The SkipTrie: Low-Depth Concurrent Search without Rebalancing

Rotem Oshman
University of Toronto
rotem@cs.toronto.edu

Nir Shavit*
MIT
shanir@csail.mit.edu

ABSTRACT

To date, all concurrent search structures that can support predecessor queries have had depth logarithmic in m, the number of elements. This paper introduces the *SkipTrie*, a new concurrent search structure supporting predecessor queries in amortized expected $O(\log\log u + c)$ steps, insertions and deletions in $O(c \log\log u)$, and using $O(m)$ space, where u is the size of the key space and c is the contention during the recent past. The SkipTrie is a probabilistically-balanced version of a y-fast trie consisting of a very shallow skiplist from which randomly chosen elements are inserted into a hash-table based x-fast trie. By inserting keys into the x-fast-trie probabilistically, we eliminate the need for rebalancing, and can provide a lock-free linearizable implementation. To the best of our knowledge, our proof of the amortized expected performance of the SkipTrie is the first such proof for a tree-based data structure.

Categories and Subject Descriptors

E.1 [**Data**]: Data Structures—*distributed data structures*

Keywords

concurrent data structures, predecessor queries, amortized analysis

1. INTRODUCTION

In recent years multicore software research has focused on delivering improved search performance through the development of highly concurrent search structures [11, 14, 7, 16, 5, 4]. Although efficient hash tables can deliver expected constant search time for membership queries [11, 19, 12, 9, 13], all concurrent search structures that support predecessor queries have had depth and search time that is logarithmic in m, the number of keys in the set (without accounting for the cost of contention, which is typically not analyzed). This contrasts with the sequential world, in which van Emde Boas Trees [21], x-fast tries and y-fast tries [22] are known to support predecessor queries in $O(\log\log \mathbf{u})$ time, where $\mathbf{u}$ is the size of the key universe. This is, in many natural cases, a significant performance gap: for example, with $m = 2^{20}$ and $\mathbf{u} = 2^{32}$, $\log m = 20$ while $\log\log \mathbf{u} = 5$. Though one can lower this depth somewhat by increasing internal node fanout [5, 16], in the concurrent case the resulting algorithms involve complex synchronization, and their performance has never been analyzed.

This paper aims to bridge the gap between the sequential and the concurrent world by presenting the *SkipTrie*,[1] a new probabilistically-balanced data structure which supports efficient insertions, deletions and predecessor queries. We give a lock-free and linearizable implementation of the SkipTrie from CAS and DCSS instructions (see below for the reasoning behind our choice of primitives), and we analyze its expected amortized step complexity: we show that if $c(op)$ is the maximum interval contention of any operation that overlaps with op (that is, the maximum number of operations that overlap with any operation op' that itself overlaps with op), each SkipTrie operation op completes in expected amortized $O(\log\log \mathbf{u} + c(op))$ steps. We can also tighten the bound and replace the term $c(op)$ by the *point contention* of op—that is, the maximum number of operations that run concurrently at any point during the execution of op—by introducing more "helping" during searches.

The SkipTrie can be thought of as a y-fast trie [22] whose deterministic load balancing scheme has been replaced by a probabilistic one, negating the need for some of the complex operations on trees that the y-fast trie uses. Let us begin by recalling the construction of the x-fast trie and the y-fast trie from [22].

Willard's x-fast trie and y-fast trie. Given a set of integer keys $S \subseteq [\mathbf{u}]$, each represented using $\log \mathbf{u}$ bits, an *x-fast trie* over S is a hash table containing all the prefixes of keys in S, together with a sorted doubly-linked list over the keys in S. We think of the prefixes in the hash table as forming a *prefix tree*, where the children of each prefix p are $p \cdot 0$ and $p \cdot 1$ (if there are keys in S starting with $p0$ and $p1$, respectively). If a particular prefix p has no left child (i.e.,

*This work was supported in part by NSF grant CCF-1217921, DoE ASCR grant ER26116/DE-SC0008923, and by grants from the Oracle and Intel corporations.

PODC'13, July 22–24, 2013, Montréal, Québec, Canada.

[1] The name "SkipTrie" has been previously used in the context of P2P algorithms to describe an unrelated algorithm [15] that involves a traditional trie and a distributed skip-graph. We nevertheless decided to use it, as we believe it is the most fitting name for our construction.

there is no key in S beginning with $p \cdot 0$), then in the hash table entry corresponding to p we store a pointer to the largest key beginning with $p \cdot 1$; symmetrically, if p has no right child $p \cdot 1$, then in the entry for p we store a pointer to the smallest key beginning with $p \cdot 0$. (Note that a prefix is only stored in the hash table if it is the prefix of some key in S, so there is always either a left child or a right child or both.)

To find the predecessor of a key x in S, we first look for the longest common prefix of x with any element in S, using binary search on the length of the prefix: we start with the top half of x (the first $\log \mathbf{u}/2$ bits), and query the hash table to check if there is some key that starts with these bits. If yes, we check if there is a key starting with the first $3 \log \mathbf{u}/4$ bits of x; if not, we check for the first $\log \mathbf{u}/4$ bits of x. After $O(\log \log \mathbf{u})$ such queries we have found the longest common prefix p of x with any element in the set S. This prefix cannot have both left and right children, as then it would not be the *longest* common prefix. Instead it has a pointer down into the doubly linked list of keys; we follow this pointer. If p has no left child, then we know that x begins with $p \cdot 0$, and the pointer leads us to the predecessor of x, which is the smallest key beginning with $p \cdot 1$; we are done. If instead p has no right child, then x begins with $p \cdot 1$, and the pointer leads us to the successor of x, which is the largest key beginning with $p \cdot 0$. In this case we take one step back in the doubly-linked list of leaves to find the predecessor of x.

The x-fast trie supports predecessor queries in $O(\log \log \mathbf{u})$ steps, but it has two disadvantages: (1) insertions and deletions require $O(\log \mathbf{u})$ steps, as every prefix of the key must be examined and potentially modified; and (2) the space required for the hash table is $O(|S| \cdot \log \mathbf{u})$, because the depth of the prefix tree is $\log \mathbf{u}$. To remedy both concerns, Willard introduced the *y-fast trie*. The idea is to split the keys in S into "buckets" of $O(\log \mathbf{u})$ consecutive keys, and insert only the smallest key from every bucket into an x-fast trie. Inside each bucket the keys are stored in a balanced binary search tree, whose depth is $O(\log \log \mathbf{u})$.

The cost of predecessor queries remains the same: first we find the correct bucket by searching through the x-fast trie in $O(\log \log \mathbf{u})$; then we search for the exact predecessor by searching inside the bucket's search tree, requiring another $O(\log \log \mathbf{u})$ steps.

