Implicit Dictionaries Supporting Searches and Amortized Updates in $O(\log n \log\log n)$ Time

Gianni Franceschini*
francesc@di.unipi.it

Roberto Grossi*
grossi@di.unipi.it

Abstract

We describe a new implicit data structure for maintaining n data values in the first n locations of an array. No information other than n and the data is to be retained, and the only operations which we may perform on the data values (other than reads and writes) are comparisons. Our structure supports searches in $O(\log n \log\log n)$ time in the worst case, and insertions and deletions in $O(\log n \log\log n)$ amortized time. The best known bound for these operations in main memory is $O(\log^2 n/ \log\log n)$ in the worst case.

1 Introduction

We consider the problem of designing data structures for dynamic search that achieve the best space saving possible, that is, they occupy *no* memory cells besides those 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 consecutive memory cells (no empty cells can be interspersed), assuming that each key occupies a single cell. 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* [3], and Williams' heap [5] is the best known example.

The importance of this problem is associated with the intriguing question whether auxiliary information and additional space are strictly necessary to have a dictionary with $O(\log n)$ time performance. Yao [6] has shown that having one single cell besides the n cells for the keys (of a bounded universe!) can get constant-time search, while having only n cells gives an $\Omega(\log n)$ search bound. In addition, if the keys can only be read, written and compared, there is the known information-theoretical lower bound of $\Omega(\log n)$, independently of the space used. It is an interesting question, therefore, to devise an implicit data structure that takes $O(\log n)$ time, thus matching both lower bounds. The former lower bound is matched as no additional cell is used; and the latter is matched because of the comparison model.

In a previous work [1], an implicit data structure for two-level memory requires $O(\log_B n)$ block transfers per operation in the worst case like B-trees, assuming that the block transfer size is $B = \Omega(\log n)$. Using it in main memory with $B = \Theta(\log n)$, it provides a bound of $O(\log^2 n/ \log\log n)$ improving a longstanding bound of $O(\log^2 n)$ of Munro [3]. That reopened the issue of finding an implicit dictionary that takes $O(\log n)$ time.

In this paper, we provide a further step towards this direction, and describe an implicit data structure supporting searches in $O(\log n \log\log n)$ time in the worst case, and insertions and deletions in $O(\log n \log\log n)$ amortized time. A lesson learned from [1] is that of using a B-tree of large fan-out with height of $o(\log n)$ while allocating variable-size nodes in compact memory lists of fixed-size allocation units. We exploit the latter feature and introduce new techniques with the main purpose of getting faster amortized updates in main memory. When used in external memory, our solution does not improve the worst-case bounds in [1]. Nevertheless, our bounds are a significant improvement over the worst-case bounds in main memory of $O(\log^2 n)$ in [4] and of $O(\log^2 n/ \log\log n)$ in [1].

Finally, the additional temporary space required by the amortized updates of our data structure can be reduced to only $O(1)$ registers in the RAM. Note that, for general data structures, amortized updating may require the usage of additional memory, which we can avoid in our case.

1.1 Overview and roadmap. We assume without loss of generality that the number n of keys is upper bounded by N, such that $\log N = \Theta(\log n)$. This assumption is not a limitation, as crucially shown by Frederickson [2], but it is useful to fix the bit-length of the pointers. In fact, we can keep $O(\log\log n)$ distinct

*Dipartimento di Informatica, Università di Pisa, via Buonarroti 2, 56127 Pisa. Supported by the University of Pisa.

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. Strictly speaking, we do not need to store the value of n, but we can use a variable encoding (like δ-codes) with $O(\log n)$ bits implicitly represented with the first keys of the data structure. We adopt the basic technique in [4] to implicitly represent the structural information by encoding a pointer or an integer of $d = \log N$ bits by using $2d$ distinct keys $x_1, y_1, x_2, y_2, \ldots, x_d, y_d$ that are pairwise permuted with the following rule: if the ith bit is 0, then $\min\{x_i, y_i\}$ precedes $\max\{x_i, y_i\}$; else, the bit is 1 and $\max\{x_i, y_i\}$ precedes $\min\{x_i, y_i\}$.

At any time, the keys in our data structure are grouped into *chunk* of $k = \Theta(\log N) = O(\log n)$ keys each. Each chunk contains (pairwise permuted) keys from a certain interval of values, and chunks are pairwise disjoint as intervals. This allows us to define a total order on any set of the chunks.

Our data structure hinges on a known tree organization of height $O(\log \log n)$, defined by a recursive partition of keys into equal sets. Given n keys, we organize them into chunks as described above. We select $\Theta(\sqrt{n})$ chunks as in distribution sorting, and distribute the remaining chunks in buckets. The selected chunks are stored in the root, and the buckets are recursively stored in $\Theta(\sqrt{n})$ children of the root. The ith bucket is of size $\Theta(\sqrt{n})$ and is stored in the ith children of the root. We stop the recursion as soon as the bucket size is $O(1)$ chunks. These buckets are stored in the leaves of the tree, all having the same level (distance from the root, which has level 0). The level of the leaves is the height h of our tree.

Formally speaking, let b be any constant power of 2, and β_i denote $b^{2^{h-i}}$ for $0 \leq i \leq h$. Our tree satisfies the following properties:

1. The root contains r chunks such that $1 \leq r \leq 4\beta_0$.
2. An arbitrary node at level $i \geq 1$ containing r chunks satisfies $\beta_i \leq r \leq 4\beta_i$
3. An arbitrary leaf contains r chunks plus at most $k - 1$ spare keys, where $b \leq r \leq 4b$. Leaves are the only nodes whose number of keys can change by 1.
4. The height is $h \leq \alpha$, where α is the minimum positive integer such that $b^{2^\alpha} = \sqrt{N}$.

