Cache-Oblivious Implicit Predecessor Dictionaries with the Working-Set Property
Gerth Stølting Brodal, Casper Kejlberg-Rasmussen

To cite this version:

HAL Id: hal-00678181
https://hal.archives-ouvertes.fr/hal-00678181
Submitted on 3 Feb 2012

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L’archive ouverte pluridisciplinaire HAL, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d’enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.
Cache-Oblivious Implicit Predecessor Dictionaries with the Working-Set Property*

Gerth Stølting Brodal† and Casper Kejlberg-Rasmussen†

1 MADALGO†, Department of Computer Science, Aarhus University, Denmark

1 Abstract

In this paper we present an implicit dynamic dictionary with the working-set property, supporting insert(e) and delete(e) in \(O(\log n)\) time, predecessor(e) in \(O(\log \ell_p(e))\) time, successor(e) in \(O(\log \ell_s(e))\) time and search(e) in \(O(\min(\ell_p(e), \ell_s(e)))\) time, where \(n\) is the number of elements stored in the dictionary, \(\ell_p\) is the number of distinct elements searched for since element \(e\) was last searched for and \(p(e)\) and \(s(e)\) are the predecessor and successor of \(e\), respectively. The time-bounds are all worst-case. The dictionary stores the elements in an array of size \(n\) using no additional space. In the cache-oblivious model the \(\log\) is base \(B\) and the cache-obliviousness is due to our black box use of an existing cache-oblivious implicit dictionary. This is the first implicit dictionary supporting predecessor and successor searches in the working-set bound. Previous implicit structures required \(O(\log n)\) time.

1998 ACM Subject Classification Algorithms and data structures, E.1 Data Structures

Keywords and phrases working-set property, dictionary, implicit, cache-oblivious, worst-case, external memory, I/O efficient

Digital Object Identifier 10.4230/LIPIcs.STACS.2012.112

1 Introduction

In this paper we consider the problem of maintaining a cache-oblivious implicit dictionary [11] with the working-set property over a dynamically changing set \(P\) of \(|P| = n\) distinct and totally ordered elements. We define the working-set number of an element \(e \in P\) to be \(\ell_e = \{e' \in P \mid \text{we have searched for } e' \text{ after we last searched for } e\}\). An implicit dictionary maintains \(n\) distinct keys without using any other space than that of the \(n\) keys, i.e. the data structure is encoded by permuting the \(n\) elements. The fundamental trick in the implicit model, [10], is to encode a bit using two distinct elements \(x\) and \(y\): if \(\min(x,y)\) is before \(\max(x,y)\) then \(x\) and \(y\) encode a 0 bit, else they encode a 1 bit. This can then be used to encode \(l\) bits using \(2^l\) elements. The implicit model is a restricted version of the unit cost RAM model with a word size of \(O(\log n)\). The restrictions are that between operations we are only allowed to use an array of the \(n\) input elements to store our data structures by permuting the input elements, i.e., there can be used no additional space between operations. In operations we are allowed to use \(O(1)\) extra words. Furthermore we assume that the number of elements \(n\) in the dictionary is externally maintained. Our structure will support the following operations:

- **Search(e)** determines if \(e\) is in the dictionary, if so its working-set number is set to 0.
- **Predecessor(e)** will find max\(\{e' \in P \cup \{-\infty\} \mid e' < e\}\), without changing any working-set numbers.

* This is an extended abstract, the full paper is available at http://arxiv.org/abs/1112.5472
† Center for Massive Data Algorithmics, a Center of the Danish National Research Foundation
The dictionary in [6] is, in addition to being implicit, also designed for the cache-oblivious model [8], where all the operations imply $O(\log_B n)$ cache-misses. Here $B$ is the cache-line length which is unknown to the algorithm. The cache-oblivious property also carries over into our dictionary. Our structure combines the two worlds of implicit dictionaries and dictionaries with the working-set property to obtain the first implicit dictionary with the working-set property supporting search, predecessor and successor queries in the working-set bound. The result of this paper is summarized in Theorem 1.

**Theorem 1.** There exists a cache-oblivious implicit dynamic dictionary with the working-set property that supports the operations insert and delete in time $O(\log n)$ and $O(\log_B n)$ cache-misses, search, predecessor and successor in time $O(\log \min(t_{p(e)}, e, t_{s(e)}))$, $O(\log t_{p(e)})$ and $O(\log s(e))$, and cache-misses $O(\log_B \min(t_{p(e)}, e, t_{s(e)}))$, $O(\log_B t_{p(e)})$ and $O(\log_B t_{s(e)})$.

| Ref. | WS prop. | Insert/ Delete(e) | Search(e) | Pred(e)/ Succ(e) | Additional
|------|----------|------------------|-----------|-----------------| words |
| [10] | $O(\log^2 n)$ | $O(\log^2 n)$ | $O(\log^2 n)$ | None |
| [5] | $O(\log n)$ | $O(\log n)$ | $O(\log n)$ | None |
| [7] | $O(\log n)$ | $O(\log n)$ | $O(\log n)$ | None |
| [6] | $O(\log n)$ | $O(\log n)$ | $O(\log n)$ | None |
| [9] | $O(\log n)$ | $O(\log \ell_e)$ | $O(\log \ell_e)$ | $O(n)$ |
| [3, Sec. 2] | $O(\log n)$ | $O(\log \ell_e)$ exp. | $O(\log \ell_e)$ | $O(\log \log n)$ |
| [3, Sec. 3] | $O(\log n)$ | $O(\log \ell_e)$ exp. | $O(\log \ell_e)$ | $O(\sqrt{n})$ |
| [4] | $O(\log n)$ | $O(\log \ell_e)$ | $O(\log \ell_e)$ | None |
| This paper | $O(\log n)$ | $O(\log \min(t_{p(e)}, e, t_{s(e)}))$ | $O(\log \ell_e)$ | None |

Table 1 The operation time and space overhead of important structures for the dictionary problem. Here $e^*$ is the predecessor or successor in the given context. In a search for an element $e$ that is not present in the dictionary $\ell_e$ is $n$.

- **Successor(e)** will find $\min\{e' \in P \cup \{\infty\} \mid e < e'\}$, without changing any working-set numbers.
- **Insert(e)** inserts $e$ into the dictionary with at working-set number of 0, all other working-set numbers are increased by one.
- **Delete(e)** deletes $e$ from the dictionary, and does not change the working-set number of any element.

There has been a continuous development of implicit dictionaries, the first milestone was the implicit AVL-tree [10] having bounds of $O(\log^2 n)$. The second milestone was the implicit B-tree [5] having bounds of $O(\log^2 n/\log \log n)$ the third was the flat implicit tree [7] obtaining $O(\log n)$ worst-case time for searching and amortized bounds for updates. The fourth milestone is the optimal implicit dictionary [6] obtaining worst-case $O(\log n)$ for search, update, predecessor and successor.

