Biased Skip Lists

Amitabha Bagchi[1*], Adam L. Buchsbaum[2], and Michael T. Goodrich[3*]

[1] Dept. of Computer Science, Johns Hopkins Univ., Baltimore, MD 21218,
bagchi@cs.jhu.edu
[2] AT&T Labs, Shannon Laboratory, 180 Park Ave., Florham Park, NJ 07932,
alb@research.att.com
[3] Dept. of Info. & Computer Science, Univ. of California, Irvine, CA 92697-3425,
goodrich@ics.uci.edu

Abstract. We design a variation of skip lists that performs well for generally biased access sequences. Given n items, each with a positive weight w_i, $1 \leq i \leq n$, the time to access item i is $O\left(1 + \log \frac{W}{w_i}\right)$, where $W = \sum_{i=1}^{n} w_i$; the data structure is dynamic. We present deterministic and randomized variations, which are nearly identical; the deterministic one simply ensures the balance condition that the randomized one achieves probabilistically. We use the same method to analyze both.

1 Introduction

The primary goal of data structures research is to design data organization mechanisms that admit fast access and update operations. For a generic n-element ordered data set that is accessed and updated uniformly, this goal is typically satisfied by dictionaries that achieve $O(\log n)$-time search and update performance; e.g., AVL-trees [2], red-black trees [12], and (a, b)-trees [13].

Nevertheless, many dictionary applications involve sets of weighted data items subject to non-uniform access patterns that are *biased* according to the weights. For example, operating systems (e.g., see Stallings [22]) deal with biasing in memory requests. Other recent examples of biased sets include client web server requests [11] and DNS lookups [6]. For such applications, a *biased search structure* is more appropriate—that is, a structure that achieves search times faster than $\log n$ for highly weighted items. Biased searching is also useful in auxiliary structures deployed inside other data structures [5,10,20].

Formally, a *biased dictionary* is a data structure that maintains an ordered set X, each element i of which has a *weight*, w_i; without loss of generality, we assume $w_i \geq 1$. The operations are as follows.

Search(X, i). Determine if i is in X.
Insert(X, i). Add i to X.
Delete(X, i). Delete i from X.
Join(X_L, X_R). Assuming that $i < j$ for each $i \in X_L$ and $j \in X_R$, create a new set $X = X_L \cup X_R$. The operation destroys X_L and X_R.

* Supported by DARPA Grant F30602-00-2-0509 and NSF Grant CCR-0098068.

P. Bose and P. Morin (Eds.): ISAAC 2002, LNCS 2518, pp. 1–13, 2002.

Split(X, i). Assuming without loss of generality that $i \notin X$, create $X_L = \{j \in X : j < i\}$ and $X_R = \{j \in X : j > i\}$. The operation destroys X.

FingerSearch(X, i, j). Determine if j is in X, exploiting a handle in the data structure to element $i \in X$.

Reweight(X, i, w'_i). Change the weight of i to w'_i.

In this paper, we study efficient data structures for biased data sets subject to these operations. We desire search times that are asymptotically optimal and update times that are also efficient. For example, consider the case when w_i is the number of times item i is accessed. Define $W = \sum_{i=1}^{n} w_i$. A biased dictionary with $O\left(\log \frac{W}{w_i}\right)$ search time for the i'th item can perform m searches on n items in $O(m(1 - \sum_{i=1}^{n} p_i \log p_i))$ time, where $p_i = \frac{w_i}{m}$, which is asymptotically optimal [1]. We therefore desire $O\left(\log \frac{W}{w_i}\right)$ search times and similar update times for general biased data (with arbitrary weights). We also seek biased structures that would be simple to implement and that do not require major restructuring operations, such as tree rotations, to achieve biasing. Tree rotations, in particular, make structures less amenable to augmentation, for such rotations often require the complete rebuilding of auxiliary structures stored at the affected nodes.

1.1 Related Prior Work

The study of biased data structures is a classic topic in algorithmics. Early work includes a dynamic programming method by Knuth [14,15] for constructing a static biased binary search tree for items weighted by their search frequencies. Subsequent work focuses primarily on achieving asymptotically optimal search times while also admitting efficient updates. Most of the known methods for constructing dynamic biased data structures use search trees, and they differ from one another primarily in their degree of complication and whether or not their resulting time bounds are amortized, randomized, or worst case.

Sleator and Tarjan [21] introduce the theoretically elegant *splay trees*, which automatically adjust themselves to achieve optimal amortized biased access times for access-frequency weights. Splay trees store no balance or weight information, but they perform many tree rotations after every access, which makes them less practically efficient than even AVL-trees in many applications [3]. These rotations can be particularly deleterious when nodes are augmented with auxiliary structures.

