Cache-Oblivious Implicit Predecessor Dictionaries with the Working-Set Property

Gerth Stølting Brodal\textsuperscript{1} and Casper Kejlberg-Rasmussen\textsuperscript{1}

\textbf{1} MADALGO*, Department of Computer Science, Aarhus University, Denmark

\begin{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 $O(\log \min(\ell_p(e),\ell_s(e)))$ time, where $n$ is the number of elements stored in the dictionary, $\ell_e$ 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.

\textbf{1998 ACM Subject Classification} Algorithms and data structures, E.1 Data Structures

\textbf{Keywords and phrases} working-set property, dictionary, implicit, cache-oblivious, worst-case, external memory, I/O efficient
\end{abstract}

\section{Introduction}

In this paper we consider the problem of maintaining a cache-oblivious implicit dictionary \cite{Bro00} with the working-set property over a dynamically changing set $P$ of $|P| = n$ distinct and totally ordered elements. We define the \textit{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, \cite{Mar97}, 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:
\begin{itemize}
  \item \texttt{Search}(e) determines if $e$ is in the dictionary, if so its working-set number is set to 0.
  \item \texttt{Predecessor}(e) will find $\max\{e' \in P \cup \{-\infty\} \mid e' < e\}$, without changing any working-set numbers.
  \item \texttt{Successor}(e) will find $\min\{e' \in P \cup \{\infty\} \mid e < e'\}$, without changing any working-set numbers.
\end{itemize}
Cache-Oblivious Implicit Predecessor Dictionaries with the Working-Set Property

There are numerous data structures and algorithms in the implicit model which range from binary heaps [16] to in-place 3-D convex hull algorithms [6]. There has been a continuous development of implicit dictionaries, the first milestone was the implicit AVL-tree [12] having bounds of \( O(n \log n) \). The second milestone was the implicit B-tree [7] having bounds of \( O(\log^2 n / \log \log n) \) the third was the flat implicit tree [9] obtaining \( O(\log n) \) worst-case time for searching and amortized bounds for updates. The fourth milestone is the optimal implicit dictionary [8] obtaining worst-case \( O(\log n) \) for search, update, predecessor and successor.

Numerous non-implicit dictionaries attain the working-set property; splay trees [15], skip list variants [2], the working-set structure in [11], 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 [11], even combines the working-set property with finger search. In finger search we have a finger located on an element \( e \) 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 \( O(\min\{f(e), \ell_e\} + d(e, f) + 2)\). Table 1 gives an overview of previous results, and our contribution.

The dictionary in [3] is, in addition to being implicit, also designed for the cache-oblivious model [10], where all the operations imply \( O(B n) \) 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.

**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 \( n \).

| Ref. | WS prop. | Insert/ Delete(\( e \)) | Search(\( e \)) | Pred(\( e \))/ Succ(\( e \)) | Additional words |
|------|----------|-------------------------|-----------------|-----------------------------|-----------------|
| [12] |          | \( O(\log^2 n) \)      | \( O(\log^2 n) \) | –                          | None            |
| [7]  |          | \( O(\log \log \log n) \) | \( O(\log \log \log n) \) | –                          | None            |
| [9]  |          | \( O(\log n) \)         | \( O(\log n) \)  | \( O(\log n) \)            | None            |
| [8]  |          | \( O(\log n) \)         | \( O(\log n) \)  | \( O(\log n) \)            | None            |
| [11] |          | \( + \)                 | \( O(\log \ell_e) \) | \( O(\log \ell_e^*) \)     | \( O(\ell_e^*) \) |
| [3, Sec. 2] |          | \( + \)                 | \( O(\log \ell_e) \) exp. | \( O(\log \ell_e) \) | \( O(\log \ell_e^*) \) |
| [3, Sec. 3] |          | \( + \)                 | \( O(\log \ell_e) \) exp. | \( O(\log \ell_e^*) \) exp. | \( O(\sqrt{\pi}) \) |
| [4]  |          | \( + \)                 | \( O(\log n) \)  | \( O(\log \ell_e) \)       | \( O(\log \ell_e^*) \) |
| This paper |          | \( + \)                 | \( O(\log \min(\ell_{e^*}, \ell_e)) \) | \( O(\log \ell_e^*) \) | None |

- **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.

The dictionary in [8] is, in addition to being implicit, also designed for the cache-oblivious model [10], where all the operations imply \( O(B n) \) 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 \( O(B n) \) cache-misses, search, predecessor and successor in time \( O(\log \min(\ell_{p(e)}, \ell_e, \ell_{s(e)})) \), \( O(\log \ell_{p(e)}) \) and \( O(\log \ell_{s(e)}), \) and cache-misses \( O(\log \min(\ell_{p(e)}, \ell_e, \ell_{s(e)})) \), \( O(\log_B \ell_{p(e)}) \) and \( O(\log_B \ell_{s(e)}), \) 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 $O(\log \log n)$ blocks $B_0, \ldots, B_m$, of double exponential increasing sizes, where $B_i$ 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 $O(\log n)$ 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 $n$ where working-set number of at least $2$ lying at level $i$ can be guarding points, i.e. not be guarding points, i.e.

