Persistent Iterators with Value Semantics

YIHE LI, National University of Singapore, Singapore
GREGORY J. DUCK, National University of Singapore, Singapore

Iterators are a fundamental programming abstraction for traversing and modifying elements in containers in mainstream imperative languages such as C++. Iterators provide a uniform access mechanism that hides low-level implementation details of the underlying data structure. However, iterators over *mutable* containers suffer from well-known hazards including invalidation, aliasing, data races, and subtle side effects. *Immutable* data structures, as used in functional programming languages, avoid the pitfalls of mutation but rely on a very different programming model based on recursion and higher-order combinators (`map`, `foldl`, `traverse`, etc.) rather than iteration. However, these combinators are not always well-suited to expressing certain algorithms, and recursion can expose implementation details of the underlying data structure.

In this paper, we propose *persistent iterators*—a new abstraction that reconciles the familiar iterator-based programming style of imperative languages with the semantics of persistent data structures. A persistent iterator snapshots the version of its underlying container at creation, ensuring safety against invalidation and aliasing. Iterator operations (`++`, `erase`, etc.) operate on the iterator-local copy of the container, giving true value semantics: variables can be rebound to new persistent values while previous versions remain accessible. We implement our approach in the form of LibFpp—a C++ container library providing persistent vectors, maps, sets, strings, and other abstractions as persistent counterparts to the *Standard Template Library* (STL). Our evaluation shows that LibFpp retains the expressiveness of iterator-based programming, eliminates iterator-invalidation, and achieves asymptotic complexities comparable to STL implementations—albeit with the higher constant-factor overheads of persistence. Our design targets use cases where persistence and safety are desired, while allowing developers to retain familiar iterator-based programming patterns.

CCS Concepts: • **Software and its engineering → General programming languages**.

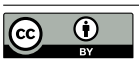
Additional Key Words and Phrases: Persistent data-structures, value semantics, iterators, containers.

ACM Reference Format:
Yihe Li and Gregory J. Duck. 2026. Persistent Iterators with Value Semantics. *Proc. ACM Program. Lang.* 10, PLDI, Article 246 (June 2026), 22 pages. https://doi.org/10.1145/3808324

1 Introduction

Iterators are one of the most ubiquitous and useful abstractions among programming languages and form the backbone of the standard libraries for popular languages like C++ [1], Python [15], and Rust [16]. Acting as *generalized pointers* into structured data, iterators provide a uniform and representation-independent interface for accessing and manipulating container elements. This uniformity makes iterators general in principle: traversal logic can be written once and reused across a wide range of data structures and algorithmic contexts. Such uniformity enables iterators to become the bases of generic algorithms such as sorting and searching, eventually leading to the birth of powerful and generic libraries like the *Standard Template Library* (STL) of C++ [1, 19] and Java's *Collections Framework* [13]. However, when coupled with *mutation*, iterators can be the source of a well-known class of subtle bugs in programs [1, 8]. When the underlying structure

Authors' Contact Information: Yihe Li, National University of Singapore, Singapore, Singapore, yihe.li@u.nus.edu; Gregory J. Duck, National University of Singapore, Singapore, Singapore, gregory@comp.nus.edu.sg.



ACM 2475-1421/2026/6-ART246
https://doi.org/10.1145/3808324

is modified, iterators may become *invalidated*, possibly leading to vulnerabilities (C++), runtime exceptions (Java/C#), or undefined/unpredictable behavior (Python). For example, the C++ STL treats iterator invalidation as an example of *undefined behavior* (UB), often leading to *use-after-free* vulnerabilities, and each modifying operation tends to have its own specific invalidation rules. Furthermore, if the container is modified concurrently through an active iterator, consistency issues may arise without careful synchronization of observers [11].

Other programming languages do not typically expose an iterator abstraction. For example, *functional programming* (FP) languages, such as Haskell, use *immutable*/*persistent* data structures that cannot be mutated after construction [2]. Under this setting, modifying operations can only create *new* versions of existing data structures without destroying the old copy. For example, Haskell's `Data.Map.insert` creates a new associative map without destroying the original, meaning that *both* the new and original map values can be used after the operation. Note that updates to persistent data structures typically do *not* require a deep copy of the entire data structure. Rather, only the specific nodes that require updating are copied, and the rest of the structure is *shared* with the original—a.k.a. *structural sharing*, which is essential for an efficient (both time and memory) implementation of persistence. The persistent programming model preempts the invalidation-via-mutation problem: each observer operates on their own local copy of the data structure without interference from concurrent updates. Prior work demonstrates that persistent data structures can achieve favorable asymptotic efficiency and practical performance for many workloads [12], albeit with nontrivial constant-factor overheads compared to their imperative counterparts.

Yet, existing persistent data structure implementations tend not to expose a general iterator abstraction. One key problem is the fundamental difference between *immutable* data structures and *mutable* references used by iterators, which are often viewed as incompatible programming paradigms. Instead, Haskell's `Data.Map` exposes a collection of higher-order *combinators* (`map`, `filter`, `foldl`, `traverse`, etc.) for data-structure traversal and modification. Such combinators capture many common traversal/transformation patterns, and can be *composed*, allowing for the construction of complex algorithms. However, while combinators capture many traversal patterns, they are fixed in arity and structure, and thus may be less natural or less idiomatic for expressing algorithms requiring stateful or early-exit iteration. For Haskell lists, one valid option is to fall back to using *recursion* instead of combinators. However, this option is rarely exposed by existing *opaque* data structure APIs, such as `Data.Map`, primarily for performance reasons. In contrast, C++-style iterators are both highly expressive and abstract.

In this paper, we bridge the gap between the two seemingly incompatible programming paradigms in the form of *persistent iterators*—an iterator-centric abstraction combined with the safety of persistent data structures. Unlike read-only *const iterators*, which deny updates via iterators altogether, persistent iterators still support updates by working over a *local copy* of the underlying data structure rather than the original via a mutable reference. This means that any update via the iterator (e.g., assignment, insertion, erasure) does not invalidate (or even modify) the original container, nor any other local copy from any other iterator. Essentially, persistent iterators support STL-style iterator operations—including updates via non-const iterators—while maintaining immutability guarantees. We show that persistent iterators retain much of the expressive power of traditional mutable iterators, as well as providing a uniform abstraction that does not expose implementation details of the underlying structure. Furthermore, persistent iterators are immune to iterator invalidation—thereby eliminating a common class of programming bug.

We have implemented and evaluated our approach in the form of LibFpp—a C++ library that provides concrete implementations of common STL-style containers, such as *vectors*, *strings*, *sets*, and *maps*, in a persistent manner. LibFpp is designed to resemble the C++ STL in terms of functionality, algorithmic complexity, and API—where the main differences relate only to persistence and value

```cpp
string filterAscii(string s) {
  string::iterator i;
  for (i = s.begin(); i != s.end(); )
    if (*i >= 0x7F) i.erase();
    else            ++i;
  return i.value();
}
```

(a) C++ LibFpp `string`

```cpp
void filterAscii(std::list<char32_t> &s) {
  for (auto i = s.begin(); i != s.end(); )
    if (*i >= 0x7F) i = s.erase(i);
    else            ++i;
}
```

(b) C++ STL `std::list<char>`

```cpp
void filterAscii(std::string &s) {
  std::string t;
  for (auto c: s) if (c < 0x7F) t += c;
  s.swap(t);
}
```

(c) C++ STL `std::string`

```haskell
filterAscii :: String -> String
filterAscii = filter (\c -> ord c < 0x7F)
```

(d) Haskell `String` (combinator)

```haskell
filterAscii :: String -> String
filterAscii [] = []
filterAscii (c:cs)
  | ord c < 0x7F = c : filterAscii cs
  | otherwise    = filterAscii cs
```

(e) Haskell `String` (recursion)

```haskell
filterAscii :: Seq Char -> Seq Char
filterAscii s = go 0 s
  where
    go i s
      | i >= Seq.length s = s
      | otherwise =
          let c = Seq.index s i
          in if ord c >= 0x7F
             then go i (Seq.deleteAt i s)
             else go (i + 1) s
```

(f) Haskell (`Seq Char`)

Fig. 1. Comparing a simple "*filter non-ASCII characters from a string*" function across paradigms using C++ STL, C++ LibFpp, and Haskell. Version (a) is the idiomatic implementation using LibFpp persistent iterators. Versions (b) and (c) are idiomatic implementations for `std::list<char>` and `std::string`, respectively. Finally, we consider (d)/(e) as the idiomatic combinator/recursive implementation over Haskell `String`s, and (f) over Haskell (`Seq Char`) from `Data.Sequence` using explicit indices.

semantics—reducing the learning curve for developers already familiar with STL-style programming. LibFpp containers are built on top of *finger trees* [5]—a persistent data structure that can be used to implement common STL container operations with comparable algorithmic complexity. Persistent iterators are built on top of optimized *zippers* [6]—a persistent data structure for representing a "pointer" or "cursor" into another persistent data structure. We show that zippers can be used to implement a rich STL-like iterator API, again with comparable algorithmic complexity. We also show that, with sufficient optimization, LibFpp achieves the algorithmic and safety benefits of persistent containers and value semantics with practical efficiency consistent with the inherent constant factor overheads of persistence.

