Zip Trees

Robert E. Tarjan* Caleb C. Levy† Stephen Timmel‡

Abstract

We introduce the *zip tree*,[1] a form of randomized binary search tree that integrates previous ideas into one practical, performant, and pleasant-to-implement package. A zip tree is a binary search tree in which each node has a numeric rank and the tree is (max)-heap-ordered with respect to ranks, with rank ties broken in favor of smaller keys. Zip trees are essentially treaps [8], except that ranks are drawn from a geometric distribution instead of a uniform distribution, and we allow rank ties. These changes enable us to use fewer random bits per node.

We perform insertions and deletions by unmerging and merging paths (*unzipping* and *zipping*) rather than by doing rotations, which avoids some pointer changes and improves efficiency. The methods of zipping and unzipping take inspiration from previous top-down approaches to insertion and deletion by Stephenson [10], Martínez and Roura [5], and Sprugnoli [9].

From a *theoretical* standpoint, this work provides two main results. First, zip trees require only $O(\log \log n)$ bits (with high probability) to represent the largest rank in an n-node binary search tree; previous data structures require $O(\log n)$ bits for the largest rank. Second, zip trees are naturally isomorphic to skip lists [7], and simplify Dean and Jones' mapping between skip lists and binary search trees [2].

1 Introducing: Zip Trees

1.1 Preliminaries

A *binary search tree* is a binary tree in which each node contains an item, each item has a key, and the items are arranged in *symmetric order*: if x is a node, all items in the left subtree of x have keys less than that of x, and all items in the right subtree of x have keys greater than that of x. Such a tree supports binary search: to find an item in the tree with a given key, proceed as follows. If the tree is empty, stop: no item in the tree has the given key. Otherwise, compare the desired key with that of the item in the root. If they are equal, stop and return the item in the root. If the given key is less than that of the item in the root, search recursively in the left subtree of the root. Otherwise, search recursively in the right subtree of the root. The path of nodes visited during the search is the *search path*. If the search is unsuccessful, the search path starts at the root and ends at a missing node corresponding to an empty subtree.

To keep our presentation simple, in this and the next section we do not distinguish between an item and the node containing it. (The data structure is *endogenous* [11].) We also assume that all nodes have distinct keys. It is straightforward to eliminate these assumptions. We call a node *binary*, *unary*, or a *leaf*, if it has two, one or zero children, respectively. We define the *depth* of a

*Department of Computer Science, Princeton University, and Intertrust Technologies; ret@cs.princeton.edu.
†Sunshine; caleb.levy@gmail.com.
‡Mathematics Department, Virginia Polytechnic Institute and State University; stimmel@vt.edu.

[1] *Zip*: "To move very fast."

node recursively to be zero if it is the root, or one plus the depth of its parent if not. We define the *height* of a node recursively to be zero if it is a leaf, or one plus the maximum of the heights of its children if not. The *left* (resp. *right*) *spine* of a node x is the path from x through left (resp. *right*) children to the node of smallest (resp. largest) key in the subtree rooted at x. The *left* (resp. *right*) *spine* of a tree is the left (resp. right) spine of its root. We represent a binary search tree by storing in each node x its left child $x.left$, its right child $x.right$, and its key, $x.key$. If x has no left (resp. right) child, $x.left = null$ (resp. $x.right = null$).

1.2 Intuition

Our goal is to obtain a type of binary search tree with small depth and small update time, one that is as simple and efficient as possible. If the number of nodes n is one less than a power of two, the binary tree of minimum depth is *perfect*: each node is either binary (with two children) or a leaf (with no children), and all leaves are at the same depth. But such trees exist only for some values of n, and updating even an almost-perfect tree (say one in which all non-binary nodes are leaves and all leaves have the same depth to within one) can require rebuilding much or all of it.

We observe, though, that in a perfect binary tree the fraction of nodes of height k is about $1/2^{k+1}$ for any non-negative integer k. Our idea is to build a good tree by assigning heights to new nodes according to the distribution in a perfect tree and inserting the nodes at the corresponding heights.

We cannot do this exactly, but we can do it to within a constant factor in expectation, by assigning each node a random rank according to the desired distribution and maintaining heap order by rank. Thus we obtain zip trees.

1.3 Definition of Zip Trees

A *zip tree* is a binary search tree in which each node has a numeric *rank* and the tree is (max)-heap-ordered with respect to ranks, with ties broken in favor of smaller keys: the parent of a node has rank greater than that of its left child and no less than that of its right child. We choose the rank of a node randomly when the node is inserted into the tree. We choose node ranks independently from a geometric distribution with mean 1: the rank of a node is non-negative integer k with probability $1/2^{k+1}$. We denote by $x.rank$ the rank of node x. We can store the rank of a node in the node or compute it as a pseudo-random function of the node (or of its key) each time it is needed. The pseudo-random function method, proposed by Aragon and Seidel [8], avoids the need to store ranks but requires a stronger independence assumption for the validity of our efficiency bounds, as we discuss in Section 3.

