Improved Bounds for Finger Search on a RAM[⋆]

Alexis Kaporis, Christos Makris, Spyros Sioutas, Athanasios Tsakalidis, Kostas Tsichlas, and Christos Zaroliagis

Computer Technology Institute, P.O. Box 1122, 26110 Patras, Greece
and Department of Computer Engineering and Informatics,
University of Patras, 26500 Patras, Greece
{kaporis,makri,sioutas,tsak,tsihlas,zaro}@ceid.upatras.gr

Abstract. We present a new finger search tree with $O(1)$ worst-case update time and $O(\log \log d)$ expected search time with high probability in the Random Access Machine (RAM) model of computation for a large class of input distributions. The parameter d represents the number of elements (distance) between the search element and an element pointed to by a finger, in a finger search tree that stores n elements. For the need of the analysis we model the updates by a "balls and bins" combinatorial game that is interesting in its own right as it involves insertions and deletions of balls according to an unknown distribution.

1 Introduction

Search trees and in particular finger search trees are fundamental data structures that have been extensively studied and used, and encompass a vast number of applications (see e.g., [12]). A *finger search tree* is a leaf-oriented search tree storing n elements, in which the search procedure can start from an arbitrary element pointed to by a *finger* f (for simplicity, we shall not distinguish throughout the paper between f and the element pointed to by f). The goal is: (i) to find another element x stored in the tree in a time complexity that is a function of the "distance" (number of leaves) d between f and x; and (ii) to update the data structure after the deletion of f or after the insertion of a new element next to f. Several results for finger search trees have been achieved on the Pointer Machine (PM) and the Random Access Machine (RAM) models of computation.

In this paper we concentrate on the RAM model. W.r.t. worst-case complexity, finger search trees with $O(1)$ update time and $O(\log d)$ search time have already been devised by Dietz and Raman [5]. Recently, Andersson and Thorup [2] improved the search time to $O(\sqrt{\log d / \log \log d})$, which is optimal since there exists a matching lower bound for searching on a RAM. Hence, there is no room for improvement w.r.t. the worst-case complexity.

⋆ This work was partially supported by the IST Programme of EU under contract no. IST-1999-14186 (ALCOM-FT), by the Human Potential Programme of EU under contract no. HPRN-CT-1999-00104 (AMORE), and by the Carathéodory project of the University of Patras.

However, simpler data structures and/or improvements regarding the search complexities can be obtained if randomization is allowed, or if certain classes of input distributions are considered. A notorious example for the latter is the method of *interpolation search*, first suggested by Peterson [16], which for random data generated according to the *uniform* distribution achieves $\Theta(\log\log n)$ expected search time. This was shown in [7,15,19]. Willard in [17] showed that this time bound holds for an extended class of distributions, called *regular*[1].

A natural extension is to adapt interpolation search into dynamic data structures, that is, data structures which support insertion and deletion of elements in addition to interpolation search. Their study was started with the works of [6, 8] for insertions and deletions performed according to the uniform distribution, and continued by Mehlhorn and Tsakalidis [13], and Andersson and Mattsson [1] for *μ-random* insertions and *random* deletions, where μ is a so-called *smooth* density. An insertion is μ-random if the key to be inserted is drawn randomly with density function μ; a deletion is random if every key present in the data structure is equally likely to be deleted (these notions of randomness are also described in [10]).

The notion of *smooth* input distributions that determine insertions of elements in the update sequence were introduced in [13], and were further generalized and refined in [1]. Given two functions f_1 and f_2, a density function $\mu = \mu[a,b](x)$ is *(f_1, f_2)-smooth* [1] if there exists a constant β, such that for all c_1, c_2, c_3, $a \leq c_1 < c_2 < c_3 \leq b$, and all integers n, it holds that

$$\int_{c_2 - \frac{c_3 - c_1}{f_1(n)}}^{c_2} \mu[c_1, c_3](x)dx \leq \frac{\beta \cdot f_2(n)}{n}$$

where $\mu[c_1, c_3](x) = 0$ for $x < c_1$ or $x > c_3$, and $\mu[c_1, c_3](x) = \mu(x)/p$ for $c_1 \leq x \leq c_3$ where $p = \int_{c_1}^{c_3} \mu(x)dx$. The class of smooth distributions is a superset of both regular and uniform classes.

In [13] a dynamic interpolation search data structure was introduced, called *Interpolation Search Tree (IST)*. This data structure requires $O(n)$ space for storing n elements. The amortized insertion and deletion cost is $O(\log n)$, while the expected amortized insertion and deletion cost is $O(\log\log n)$. The worst case search time is $O(\log^2 n)$, while the expected search time is $O(\log\log n)$ on sets generated by μ-random insertions and random deletions, where μ is a $(\lceil n^a \rceil, \sqrt{n})$-smooth density function and $\frac{1}{2} \leq a < 1$. An IST is a multi-way tree, where the degree of a node u depends on the number of leaves of the subtree rooted at u (in the ideal case the degree of u is the square root of this number). Each node of the tree is associated with two arrays: a REP array which stores a set of sample elements, one element from each subtree, and an ID array that stores a set of sample elements approximating the inverse distribution function. The search algorithm for the IST uses the ID array in each visited node to interpolate REP and locate the element, and consequently the subtree where the search is to be continued.