Bent, Sleator, and Tarjan [4] and Feigenbaum and Tarjan [9] design biased search trees for arbitrary weights that significantly reduce, but do not eliminate, the number of tree rotations needed. They offer efficient worst-case and amortized performance of biased dictionary operations but do so with complicated implementations.

Seidel and Aragon [19] demonstrate randomized bounds with *treaps*. Like splay trees, treaps perform a large number of rotations after every access. Their data structure is elegant and efficient in practice, but its performance does not achieve bounds that are efficient in a worst-case or amortized sense.

Pugh [18] introduces an alternative *skip list* structure, which efficiently implements an unbiased dictionary without using rotations. Skip lists store the items in series of a linked lists, which are themselves linked together in a leveled fashion. Pugh presents skip lists as a randomized structure that is easily implemented and shows that they are empirically faster than fast balanced search trees, such as AVL-trees. Search and updates take $O(\log n)$ expected time in skip lists, with no rotations or other rebalancing needed for updates. Exploiting the relationship between skip lists and (a, b)-trees, Munro, Papadakis, and Sedgewick [17] show how to implement a deterministic version of skip lists that achieves similar bounds in the worst case using simple promote and demote operations.

For biased skip lists, much less prior work exists. Mehlhorn and Näher [16] anticipated biased skip lists but claimed only a partial result and omitted details and analysis. Recently, Ergun *et al.* [7,8] presented a biased skip list structure that is designed for a specialized notion of biasing, in which access to an item i takes $O(\log r(i))$ expected time, where $r(i)$ is the number of items accessed since the last time i was accessed. Their data structure is incomparable to a general biased dictionary, as each provides properties not present in the other.

1.2 Our Results

We present a comprehensive design of a biased version of skip lists. It combines techniques underlying deterministic skip lists [17] with Mehlhorn and Näher's suggestion [16]. Our methods work for arbitrarily defined item weights and provide asymptotically optimal search times based on these weights. Using skip list technology eliminates tree rotations. We present complete descriptions of all the biased dictionary operations, with time performances that compare favorably with those of the various versions of biased search trees. We give both deterministic and randomized implementations. Our deterministic structure achieves worst-case running times similar to those of biased search trees [4,9] but uses techniques that are arguably simpler. A node in a deterministic biased skip list is assigned an initial level based on its weight, and simple invariants govern promotion and demotion of node levels to ensure desired access times. Our randomized structure achieves expected bounds similar to the respective amortized and randomized bounds of splay trees [21] and treaps [19]. Our randomized structure does not use partial rebuilding and hence does not need any amortization of its own. Table 1 (at the end) juxtaposes our results against biased search trees, splay trees, and treaps.

In Section 2, we define our deterministic biased skip list structure, and in Section 3 we describe how to perform updates efficiently in this structure. In Section 4 we describe a simple, randomized variation of biased skip lists and analyze its performance. We conclude in Section 5.

2 Biased Skip Lists

A *skip list* [18] S is a dictionary data structure, storing an ordered set X, the items of which we number 1 through $|X|$. Each item $i \in X$ has a key, x_i, and a corresponding *node* in the skip list of some integral *height*, $h_i \geq 0$. The *height* of S is $H(S) = \max_{i \in X} h_i$. The *depth*, d_i, of i is $H(S) - h_i$. We use the terms item, node, and key interchangeably; the context clarifies any ambiguity. We assume without loss of generality that the keys in X are unique: $x_i < x_{i+1}$, $1 \leq i < |X|$.

Each node i is implemented by a linked list or array of length $h_i + 1$, which we call the *tower* for that node. The *level-j successor* of a node i is the least node $\ell > i$ of height $h_\ell \geq j$. Symmetrically define *level-j predecessor*. For node i and each $0 \leq j \leq h_i$, the j'th element of the node contains pointers to the j'th elements of the level-j successor and predecessor of i. Two distinct nodes $x < y$ are called *consecutive* if and only if $h_z < \min(h_x, h_y)$ for all $x < z < y$. A *plateau* is a maximal sequence of consecutive nodes of equal height.

For convenience we assume sentinel nodes of height $H(S)$ at the beginning (with key $-\infty$) and end (with key ∞) of S; in practice, this assumption is not necessary. We orient the pointers so that the skip list stores items in left-to-right order, and the node levels progress bottom to top. See Figure 1(a).

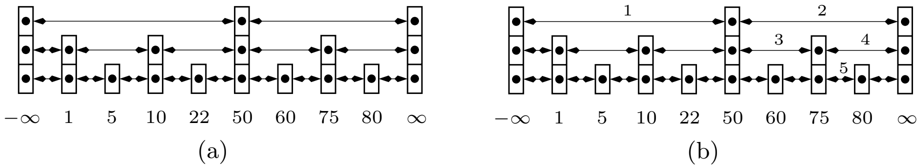

Fig. 1. (a) A skip list for the set $X = \{1, 5, 10, 22, 50, 60, 75, 80\}$. (b) Searching for key 80; numbers over the pointers indicate the order in which they are traversed.

