A Catenable, Splittable, Transient Sequence Data Structure

ARTHUR CHARGUÉRAUD, Inria & U. de Strasbourg, CNRS, ICube, France
FRANÇOIS POTTIER, Inria, France

A transient data structure is a combination of an ephemeral data structure, a persistent data structure, and fast conversions between them. We present a *transient sequence* data structure that supports efficient read and write access at an arbitrary index with worst-case time complexity $O(K \log_K n)$, pushing and popping at either end with complexity $O(K \log_K n)$, and splitting and concatenation with complexity $O(K \log_K n + \log_K^2 n)$, where K is a user-defined *chunk size* and n is the length of the sequence. We provide a detailed analysis of this data structure and show that, in many favorable scenarios, it performs much better than these pessimistic bounds might suggest. Furthermore, we describe its implementation, and provide a synthetic benchmark to evaluate the performance of *push* and *pop*. We believe that it is a good candidate for a one-size-fits-all, general-purpose sequence data structure.

CCS Concepts: • **Theory of computation → Data structures design and analysis**.

Additional Key Words and Phrases: data structure, transient, persistent, snapshot

ACM Reference Format:
Arthur Charguéraud and François Pottier. 2026. A Catenable, Splittable, Transient Sequence Data Structure. *Proc. ACM Program. Lang.* 10, ICFP, Article 308 (August 2026), 33 pages. https://doi.org/10.1145/3828706

1 Introduction

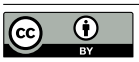
An *ephemeral* data structure is one that undergoes in-place updates, also known as destructive updates. Such updates directly correspond to the behavior of physical random-access memory; therefore they achieve maximal performance. In contrast, a *persistent* data structure supports non-destructive updates, which produce a new version of the data structure while preserving the original version. This can facilitate informal reasoning and formal verification. Furthermore, a key strength of persistent data structures is that they support taking snapshots in constant time—in fact, at no cost: every instance of the data structure is also a snapshot; the operation of taking a snapshot is implicit.

Depending on the application, the ability to take snapshots in constant time can yield asymptotic improvements over the linear-time copying required by ephemeral structures. In particular, parallel applications can benefit substantially from this ability: while some threads perform long-running analyses on snapshots, one or more other threads can concurrently process external queries and update the data structure [13].

Yet, compared with its ephemeral counterpart, an operation on a persistent data structure often is significantly more expensive. Sometimes it has greater asymptotic time complexity, such as $O(\log n)$ versus $O(1)$; sometimes it has the same asymptotic time complexity but a substantially larger constant factor.

Transient data structures aim to combine the benefits of ephemeral and persistent data structures. A transient data structure offers (1) the API of an ephemeral data structure, (2) the API of

Authors' Contact Information: Arthur Charguéraud, Inria & U. de Strasbourg, CNRS, ICube, Strasbourg, France, arthur.chargueraud@inria.fr; François Pottier, Inria, Paris, France, francois.pottier@inria.fr.



ACM 2475-1421/2026/8-ART308
https://doi.org/10.1145/3828706

a persistent data structure, (3) fast conversion operations between these forms. To enable fast conversions, the ephemeral structure and the persistent structure may *share* part of their internal representation. They allow programmers to take persistent snapshots (in constant time) when necessary, while retaining performance close to that of ephemeral data structures in sections of the code where persistence is not needed. This idea has been put to the fore by the programming language Clojure [16]. Since then, transient data structures have appeared in several libraries implemented in Scala, C++, Python, JavaScript, and Rust.

This paper presents constructions for efficient transient data structures that represent sequences of elements. We distinguish *arrays*, which support efficient random access, and *sequences*, which also support efficient insertion and deletion at either end, concatenation, and splitting.

Such sequences are extremely versatile. Naturally, they can act as FIFO stacks, LIFO queues, and resizable arrays (vectors). Furthermore, the fact that they can be split and joined makes them attractive in two major application areas.

First, they can serve as a representation of strings. Within a single program, it may be desirable in some places to rely on mutation for high-performance string construction or editing, while elsewhere treating strings as persistent values in order to support efficient substring extraction and concatenation. Besides, immutable strings simplify API specifications and reduce the risk of bugs. For example, Java and OCaml have immutable strings and offer separate data types of mutable strings and string buffers.

The second application concerns parallel divide-and-conquer algorithms in the fork-join model. For instance, to filter a sequence, one may split it into halves, recursively filter the two subsequences, and then concatenate the results. This strategy yields practical algorithms that are work-efficient and space-efficient. By moving from ephemeral to transient sequences, one gains the ability to efficiently take snapshots, without sacrificing the performance benefits of destructive updates during construction or processing.

Our construction of transient sequences exploits a per-node unique ownership principle proposed by L'orange [25] to optimize transient trees. In short, a tree node is tagged with an identifier, which encodes ownership information. In some cases, this identifier allows determining that this node is uniquely owned by this tree: that is, it is not shared with other trees. Then, an in-place update is safe. Compared with prior work on *Relaxed Radix Balanced* (RRB) trees [4, 32], we use a different tree structure, namely a persistent variant of *ephemeral chunked sequences* [2]. In such a tree, the elements that lie near the ends of the sequence are stored close to the root of the tree, hence are accessible at a smaller cost.

In addition to exploiting uniqueness to speed up operations on uniquely owned nodes, we propose techniques to decrease the cost of operations on shared nodes. To decrease the cost of *push* operations, we exploit *monotonic updates:* an update that initializes a previously uninitialized (unused) array slot is safe and can be performed in place. To decrease the cost of *pop* operations, we exploit *views*: a sequence can be described as a triple of a possibly larger persistent sequence and of two integers that characterize a subrange of that sequence.

In summary, this paper provides a self-contained presentation of a practical transient sequence data structure. We present its implementation as well as the time and space bounds associated with the ephemeral structure, the persistent structure, and the conversions between them. The paper is accompanied by a full-featured implementation in an OCaml library, named Sek, whose source code is available online [9]. Sek offers four abstract types (ephemeral sequences; persistent sequences; iterators on each of the two kinds of sequences) and about 170 operations on them.

We open the paper with the presentation of a simpler data structure, namely *transient arrays* (§2). They represent fixed-size sequences; they are practical and of independent interest. This lets us introduce high-arity trees whose nodes contain arrays as well as our technique for exploiting

```
type 'a earray              (*  ephemeral   (mutable) arrays *)
type 'a parray              (* persistent (immutable) arrays *)
module E : sig
  val create : int -> 'a -> 'a earray
  val get    : 'a earray -> int -> 'a
  val set    : 'a earray -> int -> 'a -> unit
end
module P : sig
  val create : int -> 'a -> 'a parray
  val get    : 'a parray -> int -> 'a
  val set    : 'a parray -> int -> 'a -> 'a parray
end
val snapshot : 'a earray -> 'a parray
val edit     : 'a parray -> 'a earray
```

Fig. 1. Transient arrays: interface

```
type 'a earray = 'a array     (* a mutable array *)
type 'a parray = 'a array     (* an array that is considered immutable *)
module E = struct
  let create n x = Array.create n x
  let get a i = a.(i)
  let set a i x = a.(i) <- x  (* update in place *)
end
module P = struct
  let create n x = Array.create n x
  let get a i = a.(i)
  let set a i x =             (* copy and update *)
    let b = Array.copy a in b.(i) <- x; b
end
let snapshot a = a            (* freeze: promise to never mutate again *)
let edit a = Array.copy a     (* create a fresh ephemeral copy *)
```

Fig. 2. Transient arrays: plain-array implementation

unique ownership. Then, we present our *transient sequences* (§3). There, we exploit the idea of a *chunked sequence* [2] as well as a representation of nodes as *persistent chunks*. Due to space constraints, we omit a discussion of iterators. We present our OCaml implementation of transient sequences, the manner in which we have tested it, and some preliminary benchmark results (§4).

2 Transient Arrays

2.1 Interface

The interface (that is, the API) of transient arrays appears in Figure 1. It offers an abstract type of ephemeral arrays, `earray`; an abstract type of persistent arrays, `parray`; and two operations, `snapshot` and `edit`, which convert between these types. Both kinds of arrays support the operations `create`, `get`, and `set`. On ephemeral arrays, `set` performs an in-place update and returns no result. On persistent arrays, `set` returns a new persistent array; the original array is unaffected.

	persistent	ephemeral
create	$O(n)$	$O(n)$
get	$O(1)$	$O(1)$
set	$O(n)$	$O(1)$
to-ephemeral (edit)	$O(n)$	–
to-persistent (snapshot)	–	$O(1)$
space usage in OCaml	$n+1$	
space usage in C	n	

Fig. 3. Transient arrays: costs of the plain-array implementation

2.2 Plain-Array Implementation

Figure 2 presents the simplest possible implementation of the interface in Figure 1. We refer to it as the *plain-array* implementation. An ephemeral array is represented as a primitive mutable array. A persistent array is also represented as a primitive array, which (by convention) is never modified in place. The conversion function `edit` requires a linear-time array copy operation. Updating a persistent array (`P.set`) requires first creating a copy of the input array, then updating a single cell in the copy—a *copy-on-write* operation.

The cost of each operation in the plain-array implementation is given in Figure 3. We provide two space bounds: one bound for OCaml implementations, one bound for C implementations. In OCaml, every memory block includes a one-word header. For example, an array of size n requires $n+1$ words. In C, a region-based memory allocation strategy can remove the need for block headers. We assume that this is the case; therefore we consider that an array of size n requires n words. However, if the application requires the ability to perform early deallocation of unreachable memory blocks, then one may need to employ reference-counting, which costs one word per block.

In summary, in the plain-array implementation of transient arrays, some operations have time complexity $O(1)$, while others cost $O(n)$. For arrays of bounded size, this implementation is theoretically optimal; provided a sufficiently small bound is chosen, it can be practical. Of course, as n grows large, the cost of the linear-time copies becomes prohibitive. Then, to obtain better performance, one must look for a tree-shaped data structure. In the following, we first recall the bounds that can be achieved using immutable trees; then we describe L'Orange's technique [2014] for introducing transience in a tree with a high branching factor.

2.3 Immutable Tree Implementation

An immutable tree allows non-destructive updates in time $O(D)$, where D is the depth of the tree. If the tree is balanced then D is $O(\log n)$, where n is the number of elements stored in the tree. A simple tree-based implementation of transient arrays represents a persistent array as a (balanced, immutable) tree and represents an ephemeral array as a reference to such a tree.

To begin, let us examine trees where each leaf stores two elements and each internal node has two subtrees.

```
type 'a parray = Leaf of 'a * 'a | Node of 'a parray * 'a parray     (* a tree *)
type 'a earray = 'a parray ref                              (* a reference to a tree *)
```

In such a tree, to find the element at index `i`, one traverses the tree from the root down to a leaf. At each node, one descends into either the left subtree or the right subtree, depending on the appropriate bit in the binary representation of the index `i`. The cost of a `get` or `set` operation is $O(D)$, that is, $O(\log_2 n)$. Regarding space usage, in OCaml, a binary tree with n items involves

	binary tree	K-ary tree
create	$O(n)$	$O(n)$
get	$O(\log_2 n)$	$O(\log_K n)$
set	$O(\log_2 n)$	$O(K \log_K n)$
to-ephemeral / to-persistent	$O(1)$	$O(1)$
space usage in OCaml	$3n$	$(1 + \frac{4}{K-1}) \cdot n$
space usage in C	$2n$	$(1 + \frac{1}{K-1}) \cdot n$

Fig. 4. Transient arrays: costs of the immutable-tree implementation

$\frac{n}{2}$ leaves of 3 words each and $\frac{n}{2} - 1$ nodes of 3 words each. (The constructors `Leaf` and `Node` both involve one header and two fields.) Hence, the total space usage is $3n$. In C, assuming that block headers are not needed, one can obtain a tighter bound. If the tree is perfectly balanced (all leaves lie at the same depth) and this depth is stored at the root, then no tag is needed to distinguish leaves and nodes. Then, a leaf or a node occupies just 2 words: hence the total space usage is $2n$.

This simple tree has branching factor 2. In practice, it is desirable to use a higher branching factor, at nodes and leaves. This has four major benefits: (1) reducing the number of memory accesses involved in accessing an element; (2) reducing the number of memory allocations required to construct an updated tree; (3) vastly reducing the space usage of the data structure; (4) improving locality and enabling fast batch processing of the information stored in the nodes and leaves. The main downside is increasing the cost of allocating a node. If the branching factor remains reasonably small, or if update operations are infrequent, then the benefits outweigh the costs.

In a tree with branching factor K, each leaf carries an immutable array of K elements, and each internal node carries an immutable array of K subtrees.

```
type 'a parray = Leaf of 'a array | Node of 'a parray array
```

For simplicity, we discuss just the case of perfectly balanced, complete trees, where the number of elements n is of the form K^D. Then there are K^{D-1} `Leaf` constructors, all of which lie at depth $D-1$, and each of which stores K elements. To handle the general case where $K^{D-1} < n \leq K^D$, it suffices to allow the arrays on the right fringe of the tree to store possibly fewer than K elements.

The depth D of the tree is $\log_K n$. Fetching the element at index `i` involves $\log_K n$ array read operations. The array indices involved in these operations are obtained by extracting batches of $\log_2 K$ consecutive bits from the binary representation of the index `i`. During a non-destructive update operation, $\log_K n$ nodes of size K must be allocated. Each new node consists of a copy of the previous node, with exactly one updated element—again an example of *copy-on-write*.

