A Unified Access Bound on Comparison-Based Dynamic Dictionaries [1]

Mihai Bădoiu

MIT Computer Science and Artificial Intelligence Laboratory,
32 Vassar Street, Cambridge, MA 02139, USA

Richard Cole [2]

Computer Science Department,
Courant Institute of Mathematical Sciences,
New York University,
251 Mercer Street, New York, NY 10012, USA

Erik D. Demaine [3]

MIT Computer Science and Artificial Intelligence Laboratory,
32 Vassar Street, Cambridge, MA 02139, USA

John Iacono [3,4]

Department of Computer and Information Science,
Polytechnic University,
5 MetroTech Center, Brooklyn, NY 11201, USA

Abstract

We present a dynamic comparison-based search structure that supports insert, delete, and search within the unified bound. The unified bound specifies that it is quick to access an element that is near a recently accessed element. More precisely, if $w(y)$ distinct elements have been accessed since the last access to element y, then the amortized cost to access element x is $O(\min_y \log(w(y) + d(x, y) + 2))$, where $d(x, y)$ denotes the rank distance between x and y among the current set of elements. This property generalizes the working-set and dynamic-finger properties of splay trees.

Preprint submitted to Elsevier Science *18 September 2006*

Contents

1 Introduction

Overview. The classic *dynamic optimality conjecture* states that the amortized performance of splay trees [ST85] is within a constant factor of the offline optimal dynamic binary search tree for any given sequence of operations. This conjecture has motivated the study of sublogarithmic time bounds that capture the performance of splay trees and other comparison-based data structures. For example, it is known that the performance of splay trees satisfies the following two upper bounds. The *working-set bound* [ST85] says roughly that recently accessed elements are cheap to access again. The *dynamic-finger bound* [CMSS00,Col00] says roughly that it is cheap to access an element that is near to the previously accessed element. These bounds are incomparable: one does not imply the other. For example, the access sequence $1, n, 1, n, 1, n, \ldots$ has a small working-set bound (constant amortized time per access) because each accessed element was accessed just two time units ago. In contrast, for this sequence the dynamic-finger bound is large (logarithmic time per access) because each accessed element has rank distance

Email addresses: `mihai@mit.edu` (Mihai Bădoiu), `cole@cs.nyu.edu` (Richard Cole), `edemaine@mit.edu` (Erik D. Demaine), `jiacono@poly.edu` (John Iacono).

[1] Several of the results presented here appeared in preliminary form in [BD04,Iac01a,Iac01b].

[2] Research supported in part by NSF grants CCR-0105678 and CCF-0515127.
[3] Research supported in part by NSF grant CCF-0430849.
[4] Research supported in part by NSF grant CCF-9732689.

$n-1$ from the previously accessed element. On the other hand, the access sequence $1, 2, \ldots, n, 1, 2, \ldots, n, \ldots$ has a small dynamic-finger bound because most accessed elements have rank distance 1 to the previously accessed element, whereas it has a large working-set bound because each accessed element was accessed n time units ago.

We propose a *unified bound* that is strictly stronger than these two bounds and all other proved bounds on splay trees and most other comparison-based structures. Roughly, the unified bound says that it is cheap to access an element that is near to a recently accessed element. For example, the access sequence $1, \frac{n}{2}+1, 2, \frac{n}{2}+2, 3, \frac{n}{2}+3, \ldots$ has a small unified bound because most accessed elements have rank distance 1 to the element accessed two time units ago, whereas it has large working-set and dynamic-finger bounds. It remains open whether splay trees satisfy the unified bound. However, we develop the *unified structure*, a comparison-based data structure on the pointer machine that attains the unified bound.

In the rest of this introduction, we give a more thorough overview of sublogarithmic bounds on comparison-based search structures.

Problem statement. The goal in this line of research is to understand the optimal time needed to maintain a dynamic set of elements from a totally ordered universe as it depends on the sequence of insertions, deletions, and searches performed. The model of computation is a pointer machine under the (unit-cost) comparison model. We consider a sequence of m operations (insertions, deletions, and searches) in which the ith operation involves element x_i. Thus the *access sequence* $X = \langle x_1, x_2, \ldots, x_m \rangle$ captures everything except the type of each operation. To capture insertions and deletions, we let S_i denote the set of elements in the structure just before operation i—at *time* i—and let n_i denote the number of elements in S_i. Thus our goal is to understand the optimal running time of a data structure as a function of the access sequence X and the sets $S_1, S_2, \ldots, S_m$ determined by the insertions and deletions.

Entropy bound. Let $p(x)$ denote the frequency (empirical probability) of searches to element x, i.e., the number of occurrences of x in the access sequence X divided by the length m of the sequence. The *optimal binary search trees* of [Knu71,HT71] achieve the *entropy bound*—$O(1 + \log 1/p(x_i))$ time for each access x_i—provided that the frequency values of p are known in advance. This bound is in fact optimal if the binary search tree cannot be restructured during the access sequence, or in expectation if the access sequence is generated by a stochastic process with probabilities given by p. Optimal binary search trees have been improved over the years to allow insertion and deletion, but these structures still have the fundamental limitation of requiring that the access distribution is known in advance.

On the other hand, splay trees also achieve the entropy bound of $O(1 + \log 1/p(x_i))$, only the bound is amortized rather than worst case. They achieve this bound without any prior knowledge of the input distribution. This property of splay trees was proved as the *static-optimality theorem* in [ST85].

