

The Random Access Zipper: Simple, Persistent Sequences

Kyle Headley and Matthew A. Hammer

University of Colorado Boulder
kyle.headley@colorado.edu, matthew.hammer@colorado.edu

Abstract. We introduce the Random Access Zipper (RAZ), a simple, persistent data structure for editable sequences. The RAZ combines the structure of a zipper with that of a tree: like a zipper, edits at the cursor require constant time; by leveraging tree structure, relocating the edit cursor in the sequence requires log time. While existing data structures provide these time bounds, none do so with the same simplicity and brevity of code as the RAZ. The simplicity of the RAZ provides the opportunity for more programmers to extend the structure to their own needs, and we provide some suggestions for how to do so.

1 Introduction

The singly-linked list is the most common representation of sequences for functional programmers. This structure is considered a core primitive in every functional language, and moreover, the principles of its simple design recur throughout user-defined structures that are “sequence-like”. Though simple and ubiquitous, the functional list has a serious shortcoming: users may only efficiently access and edit the *head* of the list. In particular, random accesses (or edits) generally require linear time.

To overcome this problem, researchers have developed other data structures representing (functional) sequences, most notably, *finger trees* [8]. These structures perform well, allowing edits in (amortized) constant time and moving the edit location in logarithmic time. More recently, researchers have proposed the *RRB-Vector* [14], offering a balanced tree representation for immutable vectors. Unfortunately, these alternatives lack the simplicity and extensibility of the singly-linked list.

In this paper, we introduce the *random access zipper*, or RAZ for short. Like the common linked list, the RAZ is a general-purpose data structure for persistent sequences. The RAZ overcomes the performance shortcomings of linked lists by using probabilistically-balanced trees to make random access efficient (expected or amortized logarithmic time). The key insight for balancing these trees comes from [12], which introduces the notion of probabilistically-chosen *levels*.¹ To edit sequences in a persistent setting, the RAZ also incorporates the

¹ In short, these levels represent the heights of uniformly randomly chosen nodes in a full, balanced binary tree. See Section 3.

design of a zipper [9], which provides the notion of a *cursor* in the sequence. A cursor focuses edits on (or near) a distinguished element. The user may move the cursor locally (i.e., forward and backward, one element at a time), or globally (i.e., based on an index into the sequence), which provides random access to sequence elements.

The RAZ exposes two types to the user, `'a tree` and `'a zip`, which respectively represent an unfocused and focused sequence of elements (of type `'a`). The RAZ exposes the following interface to the user based on these types:

Function : Type	Time Complexity
<code>focus : 'a tree -> int -> 'a zip</code>	$O(\log n)$ expected
<code>unfocus : 'a zip -> 'a tree</code>	$O(\log n + m \cdot \log^2 m)$ expected
<code>insert : dir -> 'a -> 'a zedit</code>	$O(1)$ worst-case
<code>remove : dir -> 'a zedit</code>	$O(1)$ amortized
<code>replace : 'a -> 'a zedit</code>	$O(1)$ worst-case
<code>move : dir -> 'a zedit</code>	$O(1)$ amortized
<code>view : 'a zip -> 'a</code>	$O(1)$ worst-case

The second and third columns of the table respectively report the type (in OCaml) and time complexity for each operation; we explain each in turn.

Function `focus` transforms a tree into a zipper, given a position in the sequence on which to place the zipper's cursor. It runs in expected logarithmic time, where n is the number of elements in the sequence. We use expected analysis for this function (and the next) since the tree is balanced probabilistically.

Function `unfocus` transforms a (focused) zipper back to an (unfocused) tree; its time complexity $O(\log n + m \cdot \log^2 m)$ depends on the length of the sequence n , as well as m , the number of zipper-based edits since the last refocusing. We summarize those possible edits below. Assuming that the number m is a small constant, the complexity of `unfocus` is merely $O(\log n)$; when m grows to become significant, however, the current design of the RAZ performs more poorly, requiring an additional expected $O(\log^2 m)$ time to process each edit in building the balanced tree. An algorithm linear in m does exist for this purpose, but a more straightforward one was a better match for the simplicity of this codebase.

Functions `insert`, `replace`, `remove`, `move` each transform the zipper structure of the RAZ. We abbreviate their types using type `'a zedit`, which we define as the function type `'a zip \rightarrow 'a zip`. Function `insert` inserts a given element to the left or right of the cursor, specified by a direction of type `dir`. Function `remove` removes the element in the given direction. Its time complexity is amortized, since removal may involve decomposing subtrees of logarithmic depth; overall, these costs are amortized across edits that require them. Function `replace` replaces the element at the cursor with the given element. Function `move` moves the cursor one unit in the specified direction; just as with `remove`, this operation uses amortized analysis. Finally, function `view` retrieves the element currently focused at the cursor (in constant time).

In the next section, we give an in-depth example of using the operations described above. In particular, we illustrate how the RAZ represents the sequence

as a tree and as a zipper, showing how the operations construct and transform these two structures.

In Section 3, we present our implementation of the RAZ. It requires well under 200 lines of OCaml, which is publicly available:

<https://github.com/cuplv/raz.ocaml>

The code described in this paper comes from the `raz_simp.ml` file. This code includes ten main functions that work over a simple set of datatypes for trees, lists of trees, and zippers, as defined in Section 3. In contrast, the finger tree [8] requires approximately 800 lines in the OCaml “Batteries Included” repo [10] to provide similar functionality.

