Data Structures with Unpredictable Timing

Darrell Bethea and Michael K. Reiter

University of North Carolina, Chapel Hill, NC, USA

Abstract. A range of attacks on network components, such as algorithmic denial-of-service attacks and cryptanalysis via timing attacks, are enabled by data structures for which an adversary can predict the durations of operations that he will induce on the data structure. In this paper we introduce the problem of designing data structures that confound an adversary attempting to predict the timing of future operations he induces, even if he has adaptive and exclusive access to the data structure and the timings of past operations. We also design a data structure for implementing a set (supporting membership query, insertion, and deletion) that exhibits timing unpredictability and that retains its efficiency despite adversarial attacks. To demonstrate these advantages, we develop a framework by which an adversary tracks a probability distribution on the data structure's state based on the timings it emitted, and infers invocations to meet his attack goals.

1 Introduction

An adversary's ability to predict the timing characteristics of selected interactions with a networked component is instrumental in a wide range of potential attacks on that component or the network it defends. For example, algorithmic denial-of-service attacks depend on the adversary crafting requests that he can predict will be particularly costly for the component to process (e.g., [1–3]). Other attacks can benefit from predictable timings, whether they be expensive or not. For example, remote timing attacks on components that use cryptographic keys (e.g., [4, 5]) benefit if the adversary is able to predict the processing time *other* than that involving the cryptographic key being cryptanalyzed, so that this "noise" can be subtracted from the observed timings to obtain those timings related to the key itself.

In this paper we abstract from these scenarios the basic problem of developing data structures for which the timing of any particular operation is unpredictable. We consider an adversary who knows the implementation of the data structure, and who has adaptive and exclusive access to it: the adversary can invoke operations on the data structure and observe their timings (and responses) in order to discern the structure's underlying state, without interference from other queries potentially modifying that state. Despite this power, we require that the data structure resist the adversary's attempts to predict how long its future invocations will take to service. Moreover, so as to rule out implementations that obscure timings by making their operations vastly more expensive, we require that the performance of the operations be competitive with other, timing-predictable implementations of the same abstract data type, even against an adversary bent on decaying their efficiency.

As a first step in this direction, we propose an implementation of a set that supports insertions, deletions, and membership queries, and that meets the requirements outlined above. Our set implementation is derived from skip lists, a popular data structure for implementing sets, but exhibits timing unpredictability unlike regular skip lists (as we will demonstrate). In particular, our implementation introduces novel techniques for modifying skip lists during queries, so as to make them more timing-unpredictable with little additional overhead.

To quantify the timing unpredictability of our proposed set implementation, we develop a methodology by which an adversary, based on the timings he observed for his previous operation invocations, can track a probability distribution on the state of the data structure. We also show how the adversary can use this distribution to infer an invocation that will best refine his ability to predict timings of future invocations, or that will best manipulate the data structure so as to make it maximally inefficient. We have implemented this attack methodology in a tool to which we subject our proposed set implementation.

The results of our evaluation indicate that our proposed set implementation is substantially more timing-unpredictable than a regular skip list. Moreover, we show that our set implementation is efficient, in that it retains its good performance despite the contrary efforts of the adversary, while the adversary achieves considerable decay of a standard skip list's performance. These advantages derive from the adversary's uncertainty as to the shape of the data structure at any point in time, in contrast to a standard skip list, which the adversary can unambiguously reverse-engineer in little time.

To summarize, the contributions of this paper are as follows. We introduce the problem of achieving timing unpredictability in data structures. We propose a novel set implementation that improves timing unpredictability over that achieved by other set implementations at little additional cost. We demonstrate these advantages through a methodology by which an adversary determines requests to best refine his ability to predict timings of future operations or to decay the performance of those operations.

2 Related Work

In this paper we explore the construction of a data structure that alters its shape (and thus its timing characteristics) randomly, even as frequently as on a per-operation basis. This high-level idea is borrowed from approaches to render timing attacks against cryptographic implementations (e.g., [4, 5]) more difficult, by randomizing the cryptographic secrets involved in the computation in each operation. A well-known example is "blinding" an RSA private key operation $m^d \bmod N$ by computing this as $(mr^e)^d r^{-1} \bmod N$ for a random $r \in \mathbb{Z}_N^*$ [4]. This paper is a first step toward applying randomized blinding techniques in data structures, as opposed to particular cryptographic implementations.

Algorithmic denial-of-service attacks, in which an adversary crafts invocations that he can predict will be costly to process, have led to proposals to use data structures less susceptible to such attacks (e.g., [2, 3]). These data structures generally fall into two categories: those that bound worst-case performance and those that attempt to make worst-case inputs unpredictable. The first category consists mainly of self-balancing data structures (e.g., splay trees [6], AVL trees [7]), which make no attempt to limit an adversary's ability to predict operation costs. Thus, while these data structures keep access costs consistently below some desirable asymptotic threshold, the costs are typically easy to predict, allowing these structures to be exploited in other forms of timing attacks. The second category consists of data structures that mitigate algorithmic denial-of-service attacks by limiting an adversary's ability to induce worst-case performance reliably. Typically, this limiting is accomplished using either a randomized insertion algorithm (e.g., randomized binary search trees [8]) or a secret unknown to the adversary (e.g., keyed hash tables [9]). We show in Section 4 that randomized insertion is not sufficient to achieve unpredictability versus an adaptive adversary. A deterministic algorithm based on a fixed secret faces the same difficulty: the adaptive adversary's ability to probe the data structure allows him to uncover its shape and thus its timings, even without knowing the secret.

Skip lists, from which our proposed set implementation is built, have been widely studied, and many variants have been proposed. Most are motivated by performance, to improve access time for certain input sequences or in certain applications (e.g., [10–13]). Others are skip-list variants that can safely be used by concurrent processes or in distributed environments (e.g., [14–16]). Aspects of some of these variants bear similarities to elements of our proposal, but none of them addresses timing predictability or performance under adversarial access.

Also related to our work is *online* algorithm analysis (e.g., [17]), which deals with algorithms that process requests as they arrive ("online" algorithms) and how they perform compared to optimal algorithms that process the same requests all at once ("offline" algorithms). Of particular interest here is the field's analysis of *adaptive* adversaries that select each request with knowledge of the random choices made by the online algorithm so far. Our adversary is weaker, selecting new requests knowing only the *duration* of each previous request. Durations leak information about the algorithm's random choices but may not reveal those choices unambiguously. Our weaker adversary is motivated both by a practical perspective — an adversary can easily measure durations but would rarely be given all random choices made by the algorithm — and also by our hope to explore the extent to which randomization can limit the adversary's knowledge of the data structure's future timing behavior. Assuming the adversary knows all prior random choices would preclude this exploration.

3 Goals

As discussed in Section 1, a common thread in many attacks is the adversary's ability to predict the timing of operations that will result from his activity (and correspondingly to manipulate the data structure to produce desirable timings). These timings can be particularly large, as in an algorithmic denial-of-service attack. Or, it may simply suffice that the timings can be predicted accurately, whether they be large or not, e.g., to minimize the "noise" associated with other activities when cryptanalyzing keys via timing attacks.

As an illustrative example, consider that a server using OpenSSL does approximately ten set lookups (implemented using hash tables) between receiving a ClientHello message and sending its ServerKeyExchange response. Because the ServerKeyExchange message often involves a private key operation — signing the parameters for Diffie-Hellman key exchange — the timing the client observes between messages involves both set lookup operations and the private key operation. As such, having an understanding of the timing of the set lookup operations can enable an adversary to obtain a more fine-grained measurement of the private key operation. As another example, popular interpreted languages such as Perl and Python incorporate associative arrays implemented as sets (specifically using hash tables) as a primary built-in data type, providing an avenue for exploiting timing in a range of applications written in those languages. Perl's hash function has already been shown to be vulnerable to denial-of-service attacks [3], and Python's hash function is intentionally trivial — integers, for example, hash to their lower-order bits.

