Randomized Binary Search Trees*

Conrado Martínez and Salvador Roura†

January 31, 1997

Abstract

In this paper we present randomized algorithms over binary search trees such that: a) the insertion of a set of keys, in any fixed order, into an initially empty tree always produces a random binary search tree; b) the deletion of any key from a random binary search tree results in a random binary search tree; c) the random choices made by the algorithms are based upon the sizes of the subtrees of the tree; this implies that we can support accesses by rank without additional storage requirements or modification of the data structures; and d) the cost of any elementary operation, measured as the number of visited nodes, is the same as the expected cost of its standard deterministic counterpart; hence, all search and update operations have guaranteed expected cost $\mathcal{O}(\log n)$, but now irrespective of any assumption on the input distribution.

1. Introduction

Given a binary search tree (BST, for short), common operations are the search of an item given its key and the retrieval of the information associated to that key if it is present, the insertion of a new item in the tree and the deletion of some item given its key. The standard implementation of searches and updates in unbalanced BSTs is (except for deletions) simple and elegant, and the cost of any of these operations is always linearly bounded by the height of the tree.

For *random binary search trees*, the expected performance of a search, whether successful or not, and that of update operations is $\mathcal{O}(\log n)$ [15, 17, 24], with small hidden constant factors involved (here and unless otherwise stated, n denotes the number of items in the tree or *size* of the tree). Random BSTs are those built using only random insertions. An insertion in a BST of size $j-1$ is random if there is the same probability for the inserted key to fall into any of the j intervals defined by the keys already in the tree.

However, if the input is not random (for instance, long ordered subsequences of keys are likely to be inserted) then the performance of the operations can dramatically degrade and become linear. Moreover, little is known about the behavior of BSTs in the presence of both insertions and deletions. None of the known deletion algorithms, including Hibbard's deletion algorithm [11] and its multiple variants, preserve randomness —a surprising fact that was first noticed by Knott [13]. There have been several interesting empirical and theoretical studies around this question [12, 14, 6, 5, 4] but their results are partial or inconclusive. There is some evidence that some deletion algorithms degrade the overall performance to $\Theta(\sqrt{n})$; for others, empirical evidence suggests that this degradation does not take place, but the experiments were only conducted for long runs of deletion/insertion pairs applied to an initially random BST.

*This research was supported by the ESPRIT BRA Project ALCOM II, contract#7141, by the ESPRIT LTR Project ALCOM-IT, contract # 20244 and by a grant from CIRIT (Comissió Interdepartamental de Recerca i Innovació Tecnològica).

†Departament de Llenguatges i Sistemes Informàtics, Universitat Politècnica de Catalunya, E-08028 Barcelona, Catalonia, Spain. E-mail: {conrado,roura}@goliat.upc.es

The traditional approach to elude these problems is to impose additional constraints on the heights, sizes, etc. of the subtrees; many kinds of *balanced search trees* have been proposed, like AVLs [1], red-black trees [10], weight-balanced trees (also known as BB[α] trees) [19], height-ratio-balanced trees [8], ... All balanced search trees guarantee logarithmic performance of the search and update operations in the worst-case. The insertion and deletion algorithms must guarantee that the resulting BST does not violate any of the constraints; typically, this is achieved using *rotations*. The disadvantage of balanced search trees is that the update algorithms are rather complex, and the constant factors in the performance can become large. Furthermore, each internal node must also contain *balance information*, needed only to verify whether the constrainsts are satisfied or not at that particular location.

Another approach to cope with the problem of worst-case sequences is the use of randomization techniques. In many situations, randomized algorithms are simple, elegant and their expected performance is the same as the worst-case performance of much more complicated deterministic algorithms; we should not forget that randomized algorithms provide a guarantee for their expected performance, which is no longer dependent on any assumption about the input distribution [21]. Therefore, unless the random choices made by these algorithms were known, a sequence of operations that forced some designated behavior (say, the worst-case) could not be constructed.

Randomization techniques were used by Aragon and Seidel [3] for their *randomized treaps* and by Pugh [20] in his definition of *skip lists*. Thanks to the randomization process, skip lists and randomized treaps achieve logarithmic expected performance.

In this paper, we consider randomized algorithms to dynamically mantain a dictionary in a BST. We call the BSTs produced by our algorithms randomized binary search trees (RBSTs). In Sections 2 and 3 we show that RBSTs are, in fact, random binary search trees, irrespective of the order in which the keys were inserted or deleted, the actual pattern of insertions and deletions, etc. Hence, RBSTs have guaranteed logarithmic expected performance, provided that the random decisions made by the algorithms remain unknown to an hypothetical adversary trying to force worst-case performance.

It is important to point out here that our deletion algorithm is the only one known that really preserves randomness in a strict sense [14]; i.e., a subsequent insertion will not destroy randomness, like it happens if Hibbard's deletion algorithm or some of its variants is used [13]. In Theorem 3.1 we formally prove that our deletion algorithm always produces a random BST, if the input tree was a random BST, and irrespective of the key that is deleted. This provides a satisfactory answer to the long standing open question about the existence of such a randomness-preserving deletion algorithm [14].

Our algorithms yield similar results to those for randomized treaps [3]. Rather than discussing similarities and differences between RBSTs and randomized treaps at different places of this paper, we will make a single general comment about these topics at this point. Considering their external behavior, randomized treaps and RBSTs yield exactly the same results under insertions and deletions, because those results are, in both cases, always random binary search trees. In particular, the main results of Sections 2 and 3 apply for both RBSTs and randomized treaps because the insertion and deletion algorithms are externally equivalent: the random choices for RBSTs and randomized treaps are made using quite different mechanisms, but the probability that a given random choice is taken is the same in both RBSTs and randomized treaps. One of the differences between our insertion and deletion algorithms and those of Aragon and Seidel is that ours generate random integers in the range $0..n$, where n is the current size of the RBST, while randomized treaps use theoretically unbounded random numbers in the unit interval (the so-called random priorities). In practice, though, only finite precision numbers need to be generated, since it is rather unlikely that two random priorities have a very large common prefix in their binary representations.

This difference between RBSTs and randomized treaps is significant, since the random choices made by our algorithms are based upon structural information, namely, the size of the subtree

rooted at each node of the RBST. Hence, RBSTs support searches and deletions by rank and the computation of the rank of a given item without additional storage requirements or modification of the data structure. In contrast, the random choices of treaps are based upon non-structural information: each node stores a random priority, a real number in $[0, 1]$, which is only useful for the randomization process. A similar discussion applies if we compare our algorithms with random skip lists.

Regarding the analysis of the performance of our algorithms, we will exploit the large amount of known results about random BSTs, after we will have explicitly noticed and proved that a RBST is always a random BST. In contrast, Aragon and Seidel (see also [21, 16]) made use of Mulmuley games to analyze the performance of randomized treaps.

The paper is organized as follows. In Sections 2 and 3, the insertion and deletion algorithms are described and their main properties stated. We present the analysis of the performance of the basic operations in Section 4. Section 5 describes other operations: a variant for the insertion of repeated keys, set operations over RBSTs and a family of related self-adjusting strategies. In Section 6 we discuss several implementation-related issues: efficient strategies for the dynamic management of subtree sizes, space requirements, etc. In Section 7 we introduce a formal framework for the description of randomized algorithms and the study of their properties, and show how to derive all the results in the preceding sections in a unified, rigorous and elegant way. We conclude in Section 8 with some remarks and future research lines.

An early version of this work appeared in [22].

2. INSERTIONS

We assume that the reader is familiar with the definition of binary search tree and its standard insertion algorithm [15, 23, 9]. To make the description and analysis simpler, we will assume w.l.o.g. that each item in the tree consists of a key with no associated information, and that all keys are nonnegative integers. The empty tree or *external node* is denoted by $\square$.

Besides the definition of random BSTs in terms of random insertions given in the introduction, there are several equivalent characterizations of random BSTs that we will find useful in our investigation [15, 17]. In particular, we will use the following nice recursive definition for random BSTs.

Definition 2.1 *Let T be a binary search tree of size n.*

- *If $n = 0$ then $T = \square$ and it is a random binary search tree;*
- *If $n > 0$, the tree T is a random binary search tree if and only if both its left subtree L and its right subtree R are independent random binary search trees, and*

$$\Pr\{\mathsf{size}(L) = i \,|\, \mathsf{size}(T) = n\} = \frac{1}{n}, \qquad i = 0, \ldots, n-1, \quad n > 0. \tag{1}$$

An immediate consequence of Equation (1) in the definition above is that any of the keys of a random BST of size $n > 0$ has the same probability, namely $\frac{1}{n}$, of being the root of the tree. This property of random BSTs is crucial in our study, as it provides the basic idea for the randomized insertion algorithm that we describe next (see Algorithm 1). Informally, in order to produce random BSTs, a newly inserted key should have some chance of becoming the root, or the root of one of the subtrees of the root, and so forth. We assume for simplicity that x, the key to be inserted, is not yet in the tree T. The algorithm is written in C-like notation and we assume that a tree is actually represented by a pointer to its root. The field $T \to \mathsf{key}$ is the key at the root of T. The fields $T \to \mathsf{size}$, $T \to \mathsf{left}$ and $T \to \mathsf{right}$ store the size, the left subtree and the right subtree of the tree T, respectively. We assume that the expression $T \to \mathsf{size}$ is correct even if $T = \square$ (that is, T is a null pointer), and evaluates to 0 in that case.

We begin generating a random integer r in the range $0..n$, where $n = T \to \mathsf{size}$. If $n = 0$ then the test $r = n$? will succeed and the call to insert_at_root will return a tree with a single node containing x at its root and two empty subtrees.

If the tree T is not empty, with probability $\frac{1}{n+1}$ we place x as the root of the new RBST using insert_at_root (notice that the new RBST will have size $n+1$), and with probability $1 - \frac{1}{n+1} = \frac{n}{n+1}$ we recursively insert x in the left or right subtree of T, depending on the relation of x with the key at the root of T. To keep the algorithm as simple as possible, we have refrained to include the code that updates the size field when a new item is inserted. We address this question later, in Section 6.

Algorithm 1 Insertion

```
bst insert(int x, bst T) {
  int n, r;

  n = T→size;
  r = random(0,n);
  if (r == n)
      return insert_at_root(x,T);
  if (x < T→key)
      T→left = insert(x, T→left);
  else
      T→right = insert(x, T→right);
  return T;
}
```

We have not said yet how to insert x at the root of T, that is, how to build a new tree T' containing x and the keys that were present in T, such that $T' \to \mathsf{key} = x$. This is the job performed by insert_at_root(x, T), which implements the algorithm developed by Stephenson [26]. The process is analogous to the partition of an array around a pivot element in the quicksort algorithm. Here, the tree T is split into two trees $T_<$ and $T_>$, which contain the keys of T that are smaller than x and larger than x, respectively. Then $T_<$ and $T_>$ are attached as the left and right subtrees of a new node holding x (see Algorithm 2). We present a recursive implementation of split(x, T), but it is also straightforward to write a non-recursive top-down implementation.

If T is empty, nothing must be done and both $T_<$ and $T_>$ are also empty. Otherwise, if $x < T \to \mathsf{key}$ then the right subtree of T and the root of T belong to $T_>$. To compute $T_<$ and the remaining part of $T_>$, that is, the subtree that contains the keys in $T \to \mathsf{left}$ which are greater than x, we make a recursive call to split$(x, T \to \mathsf{left})$. If $x > T \to \mathsf{key}$, we proceed in a similar manner.

It is not difficult to see that split(x, T) compares x against the same keys in T as if we were making an unsucessful search for x in T. Therefore, the cost of an insertion at the root is proportional to the cost of the insertion of the same item in the same tree using the standard insertion algorithm.

The randomized insertion algorithm always preserves randomness, no matter which is the item x that we insert (we should better say that the algorithm forces randomness). We give a precise meaning to this claim and prove it in Theorem 2.1, but first we need to establish an intermediate result that states that the splitting process carried out during an insertion at root also preserves randomness. This result is given in the following lemma.

