftarrow$ random_bst B; mrbst_join t_1 $t_2\} =$
random_bst $(A \cup B)$

Pushdown mrbst_join is an inverse operation to split_bst x if x is not present in the tree. We will also need an inverse operation to split_bst x in the case that x *is* present in the tree. Following Roura & Martinez, this operation is called mrbst_push_down x l r. The situation where it is used can also be thought of like this: as we have noted before, a random BST over a non-empty set A can be seen as

$$\textbf{do}\ \{x \leftarrow A;\ l \leftarrow \text{random\_bst}\ \{y \in A \mid y < x\};\ r \leftarrow \text{random\_bst}\ \{y \in A \mid y > x\};\ \text{return}\ \langle l, x, r\rangle\}\ . \tag{1}$$

Now suppose we do *not* choose x at random, but rather do the above for a fixed x:

$$\textbf{do}\ \{l \leftarrow \text{random\_bst}\ \{y \in A \mid y < x\};\ r \leftarrow \text{random\_bst}\ \{y \in A \mid y > x\};\ \text{return}\ \langle l, x, r\rangle\} \tag{2}$$

The purpose of mrbst_push_down is then to transform the 'almost random BST' distribution described by (2) into the 'proper random BST' distribution described by (1). In a sense, we have a random BST where we have looked at the root and now want forget this knowledge again.

The definition of mrbst_push_down is similar to mrbst_join except that we now have a three-way split: In every step, we toss a weighted 'three-sided coin' to determine whether to insert a branch from l, a branch from r, or x itself (which stops the recursion).

```
mrbst_push_down l x r =
  do {
    k ← pmf_of_set {0 ... |l| + |r|}
    if k < |l| then
      case l of ⟨ll, y, lr⟩ ⇒ do {r' ← mrbst_push_down lr x r; return ⟨ll, y, r'⟩}
    else if k < |l| + |r|
      case r of ⟨rl, y, rr⟩ ⇒ do {l' ← mrbst_push_down l x rl; return ⟨l', y, rr⟩}
    else
      return ⟨l, x, r⟩
  }
```

Similarly to before, we get the following correctness theorem:

Theorem 6. *If l and r are BSTs such that every element of t_1 is less than x and every element of t_2 is greater than x, then the result of mrbst_push_down l x r*

is also a BST and its elements are the unions of those of t_1 and t_2, plus x. In Isabelle notation:

theorem
  assumes $t' \in$ set_pmf (mrbst_push_down l x r) **and** bst t_1 **and** bst t_2
  assumes $\forall y \in$ set_tree t_1. $y < x$ **and** $\forall y \in$ set_tree t_2. $y > x$
  shows bst t' **and** set_tree $t' =$ {x} $\cup$ set_tree $t_1 \cup$ set_tree t_2

Moreover, if t_1 and t_2 are random BSTs, then mrbst_push_down l x r is as well, i. e.:

theorem
  assumes finite A **and** finite B **and** $\forall y \in A$. $y < x$ **and** $\forall y \in A$. $y > x$
  shows **do** {$l \leftarrow$ random_bst A; $r \leftarrow$ random_bst B; mrbst_push_down l x r} =
    random_bst ({x} $\cup$ $A \cup B$)

6.2 Main Operations

Intersection and Difference With this, we can now define the intersection and difference of two MR trees very easily:

```
mrbst_inter_diff b ⟨⟩ t₂ = return ⟨⟩
mrbst_inter_diff b ⟨l₁, x, r₁⟩ t₂ =
  do {
    let (l₂, r₂) = split_bst x t₂
    l ← mrbst_inter_diff b l₁ l₂
    r ← mrbst_inter_diff b r₁ r₂
    if (x ∈ t₂) = b then return ⟨l, x, r⟩ else mrbst_join l r
  }
```

Choosing $b :=$ true yields the intersection of t_1 and t_2; choosing $b :=$ false yields their difference. Using this unified intersection and difference algorithm, we can prove the correctness of both in one go and avoid a duplication of proofs.

The correctness theorems for intersection and difference are the following:

Theorem 7. *Let b be some Boolean value and let $\diamond$ denote the set intersection operator $\cap$ if $b =$ true and the set difference operator $\setminus$ if $b =$ false. Then the following holds:*

theorem
  assumes $t' \in$ set_pmf (mrbst_inter_diff b t_1 t_2) **and** bst t_1 **and** bst t_2
  shows bst t' **and** set_tree $t' =$ set_tree $t_1 \diamond$ set_tree t_2

Moreover, if t_1 and t_2 are random BSTs, then mrbst_inter_diff b t_1 t_2 is as well, i. e.:[2]

do {$t_1 \leftarrow$ random_bst A; $t_2 \leftarrow$ random_bst B; mrbst_inter_diff b t_1 t_2} =
  random_bst ($A \diamond B$)

[2] We omit the finiteness assumptions on A and B from now on.

