A Simplified and Dynamic Unified Structure

Mihai Bădoiu and Erik D. Demaine

MIT Computer Science and Artificial Intelligence Laboratory,
200 Technology Square, Cambridge, MA 02139, USA, {mihai,edemaine}@mit.edu

Abstract. The unified property specifies that a comparison-based search structure can quickly find an element nearby a recently accessed element. Iacono [Iac01] introduced this property and developed a static search structure that achieves the bound. We present a dynamic search structure that achieves the unified property and that is simpler than Iacono's structure. Among all comparison-based dynamic search structures, our structure has the best proved bound on running time.

1 Introduction

The classic *splay conjecture* says that the amortized performance of splay trees [ST85] is within a constant factor of the optimal dynamic binary search tree for any given request sequence. 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 nearby 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 $n - 1$ 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.

In SODA 2001, Iacono [Iac01] proposed a *unified bound* (defined below) that is strictly stronger than all other proved bounds about comparison-based structures. Roughly, the unified bound says that it is cheap to access an element that is nearby 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, Iacono [Iac01] developed the *unified structure* which attains the unified bound. Among all comparison-based data structures, this structure has the best proved bound on running time.

The only shortcomings of the unified data structure are that it is static (keys cannot be inserted or deleted), and that both the algorithms and the analysis are complicated. We improve on all of these shortcomings with a simple dynamic unified structure. Among all comparison-based dynamic data structures, our structure has the best proved bound on running time.

2 Unified Property

Our goal is to maintain a dynamic set of elements from a totally ordered universe in the (unit-cost) comparison model on a pointer machine. Consider a sequence of m operations—insertions, deletions, and searches—where the ith operation involves element x_i. Let S_i denote the set of elements in the structure just before operation i (at time i). Define the *working-set number* $t_i(z)$ of an element z at time i to be the number of distinct elements accessed since the last access to z and prior to time i, including z. Define the *rank distance* $d_i(x, y)$ between elements x and y at time i to be the number of distinct elements in S_i that fall between x and y in rank order. A data structure has the *unified property* if the amortized cost of operation i is $O(\lg \min_{y \in S_i}[t_i(y) + d_i(x_i, y) + 2])$, the *unified bound.* Intuitively, the unified bound for accessing an element x_i is small if any element y is nearby x in both time and space.

3 New Unified Structure

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

Theorem 1. *There is a dynamic data structure in the comparison model on a pointer machine that supports insertions and searches within the unified bound and supports deletions within the unified bound plus* $O(\lg \lg |S_i|)$ *time (amortized).*

An interesting open problem is to attain the unified bound for all three operations simultaneously.

3.1 Data Structure

The bulk of our unified structure consists of $\Theta(\lg \lg |S_i|)$ balanced search trees and linked lists whose sizes increase doubly exponentially; see Fig. 1. Each tree T_k, $k \geq 0$, stores between 2^{2^k} and $2^{2^{k+1}} - 1$ elements, ordered by their rank, except that the last tree may have fewer elements. We can store each tree T_k using any balanced search tree structure supporting insertions, deletions, and searches in $O(\lg |T_k|)$ time, e.g., B-trees [BM72]. List L_k stores exactly the same