The goal of our designs in this paper will be to limit an adversary's ability to predict and manipulate the timing of his future operations on a data structure. More precisely, we consider an abstract data type with predefined operations, each of which accepts some number of arguments of known types. Motivated by the examples above, and to make our discussion more concrete, we will use a set data type (Set) as a running example throughout this paper. A data structure S of type Set would typically support the following operations:

- $S.\mathsf{insert}(v)$ adds value v to S if it doesn't already exist, i.e., $S \leftarrow S \cup \{v\}$;
- $S.\mathsf{remove}(v)$ removes v if it is in S, i.e., $S \leftarrow S \setminus \{v\}$;
- $S.\mathsf{lookup}(v)$ returns v if $v \in S$, or $\perp$ otherwise.

We give an adversary adaptive access to S; i.e., the adversary can perform any invocation of his choice, and receives the response to this invocation before choosing his next. Since the adversary can time the duration until receiving the response, we model this by returning not only the return value from the invocation, but also the duration of the invocation (in some appropriate unit of time that we will leave unspecified for now). For example, an adversary's interaction with the set S might look like Figure 1.

	Invocation	*Return value*	*Duration*
1.	$S.\mathsf{insert}(7)$	"ok"	4
2.	$S.\mathsf{insert}(12)$	"ok"	6
3.	$S.\mathsf{lookup}(7)$	7	3
	⋮	⋮	⋮

Fig. 1. Example execution

The notion of timing-unpredictability that we study in this paper comprises two types of requirements, which we describe below.

Invocations must be efficient: Efficient operation is not a requirement unique to timing-unpredictability, obviously, as it has been a primary goal of algorithm design since its inception. We explicitly include it here, however, to emphasize that we cannot sacrifice (too much) efficiency in order to gain unpredictability. Here we measure efficiency in terms of the extent to which the above adaptive adversary can manipulate the data structure to render invocations of his choice as expensive as possible.
Timing of invocations must otherwise be "unpredictable": Intuitively, to be *timing-unpredictable*, we require that the adversary be unable to predict the time that invocations will take. More specifically, after observing the timings associated with operations of his choice, the adversary can generate the probability distribution of possible timings that each next possible invocation could produce. We measure unpredictability by the minimum of the entropies of the timing distributions for all next possible invocations, i.e., $\min_{\mathsf{inv}} \mathsf{H}(\mathsf{dur}(\mathsf{inv}))$ where $\mathsf{dur}(\mathsf{inv})$ is a random variable representing the timing of invocation inv, conditioned on the invocations and their timings that the adversary has observed so far, and $\mathsf{H}()$ denotes entropy. Intuitively, the entropy gives a measure of how uncertain the adversary is of the resulting timing. There are natural extensions of this property, e.g., using the *average* entropy over all invocations, i.e., $\mathrm{avg}_{\mathsf{inv}} \mathsf{H}(\mathsf{dur}(\mathsf{inv}))$. However, because the minimum entropy will always be at most the average entropy, we consider only the former here.

Two observations about the above goals are in order. First, there is a tension between performance and unpredictability, in that the efficiency requirement limits the degree of unpredictability for which we can hope. Notably, a data structure of size n that implements invocations in $O(f(n))$ time for nondecreasing f permits unpredictability (as defined above) of at most $\log_2 O(f(n)) = O(\log_2 f(n))$. One way to balance these two might leave the timing distribution across invocations on the data structure unchanged from that of a timing-predictable structure (to retain efficiency) but make it impossible to predict which invocations would produce which timings (so that timings are unpredictable).

Second, though neither of the above goals explicitly includes hiding the data structure state from the adversary, doing so can be helpful to our goals, and some of our analysis will measure what the adversary can know about that state. One approach to hide this from the adversary would be to insert a random delay prior to each invocation response. However, just as such random delays do not thwart cryptographic timing attacks (these delays can be filtered out statistically and the keys still recovered), they will only delay an adversary from recovering the data structure state. An alternative might be to slow all operations to take the same time, presumably calculated as a function of n. However, this benefits neither efficiency nor timing unpredictability, our primary goals here.

4 Skip Lists

One goal of this paper is to develop a Set implementation that meets the requirements of Section 3. We do so by building from skip lists, a well-known implementation of a Set. We first describe the skip-list structure, and then we discuss its vulnerabilities to timing attacks.

Data structure and algorithm: A skip list is a data structure that can be used to implement the Set abstract data type [18]. A skip list comprises multiple non-empty linked lists, denoted $\mathsf{list}_1, \ldots, \mathsf{list}_m$, where $m \geq 1$ can vary over the life of the skip list. Each linked list consists of *nodes*, each with a *pointer* to its successor in the list; the successor of node nd is denoted nd.nxt. List list_ℓ begins with a *head* node, denoted $\mathsf{head}[\ell]$. Each other node in list_ℓ represents a value that was inserted into the set; the value of each such node nd is nd.val. The nodes in each linked list are sorted in increasing order of their values. The first linked list, list_1, includes (a node for) each value inserted into the set. Each list_ℓ for $1 < \ell \leq m$ contains a subset of the inserted values, and satisfies the following property: if a value is in list_ℓ, then it is also a member of $\mathsf{list}_{\ell-1}$, and the node nd representing v in list_ℓ contains a pointer nd.down to the node representing v in $\mathsf{list}_{\ell-1}$. Similarly, $\mathsf{head}[\ell].\mathsf{down} = \mathsf{head}[\ell-1]$.

To lookup v in a skip list, the search begins at the head of the m-th linked list. It traverses that linked list, returning if it finds v or stopping when it reaches the last node in the list whose value is strictly less than v. In the latter case, if the current list is also list_1, then it returns $\perp$. Otherwise, the search drops to the next lower linked list and continues as before. An example of a lookup in a standard skip list is shown in Figure 2.

To remove a value v from a skip list, we navigate to v by the same method. Once located, we simply remove the nodes representing v from the linked lists. Any empty linked lists are deleted, and m is adjusted accordingly.

When inserting a value into the skip list, we first probabilistically determine its "height" in the skip list, i.e., the largest value $h \geq 1$ such that list_h will contain the new value. We sample the new height from a distribution that yields any h with probability 2^{-h}. Once the height of the new value is so determined, we find the position of the new value in list_h using the same search method as in the lookup and remove operations. Then we simply add the new value to the proper locations in lists list_h, …, list_1, creating new lists (if $h > m$) and adjusting m as necessary. As such, in expectation only 1/2 of the values are represented in list_2, only 1/4 are represented in list_3, and so on. For this reason, a skip list of n values supports lookup, insert and remove operations in $O(\log_2 n)$ time with high probability.

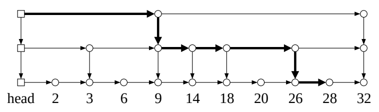


Fig. 2. Search path for lookup(28) in standard skip list

Weaknesses: Despite their randomized nature, skip lists are vulnerable to attacks on both predictability and efficiency. Section 6 details how an adversary can track the distribution of possible skip lists (that is, the distribution of different skip-list configurations that represent the same Set) given access to a skip list only via invocations and their observed durations. Using this technique, even an adversary passively observing random lookup invocations can quickly determine the internal configuration of the skip list. For example, Figure 3 shows the graph of the average entropy in bits (over 100 runs) of the skip-list distribution for such an adversary over the course of 25 observed lookup invocations and their durations on a skip list of size 5.