We evaluate the RAZ empirically in Section 4. In particular, we report the time required to build large sequences by inserting random elements into the sequence at random positions. Our evaluation demonstrates that the RAZ is very competitive with the finger tree implementation mentioned above, despite the simplicity of the RAZ compared with that of the finger tree.

In Section 5, we discuss the design decisions we considered for this implementation and exposition of the RAZ. Section 6 discusses related work, and in particular, alternative structures for persistent sequences. We give a deeper comparison to finger trees, and explain why other balanced trees designed for sets (e.g., red-black trees, or splay trees, etc.) are inappropriate for representing sequences.

2 Example

In this section, we give a detailed example of using the RAZ interface introduced above, and illustrate the internal tree and list structures of the RAZ informally, using pictures. In the next section, we make the code for these functions precise.

Consider the sequence of seven elements $\langle z, a, b, c, y, d, e \rangle$. In the example that follows, the programmer uses the RAZ to perform four operations over this sequence, editing it (functionally) in the process:

- She uses `focus` to place the cursor at offset 4 (element y),
- she uses `remove` to remove the element to the left of the cursor (element c),
- she uses `unfocus` in anticipation of refocusing, and
- she uses `focus` to place the cursor at the sequence’s start (element z).

Figure 1 (first image, top) shows the sequence of elements represented by a RAZ. As explained further in Section 3, the RAZ interposes randomly-chosen *levels* as meta-data in the sequence; we show these levels, $\langle 4, 6, 1, 2, 5, 3 \rangle$, interposed below the sequence elements. The second image shows the tree form of the RAZ, whose structure is determined by the sequence elements and levels above. This tree represents an unfocused RAZ.

Next, the programmer uses `focus` to place the cursor into the tree, in order to edit its elements. The third image in Figure 1 shows the zipper that results from focusing the sequence on element y . As can be seen, this structure consists

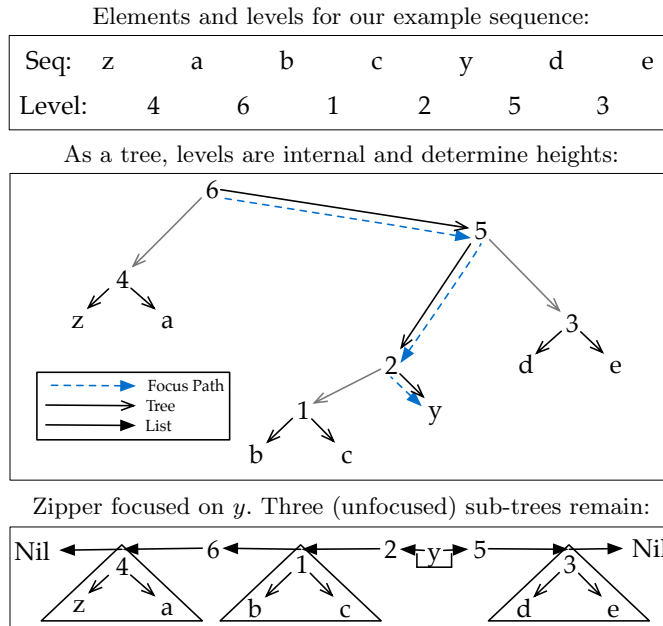


Fig. 1. The RAZ represents the sequence of elements $\langle z, a, b, c, y, d, e \rangle$ interposed with levels 1-6 (first image); these levels uniquely determine a balanced tree (second image) that permits log-time focusing on element y (third image).

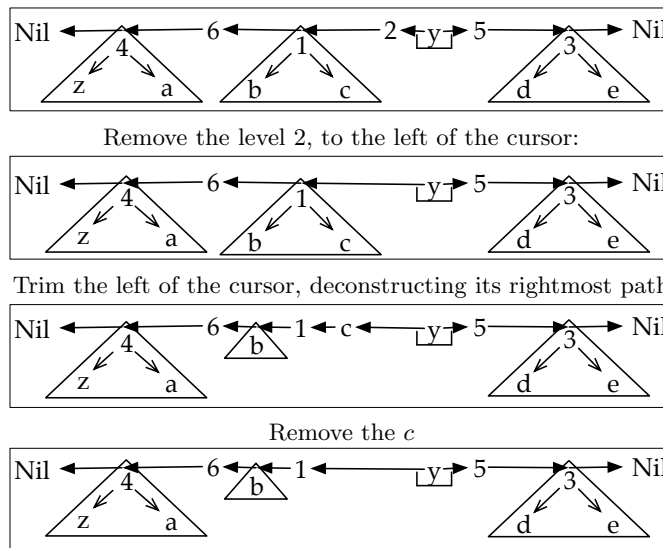
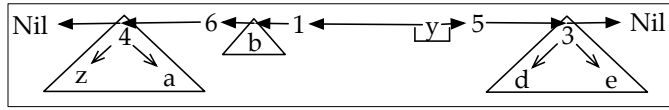
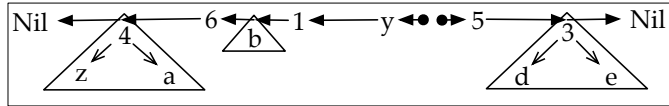