1.1 Motivating Example

As a motivating example, we consider a simple `filterAscii` function, which filters out all non-ASCII characters from a string. Several different implementations over different string representations are shown in Figure 1. A summary of the differences between different implementations is shown in Table 1. Our core argument is that, despite `filterAscii` being a relatively trivial function, the STL C++ and Haskell implementations do not simultaneously achieve all of the following properties:

- optimal $O(N)$ algorithmic *complexity*;
- structural *sharing* (i.e., will the result share unmodified structure with the input?);
- *persistence* (i.e., is the data structure *not* mutated in-place?); and
- *abstraction* (i.e., are implementation details of the container exposed?).

Optimal *complexity* is desirable for performance, structural *sharing* to minimize memory usage, *persistence* for algorithmic and safety benefits, and *abstraction* for modularity and elegance.

Table 1. Comparison of `filterAscii` implementations across different languages and paradigms. We compare the worst-case algorithmic *complexity*, whether the result *shares* structure with the input, whether the underlying data structure is *persistent*, whether the underlying data structure is *abstracted*, and whether the implementation is considered *idiomatic*. Here, (†) represents a version of Figure 1 (b) using `std::string` (with $O(N)$ erasure), (‡) represents a version of Figure 1 (c) using Immer [14] (necessary since Immer only exposes const-iterators), and (~) indicates the data structure is partly revealed by the algorithmic complexity (leaky abstraction). Only the LibFpp implementation with persistent iterators achieves sharing, persistence, and abstraction with optimal $O(N)$ algorithmic complexity.

Version	Language	API	Complexity	Sharing?	Persistent?	Abstract?	Idiomatic?
Figure 1 (b)	C++	STL `std::list<char>`	$O(N)$	✓	✗	~	✗
Figure 1 (b)†	C++	STL `std::string`	$O(N^2)$	✗	✗	~	✗
Figure 1 (c)	C++	STL `std::string`	$O(N)$	✗	✗	~	✓
Figure 1 (c)‡	C++	`immer::vector<char>`	$O(N)$	✗	✓	✓	✓
Figure 1 (d)	Haskell	`String`	$O(N)$	✗	✓	✓	✓
Figure 1 (e)	Haskell	`String` (recursion)	$O(N)$	✗	✓	✗	✓
Figure 1 (f)	Haskell	`Seq Char`	$O(N.\log N)$	✓	✓	✓	✗
Figure 1 (a)	C++	LibFpp `string`	$O(N)$	✓	✓	✓	✓

None of Figure 1 (b)-(f) achieves all four simultaneously. For example, the optimal C++ algorithm depends on the underlying string structure, with in-place erasure (Figure 1 (b)) for node-based containers `std::list<char>` and *accumulation* (Figure 1 (c)) for array-based containers `std::string`. This also bypasses the abstraction layer, as correct algorithm choice depends on container implementation internals (abstraction leakage). The Haskell implementations also have limitations. For example, the idiomatic combinator `filter` implementation (Figure 1 (d)) is compact and elegant, but it will construct a *deep copy* of the input with no structural sharing.[1] This deep copy will occur even if the input contains no non-ASCII characters. The recursive version (Figure 1 (e)) also suffers from the same problem, as well as exposing the data structure implementation (i.e., a list of `Char`). The final implementation (Figure 1 (f)) uses `Seq` from `Data.Sequence`—a persistent tree-based sequence implementation, possibly allowing for structural sharing. However, the sequence API lacks an iterator abstraction, and Figure 1 (f) circumvents this limitation by using explicit integer *indices*. However, aside from the additional verbosity, deletion-by-index is an $O(\log N)$ operation, and thus the overall algorithm is a sub-optimal $O(N.\log N)$. Finally, we note that existing C++ persistent data structure libraries only expose read-only *const* iterators, and thus does not provide the iterator interface necessary to support the in-place updates required by Figure 1 (b).

In contrast, we consider the LibFpp version of `filterAscii` implemented using *persistent iterators* in Figure 1 (a). This version uses familiar C++-style iterator programming (similar to Figure 1 (b)), and achieves all four properties outlined above, including: the algorithmic/safety benefits of persistent data structures, an optimal $O(N)$ complexity, and abstracts the underlying container representation. The main difference is that the iterator `i` implements *value* rather than reference semantics, meaning that the operation `i.erase()` erases the current character element independently of the originating container `s`. Furthermore, due to the lack of mutable reference semantics, the iterator-local copy must be explicitly extracted `i.value()`. The original container `s` and all copies are *unmodified* by the loop.

[1]For simplicity, we analyze Haskell under *strict* semantics. The time/space complexities for Haskell under *lazy* semantics are equivalent to those of *strict* semantics in the worst case.

1.2 Contributions

The `filterAscii` program is just one example, but linear scans of containers/data structures are a common programming pattern. Yet existing programming paradigms either lack persistence (imperative) or are either sub-optimal in algorithmic or memory complexity (functional). Persistent iterators combine key aspects of both approaches: iterators as a simple abstraction, persistence, and optimal time ($O(N)$ linear scan) and memory (structural sharing). Persistent iterators thus retain the programming idiom of C++ iteration, but not the reference semantics of the STL. Developers can write new code (or adapt existing algorithms) under a persistent, value-based paradigm while keeping a familiar iterator-style API. In summary, the main contributions of this paper are:

- We introduce the concept of *persistent iterators*—bridging the gap between traditional C++ iterator-based programming with the safety and power of persistent data structures. Unlike traditional mutable iterators that implement *reference semantics*, persistent iterators implement *value semantics* where each iterator object works on its own local copy of the underlying data structure—eliminating the risk of iterator invalidation bugs. And unlike const-iterators, persistent iterators allow for the (local copy) of the underlying data structure to be updated (assignment, insertion, erasure)—supporting a more traditional C++ iterator programming paradigm where containers can be modified via iterators.
- We show that persistent versions of common containers (*vectors*, *sets*, *maps*, *strings*, etc.) can be implemented using a single underlying data structure—the *finger tree* [5]—achieving a comparable algorithmic complexity with the C++ *Standard Template Library* (STL) operations. We show that persistent iterators can be implemented as optimized *zippers* [6] into persistent finger trees, and can be used to implement all of the standard C++ STL iterator operations (increment, decrement, dereference, etc.)—again with comparable algorithmic complexity. Our implementation is *uniform*—i.e., all containers use the same underlying data structure, preventing asymptotic surprises or the need for container-specific special cases (see Figure 1 (b) and (c)). To our knowledge, we are also the first implementation of zippers over finger trees.
- We implement our approach in the form of LibFpp—a C++ container library that supports persistent versions of common containers. We evaluate LibFpp against various popular C++ libraries, including the STL [19], Immer [14], Abseil [21], and Folly [10]. We show that LibFpp improves the safety in the usage of the containers with performance comparable to other persistent libraries.

This paper's core contribution is a semantic reconciliation: we show that iterator-based imperative programming can be reconciled with full value semantics over persistent data structures, without sacrificing expressiveness.

Non-goals. LibFpp is not intended as a drop-in replacement for STL containers in performance-critical code. Persistent data structures incur inherent constant-factor overheads and cannot match contiguous array-based implementations. Our goal is instead to provide an STL-like alternative for persistence-oriented use cases, where safety and value semantics are primary.

2 Background

2.1 Iterators

The *iterator abstraction* [4, 19] is widely used to provide a uniform way to access an element of a data structure in a sequential manner. Each *iterator* refers to a concrete element inside the structure (or a one-past-the-end special case), and iterators provide uniform operations that can either traverse or modify the parent collection. Modern *Standard Template Library* (STL) iterators can be advanced or receded to refer to different elements, and support element retrieval, replacement, or erasure at the specified position. Iterators are highly expressive: combinations of these simple operations can

```
void filterAscii(std::list<char32_t> &s) {
  for (auto i = s.begin(); i != s.end(); )
    if (*i >= 0x7F) s.erase(i);  // BUG!
    else            ++i;
}
```

Fig. 2. Example of a classic *iterator invalidation* bug variant of Figure 1 (b). Here, the result of `s.erase(i)` has not been reassigned back to `i`.

be used to implement many common algorithms. Furthermore, iterators are also abstract, allowing for element traversal, retrieval, and modification in a uniform way that does not directly expose the underlying container representation.

STL Iterators in C++. Iterators form the backbone of containers and algorithms in the C++ STL. Most STL algorithms express their inputs and outputs exclusively in terms of iterators or iterator pairs, thus enabling their reuse against different container types. To respect this abstraction, each standard container also provides `begin()` and `end()` member functions that return iterators referring to the first and one-past-the-end element, respectively. In C++, iterators themselves are modeled as an abstraction of a C-style pointer, thus having similar usage syntax. For example, `++it` advances the iterator to the next element in the container (conceptually similar to *pointer arithmetic*), and `*it` fetches the current element referred to by the iterator (conceptually similar to *pointer dereference*). Furthermore, C++ is among the few languages that split iterator advancement, the testing of validity, and dereferencing into three separate operations. While this model provides maximum flexibility, it also increases the complexity of iterator implementations.

STL Iterators themselves are typically implemented as lightweight copy-constructable objects, only containing a pointer that directly references a specific element in the originating container. Thus, iterators effectively implement a form of *reference semantics*, where the originating container can be accessed or even directly mutated via an iterator. However, the complexity of the API and reference semantics can lead to subtle errors when programming with iterators.

2.2 Limitations of STL Iterators

While iterators are a powerful abstraction, they also have some well-known limitations that are sometimes hard to mitigate. We present below several common misuses of STL iterators.

