Alcom-FT Technical Report Series
ALCOMFT-TR-02-40

Implicit dictionaries for internal and external memory

Alon Itai *
Department of Computer Science
Technion - IIT, Haifa, Israel

Irit Katriel†
Max-Planck-Institut für Informatik
Stuhlsatzenhausweg 85, Saarbrücken, Germany.

Abstract

A data structure is called *implicit* if it uses a constant amount of space to store information other than the data it holds. In particular, it does not use more than a constant number of pointers.

In this work we design the Sparse Table, an implicit data structure for the dictionary problem which performs searches in $O(\log n)$ time and updates in $O(\log^2 n)$ amortized time. To the best of our knowledge, this is the first implicit dictionary that achieves $O(\log n)$ search time and polylogarithmic update time.

Based on the sparse table, we construct the B-table, an implicit data structure that achieves the same I/O complexity for queries as the B^+tree. The B-table is inferior to the B^+tree in the complexity of updates. However, it can be viewed as superior in terms of space utilization, as it offers a tradeoff between time and space. It is also the first I/O efficient implicit dictionary.

1 Introduction

A data structure is called *implicit* if it uses only a constant amount of space to store information other than the data it holds. It is *semi-implicit* if it uses $o(n)$ space for structural information. Pointer-based structures such as linked lists, skip lists, the various balanced search trees and more advanced structures such as Fibonacci heaps are all examples of structures which are not implicit: the structure depends on pointers and therefore needs to store $\Omega(n)$ information about itself.

An interesting theoretical question is whether pointers are merely a programming language utility or actually provide a computational advantage. That is, can everything be done just as fast without pointers? Note that it is possible to simulate a memory area with an array, and use indices of array locations as pointers. To avoid this, we define indices as pointers as well, and require that the data structure will store at most a constant number of pointers (or indices) at any moment in time.

On the practical level, eliminating pointers is a step towards better space utilization. The ultimate goal is efficient data structures which do not take up more than a constant amount of space in addition to the data itself. In the literature, this goal sometimes appears as the definition of implicit data structures (e.g., [9]). It is a suitable definition when static structures are considered. However, for dynamic structures, where elements can be inserted and deleted, the definition is relaxed such that it is permitted to leave some unused space, as long as it is not used to hold useful data (e.g., [12]).

1.1 Our results

In this work we propose an implicit dictionary that supports searches in $O(\log n)$ worst case time and updates (insertions and deletions) in $O(\log^2 n)$ amortized time. It is a modification of the sparse table [6], a data structure which was originally designed as a priority queue. The structure was modified to allow deletions and a new complexity proof is provided, which yields $O(\log^2 n)$ amortized time for both insertions and deletions. Another modification results in a more natural algorithm, which requires a somewhat different

*itai@cs.technion.ac.il
†irit@mpi-sb.mpg.de Part of this work was done while the second author was a graduate student at the Technion - IIT. It was partially supported by the Future and Emerging Technologies programme of the EU under contract number IST-1999-14186 (ALCOM-FT).

proof. The only other polylogarithmic solution we have found is that of Munro [8], which achieves $O(\log^2 n)$ time for searches as well as for updates. In addition to improving Munro's search time from $O(\log^2 n)$ to $O(\log n)$, the structure we propose is simple, hence solves a problem that Munro has left open: to find an easily implementable solution that runs in polylogarithmic time.

Based on the sparse table, we create an I/O efficient implicit dictionary which we call the B-table. The B-table is a sparse table aggregated by a hierarchical index that allows us to search for an item with $O(\log_B n)$ I/Os, compared to $O(\log_2 n)$ with the sparse table. In addition, this structure achieves near-optimal I/O complexity (external memory access count) for queries. This is also a property of B^+trees (a B-tree [1, 3] in which all data reside in the leaves). We compare the B-table with the B^+tree in terms of other properties. In particular, we show that the B-table offers a tradeoff between update time and space utilization. For a large database which is seldom updated, the B-table is preferable because it offers better space utilization than the B^+tree without sacrificing time and I/O complexity of queries. The B-table is the first I/O efficient implicit dictionary.

1.2 Previous work

Carlsson et al. [2] exhibited an implicit priority queue with constant insertion time and logarithmic extract-minimum time. Since n insertions and n extract-minimum operations perform a sort, there is an information theoretic lower bound of $\Omega(\log n)$ for at least one of them, for comparison-based algorithms. Hence, this result is optimum.

The implicit dictionary problem, however, is still open. Numerous balanced tree structures support searches, insertions and deletion in $O(\log n)$ time. However, the best implicit dictionary prior to our work achieves $O(\log^2 n)$ time for a search, insert or delete operation. The algorithm we propose here narrows the gap, but does not close it.

The first implicit dictionary was designed by Munro [12] and achieved $O(n^{1/3} \log n)$ time for each operation. This result was later improved by Fredrickson [5]. In 1984, Munro suggested the first polylogarithmic solution [8], which achieves $O(\log^2 n)$ search and update time. He later offered a variation on the same idea, which simplifies the original structure but does not change the complexity [10, 9]. Munro's solution is admittedly complex. He leaves as open the problem of finding an easily implementable polylog solution.

There are implicit structures which support searches and insertions (but not deletions). Munro [11] offered a structure that supports searches in $O(\log^2 n)$ worst case and $O(\log n)$ average time, and insertions in $O(\log n)$ time. This structure cannot support deletions. That is, it is not that deletions take a long or unknown time, but rather they cannot be performed at all because they would leave the structure in an unstable configuration.

Itai et al., suggested the sparse table in 1981 as a pointer-free priority queue [6]. That structure supports searches in $O(\log n)$ time and insertions in $O(\log^2 n)$ amortized time, but deletions from it are not efficient. As noted above, the solution we propose here is a modification of the sparse table.

