Zip Trees: A New Approach to Concurrent Binary Search Trees

Stephen Timmel

Advisor: Professor Robert Tarjan

Submitted in partial fulfillment

of the requirements for a Bachelor's degree

Department of Mathematics

Princeton University

May 2017

Abstract

One of the most fundamental tasks in computing is the storage and retrieval of data. For this reason, binary search trees are one of the most fundamental algorithms in computer science. In the last few years, the ever-present need for data access has been accompanied by a need for multiple processors to access data concurrently. Concurrent programming is a deeply theoretical field of study focused on developing algorithms and techniques for managing this increased interest in parallel processing. Binary search trees typically present three main difficulties in a concurrent environment: their reliance on global information and balance, their use of read and write operations which traverse the tree in different directions, and the amplified cost of changing pointers near the root (where many threads must traverese). We develop a new algorithm known as a Zip Tree which overcomes all three challenges by maintaining balance based solely on locally stored random values, updating the tree structure from the top down, and inserting at a height constant in expectation within the tree.

Contents

Abstract iii

1 Introduction **1**

2 Zip Trees **7**

3 An Aside on Randomization **25**

3.1 An Aside on Randomization 25

4 A Comparison to Skip Lists **32**

5 The Performance of Zip Trees **40**

6 Conclusion **45**

6.1 conclusion 45

A Reference Zip Tree Code **47**

A.1 zip tree 47

A.2 test program 51

A.3 concurrent zip tree 52

A.4 concurrent test program 59

Chapter 1

Introduction

As computing projects have steadily increased in scope, the need for concurrency in programming environments has grown. Many large-scale projects require large numbers of independent computations, which may often be carried out by separate processors for large savings in execution time. However, the inherent non-determinism in this sort of parallel execution creates a challenging design problem. The validity of a concurrently executed program depends at some level on a careful isolation of operations, while communication and shared access to data is often essential for efficient programming. This tradeoff between ease of theoretical validation and operational efficiency have come to define a surprisingly deep theoretical field, fueled by a continued drive to produce efficient concurrent processes.
The most basic approach to this problem has been to design concurrent primitives, with the goal of ensuring the validity of any traditional algorithm in a concurrent environment. In a discussion of an atomic shared memory protocol, Herlihy describes the goal of this approach by stating that "a practical methodology should permit a programmer to design, say, a correct lock-free priority queue, without ending up with a publishable result." (1993) The simplest of these primitives are formulated as aids, designed to provide practical atomic operations for incorporation into larger struc-

tures. Spin-locks were introduced by Mellor-Crummy in 1991 to provide programmers with a signalling mechanism to allocate shared data among processes. Other primitives include CAS, an atomic operation that conditionally changes memory values, and LL/SC, an atomic read-write operation. (Gao, 2007) More ambitious approaches attempt to create an underlying global framework with general guarantees such as lock-freedom, which guarantees that at least one process will always make progress, and wait-freedom, which guarantees that every process will make progress. One popular framework is Software Transactional Memory, which forces processes to verify the integrity of shared input before modifying shared memory. (Shavit, 1997) Although Fatourou has suggested an optimized wait-free algorithm based on STM, both lock-free and wait-free algorithms generally require substantial overhead. (2011) Additionally, these primitives will not guarantee efficient operation of a poor underlying algorithm. Thus, despite the generality of concurrent primitives, there is still a fundamental need for individual algorithms which naturally support concurrency. The creation of algorithms that inherently support parallelization leads to more efficient programming and reduces the need for general primitives with high overhead. Binary search trees are fundamentally important as an algorithm, and especially essential in large distributed environments. Simple storage and fast retrieval of relational data is a necessary step in data processing and access. The traditional strategy in a sequential environment is to represent data by key-value pairs with some total ordering of keys and to place data in a tree, where every node has a left child with a smaller key and a right node with a larger key. A balanced tree is one where all paths from the root have the same length (up to at most a small constant factor), in which case the tree has height $\log_2(n)$. Thus, in a balanced search tree, any search from the root will terminate in $O(\log(n))$ time, where n is the number of nodes in the tree. Since its development, many strategies have been proposed to maintain the balance of a search tree if elements are added and removed dynamically. Popular approaches include Red-Black trees (Guibas et al., 1978) and AVL trees (Adelson-Velskii et al., 1962), which use a sequence of adjustments in parent-child relationship known as rebalancing operations to balance the height of a tree after insertion (red-black trees and AVL trees use a similar approach for deletion). Since concurrent programming typically requires operating on large data sets, efficient data storage remains ciritcal in concurrent programming. However, the translation of sequential search trees into efficient parallel algorithms is a non-trivial task.

The most natural approach to generating a concurrent binary search tree stems from the direct translation of a sequential algorithm. Ellis pioneered this approach by developing a concurrent AVL tree. (1980) Later, Varshneya applied the same style of extension to k-dimensional height-balanced trees (1994) and Park developed a concurrent Red-Black tree (2001). All three extensions are based on their respective sequential algorithms and achieve concurrency by using a sequence of locks to isolate nodes during critical operations. However, these extensions are complicated by the fact that most sequential tree algorithms perform accesses down from the root and rebalance up from the leaves. This combination naturally risks dead-lock, where an access operation travelling from the root meets an update travelling up from a leaf and neither can terminate. Avoiding this sort of deadlock complicates extensions to sequential search algorithms, requiring that the algorithm either lock a substantial portion of the tree or routinely prioritize one operation over another. These drawbacks suggest that an algorithm more closely tailored to a concurrent environment might offer important advantages to a naïve extension of a classic algorithm.

Manber suggested a simple approach to use concurrency to solve the problem of rebalancing, which has since been widely embraced in the literature. (1984) Manber created a variety of tree where insertion and deletion operations performed no direct rebalancing. Instead, newly inserted nodes were added to a queue, and a separate maintenance thread applied rebalancing operations to asynchronously rebalance the tree height. This basic approach has been repurposed since Manber's initial publication in the form of chromatic trees (Nurmi et al., 1996), hyperred-black trees (Gabarró et al., 1997), and relaxed AVL trees (Larsen, 2000). In all cases, the primary issue addressed is that of performance, and all testing is done on a fixed architecture with randomized sequences of operations. However, none of these algorithms provides any theoretical guarantee regarding the height of the resulting tree. An operating system with an unfair scheduler could starve the maintenance thread of resources. In the worst case, a long sequence of insertions in sequential order followed by near-constant accesses could create a tree with near-linear height, as every operation traverses the same path down the tree and every update adds a new node to the end of the path. This theoretical failure calls for a more innovative approach that provides both theoretical guarantees and empirical efficiency.

One approach to this dual problem stems from an entirely separate data structure known as skip lists. First developed in 1990 by Pugh, skip lists are based directly on linked lists. All nodes are assigned random values on insertion, called height. The data structure consists of a single list in sorted order. For every height k, edges are added to connect adjacent elements in the sublist containing only nodes of height at least k. Access proceeds by traversing sublists in decreasing order by height. For each height k, a search halts when the next node accessed would be too large and continues in the same place at height $k - 1$. In this manner, a search for element a finds first the next smallest node at each height, ultimately terminating at the height of a (or 0 on failure). Insertion assigns a node's height and adds it to the list in sorted order. At each height less than the new node, the node and its predecessor are assigned new successors as in a linked list. Deletion removes a node and assignes a new successor to its predecessor at each height, as in a linked list. Pugh demonstrated that all skip list operations probabilistically terminate in $O(\log(n))$ steps, giving a similar average bound to binary search trees. Following this development, a series of other papers added new properties to Pugh's initial creation. Pugh's probabilistic bounds were improved in a deterministic skip list, which changed node heights on insertion to ensure that exactly $\log_2(k)$ nodes were assigned height k. (Munro et al., 1992) Lamoureux and Nickerson demonstrated a k-dimensional skiplist, emulating binary search trees designed to order keys in multiple dimensions for fast retrieval. (2005) Herlihy et al developed a natural concurrent extension to skip lists, where insert and delete operations acquired a lock before updating each edge and edges were updated in decreasing order by height. (2006) However, the most intriguing discussion of skip lists stems from the development of skip trees by Messeguer. (1997) By demonstrating an isomorphism between skip lists and B-trees, Messeguer indicated that the two data structures are not entirely distinct, and that developments in one field might be mapped directly to the other. This suggests that a clever mapping might produce a single data structure that is natural and efficient both as a search tree and as a skip list.

The potential for such a mapping between skip lists and search trees suggests that the best approach might be a carefully designed binary search tree. Afek et al. proposed a CB (count-based) tree as a possible approach to this problem. (2014) This structure, based on the splay tree, rebalances on accessing instead of updating. This approach, while excellent, is not necessarily suitable for all applications. Many use cases require that access operations be much more frequent than updates, and accessing elements generally need not modify the tree structure (unlike insertion and deletion). Thus, CB trees have certain limitations which might be addressed in a new algorithm. A very promising new approach stems from a randomized treap algorithm designed by Seidel and Aragon. (1996) Randomized treaps forgo global balance information in favor of a random value attached to each node. Update operations seek to maintain a heap ordering among the random bits and a tree ordering among the keys. This approach probabilistically guarantees that every insertion will occur near the bottom of the tree, minimizing rotations near the root node. However, Seidel and Aragon's algorithm uses rotations which propagate up from the leaves, which complicates concurrency. This problem is neatly fixed in a discussion by Martínez and Roura on randomized search trees, which suggests foregoing rebalancing rotations in favor of subtree disassembly. (1998) These two ideas, randomized local ordering and subtree disassembly propagating down the tree, form the crux of a powerful new algorithm proposed here, dubbed the zip tree to honor its rebalancing scheme. (Tarjan, 2016)

Chapter 2

Zip Trees

Zip Trees represent a novel combination of several existing techniques to produce a tree structure suitable for concurrent operations. The first and most fundamental of these techniques is the tree structure developed by Seidel and Aragon, which forms the foundation for the basic structure of zip trees. (1996) Seidel and Aragon suggest a localized tree structure ordered using random numbers. The basic design is that of a treap, where each node contains two distinct values k, h from different ordered sets K, H (known as the key and priority). The tree is then constructed to satisfy two simultaneous properties. Any node's key is greater than any descendant to its left and smaller than any descendant to its right. Further, any node's priority is greater than any of its descendants. We call the first ordering the tree ordering, and the second ordering the heap ordering. Figure 1 illustrates the two orderings graphically.

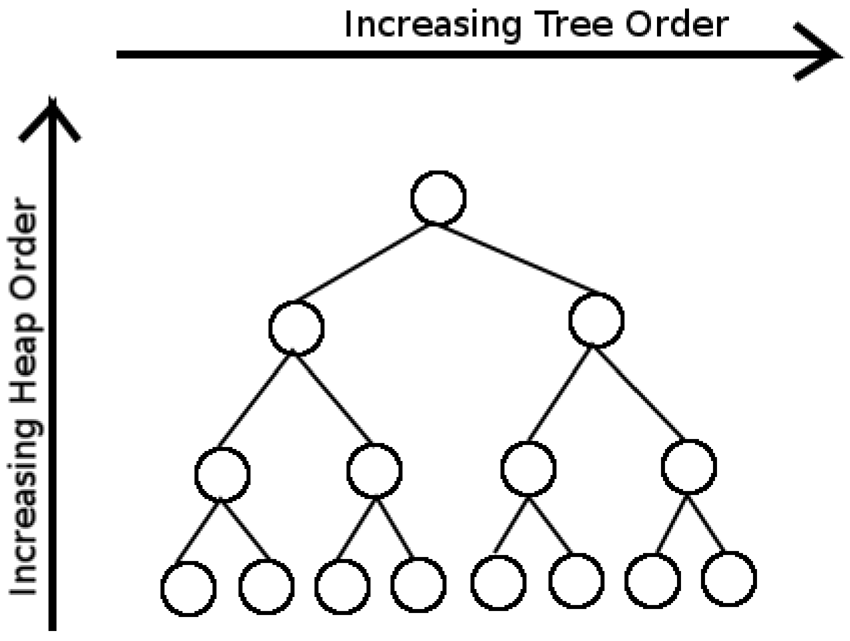

Figure 1. A visualization of the orderings in a treap

Next, we prove an important lemma which allows us to meaningfully use this construction.

