Optimal Implicit and Cache-Oblivious Dictionaries over Unbounded Universes *

Gianni Franceschini francesc@di.unipi.it

Roberto Grossi grossi@di.unipi.it

February 10, 2003

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 behind 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 the very recent bound of $O(\log^2 n/\log\log n)$. One finding in this paper is that of closing the above longstanding issue with the flat implicit tree, which is the first *optimal* data structure obtaining $O(\log n)$ time for search and update in an array of n cells. Another finding is motivated by the antithetic features of implicitness and cache-obliviousness in data structures. The flat implicit tree combines their best qualities achieving $O(\log n)$ comparisons and $O(\log_B n)$ block transfers for any block size B, where the value of B and the capacity of the memory level are *unknown* to the algorithms operating in the model. As a result, we avoid space wasting by just using the plain array of n cells for the keys and, simultaneously, we provide optimal data access at any level of the memory hierarchy.

Keywords: data structures, searching, dictionary problem, implicitness, memory-hierarchy models, cache-obliviousness.

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 [24] 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

*Dipartimento di Informatica, Università di Pisa, via Filippo Buonarroti 2, 56127 Pisa, Italy. Some of the ideas described in this paper appear in preliminary form in ACM-SIAM SODA 2003 [16], in a technical report [18] and in a submitted conference version [17]. Work partially supported by the Italian MIUR project PRIN "ALINWEB: Algorithmics for Internet and the Web".

and supporting insertions and deletions of individual keys in $O(\log n)$ time [35, 15]. 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 $O(n)$ 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, with *no* further occupied or wasted cell. All what is known is the starting position of the memory segment hosting the array, 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 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 strictly more than n cells of space because of $O(n)$ extra pointers and integers.

Previous work on implicit dictionaries. Munro and Suwanda [28] 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 [24]. 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.* [8] supporting only insertions and searches. While the term "implicit" originated in [28], it has also been the subject of papers taking a somewhat different point of view, including a long lists of results in perfect hashing [21, 13],

bounded-universe dictionaries [11, 30], and cache-oblivious data structures [10]. These results were obtained for less stringent models different from the model adopted by Munro and Suwanda and in following papers.

Yao [37] 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 [21] 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 [14] and Raman, Raman and Rao [33], encoding the keys in at most n cells 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 [20] 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 [27] 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 [26] 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.* [19] 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. [9] and Radhakrishnan and Raman [32] 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 first finding. The implicit dictionary problem is intimately related to the fundamental question whether using *extra* space gives more computational power than using *exactly* n cells, and how data ordering can help in this task. The ultimate goal is in the flavor of the theoretical result of Yao [37], 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 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 [34] and random search trees [25] 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 [29] 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. Morevoer, during the amortization, we can use only in-place algorithms as otherwise the data structures would not requires $O(1)$ RAM registers to operate but also a non-constant temporary area, which makes no sense in the implicit model. As a result, our deletion algorithm is almost symmetrical to the insertion. The reader may notice a greater level of details than in other papers on data structures. This is probably due to the fact that non 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 [26] 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 [19].

Introducing the issue of cache-obliviousness in dictionaries. The dictionary problem can be faced from another perspective, namely, that of cache-oblivious B-trees [5] and other dictionaries in the *cache-oblivious* (memory-hierarchy) model [22]. In this ideal model there are two levels of memory hierarchy, where one level is small and fast and the other level is large but slow. Data transfers between the two levels occur in blocks of B data items; however, the value of B and the capacity of the memory level are *unknown* to the algorithms operating in the model. In [22], the model is shown to simulate any realistic model with two or more memory levels with just a small constant-factor slowdown. As in [5], we do not make any assumption on the capacity of the memory levels. It is demonstrated there how to perform the dictionary operations in $O(\log_B n)$ block transfers for searching and $O(1 + r/B)$ block transfers for scanning r contiguous elements like regular B-trees, while updates take

$O(\log_B n + \log^2 n/B)$ amortized block transfers. If scans are not required, the updates take $O(\log_B n)$ amortized block transfers. Other results in this direction have obtained simplifications [7, 10] and worst-case bounds for the updates [6], and have avoided the use of pointers [10] by keeping all keys stored into an array of size $n + \Theta(n)$. More references to cache-oblivious algorithms and data structures are reported in the survey in [12].

Our second finding. The motivation for our second result arises from the observation that the solutions proposed so far to the dictionary problem are somehow antithetic in the implicit model and in the cache-oblivious model. Implicit dictionaries entirely encode the information by permuting the keys in n contiguous memory cells with *no* additional cell, thus loosing data locality. Due to the dynamic maintainance of the permutation without wasting memory cells, the algorithms give rise to irregular access patterns jumping from one memory cell to another. Cache-oblivious dictionaries carefully handle irregular access patterns to data for getting locality at the price of wasting $\Theta(n)$ memory cells. In order to preserve dynamically the locality of keys, these are suitably interleaved with empty cells whose purpose consists in delaying more expensive redistributions that have a small amortized cost in this way.

In the context of these studies it is an interesting theoretical question to see whether the appealing features of the two models can be blend together by avoiding their contrasting drawbacks mentioned above. We investigate the possibility of designing a dictionary that is simultaneously cache-oblivious and implicit, with a major focus on searching and updating. We show that the flat implicit tree requires $O(\log n)$ comparisons and $O(\log_B n)$ block transfers for searching, and $O(\log n)$ comparisons and $O(\log_B n)$ amortized block transfers for updating, while using just n memory cells. Not only we avoid space wasting, but we provide optimal data access at any level of the memory hierarchy.

A part of the flat implicit tree is inspired by the Van Emde Boas layout of [31] modified in a non-trivial way with further implicit techniques. Compared to previous work, our data structure is complementary to that in [19], as their bounds hold in the worst case and use the cache-aware model; moreover, scanning r keys takes $O(\log_B n + r/B)$ block transfers. The pointerless data structure in [10] is cache-oblivious but, as noted by its authors, it is not properly implicit as it occupies $(1 + \epsilon)n$ cells for any $\epsilon > 0$. As a final remark, our data structure does not support efficient scanning, but this is also an open problem in cache-oblivious data structures alone when the bounds are $O(\log_B n)$ as in our case.

Paper organization. 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, Section 4 giving a description of the top layer, while Section 5 discusses how to make the top layer cache-oblivious. We put all together in Section 6 for our final bounds.