Static-finger bound. Another theorem proved in [ST85] is the *static-finger theorem*. It states that, for any fixed key f (the "finger"), the amortized time to access element x_i is proportional to the logarithm of the rank distance between f and x_i at time i. The *rank distance* $d_i(x, y)$ between two elements x and y at time i is the number of elements in S_i between x and y, including x but not y. Thus the static finger theorem states that, for any fixed key f, the amortized time to access x_i is $O(\log(d_i(x_i, f) + 2))$. The $+2$ is to assure that the logarithm is always positive. If f is known, a data structure of Guibas, McCreight, Pass, and Roberts [GMPR77] achieves a worst-case running time of $O(\log(d_i(x_i, f) + 2))$ for access x_i. Splay trees achieve this running time, in the amortized sense, without any knowledge of f, i.e., simultaneously for all f.

Working-set bound. The working-set theorem, introduced in [ST85], is based upon the following idea: if an access sequence contains elements drawn only from a subset of size n' of the n elements, the amortized time for an access should be $O(\log n')$ instead of $O(\log n)$. The actual theorem uses the stronger idea that elements that have been accessed recently should take less time to access than elements that have not been accessed in a long time. Formally, let $w_i(z)$ be the number of distinct items accessed since the last access to z before time i (before the execution of access x_i). The working-set theorem of [ST85] states that the amortized time to access x_i in a splay tree is $O(\log(w_i(x_i)+2))$.

It was observed in [Iac01b] that the working-set bound is the strongest of the three bounds presented so far (entropy, static finger, and working set): a working-set theorem implies a static-finger theorem, and a static-finger theorem implies a static-optimality theorem, in any data structure. Thus the working-set bound plays an important role at least in our current understanding of splay trees.

As a warmup toward our main result, we present in Section 2 a simple data structure called the *working-set structure*. This data structure has the same $O(\log(w_i(x_i) + 2))$ performance attributed to splay trees, except that the performance of the working-set structure is worst-case instead of amortized. Indeed, the working-set structure achieves a worst-case bound of $O(\log n_i)$ per access, in contrast to the $\Theta(n_i)$ worst-case performance of a single access in splay trees. As mentioned above, the working-set bound implies that the working-set structure satisfies both the static-finger bound and the static-optimality bound, albeit only in the amortized sense. This amortization is best possible: it is easy to show that the static-finger and static-optimality bounds cannot be satisfied in the worst-case in any data structure lacking knowledge of the finger and of the frequencies.

Sequential-access bound. One sublogarithmic access bound that splay trees have but is not implied by the working-set theorem is that, if the access sequence X simply consists of searching for every element in the data structure in sorted order repeatedly, then the amortized cost per access is $O(1)$. This result is known as the *sequential access lemma* and was proved by Tarjan [Tar85], with alternative proofs by Sundar [Sun92,Sun91] and Elmasry [Elm04].

Dynamic-finger bound. A generalization of the sequential-access lemma is the dynamic-finger theorem, conjectured in [ST85] and proved by Cole et al. [CMSS00,Col00]. This bound states that an access should be fast if it is close, in terms of rank distance, to the previous access. More precisely, in splay trees, the amortized cost to access x_i is $O(\log(d_i(x_i, x_{i-1}) + 2))$. A non-self-adjusting data structure with this performance predates splay trees: The level-linked trees of Brown and Tarjan [BT80] support accesses in $O(\log(d_i(x_i, x_{i-1})+2))$ worst-case time.

Unified bound. The working-set theorem and the dynamic-finger theorem are the best currently known analyses of access sequences in splay trees, yet each is easily seen to be incomplete. Consider the following three search sequences of length $m \geq n \log n$ on the set $\{1, 2, \ldots, n\}$ for n even:

$X_1 : 1, 2, \ldots, n, 1, 2, \ldots n, 1, 2, \ldots$

$X_2 : 1, n, 1, n, 1, n, \ldots$

$X_3 : 1, \frac{n}{2} + 1, 2, \frac{n}{2} + 2, 3, \frac{n}{2} + 3, \ldots \frac{n}{2}, n, 1, \frac{n}{2} + 1, \ldots$

In X_1, the dynamic-finger theorem would tightly bound this sequence as taking $O(m)$ total time to execute on a splay tree, while the working-set theorem could only say that the running time is $O(m \log n)$. The situation is reversed in X_2, with the working-set theorem tightly bounding the execution time as $O(m)$ while the dynamic-finger theorem yields only an $O(m \log n)$ bound. More troubling is X_3. Both the working-set theorem and the dynamic-finger theorem say that this sequence takes $O(m \log n)$ time. However, these bounds are not tight: this sequence executes in $O(m)$ time in splay trees. This fact can be seen by proving a new theorem, based on the sequential access lemma. However, introducing new theorems that bound the running times of highly specific classes of sequences such as X_3 will only contribute to our fragmented understanding of splay trees. In an attempt to more accurately characterize the running time of access sequences on splay trees, we provide the following conjecture:

Conjecture 1 (Unified Conjecture)[5] *The time to search for, insert, or delete x_i in a splay tree is*

$$O\left(\min_{y \in S_i} \log \Big(t_i(y) + d_i(x_i, y) + 2\Big)\right).$$

This conjecture implies the working-set theorem and the dynamic-finger theorem, and it is strong enough to predict that X_3, and many possible variants, run in $O(m)$ time. Informally, the unified bound says that an access is fast if the access is close in key space to some element that has been accessed recently in time. In the case of X_3, the majority of the accesses are to elements that are at rank distance 1 away from the element accessed two accesses ago, so the amortized cost per access is $O(\log(1 + 2 + 2)) = O(1)$. We offer no proof of this conjecture.