Theorem 1. For any finite collection of nodes n_i each with distinct values $h \in H,\ k \in K$ selected from totally ordered sets, the two orderings defined above uniquely define a binary search tree containing the n_i

Pf. We proceed by constructing a tree containing the n_i which satisfies the given orderings. To begin, the heap ordering requires that a node's priority be greater than its descendants, so the root of the tree must be the node n_h with the largest priority. Next, we recall that the left subtree of n_h must have smaller keys than n_h, and the right descendants must have larger keys. Therefore, we partition the keys into left and right descendants on this basis. We repeat this process at each subtree, first by selecting a root to satisfy the heap ordering, then by partitioning subtrees to satisfy

the tree ordering. This construction generates a tree satisfying the desired properties, so we know that such a tree exists. Further, every step of our construction was uniquely constrained, so the final tree is uniquely determined by the nodes n_i. ■
The recursive pattern described above is illustrated below in a diagram

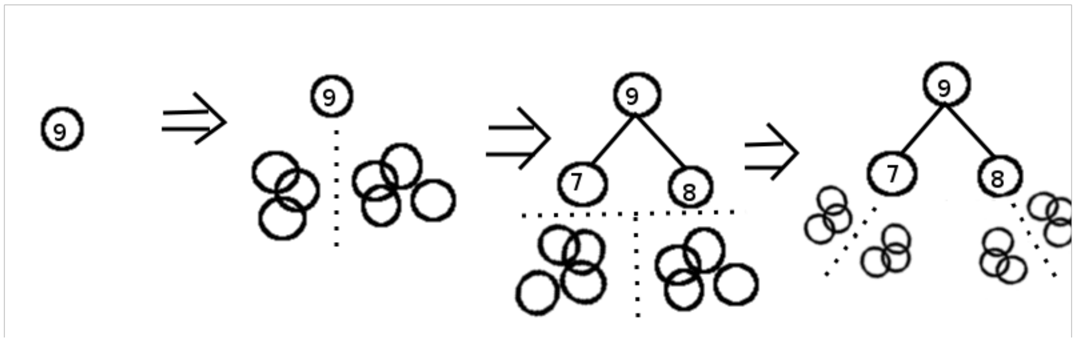

Figure 2. A diagram illustrating the recursive construction of a static zip tree structure

With the validity of this basic structure firmly established, we may begin to add some detail. Zip Trees maintain a tree ordering using keys provided by the user and a heap ordering using random numbers hidden from the user. In a significant departure from Seidel and Aragon's tree, our heap ordering does not use uniformly random numbers. (Tarjan, 2016) Instead, we use a geometric random variable which attains the value k with probability $\frac{1}{2^k}$. Intuitively, this change is intended to match our random value to a node's approximate height in the tree. More practically, we achieve the same amortized bound for random bits per node as Seidel and Aragon, but in a more direct fashion. If we generate our biased random numbers by counting the number of random bits needed to return a 1, the number of random bits required is precisely our biased random number h. We may compute the expected number of random bits per node to be

$$Ex[b] = \sum_{x=1}^{\infty} xPr[x] = \sum_{x=1}^{\infty} x\frac{1}{2^x} = 2$$

Since we need to store the value k, rather than the random values, we may calculate the maximum storage in any node to be $\log(\log(N))$ bits (a slight savings over existing algorithms). More importantly, this algorithm does not require that node weights be updated dynamically, which represents a substantial savings in programming complexity.

This change in randomization brings important advantages for the end user, but requires some initial work to validate. To begin, recall that our initial proof that treaps allow a unique representation required distinct priorities. In general, treaps with duplicate keys do not have a unique representation, as may be seen by considering a tree where every node contains the same priority. Therefore, we extend the treap structure to be "left-leaning", meaning that when we select among nodes with the same priority we always select the node with the largest key. Thus, duplicate priorities will always be left children in the heap ordering. We summarize this result in a generalized form of the first theorem.

Theorem 2. A left-leaning treap has a unique representation over any finite set of nodes n_i with distinct keys.

Pf. We proceed using a similar construction to Theorem 1. To begin, we recall that the root of a left-leaning treap must be the largest in the heap ordering. Therefore, we consider the subset of nodes with the largest priority and further select the node in this set with the largest key to be the root of our tree. This choice is unique, since we required that all keys be distinct. Next, we partition the remaining nodes into those with keys greater than the root, and nodes with keys less than the root. This partitioning is required to satisfy the tree ordering, and unique because the keys are distinct. Finally, we repeat this process recursively at each subtree, selecting root nodes to satisfy the heap ordering and partitioning nodes to satisfy the tree ordering. Our construction demonstrates that a tree structure satisfying both orderings exists, and each step was uniquely determined by the two orderings, which demonstrates uniqueness. ■

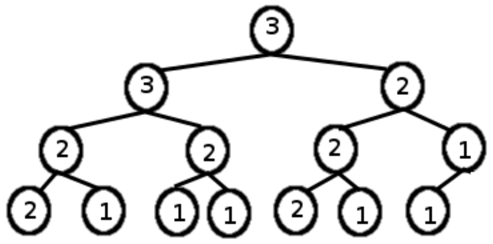

Figure 3. A depiction of ranks in a left-leaning zip tree

This discussion demonstrates that the data structure underlying zip trees is well-defined, so we continue by examining the height and balance of this structure. We must begin this analysis by assuming that the user has no knowledge of the random values used in the heap ordering. The need for this assumption is clear, since we may only assume that the heap ordering is random if the user cannot selectively insert or delete nodes priority. With this caveat, we proceed by developing our understanding of the expected height of the tree. We first denote by n_i the ith smallest node in an ordering by key. We proceed by considering properties of the ancestors of a node as an important first step in determining the height of our treap.

Lemma 3. Let $i < j$. The node n_j in a zip tree is an ancestor of node n_i iff no node n_k $i \leq k < j$ has priority strictly greater than n_j.

Pf. Consider the algorithm we used to generate left-leaning treaps in theorem 2. Suppose none of the n_k has priority strictly greater than n_j. Then every node that our algorithm selects as a root before n_j cannot be in the set $\{n_k \mid i \leq k < j\}$. Therefore, each partitioning of nodes by key includes the entire set $\{n_k \mid i \leq k < j\}$ in one partition, since the n_i are ordered by key. Therefore, when n_j is chosen to be the root of some subtree, the entire interval $\{n_k \mid i \leq k < j\}$ is partitioned as left children of n_j, and n_j is an ancestor of n_i. Suppose instead that one of the n_k has priority strictly greater than n_j. Then, the algorithm described in Theorem 2 selects one of the n_k as a root before either n_j. If n_i is selected first, it is an ancestor of n_j (and therefore not a descendant). Otherwise, we have $i < k < j$, so n_i is a left child of n_k and n_j is a right child, meaning that n_j is not an ancestor of n_i. ■

Lemma 4. Let $j < i$ Then n_j is an ancestor of n_i iff no node n_k $i \leq k < j$ has priority greater than or equal to n_j.

Pf. For this proof, we proceed in much the same manner as in Lemma 3. If no node n_k $i \leq k < j$ has priority greater than or equal to n_j, then our recursive search for root nodes does not select any of the n_k before n_j. Therefore, any partitioning by key that occurs before n_j is chosen as a root node will place the n_k in the same partition (since the nodes are ordered by key). When n_j is selected as the root of a subtree, the n_k are all partitioned as right children of n_j, meaning that n_j is an ancestor of n_i (as one of the n_k). Next, suppose that one of the n_k has priority greater than or equal to n_j. In this case, our recursive algorithm will select one of the n_k before n_j as the root of a subtree. If n_i is selected first, the resulting circuit has n_i as an ancestor of n_j, so the converse is false. Likewise, if some other node n_k is selected first, our partitioning by tree order leaves n_i as a left child of n_k and n_j as a right child of n_k, so n_j is not an ancestor of n_i. ■

Next, we use these results to demonstrate a probabilistic bound for the height of the tree.

Theorem 5. The maximum expected path length from the root of a zip tree to a leaf is exactly $\frac{3}{2}\log(n) + o(C)$.

To prove this theorem, we rely on the properties of ancestor nodes determined in

Lemma 3 and Lemma 4. The depth of a leaf node n_i in a zip tree is precisely the number of nodes which are ancestors of n_i. If we consider separately the cases where $j > i$ and $i > j$, we may apply the two lemmas to characterize these ancestor nodes. In the case $i < j$, Lemma 3 states that n_j is an ancestor of n_i iff n_j has priority greater than or equal to all n_k $i \leq k < j$. In the case $i > j$, Lemma 4 states that n_j is an ancestor of n_i iff n_j has priority strictly greater than all nodes n_k $j < k \leq i$. Therefore, the expected depth of a leaf node in the tree is the expected number of nodes satisfying these conditions. These condition may be reformulated in a simpler form. If we divide our ordering of nodes into the two intervals $n_k 1 \leq k < i$ and $n_k i < k \leq n$, it is sufficient to scan the second array left to right, counting the number of weak maxima encountered, and then the first array right to left, counting the number of strict maxima encountered. The first value is precisely the number of ancestors n_j $j > i$ and the second is the number of ancestors n_j $j < i$, so this is precisely the previous formulation. However, this question was considered by Prodinger, who calculated the number of strict maxima found when scanning an array of n nodes as $o(\frac{1}{2}\log_2(n))$ and the number of weak maxima as $o(\log_2(n))$. (1996) If we consider $c = \frac{i}{n}$, this result tells us that the number of ancestors is given by

$$o(\frac{1}{2}\log_2(cn) + \log_2((1-c)n)) = o(\frac{3}{2}\log_2(n)) + \log_2(c) + \log_2(1-c)$$

Since we know that $0 < c \leq 1$, the final two terms are always negative. Thus, the overall expression is minimized when $\log_2(c)$ is small and constant (which occurs, for instance, at $i = \frac{n}{2}$), meaning that the expected depth of an arbitrary leaf node n_i in a zip tree is less than or equal to $o(\frac{3}{2}\log_2(n)$. This completes the proof, since no leaf node has expected depth greater than the desired bound, and many leaves have expected depth equal to our bound ■

It should be noted that the expected depth of a leaf is not the expected height of the tree, which is determined by the maximum such depth. However, for our purposes, the expected depth of a leaf will prove to be a more useful measure of complexity. Thus, we know that in a static sense, zip trees are well-defined data structures with a probabilistic balance. We continue by defining an efficient update operation which is more amenable to parallelization than that proposed by Seidel and Aragon. This operation is based heavily on an algorithm proposed by Martinéz and Roura, which replaced the classic rebalancing operations with subtree deconstruction. As we will see later, this approach will produce a much simpler concurrent algorithm. (1998) In lieu of defining insert and delete operations, we create split and join operations. (Tarjan, 2016) When attempting to insert or delete a node, we first locate the unique placement which satisfies the tree and heap ordering of zip trees. Whenever we wish to delete a node n with parent p, we apply a join operations to its two child subtrees and make the root of the resulting subtree the child of p. Likewise, when we insert a node n as a child of p, we apply a split operation to divide a single child subtree of p into left and right subtrees of n. The split and join operations are relatively straightforward processes. When joining two trees where any key in the first is smaller than all keys in the second, we deconstruct the left spine of one tree and the right spine of the other, joining nodes in the two spines into a single path sorted by heap order. When splitting a single tree into left and subtrees of the node n, we search for n in the tree, removing each edge traversed. We then partition the set of nodes traversed in the search operation by key ordering into left and right descendants of n and form two trees by joining nodes in each partition in heap order. These two operations are illustrated in Figure 2. Taken as a join operation, the dotted line represents the reconstruction of nodes by heap order, and the nodes are divided by their initial subtree. Taken as a split operation, the dotted line represents the search path for n and the subtrees are divided by key order with respect to n.

