Biased Dictionaries with Fast Insert/Deletes

Funda Ergun * S. Cenk Şahinalp † Jonathan Sharp ‡ Rakesh K. Sinha §

ABSTRACT

A dictionary data structure supports efficient *search*, *insert*, and *delete* operations on n keys from a totally ordered universe. Red-black trees, 2-3 trees, AVL trees, skip lists and other classic data structures facilitate $O(\log n)$ time search, insert and deletes, matching the information theoretic lower bound when access probabilities are uniform i.i.d. If access probabilities are non-uniform but still i.i.d., there are other *weighted* data structures such as D-trees, biased search trees, splay trees and treaps which can achieve optimality.

In many applications, however, the source of non-uniformity in access probabilities is *locality of reference*: examples include memory, cache, disk and buffer management and emerging applications in internetwork traffic management. In such applications, the access probability of any given key is not i.i.d., but decreases with idle time since the last access to the key.

It is possible to adjust the weighted dictionaries to achieve optimal search time even under time dependent distributions; however insert/delete times will be suboptimal at $O(\log n)$. In this paper, we present a lazy updating scheme which can be applied to weighted dictionaries to improve their amortized insert/delete performance when access probabilities decrease with time; optimality of search time is preserved. More specifically, let $r(k)$ be the number of distinct keys accessed since the last access to key k - that is $r(k)$ is the move-to-front *rank* of k. Let $r_{max}(k)$ be the maximum rank of k during its lifetime. Then our lazy update scheme enables the abovementioned data structures to perform search in $O(\log r(k))$ time and insert/delete in $O(\log r_{max}(k))$ time. We illustrate our lazy update scheme in the context of a new Biased Skip List data structure and show that our bounds are optimal.

*afe@eecs.cwru.edu. Department of EECS, Case Western Reserve University, Cleveland, OH.

†cenk@eecs.cwru.edu. Department of EECS, Case Western Reserve University and Department of CS, University of Warwick.

‡jonathan@dcs.warwick.ac.uk. Department of CS, University of Warwick, UK.

§rsinha@ciena.com.This work was done at Bell Laboratories. The author's current affiliation is Ciena Core Switching Division.

STOC'01, July 6-8, 2001, Hersonissos, Crete, Greece.

1. INTRODUCTION

The dictionary problem is a fundamental one in computer science. Given a totally ordered universe, the goal is to build a data structure that supports efficient *search*, *insert*, and *delete* operations on n keys. When the lookup probabilities for keys are uniform i.i.d. there are many textbook data structures such as red-black trees, 2-3 trees, AVL trees, skip-lists etc. that facilitate $O(\log n)$ time search, insert and deletes, matching the information theoretic lower bound.

In many applications, the access probabilities are non-uniform, which can be exploited to improve performance. Given n keys $k_1, k_2, \ldots, k_n$, let w_i be the *weight* of key k_i, which can be set to indicate the access likelihood of k_i; let $W = \sum_i w_i$. There are a number of well known, mainly tree based, data structures that provide search, insert and delete operations with running times that are a function of weight. The best bounds achievable are $O(\log(W/w))$ for search and $O(\log(W/\min(w^+, w, w^-)))$ for insert and delete, where w, w^- and w^+ are the weights of the key and its left and right neighbors respectively [18]. The first data structure to achieve these bounds were D-trees [14, 15]. Biased search trees [1], splay trees [23] and treaps [22] provide similar running times, while supporting efficient split and join operations. They also facilitate changing the weight of a key by δ in $O(\log((w+\delta)/w)))$ time. The bounds provided by splay trees are amortized, whereas treaps provide expected running times.

When the access probability of each key is i.i.d., over time these bounds can be made information theoretically optimal by assigning frequency counts as weights [2]. However, in many applications, the source of non-uniformity in access probabilities is *locality of reference*. Classical examples include memory/cache [24], disk [20] and buffer [7] management; we elaborate on some emerging applications in network traffic management later in this section.

In such applications, the access probability of any given key is not i.i.d., but is a *non-increasing* function of the time elapsed since the last access to the key. It is possible to adjust the weights of the keys so that the bounds for search are still optimal for these distributions. However, insert/delete times will suffer: Given key k, denote by $r(k)$ the number of distinct keys accessed (searched, inserted or deleted) since the last time k was accessed. In other words, $r(k)$ denotes the move-to-front (MTF) rank of k. If the weight of k is set to be $w = 1/2^{2\cdot\lfloor\log r(k)\rfloor+1}$ (and thus $W = 1 - 1/2^{\lfloor\log n\rfloor-1}$), then these data structures enable searching for that key in time

$$O(\log(W/w)) = O(\log 2^{2\cdot\lfloor\log r(k)\rfloor+1}) = O(\log r(k)).$$

This is optimal for *any* distribution in which search probability is a non-increasing function of time elapsed since last access.[1] However, because the weights are set to be $w = 1/2^{2\cdot\lfloor\log r(k)\rfloor+1}$ each insertion or deletion requires changing the weights of at least $\log n$ keys. Thus, inserts/deletes take $\Omega(\log n)$ time regardless of rank, which can dominate the overall performance of the data structure: consider a case where insertions and deletions are frequent and performed on keys with small ranks.

Focus of the paper. Our goal in this paper is to improve the insert/delete performance of weighted dictionaries while maintaining optimal search time, in applications where access probabilities are non-increasing functions of idle time. More specifically, we are interested in settings where the probability of accessing a more recently accessed key is at least as high as that of any other key which was accessed less recently.

There are many applications in which fast insert/deletes for recently accessed keys are essential. In firewall maintenance, one needs to cater to users and connections with different characteristics. For instance, a university firewall may need to cater to the HTTP access requests (searches, insertions, and deletions) of a few active users surfing the web; these correspond to keys with small rank. At the same time most users with open browsers (constituting the majority of firewall entries) may be inactive and not be generating firewall accesses. Moreover, there may be many TELNET connections which are long lived but usually require very little attention from the firewall. There is empirical evidence that scenarios of high bias in firewall access and other Layer 4 traffic are quite commonly observed [19, 12, 6]. Another application is web proxy caches, in which a small number of pages (e.g. stock news) stay very "hot", i.e. maintain a small rank, during their lifetime but are short lived. Other pages (e.g. personal web pages) may have less popularity but do live longer. See [3] for empirical evidence and discussion.