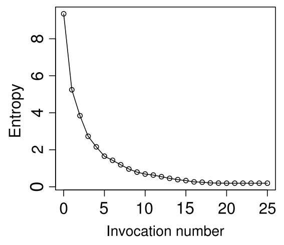


Fig. 3. Average entropy of standard skip-list distributions based on observed lookup durations. Skip list holds 5 values.

This result illustrates that the randomization that takes place during an insert operation is not enough to hide the internal configuration of the skip list from an adversary. Proposals exist for occasionally rearranging the entire internal configuration of a skip list,[1] but as these methods must operate on each value in the skip list, they are generally performed only when there is some other reason for an $O(n)$ operation (e.g., enumerating the entire contents of the skip list). We argue that these methods are insufficient to protect a skip list for two reasons. First, they are designed to repair inefficiently balanced skip lists, doing little to hinder predictability attacks unless they occur very frequently. Second, an adversary can simply choose not to invoke any operations that would result in reconfiguration, and reconfiguration is too expensive to invoke frequently in a proactive manner.

Having sufficiently reduced the entropy of the skip-list distribution, the adversary can trivially predict the timing of future invocations. Moreover, the adversary can bias the skip-list distribution toward inefficient configurations by adaptively crafting invocations using observed duration information. Specifically, an adversary might target values with heights $h > 1$, removing and re-adding them until they are inserted at height $h = 1$. Once the adversary has adjusted all values with height $h > 1$ in this way, the skip list will have been reduced to a linked list with $\Omega(n)$ performance.

5 A Timing-Unpredictable Set

In this section we describe ways to counter the weaknesses identified in Section 4, and then use these to construct a proposed timing-unpredictable Set.

Manipulating the origin: In a standard skip list, every operation begins from head$[m]$. We propose in this section to reduce the ability of the adversary to predict the timing characteristics of future operations by modifying, on a per operation basis, the starting point of a lookup, insert, or remove. To do so, we introduce a search *origin* into the skip list, and this origin will change on a per operation basis.

Intuitively, the search origin can be thought of as a new value that is inserted using an operation similar to insert, except that the height h chosen for it is $h = m$. Then, rather than starting a search for a value (or location to insert a new value) from head$[m]$, the search is begun from this origin value's node in

[1] `http://en.wikipedia.org/wiki/Skip_list#Implementation_Details`

list_m; otherwise the search behaves as normal. In order to enable values smaller than the origin value to be located, however, we make each linked list circular (as shown in Figure 4.)

In practice, it is unnecessary for the origin to be represented using its own nodes, and doing so would incur heavier operation costs than are necessary. Instead, we define the origin to be a sequence $\mathsf{ond}[m], \mathsf{ond}[m-1], \ldots, \mathsf{ond}[1]$ of nodes, each $\mathsf{ond}[\ell]$ being an existing member of list_ℓ. Each origin is constructed relative to a particular "target" value otgt in the skip list. For each $1 \le \ell \le m$, $\mathsf{ond}[\ell]$ is the node in list_ℓ with the largest value less than otgt, or if there is no node in list_ℓ with a value less than otgt, then $\mathsf{ond}[\ell]$ is the node with the largest value in list_ℓ. A search from $\mathsf{ond}[m], \mathsf{ond}[m-1], \ldots, \mathsf{ond}[1]$ starts at $\mathsf{ond}[m]$, and if the search is presently at $\mathsf{ond}[\ell+1]$, it proceeds to $\mathsf{ond}[\ell]$ if stepping to $\mathsf{ond}[\ell+1].\mathsf{nxt}$ would pass the sought value. The detailed algorithm is provided in Appendix A, and examples are given in Figure 5.

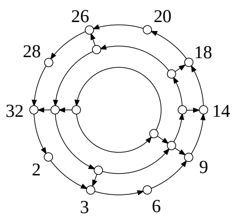
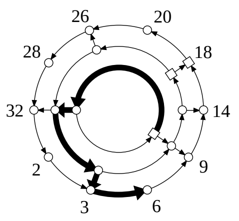
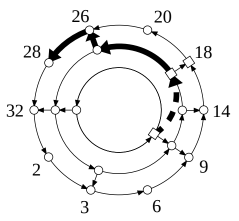


Fig. 4. A skip list with no fixed origin

In order to maximize the adversary's uncertainty as to the state of the skip list, and hence to maximize his uncertainty as to the timing it will exhibit, we choose a value v uniformly at random from the values in the skip list when establishing a new origin (relative to v). In order to select a value uniformly at random, we add to each node nd two additional fields. The first is $\mathsf{nd.skip}$, which records the number of values in the skip list that are "skipped" between nd and $\mathsf{nd.nxt}$. More precisely, if nd is in list_1, then $\mathsf{nd.skip} = 1$, and otherwise $\mathsf{nd.skip} = \sum_{i=0}^{c-1} \mathsf{nd.down}(.\mathsf{nxt})^i.\mathsf{skip}$ where $(.\mathsf{nxt})^i$ denotes i copies of "$.\mathsf{nxt}$" and $c > 0$ is the smallest value satisfying $\mathsf{nd.nxt.down} = \mathsf{nd.down}(.\mathsf{nxt})^c$. The second field is $\mathsf{nd.idx}$, which is used only when nd is a part of the origin. It records the absolute index of nd in the skip list. These fields can be maintained in the skip list across insert and remove operations (and origin changes) with no change in the asymptotic cost of these operations, as shown in Appendix A.



The search path for $\mathsf{lookup}(6)$. The search wraps from high-valued nodes to low-value nodes.



The search path for $\mathsf{lookup}(28)$. The search travels down by origin nodes until a move right has been made.

Fig. 5. Search paths to two different nodes in a circular skip list; squares (□) denote origin nodes placed with respect to $\mathsf{otgt} = 20$

Given these extra fields, establishing an origin relative to a value otgt selected uniformly at random in a skip list with n values is achieved as follows: choose a $j \in [1, n]$ at random, and then use the $\mathsf{nd.skip}$ and $\mathsf{nd.idx}$ values to navigate to the j-th value in the list (to which otgt will be set) and assemble the new origin relative to that value. Again, this can be performed with only an additive cost to the skip-list operation that does not change its asymptotic complexity.

Height adjustment: The second countermeasure to timing predictability that we employ is to "height adjust" a value in the skip list. Recall that when a value is inserted into a standard skip list, we probabilistically determine its "height" in the skip list, i.e., the largest value $h \geq 1$ such that list_h will contain the new value, by sampling from a distribution that yields any h with probability 2^{-h}. When height adjusting a value we simply re-sample from this distribution to obtain a new height for the value, and then modify linked lists to reflect this value's newly chosen height. The effect is equivalent to having removed and then re-inserted the value. However, since this is accomplished with searching to the value only once, and without removing nodes that would be re-inserted, it is far less costly than actually removing and re-inserting the value.

The TUSL skip list: There are many potential ways to combine origin movement and height adjustment to implement skip-list variants that should better resist an adversary divining and manipulating its structure. For our study in Sections 6 and 7, we consider the following variant, to which we refer as TUSL (for "Timing-Unpredictable Skip List"). We designed the TUSL such that its variations from standard skip lists would introduce only small additional costs and also not change the asymptotic complexity of the Set operations.

$\underline{\mathsf{insert}}$ To perform an $\mathsf{insert}(v)$, first select the height h for the new value. Next, search for the location of v starting from the origin. If v is not already in the skip list, insert nodes for v into $\mathsf{list}_1, \ldots, \mathsf{list}_h$. Regardless of whether v was already in the skip list, select a new otgt at random, and move the origin to be relative to it. If v was already in the skip list, adjust otgt to height h.

