Optimal Implicit Dictionaries over Unbounded Universes *

Gianni Franceschini francesc@di.unipi.it

Roberto Grossi grossi@di.unipi.it

February 27, 2005

Abstract

An array of n distinct keys can be sorted for logarithmic searching or can be organized as a heap for logarithmic updating, but it is unclear how to attain logarithmic time for both searching *and* updating. This natural question dates back to the heap of Williams and Floyd in the sixties and relates to the fundamental issue whether additional space besides those for the keys gives more computational power in dictionaries and how data ordering helps. *Implicit* data structures have been introduced in the eighties with this goal, providing the best bound of $O(\log^2 n)$ time, until a recent result showing $O(\log^2 n/\log\log n)$ time. In this paper we describe the flat implicit tree, which is the first data structure obtaining $O(\log n)$ time for search and (amortized) update using an array of n cells.

Keywords: data structures, searching, dictionary problem, implicitness.

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 cell capable of containing one key. The array can be sorted so that binary search takes $O(\log n)$ time (Knuth [19] credits Mauchly (1946) for inventing this organization of the keys). Also, the keys in the array can be permuted in heap order as shown by 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 [29, 12]. 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 $x' \geq x$. Heap-ordered arrays have the drawback of requiring $O(n)$

*Dipartimento di Informatica, Università di Pisa, Largo Bruno Pontecorvo 3, 56127 Pisa, Italy. Some of the ideas described in this paper were presented at ACM-SIAM SODA 2003 [13] and ICALP 2003 [14]. The work done in the current version of the paper is partially supported by the Italian MIUR project PRIN ALGO-NEXT.

time for searching, 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 a permutation of the keys in the array of n cells combining the best qualities of sorted arrays and heaps, so that each operation requires $O(\log n)$ time.

Both sorted arrays and heap-ordered arrays 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 an array of n memory cells that is extendible to the right, one cell at a time, and that stores n distinct keys $a_1, a_2, \ldots, a_n$ suitably permuted, for any $n > 1$, where each key a_i occupies a distinct cell of the array. The array is allocated in a contiguous segment of n adjacent memory cells, plus $O(1)$ cells for bookkeeping. All what is known is the starting position of the memory segment hosting the array, as the rest of the information is implicitly encoded by the permutation of $a_1, a_2, \ldots, a_n$. The memory segment can be enlarged or shortened to the right in constant time, one cell at a time, to dynamically extend or shrink the array to the right. Inserting a new key a_{n+1} extends the array by one cell storing a_{n+1}, and shuffles $a_1, a_2, \ldots, a_n, a_{n+1}$ accordingly to encode the resulting data structure in $n+1$ cells. Deleting key a_i yields an array of $n-1$ cells producing a new permutation of $a_1, a_2, \ldots, a_{i-1}, a_{i+1}, \ldots, a_n$.

The 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. Nonetheless, no known implicit data structure for the dictionary problem takes $O(\log n)$ time per operation. 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 $\Omega(n)$ extra pointers and integers.

Previous work on implicit dictionaries. Munro and Suwanda [23] 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 [19]. 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.* [6] supporting only insertions and searches. While the term "implicit" originated in [23], it has also been the subject of papers taking a somewhat different point of view, including a long lists of results in perfect hashing [17, 10], bounded-universe dictionaries [9, 25], and cache-oblivious data structures [8]. These results were obtained for less stringent models different from the model adopted by Munro and Suwanda and in following papers.

Yao [30] examined the special case of keys belonging to a *bounded* universe U, so that each key can be interpreted as an integer ranging in $0 \ldots |U| - 1$ 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 cells, searching requires $\Omega(\log n)$ time for sufficiently large U. However, he showed that encoding some information in *one extra* cell 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 [17] 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 $n+\omega(1)$ cells of space. Recent improvements in this direction are described in Fich and Miltersen [11] and Raman, Raman and Rao [27], encoding the keys in at most n cells of space by avoiding to store explicitly 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 [16] 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 $O(\log n)$-time searchable at any time of its execution, Munro and Poblete [22] 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 [21] 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.* [15] 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, thus leaving open the $O(\log n)$ bound.

Borodin et al. [7] and Radhakrishnan and Raman [26] 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 result. The implicit dictionary problem is intimately related to the fundamental question whether using *extra* space gives more computational power than just using $n+O(1)$ cells [1], and how data ordering can help in this task. The ultimate goal is in the flavor of the theoretical result of Yao [30], 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

[1] Strictly speaking, $n+O(1)$ cells can be further reduced to just n cells by encoding the bits in the $O(1)$ extra cells in the first $O(\log n)$ keys of the implicit data structure.

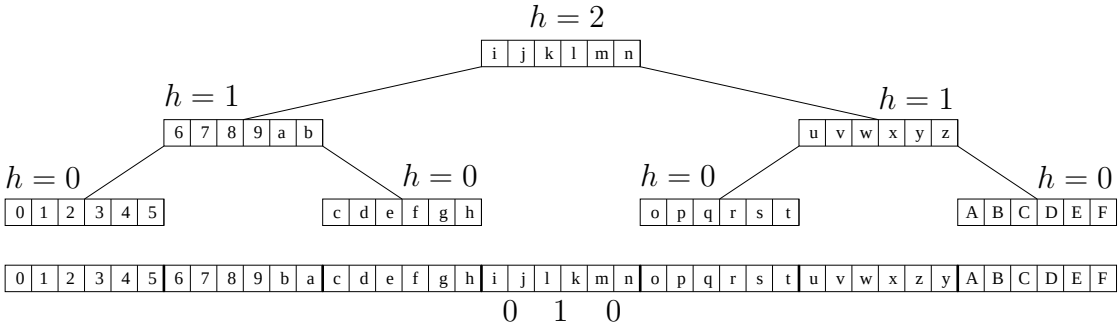

Figure 1:

occupy strictly more than $n + O(1)$ cells?

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, being implicit, is that it achieves the best space saving possible within optimal time bounds. In this context self-adjusting search trees [28] and random search trees [20] 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 $n + \Omega(n)$ cells of memory (for the pointers).

We introduce the issue of amortization in implicit data structures noting that several data structuring techniques requiring the duplication of keys cannot work in the implicit model. For example, Overmars' rebuilding technique [24] maintains a copy of the data structure; Arge's buffering technique [3] should keep a copy of the key to be deleted or a pointer to it. Moreover, during the amortization, we can use only in-place algorithms as otherwise the data structures would not require $O(1)$ RAM registers to operate, but also a non-constant temporary area, which makes no sense in the implicit model. As a result, the reader may notice a greater level of details than in other papers on data structures. This is probably due to the fact that not only we have to describe the algorithms for the supported operations, but we have to carefully handle the memory layout of the keys in the array.

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 [21] to handle a large collection of relatively small buckets of permuted keys (the intermediate nodes and the leaves). We obtain a constant depth with $O(\log n)$ accessed keys per traversed node, which is the problem left open in previous work. For this, 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 leaves augmented with spare keys. We use techniques previously introduced in [5] and [15]. Our deletion algorithm is almost symmetrical to the insertion.

Paper organization. The paper is organized as follows. Section 2 gives an overview of the flat implicit tree organized in two layers, with Section 4 describing the bottom layer and Section 3 giving a description of the top layer. We put all together in Section 3.4 for our final bounds.

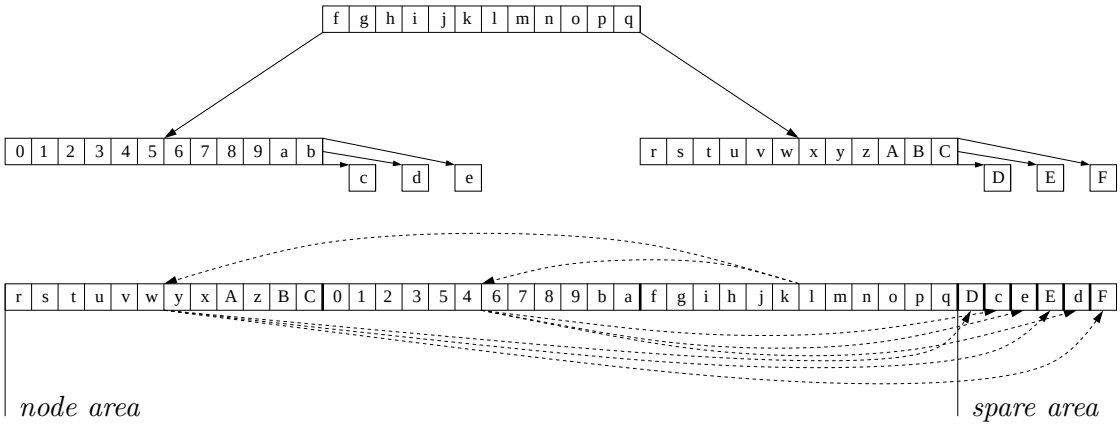

Figure 2:

2 Overview of the Flat Implicit Tree

Before giving the high-level description of our data structure, we review some basic ideas on implicit data structures by examples, so as to drive the reader into the computational flavor of the model.

