Zip Trees*

Robert E. Tarjan[†] Caleb C. Levy[‡] Stephen Timmel[§]

August 7, 2018

Abstract

We introduce the *zip tree*,[1] a form of randomized binary search tree. One can view a zip tree as a treap [8] in which priority ties are allowed and in which insertions and deletions are done by unmerging and merging paths (*unzipping* and *zipping*) rather than by doing rotations. Alternatively, one can view a zip tree as a binary-tree representation of a skip list [7]. Doing insertions and deletions by unzipping and zipping instead of by doing rotations avoids some pointer changes and can thereby improve efficiency. Representing a skip list as a binary tree avoids the need for nodes of different sizes and can speed up searches and updates. Zip trees are at least as simple as treaps and skip lists but offer improved efficiency. Their simplicity makes them especially amenable to concurrent operations.

1 Definition of Zip Trees

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.

*Research at Princeton University by the first two authors partially supported by an innovation research grant from Princeton and a gift from Microsoft.

[†]Department of Computer Science, Princeton University, and Intertrust Technologies; ret@cs.princeton.edu.

[‡]Program in Applied and Computational Mathematics, Princeton University, and Intertrust Technologies; cclevy@princeton.edu.

[§]Mathematics Department, Virginia Polytechnic Institute and State University; stimmel@vt.edu

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

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 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 tree is the path from the root to the node of smallest (resp. largest) key. The left (resp. right) spine of x contains only the root and left (resp. right) children. 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, $x.left = \texttt{null}$ (resp. $x.right = \texttt{null}$).

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 all nodes with keys less than $x.key$ and a path Q containing all nodes with keys 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, do a search to find it. Let P and Q 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-decreasing rank order, breaking ties in favor of smaller keys. 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

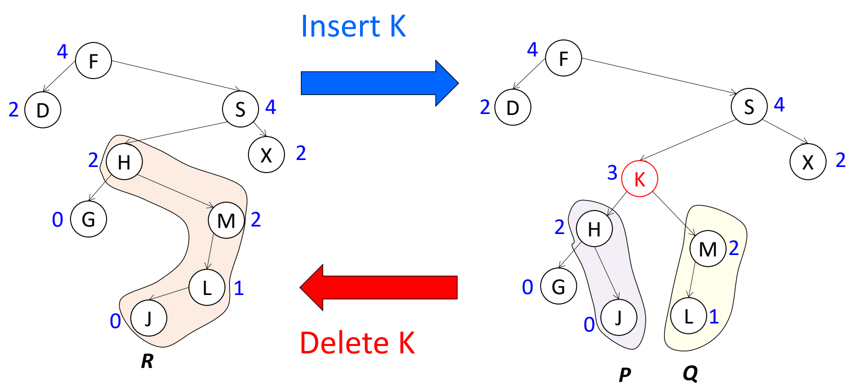

Figure 1: Insertion and deletion of a node with key “K” assigned rank 3.

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 demonstrates both insertion and 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.

In the subsequent sections of this paper we present the intuition behind zip trees (Section 2), examine the properties of zip trees (Section 3), compare them to previous data structures (Section 4), discuss some variants (Section 5), address some implementation details (Section 6), and offer a few remarks (Section 7).

2 Intuition Behind Zip Trees

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.

3 Properties of Zip Trees

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. *The structure of a zip tree is uniquely determined by the keys and ranks of its nodes.*

Proof. We use induction on n. The lemma is immediate if $n = 0$ or $n = 1$. Suppose $n > 1$. The root is the node of minimum key among those with maximum rank. The root uniquely partitions the remaining nodes into those in its left subtree and those in its right subtree. These are uniquely determined by the induction hypothesis. □

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. We can eliminate the independence assumption at the cost of constant factors in time and space by doing lazy deletions, as we discuss in Section 5. If the rank of a node is a pseudorandom function of the node, we need the following stronger independence assumption: the sequence of insertions and deletions is independent of the function generating the ranks.

