Open access • Book Chapter • DOI:10.1007/978-3-642-35261-4_55

Finger Search in the Implicit Model — Source link

Gerth Stølting Brodal, Jesper Sindahl Nielsen, Jakob Truelsen

Institutions: Aarhus University

Published on: 19 Dec 2012 - International Symposium on Algorithms and Computation

Topics: Finger search

Finger Search in the Implicit Model

Gerth Stølting Brodal, Jesper Sindahl Nielsen, Jakob Truelsen

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

Abstract. We address the problem of creating a dictionary with the finger search property in the strict implicit model, where no information is stored between operations, except the array of elements. We show that for any implicit dictionary supporting finger searches in $q(t) = \Omega(\log t)$ time, the time to move the finger to another element is $\Omega(q^{-1}(\log n))$, where t is the rank distance between the query element and the finger. We present an optimal implicit static structure matching this lower bound. We furthermore present a near optimal implicit dynamic structure supporting `search`, `change-finger`, `insert`, and `delete` in times $\mathcal{O}(q(t))$, $\mathcal{O}(q^{-1}(\log n)\log n)$, $\mathcal{O}(\log n)$, and $\mathcal{O}(\log n)$, respectively, for any $q(t) = \Omega(\log t)$. Finally we show that the `search` operation must take $\Omega(\log n)$ time for the special case where the finger is always changed to the element returned by the last query.

1 Introduction

We consider the problem of creating an implicit dictionary [4] that supports finger search. A dictionary is a data structure storing a set of elements with distinct comparable keys such that an element can be located efficiently given its key. It may also support predecessor and successor queries where given a query k it must return the element with the greatest key less than k or the element with smallest key greater than k. A dynamic dictionary also supports insertion the and deletion of elements.

A dictionary has the finger search property if the time for searching is dependent on the rank distance t between a specific element f, called the finger, and the query key k. In the static case $\mathcal{O}(\log t)$ search can be achieved by exponential search on a sorted array of elements starting at the finger. Dynamic finger search data structures have been widely studied, e.g. some of the famous dynamic structures that support finger searches are splay trees, randomized skip lists and level linked (2-4)-trees. These all support finger search in $\mathcal{O}(\log t)$ time, respectively in the amortized, expected and worst case sense. For an overview of data structures that support finger search see [3].

We consider two variants of finger search structures. The first variant is the *finger search dictionary* where the `search` operation also changes the finger to the returned element. The second variant is the *change finger dictionary*

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

where the `change-finger` operation is separate from the `search` operation. We consider the two problems in the strict implicit model where we are only allowed to explicitly store the elements and the number of elements n between operations as defined in [1, 6]. Note that the static sorted array solution does not fit into this model, since we are not allowed to use additional space to store the index of f between operations. Other papers allow $\mathcal{O}(1)$ additional words [4, 5, 8]. We call this the weak implicit model. In both models almost all structure has to be encoded in the order of the elements. In either model the only allowed operations on elements are comparisons and swaps. As there is no agreement on the exact definition of the implicit model, it is interesting to study the limits of the strict model. We show that for a static dictionary in the strict model, if we want a `search` time of $\mathcal{O}(\log t)$, then `change-finger` must take time $\Omega(n^\epsilon)$, while in the weak model a sorted array achieves $\mathcal{O}(1)$ `change-finger` time.

Much effort has gone into finding a worst case optimal implicit dictionary. Among the first [7] gave a dictionary supporting insert, delete and search in $\mathcal{O}(\log^2 n)$ time. In [5] an implicit B-tree is presented, and finally in [4] a worst case optimal and cache oblivious dictionary is presented. To prove our dynamic upper bounds we use the movable implicit dictionary presented in [2], supporting `insert`, `delete`, `predecessor`, `successor`, `move-left` and `move-right`. The operation `move-right` moves the dictionary laid out in cells i through j to $i+1$ through $j+1$ and `move-left` moves the dictionary the other direction.