2.1 A warm-up example

Let's consider first the simplest implicit data structure, a sorted array A of n entries. We can see A as a complete binary search tree in which, given a node at position i, we can easily compute the positions of its children by a mathematical rule applied to i. The nodes do not need to contain one key each. As shown in Figure 1, each node contains 6 keys.

Putting more keys in a node allows us to encode implicitly some information by pairwise (odd-even) permutation of keys [21]. We can use $2b$ distinct keys $x_1, y_1, x_2, y_2, \ldots, x_b, y_b$ to encode a pointer or an integer of b bits. Namely, we permute the keys 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 $\min\{x_i, y_i\}$. For example, in the tree of in Figure 1, each node stores its height h in $b = 3$ bits by using the 6 keys in it. For example, the root contains keys i, j, k, l, m, n, and its height is $h = 2$, which is 010 in binary. So, we need to swap the keys in the second pair and obtain the sequence i, j, l, k, m, n to encode h. It is worth noting that what we actually store is just the array in Figure 1, with some of its keys permuted. The array is quasi sorted, so that any algorithm (e.g., binary search) that requires a sorted array in input is able to run with minor modifications.

We can go a step further and let the tree be dynamic by encoding also the pointers to the children in a node, provided that a sufficient number of keys is hosted in the node. Let's consider the tree in Figure 2. The nodes contain now 12 keys each. The internal nodes encode the pointers to the children, giving a larger degree of freedom in handling them. The leaves encodes the pointers to the spare keys c, d, e, D, E, F, which do not fit in their leaves. Once again, we only store the array in Figure 2 with some entries permuted to encode the above structural information.

Unfortunately, the data structures in Figures 1 and 2 do not guarantee efficient time

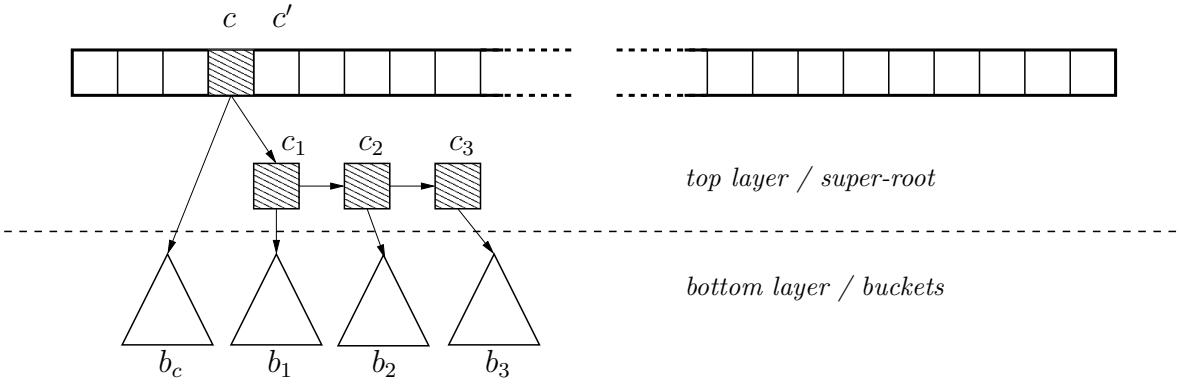

Figure 3: An actual chunk c with its associated bucket b_c of keys, where c is the root of b_c. The virtual chunks c_1, c_2, c_3 are associated with c, and their corresponding buckets are b_1, b_2, b_3 (where $c < b_c < c_1 < b_1 < \cdots < c_3 < b_3 < c'$). Note that the keys in c, c_1, c_2, c_3 will be stored in the root area of Figure 4, while the keys in b_c, b_1, b_2, b_3 will be suitably stored in the bucket area of Figure 4.

bounds for the supported operations. For example, deleting key F and inserting a key between c and d in Figure 2 causes a shift of most of the keys, as there are not enough keys in the leaves to encode 4 pointers to spare keys. Note that decoding just a single pointer requires $\Theta(\log n)$ time. We need to make the data structures a bit more flexible, thus introducing the flat implicit tree. In the following, when we talk of pointers and integers associated with the tree, we assume that their bits and the tree shape are suitably encoded by permuting keys as in the array of Figure 2. Also, the keys in the input are all distinct.

2.2 The flat implicit tree

From a high-level point of view, the flat implicit tree is a suitable collection of linked *chunks* [21, 15, 13] and *spare* keys. Each chunk contains k keys that are pairwise permuted for encoding a constant number of integers and pointers, each of $O(\log n)$ bits. Hence, we need to fix k for this purpose. Differently from previous work on implicit data structures, we keep the invariant that the chunk size is $k = \Theta(\log n')$, where n' is a power of two satisfying $n'/4 < n < n'$. The constant hidden in the notation for k is sufficiently large to encode all the structural information for an arbitrary node of the flat implicit tree. As it should be clear in the rest of the paper, we need to store a number of pointers and integers in each chunk that is independent of n, hence, a constant. Moreover, using n' for fixing k is important as it avoids to change k each time that n changes. As a result, the algorithms for maintaining our data structure are parametric in n' and k. We resize k only when either $n = n'/4$ or $n = n'$, and this event marks the beginning of a new *epoch*. The lifetime of the data structure can be divided into epochs. At the beginning of each epoch, we reconstruct our implicit dictionary in-place to guarantee that $n'/4 < n < n'$, as described in Section 3.

Our algorithms exploit the total order that can be inferred from the chunks. The keys in any chunk belong to a certain interval of values, and the chunks are pairwise disjoint when considered as intervals of keys. 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. We refer to chunks in sorted order meaning that the sequence of chunks satisfies the total order, whereas they can be permuted

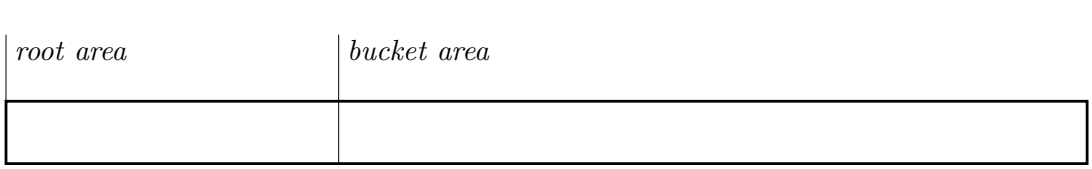

Figure 4:

internally to encode information.

Note that the keys are not necessarily all grouped into the chunks. Behind the fact that n is not necessarily a multiple of k, the dynamic maintance of our data structure needs some degree of flexibility for inserting and deleting keys. We call spare keys those keys not contained in the chunks. The spare keys are kept together in a contiguous segment of the input array, without a particular organization, and are accessed by individual pointers encoded somewhere in the chunks (like in Figure 2). The individual chunks compose the top layer in Figure 3; since they are the roots of the buckets in the bottom layer, thus forming the *super-root* of the flat implicit tree in the top layer of Figure 3, they are stored in the root area of Figure 4. Also the trees in the bottom layer of Figure 3 have nodes made up of groups of consecutive chunks, thus forming the buckets that are stored in the bucket area of Figure 4. However, they have overflow areas filled with spare keys, sharing the bucket area together with the chunks of the nodes. Consequently, we can draw a first picture of our data structure in two layers, as shown in Figure 3. The array currently storing the n keys is partitioned in two parts as shown in Figure 4, with the keys of the top layer suitably permuted and stored in the root area and those of the bottom layer kept permuted in the bucket area.