[1] A density μ is regular if there are constants b_1, b_2, b_3, b_4 such that $\mu(x) = 0$ for $x < b_1$ or $x > b_2$, and $\mu(x) \geq b_3 > 0$ and $|\mu'(x)| \leq b_4$ for $b_1 \leq x \leq b_2$.

In [1], Andersson and Mattsson explored further the idea of dynamic interpolation search by observing that: (i) the larger the ID array the bigger becomes the class of input distributions that can be efficiently handled with an IST-like construction; and (ii) the IST update algorithms may be simplified by the use of a static, implicit search tree whose leaves are associated with binary search trees and by applying the incremental global rebuilding technique of [14]. The resulting new data structure in [1] is called the *Augmented Sampled Forest (ASF)*. Assuming that $H(n)$ is an increasing function denoting the height of the static implicit tree, Andersson and Mattsson [1] showed that an expected search and update time of $\Theta(H(n))$ can be achieved for μ-random insertions and random deletions where μ is $(n \cdot g(H(n)), H^{-1}(H(n)-1))$-smooth and g is a function satisfying $\sum_{i=1}^{\infty} g(i) = \Theta(1)$. In particular, for $H(n) = \Theta(\log\log n)$ and $g(x) = x^{-(1+\varepsilon)}$ ($\varepsilon > 0$), they get $\Theta(\log\log n)$ expected search and update time for any $(n/(\log\log n)^{1+\varepsilon}, n^{1-\delta})$-smooth density, where $\varepsilon > 0$ and $0 < \delta < 1$ (note that $(\lceil n^a \rceil, \sqrt{n})$-smooth $\subset$ $(n/(\log\log n)^{1+\epsilon}, n^{1-\delta})$-smooth). The worst-case search and update time is $O(\log n)$, while the worst-case update time can be reduced to $O(1)$ if the update position is given by a finger. Moreover, for several but more restricted than the above smooth densities they can achieve $o(\log\log n)$ expected search and update time complexities; in particular, for the uniform and any bounded distribution the expected search and update time becomes $O(1)$. The above are the best results so far in both the realm of dynamic interpolation structures and the realm of dynamic search tree data structures for μ-random insertions and random deletions on the RAM model.

Based upon dynamic interpolation search, we present in this paper a new finger search tree which, for μ-random insertions and random deletions, achieves $O(1)$ worst-case update time and $O(\log\log d)$ expected search time *with high probability* (w.h.p.) in the RAM model of computation for the same class of smooth density functions μ considered in [1] (Sections 3 and 4), thus improving upon the dynamic search structure of Andersson and Mattsson with respect to the expected search time complexity. Moreover, for the same classes of restricted smooth densities considered in [1], we can achieve $o(\log\log d)$ expected search and update time complexities w.h.p. (e.g., $O(1)$ times for the uniform and any bounded distribution). We would like to mention that the expected bounds in [1,13] have not been proved to hold w.h.p. Our worst-case search time is $O(\sqrt{\log d/\log\log d})$. To the best of our knowledge, this is the first work that uses the dynamic interpolation search paradigm in the framework of finger search trees.

Our data structure is based on a rather simple idea. It consists of two levels: the top level is a tree structure, called *static interpolation search tree* (cf. Section 2) which is similar to the static implicit tree used in [1], while the bottom level consists of a family of buckets. Each bucket is implemented by using the fusion tree technique [18]. However, it is not at all obvious how a combination of these data structures can give better bounds, since deletions of elements may create chains of empty buckets. To alleviate this problem and prove the expected search bound, we use an idea of independent interest. We model the insertions and deletions as a combinatorial game of bins and balls (Section 5). This combinatorial game is innovative in the sense that it is not used in a load-balancing context, but it is used to model the behaviour of a dynamic data structure as the one we describe in this paper. We provide upper and lower bounds on the number of elements in a bucket and show that, w.h.p., a bucket never gets empty. This fact implies that w.h.p. there cannot exist chains of empty buckets, which in turn allows us to express the search time bound in terms of the parameter d. Note that the combinatorial game presented here is different from the known approaches for balls and bins games (see e.g., [3]), since in those approaches the bins are considered static and the distribution of balls uniform. On the contrary, the bins in our game are random variables since the distribution of balls is unknown. This also makes the initialization of the game a non-trivial task which is tackled by firstly sampling a number of balls and then determining appropriate bins which allow the almost uniform distribution of balls into them.

2 Preliminaries

In this paper we consider the unit-cost RAM with a word length of w bits, which models what we program in imperative programming languages such as C. The words of RAM are addressable and these addresses are stored in memory words, imposing that $w \geq \log n$. As a result, the universe U consists of integers (or reals represented as floating point numbers; see [2]) in the range $[0, 2^w - 1]$. It is also assumed that the RAM can perform the standard AC^0 operations of addition, subtraction, comparison, bitwise Boolean operations and shifts, as well as multiplications in constant worst-case time on $O(w)$-bit operands.