Iterator Invalidation. As C++ iterators are designed as a conceptual extension of the pointer model, they also suffer from the same pitfalls. When the underlying memory location managed by the parent container is mutated or de-/re-allocated, the iterators pointing to this location may be *invalidated*. In a low-level programming language such as C++, iterator invalidation can be a source of vulnerability, such as *use-after-free* (UaF) errors. A common example of iterator invalidation is illustrated in Figure 2. Here, the `s.erase(i)` operation will invalidate iterator `i`, and the correct idiom is to reassign the result of the erasure (see Figure 1 (b)).

A related problem is the possibility of iteration occurring concurrently with modifications to the container itself (*concurrent modification*). For example, if elements are added to a container during iteration, the previously obtained iterator may or may not traverse the newly added elements, depending on the underlying semantics of the container. Each container and operation has its own iterator invalidation rules. For example, the rules for `vector::shrink_to_fit` and `vector::erase` are very different: with the former (almost) always invalidating all iterators (unless no change in capacity), and the latter invalidating iterators only *after* the erased element. Many languages, such as Java and C#, attempt to detect invalidation and concurrent modification *dynamically*, and throw an error in a *fail-fast* manner (e.g., Java's `ConcurrentModificationException` [13]). Rust detects potential invalidation at *compile-time* using static borrow and lifetime checking. Python is memory safe by design, but may exhibit unintuitive behaviors if a container is modified during iteration.

Data Structure Invariants. In low-level languages such as C++, mutation via iterators can also lead to subtle issues, since elements may be modified in place. While standard containers restrict certain modifications to preserve invariants (e.g., keys in associative containers), indirect mutation through shared or aliased state can still violate logical invariants in practice. As a recent example, modification through `filter_view`'s iterator can cause elements satisfying the given predicate to be skipped [7]. The lack of deep-constness in the language allows mutation through ostensibly immutable references [3].

Uniformity and Leaky Abstractions. Although iterators provide a uniform abstraction in principle, container implementation details can still leak via asymptotic complexities. For example, the implementations of Figure 1 (b) and (c) are very different due to differences in the underlying container implementation. While porting Figure 1 (b) to `std::string` is possible, the erasure operation becomes $O(N)$, making the overall algorithm $O(N^2)$ in the worst case.

2.3 Persistent Data Structures

In conventional imperative programming, containers (e.g., C++ STL) are *mutable*: operations like `insert`, `erase`, or assignment modify the container in place. Mutation is efficient and familiar, but it tightly couples all references to a single, ever-changing state. Any update can invalidate iterators, aliases, or cached computations, and reasoning about program behavior requires tracking these side effects across potentially complex aliasing relationships.

Persistent data structures take a fundamentally different approach. A data structure is *persistent* if every update produces a new version of the structure while leaving the original version intact. The update does not overwrite (mutate) or destroy existing nodes; rather, it allocates *new* nodes only along the path that the update changes, and all unmodified portions of the structure are *structurally shared* between the old and new versions. For example, inserting an element into a persistent balanced tree allocates new nodes only along the search path, while all subtrees not affected by the insertion will be *shared*. The original and updated versions can coexist safely—providing logical immutability without "deep" copying of the entire structure. Persistent data structures, therefore, offer a number of semantic and practical advantages. Firstly, they eliminate side effects: an update cannot modify any existing version, removing an entire class of aliasing bugs. Secondly, "copying" a persistent container becomes a cheap $O(1)$ operation, making *value semantics* practical and efficient: assigning a container simply refers to the same underlying structure until one of the copies is modified. Because every version is logically independent, persistent containers are inherently thread-safe for concurrent reads and allow fine-grained rollback, undo, and snapshot semantics.

Memory Management. Because each node may be pointed to by more than one parent, memory *reclamation* must account for this sharing. *Garbage collection* (GC) naturally supports persistence by reclaiming unreachable nodes automatically. In manual-memory settings such as C++, *reference counting* provides a practical alternative: since our persistent data structures are acyclic by construction (nodes are immutable and references flow only from parents to children), simple reference counting suffices to reclaim storage deterministically.

Limitations of Persistent Data Structures. The benefits of persistent data structures come with some costs and design trade-offs. To achieve efficient updates, persistent structures usually rely on tree-like representations that support structural sharing. Array-based structures such as `std::vector` or `std::string` cannot simply copy a single contiguous buffer on every modification without losing asymptotic efficiency. Instead, practical persistent vectors and strings (as in Clojure or immer [14]) are implemented as trees of fixed-size chunks. Even then, updates allocate new nodes, producing transient garbage when older versions are no longer referenced. This can make persistent versions of mutable containers inherently slower, especially for array-based containers. We refer to this as

the "persistence tax", and it is generally the main trade-off when using persistent data structures. That said, we aim to minimize the impact as much as possible.

2.4 Value Semantics

In programming languages, *value semantics* refers to a model in which variables hold values rather than references to shared mutable state. When a value is assigned, copied, or passed to a function, the new variable conceptually receives its own independent copy. This is the semantics used by primitive types in C++. For example, consider the two program fragments:

```
int a = 10; int b = a; b++;        int x = 10; int &y = x; y++;
```

The first uses *value semantics*, meaning that the update `b++` does *not* affect the value of `a`. By contrast, the second uses *reference semantics* that associate variables with shared objects in memory, so the update `y++` alters the observable state of `x`—i.e., a *side effect*. Most standard STL containers (`std::vector`, `std::map`, etc.) follow reference semantics for their iterators and internal storage, which can lead to subtle invalidation errors when the underlying container is mutated.

Value semantics are desirable because they simplify reasoning about program behavior. Updates have no side effects beyond the scope of the variable being modified, supporting safer and more modular code. This model closely parallels the principles of *functional programming*, where functions operate on immutable values and return new ones. In practice, however, deep copying of large data structures is inefficient, which has historically limited the use of value semantics to small or immutable types. However, persistent data structures overcome this limitation through structural sharing. It is also important to clarify what value semantics *do not* mean. Value semantics do not require variables themselves to be immutable or prohibit reassignment (as would be required for full *referential transparency*). For example, consider the program fragment:

```
vector a({2,3,5}); vector b = a; b.push_back(7);
```

Programmers can freely write statements such as `b.push_back(7)`, which only changes the value referenced by `b` and not other values, such as those referenced by `a`. Value semantics ensure that each variable refers to an independent logical value, even if the underlying storage is shared.

2.5 Additional Examples and Programming Patterns

Persistent iterators sometimes enable simpler or more efficient formulations over their counterparts. In addition to the motivating example in Section 1.1, we present two small examples that highlight situations where persistent iterators can simplify common programming patterns.

Pattern 1: Stateful Filtering. Given a sequence of C-style statements (including labels and gotos), Figure 3 (a) and (b) remove any statement that is unreachable due to prior control flow. This pattern can be expressed in Haskell using *recursion*, but this relies on an underlying list representation. The pattern falls outside the scope of common Haskell *combinators* (e.g., `map`, `filter`, etc.), but can be expressed by threading a *reachability flag* through a general *left fold* (`foldl`) followed by an explicit `reverse` pass to restore ordering (Figure 3 (a)). In contrast, the LibFpp version (Figure 3 (b)) is expressed as a single-pass iterator loop, where the reachability state is maintained explicitly, and all updates occur under value semantics. This formulation is both natural and idiomatic, and remains fully abstract: the same code applies uniformly across container types without exposing the underlying data-structure. Finally, the LibFpp version exploits *structural sharing* between the input and output, whereas the Haskell version always constructs a new list.

Pattern 2: Snapshot-based State Management. Figure 3 (c) and (d) show a minimal text editor supporting *insert*, *backspace*, *cursor movement*, and *undo*/*redo*. The editor state is represented by a cursor and history stacks storing prior states.

```haskell
filterReachable :: [Stmt] -> [Stmt]
filterReachable =
 reverse . snd . foldl step (True, [])
 where
  step (reach, acc) stmt =
   let reach1 = if isLabel stmt then True else reach
       acc'   = if reach1 then stmt : acc else acc
       reach2 = if isGoto stmt then False else reach1
   in (reach2, acc')
```

(a) Haskell combinator implementation

```cpp
bool reach = true;
auto it = stmts.begin();
for (; it != stmts.end(); ) {
 if (isLabel(*it)) reach = true;
 if (reach) {
  if (isGoto(*it)) reach = false;
  ++it;
 } else it.erase();
}
stmts = it.value();
```

(b) Persistent iterator implementation

```cpp
struct Editor {
  struct Cursor { string buf; size_t pos = 0; };
  Cursor cursor;
  struct { vector<Cursor> undo, redo; } history;
  void commit() {
   history.undo.push_back(cursor);
   history.redo.clear();
  }
  void insert(char c) {
   commit();
   cursor.buf.insert(cursor.pos, 1, c);
   cursor.pos++;
  }
  void backspace() {
   if (cursor.pos == 0) return;
   commit();
   cursor.pos--;
   cursor.buf.erase(cursor.pos);
  }
  void left() {
   if (cursor.pos > 0) cursor.pos--;
  }
  void right() {
   if (cursor.pos < cursor.buf.size()) cursor.pos++;
  }
  void undo() {
   if (history.undo.empty()) return;
   history.redo.push_back(cursor);
   cursor = std::move(history.undo.back());
   history.undo.pop_back();
  }
  void redo() {
    if (history.redo.empty()) return;
    history.undo.push_back(cursor);
    cursor = std::move(history.redo.back());
    history.redo.pop_back();
  }
};
```

(c) STL editor implementation

```cpp
struct Editor {
  using Cursor = string::iterator;
  Cursor cursor;
  struct { vector<Cursor> undo, redo; } history;
  void commit() {
    history.undo.push_back(cursor);
    history.redo.clear();
  }
  void insert(char c) {
    commit();
    cursor.insert(c);

  }
  void backspace() {
    if (cursor.pos() == 0) return;
    commit();
    cursor--;
    cursor.erase();
  }
  void left() {
    if (cursor.pos() > 0) cursor--;
  }
  void right() {
    if (cursor != string::end()) cursor++;
  }
  void undo() {
    if (history.undo.empty()) return;
    history.redo.push_back(cursor);
    cursor = history.undo.back();
    history.undo.pop_back();
  }
  void redo() {
    if (history.redo.empty()) return;
    history.undo.push_back(cursor);
    cursor = history.redo.back();
    history.redo.pop_back();
  }
};
```

(d) Persistent iterator editor implementation

Fig. 3. Sample programs that illustrate the advantages of persistent iterators, including stateful filtering and a minimal text editor. We compare Haskell/STL container implementations (left) and persistent iterator implementations (right). Here, green = $O(1)$ shallow copy, red = $O(N)$ deep copy, and yellow =major differences. Note that some lines are identical syntactically but differ in complexity.