Top layer. The top layer contains the *super-root* of the flat implicit tree. It stores a sequence of $a = \Theta(n/b)$ contiguous chunks, called *actual* chunks, where a is a power of two and $b = \Omega(\log n)$ is a suitable parameter to be tweaked later on. The actual chunks are in sorted order and occupy the first ak positions of the root area. The rest of the root area hosts the *virtual* chunks, which are in no particular order. Each actual chunk c has associated at most $\alpha = O(1)$ virtual chunks $c_1, c_2, c_3 \ldots$, which are the nearest in the order of the chunks (i.e., for any other actual chunk $c' > c$, we have $c < c_1 < c_2 < c_3 < \cdots < c'$). Chunks $c, c_1, c_2, c_3 \ldots$ are kept in a linked (sorted) list whose first element is c. Note that the pointers for this list are encoded in the chunks themselves. Moreover, once we access c, we can scan $c_1, c_2, c_3 \ldots$ in its list by decoding the pointers from their scanned keys. Here is why we allow the virtual chunks to be kept in no particular order in the root area.

The root area may resize by k positions to its right to make room for one more or one less chunk. As a result, the number a of actual chunks changes only when rebuilding or when performing a full redistribution of actual and virtual chunks (Section 3).

With this organization, we can route any search key x to its proper chunk, so that the chunk contains x as an interval of keys, in $O(\log n)$ time. It suffices to run a simple variant of the binary search on the actual chunks and, once a chunk c is reached, a scan of its associated virtual chunks yields the outcome of the search.

Bottom layer. All the keys in our data structure are equally distributed, asymptotically speaking, among the chunks in the top layer to form buckets. The keys in each bucket belong to a certain interval of values, and the buckets are pairwise disjoint when considered as intervals. As a result, each chunk in the top layer is the root of a bucket containing $\Theta(b)$ keys.

Analogously to the chunks, we can define a total order on the buckets. All these buckets populate the bottom layer and they are stored in the bucket area of the array currently in use (see Figure 4). The bucket area may grow or shrink by k positions to its left and by one position to its right. Each bucket supports insertions, deletions and searches of individual keys. In addition, it must support split and merge operations. Splitting a bucket creates two smaller buckets b_1 and b_2, with $b_1 < b_2$, and a new chunk c such that $b_1 < c < b_2$ (by extending the total order to both chunks and buckets). Chunk c moves to the top layer, being associated with b_2. Bucket b_1 preserves its original root in the top layer. Merging takes two neighbor buckets b_1 and b_2 (with root c in the top layer and $b_1 < c < b_2$), and creates a single bucket containing the keys in b_1, c, b_2, whose root becomes that of b_1 in the top layer. At this stage, we do not give details on the structure of each bucket, delaying its discussion to Sections 3 and 4.

3 A Sub-Optimal Data Structure

In this section, we detail the top layer of our data structure. We also give a simple implementation of the bottom layer for yielding a sub-optimal implicit dictionary with $O(\log n)$ search time and $O(\log^2 n)$ amortized update time. In Section 4, we describe a more sophisticated implementation of the bottom layer for obtaining an optimal amortized update time.

3.1 Top layer: Density functionalities

The super-root is actually an implicit complete binary tree of height $h = O(\log n)$. Let us consider the array of actual chunks shown in the top layer of Figure 3. We partition them 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 $O(a/\hat{k}) = O(n/(bk))$ segments. These segments in R form the leaves of the implicit tree. Its internal nodes are associated with groups of consecutive segments in R. Namely, an internal node u at depth s is associated with 2^{h-s} segments (for a total of $2^{h-s}\hat{k}$ actual chunks). Specifically, let us partition R into disjoint groups of 2^{h-s} consecutive segments each. If u is the rth node at depth s (from left to right), then its associated segments are in the rth group (from left to right).

We need to encode $O(\log n)$ bits of information at each node of the implicit tree. We exploit a static mapping between the internal nodes and the segments of R. Each leaf of the implicit tree is mapped to its own segment. Since the number of nodes does not exceed the number of leaves, we can easily map each internal node to a segment. (Hence, each segment is mapped to one leaf and at most one internal node.) Note that this mapping is for enconding purposes only, while it remains valid the association of internal nodes with groups of segments mentioned in the previous paragraph. 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 to each other by

applying the notion of density [5, 18] to R. In our case, we have the further requirement of avoiding to produce empty slots and that of distributing virtual chunks among the actual chunks without violating their order (i.e., we cannot simply move one virtual chunk from a position to another of R). That allows us to maintain virtual chunks almost uniformly distributed in R, with at most α virtual chunks 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 $d(u) = \frac{|vir(u)|}{\alpha|act(u)|}$. (We show at the end of Section 3.2 that α must be chosen so as to satisfy the relation $2\alpha\rho_0 + 1 < \alpha\tau_0$ for $\alpha \geq 2$.)

The density of all nodes is constrained according to positive real constants, $0 < \rho_\infty < \rho_0 < \tau_0 < \tau_\infty < 1$. 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., underflows) if $d(u) > \tau_s$ (resp., $d(u) < \rho_s$) and recursively at least one of its children either overflows or underflows [5, 18].

3.2 Top layer: Rebalancing/redistribution of chunks

We have seen in Section 2.2 that 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 adding a new chunk to a maximal list of size α, we have to redistribute the chunks by reclassifying them as actual and virtual, while preserving their relative order.

The redistribution of chunks inside the segments (leaves) of R can operate sequentially on the whole segment in $O(k^2)$ time, as each segment contains $\hat{k}$ chunks. For example, let's take any two consecutive actual chunks c and c' in a segment of R, with associated lists L and L' of virtual chunks, respectively, each list containing at most α entries. Suppose that L becomes of size $\alpha + 1$ after a bucket splitting. The rightmost virtual chunk in L, say c_i, replaces c'. Specifically, c_i becomes actual occupying the positions of c' in R, and c' becomes virtual occupying the positions of c_i. Moreover, c' is prepended as virtual to list L', which becomes associated with c_i (now, actual). If also L' becomes of size $\alpha+1$, we go on iteratively with the next actual chunk c'' until we reach the end of the segment. Alternatively, we can proceed analogously towards the beginning of R. Handling the removal of an actual chunk after bucket merging can be treated in the same fashion. (Note that moving the chunks around in the top layer does not change their association with their buckets in the bottom layer.)

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.

Redistribution of chunks for u. In the first phase, we scan the actual chunks of u in decreasing order. We create an empty linked list L for incrementally containing 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 in decreasing order, while u stores the $|act(u)|$ largest chunks in its portion of R, in increasing order. A further scan of L for reversing the links gives the chunks in L in increasing order.

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 $\rho_t < \rho_s \leq d(v) \leq \tau_s < \tau_t$.

A special case of redistribution is in the root u of R. For this, we require that the constants satisfy the relation $2\alpha\rho_0 + 1 < \alpha\tau_0$ for $\alpha \geq 2$ (e.g., $\alpha = 2$, $\rho_0 = 0.1$, and $\tau_0 = 0.9$). 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. Note that the size of the root area in the memory layout does not change. Consequently we run redistribution inside the root area in which the second phase is slightly modified as discussed next.

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 also transform $a = |act(u)|$ virtual chunks 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 $2\alpha\rho_0 + 1 < \alpha\tau_0$. The new density 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 also transform $(1/2)a = (1/2)|act(u)|$ actual chunks 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 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 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 (because $\rho_0 < d(v) < \tau_0$ and $\rho_s < \rho_0 < \tau_0 < \tau_s$).

