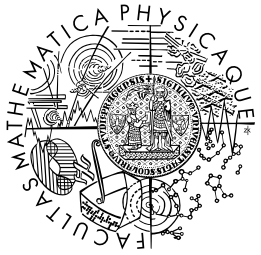

FACULTY
OF MATHEMATICS
AND PHYSICS
Charles University

BACHELOR THESIS

Matej Lieskovský

An implicit representation of sets

Department of Applied Mathematics

Supervisor of the bachelor thesis:	Mgr. Martin Mareš, Ph. D.
Study programme:	Computer Science
Study branch:	General Computer Science

Prague 2018

I declare that I carried out this bachelor thesis independently, and only with the cited sources, literature and other professional sources.

I understand that my work relates to the rights and obligations under the Act No. 121/2000 Sb., the Copyright Act, as amended, in particular the fact that the Charles University has the right to conclude a license agreement on the use of this work as a school work pursuant to Section 60 subsection 1 of the Copyright Act.

In ..... date ..... signature of the author

To my family and friends, for all their support.

Title: An implicit representation of sets

Author: Matej Lieskovský

Department: Department of Applied Mathematics

Supervisor: Mgr. Martin Mareš, Ph. D., Department of Applied Mathematics

Abstract: The 2003 paper by Gianni Franceschini and Roberto Grossi titled "Optimal Worst-Case Operations for Implicit Cache-Oblivious Search Trees" suggests a data structure that supports Insert, Find and Delete operations in $\mathcal{O}(\log n)$ worst-case time while also being implicit and cache-oblivious. We explain the general idea of the original data structure, identify flaws and gaps in its description, and describe a reimagined version of one of the two major components of the data structure.

Keywords: optimal worst-case implicit cache-oblivious search tree

Contents

Introduction

The original paper by Franceschini and Grossi (2003) outlines a data structure that supports standard ordered dictionary operations[1] in asymptotically optimal $\mathcal{O}(\log n)$ time per operation, and that is simultaneously both cache-oblivious[2] and implicit.[3] (See following sections for a formal definition of these properties.)

When studying the paper, we found several gaps and minor flaws in it. This thesis atempts to explain the data structure, mainly focusing on the top layer, where we try to fill all the gaps, fix all the flaws, and also give some insight into the design. While the bottom layer of the data structure is left unrepaired despite us relying on it having some rather non-trivial properties, it shall be the subject of a subsequent paper that will delve into the details of its internal workings.

In the first chapter of this thesis we shall introduce the cache-oblivious B-trees by Bender, Demaine, and Farach-Colton (2000). The second chapter then outlines the modifications necessary for us to turn the top layer of cache-oblivious B-trees into the top layer of our implicit structure, which works in $\mathcal{O}(\log n)$ amortized time. The third chapter shall then delve into the details of how our reimagined version of the top layer can be implemented. The fourth chapter shall then demonstrate that it is indeed possible to deamortize this data structure, provided that we can construct a powerful enough lower layer for it. The fifth and final chapter collects the changes we made to the structure and its presentation when compared to F&G. This is followed by the conclusion, the list of bibliography, and the glossary.

Please keep in mind that our results are currently mainly theoretical in nature. The focus is on making this structure easy to understand, so we shall not concern ourselves with achieving practical multiplicative constants. Furthermore, while it will be trivial to see that $\mathcal{O}(\log n)$ additional *operation memory* will be sufficient for any temporary storage needed by the operations, it is possible to reduce this to $\mathcal{O}(1)$, but this is left to the reader for the sake of conciseness.

0.1 The problem at hand

The dynamic ordered dictionary problem is the task of designing a data structure capable of maintaining a fully dynamic set of *keys* under insertion and deletion while also storing a *value* for each key in the set. We'll call a $(key, value)$ pair an *entry*.

Keys are taken from a universum with a defined total order and at any moment all stored keys are pairwise distinct. For convenience's sake, we make the common assumptions that we are given a key comparison function running in $\mathcal{O}(1)$ time and every entry fits into a single cell of memory.

Since our data structure always stores and manipulates entire entries (with the only exception being operation parameters), we can define a total order on the set of entries by looking at the total order on their keys.

[1] Insert, Find, Delete, Min, Max, Pred and Succ

[2] Asymptotically optimal for any cache of unknown parameters.

[3] Having $\mathcal{O}(1)$ overhead in memory consumption when compared to an array.

The supported operations shall enable us to do the following:

- Insert and Delete entries
- Find items by their key
- Update the value stored in a given entry
- Find the minimum and maximum keys in the data structure
- For any given element of the universum from which the keys are taken, return the nearest larger and smaller key occurring in the date structure

0.2 Succinct & implicit data structures

Succinct data structures are data structures that attempt to limit their overhead memory usage. With some data structures, this is meant as an effort to stay "close" to the information-theoretic bound on bits needed. However, in the case of container-type data structures a more common concern is the overhead when compared with an array. In either case, implicit data structures are those that require $\mathcal{O}(1)$ overhead.

We shall use a relatively strict and standard version of the implicit memory model, wherein we are only provided with an array of n cells to store the entries in and a single integer n informing us of the number of entries currently stored. The provided array is presumed to extend itself by one cell before every insertion and to shrink by one cell after every deletion.

Note that this is not optimal memory usage from the information-theoretic point of view as we can encode $\Theta(n \log n)$ bits of auxillary data in the ordering of the entries.

For a rather well known implicit data structure look at the implicit binary heap used for in-place heapsort by Williams (1964) and Floyd (1964).

0.3 Cache-oblivious data structures

Traditional algorithms assume "flat memory" — that accessing any cell of memory takes the same amount of time — but modern computers face an ever-growing difference between the speed of the CPU and the speed of memory. Accessing RAM is about three orders of magnitude slower than a CPU register, while HDD seek times can be four orders of magnitude slower than that. This gap resulted in the study of external-memory algorithms.

A formalisation of the behaviour of an algorithm whose data is mostly stored on the harddrive and which is limited by the small size of available main memory is the *external memory model*. The harddrive is partitioned into a large number of blocks of size B, and the algorithm must request the loading of individual blocks into main memory explicitly. The goal is to minimize the number of block transfers needed. B-trees by Bayer and McCreight (1972) are a classical example of an external-memory data structure.

As memory size grew, the external memory model became less of an acute problem, but the difference between the speed of the CPU and the speed of main memory caused the invention of the cache.

In the case of only a single level of cache being used, the cache provides a service for the main memory that is not dissimilar to the service the main memory provides to the harddrive. The main difference (apart from scale) is that cache handles the loading of blocks automatically. The problem of deciding which block to evict when cache is full is well-studied and deserving of a separate text. For our purposes, we can afford to assume that the cache behaves optimally, to the point that it can predict when a block will be needed again.[4]

The performance of algorithms while in the presence of cache is analysed by the *cache-aware model*. In it, algorithms are aware of the cache's parameters and are built to take advantage of that. As in the external memory model, the goal is to minimize the number of block transfers between the practically infinite but slow external memory and the fast but very much finite cache. B-trees can also serve as an example of a cache-aware algorithm, but due to the different parameters, optimizing for both external-memory and cache — or for several levels of cache at once — requires more complex algorithms with more parameters.

Current hardware can have up to four levels of CPU cache followed by main memory, harddrives (with their own caches) and network storage. Designing cache-aware algorithms capable of fully utilising such complex memory hierarchies is far too complicated.

This is why various *cache-oblivious* algorithms are being studied, as those strive to minimize the number of block transfers for any size of block and cache without knowing those values. By proving that cache-oblivious algorithms work well for any unknown cache parameters we can then show that they work well for every level of a memory hierarchy simultaneously. Cache oblivious algorithms also handle changing cache size reasonably well.[5]

0.4 Search trees and hash tables

The dictionary problem is often solved using hash tables due to their exceptional average case performance. Unfortunately, hash tables cannot be made implicit, do not support ordered dictionaries, and have a bad worst-case complexity.

Both the ordered and the unordered dictionary problem can be solved using various kinds of search trees. For us, the most interesting of those is the B-tree, which can optimize for a single level of cache by setting an appropriate block size.

Another useful data structure that we shall later explore in greater detail is the *van Emde Boas permutation* by Frigo et al. (1999), which is an implicit and cache-oblivious way to encode a complete binary tree, but it is not dynamic.

As we have already mentioned before, the cache-oblivious B-tree, which makes the vEB permutation dynamic at the cost of implicitness, will also be described during Chapter 1.

[4] While this is not an implementable algorithm, there exist $\mathcal{O}(1)$-competitive online algorithms provided that they are given a cache of twice the size.

[5] When the CPU starts multi-tasking, the various processes affect each other's cache, effectively reducing its size.

0.5 Brief history

The credit for inspiring the search for an implicit search tree goes to Williams (1964) and Floyd (1964) for their implicit binary heap. Out of the implicit data structures known at the time, the unsorted array requires $\mathcal{O}(n)$ time for searching, while a sorted array can take $\mathcal{O}(n)$ time for insertion and deletion.

Munro and Suwanda (1980) were the first to explicitly introduce the notion of implicit data structures. They proved a lower bound of $\Omega(\sqrt{n})$ per operation for any implicit data structure with a predetermined partial order of elements, such as the heap and the sorted array. Their biparental heap matched that lower bound, requiring $\mathcal{O}(\sqrt{n})$ time per operation, while another data structure demonstrated that this bound could be broken by not having a fixed partial order.

Out of the many following papers, we point out the result by Frederickson (1983), which achieved $o(n^\epsilon)$ time for any fixed $\epsilon > 0$.

Implicit AVL trees by Munro (1986) achieved $\mathcal{O}\left(\log^2 n\right)$ time per operation and were long conjectured to be optimal until the discovery of implicit B-trees by Franceschini et al. (2002) which achieved $\mathcal{O}\left(\frac{\log^2 n}{\log \log n}\right)$.

This leads us nicely to the paper by Franceschini and Grossi (2003) (F&G), which claimed to describe an implicit cache-oblivious ordered dictionary that would require only $\mathcal{O}(\log n)$ time and $\mathcal{O}(\log_B n + 1)$ block transfers per operation. This is the paper that we shall dissect here.