2 Overview of the Flat Implicit Tree

We first give some preliminaries. We encode data by a pairwise (odd-even) permutation of keys [26]. 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

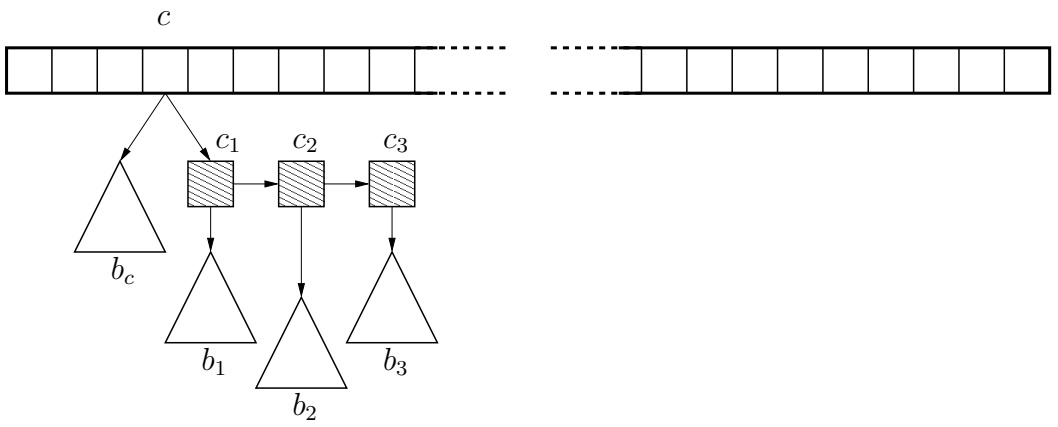

Figure 1: An actual chunk c with its associated bucket b_c of keys, where c is the root of b_c. The virtual chunks c_1, c_2, c_3 are associated with c, and their corresponding buckets are b_1, b_2, b_3. Note that the keys in c, c_1, c_2, c_3 will be stored in the root area of Figure 2, while the keys in b_c, b_1, b_2, b_3 will be suitably stored in the node area, maniple area and spare area shown in Figure 2.

$\min\{x_i, y_i\}$. Although $x_1, y_1, x_2, y_2, \ldots, x_b, y_b$ are permuted, they are quasi sorted. Any algorithm (e.g., binary search) that requires them sorted still works with minor modifications.

From a high-level point of view, the flat implicit tree is a suitable collection of *chunks* [26, 19, 16] and *spare* keys. Each chunk contains k keys that are pairwise permuted for 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. Not all the keys are stored in the chunks. A number of $O(\log n)$ keys are kept in a preamble $\mathscr{P}$ to encode some bookkeeping information. The remaining ones are the spare keys, which are kept together in a contiguous segment of memory without a particular organization.

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 the flat implicit tree; as it should be clear in the rest of the paper, the amount of this information 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. We can avoid to keep the value of n and n' explicitly, as a variable-length encoding, such as the δ-code [36], can represent them asymptotically in the preamble $\mathscr{P}$. The algorithms for maintaining the data structure are parametric in n' and k. We resize k only when either $n = n'/4$ or $n = n'$; this event marks the beginning of a new *epoch*. Hence, the lifetime of the data structure can be divided into epochs. At the beginning of each epoch, we start the process of global rebuilding our implicit dictionary to guarantee that $n'/4 < n < n'$, which is described in Section 6. After each rebuilding, the n distinct keys are organized in two layers as shown in Figure 1:

- The *top* layer is the *super-root* contains $a = \Theta(n/k^2)$ chunks, called *actual* chunks, where a is *always* a power of two. Also, it contains a number of *virtual* chunks stored in

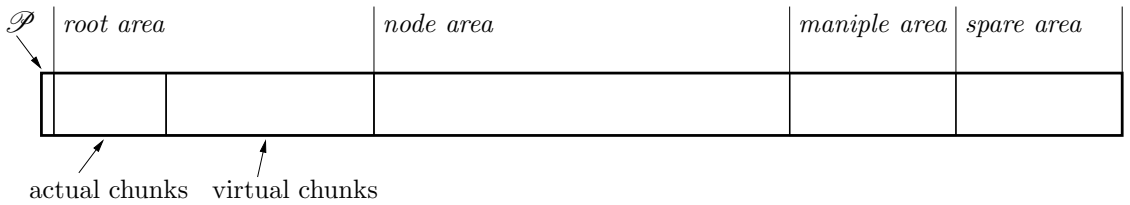

Figure 2: Memory layout of the array of n cells storing the flat implicit tree.

no particular order. As described in Section 4, we have $O(1)$ virtual chunks associated with each actual chunk. 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. Cache-obliviousness in this layer makes use of the static scheme of [31] but with the additional requirement of employing in-place algorithms to achieve implicitness as described Section 5.

- 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 the spare keys being handled differently. Each bucket tree is organized into $O(1)$ levels. The root is either an actual chunk or a virtual chunk in the top layer. The descending keys of the root are in the bottom layer. Specifically, the root has a single child, called *intermediate* node, containing $\Theta(\sqrt{k})$ bucket chunks and $\Theta(\sqrt{k})$ leaves as children. Each such leaf has associated a *maniple* of $\Theta(k)$ keys with further $\Theta(\sqrt{k})$ spare keys. The buckets are pairwise disjoint when considered as intervals. (The latter property is useful for routing keys inside a bucket.) More details are in Section 3.

While searching a key in this two-layer organization, we identify an actual chunk and, hence, its $O(1)$ associated virtual chunks (if they exist) in the top layer. Among them, we determine the root of the bucket in the bottom layer for completing 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 and $O(\log_B n)$ block transfers for any value of B.

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 $O(1)$ contiguous areas shown in Figure 2:

- The preamble $\mathscr{P}$ of $O(k)$ keys encoding $O(1)$ pointers and integers for bookkeeping purposes.
- The *root area* for the top layer storing first the keys in the actual chunks in a suitable order and, then, the virtual chunks in no particular order. Recall that these chunks are the roots of the bucket trees. The area may grow or shrink by k positions to its right.
- The area for the bottom layer, divided into *node area* (for the intermediate nodes and the leaves), *maniple area* and *spare area* for the bucket trees. The whole area for the bottom layer may grow or shrink by k positions to its left and by one position to its right.

Compared to the compactor lists in [19, 26], our memory management introduces the *compactor zones* for handling the above areas as each compactor zone stores objects of the same size in a *contiguous* segment of memory, which is crucial to achieve our bounds.

Although both compactor lists and zones succeed in storing "objects" of variable size without memory waste, a compactor zone is more suitable for cache-obliviousness. The bounds for the resulting data structure are stated below.