Preliminaries. A common implicit data structure technique is the pair encoding of bits. When we have two distinct consecutive elements x and y, then they encode a 1 if $x \leq y$ and 0 otherwise. The running time of the `search` operation is hereafter denoted by $q(t, n)$. Throughout the paper we require that $q(t, n)$ is non decreasing in both t and n, $q(t, n) \geq \log t$ and that $q(0, n) < \log \frac{n}{2}$. We define $Z_q(n) = \min\{t \in \mathbb{N} \mid q(t, n) \geq \log \frac{n}{2}\}$, i.e. $Z_q(n)$ is the smallest rank distance t, such that $q(t, n) > \log \frac{n}{2}$. Note that $Z_q(n) \leq \frac{n}{2}$ (since by assumption $q(t, n) \geq \log t$), and if q is a function of only t, then Z_q is essentially equivalent to $q^{-1}(\log \frac{n}{2})$. As an example $q(t, n) = \frac{1}{\varepsilon} \log t$, gives $Z_q(n) = \left\lceil (\frac{n}{2})^\varepsilon \right\rceil$, for $0 < \varepsilon \leq 1$. We require that for a given q, $Z_q(n)$ can be evaluated in constant time, and that $Z_q(n+1) - Z_q(n)$ is bounded by a fixed constant for all n.

We will use set notation on a data structure when appropriate, e.g. $|X|$ will denote the number of elements in the structure X and $e \in X$ will denote that the element e is in the structure X. Given two data structures or sets X and Y, we say that $X \prec Y \Leftrightarrow \forall (x, y) \in X \times Y : x < y$. We use $d(e_1, e_2)$ to denote the rank distance between two elements, that is the difference of the index of e_1 and e_2 in the sorted key order of all elements in the structure. At any time f will denote the current finger element and t the rank distance between this and the current search key.

Our results. In Section 2 we present a static *change-finger implicit dictionary* supporting `predecessor` in time $\mathcal{O}(q(t, n))$, and `change-finger` in time $\mathcal{O}(Z_q(n) + \log n)$, for any function $q(t, n)$. Note that by choosing $q(t, n) = \frac{1}{\varepsilon} \log t$,

we get a `search` time of $\mathcal{O}(\log t)$ and a change finger time of $\mathcal{O}(n^\varepsilon)$ for any $0 < \varepsilon \leq 1$.

In Section 3 we prove our lower bounds. First we prove (Lemma 1) that for any algorithm `A` on a strict implicit data structure of size n that runs in time at most τ, whose arguments are keys or elements from the structure, there exists a set $\mathcal{X}_{\mathtt{A},n}$ of at most $\mathcal{O}(2^\tau)$ array entries, such that `A` touches only array entries from $\mathcal{X}_{\mathtt{A},n}$, no matter the arguments to `A` or the content of the data structure. We use this to show that for any *change-finger implicit dictionary* with a search time of $q(t,n)$, `change-finger` will take time $\Omega(Z_q(n) + \log n)$ for some t (Theorem 1). We prove that for any *change-finger implicit dictionary* `search` will take time at least $\log t$ (Theorem 2). A similar argument applies for `predecessor` and `successor`. This means that the requirement $q(t,n) \geq \log t$ is necessary. We show that for any *finger-search implicit dictionary* `search` must take at least $\log n$ time as a function of both t and n, i.e. it is impossible to create any meaningful finger-search dictionary in the strict implicit model (Theorem 3).

By Theorem 1 and 2 the static data structure presented in Section 2 is optimal w.r.t. `search` and `change-finger` time trade off, for any function $q(t,n)$ as defined above. In the special case where the restriction $q(0,n) < \log \frac{n}{2}$ does not hold [4] provides the optimal trade off.

