Optimal Cache-Oblivious Implicit Dictionaries ⋆

Gianni Franceschini and Roberto Grossi

Dipartimento di Informatica, Università di Pisa
via Filippo Buonarroti 2, 56127 Pisa, Italy
{francesc,grossi}@di.unipi.it

Abstract. We consider the issues of implicitness and cache-obliviousness in the classical dictionary problem for n distinct keys over an unbounded and ordered universe. One finding in this paper is that of closing the longstanding open problem about the existence of an optimal implicit dictionary over an unbounded universe. Another finding is motivated by the antithetic features of implicit and cache-oblivious models in data structures. We show how to blend their best qualities achieving $O(\log n)$ time and $O(\log_B n)$ block transfers for searching and for amortized updating, while using just n memory cells like sorted arrays and heaps. As a result, we avoid space wasting and provide fast data access at any level of the memory hierarchy.

1 Introduction

The dictionary problem is a classical paradigm for studying the limitations and the characteristics of several computational models. In this problem a set of n distinct keys is maintained under insertions and deletions of individual keys while supporting searches. Several models assume that the keys are defined over an unbounded and ordered universe, and the only operations allowed on them are reads/writes and comparisons. Among others, implicit models and cache-oblivious models have recently stimulated a surge of interest.

Our first finding holds for the data structures in the implicit model [15, 16], in which the only space usage is that of the keys with no waste of memory cells. Sorted arrays [14] and heaps [7, 20] are the simplest form. While the term "implicit" originated in [16], it has also been the subject of papers taking a somewhat different point of view, including a long lists of results in perfect hashing [11, 6], bounded-universe and succinct dictionaries [5, 17, 19], and cache-oblivious data structures [4]. These results use a model different from the model adopted in [16] and following papers. The latter model extends the comparison model, so that a suitable permutation of the n keys in a contiguous segment of n memory cells encodes the whole dictionary and *no* other information is explicitly required other than the keys themselves and $O(1)$ temporary RAM registers. The segment can be enlarged or shortened to the right by one cell at a time in constant time. It is a long-standing open problem how to implement an implicit dictionary in $O(\log n)$ time per operation. A number of results (e.g. [8–10, 15, 16]) neared to this objective. We give a positive answer by describing 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.

Our second finding relates to the data structures in the cache-oblivious (memory-hierarchy) model [12], such as cache-oblivious B-trees [1] and other dictionaries (e.g., [3, 4, 2]). 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

⋆ Work partially supported by the Italian MIUR project PRIN "ALINWEB: Algorithmics for Internet and the Web".

operating in the model. Apparently, the two models above are antithetic. On one hand, implicit dictionaries loose data locality when permuting the keys. Due to the dynamic maintainance of the permutation without wasting memory cells, their algorithms give rise to irregular access patterns jumping from one memory cell to another. On the other hand, cache-oblivious dictionaries carefully handle irregular access patterns 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 this paper, we show that our flat implicit trees combine together the appealing features of the two models, avoiding their contrasting drawbacks mentioned above. In fact, our data structure requires $O(\log n)$ time and $O(\log_B n)$ block transfers for searching, and $O(\log n)$ amortized time and $O(\log_B n)$ amortized block transfers for updating, while using just n memory cells like sorted arrays and heaps. Not only we avoid space wasting; we provide optimal data access at any level of the memory hierarchy.

Compared to previous work, our flat implicit tree is complementary to that in [9] achieving $O(\log_B n)$ bounds for $B = \Omega(\log n)$, which is not restrictive in real situations. The bounds in [9] are worst case in the cache-aware model and, moreover, scanning r keys takes $O(\log_B n + r/B)$ block transfers. The pointer-less data structure in [4] is cache-oblivious but, as noted by its authors, it is not properly implicit as it occupies $(1+\epsilon)n$ cells for any $\epsilon > 0$, while permitting an $O(1+r/B)$ scanning cost. 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.

The paper is organized as follows. In Section 2, we give an overview of our data structure. Section 3 describes the bottom layer while Section 4 discusses the top layer of the structure. We put all together in Section 5 for our final bounds.

