Implicit B-trees: A New Data Structure for the Dictionary Problem*

Gianni Franceschini†

Dipartimento di Informatica
Università di Pisa
francesc@di.unipi.it

Roberto Grossi†

Dipartimento di Informatica
Università di Pisa
grossi@di.unipi.it

J. Ian Munro‡

Department of Computer Science
University of Waterloo
imunro@uwaterloo.ca

Linda Pagli†

Dipartimento di Informatica
Università di Pisa
pagli@di.unipi.it

January 27, 2004

Abstract

An implicit data structure for the dictionary problem maintains n data values in the first n locations of an array in such a way that it efficiently supports the operations insert, delete and search. No information other than that in $O(1)$ memory cells and in the input data is to be retained; and the only operations performed on the data values (other than reads and writes) are comparisons. This paper describes the *Implicit B-tree*, a new data structure supporting these operations in $O(\log_B n)$ block transfers like in regular B-trees, under the realistic assumption that a block stores $B = \Omega(\log n)$ keys, so that reporting r consecutive keys in sorted order has a cost of $O(\log_B n + r/B)$ block transfers. En route a number of space efficient techniques for handling segments of a large array in a memory hierarchy are developed. Being implicit, the proposed data structure occupies exactly $\lceil n/B \rceil$ blocks of memory after each update, where n is the number of keys after each update and B is the number of keys contained in a memory block. In main memory, the time complexity of the operations is $O(\log^2 n / \log\log n)$, disproving a conjecture of the mid 1980's.

*A preliminary version of the results in this paper has been presented at *the 43rd Annual IEEE Symposium on Foundations of Computer Science* [12].

†Dipartimento di Informatica, Università di Pisa, via Buonarroti 2, 56127 Pisa, Italy. Supported was provided in part by the Italian MIUR project "ALINWEB: Algorithmics for Internet and the Web".

‡Author supported by Natural Science and Engineering Research Council of Canada and the Canada Research Chairs program.

1 Introduction

Data storage capacity is increasing rapidly, and doubles every 18 months according to Moore's law. Processor speed is increasing even faster, so one can deploy extra computing power to squeeze more data on storage devices [28]. Minimizing storage costs for large processes is as important as ever, indeed an absolutely crucial issue is to minimize memory use in a manner such that retrievals from the "next level" in the memory hierarchy are minimized.

In this paper, we consider a fundamental problem of data organization in a fundamental way. In particular we re-examine the problem of designing data structures for dynamic search that achieve the best space saving possible, that is, they occupy $O(1)$ memory cells other than those strictly needed for the distinct keys and the problem size. We want to insert, delete and search keys in a dictionary, with all the auxiliary information (including small integers and internal pointers) being implicitly encoded by a suitable permutation of the keys. In other words, if we take a snapshot of the memory, we just see the n distinct keys permuted in n memory cells, assuming that each key occupies a single cell for the sake of presentation. After each update, a new permutation is obtained to fully encode the resulting implicit data structure at no extra memory cost. This class of data structures are known in the literature as *implicit* [20], and Williams' heap [27] is undoubtedly the best known example. There are many advantages to this organization of the keys:

- Compressing the sequence of permuted keys automatically compresses also their associated data structure. Compression of database records is an important consideration [6].
- Transmitting over a channel or dumping for backup the sequence of permuted keys provides indirectly the service also for their associated data structure at no extra cost. This property can be useful to download indexed data in mobile and personal computing.
- There is no replication in the data structure, with low storage costs and no need to maintain consistency among multiple copies of the keys. Scanning the keys is very efficient.

Implicit data structures have been studied to investigate the power of pointers. While the term originated in studying precisely the problem we now investigate [20], it has also been the subject of papers taking a somewhat different point of view, including the seminal paper of Yao [29], a long list of results on the related topics of perfect hashing [15] and the dictionary problem over a bounded universe [5, 21, 23].

We are given a set of n keys from an ordered, but unbounded universe. These keys are considered to be atomic, and we may only perform comparisons and read and write values. Under such a model, it is immediately obvious that $\log_2 n$ comparisons are necessary and sufficient to perform a search. In that sense the sorted array is the most basic "nontrivial implicit data structure". It handles searches optimally and uses no auxiliary storage for pointers etc., but it is static. The heap of Williams [27] and Floyd [8] is the archetypical implicit data structure. It supports a priority queue (operations insert and extract maximum) in the first n locations of an array in $O(\log n)$ time, matching the information theoretic lower bound. Furthermore, no other "structural information" is required, other than the ability of the array to be perceived as growing or shrinking with n. The first work on an implicit data structure for the dictionary problem, and as noted the term, dates from the late 1970's ([20], publication is far from instantaneous) with an $O(n^{1/3} \log n)$ bound. Frederickson [13] generalized an aspect of this approach to achieve a $o(n^{\epsilon})$ bound. Munro [19] improved this to $O(\log^2 n)$ with a method based on pointer encoding and AVL trees [1]. Indeed the paper speculates that $\Theta(\log^2 n)$ may be optimal.

Borodin et al. [4] and Radhakrishnan and Raman [22] gave an interesting tradeoff between data moves in performing an update and the number of comparisons necessary for a search. The bound does not, however, rule out $O(\log n)$ behavior for the problem. Closely related to our problem is that of arranging n k-key records into an n by k array so that searches can be performed quickly given any key value. The static two key problem bears some superficial similarity to the dynamic single key dictionary problem. Nevertheless, searches under this model can be supported in $O(\log n)$ time [7], where the "O" does include a factor depending on k.

In this paper, we describe the *implicit B-tree* in the context of a memory hierarchy. Our main result is showing that the worst-case cost of searching and updating is $O(\lceil(\log n)/B\rceil \log_B n)$ block transfers of size B. Reporting r consecutive keys in sorted order is output sensitive, namely, it takes $O(r/B)$ block transfers plus the search cost, which is $O(\lceil(\log n)/B\rceil \log_B n)$ in our case. These polylogarithmic costs reduce to $O(\log_B n)$ like in regular B-trees, under the realistic assumption that a block stores $B = \Omega(\log n)$ keys. Being implicit, our data structure occupies exactly $\lceil n/B\rceil$ blocks of memory after each update.

While we do not claim the method, as stated, is easily implementable, preliminary experimental results show that the data structure could be useful in practice [2]. Compared to the implementation of regular B-trees, such as those in Sedgewick's book [24] and in Berkley DB by Sleepycat [25], the constant hidden in the big-Oh notation of the time bounds for implicit B-trees is certainly larger, as expected. However, these implementations occupy six times more space than that of implicit B-trees, and this can give a payoff in performance in several cases. For example, `mmap` is a commonly used low-level routine of the file system to access memory mapped files faster than standard I/O routines, and it has a limitation of 4Gb or similar, in common operating systems (e.g.,[18, 26]). It comes out that implicit B-trees can handle up to 4Gb of input data alone with `mmap` while the other data structures can handle roughly up to 4Gb of *both* input data and auxiliary data, which translates in nearly up to 700Mb of input data alone due to their waste of space. As a result, for input data size ranging from 700Mb to 4Gb, they can be less efficient than implicit B-trees, even though their hidden constant in the time complexity is smaller.