Our main result is a relatively complicated data structure, called the *unified structure*, whose performance satisfies the unified bound. This structure demonstrates the plausibility of the unified conjecture for splay trees. It also has a worst-case running time of $O(\log n)$ per access, in contrast to the $\Theta(n)$ attained by splay trees, where n denotes the current value of n_i, the size of the set S_i. We present this structure in Section 3.

In terms of proved bounds on the running time of a comparison-based search structure, the unified structure is strictly better than splay trees. However, this is not true in terms of actual amortized performance: there are access sequences that splay trees executes asymptotically faster than the unified structure. For example, consider scaling each element in X_3 by a superconstant factor α, forming an access sequence over a set that is α times as large. Splay trees can factor out the intervals of $\alpha - 1$ unaccessed elements, but the unified bound does not capture this feature. However, we do not know whether the unified structure executes any access sequences asymptotically faster than splay trees; this requires resolving the unified conjecture.

[5] Note that a "unified bound" for splay trees is presented in [ST85], which is simply the minimum of the static-optimality, static-finger, and working-set bounds. This theorem is distinct from the conjecture presented here.

The dynamic optimality conjecture of Sleator and Tarjan [ST85] states that splay trees can execute any sufficiently long access sequence as fast as any rotation-based binary search tree, up to constant factors. The unified structure is composed of a collection of search trees, not one search tree, so we are unable to derive any statements about dynamic optimality from the results presented here. In particular, assuming the dynamic optimality conjecture does not imply the unified conjecture for splay trees. Conversely, a disproof of the unified conjecture for splay trees would not disprove the dynamic optimality conjecture.

2 Working-Set Structure

The working set structure consists of a set of $O(\log\log|S_i|)$ 2-4 trees, $T_0, T_1, \ldots, T_l$, and linked lists $L_0, L_1, \ldots, L_l$. Let L be the imaginary list which would be created by appending all of the linked lists together in index order. Let $w(x)$ denote the number of distinct elements accessed since the last access to x. The structure maintains the following invariants:

- Every element in S_i is stored in exactly one tree and exactly once in the corresponding linked list.
- For $k < l$, $\sum_{j=0}^{k} |T_j| = 2^{2^k}$.
- $\sum_{j=0}^{l} |T_j| \leq 2^{2^l}$.
- If $w(x) = h$, x appears as the hth element of L.

From these facts, we can also deduce that if $w(x) = h$ it will appear in tree $T_{\lceil \log\log h \rceil}$.

Search(x). We search for x in $T_0, T_1, \ldots$ in turn until it is found in some tree T_k. We know from the observation above that $k = \lceil \log\log w(x) \rceil$. Since x is now the most recently accessed item, it must be removed from $T_{\lceil \log\log w(x) \rceil}$ and $L_{\lceil \log\log w(x) \rceil}$ and inserted into T_1 and the front of L_1. Now, observe that T_1 is too large by one element, and $T_{\lceil \log\log w(x) \rceil}$ is too small by one element. We then proceed, for each j in the range 1 to $\lceil \log\log w(x) \rceil - 1$, to remove the oldest element from T_j and L_j and insert it into T_{j+1} and to the front of L_{j+1}. The oldest element in L_j is the last element in the list.

The running time is dominated by the tree operations. For each tree in the range 1 to $\lceil \log\log w(x) \rceil$, one insert, one delete, and one search are performed, at a total cost of $O(\sum_{j=1}^{\lceil \log\log w(x) \rceil} \log 2^{2^j}) = O(\log w(x))$.

Insert(x). If $\sum_{j=0}^{l} |T_j| < 2^{2^l}$, insert x in T_l at cost $O(\log n)$ and append x to the end of L_l at cost $O(1)$. If $\sum_{j=0}^{l} |T_j| = 2^{2^l}$, we increment l and initialize the new T_l and L_l to contain only x at cost $O(1)$. We then move x to T_0 as in the search operation.

Delete(x). Suppose x is in T_k. We delete it from T_k and L_k. Now, unless x was in the last tree, we must correct the size of T_k using a procedure analogous to the one used in the search operation. For each j in the range from k to $\lceil \log\log n \rceil - 1$, remove the newest element from T_{j+1} and L_{j+1} and insert it into T_j and to the back of L_j. The newest element in L_{j+1} is the first element in this list.

The running time is dominated by the tree operations. For each tree in the range k to $\lceil \log\log n \rceil$, one insert, one delete, and one search are performed, at a total cost of $O(\sum_{j=k}^{\lceil \log\log n \rceil} \log 2^{2^j}) = O(\log n)$.

Theorem 2 *The working-set structure supports searching for element x in $O(\log(w(x) + 2))$ worst-case time, and supports inserting and deleting an element in $O(\log n)$ worst-case time.*

3 Unified Structure

In this section, we develop our dynamic unified structure, establishing the following theorem:

Theorem 3 *There is a dynamic data structure in the comparison model on a pointer machine that supports insertions, deletions, and searches within the unified bound (amortized).*

The bulk of our unified structure consists of $O(\log\log |S_i|)$ balanced binary trees, $T_0, T_1, \ldots, T_\ell$. Each tree T_j has size 2^{2^j} whenever it is rebuilt, and is maintained to have at most $2^{2^{j+1}} + 2^{2^j} + \cdots + 2^{2^0}$ elements at all times. Furthermore, at the end of each dictionary operation in the unified structure, tree T_j will have at most $2^{2^{j+1}}$ elements. Each element is augmented with a timestamp of when it was last accessed (searched or inserted). Each element of the structure appears in at most one tree T_j at any time. The structure is maintained so that smaller trees contain more recently accessed elements, i.e., all elements in T_j were accessed more recently than all elements in T_k for all $j < k$.