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 $1/2^{c \lg n} = 1/n^c$. □

Theorem 2. *The expected depth of a node in a zip tree is at most* $(3/2)\lg n + 3$. *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. Let x be a node. We bound separately the *low ancestors* of x, those with keys less than that of x, and the *high ancestors* of x, those with keys greater than that of x. A node y is a low ancestor of x iff $y.key \leq x.key$ and $y.rank \geq z.rank$ for all nodes z between y and x in key order, inclusive. If y is a low ancestor of x, the next low ancestor of x in decreasing key order, if any, is the node y' of maximum key smaller than $y.key$ such that $y'.rank \geq y.rank$.

We can think of the low ancestors of x and their ranks as being generated by coin flips in the following way. At x we flip a fair coin until it comes up tails and give x a rank equal to the number of heads. All these flips are *relevant*. At each successive node y 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. If the next low ancestor of x after y in key order is z, the first $z.rank$ of the flips at y are *irrelevant* and the rest, if any, are *relevant*. Node y is a low ancestor of x if and only if there are least $z.rank + 1$ flips at y; if so, there is one tail and $y.rank - z.rank$ heads among the relevant flips at y. We continue flipping until every y such that $y.key \leq x.key$ has a rank.

Among the low ancestors of x, the one of smallest key has maximum rank. This rank is equal to the total number of relevant heads, which is at most the root rank. The number of low ancestors of x is equal to the total number of relevant tails. Since the expected number of relevant tails is equal to the expected number of relevant heads, the expected number of low ancestors of x is at most the expected root rank. This is at most $\lg n + 3$ by Theorem 1. Furthermore, by a Chernoff bound [1], if the root rank is at most $(c + 2)\lg n$, then the number of low ancestors of x exceeds $a\,c \lg n$ with probability at most $1/n^{c+3}$, where a is a positive constant independent of c.

We bound the number of high ancestors of x similarly, the only difference being that there are no rank ties among high ancestors: a node y is a high ancestor of x iff $y = x$, or $y.key > x.key$ and $y.rank > z.rank$ for all z between x and y in key order, including x but not y. If y is a high ancestor of x, the next high ancestor of x in increasing key order, if any, is the node y' of minimum key larger than $y.key$ such that $y'.rank > y.rank$.

We think of generating the high ancestors of x and their ranks just like the low ancestors, except that we consider nodes in increasing key order instead of decreasing key order and we call a flip relevant only if it increases the rank of a high ancestor beyond its minimum value (the rank of the preceding high ancestor plus one). The number of high ancestors of x is equal to the total number of relevant tails. Since the expected number of relevant tails is half the expected number of relevant flips, the expected number of high ancestors of x is at most half the expected root rank plus one half. This is at most $(1/2)\lg n + 2$ by Theorem 1. By a Chernoff bound, if the root rank is at most $(c + 2)\lg n$, then the number of low ancestors of x exceeds $b\,c \lg n$ with probability at most $1/n^{c+3}$, where b is a positive constant independent of c.

Combining the expected bounds on low and high ancestors and subtracting two to avoid counting x (twice) gives the first half of the theorem. Combining the high probability bounds, the probability that the depth of a given node x exceeds $(a + b)c \lg n$ is at most $2/n^{c+3}$, given that the root rank is at most

$(c+2)\lg n$. By Theorem 1, the probability that the root rank exceeds $(c+2)\lg n$ is at most $1/n^{c+2}$. Since there are $n > 1$ nodes, the probability that some node depth exceeds $(a+b)c\lg n$ is at most $2/n^{c+2} + 1/n^{c+1} \le 1/n^c$, giving the second half of the theorem. $\square$

Remark 1. Rather than proceeding from scratch, one can prove Theorem 2 using results from [6].

