Bottom-up Rebalancing Binary Search Trees by Flipping a Coin

Gerth Stølting Brodal
Department of Computer Science, Aarhus University, Aabogade 34, 8200 Aarhus N, Denmark

Abstract

Rebalancing schemes for dynamic binary search trees are numerous in the literature, where the goal is to maintain trees of low height, either in the worst-case or expected sense. In this paper we study randomized rebalancing schemes for sequences of n insertions into an initially empty binary search tree, under the assumption that a tree only stores the elements and the tree structure without any additional balance information. Seidel (2009) presented a top-down randomized insertion algorithm, where insertions take expected $O\left(\lg^2 n\right)$ time, and the resulting trees have the same distribution as inserting a uniform random permutation into a binary search tree without rebalancing. Seidel states as an open problem if a similar result can be achieved with bottom-up insertions. In this paper we fail to answer this question.

We consider two simple canonical randomized bottom-up insertion algorithms on binary search trees, assuming that an insertion is given the position where to insert the next element. The subsequent rebalancing is performed bottom-up in expected $O(1)$ time, uses expected $O(1)$ random bits, performs at most two rotations, and the rotations appear with geometrically decreasing probability in the distance from the leaf. For some insertion sequences the expected depth of each node is proved to be $O(\lg n)$. On the negative side, we prove for both algorithms that there exist simple insertion sequences where the expected depth is $\Omega(n)$, i.e., the studied rebalancing schemes are *not* competitive with (most) other rebalancing schemes in the literature.

2012 ACM Subject Classification Theory of computation → Data structures design and analysis

Keywords and phrases Binary search tree, insertions, random rebalancing

Digital Object Identifier 10.4230/LIPIcs.FUN.2024.25

Category FUN with Algorithms

Related Version *Full paper*: `https://doi.org/10.48550/arXiv.2404.08287` [12]

Funding *Gerth Stølting Brodal*: Supported by Independent Research Fund Denmark, grant 9131-00113B.

1 Introduction

Binary search trees is one of the most fundamental data structures in computer science, dating back to the early 1960s, see, e.g., Windley (1960) [28] for an early description of binary search trees and Hibbard (1962) [20] for an analysis of random insertions and deletions. Knuth [21, page 453] gives a detailed overview of the early history of binary search trees, and Andersson *et al.* [7] an overview of later developments on balanced binary search trees.

When inserting new elements into the leaves of an unbalanced binary search tree the height of the tree might deteriorate, in the sense that it becomes super-logarithmic in the number of elements stored (see Figure 1). In the literature numerous rebalancing schemes have been presented guaranteeing logarithmic height: Some are deterministic with worst-case update bounds, like AVL-trees [1], red-black trees [19]; some deterministic with amortized bounds, like splay-trees [26] and scapegoat trees [5, 18]; and others are randomized, like treaps [25] and randomized binary search trees [22], just to mention a few.

In this paper we study simple randomized rebalancing schemes for sequences of insertions

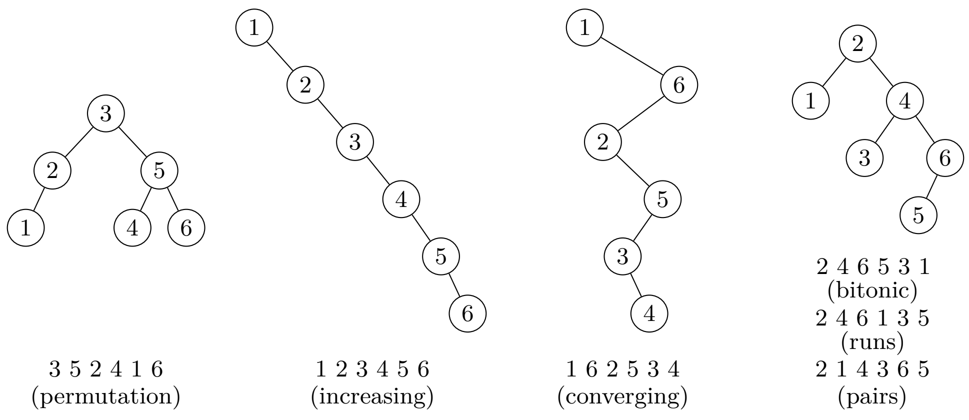

Figure 1 Unbalanced binary search trees resulting from inserting permutations of $\{1,\ldots,6\}$. The insertion order is shown below the trees. The types of permutations are defined in Table 2.

into an initially empty binary search tree. The goal of this paper is to study randomized rebalancing schemes under a set of constraints, and to study how good rebalancing schemes can be achieved within these constraints. In general the proposed rebalancing schemes in Section 2 and Section 3 are *not* competitive with existing rebalancing schemes in the literature. The constraints we consider are the following:

1. The search tree should not store any balancing information, only the tree and the elements should be stored.
2. Insertions should perform limited restructuring, say, worst-case $O(1)$ rotations.
3. Most rotations should happen near the inserted elements.
4. Rebalancing should be based on local information (tree structure) at the insertion point only (e.g., without knowledge of n nor the current height of the tree).
5. Rebalancing should be performed in expected $O(1)$ time.
6. Rebalancing should use expected few random bits per insertion, say, expected $O(1)$ bits.
7. Each node should have low expected depth, ideally $O(\lg n)$.

The constraints are motivated by the properties of the randomized treaps [25] (but treaps need to store random priorities as balance information); that the random distribution of tree structures achieved by treaps can be achieved without storing balancing information [24] (but with slower insertions); and treaps can be the basis for efficient concurrent search trees [3].

1.1 Deterministic Previous Work

Red-black trees [19] are deterministic dynamic balanced binary search, with good amortized performance. They violate constraint (1), since each node is required to store a single bit of balance information, indicating if the node is red-black. But otherwise, red-black trees are guaranteed to have height $O(\lg n)$, insertions at a leaf can be performed in amortized $O(1)$ time and perform at most two rotations, i.e., red-black trees essentially satisfy constraints (2)–(7), if expected bounds are substituted by amortized bounds.

Brown [13, 14] showed how to encode a single bit of information in the internal nodes of a binary tree by considering "supernodes" consisting of pairs of consecutive elements arranged as parent-child pairs together with a pointer to an empty leaf between the two elements. Depending of the relative placement of the two elements the encoded bit can be decoded from the placement of the pointer to the empty leaf. Brown showed how to encode 2-3 trees [2, Chapter 4] using this technique, achieving balanced binary search trees storing no balance information and supporting insertions in worst-case $O(\lg n)$ time.

