Università di Pisa

Dipartimento di Informatica

Technical Report: TR-03-03

Optimal Space-Time Dictionaries over an Unbounded Universe with Flat Implicit Trees

Gianni Franceschini
francesc@di.unipi.it

*Roberto Grossi**
grossi@di.unipi.it

January 30, 2003

ADDRESS: via Buonarroti 2, 56127 Pisa, Italy. TEL: +39 050 2212700 FAX: +39 050 2212726

Optimal Space-Time Dictionaries over an Unbounded Universe with Flat Implicit Trees

Gianni Franceschini * `francesc@di.unipi.it`

*Roberto Grossi** `grossi@di.unipi.it`

January 30, 2003

Abstract

In the classical dictionary problem, a set of n distinct keys over an unbounded and ordered universe is maintained under insertions and deletions of individual keys while supporting search operations. An implicit dictionary has the additional constraint of occupying the space merely required by storing the n keys, that is, exactly n contiguous words of space in total. All what is known is the starting position of the memory segment hosting the keys, as the rest of the information is implicitly encoded by a suitable permutation of the keys. This paper describes the flat implicit tree, which is the first implicit dictionary requiring $O(\log n)$ time per search and update operation.

1 Introduction

Almost every introductory textbook on algorithms and data structures presents, among others, two ways of storing n distinct keys a_1, a_2, $\ldots$, a_n into an array of n memory cells, each capable of containing one key. The keys can be stored in sorted order so that a binary search runs in $O(\log n)$ time. (Knuth [16] credits Mauchly (1946) for inventing this organization of the keys.) Also, the keys can be suitable permuted for obtaining the heap of Williams and Floyd (1964), thus permitting to identify the current maximum key in constant time and supporting insertions and deletions of individual keys in $O(\log n)$ time [26, 10]. Indeed, sorted arrays and heaps are well-known examples of dictionaries.

In the dictionary problem a set of n distinct keys $a_1, a_2, \ldots, a_n$ is maintained over a total order, in which the only operations allowed on the keys are reads/writes and comparisons using the standard RAM model of computation [2]. The dictionary supports the operations of searching, inserting and deleting an arbitrary key x. Besides membership, searching may also involve finding the predecessor or the successor of x, or reporting all the keys ranging in an interval $[x, x']$ where $x' \geq x$. Heaps have the drawback of requiring $O(n)$ time for searching operation, while inserting or deleting a key in the middle part of sorted arrays may take $O(n)$ time. A longstanding question is whether there exists an organization of the keys in an array of n cells combining the best qualities of sorted arrays and heaps, so that each operation requires $O(\log n)$ time.

A common feature of sorted arrays and heaps is that they both store a suitable permutation of the n keys, encoding an *implicit* tree by a partial order fixed *a priori* on the positions of the keys. No other "structural information" is required other than the keys in the permutation perceived as growing or shrinking with n. Along the same lines, we can see an *implicit* data structure for the dictionary problem as a dynamic scheme for organizing n distinct keys a_1, a_2, $\ldots$, a_n suitably permuted in n memory words, $n > 1$, where each key a_i occupies a distinct word. A snapshot of the memory shows a contiguous segment of n adjacent words containing a_1, a_2, $\ldots$, a_n permuted, with *no* further occupied or wasted word. All what is known is the starting position of the segment, as the rest of the information (including the value of n) is implicitly encoded by the permutation of $a_1, a_2, \ldots, a_n$. The segment can be enlarged or shortened to the right by one word at a time in constant time. Inserting a new key a_{n+1} grows the segment by appending one word storing a_{n+1}, and shuffles a_1, a_2, $\ldots$, a_n, a_{n+1} to encode the resulting data structure by a suitable permutation in $n+1$ words. Deleting key a_i shrinks the segment by one word, producing a new permutation of a_1, a_2, $\ldots$, a_{i-1}, a_{i+1}, $\ldots$, a_n in $n-1$ words.

The above encoding of dictionaries by permutations of $a_1, a_2, \ldots, a_n$ is admissible by a simple information-theoretical argument showing that the number of permutations is much larger than the number of trees.

*Dipartimento di Informatica, Università di Pisa, via Filippo Buonarroti 2, 56127 Pisa, Italy.

Nonetheless, no previous implicit data structure for the dictionary problem takes $O(\log n)$ time per operation, to the best of our knowledge. This fact may seem rather surprising considering that sorted arrays and heaps are long-lived examples of *implicit* dictionaries. As we shall see, the problem is algorithmically challenging and extending the implicit structure of sorted arrays and heaps is far from being an easy task. In order to support the full repertoire of insert, delete and search operations in $O(\log n)$ time, the alternative is implementing dictionaries as dynamic linked data structures such as AVL trees [1] and other balanced data structures. As a matter of fact they are not implicit, requiring strictly more than n words of space because of $O(n)$ extra pointers and integers. In this paper, we present the *flat implicit tree* (FIT), which is the first implicit data structures attaining $O(\log n)$ time for searching, inserting and deleting. The flat implicit tree provides the first optimal space-time bounds for the dictionary problem on keys over an unbounded universe, where optimality in space follows from its implicitness like sorted arrays and heaps.

History of the problem. Munro and Suwanda [20] examined the general case of ordered keys belonging to an *unbounded* universe, where the $\Omega(\log n)$ lower bound on search time derives from the comparison model [16]. They were the first to introduce the notion of implicit data structures inspired by the heap of Williams and Floyd, mentioning a previous (unpublished) result by Bentley *et al.* [4] supporting only insertions and searches. While the term "implicit" originated in [20], it has also been the subject of papers taking a somewhat different point of view, including a long lists of results in perfect hashing [14, 8], bounded-universe dictionaries [7, 21], and cache-oblivious data structures [6, 24]. These results were obtained for less stringent models different from the model adopted by Munro and Suwanda and in following papers.

Yao [27] examined the special case of keys belonging to a *bounded* universe U, so that each key can be interpreted as an integer ranging in $0 \dots |U| - 1$ and occupying a word of size $w = \Omega(\log |U|)$ bits. He proved that, independently of how the n keys are permuted inside a segment of n words, searching requires $\Omega(\log n)$ time for sufficiently large U. However, he showed that encoding some information in *one extra* word of space (e.g., the name of a hash function) gives more computational power and makes constant-time membership search possible for sufficiently large U. Since then, the two-level scheme by Fredman, Komlós and Szemerédi [14] and the many related papers provided a burst of interest in the design and the analysis of efficient algorithms for perfect hashing in constant-time search with $n + \omega(1)$ words of space. Recent improvements in this direction are described in Faith and Miltersen [9] and Raman, Raman and Rao [23], encoding the keys in at most n words of space by avoiding to store explictly a permutation the keys as required in the lower bound of Yao. We remark that these techniques are not viable in our case, since they do not support all the dictionaries primitives (e.g., range searching) and the keys in an unbounded universe are atomic and can only be read, compared, and written.