An interesting byproduct of implicit B-trees is that they disprove a long standing conjecture in implicit data structures for the dictionary problem [19], and improve our basic understanding of a theory of data ordering. It was conjectured that $\Omega(\log^2 n)$ time is needed for supporting the operations in implicit dictionaries in main memory. Fixing $B = \Theta(\log n)$, we can use implicit B-trees in main memory for attaining $O(\log^2 n/ \log\log n)$ time. This has motivated subsequent work on the topic, leading to two further data structures that we mention for the sake of completeness [9, 10, 11].

The first data structure is the flat implicit tree [9, 10] achieving optimal $O(\log n)$ time for searching and $O(\log n)$ amortized time for updating. It exploits in-place algorithms for amortizing the update bounds and introduces different kinds of chunks of $O(\log n)$ contiguous keys to delay the expensive reorganizations. Differently from implicit B-trees, it uses a two-layer tree of constant height (except very few cases), adapting the redistribution technique of [3, 17] to the implicit model. This organization yields significantly worse bounds in memory hierarchy than implicit B-trees.

The second data structure [10, 11] is cache-oblivious and obtains the bounds of $O(\log_B n)$ block transfers, 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 [16]. The top layer uses a van Emde Boas permutation [16] of the keys as a directory, and the bottom layer introduces techniques of memory layout for attaining cache obliviousness. Compared to implicit B-trees, the cost of reporting r consecutive keys in sorted order is not output sensitive in that cache-oblivious data structure as it may take $\Omega(r)$ block transfers. In general, it is an open problem to devise a cache-oblivious B-tree with output-sensitive cost [3] while allowing (amortized) bounds of $O(\log_B n)$ for any B, even with explicit data structures. The implicit B-tree is able to attain output-sensitive and worst-case bounds for $B = \Omega(\log n)$ as it is aware of the block size B.

We now recall the main features of the implicit model for dictionaries, in which no other "structural information" is explicitly stored other than the keys themselves that are suitably permuted for encoding indirectly the needed structural information. An implicit data structure for the dictionary problem is 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 just $O(1)$ 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 is implicitly encoded by the permutation of $a_1, a_2, \ldots, a_n$.

We adopt the basic technique of [19] to encode the structural information by implicitly representing a pointer or an integer of $b = \log N$ bits by using $2b$ distinct keys, say, $y_1, z_1, y_2, z_2, \ldots, y_b, z_b$ (chosen from $a_1, a_2, \ldots, a_n$) that are *pairwise permuted* with the following rule: if the ith bit is 0, then $\min\{y_i, z_i\}$ precedes $\max\{y_i, z_i\}$; else, the bit is 1 and $\max\{y_i, z_i\}$ precedes $\min\{y_i, z_i\}$. The immediate drawback of this technique is that the mere access of, say, a pointer takes b comparisons and an update of its value requires b comparisons and $O(b)$ exchanges of input elements. In our algorithms for implicit B-trees, our search performs only read accesses to the keys. When inserting or deleting, we need to move keys around to re-encode the pointers or the integers affected by the updating. In general, 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 produces an array of $n-1$ cells containing a new permutation of $a_1, a_2, \ldots, a_{i-1}, a_{i+1}, \ldots, a_n$.

In the following, we assume without loss of generality that n, the number of keys, is bounded above by N, such that $\log N = \Theta(\log n)$. This assumption is not a limitation, as shown by Frederickson [14], but it is useful to fix the bit-length of the pointers. In fact, we can keep $O(\log \log n)$ distinct data structures of size 2^{2^i}, $c \leq i \leq \lceil \log \log n \rceil$ (where c is a constant), so that the total cost is dominated by the cost of the biggest of such data structures. So, our results apply regardless of changes in n.

The paper is organized as follows. In Section 2, we give a high level definition of our implicit B-tree and the supported dictionary operations. In Section 3, we embed the structure in a collection of lists, in order to limit the waste of space due to the variable size of the tree nodes. We show how to completely remove any additional space besides that of the keys in Section 4, where the low level memory organization is given in some detail. Finally, in Section 5, we discuss the refinements to obtain our final bounds.

2 High-Level Structure of the Implicit B-tree

The implicit B-tree has a high-level structure similar to that of regular B-trees; however, the internal organization of the nodes differs and indeed the node size grows with N. This organization depends on two parameters, $k = \Theta(\log N) = O(\log n)$ and $t = \Theta(B^\epsilon)$ (for a positive real constant $\epsilon < 1/2$). The nodes contains $\Theta(tk)$ keys and have $\Theta(t)$ children (as we need k keys to encode each pointer), thus resulting in a height of $O(\log_t n) = O(\log_B n)$. Hence, we need to read $\Theta(\log n)$ keys (with $\Theta(\log n/B)$ block transfers) per node to traverse the tree from the root to a leaf, for a total of $O(\lceil \frac{\log n}{B} \rceil \log_B n)$ block transfers.

Specifically, each leaf contains between tk and $2tk + k - 1$ keys grouped in *chunks* of k keys each, except the last chunk in the leaf, which can have fewer than k. The number of keys in a leaf can increase or decrease by 1 after an update.

Each internal node contains between tk and $2tk$ keys also grouped in chunks of k keys each, and the number of keys increases or decreases by k. As usual, the root R can be smaller, namely, it stores between k and $2tk$ keys in grouped chunks. Other internal nodes have between $t+1$ and $2t+1$ children, and we use a chunk of k keys to separate their pointers instead of single keys. More specifically, if a node has $i+1$ children $c_1, c_2, \ldots, c_{i+1}$, then it stores i chunks $b_1, \ldots, b_i$ of k keys each. The keys in b_j are all greater than those in $b_{j'}$ with $1 \leq j' < j$, and all less than those in

$b_{j''}$ with $j < j'' \leq i$. The keys in b_j are also greater than those in the subtree rooted at c_j and less than those in the subtree rooted at c_{j+1}. The pointers to $c_1, \dots, c_i$ and any other auxiliary information of the nodes are implicitly encoded by a pairwise permutation of the keys as mentioned in the introduction and detailed in Section 3. At this stage of description, we assume that we have a primitive to encode and decode a pointer or a small integer of $O(k)$ bits at the cost of scanning $O(k)$ keys.

If we take a snapshot of the node, we can observe that the keys are not sequentially stored in the node in the order induced by the chunks $b_1, \dots, b_i$. Inside each chunk b_j we permute pairwise the keys to encode the pointers for children c_j and c_{j+1} and some other information introduced later on. Furthermore, we also permute chunks; as we shall see, this is crucial to handle efficiently the node with our algorithms, especially when splitting or merging. More precisely, the t smallest chunks $b_1, \dots, b_t$ are permuted in the first tk positions of the node, and the other $r = i - t$ are permuted in the remaining positions of the node. Unlike the permutation of the keys, the permutation of the chunks does not encode any bits of information but it is the mere byproduct of splitting and merging without relocation.

It is worth noting that chunks in all the nodes represent a partition of the keys into intervals. At any time, the keys in a chunk are either all greater than or all less than those in any other chunk. In other words, to merge any two consecutive chunks, it suffices to concatenate them.

