1400191463
Copia 1

Randomization of search trees by subtree size

Conrado Martínez
Salvador Roura

Report LSI–95–51–R

UPC
Facultat d'Informàtica
de Barcelona - Biblioteca
14 NOV. 1995

Randomization of Search Trees by Subtree Size*

C. MARTÍNEZ AND SALVADOR ROURA[†]

November 8, 1995

Abstract

In this paper we present probabilistic algorithms over random binary search trees such that: a) the insertion of a set of keys in any fixed order into an initially empty tree produces always a random tree; b) the deletion of any key of a random tree results in a random tree; c) the random choices made by the algorithms are based upon the sizes of the subtrees of the random tree, an information that can be used for rank searches, for instance; and d) the cost, measured as the number of visited nodes, of any elemental operation is the same as the cost of the standard deterministic version, with less than two expected rotation-like operations per update.

1. INTRODUCTION

Given a binary search tree (BST, for short), common operations are the search of a given key in the tree 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 symmetric or inorder traversal of any binary search tree yields a list of the items in the tree in ascending order of keys. For the unbalanced version of the BSTs, the implementation of searches and updates is very simple and elegant, and their cost is always linearly bounded by the depth of the tree.

For *random search trees*, the expected performance of a search, whether successful or not, and that of update operations is $\mathcal{O}(\log n)$ [4, 5], with small hidden constant factors involved. Random search trees are those built using only random insertions. By a random insertion, we mean that there is the same probability for the j-th key to fall into any of the j intervals defined by the $j-1$ keys already in the tree.

If this assumption fails (for instance, ordered sequences of keys are frequently inserted) then the performance of the operations can dramatically degrade and become linear. Furthermore, the random search tree model does not deal with deletions, and neither Hibbard's deletion algorithm [2] nor its multiple variants preserve randomness, a surprising fact that was first noticed by Knott [3].

A recent approach to avoid these problems is the use of randomization techniques, where they are exploited to guarantee that algorithms achieve their expected performance. These techniques were used by Aragon and Seidel [1] for their *randomized search trees* and by Pugh [7] in his definition

*This research was supported by the ESPRIT BRA Program of the EC under contract no. 7141, project ALCOM II. The second author was also supported by a grant from CIRIT (Comissió Interdepartamental de Recerca i Innovació Tecnològica).

[†]Departament de Llenguatges i Sistemes Informàtics. Univ. Politècnica de Catalunya. Pau Gargallo, 5. E-08028 Barcelona, Spain. E-mail: {conrado,roura}@goliat.upc.es

of *skip lists*. The good expected performance of skip lists and randomized search trees does not depend on any probabilistic assumption about the sequence of operations to be performed. Unless the random choices made by these algorithms were known, a sequence of operations that forced some designated behavior could not be constructed.

In this paper, we consider probabilistic algorithms to insert and delete in BSTs and guarantee the logarithmic expected performance of any elemental operation. This is possible since both insertions and deletions always produce random search trees, irrespective of the order in which the keys were inserted and/or deleted, the actual pattern of insertions and deletions, etc. An important difference between our algorithms and those for randomized search trees or skip lists, is that the random choices made by our algorithms are based upon structural information —the size of the subtree rooted at each node of the BST— whilst the other algorithms use information that may be labelled as "external" or "extraneous" (random priorities in the case of randomized search trees, random levels in the case of skip lists). The information about the size of subtrees can be used for other purposes as well, for instance, to do searches and deletions by rank or efficiently computing the rank of a given item.

The paper is organized as follows. In Section 2 we review some necessary definitions and introduce the notation for the rest of the paper. In Section 3, the insertion and deletion algorithms are described and their main properties stated. We present the analysis of their performance in Section 4 and show how we can profit from known results about random search trees. In Section 5 we discuss efficient strategies for the dynamic management of subtree sizes, space requirements, etc. We conclude in Section 6 with some remarks and future research lines.