The space usage of such a tree is computed as follows. There are K^d arrays at depth d, with the `Leaf` arrays at depth $D-1$. Thus, the number of arrays involved in the tree is $1+K+K^2+\ldots+K^{D-1} = \frac{K^D-1}{K-1}$. Each of these arrays has size K. In OCaml, each `Leaf` or `Node` constructor requires 2 words, and each array requires $K+1$ words. Thus, the space usage is bounded by $(K+3) \cdot \frac{K^D-1}{K-1} \leq \frac{K+3}{K-1} \cdot K^D = (1 + \frac{4}{K-1}) \cdot n$. In C, assuming again that there is no need for headers and that the tree's depth is stored at its root, the space usage is only $(1 + \frac{1}{K-1}) \cdot n$.

The costs of immutable-tree implementations of transient arrays are summarized in Figure 4. In this implementation, an ephemeral array is just a reference to a persistent array; therefore the three operations on ephemeral arrays (create, get, set) and the three operations on persistent arrays have the same costs. Therefore we show these costs just once. We distinguish the case of binary trees and the case of trees of arity K.

As pointed out earlier, increasing K has numerous benefits. A large value of K implies a shallow depth. For example, for $K = 16$, as long as a tree stores at most 4 billion elements, its depth is at most 8. For $K = 512$, up to 134 million elements, the depth is at most 3. This explains why, for $K \geq 16$, the asymptotic bound $O(\log_K n)$ can be understood by a practitioner as "effectively constant time" [32].

The main consideration that prevents the use of a large K is the cost of updates, which is $O(K \log_K n)$. In the next section, we present a technique that decreases the cost of updates on the ephemeral flavor of a transient data structure to $O(\log_K n)$.

2.4 Transient K-Way Tree Implementation

In the previous section (§2.3), a persistent array is represented as an immutable tree. An ephemeral array is represented as a reference to such a tree; only this reference is mutable. In this section, for greater efficiency, we wish to use mutable trees to represent ephemeral and persistent arrays. We allow sharing: two trees may have a common subtree. Furthermore, we wish to update a node in place when this is safe, that is, when this node is not shared. This requires setting up a mechanism to detect sharing at runtime. This mechanism may be conservative: while a shared node *must* be recognized as such, a node that is not shared may safely be considered as possibly shared.

For this purpose, L'orange [25, §4.2] proposes a simple technique, whose correctness has been verified by Moine et al. [26, §2]. Let every ephemeral array have a unique identifier (ID). Furthermore, let every tree node carry an identifier. Maintain the following key invariant: if the ID carried by a node matches the ID of an ephemeral array, then this node is *uniquely owned* by this array. A uniquely-owned node is not shared with any other (ephemeral or persistent) array. Therefore it is safe for an operation on this ephemeral array to update this node in place.

A set of OCaml data type definitions that describe this approach appear in Figure 6. In comparison with the K-way trees of the previous section (§2.3), the only change is that every node and every ephemeral array carries an ID.

Figure 5 shows an example tree. In this illustration, an ID is an integer value. One could also implement IDs as addresses of dummy heap-allocated objects (that is, values of type `unit ref`). The tree in Figure 5 represents an ephemeral array. The ID that is depicted at the root of the tree, namely 8, is this array's unique ID. The three nodes that carry this ID are uniquely owned by this array and can be updated in place by an operation on this array. The nodes that carry an ID other than 8 are possibly shared with other (persistent or ephemeral) arrays and must not be modified.

When a persistent array is converted into a new ephemeral array, the latter receives a fresh ID. The nodes of the persistent array become shared with the new ephemeral array, and are correctly recognized as *possibly shared*, because their IDs do not match the fresh ID of the new ephemeral array. When an ephemeral array is updated, every node along the update path is either updated in place or copied. If a node is uniquely owned by this ephemeral array then it is updated in place. Otherwise, this node is copied: that is, it is replaced with a fresh copy, which is uniquely owned by the ephemeral array. Figure 5 illustrates a situation where exactly one path has been copied and is now uniquely owned by the ephemeral array.

To safely convert an ephemeral array into a persistent array, it suffices to assign a fresh unique ID to this ephemeral array and to return the underlying tree. Because the previous unique ID of this ephemeral array disappears, all nodes in the tree implicitly become immutable: they cannot be recognized as uniquely owned any more.

When an ephemeral array is updated, each node on the path to the desired element is updated or copied. An in-place update costs $O(1)$; a copy costs $O(K)$. Thus, in the worst case, the cost of an update is $O(K \log_K n)$, which is the same as an update of an immutable tree (§2.3). Nevertheless, there are numerous practical situations where an update is cheaper than this worst-case bound.

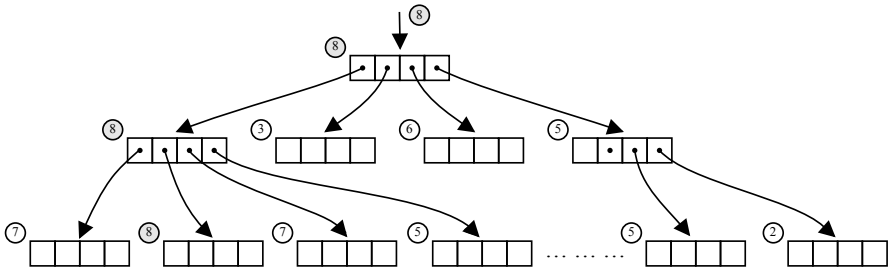


Fig. 5. A transient K-way tree. The tree has an ID (here, 8), and every node carries an ID.

```
type id        = int
type 'a node   = { id : id;  data : 'a array } (* the array [data] has size K *)
type 'a tree   =
  | Leaf of { elements: 'a node }      (* a node holding K elements *)
  | Node of { children: 'a tree node } (* a node holding K subtrees *)
type 'a earray = { depth : int; id : id; mutable tree : 'a tree }
  (* every node in [tree] whose ID is [id] is uniquely owned by this array *)
type 'a parray = { depth : int;                  tree : 'a tree }
  (* every node in [tree] is possibly shared and considered immutable *)
```

Fig. 6. Transient arrays: K-way tree implementation (type definitions)

One favorable scenario consists in applying several updates on an ephemeral array at the same index i or at nearby indices. Then, only the first update operation pays the worst-case cost $O(K \log_K n)$. Thereafter, all nodes on the path are uniquely owned; so, as long as they hit the same chunk, the following updates cost only $O(\log_K n)$.

Another favorable scenario is that of a *batch update*, where r consecutive elements of an ephemeral array are updated at once. Such a batch update can be performed by passing an array of r new values as an argument to the operation or by using an iterator. We are able to show that the cost of such a batch update is $O(r + K \log_K n)$. In particular, if r dominates $K \log_K n$, then this cost is only $O(1)$ per element.

Figure 7 summarizes the cost of operations on transient K-ary trees.

2.5 Varying the Arity

What is the optimal value of K? First, this depends on the ratio between reads and writes. At one extreme, if writes are extremely rare, then $K = n$ is optimal: implementing transient arrays as plain arrays is best. At the other extreme, if writes are much more frequent than reads, then a small value of K, perhaps as low as 2, is best. Experience suggests that, due to the benefits of locality, a value in the range of 8 to 128 is optimal in practice even in write-intensive applications.

Second, the choice of K depends on the cost of copying an array of size K. On modern hardware, memcpy is an *extremely fast* operation, typically implemented with SIMD instructions. In OCaml, for arrays of size up to 256, `Array.copy` calls memcpy. Thus, assuming $K \leq 256$, an array copy operation whose complexity is $O(K)$ may in practice be considered as "effectively constant time".

So far, we have assumed a uniform arity K for all the nodes in the tree. In general, one can use a different arity in each layer. We believe that it is desirable to use a larger value K_0 in the

	persistent	ephemeral
create	$O(n)$	$O(n)$
get	$O(\log_K n)$	$O(\log_K n)$
set	$O(K \log_K n)$	$O(K \log_K n)$
set on owned (recently modified) cells	–	$O(\log_K n)$
set on K consecutive cells	–	$O(\log_K n)$ per cell
set on $K \log_K n$ consecutive cells	–	$O(1)$ per cell
to-ephemeral (edit)	$O(1)$	–
to-persistent (snapshot)	–	$O(1)$
space usage in OCaml	$(1 + \frac{7}{K-1}) \cdot n$	
space usage in C	$(1 + \frac{2}{K-1}) \cdot n$	

Fig. 7. Transient arrays implemented as K-way trees: costs

bottom-most layer, at the leaves. The reason is that the space usage of a tree-based representation of transient arrays depends mainly on K_0.

3 Transient Sequences

This section generalizes from transient arrays, which correspond to fixed-sized sequences, to transient sequences, which support *push*, *pop*, *concat*, and *split* operations.

3.1 The Sek Tree Structure

Introducing support for concatenation and splitting gives rise to several challenges.

(1) Concatenation and splitting affect the cardinality of a sequence. Hence, a way of *dynamically adjusting the depth* of the tree is needed. This typically requires some form of *rebalancing*.
(2) After a *split* operation, the elements of a sequence are no longer aligned on multiples of powers of K. Hence, along its fringe, a tree may have *partially filled nodes*, which have fewer than K elements. If concatenation is also permitted then partially filled nodes can exist anywhere, not just along the fringe.
(3) In adversarial scenarios, after certain splitting and concatenation operations, a tree could have a majority of nodes that store very few elements. Some form of *compaction* is needed to guarantee that the average occupancy of a node remains above a certain threshold.
(4) In a standard K-way tree, all elements are stored deep in the tree. Therefore every element is accessed in time $O(\log_K n)$; the elements near the front and rear ends of the sequence cannot be accessed in time $O(1)$. Because pushing and popping at either end is frequent, we would like the elements near the two ends of the sequence to be stored near the root of the tree.

Relaxed Radix Balanced (RRB) trees [4] provide an answer to the first three challenges, but not to the last one. Instead, we propose a data structure that is inspired by *chunked sequences* [2]. Chunked sequences share similarities with *finger trees* [17] in that they store the two ends of the sequence at the root of the tree. Compared with finger trees, chunked sequences have slightly simpler (more regular) structure, based on nodes of arity up to K, and their costs can involve lower constant factors (as illustrated in our benchmark, §4.3). In the following, we first present a purely functional chunked sequence, which, compared with the original ephemeral chunked sequence [2], is slightly simplified (§3.8); then we adapt it to support transience.

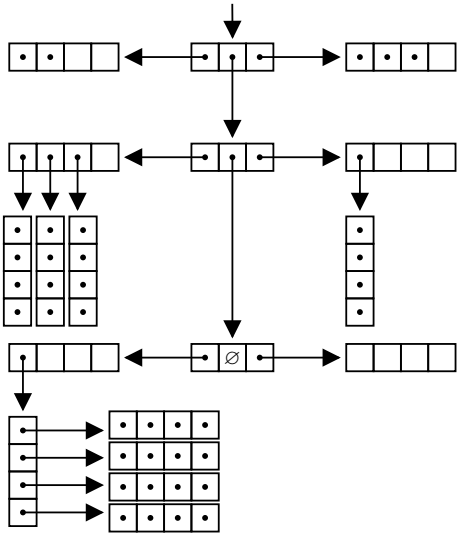


Fig. 8. The tree structure used in Sek. Each array (of size 4 in the picture) is a *chunk*. Each level consists of a front chunk, a back chunk, and a pointer to the next level (if there is one). The first level stores elements; the second level stores chunks of elements; the third level stores chunks of chunks of elements, and so on.

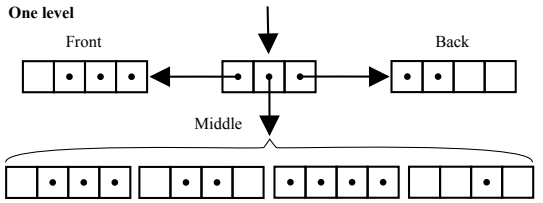


Fig. 9. Abstract view of one level of the tree structure: a front chunk, a back chunk, and a middle sequence of chunks, whose concrete structure is not shown. Each chunk is a circular buffer of capacity K. In the middle sequence, every two consecutive chunks must (together) store more than K elements.

Figure 8 offers a global view of our tree structure. The elements at the front and back of a sequence are stored in two arrays, the *front chunk* and the *back chunk*, which are directly accessible from the root of the tree. A chunk contains at most K elements. The remaining elements of the sequence, which form the middle of the sequence, are stored at greater depths in the tree. Furthermore, they are grouped in chunks: thus the *middle* sequence is recursively described as *a sequence of chunks of elements*. In summary, a *sequence of elements* either is empty or consists of *two chunks of elements* and a *sequence of chunks of elements*.

One *level* in this data structure is depicted in Figure 9. Each level must satisfy the following three key invariants, proposed by Acar et al. [2]:

(1) If the middle sequence is nonempty, then the front and back chunks must be nonempty.
(2) Every chunk in the middle sequence must be nonempty.
(3) Every two consecutive chunks in the middle sequence must together store more than K elements—otherwise, their content could fit in a single chunk.

To allow navigating the tree, every level and every chunk must store its *weight*, that is, the total number of *atomic elements* that it ultimately contains. In the case of a chunk of chunks of elements, for example, the weight of the outer chunk is not the number of its children; rather, it is the sum

```
type 'a chunk =
  { weight : int; data : 'a array } (* [data] is immutable *)

type 'a tree =
  | Empty
  | Level of { weight: int; front: 'a chunk; middle: 'a chunk tree; back: 'a chunk }
```

Fig. 10. Immutable chunked sequences: type definitions

of the weights of its children, that is, the total number of atomic elements that are transitively contained in the outer chunk.

