Zipping Segment Trees

Lukas Barth
Karlsruhe Institute of Technology, Germany
lukas.barth@kit.edu

Dorothea Wagner
Karlsruhe Institute of Technology, Germany
dorothea.wagner@kit.edu

Abstract

Stabbing queries in sets of intervals are usually answered using segment trees. A dynamic variant of segment trees has been presented by van Kreveld and Overmars [14], which uses red-black trees to do rebalancing operations. This paper presents *zipping segment trees* – dynamic segment trees based on zip trees, which were recently introduced by Tarjan et al. [13]. To facilitate zipping segment trees, we show how to uphold certain segment tree properties during the operations of a zip tree. We present an in-depth experimental evaluation and comparison of dynamic segment trees based on red-black trees, weight-balanced trees and several variants of the novel zipping segment trees. Our results indicate that zipping segment trees perform better than rotation-based alternatives.

2012 ACM Subject Classification Theory of computation → Sorting and searching; Information systems → Point lookups

Keywords and phrases Segment Trees, Dynamic Segment Trees, Zip Trees, Data Structures

Digital Object Identifier 10.4230/LIPIcs.SEA.2020.25

Related Version A full version of the paper is available at `https://arxiv.org/abs/2004.03206`.

Supplementary Material `https://github.com/tinloaf/ygg/releases/tag/version_sea2020`

Funding *Lukas Barth*: This work was partially funded by the Helmholtz Program "Enery System Design", Topic 2 "Digitalization and System Technology".

1 Introduction

A common task in computational geometry, but also many other fields of application, is the storage and efficient retrieval of segments (or more abstractly: intervals). The question of which data structure to use is usually guided by the nature of the retrieval operations, and whether the data structure must be dynamic, i.e., support updates. One very common retrieval operation is that of a *stabbing query*, which can be formulated as follows: Given a set of intervals on $\mathbb{R}$ and a query point $x \in \mathbb{R}$, report all intervals that contain x.

For the static case, a *segment tree* is the data structure of choice for this task. It supports stabbing queries in $\mathcal{O}(\log n)$ time (with n being the number of intervals). Segment trees were originally introduced by Bentley [3]. While the segment tree is a static data structure, i.e., is built once and would have to be rebuilt from scratch to change the contained intervals, van Kreveld and Overmars present a dynamic version [14], called *dynamic segment tree* (DST).

Dynamic segment trees are applied in many fields. Solving problems from computational geometry certainly is the most frequent application, for example for route planning based on geometry [7] or labeling rotating maps [8]. However, DSTs are also useful in other fields, for example internet routing [4] or scheduling algorithms [1].

In this paper, we present an adaption of dynamic segment trees, so-called *zipping segment trees*. Our main contribution is replacing the usual red-black-tree base of dynamic segment trees with zip trees, a novel form of balancing binary search trees introduced recently by

Tarjan et al. [13]. On a conceptual level, basing dynamic segment trees on zip trees yields an elegant and simple variant of dynamic segment trees. Only few additions to the zip tree's rebalancing methods are necessary. On a practical level, we can show that zipping segment trees outperform dynamic segment trees based on red-black trees in our experimental setting.

2 Preliminaries

A concept we need for zip trees are the two *spines* of a (sub-) tree. We also talk about the spines of a node, by which we mean the spines of the tree rooted in the respective node. The *left spine* of a subtree is the path from the tree's root to the previous (compared to the root, in tree order) node. Note that if the root (call it v) is not the overall smallest node, the left spine exits the root left, and then always follows the right child, i.e., it looks like $(v, L(v), R(L(v)), R(R(L(v))), \dots)$. Conversely, the *right spine* is the path from the root node to the next node compared to the root node. Note that this definition differs from the definition of a spine by Tarjan et al. [13].

2.1 Union-Copy Data Structure

Dynamic segment trees in general carry annotations of sets of intervals at their vertices or edges. These set annotations must be stored and updated somehow. To achieve the run times in [14], van Kreveld and Overmars introduce the *union-copy* data structure to manage such sets. Sketching this data structure would be out of scope for this paper. It is constructed by intricately nesting two different types of union-find data structures: a textbook union-find data structure using union-by-rank and path compression (see for example Seidel and Sharir [11]) and the $UF(i)$ data structure by La Poutré [9].

For this paper, we just assume this union-copy data structure to manage sets of items. It offers the following operations[1]:

createSet() Creates a new empty set in $\mathcal{O}(1)$.

deleteSet() Deletes a set in $\mathcal{O}(1 + k \cdot F_N(n))$, where k is the number of elements in the set, and $F(n)$ is the time the *find* operation takes in one of the chosen union-find structures.

copySet(A) Creates a new set that is a copy of A in $\mathcal{O}(1)$.

unionSets(A,B) Creates a new set that contains all items that are in A or B, in $\mathcal{O}(1)$.

createItem(X) Creates a new set containing only the (new) item X in $\mathcal{O}(1)$.

deleteItem(X) Deletes X from *all* sets in $\mathcal{O}(1 + k)$, where k is the number of sets X is in.