Splay trees [26] are the canonical deterministic amortized efficient dynamic binary search trees satisfying constraint (1), i.e., they do not store any additional information than the binary tree and the elements. Splay trees support insertions in amortized $O(\lg n)$ time, i.e., insertions are amortized efficient. The drawback of splay trees is that they do a significant amount of restructuring (memory updates) per insertion, since they rotate a constant fraction of the nodes on the path from the inserted element to the root. The number of rotations depends on what variant of splaying is applied, see [11, 26]. Albers and Karpinski [4] and Fürer [17] considered randomized variants of splaying to reduce the restructuring cost.

Scapegoat trees are another deterministic dynamic binary search tree with good amortized performance, independently discovered by Anderson [5, 6] and Galperin and Rivest [18]. Scapegoat trees achieve amortized $O(\lg n)$ insertions by maintaining the invariant that the height of a tree containing n elements is $O(\lg n)$. If an insertion causes the invariant to be violated, a local subtree is rebuild into a perfectly balanced binary tree (in the worst-case this is the root and the full tree is rebuild). A scapegoat tree only needs to store the number of elements n as a global integer value in addition to the binary tree and its elements.

1.2 Randomized Previous Work

Randomization and binary search trees can be addressed in two directions in the context of insertions: Either insertions are random (e.g., the insertion sequence is a random permutation) and we analyze the expected performance for a binary search tree with respect to the insertion distribution; or insertions can be arbitrary but the rebalancing of the binary search tree exploits random bits and we analyze the performance with respect to the random bits.

We call a search tree containing n elements inserted in random order without rebalancing a *random binary search tree*. A classic result on random binary search trees is that each element in the resulting tree has expected depth at most $2 \ln n + O(1)$ [10, 20, 28]. The important property is that the root equals each of the n elements with probability exactly $1/n$, and this property again holds recursively for the left and right subtrees. A consequence is that all valid search trees with $n \geq 3$ elements do not have the same probability. See Panny [23] for a history on deletions in random binary search trees.

The structure of random binary search trees has been used as guideline to construct different dynamic binary search trees where at each point of time the probability of a given tree equals that of a random binary search tree [22, 24, 25].

Aragon and Seidel [25] introduced the *treap*, that with each element stores an independently uniformly assigned random priority in the range $[0, 1]$, and organizes the search tree such that priorities satisfy heap order [27], i.e., the root stores the element with minimum priority. Since each element has probability $1/n$ to have the smallest priority, all elements have probability $1/n$ to be at the root, the property required to be random search trees. Insertions into treaps can be done by bottom-up rotations in expected $O(1)$ time and $O(1)$ rotations using $O(1)$ random bits. After n insertions into a treap the expected shape of the treap equals a random binary search tree. Blelloch and Reid-Miller [9] considered parallel algorithms for the set operations union, intersection and difference on treaps. Alapati *et al.* [3] considered concurrent insetions and deletions into treaps.

Martínez and Roura [22] presented a different approach denoted *randomized binary search trees* to achieve the structure of a random binary search tree after n insertions. Their approach stores at each node the size of the subtree rooted at the node, and insertions are performed top-down in expected $O(\lg n)$ time, where the inserted element is inserted in a node with

Table 1 Rebalancing cost of selected binary search trees. Space refers to the space required for additional balance information. O_A, O_E and O denote amortized, expected and worst-case bounds, respectively. *Our results do not guarantee (expected) logarithmic depth.

	Time	Rotations	Random bits	Space (bits)
Red-black tree [19]	$O_A(1)$	$O(1)$	0	1
Encoded 2-3 trees [13, 14]	$O(\lg n)$	$O(\lg n)$	0	0
Splay trees	$O_A(\lg n)$	$O_A(\lg n)$	0	0
Treaps [8]	$O_E(1)$	$O_E(1)$	$O_E(1)$	$O_E(1)$
Randomized BST [22]	$O_E(\lg n)$	$O_E(1)$	$O_E(\lg^2 n)$	$O(\lg n)$
Seidel [24]	$O_E(\lg^2 n)$	$O_E(1)$	$O_E(\lg^3 n)$	0
*Algorithms in this paper**	$O_E(1)$	$O(1)$	$O_E(1)$	0

Table 2 Different sequences of length n (assuming n is even) considered in this paper.

permutation	random permutation of $1, \ldots, n$
increasing	$1, 2, 3, \ldots, n$
decreasing	$n, n-1, n-2, \ldots, 1$
converging	$1, n, 2, n-1, 3, n-2, \ldots, \frac{n}{2}, \frac{n}{2}+1$
pairs	$2, 1, 4, 3, 6, 5, \ldots, n, n-1$
bitonic	$2, 4, 6, \ldots, n-2, n, n-1, n-3, \ldots, 5, 3, 1$
runs	$2, 4, 6, \ldots, n-2, n, 1, 3, 5, \ldots, n-3, n-1$

probability $\frac{1}{k+1}$, where k is the size of the current subtree rooted at the node (see [22] for details). Each insertion requires expected $O(\lg n)$ random integers in the range $1, \ldots, n+1$.

Seidel [24] gave a unified presentation of [25] and [22], emphasizing the similarity of the two approaches, and describes a variation of [22] that avoids storing subtree sizes, but the insertion time increases to expected $O(\lg^2 n)$ and uses $O(\lg^3 n)$ random bits. Seidel states it as an open problem if there exists a bottom-up rebalancing algorithm that without storing any balancing information can obtain the structure of random binary search trees.

1.3 Results

We consider two very simple algorithms to rebalance a binary search tree after a new element has been inserted at a leaf. Our aim is to try to meet the requirements (1)–(7), and in particular not the ambitious goal of having the same distribution as random binary search trees. Both our algorithms repeatedly flip a coin until it comes out head. Whenever the coin shows tail (with probability p) we move to the parent of the current node (starting at the new leaf, and if we reach the root, the rebalancing terminates without modifying the tree). When the coin shows head, the first algorithm (RebalanceZig in Algorithm 1) rotates the current node up, and the rebalancing terminates. The second algorithm (RebalanceZigZag in Algorithm 2) does one or two rotations, depending on if it is a zig-zag or zig-zig case (inspired by the rebalancing rules of splay trees).

Ignoring the depths of the nodes of the resulting trees, we immediately have the following fact, since the coin tosses are independent Bernoulli trials, with an expected $O(1)$ coin tosses necessary (assuming a coin with constant non-zero probability for head). It follows that both algorithms satisfy our constraints (1)–(6).

