Optimal Worst-Case Operations for Implicit Cache-Oblivious Search Trees

Gianni Franceschini and Roberto Grossi

Dipartimento di Informatica, Università di Pisa, via Buonarroti 2, 56127 Pisa, Italy
Fax +39-050-2212726, {francesc,grossi}@di.unipi.it

Abstract. We close an open issue on dictionaries dating back to the sixthies. An array of n keys can be sorted so that searching takes $O(\log n)$ time. Alternatively, it can be organized as a heap so that inserting and deleting keys take $O(\log n)$ time. We show that these bounds can be simultaneously achieved in the worst case for searching *and* updating by suitably maintaining a permutation of the n keys in the array. The resulting data structure is called implicit as it uses just $O(1)$ extra memory cells beside the n cells for the array. The data structure is also cache-oblivious, attaining $O(\log_B n)$ block transfers in the worst case for any (unknown) value of the block size B, without wasting any single cell of memory at any level of the memory hierarchy.

1 Introduction

In this paper we consider the classical dictionary problem in which a set of n distinct keys $a_1, a_2, \ldots, a_n$ is maintained over a total order, where the only operations allowed on the keys are reads/writes and comparisons using the standard RAM model of computation [1]. The dictionary supports the operations of searching, inserting and deleting an arbitrary key x.

Implicit dictionaries solve the problem by maintaining a plain permutation of $a_1, a_2, \ldots, a_n$ to encode the data structures [17]. When employed in this context, heaps [19] 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 [15]. A longstanding question is whether there exists an organization of the keys in an array of n cells combining the best qualities of sorted arrays and heaps, so that each operation requires $O(\log n)$ time. Previous work since the sixties did not achieve polylog time in both searching and updating. We refer the reader to [10] for a history of the problem.

The first milestone in this direction is the implicit AVL tree in the eighties, showing for the first time that polylog time is possible, namely $O(\log^2 n)$, by encoding bits in chunks of $O(\log n)$ permuted keys [16]. It was conjectured a bound of $\Theta(\log^2 n)$ because $\Theta(\log n)$ pointers of $\Theta(\log n)$ bits are decoded/encoded in the worst case to execute an operation in the implicit AVL tree.

The second milestone is the implicit B-tree, attaining $O(\log^2 n/ \log\log n)$ time [11]. Notwithstanding the small improvement in main memory, this recent result disproved the conjecture of the eighties, making viable the possibility of getting a bound of $O(\log n)$. The implicit B-tree uses nodes of relatively large fan-out that are augmented with a permuted directory to support fast searching inside each node. For a *known* block size $B = \Omega(\log n)$, it supports the operations in $O(\log_B n)$ block transfers like regular B-trees, while scanning r contiguous elements requires $O(\log_B n + r/B)$ block transfers.

The subsequent results leading to the flat implicit tree [9] probably represents the third milestone. It is the first implicit data structure with optimal $O(\log n)$ time for searching and $O(\log n)$ amortized time for updating. Specifically, the result of $O(\log n \log\log n)$ in [7] uses exponential trees of height $O(\log\log n)$, exploiting in-place algorithms to amortize the bounds and introducing different kinds of chunks of $O(\log n)$ contiguous keys to delay the expensive reorganizations of the updates. The result in [10] obtains $O(\log n)$ amortized time with a two-layer tree of constant height (except very few cases), adapting the redistribution technique of [3, 14] to the implicit model. Its cache-oblivious evolution in [8] attains the amortized bounds of $O(\log_B n)$, where the cache-obliviousness of the model lies in the fact that the block transfer size B is unknown to the algorithms operating in the model [13]. The top layer uses a van Emde Boas permutation [13] of the keys as a directory, and the bottom layer introduces compactor zones to attain cache-obliviousness. Compared to implicit B-trees, the update bounds are amortized and scanning is not optimal. On the other hand, achieving an optimal scanning is still an open problem in explicit cache-oblivious dictionaries even with amortized update bounds of $O(\log_B n)$. The implicit B-tree attains this goal with worst-case bounds as it is aware of the block size B.