An OCaml data type describing such a tree appears in Figure 10. Non-uniform recursion is involved: in a sequence of type `'a tree`, the `middle` field is a sequence of type `'a chunk tree`. Therefore, in general, an *item* of type `'a` is a chunk of chunks… of atomic elements.

The roles of the three aforementioned invariants are as follows. Invariants 1 and 2 ensure that the tree structure cannot be arbitrarily deep, and allow implementing fast emptiness tests. If concatenation is never used then all chunks stored in the middle sequence must be full. In this special case, one can prove that the depth is at most $\lfloor \log_K n \rfloor + 1$. If concatenation is used, then invariant 3, the *density invariant*, ensures that on average, a chunk is at least half full. Maintaining this invariant requires performing compaction as part of certain operations. Thanks to it, the depth of the tree is at most $\lfloor \log_{\frac{K+1}{2}} n \rfloor + 1$. When $K > 2$, this upper bound can be reformulated as: $\lfloor (1 + \frac{1}{(\log_2 K) - 1}) \cdot \log_K n \rfloor + 1$. The proofs may be found in Appendix B. To obtain our asymptotic time complexity bounds, it suffices to know that the depth is $O(\log_K n)$.

3.2 The Sek Tree Operations

The persistent data structure that we have just described represents *immutable sequences*. It supports *pop* with worst-case time complexity $O(\log_K n)$, *push*, *get*, and *set* with complexity $O(K \log_K n)$, and *concat* and *split* with complexity $O(K \log_K n + \log_K^2 n)$. Its implementation is analogous to that of ephemeral chunked sequences [2]. The main difference is that, in this immutable data structure, updating a chunk requires either ensuring that the update is monotonic or copying this chunk as well as the path that leads to it.

This section describes these operations at a high level. We include several auxiliary operations, namely *get-from-chunk*, *populate-sides*, *update-front*, and *merge*. Each operation is described from the perspective of one level, as depicted in Figure 9. We write f for the front chunk, m for the middle sequence, and b for the back chunk. We write w_f for the weight of the front chunk, w_m for the weight of the middle sequence, and w_b for the weight of the back chunk.

push-front. (*push-back* is symmetric.) The parameters of this operation are a level (f, m, b) and an element x. If f is not full, we create a chunk f' that contains x followed by the elements of f, then return the level (f', m, b). If f is full, two subcases arise, as follows. If b is empty, in which case (by invariant 1) m must also be empty, we return (f', m, f), where f' is a fresh chunk containing only x. If b is nonempty, we push the chunk f into the middle sequence m by means of a recursive call to *push-front*, which produces a new middle sequence m'. We build a singleton chunk f' that contains x and return (f', m', b). The cost is $O(K)$ per level. In the worst case, the recursive calls cascade all the way down through the tree: thus the total cost is $O(K \log_K n)$.

pop-front. (*pop-back* is symmetric.) This operation expects a level (f, m, b). There are two cases. First, suppose f is empty. By invariant 1, m must also be empty. We pop an element out of the back chunk b, yielding a pair (x, b'). Then, we return the element x and the level (f, m, b'). (If

this level is empty, we return just `Empty`.) Second, suppose f is nonempty. We pop an element out of f, obtaining a pair (x, f'). Then, we would like to return the element x and the level (f', m, b). However, (f', m, b) does not necessarily obey invariant 1: if f' is empty and m is nonempty then the invariant is violated. In such a situation, we recursively apply *pop-front* to the middle sequence m, obtaining a pair (f'', m'), where (thanks to invariant 2) f'' is nonempty. We then return (f'', m', b). Overall, *pop-front* executes a constant amount of work per level. In the worst case, the recursive calls to *pop-front* cascade all the way down, hence the total cost is $O(\log_K n)$.

populate-sides. This operation expects a level (f, m, b). If all three components are empty, it returns `Empty`; otherwise, it returns a level (f', m', b') that satisfies invariant 1, which means that if m' is nonempty then f' and b' must be nonempty. Assume that m is nonempty. If f is nonempty, then there is nothing to do on the front side. Otherwise, to populate the front chunk, we invoke *pop-front* on m, and obtain a pair (f', m'). By invariant 2, f' is nonempty. Then, starting from (f', m', b), we apply a similar process to populate the back chunk, applying *pop-back* to the middle sequence if needed. Overall, *populate-sides* performs at most two *pop*s, hence costs $O(\log_K n)$.

get-from-chunk. This operation expects a chunk c of items and an index i. A chunk of items can be viewed as a sequence of sequences of atomic elements. This operation locates the i-th atomic element, say x, in this sequence of sequences. It returns the element e of the chunk c that contains x as well as the index j of the element x relative to the first atomic element stored in e. In other words, j is i minus the sum of the weights of the elements that precede e in the chunk c. The implementation simply performs a linear scan, accumulating the weights of the elements that it scans, in time $O(K)$.

There is an important special case where this linear scan can be replaced with a computation whose cost is $O(1)$. This occurs when all elements in the chunk c have maximal weight, that is, weight K^d, where d is the current depth. In such a case, we say that the chunk c is *packed*. Then, because all elements have the same weight, the desired index can be computed via a division. Furthermore, to determine in time $O(1)$ whether a chunk is packed, it suffices to test whether its weight is equal to $n \cdot K^d$, where n is the logical length of this chunk.

Recognizing and accessing packed chunks in time $O(1)$ seems quite important in practice. Furthermore, we remark that if concatenation is not used in the construction of a sequence then every chunk in this sequence must be packed. Therefore, in the absence of concatenation, we obtain better worst-case time complexity bounds for some operations, such as *get*.

get. This operation expects a level (f, m, b) and an index i. Like *get-from-chunk*, it returns an element e that contains the i-th atomic element x, as well as the index of x relative to the first atomic element stored in e. There are three cases:

- If $i < w_f$ then the desired element is located in the front chunk. Hence the result is obtained by applying *get-from-chunk* to f and i.
- If $i \geq w_f + w_m$ then the desired element is located in the back chunk. Hence the result is obtained by applying *get-from-chunk* to b and $i - w_f - w_m$.
- Otherwise, the desired element is located in the middle sequence. Applying *get* to m and $i - w_f$ (a recursive call) yields a chunk c and an index j. Then the result is obtained by applying *get-from-chunk* to c and j.

Overall, the cost of *get* is $O(K)$ per level, hence $O(K \log_K n)$. As explained above, if every chunk is packed then this cost drops down to $O(1)$ per level, hence $O(\log_K n)$.

set. This operation descends into the tree by distinguishing the same cases as *get*; then it builds a modified tree. At each level, one chunk must be updated, therefore copied. Overall, the cost of *set* is $O(K)$ per level, hence $O(K \log_K n)$.

split. This operation expects a sequence s and an index i and returns a pair of sequences (s_1, s_2) such that s_1 has length i and s is the concatenation of s_1 and s_2. It is implemented in terms of a more general *3-way split* which expects a level (f, m, b) and an index i and returns a triple (s_1, e, s_2) such that the element e contains the atomic element (say x) located at index i in the sequence $f; m; b$ and s is the concatenation of the sequences s_1, e, and s_2. A 2-way *split* is achieved by first performing a 3-way split, then pushing the central element into the front of the second sequence.

The implementation of 3-way split follows the same case distinction as *get*. If the desired index i is located in the front (resp. back) chunk then it is easy to construct a sequence of the elements that precede index i (resp. follow index $i + 1$) and to then pop the desired element x. If the desired index is located in the middle sequence, then a recursive call to 3-way split is performed at index $i - w_f$. The result is a triple (m_1, c, m_2). The chunk c is then split in three parts, namely: an element e that contains the index $i - w_f - w_{m_1}$ in c, and two chunks c_1 and c_2 that contain the elements of c that precede and follow e. The final output is a triple of the level (f, m_1, c_1), the element e, and the level (c_2, m_2, b). At this stage, there is no guarantee that the two freshly constructed levels satisfy invariant 1. To ensure that they do, it suffices to populate the back side of the left level and the front side of the right level.

At each level, splitting requires local operations on chunks whose cost is $O(K)$, plus two calls to *populate-sides*, whose cost is $O(\log_K n)$. The total cost of the 3-way *split*, which descends into $O(\log_K n)$ levels, is therefore $O(K \log_K n + \log_K^2 n)$. A 2-way *split* involves a 3-way split followed by a *push* whose cost is $O(K \log_K n)$. Thus the cost of a 2-way split is also $O(K \log_K n + \log_K^2 n)$.

update-front. (*update-back* is symmetric.) This operation expects a nonempty level (f, m, b) and an element e; it replaces the first element of the sequence with e. While *update-front* is semantically equivalent to a combination of *pop-front* and *push-front*, the point of this operation is that it can be implemented without any recursive calls, achieving a time complexity of $O(K)$ instead of $O(K \log_K n)$. Its implementation is as follows. If f is nonempty, then the first element of the sequence is the first element of f: it suffices to build a chunk f', a copy of f whose first element is replaced with e, and to return the level (f', m, b). If f is empty then, by invariant 1, m is empty as well, so the first element of the sequence is the first element of b, which is replaced in the same way. In either case, because the shape of the level is preserved, no invariant can be broken.

concat. This operation expects two nonempty levels (f_1, m_1, b_1) and (f_2, m_2, b_2) and returns a level that represents the sequence $f_1; m_1; b_1; f_2; m_2; b_2$. As a preliminary step, if f_1 is empty (in which case, by invariant 1, m_1 must be empty as well) then we exchange f_1 and b_1, thereby ensuring that f_1 is nonempty. Symmetrically, we ensure that b_2 is nonempty. We then return the level $(f_1, \mathit{merge}(m_1, L, m_2), b_2)$, where L is the list of chunks $[b_1; f_2]$. The preliminary steps require time $O(1)$; therefore the cost of *concat* is the cost of *merge*.

merge. The auxiliary operation *merge* expects two sequences of chunks m_1 and m_2, as well as a list L of chunks, such that the length of L is at most 4. The operation constructs a sequence of chunks that represents the concatenation $m_1; L; m_2$, and that satisfies the density invariant (invariant 3). We first describe the two terminal cases, which involve no recursive call to *merge*.

In the first terminal case, m_1 and m_2 are both empty. Then *merge* fuses adjacent chunks in the list L, insofar as possible. Two adjacent chunks are fused if, together, they contain at most K elements. Then, this list of at most 4 chunks is easily turned into a sequence of chunks.

In the second terminal case, exactly one of m_1 and m_2 is empty, say m_1; the other case is symmetric. Let q be the first chunk in m_2. First, the adjacent chunks in the list $L; q$ are fused, insofar as possible, yielding a compressed list L'. Then, using *update-front*, the chunk q is replaced in m_2 with the last element of L'. Finally, using *push*, the remaining elements of L' are pushed onto the front of m_2.

In the non-terminal case, both m_1 and m_2 are nonempty. Let p be the last chunk of m_1 and q be the first chunk of m_2. Both of these chunks take part in the fusion: we fuse adjacent chunks, insofar as possible, in the list $p; L; q$, This yields a list of chunks $r_1; \ldots; r_j$, which satisfies the density invariant, and whose length j is comprised between 1 and 6.

If $j = 1$, we replace p with r_1 in m_1 via *update-back* and extract q out of m_2 via *pop-front*. In this case, we let L' denote the empty list. Otherwise, we have $j \geq 2$. We replace p with r_1 in m_1 via *update-back*; symmetrically, we replace q with r_j in m_2 via *update-front*. In this case, we let L' denote the list of chunks $r_2; \ldots; r_{j-1}$, which remain to be inserted between m_1 and m_2.

Let (F_1, M_1, B_1) and (F_2, M_2, B_2) stand for the levels that represent m_1 and m_2. By performing exchanges if necessary, as in *concat*, one can assume that F_1 and B_2 (which are chunks of chunks) are nonempty. As long as there is room, the elements of the list L' (which are chunks) are stored into B_1 and into F_2. The remaining elements of L' are stored in fresh chunks (of chunks). (If $K \geq 4$ then one fresh chunk suffices. In all cases, two fresh chunks suffice.) Let L'' be the list formed of B_1, the fresh chunks just mentioned, and F_2. The list L'' has length at most 4. The final result is the level $(F_1, \mathit{merge}(M_1, L'', M_2), B_2)$.

At each level, *merge* performs a constant number of operations on chunks whose cost is $O(K)$ plus possibly one *pop-front* operation, whose cost is $O(\log_K n)$. Because there are $O(\log_K n)$ levels, the total cost of these operations is $O(K \log_K n + \log_K^2 n)$. Furthermore, in the terminal cases only, *merge* performs a bounded number of *push* operations, whose cost is $O(K \log_K n)$. The total cost of *merge* is the sum of these two bounds, that is, $O(K \log_K n + \log_K^2 n)$.

3.3 Chunks with Persistence and Transience

We now present two techniques that aim to enable cheap *push* and *pop* operations. Both techniques involve in-place updates of mutable arrays, removing the need to copy an array (and the path that leads to it). First, we propose an implementation of persistent chunks that allows *monotonic* in-place updates of possibly-shared chunks. Second, we introduce ownership identifiers, as in the previous section (§2.4), so as to allow in-place updates of *uniquely-owned* chunks.

We want a *persistent chunk* to behave as a persistent (apparently immutable) data structure. Yet, for efficiency, we wish to take advantage of in-place updates. This is permitted as long as these updates cannot be observed. Our idea is to define a persistent chunk as a combination of a partially-filled mutable array of elements, the *support*, and a segment of this array, the *view*. We refer to the support also as a *raw chunk*. Several persistent chunks may share a common support. Updating a support in place, by filling a previously empty slot, is permitted provided that this update takes place *outside of the view* of every persistent chunk that points to this support.