2 Overview of the Cache-Oblivious Implicit Dictionary

We encode data by a pairwise (odd-even) permutation of keys [15]. A pointer or an integer of b bits is encoded by distinct keys $x_1, y_1, x_2, y_2, \ldots, x_b, y_b$, so that the ith bit is 0 when $\min\{x_i, y_i\}$ precedes $\max\{x_i, y_i\}$ and it is 1 when $\max\{x_i, y_i\}$ precedes $\min\{x_i, y_i\}$. From a high-level point of view, our implicit dictionary is a suitable collection of *chunks* [15, 9, 8] and *spare* keys. Each chunk contains k keys 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. Hence, 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 and the remaining ones are the spare keys, which are kept together in a contiguous segment of memory without any particular organization.

We keep the invariant that the number n of keys satisfies $n'/4 < n < n'$, where n' is a power of two, thus fixing the chunk size $k = \Theta(\log n')$. We can avoid to keep the value of n and n' explicitly, as a variable-length encoding, such as the δ-code, can represent them asymptotically in the 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 in-place global rebuilding our implicit dictionary to guarantee that $n'/4 < n < n'$. After each rebuilding, the n distinct keys are organized in two layers.

The *top* layer is the *super-root* containing $a = \Theta(n/k^2)$ chunks, called *actual* chunks, where a is *always* a power of two. Also, it contains a number of *virtual* chunks stored in no particular order. There are $O(1)$ 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. We describe this layer in Section 4.

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 keys in the descendants 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 contains $\Theta(\sqrt{k})$ bucket chunks and has associated $\Theta(\sqrt{k})$ spare keys and a *maniple* of $\Theta(k)$ keys. The buckets are pairwise disjoint when considered as intervals. More details are in Section 3.

While searching a key in this organization, we identify an actual chunk and, hence, its $O(1)$ associated virtual chunks in the top layer. Among them, we determine the root of the bottom-layer bucket for completing the search. As customary to implicit data structures, we describe the memory layout of our dictionary since this heavily affects the complexity of the operations:

- 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.

We introduce 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.

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 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 of $m + r$ cells. Letting $X = x_1, \ldots, x_m$, and $Y = y_1, \ldots, y_r$, we want to perform an in-place operation that starts from XY and obtains YX in the same segment of memory. Since $YX = \left(X^R Y^R\right)^R$, where R denotes the reversal of the sequences, we can easily reverse the given sequences by swapping their keys in a total of $O(m + r)$ time and $O((m + r)/B)$ 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/borrowing 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. Each intermediate node varies from $k\sqrt{k}$ to $4k\sqrt{k}$ keys (i.e., $\sqrt{k}$ to $4\sqrt{k}$ bucket chunks) and is the only child of its root. The number of

leaves that are children of the intermediate node varies from $\sqrt{k}$ to $4\sqrt{k}$, one leaf per chunk. The pointer to the ith child leaf is encoded by $O(\log n)$ keys in the ith chunk of the intermediate node. Leaves contain from $k\sqrt{k}$ to $4k\sqrt{k}$ keys like intermediate nodes. In addition, the first $\sqrt{k}$ chunks of each leaf have associated from 1 to 5 spare keys each. Given one such chunk seen as an interval, its spare keys belong to that interval. The leaf has also associated a maniple containing from k to $5k$ keys and varying by $\sqrt{k}$ keys at time. The keys in the maniple are larger than those in the leaf.

We keep the above invariants on the number of keys during the updates. When inserting a key, we increment by 1 the number of spare keys. If a chunk in a leaf v has associated 6 spare keys, we redistribute the keys in v. If v has associated a total of $5\sqrt{k}+1$ spare keys, we redistribute the keys between v and its associated maniple z, so that v has $\sqrt{k}$ spare keys less. When v contains $4k\sqrt{k}$ keys and z contains $5k$ keys, we split v and z creating two leaves with their associated spare keys and two maniples. 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 contains $4k\sqrt{k}+k$ keys, we need to split u as well. 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 or borrow instead of splitting u and v; borrowing in v involves an individual key whereas in u it involves an individual chunk. We now detail how to handle the keys.

3.2 Memory layout

We postpone the layout of the roots to Section 4. 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 internal nodes and the leaves in the node area and the maniples in the maniple area. 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 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. The (encoded) pointer to a broken node contains extra bits to encode that value of s_1.

The zones share common techniques; for instance, we employ the 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. 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 $3\sqrt{k}+1$ 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 $O(k\sqrt{k})$ time and $O(k\sqrt{k}/B)$ block transfers, with the initial portion of the broken node in s followed by w and by zone $s+k$. We 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 $s' = s+k, s+2k, \ldots, 4k\sqrt{k}$ (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 $4k\sqrt{k}$, that is, between the node area and the maniple area. The total cost 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.