Numerous non-implicit dictionaries attain the working-set property: splay trees [12], skip list variants [2], the working-set structure in [9], and two structures presented in [3]. All achieve the property in the amortized, expected or worst-case sense. The unified access bound, which is achieved in [1], even combines the working-set property with finger search.

In finger search we have a finger located on an element $f$ and the search cost of finding say element $e$ is a function of $d(f, e)$ which is the rank distance between elements $f$ and $e$. The unified bound combines these two to obtain a bound of $O(\min_{e \in P}\{log(\ell_e + d(e, f) + 2))\}$.

This paper gives an overview of previous results, and our contribution.

The dictionary in [6] is, in addition to being implicit, also designed for the cache-oblivious model [8], where all the operations imply $O(\log_B n)$ cache-misses. Here $B$ is the cache-line length which is unknown to the algorithm. The cache-oblivious property also carries over into our dictionary. Our structure combines the two worlds of implicit dictionaries and dictionaries with the working-set property to obtain the first implicit dictionary with the working-set property supporting search, predecessor and successor queries in the working-set bound. The result of this paper is summarized in Theorem 1.
respectively, where \( p(e) \) and \( s(e) \) are the predecessor and successor of \( e \), respectively.

Similarly to previous work \([1, 4]\) we partition the dictionary elements into \( \mathcal{O}(\log \log n) \) blocks \( B_0, \ldots, B_m \), of double exponential increasing sizes, where \( B_0 \) stores the most recently accessed elements. The structure in \([4]\) supports predecessors and successors queries, but there is no way of knowing if an element is actually the predecessor or successor, without querying all blocks, which results in \( \mathcal{O}(\log n) \) time bounds. We solve this problem by introducing the notion of intervals and particularly a dynamic implicit representation of these. We represent the whole interval \([\min(P); \max(P)]\) by a set of disjoint intervals spread across the different blocks. Any point that intersects an interval in block \( B_i \) will lie in block \( B_i \) and have a working-set number of at least \( 2^i \). This way when we search for the predecessor or successor of an element and hit an interval, then no more points can be contained in the interval in higher blocks, and we can avoid looking at these, which give working-set bounds for the search, predecessor and successor queries.

2 Data structure

We now describe our data structure and its invariants. We will use the moveable dictionary from \([4]\) as a black box. The dictionary over a point set \( S \) is laid out in the memory addresses \([i; j]\). It supports the following operations in \( \mathcal{O}(\log n') \) time and \( \mathcal{O}(\log_B n') \) cache-misses, where \( n' = j - i + 1 \):

- **Insert-left** \((e) \) inserts \( e \) into \( S \) which is now laid out in the addresses \([i - 1; j]\).
- **Insert-right** \((e) \) inserts \( e \) into \( S \) which is now laid out in the addresses \([i; j + 1]\).
- **Delete-left** \((e) \) deletes \( e \) from \( S \) which is now laid out in the addresses \([i + 1; j]\).
- **Delete-right** \((e) \) deletes \( e \) from \( S \) which is now laid out in the addresses \([i; j - 1]\).
- **Search** \((e) \) determines if \( e \in S \), if so the address of element \( e \) is returned.
- **Predecessor** \((e) \) returns the address of the element \( \max\{e' \in S \mid e' < e\} \) or that no such element exists.
- **Successor** \((e) \) returns the address of the element \( \min\{e' \in S \mid e < e'\} \) or that no such element exists.

From these operations we notice that we can move the moveable dictionary, say left, by performing a delete-right operation for an arbitrary element and re-inserting the element again by an insert-left operation. Similarly we can also move the dictionary one position to the right.

Our structure consists of \( m = \Theta(\log \log n) \) blocks \( B_0, \ldots, B_m \), each block \( B_i \) is of size \( \mathcal{O}(2^{i+k}) \), where \( k \) is a constant. Elements in \( B_i \) have a working-set number of at least \( 2^{i+k-1} \). The block \( B_i \) consists of an array \( D_i \) of \( w_i = d \cdot 2^{i+k} \) elements, where \( d \) is a constant, and moveable dictionaries \( A_i, R_i, W_i, H_i, C_i \) and \( G_i \), for \( i = 0, \ldots, m - 1 \), see Figure 1. For block \( B_m \) we only have \( D_m \) if \( |B_m \setminus \{\min(P), \max(P)\}| \leq w_m \), otherwise we have the same structures as for the other blocks. We use the block \( D_i \) to encode the sizes of the movable dictionaries \( A_i, R_i, W_i, H_i, C_i \) and \( G_i \) so that we can locate them. Discussion of further details of the memory layout is postponed to Section 3.

We call elements in the structures \( D_i \) and \( A_i \) for **arriving** points, and when making a non-arriving point arriving, we will put it into \( A_i \) unless specified otherwise. We call elements in \( R_i \) for **resting** points, elements in \( W_i \) for **waiting** points, elements in \( H_i \) for **helping** points, elements in \( C_i \) for **climbing** points and elements in \( G_i \) for **guarding** points.

Crucial to our data structure is the partitioning of \([\min(P); \max(P)]\) into intervals. Each interval is assigned to a **level** and level \( i \) corresponds to block \( B_i \). Consider an interval lying at level \( i \). The endpoints \( e_1 \) and \( e_2 \) will be guarding points stored at level \( 0, \ldots, i \).
All points inside of this interval will lie in level $i$ and cannot be guarding points, i.e. $\|e_1; e_2\| \cap (\bigcup_{j \neq i} B_j \cup G_i) = \emptyset$. We do not allow intervals defined by two consecutive guarding points to be empty, they must contain at least one non-guarding point. We also require $\min(P)$ and $\max(P)$ to be guarding points in $G_0$ at level 0, but they are special as they do not define intervals to their left and right, respectively. A query considers $B_0, B_1, \ldots$ until $B_i$ where the query is found to be in a level $i$ interval where the answer is guaranteed to have been found in blocks $B_0, \ldots, B_i$.

The basic idea of our construction is the following. When searching for an element it is ready to go to level $i$, i.e. there is at least one non-guarding point. We also require $\min(P)$ and $\max(P)$ to be guarding points in $G_0$ at level 0, but they are special as they do not define intervals to their left and right, respectively. A query considers $B_0, B_1, \ldots$ until $B_i$ where the query is found to be in a level $i$ interval where the answer is guaranteed to have been found in blocks $B_0, \ldots, B_i$.