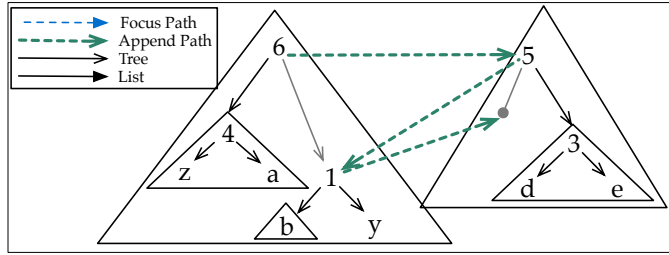
Fig. 2. An example of editing a focused RAZ: Remove the c to the left of the cursor by removing level 2 (second image), trimming the left tree (third image), and removing the c (last image)



Store element y in the sequence to be unfocused:



Append the left and right sequences, represented as trees:



Result of appending:

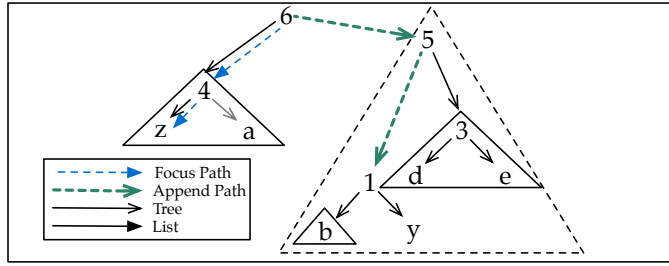


Fig. 3. Unfocus the RAZ

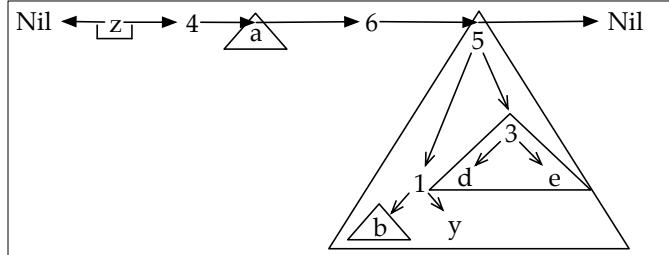


Fig. 4. Focus on the first element, z , creating left and right lists of (unfocused) subtrees.

of left and right lists that each contain levels and unfocused subtrees from the original balanced tree. The focusing algorithm produces this zipper by descending the balanced tree along the indicated focus path (second image of Figure 1), adding levels and subtrees along this path to the left and right lists, according to the branch taken. Notice that the elements nearest to the cursor consist of the subtrees at the end of this path; in expectation, these lists order subtrees in ascending size.

After focusing on element y , the programmer uses the `remove` function. Figure 2 shows the three steps for removing the c to the left of cursor. First, we remove the level 2 from the left of the cursor, making c the next element to the immediate left of the cursor (the second image). Next, since the c resides as a leaf in an unfocused tree, we *trim* this left tree by deconstructing its rightmost path (viz., the path to c). Section 3 explains the trim operation in detail. Finally, with c exposed in the left list, we remove it (fourth image).

After removing element c , the programmer unfocuses the RAZ. Beginning with the final state of Figure 2, Figure 3 illustrates the process of unfocusing the sequence. First, we add element y to one of the lists, storing its position in the sequence (second image). Next, we build trees from the left and right lists as follows: For each list, we fold its elements and trees, appending them into balanced trees; as with the initial tree, we use the levels to determine the height of internal nodes (third image). Having created two balanced trees from the left and right lists, we append them along their rightmost and leftmost paths, respectively; again, the append path compares the levels to determine the final appended tree (fourth image).

Finally, the programmer refocuses the RAZ. In Figure 4 (as in Figure 1), we descend the focus path to the desired element, this time z . As before, this path induces left and right lists that consist of levels and unfocused subtrees.

3 Technical Design

The full implementation of the RAZ in OCaml consists of about 170 lines. In this section, we tour much of this code, with type signatures for what we elide for space considerations.

Figure 5 lists the type definitions for the RAZ structure, which is stratified into three types: `tree`, `tlist`, and `zip`. The `tree` type consists of (unfocused) binary trees, where leaves hold data, and where internal binary nodes hold a level `lev` and total element count of their subtrees (an `int`). The `tlist` type consists of ordinary list structure, plus two `Cons`-like constructors that hold `levs` and `trees` instead of ordinary data. Finally, a (focused) `zip` consists of a left and right `tlist`, and a focused element that lies between them.

Levels for probabilistically-balanced trees. As demonstrated in the code below for `append`, the levels associated with each `Bin` node are critical to maintaining balanced trees, in expectation. This property of balance is critical to the time complexity bounds given for many of the RAZ's operations, including focusing, unfocusing and many local edits.

```

type lev = int (* tree level *)
type dir = L | R (* directions for moving/editing *)

type 'a tree = (* binary tree of elements *)
  | Nil
  | Leaf of 'a
  | Bin of lev * int * 'a tree * 'a tree

type 'a tlist = (* list of elements, levels and trees *)
  | Nil
  | Cons of 'a * 'a tlist
  | Level of lev * 'a tlist
  | Tree of 'a tree * 'a tlist
type 'a zip = ('a tlist * 'a * 'a tlist) (* tlist zipper *)

```

Fig. 5. RAZ defined as a zipper of tree-lists.