3.3 Top layer: Analysis

The analysis of the amortized cost for an internal node v on depth $s+1$ is a variation of that given in [5]. The cost for rebalancing v is $O((|act(u)| + |vir(u)|) \cdot k)$ time, where u is the parent of v (and has depth s). When v overflows, we can charge the rebalancing cost to at least $\alpha|act(v)|(\tau_{s+1} - \tau_s)\Theta(b)$ insertions 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

$$\frac{(|act(u)| + |vir(u)|)\, O(k)}{\alpha|act(v)|(\tau_{s+1} - \tau_s)\Theta(b)} = \frac{(|act(u)| + |vir(u)|)\, O(k)h}{\alpha|act(v)|(\tau_\infty - \tau_0)\Theta(b)} \leq \frac{(\alpha+1)|act(u)|O(k)h}{\alpha|act(v)|(\tau_\infty - \tau_0)\Theta(b)} = O\left(\frac{kh}{b}\right),$$

noting that $|act(u)| = 2|act(v)|$. A similar analysis holds for deletions. When v underflows, we can charge the rebalancing cost to at least $\alpha|act(v)|(\rho_s - \rho_{s+1})\Theta(b)$ deletions in the buckets associated with the actual and virtual chunks in the portion of R corresponding to v. Thus the amortized cost per operation on v is

$$\frac{(|act(u)| + |vir(u)|)\, O(k)}{\alpha|act(v)|(\rho_s - \rho_{s+1})\Theta(b)} = \frac{(|act(u)| + |vir(u)|)\, O(k)h}{\alpha|act(v)|(\rho_0 - \rho_\infty)\Theta(b)} \leq \frac{(\alpha+1)|act(u)|O(k)h}{\alpha|act(v)|(\rho_0 - \rho_\infty)\Theta(b)} = O\left(\frac{kh}{b}\right).$$

Since there are h levels, each update operation has an amortized cost of $O(kh^2/b) = O((\log^3 n)/b)$ 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.

Lemma 1 *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 $O((\log^3 n)/b)$ time for a bucket size $\Theta(b)$. At any time, only $O(1)$ auxiliary locations are required to operate dynamically.*

3.4 A simple bottom layer: Final cost

We can obtain a simple sub-optimal implicit dictionary by populating the bottom layer with buckets of size $b = \Theta(\log n)$. In order to guarantee that a bucket splits/merges every other $\Theta(\log n)$ insertions and deletions, we have to bound the minimal and maximal size. More specifically, a bucket is merely a sequence of contiguous entries containing from k to $7k - 1$ keys (other choices for the multiplicative constants are possible). We preserve the invariant on the bucket size by employing simple split and merge operations. For example, a bucket with $7k$ keys splits into two contiguous sequences (buckets) of size $3k$ each, plus a chunk formed by the k median keys of the original bucket. We can handle the entire set of buckets without wasting memory by employing in a straight way the lists introduced in [21]. Namely, the buckets of the same size are packed together into a compactor list of chunks. Hence, there so many lists as the possible sizes of the buckets (from k to $7k - 1$). The heads of

the $O(\log n)$ resulting compactor lists are the only ones consisting of chunks partially filled. They are therefore stored all together at the end of the bucket area. We need to scan all of them in $O(b^2)$ time for inserting or deleting a key. Hence, with this technique we can manage the bottom layer using $O(b^2)$ time for each modification of a single bucket. Searching takes $O(b)$ time in a bucket by a simple scan. By plugging parameter $b = \Theta(\log n)$ into Lemma 1, we obtain an amortized update cost of $O(\log^2 n)$ time for the resulting implicit dictionary, while searching takes $O(\log n)$ time.

We need also to remove the assumption that $n'/4 < n < n'$ holds at any time, where n' is a power of two. That value of n' is important to fix the parameter k discussed in Section 2. Note that the time complexity of our algorithms is parametric in k and that we fixed $k = \Theta(\log n')$ to get our claimed bounds. To preserve the invariant on $n'/4 < n < n'$ when $n = n'$, we double n', update the value of k and rebuild by a sequence of $O(n)$ insertions, with the only difference that we fix $k = \Theta(\log n')$ even if we may have re-inserted less than $n'/4$ keys during the rebuilding (note that changing the value of k is the cause of rebuilding, so we keep k fixed since the beginning of the rebuilding). We perform the complete rebuilding all at once. Analogously, when $n = n'/4$ we halve n', update the value of k, and rebuild. In both cases, the event marks the beginning of a new epoch, with $n = n'/2$ for the new value of n' after rebuilding. Hence our invariant is maintained with n half on the way between $n'/4$ and n'. The amortized cost of rebuilding is given by the total cost of $O(n')$ insertions divided by the number of insertions and deletions performed in an epoch, which is $\Omega(n')$. As a result, the amortized cost of the rebuilding is the cost of $O(1)$ insertions (here it should be clear why we do not start a nested sequence of rebuilding operations, since we keep $k = \Theta(\log n')$ unchanged for all the rebuilding).

Theorem 1 *There exists an implicit dynamic dictionary storing n distinct keys that supports searches in $O(\log n)$ time and updates in $O(\log^2 n)$ amortized time.*

In order to improve over Theorem 1, we need a more sophisticated organization of the bottom layer, as described next.

4 An Optimal Data Structure

In this section, we show how to reduce the update cost to $O(\log n)$ amortized time while preserving the search cost to $O(\log n)$ time in the worst case. A moment of reflection indicates that the top layer is not much a problem. If we fix $b = \Theta(\log^2 n)$ in Lemma 1, we obtain the claimed bounds for that level. Unfortunately, making the buckets of size $\Theta(\log^2 n)$ is not that easy and requires a more sophisticated organization of the keys in the bottom layer.

As a basic tool we use the well-known technique of *block interchange*. A block S of s keys can be reversed in-place and in $O(s)$ time by swapping the first key with the last key, the second key with the $(s-1)$st key, and so on. We denote the resulting block as S^R. Two consecutive blocks X and Y can be exchanged in-place and in linear time in three steps: first compute X^R, then Y^R, and finally $(X^R Y^R)^R$. Since $(X^R Y^R)^R = YX$, we are done. (Note that the direct, pairwise exchange of keys cannot interchange X and Y if their number of keys is not a multiple of each other.)

In the rest of the section, we focus on how to lay out these buckets in the bottom layer for supporting the following operations: searching, inserting, deleting, merging and splitting.

As previously mentioned, there are $\Theta(\log^2 n)$ keys in each bucket and, for getting amortized bounds, we require that an individual bucket splits or merges every other $\Theta(\log^2 n)$ insertions and deletions in that bucket. Analogously to the top level, keys are grouped into chunks, plus a number of spare keys being handled differently.

The bottom layer stores the buckets in the form of an implicit dynamic forest. Each bucket is a tree organized into a constant number of levels. The root is either an actual chunk or a virtual chunk in the top layer. The root has a single child, called *intermediate node*, containing $\Theta(\sqrt{k})$ chunks and having $\Theta(\sqrt{k})$ leaves as children. Each leaf also contains $\Theta(\sqrt{k})$ chunks and has associated a *maniple* of $\Theta(k)$ keys with further $\Theta(\sqrt{k})$ spare keys. As previously mentioned, the buckets are pairwise disjoint when considered as intervals of keys, so that we can write either $b_1 < b_2$ or $b_1 > b_2$ for any two buckets (analogously to what depicted in Figure 3). We now detail the constants hidden in the Θ-notation to guarantee that an individual bucket splits or merges every other $\Theta(\log^2 n)$ insertions and deletions in it, noting that other choices of the constants are possible.

4.1 Buckets: Intermediate nodes

Each intermediate node is the only child of its root and varies by a chunk at a time. The pointer to the ith child leaf is encoded by $O(\log n)$ keys in the ith chunk of the intermediate node. To bound the number of chunks (hence, of children) of an intermediate node u, we follow the approach of weight-balanced trees (e.g., [4]) by defining the weight $w(v)$ of a leaf v as the number of keys in v, and the weight $w(u)$ of an intermediate node u as the number of keys in the leaves that are children of u. We keep the invariant that $k\sqrt{k} \leq w(v) \leq 4k\sqrt{k}$ and $4k^2 \leq w(u) \leq 16k^2$, for any leaf v and any intermediate node u. As a result, the number of chunks in u ranges from its minimum weight divided by the largest weight in a leaf, to its maximum weight divided by the smallest weight in a leaf, namely, from $\sqrt{k}$ to $16\sqrt{k}$ chunks. The chunks inside an intermediate node u are not maintained in any particular order. Since there are $\Theta(\sqrt{k})$ of them, routing a search for a key x in u consists in a simple linear scanning of the set of keys composed by the first key of each chunk of u, plus a full scan of an individual chunk. Node u supports the following primitives:

- InsertChunk(c, u) assumes that c is contiguous to u in memory. Under this hypothesis, we have nothing to do since the chunks of u can be in any order.