1. **Insert-left**($e$) inserts $e$ into $S$ which is now laid out in the addresses $[i - 1; j]$.
2. **Insert-right**($e$) inserts $e$ into $S$ which is now laid out in the addresses $[i; j + 1]$.
3. **Delete-left**($e$) deletes $e$ from $S$ which is now laid out in the addresses $[i + 1; j]$.
4. **Delete-right**($e$) deletes $e$ from $S$ which is now laid out in the addresses $[i; j - 1]$.
5. **Search**($e$) determines if $e \in S$, if so the address of element $e$ is returned.
6. **Predecessor**($e$) returns the address of the element $\max\{e' \in S \mid e' < e\}$ or that no such element exists.
7. **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 $O(2^{2^{i+k}})$, where $k$ is a constant. Elements in $B_i$ have a working-set number of at least $2^{2^{i+k-1}}$. The block $B_i$ consists of an array $D_i$ of $w_i = d \cdot 2^{2^{i+k}}$ 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 moveable 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.
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 $S \leq_j$ we mean $\bigcup_{j=0}^i S_j$ where $S_j \subseteq P$ for $j = 0, \ldots, i$.

For $S \subseteq P$, we define $\text{GLS}_S(e) = S \cap [p_{P \setminus S}(e); e]$ 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, see Figure 3. Similarly we define $\text{GRS}_S(e) = S \cap [s_{P \setminus S}(e); p_{S}(e)]$ to the right of $e$. We will notice that we will never find all points of $\text{GLS}_S(e)$ unless $|\text{GLS}_S(e)| < c$, the same applies for $\text{GRS}_S(e)$. For $S \subseteq P$, we define $\text{FGLS}_S(e) = S \cap [p_{P \setminus S}(p_{S}(e)); p_{S}(e)]$ to be the First Group of points from $S$ Left of $e$, i.e. the...
We assume that only the smallest and largest guarding points will not participate in the definition of two intervals, all other guarding points will.

Only the smallest and largest guarding points will not participate in the definition of two intervals, all other guarding points will.

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 $X \xrightarrow{b} Y$, 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 $i$.

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 $e_1 \in G_{\leq i-1}$ 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: $\text{insert}(e)$, $\text{delete}(e)$, $\text{search}(e)$, $\text{predecessor}(e)$ and $\text{successor}(e)$. We maintain the following invariants which uniquely determine the intervals:

1. A guarding point is part of the definition of at most two intervals 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 $[\min(P);\max(P)]$.

1.2 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.

---

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.

\[ p_{P\setminus S}(p_S(e_1)) \quad p_S(e_1) \quad e_1 \quad p_{P\setminus S}(e_2) \quad e_2 \]

| FGL$_S$(e$_1$) | GIL$_S$(e$_2$) |

Legend: $\bullet S$ $\circ P \setminus S$ $\circ P$

**Figure 3** Here is a illustration of FGL and GIL. Notice that GIL$_S$(e$_1$) = $\emptyset$ whereas FGL$_S$(e$_1$) $\neq \emptyset$.
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.

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

1.4 Each arriving point in $D_i$ and $A_i$ 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 $2^{2^{\max(i,j)-1+k}}$.

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

1.5 $|D_0| = \min(|B_0| - 2, w_0)$ and $|D_i| = \min(|B_i|, w_i)$ for $i = 1, \ldots, m$.

1.6 $|R_i| \leq 2^{d+i}$ and $|W_i| + |H_i| + |C_i| \neq 0 \Rightarrow |R_i| = 2^{d+i}$ for $i = 0, \ldots, m$.

1.7 $|A_i| + |W_i| = 2^{2^{i+k}}$ for $i = 0, \ldots, m - 1$, and $|A_m| + |W_m| \leq 2^{m+k}$.

1.8 $|A_i| < 2^{d+k}$ for $i = 0, \ldots, m$.

1.9 $|H_i| + |C_i| = 4 \cdot 2^{2^{i+k}} + c_i$, where $c_i \in [-c; c]$, for $i = 0, \ldots, m - 1$.

From the above invariants we have the following observation:

O.1 From I.4 all points in $G_i$ are endpoints of intervals in level $i$, and each interval at most two endpoints. Hence for $i = 0, \ldots, m$ we have that

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

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

From I.4 we have the following lemma.

Lemma 1. Let $e$ be an element, $e_1 = p_{G_{≤i}}(e), e_2 = s_{G_{≤i}}(e)$ and $i$ be the smallest integer for which $I(e_1, e_2, i) = e_1; e_2 \cup \bigcup_{j=0}^{i} B_j \neq \emptyset$. 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.