In Section 2 we describe the sparse table. In Section 3 we describe the algorithm for inserting and deleting items. In Section 4 we prove the aforementioned complexity result for updates. In Section 5 we prove the same complexity result for a more elegant version of the sparse table. In Section 6 we describe the B-table and compare it with the B^+tree. This paper is based on an MSc thesis [7], which contains parts that are not included here for lack of space.

2 The Sparse Table

Our implementation follows the implicit priority queue model proposed in [6]: the n items are stored in an array of size m, sorted by increasing key order. The array is a *cyclic* one. That is, the first and last slots are viewed as neighbors. Three pointers are necessary: one pointer to the beginning of the array, another to its end, and a third to the location in which the minimal key is stored. Finding an item can be done in $O(\log m)$ time with a binary search. By maintaining that $m = O(poly(n))$, we achieve $O(\log n)$ search time. In our implementation, $m \leq 2n$.

Inserting a new item may require shifting records to make space for it. In order to achieve efficient insert time, the items are not stored contiguously but rather spaces are left between them. That is, $m > n$. This enables inserting a new item without necessarily shifting many existing records. Since the array is cyclic, we are sure to find an empty space, if one exists.

Whenever the number of items in the table is too large, a larger array is allocated and the records are distributed as evenly as possible in it. The condition for allocating a larger array is a part of the definition of the algorithm. In the one proposed in [6], the condition was that this number exceeds some fraction (say 80 percent) of the array's size.

In order to complete the description of the implementation, it is necessary to determine the algorithms for inserting and deleting items. A lazy approach for insertions was proposed in [6], by which existing items are shifted one place to the right until an empty slot is encountered, and no further work is done. It was shown that the expected number of such shifts per insertion is constant. Furthermore, it was shown that it is possible to ensure $O(\log^2 n)$ amortized worst case time for insertions by applying a local reorganization heuristic after each insertion. In order to do this, the array is viewed as if it is logically divided into blocks and an auxiliary binary tree structure exists with these blocks as leaves. This auxiliary structure plays a role in determining the sizes of the local reorganizations.

A scheme for deletions was not offered, and although is seems to be simple (just mark the slot as empty), it causes a problem. In order to be able to perform a binary search in $O(\log n)$ time, it must be possible, in time $O(1)$, to determine whether the search should continue to the left or to the right of a certain index. Therefore, it was suggested that all empty slots contain the key of the next occupied slot. An occupied slot can be identified by the fact that its key is different from the key in the next slot. With this implementation detail, it is clear how to perform a binary search but it is not obvious how to efficiently perform deletions: after deleting an item, all spaces between it and the next occupied slot must be updated.

We offer a different complexity proof, which applies to deletions as well. We then offer an analogous scheme which is more elegant in the sense that it views the array as a continuous sequence of slots, and does not require logically splitting it into blocks and imposing a tree structure on them. In both cases, we avoid the need to update empty slots after deletions by making sure that there are never two consecutive spaces. Then, empty slots can be marked with an invalid key value or a flag. During a binary search, if we need to decide which way to go according to an empty slot, we just look at one of its neighbors, which is guaranteed to contain a valid key.

3 The Update Algorithm

The algorithm for updating the array consists of three steps:

1. Search: find the slot for insertion (deletion), by a binary search.
2. Update: Insert (delete) the item to (from) its slot.
3. Reorganize: if necessary, perform a local reorganization or resize the array.

3.1 Local Reorganizations

In a *local reorganization*, the items in some consecutive sequence of array slots are redistributed in it, such that they are spread as evenly as possible. The algorithm that finds a good section to reorganize is described in detail in this section. When a good candidate section for a local reorganization cannot be found, the array is resized.

Definition: *An array section of size k containing x items is said to have* density x/k.

The purpose of the reorganizations is to bound the density in the vicinity of the slot that we have just updated. A future insertion to this area will be expensive if the density is very high and many items need to be shifted to make space for the new one. On the other hand, a future deletion from this area will be expensive if the density is very low, and many items need to be brought from far away to guarantee that there are no two adjacent empty slots. Hence, the reorganizations prevent local densities from becoming either too high or too low.

Following [6], we define $h = \lfloor \log m - \log\log m \rfloor$ and $b = \frac{m}{2^h}$. Note that $b = \Theta(\log m)$. We divide the array into $\frac{m}{b}$ blocks $B_0, B_1, \cdots, B_{2^h-1}$, where $2^h\lceil b\rceil - m$ blocks are of size $\lfloor b\rfloor$ and the rest are of size $\lceil b\rceil$.

We then construct a full binary tree of height h with leaves $L_0, L_1, \cdots, L_{2^h-1}$, such that for each internal node v, the indices of the leaves of the subtree rooted at v form a contiguous subsequence of $0, 1, \cdots, 2^h - 1$. Each leaf or internal node v of the tree is associated with a segment $s(v)$ of the array, as follows:

1. For $0 \le i < 2^h$, leaf L_i is associated with block B_i.
2. For an internal node v with children $\mathit{left}(v)$ and $\mathit{right}(v)$, $s(v) = s(\mathit{left}(v)) \cup s(\mathit{right}(v))$.

The tree plays a role in determining, upon an insertion or deletion, the size of the section that is to be reorganized. We will later show that the tree is not necessary, but first we show an algorithm that does use this tree, because it makes the complexity proof simpler and more intuitive. Note in the following that the tree does not actually need to be implemented. The algorithm can run as if it exists by computing the start and end of a block as necessary.

For each node v, denote by $m(v)$ the size of the array section $s(v)$ and by $e(v)$ the number of elements in $s(v)$. Let the *height* of a node be the number of edges on a path from the node to a leaf. Then it is easy to see that for a node v at height i, the size of $s(v)$, is 2^i blocks, that is: $m(v) = \Theta(2^i \log m)$. Below we describe the algorithm that decides which section of the array should be reorganized after an update operation. Before that, we show how to efficiently perform "good" reorganizations. But first, we need to define what makes a distribution of items good.