In this paper we focus on the worst-case complexity of implicit dictionaries. The best bound is that of $O(\log^2 n/\log\log n)$ with the implicit B-trees. For explicit cache-oblivious data structures, the best space occupancy in [5] is $(1+\epsilon)n$ cells for any $\epsilon > 0$ with an $O(1 + r/B)$ scanning cost for r keys, but the update bounds are amortized, whereas the worst-case result in [4] uses more space. Here, we propose a new scheme for implicit data structures that takes $O(\log n)$ time and $O(\log_B n)$ block transfers in the worst case for any unknown B, as in the cache-oblivious model. The optimality of our data structure is at any level of the memory hierarchy as it uses just $n + O(1)$ cells. This closes the problem of determining a permutation of the keys in an array, so that both searching and updating are logarithmic in the worst case as explicit dictionaries.

We introduce new techniques to design our data structures. First, we use some spare keys and some chunks, called filling chunks, to allocate nodes of the tree in an implicit way. When we actually need a chunk, we replace the filling chunk with the routing chunk, and relocate the filling chunk. We also design a bottom layer that can be updated very quickly. We reuse techniques from previous work, but we apply them in a novel way since we have to perform the memory management of the keys in the array. Consequently, our algorithms are slightly more involved than algorithms for explicit data structures, as the latter assume to have a powerful memory manager performing the "dirty" work for them in a transparent way. Instead, we have to carefully orchestrate data movement as we cannot leave empty slots in any part of the array. In the full paper, we show how to extend to our data structure to a multiset, namely, containing some repeated keys.

The paper is organized as follows. In Section 2, we review some basic techniques that we apply to implicit data structures. We then describe our main data structure in two parts, in Section 3–4, putting them together in Section 5 for the sketch of the final analysis of the supported operations.

2 Preliminary Algorithmic Tools

We encode data by a pairwise (odd-even) permutation of keys [16]. 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 with the rule: if the ith bit is 0, then $\min\{x_i, y_i\}$ precedes $\max\{x_i, y_i\}$; else, the bit is 1 and $\max\{x_i, y_i\}$ precedes $\min\{x_i, y_i\}$.

Adjacent keys in the array are grouped together into *chunks*, where each chunk contains $O(k)$ (pairwise permuted) keys encoding a constant number of integers and pointers, each of $b = O(\log n)$ bits. The keys in any chunk belong to a certain interval of values, and the chunks are pairwise disjoint when considered as intervals, thus yielding a total order on any set of the chunks. We introduce some terminology on the chunks to clarify their different use. We have *routing* chunks that help us in routing the search of individual keys, and *filling* chunks that provide a certain flexibility in filling the entries of the array in that we can keep them in no particular order. Access to the latter is via the routing chunks. The number of keys in a chunk is fixed to be either k or $k - \alpha$ for a certain constant $\alpha > 1$, which is clear from the context. We also use a set of *spare* keys that can be individually relocated and referenced for a finer level of flexibility in filling the array, associating $O(1)$ spare keys to some chunks. When considered as intervals, the chunks include the spare keys although the latter physically reside elsewhere in the array.

Our algorithms employ some powerful tools to achieve their worst-case and cache-oblivious bounds. One tool is Willard's algorithm [18] and its use in Dietz-Sleator lists [6]. Suppose we have an array Z of N slots (for a fixed N) storing a *dynamic* set S of $n \leq N$ objects, drawn from a totally ordered universe. At any time, for every pair of object $s_1, s_2 \in S$, if $s_1 < s_2$ then the slot storing s_1 precedes that storing s_2. The data structure proposed by Willard in [18] achieves this goal using a number of $O(\log^2 N)$ arithmetical operations, comparisons and moves, in the worst case, for the insertion or the deletion of an individual object in Z. In our use of Willard's scheme, the routing chunks play the role of the full slots while the filling chunks that of the empty slots. It is possible to insert a new routing chunk (thus replacing a filling chunk that goes elsewhere) and delete a routing chunk (putting in its place a filling chunk taken somewhere). These operations have to maintain the invariant of Willard's scheme according to the total order of the routing chunks stored in the slots. Since the slots are of size $O(k)$ in our case, the bounds of Willard's scheme have to multiplied by a factor of $O(k)$ time or $O(k/B)$ block transfers to insert or delete a routing chunk.