Proof. Assume that $i$ is the minimal $i$ that fulfills $I(e_1, e_2, i) \neq \emptyset$, where $e_1 = p_{G_{≤i}}(e)$ and $e_2 = s_{G_{≤i}}(e)$. We will have two cases depending on if $e$ is guarding or not.

Let’s first handle case 2) where $e$ is guarding and hence in the dictionary: Since $e$ is in the dictionary and $e_1 < e < e_2$, we have from the minimality of $i$ that $e$ lies in level $i$, and from I.4 $e$ is then part of an interval lying in level $i$ either to the left or to the right. Say $e$ is part of an interval to the left i.e. the interval $(e'_1; e)$. If $e_1 < e'_1$ then this would contradict that $e_1 = p_{G_{≤i}}(e)$ hence $e'_1 \leq e_1$, but since $e'_1$ is the predecessor of $e$ we have that $e'_1 = e_1$. So we know that $(e_1; e)$ defines an interval at level $i$. The argument for $(e; e_2)$ is symmetric.

In the case 1) $e$ is non-guarding and $e$ may lie in the dictionary or not: Since $e_1 < e < e_2$ we have from the minimality of $i$ that $e$ lies in level $i$, hence from I.4 we have that the interval $(e_1; e_2)$ lies at level $i$. ▶
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 \([I_1, I_3]\). Calls to fix for the levels we have altered will ensure that \([I_2, I_5]\) will be maintained, finally a call to rebalance-below\((i - 1)\) will ensure that \([I_1]\) 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. \([R\&G: I_1, I_3]\).
- **Fix**\((i)\) - moves points around inside of \(B_i\) to ensure the size invariants for each type of point. Fix\((i)\) might violate \([I_3]\) for level \(i\). \([R: I_1, I_3\] and that there exist \(c_1,\ldots,c_6\) such that \(|D_i| + \tilde{c}_1,|A_i| + \tilde{c}_2,|R_i| + \tilde{c}_3,|W_i| + \tilde{c}_4,|H_i| + \tilde{c}_5,|C_i| + \tilde{c}_6\) fulfill \([I_2, I_5]\) where \([\tilde{c}_i] = O(1)\) for \(i = 1,\ldots, 6\). \([G: I_1, I_3]\).
- **Shift-down**\((i)\) - will move at least 1 and at most \(c\) points from level \(i\) into level \(i - 1\). \([R: I_1, I_3\] and \(|D_i| + |C_i| = 4e2^{\Delta + k} + c_i', where \(0 \leq c_i' = O(1)\). \([G: I_1, I_3, I_4]\).
- **Shift-up**\((i)\) - will move at least 1 and at most \(c\) points from level \(i\) into level \(i + 1\). \([R: I_1, I_3\] and \(|D_i| + |C_i| = 4e2^{\Delta + k} + c_i', where \(c_i' = O(1)\). \([G: I_1, I_3, I_4]\).
- **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 \(j\). \([R\&G: I_1, I_3, I_4]\).
- **Rebalance-below**\((i)\) - If any \(c < c_i\) for \(l = 0,\ldots, i\) rebalance-below\((i)\) will correct it so \([I_1]\) will be fulfilled again for \(l = 0,\ldots, i\). \([R: I_1, I_3, I_4]\) and \(\sum_{l=0}^{i} \text{slack}(c_l) = O(1)\), where

\[
\text{slack}(c_l) = \begin{cases} 
0 & \text{if } c_l \in [-c; c] \\
|c_l| - c & \text{otherwise .} 
\end{cases}
\]

\([G: I_1, I_3, I_4]\).
- **Rebalance-above**\((i)\) - If any \(c_i < -c\) for \(l = i,\ldots, m - 1\) rebalance-above\((i)\) will correct it so \([I_1]\) will be fulfilled again for \(l = i,\ldots, m - 1\). \([R: I_1, I_3, I_4]\) and \(\sum_{l=i}^{m-1} \text{slack}(c_l) = O(1)\). \([G: I_1, I_3, I_4]\).

**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 \(e_1 = \rho_{G_{\leq}}(e), e_2 = s_{G_{\leq}}(e), p = \rho_{B_1\backslash G_{\leq}}(e)\) and \(s = s_{B_1\backslash G_{\leq}}(e)\). 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 \(e\). See Figure 4 for an example of the execution.