Contributions. Our main contribution is a lazy updating scheme which can be applied to all the above weighted dictionary data structures to improve insert/delete performance under the settings of interest. Let $r_{max}(k)$ be the maximum rank which a key k attains during its lifetime; thus $r_{max}(k)$ is a measure on how "hot" a key has been. Our lazy updating scheme enables a weighted dictionary data structure to perform inserts/deletes in amortized $O(\log r_{max}(k))$ time, while also enabling search in optimal $O(\log r(k))$ time. We show that under our settings these bounds are *optimal*: Given any sequence of search, insert and delete operations generated by a probability distribution described above, no data structure can achieve a total time of $o(\sum_{\forall k}\log(r_{max}(k)) + \sum_i \log(r(k_i)))$ where $r(k_i)$ is the rank of i^{th} key accessed (searched, inserted or deleted) and $\sum_{\forall k}\log(r_{max}(k))$ is the sum of the logarithms of the maximum ranks of each key ever inserted in the data structure.

We illustrate our lazy updating scheme on the *biased skip list* (BSL) data structure. BSL is an improvement on the Skip Lists, a data structure for accessing unbiased keys [21]. Introduced in [8, 9], BSL (without lazy updating) provides an expected search time of $O(\log r(k))$ and an expected insertion/deletion time of $O(\log n)$. It achieves this by modifying skip lists [21] through setting promotion probabilities non-uniformly.[2] BSL is different from other *weighted skip lists* [17, 13] as these variants promote keys via coin tosses biased according to their weights, from the bottommost level upwards. BSL (with lazy updating), on the other hand, confines the recently accessed keys to the higher levels of the skip list in order to facilitate fast insertions and deletions. Each key starts to appear at a level determined according its rank and is then promoted by a fixed probability independent of the rank of that key. This structure brings on new challenges for maintaining levels while facilitating fast searches; for instance, BSL randomly promotes keys to lower levels in addition to upper levels. As a result it can perform search in $O(\log r(k))$ expected time and insert/delete in $O(\log r_{max}(k))$ amortized expected time.[3]

Organization of the Paper. In Section 2 we describe the basics of BSL when the ranks of keys are fixed - hence keys do not move to front after they are searched. In Section 2.1 we show how to add update features to BSL and analyze its effect on performance. Our main result on fast amortized insertion and deletion times, via lazy updates; we present it in the context of BSL in Section 3. We then summarize our modification on D-trees and other weighted dictionaries and show that our bounds are best possible.

2. BIASED SKIP LIST (BSL)

Skip Lists in a nutshell. A skip list is a search data structure for n ordered keys [21]. It can be constructed in $O(n)$ expected time if the keys are given in sorted order; search, insert, and delete operations each take $O(\log n)$ expected time. To construct a skip list, we start with a doubly linked list of all keys in the data structure in sorted order; this makes up level $\log n$ of the skip list.[4] We iteratively build subsequent levels $\log n - 1, \ldots, 2, 1$ in a randomized fashion. Each level i is a sorted linked list consisting of a subset of the keys in level $i+1$ obtained as follows: each key in level $i+1$ is copied to level i independently with probability 1/2. Each key has links to its copies (if they exist) on adjacent levels. Because there are $\log n$ levels, only one key is expected on level one.

[1] Because k is the $r(k)^{th}$ most recently accessed key, there are $r(k)$ keys whose search time should be at most that of k's. Thus the best possible search time for k is $O(\log r(k))$.

[2] Both the skip list and BSL can be seen as an application of the *fractional cascading* technique of Chazelle and Guibas [5], which enables one to search for an item among d lists out of n lists in time $O(d + \log N)$; here N is the total size of the n lists. One way to dynamically implement fractional cascading is given in [16], which provides an amortized insertion and deletion time of $O(\log\log N)$, while supporting $O((d + \log N)\log\log N)$ search time.

[3] One interesting application of BSL and other weighted dictionaries under lazy updates is move-to-front (MTF) based data compression [2] for highly dynamic alphabets. MTF based data compression has attracted new attention due to its use in the context of Burrows-Wheeler transform [4]). Recent results on space efficient data structures are based on such compression [10].

[4] From this point on $\log n$ denotes $\lceil\log_2 n\rceil$

To search for a key k in a skip list we start from the smallest (leftmost) key in the highest level. On each level, we follow the linked list to the right until we encounter a key which is greater than k. When that happens, we take a step to the left and go down one level (which deposits us in a key which is less than k). The search ends when k is found or when, at the lowest level, a key greater than k is reached. To delete a key, we perform a search to find its highest occurrence in the structure and delete it from every level it appears in. To insert a new key k, we first search for k in the skip list. This search locates the correct place to insert k in the bottom level. Once the key is inserted into the bottom level, a fair coin is tossed to decide whether to copy it on to the level above the current one. If k is indeed copied, the procedure is repeated; otherwise the insertion is complete.

BSL. BSL's main feature is an assignment of *ranks* to the keys maintained. We first describe a simple variant when the ranks are static. BSL keeps all keys in a linked list in ascending rank order. We partition this list into *classes* $C_1, C_2 \dots, C_{\log n}$ contiguously: if keys x and y are in classes C_k and C_{k+1} respectively, then $r(x) < r(y)$. The class sizes are geometric, $|C_i| = 2^{i-1}$.

Construction. BSL is constructed randomly in a bottom-up fashion. It comprises several levels, each one being a doubly linked list of keys in sorted order. The levels of the data structure from the bottom to the top are named $L_{\log n}, L'_{\log n-1}, L_{\log n-1}, \dots, L_2, L'_1, L_1$. The bottommost level, $L_{\log n}$, includes all of the keys in all classes. To obtain the keys in any level L'_i we copy the following keys from level L_{i+1}: (1) all keys from classes $C_i, C_{i-1} \dots, C_1$ (by definition all should be present in L_{i+1}) – there will be $2^i - 1$ of them, and (2) a subset of the remaining keys in L_{i+1} each picked independently with probability $1/2$ – the expected number of these keys will be 2^{i-1}. We obtain the keys in level L_i slightly differently by copying the following keys from L'_i: (1) all keys from classes $C_{i-1}, C_{i-2} \dots, C_1$, and (2) a subset of the remaining keys in L'_i each picked independently with probability $1/2$. For an example of a BSL see Figure 1.