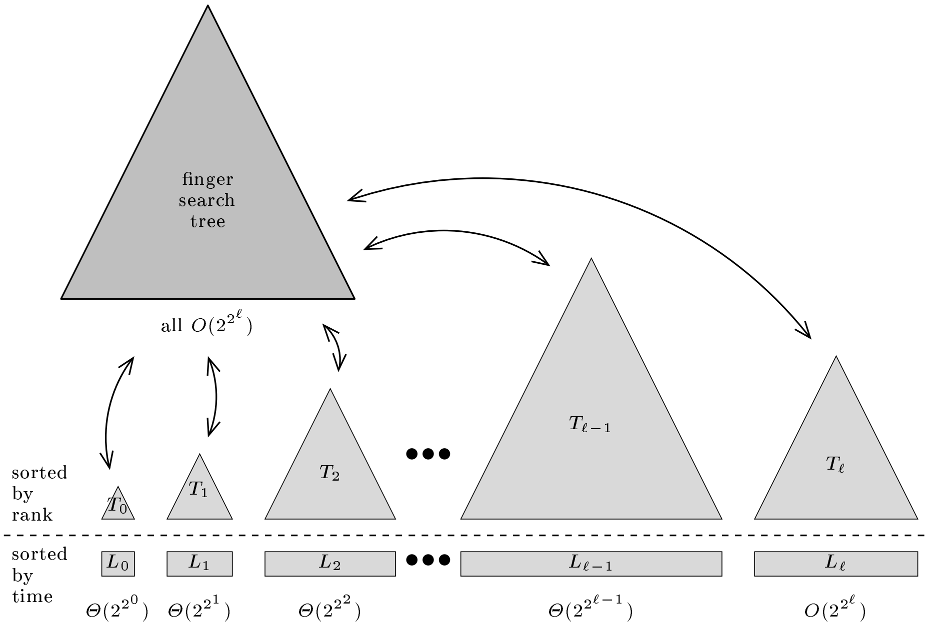

Fig. 1. Overview of our dynamic unified structure. In addition to a single finger search tree storing all elements in the dynamic set S_i, there are $\ell + 1 = \Theta(\lg \lg |S_i|)$ balanced search trees and lists whose sizes grow doubly exponentially. (As drawn, the heights accurately double from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

elements stored in T_k, but ordered by the time of access. We store pointers between corresponding nodes in T_k and L_k.

Each element x may be stored by several nodes in various trees T_k, or possibly none at all, but x appears at most once in each tree T_k. Each tree node storing element x represents an access to x at a particular time. At most one tree node represents each access to x, and some accesses to x have no corresponding tree node. We maintain the invariant that the access times of nodes in tree T_k are all more recent than access times of nodes in tree T_{k+1}. Thus, the concatenation of corresponding nodes in lists $L_0, L_1, L_2, \ldots$ is also ordered by access time.

Our unified structure also stores a single finger search tree containing all n elements. We can use any finger search tree structure supporting insertions, deletions, and searches within rank distance r of a previously located element in $O(\lg(r + 2))$ amortized time, e.g., level-linked B-trees [BT80]. Each node in tree T_k stores a pointer to the unique node in this finger search tree corresponding to the stored element.[1]

[1] In fact, because nodes in the finger search tree may move (e.g., from a B-tree split), each node in tree T_k stores a pointer to an indirect node, and each indirect node is connected by pointers to the corresponding node in the finger search tree. The

3.2 Search

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 $t_i(y) \le 2^{2^k}$. We maintain the invariant that all such elements x are within rank distance $3 \cdot 2^{2^k}$ of some element y' in $T_0 \cup T_1 \cup \cdots \cup T_k$. (This invariant is proved below in Lemma 1.)

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 $3 \cdot 2^{2^k}$ 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, $j \le k$, we maintain the two elements so far encountered among these T_j's that are closest to the target x, and just search around those two elements. If any of the searches from any of the elements would be successful, one of these two searches will be successful.

More precisely, our algorithm to search for an element x proceeds as shown in Algorithm 1. The variables L and U store pointers to elements in the finger search tree such that $L \le x \le U$. 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 $3 \cdot 2^{2^k}$ of L and U.

Thus, if x is within rank distance $3 \cdot 2^{2^k}$ of an element in $T_0 \cup T_1 \cup \cdots \cup T_k$, then the search algorithm will complete in round k. The total running time of k rounds is $\sum_{i=0}^{k} O(\lg |T_i|) = O(2^k)$. Thus, the search algorithm attains the unified bound, provided we have the invariant in Lemma 1 below.

When the search algorithm finds x, it records this most recent access by inserting a node storing x into the smallest tree T_0. This insertion may cause T_0 to grow too large, triggering the overflow algorithm described next.