In the STL implementation (Figure 3 (c)), storing editor state requires an $O(N)$ deep copy of the entire string buffer. As such, a simple snapshot-based design is not practical, and real text editors typically use more complex designs (e.g., *command logs*). With Figure 3 (d), the cursor is naturally represented as a *persistent iterator* encoding both the buffer and position. Editor operations map directly to iterator operations (`++`, `--`, `insert`, `erase`). This enables a design where editor state can be captured and restored directly as *values*, without auxiliary data structures or restructuring of the algorithm. In contrast to traditional containers, persistent iterators make this design both

practical and idiomatic: the cursor is a value that encodes both position and structure, and undo/redo reduces to storing snapshots in $O(1)$ time via structural sharing. This pattern generalizes to other applications requiring undo/redo, speculative execution, or versioned state.

3 Design of Persistent Containers

Our persistent container design is built on top of *finger trees* [5]—a general-purpose, persistent data structure that provides asymptotic performance comparable to mutable containers such as those in the C++ STL. For example, and uniquely for finger trees, adding or removing elements from the back (or front) of a tree is an amortized $O(1)$ operation. Similarly, in imperative settings, `std::vector::push_back` or `pop_back` is also an amortized $O(1)$ operation, by storing elements in a contiguous buffer that occasionally resizes and copies. Thus, and already for this example, finger trees are a good fit for a persistent vector implementation: both can be used to represent sequences of elements with a similar algorithmic complexity.

3.1 Finger Trees

A *finger tree* [5] is a specially balanced tree optimized for sequences and fast access near its ends. Conceptually, a finger tree consists of a list of recursive *spine nodes*, each of which emits two *finger nodes* (a left and a right) representing the sequence *prefix* and *suffix* of the child spine node. Each finger node stores typically 1-4 *digits*, where each digit is a balanced 2-3 tree containing the actual elements of the sequence. The key to finger trees is the depth of each digit, with the top-most spine node comprising digits of depth 0 (i.e., individual elements), the second top-most comprising digits of depth 1, etc. Thus, accessing the back (or front) of a finger tree is therefore an $O(1)$ operation, as only the root spine node and rightmost (or leftmost) digit need be accessed. Furthermore, the overall finger tree structure is balanced and maintains logarithmic depth relative to the total number of elements. Thus, top-level fingers provide constant-time access to both ends, while the recursive spine ensures logarithmic access to the interior.

Inserting an element (at either end) modifies only the shallow digits of the corresponding finger. When a digit overflows, a small constant number of nodes (typically 2) are recursively promoted down the spine, maintaining balance. As a result, inserting an element requires only touching $O(1)$ nodes on average. Similarly, deleting an element (at either end) is also an amortized $O(1)$ operation. In addition, finger trees support appending entire sequences, splitting, and accessing/insertion/deletion of elements at random positions, all in $O(\log N)$ time. And since finger trees are trees, updates can be made *persistently* by re-allocating only the nodes along the modified path while reusing all unaffected subtrees for structural sharing. The asymptotic time complexities of finger trees closely resemble their mutable STL counterparts, but with persistence guarantees.

Example. Example finger trees are illustrated in Figure 4. Here, tree Ⓐ illustrates the unique finger tree structure, with a central spine of nodes (red diamonds), each with two fingers (blue boxes), with some number of 2-3 tree digits (green/yellow circles) of increasing depth. This example assumes a branching factor of 2, but in general fingers may have 1-4 digits, and tree nodes may have 2-3 children (i.e., 2-3 trees). Tree Ⓑ is constructed from Ⓐ using a `push_back(30)` operation. Since this updates the end of the structure, only a limited number of nodes (namely, 3) need to be copied, and a significant portion of Ⓐ's structure is shared by Ⓑ. The finger tree structural invariant and operations are complex, so we omit many details for brevity (see the original paper [5] for details).

3.1.1 Persistent Vectors as Finger Trees. Persistent vectors are implemented directly as finger trees. Each `push_back`, `pop_back`, or `back` operation only touches the right-hand finger, providing amortized $O(1)$ performance analogous to `std::vector`. The difference lies in semantics: rather

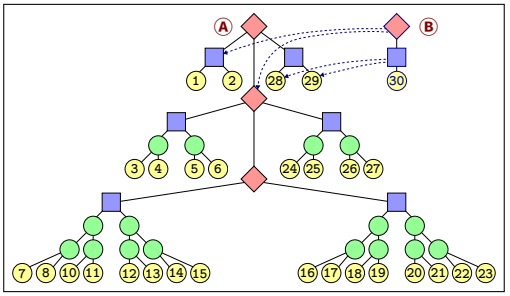


Fig. 4. Example finger trees representing the sequences Ⓐ=⟨1, 2, ..., 8, 10, ...29⟩ and Ⓑ=⟨1, 2, ..., 8, 10, ...30⟩. Here, *spine* nodes are represented by red diamonds, *finger nodes* by blue boxes, 2-3 tree nodes as green circles, and *element* nodes by yellow circles with the corresponding value. Furthermore, tree Ⓑ is created via a `push_back(30)` operation on tree Ⓐ, and benefits from structural sharing. Here, dashed edges represent references from Ⓑ to the original Ⓐ tree nodes.

than mutating the container in place, `push_back` allocates a few new nodes to represent the updated path and returns a new persistent vector, meaning that any previous version will remain valid.

3.1.2 Finger Trees as a Unifying Abstraction (Persistent Containers as Finger Trees). A major insight of our design is that the finger tree serves as a single unifying abstraction for *all* common container types—not only *vectors* but also *(multi)sets*, *(multi)maps*, and *strings*. These conventional containers only fundamentally differ in the invariants that are maintained over the underlying sequence. This gives rise to a natural type hierarchy:

- A *vector* is a sequence;
- A *multiset* is a sorted vector;
- A *set* is a multiset with no duplicate elements;
- A *multimap* is a multiset of key-value pairs ordered by key;
- A *map* is a multimap with unique keys; and
- A *string* is a vector of characters under the UTF-8 encoding.

Essentially, *all* common container types are special cases of sequences, and sequences have a universal persistent implementation: the finger tree. In our implementation, each container is realized as a finger tree equipped with a specialized *monoid* [5] summarizing subtree information. For *(multi)sets* and *(multi)maps*, the monoid is the *minimum element* of the left subtree, and for *strings*, the monoid is the total Unicode length under the UTF-8 encoding (rather than the total single-byte characters). All containers, therefore, share the same structural backbone and persistence mechanism, and only their monoid semantics differ. Since vectors already use a tree-based representation, accessing/insertion/deletion of elements in sets and maps still uses a tree-based search in $O(\log N)$ time—the same as the STL counterparts. This uniform design yields three significant benefits:

- *Zero-cost upcasts*: containers can be safely upcast without conversion overhead. For example, viewing a set as a vector requires no copying since the underlying data representation is identical. Similarly, UTF8-strings can be viewed as vectors of single-byte characters.
- *Semantic uniformity*: algorithms can be written generically over a common finger-tree implementation, avoiding multiple diverging container-specific implementations.
- *Implementation simplicity*: the entire library is built atop a single core data structure, greatly reducing duplication and the potential for asymptotic "surprises" across container types.

For example, the algorithmic choice in Figure 1 (b) and (c) depends on the implementation details of the underlying container type, which is a form of abstraction leakage. Under our design, the problem would not exist, since all containers use the same unified finger tree implementation.

4 Design of Persistent Iterators

Having established a persistent foundation for containers, we now turn to *persistent iterators*—our mechanism for traversing and modifying containers while retaining full persistence. The goal is to

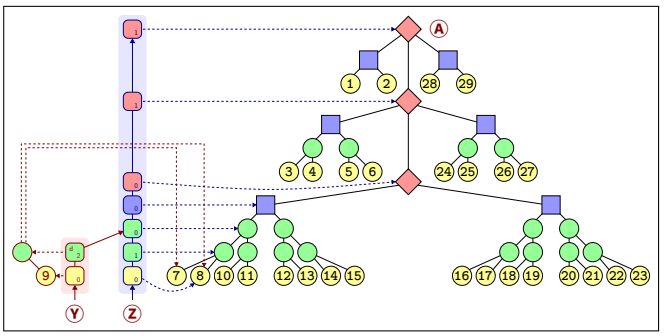


Fig. 5. Example finger tree (A) with a zipper (Z) pointing to element 8, and a zipper (Y) pointing to a newly *inserted* element 9. Note that zippers are also persistent structures with complex structural sharing relationships (even between zippers), as represented by the dashed lines.

provide a familiar, STL-style iterator interface without reintroducing the aliasing and invalidation hazards that arise from the default STL iterator *reference semantics*.