To search for an item with key K we start at level $H(S)$ of the left sentinel. When searching at level i from some node we follow the level-i links to the right until we find a key matching K or a pair of nodes j, k such that k is the level-i successor of j and $x_j < K < x_k$. We then continue the search at level $i-1$ from node j. The search ends with success if we find a node with key K, or failure if we find nodes j and k as above on level 0. See Figure 1(b).

We describe a deterministic, biased version of skip lists. In addition to key x_i each item $i \in X$ has a *weight*, w_i; without loss of generality, assume $w_i \geq 1$. Define the *rank* of item i as $r_i = \lfloor \log_a w_i \rfloor$, where a is a constant parameter.

Definition 1. *For a and b such that $1 < a \leq \left\lfloor \frac{b}{2} \right\rfloor$, an (a,b)*-biased skip list *is one in which each item has height $h_i \geq r_i$ and the following invariants hold.*

(I1) *There are never more than b consecutive items of any height in $[0, H(S)]$.*
(I2) *For each node x and all $r_x < i \leq h_x$, there are at least a nodes of height $i-1$ between x and any consecutive node of height at least i.*

To derive exact bounds for the case when an item does not exist in the skip list we eliminate redundant pointers. For every pair of adjacent items $i, i+1$, we ensure that the pointers between them on level $\min(h_i, h_{i+1}) - 1$ are nil; the pointers below this level are undefined. (In Figure 1, for example, the level-0 pointers between $-\infty$ and 1 become nil.) When searching for an item $i \notin X$, we assert failure immediately upon reaching a nil pointer.

Throughout the remainder of the paper, we define $W = \sum_{i \in X} w_i$ to be the weight of S before any operation. For any key i, we denote by i^- the item in X with largest key less than i, and by i^+ the item in X with smallest key greater than i. The main result of our definition of biased skip lists is summarized by the following lemma, which bounds the depth of any node.

Lemma 1 (Depth Lemma). *The depth of any node i in an (a,b)-biased skip list is $O\left(\log_a \frac{W}{w_i}\right)$.*

Before we prove the depth lemma, consider its implication on *access time* for key i: the time it takes to find i in S if $i \in X$ or pair i^-, i^+ in S if $i \notin X$.

Corollary 1 (Access Lemma). *The access time for key i in an (a,b)-biased skip list is $O\left(1 + b\log_a \frac{W}{w_i}\right)$ if $i \in X$ and $O\left(1 + b\log_a \frac{W}{\min\left(w_{i^-}, w_{i^+}\right)}\right)$ if $i \notin X$.*

Proof. By **(I1)**, at most $b+1$ pointers are traversed at any level. A search stops upon reaching the first nil pointer, so the Depth Lemma implies the result.

It is important to note that while all the bounds we prove rely on W, the data structure itself need not maintain this value.

To prove the depth lemma, observe that the number of items of any given rank that can appear at higher levels decreases geometrically by level. Define $N_i = |\{x : r_x = i\}|$ and $N'_i = |\{x : r_x \leq i \wedge h_x \geq i\}|$.

Lemma 2. $N'_i \leq \sum_{j=0}^{i} \frac{1}{a^{i-j}} N_j$.

Proof. By induction. The base case, $N'_0 = N_0$, is true by definition. For $i > 0$, **(I2)** implies that $N'_{i+1} \leq N_{i+1} + \left\lfloor \frac{1}{a} N'_i \right\rfloor \leq N_{i+1} + \frac{1}{a} N'_i$, which, together with the induction hypothesis, proves the lemma.

Intuitively, a node promoted to a higher level is supported by enough weight associated with items at lower levels. Define $W_i = \sum_{r_x \leq i} w_x$.

Corollary 2. $W_i \geq a^i N'_i$.

Proof. By definition, $W_i \geq \sum_{j=0}^{i} a^j N_j = a^i \sum_{j=0}^{i} \frac{1}{a^{i-j}} N_j$. Apply Lemma 2.

Define $R = \max_{x \in X} r_x$. Any nodes with height exceeding R must have been promoted from lower levels to maintain the invariants. **(I2)** thus implies that $H(S) \leq R + \log_a N'_R$, and therefore the maximum possible depth of an item i is $d_i \leq H(S) - r_i \leq R + \log_a N'_R - r_i$.

By Corollary 2, $W = W_R \geq a^R N'_R$. Therefore $\log_a N'_R \leq \log_a W - R$. Hence, $d_i \leq \log_a W - r_i$. The Depth Lemma follows, because $\log_a w_i - 1 < r_i \leq \log_a w_i$.