Ultimately, we want the support to be double-ended, that is, to act as a (fixed-capacity) circular buffer, so that our persistent chunks support *push* and *pop* at either end. For the moment, though, to simplify our explanation, we present persistent chunks that support *push* and *pop* at the right end only. They are (fixed-capacity) persistent stacks.

As indicated above, a persistent chunk is a pair of a support and a view (Figure 11). A *support* includes a *data* array of capacity K and an integer *size* that delimits an occupied, immutable prefix and an unoccupied, mutable suffix. This size is comprised between 0 and K. A *view* is just an integer index, which delimits a possibly smaller occupied immutable prefix of the support: its value is at most *size*. A single support can be shared by several persistent stacks, with possibly different views.

```
type 'a support = { data : 'a array; mutable size : int; default : 'a }
  (* an array of capacity K *)
  (* an immutable prefix of this array: 0 <= size <= K *)
  (* beyond size, the array is mutable *)

type 'a pstack = { support : 'a support; view : int }
  (* a support and a view onto it: 0 <= view <= support.size *)

let get (p : 'a pstack) (i : int) : 'a =
  assert (0 <= i && i < p.view);        (* cannot access outside of the view *)
  p.support.data.(i)

let copy_and_update (data : 'a array) (i : int) (x : 'a) : 'a array =
  let data = Array.copy data in
  data.(i) <- x;
  data

let set (p : 'a pstack) (i : int) (x : 'a) : 'a pstack =
  assert (0 <= i && i < p.view);        (* cannot access outside of the view *)
  let support = { p.support with data = copy_and_update p.support.data i x } in
  { p with support }

let pop (p : 'a pstack) : 'a * 'a pstack =
  let n = p.view in
  assert (n > 0);                           (* cannot pop from an empty stack *)
  p.support.data.(n-1),
  { support = p.support; view = n - 1 }

let push (p : 'a pstack) (x : 'a) : 'a pstack =
  let n = p.view in
  let s = p.support in
  assert (n < Array.length p.support.data); (* cannot push into a full stack *)
  let support =
    if n = s.size then
      ( s.data.(n) <- x; s.size <- n + 1; s )   (* in-place monotonic update *)
    else
      { data = copy_and_update s.data n x; size = n + 1 }
  in
  { support; view = n + 1 }
```

Fig. 11. Persistent fixed-capacity stacks: implementation

In a support, we also store a default value, which we use to fill a slot that has been logically emptied by a pop operation. The need to fill empty slots with a default value is rather painful, for two reasons: first, it has a cost; second, the need to supply a default value is visible in the API that we present to the end user. To remove this need, we would very much like OCaml to offer primitive support for using a memory block as a circular array. The garbage collector would recognize such a block and would avoid scanning its logically empty slots; thus there would be no need to fill them with a default value.

The remainder of Figure 11 describes the operations. To read a persistent stack at index `i`, we access its support at index `i`. To update a persistent stack, we copy and update its support.

(The code that is shown copies all of the data; it would suffice to copy just the data in the view.) To pop an element from a persistent stack, we simply shrink its view by one unit. When pushing an element into a persistent stack, two cases arise:

- If the view is *aligned* with the support, meaning that the size of the support coincides with the view, then the incoming element can be written in place in the support array. The `size` field of the support is incremented, and the `view` field of the new persistent stack is the new size of the support. In this case, the *push* operation costs $O(1)$. This is a monotonic update: an empty slot becomes an occupied slot.
- Otherwise, `view` must be less than `size`, and a new support must be allocated. This situation typically arises when an independent *push* operation on this persistent stack has already taken place. In this case, the *push* operation costs $O(K)$.

A monotonic update, as described above, is not *thread-safe*: if two threads concurrently attempt to extend the same support, a data race will occur. To make our implementation safe in a concurrent setting, we envision two approaches. One approach is to avoid the data race by using an atomic operation: we leave the details to future work. The other approach is to abandon in-place monotonic updates. The tradeoff is between executing an atomic operation on every *push* versus allocating an array of length K on every *push*.

This sums up our approach in the case of one-ended stacks. Now, we want to go from stacks to double-ended queues, that is, to allow *push* and *pop* at either end. To do so, we add one field, in the support and in the view, to represent the index of the first element. We let the type `range` describe a possibly circular segment of indices. Moreover, as in the previous section (§2.4), we include an `id` field, which lets us recognize whether a chunk is uniquely owned or shared. A key invariant is: if a chunk is uniquely owned, then its `view` field and the `range` field of its support coincide.

Figure 12 shows our data types as well as the implementation of some operations. The *get* operation on chunks accounts for the fact that the index of the access is relative to the view. The *push* operation that is shown pushes an element at the right end of a chunk that is part of some ephemeral tree. It compares the identifiers of the chunk and of the tree to determine whether the chunk is uniquely owned. If so, an in-place update is possible (`ephemeral_push`). Otherwise, the chunk must be considered persistent (`persistent_push`): then an in-place update is possible if it is monotonic, that is, if the support's end index and the view's end index are aligned and if there is still an empty slot in the support at this index.

As mentioned earlier (§3.1 and Figure 10), each chunk must store its *weight*. In Figure 12, for greater readability, the `weight` field and the instructions that update this field are omitted.

The cost of the operations on persistent chunks is summarized in Figure 13. Regarding space usage, we assume that the type `range` fits in one word. In OCaml, a "support" object requires 5 words for the record of type `support` plus $K + 1$ words for the `data` array, hence $K + 6$ words in total. A chunk requires 4 words for a record of type `chunk` plus $K + 6$ words for the support. (The support may or may not be shared with other chunks.) Hence, a chunk requires at most $K + 10$ words in total. In C, a support requires $K + 2$ words and a chunk requires 3 words plus a support, hence at most $K + 5$ words in total.

3.4 Sek Tree with Transience

Let us put together all the ingredients that we have described so far. We have:

(1) The tree structure described earlier (§3.1).
(2) The idea of transience, where identifiers are used to recognize unique ownership (§2.4).
(3) The transient chunks described in the previous section (§3.3),
which consist of circular views on a circular support.

```
type range = { head : int; size : int }
  (* 0 <= head < K && 0 <= size <= K *)
  (* in reality, head and size can be compressed into one word *)

type 'a support = { data : 'a array; mutable range : range; default : 'a }
  (* occupied: from [head] included to [head+size] excluded, circularly *)

type 'a chunk = { support : 'a support; mutable view : range; id : id }
  (* [view] must be a subrange of [support.range] *)
  (* for simplicity, the [weight] field and its maintenance are omitted *)

let extend range =                    (* extend a range at its right end *)
  { range with size = range.size + 1 }

let wrap (i : int) : int =         (* handle wrap-around at the right end *)
  if i < k then i else i - k

let get (c : 'a chunk) (i : int) : 'a =
  assert (0 <= i && i < c.view.size);    (* cannot access outside of view *)
  c.support.data.(wrap (c.view.head + i))

let ephemeral_push (c : 'a chunk) (x : 'a) : 'a chunk =
  assert (c.view = c.support.range);               (* we must be aligned *)
  c.support.data.(wrap (c.view.head + c.view.size)) <- x;
  c.support.range <- extend c.support.range;
  c.view <- extend c.view;
  c

let persistent_push (tree_id : id) (c : 'a chunk) (x : 'a) : 'a chunk =
  let s = c.support in
  let r = s.range in
  let n = c.view.size in
  let view_back = wrap (c.view.head + n) in       (* the view's end index *)
  let support_back = wrap (r.head + r.size) in (* the support's end index *)
  let support, id =
    if r.size < k && view_back = support_back then begin
      s.data.(support_back) <- x;
      s.range <- extend s.range;
      s, c.id
    end else
      let data = Array.init k @@ fun i ->
        if i <= n then s.data.(wrap (c.view.head + i)) else x
      and range = { head = 0; size = n + 1 } in
      { data; range }, tree_id
  in
  let view = extend c.view in
  { support; view; id }

let push (tree_id : id) (c : 'a chunk) (x : 'a) : 'a chunk =
  if c.id = tree_id then ephemeral_push c x else persistent_push tree_id c x
```

Fig. 12. Transient chunks: implementation (weight maintenance omitted)

	persistent	ephemeral
create	$O(K)$	$O(K)$
push, pop, set if chunk is uniquely owned	–	$O(1)$ remains uniquely owned
get	$O(1)$	$O(1)$
pop	$O(1)$	$O(1)$
set	$O(K)$ then support aligned	$O(K)$ then uniquely owned
push if support is aligned	$O(1)$ then remains aligned	$O(1)$ then remains aligned
push if support is not aligned	$O(K)$ then support aligned	$O(K)$ then uniquely owned
to-ephemeral (edit)	$O(K)$ then uniquely owned	–
to-persistent (snapshot)	–	$O(1)$ then support aligned
space usage in OCaml	$K + 10$	
space usage in C	$K + 5$	

Fig. 13. Transient chunks of capacity K: costs

```
type 'a ptree =                                  (* a persistent sequence *)
  | Empty of { default : 'a }
  | Level of { weight  : int;
               front   : 'a chunk;               (* chunk as in Figure 12 *)
               middle  : 'a chunk ptree;
               back    : 'a chunk }
type 'a etree =                                  (* an ephemeral sequence *)
  { id : id; mutable tree : 'a ptree }
```

Fig. 14. Transient chunked sequences: type definitions (without optimizations)

The (unoptimized) data types of our persistent and ephemeral sequences are shown in Figure 14. In the type `ptree`, the constructor `Empty` carries a default value, which is used to initialize new arrays when a new level is constructed. The constructor `Level` does not carry a default value because such a value may be found, for example, inside the front chunk.

In the following, we present two further optimizations. For persistent sequences, the proposed optimization improves the space usage of "short" sequences. For ephemeral sequences, the proposed optimization ensures that the amortized complexity of *push* and *pop* is "effectively constant" and aims to reduce the associated constant factors.

3.5 Optimizations for Persistent Sek

A persistent sequence that stores only a few elements nevertheless has a space usage of $O(K)$ words, because the front and back chunks involve arrays of capacity K. Yet we would like its space usage to be $O(n)$, where the constant factor is independent of K. Thus, we include additional constructors in the data type `'a ptree`, with the aim of optimizing the memory representation of short sequences.

```
type 'a ptree =                                  (* a persistent sequence *)
  | Empty of { ... }                             (* as in Figure 14 *)
  | One   of { default : 'a; x : 'a }
  | Short of { default : 'a; data : 'a array }
  | Level of { ... }                             (* as in Figure 14 *)
type 'a etree =                                  (* an ephemeral sequence *)
  { id : id;
    mutable front  : 'a support;                 (* front chunk *)
    mutable ifront : 'a support;                 (* inner front chunk *)
    mutable middle : 'a chunk ptree;
    mutable iback  : 'a support;                 (* inner back chunk *)
    mutable back   : 'a support }                (* back chunk *)
```

Fig. 15. Transient chunked sequences: type definitions (with optimizations)

For the case $n = 1$, we introduce a constructor `One`. A sequence of length n, represented via such a length-specific constructor, requires $n + 2$ words. For the case of "short" sequences, whose length is greater than 1 and at most T, we introduce a constructor `Short`, which lets us store the elements of a sequence of length n in an array of length n. This representation requires $n + 4$ words. We explain in §4.1 how we set the threshold T. In our asymptotic complexity analyses, we assume $T = \Theta(K)$.

Our modified definition of the type `'a ptree` is shown in Figure 15. In our implementation, we allow the constructors `One` and `Short` to appear at the top level only: this makes the code slightly simpler. As an alternative design, one could let `One` and `Short` appear at any depth. That would make the memory footprint slightly smaller. Our space bounds assume the latter design.

3.6 Optimizations for Ephemeral Sek

Our unoptimized definition of the type `'a etree` is a record whose fields are an ID and a tree of type `'a ptree` (Figure 14). We adopt the convention that the tree must have the constructor `Level` at its root; then we unbox this constructor, that is, we turn its arguments into fields of the record type `'a etree`. To represent the empty sequence, we allow the front and back chunks to be both empty. We suppress the `weight` field, thereby saving a write on every *push* or *pop* operation. The weight of an ephemeral sequence is obtained by adding the weights of the components `front`, `ifront`, `middle`, `iback`, and `back`. Our modified definition of the type `'a etree` is shown in Figure 15.

We adopt the convention that the front and back chunks at the top level are uniquely owned by the ephemeral sequence. Therefore the fields `front` and `back` can have type `'a support` as opposed to `'a chunk`. This removes an indirection and removes the need to compare identifiers at every *push* and *pop* operation.

In addition, an ephemeral sequence stores two extra chunks, named *inner front* and *inner back*. Each of these two chunks must be either empty or full. It stores elements that lie between the *front* (resp. *back*) chunk and the middle sequence. An empty inner chunk is always represented by a dedicated *dummy* chunk that is allocated once and for all. The motivation for these inner chunks has been described in earlier work on ephemeral chunked sequences [2]; we discuss their role further on (§3.8). In short, they forbid certain pathological scenarios where two *push* operations cascade all the way down the tree, two subsequent *pop* operations again cascade all the way down, and this can be repeated at will.