4.1 Persistent Iterators as Zippers

We implement persistent iterators using a variant of the *zipper* [6], a pure functional data structure that represents a "cursor" or "position" into another persistent structure. A zipper decomposes a tree (such as a finger tree) into a *reversed path* from a node (such as a leaf element) back to the tree's root, capturing the context necessary to reconstruct the full tree structure. This representation enables direct $O(1)$ access to the *current element*, which is at the *head* of the reversed path, rather than $O(\log N)$ access from the root. Navigation operations (`++`, `--`, etc.) are implemented by moving up the reversed path to the shared ancestor, and then down a neighboring branch.

In our implementation, each zipper node encodes a tuple:

$$\langle \textit{tree-node}, \textit{child-index}, \textit{dirty-flag}, \textit{parent-node} \rangle$$

Here, *tree-node* is the finger tree node in the path, *child-index* indicates which child is taken along the path, *dirty-flag* (explained below) indicates whether *tree-node* has been modified from the parent, and *parent-node* is the zipper node containing the parent of *tree-node* (or *null* for the root node). Unlike the original zipper design [6], our variant minimizes allocations by storing direct references to existing finger tree nodes along with their child indices, rather than duplicating path information. The top-level zipper head node corresponds to the current element, and its parent pointer recursively encodes the rest of the reversed path toward the root.

Example. An example finger tree (A) and zipper (Z) is illustrated in Figure 5. The (Z) zipper is a list-like structure starting from the bottom up. The head of the zipper contains a reference to an *element node* of the finger tree, in this case, the node containing `8`. Thus, the element can be accessed in $O(1)$ time from the head node of the zipper, as opposed to $O(\log N)$ time from the tree root. The remaining zipper nodes contain references to finger tree nodes and a corresponding *child-index* (represented as a sub-script), and thus are a complete encoding of a finger tree and (reversed) path to a specific element.

4.1.1 Navigation. Iterator movement corresponds to reconstructing or extending the reversed path. To move from one position to another, the zipper ascends until it reaches the *lowest common ancestor* of the source and destination elements, and then descends along the new path. For local operations such as increment `++it` or decrement `--it`, this ancestor is typically close, yielding amortized $O(1)$ complexity. Larger jumps, such as (`it += N`), take $O(\log N)$ time in general. Importantly, zippers—and therefore iterators—are themselves persistent. Multiple iterators derived from the same container will share common path segments (i.e., *structural sharing*), and modifying or advancing one iterator does not invalidate or modify any other.

4.1.2 Element Update, Insertion, and Deletion. Persistent iterators support in-place-style updates, such as *assignment*, *insertion*, and *erasure*, without violating persistence. To achieve this, we implement a lazy update mechanism based on the zipper's *dirty-flag*. When an element is replaced (assigned) to new value (x) via an iterator `i`, a new iterator value `j` will be created by:

- Allocating a new finger tree leaf node n with x as the updated element value;
- Allocating a new zipper head node `j` that references the new element node; and
- Marking the new zipper head's *dirty-flag* as set.

In other words, the new iterator value `j` is defined as follows:

$$\texttt{j} = \langle n,\ \texttt{i}.\textit{child-index},\ \textit{true},\ \texttt{i}.\textit{parent-node} \rangle$$

As such, assignment is an $O(1)$ operation. The dirty flag indicates that the current subtree below the parent has been updated and must be reconstructed upon traversal. When the iterator subsequently moves, the dirty flag is propagated upward, causing the affected finger tree nodes to be lazily rebuilt along the modified path until eventually reaching the root. This design allows multiple element updates to be "stacked" efficiently, without requiring a full $O(\log N)$ path reconstruction for each update.

Element *insertion* and *deletion* (a.k.a. *erasure*) are handled similarly. However, since these operations may affect local balance, the zipper will unwind to the nearest ancestor such that balance can be restored. The unwinding depth is amortized $O(1)$ steps. Once the finger tree invariant is restored, the zipper node is marked as dirty, and the operation completes. As a result, persistent iterator assignments, insertions, and deletions can also be "stacked", and each operation runs in (amortized) $O(1)$ time for localized updates while preserving persistence and structural sharing.

Example. Another zipper Ⓨ is also illustrated in Figure 5. Here, the missing element 9 has been *inserted* into the sequence through the original zipper Ⓩ. This involves unwinding the original zipper to the nearest ancestor such that a new balanced subtree can be constructed. In this example, we need to unwind to depth 1 and convert the corresponding 2-node into a 3-node which holds the additional 9 element. The modified finger tree is *not* reconstructed in its entirety. Rather, the *dirty-flag* (d) is merely marked on the corresponding zipper node, and the modified finger tree will be reconstructed *lazily*.

Crucially, the new zipper Ⓨ is still a *persistent* data structure, and structurally shares its tail with the original zipper Ⓩ. A complex web of structural sharing references is created, as illustrated in Figure 5, and deciphering this web is left as an exercise for the reader. Nevertheless, the original finger tree Ⓐ and the original zipper Ⓩ remain *valid* and have not been mutated—i.e., this yields side-effect-free value semantics.

4.1.3 Value Extraction. Because persistent iterators possess *value semantics* rather than *reference semantics*, an iterator may represent a locally modified view of the container that differs from the original version. To materialize this view as a first-class container value, we provide the `it.value()` operation that returns the iterator's current logical container—a persistent version that reflects all updates made through the iterator. The implementation of `value` is conceptually simple: starting from the iterator's current zipper node, the operation traverses the reversed path back to the root of the finger tree, an $O(\log N)$ operation in the worst case. During unwinding, any nodes marked as dirty are reconstructed to incorporate pending modifications. Importantly, `value` is a pure operation: it does not invalidate or alter the iterator state.

Example. Extracting the value from the zipper Ⓨ from Figure 5 will propagate the *dirty-flag* updates and reconstruct a finger tree over the full sequence $\langle 1, 2, \ldots, 29 \rangle$ including the previously missing element 9.

```
# allocation (result in %rax)

mov HEAP, %rbx
mov (%rbx), %rax
lea 64(%rbx, %rax), %rax
mov %rax, HEAP
```

```
# deallocation (node pointer in %rax)
mov HEAP, %rbx
sub %rax, %rbx
sub $64, %rbx
mov %rbx, (%rax)
mov %rax, HEAP
```

Fig. 6. Highly optimized (de)allocation routines (x86_64, single-threaded), with 4 inlined instructions for allocation, and 5 inlined instructions for deallocation.

4.1.4 Zippers as a Unifying Iterator Abstraction. As with persistent containers, our persistent iterator design is uniform across all common container types. Since every container in the library (*vectors*, *sets*, *maps*, and *strings*) is ultimately a specialization of the same finger tree abstraction, all iterators share a single implementation.

5 Optimization

Although persistent data structures offer strong semantic and algorithmic advantages, they can easily become allocation-bound. This is because each update typically requires creating and releasing $O(\log N)$ nodes, and this cost may dominate execution time for some workloads. Moreover, tree-based data structures are inherently non-contiguous in memory, which can harm locality. These issues are the primary reason behind the "persistence tax". This section describes the low-level optimizations used by LibFpp to help minimize this tax.

5.1 A Very Fast Custom Memory Allocator (CMA)

Our first optimization tackles the cost of allocation by designing a specialized allocator and deallocator that execute in just a few machine instructions. Our design is based on two key insights.

Uniform Node Size. The first insight is that all finger-tree and zipper nodes in LibFpp have roughly uniform size and fit within a single 64-byte cache line. This allows us to implement a *fixed-size* free-list allocator, where: *allocation* pops an entry from the head of the free-list, and *deallocation* pushes the node back. Because no variable-size blocks need to be handled, the allocator is trivially simple and efficient.

Zero-initialized Memory as a Linked List. The second insight is that zeroed memory can itself encode a valid linked list. Instead of storing a raw pointer to the *next* free node, each node stores the *offset* between the next free node (in the list) and the next node in memory. Thus, if the next free node (in the list) happens to be the next free node (in memory), the offset will be `0x0`.

This means that a zeroed memory region is already a valid free-list over its length, without any explicit initialization pass. The *heap* for the CMA is created with a single call to `mmap` with the `NO_RESERVE` flag—ensuring that the heap is zero'ed and *lazily committed* and grows by demand as the address space expands. Furthermore, heap allocation never *fails* in the traditional sense, i.e., by returning `NULL`. Rather, heap allocation behaves conceptually like stack allocation over a bounded region: physical memory usage is based on demand. The resulting allocator is *extremely compact* for single-threaded code, and (de)allocation is reduced to a handful of instructions (see Figure 6). Our implementation supports different allocators, including a multi-threaded variant of the CMA, as well as wrappers for the standard allocators `new` and `delete`.

5.2 Other Optimizations

While node allocation is extremely cheap, it is also not free. LibFpp therefore performs *destructive updates* whenever it is safe to do so—specifically, when all nodes along an update path have a reference count of exactly one. In such cases, the affected nodes are updated *in-place* rather than

copied. LibFpp applies this to many common operations, including iterator increment `++it`. In practice, a large fraction of nodes meet the "unique reference" condition, so a destructive update can eliminate the majority of allocations in typical workloads. Although not novel (also implemented by Immer [14] and other systems), this optimization helps minimize algorithmic constant factors.

Finally, LibFpp packs multiple elements into each leaf node, as much as possible to fill an entire cache line. This reduces tree depth, improves spatial locality, and minimizes traversal overhead. Fat leaves are especially beneficial for sequential scans, where they exploit hardware prefetching and contiguous memory access patterns.