2. BASIC DEFINITIONS AND NOTATION

Given a finite set of keys K, we shall denote $\mathcal{B}(K)$ the set of all binary search trees that contain all the keys in K. For simplicity, we will assume that $K \subset \mathbb{N}$. The empty tree is denoted by $\Box$. Similarly, we denote $\mathcal{P}(K)$ the set of all permutations —sequences without repetition— of the keys in K. The empty sequence is denoted by λ and $U \,|\, V$ denotes the concatenation of the sequences U and V.

The following equations relate sequences in $\mathcal{P}(K)$ and binary search trees in $\mathcal{B}(K)$. Given a sequence S, $\mathrm{bst}(S)$ is the BST resulting after the insertion of the keys in S into an initially empty tree.

$$\mathrm{bst}(\lambda) = \Box, \qquad \mathrm{bst}(x \,|\, S) = \begin{tikzcd}[column sep=small] & \textcircled{x} \arrow[dl, dash] \arrow[dr, dash] & \\ \mathrm{bst}(\mathrm{sep}_<(x,S)) & & \mathrm{bst}(\mathrm{sep}_>(x,S)) \end{tikzcd},$$

where the function $\mathrm{sep}_<(x, S)$ returns the subsequence of elements in S smaller than x, and $\mathrm{sep}_>(x, S)$ returns the subsequence of elements in S larger than x.

While the behavior of deterministic algorithms can be neatly described by means of algebraic equations, this is not the case for probabilistic algorithms. We now present an algebraic-like notation that allows a concise and rigorous description and further reasoning about these algorithms, following the ideas introduced in [6]. The key idea is to consider any probabilistic algorithm F as a function from the input set A to the *set of probability functions* (or PFs, for short) over B. Hence, $F(x)$ denotes the PF over the output set B, when x is the input of the algorithm F; and for any $y \in B$, $[F(x)](y)$ is the probability that, on input x, the algorithm F produces output y.

We will use the convention that each element y in B also denotes a PF over B, namely, $y(b) = 1$ if $b = y$ and 0, otherwise. Typically, the set of possible outputs of $F(x)$ is finite and then $F(x)$ may be expressed as a *linear combination* of PFs,

$$F(x) = \alpha_1 y_1 + \cdots + \alpha_m y_m = \sum_{1 \le i \le m} \alpha_i y_i,$$

where $\{y_1, \ldots, y_m\}$ is the set of possible outputs of F given input x, and

$$\alpha_i = \Pr\{\text{on input } x,\ F \text{ outputs } y_i\} = [F(x)](y_i).$$

Notice that the convention that any element of a set denotes also a PF over that set, makes the notation useful for the description of both deterministic and probabilistic algorithms. 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.

Finally, we give a precise meaning to the (sequential) composition of probabilistic algorithms. The problem reduces to the following question: if the input of an algorithm F is chosen according to some probability function g over the set of inputs A, which is the probability that a given element y is the output of F? By $F(g)$, we will denote the probability function over the set of outputs 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.

$$F(g) = \sum_{x \in A} g(x) F(x). \tag{1}$$

Recall that even a single element a from A may be considered as a PF over A, and then the definition above makes sense even when $g = a$.

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

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

Random search trees can be defined in the following elegant way:

$$\mathit{Random\_BST}(K) = \mathrm{bst}(\mathit{Random\_Perm}(K)).$$

Another equivalent characterization of a random search tree is given by the following definition [4, 5]: the empty tree is a random search tree; and a non-empty tree T is random if both its left subtree T_{left} and its right subtree T_{right} are random, and

$$\Pr\{\text{size}(T_{\text{left}}) = i \,|\, \text{size}(T) = n\} = \frac{1}{n}, \qquad i = 0, \ldots, n-1, \quad n > 0. \tag{2}$$

This last equation simply says that any of the n keys in T has the same probability, namely $1/n$, of being the root of T.