Theorem 1 *There exists a dynamic data structure storing n distinct keys that is both implicit and cache-oblivious and that supports the following operations with just $O(1)$ registers:*

- *searching with a cost of $O(\log n)$ time and of $O(\log_B n)$ block transfers per operation;*
- *inserting and deleting with an amortized cost of $O(\log n)$ time and of $O(\log_B n)$ block transfers.*

We prove Theorem 1 in the rest of the paper. In what follows, the bounds are intended to hold in the worst case, unless we explicitly mention them as amortized bounds. We define the size of the several entities (nodes, areas, zones, etc.) as the number of keys contained in each of them.

Also, we use a primitive for *cumulative shifts* for a set of contiguous keys $x_1, \ldots, x_m, y_1, \ldots, y_r$ in a segment X of $m+r$ cells, where r is a multiple of m. The primitive produces the sequence $y_1, \ldots, y_r, x_1, \ldots, x_m$ in X by running r/m phases in-place. Sequence $y_1, \ldots, y_r$ is partitioned into r/m equal parts. Before phase i, for $1 \leq i \leq r/m$, the segment X contains the first $i-1$ parts followed by $x_1, \ldots, x_m$ and then by the $r/m - i + 1$ remaining parts. After phase i, the segment X contains the first i parts followed by $x_1, \ldots, x_m$ and then by the $r/m - i$ remaining parts. During the phase, x_j is swapped with the jth key in the ith part, for $j = 1, 2, \ldots, m$. Data access is sequential, hence, the total cost is $O(r)$ time with $O(1)$ registers and $O(r/B)$ block transfers.

3 Bottom Layer: Buckets as Implicit Dynamic Forest

We describe the bottom layer storing the buckets in the form of an implicit dynamic forest. Each bucket is a 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 problem. In case of bucket overflows (too many keys) and bucket underflows (too few keys), we need splitting the bucket or merging it with a neighbor bucket, while preserving implicitness and cache-obliviousness.

3.1 Bucket organization

As previously mentioned, each bucket is a tree made up of bucket chunks and spare keys, except for the root, which is either an actual or a virtual chunk. 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 $4\sqrt{k}$ bucket chunks). 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]) defining the weight $w(u)$ as the number of keys in the leaves that are children of u. We keep on u the invariant $4k^2 \leq w(u) \leq 16k^2$. As a result, the number of chunks in u ranges from its minimum weigth divided by the largest number of keys in leaf to its its maximum weigth divided by the smallest number of keys in leaf, that is, from $\sqrt{k}$ to $16\sqrt{k}$ chunks.

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 are larger than those 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 (also mini-chunks are disjoint when considered as intervals). Hence, each maniple varies by a mini-chunk at a time. In order to manage the insertion and deletion of individual keys, we associate the $\Theta(\sqrt{k})$ spare keys with the last $\sqrt{k}$ mini-chunk in each maniple. From 1 to 5 spare keys are associated with each mini-chunk, and they are contained in the interval represented by the mini-chunk. However, we cannot encode in z the pointers to the $\Theta(\sqrt{k})$ spare keys in z. We encode them in the first $\Theta(\sqrt{k})$ chunks of the corresponding leaf v, which is sufficiently large to allow this encoding with its keys.

We keep the above invariants on the number of keys during the updates. We show later that inserting a key is tantamount to incrementing by 1 the number of spare keys in a maniple z. If a mini-chunk in z has associated 6 spare keys, we suitably redistribute the keys in z. If z has associated a total of $5\sqrt{k}+1$ 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.

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. 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. Finally, we 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 the ideas sketched above in the rest of the section. In Section 3.2, we describe how to allocate the keys in memory and, in Section 3.3, we focus on the internal organization of the trees. In Section 3.4, we show how to implement the search and update operations on the buckets using the techniques introduced in Sections 3.2–3.3. Finally, in Section 3.5, we provide the analysis of our algorithms.

3.2 Memory management of the trees

We postpone the layout of the roots to Sections 4 and 5. Here we discuss the layout of the intermediate nodes, the leaves, the maniples, and the spare keys in the bottom layer. We store the spare keys in no particular order in the spare area. Using *compactor zones* (or, simply, *zones*) we accommodate the intermediate nodes and the leaves in the node area and the maniples in the maniple area. The proposed primitives are invoked when a node or a maniple changes size and so its keys must be suitable relocated in memory. What we describe next for the nodes holds also for the maniples.

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 the beginning of each zone. In this case, we maintain the property that the node is stored

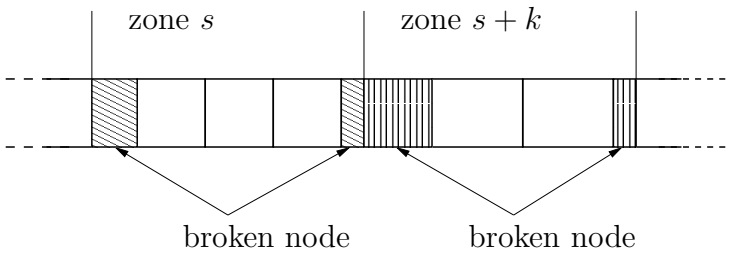

Figure 3: Compactor zones s and $s+k$.

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 3). The (encoded) pointer to a broken node contains extra $O(\log k)$ bits to encode that value of s_1.

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* 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 the cumulative shifts 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. We search one key for each of the latter nodes re-encoding the $O(1)$ pointers to it that are identified through the search. We also re-encode the starting position of zone s. The cost of a rotation is $O(k+m)$ time and $O(\lceil (k+m)/B \rceil)$ block transfers, plus the cost of $O(1)$ searches. We now give more details on the zones.

Node area. It contains $15\sqrt{k} + 1$ compactor zones in increasing order of index 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 exchange these $O(k\sqrt{k})$ keys by cumulative shifts in $O(k^2)$ time and $O(\sqrt{k}\lceil k\sqrt{k}/B \rceil)$ block transfers, with the initial portion of the broken node in s followed by w and by zone $s+k$. Specifically, we apply a cumulative shift to each of the $O(\sqrt{k})$ chunks thus involved (fixing $m = k$ and r equal to the rest of the keys, with r a multiple of m). 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 and update the new starting positions

of zone s'. At the end, we have w to the right of zone $16k\sqrt{k}$, that is, between the node area and the maniple area.

- 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 1 *The cost of* ExtractNode *and* InsertNode *in the node area is* $O(k^2)$ *time and* $O(\sqrt{k}\lceil k\sqrt{k}/B\rceil)$ *block transfers, plus the cost of* $O(\sqrt{k})$ *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.