Several papers faced the problem of designing an implicit data structure for the dictionary problem over an unbounded universe. Munro and Suwanda gave a lower bound of $\Omega(\sqrt{n})$ time per operation on implicit data structures based on *a priori* partial orders, such as the sorted array and the heap. Their biparental heap matches that lower bound, giving $O(\sqrt{n})$ time per operation. They also showed how to beat the lower bound by using a partial order that is *not* fixed *a priori* and that is based on rotated lists, achieving $O(n^{1/3} \log n)$ time per operation. Frederickson [13] presented a collection of data structures recursively using rotated lists, requiring just $O(1)$ RAM registers to operate dynamically. Search time is $O(\log n)$ while insertions and deletions are supported in $O(n^{\sqrt{2/\log n}} \log^{3/2} n) = o(n^\epsilon)$ time, for any fixed value of $\epsilon > 0$. Exploiting the properties of an in-place merge that is $O(n)$-time searchable at any time of its execution, Munro and Poblete [19] provided an implicit data structure with $O(\log^2 n)$ search time supporting only insertions in $O(\log n)$ time. In the mid 80s, Munro [18] achieved the first poly-logarithmic bounds holding simultaneously for searches and updates. Going through the crucial idea of encoding $O(\log n)$ bits by a permutation of $O(\log n)$ keys as described in Section 2, he attained $O(\log^2 n)$ time by a variant of AVL trees [1], with $O(\log n)$ height and $O(\log n)$ accessed keys per level to encode pointers. The paper speculated that $\Theta(\log^2 n)$ may be optimal until very recently. Franceschini *et al.* [12] reopened the issue of obtaining $o(\log^2 n)$ time per operation and introduced the implicit B-tree, whose cost is $O(\log_B n)$ memory transfers for a block size $B = \Omega(\log n)$. When employed in main memory, its running time is $O(\log^2 n / \log\log n)$ per operation as its height is $O(\log n / \log\log n)$, with $O(\log n)$ accessed keys per level. Franceschini and Grossi proved in [11] that $O(\log n \log\log n)$ time per operation is doable in the amortized sense using a data structure of height $O(\log\log n)$, thus leaving open the $O(\log n)$ bound.

Borodin et al. [5] and Radhakrishnan and Raman [22] gave an interesting tradeoff between data moves in performing an update and the number of comparisons necessary for a search. Their lower bound does not, however, rule out the $O(\log n)$ behavior for the problem. They motivate the study of implicit data structures as an important topic in characterizing the set of permutations that are searchable and updatable in logarithmic time.

Our results. The implicit dictionary problem is intimately related to the fundamental question whether using *extra* space gives more computational power than using *exactly* n words, and how data ordering can help in this task. The ultimate goal is in the flavor of the theoretical result of Yao [27], but in a *dynamic* setting and with an *unbounded* universe: does it exist an implicit dictionary over an unbounded universe with optimal $\Theta(\log n)$ time complexity per search, insert and delete operation? or should any $\Theta(\log n)$-time dictionary occupy strictly more than n words?

In this paper, we give a positive answer to the long-standing question above. We describe an implicit data structure, called *flat implicit tree*, which requires $O(\log n)$ time for searching and $O(\log n)$ amortized time for updating, with just $O(1)$ RAM registers needed to operate dynamically. An interesting feature of our data structure is that it achieves the best space saving possible. In this context self-adjusting search trees [25] and random search trees [17] are often mentioned for their ability, among others, to avoid balancing information for saving space while requiring $O(\log n)$ amortized or expected time. However, they still need $n + \Omega(n)$ words of memory (for the pointers). Compared to previous work in the field, the flat implicit tree is relatively simple, with its simplicity being made possible by the significant steps in data structuring analysis described in [3] and by the recent findings in [12] and [11]. The programming difficulty of maintaining our implicit data structure is comparable to that of other (explicit) balanced data structures.

Our solution hinges on a hybrid approach using the partial order fixed *a priori* in sorted arrays to handle a large segment of quasi-sorted data (the super-root), and the flexible encoding of [18] to handle a large collection of relatively small buckets of permuted keys (the intermediate nodes and the leaves). Ideally, we would like to obtain a constant depth with $O(\log n)$ accessed keys per traversed node, which is the problem left open in previous work. We introduce new ideas to maintain a flat data structure with just three levels of nodes: a super-root, a level of large intermediate nodes, and a level of small leaves. Although the initial portions of some nodes and leaves may be fragmented in more than a constant number of parts in some cases, we can guarantee that just $O(\log n)$ keys are accessed in each of the three levels.

The paper is organized as follows. Section 2 gives an overview of the flat implicit tree organized in two layers, with Section 3 describing the bottom layer (intermediate nodes and leaves) and Section 4 giving a description of the top layer (super-root). Section 5 describes the cost of rebuilding the data structure and the final analysis of the supported operations.

2 Overview

In this paper, we encode data by a pairwise (odd-even) permutation of keys [18]. To encode a pointer or an integer of b bits by using $2b$ distinct keys $x_1, y_1, x_2, y_2, \ldots, x_b, y_b$, we permute them in pairs x_i, y_i by the following rule: if the ith bit is 0, then $\min\{x_i, y_i\}$ precedes $\max\{x_i, y_i\}$; else, the bit is 1 and $\max\{x_i, y_i\}$ precedes $\min\{x_i, y_i\}$. We distinguish between encoding *absolute* pointers and *relative* pointers. The former pointers are always encoded as displacements from the first position in the memory segment allocated for the keys, while the latter ones are encoded referring to some internal position p in the segment. When the value of p changes, we have to re-encode all the relative pointers referring to p.

The flat implicit tree is a suitable collection of *chunks* [18, 12, 11], where each chunk contains k (pairwise permuted) keys encoding a constant number of integers and pointers, each of $b = O(\log n)$ bits. The keys in any chunk belong to a certain interval of values, and the chunks are pairwise disjoint when considered as intervals. This allows us to define a total order on any set of the chunks, so that we can write $c_1 < c_2$ for any two chunks meaning that the keys in chunk c_1 are all smaller than those in chunk c_2. Differently from previous work on implicit data structures, we keep the invariant that the number of keys satisfies $n'/4 < n < n'$, where n' is a power of two, thus fixing the chunk size $k = \Theta(\log n')$. We resize k only when either $n = n'/4$ or $n = n'$; we describe how to maintain this invariant in Section 5. We avoid to keep the value of n and n' explicitly, as a variable-length encoding, such as the δ-code, can represent them asymptotically in the first $O(\log n)$ permuted keys of the data structure.