Lemma 2.1 *Let $T_<$ and $T_>$ be the BSTs produced by* split(x, T). *If T is a random BST containing the set of keys K, then $T_<$ and $T_>$ are independent random BSTs containing the sets of keys $K_{<x} = \{y \in T \mid y < x\}$ and $K_{>x} = \{y \in T \mid y > x\}$, respectively.*

Algorithm 2 Insertion at the root

```
bst insert_at_root(int x, bst T) {
    bst S, G;

    split(x, T, &S, &G); /* S ≡ T_<, G ≡ T_> */
    T = new_node();
    T→key = x; T→left = S; T→right = G;
    return T;
}

void split(int x, bst T, bst *S, bst *G) {

    if (T == □) {
        *S = *G = □;
        return;
    }
    if (x < T→key) {
        *G = T;
        split(x, T→left, S, &(*G→left));
    }
    else { /* x > T→key */
        *S = T;
        split(x, T→right, &(*S→right), G);
    }
    return;
}
```

Proof. We prove the lemma by induction on n, the size of T. If $n = 0$, then $T = \square$ and $\mathsf{split}(x, T)$ yields $T_< = T_> = \square$, so the lemma trivially holds.
Now assume $n > 0$ and let $y = T \to \mathsf{key}$, $L = T \to \mathsf{left}$ and $R = T \to \mathsf{right}$. If $x > y$ then the root of $T_<$ is y and its left subtree is L. The tree $T_>$ and the right subtree of $T_<$ (let us call it R') are computed when the splitting process is recursively applied to R. By the inductive hypothesis this splitting process will produce two trees that are independent random BSTs; one of them, R', contains the set of keys $\{z \in t \,|\, y < z < x\}$ and the other is $T_>$. The subtree L of T is not modified in any way and is a random BST, since T is, by hypothesis, a random BST. Furthermore, L is independent of R' and $T_>$, since L and R are independent, too. It follows then that the trees $T_<$ and $T_>$ are also independent because R' and $T_>$ are independent. Finally, in order to show that the lemma is true when $n > 0$ and $x > y = T \to \mathsf{key}$, we have to prove that, for any $z \in T_<$, the probability that z is the root of $T_<$ is $\frac{1}{m}$, where m is the size of $T_<$. Indeed, it is the case, since

$$\mathbb{P}\left[z \text{ is root of } T_< \,|\, \text{root of } T \text{ is} < x\right] = \frac{\mathbb{P}\left[z \text{ is root of } T \text{ and root of } T \text{ is} < x\right]}{\mathbb{P}\left[\text{root of } T \text{ is} < x\right]} = \frac{1/n}{m/n} = \frac{1}{m}.$$

The same reasoning applies for the case $x < y$, interchanging the roles of $T_<$ and $T_>$, left and right, and so on. □

We are now ready to state the main result of this section, which is also one of the main results in this work.

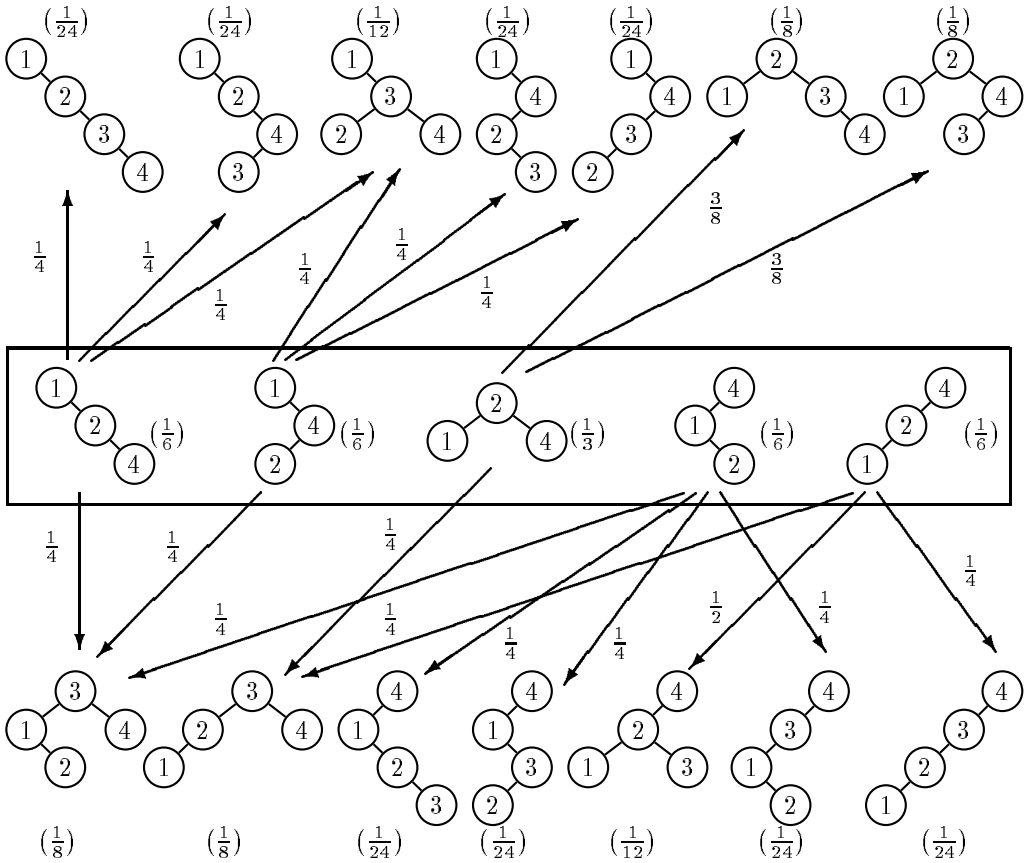


Figure 1: Insertion of $x = 3$ in a random BST for the set $K = \{1, 2, 4\}$.

Theorem 2.1 *If T is a random BST that contains the set of keys K and x is any key not in K, then* insert(x, T) *produces a random BST containing the set of keys $K \cup \{x\}$.*

Proof. Again, we use induction on n, the size of T. If $n = 0$ then $T = \square$ (indeed it is a random BST), and insert(x, T) returns the random BST with x at its root and two empty subtrees.
Assume now that $n > 0$ and that the theorem is true for all sizes $< n$. Consider an item $y \in K$. The probability that y is the root of T, before the insertion of x, is $\frac{1}{n}$, since T is a random BST. The only way for y to stay as the root of T after the insertion of x is that the random choice made during the first stage of the insertion is not to insert x at the root. This happens with probability $\frac{n}{n+1}$; hence, the probability that some item $y \in K$ is the root of $T' =$ insert(x, T) is $\frac{1}{n} \times \frac{n}{n+1} = \frac{1}{n+1}$. Moreover, if x is not inserted at the root during the first stage, it will be inserted at either the left or the right subtree of T; by the inductive hypothesis the result will be a random BST. To finish the proof we shall now consider the case where x is inserted at the root of T. First, this is the only way to have x at the root of T'; then we may conclude that x is the root of T' with probability $\frac{1}{n+1}$, as expected. On the other hand, from Lemma 2.1 we know that both the left and the right subtrees of T', constructed by the splitting process, are independent random BSTs; therefore, T' is a random BST. $\square$

Figure 1 shows the effects of the insertion of $x = 3$ in a random BST when $K = \{1, 2, 4\}$. The probability that each BST has according to the random BST model appears enclosed in parentheses. The arrows indicate the possible outcomes of the insertion of $x = 3$ in each tree and are labelled by

the corresponding probabilities. This figure also gives us an example of Lemma 2.1. Consider only the arrows that end in trees whose root is 3. These arrows show the result of inserting at root the key 3 in a random BST for the set of keys $\{1, 2, 4\}$. Notice that the root of the left subtree is either 1 or 2 with the same probability. Hence, the tree containing the keys smaller than 3 in the original random BST is also a random BST for the set of keys $\{1, 2\}$.

As an immediate consequence of the Theorem 2.1, the next corollary follows.

Corollary 2.1 *Let $K = \{x_1, \ldots, x_n\}$ be any set of keys, where $n \geq 0$. Let $p = x_{i_1}, \ldots, x_{i_n}$ be any fixed permutation of the keys in K. Then the RBST that we obtain after the insertion of the keys of p into an initially empty tree is a random binary search tree. More formally, if*

$$T = \mathsf{insert}(x_{i_n}, \mathsf{insert}(x_{i_{n-1}}, \ldots, \mathsf{insert}(x_{i_1}, \square) \ldots))$$

then T is a random BST for the set of keys K.

This corollary can be put into sharp contrast with the well known fact that the standard insertion of a *random* permutation of a set of keys into an initially empty tree yields a random BST; the corollary states that for any *fixed* permutation we will get a random BST if we use the RBST insertion algorithm.

3. Deletions

The deletion algorithm uses a procedure called join, which actually performs the removal of the desired key (see Algorithm 3). To delete a key x from the given RBST we first search for x, using the standard search algorithm until an external node or x is found. In the first case, x is not in the tree, so nothing must be done. In the second case, only the subtree whose root is x will be modified. Notice that most (if not all) deletion algorithms work so.

Algorithm 3 Deletion

```
bst delete(int x, bst T) {
  bst Aux;

  if (T == □)
      return □;
  if (x < T→key)
      T→left = delete(x, T→left);
  else if (x > T→key)
      T→right = delete(x, T→right);
  else { /* x == T→key */
      Aux = join(T→left, T→right);
      free_node(T);
      T = Aux;
  }
  return T;
}
```

Let T be the subtree whose root is x. Let L and R denote the left and right subtrees of T, respectively, and $K_{<x}$ and $K_{>x}$ denote the corresponding sets of keys. To delete the node where x is located (the root of T) we build a new BST $T' = \mathsf{join}(L, R)$ containing the keys in the set $K_{<x} \cup K_{>x}$

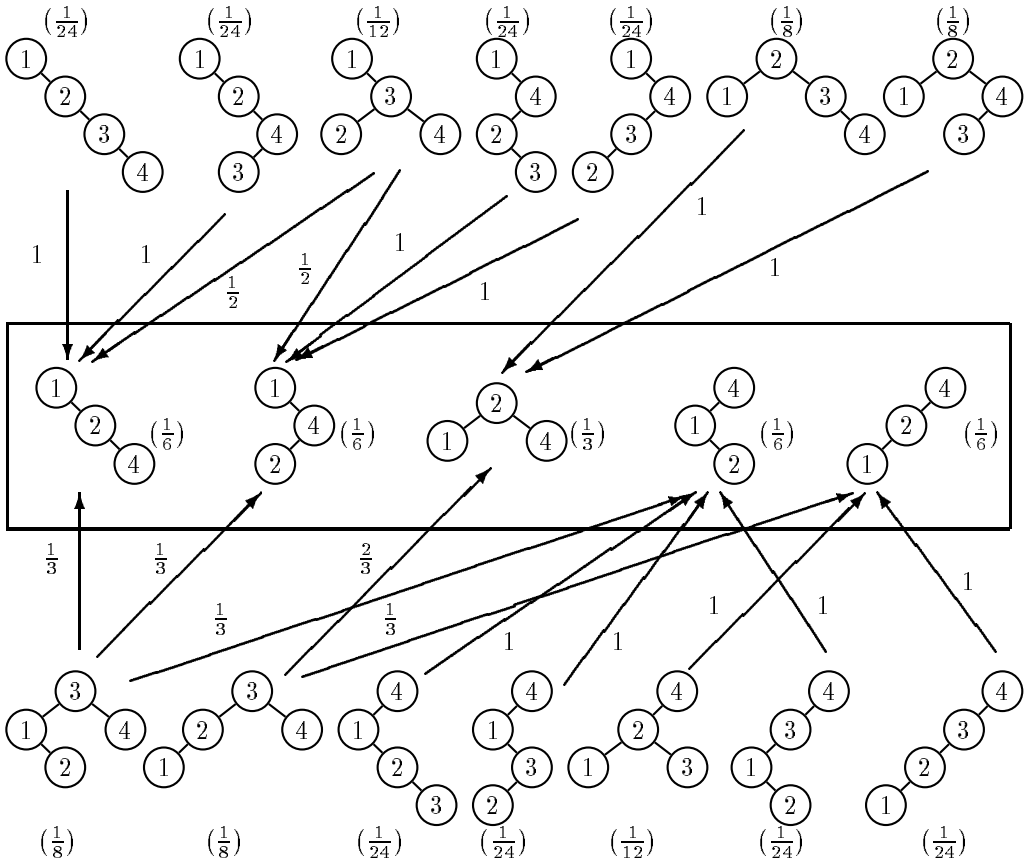


Figure 2: Deletion of $x = 3$ from a random BST for the set $K = \{1, 2, 3, 4\}$.

and replace T by T'. By hypothesis, the join operation does only work when none of the keys in its first argument is larger than any of the keys in its second argument.

Our definition of the result of joining two trees when one of them is empty is trivial: $\mathsf{join}(\square, \square) = \square$, $\mathsf{join}(L, \square) = L$, and $\mathsf{join}(\square, R) = R$. Notice, however, that Hibbard's deletion algorithm does not follow all these equations.