We can store each tree T_k using any balanced search tree structure supporting insertions, deletions, and searches in $O(\log |T_k|) = O(2^k)$ time, and supporting

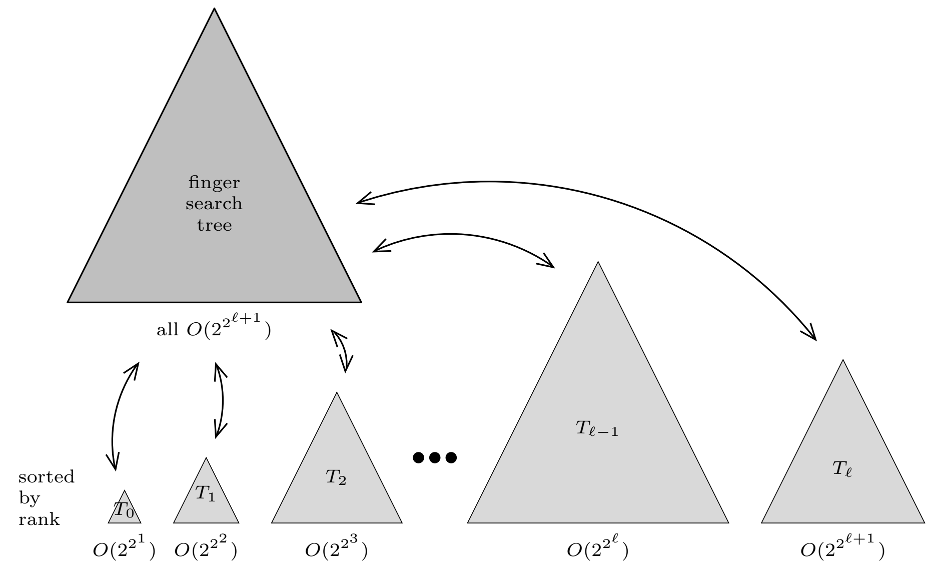

Fig. 1. Overview of our dynamic unified structure. In addition to a single finger search tree storing all elements in the dynamic set S_i, there are $\ell + 1 \approx \log\log|S_i|$ balanced search trees whose sizes grow doubly exponentially. (As drawn, the heights accurately double from left to right.)

insertions and deletions with a pointer to the relevant location in the tree in $O(1)$ amortized time. For example, B-trees [BM72] support these time bounds.

Our unified structure also stores a single finger search tree containing all elements of S_i. We can use any finger search tree structure supporting insertions, deletions, and searches within rank distance r of a previously located element in $O(\log(r+2))$ amortized time, e.g., level-linked B-trees [BT80]. We represent each element in the set S_i by a separate *indirect node*, with pointers between this indirect node and the node of the finger search tree that currently stores the element. This indirection is necessary because the elements may move from node to node as the finger search tree changes; during such changes, we can easily maintain the pointers from the indirect node into the finger search tree. Also, we store pointers between each indirect node and the node in one of the trees T_k, if any, that stores the element. In this way, we can quickly cross-index between elements as stored in the trees $T_0, T_1, \ldots, T_\ell$ and as stored in the finger search tree.

3.1 Potential Function

We use the potential method to analyze the amortized running time of each operation in our structure. The potential function has two components, death potential and overflow potential.

The *death potential* of the structure at a given time is four times the size of all of the trees times a constant c which will be defined later: $4c\sum_{j=0}^{\ell}|T_j|$.

To define the overflow potential, we introduce the *j-graph*, defined as follows. The nodes in the j-graph consist of all the nodes in $T_0, \ldots, T_j$. There is an edge in the j-graph between every pair of nodes of rank difference at most $2^{2^{j+1}}$. We define the *j-components* to be the connected components of the j-graph. We define the *extent* of a j-component to be the rank difference between the smallest and largest items in the j-component.

The overflow potential of the structure comprises several terms. The overflow potential of an individual j-component with extent e is $4c \cdot 2^j[1+e/2^{2^{j+1}}]$. The motivation behind this definition is that we will overflow roughly $\lfloor 1+e/2^{2^{j+1}} \rfloor$ items from a j-component from tree T_j to T_{j+1}, and each such item will cost $\Theta(2^j)$. The j-overflow potential is the sum of the overflow potentials of each j-component. The *potential* of the entire structure is the sum of the j-overflow potentials for $j = 0, 1, \ldots, \ell$.

Lemma 4 *Removing an item x from T_k cannot increase the overflow potential.*

PROOF. For $j < k$, the j-overflow potential does not change. For a given $j \geq k$, the loss of x can cause one of three things to happen. Let C denote the j-connected component that contains x.

Case 1: The extent of C remains the same. No change.

Case 2: The extent of C shrinks. There is a loss in potential.