2.2 Dynamic Segment Trees

This section recapitulates the workings of dynamic segment trees as presented by van Kreveld and Overmars [14] and outlines some extensions. Before we describe the dynamic segment tree, we briefly describe a classic static segment tree and the *segment tree property*. For a more thorough description, see de Berg et al. [5, 10.2]. Segment trees store a set $\mathcal{I}$ of n intervals. Let $x_1, x_2, \dots x_{2n}$ be the ordered sequence of interval end points in $\mathcal{I}$. For the sake of clarity and ease of presentation, we assume that all interval borders are distinct, i.e., $x_i > x_{i+1}$. We also assume all intervals to be closed. Lifting these two restrictions is straightforward.

[1] The data structure presented by Kreveld and Overmars provides more operations, but the ones mentioned here are sufficient for this paper.

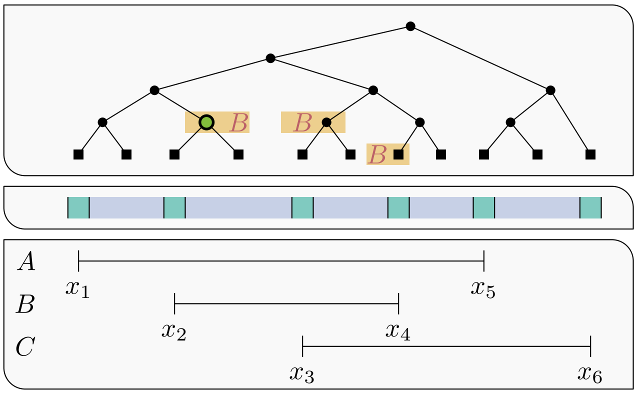


Figure 1 A segment tree (top) for three intervals (bottom). The middle shows the elementary intervals. Note that the green intervals do actually contain just one point and are only drawn fat so that they can be seen. The nodes marked with B are the nodes that carry the annotation for interval B.

In the first step, we forget whether an x_i is a start or an end of an interval. The intervals

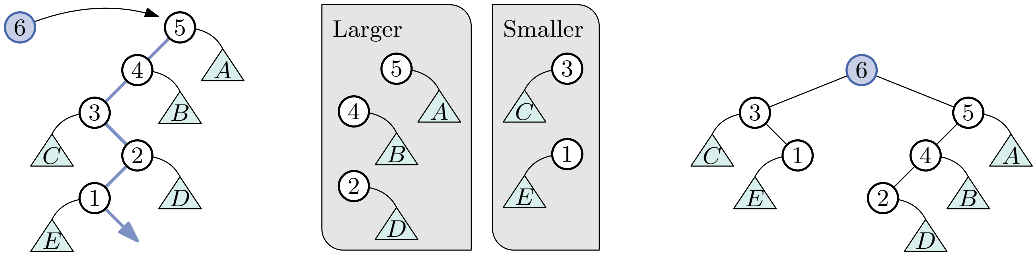
$$(-\infty, x_1), [x_1, x_1], (x_1, x_2), [x_2, x_2], \ldots (x_{2n-1}, x_{2n}), [x_{2n}, x_{2n}], (x_{2n}, \infty)$$

are called the *elementary intervals* of $\mathcal{I}$. To create a segment tree, we create a leaf node for every elementary interval. On top of these leaves, we create a binary tree. The exact method of creating the binary tree is not important, but it should adhere to some balancing guarantee to provide asymptotically logarithmic depths of all leaves.

Such a segment tree is outlined in Figure 1. The lower box indicates the three stored intervals and their end points $x_1, \ldots x_6$. The middle box contains a visualization of the elementary intervals, where the green intervals are the $[x_i, x_i]$ intervals (note that while of course they should have no area, we have drawn them "fat" to make them visible) while the blue intervals are the (x_i, x_{i+1}) intervals. The top box contains the resulting segment tree, with the square nodes being the leaf nodes corresponding to the elementary intervals, and the circular nodes being the inner nodes.

We associate each inner node v with the union of all the intervals corresponding to the leaves in the subtree below v. In Figure 1, that means that the larger green inner node is associated with the intervals $[x_2, x_3)$, i. e., the union of $[x_2, x_2]$ and (x_2, x_3), which are the two leaves beneath it. Recall that a segment tree should support fast stabbing queries, i.e., for any query point q, should report which intervals contain q. To this end, we annotate the nodes of the tree with sets of intervals. For any interval I, we annotate I at every node v such that the associated interval of v is completely contained in I, but the associated interval of v's parent is not. In Figure 1, the annotations for B are shown. For example, consider the larger green node. Again, its associated interval is $[x_2, x_3)$, which is completely contained in $B = [x_2, x_4]$. However, its parent is associated with $[x_1, x_3)$, which is not contained in B. Thus, the large green node is annotated with B.

A segment tree constructed in such a way is semi-dynamic. Segments cannot be removed, and new segments can be inserted only if their end points are already end points of intervals in I. To provide a fully dynamic data structure with the same properties, van Kreveld and Overmars present the dynamic segment tree [14]. It relaxes the property that intervals are always annotated on the topmost nodes the associated intervals of which are still completely contained in the respective interval. Instead, they propose the *weak segment tree property*:



(a) Before unzipping. **(b)** Unzipped parts. **(c)** After reassembly.

Figure 2 Illustration of the process of unzipping a path in zip trees. Nodes' names are simultaneously their ranks. Node keys are not shown.