Now, assume that L and R are trees of size $m > 0$ and $n > 0$, respectively. A common way to perform the join of two non-empty trees, L and R, is to look for the maximum key in L, say z, delete it from L and let z be the common root of L' and R, where L' denotes the tree that results after the deletion of z from L. Alternatively, we might take the minimum item in R and put it as the root of $\mathsf{join}(L, R)$, using an analogous procedure. Deleting the maximum (minimum) item in a BST is easy, since it cannot have a non-empty right (left) subtree.

Our definition of join, however, selects either the root of L or the root of R to become the root of the resulting tree and then proceeds recursively. Let L_ℓ and L_r denote the left and right subtrees of L and similarly, let R_ℓ and R_r denote the subtrees of R. Further, let a and b denote the roots of the trees L and R. As we already said, join chooses between a and b to become the root of $T' = \mathsf{join}(L, R)$. If a is selected, then its left subtree is L_ℓ and its right subtree is the result of joining L_r with R. If b were selected then we would keep R_r as the right subtree of T' and obtain the left subtree by joining L with R_ℓ. The probability that we choose either a or b to be the root of T' is $\frac{m}{m+n}$ for a and $\frac{n}{m+n}$ for b.

Just as the insertion algorithm of the previous section preserves randomness, the same happens

with the deletion algorithm described above (Theorem 3.1). The preservation of randomness of our deletion algorithm stems from the corresponding property of join: the join of two random BSTs yields a random BST. As we shall see soon, this follows from the choice of the probabilities for the selection of roots during the joining process.

Lemma 3.1 *Let L and R be two independent random BSTs, such that the keys in L are strictly smaller than the keys in R. Let K_L and K_R denote the sets of keys in L and R, respectively. Then $T = \mathsf{join}(L, R)$ is a random BST that contains the set of keys $K = K_L \cup K_R$.*

Proof. The lemma is proved by induction on the sizes m and n of L and R. If $m = 0$ or $n = 0$ the lemma trivially holds, since $\mathsf{join}(L, R)$ returns the non-empty tree in the pair, if there is any, and $\square$ if both are empty. Consider now the case where both $m > 0$ and $n > 0$. Let $a = L \rightarrow \mathsf{key}$ and $b = R \rightarrow \mathsf{key}$. If we select a to become the root of T, then we will recursively join $L_r = L \rightarrow \mathsf{right}$ and R. By the inductive hypothesis, the result is a random BST. Therefore, we have that: 1) the left subtree of T is $L \rightarrow \mathsf{left}$ and hence it is a random BST (because so was L); 2) the right subtree of T is $\mathsf{join}(L_r, R)$, which is also random; 3) both subtrees are independent; and 4) the probability that any key x in L becomes the root of T is $\frac{1}{m+n}$, since this probability is just the probability that x was the root of L times the probability that it is selected as the root of T: $\frac{1}{m} \times \frac{m}{m+n} = \frac{1}{m+n}$.
The same reasoning shows that the lemma is also true if b were selected to become the root of T. $\square$

Algorithm 4 Join of two RBSTs

```
bst join(bst L, bst R) {
  int m, n, r, total;

  m = L→size; n = R→size; total = m + n;
  if (total == 0) return □;
  r = random(0, total - 1);
  if (r < m) {              /* with probability m/(m+n) */
      L→right = join(L→right, R);
      return L;
  }
  else {                    /* with probability n/(m+n) */
      R→left = join(L, R→left);
      return R;
  }
}
```

Theorem 3.1 *If T is a random BST that contains the set of keys K, then $\mathsf{delete}(x, T)$ produces a random BST containing the set of keys $K \setminus \{x\}$.*

Proof. If x is not in T, then delete does not modify T and the theorem trivially holds.
Let us supose now that x is in T. The theorem is proved by induction on the size n of T. If $n = 1$, then $\mathsf{delete}(x, T)$ produces the empty tree, and the theorem holds. Let us assume that $n > 1$ and the theorem is true for all sizes $< n$. If x was not the root of T, then we delete x from the left or right subtree of T and, by induction, this subtree will be a random BST. If x was the root of T, $T' = \mathsf{delete}(x, T)$ is the result of joining the left and right subtrees of T, which by last lemma will be a random BST. Therefore, both the left and right subtrees of T' are independent random BSTs. It

is left to prove that every $y \in K$ not equal to x has the same probability, $\frac{1}{n-1}$, of being the root of T'. This can be easily proved.

$$\begin{aligned}\mathbb{P}[y \text{ is the root of } T'] &= \mathbb{P}[y \text{ is the root of } T' \mid x \text{ was the root of } T] \times \mathbb{P}[x \text{ was the root of } T] \\ &+ \mathbb{P}[y \text{ is the root of } T' \mid x \text{ was not the root of } T] \times \mathbb{P}[x \text{ was not the root of } T] \\ &= \mathbb{P}[\mathsf{join} \text{ brings } y \text{ to the root of } T'] \times 1/n \\ &+ \mathbb{P}[y \text{ was the root of } T \mid x \text{ was not the root of } T] \times (n-1)/n \\ &= \frac{1}{n-1} \times \frac{1}{n} + \frac{1}{n-1} \times \frac{n-1}{n} = \frac{1}{n-1}.\end{aligned}$$

□

Figure 2 shows the effects of the deletion of $x = 3$ from a random BST when $K = \{1, 2, 3, 4\}$. The labels, arrows, etc. follow the same conventions as in Figure 1. Again, we can use this figure to give an example of the result of another operation, in this case join. Notice that the arrows that start in trees with root 3 show the result of joining two random BSTs, one with the set of keys $K_L = \{1, 2\}$ and the other with the set of keys $K_R = \{4\}$. The outcome is certainly a random BST with the set of keys $\{1, 2, 4\}$.

On the other hand, comparing Figures 1 and 2 produces this nice observation: For any fixed BST T, let $\mathbb{P}[T]$ be the probability of T according to the random BST model. Let T_1 and T_2 be any given BSTs with n and $n+1$ keys, respectively, and let x be any key not in T_1. Then

$$\mathbb{P}[T_1] \times \mathbb{P}[\text{Inserting } x \text{ in } T_1 \text{ produces } T_2] = \mathbb{P}[T_2] \times \mathbb{P}[\text{Deleting } x \text{ from } T_2 \text{ produces } T_1].$$

Combining Theorem 2.1 and Theorem 3.1 we get this important corollary.

Corollary 3.1 *The result of any arbitary sequence of insertions and deletions, starting from an initially empty tree is always a random BST. Furthermore, if the insertions are random (not arbitrary), then the result is still a random BST even if the standard insertion algorithm or Stephenson's insertion at root algorithm is used instead of the randomized insertion algorithm for RBSTs.*

4. PERFORMANCE ANALYSIS

The analysis of the performance of the basic algorithms is immediate, since both insertions and deletions guarantee the randomness of their results. Therefore, the large collection of results about random BSTs found in the literature may be used here. We will use three well known results (see for instance [15, 17, 24, 27]) about random BSTs of size n: the expected depth of the i-th internal node, the expected depth of the i-th external node (leaf), and the total expected lenght of the right and left *spines* of the subtree whose root is the i-th node. We will denote the corresponding random variables $\mathcal{D}_n^{(i)}$, $\mathcal{L}_n^{(i)}$ and $\mathcal{S}_n^{(i)}$. Recall that the right spine of a tree is the path from the root of the right son to the smallest element in that subtree. Analogously, the left spine is the path from the root of the left son to its largest element (see Figure 3). The expected values mentioned above are:

$$\begin{aligned}\mathbb{E}\left[\mathcal{D}_n^{(i)}\right] &= H_i + H_{n+1-i} - 2, \qquad i = 1, \ldots, n; \\ \mathbb{E}\left[\mathcal{L}_n^{(i)}\right] &= H_{i-1} + H_{n+1-i}, \qquad i = 1, \ldots, n+1; \\ \mathbb{E}\left[\mathcal{S}_n^{(i)}\right] &= \mathbb{E}\left[\mathcal{L}_n^{(i)} + \mathcal{L}_n^{(i+1)} - 2(\mathcal{D}_n^{(i)} + 1)\right] = 2 - \frac{1}{i} - \frac{1}{n+1-i}, \qquad i = 1, \ldots, n;\end{aligned}$$

where $H_n = \sum_{1 \le j \le n} 1/j = \ln n + \gamma + \mathcal{O}(1/n)$ denotes the n-th harmonic number, and $\gamma = 0.577\ldots$ is Euler's constant.

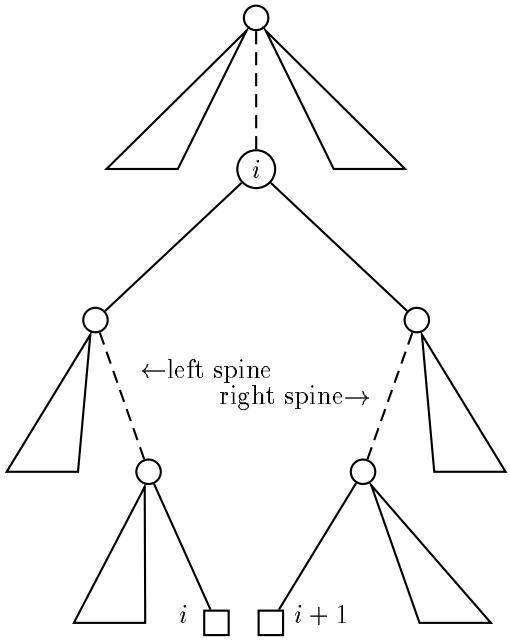


Figure 3: Left and right spines of the i-th internal node.

To begin with, let $S_n^{(i)}$ and $U_n^{(i)}$ be the number of comparisons in a successful search for the i-th key and the number of comparisons in a unsuccessful search for a key in the i-th interval of a tree of size n, respectively. It is clear that

$$S_n^{(i)} = \mathcal{D}_n^{(i)} + 1, \qquad i = 1, \ldots, n.$$
$$U_n^{(i)} = \mathcal{L}_n^{(i)}, \qquad i = 1, \ldots, n+1.$$

Let us consider now the cost of an insertion in the i-th interval ($1 \leq i \leq n+1$) of a tree of size n. If this cost is measured as the number of visited nodes, then its expected value is $\mathbb{E}\left[\mathcal{L}_n^{(i)}\right] + 1$, since the visited nodes are the same as those visited in an unsuccessful search that ends at the i-th external node, plus the new node. However, the insertion of a new item has two clearly differentiated phases and the cost of the stages differ in each of these two phases. Before the insertion at root, we generate a random number in each stage, compare it to another integer, and (except in the last step) compare the new key with the key at the current node. Notice that a top-down (non-recursive) implementation of the insertion algorithm would not update the pointer to the current node, as Algorithm 1 does. In each stage of the insertion at root, a comparison between keys and updating of pointers take place, but no random number is generated.

Hence, for a more precise estimate of the cost of an insertion, we will divide this cost into two contributions: the cost of the descent until the new item reaches its final position, $R_n^{(i)}$, plus the cost of restructuring the tree beneath, that is, the cost of the insertion at the root, $I_n^{(i)}$. We measure these quantities as the number of steps or visited nodes in each. Consider the tree after the insertion. The number of nodes in the path from the root of the tree to the new item is $R_n^{(i)}$. The nodes visited while performing the insertion at root are those in the left and the right spines of the subtree whose root is the i-th node. Since the tree produced by the insertion is random, we have that

$$R_n^{(i)} = \mathcal{D}_{n+1}^{(i)} + 1, \qquad I_n^{(i)} = \mathcal{S}_{n+1}^{(i)}, \qquad i = 1, \ldots, n+1.$$

As expected, $\mathbb{E}\left[R_n^{(i)} + I_n^{(i)}\right] = \mathbb{E}\left[\mathcal{L}_n^{(i)}\right] + 1$. A more precise estimation of the expected cost of an insertion in the i-th interval is then

$$\alpha\,\mathbb{E}\left[R_n^{(i)}\right] + \beta\,\mathbb{E}\left[I_n^{(i)}\right] = \alpha\left(H_i + H_{n+2-i} - 1\right) + \beta\left(2 - \frac{1}{i} - \frac{1}{n+2-i}\right),$$

where α and β are constants that reflect the different costs of the stages in each of the two phases of the insertion algorithm. Notice that the expected cost of the insertion at root phase is $\mathcal{O}(1)$, since less than two rotation-like operations take place (on average).