The key insight is choosing these levels from a *negative binomial distribution*; intuitively, drawing random numbers from this distribution yields smaller numbers much more often (in expectation) than larger numbers. More precisely, drawing the level 1 is twice as likely as drawing the level 2, which is twice as likely as level 3, and so on. This means that, in expectation, a sequence of levels drawn from this distribution describes the sizes of subtrees in a perfectly-balanced binary tree. As described in Section 6, this insight comes from [12], who define the notion of level in a related context.

Fortunately, we can choose these levels very quickly, given a source of (uniformly) random numbers and a hash function. We do so by hashing a randomly-chosen number, and by counting the number of consecutive zeros in this hash value's least-significant bits.

```

let focus : 'a tree → int → 'a zip =
fun t p →
  let c = elm_count t in
  if p >= c || p < 0 then failwith "out of bounds" else
  let rec loop = fun t p (l,r) → match t with
  | Nil → failwith "internal Nil"
  | Leaf(elm) → assert (p == 0); (l,elm,r)
  | Bin(lv, _, bl, br) →
    let c = elm_count bl in
    if p < c then loop bl p (l,Level(lv,Tree(br,r)))
    else loop br (p - c) (Level(lv,Tree(bl,l)),r)
  in loop t p (Nil,Nil)

```

Fig. 6. The focus operation transforms a tree into a zip.

Focusing the RAZ. The `focus` operation in Figure 6 transforms an unfocused tree to a focused zipper. Given an index in the sequence, `p`, and an $O(1)$ -time `elm_count` operation on sub-trees, the inner `loop` recursively walks through one path of `Bin` nodes until it finds the desired `Leaf` element. At each recursive step of this walk, the `loop` accumulates un-walked subtrees in the pair (l,r) . In the base case, `focus` returns this accumulated (l,r) pair as a `zip` containing the located leaf element.

Proposition 31 *Given a tree t of depth d , and an $O(1)$ -time implementation of `elm_count`, the operation `focus t p` runs in $O(d)$ time.*

```

let head_as_tree : 'a tlist → 'a tree
let tail : 'a tlist → 'a tlist

let grow : dir → 'a tlist → 'a tree =
  fun d t →
    let rec loop = fun h1 t1 →
      match t1 with Nil → h1 | _ →
        let h2 = head_as_tree t1 in
        match d with
        | L → loop (append h2 h1) (tail t1)
        | R → loop (append h1 h2) (tail t1)
    in grow (head_as_tree t) (tail t)

let unfocus : 'a zip → 'a tree =
  fun (l,e,r) → append (grow L l) (append (Leaf(e)) (grow R r))

```

Fig. 7. Unfocusing the RAZ using `append` and `grow`.

Unfocusing the RAZ. Figure 7 lists the `unfocus` operation, which transforms a focused zipper into an unfocused tree. To do so, `unfocus` uses auxiliary operations `grow` and `append` to construct and append trees for the left and right `tlist` sequences that comprise the zipper. The final steps of `unfocus` consists of appending the left tree, focused element `e` (as a singleton tree), and the right tree. We explain `append` in detail further below.

The `grow` operation uses `append`, and the simpler helper function `head_as_tree`, which transforms the head constructor of a `tlist` into a `tree`; conceptually, it extracts the next tree, leaf element or binary node level as `tree` structure. It also uses the function `tail`, which is standard. The `grow` operation loops over successive trees, each extracted by `head_as_tree`, and it combines these trees via `append`. The direction parameter `d` determines whether the accumulated tree grows from left-to-right (`L` case), or right-to-left (`R` case). When the `tlist` is `Nil`, the `loop` within `grow` completes, and yields the accumulated tree `h1`.

Under the conditions stated below, `unfocus` is efficient, running in polylogarithmic time for balanced trees with logarithmic depth:

Proposition 32 *Given a tree t of depth d , performing `unfocus (focus t p)` requires $O(d)$ time.*

We sketch the reasoning for this claim as follows. As stated above, the operation `focus t p` runs in $O(d)$ time; we further observe that `focus` produces a zipper with left and right lists of length $O(d)$. Likewise, `head_as_tree` also runs in constant time. Next, the `unfocus` operation uses `grow` to produce left and right trees in $O(d)$ time. In general, `grow` makes d calls to `append`, combining trees of height approaching d , requiring $O(d^2)$ time. However, since these trees were placed *in order* by `focus`, each `append` here only takes constant time. Finally, it `appends` these trees in $O(d)$ time. None of these steps dominate asymptotically, so the composed operations run in $O(d)$ time.

```
let rec append : 'a tree → 'a tree → 'a tree =
  fun t1 t2 →
    let tot = (elm_count t1) + (elm_count t2) in
    match (t1, t2) with
    | Nil, _ → t2 | _, Nil → t1
    | Leaf(_), Leaf(_) → failwith "leaf-leaf should not arise"
    | Leaf(_), Bin(lv,_,l,r) → Bin(lv, tot, append t1 l, r)
    | Bin(lv,_,l,r), Leaf(_) → Bin(lv, tot, l, append r t2)
    | Bin(lv1,_,t1l,t1r), Bin(lv2,_,t2l,t2r) →
      if lv1 >= lv2 then Bin(lv1, tot, t1l, append t1r t2)
      else Bin(lv2, tot, append t1 t2l, t2r)
```

Fig. 8. Append the sequences of two trees into a single sequence, as a balanced tree.