Directories for permuted internal nodes. In order to efficiently route the keys during a traversal of the B-tree, we maintain at most two directories per node. The first directory occupies the first $2t$ positions of the node. Let $b_{\sigma(1)}, \dots, b_{\sigma(t)}$ denote the chunks $b_1, \dots, b_t$ shuffled according to the permutation σ, so that $\sigma(j) = \ell$ indicates that chunk b_ℓ has position j in the permutation σ. The directory for these chunks is made up of the smallest and the largest key of $b_{\sigma(1)}, \dots, b_{\sigma(t)}$, in this order. As a result, each chunk $b_{\sigma(j)}$ has two keys moved to the directory. The directory for the permutation of the trailing r chunks is analogous and occupies $2r$ positions starting at $tk + 1$. At any time, the permutation of the pairs of keys in the directories reflects that of the chunks in the nodes. The first part of the node contains the first t chunks and their directory, and the second part contains the trailing r chunks and their directory. This organization will be helpful when a node is split.

Routing a search for key x in a node (other than the root) requires a sequential scan of the $O(t)$ keys in the directories to identify a pair of keys x_L and x_R such that $x_L \leq x \leq x_R$ and no other key x' in the directories satisfies either $x_L < x' \leq x$ or $x \leq x' < x_R$. We have two cases:

- x_L and x_R belong to the same chunk $b_{\sigma(j)}$ (i.e., x_L occupies an odd numbered position, $2\sigma(j) - 1$, in the directory). We route the key x inside the node, indeed inside $b_{\sigma(j)}$.

- x_L and x_R belong to different chunks $b_{\sigma(j_L)}$ and $b_{\sigma(j_R)}$, respectively (i.e., x_L occupies an even numbered position $2\sigma(j_L)$ in the directory, but x_R an odd numbered position $2\sigma(j_R) - 1$). These chunks would be consecutive if not permuted according to σ, and can be located in constant time by the positions of keys x_L and x_R in the directories. Note that x_L and x_R are not necessarily consecutive in the directories, but $\sigma(j_L) + 1 = \sigma(j_R)$. So they both encode the pointer to child $c_{\sigma(j_R)}$, and routing to x proceeds recursively to that child.

Our encoding scheme for the auxiliary information in the nodes is based on the pairwise permutation, and includes also the keys in the directories. This is not much of a problem, as the pairwise permutation of the keys for encoding bits and the permutation σ in a directory do not interfere each other. To see why, it suffices to consider first the smallest and then the largest in each pair of keys in a directory. Hence, we can handle the two cases above with minor modifications.

Lemma 1 *Following the search path for a key in a permuted internal node can be performed with* $O(t+k) = O(B^\epsilon + \log n)$ *comparisons and* $O((t+k)/B) = O(\frac{\log n}{B})$ *block transfers.*

Searching for a key. The search operation for a key x is rather standard. We start out from the root R and, if $x \notin R$, we route x in each traversed node until we find x inside a chunk or we reach the leaf, whose keys are scanned to find the key.

Lemma 2 *Searching for a key in the implicit B-tree takes* $O((t+k)\log n/\log t) = O((B^\epsilon + \log n)\log_B n)$ *comparisons and* $O\left(\left\lceil \frac{\log n}{B} \right\rceil \log_B n\right)$ *block transfers.*

Proof: The number of visited nodes is $O(\log n/\log t) = O(\log_B n)$ and, for each node, we pay $O(t+k)$ comparisons and a cost of $O((t+k)/B) = O(\lceil \log n/B \rceil)$ block transfers to route the key by Lemma 1, and of $O(k/B) = O(\lceil \log n/B \rceil)$ block transfers to decode the pointer to one of its children, as $t = \Theta(B^\epsilon)$ and $k = O(\log n)$. □

Inserting a key. Let us now examine the insert operation for a new key x. We run the search for x as described above. Once we find the position of x inside chunk b_j of a node, say u, we make room for x in b_j according to two cases:

- If u is a leaf, we shift by one position to the right the keys in b_j following x, as well as the keys in the following chunks $b_{j+1}, b_{j+2}, \ldots$, so that we have one position free in u to store x. The size of the leaf u increases by 1.
- If u is an internal node, we shift by one position to the right the keys in b_j following x. The rightmost shifted key replaces the rightmost key of b_j in the directory. The latter key is then inserted in the leftmost leaf of the subtree rooted at c_{j+1}, since the chunks of the B-tree are disjoint intervals as previously noted. The leftmost leaf is reached by a downward traversal of the subtree.

In both cases, we end up in a leaf whose number of keys increases by 1. No internal node has yet changed size. However, after the insertion, if the leaf involved becomes of size $2tk + k$, we have to split into two leaves storing t chunks each, respectively, $b_1, \ldots, b_t$ and $b_{t+2}, \ldots, b_{2t+1}$. Then, we have to insert the median chunk b_{t+1} into the leaf's parent (which we know because it is on the root-to-leaf path of the insertion), updating two pointers in the parent to the two leaves just created.

At this point we may trigger a sequence of split operations on the internal nodes along the root-to-leaf path traversed during the insertion. Here is how we handle it. Let us consider the general case of an internal node u of maximal size $2tk$, in which we have to insert the median chunk b originating from the split of its child, say c_j, into two nodes c'_j and c''_j. Assume without loss of generality that $j \leq t$, so that we have to update the first half of node u and its directory. We first identify the position i of the chunk b_t in the permutation (with $\sigma(i) = t$) containing the largest pair of keys in the directory of u. We move b_t in a temporary area and store b in its place, putting the smallest and the largest key of b in the two directory positions left free by the homologous keys in b_t. We then access the $O(k)$ keys in u encoding the two child pointers associated with b_t, and permute these keys pairwise to encode the pointers to c'_j and c''_j associated with b. We then locate the (at most two) neighbor chunks of b in u by using the directory, and re-encode one of their pointers to one of the new children c'_j and c''_j, respectively. We finally record in the auxiliary information for the left half u' of u the fact that it is now an independent node with tk keys, and do the same in the right half u'' of u. Note that we do *not* need to move their keys further. As a result, we now have to recursively insert b_t in u's parent, and replace the parent's pointer to u with the pointers to u' and u''. Hence the splitting process in the internal nodes caused by an insert operation does not involve all the keys of each such node, but only those contained in the directories and in $O(1)$ chunks.

Lemma 3 *Splitting a permuted internal node (of maximal size $2tk = O(B^\epsilon \log n)$) can be performed with $O((t+k)/B) = O(\frac{\log n}{B})$ block transfers by relocating only $O(\log n)$ keys in $O(1)$ chunks.*

Proof: Accessing the directory of the node requires $O(t/B)$ block transfers. Replacing the keys in the selected chunk and re-encoding $O(1)$ pointers requires to access $O(k)$ keys with $O(t/B)$ block transfers. In summary, $O(1)$ chunks are involved and only their $O(k)$ keys are relocated. □

In the above process, when we reach an internal node with fewer than $2t$ chunks or when we create a new root, we perform an update by reading all the $O(tk)$ keys in it. We must also relocate the node because its number of keys increases by k. This operation requires $O(tk/B) = O(\log n/B^{1-\epsilon}) = O(\log_B n)$ block transfers and a (costly) node relocation. Fortunately, our B-tree organization guarantees that such an expensive operation is needed at most twice, namely, in the leaf originating the split and in the node in which the splitting process terminates.