▶ **Fact 1.** *The rebalancing done by* RebalanceZig *with* $0 \le p < 1$ *takes expected* $O(1)$ *time, uses expected* $O(1)$ *random bits, and performs at most one rotation.* RebalanceZigZag *performs at most two rotations, but otherwise with identical performance.*

To study to what extend the proposed algorithms achieve logarithmic depth of the nodes, constraint (7), we study the behavior of the algorithms on the insertion sequences listed in Table 2.

In Section 2 we study RebalanceZig. Our first result is that RebalanceZig is sufficient to achieve a balanced binary search tree when inserting elements in increasing order (and symmetrically decreasing order). The below theorem follows from Lemma 9.

▶ **Theorem 1.** *Executing* RebalanceZig *with* $0 < p < 1$ *on an increasing and decreasing sequence of* n *insertions results in a binary search tree, where each node has expected depth* $O(\lg n)$*.*

We then show that there are very simple insertion sequences where RebalanceZig fails to achieve a balanced tree. We denote the sequence $1, n, 2, n-1, \ldots, n/2, n/2+1$ the *converging* sequence. The below theorem follows from Lemma 11.

▶ **Theorem 2.** *Executing* RebalanceZig *with* $0 \leq p \leq 1$ *on a converging sequence of* n *insertions results in a binary search tree with expected average node depth* $\Theta(n)$*.*

Another sequence where RebalanceZig fails to achieve logarithmic depth is on the *pairs* sequence $2, 1, 4, 3, 6, 5, \ldots, n, n-1$, provided $p \neq \frac{1}{2}$. The following theorem restates Lemma 12.

▶ **Theorem 3.** *Executing* RebalanceZig *with* $0 \leq p < \frac{1}{2}$ *or* $\frac{1}{2} < p \leq 1$ *on a pairs sequence of* n *insertions results in a binary search tree with expected average node depth* $\Theta(n)$*.*

For $p = \frac{1}{2}$ algorithm RebalanceZig behaves significantly better on pairs sequences. We conjecture the expected average node depth to be $O\left(\sqrt{n}\right)$.

In Section 3 we study the second algorithm RebalanceZigZag. For increasing (decreasing) sequences, where the new leaf is always the rightmost (leftmost) node in the tree, RebalanceZigZag is essentially identical to RebalanceZig, i.e., our result for increasing and decreasing sequences for RebalanceZig immediately carries over to RebalanceZigZag. The following theorem restates Corollary 14.

▶ **Theorem 4.** *Executing* RebalanceZigZag *with* $0 < p < 1$ *on an increasing or decreasing sequence of* n *insertions results in a binary search tree where each nodes has expected depth* $O(\lg n)$*.*

In Section 3.2 we generalize the proof to also hold for the convergent sequence for RebalanceZigZag (where RebalanceZig failed to achieve logarithmic depth), and more generally finger sequences, where the next insertion always becomes the successor or predecessor of the last insertion. The following theorem restates Lemma 15.

▶ **Theorem 5.** *Executing* RebalanceZigZag *with* $\frac{1}{2}\left(\sqrt{5}-1\right) < p < 1$ *on a convergent or finger sequence of* n *insertions results in a binary search tree where each nodes has expected depth* $O(\lg n)$*.*

On the negative side, we prove that RebalanceZigZag fails to achieve balanced trees for pairs sequences, for all $0 \leq p \leq 1$. The following theorem restates Lemma 16.

▶ **Theorem 6.** *Executing* RebalanceZigZag *with* $0 \leq p \leq 1$ *on a pairs sequence of* n *insertions results in a binary search tree with expected average node depth* $\Theta(n)$*.*

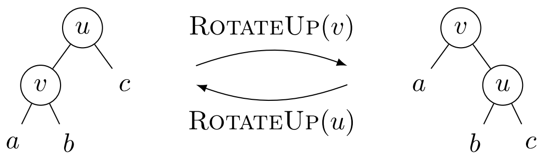

Figure 2 (Left-to-right) The right rotation of u rotates v up; note that a is moved one level up in the tree, b remains at the same level, and c is moved down one level. (Right-to-left) the left rotation of v rotates u up.

We complement our theoretical findings by an experimental evaluation of REBALANCEZIG and REBALANCEZIGZAG in Section 5, supporting our theoretical findings. We briefly discuss random permutations in Section 4, but otherwise only have an experimental evaluation of the rebalancing algorithms on inserting random permutations.

If the insights from our results can lead to an improved bottom-up randomized rebalancing scheme for binary search trees remains open.

1.4 Notation and Terminology

Throughout this paper n denotes the number nodes in a binary search tree, i.e., the number of insertions performed. The *depth* of a node is the number of edges from the node to the root, i.e., the root has depth zero. The height of a tree is the maximum depth of a node. Rebalancing will be done by the standard primitives of left and right *rotations*, see Figure 2. Both rotate up a node one level in the tree. Since our updates are performed bottom-up, we assume that each node v stores an element and pointers to its left child $v.l$, right child $v.r$, and parent $v.p$ (possibly equal to NIL if no such node exists). We let $\lg n$ and $\ln n$ denote the binary and natural logarithm of n, respectively.

2 Algorithm REBALANCEZIG

In this section we show that on increasing and decreasing sequences applying algorithm REBALANCEZIG results in binary search trees where each node has expected depth $O(\lg n)$. We also show that on the converging and pairs sequences $\left(p \neq \frac{1}{2}\right)$ the expected average node depth is linear.

Assume a new element has been inserted into a binary search tree as a new leaf v (before rebalancing the tree). Algorithm REBALANCEZIG rebalances the tree as follows: After inserting the new node v, we flip a coin, that with probability p is tail and $1-p$ is head, for a constant $0 \leq p \leq 1$. If the coin is head, we rotate v up, and the insertion terminates. Otherwise, we recursively move to the parent, i.e., set $v \leftarrow v.p$, flip a coin, and rotate the parent up if the coin is head, or continue recursively at the grandparent if the coin is tail. The rebalancing terminates when the first rotation has been performed or when we reach the root. See the pseudo-code in Algorithm 1.

Note that $p = 1$ is the special case where we always move up and never rotate, i.e., identical to insertions without rebalancing. When $p = 0$ the new node is always the node rotated up. In this case the tree is a single path containing all n nodes, since inserting a node v as a child of u on the path, rotating up v causes v to be inserted into the path as the parent u. See Figure 3. In the following we assume $0 < p < 1$. That REBALANCEZIG can not achieve the same tree distribution as random binary search trees (like treaps and randomize binary search trees do) follows by the example in Figure 4.