To insert a new node x into a zip tree, we search for x in the tree until reaching the node y that x will replace, namely the node y such that $y.rank \leq x.rank$, with strict inequality if $y.key < x.key$. From y, we follow the rest of the search path for x, *unzipping* it by splitting it into a path P containing each node with key less than $x.key$ and a path Q containing each node with key greater than $x.key$. Along P from top to bottom, nodes are in increasing order by key and non-increasing order by rank; along Q from top to bottom, nodes are in decreasing order by both rank and key. Unzipping preserves the left subtrees of the nodes on P and the right subtrees of the nodes on Q. We make the top node of P the left child of x and the top node of Q the right child of x. Finally, if y had a parent z before the insertion, we make x the left or right child of z depending on whether its key is less than or greater than that of z, respectively (x replaces y as a child of z); if y was the root before the insertion, we make x the root.

Deletion is the inverse of insertion. To delete a node x, we do a search to find it. Let P and Q

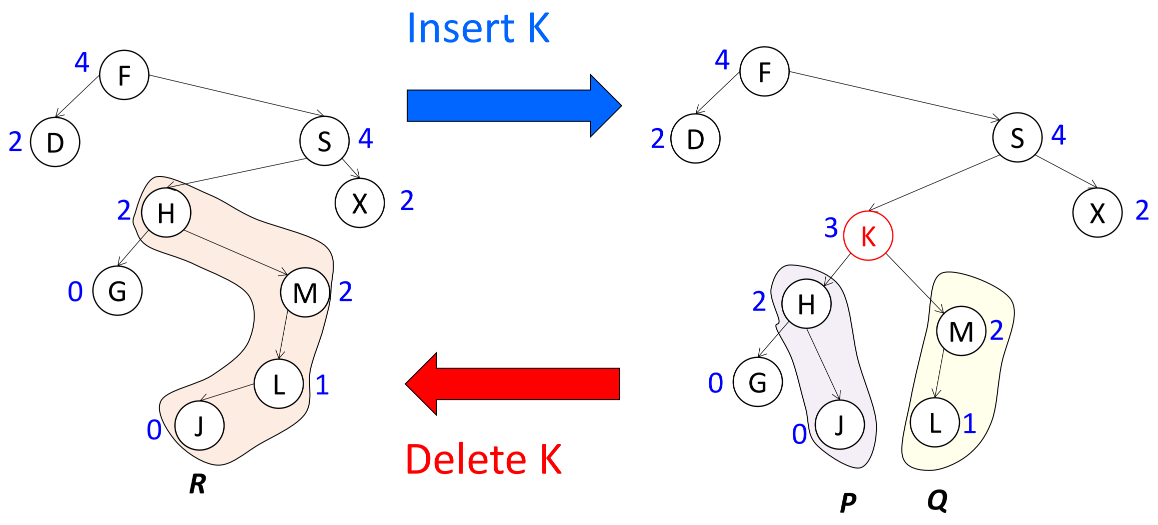

Figure 1: Insertion and deletion of a node with key “K” assigned rank 3.

be the right spine of the left subtree of x and the left spine of the right subtree of x. *Zip* P and Q to form a single path R by merging them from top to bottom in non-increasing rank order, breaking a tie in favor of the smaller key. Zipping preserves the left subtrees of the nodes on P and the right subtrees of the nodes on Q. Finally, if x had a parent z before the insertion, make the top node of R (or *null* if R is empty) the left or right child of z, depending on whether the key of x is less than or greater than that of z, respectively (the top node of R replaces x as a child of z); if x was the root before the insertion, make the top node of R the root. Figure 1 illustrates both an insertion and a deletion in a zip tree.

An insertion or deletion requires a search plus an unzip or zip. The time for an unzip or zip is proportional to one plus the number of nodes on the unzipped path in an insertion or one plus the number of nodes on the two zipped paths in a deletion.

2 Related Work

Zip trees closely resemble two well-known data structures: the treap of Seidel and Aragon [8] and the skip list of Pugh [7].

2.1 Treaps

A *treap* is a binary search tree in which each node has a real-valued random rank (called a *priority* by Seidel and Aragon) and the nodes are max-heap ordered by rank. The ranks are chosen independently for each node from a fixed, uniform distribution over a large enough set that the probability of any rank tie is small. Insertions and deletions are done using *rotations* to restore heap order. A rotation at a node x is a local transformation that makes x the parent of its old parent while preserving symmetric order. See Figure 2. In general a rotation changes three children.