LEMMA 1. *A BSL with n keys can be constructed in $O(n)$ expected time, provided the keys are given in sorted order.*

PROOF. Let us look at how many times an element will be copied up. Since the copying probability is $1/2$, the expected number of levels on which an element will be copied as a result of the coin toss is 1. Let us count the automatically made copies.

$C_{\log n}$, which is of size $(n+1)/2$, is not subject to automatic copying. $C_{\log n-1}$, which is of size $(n+1)/4$, is copied automatically on to two levels. $C_{\log n-2}$ similarly gets copied on to four levels; in general $C_{\log n-p}$ similarly gets copied on to $2p$ levels. Observing the number of automatic copies on each level and summing over all levels (or over all classes), we see the the total number of automatically made copies is $O(n)$; combined with the randomly made copies, the total number of keys (with repetitions) in the data structure is $O(n)$. One needs to spend $O(1)$ time per copy, as a result, the construction time is $O(n)$. $\square$

Search. Searching for a key k in a BSL is similar to searching in a regular skip list. We start from the smallest key in the topmost level. On each level, we follow the linked list to the right until we encounter a key which is greater than k. Then we take a step back (left) and go down one level. We end the search successfully whenever we locate k, unsuccessfully when we reach a key greater than k in the bottommost level, indicating that k is not present in the data structure.

LEMMA 2. *A successful search for an element k in a BSL of n keys takes $O(\log r(k))$ expected time, whereas an unsuccessful search takes $O(\log n)$ expected time.*

PROOF. First, consider a successful search. Due to the partitioning into classes, key k belongs to class C_c, where $c \le \log r(k) + 1$. To bound the search time, we bound the horizontal and vertical distance that we travel until we encounter k. The construction method for the BSL dictates that all the keys in Class C_c are present on level L'_c, which is at depth $2c$ from the top of the BSL. Once we reach level L'_c, we are guaranteed to find k, thus we travel at most $2c-1$ vertical links from the top of the BSL. We now analyze how many (expected) horizontal links are traversed per level. On the top level, there is an expected single key, corresponding to at most 2 links. Let us now consider our actions on level l. We must have come to level l from the level above, say l', from some key k'. Let k'' be the key immediately to the right of k' on level l'. By our search strategy, $k' < k < k''$. Thus, on level l, in the worst case, we need to traverse the links between k' and k'', which number one more than the number of keys on level l between k' and k'' not copied up to level l'. In a situation where all keys are always subject to the randomized process to decide whether they will be copied up, the expected number of keys that are skipped between two keys that are copied is 1. In our scheme, since some keys get copied automatically, the expected number is even less. Thus, the expected number of links on level l, between k' and k'', is less than 2. Since we visit at most $2c-1$ levels, traversing at most an expected 2 links on each level, the total (expected) running time to find k in the data structure is $O(c)$; substituting the value for c, this is $O(\log r(k))$.

For an unsuccessful search, the number of horizontal links traversed per level stays the same; however, one has to go all the way down to the bottom of the data structure. Therefore, the vertical distance traveled is $O(\log n)$ levels, with an expected constant number of links per level, giving an overall expected running time of $O(\log n)$. $\square$

2.1 Dynamic BSL

BSL as described above is a static structure and does not support fast updates in the form of rank changes, insertions or deletions. To observe this, note that elements of higher rank have many (up to $2\log n$) copies in the data structure. Note that in the implementation of an MTF list, an insertion always brings a key to the topmost class, which, in this scenario, requires the making of $2\log n$ copies. Deletions can also involve the deletion of many copies. In this section we describe a modified version of BSL, the dynamic BSL, which allows for fast updates. A fundamental difference between the dynamic BSL and the static BSL is in how each of the levels are constructed. To facilitate efficient insertions and deletions, dynamic BSL must avoid keeping copies of all keys at the bottom level - if a key maintains a very small rank throughout its lifetime, it would be inefficient to keep copies of it at lower levels.

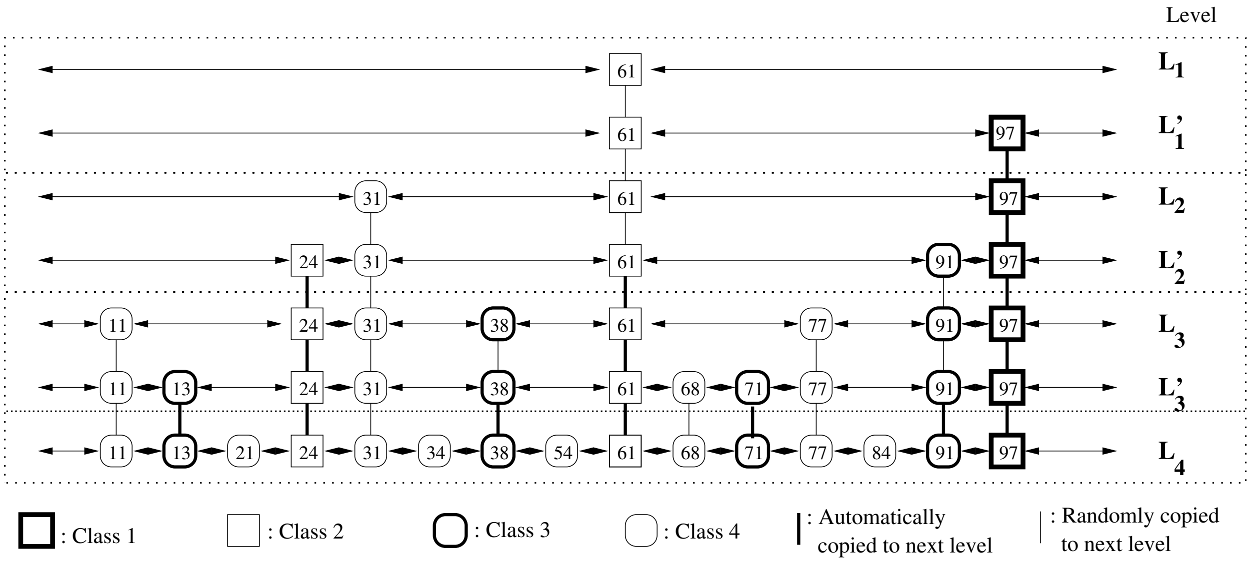

Figure 1: A simpler version of BSL and its levels.