Finally in Section 4 we outline a construction for creating a dynamic *change-finger implicit dictionary*, supporting `insert` and `delete` in time $\mathcal{O}(\log n)$, `predecessor` and `successor` in time $\mathcal{O}(q(t,n))$ and `change-finger` in time $\mathcal{O}(Z_q(n) \log n)$. Note that by setting $q(t,n) = \frac{2}{\varepsilon} \log t$, we get a `search` time of $\mathcal{O}(\log t)$ and a `change-finger` time of $\mathcal{O}(n^{\varepsilon/2} \log n) = \mathcal{O}(n^\varepsilon)$ for any $0 < \varepsilon \leq 1$, which is asymptotically optimal in the strict model. It remains an open problem if one can get better bounds in the dynamic case by using $\mathcal{O}(1)$ additional words.

2 Static finger search

In this section we present a simple *change-finger implicit dictionary*, achieving an optimal trade off between the time for `search` and `changer-finger`.

Given some function $q(t,n)$, as defined in Section 1, we are aiming for a `search` time of $\mathcal{O}(q(t,n))$. Let $\Delta = Z_q(n)$. Note that we are allowed to use $\mathcal{O}(\log n)$ time searching for elements with rank-distance $t \geq \Delta$ from the finger, since $q(t,n) = \Omega(\log n)$ for $t \geq \Delta$.

Intuitively, we start with a sorted list of elements. We cut the $2\Delta + 1$ elements closest to f (f being in the center), from this list, and swap them with the first $2\Delta + 1$ elements, such that the finger element is at position $\Delta + 1$. The elements that were cut out form the *proximity structure* P, the rest of the elements are in the *overflow structure* O (see Figure 2). A `search` for x is performed by first doing an exponential search for x in the proximity structure, and if x is not found there, by doing binary searches for it in the remaining sorted sequences.

The proximity structure consists of sorted lists $XS \prec S \prec \{f\} \prec L \prec XL$. The list S contains the up to Δ elements smaller then f that are closest to f w.r.t. rank distance. The list L contains the up to Δ closest to f, but larger than f.

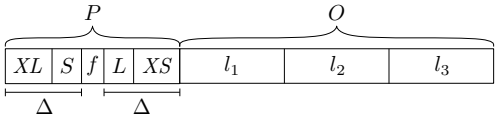

Fig. 1. Memory layout of the static dictonary.

Both are sorted in ascending order. XL contains a possibly empty sorted sequence of elements larger than elements from L, and XS contains a possibly empty sorted sequence of elements smaller than elements from S. Here $|XL| + |S| = \Delta = |L| + |XS|$, $|S| = \min\{\Delta, \text{rank}(f) - 1\}$ and $|L| = \min\{\Delta, n - \text{rank}(f)\}$. The overflow structure consists of three sorted sequences $l_2 \prec l_1 \prec \{f\} \prec l_3$, each possibly empty.

To perform a `change-finger` operation, we first revert the array back to one sorted list and the index of f is found by doing a binary search. Once f is found there are 4 cases to consider, as illustrated in Figure 2. Note that in each case, at most $2|P|$ elements have to be moved. Furthermore the elements can be moved such that at most $\mathcal{O}(|P|)$ swaps are needed. In particular case 2 and 4 can be solved by a constant number of list reversals.

For reverting to a sorted array and for doing `search`, we need to compute the lengths of all sorted sequences. These lengths uniquely determine the case used for construction, and the construction can thus be undone. To find $|S|$ a binary search for the split point between XL and S, is done within the first Δ elements of P. This is possible since $S \prec \{f\} \prec XL$. Similarly $|L|$ and $|XS|$ can be found. The separation between l_2 and l_3, can be found by doing a binary search for f in O, since $l_1 \cup l_2 \prec \{f\} \prec l_3$. Finally if $|l_3| < |O|$, the separation between l_1 and l_2 can be found by a binary search, comparing candidates against the largest element from l_2, since $l_2 \prec l_1$.