As we insert and remove elements from the y-fast trie, a bucket may grow too large or too small, and we may need to split it into two sub-buckets or merge it with an adjacent bucket, to preserve a bucket size of, say, between $\log \mathbf{u}$ and $4 \log \mathbf{u}$ elements (the constants are arbitrary). To split a bucket, we must split its search tree into two balanced subtrees, remove the old representative of the bucket from the x-fast trie, and insert the representatives of the new buckets. This requires $O(\log \mathbf{u})$ steps, but it is only performed "once in every $O(\log \mathbf{u})$ insertions", because there is a slack of $O(\log \mathbf{u})$ in the allowed bucket size. Therefore the amortized cost is $O(1)$. Similarly, to merge a bucket with an adjacent bucket, we must merge the balanced search-trees, remove the old representatives and insert the new one; we may also need to split the merged bucket if it is too large. All of this requires $O(\log \mathbf{u})$ steps but is again performed only once in every $O(\log \mathbf{u})$ operations, for an amortized cost of $O(1)$.

The y-fast trie has an amortized cost of $O(\log \log \mathbf{u})$ for all operations (the search for the predecessor dominates the cost), and its size is $O(m)$. However, this only holds true if we continuously move the elements among the collection of binary trees, so that each tree always has about $\log \mathbf{u}$ elements. This kind of rebalancing—moving items between binary trees, into and out of the x-fast trie, and finally rebalances the trees—is quite easy in a sequential setting, but can prove to be a nightmare in a concurrent one. It is more complex than simply implementing a balanced lock-free binary search tree, of which no completely proven implementation is known to date [7]. One might instead use a balanced B+ tree, for which there is a known lock-free construction [3]; but this construction is quite complicated, and furthermore, the cost it incurs due to contention has not been analyzed. Instead we suggest a more lightweight, probabilistic solution, which does not incur the overhead of merging and splitting buckets.

The SkipTrie. The main idea of the SkipTrie is to replace the y-fast trie's balanced binary trees with a very shallow, truncated skiplist [18] of depth $\log \log \mathbf{u}$. Each key inserted into the SkipTrie is first inserted into the skiplist; initially it rises in the usual manner, starting at the bottom level and tossing a fair coin to determine at each level whether to continue on to the next level. If a key rises to the top of the truncated skiplist (i.e., to height $\log \log \mathbf{u}$), we insert it into the x-fast trie (see Fig. 1). We do not store any skiplist levels above $\log \log \mathbf{u}$. The nodes at the top level of the skiplist are also linked backwards, forming a doubly-linked list. The effect is a probabilistic version of the y-fast trie: when a key x rises to the top of the skiplist and is inserted into the x-fast trie, we can think of this as splitting the bucket to which x belongs into two sub-buckets, one for the keys smaller than x and one for the keys at least as large as x. The probability that a given node will rise to level $\log \log \mathbf{u}$ is $2^{-\log \log \mathbf{u}} = 1/\log \mathbf{u}$, and therefore in expectation the number of keys between any two top-level keys is $O(\log \mathbf{u})$. Thus we achieve in a much simpler manner the balancing among buckets required for the y-fast trie. In our version, we never have to rebalance, or take keys in and out of the x-fast trie to make sure they are "well spaced-out."

To find the predecessor of x in the SkipTrie, a thread first traverses the x-fast trie to find the predecessor of x among the top-level skiplist elements, and then traverses the skiplist to find the actual predecessor of x. Finding the right top-level skiplist element takes $O(\log \log \mathbf{u})$ steps, and searching the skiplist takes an expected $O(\log \log \mathbf{u})$ additional steps, for a total of $O(\log \log \mathbf{u})$. To delete a key, we first delete it from the skiplist, and if it was a top-level node, we also remove it from the x-fast trie. As with the y-fast trie, inserting or removing a key from the x-fast trie requires $O(\log \mathbf{u})$ steps, but the amortized cost is $O(1)$, because in expectation only one in every $O(\log \mathbf{u})$ keys rises into the x-fast trie. Finally, the expected size of the SkipTrie is $O(m)$, where m is the number of keys: the skiplist's size is well-known to be $O(m)$ (in our case it is even smaller, because it is truncated), and an x-fast trie over an expected $O(m/\log \mathbf{u})$ keys requires $O(m)$ space in expectation.

So how do we design a concurrent SkipTrie? Our data structure is the composition of a concurrent hash table, a concurrent skiplist, a doubly-linked list (or two singly-linked lists sorted in opposite directions), and a concurrent implementation of an x-fast trie. For the hash table we use Split-Ordered Hashing [19], a resizable lock-free hash table

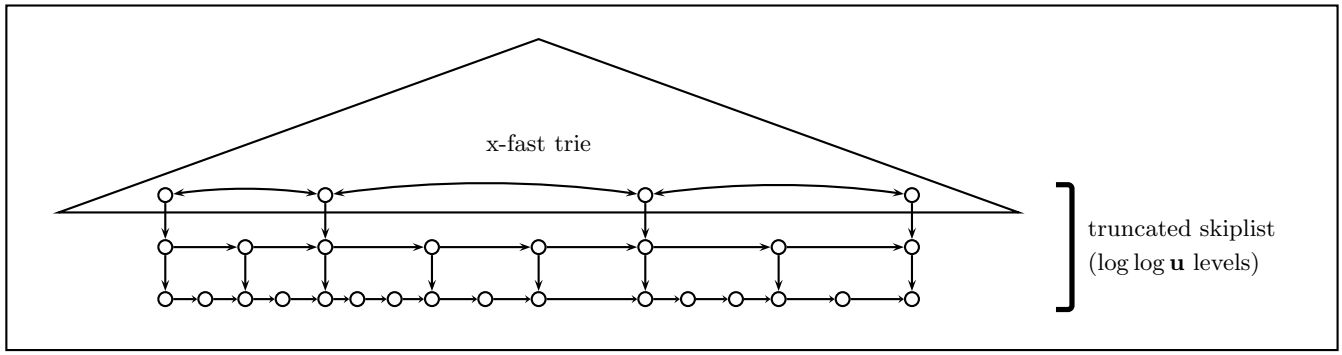

Figure 1: Illustration of a SkipTrie. The bottom is a truncated skiplist comprising $\log\log \mathbf{u}$ levels. Top-level skiplist nodes are linked in a doubly-linked list and inserted into an x-fast trie.

that supports all operations in expected $O(1)$ steps. The other components we construct and analyze in this paper: although lock-free versions exist in the literature (e.g.,[8, 11, 14]), their step complexity has not been analyzed, and it appears that the penalty these implementations take for contention is too high for our desired amortized bounds. In terms of the construction itself, the most novel part is the x-fast trie, which to our knowledge has never been implemented concurrently before; in this extended abstract we focus on the x-fast trie and give only a quick sketch of the skiplist implementation. In addition, for lack of space, we give a high-level description of the analysis.

On the choice of atomic primitives. Our implementation of the SkipTrie uses single-word compare-and-swap (CAS) and double-wide double-compare-single-swap (DCSS) operations. A DCSS(X, old_X, new_X, Y, old_Y) instruction sets the value of X to new_X, conditioned on the current values of X and Y being old_X and old_Y, respectively. We use DCSS to avoid swinging list and trie pointers to nodes that are marked for deletion: we condition the DCSS on the target of the pointer being unmarked, so that we can rest assured that once a node has been marked and physically deleted, it will never become reachable again.