$\underline{\mathsf{remove}}$ To perform a $\mathsf{remove}(v)$, search for v starting from the origin. If v is found, remove its nodes from the linked lists. Whether or not v was found, select a new otgt at random, and move the origin to be relative to it. Finally, height adjust otgt.

$\underline{\mathsf{lookup}}$ To perform a $\mathsf{lookup}(v)$, search for v starting from the origin. After the return value is determined (v or $\perp$), select a new otgt at random, and move the origin to be relative to it. Finally, height adjust otgt.

Note that each operation selects a height for one value, namely the new otgt or a newly inserted value. These operations are a small constant factor more expensive than those of a standard skip list, but we will show in Section 7 that a TUSL can outperform a standard skip list against an adversary intent on decaying its performance, even when skip lists are small.

6 Predictability Evaluation

In this section we perform an adversarial evaluation of the extent to which our TUSL design in Section 5 achieves unpredictability. We begin by presenting how the adversary can track the distribution on skip lists based on the timing he observes for each of his invocations. We then present results about the entropy of this distribution, and then we build on these results to demonstrate the timing unpredictability of our TUSL construction.

Tracking the skip-list distribution: The timings observed by the adversary and the skip-list algorithm itself (which he knows), induce a probability distribution on the space of skip lists from his perspective. Let $I_i = \langle(\mathsf{inv}_1, \mathsf{dur}(\mathsf{inv}_1))$, …, $(\mathsf{inv}_i, \mathsf{dur}(\mathsf{inv}_i))\rangle$ denote a sequence of invocations and their durations. Each $\mathsf{inv}_{i'}$ is applied to the skip list $S_{i'-1}$ (i.e., the skip list resulting from invocations $\mathsf{inv}_1 \ldots \mathsf{inv}_{i'-1}$) in sequence, taking time $\mathsf{dur}(\mathsf{inv}_{i'})$ (a random variable) and yielding $S_{i'}$ (also a random variable). When we use $I_i = \langle(\mathsf{inv}_1, d_1), \ldots, (\mathsf{inv}_i, d_i)\rangle$ to denote an event, the event quantifies the durations of the (fixed) invocations $\mathsf{inv}_1, \ldots, \mathsf{inv}_i$; i.e., $\mathsf{Pr}[I_i]$ is the probability that fixed invocations $\mathsf{inv}_1, \ldots, \mathsf{inv}_i$ satisfy $\mathsf{dur}(\mathsf{inv}_1) = d_1$, …, $\mathsf{dur}(\mathsf{inv}_i) = d_i$.

To explain how the adversary can track the distribution on TUSLs, i.e., how he can compute $\mathsf{Pr}[S_i = s \mid I_i]$, we introduce the following additional notation. Let O_i denote the value of otgt at the end of (i.e., chosen in) inv_i. Let H_i denote the value of the height chosen in inv_i; this height is chosen for the value O_i or for the new value if inv_i inserted one. Let n_i denote the number of values in S_i, and let $v_1, \ldots, v_{n_i}$ denote an enumeration of the values in S_i. Then, the adversary can compute $\mathsf{Pr}[S_{i+1} = s' \mid I_{i+1}]$ inductively as:

$$\frac{\displaystyle\sum_{s}\sum_{h=1}^{\infty}\sum_{j=1}^{n_{i+1}}\left(\begin{array}{l}2^{-h}\cdot\mathsf{Pr}[S_i = s \mid I_i]\ \cdot\\ \mathsf{Pr}[S_{i+1}=s' \wedge \mathsf{dur}(\mathsf{inv}_{i+1})=d_{i+1} \mid S_i=s \wedge H_{i+1}=h \wedge O_{i+1}=v_j]\end{array}\right)}{\displaystyle\sum_{s}\sum_{h=1}^{\infty}\sum_{j=1}^{n_{i+1}}\left(\begin{array}{l}2^{-h}\cdot\mathsf{Pr}[S_i = s \mid I_i]\ \cdot\\ \mathsf{Pr}[\mathsf{dur}(\mathsf{inv}_{i+1}) = d_{i+1} \mid S_i = s \wedge H_{i+1} = h \wedge O_{i+1} = v_j]\end{array}\right)} \tag{1}$$

We derived this equation as an application of Bayes' theorem, but we omit its lengthy derivation here due to space limitations. Note that $\mathsf{Pr}[S_{i+1} = s' \wedge \mathsf{dur}(\mathsf{inv}_{i+1}) = d_{i+1} \mid S_i = s \wedge H_{i+1} = h \wedge O_{i+1} = v_j]$ in the numerator and $\mathsf{Pr}[\mathsf{dur}(\mathsf{inv}_{i+1}) = d_{i+1} \mid S_i = s \wedge H_{i+1} = h \wedge O_{i+1} = v_j]$ in the denominator are either identically 0 or identically 1, in that the conditions and the invocation unambiguously specify whether $S_{i+1} = s'$ and $\mathsf{dur}(\mathsf{inv}_{i+1}) = d_{i+1}$.

In addition to computing a distribution on skip lists on the basis of timings actually observed from invocations on S, the adversary can also compute posterior distributions conditioned on a hypothetical invocation and the distribution of timings for that invocation that the prior distribution on skip lists dictates. In this way, the adversary can compute not only a distribution on the current state of the skip list, but also can compute the probability that a particular invocation will yield a particular timing and, thus, the posterior distribution on the skip list that would result.

Entropy of the skip-list distribution: To provide insight into the results we report below, we first present tests in which the adversary, when selecting inv_{i+1}, chooses the invocation that minimizes $\mathsf{H}(S_{i+1} \mid I_i)$, i.e., that minimizes the entropy of the skip-list distribution that results from the chosen invocation. We measure $\mathsf{H}(S_{i+1} \mid I_{i+1})$, i.e., the extent to which the adversary succeeds in minimizing that entropy. Although minimizing the entropy of the skip-list distribution is not a stated goal in Section 3, this measure provides insight into the uncertainty that the adversary faces in trying to predict timings for future invocations or to manipulate the skip list to slow its performance.

In each test, the adversary is launched with an empty skip list and a target size N. Each run begins by the adversary performing N random insert invocations, to bring the skip list to its initial size. The adversary monitors the time that each of these invocations takes, as well as all subsequent invocations. Once the skip list contains N values, the adversary performs lookup invocations only, chosen to minimize $\mathsf{H}(S_{i+1} \mid I_i)$ in each step $i+1$. We disallow remove invocations in these tests, in particular, so that the adversary cannot decrease $\mathsf{H}(S_i \mid I_i)$ simply by removing elements. After performing the lookup invocation and measuring its duration, the adversary updates his skip-list distribution using (1), and continues with searching for his next invocation, etc. To limit the number of possible skip lists in our tests, we remove at each step (after the initial N insert invocations) skip lists with probability less than $\epsilon = 4^{-n}$, where n is the current skip-list size. ($n = N$ always in the tests of this section.)

In our analysis, the "time" that the adversary measures for an invocation is a count of skip-list node visits plus, in the case of an insert operation (or a remove, though again, none of these were performed in the tests in this section), the changes to linked lists in the skip list. This information is not clouded by other factors that could influence time measurements and so discloses more precise information than the adversary might expect in practice.

The results of our tests are shown in Figure 6 for $N \in \{4, 5, 6, 7\}$. As these figures show, the average entropy of a TUSL grows linearly in N for these values, even when the adversary chooses the *best* next invocation to minimize that entropy. This observation provides insight into the results that will follow.