The amortized cost analysis of *push* and *pop* operations performed by Acar et al. [2] in the case of ephemeral sequences cannot be applied to transient sequences. There are two main reasons why

	persistent	ephemeral
create of length n	$O(n)$	$O(n+K)$ then all chunks uniquely owned
get	$O(K \log_K n)$	
set	$O(K \log_K n)$	$O(K \log_K n)$ then chunks in path uniquely owned
pop	if $n \leq T+1$ then $O(T)$ else $O(\log_K n)$	$O(\log_K N)$ amortized
push	if $n \leq T-1$ then $O(T)$ else $O(K \log_K n)$	$O(\log_K N)$ amortized
concat	$O(K \log_K n + \log_K^2 n)$	
split	$O(K \log_K n + \log_K^2 n)$	
to-ephemeral (edit)	$O(K \log_K N)$ amortized $O(K)$ in isolation	–
to-persistent (snapshot)	–	if $n \leq T$ then $O(n)$ else $O(1)$
space usage in OCaml	$(2 + \frac{22}{K-1})n + O(K \log_K n)$	
space usage in C	$(2 + \frac{12}{K-1})n + O(K \log_K n)$	
space usage in OCaml when concat is not used	$(1 + \frac{11}{K-1})n + O(K \log_K n)$	
space usage in C when concat is not used	$(1 + \frac{6}{K-1})n + O(K \log_K n)$	

Fig. 16. Transient sequences with chunks of capacity K: costs. T denotes the threshold for a representation as an array of length exactly n. N denotes an upper bound on the length of the ephemeral sequence.

this is so. First, Acar et al. charge the cost of a *pop* operation onto the corresponding *push* operation. In their setting, this is sound because an element that is pushed (inserted) into the data structure can be popped (extracted) at most once. In our setting, this affine usage property is violated by *edit*, because the result of *edit* is a persistent data structure, which can be used many times. Second, our complexity bound for *push* and *pop* is not $O(1)$: because we store inner front and inner back chunks only at the top level, it is $O(\log_K n)$. We discuss this point in greater depth later on (§3.8).

Our key novel result is that *push* and *pop* at either end of an ephemeral sequence have amortized cost $O(\log_K N)$. This holds even though the sequence `middle` possibly contains shared chunks, which possibly cannot be updated in place. Our analysis relies on the convention that *edit*, which converts a persistent sequence into an ephemeral one, advertises an amortized cost of $O(K \log_K N)$. Here, to simplify the proof, we let N denote an upper bound on the size that the ephemeral sequence might reach. The proof may be found in Appendix B. It is based on a potential function defined as $\left(2 \cdot |K - n_{\mathsf{front}} - n_{\mathsf{ifront}}| + (K - n_{\mathsf{middle\text{-}fst}})\right) \cdot O(\log_K N)$ for the front, plus a symmetric amount for the back, where n_{front} is the size of `front`, n_{ifront} is the size of `ifront`, and $n_{\mathsf{middle\text{-}fst}}$ is the size of the first chunk in `middle` and is K if there is no such chunk.

3.7 Analysis of Sek's Transient Sequences

Figure 16 summarizes the costs of all operations, in terms of the length n, the chunk capacity K (where $K \geq 2$), and the threshold T, which is assumed to be $\Theta(K)$. A sequence is *short* if its length n is at most T. A short persistent sequence is represented as an array of length exactly n.

The creation of a persistent sequence has time complexity $O(n)$. Indeed, in the case of a short sequence, creation obviously costs $O(n)$, and in the case of a long sequence, whose length is $\Omega(K)$,

creation costs $O(K + n)$, which in this case is the same as $O(n)$. Creating an ephemeral sequence costs $O(K + n)$. The presence of the summand K is due to our decision to allocate two arrays (namely, the front and back chunks) up front in order to decrease the cost of subsequent operations.

As explained in §3.2, the worst-case cost of *get* and *set* is $O(K \log_K n)$. Indeed, each level may require a linear scan (and, in the case of *set*, a copy) whose cost is $O(K)$; and the depth of the tree is $O(\log_K n)$. If every chunk is packed (§3.2) then the linear scan disappears so *get* costs only $O(\log_K n)$. If every chunk is packed and uniquely owned then the linear scan and the copy disappear so *set* costs only $O(\log_K n)$.

In the worst case, a *pop* operation on a persistent or ephemeral structure costs $O(\log_K n)$, due to the possibility of cascading operations (§3.2). That said, if a *pop* operation affects only the top level then its cost is only $O(1)$, and (intuitively) this should be the most frequent case.

If a *pop* operation on a persistent sequence results in a short sequence then the sequence is converted into an array representation, at cost $O(T)$. This conversion is required to preserve the space bounds; setting $T = 0$ deactivates this feature.

In the worst case, a *push* operation on a persistent or ephemeral structure costs $O(K \log_K n)$, due to the possibility of cascading operations and to the potential need for copy-on-write operations at every level, as explained in §3.2. That said, if a *push* operation affects only the top level then it is cheaper: its cost is $O(1)$ if the sequence is ephemeral, $O(1)$ if the sequence is persistent and a monotonic update is possible, and $O(K)$ otherwise.

Our key result (§3.6) is that a series of *push* and *pop* operations on an ephemeral sequence has amortized cost $O(\log_K N)$ per operation, where N denotes an upper bound on the length of the sequence. This analysis also allows *edit* with an amortized cost of $O(K \log_K N)$.

Concatenation and splitting each cost $O(K \log_K n + \log_K^2 n)$, as explained in §3.2.

Converting a persistent sequence into an ephemeral one (*edit*) requires time $O(K)$ to allocate fresh front and back arrays, plus $O(K \log_K N)$ to pay for the potential function associated with ephemeral *push* and *pop* operations (§3.6). Thus the worst-case cost of *edit* in isolation is $O(K)$, but its amortized cost as part of a scenario that involves also *push* and *pop* operations is $O(K \log_K N)$.

Converting a long ephemeral sequence into a persistent one (*snapshot*) requires merely wrapping the front and back arrays into chunks, which costs $O(1)$. Converting a short ephemeral sequence into a persistent one requires copying its elements into an array, which costs $O(n)$.

The space bounds appear near the bottom of Figure 16. For these bounds, in order to simplify the statements and proofs, we assume $K > 2$. The proofs may be found in Appendix B. To illustrate how a space bound can be interpreted, assume that the structure is used as a stack (so concatenation is not used), with $K = 16$. The asymptotic space usage is roughly $1.74n$ in OCaml, and $1.4n$ in C. This asymptotic space usage is strictly better than that of vectors, which require up to $2n$ words as the vector grows, and up to $4n$ words as the vector shrinks. (Other growth factors are possible for vectors, but smaller growth factors significantly worsen the constant factors of *push* and *pop*).

3.8 Discussion about the Role of Inner Chunks

In Acar et al.'s *ephemeral chunked sequences* [2], *inner front* and *inner back* chunks appear at every level of the tree structure. In contrast, our *transient sequences* (§3.1) are described by the type `ptree` (Figures 14 and 15), which does not have inner chunks. Our *ephemeral sequences* (§3.6) are described by the type `etree` (Figure 15), which does have inner chunks; but this type describes only the top level of the data structure, that is, only the root of the tree. In this section, we explain the motivation for these design decisions.

In Acar et al.'s purely ephemeral setting, the presence of inner chunks at every level is necessary to achieve $O(1)$ amortized time complexity for *push* and *pop*. It defeats a bad scenario where one

alternates between *push* and *pop* and where (in the absence of inner chunks) every operation would cost $O(K)$. This is a *linear* scenario, where each operation acts on the result of the previous one.

In our ephemeral sequences, the presence of inner chunks at the top level is sufficient to protect against a similar worst-case scenario that could cost $O(K \log_K n)$ per operation—the cost of a *push* operation that cascades all the way down. The presence of inner chunks ensures that this cost is amortized over at least K operations; therefore our *push* and *pop* operations on ephemeral sequences have amortized complexity $O(\log_K N)$. This bound appears in Figure 16 and is proved in §B.3. We view it as *effectively constant time*.

In the case of persistent sequences, one must also consider *non-linear* scenarios, where the result of an operation can be used as an argument in several (independent) subsequent operations. There, one cannot exploit the same amortization argument as in the *linear* scenarios. Our persistent sequences, *with or without inner chunks*, would have the same worst-case cost, namely $O(\log_K n)$ for *pop* and $O(K \log_K n)$ for *push*.

The worst-case scenario for *push* first constructs a sequence where every chunk at every level is full; then, the next *push* operation reaches the worst-case cost $\Omega(K \log_K n)$. Because the sequence is persistent, this costly *push* can be repeated at will: thus its cost cannot be amortized.

In summary, exploiting inner chunks only in the ephemeral variant and only at the top level suffices to achieve the desired bounds. Storing inner chunks in the persistent variant or in the deeper levels of the ephemeral variant would not allow us to offer improved bounds. Moreover, it would make the implementation more complex, and could degrade the constant factors of other operations. All these observations justify our design decision to include inner chunks only for the ephemeral structure, and only at the top level.

4 Implementation

4.1 OCaml Implementation

Our implementation of transient sequences in OCaml is available via the `opam` package manager. It is about 6000 non-blank, non-comment lines of code plus about 4000 non-blank lines of comments. The user is allowed to adjust several settings, such as the chunk size K. In fact, we allow the chunk size to depend on the depth where the chunk appears in the tree. By default, we use 128 at the leaves and 16 at internal nodes. The other settings are:

- When an array slot becomes logically empty due to a *pop* operation, should it be overwritten with a default value? Leaving it alone saves one write but (due to tracing garbage collection) gives rise to a potential memory leak. Overwriting is safer; it is the default.[1]
- How to set the threshold T where one switches from the simple-minded representation of persistent sequences (§3.5) to the more complex general representation? A low threshold implies switching early. Because switching implies allocating two chunks of size K, switching early implies using a lot of space per element. On the other hand, switching late implies tolerating costly operations on short sequences. Our default threshold is 32.
- Should the use of an invalid iterator be detected at runtime? This defensive mechanism detects programmer mistakes but adds overhead. By default, it is enabled.

The code is organized in six main modules:

(1) ephemeral chunks, also known as raw chunks: partially empty circular arrays;
(2) shareable chunks: raw chunks wrapped with view, owner, and weight information;
(3) shareable sequences, implemented on top of shareable chunks;
(4) ephemeral sequences, implemented as a shareable sequence surrounded by two raw chunks;

[1]This setting applies to ephemeral sequences only. In persistent sequences, *pop* reduces a view (§3.3) and does not update the underlying support. Thus the popped value remains accessible and a potential memory leak cannot be avoided.

(5) iterators on (ephemeral or shareable) sequences;
(6) code that handles our special representation of short persistent sequences.

In our ephemeral sequences, the front and back chunks are raw chunks: they are always uniquely owned. Our shareable chunks have an optional mechanism to detect the situation where a chunk has been considered shared in the past, yet is deemed uniquely owned again: this must be a bug. Enabling this mechanism during random testing has been extremely useful and has helped us recognize some proposed variations of our unique ownership discipline as incorrect.

4.2 Testing and Verification

Implementing a transient data structure in a correct manner is generally challenging. This is particularly true here, as our data structure has a complex tree structure and relies on a subtle unique ownership and sharing discipline. To address this challenge, we have performed intensive random testing of our library.

The library's API involves 4 abstract data types and about 170 functions. Therefore, inventing and implementing testing scenarios by hand seems intractable: it would be time-consuming and would result in poor coverage. Furthermore, because our data structure has mutable internal state, we cannot just test simple QuickCheck-style properties such as $pop\,(push\,x\,xs) = x$. We must construct complex, non-linear scenarios where several instances of the data structure, possibly sharing common ancestors, are operated upon. We must construct only legal scenarios, where the precondition of each operation is met: for example, an index that is passed to a *get* or *set* operation must be within bounds; and an iterator on an ephemeral sequence must not be used after it has been invalidated, that is, after the sequence has been modified.

For this purpose, we have used Monolith [30]. Monolith provides a set of combinators that let us describe the specification of each operation, including its type and its precondition. Monolith builds well-typed, legal scenarios and runs them. In each run, the candidate implementation is monitored and is compared with a reference implementation, which the user provides. In our case, the reference implementation implements transient sequences using lists and references. If the candidate implementation crashes or if the two implementations exhibit different behaviors then Monolith reports the problematic scenario and starts searching for shorter problematic scenarios, thereby performing a form of implicit automated shrinking. Monolith can perform purely random testing or collaborate with a fuzzer such as AFL [34].

The reference implementation not only enables a comparison of the observable outcomes of the two implementations, but also serves as an oracle during the construction of legal scenarios. By answering questions such as "what is the length of this sequence?" and "is this iterator still valid?", it can check whether a proposed operation is valid and can help efficiently generate valid operations.

The issue of iterator validity is somewhat subtle, as our library operates in one of two modes (§4.1): a mode where an attempt to use an invalid iterator is detected and a mode where such an attempt is forbidden. In the first mode, we generate scenarios where invalid iterators are used and we check that these illegal uses are correctly detected by the candidate implementation. In the second mode, we avoid generating such scenarios. In either case, we rely on the reference implementation to determine which iterators are valid.

Monolith allows the candidate implementation to provide self-validation functions, which check that the invariant of a data structure holds. Monolith invokes these functions at each step in a test scenario: this increases the likelihood that a bug is quickly detected. In our library, these functions are non-trivial: their total size is about 200 non-blank lines of code. As an example, Figure 19 in Appendix A shows a validation function for persistent sequences.