Appending trees. The `append` operation in Figure 8 produces a tree whose elements consist of the elements (and levels) of the two input trees, in order. That is, an in-order traversal of the tree result of `append t1 t2` first visits the elements (and levels) of tree t_1 , followed by the elements (and levels) of tree t_2 . The algorithm works by traversing a path in each of its two tree arguments, and producing an appended tree with the aforementioned in-order traversal property. In the last `Bin` node case, the computation chooses between descending into the sub-structure of argument t_1 or argument t_2 by comparing their levels and by choosing the tree named with the higher level. As depicted in the example in Figure 4 (from Section 2), this choice preserves the property that `Bin` nodes with higher levels remain higher in the resulting tree. Below, we discuss further properties of this algorithm, and compare it to prior work.

```

let trim : dir → 'a tlist → 'a tlist =
  fun d t1 → match t1 with
  | Nil | Cons _ | Level _ → t1
  | Tree(t, rest) →
    let rec trim = fun h1 t1 →
      match h1 with
      | Nil → failwith "malformed tree"
      | Leaf(elm) → Cons(elm,t1)
      | Bin(lv,_,l,r) →
        match d with
        | L → trim r (Level(lv,Tree(l,t1)))
        | R → trim l (Level(lv,Tree(r,t1)))
    in trim t rest

```

Fig. 9. Function `trim` exposes the next sequence element.

Trimming a tree into a list. The `trim` operation in Figure 9 prepares a `tlist` for edits in the given direction `dir`. It returns the same, unchanged `tlist` if it does not contain a tree at its head. If the `tlist` does contain a tree at its head, `trim` deconstructs it recursively. Each recursive call eliminates a `Bin` node, pushing the branches into the `tlist`. The recursion ends when `trim` reaches a `Leaf` and pushes it into the `tlist` as a `Cons`.

The `trim` operation works most efficiently when it immediately follows a re-focusing, since in this case, the cursor is surrounded by leaves or small subtrees, which each trim in constant time. If the cursor moves far through the zipper, however, it can encounter a node from high in the original tree, containing a significant proportion of the total elements of the sequence.

These facts suggest the following propositions:

Proposition 33 *Given a direction d , a position p , a tree t of size n , and a `tlist` l from one side of a zipper created by `focus t p`, `trim d l` runs in $O(1)$ time.*

Proposition 34 *Given a direction d , a position p , a tree t of size n , and a `tlist` l from one side of a zipper created by `focus t p`, a sequence of k calls to `trim d l` composed with `move d` runs in $O(k \log n)$ time.*

Figure 10 lists the code for inserting and removing elements from the zipper. The function `insert` uses `rnd_level` to generate a random level to accompany the newly-inserted element `ne`. Based on the removal direction, the function `remove` uses an internal helper `remove'` to remove the next sequence element in the given direction, possibly by looping. In particular, the `Cons` case is the base case that removes the element of the `Cons` cell; the `Nil` case is erroneous, since it means that there is no element to remove. The two remaining cases recur internally; specifically, the `Tree` case uses `trim`, explained above.

Figure 10 lists the type signatures of several other zipper-based editing functions: `view` accesses the next element to the right or left of the cursor, `replace`

```

let insert : dir → 'a → 'a zip → 'a zip =
  fun d ne (l,e,r) →
  match d with
  | L → (Level(rnd_level(),Cons(ne,l)),e,r)
  | R → (l,e,Level(rnd_level(),Cons(ne,r)))

let remove : dir → 'a zip → 'a zip =
  let rec remove' d s = match s with
    | Nil          → failwith "no elements"
    | Cons(_,rest) → rest
    | Level(lv,rest) → remove' d rest
    | Tree _       → remove' d (trim d s)
  in fun d (l,e,r) → match d with
  | L → (remove' L l,e,r)
  | R → (l,e,remove' R r)

let view      : dir → 'a zip → 'a
let replace   : dir → 'a → 'a zip → 'a zip
let move      : dir → 'a zip → 'a zip

let view_cursor : 'a zip → 'a
let replace_cursor : 'a → 'a zip → 'a zip

```

Fig. 10. Zipper edits: Element insertion, element removal and several other variants.

replaces this element with another given one, and `move` moves the cursor to focus on an adjacent element. Finally, `view_cursor` and `replace_cursor` are similar to `view` and `replace`, respectively, except that they act on the element at the cursor, rather than an element that is adjacent to it.

4 Evaluation

In this section we evaluate the RAZ in comparison to a data structure with similar theoretic behavior, the finger tree [8], which we elaborate on in related work, Section 6. We demonstrate that the RAZ is comparable in performance with this more complex structure.