Lemma 4 *Inserting a key in the implicit B-tree takes $O\left(\left\lceil \frac{\log n}{B} \right\rceil \log_B n\right)$ block transfers plus the cost of resizing $O(1)$ nodes.*

Proof: Finding the position of the key in a suitable chunk requires $O\left(\left\lceil \frac{\log n}{B} \right\rceil \log_B n\right)$ block transfers by Lemma 2. After that, a suitable leaf must host one more key, and this costs $O(tk/B) = O(\log n/B^{1-\epsilon}) = O(\log_B n)$ block transfers plus the resizing of the leaf. In the worst case, the leaf and its $O(\log_B n)$ ancestor nodes have to be split, each split operation costing $O((t+k)/B) = O\left(\left\lceil \frac{\log n}{B} \right\rceil\right)$ by Lemma 3. Finally, an internal node has to be resized like the leaf. The total cost is $O\left(\left\lceil \frac{\log n}{B} \right\rceil \log_B n\right)$ plus the resizing of at most two nodes. □

Deleting a key. As for the delete operation with a key x, we identify the chunk b_j of a node u, such that x is in b_j, by running the search previously described. We then take the smallest key in the successor chunk of b_j in the leftmost leaf descending from the child c_{j+1} of u, since the chunks of the B-tree are disjoint intervals as previously noted. That key replaces the rightmost key of b_j in the directory of u, since the latter key reuses the space left free by x in b_j (whose auxiliary information has to be re-encoded). We deal with $O(t)$ keys in the directory, $O(k)$ keys in the chunk b_j of u, and $O(tk)$ keys in the leftmost leaf mentioned above. Here, we have a similar situation to that of insertion. We end up deleting a key in a leaf, which can initiate a merge operation or the sharing of a chunk of a node with one of its siblings. Merging can be treated similarly to splitting, and sharing is essentially reducible to the problem of increasing the size of a node by k and of decreasing the size of one its adjacent siblings by k. As for merging, we can merge two nodes of size tk by recording both the fact they are now a single node and a reference in one to the other. We can borrow a chunk from a node's parent in a manner analogous to splitting.

Lemma 5 *Merging two permuted internal nodes (of minimal size tk) can be performed with $O((t+k)/B) = O(\frac{\log n}{B})$ block transfers by relocating only $O(\log n)$ keys in $O(1)$ chunks.*

Proof: Analogous to that of Lemma 3. □

Again, the expensive operations are at the leaf originating the merge and at the node in which the merging process stops (because it contains more than t chunks or it can borrow a chunk from one of its adjacent siblings with the sharing operation). This takes $O(tk/B) = O(\log_B n)$ block transfers and at most three node relocations because of the change of their size.

Lemma 6 *Deleting a key in the implicit B-tree takes* $O\left(\left\lceil \frac{\log n}{B} \right\rceil \log_B n\right)$ *block transfers plus the cost of resizing* $O(1)$ *nodes.*

Proof: Finding the position of the key in a suitable chunk requires $O\left(\left\lceil \frac{\log n}{B} \right\rceil \log_B n\right)$ block transfers by Lemma 2. After that, a suitable leaf has to host one less key, with $O(tk/B) = O(\log n/B^{1-\epsilon}) = O(\log_B n)$ block transfers and the resizing of the leaf. In the worst case, the leaf and its $O(\log_B n)$ ancestor nodes have to be merged, each merge requiring $O((t+k)/B) = O\left(\left\lceil \frac{\log n}{B} \right\rceil\right)$ by Lemma 3. Finally, an internal node has to be resized because it does not propagate the merge or it applies sharing to borrow a chunk from one of its siblings. The total cost is $O\left(\left\lceil \frac{\log n}{B} \right\rceil \log_B n\right)$ plus the resizing of $O(1)$ nodes. □

At this level of detail, we want to maintain the implicit B-tree as a permutation of its stored keys occupying a fixed and compact portion of n memory cells (each capable of storing exactly one key). So an update can be seen as a transition from a permutation of n keys to a permutation of $n+1$ or $n-1$ keys. The main computational difficulty in maintaining this interesting property is that nodes can change their sizes after an update (searching is not a problem in this sense). To overcome this problem, we must recompact some nodes by shifting their keys and possibly relocating them by exchanging positions in the permutation. Our organization guarantees that the expensive relocation mentioned above is necessary for $O(1)$ nodes (a leaf and possibly one ancestor node and its sibling). The rest of the nodes in a root-to-leaf path are involved in split or merge operations that are computationally cheaper, and do not need relocation as they locally permute only $O(k)$ keys (see Lemmas 3 and 5). The importance of this fact is clear; in the worst case, an update operation requires a constant number of transformations of the first type, while the number of splits or merges depends on the height $O(\log_B n)$ of the tree.

In the rest of the paper, we proceed to accommodate the keys of the nodes in memory, according to a two-level scheme. In the first level (Section 3), we pack together the nodes of identical size, embedding them in *compactor* lists to eliminate most of the memory waste. In this context, we preserve the invariant that lists are made up of fixed-size *allocation units*, and each node is stored in $O(1)$ allocation units and must encode also the auxiliary information for the lists. We use an abstract view of the memory manager, which we describe later on in the second level (Section 4). Here, we assign contiguous memory to the allocation units of the lists and dynamically maintain a further packing of the initial unit (*head*) of each list, as the heads are the only allocation units partially filled.

3 Embedding the B-tree Nodes into the Compactor Lists

In this section, we show how to pack the variable-size nodes into a suitable collection of compactor lists of fixed-size allocation units, which will be stored compactly as discussed in Section 4 to relocate memory areas easily and without memory waste.

Before describing the compactor lists in detail, we must fix the information implicitly encoded in the nodes. That is, given a B-tree node u storing ik keys, where $t \leq i \leq 2t$, we can read the information stored in u in two ways:

- *Explicit:* u is a sequence of ik keys, in which the first $2t$ keys form the directory, followed by what remains of t chunks $b_1, \ldots, b_t$. Then, if $i > t$, we have the second directory with $2r$ keys followed by what remains of r chunks $b_{t+1}, \ldots, b_i$, where $r = t - i$. Recall that these chunks and the corresponding pairs of keys in the directories are shuffled according to a permutation σ induced by the update operations (see Section 2). To recover chunk $b_{\sigma(j)}$ in position j of the permutation, we have also to take the pairs of keys in positions $2j-1$ and $2j$ inside the directory.

- *Implicit:* u can encode up to $ik/2$ bits of auxiliary information partitioned into i *segments* of $k/2$ bits each. Each segment is encoded by the pairwise exchange of keys, so it needs k keys. The value of $k/2$ is enough to encode an initial reserved area that we will use shortly to encode the auxiliary information for the compactor lists, and two pointers to the children of u. It is crucial to keep the segments shuffled by the same permutation σ of the chunks described in the previous point.

The two ways of reading the keys in u are not aligned because the directories in u cause different displacements. But we can retrieve a chunk and its associated auxiliary information by reading the directory and accessing a suitable segment. Indeed the information encoded for the jth chunk in the permutation is encoded in the jth segment. As a result, the whole structure of the B-tree node is efficiently encoded by its keys alone. One important consideration is that of the information in the reserved area of segments. As we shall see, we must physically move the nodes from one part of the memory to another. In performing this task, we must take care to re-encode (permute) the information in the reserved area after reorganizing a node.

Keeping the above structure in mind, we can classify nodes according to their size. We pack those having identical size ik into a list labeled i to reflect that each of them contains i chunks of k keys. The total number of lists thus created is $t+1$.