When performing the `search` operation for some key k, we first determine if $k < f$. If this is the case, an exponential search for k in S is performed. We can detect if we have crossed the boundary to XL, since $S \prec \{f\} \prec XL$. If the

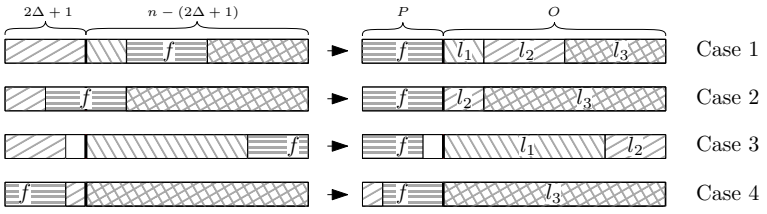

Fig. 2. Cases for the `change-finger` operation. The left side is the sorted array. In all cases the horizontally marked segment contains the new finger element and must be moved to the beginning. In the final two cases, there are not enough elements around f so P is padded with what was already there. The emphasized bar in the array is the $2\Delta + 1$ break point between the proximity structure and the overflow structure.

element is found it can be returned. If $k > f$ we do an identical search in L. Otherwise the element is neither located in S nor L, and therefore $d(k, f) > \Delta$. All lengths are reconstructed as above, and the element is searched for using binary search in X_l and l_3 if $k > f$ and, otherwise in X_s, $l1$ and l_2.

Analysis The `change-finger` operation first computes the lengths of all lists in $\mathcal{O}(\log n)$ time. The case used for constructing the current layout is then identified and reversed in $\mathcal{O}(\Delta)$ time. We locate the new finger f' by binary search in $\mathcal{O}(\log n)$ time and afterwards the $\mathcal{O}(\Delta)$ elements closest to f' are moved to P. We get $\mathcal{O}(\Delta + \log n)$ time for `change-finger`.

For searches there are two cases to consider. If $t \leq \Delta$, it will be located by the exponential search in P in $\mathcal{O}(\log t) = \mathcal{O}(q(t, n))$ time, since by assumption $q(t, n) \geq \log t$. Otherwise the lengths of the sorted sequences will be recovered in $\mathcal{O}(\log n)$ time, and a constant number of binary searches will be performed in $\mathcal{O}(\log n)$ time total. Since $t \geq \Delta \Rightarrow q(t, n) \geq \log \frac{n}{2}$, we again get a search time of $\mathcal{O}(q(t, n))$.

3 Lower bounds

To prove our lower bounds we use an abstracted version of the strict implicit model. The strict model requires that *nothing* but the elements and the number of elements are stored between operations, and that during computation elements can only be used for comparison. With these assumptions a decision tree can be formed for a given n, where nodes correspond to element comparisons and loads and leaves contain the answers. Note that in the weak model a node could probe a cell containing an integer, giving it a degree of n, which prevents any of our lower bound arguments.

Lemma 1. *Let* `A` *be an operation on an implicit data structure of length n, running in time τ worst case, that takes any number of keys as arguments. Then there exists a set $\mathcal{X}_{\mathtt{A},n}$ of size 2^τ, such that executing* `A` *with any arguments will touch only cells from $\mathcal{X}_{\mathtt{A},n}$ no matter the content of the data structure.*

Proof. Before loading any elements from the data structure, `A` can reach only a single state which gives rise to a root in a decision tree. When `A` is in some node s, the next execution step may load some cell in the data structure, and transition into another fixed node, or `A` may compare two previously loaded elements or arguments, and given the result of this comparison transition into one of two distinct nodes. It follows that the total number of nodes `A` can enter within its τ steps is $\sum_{i=0}^{\tau-1} 2^i < 2^\tau$. Now each node can access at most one cell, so it follows that at most 2^τ different cells can be probed by any execution of `A` within τ steps. □

Observe that no matter how many times an operation that take at most τ time is performed they will only be able to reach the same set of cells, since the decision tree is the same for all invocations.