Experimental Setup. We present two experiments, each performed on both a RAZ and a finger tree. In the first experiment, we construct a sequence from scratch by inserting elements, each in a randomly chosen position. Insertion into the RAZ is by focusing, inserting a value, then unfocusing; insertion into the finger tree is by splitting, pushing a value, then appending. Upon reaching a target length, we record the total time taken. We use target lengths of 10k to 1M elements, and repeat the process for a total of five data points for each target. We plot the median of the results, shown in Figure 11.

For the second experiment, we also insert elements into random positions in a sequence, but maintain a single sequence throughout the experiment. We measure the time taken for the first sequential group of 1M insertions, then the next group of 1M insertions, repeating until the sequence reaches 100M elements. We plot these measured times, shown in Figure 12.

We compiled our code through opam with ocamlc version 4.02 native mode, and ran it on a 2.8 GHz Thinkpad with 16 GB of RAM running Ubuntu 16.04. We use the “Batteries Included” [10] code for finger trees. The results in Figure 11 were collected by using a separate process for each measurement, and those in Figure 12 used one process per data structure. Ocaml’s `min_heap_size` parameter was set to 800MB. We report 100 values per data structure per plot.

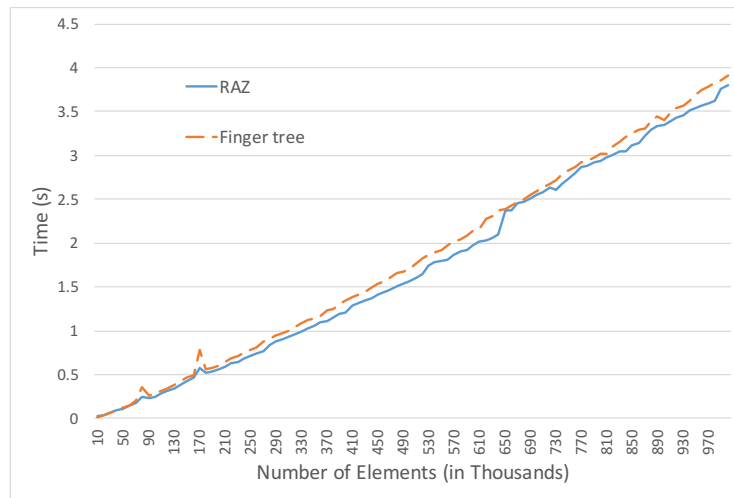


Fig. 11. Constructing sequences of different lengths from scratch

Results. Figure 11 shows the RAZ as slightly faster than a finger tree, but maintaining the same asymptotic behavior. We average the ratio of times, showing that the RAZ uses 5-6% less time on average. At 500k elements the RAZ takes 1.57s vs the finger tree’s 1.71s, and at 950k elements the times are 3.54s and 3.70s, respectively. Figure 12 shows a great variance in times. Even with a million elements added for each measurement, the plot is not smooth. This is true for the RAZ with its probabilistic structure, but also for the more consistently structured finger trees. The average time in the last 20 entries plotted is 10.01s for the RAZ and 9.77s for the finger tree. We suspect that garbage collection effects may be (partly) responsible for this variance, but determining this with certainty is beyond the scope of this initial evaluation.

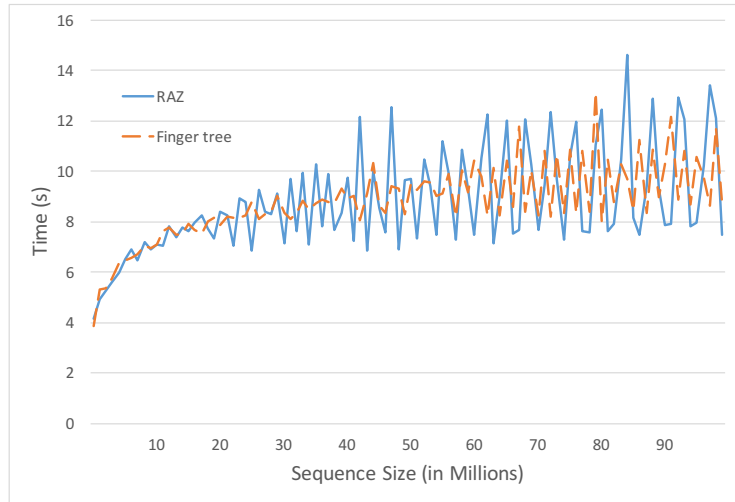


Fig. 12. Time taken to insert 1M elements into a sequence of varying size.

5 Discussion

A benefit of building a tree with specific heights is that of a stable form. The structure of the RAZ as a tree does not depend on the order of operations, but on the stored levels. This results in minimal changes, specifically, only the path from the adjusted elements to the root will be different. This is of great benefit for incremental computation, which is most effective for small changes. Early incremental structures by [12] used elements to determine heights, and had trouble with identical values. The RAZ sidesteps this issue with explicit stored levels for tree heights.

We struggled with the question of where to put the cursor when focusing into the RAZ. A RAZ may be focused on a element (as in the code described here), a level, or between the two. An early version of the RAZ placed the cursor between element and level, but the code was confusing and asymmetric. By focusing on an element, the code was reduced in length by about 25%, but may have made local edits a bit unintuitive. We have a rewrite that focuses onto a level, but it requires additional logic to deal with the ends of the sequence, which are bounded by elements.