(I1) and **(I2)** resemble the invariants defining (a,b)-skip lists [17], but **(I2)** is stronger than their analogue. Just to prove the Depth Lemma, it would suffice for a node of height h exceeding its rank, r, to be supported by at least a items to each side only at level $h-1$, not at every level between r and $h-1$. The update procedures in the next section, however, require support at every level.

3 Updating Deterministic Biased Skiplists

We describe insertion in detail and then sketch implementations for the other operations. All details will be available in the full paper.

The *profile* of an item i captures its predecessors and successors of increasingly greater level. For $h_{i^-} \leq j \leq H(S)$, let L_j^i be the level-j predecessor of i; for $h_{i^+} \leq j \leq H(S)$, let R_j^i be the level-j successor of i. Define the ordered set $PL(i) = \left(j : h_{L_j^i} = j,\ h_{i^-} \leq j \leq H(S)\right)$: the set of distinct heights of the nodes to the left of i. Symmetrically define $PR(i) = \left(j : h_{R_j^i} = j,\ h_{i^+} \leq j \leq H(S)\right)$. We call the ordered set $\left(L_j^i : j \in PL(i)\right) \cup \left(R_j^i : j \in PR(i)\right)$ the *profile* of i. We call the subset of predecessors the *left profile* and the subset of successors the *right profile* of i. For example, in Figure 1, $PL(60) = (3)$; $PR(60) = (2,3)$; the left profile of 60 is (50); and the right profile of 60 is $(75,\infty)$.

These definitions assume $i \in S$ but are also precise when $i \notin S$, in which case they apply to the (nonexistent) node that would contain key i. Given node i or, if $i \notin S$, nodes i^- and i^+, we can trace i's profile from lowest-to-highest nodes by starting at i^- (rsp., i^+) and, at any node x, iteratively finding its level-(h_x+1) predecessor (rsp., successor), until we reach the left (rsp., right) sentinel.

3.1 Inserting an Item

The following procedure inserts a new item with key i into an (a,b)-biased skip list S. If i already exists in the skip list, we discover it in Step 1.

1. Search S for i to discover the pair i^-, i^+.
2. Create a new node of height r_i to store i, and insert it between i^- and i^+ in S, splicing predecessors and successors as in a standard skip list [18].
3. Restore **(I2)**, if necessary. Any node x in the left (sym., right) profile of i might need to have its height demoted, because i might interrupt a plateau of height less than h_x, leaving fewer than a nodes to x's left (sym., right). In this case, x is demoted to the next lower height in the profile (or r_x, whichever is higher). More precisely, for j in turn from h_{i^-} up through r_i, if $j \in PL(i)$, consider node $u = L_j^i$. If **(I2)** is violated at node u, then demote u to height r_u if $u = i^-$ and otherwise to height $\max(j', r_u)$, where j' is the predecessor of j in $PL(i)$; let h'_u be the new height of u. If the demotion violates **(I1)** at level h'_u, then, among the $k \in (b, 2b]$ consecutive items of height h'_u, promote the $\lfloor \frac{k}{2} \rfloor$'th node (in order) to height $h'_u + 1$. (See Figure 2.) Iterate at the next j. Symmetrically process right profile of i.

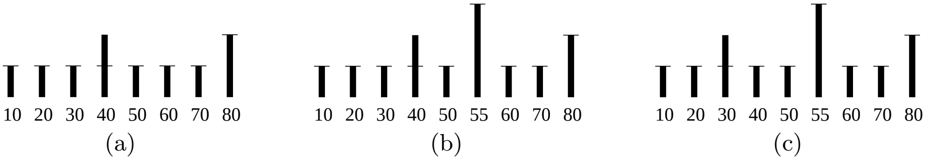

Fig. 2. (a) A (2,4)-biased skip list. Nodes are drawn to reflect their heights; hatch marks indicate ranks. Pointers are omitted. (b) After inserting 55 with rank 3, node 40 violates **(I2)**. (c) After demotion of 40 and compensating promotion of 30.

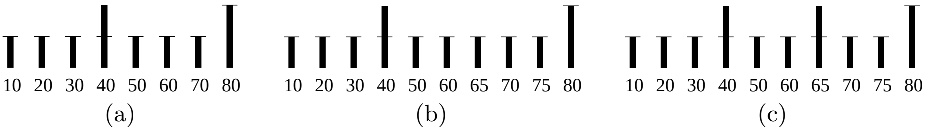

Fig. 3. (a) The (2,4)-biased skip list of Figure 2(a). (b) **(I1)** is violated by the insertion of 65 and 75 with rank 1 each. (c) After promoting node 65.

4. Restore **(I1)**, if necessary. Starting at node i and level $j = r_i$, if node i violates **(I1)** at level j, then, among the $b+1$ consecutive items of height j, promote the $\lfloor \frac{b+1}{2} \rfloor$'th node (in order), u, to height $j+1$, and iterate at node u and level $j+1$. Continue until the violations stop. (See Figure 3.)