The cost of the deletion, measured as the number of visited keys, of the i-th key of a tree of size n is also easy to analyze. We can divide it into two contributions, as in the case of insertions: the cost of finding the key to be deleted, $F_n^{(i)}$, plus the cost of the join phase, $J_n^{(i)}$. Since the input tree is random, we have that

$$F_n^{(i)} = \mathcal{D}_n^{(i)} + 1, \qquad J_n^{(i)} = \mathcal{S}_n^{(i)}, \qquad i = 1, \ldots, n.$$

Notice that the number of visited nodes while deleting a key is the same as while inserting it, because we visit the same nodes. The expected number of local updates per deletion is also less than two. A more precise estimation of the expected cost of the deletion of the i-th element is

$$\alpha'\left(H_i + H_{n+1-i} - 1\right) + \beta'\left(2 - \frac{1}{i} - \frac{1}{n+1-i}\right),$$

where α' and β' are constants that reflect the different costs of the stages in each of the two phases of the deletion algorithm. Altogether, the expected cost of any search (whether successful or not) and the expected cost of any update operation is always $\Theta(\log n)$.

The algorithms in the next section can also be analyzed in a straightforward manner. It suffices to relate their performance to the probabilistic behavior of well known quantities like the depth of internal or external nodes in random BSTs, like we have done here.

5. OTHER OPERATIONS

5.1. Duplicate keys. In Section 2 we have assumed that whenever we insert a new item in a RBST its key is not present in the tree. An obvious way to make sure that this is always the case is to perform a search of the key, and then insert it only if it were not present. But there is an important waste of time if this is naïvely implemented. There are several approaches to cope with the problem efficiently.

The bottom-up approach performs the search of the key first, until it either finds the sought item or reaches an external node. If the key is already present the algorithm does nothing else. If the search was unsuccessful, the external node is replaced with a new internal node containing the item. Then, zero or more single rotations are made, until the new item gets into its final position; the rotations are done as long as the random choices taken with the appropiate probabilities indicate so. This is just like running the insertion at root algorithm backwards: we have to stop rotations at the same point where we would have decided to perform an insertion at root. We leave the details of this kind of implementations as an exercise.

There is an equivalent recursive approach that uses a variant of split that does nothing (does not split) if it finds the key in the tree to be split. The sequence of recursive calls signal back such event and the insertion is not performed at the point where the random choice indicated so.

Yet there is another solution, using the top-down approach, which is more efficient than the other solutions considered before. We do the insertion almost in the usual way, with two variations:

1. The insertion at root has to be modified to remove any duplicate of the key that we may find below (and we will surely find it when splitting the tree). This is easy to achieve with a slight modification of the procedure split;

2. If we find the duplicate while performing the first stage of the insertion (that is, when we are finding a place for the inserted key), we have to decide whether the key remains at the position where it has been found, or we push it down.

The reason for the second variation is that, if we never pushed down a key which is repeatedly inserted, then this key would promote to the root and have more chances than other keys to become the root or nearby (see Subsection 5.3). The modified insertion algorithm is exactly like Algorithm 1 given in Section 2, except that it now includes

```
if (x == T→key)
    return push_down(T);
```

after the comparison `r == n` has failed.

In order to push down an item, we basically insert it again, starting from its current position. The procedure push_down(T) (see Algorithm 5) pushes down the root of the tree T; in each step, we decide either to finish the process, or to push down the root to the left or to the right, mimicking single rotations. The procedure push_down(T) follows next theorem.

Theorem 5.1 *Let T be a BST such that its root is some known key x, and its left and right subtrees are random BSTs. Then* push_down(T) *produces a completely random BST (without information on the root of the tree).*

Algorithm 5 Push down

```
bst push_down(bst T) {
/* T ≠ □ */
  bst P;
  int m, n, r, total;

  m = T→left→size; n = T→right→size;
  total = m + n;
  r = random(0, total);
  if (r < m) {              /* with probability m/(m+n+1) */
      P = T→left;
      T→left = T→left→right;
      P→right = push_down(T);
      return P;
  }
  else if (r < total) {     /* with probability n/(m+n+1) */
      P = T→right;
      T→right = T→right→left;
      P→left = push_down(T);
      return P;
  }
  return T;                 /* with probability 1/(m+n+1) */
}
```

We shall not prove it here, but the result above allows us to generalize Theorem 2.1 to cope with the insertion of repeated keys.

Theorem 5.2 *If T is a random BST that contains the set of keys K and x is any key (that may or may not belong to K), then* insert(x, T) *produces a random BST containing the set of keys $K \cup \{x\}$.*

Although, for the sake of clarity, we have given here a recursive implementation of the insertion and push_down procedures, it is straightforward to obtain efficient iterative implementations of both procedures, with only additional constant auxiliary space (no stack) and without using pointer reversal.

5.2. Set operations. We consider here three set operations: union, intersection and difference.

Given two trees A and B, union(A, B) returns a BST that contains the keys in A and B, deleting the duplicate keys. If both trees are empty the result is also the empty tree. Otherwise, a root is selected from the roots of the two given trees, say, we select $a = A \rightarrow$ key. Then, the tree whose root was not selected is split w.r.t. the selected root. Following the example, we split B with respect to a, yielding $B_<$ and $B_>$. Finally, we recursively perform the union of the left subtree of A with $B_<$ and the union of the right subtree of A with $B_>$. The resulting unions are then attached to the common root a. If we select $b = B \rightarrow$ key to be the root of the resulting tree, then we check if b was already in A. If this is the case, then b was duplicate and we push it down. Doing so we compensate the fact that b has had twice the chances of being the root of the final tree than any non-duplicate key.

Algorithm 6 Union

```
bst union(bst A, bst B) {
  bst Al, Ar, Bl, Br;
  int m, n, u, total, rep;

  m = A→size; n = B→size; total = m + n;
  if (total == 0) return □;
  u = random(1, total);
  if (u <= m) {              /* with probability m/(m+n) */
      split(A→key, B, &Bl, &Br);
      A→left = union(A→left, Bl);
      A→right = union(a→right, Br);
      return A;
  }
  else {                    /* with probability n/(m+n) */
      rep = split(B→key, A, &Al, &Ar);
      B→left = union(B→left, Al);
      B→right = union(B→right, Ar);
      if (rep) return push_down(B);
       else    return B;
  }
}
```

This algorithm uses a slight variant of the procedure split, which behaves like the procedure described in Section 2, but also removes any duplicate of x, the given key, and returns 0 if such a duplicate has not been found and 1 otherwise.

The correctness of the algorithm is clear. A bit more involved proof shows that union(A, B) is a random BST if both A and B are random BSTs. The expected cost of the union is $\Theta(m+n)$, where $m = A \rightarrow$ size and $n = B \rightarrow$ size.

The intersection and set difference of two given trees A and B are computed in a similar vein. Algorithms 7 and 8 always produce a RBST, if the given trees are RBSTs. Notice that they do not need to use randomness. As in the case of the union of two given RBSTs, their expected performance is $\Theta(m+n)$. Both intersection and difference use a procedure free_tree(T), which returns all the nodes in the tree T to the free storage.

Algorithm 7 Intersection

```
bst intersection(bst A, bst B) {
  bst Bl, Br, il, ir;
  int rep;

  if (A == □) {
      free_tree(B);
      return □;
  }
  rep = split(A→key, B, &Bl, &Br);
  il = intersection(A→left, Bl);
  ir = intersection(A→right, Br);
  if (rep) {
      A→left = il;
      A→right = ir;
      return A;
  }
  else {
      free_node(A);
      return join(il, ir);
  }
}
```

As a final remark, notice that for all our set algorithms we have assumed that their parameters had to be not only combined to produce the output, but *destroyed*. It is easy to write slightly different versions that preserve their inputs.

5.3. Self-adjusting strategies. In Subsection 5.1 we have mentioned that to cope with the problem of duplicate keys, we either make sure that the input tree is not modified at all if the new key is already present, or we add a mechanism to "push down" duplicate keys. The reason is that, if we did not have such a mechanism, a key that were repeatedly inserted would promote to the root. Such a key would be closer to the root with probability larger than the one corresponding to the random BST model, since that key would always be inserted at the same level that it already was or closer to the root.

This observation immediately suggests the following self-adjusting strategy: each time a key is searched for, the key is inserted again in the tree, but now it will not be pushed down if found during the search phase, we just stop the insertion. As a consequence, frequently accessed keys will get closer and closer to the root (because they are never inserted at a level below the level they were before the access), and the average cost per access will decrease.

Algorithm 8 Difference

```
bst difference(bst A, bst B) {
  bst Bl, Br, dl, dr;
  int rep;

  if (a == □) {
      free_tree(B);
      return □;
  }
  rep = split(A→key, B, &Bl, &Br);
  dl = difference(A→left, Bl);
  dr = difference(A→right, Br);
  if (rep) {
      free_node(A);
      return join(dl, dr);
  }
  else {
      A→left = dl;
      A→right = dr;
      return A;
  }
}
```

We can rephrase the behavior of this self-adjusting strategy as follows: we go down the path from the root to the accessed item; at each node of this path we decide, with probability $\frac{1}{n}$, whether we replace that node with the accessed item and rebuild the subtree rooted at the current node, or we continue one level below, unless we have reached the element we were searching for. Here, n denotes the size of the subtree rooted at the current node. When we decide to replace some node by the sought element, the subtree is rebuilt and we should take care to remove the duplicate key.

Since it is not now our goal to maintain random BSTs, the probability of replacement can be totally arbitrary, not necessarily equal to $\frac{1}{n}$. We can use any function $0 < \alpha(n) \leq 1$ as the probability of replacement. If $\alpha(n)$ is close to 1 then the self-adjusting strategy reacts quickly to the pattern of accesses. The limit case $\alpha(n) = 1$ is the well known *move-to-root* strategy [2], because we always replace the root with the most recently accessed item and rebuild the tree in such a way that the result is the same as if we had moved the accessed item up to the root using single rotations. If $\alpha(n)$ is close to 0, convergence occurs at a slower rate, but it is more difficult to fool the heuristic with transient patterns. Obviously, the self-adjusting strategy that we have originally introduced is the one where $\alpha(n) = \frac{1}{n}$.

Notice that the different self-adjusting strategies that we consider here are just characterized by their corresponding function α; no matter what $\alpha(n)$ is we have the following result, that will be proved in Section 7.

Theorem 5.3 *Let $X = \{x_1, \ldots, x_N\}$ and let T be a BST that contains the items in X. Furthermore, consider a sequence of independent accesses to the items in X such that x_i is accessed with probability p_i. If we use any of the self-adjusting strategies described above to modify T at each access, the asymptotic probability distribution is the same for all strategies and independent of $\alpha(n)$, namely, it is the one for the move-to-root strategy.*

We know thus that after a large number of accesses have been made and the tree reorganized according to any of the heuristics described above, the probability that x_i is an ancestor of x_j is, for $i < j$, $p_i/(p_i + \cdots + p_j)$. Moreover, the average cost of a successful search is

$$\overline{C}_N = 1 + 2 \sum_{1 \leq i < j \leq N} \frac{p_i p_j}{p_i + \cdots + p_j}.$$

In the paper of Aragon and Seidel [3], they describe another self-adjusting strategy for randomized treaps: each time an item is accessed, a new random priority is computed for that item; if the new priority is larger than its previous priority, the older priority is replaced by the newer and the item is rotated upwards until heap order in the tree is restablished. This strategy does not correspond to any of the strategies in the family that we have discussed before, but it also satisfies Theorem 5.3. The analysis of the strategy suggested by Aragon and Seidel becomes simpler once we notice that its asymptotic distribution is the same as that of move-to-root.

The main difference between the self-adjusting strategies that we have discussed here is that they have different rates of convergence to the asymptotic distribution. In [2], Allen and Munro show that, for move-to-root, the difference between the average cost in the asymptotic distribution and the average cost after t accesses is less than 1 if $t \geq \frac{N \log N}{e}$ and the initial tree is random. If $\alpha(n) < 1$, it is clear that the rate of convergence should be slower than that for move-to-root. We have been able to prove that if $\alpha(n)$ is constant, then the result of Allen and Munro holds for $t \geq \frac{N \log N}{\alpha \cdot e}$. We conjecture that this is also true for any $\alpha(n)$, but we have not been able to prove it. On the other hand, Aragon and Seidel did not address the question of the rate of convergence for their strategy and it seems also quite difficult to compute it.