For any point q and any interval I that contains q, the search path of q in the segment tree contains exactly one node that is annotated with I. For any q and any interval J that does not contain q, no node on the search path of q is annotated with J. Thus, collecting all annotations along the search path of q yields the desired result, all intervals that contain q. It is easy to see that this property is true for segment trees: For any interval I that contains q, some node on the search path for q must be the first node the associated interval of which does not fully contain I. This node contains an annotation for I.

Dynamic segment trees also remove the distinction between leaf nodes and inner nodes. In a dynamic segment tree, every node represents an interval border. To insert a new interval, we insert two nodes representing its borders into the tree, adding annotations as necessary. To delete an interval, we remove its associated nodes. If the dynamic segment tree is based on a classic red-black tree, both operations require rotations to rebalance. Performing such a rotation without adapting the annotations would cause the weak segment tree property to be violated. Also, the nodes removed when deleting an interval might have carried annotations, which also potentially violates the weak segment tree property.

We must thus fix the weak segment tree property during rotations. We must also make sure that any deleted node does not carry any annotations, and we must specify how we add annotations when inserting new intervals.

3 Zipping Segment Trees

In Section 2.2 we have described a variant of the dynamic segment trees introduced by van Kreveld and Overmars [14]. These are built on top of a balancing binary search tree, for which van Kreveld and Overmars suggested using red-black trees. The presented technique is able to uphold the weak segment tree property during the red-black tree's operations: node rotations, node swaps, leaf deletion and deletion of vertices of degree one. These are comparatively many operations that must be adapted to dynamic segment trees. Also, each each operation incurs a run time cost for the repairing of the weak segment tree property.

Thus it stands to reason to look at different underlying trees which either reduce the number of necessary balancing operations. One such data structure are *zip trees* introduced by Tarjan et al. [13]. Instead of inserting nodes at the bottom of the tree and then rotating the tree as necessary to achieve balance, these trees determine the height of the node to be inserted before inserting it in a randomized fashion by drawing a *rank*. The zip tree then forms a *treap*, a combination of a search tree and a heap: While the key of $L(v)$ (resp. $R(v)$) must always be smaller or equal (resp. larger) to the key of v, the ranks of both $L(v)$ and $R(v)$ must also be smaller or equal to the rank of v. Thus, any search path always sees nodes' ranks in a monotonically decreasing sequence. The ranks are chosen randomly in such a way that we expect the result to be a balanced tree. In a balanced binary tree, half of the nodes will be leaves. Thus, we assign rank 0 with probability 1/2. A fourth of the nodes in a balanced binary tree are in the second-to-bottom layer, thus we assign rank 1 with probability 1/4. In general, we assign rank k with probability $(1/2)^{k+1}$, i.e., the ranks follow a geometric distribution with mean 1. With this, Tarjan et al. show that the expected length of search paths is in $\mathcal{O}(\log n)$, thus the tree is expected to be balanced.

Zip trees do not insert nodes at the bottom or swap nodes down into a leaf before deletion. If nodes are to be inserted into or removed from the middle of a tree, other operations than rotations are necessary. For zip trees, these operations are *zipping* and *unzipping*. In the remainder of this section, we examine these two operations of zip trees separately and explain how to adapt them to preserve the weak segment tree property. For a more thorough description of the zip tree procedures, we refer the reader to [13].

3.1 Insertion and Unzipping

Figure 2 illustrates the unzipping operation that is used when inserting a node. Note that we use the numbers 1 through 6 as nodes' names as well as their ranks in this example. The triangles labeled A through E represent further subtrees. The node to be inserted is ⑥, the fat blue path is its search path (i.e., its key is smaller than the keys of ⑤, ④ and ②, but larger than the keys of ③ and ①). Since ⑥ has the largest rank in this example, the new node needs to become the new root. To this end, we unzip the search path, splitting it into the parts that are – in terms of nodes' keys – larger than ⑥ and parts that are smaller than ⑥. In other words: We group the nodes on the search path by whether we exited them to the left (a larger node) or to the right (a smaller node). Algorithm 1, when ignoring the highlighted parts, provides pseudocode for the unzipping operation.

We remove all edges on the search path (Step 1 in Algorithm 1). The result is depicted in the two gray boxes in Figure 2b: several disconnected parts that are either larger or smaller than the newly inserted node. Taking the new node ⑥ as the new root, we now reassemble these parts below it. The smaller parts go into the left subtree of ⑥, stringed together as each others' right children (Step 3 in Algorithm 1). Note that all nodes in the "smaller" set must have an empty right subtree, because that is where the original search path exited them – just as nodes in the "larger" set have empty left subtrees. The larger parts go into the right subtree of ⑥, stringed together as each others' left children. This concludes the unzipping operation, yielding the result shown in Figure 2c. With careful implementation, the whole operation can be performed during a single traversal of the search path.

To insert a segment into a dynamic segment tree, we need to do two things: First, we must correctly update annotations whenever a segment is inserted. Second, we must ensure that the tree's unzipping operation preserves the weak segment tree property.

We will not go into detail on how to achieve step one. In fact, we add new segments in basically the same fashion as red-black-tree based DSTs do. We first insert the two nodes representing the segment's start and end. Take the path between the two new nodes. The nodes on this path are the nodes at which a static segment tree would carry the annotation of the new segment. Thus, annotating these nodes (resp. the appropriate edges) repairs the weak segment tree property for the new segment.