0.6 Conventions

Throughout this thesis, we shall follow a few conventions for clarity.

Terms are emphasized the first time they are used, and you can find a glossary of them at the end of the thesis.

All logarithms are binary unless specified otherwise.

We take shall take care to divide the constriucts within the data structure into *physical* and *logical entities*. A physical entity occupies a single range of memory, using all the cells in a closed interval of adresses. This is in contrast to logical entities, which are not similarily constrained.

In all illustrations, black elements describe the physical location in memory, red explains the logical structure, while blue shows the relative values of keys.

In analysis of cache-oblivious algorithms a "tall cache" is commonly assumed, meaning that the number of blocks that can be stored in cache at once is $\Omega(B)$. For our purposes, the ability to store at least two blocks will suffice.

1. Cache-Oblivious B-tree

Let us begin our journey to an implicit cache-oblivious dictionary by introducing a non-implicit data structure that we shall later modify to achieve implicitness. This preliminary data structure is the cache-oblivious B-tree by Bender, Demaine, and Farach-Colton (2000).

Ordinary B-trees require explicit beforehand knowledge of block size B. While cache-oblivious B-trees work as a direct cache-oblivious replacement for B-trees, the internal workings of the two data structures have little in common.

1.1 Static case

In case that the set of entries does not change, the *van Emde Boas permutation* allows for $\mathcal{O}(\log n)$-time search, while also requiring only $\mathcal{O}(\log_B n + 1)$ block transfers for any (unknown) block size B. The vEB permutation of a binary search tree is defined recursively: Given a complete binary search tree of height $h = 2^k$, we output the vEB permutation of the top $h/2$ levels, followed by the sequence of vEB permutations of the subtrees of $h/2$ lower levels.

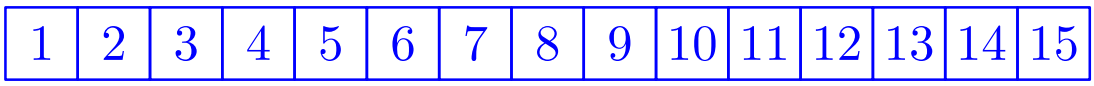

Figure 1.1: The entries viewed from the perspective of key order.

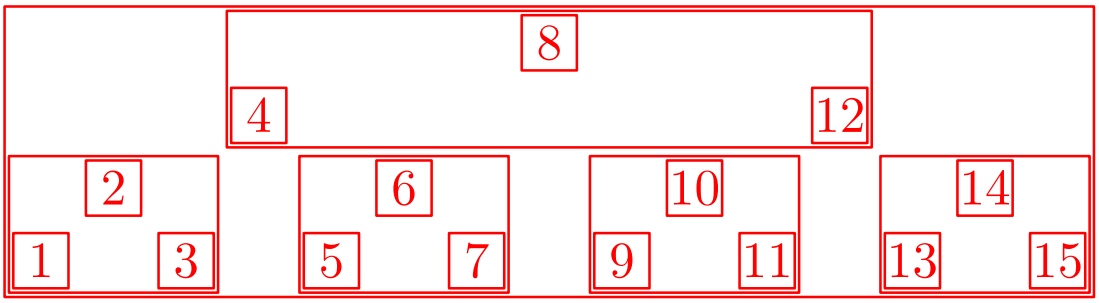

Figure 1.2: The logical structure of the vEB-permutation.

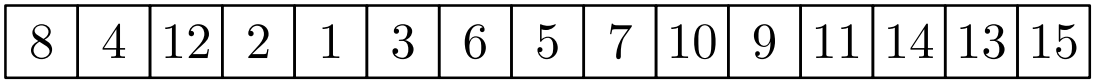

Figure 1.3: The resulting order of the entries in memory.

Note that for any block size B not larger than n there is some level of recursion during the construction of the vEB permutation where the size of the subtrees that are being encoded is between $\sqrt{B}$ and B. Every such encoded subtree therefore requires the transfer of at most two blocks and has a branching factor of $\Omega\left(\sqrt{B}\right)$. This means that the transfer of no more than 4 blocks is sufficient

for a branching factor of B to be achieved. Thus the vEB permutation achieves the goal of requiring $\mathcal{O}(\log_B n + 1)$ block transfers for a tree of height 2^k.

Let us now observe that $\Omega(\log_B n + 1)$ block transfers are indeed needed as $\Omega(\log n)$ bits of information are needed to identify the position of an entry, while only $\Theta(\log B)$ bits can be gained per block transfer (the rank of the entry relative to the contents of the block) and at least one block transfer is needed regardless of the block size.

In case the height of the tree to be encoded is odd, the top $\lceil h/2 \rceil$ levels shall be encoded first, followed by the subtrees of height $\lfloor h/2 \rfloor$. Two block transfers are still guaranteed to read a subtree of size at least $\frac{1}{2}\sqrt{B}$, worsening the bound on total block transfers by at most a multiplicative constant.

1.2 Ordered File Maintenance

To build a dynamic structure matching the performance of the vEB permutation, we shall need an *ordered file maintenance* algorithm. Ordered file maintenance is — together with list labeling and order maintenance — one of three closely related problems where a total order is maintained over a set of elements while supporting precedence queries, insertions and deletions.

Order maintenance is the task of designing a data structure that can support the following operations:

- Order(x,y): Return true if x is before y in the total order.
- InsertAfter(x,y): Insert y into the set and into the total order immediately after x. It is assumed that y is not in the set already.
- Delete(x): Delete x from the set and the total order.

A common special case of order maintenance is list labeling, in which integer labels of polynomial size are assigned to the elements in such a way that the total order of the used labels matches the order that is to be maintained, with elements being relabeled as needed.

A further restriction of list labeling is ordered file maintenance. In OFM, only labels of size $\mathcal{O}(n)$ can be used, emulating the problem of storing an ordered dynamic file with gaps for insertion. Elements are relabeled (moved) as needed by the insert and delete operations.

The order maintenance problem in general was first studied by Dietz (1982) and an $\mathcal{O}(1)$ worst-case solution was first published by Dietz and Sleator (1987). This relies on a solution to OFM in $\mathcal{O}\left(\log^2 n\right)$ worst-case time by Willard (1992). Willard's algorithm is also used by F&G, but we prefer the newer and simpler solution by Bender, Cole, Demaine, Farach-Colton, and Zito (2002). For a basic insight into the OFM problem, we shall now devote the remainder of this section to a short description of the much simpler $\mathcal{O}\left(\log^2 n\right)$ amortized time solution by Itai, Konheim, and Rodeh (1981). Note that we further simplify the algorithm by sacrificing the (for us unnecessary) ability to scan across elements in linear time, which would place a bound on the gaps between elements.

1.2.1 Amortized OFM algorithm

Suppose that we are given an array of some 2^{i+1} cells and given the task to store a dynamic set of up to $N = 2^i$ elements in the arbitrary order that will be specified by the insertion operations.

Firstly, we shall divide the entire array into 2^j *pages*, each of size $k \in \Theta(i)$. This makes them small enough for us to maintain them by complete rewrites.

Secondly, imagine a complete binary tree of height $h = j - 1$, with leaves corresponding to pages. Internal nodes will now correspond to intervals of pages. The interval corresponding to a node at level l has a capacity of $k \cdot 2^{h-l}$, meaning that it can store up to that many elements. Note that both tree height and node level are indexed from zero, with level 0 being the root.

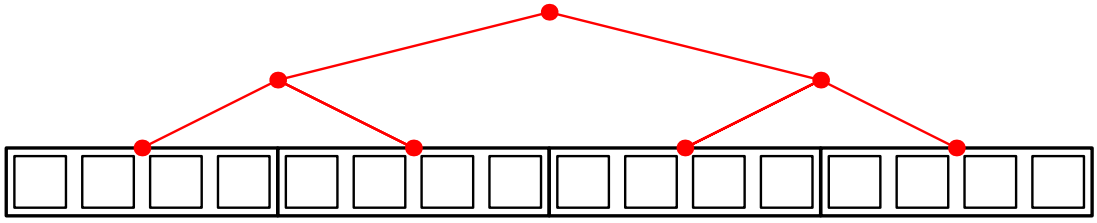

Figure 1.4: The array divided into pages with the imaginary tree of height 2.

For every node we define *density* as the number of elements actually stored in the corresponding interval divided by the capacity of the interval. We will redistribute elements so that the density of nodes at level l will be at most $\frac{1}{2} + \frac{l}{2h}$. Therefore the density of leaves can approach 1, with the restriction getting stricter with decreasing levels, up to the root having a density of up to $\frac{1}{2}$.[1]

When a new element needs to be inserted, we simply insert it into a page according to its position in the order. The entire page can be completely rewritten in $\Theta(i)$ time, so there are no problems there.

If a page reaches density of 1, we progress up the tree, looking for the first (lowest) node that has a permissible density.[2] The interval corresponding to that node is then completely redistributed. If no such node is found, we have violated the restriction on total number of elements in the data structure.

1.2.2 Amortization

Suppose that we are currently redistributing an interval corresponding to some node of level l and capacity K. This can be done in $\mathcal{O}(K)$ time, which we shall need to amortize. We are redistributing this interval because a child's interval exceeded its maximal density of $\frac{1}{2} + \frac{l+1}{2h}$. During the previous redistribution of this node's interval, the currently problematic child had its density set to some value that would be permissible for this node. That density was no more than $\frac{1}{2} + \frac{l}{2h}$, so at least $\frac{1}{2h}K$ elements must have been inserted into that child since then. We need each of the $\Omega\left(\frac{K}{h}\right)$ insertions to contribute $\Theta(h) \in \Theta(i)$ towards the next redistribution of each of the j nodes on the path to the page, so $\mathcal{O}(i^2)$ in total.

[1] We shall keep the density of leaves strictly lower than 1, which requires $k \geq 2$.

[2] It is possible that several predecessors of our page's node have impermissible density too; rebalancing is only triggered by pages.

1.2.3 Important observations

The binary tree over the pages of the array can be purely conceptual in nature. The limit for the density of a given node is easily calculated whenever needed, and we shall also calculate the density when necessary by scanning the entire interval and counting encountered elements. Since the capacity of the interval doubles with every decrease in level, the total cost of all scanning can be covered by the cost of the last scan, which is obviously not asymptotically more expensive than the immediately following redistribution.