Definition *A distribution of items in $s(v)$ of some tree node v is q-balanced if for every internal node u in the subtree rooted by v, $|e(\mathrm{left}(u)) - e(\mathrm{right}(u))| \le q$.*

Lemma 1 *Given a section $s(v)$ of some tree node v, the items in $s(v)$ can be redistributed in it to give a 1-balanced distribution, in time $O(m(v))$.*

Proof Let r be the number of blocks in the section and x the number of items to be distributed in it. The algorithm will first determine how many items should be placed in each block, and then place the items in the blocks in sorted order.

We take x identical casino chips which we use as space-holders for the items, and begin by placing $\lfloor \frac{x}{r} \rfloor$ chips in each block. This gives a 0-balanced distribution, and leaves at most $r-1$ chips that were not placed in any block. For each of them, we traverse the tree starting from v, as follows:

1. $u \Leftarrow v$
2. while u is not a leaf:
   if $e(\mathit{left}(u)) > e(\mathit{right}(u))$ then $u \Leftarrow \mathit{right}(u)$.
   else $u \Leftarrow \mathit{left}(u)$.
3. put the chip in block $s(u)$.

Finally, we go over the blocks and replace the chips by the data items, in sorted order. In [7] we prove that the distribution obtained is 1-balanced and that the algorithm runs in $O(m(v))$ time. The proof is very simple and it omitted here. ∎

3.2 Determining which section should be reorganized

We set three constants $\frac{1}{2} < \tau_L < \tau_0 < \tau_U < 1$ such that $\tau_U - \tau_0 = \tau_0 - \tau_L = \delta$. The constant τ_L is a lower bound and the constant τ_U is an upper bound on the density of the whole array. The constant τ_0 is approximately the density in the array after it is resized (when resizing the array, we will select its size to be close to $\frac{n}{\tau_0}$, where n is the number of items). Based on τ_L, τ_0, τ_U, we define two *density threshold functions*:

$$\mathit{thresh}^I(k) = \tau_U - \frac{\log_2(k)}{\log_2(m)}\delta \quad ; \quad \mathit{thresh}^D(k) = \tau_L + \frac{\log_2(k)}{\log_2(m)}\delta.$$

The parameter k is the size of the array section that is begin considered. The parameter $\mathit{thresh}^I(k)$ is used for insertions, and $\mathit{thresh}^D(k)$ for deletions.

Scan: how far to reorganize?
Assume that an item was inserted into block p of the array. The *scan* procedure finds whether some array section that includes block p should be reorganized. It begins by checking the density in block p, and comparing it to $\mathit{thresh}^I(m(p))$. If the density is not larger than the threshold, no further work is necessary. If, however, the density in the block is above the threshold, the algorithm climbs up the tree towards the root, and for each node v it encounters, compares the density in $s(v)$ with $\mathit{thresh}^I(m(v))$. If the density in $s(v)$ is higher than $\mathit{thresh}^I(m(v))$, then $s(v)$ is too dense. Once the algorithm finds a section which is not too dense, it reorganizes the items in this section, as described above.

When the scan procedure climbs all the way to the root and does not find a section which is not too dense, the array is *resized*: a new array of size $\lceil \frac{n}{\tau_0} \rceil$ is allocated and the items are distributed in it with density $\frac{n}{\lceil n/\tau_0 \rceil}$. It is easy to see that $|\frac{n}{\lceil n/\tau_0 \rceil} - \tau_0| < \frac{\tau_0}{n}$. Hence if we denote the density after resizing the array by $\hat{\tau}_0$, we have that $\tau_0(1-\frac{1}{n}) < \hat{\tau}_0 < \tau_0(1+\frac{1}{n})$.

After *deleting* an item, we need to make sure that there are not two adjacent empty slots. If there are, the lazy approach is to shift items towards the excess empty slots, without moving items more than the minimum that is required to solve the local problem. However, the worst case time of that would be $O(n)$, as in the lazy algorithm for insertions. Therefore, we apply local reorganizations to deletions as well. The algorithm for finding whether a reorganization is necessary is analogous to the one for insertions, with the difference being that it looks for a section $s(v)$ for some node v on the path from the root to the block from which an item was deleted, which is not too *sparse*. That is, which has a density of at least $\mathit{thresh}^D(m(v))$. Once such a section is found, it is reorganized. If the algorithm reaches the root of the tree without finding such a section, the array is resized (this time a smaller array is allocated) and the items are distributed in it to obtain a 1-balanced distribution.

4 Complexity Analysis

Denote by t^S, t^U and t^R, the actual time spent by the algorithm during the Search, Update and Reorganize steps of an insertion/deletion, respectively. Then the total actual time spent is $t^S + t^U + t^R = O(\log m) + O(\mathit{block\ \ size}) + O(d) = O(\log n) + O(d)$, where d is the amount of work spent on the reorganization or resizing, and by Lemma 1 is proportional to the size of the affected section of the array. We use the accounting method to show that the amortized cost of insertions and deletions is $O(\log^2 n)$.

Suppose that a reorganization of size d costs d Euros, and that resizing the array costs $\max\{m, m'\}$ Yens, where m is the size of the array before the resizing and m' its size after. Each node of the tree has a Euro account with which it pays for reorganizations of its section. The whole structure has a Yen account, with which it pays for resizing the array. When an item is inserted into or deleted from block p of the array, it deposits $\frac{\log m}{\delta}$ Euros in the account of each node on the path from the root to p. In addition, it deposits $\frac{4 \log m}{\delta}$ Yens in the structure's resizing account. Since there are $O(\log m)$ nodes on the path from the root to any leaf and $m \le 2n$, the total sum that the item deposits when it is inserted is $O(\log^2 m) = O(\log^2 n)$.