Lemma 2 *The cost of* TransferChunk *in the node area is* $O(k\sqrt{k})$ *time and* $O(\sqrt{k}\lceil k/B\rceil)$ *block transfers, plus the cost of* $O(\sqrt{k})$ *searches.*

Maniple area. It contains $4\sqrt{k}+1$ compactor zones in increasing order of index s. Each zone s contains the maniple 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 that we will employ to implement the update operations in Section 2:

- 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 3 *The cost of* ExtractManiple, InsertManiple *and* TransferKeys *in the maniple area is* $O(k\sqrt{k})$ *time and* $O(\sqrt{k}\lceil k/B\rceil)$ *block transfers, plus the cost of* $O(\sqrt{k})$ *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 4 *Extracting or inserting a spare key in the spare area has a cost of $O(k)$ time and $O(\lceil k/B \rceil)$ block transfers 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 4, giving a total cost of $O(km)$ time and $O(m\lceil k/B \rceil)$ block transfers plus the cost of $O(m)$ searches.

3.3 Internal structure of the trees

The internal structure of the bucket trees allows for quick searching and updating even though their size is $\Theta(k^2)$. We assume that the algorithms described in Section 3.2 provide the necessary memory support for what we discuss below. Hence, the analysis of the costs in this section does not account for the costs of the memory management given in Section 3.2.

Intermediate nodes. The simplest organization is that of the intermediate nodes. Given an intermediate node u, containing $t = \Theta(\sqrt{k})$ chunks $c_1, \ldots, c_t$, we keep a directory in the first $2t$ positions of u containing the smallest and the greatest key of $c_1, \ldots, c_t$, respectively, in this order. If the keys in a chunk c_j change, it is a simple task to update c_j and the directory. Routing a search for key x in u examines some of the $O(t)$ keys in the directories to identify a pair of keys x_1 and x_2 such that $x_1 \leq x \leq x_2$. These two keys belongs to at most two consecutive chunks c_j and c_{j+1} that are accessed by simple offsetting. We also support the following primitives for an intermediate node u:

- BUILDDIRECTORY(u) when the keys of u are in sorted order. For $j = 1, 2, \ldots, t$, we shift the leftmost and rightmost key in each chunk c_j.
- INSERTCHUNK(c, u) assumes that c is contiguous to u in memory. We perform a cumulative shift of c without its leftmost and rightmost key (fixing $m = k - 2$), until it reaches its position in u, say after chunk c_j. Then, we shift the two aforementioned keys of c to their positions in the directory of u, just after those of c_j.
- EXTRACTCHUNK(c, u) is analogous to INSERTCHUNK.

Lemma 5 *Given an intermediate node u, routing a key in u using its directory requires $O(k)$ time and $O(\lceil k/B \rceil)$ block transfers.* BUILDDIRECTORY *costs $O(k^2)$ time and $O(\sqrt{k}\lceil k\sqrt{k}/B \rceil)$ block transfers.* INSERTCHUNK *and* EXTRACTCHUNK *require $O(k\sqrt{k})$ time and $O(\lceil k\sqrt{k}/B \rceil)$ block transfers.*

Leaves. The leaves undergo a more involved organization because they delegate the insertion and deletion of individual keys to their associated maniples (see Figure 4). In particular, when inserting a key into a leaf v, we start a sequence of internal rotations that quickly make room for the new key and kick out the greatest 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 greater than any other key in v. Reducing the problem of inserting or deleting a key in v to that of handling the operation suitably in its maniple z is the rationale for the rotational machinery described next.

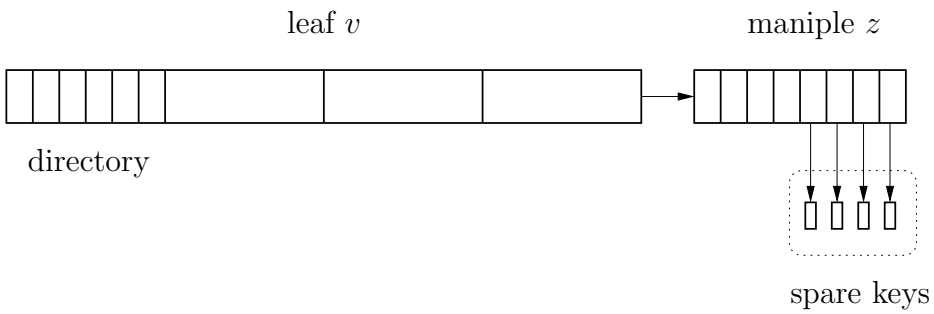

Figure 4: A leaf v, its associated maniple z with the spare keys.

We give a two-step description, in which we first describe how to organize the keys in the leaf v for working in main memory, and then how to make this organization cache-oblivious in $O(\log_B n')$ block transfers. For the sake of discussion, let's 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 (extracted as the greatest from the chunk to the left of c') while extracting the maximum key (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, we cannot afford the $O(k)$ cost of shifting all the keys in the remaining chunks c', as they can be $\Theta(\sqrt{k})$ in number.

Consequently, we organize each chunk c' in v as follows, denoting by x_1 the minimum key to be inserted into c' and by x_2 the maximum key to be extracted from c' in the generic iteration step (the symmetric operation of deleting x_1 and inserting x_2 is analogous):

1. Keys $a_1, a_2, \ldots, a_k$ are kept *rotated* by an *offset* r, occurring as $a_{r+1} \cdots a_k \cdot a_1 \cdots a_r$ in c'.
2. Inserting x_1 while extracting $x_2 = a_k$ gives $a_{r+1} \cdots a_{k-1} \cdot x_1 a_1 \cdots a_r = a'_{r+2} \cdots a'_k \cdot a'_1 \cdots a'_r a'_{r+1}$.
3. 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).
4. $a_{\sqrt{k}+1}, \ldots, a_{k-\sqrt{k}}$ encode the $O(1)$ pointers and integers for the structural information in c'.

Preserving the above organization crucially relies on keeping condition 3 valid. When the latter condition is violated in a chunk, we reset its rotation with a cumulative shift taking $O(k)$ time (fixing $m = \sqrt{k}$). However, this organization is not yet suitable for cache-obliviousness because the sequence of rotations previously mentioned access at least one key per traversed chunk, which is a problem for small values of the (unknown) block size B. We observe that the accessed keys in this task are always kept in the first or the last mini-chunk of each chunk c', namely, $a_1, \ldots, a_{\sqrt{k}}$ and $a_{k-\sqrt{k}+1}, \ldots, a_k$ by condition 3. To obtain cache-obliviousness, we form a directory at the beginning of leaf v with these two mini-chunks for each chunk. We obtain a larger directory than that of intermediate nodes, still the number of keys in the directory being $O(k)$. As a result, the rotation of each chunk c' will always involve its keys occurring inside that directory. Routing a search key x inside v uses the directory in a way similar to that described for the intermediate nodes. We also support the following primitives for a leaf v:

- BUILDDIRECTORY(v) when the keys of u are in sorted order. For each chunk c in v, we apply a cumulative shift (fixing $m = \sqrt{k}$) to the leftmost and the rightmost mini-chunk in c.