In the remainder of this section, we explain how to adapt the unzipping operations of zip trees to repair the weak segment property. Let the annotation of an edge e before unzipping be $S(e)$, and let the annotation after unzipping be $S'(e)$. As an example how to fix the annotations after unzipping, consider in Figure 2 a search path that descends into subtree D before unzipping. It picks up the annotations on the unzipped path from ⑤ up to ②, i.e., $S\big(\vec{L}(5)\big)$, $S\big(\vec{L}(4)\big)$, $S\big(\vec{R}(3)\big)$, and on the edge going into D, i.e., $S\big(\vec{R}(2)\big)$. After unzipping, it

Algorithm 1 Unzipping routine. This inserts *new* into the tree at the position currently occupied by v by first disassembling the search path below v, and then reassembling the different parts as left and right spines below *new*. The highlighted parts are used to repair the dynamic segment tree's annotations. Note that in an efficient implementation, one would interleave all four steps.

```
Input : new: Node to be inserted
Input : v: Node to be replaced by new
cur ← v;
oldParent ← P(v);
smaller ← newList();
larger ← newList();
collected ← createSet();
/* Step 1: Remove edges along search path.                          */
while cur ≠ ⊥ do
    if new < cur then
        larger.append(cur);
        next ← L(cur);
        S(R⃗(cur)) ← unionSets(S(R⃗(cur)), collected);
        collected ← unionSets(collected, S(L⃗(cur)));
        next ← L(cur);
        L(cur) ← ⊥;
        cur ← next;
    else
        /* Omitted, symmetric to the case new < cur. Collect parts in
            smaller.                                                */
/* Step 2: Insert new.                                              */
if L(oldParent) = v then
    L(oldParent) ← new;
else
    R(oldParent) ← new;
/* Step 3: Reassemble left spine from parts smaller than new        */
parent ← new;
for n ∈ smaller do
    if parent = new then
        L(parent) ← n;
        deleteSet(S(L⃗(parent))); S(L⃗(parent)) ← createSet();
    else
        R(parent) ← n;
        deleteSet(S(R⃗(parent))); S(R⃗(parent)) ← createSet();
/* Step 4: Reassemble right spine. This is symmetric to the left
    spine and thus omitted.                                         */
```

picks up the annotations on all the new edges on the path from ⑥ to ② plus $S'\big(\vec{R}(2)\big)$. We set the annotations on all newly inserted edges to $\emptyset$ after unzipping. Thus, we need to add the annotations before unzipping, i.e., $S\big(\vec{L}(5)\big) \cup S\big(\vec{L}(4)\big) \cup S\big(\vec{R}(3)\big)$, to the edge going into D. We therefore set $S'\big(\vec{R}(2)\big) = S\big(\vec{R}(2)\big) \cup S\big(\vec{L}(5)\big) \cup S\big(\vec{L}(4)\big) \cup S\big(\vec{R}(3)\big)$ after unzipping.

In Algorithm 1, the blue highlighted parts are responsible for repairing the annotations. While descending the search path to be unzipped, we incrementally collect all annotations we see on this search path (line 11), and at every visited node add the previously collected annotations to the other edge (line 10), i. e., the edge that is not on the search path. By setting the annotations of all newly created edges to the empty set (lines 24 and 27), we make sure that after reassembly, every search path descending into one of the subtrees attached to the reassembled parts picks up the same annotations on the edge into that subtree as it would have picked up on the path before disassembly.

3.2 Deletion and Zipping

Deleting segments again is a two-staged challenge: We need to remove the deleted segment from all annotations, and must make sure that the *zipping* operation employed for node deletion in zip trees upholds the weak segment tree property. Removing a segment from all annotations is trivial when using the union-copy data structure outlined in Section 2.1: The *deleteItem()* method does exactly this.

We now outline the *zipping* procedure and how it can be amended to repair the weak segment tree property. Zipping two paths in the tree works in reverse to unzipping. Pseudocode is given in Algorithm 2. Again, the pseudocode without the highlighted parts is the pseudocode for plain zipping, independent of any dynamic segment tree. Assume that in the situation Figure 2c, we want to remove ⑥, thus we want to arrive at the situation in Figure 2a. The zipping operation consist of walking down the left spine (consisting of ③ and ① in the example) and the right spine (consisting of ⑤, ④ and ② in the example) simultaneously and zipping both into a single path. This is done by the loop in line 7. At every point during the walk, we have a current node on both spines, call it the *current left* node l and the *current right* node r. Also, there is a *current parent* p, which is the bottom of the new zipped path being built. In the beginning, the current parent is the parent of the node being removed.[2] In each step, we select the current node with the smaller rank, breaking ties arbitrarily (line 8). Without loss of generality, assume the current right node is chosen (the branch starting in line 20). We attach the chosen node to the bottom of the zipped path (p), and then r itself becomes p. Also, we walk further down on the right spine.

Note that the choice whether to attach left or right to the bottom of the zipped path (made via *attachRight* in Algorithm 2) is made in such a way that the position in which we attach previously was part of one of the two spines being zipped. For example, if p came from the right spine, we attach left to it. However, $\vec{L}(p)$ comes from the right spine. This method of attaching nodes always upholds the search tree property: When we make a node from the right spine the new parent (line 29), we know that the new p is currently the largest remaining nodes on the spines. We always attach left to this p (line 31). Since all other nodes on the spine are smaller than p, this is valid. The same argument holds for the left spine.