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.

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'. Note that, since c has size k, the total cost 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. The primitives here supported are:

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. It is analogous to ExtractNode, but it takes $O(k\sqrt{k})$ time and $O(\sqrt{k}\lceil k/B\rceil)$ block transfers, plus the cost of $O(\sqrt{k})$ searches.

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. Its cost is $O(k\sqrt{k})$ time and $O(\sqrt{k}\lceil k/B\rceil)$ block transfers, plus the cost of $O(\sqrt{k})$ searches.

TransferKeys(m) transfers m contiguous keys from the left of the maniple area to its right, or vice versa, where $m \le k$. It is like TransferChunk, except that each zone rotates by m positions. The total cost 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. So, its cost is $O(k)$ time and $O(\lceil k/B\rceil)$ block transfers plus the cost of $O(1)$ searches. We may have 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, 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 Node management

The internal structure of the nodes in the bucket trees allows for quick searching and updating even though their size is $\Theta(k\sqrt{k})$. We give more details on how to achieve this goal.

The simplest organization is that of the internal nodes. Given an internal 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 in $O(k)$ time and $O(\lceil k/B \rceil)$ block transfers. Moreover, given a node u with all of its keys in sorted order, we can build the directory for u by shifting the leftmost and rightmost key in each chunk, in a total of $O(k^2)$ time and $O(\sqrt{k}\lceil k\sqrt{k}/B \rceil)$ block transfers. 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. The cost for routing is $O(k)$ time and $O(\lceil k/B \rceil)$ block transfers.

The leaves undergo a more involved organization because they have associated spare keys. We give a two-step description, in which we first describe how to maintain the invariant on the number of spare keys in $O(\log n)$ time 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 the number of spare keys in a leaf v is non-maximum (i.e., less than $5\sqrt{k}$ keys in the first $\sqrt{k}$ chunks) and that we have to add one more key x to a chunk c in v that has the maximum number of spare keys, i.e., 5. Among the first $\sqrt{k}$ chunks in v, let c' be the nearest chunk, say, at some position to the right of c, so that c' has less than 5 spare keys associated. 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. For any chunk c'' lying between c and c' exclusive, we want to 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 c', we insert into it the key arriving from the chunk to its left and we shift its keys to add one more spare key into c'. Consequently, associating one more spare key with c can be translated into increasing the number of spare keys in c' (if any). While we can shift the keys in c and c', we cannot afford the $O(k)$ cost of shifting all the keys in the intermediate chunks c'' between c and c', as they can be $\Theta(\sqrt{k})$ in number.

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:

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 pointers to the spare keys y for c'', and $a_{\sqrt{k}+1} < y < a_{k-\sqrt{k}}$.

The cost of implementing the above rotation is $O(\log k)$ time since we need to recover the value of the offset r encoded in $O(\log k)$ keys of c''. Immediately before that the condition 3 is violated, we can restore that condition by scanning all the keys in c'' in $O(k)$ time, possibly changing the spare keys and re-encoding the pointers to them. The symmetric operation of deleting x_1 and inserting x_2 is analogous.

Our organization is not yet suitable for cache-obliviousness because encoding and decoding the offsets for rotations of the chunks require an access to the keys in each chunk, which is a problem for small values of the (unknown) block size B. To make it cache-oblivious, we form a directory at the beginning of leaf v by a cumulative shift of $a_1, \ldots, a_{\sqrt{k}}$ and $a_{k-\sqrt{k}+1}, \ldots, a_k$ for each c''. We obtain a larger directory than that of internal nodes, still the number of keys in the directory is $O(k)$. As a result, the rotation of each chunk c'' will always occur inside that directory. We can build from scratch the directory for v in sorted order by applying a cumulative shift to the $\sqrt{k}$ leftmost and the $\sqrt{k}$ rightmost keys in each chunk, in $O(k^2)$ time and $O(\sqrt{k}\lceil k\sqrt{k}/B \rceil)$ block transfers. Note that all rotations are set to zero after the operation. Routing a search key x inside v uses the directory, in $O(k)$ time and $O(\lceil k/B \rceil)$ block transfers. The primitives here supported are:

InsertKey(x, v) with the algorithms just mentioned. Note the number of keys in v does not change. The new spare key of c' is hosted in the spare area described in Section 3.2. The cost is $O(k)$ time and $O(\lceil k/B \rceil)$ block transfers plus $O(1)$ searches, unless condition 3 is violated for a chunk in v. In the latter case, at least $\Omega(\sqrt{k})$ updates occurred in that chunk and the cost of $O(k)$ time and $O(\lceil k/B \rceil)$ block transfers for restoring that condition with a cumulative shift amortizes (divided by $\sqrt{k}$).