In the following, we make use of another search tree data structure on a RAM called *q^*-heap* [18]. Let M be the current number of elements in the q^*-heap and let N be an upper bound on the maximum number of elements ever stored in the q^*-heap. Then, insertion, deletion and search operations are carried out in $O(1 + \log M / \log\log N)$ worst-case time after an $O(N)$ preprocessing overhead. Choosing $M = \text{polylog}(N)$, all operations are performed in $O(1)$ time.

In the top level of our data structure we use a tree structure, called *static interpolation search tree*, which is an explicit version of the static implicit tree used in [1] and that uses the REP and ID arrays associated with the nodes of IST. More precisely, the *static interpolation search tree* can be fully characterized by three nondecreasing functions $H(n)$, $R(n)$ and $I(n)$. A static interpolation search tree containing n elements has height $H(n)$, the root has out-degree $R(n)$, and there is an ID array associated with the root that has size $I(n) = n \cdot g(H(n))$ such that $\sum_{i=1}^{\infty} g(i) = \Theta(1)$. To guarantee the height of $H(n)$, it should hold that $n/R(n) = H^{-1}(H(n) - 1)$. The children of the root have $n' = \Theta(n/R(n))$ leaves. Their height will be $H(n') = H(n) - 1$, their out-degree is $R(n') = \Theta(H^{-1}(H(n) - 1)/H^{-1}(H(n) - 2))$, and $I(n') = n' \cdot g(H(n'))$. In general, for an internal node v at depth i containing n_i leaves in the subtree rooted at v, we have that $R(n_i) = \Theta(H^{-1}(H(n) - i + 1)/H^{-1}(H(n) - i))$, and $I(n_i) = n_i \cdot g(H(n) - i)$. As in the case of IST [13], each internal node is associated with an array of sample elements REP, one for each of its subtrees, and an ID array. By using the ID array, we can interpolate the REP array to determine the subtree in which the search procedure will continue. In particular, the ID array for node v is an array ID$[1..m]$, where m is some integer, with ID$[i] = j$ iff REP$[j] < \alpha + i(\beta - \alpha)/m \leq$ REP$[j+1]$, where α and β are the minimum and the maximum element, resp., stored in the subtree rooted at v. Let x be the element we seek. To interpolate REP, compute the index $j =$ ID$[\lfloor((x - \alpha)/(\beta - \alpha))\rfloor]$, and then scan the REP array from REP$[j+1]$ until the appropriate subtree is located. For each node we explicitly maintain parent, child, and sibling pointers. Pointers to sibling nodes will be alternatively referred to as *level links*. The required pointer information can be easily incorporated in the construction of the static interpolation search tree. Throughout the paper, we say that an event E occurs *with high probability* (w.h.p.) if $Pr[E] = 1 - o(1)$.

3 The Data Structure

The data structure consists of two separate structures T_1 and T_2. T_2 is attached a flag *active* denoting whether this structure is valid subject to searches and updates, or invalid. Between two global reconstructions of the data structure, T_1 stores all available elements while T_2 either stores all elements (*active*=TRUE) or a past instance of the set of elements (*active*=FALSE). T_1 is a finger search tree implemented as in [2]. In this way, we can always guarantee worst-case time bounds for searches and updates. In the following we focus on T_2.

T_2 is a two-level data structure, similar to the Augmented Sampled Forest (ASF) presented in [1], but with the following differences: (a) we use the static interpolation search tree defined in Section 2; (b) we implement the buckets associated with the leaves of the static interpolation search tree using q^*-heaps, instead of simple binary search trees; (c) our search procedure does not start from the root of the tree, but we are guided by a finger f to start from an arbitrary leaf; and (d) our reconstruction procedure to maintain our data structure is quite different from that used in [1]. More specifically, let S_0 be the set of elements to be stored where the elements take values in $[a, b]$. The two levels of T_2 are as follows. The bottom level is a set of ρ buckets. Each bucket $\mathcal{B}_i$, $1 \leq i \leq \rho$, stores a subset of elements and is represented by the element $rep(i) = \max\{x : x \in \mathcal{B}_i\}$. The set of elements stored in the buckets constitute an ordered collection $\mathcal{B}_1, \ldots, \mathcal{B}_\rho$ such that $\max\{x : x \in \mathcal{B}_i\} < \min\{y : y \in \mathcal{B}_{i+1}\}$ for all $1 \leq i \leq \rho - 1$. In other words, $\mathcal{B}_i = \{x : x \in (rep(i-1), rep(i)]\}$, for $2 \leq i \leq \rho$, and $\mathcal{B}_1 = \{x : x \in [rep(0), rep(1)]\}$, where $rep(0) = a$ and $rep(\rho) = b$. Each $\mathcal{B}_i$ is implemented as a q^*-heap [18]. The top level data structure is a static interpolation search tree that stores all elements.