Theorem 1. *For any* change-finger implicit dictionary *with a search time of $q(t,n)$ as defined in Section 1,* `change-finger` *requires $\Omega(Z_q(n)+\log n)$ time.*

Proof. Let $e_1 \dots e_n$ be a set of elements in sorted order with respect to the keys $k_1 \dots k_n$. Let $t = Z_q(n) - 1$. By definition $q(t+1,n) \geq \log \frac{n}{2} > q(t,n)$. Consider the following sequence of operations:

for $i = 0 \dots \frac{n}{t}$:
  `change-finger`(k_{it})
  for $j = 0 \dots t-1$: `search`(k_{it+j})

Since the rank distance of any query element is at most t from the current finger and q is non-decreasing each search operation takes time at most $q(t,n)$. By Lemma 1 there exists a set $\mathcal{X}$ of size $2^{q(t,n)}$ such that all queries only touch cells in $\mathcal{X}$. We note that $|\mathcal{X}| \leq 2^{q(t,n)} \leq 2^{\log(n/2)} = \frac{n}{2}$.

Since all n elements were returned by the query set, the `change-finger` operations, must have copied at least $n - |\mathcal{X}| \geq \frac{n}{2}$ elements into $\mathcal{X}$. We performed $\frac{n}{t}$ `change-finger` operations, thus on average the `change-finger` operations must have moved at least $\frac{t}{2} = \Omega(Z_q(n))$ elements into $\mathcal{X}$.

For the $\log n$ term in the lower bound, we consider the sequence of operations `change-finger`(k_i) followed by `search`(k_i) for i between 1 and n. Since the rank distance of any search is 0 and $q(0,n) < \log \frac{n}{2}$ (by assumption), we know from Lemma 1 that there exists a set $\mathcal{X}_s$ of size at most $2^{\log(n/2)}$, such that `search` only touches cells from $\mathcal{X}_s$. Assume that `change-finger` runs in time $c(n)$, then from Lemma 1 we get a set $\mathcal{X}_c$ of size at most $2^{c(n)}$ such that `change-finger` only touches cells from $\mathcal{X}_c$. Since every element is returned, the cell initially containing the element must be touched by either `change-finger` or `search` at some point, thus $|\mathcal{X}_c| + |\mathcal{X}_s| \geq n$. We see that $2^{c(n)} \geq |\mathcal{X}_c| \geq n - |\mathcal{X}_s| \geq n - 2^{\log(n/2)} = 2^{\log(n/2)}$, i.e. $c(n) \geq \log \frac{n}{2}$. □

Theorem 2. *For a* change-finger implicit dictionary *with* `search` *time $q'(t,n)$, where q' is none decreasing in both t and n, it holds that $q'(t,n) \geq \log t$.*

Proof. Let $e_1 \dots e_n$ be a set of elements with keys $k_1 \dots k_n$ in sorted order. Let $t \leq n$ be given. First perform `change-finger`(k_1), then for i between 1 and t perform `search`(k_i). From Lemma 1 we know there exists a set $\mathcal{X}$ of size at most $2^{q'(t,n)}$, such that any of the `search` operations touch only cells from $\mathcal{X}$ (since any element searched for has rank distance at most t from the finger). The `search` operations return t distinct elements so $t \leq |\mathcal{X}| \leq 2^{q'(t,n)}$, and $q'(t,n) \geq \log t$. □

Theorem 3. *For* finger-search implicit dictionary*, the* `finger-search` *operation requires at least $g(t,n) \geq \log n$ time for any rank distance $t > 0$ where $g(t,n)$ is non decreasing in both t and n.*

Proof. Let $e_1 \dots e_n$ be a set of elements with keys $k_1 \dots k_n$ in sorted order. First perform `finger-search`(k_1), then perform `finger-search`(k_i) for i between 1 and n. Now for all queries except the first, the rank distance $t \leq 1$ and by

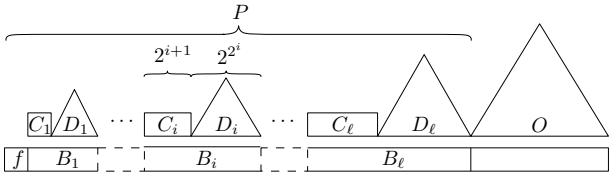

Fig. 3. Memory layout.