3. THE ALGORITHMS

The insertion algorithm is suggested by the last shown characterization of a random BST. Let T be a tree of size n and x be a key not in T. To insert x in T, "place" x at its root with a probability equal to $1/(n+1)$ (notice that the new BST will have $n+1$ keys) and otherwise recursively insert x in the left or right subtree of T, depending on the relation of x with the key in the root. Using the notation introduced in last section, the algebraic equations describing the behavior of *insert* are

$$
\begin{aligned}
\text{insert}(x, \square) &= \begin{smallmatrix} & \textcircled{x} & \\ \diagup & & \diagdown \\ \square & & \square \end{smallmatrix}, \\
\text{insert}\left(x, \begin{smallmatrix} & \textcircled{y} & \\ \diagup & & \diagdown \\ L & & R \end{smallmatrix}\right) &= \frac{1}{n+1} \cdot \text{place\_at\_root}\left(x, \begin{smallmatrix} & \textcircled{y} & \\ \diagup & & \diagdown \\ L & & R \end{smallmatrix}\right) \\
&+ \frac{n}{n+1} \cdot \left([x<y] \cdot \begin{smallmatrix} & \textcircled{y} & \\ \diagup & & \diagdown \\ \text{insert}(x,L) & & R \end{smallmatrix} + [x>y] \cdot \begin{smallmatrix} & \textcircled{y} & \\ \diagup & & \diagdown \\ L & & \text{insert}(x,R) \end{smallmatrix} \right),
\end{aligned} \tag{3}
$$

assuming that it is never inserted a key already in the tree.

The question now is how to place x at the root of T, that is, how to build a new T' that contains x and the keys that were present in T such that $\text{root}(T') = x$. This problem was solved by Stephenson [8, 9] as early as 1976. The function *place_at_root* returns a tree whose root is x, its left subtree is $\text{split}_<(x,T)$ and its right subtree is $\text{split}_>(x,T)$; $\text{split}_<$ and $\text{split}_>$ are functions that given 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 function $\textit{split}_<$ can be written in algebraic form as follows:

$$
\begin{aligned}
\text{split}_<(x, \square) &= \square, \\
\text{split}_<\left(x, \begin{smallmatrix} & \textcircled{y} & \\ \diagup & & \diagdown \\ L & & R \end{smallmatrix}\right) &= [x<y] \cdot \text{split}_<(x,L) + [x>y] \cdot \begin{smallmatrix} & \textcircled{y} & \\ \diagup & & \diagdown \\ L & & \text{split}_<(x,R) \end{smallmatrix}.
\end{aligned} \tag{4}
$$

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

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

It can be shown that $\text{split}_<(x,T)$ and $\text{split}_>(x,T)$ can be computed at the same time with a total cost (measured as the number of keys compared against x) equal to the the cost of the standard insertion of x in T. Note that both *place_at_root* and *split* are deterministic functions, i.e. for any input they produce only one possible output. The following lemma describes the result of *split* when applied to a fixed BST.

Lemma 3.1 *Let S be any permutation of keys and let x be any key not in S. Then*

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

From this lemma we can describe the behavior of *split* when applied to a random BST.

Theorem 3.1 *Let K be any set of keys and let x be any key such that $x \notin K$. Let $K_<$ and $K_>$ denote the set with the keys in K less than x and the set with the keys in K greater than x, respectively. Then*

$$\textit{split}(x, \textit{Random\_BST}(K)) = \Big[\ \textit{Random\_BST}(K_<)\ ,\ \textit{Random\_BST}(K_>)\ \Big].$$

Notice that the theorem states that splitting a random BST produces a pair of *independent* random BSTs. Lemma 3.1 relates *split* with *sep* using *bst*, the "mapping" between trees and permutations of keys. Our next objective is to relate *insert* with some functions over permutations. To this end we introduce two new functions, *shuffle* and *equiv*.