A similar argument places an upper bound on the number of block transfers, since scanning an interval of capacity K takes at most $K/B + 2$ block transfers and for $K < B$, the entire previous interval is already in cache.

1.3 Dynamization

In our first attempt at a dynamic version of a vEB permutation, the individual entries are stored in a separate array managed by an OFM algorithm.

A search tree ensures that the occupied slots can be searched in $\mathcal{O}(\log N)$ time, with every internal node either storing a copy of the minimal key from its interval or indicating that the relevant interval is empty. It is stored in a vEB permutation to make the search cache-oblivious.

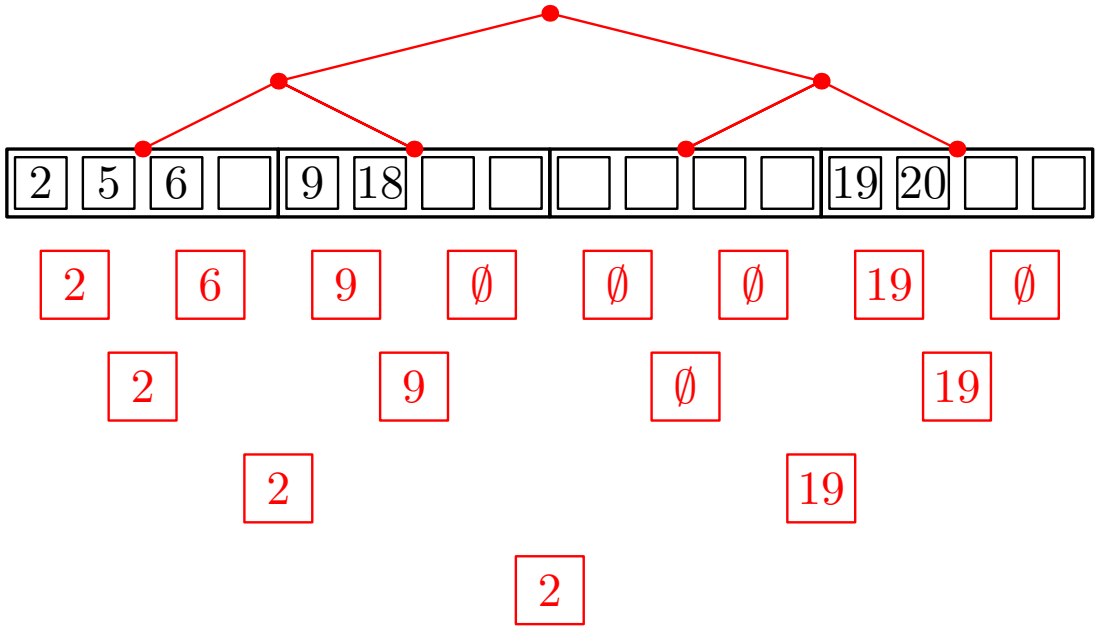

Figure 1.5: The array with OFM and search trees.

Searching is now done in $\mathcal{O}(\log N)$ time and cache-oblivious, but insertions and deletions now have to pay $\mathcal{O}(\log N)$ for the update of the vEB permutation, and $\mathcal{O}\left(\log^2 N\right)$ for OFM. Note that OFM is not slowed down by the updating of the search tree, since we can identify the search subtree that corresponds to the redistributed node and update that entire subtree in linear time.

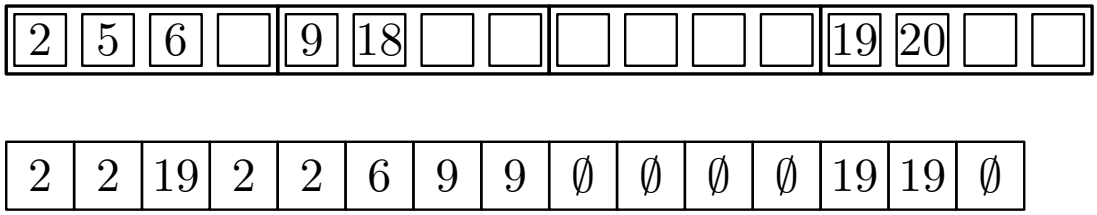

Figure 1.6: Resulting data in memory.

In order to reduce the time complexity of insertions and deletions to $\mathcal{O}(\log N)$, we use indirection with $\mathcal{O}(\log N)$-sized buckets to amortize the cost of OFM. The buckets are kept between full and $\frac{1}{4}$-full by splits, merges and the transfer of entries between neighbouring buckets. Thanks to the small size of the buckets, these can be rewritten in their entirety in $\mathcal{O}(\log_B N + 1)$ block transfers.

We rebuild the entire data structure as needed to keep $\mathcal{O}(\log N) \in \mathcal{O}(\log n)$. The cache-oblivious B-tree thus achieves the bounds of $\mathcal{O}(\log n)$ amortized time per operation and $\mathcal{O}(\log_B n + 1)$ amortized block transfers per operation.

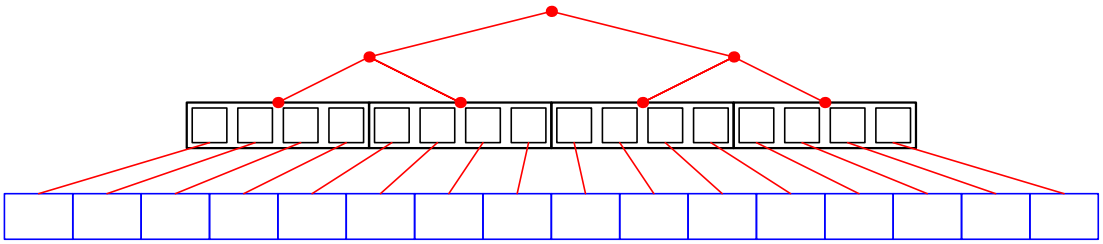

Figure 1.7: Cache-oblivious B-tree. The search tree is omitted for clarity.

Please keep in mind that while cache-oblivious B-trees work in amortized time and we shall initially focus on amortized-time analysis for our data structure, the fourth chapter shall demonstrate a way of deamortizing our data structure. While the same could be done for the B-trees through the use of an unamortized OFM algorithm and by using larger and more complex buckets, we chose to omit the details for brevity.

2. Road to Implicitness

While cache-oblivious B-trees achieve much of what we have set out to achieve but lack the important property of implicitness. Before we delve into the details of how to implement the implicit structure, let us first go over the list of problems to be solved and foreshadow their solutions.

For reasons that will hopefully become clear during this chapter, we shall call what is now the vEB permuted search tree of the cache-oblivious B-tree the *layer of districts* or $\mathcal{D}$. The buckets shall from now on be collectively called the *layer of buckets* or $\mathcal{B}$ for fairly obvious reasons.

2.1 Chunks and buckets

First of all, we need to admit that all the upcoming modifications will make insertions and deletions in $\mathcal{D}$ slower. To amortize the slower operations of $\mathcal{D}$, we shall need to use larger buckets to make the updates less frequent. We will also need to make the entire $\mathcal{B}$ implicit.

F&G describes a way to make $\mathcal{B}$ implicit for any polylogarithmic bucket size. We shall not delve into the details and will assume that we do have an implicit, cache-oblivious version of $\mathcal{B}$ for buckets of polylogarithmic size.

Despite having larger (and therefore fewer) buckets, we cannot afford to store explicit pointers to them in $\mathcal{D}$. In order to encode the needed pointers in the data, the layer of districts shall store a *chunk* of $\mathcal{O}(\log n)$ entries from every bucket, where we can encode a pointer address in the permutation of the chunk's entries. While this also allows us to encode some limited amount of other information, it makes the entire OFM algorithm slower by a factor of $\mathcal{O}(\log n)$, which we shall compensate for by increasing the bucket size as mentioned above.

We also cannot afford to leave empty slots in the ordered file. This means that we shall use *routing* and *filling* chunks, where routing chunks act as the full slots of the ordered file while filling chunks act as the empty slots. Filling chunks will have to be capable of being stored in arbitrary order, easily relocated when their slot is needed by a routing chunk, and still somehow accessible. Every bucket shall thus provide a *group* contsisting of one routing and $\mathcal{O}(1)$ *associated* filling chunks to $\mathcal{D}$. Every routing chunk will encode a pointer to the *containing* bucket and will be the head of a linked list of all the chunks within that group.

Note that the task of $\mathcal{D}$ is identifying the relevant bucket during an operation and handling group insertions and extractions whenever buckets split or merge. $\mathcal{B}$ guarantees that the entries stored in any given routing chunk are smaller than all other entries in that bucket, which allows the use of the routing chunk for bucket identification. Actual insertion and deletion of keys within a bucket (including when this affects the entries stored in the group of chunks) is handled by $\mathcal{B}$ which shall make sure not to disturb data encoded by $\mathcal{D}$.

2.2 Problems of scale

A significant problem for our data structure is that the value of n can vary widely. Among myriad other problems, this makes constructing chunks of $\mathcal{O}(\log n)$ size highly problematic. For the amortized version of the data structure we can afford to rebuild the data structure when over- or underfull, much like with a resizeable array. Chapter four shall introduce an elegant solution for avoiding this problem. From now on let us use N as the current maximal number of entries rebuilding the entire data structure to keep n between $\sqrt{N}$ and N.

Rebuilding shows us a good example of why implicit data structures are useful: given an array of n entries, we can declare a prefix of length 1 as the new structure and then incrementally insert into it all the remaining entries in the same order that they were stored in in the previous structure. This however shows us that we must not ignore the need for the data structure to function in $\mathcal{O}(\log N)$ time even when n is $o(N)$, which will also be later needed by the unamortized version.