In a more practical setting, the strategy of Aragon and Seidel has a serious drawback, since frequently accessed items get priorities close to 1. Then the lenght of the prorities tends to infinity as the number of accesses grows. The situation gets even worse if the p_i's, the probabilities of access, changed from time to time, since their algorithms react very slowly after a great number of accesses have been made.

6. Implementation issues

6.1. Number of random bits. Let us now consider the complexity of our algorithms from the point of view of the number of needed random bits per operation.

For insertions, a random number must be generated for each node visited before the placement of the new item at the root of some subtree is made. If the currently visited node y is the root of a subtree of size m, we would generate a random number between 0 and m; if this random number is m then we insert at root the new element, otherwise the insertion continues either on the left or right subtree of y. If random numbers are generated from high order to low order bits and compared with prefixes of the binary representation of m, then the expected number of generated random bits per node is $\Theta(1)$ —most of the times the comparison fails and the insertion continues at the appropriate subtree—. Recall that the expected number of nodes visited before we insert at root the new item is $\Theta(\log n)$. The total expected number of random bits per insertion is thus $\Theta(\log n)$. Further refinements could reduce the expected total number of random bits to $\Theta(1)$. Nevertheless, the reduction is achieved at the cost of performing rather complicated arithmetic operations for each visited node during the insertion.

In the case of deletions, the expected lenght of left and right spines of the node to be deleted is constant, so the expected number of random bits is also constant.

A practical implementation, though, will use the straightforward approach. Typical random number generators produce one random word (say of 32 or 64 bits) quite efficiently and that is enough for most ordinary applications.

6.2. Non-recursive top-down implementation of the operations. We have given recursive implementations of the insertion and deletion algorithms, as well as for other related procedures. It is not very difficult to obtain efficient non-recursive implementations of all considered operations, with two interesting features: they only use a constant amount of auxiliary space and they work in pure top-down fashion. Thus these non-recursive implementations do not use stacks or pointer reversal, and never traverse backwards a search path. We also introduce an apparently new technique to manage subtree sizes without modifying the top-down nature of our algorithms (see next Subsection for more details). As an example, Algorithm 10 in Appendix A shows a non-recursive implementation of the deletion algorithm, including the code to manage subtree sizes.

6.3. Management of subtree sizes & space complexity. Up to now, we have not considered the problem of managing the sizes of subtrees. In principle, each node of the tree has to store information from which we can compute the size of the subtree rooted at that particular node. If the size of each subtree is stored at its root then we face the problem of updating this information for all nodes in the path followed during insertion and deletion operations. The problem gets more complicated if one has to cope with insertions that may not increase the size of the tree (when the element was already in the tree) and deletions that may not decrease the size (when the element to be deleted was not in the tree).

A good solution to this problem is to store at each node the size of its left son or its right son, rather than the size of the subtree rooted at that node. An additional *orientation* bit indicates whether the size is that of the left or the right subtree. If the total size of the tree is known and we follow any path from the root downwards, it is easy to see that, for each node in this path, we can trivially compute the sizes of its two subtrees, given the size of one of them and its total size. This trick notably simplifies the management of the size information: for instance, while doing an insertion or deletion, we change the size and orientation bit of the node from left to right if the operation continues in the left subtree and the orientation bit was 'left'; we change from right to left in the symmetric case. When the insertion or deletion finishes, only the global counter of the size of tree has to be changed if necessary (see Algorithm 10 in Appendix A). Similar rules can be used for the implementation of splits and joins.

We emphasize again that the information about subtree sizes —required by all our algorithms— can be advantageously used for operations based on ranks, like searching or deleting the i-th item. In fact, since we store either the size of the left or the right subtree of each node, rank operations are easier and more efficient. By contrast, if the size of the subtree were stored at the node, one level of indirection (examining the left subtree root's size, for instance) would be necessary to decide if the rank operation had to continue either to the left or to the right.

Last but not least, storing the sizes of subtrees is not too demanding. The expected total number of bits that are necessary to store the sizes is $\Theta(n)$ (this result is the solution of the corresponding easy divide-and-conquer recurrence). This is well below the $\Theta(n \log n)$ number of bits needed for pointers and keys.

7. Formal framework and algebraic proofs

7.1. Randomized algorithms. While the behaviour of deterministic algorithms can be neatly described by means of algebraic equations, this approach has never been used for the study of randomized algorithms in previous works. We now present an algebraic-like notation that allows a concise and rigorous description and further reasoning about randomized algorithms, following the ideas introduced in [18].

The main idea is to consider any randomized algorithm F as a function from the set of inputs A to the set of *probability functions* (or PFs, for short) over the set of outputs B. We say that f is a probability function over B if and only if $f : B \to [0, 1]$ and $\sum_{y \in B} f(y) = 1$, as usual.

Let $f_1, \ldots, f_n$ be PFs over B and let $\alpha_1, \ldots, \alpha_n \in [0,1]$ be such that $\sum_{1 \leq i \leq n} \alpha_i = 1$. Consider the following process:

1. Choose some PF from the set $\{f_i\}_{i=1,\ldots,n}$ in such a way that each f_i has a probability α_i of being selected.
2. Choose one element from B according to the probabilities defined by the selected f_i, namely, choose $y \in B$ with probability $f_i(y)$.

Let h be the PF over B related to the process above: for any $y \in B$, $h(y)$ is the probability that we select the element y as the outcome of the whole process. Clearly, h is the *linear combination* of the f_i's with coefficients α_i's:

$$h = \alpha_1 \cdot f_1 + \cdots + \alpha_n \cdot f_n = \sum_{1 \leq i \leq n} \alpha_i \cdot f_i. \tag{2}$$

The linear combination of PFs modelizes the common situation where a randomized algorithm h makes a random choice and depending on it performs some particular task f_i with probability α_i.

Let f be a PF over B such that $f(b) = 1$ for some $b \in B$ (i.e. $f(y) = 0$ for all $y \neq b$). Then we will write $f = b$. Thus we are considering each element in B as a PF over B itself:

$$b(y) = \begin{cases} 1, & \text{if } y = b, \\ 0, & \text{otherwise.} \end{cases} \tag{3}$$

This convention is useful, since it will allow us to uniformly deal with both randomized and deterministic algorithms.

Let f be any PF over $B = \{y_i\}_i$, and let p_i denote $f(y_i)$. Then the convention above allows us to write $f = \sum_i p_i \cdot y_i$, since for any $y_j \in B$ we have that $f(y_j) = [\sum_i p_i \cdot y_i](y_j) = \sum_i p_i \cdot y_i(y_j) = p_j$. Taking into account that $p_i = f(y_i)$, we get the following equality, that may look amazing at first:

$$f = \sum_{y \in B} f(y) \cdot y. \tag{4}$$

Let F be a randomized algorithm from the input set A to the output set B. Fix some input $x \in A$. We denote by $F(x)$ the PF over B such that, for any $y \in B$, $[F(x)](y)$ is the probability that the algorithm F outputs y when given input x. Let $\{y_1, \ldots, y_m\}$ be the set of possible outputs of F, when x is the input given to F, and let p_i be the probability that y_i is the actual output. Then, using the notation previously introduced, we may write

$$F(x) = p_1 \cdot y_1 + \cdots + p_m \cdot y_m. \tag{5}$$

Notice that, if for some $a \in A$ the result of $F(a)$ is always a fixed element $b \in B$, then the expression above reduces to $F(a) = b$.

Finally, we characterize the behaviour of the sequential composition of randomized algorithms. Let g be a PF over A, and let F be a function from A to B. By $F(g)$ we will denote the PF over B such that $[F(g)](y)$ is the probability that y is the output of algorithm F when the input is selected according to g. It turns out that $F(g)$ is easily computed from g and the PFs $F(x)$ for each element x in A.

$$F(g) = F\left(\sum_{x \in A} g(x) \cdot x\right) = \sum_{x \in A} g(x) \cdot F(x). \tag{6}$$

Recall that a single element a from A may be considered as a PF over A, and then the definition above is consistent with the case $g = a$, that is, when the input of F is fixed, since $\sum_{x \in A} a(x) \cdot F(x) = F(a)$.

We shall also use Iverson's bracket convention for predicates, that is, $[\![P]\!]$ is 1 if the predicate P is true, and 0 otherwise. This convention allows expressing the definitions by cases as linear combinations.

7.2. Binary search trees and permutations. We now introduce some definitions and notation concerning the main types of objects we are coping with: keys, permutations and trees.

Given a finite set of keys K, we shall denote $\mathcal{B}(K)$ the set of all BSTs that contain all the keys in K. For simplicity, we will assume that $K \subset \mathbb{N}$. The empty tree is denoted by $\square$. We will sometimes omit drawing empty subtrees, to make the figures simpler. A tree T with root x, left subtree L and right subtree R is depicted

$$T = \begin{array}{ccc} & x & \\ / & & \backslash \\ L & & R \end{array}.$$

Similarly, we denote $\mathcal{P}(K)$ the set of all permutations (sequences without repetition) of the keys in K. We shall use the terms sequence and permutation with the same meaning for the rest of this paper. The empty sequence is denoted by λ and $U \,|\, V$ denotes the concatenation of the sequences U and V (provided that U and V do not have common elements).

The following equations relate sequences in $\mathcal{P}(K)$ and BSTs in $\mathcal{B}(K)$. Given a sequence S, $\mathsf{bst}(S)$ is the BST resulting after the standard insertion of the keys in S from left to right into an initially empty tree.

$$\mathsf{bst}(\lambda) = \square, \qquad \mathsf{bst}(x \,|\, S) = \begin{array}{ccc} & x & \\ / & & \backslash \\ \mathsf{bst}(\mathsf{sep}_<(x,S)) & & \mathsf{bst}(\mathsf{sep}_>(x,S)) \end{array}, \tag{7}$$

where the algebraic function $\mathsf{sep}_<(x, S)$ returns the subsequence of elements in S smaller than x, and $\mathsf{sep}_>(x, S)$ returns the subsequence of elements in S larger than x. Both functions respect the order in which the keys appear in S.

Let Random_Perm be a function such that, given a set K with $n \geq 0$ keys, returns a randomly chosen permutation of the keys in K. It can be compactly written as follows:

$$\mathsf{Random\_Perm}(K) = \sum_{P \in \mathcal{P}(K)} \frac{1}{n!} \cdot P. \tag{8}$$

Random BSTs can be defined in the following succint way, which turns out to be equivalent to the assumption that a RBST of size n is built by performing n random insertions in an initially empty tree:

$$\mathsf{RBST}(K) = \mathsf{bst}(\mathsf{Random\_Perm}(K)) = \sum_{x \in K} \frac{1}{n} \cdot \begin{array}{ccc} & x & \\ / & & \backslash \\ \mathsf{RBST}(K_{<x}) & & \mathsf{RBST}(K_{>x}) \end{array}. \tag{9}$$

The last expression in the equation above is clearly equivalent to the one we gave in Section 2, provided we define its value to be $\square$ if $n = 0$.

For instance,

$$\begin{aligned} \mathsf{RBST}(\{1,5,7\}) &= \mathsf{bst}(\mathsf{Random\_Perm}(\{1,5,7\})) \\ &= \mathsf{bst}\left(\frac{1}{6}\cdot 157 + \frac{1}{6}\cdot 175 + \frac{1}{6}\cdot 517 + \frac{1}{6}\cdot 571 + \frac{1}{6}\cdot 715 + \frac{1}{6}\cdot 751\right) \end{aligned}$$

$$= \frac{1}{6}\cdot \begin{array}{ccc} & 1 & \\ & & \backslash \\ & 5 & \\ / & & \backslash \\ & 7 & \end{array} + \frac{1}{6}\cdot \begin{array}{ccc} & 1 & \\ & & \backslash \\ & 7 & \\ / & & \\ 5 & & \end{array} + \frac{1}{3}\cdot \begin{array}{ccc} & 5 & \\ / & & \backslash \\ 1 & & 7 \end{array} + \frac{1}{6}\cdot \begin{array}{ccc} & 7 & \\ / & & \\ 1 & & \\ & \backslash & \\ & 5 & \end{array} + \frac{1}{6}\cdot \begin{array}{ccc} & 7 & \\ & / & \\ & 5 & \\ / & & \\ 1 & & \end{array}.$$

We introduce now three useful algebraic functions over sequences: rm, shuffle and equiv. The first function removes a key x from a sequence $S \in \mathcal{P}(K)$, if present, without changing the relative order of the other keys. For instance, $\mathsf{rm}(3, 2315) = 215$, $\mathsf{rm}(4, 2315) = 2315$.