- ExtractChunk(c, u) has to enforce the post-condition that c is one extreme of u. Therefore, we simply exchange chunk c with either the first or the last chunk in u.

Lemma 2 *Given an intermediate node u, routing a key in u requires $O(k) = O(\log n)$ time. Operation* InsertChunk *requires $O(1)$ time and* ExtractChunk *requires $O(k) = O(\log n)$ time.*

4.2 Buckets: Leaves

The leaves vary by a chunk at a time and contain from $k\sqrt{k}$ to $4k\sqrt{k}$ keys (i.e., from $\sqrt{k}$ to $4\sqrt{k}$ chunks). We define the weight of a leaf as the number of keys in it. The leaves undergo a more involved internal organization because they delegate the insertion and deletion of

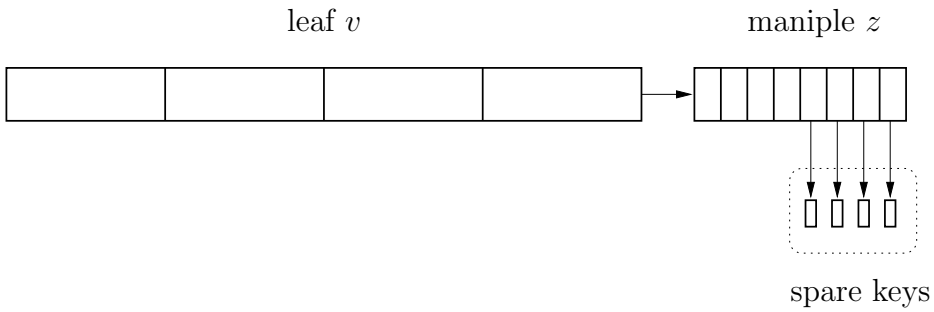

Figure 5: A leaf v made up of chunks, and its associated maniple z with the spare keys.

individual keys to their associated maniples. Given a leaf, we maintain its chunks in sorted order. However, each chunk is internally rotated. Let $a_1, a_2, \ldots, a_k$ be the keys contained in any chunk in a leaf:

1. $a_1, a_2, \ldots, a_k$ are kept *rotated* by an *offset* r, occurring as $a_{r+1} \cdots a_k \cdot a_1 \cdots a_r$ in the chunk;
2. either $1 \leq r \leq \sqrt{k}$ or $k - \sqrt{k} + 1 \leq r \leq k$ (i.e., keys $a_{\sqrt{k}+1} \cdots a_{k-\sqrt{k}}$ are not rotated);
3. $a_{\sqrt{k}+1}, \ldots, a_{k-\sqrt{k}}$ are reserved for the $O(1)$ pointers and integers to be encoded as structural information in the chunk.

Preserving the above organization crucially relies on keeping condition 2 valid. When the latter condition is violated in a chunk, we reset its rotation with a block interchange of $a_{r+1} \cdots a_k$ and $a_1 \cdots a_r$ taking $O(k)$ time. Routing a search key x inside v can be done with the same method used for the intermediate nodes.

When inserting a key into v, we start a sequence of internal rotations that quickly make room for the new key and eject the largest key in v. Analogously, when removing a key from v, we want the opposite situation, namely, remove the deleted key and add to v a key that is largest than any other key in v. Since the chunks are placed in arbitrary order inside v, each time we need to identify the successor chunk or predecessor chunk of the current chunk, on the fly, scanning one key for each chunk in v (recall that we have $\Theta(\sqrt{k})$ chunks in v, and we need to perform the on-the-fly scan $O(\sqrt{k})$ times). Reducing the problem of inserting or deleting a key in v to that of handling the operation suitably in its maniple is the rationale for the rotational machinery described in the conditions 1–3.

Let's detail a bit more the insertion, and assume that we have to add one more key x to a chunk c in v. What we have to do is inserting x into c by shifting its keys while extracting the maximum key in c, which we insert into the chunk just to the right of c. Hence, for any chunk c' to the right of c, we insert into c' the minimum key x_1 (extracted as the largest from the chunk to the left of c') while extracting the maximum key x_2 (being inserted as the smallest into the chunk to the right of c'). When we reach the right end of v, we have extracted the maximum key in v. While we can shift the keys in c in $O(k)$ time, we cannot afford that cost for the remaining chunks c' to the right of c, as they can be $\Theta(\sqrt{k})$ in number. Using the organization in conditions 1–2, we pay just $O(\log k)$ time per chunk, whose total cost is $O(k)$. Given one such chunk c', we show how to insert x_1 and extract $x_2 = a_k$. We first find the value of rotation r by a variant of the binary search applied to the keys in c'. Next, we identify the location of a_k inside c', and replace that key with x_1, obtaining $a_{r+1} \cdots a_{k-1} x_1 a_1 \cdots a_r = a'_{r+2} \cdots a'_k a'_1 \cdots a'_r a'_{r+1}$. At the same time, we

have extracted $x_2 = a_k$ for the next round.

We postpone the analysis of the cost of how to maintain the invariant for conditions 1–3 to Section 4.6. The other operations supported are:

- InsertChunk(c, v) assumes that c is contiguous to v in memory. Without loss of generality, let us assume that c is to the right of v. Since we have to maintain the sequence chunks of v in sorted order, we insert c in its right position with a sequence of block interchanges involving c and each of the chunks of v that are larger than c. Note that the initial rotation for c has offset 0.

- ExtractChunk(c, v) is analogous to InsertChunk.

Lemma 3 *Given a leaf v, routing a key in v requires $O(k) = O(\log n)$ time. Operations* InsertChunk *and* ExtractChunk *require $O(k\sqrt{k}) = O(\log^{1.5} n)$ time.*

4.3 Buckets: Maniples and spare keys

In addition, each leaf has associated a maniple varying by $\sqrt{k}$ keys at a time, thus containing from k to $5k$ keys that are larger than those contained in the leaf. We call a group of $\sqrt{k}$ contiguous keys a *mini-chunk*, so that $\sqrt{k}$ mini-chunks give rise to a single chunk (where mini-chunks are disjoint when considered as intervals). Hence, each maniple varies by a mini-chunk at a time.

The last $\sqrt{k}$ mini-chunks in the maniple have the spare keys associated, namely, from 1 to 5 spare keys per mini-chunk. These keys are contained in the interval represented by their mini-chunk. This property is crucial for searching in the maniple and decoding only $O(1)$ pointers to the spare keys. (Note that associating more spare keys to the maniple does not improve time bounds.) However, we cannot encode these pointers in the maniple; we encode them in the first $\Theta(\sqrt{k})$ chunks of the corresponding leaf, since they are sufficiently large to allow this encoding with its keys. Specifically, the pointer assigned to a chunk is encoded by the reserved keys mentioned in condition 3 on the invariant for the leaves (see Section 4.2). The association is static and, independently of how many spare keys are associated with a given mini-chunk, we reserve keys for encoding 5 pointers in the leaves (some of these can be null pointers, encoded as 0). When searching in the maniple, we scan one key per mini-chunk and end up in a mini-chunk that is fully scanned. We decode at most 5 of these pointers to spare leaves, namely, those corresponding to the mini-chunk in which our search ends. We also support the following primitives on a maniple z:

- InsertKey(x, z) applies to the case when there are less than $5\sqrt{k}$ spare keys in total in z, contained in the interval represented by their mini-chunk (the case in which there are exactly $5\sqrt{k}$ spare keys is treated in the general scheme of Section 4.5). We identify the mini-chunk c of z that encloses x. If c has less than 5 spare keys, we insert x into c as one more spare key, encoding its pointer in the proper chunk of the corresponding leaf v. Otherwise, we extract the largest key y of c and consider the successor mini-chunks of c. We identify the first among them, say c', that has less than 5 spare keys. We shift by one position to the right all the keys in the successors of c up to c' (excluded) to make room for y. Note that in this process we move the keys in the mini-chunks. However, due to the shift inside these mini-chunks, some of

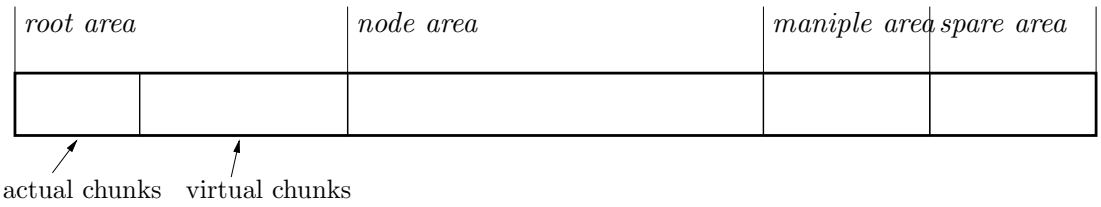

Figure 6: A refined view of the layout in Figure 4.

them may have their spare keys that become the largest in their mini-chunks. In this case, we rearrange these mini-chunk so that their spare keys are no more the largest. At the end, c' as one extra key to be inserted into it that comes from its predecessor mini-chunk, incrementing its spare keys by one as described above. Alternatively, we can proceed by looking at the predecessor mini-chunks of c.