Another useful tool is the van Emde Boas (VEB) layout of Prokop [13]. Given a complete binary search tree T with $n = 2^i - 1$ nodes, the VEB layout of T allows for searching with $O(\log_B n)$ block transfers in a cache-oblivious fashion. Brodal et al. [5] describe how to avoid pointers in the VEB layout, still using extra memory. Franceschini and Grossi [9] show how to make the VEB layout implicit in the form of a VEB permutation of the n keys.

The last tool is for memory management of nodes of variable size with compactor lists [11] and compactor zones [9]. Nodes in the design of implicit data structures are sets of permuted keys that should be maintained as contiguous as possible. For this, nodes of the *same* size are kept together in a segment of contiguous cells (the compactor zone) or in a linked list of fixed size allocation units (the compactor list). Their use make possible to avoid to create empty cells during the operations since the nodes of the same size are collected together. However, when a node changes size, we have to relocate the node from one compactor zone (or list) to another. Since we want to achieve worst-case bounds, we use compactor lists for nodes of size $\Theta(\sqrt{\log n})$ since they are efficient with small size nodes, and compactor zones for nodes of size $\Theta(\log n)$ since they can be incrementally maintained still permitting searching. For larger nodes, we use a different approach described in Section 4.

3 Districts of Chunks

The array of n keys is partitioned into $O(\log \log n)$ portions as in Frederickson [12], where the pth portion stores 2^{2^p} keys, except the last portion, which can store less keys than expected. Inserting or deleting a key in any portion can be reduced to performing the operation (possibly with a different key) in the last portion, while searching is applied to each portion. Achieving a logarithmic cost in each portion sums up to $O(\log_B n)$ block transfers, which is the final cost of the supported operations.

In the rest of the paper we focus on the last portion A of the array, assuming without loss of generality that A is an array of n keys, where $N = 2^{2^p}$ is the maximum size of A for some given integer $p > 0$ and $n \leq N$ is sufficiently large to let us fix $k = \Theta(\log N) = \Theta(\log n)$. (The implicit model assumes that A occupies just $n + O(1)$ cells and that it can be extended to the right one cell at a time up to $n = N$ cells.) This condition is guaranteed using Frederickson's partitioning.

The first $O(\log N)$ keys of A form a preamble encoding some bookkeeping information for A. We partition the rest of A into two parts, the layer $\mathscr{D}$ of the *districts* and the layer $\mathscr{B}$ of the *buckets*. We defer the discussion of layer $\mathscr{B}$ to Section 4. Here, we focus on the districts in layer $\mathscr{D}$ in which we use chunks of size $k - \alpha$ for a certain constant $\alpha > 1$. We partition the initial part of layer $\mathscr{D}$ into a number (at most logarithmic) of consecutive districts $D_0, D_1, \ldots,$ so that each D_i contains 2^i chunks and $\Theta(2^i)$ spare keys according to the invariants we give next. Here, we denote the zone of $\mathscr{D}$ to the right of the districts by F (see Figure 1).

1. The chunks in layer $\mathscr{D}$ are routing chunks and filling chunks, each with $\alpha = \Theta(1)$ spare keys associated. The routing chunks occur only in the districts $D_0, D_1, \ldots,$ while the filling chunks can occur anywhere in $\mathscr{D}$ (i.e., both in $D_0, D_1, \ldots$ and in F).

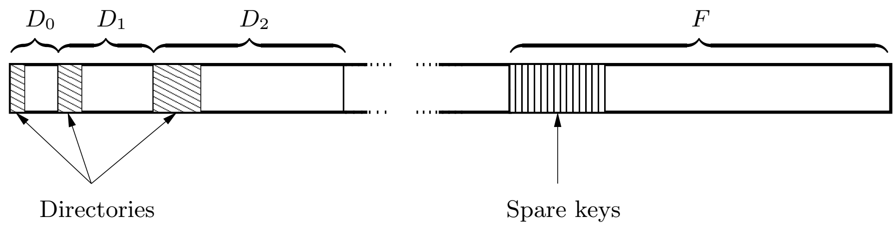

Fig. 1. The districts in layer $\mathscr{D}$.