6 Evaluation

To evaluate the effectiveness of LibFpp persistent containers/iterators, we consider the following:

(Algorithmic Complexity) What is the algorithmic complexity of LibFpp container operations compared to the baseline?
(Constant Factors) What are the constant factors of LibFpp container operations?
(Memory Overheads) What is the memory overhead of LibFpp containers?

For the constant factor and memory overheads, we focus on operation-level micro-benchmarks to isolate the cost of persistence and functional updates.

6.1 Experimental Setup

As LibFpp is intended to act as a persistent container alternative for general container and iterator usages, we compare it against several popular libraries in this space as baselines. Specifically, we compared against the following libraries:

- *Standard Template Library* (STL) (`libc++` 21.0.0): Containers and iterators have been present since the first iteration of the Standard Template Library back in 1995 [19]. Due to its inclusion in the `C++` standard library, commonly shipped together with popular `C++` compilers, the STL has become the default for container and iterator usage in `C++` programming.
- Immer (0.8.1): Immer [14] is an alternative persistent container library realizing *Relaxed Radix Balanced* (RRB) trees for sequence-like containers *Compressed Hash-Array Mapped Prefix-tree* (CHAMP) structures [18] for maps/sets. Unlike LibFpp, Immer only supports *const* rather than *persistent iterators*, meaning that it is not possible to update containers via iteration.
- Abseil (20250814.1): Abseil [21] is a popular open-source collection of `C++` library code designed to augment the `C++` standard library, containing a significant number of useful functionalities used internally at Google that are not present in the STL. Alternative containers also form a significant component of Abseil, including Swiss table versions of unordered containers, such as `absl::flat_hash_map`, and B-Tree versions of ordered containers, such as `absl::btree_map`. Similar to LibFpp, Abseil containers are designed to be replacements for their STL counterparts, following the standard container API if possible. Unlike LibFpp and Immer, Abseil containers are not persistent.
- Folly (2025.11.03.00): Folly [10] (Facebook Open Source Library) is another popular open-source `C++` library with components designed with practicality and efficiency in mind. Similar to Abseil, Folly is essentially a collection of useful internal components extensively deployed inside Meta, and only strives to complement the functionalities of the STL instead of replacing it. Containers also form a significant component of Folly, including several items that the standard library neglects to provide, such as high-performance atomic data structures, a drop-in replacement version of `std::string` and `std::vector`, intrusive linked lists, and more. Unlike LibFpp and Immer, Folly containers are not persistent.

Finally, as the high-performance thread-local CMA is used in the default implementation of LibFpp, an alternative version marked as "LibFpp (unopt)" is also evaluated with the standard allocator instead to reduce non-algorithmic impact on performance evaluations.

6.2 Algorithmic Complexity

To evaluate the theoretical efficiency of LibFpp containers, we performed a complexity analysis of all core operations and compared them with their counterparts in the C++ *Standard Template Library* (STL). The analysis is based on the known asymptotic properties of the persistent data structures underlying LibFpp, namely, finger trees and zippers. For each operation, we consider either the *amortized* or *worst-case* cost, depending on the semantics of the corresponding STL operation. The results are summarized in Table 2.

Results. LibFpp provides persistent counterparts to nearly all common STL container and iterator operations, while preserving similar syntax and equivalent time/space complexities in most cases. This equivalence is by design: finger trees achieve $O(1)$ amortized append and access to the ends of a sequence, and zippers support traversal in constant time increments, yielding asymptotic behavior comparable to standard mutable containers.

That said, there are some notable differences arising from the fundamental distinction between array-based and tree-based representations. For example, random access `v[idx]` and arbitrary iterator jumps (`it += n`) require $O(\log N)$ time in LibFpp, whereas contiguous STL vectors perform the same operations in $O(1)$. This is a direct consequence of persistence: maintaining previous versions efficiently requires node-based structural sharing rather than contiguous arrays. Such logarithmic penalties are typical of all general-purpose persistent sequences.

Conversely, several operations benefit from the persistent design. Element insertion and erasure through iterators (`it.insert(x)`, `it.erase()`) are amortized $O(1)$ operations under LibFpp, compared to $O(N)$ or their STL vector equivalents. String concatenation and splitting are likewise logarithmic rather than linear, as finger trees support efficient join and split operations. Sequence containers in LibFpp naturally support both `push_front` and `push_back` in amortized $O(1)$ time, similar to `std::deque` but without additional container specialization. The results from Table 2 also show that the complexity for *any* standard operation is no more than $O(\log N)$. This reduces the risk of "complexity hazards" when implementing natural versions of standard algorithms, such as those that exist in the STL (e.g., see Figure 1 (b) over `std::string`).

> LibFpp achieves time and space complexities comparable to the STL across nearly all operations while adding full value semantics without complexity hazards. In cases where asymptotic costs differ, the trade-offs stem from persistence requirements rather than inefficiency in design.

6.3 Constant Factors

While asymptotic complexity provides a useful theoretical baseline, the practical performance of a container implementation also depends on the associated constant factors. Persistent data structures are inherently disadvantaged in this regard: structural sharing and indirection reduce spatial locality, and operations such as updates and traversal often involve pointer chasing across disjoint memory regions. Consequently, we do not expect persistent containers to outperform their mutable counterparts. Nevertheless, the following section focuses on evaluating constant-factor effects through targeted micro-benchmarks to quantify the impact of persistence.

6.3.1 Containers. All experiments were carried out on an macOS `arm64` machine operating on M2 Pro with 32 GB of memory, compiled with Clang 21.1 released in August 2025. The benchmarks

Table 2. Comparison of time and space complexities between the *Standard Template Library* (STL) and LibFpp containers. Key: v=*vector*, m=*map*/*set*, s/t=*strings*, idx/n=*integers*, x=*value*, k=*key*, c=*character*, green =better, yellow =same, red =worse.

STL Operation	STL Time	STL Space	LibFpp Operation	LibFpp Time	LibFpp Space
			Vector		
`v.size()`	$O(1)$	$O(1)$	`v.size()`	$O(1)$	$O(1)$
`v[idx]`	$O(1)$	$O(1)$	`v[idx]`	$O(\log N)$	$O(1)$
`v.back()`	$O(1)$	$O(1)$	`v.back()`	$O(1)$	$O(1)$
`v.front()`	$O(1)$	$O(1)$	`v.front()`	$O(1)$	$O(1)$
`v.push_back(x)`	$O(1)$	$O(1)$	`v.push_back(x)`	$O(1)$	$O(1)$
n/a	-	-	`v.push_front(x)`	$O(1)$	$O(1)$
`v.pop_back()`	$O(1)$	$O(1)$	`v.pop_back()`	$O(1)$	$O(1)$
n/a	-	-	`v.pop_front()`	$O(1)$	$O(1)$
`it = v.begin()`	$O(1)$	$O(1)$	`it = v.begin()`	$O(1)$	$O(1)$
`it = v.insert(it,x)`	$O(N)$	$O(1)$	`it.insert(x)`	$O(1)$	$O(1)$
`it = v.erase(it)`	$O(N)$	$O(1)$	`it.erase()`	$O(1)$	$O(1)$
`*it = x`	$O(1)$	$O(1)$	`it.assign(x)`	$O(1)$	$O(1)$
`++it/--it/*it`	$O(1)$	$O(1)$	`++it/--it/*it`	$O(1)$	$O(1)$
`it += n`	$O(1)$	$O(1)$	`it += n`	$O(\log N)$	$O(1)$
			Set and **Map**		
`m.size()`	$O(1)$	$O(1)$	`m.size()`	$O(1)$	$O(1)$
`m[k]`	$O(\log N)$	$O(1)$	`m[k]`	$O(\log N)$	$O(1)$
n/a	-	-	`m[idx]`	$O(\log N)$	$O(1)$
`m.contains(k)`	$O(\log N)$	$O(1)$	`m.contains(k)`	$O(\log N)$	$O(1)$
`m.insert(x)`	$O(\log N)$	$O(1)$	`m.insert(x)`	$O(\log N)$	$O(1)$
`m.erase(x)`	$O(\log N)$	$O(1)$	`m.erase(x)`	$O(\log N)$	$O(1)$
`it = m.begin()`	$O(1)$	$O(1)$	`it = m.begin()`	$O(1)$	$O(1)$
`it = m.find(x)`	$O(\log N)$	$O(1)$	`it = m.find(x)`	$O(\log N)$	$O(\log N)$
`it = m.erase(it)`	$O(1)$	$O(1)$	`it.erase()`	$O(1)$	$O(1)$
`++it/--it/*it`	$O(1)$	$O(1)$	`++it/--it/*it`	$O(1)$	$O(1)$
n/a	-	-	`it += n`	$O(\log N)$	$O(1)$
			String		
`s.size()`	$O(1)$	$O(1)$	`s.size()`	$O(1)$	$O(1)$
`s[idx]`	$O(1)$	$O(1)$	`s[idx]`	$O(\log N)$	$O(1)$
`s += c`	$O(1)$	$O(1)$	`s += c`	$O(1)$	$O(1)$
n/a	-	-	`s.push_front(c)`	$O(1)$	$O(1)$
`s += t`	$O(N+M)$	$O(N+M)$	`s += t`	$O(\min(\log N, \log M))$	$O(\min(\log N, \log M))$
`s.substr(idx)`	$O(N)$	$O(N)$	`s.substr(idx)`	$O(\log N)$	$O(\log N)$
`it = s.begin()`	$O(1)$	$O(1)$	`it = s.begin()`	$O(1)$	$O(1)$
`it = s.insert(it,c)`	$O(N)$	$O(1)$	`it.insert(c)`	$O(1)$	$O(1)$
`it = s.erase(it)`	$O(N)$	$O(1)$	`it.erase()`	$O(1)$	$O(1)$
`*it = c`	$O(1)$	$O(1)$	`it.assign(c)`	$O(1)$	$O(1)$
`++it/--it/*it`	$O(1)$	$O(1)$	`++it/--it/*it`	$O(1)$	$O(1)$
`it += n`	$O(1)$	$O(1)$	`it += n`	$O(\log N)$	$O(1)$

in these experiments use the popular testing framework Catch2 [22] (3.11.0), which provides comprehensive unit-testing for LibFpp itself. To evaluate the constant factors associated with LibFpp's persistent containers, we use the four micro-benchmarks adapted from [14].