Although DCSS is not supported as a hardware primitive, we believe that given current trends in hardware and software transactional memory, it is quite reasonable to use lightweight transactions to implement a DCSS; indeed, support for hardware transactional memory is available in Intel's new Haswell microarchitecture, and the specialized STM of [6] sometimes outperforms hardware atomic primitives for very short transactions. Our implementation requires DCSS only for its amortized performance guarantee; we prove that even if some or all DCSS instructions are replaced with CAS (by dropping the second guard), the implementation remains linearizable and lock-free. In particular, after attempting the DCSS some fixed number of times and aborting, it is permissible to fall back to CAS.

We believe that our explicit use of DCSS captures some design patterns that are implicit in, e.g., [8], and other lock-free data structures built from CAS alone; these data structures often "need" a DCSS, so they implement it from CAS, in a way that essentially amounts to a pessimistic transaction. We believe that on modern architecture it is preferable to use an actual DCSS (i.e., a short transaction) for this type of operation, as this allows for an optimistic implementation (as well as the pessimistic one if desired). Using DCSS also reduces the amount of "helping", which we view as an advantage (see below).

The disadvantage to using non-pessimistic transactional memory is that transactions may abort spuriously, while our analysis assumes the usual semantics of a DCSS, with no spurious aborts. However, a reasonable interpretation, at least for hardware transactional memory with very short transactions, is that spurious aborts occur only with very low probability, so their cumulative expected effect is small. Moreover, because our implementation remains correct even if DCSS is replaced with CAS, we can place a limit on the number of times a DCSS spuriously aborts, after which we fall back

Addressing the cost of contention. In order to bound the step complexity of operations as a function of the contention, we show that each step taken by an operation op can be *charged* to some operation op' that is part of the "contention" on op'. For example, if op traverses across a deleted node u, this step will be charged to the operation op' that deleted node u; we must then show that op and op' contend with each other, and also that op' is only charged a constant number of times by op.

The literature defines several measures for the amount of contention on an operation op. Among them are *interval contention* [1], which is the number of operations that overlap with op, and the *point contention* [2], which counts the maximum number of operations that run concurrently at any point during op. However, we believe that attempting to bound the step complexity as a function of the interval or point contention may be too pessimistic a design philosophy, as it spends significant effort addressing situations that are unlikely to arise in practice; it requires every operation to eagerly help operations in its vicinity to complete. This creates extra write contention.

Let us illustrate this idea using the doubly-linked list that will be presented in Section 3. The list maintains both `next` and `prev` pointers in each node, but since we do not use a double-compare-and-swap (DCAS), we cannot update both pointers at the same time. Therefore, when inserting a new node, we first update the `next` pointer of its predecessor (this is the linearization point of an insert), and then we update the `prev` pointer of its successor. There is a short

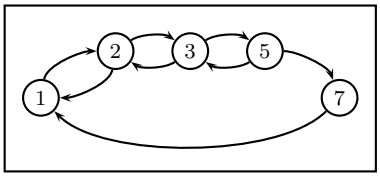

Figure 2: The doubly-linked list from the example in Section 1

interval during which the `next` pointer of the predecessor already points to the new node, but the `prev` pointer of the successor points behind it.

Suppose that the list contains nodes 1 and 7, and we begin to insert node 5 (see Fig. 2). (We abuse notation slightly by using a number $i \in \mathbb{N}$ to refer to both the key i and the node that stores key i.) After linking node 5 in the *forward* direction, but before updating the `prev` pointer of node 7, the thread inserting node 5 is preempted. Then other threads insert nodes 2 and 3 into the list. We now have a gap of 3 nodes in the backwards direction: the `prev` pointer of node 7 still points to node 1, but in the forward direction we have nodes 2, 3 and 5 between nodes 1 and 7.

If a predecessor query Q searching for 6 begins at node 7 (e.g., because it has found node 7 by searching the x-fast trie), it will step back to node 1, then forward across nodes 2, 3, and 5. We must account for the three extra steps taken by Q. Note that the insertion of node 5 is not yet complete when Q begins its search, so stepping across node 5 can be charged to the point contention of Q. But what about nodes 2 and 3? The operations inserting them are done by the time Q begins. We cannot charge these steps to the point or interval contention of Q. There are two solutions:

(1) Prevent this situation from arising, by requiring inserts to "help" other inserts. If we wish to charge the extra steps to the interval or point contention of Q, the operations `insert`(2) and `insert`(3) *must not be allowed to complete* until the list has been repaired. This can be achieved by adding a flag, u.`ready`, which indicates whether the `prev` pointer of u's successor has been set to u. In order to set u.`ready` to 1, a thread must first ensure that u.`next`.`ready` = 1 (helping u.`next` if necessary), then it must set u.`next`.`prev` to point back to u, and only then can it set u.`ready` to 1. Because of the transitive nature of helping, we may have chains of `insert`s helping each other; however, there can be no deadlock, because nodes only help their successors.

If we use this solution, then in our example, when `insert`(5) is preempted, the `ready` flag of node 5 will not be set. Hence, `insert`(3) will help `insert`(5) by trying to set 7.`prev` to node 5. In turn, if `insert`(2) and `insert`(3) proceed in lockstep, then `insert`(2) will observe that `insert`(3) needs its help; it will help node 3 by helping node 5. As a result we will have `insert`(2), `insert`(3) and `insert`(5) all contending on 7.`prev`.

If our search query, starting from node 7 and searching for node 6, encounters the scenario depicted in Fig. 2, we can now charge `insert`(2) and `insert`(3) for the extra steps, because these operations cannot have completed when the query begins. However, in order to avoid the potential extra *reads* by search queries that may or may not happen, we have created extra *write* contention by eagerly helping other operations complete. This pattern is common; for example, in the singly-linked list of [8], search queries must help clean the data structures by removing marked nodes that they come across, even though there is already a pending delete operation that will remove the marked node before it completes. This creates extra write contention on the base objects accessed by the pending delete, and is likely to slow its progress. We believe that such eager helping is not, in fact, helpful.[2]

The alternative is the following:

(2) Forgo eager helping, and instead relax our guarantee on the step complexity. This is the choice we make in the current paper. We observe that the "damage" caused by `insert`(2) and `insert`(3) is transient, and will be repaired as soon as `insert`(5) completes, setting 7.`prev` to node 5. In practice, it is unlikely that long gaps will form in the list.[3] Furthermore, the temporary gap in the list affects only *searches*, causing them to make extra reads; it does not create extra writes for any operation. Thus we allow the scenario depicted in Fig. 2, and similar scenarios. We allow operations to ignore temporary local obstacles that they come across as they traverse the list, as long as it is guaranteed that *some* operation will correct the problem before it completes.