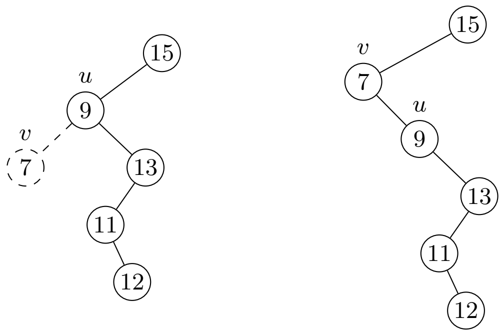

Figure 3 REBALANCEZIG with $p = 0$ always rotates up the inserted node, and maintains the invariant that the tree is a single path. (left) insertion point of 7; (right) 7 is rotated up onto the path.

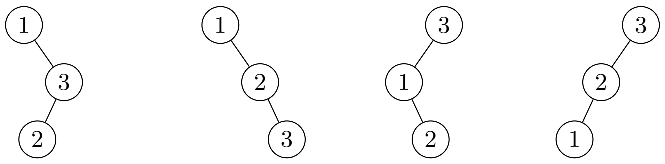

Figure 4 Binary search trees resulting from inserting the sequence 1, 3, 2 using REBALANCEZIG. Each of the four search trees has probability 1/4 when $p = 1/2$. Note that the perfect balanced binary search tree on three nodes cannot by achieved REBALANCEZIG on this insertion sequence.

2.1 Increasing Sequences

We let the *right height* of a tree denote the depth of the rightmost node in the tree. When inserting elements in increasing order the new element will always be inserted as a rightmost node in the tree, i.e., the right height increases by one before any rebalancing is performed. If REBALANCEZIG performs a (left) rotation on the rightmost path, the right height is reduced by one again, and the right height does not change by the insertion.

▶ **Lemma 7.** *If the right height is d before inserting a new rightmost node and applying* REBALANCEZIG*, then afterwards the right height is d or $d + 1$ with probability $1 - p^{d+1}$ or p^{d+1}, respectively.*

Proof. The right height increases if and only if no rotation is performed, i.e., we reach the root because the coin $d+1$ times in a row shows tail, which happens with probability p^{d+1}. ◀

▶ **Lemma 8.** *After inserting n elements in increasing order using* REBALANCEZIG*, the right height is at most $\lceil (c + 1) \cdot \log_{1/p} n \rceil$ with probability $1 - 1/n^c$, for any constant $c > 0$.*

Proof. Assume that the right height at some point of time during the insertions is $d = c' \cdot \lg n$. By Lemma 7, the probability that the next insertion increases the right height is p^{d+1}. The probability of any of the at most n remaining insertions increases the right height is at most $np^{d+1} = np^{1+c' \lg n} \le np^{c' \lg n} = n^{1+c' \lg p} \le n^{-c}$ for $c' \ge -(c + 1)/\lg p$. It follows that the right height after n insertions is at most $\lceil -(c + 1)/\lg p \cdot \lg n \rceil = \lceil (c + 1) \log_{1/p} n \rceil$ with probability $1 - 1/n^c$. ◀

Lemma 8 gives a high probability guarantee on the expected depth of the nodes on the rightmost path. We now prove an expected depth for all nodes in the tree.

Algorithm 1 REBALANCEZIG(v)

```
while $v.p \neq$ NIL and coin flip is tail do
    $v \leftarrow v.p$
if $v.p \neq$ NIL then
    ROTATEUP($v$)
```

▶ **Lemma 9.** *After inserting n elements in increasing order using* REBALANCEZIG*, with $0 < p < 1$, each node has expected depth $O\big(1/p \cdot \log_{1/p} n\big)$.*

Proof. Consider an element inserted in a node v during the sequence of insertions. The element goes through the following five phases:

1. The element is not yet inserted. The less than n elements inserted before v create a tree with right height $O\big(\log_{1/p} n\big)$ with high probability (Lemma 8).
2. v is created as the rightmost node with depth $O\big(\log_{1/p} n\big)$ with high probability.
3. v remains on the rightmost path for the subsequent insertions until v's right child u becomes the target for being rotated up. An insertion that rotates up v or an ancestor of v will decrease the depth of v. Rotations below u do not change the depth of v.
4. v is moved out of the rightmost path by rotating up the right child u of v, making v the left child of u. This increases the depth of v by one.
5. v is in the left subtree of a node u on the rightmost path (u can change over the subsequent insertions, but the depth of the branching node u can never increase). Each insertion can affect the position of v by the rotation performed:
   a. No rotation is performed and the path from the root through u to v is unchanged. The right height increases by one.
   b. An ancestor of u is rotated up, where u remains the branching node to v, the depths of both u and v decrease by one.
   c. u is rotated up, where u remains the branching node to v, the depth of u decreases by one, and the depth of v stays unchanged.
   d. Rotating up the right child w of u increases the depth of v by one and w replaces u as the branching node to v on the rightmost path (with the same depth as u had before the rotation).
   e. Rotations below the right child of u do not change the path from the root through u and v.

From Lemma 8 it follows that the depth of v after phases 1–4 is $O\big(\log_{1/p} n\big)$, with high probability. Cases 5a, 5b, 5c and 5e do not increase the depth of v. What remains is to bound the expected number of times case 5d occurs and increases the depth of v by one. For case 5d to happen, a coin must have been flipped at w showing head. Over all insertions in phase 5, a subsequence of the insertions flips a coin at the child w of the current branching node u. If an insertion flips a coin at w, there are two cases: The coin shows head with probability $1 - p$ and case 5d happens; or the coin shows tail with probability p, and case 5a, 5b or 5c happens. Since cases 5a, 5b and 5c at most happens $O\big(\log_{1/p} n\big)$ times with high probability (case 5a increases the right height; cases 5b and 5c decrease the depth of the branching node u to v), i.e., the coin shows tail at w at most $O\big(\log_{1/p} n\big)$ times with high probability. Since the expected number of times we need to flip a coin to get a tail is $1/p$, the expected number of times we flip a coin at the right child w of the branching node u to v is $O\big(1/p \cdot \log_{1/p} n\big)$, with high probability. This is then also an upper bound on the expected number of times the depth of v can increase by case 5d. It follows that with high probability, the expected depth of v is $O\big(\log_{1/p} n + 1/p \cdot \log_{1/p} n\big) = O\big(1/p \cdot \log_{1/p} n\big)$. Since the depth of v is at most $n-1$, the expected depth of v is $O\big(1/p \cdot \log_{1/p} n\big)$ after all insertions is (without the high probability assumption). ◀

The following lemma states that the node rotated up is expected to be close to the inserted leaf, and states the number of coin flips as a function of the tail probability p.