Case 3: The removal of x causes C to split into C_l and C_r. For this to happen, the rank gap between C_l and C_r after the removal of x must be more than $2^{2^{j+1}}$. If $e(C)$ denotes the extent of C before removal of x, and $e(C_l)$ and $e(C_r)$ denote the extents of C_l and C_r after the removal of x, then $e(C_l) + e(C_r) \leq e(C) - 2^{2^{j+1}}$. Thus the total rank potential of C_l and C_r after the removal of x is $4c \cdot 2^j[2+[e(C_l)+e(C_r)]/2^{2^{j+1}}] \leq 4c \cdot 2^j[1+e(C)/2^{2^{j+1}}]$, so the rank potential does not increase. $\square$

3.2 Overflow

The overflow is an important subroutine that will be used in the implementation of the search and insert operations. It has 0 amortized cost. The idea is to fix a tree that has grown too large by rebuilding the tree, and the smaller trees, to have the desired minimal size (2^{2^j} for tree T_j). We must then decide what to do with the extra nodes left over from this rebuilding. We cannot afford to insert all of them into the next larger tree. Instead we retain only those that would be necessary to help with future searches.

3.2.1 Description

We overflow at level k when T_k grows to size at least $2^{2^{k+1}}$. The overflow operation restores the size of the trees T_j, $0 \leq j \leq k$, to be 2^{2^j}, and inserts some excess items into T_{k+1}. These insertions may trigger (after completion of this overflow) an overflow at the next larger level.

An overflow at level k works as follows. Take all the items in levels $T_0, \ldots, T_k$. Populate each tree T_j, $0 \leq j \leq k$ with the 2^{2^j} most recent items not in a tree T_h with $h < j$. Of the remaining items, remove every item for which there is another smaller item within rank distance $2^{2^{k+1}}$. Insert the still remaining items into T_{k+1}. In order to obtain the proper running time, this overflow algorithm must be carried out with some care. Algorithm 1 gives pseudocode that efficiently implements the overflow algorithm.

```
 1: $L \leftarrow$ empty linked list
 2: **for** $j = 0$ to $k$ **do**
 3:     Merge $L$ with an in-order traversal of the items in $T_j$.
 4:     Store the resulting list, sorted by key value, in $L$.
 5: **end for**
 6: Let $a$ be the $\left(\sum_{j=0}^{k} 2^{2^j}\right)$th youngest timestamp among items in $L$.
 7: $L_{\text{overflow}} \leftarrow$ list of items from $L$ with timestamp older than $a$.
 8: $L \leftarrow$ list of items from $L$ with timestamp at least as young as $a$.
 9: **for** $j = k$ downto 0 **do**
10:     Let $a$ be the $2^{2^j}$th oldest timestamp among items in $L$.
11:     $L_{T_j} \leftarrow$ list of items from $L$ with timestamp at least as old as $a$.
12:     $L \leftarrow$ list of items from $L$ with timestamp younger than $a$.
13: **end for**
14: **for** $j = 0$ to $k$ **do**
15:     Build the new tree $T_j$ using the contents of $L_{T_j}$.
16: **end for**
17: **for** $x$ in $L_{\text{overflow}}$, except the first item **do**
18:     If the rank of $x$ minus the rank of the previous remaining item in
            $L_{\text{overflow}}$ is $\leq 2^{2^{k+1}}$, then remove $x$ from $L_{\text{overflow}}$.
19: **end for**
20: **for** $x$ in $L_{\text{overflow}}$ **do**
21:     Insert $x$ into $T_{k+1}$.
22: **end for**
23: If $T_{k+1}$ now has size at least $2^{2^{k+2}}$, then overflow at level $k + 1$.
```

Algorithm 1. Pseudocode for the overflow algorithm at level k.

3.2.2 Analysis

The elements that were in $T_0, \ldots, T_k$ will fall into three categories: r elements that remain in $T_0, \ldots, T_k$, p elements that are added to T_{k+1}, and d elements that are deleted.

Lemma 5 *(i)* $r + d + p \geq |T_k| \geq 2^{2^{k+1}}$.
(ii) $r = 2^{2^k} + \cdots + 2^{2^0}$.
(iii) $p \leq d + p < 2^{2^{k+1}} + 2^{2^k} + \cdots + 2^{2^0}$.
(iv) The invariant on $|T_{k+1}|$, $|T_{k+1}| \leq 2^{2^{k+2}} + 2^{2^{k+1}} + \cdots + 2^{2^0}$, *remains true after the overflow of* p *items into* T_{k+1}.

PROOF. (i) is immediate because the total $r + d + p$ is the number of elements originally in $T_0, \ldots, T_k$, which is at least the number of items in the overflowing T_k. (ii) holds by construction.

(iii) can be obtained as follows. Because T_k is the tree overflowing (and $T_0, \ldots, T_{k-1}$ are not), $|T_j| < 2^{2^{j+1}}$ for $j < k$. Also, by the invariant on $|T_k|$, $|T_k| \leq 2^{2^{k+1}} + 2^{2^k} + \cdots + 2^{2^0}$. Thus $r+d+p = |T_0| + \cdots + |T_k| < 2^{2^{k+1}} + 2 \cdot (2^{2^k} + \cdots + 2^{2^0})$. Substituting for r from (ii) yields (iii).

(iv) follows because at most $p \leq 2^{2^{k+1}} + 2^{2^k} + \cdots + 2^{2^0}$ elements are inserted into T_{k+1}; even if these insertions into T_{k+1} push $|T_{k+1}|$ beyond $2^{2^{k+2}}$, we still have that $|T_{k+1}| \leq 2^{2^{k+2}} + 2^{2^{k+1}} + \cdots + 2^{2^0}$. $\square$

For technical reasons, the following analysis works only for $k \geq k_0$ for a suitable constant k_0. (The exact constraints on k are pointed out below.) For $k < k_0$, the amortized cost can easily be shown to be $O(1)$.

Actual cost. We claim that the actual running time is at most $c(r+d+p \cdot 2^k)$ for an appropriately chosen constant c.