Theorem 1. *Inserting an item i in an (a,b)-biased skip list can be done in $O\left(1 + b\log_a \frac{W+w_i}{\min(w_{i^-},w_i,w_{i^+})}\right)$ time.*

Proof. We omit the proof of correctness. By the Depth and Access Lemmas, Steps 1 and 2 take $O\left(1 + b\log_a \frac{W+w_i}{\min(w_{i^-},w_i,w_{i^+})}\right)$ time. If $\min(h_{i^-}, h_{i^+}) \leq r_i$, Step 3 performs $O(b)$ work at each level between $min(h_{i^-}, h_{i^+})$ and r_i; Step 4 performs $O(b)$ work at each level from r_i through $H(S)$. Again apply the Depth Lemma.

3.2 Deleting an Item

Deletion is the inverse of insertion. After finding i, i^-, and i^+, remove i and splice predecessors and successors as required. Then restore **(I1)**, if necessary, as removing i might unite plateaus into sequences of length exceeding b. This is done analogously to Step 4 of insertion, starting at level $\min(h_{i^-}, h_{i^+})$ and proceeding up through level $h_i - 1$. Finally, restore **(I2)**, if necessary, as removing i might decrease the length of a plateau of height h_i to $a-1$. This is done analogously to Step 3 of insertion, starting at level h_i. The proof of correctness is analogous to that for insertion, and the time is $O\left(1 + b\log_a \frac{W}{\min(w_{i^-},w_i,w_{i^+})}\right)$.

3.3 Joining Two Skiplists

Consider biased skip lists S_L and S_R of total weights W_L and W_R, rsp. Denote the item with the largest key in S_L by $L_{\max}$ and that with the smallest key in S_R by $R_{\min}$. Assume $L_{\max} < R_{\min}$. To join S_L and S_R, trace through the profiles of $L_{\max}$ and $R_{\min}$ to splice S_L and S_R together. Restore **(I1)**, if necessary, starting at level $\max(h_{L_{\max}}, h_{R_{\min}})$ and proceeding through level $\max(H(S_L), H(S_R))$, as in Step 4 of insertion. **(I2)** cannot be violated by the initial splicing, as plateaus never shrink, nor by the promotion of the node in the middle in the restoration of **(I1)**. The time is $O\left(1 + b \log_a \frac{W_L}{w_{L_{\max}}} + b \log_a \frac{W_R}{w_{R_{\min}}}\right)$.

3.4 Splitting a Skiplist

We can split a biased skip list S of total weight W into two biased skip lists, S_L and S_R, containing keys in S less than (rsp., greater than) some $i \notin S$. First insert i into S with weight $w_i = a^{H(S)+1}$. Then disconnect the pointers between i and its predecessors (rsp., successors) to form S_L (rsp., S_R). **(I1)** and **(I2)** are true after inserting i by the correctness of insertion. Because i is taller than all of its predecessors and successors, disconnecting the pointers between them and i does not violate either invariant. The time is $O\left(1 + b \log_a \frac{W}{\min(w_{i^-}, w_{i^+})}\right)$.

3.5 Finger Searching

We can search for a key j in a biased skip list S starting at any node i to which we are given an initial pointer (or *finger*). Assume without loss of generality that $j > i$. The case $j < i$ is symmetric.

At any point in the search, we are at some height h of some node u. Initially, $u = i$ and $h = r_i$. In the *up phase*, while $R_h^u < j$, we continually set $h \leftarrow h + 1$ when $h < h_u$ and $u \leftarrow R_h^u$ when $h = h_u$. Once $R_h^u \geq j$, we enter the *down phase*, in which we search from u at height h using the normal search procedure.

The up phase moves up and to the right until we detect a node $u < j$ with some level-h successor $R_h^u > j$. That the procedure finds j (or j^-, j^+ if $j \notin S$) follows from the correctness of the vanilla search procedure and that we enter the down phase at the specified node u and height h.

Define $V(i, j) = \sum_{i \leq u \leq j} w_u$. For any node u and $h \in [r_u, h_u]$, it follows by induction that $V(L_h^u, u) \geq a^h$ and $V(u, R_h^u) \geq a^h$. Using this fact we can show that sufficient weight supports either the link into which u is originally entered during the up phase or the link out of which u is exited during the down phase. It follows that the time is $O\left(1 + b \log_a \frac{V(i,j)}{\min(w_i, w_j)}\right)$ if $j \in X$ and $O\left(1 + b \log_a \frac{V(i,j^+)}{\min(w_i, w_{j^-}, w_{j^+})}\right)$ if $j \notin X$.

3.6 Changing the Weight of an Item

We can change the weight of an item i to w_i' without deleting and reinserting i. Let $r_i' = \lfloor \log_a w_i' \rfloor$. If $r_i' = r_i$, then stop. If $r_i' > r_i$, then stop if $h_i \geq r_i'$.