To account for "transient" inconsistencies in the data structure, which are incurred during a recent operation and will be eliminated when the operation completes, we use the following definition:

Definition 1.1 (Overlapping-Interval Contention). *The* overlapping-interval contention $c_{OI}(op)$ *of an operation* op *is the maximum interval contention of an operation* op' *that overlaps with* op.

In our example above, the overlapping-interval contention of the query Q is at least the interval contention of `insert`(5), because these two operations overlap. In turn, the interval contention of `insert`(5) is at least 3, because, in addition to `insert`(5) itself, the operations `insert`(2) and `insert`(3) overlap with `insert`(5). Therefore we can charge the extra steps taken by Q to its overlapping-interval contention.

We remark that in our data structure, the overlapping-interval contention is only used to account for *reads*, never *writes*. The number of extra *write* steps performed by an operation op is bounded by the point contention of op.

Organization. The remainder of the paper is organized as follows. In Section 2 we sketch our new skiplist and its amortized analysis; for lack of space, we give only a brief overview. In Section 3 we give the construction of the doubly-linked list of the SkipTrie, and in Section 4 we construct the x-fast trie and sketch the proof of its linearizability and amortized step complexity.

[2] In Hebrew and Danish, the appropriate phrase is "a bear's help", referring to the tale of a man whose pet bear saw a fly on the man's nose and swatted him to death to get rid of it.

[3] For use-cases where many inserts or deletes with successive keys are frequent, it seems that approaches like flat combining are better suited in the first place.

Notation and definitions. We use $p \preceq p'$ to denote the fact that p is a prefix of p', and $p \prec p'$ to denote a proper prefix. When referring to a trie node representing a prefix p, the 0-*subtree* (resp. 1-*subtree*) of p is the set of trie nodes representing prefixes or keys p' such that $p0 \preceq p'$ (resp. $p1 \preceq p'$). The *direction of a key x under a prefix $p \prec x$* is the bit d such that x belongs to the d-subtree of p.

We let p_i denote the i-th bit of p, and $p_{[i,\ldots,j]}$ denote bits i through j of p (inclusive), and we use $\mathsf{lcp}(x, y)$ to denote the longest common prefix of x and y.

2. AN EFFICIENT LOCK-FREE CONCURRENT SKIPLIST: A BRIEF OVERVIEW

For lack of space, we give only a brief overview of our skiplist construction. Our skiplist is constructed along similar lines as Lea's skiplist [14], and uses the idea of *back links* from [8, 20, 17]. It achieves an expected amortized cost of $O(\log\log \mathbf{u} + c_I)$, where c_I is the interval contention. If we also have traversals "help" inserts and deletes by raising and lowering towers of nodes they come across, we can tighten the bound and replace c_I with the point contention c_P.

The skiplist consists of $\log\log \mathbf{u}$ levels, each of which is itself a sorted linked list. We use the logical deletion scheme from [10], storing each node's `next` pointer together with its `marked` bit in one word. In addition, a list node contains a pointer `back` pointing backwards in the list, which allows operations to recover if the node is deleted "from under their feet"; a Boolean flag, `stop`, which is set to 1 when an operation begins deleting the node's tower in order to stop inserts from raising the tower further; and a pointer `down`, which points to the corresponding tower node on the level below.

A key procedure of the skiplist implementation is the search procedure, $\texttt{listSearch}(x, \texttt{start})$, which takes a node `start` on some level ℓ with $\texttt{start.key} < x$, and returns a pair $(\texttt{left}, \texttt{right})$ of nodes on level ℓ such that $\texttt{left.key} < x \leq \texttt{right.key}$, and moreover, at some point during the invocation, `left` and `right` were both unmarked, and we had $\texttt{left.next} = \texttt{right}$. A similar function is used in the linked lists of [10] and [8]. This function also performs cleanup if necessary: if a marked node prevents it from finding an unmarked pair of nodes $(\texttt{left}, \texttt{right})$ as required (e.g., because there is a marked node between `left` and `right`), then `listSearch` will unlink the node. We use `listSearch` whenever we need to find the predecessor of a key x on a given level.

A traversal of the skiplist to find the predecessor of a key x begins at the top level, `TOP`, from some node $\texttt{start}_{\texttt{TOP}}$ that has $\texttt{start}_{\texttt{TOP}}\texttt{.key} < x$. When we descend to level ℓ, we make a call to $\texttt{listSearch}(x, \texttt{start}_\ell)$, where $\texttt{start}_\ell$ is the point from which we began the traversal on level ℓ, to obtain a pair $(\texttt{left}_\ell, \texttt{right}_\ell)$ bracketing the key x. Then we set $\texttt{start}_{\ell-1} \leftarrow \texttt{left}_\ell\texttt{.down}$ and descend to the next level, where we call $\texttt{listSearch}(x, \texttt{start}_{\ell-1})$. Eventually we reach level 0; if $\texttt{right}_0\texttt{.key} = x$ then we return $\texttt{right}_0$, and otherwise we return $\texttt{left}_0$.

To insert a new key x, we first descend down the skiplist and locate the predecessor of x on every level. We choose a height $H(x) \sim \mathit{Geom}(1/2)$ for x. We first create a new node for x and insert it on the bottom level; this node is called the *root*. Then we move up and insert x into every level up to $\min\{H(x), \texttt{TOP}\}$. Each insertion is conditioned on the `stop` flag of the root remaining unset. To delete a key x, we find the root node of the tower corresponding to x, and set its `stop` flag. Then we delete x's tower nodes top-down, starting at the highest level on which x has been inserted.

Perhaps the most interesting feature of our skiplist is the analysis of its expected amortized step complexity. There are two key ideas:

1. How many times can an insert or delete operation "interfere" with other operations? If an insert or delete *op* causes a CAS or DCSS performed by another operation *op′* to fail, then *op* is charged for the extra step by *op′*. As two inserts go up the skiplist in parallel, or as two deletes go down, they may interfere with each other at most once on every level, because one successful CAS or DCSS is all that is necessary for an operation to move on from its current level.[4] However, they *can* interfere with each other on multiple levels. This might seem to lead to an amortized cost of $O(c_{OI} \cdot \log\log \mathbf{u})$ instead of $O(\log\log \mathbf{u} + c_{OI})$, as we may end up charging each operation once per level. However, we are saved by observing that an operation operation working on key x only "interferes" on levels $h \leq H(x)$, and $H(x)$ is geometrically distributed. Thus, in expectation, each operation only interferes with others on a constant number of levels, and the cost of contention is linear in expectation.

2. How can we bound the expected amortized cost of a skiplist traversal? We modify the beautiful analysis due to Pugh in [18], where he argues as follows: consider the traversal *in reverse order*, from the node found on the bottom level back to the top. Each time we make a "left" step in the inverted traversal, this corresponds to a node that made it to the current level, but did not ascend to the next level—otherwise we would have found it there, and we would not have needed to move to it on the level below. The probability of this is 1/2, and the expected number of "left" steps between "up" steps is 2. Therefore, in expectation, the total number of "left" steps in the traversal is on the order of the height of the skiplist.