ExtractKey(v, x) is executed analogously to InsertKey, so that the cost is the same. After the operation, x is the rightmost key in the spare area.

3.4 Bucket management

We now discuss the operations on the buckets. Searching key x after visiting the root of the bucket in the top layer consists of routing x inside the intermediate node u, which is the only child of the root. If we find the key as a member of a chunk in u, we are done. Otherwise, we identify a chunk c_j in u, reaching the leaf v whose pointer is encoded in c_j. We route x also inside v. Here, either x is a member of a chunk in v or it is a spare key in it; or it is larger than the keys in v and so we scan the keys in its maniple z. Otherwise, we can infer that x is not stored in the bucket. Since routing requires $O(k)$ time and $O(\lceil k/B \rceil)$ block transfers in u and v (see Section 3.3), searching x in a bucket takes $O(k)$ time and $O(\lceil k/B \rceil)$ block transfers.

We now discuss the insertion of x, in which we first search x identifying a leaf v (if x's position is inside a chunk c of the intermediate node u, we extract the rightmost key in c and set x to be this key, which should be inserted to v). If the number of spare keys in v is less than $5\sqrt{k}$, we can insert x into v using InsertKey(x, v). Otherwise, we have to reorganize v, its associated maniple z and its spare keys, so that the number of spare keys in v reduces to less than $5\sqrt{k}$.

Case 1: Maniple z contains less than $5k$ keys. We move the largest $\sqrt{k}$ keys in v to z as follows. We remove the largest $\sqrt{k}$ keys in v using ExtractKey for $\sqrt{k}$ times. Each time we extract one such key, we find it as the rightmost key in the spare area. We then move incrementally these $\sqrt{k}$ extracted keys from the end of the spare area to its beginning, thus shortening the spare area by $\sqrt{k}$ positions to the left. Now, these keys are between the maniple area and the spare area. We run ExtractManiple(z) moving z between the maniple area and the spare area, so that z and the $\sqrt{k}$ keys are contiguous. At this point, we add the latter keys to z and then insert back z into the maniple area by performing InsertManiple(z).

Case 2: Maniple z contains exactly $5k$ keys and leaf v contains less than $4k\sqrt{k}$ keys. We lower the number of spare keys in v by redistributing some keys between v and z. We proceed as in case 1, except that ExtractManiple(z) moves z between the node area and the maniple area. Then, we execute TransferKeys$(\sqrt{k})$ to move the $\sqrt{k}$ keys from their position between the maniple area and the spare area to their new position between the node area and the maniple area, near to z. We remove the first k keys of the just extended z to form a chunk c, with the smallest keys in it. We then insert back z in the maniple area by performing InsertManiple(z). Next, we run ExtractNode(v) bringing v near to c between the node area and the maniple area. We add c to v by a cumulative shift to move c to its correct positions inside v. We then perform a cumulative shift of the leftmost and the rightmost $\sqrt{k}$ keys in c to their position in the directory of v described in Section 3.3. We then insert back v with InsertNode(v).

Case 3: Maniple z contains exactly $5k$ keys and leaf v contains exactly $4k\sqrt{k}$ keys. We create 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. We possibly propagate the split to u, the parent of v. If also u splits, we create two buckets from the current one. We now detail these steps.

We proceed in a way similar to cases 1–2, moving v, z and the spare keys to the position between the node area and the maniple area. We perform an in-place merge of the keys in v and the sorted sequence of the spare keys. As a result, we have a sorted sequence of keys by reading v, the spare keys and z in this order. We divide this sequence in six parts: $v_1, s_1, z_1, c, v_2, s_2, z_2$, where c is the median chunk to be inserted in v's parent u. Leaf v_1 contains $2k\sqrt{k}$ keys and has associated 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 maniple z_2 of $2k+\sqrt{k}$ keys and $2\sqrt{k}+1$ spare keys in s_2. We perform cumulative shifts in these $O(k\sqrt{k})$ keys obtaining $v_1, v_2, c, s_1, s_2, z_1, z_2$ in this order (we fix $m = k$ or $m = \sqrt{k}$ in the cumulative shifts according to the cases). We build the directories of v_1 and v_2 as described in Section 3.3. We 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 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 (it was shortened by the several calls to ExtractKey). Since the keys in s_1 and s_2 are not spare keys, we execute InsertKey(x, v_1) for $x \in s_1$ and InsertKey(x, v_2) for $x \in s_2$. As mentioned in Section 3.2, InsertKey adds a spare key x' at the end of the spare area. We move x' to the memory cell freed by x after its insertion.