3.3 Overflow

It remains to describe what we do when a tree T_k becomes too full; see Algorithm 2. The main idea is to promote all but the most recent 2^{2^k} elements from T_k to T_{k+1}, by repeated insertion into T_{k+1} and deletion from T_k. In addition, we discard elements that would be promoted but are within $2^{2^{k+1}}$ of other promoted elements. Such discards are necessary to prevent excessive overflows in the future. The intuition of why discards do not substantially slow future searches is that, for the purposes of searching for an element x within rank distance $2^{2^{k+1}}$ of elements in T_{k+1}, it is redundant up to a factor of 2 to have more than one

former pointers never change, and the latter pointers can be easily maintained when nodes move in the finger search tree.

Algorithm 1. Searching for an element x.

- Initialize $L \leftarrow -\infty$ and $U \leftarrow \infty$.
- For $k = 0, 1, 2, \ldots$:[a]

1. Search for x in T_k to obtain two elements L_k and U_k in T_k such that $L_k \le x \le U_k$.
2. Update $L \leftarrow \max\{L, L_k\}$ and $U \leftarrow \min\{U, U_k\}$.
3. Finger search for x within the rank ranges $[L, L+3\cdot 2^{2^k}]$ and $[U-3\cdot 2^{2^k}, U]$.
4. If we find x in the finger search tree:
   (a) Insert x into tree T_0 and at the front of list L_0, unless x is already in T_0.
   (b) If T_0 is too full (storing 2^{2^1} elements), *overflow* T_0 as described in Algorithm 2.
   (c) Return a pointer to x in the finger search tree.

[a] If we reach a k for which T_k does not exist, then $k = \Theta(\lg\lg n)$ and we can afford to search in the global finger tree.

Algorithm 2. Overflowing a tree T_k.

1. Remove the 2^{2^k} most recently accessed elements from list L_k and tree T_k.
2. Build a balanced search tree T'_k and a list L'_k on these 2^{2^k} elements.
3. For each remaining element z in T_k, in rank order, if the predecessor of z in T_k (among the elements not yet deleted) is within rank distance $2^{2^{k+1}}$ of z, then delete z from T_k and L_k.
4. For each remaining element z in L_k, in access-time order:
   (a) Search for z in T_{k+1}.
   (b) If found, remove z from L_{k+1}.
   (c) Otherwise, insert z into T_{k+1}.
5. Concatenate L_k and L_{k+1} to form a new list L_{k+1}.
6. Replace $L_k \leftarrow L'_k$; $T_k \leftarrow T'_k$.
7. If T_{k+1} is now too full (stores at least $2^{2^{k+2}}$ elements), recursively overflow T_{k+1}.

element in T_{k+1} within a rank range of $2^{2^{k+1}}$. This intuition is formalized by the following lemma:

Lemma 1. *All elements within rank distance 2^{2^k} of an element y with working-set number $t_i(y) \le 2^{2^k}$ are within rank distance $3 \cdot 2^{2^k}$ of some element y' in $T_0 \cup T_1 \cup \cdots \cup T_k$.*

Proof. We track the evolution of y or a nearby element from when it was last accessed and inserted into T_0, to when it moved to T_1, T_2, and so on, until access i. If the tracked element y' is ever discarded from some tree T_j, we continue by tracking the promoted element within rank distance $2^{2^{j+1}}$ of y'. The tracked element y' makes monotone progress through $T_0, T_1, T_2, \ldots$ because, even if y' is accessed and inserted into T_0, the tracked node storing y' is not deleted. The tracked node also cannot expire from T_k (and get promoted or discarded), because at most 2^{2^k} distinct elements have been accessed in the time window under consideration, so y' must be among the first 2^{2^k} elements in the list L_k when it reaches L_k. Therefore, y' remains within rank distance $2^{2^k} + 2^{2^{k-1}} + \cdots + 2^{2^1} + 2^{2^0}$ of y, so we obtain the stronger bound that all elements within rank distance 2^{2^k} of y are within rank distance $2 \cdot 2^{2^k} + 2^{2^{k-1}} + \cdots + 2^{2^1} + 2^{2^0}$ of an element y' in $T_0 \cup T_1 \cup \cdots \cup T_k$.