Our data structure is maintained by incrementally performing global reconstructions [14]. More precisely, let S_0 be the set of stored elements at the latest reconstruction, and assume that $S_0 = \{x_1, \ldots, x_{n_0}\}$ in sorted order. The reconstruction is performed as follows. We partition S_0 into two sets S_1 and S_2, where $S_1 = \{x_{i \cdot \ln n_0} : i = 1, \ldots, \frac{n_0}{\ln n_0} - 1\} \cup \{b\}$, and $S_2 = S_0 - S_1$. The i-th element of S_1 is the representative $rep(i)$ of the i-th bucket $\mathcal{B}_i$, where $1 \leq i \leq \rho$ and $\rho = |S_1| = \frac{n_0}{\ln n_0}$. An element $x \in S_2$ is stored twice: (i) In the appropriate bucket $\mathcal{B}_i$, iff $rep(i-1) < x \leq rep(i)$, for $2 \leq i \leq \frac{n_0}{\ln n_0}$; otherwise ($x \leq rep(1)$), x is stored in $\mathcal{B}_1$. (ii) As a leaf in the top level structure where it is marked redundant and is equipped with a pointer to the representative of the bucket to which it belongs. We also mark as redundant all internal nodes of the top level structure that span redundant leaves belonging to the same bucket and equip them with a pointer to the representative of the bucket. The reason we store the elements of S_2 twice is to ensure that all elements are drawn from the same μ-random distribution and hence we can safely apply the analysis presented in [1,13]. Also, the reason for this kind of representatives will be explained in Section 5. Note that, after reconstruction, each new element is stored only in the appropriate bucket. Each time the number of updates exceeds rn_0, where r is an arbitrary constant, the whole data structure is reconstructed. Let n be the number of stored elements at this time. After the reconstruction, the number of buckets is equal to $\frac{n}{\ln n}$ and the value of the parameter N, used for the implementation of $\mathcal{B}_i$ with a q^*-heap, is n. Immediately after the reconstruction, if every bucket stores less than polylog(n) elements, then *active*=TRUE, otherwise *active*=FALSE.

In order to insert/delete an element immediately to the right of an existing element f, we insert/delete the element to/from T_1 (using the procedures in [2]), and we insert/delete the element to/from the appropriate bucket of T_2 if *active*=TRUE (using the procedures in [18]). If during an insertion in a bucket of T_2, the number of stored elements becomes greater than polylog(n), then *active*=FALSE. The search procedure for locating an element x in the data structure, provided that a finger f to some element is given, is carried out as follows. If *active*=TRUE, then we search in parallel both structures and we stop when we first locate the element, otherwise we only search in T_1. The search procedure in T_1 is carried out as in [2]. The search procedure in T_2 involves a check as to whether x is to the left or to the right of f. Assume, without loss of generality, that x is to the right of f. Then, we have two cases: (1) Both elements belong to the same bucket $\mathcal{B}_i$. In this case, we just retrieve from the q^*-heap that implements $\mathcal{B}_i$ the element with key x. (2) The elements are stored in different buckets $\mathcal{B}_i$ and $\mathcal{B}_j$ containing f and x respectively. In this case, we start from $rep(i)$ and we walk towards the root of the static interpolation search tree. Assuming that we reach a node v, we check whether x is stored in a descendant of v or in the right neighbour z of v. This can be easily accomplished by checking the boundaries of the REP arrays of both nodes. If they are not stored in the subtrees of v and z, then we proceed to the parent of v, otherwise we continue the search in the particular subtree using the ID and REP arrays. When a redundant node is reached, we follow its associated pointer to the appropriated bucket.

4 Analysis of Time and Space Complexity

In this section we analyze the time complexities of the search and update operations. We start with the case of $(n/(\log\log n)^{1+\epsilon}, n^{1-\delta})$-smooth densities, and later on discuss how our result can be extended to the general case. The tree structure T_2 is updated and queried only in the case where all of its buckets have size polylog(n) (*active*=TRUE), where n is the number of elements in the latest reconstruction. By this and by using some arguments of the analysis in [2] and [18] the following lemma is immediate.

Lemma 1. *The preprocessing time and the space usage of our data structure is $\Theta(n)$. The update operations are performed in $O(1)$ worst-case time.*

The next theorem gives the time complexity of our search operation.

Theorem 1. *Suppose that the top level of T_2 is a static interpolation search tree with parameters $R(s_0) = (s_0)^{1-\delta}$, $I(s_0) = s_0/(\log\log s_0)^{1+\epsilon}$, where $\epsilon > 0$, $0 < \delta < 1$, and $s_0 = \frac{n_0}{\ln n_0}$ with* active=*TRUE. Then, the time complexity of a search operation is equal to $O(\min\{\frac{\log|\mathcal{B}_i|}{\log\log n} + \frac{\log|\mathcal{B}_j|}{\log\log n} + \log\log d, \sqrt{\log d/\log\log d}\})$, where $\mathcal{B}_i$ and $\mathcal{B}_j$ are the buckets containing the finger f and the search element x respectively, d denotes the number of buckets between $\mathcal{B}_i$ and $\mathcal{B}_j$, and n denotes the current number of elements.*

Proof (Sketch). Since *active*=TRUE, the search time is the minimum of searching in each of T_1 and T_2. Searching the former equals $O(\sqrt{\log d/\log\log d})$. It is not hard to see that the search operation in T_2 involves at most two searches in buckets $\mathcal{B}_i$ and $\mathcal{B}_j$, and the traversal of internal nodes of the static interpolation search tree, using ancestor pointers, level links and interpolation search. This traversal involves ascending and descending a subtree of at most d leaves and height $O(\log\log d)$, and we can prove (by modifying the analysis in [1,13]) that the time spent at each node during descend is $O(1)$ w.h.p. $\square$

To prove that the data structure has a low expected search time with high probability we introduce a combinatorial game of balls and bins with deletions (Section 5). To get the desirable time complexities w.h.p., we provide upper and lower bounds on the number of elements in a bucket and we show that no bucket gets empty (see Theorem 6). Combining Theorems 1 and 6 we get the main result of the paper.