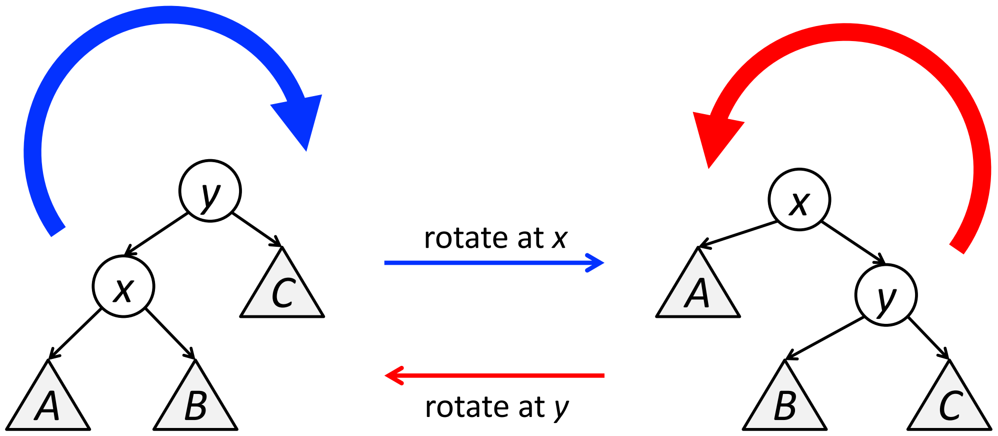

Figure 2: Rotation at node x with parent y, and reversing the effect by rotating at y. Triangles denote subtrees.

To insert a new node x in a treap, we generate a rank for x, follow the search path for x until reaching a missing node, replace the missing node by x, and rotate at x until its parent has larger rank or x is the root. To delete a node x in a treap, while x is not a leaf, we rotate at whichever of its children has higher rank. Once x is a leaf or a unary node, we replace it by its child if it has one or by a missing node if not.

One can view a zip tree as a treap but with a different choice of ranks and with different insertion and deletion algorithms.[2] Our choice of ranks reduces the number of bits needed to represent them from $O(\log n)$ to $\lg \lg n + O(1)$ (Theorem 1), if ranks are stored rather than computed as a function of the node or its key. Treaps have the same expected depth as search trees built by uniformly random insertions, namely $2 \ln n$, about $1.39 \lg n$, as compared to $1.5 \lg n$ for zip trees. The results in Section 3 correspond to results for treaps. Allowing rank ties as we do thus costs about 8% in average depth (and search time) but allows much more compact representation of priorities.

A precursor of the treap is the *cartesian tree* of Jean Vuillemin [13]. This is a binary search tree built by leaf insertion (search for the item; insert it where the search exits the bottom of the tree), with each node having a priority equal to its position in the sequence of insertions. Such a tree is min-heap ordered with respect to priorities, and its distributional properties are the same as those of a treap if items are inserted in an order corresponding to a uniformly random permutation.

2.2 Skip Lists

A *skip list* is an alternative randomized data structure that supports logarithmic comparison-based search. It consists of a hierarchy of sublists of the items. The level-0 list contains all the items. For $k > 0$, the level-k list is obtained by independently adding each item of the level-$(k-1)$ list with probability $1/2$ (or, more generally, some fixed probability p). Each list is in increasing order by key. A dummy item with key less than those of all real items is added to each list.

A search starts in the top-level list and proceeds through the items in increasing order by key until finding the desired item, reaching an item of larger key, or reaching the end of the list. In either of the last two cases, the search backs up to the item of largest key less than the search key,

[2]Seidel and Aragon [8] hint at the possibility of doing insertions and deletions by unzipping and zipping: in a footnote they say, "In practice it is preferable to approach these operations the other way around. Joins and splits of treaps can be implemented as iterative top-down procedures; insertions and deletions can then be implemented as accesses followed by splits or joins." But they provide no further details.

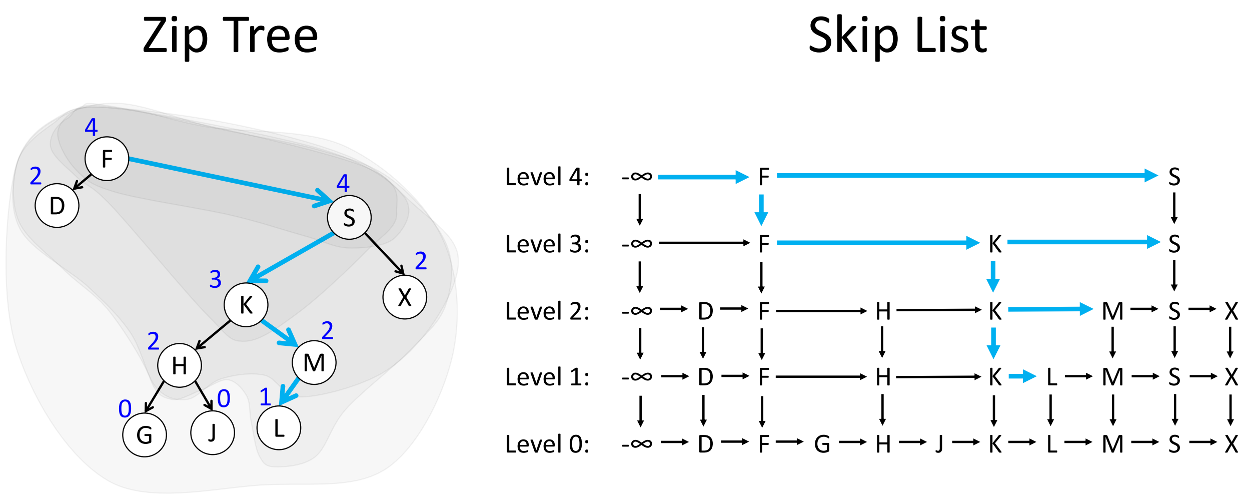

Figure 3: Representation of the zip tree in Figure 1 as a skip list. The level-k sublist comprises nodes of the zip tree of rank k or less. The search for L traverses the blue arcs in the zip tree and in the corresponding skip list.

descends to the copy of this item in the next lower-level list, and searches in this list in the same way. Eventually the search either finds the item or discovers that it is not in the level-0 list.