- Access: The sum of all values in a n element container is computed by sequential indexes;
- Append: An n element container is produced by sequentially appending n elements;
- Update: Every element in an n element container is updated using sequential indexes;
- Concat: An n element container is produced by concatenating 10 equally sized containers.

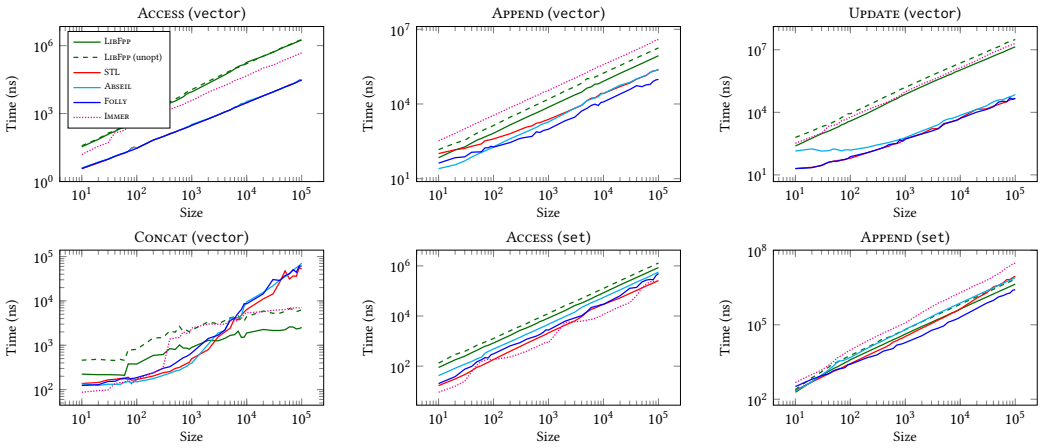


Fig. 7. Figure showing container micro-benchmark results for *vectors* and *sets* in LibFpp and other libraries under different container sizes. Scales are logarithmic. For Immer, `flex_vector` is used for Concat support. Compared to the STL baseline, LibFpp is 56.9×/3.4×/262.0× slower for the Access/Append/Update (`vector`) benchmarks, 3.3× slower for the Access benchmark, 7.6× faster for the Concat (`vector`) benchmark, and 1.8× faster for the Append (`set`) benchmark.

We evaluate *vectors* as a representative of sequence-based containers (*vectors*, *strings*, etc.), and *sets* as a representative of node-based containers (*multisets*, *maps*, etc.). Note that Update and Concat are for sequence-based containers only.

Results. The micro-benchmark results are summarized in Figure 7. As expected, the performance characteristics of LibFpp are broadly similar to those of Immer, since both employ persistent tree-based representations with structural sharing. For sequence-based containers, however, the mutable counterparts retain a substantial advantage due to their *array-based* implementations. For example, index-based access `v[idx]` on a contiguous array is generally compiled into just a few machine instructions, and also benefits from excellent cache locality. This difference in representation results in a 3.4× slowdown for the Append (`vector`) benchmark, where both LibFpp and STL implement an amortized $O(1)$ operation. For Access and Update (`vector`), both persistent implementations must traverse a tree structure to locate the element ($O(\log n)$ versus $O(1)$), resulting in higher observed overheads in the order of 10×-100× or more. For Concat (`vector`) the opposite occurs, where the tree-based representation offers an algorithmic advantage over array-based containers, resulting in a speedup. These cost differences are fundamental to persistence and are not easily eliminated without sacrificing immutability.

Within the persistent domain, the design trade-offs of LibFpp and *Immer* become more apparent. Immer employs *Relaxed Radix Balanced* (RRB) [20] trees for sequence containers and *Compressed Hash-Array Mapped Prefix-tree* (CHAMP) structures [18] for maps/sets, each with a large node fan-out (typically up to 32). LibFpp, by contrast, uses a uniform finger-tree foundation with a fixed node size of 64 bytes. This distinction is reflected in the results: LibFpp shows higher constant-factor overheads for indexed benchmarks such as Access, but achieves lower costs for modification-heavy workloads such as Append, Update, and Concat, regardless of allocator choice. The performance gap is primarily attributable to differences in high fan-out RRBs/CHAMPs versus finger trees, with the former allowing for shallower lookups at the cost of more expensive updates.

6.3.2 Iterators. To test persistent iterators, we consider the following modified micro-benchmarks:

- Access*: The sum of all values in a n element container is computed by sequential iteration;

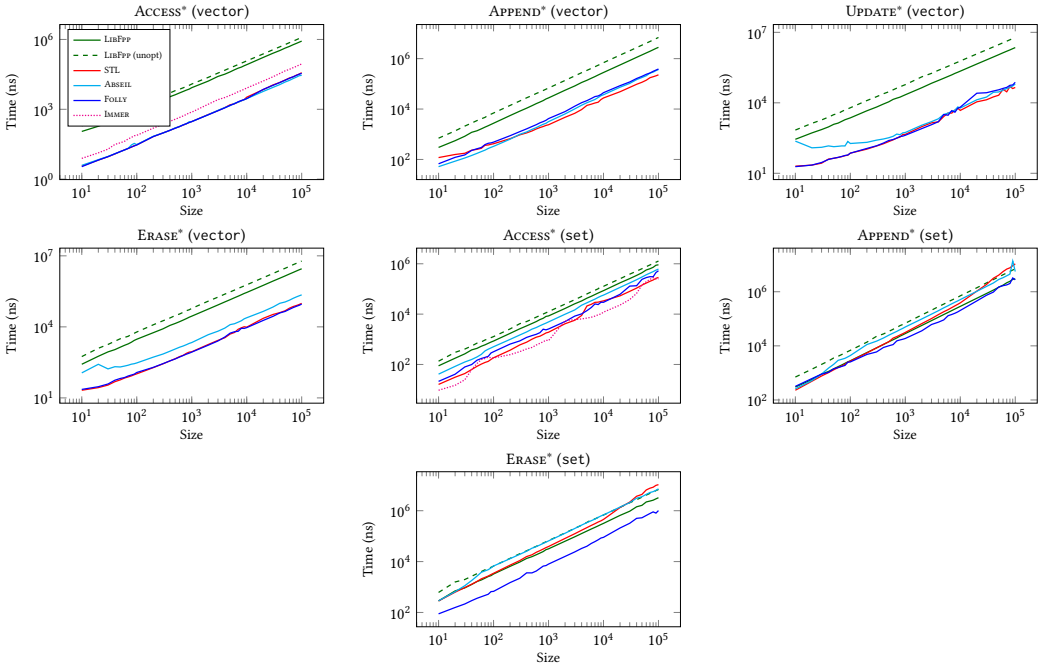


Fig. 8. Figure showing iterator micro-benchmark results for *vectors* and *sets* in LibFpp and other libraries under different container sizes. Scales are logarithmic. Note that Immer's const-iterators can only be applied to Access*. Compared to the STL baseline, LibFpp is 23.6×/11.8×/45.2×/28.2× slower for the Access*/Append*/Update*/Erase* (vector) benchmarks, 3.1× slower for the Access* (set) benchmark, and 2.9×/2.9× faster for the Append*/Erase* (set) benchmarks.

- Append*: An n element container is produced by appending n elements via an iterator;
- Update*: Every element in an n element container is updated via an iterator;
- Erase*: Erase n elements of an n element container via an iterator.

Note that (Access*) also appears in the Immer paper [14]. That said, since Immer only supports *const* iterators, it cannot be evaluated on (Append*), (Update*), nor (Erase*), since these micro-benchmarks require updates via an iterator. Note also that the (Concat) benchmark, from Section 6.3.1, has been replaced by (Erase*) since (1) the iterator version of (Concat) is similar to (Append*), and (2) to test erasure via an iterator.

Results. The iterator-focused micro-benchmark results are summarized in Figure 8. In sequential containers, LibFpp has the same algorithmic complexity trend as other popular libraries, but requires a larger constant factor in access-frequent benchmarks due to the inherent internal overhead of persistent iterators. This is not surprising, since iterator operations like `++it` compile down into a few instructions for array-based container implementations, but require a *zipper move* operation for persistent iterators. The overheads are much narrower for node-based containers, reflecting the inherent cost of node traversal. The tradeoff is a much safer usage of iterators with value semantics, and many operations (such as insertion and deletion) carry no risk of iterator invalidation or other aliasing hazards. LibFpp also showed competitive performance advantages when employing modification-heavy workloads such as (Erase*). Comparison with a slower version of LibFpp also shows that the allocator choice manifested as a constant overhead independent of container size,

and thus does not affect the complexity trend of the library. Finally, Immer, with only support for *const* iterators, is unable to execute all of the micro-benchmarks.