The merging of all of the trees (Lines 1–5) takes time at most $\sum_{j=0}^{k} O(|T_j| + \sum_{h=0}^{j-1} |T_h|) = \sum_{j=0}^{k} O(2^{2^{j+1}} + \sum_{h=0}^{j-1} 2^{2^{h+1}}) = \sum_{j=0}^{k} O(2^{2^{j+1}}) = O(2^{2^{k+1}}) = O(r + d + p)$.

The order statistic on Line 6 takes $O(r + d + p)$ time by transferring the items into an array and using linear-time median finding. The filtering on Lines 7–8 also takes $O(r + d + p)$ time.

The loop on Lines 9–13 takes time $O(\sum_{j=0}^{k} \sum_{h=0}^{j} 2^{2^h}) = O(2^{2^k}) = O(r)$.

Because a balanced binary tree can be built in linear time from a sorted list, the loop on Lines 14–16 take time $O(\sum_{j=0}^{k} 2^{2^j}) = O(2^{2^k}) = O(r)$.

Removing the data to be deleted on Lines 17–19 takes $O(|L_{\text{overflow}}|) = O(d+p)$ time. Inserting the remaining p items in Lines 20–22 into T_{k+1} which has size $O(2^{2^{k+2}})$ takes $O(p \cdot 2^k)$ time.

Thus, the actual cost is $O(r + d + p \cdot 2^k)$, or by choosing c sufficiently large, at most $c \cdot (r + d + p \cdot 2^k)$.

Change in potential. The death potential is reduced by $4cd$.

For $j > k$, the j-overflow potential does not increase. This claim follows by Lemma 4 because, from the point of view of the j-graph, we have just deleted d items.

For $j \le k$, we analyze the change in j-overflow potential as if the overflow first deleted all of $T_0, \ldots, T_k$, then added the p elements to T_{k+1}, then rebuilt $T_1 \ldots T_k$.

Deleting all of the elements will result in a loss of at least $4cp2^k$ units in the k-overflow potential. The reason is that, for each k-component of extent e, we will insert at most $\lfloor 1 + e/2^{2^{k+1}} \rfloor$ elements into T_{k+1} (this would be exact if we did not populate $T_0, \ldots, T_k$ with r elements). Thus $p \le \sum_C \lfloor 1 + e(C)/2^{2^{k+1}} \rfloor$, and so $4cp2^k$ is less than or equal to the old k-overflow potential.

Adding the p elements to T_{k+1} does not change the j-overflow potential for $j \le k$.

For each $j \le k$, re-inserting the $\sum_{h=0}^{j} 2^{2^h}$ items into the j-graph could cause, in the worst-case, each of them to be in a separate j-component. In this case there could be a j-overflow potential increase of up to $\sum_{h=0}^{j} 4c \cdot 2^j 2^{2^h} \le 6c \cdot 2^j 2^{2^j}$. The total increase in overflow potential, for $j \le k$, is thus at most $\sum_{j=0}^{k} 6c \cdot 2^j 2^{2^j} \le 7.5 \cdot c \cdot 2^k 2^{2^k}$.

Amortized cost. Therefore the gain in potential is at most $7.5c2^k2^{2^k} - 4cp2^k - 4cd \le 7.5c2^k2^{2^k} - cp2^{k+1} - cd - 2c(p+d)$. Now, $p + d \ge |T_k| - r \ge 2^{2^{k+1}} - \sum_{j=0}^{k} 2^{2^j}$. Also, $7.5 \cdot 2^k 2^{2^k} \le 2^{2^{k+1}} - \sum_{j=0}^{k} 2^{2^j}$ for $k \ge 3$. Thus the gain in potential is at most $-cp2^{k+1} - cd - c(p+d)$. Because $p + d \ge r$ ($2^{2^{k+1}} - \sum_{j=0}^{k} 2^{2^j} \ge \sum_{j=0}^{k} 2^{2^j}$ for all $k \ge 0$), the gain in potential is at most $-cp2^{k+1} - cd - cr \le -c(p2^k + d + r)$. This is the negation of the actual cost of the overflow. Thus we have shown

Lemma 6 *The amortized cost of overflow is at most* 0.

3.3 Search

3.3.1 Description

Up to constant factors, the unified property requires us to find an element $x = x_i$ in $O(2^k)$ time if it is within rank distance 2^{2^k} of an element y with working-set number $t_i(y) \leq 2^{2^k}$. The data structure maintains the invariant that all such elements x are within rank distance $(k+4) \cdot 2^{2^k}$ of some element y' in $T_0 \cup T_1 \cup \cdots \cup T_k$. (This invariant is proved below in Lemma 7.)

At a high level, then, our search algorithm will investigate the elements in $T_0, T_1, \ldots, T_k$ and, for each such element, search among the elements within rank distance $(k+4) \cdot 2^{2^k}$ for the query element x. The algorithm cannot perform this procedure exactly, because it does not know k. Thus we perform the procedure for each $k = 0, 1, 2, \ldots$ until success. To avoid repeated searching around the elements in T_j, $j \leq k$, we maintain the two elements so far encountered among these T_j's that straddle the target x, and just search inside those two elements. If any of the searches from any of the elements would be successful, one of these two searches will be successful. After finding x in the finger structure, we are able to obtain a pointer to the tree, call it T_p, that x is in (if it is in a tree). We remove x from T_p using the pointer and then insert it into T_0.