Construction. To construct the dynamic BSL we start with the bottommost level $L_{\log n}$ and make our way up. Level $L_{\log n}$ is made up of all the keys that belong to class $C_{\log n}$. For $i < \log n$, an upper level L'_i is constructed from level L_{i+1} by copying those keys of level L_{i+1} which are chosen independently with probability $1/2$. L'_i also includes all keys in class C_i; these are called the *default keys* of L'_i. To facilitate efficient search some of the default keys of L'_i are copied to the lower level L_{i+1}. This is done independently with probability $1/2$. Each key copied to level L_{i+1} may be copied further to lower levels $L'_{i+1}, L_{i+2}, L'_{i+2}, \ldots$ with independent coin tosses. Once we have the keys in level L'_i, we construct level L_i by simply copying those keys in level L'_i which are chosen independently with probability $1/2$. For an example of a dynamic BSL see Figure 2.

LEMMA 3. *A dynamic BSL with n keys can be constructed in $O(n)$ expected time, provided the keys are given in sorted order.*

PROOF. The probabilistic copying down of keys does not add significantly to the number of keys per level since the expected number of keys which are copied down decreases faster than the expected number of keys on each level as we go down. Thus, we still maintain that any level L_i or L'_i will contain $O(2^i)$ keys. Let us look at the cost on a per level basis. Level L'_i is formed from the previous level and the class C_i in time $O(|C_i|)$. Level L_i is formed from L'_i again in $O(|C_i|)$ time. The only remaining cost is the cost of copying down of keys originating from L'_i. Note that the expected number of times that a key will be copied down is just under 1. Thus, the expected total number of copies from C_i that will lie below L'_i is $O(|C_i|)$. To copy a key down, we go to its left neighbor on the same level, attempt to go down or keep going left until we can go down one level. After that, we go right until we come to a key greater than the one that we need to copy, and insert our key right before it. The expected number of steps that we take to the left is $O(1)$. This is because the probability that a key on this level will not exist on the previous level is less than $1/2$. With a similar argument, the expected number of steps that we take to the right is $O(1)$. Thus, we spend $O(1)$ time copying one key down, and $O(|C_i|)$ time taking care of the copying of the entire C_i. All the costs associated with levels L'_i and L_i hence add up to $O(|C_i|)$. The sum of these costs over all i gives us $O(n)$. $\square$

Search. To search for a key k in a dynamic BSL we start from the smallest key in the topmost level. Similar to the static BSL, on each level we follow the linked list to the right until we encounter a key j which is greater than or equal to k. If $j = k$, we successfully terminate the search. Otherwise we go left until we reach a key l that has a copy one level below.[5] Then we simply go one level down and iterate starting from the copy of l at that level. If the level is the bottommost one and while going right we have encountered a key which is larger than k, then we conclude that k is not present in the BSL, and terminate.

Rank Assignment. When a key is searched for (or inserted/deleted), the ranks of some of the other keys change according to a *rank assignment policy.* We describe how BSL can be adapted to implement MTF based rank assignment.

3. LAZY UPDATING IN BSL

To facilitate efficient implementation of insertions and deletions, we employ a *lazy updating scheme* of levels and allow flexibility in class sizes. The size of a class C_i is allowed to be in the range $[2^{i-1}, 2^{i+1})$, while its default size is 2^i. The lazy updating allows us to keep an insertion or deletion local to upper levels for most insertions/deletions. As a result, the insertion or deletion of a key k takes $O(\log r_{max}(k))$ amortized time, where $r_{max}(k)$ is the maximum rank of key k in its lifespan.

Search. In accordance with the MTF based rank assignment policy, when a key k is searched for its rank becomes 1 and its default level is changed to L'_1; the ranks of all keys

[5] Remember that a key at level L'_i does not necessarily have a copy in level L_{i+1}.

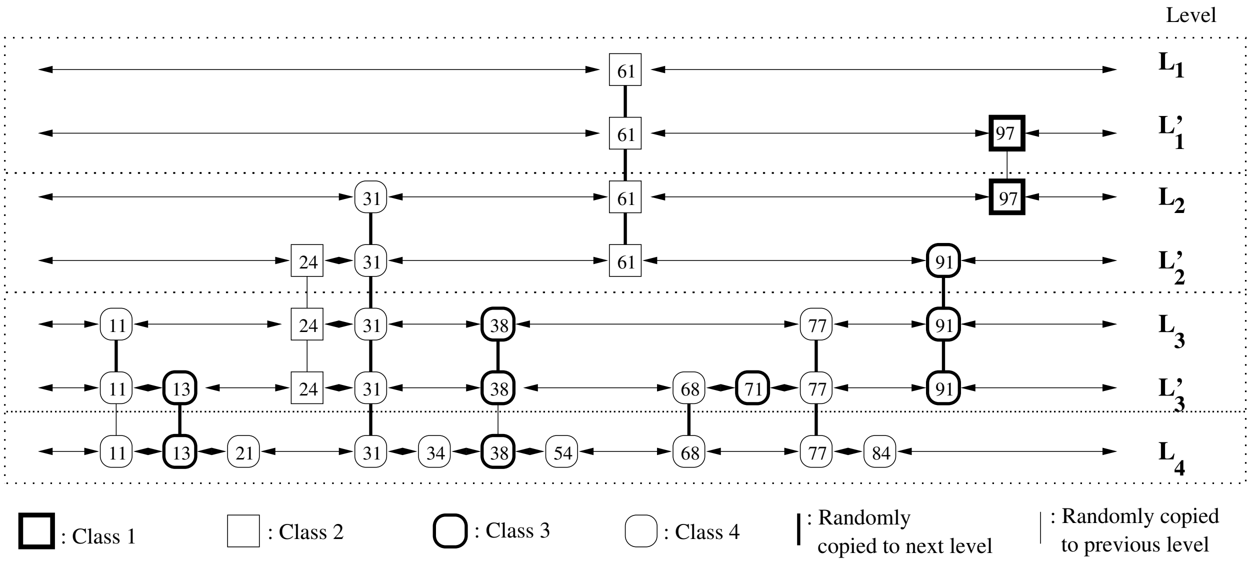

Figure 2: Dynamic BSL.

whose rank was smaller than k are incremented by one. The rank changes are reflected in the BSL without changing the number of keys in any class or level; the lowest ranked key in each class, upto the class originally containing k, is moved to the next class and the corresponding key entries in BSL are shifted down two levels. The class sizes may only change after an insertion or a deletion.