- INSERTCHUNK(c, v) assumes that c is contiguous to v in memory. We perform a cumulative shift of c without its leftmost and rightmost mini-chunk (fixing $m = k - 2\sqrt{k}$), until it reaches its position in v, say after chunk c'. Then, we shift the two aforementioned min-chunks of c to their positions in the directory of v, just after those of c'. Note that the rotation for c has offset 0.

- EXTRACTCHUNK(c, v) is analogous to INSERTCHUNK.

Lemma 6 *Given a leaf v, routing a key in v using its directory requires $O(k)$ time and $O(\lceil k/B \rceil)$ block transfers.* BUILDDIRECTORY *costs $O(k^2)$ time and $O(\sqrt{k}\lceil k\sqrt{k}/B \rceil)$ block transfers.* INSERTCHUNK *and* EXTRACTCHUNK *require $O(k\sqrt{k})$ time and $O(\lceil k\sqrt{k}/B \rceil)$ block transfers.*

We postpone the cost of how to maintain the invariant for conditions 1–4 to Section 3.5.

Maniples and spare keys. Given a leaf v, its associated maniple z is divided into mini-chunks. The keys in z are all greater than those in v, and they are kept in sorted order among the mini-chunks. The last $\sqrt{k}$ mini-chunks in z 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. These pointers are encoded in the first $5\sqrt{k}$ chunks of v, specifically, in the keys of v mentioned in condition 4 on the invariant for the leaves. The association is static and, independently of how many spare keys a mini-chunk has, we encode 5 pointers in v for its spare keys (some of the can be nil, encoded as 0). When searching in z, we decode at most 5 of these pointers, namely, those corresponding to the mini-chunk in which our search ends. We also support the following primitives:

- INSERTKEY(x, z) assumes that there are less than $5\sqrt{k}$ spare keys in total in z and that they are contained in the interval represented by their mini-chunk. Since z is in sorted order, we identify the mini-chunk c of z that encloses x. If c has less than 5 spare keys, we insert x into c incrementing its spare keys by one, encoding its pointer in the proper chunk of v. Otherwise, we extract the greatest key y of c and identify the closest mini-chunk c' to the right of c (or to its left) that has less than 5 spare keys. We shift by one position to the right (or to the left) all the keys between c and c' to make room for y in sorted order. Now, c' as one extra key to be inserted into it, incrementing its spare keys by one as described above.

- EXTRACTKEY(x, z): Analogous to INSERTKEY.

As we shall see, 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.

Lemma 7 *Given a maniple z and its leaf v, routing a key in z requires $O(k)$ time and $O(\lceil k/B \rceil)$ block transfers.* InsertKey *and* ExtractKey *take $O(k)$ time and $O(\lceil k/B \rceil)$ block transfers.*

3.4 Supporting the operations on the buckets

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

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 5. 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 6. 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 7 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 8 *Given the root of a bucket, searching x in the bucket takes $O(k)$ time and $O(\lceil k/B \rceil)$ block transfers.*

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 8. 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 3.3(Leaves) 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 3.3).

   (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 (as it is the smallest mini-chunk) by cumulative shifts (fixing $m = \sqrt{k}$). 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 jump to 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 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 cumulative shifts in these $O(k\sqrt{k})$ keys obtaining $v_1, v_2, c, z_1, z_2, s_1, s_2$, (we fix $m = k$ or $m = \sqrt{k}$ according to the cases).

iv. We apply BuildDirectory to v_1 and v_2, and then 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 merge its directory with the rest of its keys. Now the keys in u are sorted, 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 apply BuildDirectory to u_1 and u_2, and 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.

3.5 Analysis of the costs

We provide a detailed analysis for the amortized cost of the insertions. We closely follow the algorithmic scheme presented in Section 3.4 to show that the cost of each step is $O(k+\log n')$ time and $O(k/B + \log_B n')$ block transfers.

Step 1 is worst-case by Lemma 8 and by the fact the $O(1)$ chunks are accessed.

Step 2 is amortized. The rotations are all performed in the first $O(k)$ keys. What is more expensive is the reset of the rotation in the chunks of the leaf that otherwise would violate conditions 1–4. Resetting each chunk costs $O(k)$ time and $O(\lceil k/B \rceil)$ block transfers 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 and $O(\lceil k/B \rceil)$ block transfers.

Step 3 is amortized. Step 3a requires $O(k\sqrt{k})$ time and $O(\sqrt{k}\lceil k/B \rceil)$ block transfers because of decoding the pointers to the $\Theta(\sqrt{k})$ spare keys. Step 3b is worst-case by Lemma 7. Step 3c has the same cost as that of Step 3a, plus the cost of $O(\sqrt{k})$ searches by Lemma 4. Step 3d costs as Step 3c by Lemma 3 and the by the cost of the cumulative shift. In summary, the total cost of Step 3 is $O(k\sqrt{k})$ time and $O(\sqrt{k}\lceil k/B \rceil)$ block transfers 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 and $O(\lceil k/B \rceil)$ block transfers 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 1, that of Step 4b by Lemma 3 and Lemma 5. This gives an amortized cost of $O(k)$ time and $O(\lceil k/B \rceil)$ block transfers plus the cost of $O(1)$ searches. The cost of Step 4c is detailed as follows: that of Step 4(c)i is given by Lemma 3 and Lemma 4; Step 4(c)ii is $O(\sqrt{k}\,\lceil k/B \rceil)$ for the $O(\sqrt{k})$ repeated insertions; Step 4(c)iii is $O(k\lceil \sqrt{k}/B \rceil)$ for the cumulative shifts; Step 4(c)iv by Lemma 1, Lemma 3 and Lemma 5; Step 4(c)iv by Lemma 3. The total cost of Step 4c is $O(k^2)$ time and $O(\sqrt{k}\,\lceil k\sqrt{k}/B \rceil)$ block transfers plus the cost of $O(\sqrt{k})$ searches, giving an amortized cost of $O(k)$ time and $O(\lceil k/B \rceil)$ block transfers 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 1 and Lemma 5; that of Step 5b by Lemma 1; finally, that of Step 5c by $O(k^2)$ time and $O(\sqrt{k}\,\lceil k/B \rceil)$ block transfers for the merge done with $O(\sqrt{k})$ repeated insertions, and by Lemma 1, Lemma 4 and Lemma 5. 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.