More precisely, our algorithm to search for an element x proceeds as shown in Algorithm 2. The variables L and U store pointers to elements in the finger search tree such that $L \leq x \leq U$. These variables represent the tightest known bounds on x among elements that we have located in the finger search tree as predecessors and successors of x in $T_0, T_1, \ldots, T_k$. In each round, we search for x in the next tree T_k, and update L and/or U if we find elements closer to x. Then we search for x in the finger search tree within rank distance $(k+4) \cdot 2^{2^k}$ of L and U.

3.3.2 Unified Invariant

Lemma 7 *All elements within rank distance 2^{2^k} of an element y with working-set number $t_i(y) \leq 2^{2^k}$ are within rank distance $(k+4) \cdot 2^{2^k}$ of some element y' in $T_0 \cup T_1 \cup \cdots \cup T_k$.*

PROOF. We consider the time interval between y's last access (before time i) and time i, which consists of $t_i(y) \leq 2^{2^k}$ distinct accesses. During this time interval, we track the motion of an element y', initially y, through the trees $T_0, T_1, \ldots, T_\ell$. Initially, because $y' = y$ was just accessed, y' is in T_0. The only time at which we change the element y' being tracked is when overflowing T_j

Algorithm 2. To search for an element x.

```
• Initialize $L \leftarrow -\infty$ and $U \leftarrow \infty$.
• For $k = 0, 1, 2, \ldots, \log\log n$:
(1) If $k \leq \ell$:
    (a) Search for $x$ in $T_k$ to obtain two elements $L_k$ and $U_k$ in $T_k$ such that
        $L_k \leq x \leq U_k$.
    (b) Update $L \leftarrow \max\{L, L_k\}$ and $U \leftarrow \min\{U, U_k\}$.
(2) Finger search for $x$ within the rank ranges $[L, L + (k+4) \cdot 2^{2^k}]$ and $[U -
    (k+4) \cdot 2^{2^k}, U]$.
(3) If we find $x$ in the finger search tree:
    (a) Delete $x$ from whatever tree $T_p$ contains it, if any (found using the
        pointer in the finger search tree).
    (b) Insert $x$ into tree $T_0$.
    (c) If $T_0$ is too full (storing $2^{2^1}$ elements), *overflow* $T_0$ as described in
        Algorithm 1.
    (d) Return a pointer to $x$ in the finger search tree.
```

causes y' to be discarded, in which case we continue by tracking the element within rank distance $2^{2^{j+1}}$ of y' that gets promoted to tree T_{j+1}. Each such "jump" of the tracked element changes the rank of y', and hence increases the rank distance between y and y', but by at most $2^{2^{j+1}}$.

The tracked element y' may switch trees for several reasons: it may be accessed, in which case it returns to T_0; it may move to a smaller tree because of an overflow (if smaller trees were undersized); and it may move to the next larger tree because of an overflow (and either it was promoted or it was deleted and then the tracked element changed to a different, promoted element). Only the last case can cause a jump in y'. This last case can happen relatively easily once in each T_j, if T_j was already near overflowing at the beginning of the time interval. However, for the same tree T_j to overflow more than once in the time interval, there must be accesses to at least $2^{2^{j+1}} - 2^{2^j}$ distinct elements in between every two consecutive overflows.

Suppose that y' overflows o_j times from tree T_j during the time interval. First we observe that $o_k = o_{k+1} = \cdots = o_\ell = 0$, because y' remains one of the 2^{2^k} youngest elements during the time interval, so it must remain in $T_0, T_1, \ldots, T_k$ during an overflow of T_k, and thus could not reach $T_{k'}$ for $k' > k$. As argued above, if $o_j > 1$, there must be $(j-1)(2^{2^{j+1}} - 2^{2^j})$ distinct accesses that cause T_j to overflow. (However, the same accesses may cause overflows at several levels.) Because the total number of distinct accesses in the time window is at most 2^{2^k}, for any $j < k$,

$$(o_j - 1)(2^{2^{j+1}} - 2^{2^j}) \leq 2^{2^k}.$$

It follows that

$$(o_j - 1)2^{2^{j+1}}(1 - 1/2^{2^j}) \leq 2^{2^k}.$$

The total distance that y' may jump over the course of the time interval is at most

$$\begin{aligned}
\sum_{j=0}^{k-1} o_j 2^{2^{j+1}} &= \sum_{j=0}^{k-1} 2^{2^{j+1}} + \sum_{j=0}^{k-1}(o_j - 1)2^{2^{j+1}} \\
&\leq \sum_{j=1}^{k} 2^{2^j} + \sum_{j=0}^{k-1} \frac{2^{2^k}}{1 - 1/2^{2^j}} \\
&\leq 1.25 \cdot 2^{2^k} + 2^{2^k} \sum_{j=0}^{k-1} \frac{2^{2^j}}{2^{2^j} - 1} \\
&= 1.25 \cdot 2^{2^k} + 2^{2^k} \sum_{j=0}^{k-1} \left(1 + \frac{1}{2^{2^j} - 1}\right) \\
&\leq 1.25 \cdot 2^{2^k} + 2^{2^k}(k + 1.5) \\
&\leq (k+3) \cdot 2^{2^k}.
\end{aligned}$$

Finally, insertions between y' and the original value y can increase the rank distance between y and y' by an additional 2^{2^k}. Deletions only decrease the distance. □

3.3.3 Analysis

In Section 3.2, we showed that the amortized cost of an overflow is nonpositive. Therefore we only need to analyze the operations other than the overflow. There are two quantities we must examine in order to bound the amortized cost. The first quantity is the actual running time of the operation. The second quantity is the change of potential caused by inserting x into T_0 and possibly removing it from another tree T_p.