LEMMA 4. *The BSL with lazy updates enables searching for an existing key k in the data structure in $O(\log r(k))$ expected time. Searching for a key which is not in the data structure takes $O(\log n)$ expected time.*

PROOF. To analyze the number of vertical steps taken during the search, note that, even with the flexible class sizes, k will be at most $O(\log r(k))$ levels down from the top. On each level that we cross, the expected number of steps that we need to take to the right to locate a key greater than k is at most 2 with an argument similar to that given in the proof for the static BSL. We might, however, need to go left in order to find a key from which we can go down, and after going down we may need to go right to compensate for the left move. The expected number of steps for these operations is $O(1)$, again with a similar argument. Therefore, the number of horizontal steps per level is constant, and the whole search takes $O(\log r(k))$ expected time. Adjusting the ranks of the keys also takes $O(\log r(k))$ expected time, since $O(\log r(k))$ keys must be shifted two levels down, taking $O(1)$ expect time each. If the key is not in the data structure, the number or horizontal steps per level does not change; however, we need to look at all $O(\log n)$ levels to be sure the key is not present. □

Insertion/deletion. When a key k is inserted to the BSL it is given rank 1 and the ranks of all keys in the data structure are incremented by one. After an insertion, if the size of a class C_i reaches its upper limit, i.e. 2^{i+1}, then the highest ranked half of the keys in C_i change their default level from L_i' to L_{i+1}'. This is done by moving the topmost and bottommost levels of each such key by two. One can observe that such an operation can be very costly; if, for example, all classes $C_1, C_2, \ldots, C_l$ are full, an insertion will change the default levels of 2 key in class C_1, 4 keys in class C_2, and in general 2^i keys from class C_i. However it is possible to amortize more costly insertions to less costly ones: we show that the average insertion time for a key k is $O(\log r_{max}(k))$, where $r_{max}(k)$ denotes the maximum rank of k in its lifespan.

To delete a key k we first search for k in BSL, which will change its rank to 1 and simply delete it from all the levels it has copies. If the number of keys in class C_i after the deletion is above its lower limit of $2^{i-1} - 1$ then we stop. Otherwise, we go to the next class C_{i+1} and move its lowest ranked 2^i keys to class C_i by moving their default levels up by two. Similar to insertion it is possible to amortize more costly deletions to less costly ones; the main result in this section is that the average deletion time for a key k is $O(\log r_{max}(k))$.

THEOREM 5. *The BSL enables insertion or deletion of a key k in $O(\log r_{max}(k))$ amortized expected time, where $r_{max}(k)$ is the maximum rank of key k during its lifespan.*

PROOF. Observe that the time for an insertion or a deletion is determined solely by the number of keys in each level at that instant. Because a search does not change the number of keys in any level, they can be ignored in the analysis of insertion and deletion times. Also observe that given a sequence of insertions and deletions on an initially empty BSL, the number of keys in each level is determined by the order in which insertions and deletions are performed. This implies that we can ignore the values of the keys inserted and deleted when analyzing the time BSL needs per insertion/deletion. Let $i[k]$ denote the insertion of key k, and $d[k]$ its deletion. The sequence $i[k], i[l], d[k], d[l]$ on an empty BSL has the same running time as the sequence $i[k], i[l], d[l], d[k]$.

Let $S = s_1, s_2, \ldots, s_n$ represent a sequence of insertions and deletions on an empty BSL, where $s_j = i$ denotes an insertion and $s_j = d$ denotes a deletion; for instance, $S = i, i, d, d$ denotes a sequence of two insertions followed by two deletions. For clarity, we assume that S starts on an empty BSL and has n insertions and n deletions; the theorem holds for the general case as well. A *matching* is a pairing of insertions to deletions in a sequence such that if the paired operations were performed on the same key (with no other operations on the same key between them), this would yield a legal sequence of operations.[6] Using this notion, we prove the theorem in two steps. (1) We first describe a specific matching scheme and show that, given a sequence S matched according to this scheme, the execution of S takes $O(\sum_{k \in K} \log r_{max}(k))$ expected time. Here, K is the set of all inserted/deleted keys.[7] (2) We then show that the matching described above provides a lower bound for $\sum_{k \in K} \log r_{max}(k)$ for *any* matching over S. Since the cost of a sequence is independent of the matching, we conclude our proof.

Part (1). The expected time for an insertion/deletion in a BSL is proportional to the total number of keys which move across classes (from C_i to C_{i+1} or to C_{i-1}). Our analysis hence concentrates on the total number of key moves between each C_i and C_{i+1} during the execution of sequence S.

Let $S(l, j)$ denote the subsequence $s_l, s_{l+1}, \ldots, s_{j-1}, s_j$ and let $i(l, j)$ and $d(l, j)$ denote the number of insertions and deletions in $S(l, j)$ respectively. We partition S into subsequences that we call *blocks*. Block 1 is defined as $S(1, j_1)$ where j_1 is the smallest position for which $|i(1, j) - d(1, j)| = 2$. Inductively block b is defined as $S(j_{b-1} + 1, j_b)$ where j_b is the smallest position for which $|i(j_{b-1} + 1, j_b) - d(j_{b-1} + 1, j_b)| = 2$.

A given block b is called an *insertion block* if $i(j_{b-1} + 1, j_b) > d(j_{b-1} + 1, j_b)$, otherwise it is called a *deletion block*. We replace each insertion block with an "i" and each deletion block with a "d" to obtain a sequence that we call the *first summary* of S, denoted S^1. Iteratively, we can obtain S^m, the m^{th} summary of S, by applying our aforementioned block partitioning scheme on S^{m-1}. For generality we set $S^0 = S$. See Figure 3 for an example. Even though a block in S^i may contain only two characters, this is in essence a summary of many more operations in S. We say that a block in S^m *covers* a subsequence of operations in S to indicate that the subsequence is a subset of the operations summarized by that block. For example, in Figure 3, the first insertion block in S^2 (which corresponds to the first "i" in S^3) covers $S(1, 12)$. The full subsequence covered by an insertion block in S^{m-1} (the single character that summarizes that block is in S^m) contains 2^m more "i"s than "d"s; the converse is true for a deletion block.

