A Novel Dynamically Truncating Object Pool for Game Loops and Other Use Cases in Rust

Master’s Thesis
University of Turku
Department of Computing
Computer Science
February 2025
Author: Frans Saukko
Supervisor: Jouni Smed

The originality of this thesis has been checked in accordance with the University of Turku quality assurance system using the Turnitin Originality Check service.

UNIVERSITY OF TURKU
Department of Computing

FRANS SAUKKO: A Novel Dynamically Truncating Object Pool for Game Loops and Other Use Cases in Rust

Master's thesis, 73 pp.
Computer Science
February 2025

Rust programming language, introduced in 2012, presents a new model of memory management called ownership: each value has an exclusive owner, and values are automatically destroyed when their owner leaves the scope. As adherence to ownership rules is checked during compile-time static analysis, runtime memory safety is guaranteed without sacrificing performance to garbage collection. However, the strict rules of ownership and the rigid memory hierarchy they introduce complicate certain things. Cross-iteration – comparing every element in a collection to every other element in the same collection, possibly mutating their data in the process – is not trivial to implement in Rust. This limitation has made implementing video game loops in Rust generally a complex and difficult task.

This thesis presents a novel collection type, StableList, a simple-to-use and efficient data structure that supports mutable cross-iteration and dynamic truncation, implemented in safe Rust. This thesis introduces its public interface, explains its internal workings, and analyzes its performance against other Rust collections with similar features and purpose through collected benchmark data. As the research data shows, StableList is at least mediocre in its running time for all the essential operations, in the superior subgrouping for cases with highly varying numbers of short-lived elements, and unrivaled for mutable cross-iteration. StableList is a potent data structure for its purpose of powering video game loops and is highly viable for many other use cases, including worker, thread and connection pools.

Keywords: Rust, safe Rust, data structure, algorithm, collection, object pool, pool, arena, video game, game engine, game loop, iteration, cross-iteration, truncation, RAII

Table of contents

1 Introduction....1
2 Background....3
2.1 Video games & game loops....3
2.2 Memory management in Rust....3
2.3 Cross-iteration....6
2.4 ABA problem....6
2.5 Ephemeral elements....7
3 Comparable collections....10
3.1 Vec....13
3.2 Rc<RefCell<T>>....14
3.3 StableVec....16
3.4 Slab....17
3.5 StableGraph....18
3.6 SlotMap....19
3.7 Generational Arena....22
4 StableList public interface....25
4.1 Accessing StableList with StableIndex....25
4.2 Constructors & mutators....26
4.3 Single-element access....27
4.4 Cross-iteration....30
5 Internal implementation of StableList....36
5.1 Access key implementation....36
5.2 Memory layout....39
5.3 StableAccess-powered cross-iteration....46
5.4 Lack of indexing access....48
5.5 Lack of support for idiomatic iteration....52
5.6 Improvements & alternative implementations....56
5.6.1 Generic index types....56
5.6.2 Priority queue–implemented freelist....57
6 Performance comparison....60
6.1 Basic operations....61
6.2 Cross-iteration simulation....65
6.3 Ephemeral data simulation....66
6.4 Performance analysis conclusion....70
7 Conclusion....72
References....74

1 Introduction

Rust programming language that reached its first stable release 1.0 in 2015 [1][2] presents a novel approach to memory management [3]. Whereas many older programming languages, such as C and C++, rely solely on manual memory management [4][5][6][7][8] and garbage collection is popular among more modern languages [4][9], such as Java [10] and C# [11], Rust enforces the memory safety through its *ownership* system reminiscent of the RAII (or ”Resource Acquisition Is Initialization”) design pattern in C++ [7], where each data value in memory is *owned* by some *owner*: local values are owned by their assigned variables, fields are owned by their encasing data structures, and elements are owned by their encasing collection. Any data is removed from memory only when the owner of that data is removed.[12][13]

While Rust’s memory model provides runtime safety, moving appropriate errors to compile time [3][12][13], the strict rules of ownership complicate certain things. This thesis shall demonstrate that of special difficulty is the case of iterating over a collection, comparing every element to every other element in the same collection, possibly mutating their inner data in the process. In this thesis, we refer to this problem that sits at the core of video game engines among many other application domains as *cross-iteration*.

In this thesis, a new collection data type is proposed for powering game loops in the Rust programming language: **`StableList`** provides a simple-to-use public interface for easy insertion, removal, random access, iteration and mutable cross-iteration of elements. Moreover, **`StableList`** features dynamic truncation, where the collection proactively attempts to insert elements into such positions that future truncation is more likely possible – and then truncates it automatically, when the required conditions are met. **`StableList`** is an easy, efficient way of implementing a game object pool [14][15][16] and its associated game loop for video games for Rust programmers, who have previously often had to rely on game engine frameworks even for simple video game experiments. **`StableList`** is also viable for any other use where cross-iteration is needed and/or highly varying amounts of data are inserted and removed frequently, e.g. for implementing different kinds of worker, thread or connection pools.

The rest of the thesis is organized as follows:

In Chapter 2, *Background*, we introduce better the setting and circumstances that led to designing and developing the **`StableList`** data type. We discuss what is required of collection data types to allow for implementing a video game engine in Rust and how Rust’s strict ownership rules affect this endeavor. We formally define cross-iteration and illustrate the difficulty of performing it in Rust. We also define a few other problems that some of the collections discussed in the scope of this thesis suffer from. Finally we

define the concept of ephemeral data, as it both exists in the context of video games but is also used in the benchmarks later in the thesis especially for demonstrating performances of different collection data types when used for certain kinds of pool application domains.

In Chapter 3, *Comparable collections*, we introduce a number of comparable collections that share some subset of design goals and/or application domains with `StableList` – both Rust standard library–provided approaches as well as third-party software libraries. We describe and analyze them in detail, finding their relative pitfalls and advantages in comparison to each other.

In Chapter 4, *StableList public interface*, we give a complete description over the public API, or application programming interface, of `StableList`. We explain in detail how to instantiate an instance of `StableList` and use the basic operations insertion, removal, random access and single iteration as well as the more specific method of cross-iteration. After this chapter, the reader should be able to use `StableList` in their own program.

In Chapter 5, *Internal implementation of StableList*, gives a more technical, deeper insight of how `StableList` internally operates: how the contained elements are arranged in the memory and how insertion, removal, random access and iteration logically work under the surface. We analyze the time complexities of different operations of `StableList`, and we propose a number of further improvements and alternative, parallel implementations for the internal implementation described herein.

In Chapter 6, *Performance comparison*, we analyze data collected from benchmarking the basic operations, insertion, removal, single iteration and random access, from `StableList` and all the comparable collections introduced in Chapter 3. Additionally, we also analyze data collected from a separate cross-iteration benchmark as well as for managing ephemeral data.

In Chapter 7, *Conclusions*, we describe the intended purpose and design goal of `StableList` as a data type aimed towards game loops while at the same time addressing other possible application domains it might be suitable for, judging from its public API, technical details and the benchmarking results. We once more mention the possible improvements to be possibly committed to in future work and enumerate a number of future research questions.

2 Background

In this chapter we study the background of concepts that are discussed in this thesis at large.

In Section 2.1 we define game loops that are at the core of the the research conducted in this thesis. Section 2.2 discusses memory management concepts of safe Rust as well as the definition of ”safe Rust”, especially in the context of this thesis. In Section 2.3 we define cross-iteration and discuss why it is of special research interest in Rust. Section 2.4 focuses on defining and discussing the ABA problem and its different subvariations, and Section 2.5 defines the concept of ephemeral elements that are essential to the benchmark analyzed in Section 6.3.

2.1 Video games & game loops

One way to define video games is that they are essentially pieces of software programs [17] that involve decision making, controlling game objects and pursuing a set goal.[18] Video games are different to many other forms of entertainment, such as film and television, in that they are interactive – the state of the game world changes responding to the input of the player or players of the game.[19][20]

A video game’s internal core software components as a whole are referred to as a *game engine*. The game engine is responsible for running the internal logic of the game. Art assets, game world architectures and the rules of play combine together with the game engine to ultimately result in the actual complete video game experience.[21]

While the game engine is a combination of multiple subsystems, at its core resides the *game loop*. The game loop, in its traditional, most primordial form, comprises three phases: receiving input (usually mouse and keyboard input in the case of computer games), updating the game world data state, and producing output, usually in the form of graphics drawn on the screen and sound played through the audio output.[22][23] The distinction between the video game as a single monolithic whole unit and the game engine proper at its core might not be a clear nor a solid one in all cases [21], while in other cases video games have been built on top of third-party, separately shipped external video game engines [21][23].

2.2 Memory management in Rust

Game development in Rust is especially focused around the use of entity-component-system design pattern or ECS.[24] One reason for this is most likely Rust offering low-level programming capabilities and high

performance [25][26] that translates well into cache-oriented ECS [27]. On the other hand, we shall also reach an observation in the course of this thesis that the more traditional OOP approach revolving around encapsulated game objects stored in linear collections [28] introduces surprising difficulties in Rust. However, the focus of this thesis is not on ECS but specifically on the latter: facilitating writing video game engines and game loops in idiomatic Rust without resorting to ECS.

Rust standard library provides **`Vec`** [29][30][31][32] that is the recommended primary collection for most use cases of storing any kind of data [33][34][35]. A dynamically resizable array [30][32] with random access time complexity of *O(1)*, push-at-end/pop-at-end time complexity of *O(1)*, and insert-anywhere/remove-anywhere time complexity of *O(n)* [29], **`Vec`** is a sequential collection analogous with **`std::vector`** in C++ [36], **`ArrayList`** in Java [37] and **`List`** in C# [38].

In the next chapter we will inspect the capabilities of **`Vec`** to reach the conclusion that it in itself is not sufficient for game loops. For this reason we also inspect a number of custom collection types from third-party libraries – or *crates*, as they are known in Rust nomenclature [39][40][41] – to assess their features and capabilities compared to **`Vec`** in the context of their use in video game engine and game loop implementation. In the course of the rest of this thesis, we will refer to these third-party libraries as *comparable collections*.

Rust also allows the use of raw pointers, in a manner similar to programming languages with manual memory management [42][43], and these can be used for heap-allocated game object management. However, as manual memory management is an infamous source of software errors and security hazards [44][45][46], Rust regards their use as ”unsafe Rust”. What this means in practice is that the use of raw pointers and any other unsafe Rust forsakes many of the safety guarantees Rust commits to, which is at the core design philosophy of Rust as a programming language [25][26].

To understand what *unsafe Rust* is, we have to first define what is *safe Rust*.

First of all safe Rust enforces strong type safety. All types and their compatibilities are checked at compile time static analysis.[3][12][47][48] Secondly Rust does not implement manual memory management, like C and C++, nor garbage collection, like Java or C#.[12][13] Instead, it imposes the following rules regarding *ownership* on all values within the program [12]:

- Each value in Rust has an *owner.*
- There can only be one owner at a time.
- When the owner goes out of scope,
  the value will be dropped.

In practice this creates a setting similar to the RAII design pattern in C++ [7][12][49][50]: data is hierarchically encased as part of their encasing unit, and it will be automatically freed when the encasing unit expires, i.e. goes out of the scope. Compared to C++ RAII, the difference in Rust is that all safe data managed in any way in the program is part of this hierarchical nature – including but not limited to heap-allocated data, individual elements in a collection, temporary values assigned to local variables only existing in the inner scope of a function, and data returned from inside functions.

The ownership of values can change over the course of a program – assigning them usually either moves or copies them, depending on the type, and many values can also be explicitly cloned, which is similar to copying. References to values can also be passed around. These references can either be shared and read-only – a situation also known as an *(immutably) borrowed* value – or unique, exclusive read-and-write – i.e. the value being *mutably borrowed*. To prevent race conditions, Rust mutually exclusively allows only a single mutable borrow or an arbitrary number of read-only borrows to a value at a time.[12][13] The references also have convoluted additional logic and syntax associated with them in the form of *lifetimes* – bounds that Rust compiler demands to be defined on certain situations to guarantee that references will not outlive their source data, categorically preventing dangling references.[51][52] Furthermore, the references can never be null, which together with the Rust's standard-library-provided **`Option`** type (a type of *algebraic data type* or ADT [53]) prevents null pointer exceptions that are a common occurrence in the field of software [52][54][55][56]. Due to null references not existing in Rust, it is also impossible to introduce a double free error.[57]

These ownership rules are at the core of the Rust programming language's philosophy of memory safety.[3] The ownership rules guarantee compile time safety [12][13], i.e. eliminating a whole class of memory errors during the compile-time static analysis instead of encountering them at runtime [12]. This is many times crystallized in Rust community as a general rule that is exclaimed as follows: "If it compiles, it works."

Unsafe Rust is, by definition, the complement of safe Rust. As has been established above, using references as the way of accessing non-owned yields us all these safety guarantees of avoiding memory errors in runtime. However, Rust still gives the option to use raw pointers, if the programmer so wills – but in doing so, these safety guarantees are revoked. As "the Rust book", as *Rust Programming Language* is often referred to as, itself states "**`unsafe`** does not mean the code inside the block is necessarily dangerous or that it will definitely have memory safety problems: the intent is that as the programmer, you'll ensure the code inside an unsafe block will access memory in a valid way." The use of raw pointers is the single primary thing that constitutes, for the intents and relevance of this thesis, unsafe Rust.[58][59]

2.3 Cross-iteration

A common problem connected to safe Rust is that it is somewhat complicated for collection types in Rust to implement cross-iteration capabilities. In this thesis we define *cross-iteration* as doubly iterating the same collection in such a way that for every element $e_i \in E$ in the collection, every other element $e_j \in E \setminus \{e_i\}$ in the collection is also iterated through. We also require that in the course of iteration it is possible to mutate both e_i and e_j at will. For the set E of all the elements $e \in E$ in a collection, we can formally define cross-iteration as the Cartesian product [60] of the set E with itself, excluding each element paired with itself: $E \times E \setminus I$, where $I = \{(e,e) \mid e \in E\}$. This can also be expressed as $E^2 \setminus I$ [60].