Otherwise, promote i to height r'_i; restore **(I2)** as in insertion, starting at height $h_i + 1$; and restore **(I1)** as in insertion, starting at height r'_i. Finally, if $r'_i < r_i$, then demote i to height r'_i; restore **(I1)** as in deletion, starting at height r'_i; and restore **(I2)** as in deletion, starting at the least $j \in PL(i)$ greater than r'_i.

Correctness follows analogously as for insertion (in case $r'_i > r_i$) or deletion (in case $r'_i < r_i$). The time is $O\left(1 + b \log_a \frac{W + w'_i}{\min(w_i, w'_i)}\right)$.

4 Randomized Updates

We can randomize the structure to yield expected optimal access times without any promotions or demotions. Mehlhorn and Näher [16] suggested the following approach but claimed only that the expected maximal height of a node is $\log W + O(1)$. We will show that the expected depth of a node i is $E[d_i] = O\left(\log \frac{W}{w_i}\right)$.

A *randomized biased skip list* S is parameterized by a positive constant $0 < p < 1$. Here we define the *rank* of an item i as $r_i = \lfloor \log_{\frac{1}{p}} w_i \rfloor$. When inserting i into S, we assign its height to be $h_i = r_i + e_i$ with probability $p^{e_i}(1-p)$ for $e_i \in \mathbb{Z}$, which we call the *excess height* of i. Algorithmically, we start node i at height r_i and continually increment the height by one as long as a biased coin flip returns heads (with probability p).

Reweight is the only operation that changes the height of a node. The new height is chosen as for insertion but based on the new weight, and the tower is adjusted appropriately. The remaining operations (insert, delete, join, split, and (finger) search) perform no rebalancing.

Lemma 3 (Randomized Height Lemma). *The expected height of any item i in a randomized, biased skip list is $\log_{\frac{1}{p}} w_i + O(1)$.*

Proof. $E[h_i] = r_i + E[e_i] = r_i + \sum_{j=0}^{\infty} j p^j (1-p) = r_i + \frac{p}{1-p} = \lfloor \log_{\frac{1}{p}} w_i \rfloor + O(1)$.

The proof of the Depth Lemma for the randomized structure follows that for the deterministic structure. Recall the definitions $N_i = |\{x : r_x = i\}|$; $N'_i = |\{x : r_x \le i \wedge h_x \ge i\}|$; and $W_i = \sum_{r_x \le i} w_x$.

Lemma 4. $E[N'_i] = \sum_{j=0}^{i} p^{i-j} N_j$.

Proof. By induction. By definition, $N'_0 = N_0$. Since the excess heights are i.i.d. random variables, for $i > 0$, $E[N'_{i+1}] = N_{i+1} + pE[N'_i]$, which, together with the induction hypothesis, proves the lemma.

Corollary 3. $E[N'_i] \le p^i W_i$.

Lemma 5 (Randomized Depth Lemma). *The expected depth of any node i in a randomized, biased skip list S is $O\left(\log_{\frac{1}{p}} \frac{W}{w_i}\right)$.*

Proof. The depth of i is $d_i = H(S) - h_i$. Again define $R = \max_{x \in X} r_x$. By standard skip list analysis [18],

$$\begin{aligned} E[H(S)] &= R + O(E[\log_{\frac{1}{p}} N'_R]) \leq R + cE[\log_{\frac{1}{p}} N'_R] \text{ for some constant } c \\ &\leq R + c \log_{\frac{1}{p}} E[N'_R] \text{ by Jensen's inequality} \\ &\leq R + c\left(\log_{\frac{1}{p}} W_R - R\right) \text{ by Corollary 3} \\ &= c \log_{\frac{1}{p}} W - (c-1)R. \end{aligned}$$

By the Randomized Height Lemma, therefore, $E[d_i] \leq c \log_{\frac{1}{p}} W - (c-1)R - \log_{\frac{1}{p}} w_i$. The lemma follows by observing that $R \geq \lfloor \log_{\frac{1}{p}} w_i \rfloor$.

Corollary 4 (Randomized Access Lemma). *The expected access time for any key i in a randomized, biased skip list is $O\left(1 + \frac{1}{p} \log_{\frac{1}{p}} \frac{W}{w_i}\right)$ if $i \in X$ and $O\left(1 + \frac{1}{p} \log_{\frac{1}{p}} \frac{W}{\min(w_{i^-}, w_{i^+})}\right)$ if $i \notin X$.*

Proof. As $n \to \infty$, the probability that a plateau starting at any given node is of size k is $p(1-p)^{k-1}$. The expected size of any plateau is thus $1/p$.

The operations discussed in Section 3 become simple to implement.

