A Cache-Oblivious Implicit Dictionary with the Working Set Property

Gerth Stølting Brodal, Casper Kejlberg-Rasmussen, Jakob Truelsen

MADALGO*, Department of Computer Science, Aarhus University, Denmark.
{gerth,jakobt,ckr}@madalgo.au.dk

Abstract. In this paper we present an implicit dictionary with the working set property i.e. a dictionary supporting insert(e), delete(x) and predecessor(x) in $\mathcal{O}(\log n)$ time and search(x) in $\mathcal{O}(\log \ell)$ time, where n is the number of elements stored in the dictionary and ℓ is the number of distinct elements searched for since the element with key x was last searched for. The dictionary stores the elements in an array of size n using *no* additional space. In the cache-oblivious model the operations insert(e), delete(x) and predecessor(x) cause $\mathcal{O}(\log_B n)$ cache-misses and search(x) causes $\mathcal{O}(\log_B \ell)$ cache-misses.

1 Introduction

In this paper we consider the problem of creating an *implicit dictionary* [9] with the *working set property*. An implicit dictionary maintains a set of n distinct keys, and encodes a data structure supporting fast insertions, deletions, predecessor queries and searches in the permutation of these keys as they are laid out in an array [9]. Between operations *no* additional space usage is allowed, while during an operation only a constant number of word registers may be used. The number of elements n is assumed externally maintained. Computation is done in a machine with a constant number of registers with a word size of $\Theta(\log n)$ bits. All operations are unit cost, similar to the RAM model. Extensive research has been done in the implicit/in-place model, from as early as binary heaps [11], to an in-place 3-d convex hull algorithm [4]. Implicit dictionaries have been the topic of several papers culminating in a dictionary supporting all operations in $\mathcal{O}(\log n)$ time [5]. For a more extensive overview see [8].

The working set property states that the time to search for an element e with key x must be $\mathcal{O}(\log \ell)$, where ℓ is the number of distinct elements searched for since e was last searched for. This property has been achieved by numerous structures. The splay tree [10], a skip list variant [2], and the working set structure [7], all achieve the property in the amortized, expected or worst-case sense. The unified access bound, which is a generalization of the working set bound, is achieved in [1]. The unified access bound states that, if $\ell(g)$ is the

* Center for Massive Data Algorithmics, a Center of the Danish National Research Foundation.

Reference	Insert/Delete	Search	Predecessor	Additional space (words)
[5]	$\mathcal{O}(\log n)$	$\mathcal{O}(\log n)$	$\mathcal{O}(\log n)$	None
[7]	$\mathcal{O}(\log n)$	$\mathcal{O}(\log \ell)$	$\mathcal{O}(\log \ell)$	$\mathcal{O}(n)$
[3, Sec. 2]	$\mathcal{O}(\log n)$	$\mathcal{O}(\log \ell)$ exp.	$\mathcal{O}(\log n)$	$\mathcal{O}(\log \log n)$
[3, Sec. 3]	$\mathcal{O}(\log n)$	$\mathcal{O}(\log \ell)$ exp.	$\mathcal{O}(\log \ell)$ exp.	$\mathcal{O}(\sqrt{n})$
This paper	$\mathcal{O}(\log n)$	$\mathcal{O}(\log \ell)$	$\mathcal{O}(\log n)$	None

Table 1. The operation time, and space overhead of important structures for the dictionary problem.

number of distinct elements accessed since g was last accessed, and $d(g, e)$ denotes the rank distance between g and e, then the search time for e must be $\mathcal{O}(\min_g \log(\ell(g) + d(g, e) + 2))$. In [3] two structures with low space overhead are presented, achieving the working set property in the expected sense, see Table 1.

The dictionary in [5] is, in addition to being implicit, also designed for the cache-oblivious model [6] where all the operations imply $\mathcal{O}(\log_B n)$ cache-misses, where B is the cache-line length that is unknown to the algorithm.

1.1 Our results

We present an implicit dictionary with the working set property that supports insertions, deletions, and predecessor queries in $\mathcal{O}(\log n)$ time and search queries in $\mathcal{O}(\log \ell)$ time. Our result improves the construction of [3, Section 2] by requiring no additional space. Furthermore our structure is cache-oblivious and supports insert, delete and predecessor operations in $\mathcal{O}(\log_B n)$ cache-misses and search in $\mathcal{O}(\log_B \ell)$ cache misses.

Our implicit cache-oblivious dictionary makes essential use of the notion of an implicit *moveable* dictionary, i.e. a dictionary stored in a consecutive subarray that can be moved to the left or the right, one position at a time. We construct a moveable dictionary from a constant number of the implicit and cache-oblivious dictionaries from [5], achieving a dictionary inheriting the same properties, but which is also moveable. The moveable dictionary is in itself an interesting result because it is a general transformation, that can be applied to any data structure that can be laid out in an array and grows/shrinks in one end and supporting insertions and deletions. Hence we can plug in say a binary heap, and get a moveable binary heap.