**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 \(e_1 = \rho_{G_{\leq}}(e), p = \rho_{B_1}(e), e_2 = s_{G_{\leq}}(e)\) and \(s = s_{B_1}(e)\). 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 \(\max(e_1, p)\) \((\min(s, e_2))\). See Figure 4 for an example of the execution.
**Find/Predecessor/Successor(e)**

![Figure 4 The last three iterations of the while-loop of find(e), predecessor(e) and successor(e).](image)

| 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 \( c_l = \text{GLC}_l(e), c_r = \text{GLR}_l(e), h_l = \text{GLH}_l(e) \) and \( h_r = \text{GLH}_r(e) \). We find the level \( i \) of the interval \((e_l; e_2)\) 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 \((e_l; e_2)\) is of type \((e_l; e_2)\) 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 \( i \). 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 \([1][1][8]\) but \([1][7]\) 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 \( H_i \cup C_i \xrightarrow{h_1} W_i, W_i \xrightarrow{h_2} R_i, R_i \xrightarrow{h_3} A_i \) and \( A_i \xrightarrow{h_4} D_i \) 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 \( |R_i| < 2^{2+k} \) then perform \( H_i \cup C_i \xrightarrow{h_1} W_i \) and \( W_i \xrightarrow{h_2} R_i \) where \( h_1 = \min(2^{2+k} - |R_i|, |H_i| + |C_i|) \) and \( h_2 = \min(2^{2+k} - |R_i|, |W_i| + h_1) \). If \( |R_i| > 2^{2+k} \) then perform \( R_i \xrightarrow{h_3} A_i \) where \( h_1 = |R_i| - 2^{2+k} \).

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

If \( |A_i| \geq 2^{2+k} \) then let \( h_1 = |A_i| - 2^{2+k} \), 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 \( R_i \).}
X to $A_i$ and perform $W_i \xrightarrow{h_i} H_i \cup C_i$.

Performing $W_i \rightarrow H_i \cup C_i$: Let $w = s_{W_i}(-\infty)$, $c_2 = \text{GL}_C(w)$, $c_r = \text{GR}_C(w)$, $h_1 = \text{GL}_H(w)$ and $h_r = \text{GR}_H(w)$. If $|c_1| > 0$ or $|c_r| > 0$ or $(c_1; c_2)$ is of type $[c_1; c_2]$ and only contains climbing points then make $w$ climbing at level $i$. Else if $|h_1| + |h_r| \geq c$ then make $h_1$, $w$ 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 $c_r = \text{GR}_C(w)$. 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 $i - 1$, see Figure 4. If $|D_i| < |D_{i-1}|$ then we let $a$ be some element in $D_i$. If $|D_i| < |B_i|$ then: if $|A_i| = 0$ we perform $H_i \cup C_i \xrightarrow{h_i} W_i$, $W_i \xrightarrow{c_i} R_i$ 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 $s_1 = s_{C_i}(-\infty)$, $e_1 = p_{G_{c_i}}(s_1)$ 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 $c_r = \text{GR}_C(s_4)$ be the group of climbing elements to the immediate right of $s_4$, if they exist, see Figure 5. 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 $c_l$ 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 $c_l \leq c$ 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 $p(s_1) = e_1$ 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$(e, i, j, \text{before, after})$ Depending on the type $t_{\text{before}}$ of point $e$ we have different cases, see Figure 5.

Non-guarding Let $e_1 = p_{G_{c_i}}(e)$, $e_2 = s_{G_{c_i}}(e)$ and let $l$ be the level of the left interval of $e_1$ and $r$ the level of the right interval of $e_2$. Also let $p_2 = p_{B_i \setminus G_{c_i}[e_1; e_2]}(p_1)$, $p_1 = p_{B_i \setminus G_{c_i}[e_1; e_2]}(e)$.

---

3 The move $H_i \cup C_i \xrightarrow{h_i} W_i$ will be performed the same way as we did it in fix.

4 See the analysis in Section 4 for a proof that $|C_i| > 0$.  

STACS 2012
$s_1 = s_{B\setminus G_i \cap [e_1, e_2]}(e)$ and $s_2 = s_{B\setminus G_i \cap [e_1, e_2]}(s_1)$, also let $c_l = FGL_{C_l \cap [e_1, e_2]}(e)$ be the elements in the first climbing group left of $e$, likewise let $c_r = FGR_{C_r \cap [e_1, e_2]}(e)$ be the elements in the first climbing group right of $e$.

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 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 $c'_l$ denote the elements of $c_l$ 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 $\langle e_1; e'_1 \rangle$ or contains non-climbing points) and $|c'_l| < c$ then make $c'_l$ helping at level $i$. If $\langle [e'_2; e_2] \rangle$ is not of type $\langle e'_2; e_2 \rangle$ or contains non-climbing points) and $|c'_r| < c$ then make $c'_r$ helping at level $i$. Call fix($i$), fix($j$), fix($\min(l, i)$) and fix($\min(i, r)$).

**Guarding** If $e = \min(P)$ or $e = \max(P)$ we simply do nothing and return. Let $e_1 = p_{G_{<h}}(e)$ be the left endpoint of the left interval $[e_1; e']$ lying at level $h$ and $e_2 = s_{G_{<h}}(e)$ 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 $l$ be the level of the left interval of $e_1$ and $r$ the level of the right
interval of $e_2$. Let $p_2 = p_{B_h \setminus G \cap [e_1 : e_2]}(p_1)$ and $p_1 = p_{B_h \setminus G \cap [e_1 : e]}(e)$ be the two left points of $e$, if they exists, $s_1 = s_{B_h \setminus G \cap [e_1 : e_2]}(e)$ and $s_2 = s_{B_h \setminus G \cap [e_1 : e]}(s_1)$ the two right points of $e$, if they exits. Also let $c_l = \text{FGC}_l(e)$ and $c_r = \text{FGC}_r(e)$.