One can view a zip tree as a compact representation of a skip list. There is a natural isomorphism between zip trees and skip lists. (See Figure 3.) Given a zip tree, the isomorphic skip list contains item e in the level-k sublist if and only if e has rank at least k in the zip tree. Given a skip list, the isomorphic zip tree contains item e with rank k if and only if e is in the level-k sublist but not in the level-$(k+1)$ sublist. Let e be an item in the zip tree with left and right children e' and e'', respectively. Let e, e', and e'' have ranks k, k', and k'', respectively. A search in the skip list that reaches an occurrence of e will reach it first in the level-k sublist. The next node visited during the search that is not an occurrence of e will be the occurrence of e' in the level-k' sublist or the occurrence of e'' in the level-k'' sublist, depending on whether the search key is less than or greater than the key of e. Our rule for breaking rank ties in zip trees is based on the search direction in skip lists: from smaller to larger keys.

A search in a zip tree visits the same items as the search in the isomorphic skip list, except that the latter may visit items repeatedly, at lower and lower levels. Thus a zip tree search is no slower than the isomorphic skip list search, and can be faster. The skip list has at least as many pointers as the corresponding zip tree, and its representation requires either variable-size nodes, in which each item of rank k has a node containing $k+1$ pointers; or large nodes, all of which are able to hold a number of pointers equal to the maximum rank plus one; or small nodes, one per item per level, requiring additional pointers between levels. We conclude that zip trees are at least as efficient in both time and space as skip lists.

Dean and Jones were the first to provide a mapping that converts a skip list into a binary search tree [2], but it is not the natural isomorphism given in the previous paragraph. They store ranks in the binary search tree in difference form. They map the insertion and deletion algorithms for a skip list into algorithms on the corresponding binary search tree by using rotations.

2.3 Other Binary Search Tree Algorithms

Martínez and Roura [5] proposed insertion and deletion algorithms that produce trees with the same distribution as treaps. Instead of maintaining a heap order with respect to random priorities, they do insertions and deletions via random rotations that depend on subtree sizes. These sizes must be stored, at a cost of $O(\log n)$ bits per node, and they must be updated after each rotation. This suggests using their method only in an application in which subtree sizes are needed for some other purpose.

Doing insertions and deletions via unzipping and zipping takes at most one child change per node on the restructured path or paths, saving a constant factor of at least three over using rotations. Stephenson used unzipping in his root insertion algorithm [10]; insertion by unzipping is a hybrid of his algorithm and leaf insertion. Sprugnoli [9] was the first to propose insertion by unzipping. He used it to insert a new node at a specified depth, with the depth chosen randomly. His proposals for the depth distribution are complicated, however, and he did not consider the possibility of choosing an approximate depth rather than an exact depth. Zip trees choose the insertion *height* approximately rather than the depth, a crucial difference.

3 Properties of Zip Trees

If we ignore constant factors, the properties of zip trees are shared with treaps, skip lists, and the randomized search trees of Martínez and Roura. Because zip trees use a different rank distribution than treaps and Martínez and Roura's trees, and a different representation than skip lists, we reprove a selection of these properties.

3.1 Behavior of Node Ranks

We denote by n the number of nodes in a zip tree. To simplify bounds, we assume that $n > 1$, so $\log n$ is positive. We denote by $\lg n$ the base-two logarithm. The following lemma extends a well-known result for trees symmetrically ordered by key and heap-ordered by rank [8] to allow rank ties:

Lemma 1 (From [8])**.** *The structure of a zip tree is uniquely determined by the keys and ranks of its nodes.*

Proof. The lemma is immediate by induction on n, since the root is the node of largest rank whose key is smallest, and the nodes in the left and right subtrees of the root are those with keys less than and greater than the key of the root, respectively. □

By Lemma 1, a zip tree is *history-independent*: its structure depends only on the nodes it currently contains (and their ranks), independent of the sequence of insertions and deletions that built it.

In our efficiency analysis we assume that each deletion depends only on the sequence of previous insertions and deletions, independent of the node ranks. (If an adversary can choose deletions based on node ranks, it is easy to build a bad tree: insert items in arbitrary order; if any item has a rank greater than 0, immediately delete it. This will produce a path containing half the inserted nodes on average.)