The function shuffle produces a random shuffling of two given sequences with no common elements. Let K_1 and K_2 be two disjoint sets with m and n keys, respectively. Let $U = u_1 \,|\, \ldots \,|\, u_m \in \mathcal{P}(K_1)$ and $V = v_1 \,|\, \ldots \,|\, v_n \in \mathcal{P}(K_2)$. We define $\mathcal{S}(U, V)$ as the set of all the permutations of the keys in $K_1 \cup K_2$ that could be obtained by shuffling U and V, without changing the relative order of the keys of U and V. Hence, $\mathcal{S}(U, V)$ is the set of all $Y = y_1 \,|\, \ldots \,|\, y_{m+n} \in \mathcal{P}(K_1 \cup K_2)$ such that if $y_i = u_j$ and $y_{i'} = u_{j'}$, then $i < i'$ if and only if $j < j'$ (and the equivalent condition for the keys of V). For instance, $\mathcal{S}(21, ba) = \{21ba, 2b1a, 2ba1, b21a, b2a1, ba21\}$. The number of elements in $\mathcal{S}(U, V)$ is clearly equal to $\binom{m+n}{m}$. Therefore, we can rigorously define shuffle as a function that, given U and V, returns a randomly choosen element from $\mathcal{S}(U, V)$. For instance, $\mathsf{shuffle}(21, ba) = \frac{1}{6} \cdot 21ba + \frac{1}{6} \cdot 2b1a + \frac{1}{6} \cdot 2ba1 + \frac{1}{6} \cdot b21a + \frac{1}{6} \cdot b2a1 + \frac{1}{6} \cdot ba21$. The algebraic equations for shuffle are

$$\begin{aligned} &\mathsf{shuffle}(\lambda, \lambda) = \lambda, \qquad \mathsf{shuffle}(\lambda, v \,|\, V) = v \,|\, V, \qquad \mathsf{shuffle}(u \,|\, U, \lambda) = u \,|\, U, \\ &\mathsf{shuffle}(u \,|\, U, v \,|\, V) = \frac{m}{m+n} \cdot u \,|\, \mathsf{shuffle}(U, v \,|\, V) + \frac{n}{m+n} \cdot v \,|\, \mathsf{shuffle}(u \,|\, U, V), \end{aligned} \tag{10}$$

where m and n are the sizes of $u \,|\, U$ and of $v \,|\, V$, respectively. It is no difficult to prove by induction that this definition of the function shuffle is correct.

Let K be a set of keys and x a key not in K. We can use shuffle to define the function Random_Perm in the following inductive way, equivalent to definition (8):

$$\begin{aligned} \mathsf{Random\_Perm}(\emptyset) &= \lambda, \\ \mathsf{Random\_Perm}(K \cup \{x\}) &= \mathsf{shuffle}(x, \mathsf{Random\_Perm}(K)). \end{aligned} \tag{11}$$

We define equiv as a function such that given input a sequence $S \in \mathcal{P}(K)$, it returns a randomly chosen element from the set of sequences that produce the same BST as S, i.e.

$$\mathcal{E}(S) = \{E \in \mathcal{P}(K) \mid \mathsf{bst}(E) = \mathsf{bst}(S)\}.$$

For example, $\mathcal{E}(3124) = \{3124, 3142, 3412\}$ and $\mathsf{equiv}(3124) = \frac{1}{3} \cdot 3124 + \frac{1}{3} \cdot 3142 + \frac{1}{3} \cdot 3412$, since $\mathsf{bst}(3124) = \mathsf{bst}(3142) = \mathsf{bst}(3412)$ and no other permutation of the keys $\{1, 2, 3, 4\}$ produces the same tree. Using the function shuffle, the equational definition of equiv is almost trivial:

$$\mathsf{equiv}(\lambda) = \lambda, \qquad \mathsf{equiv}(x \,|\, S) = x \,|\, \mathsf{shuffle}(\mathsf{equiv}(\mathsf{sep}_<(x, S)), \mathsf{equiv}(\mathsf{sep}_>(x, S))). \tag{12}$$

7.3. The algorithms. In this subsection we give equational definitions for the basic algorithms in our study: insert, insert_at_root, delete, join, etc.

Using our notation, the algebraic equations describing the behaviour of insert are

$$\mathsf{insert}(x, \square) = \begin{array}{c} x \\ \square \quad \square \end{array},$$

$$\mathsf{insert}\left(x, \begin{array}{c} y \\ L \quad R \end{array}\right) = \frac{1}{n+1} \cdot \mathsf{insert\_at\_root}\left(x, \begin{array}{c} y \\ L \quad R \end{array}\right) \tag{13}$$

$$+ \frac{n}{n+1} \cdot \left([\![x < y]\!] \cdot \begin{array}{c} y \\ \mathsf{insert}(x, L) \quad R \end{array} + [\![x > y]\!] \cdot \begin{array}{c} y \\ L \quad \mathsf{insert}(x, R) \end{array} \right),$$

assuming that we never insert a key that was already in the tree.

The function insert_at_root can be defined as follows.

$$\mathsf{insert\_at\_root}(x,T) = \quad \overset{x}{\diagup\ \diagdown}\ \begin{matrix}\mathsf{split}_<(x,T) & \mathsf{split}_>(x,T)\end{matrix}, \tag{14}$$

where $\mathsf{split}_<$ and $\mathsf{split}_>$ are functions that, given a tree T and a key $x \notin T$, return a BST with the keys in T less than x and a BST with the keys in T greater than x, respectively. The algebraic equations for the function $\mathsf{split}_<$ are

$$\begin{aligned}&\mathsf{split}_<(x,\square) = \square,\\ &\mathsf{split}_<\left(x,\ \overset{y}{\diagup\ \diagdown}\ \begin{matrix}L & R\end{matrix}\right) = [\![x<y]\!]\cdot \mathsf{split}_<(x,L)\ \ +\ \ [\![x>y]\!]\cdot\ \overset{y}{\diagup\ \diagdown}\ \begin{matrix}L & \mathsf{split}_<(x,R)\end{matrix}.\end{aligned} \tag{15}$$

The function $\mathsf{split}_>$ satisfies symmetric equations. We define

$$\mathsf{split}(x,T) = [\mathsf{split}_<(x,T), \mathsf{split}_>(x,T)].$$

Let us now shift our attention to the deletion algorithm. Its algebraic form is

$$\begin{aligned}&\mathsf{delete}(x,\square) = \square,\\ &\mathsf{delete}\left(x,\ \overset{y}{\diagup\ \diagdown}\ \begin{matrix}L & R\end{matrix}\right) = [\![x<y]\!]\cdot\ \overset{y}{\diagup\ \diagdown}\ \begin{matrix}\mathsf{delete}(x,L) & R\end{matrix}\ +\ [\![x>y]\!]\cdot\ \overset{y}{\diagup\ \diagdown}\ \begin{matrix}L & \mathsf{delete}(x,R)\end{matrix}\\ &\qquad\qquad +\,[\![x=y]\!]\cdot\mathsf{join}(L,R).\end{aligned} \tag{16}$$

The probabilistic behaviour of join can in turn be described as follows, when at least one of its arguments is an empty tree

$$\mathsf{join}(\square,\square) = \square, \qquad \mathsf{join}(L,\square) = L, \qquad \mathsf{join}(\square,R) = R.$$

On the other hand, when both arguments are non-empty trees, with sizes $m = L \to \mathsf{size} > 0$ and $n = R \to \mathsf{size} > 0$, respectively, we have

$$\mathsf{join}(L,R) = \frac{m}{m+n}\cdot\ \overset{a}{\diagup\ \diagdown}\ \begin{matrix}L_\ell & \mathsf{join}(L_r,R)\end{matrix}\ +\frac{n}{m+n}\cdot\ \overset{b}{\diagup\ \diagdown}\ \begin{matrix}\mathsf{join}(L,R_\ell) & R_r\end{matrix}, \tag{17}$$

where $a = L \to \mathsf{key}$, $L_\ell = L \to \mathsf{left}$, $L_r = L \to \mathsf{right}$ and $b = R \to \mathsf{key}$, $R_\ell = R \to \mathsf{left}$, $R_r = R \to \mathsf{right}$.

7.4. The proofs. We consider here several of the results that we have already seen in previous sections as well as some intermediate lemmas that are interesting on their own. We will not provide proofs for all them, for the sake of brevity. Only the proof of Lemma 7.2 and Theorem 7.2 will be rather detailed; in other cases, the proofs will be sketchy or just missing. However, the ones given here should suffice to exemplify the basic manouvers that the algebraic notation allows and typical reasoning using it.

The following lemma describes the result of split when applied to a fixed BST.

Lemma 7.1 *Let S be any permutation of keys and let x be any key not in S. Then*

$$\mathsf{split}(x, \mathsf{bst}(S)) = [\mathsf{bst}(\mathsf{sep}_<(x, S)), \mathsf{bst}(\mathsf{sep}_>(x, S))].$$

From this lemma we can describe the behaviour of split when applied to a random BST. Our next theorem is nothing but Lemma 2.1, now in the formal setting of this section.

Theorem 7.1 *Let K be any set of keys and let x be any key not in K. Let $K_{<x}$ and $K_{>x}$ denote the set with the keys in K less than x and the set with the keys in K greater than x, respectively. Then*

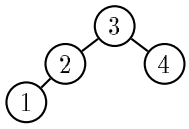
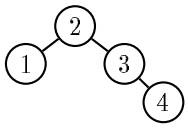
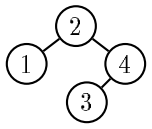
$$\mathsf{split}(x, \mathsf{RBST}(K)) = [\mathsf{RBST}(K_{<x}), \mathsf{RBST}(K_{>x})].$$

Lemma 7.1 relates split with sep, the analogous function over sequences. Our next objective is to relate insert with shuffle and equiv, the functions that we have defined before. The idea is that the insertion of a new item in a tree T has the same effect as taking at random any of the sequences that would produce T, placing x anywhere in the chosen sequence (an insertion-like operation in a sequence) and then rebuilding the tree. This is formally stated in the next lemma.

Lemma 7.2 *Let S be any permutation of keys and let x be any key not in S. Then*

$$\mathsf{insert}(x, \mathsf{bst}(S)) = \mathsf{bst}(\mathsf{shuffle}(x, \mathsf{equiv}(S))).$$

For instance, using Equations (7), (13), (14) and (15) we get

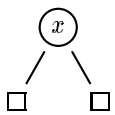
$\mathsf{insert}(3, \mathsf{bst}(241)) = \frac{1}{4} \cdot$  $+ \frac{3}{8} \cdot$  $+ \frac{3}{8} \cdot$  .

On the other hand, by the definitions of shuffle and equiv (Equations (10) and (12)),

$$\begin{aligned}\mathsf{bst}(\mathsf{shuffle}(3, \mathsf{equiv}(241))) &= \mathsf{bst}\left(\mathsf{shuffle}\left(3, \frac{1}{2} \cdot 241 + \frac{1}{2} \cdot 214\right)\right) \\ &= \mathsf{bst}\left(\frac{1}{2} \cdot \mathsf{shuffle}(3, 241) + \frac{1}{2} \cdot \mathsf{shuffle}(3, 214)\right) \\ = \mathsf{bst}\Big(\frac{1}{8} \cdot 3241 + \frac{1}{8} \cdot 2341 + \frac{1}{8} \cdot 2431 + \frac{1}{8} \cdot 2413 + \frac{1}{8} \cdot 3214 + \frac{1}{8} \cdot 2314 + \frac{1}{8} \cdot 2134 + \frac{1}{8} \cdot 2143\Big),\end{aligned}$$

which gives the same result as $\mathsf{insert}(3, \mathsf{bst}(241))$, since the sequences in {3241, 3214} produce the first tree, those in {2341, 2314, 2134} produce the second tree, and the ones in {2431, 2413, 2143} produce the third.

Proof. We prove the lemma by induction on n, the length of S. If $n = 0$,

$\mathsf{insert}(x, \mathsf{bst}(\lambda)) = \mathsf{insert}(x, \square) =$  ,

but on the other hand, $\mathsf{equiv}(\lambda) = \lambda$ and $\mathsf{shuffle}(x, \lambda) = x$, so the lemma is true.
Assume now that $n > 0$, $S = y \mid P$ and x is not present in S. Moreover, let us assume that $x < y$ (the case $x > y$ is very similar).