In the literature the working set property is often stated in terms of the number of operations. We note that if we perform a search for an element whenever it is inserted, we will also satisfy these kinds of bounds.

This paper is organized as follows. In Section 2 we present our implicit moveable dictionary. In Section 3 we show how our implicit working set dictionary structure is constructed by composing $\mathcal{O}(\log \log n)$ implicit moveable dictionaries.

2 A moveable dictionary

In this section we describe an implicit moveable dictionary which can be laid out in an array in the range $[i; j]$, where $n = j - i + 1$ is the number of elements in the dictionary. When deleting an element from the dictionary we are allowed to shrink the dictionary from the left or the right end, such that the structure now lies in the range $[i + 1; j]$ or $[i; j - 1]$, respectively. Likewise we can insert and expand the dictionary at the left or right end such that the structure now lies in the range $[i - 1; j]$ or $[i; j + 1]$, respectively. The structure also supports search and predecessor operations. All operations run in $\mathcal{O}(\log n)$ time. The moveable dictionary is implicit except for $\mathcal{O}(\log n)$ extra bits that need to be stored/encoded externally (in the D_i structures in Section 3).

The dictionary supports the following operations:

- Insert-left(e) and insert-right(e): inserts an element e into the dictionary which grows in the left and right side, respectively.
- Delete-left(x) and delete-right(x): deletes the element with key x from the dictionary which shrinks in the left and right side, respectively.
- Search(x): returns the element e with key x in the dictionary if such an element exits, otherwise **none** is returned.
- Predecessor(x): is given a key x and returns the element e in the dictionary with the largest key less than x.

An amortized solution can be obtained using two of the dictionaries by Franceschini and Grossi [5] (in the following denoted FG dictionaries). Let r be an index in the range $i \leq r \leq j$. One FG dictionary denoted R is located in the range $[r; j]$ and grows to the right as normal, and one FG dictionary denoted L is located in the range $[i; r - 1]$ and grows to the left, i.e. for L we have inverted all the indexes of the original FG dictionary. The insert-left and insert-right operations insert elements into L and and R, respectively. The delete-left operation searches for the element e to be deleted in L and R. If e is in L it is deleted from L and we are done. Otherwise e is deleted from R and an arbitrary element is deleted from L and inserted into R – provided L is non-empty. If L is empty we first rebuild the data structure such that L and R differ in size by at most one, by repeatedly reinserting into new L and R structures starting from the new index $r = \lceil \frac{i+j}{2} \rceil$. The delete-right operation is handled symmetrically. To search for an element with a given key we search in L and then in R; to find the predecessor element of a given key we find the predecessor in L and R and return the largest of the two. Since [5] supports all operations in $\mathcal{O}(\log n)$ time, all operations run in $\mathcal{O}(\log n)$ amortized time, which e.g. can be seen using the potential function $\Phi = \left| |L| - |R| \right|$.

In the following we describe how to deamortize the above construction using incremental rebalancing of L and R. An additional FG dictionary C is placed between L and R (see Figure 1). In the following we w.l.o.g. assume that $n \geq 24$, such that all intervals stated below are guaranteed to include an integer. If L or R get outside the range $[\frac{3}{24}n, \frac{7}{24}n]$, say L is getting too big/small, we initialize an incremental *job* to make L smaller/bigger by transferring elements to/from C.

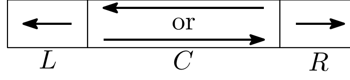

Fig. 1. We have three FG dictionaries L, C and R, where L always grows/shrinks in the left direction, and R grows/shrinks in the right, and C will change direction during the execution of the jobs to shrink or grow L or R.

Each time an insert and delete operation is executed we perform a constant number of steps of the current job. While resizing L there might be a pending job waiting for resizing R, and vice versa. During the execution of a job we have a temporary FG dictionary, which can be one of either L', C' or R', depending on how far we are in the execution of the job (see Figure 2).

2.1 Methods and jobs

The insert-left and delete-left operations, and the grow-left and shrink-left jobs described here have analogous right-versions.

Search(x) We always have the structures L, C and R, and possibly one of the structures L', C' or R'. We search each of the at most four structures. If we find an element e with key x we return e, otherwise we return **none**.

Predecessor(x) As in search we search for the predecessor in each of the structures L, C, R and possibly one of L', C' or R', and return the largest of the four candidates found.