The number of buckets can also vary, being only somewhat loosely correlated with the number of entries. While it is tempting to try and solve this problem by similarily rebuilding $\mathcal{D}$ when the number of buckets changes significantly, analyzing the rate at which buckets can split and merge is a little too complicated for our taste. Instead, we shall construct $\mathcal{D}$ gradually. The individual sections of $\mathcal{D}$ shall be called *districts*.[1]

We will number the individual districts $D_0, D_1, D_2, ...$ and we shall later ensure that a sufficiently large number of filling chunks will be available when needed for their size to grow exponentially, limiting their maximal number to $\mathcal{O}(\log n)$. An *index* will contain the minimal entry from each district, enabling fast searching among the districts. This will require us to keep the routing chunks in sorted order across all the districts. Together with the need to keep the routing chunks well-distributed among the districts this will once again lead to the slowing down of $\mathcal{D}$ requiring a further increase in the size of the buckets.

2.3 No cloning allowed

The final obstacle is that we have no place to store the search trees of the districts which we shall call *directories*. Much like with the gaps in the ordered file we will need to find some source of easily moved data to act the part of the nodes with empty subtrees in the vEB permutation. Since we need to distinguish between empty and non-empty nodes, we shall use pairs of keys and encode a flag bit in their order. Initially, a disctrict's directory will be filled with empty nodes that will be made out of *spairs*.[2] There are two spairs *dislocated* from every chunk with the remainder of the chunk — which we shall call *hunk* — encoding pointers to its spairs. Spairs serve as the much-needed small and easily-relocated units of "filler" data.

When a chunk is inserted into a district, the relevant path in the directory needs to be updated. This is done by using the minimal entries from the chunk to replace the elements in those nodes for which the new chunk is minimal.

[1] Hence the name "layer of districts".

[2] F&G called these "pairs of spare keys".

When replacing spairs, those are moved to the chunk to replace the used keys. When replacing nodes containing the entries of a different previously minimal chunk, that chunk takes its pair of entries back and returns the original spair, which will now be stored in the newly added chunk. Deletion of a chunk is handled similarily. While the pointer in the hunk from which the spair was dislocated is not updated, the above system shall ensure that if a spair was stored in a node that is no longer empty, it is currently located in the minimal chunk of that node's interval.

2.4 Auxiliary data

We will later find it convenient to encode some $\mathcal{O}(\log N)$ bits of data in an easily accessible location. A *preamble* of $\mathcal{O}(\log N)$ entries will be stored at the beginning of the data structure, encoding $\mathcal{O}(\log N)$ bits of necessary data in their order. Due to the preamble's small size, it can be read in its entirety during every operation and entries deleted from it are replaced by arbitrary entries removed from the rest of the structure.

3. Amortized Data Structure

Let us now describe the internals of the amortized version. The goal is to achieve $\mathcal{O}(\log N)$ time and $\mathcal{O}(\log_B N + 1)$ block transfers per operation when amortized, while using only $\mathcal{O}(1)$ extra cells of memory.

In the $\mathcal{O}(1)$ permitted extra data we shall only store the number of entries n. The current value of N shall be kept encoded at the beginning of the preamble using Elias gamma coding, which first encodes the number of bits of N in unary and then encodes N itself.(Elias 1975) When N is smaller than some $\mathcal{O}(1)$ limit, the entire data structure is reduced to a sorted array, making N unimportant. While rebuilding the data structure, the preamble might not yet be long enough to encode N, but *operation memory* can be used to store N temporarily.

3.1 Bits and pointers

Because of the requirement for the entire data structure to be implicit, we will need to encode any and all auxillary data in the order of the entries stored within. If we group the entries into pairs, we can encode a single bit of data in the order of the two entries. We will encode a zero by having the minimal of the two keys precede the maximal one, while a one shall be encoded by the inverse order. Since all keys are distinct, we can always distinguish between the two cases. Also, reading an entire pair instead of a single entry does not affect the costs of operations by more than a multiplicative constant.

Note that we could store more data by encoding it in permutations of more than two elements. The permutations of four entries can store more than four bits while under our schema four entries encode only two bits. While the improvement could be by more than a multiplicative constant, the use of only two bits makes the entire system much simpler. This becomes important when analyzing cache behaviour for small cache sizes as there would be a need to use a more involved algorithm for encoding data in groups of entries longer than cache size.

Other than encoding single bits, we shall also need to encode some pointers. The data structure contains at most N entries, making the pointers $\mathcal{O}(\log N)$ bits long when stored as an offset from the zeroth entry. Pointers and $\mathcal{O}(\log N)$-bit integers, such as the sizes of various sections of the data structure, shall be stored by encoding the individual bits into $\mathcal{O}(\log N)$ pairs of entries.

3.2 Layers

The entire memory occupied by the data structure is divided into the following physical entities:

Some $\mathcal{O}(\log N)$ entries are stored at the beginning entirely separately from the rest of the data structure and form the preamble. The preamble encodes important auxillary data such as the current value of N, which is always encoded at the very beginning.

The preamble is immediately followed by the layer of districts. This is the part that we are mainly concerned with in this thesis.

Located last and usually the largest of all the layers is the layer of buckets. While the algorithms managing $\mathcal{B}$ will sometimes manipulate entries stored in $\mathcal{D}$ and they are certainly permitted to encode some information in the preamble, the layer of buckets is where these reign without interference by other algorithms. In this thesis, we shall refrain from explaining the internals of $\mathcal{B}$ in much detail. In case of curiosity, some insights can be gleamed from F&G and a detailed description shall be the subject of a subsequent paper.

During a split of a bucket or a merge of two consecutive buckets, a temporary transitional layer will be created between $\mathcal{B}$ and $\mathcal{D}$, consisting of a single group of chunks while it is being transferred from one layer to the other.

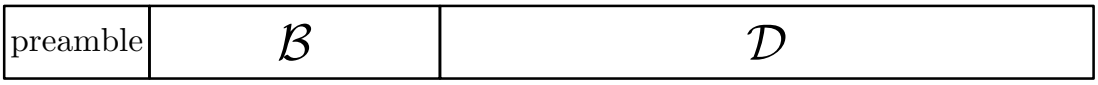

Figure 3.1: The layers as arranged in memory.

3.3 Buckets and chunks

Buckets are the main logical entities that $\mathcal{B}$ works with. There is both an upper and a lower bound on the size of a bucket, both of which are polylogarithmic with respect to N. If we were to take all the buckets in the data structure and for each construct the minimal interval of values that the keys of the entries in that bucket belong to, those intervals will be pairwise disjoint, yielding a total order on the set of all buckets. Buckets are kept within the permissible size range by splits and merges, which are managed by $\mathcal{B}$.

Figure 3.2: Buckets are pairwise disjoint intervals of keys.

The layer of districts mainly works with a different type of logical entities that we call chunks. Every chunk consists of some constant $k \in \Theta(\log N)$ entries, allowing us to encode a constant number of integers and pointers within it. Chunks enter $\mathcal{D}$ in groups and every group consists of a single routing chunk and 4 associated filling chunks.

As far as $\mathcal{D}$ is concerned, every time $\mathcal{B}$ splits a bucket, it genreates a single group for insertion. Immediately before $\mathcal{B}$ merges two neighbouring buckets, it requests the removal of the group contained by the latter of the two buckets. The entries contained by the routing chunk are smaller than any other entries within the bucket, so during an operation concerned with key x, the relevant bucket can be found by searching the routing chunks.[1] Note that much like we did with buckets, we define a total order on the set of all routing chunks.

[1]The relevant bucket is the bucket containing the routing chunk that contains the largest entry that is not greater than x. If no such bucket exists, the very first bucket is relevant.

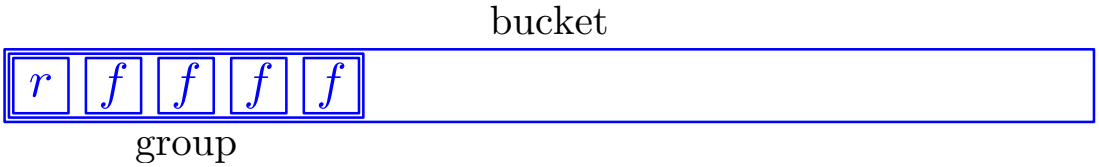

Figure 3.3: The entries of routing and filling chunks within a bucket.

3.4 Spairs

Spairs are pairs of entries *dislocated* from a chunk. Every hunk has two dislocated spairs and encodes a pointer to each of them. Spairs can contain arbitrary entries from the chunk with the exception of some $\mathcal{O}(\log N)$ smallest entries of a routing chunk, which might be needed for path activation in the directory.

When a spair is moved, the hunk from which it was dislocated needs to have its pointer updated. This is done by searching for one of the keys in the spair, which locates the routing hunk of the correct group, and finding the right pointer to update within a group is trivial.

Note that updating the pointer is expensive, which is why we avoided having to do so too often in section 2.3.

3.5 Structure of $\mathcal{D}$

The layer of districts is divided into the following physical entities, listed in order:

- $\mathcal{O}(\log N)$ *districts* $D_0, D_1, \ldots$
- A single *filling zone* $\mathcal{F}$
- An index

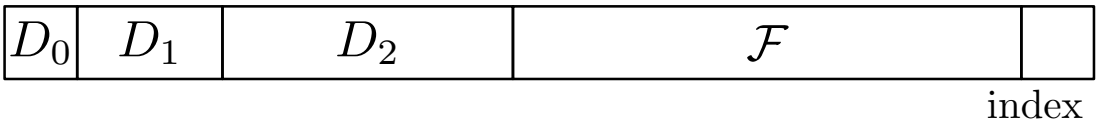

Figure 3.4: Physical subdivisions of $\mathcal{D}$.

The district D_i logically contains $2^{i+2} - 1$ spairs and 2^{i+1} hunks, of which at most 2^i are routing. The district D_0 is an exception, containing nothing more than a single spair and routing hunk. Physically, every district is always 2 entries smaller due to the index, but we shall ignore that for now. All routing hunks are stored in the districts.

Across all the districts, routing hunks are stored in order, with every routing hunk in D_{i+1} being greater than any routing hunk in D_i. The first i_{max} districts each contain their maximum number of routing hunks, $D_{i_{max}}$ contains at most $2^{i_{max}}$ routing hunks and all following districts contain no routing hunks. A district containing the maximal or minimal number of routing hunks will be called *full* or *empty* respectively. Any remaining spairs and filling hunks are located in $\mathcal{F}$.