2. The total left-to-right sequence of routing chunks among all districts in $\mathscr{D}$ is in order, while the filling chunks are not in any particular order. Given any two routing chunks (as closest as possible), the sequence of filling chunks can be arbitrarily long.
3. With each routing chunk c, there are $\Theta(1)$ filling chunks associated. Their number can range between two suitable constants, so that the overall number of filling chunks in F is at least 2^{i+1}. The filling chunks associated with c are the nearest to c in the total order of the chunks, and the pointers to them are encoded in c.
4. The first $\Theta(2^i)$ positions of each district D_i are initially occupied by some spare keys associated with the filling chunks in $\mathscr{D}$. We require that, at any time, the number of these positions is a multiple of the chunk size. The keys in these positions form a *directory* for quickly searching the routing chunks in D_i.
5. The routing chunks in D_i are to the right of their directory, and the first chunk c immediately after the directory is always routing. We maintain the smallest key of c as a spare key that is stored in the preamble of A. In this way, we can discover in which district to search by first reading $O(\log n)$ adjacent spare keys in that preamble.
6. The rest of the spare keys are in F, at the beginning (a multiple of the chunk size) and at the end (any number of them). We incrementally move the spare keys from the end of F to the beginning of F (or vice versa), when adding (or removing) routing chunks in D_i, the rightmost district in $\mathscr{D}$. When the number of routing chunks in $\mathscr{D}$ is sufficiently large, the keys at the beginning of F are already organized to create D_{i+1}, thus shortening F and preparing for D_{i+2} (if any). An analogous situation occurs when D_i has no more routing chunks, and D_{i-1} becomes the rightmost district.

How to search a key in $\mathscr{D}$. The organization mentioned in points 1–6 is not yet suitable for searching. As mentioned in point 5, we can identify efficiently in which district, say D_i, we must go on searching. Once we identify the correct routing chunk in D_i, it is a simple task to examine its associated filling chunks. Consequently, it suffices to show how to search a key in D_i, so that a routing chunk can be identified with $O(\log_B n)$ block transfers.

With this goal in mind, we set up the directory of D_i following the VEB permutation mentioned in Section 2. Since this scenario is well exploited and studied in previous work [9], we do not describe the one-to-one mapping between the VEB permutation in the directory and the 2^i-1 nodes of a complete binary tree. In turn, the nodes of the tree are in one-to-one correspondence with the 2^i-1 chunks in D_i. Although the tree is not yet a search tree, we can activate the search path in it for each routing chunk in D_i. At the beginning, the directory is made up of spare keys from filling chunks and no search path is active.

Given a routing chunk c, let u be the corresponding node in the complete binary tree encoded by the VEB permutation. Since c contains $\Theta(\log n)$ keys and the chunks are disjoint as intervals, we can exchange the smallest keys in c with the spare keys found in the upward path from u. The general property we maintain is that the exchanged keys of c must guide the search towards u when searching keys that fall inside c as an interval. In other words, the paths activated for all the routing chunks form a search tree. The nodes along these paths contain keys taken from the routing chunks, while the rest of the keys in the directory are spare keys from the filling chunks. The routing chunks host temporarily the spare keys that they exchanged in the directory. As a result, the spare keys hosted inside the routing chunk c can be retrieved from the pointers encoded in their filling chunks. Vice versa, the keys in c that are currently in the directory stay along some of the nodes in the upward path from u, and they can be retrieved with a cost of $O(\log_B n)$ block transfers.

With this organization of the directory, searching is a standard task with the VEB permutation as each node have now a routing key when needed. What can be observed is that we actually exchange keys in pairs to encode a flag bit indicating whether u has associated spare keys or keys from a routing chunk. The rest of the searching in the VEB permutation is unchanged.

Lemma 1. *Any key x can be searched in $\mathscr{D}$ with $O(k/B+\log_B n)$ block transfers, identifying the (routing or filling) chunk that contains x.*

How to update $\mathscr{D}$. Within a district D_i, we focus on how to maintain its organization of the keys when the routing chunks are added or removed. The first routing chunk in D_{i+1} is to the immediate right of the directory, in which case we exchange $O(\log n)$ keys with the directory. For the following routing chunks, we apply Willard's algorithm to D_i (without its directory) as described in Section 2:

- The number of routing chunks in each district D_i is dictated by Willard's algorithm. In particular, if D_i is the last district, each of $D_0, D_1, \ldots, D_{i-1}$ has the maximum number of routing chunks according to Willard's algorithm, and the rest are filling chunks. Instead, D_i is not necessarily maximal.
- The structural information needed by Willard's algorithm can be entirely encoded and maintained in layer $\mathscr{D}$.