We now explain how the edge annotations can be repaired so that the weak segment tree property is upheld. Assume that for an edge e, $S(e)$ is the annotation of e before zipping, and $S'(e)$ is the annotation of e after zipping. Again, we argue via the subtrees that search paths can descend into. A search path descending into a subtree on the right of a node on the right spine, e.g., subtree B attached to ④ in Figure 2c, will before zipping pick up the annotation on the right edge of the node being removed plus all annotations on the spine up to the respective node, e.g., $S\big(\vec{R}(6)\big) \cup S\big(\vec{L}(5)\big)$, before descending into the respective subtree (B in the example). To preserve these picked up annotations, we again push them down onto the edge that actually leads away from the spine into the respective subtree.

[2] If the root is being removed, pretend there is a pseudonode above the root.

Algorithm 2 Zipping routine. This removes v from the tree, zipping the left and right spines of v. The highlighted parts are used to repair the dynamic segment tree's annotations.

```
Input : n: Node to be removed
l ← L(n);                          // Current node descending v's left spine
r ← R(n);                          // Current node descending v's right spine
p ← P(n);                          // Bottom of the partially zipped path
attachRight ← R(P(n)) = v;
collected_l = copySet(S(L⃗(n)));
collected_r = copySet(S(R⃗(n)));
while l ≠ ⊥ ∨ r ≠ ⊥ do
    if (l ≠ ⊥) ∧ ((r = ⊥) ∨ (rank(l) > rank(r))) then
        if attachRight then
            R(p) ← l;
            deleteSet(S(R⃗(p))); S(R⃗(p)) ← createSet();
        else
            L(p) ← l;
            deleteSet(S(L⃗(p))); S(L⃗(p)) ← createSet();
        S(L⃗(l)) ← unionSets(S(L⃗(l)), collected_l);
        collected_l ← unionSets(collected_l, S(R⃗(l)));
        p ← l;
        l ← R(l);
        attachRight ← true;
    else
        if attachRight then
            R(p) ← r;
            deleteSet(S(R⃗(p))); S(R⃗(p)) ← createSet();
        else
            L(p) ← r;
            deleteSet(S(L⃗(p))); S(L⃗(p)) ← createSet();
        S(R⃗(r)) ← unionSets(S(R⃗(r)), collected_r);
        collected_r ← unionSets(collected_r, S(L⃗(r)));
        p ← r;
        r ← L(r);
        attachRight ← false;
```

Formally, during zipping, we keep two sets of annotations, one per spine. In Algorithm 2, these are $collected_l$ and $collected_r$, respectively. Let n be the node to be removed. Initially, we set $collected_l = S\big(\vec{L}(n)\big)$ and $collected_r = S\big(\vec{R}(n)\big)$. Then, whenever we pick a node c from the left (resp. right) spine as new parent, we set $S'\big(\vec{L}(c)\big) = S\big(\vec{L}(c)\big) \cup collected_l$ (resp. $S'\big(\vec{R}(c)\big) = S\big(\vec{R}(c)\big) \cup collected_r$). This pushes down everything we have collected to the edge leading away from the spine at c. Then, we set $collected_l = collected_l \cup S\big(\vec{R}(c)\big)$ and $S'\big(\vec{R}(c)\big) = \emptyset$ (resp. $collected_r = collected_r \cup S\big(\vec{L}(c)\big)$ and $S'\big(\vec{L}(c)\big) = \emptyset$). This concludes the techniques necessary to use zip trees as a basis for dynamic segment trees, yielding *zipping segment trees*.

3.3 Complexity

Zip trees are randomized data structures, therefore all bounds on run times are expected bounds. In [13, Theorem 4], Tarjan et al. claim that the expected number of pointers changed during a zip or unzip is in $\mathcal{O}(1)$. However, they actually even show the stronger claim that the number of nodes on the zipped (or unzipped) paths is in $\mathcal{O}(1)$. Observe that the loops in lines 6 and 21 of Algorithm 1 as well as line 7 of Algorithm 2 are executed at most once per node on the unzipping (resp. zipping) path. Inside each of the loops, a constant number of calls are made to each of the *copySet*, *createSet*, *deleteSet* and *unionSets* operations. Thus, the rebalancing operations incur expected constant effort plus a constant number of calls to the union-copy data structure.

When inserting a new segment, we add it to the sets annotated at every vertex along the path between the two nodes representing the segment borders. Since the depth of every node is expected logarithmic in n, this incurs expected $\ln(n)$ calls to *unionSets*. The deletion of a segment from all annotations costs exactly one call to *deleteItem*.

All operations but *deleteSet* and *deleteItem* are in $\mathcal{O}(1)$ if the union-copy data structure is appropriately built. The analysis for the two deletion functions is more complicated and involves amortization. The rough idea is that every non-deletion operation can increase the size of the union-copy's representation only by a limited amount. On the other hand, the two deletion operations each decrease the representation size proportionally to their run time.