- ExtractKey(x, z) is analogous to InsertKey.

The repeated invocation of InsertKey can violate the invariant that the spare keys are contained in their mini-chunk, considered as an interval of keys. Consequently, we periodically scan all mini-chunks and their spare keys choosing as new spare keys the medians of each mini-chunk. This preserves the crucial property that the spare keys are inside the mini-chunks as intervals for any sequence of InsertKey and ExtractKey operations. We will analyze the cost of this reorganization of maniples in Section 4.6.

Lemma 4 *Given a maniple z and its leaf v, routing a key in z requires $O(k) = O(\log n)$ time. Operations* InsertKey *and* ExtractKey *take $O(k) = O(\log n)$ time each.*

4.4 Memory layout of the buckets

We now describe where the keys are physically stored in the bucket area for the bottom layer. With reference to Figure 4, the bucket area is further divided into *node area* (for accommodating the intermediate nodes and the leaves), *maniple area* and *spare area* for the rest of the bucket trees, as shown in Figure 6. Recall that the bucket area may grow or shrink by k positions to its left and by one position to its right.

We store the spare keys in no particular order in the spare area. Differently from [15, 21], our memory management introduces the *compactor zones* (or, simply, *zones*) for accommodating the intermediate nodes and the leaves in the node area, and the maniples in the maniple area. We maintain a dynamic partition of the node area (respectively, the maniple area) into zones of variable dimension, the compactor zones, with the property that each compactor zone stores nodes and leaves (respectively, maniples) of identical size in a *contiguous* segment of memory. Hence, no two equal-sized nodes can reside in two distinct zones, nor a node of size different from the others can share the same zone. We give more details below.

We pack together the nodes of *identical* size s, embedding them in a suitable zone devoted to nodes of size s and called *zone s*. When a node changes size, it also changes zone. Each node in zone s occupies a contiguous segment of s memory cells, except possibly the node at

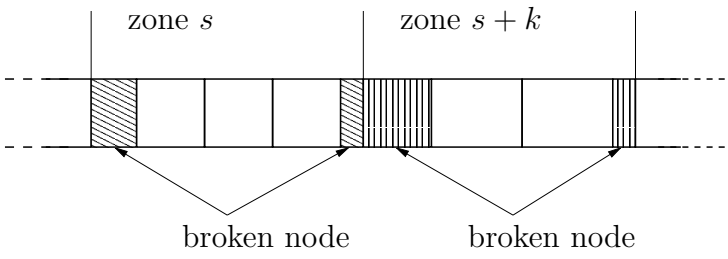

Figure 7: Compactor zones s and $s+k$.

the beginning of each zone. In this case, we maintain the property that the node is stored in two segments of s_1 and s_2 cells, respectively, where $s_1 + s_2 = s$. The last s_2 keys of the nodes are at the beginning of zone s and the first s_1 keys of the node are at the end of zone s. We call *broken* this node and *unbroken* any other node in the zone. Hence, all nodes are unbroken except possibly one node per zone (see Figure 7). The (encoded) pointer to a broken node contains extra $O(\log k)$ bits to encode that value of s_1. A similar picture holds for the maniple area, which is also handled with zones.

The zones share common techniques, such as the following primitive for rotating a zone s. Suppose we have m keys to the right of zone s, and let X be the memory segment hosting the whole zone s along with the m keys to its right. A *rotation* of zone s is an in-place primitive that incrementally moves the m keys to the beginning of X and the first m keys in zone s to the end of X (or vice versa) *without* scanning the whole X, contrarily to what done by block interchanges (described in Section 2). For $i = 1, 2, \ldots, m$, it exchanges the ith key of zone s with the ith among the m keys. At the end, the broken node (if any) in zone s may become unbroken and at most one unbroken node may become broken. Since these two nodes change physical allocation inside the node area, we need to update the incoming pointers to them encoded in their parents (i.e. their roots if they are intermediate nodes, or their intermediate nodes if they are leaves). In order to identify these parents, we search one key for each of the nodes involved in the rotation. We then re-encode the $O(1)$ pointers to them that are identified through the search. The cost of a rotation is $O(k + m)$ time plus the cost of $O(1)$ searches.

We now give more details on the zones. The associated primitives are invoked when a node or a maniple changes size, and so its keys must be suitable relocated in memory.

Node area. It contains $15\sqrt{k} + 1$ compactor zones in increasing order of index s, to cover the range of all possible node size s. Each zone s is adjacent to zone $s - k$ (to its left) and to zone $s + k$ (to its right), since the size of the nodes is a multiple of k. The starting positions of all the zones in this area are encoded in the first $\sqrt{k}$ chunks of the area itself. We support the following basic primitives in this area:

- ExtractNode(w) extracts node w placing it between the node area and the maniple area. Let s be the size of w. We exchange w with the rightmost unbroken node in zone s (note that w may be the broken node). Now, we have w followed by the initial portion of the broken node (if any) in zone s and by zone $s + k$ (if any). We perform a block interchange between w and the portion of the broken node in $O(k\sqrt{k})$ time. We also need to perform $O(1)$ searches to re-encode $O(1)$ pointers in their parents. As

a result, we shorten zone s by s positions to the right and have w between zones s and $s+k$. For $s' = s+k, s+2k, \ldots, 16k\sqrt{k}$ (i.e., incrementing s' by k), we move w from the left to the right of zone s' by a rotation of zone s' and update its new starting position. At the end, we have w to the right of zone $16k\sqrt{k}$, that is, between the node area and the maniple area. Note that $O(\sqrt{k})$ zones are rotated.

- InsertNode(w) inserts node w into its suitable compactor zone and is similar to ExtractNode (with the same cost), where w is between the node area and the maniple area.

Lemma 5 *The cost of* ExtractNode *and* InsertNode *in the node area is* $O(k^2) = O(\log^2 n)$ *time plus the cost of* $O(\sqrt{k}) = O(\sqrt{\log n})$ *searches.*

- TransferChunk(c) transfers chunk c from an area to another. Either c is between the node area and the maniple area and we want it laying between the root area and the node area, or vice versa. We proceed like in ExtractNode and InsertNode. However, we do not insert or delete c inside any zone, but we simply rotate each zone s' to move c from one side to another of zone s'. The cost of the primitive takes into account the fact that c has size k. (This is the only difference behind Lemma 6 with respect to Lemma 5.)

Lemma 6 *The cost of* TransferChunk *in the node area is* $O(k\sqrt{k}) = O(\log^{1.5} n)$ *time plus the cost of* $O(\sqrt{k}) = O(\sqrt{\log n})$ *searches.*

Maniple area. It contains $4\sqrt{k}+1$ compactor zones in increasing order of index s. Each zone s contains the maniples of identical size s. Its neighbors are zone $s-\sqrt{k}$ (to its left) and zone $s+\sqrt{k}$ (to its right), since the size of the maniples is a multiple of $\sqrt{k}$. The starting points of all the zones are encoded in the first $\sqrt{k}$ chunks of the node area. We support the following primitives:

- ExtractManiple(z) extracts maniple z and place it either between the node area and the maniple area, or between the maniple area and the spare area. Its implementation is analogous to ExtractNode, taking into account the size $O(k)$ of the maniples.

- InsertManiple(z) inserts maniple z into its zone analogously to InsertNode, where z is either between the node area and the maniple area, or between the maniple area and the spare area.

- TransferKeys(m) transfers m contiguous keys from the left of the maniple area to its right, or vice versa, where $m \leq k$. It is like TransferChunk, except that each zone rotates by m positions.