From a high-level point of view, our data structure contains n distinct keys organized in two layers and is parametric in n' and k, which are periodically resized with a reconstruction.

The *top* layer is the *super-root* containing $a = \Theta(n/k^2)$ chunks stored in sorted order, called *actual* chunks, where a is *always* a power of two. Next, it contains a number of *virtual* chunks, stored in no particular order, with $O(1)$ virtual chunks associated with each actual chunk, as described in Section 4. Specifically, for an actual chunk c, we require that c and its associated virtual chunks are consecutive in the total order defined over all the chunks in the top layer.

The *bottom* layer is a dynamic implicit forest, where each tree implements a bucket of $\Theta(k^2)$ keys organized into chunks, called *bucket* chunks, plus some spare keys that will be handled differently. The root of each tree is either an actual chunk or a virtual chunk, so the keys in the roots of the forest are stored in the top layer. Given any two chunks c_1 and c_2 in the top layer with $c_1 < c_2$, the keys in the bucket having c_1 as root are all greater than the keys in c_1 and smaller than the ones in c_2.

Since the actual chunks are in a sorted array of the top layer, we can exploit their order to access the associated virtual chunks. As a result, we can quickly identify the bucket of keys that might contain a given key. Searching follows this idea by performing first a binary search on the actual chunks (one key per chunk is enough). Once an actual chunk c is identified, we can retrieve its associated $O(1)$ virtual chunks (if they exist) and, among their $O(1)$ buckets, we can identify the bucket in which to perform the search. We aim at decoding integers and pointers from a total of $O(\log n)$ keys, so that the total cost is $O(\log n)$ time.

As customary to implicit data structures, we provide a view of the memory layout of the flat implicit tree as this heavily affects the complexity of the operations. We have three contiguous areas (see Figure 1):

- A preamble $\mathscr{P}$ of $O(k)$ keys encoding $O(1)$ pointers and integers for bookkeeping purposes.
- The super-root area storing the keys in the top layer: first the actual chunks in sorted order and, then, the virtual chunks, in no particular order. The area may grow or shrink by k positions to its right.
- The intermediate and leaf area storing the bucket chunks and the spare keys in the bottom layer. The area may grow or shrink by k positions to its left and by one position to its right.

In what follows, the bounds are intended to hold in the worst case, unless we explicitly mention them as amortized bounds. We also define the size of the several entities (nodes, areas, zones, etc.) as the number of keys contained in each of them. In the rest of the paper, we give the details of our main result:

Theorem 1 *The flat implicit tree for n keys can be maintained under insertions and deletions in $O(\log n)$ amortized time per operation using just $O(1)$ RAM registers, so that searching a key takes $O(\log n)$ time. The only operations performed on the keys are comparisons and moves and, furthermore, no additional memory cells are required besides those storing the keys.*

3 Bottom Layer: Buckets as Implicit Dynamic Forest

We describe the bottom layer, where we store the set of buckets in the form of an implicit dynamic forest. From a high-level point of view, each bucket is an implicit tree whose root is either an actual or a virtual chunk in the top layer. We need to insert, to delete and to search a key in a bucket, as required by the dictionary. Moreover, in case of bucket overflows (too many keys) and bucket underflows (too few keys), we need splitting the bucket or merging/sharing it with a neighbor bucket, while preserving implicitness.

Bucket chunks. Each bucket is a tree made up of bucket chunks, except for the root, which is either an actual or a virtual chunk. The leaves contain from 1 to 4 chunks plus at most $k-1$ keys (i.e, their size is from k to $5k-1$). The parent of the leaves, called *intermediate*, is the only child of the root. It contains from k to $4k$ bucket chunks (i.e, its size is from k^2 to $4k^2$), with so many children. The pointer to the ith child leaf is encoded by $O(\log n)$ keys in the ith chunk. In their lifetime, the intermediate nodes may change size by k (one chunk) at a time, while the leaves may change size by 1 (one key).

Memory layout. The layout of the roots is discussed in Section 4. Here we detail how to lay out the intermediate nodes and the leaves in the bottom layer. We accommodate the keys in fixed-size *allocation units*, which are made up of keys stored in contiguous memory cells. We link the allocations units in several *compactor* lists to eliminate most of the memory waste [12] (and we encode the links). We will handle k^2-lists, k-lists and ℓ-lists, where the parameter specifies the size of the allocation units. For example, in k^2-lists, the allocation units are of size k^2, while they are of size $\ell = \Theta(\log\log n)$ in ℓ-lists (the precise value of ℓ will be defined later on). Apart from this difference, they share common techniques. For instance, when relocating an allocation unit from one position in memory to another, we have to search some keys in the flat implicit tree. Indeed $O(1)$ pointers entering that allocation unit must be redirected by re-permuting their $O(\log n)$ encoding keys, which are identified through the search.

We pack together the nodes of *identical* size s, embedding them in a suitable compactor list devoted to nodes of size s. When a node changes size, then it also changes compactor list. Each node possibly occupies

$O(1)$ allocation units encoding the links among those units. As a result, to retrieve the keys in a node we may have to jump in $O(1)$ locations. Moreover, the first and last occupied unit may be shared with other nodes to guarantee that the allocation units are completely filled. In each compactor list, however, the first allocation unit (the *head*) is special since it is the only one being partially filled in that list. In general, since the heads are the only allocation units partially filled, we have to dynamically maintain them packed in a proper way. Before going on, we need to refine the partition of the memory layout (see Figure 1):

- the intermediate area contains all the intermediate nodes of the buckets, and grows or shrinks by k positions at a time to both its left and right;
- the leaf area contains all the leaves of the buckets, and grows or shrinks by k positions at a time to its left and by one position to its right.

Intermediate area. The intermediate area stores the intermediate nodes in k^2-lists, which are at most $3k+1$ in number as each intermediate node stores from k to $4k$ bucket chunks. The area is divided into zone $\mathscr{H}'$ and zone $\mathscr{A}'$, whose starting positions are encoded in the preamble $\mathscr{P}$.

Zone $\mathscr{A}'$ stores the full allocation units of the k^2-lists, using absolute pointers for addressing. The first allocation unit is actually split in two parts, at the beginning and at the end of zone $\mathscr{A}'$, so as to allow quick sliding of zone $\mathscr{A}'$ by k positions to the left or to the right. Any such sliding costs $O(k)$ time, plus the cost of searching $O(1)$ keys, whenever needed, to identify and relocate the $O(1)$ pointers entering the first allocation unit. The other units in zone $\mathscr{A}'$ are in no particular order.