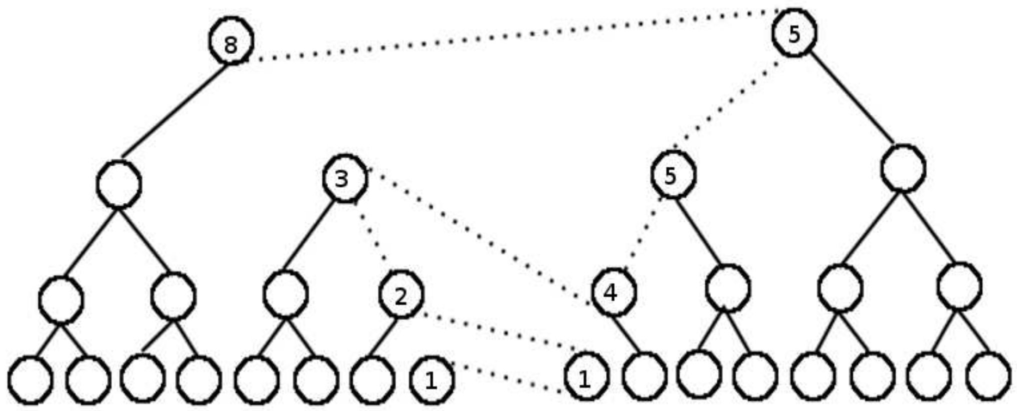

Figure 4. A depiction of split and join operations applied to zip trees

We proceed by formalizing these operations in a proof.

Theorem 6. The join operation described above transforms two valid zip trees into one valid zip tree.

Pf. To begin, we assume that all nodes in the two trees have independently random priorities and distinct keys, so we need only check that the operations satisfy the tree and heap orderings for zip trees. We therefore show that the joined tree satisfies our tree and heap orderings. At the beginning of a join operation, we have two subtrees with disjoint key spaces (by which we mean that all keys in the first tree are strictly less than all keys in the second tree). Deconstructing the right spine of the first tree and the left spine of the second tree leaves a set of atomic sub-trees which satisfy the heap and tree orderings, so it remains to show that our reconstruction does not violate these orderings. Recall that we reconstruct the tree by joining nodes in heap order. Therefore, we know that any two nodes within a subtree satisfy the heap ordering and any two nodes joined in our reconstruction satisfy the heap ordering by construction, so it follows by transitivity that any two nodes in the tree satisfy the heap ordering. Likewise, our reconstruction only assigns left children to nodes in the second tree and right children to nodes in the first tree. Therefore, the nodes that we join satisfy the tree ordering because we assume that every node in the first tree is smaller than every node in the second tree. Since we know that these nodes satisfy the tree ordering and any two nodes within our subtrees satisfy the tree ordering, it follows by transitivity that every node in the tree satisfies our tree ordering. This completes the proof, as the join operation forms a single treap with independently random priorities and a valid heap and tree ordering. ■

Theorem 7. The split operation described above transforms a single zip tree into two valid zip trees, one with keys less than a given key k and a second with keys greater than k.

Pf. As before, we begin by assuming that the priorities in our initial tree are independent and randomly generated, so any trees generated from these nodes will have distinct keys and independent randomly generated priorities (according to the same distribution as that used for zip trees). Therefore, it remains to show that the tree and heap orderings are satisfied in the two subtrees. To begin, we consider an arbitrary key k not in the tree which we use in the splitting operation (in the full insertion operation, this corresponds to the key we insert). We begin by searching for k in the tree, which involves recursively comparing the key k to keys in the tree and selecting left or right children as appropriate. Each time we traverse an edge during this process, we delete it. This operation leaves a set of subtrees, each of which follows the tree and heap orderings. Therefore, it remains to show that our reconstruction of subtrees satisfies both orderings. We begin by noting that our search operation separates the left child of any key in the path greater than k and the right child of any key less than k. Additionally, a root node with a missing right child is the largest node in the resulting tree, and a root node with missing left child is the smallest node in its subtree. Therefore, when we partition these nodes into those greater than k and those less than k, this construction ensures that all nodes in one partition are greater than k and all nodes in the other are less than k. Furthermore, all nodes in one partition have missing left children and all nodes in the other partition have missing right children. Using these facts, we consider the reconstruction of these partitions into subtrees. To begin, since our partitioning divides nodes into those with missing left children and those with missing right children, the edges we add correspond precisely to a spine of the resulting tree. We order our subtrees so each root node is assigned a child with a smaller priority. This ensures that the reconstructed subtrees maintain a heap ordering by transitivity, since heap ordering is maintained within atomic subtrees and between their root nodes. Likewise, we know by construction that any root node of an atomic subtree is the descendant of root nodes with higher priority and the ancestor of nodes with lower priority. This means that for two root nodes with missing right children, the node with higher priority has smaller key (and symmetrically, for any two nodes with missing left children, the node with higher priority has larger key). This assertion holds because the node with lower priority was constructed to be in the right subtree of the node with higher priority, so must have larger key (and symmetrically, a node with lower priority is in the left subtree of a node with missing left-child). Therefore, when we build trees within the two partitions, the conditions shown above guarantee that the tree ordering is maintained among root nodes, since larger keys are assigned as right children to smaller keys and smaller keys as left children to larger keys. Thus, by transitivity, the new trees maintain a global tree ordering by transitivity. This completes the proof, since our split operation converts a valid zip tree and key k into two subtrees, one containing all keys less than k and one containing all keys greater than k. ■

Corollary 8. Any sequence of insert and delete operations converts a valid zip tree to a valid zip tree

Pf. To begin, we note that our construction assigns independent, geometrically distributed random values on insertion. Further, since the heap ordering is hidden from the user, deleted nodes are assumed to have randomly chosen priority. Insertion consists of placing a node to satisfy the tree and heap orderings of our zip tree, then splitting its child subtree. Our placement of the node and theorem 7 guarantee that this operation produces a single valid zip tree, under the recursive definition given in theorem 2. Likewise, when deleting a node, we remove the node, join its children, and link the resulting subtree to the node's parent. Theorem 6 guarantees that this construction produces a single zip tree under the definition given in theorem 2, although it is worth noting that the ordering between the subtree and its new parent is preserved because every node in the new subtree was originally part the same subtree of the parent. Therefore, any finite sequence of such operations produces a valid zip tree. ■

Before continuing, it is worth noting that theorem 6 gives us the following bounds for operations on zip trees

Theorem 9. Zip trees have insertion, deletion, and access time given by $O(\log n)$ in the average case

Pf. When accessing a node, we trace its path from the root. Theorem 6 tells us this path contains at most an expected $O(\log n)$ nodes, so any access operation terminates in an expected $O(\log n)$ steps. Insertion and deletion operations both consist of an access operation followed by a constant number of traversals of a single path during subtree reconstruction, so theorem 6 again guarantees $O(\log n)$ steps in the average case. ■

It is worth noting that we do not consider finger searches or other properties that Seidel and Aragon consider in their discussion of treaps, in order to facilitate the transition to a concurrent algorithm.

We continue our discussion by developing a concurrent version of the zip tree algorithm. In this algorithm, we use a non-exclusive read lock, an exclusive write lock, and an atomic bit flag as signaling primitives to ensure validity in our operations. The read lock is included to clarify the relation between read and write operations, and our goal will be to indicate the circumstances in which it may be removed. The exclusive write lock is essential to enforcing an ordering among write operations, and thus the validity of the data structure after concurrent update operations. Finally, the atomic bit flag is included to minimize contention between update operations. Instead of restarting an update operation when another update forces it into an invalid state, as is done in many other implementations (cf. Herlihy 2006 and Ellis 1994), the bit flag allows an operation in this state to recover without losing previous progress. With these objectives in mind, we combine these primitives into a full algorithm. We construct access operations in the natural way by acquiring a read-lock to a node, comparing keys, acquiring a read-lock to one of its children, and releasing the first read-lock. Once the desired key is found in the tree, the function releases its final read-lock and returns. The only purpose of a non-exclusive read-lock in this construction is to ensure consistency and existence of nodes during access. Update operations proceed using a somewhat more complex sequence of locks. Both insertion and deletion operations begin with a variation of the access algorithm. Instead of acquiring a new lock and releasing its parent, we maintain a lock to the parent and release the grandparent of our new lock. This progression of read-locks proceeds until we find the location to be modified. At this point, our algorithm holds three locks: one to the parent of the node to be inserted/deleted (called P for convenience), and one each to a parent and child of P. We immediately release the lock to P's child, and attempt to set the update flag on P. If the flag is already set, another process is in conflict with our update, so we release the lock to P and wait for the flag to clear. Once the flag is clear, we resume the search from P's parent (since the tree structure below that point has changed). If we successfully set the update flag on P, we acquire a write lock to P, then clear the update flag and unlock P's parent. Next, we acquire a write lock to P's child (as we will see, we do not need to consider the bit flag here or for later nodes). Having acquired locks on all relevant nodes, we perform the update operation by modifying pointers to insert/delete the desired node as a child of P. At this point, we release our write lock to P, and proceed to rebuild the subtree(s) of the modified node in the expected manner: acquiring a write lock to a node, changing any necessary child pointers, write-locking the modified children, and releasing the lock.

We begin the discussion of this algorithm with a few comments about the specific concurrent primitives used. The specific implementation used here is designed with an implementation in C++ in mind, guiding our attention towards non-exclusive read locks and away from other features, such as garbage collection or reading our atomic bit flag. Although it is not practical to define a separate implementation for every combination of known primitives, a few notes may prove helpful. The non-exclusive read-lock may be safely ignored in an environment with garbage collection by using some implementation of flag that allows threads to check its state and by checking flags before acquiring write locks. If a simple locking framework is essential, this implementation could be modified to only lock two nodes at a time on update operations (starting over in case of conflict) or even acquire write locks starting from the root. However, both modifications are expected to substantially impact performance, as most updates will occur near the bottom of the tree.

Next, we proceed with a proof of correctness for our implementation (mentioning alternative implementations in the proof, as appropriate). In the course of this proof, we will require a concurrent version of our earlier requirement that zip trees contain distinct keys. We consider only series of concurrent operations which cannot linearize to contain a sequential state containing duplicate keys. Thus, while a node may be deleted and re-inserted, an insertion may only start once the operation deleting the node has returned.

Theorem 10. The primary implementation described above represents a valid concurrent algorithm.