Theorem 2 *In the bottom layer, each bucket contains $\Theta(k^2)$ keys at any time. Searching a bucket takes $O(k)$ time and $O(\lceil k/B \rceil)$ block transfers. The amortized cost of updating a bucket is $O(k)$ time and $O(\lceil k/B \rceil)$ block transfers per insert/delete operation plus the cost of $O(1)$ searches. 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. We describe in this section how to handle efficiently the super-root in main memory. We discuss how to make it cache-oblivious in Section 5.

4.1 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 associated at most $\alpha = O(1)$ virtual chunks, 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 (see Section 3.3 and Theorem 2). The number a of actual chunks changes only when rebuilding (see Section 6) 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 9 *Routing a key in the super-root takes $O(\log n')$ time to identify the correct bucket to access.*

4.2 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 by applying the notion of density [5, 10, 23] to R. 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., underflows) if $d(u) > \tau_s$ (resp., $d(u) < \rho_s$) and recursively at least one of its children either overflows or underflows [5, 10, 23].

4.3 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 (see Section 3.3 and Theorem 2). 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.

Redistribution of chunks for u. 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. 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 $\rho_s < \rho_0 < \tau_0 < \tau_s$.

4.4 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). 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 (see Theorem 2), and thus the amortized cost per operation on v is

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

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 (see Theorem 2), and thus the amortized cost per operation on v is

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

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 Top Layer: Cache-Obliviousness

We show how to modify the organization described in Section 4 so that it becomes cache-oblivious with optimal bounds as well. We recall that the top layer collects all the actual chunks and the virtual chunks, which are the roots of the buckets in the bottom layer discussed in Section 3. The memory layout in the root area is simple; first, all the a actual chunks in sorted order, where $a = 2^h = \Theta(n'/\log^2 n')$ is *always* a power of two; then, all the virtual chunks in no particular order. 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 root area resizes by k positions to the right at a time to handle one more or one less chunk after bucket splitting or merging (see Section 3.3 and Theorem 2). The number a of actual chunks changes only when rebuilding or when performing a full redistribution of actual and virtual chunks.

Here we discuss how to make cache-oblivious the access to the actual chunks since the virtual chunks are not much of a problem. In the following, we assume that the rightmost actual chunk is treated separately and we are left with $a - 1 = 2^h - 1$ actual chunks. The main idea is to build an internal directory for the root similarly to what done for internal nodes u in Section 3.3. We refer to actual chunks both when they contain k keys or when they have only $k - 2$ keys since their smallest and greatest key are in the directory. It is worth noting that the directory is permuted while the actual chunks are kept in increasing order. Moreover, the keys in the directory are located between the actual chunks and the virtual chunks. Conceptually, we treat each pair of keys in the directory as a single interval. When searching a key x, we compare it to each intervals by exploiting the fact that the intervals are disjoint: either x is inside the interval, or it is to the left or the right of the interval. If we have cache-oblivious access to the directory, we can access an actual chunk and its associated virtual chunks in $O(k)$ comparisons and $O(\lceil k/B \rceil)$ block transfers.

We focus therefore on how to permute the directory assuming to have just $2^h - 1$ keys (rather than $2^h - 1$ pairs of keys encoding disjoint intervals) for the sake of discussion.

5.1 Building the VEB-permutation

We define the *Van Emde Boas permutation* (shortly, VEB-permutation) of $2^h - 1$ keys following Prokop's recursive scheme for Van Emde Boas trees [31]. Suppose to have a complete binary tree with $h \geq 1$ levels and $2^h - 1$ nodes, where $h = 1$ indicates that the tree has just one node. The nodes store the sorted sequence of $2^h - 1$ keys in symmetric order. Since we do not keep this tree anywhere, we permute the keys recursively according to the tree structure. In what follows, let A denote the memory segment hosting these $2^h - 1$ keys with the scheme of the VEB-permutation and let VEB-tree indicate the complete binary tree mentioned above. If $h = 1$, we simply store the key associated with the unique node in the VEB-tree. Otherwise, let $h_T = \lceil h/2 \rceil$ and $h_B = h - h_T$. We recursively store the $2^{h_T} - 1$ entries in the top tree of height h_T in the first $2^{h_T} - 1$ cells of A. Then, for $i = 0, 1, \ldots, 2^{h_T} - 1$, we recursively store the $2^{h_B} - 1$ entries of the bottom tree number i (from left to right) in the ith portion of A (i.e., starting from $A[2^{h_T} + i \cdot (2^{h_B} - 1)]$).

We now describe how to build a VEB-permutation *in-place* in $O(2^h h) = O(a \log a) = O(n'/k)$ comparisons and block transfers. (We can achieve a better bound but this does not improve the final bounds.) We first run heapsort on the sequence of $2^h - 1$ entries.

We then apply the recursive scheme mentioned above. The base case for $h = 1$ is easy to handle, so let's suppose $h > 1$ and compute h_T and h_B. For $j = 1, \ldots, 2^{h_T} - 1$, we swap the key, say t_j, in position j with the key in position $j \cdot 2^{h_B}$. Now, the keys associated with the top tree are in order in the first $2^{h_T} - 1$ positions of A. We now sort the keys in the rest of A (i.e., $A[2^{h_T} \ldots 2^h - 1]$) as described next, and recursively apply our construction to the keys in $A[1, \ldots 2^{h_T} - 1]$ (the top tree) and, for $i = 0, 1, \ldots, 2^{h_T} - 1$, to the keys in $A[2^{h_T} + i(2^{h_B} - 1) \ldots 2^{h_T} + (i + 1) \cdot (2^{h_B} - 1) - 1]$ (the bottom subtree number i). To sort the keys in $A[2^{h_T} \ldots 2^h - 1]$, we exploit the fact they are almost sorted, apart from the keys $t_1, \ldots, t_{2^{h_T}}$ (the latter are in relative order). We perform some cumulative shifts. We start from $t_{2^{h_T}}$, which is shifted until it meets $t_{2^{h_T}-1}$. In general, for $j < 2^{h_T}$, we have shifted and collected $t_{j+1}, \ldots, t_{2^{h_T}}$ to the right of t_j, and we want to shift the former keys near the latter. We execute a cumulative shift (fixing $m = 2^{h_T} - j$) to move $t_{j+1}, \ldots, t_{2^{h_T}}$ as closest as possible to t_j; then, we move t_j towards them, thus obtaining $t_j, t_{j+1}, \ldots, t_{2^{h_T}}$ collected together, as expected.