At this point, we are left with handling c between the node area and the maniple area. Let u be the intermediate node, parent of v. We move u near to c between the node area and the maniple area by executing ExtractNode(u). If u has less than $4k\sqrt{k}$ keys, we add c to u by a cumulative shift, analogously to what done for v in case 2, noting that its reorganization is much simpler (see Section 3.3). If u has exactly $4k\sqrt{k}$ keys, we add c to u by a cumulative shift and apply an in-place merging of the resulting directory and chunks. Now the keys in the just extended u are sorted, so we split them into u_1, c' and u_2 in this order. We build the internal directories of u_1 and u_2 as described in Section 3.3. Each of u_1 and u_2 contains $2k\sqrt{k}$ keys, while c' is the median chunk. We have thus created two buckets, and c' is the root of the new bucket containing u_2. We insert u_1 and u_2 into the node area using InsertNode. We also move c' using TransferChunk so that it reaches the position between the root area and the node area, becoming part of the root area as actual or virtual chunk as described in Section 4.

The amortized cost for the insertion is informally given by the worst-case cost spread among $\Omega(\sqrt{k})$ updates in case 1, among $\Omega(k)$ updates in case 2, and among $\Omega(k\sqrt{k})$ updates in case 3, respectively. As a result, the amortized cost for inserting a key into a bucket is $O(k)$ time and $O(\lceil k/B \rceil)$ block transfers.

It remains to discuss how to implement the deletion of key x in a bucket. Let u be the intermediate node and v be the leaf reached by the search. If x belongs to a chunk c of u, we delete it, set x to be the leftmost key in v, add x to c, and recursively remove x from v. Hence we solve the problem of deleting x from v. We have four cases; three of them are the exact counterpart of cases 1–3 discussed for the insertion and dealing with merging leaves and nodes. In addition, the fourth case deals with borrowing, provided that borrowing in v involves an individual key whereas in u it involves an individual chunk. The amortized cost for deleting a key from a bucket is $O(k)$ time and $O(\lceil k/B \rceil)$ block transfers.

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. At any time, only $O(1)$ RAM registers are required to operate dynamically.*

4 Top Layer: Cache-Obliviousness

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 a cache-oblivious fashion.

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. The number a of actual chunks changes only when rebuilding (see Section 5) or when performing a full redistribution of actual and virtual chunks. Since the actual chunks are kept sorted, we can route a key to its (actual or virtual) chunk in the top layer in $O(\log n')$ time.

We now 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 interval 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 $O(k)$ time 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.

4.2 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 [18]. 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 $O(2^h h^2) = O(a \log^2 a) = O(n)$ time and block transfers. (We can achieve a bound of $O(a \log a)$ 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

```
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)
```

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

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 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 execute the heapsort of the keys in the rest of A (i.e., $A[2^{h_T} \ldots 2^h - 1]$). We then 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).

The cost of this construction is asymptotically upper bounded by the solution to the recurrence $C(2^h - 1) = C(2^{h_T} - 1) + 2^{h_T} C(2^{h_B} - 1) + O(2^h h)$. For a suitable constant d, that solution is $C(2^h - 1) \leq d2^h h^2$ 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 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 implicit model, as temporary information can be stored in $O(1)$ RAM registers. Using this "implicit" stack in a register and additional $O(1)$ register, we are able to build in-place the VEB-permutation in $O(n)$ time and block transfers.

4.3 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 [18] 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 [4], which is not permitted in the model for implicit data structures. We use procedure FIND(x, h) in Figure 1 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 *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 $\textsc{Find}(x, h)$ by induction on h (see Figure 1). 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 $\textsc{Find}(x, h_T)$ to route x in S_T. By induction, *rank* = *bfs* mod 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 $i + (2^{h_B} - 1) \cdot \mathit{rank} + 1$ of A. So we update i to this new value. We can invoke $\textsc{Find}(x, h_B)$ 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 $(2^{h_T} - 1 - \mathit{rankroot})$ 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.