Before we introduce the invariants we need to define some notation. For a subset $S \subseteq P$, we define $p_S(e) = \max\{s \in S \cup \{-\infty\} \mid s < e\}$ and $s_S(e) = \min\{s \in S \cup \{\infty\} \mid e < s\}$. When we write $S_{\leq i}$ we mean $\bigcup_{j=0}^{i} S_j$ where $S_j \subseteq P$ for $j = 0, \ldots, i$.

For $S \subseteq P$, we define $\mathrm{GL}_S(e) = S \cap [p_{P \setminus S}(e); e]$ to be the Group of Immediate Left points of $e$ in $S$ which does not have any other point of $P \setminus S$ in between them. Similarly we define $\mathrm{GR}_S(e) = S \cap [s_{P \setminus S}(e); e]$ to the right of $e$. We will notice that we will never find all points of $\mathrm{GL}_S(e)$ unless $|\mathrm{GL}_S(e)| < c$, the same applies for $\mathrm{GR}_S(e)$. For $S \subseteq P$, we define $\mathrm{FGL}_S(e) = S \cap [p_{P \setminus S}(p_S(e)); p_S(e)]$ to be the First Group of points from $S$ Left of $e$, i.e. the group does not have any points of $P \setminus S$ in between its points. Similarly we define $\mathrm{FGR}_S(e) = S \cap [s_{P \setminus S}(s_S(e)); s_{P \setminus S}(e)]$. We will notice that we will never find all points of $\mathrm{FGL}_S(e)$ unless $|\mathrm{FGL}_S(e)| < c$, the same applies for $\mathrm{FGR}_S(e)$.

We will sometimes use the phrasings a group of points or $e$’s group of points. This refers to a group of points of the same type, i.e. arriving, resting, etc., and with no other types of points in between them. Later we will need to move elements around between the structures $D_i, A_i, R_i, W_i, H_i, C_i$ and $G_i$. For this we have the notation $X \xrightarrow{h} Y$, meaning that we move $h$ arbitrary points from $X$ into $Y$, where $X$ and $Y$ can be one of $D_i, A_i, R_i, W_i, H_i, C_i$ and $G_i$ for any $i$. 

---

**Figure 1** Overview of how the working set dictionary is laid out in memory. The dictionary grows and shrinks to the right when elements are inserted and deleted.
When we describe the intervals we let \([a; b]\) be an interval from \(a\) to \(b\) that is open at \(a\) and closed at \(b\). We let \((a; b)\) be an interval from \(a\) to \(b\) that can be open or closed at \(a\) and \(b\). We use this notation when we do not care if \(a\) and \(b\) are open or closed. In the methods updating the intervals we will sometimes branch depending on which type an interval is. For clarity we will explain how to determine this given the level \(i\) of the interval and its two endpoints \(e_1\) and \(e_2\). The interval \((e_1; e_2)\) is of type \([e_1; e_2]\) if \(e_1 \in G_i\), else \(e_1 \in G_{i-1}\) and the interval is of type \([e_1; e_2]\). This is symmetric for the other endpoint \(e_2\).

### 2.2 Invariants

We will now define the invariants which will help us define and prove correctness of our interface operations: insert\((e)\), delete\((e)\), search\((e)\), predecessor\((e)\) and successor\((e)\). We maintain the following invariants which uniquely determine the intervals\(^1\):

\begin{enumerate}
  \item A guarding point is part of the definition of at most two intervals\(^2\), one to the left at level \(i\) and/or one to the right at level \(j\), where \(i \neq j\). The guarding point \(e\) lies at level \(\min(i,j)\). The interval at level \(\min(i,j)\) is closed at \(e\), and the interval at level \(\max(i,j)\) is open at \(e\). We also require that \(\min(P)\) and \(\max(P)\) are guarding points stored in \(G_0\), but they do not define an interval to their left and right, respectively, and the intervals they help define are open in the end they define. A non-guarding point intersecting an interval at level \(i\), lies in level \(i\). Each interval contains at least one non-guarding point. The union of all intervals give \(|\min(P); \max(P)|\).
  \item Any climbing point, which lies in an interval with other non-climbing points, is part of a group of at least \(c\) points. In intervals of type \([e_1; e_2]\) which only contain climbing points, we allow there to be less than \(c\) of them.
  \item Any helping point is part of a group of size at most \(c - 1\). A helping point cannot have a climbing point as a predecessor or successor. An interval of type \([e_1; e_2]\) cannot contain only helping points.
\end{enumerate}

We maintain the following invariants for the working-set numbers:

\begin{enumerate}
  \item Each arriving point in \(D_1\) and \(A_1\) has a working set value of at least \(2^{2^{i-1+k}}\), arriving points in \(D_0\) and \(A_0\) have a working-set value of at least \(0\). Each resting point in \(R_i\) will have a working-set value of at least \(2^{2^{i-1+k}} + |A_i|\), resting points in \(R_0\) have a working-set value of at least \(|A_0|\). Each waiting, helping or climbing point in \(W_i, H_i\) and \(C_i\), respectively, will have a working-set value of at least \(2^{2^{i+k}}\). Each guarding point in \(G_i\), who’s left interval lies at level \(i\) and right interval lies at level \(j\), has a working set value of at least \(2^{2^{\max(i,j)-1+k}}\).
\end{enumerate}

We maintain the following invariants for the size of each block and their components:

\begin{enumerate}
  \item \(|D_0| = \min(|B_0| - 2, w_0)|\) and \(|D_i| = \min(|B_i|, w_i)\) for \(i = 1, \ldots, m\).
  \item \(|R_i| \leq 2^{2^{i+k}}\) and \(|W_i| + |H_i| + |C_i| \neq 0 \Rightarrow |R_i| = 2^{2^{i+k}}\) for \(i = 0, \ldots, m\).
  \item \(|A_i| + |W_i| = 2^{2^{i+k}}\) for \(i = 0, \ldots, m - 1\), and \(|A_m| + |W_m| \leq 2^{2^{m+k}}\).
  \item \(|A_i| \leq 2^{2^{i+k}}\) for \(i = 0, \ldots, m\).
  \item \(|H_i| + |C_i| = 4e2^{2^{i+k}} + c_i\), where \(c_i \in [-c; c]\), for \(i = 0, \ldots, m - 1\).
\end{enumerate}

\(^1\) We assume that \(|P| = n \geq 2\) at all times if this is not the case we only store \(G_0\) which contains a single element and we ignore all invariants.

\(^2\) Only the smallest and largest guarding points will not participate in the definition of two intervals, all other guarding points will.
From the above invariants we have the following observation:

O.1 From I.1 all points in $G_i$ are endpoints of intervals in level $i$, and each interval has at most two endpoints. Hence for $i = 0, \ldots, m$ we have that

$$|G_i| \leq 2(3|D_i| + |A_i| + |R_i| + |W_i| + |H_i| + |C_i|) \leq (4 + 2d + 8c) \cdot 2^{2^{i+1}} + 2c,$$

where in $(\ast)$ we have used I.5, I.6, I.7 and I.9.