Since the leaves have size ranging from tk to $2tk+k-1$, we treat them differently. We store each leaf as a *leading part* being of size a multiple of k, plus at most two *maniples* each of size at least $k/2$, indicated respectively as *primary* and *secondary*. This minimal length property allows us to encode logical pointers. The size of the leading part changes by k and that of the maniples by 1, as we preserve the following invariants:

1. If the primary maniple exists, then it contains between $k/2$ and $k-1$ keys. These keys are all greater than those in the leading part.

2. If the secondary maniple exists, then: (a) the primary maniple also exists; (b) the secondary maniple must contain exactly $k/2$ keys; (c) these keys are greater than those in the primary maniple and in the leading part.

3. The rest of the keys (i.e., those in the leading part) must be a multiple of k.

Note that we can preserve the invariants under the insertion and deletion of a single key by scanning all the keys in the leaf. It implies that the leading part contains ik keys with $t-1 \leq i \leq 2t$. As previously mentioned, the maniples have size ranging from $k/2$ to $k-1$. This motivates the creation of another collection of $k/2$ lists for the maniples, and an additional list $t-1$ to store the leading parts of size $(t-1)k$.

We refer to the two collections of lists above as *compactor lists*, as our allocation scheme hinges on their efficient manipulation. Specifically, we have $t+2$ lists for the B-tree nodes, numbered from $t-1$ to $2t$, where list i packs the B-tree nodes and the leaves' leading parts of size ik. We call them *tk-lists* to reflect that each list is made up of doubly linked *allocation units* of size tk.

We allocate the B-tree nodes to lists according to the size of the nodes. If a node is of size tk, it is stored in a single allocation unit. If its size is $2tk$, it occupies two allocation units. The remaining nodes contain each a number of keys that is not a multiple of the size tk of an allocation unit. Hence we pack the keys of each such node in at most three allocations units, where the first and the last units can be shared with other nodes. Each allocation unit contains the keys of at most two B-tree nodes. For the sake of presentation, we will discuss only B-tree nodes as the leading parts of leaves are handled identically in the lists.

We need a further property to encode the pointers in the list. Since the size of a unit is a multiple of k, the unit must start with a segment of a node and the auxiliary information of the unit is encoded in the reserved area of that segment. In particular, we encode the two pointers that link the allocation unit in its list, and the displacement of the beginning (if any) of the B-tree node inside that unit (the first key in a node is not necessarily the first key in the unit).

We also have $k/2$ lists for the maniples, numbered from $k/2$ to $k-1$, where list i packs the maniples of size i. We call them *k-lists* to reflect that each list is made up of doubly linked *allocation units* of size k. They are handled in a manner analogous to the tk-lists. Note that, since each maniple has size at least $k/2$, each allocation unit can store the keys of at most two maniples and encode two pointers for its list.

We point out the crucial property that only $O(1)$ pointers refer to a key or a node in any allocation unit. This follows form the fact that each allocation unit stores the keys of at most two B-tree nodes or maniples. It will be useful when relocating the units from one area to another of the memory in Section 4.

Resizing the maniples. We now discuss how to resize a maniple after inserting or deleting a key from its leaf. As previously mentioned, we can maintain the invariants on the maniples of the leaf by a linear scan of its keys with $O(tk/B) = O(\log_B n)$ block transfers. Because of this, we may have to change the size of the maniples and of the leading part of the leaf. As for the leading part, we can increase or decrease its size by k, as outlined in the next paragraph.

As the size of a maniple changes, we simply move it from one k-list to another. For example, a maniple can move from list i to list $i+1$ in case of an insertion without violating the size invariant, or vice versa in case of a deletion. If the list is $i = k-1$, an insertion removes the maniple from that list, and the k keys (including the newly inserted one) are added to the leading part. If $i = k/2$ and the maniple is primary, the deletion of a key may cause the redistribution of the keys with the secondary maniple (or its creation, if it does not exist, by taking the keys from its leading part).

There are few such cases to handle and all of them require moving one or two maniples from their k-lists. Consequently, we discuss how to move a maniple m from list i to list j. We first exchange m with the maniple at the beginning of list i to preserve the packing of the maniples in the list. We then change the size of m from i to j as required by the operation at hand. Next we move m from the beginning of list i to its proper destination at the beginning of list j. In other words, we reduce the problem of resizing a maniple to the problem of moving the *head* (i.e., allocation unit at the beginning) of two k-lists.

Lemma 7 *Resizing a maniple in the k-lists requires the relocation of $O(1)$ maniples plus $O\left(\left\lceil \frac{\log n}{B} \right\rceil \log_B n\right)$ block transfers.*

Proof: We relocate two maniples in the same list, and move the head from one list to another. After that, we need to identify the $O(1)$ leaves containing the relocated maniples by choosing one key per relocated maniple and searching that key in the data structure (Lemma 2). □

Resizing the B-tree nodes. We resize and relocate the leading part of a leaf after preserving the invariants on its maniples, and do the same with an internal node after a sequence of split or merge operations. Sharing keys with a sibling node can be seen analogously, as we increase the size by k in a node and decrease the size by k in the sibling node. In these cases, we move a node from list i to list $i+1$ (if a chunk of k keys is added) or from list $i+1$ to list i (if a chunk is deleted). Handling them is analogous to that of k-lists, except that we move nodes around instead of maniples, and allocation units are of size tk.

A different situation arises when the size of an internal node changes because of a split (Lemma 3) or a merge (Lemma 5). Although we logically change lists (from list t to list $2t$, or vice versa), we

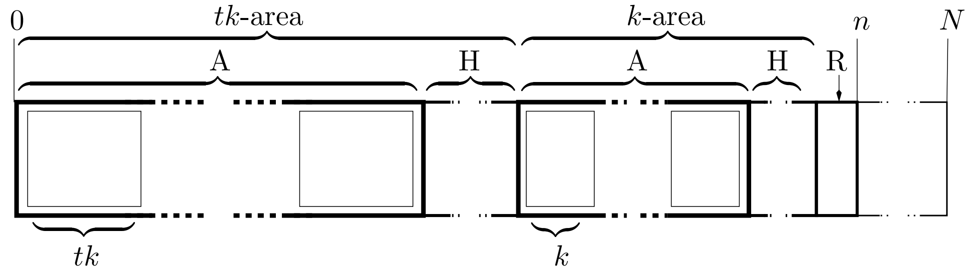

Figure 1: Low-level memory organization in three areas.

need *not* relocate the nodes and search in the data structure. It suffices to re-encode $O(1)$ pointers in the tk-lists because we keep the directories in the internal nodes.

Lemma 8 *Resizing a node in the tk-lists has a cost given by relocating* $O(1)$ *nodes plus* $O\left(\left\lceil \frac{\log n}{B} \right\rceil \log_B n\right)$ *block transfers. The cost of resizing reduces to* $O\left(\left\lceil \frac{\log n}{B} \right\rceil\right)$ *when splitting or merging the node (no relocation needed).*

Proof: We relocate two nodes in the same list, and move the head from one list to another. After that, we need to identify the $O(1)$ nodes whose encoded pointers refer to the relocated nodes. Again, we choose one key per relocated node and search that key in the data structure (Lemma 2). When splitting or merging, we only have to re-encode $O(1)$ pointers by accessing their keys in $O(k/B)$ block transfers. $\square$

We can now store the compactor lists by allocating contiguous memory to each allocation unit. As a result, we obtain an almost implicit data structure in that the heads of the lists are the only allocation units that are partially filled. In order to achieve a fully implicit data structure in n memory cells, we have to handle the floating heads of the compactor lists as described in Section 4.