Individual hunks and spairs from a group can be located in different districts with spairs and filling hunks also being able to end up in $\mathcal{F}$.

3.5.1 Structure of a district

When empty, the first $2^{i+3}-2$ cells of D_i are occupied by $2^{i+2}-1$ arbitrary spairs, which form a *directory*. The remainder of the district is an *array* of 2^{i+1} hunks, all of which are initially filling.

The $2^{i+2}-1$ pairs of entries in the directory represent the nodes of a complete binary tree the leaves of which correspond to the individual hunks of the array. The entire binary tree is encoded using a vEB permutation, which is crucial for the cache-obliviousness of the data structure.

Each node of the directory is in one of two states, between which we distinguish using the bit encoded by the order of the two entries. A zero bit indicates that the node's subtree contains no routing hunks and that the node itself consists of a spair dislocated from an arbitrary hunk. This is the initial state of all nodes. A node whose subtree contains at least one routing hunk can be easily identified by it encoding a one bit and such a node consists of the two entries with minimal keys from the minimal hunk in its subtree, with the exception of any keys which are currently used in this manner by this node's predecessors. We shall discuss how the directory is maintained later.

The array itself is maintained by the OFM algorithm. Note that the directory can be used for computing the densities, saving us from spending time scanning over the much larger hunks.

3.5.2 Special properties of D_0

The zeroth district differs from the others. Here is a brief overview of its special properties.

Since D_0 is created by the first bucket, $\mathcal{F}$ is non-existant before its creation. As such, we need to conserve filling hunks and spairs, which might soon be needed for D_1. This is why D_0 contains just a single routing hunk, no filling hunks, and a spair. The spair is initially used for the directory, but it is immediately replaced by the minimal entries of the routing hunk.

Even without a filling hunk in it, the creation of D_0 causes fewer filling hunks to be added to $\mathcal{F}$ than a regular group insertion, because there was no filling hunk in D_0 that would get displaced by the routing hunk. This is the sole reason why a group needs to contain 4, rather than 3, filling hunks.

Since routing hunks contain the entries with the smallest keys of their group and the routing hunk in D_0 belongs to the group with the lowest key values among all the buckets, splits and merges never affect D_0 directly, as it is always the groups with larger keys that are inserted or deleted. This fact helps reduce the number of special cases needed for handling D_0.

3.5.3 Explanation of the index

The last remaining gap in our understanding of the process of bucket identification is the identification of the correct district to search. This is the task of the index, which physically contains the root node of every directory, causing every district to be physically 2 entries shorter than it is logically. Due to the inequality among the routing hunks of different districts and the fact that the root of the directory of an nonempty district consists of the two entries with the smallest keys among

all the buckets reachable through that district, we can use the index to identify the relevant district. Due to its length of $\mathcal{O}(\log N)$, the entire index can be read during every operation. For convenience's sake, we shall assume that the index is reversed when compared with the order of the districts.

3.5.4 How $\mathcal{F}$ works

In order to understand how $\mathcal{F}$ works, we need to demonstrate that there is always an excess of spairs in it. The filling zone is created at the same time as D_0. This causes 4 filling hunks and 9 spairs to be inserted into $\mathcal{F}$. Every subsequent group insertion causes 5 filling hunks (4 from the group + 1 that got displaced by the routing hunk) and 10 spairs (2 for each chunk in group) to be added to $\mathcal{F}$. An empty district D_{i+1} requires 2^{i+2} hunks and $2^{i+3} - 1$ spairs, while filling D_i caused $5 \cdot 2^i$ hunks and $10 \cdot 2^i$ spairs to be inserted.[2] Therefore, the filling zone $\mathcal{F}$ can always complete a new empty district before all districts become full, and there is an excess of roughly one spair per existing district in it.

Immediately after completing a new district, $\mathcal{F}$ contains only some spairs. These will become the directory of the next district. When a new group is being inserted into $\mathcal{D}$, spairs are used to continue building the directory. If the directory is already complete, excess spairs are stored at the very end of $\mathcal{F}$. Note that since there are $\mathcal{O}(\log N)$ of them, we will be able to amortize their relocation when filling hunks get inserted in front of them. It will be from these excess spairs that we will take the one spair to be inserted into the index.

While building a district, filling hunks are stored after the spairs that will form the directory. Since the future directory is usually not complete yet, we need some way to avoid having to shift all the filling hunks while building the directory. This is why we shall store the array filling hunks rotated such that the divides between filling hunks are aligned with their final positions. This change causes at most one filling hunk to be split into two parts, which will at worst require some auxillary data in the preamble, and every insertion or deletion of a spair will now cause at most one hunk to be merged and one hunk to be split.

During a group extraction, all hunks and spairs from the group are replaced by filling hunks from $\mathcal{F}$, which begins the disassembly of an empty district if needed.

3.6 Maintaining a small data structure

As we discussed before, we cannot avoid the situation where the data structure contains few entries. In the amortized version, rebuilding causes this temporarily, while the deamortized version will need this too.

If the size of the entire data structure is $\mathcal{O}(\log N)$, it can always be read whole, making operations trivial.

As n increases, the preamble forms first, initially containing the information that there are no buckets. We shall rely on the layer of buckets being capable of constructing an undersized bucket: provided $\mathcal{O}(\log N)$ entries, it shall construct a single undersized bucket whose root will be referred to by the preamble.

[2] With the exception of D_0 of course.

This undersized bucket can then gradually grow towards its polylogarithmic regular size. When the bucket is comfortably larger than its minimum size under regular operation, it will provide its group to $\mathcal{D}$, forming D_0 and commencing regular operation.

Similarily, once only a single bucket remains and it has become very small, the group is returned to it, a pointer to the root is encoded in the preamble, and exceptional state is commenced. The bucket then can become undersized, eventually dissolving into $\mathcal{O}(\log N)$ entries.

3.7 Operations and procedures

Now that we know what $\mathcal{D}$ should look like, let us consider how to conduct the supported operations without the layer breaking. The main operations that the entire data structure supports are mostly handled by $\mathcal{B}$. This section lists operations and procedures supported by $\mathcal{D}$.

Note that we assume regular operations — if exceptional state is in effect, $\mathcal{D}$ does not exist and operations are trivial. Also note that the data structure's operations also have to check the preamble. Since the possible resulting special cases are rather trivial, we shall ignore that.

3.7.1 Bucket identification

Arguably the most important operation, since it is an indispensable component of all of the data structure's operations. This operation takes a key and returns a pointer to the root of the relevant bucket.

This is how bucket identification works during an Insert, Find or Delete: First, a district is found by looking into the index. Then the directory is used to identify a single routing hunk. The routing hunk is read, decoding the pointer to the containing bucket.

The globally minimal entry is the minimal entry of the routing hunk in D_0, which makes it the root of the directory of D_0, which places it into the index, where it is the least entry.

The globally maximal entry is located in the bucket of the last routing hunk in the last non-empty district. We can easily find the last non-empty district using the index, and then search for the last routing hunk in the directory.

The Pred(x) operation is a little more complex. We have to first use the index to identify the last district that contains an entry smaller than x. We then locate the last routing hunk that contains an entry strictly smaller than x. Note that this is made slightly harder by the fact that we might encounter all such entries from that chunk while searching the directory. Therefore it might happen that all non-empty child nodes contain entries larger than x. In such cases, we either follow the path of least entries possible, or make use of the fact that we must have seen the correct entry along the way.

During the Succ(x) operation, we identify the relevant bucket as normal, but we also need to take note of the minimal entry of the nearest larger bucket, which we are guaranteed to encounter while searching.

3.7.2 Group insertion

When $\mathcal{B}$ decides to split a bucket in two, the latter of the resulting buckets generates a group and leaves them in the transitional layer between $\mathcal{D}$ and $\mathcal{B}$. The earlier of the two buckets continues using the original group.

The new routing hunk r now needs to be inserted into the main area. It should be inserted either into the last district that contains a key smaller than r, or (if that district is full and r would be maximal in it) into the district immediately following that. There are two things to note here:

- $\mathcal{F}$ generates a new district no later than when all districts become full.
- Since D_0 already exists, we cannot be inserting a globally minimal hunk, because of the way buckets are split.

Since all but the last district are kept full, we might need to shift some routing hunks in order for r to fit in. If we are inserting into a district that is already full, we take the maximal hunk from the last full district and move it into the following non-full district, where it will be the new minimal hunk. We continue removing and inserting routing hunks between the preceding districts until the district into which r should be inserted becomes non-full.

When the routing hunk is inserted, $\mathcal{F}$ takes the spairs of the new group and either integrates them into the future directory of the next district or leaves them at the end as excess spairs. After updating the relevant pointers to spairs, the five filling hunks (including the one displaced by the routing hunk insertion) are integrated into the rotated array of filling hunks. If a district is completed during this process, it is immediately output by the filling zone, which shrinks and then continues incorporating the remaining filling hunks.

3.7.3 Group Extraction

A group extraction is almost exactly the same as group insertion but in reverse. Five filling hunks are moved from $\mathcal{F}$ into the transitional layer, which might cause the dismantling of another empty district. Ten arbitrary spairs are set aside. All of these filling hunks and spairs are replaced by the elements of the group to be extracted. Routing hunks are then moved between neighbouring districts as needed to reestablish the invariant on district fullness. Finally, control over the transitional layer is handed over to $\mathcal{B}$, concluding group extraction.

3.7.4 Directory maintenance

When a routing hunk is being moved, regardless of what caused the relocation, the directory needs to be updated accordingly.

When a hunk is moved into a position, the path leading to it in the directory needs to be *activated*. When traversing the path from the root of the directory to the new position of the hunk, we observe three phases, which are distinguished by the entries stored in the encountered nodes:

1. Nodes containing entries from a smaller routing hunk
2. Nodes containing entries from the nearest larger routing hunk
3. Nodes containing a spair

Note that not all phases need to actually occur. Nothing needs to be done during the first phase and the third phase is trivial. During the second phase, we identify the nearest larger routing hunk and have it conduct a *simplified path deactivation* during which it simply replaces the nodes occupied by its entries with the spairs that were originally located there. We can then simply activate the path with our entries, followed by the simple activation of the path to that nearest larger routing hunk. Both simple path activations are now guaranteed not to have a second phase, the directory is left in the correct state and all spairs are where they are expected to be according to Section 2.3.