Looking at the properties 1–3, we can figure out the tree as a B-tree of doubly logarithmic height $h = O(\log \log n)$, whose nodes are of variable size depending on their level. The chunks are the basic items encoding the pointers to children in each node.

Updating relies on splitting, merging and sharing of nodes. However, as we shall see, restructuring the tree is a non-trivial task since the nodes changing size must be relocated in an expensive way. We opt for amortization in this case to distribute the costs during the lifetime of the data structure.

A crucial component of our solution is the memory layout of the nodes to support the amortized updates. In particular, we keep some invariants on the chunks (and their encoded pointers to the children) in the nodes so that, given an *internal* node u, we classify its chunks as being *actual* and *virtual* (leaves are only made of actual chunks). The actual chunks physically reside inside the memory area allocate to u, while the virtual chunks are interleaved with the latter chunks, but stored independently in another special area of the memory. From a logical point of view, node u is the merge of the two kinds of chunks, and so a simple variant of the binary search inside u is sufficient to route the keys to its children. We introduce virtual chunks to accumulate enough potential in the amortized insertions, so as to be able to absorb the cost of reallocating u to a larger memory area. We need therefore to store virtual chunks in a common area shared by all nodes so as not to change frequently the number of actual chunks during the insertions. We further classify actual chunks into *original* and *patch* to accumulate operations for the amortized deletions so that we postpone the reallocation of u in a smaller memory area. The invariants on all these chunks guarantee a correct balance, so that amortization can work in $O(\log n)$ time per level.

The paper is organized as follows. In Section 2, we first focus on an semi-implicit dictionary that supports only insertions and searches, and introduce the mathematical definition of actual and virtual chunks. In Section 3, we then consider the full case of an implicit dictionary supporting insertions, deletions and searches, introducing original and patch chunks and their invariant. In Section 4, we provide the amortized analysis of our data structure. In the following, we do not discuss in detail how to handle the root of the data structure, which is a bit more involved than the other nodes but does not provide a technical contribution to the paper.

2 A Semi-Dynamic Implicit Dictionary

In this section, we describe a first version of our implicit dictionary, which is able to support searches in $O(\log n \log \log n)$ time and insertions in $O(\log n \log \log n)$ amortized time. We will describe in Section 3 how to deal also with deletions.

2.1 Actual and virtual chunks. Chunks are at the heart of our data structure as they can implicitly encode bits of information by a suitable permutation of their keys. We need to classify chunks according to their physical allocation in memory. The chunks that are stored in the memory area allocated to a node v are called *actual*. We also employ chunks called *virtual* that are still associated with that node v, but are stored in a separate memory area (note that we do *not* need virtual chunks in the leaves). We denote the set of actual chunks by $act(v)$ and the set of virtual chunks by $vir(v)$. The former can be retrieved by accessing v. The latter is represented by a doubly linked list of virtual chunks, in which the pointer to the head of the list is encoded at the beginning of node v, and the pointers to the adjacent chunks of $c' \in vir(v)$ are encoded in c' itself. Actual chunks and virtual chunks are related, so that

1. each chunk $c \in act(v)$ has associated at most one chunk $c' \in vir(v)$, with a pointer to c' being encoded in c, and
2. if c' exists, it is the successor of c in node v (hence, c' cannot be the successor of another virtual chunk in v).

As a result, the chunks in $act(v)$ and $vir(v)$ are interleaved and encodes the pointers to the children of v (see Figure 1).

We preserve an invariant on the number of actual chunks and virtual chunks. Recall from Section 1.1 that the actual chunks satisfy the conditions $1 \leq |act(v)| \leq 4\beta_0$ for $i = 0$ (the root) and $\beta_i \leq |act(v)| \leq 4\beta_i$ otherwise. Moreover, our updating algorithms guarantee that $|act(v)|$ is always a *multiple* of $\sqrt{\beta_i}$. As for the virtual chunks, they satisfy $\sqrt{\beta_i} \leq |vir(v)| \leq 4\sqrt{\beta_i}$ for $i \geq 1$.

In the rest of the paper, given an internal node v, we define the *size* of v as the number $|act(v)|$ of its *actual* chunks (intuitively, changing their number causes a relocation).

2.2 Routing and searching keys. In order to efficiently route the keys during a traversal of the tree, we perform a binary search in each traversed node v. Specifically, we perform first the search on the set $act(v)$, which is available inside v. Once a chunk $c \in act(v)$ is identified by the search, we access its associated chunk $c' \in vir(v)$ (if any) in $O(\log n)$ time and check also c'. As one can see, there is no need to scan the list representing $vir(v)$. We anticipate that the layout of node v in memory can store $act(v)$ in $O(1)$ memory areas, but this is not much a problem as it does not affect the complexity.

LEMMA 2.1. *Routing a key in a node requires $O(\log n)$ time.*

Searching is standard, with the observation that we must pay $O(k) = O(\log n)$ time to decode a pointer from a node v to one of its children. The cost of searching is $O(\log n \log\log n)$ in total as the height is $h = O(\log\log n)$.

LEMMA 2.2. *Searching a key in the semi-dynamic implicit dictionary requires $O(\log n \log\log n)$ time.*

2.3 Layout in Memory. We now detail how the nodes and the virtual chunks are allocated in memory. We divide the memory in $O(h)$ zones:

- Zone P has the only purpose to store $O(\log\log n)$ integers and pointers of $O(\log n)$ bits each as bookkeeping information related to the several levels in the data structure.
- Zone Z_i storing the nodes at level i, for $0 \leq i \leq h$.
- Zone V storing all the virtual chunks at the several levels.