Willard's scheme preserves the distribution of routing chunks among the filling chunks in $O(\log^2 n)$ steps. In each step, it relocates a routing chunk c from one position to another in D_i by exchanging it with a filling chunk c'. This step requires exchanging the keys of the two chunks incrementally, then performing a search to locate and re-encode the incoming pointer to c'.

However this alone does not guarantee searchability as we need to update the VEB permutation. We therefore divide the step in further $O(\log n)$ substeps that essentially remove c and its search path in the directory and re-insert it into another position, along with its new search path. Specifically, in each substep we retrieve one of the $O(\log n)$ keys of c that are in the directory and put it back in c by exchanging it with the corresponding spare key temporarily hosted in c (note that each spare key requires a search). Next, we exchange c with c', and propagate the same exchange in the VEB permutation of the directory. We then run further $O(\log n)$ substeps to trace the path for the new position of c and exchange its keys so that it is now searchable in the directory. During the substeps, c is the only chunk not searchable in D_i. But we can encode a pointer to it in the preamble, so that searching treats c as a special case. When the task for c is completed, Willard's scheme takes another routing chunk, which becomes the new special case. In summary, each of the $O(\log^2 n)$ steps in Willard's scheme can be divided into further $O(\log n)$ substeps, each costing $O(k + \log n) = O(\log n)$ time and $O(k/B + \log_B n) = O(\log_B n)$ block transfers. It is crucial noting that after each substep, we can run the search as stated in Lemma 1 plus the special case for the current c.

When inserting and deleting routing chunks in a district D_j, for $j < i$, we perform the same steps as in D_i. However we must preserve the property that the number of routing chunks is maximal. This means inserting/deleting a routing chunk also in each of $D_{j+1}, \ldots, D_i$. Since there are $O(\log n)$ districts, we have an extra logarithmic factor in the number of substeps for the districts in the entire layer $\mathscr{D}$.

Theorem 1. *Layer $\mathscr{D}$ can be maintained under insertion and deletion of single routing chunks and filling chunks by performing no more than $O(\mathrm{polylog}(n))$ incremental substeps, each requiring $O(\log n)$ time and $O(\log_B n)$ block transfers. After executing each single substep, searching a key for identifying its chunk takes $O(\log n)$ time and $O(\log_B n)$ block transfers for any (unknown) value of B.*

4 Indirection with Dynamic Buckets

The layer $\mathscr{B}$ of the array A introduced in Section 3 is populated with buckets containing from $\Omega(k^{d-1})$ to $O(k^d)$ keys, for a constant $d \geq 5$. Each bucket is a balanced tree of constant height. A tree is maintained balanced by split and merge operations applied to the nodes. Unlike regular B-trees, the condition that causes a rebalancing for a node is defined with a parameter that depends on the whole size of the subtree rooted in the node (e.g., see the weight-balanced B-trees [2]). We now give a high level description of the buckets assuming that the size of each chunk is k and that we can rely on a suitable memory layout of the nodes. We postpone the discussion of the layout to Section 4.2, which is crucial for both implicitness and cache-obliviousness.

4.1 The structure of the buckets

A bucket has a constant number d of levels. Each bucket is associated with either a routing chunk or a filling chunk of layer $\mathscr{D}$, and all the keys in the bucket are greater than those in that chunk.

Leaves. A leaf of a bucket contains from k to $16k$ keys. Moreover, a leaf has associated a maniple that contains from $\sqrt{k}$ to $5\sqrt{k}$ keys and a number of filling chunks that ranges from r to $4r$ for a suitable constant r. The exact value of r concerns the memorization of the internal nodes of the buckets, as clarified in Section 4.2. The filling chunks of a leaf l are maintained in increasing sorted order in a linked list, say $f_1, \ldots, f_s$. Letting m be the maniple associated with l, we have that (1) f_j is the predecessor of f_{j+1} for $1 \leq j < s$, and (2) for each choice of keys $x \in f_s$, $x' \in l$ and $x'' \in m$, we have $x < x' < x''$.

As we shall see, each leaf l, its maniple m and its filling chunks are maintained in a constant number of zone of contiguous memory. Hence, searching in these objects requires a total of $O(k+\log n)$ time and $O(k/B+\log_B n)$ block transfers.