By Theorem 2, the expected number of nodes visited during a search in a zip tree is at most $(3/2)\lg n + 4$, 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 $(3/2)k + 1/2$. There is a constant $c > 1$ such that this number is $O(k)$ with probability at least $1 - 1/(2c^k)$.*

Proof. The proof is like that of Theorem 2. 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, after it is inserted. Thus we need only consider the case of a 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 x' and x'' be the predecessor and successor of x in key order, respectively. The nodes on P are exactly the low ancestors of x' that have rank less than $x.rank$; the nodes on Q are exactly the high ancestors of x'' that have rank at most $x.rank$.

We can think of the nodes on P and their ranks as being generated by coin flips in the following way. At x' we flip a fair coin $x.rank$ times or until it comes up tails. If all the flips are heads we stop and make P empty. Otherwise we add x' to P and give it a rank equal to the number of heads. In either case all flips are *relevant*. At each successive node y in decreasing key order we flip a fair coin $x.rank$ times or until it comes up tails. If all the flips are heads we stop: P is complete. Otherwise we give y a rank equal to the number of heads and add y to P if its rank is no less than the rank of the last node z added to P. In either case the outcomes of the first $z.rank$ flips at y are *irrelevant* and the rest are *relevant*. We continue until we get k heads in a row or we process the smallest node in key order.

The relevant flips include at most k heads. The number of nodes on P equals the total number of relevant tails. Since the expected number of relevant tails is equal to the expected number of relevant heads, the expected number of nodes on P is at most k. By a Chernoff bound, there is a constant $d > 1$ such that the number of nodes on P is $O(k)$ with probability at least $1 - 1/(4d^k)$.

Symmetrically, we think of the nodes of Q and their ranks as being generated by coin flips in increasing key order, calling a flip relevant only if it increases the rank of a node on Q beyond its minimum value (the rank of the preceding node on Q plus one). The number of relevant flips is at most $k+1$. The number of nodes on Q equals the total number of relevant tails. Since the expected number of relevant tails is half the expected number of relevant flips, the expected number of nodes on Q is at most $(k+1)/2$. By a Chernoff bound, there is a constant $g > 1$ such that the number of nodes on Q is $O(k)$ with probability at least $1 - 1/(4g^k)$.

Combining the expected bounds on the numbers of nodes on P and Q gives the first half of the theorem; combining the two Chernoff bounds gives the second half, with $c = \min\{d, g\}$. $\square$

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 pointers is at most $1/c^k$ for some $c > 0$.*

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 $(3/2)k + 1/2$ by Theorem 1. Summing over all possible values of k gives the first half of the theorem.

By the second half of Theorem 3, there are constants $a > 1$ and $b > 0$ such that if the rank of x is at most $k/(ab)$, then the probability that the insertion or deletion of x changes at most k pointers is at most $1/(2d^k)$, where $d = a^{1/(ab)}$. The probability that the rank of x exceeds $k/(ab)$ is at most $2^{k/(ab)+1} = 1/(2g^k)$, where $g = 2^{1/(ab)}$. The probability that the insertion or deletion of x changes more than k pointers is thus at most $1/(\min\{d, g\})^k$, giving the second half of the theorem. $\square$

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 $(3/2)\lg n + 3$.*

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. 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 2^k, which means that 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 $p = 2^{k+1}$. The probability that the i^{th} node is the first of rank greater than k is $(1 - 2^{k+1})^{i-1}/2k + 1$. 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 $(3/2)\lg n + 3$. $\square$

4 Previous 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]. 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 rank ties 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 child while preserving symmetric order. In general a rotation changes three children. 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 (or at its only child if it has only one child). Once x is a leaf, we replace it by a missing node.

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 $\lg\lg n + O(1)$, 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 $\ln n$, about $1.44 \lg n$, 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 4% in average depth (and search time) but allows much more compact representation of priorities. In Section 5 we show how to break rank ties in zip trees by using fractional ranks, while preserving an $O(\lg\lg n)$ high-probability bound on the number of bits needed to represent ranks.