From I.1 we have the following lemma.

Lemma 1. Let $e$ be an element, $e_1 = \text{pre}_{G \subseteq i}, (e)$, $e_2 = \text{seg}_{\leq j}, (e)$ and $i$ be the smallest integer for which $I(e_1, e_2, i) \in G_i$ or $\bigcup_{j=0}^i B_j \neq \emptyset$. Then (1) $(e_1; e_2)$ is an interval at level $i$ if $e$ is non-guarding and 2) $(e_1; e)$ or $(e; e_2)$ is an interval at level $i$ if $e$ is guarding.

2.3 Operations

We will briefly give an overview of the helper operations and state their requirements $(R)$ and guarantees $(G)$, then we will describe the helper and interface operations in details. Search$(e)$ uses the helper operations as follows: when a search for element $e$ is performed then the level $i$ where $e$ lies is found using find, then $e$ and $O(1)$ of its surrounding elements are moved into level 0 by use of move-down while maintaining I.1–I.4. Calls to fix for the levels we have altered will ensure that I.5–I.8 will be maintained, finally a call to rebalance-below$(i - 1)$ will ensure that I.9 is maintained by use of shift-up$(j)$ which will take climbing points from level $j$ and make them arriving in level $j + 1$ for $j = 0, \ldots, i - 1$. Insert$(e)$ uses find to find the level where $e$ intersects, then it uses fix to ensure the size constraints and finally $e$ is moved to level 0 by use of search.

- **Find**$ (e)$ - returns the level $i$ of the interval that $e$ intersects along with $e$'s type and whatever $e$ is in the dictionary or not. \([\mathcal{R} \& \mathcal{G}: I.1–I.9]\)

- **Fix**$ (i)$ - moves points around inside of $B_i$ to ensure the size invariants for each type of point. Fix$ (i)$ might violate I.9 for level $i$. \([\mathcal{R}: I.1–I.4 \text{ and that there exist } \hat{c}_1, \ldots, \hat{c}_6 \text{ such that } |D_i| + \hat{c}_1, |A_i| + \hat{c}_2, |R_i| + \hat{c}_3, |W_i| + \hat{c}_4, |H_i| + \hat{c}_5, |C_i| + \hat{c}_6 \text{ fulfill I.5–I.8, where } |\hat{c}_i| = \mathcal{O}(1) \text{ for } i = 1, \ldots, 6. \text{ \(G: I.1–I.8\).}]

- **Shift-down**$ (i)$ - will move at least 1 and at most $c$ points from level $i$ into level $i - 1$. \([\mathcal{R}: I.1–I.8 \text{ and } |H_i| + |C_i| = 4e2^{2^{i+1}} + c', \text{ where } 0 \leq c' = \mathcal{O}(1). \text{ \(G: I.1–I.8\).}]

- **Shift-up**$ (i)$ - will move at least 1 and at most $c$ points from level $i$ into level $i + 1$. \([\mathcal{R}: I.1–I.8 \text{ and } |H_i| + |C_i| = 4e2^{2^{i+1}} + c', \text{ where } c \leq c' = \mathcal{O}(1). \text{ \(G: I.1–I.8\).}]

- **Move-down**$ (e, i, j, t_{\text{before}}, t_{\text{after}})$ - if $e$ is in the dictionary at level $i$ it is moved from level $i$ to level $j$, where $i \geq j$. The type $t_{\text{before}}$ is the type of $e$ before the move and $t_{\text{after}}$ is the type that $e$ should have after the move, unless $i = j$ in which case $e$ will be made arriving in level $j$. \([\mathcal{R} \& \mathcal{G}: I.1–I.8]\).

- **Rebalance-below**$ (i)$ - If any $c < c_l$ for $l = 0, \ldots, i$ rebalance-below$(i)$ will correct it so I.9 will be fulfilled again for $l = 0, \ldots, i$. \([\mathcal{R}: I.1–I.8 \text{ and } \sum_{i=0}^j \text{slack}(c_i) = \mathcal{O}(1), \text{ where } \text{slack}(c_i) = \begin{cases} 0 & \text{if } c_i \in [-c; c], \\ |c_i| - c & \text{otherwise.} \end{cases} \text{ \(G: I.1–I.9\).}]

- **Rebalance-above**$ (i)$ - If any $c_i < -c$ for $l = i, \ldots, m - 1$ rebalance-above$(i)$ will correct it so I.9 will be fulfilled again for $l = i, \ldots, m - 1$. \([\mathcal{R}: I.1–I.8 \text{ and } \sum_{l=i}^{m-1} \text{slack}(c_l) = \mathcal{O}(1). \text{ \(G: I.1–I.9\).}]

STACS’12
Find($e$) We start at level $i = 0$. If $e < \min(P)$ or $\max(P) < e$ we return false and 0. For each level we let $e_1 = p_{G_{e_1}}(e)$, $e_2 = s_{G_{e_2}}(e)$. $p$ is $p_{B_{e_1}}(e)$ and $s$ is $s_{B_{e_2}}(e)$. We find $p$ and $s$ by querying each of the structures $D_i$, $A_i$, $R_i$, $W_i$, $H_i$, and $C_i$, we find $e_1$ and $e_2$ by querying $G_i$ and comparing with the values of $e_1$ and $e_2$ from level $i - 1$. While $p < e_1$ and $e_2 < s$ we continue to the next level, that is we increment $i$. Now outside the loop, if $e \in B_i$ we return $i$, the type of $e$ and the boolean true as we found $e$, else we return $i$ and false as we did not find $e$.

Predecessor($e$) (successor($e$)) We start at level $i = 0$. If $e < \min(P)$ then return $-\infty$ ($\min(P)$). If $\max(P) < e$ then return $\max(P)$ ($\infty$). For each level we let $e_1 = p_{G_{e_1}}(e)$, $p = p_{B_{e_1}}(e)$, $e_2 = s_{G_{e_2}}(e)$ and $s = s_{B_{e_2}}(e)$. While $p < e_1$ and $e_2 < s$ we continue to the next level, that is we increment $i$. When the loop breaks we return $\max(e_1, p)$ ($\min(s, e_2)$).

Insert($e$) If $e < \min(P)$ we swap $e$ and $\min(P)$, call fix(0), rebalance-below($m$) and return. If $\max(P) < e$ we swap $e$ and $\max(P)$, call fix(0), rebalance-below($m$) and return.

Let $c_1 = GL_{C_1}(e)$, $c_r = GR_{C_1}(e)$, $h_l = GL_{H_l}(e)$ and $h_r = GR_{H_l}(e)$. We find the level $i$ of the interval $(e_1; e_2)$ which $e$ intersects using find($e$).