Cross-iteration is an especially common occurrence in video games, e.g. in the cases of naive implementations of collision detection [61][62], finding the nearest neighbor or comparing data between all relevant elements – which could only be possible to access as a subset of all existing elements. The requirement to also allow mutating the elements, instead of only inspecting, is essential, as during the iteration, we *update* the game objects [21][22], which typically involves altering the parameters of the game objects. The spatial position of each game object is updated, based on their current velocity, on every update. Collision detection likewise involves iteration, and collisions detected between two game objects may have various mutating consequences for the game objects involved, including but not limited to moving them away from their point of collision with each other or reducing their health values. Section 4.4 demonstrates actual code examples of some uses of cross-iteration, underlining its importance for game loops.

While it is trivial to implement in manual memory management languages [4], such as C or C++ [5][6][7], or garbage-collected managed memory languages [9], such as Java and C# [10][11], the Rust memory model with its strict adherence to its concepts of ownership and lifetimes at times makes it difficult to mutate two different parts of the same collection[1], as they are considered to belong to the same *borrow unit* that is then governed as a whole by the restriction of one mutable access at any one time.

2.4 ABA problem

Another common problem of naive, simple implementations of collection types is the ABA problem. The letter combination ABA is not an acronym but exemplifies the situation where the value of a certain piece of data may change between subsequent accesses from A to B and potentially back to A [63]. While this term is often used in case of multithreaded settings, where the same data can be read and written from multiple

1 In Rust, mutating even a single element within a collection implies mutating the whole collection. Unique mutable access to element `e` stored at index `i` in the collection `v` of type `Vec` is acquired through the method call `let e = v.get_mut(i)`. A mutating method, this call requires unique mutable access to `v` to pass compile-time static analysis. The variable `e` then continues to hold this unique mutable access to `v` as long as it exists, preventing any other access to `v` within the same scope – including accessing any other element (mutably or immutably) within `v` for as long as `e` exists.

different sources with no guarantee over the order of these separate instances of access [63], because of the the strict memory model of Rust, the same term is also used to discuss problems in linear, non-asynchronous mutations of single-thread-accessed collections. In this meaning it is also mentioned in the documentations of the comparable collection crates *slotmap* [64] and *generational-arena* [65].

At its core the ABA problem is a phenomenon where random-accessing a collection using an index i returns an element e' that is not the same element e where the index i was pointing at the time of insertion. This can happen through intermittent removal – a case that we shall dub the *remove-reinsert* subcase of ABA: an element e_i is inserted at memory position i and later removed, after which the element e_i' is inserted at the same memory position. Attempting to random-access the already removed element e_i using its index i will instead return e_i'. It can also happen by means of element space shifting, another situation that we shall dub the *shift-N* subcase: an element e_i is inserted at the memory position i, but for some reason the collection later either moves the element at i to another memory position j or shifts any number N of other, unrelated elements in the same memory space to another position. If now the index i is used to random-access the collection, another element e_i' (or no element at all) is returned instead – element e_i could only be accessed by using the index j.

In other words, a collection suffering from the ABA problem cannot guarantee that its indices are stable over all public API–allowed operations on the collection. To overcome the ABA problem, a collection has to be able to detect whether or not the element e retrieved from a memory position $i_{key(e)}$ within the collection is the same element e that an encapsulated access key *key(e)* used to access it was originally associated with. If not, i.e. the memory position $i_{key(e)}$ contains $e' \neq e$, a missing value Ø should be returned instead of the incorrect e'.

It is worthy of noting that if the collection only suffers from the *remove-reinsert* subcase of ABA, it is still possible to guard against this behavior, as this kind of index invalidation only concerns one known element at the exact position of access. These kinds of countermechanisms to check against ABA can be implemented on the element side.

The *shift-N* subcase of ABA cannot, however, be reliably guarded against, as every time there occurs a shift of $N \geq 1$ other elements in the memory space, all of the indices of these N elements are invalidated, and the software might have stored some of these now-invalidated indices for subsequent use elsewhere in the program architecture, the invalid access through these indices only happening at an unspecified, later point in time.

2.5 Ephemeral elements

One additional area of interest in assessing the comparable collections is how well do they fare with dynamic

amounts of contained elements. Most of the comparable collections reuse the space that has been vacated by prior removals of elements, but this is not, in general, an absolute rule. There exist collections designed so that vacated elements are never again reinserted into, which, if given sufficiently long period of time, leads to the memory requirements growing boundlessly. All of the comparable collections also slow down in iteration times as more elements are inserted, as this is a natural outcome of increased computation and increased number of memory reads and/or writes. However, within the comparable collections there is a subset that recover their near-original iteration time when the amount of elements is again reduced to lower numbers, while the clear majority does not show any positive indication in their iteration times after such an event.

In Section 6.3 we measure this capability for iteration speed recovery by simulating *ephemeral elements*, as we shall call the phenomenon in this thesis. An ephemeral element here refers to any such varying data that is are repeatedly inserted into the collection and that have more or less clearly definable upper bound associated with them in relation to their period of existence or lifetime.

In video games, a common example of such ephemeral elements are particles used for visual effects: usually great numbers of these are created or ”spawned” in burst – either at a single instant (explosion), over a limited period of time (blood spraying out) or at a regular pace when they are in the field of vision (fire) – and each of the particles only last for a duration of time defined at spawning, until they expire or ”decay”. [66] Another example are visual projectiles that are created when an attack is initiated and then destroyed at the end of their trajectory or when they are regarded as having hit their target or terrain after missing their intended target.

Even enemies in many video games can be regarded as ephemeral elements, if they are spawned continuously but each of them is guaranteed to be eventually destroyed. A trivial example of this is in scrolling shooters where enemies fly through the view, spawning at one edge and being automatically destroyed when they reach the other edge, such as *Gradius* [67] or *Gun.Smoke* [68]. An implicit upper limit is sometimes indirectly enforced by the gameplay mechanism that the game does not allow progressing to the next view, even within the same continuous gameplay area, until all the enemies in the current view are destroyed. This is the case in such games as *River City Ransom* [69], *Batman Returns* [70] and *Battletoads* [71].

It has to be kept in mind that some games might at first seem as if featuring enemy characters as ephemeral elements, even when this is not the case. In many modern bullet heaven games, as popularized by the pioneer of the genre Vampire Survivors [72][73], new enemies may keep spawning only until a maximum number of enemies at one time present in the view is reached. However, these enemies cannot be regarded as truly ephemeral, as an enemy $enemy_t$ created at a point of time t cannot be guaranteed to have been destroyed at any point of time in the future t': depending on player behavior, $enemy_t$ might evade destruction even while a

sufficient number of other enemies are destroyed, leading to new enemies being spawned, while *enemy*$_t$ survives. Thus no definite upper bound for the lifetime of *enemy*$_t$ can be defined, and it can technically exist indefinitely (here of course meaning until the game session ends in the player either facing defeat or voluntarily exiting).

While favoring iteration speed recovery may be somewhat tangential in the case of video games, as they are expected be designed so that they run with full or at least tolerable frame update frequency even when the game world is populated by a maximal or near-maximal number of game objects [74][75], these kinds of collections may find use outside of video games in other application domains. As the collections that we regard in this thesis as suitable for managing game objects and game loops can be regarded as game object *pools* [14][15][16] – also often referred by the near-synonymical term *arena* [65][76][77] – they are usually well fit for other pooling solutions as well, i.e. when the same allocated memory space is intended to indefinitely be experiencing removals and reinsertions of ephemeral or near-ephemeral objects. This makes the collections that support iteration speed recovery much more suitable for use in, for example, different kinds of worker pools, thread pools or connection pools [78][79][80], as their iteration speed won’t be permanently hampered by temporarily high element counts that have occurred as isolated instances in the past.

3 Comparable collections

To understand the criteria by which we judge the comparable collections, we first have to enumerate our design goals for the **`StableList`** collection that we are comparing against. **`StableList`** is originally purposed for being used as an object pool as part of game loops in video game engines. This sets its design criteria to the following points:

1. Supports simple insertion, removal and random access.
2. In addition to typical single iteration also supports mutable cross-iteration.
3. Does not suffer from the ABA problem.
4. Reuses space vacated by removed elements for new inserted elements.
5. Is implemented in safe Rust.

In addition to this **`StableList`** was later implemented to also proactively fill the elements to vacated memory positions in such a way that subsequent truncation becomes more likely, and to automatically truncate itself when there is a sufficient amount of unused space at the end of the memory space. We shall add this additional later feature as the sixth criterion:

6. Automatically truncates in size when reaching or nearing full vacancy.

As an additional comparable feature we can include sortability – i.e. if the collection allows sorting its elements. As sorting by definition moves the elements into different memory positions in the memory, we can easily come to the conclusion that directly implies ABA problem. Sortability is not a feature that is sought-after in object pools, and it is clear from this that their desired design is actually in direct violation of it. It is also why pools, as a rule, lack this feature. However, it is a curiosity to assess in this comparison, and has been thus included within the comparison as criterion #0:

0. Provides a mechanism to sort the contained elements.

It is to be reminded that while the collections can fulfill the criterion 3, not suffering from the ABA problem, elements that do not satisfy this criterion can be subcategorized into two categories of severity of their ABA problem.

The more severe subcategory, *shift-N* are the collections that may invalidate $N \geq 1$ number of elements, usually by a large region of shifting elements on removal. Collections suffering from *shift-N* are virtually

unusable for game object lists, as moving elements to a different memory position within the collection results in all indices existing at that point being invalidated, even those that have been stored elsewhere in the program logic, so the index invalidation might become apparent even only much in delay.

The less severe subcategory, *remove-reinsert*, only invalidate a maximum of 1 element, if an element e_i is removed at the index i and another element e'_i is later inserted at the same index i, and $e_i \neq e'_i$. *remove-reinsert* can be prevented by identity checking outside of the collection logic (which of course imposes extra work compared to if this were provided by the collection itself). The simplest way of implementing this is an identifier field for each element, e.g. as a running number of unsized integer type that is incremented every time a new element is instantiated and inserted into the collection. The indices returned from the collection at insertion would also be reinforced with the identifier data of their associated elements, so that it is possible to check if the identifier of the index matches with the identifier of the collection at that particular index. Collections suffering from *remove-reinsert* are still viable possibilities for implementing a game object list and a game loop, if this kind of custom ABA-countering solution is taken care of.

In this chapter we compare eight standard-library or third-party collections that each share a considerable subset of features as well as the design purpose and/or use case with **`StableList`**. The design criteria enumerated above and how the discussed collections satisfy them are collected into Table 3.2, for easier comparison with one another and with **`StableList`**.

The third-party collections here are from five crates publicly accessible at *crates.io*, the official Rust programming library repository, as well as being open-source and hosted publicly on GitHub. Each of these libraries are well-established, as can be witnessed from Table 3.1.

Table 3.1 *Rust crate repository downloads and GitHub recommendations of comparable collections* [81–90]

crate name	**relevant collections**	**crates.io downloads** (rounded to 1000s)	**GitHub stars** (rounded to 100s)
stable-vec	StableVec	191 000	20
slab	Slab	250 172 000	728
slotmap	SlotMap HopSlotMap DenseSlotMap	18 137 000	1 200
generational-arena	Arena	5 554 000	674
petgraph	StableGraph	126 984 000	3 100

Table 3.2 *Features of comparable collections and StableList*

	insertion, removal, random **(1)**	*space-reusing* **(2)**	*ABA-immune* **(3)**	*mutable cross-iteration* **(4)**	*safe Rust* **(5)**	*auto-truncating* **(6)**	*sortable* **(0)**
Vec<T>	✓	✓	shift-N, rM-rI	*			✓
Vec<Rc< RefCell<T>>>	✓	✓	✓	✓			
StableVec	✓		✓				
Slab	✓	✓	rM-rI	✓	§		
StableGraph	✓	✓	rM-rI	✓			
SlotMap	✓	✓	✓	✓			
HopSlotMap	✓	✓	✓	✓		†	
DenseSlotMap	✓	✓	✓	✓		‡	
Arena	✓	✓	✓	✓	✓		
StableList	✓	✓	✓	✓	✓	✓	

shift-N — Severe ABA problem, subcase *shift-N*. Virtually prevents using the collection for game loops.

rM-rI — Mild ABA problem, subcase *remove-reinsert*. Invalidates only a maximum of one index at insertion, never shifting multiple elements in the element memory space. The problem can be countered by element-side guard mechanisms.

* — `Vec<T>` provides facilities to custom-implement mutable cross-iteration, but its public interface does not readily allow it in a manner that would be effortlessly replicable or immune to frequent developer mistakes.

§ — `Slab` includes a small number of auxiliary, non-essential unsafe methods. They are not required for typical usage of the collection. Apart from these few, no other methods feature any unsafe code even in their internal implementation.

† — `HopSlotMap` has its iteration time linear only to the current number of elements present in the collection, instead of the number of memory positions in use. However, the elements space does not actually truncate automatically in any case.