Theorem 1. *The expected rank of the root in a zip tree is at most* $\lg n + 3$*. For any* $c > 0$*, the root rank is at most* $(c + 1) \lg n$ *with probability at most* $1 - 1/n^c$*.*

Proof. The root rank is the maximum of n samples of the geometric distribution with mean 1. For $c > 0$, the probability that the root rank is at least $\lg n + c$ is at most $n/2^{\lg n + c} = 1/2^c$. It follows that the expected root rank is at most $\lceil \lg n \rceil + \sum_{i=1}^{\infty} i/2^i \leq \lceil \lg n \rceil + 2 \leq \lg n + 3$. For $c > 0$, the probability that the root rank exceeds $(c+1)\lg n$ is at most $1/2^{c \lg n} = 1/n^c$. □

Let x be a node in a zip tree. If y is on the search path for x then y is an *ancestor* of x and x is a *descendant* of y. The *low* (respectively *high*) ancestors of x are the ancestors of x with key less than (respectively greater than) that of x.

Lemma 2. *Let x be a node and let ℓ be the number of low ancestors of x. Then the expected value of ℓ is at most $k+1$, and for any $\delta > 0$, $\ell \leq (1+\delta)k + 1$ with probability at least $1 - e^{-\frac{\delta^2 k}{2+\delta}}$.*

Proof. If we order the low ancestors of x in increasing order by key, they are in non-increasing order by rank. We can think of these ancestors and their ranks as being generated by coin flips in the following way. At each successive node y less than x in decreasing key order we flip a fair coin until it comes up tails and give y a rank equal to the number of heads. Given such a y, let z be the low ancestor of smallest key greater than that of y if there is such a low ancestor; otherwise, let $z = x$. Then y is a low ancestor of x if and only if its rank is at least the rank of z. We call the first $z.rank$ coin flips at y *irrelevant* and the rest *relevant*.

For example, consider the tree on the left in Figure 1, and refer to nodes by their keys. If x is node J, the successive nodes less than x in decreasing order are H, G, F, D. Initially y is H and z is J. The flips at H are head, head, tail, giving H a rank of $2 \geq 0$, so H is a low ancestor of J and is the next value of z. All three flips at H are relevant. The next value of y is G, for which the first flip is a tail, giving G a rank of 0. Since $0 < 2$, G is not a low ancestor of J; z stays equal to H. The next value of y is F, for which the flips are four heads followed by a tail, making F a low ancestor of J and changing z to F. The first two flips are irrelevant and the last three are relevant. The last value of y is D, for which the flips are two heads followed by a tail, giving D a rank of 2. Since $2 < 4$, D is not a low ancestor of J.

Node y is a low ancestor of x if and only if at least one flip at y is relevant. The relevant flips are a sequence of Bernoulli trials in which the number of tails is the number of low ancestors of x produced so far and the number of heads is at most the rank of the low ancestor of x of highest rank produced so far. Thus ℓ is the number of tails in a sequence of flips containing at most k heads and ending with a tail. Since the expected number of tails preceding each head is 1, $\ell \leq k+1$ in expectation: the "+1" counts the last tail. The second half of the lemma follows by a Chernoff bound [1]. □

Lemma 3. *Let x be a node and let h be the number of high ancestors of x. Then the expected value of h is at most $k/2$, and for any $\delta > 0$, $h \leq (1+\delta)k/2$ with probability at least $1 - e^{-\frac{\delta^2 k}{2(2+\delta)}}$.*

Proof. The proof is like that of Lemma 2. We think of generating the high ancestors of x and their ranks by flipping a fair coin until it comes up tails at each node y greater than x in increasing key order and giving y a rank equal to the number of heads. Given such a y, let z be the high ancestor of x of largest key smaller than that of y, or x if there is no such high ancestor. We call the first $z.rank + 1$ flips at y irrelevant and the rest relevant.

Node y is a high ancestor of x if and only if at least one flip at y is relevant. The relevant flips are a sequence of Bernoulli trials in which the number of tails is the number of high ancestors of x produced so far and the number of flips is at most the rank of the high ancestor of x of highest rank produced so far. Thus, h is the number of tails in a sequence of at most k flips. This is at most $k/2$ in expectation. The second half of the lemma follows by a Chernoff bound. □

3.2 Expected Zip Tree Structure

Theorem 2. *The expected depth of a node in a zip tree is at most $(3/2)\lg n + O(1)$. For $c \geq 1$, the depth of a zip tree is $O(c \lg n)$ with probability at least $1 - 1/n^c$, where the constant inside the big "O" is independent of n and c.*

Proof. One can prove this theorem using results from [6], but for completeness we prove it from scratch. The expected rank of the root is at most $\lg n + 3$ by Theorem 1. By Lemmas 2 and 3, the expected number of ancestors of a node x, including x, is at most $(3/2)\lg n + O(1)$. The second half of the Theorem follows from the high-probability bounds in Theorem 1 and Lemmas 2 and 3. □

By Theorem 2, the expected number of nodes visited during a search in a zip tree is at most $(3/2)\lg n + O(1)$, and the search time is $O(\log n)$ with high probability.