Actual cost. By Step 2 of the algorithm, if x is within rank distance $(k+4) \cdot 2^{2^k}$ of an element in $T_0 \cup T_1 \cup \cdots \cup T_k$, then the search algorithm will complete in round k. The actual total running time of k rounds is $\sum_{j=0}^{k} O(\log |T_j|) = O(2^k)$. The insertion and possible deletion take $O(1)$ time, since we have pointers to where the item will be deleted and inserted. Thus, the search algorithm attains the unified bound, provided we have the invariant in Lemma 7 above.

Potential change. Recall that we do not need to consider the effects of the overflow.

There is no change in death potential.

For $j \leq k$, for each j-graph, in the worst case a new connected component is formed; it has potential $O(2^j)$. This gives a total potential gain of $\sum_{j=0}^{k} O(2^j) = O(2^k)$.

Call y the item in tree T_k that was used as the starting point of the successful finger search. For $j > k$, we note that x will always appear in the same j-component as y. The growth of these components gives a potential gain of at most $\sum_{j=k+1}^{\infty} 4c2^j \frac{1}{2^{2^{j+1}}} = O(1)$.

Summary. The amortized cost of a search is given by the unified bound.

3.4 Insert

To perform an insertion, we search for the element as in Algorithm 2, but stop when we find the predecessor. Then we add the new element into T_0 (and overflow as usual). This new element appears in the same connected component as its predecessor, and thus it increases the overflow potential by $O(1)$ (as in the search analysis). Also, the increase in death potential is $O(1)$.

Thus the running time is dominated by the search for the predecessor, so the amortized cost of the insert operation is the unified bound for the predecessor, which is within a constant factor of the unified bound for the element itself.

3.5 Delete

To delete an element, we search for it as in Algorithm 2; once it is found, we simply delete it from the tree T_k, if any, in which we find it, and also from the finger search tree.

The actual cost can be bounded as follows. One or two tree deletions are performed at $O(1)$ amortized cost each, because the search gives us pointers to the nodes to be deleted. Suppose that the search terminated in tree T_k, so that the cost for the search is $O(2^k)$.

Next we consider the change in potential. The death potential decreases. We claim that the overflow potential also does not increase. By Lemma 4, removing the element from the tree does not increase the potential. Removing the element from the set of stored elements causes the rank difference between some elements to decrease. Thus additional edges may appear in some of the j-graphs. However, these edges do not cause any combining of connected components because, if a j-edge from w to z now appears, there must have already been a j-edge from w to the deleted element x and a j-edge from x to z. Thus the potential does not increase.

Therefore the amortized cost of a deletion is within a constant factor of the unified bound.

Acknowledgments

We thank Michael L. Fredman for helpful discussions.

References

[BD04] Mihai Bădoiu and Erik D. Demaine. A simplified and dynamic unified structure. In *Proceedings of the 6th Latin American Symposium on Theoretical Informatics*, volume 2976 of *Lecture Notes in Computer Science*, pages 466–473, Buenos Aires, Argentina, April 2004.

[BM72] Rudolf Bayer and Edward M. McCreight. Organization and maintenance of large ordered indexes. *Acta Informatica*, 1(3):173–189, February 1972.

[BT80] Mark R. Brown and Robert Endre Tarjan. Design and analysis of a data structure for representing sorted lists. *SIAM Journal on Computing*, 9(3):594–614, 1980.

[CMSS00] Richard Cole, Bud Mishra, Jeanette Schmidt, and Alan Siegel. On the dynamic finger conjecture for splay trees. Part I: Splay sorting $\log n$-block sequences. *SIAM Journal on Computing*, 30(1):1–43, 2000.

[Col00] Richard Cole. On the dynamic finger conjecture for splay trees. Part II: The proof. *SIAM Journal on Computing*, 30(1):44–85, 2000.

[Elm04] Amr Elmasry. On the sequential access theorem and deque conjecture for splay trees. *Theoretical Computer Science*, 314(3):459–466, April 2004.

[GMPR77] Leo J. Guibas, Edward M. McCreight, Michael F. Plass, and Janet R. Roberts. A new representation for linear lists. In *Conference Record of the 9th Annual ACM Symposium on Theory of Computing*, pages 49–60, Boulder, Colorado, May 1977.

[HT71] T. C. Hu and A. C. Tucker. Optimal computer search trees and variable-length alphabetic codes. *SIAM Journal on Applied Mathematics*, 21(4):514–532, December 1971.

[Iac01a] John Iacono. Alternatives to splay trees with $O(\log n)$ worst-case access times. In *Proceedings of the 12th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 516–522, Washington, D.C., January 2001.

[Iac01b] John Iacono. *Distribution Sensitive Data Structures.* PhD thesis, Rutgers, The State University of New Jersey, New Brunswick, New Jersey, 2001.

[Knu71] Donald E. Knuth. Optimum binary search trees. *Acta Informatica*, 1:14–25, 1971.

[ST85] Daniel Dominic Sleator and Robert Endre Tarjan. Self-adjusting binary search trees. *Journal of the ACM*, 32(3):652–686, July 1985.

[Sun91] Rajamani Sundar. *Amortized Complexity of Data Structures.* PhD thesis, New York University, 1991.

[Sun92] Rajamani Sundar. On the deque conjecture for the splay algorithm. *Combinatorica*, 12:95–124, 1992.

[Tar85] R. E. Tarjan. Sequential access in splay trees takes linear time. *Combinatorica*, 5(4):367–378, September 1985.