‡ — `DenseSlotMap` accesses its elements through two-layer indirection. The element space is densely occupied, which allows stopping iteration at the first vacant position, as it can never be followed in sequence by any occupied positions. However, the index space that is used to access the actual elements space still does not ever automatically truncate.

3.1 Vec

The **`Vec`** or vector is a resizable linear, sequential collection in the Rust standard library [29][30][31][32] akin to **`std::vector`** in C++ [36], **`ArrayList`** in Java [37] or **`List`** in C# [38].

`Vec` is the go-to basic collection in Rust. For most problems in most application domains, **`Vec`** provides the most or near-most efficient solution and is thus the recommended collection as a starting point.[33][34][35]

Being an essential building block for most of Rust-written infrastructure, it should come as no surprise that **`Vec`** is also heavily employed in the implementation of all the collections discussed in this chapter and whole thesis: all of these that are labeled space-reusing in the above table implement a *freelist* [14][91] using a **`Vec`**. What this technically means is a logical stack, implemented as a singly-linked list in a **`Vec`**, where each vacant memory position, i.e. that has had its element removed, points to the chronologically immediately previous vacant memory position. Thus, when inserting new elements, they are first inserted into previously vacated memory positions in chronologically reverse order, before appending them into new memory positions at the tail of the collcetion.

However, as a trivial sequential collection, offering only the most trivial of features, **`Vec`** only fulfills the criteria 1 and 2: it allows easy insertion, removal and random access, and it reuses space freed by removed elements. The latter is a natural consequence of the three different approaches that it supports for element removal: i) popping the last element, which can be regarded as a subcase of either of the following two approaches, ii) removing any element but moving all the elements following the removed element one place before, iii) swap-removing any element, which consists of first swapping the element to be removed with the last element, after which the now-last to-be-removed element is popped.[29] None of the three methods leave any non-reusable idle space into the memory structure of the **`Vec`**.[92]

`Vec` does readily provide facilities for fulfilling the criterion 4, mutable cross-iteration, as well. To understand the nature of the limitation that the criterion 4 tries to surpass, we have to understand Rust's limitations on mutability and its rules for *borrowing*, as the concept of passing references to data – be they immutable or mutable – is called in Rust.

`Vec` readily provides different split methods that allow a single mutable **`Vec`**, or a reference to one, to be split into two mutable references. Using these methods, we are able to manifest one mutable **`Vec`** into the observed pivot element x, the elements preceding it $\{ a_1, a_2, a_3, \ldots, a_n \}$ and the elements following it $\{ b_1, b_2, b_3, \ldots, b_n \}$ – each of which we can also mutate during iteration. However, doubly splitting Rust vectors into these three separate partitions in the described manner requires custom code for every site of use, and as the index

shift of post-pivot elements has to be taken into account, it is in no way made low-effort or mistake-free by the `Vec` public interface. It is exactly this lack of a robust API for this functionality that led into implementing the number of layers of abstraction that now is `StableList`. It is also because of this lack that we do not regard `Vec` meeting the criterion 4 for the purposes of this research.

Using a pure `Vec` also suffers from the ABA problem, contradicting the criterion 3:

Let vector V contain n elements. Let there also exist the element e_m, where $m < n$: $\{ e_1, e_2, \ldots, e_m, \ldots, e_n \}$. Random-accessing elements is done through each of their corresponding indices: e_1 can be found at index 1, e_2 at index 2, and e_m at index m. However, in the case of `Vec`, these indices are not stable over transformations of the collection: if we are to remove any element at an index i, where $i < m$, the element e_m is no longer found at index m. If we removed the element at index i, transposing all following elements to one position prior, the element e_m now resides at index $m - 1$. If we swap-removed e_i, the element e_n now resides at index i. If we pop-remove the last element e_n of the collection, accessing an element at index n is an invalid operation until a new element e_n' is pushed into the collection – after which it returns the wrong, newly pushed element e_n'.

`Vec` is internally implemented with significant amounts of unsafe code. This is to be expected, however, for an essential, close-to-architecture data structure that is part of the standard library of the programming language. Its status as such also means that it has been thoroughly tested against errors and undefined behavior during development and test iterations and continues to be constantly tested in real use cases by the community using it.

As `Vec` satisfies only a bare subset of two of the six criteria (three with reservations) – especially failing to offer a solution for the ABA problem that is essentially intolerable in a fully operational game loop – analysis of performance of `Vec` is not included in the benchmark analysis.

Out of all the comparable collections, `Vec` is the only one that provides a mechanism to sort its contained elements.

3.2 Rc<RefCell<T>>

Another approach that is more advanced yet still readily supported by the Rust standard library is using a standard `Vec` to store the elements by first wrapping the elements within the composite data structure `Rc<RefCell<T>>` – a pattern known in Rust as *interior mutability*.[93][94]

The above notation can be interpreted into natural language in the following way: the type **T**, that is the type of the elements we want to store in the collection, is wrapped within **RefCell**, an encapsulation that allows its internal data be mutated and for the borrow rules of this mutation to be checked only in runtime instead of compile time. **RefCell** itself is then wrapped into **Rc**, a reference-counting smart pointer – a pointer that can have an arbitrary number of multiple owners.[93][95]

What this multi-layered abstraction manages to achieve is very similar to the way objects are handled in Java and (safe) C#. Declaring objects with the type **Rc<RefCell<T>>** instead of plain **T**, firstly, makes all the initialized objects be individually created in the heap instead of adjacently initialized values. Secondly, it essentially allows them to be mutated even within an immutable collection, practically circumventing the restrictive compile-time borrow rules of Rust in the vast majority of use cases. This allows practically simulating the idioms of object management standard to Java and C# in the context of Rust.

Changing the definition of elements within **Vec<T>** from **T** to **Rc<RefCell<T>>**, we are able to upgrade from criteria 1–3 met by **Vec<T>** to 1–4 met by **Vec<Rc<RefCell<T>>>**: the interior mutability in itself allows mutating any element at any one time (criterion #4), while **Rc** as a smart pointer allows for referring to the data contained within through the pointer itself, propagating shared access by supporting cloning instances of **Rc**. This circumvents the need to refer to individual elements by their index within the **Vec**, and as any instance of **Rc** on inspection (benefiting from Rust's support for algebraic option types) returns either a **Some(T)** value for still existing contained values or a **None** value for already outlived, non-existing values, there is no risk of accessing an element ever returning an incorrect element, thus fully meeting the criterion #3 – undoing the ABA problem.

What are the drawbacks and tradeoffs of **Rc<RefCell<T>>**, then?

Firstly, our instances, as said, will be individually initialized in the heap and managed as references instead of being initialized as pure values sequentially and adjacent to each other. This takes it toll on all operations that access the inner data of elements; this is especially visible in the insertion and random access running times in comparison to the benchmark results: insertion requires allocation, while random access requires two layers of indirection: first to access the heap-allocated **Vec** element space, then to access the heap-allocated **Rc** payload.

Secondly, as by definition of **RefCell**, we are switching from checking the borrow rules in compile time to checking them in runtime. In doing this we redeem the benefit of complete compile time checking that is central to Rust's broader philosophy. In other words, whereas in compile time checking we can be certain that no errors arise during running of the software, by including interior mutability logic into our codebase we revoke this guarantee.

3.3 StableVec

Outside of the approaches provided by the standard library, community-developed Rust libraries offer more advanced and robust solutions for achieving the same or similar outcomes that `StableList` is also pursuing.

A very primitive improvement on standard `Vec` is the crate *stable_vec* with its data structure `StableVec`. `StableVec` solves the ABA problem associated with `Vec<T>`.[96]

`StableVec` solves the ABA problem by implementing an alternative logic for removing an element. While `Vec` removes the element at the given index `i` and moves all the following elements one position to the left, `StableVec` instead labels any removed elements as inaccessible for the future. Thus all indices to any elements in any `StableVec` will always remain stable – elements are guaranteed to not change their position within `StableVec`. In case of trying to access a removed element, `StableVec` returns the empty variant of the aforementioned option type, `None`, which signifies to the accessor that the element in question no longer exists and that the access operation was invalid due to this.

Logically speaking this is analogical to implementing a `Vec<Option<T>>`, with potential performance gains from using unsafe code in its internal implementation. On insertion, instead of simple data `x` of type `T`, option-wrapped data `Some(x)` would be inserted that could also be trivially accessed at its index. On removal, `Some(x)` at the given index would be transformed into `None`. Thus the removed data would be rendered inaccessible and its index within `StableVec` non-reusable. `StableVec`, as a third-party crate–provided abstraction, simply abstracts similar behavior to this to be used readily out of the box.

As is evident from analyzing its primitive removal strategy and is also openly proclaimed on the official crate page of `StableVec`, the approach "has the very obvious disadvantage that deleted objects (so called empty slots) just waste space" [96]. A lesser negative aspect is that, as said, `StableVec` uses internal unsafe code to achieve higher performance than a simple `Vec<Option<T>>` would be able to reach. While unsafe code is not uncommon in both community-implemented and standard libraries and is often needed for achieving top performance, it needs to be thoroughly tested to be fairly certain for all instances of undefined behavior or runtime errors to be eliminated. The creators of `StableVec` are not ready to completely positively state that in their case there are no possible ill effects that could arise as a consequence of the unsafe code use: "[O]f course it cannot be guaranteed this crate is perfectly safe."[96]

`StableVec` circumvents the ABA problem, meeting the criterion #3. Its API supports simple insert, removal, and random access (through indices) operations, satisfying the criterion #1 as well. However, by over time

increasingly wasting memory space previously occupied by now-removed elements, it fails to meet the criterion #2. Moreover it doesn't implement any kinds of split methods that **`Vec`**, for example, does, making mutable cross-iteration categorically impossible and failing to satisfy criterion #2.

Its inability to meet these two criteria makes **`StableVec`** an especially unfit choice for implementing game object collections and game loops. It is because of this that **`StableVec`** is not included in the comparison benchmark, as is the case with **`Vec`**.

3.4 Slab

Another Rust crate by the title *slab*, built around the correspondingly named main data structure **`Slab`**, represents an alternative, parallel development to **`StableVec`** over the simple basis the standard **`Vec`** provides. [97]

Instead of labeling removed indices as permanently unusable, **`Slab`** maintains a *freelist* [91] – a stack of indices that have been removed and that are thus open for reinsertion. When elements are removed from a **`Slab`** collection, the removed index is pushed into the freelist. Accordingly, when a new element is inserted into the collection, the most recently removed index is popped out of the stack-implemented freelist, and the new element is inserted into the memory position indicated by this index. If the freelist is empty, the new element is appended at the end of the collection, as is always done in the cases of **`Vec<T>`** or **`StableVec<T>`**.

The concrete benefit of this approach is that **`Slab`** effectively reuses the space left vacant by removed elements. Moreover, compared to the standard **`Vec`** that also does so, **`Slab`** exhibits far more optimal performance when removing non-terminal elements from the collection (if not considering the reordering-forcing swap-remove method of **`Vec`**), as its strategy for removal does not involve shifting potentially large amounts of data in memory but a simple data write at one position in memory to update the freelist.

However, whereas **`Slab`** thusly satisfies the criteria 1–2, similarly to **`Vec`**, it suffers from the ABA problem just like **`Vec`** – even though at a comparably lesser measure. While **`Vec`** necessarily invalidates at least one index, potentially all subsequent indices, when performing a non-terminal remove, regardless of the method of removal being the shift-remove or the swap-remove, **`Slab`** only potentially invalidates indices on reinsertion, as the following example demonstrates:

Let there be element e_i at index i. On removal the index i is left vacant, and accessing it through an option-typed return value is possible without it being regarded as an invalid operation. However, if we are to insert a new element and it is this index's turn in the remove list, we will now find a different element e_i' at the same

index *i*. This non-identical new element e_i' could now be accessed through the same index as the old, original element e_i, and there are no built-in mechanisms to guard against this – nor even signal its occurrence.

This comparison makes it evident that **`StableVec`** and **`Slab`** are effectively complements in the way they each have improved upon the standard **`Vec`**: whereas **`StableVec`** prioritizes solving the ABA problem that **`Slab`** fails to completely remedy, **`Slab`** seeks more efficient way of dealing with reusing the space left vacant by removed data that **`StableVec`** chooses to entirely ignore.

However, as **`Slab`** only features ABA problem due to unguarded intermittent remove-reinsertion, it is possible to use this collection even for game loops, if ABA problem is countered on the element side, for example, by manually maintaining element identifiers. This is also why **`Slab`**, unlike **`Vec`** and **`StableVec`**, is included in the benchmark comparison in this thesis, together with the fact that it effectively meets the criterion #4 through a specialized method implemented in its API that allows retrieving mutable references to two elements at the same time, allowing mutable cross-iteration out of the box. While **`Slab`** could not be employed in game loop without additional custom work due to partially existent ABA problem, the benchmark data provided by comparing against raw **`Slab`** gives interesting frame of reference in what kind of performance margins have been introduced by more complex collections, including **`StableList`**.

As a further difference to **`StableVec`**, **`Slab`** does not include unsafe code in the implementation of its essential methods and mechanisms. There are a number of implemented methods that require even the caller to declare the calling point as unsafe, but these optional methods only provide additional optimization and are completely voluntary to use, not required for typical operation of the collection. The basic insertion, removal, access and even cross-iteration-supporting methods both implement a safe public API and are internally implemented using safe-only code. **`Slab`** is thusly regarded as meeting the criterion #5 as well, as for the purpose of this research.