Theorem 2. *There exists a finger search tree with $O(\log\log d)$ expected search time with high probability for μ-random insertions and random deletions, where μ is a $(n/(\log\log n)^{1+\varepsilon}, n^{1-\delta})$-smooth density for $\varepsilon > 0$ and $0 < \delta < 1$, and d is the distance between the finger and the search element. The space usage of the data structure is $\Theta(n)$, the worst-case update time is $O(1)$, and the worst-case search time is $O(\sqrt{\log d/\log\log d})$.*

We can generalize our results to hold for the class of $(n\cdot g(H(n)), H^{-1}(H(n)-1))$-smooth densities considered in [1], where $H(n)$ is an increasing function representing the height of the static interpolation tree and g is a function satisfying $\sum_{i=1}^{\infty} g(i) = \Theta(1)$, thus being able to achieve $o(\log\log d)$ expected time complexity, w.h.p., for several distributions. The generalization follows the proof of Theorem 1 by showing that the subtree of the static IST has now height $O(H(d))$, implying the same traversal time w.h.p. (details in the full paper [9]).

Theorem 3. *There exists a finger search tree with $\Theta(H(d))$ expected search time with high probability for μ-random insertions and random deletions, where d is the distance between the finger and the search element, and μ is a $(n \cdot g(H(n)), H^{-1}(H(n)-1))$-smooth density, where $\sum_{i=1}^{\infty} g(i) = \Theta(1)$. The space usage of the data structure is $\Theta(n)$, the worst-case update time is $O(1)$, and the worst-case search time is $O(\sqrt{\log d / \log\log d})$.*

For example, the density $\mu[0,1](x) = -\ln x$ is $(n/(\log^* n)^{1+\epsilon}, \log^2 n)$-smooth, and for this density $R(n) = n/\log^2 n$. This means that the height of the tree with n elements is $H(n) = \Theta(\log^* n)$ and the method of [1] gives an expected search time complexity of $\Theta(\log^* n)$. However, by applying Theorem 3, we can reduce the expected time complexity for the search operation to $\Theta(\log^* d)$ and this holds w.h.p. If μ is bounded, then it is $(n,1)$-smooth and hence $H(n) = O(1)$, implying the same expected search time with [1] but w.h.p.

5 A Combinatorial Game of Bins and Balls with Deletions

In this section we describe a balls-in-bins random process that models each update operation in the structure T_2 presented in Section 3. Consider the structure T_2 immediately after the latest reconstruction. It contains the set S_0 of n elements (we shall use n for notational simplicity) which are drawn randomly according to the distribution $\mu(\cdot)$ from the interval $[a,b]$. The next reconstruction is performed after rn update operations on T_2, where r is a constant. Each update operation is either a uniformly at random deletion of an existing element from T_2, or a μ-random insertion of a new element from $[a,b]$ into T_2. To model the update operations as a balls-in-bins random process, we do the following.

We represent each selected element from $[a,b]$ as a *ball*. We partition the interval $[a,b]$ into $\rho = \frac{n}{\ln n}$ parts $[rep(0), rep(1)] \cup (rep(1), rep(2)] \cup \ldots \cup (rep(\rho - 1), rep(\rho)]$, where $rep(0) = a$, $rep(\rho) = b$, and $\forall i = 1, \ldots, \rho - 1$, the elements $rep(i) \in [a,b]$ are those defined in Section 3. We represent each of these ρ parts as a distinct *bin*. During each of the rn insertion/deletion operations in T_2, a μ-random ball $x \in [a,b]$ is inserted in (deleted from) the i-th bin $\mathcal{B}_i$ iff $rep(i-1) < x \leq rep(i)$, $i = 2, \ldots, \rho$; otherwise x, is inserted in (deleted from) $\mathcal{B}_1$.

Our aim is to prove that w.h.p. the maximum load of any bin is $O(\ln n)$, and that no bin remains empty as $n \to \infty$. If we were knowing the distribution $\mu(\cdot)$, then we could partition the interval $[a,b]$ into ρ distinct bins $[rep_\mu(0), rep_\mu(1)] \cup (rep_\mu(1), rep_\mu(2)] \cup \ldots \cup (rep_\mu(\rho-1), rep_\mu(\rho)]$, with $rep_\mu(0) = a$ and $rep_\mu(\rho) = b$, such that a μ-random ball x would be equally likely to belong into any of the ρ corresponding bins with probability $Pr[x \in (rep_\mu(i-1), rep_\mu(i)]] = \int_{rep_\mu(i-1)}^{rep_\mu(i)} \mu(t)dt = \frac{1}{\rho} = \frac{\ln n}{n}$. The above expression implies that the sequence $rep_\mu(0), \ldots, rep_\mu(\rho)$ makes the event "insert (delete) a μ-random (random) element x into (from) the structure" equivalent to the event "throw (delete) a ball uniformly at random into (from) one of ρ distinct bins". Such a uniform distribution of balls into bins is well understood and it is folklore to find conditions such that no bin remains empty and no bin gets more than $O(\ln n)$ balls.