The red-black-tree-based DSTs by van Kreveld and Overmars [14] also need $\Omega(\ln n)$ calls to *copySet* during the insertion operation, and at least a constant number of calls during tree rebalancing and deletion. Therefore, for every operation on zipping segment trees, the (expected) number of calls to the union-copy data structure's functions is no larger than the number of calls in the red-black-tree-based implementation and we achieve the same (but only expected) run time guarantees, which are $\mathcal{O}(\log n)$ for insertion, $\mathcal{O}(\log n \cdot a(i, n))$ for deletion (with $a(i, n)$ being the row-inverse of the Ackermann function, for some constant i) and $\mathcal{O}(\log n + k)$ for stabbing queries, where k is the number of reported segments.

3.4 Generating Ranks

Nodes' ranks play a major role in the rebalancing operations of zip trees. In Section 3, we already motivated why nodes' ranks should follow a geometric distribution with mean 1; it is the distribution of the node depths in a perfectly balanced tree.

A practical implementation needs to somehow generate these values. The obvious implementation would be to somehow generate a (pseudo-) random number and determine the position of the first 1 in its binary representation. The rank generated in this way is then stored at the respective node.

Storing the rank at the node can be avoided if the rank is generated in a reproducible fashion. Tarjan et al. [12] already point out that one can "compute it as a pseudo-random function of the node (or of its key) each time it is needed." In fact, the idea already appeared earlier in the work by Seidel and Aragon [10] on treaps. They suggest evaluating a degree 7 polynomial with randomly chosen coefficients at the (numerical representation of) the node's key. However, the 8-wise independence of the random variables generated by this technique is not sufficient to uphold the theoretical guarantees given by Tarjan et al. [12].

However, without any theoretical guarantees, a simpler method for reproducible ranks can be achieved by employing simple hashing algorithms. Note that even if applying universal hashing, we do not get a guarantee regarding the probability distribution for the values of individual bits of the hash values. However, in practice, we expect it to yield results similar to true randomness. As a fast hashing method, we suggest using the $2/m$-almost-universal multiply-shift method from Dietzfelbinger et al. [6]. Since we are interested in generating an entire machine word in which we then search for the first bit set to 1, we can skip the “shift” part, and the whole process collapses into a simple multiplication.

4 Experimental Evaluation of Dynamic Segment Trees Bases

In this section, we experimentally evaluate zipping segment trees as well as dynamic segment trees based on two of the most prominent rotation-based balanced binary search trees: red-black trees and weight-balanced trees. Weight-balanced trees require a parametrization of their rebalancing operation. In [2], we perform an in-depth engineering of weight-balanced trees. For this analysis of dynamic segment trees, we pick only the two most promising variants of weight-balanced trees: top-down weight-balanced trees with $\langle \Delta, \Gamma \rangle = \langle 3, 2 \rangle$ and top-down weight-balanced trees with $\langle \Delta, \Gamma \rangle = \langle 2, 3/2 \rangle$.

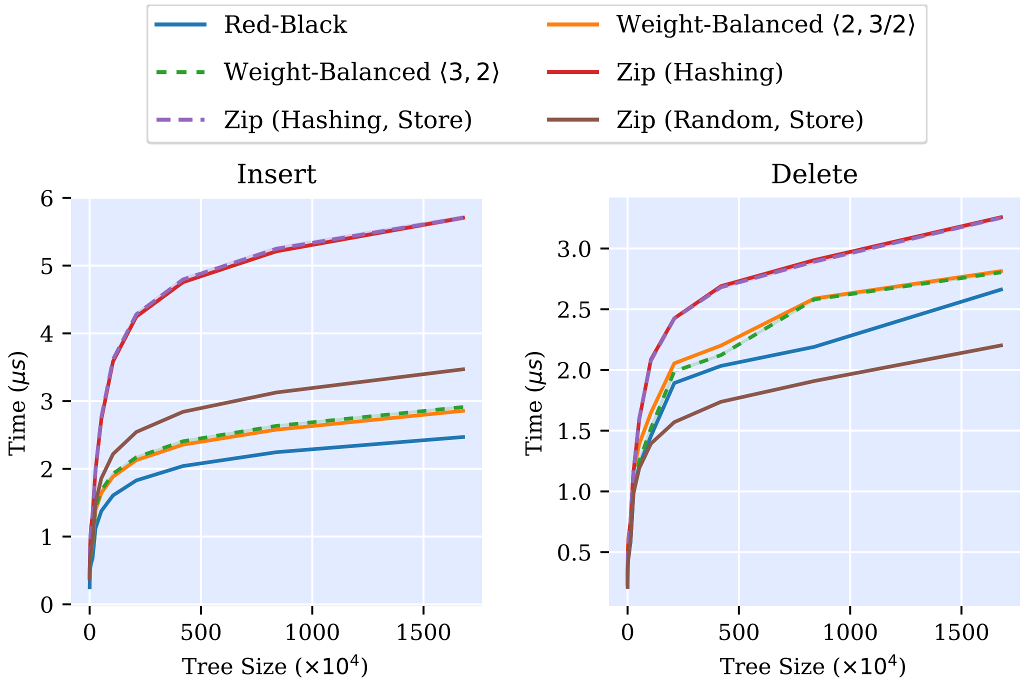
Note that since we are only interested in the performance effects of the trees underlying the DST, and not in the performance of an implementation of the complex union-copy data structure, we have implemented a simplified variant of DSTs which only reports the aggregate value of weighted segments at a stabbing query, instead of a list of the respective segments. See the full version of this paper for details. Evaluating the performance of the union-copy data structure is out of scope of this work.