3.5 StableGraph

The crate *petgraph* is mainly intended for modeling graph topologies and architectures [98], but its collection **`StableGraph`** is well suitable for representing game object lists and game loops as well – and thusly also for being included in the comparison with other collections in this thesis. This can be achieved by entirely ignoring such provided features as edges or directionality and simply managing game object instances as nodes, with node indices acting as internal references from one game object to another.

Of course one is free to represent references held by elements to other elements as edges, but this is excess in the sense that a node index in itself is sufficient for this, and introducing and maintaining edges impose an additional memory and computation cost without providing any added value, at least in normal cases, where

the references between elements do not have any edge weight associated to them. For this reason, in the scope of this thesis we shall not discuss edges at all.

Internally, **`StableGraph`** wraps within itself a more primitive data structure from within the same crate, simply named **`Graph`**. **`Graph`** is a simple collection that stores the nodes in a **`Vec`**. It handles element removals using the swap-remove strategy of Vec, thus always invalidating indices on removal of any non-terminal elements. **`StableGraph`**, on the other hand, and as is evident in its name, is a stable extension of the **`Graph`**, keeping indices stable and guaranteeing no index invalidation takes place under any circumstances. This is done using a freelist in the same manner as **`Slab`**.

`StableGraph` meets the criteria 1–3. Identically to **`Slab`**, it does not completely undo the ABA problem, but its guarantee of invalidating a maximum of one memory position only on reinsertion allows countering the effect through element side–implemented mechanisms. **`StableGraph`** allows mutable cross-iteration through its method **`fn index_twice_mut(…)`** that corresponds to the similar methods **`fn get2_mut(…)`** and **`fn get_disjoint_mut(…)`** in the other data structures discussed earlier, but it is the said method that is not implemented in safe Rust, not satisfying the criterion #5.

However, **`StableGraph`** random-accesses its individual nodes through a bare unsized integer **`usize`** as an index. As such, it has no means of checking that a random-accessed element is the very same that the index originally referred to or a different element later inserted at the same memory position, so ABA problem may well occur while using it. It has still been included in the comparison between the collections in this thesis on the same basis as **`Slab`** that possesses the same shortcoming.

`StableGraph` is also inferior in comparison to all other collections presented here in that, by design choice, preallocating space for *n* elements in fact also initializes *n* vacant memory positions. This has the concrete consequence that an empty **`StableGraph`** preallocated with *n* elements takes as long to iterate as a **`StableGraph`** that has had *n* elements inserted and then removed. Other collections presented here have iteration running times identical when they are empty regardless of their preallocated capacity, only initializing iterable memory positions when actual insertions occur. This is clearly visible in the benchmark analysis in the Chapter 6.

3.6 SlotMap

The Rust crate by the name *slotmap* offers a set of more comprehensive solution. In fact, the crate contains three different data structures to be used as collections: the **`SlotMap`** proper and its two alternative variants **`HopSlotMap`** and **`DenseSlotMap`**. All three collections fulfill the criteria 1–4. The implication that the collections within *slotmap* act as a union of features from **`StableVec`** and **`Slab`**, in a manner similar to **`Slab`**,

is clear from the crate's official documentation, where the question "Why not index a **`Vec`**, or use **`slab`**, **`stable-vec`**, etc?" is answered thus [99]:

> Those solutions either can not reclaim memory from
> deleted elements or suffer from the ABA problem.

As for how the different slotmap implementations achieve this combination of effects, the key mechanism is that when inserting data into the collection, hence creating a new element, an unique key is returned at the insertion point. These keys are the only way to access – mutably or immutably – the elements contain in whichever type of slotmap is being used. If an element is removed from the collection and a new element is later inserted into the same memory position, there is still no risk of invalid access, as the collection will detect the mismatch between the obsolete key associated with the now-removed element and the new element now residing at that memory location – and will signal this by returning an empty option type, **`None`**.

The access keys are implemented as **`(data, version)`** pairs. When inserting an element, the data is added to the collection as the first member of the **`(data, version)`** tuple. The second member of the tuple is a running integer $v = [0, 2^{31})$ that is set to 0 at first insertion to this memory position, incremented by 1 every time there is a reuse of the slot due to reinsertion into a previously vacated slot and that will be reset to 0 after 2^{31} insertions.

When attempting to access data using a key, the field **`version`** of the data–version pair is checked, and only if the version of the key matches the version of the present data, the data is returned in an option type **`Some(data)`**.

Due to the version being indicated as a running number within the range $[0, 2^{31})$, there is a chance of version collision where a key created 2^{31} insertions before could technically be used to retrieve data inserted 2^{31} removal-insertions later. However, this amount is a notoriously high number, exceeding 2 billion, so firstly it is very unlikely to reach in most use cases – including most video game architectures – and secondly frames such a long timespan that it is unlikely that keys that old would be still stored elsewhere in the architecture for later access. It is, however, important to be aware that it is still not completely impossible for them to occur, even though "[i]t is incredibly unlikely".[99]

As a matter of fact, the access key–based approach is opaque in regard to the indices pointing to the elements and entirely hides the underlying architecture in general. This is also useful for the multiple different implementations within the *slotmap* crate in that they share a similar interface but exhibit very different behavior under the hood.

`HopSlotMap` is the first of specialized variants of **`SlotMap`**. It has built-in encapsulated mechanism that tracks contiguous ranges of vacant element memory positions, allowing them to be completely skipped during iteration. This naturally speeds up the iteration. However, checking and updating this information on every insertion and removal comes with a tradeoff, and the official documentation of the crate estimates “that insertion and removal is roughly twice as slow” but “[r]andom access is the same speed [as **`SlotMap`**]”.[99]

The contiguous vacant range tracking mechanism utilized by **`HopSlotMap`** is not in the very focus of this thesis, but a similar feature is discussed in Section 5.6.2 for **`StableList`**. Briefly described, **`HopSlotMap`** checks on removal of an element at index i if there already resides a vacant element at the indices $i - 1$ and $i + 1$, and, according to the situation, appends index i to the preceding empty range, prepends index i to the following vacant range, merges these surrounding vacant ranges together or creates a new single-index vacant range at index i. In iteration, when encountering a vacant range, the index can be immediately shifted to $end + 1$, where end is the tracked end index of the vacant range.[100]

As with **`HopSlotMap`**, the goal of the yet another specialization, **`DenseSlotMap`**, is to improve the iteration speed. However, instead of detecting and skipping over vacant ranges of memory positions, **`DenseSlotMap`** offers performance boost by placing all the elements in the element memory space densely together – with no intervening vacant memory positions between any occupied memory positions at any point.

This density of elements in memory is achieved by using a mechanism already present in the standard **`Vec`**: non-terminal elements are always swap-removed so that the last element is swapped on their place, after which this newly terminal element can now be safely popped from the end.

While this in its plain form results in the ABA problem, as discussed when describing the standard **`Vec`**, **`DenseSlotMap`** nullifies this issue by introducing another level of indirection for accessing the data. Within the data structure, the access key is used to point at a memory position i within the first, so-called *outer* list, from which the access position j of the actual data within the second, so-called *inner* list that represents the contiguous data, is retrieved. In other words, when using an access key $key(e)$, the element e that it is associated with is found at the memory position $inner[outer[index_{key(e)}]]$.

Likewise, element e can only be removed from a **`DenseSlotMap`** using its access key $key(e)$. The *inner* index $i_{inner,e}$ where e is stored is the value $i_{inner,e} = outer[index_{key(e)}]$. The element e at $inner[i_{inner,e}]$ is swapped with the last element e' in *inner*. Now e' resides at $inner[i_{inner,e}]$, while the new last element of *inner* is e. By removing the last element of *inner*, the element e is effectively removed. While this swap-remove does not shift any other elements, the inner indices stored in *outer* still need to be updated for both of the swapped elements e and e'. For the removed e the link is simply invalidated: $outer[index_{key(e)}] \leftarrow Ø$ (this is actually done through version updating in **`DenseSlotMap`**, but our example here illustrates the superficially similar outcome of a

non-existent value Ø to be returned on subsequent accesses). For e' residing at $inner[i_{inner,e}]$ we set $outer[outer_index_{e'}] \leftarrow i_{inner,e}$ (this is made possible by outer and inner lists reciprocally storing each other's indices in a doubly linked manner). As a result the element e has been removed, the outer list entry for the swapped e' correctly points to the new memory position, and the inner list upholds its dense memory structure.[101]

Due to the two levels of indirection, even with the additional level only containing indices into the other structure, the memory requirements are slightly higher and random access is slower than either the standard **`SlotMap`** or even a **`HopSlotMap`** due to having to traverse through two separate heap-stored memory locations. However, the iteration times of **`DenseSlotMap`** are even faster than that of **`HopSlotMap`**, "as fast as a normal **`Vec`**", as there is no excess overhead of any kind when traversing the tightly-packed, strictly adjacently positioned elements and iteration is ended immediately after visiting the last present element, with no time wasted checking vacant memory positions.[99]

Apparently the three different variants contained within the crate *slotmap* were developed with largely the same criteria as **`StableList`** – an observation we will judge even better later in the this thesis as we investigate the latter's mechanisms, inner workings and use implications. In the scope of this thesis **`SlotMap`** ultimately comes closest to the utility of **`StableList`** as a collection of choice for implementing game object collections and game loops.

From our study into the mechanisms at work with the different slotmaps we can conclude that all its variants meet the criterion #2 about reusing space left vacant by removed elements and criterion #3 about counteracting the ABA problem. All the slotmaps also provide the same common API for inserting new elements and accessing and removing existing elements using access keys procured at insertion, satisfying the criterion #1.

Furthermore, the API provides the method **`fn get_disjoint_mut(keys)`** that allows as its arguments n number of access keys, returning an equal number of mutable references into the elements pointed to by these keys. Thusly the slotmaps also offer robust support for mutable cross-iteration, satisfying the criterion #4.

All slotmap implementations, just as **`StableVec`** did, rely on unsafe code to allow for better, optimized performance.[99] Thusly none of the slotmaps satisfy the criterion #5.

3.7 Generational Arena

The Rust crate *generational_arena* introduces the collection data structure **`Arena`**.[65]

The documentation page for *generational_arena* claims the reasons for developing the solution as follows [65]:

> Imagine you are working with a graph and you want to add and delete individual nodes at a time, or you are writing a game and its world consists of many inter-referencing objects with dynamic lifetimes that depend on user input. These are situations where matching Rust's ownership and lifetime rules can get tricky.
>
> It doesn't make sense to use shared ownership with interior mutability (i.e. `Rc<RefCell<T>>` or `Arc<Mutex<T>>`) nor borrowed references (i.e. `&'a T` or `&'a mut T`) for structures. The cycles rule out reference counted types, and the required shared mutability rules out borrows. Furthermore, lifetimes are dynamic and don't follow the borrowed-data-outlives-the-borrower discipline.

The above is not, though, exactly true. Reference-counted types do support cycles through their weak pointers [102] mechanism, as is demonstrated in the `Rc<RefCell<T>>` example developed as part of the research work of this thesis.

What is said otherwise holds true, and the documentation proceeds to describe the very same approach to solving the problem that has been employed through the course of this thesis [65]:

> In these situations, it is tempting to store objects in a `Vec<T>` and have them reference each other via their indices. No more borrow checker or ownership problems! Often, this solution is good enough.
>
> However, now we can't delete individual items from that `Vec<T>` when we no longer need them, because [of the ABA problem].

Similar to versioning in slotmaps, `Arena` also uses a "monotonically increasing generation counter" to keep track about if retrieved elements match the ones referred to by the custom-implemented index types.[65]

However, `Arena` differs from slotmaps by associating the incrementing value with the entire collection instance. In other words, every time a new element is inserted into a generational arena, the element inserted is associated with the then-current generation of the arena in question, after which the generation value of the whole arena is incremented.

We will later, when studying **`StableList`**, see that this collection-wide incremental generation value is redundant and can be simplified into per-element basis.

However, aside from that, generational arena is by the criteria set for this chapter the closest implementation to **`StableList`**. It satisfies the criteria 1–5, implementing cross-iteration support in its API through the method **`fn get2_mut(…)`** and even explicitly mentioning in the documentation “[z]ero unsafe”.[65]

4 StableList public interface

`StableList` is a novel collection data type for Rust that is intended to be employed as a pool of game objects in a video game engine. It satisfies all the criteria set in Chapter 3: it is ABA-immune, reuses previously vacated space for new inserted elements, and it allows mutable cross-iteration of its elements. It has also been completely implemented in safe Rust.

`StableList` does not support sorting, as is common for pool data types, e.g. the slotmaps, **`StableGraph`** and **`Arena`** among the already presented comparable collections. **`StableList`** also does not support direct indexing into its elements using the bracket notation of Rust [103] or iteration in the Rust-idiomatic manner [104]. The technicalities involving implementing these two traits and why it cannot or has not decided to not be implemented for **`StableList`** will be iterated over in Chapter 5.

4.1 Accessing StableList with StableIndex

Individual elements within any **`StableList`** are accessed using instances of type **`StableIndex`**, also referred to as *access key* within the scope of this thesis. This is similar to how elements are accessed in the previously discussed slotmaps, for example, but differs from many other collections, such as **`Vec`**, where a plain integer suffices.

Dictionary-type data structures [105][106] do resemble the approach of **`StableList`** more in the sense that non-primitive types can be used to access the data in the collection, but whereas for dictionaries one is usually given liberty to quite freely choose the key type specific to their needs, **`StableList`** only allows the single type **`StableIndex<T>`** to be provided as the access key, albeit the inner identifier type can be customized within given bounds.