4 Low-Level Memory Organization of the Compactor Lists

We are left with the problem of storing the compactor lists in adjacent memory cells with no waste of space. As previously noted, the allocation units are full except the heads of the lists. We have to deal with these floating heads by packing them in a suitable area of the memory in a dynamic setting. Given n adjacent memory cells in which we have to store the B-tree, we lay out the compactor lists according to the following low-level memory organization in three consecutive main areas (see Figure 1):

***tk*-area** contains all the tk-lists in a total space that is a multiple of k. The area is divided into two zones. Zone A stores all the allocation units that are not the heads in their lists. Hence, its size increases and decreases by tk. Zone H is floating and packs together the keys stored in the heads of the lists. Its size increases and decreases by k.

***k*-area** contains all the k-lists. Again, here zone A contains all the allocation units that are not the heads in their lists, and its size increases and decreases by k. Zone H is floating and packs together the keys stored in the heads of the lists. Its size increases and decreases by 1.

root area contains the root R of the B-tree which is treated separately from the other B-tree nodes. Its keys are stored at the end of the allocated memory area shown in Figure 1.

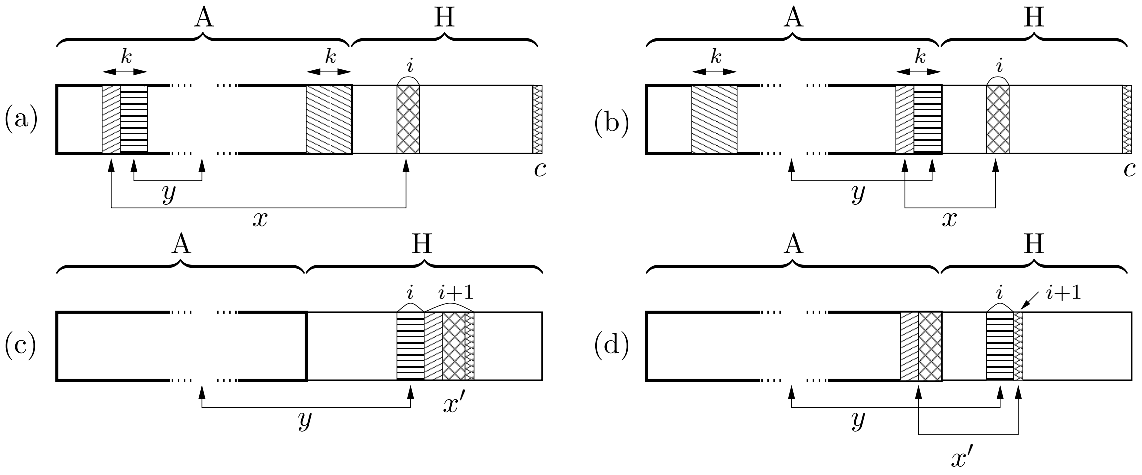

Figure 2: An example of relocation of maniple x being resized as maniple x' (with one more key) in the k-area. (a) The head of list i contains part of x, and the next-to-head contains the rest of x and part of y (if any). (b) The next-to-head is swapped with the last allocation unit in zone A. (c–d) After the addition of a new key in cell c to obtain x', maniple x disappears from list i and the next-to-head of list i becomes the new head containing part of y only; the new head of list $i+1$ becomes x'.

We also have to store the $t+2$ pointers to the allocation units that are heads of the tk-lists (there are $t+2$ of them) and a further $t+2$ pointers to the next-to-head allocation units. We also require the analogous pointers to the head and next-to-head units in the k-lists (there are $k/2$ of them), which adds up to further k pointers. Globally, if our data structure stores $\Omega((t+k)k)$ keys, we can encode these pointers and spread them in the reserved areas of the first segments in the tk-area. Then each of them can be retrieved by scanning $O(k)$ keys. (In the border case of $O((t+k)k)$ keys, we implement each operation on our data structure by a sequential scan of these few keys; see Section 5 for reducing their number.) We now have to show how to manage the compactor lists according to the above memory organization. By Lemma 7 and 8, we can focus on moving keys from one head to another.

Relocation in the k-area. Suppose we want to move the maniple x packed at the beginning of list i to the beginning of list $i+1$ because there is a new key added to x for obtaining x' with $i+1$ keys (the other operations are treated analogously). An example is shown in Figure 2. We begin by shifting all the $O(tk)$ keys in root R, one position to the right, to make room for one more key in zone H. Let c be this free memory cell (which we fill with the extra key), and y be the maniple that follows x in list i, such that the last keys in x and the first keys in y share the same allocation unit (y may not exist). Since x may occupy at most the first two allocation units of list i, we know that the first unit is in zone H of the k-area and the second unit, if any, is in zone A (Figure 2.a). We swap the latter unit with the last unit in zone A, at the cost of moving $O(k)$ keys and searching the data structure to identify the $O(1)$ leaves containing the involved maniples (Figure 2.b).

At this point, we have the head of list i in zone H and the next-to-head at the end of zone A, immediately adjacent to zone H. Scanning the $O(k^2)$ keys in the last unit of zone A and in the whole zone H, we can make y be the beginning of list i, and x' be the beginning of list $i+1$ (Figure 2.c). (We will describe in Section 5 how to scan fewer keys.) If x' contains k keys or less, we (logically) shorten the end of zone A by k positions and increase zone H by k positions to its left (this is the case illustrated in Figure 2.c). Otherwise, x' contains more than k keys: we take the last k keys in

x' and form an allocation unit in list $i+1$ to be placed at the end of zone A, and keep the rest of the keys in x' in the head of list $i+1$ in zone H (see Figure 2.d).

We need to update $O(k)$ pointers to the heads and next-to-heads (of k-lists) that are encoded in the tk-area, because the change of size of one head causes the shift of the others.

Lemma 9 *Relocating a maniple in the k-area requires $O\left(\left\lceil \frac{\log n}{B} \right\rceil \log_B n + k^2/B\right) = O(\left\lceil \frac{\log n}{B} \right\rceil \log_B n + \frac{\log^2 n}{B})$ block transfers.*

Proof: The total cost of the operation is given by scanning and/or moving $O(k^2)$ keys (in the whole zone H and in at most two allocation units of zone A), updating $O(k)$ pointers by scanning $O(k^2)$ keys (the pointers to the heads and the next-to-heads), and performing $O(1)$ searches (Lemma 2) in the data structure. □

Relocation in the *tk*-area. Suppose we want to move a node at the beginning of list i to list $i+1$. This is caused by the fact that we wish to add a chunk of k keys to an internal node or to the leading part of a leaf. In other words, a chunk of size k in the k-area has to be moved to increase the size of a node in the tk-area. From the discussion above on the relocation in the k-area, this chunk occupies the last allocation unit in zone A of the k-area. We exchange it with the first allocation unit in zone A, so that it immediately follows zone H of the tk-area. We logically increase the latter area by k positions to its right and decrease the k-area by k positions to its left. We now have all relevant keys in zone H of the tk-area, proceeding in a fashion similar to what done in the k-area.