For the zip trees, we choose a total of three variants, based on the choices explained in Section 3.4: The first variant, denoted *Hashing*, generates nodes' ranks by applying the fast hashing scheme by Dietzfelbinger et al. [6] to the nodes' memory addresses. In this variant, node ranks are not stored at the nodes but re-computed on the fly when they are needed. The second variant, denoted *Hashing, Store* also generates nodes' ranks from the same hashing scheme, but stores ranks at the nodes. The last variant, denoted *Random, Store* generates nodes' ranks independent of the nodes and stores the ranks at the nodes.

We first individually benchmark the two operations of inserting (resp. removing) a segment to (resp. from) the dynamic segment tree. Our benchmark works by first creating a base dynamic segment tree of a certain size, then inserting new segments (resp. removing segments) into that tree. The number of new (resp. removed) segments is chosen to be the minimum of 10^5 and 5% of the base tree size. Segment borders are chosen by drawing twice from a uniform distribution. All segments are associated with a real-valued value. We conduct our experiments on a machine equipped with 128 GB of RAM and an Intel® Xeon® E5-1630 CPU, which has 10 MB of level 3 cache. We compile using GCC 8.1, at optimization level “-O3 -ffast-math”. We do not run experiments concurrently. Each experiment is repeated for ten different seed values, and repeated five times for each seed value to account for measurement noise. All our code is published, see the link at the beginning of this paper.

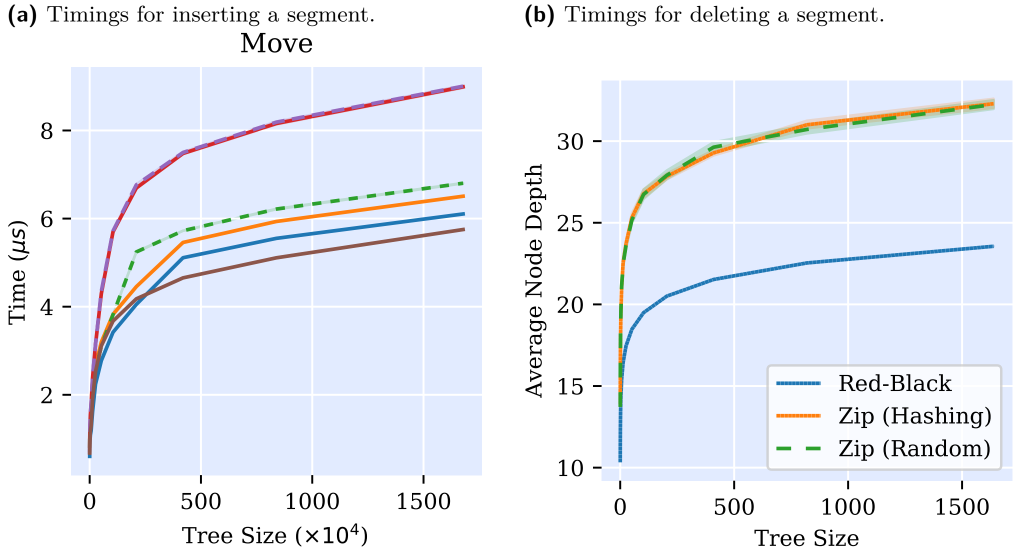
Figure 3a displays the results for the insert operation. We see that the red-black tree performs best for this operation, about a 30% faster ($\approx 2.5\mu s$ per operation at $1.5 \cdot 10^7$ nodes) than the fastest zip tree variant, which is the variant using random rank selection ($\approx 3.5\mu s$ per operation). The two weight-balanced trees lie between the red-black tree and the randomness-based zip tree. Both hashing-based zip trees are considerably slower.

For the deletion operation, shown in Figure 3b, the randomness-based zip tree is significantly faster than the best competitor, the red-black tree. Again, the weight-balanced trees are slightly slower than the red-black tree, and the hashing-based zip trees fare the worst.



(a) Timings for inserting a segment.

(b) Timings for deleting a segment.



(c) Timings for moving a segment.

(d) Average node depths.

Figure 3 (a) – (c): Benchmark times for dynamic segment trees based on different balancing binary search trees. The y axis indicates the measured time per operation, while the x axis indicates the size of the tree that the operation is performed on. The lines indicate mean values. The standard deviation is all cases too small to be visible.
(d): Average depths of the nodes in DSTs based on red-black trees and zip trees. The x axis specifies the number of inserted segments. Shaded areas indicate the standard deviation.

Since zip trees are the fastest choice for deletion and red-black trees are the fastest for insertion, benchmarking the combination of both is obvious. Also, using an *dynamic* segment tree makes no sense in the absence of deletion. Thus, we next benchmark a *move* operation, which consists of first removing a segment from the tree, changing its borders, and re-inserting it. The results are shown in Figure 3c. We see that the randomness-based zipping segment tree is the best-performing dynamic segment tree for trees with at least $2.5 \cdot 10^6$ segments.