If $e$ is already in the dictionary we give an error. If $|c_l| > 0$ or $|c_r| > 0$ or $(e_1; e_2)$ is of type $[e_1; e_2]$ and does not contain non-climbing points then insert $e$ as climbing at level $i$. Else if $|h_l| + 1 + |h_r| \geq c$ then insert $e$ as climbing at level $i$ and make the points in $h_l$ and $h_r$ climbing at level $i$. Else insert $e$ as helping at level $i$. Finally we call rebalance-below($m$) and then search($e$) to move $e$ from the current level $i$ down to level 0.

Search($e$) We first find $e$'s current level $i$ and its type $t$, by a call to find($e$). If $e$ is in the dictionary then we call move-down($e$, $i$, $0$, $t$, arriving) which will move $e$ from level $i$ down to level 0 and make it arriving, while maintaining 1.1–1.8, but 1.9 might be broken so we finally call rebalance-below($i - 1$) to fix this.

Fix($i$) In the following we will be moving elements around between $D_i$, $A_i$, $R_i$, $W_i$, $H_i$ and $C_i$. The moves $A_i \rightarrow R_i$ and $R_i \rightarrow W_i$, i.e. between structures which are next to each other in the memory layout, are simply performed by deleting an element from the left structure and inserting it into the right structure. The moves $W_i \rightarrow H_i \cup C_i$ and the other way around $H_i \cup C_i \rightarrow W_i$ will be explained below.

If $|D_i| > w_i$ then perform $D_i \xrightarrow{h} A_i$ where $h = |D_i| - w_i$. If $|D_i| < w_i$ and $|B_i| \{\min(P), \max(P)\} > |D_i|$ then perform $H_i \cup C_i \xrightarrow{h} W_i$, $W_i \xrightarrow{h} R_i$, $R_i \xrightarrow{h} A_i$ and $A_i \xrightarrow{h} D_i$ where $h_1 = \min(w_i - |D_i|, |H_i| + |C_i|)$, $h_2 = \min(w_i - |D_i|, |W_i| + h_1)$, $h_3 = \min(w_i - |D_i|, |R_i| + h_2)$ and $h_4 = \min(w_i - |D_i|, |A_i| + h_3)$.

If $|W_i| + |H_i| + |C_i| \neq 0$ and $|R_i| < 2^{2^{i+k}}$ then perform $H_i \cup C_i \xrightarrow{h} W_i$ and $W_i \xrightarrow{h} R_i$ where $h_1 = \min(2^{2^{i+k}} - |R_i|, |H_i| + |C_i|)$ and $h_2 = \min(2^{2^{i+k}} - |R_i|, |W_i| + h_1)$. If $|R_i| > 2^{2^{i+k}}$ then perform $R_i \xrightarrow{h} A_i$ where $h_1 = |R_i| - 2^{2^{i+k}}$.

If $i < m$ and $|A_i| + |W_i| < 2^{2^{i+k}}$ then perform $H_i \cup C_i \xrightarrow{h} W_i$, where $h_1 = \min(2^{2^{i+k}} - (|A_i| + |W_i|), |H_i| + |C_i|)$. If $|A_i| + |W_i| > 2^{2^{i+k}}$ then perform $W_i \xrightarrow{h} H_i \cup C_i$ where $h_1 = \min(|A_i| + |W_i| - 2^{2^{i+k}}, |W_i|)$.

If $|A_i| \geq 2^{2^{i+k}}$ then let $h_1 = |A_i| - 2^{2^{i+k}}$, delete $W_i$ as it is empty and rename $R_i$ to $W_i$. Now move $h_1$ elements from $A_i$ into a new moveable dictionary $X$, rename $A_i$ to $R_i$, rename $X$ to $A_i$ and perform $W_i \xrightarrow{h} H_i \cup C_i$. 


Performing $W_i \rightarrow H_i \cup C_i$: Let $w = s_{H_i}(-\infty)$, $c_l = \text{GLC}_i(w)$, $c_r = \text{GIRC}_i(w)$, $h_i = \text{GILH}_i(w)$ and $h_r = \text{GIRH}_i(w)$. If $|c_l| > 0$ or $|c_r| > 0$ or $(e_1; e_2)$ is of type $[e_1; e_2]$ and only contains climbing points then make $w$ climbing at level $i$. Else if $|h_i| + |h_r| \geq c$ then make $h_i$, $w$ and $h_r$ climbing at level $i$. Else make $w$ helping at level $i$.

Performing $H_i \cup C_i \rightarrow W_i$: Let $w$ be the minimum element of $s_{H_i}(-\infty)$ and $s_{C_i}(-\infty)$, and let $c_r = \text{GIRC}_i(w)$. Make $w$ waiting at level $i$. If $w$ was climbing and $|c_r| < c$ then make $c_r$ helping at level $i$.

Shift-down($i$) We move at least one element from level $i$ into level $i-1$. If $|D_i| < w_i$ then let $a$ be some element in $D_i$. If $|D_i| < |B_i|$ then: if $|A_i| = 0$ we perform $^3 H_i \cup C_i \rightarrow W_i$, $W_i \rightarrow R_i$ and $R_i \rightarrow A_i$, where $h_1 = \min(1, |H_i| + |C_i|)$ and $h_2 = \min(1, |W_i| + h_1)$, now we know that $|A_i| > 0$ so let $a = s_{A_i}(-\infty)$, i.e., $a$ is the leftmost arriving point in $A_i$ at level $i$. We call move-down($a, i, i-1, \text{arriving, climbing}$).

Shift-up($i$) Assume we are at level $i$, we want to move at least one and at most $c$ arbitrary points from $B_i$ into $B_{i+1}$. Let $s_1 = s_{C_i}(-\infty)$, $e_1 = \text{p}_{G_{\leq i}}(s_1)$ and $e_2 = s_{C_i}(s_1)$, and let $s_2 = s_{C_i\cap[e_1; e_2]}(s_1)$, $s_3 = s_{C_i\cap[e_1; e_2]}(s_2)$, $s_4 = s_{C_i\cap[e_1; e_2]}(s_3)$ and $s_5 = s_{C_i\cap[e_1; e_2]}(s_4)$, if they exist, also let $c_r = \text{GIRC}_i(s_4)$ be the group of climbing elements to the immediate right of $s_4$, if they exist. We will now move one or more climbing points from $B_i$ into $B_{i+1}$ where they become arriving points. If $i = m - 1$ or $i = m$ then we put arriving points into $D_{i+1}$, which we might have to create, instead of $A_{i+1}$.