Let K_1 and K_2 be two disjoint sets with n and m 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, i.e. $\mathcal{S}(U,V)$ is the set of all $Y = y_1 \,|\, \ldots \,|\, y_{m+n} \in \mathcal{P}(K_1 \cup K_2)$ such that for all $y_{i_1} = u_{j_1}$ and $y_{i_2} = u_{j_2}$, $i_1 < i_2$ if and only if $j_1 < j_2$ (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 $\binom{m+n}{m}$. We define *shuffle* as a function such that given as input U and V, returns a randomly choosen element from $\mathcal{S}(U,V)$. For instance,

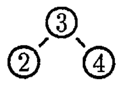
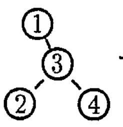
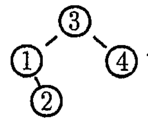
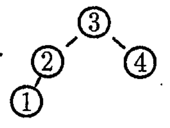
$$\text{shuffle}(21, ba) = \frac{1}{6}\cdot 21ba + \frac{1}{6}\cdot 2b1a + \frac{1}{6}\cdot 2ba1 + \frac{1}{6}\cdot b21a + \frac{1}{6}\cdot b2a1 + \frac{1}{6}\cdot ba21.$$

Let $S \in \mathcal{P}(K)$. We define *equiv* as a function such that given input S returns a randomly chosen element from the set $\{E\}$ of permutations such that $\text{bst}(E) = \text{bst}(S)$. For example, $\text{equiv}(3124) = 1/3\cdot 3124 + 1/3\cdot 3142 + 1/3\cdot 3412$, since $\text{bst}(3124) = \text{bst}(3142) = \text{bst}(3412)$ and no other permutation of the keys $\{1,2,3,4\}$ produces the same tree. Next lemma describes the behavior of *insert*.

Lemma 3.2 *Let S be any permutation of keys and let x be any key not in S. Then*

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

For instance,

insert(1, bst(342)) = insert(1, ) = $\frac{1}{4}\cdot$  $+\frac{3}{8}\cdot$  $+\frac{3}{8}\cdot$  .

On the other hand,
$\text{bst}(\text{shuffle}(1, \text{equiv}(342))) = \text{bst}(\text{shuffle}(1, \frac{1}{2}\cdot 342 + \frac{1}{2}\cdot 324)) =$
$= \text{bst}(\frac{1}{8}\cdot 1342 + \frac{1}{8}\cdot 3142 + \frac{1}{8}\cdot 3412 + \frac{1}{8}\cdot 3421 + \frac{1}{8}\cdot 1324 + \frac{1}{8}\cdot 3124 + \frac{1}{8}\cdot 3214 + \frac{1}{8}\cdot 3241)$,
which gives the same result as insert(1, bst(342)), since {1342, 1324} produce the first tree, {3142, 3412, 3124} produce the second one and {3421, 3214, 3241} produce the third.

The lemma above relating *insert* with *shuffle* and *equiv* is the basis for the next important theorem, that describes the behavior of *insert* when applied to a random BST.

Theorem 3.2 *Let K be any set of keys and let x be any key such that $x \notin K$. Then*

$$\textit{insert}(x, \textit{Random\_BST}(K)) = \textit{Random\_BST}(K \cup \{x\}).$$

Proof. Notice that taking $S \in \mathcal{P}(K)$ at random and then chosing any permutation equivalent to S, is the same as taking a permutation in $\mathcal{P}(K)$ at random. Notice also that the result of shuffling x into a random permutation of the keys in K gives a random permutation of the keys in $K \cup \{x\}$.