`StableIndex<T>` is instantiated and returned at the call site when inserting any element into any **`StableList<T>`**. **`StableIndex<T>`** is required by **`StableList<T>`** as an argument in most methods that provide access to single elements, and in methods that iterate over all currently contained elements, **`StableIndex<T>`** provides the identifier data that can be used for distinguishing and identifying elements from each other. Similarly, removing any single element within any **`StableList`** can only be done by providing the appropriate **`StableIndex<T>`** to the associated method.

`StableIndex<T>` implements the traits **`Clone`** and **`Copy`**. The latter requires the former, allowing any instance of the type to be copied when making an assignment operation, instead of a move:

```
//let ’index’ be some instance of StableIndex
let index2 = index.clone(); //’index2’ is a clone of ’index’
let index3 = index2;
//’index3’ is a copy of ’index2’
//and, by transitivity, of ’index’
```

It should be noted that **`StableIndex<T>`**, while being a generic type that is has guaranteed compile-time-checked incompatibility with any instances of **`StableList<T'>`**, where **`T' ≠ T`**, does not distinguish which **`StableList<T>`** it originates from. It is thus possible to (unintentionally) use access keys referring to one **`StableList<T>`** to access elements in another **`StableList<T>`**, and while most likely the access method will return a **`None`** value due to index–identifier mismatch (discussed more in detail in Section 5.1), it is also possible to get a data hit, where the parameters of the access index, by coincidence, match an element within the incorrect **`StableList<T>`**. This is not desired behavior any more than using known indices of one **`Vec<T>`** to access elements within another **`Vec<T>`**, and should be consciously avoided.

More in-depth analysis of how **`StableIndex`** operates, covering, among other things, how it identifies valid entries within the collection and what is the purpose and impact of different identifier types customizable to be used, is restricted to the Section 5.1. This chapter only serves to discuss **`StableIndex`** from the perspective of providing a software developer new to the **`StableList`** library the overall understanding over the public interface and the basic means to utilize the collection’s core functionality, with minimal focus on performance and optimization details.

4.2 Constructors & mutators

The constructor method **`fn StableList::new()`** instantiates an instance of **`StableList`** with default parameters.

For customized parameters – non-default capacity, index identifier type or other parameters – an alternative constructor **`fn StableList::with(opts: StableListBuild)`** is available. Using this requires first instantiating a struct of type **`StableListBuild`** to set the desired parameters that is then passed on as the argument to the constructor.

The methods[2] **fn len()** and **fn is_empty()** return an unsigned integer[3] representing the amount of elements in the collection and a Boolean value reflecting whether the amount of elements is greater than zero, respectively.

To insert an element into an instance of **StableList<T>**, the method **fn add(elem: T)**[4] is used. The method returns an instance the access key data type **StableIndex<T>**. This access key is used to access the inserted element in the future, as there is, by design, no direct way to do that without it.

The method **fn remove(i: StableIndex<T>)** removes the element pointed to by access key **i** from the collection, returning the element wrapped in an instance of the algebraic data type **Option::Some(T)** [53]. If the access key was invalid, i.e. the element pointed to by it had already been priorly removed, the operation does not mutate the collection and returns **Option::None** instead.

The method **fn clear()** empties the whole collection, leaving it with no elements. Whether or not the method will truncate the memory reserved for future insertions depends on the options provided on instantiation – with default options no memory space truncation will take place.

4.3 Single-element access

To access the elements within the collection, **StableList** provides a number of different methods for different purposes and use cases.

The possibility to simply access the single element pointed to by an access index is provided by the method **fn at(i: StableIndex<T>)**. With a valid access index provided, the method returns a reference to the element wrapped in an instance of the type **Some(&T)**[5]. Using an invalid access index returns **None**.

---

2 Methods in Rust always feature a *receiver* as their first parameter for accessing the instance: **&self** for methods that do not mutate the instance, **&mut self** for those that do. **instance.method(arg1,arg2)** is syntactic sugar for **Struct::method(&instance,arg1,arg2)**. For brevity, this thesis omits receivers in method signatures: all functions called ”methods” in the text should be assumed to include an appropriate receiver as their first parameter in the codebase: **fn len(&self)**, **fn add(&mut self, elem: T)**.

3 Rust's type safety extends to different numeric primitives. The method **fn len()** returns the amount of contained elements as the type **usize**, which is an unsigned integer of hardware-specific size: 32 bits for 32-bit machines, 64 bits for 64-bit machines. Returning the length of **StableList** as an **usize** is analogous to how **Vec** and many other third-party collections report this as well.

4 In Rust function and method signatures, as well any other type declarations, the parameter name precedes the parameter type, separated by a colon. In other words, the parameter declaration **Type type** in C-like syntax corresponds to **type: Type** in Rust.

5 In Rust, references to a type **Type** are indicated with a preceding ampersand: **&Type**. As Rust also strongly distinguishes between immutable, shared, read-only access and mutable, unique, read-and-write access, the latter is indicated by **&mut Type**, while the unmarked **&Type** indicates the former.

Returning a reference to the element – or *borrowing* – as opposed to returning the value itself, allows for inspection of the element without moving it out of the collection, which, in comparison, would require mutating the collection to remove the element from it, thusly also disallowing any future access to the element as part of it.

However, the reference returned by the unmarked method `fn at(i: StableIndex<T>)` is of type `Option<&T>`, meaning that the access to the element returned as a reference is limited to read-only inspection.

If the ability to mutate the element inside the collection is required, the mutability-providing variant of the method needs to be used. The method `fn at_mut(i: StableIndex<T>)` returns `Option<&mut T>` instead of `Option<&T>`. The method still returns a reference and not a value, meaning that the element is not moved out of the collection in the process. However, it permits the caller to mutate the borrowed element instead of only being restricted to inspecting it.

As per the rules regarding the concepts of ownership and borrowing, i.e. the rules in place to uphold the safety guarantees provided by the memory model of Rust, only one mutable borrow may exist at any one time. More precisely, either a maximum of one mutable borrow or any number of immutable borrows may coexist, but never may a mutable borrow coexist even with a single immutable borrow.[12][13]

It is also noteworthy that `fn at_mut(i: StableIndex<T>)` only provides mutable access to the element directly pointed to by the access key. Due to the mentioned strict nature of mutually excluding mutable and immutable borrows, if an element `e` is mutably borrowed, no other element `e'` may be even immutably inspected when the mutable access of `e` is in scope.

`StableList` provides a pair of public methods to iterate over all elements contained within it, each independently.

The method `fn each<F>(mut f: F)` takes a generic type `F` as its sole argument. There are additional trait bounds [107][108] defined for the method that restrict the argument type further: `F: FnMut(StableIndex<T>,&T)` dictates that `F` should implement the trait `FnMut`, which applies to any closure (anonymous functions analogous to *lambda expressions* in some other languges) [109] that is allowed to mutate its captured environment, and that the arguments to this function should be of type `StableIndex<T>` and `&T` (a non-mutable reference to a type `T`), in this order. The mutability of the captured environment, syntactically contained within the type `F` in this case, is also the reason why the parameter name is required to be preceded by the mutability indicator: `mut f: F`.

The first argument in the closure provides access to the access index of each element, as occasionally it is useful to be able to inspect each element's identifier. The second argument, on the other hand, provides immutable read access to the element itself, allowing us to iterate over all the elements – even though with a syntax different from the default **for** loop usually utilized for iteration, the support for which is lacking in **StableList**.

Using the method **fn each<F>(mut f: F)**, we can, for example, calculate the sum of all the elements in the instance of **StableList<u32>**[6] named **integers**:

```
let mut sum: i32 = 0; //type definition not necessary here, added for explicitness
integers.each(|_,x| sum += x); //underline indicates an unused argument
```

To demonstrate the use of accessing the access index data of each element, the following piece of code outputs each element's identifier together with the element's associated value:

```
integers.each(|i,x| {
        let id = i.id;
        println!("{id}: {x}");
});
```

The method **fn each_mut<F>(mut f: F)**, with further trait bounds **F: FnMut(StableIndex<T,Id>,&mut T)**, is the mutable counterpart of **fn each<F>(mut f: F)**, allowing mutation of each elements within the collection.

The following piece of code increments the value of every element in the instance of **StableList<u32>** named **integers**[7]:

```
integers.each_mut(|_,x| *x += 1);
```

Similarly to its immutable counterpart, we can also access the access key during iteration. However, it is worth noting that the signature of the closure defined in the trait bound is **FnMut(StableIndex<T,Id>,&mut T)**, without a mutable reference **&mut** specifier prefixed to the first argument that is the access key. This means that only the element data itself is mutable, and the access indices associated with each element

6 With the exception of the hardware-dependent integer types **usize** and **isize**, integer type sizes are always explicitly stated in the type declaration in Rust: **u32** denotes an unsigned 32-bit integer, while **i8**, for comparison, would correspond to a signed 8-bit integer.

7 Mutation of the variable **x** is marked with the dereference operator *****. This is because **x** is not actually a value of 32-bit integer type **i32** but a (mutable) reference to a value, **&mut i32**. The reference needs to be dereferenced into value ***&mut i32 = mut i32** to be able to use integer-specific operations, such as addition assignment **+=**, on it.

cannot be manipulated. This design choice is justifiable, as firstly there should be no need to modify any element's means to access it, and secondly any such manipulation would invalidate all prior access keys that already exist.

The following example sets any element within the instance of `StableList<u32>` named `integers` to value `0` if the element's access index identifier `id` is an even number:

```
integers.each_mut(|i,x| {
	if i.id % 2 == 0 {
		*x = 0;
	}
});
```

4.4 Cross-iteration

For cross-iteration support, i.e. inspecting and/or mutating multiple contained elements at the same time, `StableList` provides additional specialized methods.

As per Rust's strict memory model built on the concepts of ownership and borrowing, inspecting any one element within the collection in this rule system automatically implies inspecting the whole collection. Additionally, a maximum of one instance of mutable access is mutually exclusive with an arbitrary number of shared access. These circumstances result in a situation where a single element cannot be mutated while any other element in the same collection is inspected, as the former case counts as mutating the entire collection and the latter as inspecting the entire collection, which are contradictory settings and will not be passed by the compiler.

To circumvent this problem and facilitate cross-iteration, all methods providing cross-iteration functionality provide an auxiliary data type `StableAccess`.

This data type works as an abstraction over all the remaining elements of the original `StableList` in relation to the element pointed to by the access key – or the *pivot element* – insofar that it then allows accessing these remaining elements individually with the same methods `fn at(i: StableIndex), fn at_mut(i: StableIndex), fn each<F>(mut f: F), fn each_mut<F>(mut f: F)` as is the case with accessing any element in a `StableList`. This polymorphism is achieved by both `StableList` and `StableAccess` implementing an auxiliary trait `StableView` that requires these methods to be implemented in its trait interface.

It is worth noting that **StableAccess** should perhaps be better regarded as a temporary, auxiliary pseudo–data structure than a fully functioning persistent data structure due to it existing only within the scope and context of closures associated with the two methods used for cross-iteration. Instances of **StableAccess** cannot outlive or be transferred outside of these contexts, nor can they be manually instantiated or stored anywhere else for later use. They are as if temporary ”views” into the other elements when the pivot element is accessed.

The first of the two cross-iteration-supporting methods is `fn apply<F>(i: StableIndex, mut f: F)` that has further trait bound `where F: FnMut(&mut T,&mut StableAccess<T>)`. This method is basically a functional extension over `fn at_mut(i: StableIndex<T>)`: it primarily accesses a single element within the instance of **StableList**, granting mutability; however, compared to `fn at_mut(i: StableIndex<T>)`, `fn apply<F>(i: StableIndex, mut f: F)` also allows inspecting and mutating any or all of the other elements of the **StableList**.

In other words, the method `fn apply<F>(i: StableIndex, mut f: F)` allows accessing and mutating the accessed element in relation to any other element within the collection, while also giving the possibility to mutate these other elements in relation to the primarily accessed element.

An example of a use case for the functionality provided by the method `fn apply<F>(i: StableIndex, mut f: F)` is where a character in a video game would get a boost to a certain stat of theirs, such as *power* in this example, according to and linearly scaling with the amount of friendly characters it has around within a certain radius:

```
//let variable 'characters' be a StableList of all the currently existing
//game characters in the game world
characters.apply(boosted_index,|boosted_character,other_characters| {
      let mut allies_in_vicinity = 0;
      other_characters.each(|other| if other.is_friendly
      && boosted_character.distance_to(other) < boost_radius {
            allies_in_vicinity += 1;
      });
      boosted_character.power = boosted_character.base_power +
      allies_in_vicinity*power_bonus;
});
```

Another example is a projectile in a video game hitting its intended target that also causes damage to any other enemies residing within a defined radius:

```
//the variable 'characters' is a StableList of all the currently existing
```

```
//game characters in the game world,
//the variable 'projectile' is the projectile data
characters.apply(target_index,|target,other_characters| {
	target.damage(projectile.damage);
	other_characters.each_mut(|other| if other.is_enemy
	&& target.distance_to(other) < projectile.radius {
		other.damage(projectile.damage);
	});
});
```

As a difference to its single-element analogy **`fn at_mut(i: StableIndex)`**, the method **`fn apply<F>(i: StableIndex, mut f: F)`** does not come accompanied by a non-mutable counterpart. Such was not implemented as part of the interface of **`StableList`**, as its use would be most likely rather limited: if one is to mutate only the primary element and leave the other elements untouched, as in the first example, the method is still required to be declared mutable. If one is to mutate only the other elements and not the primary elements, as in the second example, the method is required to be declared mutable. Only in the case where one immutably only inspects both the primary element and the other elements can the method be declared immutable, which allows it to be called even from immutable contexts, with no limitation on how many immutable references are pointing at it simultaneously. However, at least in the anecdotal experience of the author, such cases of immutable context required for cross-iterative inspection over collection present themselves quite rarely – generally they are associated with the right to mutate at will.