Zone $\mathscr{H}'$ precedes zone $\mathscr{A}'$ and stores the heads of the k^2-lists. It contains at most $3k+1$ heads, each containing $O(k)$ bucket chunks (i.e., $O(k^2)$ keys per head). The absolute pointers to these heads are encoded in the first (or second) allocation unit of zone $\mathscr{A}'$. In general, the pointers entering zone $\mathscr{H}'$ from outside that zone are absolute, while the pointers having origin and destination inside zone $\mathscr{H}'$ are relative to the first position of zone $\mathscr{H}'$. Thus the relative pointers must be all re-encoded in $O(k^3)$ time plus $O(k)$ searches when sliding zone $\mathscr{H}'$.

Each head in zone $\mathscr{H}'$ contains an initial portion of an intermediate node, say, the first $s \leq k$ chunks in it. These s chunks are organized as a standard search tree of height $O(\log s) = O(\log k)$, where the relative pointers among chunks are encoded with $O(\log k)$ keys. Adding or removing chunks requires a complete in-place reconstruction of the search tree representing the head, in $O(k^2)$ time. In general, resizing a head can be handled in $O(k^2)$ time plus $O(k)$ searches, instead of $O(k^3)$ time, because we must relocate $O(k)$ other chunks in zone $\mathscr{H}'$, each of them requiring a search to identify its incoming pointer from outsize zone $\mathscr{H}'$.

Lemma 1 *In the intermediate area, zone $\mathscr{A}'$ can grow or shrink by k positions to its left or to its right in $O(k)$ time plus the cost of searching a key in the flat implicit tree. In zone $\mathscr{H}'$, sliding takes $O(k^3)$ time plus $O(k)$ searches. Moreover, changing the size of a head in zone $\mathscr{H}'$ requires $O(k^2)$ time plus $O(k)$ searches.*

Routing a key x in an intermediate node u is a standard binary search on each of its $O(1)$ allocation units, in $O(\log k)$ time plus the $O(k)$ cost of decoding the pointer to a child leaf, unless u is the first allocation unit in zone $\mathscr{A}'$ or u is partially stored in a head of zone $\mathscr{H}'$. In the former case, we must search also in the last positions of zone $\mathscr{A}'$. In the latter case, we decode an absolute pointer of size $O(k)$ in u (its part in zone $\mathscr{A}'$) to identify the search tree in zone $\mathscr{H}'$ representing its head. We perform a search in that tree, decoding $O(\log k)$ relative pointers of size $O(\log k)$ along a root-to-leaf path and running an $O(\log k)$-time binary search in each traversed node in the path, for a contribution of $O(\log^2 k) = o(k)$ time. After that, we need $O(k)$ time to decode an absolute pointer to the proper leaf of the intermediate node.

Lemma 2 *Routing a key in an intermediate node takes $O(k)$ time to identify the correct leaf to branch.*

Leaf area. The leaf area stores the leaves in k-lists and ℓ-lists. To make things simpler, we require $\ell = d\lceil \log\log n' \rceil$ to be an even integer for a positive constant d, such that we can always choose $k = d\lceil \log n' \rceil + \ell'$ $(0 \leq \ell' < \ell)$ to be a multiple of ℓ. Constant d is sufficiently large to let the keys encode the $O(1)$ integers and pointers needed per chunk. Since each leaf may contain from k to $5k-1$ keys, the k-lists are at most $4k$ in number. The area is divided in zone $\mathscr{H}$ and zone $\mathscr{A}$, with starting positions encoded in the preamble $\mathscr{P}$.

Zone $\mathscr{A}$ contains the full allocation units of the k-lists, being handled as zone $\mathscr{A}'$ in the intermediate area, except that here the allocation units are of size k instead of k^2. Zone $\mathscr{A}$ grows or shrinks by up to k positions to its left and one position to its right, in $O(k)$ time plus the cost of a search to relocate an incoming pointer.

Zone $\mathscr{H}$ precedes zone $\mathscr{A}$ and contains the $O(k)$ heads of the k-lists, each head storing at most k keys. In zone $\mathscr{H}$ we do not store each head as is, but we split it into $O(k/\ell)$ allocation units of size ℓ, maintaining them in a simple list linked with relative pointers. Since each such pointer requires $O(\log k)$ keys, we can encode $O(1)$ relative pointers in the ℓ keys of each allocation unit. Hence, we represent each head as this list plus a "remainder" of less than ℓ keys. The starting positions of each simple list and its remainder (per head) are encoded as relative pointers inside zone $\mathscr{H}$; however, they are stored in zone $\mathscr{A}$, even though the pointers are relative to zone $\mathscr{H}$. The starting positions of the simple list and the remainder for any given head can be retrieved in $O(\ell)$ time by simple offsetting and decoding.

All the $O(k)$ remainders thus produced contribute to a total of $O(k\ell)$ keys, and they are recursively stored as ℓ-lists, which are less than ℓ in number. Their heads (called *mini-heads*) contain $O(\ell^2)$ keys in total. Because of this, zone $\mathscr{H}$ is divided into zones $\mathscr{H}_1$, $\mathscr{H}_2$ and $\mathscr{H}_3$:

- Zone $\mathscr{H}_1$ contains the simple lists for the heads, without the "remainders." The allocation units (of size ℓ) in these lists are stored in no particular order. This zone grows or shrinks by k positions to its left and by multiples of ℓ positions to its right, taking $O(k)$ time plus the cost of a search to relocate an incoming pointer.
- Zone $\mathscr{H}_2$ contains the allocation units (but not the mini-heads) of the ℓ-lists storing the "remainders," in no particular order. It grows or shrinks by multiples of ℓ positions to its left and to its right, taking time proportional to the number of positions plus the cost of a search to relocate an incoming pointer from outsize zone $\mathscr{H}$.
- Zone $\mathscr{H}_3$ contains the mini-heads of the ℓ-lists in zone $\mathscr{H}_2$. The number of keys in zone $\mathscr{H}_3$ is at most $\ell^2 = O(\log^2 k)$. Hence, they can be fully scanned each time. This zone grows or shrinks by a multiple of ℓ positions to its left and by up to k positions to its right, taking $O(k)$ time.

Note that all the relative pointers refer to the starting position of zone $\mathscr{H}$. Sliding the whole zone $\mathscr{H}$ by k positions to its left or to its right takes $O(k^2)$ time for re-encoding all of the relative pointers in it. Moreover, given a pointer to a leaf and its size, we can infer whether the leaf is partially stored in a head. The keys in the leaf can be potentially stored in the four zones of the leaf area, as illustrated in Figure 2. However, retrieving the keys for that leaf in $O(k)$ time is not much of a problem.