We now deal with the case where $(e_1; e_2)$ is of type $[e_1; e_2]$ and only contains climbing points. Let $l$ be the level of $e_1$'s left interval, and $r$ the level of $e_2$'s right interval, also let $e_l$ be the number of climbing points in the interval. If $l = i + 1$ we make $e_1$ arriving, else we make it guarding, at level $i + 1$. Make the points of $s_1$, $s_2$, $s_3$ and $s_4$ that exist arriving at level $i + 1$. If $e_l \leq c$ then make $s_5$ arriving at level $i + 1$ if it exists, also if $r = i + 1$ we make $e_2$ arriving, else we make it guarding, at level $i + 1$. Else make $s_5$ guarding at level $i$.

We now deal with the cases where $(e_1; e_2)$ might contain non-climbing points. If $\text{p}(s_1) = e_1$ we make $s_1$ and $s_2$ waiting and guarding at level $i$, respectively, else we make $s_1$ guarding at level $i$ and $s_2$ arriving at level $i + 1$. Now in both cases we make $s_3$ arriving at level $i + 1$ and $s_4$ guarding at level $i$. If $(s_4; e_2)$ is not of type $[s_4; e_2]$ or contains non-climbing points) and $|c_r| < c$, i.e., there are less than $c$ consecutive climbing points to the right of $s_4$, then we make the points $c_r$ helping at level $i$.

We have moved climbing points from $B_i$ into $B_{i+1}$, and made them arriving. Finally we call fix($i + 1$).

Move-down($e, i, j, t_{\text{before}}, t_{\text{after}}$) Depending on the type $t_{\text{before}}$ of point $e$ we have different cases.

Non-guarding Let $e_1 = \text{p}_{G_{\leq i}}(e)$, $e_2 = s_{G_{\leq i}}(e)$ and let $t$ be the level of the left interval of $e_1$ and $r$ the level of the right interval of $e_2$. Also let $p_2 = \text{p}_{B_{i} \setminus G_{\cap[e_1; e_2]}(p_1)}$, $p_1 = \text{p}_{B_{i} \setminus G_{\cap[e_1; e_2]}}(e)$, $s_1 = s_{B_{i} \setminus G_{\cap[e_1; e_2]}}(e)$ and $s_2 = s_{B_{i} \setminus G_{\cap[e_1; e_2]}}(s_1)$, also let $c_l = \text{FGC}_{G_{\cap[e_1; e_2]}(e)}$ be the elements in the first climbing group left of $e$, likewise let $c_r = \text{FGC}_{G_{\cap[e_1; e_2]}(e)}$ be the elements in the first climbing group right of $e$.

$^3$ The move $H_i \cup C_i \rightarrow W_i$ will be performed the same way as we did it in fix.
Case \( i = j \): make \( e \) arriving in level \( j \), if \( |c_l| < c \) then make the points in \( c_l \) helping at level \( j \), if \( |c_r| < c \) then make the points in \( c_r \) helping at level \( j \). Finally call \( \text{fix}(j) \).

Case \( i > j \): If both \( p_2 \) and \( p_1 \) exists we make \( p_1 \) guarding in level \( j \) and let \( e_1' \) denote \( p_1 \), else if only \( p_1 \) exists we make \( e_1 \) guarding at level \( \min(l, j) \) and \( p_1 \) of type \( t_{\text{after}} \) at level \( j \) and let \( e_1' \) denote \( e_1 \), else we make \( e_1 \) guarding in level \( \min(l, j) \), and let \( e_1' \) denote \( e_1 \). If both \( s_1 \) and \( s_2 \) exists we make \( s_1 \) guarding at level \( j \), and let \( e_2' \) denote \( s_1 \), else if only \( s_1 \) exists we make \( s_1 \) of type \( t_{\text{after}} \) at level \( j \) and make \( e_2 \) guarding at level \( \min(j, r) \) and let \( e_2' \) denote \( e_2 \), else we make \( e_2 \) guarding at level \( \min(j, r) \) and let \( e_2' \) denote \( e_2 \). Lastly we make \( e \) of type \( t_{\text{after}} \) in level \( j \). Now let \( c_r' \) denote the elements of \( c_r \) which we have not moved in the previous steps, likewise let \( c_r' \) denote the elements of \( c_r \) which we have not moved. If \( \langle e_1; e_1' \rangle \) is not of type \( |c_1| < c \) or contains non-climbing points) and \( |c_r'| < c \) then make \( c_r' \) helping at level \( i \). If \( \langle e_2; e_2' \rangle \) is not of type \( |e_2| < c \) or contains non-climbing points) and \( |c_r'| < c \) then make \( c_r' \) helping at level \( i \). Call \( \text{fix}(i) \), \( \text{fix}(j) \), \( \text{fix}(\min(l, i)) \) and \( \text{fix}(\min(i, r)) \).

**Guarding** If \( e = \min(P) \) or \( e = \max(P) \) we simply do nothing and return. Let \( e_1 = p_{G_{c_h}}(e) \) be the left endpoint of the left interval \([e_1, e] \) lying at level \( h \) and \( e_2 = s_{G_{c_h}}(e) \) be the right endpoint of the right interval \([e, e_2] \) lying at level \( i \), we assume w.l.o.g. that \( h > i \), the case \( h < i \) is symmetric. Also let \( b \) be the level of the left interval of \( e_1 \) and \( r \) of the level of the right interval of \( e_2 \). Let \( p_2 = p_{B_{h \setminus G_{d_r}[e_1, e]}(p_1)} \) and \( p_1 = p_{B_{h \setminus G_{d_r}[e_1, e]}(e)} \) be the two left points of \( e \), if they exist, \( s_1 = s_{B_{h \setminus G_{d_r}[e_2]}(e)} \) and \( s_2 = s_{B_{h \setminus G_{d_r}[e_2]}(e)} \) the two right points of \( e \), if they exist. Also let \( c_l = G_{\text{FlC}_{C_{i}}[e_1; e]}(e) \) and \( c_r = G_{\text{FlC}_{C_{i}}[e_2]}(e) \).

If \( p_2 \) does not exist then we make \( e_1 \) guarding at level \( \min(l, j) \), we make \( p_1 \) of type \( t_{\text{after}} \) at level \( j \) and let \( e_1' \) denote \( e_1 \), else we make \( p_1 \) guarding at level \( j \) and let \( e_1' \) denote \( p_1 \). If it is the case that \( i > j \) then we check: if \( s_2 \) does not exist then we make \( s_1 \) of type \( t_{\text{after}} \) at level \( j \), \( e_2 \) guarding at level \( \min(j, r) \) and let \( e_2' \) denote \( e_2 \), else we make \( s_1 \) guarding at level \( j \) and let \( e_2' \) denote \( s_1 \). We make \( e \) of type \( t_{\text{after}} \) at level \( j \).