First of all, notice that, if $y \mid Q$ is any permutation of n keys and x is any key not in $y \mid Q$, then $\mathsf{shuffle}[x, y \mid Q] = \frac{1}{n+1} \cdot x \mid y \mid Q + \frac{n}{n+1} \cdot y \mid \mathsf{shuffle}[x, Q]$. Therefore,

$$\begin{aligned}
& \mathsf{shuffle}[x, \mathsf{equiv}(y \mid P)] \\
\{\text{by definition of } \mathsf{equiv}\} =& \mathsf{shuffle}[x, y \mid \mathsf{shuffle}[\mathsf{equiv}(\mathsf{sep}_<(y, P)), \mathsf{equiv}(\mathsf{sep}_>(y, P))]] \\
\{\text{by the observation above}\} =& \frac{1}{n+1} \cdot x \mid y \mid \mathsf{shuffle}[\mathsf{equiv}(\mathsf{sep}_<(y, P)), \mathsf{equiv}(\mathsf{sep}_>(y, P))] \\
& + \frac{n}{n+1} \cdot y \mid \mathsf{shuffle}[x, \mathsf{shuffle}[\mathsf{equiv}(\mathsf{sep}_<(y, P)), \mathsf{equiv}(\mathsf{sep}_>(y, P))]].
\end{aligned}$$

Now we can use the definition of equiv back in the first line. Furthermore, the $\mathsf{shuffle}$ operation is associative —we do not prove it, but it is easy to do it and rather intuitive—. Therefore,

$$\begin{aligned}
\mathsf{shuffle}[x, \mathsf{equiv}(y \mid P)] =& \frac{1}{n+1} \cdot x \mid \mathsf{equiv}(y \mid P) \\
& + \frac{n}{n+1} \cdot y \mid \mathsf{shuffle}[\mathsf{shuffle}[x, \mathsf{equiv}(\mathsf{sep}_<(y, P))], \mathsf{equiv}(\mathsf{sep}_>(y, P))].
\end{aligned}$$

Distributing through bst, we get

$$\begin{aligned}
\mathsf{bst}(\mathsf{shuffle}[x, \mathsf{equiv}(y \mid P)]) =& \frac{1}{n+1} \cdot \mathsf{bst}(x \mid \mathsf{equiv}(y \mid P)) \\
& + \frac{n}{n+1} \cdot \mathsf{bst}(y \mid \mathsf{shuffle}[\mathsf{shuffle}[x, \mathsf{equiv}(\mathsf{sep}_<(y, P))], \mathsf{equiv}(\mathsf{sep}_>(y, P))]).
\end{aligned} \tag{18}$$

Now we can use the facts that $\mathsf{bst}(x \mid Q) = \mathsf{insert\_at\_root}(x, \mathsf{bst}(Q))$ (this follows from Lemma 7.1) and $\mathsf{bst}(\mathsf{equiv}(Q)) = \mathsf{bst}(Q)$ to manipulate the expression in the first line of (18) as follows.

$$\begin{aligned}
\mathsf{bst}(x \mid \mathsf{equiv}(y \mid P)) =& \mathsf{insert_at_root}(x, \mathsf{bst}(\mathsf{equiv}(y \mid P))) \\
=& \mathsf{insert_at_root}(x, \mathsf{bst}(y \mid P)) \\
=& \mathsf{insert_at_root}(x, \mathsf{bst}(S)).
\end{aligned}$$

Let $A = \mathsf{shuffle}[\mathsf{shuffle}[x, \mathsf{equiv}(\mathsf{sep}_<(y, P))], \mathsf{equiv}(\mathsf{sep}_>(y, P))]$. The definition of bst allows us to write the expression in the second line of (18) as

$$\mathsf{bst}(y \mid A) = \begin{array}{c} y \\ \mathsf{bst}(\mathsf{sep}_<(y, A)) \quad \mathsf{bst}(\mathsf{sep}_>(y, A)) \end{array}.$$

Notice that $\mathsf{shuffle}[x, \mathsf{equiv}(\mathsf{sep}_<(y, P))]$ does only contain keys smaller than y. Therefore,

$$\mathsf{sep}_<(y, A) = \mathsf{shuffle}[x, \mathsf{equiv}(\mathsf{sep}_<(y, P))], \qquad \mathsf{sep}_>(y, A) = \mathsf{equiv}(\mathsf{sep}_>(y, P)).$$

Recall that $\mathsf{shuffle}$ does not modify the relative order of the keys in its parameters. Then,

$$\mathsf{bst}(y \mid A) = \begin{array}{c} y \\ \mathsf{bst}(\mathsf{shuffle}[x, \mathsf{equiv}(\mathsf{sep}_<(y, P))]) \quad \mathsf{bst}(\mathsf{equiv}(\mathsf{sep}_>(y, P))) \end{array}.$$

Applying the inductive hypothesis in the left subtree, and the property $\mathsf{bst}(\mathsf{equiv}(Q)) = \mathsf{bst}(Q)$ in the right one, we have

$$\mathsf{bst}(y \mid A) = \begin{array}{c} y \\ \mathsf{insert}(x, \mathsf{bst}(\mathsf{sep}_<(y, P))) \quad \mathsf{bst}(\mathsf{sep}_>(y, P)) \end{array}.$$

Finally, using the definition of bst and insert, we prove the statement of the lemma:

$$\mathsf{bst}(\mathsf{shuffle}(x, \mathsf{equiv}(S))) = \mathsf{insert}(x, \mathsf{bst}(S)).$$

□

Last lemma is relevant since it paves the way for the proof of one of the main results of this work, namely, Theorem 2.1. We rephrase it again, using the notation introduced so far.

Theorem 7.2 *Let K be any set of keys and let x be any key not in K. Then*

$$\mathsf{insert}(x, \mathsf{RBST}(K)) = \mathsf{RBST}(K \cup \{x\}).$$

Proof. Notice that taking $S \in \mathcal{P}(K)$ at random and then chosing any permutation equivalent to S, is the same as taking $S \in \mathcal{P}(K)$ at random. Then,

$$\begin{aligned}
& \mathsf{insert}(x, \mathsf{RBST}(K)) \\
\{\text{by definition of RBST}\} =& \mathsf{insert}(x, \mathsf{bst}(\mathsf{Random_Perm}(K))) \\
\{\text{by Lemma 7.2}\} =& \mathsf{bst}(\mathsf{shuffle}(x, \mathsf{equiv}(\mathsf{Random_Perm}(K)))) \\
\{\text{by the observation above}\} =& \mathsf{bst}(\mathsf{shuffle}(x, \mathsf{Random_Perm}(K))) \\
\{\text{by definition of Random_Perm}\} =& \mathsf{bst}(\mathsf{Random_Perm}(K \cup \{x\})) \\
\{\text{by definition of RBST}\} =& \mathsf{RBST}(K \cup \{x\}).
\end{aligned}$$

□

The following results describe the behaviour of join when applied to a fixed BST and when applied to a random BST, respectively. Notice that Theorem 7.3 is only a reformulation of Lemma 3.1 in Section 3.

Lemma 7.3 *Let U and V be two permutation of keys such that the keys in U are smaller than the keys in V. Then*

$$\mathsf{join}(\mathsf{bst}(U), \mathsf{bst}(V)) = \mathsf{bst}(\mathsf{shuffle}(\mathsf{equiv}(U), \mathsf{equiv}(V))).$$

Theorem 7.3 *Let $K_<$ and $K_>$ be two sets of keys such that the keys in $K_<$ are all smaller than the keys in $K_>$. Then*

$$\mathsf{join}(\mathsf{RBST}(K_<), \mathsf{RBST}(K_>)) = \mathsf{RBST}(K_< \cup K_>).$$

It only remains to describe the behaviour of delete, related with rm and equiv.

Lemma 7.4 *Let S be any permutation of keys and let x be any key. Then*

$$\mathsf{delete}(x, \mathsf{bst}(S)) = \mathsf{bst}(\mathsf{rm}(x, \mathsf{equiv}(S))).$$

Theorem 3.1 follows as an immediate consequence from the results above. We state it again, for the sake of completeness.

Theorem 7.4 *Let K be any set of keys and let x be any key. Then*

$$\mathsf{delete}(x, \mathsf{RBST}(K)) = \mathsf{RBST}(K \setminus \{x\}).$$

Notice that the theorem holds even if $x \notin K$, since in this case $K \setminus \{x\} = K$. It is also possible to prove that any deletion algorithm such that

1. it only modifies the subtree beneath the key x to be deleted;
2. its behaviour does not depend on the location of x within the BST;
3. its outcome is the result of rotating the key x left or right until it reaches a leaf; and
4. it preserves randomness

has to be equivalent to the one presented in this paper from the point of view of its probabilistic behaviour. In other words, any deletion algorithm that fullfils the conditions 1–4 above must satisfy Equations 16 and 17. The algorithm may use different mechanisms to make the random choices, but its probabilistic behaviour must be the same as that of our algorithm. We claim thus that the deletion algorithm that we present in this paper is, in a very strong sense, *the* deletion algorithm for random BSTs.

The notation presented in this section has allowed to state most of the results in this work in a very concise and rigorous manner. Our purpose has been to show that the notation is not difficult to understand. Its meaning is rather intuitive, while at the same time it does not sacrifice rigour; and it is not arduous to carry computations using it. As we have already seen, proofs become a mere issue of rewriting and applying induction where appropriate.

7.5. Self-adjusting strategies. Last but not least, we apply the tools of this section to give a proof of Theorem 5.3. The procedure self_adj performs a *successful* search of x in T and returns the tree after reorganization (presumably, it also returns the information associated to x, but we will not care about this associated information when describing the behaviour of self_adj). Let n denote the number of elements in the tree. Let $y = T \to \mathsf{key}$, $L = T \to \mathsf{left}$ and $R = T \to \mathsf{right}$. The equation for self_adj is

$$\mathsf{self\_adj}(x,T) = \alpha(n)\cdot \mathsf{insert\_at\_root}(x,T)$$
$$+ (1-\alpha(n))\cdot \Big[[\![x<y]\!] \; \overset{y}{\underset{\mathsf{self\_adj}(x,L)}{\diagup\diagdown}} \; R + [\![x>y]\!] \; \overset{y}{\underset{L}{\diagup\diagdown}} \; \mathsf{self\_adj}(x,R) + [\![x=y]\!]\, T \Big].$$

To modelize the sequence of successful accesses, let K be the set of keys, and for each key x in K, let $p(x) > 0$ denote the probability of access to x. Of course, we have

$$\sum_{x\in K} p(x) = 1.$$

An independent successful access to an item in K, according to p, is given by

$$a(K) = \sum_{x\in K} p(x)\cdot x.$$

Let π denote the asymptotic distribution that some self-adjusting strategy induces after an infinite number of successful accesses have been made. First, we should prove that any of these self-adjusting strategies produces a stationary probability distribution. Consider the (finite) Markov chain that describes the transition probabilities of this process (see [7], for instance). On the one hand, we have that all the states intercommunicate, that is, given two trees T_1 and T_2 built over the set K, the probability of ever reaching T_2 starting from T_1 is strictly greater than zero. On the other hand, there are no transitions with probability 1 (except when the tree contains only one element). Therefore, we have an irreducible, persistent, aperiodic Markov chain, irrespective of $\alpha(n)$. This implies the existence of such a unique stationary probability distribution.

The characteristic feature of any asymptotic distribution π is that it is the unique fixed point for its corresponding self_adj operation, when accesses are made according to $a(K)$,

$$\mathsf{self\_adj}(a(K), \pi) = \pi.$$

Clearly, π is a PF over the set $\mathcal{B}(K)$, and eventually depends on our choice of $\alpha(n)$, $\pi \equiv \pi(\alpha(n))$.

Theorem 5.3 states that π is the same for all possible $\alpha(n)$. Since it is true for $\alpha(n) = 1$ we have that π is the asymptotic distribution for the move-to-root strategy. Once we have shown that π is the same for all $\alpha(n)$, the other claims in Subsection 5.3 follow directly from the results of Allen and Munro concerning move-to-root [2].

Before we restate Theorem 5.3, we need some additional definitions.

Given a probability distribution p over the set of keys K, and a nonempty subset $A \subseteq K$, let

$$\Pi(A) = \sum_{y \in A} \frac{p(y)}{p(A)} \cdot \begin{array}{ccc} & (y) & \\ \Pi(A_{<y}) & & \Pi(A_{>y}) \end{array},$$

where $p(A) = \sum_{y \in A} p(y)$, $A_{<y} = \{z \in A \,|\, z < y\}$ and $A_{>y} = \{z \in A \,|\, z > y\}$. Notice that the definition of $\Pi(A)$ is independent of $\alpha(n)$. Furthermore, $\pi = \Pi(K)$ is the asymptotic distribution for move-to-root, since y is the root of the tree with probability $p(y)$, and the same applies recursively, after proper normalization, for the left and right subtrees. By definition, $\Pi(\emptyset) = \square$.