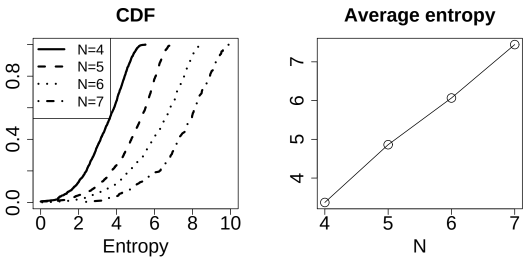


Fig. 6. Distribution of $\mathsf{H}(S_i \mid I_i)$

We were unable to extend past $N = 7$ in our tests due to the computational difficulty of doing so. To get a sense of the immensity of these tests — and the task the adversary faces, as well — consider the following rough calculation for a distribution on skip lists of size $N = 6$: The adversary uses (1) to update the skip-list distribution (from S_i to S_{i+1}) to account for a single observed duration. The summations in the equation occur over each possible TUSL s (typically about 160), all sufficiently plausible heights (we consider only 7 for this example), and all possible positions for a new otgt (there are N of these). Thus, the inner term of each summation must be evaluated approximately $160 * 7 * 6 = 6,720$ times. Also, this calculation must be done once for each s', meaning that to transform a distribution for S_i into one for S_{i+1} for a single invocation/duration pair, the adversary must do $160 * 6,720 \approx 1$ million calculations. Now consider that the adversary's search of next invocations includes N possible lookup invocations, each with about 30 possible durations. So, even choosing the next invocation to perform requires examining $6*30 = 180$ possible distributions, and the adversary must do $180 * 1,075,200 \approx 200$ million evaluations of the inner term of (1) to generate *a single sample* for the distribution for $N = 6$ in Figure 6. For the $N = 7$ plot, the cost jumps to ≈ 750 million evaluations per sample. This computational cost has limited our ability to scale our tests beyond $N = 7$ at present.

Timing unpredictability: We now move on to tests in which the adversary attacks timing unpredictability. These tests were performed with the same methodology as those above, except that the adversary chooses as his next invocation $\arg\min_{\mathsf{inv}_{i+1}} \mathsf{H}(\mathsf{dur}(\mathsf{inv}_{i+1}) \mid I_i)$. We record $\mathsf{H}(\mathsf{dur}(\mathsf{inv}_{i+1}) \mid I_i)$ for that invocation inv_{i+1} at each step, as evidence of the extent to which an adversary can minimize the timing predictability of the data structure.

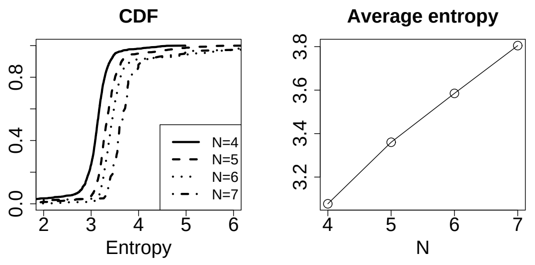


Fig. 7. Distribution of $\min_{\mathsf{inv}_{i+1}} \mathsf{H}(\mathsf{dur}(\mathsf{inv}_{i+1}) \mid I_i)$

Figure 7 shows the results of these tests. The plots show that the timing entropy is less than the entropy of the skip-list distribution, as can be seen by comparing Figures 6 and 7. This occurs because many different skip-list configurations can give rise to the same timing for certain invocations, and so not all of the uncertainty of the skip-list configuration carries over to uncertainty for timing behavior. Figure 7 suggests that the timing entropy grows roughly linearly for the range of N that we have been able to explore. (These tests are limited by the same computational challenges described earlier.) However, because for an adversary who does not try to slow the skip-list invocations (or is unable to do so, see Section 7), the skip-list implements lookup invocations in $O(\log_2 N)$ time with high probability, the timing entropy is limited to $O(\log_2 \log_2 N)$ as N grows, as discussed in Section 3.

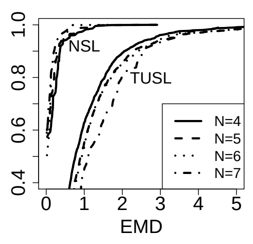


Fig. 8. CDF of EMD between adversary's and actual timing distributions for inv_{i+1}. NSL = normal skip list.

While $\min_{\mathsf{inv}_{i+1}} \mathsf{H}(\mathsf{dur}(\mathsf{inv}_{i+1}) \mid I_i)$ indicates the timing unpredictability of the data structure, it nevertheless provides little insight into how erroneous the adversary's view of the timing might be. For example, if the adversary assigns equal likelihood to two timings for inv_{i+1}, we might consider him to be better off if these timings are both close to the correct answer than if one is wildly incorrect; $\mathsf{H}(\mathsf{dur}(\mathsf{inv}_{i+1}) \mid I_i)$ does not distinguish between these cases. To further clarify, in Figure 8 we plot the CDF of the *earth mover's distance* (EMD) [19, 20] between (i) the adversary's distribution for $\mathsf{dur}(\mathsf{inv}_{i+1})$ conditioned on I_i and (ii) the distribution $\mathsf{dur}(\mathsf{inv}_{i+1})$ for that invocation on the *actual* skip list that the adversary is attacking. Intuitively, if each distribution is a way of piling one unit of dirt, EMD measures the cost (the amount of dirt moved times the distance it is moved) of turning one distribution into the other. This plot shows that the uncertainty the adversary faces is not solely due to the randomized implementation of inv_{i+1} but rather is compounded by the entropy of the skip-list distribution shown in Figure 6. That is, if the adversary's skip-list distribution had no entropy (i.e., if the adversary knew exactly the configuration of the skip list), his distribution would match the real distribution, and the EMD would be zero. As can be seen in Figure 8, this is very nearly the case for normal skip lists.

7 Efficiency Evaluation

We now evaluate how TUSLs fare in terms of performance against the adaptive adversary of Section 3. Our evaluation is like that of Section 6, with a few important differences. First, to maximize the invocation times (versus simply reducing entropy for skip lists of a fixed size or their timing behaviors), the adversary must be allowed to remove and insert elements. For example, an adversary might prefer to remove an element that he discerns to have a large height in the skip list, in an effort to make all elements have the same height (which yields worst-case performance for the skip list). For this reason, in these tests the adversary also examines remove and insert operations at each step, though we restrict the adversary to maintaining the size of the skip list in the range $N \pm 2$. This restriction prevents the adversary from "attacking" efficiency, for example, by simply always inserting more values. Second, to discern that a remove–insert pair, for example, might decay the performance of the skip list, it is necessary to permit the adversary to look ahead multiple moves to find a sequence that best accomplishes his goals. So, to enable these tests we implement a search for sequences of invocations that yield a heuristically optimal attack for the adversary (albeit while further compounding the cost of computing the attack).

Searching for a nearly optimal attack: Suppose that $I_i = \langle(\mathsf{inv}_1, d_1), \ldots, (\mathsf{inv}_i, d_i)\rangle$ is the sequence of invocations that the adversary performed and the durations that resulted from them. As shown in (1), the adversary can thus compute $\mathsf{Pr}\left[S_i = s \mid I_i\right]$. The adversary now wishes to predict the next invocation inv_{i+1} that will lead toward a skip-list configuration in which some operations are very expensive, thus violating our efficiency goals. To do so, he employs a function score that, when applied to a sequence I_{i+k} that extends I_i, produces a value that indicates the benefit or detriment to the adversary's goal of reducing performance. We will describe such a score function below.