Unfortunately, the probability density $\mu(\cdot)$ is unknown. Consequently, our goal is to *approximate* the unknown sequence $rep_\mu(0), \ldots, rep_\mu(\rho)$ with a sequence $rep(0), \ldots, rep(\rho)$, that is, to partition the interval $[a,b]$ into ρ parts $[rep(0), rep(1)] \cup (rep(1), rep(2)] \cup \ldots \cup (rep(\rho-1), rep(\rho)]$, aiming to prove that each bin (part) will have the key property: $Pr[x \in (rep(i-1), rep(i)]] = \int_{rep(i-1)}^{rep(i)} \mu(t)dt = \Theta\left(\frac{1}{\rho}\right) = \Theta\left(\frac{\ln n}{n}\right)$. The sequence $rep(0), \ldots, rep(\rho)$ makes the event "insert (delete) a μ-random (random) element x into (from) the structure" equivalent to the event "throw (delete) a ball *almost* uniformly at random into one of ρ distinct bins". This fact will become the cornerstone in our subsequent proof that no bin remains empty and almost no bin gets more than $\Theta(\ln n)$ balls.

The basic insight of our approach is illustrated by the following random game. Consider the part of the horizontal axis spanned by $[a,b]$, which will be referred to as the $[a,b]$ *axis*. Suppose that only a wise man knows the positions on the $[a,b]$ axis of the sequence $rep_\mu(0), \ldots, rep_\mu(\rho)$, referred to as the *red dots*. Next, perform n independent insertions of μ-random elements from $[a,b]$ (this is the role of the set S_0). In each insertion of an element x, we add a *blue dot* in its position on the $[a,b]$ axis. At the end of this random game we have a total of n blue dots in this axis. Now, the wise man reveals the red dots on the $[a,b]$ axis, i.e., the sequence $rep_\mu(0), \ldots, rep_\mu(\rho)$. If we start counting the blue dots *between* any two consecutive red dots $rep_\mu(i-1)$ and $rep_\mu(i)$, we almost always find that there are $\ln n + o(1)$ blue dots. This is because the number X_i^μ of μ-random elements (blue dots) selected from $[a,b]$ that belong in $(rep_\mu(i-1), rep_\mu(i)]$, $i = 1, \ldots, \rho$, is a Binomial random variable, $X_i^\mu \sim B(n, \frac{1}{\rho} = \frac{\ln n}{n})$, which is sharply concentrated to its expectation $E[X_i^\mu] = \ln n$.

The above discussion suggests the following procedure for constructing the sequence $rep(0), \ldots, rep(\rho)$. Partition the sequence of n blue dots on the $[a,b]$ axis into $\rho = \frac{n}{\ln n}$ parts, each of size $\ln n$. Set $rep(0) = a$, $rep(\rho) = b$, and set as $rep(i)$ the $i \cdot \ln n$-th blue dot, $i = 1, \ldots, \rho - 1$. Call this procedure `Red-Dots`.

The above intuitive argument does not imply that $\lim_{n\to\infty} rep(i) = rep_\mu(i)$, $\forall i = 0, \ldots, \rho$. Clearly, since $rep_\mu(i)$, $i = 0, \ldots, \rho$, is a real number, the probability that at least one blue dot *hits* an invisible red dot is insignificant. The above argument stresses on the following fact whose proof can be found in [9].

Theorem 4. *Let $rep(0), rep(1), \ldots, rep(\rho)$ be the output of procedure* `Red-Dots`*, and let $p_i(n) = \int_{rep(i-1)}^{rep(i)} \mu(t)dt$. Then:*
$\Pr\left[\exists\ i \in \{1, \ldots m\} : p_i(n) \neq \Theta\left(\frac{1}{\rho}\right) = \Theta\left(\frac{\ln n}{n}\right)\right] \to 0.$

The above discussion and Theorem 4 imply the following.

Corollary 1. *If n elements are μ-randomly selected from $[a,b]$, and the sequence $rep(0), \ldots, rep(\rho)$ from those elements is produced by procedure* `Red-Dots`*, then this sequence partitions the interval $[a,b]$ into ρ distinct bins (parts) $[rep(0), rep(1)] \cup (rep(1), rep(2)] \cup \ldots \cup (rep(\rho-1), rep(\rho)]$ such that a ball $x \in [a,b]$*

can be thrown (deleted) independently *of any other ball in* $[a, b]$ *into (from) any of the bins with probability* $p_i(n) = \Pr[x \in (rep(i-1), rep(i)]] = \frac{c_i \ln n}{n}$*, where* $i = 1, \ldots, \rho$ *and* c_i *is a positive constant.*

Definition 1. *Let* $c = \min_i\{c_i\}$ *and* $C = \max_i\{c_i\}$*,* $i = 1, \ldots, \rho$*, where* $c_i = \frac{np_i(n)}{\ln n}$*.*

We now turn to the randomness properties in each of the rn subsequent insertion/deletion operations on the structure T_2 (r is a constant).