Lemma 10 *A VEB-permutation for $2^h - 1$ keys can be built in $O(2^h h) = O(a \log a) = O(n'/k)$ comparisons and block transfers, where $a = 2^h = O(n'/k^2)$.*

Proof: The cost of this construction is asymptotically upper bounded by the solution to the recurrence $C(2^h - 1) = C(2^{h_T} - 1) + 2^{h_T} C(2^{h_B} - 1) + O(2^h)$. Note that putting $t_1, \ldots, t_{2^{h_T}}$ in their suitable order (w.r.t. the other keys) gives the sorting and costs only $O(2^h)$ as we scan A from the right to the left. For a suitable constant d, that solution is $C(2^h - 1) \leq d 2^h h$ by substitution on the right hand side of the recurrence. We remark that the construction of the VEB-permutation is not fully in-place as it uses the recursion. We therefore handle directly the non-tail recursion by using a stack storing only the pairs of values h_T, h_B thus found at each recursive level. Since h_T and h_B can be encoded in $O(\log h)$ bits and we keep $O(\log h)$ of them during the recursion, the total number of bits for the full stack is $O(\log^2 h) = o(\log n')$. We can handle this stack in $O(1)$ registers in constant time per operation, using simple arithmetic operations for push and pop. We do not violate the assumptions of the RAM uniform cost. Using this "implicit" stack in a register and additional $O(1)$ register, we are able to build in-place the VEB-permutation in $O(n'/k)$ comparisons and block transfers. □

5.2 Searching the VEB-permutation

We now get to the main point, namely, how to search a key x in the VEB-permutation for $2^h - 1$ keys. In [31] it is shown that traversing the path from the root to a node takes $O(\log_B(2^h - 1)) = O(\log_B n')$ block transfers. We show how to do it without the extra information required in [10], which is not permitted in the model for implicit data structures. We use procedure FIND(x, h) in Figure 5 to achieve our goal. Before invoking it, we know that the segment $M = A[i \ldots i + 2^h - 2]$ of $2^h - 1$ keys corresponds to a subtree S of the VEB-tree. We also know that $A[i]$ is the key in the root of S, and that the breadth-first number of $A[i]$ in the VEB-tree is *bfs*. (Initially, S is the VEB-tree, $M = A$, and so $i = 1$ and $bfs = 1$.)

When FIND(x, h) completes, it has traversed the path from the root of S to one leaf v of S and i has reached the position of the last key in M. The routing of key x must go on either to the left or to the right of v in the rest of the VEB-tree, and we crucially know the

```
FIND(x, h):
 1: IF h = 1 THEN
 2:    IF x ≤ A[i] THEN
 3:       bfs ← bfs × 2
 4:    ELSE
 5:       bfs ← bfs × 2 + 1
 6: ELSE
 7:    h_T ← ⌈h/2⌉, h_B ← h − h_T
 8:    FIND(x, h_T)
 9:    rank ← bfs mod 2^{h_T}
10:    i ← i + (2^{h_B} − 1) × rank + 1
11:    FIND(x, h_B)
12:    bfsroot ← bfs/2^{h_B+1}
13:    rankroot ← bfsroot mod 2^{h_T}
14:    i ← i + (2^{h_B} − 1) × (2^{h_T} − 1 − rankroot)
```

Figure 5: Procedure FIND to search x in a VEB-permutation of $2^h - 1$ keys stored.

bfs of the next node to visit. In any case, *bfs* mod 2^h gives the rank of x among the keys in M. Identifying the position j such $A[j] = x$ (if any) is a minor modification.

We now show how to keep the invariant on FIND(x, h) by induction on h (see Figure 5). If $h = 1$, this is immediate. Let's take the case of $h > 1$. We compute the number of levels h_T and h_B for the top and bottom subtrees of S, respectively called S_T and S_B. We want to identify their corresponding segments M_T and M_B in A. First, note that $M_T = A[i \dots i + 2^{h_T} - 2]$. So, we can invoke directly FIND(x, h_T) to route x in S_T. By induction, $rank = bfs \bmod 2^{h_T}$ tells us the number of S_B (starting from 0 and going from left to right in S). Since each bottom subtree has size $2^{h_B} - 1$, we can infer that M_B starts at position $i + (2^{h_B} - 1) \cdot rank + 1$ of A. So we update i to this new value. We can invoke FIND(x, h_B) to route x in S_B. By induction, we correctly compute *bfs* in the VEB-tree for the next node below S_B that is also the next node below S. In order to preserve the invariant, we need to update i so that it is the last position of M in A. We have to compute *rank* again, since we cannot keep this value due to the implicitness. We therefore, compute the breadth-first number of the root of S_B, which is the current value of *bfs* divided by 2^{h_B+1} (the exponent $h_B + 1$ comes from the fact that *bfs* refers to one level below the leaves of S_B). We then take the modulo of the resulting breadth-first number in *rankroot*, as we did for *rank* (indeed lines 9 and 13 compute the same value but at different times in the recursion; so we cannot keep the values of the variables *rank* in the recursive levels to obtain an in-place algorithm). We finally increment i noting that we have to jump over the keys of $(2^{h_T} - 1 - rankroot)$ bottom subtrees of size $2^{h_B} - 1$. As a result, we keep the invariant for S. Finally, we observe that we traversed a path from the root of S to a leaf of it. The only accessed keys in A are at line 2 updating *bfs*, so that the next key to be compared with x is either the left child or the right child of $A[i]$ but not both.

Lemma 11 *A VEB-permutation for $2^h - 1$ keys can be searched in $O(h) = O(\log n')$ comparisons and $O(h/\log B) = O(\log_B n')$ block transfers.*

Proof: The cost of searching is asymptotically upper bounded by the solution to the recur-

rence $C(2^h-1) = C(2^{h_T}-1) + C(2^{h_B}-1) + O(1)$. The crucial observation is that FIND traverses a downward path from the root to an internal node, whose length is asymptotically bounded by $C(2^h-1)$. For suitable constant d_1 and d_2, the solution is $C(2^h-1) \leq d_1 h - d_2$ by substitution on the right hand side of the recurrence. Hence, a VEB-permutation for 2^h-1 keys can be searched in $O(h)$ comparisons and $O(h/\log B)$ block transfers. □

5.3 Maintaining the VEB-permutation

To maintain the VEB-directory when redistributing the actual and virtual chunks like in Section 4.3, we are allowed to pay a search in the VEB-directory for each chunk involved in the redistribution with its two smallest and greatest keys, replacing them when the corresponding chunk is exchanged.