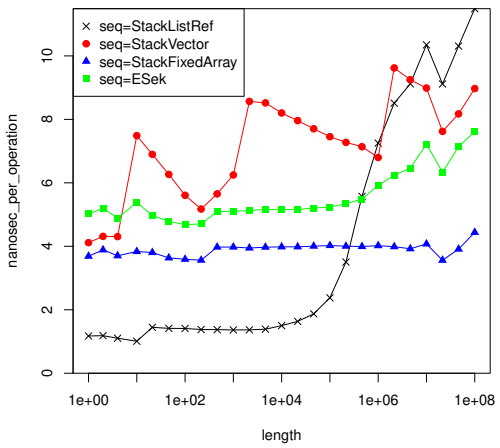


Fig. 17. *push*/*pop* on ephemeral sequences

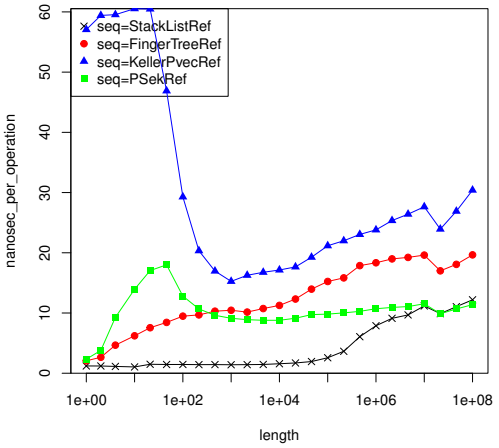


Fig. 18. *push*/*pop* on persistent sequences

Beyond testing, program verification is another way of detecting bugs. Furthermore, it allows verifying the functional correctness and possibly the time complexity of an implementation. In prior joint work with Moine [26], we have verified a scaled-down version of our library, which offers transient *stacks* as opposed to transient *sequences*, using CFML [10], an implementation of Separation Logic for OCaml. The chunk ownership discipline is the same in both libraries. Due to its much larger scale, verifying our library would be a major endeavor, which we leave to future work.

4.3 A Limited Benchmark

Benchmarking all aspects of a transient data structure is a formidable task. An operation cannot be analyzed in isolation. Its performance depends not only on the logical size(s) of the data structure(s) to which it is applied but also on the internal structure of the data: whether the chunks are uniquely owned or shared, whether they are aligned (§3.3), whether they are close to being full, and so on. An in-depth comparison of several transient sequence structures under a set of realistic scenarios would require an entire paper.

Although we have performed several benchmarks, in this paper, we present just one. We measure the performance of *push* and *pop* on our ephemeral and persistent sequences and compare them with several preexisting data structures. In Figure 17, we compare our ephemeral sequence data structure with a linked list (StackListRef), a resizable array (or vector) whose capacity is doubled when the vector is full and halved when its occupancy is less than $\frac{1}{4}$ (StackVector); and a stack held in a large array that is never resized (StackFixedArray). In Figure 18, we compare our persistent sequence data structure with a linked list (StackListRef), a finger tree [12] (FingerTreeRef), and a persistent vector [22] that is reportedly inspired by Clojure's persistent vectors (KellerPvecRef).

Our benchmark scenario has two integer parameters p and n. It can be described as follows: starting with an empty stack, repeat p/n times: perform n *push* operations, followed by n *pop* operations. It is a linear scenario: only one data structure is involved and every operation is applied to the current version of this data structure. Persistence is not exploited.

We fix p, the total number of *push* operations, to 100 million, so that each run takes a few seconds. We use a growing series of values of n, the peak length of the stack. The elements of the stack are integer values. When this is relevant, the data structures under test are configured to overwrite a cell that is emptied by a *pop* operation with a default value. Sek is configured with its default parameters (§4.1), so that, for $n \leq 100$m, the depth is at most 6. Our benchmark machine is an Apple MacBook Pro equipped with an M2 Max processor and 64GB of main memory. We use OCaml 4.14.2+flambda with a minor heap size of 8 MB. Each dot in the chart is the average of 3 runs.

Ephemeral sequences. The results of our benchmark for ephemeral sequences appear in Figure 17. These results can be summarized as follows.

First, we confirm the folklore knowledge: in OCaml, for sequences of small to moderate size, lists are approximately twice as fast as arrays. Indeed, memory allocation is extremely cheap, whereas mutation involves a write barrier. For greater values of n, an array-based stack outperforms a list-based stack. The break-even value depends on factors such as the size of the minor heap and the size of the hardware caches. In our benchmark it is about 10^5. At $n = 10^8$, arrays are about 3x faster than lists. The main explanation, we believe, is that the memory footprint of a list is 3x larger; this leads to more cache misses and increased GC activity.

Compared with arrays, Sek's ephemeral sequences are at most 1.5x slower up to $n = 10^6$, and at most 2x slower in general. We are not surprised to find that Sek is slower than an array whose fixed size is n. The fact that Sek is *only* 1.5x-2x slower than arrays is quite an achievement, we believe, considering that Sek efficiently supports a much wider range of operations.

Compared with vectors, Sek's ephemeral sequences are consistently faster, except at tiny sizes. They can be faster by a factor of up to 2x. A decisive advantage of Sek is that, for stacks and queues, once an element has been stored in a chunk, it never needs to be copied.

Compared with a reference on an immutable list, Sek's ephemeral sequences can be up to 3x slower. However, beyond roughly 1 million elements, Sek significantly outperforms lists.

Persistent sequences. Our benchmark scenario is *not* a natural use case of our persistent sequences: in such a linear scenario, it would be more natural to use an ephemeral sequence and to convert it to a persistent sequence only if and when necessary. Nevertheless, this benchmark lets us evaluate the cost of (artificially) using a persistent data structure all along. Its results appear in Figure 18.

For large stacks ($n \geq 10^7$), our persistent sequences are just as fast as lists, and are about twice as fast as finger trees. For smaller stacks, lists can be much faster; but lists do not support efficient splitting and concatenation operations.

For short stacks ($n \leq 200$), our persistent sequences are up to twice slower than finger trees. This seems mainly due to the fact that we impose a plain-array representation of short sequences ($n \leq T$) in order to optimize their space usage. The value of the threshold T may need to be fine-tuned depending on the needs of each application. We believe that it is still an open problem to find a persistent representation of short sequences that offers reasonable space usage, reasonable costs for *push* and *pop*, and a smooth transition towards a tree representation that supports deque-style operations, split, concatenation, and transience.

In comparison with our persistent sequences and with finger trees, we find Keller's persistent vectors [22] to be consistently slower, especially at small sizes ($n \leq 1000$).

5 Related Work

5.1 Persistent Catenable and Splittable Sequences

Driscoll et al. [14] propose a generic method, based on "fat nodes" and version numbers, to turn an ephemeral linked data structure into a persistent one. From a theoretical perspective, their result is

very strong. In practice, however, its applications are hindered by the large constant factors that this method introduces in the time complexity of every operation. Furthermore, this method does not allow binary operations, such as concatenation. Driscoll et al. do not explore the possibility of decreasing asymptotic costs under certain access patterns, as achieved by transient data structures.

Another generic way of constructing a persistent or transient data structure is to implement an ephemeral version of it on top of a store that supports snapshots, such as the `store` library [3]. In fact, there are loose similarities between the implementations of Sek and `store`. `store`'s key optimization, record elision, relies on the fact that there is no need to record undo information for a write operation to location ℓ if no snapshot has been taken since the previous write to ℓ took place. The absence of a snapshot means that the intermediate state has not been shared and is not observable. Thus, in essence, uniqueness enables an in-place update. That said, `store` does not have monotonic updates and does not seem to have built-in support for arrays, though it does allow the user to define their own "storable types" [11]. We expect Sek to be more efficient than ephemeral chunked sequences running on top of `store`. It would be interesting to test whether this is true and (if so) to find out what improvements to `store` might allow narrowing the gap.

Kaplan and Tarjan obtain a series of results about catenable persistent sequences, with or without support for efficient random access and splitting [19–21]. These data structures use *recursive slowdown*, which is a combination of two ideas: first, an operation at depth $d + 1$ should be needed only once every K operations at depth d; second, the place where work is needed should be accessible in constant time. This technique can achieve excellent worst-case time complexity bounds; it is however quite complex. Kaplan and Tarjan's real-time deques [21] have been implemented in OCaml and verified using Rocq by Viennot et al. [33], who provide some benchmarks. Okasaki [27] [28, §11.2] and Kaplan et al. [18] propose simpler implementations of persistent catenable deques without splitting. Both rely on mutable internal state, which in both cases is updated in a monotonic way; this explains why in-place updates are safe even though the data structure may be shared. The data structure from Kaplan et al. [18] has been implemented in OCaml and verified using Rocq and Iris by Ponsonnet and Pottier [29].

Buchsbaum [8] introduces the idea of *structural decomposition*, which Okasaki [28, §10.1] connects with *non-uniform recursion*. As a typical example, Okasaki implements a *sequence* of items as either an empty sequence or a pair of one item and a *sequence* of pairs of items. Non-uniform recursion is also exploited in *chunked sequences* [2], whose structure we reuse in Sek.

Buchsbaum [8] also introduces the idea of *structural abstraction*, which reduces concatenation to insertion by inserting one data structure into the other. Okasaki [28, §10.2.1] presents an illustration of this idea: he defines a *catenable list* as either empty or a pair of one element and a queue of catenable lists. This is known as a nested data type. One can view this data type as the description of a tree where nodes have flexible and unbounded arity. In Sek, the nodes have bounded capacity K, which still offers significant flexibility.

Hinze and Paterson [17] present a simple sequence data structure, *finger trees*, which is inspired by Okasaki's implicit queues [28, §11] but supports efficient splitting. A *finger tree* of items is either empty, a singleton, or a triple of a front buffer, a lazy finger tree of nodes, and a back buffer. A *buffer* is a packet of 1 to 4 items. A *node* is a packet of 2 to 3 items. This data structure supports *push* and *pop* operations at either end with amortized time complexity $O(1)$ as well as splitting and concatenation with amortized time complexity $O(\log n)$. Finger trees have been implemented and verified in Rocq by Sozeau [31].

Whereas finger trees support *push* and *pop* with amortized time complexity $O(1)$, our persistent sequences support *push* and *pop* with worst-case time complexity $O(K \log_K n)$ and $O(\log_K n)$, respectively (Figure 16). That said, in many scenarios, we expect our *push* and *pop* operations to be more efficient than predicted by the worst-case bound.

5.2 Sequences Storing Elements in Chunks

Since the early days of programming, sequences have often been represented as linked lists of *fixed-capacity arrays*, also known as *chunks*. This data structure is ubiquitous in the source code of hardware drivers. Chunks also appear in the C++ standard library class `std::deque` [15]. There, a deque is represented as a vector of chunks, where the vector serves as a resizable circular buffer. Pointers to the first and last items allow constant-time access to the two ends. Random access is very efficient: every item is accessible via just two indirections. On the other hand, concatenation and splitting have linear time complexity. Furthermore, the performance of *push* and *pop* can degenerate to $\Theta(K)$ in some scenarios of the form "push-push-pop-pop" that fall just on the edge of a chunk [2].

An ephemeral unbalanced *rope* [6] is a tree data structure that represents a sequence and supports efficient concatenation. A leaf carries an array of items; a binary node describes the concatenation of two sequences. With unbalanced ropes, concatenation has complexity $O(1)$, but random access and splitting require linear time in the worst case. An ephemeral balanced rope [6] is obtained by applying a balancing strategy that guarantees logarithmic depth. Then, random access, splitting, and concatenation have worst-case time complexity $O(\log n)$. A persistent balanced rope can be obtained by making nodes immutable and by storing persistent arrays at the leaves.

Acar et al. [2] describe *ephemeral chunked sequences*, a data structure that allows *push* and *pop* operations with amortized complexity $O(1)$ and concatenation and splitting with amortized complexity $O(K \log_K n)$. They implement chunked sequences in C++ and report that, on *push* and *pop* operations, it is competitive with vectors. Our data structure is a chunked sequence that supports an ephemeral variant, a persistent variant, and efficient conversions between them.

The Haskell library Yi Rope [5] implements persistent sequences as finger trees of fixed-capacity chunks. Because it does not have a strategy for compacting adjacent chunks during concatenation operations, its time and space efficiency can degenerate [2, §2].

5.3 Transience

The Clojure programming language [16], which aims to "simplify multi-threaded programming", places strong emphasis on the idea of transient data structures. Persistent data structures should be preferred by default; ephemeral data structures should be used only temporarily where required to improve performance.

Clojure v1.1, released in 2010, introduces transient vectors, hash maps, and hash sets. According to L'orange [24], its transient vectors consist of a tree of chunks that form the front of the sequence and a chunk that forms the tail of the sequence. Each chunk stores 32 items. These chunks are persistent, so every update requires a copy, except if the vector is transient, in which case updates can be performed in place.

This release of Clojure did not offer a transient sequence data structure. This observation motivated the development of transient relaxed radix-balanced (RRB) trees [25, 7, 1], a transient variant of RRB trees [4, 32], which originally are immutable. Transient RRB trees can be viewed as a variant of the transient K-way trees that we have presented (§2.4), where nodes have a flexible arity of up to 32. In contrast with RRB trees, in Sek, the elements near the start and end of the sequence are stored at the root of the tree. Moreover, our persistent chunks exploit *monotonicity* and *views* to decrease the cost of *push* and *pop* operations, respectively.

5.4 Iterators

An *iterator* is an abstract object that offers access to the elements of a collection, on demand, one at a time. An iterator allows iteration to be controlled (that is, suspended, resumed, aborted) by

the consumer. In contrast, higher-order iteration functions, which in functional programming languages are more common, give control to the producer. This makes them easier to implement but limits the range of their applications.

Li and Duck [23] argue in favor of offering *persistent iterators* on persistent collections. These iterators support updates. An update does not affect the underlying collection: instead, it simply produces a new iterator, which represents an updated sequence of elements. There is no danger of iterator invalidation.