Before a routing hunk can be moved out of its current position, the path leading to it needs to be *deactivated*. We identify the nearest larger routing hunk, then conduct a simplified path deactivation for both the hunk to be removed and the nearest larger routing hunk. A simple path activation for the nearest larger routing hunk is the last step required before the hunk can be moved.

Notice that both path activations and deactivations require $\mathcal{O}(1)$ traversals of some path in the directory, resulting in an upper bound of $\mathcal{O}(\log N)$ time and $\mathcal{O}(\log_B N + 1)$ block transfers.

3.8 Complexity

Let us analyze the complexity of all operations of $\mathcal{D}$.

Bucket identification takes $\mathcal{O}(\log N)$ time and $\mathcal{O}(\log_B N + 1)$ block transfers in the worst case due to the properties of the vEB permutation.

When $\mathcal{B}$ will be changing the entries stored in the group, all such modifications will require $\mathcal{O}(1)$ scans of the entire group. The typical activities of $\mathcal{B}$ in such situations are value changes, entry insertion and deletion (which will require some entries to shift), and changing the data encoded by $\mathcal{B}$ in those parts of the chunk where there is no data encoded by $\mathcal{D}$.

Notice that the group consists of $\mathcal{O}(1)$ hunks and $\mathcal{O}(1)$ spairs. The filling hunks form a linked list behind the routing hunk, while spairs are found by looking at the location indicated by the pointers and potentially traversing the directory to find the chunk that replaced that node with its entries. All of the above sums up to $\mathcal{O}(\log N)$ time and $\mathcal{O}(\log_B N + 1)$ block transfers in the worst case.

Group insertion and extraction are the only difficult operations. $\mathcal{F}$ spends $\mathcal{O}\left(\log^2 N\right)$ time updating the $\mathcal{O}(\log N)$ pointers to the $\mathcal{O}(\log N)$ excess spairs that need shifting. The index is similarily slow to make way for the expanding $\mathcal{F}$ due to the need to update the pointers to the spairs that were used for the root nodes of the individual directories.

All of the above is fast in comparison with the insertion of the routing hunk, which requires $\mathcal{O}(\log N)$ routing hunk insertions and extractions in the districts to make space for the inserted routing hunk and to reestablish the invariant on district fullness. Every routing hunk insertion and deletion in a district causes $\mathcal{O}\left(\log^2 N\right)$ amortized hunk moves within the district due to OFM, each of which requires $\mathcal{O}(\log N)$ time for directory updates, swapping of entries and updating the pointers to moved hunks.[3]

[3]Note that this also covers the cost of rearranging the $\mathcal{O}(\log N)$ hunks within a page.

In total, even with amortization, $\mathcal{O}\left(\log^4 N\right)$ time is needed for a single group insertion or extraction with a similar bound on the number of block transfers. This can be solved by $\mathcal{B}$ making the individual buckets large enough to amortize a group insertion or deletion over $\Omega\left(\log^3 N\right)$ Insert and Delete operations.

4. Deamortization

All that remains for us to show, is that all that amortization is unnecessary. We used amortized analysis four times in the preceding chapters:

- We rebuild the entire data structure periodically to keep $\log n \in \Theta(\log N)$.
- We use an OFM algorithm with $\mathcal{O}\left(\log^2 N\right)$ amortized time complexity.
- We amortize a group insertion or deletion over $\Omega\left(\log^3 N\right)$ operations.
- We ignore the possibility of two buckets spliting close after one another.

Here is how to solve all of these problems without amortization.

4.1 Portions

Let us partition the entire memory into $\mathcal{O}(\log\log n)$ *portions* where the p^{th} portion is a copy of the amortized data structure using 2^p-bit pointers and therefore having maximal size $N \geq 2^{2^p}$. Portions are physical entities and are stored in order with the 0^{th} portion occupying the first 2 cells and the $(p+1)^{\text{st}}$ portion occupying the $2^{2^{p+1}}$ cells immediately following the p^{th} portion.

There is no restriction on which entries end up in which portion. This means that at least in some cases searching will have to be applied to every portion. Insertion shall always occur in the last portion unless it is full, in which case a new portion is started instead. Deletion from a portion that is not the last will create a temporary gap between the affected portion and the following one. This is immediately remedied by a transfer of an arbitrary entry from the last portion to the affected one.

4.1.1 Achieving logarithmic complexity

For now we shall assume that all three operations can be completed on a portion of maximal size N in $\mathcal{O}(\log N)$ time.[1] We now demonstrate that this is sufficient for a total time bound of $\mathcal{O}(\log n)$.

Observe that if the p^{th} portion exists in our data structure, either n is strictly larger than $2^{2^{p-1}}$ or that is the first and only portion and the entire structure is trivially small. In the latter case, we declare all operations in the 0^{th} portion, which contains $\mathcal{O}(1)$ entries, to take $\mathcal{O}(1)$ time. In the former case, we can see that $\log(2^{2^p}) \in \mathcal{O}(\log n)$ and therefore we can spend $\mathcal{O}(\log N)$ time on an operation in a portion of maximal size N. This alone suffices to show that Insert operations do not exceed the time restriction.

Deletion in any portion that is not the last simply needs to be followed by a deletion of an arbitrary entry in the last portion which is then inserted into the affected portion. This causes the affected portion to contract and then expand by one cell of memory, while the last portion (and the entire structure) contracts by one cell, leaving no unused cells in the data structure. Since both insertions and deletions will run in $\mathcal{O}(\log N)$ time, the entire deletion operation will be completed in $\mathcal{O}(\log n)$ time.

[1] We have already demonstrated that this is the case in amortized time.

As we established earlier, searching needs to be applied to every portion. We already know that searching the last portion always runs in $\mathcal{O}(\log n)$ time. All the remaining portions are full, and the logarithm of the size of each portion is half the logarithm of the size of the following portion, while the last and largest of them is not larger than n. Since the times needed to search these portions form a geometric progression, we can see that the total time needed for executing any $\mathcal{O}(\log N)$-time operation on all portions adds up to $\mathcal{O}(\log n)$ time in total. This proves that none of the operations exceeds the $\mathcal{O}(\log n)$ time bound.

Similarily, we can also show that if an operation applied to all portions uses only $\mathcal{O}(\log_B N + 1)$ block transfers per portion in the cache-oblivious model, then it being applied to all portions will use $\mathcal{O}(\log_B n + 1)$ total block transfers.

Note that the $+1$ in the portion block transfer complexity is only important for $\log_B(N) < 1$, which implies $B > N$. If the entire portion is smaller than a single block, then it will certainly require no more than two block transfers. Due to how the portions grow in size, all previous portions are in total significantly smaller than a single block, which implies that all the $+1$s will never add up to more than $\mathcal{O}(+1)$, avoiding the need for $+\log(\log(n))$ that would be otherwise necessary.

4.2 Unamortized OFM

The unamortized OFM algorithm functions similarily to the amortized version. The entire array is divided into some 2^j pages of size $\Theta(\log N)$ and a binary tree of intervals of pages is constructed.[2] Intervals of pages have a limit on their density that gets tighter for larger intervals.

Unlike the amortized algorithm, the nodes with a dangerously high density are merely flagged as such, and their interval is redistributed over the following insertions and deletions. The complicated part is ensuring that the algorithm avoids "threshing" — a situation where two redistributions interfere, which has the potential to make the entire system fail.

Note that in this case, some extra data will need to be stored by the algorithm. Fortunately, we can encode $\mathcal{O}(N)$ auxillary data in the chunks of that district. The $\mathcal{O}(\log N)$ slow-down caused by the algorithm decoding and encoding data will be hidden by the fact that moving hunks is also $\mathcal{O}(\log N)$-times slower than moving single elements.

The known strategies used to manage the redistributions are far too involved for us to detail here, which is why we recommend the papers by Willard (1992) and Bender et al. (2002) to curious readers.

We can no longer assume that the redistributions will not need to shift the routing hunks within a page. In the amortized version of the OFM algorithm, entire pages were rewritten at once, making it simple to avoid the $\mathcal{O}\left(\log^2 N\right)$ cost of shifting a page. In the deamortized version, redistributions work gradually and can be interrupted, possibly causing the page to shift with every hunk move. Left as is, this would increase the cost of a group insertion by a factor of $\mathcal{O}(\log N)$, forcing us to use larger buckets.

[2] In this section, N signifies the capacity of the array managed by OFM, not the maximal portion size.

In order to avoid this, we shall store the page out of order. This would complicate path activation and deactivation in the directory, since manipulating just the path to the nearest larger routing hunk might no longer be sufficient. This is why we instead construct a smaller directory whose leaves correspond to pages and we use the minimal routing hunk of the page for path activation. Some of the space freed up by the smaller directory we use for what we call *page overviews* — for every page, an array of one spair for every hunk in the page will be allocated and a pair of entries from each routing hunk will be stored there, similarily to the directory, in the same order the hunks are physically in.

When a page is identified by the directory, reading the relevant page overview in $\mathcal{O}(\log N)$ time provides us with all the information needed for us to locate the relevant hunk. This keeps directory maintenance easy and directory use fast.

4.3 Cycles and steps

Although inserting or extracting a group now takes $\mathcal{O}\left(\log^4 N\right)$ time in the worst case, we now need to split this task into $\mathcal{O}(\log N)$-time *steps*, one of which will be conducted along with each of the next $\mathcal{O}\left(\log^3 N\right)$ operations. We shall, for now, ignore the possibility of two buckets splitting soon after one another.

Inserting and extracting filling hunks and spairs into and from the filling zone can be conducted in $\mathcal{O}(\log N)$ time as before, but this causes $\mathcal{O}(\log N)$ pointers to need updating, which we lack the time to do immediately. Instead, we store the range of positions the outdated pointers point to in the preamble and over the next $\mathcal{O}(\log N)$ operations we locate and update individual pointers, shrinking the range as we do. Note that the range of position is shorter than the distance the pointers moved, which helps us avoid ambiguity.[3]

As for routing hunk insertion and extraction, it consists of $\mathcal{O}(\log N)$ transfers of a hunk between neighbouring districts each of which is followed by $\mathcal{O}\left(\log^2 N\right)$ hunk moves as required by the OFM algorithm. Since we can encode where in this process we left off last time and $\mathcal{O}(1)$ auxillary data in the preamble, we can quite naturally divide this task into the $\mathcal{O}\left(\log^3 N\right)$ steps needed.