Lemma 1 there exists a set of memory cells $\mathcal{X}$ of size $2^{g(1,n)}$ such that all these queries only touch cells in $\mathcal{X}$. Since all elements are returned by the queries we have $|\mathcal{X}| = n$, so $g(1,n) \geq \log n$, since this holds for $t = 1$ it holds for all t. □

We can conclude that it is not possible to achieve any form of meaningful finger-search in the strict implicit model. The static *change-finger implicit dictionary* from Section 2 is by Theorem 1 optimal within a constant factor, with respect to the `search` to `change-finger` time trade off, assuming the running time of `change-finger` depends only on the size of the structure.

4 A dynamic structure

For any function $q(t,n)$, as defined in the introduction, we present a dynamic *change-finger implicit dictionary* that supports `change-finger`, `search`, `insert` and `delete` in $\mathcal{O}(\Delta \log n), \mathcal{O}(q(t,n)), \mathcal{O}(\log n)$ and $\mathcal{O}(\log n)$ time respectively, where $\Delta = Z_q(n)$ and n is the number of elements when the operation was started.

The data structure consists of two parts: a *proximity structure* P which contains the elements near f and an *overflow structure* O which contains elements further from f w.r.t. rank distance. We partition P into several smaller structures $B_1, \ldots, B_\ell$. Elements in B_i are closer to f than elements in B_{i+1}. The overflow structure O is an *implicit movable dictionary* [2] that supports `move-left` and `move-right` as described in the Section 1. See Figure 3 for the layout of the data structure. During a `change-finger` operation the proximity structure is rebuilt such that $B_1, \ldots, B_\ell$ correspond to the new finger, and the remaining elements are put in O.

The total size of P is $2\Delta + 1$. The i'th block B_i consists of a counter C_i and an implicit movable dictionary D_i. The counter C_i contains a pair encoded number c_i, where c_i is the number of elements in D_i smaller than f. The sizes within B_i are $|C_i| = 2^{i+1}$ and $|D_i| = 2^{2^i}$, except in the final block B_ℓ where they might be smaller (B_ℓ might be empty). In particular we define:

$$\ell = \min\left\{ \ell' \in \mathbb{N} \,\middle|\, \sum_{i=0}^{\ell'} \left(2^{i+1} + 2^{2^i}\right) > 2\Delta \right\}.$$

We will maintain the following invariants for the structure:

I.1 $\forall i < j, e_1 \in B_i, e_2 \in B_j : d(f, e_1) < d(f, e_2)$
I.2 $\forall e_1 \in B_1 \cup \cdots \cup B_\ell,\ e_2 \in O : d(f, e_1) \leq d(f, e_2)$
I.3 $|P| = 2\Delta + 1$
I.4 $|C_i| \leq 2^{i+1}$
I.5 $|D_i| > 0 \Rightarrow |C_i| = 2^{i+1}$
I.6 $|D_\ell| < 2^{2^\ell}$ and $\forall i < \ell : |D_i| = 2^{2^i}$
I.7 $|D_i| > 0 \Rightarrow c_i = |\{e \in D_i \mid e < f\}|$

We observe that the above invariants imply:

O.1 $\forall i < \ell : |B_i| = 2^{i+1} + 2^{2^i}$ (From I.5 and I.6)
O.2 $|B_\ell| < 2^{\ell+1} + 2^{2^\ell}$ (From I.4 and I.6)
O.3 $d(e, f) \leq 2^{2^k - 1} \leq \Delta \Rightarrow e \in B_j$ for some $j \leq k$ (From I.1 – I.6)

4.1 Block operations

The following operations operate on a single block and are internal helper functions for the operations described in Section 4.2.

`block_delete`(k, B_i): Removes the element e with key k from the block B_i. This element *must* be located in B_i. First we scan C_i to find e. If it is not found it must be in D_i, so we `delete` it from D_i. If $e < f$ we decrement c_i. In the case where $e \in C_i$ and D_i is nonempty, an arbitrary element g is deleted from D_i and if $g < f$ we decrement c_i. We then overwrite e with g, and fix C_i to encode the new number c_i. In the final case where $e \in C_i$ and D_i is empty, we overwrite e with the last element from C_i.