The cost of searching is asymptotically upper bounded by the solution to the recurrence $C(2^h - 1) = C(2^{h_T} - 1) + C(2^{h_B} - 1) + O(1)$. 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 $O(h)$ time and $O(h/\log B)$ block transfers.

4.4 Maintaining the top layer and the VEB-permutation

The association of virtual chunks with actual chunks is dynamically maintained when bucket splitting creates new chunks and bucket merging removes chunks. As long as we can keep at most α virtual chunks associated with each actual chunk, we do not need to redistribute chunks. However, when removing an actual chunk that has no virtual chunk associated with it, or when creating a new chunk from a chunk in a maximal list of size α, we have to redistribute the chunks by reclassifying them as actual and virtual, while preserving their order. We employ a notion of density [1, 4, 13] for this purpose, with the further requirements of avoiding to produce empty slots and of distributing virtual chunks among the actual chunks without violating their order. Note that 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. We refer the reader to the full version for the several details on the redistribution.

5 In-Place Rebuilding and Final Bounds

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 case, we have $n = n'/2$ for the new value of n' after rebuilding, and so our invariant is maintained with n half on the way between $n'/4$ and n'. The amortized cost of rebuilding is given by the total cost of $O(n')$ insertions divided by the number of insertions and deletions performed in an epoch, which is $\Omega(n')$. As a result, the amortized cost of the rebuilding is the cost of $O(1)$ insertions (here it should be clear why we do not start a nested sequence of rebuilding operations, since we keep $k = \Theta(\log n')$ unchanged for all the rebuilding). Apart from the rebuilding, the cost of search, insert and delete is that stated in Sections 3 and 4. From these costs, it follows our main result, Theorem 1.

References

1. M. A. Bender, E. D. Demaine, and M. Farach-Colton. Cache-oblivious b-trees. In *Proc. 41st Symposium on Foundations of Computer Science*, pages 399–409, 2000.
2. Michael A. Bender, Richard Cole, and Rajeev Raman. Exponential structures for efficient cache-oblivious algorithms. *Lecture Notes in Computer Science*, 2380:195–207, 2002.
3. Michael A. Bender, Ziyang Duan, John Iacono, and Jing Wu. A locality-preserving cache-oblivious dynamic dictionary. In *Proc. 13th Annual Symposium On Discrete Mathematics*, pages 29–38, 2002.
4. 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.
5. Andrej Brodnik and J. Ian Munro. Membership in constant time and almost-minimum space. *SIAM Journal on Computing*, 28(5):1627–1640, 1999.
6. Amos Fiat, Moni Naor, Jeanette P. Schmidt, and Alan Siegel. Nonoblivious hashing. *Journal of the ACM*, 39(4):764–782, October 1992.
7. Robert W. Floyd. Algorithm 245 (TREESORT). *Communications of the ACM*, 7:701, 1964.
8. Gianni Franceschini and Roberto Grossi. Implicit dictionaries supporting searches and amortized updates in $O(\log n \log\log n)$. In *Proc. 14th Annual Symposium on Discrete Algorithms*, 2003.
9. 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.
10. Greg N. Frederickson. Implicit data structures for the dictionary problem. *Journal of the ACM*, 30(1):80–94, 1983.
11. 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.
12. M. Frigo, C. E. Leiserson, H. Prokop, and S. Ramachandran. Cache-oblivious algorithms. In *Proc. 40th Annual Symposium on Foundations of Computer Science*, pages 285–297, 1999.
13. Alon Itai, Alan G. Konheim, and Michael Rodeh. A sparse table implementation of priority queues. *Proc. Intern. Colloquium on Automata, Languages and Programming*, LNCS 115, pages 417–431, 1981.
14. D. E. Knuth. *The Art of Computer Programming III: Sorting and Searching.* Addison–Wesley, Reading, Massachusetts, 1973.
15. 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.
16. 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.
17. Rasmus Pagh. Low redundancy in static dictionaries with constant query time. *SIAM Journal on Computing*, 31(2):353–363, 2002.
18. H. Prokop. Cache-oblivious algorithms. Master's thesis. MIT, Cambridge, MA, 1999.
19. 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.
20. J. W. J. Williams. Algorithm 232: Heapsort. *Communications of the ACM*, 7:347–348, 1964.