Insert-left(e) We insert e into L. If $|L| > \frac{7}{24}n$ we initialize a shrink-left job unless a left job is already running/pending.

Delete-left(x) We delete the element with key x from L. We can do this even though the element we want to delete resides in L', C, C', R or R' by swapping the element we want to delete with one from L. We can swap elements by performing two deletions and two insertions. If $|L| < \frac{3}{24}n$ we initialize a grow-left job unless a left job is already running/pending.

Grow-left The job consists of the following steps to be performed incrementally (see Figure 2 (left)). Notice that during the incremental work, deletions and insertions are performed on L and R by the update operations. We let n_{init} denote the size of the dictionary when the job is initialized, and assume that n_{init} is remembered when the job is initialized.

1) If C is not growing to the left then turn C around so it grows toward L. We turn C around by creating a new C' in the growing end of C which grows towards C, into which we insert all the elements of C, one element at a time.
2) Construct L' of size $\lceil \frac{2}{24}n_{\text{init}} \rceil$ at the beginning of L, growing to the right, by deleting elements from C and inserting them into L'.

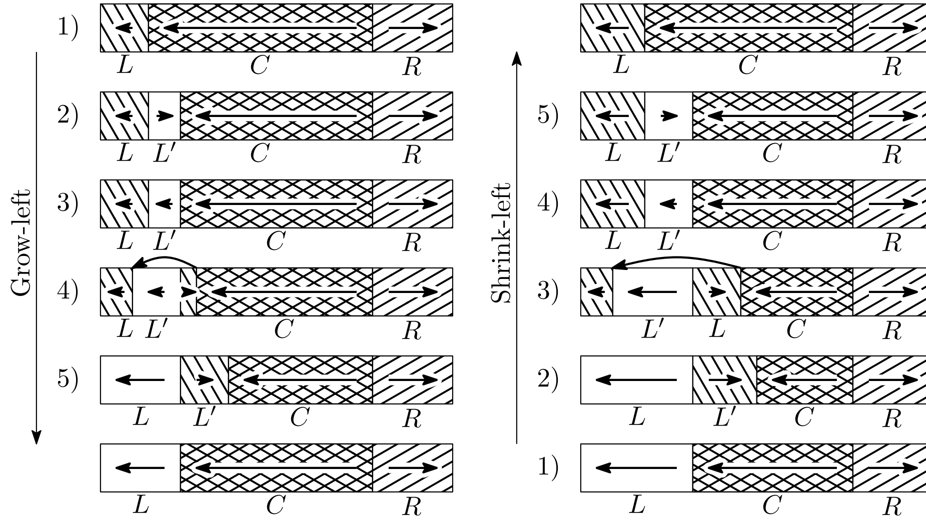

Fig. 2. The steps of the two operations grow-left and shrink-left, notice that they are almost each other's inverse. (Left) The five steps of the grow-left operation, notice that in step 4) the arrow at the top means that we have split L up into two by use of address-mapping. (Right) The five steps of the shrink-left operation, in step 3) we have again used address-mapping to split L in two.

3) Turn L' around so it faces L, like we turned C in step 1).
4) Continue deleting an element from C and inserting it into L', so L' expands into L. The element overridden in L is moved into the empty place in C where we took the element to place in L'. We do this by splitting L into two pieces by address-mapping, see steps 3) and 4) in Figure 2 (left). When we have moved L completely to the right of L', we swap the names of L and L'.
5) Merge L' back into C, by deleting an element from L' and inserting it into C until L' is empty.

Shrink-left The job consists of the following steps (see Figure 2 (right)). Notice the similarity to grow-left.

1) If C is not growing to the left then turn C around so it grows toward L.
2) Create L' by deleting $\lceil \frac{5}{24} n_{\text{init}} \rceil$ elements from C, one element at a time and inserting them into L', which we create to the left of C.
3) Swap the names of L and L'. Delete an element from L' and insert it into C so it expands into L, then move the element overridden in L to the empty space to the left of L', do this one element at a time until L is moved completely to the left of L'.
4) Turn L' around so it faces C.
5) Merge L' back into C.

2.2 Correctness

The correctness of the search and predecessor operations follows directly from the fact that the dictionary consists of at most four FG dictionaries. Similarly the insert-left and insert-right operations insert a single new element into an FG dictionary and otherwise only moves elements between the FG dictionaries. The only operations remaining to be considered are the delete-left and delete-right operations. In the following we only consider the delete-left operation (delete-right is symmetric). The only technical detail we need to argue about is that there always is a non-empty FG dictionary L oriented to the left that has its leftmost element stored in the leftmost entry in the subarray.

In the following when considering a job, we let n_{init}, n_0, n_{finish} denote the size of the moveable dictionary: when the job was initialized, when the execution of the job started, and just after it is finished, respectively.