Observe that before the process of rn insertions/deletions starts, each bin $\mathcal{B}_i$ (i.e., part $(rep(i-1), rep(i)]$) contains exactly $\ln n$ balls (blue dots on the $[a, b]$ axis) of the n initial balls of the set S_0. For convenience, we analyze a slightly different process of the subsequent rn insertions/deletions. Delete all elements (balls) of S_0 except for the representatives $rep(0), rep(1), \ldots, rep(\rho)$ of the ρ bins. Then, insert μ-randomly n/c (see Definition 1) new elements (balls) and subsequently start performing the rn insertions/deletions. Since the n/c new balls are thrown *μ-randomly* into the ρ bins $[rep(0), rep(1)] \cup (rep(1), rep(2)] \cup \ldots \cup (rep(\rho-1), rep(\rho)]$, by Corollary 1 the initial number of balls into $\mathcal{B}_i$ is a Binomial random variable that obeys $B(n/c, p_i(n))$, $i = 1, \ldots, \rho$, instead of being fixed to the value $\ln n$. Clearly, if we prove that for this process no bin remains empty and does not contain more than $O(\ln n)$ balls, then this also holds for the initial process.

Let the random variable $M(j)$ denote the number of balls existing in structure T_2 at the end of the j-th insertion/deletion operation, $j = 0, \ldots, rn$. Initially, $M(0) = n/c$. The next useful lemma allows us to keep track of the statistics of an arbitrary bin. Part (i) follows by Corollary 1 and an induction argument, while part (ii) is an immediate consequence of part (i).

Lemma 2. *(i) Suppose that at the end of j-th insertion/deletion operation there exist $M(j)$ distinct balls that are μ-randomly distributed into the ρ distinct bins. Then, after the $(j+1)$-th insertion/deletion operation the $M(j+1)$ distinct balls are also μ-randomly distributed into the ρ distinct bins. (ii) Let the random variable $Y_i(j)$ with $(i, j) \in \{1, \ldots, \rho\} \times \{0, \ldots, rn\}$ denote the number of balls that the i-th bin contains at the end of the j-th operation. Then, $Y_i(j) \sim B(M(j), p_i(n))$.*

To study the dynamics of $M(j)$ at the end of j-th operation, observe that in each operation, a ball is either inserted with probability $p > 1/2$, or is deleted with probability $1 - p$. $M(j)$ is a discrete random variable which has the nice property of sharp concentration to its expected value, i.e., it has small deviation from its mean compared to the total number of operations. In the following, instead of working with the actual values of j and $M(j)$, we shall use their *scaled* (divided by n) values t and $m(t)$, resp., that is, $t = \frac{j}{n}$, $m(t) = \frac{M(tn)}{n}$, with range $(t, m(t)) \in [0, r] \times [1, m(r)]$. The sharp concentration property of $M(j)$ leads to the following theorem (whose proof can be found in [9]).

Theorem 5. *For each operation $0 \leq t \leq r$, the scaled number of balls that are distributed into the $\frac{n}{\ln(n)}$ bins at the end of the t-th operation equals $m(t) = (2p-1)t + o(1)$, w.h.p.*

Remark 1. Observe that for $p > 1/2$, $m(t)$ is an increasing positive function of the scaled number t of operations, that is, $\forall\ t \geq 0,\ M(tn) = m(t)n \geq M(0) = m(0)n = n/c$. This implies that if no bin remains empty before the process of rn operations starts, since for $p > 1/2$ the balls accumulate as the process evolve, then no bin will remain empty in each subsequent operation. This is important on proving part (i) of Theorem 6.

Finally, we turn to the statistics of the bins. We prove that before the first operation, and for all subsequent operations, w.h.p., no bin remains empty. Furthermore, we prove that during each step the maximum load of any bin is $\Theta(\ln(n))$ w.h.p. For the analysis below we make use of the Lambert function $LW(x)$, which is the analytic at zero solution with respect to y of the equation: $ye^y = x$ (see [4]). Recall also that during each operation $j = 0, \ldots, rn$ with probability $p > 1/2$ we insert a μ-random ball $x \in [a, b]$, and with probability $1-p$ we delete an existing ball from the current $M(j)$ balls that are stored in the structure T_2.

Theorem 6. *(i) For each operation* $0 \leq t \leq r$*, let the random variable* $X(t)$ *denote the current number of empty bins. If* $p > 1/2$*, then for each operation* t*,* $E[X(t)] \to 0$*. (ii) At the end of operation* t*, let the random variable* $Z_\kappa(t)$ *denote the number of bins with load at least* $\kappa \ln(n)$*, where* $\kappa = \kappa(t)$ *satisfies* $\kappa \geq (-Cm(t) + 2)/(C \cdot LW(-\frac{Cm(t)-2}{Cm(t)e})) = O(1)$*, and* C *is the positive constant defined in Definition 1. If* $p > 1/2$*, then for each operation* t*,* $E[Z_\kappa(t)] \to 0$*.*

Proof. (i) Recall the definitions of the positive constants c and C (Definition 1). From Lemma 2, $\forall\ i = 1, \ldots, \rho = \frac{n}{\ln(n)}$, it holds:

$$Pr[Y_i(t) = 0] \leq \left(1 - c\frac{\ln(n)}{n}\right)^{m(t)n} \sim e^{-cm(t)\ln(n)} = \frac{1}{n^{cm(t)}}. \qquad (1)$$