▶ **Lemma 10.** *The distance from the inserted node to the node rotated up by* RebalanceZig *is expected at most* $\frac{p}{1-p}$*. The number of coin flips is at most* $\frac{1}{1-p}$*.*

Proof. The expected distance to the node rotated up is at most

$$\sum_{d=0}^{\infty} d(1-p)p^d = (1-p)\sum_{d=0}^{\infty} dp^d = (1-p)\frac{p}{(1-p)^2} = \frac{p}{1-p} \ ,$$

since the new node inserted (at distance 0) is rotated up with probability $1-p$, its parent with probability $p(1-p)$, etc. The d'th ancestor is rotated up with probability $(1-p)p^d$, provided a d'th ancestor exists. The number of coin flips is one plus the distance, i.e., at most $1 + \frac{p}{1-p} = \frac{1}{1-p}$. ◀

Note that inserting n elements in decreasing order is the symmetric case to the increasing order where the new node is inserted as the leftmost node, and at most one right rotation is performed on the leftmost path. If follows that Lemmas 8 and 10 also apply to decreasing sequences, by replacing the rightmost path by the leftmost path in the arguments,

2.2 Converging Sequences

Assume we have a *finger* into the sorted inserted sequence of elements pointing to the most recently inserted element, and whenever a new element is inserted it must be the new predecessor or successor of the element at the finger, i.e., we can only insert elements that are in the interval defined by the current predecessor and successor of the element at the finger. We call sequences satisfying this property for *finger insertions*. Increasing and decreasing sequences are examples of finger insertions.

The following sequence of insertions also consists of finger insertions (for simplicity, we assume n is even). We denote this insertion sequence the *converging sequence*. See Figure 1 for an illustration of $n = 6$.

$$1, n, 2, n-1, 3, n-2, 4, n-3, \ldots, n/2, n/2+1$$

As can be seen in the experimental evaluation in Figure 7(a,b), the average depth appears to be linear for the nodes in a binary search tree resulting from applying RebalanceZig to the converging sequence. The below lemma confirms this.

▶ **Lemma 11.** *Executing* RebalanceZig *with insertions* $1, n, 2, n-1, \ldots, n/2, n/2+1$, *assuming* n *even, results in a binary search tree with an (external) leaf with expected depth at least* $\frac{p(1-p)}{2}n$*, for* $0 < p < 1$*. For* $p = 0$ *and* $p = 1$ *the resulting tree is a single path.*

Proof. For $p = 1$ we do no rebalancing, and the converging sequence results in a single path (see Figure 1). For $p = 0$, the new node is always rotated up onto an existing single path. We let the *insertion point* denote the (external) leaf, where the next insertion is going to create a node v. Since the converging sequence is a sequence of finger insertions, the next insertion point is always a child of the created node v (before rebalancing), i.e., each insertion increases the depth of insertion point by one (before rebalancing). Unfortunately, the rebalancing done

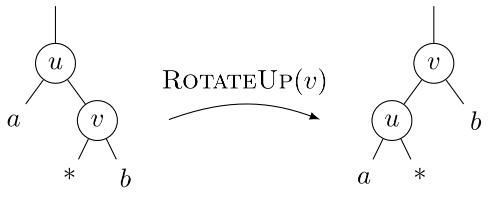

Figure 5 Bad zag-zig case for REBALANCEZIG(v), where rotating up v does not decrease the depth of the insertion point $*$.

by REBALANCEZIG does not always decrease the depth one. Consider inserting i, where $1 \leq i \leq n/2$, that creates a node u followed by inserting $n+1-i$ that creates a node v. Assume that the rebalancing after inserting i does not rotate up u (but possibly an ancestor of u has been rotated up, and possibly decreasing the depth of the insertion point by one again), and the insertion of $n+1-i$ causes v to be rotated up. This case is shown in Figure 5. We borrow the terminology from splay trees that if a path branches left, we say it is a zig, and if it branches right it is a zag. If a left branch is followed by a right branch it is a zig-zag. We denote the case in Figure 5 the zag-zig case. In this case the insertion point moves from being a child of v to being a child of u, but retains the same depth, i.e., the insertions and REBALANCEZIG increased the depth of the insertion point by one. The probability that u was not rotated up is p and the probability that v is rotated up is $1-p$, i.e., the insertion of i and $n+1-i$ causes the depth of the insertion point to increase by one with probability at least $p(1-p)$. It follows that after the insertion of all n elements, the expected depth of the insertion point is at least $\frac{1}{2}np(1-p)$. ◀

If an external leaf has depth d, then the sum of the depths of the d internal nodes on the path to the external leaf is $\sum_{i=0}^{d-1} i = \frac{1}{2}d(d-1)$. The average depth of all nodes in the tree is then at least $\frac{1}{2}d(d-1)/n$, and Theorem 2 follows from Lemma 11.

It should be noted that the rotation after an insertion can happen higher in the tree, where there is also a zag-zig case or symmetric zig-zag case, where REBALANCEZIG also fails to decrease the depth of the insertion point after the insertion. This explains the gap between the experimental constant observed in Figure 7 and the theoretical analysis.

2.3 Pairs Sequences

The pairs sequence consists of $2, 1, 4, 3, 6, 5, \ldots, n, n-1$. It is essentially an increasing sequence, with pairs $2i-1$ and $2i$ swapped. Pair sequences are not finger sequences. Interestingly, experiments show that REBALANCEZIG is challenged by this sequence. In our experimental evaluation, Figure 7(a), it appears that $p = \frac{1}{2}$ is a local minima for the average node depth when rebalancing pairs sequences using REBALANCEZIG, with increased average node depth for both p smaller than and larger than $\frac{1}{2}$. In [12] we prove the following lemma.