When halving the number of actual chunks, we must move the $a/2$ largest chunks to the right of the VEB-directory (thus halving the VEB-directory), adding them to the list of virtual chunks mentioned in Section 4.3. We first run heapsort on the VEB-directory to get it in decreasing order. We then merge the $a/2$ largest chunks with the left half of the sorted VEB-directory. We then apply a cumulative shift by $k \cdot a/2$ positions to the left of the VEB-directory (fixing $m = a/2$ and $r = k \cdot a/2$)), and rebuild the VEB-directory as described in Section 5.1. Finally, we transform these $a/2$ largest chunks into virtual chunks and complete the redistribution described in Section 4.3 (we replace keys in the VEB-directory when redistributing).

When doubling the number of actual chunks, we have the opposite situation, namely, the a smallest virtual chunks to the right of the VEB-directory send their smallest and greatest key in the VEB-directory, and their $k-2$ remaining keys become actual chunks to the left of the VEB-directory. We process these virtual chunks one by one in order, assuming that each chunk c of them is adjacent to the VEB-directory (a simple exchange of two virtual chunks can achieve this as we have the virtual chunks linked in order). We take the greatest key in c and move it adjacent to the smallest key. We then exchange incrementally the other $k-2$ keys with the first $k-2$ keys currently in the VEB-directory (note that, because of this, the VEB-directory is scrambled). At the end, we transform these a virtual chunks into actual chunks and rebuild the VEB-directory from scratch. We then complete the redistribution described in Section 4.3.

Lemma 12 *A VEB-permutation for 2^h-1 keys can be updated when halving or doubling the number of actual chunks in no more than $O(a \log a + ak)$ time and block transfers.*

5.4 Analysis

The analysis of the amortized cost for an internal node v on depth $s+1$ is a simple extension of that given in Section 4.4, in which the cost in terms of block transfers is $(|act(u)| + |vir(u)|)\, O(\lceil k/B \rceil)$, giving rise to

$$\frac{(|act(u)| + |vir(u)|)\, hO(\lceil k/B \rceil)}{\alpha |act(v)| (\tau_\infty - \tau_0)\Theta(k^2)} = O\left(\left\lceil \frac{k}{B} \right\rceil \frac{h}{k^2}\right).$$

When v is the root, however, we must replace the cost of $(|act(u)| + |vir(u)|)\, O(\lceil k/B \rceil)$ by $O(a\lceil k/B \rceil)$ plus the cost of rebuilding and maintaining the VEB-permutation as stated in

Lemma 10 and 12, which is bounded by $O(a \log a + ak) = O(n'/k)$ in total. As a result, noting that $|act(v)| = \Theta(n'/k^2)$ when v is the root, the amortized cost for the root is

$$\frac{O(n/k)}{\alpha\Theta(n/k^2)(\tau_\infty - \tau_0)\Theta(k^2)} = O(h/k) = O(1).$$

Since there are h levels, of which one is for the root, we obtain a total amortized cost of $O(h^2/k^2\lceil k/B \rceil) = O(\lceil k/B \rceil)$ block transfers.

Theorem 4 *In the top layer, routing a key to its bucket takes $O(\log n')$ comparisons and $O(\log_B n')$ block transfers. Inserting or deleting a chunk in any position of the root area has an amortized cost of $O(k)$ comparisons and of $O(\lceil k/B \rceil)$ block transfers plus the cost of $O(1)$ searches.*

6 Rebuilding and Final Analysis

In our algorithms we assumed that $n'/4 < n < n'$ 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 (this is important to avoid triggering a sequence of recursive rebuilding). 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 operatiosn, since we keep $k = \Theta(\log n')$ unchanged for all the rebuilding). Apart from the rebuilding, the cost of search, insert and delete is that of Theorems 2–4. From these costs, it follows our main result, Theorem 1.

7 Conclusions

In this paper we presented the flat implicit tree, which avoids space wasting by just using the plain array of n cells for the keys and, simultaneously, provides optimal data access at any level of the memory hierarchy. The flat implicit tree is the first optimal data structure obtaining $O(\log n)$ time for search and update in an array of n cells since the invention of the heaps in the sixties. The flat implicit tree is also cache-oblivious achieving $O(\log n)$ comparisons and $O(\log_B n)$ block transfers for any block size B, where the value of B and the capacity of the memory level are unknown to the algorithms operating in the model.

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] Michael A. Bender, Richard Cole, and Rajeev Raman. Exponential structures for efficient cache-oblivious algorithms. *Lecture Notes in Computer Science*, 2380:195–??, 2002.

[7] Michael A. Bender, Ziyang Duan, John Iacono, and Jing Wu. A locality-preserving cache-oblivious dynamic dictionary. In *Proceedings of the 13th Annual ACM-SIAM Symposium On Discrete Mathematics (SODA-02)*, pages 29–38, New York, January 6–8 2002. ACM Press.

[8] 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.

[9] 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.

[10] 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.

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

[12] Erik D. Demaine. Cache oblivious algorithms and data structures. Lecture notes for EEF Summer School on Massive Data Sets, 2002.

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

[14] 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.

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

[16] 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.

[17] Gianni Franceschini and Roberto Grossi. Optimal cache-oblivious implicit dictionaries. Submitted to conference, 2003.

[18] Gianni Franceschini and Roberto Grossi. Optimal space-time dictionaries over an unbounded universe with flat implicit trees. Technical report TR-03-03, January 30, 2003.

[19] 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.

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

[21] 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.

[22] M. Frigo, C. E. Leiserson, H. Prokop, and S. Ramachandran. Cache-oblivious algorithms. In IEEE, editor, *40th Annual Symposium on Foundations of Computer Science: October 17–19, 1999, New York City, New York,*, pages 285–297, 1109 Spring Street, Suite 300, Silver Spring, MD 20910, USA, 1999. IEEE Computer Society Press.

[23] 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.

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

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

[26] 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.

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

[28] 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.

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

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

[31] H. Prokop. Cache-oblivious algorithms. Master's thesis. MIT, Cambridge, MA, 1999.

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

[33] 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.

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

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

[36] Ian H. Witten, Alistair Moffat, and Timothy C. Bell. *Managing Gigabytes: Compressing and Indexing Documents and Images*. Morgan Kaufmann Publishers, Los Altos, CA 94022, USA, second edition, 1999.

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