4.1 Paying for reorganizations

The following lemma gives, for any section of the array that corresponds to a node v in the tree, a lower bound on the number of insert or delete operations that are performed between two reorganizations of this section.

Lemma 2 *For any internal node v of the tree, between two reorganizations of $s(v)$, there are at least $\frac{m(v)\delta}{2\log m} - O(1)$ update operations in $s(v)$.*

Proof First we assume that $m(v) = 2k$ and $m(\mathit{left}(v)) = m(\mathit{right}(v)) = k$. Let x_1 denote the number of items in $s(\mathit{left}(v))$ and x_2 the number of items in $s(\mathit{right}(v))$. By Lemma 1, $|x_1 - x_2| \le 1$ after the array is resized or $s(v)$ is reorganized. Suppose that another reorganization of $s(v)$ is about to be performed. There are two cases:

- Case 1: The reorganization was triggered by an insertion. Assume, w.l.o.g., that the insertion was to $s(\mathit{left}(v))$. By construction of the algorithm, we know that the density of $s(\mathit{left}(v))$ was too high, because otherwise the scan would have stopped earlier, and that the density in $s(v)$ is not too high, because otherwise the scan would have continued. That is:

$$\frac{x_1}{k} > \mathit{thresh}^I(k) = \tau_U - \frac{\log k}{\log m}\delta \quad ; \quad \frac{x_1+x_2}{2k} \leq \mathit{thresh}^I(2k) = \tau_U - \frac{\log 2k}{\log m}\delta \ .$$

Or: $$x_1 > k\tau_U - \frac{k\log k}{\log m}\delta \quad ; \quad x_2 \leq k\tau_U - \frac{k\delta}{\log m}\left[2\log 2k - \log k\right] \ ,$$

Giving: $$x_1 - x_2 > \frac{2k\delta}{\log m} = \frac{m(v)\delta}{\log m} \ .$$

- Case 2: The reorganization was triggered by a deletion. Again, assume without loss of generality that the deletion was from $s(\mathit{left}(v))$. We know from the construction of the algorithm, that

$$\frac{x_1}{k} < \mathit{thresh}^D(k) = \tau_L + \frac{\log k}{\log m}\delta \quad ; \quad \frac{x_1+x_2}{2k} \geq \mathit{thresh}^D(2k) = \tau_L + \frac{\log 2k}{\log m}\delta.$$

Similarly giving: $$x_1 - x_2 > \frac{2k\delta}{\log m} = \frac{m(v)\delta}{\log m} \ .$$

In both cases, for an insertion or deletion to trigger a reorganization, the difference between the number of items in $s(\mathit{left}(v))$ and the number of items in $s(\mathit{right}(v))$ increased from at most 1 to at least $\frac{2k\delta}{\log m}$. The only operations that can increase this difference are insertions to and deletions from $s(v)$, and each of them increases the difference by at most 1. Therefore, between two reorganizations of $s(v)$, at least $\frac{2k\delta}{\log m} - 1$ insert/delete operations must be performed in s(v). In [7] we show that the lemma holds also when we drop the assumption that $m(\mathit{left}(v)) = m(\mathit{right}(v))$. The proof is along the same lines as the one above; it is only slightly more complicated, and it is omitted here. ∎

Corollary 1 *A reorganization of $s(v)$ costs at most $O(\frac{\log m}{\delta})$ Euros more than what v has in its account.*

Proof By Lemma 4.1, between two reorganizations of $s(v)$, at least $\frac{m(v)\delta}{2\log m} - O(1)$ items are inserted or deleted from $s(v)$. Each of them deposits $\frac{2\log m}{\delta}$ Euros to v's account. Therefore, there are at least $(\frac{m(v)\delta}{2\log m} - O(1)) \cdot \frac{2\log m}{\delta} = m(v) - O(\frac{\log m}{\delta})$ Euros in the account. The corollary follows. ∎

4.2 Paying for resizing the array

Suppose that an insertion triggered a resizing of the array. Let m be its size before the resizing and m' after. Since $n \leq m$, $\tau_0 > \frac{1}{2}$ and for $n > 4$, $m' = \frac{n}{\hat{\tau}_0} < \frac{m}{\tau_0 - \frac{1}{n}} < 4m$. Hence the resizing costs at most $4m$ Yens.

By construction of the algorithm, we know that $s(v)$ for a node v which is a child of the root, is too dense before the resizing. Otherwise the scan wouldn't have decided that the array should be resized. That is, a section of size $k = \frac{m}{2}$ or $k = \frac{m-1}{2}$ has density higher than $\mathit{thresh}^I(k)$. Therefore the number of items in this section is at least $\frac{m}{2}\tau_U - \frac{m}{2}\frac{\log m/2}{\log m}\delta$ in the first case and at least $\frac{m-1}{2}\tau_U - \frac{m-1}{2}\frac{\log(\frac{m-1}{2})}{\log m}\delta$ in the second case. After the previous resizing, the number of items in the array was $\frac{m}{2}\hat{\tau}_0$. Therefore in both cases, at least $\frac{(m-1)\delta}{2\log m} - \frac{m}{2n} < \frac{(m-1)\delta}{2\log m} - 1$ items were added during this phase. Each of the insertions deposited $\frac{8\log m}{\delta}$ Yens in the structure's account, therefore it has at least $4(m-1) - \frac{8\log m}{\delta}$ Yens and it can pay for the resizing, short of $O(\log n)$ Yens. Resizing the array, then, costs $O(\log n)$. An analogous calculation shows the same for the case in which a deletion triggers a resizing of the array.

Theorem 4.1 *The amortized complexity of insertions and deletions is $O(\log^2 n)$.*

Proof By Corollary 1, the amortized cost of a reorganization is $O(\log n)$ Euros. Therefore, the cost of insertions is $O(\log n)$ for the search, $O(\log n)$ for updating the contents of one block, $O(\log n)$ for the reorganization and $O(\log^2 n)$ for depositing money, giving a total of $O(\log^2 n)$.