▶ **Lemma 12.** *Applying* REBALANCEZIG *to the pairs sequence with n elements, for n even and constant $p \neq \frac{1}{2}$, $0 \leq p \leq 1$, the resulting tree has expected average node depth $\Theta(n)$.*

The last inserted element has expected depth $\Theta(n)$ for $0 < p < \frac{1}{2}$ and $O(1)$ for $\frac{1}{2} < p < 1$, so Lemma 12 does not give any bounds on the expected depth of specific elements. Lemma 12 addresses pairs sequences for $p \neq \frac{1}{2}$, where the expected average node depth is linear. For $p = \frac{1}{2}$ we give the following conjecture, stating that the complexity is significantly different. See [12] for an experimental and theoretical motivation of the conjecture.

▶ **Conjecture 13.** *Applying* RebalanceZig *to the pairs sequence with n elements, for n even and $p = \frac{1}{2}$, the resulting tree has expected average node depth $O(\sqrt{n})$.*

3 Algorithm RebalanceZigZag

To address the shortcomings of algorithm RebalanceZig in the case where v is in a zig-zag or zag-zig state, we borrow terminology from splay trees [26], and apply the zig-zag transformation to the tree (see Figure 6(right) and [26, Figure 3]) by rotating up v twice. In the zig-zig case, instead of rotating up v, we rotate up the parent of v (see Figure 6(left)). These two transformations ensure that everything in the subtree of v is moved one level up in the tree when applying the transformantion at node v. The pseudo-code for algorithm RebalanceZigZag is shown in Algorithm 2.

Algorithm 2 RebalanceZigZag(v)

```
while $v.p \neq$ NIL and coin flip is tail do
  $v \leftarrow v.p$
if $v.p \neq$ NIL and $v.p.p \neq$ NIL then
  if ($v = v.p.l$ and $v.p = v.p.p.l$) or ($v = v.p.r$ and $v.p = v.p.p.r$) then
    RotateUp($v.p$)                                  ▷ zig-zig or zag-zag case
  else
    RotateUp($v$)                                    ▷ zig-zag or zag-zig case
    RotateUp($v$)
```

3.1 Increasing Sequences

RebalanceZigZag handles increasing and decreasing sequences identical to Rebalance-Zig, except that rotations happen one level higher, i.e., the trees are identical if ignoring the rightmost inserted node. Equivalently, this corresponds to inserting the next element without rebalancing, and first performing the rebalancing just before inserting the next element. From Theorem 1 we have the following corollary.

▶ **Corollary 14.** *For $0 < p < 1$, after inserting n elements in increasing or decreasing order using* RebalanceZigZag*, each node has expected depth $O(1/p \cdot \log_{1/p} n)$.*

3.2 Finger Sequences

We will prove that using RebalanceZigZag to rebalance finger sequences (like increasing, decreasing and converging sequences), ensures that the resulting tree is expected to be balanced, for p sufficiently large. Recall that a finger sequence is defined such that the next element is always the immediate predecessor or successor of the most recently inserted element among all elements inserted so far. In an unbalanced search tree, this means that the next node will be a (left or right) child of the most recently inserted node, i.e., the resulting tree is always a path. We denote the external leaf where to create the next node the *insertion point*. The crucial property of the restructuring done by RebalanceZigZag in Figure 6 is that if the insertion point is at an external leaf in the subtree rooted at v before the rotation, then the depth of this external leaf is reduced by exactly one in both cases.

▶ **Lemma 15.** *After inserting a finger sequence with n elements using* RebalanceZigZag*, each node has expected depth $O(\lg n)$, for $\frac{1}{2}(\sqrt{5}-1) < p < 1$.*

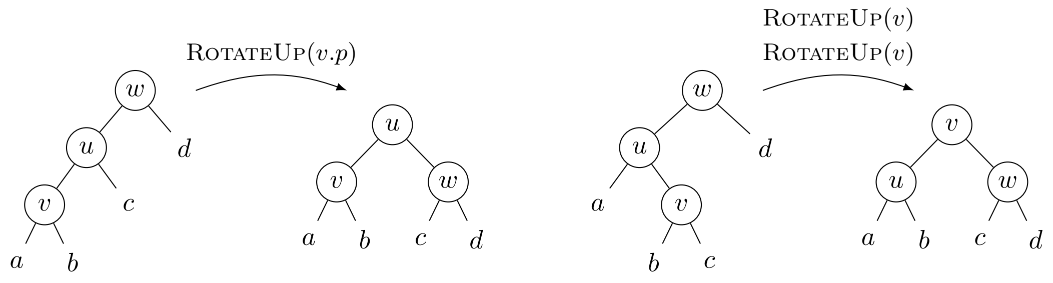

Figure 6 The rebalancing performed by algorithm RebalanceZigZag(v) in (left) zig-zig case and (right) zig-zag case.

Proof. The proof follows the same idea as in Section 2.1 for the analysis of RebalanceZig on increasing sequences. Instead of right height we consider *insertion depth*, i.e., the depth d of the parent node of the insertion point. See [12] for proof details. ◀

In our experiments, see Figure7(c), it shows up that RebalanceZigZag performs better for $p > \frac{1}{2}$ than $p < \frac{1}{2}$ on the converging sequence, that is an example of a finger sequence. We leave open the question what the dependency on p is for RebalanceZigZag for $0 < p \leq \frac{1}{2}\left(\sqrt{5} - 1\right)$.

3.3 Pairs Sequences

While RebalanceZigZag achieves better average node depth on converging sequences compared to RebalanceZig, it fails on pairs sequences, where the expected average node depth is linear for all values $0 \leq p \leq 1$ (for $p = \frac{1}{2}$ this is worse than RebalanceZig, if our conjecture turns out to be true). In [12] we prove the following lemma.

▶ **Lemma 16.** *Applying* RebalanceZigZag *to the pairs sequence with n elements, for n even and $0 \leq p \leq 1$, the resulting tree has expected average node depth $\Theta(n)$.*

4 Random Permutations

We do not prove anything for random permutations, for neither RebalanceZig nor RebalanceZigZag. If $p = 1$ no rebalancing is performed, and it is known that the expected depth of a node is $O(\lg n)$ [10, 20, 28], whereas for $p = 0$, a rotation is always performed at the inserted leaf, and the tree will always be a single path. How exactly the average node depth depends on p is an open problem. In the experiments, see [12], it appears that RebalanceZigZag is "about" logarithmic for $p \geq 0.7$.

5 Experimental Evaluation

In this section we present an experimental evaluation of our algorithms RebalanceZig and RebalanceZigZag. See [12] for more experimental results.