Internal nodes. An internal node contains routing chunks and filling chunks, and the pointer to the jth child is encoded by $O(\log n)$ keys in the jth chunk, which must be routing. Following an approach similar to that in [2], we define the *weight* $w(v)$ of an internal node v at level i (here, the leaves are at level 1) as the number of keys in the leaves descending from v. We maintain the weight ranging from $4^i k^i$ to $4^{i+1} k^i$. For this reason the number of chunks of an internal node can range from k to $16k$. For the root of a bucket, we only require the upper bound on its weight, since the bucket size can be $\Omega(k^{d-1})$ and the number of chunks in the root can be $O(1)$.

In order to pay $O(\log_B n)$ block transfers when searching and updating an internal node v, we maintain a directory of $\Theta(k)$ keys in v, analogously to what done in [11]. Thus the chunks of v are not maintained in sorted order, but their order can be retrieved by scanning the directory in v. In this way, any operation on v involves only $O(1)$ chunks and portions of $\Theta(k)$ contiguous keys each.

Handling insertions and deletions. If we ignore the memory management, the insertion or the deletion of a key in a bucket is a relatively standard task. If x is the key to insert into chunk c, the root of a bucket, we place x in its position inside c, shifting at most k keys to extract the maximum key in that chunk. We obtain the new key x to insert into the node whose pointer is encoded in c. In general, inserting x into a chunk of an internal node u goes along the same lines.

When we reach a leaf l, we perform a constant number of shifts and extractions of the maximum key in its filling chunks $f_1, \ldots, f_s$ and in l itself. We end up inserting a key into the maniple m of l. If the size of m exceeds the maximum allowed, we extract the $\sqrt{k}$ smallest keys from m and insert them into l. If the size of l is less than $16k$, we are done. On the contrary, if also l exceeds the maximum allowed but the number of its filling chunks is still less than $4r$, we extract the smallest chunk of l and create a new filling chunk f_{s+1}.

Instead, if the number of filling chunks is the maximum allowed, $4r$, we "split" the whole group made up of the leaf l, its maniple z and its filling chunks. That is to say, we partition all the keys so that we have two new groups of the same kind, each group member satisfying all the invariants with their values half on the way between the maximum and the minimum allowed. We also generate a median (routing) chunk that have to be inserted in the parent of l, encoding a pointer in that chunk to the new leaf. We then examine all the ancestors of l, except the root, splitting every ancestor that exceeds its maximum allowed weight, obtaining two nodes of roughly the same weight. Deleting a key is analogous, except that we merge two internal nodes, although we may split once after a merge when the resulting node is too big. For the leaves we need merging and borrowing with an individual key. Merging and splitting the root of a bucket fall inside the control of a mechanism for the synchronization between layer $\mathscr{D}$ and layer $\mathscr{B}$, described in Section 5.

4.2 Memory layout

We now discuss how to store the buckets in a contiguous portion of memory, which is divided into three areas.

- The *filling area* stores all filling chunks of layer $\mathscr{B}$ and the routing chunks of the internal nodes of the buckets.
- The *leaf area* stores all the leaves of the buckets using a new variation of the technique of *compactor zones* [9] that is suitable for de-amortization.
- The *maniple area* stores all the maniples using a set of *compactor lists.*

Filling area. We use the filling chunks to allocate the internal nodes. We need here to make some clear remarks on what we mean by "allocate." Suppose we want to allocate an empty node v with $16k$ chunks. We take a segment of $16k$ filling chunks that are contiguous and devote them to v. Since each filling chunk can be placed everywhere in the memory, when we need to insert a routing chunk c into v, we can replace the leftmost available filling chunk in v with c, moving that filling chunk elsewhere at the cost of searching one of its keys and of re-encoding the pointer to it, with $O(\log n)$ time and $O(k/B)$ block transfers.

Keeping the above remark in mind, we logically divide the filling zone into segments of $16k$ filling chunks each, since we can have a maximum of $16k$ routing chunks for an internal node. A segment is considered "free memory" if it contains only filling chunks. An internal node v with t routing chunks is stored in a segment with the first t routing chunks permuted and the remaining $16k - t$ filling chunks. When a routing chunk needs to be inserted into an internal node v whose weight is not maximal, we put the chunk in place of a filling chunk in the segment assigned to v. The replaced filling chunk will find a new place in

- either the segment of the child u of v, if u is an internal node that splits,
- or between the filling area and the leaf area, if u is a leaf that splits (the filling area increases by one chunk).