Lemma 3 *In the leaf area, zone $\mathscr{A}$ can grow or shrink by up to k positions to its left or by one position to its right in $O(k)$ time plus the cost of searching a key in the flat implicit tree. Zone $\mathscr{H}$ can grow or shrink by k positions to its left and by up to k positions to its right at the same cost.*

Searching a key x in a leaf v is a standard binary search on each of its $O(1)$ allocation units in $O(\log k)$ time, with minor modifications when v occupies the first allocation unit in zone $\mathscr{A}$ or v is partially stored in a head of zone $\mathscr{H}$, as shown in Figure 2. The total cost is $O(k + \ell^2) = O(k)$ time.

Lemma 4 *Searching a key in a leaf takes $O(k)$ time.*

Bucket maintainance. We show how to maintain the buckets in the bottom layer by inserting or deleting individual keys in $O(\log n)$ amortized time, still supporting searches in $O(\log n)$ time per key by Lemma 2 and Lemma 4. We trace the worst behavior in terms of running time for an update operation, referring to Figures 1–2. Any update operation just performs a subset of the operations presented here.

Suppose we want to insert a key x into a bucket with intermediate node u, having routed x to the leaf v, child of u, in $O(k)$ time according to Lemma 2 and Lemma 4. (If the position of x is inside a chunk c of u, we insert x into c, set x to be the rightmost key in c, and then relocate x by recursively inserting x into v.)

1. Let $\mathscr{L}$ be the k-list that contains v, and let s be the common size of the leaves in $\mathscr{L}$. We swap v with the leaf (partially) stored in the head of $\mathscr{L}$ (if v is not already in the head). Now, v is partially stored in a head, and so its layout is that shown in Figure 2.
2. We have to add x to v, moving v from the head of $\mathscr{L}$ to the head of the k-list for the leaves of size $s+1$. We slide zone $\mathscr{A}$ of one position to the right, creating room in zone $\mathscr{H}_3$ for hosting x (Fig. 2a).
3. We may shrink zone $\mathscr{H}_3$ to its left and enlarge zone $\mathscr{H}_2$ to its right for handling the ℓ-list for x (Fig. 2b). When $s+1$ is a multiple of ℓ, we shorten zone $\mathscr{H}_2$ to its left and extend the ℓ-list in zone $\mathscr{H}_1$ (Fig. 2c).
4. If the ℓ-list in zone $\mathscr{H}_1$ contains k keys, we collect all the keys in it and form a chunk that is stored at the end of zone $\mathscr{H}$. We extend zone $\mathscr{A}$ to the left for including that chunk (Fig. 2d).
5. If the size of leaf v becomes $5k$ (i.e., 5 chunks), we have to split v into two leaves v' and v'' and a median chunk c. (The keys in v are all contained in 5 allocation units of zone $\mathscr{A}$.)

6. We insert c into u to reflect the split of v, sliding zone $\mathscr{H}$ and re-encoding the relative pointers. We also slide zone $\mathscr{A}'$, and change k^2-list for u analogously to what done for v in step 2. (See Fig. 1)
7. If the size of u is $4k^2 + k$, we split it into three parts u_L, m and u_R, where m is the median chunk. We create a new bucket with root m and child u_R (including its descending leaves), sliding zone $\mathscr{H}'$ to enlarge the super-root area by k positions, while u is replaced by u_L and its remaining leaves.

Suppose we want to delete x in a bucket with intermediate node u, having reached a leaf v. (If x belongs to a chunk c of u, we delete it, set x to be the leftmost key in v, add x to c, and recursively remove x from v.) We execute the delete operation with steps similar to steps 1–7, except that we merge or share instead of splitting u and v; in particular, sharing in v involves an individual key whereas in u it involves an individual chunk. More details will be reported in the full version.

Theorem 2 *In the bottom layer, each bucket contains $\Theta(k^2)$ keys at any time. Performing an update by inserting or deleting a key in a bucket takes $O(k)$ amortized time. Any newly created bucket will merge or split after further $\Omega(k^2)$ update operations inside that bucket. At any time, only $O(1)$ RAM registers are required to operate dynamically.*

4 Top Layer: The Super-Root

The top layer contains the super-root of the flat implicit tree and collects all the actual chunks and the virtual chunks. We remark that these chunks are the roots of the buckets in the bottom layer discussed in Section 3. The memory layout in the super-root area is simple; first, all the actual chunks in sorted order and, then, all the virtual chunks in no particular order.

Actual chunks and virtual chunks. The a actual chunks are stored in sorted order in the first ak positions of the super-root area, where $a = O(n/k^2)$ is *always* a power of two. Each actual chunk has at most $\alpha = O(1)$ virtual associated with it, which are the nearest in the order of the (actual and virtual) chunks. They are kept in a linked (sorted) list starting from the actual chunk. The rest of the area contains the virtual chunks in no particular order as the linked lists allow their retrieval. The super-root area resizes by k positions to the right at a time to make room for one more or less chunk after bucket splitting or merging. The number a of actual chunks changes only when rebuilding (see Section 5) or when performing a full redistribution of actual and virtual chunks. Since the actual chunks are kept sorted, we can route a key to its (actual or virtual) chunk in the top layer in $O(\log n)$ time.

Lemma 5 *Routing a key in the super-root takes $O(\log n)$ time to identify the correct bucket to access.*

Density functionalities. We partition the actual chunks into $a/\hat{k}$ segments of $\hat{k}$ adjacent chunks each, where $\hat{k}$ is the power of two such that $\hat{k} \leq k < 2\hat{k}$, obtaining a sorted array R of $O(a/\hat{k}) = O(n/k^3)$ segments. The segments in R are the leaves of an implicit complete binary tree of height $h = O(\log(n/k^3))$, and an internal node u represents a portion of consecutive segments in R. If u is at depth s, then its portion in R contains $2^{h-s}\hat{k}$ actual chunks. There is a static mapping between the internal nodes and the segments of R (except the last). In other words, each segment is associated with a leaf and an internal node; the last segment is associated with the rightmost leaf. In the following, when we refer to the nodes and the leaves of R we mean those in the implicit tree defined above.

The number of actual chunks and that of virtual chunks are related each other using R. For this, we define the density of virtual chunks with respect to actual chunks [3, 6, 15]. In our case, we have the further requirements of avoiding to produce empty slots and of distributing virtual chunks among the actual chunks without violating their order (i.e., we cannot simply move one virtual from a position to another of R). That allows us to maintain virtual chunks almost uniformly distributed in R, with at most α virtual per actual chunk. Let $act(u)$ denote the actual chunks descending from u. As previously mentioned, if u has depth $s \leq h$, then $|act(u)| = 2^{h-s}\hat{k}$. Let $vir(u)$ denote the virtual chunks associated with the actual chunks in $act(u)$, where $|vir(u)| \leq \alpha|act(u)|$. We encode the value of $|vir(u)|$ in the segment of R associated with u. We define the *density* of u as $d(u) = \frac{|vir(u)|}{\alpha|act(u)|}$.