By performing the incremental work sufficiently fast, we will be able to perform the job during at most βn_0 moveable dictionary updates, for any constant $\beta > 0$. An upper bound on the number of primitive steps (that is movement of one element from one FG dictionary to another one, and possibly move in memory) per update is: During the execution of the job at most βn_0 insertions can take place, i.e. the dictionary always has size at most $(1 + \beta)n_0$. Therefor each of the five steps of a job require at most $(1 + \beta)n_0$ primitive steps. In total there are at most $5(1 + \beta)n_0$ primitive steps. By performing at least $5(1 + \beta)/\beta$ primitive steps per update, the job finishes within βn_0 updates.

To relate n_{init} and n_0 we make the observation that any job under execution will finish during the next βn updates, where n is the current number of elements in the dictionary. To see this, observe that a job that has run for d updates needs to be executed for at most $\beta n_0 - d \leq \beta(n_0 - d) \leq \beta n$ further updates, provided $\beta \leq 1$. From this it follows that when a job is initialized, it at most takes βn_{init} updates before the current job finishes and the new job starts being executed, i.e. $(1 - \beta)n_{\text{init}} \leq n_0 \leq (1 + \beta)n_{\text{init}}$.

Let t_{finish} denote the number of updates between the initialization of a job until it it is finished. We have $t_{\text{finish}} \leq \beta n_{\text{init}} + \beta n_0 \leq \beta n_{\text{init}} + \beta(1 + \beta)n_{\text{init}} = (\beta^2 + 2\beta)n_{\text{init}}$. We get $n_{\text{finish}} \leq n_{\text{init}} + t_{\text{finish}} \leq (1 + \beta^2 + 2\beta)n_{\text{init}}$ and $n_{\text{finish}} \geq n_{\text{init}} - t_{\text{finish}} \geq (1 - \beta^2 - 2\beta)n_{\text{init}}$.

During the lifetime of a job, i.e. between its initialization and its the time it finish, there are always at least $\frac{3}{24}n_{\text{init}} - t_{\text{finish}} \geq (\frac{3}{24} - \beta^2 - 2\beta)n_{\text{init}}$ elements that still can be deleted from the leftmost FG dictionaries which shrink the subarray from the left. By selecting β sufficiently small such that $\beta^2 + 2\beta < \frac{3}{24}$, this number is always non-zero.

What remains to be argued is that *i*) $\frac{3}{24}n_{\text{finish}} \leq |L| \leq \frac{7}{24}n_{\text{finish}}$ when a left job is finished, *ii*) $|C| \geq \lceil \frac{2}{24}n_{\text{init}} \rceil$ when a grow job starts its execution, and *iii*) $|L| \geq \lceil \frac{5}{24}n_{\text{init}} \rceil$ immediately before step 3) in a shrink job. We need *i*) to ensure that $\frac{3}{24}n \leq |L| \leq \frac{7}{24}n$ holds just before a job is initialized, and *ii*) and *iii*) to ensure that grow-left and shrink-left are well defined, respectively.

The above can be shown by the following observations: *i*) After a shrink or grow job $|L| \leq \frac{5}{24}n_{\text{init}} + 1 + t_{\text{finish}} \leq \frac{6}{24}n_{\text{init}} + t_{\text{finish}}$ which is less than $\frac{7}{24}n_{\text{finish}}$

for $\beta^2 + 2\beta \leq \frac{1}{31}$. Similarly after a shrink or grow job $|L| \geq \frac{5}{24} n_{\text{init}} - t_{\text{finish}}$ which is greater than $\frac{3}{24} n_{\text{finish}}$ for $\beta^2 + 2\beta \leq \frac{2}{27}$. *ii)* Before grow-left $|C| \geq n_0 - |L| - |R| \geq (n_{\text{init}} - \beta n_{\text{init}}) - (\frac{7}{24} n_{\text{init}} + \beta n_{\text{init}}) - \frac{7}{24} n_{\text{init}}(1+\beta)$ which is greater than $\frac{3}{24} n_{\text{init}} \geq \lceil \frac{2}{24} n_{\text{init}} \rceil$ for $\beta \leq \frac{7}{55}$. *iii)* In shrink-left $|L| \geq \frac{7}{24} n_{\text{init}} - t_{\text{finish}}$ which is greater than $\frac{6}{24} n_{\text{init}} \geq \lceil \frac{5}{24} n_{\text{init}} \rceil$ for $\beta^2 + 2\beta \leq \frac{1}{24}$. We note that setting $\beta = \frac{1}{63}$ will satisfy all the stated constraints.