Pf. We proceed using a technique known as linearization described by Herlihy and Wing. (1990) An algorithm is said to be linearizable if it is equivalent to a sequential algorithm where every operation occurs instantaneously at some point during its execution. For our proof, we place successful access operations at the moment they return, insert and successful delete operations at the moment they successfully acquire a write lock to P, and demonstrate that there exists some consistent ordering . We proceed to prove that our concurrent implementation creates a valid zip tree structure after any sequence of concurrent insert/delete/access operations, and that these concurrent operations are equivalent to sequential operations occurring at the times described above. We proceed with a separate discussion for each type of operation. Access operations acquire a read lock on each node before searching child pointers (excluding write locks on those node), and both access and update operations propagate down the tree. Therefore, each node locked by an access operation has been modified by the same update operations (those operating further down the tree) and the full access path matches that of a single state of a sequential zip tree. Since any access operation matches a sequential traversal of some valid tree state, the operation succeeds or fails based on whether this tree state contains the desired node. If the operation succeeds, it is sufficient to linearize to the return point, since at this point the tree contains a node with the correct key/value pair. If the operation fails, we must take into account any update operations which might add or remove the key during the course of the access operation. Since the access operation failed, we know that a node with the desired key cannot have been continuously present in the tree, since it would have been on the access path. Further, the requirement presented above regarding duplicate keys guarantees that any pairing of delete/insert updates to a single node contain a linearized point where the node is not present. Therefore, we may linearize the access operation to this point. Next, we verify our linearization point for successful insert/delete operations. We begin this discussion with an important point: the final state of a zip tree after two update operation is independent of their order. This conclusion immediately follows from Theorem 2, where we showed that a left-leaning treap with distinct keys has a uniquely defined structure. Any ordering of update operations leaves a valid treap with identical nodes, which must therefore represent the same structure. Thus, the ordering of update operations on different nodes is irrelevant, as long they can be placed in sequence and ordered relative to access operations and update operations to the same node. The manner in which we acquire the first write lock on an update operation effectively resolves both of these orderings. Write locks are exclusive, meaning that we can easily order an update to a node with any other operation by the order in which locks to the updated node are acquired. However, our algorithm only updates a node after acquiring a lock to its parent (P in our earlier discussion), so this ordering is equivalent to our choice of linearization point. Furthermore, every update operation modifies subtrees by acquiring write locks in sequence down the tree. At every point after the node P is found and exclusively marked, the update operation maintains either a write lock or (before the first write lock is acquired) a read-lock to P's parent. In the case where the bit-flag to P is already set, the algorithm maintains a read lock to P's parent (which is still a valid node on the search path to the desired node) and continues its access path after the previous operation has completed. This combination ensures that every node modified by an update operation has been updated by the same set of previous updates (those further down the tree), and thus that each update is a valid sequential operation in its own right. This means that our linearization of successful update operations is correct, since it is consistent with access operations and leaves the tree in the in the same state as the linearized sequential operations. Next, we note that failed insertion operations cannot occur, since this would imply an attempt to insert an already existing node, which we assumed will not happen. Finally, we consider the case of failed deletion operations. These operations perform exactly the same operation as a failed access, in that they acquire read-locks in sequence down an access path in the tree and reach the bottom without locating the desired node. Thus, we may linearize these operations in the same manner as failed accesses. This completes the proof, since we have constructed a valid linearization of all operations performed by our concurrent zip-tree implementation. ■

We close this section with a comment regarding our assumption that keys in the tree are unique. This assumption is consistent with Seidel's sequential algorithm (1996), but may present some additional challenges in a concurrent environment for certain applications. In a sequential zip tree, a globally unique set of keys may be artificially imposed by attempting to access a key before inserting it. However, this does not trivially extend to a concurrent environment, since we explicitly require a separation of operations using the same key. A similar result may be attained (with a potentially significant cost to performance) by searching for the key on insertion and read-locking the entire access path prior to the linearization point. This removes any risk of a duplicate insertion by ensuring that no node may be inserted along the same access path by a concurrent process. Further, the nodes which are read-locked the most by this implementation are those near the root, which are probabilistically accessed the most and updated the least. We leave a more thorough description of this variant to future study in order to continue with our main algorithm.

Chapter 3

An Aside on Randomization

3.1 An Aside on Randomization

In any thorough discussion of an algorithm, it is important to explore both the validity and necessity of each of the constituent operations. In the last section, we made significant progress in demonstrating the validity of zip trees as a concurrent binary search tree. However, it is worth further discussing the necessity of the operations used in this data structure. Afek et al. correctly identify the creation and storage of random variables as an expensive operation in treaps, although they consider a self-adjusting treap which uses randomization much more extensively than zip trees. (2014) Much as we might like to dispense entirely with the cost of producing and storing large numbers of random values, we cannot do so without wholly abandoning our use of a randomized treap. However, a careful consideration of our use of randomization within zip trees raises a more subtle question. While we may not be able to dispense with randomization entirely, we have not yet justified our desire to store random values within the tree. This is an important point to consider, as the inclusion of even a small random value within every node of the tree has the potential to add significant complexity and memory usage to an implementation.

In order to demonstrate the need for stored random variables, we consider a natural variation of the zip tree algorithm which discards node ranks at the end of insertion. In this variation, we begin insert operations by following the access path for a node n_i to be inserted. We then select a geometric variable r and insert the node as a child of the node r steps from the bottom of the tree. We finish by employing the split operation used in zip trees. Access and delete operations proceed in the same manner as in zip trees. Functionally, this modification inserts a node above r other nodes, instead of above nodes with priority less than r. A careful analysis of this modification certainly does not rule out the existence of a randomized search tree which does not store random values. However, by eliminating the simplest alternative, we can at least motivate our use of stored random values within zip trees.

In order to pursue this new focus, we begin by considering a few useful techniques that will guide our study. We start by representing an example sequence of geometric random variables with the ruler sequence. This sequence has been extensively referenced by Knuth (2012), Sloan (1973) and others in the course of describing trailing zeros in binary numbers, and may be represented simply using the recursive composition $F_n = F_{n-1} \cdot n \cdot F_{n-1}$. For our purposes, we will use the fact that the sequence F_n forms a geometric distribution. This may be quickly demonstrated by noting that $n \notin F_k$ $k < n$ and F_n contains two copies of F_{n-1}. Thus, any two values $k, k-1$ appear in the ratio $2:1$ as desired in our geometric distribution. We proceed by considering a few basic insertion patterns with random variables determined by the ruler sequence.

Lemma 11. Inserting $2^n - 1$ nodes in sorted order using random values selected from the ruler sequence results in a perfectly balanced tree of height n.

Pf. We demonstrate this assertion by induction. For $n = 2$, the ruler sequence is given by 121. Thus, we insert a node of priority 1 at the root, then a node of priority 2 as its parent, then another node of priority 1 as a right child. Further, we may follow the same progression starting with the right child of an existing node and achieve a balanced subtree rooted at the parent pointer. Note that this last assertion is necessary, since the sequence 131 (for instance) satisfies the first condition but not the second. For the inductive step, we suppose that an insertion of sorted values over the random sequence F_{n-1} satisfies both conditions given above (balanced tree when inserted at the root or a right child of another node). To show the same for F_n, we begin from the left and construct a balanced tree with the first subsequence F_{n-1}. Because we know this tree to be balanced, the right spine contains exactly $n - 1$ nodes. Therefore, when we insert the next node (of priority n) we insert above the $n - 1$ nodes in the right spine of the subtree, placing our new node precisely at the root (regardless of whether the tree F_n is a full tree or a right subtree). Finally, we use the fact that F_{n-1} may be built as a right child to place the remaining nodes with priorities F_{n-1}. This gives us a perfectly balanced tree containing a node of priority n as the root and two balanced subtrees of height $n - 1$ as desired. ■

Lemma 12. Inserting $2n$ nodes in the alternating order $\{0, 1, -1, 2, -2, \ldots\}$ using random values selected from the ruler sequence results in an unbalanced tree of height $2n$

Pf. As before, we proceed by induction, only this time on pairs of inserted nodes. To begin, we note that inserting one pair of nodes requires inserting the keys $\{0, 1\}$ with priority $\{1, 2\}$. We first insert the node 0 at the root and then the node 1 as its parent, giving us a set of two nodes with empty right children. Next, suppose that inserting $2n$ nodes in this sequence yields a tree of height $2n$ where all right child pointers are unused. Then, the next pair of nodes we insert will have keys $\{-n, n\}$

and priorities $\{1, k\}$ where $k \geq 2$. We proceed to insert the node $-n$ as a left child of the one leaf in the existing tree and then the node n at the root, which produces a tree of height $n+2$ with no right children. ■

This last result is already enough to suggest that our modified algorithm behaves pathologically for some common insertion sequences and sequences of random values. However, we need a bound in expectation to fully explore our modified algorithm. To this end, we begin with an insertion of sorted elements using arbitrary random values.

Lemma 13. Inserting N nodes in sorted order produces a tree whose right spine has expected depth $\log_2(N+1)$, using an expected $\log_2(N+1)$ insertions at the root.

Pf. Since we insert in sorted order, each new node will be placed on the right spine of the tree. Every node we insert at height 1 is inserted as a leaf and increases the length of the spine by 1. Any other node inserted at height $r < S$ acquires the lowest $r-1$ elements of the right spine as a left subtree and decreases the length of the spine by $r-2$. In the case $r \geq S$, we insert at the root and reset the length of the right spine to 1. We proceed by computing the expectation of this change in the length of the spine.

$$\Delta S = \begin{cases} 1 & r = 1 \\ 2 - r & 1 < r \leq S \\ 1 - S & r > S \end{cases}$$

$$Ex[\Delta S] = \frac{1}{2} + \sum_{k=2}^{S} \frac{2-k}{2^k} + \frac{1-S}{2^S} = 2^{-S}$$

If we extend this expression to the reals, we get a separable differential equation for a function $S(N)$ describing the length of the right spine after N sorted insertions

$$\frac{dS}{dN} = 2^{-S} S = \log_2 (N+1)$$

This means that every time we double the size of the tree, the expected depth of the right spine increases by 1. We conclude this lemma by considering the expected number of insertions that we make at the root over the course of our algorithm. Considering an arbitrary sequence of geometric values, we note that we may divide such a sequence into sub-sequences containing a (possibly empty) string of 1s of length ℓ terminated by some value $k > 1$. The expected length ℓ and the expected value of $k-2$ form identical distributions, so the expectation $Ex[\ell - k + 2] = 0$. When we insert a node into our tree, we add 1 to the length of the spine if the node has rank $r = 1$ and subtract $k-2$ from its length otherwise. Therefore, the expected length $S = \log_2(1+N)$ differs from a distribution with expectation 0 only by the truncation of ranks when nodes are inserted at the root. By linearity of expectation, the sum of all such truncations is $\log_2(1+N)$. Since our random sequence is independent and geometric, we conclude that the expected truncation for any node inserted at the root is 1 (since $\frac{1}{2}$ are not truncated at all, $\frac{1}{4}$ are truncated by 1, etc) and thus that the number of insertions at the root is $\log_2(1+N)$. ■

Theorem 14. Inserting N nodes in sorted order produces an unbalanced tree.

We have already determined some helpful properties regarding the length of the right spine of this tree, so we turn our attention to the interior. The left subtree of a node n is composed of the portion of the right spine of the tree displaced on inserting n. In particular, we know that the expected length of the right spine of the left subtree of n is equal to 1, since that is the expected number of nodes displaced by any node of rank $r \geq 2$. Further, we know that each insertion depth k is twice as likely as $k+1$. Therefore, recalling that the left spine of the left subtree of n is the number of nodes inserted at n before some smaller node $n' < n$, we see that both spines of the subtree have expected length 1. However, we also know that the spine lengths are given by a geometric distribution, which means that each spine has non-trivial length with probability $\frac{1}{2}$. From our insertion pattern, we know that any subtree not on the right spine was removed from the right spine in a similar fashion, so we may begin to recursively analyze the tree structure. For any left-child n, we have a probability $\frac{1}{2^r}$ that n has $r-1$ left-children and a probability of $\frac{1}{2^r}$ that n has r right children. Further, the geometric distribution we studied on the right spine indicates that the expected number of left children stays constant as we move down the right spine of a subtree. This is true because each rank r occurs with probability double that of $r+1$, so we insert at a given height an expected one time before inserting at a parent node. These properties give us everything we need to construct a recursive function F predicting the expected number of left-children produced by tracing a zig-zag path from any left-child. In order to avoid double-counting, we suppose that the node n has no left children. Thus, there is a probability $\frac{1}{2}$ that our left child n has no right children, and a probability $\frac{1}{2^k}$ that n has k nodes on the right spine of its subtree, each with an expected one left child. Summing over these probabilities, we see that each left-child generates an expected 1 child accessible by a right and then a left child pointer. Since the right spine has expected length $\log(n)$, there are an expected $n - \log(n)$ nodes not on the right spine of the tree, all of which lie on an expected $\log(n)$ paths rooted in the right spine (formed by alternating right and left child pointers). Therefore, each path must have expected length $\frac{n}{\log(n)}$, which completes the proof $\blacksquare$.