The primary component of the adversary's attack is calculating, for a *fixed* sequence of invocations $\mathsf{inv}_{i+1}, \ldots, \mathsf{inv}_{i+k}$, the expected outcome:

$$\mathbb{E}_{\mathsf{inv}_{i+1},\ldots,\mathsf{inv}_{i+k}}\left[\mathsf{score}(I_{i+k}) \mid I_i\right] = \sum_g g \cdot \mathsf{Pr}\left[\mathsf{score}(I_{i+k}) = g \mid I_i\right] \qquad (2)$$

In (2), it is understood that I_{i+k} extends I_i with invocations $\mathsf{inv}_{i+1}, \ldots, \mathsf{inv}_{i+k}$. It is, however, treated as a random variable here, taking on durations for the invocations $\mathsf{inv}_{i+1}, \ldots, \mathsf{inv}_{i+k}$.

When choosing inv_{i+1}, ..., inv_{i+k} to compute (2), the adversary faces an apparently difficult problem in that there are infinitely many invocations that are *possible* for each $\mathsf{inv}_{i+k'}$. Notably, the adversary can insert any value into the skip list. However, the adversary need only consider inserting a value after each value already in the skip list — all insertions between the same two existing values are equivalent from a timing point of view — yielding $n_{i+k'-1}$ possible insert operations for a skip list already containing $n_{i+k'-1}$ values (i.e., where $n_{i+k'-1}$ is the size of $S_{i+k'-1}$). That is, for each $\mathsf{inv}_{i+k'}$, $1 \leq k' \leq k$, the adversary need only consider $n_{i+k'-1}$ remove invocations, $n_{i+k'-1}$ insert invocations, and $n_{i+k'-1}$ lookup invocations, i.e., $3n_{i+k'-1}$ in total.

Heuristics: There are two remaining choices that an adversary must make to search for his next invocation to perform: (i) He must decide for which invocation sequences inv_{i+1}, ..., inv_{i+k} to compute Equation (2), and in particular how many such invocations to consider. (ii) He must choose a score function to guide his search. We adopt heuristic solutions (described below) to (i) and (ii), and as such, our search yields only a heuristically optimal choice.

To address (i), we define a function $\beta : \mathbb{N} \rightarrow (0, 1)$ such that if $\mathsf{Pr}\left[\left(\bigwedge_{k'=1}^{k} \mathsf{dur}(\mathsf{inv}_{i+k'}) = d_{i+k'}\right) \;\middle|\; I_i\right] \leq \beta(k)$ for values $d_{i+1} \ldots d_{i+k}$, then this probability is rounded down to zero. Then, only invocation sequences inv_{i+1}, ..., inv_{i+k} for which (2) is nonzero (per this coarsening) need be considered. In particular, k is not the same across sequences, but rather can be different per sequence. The intuitive justification for such a use of β is that durations for invocation sequences inv_{i+1}, ..., inv_{i+k} that are so improbable are not interesting to the adversary. In our tests below, β is determined empirically to strike a balance between exploring as many invocation sequences inv_{i+1}, ..., inv_{i+k} as possible and limiting search time. Moreover, β was set differently for TUSL adversaries and adversaries attacking a standard skip list to allow a TUSL adversary substantially more time to search for an effective next invocation. In fact, the average time allotted to the adversary to search for his next invocation was more than *three orders of magnitude* larger for the TUSL adversary, per value of N. As such, the results reported below that demonstrate advantages over basic skip lists are very conservative in this regard.

To address (ii), the adversary scores I_{i+k} on the basis of the expected duration it induces for the most expensive subsequent invocation, i.e., $\mathsf{score}(I_{i+k}) = \max_{\mathsf{inv}_{i+k+1}} \mathbb{E}\left[\mathsf{dur}(\mathsf{inv}_{i+k+1}) \mid I_{i+k}\right]$. When his search concludes, he chooses the next invocation inv_{i+1} to actually perform to be the most promising next invocation, specifically $\arg\max_{\mathsf{inv}_{i+1}} \sum_{\mathsf{inv}_{i+2},\ldots,\mathsf{inv}_{i+k}} \mathbb{E}_{\mathsf{inv}_{i+1},\ldots,\mathsf{inv}_{i+k}}\left[\mathsf{score}(I_{i+k}) \mid I_i\right]$, where the sum is taken over maximal sequences for which (2) was computed.

Results: After observing the i-th invocation duration, suppose the adversary outputs $\arg\max_{\mathsf{inv}_{i+1}} \mathbb{E}\left[\mathsf{dur}(\mathsf{inv}_{i+1}) \mid I_i\right]$, i.e., the invocation the adversary believes to be the most expensive. Figure 9 plots $\mathbb{E}\left[\mathsf{dur}(\mathsf{inv})\right]$ for this invocation inv, for the current state of the *actual* skip list he is attacking, averaged over all runs, as a measure of performance. (○ denotes a standard skip list, and + denotes a TUSL.) Figure 9 also shows the average performance of *randomly* selected invocations (where × and ◇ denote standard skip lists and TUSLs, respectively). Together these curves show that the adversary can cause his chosen invocations for a standard skip list to diverge in cost from random invocations. In contrast, the adversary is unsuccessful in causing this divergence with TUSLs, despite expending three orders of magnitude more effort. A consequence is that the adversary can quickly decay a standard skip list, even of size as small as 7 ± 2, to performance that is comparable to or worse than that to which the adversary can decay a TUSL, which appears to be little to none. As N grows, we expect these trends to continue, with the adversary maintaining average-case ($O(\log_2 N)$) performance against TUSLs and worst-case performance ($O(N)$) against standard skip lists, such that the TUSL should soon easily outperform a standard skip list during an attack.

8 Conclusion

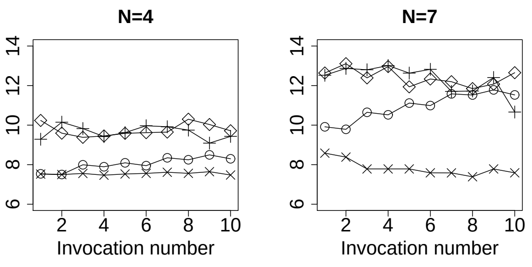


Fig. 9. Average expected invocation duration after the first N inserts. ○: standard skip list; ×: standard skip list, random invocations; +: TUSL; ◇: TUSL, random invocations

This paper is, to our knowledge, the first exploration of constructing data structures that will make it difficult for an adversary with adaptive access to the structure to predict the duration of future invocations or to manipulate the data structure to decay its efficiency. We presented a design for a Set abstract data type based on skip lists but enhanced to permit both searching for a value from a random origin and adjusting the height of a value's nodes per operation. We presented an instance of this design, called TUSL, which we showed offers benefits to both timing-unpredictability and efficiency against adaptive adversaries. To do so, we developed a framework that permits an adversary to track a distribution on skip lists implied by the invocation durations he has observed so far and to search for invocations that heuristically maximize his effectiveness in attacking efficiency or unpredictability.

As far as we are aware, this paper opens up a new research direction that could help to counteract a range of timing-related attacks, both known (e.g., [1–5]) and as-yet-unknown. Numerous areas remain unexplored, such as more formal foundations for the goal of timing unpredictability, and other designs for timing-unpredictable data structures.

Acknowledgements: This work was funded in part by NSF grant CNS-0756998. We are grateful to the security group at UNC for suggestions for improving this work, and to the anonymous reviewers for their comments.