We implemented the algorithms in Python 3.12, and ran the algorithms with different choices of p on the types of insertion sequences listed in Table 2 with sequence lengths being powers of two. Each data point in Figure 7 is the average over 25 runs. The increasing, decreasing and converging sequences are examples of finger insertions. Inserting the pairs, bitonic and runs sequences into a search tree without rebalancing result in identical search trees (see Figure 1). Note that the first half of the bitonic sequence is an increasing sequence, whereas the second part evenly distributes the remaining elements into the created leaves right-to-left. The runs sequences is identical to the bitonic sequences, except that the second part is performed left-to-right.

Figure 7 shows our main experimental findings. It shows the resulting average node depths of running the algorithms on the different types of insertion sequences from Table 2 with insertion sequences of length between two and 1024 and various values of p in the range zero to one. Note that for a path with n nodes, the root has depth 0 and the bottommost node depth $n-1$, i.e., the average depth is $\frac{1}{n}\sum_{d=0}^{n-1} = \frac{1}{n} \cdot \frac{n(n-1)}{2} = \frac{n-1}{2}$. For $n = 1024$, an average node depth of 511.5 implies that the tree is a path. In Figure 7(a) this explains why all curves share the top-left point, since RebalanceZig always generates a path when $p = 0$, independent of the insertion sequence, as discussed in Section 2. In Figure 7(left) the rightmost data point ($p = 1$) for random permutations corresponds to the average node depth in unbalanced binary search trees. Note that the pairs, bitonic and runs insertion sequences end up with different average node depth characteristics for each of the three algorithms (dashed curves in Figure 7), even that they would generate the same trees without rebalancing.

Figure 7(a, b) clearly shows that RebalanceZig has problems with the converging sequences (consistent with Theorem 2); Figure 7(c, d) that RebalanceZigZag has problems with the pairs sequences (consistent with Theorem 6).

6 Conclusion and Open Problems

This paper leaves more open problems than it solves. None of the considered randomized rebalancing algorithms meets all conditions (1)–(7) introduced in Section 1. Inspired by a question raised by Seidel [24], we considered bottom-up randomized rebalancing schemes for binary search trees without storing any balance information. We studied randomized rebalancing strategies, inspired by the rebalancing primitives from splay trees [26]. They meet conditions (1)–(6), but fail to achieve logarithmic depth on all insertion sequences. In the experiments RebalanceZigZag appears often to have the best performance, although it provably does not achieve expected logarithmic average depth for all insertion sequences. It remains an open problem if a randomized bottom-up rebalancing scheme exists that can guarantee expected logarithmic average node depths for all insertion sequences and satisfies requirements (1)–(6), or what the best depth guarantee can be given requirements (1)–(6), or how much these requirements need to be relaxed to enable expected logarithmic average node depths.

We did not consider deletions at all in this paper (see [15, 16, 23] for challenges on performing deletions in random binary search trees).

References

1 Georgy Adelson-Velsky and Evgenii Landis. An algorithm for the organization of information. *Proceedings of the USSR Academy of Sciences (in Russian)*, 146:263–266, 1962.

2 Alfred V. Aho, John E. Hopcroft, and Jeffrey D. Ullman. *The Design and Analysis of Computer Algorithms.* Addison-Wesley, 1974.

3 Praveen Alapati, Swamy Saranam, and Madhu Mutyam. Concurrent treaps. In Shadi Ibrahim, Kim-Kwang Raymond Choo, Zheng Yan, and Witold Pedrycz, editors, *Algorithms and Architectures for Parallel Processing - 17th International Conference, ICA3PP 2017, Helsinki, Finland, August 21-23, 2017, Proceedings*, volume 10393 of *Lecture Notes in Computer Science*, pages 776–790. Springer, 2017. doi:10.1007/978-3-319-65482-9_63.

4 Susanne Albers and Marek Karpinski. Randomized splay trees: Theoretical and experimental results. *Inf. Process. Lett.*, 81(4):213–221, 2002. doi:10.1016/S0020-0190(01)00230-7.

5 Arne Andersson. Improving partial rebuilding by using simple balance criteria. In Frank K. H. A. Dehne, Jörg-Rüdiger Sack, and Nicola Santoro, editors, *Algorithms and Data Structures, Workshop WADS '89, Ottawa, Canada, August 17-19, 1989, Proceedings*, volume 382 of *Lecture Notes in Computer Science*, pages 393–402. Springer, 1989. doi:10.1007/3-540-51542-9_33.

6 Arne Andersson. General balanced trees. *J. Algorithms*, 30(1):1–18, 1999. doi:10.1006/JAGM.1998.0967.

7 Arne Andersson, Rolf Fagerberg, and Kim S. Larsen. Balanced binary search trees. In Dinesh P. Mehta and Sartaj Sahni, editors, *Handbook of Data Structures and Applications*, chapter 10. Chapman and Hall/CRC, 2004. doi:10.1201/9781420035179.CH10.

8 Cecilia R. Aragon and Raimund Seidel. Randomized search trees. In *30th Annual Symposium on Foundations of Computer Science, Research Triangle Park, North Carolina, USA, 30 October - 1 November 1989*, pages 540–545. IEEE Computer Society, 1989. doi:10.1109/SFCS.1989.63531.

9 Guy E. Blelloch and Margaret Reid-Miller. Fast set operations using treaps. In Gary L. Miller and Phillip B. Gibbons, editors, *Proceedings of the Tenth Annual ACM Symposium on Parallel Algorithms and Architectures, SPAA '98, Puerto Vallarta, Mexico, June 28 - July 2, 1998*, pages 16–26. ACM, 1998. doi:10.1145/277651.277660.

10 Andrew Donald Booth and Andrew J. T. Colin. On the efficiency of a new method of dictionary construction. *Information and Control*, 3(4):327–334, 1960. doi:10.1016/S0019-9958(60)90901-3.

11 Gunnar Brinkmann and Jan Degraer anda Karel De Loof. Rehabilitation of an unloved child: semi-splaying. *Software: Practice and Experience*, 39(1):33–45, 2009. doi:10.1002/SPE.886.

12 Gerth Stølting Brodal. Bottom-up rebalancing binary search trees by flipping a coin. *CoRR*, abs/2404.xxxxx, 2024. doi:10.48550/ARXIV.2404.xxxxxx.

13 Mark R. Brown. A storage scheme for height-balanced trees. *Information Processing Letters*, 7(5):231–232, 1978. doi:10.1016/0020-0190(78)90005-4.