Insert(S, i). Locate i^- and i^+ and create a new node between them to hold i. The expected time is $O\left(1 + \frac{1}{p} \log_{\frac{1}{p}} \frac{W + w_i}{\min(w_{i^-}, w_i, w_{i^+})}\right)$.

Delete(S, i). Locate and remove node i. The Randomized Depth and Access Lemmas continue to hold, because S is as if i had never been inserted. The expected time is $O\left(1 + \frac{1}{p} \log_{\frac{1}{p}} \frac{W}{\min(w_{i^-}, w_i, w_{i^+})}\right)$.

Join(S_L, S_R). Trace through the profiles of $L_{\max}$ and $R_{\min}$ to splice the pointers leaving S_L together with the pointers going into S_R. The expected time is $O\left(1 + \frac{1}{p} \log_{\frac{1}{p}} \frac{W_L}{w_{L_{\max}}} + \frac{1}{p} \log_{\frac{1}{p}} \frac{W_R}{w_{R_{\min}}}\right)$.

Split(S, i). Disconnect the pointers that join the left profile of i^- to the right profile of i^+. The expected time is $O\left(1 + \frac{1}{p} \log_{\frac{1}{p}} \frac{W}{\min(w_{i^-}, w_{i^+})}\right)$.

FingerSearch(S, i, j). Perform **FingerSearch(S, i, j)** as described in Section 3.5. The expected time if $j \in X$ is $O\left(1 + \frac{1}{p} \log_{\frac{1}{p}} \frac{V(i,j)}{\min(w_i, w_j)}\right)$ and if $j \notin X$ is $O\left(1 + \frac{1}{p} \log_{\frac{1}{p}} \frac{V(i,j^+)}{\min(w_i, w_{j^-}, w_{j^+})}\right)$.

Reweight(S, i, w'_i). Reconstruct the tower for node i. The expected time is $O\left(1 + \frac{1}{p} \log_{\frac{1}{p}} \frac{W + w'_i}{\min(w_i, w'_i)}\right)$.

Table 1. Time bounds for biased data structures. In all bounds, W is the total weight of all items before the operation; $V(i,j) = \sum_{k=i}^{j} w_k$. For each table entry, E, the associated time bound is $O(1+E)$.

	Biased Search Trees [4]	Splay Trees [21] amort.	Treaps [19] rand.	Biased Skip Lists w.c. & rand.
Search(X,i)	$\log \frac{W}{w_i}$ amort./w.c.	$\log \frac{W}{w_i}$	$\log \frac{W}{w_i}$	$\log \frac{W}{w_i}$
Insert(X,i)	$\log \frac{W+w_i}{\min(w_{i-}+w_{i+},w_i)}$ amort. $\log \frac{W}{w_{i-}+w_{i+}} + \log \frac{W+w_i}{w_i}$ w.c.	$\log \frac{W}{\min(w_{i-},w_{i+})} + \log \frac{W+w_i}{w_i}$	$\log \frac{W+w_i}{\min(w_{i-},w_i,w_{i+})}$	$\log \frac{W+w_i}{\min(w_{i-},w_i,w_{i+})}$
Delete(X,i)	$\log \frac{W}{w_i}$ amort. $\log \frac{W}{w_i} + \log \frac{W-w_i}{w_{i-}+w_{i+}}$ w.c.	$\log \frac{W}{w_i} + \log \frac{W-w_i}{w_{i-}}$	$\log \frac{W+w_i}{\min(w_{i-},w_i,w_{i+})}$	$\log \frac{W+w_i}{\min(w_{i-},w_i,w_{i+})}$
Join(X_L,X_R)	$\log \frac{W_L+W_R}{w_{L_{\max}}+w_{R_{\min}}}$ w.c.	$\log \frac{W_L+W_R}{w_{L_{\max}}}$	$\log \frac{W_L}{w_{L_{\max}}} + \log \frac{W_R}{w_{R_{\min}}}$	$\log \frac{W_L}{w_{L_{\max}}} + \log \frac{W_R}{w_{R_{\min}}}$
Split(X,i)	$\log \frac{W}{w_{i-}+w_{i+}}$ amort./w.c.	$\log \frac{W}{\min(w_{i-},w_{i+})}$	$\log \frac{W_L}{w_{L_{\max}}} + \log \frac{W_R}{w_{R_{\min}}}$	$\log \frac{W}{\min(w_{i-},w_{i+})}$
Reweight(X,i,w_i')	$\log \frac{\max(W,W')}{\min(w_i,w_i')}$ amort. $\log \frac{W}{w_i} + \log \frac{W'}{w_i'}$ w.c.		$\log \frac{\max(w_i,w_i')}{\min(w_i,w_i')}$	$\log \frac{W'}{\min(w_i,w_i')}$
FingerSearch(X,i,j)			$\log \frac{V(i,j)}{\min(w_i,w_j)}$	$\log \frac{V(i,j)}{\min(w_i,w_j)}$

5 Conclusion

Open is whether a deterministic biased skip list can be devised that has not only the worst-case times that we provide but also an amortized bound of $O(\log w_i)$ for *updating* node i; i.e., once the location of the update is discovered, inserting or deleting should take $O(\log w_i)$ amortized time.