References

1. McIlroy, M.D.: A killer adversary for quicksort. Software – Practice and Experience **29** (April 1999) 341–344
2. Fisk, M., Varghese, G.: Fast content-based packet handling for intrusion detection. Technical Report CS2001-0670, University of California at San Diego (May 2001)
3. Crosby, S.A., Wallach, D.S.: Denial of service via algorithmic complexity attacks. In: Proceedings of the 12th USENIX Security Symposium. (August 2003)
4. Kocher, P.C.: Timing attacks on implementations of Diffie-Hellman, RSA, DSS, and other systems. In: Advances in Cryptology – CRYPTO '96. (August 1996)
5. Brumley, D., Boneh, D.: Remote timing attacks are practical. Computer Networks: The International Journal of Computer and Telecommunications Networking **48**(5) (August 2005) 701–716
6. Sleator, D.D., Tarjan, R.E.: Self-adjusting binary search trees. J. ACM **32**(3) (1985) 652–686
7. Adelson-Velskii, G., Landis, E.M.: An algorithm for the organization of information. Proceedings of the USSR Academy of Sciences **146** (1962) 263–266 (Russian) English translation by M. J. Ricci in *Soviet Math. Doklady* 3:1259–1263, 1962.
8. Seidel, R., Informatik, F., Aragon, C.R.: Randomized search trees. In: Algorithmica. (1989) 540–545
9. Carter, J.L., Wegman, M.N.: Universal classes of hash functions (extended abstract). In: STOC '77: Proceedings of the ninth annual ACM symposium on Theory of computing, New York, NY, USA, ACM (1977) 106–112
10. Bagchi, A., Buchsbaum, A.L., Goodrich, M.T.: Biased skip lists. Algorithmica **42** (2005) 2005
11. Cho, S., Sahni, S.: Biased leftist trees and modified skip lists. Technical Report 96-002, University of Florida (1996)
12. Ergun, F., Ahinalp, S.C.S., Sinha, R.K.: Biased skip lists for highly skewed access patterns. In: In Proceedings of the 3rd Workshop on Algorithm Engineering and Experiments, Springer (2001) 216–29
13. Pugh, W.: A skip list cookbook. Technical Report UMIACS-TR-89-72.1, University of Maryland (1990)
14. Aspnes, J.: Skip graphs. In: Proceedings of the fourteenth annual ACM-SIAM symposium on Discrete algorithms. (2003) 384–393
15. Messeguer, X.: Skip trees, an alternative data structure to skip lists in a concurrent approach. Informatique Théorique et Applications **31**(3) (1997) 251–269
16. Pugh, W.: Concurrent maintenance of skip lists. Technical Report CS-TR-2222.1, University of Maryland (1989)
17. Borodin, A., El-Yaniv, R.: Online Computation and Competitive Analysis. Cambridge University Press (1998)
18. Pugh, W.: Skip lists: a probabilistic alternative to balanced trees. Communications of the ACM **33**(6) (June 1990) 668–676
19. Mallows, C.L.: A note on asymptotic joint normality. Annals of Mathematical Statistics **43**(2) (1972) 508–515
20. Elizaveta, L., Bickel, P.: The earth mover's distance is the Mallows distance: Some insights from statistics. In: Proceedings of the 8th International Conference on Computer Vision. (2001) 251–256

A Implementing a moving origin

In Section 5, we introduced a primitive that we use in our TUSL implementation, namely manipulating the search origin per operation. We first detail the lookup operation that performs a search using such an origin. Then, we detail the maintenance of the nd.skip and nd.idx fields, which can be used to locate a value in the skip list uniformly at random, relative to which the new origin will be placed. In this section, we show each of these can be achieved during skip-list operations without an asymptotic increase in running time.

A.1 Searching from the origin

```
lookup($v$)
100: $\ell \leftarrow m$;
101: nd $\leftarrow$ ond[$m$];
102: while true do
103:     if (nd.nxt.val $<$ $v$ $<$ otgt $\wedge$ nd.nxt.val $<$ nd.val)
          $\vee$ (nd.nxt.val $<$ $v$ $<$ otgt $\wedge$ nd.nxt.val $=$ nd.val
                                              $\wedge$ departed $=$ $false$)
          $\vee$ (nd.val $<$ nd.nxt.val $<$ $v$ $\wedge$ nd $\neq$ ond[$\ell$])
          $\vee$ (nd.val $<$ nd.nxt.val $<$ $v$ $\wedge$ departed $=$ $false$) then
104:         nd $\leftarrow$ nd.nxt;
105:         departed $\leftarrow$ $true$;
106:     else if nd.nxt.val $=$ $v$ then
107:         return $v$;
108:     else if $\ell > 1$ then
109:         $\ell \leftarrow \ell - 1$;
110:         if departed $=$ $false$ then
111:             nd $\leftarrow$ ond[$\ell$];
112:         else
113:             nd $\leftarrow$ nd.down;
114:         end if
115:     else
116:         return $\perp$;
117:     end if
118: end while
```

Fig. 10. Pseudocode for lookup(v) using origin ond[m], …, ond[1], chosen with respect to otgt

Conducting a search for a value from a randomly selected origin proceeds much like a search in a standard skip list; we provide pseudocode in Figure 10. There are two notable changes in the algorithm from that of a standard skip list. First, the lookup algorithm changes to account for wrapping to the beginning of a linked list; this is detected with the condition nd.nxt.val $<$ nd.val in the first disjunct of line 103. In this case, the algorithm wraps to the beginning of the list (line 104) only if v is not the beginning (nd.nxt.val $< v$) and if v occurs before the value with respect to which the origin was previously placed ($v <$ otgt). Another case that causes nd to advance on the linked list is the second disjunct in line 103, in which nd is the only node in this linked list (nd.nxt.val $=$ nd.val) and no moves along any linked list (line 104) have yet occurred (see also line 105). The third and fourth disjuncts in line 103 cover cases in which advancing on this linked list does not wrap (nd.val $<$ nd.nxt.val) and will not exceed the target value (nd.nxt.val $< v$).

The second notable change from this operation for a standard skip list is that when descending to a lower linked list (line 109–114), descent follows origin nodes (line 111) until the origin is departed in the search (departed $=$ $true$). Examples of this search algorithm are given in Figure 5. This search is of the same time complexity as the original, $O(log_2 n)$.

A.2 The nd.skip field

Each node nd in the skip list contains a field nd.skip which stores the number of values which would be “skipped” by following the link nd.nxt. Intuitively, nd.skip is the number of values in the skip list greater than or equal to nd.val but less than nd.nxt.val, though this count can also wrap around the edge of the skip list. Because the value of nd.skip depends only on the membership of each list_ℓ and not on the origin nodes, it suffices to show that we can maintain it across insert operations, remove operations, and height adjustments.

```
insert($v$)
200: $\ell \leftarrow m$;
201: nd $\leftarrow$ ond[$m$];
202: $h \leftarrow$ ChooseNewHeight();
203: Array new_nd[$h$] $\leftarrow$ Init($v$,$h$);
204: Array anode[$m$];
205: Array anodesum[$m$] $\leftarrow [0, 0, \ldots, 0, 1]$;
206: while $\ell \geq 1$ do
207:   if (nd.nxt.val $< v <$ otgt $\wedge$ nd.nxt.val $<$ nd.val)
        $\vee$ (nd.nxt.val $< v <$ otgt $\wedge$ nd.nxt.val $=$ nd.val
                                        $\wedge$ departed $= false$)
        $\vee$ (nd.val $<$ nd.nxt.val $< v \wedge$ nd $\neq$ ond[$\ell$])
        $\vee$ (nd.val $<$ nd.nxt.val $< v \wedge$ departed $= false$) then
208:     if $\ell \neq m$ then
209:       anodesum[$\ell+1$] $\leftarrow$ anodesum[$\ell+1$] + nd.skip
210:     end if
211:     nd $\leftarrow$ nd.nxt;
212:     departed $\leftarrow true$;
213:   else
214:     $\ell \leftarrow \ell - 1$;
215:     /* if $\ell < h$, create new_nd[$\ell+1$] as nd.nxt */
216:     if departed $= false$ then
217:       if $\ell > 0$ then
218:         nd $\leftarrow$ ond[$\ell$];
219:       end if
220:       anode[$\ell+1$] $\leftarrow$ ond[$\ell+1$];
221:     else
222:       anode[$\ell+1$] $\leftarrow$ nd;
223:       nd $\leftarrow$ nd.down;
224:     end if
225:   end if
226: end while
227: $\ell \leftarrow 1$;
228: while $\ell < m$ do
229:   anodesum[$\ell+1$] $\leftarrow$ anodesum[$\ell+1$] + anodesum[$\ell$]
230:   $\ell \leftarrow \ell + 1$
231: end while
232: while $\ell > 1$ do
233:   if $\ell > h$ then
234:     anode[$\ell$].skip $\leftarrow$ anode[$\ell$].skip + 1
235:   else
236:     new_nd[$\ell$].skip $\leftarrow$ anode[$\ell$].skip $-$ anodesum[$\ell$] + 1
237:     anode[$\ell$].skip $\leftarrow$ anodesum[$\ell$]
238:   end if
239:   $\ell \leftarrow \ell - 1$
240: end while
```