The zones in the memory layout appear in the order given by $P, Z_1, Z_2, \ldots, Z_{h-1}, V, Z_h, Z_0$.

Let's focus on any zone Z_i, for $1 \leq i \leq h-1$. It stores the internal nodes at level i in compacted lists. We encode the memory address of zone Z_i and the value of β_i in zone P. Knowing that the number of actual chunks in each node is a multiple of $\sqrt{\beta_i}$ ranging from β_i to $4\beta_i$, we can see that the size of a node can have at most $3\sqrt{\beta_i}+1$ different values at level i. We therefore keep so many lists for their storage in memory, where doubly linked list j ($0 \leq j \leq 3\sqrt{\beta_i}$) packs all the nodes having size $j\sqrt{\beta_i} + \beta_i$ into allocation units of β_i chunks each (a node can be stored in more than one allocation unit). The allocation units are all full, except the head allocation unit, which can be partially filled. In order to preserve the implicitness of the data structure, we divide Z_i in two parts, denoted A_i and H_i, so that A_i contains the allocation units that are not heads of lists and H_i contains their heads in a *compacted form* (concatenation of all the keys in the heads in a single sequence). Note that H_i contains at most $(3\sqrt{\beta_i}+1)\beta_i$ chunks. We refer the reader to the compactor lists described in [1] for further details, especially on how to encode the pointers of the lists inside their allocation units. We manage the heads in H_i differently from what done in [1], taking $O(\beta_i k + \sqrt{\beta_i} \log n \log\log n)$ time. Part H_i is a collection of $O(\sqrt{\beta_i})$ heads, each containing a multiple of $\sqrt{\beta_i}$ actual chunks. Each such head is kept fragmented into $O(\sqrt{\beta_i})$ pieces of $\sqrt{\beta_i}$ linked actual chunks, so that relocation takes $O(\beta_i k)$ time plus $O(\sqrt{\beta_i} \log n \log\log n)$ time for redirecting $O(\sqrt{\beta_i})$ pointers.

Next in the memory layout, we have zone V for the virtual chunks. This is a simple sequence of allocation

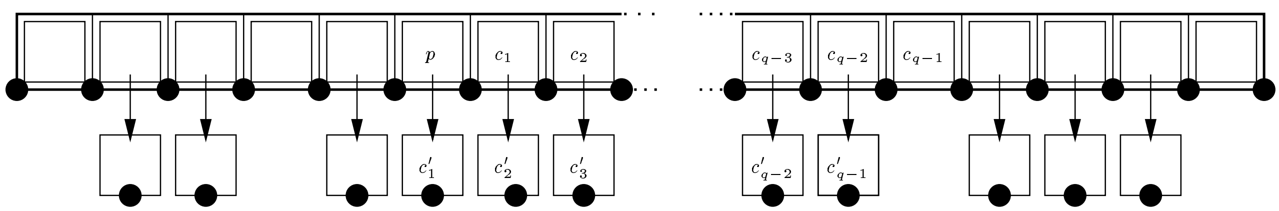

Figure 1: The set $act(v)$ of actual chunks (top row) and the set $vir(v)$ of virtual chunks (bottom row) in a node v, along with their encoded pointers (the bullets) to the children of v.

units, each containing exactly one chunk. Its starting position is encoded in zone P. Following V, we have zone Z_h for the leaves of the data structure. Here, the organization of part A_h is identical to that of the other parts A_i in the previous levels, except that the allocation unit of the lists is exactly one chunk of k keys and the number of these lists is $O(k)$ (since the number of keys of a leaf can increase or decrease by 1). As a result, H_h contains $O(k)$ heads, each with $O(k)$ keys. We store these $O(k^2)$ keys in H_h as smaller lists having smaller allocation unit of $\ell = \Theta(\log k)$, where k is a multiple of ℓ. Handling these smaller list is like in [1], and so we do not detail them any further. We need to encode $O(k)$ pointers of only $O(\log k)$ bits internally in part H_h. Finally, we have zone Z_0 that contains only the root. We handle the whole Z_0 (i.e., the root) with a cyclic representation, in which we break it into its single keys. We need $O(1)$ integers and pointers of $O(\log n)$ bits, which are encoded inside zone P for this purpose.

2.4 Updating the Layout. During the lifetime of the data structure, we need to keep zone P filled with $O(\log N \log\log N) = O(\log n \log\log n)$ arbitrary keys for bookkeeping. They are completely scanned each time at a cost of $O(\log n \log\log n)$ time. If one of them is deleted, we replace it with another key that is removed from the rest of the data structure. So, we can focus on how to update the remaining zones (described in Section 2.3) when a node change size.

As for zone Z_i, with $1 \leq i \leq h-1$, the reorganization comes in two fashions.

Sliding. We need to translate the whole zone Z_i by no more than $\beta_j k$ positions either to the left or to the right, provided that the newly occupied positions have been previously emptied, where $j < i$. This can be accomplished in $O(\sqrt{\beta_i}\beta_i k) = O(\beta_j k)$ time by rotating part A_i and sliding the whole H_i. We take the first $\sqrt{\beta_j}/\beta_i$ allocation units in A_i and append them to the end of A_i. We then completely slide by $\beta_j k$ positions the $O(\sqrt{\beta_i}\beta_i) = O(\beta_j)$ chunks in H_i. We have also to redirect the pointers to the β_j/β_i allocation units of A_i that are relocatred, in $O(\sqrt{\beta_j}/\beta_i \log n \log\log n)$ time.