Theorem 3. *If x is a node of rank at most k, the expected number of nodes on the path that is unzipped during its insertion, and on the two paths that are zipped during its deletion, is at most $(3/2)k + 2$. For any $\delta > 0$, this number is at most $(1+\delta)(3/2)k + 2$ with probability at least $1 - 2e^{-\frac{\delta^2 k}{2(2+\delta)}}$.*

Proof. Let x be a node of rank at most k. If x is not in the tree but is inserted, the nodes on the path unzipped during its insertion are exactly those on the two paths that would be zipped during its deletion. Thus, we need only consider deletion. Let P and Q be the two paths zipped during the deletion of x, with P containing the nodes of smaller key and Q containing the nodes of larger key. Let y and z be the predecessor and successor of x in key order, respectively. Then the nodes on P are y and the low ancestors of y of rank less than $x.rank$, and the nodes on Q are z and the high ancestors of z of rank at most $x.rank$. The theorem follows from Lemmas 2 and 3. □

Theorem 4. *The expected number of pointer changes during an unzip or zip is $O(1)$. The probability that an unzip or zip changes more than $k + O(1)$ pointers is at most $2/c^k$ for some $c > 1$ independent of k.*

Proof. The expected number of pointer changes is at most one plus the number of nodes on the unzipped path during an insertion or the two zipped paths during a deletion. For a given node x, these numbers are the same whether x is inserted or deleted. Thus we need only consider the case of deletion. The probability that x has rank k is $1/2^{k+1}$. Given that x has rank k, the expected number of nodes on the two zipped paths is at most $(3/2)k + 2$ by Theorem 3. Summing over all possible values of k gives the first half of the theorem.

Choosing $\delta = 1$ in the second half of Theorem 3, we find that if x is a node of rank at most $k/3$, the number of parent changes during its insertion or deletion is more than $k + O(1)$ with probability at most $2/e^{k/18}$. Thus $c = e^{1/18}$ satisfies the second half of the theorem. □

By Theorem 4, the expected time to unzip or zip is $O(1)$, and the probability that an unzip or zip takes k steps is exponentially small in k.

In some applications of search trees, each node contains a secondary data structure, and making any change to a subtree may require rebuilding the entire subtree, in time linear in the number of nodes. The following result implies that zip trees are efficient in such applications.

Theorem 5. *The expected number of descendants of a node of rank k is at most $3(2^k) - 1$. The expected number of descendants of an arbitrary node is at most $(3/2)\lg n + 2$.*

Proof. Let x be a node of rank k. Consider the nodes with key less than that of x. Think of generating their ranks in decreasing order by key. The first such node that is not a descendant of x is the first one whose rank is at least k. All nodes smaller than this one are also non-descendants of x. The probability that a given node has rank at least k is $1/2^k$. The probability that the i^{th} node is the first of rank at least k is $(1 - 1/2^k)^{i-1}/2^k$. The expected value of i is the expected number of flips of a biased coin to get a head, if the probability of a head is $p = 1/2^k$. By a standard calculation this number is $1/p = 2^k$. Thus the expected number of descendants of x of smaller key is at most $2^k - 1$. (The expected value of i minus one is an overestimate because there are at most $n - 1$ nodes of key less than that of x and they may all have smaller rank.)

Similarly, among the nodes with key greater than that of x, the first one that is not a descendant of x is the first one with rank greater than k. A given node has rank greater than k with probability $p = 1/2^{k+1}$. The probability that the i^{th} node is the first of rank greater than k is $(1 - 1/2^{k+1})^{i-1}/2^{k+1}$. The expected value of i is 2^{k+1}, so the expected number of descendants of x of larger key is at most $2^{k+1} - 1$.

We conclude that the expected number of descendants of x, including x itself, is at most $3(2^k) - 1$. The expected number of descendants of an arbitrary node is the sum over all k of the probability that the node has rank k times the expected number of descendants of the node given that its rank is k. Using the fact that the number of descendants is at most n, this sum is at most $(3/2)\lg n + 2$, since the terms for $k \leq \lg n - 1$ sum to at most $(3/2)\lg n$ and the terms for $k > \lg n - 1$ sum to at most 2. $\square$

4 Comments and Extensions

Zip trees combine two independent ideas: the use of random ranks distributed geometrically and the use of unzipping and zipping to perform insertion and deletion. The former saves space as compared to treaps and makes zip trees isomorphic to skip lists but more efficient. In practice, allocating a byte (eight bits) per rank should suffice. The latter makes updates faster as compared to using rotations. Either idea may be used separately.

As compared to other kinds of search trees with logarithmic search time, zip trees are simple and efficient: insertion and deletion can be done purely top-down, with $O(1)$ expected restructuring time and exponentially infrequent occurrences of expensive restructuring. Certain kinds of deterministic balanced search trees, in particular weak AVL trees and red-black trees, achieve these bounds in the amortized case [3], but at the cost of somewhat complicated update algorithms.

Zipping and unzipping make catenating and splitting zip trees simple. To catenate two zip trees T_1 and T_2 such that all items in T_1 have smaller keys than those in T_2, zip the right spine of T_1 and the left spine of T_2. The top node of the zipped path is the root of the new tree. To split a tree into two, one containing items with keys at most k and one containing items with keys greater than k, unzip the path from the root down to the node x with key k, or down to a missing node if no item has key k. The top nodes of the two unzipped paths are the roots of the new trees.