Union The union of two MR tree is somewhat trickier to define:

$$
\begin{array}{l}
\text{mrbst\_union } \langle\rangle\ t_2 = \text{return } t_2 \\
\text{mrbst\_union } t_1\ \langle\rangle = \text{return } t_1 \\
\text{mrbst\_union } t_1\ t_2 = \\
\quad \textbf{do } \{ \\
\quad\quad \textbf{let } \langle l_1, x, r_1\rangle = t_1 \ \textbf{ and } \ \langle l_2, x, r_2\rangle = t_2 \\
\quad\quad b \leftarrow \text{bernoulli\_pmf } \left(\frac{|t_1|}{|t_1|+|t_2|}\right) \\
\quad\quad \textbf{if } b \textbf{ then do } \{ \\
\quad\quad\quad \textbf{let } (l_2', r_2') = \text{split\_bst } x\ t_2 \\
\quad\quad\quad l \leftarrow \text{mrbst\_union } l_1\ l_2' \\
\quad\quad\quad r \leftarrow \text{mrbst\_union } r_1\ r_2' \\
\quad\quad\quad \text{return } \langle l, x, r\rangle \\
\quad\quad \} \textbf{ else do } \{ \\
\quad\quad\quad \textbf{let } (l_1', r_1') = \text{split\_bst } y\ t_1 \\
\quad\quad\quad l \leftarrow \text{mrbst\_union } l_1'\ l_2 \\
\quad\quad\quad l \leftarrow \text{mrbst\_union } r_1'\ r_2 \\
\quad\quad\quad \textbf{if } y \in t_1 \textbf{ then} \\
\quad\quad\quad\quad \text{mrbst\_push\_down } l\ y\ r \\
\quad\quad\quad \textbf{else} \\
\quad\quad\quad\quad \text{return } \langle l, y, r\rangle \\
\quad\quad \} \\
\quad \}
\end{array}
$$

The correctness theorem looks like this:

Theorem 8.

theorem
assumes $t' \in \text{set\_pmf } (\text{mrbst\_union } t_1\ t_2)$ **and** $\text{bst } t_1$ **and** $\text{bst } t_2$
shows $\text{bst } t'$ **and** $\text{set\_tree } t' = \text{set\_tree } t_1 \cup \text{set\_tree } t_2$

Moreover, if t_1 and t_2 are random BSTs, then mrbst_union t_1 t_2 is as well, i. e.:

$$
\begin{array}{l}
\textbf{do } \{t_1 \leftarrow \text{random\_bst } A;\ t_2 \leftarrow \text{random\_bst } B;\ \text{mrbst\_union } t_1\ t_2\} = \\
\quad \text{random\_bst } (A \cup B)
\end{array}
$$

Derived Operations We omit the definitions of the insertion and deletion algorithms here since they are just specialised and 'inlined' versions of the union and difference algorithms with the first (resp. second) tree taken to be a singleton tree $\langle\langle\rangle, x, \langle\rangle\rangle$. This is also how the correctness of insertion and deletion was shown in Isabelle: The algorithms were defined recursively just like given by Roura & Martinez. Then, we show

$$
\begin{array}{l}
\text{mrbst\_insert x t} = \text{mrbst\_union } \langle\langle\rangle, x, \langle\rangle\rangle\ t \\
\text{mrbst\_delete x t} = \text{mrbst\_diff } t\ \langle\langle\rangle, x, \langle\rangle\rangle
\end{array}
$$

by a straightforward (and fully automatic) induction, so that the correctness results for *mrbst_union* and *mrbst_diff* carry over directly.

6.3 Proofs

The correctness proofs are all straightforward. The key ingredients are (1) and the fact that choosing uniformly at random from a disjoint union $A \cup B$ can be replaced by tossing a coin weighted with $\frac{|A|}{|A|+|B|}$ and then choosing uniformly at random either from A (for heads) or from B (for tails). The rest is mostly a matter of rearranging in the Giry monad using the basic monad laws in addition to commutativity

$$\mathbf{do}\ \{x \leftarrow A;\ y \leftarrow B;\ C\ x\ y\} = \mathbf{do}\ \{y \leftarrow B;\ x \leftarrow A;\ C\ x\ y\}$$

and the absorption property

$$\mathbf{do}\ \{x \leftarrow A;\ B\} = B \qquad \text{(where } B \text{ does not depend on } A\text{)}\ .$$

7 Related Work

The earliest analysis of randomised algorithms in a theorem prover was probably by Hurd [26] in the HOL system, who modelled them by assuming the existence of an infinite sequence of random bits which programs can consume. He used this approach to formalise the Miller–Rabin primality test.