Resizing. We need to move a node from a list to another list internally to Z_i because its size changes by $\sqrt{\beta_i}$ chunks. In this case, we need to exchange the locations of $O(1)$ allocation units in A_i so as to work on the heads of the involved lists. The rest or reorganization involves the heads in H_i and it costs $O(\beta_i k + \sqrt{\beta_i} \log n \log\log n)$, as previously mentioned.

Zone V is easy to handle as its allocation units are single chunks. In zone Z_h, we manage A_h analogously to the other A_i's, and H_h in $O(\log n \log\log n)$ time analogously to what done in [1]. Finally, the root in Z_0 is rotated one key at time, so its cost of relocation is directly proportional to the number of shifted positions.

2.5 Handling insertions. Inserting a new key x follows a downward traversal of the tree like in the search, eventually finding the chunk of a node u in which the key must be inserted. We insert x in the chunk and, if u is not a leaf, we remove the largest key in the chunk and insert it in the leftmost (or rightmost, depending on the case) leaf descending from the right pointer encoded in the chunk. As a result, we end up increasing the number of keys in a leaf by one. To make room for one more key in the leaf, which is allocated in zone Z_h (Section 2.3), we rotate the root in zone Z_0 by one position to the right. We also increase the number of keys of the leaf by 1, and therefore we need to exchange it with the node in the head of its list and move then to the head of the target list, updating the layout as described in Section 2.4. If the number of keys in the leaf is still less than $4bk + k$, we can conclude the operation at the cost of $O(\log n \log\log n)$ time. However, if the number of keys becomes $4bk + k$, we have to split the leaf and some of its ancestor nodes.

The process of restructuring the tree is quite expensive and a possible way to deal with it in the amortized sense is to delay the change of size of the nodes that are ancestors of the leaf. Virtual chunks are introduced for this purpose, and we accumulate enough of them before splitting an internal node (we recall that we do not need virtual chunks in the leaves). Before describing this process in some detail below, we need to define the following notion. We say that a given subset C of the chunks $act(v) \cup vir(v)$ associated with a node v is a *chain* starting at actual chunk c in node v if the following conditions hold:

1. C is the union of a set of q actual chunks $\{c, c_1, \ldots, c_{q-1}\} \subseteq act(v)$ and of a set of $q-1$ virtual chunks $\{c'_1, \ldots, c'_{q-1}\} \subseteq vir(v)$, for an integer $q > 0$;

2. In node v, chunk c'_1 is the successor of c and the predecessor of c_1; chunk c'_j is the successor of c_{j-1} and the predecessor of c_j, for $2 \leq j \leq q-1$.

3. Chunk c_{q-1} has no virtual chunk associated with it (i.e., its successor is another actual chunk, if it exists in v).

We refer to the number q of actual chunks in C as the length of C, denoted by $|C|$. An example of chain is illustrated in Figure 1.

We can now focus on the reorganization of an internal node v at level i, where $1 \leq i \leq h-1$, by exploiting the invariant that it satisfies on the number of virtual chunks (see Section 2.1). Let c be the chunk that must be inserted into v and has been originated from the split of its child u into two nodes u' and u''. We take three possible actions described next.

Increasing the number of virtual chunks. This case holds when the number of virtual chunks $|vir(v)| < 4\sqrt{\beta_i}$ is not maximal. We have room for at least one more virtual chunk, and we do not change the number $|act(v)|$ of actual chunks. We find the actual chunk p that is the predecessor of c in node v by a simple binary search (if p does not exist, we take the successor with minor modifications). We take the chain $C = pc'_1c_1 \ldots c'_{q-1}c_{q-1}$ starting at p (by definition of chain, c_{q-1} has no virtual chunk associated). We find the position of c in C (either before or after c'_1) and obtain a sorted sequence S, which is the sequence of chunks in C with c added in its suitable position. At this point, we create a new chain C' from S to replace C with C' in the memory layout. Namely, the chunks in odd positions become actual and those in even positions become virtual (see Figure 2 for an example when c precedes c'_1). The crux of this operation relies in the property that $|C| = |C'|$, that is, the number of actual chunks in chains C and C' does not change (although we change the sets $act(v)$ and $vir(v)$ and increase $|vir(v)|$ by 1). After this operation, we preserve the invariant on virtual chunks defined in Section 2.1. Since the total number of chunk involved in chains C and C' is at most twice the number of virtual chunks, we have:

LEMMA 2.3. *The cost for increasing the number of virtual chunks in a node at level i is $O(\sqrt{\beta_i}k)$ time.*

Increasing the number of actual chunks. This case applies when the number of virtual chunks is maximal $|vir(v)| = 4\sqrt{\beta_i}$, while the number $r \leq 4\beta_i - \sqrt{\beta_i}$ of actual chunks is not maximal in node v. In this case, we suspend the insertion of c in node v, and we increase the size of v by $\sqrt{\beta_i}$ actual chunks, which are chosen from the virtual chunks available in v. After restructuring, node v contains $r + \sqrt{\beta_i}$ actual chunks and $3\sqrt{\beta_i}$ virtual chunks. Now we can resume the insertion of c in v with the operation described in the previous paragraph, increasing the number of virtual chunks by 1.

We detail how to add $\sqrt{\beta_i}$ actual chunks to v. We first select $\sqrt{\beta_i}$ virtual chunks in v, which are all in zone V as described in Section 2.3. We then recompact zone V by filling the $\sqrt{\beta_i}$ slots left empty by the selected virtual chunks with other chunks in V (the first $\sqrt{\beta_i}$ remaining ones in it). As a result, we shrink zone V by $\sqrt{\beta_i}$ chunks to the left.