If the rank of a node is a pseudo-random function of its key, then search and insertion can be combined into a single top-down operation that searches until reaching the desired node or the insertion position. Similarly, search and deletion can be so combined. Furthermore ranks need not be stored in nodes, but can be computed as needed. However, for our efficiency analysis to hold, this approach requires the stronger independence assumption that the sequence of insertions and deletions is independent of the function generating the ranks.[3]

[3] This issue is not merely theoretical. Reuse of random seeds has led to real-world "denial-of-service" attacks for a number of programming libraries. See `http://ocert.org/advisories/ocert-2011-003.html`.

One additional nice feature of zip trees is that deletion does not require swapping a binary node for a leaf or unary node before deleting it, as in Hibbard deletion [4].

As compared to treaps, zip trees have an average height about 8% greater. By choosing the ranks using a geometric distribution with higher mean, we can reduce this discrepancy, at the cost of increasing the number of bits needed to represent the ranks. Whether this is worthwhile is a question for experimental study.

We believe that the properties of zip trees make them a good candidate for concurrent implementation. The third author developed a preliminary, lock-based implementation of concurrent zip trees in his senior thesis [12]. We plan to develop a non-blocking implementation.

5 Implementations

In this section we present pseudocode implementing zip tree insertion and deletion. We leave the implementation of search as an exercise. Our pseudocode assumes an endogenous representation (nodes are items), with each node x having a key $x.key$, a rank $x.rank$, and pointers to the left and right children $x.left$ and $x.right$ of x respectively. We give two implementations designed to achieve different goals.

Our first goal is to minimize lines of code. This we do by using recursion. Our recursive methods for insertion and deletion appear in Algorithm 1. Method `insert`$(x, root)$ inserts node x into the tree with root *root* and returns the root of the resulting tree. It requires that x not be in the initial tree. Once the last line of `insert` ("return *root*") in Algorithm 1 is reached, `insert` can actually return from the outermost call: all further tests will fail, and no additional assignments will be done. Method `delete`$(x, root)$ deletes node x from the tree with root *root* and returns the root of the resulting tree. It requires that x be in the initial tree. Unzipping is built into the insertion method; in deletion, zipping is done by the separate method `zip`(x, y), which zips the paths with top nodes x and y and returns the top node of the resulting path. It requires that all descendants of x have smaller key than all descendants of y.

Our second goal is to do updates completely top-down and to minimize pointer changes. This results in longer, less elegant, but more straightforward methods. We treat *root* as a global variable, with *root* = `null` indicating an empty tree. Method `insert`(x) in Algorithm 2 inserts node x into the tree with root *root*, assuming that x is not already in the tree. Method `delete`(x) in Algorithm 3 deletes node x from the tree with root *root*, assuming it is in the tree.

These methods do some redundant tests and assignments to local variables. These could be eliminated by loop unrolling, but might also be eliminated by a good optimizing compiler.

Algorithm 1: Recursive versions of insertion and deletion.

```
insert($x$, $root$):
  if $root = null$ then {$x.left \leftarrow x.right \leftarrow null$; $x.rank \leftarrow$ RandomRank; return $x$}
  if $x.key < root.key$ then
    if insert($x$, $root.left$) $= x$ then
      if $x.rank < root.rank$ then $root.left \leftarrow x$
      else {$root.left \leftarrow x.right$; $x.right \leftarrow root$; return $x$}
  else
    if insert($x$, $root.right$) $= x$ then
      if $x.rank \leq root.rank$ then $root.right \leftarrow x$
      else {$root.right \leftarrow x.left$; $x.left \leftarrow root$; return $x$}
  return $root$

zip($x$, $y$):
  if $x = null$ then return $y$
  if $y = null$ then return $x$
  if $x.rank < y.rank$ then {$y.left \leftarrow$ zip($x$, $y.left$); return $y$}
  else {$x.right \leftarrow$ zip($x.right$, $y$); return $x$}

delete($x$, $root$):
  if $x.key = root.key$ then return zip($root.left$, $root.right$)
  if $x.key < root.key$ then
    if $x.key = root.left.key$ then
      $root.left \leftarrow$ zip($root.left.left$, $root.left.right$)
    else delete($x$, $root.left$)
  else
    if $x.key = root.right.key$ then
      $root.right \leftarrow$ zip($root.right.left$, $root.right.right$)
    else delete($x$, $root.right$)
  return $root$
```