Audebaud and Paulin-Mohring [27] created a shallowly-embedded formalisation of (discrete) randomised algorithms in Coq and demonstrate its usage on two examples. Barthe *et al.* [28] used this framework to implement the *CertiCrypt* system to write machine-checked cryptographic proofs for a deeply embedded imperative language. Petcher and Morrisett [29] developed a similar framework but based on a monadic embedding. Another similar framework was developed for Isabelle/HOL by Lochbihler [30].

The expected running time of randomised quicksort (possibly including repeated elements) was first analysed in a theorem prover by van der Weegen and McKinna [31] using Coq. They proved the upper bound $2n\lceil \log_2 n \rceil$, whereas we actually proved the closed-form result $2(n+1)H_n - 4n$ and its precise asymptotics. Although their paper's title mentions 'average-case complexity', they, in fact, 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 us, they allow lists to have repeated elements even in the average case, but they proved the expectation bounds separately and independently, while we assumed that there are 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.* [32] presented a Hoare-style calculus for analysing the expected running time of imperative programs and used it to analyse a one-dimensional random walk and the Coupon Collector's problem. Hölzl [33] formalised this approach in Isabelle and found a mistake in their proof of the random walk in the process.

At the same time as our work and independently, Tassarotti and Harper [34] gave a Coq formalisation of a cookbook-like theorem based on work by Karp [35] that is able to provide tail bounds for a certain class of randomised recurrences such as the number of comparisons in quicksort and the height of a random BST. In contrast to the expectation results we proved, such bounds are very difficult to obtain on a case-by-case basis, which makes such a cookbook-like result particularly useful.

Outside the world of theorem provers, other approaches exist for automating the analysis of such algorithms: Probabilistic model checkers like PRISM [36] can check safety properties and compute expectation bounds. The $\Lambda\gamma^{\Omega}$ system by Flajolet *et al.* [37] conducts fully automatic analysis of average-case running time for a restricted variety of (deterministic) programs. Chatterjee *et al.* [38] developed a method for deriving bounds of the shape $O(\ln n)$, $O(n)$, or $O(n \ln n)$ for certain recurrences that are relevant to average-case analysis automatically and applied it to a number of interesting examples, including quicksort.

8 Future Work

We have closed a number of important gaps in the formalisation of classic probabilistic algorithms related to binary search trees, including the thorny case of treaps, which requires measure theory. Up to that point we claim that these formalisations are readable (the definitions thanks to the Giry monad and the proofs thanks to Isar [39]), but for treaps this becomes debatable: the issue of measurability makes proofs and definitions significantly more cumbersome and less readable. Although existing automation for measurability is already very helpful, there is still room for improvement. Also, the construction of the measurable space of trees generalises to other data types and could be automated.

All of our work so far has been at the functional level, but it would be desirable to refine it to the 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, Johannes Hölzl for his help with the construction of the tree space, and Bohua Zhan and Maximilian P. L. Haslbeck for comments on a draft. We also thank the reviewers for their suggestions.

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. Aslam, J.A.: A simple bound on the expected height of a randomly built binary search tree. Technical Report TR2001-387, Dartmouth College, Hanover, NH (2001) Abstract and paper lost.
23. Vuillemin, J.: A unifying look at data structures. Commun. ACM **23**(4) (April 1980) 229–239

24. Seidel, R., Aragon, C.R.: Randomized search trees. Algorithmica **16**(4) (Oct 1996) 464–497
25. Martínez, C., Roura, S.: Randomized binary search trees. Journal of the ACM **45** (1997)
26. Hurd, J.: Formal Verification of Probabilistic Algorithms. PhD thesis, University of Cambridge (2002)
27. Audebaud, P., Paulin-Mohring, C.: Proofs of randomized algorithms in Coq. Sci. Comput. Program. **74**(8) (2009) 568–589
28. 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
29. 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
30. 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
31. 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
32. 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
33. 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
34. Tassarotti, J., Harper, R.: Verified tail bounds for randomized programs. In Avigad, J., Mahboubi, A., eds.: Interactive Theorem Proving, Cham, Springer International Publishing (2018)
35. Karp, R.M.: Probabilistic recurrence relations. J. ACM **41**(6) (November 1994) 1136–1150
36. 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
37. 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
38. 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
39. 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`.