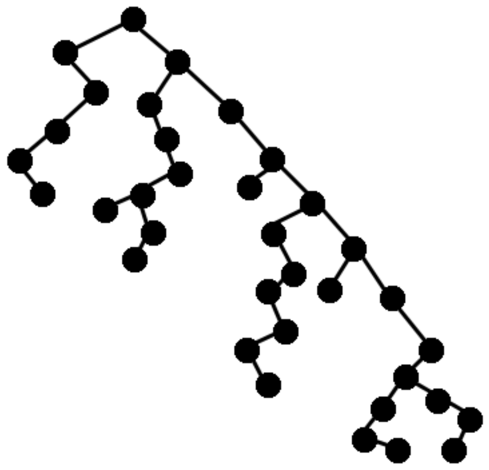

Figure 5. A skip tree with the expected number of left and right children described in the above theorem.

Although this is by no means the only possible way to design a random insertion pattern, it is by far the most natural given our study of randomization in zip trees. Thus, while we have not shown the impossibility of a zip tree without stored random values, we have demonstrated that such a tree would require non-trivial changes to the insertion pattern used for zip trees. With that qualification in mind, we continue storing random values in zip trees with at least some understanding of their place in the algorithm.

Chapter 4

A Comparison to Skip Lists

Since their initial invention, the study and development of skip lists has closely paralleled that of binary search trees. Initially developed by Pugh in 1990, skip lists achieve the same basic objectives of logarithmic insertion, deletion, and access of distinct keys, but using a fundamentally different approach. The basic structure underlying the skip list is a linked list with edges added for efficiency. On insertion, each node is assigned a random geometric value k, placed in the skip list in sorted order, and linked $\forall 1 \leq r \leq k$ by an edge to the next largest and smallest nodes of rank at least r. We note under this construction that the subset of all nodes with rank $r \geq r_0$ forms a sorted linked list in its own right, and define a set of edges of level r_0 which connects each node in the subset. An access operation for key k begins by scanning the highest level of edges. When a node is found with key greater than k, the search is restarted at the previous node and the next lowest level. The search terminates on finding either a node with key k or a node with key larger than k at level 0.

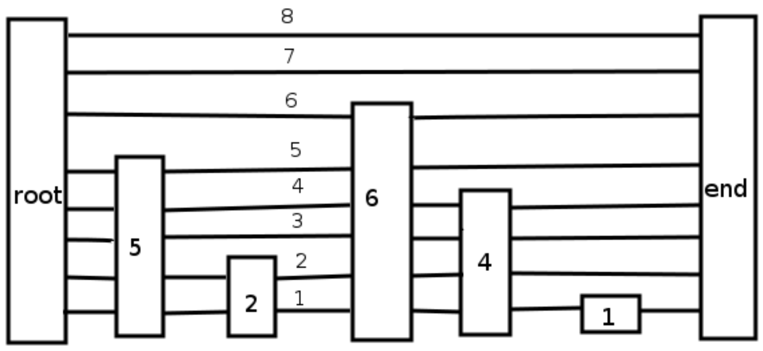

Figure 6. A rendition of a skip list, with node ranks and edge heights marked

Since their initial invention, skip lists and search trees have been shown to be surprisingly similar, despite fundamental differences in approach. Skip lists inherently depend on randomization for efficiency and never change node ranks. Search trees are typically deterministic and frequently change node heights in the tree to maintain balance. However, some effort has been made to reconcile these differences in approach. Munroe et al. developed a deterministic skip list, which inserts nodes at rank 1 and occasionally increases a node's rank to preserve a geometric balance (1992). In doing this, the authors directly acknowledged a close connection between their algorithm and 2-3 trees. Messeguer clarified this connection further by defining an explicit mapping between skip lists and a probabilistic B-trees. (1997) Martínez et. al brought the two data structures closer still by developing a search tree with probabilistic balance. (1998)

In many ways, zip trees represent the culmination of these efforts to combine the approaches behind search trees and skip lists. In fact, the sequential zip tree algorithm given earlier may be mapped readily to an efficient set of skip list operations.

For the sake of intuitive clarity in the following proofs, we begin with a lemma regarding our requirement that zip trees be left-leaning.

Lemma 15. Consider a zip tree structure where at the end of every partitioning step, the left partition is defined as left-leaning and the right partition is defined as right-leaning (we will refer to this property as direction). This structure has leaves with expected depth at most $2\log(N)$ and maps bijectively to the set of left-leaning zip trees.

Pf. To begin, we note that this partitioning scheme preserves direction across successive partitioning steps with nodes of the same rank. To see this, note that any insertion of a node in a left-leaning partition sends all nodes of the same rank to the left partition, so the root of the left subtree will be the largest node of the same rank and all remaining nodes of the same rank will again be sent to the left partition. Likewise, a node inserted in a right-leaning partition will send all nodes of the same rank to the right partition, meaning that the root of the right subtree will be the smallest node of the same rank and all remaining nodes of that rank will again be sent to the right partition. Therefore, sequences of adjacent nodes of the same rank always lie entirely on either the left or right spine of a subtree. We proceed to define a bijection mapping these trees to left-leaning zip trees. Consider a subtree where some number of nodes on the right spine have the same rank as the root. We replace the left subtree of each of these nodes with a pointer to its parent (except for the root, which we ignore) and replace the former parent's right child pointer with a pointer to the left subtree. If we replace our pointer to this subtree to point to the new root, we are left with a valid zip tree where a right-leaning sequence of nodes was replaced by a left-leaning sequence. Repeating this recursively gives us a mapping from our new partitioning scheme to left-leaning zip trees. To show that this mapping is a bijection, we define an inverse mapping which applies the reverse transformation to sequences of adjacent nodes of the same rank whose largest member is the right child of some other node. The reverse transformation referred to above replaces the right subtree of each node (except for the largest) with a pointer to its parent, and replaces the parent's left child pointer with a pointer to the right subtree. Iteratively applying this transformation inverts the mapping to left-leaning zip trees. We conclude by noting that this variety of zip tree has leaves of expected depth bounded by $2\log(n)$ as may be readily seen by repeating the proof of theorem 5 for a tree where we scan for weak maxima in both directions. ■

Theorem 16. There is a mapping from the directioned zip trees defined above to skip lists, and a mapping from every edge in the directioned zip tree to a corresponding edge in the skip list.

Pf. We define a mapping from zip trees to skip lists in the following manner. Map every node in the zip tree to a node with the same key k and rank r in a skip list. For every subtree, form an edge to the root from the largest node of each rank on the right spine of the left subtree and from the root to the smallest node of each rank on the left spine of the right subtree (we handle parent pointers recursively). If any edge was not assigned by this process, we duplicate the edge of next highest rank from the same node, if one exists. We proceed by proving that this mapping satisfies the conditions of the theorem. To begin, note that the structure we map to has geometrically distributed node ranks, and we only construct directed edges from smaller keys to larger keys, so the nodes appear in sorted order. Next, we show that every node has precisely r edges from the left and to the right, and that these edges correspond to the next smallest and largest nodes at each rank. To show this, we first consider edges connecting a node n to nodes of strictly lesser rank. We chose directed edges connecting n to precisely one node of each rank on the spines of the two subtrees. Therefore, these nodes contribute at most $r-1$ edges both to and from n. Further, our choice of the closest node of each rank to n on the spine is equivalent to choosing the closest node to n of rank at least ℓ. To see this, suppose that some other node n' of rank at least ℓ is closer than a node n_ℓ satisfying these properties. We assume wlog that these nodes satisfy the ordering of keys $n_\ell > n$. In our recursive zip tree construction, the node n_ℓ was placed in the same partition as n after every choice of root prior to n, in the right partition after choosing n as the root, and the left partition of every following choice of root node. We proceed by tracing where the node n' is partitioned at each stage in the recursive construction of our zip tree. The key orderings $n' > n$ and $n' < n_\ell$ tell us that n' must be in the same partition as n, n_ℓ when n is selected as the root of a subtree (by applying lemma 4 to the ancestor n of n_ℓ). Since $n' > n$, n' must be partitioned into the right subtree of n, and since $n' < n_\ell$, n' must be placed in the left partition at every stage until one node is chosen as the root. However, if n' has rank ℓ this violates our choice of the smallest node on the subtree of rank ℓ and if n' has larger rank, this means we chose n_ℓ as the root of a subtree containing a node n' of larger rank (contradiction). Therefore, we need only verify the two edges at height r. Suppose wlog that the node n is a left child of some other node (or is the root). First, note that n cannot have an edge of height r to its right child, since n is directed left and any nodes of the same rank must be left children. Further, n must have an edge pointing to its parent (unless n is the root), since n is directed left and is therefore the smallest node of its rank on the right spine of the left subtree of its rank. Finally, suppose n is on the left spine of the right subtree of some other node n'. Then, if n has a left child of the same rank, n will not be adjacent to n', since it is not the smallest node of its rank on the left spine of the right subtree of n'. However, if n does not have a left child of the same rank, n is the smallest node of its rank on the left spine of the right subtree of n' and is joined by an edge of rank r to n'. Since both conditions do not happen simultaneously, we have demonstrated that every node has at most r directed edges from the left and the right. We note as an aside that any node n must have a pointer to its left and right children. Since its left child is directed left and its right child is directed right, both nodes are the smallest of their rank on the appropriate spine of their respective subtrees. This is sufficient to demonstrate that every edge from our zip tree is included in the skip list (although source and destination nodes may be swapped). We next seek to consider the case when the above inclusions generate fewer than r edges in one direction. To begin, suppose that a node n is missing an edge of maximal rank k. Suppose wlog that the missing edge should be directed from n to the right. Then, n must be a right child (or the root), since we just asserted that n has a pointer to its parent. Further, no ancestor n' of n satisfies $n' > n$, since otherwise n would be on the right spine of a left subtree of n'. Therefore, we see that n must be on the right spine of the tree. Depending on the implementation, we either link these edges to dummy root/end nodes or leave them empty. Next, suppose that an edge of rank $\ell < r$ is missing, but an edge to a node of larger rank is present. Again suppose wlog that the edge is directed right. Let n' be the node of smallest rank $\ell_0 > \ell$. Suppose that n' is not the node of smallest key $n' > n$ and rank at least ℓ_0. Then there must be some smaller node $\hat{n}$ satisfying both conditions. Since $n < \hat{n} < n'$, lemma 4 tells us that $\hat{n}$ must be in the same partition as n' when n is selected as the root of a subtree. At this point, we know $\hat{n} < n'$ and n' is on the left spine of the right subtree of n. Therefore, both $n, \hat{n}$ are always in the left partition of each subtree before one of them is selected as a root. If $\hat{n}$ is chosen as a root, first, n' must be a right child, contradicting our assumption that n' is on the left spine of the subtree. If n' is chosen as root first, we contradict our assumption that n' has smallest rank at least ℓ on the left spine and smallest key $n' > n$. This means we may duplicate the pointer to n' (which should be the smallest pointer larger than the missing pointer at ℓ) without violating the edge requirements for skip lists.

Corollary 17. The mapping described above is a bijection.