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 leaves the bottom of the tree), with each node having a priority equal

[2]Seidel and Aragon [8] hinted 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.

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.

Martinez 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.

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 p). Each list is in increasing order by key. 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, 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. To guarantee that backing up is always possible, all the lists contain a dummy item whose key is less than all others.

One can view a zip tree as a compact representation of a skip list. To convert a zip tree to the corresponding skip list, add a dummy node of rank infinity and key smaller than all others. Let k be the maximum finite rank. For each rank i from 0 to k, construct a list containing all items in nodes of rank i or larger. To convert a skip list to the corresponding zip tree, give each item a rank equal to the level of the highest-level list containing it. The parent of a node x is either the node y of greatest key less than $x.key$ having rank at least $x.rank$, or the node z of smallest key greater than $x.key$ having rank greater than $x.rank$, whichever has smaller rank, with a tie broken in favor of z. If one of y and z does not exist, the other is the parent of x; if neither exists, x is the root.

The mapping that converts a skip list into a binary search tree was given by Dean and Jones [2], except that they store ranks in the binary search tree in difference form. They also mapped the insertion and deletion algorithms for a skip list into algorithms on the corresponding binary search tree, but their algorithms do rotations rather than unzipping and zipping.

A search in a zip tree visits the same items as the search in the corresponding 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 corresponding 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.

5 Variants

We discuss three variants of zip trees, two of them useful, one not.

The first variant uses fractional ranks to reduce the frequency of rank ties, thereby reducing the expected depth by about 4% to that of treaps. To do this we allow fractional ranks. A rank is of the form $k + f$, where k is an integer chosen as in Section 1 and f is a rational number chosen uniformly at random from a finite sample of $[0, 1)$. We can choose f for a node x all at once when x is inserted, or choose it bit by bit as needed to break rank ties. The question is how many bits of precision each f needs to make rank ties on insertion unlikely. Suppose we insert x with integer rank k. We can generate the nodes of rank k that x might encounter on insertion by a coin-flipping process like those in the proofs of Theorems 2 and 3. Consider each node y of key greater than that of x in increasing key order. For each, generate its rank by coin-flipping. Once a node has a rank greater than $x.rank$, stop. For a node y to have rank k given that it has rank at least k, its $k+1^{\text{st}}$ flip must be a tail, which happens with probability $1/2$. The expected number of ties is thus at most 1, and is $O(c \log n)$ with probability at most $1 - 1/n^c$. The same is true for nodes of key less than that of x. We conclude that with high probability $O(\log \log n)$ bits of precision in f suffice to break all rank ties during all insertions.

The second variant, due to Aragon and Seidel [8], uses recomputation of ranks to bias the tree so that frequently accessed items are faster to access than infrequently accessed ones: when accessing a node, we compute a new rank for it, and set its rank equal to the maximum of the old and new values. When the rank of a node increases, rotate at the node until heap order is restored. With this method, the expected time to access an item is $O(\log(1/p))$, where p is its empirical access probability, namely the number of times it is accessed divided by the total number of accesses. The probability of doing a rotation as a result of a rank increase is exponentially small in the rank, so the expected total restructuring time due to all accesses of a given item is $O(1)$. For further details see [8]. This variant has a weaker history-independence property: the tree structure depends only on the number of accesses to each node currently in the tree.

The third variant changes the probability distribution used to generate (the integer part of) ranks: we give a new node a rank of k with probability $p^k(1-p)$, for some fixed p strictly between 0 and 1. This is equivalent to the corresponding generalization of skip lists. As p increases, the expected maximum rank increases but the expected number of rank ties decreases, reducing the expected tree depth. Increasing p penalizes skip lists by increasing the expected node size and the number of pointers, but it penalizes zip trees much less, since only the number of bits needed to represent the ranks grows. Nevertheless, the use of fractional ranks eliminates rank ties much more efficiently than increasing p. Thus we see no reason to choose a value of p other than $1/2$.