Now let \( c_r' \) be the points of \( c_l \) which was not moved and \( c_r' \) the points of \( c_r \) which was not moved. If \( |c_r'| < c \) then make \( c_r' \) helping at level \( h \). We now have two cases if \( e_2' \) exists: then if \( |c_r'| < c \) then make \( c_r' \) helping at level \( i \). The other case is if \( e_2' \) does not exist: then if \( \langle e_1' e_2 \rangle \) is not of type \( |e_1| < c \) or contains non-climbing points) and \( |c_r'| < c \) then make \( c_r' \) helping at level \( i \). In all cases call \( \text{fix}(\min(l, h)) \), \( \text{fix}(h) \) and \( \text{fix}(i) \). If \( i > j \) then call \( \text{fix}(j) \) and \( \text{fix}(\min(j, r)) \).

**Delete** We first call \( \text{find}(e) \) to get the type of \( e \) and its level \( i \), if \( e \) is not in the dictionary we just return. If \( e \) is in the dictionary we have two cases, depending on if \( e \) is guarding or not.

**Non-guarding** Let \( c_l = G_{\text{FlC}_{C_{i}}[e]} \) be the elements in the climbing group immediately left of \( e \), let \( c_r = G_{\text{FlC}_{C_{i}}[e]} \) be the elements in the climbing group immediately right of \( e \), let \( h_l = G_{\text{FlH}_{i}}(e) \) be the elements in the helping group immediately left of \( e \), and let \( h_r = G_{\text{FlH}_{i}}(e) \) be the elements in the helping group immediately right of \( e \). Let \( e_1 = p_{G_{c_l}}(e) \) and \( e_2 = s_{G_{c_l}}(e) \). Let \( b \) be the level of the interval left of \( e_1 \) and \( r \) the level of the interval right of \( e_2 \).

We have two cases, the first is \( |e_1; e_2| < B_l | = 1 \): if \( l > r \) make \( e_1 \) guarding and \( e_2 \) arriving at level \( r \), if \( l < r \) then make \( e_1 \) arriving and \( e_2 \) guarding at level \( l \). If \( l = r \) and \( |P| = n \geq 4 \) then make \( e_1 \) and \( e_2 \) arriving at level \( l = r \). Delete \( e \), call \( \text{fix}(r) \), \( \text{fix}(l) \), \( \text{fix}(i) \) and \( \text{rebalance-above}(1) \).

The other case is \( |e_1; e_2| > B_l | > 1 \): If \( \langle e_1; e_2 \rangle \) is not of type \( |e_1; e_2| \) or contains non-climbing points) and \( |e_1| + |e_2| < c \) then make \( c_l \) and \( c_r \) helping at level \( i \). If \( |h_l| + |h_r| > c \) then make \( h_l \) and \( h_r \) climbing at level \( i \). Delete \( e \), call \( \text{fix}(i) \) and \( \text{rebalance-above}(1) \).
We will now deal with the memory layout of the data structure. We will put the blocks in the

also the methods shift-down, search, rebalance-below and rebalance-above only calls other
methods, hence their memory management is handled by the methods they call. The only
methods where actual memory management comes into play are in insert, shift-up, fix, move-
down and delete. We will now describe two methods internal-movement – which handles
movement inside a single block/level – and external-movement – which handles movement
across different blocks/levels. Together these two methods handle all memory management.

**Min-guarding** If \( e = \min(P) \) then let \( e' = s_{G_{\leq m}}(e) \) and \( e'' = s_{G_{\leq m}}(e') \) where 0 is the level
of \( (e; e') \) and \( i \) is the level of \( (e'; e'') \). The case of \( e = \max(P) \) is symmetric. Also let
\( s_1 = s_{B_i \cup \mathcal{G}_c \setminus [e; e']}(e), s_2 = s_{B_i \cup \mathcal{G}_c \setminus (e; e'']}(s_1), t_1 = s_{B_i \cup \mathcal{G}_c \setminus (e'; e'']}(e') \) and \( t_2 = s_{B_i \cup \mathcal{G}_c \setminus [e''; e'']}(t_1) \).

If \( s_2 \) exists then delete \( e \) make \( s_1 \) guarding at level 0 and call fix(0). If \( s_2 \) does not exist and \( t_2 \) exists then delete \( e \) make \( s_1 \) and \( t_1 \) guarding and \( e' \) arriving at level 0 and finally call
fix(0) and fix(i). If \( s_2 \) does not exist and \( t_2 \) does not exist then delete \( e \), make \( s_1 \) and \( e'' \)
guarding and \( e' \) and \( t_1 \) arriving at level 0 and finally call fix(0) and fix(i). In all the previous
cases return.

**Guarding** Let \( h \) be the level of the left interval \( (e_1 : e_i] \), let \( i \) the level of the right interval
\( [e : e_2) \) that \( e \) participates in. We assume w.l.o.g. that \( h > i \), the case \( h < i \) is symmetric. Let
\( l \) the level of the left interval that \( e_1 \) participates in, where \( e_1 = p_{G_{\geq h}}(e) \) and \( e_2 = s_{G_{\geq h}}(e) \).
Let \( p_1 = p_{B_i \cup \mathcal{G}_c \setminus [e_1; e_1]}(p_1) \) and \( p_2 = p_{B_i \cup \mathcal{G}_c \setminus [e: e]}(e) \). Let \( c_l = FGL_{C_l}(e) \) be the points in
the first group of climbing points left of \( e \).

If \( p_2 \) exist we make \( p_1 \) guarding at level \( i \), and let \( e' \) denote \( p_1 \), else we make \( e_1 \) guarding at
level \( \min(l, i) \), let \( e' \) denote \( e_1 \) and if \( (e'; e_2) \) is of type \( [e'; e_2] \) and contains only climbing points
then we make \( p_1 \) climbing at level \( i \) else we make \( p_1 \) waiting at level \( i \). Let \( c_i' \) be the points in
\( c_l \) which was not moved in the previous movement of points. If \( |c_i'| < c \) make \( c_i' \) helping at
level \( h \). If \( e' \) is \( e_1 \) then call fix(l). Delete \( e \), call fix(h), fix(i) and rebalance-above(1).

**Rebalance-below(i)** For each level \( l = 0, \ldots , i \) we perform a shift-up(l) while \( e < c_i \).

**Rebalance-above(i)** For each level \( l = i, \ldots , m - 1 \) we perform shift-down(l + 1) while
\( e_l < -c \).

**3 Memory management**

We will now deal with the memory layout of the data structure. We will put the blocks in the
order \( B_{i_0}, \ldots , B_{i_m} \), where block \( B_i \) further has its dictionaries in the order \( D_i, A_i, R_i, W_i, H_i, C_i \)
and \( G_i \), see Figure 1. Block \( B_m \) grows and shrinks to the right when elements are inserted
and deleted from the working set dictionary.