In our case, in addition to "left" and "up" (i.e., reversed "right" and "down") steps, we also have the following types of reversed steps:

- "Right" (reversed "left"): the current node in the traversal was marked, and we had to backtrack using its `back` pointer. We charge this step to the operation that marked the node. We show that the operation overlaps with the traversal, so this is permitted, and that we never step back over this node again.
- "Cleanup": the traversal stays in place, but CAS-es a marked node out of the list. This is again charged to the operation that marked the node.
- "Left that should have been up": we move to a node u on level ℓ, but the "true" height $H(u)$ chosen for u is greater than ℓ. Pugh's analysis does not account for this case, because it concerns only the "true" heights chosen for the nodes. In our case, we can show that since u's "true" height is greater than ℓ, we were "supposed" to find it before descending to level ℓ, and the reason we did not is that either u's tower is currently being raised by an insert, that had not reached level

[4]Unless the CAS or DCSS is performed in order to "help" some other operation, but then the other operation will be charged instead.

$\ell + 1$ when we descended to level ℓ, or u's tower is being lowered by a delete that had already removed the tower node from level $\ell + 1$ but has not yet reached level ℓ. We charge the inserting or deleting operation. Note that because traversals do not "help" inserts or deletes, we may charge the insert or delete once for every traversal that it overlaps with; this is why our bound is stated in terms of the interval contention rather than the point contention. To get the bound down to the point contention, traversals must help raise or lower towers, so that this situation is avoided.

The only steps that are not accounted for by charging other operations are the steps that Pugh's original analysis covers. Thus the expected amortized step complexity is $O(\log\log \mathbf{u} + c_I)$.

3. THE DOUBLY-LINKED LIST

Our doubly-linked list is built on top of the skiplist sketched in Section 2. We add to each top-level skiplist node a pointer, `prev`, which points to a node with a strictly smaller key. For linearizability we rely only on the forward direction of the list; the `prev` pointers are used only as "guides".

To insert a key into the list we call `topleveLinsert`, a procedure whose code is omitted here. In `topleveInsert` we create a new node, u, and insert it into the skiplist. If node u has reached the top level of the skiplist, we call `fixPrev` to set node u's `prev` pointer. Inside `fixPrev`, we locate node u's predecessor v in the list by calling `listSearch` with u's key, and then we attempt to set u.`prev` to v, provided that v remains unmarked and has v.`next` $= u$. If our attempt fails, we re-try, until we either succeed in setting u.`prev` or node u becomes marked. Upon success we set u.`ready` $\leftarrow 1$ to indicate that we have finished inserting node u into the doubly-linked list.

To delete a node u from the top level of the skiplist, we first ensure that u has been completely inserted (i.e., its `prev` pointer has been set), and if not, we finish inserting u by calling `fixPrev`. Then we delete it from all levels of the skiplist (top-down), which ensures that no `next` pointer points to u on any level. Finally, we remove u from the backwards direction on the top level by finding its successor v on the top level, and calling `fixPrev` to adjust v.`prev` so that it will no longer point back to u. If v became marked in the process, we find the new successor of u and try again. This ensures a type of local consistency for the doubly-linked list: the operation deleting u cannot complete until it has "observed" (inside `fixPrev`) a pair of unmarked nodes w, z that bracket u's key (w.`key` $<$ u.`key` $\leq$ z.`key`) and have w.`next` $= z$ and z.`prev` $= w$.

The key property of the top-level `prev` pointers is the following (addressing the scenario we described in Section 1 and Fig. 2):

LEMMA 3.1. *Suppose that u is an unmarked top-level node with u.`prev` $= v$, and there is a chain of nodes $u_0, \ldots, u_k$ such that $k > 1$, $u_0 = v$, $u_k = u$ and for each $i = 0, \ldots, k-1$ we have u_i.`next` $= u_{i+1}$, then the operation op inserting node u_{k-1} is still active, and furthermore, if $k > 2$, the operations that inserted nodes $u_1, \ldots, u_{k-2}$ overlap with op.*

Lemma 3.1 allows us to account for extraneous forward-steps in the list: it shows that if, after following the `prev` pointer of some node u, we must step forward across a chain of nodes $u_0, \ldots, u_k$, then the extra steps to cross nodes $u_1, \ldots, u_{k-1}$ are covered under the interval contention of the operation inserting node u_{k-1}, and this operation is still active.

In addition to the lemma, we also rely on the fact that `prev` pointers are never set to marked nodes (this is a condition of the DCSS in `fixPrev`). Thus, if we follow a pointer u.`prev` and reach a marked node v, we know that v was marked after u.`prev` was updated for the last time. We will see in Section 4.2 that we only need to follow u.`prev` if node u was inserted during the current operation. Therefore we can conclude that crossing the marked node, v, is covered under the overlapping-interval contention of the current operation.

4. A LOCK-FREE CONCURRENT IMPLEMENTATION OF AN X-FAST TRIE

In this section we describe our lock-free implementation of an x-fast trie, and sketch its proof of linearizability and the analysis of its expected amortized performance.

In a sequential x-fast trie, only *unary nodes*, which are missing either a 0-subtree or a 1-subtree, store pointers into the linked list of keys. This is sufficient, because a binary search for the longest common prefix can never end at a binary node—if a prefix has both 0-children and 1-children, it is not the longest common prefix with any key. However, in a concurrent implementation it is useful to store pointers into the linked list even in binary nodes: suppose that a predecessor query Q looking for the predecessor of x looks up a prefix $p \prec x$ and sees that it exists in the trie, but the node representing it is binary and therefore stores no pointer into the linked list. Immediately afterwards, delete operations remove all the 0-children of p. Because Q found p in the trie, its future lookups will all be for longer prefixes p', where $p \cdot 0 \preceq p'$. But because all 0-children of p were deleted, these lookups will all fail, leaving Q "stuck" with no pointer into the linked list. The only possible recovery is to try to backtrack up the trie or to re-start the search, but these solutions are too expensive. Instead, we store in each trie node two pointers, to the largest child in the 0-subtree and the smallest child in the 1-subtree. This ensures that a query always holds a pointer to a top-level skiplist node, and indeed to a node that is not too far from its destination (and is updated to closer and closer nodes as the search progresses).

The data structure. The concurrent x-fast trie consists of a hash table `prefixes`, mapping prefixes to tree nodes representing them. A tree node `n` has a single field, `n.pointers`, which stores two pointers `n.pointers`$[0]$, `n.pointers`$[1]$ to the largest element in the 0-subtree and the smallest element in the 1-tree, respectively. Recall that "underneath" the x-fast trie we store all keys in a skiplist; the nodes pointed to by `n.pointers`$[d]$ for $d \in \{0, 1\}$ are top-level skiplist nodes. A value of `n.pointers`$[d]$ = `null` indicates that node `n` has no children in its d-subtree (except possibly new children currently being inserted).