Now, zone Z_{h-1} preceding zone V has room for sliding $\sqrt{\beta_i}k$ positions to its right as described in Section 2.4. We apply an identical sliding to the other zones $Z_{h-2}, Z_{h-3}, \ldots, Z_{i+1}$ in this order. We end up with additional room to the right of zone Z_i, which has its part H_i enlarged to host the $\sqrt{\beta_i}$ selected chunks. They are turned into actual chunks in node v. We then reorganize part H_i as described in Section 2.4.

The relocation of $\sqrt{\beta_i}$ chunks in zone V requires to perform so many searches to locate the nodes having them associated as virtual chunks. We also need to perform further $O(1)$ searches to handle the $O(1)$ relocated nodes in zone Z_i.

In order to analyze the cost of this case, we first observe that recompacting zone V requires $O(\sqrt{\beta_i} \log n \log\log n)$ time as the cost is dominated by the searches (Lemma 2.2). The sliding of each zone Z_j, $j > i$, takes $O(\beta_i k + \sqrt{\beta_i}/\beta_j \log n \log\log n)$ time (see Section 2.4) and is repeated for at most $h = O(\log\log n)$ zones. The internal reorganization of zone Z_i takes $O(\beta_i k + \sqrt{\beta_i} \log n \log\log n)$ time. So the total cost is $O(\beta_i k + \sqrt{\beta_i} \log n (\log\log n)^2)$.

LEMMA 2.4. *The cost for increasing the number of actual chunks in a node at level i is $O(\beta_i k + \sqrt{\beta_i} \log n (\log\log n)^2)$ time.*

Splitting. The last case is when both the actual chunks and the virtual chunks are maximal in number (i.e., $|act(v)| = 4\beta_i$ and $|vir(v)| = 4\sqrt{\beta_i}$). We insert chunk c in node v, and we take the median chunk m in the set $act(v) \cup vir(v) \cup \{c\}$, and propagate it upward to

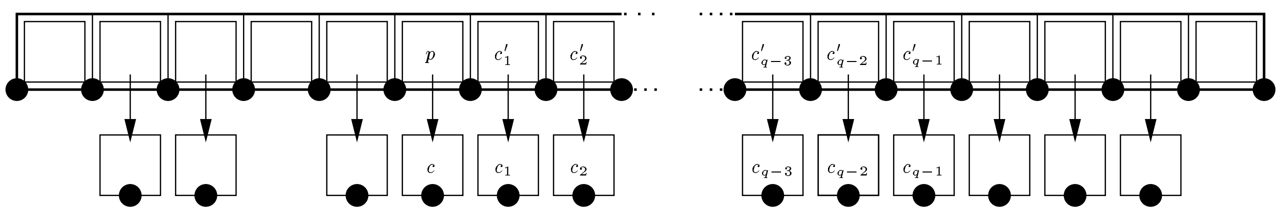

Figure 2: Processing a chain Figure 1.

v's parent. We then split v as two nodes v' and v'', each node with $2\beta_i$ actual chunks and $2\sqrt{\beta_i}$ virtual chunks. In general, the actual chunks and the virtual chunks are not so evenly partitioned. Without getting into detail, we redistribute them evenly by in-place merging the chunks in $act(v) \cup vir(v) \cup \{c\} - \{m\}$. The total cost is $O(\beta_i k)$ time.

LEMMA 2.5. *The cost of splitting a node at level i is $O(\beta_i k)$ time.*

3 Fully-Dynamic Implicit Dictionary for Handling Deletions

In this section, we describe how to deal with deletions in the data structure presented in Section 2. The virtual chunks can mostly help us in handing deletions. Unfortunately, this is not the case when merging two nodes with the chunk c removed from their parent, and c is actual with no associated virtual chunk. In this case, we cannot find easily a replacement for that chunk in the parent node, and we must completely reorganize it or change its size (we recall that the invariants are defined in terms of the size of the internal nodes, i.e., their number of actual chunks). This enforces an update of the layout, as described in Section 2.4, that we cannot amortize. We introduce patch chunks as replacements when this kind of situation arises.

3.1 Extension of the main structure. Given an internal node v, we divide its actual chunks in a set $seg(v)$ made up of *segments*, each of $\sqrt{\beta_i}$ adjacent chunks (recall that the size of v is multiple of $\sqrt{\beta_i}$, i.e., $|act(v)| = \sqrt{\beta_i}\,|seg(v)|$).

Each segment $z \in seg(v)$ is further divided in two parts, one storing the actual chunks called *original* and the other the actual chunks called *patch* (see Figure 3). We denote by $orig(v)$ the set of original chunks in all segments of v, and by $pat(v)$ the set of all patch chunks in those segments. Note that each of these sets is in order if we take the segments in a left-to-right scan of v. We keep the following invariant on each internal node v and each segment $z \in seg(v)$ to make binary search possible on the keys stored in v:

(I1) $|pat(v)| < |seg(v)| + \sqrt{\beta_i}$;

(I2) the first chunk of z is in $orig(v)$ and the last chunk of z is in $pat(v)$.

Note that each segment z is partitioned in two parts, with the left part storing only the original chunks and the right part storing only the patch chunks. Moreover, we encode in the first chunk of z the last position in z occupied by an original chunk.