5 Eliminating the tree

In this section we show that although the binary tree makes the algorithm and proof more intuitive, it is not necessary. In the algorithm, the role of the tree is to define the sections that could be reorganized: only a section $s(v)$ which corresponds to a node v of the tree can be reorganized. We now propose a different scheme.

After an insertion to (deletion from) slot c of the array, the scan procedure looks at the array section consisting of slots $c, c+1, \cdots, c+b-1$, where $b = \Theta(\log m)$ and all indices are computed modulu m. Like the previous algorithm, it will compare the density of items in this section with $\mathit{thresh}^I(b)$ ($\mathit{thresh}^D(b)$).

If the section is too dense (sparse), *scan* looks at a section double in size, i.e., at slots $c, c+1, \cdots, c+2b-1$. We continue in this fashion until finding a section $c, c+1, \cdots, c+2^ib-1$ of size 2^ib. When *scan* finds a section of size 2^ib which is not too dense (sparse), it reorganizes it. If the entire array is scanned without finding such a section, the array is resized.

The exact fashion by which the section is enlarged is not important. It could grow equally in both directions, or grow only to one direction (which is perhaps easier to implement). Note also that the possible sections are not the same as with the tree-based scheme, where a section was a collection of pre-defined blocks.

As before, the sizes of reorganizations are restricted: they can only be of a size 2^ib for integer i. But unlike in the tree-based algorithm, we cannot claim that for two reorganizations of sizes k, k' where $k' > k$, the k-size section must be either included in or disjoint from the k'-size section. This possible overlapping calls for a different proof. Note that in the new algorithm, unlike in the tree based one, we can assume that the size of a reorganization is always an even integer.

5.1 Complexity analysis

Again, we use the accountant method, but the Euros which pay for reorganizations go into accounts that belong to the array slots instead of the tree nodes, which no longer exist.

We begin by a complexity analysis of insertions. Each array slot has $\log_2 \frac{m}{b}$ insertions accounts, which are numbered $1, 2, \cdots, \log_2 \frac{m}{b}$. The ith account of slot x, which we denote by $A_i(x)$, is used to pay for insertion-triggered-reorganizations that involve x and are of size $2^{i+1}b$.

When an item is inserted, the algorithm deposits $\frac{8\log m}{\delta}$ Euros in each of the accounts of the slot that the item was inserted to. This, again, requires depositing a total of $\sum_{i=1}^{\log_2 \frac{m}{b}} \frac{8\log m}{\delta} = O(\log^2 m)$ Euros.

We denote the balance in $A_i(x)$ by $\mathit{bal}_i(x)$. For a section s of the array of size k which consists of slots $x_1, x_2, \cdots, x_k$, we denote by $\mathit{bal}_i(s) = \sum_{j=1}^{k} \mathit{bal}_i(x_j)$ the total sum in the A_i's of the slots in the section.

Socialism rules: After a reorganization is performed, then for all $1 \le i \le \log_2 \frac{m}{b}$, the Euros in the A_i's of all the slots in the section are distributed evenly among the occupied slots of the section. Formally: let ℓ be the number of items in the section and $x_1, \cdots, x_\ell$ the indices of occupied slots after the reorganization then for each $1 \le i \le \log_2 \frac{m}{b}$ and $1 \le j \le \ell$, we set $\mathit{bal}_i(x_j) = \frac{\mathit{bal}_i(s)}{\ell}$. When an item is deleted, the emptied slot keeps whatever is in its accounts. After the array is resized, the balance in all accounts is set to 0.

The following lemma states that when a section of the array is reorganized or the array is resized, there are enough Euros to pay for it in the relevant accounts of the slots of this section (in other words: the balance is never negative). It does that by proving a certain invariant relating the balance in the accounts with the density of the section.

Lemma 3 *Let τ be the density of elements in a section s of the array of size k. Then for every $1 \le i \le \log_2 \frac{m}{b}$,*

1. $\mathit{bal}_i(s) \ge 8k\frac{\log m}{\delta}\left[\tau - \mathit{thresh}^I(2^{i+1}b)\right]$.
2. $\mathit{bal}_i(s) \ge 0$.

Proof

We prove this by induction on the operations performed on the array. Note that it is only necessary to prove that a change in the array preserves the conditions in an array section that was affected by it, because both conditions are linear in k.

- Assume that a reorganization of a section of size $2^{i+1}b$ was performed. Denote the first half of this section by s_1 the second half by s_2. Let τ_1 be the density in s_1 and τ_2 the density in s_2, before the reorganization. By the induction hypothesis, before the reorganization, for every $1 \le j \le \log_2 \frac{m}{b}$,

  $$bal_j^{before}(s_1) \ge 8 \cdot 2^i b \frac{\log m}{\delta}\left[\tau_1 - thresh^I(2^{j+1}b)\right] \quad ; \quad bal_j^{before}(s_2) \ge 8 \cdot 2^i b \frac{\log m}{\delta}\left[\tau_2 - thresh^I(2^{j+1}b)\right]$$

  and both balances were non-negative.