Our goal is to ensure that `n.pointers`$[0]$ always points to the largest node in the 0-subtree that has been completely inserted (and not deleted yet), and symmetrically for `n.pointers`$[1]$; and furthermore, that if the deletion of top-level skiplist node u has completed, then it is not pointed to by any trie node. In a sense, we can think of each trie node as a linearizable pair of pointers, reflecting all insert

```
1  (left, right) ← listSearch(node.key, pred)
2  while !node.marked do
3  |  node_prev ← node.prev
4  |  if DCSS(node.prev, node_prev, left, left.succ, (node, 0)) then return
5  |  (left, right) ← listSearch(node.key, pred)
6  node.ready ← 1
```

Algorithm 1: `fixPrev(pred, node)`

```
1  if !node.ready then
2  |  fixPrev(pred, node)
3  skiplistDelete(pred, node)
4  repeat
5  |  (left, right) ← listSearch(node.key, right)
6  |  fixPrev(left, right)
7  until !right.marked
```

Algorithm 2: `toplevelDelete(pred, node)`

and delete operations that have already "crossed" the level of the trie node.

The hash table. As mentioned in Section 1, we use Split-Ordered Hashing [19] to implement the `prefixes` hash table. We require one additional method, `compareAndDelete`(p, n), which takes a prefix p and a trie node n, and removes p from `prefixes` iff the entry corresponding to p contains node n. This is easily achieved in the hash table of [19] by simply checking that p's entry corresponds to n before marking it.

4.1 X-Fast Trie Operations

Predecessor queries. For convenience, we divide our predecessor query into three procedures.

To find the predecessor of a key x, we first find its longest common prefix in the x-fast-trie using the `LowestAncestor` function, which performs a binary search on prefix length to locate the lowest ancestor of x in the tree; this is the node representing the longest common prefix that x has with any key in the trie. During the binary search, the query always remembers the "best" pointer into the linked list it has seen so far— the node whose key is closest to x. After $\log\log \mathbf{u}$ steps, `LowestAncestor` finishes its binary search and returns the "best" pointer into the doubly-linked list (i.e., the top level of the skiplist) encountered during the search.

The node returned by `LowestAncestor` may be marked for deletion, and its key may be greater than x. Therefore, inside the procedure `xFastTriePred`, we traverse `back` pointers (if the node is marked) or `prev` pointers (if the node is unmarked) until we reach a top-level skiplist node whose key is no greater than x. Finally, we call `skiplistPred` to find the true predecessor of x among all the keys in the SkipTrie.

```
1  curr ← LowestAncestor(key)
2  while curr.key > key do
3  |  if curr.marked then curr ← curr.back
4  |  else curr ← curr.prev
5  return curr
```

Algorithm 4: `xFastTriePred(key)`

Insert operations. To insert a new key x, we first insert it into the skiplist; this is the linearization point of a successful insert, as after this point all searches will find x (until it is deleted). Then, if x reached the top level of the skiplist, we insert the prefixes of x into the trie as follows: for each prefix $p \preceq x$ from the bottom up (i.e., longer prefixes first), we look up the tree node corresponding to p in the `prefixes` hash table. If no such node is found, we create a new tree node pointing down to x and try to insert it into `prefixes`. Upon success, we return; upon failure, we start the current level over.

If a node is found but its `pointers` field is $(\texttt{null}, \texttt{null})$, we know that it is slated for deletion from `prefixes`; we help the deleting operation by deleting the node from `prefixes` ourselves. Then we start the current level over.

Finally, suppose that a tree node `n` is found in the hash table, and it has `n.pointers` $\neq (\texttt{null}, \texttt{null})$. Let d be the direction of x's subtree under p, that is, $d \in \{0, 1\}$ satisfies $p \cdot d \preceq x$. If $d = 0$, then `n.pointers`$[d]$ should point to the largest key in `n`'s subtree and if $d = 1$ then `n.pointers`$[d]$ should point to the smallest key in `n`'s subtree. Thus, if $d = 0$ and `n.pointers`$[d]$`.key` $\geq x$, or if $d = 1$ and `n.pointers`$[d]$`.key` $\leq x$, then x is adequately represented in `n`, and we do not need to change anything; otherwise we try to swing the respective pointer `n.pointers`$[d]$ to point down to x, conditioned on x remaining unmarked.

```
1  return skiplistPred(key, xFastTriePred(key))
```

Algorithm 5: `predecessor(key)`

Delete operations. To delete a key x, we first locate its predecessor `pred` among all top-level skiplist nodes, then try to delete x from the skiplist by calling either `skiplistDelete`, if the node is not a top-level node, or `toplevelDelete` if it is a top-level node. If we succeed, and if x was a top-level skiplist node, then we need to update x's prefixes in the trie so that they do not point down to the node u we just deleted. We go over the prefixes top-down (in increasing length), and for each prefix $p \prec x$, we check if `prefixes` contain an entry for p; if not, we move on to the next level. If it does contain an entry `n` for p, but `n.pointers` does not point down to u, we also move on. Finally, if `n.pointers`$[d] = u$ (where d is the direction of u under p, that is, $p \cdot d \preceq x$), then we call `listSearch` to find a pair of nodes $(\texttt{left}, \texttt{right})$ that "bracket" key x on the top level of the skiplist: these nodes

```
1  common_prefix ← ε
2  start ← 0 // The index of the first bit in the search window
3  size ← log u/2 // The size of the search window
4  ancestor ← prefixes.lookup(ε).pointers[key_0] // The lowest ancestor found so far
5  while size > 0 do
6      query ← common_prefix · (key_[start,start+size−1])
7      direction ← key_{start+1}
8      query_node ← prefixes.lookup(query)
9      if query_node ≠ null then
10         candidate ← query_node.pointers[direction]
11         if candidate ≠ null and query ⪯ candidate.key then
12             if |key − candidate.key| ≤ |key − ancestor.key| then
13                 ancestor ← candidate
14             common_prefix ← query
15             start ← start + size
16     size ← size/2
17 return ancestor
```

Algorithm 3: `LowestAncestor(key)`

```
1  pred ← xFastTriePred(key); if pred.key = key then return false
2  node ← toplevelInsert(key, pred)
3  if node = null then return false // key was already present
4  if node.orig_height ≠ TOP then return true // A non-top-level node was created
   // Insert into the prefix tree
5  for i = log u − 1, ..., 0 do
6      p ← key_[0,...,i]; direction ← key_{i+1}
7      while !node.marked do
8          tn ← prefixes.lookup(p)
9          if tn = null then // Create an entry for prefix p
10             tn ← new treeNode()
11             tn.pointers[direction] ← node
12             if prefixes.insert(p, tn) then break
13         else if tn.pointers = (null, null) then // The entry for p is in the process of being deleted; help delete it
14             prefixes.compareAndDelete(p, tn)
15         else
16             curr ← tn.pointers[direction]
17             if curr ≠ null and [(direction = 0 and curr.key ≥ key) or (direction = 1 and curr.key ≤ key)] then break
18             node_next = node.next
19             if DCSS(tn.pointers[direction], curr, node, node.succ, (node_next, 0)) then break
20 return true
```

Algorithm 6: `insert(key)`

satisfy `left.next` = `right` and `left.key` $< x \leq$ `right.key`, and both of them are unmarked. If $d = 0$, then we swing `n.pointers`$[d]$ backwards to point to `left`, and if $d = 1$ we swing `n.pointers`$[d]$ forward to point to `right`. In both cases we condition the switch on the new target, `left` or `right`, remaining unmarked and adjacent to `right` or `left` (respectively) on the top level of the skiplist.