The density of all nodes is constrained according to their depth and to some positive integer constants α and positive real constants ρ_0 and τ_0. In order to amortize the cost of updating, we require that the constants satisfy the relation $2\alpha\rho_0 + 1 < \alpha\tau_0$ for $\alpha \geq 2$ and $0 < \rho_0 < \tau_0 < 1$ (e.g., $\alpha = 2$, $\rho_0 = 0.1$, and $\tau_0 = 0.9$). We also need two further constants ρ_∞ and τ_∞, such that $0 < \rho_\infty < \rho_0 < \tau_0 < \tau_\infty < 1$.

Let's define $\rho_s = \rho_0 - \frac{s}{h}(\rho_0 - \rho_\infty)$ and $\tau_s = \tau_0 + \frac{s}{h}(\tau_\infty - \tau_0)$ (note that $\rho_h = \rho_\infty$ and $\tau_h = \tau_\infty$). The density of a node u at depth s must satisfy $\rho_s \leq d(u) \leq \tau_s$. We say that a leaf u *overflows* if $d(u) > \tau_h$ and *underflows* if $d(u) < \rho_h$. An internal node u at depth s overflows (resp., underflow) if $d(u) > \tau_s$ (resp., $d(u) > \rho_s$) and recursively at least one of its children either overflows or underflows [3, 6, 15].

Rebalancing/redistribution of chunks. The association of virtual chunks with actual chunks is dynamically maintained when bucket splitting creates new chunks and bucket merging removes chunks. As long as we can keep at most α virtual chunks associated with each actual chunk, we do not need to redistribute chunks. However, when removing an actual chunk that has no virtual chunk associated with it, or when creating a new chunk from a chunk in a maximal list of size α, we have to redistribute the chunks by reclassifying them as actual and virtual, while preserving their order.

The redistribution of chunks inside the segments (leaves) of R can operate sequentially on the whole segment in $O(k^2)$ time. For example, let's take two consecutive actual chunks a_i and a_{i+1} in a segment of R, with associated lists L_i and L_{i+1} of virtual chunks, each list containing at most α entries. Suppose that, after a bucket splitting, L_i becomes of size $\alpha + 1$. The rightmost virtual chunk in L_i, say c, replaces a_{i+1}. Specifically, c becomes actual occupying the positions of a_{i+1} in R, and a_{i+1} becomes virtual occupying the positions of c. Moreover, a_{i+1} is prepended to list L_{i+1}, which becomes associated with c. If also L_{i+1} becomes of size $\alpha + 1$, we go on iteratively with the next actual chunk a_{i+2} until we reach the end of the segment. Alternatively, we can consider a_{i-1} and a_i and proceed analogously towards the beginning of R. Handling the removal of an actual chunk after bucket merging can be treated in the same fashion.

When the density of a segment reaches either its maximum or its minimum value, we run the redistribution of chunks in an internal node u of R as described next. We work under the hypothesis that u is the deepest ancestor of the segment overflowing or underflowing, such that at least one of the children of u overflows or underflows whereas u does not, that is, its density is $\rho_s \leq d(u) \leq \tau_s$, where s is the depth of u. We now show how to redistribute the actual and virtual chunks in the portion of R corresponding to u, denoting by $u[i]$ the ith actual chunk in that portion, for $1 \leq i \leq |act(u)| = 2^{h-s}\hat{k}$. Although we change the status of some actual and virtual chunks, thus changing the chunks in sets $act(u)$ and $vir(u)$, we preserve their size $|act(u)|$ and $|vir(u)|$ (hence, the density of u). The redistribution for u is in-place to preserves implicitness at any time and runs in two phases of $O(k^2)$ time.

In the first phase, we scan the actual chunks of u in decreasing order. We employ a linked list L, which is initially empty, and it incrementally contains virtual chunks. For $i = 2^{h-s}\hat{k}, \ldots, 2, 1$, we execute the following steps:

1. If the actual chunk in $u[i]$ has virtual chunks associated with it, we append them to the end of L in decreasing order (no chunk relocation is needed, just re-encoding of pointers).
2. The chunk at the beginning of L replaces the actual chunk in $u[i]$, which is appended to the end of L (here we switch an actual chunk with a virtual chunk).

At the end of the first phase, list L contains the $|vir(u)|$ smallest chunks in u, while u stores the $|act(u)|$ largest chunks in its portion of R.

In the second phase, we scan all the chunks in u and distribute $|vir(u)|/|act(u)|$ virtual chunks per actual chunk. We employ the list L computed in the first phase, executing the following steps for $i = 1, 2, \ldots, 2^{h-s}\hat{k}$:

1. The chunk at the beginning of L replaces the chunk in $u[i]$, which is appended to the end of L.
2. The next $|vir(u)|/|act(u)|$ chunks are removed from L to create the linked list of virtual chunks associated with $u[i]$.

At the end of the second phase the virtual chunks are uniformly distributed among the actual chunks by preserving their order. The density $d(u)$ of u does not change, whereas the density of any descendant v of u is $d(v) = d(u)$. Hence, it satisfies $\rho_t \leq d(v) \leq \tau_t$, where $t > s$ is the depth of v, since $\rho_t < \rho_s < \tau_s < \tau_t$.

A special case of redistribution is in the root u of R. The two-phase algorithm is quite similar to what described so far, except that the number a of actual chunks in R doubles (root overflow) or halves (root underflow) in the second phase. This implies that some virtual chunks becomes actual, or vice versa. Note that the size of the super-root area in the memory layout does not change. Consequently we run a mere reorganization inside the super-root area, and here is why.

When the root u overflows, we have $d(u) > \tau_0$ and so $|vir(u)| > \alpha\tau_0|act(u)|$. In the second phase, we transform $a = |act(u)|$ virtual chunk in L into actual chunks, so that R contains $2a$ actual chunks. After the distribution, the new density for u is $d(u) > \frac{\alpha\tau_0 - 1}{2\alpha} > \rho_0$, since we fix our constants satisfying $2\alpha\rho_0 + 1 < \alpha\tau_0$. The new density also verifies $d(u) < \tau_0$, since we double the number of actual chunks while decreasing the number of virtual chunks.

When u underflows, we have $d(u) < \rho_0$ and so $|vir(u)| < \alpha\rho_0|act(u)|$. In the second phase, we transform $(1/2)a = (1/2)|act(u)|$ actual chunk in R into virtual chunks in L, so that R remains with $(1/2)a$ actual chunks. After the distribution, the new density for u is $d(u) < \frac{2\alpha\rho_0 + 1}{\alpha} < \tau_0$ as $2\alpha\rho_0 + 1 < \alpha\tau_0$. The new density also verifies $d(u) > \rho_0$, since we halve the number of actual chunks while increasing the number of virtual chunks.