It is to be noted, though, that should such need present itself, the public interface of **`StableList`** can be easily expanded to include an immutable analogy of the method **`fn apply<F>(i: StableIndex, mut f: F)`**, where a single element pointed to by the access key could be only inspected, not mutated, in relation to all the other elements in the collection and vice versa. This would not even constitute a breaking change in relation to the currently existing versions of the public interface, as the current mutable method **`fn apply<F>(…)`** should not be renamed **`fn apply_mut<F>(…)`** in order for a new, immutable method **`fn apply<F>(…)`** to take its name – the name ”apply” is clearly a misnomer for an immutable method that cannot actually *apply* any effect on the collection or any part of it thereof. A much better naming option for the new immutable method would be **`fn inspect<F>(i: StableIndex, mut f: F)`**, for example, as it provides cross-iterative inspection capabilities.

The pieces of code demonstrated above exemplify what is meant in the context of this thesis with the term *cross-iteration* and what is the central role of **`StableAccess`**, represented in both by the variable named **`characters`**: through **`StableAccess`** we are able to further iterate over all the other elements of the collection while already operating inside the scope of the collection – in this case inspecting or mutating a single element.

For traversing the whole collection in a cross-iterative manner, `StableList` provides the aptly named method `fn traverse<F>(mut f: F)`, further specified with the trait bound `where F: FnMut(StableIndex<T>,&mut T,&mut StableAccess<T>)`.

Whereas the method `fn apply<F>(i: StableIndex, mut f: F)` allows accessing a single primary element within the collection and mutating it in relation to the rest of the elements, and vice versa, `fn traverse<F>(mut f: F)` iterates over each element within the collection one at a time, providing the possibility to mutate every other element in relation to the then-relevant set of remaining elements – and vice versa.

Whereas `fn apply<F>(…)` is the cross-iteration-providing analogy to `fn at_mut(…)`, `fn traverse<F>(…)` can be thought as a cross-iteration-providing analogy to `fn each_mut(…)`. `fn apply<F>(…)` provides cross-iteration capability for a single element pointed to by the access key, while `fn traverse<F>(…)` provides cross-iteration capability for all the elements contained within the collection at that point.

Compared to the closure provided as an argument to `fn apply<F>(…)`, the closure for `fn traverse<F>(…)` requires one additional argument: the argument of the type `StableIndex<T>` can be used to inspect the access key data of each element, if needed – identically to how the same possibility is offered by the methods `fn each<F>(…)` and `fn each_mut<F>(…)`. This access key data is not provided as a separate closure argument for `fn apply<F>(…)`, as the access key is already known at the point of calling the method, since it is one of its required arguments.

The method `fn traverse<F>(…)` forms the basis of the game loop in cases where `StableList` is used to store the instances of game objects in a game engine.

A prototypical example of employing `StableList` and its method `fn traverse<F>(…)` to run the game logic update loop would be as follows:

```
game_objects.traverse(|i,x,others| x.update(i,others));

struct GameObject {
	//any fields required for game objects
}

//function and method declarations for ’struct GameObject’
impl GameObject {
	pub fn update<A: StableAccess> (&mut self, i: StableIndex, others: &mut A) {
```

```
                //any logic to interact with other game objects
        }
}
```

To make game objects move around and report collision through terminal output when they come within a certain threshold with any other game object in a two-dimensional space, one could define the method **`fn GameObject::update(…)`** as follows:

```
impl GameObject {
        pub fn update<A: StableAccess> (&mut self, i: StableIndex, others: &mut A) {
                self.x += self.vel_x; //update x-position by adding x-velocity
                self.y += self.vel_y; //update y-position by adding y-velocity
                others.each(|i_other,other| {
                        if (self.x - other.x).abs() < self.radius
                        && (self.y - other.y).abs() < self.radius {
                                println!("{} collided with {}",i.id,i_other.id);
                        }
                });
        }
}
```

One can of course notice from the example above that it does not use **`fn each_mut<F>(…)`**, as there is no need to mutate the other elements except the currently traversed one. For this reason the Rust compiler would also output a warning about the method signature, pointing out that the argument **`others`** does not need to be declared mutable – **`others: &A`** would be sufficient.

However, thanks to the robust implementation of **`StableList`** and **`fn traverse<F>(…)`**, **`fn each<F>(…)`** could be replaced with **`fn each_mut<F>(…)`** and the logic inside be modified to mutate the other elements as well, and the code would still compile successfully. One could, for example, make such adjustments that game objects colliding with each other would reduce each colliding game object's health value:

```
                //...
                others.each_mut(|i_other,other| {
                        if (self.x - other.x).abs() < self.radius
                        && (self.y - other.y).abs() < self.radius {
                                self.hp -= other.damage;
                                other.hp -= self.damage;
                        }
                });
                //...
```

In essence, wrapping our game loop into **`fn traverse<F>(…)`** provides us the same functionality and possibilities that can be achieved in languages with less strict, non-ownership-based memory models by simple nested **`for`** loops:

```
for object in game_objects {
	//this scope could also be converted into
	//method 'update(other_objects: L)', for example,
	//where L is a list or any other collection of
	//game objects of choice
	object.x += object.vel_x;
	object.y += object.vel_y;
	for other_object in game_objects {
		if object != game_object {
			//check for collision, for example
		}
	}
}
```

However, as already mentioned and as we shall study more closely in the following chapter, there is currently no way for **`StableList`** to support cross-iteration powered by **`for`** syntax.

5 Internal implementation of StableList

Previous chapter introduced **`StableList`** through its public interface – i.e. how one can, in practice, employ **`StableList`** as an external library in one's own program code and software project for utilizing any features it provides.

This chapter will discuss the internal technical implementation of **`StableList`** and more closely explain details about its performance and behavior in different situations and reveal how to possibly optimize or develop the data structure further.

In Section 5.1, we inspect the internal composition and operation of the data type **`StableIndex`** that is used to random-access and remove elements. In Section 5.2 we inspect the internal composition of **`StableList`** itself, and its behavior when inserting and removing elements. Section 5.3 examines the auxiliary data type **`StableAccess`** that is essential for allowing cross-iteration. Sections 5.4 and 5.5 discuss the technical reasons why indexing and iteration have not been able to be implemented for **`StableList`**. Section 5.6 introduces a few possible improvements and alternative implementations for the **`StableList`**, as it currently exists.

5.1 Access key implementation

The access index **`StableIndex`** has been mentioned repeatedly in the course of this thesis. It is instantiated and returned to the caller on insertion of a new element. The access index is the only means of random-accessing and removing any element from within the collection. Likewise the collection also provides the associated access index data for each element when iterating through the collection, be it simple single iteration or cross-iteration.

By its inner structure, **`StableIndex`** is a very simple data structure. It only consists of three fields:

```
pub struct StableIndex<T,Id: StableId = u32> {
        index: usize,
        pub id: Id,
        marker: PhantomData<T>,
}
```

Of these two, however, only **`index`** and **`id`** are runtime fields. The field **`marker`** is a standard-library-provided type for indicating any kind of phantom data – metadata that will be checked for type correctness during compile time static checking but will be completely eliminated for runtime. This concretely means that the

runtime size of the field `marker` is always 0 – it does not exist.[110] Here the field `marker` is in place only to guarantee that the user of `StableList<T>`, where `T` is some concrete type, will only ever be using access indices of the same concrete inner type `StableIndex<T>` to operate on it – if `StableIndex<U>`, where `T ≠ U`, is used to attempt to access `StableList<T>`, the compiler will prompt an error and not proceed with compilation. Including the non-runtime field `marker: PhantomData<T>` is our an allowed workaround for us to allow this type checking – if we leave the field out, the compiler will reject the unused generic type parameter `T` in the structure signature, as it is not being employed by any of its fields.

However, there are no mechanisms in place for `StableList` that would prevent the user from trying to access $StableList<T>_A$ with access indices from another $StableList<T>_B$. If this, by coincidence, happens to return existing option type variant `Some(x)` values from $StableList<T>_B$, this is still to be considered logical error – the same way as using integer indices intended to refer to $Vec<T>_A$ to access elements in another $Vec<T>_B$ is – as the indices and the elements originate in different collections.

The field `index` is simply the index in the internal element space of `StableList` where the associated element e is supposed to reside. However, as has been mentioned about the ABA problem, by only using an index to access an element, there is a possibility that our associated element was already removed from the list. This is not a problem per se, as trying to retrieve an element from the element space immediately after its removal would return the absent option type variant `None` value, clearly signaling that the element is no longer present. However, if a new element e' has since been inserted into the collection, and the collection has placed it into this same memory position where our associated element e used to reside before its removal, without any additional guard mechanisms implemented we will now be returned e' – without any kind of indication this is actually not the same element that our index was associated with.

The field `id` is precisely the kind of additional guard mechanism to counteract the ABA problem. It is similar to the `version` field in slotmaps presented as part of the comparable collections.

Technically speaking the field `id` takes a generic type whose only requirement is that it implements the `stablelist`-crate-provided `trait StableId`. The trait requires two methods to be implemented: `fn init()`, to set the initial value that will be used for the first identifier assigned, and `fn up(&mut self)` that will ”increment” the identifier. Moreover the types to implement `trait StableId` also need to fulfill the following trait bounds: `Copy` to allow for syntactically easy copying of identifiers and, as a result, their encasing `StableIndex` structures, `Eq` to able to verify the identifier equalities, and `Display` for printing – mostly for debugging purposes and since `StableIndex` already guarantees printability with its implementation of `Display` as well.

The `stablelist` crate provides the implementation `impl StableId for u32`, and `u32` is also the default

identifier type that any **StableList** will use, unless particularly specified otherwise. The initial value of **u32** as an identifier is *0*, and it will be incremented by *1* on update.

As the identifier is a generic type, one is free to use a type of their own choice, as far as they first implement **trait StableId** on it first. **String**, for example, is unallowed, as **String** does not support the trait **Copy** (and cannot implement it [111]). Of course the question is also, what would be the intended purpose and the method of using a string of characters as an incrementing identifier?

Some good candidates, however, are universally unique identifiers or UUIDs [112]. The Rust crate *uuid* provides ready implementations for these, and the provided structure **Uuid** already implements **Copy**, **Eq** and **Display**. A trivial, fully working approach would be to have both **StableId**-trait-provided methods **fn init()** and **fn up(&mut self)** generate a new UUID. The easiest way to do this is to call **fn uuid::Uuid::new_v4()** that employs the system-specific random number generator or **fn uuid::Uuid::now_v7()** that does the same and also factors in the current timestamp.

Regardless of the concrete type that has been specified for the identifier, the instance of **StableList** initializes its own copy of such an identifier and maintains it. Each element inserted into **StableList** will be assigned the identifier from the then-current copy of the **StableList**, after which **StableList** updates its own copy after every insertion. The identifier, thusly, works as a running number, with every element inserted into the collection having a distinct identifier until the maximum capacity of the identifier type is reached, after which the identifier value rolls over and starts over. For the default **u32** unsigned integer type, this would happen after 2^{32} = 4 294 967 296 insertions.

When random-accessing an element in the collection, the field **index** is used to pick the memory position to inspect within the internal element space. If this memory position has an existing element in it, the access index **id** is compared to the **id** stored within that memory position. If they match, an option-wrapped existing reference to the element **Some(&T)** is returned. In all other cases, **None** is returned.

Removal of individual elements works mainly with the same logic, but the possible transformations that **StableList** might impose on its element space in the process are discussed more in depth in the next section.

For random access and removal, the accessed element is guaranteed to be the same as for which the access index was initiated if no overflows have yet occurred for the identifier value. This means that for fewer than 4.3 billion insertions occurred, collision matches – i.e. returning the incorrect element that has the same identifier associated with it – are guaranteed to not occur. Even after overflows have occurred, the chances of identifier collisions are still very low: with the identifier space having overflowed once, for every insertion,

there is only a $1/2^{32}$ chance that the identifier of the newly inserted element would match the identifier of an element that used to reside at that position. Accordingly, after two overflows the chance is still only $2/2^{32}$, after three $3/2^{32}$, etc. Furthermore, even in the case of an identifier match, there is a possibility of the element being unintentionally accessed only if there still are access indices existing from the time when the previous element with the identical identifier existed within the collection.

Only the field `id` is declared public in `StableIndex`. This is to allow for easy reading of the field, as this can be also used to identify elements elsewhere in the target application, not only for accessing them in `StableList`. This means that this field can technically be mutated, if one wants, but this is strongly counteradvised, as this invalidates the access index in one way or another. The field `index` is hidden from the user, so that its value is not readable at all. The field `marker`, as mentioned, does not exist at all in runtime.

5.2 Memory layout

`StableList` stores its elements internally as a sequential container, a standard-library-provided `Vec`. However, they are encapsulated into an intermediate abstraction: instead of being stored as `Vec<T>`, where `T` is the element type, the actual implementation of the field in `StableList` is `elems: Vec<StableCell<T,Id: StableId = u32>>`, where `StableCell<T,Id: StableId = u32>` is a custom enumeration [113][114][115].