  The reorganization was paid for with the A_i's of slots in the section. For every $j \ne i$, the money in the A_j's was not spent, but due to *socialism* was distributed evenly between all the slots. So for $j \ne i$:

  $$bal_j^{after}(s_1) \quad = \quad bal_j^{after}(s_2) = \frac{1}{2}(bal_j^{before}(s_1) + bal_j^{before}(s_2)) \ge 8 \cdot 2^i b \frac{\log m}{\delta}\left[\frac{\tau_1 + \tau_2}{2} - thresh^I(2^{j+1}b)\right].$$

  Since the density after the reorganization is $\frac{\tau_1+\tau_2}{2}$, this proves Condition 1 for $j \ne i$. Condition 2 is trivially preserved.
  For $j = i$, the above does not hold because $bal_i(s_1) + bal_i(s_2)$ has changed due to the sum paid for the reorganization. By construction of the algorithm, we know that s_1 was too dense before the redistribution (otherwise the scan wouldn't have continued to a larger section). That is, $\tau_1 > thresh^I(2^i b)$. By the induction hypothesis,

  $$\begin{aligned} bal_i^{before}(s_1) \quad &\ge \quad 8 \cdot 2^i b \frac{\log m}{\delta}\left[\tau_1 - thresh^I(2^{i+1}b)\right] \\ &> \quad 8 \cdot 2^i b \frac{\log m}{\delta}\left[thresh^I(2^i b) - thresh^I(2^{i+1}b)\right] = 8 \cdot 2^i b \frac{\log m}{\delta}\frac{\delta}{\log m} = 8 \cdot 2^i b, \end{aligned}$$

  which is more than the cost of the reorganization. Hence, there are enough Euros to pay for the reorganization, so $bal_i^{after}(s_1) = bal_i^{after}(s_2) \ge 0$ and Condition 2 holds.
  After the reorganization, the density in s is $\tau = (\tau_1 + \tau_2)/2 \le thresh^I(2^{i+1}b)$, hence again $\tau - thresh^I(2^{i+1}b) \le 0$ and Condition 1 also holds.

- We now consider the effect of inserting an item to slot c of the array. That is, just the actual addition of an item, without any reorganization work. For a section s of size k that includes slot c, the density increased from τ to $\tau + \frac{1}{k}$. For every $0 \le i \le \log \frac{m}{b}$, $bal_i^{after}(s) = bal_i^{before}(s) + \frac{8 \log m}{\delta}$.
  By the induction hypothesis, for all $1 \le i \le \log_2 \frac{m}{b}$,

  $$bal_i^{before}(s) \ge 8k \frac{\log m}{\delta}\left[\tau - thresh^I(2^{i+1}b)\right],$$

  hence

  $$bal_i^{after}(s) \ge 8k \frac{\log m}{\delta}\left[\tau + \frac{1}{k} - thresh^I(2^{i+1}b)\right],$$

  which proves Condition 1. Condition 2 is trivially preserved.

- Assume that an item was deleted from slot c. We are, at this stage, concerned only with the complexity of insertions. Hence, we do not need to account here for the cost of performing the deletion. We only need to show that the conditions regarding the insertions-accounts are preserved.
  In any section which contains slot c, the balance in the insertions accounts is unchanged, and the number of items is smaller, hence the density decreased. Therefore both conditions of the lemma are preserved.

- Before an insertion-triggered resizing of the array, the density in a section s with size $\frac{m}{4} \le k \le \frac{m}{2}$ must have been too dense. That is, the density in it, τ, was at least $thresh^I(\frac{m}{2})$. By the induction hypothesis,

  $$bal_i(s) \ge \frac{m}{2}\frac{4 \log m}{\delta}\left[thresh^I(\frac{m}{2}) - thresh^I(m)\right] = 2m \ .$$

Since the reorganization costs at most $2m$ Euros, this is enough to pay for it and the balance remaining is non-negative, proving Condition 2.
After the array is resized, the density in it, and in any subsection of it, is $\hat{\tau}_0$. The balance in any section of the array is 0. Since $\hat{\tau}_0 < \mathit{thresh}^I(2^{i+1}b)$ for all i, $8k\frac{\log m}{\delta}\left[\tau - \mathit{thresh}^I(2^{i+1}b)\right] < 0$, which completes the proof. ∎

Deletions
The complexity analysis of deletions is analogous. To each slot x of the array, we add $\log \frac{m}{b}$ deletions accounts, which are numbered $1, \cdots, \log \frac{m}{b}$. When an item is deleted, it deposits $\frac{8 \log m}{\delta}$ Euros in each of the deletions accounts of the slot it was in. For a slot x of the array, we denote the balance in the ith deletions account by $\mathit{dbal}_i(x)$. For a section s of size k which consists of slots $x_1, \cdots, x_k$, we denote $\mathit{dbal}_i(s) = \sum_{j=1}^{k} \mathit{dbal}_i(x_j)$.

When a section is reorganized (due to an insertion or a deletion), the socialism rules are applied both to the insertions accounts as before, and to the deletions accounts. That is, using the same formalization, for each $1 \leq i \leq \log_2 \frac{m}{b}$ and $1 \leq j \leq \ell$, we set $\mathit{bal}_i(x_j) = \frac{\mathit{bal}_i(s)}{\ell}$ and $\mathit{dbal}_i(x_j) = \frac{\mathit{dbal}_i(s)}{\ell}$. Recall that a deletion does not change the balance in the insertions accounts of the affected slot. In the same manner, an insertion does not affect the balance in the deletions account of the slot. We can then prove the following lemma:

Lemma 4 *Let τ be the density of elements in a section s of the array of size k. Then for every $1 \leq i \leq \log_2 \frac{m}{b}$,*

1. $\mathit{dbal}_i(s) \geq 8k\frac{\log m}{\delta}\left[\mathrm{thresh}^D(2^{i+1}b) - \tau\right]$.
2. $\mathit{dbal}_i(s) \geq 0$.

The proof is analogous to that of Lemma 3.

Theorem 5.1 *Insertions and deletions take $O(\log^2 m)$ amortized time.*

Proof
By Lemma 3, reorganizations and resizings can always be paid for with the money that is deposited during insertions and deletions. Their amortized cost is therefore $O(1)$. The cost of an update is $O(\log n)$ for finding the place for insertion or deletion, $O(1)$ for the update and reorganization and $O(\log^2 n)$ for depositing money. In total: $O(\log^2 n)$.