From Eq. (1), by linearity of expectation, we obtain:

$$E[X(t) \mid m(t)] \leq \sum_{i=1}^{\rho} Pr[Y_i(t) = 0] \leq \frac{n}{\ln(n)} \cdot \frac{1}{n^{cm(t)}}. \qquad (2)$$

From Theorem 5 and Remark 1 it holds: $\forall\ t \geq 0,\ \frac{1}{n^{cm(t)}} \leq \frac{1}{n^{cm(0)}} = \frac{1}{n}$. This inequality implies that in order to show for each operation t that the expected number $E[X(t) \mid m(t)]$ of empty bins vanishes, it suffices to show that before the process starts, the expected number $E[X(0) \mid m(0)]$ of empty bins vanishes. In this line of thought, from Theorem 5, Eq. (2) becomes,

$$E[X(0) \mid m(0)] \leq \frac{n}{\ln(n)} \cdot \frac{1}{n^{cm(0)}} = \frac{n}{\ln(n)} \cdot \frac{1}{n} = \frac{1}{\ln(n)} \to 0.$$

Finally, from Markov's inequality, we obtain

$$Pr[X(t) > 0 \mid m(t)] \leq E[X(t) \mid m(t)] \leq E[X(0) \mid m(0)] \to 0.$$

(ii) In the full paper [9] due to space limitations. □

References

1. A. Andersson and C. Mattson. Dynamic Interpolation Search in $o(\log\log n)$ Time. In *Proc. ICALP'93*.
2. A. Anderson and M. Thorup. Tight(er) Worst-case Bounds on Dynamic Searching and Priority Queues. In *Proc. 32nd ACM Symposium on Theory of Computing* – STOC 2001, pp. 335–342. ACM, 2000.
3. R. Cole, A. Frieze, B. Maggs, M. Mitzenmacher, A. Richa, R. Sitaraman, and E. Upfal. On Balls and Bins with Deletions. In *Randomization and Approximation Techniques in Computer Science* – RANDOM'98, Lecture Notes in Computer Science Vol. 1518 (Springer-Verlag, 1998), pp. 145–158.
4. R.M. Corless, G.H. Gonnet, D.E.G. Hare, D.J. Jeffrey, and D.E. Knuth. On the Lambert W Function. *Advances in Computational Mathematics* 5:329–359, 1996.
5. P. Dietz and R. Raman. A Constant Update Time Finger Search Tree. *Information Processing Letters*, 52:147–154, 1994.
6. G. Frederickson. Implicit Data Structures for the Dictionary Problem. *Journal of the ACM* 30(1):80–94, 1983.
7. G. Gonnet, L. Rogers, and J. George. An Algorithmic and Complexity Analysis of Interpolation Search. *Acta Informatica* 13(1):39–52, 1980.
8. A. Itai, A. Konheim, and M. Rodeh. A Sparse Table Implementation of Priority Queues. In *Proc. ICALP'81*, Lecture Notes in Computer Science Vol. 115 (Springer-Verlag 1981), pp. 417–431.
9. A. Kaporis, C. Makris, S. Sioutas, A. Tsakalidis, K. Tsichlas, and C. Zaroliagis. Improved Bounds for Finger Search on a RAM. Tech. Report TR-2003/07/01, Computer Technology Institute, Patras, July 2003.
10. D.E. Knuth. Deletions that preserve randomness. *IEEE Trans. Softw. Eng.* 3:351–359, 1977.
11. C. Levcopoulos and M.H. Overmars. A Balanced Search Tree with $O(1)$ Worst Case Update Time. *Acta Informatica*, 26:269–277, 1988.
12. K. Mehlhorn and A. Tsakalidis. *Handbook of Theoretical Computer Science – Vol I: Algorithms and Complexity*, Chapter 6: Data Structures, pp. 303-341, The MIT Press, 1990.
13. K. Mehlhorn and A. Tsakalidis. Dynamic Interpolation Search. *Journal of the ACM*, 40(3):621–634, July 1993.
14. M. Overmars, J. Leeuwen. Worst Case Optimal Insertion and Deletion Methods for Decomposable Searching Problems. *Information Processing Letters*, 12(4):168–173.
15. Y. Pearl, A. Itai, and H. Avni. Interpolation Search – A $\log\log N$ Search. *Communications of the ACM* 21(7):550–554, 1978.
16. W.W. Peterson. Addressing for Random Storage. *IBM Journal of Research and Development* 1(4):130–146, 1957.
17. D.E. Willard. Searching Unindexed and Nonuniformly Generated Files in $\log\log N$ Time. *SIAM Journal of Computing* 14(4):1013–1029, 1985.
18. D.E. Willard. Applications of the Fusion Tree Method to Computational Geometry and Searching. In *Proc. 3rd ACM-SIAM Symposium on Discrete Algorithms* – SODA'92, pp. 286–295, 1992.
19. A.C. Yao and F.F. Yao. The Complexity of Searching an Ordered Random Table. In *Proc. 17th IEEE Symp. on Foundations of Computer Science* – FOCS'76, pp. 173–177, 1976.