In contrast, we offer mutable iterators on persistent sequences, which disallow updates, and mutable iterators on ephemeral sequences, which allow updates, but apply them directly to the underlying data structure. By combining an iterator on a persistent source sequence and an ephemeral destination sequence, we can implement Li and Duck's motivating example, a *filter* function on persistent sequences, with time complexity $O(n + K)$. We offer an optional iterator invalidation detection mechanism. Enabling it guarantees that iterator invalidation (a programming error) is detected at runtime, at a cost of $O(1)$ per iterator operation.

6 Conclusion

In summary, we have presented, analyzed, and evaluated an efficient transient data structure for sequences. We have implemented this data structure in OCaml. We believe that it is a good candidate for a one-size-fits-all, general-purpose sequence data structure.

More generally, we believe that transient data structures may deserve further study by the functional programming community, so that programmers need not choose between ephemeral data structures, which are potentially faster but more restricted, and persistent data structures, which are more convenient but slower.

In the future, we might wish to provide optimized implementations of insertion and deletion, which presently must be expressed in terms of splitting and concatenation.

It would also be desirable and interesting to make our persistent data structure safe for use in a concurrent setting, without compromising its performance in a sequential setting. For performance reasons, our ephemeral data structure would deliberately remain non-thread-safe: that is, two concurrent accesses to it would be forbidden unless both are read accesses. Under these conventions, we believe that we have only one potential source of data races, namely, competing monotonic updates on the support of a persistent chunk (§3.3). Thus we would need to either deactivate monotonic updates or implement them using atomic operations. We expect the optimal choice to depend on the chunk capacity K, the hardware, and the runtime system.

Finally, it would be worth devoting additional effort to the question of using our data structure to represent strings. At the moment, our data types are parameterized with the type `'a` of the elements of the sequences. The OCaml compilers do not perform monomorphization or specialization: every element is a *value* and occupies one word of memory. A value is either an immediate value (an integer; a floating-point number; a character; an element of an enumerated type) or a pointer to a heap-allocated block. If our data structure is used to represent strings as sequences of 8-bit characters, then a considerable amount of space is wasted, as each character occupies one word. At this time, we have not attempted to specialize our code for this use case, as we did not wish to manually specialize—and subsequently maintain—thousands of lines of code. This code transformation seems particularly tricky, and difficult to automate, because this data structure is non-uniform [28, §10.1]: whereas every node in the bottom-most layer contains an array of elements, every node in the next layer up contains an array of arrays of elements, and so on. Thus, only the bottom-most layer can be specialized to use an ad hoc type of arrays of characters, which in OCaml is named `bytes`. In the long term, we would like to implement a tool that automates this source-to-source transformation.

```
let rec check : 'a. 'a ptree -> 'a measure -> depth -> id -> unit =
fun s m depth id ->
  match s with
  | Zero _ | One _ -> ()
  | Short { data; _ } ->
      let n = Array.length data in
      assert (2 <= n && n <= threshold)
  | Level { weight; front; middle; back } ->
      assert (weight > 0);
      if Chunk.is_empty front || Chunk.is_empty back
        then assert (is_empty middle);     (* check invariant 1 *)
      Chunk.check m id front;              (* check the [front] chunk *)
      Chunk.check m id back;               (* check the [back] chunk *)
      check_middle middle m depth id;      (* recursive call *)
      assert (weight =                     (* validate the [weight] field *)
        Chunk.weight front +
        fold_left (fun sum p -> sum + Chunk.weight p) 0 middle +
        Chunk.weight back)

and check_middle : 'a. 'a chunk ptree -> 'a measure -> depth -> id -> unit =
fun middle m depth id ->
  check middle MSWeight (depth + 1) id;    (* recursive call *)
  middle |> iter Front (Chunk.check m id); (* check every element of [middle] *)
  middle |> iter Front (fun p ->           (* check invariant 2 *)
    assert (Chunk.length p > 0));
  let k = capacity depth in
  let previous_length = ref k in
  middle |> iter Front (fun p ->           (* check invariant 3 *)
    assert (!previous_length + Chunk.length p > k);
    previous_length := Chunk.length p)
```

Fig. 19. A runtime validation function for persistent sequences

Acknowledgments

The authors wish to thank Alexandre Moine and the anonymous reviewers for their comments and suggestions. Artificial intelligence was used to proof-read the paper and to help design and implement the *merge* function (§3.2). Under the guidance of our random testing framework, it proposed a correct design and implementation, which we then reviewed and improved.

A A Runtime Validation Function

Figure 19 shows a runtime validation function for our persistent sequence data structure. This function is part of our random testing infrastructure; it documents and tests the invariants of the data structure. It acts on the type `chunk` (Figure 12) and `'a ptree` (Figure 15). (The code has been slightly edited to match the notation of the paper.) A similar runtime validation function acts on the type `'a etree` (Figure 15), and invokes this function, as the middle sequence of an ephemeral sequence is a persistent sequence.

The parameter `m` in Figure 19 is redundant with the parameter `depth`: it is `MSUnit` at depth 0 and `MSWeight` at greater depths. This parameter determines how to compute the weight of an element

of type `'a`. At depth 0, an element has unit weight; at greater depths, an element is a chunk whose weight is stored in its `weight` field. The parameter `m` is used by the function `Chunk.check`, which (among other checks) verifies that the weight of a chunk is the sum of the weights of its elements. Although it is redundant with `depth`, this parameter cannot be removed: it serves to convince the OCaml type-checker that if `depth` is nonzero then reading the `weight` field of an element is safe.

B Proofs of Bounds

B.1 Bounds on Depth and Number of Chunks in Chunked Sequences

In the setting of ephemeral chunked sequences, Acar et al. establish a bound on the number of chunks in a middle sequence and on the depth of the tree [2, Appendix A, Lemma 1 and 2]. Our Lemmas B.1 to B.4 reproduce their results, with minor clarifications.

Recall that each level features a *middle sequence*, a sequence of chunks. Each of these chunks contains *items*, which at the top level are basic elements and at deeper levels are chunks.

Lemma B.1 (Bound on the number of chunks in a middle sequence). *If a middle sequence stores n items (where $n > 0$) then it contains at most $\lfloor \frac{n-1}{B} \rfloor + 1$ chunks, where $B = \frac{K+1}{2}$.*

Proof. Let p be the number of chunks. If $p \leq 1$, then the result is trivial. Otherwise $p \geq 2$. Let the sequence of values $(x_i)_{i \in 1..p}$ describe the sizes of the chunks in the middle sequence. The number of items stored in the chunks is $n = \sum_{i \in 1..p} x_i$. By invariant 2, the chunks in the middle sequence are nonempty, so $x_i \geq 1$ for all i. Moreover, by invariant 3, for every $i \in 1..(p-1)$, we have $x_i + x_{i+1} \geq K + 1$. Therefore:

$$\begin{array}{rcl} 2n & = & \left(\sum_{i \in 1..p} x_i\right) + \left(\sum_{i \in 1..p} x_i\right) \\ & = & \left(x_p + \sum_{i \in 1..(p-1)} x_i\right) + \left(x_1 + \sum_{i \in 2..p} x_i\right) \\ & = & (x_p + x_1) + \left(\sum_{i \in 1..(p-1)} (x_i + x_{i+1})\right) \\ & \geq & 2 + \sum_{i \in 1..(p-1)} (K+1) \\ & \geq & 2 + (p-1)(K+1) \end{array}$$

It follows that $p \leq \frac{2(n-1)}{K+1} + 1$. Since p is an integer, we deduce $p \leq \lfloor \frac{2(n-1)}{K+1} \rfloor + 1$. □

Definition B.2 (Lengths of the middle sequences, in depth). Let n_0 be n, the length of the sequence. For every $d \geq 1$, let n_d be the length of the middle sequence at depth d, that is, the number of *chunks* in the middle sequence. (If depth d does not exist, n_d is zero.) The top level is numbered 1, so n_1 is the length of the middle sequence of the top level.

Lemma B.3 (Bound on the lengths of the middle sequences). *The values n_d are bounded by:*

$$n_d \leq \frac{n-1}{B^d} + 1 \qquad \text{where} \qquad B = \frac{K+1}{2}$$

Proof. By induction on the depth. In the base case, where $d = 0$, we have $n = n_0$ and $\frac{n-1}{B^0} + 1 = n$. Hence the bound is verified.

In the inductive case, suppose $n_d \leq \frac{n-1}{B^d} + 1$, and let us establish the desired bound on n_{d+1}. Consider a level at depth d. If its middle sequence is empty, then $n_{d+1} = 0$ and the bound is trivially verified. Otherwise, because invariant 2 guarantees that the front and back chunk cannot be both empty, the middle sequence contains at most $n_d - 1$ items. By Lemma B.1, the length of this middle sequence is bounded as follows: $n_{d+1} \leq \frac{2(n_d-1)}{K+1} + 1$. (The floor operation is not needed.) This inequality is equivalent to $n_{d+1} - 1 \leq \frac{n_d-1}{B}$. By the induction hypothesis, we have $n_d - 1 \leq \frac{n-1}{B^d}$. Hence, we have $n_{d+1} - 1 \leq \frac{n-1}{B^{d+1}}$, which entails the desired bound. □

The depth of a tree is the number of its levels, that is, the number of `Level` data constructors in this tree. Here, we assume that the data constructor `One` (Figure 15) can be used, at an arbitrary

depth, to represent a sequence of one element. (In our OCaml implementation, this data constructor is used at the top level only.) If this is not the case then one additional level, whose middle sequence is empty, is needed, so one must add 1 to the bound offered by the following lemma.

Lemma B.4 (Bound on the depth, original presentation). *If a tree stores n elements and has depth D then $D \leq \lfloor \log_B n \rfloor + 1$ holds, where $B = \frac{K+1}{2}$.*

Proof. Let Δ stand for $\lfloor \log_B n \rfloor + 1$. The goal is to prove $D \leq \Delta$. We have $\log_B n < \lfloor \log_B n \rfloor + 1 = \Delta$. Hence, $n < B^\Delta$, and $\frac{n}{B^\Delta} < 1$.

Besides, recall that n_d denotes the length of the middle sequence at depth d, or zero if this level does not exist. By Lemma B.3, for every level d, we have $n_d \leq \frac{n-1}{B^d} + 1$. In particular, at level Δ, we have $n_\Delta \leq \frac{n-1}{B^\Delta} + 1$, which implies $n_\Delta \leq \frac{n}{B^\Delta} + 1$.

By exploiting the earlier result $\frac{n}{B^\Delta} < 1$, we derive $n_\Delta < 2$. Since n_Δ is an integer, $n_\Delta \leq 1$ follows. This means that the middle sequence at level Δ does not exist or is empty (in which case it is represented using the constructor `Empty`) or has length 1 (in which case it is represented using the constructor `One`). Thus, the last non-trivial level, represented using the constructor `Level`, lies at depth at most Δ. We conclude that the depth D of the tree cannot exceed Δ, that is, $D \leq \Delta$. □

First, let us reformulate this bound using logarithms in base K instead of base B. To simplify things, we assume $K > 2$ instead of just $K \geq 2$, and we apply a minor overapproximation by replacing $\log_{\frac{K+1}{2}} n$ with $\log_{\frac{K}{2}} n$.

Lemma B.5 (Bound on the depth, simplified statement). *Assume $K > 2$. A tree that stores n elements has depth at most $\lfloor (1+\epsilon) \cdot \log_K n \rfloor + 1$, where $\epsilon = \frac{1}{(\log_2 K) - 1}$.*

Proof. We have: $\log_B n = \log_{\frac{K+1}{2}} n \leq \log_{\frac{K}{2}} n = \frac{1}{\log_K (\frac{K}{2})} \cdot (\log_K n)$.
The leading factor can be rewritten as follows: $\frac{1}{\log_K (\frac{K}{2})} = \frac{\log_2 K}{\log_2 (\frac{K}{2})} = \frac{\log_2 K}{(\log_2 K) - 1} = 1 + \frac{1}{(\log_2 K) - 1}$. □

For example, if $K = 16 = 2^4$, the depth is bounded by $\lfloor \frac{4}{3} \cdot \log_{16} n \rfloor + 1$.

We now refine this bound in the case where every chunk stored in every middle sequence is full. (A chunk is *full* if its length is K.) If concatenation has not been used in the construction of the sequence then this property must be satisfied. The front and back chunks, at each level, may still be partially filled.

Lemma B.6 (Bounds when chunks in middle sequences are full). *If every chunk in every middle sequence is full, then all of the aforementioned bounds, where B is replaced with K, hold. In particular, the depth is at most $\lfloor \log_K n \rfloor + 1$.*

Proof. The factor B originates in Lemma B.1. Let us prove that if a middle sequence stores n items ($n > 0$) and if all its chunks are full then it is formed of at most $\lfloor \frac{n-1}{K} \rfloor + 1$ chunks. Each chunk stores K elements, thus n must be divisible by K and the number of chunks is equal to $\frac{n}{K}$. The claimed bound holds because, since n is divisible by K, we have: $\frac{n}{K} = (\frac{n}{K} - 1) + 1 = \lfloor \frac{n-1}{K} \rfloor + 1$. □

Finally, we adapt a lemma by Acar et al. [2, Appendix A] to bound the total number of chunks involved (at all levels) in the representation of a sequence.