There are two improvements we are considering as future work. One is to generalize the annotation of RAZ subtrees. The code presented here is annotated with size info to focus on a particular location. However, the RAZ could easily support arbitrary monoidal annotations of the kind described for finger trees in [8]. Another potential enhancement is to include arrays of elements in the leaves of the RAZ rather than a single element. Arrays may improve the speed of computations over the RAZ by allowing for better cache coherency. They may

also reduce the effect of probabilistic imbalance, by cutting down on local seek time.

6 Related Work and Alternative Approaches

We review related work on representing purely-functional (persistent) sequences that undergo small (constant-sized) edits, supplementing the discussions from earlier sections. We also discuss hypothetical approaches based on (purely-functional) search trees, pointing out their differences and short-comings for representing sequences.

The “Chunky Decomposition Scheme”. The tree structure of the RAZ is inspired by the so-called “chunky decomposition scheme” of sequences, from Pugh and Teitelbaum’s 1989 POPL paper on purely-functional incremental computing [12]. Similar to skip lists [11], this decomposition scheme hashes the sequence’s elements to (uniquely) determine a probabilistically-balanced tree structure. The RAZ enhances this structure with a focal point, local edits at the focus, and a mechanism to overcome its inapplicability to sequences of repeated (non-unique) elements. In sum, the RAZ admits efficient random access for (persistent) *sequence editing*, to which the ’89 paper alludes, but does not address.

Finger trees. As introduced in Section 1, a finger tree represents a sequence and provides operations for a double-ended queue (aka, deque) that push and pop elements to and from its two ends, respectively. The 2-3 finger tree supports these operations in amortized constant time. Structurally, it consists of nodes with three branches: a left branch with 1–4 elements, a center for recursive nodes, and a right branch with 1–4 elements. Each center node consists of a *complete* 2-3 tree. This construction’s shallow left and right (non-center) branches admit efficient, amortized constant-time access to the deque’s ends. This construction also provides efficient (log-time) split and append operations, for exposing elements in the center of the tree, or merging two trees into a single 2-3 tree. The split operation is comparable to the focus operation of the RAZ; the append operation is comparable to that of the RAZ.

While similar in asymptotic costs, in settings that demand *canonical forms* and/or which employ *hash consing*, the 2-3 finger tree and RAZ are significantly different; this can impact the asymptotics of comparing sequences for equality. In the presence of hash-consing, structural identity coincides with physical identity, allowing for $O(1)$ -time equality checks of arbitrarily long sequences. As a result of their approach, 2-3 finger trees are history dependent. This fact makes them unsuitable for settings such as memoization-based incremental computing [12,7,6].

Figure 13 depicts both a RAZ and a finger tree containing 15 elements, numbered sequentially. Elements are shown as circles, internal trees have a triangle around them, with diagonal lines denoting internal tree structure. Horizontal

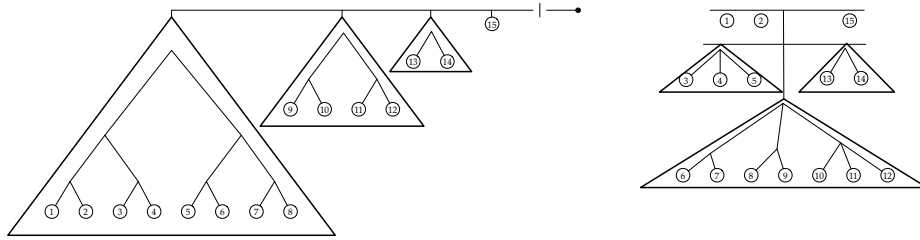


Fig. 13. A RAZ (left) and finger tree (right) representing the same sequence

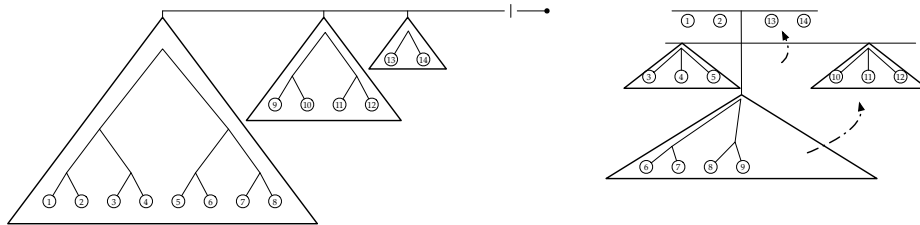


Fig. 14. Removing an element from a RAZ (left) and a finger tree (right), with structure maintenance on the finger tree

and vertical lines elide a simple structure for access to elements: a list in the RAZ case and an set of data types for the finger tree. Both these data structures provide access to the upper right element, labeled “15”. We elide the current item and right side of the RAZ, as it is not important for this example.

One major difference between the finger tree and RAZ is when they need to adjust their structure to maintain invariants. Figure 14 shows the result of deleting element 15 in both our example structures. They both expose this element, but the RAZ requires no maintenance at this point, while the finger tree does, since there are no elements left in the top finger. This is done by promoting a tree from the next deeper finger. In this case, the finger tree must promote another tree from even deeper. These promotions are indicated by the arrows in the figure.