The $\mathcal{O}(\log n)$ time bounds for the operations follow from the $\mathcal{O}(\log n)$ time bounds of the FG dictionaries. In the cache-oblivious model we notice that because the FG dictionary is cache-oblivious and we only use a constant number of FG dictionaries, where we split at most one of them into two parts by address-mapping then we only multiply the bound on the cache-misses from the FG dictionary by a constant factor. Hence all operations cause $\mathcal{O}(\log_B n)$ cache-misses.

We notice that we can make the moveable dictionary implicit such that we do not need to store $\mathcal{O}(\log n)$ bits between operations. We do this by introducing a block D of $\mathcal{O}(\log n)$ elements to the left of L which *pair-encodes* the $\mathcal{O}(\log n)$ bits. With pair-encoding we mean that each consecutive pair of elements encodes a bit. If the key of the first element is lower than the key of the second, the pair-encodes a 0 bit. If on the other hand the key of the first element is greater than the key of the second, the pair-encodes a 1 bit. As we need to read this block to get the $\mathcal{O}(\log n)$ bits, we can maintain (and possibly move) D when we perform insert-left, insert-right, delete-left and delete-right operations. From a cache-oblivious viewpoint this does also not change the asymptotic bound on the number of cache-misses.

3 Construction of the working set dictionary

In the following we describe our working set dictionary archiving insertions, deletions and predecessor searches in $\mathcal{O}(\log n)$ time and searches in $\mathcal{O}(\log \ell)$. We first describe the overall structure leaving the details of the memory layout to be handled in Section 3.3. The structure is composed of $\mathcal{O}(\log \log n)$ blocks, where the i'th block B_i stores $\mathcal{O}(2^{2^i})$ elements. The main design goal is to have elements that have been searched for within the last ℓ distinct searches located in one of the first $\mathcal{O}(\log \log \ell)$ blocks.

Block B_i consists of a list D_i of size w_i where $w_i = \alpha 2^i$ for some appropriate constant α, and three implicit moveable dictionaries, L_i, C_i and R_i. We use D_i to pair-encode $\mathcal{O}(2^i)$ bits, used for memory management in the working set dictionary and storing data needed between operations in the moveable dictionaries L_i, C_i and R_i. Block B_i contains exactly $2 \cdot 2^{2^i} + w_i$ elements, except for the last block B_m that might contain less than $2 \cdot 2^{2^i} + w_i$ elements, as this is the block that grows or shrinks when we insert or delete, respectively.

When an element e is searched for it is moved from its current block B_j to the first block B_0. To make room for this in B_0, we move an element from each block B_i to B_{i+1} until we reach the block B_j where e was originally located. We move elements from R_i to L_{i+1}, for $i = 0, \ldots, j-1$ (see Figure 3). Once R_i

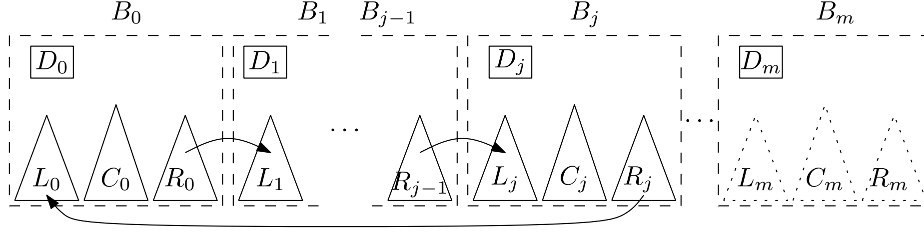

Fig. 3. Layout of the data structure. The arrows indicate the movement of elements after an element in R_j has been searched for. The dotted lines in block B_m indicate that the strucutres do not necessarily exist.

is empty we move C_i to R_i, and L_i to C_i. Doing this we can guarantee that at least 2^{2^i} distinct elements have been searched for since any element in R_i was last searched for. We can give this guarantee because an element will be located in C_i at least until searches for 2^{2^i} other elements have been performed.

3.1 Invariants

Our data structure satisfies the invariants below. Here I.1 to I.4 are about the sizes of data structures and are important for memory management. On the other hand I.5 to I.8 are about the location of elements according to when they were last searched for and are important for achieving the working set property.

I.1 $|C_i| \leq 2^{2^i}$ and $|R_i| \neq 0 \Rightarrow |C_i| = 2^{2^i}$, for all i.
I.2 $|D_i| \leq w_i$ and $|L_i| + |C_i| + |R_i| \neq 0 \Rightarrow |D_i| = w_i$, for all i.
I.3 $|L_i| + |R_i| = 2^{2^i}$, for all $i < m$, and $|L_m| + |R_m| \leq 2^{2^m}$.
I.4 $|L_i| < 2^{2^i}$, for all i.
I.5 All elements searched for since L_i was last empty are contained in L_i, D_i or B_j for some $j < i$.
I.6 For any e in some C_i either at least $|L_i|$ distinct elements have been searched for after e was last searched for or e has never been searched for.
I.7 For any e in some R_i either at least 2^{2^i} distinct elements have been searched for after e was last searched for or e has never been searched for.
I.8 For any e in D_i, L_i or C_i, for $i > 0$, either at least $2^{2^{i-1}}$ distinct elements have been searched for after e was last searched for, or e has never been searched for.