The deletion of a routing chunk in v is analogous. We replace the chunk with a filling chunk that arises either from the two merged children of v, if these children are internal nodes, or from the last position of the filling area, if these children are leaves (and the filling area decreases by one chunk). Thus, using the directory for the routing as described above, we are able to insert or delete a chunk in an internal node in $O(\log n)$ time and $O(k/B)$ block transfers.

When we have to split an internal node v in two nodes v', v'', we allocate a new segment a for v'' while re-using the segment of v for v', and exchange incrementally the routing chunks in the segment of v' with filling chunks of a, the segment for v''. We exchange a constant number of chunks at each step, and

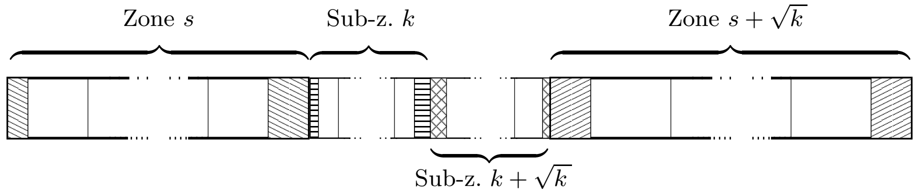

Fig. 2. Compactor zones and sub-zones with broken items highlighted.

these $s = O(k)$ steps are spread through the subsequent s operations operating through v. Note that, during this transition, v is considered not split but only partitioned in two segments instead of one. The execution of a merge is analogous. The invariants defined on the buckets guarantee that we can terminate an incremental transfer before that a further split or merge occurs.

The management of segments is through a simple linked list of free segments. The constant r that bounds the minimum number of filling chunks associated with a leaf can be easily chosen so that we can guarantee that there exists a sufficient number of filling chunks in layer $\mathscr{B}$ for all internal nodes.

Leaf area. The size of the leaves ranges from k to $16k$ keys, and vary by $\sqrt{k}$ keys at a time. Using the technique of the compactor zones, we maintain $15\sqrt{k}+1$ zones of contiguous memory, one for each possible size. Each zone is indexed by the size of the leaves it contains. The zones are in order by this index, so that zone s precedes zone $s+\sqrt{k}$, for each $s = k, k+\sqrt{k}, k+2\sqrt{k}, \ldots, 16k-\sqrt{k}$. When we have to add $\sqrt{k}$ keys to a leaf l of size s, we would like to extract l out of all compactor zones, moving l near to the $\sqrt{k}$ keys to be added by rotating each traversed zone by s keys. As a result, all the leaves are in a contiguous portion of memory except for a single leaf that can be "broken" in two pieces because of the rotation. This scheme is simple and powerful but too costly. We achieve our worst-case bounds with a two-step modification of this scheme. The first step exploits the fact that, for each leaf l,

1. $\Omega(\sqrt{k})$ update operations occur in its maniple between two consecutive variations of $\sqrt{k}$ in the size of l;
2. $\Omega(k)$ update operations occur in its maniple between two consecutive variations of k in the size of l (due to the creation/destruction of a filling chunk);
3. $\Omega(k)$ update operations occur in its filling chunks and its maniple between two consecutive splits or merges of l.

Consequently, we have a sufficient number of operations to perform incrementally the updates involving a leaf l. The basic idea is to execute a constant number of rotations from zone to zone in a single operation.

The second step introduces two *commuting sub-zones* between any two compactor zones. These two sub-zones work like the compactor zones but contain blocks of keys in transit between zones (see Figure 2). For any pair of sub-zones, the first sub-zone contains the blocks of $k+\sqrt{k}$ keys that have to be inserted in or deleted from a leaf. The second sub-zone contains

– chunks that have to be inserted or deleted in a leaf;
– all the chunks of the leaves to be split or merged.

For example, when a leaf reaches its maximum number of keys, it is transformed into a linked list of $O(1)$ chunks going to the second sub-zone near zone $16k$. At this point, we incrementally move these chunks until we reach the sub-zone near zone $8k$; we split the list into two parts and put them as two new leaves of size $8k$. Note that the leaf is still searchable while traversing the zones.