Lemma B.7 (Bound on the number of chunks). *Let M denote the total number of chunks involved in the sequence. Then $M \leq \frac{n-1}{B-1} + 3D$ where D is the depth of the tree.*

Proof. At depth d, there is one front chunk, one back chunk, plus n_d chunks in the middle sequence. Hence the total number of chunks is $M = \sum_{d=1}^{D} (n_d + 2)$. By Lemma B.3, $n_d \leq \frac{n-1}{B^d} + 1$,

with $B = \frac{K+1}{2}$. Hence, $M \leq \sum_{d=1}^{D}(\frac{n-1}{B^d} + 1 + 2) = (n-1) \cdot \left(\sum_{d=1}^{D} \frac{1}{B^d}\right) + 3D$. We exploit the well-known bound on geometric series: $\sum_{d=1}^{D} \frac{1}{B^d} \leq \frac{1}{B-1}$. Therefore, $M \leq \frac{n-1}{B-1} + 3D$. □

Under the assumption $K > 2$, let us reformulate and simplify this bound by eliminating B and D.

Lemma B.8 (Bound on the number of chunks, simplified). *Assume $K > 2$. Let M be the total number of chunks involved in the sequence. We have: $M \leq \frac{2n}{K-1} + 3 \cdot \left(1 + \epsilon\right) \cdot (\log_K n) + 3$ where $\epsilon = \frac{1}{(\log_2 K) - 1}$.*

Proof. We start from $M \leq \frac{n-1}{B-1} + 3D$, which is given by Lemma B.7. We substitute $B = \frac{K+1}{2}$, and exploit $D \leq \lfloor (1+\epsilon) \cdot \log_K n \rfloor + 1$ (Lemma B.5). We obtain $M \leq 2 \cdot \frac{(n-1)}{K-1} + 3\big((1+\epsilon) \cdot \log_K n + 1\big)$, from which the desired bound follows. □

Lemma B.9 (Bound on the number of chunks, when chunks in middle sequences are full). *Let M be the total number of chunks involved in the sequence. If all chunks in the middle sequences are full, then $M \leq \frac{n}{K-1} + 3\lfloor \log_K n \rfloor + 3$.*

Proof. We start from $M \leq \frac{n-1}{B-1} + 3D$, which is given by Lemma B.7. This time, we exploit Lemma B.6, substitute $B = K$, and exploit $D \leq \lfloor \log_K n \rfloor + 1$. We get $M \leq \frac{n}{K-1} + 3\lfloor \log_K n \rfloor + 3$. □

B.2 Space Bounds for Chunked Sequences

Lemma B.10 (space bound for persistent variant of transient sequences). *The space usage is at most $(2 + \frac{22}{K-1})n + O(K \log_K n)$ words in OCaml, and $(2 + \frac{12}{K-1})n + O(K \log_K n)$ words in C.*

Proof. We must measure the space usage of the *spine* plus the space usage of the chunks. The spine consists of the memory blocks that correspond to the data constructor `Level`, plus possibly (at the bottom) a memory block that corresponds to the data constructor `One`.

In OCaml, a `Level` block takes up 5 words and a `One` block takes up 3 words. Hence, the spine occupies $5D + 3$ words, which is $O(\log_K n)$.

In OCaml, a chunk requires $K + 10$ words and, by Lemma B.8, the number of chunks is bounded by $\frac{2n}{K-1} + O(\log_K n)$. Hence, the total space usage is $2(1 + \frac{11}{K-1})n + O(K \log_K n)$.

In C, a chunk requires $K+5$ words. Hence, the leading term for the space usage is: $2(1+\frac{6}{K-1})n$. □

Lemma B.11 (space bound, in case chunks in the middle sequences are full). *If all chunks in the middle sequences are full, then the space usage is at most $(1 + \frac{11}{K-1})n + O(K \log_K n)$ words in OCaml and $(1 + \frac{6}{K-1})n + O(K \log_K n)$ words in C.*

Proof. Compared with the previous lemma, the space occupied by the spine does not change. By Lemma B.9, the number of chunks is bounded by: $\frac{n}{K-1} + O(\log_K n)$.

In OCaml, a chunk requires $K+10$ words, hence the leading term in the space usage is: $(1+\frac{11}{K-1})n$.

In C, a chunk requires $K + 5$ words, hence the leading term in the space usage is: $(1 + \frac{6}{K-1})n$. □

For the ephemeral variant of transient sequences, only the layout of the first layer differs; the asymptotics for space usage are exactly the same.

B.3 Cost of Push and Pop on Ephemeral Sequences

In this subsection, we prove that *push* and *pop* have amortized time complexity $O(\log_K N)$, as stated in §3.6. This is an original result.

We first bound the cost of a *push* into the middle sequence. A *push-front* operation on a persistent sequence can trigger either a *push* operation into the front chunk, which costs $O(K)$, or the allocation of a fresh chunk plus a *push* operation at the next level, which also costs $O(K)$. There are at most

$\log_K N$ levels. Let C be a constant such that $CK \log_K N$ bounds the cost of a *push* or a *pop* operation on a persistent sequence. Let C' be a constant such that $C'K$ bounds the cost of allocating a fresh chunk or copying K elements into an empty chunk. Let C'' stand for $C \log_K N + C'$.

The advertised cost of each *push* or *pop* operation is $C'' + O(1)$. This amount is $O(\log_K N)$, which for all practical purposes is a small integer (e.g., 7), hence "effectively constant time". Thereafter, let n_c stand for the size of a support c.

We define the potential of an ephemeral tree structure as the sum of the *front-potential* and the *back-potential*. The front-potential is defined as $\big(2 \cdot |K - n_{\text{front}} - n_{\text{ifront}}| + (K - n_{\text{middle-fst}})\big) \cdot C''$, where $n_{\text{middle-fst}}$ is the size of the first chunk in the middle sequence, or K in case the middle sequence is empty. The back-potential is defined symmetrically. When a persistent sequence is converted to an ephemeral sequence, its potential must be initialized: this requires *edit* to advertise a cost of $O(K \log_K n)$. We now focus on the front.

- If a *push-front* or *pop-front* operation acts only on `front` and `ifront`, then it costs $O(1)$. The quantity $n_{\text{front}} + n_{\text{ifront}}$ varies by at most one unit. Hence, the potential increases by at most C''. The direct cost and the increase in potential are bounded by $C'' + O(1)$.
- Consider a *push-front* operation in a case where both `front` and `ifront` are full. The chunk `ifront` is pushed into the middle sequence for a cost of at most $CK \log_K N$. The chunk `front` is assigned to `ifront`. The `front` is assigned to a fresh support, allocated for a cost of $C'K$. The direct cost is thus $CK \log_K N + C'K + O(1) = KC'' + O(1)$.
  Before the *push* operation, $n_{\text{front}} = n_{\text{ifront}} = K$, hence the front-potential is at least $2K \cdot C''$. After this operation, $n'_{\text{front}} = 1$ and $n'_{\text{ifront}} = K$ and $n'_{\text{middle-fst}} = K$, hence the front-potential is $2C''$. The advertised cost $C'' + O(1)$ plus the decrease in potential $(2K - 1)C''$ cover the direct cost.
  In case the *push-front* operation causes the middle sequence to become nonempty, the back-potential subexpression may seem impacted. Yet, in fact, $K - n_{\text{middle-last}}$ remains zero, because the chunk that is pushed is full.
- Consider a *pop-front* operation in a case where both `front` and `ifront` are empty. If the middle sequence is empty, the `front` chunk is swapped with either `iback` or `back`, which costs $O(1)$. Otherwise, a *pop* operation on the middle sequence is performed, which costs at most $CK \log_K N$. If the extracted chunk is uniquely owned, its support can be assigned to the `front` field in time $O(1)$. Otherwise, its elements must be copied into the `front` chunk, which costs at most $C'K$. The direct cost is thus $CK \log_K N + C'K$, which is $KC'' + O(1)$.
  Before the *pop* operation, $n_{\text{front}} = n_{\text{ifront}} = 0$, so the front-potential is $\big(2K + (K - n_{\text{middle-fst}})\big) \cdot C''$. After this operation, $n'_{\text{front}} = n_{\text{middle-fst}}$ and $n'_{\text{ifront}} = 0$, hence the potential is $\big(2(K - n_{\text{middle-fst}}) + (K - n'_{\text{middle-fst}})\big) \cdot C''$. The decrease in potential is $(n_{\text{middle-fst}} + n'_{\text{middle-fst}}) \cdot C''$. By invariant 3, we have $n_{\text{middle-fst}} + n'_{\text{middle-fst}} > K$. The advertised cost $C'' + O(1)$ plus the decrease in potential (which exceeds KC'') cover the direct cost.

References

[1] Araz Abishov. 2020. *A Vector Implementation Based On RRB-Tree for Rust.* Master's thesis. Institute of Informatics, Faculty of Mathematics and Natural Sciences, University of Oslo.

[2] Umut A. Acar, Arthur Charguéraud, and Mike Rainey. 2014. Theory and Practice of Chunked Sequences. In *Algorithms (ESA)*, Vol. 8737. Springer, 25–36.

[3] Clément Allain, Basile Clément, Alexandre Moine, and Gabriel Scherer. 2024. Snapshottable Stores. *Proceedings of the ACM on Programming Languages* 8, ICFP, Article 248 (Aug. 2024).

[4] Philip Bagwell and Tiark Rompf. 2011. RRB-Trees: Efficient Immutable Vectors. (2011). Unpublished.

[5] Jean-Philippe Bernardy, Gustav Munkby, and Mateusz Kowalczyk. 2018. `yi-rope`.

[6] Hans-J. Boehm, Russ Atkinson, and Michael Plass. 1995. Ropes: An alternative to strings. *Software: Practice and Experience* 25, 12 (1995), 1315–1330.

[7] Juan Pedro Bolívar Puente. 2017. Persistence for the masses: RRB-vectors in a systems language. *Proceedings of the ACM on Programming Languages* 1, ICFP (2017), 16:1–16:28.
[8] Adam Louis Buchsbaum. 1993. *Data-Structural Bootstrapping and Catenable Deques*. Ph. D. Dissertation. Princeton University.
[9] Arthur Charguéraud and François Pottier. 2026. *Sek*.
[10] Arthur Charguéraud. 2024. *The CFML tool and library*.
[11] Basile Clément, Camille Noûs, and Gabriel Scherer. 2025. Storable types: free, absorbing, custom. In *Journées Françaises des Langages Applicatifs (JFLA)*.
[12] The Batteries Included Community. 2026. *Batteries Included*.
[13] Laxman Dhulipala, Guy E. Blelloch, and Julian Shun. 2019. Low-latency graph streaming using compressed purely-functional trees. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Phoenix, AZ, USA) *(PLDI 2019)*. Association for Computing Machinery, New York, NY, USA, 918–934.
[14] James R. Driscoll, Neil Sarnak, Daniel Dominic Sleator, and Robert Endre Tarjan. 1989. Making Data Structures Persistent. *J. Comput. System Sci.* 38, 1 (1989), 86–124.
[15] The Free Software Foundation. 2026. `stl_deque.h`.
[16] Rich Hickey. 2006. *The Clojure programming language*.
[17] Ralf Hinze and Ross Paterson. 2006. Finger trees: a simple general-purpose data structure. *Journal of Functional Programming* 16, 2 (2006), 197–217.
[18] Haim Kaplan, Chris Okasaki, and Robert E. Tarjan. 2000. Simple Confluently Persistent Catenable Lists. *SIAM J. Comput.* 30, 3 (2000), 965–977.
[19] Haim Kaplan and Robert Endre Tarjan. 1995. Persistent lists with catenation via recursive slow-down. In *Symposium on the Theory of Computing*. 93–102.
[20] Haim Kaplan and Robert E. Tarjan. 1996. Purely Functional Representations of Catenable Sorted Lists. In *Symposium on the Theory of Computing*. 202–211.
[21] Haim Kaplan and Robert E. Tarjan. 1999. Purely functional, real-time deques with catenation. *Journal of the ACM* 46, 5 (1999), 577–603.
[22] Patrik Keller. 2022. *Pvec*.
[23] Yihe Li and Gregory J. Duck. 2026. Persistent Iterators with Value Semantics. *Proceedings of the ACM on Programming Languages* 10, PLDI (June 2026).
[24] Jean Niklas L'orange. 2013. Understanding Clojure's Persistent Vectors. Blog post.
[25] Jean Niklas L'orange. 2014. *Improving RRB-Tree Performance through Transience*. Master's thesis. Department of Computer and Information Science, Norwegian University of Science and Technology.
[26] Alexandre Moine, Arthur Charguéraud, and François Pottier. 2022. Specification and Verification of a Transient Stack. In *Certified Programs and Proofs (CPP)*. 82–99.
[27] Chris Okasaki. 1997. Catenable Double-Ended Queues. In *International Conference on Functional Programming (ICFP)*. 66–74.
[28] Chris Okasaki. 1999. *Purely Functional Data Structures*. Cambridge University Press.
[29] Juliette Ponsonnet and François Pottier. 2026. Verified Persistent Catenable Deques. In *Journées Françaises des Langages Applicatifs (JFLA)*.
[30] François Pottier. 2021. Strong Automated Testing of OCaml Libraries. In *Journées Françaises des Langages Applicatifs (JFLA)*.
[31] Matthieu Sozeau. 2007. Program-ing Finger Trees in Coq. In *International Conference on Functional Programming (ICFP)*. 13–24.
[32] Nicolas Stucki, Tiark Rompf, Vlad Ureche, and Phil Bagwell. 2015. RRB vector: a practical general purpose immutable sequence. In *International Conference on Functional Programming (ICFP)*. 342–354.
[33] Jules Viennot, Arthur Wendling, Armaël Guéneau, and François Pottier. 2026. Verified Purely Functional Catenable Real-Time Deques. (Jan. 2026). Unpublished.
[34] Michal Zalewski. 2020. *American Fuzzy Lop*.

Received 2026-02-22; accepted 2026-05-13