Lemma 7 *The cost of* ExtractManiple, InsertManiple *and* TransferKeys *in the maniple area is* $O(k\sqrt{k}) = O(\log^{1.5} n)$ *time plus the cost of* $O(\sqrt{k}) = O(\sqrt{\log n})$ *searches.*

Spare area. Here there are no compactor zones, but the spare keys are stored contiguously without any specific order. So, the basic primitives are simple to describe. One primitive inserts a new spare key to the right of the spare area and extends the right border of the area to include the new spare key. Another primitive extracts a spare key leaving a hole inside the spare area that is filled with the rightmost spare key in the area, thus shortening the right border of the spare area by one position. In all cases, we search the key to update its pointer encoded in a suitable chunk of a leaf.

Lemma 8 *Extracting or inserting a spare key in the spare area has a cost of* $O(k) = O(\log n)$ *time plus the cost of* $O(1)$ *searches.*

We may also want to collect m spare keys between the maniple and the spare area. In that case, we can see this operation as a sequence of m extractions according to Lemma 8, giving a total cost of $O(km)$ time plus the cost of $O(m)$ searches.

4.5 Supporting the operations on the buckets

We now have all the ingredients to discuss how to implement the search, insert, and delete operations on the buckets.

We need to keep the invariants described in Sections 4.1– 4.4 on the number of keys during the updates. Let's illustrate this point by a quick overview of the insertion. We search the key to insert in the top layer and then identify a bucket in which we route the search. We go through the intermediate node and one of its leaves. At this point, we have to insert the key in one of the chunks in the target leaf. This causes the removal of the largest key in the chunk and its insertion into its successor chunk, propagating this operation to all the successors, until the largest key is extracted from the leaf and inserted into its maniple z, in which we have an analogous propagation until a mini-chunk with spare keys is reached. Hence, inserting a key is tantamount to incrementing by 1 the number of spare keys in z. If the mini-chunk in z hosting the new key has already associated 5 spare keys, we suitably redistribute the keys in z. If z has already associated a total of $5\sqrt{k}$ spare keys, we extract $\sqrt{k}$ spare keys and transform them into a mini-chunk that is added to z. However, if z has its maximum capacity of $5k$ keys, we extract the smallest chunk in z inserting it as the largest chunk in its corresponding leaf v.

At this point, if v contains $4k\sqrt{k}$ keys, we go on splitting both v and z, creating two leaves with their associated maniples and spare keys. All satisfy the invariants on their number of keys, which is roughly half on the way between the minimum and maximum allowed. The split may cause a chunk c to be inserted in the intermediate node u, parent of v. If u has maximum weight $w(u) = 16k^2$, we need to split u into two nodes of roughly the same weight (the weights of the two resulting nodes may differ by just $O(\sqrt{k}\,k)$). We create two buckets from the current bucket and the chunk c' resulting from the split of u becomes the root of the new bucket with the largest keys. Note that $\Omega(k^2)$ insertions and deletions must have been performed in the bucket. In this way, we have enough credits to add c' to the root area as actual or virtual chunk. Deleting a key is analogous, except that we merge instead of splitting u (although, after a merge, we may split once); as for v, we need merging and borrowing with an individual key. We now detail more the several operations.

Searching a key in a bucket. Searching key x after visiting the root of the bucket in the top layer consists of routing x inside the intermediate node u, the only child of the root, as stated in Lemma 2. If we find the key as a member of a chunk in u, we are done. Otherwise, we identify a chunk c in u, reaching the leaf v whose pointer is encoded in c. We route x also inside v according to Lemma 3. If x is a member of v, we are done. Otherwise, either x is not stored in the dictionary, or it is stored in its associated maniple z. We search in z according to Lemma 4 and identify a mini-chunk c' in z. If x is a member of c' or equals one of its $O(1)$ spare keys, we are done. Otherwise, we can infer that the search of x is unsuccessful.

Lemma 9 *Given the root of a bucket, searching x in the bucket takes $O(k) = O(\log n)$ time.*

Inserting a key into a bucket. We now discuss the insertion of x into a bucket. We describe our insertion in two phases, a descending phase (steps 1–3) and an ascending phase (steps 4–5).

1. We begin by searching x as stated in Lemma 9. If x belongs to the chunk that is the root, we add x and shift at most k keys to extract the maximum key in that chunk, obtaining the new key x to insert into the intermediate node u, the only child of the bucket root. In general, inserting x into a chunk of u goes along the same lines. The maximum of the chunk is extracted to make room for the new key. We set x to this maximum key, which should be inserted into the suitable child leaf v of u.

2. We insert x into a leaf v with maniple z. We run the algorithm described in Section 4.2 for inserting x into v while extracting its maximum key by means a sequence of internal rotations. We set x to be the extracted key, which should be inserted into z.

3. We insert x into maniple z using the following substeps:

   (a) If $\sqrt{k}$ insertions and deletions have been performed on z, we reset the counting for z and choose its new spare keys as the medians of each mini-chunk (see Section 4.3).

   (b) If the number of spare keys in z is less than $5\sqrt{k}$, we perform InsertKey(x, z), increasing by one the number of spare keys associated with z and Exit.

   (c) If z has the maximum number of spare keys, $5\sqrt{k}$, we move the smallest $\sqrt{k}$ spare keys, say $s_1, s_2, \ldots, s_{\sqrt{k}}$, in z at the beginning of the spare area, shortening that area. Then, we proceed by induction for $i = 1, 2, \ldots, \sqrt{k}$, where the smallest $i-1$ keys in z have been exchanged with $s_1, \ldots, s_{i-1}$. We find the rank r_i of s_i among the current keys of z. We then shift the first r_i keys in z by one position to the left to make room for s_i, with the smallest key "ejected" from z and stored in the cell left free by s_i. With an analogous method, we can redistribute the remaining spare keys in z, so that they are the medians of their corresponding mini-chunks.

   (d) We run ExtractManiple(z) to bring z between the node area and the maniple area, and apply TransferKeys$(\sqrt{k})$ to move near to z the $\sqrt{k}$ extracted keys, which form a mini-chunk. We move this mini-chunk at the beginning of z by block interchange. We extend z with this mini-chunk. If z has still no more than $5k$

keys, we run InsertManiple(z) to put z back to its proper compactor zone and execute only step 3a.

4. Here, z has $5k + \sqrt{k}$ keys and stays between the node area and the maniple area.
   (a) We run ExtractNode(v) for the leaf v having z associated, putting v between the node area and the maniple area, just to the left of z.
   (b) If v has less than $4k\sqrt{k}$ keys, we separate the first chunk c of z from the rest of it, thus shortening z by a chunk and creating a chunk that must be inserted into the leaf v with InsertChunk(c, v). We then execute InsertNode(v) and InsertManiple(z) to their compactor zones, and jump to step 3a.
   (c) If v has $4k\sqrt{k}$ keys, we split v into two leaves v_1 and v_2, their maniples z_1 and z_2 and their spare keys from those in v and z and from the spare keys associated with v.
      i. We move the spare keys of z at the beginning of the spare area and apply TransferKeys to bring them to the right of z between the node area and the maniple area.
      ii. We perform an in-place merge of the sorted keys in z and the sorted sequence of the spare keys by repeated insertions. As a result, we have a sorted sequence of keys by scanning v and the merge of z with the spare keys.
      iii. We divide the sequence in six parts: $v_1, z_1, s_1, c, v_2, z_2, s_2$, where c is the median chunk. Leaf v_1 contains $2k\sqrt{k}$ keys and has associated the maniple z_1 of $2k$ keys and $2\sqrt{k}$ spare keys in s_1. Leaf v_2 contains $2k\sqrt{k}$ keys and has associated the maniple z_2 of $2k+\sqrt{k}$ keys and $2\sqrt{k}$ spare keys in s_2. We perform some block interchanges in these $O(k\sqrt{k})$ keys obtaining $v_1, v_2, c, z_1, z_2, s_1, s_2,$.
      iv. We reinsert v_1 and v_2 into the node area with InsertNode, and z_2 and z_1 into the maniple area with InsertManiple. We are left with the keys in c, s_1, s_2 between the node area and the maniple area. We encode in c a pointer to the position of v_2.
      v. We apply TransferKeys moving the keys in s_1, s_2 to the positions between the maniple area and the spare area. We extend the spare area by $|s_1| + |s_2|$ positions to the left. Since the keys in s_1 and s_2 are not spare keys, we redistribute these spare keys among the keys of their maniples, in a way similar to that presented in step 3c.