Maniple area The maniple area is handled with compactor lists [11]. However, we use allocation units of size $\sqrt{k}$, and so the structural information for them must be encoded in the leaves associated with the maniples. Each time we need a structural information (e.g., next allocation unit in a list), we perform a search to locate the corresponding leaf. There are $O(\sqrt{k})$ heads of size at most $\sqrt{k}$, so the whole head area occupies $O(k)$ positions and can be scanned each time.

Theorem 2. *Searching, inserting and deleting a key in a bucket of layer $\mathscr{B}$ takes $O(\log n)$ time and $O(\log_B n)$ block transfers for any (unknown) value of B.*

5 Synchronization between Layer $\mathscr{D}$ and Layer $\mathscr{B}$

We combine the two layers described in Sections 3–4 by using a simple variation of the Dietz-Sleator list [6]. Every other $\Omega(\text{polylog}(n))$ operations in layer $\mathscr{B}$, we eventually split the largest bucket and we merge the smallest bucket. This causes the insertion and the deletion of a routing chunk in layer $\mathscr{D}$. By setting up the suitable multiplicative constants, we provide a time slot that is sufficient to complete the algorithms operating in layer $\mathscr{D}$ by Theorem 1.

Theorem 3. *An array of n keys can be maintained under insertions and deletions in $O(\log n)$ worst-case time per operation using just $O(1)$ RAM registers, so that searching a key takes $O(\log n)$ time. The only operations performed on the keys are comparisons and moves. They require $O(\log_B n)$ block transfers in the worst case for the cache-oblivious model, where the block size B is unknown to the operations.*

References

1. Alfred V. Aho, John E. Hopcroft, and J. D. Ullman. *The Design and Analysis of Computer Algorithms.* Addison-Wesley, Reading, 1974.
2. L. Arge and J. S. Vitter. Optimal dynamic interval management in external memory. In IEEE, editor, *37th Annual Symposium on Foundations of Computer Science: October 14–16, 1996, Burlington, Vermont*, pages 560–569, USA, 1996. IEEE Computer Society Press.
3. M. A. Bender, E. D. Demaine, and M. Farach-Colton. Cache-oblivious B-trees. In IEEE, editor, *41st Annual Symposium on Foundations of Computer Science: proceedings: 12–14 November, 2000, Redondo Beach, California*, pages 399–409, USA, 2000. IEEE Computer Society Press.
4. Michael A. Bender, Richard Cole, and Rajeev Raman. Exponential structures for efficient cache-oblivious algorithms. *International Colloquium on Automata, Languages and Programming, LNCS*, 2380:195–206, 2002.

5. 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.
6. P. Dietz and D. Sleator. Two algorithms for maintaining order in a list. In Alfred Aho, editor, *Proceedings of the 19th Annual ACM Symposium on Theory of Computing*, pages 365–372, 1987. ACM Press.
7. 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.
8. Gianni Franceschini and Roberto Grossi. Optimal cache-oblivious implicit dictionaries. *International Colloquium on Automata, Languages and Programming, LNCS*, 2003.
9. Gianni Franceschini and Roberto Grossi. Optimal implicit and cache-oblivious dictionaries over unbounded universes. Full version, 2003.
10. 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.
11. 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.
12. Greg N. Frederickson. Implicit data structures for the dictionary problem. *Journal of the ACM*, 30(1):80–94, 1983.
13. 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.
14. Alon Itai, Alan G. Konheim, and Michael Rodeh. A sparse table implementation of priority queues. In Shimon Even and Oded Kariv, editors, *International Colloquium on Automata, Languages and Programming*, volume 115 of *Lecture Notes in Computer Science*, pages 417–431, 1981.
15. D. E. Knuth. *The Art of Computer Programming III: Sorting and Searching*. Addison–Wesley, Reading, Massachusetts, 1973.
16. 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.
17. 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.
18. Dan E. Willard. A density control algorithm for doing insertions and deletions in a sequentially ordered file in good worst-case time. *Information and Computation*, 97(2):150–204, April 1992.
19. J. W. J. Williams. Algorithm 232: Heapsort. *Communications of the ACM*, 7:347–348, 1964.
20. Andrew C. Yao. Should tables be sorted? *J. Assoc. Comput. Mach.*, 31:245–281, 1984.