Moreover, for any nonempty $A \subseteq K$, let

$$a(A) = \sum_{x \in A} \frac{p(x)}{p(A)} \cdot x.$$

Thus, $a(A)$ defines the event "choose an element from A according to p (renormalized)".

Theorem 7.5 *For any $\alpha(n)$ such that $0 < \alpha(n) \leq 1$, and for any nonempty subset of keys $A \subseteq K$,*

$$\mathsf{self\_adj}(a(A), \Pi(A)) = \Pi(A).$$

Proof. We prove this result by induction on the size of A. If A contains only one key, then the claim of the theorem is trivially true. If A contains more than one element, we have

$$\mathsf{self\_adj}(a(A), \Pi(A)) = \alpha(n) \cdot \mathsf{insert\_at\_root}(a(A), \Pi(A))$$

$$+ (1 - \alpha(n)) \cdot \sum_{y \in A} \frac{p(y)}{p(A)} \cdot \left[\frac{p(A_{<y})}{p(A)} \cdot \begin{array}{ccc} & (y) & \\ \mathsf{self\_adj}(a(A_{<y}), \Pi(A_{<y})) & & \Pi(A_{>y}) \end{array} \right.$$

$$+ \frac{p(A_{>y})}{p(A)} \cdot \begin{array}{ccc} & (y) & \\ \Pi(A_{<y}) & & \mathsf{self\_adj}(a(A_{>y}), \Pi(A_{>y})) \end{array}$$

$$\left. + \frac{p(y)}{p(A)} \cdot \begin{array}{ccc} & (y) & \\ \Pi(A_{<y}) & & \Pi(A_{>y}) \end{array} \right].$$

Notice that, when y is the smallest element in A, the term $a(A_{<y})$ is not defined, since we cannot choose an element from the empty set. However, the factor $p(A_{<y})$ is zero. For the sake of completness, it is enough to assume that $0 \cdot$ "undefined" $= 0$. The same comment applies when y is the largest element in A.

Consider the equality above. On the one hand, we have that $\mathsf{insert\_at\_root}(a(A), \Pi(A)) = \Pi(A)$ (this is just the result of Allen and Munro). On the other, we can use the induction hypothesis to deduce that $\mathsf{self\_adj}(a(A_{<y}), \Pi(A_{<y})) = \Pi(A_{<y})$ and $\mathsf{self\_adj}(a(A_{>y}), \Pi(A_{>y})) = \Pi(A_{>y})$. Therefore,

$$\begin{aligned}\mathsf{self\_adj}(a(A), \Pi(A)) &= \alpha(n) \cdot \Pi(A) \\ &\quad + (1-\alpha(n)) \cdot \sum_{y \in A} \frac{p(y)}{p(A)} \cdot \begin{array}{c} y \\ \Pi(A_{<y}) \quad \Pi(A_{>y}) \end{array} \\ &= \Pi(A).\end{aligned}$$

□

8. CONCLUSIONS

We have presented randomized algorithms to insert and delete items into and from binary search trees that guarantee that, given as input a random binary search tree, its output is also a random binary search tree. Particularly important is the deletion algorithm, since it is the only one known that preserves the random BST model. Furthermore, and as far as we know, this is the first time that the randomness-preserving property of the deletion algorithm has been established.

Searches, insertions and deletions by key; splits and joins; searches and deletions by rank and computations of rank can all be performed in $\Theta(\log n)$ expected time, independently of the input distribution, where n is the size of the involved trees. All these operations should be fast in practice (if generating random numbers is not very expensive), since they visit the same nodes as their standard deterministic counterparts, and they can be implemented in a top-down fashion. We have shown that these results can be achieved using only structural information, the subtree sizes. As a consequence, an efficient implementation of rank operations follows at no additional cost.

Also, set operations (unions, intersections and differences) yielding random BSTs if their input is a pair of random BSTs, can also be implemented in $\Theta(n)$ expected time using similar ideas.

Another question that we have considered in this paper is that of self-adjusting strategies. We have been able to prove that there exists a general family of self-adjusting strategies that behave identically in the asymptotic state. The family includes the well known move-to-root heuristic. However, the different strategies in this family must exhibit different rates of convergence to the asymptotic distribution; we still lack of a quantitave analysis of this question and thus leave open this problem. Other open problem (probably very difficult) concerns the robustness against malicious adversaries. Since all the self-adjusting strategies that we have considered are randomized, the only exception being move-to-root, it may be well that one or more of these strategies were competitive against an oblivious adversary.

Other further lines of research that we are pursuing include the application of the techniques in this paper to other kind of search trees, like m-ary trees, quadtrees, etc.

The randomized treaps of Aragon and Seidel satisfy all the Theorems and Lemmas of sections 2 and 3. In particular, their algorithms always produce random binary search trees. However, little use of this fact was made by the authors when they analyzed their algorithms. Their recently published work on randomized treaps [25] also mentions that the random priorities for randomized treaps can be simulated using the sizes of subtrees, pointing out thus the main idea of the present work, but the idea is not further developed there. From our point of view, randomizing through the sizes of subtrees is more advantageous than through random priorities, since the mechanism that allows the randomization process is useful information in the former case, while is not in the later.

ACKNOWLEDGEMENTS

We wish to thank Ricardo Baeza-Yates, Rafael Casas, Josep Díaz, Philippe Flajolet and Robert Sedgewick, as well as the anonymous referees who made many useful suggestions and comments on early versions of this work.

REFERENCES

[1] G. Adel'son-Vel'skii and E. Landis. An algorithm for the organization of information. *Dokladi Akademia Nauk SSSR*, 146(2):263–266, 1962. English translation in Soviet Math. Doklay 3 ,1962, 1259-1263.

[2] B. Allen and J. Munro. Self-organizing search trees. *J. ACM*, 25(4):526–535, 1978.

[3] C. R. Aragon and R. G. Seidel. Randomized search trees. In *Proc. of the 30th Annual IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 540–545, 1989.

[4] R. Baeza-Yates. A trivial algorithm whose analysis isn't: A continuation. Technical Report CS-86-67, Dept. of Computer Science,University of Waterloo, Dec. 1986.

[5] J. Culberson. The effect of updates in binary search trees. In *Proc. of the 17th ACM Symposium on the Theory of Computation (STOC)*, pages 205–212, 1985.

[6] J. Eppinger. An empirical study of insertion and deletion in binary search trees. *Comm. ACM*, 26(9):663–669, 1983.

[7] W. Feller. *An Introduction to Probability Theory and its Applications, vol. I.* J. Wiley, New York, 1968.

[8] G. Gonnet, H. Olivie, and D. Wood. Height-ratio-balanced trees. *Computer Journal*, 26(2):106–108, 1983.

[9] G. H. Gonnet and R. Baeza-Yates. *Handbook of Algorithms and Data Structures - In Pascal and C.* Addison-Wesley, 2nd edition, 1991.

[10] L. Guibas and R. Sedgewick. A dichromatic framework for balanced trees. In *Proc. of the 19th Annual IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 8–21, Oct. 1978.

[11] T. N. Hibbard. Some combinatorial properties of certain trees with applications to searching and sorting. *J. ACM*, 9(1):13–18, 1962.

[12] A. Jonassen and D. Knuth. A trivial algorithm whose analysis isn't. *JCSS*, 16(3):301–322, 1978.

[13] G. D. Knott. *Deletions in Binary Storage Trees.* PhD thesis, Computer Science Dept., Stanford University, 1975.

[14] D. Knuth. Deletions that preserve randomness. *IEEE Trans. Software Engineering*, 3:351–359, 1977.

[15] D. E. Knuth. *The Art of Computer Programming: Sorting and Searching*, volume 3. Addison-Wesley, 1973.

[16] D. C. Kozen. *The Design and Analysis of Algorithms.* Springer-Verlag, 1992.

[17] H. M. Mahmoud. *Evolution of Random Search Trees.* Wiley Interscience, 1992.

[18] C. Martínez and X. Messeguer. Deletion algorithms for binary search trees. Technical Report LSI-90-39, LSI-UPC, 1990.

[19] J. Nievergelt and E. Reingold. Binary search trees of bounded balance. *SIAM J. Comp.*, 2(1):33–43, 1973.

[20] W. Pugh. Skip lists: A probabilistic alternative to balanced trees. *Comm. ACM*, 33(6):668–676, 1990.

[21] P. Raghavan and R. Motwani. *Randomized Algorithms*. Cambridge University Press, 1995.

[22] S. Roura and C. Martínez. Randomization of search trees by subtree size. In J. Díaz and M. Serna, editors, *Proc. of the 4th European Symposium on Algorithms (ESA)*, volume 1136 of *LNCS*, pages 91–106. Springer, 1996.

[23] R. Sedgewick. *Algorithms*. Addison-Wesley, 2nd edition, 1988.

[24] R. Sedgewick and P. Flajolet. *An Introduction to the Analysis of Algorithms*. Addison-Wesley, 1996.

[25] R. Seidel and C. Aragon. Randomized search trees. *Algorithmica*, 16:464–497, 1996.

[26] C. J. Stephenson. A method for constructing binary search trees by making insertions at the root. *International Journal of Computer and Information Sciences*, 9(1), 1980.

[27] J. S. Vitter and P. Flajolet. Average-case analysis of algorithms and data structures. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, chapter 9. North-Holland, 1990.

APPENDIX A. NON-RECURSIVE IMPLEMENTATION OF THE DELETION ALGORITHM

This appendix contains the non-recursive implementation of the deletion algorithm delete(x, T) as well as the implementation of the auxiliary procedure join(A, B). This implementation also considers the management of subtree sizes (see Section 6).

The representation of trees has been slightly changed with respect to the one considered along the main text. A tree T is now represented by a pointer to a record with two fields: $T \to$ size is the total number of internal nodes in T, and $T \to$ root is a pointer to the root node. Each internal node stores a key, pointers to the left and right subtrees, the size of one of its subtrees, and an orientation bit which may be either LEFT or RIGHT thus indicating which is the subtree whose size is mantained by the node.

Both delete and join make use of a function flip_orientation which is convenient for bookkeeping.

Algorithm 9 Flip_orientation

```
/*   LEFT = 0, RIGHT = -1   */
/* Given the total size n of the tree rooted at p, flip_orientation
   changes the orientation bit of p to indicate the opposite subtree,
   assigns the old value of p→size to n,
   and p→size is replaced by the size of the other subtree */

void flip_orientation(int *n, struct node *p) {
  int aux;

  aux = *n - 1 - p→size;
  *n = p→size;
  p→size = aux;
  p→orientation ^= -1;
}
```

Algorithm 10 Nonrecursive deletion

```
void delete(int x, bst T) {
  struct node *p, *parent, *aux;
  int n;

  n = T→size;
  p = T→root; parent = □;
  while (p ≠ □) {
      if (x == p→key) {
          aux = join(p, n);
          T→size--;
          if (parent == □) T→root = aux;
          else if (parent→key > x) parent→left = aux;
               else parent→right = aux;
          break;
      }
      parent = p;
      if (x < p→key) {
          if (p→orientation == LEFT) flip_orientation(&n, p);
          p = p→left;
      }
      else {
          if (p→orientation == RIGHT) flip_orientation(&n, p);
          p = p→right;
      }
  }
}
```

Algorithm 11 Nonrecursive join

```
/* Given a pointer p to the root of a tree and the total size gs of this tree,
   join performs the joining of the subtrees of p */

struct node *join(struct node *p, int gs) {
  struct node *l, *r, *result;
  (struct node*) *parent;
  int m, n, u, total;

  if (p→orientation == LEFT) { m = p→size; n = gs - 1 - m; }
  else { n = p→size; m = gs - 1 - n; }
  total = m + n;

  if (total == 0) return □;

  parent = &result;
  l = p→left; r = p→right;

  while (total > 0) {
      u = random(1, total);
      if (u <= m) {
          *parent = l; parent = &(l→right);
          if (l→orientation == RIGHT) flip_orientation(&m, l);
          l = l→right;
      }
      else {
          *parent = r; parent = &(r→left);
          if (r→orientation == LEFT) flip_orientation(&n, r);
          r = r→left;
      }
      total = m + n;
  }
  free_node(p);
  return result;
}
```