The `StableCell<T,Id: StableId = u32>` is a analogical to the standard-library `Option<T>`, or option type. Whereas the option type has two variants – `Some(T)` representing existing encapsulated data `T` and `None` representing the absence of any data – `StableCell<T,Id: StableId = u32>` as well has two variants: `Has(T,Id)` represents an existing, present element, while `Not(Option<usize>)` represents the absence of an element. The contained data, as is apparent, differs from that of a standard option.

`Has(T,Id)` does contain the element data `T` inside, just as `Some(T)` does. However, this data is also coupled with identifier data of the generic type `Id` (default `u32`, as with `StableIndex`).

As described in the previous section with the mechanism of random access, the field `index: usize` in `StableIndex` corresponds to the actual index in the field `elems: Vec<StableCell<T,Id>>` of `StableList`, and if the memory position at that index contains holds the `Has(T,Id)` variant, the field `id` in `StableIndex` is checked for equality with the second field of the type `Id` of `Has(T,Id)`. If they do match, an option-wrapped existing reference `Some(&T)` to the first field of the type `T` – the element itself – is returned. In all other cases, an absence of reference `None` is returned.

If an element has been removed from a memory position, the variant of the type **Not(Option<usize>)** takes its place. In comparison to the standard **Option::None** is, **Not(Option<usize>)** is not a singleton-like empty type but instead contains an optional **usize** integer inside. This is used to achieve the effects of an freelist and plays a significant part in insertion and removal.

The typical approach to implementing an freelist – used by **Slab**, **StableGraph** and the slotmaps – is built on the same basis. Each memory position can indicate through their absent variant (here: **Not(Option<usize>)**) the index of the chronologically previous vacated memory position. If the freelist is empty, removing an element e_i at position i will set the memory position to hold a **Not(None)** value, as there do not exist any vacated positions that are chronologically prior to the first vacated one. However, the subsequent removals will set indices to any priorly vacated positions correspondingly. Subsequently removing an element e_j at the index j will set this memory position to **Not(Some(i))** value, and then removing an element e_k at the index k will set this memory position to **Not(Some(k))**. The tail of a freelist will always store a **None** value to indicate the end of the freelist. On each of these removals, the head of the freetail at all points is stored into a field of the collection, in the case of **StableList** named **next: Option<usize>**. If **next** is **None**, the freelist is empty, otherwise there are vacant memory positions tracked in the freelist. Visualized, the example described here behaves like this over time:

Initial state:

next: None,

Remove element at **i**:

next: Some(i), **elems[i]: Not(None)**

Remove element at **j**:

next: Some(j), **elems[j]: Not(Some(i)),**
elems[i]: Not(None)

Remove element at **k**:

next: Some(k), **elems[k]: Not(Some(j)),**
elems[j]: Not(Some(i)), **elems[i]: Not(None)**

Obviously this mechanism creates a single-linked list: each element links to the next element, and traversing it until the end requires visiting all of the elements, resulting in *O(n)* time.

Where **StableList** differs from this typical, trivial approach used by other comparable Rust collections is that it employs partitioning of the element space.

StableList itself contains the following metadata fields to track different counts related to its content:

```
max: usize,
len: usize,
open: usize,
```

The field `max` is the capacity of the collection – how many elements it can hold before having to preallocate more memory. The field `len` is the actual number of elements it contains, i.e. the number of `Has(T,Id)` positions at the current moment within the element space. The field `open` is how many memory positions have been touched. This is the sum of the currently existing `Has(T,Id)` variants and already vacated `Not(Option<usize>)` variants. A `StableList` that has been preallocated for 1 000 elements, has had 200 elements inserted and then 50 elements removed will have $max = 1000$, $len = 150$ and $open = 200$.

It is worth noting that in the case of `StableList`, the immediate fields of the structure do not actually include any field `next: usize`, i.e. the index pointing to the head of the freelist, among them. Instead, there are the metadata fields `q1`, `q2`, `h2`, each of the custom-defined type `Partition`. Each of these `Partition` instances, however, includes the following fields:

```
from: usize,
max: usize,
len: usize,
open: usize,
next: Option<usize>,
```

The idea at the center of `StableList` is that the element space of length `L` is at all times subdivided into three partitions: the first quarter q_1 governs the memory positions in the index range $[0, \frac{L}{4})$, the second quarter q_2 governs the memory positions in the index range $[\frac{L}{4}, \frac{L}{2})$, and the second half h_2 governs the latter, remaining half of the memory positions $[\frac{L}{2}, L)$. The field `from` present in each partition stores the information about the first memory position for their partition, while `max`, `len` and `open` are analogous to the respective `StableList`-wide fields, however only bookkeeping the appropriate counts for their own partition in question.

Each of these partitions govern their own freelist, and these partitions point each only to their own freelist's head, through their field `next: Option<usize>`.

Updating the partition-specific freelists works exactly the same as in the case of a single, unified freelist, with the exception that `StableList` now checks in which partition the removal occurred. In other words, a check is first conducted on the basis of which range the removed index i belongs to: for each partition p, check if $p.from <= i < p.from + p.max$, and if so, set $elems[i] \leftarrow Not(Some(p.next))$ and and set $p.next \leftarrow i$.

With this approach it is guaranteed that for each partition p, all the indices in the freelist from its head to the tail will reside within the range of the partition.[8]

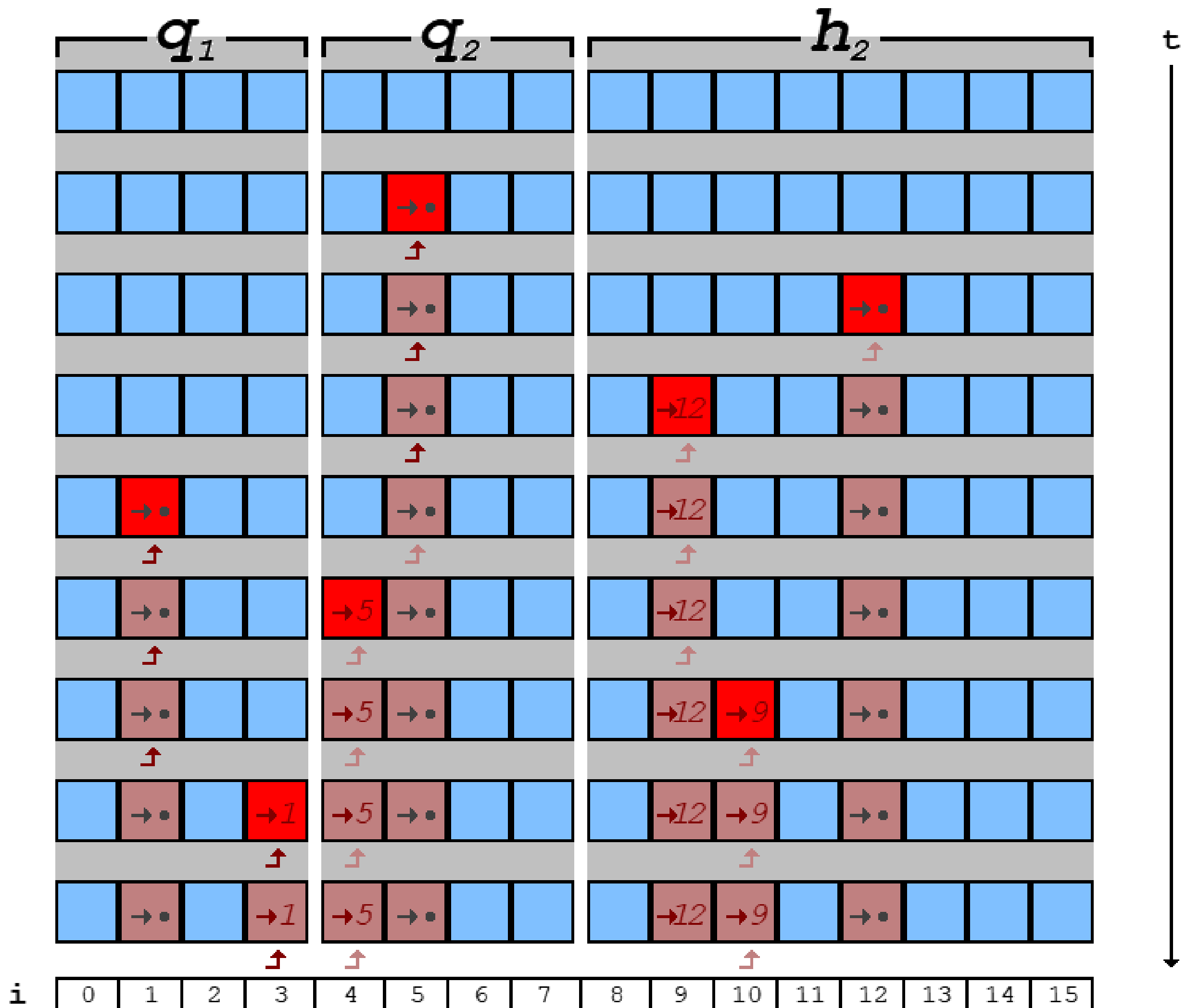

Figure 5.1. StableList with removals taking place. New freelist indices are added on each removal in a linked list–like fashion, and all the indices of a certain partition's freelist are guaranteed to point only to the same partition. If no intervening removals take place, the next elements would be inserted at the memory positions 3, 1, 4, 5, 10, 9, 12, in this order, before expanding the element space.

The partitioning of **`StableList`** is at the core of its functionality as the only compared collection that proactively favors truncation. As we now have three partitions q_1, q_2 and h_2, every time an insertion takes place, these partitions are checked in this order. The element will be inserted into the first encountered

8 One can prove this through induction. If the freelist for partition p is empty, then $next(p) = Ø$. If now a removal occurs at the index i that resides within the governed range of p, set $elem[i] \leftarrow Ø$ and $next(p) \leftarrow i$. If then another removal occurs at the index j within the governed range of p, set $elem[j] \leftarrow i$ and $next(p) \leftarrow j$. Only removals in the range of p will ever be recorded into $next(p)$ and propagated further when subsequent removals occur. Any removal occurring outside of the range of p will never be recorded into $next(p)$.

partition p that has free space left, i.e. $len_p < max_p$, be it through appending at the partition's end or reinserting into a previously vacated position that is found by popping the head of the partition's freelist. Over time this results in reinserted elements being placed more towards the beginning of the element space instead of uniformly distributed throughout, potentially allowing for more frequent truncations, as illustrated in the Figure 5.1. We will see this in effect especially in the ephemeral data benchmark in Section 6.3, where a case with frequently inserted and removed elements with an upper-bounded lifetime is simulated.

Insertion alters only a single memory position pointed directly by the respective freelist head, after which it will update this freelist head to point to the next index in the freelist (if any). This renders insertion a constant-time operation with time complexity $O(1)$.

In case the number of existing elements **`len`** in **`StableList`** is to exceed **`max`**, the maximum length **`max`** is set to $2 \cdot max$ and memory space for this amount of elements is reallocated. This is analogous with all the comparable collections and is the default reallocation policy for the standard **`Vec`** as well. However due to partitions, **`StableList`** has to also update their relevant metadata to reflect the new length. This is in practice simple, as if the **`StableList`** needs to be expanded, it is full; if **`StableList`** is full, all of its partitions are full; and if the partitions are full, each of their freelists are empty, as there are no vacant memory positions in the whole collection. Thusly updating the governed range metadata and the capacity metadata to reflect the new partition bounds is sufficient. The elements count of q_1 and q_2 is also doubled in the expansion, while the elements count of h_2 is set to 1, as it has only one new element appended into the otherwise empty, new partition. Figure 5.2 exemplifies the repartitioning of the **`StableList`** when inserting into a full collection.

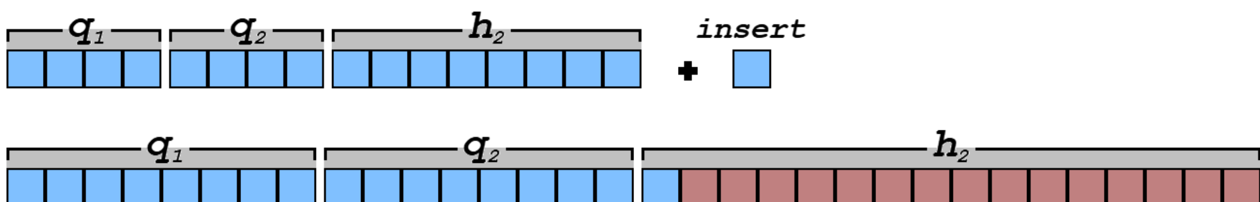

Figure 5.2. Repartitioning of a StableList when inserting into full element space.

In case as a result of element removal the partition h_2 ends up empty, i.e. $len_{h2} = 0$, truncation may take place. In this case **`max`** is set to $^{max}/_2$.

Furthermore, the collection in its new, half-shorter length will be repartitioned. First of all, all the partitions p_i will have their range metadata updated to reflect the new half-length.

A new latter-half partition h_2' metadata will be formed by capturing the freelist of q_2. As freelists are stacks implemented as linked lists, reassigning the head is sufficient: $next(h_2') \leftarrow next(q_2)$.

After this new q_1' and q_2' metadata will be formed from the old q_1. The freelist of q_2' is set to empty, as it is yet to be defined: $next(q_2') \leftarrow \varnothing$. The freelist of old q_1 will be traversed through one index at a time. Each of these indices is checked for if they reside within the newly bounded q_1' and q_2' – in the latter case, they are removed from $next(q_1')$ and added to $next(q_2')$ in the same way as with regular element removal. After the whole freelist of old q_1 has been traversed through, both q_1' and q_2' will have updated freelists that are valid in terms of the new partition bounds. It is worth noting that as this approach creates the freelist for new q_2' by popping the appropriate freelist indices from old q_1 and then pushing them into the new q_2', the indices in this new freelist of q_2' will be in reverse order compared to how they were ordered in old q_1. Figure 5.3 demonstrates the repartitioning of **`StableList`** over truncation.