From the invariants we make the following observations:

O.1 $|D_i| = w_i$ for all $i < m$ (from I.2 and I.3).
O.2 $|R_i| > 0$ for all $i < m$ (from I.3 and I.4).
O.3 $|C_i| = 2^{2^i}$ for all $i < m$ (from I.1 and O.2).
O.4 $|B_i| = w_i + 2 \cdot 2^{2^i}$ for all $i < m$ (from O.1, O.3, and I.3).
O.5 For $i > 0$ and any e in B_i, either at least $2^{2^{i-1}}$ distinct elements have been searched for after e was last searched for or e has never been searched for (from I.7 and I.8).

3.2 Operations

Our data structure uses the operations shift and find internally, and supports the operations insert, delete, predecessor and search. Below is a detailed description of all operations.

Shift(j) handles the case when $|R_j| = 0$ and $|L_j| = 2^{2^j}$, i.e. I.4 is violated for block B_j. This is done by discarding R_j, renaming C_j to R_j, renaming L_j to C_j, and creating a new empty L_j. After shift(j) finishes I.4 also holds for B_j.

Find(x) finds the data structure S_i containing the element with key x or returns **none** if no such element exists. Here S_i will be either D_i, L_i, C_i or R_i for some i. This is done by searching for x in the blocks starting with B_0 and going in an incremental linear fashion towards B_m. Within each block, x is searched for in D_i using a linear scan, and the implicit moveable dictionaries L_i, C_i and R_i are searched for x using their built-in search operation. As soon as x is found, a reference to the data structure S_i containing the element is returned, and no further blocks are considered. In the case when x is not found in any of the blocks **none** is returned.

Predecessor(x) returns the element e in the data structure with the largest key less than x. This is done for $B_0, \ldots, B_m$ by a linear scan of D_i and invoking the built-in predecessor operation on L_i, C_i and R_i and returning the element among the results with the highest key.

Insert(e) inserts the element e into the data structure. This is done by inserting e into one of the data structures in B_m. It is inserted into D_m if $|D_m| < w_m$. Otherwise, if $|C_m| < 2^{2^m}$ it is inserted into C_m, else it is inserted into R_m. If this makes $|L_m| + |R_m| = 2^{2^m}$, then a new block B_{m+1} is initialized by incrementing m by one.

Delete(x) deletes the element with key x from the data structure. We first check if x is in the dictionary by performing a find(x) operation. If x is not found we return. Here S_j will be one of D_j, L_j, C_j or R_j. If B_m is empty, m is decremented by one. An arbitrary element e is deleted from the first of the structures R_m, C_m, L_m and D_m that is non-empty. If e has key x we return, else the element with key x is deleted from S_j and e is inserted into S_j.

Search(x) returns the element e with key x or **none** if such an element does not exist. This is done by performing a find(x) operation, finding the data structure S_j containing x. If x is not in the data structure then **none** is returned.

If x is found in a data structure S_j then the element e with key x is found by running the built-in search method on S_j. If S_j is either D_0 or L_0 we return e immediately. If $S_j = C_j$ and $|R_j| > 0$, an arbitrary element g is removed from R_j, e is removed from C_j and g is inserted in C_j. In the other case where $S_j \neq C_j$ or $|R_j| = 0$, the element e with key x is deleted from S_j.

In all cases we then proceed by deleting an arbitrary element h from R_{i-1} and inserting it into L_i, for $i = j, \ldots, 1$. In the special case where $i = j$ and $S_j = D_j$ we insert h into D_j instead of L_j.

We then insert e into L_0. Now for $i = 0, \ldots, j$ we check whether $|L_i| = 2^{2^i}$, and if this is the case we perform a shift(i) operation. Finally we return e.

3.3 Memory management

From O.4 we know that any block except the last will contain a fixed number of elements, namely $2 \cdot 2^{2^i} + w_i$. This implies that we can lay out the blocks sequentially in the array, and then we only have to worry about memory management inside each block. The last block B_m can vary in size, and is located at the end of the array where growing and shrinking must occur.

By I.2 we know that D_i will be completely constructed before the other structures are needed, therefore we lay it out sequentially in the beginning of the block. The remaining structures will be laid out sequentially in the order: L_i, C_i, R_i.