Since we keep $orig(v)$ and $pat(v)$ sorted independently of each other, we associate virtual chunks with each of them separately. In particular, $orig(v)$ and its virtual chunks satisfy properties 1–2 of the invariant described in Section 2.1, as well as do $pat(v)$ and its virtual chunks. Note that $orig(v) \cup pat(v)$ and the union of their virtual chunks do not necessarily satisfy that invariant. When needed, we can run an in-place merge and a redistribution of the chunks to satisfy the invariant globally in the node v at level i, at a cost of $O(\beta_i k)$ time. We can also define a notion of chain analogous to that given in Section 2.5, applied to either $orig(v)$ and its virtual chunks or to $pat(v)$ and its virtual chunks.

In order to route keys in the new organization of node v, we perform first a binary search on $orig(v)$ and its virtual chunks, and then another binary search on $pat(v)$ and its virtual chunks. As chunks are disjoint, either we end up inside a chunk and we stop the search, or we have to access one child of v. That child can be identified by comparing the outcome of the two binary searches above. The total cost of routing stated in Lemma 2.1 remains unchanged. We therefore have:

LEMMA 3.1. *Searching a key in the implicit dictionary takes $O(\log n \log\log n)$ worst-case time.*

The algorithm for inserting a key in the new organization does not change much. We end up in a leaf as described in Section 2.5. At this point, if the leaf splits, we still have three possible actions on its ancestor nodes that split as well (see Section 2.5). (a) If the total number of virtual chunks is not maximal, we find a

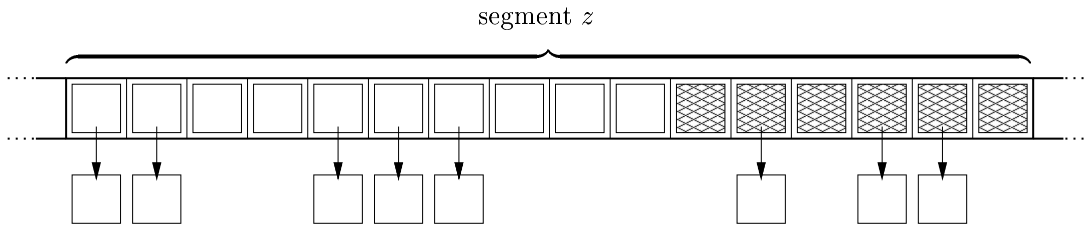

Figure 3: A partition of the actual chunks in segment z into original chunks (left) and patch chunks (right), shown in the top row. Their associated virtual chunks are shown in the bottom row.

chain either in $orig(v)$ or in $pat(v)$, and proceed analogously by adding one more virtual chunk. (b) If the total number of virtual chunks is maximal but the number of actual chunks is not maximal, we proceed as before by adding $\sqrt{\beta_i}$ virtual chunks. (c) Splitting does not vary. In addition to cases (b) and (c), we turn all the actual chunks in the node into original and distribute them in the segments of the node, except the last chunk in each segment (which remain patch to reflect the invariant in Section 3.1).

It is worth noting that Lemma 2.3, 2.4, and 2.5 still hold in the new organization of our data structure.

3.2 Handling deletions. Deleting a key x follows a downward traversal of the tree like in the search, eventually finding the chunk of a node u in which the key must be deleted. We remove x from the chunk and, if u is not a leaf, we insert in the chunk the smallest key in the leftmost (or rightmost, depending on the case) leaf descending from the right pointer encoded in the chunk (and delete that key from the leaf). As a result, we decrease the number of keys in a leaf by one. As the leaf is allocated in zone Z_h (Section 2.3), we rotate the root in zone Z_0 by one position to the left. We need to exchange the leaf with the node in the head of its list and move then to the head of the target list, updating the layout as described in Section 2.4. If the number of keys in the leaf is at least bk, we can conclude the operation at the cost of $O(\log n \log\log n)$ time. However, if the number of keys becomes strictly less than bk, we have to merge the leaf with a sibling having bk keys or share it with a sibling having at least $(b+1)k$ keys. We are in the situation in which a chunk c of an internal node v must be moved to two children u' and u'' so as to form a single child u of v. We take four possible actions to remove c from v, as described below.

Decreasing the number of virtual chunks. This case holds when the number of virtual chunks $|vir(v)| > \sqrt{\beta_i}$ is not minimal. Hence, we can remove virtual chunks without changing the number $|act(v)|$ of actual chunks. If c is virtual, we can safely remove it from v. If c is actual (original or patch) and has a virtual chunk c' associated, we can remove c and replace it with c', which becomes actual in v. However, we have a problem when c is actual but has no virtual chunk associated. We have two cases.

1. If c is patch, we remove it and replace with any virtual chunk c' of v. Then, we insert c' in its ordered position in $pat(v)$ and reorganize the whole set $pat(v)$. Since it contains at most $|pat(v)| < |seg(v)| + \sqrt{\beta_i} = O(\sqrt{\beta_i})$ chunks, we can (in-place) redistribute them in the segments of v to account for the removal of c and the insertion of c' in the set. The operation terminates with a cost of $O(\sqrt{\beta_i}\,k)$ time.

2. If c is original, we take the segment z containing c, and slide c to the end of the original chunks in z, so that z contains first all the original except c, then c, and finally all the patch chunks. At this point, we remove c from z and replace it with any virtual chunk c' of v. We then turn c' into patch, and reorganize the whole set $pat(v)$ as described in case 1. The cost is $O(\sqrt{\beta_i}\,k)$ time.

Note that, in case 2, we increase the number of patch chunks in v by 1. If invariant (I1) is violated in this way, we reorganize the whole node v, merging (in-place) the original and the patch chunks and their virtual chunks, and redistributing the chunks in the segments of v, so that we preserve the number of virtual chunks and actual chunks, and we only leave one patch chunk in the last position of each segment to maintain the invariants (I1) and (I2) in Section 3.1. This reorganization takes $O(\beta_i k)$ time.