If $p_2$ does not exist 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 $c'_1$ be the points of $e_1$ which was not moved and $c'_r$ the points of $c_r$ which was not moved. If $|c'_1| < c$ then make $c'_1$ 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 $|(e'_1 ; e_2)$ is not of type $[c'_1 ; e_2]$ or contains non-climbing points and $|c'_r| < c$ then make $c'_r$ helping at level $i$. In all cases call $\text{fix}(\min(i,h))$, $\text{fix}(h)$ and $\text{fix}(i)$. If $i > j$ then call $\text{fix}(j)$ and $\text{fix}(\min(j,r))$.

Delete$(e)$ 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 $c_l = \text{GLC}_l(e)$ be the elements in the climbing group immediately left of $e$, let $c_r = \text{GRC}_l(e)$ be the elements in the climbing group immediately right of $e$, let $h_l = \text{GLH}_l(e)$ be the elements in the helping group immediately left of $e$, and let $h_r = \text{GRH}_r(e)$ be the elements in the helping group immediately right of $e$. Let $e_1 = p_{G \cap [e_1 ; e]}(e)$ and $e_2 = s_{G \cap [e_1 ; e]}(e)$. Let $l$ 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 $|e_1; e_2| \cap B_i = 1$: 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 rebalance-above(1).

The other case is $|e_1; e_2| \cap B_i > 1$: If $|(e_1; e_2)$ is not of type $[e_1; e_2]$ or contains non-climbing points and $|c_i| + |c_r| < c$ then make $c_l$ and $c_r$ helping at level $i$. If $|h_l| + |h_r| \geq c$ then make $h_l$ and $h_r$ climbing at level $i$. Delete $e$, call $\text{fix}(i)$ and rebalance-above(1).

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 $s_1 = s_{B_h \setminus G \cap [e_1 : e]}(e)$, $s_2 = s_{B_h \setminus G \cap [e_1 : e]}(s_1)$, $t_1 = s_{B_h \setminus G \cap [e'_1 : e']}(e')$ and $t_2 = s_{B_h \setminus G \cap [e'_1 : e']}(|t_1|)$.

If $s_2$ exists then delete $e$ make $s_1$ guarding at level 0 and call $\text{fix}(0)$. 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 $\text{fix}(0)$ and $\text{fix}(i)$. 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 $\text{fix}(0)$ and $\text{fix}(i)$. In all the previous cases return.

Guarding Let $h$ be the level of the left interval $(e_1 : e_2)$, 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 $e_1 = p_{G \leq h}(e)$ and $e_2 = s_{G \leq h}(e)$. Let $p_2 = p_{B_h \setminus G \cap [e_1 : e_2]}(p_1)$ and $p_1 = p_{B_h \setminus G \cap [e_1 : e]}(e)$. Let $c_l = \text{FGC}_l(e)$ 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 $c'_l$ be the points
in $c_l$ which was not moved in the previous movement of points. If $|c'_l| < c$ make $c'_l$ helping at level $h$. If $c'$ is $c_1$ then call $\text{fix}(l)$. Delete $e$, call $\text{fix}(h)$, $\text{fix}(i)$ and $\text{rebalance-above}(1)$.

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

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

### 3 Memory management

We will now deal with the memory layout of the data structure. We will put the blocks in the order $B_0, \ldots, B_m$, 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 $w_i = d2^{i+k}$ 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 $O(2^{i+k})$ bits which we store here. As each of the moveable dictionaries in $B_i$ have size $O(2^{i+k})$ we need to encode numbers in $O(2^{i+k})$ 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 $l = O(1)$ 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
  - that $m_j, \gamma \leq m_h, \gamma$, for $j \leq h$,
  - the set of elements $S_{in}$ to put into $\gamma$, where $|S_{in}| = O(1)$,
  - the set of elements $S_{out}$ to take out of $\gamma$, where $|S_{out}| = O(1)$ 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 $|m_j, \delta|$ 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 5.

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

---

5 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.
Internal-movement($m_1, \ldots, m_l$)

$$S_{\text{end}} \leftarrow [D, m_1.\gamma, m_1.\gamma + 1]$$

| $m_{i+1}$.delta | $m_{i+2}$.gamma | $m_{i+3}$.gamma+1 | $m_{i+4}$.gamma+2 | $m_{i+5}$.gamma+3 | $m_{i+6}$.gamma+4 | $m_{i+7}$.gamma+5 | $m_{i+8}$.gamma+6 | $m_{i+9}$.gamma+7 | $m_{i+10}$.gamma+8 |
|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|
| $S_{\text{in}}$ | $m_1.\gamma$    | $m_2.\gamma$    | $m_3.\gamma$    | $m_4.\gamma$    | $m_5.\gamma$    | $m_6.\gamma$    | $m_7.\gamma$    | $m_8.\gamma$    | $m_9.\gamma$    | $m_{10}.\gamma$ |





Figure 6 (Left) Memory movement of internal-movement inside of a block $B_i$. (Right) Memory movement of external-movement across multiple blocks $B_{M_1.\gamma}, \ldots, B_{M_{l.\gamma}}$.