Pf. We proceed by constructing an inverse mapping. From an arbitrary skip list, select the node of largest rank (and largest key, should the largest rank not be unique) as the root. Next, select the left child of the root to be the source of the edge of largest rank directed to the root from the left, and the right child of the root to be the edge of largest rank directed from the root to the right. Next, delete the root and all adjacent edges and repeat the process recursively over the set of skip lists thus generated. This is precisely the formula described in theorem 2, except we include the largest node of a given rank as the left child and the smallest node of a given rank as the right child. Thus, we have a mapping from skip lists to directed zip trees, which completes the proof of bijection ■

Corollary 18. Any insertion or deletion operation in a zip tree is analogous to the corresponding operation on a skip list

Pf. For simplicity, we use the inverse mapping defined by corollary 17. An insertion operation in a skip list of a node with key k and rank r breaks precisely those edges of rank $r_0 < r$ with source $k_0 < k$ and destination $k_1 > k$. Since the access path followed during insertion in a zip tree traverses all edges satisfying $k_0 < k$ and $k_1 > k$ for source/destination nodes k_0, k_1 (in some order), it is sufficient to consider changes to edges on the access path for k in a zip tree. In a skip list, we follow edges of rank greater than r, and then break smaller edges to link the new node to its predecessors and successors. In a zip tree, we follow edges of rank greater than r, insert the node and perform a split operation on the subtree of our selected parent for r. The split operation removes edges between nodes $k_0 < k < k_1$ and replaces them with links from a node n' to a node on the left spine of its right subtree (or symmetrically the right spine of its left subtree). Interestingly, only the parent and children of n in a zip tree actually add the same edges as in a skip list. All other decendents of the inserted node remove one edge of maximal rank from the zip tree and add the other incident edge of the same rank (reversing the edge's direction in the process). Thus, the insertion pattern is broadly analogous, up to a predictable change in the mapping between edges. Similarly, the deletion operation treats adjacent edges to the parent and children of n identically in skip list and zip tree, but for all edges removed the subtrees of the deleted node we add the same edge to zip tree as to skip list, but delete the opposite edge of maximal rank from the zip tree representation. This completes the proof, as update operations in the two structures are equivalent up to a change in the edge mapping from zip trees to skip lists. ■

This suffices to show that zip trees are very similar in structure to skip lists, although we must be cautious with edge direction when discussing the two. From now on we largely abandon the variants and notation developed so far, and return to the more convenient notation of left-leaning zip trees.

Chapter 5

The Performance of Zip Trees

Now that we have established this close connection between zip trees and skip lists, we seek to compare our concurrent zip tree implementation to existing concurrent skip list algorithms. Although several implementations of concurrent skip lists exist, for our primary discussion of the practical efficiency of zip trees, we turn to the one most concerned with optimization. This implementation was developed by Herlihy et al. in an effort to improve on a concurrent skip list library packaged with Java. (2006) Herlihy also helpfully provides detailed pseudocode in his paper, which provides the basis for a fair comparison with his algorithm. Most importantly, however, Herlihy performed detailed testing on a very similar implementation to the skip list version of zip trees. Herlihy's algorithm first finds all predecessors and successors of the desired update location and stores references. He then acquires locks to nodes in increasing order by rank, validating nodes in the process. If a node has changed since storing its reference, his algorithm releases all locks and restarts. Otherwise, once he acquires all locks, he reassigns pointers and performs the update, finally releasing all locks again on exit. This algorithm does not acquire any sort of lock during read operations, instead relying on atomic pointers and garbage collection to avoid referencing nodes which have already been deleted. Zip trees (represented as skip

lists) perform updates by acquiring write locks incrementally from the top down and changing pointers at each level. We never need to release locks in case of contention, as the constant progression of operations down the tree and exclusive write-locking at each level guarantee that we will never reach such a point.
Our preferred platform for comparing these two algorithms is a Raspberry Pi 2v1.2 running C++ and using the freely available POCO library. Our goal in doing so is to introduce some degree of reproducibility, as the Raspberry Pi is an inexpensive and widely available multi-core processor. Further, our choice of C++ was intended to expose low level primitives and reveal the underlying assumptions of both algorithms. This combination of equipment does have one main disadvantage, in that POCO is a large library and the Raspberry Pi is a rather small processor for this sort of concurrent testing. In the interest of providing a reproducible platform, we worked within the framework of this limited hardware to the degree possible.

The first step in providing this comparison was rendering Herlihy's implementation in C++. This provided some interesting challenges, as Herlihy's algorithm relies very directly on garbage collection, which is not provided in C++. While it is theoretically possible to convert his implementation from atomic pointers and garbage collection to the non-exclusive read locks used in our implementation, or even implement some manner of reference counting to prevent memory errors, both measures were deemed to stray too far from Herlihy's original design. In the interest of a fair comparison, we incorporated a controlled memory leak into our implementation. Rather than base our results on the version of garbage collection implemented, our comparison will assume that Herlihy's algorithm has access to a free garbage collector and simulate this impossibility by never deleting nodes from memory. It is expected that this will make his algorithm somewhat faster than expected, but this will turn out to have little effect on our experimental results. The only other large changes we make to the pseudocode provided in Herlihy's paper are removing some input validation routines which our implementation of zip trees does not perform, replacing his pointers with C++ atomic types and removing a try-finally block which does not match a C++ primitive.

We begin our discussion with a direct comparison of sequential skip lists and zip trees. To aid in testing, we implemented a wrapper function which would accept key parameters as command line arguments and perform randomized operations with randomized keys. All keys were selected randomly for insertion and then placed in an array visible to access and delete operations. Although our program allowed the total number of operations and ratio of update to access operations to be specified at runtime, the ratio of insert to delete operations was fixed at 3:1 to allow the tree to grow quickly and mitigate the effects of memory on Herlihy's algorithm. Finally, all testing alternated between the two algorithms to control for the operational health of the hardware. The below graphs show simulation results generated using this code, which is provided in Appendix A.

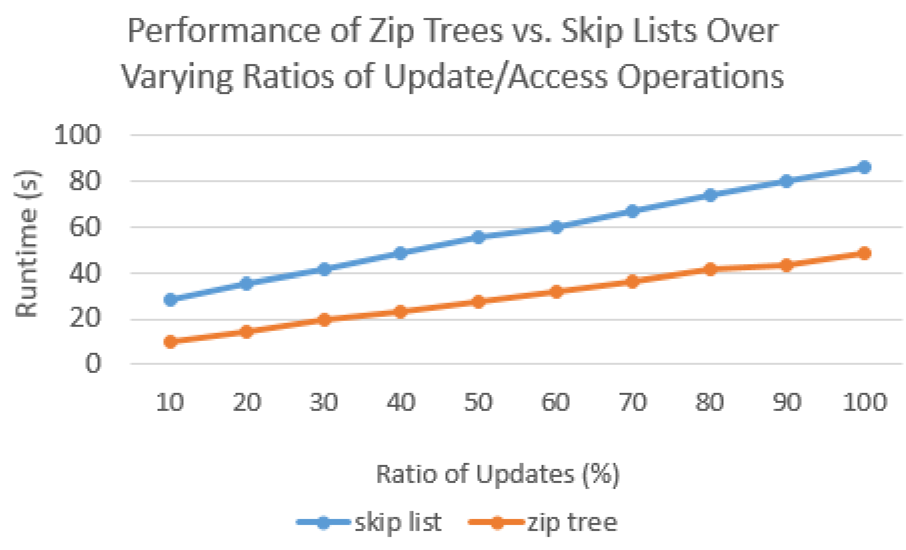

Figure 7. A comparison of skip lists and zip trees over 2 million operations

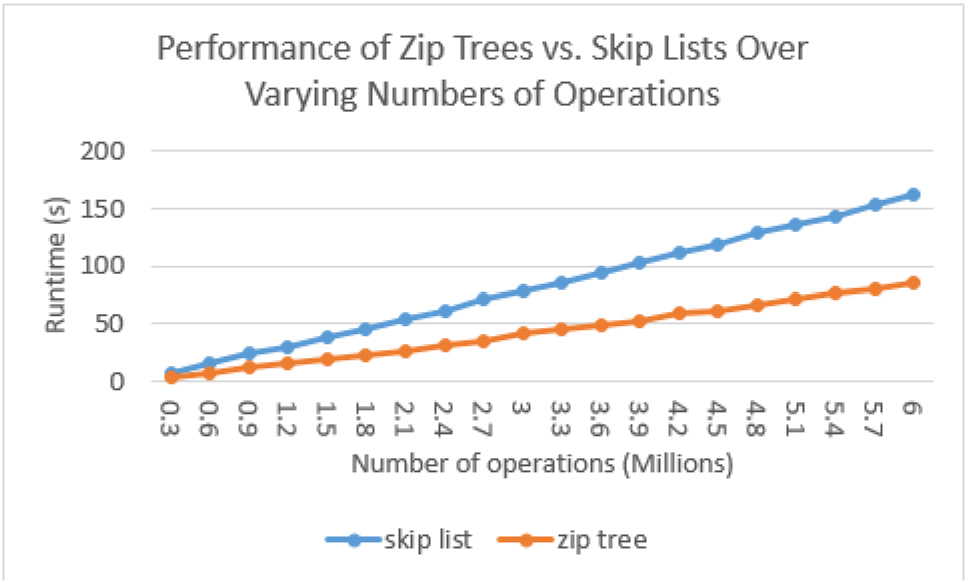

Figure 8. A comparison of sequential skip lists and zip trees with a 50% insertion ratio.

The most impressive result here is that despite their apparent similarities, zip trees perform roughly twice as fast in practice as skip lists. This is very likely due to the number of pointers that are created and maintained in a skip list implementation.

Next, we repeat a similar set of simulations for concurrent versions of the algorithms, again using code reproduced in the appendices. We tested a concurrent zip tree implementation and our close modification of Herlihy's algorithm using a test program which accepted a number of threads, number of operations, and update/access ratio at runtime. In order to share data between threads in the test program, we defined a mutex protecting our random number generator and array of keys. In order to best suit the implementation focuses of the algorithms, we implemented zip trees with POCO's RWlocks (which allow a non-exclusive read xor an exclusive write) and an atomic flag,and implement Herlihy's algorithm using atomic arrays and a single mutex per node. Unfortunately, we cannot fully consider the performance of the two algorithms over varying numbers of threads, as a combination of hardware limitations and the two mutexes in our test code removed any substantial performance gains over multiple cores regardless of the algorithm. However, both programs have been tested to 100 threads on the scales described below, and the tests we will discuss were not impacted by these hardware limitations

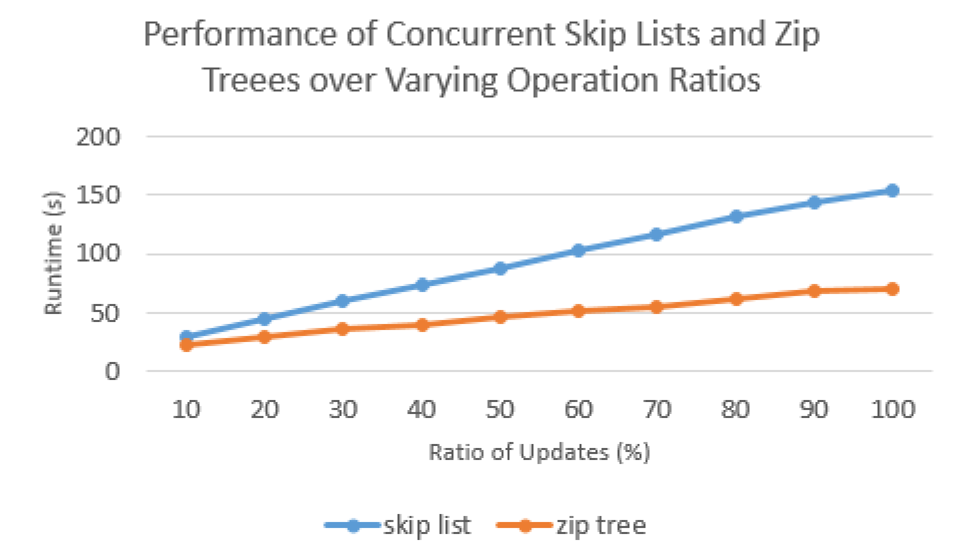

Figure 9. A comparison of concurrent skip lists and zip trees over 1 million operations and 10 threads.

We observe again that zip trees consistently outperform skip lists on mixed update and access operations. However, perhaps the most interesting result is that zip trees perform much better with large numbers of update operations, while Herlihy's algorithm almost catches up for large numbers of accesses. This primarily represents a difference in implementation focus. Herlihy uses the built-in garbage collection provided by Java to avoid read-locking entirely. Our implementation of zip trees, by contrast, is somewhat more focused on update operations and uses more expensive non-exclusive read locks. Based on these results, we note that our zip tree implementation could be redesigned to use atomic pointers and modify the tree structure in a way that optimizes access operations. However, we leave a full implementation of a zip tree with lazy modification of pointers for future study.