In this way, $\mathcal{B}$ initiating a group insertion or extraction triggers a *cycle* of $\mathcal{O}\left(\log^3 N\right)$ steps. The final problem is now ensuring that $\mathcal{B}$ can wait with initiating a cycle until the previous cycle concludes.

4.4 Splits and merges

In the amortized version, buckets could be restricted to $\Theta\left(\log^3 N\right)$ size by splits and merges.

In our unamortized data structure, if two buckets were to exceed the bounds during consecutive operations, the latter of the two could be forced to wait $\Omega\left(\log^3 N\right)$ operations for the cycle initiated by the earlier bucket to complete.

[3]There is roughly one excess spair and exactly one index node per district, of which there are no more than $\log N$. These got moved by 5 hunk sizes with every hunk having to be long enough to encode several $\log N$-bit pointers.

During this time, $\Omega\left(\log^3 N\right)$ more buckets could exceed their bounds. This might overwhelm $\mathcal{B}$, letting some buckets to reach extreme sizes, potentially making their implementation fail.

In order to avoid this, we make the buckets larger, increasing the separation between the absolute minimum possible size, the bound for triggering a merge, the bound for triggering a split and the absolute maximum possible size. We shall also require that $\mathcal{B}$ supports the transfer of entries between two neighbouring buckets (according to the order based on the contained entries), so that we avoid merging a small bucket into an already large one. How precisely is this done shall be explained in our future work on the details of $\mathcal{B}$, so for now we shall assume that the result is similar to merging and then splitting the two buckets, and is no slower (when measured in steps taken) than a regular split or merge.[4]

Up to $\Theta\left(\log^3 N\right)$ buckets might exceed their size bounds during a single cycle with further $\Theta\left(\log^3 N\right)$ exceeding them for every bucket that we deal with. Clearly, we lack the power to prevent almost all of the buckets to be outside their bounds at once. With the buckets being supposed to be merely polylogarithmic in size, there can clearly be more than a polylogarithmic number of them and so, unless we pick which bucket to process next carefully, a single bucket could grow to non-polylogarithmic size (or, conversly, become completely empty) in the time needed to process all of the buckets outside their bounds. Unfortunately, we lack the time to track the precise order of bucket sizes, as it can change too quickly. Clearly a more refined solution is needed.

As we shall later see, we shall need the sizes of the buckets to be $\Theta\left(\log^4 N\right)$. Without loss of generality, let us assume that all the buckets are growing. If we can show that none of them can exceed some $\Theta\left(\log^4 N\right)$ upper limit on size, we can then apply a similar algorithm to ensure that no bucket gets below some $\Theta\left(\log^4 N\right)$ lower limit, with the two algorithms clearly being capable of working together as they work on disjoint sets of buckets and can alternate in conducting splits and merges.

4.4.1 The game of split-a-bucket

For the sake of simplicity, let us set $L \in \Theta(\log N)$ such that the length of a cycle is at most L^3 and $L > \log N$. We then construct buckets with absolute maximal size of at least $(a+2)L^4$ for some $a \geq 2$ and set the (by now mostly symbolic) upper bound on their size to $(a+1)L^4$. This allows us to split all the possible bucket sizes into *bands* of L^3 consecutive sizes.

For our analysis, we shall now keep track of the number of buckets whose size falls into any of the $L+1$ largest bands, with all the remaining buckets being lumped together as uninteresting. In the final implementation, $\mathcal{B}$ shall maintain a linked list of buckets for each of the L largest bands. For convenience, we shall number the bands with Q_1 to Q_L being the bands between the upper bound and the absolute limit, Q_0 being the largest band below the limit, and we shall use Q_- to refer to the sum of all the remaining bands.

Our algorithm is relatively simple: while buckets larger than the bound exist, we split an arbitrary bucket from the largest non-empty band every L^3 operations.

[4] It will in fact be significantly faster.

The remainder of this subsection we shall devote to showing that this is sufficient to prevent any buckets from exceeding the absolute limit.

4.4.2 Analysis of our strategy

There are clearly no more than N buckets. Since no buckets are split unless they exceed the size bound, our strategy does not prevent the size of all the buckets from being exactly at the size bound. For the sake of the argument, let us consider the unrealistically bad situation, when there are N buckets at the size bound. They are still in Q_0, but a single insertion can push any of them into Q_1.

Note that in order to conclusively demonstrate that our strategy will never be overwhelmed, we currently ignore the fact that having N buckets would require an impossible number of entries, especially if they are to be all at the size bound making further insertions and splitting nonsensical.

Let us now consider a single bucket as it ascends through the size bands. Between any two band-changes, there have to be at least L^3 operations involving the bucket, which gives us the time to complete a single split. Since we always split a bucket from the largest non-empty band, only one half of all the buckets that ever reach Q_i will reach Q_{i+1}.

Notice that when a bucket is split, the two resulting buckets end up in Q_-, making the option of getting either of them to Q_1 cost the "antagonist" a more valuable bucket from some higher band.[5]

We can now conclude that while all buckets can be moved to Q_1, we will have the time needed to ensure that no more than half of them get to Q_1 and so on. A single bucket might reach Q_L, but in the time it needs to grow by the final L^3 entries, a cycle is quaranteed to start and the bucket will be split mercilessly.

[5]If no buckets larger than the bound remain, we have won.

5. Progress made

While this thesis might appear to simply reiterate a part of the claims of F&G, we made many improvements to the original paper, many problems of which stemmed from the fact that it was merely an extended abstract of a paper that was not published afterwards.

While F&G dives directly into the implementation of the worst-case version, we decided to take the gentler path of first introducing cache-oblivious B-trees, which are surprisingly similar to the final data structure, describing the amortized version second, and only then deamortizing it.

We strove to make the text as easy to follow as possible. A lot of terminology was changed to ensure that different constructs have different names. Care was taken to distinguish physical and logical entities, including the use of three colours in the illustrations.

Individual spare keys were replaced by spairs both to reduce the number of pointers encoded in each hunk, and to make the structure of the directories slightly simpler. We also invented page overviews, avoiding the use of the more involved version of Willard's algorithm.

F&G describes storing the index in the preamble. When a new district is formed, this arrangement would cause the displacement of some entries from the preamble and those would have to be inserted into the rest of the structure. The index is small enough to be kept at the end of $\mathcal{D}$, which seems to us to be a more elegant solution.

A thorough examination of F&G reveals a contradiction: If every chunk has $\mathcal{O}(1)$ spare keys, every routing chunk has $\mathcal{O}(1)$ associated filling chunks and D_0 contains $\mathcal{O}(1)$ routing chunks, then there will eventually be a portion in which $\Omega(\log N)$ spare keys will not accumulate in time for the formation of D_1, which the original paper states to be some (presumably non-zero) multiple of chunk size in length. While it is clearly necessary to avoid such a constraint, the future directory is directly followed by the incomplete array of filling hunks, which is too long for us to shift with each addition of spairs. We solve the problem by storing the array of filling hunks rotated as described in section 3.5.

The original paper took $\mathcal{O}(\log N)$ steps for path activation and deactivation, which meant that the directory was not always up to date. We discovered a simple and fast way of updating the directory in a single step.

F&G does not present any source of filling chunks for $\mathcal{D}$, with only the fact that there are $\Theta(1)$ of them associated with every routing chunk, and that the number can range between two suitable constants. Furthermore, a bucket in the original paper can be associated with either a routing or a filling chunk. Unfortunately, the section that promised to describe the creation of filling and routing chunks due to the splitting of buckets consists of mere two paragraphs and explicitly states that routing chunks are created. We decided to simply have every bucket provide a single group of one routing and four filling chunks.

We also had to figure out an effective strategy for the splitting and merging of buckets, which revealed the need for an $\mathcal{O}(\log N)$ increase in bucket size.

Finally, we also deduced some acceptable values for constants from F&G, generally trying to make the entire data structure as simple as possible.

Conclusion

In our attempt to understand the original paper by Franceschini and Grossi, we identified several gaps and minor flaws in it. Focusing on the upper layer of the data structure, we filled in the gaps, fixed the flaws that we encountered, and introduced several improvements. We then focused on explaining the layer in greater detail in an effort to both make it easier to understand the result and to demonstrate that it is correct.

We shall continue in our work on this data structure and we expect to publish a paper on the layer of buckets in the near future. The goal of our research is to reconstruct the entire data structure, providing a detailed explanation and a legible reference implementation, hopefully finally answering the question of the existence of implicit ordered dictionaries.

Bibliography

Rudolf Bayer and Edward Meyers McCreight. Organization and Maintenance of Large Ordered Indexes. *Acta Informatica*, 1(3):173–189, Sep 1972. ISSN 1432-0525. doi: 10.1007/BF00288683. URL `https://doi.org/10.1007/BF00288683`.

Michael Anthony Bender, Erik Duncan Demaine, and Martin Farach-Colton. Cache-Oblivious B-trees. In *Foundations of Computer Science, 2000. Proceedings. 41st Annual Symposium on*, pages 399–409. IEEE, 2000.

Michael Anthony Bender, Richard Cole, Erik Duncan Demaine, Martin Farach-Colton, and Jack Zito. Two Simplified Algorithms for Maintaining Order in a List. In *European Symposium on Algorithms*, pages 152–164. Springer, 2002.

Paul Frederick Dietz. Maintaining Order in a Linked List. In *Proceedings of the fourteenth annual ACM symposium on Theory of computing*, pages 122–127. ACM, 1982.

Paul Frederick Dietz and Daniel Dominic Kaplan Sleator. Two Algorithms for Maintaining Order in a List. In *Proceedings of the nineteenth annual ACM symposium on Theory of computing*, pages 365–372. ACM, 1987.

Peter Elias. Universal Codeword Sets and Representations of the Integers. *IEEE Transactions on Information Theory*, 21(2):194–203, March 1975. ISSN 0018-9448. doi: 10.1109/TIT.1975.1055349.