RRB Vector. The *RRB-Vector* [14] uses a balanced tree to represent immutable vectors, focusing on practical issues such as parallel performance and cache locality. These performance considerations are outside the scope of our current work, but are interesting for future work.

Balanced representations of Sets. Researchers have proposed many approaches for representing sets as balanced search trees, many of which are amenable to purely-functional representations (e.g., Treaps [2], Splay Trees [13], AVL Trees [1], and Red-Black Trees [3]). Additionally, skip lists [11] provide a structure that is tree-like, and which is closely related to the probabilistic approach

of the RAZ. However, search trees (and skip lists) are *not* designed to represent sequences, but rather sets (or finite mappings).

Structures for sequences and sets are fundamentally different. Structures for sequences admit operations that alter the presence, absence and ordering of elements, and they permit elements to be duplicated in the sequence (e.g., a list of n repeated characters is different from the singleton list of one such character). By contrast, structures for sets (and finite maps) admit operations that alter the presence or absence of elements in the structure, but not *the order* of the elements in the structure—rather, this ordering is defined by the element’s type, and is not represented by the set. Indeed, the set representation uses this (fixed) element ordering to efficiently search for elements. Moreover, set structures typically do not distinguish between the sets with duplicated elements—e.g., `add(elm, set)` and `add(elm, add(elm, set))` are the same set, whereas a sequence structure would clearly distinguish these cases.

Encoding sequences with sets. In spite of these differences between sets and sequences, one can *encode* a sequence using a finite map, similar to how one can represent a mutable array with an immutable mapping from natural numbers to the array’s content; however, like an array, editing this sequence by element insertion and removal is generally problematic, since each insertion or removal (naively) requires an $O(n)$ -time re-indexing of the mapping. Overcoming this efficiency problem in turn requires employing so-called *order maintenance data structures*, which admit (amortized) $O(1)$ -time insertion, removal and comparison operations for a (mutable) total order [5,4]. Given such a structure, the elements of this order could be used eschew the aforementioned re-indexing problem that arises from the naive encoding of a sequence with a finite map. Alas, existing order maintenance data structures are *not* purely-functional, so additional accommodations are needed in settings that require persistent data structures. By contrast, the RAZ is simple, efficient and persistent.

7 Conclusion

We present the Random Access Zipper (RAZ), a novel data structure for representing a sequence. We show its simplicity by providing most of the code, which contains a minimal number of cases and helper functions. We describe some of the design decisions that increase simplicity. We evaluate the RAZ, demonstrating time bounds on par with far more complex data structures. Finally, we suggest multiple ways to enhance the RAZ to suit additional use cases.

References

1. M AdelsonVelskii and Evgenii Mikhailovich Landis. An algorithm for the organization of information. Technical report, DTIC Document, 1963.
2. 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, 1989.

3. Rudolf Bayer. Symmetric binary B-trees: Data structure and maintenance algorithms. *Acta Inf.*, 1:290–306, 1972.
4. Michael A. Bender, Richard Cole, Erik D. Demaine, Martin Farach-Colton, and Jack Zito. Two simplified algorithms for maintaining order in a list. In *Algorithms - ESA 2002, 10th Annual European Symposium, Rome, Italy, September 17-21, 2002, Proceedings*, pages 152–164, 2002.
5. Paul F. Dietz and Daniel Dominic Sleator. Two algorithms for maintaining order in a list. In *Proceedings of the 19th Annual ACM Symposium on Theory of Computing, 1987, New York, New York, USA*, pages 365–372, 1987.
6. Matthew A. Hammer, Joshua Dunfield, Kyle Headley, Nicholas Labich, Jeffrey S. Foster, Michael W. Hicks, and David Van Horn. Incremental computation with names. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2015, part of SPLASH 2015, Pittsburgh, PA, USA, October 25-30, 2015*, pages 748–766, 2015.
7. Matthew A. Hammer, Yit Phang Khoo, Michael Hicks, and Jeffrey S. Foster. Adapton: composable, demand-driven incremental computation. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '14, Edinburgh, United Kingdom - June 09 - 11, 2014*, page 18, 2014.
8. Ralf Hinze and Ross Paterson. Finger trees: A simple general-purpose data structure. *Journal of Functional Programming*, 16(2):197–217, 2006.
9. Gérard Huet. The zipper. *Journal of Functional Programming*, 1997.
10. ocaml-batteries team. Ocaml batteries included. <https://github.com/ocaml-batteries-team/batteries-included>. Accessed: 2016-07-12.
11. William Pugh. Skip lists: A probabilistic alternative to balanced trees. In *Algorithms and Data Structures, Workshop WADS '89, Ottawa, Canada, August 17-19, 1989, Proceedings*, pages 437–449, 1989.
12. William Pugh and Tim Teitelbaum. Incremental computation via function caching. In *POPL*, 1989.
13. Daniel Dominic Sleator and Robert Endre Tarjan. Self-adjusting binary trees. In *Proceedings of the 15th Annual ACM Symposium on Theory of Computing, 25-27 April, 1983, Boston, Massachusetts, USA*, pages 235–245, 1983.
14. Nicolas Stucki, Tiark Rompf, Vlad Ureche, and Phil Bagwell. RRB vector: a practical general purpose immutable sequence. In *ICFP 2015*, 2015.