Then,

$$
\begin{aligned}
\text{insert}(x, \textit{Random\_BST}(K)) &= \text{insert}(x, \text{bst}(\textit{Random\_Perm}(K))) \\
\{\text{by Lemma } 3.2\} &= \text{bst}(\text{shuffle}(x, \text{equiv}(\textit{Random\_Perm}(K)))) \\
&= \text{bst}(\text{shuffle}(x, \textit{Random\_Perm}(K))) \\
&= \text{bst}(\textit{Random\_Perm}(K \cup \{x\})) \\
&= \textit{Random\_BST}(K \cup \{x\})
\end{aligned}
$$

□

As an immediate consequence of the theorem above, our next theorem follows:

Theorem 3.3 *Let $K = \{x_1, \ldots, x_n\}$ be any set of keys, where $n \geq 0$. Let $x_{i_1}, \ldots, x_{i_n}$ be* any fixed *permutation of the keys in K. Then*

$$\textit{insert}(x_{i_n}, \textit{insert}(x_{i_{n-1}}, \ldots, \textit{insert}(x_{i_1}, \square)\ldots)) = \textit{Random\_BST}(K).$$

Next we describe the deletion algorithm. To delete any given key x from any given random BST first find it, using the standard search algorithm until a leaf or x is found. In the first case the key to be deleted is not in the tree, so nothing must be done. In the second case, only the subtree whose root is x will be modified.

Let T be that subtree. Let L and R denote the left son of T and the right son of T, respectively, and let $K_<$ be the set of keys in T less than x and $K_>$ be the set of keys in T greater than x. First, we delete the node where x is (the root of T). Then, from the pair of BSTs (L, R) we build a new BST $T' = \text{join}(L, R)$ containing the keys in the set $K_< \cup K_>$ and place T' where x has been deleted. The algebraic form of *delete* is:

$$
\begin{aligned}
\text{delete}(x, \square) &= \square \\
\text{delete}\left(x, \begin{array}{c} \textcircled{y} \\ \diagup \quad \diagdown \\ L \qquad R \end{array}\right) &= [x < y] \cdot \begin{array}{c} \textcircled{y} \\ \diagup \quad \diagdown \\ \text{delete}(x, L) \qquad R \end{array} + [x > y] \cdot \begin{array}{c} \textcircled{y} \\ \diagup \quad \diagdown \\ L \qquad \text{delete}(x, R) \end{array} \\
&+ [x = y] \cdot \text{join}(L, R).
\end{aligned}
\tag{5}
$$

Assume that L and R are trees of size $m > 0$ and $n > 0$, respectively. Let L_{left}, L_{right}, R_{left} and R_{right} denote the left son of L, the right son of L, the left son of R and the right son of R, respectively. The probabilistic behavior of *join* can be described as follows.

$$
\text{join}(L, R) = \frac{m}{m+n} \cdot \begin{array}{c} \text{root}(L) \\ \diagup \quad \diagdown \\ L_{\text{left}} \qquad \text{join}(L_{\text{right}}, R) \end{array} + \frac{n}{m+n} \cdot \begin{array}{c} \text{root}(R) \\ \diagup \quad \diagdown \\ \text{join}(L, R_{\text{left}}) \qquad R_{\text{right}} \end{array} .
\tag{6}
$$

For the basic cases, we define $\text{join}(\square, \square) = \square$, $\text{join}(L, \square) = L$, and $\text{join}(\square, R) = R$.

There is an informal justification for the probabilities we have used, based on the Eq. 2 for random BSTs. Let x be any key in L. We know that x has a probability $\frac{1}{m}$ to be the root of L. Then, after joining L and R the probabily for x to be the root of T' will be $\frac{1}{m} \cdot \frac{m}{m+n} = \frac{1}{m+n}$, which is the probability that any key in T' has of being the root. The same reasoning applies to any key in R.

The following lemma and theorem describe the behavior of *join* when applied to a fixed BST and when applied to a random BST, respectively.