4.2 Analysis of the Trie

Because `prefixes` is a linearizable hash table, our analysis treats it as an atomic object. For convenience we let `prefixes`$[p]$ denote the value currently associated with p in the hash table (or `null` if there is none); that is, the result of `prefixes.lookup`$[p]$. We also abuse notation slightly by using `insert`(u) and `delete`(u) to refer to an operation in the SkipTrie whose top-level skiplist node is u (i.e., an `insert` that created node u, or a `delete` that marked node u).

Linearizability of the trie is very easy to show: all we have to show is that for each prefix p and direction d, if `prefixes`$[p] \neq$ `null` and `prefixes`$[p]$`.pointers`$[d] \neq$ `null`, then `prefixes`$[p]$`.pointers`$[d]$ points to some node that was reachable in the top level of the skiplist at some point in time. This holds even if CAS is used instead of DCSS. The properties of the doubly-linked list and the skiplist, which are themselves linearizable, then guarantee that we will find the predecessor of x.

For the amortized step complexity, we define the notion of an operation *crossing* a certain level, which is informally when its changes "take effect" on that level. We say that `insert`(u) *crosses level* ℓ when one of the following occurs inside the ℓ-th iteration of the for-loop: (1) a new node for prefix p is successfully inserted into `prefixes` in line 12, (2) the condition in line 17 evaluates to **true**, or (3) a successful DCSS occurs in line 19. We say that `delete`(u) *crosses level* ℓ when one of the following occurs inside the ℓ-th iteration of the for-loop: (1) we set `tn` to `null` in line 8, or (2) we set `curr` to a value different from `node` in line 10 or line 18. We can show that once an operation has crossed a certain

```
1  pred ← predecessor(key − 1);
2  (left, node) ← listSearch(key, pred)
3  if node.orig_height ≠ TOP then return skiplistDelete(left, node)
4  if !toplevelDelete(left, node) then return false
5  for i = 0, ..., log u − 1 do
6      p ← key_[0,...,i]
7      direction ← key_{i+1}
8      tn ← prefixes.lookup(p)
9      if tn = null then continue
10     curr ← tn.pointers[direction]
11     while curr = node do
12         (left, right) ← listSearch(key, left)
13         if direction = 0 then
14             DCSS(pointers[direction], curr, left, left.succ, (right, 0))
15         else
16             makeDone(left, right)
17             DCSS(pointers[direction], curr, right, (right.prev, right.marked), (left, 0))
18         curr ← tn.pointers[direction]
19     if !(p ⪯ curr.key) then // the sub-tree corresponding to p · direction has become empty
20         CAS(tn.pointers[direction], curr, null)
21     if tn.pointers = (null, null) then // the entire sub-tree corresponding to p has become empty
22         prefixes.compareAndDelete(p, tn)
23 return true
```

Algorithm 7: `delete(key)`

level, its effects on that level are persistent. For `insert`(u), this means that the trie points to a node "at least as good as u", and for `delete`(u), this means that u will never be reachable from the trie.

LEMMA 4.1. *Suppose that* `insert`(u) *has crossed level* ℓ *and* u *is unmarked. Let* p *be the length-*ℓ *prefix of* u.`key`*, and let* $d \in \{0,1\}$ *be such that* $p \cdot d \preceq u$.`key`*. Then*

(a) `prefixes`$[p] \neq$ `null` *and* `prefixes`$[p]$.`pointers`$[d] \neq$ `null`*, and*

(b) If $d = 0$ *then* `prefixes`$[p]$.`pointers`$[d]$.`key` $\geq u$.`key`*, and if* $d = 1$ *then* `prefixes`$[p]$.`pointers`$[d]$.`key` $\leq u$.`key`*.*

If `delete`(u) *has crossed level* ℓ*, then either* `prefixes`$[p] =$ `null`*, or* `prefixes`$[p]$.`pointers`$[d] \neq u$.`key`*.*

When $Q =$ `predecessor`(x) finishes the binary search in the trie and lands in the top level of the skiplist, it may have to traverse `prev` pointers, `back` pointers of deleted nodes, and `next` pointers of unmarked nodes that stand between Q and its destination, the predecessor of x on the top level. We call the length of this top-level traversal *the list cost of* Q. To account for this cost, we calculate the *charge* that Q picks up as it searches through the tree: every time Q queries some level and "misses" an `insert` or `delete` that has not yet crossed that level, that operation pays Q one budget unit, which Q can then consume to traverse through the doubly-linked list.

More formally, we represent Q's binary search through the trie as a complete binary tree, where each node a is labeled with a search query $q(a) \in \{0,1\}^*$. The root is labeled with the first query $x_{[0,\ldots,\log \mathbf{u}/2]}$ performed by the search. For an inner node a, the left child of a is labeled with the query performed if $q(a)$ is not found in the hash table, and the right child of a is labeled with the query performed if $q(a)$ *is* found. For example, the root's children are labeled $x_{[0,\ldots,\log \mathbf{u}/4]}$ and $x_{[0,\ldots,3\log \mathbf{u}/4]}$ respectively.

The leaves of the tree correspond to the result of the binary search; for a sequential trie, this is the longest common prefix of x with any key in the trie. In the concurrent trie this is no longer true, due to concurrent insertions and deletions: the contents of the `prefixes` hash table does not accurately reflect keys which are currently being inserted or removed. However, we can show that the search behaves properly with respect to keys which are *not* currently being inserted or removed.