14 Mark R. Brown. Addendum to "a storage scheme for height-balanced trees". *Information Processing Letters*, 8(3):154–156, 1979. doi:10.1016/0020-0190(79)90009-7.

15 Joseph C. Culberson and J. Ian Munro. Explaining the behaviour of binary search trees under prolonged updates: A model and simulations. *Comput. J.*, 32(1):68–75, 1989. doi:10.1093/COMJNL/32.1.68.

16 Joseph C. Culberson and J. Ian Munro. Analysis of the standard deletion algorithms in exact fit domain binary search trees. *Algorithmica*, 5(3):295–311, 1990. doi:10.1007/BF01840390.

17 Martin Fürer. Randomized splay trees. In Robert Endre Tarjan and Tandy J. Warnow, editors, *Proceedings of the Tenth Annual ACM-SIAM Symposium on Discrete Algorithms, 17-19 January 1999, Baltimore, Maryland, USA*, pages 903–904. ACM/SIAM, 1999. URL: https://dl.acm.org/doi/10.5555/314500.315079.

18 Igal Galperin and Ronald L. Rivest. Scapegoat trees. In Vijaya Ramachandran, editor, *Proceedings of the Fourth Annual ACM-SIAM Symposium on Discrete Algorithms, 25-27 January 1993, Austin, Texas, USA*, pages 165–174. ACM/SIAM, 1993. URL: https://dl.acm.org/doi/10.5555/313559.313676.

19 Leonidas J. Guibas and Robert Sedgewick. A dichromatic framework for balanced trees. In *19th Annual Symposium on Foundations of Computer Science, Ann Arbor, Michigan, USA, 16-18 October 1978*, pages 8–21. IEEE Computer Society, 1978. doi:10.1109/SFCS.1978.3.

20 Thomas N. Hibbard. Some combinatorial properties of certain trees with applications to searching and sorting. *Journal of the ACM*, 9(1):13–28, 1962. doi:10.1145/321105.321108.

21 Donald Ervin Knuth. *The art of computer programming, Volume III, 2nd Edition.* Addison-Wesley, 1998.

22 Conrado Martínez and Salvador Roura. Randomized binary search trees. *J. ACM*, 45(2):288–323, 1998. doi:10.1145/274787.274812.

23 Wolfgang Panny. Deletions in random binary search trees: A story of errors. *Journal of Statistical Planning and Inference*, 140(8):2335–2345, 2010. doi:10.1016/j.jspi.2010.01.028.

24 Raimund Seidel. Maintaining ideally distributed random search trees without extra space. In Susanne Albers, Helmut Alt, and Stefan Näher, editors, *Efficient Algorithms, Essays Dedicated to Kurt Mehlhorn on the Occasion of His 60th Birthday*, volume 5760 of *Lecture Notes in Computer Science*, pages 134–142. Springer, 2009. doi:10.1007/978-3-642-03456-5_9.

25 Raimund Seidel and Cecilia R. Aragon. Randomized search trees. *Algorithmica*, 16(4/5):464–497, 1996. doi:10.1007/BF01940876.

26 Daniel Dominic Sleator and Robert Endre Tarjan. Self-adjusting binary search trees. *Journal of the ACM*, 32(3):652–686, 1985. doi:10.1145/3828.3835.

27 J. W. J. Williams. Algorithm 232: Heapsort. *Communications of the ACM*, 7(6):347–348, 1964. doi:10.1145/512274.512284.

28 P. F. Windley. Trees, forests and rearranging. *The Computer Journal*, 3(2):84–88, 1960. doi:10.1093/COMJNL/3.2.84.

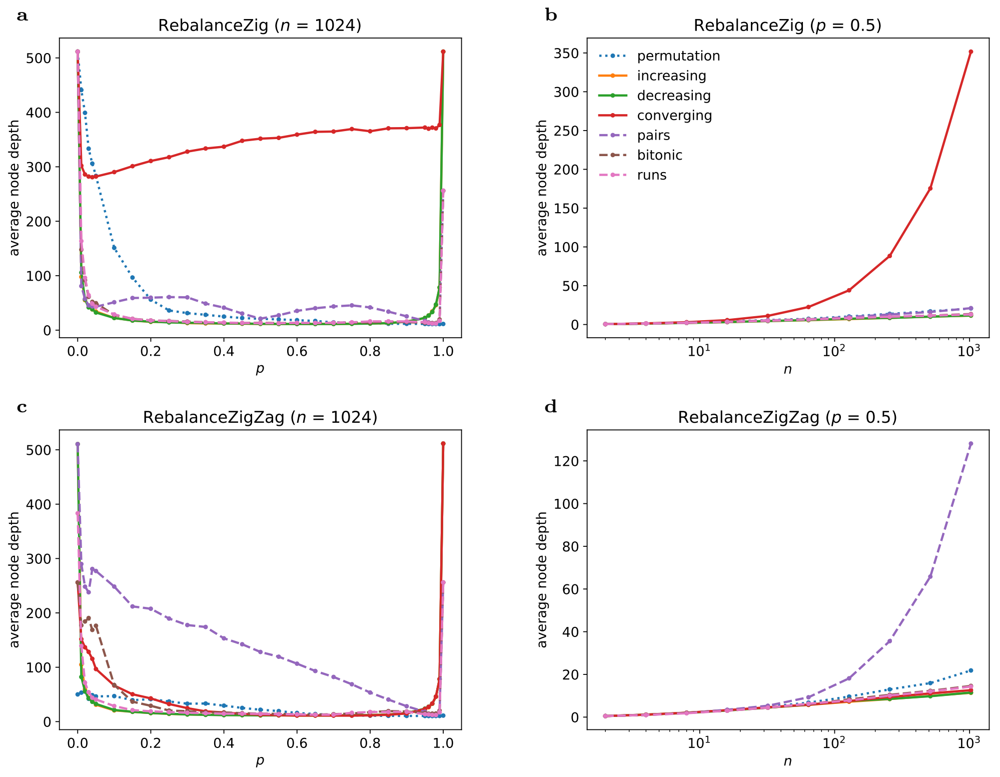

Figure 7 (left) The average node depths in binary search trees created by RebalanceZig and RebalanceZigZag, respectively, for various types of choices of p for insertion sequences of length 1024; (right) similar results but for fixed coin probability $p = \frac{1}{2}$, and sequence lengths in the range 1 to 1024. For $p = 0$ all rotations are performed at the inserted node (and RebalanceZig always creates a path), whereas no rotations are performed when $p = 1$, i.e., unbalanced binary search trees.