In both cases, we re-encode the new values of $|vir(u)|$ in the suitable chunks of R. Since the height h of the implicit tree changes, we encode this information in the preamble $\mathscr{P}$ of the memory layout. This helps us to define the new values of ρ_s and τ_s for the several levels s on the fly. Since the density of the resulting root u of R satisfies $\rho_0 < d(u) < \tau_0$ after the redistribution, all the descendants v of u have density $d(v)$ satisfying $\rho_s < d(v) < \tau_s$, where s is the depth of v, since $\rho_0 < d(v) < \tau_0$ and $\rho_s < \rho_0 < \tau_0 < \tau_s$.

Analysis. The analysis of the amortized cost for an internal node v on depth s is a variation of that given in [3]. The cost for rebalancing v is $O((|act(u)| + |vir(u)|) \cdot k)$ time, where u is the parent of v. In order for v to overflow again, we need at least additional $\alpha|act(v)|(\tau_{s+1} - \tau_s)\Theta(k^2)$ insertions in the buckets associated with the actual and virtual chunks in the portion of R corresponding to v, and thus the amortized cost is

$$\frac{(|act(u)| + |vir(u)|)\, O(k)}{\alpha|act(v)|(\tau_{s+1} - \tau_s)\Theta(k^2)} = \frac{(|act(u)| + |vir(u)|)\, h}{\alpha|act(v)|(\tau_\infty - \tau_0)\Theta(k)} \le \frac{(\alpha+1)|act(u)|h}{\alpha|act(v)|(\tau_\infty - \tau_0)\Theta(k)} = O(h/k) = O(1),$$

noting that $|act(u)| = 2|act(v)|$. A similar analysis holds for deletions. In order for v to underflow again, we need at least additional $\alpha|act(v)|(\rho_{s+1} - \rho_s)\Theta(k^2)$ deletions in the buckets associated with the actual and virtual chunks in the portion of R corresponding to v, and thus the amortized cost per operation on v is

$$\frac{(|act(u)| + |vir(u)|)\, O(k)}{\alpha|act(v)|(\rho_{s+1} - \rho_s)\Theta(k^2)} = \frac{(|act(u)| + |vir(u)|)\, h}{\alpha|act(v)|(\rho_0 - \rho_\infty)\Theta(k)} \le \frac{(\alpha+1)|act(u)|h}{\alpha|act(v)|(\rho_0 - \rho_\infty)\Theta(k)} = O(h/k) = O(1),$$

Since there are h levels, each update operation has an amortized cost of $O(h^2/k) = O(\log n)$ for the whole maintainance of R and the super-root. We assume that the root u of R satisfies $\rho_0 < d(u) < \tau_0$, which is guaranteed initially and, later, by each redistribution of chunks in the root as previously discussed.

Theorem 3 *In the top layer, routing a key to its bucket takes $O(\log n)$ time. Inserting or deleting a chunk in any position of the super-root has an amortized cost of $O(h^2/k) = O(\log n)$ time, where $h = O(\log n)$. At any time, only $O(1)$ RAM registers are required to operate dynamically.*

5 Rebuilding and Final Analysis

We described the flat implicit tree for n distinct keys assuming that $n'/4 < n < n'$ at any time, where n' is a power of two. That value of n' is important to fix the parameters k and ℓ, as discussed in Section 3. Note that the time complexity of our algorithms is parametric in k and ℓ, and that we fixed $k = \Theta(\log n')$ and $\ell = \Theta(\log\log n')$ to get our claimed bounds. To preserve the invariant on $n'/4 < n < n'$ when $n = n'$, we double n', update the values of k and ℓ, and rebuild our data structure as explained below. Analogously, when $n = n'/4$ we halve n', update the values of k and ℓ, and rebuild. After rebuilding, we have $n = n'/2$ for the new value of n', and so our invariant is maintained with n half on the way between $n'/4$ and n'.

Rebuilding is performed by a sequence of $O(n)$ insertions, with the only difference that we fix $k = \Theta(\log n')$ and $\ell = \Theta(\log\log n')$ even if we may have re-inserted less than $n'/4$ keys during the rebuilding (this is important to avoid triggering a sequence of recursive rebuildings). Specifically, we take the first $\Theta(k^3)$ keys and build a single segment of buckets with those keys by brute force (for each key to insert, we shift one by one all the keys if needed). Consequently, the running time for the first segment is $O(k^6) = o(n)$. For the rest of the keys, we re-insert them one at time using our algorithms with the *fixed* value of $k = \Theta(\log n')$, requiring a total of $O(n\log n)$ time for the reconstruction.

We are now ready to prove our main result, Theorem 1. We use credits with the accounting method, with three accounts BUCKET (for buckets), SUPERROOT (for rebalancing the super-root) and CHUNKRESIZE (for rebuilding when n' and k changes). Each update deposits $O(\log n)$ credits in each account before execution. We sketch the amortized analysis for the cost of an insertion. Apart from the $O(\log n)$ search cost required by an update, the insertion of a key x withdraws credits as follows.

From the account BUCKET, we use $O(\log n)$ credits for inserting x in the suitable leaf v; for splitting v and increasing the size of u (the intermediate node that is parent of v); for splitting u, thus creating two buckets from the current bucket; finally, for sharing, namely, giving one leaf and the root to a neighbor bucket with $k-1$ leaves. From the account SUPERROOT, we use $O(\log n)$ credits for redistributing the chunks of the segment containing x's bucket, when that bucket is split; for redistributing chunks in R (with $O(1)$ credits per level of the virtual tree in R). From the account CHUNKRESIZE, we use $O(\log n)$ credits for only the next reconstruction due to a change in n' and so in k.

The analysis for a deletion is similar, except that we do not withdraw credits for sharing as they are paid by the insertions in the bucket borrowing the chunk to its sibling. This completes the proof of Theorem 1.

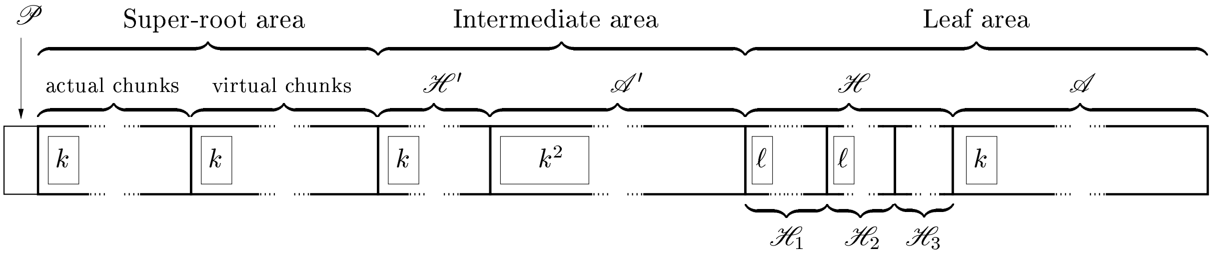

Figure 1: Memory layout of the flat implicit tree.