Chapter 6

Conclusion

6.1 conclusion

In the course of this discussion, we have demonstrated that zip trees are theoretically sound and experimentally efficient. We also demonstrated a close link between concurrent algorithms on search trees and skip lists and developed some novel insight regarding the need to store random values in treaps. However, there remain many loose ends scattered through our discussion and many ways it could have been carried further.

Our discussion of randomized search trees without stored rank is perhaps the biggest of these. Our earlier analysis was suggestive, but ultimately only narrowed down the space of possible randomizations by a single likely candidate. The question we posed is worthy of study in its own right, and hopefully that question will someday be more convincingly answered.

From a more pragmatic standpoint, the lack of a universally reproducible test environment for concurrent algorithms is somewhat worrying. The process of implementing a concurrent algorithm often relies heavily on features that vary greatly between programming languages. Some standardization in test environment would at least make this difference in perspective more explicit, but ultimately some agreement on allowable resources and objectives will be more useful. Otherwise, any attempt to reproduce experimental results will result in something very much like our comparison of Herlihy's algorithm, where everything from Java's garbage collector to a need to validate user input becomes essential to the algorithm's design. Unfortunately, we fell into this trap to some degree in describing zip trees, by crafting an algorithm with no reliance on garbage collection and a specific implementation of locking mechanism. However, this does leave open the possibility for a future discussion of what language features, if any, are essential to zip trees as a data structure.

Finally, we made no direct comparison between zip trees and the varieties of search tree most noted for sacrificing theoretical soundess for practical speed. It is our belief that zip trees are efficient enough to merit practical use, but this will eventually require some form of contest against algorithms with much looser theoretical guarantees.

Appendix A

Reference Zip Tree Code

Below are implementations of sequential and concurrent zip trees, along with test code, in the hope that it will be of use. For the sake of brevity, skip list implementations are not shown, and the interested reader is encouraged to consult Pugh (1990) and Herlihy et al. (2006) for similar implementations.

A.1 zip tree

```
#include <stdio.h>
#include "zip_tree.h"
#include <iostream>
#include <stdexcept>
Zip_tree::Zip_tree(void) {
  root = NULL;
  generator = new std::geometric_distribution<int> (0.5);
  seed = new std::default_random_engine();
}
void Zip_tree::insert(int key, float value) {
  int rank = (*generator)(*seed);
  node* new_node = root;
  node* old_node = NULL;
  while (new_node != NULL && new_node->rank > rank) {
    old_node = new_node;
    if (key > old_node->key) {
      new_node = old_node->right_child;
    }
    else {
      new_node = old_node->left_child;
    }
  }
  while (new_node != NULL && new_node->rank >= rank
                          && new_node->key > key) {
    old_node = new_node;
    new_node = old_node->left_child;
  }
  node* inserted = new node();
  inserted->rank = rank;
  inserted->key = key;
  inserted->value = value;
  if (old_node == NULL) {
    root = inserted;
  }
  else {
    if (old_node->key > key) {
      old_node->left_child = inserted;
    }
    else {
      old_node->right_child = inserted;
    }
  }
  node* right_subtree = NULL;
  node* left_subtree = NULL;
  while (new_node != NULL) {
    if (new_node->key > key) {
      if (right_subtree == NULL) {
  inserted->right_child = new_node;
  right_subtree = new_node;
  new_node = new_node->left_child;
  right_subtree->left_child = NULL;
      }
      else {
right_subtree->left_child = new_node;
right_subtree = new_node;
new_node = new_node->left_child;
right_subtree->left_child = NULL;
      }
    }
    else {
      if (left_subtree == NULL) {
inserted->left_child = new_node;
left_subtree = new_node;
new_node = new_node->right_child;
left_subtree->right_child = NULL;
      }
      else {
left_subtree->right_child = new_node;
left_subtree = new_node;
```

```
new_node = new_node->right_child;
left_subtree->right_child = NULL;
      }
    }
  }
}
bool Zip_tree::contains (int key) {
  node *search_node = root;
  while (search_node != NULL) {
if (search_node->key > key) {
  search_node = search_node->left_child;
}
else if (search_node->key < key) {
  search_node = search_node->right_child;
}
else {
  return true;
}
  }
  return false;
}
void Zip_tree::remove (int key) {
  node *new_node = root;
  node *old_node = NULL;
  bool which_child; //false is left, true is right
  while (new_node != NULL && new_node->key != key) {
    old_node = new_node;
    if (old_node->key > key) {
      new_node = old_node->left_child;
  which_child = false;
    }
    else if (old_node->key < key) {
      new_node = old_node->right_child;
  which_child = true;
    }
  }
  if (new_node == NULL)
    throw std::invalid_argument("node not in tree");
  node *left_child = new_node->left_child;
  node *right_child = new_node->right_child;
  delete new_node;
  node** parent;
  if (old_node == NULL) {
  parent = &root;
  root = NULL;
  }
  else {
  if (which_child) parent = &(old_node->right_child);
  else parent = &(old_node->left_child);
  *parent = NULL;
  }
  while (left_child != NULL && right_child != NULL) {
    if (left_child->rank > right_child->rank) {
      *parent = left_child;
      parent = &(left_child->right_child);
      left_child = left_child->right_child;
      *parent = NULL;
    }
    else {
      *parent = right_child;
      parent = &(right_child->left_child);
      right_child = right_child->left_child;
      *parent = NULL;
    }
  }
  if (left_child == NULL && right_child != NULL)
    *parent = right_child;
  if (left_child != NULL && right_child == NULL)
    *parent = left_child;
  return;
}
float Zip_tree::access (int key) {
  node *search_node = root;
  while (search_node != NULL) {
    if (search_node->key > key) {
      search_node = search_node->left_child;
    }
    else if (search_node->key < key) {
      search_node = search_node->right_child;
    }
    else {
      return search_node->value;
    }
  }
  throw std::invalid_argument("key not in tree");
}
Zip_tree::~Zip_tree() {
}
```

A.2 test program

```
#include<stdlib.h>
#include<time.h>
#include<stdio.h>
#include "zip_tree.h"
#include <iostream>
#include <assert.h>
using zip_tree::Zip_tree;
int main(int argc, char **argv) {
int ins_to_del = 25;
int num_ops = atoi(argv[1]);
int ratio = atoi(argv[2]);
std::vector<int> keys {};
srand(time(NULL));
clock_t start = clock();
Zip_tree* test_tree = new Zip_tree();
for (int i = 0; i < num_ops; i++) {
        int op_type = (int)(100.0 * rand() / (RAND_MAX + 1.0)) + 1;
if (op_type < ratio || keys.empty()) {
  int ins_del = (int)(100.0 * rand() / (RAND_MAX + 1.0)) + 1;
  if (ins_del > ins_to_del || keys.empty()) {
    int key = rand();
    while (test_tree->contains(key)) key = rand();
    float value = (float) rand() / (float) RAND_MAX;
    test_tree->insert(key, value);
    keys.push_back(key);
  }
  else {
    int loc = (int)(keys.size() * rand() / (RAND_MAX + 1.0));
    int key = keys[loc];
    test_tree->remove(key);
    keys[loc] = keys.back();
    keys.pop_back();
  }
}
else {
  int loc = (int)(keys.size() * rand() / (RAND_MAX + 1.0));
  int key = keys[loc];
  float value = test_tree->access(key);
}
}
double duration = ( clock() - start ) / (double) CLOCKS_PER_SEC;
std::cout << "end size = " << std::to_string(keys.size())
          << "\n time = " << std::to_string(duration) << "\n";
}
```