Algorithm 2: Iterative insertion.

```
insert($x$)
  $rank \leftarrow x.rank \leftarrow$ RandomRank
  $key \leftarrow x.key$
  $cur \leftarrow root$
  while $cur \neq$ null and $(rank < cur.rank$ or $(rank = cur.rank$ and $key > cur.key))$ do
    $prev \leftarrow cur$
    $cur \leftarrow$ if $key < cur.key$ then $cur.left$ else $cur.right$

  if $cur = root$ then $root \leftarrow x$
  else if $key < prev.key$ then $prev.left \leftarrow x$
  else $prev.right \leftarrow x$

  if $cur =$ null then $\{x.left \leftarrow x.right \leftarrow$ null; return$\}$
  if $key < cur.key$ then $x.right \leftarrow cur$ else $x.left \leftarrow cur$
  $prev \leftarrow x$

  while $cur \neq$ null do
    $fix \leftarrow prev$
    if $cur.key < key$ then
      repeat $\{prev \leftarrow cur;\ cur \leftarrow cur.right\}$
      until $cur =$ null or $cur.key > key$
    else
      repeat $\{prev \leftarrow cur;\ cur \leftarrow cur.left\}$
      until $cur =$ null or $cur.key < key$
    if $fix.key > key$ or $(fix = x$ and $prev.key > key)$ then
      $fix.left \leftarrow cur$
    else
      $fix.right \leftarrow cur$
```

Algorithm 3: Iterative deletion.

```
delete($x$)
    $key \leftarrow x.key$
    $cur \leftarrow root$
    while $key \neq cur.key$ do
        $prev \leftarrow cur$
        $cur \leftarrow$ if $key < cur.key$ then $cur.left$ else $cur.right$
    $left \leftarrow cur.left$; $right \leftarrow cur.right$

    if $left =$ null then $cur \leftarrow right$
    else if $right =$ null then $cur \leftarrow left$
    else if $left.rank \geq right.rank$ then $cur \leftarrow left$
    else $cur \leftarrow right$

    if $root = x$ then $root \leftarrow cur$
    else if $key < prev.key$ then $prev.left \leftarrow cur$
    else $prev.right \leftarrow cur$

    while $left \neq$ null and $right \neq$ null do
        if $left.rank \geq right.rank$ then
            repeat $\{prev \leftarrow left;\ left \leftarrow left.right\}$
            until $left =$ null or $left.rank < right.rank$
            $prev.right \leftarrow right$
        else
            repeat $\{prev \leftarrow right;\ right \leftarrow right.left\}$
            until $right =$ null or $left.rank \geq right.rank$
            $prev.left \leftarrow left$
```

Acknowledgments

We thank Dave Long for carefully reading the manuscript and offering many useful suggestions, most importantly helping us simplify the iterative insertion and deletion algorithms. We thank Sebastian Wild for correcting the bound on expected node depth in treaps in Section 2 and for his ideas on breaking rank ties. We thank Leon Sering for correcting the statement and proof of Lemma 2. Finally, we are grateful to Dominik Kempa for providing us with C++ zip-tree implementations, benchmarks, and general comments. Research at Princeton University by the first two authors was partially supported by an innovation research grant from Princeton and a gift from Microsoft.

References

[1] Herman Chernoff. “A measure of asymptotic efficiency for tests of a hypothesis based on the sum of observations”. In: *The Annals of Mathematical Statistics* 23.4 (1952), pp. 493–507.

[2] Brian Dean and Zachary Jones. “Exploring the duality between skip lists and binary search trees”. In: *ACM Southeast Regional Conference (ACMSE)*. Winston-Salem, North Carolina, United States: ACM Press, 2007, pp. 395–400.

[3] Bernhard Haeupler, Siddhartha Sen, and Robert Tarjan. “Rank-Balanced Trees”. In: *ACM Transactions on Algorithms* 11.4 (2015), pp. 1–26.

[4] Thomas Hibbard. “Some Combinatorial Properties of Certain Trees With Applications to Searching and Sorting”. In: *Journal of the ACM* 9.1 (1962), pp. 13–28.

[5] Conrado Martínez and Salvador Roura. “Randomized binary search trees”. In: *Journal of the ACM* 45.2 (1998), pp. 288–323.

[6] Helmut Prodinger. “Combinatorics of geometrically distributed random variables: left-to-right maxima”. In: *Discrete Mathematics* 153.1-3 (1996), pp. 253–270.

[7] William Pugh. “Skip lists: a probabilistic alternative to balanced trees”. In: *Communications of the ACM* 33.6 (1990), pp. 668–676.

[8] Raimund Seidel and Cecilia Aragon. “Randomized search trees”. In: *Algorithmica* 16.4-5 (1996), pp. 464–497.

[9] Renzo Sprugnoli. “Randomly balanced binary trees”. In: *Calcolo* 17.2 (1980), pp. 99–117.

[10] C. J. Stephenson. “A method for constructing binary search trees by making insertions at the root”. In: *International Journal of Computer & Information Sciences* 9.1 (1980), pp. 15–29.

[11] Robert Tarjan. *Data Structures and Network Algorithms*. Philidelphia, Pennsylvania, United States: Society for Industrial and Applied Mathematics, 1983.

[12] Stephen Timmel. “Zip Trees: A new approach to concurrent binary search trees”. Senior Thesis, Department of Mathematics, Princeton University. https://arks.princeton.edu/ark:/88435/dsp01gh93h214f. 2017.

[13] Jean Vuillemin. “A unifying look at data structures”. In: *Communications of the ACM* 23.4 (1980), pp. 229–239.