6 Implementation

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 Algorithms 1 and 2. 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. 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 in Algorithm 3. Method `zip` 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.

Remark 2. Once the last line of `insert` ("return *root*") is reached, `insert` can actually return from the outermost call: all further tests will fail, and no additional assignments will be done.

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 4 inserts node x into the tree with root *root*, assuming that x is not already in the tree. Method `delete`(x) in Algorithm 5 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 Insertion

function `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 $root$

Algorithm 2: Recursive Deletion

function `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 3: Recursive Zip

function `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}

Algorithm 4: Iterative Insertion

function $\texttt{insert}(x)$

$rank \leftarrow x.rank \leftarrow$ `RandomRank`
$key \leftarrow x.key$
$cur \leftarrow x.root$
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 5: Iterative Deletion

```
function delete(x)
    key ← x.key
    cur ← root
    while key ≠ cur.key do
        prev ← cur
        cur ← if key < cur.key then cur.left else cur.right
    left ← cur.left; right ← cur.right

    if left = null then cur ← right
    else if right = null then cur ← left
    else if left.rank ≥ right.rank then cur ← left
    else cur ← right

    if root = x then root ← cur
    else if key < prev.key then prev.left ← cur
    else prev.right ← cur

    while left ≠ null and right ≠ null do
        if left.rank ≥ right.rank then
            repeat {prev ← left; left ← left.right}
            until left = null or left.rank < right.rank
            prev.right ← right
        else
            repeat {prev ← right; right ← right.left}
            until right = null or left.rank ≥ right.rank
            prev.left ← left
```

7 Remarks

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 sense [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 roots 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.

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

The properties of a zip tree make it 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 are currently developing a non-blocking implementation.

Acknowledgements

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 in Section 6.

References

[1] H. Chernoff. A Measure of Asymptotic Efficiency for Tests of a Hypothesis Based on the sum of Observations. *The Annals of Mathematical Statistics*, 23(4):493–507, 1952.

[2] B. C. Dean and Z. H. Jones. Exploring the duality between skip lists and binary search trees. In *Proceedings of the 45th Annual Southeast Regional Conference*, pages 395–400. ACM Press, 2007.

[3] B. Haeupler, S. Sen, and R. E. Tarjan. Rank-Balanced Trees. *ACM Transactions on Algorithms*, 11(4):1–26, 2015.

[4] T. N. Hibbard. Some Combinatorial Properties of Certain Trees With Applications to Searching and Sorting. *Journal of the ACM*, 9(1):13–28, 1962.

[5] C. Martínez and S. Roura. Randomized binary search trees. *Journal of the ACM*, 45(2):288–323, 1998.

[6] H. Prodinger. Combinatorics of geometrically distributed random variables: Left-to-right maxima. *Discrete Mathematics*, 153(1-3):253–270, 1996.

[7] W. Pugh. Skip lists: a probabilistic alternative to balanced trees. *Communications of the ACM*, 33(6):668–676, 1990.

[8] R. Seidel and C. R. Aragon. Randomized search trees. *Algorithmica*, 16(4-5):464–497, 1996.

[9] R. Sprugnoli. Randomly balanced binary trees. *Calcolo*, 17(2):99–117, 1980.

[10] C. J. Stephenson. A method for constructing binary search trees by making insertions at the root. *International Journal of Computer & Information Sciences*, 9(1):15–29, 1980.

[11] R. E. Tarjan. *Data Structures and Network Algorithms*. Society for Industrial and Applied Mathematics, 1983.

[12] S. Timmel. Zip Trees: A new approach to concurrent binary search trees. Senior Thesis, Department of Mathematics, Princeton University, 2017.

[13] J. Vuillemin. A unifying look at data structures. *Communications of the ACM*, 23(4):229–239, 1980.