3.4 Overflow Analysis

To analyze the amortized cost of the overflow algorithm, we consider the cost of overflowing T_k into T_{k+1} for each k separately. To be sure that we do not charge to the same target for multiple k, we introduce the notion of a *coin* c_k which can be used to pay for one node to overflow from T_k to T_{k+1} as well as for the node to later be discarded from T_{k+1}. A coin c_k cannot pay for overflows or discards at different levels. We assign coin c_k an intrinsic value of $\Theta(2^k)$ time units, but twice what is required to pay for a node to overflow and be discarded, so that whenever paying with a coin c_k we are also left with a fractional coin $\frac{1}{2}c_k$.

For each k, we consider the time interval after the previous overflow from T_k into T_{k+1} up to the next overflow from T_k into T_{k+1}. At the beginning of this time interval, we have just completed an overflow involving $\Theta(2^{2^{k+1}})$ elements each with a c_k coin. From the use of these coins we obtain fractional coins of half the value, which we can combine to give two whole c_k coins to every node of $T_0, T_1, \ldots, T_k$, because there are only $\sum_{j=0}^{k} 2^{2^j} = O(2^{2^k}) = o(2^{2^{k+1}})$ such nodes.

Consider a search for an element x during the time interval between the previous overflow and the next overflow of T_k. Suppose x was found at round ℓ of the search algorithm. The cost of searching for x is $O(2^\ell)$ time. We can therefore afford to give x two c_m coins for each $m \leq \ell$. We also award x with fractional coins $(6/2^{2^m})c_m$ for each $m > \ell$, which have total worth $o(1)$. We know that x was within rank distance $3 \cdot 2^{2^\ell}$ of an element y in $T_0 \cup T_1 \cup \cdots \cup T_\ell$. If $\ell < k$, we assign y as the *parent* of x. (This assignment may later change if we search for x again.)

Now consider each element x that gets promoted from T_k to T_{k+1} when T_k next overflows. If x has not been searched for since the previous overflow of T_k, then it was in $T_0 \cup T_1 \cup \cdots \cup T_k$ right after the previous overview, so x has two coins c_k. If the last search for x terminated in round ℓ with $\ell \geq k$, then x also has two coins c_k. In either of these cases, x uses one of its own c_k coins to pay for the cost of its overflow (and wastes the other c_k coin).

If x is within rank distance $2^{2^{k+1}}$ of a such an element y in $T_0 \cup T_1 \cup \cdots \cup T_k$ with two c_k coins, then y must not be promoted during this overflow. For if y expires from T_k during this overflow, then at most one of x and y can be promoted (whichever is larger), and we assumed it is x. Thus, x can steal one of y's c_k coins and use it for promotion. Furthermore, y can have a c_k coin stolen at most twice, once by an element $z < y$ and once by an element $z > y$, so we cannot over-steal. If y remains in $T_0 \cup T_1 \cup \cdots \cup T_k$, its c_k coins will be replenished after this overflow, so we also need not worry about y.

If x has no nearby element y with two c_k coins, we consider the chain connecting x to x's parent to x's grandparent, etc. Because every element without a c_k coin has a parent, and because we already considered the case in which an element with a c_k coin is within rank distance $2^{2^{k+1}}$ of x, the chain must extend so far as to reach an element with rank distance more than $2^{2^{k+1}}$ from x. Because every edge in the chain connects elements within rank distance $3 \cdot 2^{2^k}$, the chain must consist of at least $2^{2^{k+1}}/(3 \cdot 2^{2^k}) = 2^{2^k}/3$ elements within rank distance

$2^{2^{k+1}}$ of x. Because each of these elements has a parent, they must have been searched for since the last overflow of T_k, and were therefore assigned fractional coins of $(6/2^{2^k})c_k$. As before, none of these elements could be promoted from T_k during this overflow because they are too close to the promoted element x. Thus, x can steal a fractional coin of $(3/2^{2^k})c_k$ from each of these $2^{2^k}/3$ elements' fractional $(6/2^{2^k})c_k$ coins. Again, this stealing can happen at most twice for each fractional $(6/2^{2^k})c_k$ coin, so we do not over-steal.