Right before we insert an element into L_i, we move C_i and R_i one position to the right to make room. We can move L_i, C_i or R_i to the right by performing a delete-left operation on an arbitrary element e followed by an insert-right(e). This moving will take time $\mathcal{O}(2^i)$. We do the same when inserting into C_i, but here we only move R_i one position to the right. We never need to move structures to the left.

To perform queries on the substructures in a block B_i we need to store various information in D_i. We need n_{L_i}, n_{C_i} and n_{R_i}: the size of L_i, C_i and R_i, respectively. We store n_{C_i} and n_{R_i} in D_i explicitly using 2^i bits each, whereas n_{L_i} can be computed as $n_{L_i} = |B_i| - w_i - n_{C_i} - n_{R_i}$. Furthermore we store in D_i the $\Theta(2^i)$ bits we allow the moveable dictionaries L_i, C_i and R_i to maintain between operations, denoted $data_{L_i}$, $data_{C_i}$ and $data_{R_i}$.

We maintain all these bits in D_i, using pair-encoding. The fields are stored in the following order: $data_{L_i}$, n_{C_i}, $data_{C_i}$, n_{R_i}, $data_{R_i}$. Whenever we add an element to or remove an element from D_i we maintain the ordering of the pair by performing a swap if needed.

To perform an operation on block B_i we need to know the index b_i of the first element, which can be computed as $b_0 = 0$, and $b_i = b_{i-1} + 2 \cdot 2^{2^{i-1}} + w_{i-1}$. We may also need $|D_i|$ which can be computed as $|D_i| = \min(w_i, n - b_i)$, and $|B_i|$ which can be computed as $|B_i| = w_i + 2 \cdot 2^{2^i}$ if $i < m$ and $|B_m| = n - b_m$ otherwise.

Whenever we want to perform an operation on C_i, we first extract n_{L_i}, n_{C_i}, n_{R_i} and $data_{C_i}$ from the pair-encoding in D_i and put them into registers. From the sizes and the value of b_i we can compute the index of the first element in C. Using that information we can run the operation on the implicit moveable dictionary. Once that is done we $data_C$ write back to the pair-encoding in D_i. Totally this requires $\mathcal{O}(2^i)$ time. We do similarly if we perform an operation on L_i or R_i.

When performing a shift operation we override n_{R_i} and $data_{R_i}$ with n_{C_i} and $data_{C_i}$ and we override n_{C_i} and $data_{C_i}$ with n_{L_i} and $data_{L_i}$. This renames the data structures, initiating a new empty L_i before the old full one, and "deletes" the old empty R_i.

During an insert operation, when D_i increases to w_i, we initialize n_{L_i}, $data_{L_i}$, n_{C_i}, $data_{C_i}$, n_{R_i} and $data_{R_i}$. Finally we calculate m when it is needed as the minimal value where $\sum_{j=0}^{m} 2 \cdot 2^{2^j} + w_j > n$.

3.4 Analysis

To see that the invariants are maintained for the operations we need to show that each invariant is maintained for each operation. In general this is tedious but trivial. As an example below we give the proof for the shift operation to give a taste of how the proofs go. In the following S_i refers to a data structure before the shift operation and S'_i refers to the same data structure after the operation, similarly for m.

We now prove that shift is correct. We assume that B_j satisfies all invariants except I.4 before shift(j). Since the shift operation requires that $|L_j| = 2^{2^j}$ and $C'_j = L_j$ I.1 holds for j after the shift operation. Because $|L_j| = 2^{2^j}$ and I.2 holds before the operation we know that $|D_j| = w_j$. Since the shift operation did not change D_j, I.2 also holds for j after the operation. When verifying I.3 we have two cases: if $j = m$, then from I.1 we know that $|C_m| \leq 2^{2^m}$ so $|L'_m| + |R'_m| = 0 + |C_m| \leq 2^{2^m}$ so I.3 holds. Else if $j < m$ then by O.3 we know that $|C_j| = 2^{2^j}$. Now since L'_j is empty and $R'_j = C_j$, then I.3 holds for $j < m$ after the operation. Since L'_j is empty, no elements have been accessed after it was last empty thus I.5 trivially holds for j. Likewise because $\left|L'_j\right| = 0$, then I.6 is maintained for j. Because shift(j) assumed $|L_j| = 2^{2^j}$, and because $R'_j = C_j$, then I.6 immediately implies that I.7 holds for j after shift(j). Lastly, since all elements in L'_j and C'_j come from L_j, and D'_j contains the same elements as D_j then I.8 held for j since it holds before the shift(j) operation.