Lemma 10 *Relocating a node in the tk-area requires $O\left(\left\lceil \frac{\log n}{B} \right\rceil \log_B n + t^2k/B\right) = O\left(\left\lceil \frac{\log n}{B} \right\rceil \log_B n\right)$ block transfers.*

Proof: The total cost of the operation is given by scanning and/or moving $O(t^2k)$ keys (in the whole zone H and in at most two allocation units of zone A), updating $O(t)$ pointers by scanning $O(tk)$ keys (the pointers to the heads and the next-to-heads), and performing $O(1)$ searches (Lemma 2) in the data structure. □

Notable exceptions are the merge and split operations in the internal nodes. Here, we logically merge or split two allocation units of tk keys each, and note that this does not require physically moving the heads of lists tk and $2tk$. The cost is just that of scanning $O(t+k)$ keys, even though the total size of the segments is $\Theta(tk)$, as anticipated in Lemma 3, Lemma 5 and Lemma 8.

As a general rule in our allocation scheme, a pointer from an internal node to one of its children uses a level of indirection if the child is head of one of the tk-lists. Similarly, the pointer in the leading part of a leaf uses indirection if the maniple is head of one of the k-lists. In order to check this rule, it suffices to see if the pointer refers to a reserved area inside a segment by a simple mathematical computation. This rule requires minor modification in our algorithms and does not change the complexity of the algorithms.

Lemma 11 *An implicit B-tree for n keys stored in n memory cells can be maintained under insertions and deletions with $O\left(\left\lceil \frac{\log n}{B} \right\rceil \log_B n + \frac{\log^2 n}{B}\right)$ block transfers per key in the worst case. Searching a key takes $O\left(\left\lceil \frac{\log n}{B} \right\rceil \log_B n\right)$ block transfers in the worst case. The only operations performed on the keys are comparisons and moves; and furthermore no additional space is required for the data structure. That is, the n memory cells store both the keys and the (implicit) data structure.*

Proof: The cost of searching derives from Lemma 2. For the insertion, each operation requires $O\left(\left\lceil \frac{\log n}{B} \right\rceil \log_B n\right)$ block transfers plus the cost of resizing $O(1)$ nodes by Lemma 4. Resizing the node not involved in a split or the leaf hosting one more key and its maniples take $O\left(\left\lceil \frac{\log n}{B}\right\rceil \log_B n\right)$ block transfers plus the relocation of $O(1)$ nodes and maniples by Lemma 7 and 8. Relocating $O(1)$ nodes and maniples take $O\left(\left\lceil \frac{\log n}{B}\right\rceil \log_B n + t^2k/B + k^2/B\right)$ block transfers by Lemma 9 and 10. Finally, it takes $O\left(\left\lceil \frac{\log n}{B}\right\rceil\right)$ for each split node (with at most $O(\log_B n)$ such nodes) by Lemma 8. Hence, the total cost for insertion is

$$O\left(\left\lceil \frac{\log n}{B}\right\rceil \log_B n + t^2k/B + k^2/B\right) = O\left(\left\lceil \frac{\log n}{B}\right\rceil \log_B n + \frac{\log^2 n}{B}\right),$$

since $k = O(\log n)$ and $t = \Theta(B^\epsilon)$, for a positive real constant $\epsilon < 1/2$. The analysis for a deletion is similar by replacing Lemma 4 by Lemma 6 in the analysis for an insertion. □

5 Refining the Solution

In this section, we refine the data structure in order to get better bounds by lowering the $\frac{\log^2 n}{B}$ additive term in Lemma 11. We focus on the case $B^{1-\epsilon} < \log_B n$ in the rest of this section, as that term is bounded by $O(\log_B n)$ when $B^{1-\epsilon} \geq \log_B n$. We must avoid accessing $\Theta(k^2)$ keys when relocating the maniples in the k-area (see Lemma 9). We deal with two situations:

1. We have to access the $\Theta(k)$ pointers to the heads and to the next-to-heads of k-lists. Recall that these pointers are encoded by $\Theta(k^2)$ keys in the tk-area, namely, in the reserved area of the segments in the first nodes of zone A.

2. We have to shift keys in zone H of the k-area, when resizing a maniple stored in one of the k-list heads. Recall that all heads of k-lists are packed together in zone H and contain $\Theta(k^2)$ keys in total.

In the following, when we refer to zone H, we mean inside the k-area.

Solving point 1 is fairly easy. We use a further level of indirection by encoding the $\Theta(k)$ pointers to the heads inside zone H and by putting a single pointer at the beginning of the tk-area to reflect this displacement. Note that each pointer encoded in zone H is tagged with the number j of its k-list. If the list is empty, we can avoid to store the pointer. Letting $g \leq k/2$ be the number of nonempty lists, at the beginning of the tk-area we keep the g pointers to the next-to-heads as they are not changed by a relocation of keys inside zone H. The number of encoded pointers at the beginning of the tk-area drops from $\Theta(t + k)$ to $\Theta(t + g)$, requiring thus only $\Theta((t + g)k)$ keys to set up our data structure, which is always guaranteed by definition of g. The smallest number of keys in this case is for $g = O(1)$ and an individual node satisfies the above requirement since it stores $\Theta(tk)$ keys. A larger value of g means that we have $\Omega(gk)$ additional keys, and so we may encode the g pointers. On the other hand, if we have $o(tk)$ keys, we can run brute-force updates and searches by scanning all the keys.

As for point 2, we extend the definition of heads in k-lists, so that nonempty lists have heads containing $\Theta(k)$ keys (e.g., we can unite the previously defined heads with their next-to-heads and make next-to-heads point to the successors of the resulting heads). The main idea is to fragment each head of a k-list in zone H into allocation units of size $\ell = \Theta(\log_B n)$, where k is a multiple of ℓ. In a certain sense, we wish to represent the heads in zone H as *ℓ-lists*, so that we transform the $k/2$ heads of the k-lists into $k/2$ lists whose allocation units have size ℓ inside zone H. The association rule is immediate, namely, the whole ℓ-list j stores the keys in the head of k-list j (so, a pointer to the latter head and the pointer to the former list are logically the same). The full allocation units of these ℓ-lists are stored at the beginning of zone H. The heads of the ℓ-lists are stored in a compact way at the end of zone H. Note that the latter heads contain $O(k\ell)$ keys, which we can fully scan with $O(k\ell/B) = O(\lceil\frac{\log n}{B}\rceil \log_B n)$ block transfers.

We encode the pointers to the heads and to the next-to-heads of the g nonempty ℓ-lists in the first $O(g)$ allocation units of size ℓ in zone H, as each such pointer requires $O(\ell)$ bits. We also tag each pointer with the number j of its ℓ-list, encoding j by a pairwise permutation of the keys. Note that, to have direct access to one of the g pointers to next-to-heads in point 1, we can scan the pointers to the g nonempty ℓ-lists in the first $O(g\ell)$ keys of zone H and find its rank h. Then, we simply decode the hth pointer to next-to-heads in point 1. At any time, the order of the pointers to the ℓ-lists in zone H reflects that of the g pointers in point 1.