Therefore, a promoted element x from the overflow of T_k can find a full coin c_k to pay for its promotion. The $O(2^k)$ cost of discarding an element x from T_k can be charged to the coin c_{k-1} that brought x there, or if $k = 0$, to the search that brought x there. This concludes the amortized analysis of overflow.

3.5 Insert

To insert an element x, we first call a slight variation of the search algorithm from Section 3.2 to find where x fits in the finger search tree. Specifically, we modify Step 4 to realize when the search has gone beyond x, at which point we can find the predecessor and successor of x in the finger search tree. Then, as part of Step 4, we insert x at that position in the finger search tree in $O(1)$ amortized time. We execute Steps 4(a–c) as before, inserting x into tree T_0 and list L_0.

Because this algorithm is almost identical to the search algorithm, we can use essentially the same analysis as in Section 3.4. More precisely, when we insert an element x, suppose we find where x fits during round ℓ of the algorithm Then we assign x a parent as before, and award x two c_m coins for each $m \leq \ell$ and fractional coins $(6/2^{2^m})c_m$ for each $m > \ell$. The only new concern in the amortized analysis is that the rank order changes by an insertion. Specifically, the rank distance between an element z and its parent y can increase by 1 because of an element x inserted between z and y. In this case, we set z's parent to x immediately after the insertion, and the proof goes through. Thus, the amortized cost of an insertion is proportional to the amortized cost of the initial search.

3.6 Delete

Finally we describe how to delete an element within the unified bound plus $O(\lg\lg|S_i|)$ time. Once we have found the element x to be deleted within the unified bound via the search algorithm, we remove x from the finger tree in $O(1)$ time and replace all instances of x in the T_k's with the successor or predecessor of x in the finger tree. To support each replacement in $O(1)$ time, and obtain a total bound of $O(\lg\lg|S_i|)$,[2] we maintain a list of back pointers from each element in the finger tree to the instances of that element as tree nodes in the

[2] We maintain the invariant that the number of trees T_k is at most $1+\lg\lg|S_i|$ simply by removing a tree T_k if k becomes larger than $\lg\lg|S_i|$. Such trees are not necessary for achieving the unified bound during searches.

T_k's. If more than one node in the same tree T_k ever points to the same element, we remove all but one of them.

The amortized analysis is again similar to Section 3.4, requiring only the following changes. Whenever we delete an element x and replace all its instances by its rank predecessor or successor y, element y inherits all of x's coins and takes over all of x's responsibilities in the analysis. We can even imagine x and y as both existing with equal rank, and handling their own responsibilities, with the additional advantage that if either one gets promoted the other one will be discarded (having the same rank) and hence need not be accounted for. An edge of a chain can only get shorter by this contraction in rank space, so the endpoints remain within rank distance $3 \cdot 2^{2^k}$ as required in the analysis. The unified bound to access an element z may also go down because it is closer in rank space to some elements, but this property is captured by the removal of x in the finger tree, and hence finger searches are correspondingly faster. Each tree T_k might get smaller (if both x and y were in the same tree), requiring us to break the invariant that T_k stores at least 2^{2^k} elements. However, we use this invariant only in proving Lemma 1, which remains true because the working-set numbers $t_i(z)$ count accesses to deleted elements and hence do not change.

Acknowledgments

We thank John Iacono for helpful discussions.

References

[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.

[Iac01] 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.

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