The \( D_i \) structure is not a moveable dictionary as the other structures in a block are, it
is simply an array of \( w_i = d^{2^{i+k}} \) elements which we use to encode the size of each of the
structures \( A_i, R_i, W_i, H_i, C_i \) and \( G_i \) along with their own auxiliary data, as they are not
implicit and need to remember \( \mathcal{O}(2^{i+k}) \) bits which we store here. As each of the moveable
dictionaries in \( B_i \) have size \( \mathcal{O}(2^{2^{i+k}}) \) we need to encode numbers of \( \mathcal{O}(2^{i+k}) \) bits in \( D_i \).

We now describe the memory management concerning the movement, insertion and
deletion of elements from the working-set dictionary. First notice that the methods find,
predecessor and successor do not change the working-set dictionary, and layout in memory.
Also the methods shift-down, search, rebalance-below and rebalance-above only calls other
methods, hence their memory management is handled by the methods they call. The only
methods where actual memory management comes into play are in insert, shift-up, fix, move-
down and delete. We will now describe two methods internal-movement – which handles
movement inside a single block/level – and external-movement – which handles movement
across different blocks/levels. Together these two methods handle all memory management.
Internal-movement($m_1, \ldots, m_l$) Internal-movement in level $i$ takes a list of internal moves $m_1, \ldots, m_l$ to be performed on block $B_i$, where $l = \mathcal{O}(1)$ and move $m_j$ consists of:

- the index $\gamma = D_i, A_i, R_i, W_i, H_i, C_i, G_i$ of the dictionary to change, where we assume\(^4\) that $m_j.\gamma < m_h.\gamma$, for $j < h$,
- the set of elements $S_{in}$ to put into $\gamma$, where $|S_{in}| = \mathcal{O}(1)$,
- the set of elements $S_{out}$ to take out of $\gamma$, where $|S_{out}| = \mathcal{O}(1)$ and
- the total size difference $\Delta = |S_{in}| - |S_{out}|$ of $\gamma$ after the move.

For $j = 1, \ldots, l$ do: if $m_j.\delta < 0$ then remove $S_{out}$ from $\gamma$, insert $S_{in}$ into $\gamma$ and move $\gamma + 1, \ldots, G$ left $m_j.\gamma$ positions, where we move them in the order $\gamma + 1, \ldots, G$. If $m_j.\delta > 0$ then move $\gamma + 1, \ldots, G$ right $m_j.\delta$ positions, where we move them in the order $G, \ldots, \gamma + 1$, remove $S_{out}$ from $\gamma$ and insert $S_{in}$ into $\gamma$. See Figure 2.

It takes $\mathcal{O}(\log(2^{2^i+k})) = \mathcal{O}(2^k)$ time and $\mathcal{O}(\log_B(2^{2^i+k})) = \mathcal{O}(2^k)$ cache-misses to perform move $j$. In total all the moves $m_1, \ldots, m_l$ use $\mathcal{O}(2^k)$ time and $\mathcal{O}(2^k)$ cache-misses, as $l = \mathcal{O}(1)$.

External-movement($M_1, \ldots, M_l$) External-movement takes a list of external moves $M_1, \ldots, M_l$, where $l = \mathcal{O}(1)$. Move $M_j$ consists of:

- the index $0 \leq \gamma \leq m$ of the block/level to perform the internal moves $m_1, \ldots, m_q$ on, where $M_j.\gamma < M_h.\gamma$, for $j < h$,
- the list of internal moves $m_1, \ldots, m_q$ to perform on block $\gamma$, where $q = \mathcal{O}(1)$, and
- the total size difference $\Delta = \sum_{h=1}^q m_h.\delta$ of block $\gamma$ after all the internal moves $m_1, \ldots, m_q$ have been performed.

Let $\overline{\Delta} = \sum_{i=1}^l M_i.\Delta$ be the total size change of the dictionary after the external-moves have been performed. If $\overline{\Delta} = 0$ then we let $\gamma_{end} = M_1.\gamma$ else we let $\gamma_{end} = m$. Let $p_{end} = \sum_{j=0}^{\gamma_{end}} |B_j|$ and $\overline{\Delta}$ be the last address of the right most block that we need to alter. Let $s_1, \ldots, s_k$ be the sublist of the indexes $\{1, \ldots, l\}$ where $M_{a_i}.\Delta \leq 0$ for $i = 1, \ldots, k$. Let $a_1, \ldots, a_h$ be the sublist of the indexes $\{1, \ldots, l\}$ where $M_{a_i}.\Delta > 0$ for $i = 1, \ldots, h$.

We first perform all the internal moves of each of the external moves $M_{s_1}, \ldots, M_{s_k}$. Then we compact all the blocks with index $i$ where $M_{i,\gamma} \leq i \leq \gamma_{end}$ so the rightmost block ends at position $p_{end}$. Finally for each external move $M_{a_i}$ for $i = 1, \ldots, h$: move $B_{M_{a_i},\gamma}$ left so it aligns with $B_{M_{a_i},\gamma-1}$ and perform all the internal moves of $M_{a_i}$, then compact the blocks $B_{M_{a_i},\gamma+1}, \ldots, B_{M_{a_i},\gamma-1}$ at the left end so they align with block $B_{M_{a_i},\gamma}$.

It takes $\mathcal{O}\left(l \log(2^{2^i+k})\right) = \mathcal{O}(2^k)$ time and $\mathcal{O}\left(l \log_B(2^{2^i+k})\right) = \mathcal{O}(2^k)$ cache-misses to perform the internal moves on level $i$. In total all the external moves $M_1, \ldots, M_l$ use

\(^4\) We will misuse notation and let $\gamma + 1$ denote the next in the total order $D, A, R, W, H, C, G$. We will also compare $m_j.\gamma$ and $m_h.\gamma$ with $\leq$ in this order.
$O(2^{\gamma_{\text{end}}+k})$ time and $O\left(\frac{2^{\gamma_{\text{end}}+k}}{\log B}\right)$ cache-misses, as the external move at level $\gamma_{\text{end}}$ dominates the rest and $l = O(1)$.

### 3.1 Memory management in updates of intervals

With the above two methods we can perform the memory management when updating the intervals in Section 2.3: Whenever an element moves around, is deleted or inserted, it is simply put in one or two internal moves. All internal moves in a single block/level are grouped into one external move. Since all updates of intervals only move around a constant number of elements, the requirements for internal/external-movement that $l = O(1)$ and $q = O(1)$ are fulfilled. From the above time and cache bounds for the memory management the bounds in Theorem 1 follows.

---

**References**