Lemma 12 *Relocating a maniple in the k-area, as stated in Lemma 9, can be performed with $O\left(\left\lceil\frac{\log n}{B}\right\rceil \log_B n\right)$ block transfers.*

Proof: Let's first see how to recover the head of k-list j. We identify the head of ℓ-list j by scanning all heads of size $O(\ell)$ in the last part of zone H, examining $O(k\ell)$ adjacent keys with $O(k\ell/B) = O(\lceil\frac{\log n}{B}\rceil \log_B n)$ block transfers. We traverse the whole ℓ-list j in zone H with further $O(\lceil\ell/B\rceil k/\ell) = O(k/B + k/\ell) = O((\log n)/B + \log B) = O(\log_B n)$ block transfers, since $B^{1-\epsilon} < \log_B n$. Now, when relocating a maniple, we have to handle two heads, which can recovered and recompacted as described above, with $O(\lceil\frac{\log n}{B}\rceil \log_B n)$ block transfers, plus the cost of $O(1)$ searches (Lemma 2). We also need to update the pointers to the heads of the k-lists, encoded in the first $O(g\ell) = O(k\ell)$ keys of zone H. The total cost is $O(\lceil\frac{\log n}{B}\rceil \log_B n)$. When we have to resize one of the maniple, we can proceed similarly with a cost of $O(k/\ell + k\ell/B) = O(\lceil\frac{\log n}{B}\rceil \log_B n)$, as we can move single units of size $O(\ell)$. Note that it is not necessary to maintain the keys in sorted order. $\square$

We have described a single data structure for implicit B-trees. In general, we adopt the scheme by Frederickson [14], keeping $O(\log\log n)$ distinct data structures of size 2^{2^i}, $c \leq i \leq \lceil\log\log n\rceil$ (where c is a constant), so that the total cost is dominated by the cost of the largest of such data structures. Insertions are always performed on the largest structure; a deletion is performed by moving an element from the largest structure into the one where the element to be deleted is found. We obtain our main result.

Theorem 1 *An implicit B-tree for n keys can be maintained in n memory cells so as to support searching, inserting and deleting with $O\left(\left\lceil\frac{\log n}{B}\right\rceil \log_B n\right)$ block transfers in the worst case. Reporting r consecutive keys in sorted order takes $O\left(\left\lceil\frac{\log n}{B}\right\rceil \log_B n + r/B\right)$ block transfers.*

Proof: If $B^{1-\epsilon} \geq \log_B n$, we can use Lemma 11. Otherwise, it suffices to replace Lemma 9 by Lemma 12 in the proof of Lemma 11, noting that the cost of $O(\lceil\frac{\log n}{B}\rceil \log_B n)$ to retrieve at most two maniples in a leaf does not change the overall complexity. In order to report r consecutive keys in sorted order and belonging to a range $[x, y]$, let's suppose to have performed a search of x and y to identify two paths in the implicit B-tree. We then scan all the heads of size $O(\ell)$ in the last part of zone H, examining $O(k\ell)$ adjacent keys with $O(k\ell/B) = O(\lceil\frac{\log n}{B}\rceil \log_B n)$ block transfers to report those contained in the interval $[x, y]$. Then, it suffices to report all the keys traversed in the nodes delimited by the two paths. The procedure is essentially that of regular B-trees, with three main observations. (1) We do not have to scan the keys in the heads of size $O(\ell)$, because they are processed as mentioned above. (2) The $O(k/B + k/\ell) = O(\log_B n)$ cost of recovering the keys in the maniples of a leaf that are not in (1), is dominated by the cost $O(tk/B) = O(\log_B n)$ of accessing the rest of the keys in the leaf; indeed, $k/B + k/\ell = O(tk/B)$ if and only if $B/t < \ell$, which is the condition $B^{1-\epsilon} < \log_B n$ that we satisfy. (3) The $O(k)$ keys needed to decode a pointer either contribute to the cost $O(\lceil \frac{\log n}{B} \rceil \log_B n)$ or they the are part of the output, and so the cost of decoding that pointer is absorbed by the output-sensitive cost. To output the keys in order, it suffices to make an on-line merge the keys mentioned in (1) with the keys retrieved with the traversal of the implicit B-tree. □

Under the realistic assumption that $B = \Omega(\log n)$, the term $O(\lceil \frac{\log n}{B} \rceil \log_B n)$ in Theorem 1 reduces to $O(\log_B n)$ like in regular B-trees. Note that previous B-tree-like data structures occupied $n+\Omega(n)$ memory cells with the above bounds, while we occupy exactly n cells (the minimum possible with no waste of space) in $\lceil n/B \rceil$ blocks of memory.

Corollary 1 *When $B = \Omega(\log n)$, the implicit B-tree takes $O(\log_B n)$ block transfers per operation, reporting r consecutive keys in $O(\log_B n + r/B)$ block transfers. The memory occupancy is optimal, i.e., $\lceil n/B \rceil$ blocks of memory.*

In main memory, we can fix $B = \Theta(\log n)$ and multiply the number of block transfers by $O(B)$ to get the total (CPU) running time.

Theorem 2 *Fixing $B = \Theta(\log n)$, an implicit B-tree for n keys stored in n cells of main memory supports insertions, deletions and searches in $O(\log^2 n / \log\log n)$ time.*

6 Conclusions

We have introduced a new data structure, the implicit B-tree, to provide new bounds for searching and updating an implicit dictionary, which is entirely encoded by a suitable permutation of its keys. In hierarchy memory, these operations require $O\left(\left\lceil \frac{\log n}{B} \right\rceil \log_B n\right)$ block transfers in the worst case, and reporting r consecutive keys in sorted order takes $O\left(\left\lceil \frac{\log n}{B} \right\rceil \log_B n + r/B\right)$ block transfers. Under the realistic assumption that $B = \Omega(\log n)$, the above bounds reduce to $O(\log_B n)$ like in regular B-trees. In main memory, we can implement the dictionary operations in $O(\log^2 n / \log\log n)$ running time, improving a longstanding bound of $O(\log^2 n)$ for the problem.

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] Andrea Bello and Francesco Fantoni. Studio sperimentale di una nuova struttura dei dati implicita per B-alberi. Master thesis (in Italian), March 2003.

[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, 1109 Spring Street, Suite 300, Silver Spring, MD 20910, USA, 2000. IEEE Computer Society Press.

[4] 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.

[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] Zhiyuan Chen. Compression from the database perspective. In *DIMACS Working Group Meeting on Data Compression in Networks and Applications*, 2002.

[7] Amos Fiat, J. Ian Munro, Moni Naor, Alejandro A. Schäffer, Jeanette P. Schmidt, and Alan Siegel. An implicit data structure for searching a multikey table in logarithmic time. *Journal of Computer and System Sciences*, 43(3):406–424, December 1991.

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

[9] 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.

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

[11] Gianni Franceschini and Roberto Grossi. Optimal worst-case operations for implicit cache-oblivious search trees. In *Proceedings of the 8th International Workshop on Algorithms and Data Structures (WADS)*, 2003.

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

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

[14] Greg N. Frederickson. Self-organizing heuristics for implicit data structures. *SIAM Journal on Computing*, 13(2):277–291, 1984.

[15] 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.

[16] 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.

[17] 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.

[18] Microsoft. Microsoft developer network, 2003. http://msdn.microsoft.com.

[19] 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.

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

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

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

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

[24] Robert Sedgewick. *Algorithms in C++*. Addison-Wesley, Reading, MA, USA, 1992.

[25] SleepyCat Software. Berkeley DB, 2003. http://www.sleepycat.com.

[26] The Open Group. The single Unix specification, 2003. http://www.unix-systems.org/online.html.

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

[28] 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.

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