Suppose that inside Q, `LowestAncestor`(x) eventually returns a node u with u.`key` $= y$. The path Q follows through its binary search tree is the path from the root to $\mathsf{lcp}(x, y)$, as y is the key returned. For a key $z \neq y$, define the *critical point for* z in Q's binary search tree to be the highest node where the path from the root to $\mathsf{lcp}(x, y)$ diverges from the path to $\mathsf{lcp}(x, z)$. Then using Lemma 4.1 we can show:

LEMMA 4.2. *If* v *is an unmarked node with* v.`key` $= z$ *such that* $|z - x| < |y - x|$ *and* $\text{sign}(z - x) = \text{sign}(y - x)$ *(that is,* y *and* z *are "on the same side" of* x*), and* a *is the critical point for* z *in* Q*'s binary search tree, then when* Q *queried* $q(a)$*, the operation that inserted* v *had not yet crossed level* $|q(a)|$*.*

Thus, on a very high level, for any unmarked node v that is closer to the target node (i.e., closer to the predecessor of x on the top level of the skiplist) than the node returned by `LowestAncestor`, we can charge the operation that was not yet done inserting v when Q passed the critical point for v.`key`. In particular, when we follow `prev` pointers, the nodes we cross were inserted during Q. For marked nodes that are traversed, we show that we can charge the operation that marked them, and this is covered under the overlapping-interval contention. And finally, if, after moving backwards following `prev` pointers, we went "too far back" and must make extra forward steps, then Lemma 3.1 shows that all nodes we cross are either newly-inserted nodes or are covered under the overlapping-interval contention of Q. Therefore Q is adequately compensated for its list cost, and we can show

that its expected amortized step complexity, considering just the x-fast trie (i.e., the binary search and the traversal in the top level of the skiplist), is $O(\log\log \mathbf{u} + c_{OI})$.

The remainder of the proof consists of "assigning the blame" for each DCSS instruction and hash-table operation executed by `insert` and `delete`. To cross a level, each operation needs to perform only one successful DCSS or hash-table operation. Every *failed* DCSS instruction or operation on `prefixes` is charged to the operation that caused it to fail, by executing some successful DCSS or operation on `prefixes`. The various trie operations can charge each other at most once per level, for a total amortized cost of at most $O(c_P \cdot \log \mathbf{u})$ when considering the x-fast trie by itself. However, because in expectation only one in every $\log \mathbf{u}$ SkipTrie operation must insert or delete from the x-fast trie, the expected amortized cost of SkipTrie insertions and deletions in the x-fast trie is only $O(c_P + \log\log \mathbf{u})$. Thus, the dominant cost is the cost of predecessor queries. (Note that $c_P \leq c_I \leq c_{OI}$ for any operation.)

Because expected amortized step complexity is compositional, we can combine our analysis of the various components of the SkipTrie to obtain the following bounds:

THEOREM 4.3. *The SkipTrie is a linearizable, lock-free data structure supporting* `insert`, `delete` *and* `predecessor` *queries in expected amortized* $O(\log\log \mathbf{u} + c_{OI})$*, where* c_{OI} *is the overlapping-interval contention of the operation.*

5. REFERENCES

[1] Yehuda Afek, Gideon Stupp, and Dan Touitou. Long lived adaptive splitter and applications. *Distrib. Comput.*, 15(2):67–86, April 2002.

[2] Hagit Attiya and Arie Fouren. Algorithms adapting to point contention. *J. ACM*, 50(4):444–468, July 2003.

[3] Anastasia Braginsky and Erez Petrank. A lock-free b+tree. In *Proceedings of the 24th ACM Symposium on Parallelism in Algorithms and Architectures, SPAA '12, Pittsburgh, PA, USA, June 25-27, 2012*, pages 58–67, 2012.

[4] Nathan G. Bronson, Jared Casper, Hassan Chafi, and Kunle Olukotun. A practical concurrent binary search tree. *SIGPLAN Not.*, 45:257–268, January 2010.

[5] R. Brown. Calendar queues: A fast priority queue implementation for the simulation event set problem. *Communications of the ACM*, 31(10):1220–1227, 1988.

[6] Aleksandar Dragojević and Tim Harris. Stm in the small: trading generality for performance in software transactional memory. In *Proceedings of the 7th ACM european conference on Computer Systems*, EuroSys '12, pages 1–14, 2012.

[7] Faith Ellen, Panagiota Fatourou, Eric Ruppert, and Franck van Breugel. Non-blocking binary search trees. In *Proceedings of the 29th ACM SIGACT-SIGOPS symposium on Principles of distributed computing*, PODC '10, pages 131–140, New York, NY, USA, 2010. ACM.

[8] M. Fomitchev and E. Ruppert. Lock-free linked lists and skip lists. In *PODC '04: Proceedings of the twenty-third annual ACM symposium on Principles of distributed computing*, pages 50–59, New York, NY, USA, 2004. ACM Press.

[9] Hui Gao, Jan Friso Groote, and Wim H. Hesselink. Almost wait-free resizable hashtable. In *IPDPS*, 2004.

[10] T. Harris. A pragmatic implementation of non-blocking linked-lists. In *Proceedings of the 15th International Conference on Distributed Computing*, DISC '01, pages 300–314, 2001.

[11] Maurice Herlihy and Nir Shavit. *The Art of Multiprocessor Programming*. Morgan Kaufmann, NY, USA, 2008.

[12] Maurice Herlihy, Nir Shavit, and Moran Tzafrir. Hopscotch hashing. In *Proceedings of the 22nd international symposium on Distributed Computing*, DISC '08, pages 350–364, Berlin, Heidelberg, 2008. Springer-Verlag.

[13] D. Lea. Concurrent hash map in JSR166 concurrency utilities. `http://gee.cs.oswego.edu/dl/concurrency-interest/index.html`.

[14] D. Lea. Concurrent skiplist map. in java.util.concurrent. `http://gee.cs.oswego.edu/dl/concurrency-interest/index.html`.

[15] Li Meifang, Zhu Hongkai, Shen Derong, Nie Tiezheng, Kou Yue, and Yu Ge. Pampoo: an efficient skip-trie based query processing framework for p2p systems. In *Proceedings of the 7th international conference on Advanced parallel processing technologies*, APPT'07, pages 190–198, Berlin, Heidelberg, 2007. Springer-Verlag.

[16] Aleksandar Prokopec, Nathan G. Bronson, Phil Bagwell, and Martin Odersky. Concurrent tries with efficient non-blocking snapshots. In *Proceedings of Principles and Practice of Parallel Programming*, New Orleans, USA, 2012.

[17] W. Pugh. Concurrent maintenance of skip lists. Technical Report CS-TR-2222.1, Institute for Advanced Computer Studies, Department of Computer Science, University of Maryland, 1989.

[18] W. Pugh. Skip lists: a probabilistic alternative to balanced trees. *ACM Transactions on Database Systems*, 33(6):668–676, 1990.

[19] O. Shalev and N. Shavit. Split-ordered lists: Lock-free extensible hash tables. In *The 22nd Annual ACM Symposium on Principles of Distributed Computing*, pages 102–111. ACM Press, 2003.

[20] John D. Valois. Lock-free linked lists using compare-and-swap. In *Proceedings of the fourteenth annual ACM symposium on Principles of distributed computing*, PODC '95, pages 214–222, 1995.

[21] P. van Emde Boas. Preserving order in a forest in less than logarithmic time. In *Proceedings of the 16th Annual Symposium on Foundations of Computer Science*, pages 75–84, Washington, DC, USA, 1975. IEEE Computer Society.

[22] Dan E. Willard. Log-logarithmic worst-case range queries are possible in space theta(n). *Inf. Process. Lett.*, 17(2):81–84, 1983.