6 The B-table

6.1 I/O complexity

In the commonly used random-access computational model, a program can access any memory address in constant time. In practice, this model is partial. Computers typically have a small internal (main) memory and store most of the data on slow secondary storage (e.g., hard disk). If the program accesses data which is not currently in the main memory, the data is fetched from secondary storage, and this is a costly operation: it is realistic for a real life application to spend 80% of its running time on such accesses.

The external memory computational model is as follows: memory is divided into blocks of size B (B is the number of data structure records that fit into one block). Data is read from and written to the external memory in whole blocks, regardless of how much data from the block is actually needed. M blocks fit into the main memory. When the main memory is full and a new block needs to be brought, a replacement algorithm (e.g. FIFO or LRU) determines which page in the main memory will be overwritten.

The *I/O complexity* of an algorithm is the number of block-read and block-write operations performed. Obviously, reading n data items requires at most n I/Os (if each item is in a different block) and at least $\frac{n}{B}$ I/Os if the n items are neatly packed in $\frac{n}{B}$ blocks. A notable feature of the sparse table as a search structure is its near-optimal I/O complexity of sequential accesses (range queries): reading k consecutive records requires at most $\left\lceil \frac{k}{\tau_L B} \right\rceil + 1 = O(\frac{k}{B})$ I/Os. This is also a property of the B^+tree (a B-tree [1, 3] in which the size of a node is equal to the size of a memory block and all the data records are in the leaves).

In terms of I/O complexity of searches, the B^+tree has $O(\log_B n)$ I/O complexity, compared to the sparse table's $O(\log_2 n)$. For this reason, we construct the B-table, which is based on the sparse table and achieves $O(\log_B n)$ I/O complexity for searches. We will later compare the B-table and the B^+tree in terms of other properties.

6.2 A semi-implicit B-table

We describe the B-table in two steps. First, we show a semi-implicit B-table, and later show how to turn it into an implicit structure with similar properties. The semi-implicit B-table is stored in a single array. The first m locations of the array are a sparse table over the elements in the structure. We will call them level-0 of the B-table. The next $\frac{m}{B}$ locations are level-1 and serve as an index over the keys of the B-size blocks of level-0. That is, the i-th key in level-1 is the smallest key of block i of the sparse table, i.e., locations $(i-1)B, \ldots, iB-1$. This index is dense—all its slots contain keys. Similarly, for all $i > 0$, level-i is an index over the B-size blocks in level-$(i-1)$ and requires $\frac{m}{B^i}$ array locations.

The structure of the index changes only when level-0 is resized. Thus all during the phase the *root*, the highest level of the index, resides in the last block of the structure, its children in the previous s blocks, where s is the number of keys in the root. The next higher level occupies the sB previous blocks etc. Given the address of a block b, one can compute the address of its parent and jth child solely from m. Thus there is no need to hold pointers in the index.

When a sparse table would have required an array of size $m = \frac{n}{\tau_0}$, the B-table requires an array of size

$$\sum_{i=0}^{\log_B m} \frac{m}{B^i} = \frac{mB-1}{B-1} < m\frac{B}{B-1} = m + \frac{m}{B-1}\ .$$

The amount of space used for the index (non-data information) is at most $m/(B-1) \leq 2n/(B-1) = o(n)$, so the structure is semi-implicit.

A search for key x starts at the root. Suppose that $x_j \leq x < x_{j+1}$, $(x_j, x_{j+1} \in \mathit{root})$. We continue to block j of the next lower level. If the search leads us to level-0 blocks and x is not found, then x does not belong to the table. Thus, in a sense, the B-table has the structure of a B^+tree. Since the number of levels is $\log_B n$, this is also the number of I/Os required for a search.

Updating the B-table is done as follows: the update is done at level 0, just as in a sparse table. Note that when inserting key x to a block b it is always larger than the first key of b. Thus insertions do not require updating the index. Also one may delete the first key of a block without need to update the index. Only reorganization that span more than one block trigger updates to the index.

After reorganizing a level-0 section of $q > 1$ blocks, the identity of the first key of $q-1$ blocks has changed, thus it is necessary to change $q-1$ keys in the index. This requires updating a section of size at most $\frac{q}{B^i}$ in level i. Note that the structure of the index does not change, just some keys are replaced by others. The total time required for this is $\sum_{i=0}^{\log_B q} \frac{q}{B^i} = O(q)$. In other words, the reorganization of the index does not increase the complexity of an update by more than a constant factor. When the sparse table (level-0) is resized the entire index is recalculated in linear time.

6.3 An implicit B-table

To turn the semi-implicit B-table into an implicit one, we need to eliminate the duplication of keys in the structure. If the semi-implicit structure resembles a B^+tree (in which all data is in the leaves), the implicit variant resembles a B-tree (in which the data items can reside also in internal nodes).

One may view the implicit structure as a semi-implicit B-table composed of virtual blocks of size $B+1$. The physical layout avoids duplication by storing all but the first item of each virtual block in a physical block of size B. The first item of a level-i virtual block (except the root) is stored only in higher levels physical blocks. Thus, the first key of a level-0 virtual block either resides in a level-1 physical block, or if it is the first key of the level-1 virtual block, in a block of a higher level.

The search is similar to that of the semi-implicit structure, except that one should stop when finding the key in an index, not only at level-0.

If a reorganization involves several blocks, the identity of the first key of all but the first block of the reorganization changes. Thus we have to move the new first key of block b to the index and bring the first key of the corresponding virtual block from the index back to block b. It is easy to organize these swaps so that every index block is read and written at most once. If q level-0 blocks were involved in the reorganization, at most $\lceil q/B \rceil + h$ index blocks were involved, where $h = \log_B m$ is the height of the index. Thus the extra cost of accessing the index blocks is at most a constant multiplicative factor and an $O(h)$ additive factor.