The core of the analysis of the running times of the predecessor and search operations stems from the find operation. Let ℓ be the number of distinct elements searched for since we last searched for some element e in some block B_j. By O.5 we know that at least $2^{2^{j-1}}$ elements have been searched for after e was last searched for so $\ell \geq 2^{2^{j-1}}$, i.e. $2^j = \mathcal{O}(\log \ell)$. For each block we use constant time to calculate b_i, $|D_i|$, and whether $i = m$. This can be done since we have already computed b_{i-1} once b_i is needed. The time used for the find operation in block B_i is $\mathcal{O}(2^i)$, plus the time for doing the linear scan in D_i, and $\mathcal{O}(\log 2^{2^i}) = \mathcal{O}(2^i)$ for doing searches in L_i, C_i and R_i from the bounds on the moveable dictionary. The total time for doing searches in all blocks $0, \ldots, j$ is then $\mathcal{O}(\sum_{i=0}^{j} 2^i) = \mathcal{O}(2^j) = \mathcal{O}(\log \ell)$ which becomes the time for the search operation. Since predecessor queries need to access all blocks, they require $\mathcal{O}(\log n)$ time. Similarly insertions and deletions require $\mathcal{O}(\log n)$ time.

From the cache-oblivious viewpoint we incur $\mathcal{O}(2^i/B)$ cache-misses when searching D_i and $\mathcal{O}(\log_B 2^{2^i}) = \mathcal{O}(2^i/\log B)$ cache-misses when searching in

L_i, C_i and R_i. Since the blocks $B_0, \ldots, B_{\lfloor \log \log B \rfloor}$ in total store $\mathcal{O}(B)$ elements and are stored consecutively in the array, the accesses to these blocks imply a total of $\mathcal{O}(1)$ cache-misses. For the remaining blocks $2^{2^i} \geq B$ and in total we incur $\mathcal{O}(\sum_{i=\lfloor \log \log B \rfloor + 1}^{j} 2^i / \log B) = \mathcal{O}(\log_B \ell)$ cache-misses for the find operation. It follows that search implies $\mathcal{O}(\log_B \ell)$ cache-misses, and predecessor queries, insert and delete operations imply $\mathcal{O}(\log_B n)$ cache-misses.

Acknowledgements

We would like to thank Mark Greve and Freek van Walderveen for their help on proofreading this paper.

References

1. Bădoiu, M., Cole, R., Demaine, E.D., Iacono, J.: A unified access bound on comparison-based dynamic dictionaries. Theoretical Computer Science 382(2), 86–96 (2007)
2. Bose, P., Douïeb, K., Langerman, S.: Dynamic optimality for skip lists and B-trees. In: Proc. 19th Annual ACM-SIAM Symposium on Discrete algorithms. pp. 1106–1114. SIAM, Philadelphia, PA, USA (2008)
3. Bose, P., Howat, J., Morin, P.: A distribution-sensitive dictionary with low space overhead. In: Proc. 11th International Symposium on Algorithms and Data Structures. LNCS, vol. 5664, pp. 110–118. Springer-Verlag (2009)
4. Chan, T.M.Y., Chen, E.Y.: Optimal in-place algorithms for 3-d convex hulls and 2-d segment intersection. In: Proc. 25th Annual Symposium on Computational Geometry. pp. 80–87. ACM (2009)
5. Franceschini, G., Grossi, R.: Optimal worst-case operations for implicit cache-oblivious search trees. In: Proc. 8th International Workshop on Algorithms and Data Structures. LNCS, vol. 2748, pp. 114–126. Springer-Verlag (2003)
6. Frigo, M., Leiserson, C.E., Prokop, H., Ramachandran, S.: Cache-oblivious algorithms. In: Proc. 40th Annual IEEE Symposium on Foundations of Computer Science. pp. 285–297. IEEE (1999)
7. Iacono, J.: Alternatives to splay trees with $\mathcal{O}(log(n))$ worst-case access times. In: Proc. 12th Annual ACM-SIAM symposium on Discrete algorithms. pp. 516–522. SIAM (2001)
8. Mortensen, C.W., Pettie, S.: The complexity of implicit and space-efficient priority queues. In: Proc. 9th Biennial Workshop on Algorithms and Data Structures. LNCS, vol. 3608, pp. 49–60. Springer-Verlag (2005)
9. Munro, J.I., Suwanda, H.: Implicit data structures for fast search and update. Journal of Computer and System Sciences 21(2), 236–250 (1980)
10. Sleator, D.D., Tarjan, R.E.: Self-adjusting binary search trees. J. ACM 32(3), 652–686 (1985)
11. Williams, J.W.J.: Algorithm 232: Heapsort. Communications of the ACM 7(6), 347–348 (1964)