Lemma 3.3 *Let U and V be two permutation of keys such that any of the keys in U is smaller than any of the keys in V. Then*

$$join(bst(U), bst(V)) = bst(shuffle(equiv(U), equiv(V))).$$

Theorem 3.4 *Let $K_<$ and $K_>$ be two sets of keys such that the keys in $K_<$ are less than the keys in $K_>$. Then*

$$join(Random\_BST(K_<), Random\_BST(K_>)) = Random\_BST(K_< \cup K_>).$$

Notice that there are many different deterministic strategies to join BSTs (see [9], for instance), but none of them can satisfy the theorem above. Take the particular case $K_< = \{1\}, K_> = \{3\}$, for instance. The pair $(\mathrm{bst}(1), \mathrm{bst}(3))$ is the only possible input for *join*, whilst the output must be randomly chosen from $\{\mathrm{bst}(13), \mathrm{bst}(31)\}$.

It only remains to describe the behavior of *delete*. We first define a new function, *rm*, such that given any permutation of keys S and any key x, it returns the permutation of keys that results after removing x from S. For instance, $\mathrm{rm}(3, 2315) = 215$, $\mathrm{rm}(4, 2315) = 2315$. Now we are ready to relate *delete* with *rm* and *equiv*.

Lemma 3.4 *Let S be any permutation of keys and let x be any key. Then*

$$delete(x, bst(S)) = bst(rm(x, equiv(S))).$$

Theorem 3.5 *Let K be any set of keys and let x be any key. Then*

$$delete(x, Random\_BST(K)) = Random\_BST(K - \{x\}).$$

Notice that the theorem holds even if $x \notin K$, since in this case $K - \{x\} = K$.

We end up this section with a few comments on the proofs of the lemmas and theorems in this section. Both *shuffle* and *equiv* can be defined using the notation in Section 2, and then most proofs can be done by induction on the size of the input tree(s) and applying simple algebraic manipulations. Furthermore, most of them follow a common pattern and reduce to pure algebraic manipulation. They are covered in detail in the full version of this extended abstract.

4. PERFORMANCE ANALYSIS

The analysis of the performance of the various algorithms is quite easy, since both insertions and deletions guarantee the randomness of their results. In fact, we only need three results about random search trees of size n: the depth of the i-th internal node, the depth of the i-th leaf, and the lenght of the right and left *spines* of the subtree whose root is the i-th node. We will denote them $\mathcal{D}_n^{(i)}$, $\mathcal{L}_n^{(i)}$ and $\mathcal{S}_n^{(i)}$, respectively. 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.

It has been long known (see for instance [4]) that

$$\begin{aligned}
\mathrm{E}(\mathcal{D}_n^{(i)}) &= H_i + H_{n+1-i} - 2, \qquad i = 1, \ldots, n. \\
\mathrm{E}(\mathcal{L}_n^{(i)}) &= H_{i-1} + H_{n+1-i}, \qquad i = 1, \ldots, n+1. \\
\mathrm{E}(\mathcal{S}_n^{(i)}) &= \mathrm{E}(\mathcal{L}_n^{(i)} + \mathcal{L}_n^{(i+1)} - 2(\mathcal{D}_n^{(i)} + 1)) = 2 - \frac{1}{i} - \frac{1}{n+1-i}, \qquad i = 1, \ldots, n.
\end{aligned}$$

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

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

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

Now, let us consider the cost (measured as the number of visited nodes) of an insertion in the i-th interval ($1 \leq i \leq n+1$) of a tree of size n. We may divide this cost into two contributions: the cost of the descent until the new item reaches in its final position, $R_n^{(i)}$, plus the cost of restructuring the tree beneath or cost of the "placement at the root", $P_n^{(i)}$. Consider the tree after the insertion. Let T be the subtree whose root is the inserted key. The length of the path from the root of the resulting tree to T gives us $R_n^{(i)}$, whilst the lenght of the right and left spines of T gives us $P_n^{(i)}$. Since the tree produced by the insertion is random, we conclude that

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

Notice that $\mathrm{E}(R_n^{(i)} + P_n^{(i)}) = H_{i-1} + H_{n+1-i} + 1 = \mathrm{E}(\mathcal{L}_n^{(i)} + 1)$, which is the expected cost of the standard insertion in the i-th interval of a tree of size n.