Recall that a matching of a sequence is a pairing of insertions to deletions. For our proof, we use the *parentheses matching*, denoted P, where one can imagine each insertion being replaced by a "(" and each deletion being replaced by a ")". Thus, every insertion at the l^{th} operation is matched with the closest deletion at the j^{th} operation such that $i(l, j) = d(l, j)$ [11]; see Figure 4 for an example.

We now bound the amortized cost of an insertion/deletion in S matched in this manner.

[6]We *do not* have to know what the actual keys were in the sequence; we *assign* the matching ourselves.

[7]If the same key k is inserted and deleted multiple times, for simplicity, consider each time as a separate key k_i.

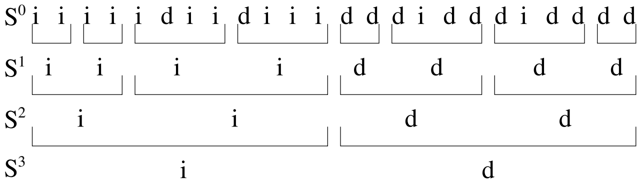

Figure 3: A sequence of insertions and deletions and its summaries.

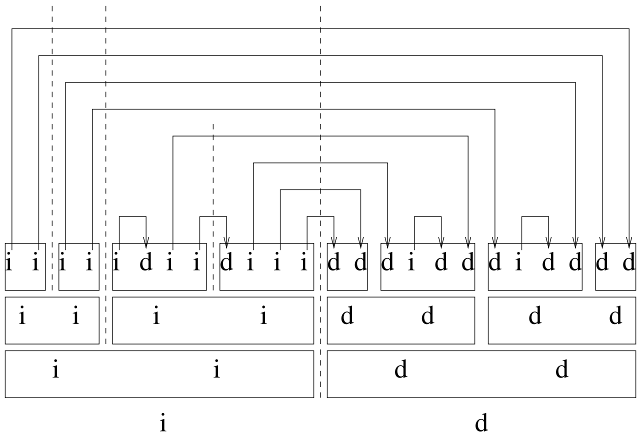

Figure 4: Parentheses matching of "i"s and "d"s and their incurred costs.

LEMMA 6. *The expected amortized cost of the insertion or deletion of a key k in a sequence S matched with respect to the parentheses matching is $O(\log r^P_{max}(k))$, where $\log r^P_{max}(k)$ denotes the highest rank that k obtains under the parentheses matching in its lifetime.*

PROOF. During a sequence of insertions we notice that keys move from C_1 to C_2 every other insert, moving 2 keys between the classes. If we mix insert and delete operations, then keys move from C_1 to C_2 every time the number of insertions are 2 more than the number of deletions; that is every insertion block in S^0 results in a move. Extending this analysis, a fundamental observation for our proof is that the number of operations which cause keys to move from C_i and C_{i+1} and vice versa is bound from above by $|S^i|$ and 2^i keys move between the classes.

Consider the operations in S during which the highest class affected was C_{m+1} (keys were moved between C_m and C_{m+1}). The total number of keys moved during each such operation is 2^m. Since there are at most $|S^m|$ of those operations, we perform what we call an "S^m charging phase". We assign each such operation to a different character in S^m. Consider operation s_j; the character that it is assigned signifies a block in S^{m-1} (WLOG, a deletion block) covering a subsequence $S(k, l)$ of S. $S(k, l)$ has 2^m more "d"s than "i"s. Then, because of our particular matching, there must be exactly 2^m "d"s in $S(k, l)$ which are not matched within $S(k, l)$. Charge each such "d" 2 of the 2^m moves caused by s_j.

Clearly, this way, all moves caused by S are charged to some operation. Let us count the number of charges that each operation receives. Consider operation s_j. Assume

WLOG that it is a deletion of key k, matched to an insertion s_m. s_j is charged either 0 or 2 during each "S^m charging phase"; consider a specific S^m, and the block (call it B) in S^{m-1} that covers s_j. s_j will be charged only if s_m is not covered by B. Note that there can be at most $r^P_{max}(k)$ keys between s_j and s_m in S. The minimum number of element covered by a block is doubled from one summary to the next. Together with our matching, this implies that s_m and s_j cannot be covered by different blocks in the summaries after $S^{\log r^P_{max}(k)}$, thus s_j cannot be charged during the corresponding charging phases. The total charges that s_j receives then is $O(\log r^P_{max}(k))$, which gives us the desired expected time. □

Part (2). We now bound the maximum ranks in the parentheses matching P in terms of the maximum ranks in *any* matching M.

LEMMA 7. *Given a sequence S of n insertions and n deletions and an arbitrary matching M of insertions to deletions,*

$$\sum_{k\in K} \log r^P_{max}(k) \le \sum_{k\in K} \log r^M_{max}(k)$$

where $\log r^M_{max}(k)$ *denotes the maximum rank of k in M.*

PROOF. For each insertion (resp. deletion) operation s_j, denote by k_j the key inserted (resp. deleted). For any matching M, let $\phi_M(s_j)$ denote the deletion (resp. insertion) operation s_k that matches s_j in M.[8] Notice that $\phi_M = \phi_M^{-1}$.

To show the above inequality, we start with an arbitrary matching M and change it in stages into P, while reducing the total maximum rank with each stage. Let M_i denote the matching obtained after stage i. We now show how to obtain M_{i+1} from M_i. Let s_l be the leftmost insertion in S such that $\phi_{M_i}(s_l) \ne \phi_P(s_l)$. Let $s_q = \phi_P(s_l)$, and $s_m = \phi_{M_i}(s_l)$. Then, by definition of the parentheses matching, $q > m$. Additionally, $i[l,m] > d[l,m]$ (otherwise we would have $s_m = \phi_P(s_l)$).