The following counterexample demonstrates that our initial method of promotion and demotion does not yield this bound. Consider a node i such that $h_i - r_i$ is large and, moreover, that separates two plateaus of size $b/2$ at each level j between $r_i + 1$ and h_i and two plateaus of size $b/2$ and $b/2 + 1$, rsp., at level r_i. Deleting i will cause a promotion starting at level r_i that will percolate to level h_i. Reinserting i with weight a^{r_i} will restore the structural condition before the deletion of i. This pair of operations can be repeated infinitely often; since $h_i - r_i$ is arbitrary, the cost of restoring the invariants cannot be amortized.

We might generalize the promotion operation to split a plateau of size exceeding b into several plateaus of size about b/η each, for some suitable constant η. Above, $\eta = 2$. The counterexample generalizes, however.

References

1. N. Abramson. *Information Theory and Coding.* McGraw-Hill, New York, 1963.
2. G. M. Adel'son-Vel'skii and Y. M. Landis. An algorithm for the organisation of information. *Dokl. Akad. Nauk SSSR*, 146:263–6, 1962. English translation in *Soviet Math. Dokl.* **3**:1259–62, 1962.
3. J. Bell and G. Gupta. Evaluation of self-adjusting binary search tree techniques. *Soft. Prac. Exp.*, 23(4):369–382, 1993.
4. S. W. Bent, D. D. Sleator, and R. E. Tarjan. Biased search trees. *SIAM J. Comp.*, 14(3):545–68, 1985.
5. Y.-J. Chiang and R. Tamassia. Dynamization of the trapezoid method for planar point location in monotone subdivisions. *Int'l. J. Comp. Geom. Appl.*, 2(3):311–333, 1992.
6. E. Cohen and H. Kaplan. Proactive caching of DNS records: Addressing a performance bottleneck. In *Proc. SAINT '01*, pages 85–92. IEEE, 2001.
7. F. Ergun, S. C. Sahinalp, J. Sharp, and R. K. Sinha. Biased dictionaries with fast inserts/deletes. In *Proc. 33rd ACM STOC*, pages 483–91, 2001.
8. F. Ergun, S. C. Sahinalp, J. Sharp, and R. K. Sinha. Biased skip lists for highly skewed access patterns. In *Proc. 3rd ALENEX*, volume 2153 of *LNCS*, pages 216–29. Springer, 2001.
9. J. Feigenbaum and R. E. Tarjan. Two new kinds of biased search trees. *BSTJ*, 62(10):3139–58, 1983.
10. M. T. Goodrich and R. Tamassia. Dynamic ray shooting and shortest paths in planar subdivisions via balanced geodesic triangulations. *J. Alg.*, 23:51–73, 1997.
11. S. D. Gribble and E. A. Brewer. System design issues for internet middleware services: Deductions from a large client trace. In *Proc. 1st USENIX Symp. on Internet Tech. and Syst.*, 1997.
12. L. J. Guibas and R. Sedgewick. A dichromatic framework for balanced trees. In *Proc. 19th IEEE FOCS*, pages 8–21, 1978.

13. S. Huddleston and K. Mehlhorn. A new data structure for representing sorted lists. *Acta Inf.*, 17:157–84, 1982.
14. D. E. Knuth. Optimum binary search trees. *Acta Inf.*, 1:14–25, 1971.
15. D. E. Knuth. *The Art of Computer Programming*, volume 3: *Sorting and Searching*. Addison-Wesley, 1973.
16. K. Mehlhorn and S. Näher. Algorithm design and software libraries: Recent developments in the LEDA project. In *Proc. IFIP '92*, volume 1, pages 493–505. Elsevier, 1992.
17. J. I. Munro, T. Papadakis, and R. Sedgewick. Deterministic skip lists. In *Proc. 3rd ACM-SIAM SODA*, pages 367–75, 1992.
18. W. Pugh. Skip lists: A probabilistic alternative to balanced trees. *C. ACM*, 33(6):668–76, June 1990.
19. R. Seidel and C. R. Aragon. Randomized search trees. *Algorithmica*, 16(4/5):464–97, 1996.
20. D. D. Sleator and R. E. Tarjan. A data structure for dynamic trees. *JCSS*, 26(3):362–91, 1983.
21. D. D. Sleator and R. E. Tarjan. Self-adjusting binary search trees. *J. ACM*, 32(3):652–86, 1985.
22. W. Stallings. *Operating Systems: Internals and Design Principles.* Prentice-Hall, 4th edition, 2001.