\textbf{External-movement}($M_1, \ldots, M_l$)  

External-movement takes a list of external moves $M_1, \ldots, M_l$, where $l = O(1)$. 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 $M_j.\gamma < M_k.\gamma$ for $j < h$,
- the list of internal moves $m_1, \ldots, m_q$ to perform on block $\gamma$, where $q = O(1)$, 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 $\Delta = \sum_{i=1}^l M_i.\Delta$ be the total size change of the dictionary after the external-moves have been performed. If $\Delta = 0$ then we let $\gamma_{\text{end}} = M_l.\gamma$ else we let $\gamma_{\text{end}} = m$. Let $p_{\text{end}} = \sum_{j=0}^{\gamma_{\text{end}}} |B_j| + \Delta$ 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 $M_i.\Delta < 0$ for $i = 1, \ldots, k$. Let $a_1, \ldots, a_k$ be the sublist of the indexes $\{1, \ldots, l\}$ where $M_i.\Delta > 0$ 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 $M_i.\gamma \leq i \leq \gamma_{\text{end}}$ so the rightmost block ends at position $p_{\text{end}}$. 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 $B_{M_{a_i}.\gamma+1}, \ldots, B_{M_{a_i+1}.\gamma-1}$ at the left end so they align with block $B_{M_{a_i}.\gamma}$.

It takes $O\left(\log \left(2^{2^i+k}\right)\right) = O\left(2^i+k\right)$ time and $O\left(\log_B \left(2^{2^i+k}\right)\right) = O\left(\frac{2^i+k}{\log B}\right)$ cache-misses to perform the internal moves on level $i$. In total all the external moves $M_1, \ldots, M_l$ use $O(2^{2^{\gamma_{\text{end}}+k}})$ time and $O\left(\frac{2^{2^{\gamma_{\text{end}}+k}}}{\log B}\right)$ cache-misses, as the external move at level $\gamma_{\text{end}}$ dominates the rest and $l = O(1)$.

\subsection{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.

4 Analysis

We will leave it for the reader to check that the pre-conditions for each methods in Section 2.3 are fulfilled and that the methods maintains all invariants. We will instead concentrate on using the invariants to prove correctness of the find, predecessor, successor and shift-up operations along with proving time and cache-miss bounds for these. We will leave the time
and cache-miss bounds of search, rebalance-above, rebalance-below, shift-down, insert, delete, and fix for the reader as they are all similarly in nature.

Find(e) We only consider the cases where \( \min(P) < e < \max(P) \), the other cases trivially give the correct answer in \( \mathcal{O}(1) \) time and cache-misses as \( \min(P), \max(P) \in G_0 \). Assume that \( \text{find}(e) \) stops at level \( i \), then we have that \( e_1 \leq p \) or \( s \leq e_2 \) so \( \mathcal{I}(e_1, e_2, i) \neq \emptyset \) and \( i \) is the minimal \( i \) where this happens, see lemma \( \text{I} \). Notice that \( e_1 = p_{G_{\leq i}}(e) \) and \( e_2 = s_{G_{\leq i}}(e) \), so \( e_1 \) and \( e_2 \) are the same as in lemma \( \text{I} \). When the while loop breaks we have all the preconditions for lemma \( \text{II} \). Now \( e \) is either in the dictionary, or not, and if \( e \) is in the dictionary it is either guarding or not, so we have three cases.

Case 1) \( e \) is in the dictionary and is non-guarding: then we have from lemma \( \text{II} \) that \((e_1; e_2)\) is an interval at level \( i \) and \( e \in B_i \). From this we also have that \( \log(\ell_e) \geq \log(2^{2i+k+1}) \).

Case 2) \( e \) is not in the dictionary: from lemma \( \text{II} \) \((e_1; e_2)\) lie at level \( i \) and we know that \( e \) intersects it. Since \( e \) is not in the dictionary \( \ell_e = n \) and then \( \log(\ell_e) \geq \log(2^{2i+k+1}) \).

Case 3) \( e \) is in the dictionary and is guarding: from lemma \( \text{II} \) we have that either \((e_1; e)\) or \((e; e_2)\) lie in level \( i \), hence \( e \in G_i \subseteq B_i \). From this we also have that \( \log(\ell_e) \geq \log(2^{2\max(i,j)+k+1}) \geq \log(2^{2i+k+1}) \).