LEMMA 3.2. *The cost for decreasing the number of virtual chunks in a node at level i is $O(\sqrt{\beta_i}\,k)$ time. If invariant (I1) is violated, the cost is $O(\beta_i k)$ time.*

Decreasing the number of actual chunks. This case applies when the number of virtual chunks is minimal $|vir(v)| = \sqrt{\beta_i}$, while the number $r \geq \beta_i + \sqrt{\beta_i}$

of actual chunks is not minimal in node v. In this case, we suspend the deletion of c in node v, and we decrease the size of v by $\sqrt{\beta_i}$ actual chunks, which become virtual chunks in v. After restructuring, node v contains $r - \sqrt{\beta_i}$ actual chunks and $2\sqrt{\beta_i}$ virtual chunks. Now we can resume the deletion of c in v with the operation described in the first case (decreasing the number of virtual chunks), decreasing the number of virtual chunks by 1. Making $\sqrt{\beta_i}$ actual chunks virtual, requires a reorganization of the memory layout symmetrical to what discussed in the second case of Section 2.5.

LEMMA 3.3. *The cost for decreasing the number of actual chunks in a node at level i is $O(\beta_i k + \sqrt{\beta_i} \log n (\log\log n)^2)$ time.*

Merging. This case holds when both the actual chunks and the virtual chunks are minimal in number (i.e., $|act(v)| = \beta_i$ and $|vir(v)| = \sqrt{\beta_i}$), and there is a sibling v' of v that satisfies the same conditions on its chunks. Let c' be the chunk in the common parent that separates v from v'. We recursively remove c' from the parent, and merge (in-place) c' and all the chunks (actual and virtual) in v' and in v, except c. We create a single node from v and v', and redistribute the sorted chunks in the segments of the new node, so that we assign $2\sqrt{\beta_i}$ virtual chunks and $2\beta_i$ actual chunks, leaving one patch chunk in the last position of each segment to maintain the invariants (I1) and (I2) in Section 3.1.

LEMMA 3.4. *The cost of merging two sibling nodes at level i is $O(\beta_i k)$ time.*

Sharing. The last case applies whenever none of the previous three cases holds. Namely, both the actual chunks and the virtual chunks in v are minimal in number (i.e., $|act(v)| = \beta_i$ and $|vir(v)| = \sqrt{\beta_i}$), but any of its sibling v' (say to the right) has more than $\beta_i + \sqrt{\beta_i}$ chunks. Let $\hat{c}$ be the chunk in the common parent that separates v from v'. We remove c from v performing the same operations described in the first case (decreasing the number of virtual chunks). After that, v has $\sqrt{\beta_i} - 1$ virtual chunks. We then identify the minimal chunk c' (actual or virtual) in v', and remove it from v' according to one of the first two cases previously discussed (decreasing the number of virtual chunks, or decreasing the number of actual chunks). We replace $\hat{c}$ with c' in the common parent of v and v' (note that it is a mere substitution, no need to reorganize the parent). We then insert $\hat{c}$ in v as described in the first case (increasing the number of virtual chunks) of Section 2.5. As a result, v' has one less chunk, the common parent of v and v' does not change the number of its actual (original and patch) chunks and their virtual chunks, satisfying the invariants (I1) and (I2) of Section 3.1. Finally, v has $\sqrt{\beta_i}$ virtual chunks and β_i actual chunks. The cost of sharing depends on how c' is removed from v':

LEMMA 3.5. *Sharing two sibling nodes at level i has the same time complexity of either decreasing the number of virtual chunks (Lemma 3.2) or decreasing the number of actual chunks (Lemma 3.3).*

Being more explicit, the time bound implied by Lemma 3.5 is

1. $O(\sqrt{\beta_i} k)$ when v borrows a chunk from v' by decreasing the number of virtual chunks in v' under invariant (I1);
2. $O(\beta_i k)$ when v borrows a chunk from v' by decreasing the number of virtual chunks in v' under invariant (I2);
3. $O(\beta_i k + \sqrt{\beta_i} \log n (\log\log n)^2)$ when v borrows a chunk from v' by decreasing the number of actual chunks in v'.

4 Amortized Analysis

In this section, we provide a sketch of the amortized analysis of the update operations supported by our data structure T. We use a potential function $\Phi(T) = \sum_{v \in T} \Phi(v)$ defined as the sum of the potential on each node v:

$$\Phi(v) = \begin{cases} \mathscr{A}_i(v) + \mathscr{V}_i(v) + \mathscr{P}_i(v) & v \text{ node at level } i \geq 1 \\ \mathscr{L}(v) & v \text{ leaf} \end{cases}$$

Let $\gamma = k \log n \log\log n$ be the "potential" of a chunk of k keys. In the formula for $\Phi(v)$, we denote by $\mathscr{A}_i(v)$ the potential provided by the actual chunks, having value $c_1 \beta_i \gamma$ for a constant c_1 when the actual chunks are minimal or maximal, and having value 0 when they are half on the way ($|act(v)| = 2\beta_i$):

$$\mathscr{A}_i(v) = \begin{cases} c_1(-|act(v)| + 2\beta_i)\gamma & |act(v)| \leq 2\beta_i \\ c_1(\frac{1}{2}|act(v)| - \beta_i)\gamma & |act(v)| > 2\beta_i \end{cases}$$