LibFpp achieves practical performance comparable to existing persistent libraries, while offering a more general and expressive persistent iterator design.

6.4 Memory Overheads

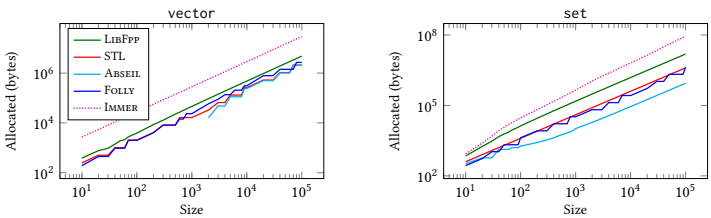


Fig. 9. Figure showing heap consumption for *vectors* and *sets* in LibFpp and other libraries under different container size. Scales are logarithmic. For Immer, transient-mode is not used.

In addition to the asymptotic complexity and run time of containers, memory consumption is also an important metric. In light of this, each of LibFpp and other baseline containers are parameterized with unsigned 64-bit integers, and their heap consumption with regard to different numbers of elements stored is summarized in Figure 9.

Due to the more complex data structure hidden within LibFpp and Immer's containers, their respective heap memory consumption is higher than that of comparatively simple containers, such as `std::vector`. However, as the scale of the dataset grows, LibFpp's provided vectors and sets can still attain comparable memory consumption compared to baseline offerings, proving its scalability across a wide range of data sizes. One notable thing from the figure is the behavior of `absl::InlinedVector`, which by default stores the first 10^3 elements on the stack for better performance, and thus will have 0 heap consumption when using smaller sizes.

LibFpp achieves reasonable memory consumption comparable to existing baseline libraries, while further reducing its memory overhead in real workflows due to the usage of copy-on-write-like techniques that avoid copying until necessary.

7 Related Work

Functional and Declarative Languages. Functional and declarative languages (Haskell, ML, and their derivatives) support persistent data structures as a natural consequence of immutability. However, these languages generally do not provide iterator-like abstractions similar to C++. Rather, traversal in these languages is typically achieved via higher-order *combinators* (e.g., `map`, `foldl`, `zipWith`) or recursion. However, as noted in Section 1.1, these have limitations in natural expressiveness, abstraction, or performance.

*Zippers. Zipper*s have been explored in functional programming literature and practice—e.g., from Huet's original paper [6], to later extensions for trees, syntax manipulation, and XML editing. However, zippers are rarely used in mainstream standard libraries for functional languages, and mostly appear as third-party packages (e.g., `Data.Tree.Zipper`), if at all. Similarly, ML-family languages do not include zipper-based traversal in their core persistent collection libraries. The problem with zippers in functional/declarative languages is two-fold. Firstly, each movement

constructs a new path context, generating transient allocations and GC pressure. Secondly, zippers introduce a *stateful* cursor abstraction that tends to feel unnatural in declarative languages. LibFpp revisits the zipper concept in an imperative setting where iterators are familiar and idiomatic, and performance is addressed through custom memory management and optimizations such as destructive update. As far as we are aware, LibFpp is the first implementation of zippers for finger trees and the first to use zippers to implement an STL-like iterator-like API.

Immer. Immer [14] is a modern `C++` library providing persistent containers, and is perhaps the closest existing system to our work. Immer implements sequence containers using RRB-trees [20] and associative containers using CHAMPs [18]. Immer uses a large node fan-out for performance, allowing for fast random $O(\log_{32} N)$ access while also preserving structural sharing. In contrast, our design prioritizes a uniform abstraction over recovering the constant factors of mutable data structures. Immer's public iterators are *read-only*: i.e., `iterator` and `const_iterator` alias to the same type. In contrast, LibFpp provides full *persistent* iterators with value semantics, allowing for updates (to a local copy) via the iterator. Finally, Immer exposes *persistence* and *transience* through explicit modes, allowing developers to manually switch between the two for performance tuning. In contrast, LibFpp is *transparent*: persistence is automatic and orthogonal to syntax, supporting familiar STL-style while also benefiting from efficient structural sharing.

Abseil, Folly, and Boost. Several widely used `C++` libraries extend or complement the STL, including Abseil [21] (Google), Folly [10] (Meta), and Boost [17]. These frameworks share a common goal of improving the performance, safety, and ergonomics of standard containers, but generally do not attempt to provide persistent or immutable data structures. LibFpp is also intended to complement the STL, by providing persistent versions of containers/iterators while also supporting STL-like patterns and idioms.

8 Conclusion

Persistent data structures are often perceived as elegant but impractical for systems programming. This work demonstrates that, with careful engineering, they can be made practical and ergonomic within the expected trade-offs of persistence. We introduced *persistent iterators*, a design that reconciles the iterator-based programming model of imperative languages with the immutability guarantees of persistent data structures. By combining finger trees with zipper-based traversal, LibFpp provides a uniform, STL-like interface supporting structural sharing, persistence, and STL-style iterator operations under value semantics; thereby eliminating invalidation and aliasing hazards.

Our analysis and evaluation show that LibFpp achieves asymptotic complexities comparable to standard STL containers, albeit with higher constant factors consistent with persistence. We show these additional overheads can be reduced through a combination of custom memory management, destructive update optimizations, and cache-aware node layouts. The resulting library makes persistence and pure value semantics more accessible under traditional (imperative, mutation-based) programming styles, and can serve as a practical alternative for modern software where persistence and immutability are desired. By bringing persistence and value semantics into the familiar idioms of iterator-based programming, LibFpp enables developers to write safer, side-effect-free, and conceptually simpler code.

Acknowledgements

This research is supported by the National Research Foundation, Singapore, under its National Cybersecurity R&D Programme (Award No. CRPO-GC5-NUS-004).

Data-Availability Statement

LibFpp is released under a permissive open-source license:

https://github.com/GJDuck/libfpp

The version of LibFpp used in the artifact evaluation is available at [9].

References

[1] 2024. *Programming Languages — C++*. Technical Report 14882:2024. ISO/IEC.

[2] J. Driscoll, N. Sarnak, D. Sleator, and R. Tarjan. 1986. Making data structures persistent. In *Symposium on Theory of computing*. ACM.

[3] J. Eyolfson and P. Lam. 2016. C++ const and immutability: an empirical study of writes-through-const. In *European Conference on Object-Oriented Programming*. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. doi:10.4230/LIPIcs.ECOOP.2016.8

[4] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. 1995. *Design patterns: elements of reusable object-oriented software*. Addison-Wesley.

[5] R. Hinze and R. Paterson. 2006. Finger trees: a simple general-purpose data structure. *Journal of Functional Programming* 16, 2 (2006), 197–217. doi:10.1017/S0956796805005769

[6] G. Huet. 1997. The Zipper. *Journal of Functional Programming* 7, 5 (1997), 549–554. doi:10.1017/S0956796897002864

[7] N. Josuttis. 2024. *Healing the Filter View*. Technical Report P3329R0. ISO/IEC JTC1/SC22/WG21 C++ Standards Committee. https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2024/p3329r0.pdf WG21 paper.

[8] J. Lakos. 1996. *Large-Scale C++ Software Design*. Addison-Wesley.

[9] Yihe Li and Gregory Duck. 2026. *Artifact for the paper "Persistent Iterators with Value Semantics" - Version 2*. doi:10.5281/zenodo.19392634

[10] Meta. 2026. Folly: An open-source C++ library developed and used at Facebook. https://github.com/facebook/folly.

[11] Y. Nikolakopoulos, A. Gidenstam, M. Papatriantafilou, and P. Tsigas. 2015. *Of Concurrent Data Structures and Iterations*. Springer International Publishing, Cham, 358–369. doi:10.1007/978-3-319-24024-4_20

[12] C. Okasaki. 1995. Simple and efficient purely functional queues and deques. *Journal of Functional Programming* 5, 4 (1995), 583–592. doi:10.1017/S0956796800001489

[13] Oracle Corporation. 2026. Java Collections Framework. https://docs.oracle.com/en/java/javase/21/docs/api/java.base/java/util/doc-files/coll-overview.html

[14] J. Puente. 2017. Persistence for the masses: RRB-vectors in a systems language. In *International Conference on Functional Programming*. ACM. doi:10.1145/3110260

[15] Python Software Foundation. 2026. Iterator Types. https://docs.python.org/3/library/stdtypes.html#iterator-types

[16] Rust Project Developers. 2026. Iterator Trait. https://doc.rust-lang.org/std/iter/trait.Iterator.html

[17] Boris Schling. 2011. *The Boost C++ Libraries*. XML Press.

[18] M. Steindorfer and J. Vinju. 2015. Optimizing hash-array mapped tries for fast and lean immutable JVM collections. In *Object-Oriented Programming, Systems, Languages, and Applications*. ACM. doi:10.1145/2814270.2814312

[19] A. Stepanov and M. Lee. 1995. *The Standard Template Library*. Technical Report. Hewlett-Packard Laboratories.

[20] N. Stucki, T. Rompf, V. Ureche, and P. Bagwell. 2015. RRB vector: a practical general purpose immutable sequence. In *International Conference on Functional Programming*. ACM. doi:10.1145/2784731.2784739

[21] The Abseil Team. 2026. Abseil Common Libraries (C++). https://github.com/abseil/abseil-cpp.

[22] The Catch2 Team. 2026. Catch2: A modern, C++-native, test framework for unit-tests, TDD and BDD. https://github.com/catchorg/Catch2.

Received 2025-11-14; accepted 2026-04-03