A.3 concurrent zip tree

```
#include <stdio.h>
#include "conc_zip_tree.h"
#include <iostream>
#include <stdexcept>
#include "/usr/local/include/Poco/Thread.h"
#include <assert.h>
Conc_Zip_tree::Conc_Zip_tree(void) {
  generator = new std::geometric_distribution<int> (0.5);
  seed = new std::default_random_engine();
  random = new Poco::Mutex();
  root = new node();
  root->update_node = new Poco::RWLock();
  root->writer_waiting = new std::atomic_flag(ATOMIC_FLAG_INIT);
  root->left_child = new node();
  root->left_child->update_node = new Poco::RWLock();
  root->left_child->writer_waiting = new std::atomic_flag(ATOMIC_FLAG_INIT);
}
void Conc_Zip_tree::insert(int key, float value) {
  random->lock();
  int rank = (*generator)(*seed);
  random->unlock();

  node* inserted = new node();
  inserted->rank = rank;
  inserted->key = key;
  inserted->value = value;
  inserted->update_node = new Poco::RWLock();
  inserted->writer_waiting = new std::atomic_flag(ATOMIC_FLAG_INIT);

  node *grandparent = root;
  node *old_node = root;
  node *new_node = root;
  grandparent->update_node->readLock();

  while (true) {
    while (new_node == root || new_node == root->left_child) {
      if (new_node != grandparent) {
        new_node->update_node->readLock();
      }
      grandparent = old_node;
      old_node = new_node;
      new_node = new_node->left_child;
    }

    while (new_node != NULL && new_node->rank > rank) {
      if (new_node != grandparent) {
        new_node->update_node->readLock();
      }
      if (old_node != grandparent) {
    grandparent->update_node->unlock();
      }
      grandparent = old_node;
      old_node = new_node;
      if (key > old_node->key) {
new_node = new_node->right_child;
      }
      else {
new_node = new_node->left_child;
      }
    }

    while (new_node != NULL && new_node->rank >= rank && new_node->key > key) {
      if (new_node != grandparent) {
new_node->update_node->readLock();
  }
      if (old_node != grandparent) {
grandparent->update_node->unlock();
      }
      grandparent = old_node;
      old_node = new_node;
      new_node = new_node->left_child;
    }

    if (old_node->writer_waiting->test_and_set(std::memory_order_seq_cst)) {
      old_node->update_node->unlock();
      std::atomic_flag *flag = old_node->writer_waiting;
      new_node = grandparent;
      old_node = grandparent;
      while (flag->test_and_set(std::memory_order_seq_cst));
      flag->clear(std::memory_order_seq_cst);
      continue;
    }
    else {
      old_node->update_node->unlock();
      old_node->update_node->writeLock();
      if (old_node->key > key || old_node == root->left_child) {
old_node->left_child = inserted;
      }
      else {
old_node->right_child = inserted;
      }
      inserted->update_node->writeLock();
      old_node->writer_waiting->clear(std::memory_order_seq_cst);
      old_node->update_node->unlock();
      grandparent->update_node->unlock();
      break;
    }
  }
  node* right_subtree = NULL;
  node* left_subtree = NULL;

  while (new_node != NULL) {
    new_node->update_node->writeLock();
    if (new_node->key > key) {
      if (right_subtree == NULL) {
inserted->right_child = new_node;
    if (left_subtree != NULL) {
     inserted->update_node->unlock();
    }
    right_subtree = new_node;
    new_node = new_node->left_child;
    right_subtree->left_child = NULL;
      }
      else {
    right_subtree->left_child = new_node;
       right_subtree->update_node->unlock();
    right_subtree = new_node;
    new_node = new_node->left_child;
    right_subtree->left_child = NULL;
      }
    }
    else {
      if (left_subtree == NULL) {
    inserted->left_child = new_node;
    if (right_subtree != NULL) {
      inserted->update_node->unlock();
    }
    left_subtree = new_node;
    new_node = new_node->right_child;
    left_subtree->right_child = NULL;
      }
      else {
    left_subtree->right_child = new_node;
    left_subtree->update_node->unlock();
    left_subtree = new_node;
    new_node = new_node->right_child;
    left_subtree->right_child = NULL;
      }
    }
  }
  if (left_subtree == NULL || right_subtree == NULL) {
    inserted->update_node->unlock();
  }
  if (left_subtree != NULL) {
    left_subtree->update_node->unlock();
  }
  if (right_subtree != NULL) {
    right_subtree->update_node->unlock();
  }
}

bool Conc_Zip_tree::contains (int key) {
  node *search_node = root->left_child;
  node *last_lock = search_node;
  search_node->update_node->readLock();
  search_node = search_node->left_child;

  while (search_node != NULL) {
    search_node->update_node->readLock();
    last_lock->update_node->unlock();
    if (search_node->key > key) {
      last_lock = search_node;
      search_node = search_node->left_child;
    }
    else if (search_node->key < key) {
      last_lock = search_node;
      search_node = search_node->right_child;
    }
    else {
      float return_val = search_node->value;
      search_node->update_node->unlock();
      return true;
    }
  }
  last_lock->update_node->unlock();
  return false;
}

void Conc_Zip_tree::remove (int key) {
  node** parent;
  Poco::RWLock *parent_lock;
  node *left_child;
  node *right_child;
  node *grandparent = root;
  grandparent->update_node->readLock();
  node *old_node = root;
  node *new_node = root;
  bool which_child;
  while (true) {
    while (new_node == root || new_node == root->left_child) {
      if (new_node != grandparent) {
    new_node->update_node->readLock();
      }
      grandparent = old_node;
      old_node = new_node;
      new_node = new_node->left_child;
    }
    while (new_node != NULL && new_node->key != key) {
      if (grandparent != new_node) {
    new_node->update_node->readLock();
  }
      if (grandparent != old_node) {
    grandparent->update_node->unlock();
      }
      grandparent = old_node;
      old_node = new_node;
      if (key > old_node->key) {
    new_node = new_node->right_child;
    which_child = false;
      }
      else {
    new_node = new_node->left_child;
    which_child = true;
      }
    }
    if (new_node == NULL) {
      throw std::invalid_argument("node not in tree");
    }
    if (old_node->writer_waiting->test_and_set(std::memory_order_seq_cst)) {
      old_node->update_node->unlock();
      std::atomic_flag *flag = old_node->writer_waiting;
      new_node = grandparent;
      old_node = grandparent;
      while (flag->test_and_set(std::memory_order_seq_cst));
      flag->clear(std::memory_order_seq_cst);
      continue;
    }
    else {
      old_node->update_node->unlock();
      old_node->update_node->writeLock();
      if (which_child || grandparent == root) {
        parent = &(old_node->left_child);
      }
      else {
    parent = &(old_node->right_child);
      }
      parent_lock = old_node->update_node;
      *parent = NULL;
      new_node->update_node->writeLock();
      left_child = new_node->left_child;
      right_child = new_node->right_child;
      new_node->update_node->unlock();
      delete new_node->update_node;
```

```
      delete new_node->writer_waiting;
      delete new_node;
      old_node->writer_waiting->clear(std::memory_order_seq_cst);
      grandparent->update_node->unlock();
      break;
    }
  }
  while (left_child != NULL && right_child != NULL) {
    if (left_child->rank <= right_child->rank) {
      right_child->update_node->writeLock();
      *parent = right_child;
      parent_lock->unlock();
      parent = &(right_child->left_child);
      parent_lock = right_child->update_node;
      right_child = right_child->left_child;
      *parent = NULL;
    }
    else {
      left_child->update_node->writeLock();
      *parent = left_child;
      parent_lock->unlock();
      parent = &(left_child->right_child);
      parent_lock = left_child->update_node;
      left_child = left_child->right_child;
      *parent = NULL;
    }
  }
  if (left_child == NULL && right_child != NULL) {
    *parent = right_child;
  }
  if (left_child != NULL && right_child == NULL) {
    *parent = left_child;
  }
  parent_lock->unlock();
  return;
}

float Conc_Zip_tree::access (int key) {
  node *search_node = root->left_child;
  node *last_lock = search_node;
  search_node->update_node->readLock();
  search_node = search_node->left_child;
  while (search_node != NULL) {
    search_node->update_node->readLock();
    last_lock->update_node->unlock();
    if (search_node->key > key) {
      last_lock = search_node;
      search_node = search_node->left_child;
    }
    else if (search_node->key < key) {
      last_lock = search_node;
      search_node = search_node->right_child;
    }
    else {
      float return_val = search_node->value;
      search_node->update_node->unlock();
      return return_val;
    }
  }
  throw std::invalid_argument("key not in tree");
}

Conc_Zip_tree::~Conc_Zip_tree() {
}
```

A.4 concurrent test program

```
#include "/usr/local/include/Poco/Thread.h"
#include "/usr/local/include/Poco/Runnable.h"
#include <iostream>
#include<stdlib.h>
#include<time.h>
#include<stdio.h>
#include "conc_zip_tree.h"
#include <iostream>
#include <assert.h>
#include "/usr/local/include/Poco/Mutex.h"
#include "/usr/local/include/Poco/RWLock.h"

using conc_zip_tree::Conc_Zip_tree;

int ins_to_del;
int num_ops;
int ratio;
int num_threads;
Poco::Mutex *random_lock;
Poco::RWLock *keys_lock;
std::vector<long> keys;
Conc_Zip_tree* test_tree;

class Test_thread: public Poco::Runnable
{
  virtual void run()
  {
    std::string printstr;
    for (int i = 0; i < num_ops/num_threads; i++) {
      random_lock->lock();
      int op_type = (int)(100.0 * rand() / (RAND_MAX + 1.0)) + 1;
      random_lock->unlock();
      keys_lock->readLock();
      bool empty = keys.empty();
      if (op_type < ratio || empty) {
keys_lock->unlock();
random_lock->lock();
int ins_del = (int)(100.0 * rand() / (RAND_MAX + 1.0)) + 1;
random_lock->unlock();
keys_lock->writeLock();
bool empty = keys.empty();
if (ins_del > ins_to_del || empty) {
  keys_lock->unlock();
     random_lock->lock();
     float value = (float) rand() / (float) RAND_MAX;
     int key = rand();
     random_lock->unlock();
     while (test_tree->contains(key)) {
     random_lock->lock();
     key = rand();
     random_lock->unlock();
     }
     test_tree->insert(key, value);
     keys_lock->writeLock();
     keys.push_back(key);
     keys_lock->unlock();
   }
   else {
      random_lock->lock();
      int loc = (int)(keys.size() * rand() / (RAND_MAX + 1.0));
      random_lock->unlock();
      int key = keys[loc];
      keys[loc] = keys.back();
      keys.pop_back();
      keys_lock->unlock();
      test_tree->remove(key);
    }
      }
      else {
    random_lock->lock();
    int loc = (int)(keys.size() * rand() / (RAND_MAX + 1.0));
    random_lock->unlock();
    int key = keys[loc];
    float value = test_tree->access(key);
    keys_lock->unlock();
      }
    }
  }
};
int main(int argc, char** argv)
{
ins_to_del = 25;
num_ops = atoi(argv[1]);
ratio = atoi(argv[2]);
num_threads = atoi(argv[3]);
random_lock = new Poco::Mutex();
keys_lock = new Poco::RWLock();
keys = {};
srand(time(NULL));
clock_t start = clock();

test_tree = new Conc_Zip_tree();
Test_thread **operations = (Test_thread **)malloc(num_threads * sizeof(Test_thread*));
for (int i = 0; i < num_threads; i++)
  operations[i] = new Test_thread();
Poco::Thread **threads = (Poco::Thread **)malloc(num_threads * sizeof(Poco::Thread*));
```

```
for (int i = 0; i < num_threads; i++)
  threads[i] = new Poco::Thread();
for (int i = 0; i < num_threads; i++)
threads[i]->start(*operations[i]);
for (int i = 0; i < num_threads; i++)
threads[i]->join();
double duration = ( clock() - start ) / (double) CLOCKS_PER_SEC;
std::cout << "end size = " << std::to_string(keys.size())
          << "\n time = " << std::to_string(duration) << "\n";
return 0;
}
```

Bibliography

Adelson-Velskii, G. M., & Landis, E. M. (1962). An information organization algorithm. *In Doklady Akademia Nauk SSSR*, (Vol. 146, pp. 263-266).

Afek, Y., Kaplan, H., Korenfeld, B., Morrison, A., & Tarjan, R. E. (2014). The CB tree: a practical concurrent self-adjusting search tree. *Distributed computing*, 27(6), 393-417.

Ellis, C. S. (1980). Concurrent Search and Insertion in AVL. *IEEE Transactions on Computers*, 100(29).

Fatourou, P., & Kallimanis, N. D. (2011, June). A highly-efficient wait-free universal construction. *In Proceedings of the twenty-third annual ACM symposium on Parallelism in algorithms and architectures*, (pp. 325-334). ACM.

Gabarr, J., Messeguer, X., & Riu, D. (1997, November). Concurrent rebalancing on HyperRed-Black trees. *In Computer Science Society, 1997. Proceedings., XVII International Conference of the Chilean*, (pp. 93-104). IEEE.

Gao, H., & Hesselink, W. H. (2007). A general lock-free algorithm using compare-and-swap. *Information and Computation*, 205(2), 225-241.

Guibas, L. J., & Sedgewick, R. (1978, October). A dichromatic framework for balanced trees. *In Foundations of Computer Science, 1978., 19th Annual Symposium on*, (pp. 8-21). IEEE.

Herlihy M. (1993). A methodology for implementing highly concurrent data objects. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 15(5), 745-770.

Herlihy, M., Lev, Y., Luchangco, V., & Shavit, N. (2006). A provably correct scalable concurrent skip list. *In Conference On Principles of Distributed Systems (OPODIS).*

Herlihy, M. P., & Wing, J. M. (1990). Linearizability: A correctness condition for concurrent objects. *IACM Transactions on Programming Languages and Systems (TOPLAS)*, 12(3), 463-492.

Lamoureux, M. G., & Nickerson, B. G. (2005). A deterministic skip list for k-dimensional range search. *Acta informatica*, 41(4), 221-255.

Larsen, K. S. (2000). AVL trees with relaxed balance. *Journal of Computer and System Sciences*, 61(3), 508-522.

Manber, U. (1984). Concurrent maintenance of binary search trees. *IEEE transactions on software engineering*, (6), 777-784.

Martínez, C., & Roura, S. (1998). Randomized binary search trees. *Journal of the ACM (JACM)*, 45(2), 288-323.

Mellor-Crummey, J. M., & Scott, M. L. (1991). Algorithms for scalable synchronization on shared-memory multiprocessors. *ACM Transactions on Computer Systems (TOCS)*, 9(1), 21-65.

Messeguer, X. (1997). Skip trees, an alternative data structure to skip lists in a concurrent approach. *Informatique Theorique et applications*, 31(3), 251-269.

Munro, J. I., Papadakis, T., & Sedgewick, R. (1992, September). Deterministic skip lists. *In Proceedings of the third annual ACM-SIAM symposium on Discrete algorithms*, (pp. 367-375). Society for Industrial and Applied Mathematics.

Nurmi, O., & Soisalon-Soininen, E. (1996). Chromatic binary search trees. *Acta informatica*, 33(6), 547-557.

Park, H., & Park, K. (2001). Parallel algorithms for redblack trees. *Theoretical Computer Science*, 262(1-2), 415-435.

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

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

Seidel, R., & Aragon, C. R. (1996). Randomized search trees. *Algorithmica*, 16(4), 464-497.

Shavit, N., & Touitou, D. (1997). Software transactional memory. *Distributed Computing*, 10(2), 99-116.

Tarjan R. (2016). *Zip Trees*. Unpublished Manuscript

Varshneya, A., Madan, B. B., & Balakrishnan, M. (1994, April). Concurrent search and insertion in K-dimensional height balanced trees. *In Parallel Processing Symposium, 1994. Proceedings., Eighth International*, (pp. 883-887). IEEE.