`block_insert`(e, B_i): Inserts e into block B_i. If $|C_i| < 2^{i+1}$, e is inserted into C_i and we return. Else we insert e into D_i. If D_i was empty we set $c_i = 0$. In either case if $e < f$ we increment c_i.

`block_search`(k, B_i): Searches for an element e with key k in the block B_i. We scan C_i for e, if it is found we return it. Otherwise if D_i is nonempty we perform a `search` on it, to find e and we return it. If the element is not found `nil` is returned.

`block_predecessor`(k, B_i): Finds the predecessor element for the key k in B_i. Do a linear scan through C_i and find the element l_1 with largest key less than k. Afterwards do a predecessor search for key k on D_i, call the result l_2. Return $\max(l_1, l_2)$, or that no element in B_i has key less than k.

4.2 Operations

In order to maintain correct sizes of P and O as the entire structure expands or contracts a `rebalance` operation is called in the end of every `insert` and `delete` operation. This is an internal operation that does not require I.3 to be valid before invocation.

`rebalance`(): Balance B_ℓ such that the number of elements in P less than f is as close to the number of elements greater than f as possible. We start by

evaluating $\Delta = Z_q(n)$, the new desired proximity size. Let s be the number of elements in B_ℓ less than f which can be computed as $c_\ell + |\{e \in C_\ell \mid e < f\}|$. While $2\Delta + 1 > |P|$ we move elements from O to P. We move the predecessor of f from O to B_ℓ if $O \prec \{f\} \vee (s < \frac{|B_\ell|}{2} \wedge \neg(\{f\} \prec O))$ and otherwise we move the successor of f to O. While $2\Delta + 1 < |P|$ we move elements from B_ℓ to O. We move the largest element from B_ℓ to O if $s < \frac{B_\ell}{2}$. Otherwise we move the smallest element.

`change-finger`(k): To change the finger of the structure to k, we first insert every element of $B_\ell \dots B_1$ into O. We then remove the element e with key k from O, and place it at index 1 as the new f, and finish by performing `rebalance`.

`insert`(e): Assume $e > f$. The case $e < f$ can be handled similarly. Find the first block B_i where e is smaller than the largest element l_i from B_i (which can be found using a predecessor search) or $l_i < f$. Now if $l_i > f$ for all blocks $j \geq i$, `block_delete` the largest element and `block_insert` it into B_{j+1}. In the other case where $l_i < f$ for all blocks $j \geq i$, `block_delete` the smallest element and `block_insert` it into B_{j+1}. The final element that does not have a block to go into, will be put into O, then we put e into B_i. In the special case where e did not fit in any block, we insert e into O. In all cases we perform `rebalance`.

`delete`(k): We perform a `block_search` on all blocks and a `search` in O to find out which structure the element e with key k is located in. If it is in O we just `delete` it from O. Otherwise assume $k < f$ (the case $k > f$ can be handled similarly), and assume that e is in B_i, then `block_delete` e from B_i. For each $j > i$ we `block_delete` the predecessor of f in B_j, and insert it into B_{j-1} (in the case where there is no predecessor, we `block_delete` the successor of f instead). We also delete the predecessor of f from O and insert it in B_ℓ. The special case where $k = f$, is handled similarly to $k < f$, we note that after this the predecessor of f will be the new finger element. In all cases we perform a `rebalance`.

`search`(k), `predecessor`(k) and `successor`(k), all follow the same general pattern. For each block B_i starting from B_1, we compute the largest and the smallest element in the block. If k is between these two elements we return the result of `block_search`, `block_predecessor` or `block_successor` respectively on B_i, otherwise we continue with the next block. In case k is not within the bounds of any block, we return the result of `search`(k), `predecessor`(k) or `successor`(k) respectively on O.

4.3 Analysis

By the invariants, we see that every C_i and D_i except the last, have fixed size. Since O is a movable dictionary it can be moved right or left as this final C_i or D_i expands or contracts. Thus the structure can be maintained in a contiguous memory layout.