Fig. 11. Partial pseudocode for insert(v).

Figure 11 gives pseudocode for the operation insert(v), where the v is inserted with height h. The array new_nd[h] of new nodes is initialized in line 203 by Init(), which creates h new nodes new_nd[h] $\ldots$ new_nd[1] where new_nd[ℓ].val $= v$ and new_nd[ℓ].down $=$ new_nd[$\ell - 1$] for $1 < \ell \leq h$. For the purposes of clarity, the algorithm as shown assumes that v is not already in the TUSL, and we condense the work of correctly setting new_nd[ℓ].nxt into a comment in line 215. In addition, we omit those parts of the algorithm that move the origin and those that handle the case when $h > m$, as these are irrelevant to the calculations for the nd.skip fields.

The algorithm is similar to the one in Figure 10 but has a few key differences. First, it allocates two arrays of size m called anode[] and anodesum[]. The first is an array of nodes which were the last nodes traversed in each row on the path to v. That is, anode[ℓ] is the node that would be assigned to ond[ℓ] if otgt were v. The second array accumulates nd.skip values along each row ℓ on the path to v, but only for nodes which are not anode[ℓ]. If $\ell > h$, the new value for anode[ℓ].skip will be simply be one more than the old value (line 234). But if $\ell \leq h$, the new value for $\mathsf{anode}[\ell].\mathsf{skip}$ is $\mathsf{anode}[\ell].\mathsf{skip} = \sum_{i=1}^{\ell} \mathsf{anodesum}[i]$.

The while loop beginning on line 228 computes this sum for all rows ℓ, storing the result in $\mathsf{anodesum}[\ell]$. Finally, the while loop beginning at line 232 updates $\mathsf{anode}[\ell].\mathsf{skip}$ appropriately. Each of these loops runs only m iterations, so they do not increase the asymptotic running time.

The algorithm for $\mathsf{remove}(v)$ is simpler: we search for the value as in Figure 10, except that we proceed until reaching list_1 (as in Figure 11). If the value v is not in the current row ($h < \ell$), then simply decrement $\mathsf{anode}[\ell].\mathsf{skip}$ by 1. Otherwise, each time we find a node such that $\mathsf{nd.nxt.val} = v$, we know that the node $\mathsf{nd.nxt}$ is set for removal. Therefore, the node nd will "inherit" all the nodes that used to be "skipped" by traversing $\mathsf{nd.nxt}$. Adding these two values (and subtracting one for the node to be removed), we see that

$$\mathsf{nd.skip} = \mathsf{nd.skip} + \mathsf{nd.nxt.skip} - 1$$

Finally, when doing height adjustments for a particular value, we can use elements of the insert and remove versions of the algorithm. That is, when a value is adjusted to have greater height, the algorithm behaves like a partial insert. When a value is adjusted to have a lesser height, the algorithm behaves like a partial remove.

```
$\ell \leftarrow m$
$\mathsf{nd} \leftarrow \mathsf{ond}[m]$
$c \leftarrow \mathsf{ond}[m].\mathsf{idx}$
while $\ell > 1$ do
    if $((c + \mathsf{nd.skip}) \bmod n < j < \mathsf{otgtindex}$
                                  $\wedge\ (c + \mathsf{nd.skip}) \bmod n < c)$
       $\vee\ ((c + \mathsf{nd.skip}) \bmod n < j < \mathsf{otgtindex}$
             $\wedge\ (c + \mathsf{nd.skip}) \bmod n = c \wedge\ \mathsf{departed} = \mathit{false})$
       $\vee\ (c < (c + \mathsf{nd.skip}) \bmod n < j \wedge \mathsf{nd} \neq \mathsf{ond}[\ell])$
       $\vee\ (c < (c + \mathsf{nd.skip}) \bmod n < j$
                                  $\wedge\ \mathsf{departed} = \mathit{false})$ then
        $\mathsf{nd} \leftarrow \mathsf{nd.nxt}$
        $\mathsf{departed} \leftarrow \mathit{true}$
        $c \leftarrow (c + \mathsf{nd.skip}) \bmod n$
    else
        $\ell \leftarrow \ell - 1$
        if $\mathsf{departed} = \mathit{false}$ then
            $\mathsf{nd} \leftarrow \mathsf{ond}[\ell]$
            $c \leftarrow \mathsf{ond}[\ell].\mathsf{idx}$
        else
            $\mathsf{ond}[\ell + 1] \leftarrow \mathsf{nd}$
            $\mathsf{ond}[\ell + 1].\mathsf{idx} \leftarrow c$
            $\mathsf{nd} \leftarrow \mathsf{nd.down}$
        end if
    end if
end while
$\mathsf{otgtindex} \leftarrow j$
```

Fig. 12. Pseudocode for moving origin to j.

A.3 The nd.idx field

In order to locate a specific index in the skip list (i.e., the ith element, regardless of its value), we augment each node in the skip list with a field called $\mathsf{nd.idx}$. This field stores the absolute index of the node nd. We will only reference this value for origin nodes. We will also define $\mathsf{otgtindex}$ as the index relative to which we place the origin.

Locating a new origin index (called j here to avoid confusion with the previous $\mathsf{otgtindex}$) proceeds very much like the algorithm in Figure 10, except that instead of making decisions based on values at certain nodes, we decide based on indices at those nodes. As such, the new algorithm (shown in Figure 12) uses the current index c, rather than the current value $\mathsf{nd.val}$. When considering a move along nd.nxt, we examine the index of nd.nxt — which is $(c + \mathsf{nd.skip}) \bmod n$ — rather than nd.nxt.val. We use otgtindex instead of otgt.

Finally, maintaining these nd.idx fields is relatively simple. Because these fields track a value's absolute index in the skip list, they change only as a result of an insert or remove operation and are unaffected by height changes. The algorithm is as follows: when a new value v is added to the skip list, all origin nodes $\mathsf{ond}[\ell]$ with values $\mathsf{ond}[\ell].\mathsf{val} > v$ increment their indices by one. If $\mathsf{otgt} > v$, otgtindex is also incremented. When a value v is removed from the skip list, all origin nodes $\mathsf{ond}[\ell]$ with values $\mathsf{ond}[\ell].\mathsf{val} > v$ decrement their indices by one. If $\mathsf{otgt} > v$, otgtindex is also decremented. These adjustments represent another loop of order m, which again does not increase the asymptotic running time of the algorithm.