When the table needs to be resized, the entire index is reorganized from scratch. As before, the cost is linear in the size of the table. Deletions are conducted in a similar fashion. If the deleted key resided in the index, it is replaced by a level-0 key. For a range query with output of size k, not only the relevant range of the sparse table needs to be scanned, but also the corresponding sections of the index, because they contain data that should be output. The I/O complexity of this, is $O(k/B + \log_B m)$ as with the semi-implicit structure.

6.4 B^+tree vs. B-table

I/O complexity of updates

Updating a B^+tree triggers $O(\log_B n)$ I/Os while traversing a path from the root to a leaf, and possibly climbing back up towards the root to perform merges or splits of nodes. Updating the B-table requires finding the place for insertion or deletion in $O(\log_B n)$ I/Os, inserting or deleting an item and then possibly reorganizing. The amortized size of the reorganized section of L_0 is $O(\log_2^2 n)$ and therefore $O((\log_2^2 n)/B)$ I/Os occur. The total amortized number of I/Os for updating all levels as necessary is therefore $O(\log_B n + (\log_2^2 n)/B)$. The total I/O complexity for an update is then also $O(\log_B n + (\log_2^2 n)/B)$. For $B \geq (\log_2^2 n)/\log_B n$, the first term dominates the second and the I/O complexity of updates on the B-table is as that of the B^+tree. Note that this condition, while not guaranteed, is realistic because B is only meaningful if it is not very small. For example, for $B = 500$, the condition holds for all $n \leq 10^{16}$.

Space utilization

The worst case space requirement for an n-item B^+tree is $SP_{B+tree} = 2n$, when all nodes of the tree are half full. Fringe analysis [13, 4] shows that the expected space utilization is about $\ln 2 < 0.7$. Therefore, the expected size of a B^+tree is at least $E(SP_{B+tree}) > \frac{n}{0.7}$. The size of a B-table, however, is at most $m\frac{B}{B-1}$. Since $m \leq \frac{n}{\tau_L}$, we get that $SP_{B-table} < \frac{n}{\tau_L}\frac{B}{B-1}$. If we select $\tau_L \geq 0.7\frac{B}{B-1}$, then $SP_{B-table} < E(SP_{B+tree})$.

In the sparse table complexity analysis, we regarded τ_0, τ_U, and therefore also δ, as constants. However, to increase the worst case space utilization we need to increase τ_0, which decreases the maximal value that δ can have. If we return to the complexity analysis and do not regard δ as a constant, we get that the amortized update time is $O(\frac{\log^2 n}{\delta})$. In other words, there is a tradeoff between the density of the array and update time. Note that only the update time grows with the density. The query times only improve when the percentage of empty array slots shrinks. Hence, if we have a large database which is not frequently updated, we might prefer the B-table over a B^+tree. Table 1 summarizes some properties of the B^+tree and the B-table.

7 Conclusion and open questions

Prior to our work, the best known implicit dictionary had $O(\log^2 n)$ time complexity for every operation. It is based on a scheme of encoding pointers with the structure's data contents, which seems to inherently multiply the time complexity by an $O(\log n)$ factor (decoding such a pointer takes $O(\log n)$ time instead of $O(1)$).

Using the concept of local reorganizations, we achieved a better pointer-free dictionary. We do not know whether $O(\log^2 n)$ is also a lower bound for the algorithm we propose. Of course, the main open question is whether a pointer free dictionary exists with $O(\log n)$ complexity for all operations.

		B-Table	B^+tree
Search Time		$O(\log n)$ worst case	$O(\log n)$ worst case
Update Time		$O(\log^2 n)$ amortized	$O(\log n)$ worst case
Search I/Os		$O(\log_B n)$ worst case	$O(\log_B n)$ worst case
Update I/Os		$O(\log_B n)$ amortized under reasonable constraints	$O(\log_B)$ worst case
Space Utilization		tradeoff with update time	Minimum: 0.5 Expected: 0.7
Memory Allocation (heap accesses)		Rare: once a phase	Frequent: on every split
Large Reorganizations		Do occur	avoided completely

Table 1: Comparison between the B-table and the B^+tree

References

[1] R. Bayer and C. McCreight. Organization and maintenance of large ordered indexes. *Acta Informatica*, 1(3):173–189, 1972.

[2] Svante Carlsson, J. Ian Munro, and Patricio V. Poblete. An implicit binomial queue with constant insertion time. In *Proceedings of 1st Scandinavian Workshop on Algorithm Theory*, volume 318 of *Lecture Notes in Computer Science*, pages 1–13. Springer-Verlag, 1988.

[3] D. Comer. The ubiquitous B-tree. *ACM Computer Surveys*, 11(2):121–137, 1979.

[4] Eisenbarth, Ziviani, Gonnet, K. Mehlhorn, and D. Wood. The theory of fringe analysis and its applicaiton to 2-3 trees and b-trees. *Inf. and Control*, 55:125–174, 1982.

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

[6] 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.

[7] Irit Katriel. Implicit data structures based on local reorganizations. Master's thesis, Computer Science department, Technion - Israel Institute of Technology, 2001.

[8] J. I. Munro. An implicit data structure for the dictionary problem that runs in polylog time. In *25th annual Symposium on Foundations of Computer Science, October 24–26, 1984, Singer Island, Florida*, pages 369–374. IEEE Computer Society Press, 1984.

[9] J. Ian Munro. Developing implicit data structures. In *Mathematical Foundations of Computer Science 1986*, number 233 in Lecture Notes in Computer Science, pages 168–176. Springer-Verlag, 1986.

[10] 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, August 1986.

[11] J. Ian Munro and Patricio V. Poblete. Searchability in merging and implicit data structures. In Josep Díaz, editor, *Automata, Languages and Programming, 10th Colloquium*, volume 154 of *Lecture Notes in Computer Science*, pages 527–535, Barcelona, Spain, 18–22 July 1983. Springer-Verlag.

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

[13] Andrew Chi-Chih Yao. On random 2-3 trees. *Acta Informatica*, 9(2):159–170, April 1978.