Let $p = \phi_{M_i}(s_q)$, which must be an insertion. Repeat $p \leftarrow \phi_{M_i}(\phi_P(p))$ until $p = s_j$ for some $j < m$. Note that the termination condition will eventually be satisfied since $i[l,m] > d[l,m]$ and thus, there is at least one insertion s_i with $l < i < m$ such that $\phi_{M_i}(s_i)$ is a deletion to the left of s_m. Now define M_{i+1} as follows. Let $s_h = \phi_{M_i}(s_j)$ (it may be that $h = q$). Set $\phi_{M_{i+1}}(s_j) = s_m$ and $\phi_{M_{i+1}}(s_l) = s_h$; for all other operations s_u, $\phi_{M_{i+1}}(s_u) = \phi_{M_i}(s_u)$. See Figure 5 for the changes made in stage $i+1$.

First, note that as long as $M_i \ne P$, this algorithm will progress without going into a loop (s_l is matched to an operation further to the right each time until it is matched according to P); when P is finally obtained, it will stop. We now need to show that

$$\sum_{k\in K} \log r^{M_{i+1}}_{max}(k) \le \sum_{k\in K} \log r^{M_i}_{max}(k)$$

for valid i. Consider the insertions s_l, s_j and deletions s_m, s_h above. Those are the only operations whose matchings have

[8] We make two notational simplifications. First, do not mention that S must be a parameter to ϕ. Second, we assume that the data structure representing each operation s_j carries information about j as well as whether the operation is an insertion or deletion. This way ϕ can obtain this information from its parameter s_j.

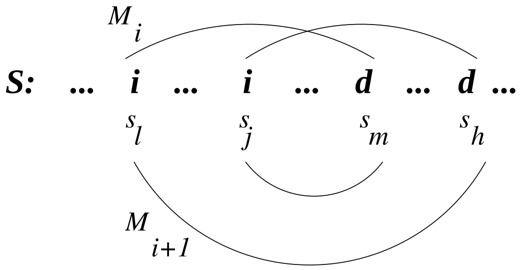

Figure 5: The changes from M_i to M_{i+1}.

changed. Note that $l < j < m < h$. We first argue that $r^{M_{i+1}}_{max}(k_l) - r^{M_i}_{max}(k_l) \le r^{M_i}_{max}(k_j) - r^{M_{i+1}}_{max}(k_j)$.

Consider the interval $S(m,h)$, and the left hand side of the inequality. In this interval, the rank of k_l with respect to M_{i+1} increases by 1 with each insertion and decreases by 1 with each deletion. This is because all the deletions in this interval are matched to an insertion which takes place after s_l due to the construction of M_{i+1}. Now consider the right hand side. How much would the rank of k_j increase by going from M_{i+1} to M_i? The rank of k_j with respect to M_i goes up with each insertion in $S(m,h)$, but it does not necessarily go down with each deletion, since the corresponding insertion might have been performed before s_j.

Thus, any increase in the maximum rank of k_l due to its longer lifespan in M_{i+1} is at least made up for by the decrease in the maximum rank of k_j, and the inequality follows. Note that if $a + b \le c + d$ and $max(c,d) \le max(a,b)$, then $\log a + \log b \le \log c + \log d$. Observing that $r^{M_{i+1}}_{max}(k_l) \ge max(r^{M_i}_{max}(k_j), r^{M_i}_{max}(k_l))$, we have that $\log r^{M_{i+1}}_{max}(k_l) - \log r^{M_i}_{max}(k_l) \le \log r^{M_i}_{max}(k_j) - \log r^{M_{i+1}}_{max}(k_j)$.

It can be seen that for all the other insertions s_o in S, unless $l < o < j$, the maximum rank that k_o attains remains the same by going from M_i to M_{i+1}. For $l < o < j$, the maximum rank of k_o either stays the same, or decreases. Thus, the change from M_i to M_{i+1} can only decrease the value of the summation, and the lemma follows. □

Consider S together with information about the actual keys inserted/deleted. It must be consistent with some matching M. The cost is independent of the matching; S matched according to M has the same cost as S matched according to P, which is $O(\sum_{k\in K} \log r^S_{max}(k))$. Combining with the above lemma, Theorem 5 follows.

Optimality of Insertions and Deletions. We now show that the upper bounds we provide for search, insertion and deletion are best possible.

LEMMA 8. *Given a set of n keys, suppose that each key is assigned a unique rank in $\{1 \dots n\}$, such that for any two keys k and l, with $r(k) > r(l)$, the probability of accessing k is at least as much as the probability of accessing l. Then no comparison based data structure can perform a search operation for any key k in time $o(\log r(k))$.*

PROOF. The lemma follows immediately from the observation that if the data structure supports search for k in time t, it should provide support searching for $\Omega(r(k) - 1)$ keys (whose access probabilities are at least as much as k) in $O(t)$ time. □

LEMMA 9. *Any comparison based data structure D for maintaining totally ordered keys, which assigns a unique key $r(k)$ to each key k and supports searching for key k in $\Omega(\log r(k))$ time, needs to spend $\Omega(\log r_{max}(l))$ time for inserting or deleting a key l whose maximum rank in its lifetime is $r_{max}(l)$.*

PROOF. Consider a sequence of n insertions and n corresponding deletions to D, without any searches. Let $i = 1 \ldots n$ denote all the keys that get inserted and deleted; let $r_t(i)$ denote the rank of key i at time t and let s_i and e_i denote the insertion and deletion times of key i. Consider the sequence of lists $L = L_1, L_2, \ldots, L_{2n}$, where L_t is a list of keys, where each key i is sorted according to its rank $r_t(i)$. Observe that $\frac{\sum_{t=1}^{2n} \sum_{i=1}^{n} r_t(i)}{\sum_{i=1}^{n}(e_i - s_i + 1)}$ (the average rank per key per time step), upper bounds $\frac{1}{n}\sum_{i=1}^{n} r_{max}(i)$. (Each key provides an envelope in L: $r_t(i)$ is either $r_{t-1}(i)+1$ or $r_{t+1}(i)-1$ and if $r_{t-1}(i) > r_{t-1}(j)$, then $r_t(i) > r_t(j)$. Notice that the envelopes of all items behave identically; if the rank of one item increases (decreases), the rank of all other items increase (decrease).) This implies that for an "average" key k, its rank averaged over its lifetime is in the order of its maximum rank. Notice that in order to delete a key k, one must perform a search on it. As shown above, searching for k at time step t requires $\Omega(\log r_t(k))$ time which is equal to $\Omega(\log r_max(k))$ when averaged over all keys k and time steps t. □

4. LAZY MAINTENANCE OF WEIGHTED DICTIONARIES

We apply the lazy class maintenance scheme we introduced for BSL to D-trees and other weighted dictionaries. To do this we relax the constraint on maintaining $W = 1 - 1/2^{\lfloor \log n \rfloor - 1}$ and insisting that the weight of key k is $w = 1/2^{2 \cdot \lfloor \log r(k) \rfloor + 1}$. This relieves the burden of having to reflect the change in total weight to all classes without being penalized in search time.