The obvious measurement to explore why different trees perform differently is the trees' balance, i.e., the average depth of a node in the respective trees. We conduct this experiment as follows: For each of the trees under study, we create trees of various sizes with randomly generated segments. In a tree generated in this way, we only see the effects of the *insert* operation, and not the *delete* operation. Thus, we continue by moving each segment once by removing it, changing its interval borders and re-inserting it. This way, the effect of the *delete* operation on the tree balance is also accounted for. Since the weight-balanced trees were not competitive previously, we perform this experiment only for the red-black and zip trees. We repeat the experiment with 30 different seeds to account for randomness. The results can be found in Figure 3d. We can see that zipping segment trees, whether based on randomness or hashing, are surprisingly considerably less balanced than red-black-based DSTs. Also, whether ranks are generated from hashing or randomness does not impact balance.

Concluding the evaluation, we gain several insights. First, deletions in zipping segment trees are so much faster than for red-black-based DSTs that they more than make up for the slower insertion, and the fastest choice for moving segments are zipping segment trees with ranks generated randomly. Second, we see that this speed does not come from a better balance, but in spite of a worse balance. The speedup must therefore come from more efficient rebalancing operations. Third, and most surprising, the question of how ranks are generated does not influence tree balance, but has a significant impact on the performance of deletion and insertion. However, the hash function we have chosen is very fast. Also, during deletion, no ranks should be (re-) generated for the variant that stores the ranks at the nodes. Thus, the performance difference can not be explained by the slowness of the hash function. Generating ranks with our chosen hash function must therefore introduce some disadvantageous structure into the tree that does not impact the average node depth.

5 Conclusion

We have presented zipping segment trees – a variation of dynamic segment trees, based on zip trees. The technique to maintain the necessary annotations for dynamic segment trees is comparatively simple, requiring only very little adaption of zip trees' routines. In our evaluation, we were able to show that zipping segment trees perform well in practice, and outperform red-black-tree or weight-balanced-tree based DSTs with regards to modifications.

However, we were not yet able to discover exactly why generating ranks from a (very simple) hash function does negatively impact performance. Exploring the adverse effects of this hash function and possibly finding a different hash function that avoids these effects remains future work. Another compelling future experiment would be to evaluate the performance when combined with the actual data structure by van Kreveld and Overmars.

All things considered, their relatively simple implementation and the superior performance when modifying segments makes zipping segment trees a good alternative to classical dynamic segment trees built upon rotation-based balancing binary trees.

References

1 Lukas Barth and Dorothea Wagner. Shaving peaks by augmenting the dependency graph. In *Proceedings of the Tenth ACM International Conference on Future Energy Systems*, pages 181–191. ACM, 2019. doi:10.1145/3307772.3328298.

2 Lukas Barth and Dorothea Wagner. Engineering top-down weight-balanced trees. In *2020 Proceedings of the Twenty-Second Workshop on Algorithm Engineering and Experiments (ALENEX)*, pages 161–174. SIAM, 2020. doi:10.1137/1.9781611976007.13.

3 Jon Louis Bentley. Algorithms for Klee's rectangle problems. Technical report, Department of Computer Science, Carnegie-Mellon University, 1977. Unpublished notes.

4 Yeim-Kuan Chang and Yung-Chieh Lin. Dynamic segment trees for ranges and prefixes. *IEEE Transactions on Computers*, 56(6):769–784, 2007. doi:10.1109/TC.2007.1037.

5 Mark de Berg, Otfried Cheong, Marc van Kreveld, and Mark Overmars. *Computational Geometry*. Springer, 2008. doi:10.1007/978-3-540-77974-2.

6 Martin Dietzfelbinger, Torben Hagerup, Jyrki Katajainen, and Martti Penttonen. A reliable randomized algorithm for the closest-pair problem. *Journal of Algorithms*, 25(1):19–51, 1997. doi:10.1006/jagm.1997.0873.

7 Stefan Edelkamp, Shahid Jabbar, and Thomas Willhalm. Geometric travel planning. *IEEE Transactions on Intelligent Transportation Systems*, 6(1):5–16, 2005. doi:10.1109/TITS.2004.838182.

8 Andreas Gemsa, Martin Nöllenburg, and Ignaz Rutter. Evaluation of labeling strategies for rotating maps. *Journal of Experimental Algorithmics (JEA)*, 21:1–21, 2016. doi:10.1145/2851493.

9 Johannes Antonius La Poutré. New techniques for the union-find problem. In *Proceedings of the first annual ACM-SIAM symposium on Discrete algorithms (SODA)*, pages 54–63, 1990.

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

11 Raimund Seidel and Micha Sharir. Top-down analysis of path compression. *SIAM Journal on Computing*, 34(3):515–525, 2005. doi:10.1137/S0097539703439088.

12 Robert E Tarjan, Caleb C Levy, and Stephen Timmel. Zip trees. *CoRR*, abs/1806.06726, 2018. arXiv:1806.06726.

13 Robert E Tarjan, Caleb C Levy, and Stephen Timmel. Zip trees. In *Workshop on Algorithms and Data Structures (WADS)*, pages 566–577. Springer, 2019. doi:10.1007/978-3-030-24766-9_41.

14 Marc J van Kreveld and Mark H Overmars. Union-copy structures and dynamic segment trees. *Journal of the ACM (JACM)*, 40(3):635–652, 1993. doi:10.1145/174130.174140.