Robert W Floyd. Algorithm 245: Treesort. *Communication of the ACM*, 7(12): 701, December 1964. ISSN 0001-0782. doi: 10.1145/355588.365103. URL `http://doi.acm.org/10.1145/355588.365103`.

Gianni Franceschini and Roberto Grossi. Optimal Worst-Case Operations for Implicit Cache-Oblivious Search Trees. In Frank Dehne, Jörg-Rüdiger Sack, and Michiel Smid, editors, *Algorithms and Data Structures*, pages 114–126, Berlin, Heidelberg, 2003. Springer Berlin Heidelberg. ISBN 978-3-540-45078-8.

Gianni Franceschini, Roberto Grossi, James Ian Munro, and Linda Pagli. Implicit B-trees: New Results for the Dictionary Problem. In *Foundations of Computer Science, 2002. Proceedings. The 43rd Annual IEEE Symposium on*, pages 145–154. IEEE, 2002.

Greg Norman Frederickson. Implicit Data Structures for the Dictionary Problem. *Journal of the ACM (JACM)*, 30(1):80–94, 1983.

Matteo Frigo, Charles Eric Leiserson, Harald Prokop, and Sridhar Ramachandran. Cache-Oblivious Algorithms. In *Foundations of Computer Science, 1999. 40th Annual Symposium on*, pages 285–297. IEEE, 1999.

Alon Itai, Alan Gustave Konheim, and Michael Rodeh. A Sparse Table Implementation of Priority Queues. In *International Colloquium on Automata, Languages, and Programming*, pages 417–431. Springer, 1981.

James Ian Munro. An Implicit Data Structure Supporting Insertion, Deletion, and Search in $\mathcal{O}\left(\log^2 n\right)$ Time. *Journal of Computer and System Sciences*, 33 (1):66–74, 1986.

James Ian Munro and Hendra Suwanda. Implicit Data Structures for Fast Search and Update. *Journal of Computer and System Sciences*, 21(2):236–250, 1980. ISSN 0022-0000. doi: https://doi.org/10.1016/0022-0000(80)90037-9. URL `http://www.sciencedirect.com/science/article/pii/0022000080900379`.

Dan Edward Willard. A Density Control Algorithm for Doing Insertions and Deletions in a Sequentially Ordered File in a Good Worst-Case Time. *Information and Computation*, 97(2):150–204, 1992.

John William Joseph Williams. Algorithm 232: Heapsort. *Communications of the ACM*, 7(6):347–348, 1964.

Glossary

activation — The action of updating the directory in order to make it possible to locate a newly placed routing hunk. See Subsection 3.7.4.

array — Space used for the storage of some elements, which is maintained by an OFM algorithm. In the final data structure, the individual elements are routing hunks, with empty positions filled by filling hunks. See Subsection 1.2.1.

associated — Every routing chunk is associated with the four filling chunks of its group. See Section 2.1.

B — The size of a block in cache. See Section 0.3.

$\mathcal{B}$ — See "layer of buckets".

band — A (possibly empty) set of all the buckets of one of $\mathcal{O}(\log N)$ consecutive sizes. See Subsection 4.4.1.

bucket — The logical entity that $\mathcal{B}$ works with. All entries are either in a bucket or in the preamble. In $\mathcal{D}$, every bucket is represented by a group, the routing chunk of which serves to identify the bucket. See Section 2.1 and 3.3.

cache-aware model — A model simulating the behaviour of an algorithm on a machine with automatically managed cache with known parameters. See Section 0.3.

cache-oblivious model — A model simulating the behaviour of an algorithm on a machine with automatically managed cache with unknown parameters. See Section 0.3.

chunk — A logical entity consisting of a hunk and two spairs. Every chunk is a member of a group and is either routing or filling. See Section 2.1 and 3.3.

containing — The bucket responsible for the creation of the group. The routing chunk of the group is used to identify the bucket and the bucket handles insertions and deletions for the group. See Section 2.1 and 3.3.

cycle — A sequence of $\mathcal{O}\left(\log^3 N\right)$ steps that result from a bucket being split or merged. See Section 4.3.

$\mathcal{D}$ — See "layer of districts".

deactivation — The action of updating the directory in order for it to stop directing searches to a routing hunk that is going to be relocated soon. See Subsection 3.7.4.

density — The number of elements (routing hunks in the final data structure) stored in the relevant part of the data structure divided by the maximal number of such elements that could be stored. See Subsection 1.2.1.

directory — The vEB-permutation of the search tree that serves to facilitate bucket identification. See Section 3.5.

dislocated — Two spairs are dislocated from every chunk, with the remaining hunk encoding a pointer to each of them. See Section 2.3.

district — The entirety of $\mathcal{D}$ is divided into $\mathcal{O}(\log N)$ districts, $\mathcal{F}$ and the index. Districts are gradually created and dismantled as needed by $\mathcal{F}$, and serve to allow the OFM algorithm and the directory to be completely constructed whenever they contain a routing hunk. See Section 2.2.

empty — Not containing any routing hunks.

entry — A pair (key, value) that we consider the atomic unit of stored data. We assume that every entry fits perfectly into any single cell of memory. See Section 0.1.

external memory model — A mode simulating the behaviour of an algorithm on a machine with manually managed loading of blocks from a harddrive. See Section 0.3.

$\mathcal{F}$ — See "filling zone".

F&G — The original paper by Franceschini and Grossi (2003).

filling — A chunk or hunk that is not routing. Filling hunks of a group form a linked list behind the routing hunk. A filling hunk encodes a pointer to each of its two spairs and to the following filling hunk (if that exists). Filling hunks serve as the larger unit of easily relocated data. Compare with routing hunks and spairs. See Section 2.1 and 3.3.

filling zone — A physical entity located in $\mathcal{D}$ after the last district. The filling zone (also known as $\mathcal{F}$) contains otherwise unneeded spairs and filling buckets, assembling and disassembling districts as needed. See Subsection 3.5.4.

full — Containing the maximal permissible number of routing hunks.

group — A logical entity consisting of one routing and four filling chunks. Every bucket provides a single group to $\mathcal{D}$. See Section 2.1 and 3.3.

hunk — The physical entity that, along with its two associated spairs, forms a chunk. A hunk consists of $\mathcal{O}(\log N)$ entries and encodes $\mathcal{O}(1)$ pointers. See Section 2.1 and 3.3.

index — A physical entity located in $\mathcal{D}$. The index contains the root node of every directory, making the directories shorter. See Subsection 3.5.3.

key — An element of a universum with defined total order used to uniquely identify the entry within the data structure. See Section 0.1.

layer — The entire final data structure (a portion in the deamortized version) is divided into the preamble and two layers: layer of buckets and layer of districts. See Chapter 2.

layer of buckets — The layer of buckets contains a large number of buckets. The layer of buckets (also known as $\mathcal{B}$) takes care of operations within the relevant bucket, and initiates bucket splits and merges as needed. See Chapter 2.

layer of districts — The layer of district contains a group of chunks for every bucket in $\mathcal{B}$. The layer of districts (also known as $\mathcal{D}$) takes care of identifying the relevant bucket during an operation, and processes the group insertions and deletions that result from bucket splits and merges. See Chapter 2.

list labeling — The task of assigning labels of polynomial size to a dynamic set of elements and then updating the labels as needed for the order of labels to match the order specified by the "InsertAfter" operations. See Section 1.2.

logical entity — A construct within the data structure that is mainly conceptual in nature and whose contents can be scattered across memory. Compare with physical entities.

memory — Unlike array, this term always means the underlying sequence of cells that each fit a single entry. Do not confuse with operation memory.

n — The number of entries currently stored in the data structure.

N — The maximal number of elements that can be stored in the data structure.

OFM — See "ordered file maintenance".

operation memory — Additional memory used for the duration of an operation. Represents the variables allocated on stack.

ordered file maintenance — List labeling restricted to labels taken from $\mathcal{O}(N)$. Used by us to maintain elements physically in order with $\mathcal{O}(N)$ memory overhead. See Section 1.2.

p — The sequential number of the portion, counting from zero. The p^{th} portion has $N \geq 2^{2^p}$ and uses 2^p-bit pointers and hunks of lenght 2^p. See Section 4.1.

page — A physical entity of $\mathcal{O}(\log N)$ hunks that forms the shortest interval of the array that is managed by OFM. See Subsection 1.2.1.

page overview — A physical entity consisting of one spair for every filling hunk and a pair of entries from every routing hunk stored in that page. The overview enables us to locate the relevant hunk in $\mathcal{O}(\log N)$ time. Used as a replacement of the last $\mathcal{O}(\log \log N)$ levels of the directory while allowing us to store hunks out of order within a page. See Section 4.2.

physical entity — A construct within the data structure that consists of a single range of memory, using all the cells in a closed interval of memory adresses. Compare with logical entities.

portion — A physical entity. Memory used by the data structure is divided into $\mathcal{O}(\log \log n)$ portions. The p^{th} portion contains 2^{2^p} entries, with the exception of the last portion, which might be smaller. See Section 4.1.

preamble — $\mathcal{O}(\log N)$ entries stored apart from the rest of the data structure, which are used to encode important auxiliary data. See Section 2.4.

routing — The minimal chunk or hunk within a group. A routing chunk serves to identify the containing bucket. Every routing hunk encodes a pointer to each of its two spairs, to its containing bucket, and to the first of its four associated filling hunks. See Section 2.1 and 3.3.

spair — Every chunk contains two of these physical entities. A pair of entries that is stored separately from the hunk. Used as the smaller unit of easily relocated data. Compare with filling hunks. See Section 2.3.

step — A part of the process of conducting a group insertion or extraction that is completed in $\mathcal{O}(\log N)$ time. An atomic part of a cycle. See Section 4.3

term — A word or a phrase the meaning of which can be found in this glossary. Terms are emphasized when they first occur in the text.

value — Arbitrary data stored with a given key. See Section 0.1.

van Emde Boas permutation — A recursively defined encoding of a complete binary search tree, that allows for cache-oblivious search in $\mathcal{O}(\log_B N + 1)$ block transfers. See Section 1.1.