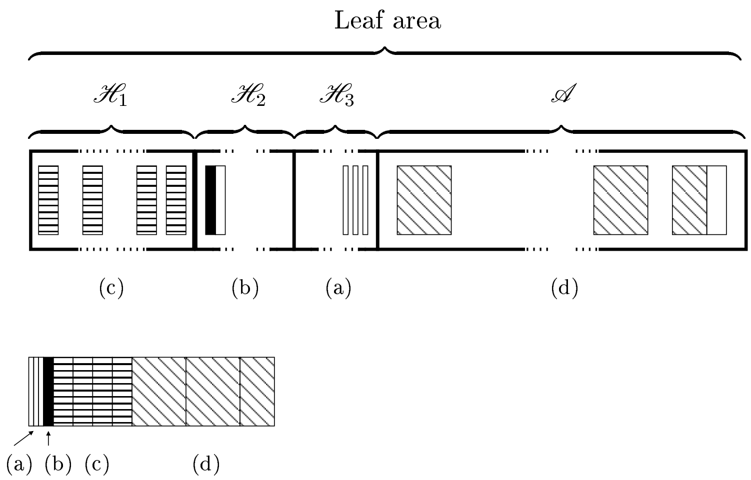

Figure 2: A leaf v (bottom) and its layout in the leaf area (top). Parts (a)–(d) of v are stored, respectively, in zones $\mathscr{H}_3$, $\mathscr{H}_2$, $\mathscr{H}_1$ and $\mathscr{A}$ of the leaf area shown in Figure 1.

References

[1] G. M. Adel'son-Vel'skii and E. M. Landis. An algorithm for the organization of information. *Soviet Mathematics Doklady*, 3:1259–1263, 1962.

[2] Alfred V. Aho, John E. Hopcroft, and J. D. Ullman. *The Design and Analysis of Computer Algorithms*. Addison-Wesley, Reading, 1974.

[3] M. A. Bender, E. D. Demaine, and M. Farach-Colton. Cache-oblivious b-trees. In IEEE, editor, *41st Annual Symposium on Foundations of Computer Science*, Redondo Beach, CA, pages 399–409, USA, IEEE, 2000.

[4] J. L. Bentley, D. Detig, L. Guibas, and J. B. Saxe. An optimal data structure for dynamic member searching. *(Unpublished notes)*, pages 1–10, Spring 1978.

[5] Allan Borodin, Faith E. Fich, Friedhelm Meyer auf der Heide, Eli Upfal, and Avi Wigderson. A tradeoff between search and update time for the implicit dictionary problem. *Theoretical Computer Science*, 58(1-3):57–68, 1988.

[6] Gerth Stølting Brodal, Rolf Fagerberg, and Riko Jacob. Cache-oblivious search trees via trees of small height. In *Proc. 13th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 39–48, 2002.

[7] Andrej Brodnik and J. Ian Munro. Membership in constant time and almost-minimum space. *SIAM Journal on Computing*, 28(5):1627–1640, 1999.

[8] Amos Fiat, Moni Naor, Jeanette P. Schmidt, and Alan Siegel. Nonoblivious hashing. *Journal of the ACM*, 39(4):764–782, October 1992.

[9] Faith E. Fich and Peter Bro Miltersen. Tables should be sorted (on random access machines). In *Algorithms and Data Structures, 4th International Workshop*, LNCS 955, pages 482–493, Ontario, Canada.

[10] Robert W. Floyd. Algorithm 245 (TREESORT). *Communications of the ACM*, 7:701, 1964.

[11] Gianni Franceschini and Roberto Grossi. Implicit dictionaries supporting searches and amortized updates in $O(\log n \log\log n)$. In *Proc. the 14th Annual ACM-SIAM Symposium on Discrete Algorithms*, to apper, 2003.

[12] Gianni Franceschini, Roberto Grossi, J. Ian Munro, and Linda Pagli. Implicit B-trees: New results for the dictionary problem. In *IEEE Symposium on Foundations of Computer Science (FOCS)*, 2002.

[13] Greg N. Frederickson. Implicit data structures for the dictionary problem. *J. ACM*, 30(1):80–94, 1983.

[14] Michael L. Fredman, János Komlós, and Endre Szemerédi. Storing a sparse table with $O(1)$ worst case access time. *J. ACM*, 31(3):538–544, 1984.

[15] Alon Itai, Alan G. Konheim, and Michael Rodeh. A sparse table implementation of priority queues. In *Automata, Languages and Programming, 8th Colloquium*, LNCS 115, pages 417–431, Israel, 1981.

[16] D. E. Knuth. *The Art of Computer Programming III: Sorting and Searching.* Addison–Wesley, Reading, Massachusetts, 1973.

[17] Conrado Martínez and Salvador Roura. Randomized binary search trees. *Journal of the ACM*, 45(2):288–323, March 1998.

[18] J. Ian Munro. An implicit data structure supporting insertion, deletion, and search in $O(\log^2 n)$ time. *Journal of Computer and System Sciences*, 33(1):66–74, 1986.

[19] J. Ian Munro and Patricio V. Poblete. Searchability in merging and implicit data structures. *BIT*, 27(3):324–329, 1987.

[20] J. Ian Munro and Hendra Suwanda. Implicit data structures for fast search and update. *Journal of Computer and System Sciences*, 21(2):236–250, 1980.

[21] Rasmus Pagh. Low redundancy in static dictionaries with constant query time. *SIAM Journal on Computing*, 31(2):353–363, 2002.

[22] Jaikumar Radhakrishnan and Venkatesh Raman. A tradeoff between search and update in dictionaries. *Information Processing Letters*, 80(5):243–247, 2001.

[23] Rajeev Raman, Venkatesh Raman, and S. Srinivasa Rao. Succinct indexable dictionaries with applications to encoding k-ary trees and multisets. In *ACM-SIAM Symposium on Discrete Algorithms*, pages 233–242, 2002.

[24] Peter Sanders. Personal communication (ALGO'02), 2002.

[25] Daniel Dominic Sleator and Robert Endre Tarjan. Self-adjusting binary search trees. *Journal of the ACM*, 32(3):652–686, July 1985.

[26] J. W. J. Williams. Algorithm 232: Heapsort. *Communications of the ACM*, 7:347–348, 1964.

[27] Andrew C. Yao. Should tables be sorted? *J. ACM*, 31:245–281, 1984.