The correctness of the operations follows from the fact that I.1 and I.2, implies that elements in B_j or O are further away from f than elements from B_i where $i < j$. We now argue that `search` runs in time $\mathcal{O}(q(t, n))$. Let e be the element we are searching for. If e is located in some B_i then at least half

the elements in B_{i-1} will be between f and e by I.1. We know from O.1 that $t = d(f, e) \geq \frac{|B_{i-1}|}{2} \geq 2^{2^{i-1}-1}$. The time spent searching is $\mathcal{O}(\sum_{j=1}^{i} \log |B_j|) = \mathcal{O}(2^i) = \mathcal{O}(\log t) = \mathcal{O}(q(t, n))$. If on the other hand e is in O, then by I.3 there are $2\Delta + 1$ elements in P, of these at least half are between f and e by I.2, so $t \geq \Delta$, and the time used for searching is $\mathcal{O}(\log n + \sum_{j=1}^{k} \log |B_j|) = \mathcal{O}(\log n) = \mathcal{O}(q(t, n))$. The last equality follows by the definition of Z_q. The same arguments work for `predecessor` and `successor`.

Before the `change-finger` operation the number of elements in the proximity structure by I.3 is $2\Delta + 1$. During the operation all these elements are inserted into O, and the same number of elements are extracted again by `rebalance`. Each of these operations are just `insert` or `delete` on a movable dictionary or a block taking time $\mathcal{O}(\log n)$. In total we use time $\mathcal{O}(\Delta \log n)$.

Finally to see that both `Insert` and `Delete` run in $\mathcal{O}(\log n)$ time, notice that in the proximity structure doing a constant number of queries in every block is asymptotically bounded by the time to do the queries in the last block. This is because their sizes increase double-exponentially. Since the size of the last block is bounded by n we can guarantee $\mathcal{O}(\log n)$ time for doing a constant number of queries on every block (this includes predecessor/successor queries). In the worst case, we need to insert an element in the first block of the proximity structure, and "bubble" elements all the way through the proximity structure and finally insert an element in the overflow structure. This will take $\mathcal{O}(\log n)$ time. At this point we might have to rebalance the structure, but this merely requires deleting and inserting a constant number of elements from one structure to the other, since we assumed $Z_q(n)$ and $Z_q(n + 1)$ differ by at most a constant. Deletion works in a similar manner.

References

1. Borodin, A., Fich, F.E., Meyer auf der Heide, F., Upfal, E., Wigderson, A.: A tradeoff between search and update time for the implicit dictionary problem. In: Proc. 13th ICALP, LNCS, vol. 226, pp. 50–59. Springer (1986)
2. Brodal, G.S., Kejlberg-Rasmussen, C., Truelsen, J.: A cache-oblivious implicit dictionary with the working set property. In: Proc. 21st ISAAC, Part II. LNCS, vol. 6507, pp. 37–48. Springer (2010)
3. Brodal, G.S.: Finger search trees. In: Mehta, D., Sahni, S. (eds.) Handbook of Data Structures and Applications, chap. 11. CRC Press (2005)
4. Franceschini, G., Grossi, R.: Optimal worst case operations for implicit cache-oblivious search trees. In: Proc. 8th, WADS, LNCS, vol. 2748, pp. 114–126. Springer (2003)
5. Franceschini, G., Grossi, R., Munro, J.I., Pagli, L.: Implicit B-Trees: New results for the dictionary problem. In: Proc. 43rd FOCS. pp. 145–154. IEEE (2002)
6. Frederickson, G.N.: Implicit data structures for the dictionary problem. JACM 30(1), 80–94 (1983)
7. Munro, J.I.: An implicit data structure supporting insertion, deletion, and search in $\mathcal{O}(\log^2 n)$ time. JCSS 33(1), 66–74 (1986)
8. Munro, J.I., Suwanda, H.: Implicit data structures for fast search and update. JCSS 21(2), 236–250 (1980)