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 separate 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

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

5. IMPLEMENTATION ISSUES

From the recursive Equations 3, 4, 5 and 6, it is straightforward to obtain a recursive implementation of the insertion and deletion operation. An additional little effort yields efficient non-recursive implementations of these operations, using only a constant amount of auxiliary space and working in a pure top-down fashion.

One of the questions that we have not dealt with in previous sections is the *insertion of repeated elements*. Making a search before performing the insertion is clearly inadequate, if the insertion of repeated keys is not frequent. We have developed a variant of the insertion algorithm, called *insertion with push_down* to cope with this problem. It is described in the full version of this paper. This variant of the insertion works also in a top-down fashion, and it can be shown that using it, Theorem 3.2 is valid even when $x \in K$.

Let us now consider the complexity of our algorithms from the point of view of the number of random bits needed per operation. In the case of deletions, the expected size of the subtree rooted at the node to be deleted is constant, so the expected number of random bits is also constant. For insertions, a random number must be generated for each node visited before the place at root is done. If the currently visited node x 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 place at root the element to be inserted, otherwise the insertion continues either on the left or right subtree of x. Recall that the expected number of nodes visited before we "place at root" the new key is $\mathcal{O}(\log n)$, for a BST of size n. 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 random bits generated per node is $\mathcal{O}(1)$ —most of the times the comparison fails and the insertion continues at the appropriate subtree—. The total expected number of random bits per insertion is thus $\mathcal{O}(\log n)$. Further refinements may reduce this to $\mathcal{O}(1)$; the reduction is achieved at the cost of performing several arithmetic operations for each visited node during the insertion.

Up to now, we have not considered the problem of managing the sizes of subtrees. Obviously, they shouldn't be computed on-the-fly; hence, each node of the tree has to store information about the size of subtree rooted at that particular node.

If the size of each subtree is stored at its root then we would face the problem of updating this information for all nodes in the path followed during the operation. The problem gets worse 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).

Probably, the best 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 the 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. 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 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. Similar rules can be used for the implementation of split and join.

We should emphasize that this 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.

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

6. Conclusions

We have presented probabilistic algorithms that guarantee that given a random search tree its output is also a random search tree. Searches by key, insertions, deletions, splits, joins, searches and deletions by rank can be performed in $\mathcal{O}(\log n)$ expected time, 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. Although not mentioned in this paper, set operations (unions, intersections and differences) yielding random search trees if their input is a pair of random trees, can be implemented in $\mathcal{O}(n)$ expected time by similar algorithms.

It can also be shown that the randomized search trees of Aragon and Seidel [1] satisfy all the Theorems and Lemmas of Section 3. In particular, their algorithms produce always random search trees, although the authors didn't mention it explicitly. In this paper, we have shown that the same results can be achieved using only structural information, namely, subtree sizes.

In Section 3 we have mentioned that no deterministic join algorithm can preserve randomness. This observation can be generalized to the stronger claim that no "reasonable" deterministic deletion algorithm can preserve randomness. By reasonable we mean that the deletion of a key only affects the subtree whose root is the deleted key.

We are now investigating the application of the techniques in this paper to other kind of search trees, like m-ary trees, quadtrees, etc. We are also exploring a family of self-adjusting strategies for binary search trees, based upon the "insertion with push_down" algorithm.

REFERENCES

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

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

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

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

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

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

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

[8] C. J. Stephenson. A method for constructing binary search trees by making insertions at the root. Technical Report RC 6298, IBM Thomas J. Watson Res. Center, Yorktown Heights, N.Y., 1976.

[9] J. Vuillemin. A unifying look at data structures. *Comm. ACM*, 23(4):229–239, 1980.