5. Here, chunk c stays between the node area and the maniple area.
   (a) We run ExtractNode(u) for the parent u of v, so that u is near to c between the node area and the maniple area. We add c to u with InsertChunk(c, u).
   (b) If u has weight $w(u) \leq 16k\sqrt{k}$ keys, we run InsertNode(u) back to its compactor zone and jump to step 3a.
   (c) If u has has weight $w(u) > 16k\sqrt{k}$, we sort in-place the keys in u and we split them into u_1, c' and u_2 in this order so that $w(u_1)$ and $w(u_2)$ are approximately $8k\sqrt{k}$, and c' is the median chunk. We insert u_1 and u_2 into the node area using InsertNode. We encode in c' a pointer to the position of u_2 and move c'

using TransferChunk so that it reaches the position between the root area and the node area (it will become part of the root area). We have thus created two buckets, and c' is the root of the new bucket containing u_2. We jump to step 3a.

Deleting a key from a bucket. The delete operation for a key x admits an implementation that is symmetrical to that of the insertion. Consequently, we do not give here the details. We only mention here that, once that x is identified in the bucket tree, we delete it from its chunk adding a new maximum key from a child. The deletion in a maniple z starts out an ascending phase that uses merges in place of splits. When a merge is not doable for a leaf v, we borrow a single key from a neighbor (leaf with its maniple and associated spare keys). In all the other case, we merge v and, if necessary, its parent u. The latter may require a split as second operation to preserve the invariant on its weight $w(u)$. Moreover, the insertions and the deletions interact in that they change the rotation of the chunks in the leaves and the position of the spare keys inside the mini-chunks of the maniples.

4.6 Analysis of the costs

We provide the amortized analysis for the cost of the insertions. We closely follow the algorithmic scheme presented in Section 4.5 to show that the cost of each step is $O(k+\log n')$ time.

Step 1 is worst-case by Lemma 9 and by the fact the $O(1)$ chunks are accessed.

Step 2 is amortized. What is more expensive is the reset of the rotation in the chunks of the leaf that otherwise would violate conditions 1–3 in Section 4.2. Resetting each chunk costs $O(k)$ time and there can be $O(\sqrt{k})$ of them. Since for any two consecutive resettings there are $\Omega(\sqrt{k})$ insertions and deletions involving leaf v, the amortized cost of a resetting is $O(k)$ time.

Step 3 is amortized. Step 3a requires $O(k\sqrt{k})$ time because of decoding the pointers to the $\Theta(\sqrt{k})$ spare keys. Step 3b is worst-case by Lemma 4. Step 3c has the same cost as that of Step 3a, plus the cost of $O(\sqrt{k})$ searches by Lemma 8. Step 3d costs as Step 3c by Lemma 7 and the by the cost of the block interchange. In summary, the total cost of Step 3 is $O(k\sqrt{k})$ time plus the cost of $O(\sqrt{k})$ searches. Since Steps 3a–3d occur every $\Omega(\sqrt{k})$ insertions and deletions involving z, the amortized cost is $O(k)$ time plus the cost of $O(1)$ searches.

Step 4 is amortized and occurs every $\Omega(k)$ insertions and deletions involving z, whose leaf is v, except for Step 4c, which occurs every $\Omega(k\sqrt{k})$ insertions and deletions involving v and z. The cost of Step 4a is given by Lemma 5, that of Step 4b by Lemma 7 and Lemma 2. This gives an amortized cost of $O(k)$ time plus the cost of $O(1)$ searches. The total cost of Step 4c is $O(k^2)$ time plus the cost of $O(\sqrt{k})$ searches, giving an amortized cost of $O(k)$ time plus the cost of $O(1)$ searches, as expected.

Step 5 is amortized and occurs every $\Omega(k\sqrt{k})$ insertions and deletions involving leaf v, whose parent is the intermediate node u. The cost of Step 5a is given by Lemma 5 and Lemma 2; that of Step 5b by Lemma 5; finally, that of Step 5c by $O(k^2)$ time for the merge done with $O(\sqrt{k})$ repeated insertions, and by Lemma 5, Lemma 8 and Lemma 2. This concludes the analysis of the insertions.

A detailed analysis of the deletions can done analogously to that of the insertions and we leave the details to the interested reader. We obtain the main result of this section.

Lemma 10 *In the bottom layer, each bucket contains $\Theta(k^2) = \Theta(\log^2 n)$ keys at any time. Searching a bucket takes $O(k) = O(\log n)$ time. The amortized cost of updating a bucket is $O(k) = O(\log n)$ time per insert/delete operation plus the cost of $O(1)$ searches. Any newly created bucket will merge or split after further $\Omega(k^2) = \Omega(\log^2 n)$ update operations inside that bucket. At any time, only $O(1)$ auxiliary locations are required to operate dynamically.*

Using Lemma 1 with $b = \Theta(\log^2 n)$ for the top layer, and Lemma 10 for the bottom layer, we obtain the main result of the paper.

Theorem 2 *There exists an implicit dynamic dictionary storing n distinct keys that supports searches in $O(\log n)$ time and updates in $O(\log n)$ amortized time.*

5 Conclusions

In this paper we presented the flat implicit tree, which avoids space wasting by just using an array of $n + O(1)$ cells for the keys. The flat implicit tree is the first optimal data structure obtaining $O(\log n)$ time for search and update in an array of $n+O(1)$ cells since the invention of the heaps in the sixties.

Acknowledgments

We are grateful to the Referees for many comments on how to improve the presentation of the techniques in this paper.

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] Lars Arge. The buffer tree: A new technique for optimal I/O-algorithms. In Selim G. Akl, Frank Dehne, Jörg-Rüdiger Sack, and Nicola Santoro, editors, *Proceedings of the 4th International Workshop on Algorithms and Data Structures (WADS'95)*, volume 955 of *LNCS*, pages 334–345, Berlin, GER, August 1995. Springer.

[4] Lars Arge and Jeffrey Scott Vitter. Optimal dynamic interval management in external memory (extended abstract). In *37th Annual Symposium on Foundations of Computer Science*, pages 560–569, Burlington, Vermont, 14–16 October 1996. IEEE.

[5] 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: proceedings: 12–14 November, 2000, Redondo Beach, California*, pages 399–409, 1109 Spring Street, Suite 300, Silver Spring, MD 20910, USA, 2000. IEEE Computer Society Press.

[6] 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.

[7] 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.

[8] 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.

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

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

[11] Faith E. Fich and Peter Bro Miltersen. Tables should be sorted (on random access machines). In Selim G. Akl, Frank K. H. A. Dehne, Jörg-Rüdiger Sack, and Nicola Santoro, editors, *Algorithms and Data Structures, 4th International Workshop*, volume 955 of *Lecture Notes in Computer Science*, pages 482–493, Kingston, Ontario, Canada, 16–18 August 1995. Springer.

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

[13] Gianni Franceschini and Roberto Grossi. Implicit dictionaries supporting searches and amortized updates in $O(\log n \log\log n)$. In *Proceedings of the 14th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA'03)*, pages 670–678. SIAM, 2003.

[14] Gianni Franceschini and Roberto Grossi. Optimal cache-oblivious implicit dictionaries. In *Proceedings of the 30th International Colloquium on Automata, Languages and Programming (ICALP)*, 2003.

[15] 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.

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

[17] 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.

[18] Alon Itai, Alan G. Konheim, and Michael Rodeh. A sparse table implementation of priority queues. In Shimon Even and Oded Kariv, editors, *Automata, Languages and Programming, 8th Colloquium*, volume 115 of *Lecture Notes in Computer Science*, pages 417–431, Acre (Akko), Israel, 13–17 July 1981. Springer-Verlag.

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

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

[21] 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.

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

[23] 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.

[24] M. H. Overmars. *The Design of Dynamic Data Structures*, volume 156. Springer, Berlin, 1983.

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

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

[27] 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.

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

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

[30] Andrew C. Yao. Should tables be sorted? *J. Assoc. Comput. Mach.*, 31:245–281, 1984.