We denote by $\mathscr{V}_i(v)$ the potential provided by the virtual chunks, having value $c_2 \beta_i \gamma$ for a constant c_2 when the virtual chunks are minimal or maximal, and having value 0 when they are half on the way ($|vir(v)| = 2\sqrt{\beta_i}$):

$$\mathscr{V}_i(v) = \begin{cases} c_2(-\sqrt{\beta_i}|vir(v)| + 2\beta_i)\gamma & |vir(v)| \leq 2\sqrt{\beta_i} \\ c_2(\frac{1}{2}\sqrt{\beta_i}|vir(v)| - \beta_i)\gamma & |vir(v)| > 2\sqrt{\beta_i} \end{cases}$$

We denote by $\mathscr{P}_i(v)$ the potential for the patch chunks having value 0 when they are minimal ($|pat(v)| =$

$|seg(v)|)$ and value $c_3\beta_i\gamma$ for a constant c_3 when they are maximal:

$$\mathscr{P}_i(v) = c_3\sqrt{\beta_i}\gamma\left(|pat(v)| - |seg(v)|\right)$$

Finally, we have the potential $\mathscr{L}(v)$ for each leaf v, having value $c_4\gamma$ for a constant c_4 when the number of keys are minimal or maximal, and having value 0 when they are half on the way (i.e., $2bk$ keys):

$$\mathscr{L}(v) = \begin{cases} c_4(-\frac{1}{bk}|v| + 2)\gamma & |v| \le 2bk \\ c_4\left(\frac{1}{2bk+k-1}|v| - \frac{2bk}{2bk+k-1}\right)\gamma & |v| > 2bk \end{cases}$$

THEOREM 4.1. *Searching a key in the implicit data structure requires $O(\log n \log\log n)$ time in the worst case. Inserting or deleting a keys takes $O(\log n \log\log n)$ amortized time.*

Proof. Let us consider an arbitrary sequence of n insert and delete operations. Taking the first $s \le n$ operations into account, we can observe that $\Phi(T_s) \ge 0$ by definition of Φ, where T_s denotes the data structure resulting from applying the s operations to the initial empty data structure T_0. What we have to prove is that the amortized cost of the sth operation is $\hat{t}_s = t_s + \Phi(T_s) - \Phi(T_{s-1}) = O(\log n \log\log n)$, where t_s denote the actual cost of the sth operation.

We do not discuss all possible cases here, but we choose one of them and present its analysis. (The rest of the cases can be treated analogously.) Suppose that the sth operation inserts a key into T_{s-1} causing a sequence of splits from a leaf l to a child of node v (say, at level i), where we stop splitting as we can increase the number of actual chunks of v (see Section 2.5). In the following, we denote by v_j, $i < j < h$, the intermediate nodes along the path from v to l, where each v_j is split into v'_j e v''_j.

In order to evaluate the difference in potential, $\Phi(T_s) - \Phi(T_{s-1})$, produced by the sth operation in this case, we observe that the leaf l, with maximal number of keys, is split into l' and l'', each with $2bk$ keys. This gives a *negative* contribution of $\Phi(l) - (\Phi(l') + \Phi(l'')) = c_4\gamma = O(\log^2 n \log\log n)$, since $\Phi(l') = \Phi(l'') = 0$.

For the intermediate nodes v_j, we have an analogous situation, where $|act(v_j)| = 4\beta_j$ and $|vir(v_j)| = 4\sqrt{\beta_j}$, giving a *negative* contribution of $\Phi(v) - (\Phi(v'_j) + \Phi(v''_j)) = (c_1 + c_2)\beta_j\gamma + \mathscr{P}_j(v_j)$, where $\Phi(v'_j) = \Phi(v''_j) = 0$ and $\mathscr{P}_j(v_j) \ge 0$.

Finally, for node v in T_{s-1}, we have $|vir(v)| = 4\sqrt{\beta_i}$ and $|act(v)| = r \le 4\beta_i - \sqrt{\beta_i}$ since we can increase the number of actual chunks by hypothesis. Assuming without loss of generality that $r \ge 2\beta_i$, it follows that $\Phi(v) = c_1(\frac{1}{2}r - \beta_i)\gamma + c_2\beta_i\gamma + \mathscr{P}_i(v)$. After the sth operation, v is modified and we denote the resulting node by v' in T_s. Note that $|act(v')| = r + \sqrt{\beta_i}$ e $|vir(v')| = 3\sqrt{\beta_i}$. Hence, $\Phi(v') = c_1(\frac{1}{2}(r + \sqrt{\beta_i}) - \beta_i)\gamma + c_2(\frac{3}{2}\beta_i - \beta_i)\gamma - \mathscr{P}_i(v')$. Since $\mathscr{P}_i(v) = \mathscr{P}_i(v')$, we get a *negative* contribution of

$$\Phi(v) - \Phi(v') = \frac{1}{2}\gamma(c_2\beta_i - c_1\sqrt{\beta_i}) = \Omega(\beta_i \log^2 n \log\log n).$$

The changes in the potential described above are the only ones performed when transforming T_{s-1} into T_s. We can bound the total change in terms of the sum of the negative contribution by $\Phi(T_{s-1}) - \Phi(T_s) > \Phi(v) - \Phi(v')$. It remains to evaluate t_s by summing the costs in Lemma 2.2, Lemma 2.4 and Lemma 2.5, obtaining

$$t_s = O(\beta_i kn + \sqrt{\beta_i}\log n(\log\log n)^2),$$

so that we can conclude that

$$\begin{aligned} \hat{t}_s &= t_s - (\Phi(T_{s-1}) - \Phi(T_s)) \\ &< t_s - (\Phi(v) - \Phi(v')) \\ &= O(\log n \log\log n). \end{aligned}$$

by choosing suitably c_1 and c_2.

References

[1] 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)*, to appear, 2002.

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

[3] 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.

[4] 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.

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

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