From the above we see that \( \text{find}(e) \) runs in \( \mathcal{O}(\log(2^{2i+k+1})) = \mathcal{O}(\log \min(\ell_p(e), \ell_e, \ell_s(e))) \) time. When we look at the cache-misses we will first notice that the first \( \log(\log B) \) levels will fit in a single cache-line because all levels are next to each other in the memory layout, so the total cache-misses will be

\[
\mathcal{O} \left( 1 + \sum_{j=\lceil \log \log B \rceil + 1}^i \left( 1 + \log_B \left( 2^{2j+k} \right) \right) \right) = \mathcal{O} \left( \frac{2^{i+k}}{\log B} \right) = \mathcal{O} \left( \log_B \min(\ell_p(e), \ell_e, \ell_s(e)) \right). 
\]

Predecessor(e) (and successor(e)) We will only handle the predecessor operation, the case for the successor is symmetric. Since we have the same condition in the while loop as for find, we know that when it breaks it implies that \( \mathcal{I}(e_1, e_2, i) \neq \emptyset \). So from lemma \( \text{II} \) \( e \) intersects the interval at level \( i \) and the predecessor of \( e \) is now \( \max(e_1, p) \).

From lemma \( \text{II} \) we know that \( \log(\ell_p) \geq \log(2^{2i+k+1}) \) and the total time usage is \( \sum_{j=0}^i \mathcal{O}(\log(2^{2j+k})) = \mathcal{O}(2^{i+k}) = \mathcal{O}(\log(\ell_p)) \). Like in find, the first \( \log(\log B) \) levels fit into one block/cache-line hence the total cache-misses will be \( \mathcal{O}(\log_B(\ell_p)) \).

Shift-up(i) For shift-up to work for level \( i \) it is mandatory that \( |C_i| > 0 \) so that \( s_{C_i}(\infty) \) will return a element which can be moved to level \( i + 1 \). From the precondition that \( |H_i| + |C_i| = 4c2^{2i+k} + c_i \), where \( c \leq c_i = \mathcal{O}(1) \), we have that

\[
|C_i| = 4c2^{2i+k} + c_i - |H_i| \geq 4c2^{2i+k} - c - |H_i|
\]

so proving that \( |H_i| < 4c2^{2i+k} - c \) is enough. From lemma \( \text{III} \) we can at most have \( c - 1 \) helping points in a helping group, so for every \( c - 1 \) helping points we need a separating point, the role of the separating point can be played by a point from \( D_i, A_i, R_i, W_i \) or \( G_{\leq i-1} \). These are the only ways to contribute points to \( H_i \) hence for \( i \geq 1 \) we have this bound

\[
|H_i| \leq (c-1)(|D_i| + |A_i| + |R_i| + |W_i| + |G_{\leq i-1}|)
\]

\[
\leq (c-1) \left( w_i + 2 \cdot 2^{i+k} + \sum_{j=0}^{i-1} (4 + 2d + 8c)2^{2j+k} + 2c \right)
\]

\[
\leq (c-1) \left( d \cdot 2^{i+k} + 2 \cdot 2^{i+k} + (4 + 2d + 8c) \cdot 2 \cdot 2^{2i+k-1} + 2ci \right)
\]
Where we in (⋆) have used 1[18], 1[7] and 0[7] and in (⋆⋆) have used that $2^{2^l} = 2^{2^{l-1}} \cdot 2^{2^{l-1}}$ and $2^{2^{l-1}} \geq l$ for $l \geq 1$. If we use that $c = 5$ then for $k > \log \log(380 + 20d) + 1$ we have that $|C_i| \geq 4c2^{2^{i+k}} - c - |H_i| > 0$ for $i = 1, \ldots, m - 1$.

For $i = 0$ we have a different bound as $G \leq i$ is empty, we get the bound

$$|H_0| \leq (c - 1)(|D_i| + |A_i| + |R_i| + |W_i|) \leq (c - 1) \left( d \cdot 2^{i+k} + 2 \cdot 2^{2^{i+k}} \right)$$

but for $k > \log \log(380 + 20d) + 1$ this is of course still sufficient as $|H_0|$ only got smaller. So we have proved that $|C_i| > 0$ for level $i = 0, \ldots, m - 1$.

**Move-down**

Move-down moves a constant number of points around and into level $j$ from $i$. If $e$ is non-guarding we call fix($i$), fix($j$), fix(min($l$, $i$)) and fix(min($i$, $r$)). If $e$ is guarding we call fix(min($l$, $h$)), fix($h$) and fix($i$), and if $i > j$ we also call fix($j$) and fix(min($j$, $r$)). In the non-guarding case the time is bounded by $O(\log 2^{2^{i+k}}) = O(\log \ell_e)$ and the cache-misses are dominated by $O(\log B 2^{2^{i+k}}) = O(\log B \ell_e)$. In the guarding case the time is bounded by $O(\log 2^{2^{i+k}}) = O(\log \ell_e)$ and the cache-misses are dominated by $O(\log B 2^{2^{i+k}}) = O(\log B \ell_e)$.