THEOREM 10. *Consider any weighted dictionary scheme, which, for any given any key with weight w, can perform search, insert and delete operations in $O(W/w)$ time and double or halve its weight in $O(1)$ time; examples include D-trees, splay trees and treaps. Such a dictionary scheme can be modified to achieve $O(\log r(k))$ amortized search time and $O(\log r_{max}(k))$ amortized insertion/deletion time, which are optimal when access probabilities are a non-increasing function of idle time.*

PROOF. The crux of the proof lies in the fact that one can implicitly represent the class structure that we relied on in our earlier proofs by assigning each key in a class an appropriate weight. This implies that a class change operation can simply be implemented as a weight change operation.

Basically to represent that an key is in class C_i, we assign it weight $1/2^{2i-1}$. Because the size of class C_1 is upper bounded by 2, the size of class C_2 is upper bounded by 4 and in general the size of class C_i is upper bounded by 2^i, the total weight of keys W will be at most:

$$\sum_i 2^i \cdot 1/2^{2i-1} = \sum_i 1/2^{i-1} \leq 2.$$

Given any key k, the lazy updating scheme ensures that its class $C(k)$ is one of $C_{\log \lfloor r(k) \rfloor - 1}, C_{\log \lfloor r(k) \rfloor}, C_{\log \lfloor r(k) \rfloor + 1}$. Thus its weight $w(k)$ is at most $1/2^{2 \log \lfloor r(k) \rfloor - 1}$ and at least $1/2^{2 \log \lfloor r(k) \rfloor + 1}$.

To search for a key k, we first perform weighted dictionary search, which can be done in $O(W/w(k)) = O(\log r(k))$ time. Then for all $C_i = C_1 \ldots C(k) - 1$, we update the class of the highest ranked key in each class C_i to C_{i+1}. This is done by changing the weight of each key from some w to $w/4$, which will only take $O(1)$ time each. Thus the whole process requires $O(\log r(k))$ time in total.

To insert or delete the key, we simply perform the necessary class changes of keys through appropriate weight changes. By following the same steps for search, it is not difficult to see that the claimed time bounds can be achieved. □

5. REFERENCES

[1] S. Bent, D. Sleator, and R. Tarjan. Biased search trees. *SIAM Journal of Computing*, 14, 1985.

[2] J. L. Bentley, D. S. Sleator, R. E. Tarjan, and V. Wei. A locally adaptive data compression scheme. *Communication of the ACM*, 29:320–330, 1986.

[3] L. Breslau, P. Cao, L. Fan, G. Phillips, and S. Shenker. Web caching and zipf-like distributions: Evidence and implications. In *Proc. IEEE INFOCOM*, 1999.

[4] M. Burrows and D. Wheeler. A block-sorting lossless data compression algorithm. *DEC SRC Research Report*, 124, 1994.

[5] B. Chazelle and L. J. Guibas. Fractional cascading. In *Algorithmica*, volume 1, 1986.

[6] K. Claffy, H.W.Braun, and G.C.Polyzos. A parameterizable methodology for internet traffic flow profiling. In *IEEE Journal on Selected Areas in Communications*, 1995.

[7] A. Dan and D. Towsley. An approximate analysis of the LRU and FIFO buffer replacement schemes. In *Proc. ACM SIGMETRICS*, 1990.

[8] F. Ergun, S. Mittra, C. Sahinalp, J. Sharp, and R. Sinha. A dynamic lookup scheme for bursty access patterns. In *Proceedings of IEEE INFOCOM*, 2001.

[9] F. Ergun, C. Sahinalp, J. Sharp, and R. Sinha. Biased skip lists for highly skewed access patterns. In *Proceedings of ALENEX*, 2001.

[10] P. Ferragina and G. Manzini. Opportunistic data structures with applications. In *Proc. IEEE Symposium on Foundations of Computer Science*, 2000.

[11] R. Gutting and D. Wood. The parentheses tree. *Information Science*, 27, 1982.

[12] S. Lin and N. McKeown. A simulation study of IP switching. In *Proc. ACM SIGCOMM*, 1997.

[13] C. Martinez and S. Roura. Optimal and nearly optimal static weighted skip lists. Technical report, Universitat Politecnica de Catalunya, 1995.

[14] K. Mehlhorn. Dynamic binary search. *SIAM Journal of Computing*, 8, 1979.

[15] K. Mehlhorn. Arbitrary weight changes in dynamic trees. *RAIRO Inform. Theor.*, 15, 1981.

[16] K. Mehlhorn and S. Naher. Dynamic fractional cascading. *Algorithmica*, 5, 1990.

[17] K. Mehlhorn and S. Naher. Algorithm design and software libraries: recent developments in leda project. In *Algorithms, Software, Architectures, Information Processing*, volume 92. Elsevier, 1995.

[18] K. Mehlhorn and A. Tsakalidis. Data structures. In *Handbook of Theoretical Computer Science*, volume Algorithms and Complexity. Elsevier, 1997.

[19] S. Mittra and A. Basu. Packet classification: An argument for a working set model. Technical report, Bell Laboratories, 1999.

[20] E. O'Neil, P. O'Neil, and G. Weikum. The LRU-K page replacement algorithm for database disk buffering. In *Proc. 1993 ACM-SIGMOD Conference on Management of Data*, 1993.

[21] W. Pugh. Skip lists: a probabilistic alternative to balanced trees. *Communications of the ACM*, 33,6, 1990.

[22] R. Seidel and C. Aragon. Randomized search trees. *Algorithmica*, 16, 1996.

[23] D. Sleator and R. Tarjan. Self-adjusting binary search trees. *Journal of the ACM*, 32, 1985.

[24] A. Tanenbaum. *Operating Systems: Design and Implementation, 2nd edition*. Prentice-Hall, 1997.