### 5 Further work

We still have some open problems. Is it possible to change the insert operation such that when we insert a new point it will get a working-set value of $n + 1$ instead of 0? We can actually achieve this in our structure by loosening the invariant on the working-set number of guarding points to only require that they have a working-set number of at least $2^{\min(c+1, 2^{i+k})}$, but then for search the time will increase to $O(\log \min(\ell_e, \max(\ell(p(e), \ell(s(c))))$ and the cache-misses to $O(\log B \min(\ell_e, \max(\ell(p(e), \ell(s(c))))$ and the bounds for predecessor and successor queries would increase to $O(\log \max(\ell(p(e), \ell(s(c))))$ and $O(\log B \max(\ell(p(e), \ell(s(c))))$ cache-misses.

Another interesting question is if we can have a dynamic dictionary supporting efficient finger searches [5] in the implicit model, i.e., we have a finger $f$ located at a element and then we want to find an element $e$ in time $O(\log d(f, e))$, where $d(f, e)$ is the rank distance between $f$ and $e$. But very recently [14] have shown that finger search in $O(\log d(e, f))$ time is not possible in the implicit model. They give a lower bound of $\Omega(\log n)$. Now we could instead separate the finger search and the update of the finger, say we allow the finger search to use $O(q(d(e, f)))$ time for some function $q$. In this setting they also prove a lower of $\Omega(q^{-1}(\log n))$ for the update finger operation, where $q^{-1}$ is the inverse function of $q$. They also give almost tight upper bounds for this setting, in the form of a trade-off bound between the finger search and the update finger operations. The finger search operation uses $O(\log d(e, f)) + q(d(e, f))$ time, and the update finger operation uses $O(q^{-1}(\log n) \log n)$ time. But even given their result it still remains an open problem whatever dynamic finger search with an externally maintained finger is possible in $O(\log d(e, f))$ time. So in other words is it possible to do finger search in $O(\log d(e, f))$ time if we allow the data structure to store $O(\log n)$ bits of data that can store the finger?
References

1. Mihai Bădaiu, Richard Cole, Erik D. Demaine, and John Iacono. A unified access bound on comparison-based dynamic dictionaries. *Theoretical Computer Science*, 382(2):86–96, 2007.

2. Prosenjit Bose, Karim Douïeb, and Stefan Langerman. Dynamic optimality for skip lists and B-trees. In *Proc. 19th Annual ACM-SIAM Symposium on Discrete algorithms*, pages 1106–1114. SIAM, 2008.

3. Prosenjit Bose, John Howat, and Pat Morin. A distribution-sensitive dictionary with low space overhead. In *Proc. 11th International Symposium on Algorithms and Data Structures*, volume 5664 of *LNCS*, pages 110–118. Springer-Verlag, 2009.

4. Gerth Brodal, Casper Kejlberg-Rasmussen, and Jakob Truelsen. A cache-oblivious implicit dictionary with the working set property. In *Proc. 12th International Symposium on Algorithms and Data Structures*, volume 6507 of *LNCS*, pages 37–48. Springer-Verlag, 2010.

5. Gerth Stølting Brodal. Finger search trees. In Dinesh Mehta and Sartaj Sahni, editors, *Handbook of Data Structures and Applications*, chapter 11. CRC Press, 2005.

6. Timothy Moon-Yew Chan and Eric Y. Chen. Optimal in-place algorithms for 3-D convex hulls and 2-D segment intersection. In *Proc. 25th Annual Symposium on Computational Geometry*, pages 80–87. ACM, 2009.

7. G. Franceschini, R. Grossi, J.I. Munro, and L. Pagli. Implicit B-trees: New results for the dictionary problem. In *Proc. 43rd Annual IEEE Symposium on Foundations of Computer Science*, pages 145–154, 2002.

8. Gianni Franceschini and Roberto Grossi. Optimal worst-case operations for implicit cache-oblivious search trees. In *Proc. 8th International Workshop on Algorithms and Data Structures*, volume 2748 of *LNCS*, pages 114–126. Springer-Verlag, 2003.

9. Gianni Franceschini and Roberto Grossi. Optimal implicit dictionaries over unbounded universes. *Theory of Computing Systems*, 39:321–345, 2006.

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

11. John Iacono. Alternatives to splay trees with $O(\log n)$ worst-case access times. In *Proc. 12th Annual ACM-SIAM Symposium on Discrete algorithms*, pages 516–522. SIAM, 2001.

12. James Ian Munro. An implicit data structure supporting insertion, deletion, and search in $O(\log^2 n)$ time. *Journal of Computer and System Sciences*, 33(1):66–74, 1986.

13. 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.

14. Jesper Sindahl Nielsen and Jakob Truelsen. Finger search in the implicit model. *Work in progress*, 2011.

15. Daniel Dominic Sleator and Robert Endre Tarjan. Self-adjusting binary search trees. *J. ACM*, 32(3):652–686, 1985.

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