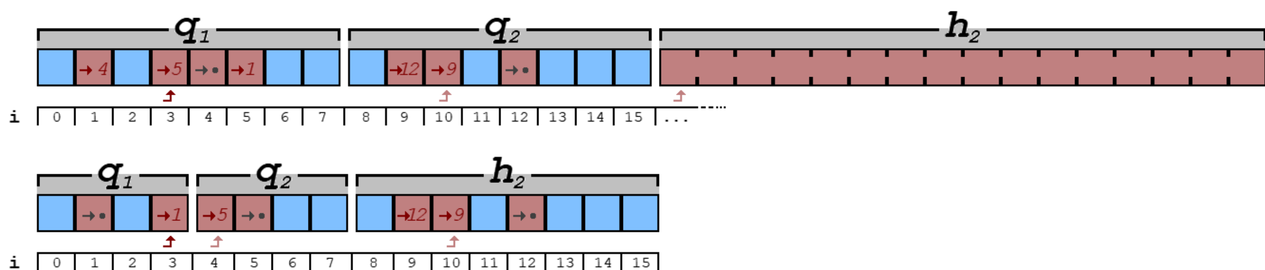

Figure 5.3. StableList before and after truncation. The freelist of old partition q_2 is effectively repurposed as the freelist of the new partition h_2. The freelist indices of the old partition q_1 are redistributed between the new partitions q_1 and q_2. Do notice that the order of visiting the freelist indices in the new q_2 is reversed compared to the order they would have been visited in the old q_1.

Truncation will be repeated over and over again until either the $len_{h2} > 0$ or the *max* value of **`StableList`** has reached $max = min$, where *min* is a yet another field of **`StableList`** that can be set when instantiating it with custom parameters through **`fn with(build: StableListBuild)`**. It is always rounded upwards to be divisible by 4 to guarantee the correctness of the partitioning behavior. Furthermore, truncation can only happen if the truncation strategy has been set to either **`Truncate::Iteration`** (default) or **`Truncation::Memory`**. Setting it to latter will also truncate the allocated memory space every time element space is truncated. This will decrease the current memory requirements of the program, but if the collection is again to expand, reallocating will take additional time.

Removal, especially with repeated truncation, may alter n' elements, where n' approaches the pre-removal touched element space length n_{open}. For example, a **`StableList`** with $n_{open} = 16\,384$, $min = 4$ and 3 present elements at indices $i = \{\ 0,\ 1,\ 16\,383\ \}$ that has its element at $i_{16\,383}$ removed would be truncated to length 8.

In the first truncation step the length of **`StableList`** is reduced to *8 192*, and the freelist of the old q_1, $max(q_1) = 4\,096$, is iterated through. As it has two non-vacant indices, the length of freelist is $4\,096 - 2 = 4\,094$. After that the collection length is again halved and $2\,048 - 2 = 2\,046$ freelist indices would be iterated through, after which *1 022*, *5 10*, *2 54*, *1 26*, *62*, *30*, *14*, $8 - 2 = 6$, $4 - 2 = 2$, and finally $2 - 2 = 0$. The maximum number of visits of indices $i \in q_1 \cup q_2$ to occur during the first truncation is the amount of vacated memory positions $n_{removes(q1,q2)}$ in q_1 and q_2 in total. As $n_{removes(q1,q2)}$ is naturally upper-bounded by *max/4*, the sum of index visits to occur can further be upper-bounded at

$$
\begin{aligned}
& max/4 + max/8 + max/16 + \ldots + min/2 \\
& \leq max/4 + max/8 + max/16 + \ldots \\
& = max/4 \cdot (1 + ½ + ¼ + \ldots) \qquad | \; ((½)^0 + (½)^1 + (½)^2 + \ldots) = 1/(1 - ½) = 2 \text{ [116]} \\
& = 2 \cdot max/4 \\
& = max/2
\end{aligned}
$$

Our example above complies to this formula, as the example **`StableList`** with $max = 16\,384$ had

$$
\begin{aligned}
& 4\,094 + 2\,046 + 1\,022 + 5\,10 + 2\,54 + 1\,26 + 62 + 30 + 14 + 6 + 2 \\
& = 8\,166 \leq 8\,192
\end{aligned}
$$

index visits performed.

The number of visits is grows in correlation to how large the *max* value is in comparison to *min*. Still, due to the logarithmic half-partitioning nature of **`StableList`**, this holds even with smaller margins of difference: with *min* at as high as *2 046*, a total of *7 162* index visits would still be conducted, only ~12.2% less than the *8 166* for $min \leq 8$. With *min* at *4 096* – stopping at the ”next-lower step” only after one truncation – *4 094*, roughly half. This makes sense, as the amount of iterations is increased by ~50% for every further partitioning.

The magnitude of *max* is thus, in fact, a very accurate one and sets a reliable upper bound.

It has to still be noted, that truncations can only occur ever so often – and only in those cases, where a sufficient number of prior removals have taken place, so that the next removal can trigger h_2 becoming completely vacant. Even though the upper bound can be derived from *max*, it is the number of prior removals from q_1 and q_2, $n_{removes(q1,q2)}$ that, at the moment, also upper-bounds the number of index iterations.

In other words, while the number of index visits x is at worst $x \approx max/2$, this can only happen if $n_{removes(q1,q2)} = x$. Thusly x removes have to have already taken place, only after which there would be x index visits

performed. Analyzing this over time, for every x removes, an x number of index visits will occur. This renders the amortized time complexity of `StableList` remove operation at $O(x/x) \approx O(^{max/2}/_{max/2}) = O(1)$.

Amortized, `StableList` removal operation is constant time. However the worst-case scenario is $O(max/2) = O(n/2) = O(n)$, and the frequency of its occurrence is at most at every $max/2$ removals.

For future research an optimization over the current `StableList` would be to perform the q_1 freelist repartitioning only after any number of truncations have taken place as a result of a single removal – not after every single consecutive iteration done. This is an early design oversight of the collection, but it is unclear if it is straightforward to amend due to a current technical peculiarity: without iterating through q_1 the partition length values might represent invalid state in cases where removals took place before the partition $len_p = max_p$.

This change, if successfully implemented, would reduce the concrete number of visits in certain edge cases tremendously – in the presented example case down from 8 166 to 2. Strictly speaking this change would set the upper bound at $max/4$ instead of the current $max/2$, as $max/4$ indices in q_1 would be only visited if truncation would take place once. As a general rule, it would reduce the amount of visits required for every subsequent truncation by 50%, instead of increasing them by 50%, as is now the case. However, the change would keep the time complexity at in the same time complexity class, as the only the constant coefficient would change: current $O(max/2) = O(1)$ would change to $O(max/4) = O(1)$.

5.3 StableAccess-powered cross-iteration

As introduced in Section 4.1, `StableAccess` is a data type that in relation to a primarily accessed *pivot element* encapsulates all the remaining elements of the source `StableList<T,Id>` into a temporary pseudo-collection – a kind of view into the `StableList`.

`StableAccess` is a thin and lightweight wrapper accompanied by minimal functionality. It primarily consists of the fields `head` and `tail`, both of the type `&’a mut[StableCell<T,Id>]`, i.e. a slice [117][118] with elements of the type `StableCell<T,Id>`. The lifetime annotation [51] `’a` governs the lifetime of the slice and is further bounded by the type signature `pub struct StableAccess<’a,T,Id: StableId = u64>` to force that the data referred by the slices `head` and `tail` are guaranteed to exist for as long as the encasing `StableAccess<T,Id>` itself. Technically this means that any `StableAccess` can never outlive their source `StableList` and will always expire at the latest when the source dies – if violation of this is attempted, the compile time static analysis will prompt an error and not halt the compilation.[51]

The field **`head`** stores the element data preceding the pivot element e_i at the index **`i`** in a sequential manner, while the field **`tail`** stores the element data following the **`e`** in a similar manner:

element data at source	=	$[e_1, e_2, \ldots, e_{i-1}, e_i, e_{i+1}, e_{i+2}, \ldots, e_n]$
pivot selected	=	e_i
	→	
`head`	=	$[e_1, e_2, \ldots, e_{i-1}]$
`tail`	=	$[e_{i+1}, e_{i+2}, \ldots, e_n]$

As is clear from the above figure, all the indices included in **`head`** preserve their index: for each element e_{head}, $i_{head} = i_{source}$. However, the indices of the elements in **`tail`** have shifted: the element now residing at the index 0 in **`tail`** is the one that resided at index $len_{head} + 1$ (with the pivot element having resided at len_{head} and the last element in head at $len_{head} - 1$). In general all the elements in **`tail`** have been shifted backwards by the same offset $len_{head} + 1$. Even with the pivot element at index 0, all the indices in **`tail`** shift backwards by 1.

The essential purpose of **`StableAccess`** is to provide the same public interface as **`StableList`** for accessing its elements: **`fn at(i: StableIndex)`**, **`fn at_mut(i: StableIndex)`**, with both using **`StableIndex`** as the index. **`StableAccess`** is necessary for this to convert the original source indices into correctly corresponding indices in the offset **`tail`**. Other than facilitating the offset transposition, **`StableAccess`** is basically an abstraction around the **`Vec`**-provided standard methods for splitting the **`Vec`** into appropriate portions. The basis of **`StableAccess`** is the following logic, similar to the one used on its instantiation:

```
let (left, right_with_pivot): (&mut [T], &mut [T]) =
elems.split_at_mut(pivot_index);
let right_split: Option<(&mut T, &mut [T])> =
right_with_pivot.split_first_mut();
let (head, tail) = match right_split {
	Some((pivot, right_without_pivot)) => (left,right),
	None => unreachable!(), //programmer claim that this case never happens
};
```

This splitting mechanism is absolutely required to permit cross-iteration, as due to the ownership and borrowing rules of Rust the field **`elems: Vec<T>`** can only have one mutable borrow into it at one time. Attempting to mutably borrow two memory positions in it counts as two. Splitting the element space first into *(left,right)* at *pivot* forms two different *borrow units* (ad hoc expression) off it, each with their separate one-mutable-borrow limit. The pivot element, however, is still a part of *right* (as a design choice of split [119]), meaning that one could not actually mutate the pivot element and any element following it in element

space at the same time. Thus the *right* will be split once more, separating the first element that is our intended pivot element from the rest. Now we have three different borrow units: *head* = *left*, *pivot*, *tail* = *right* \ *{pivot}*. As such, the compiler allows us to now mutably access the pivot element while at the same time mutably accessing one element at a time in *head* or *tail*, as demonstrated in Figure 5.4. Technically this setting would allow us to even access the pivot element while accessing both *head* and *tail* at the same time, but this has not been implemented in the public interface, as no use case requiring this feature has presented itself so far.

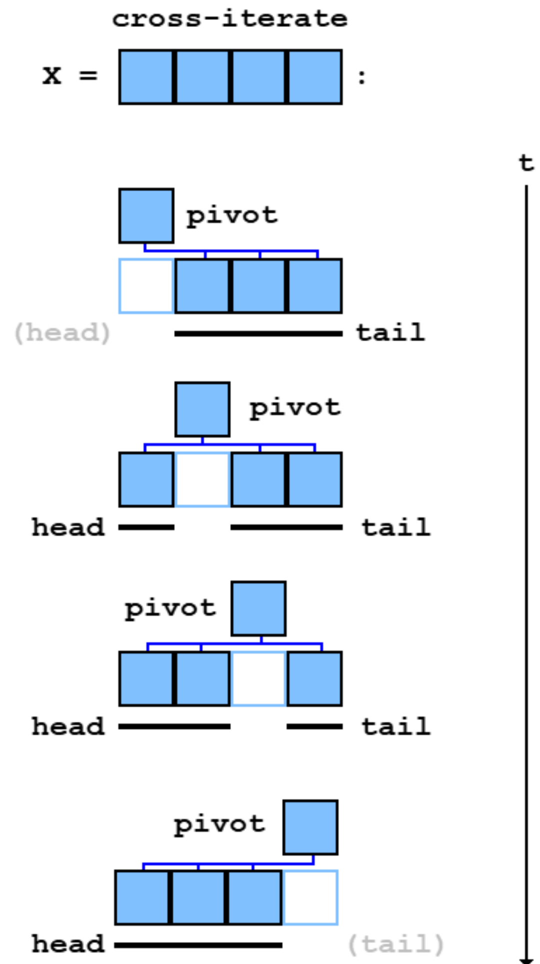

Figure 5.4. Cross-iteration over elements of a StableList of length 4, split in memory by the StableAccess abstraction into separate head and tail regions.

Additionally, **`StableAccess`** and **`StableList`** both implement the trait **`StableView`** that provides the identically named required methods **`fn at(…)`**, **`fn at_mut(…)`**, **`fn each(…)`** and **`fn each_mut(…)`** as **`StableList`**. This allows the user of the library to define the abstract trait **`StableView`** as the input type for appropriate methods, instead of the concrete types **`StableList`** or **`StableAccess`**, employing polymorphism and making method interfaces type-agnostic.

5.4 Lack of indexing access

As mentioned before in the beginning of Chapter 4, **`StableList`** does not support accessing its contained elements through direct indexing. Instead only direct access through an accessor function is possible:

```
//'stable_list' is an instance of type StableList,
```

```
//'index' is an instance of type StableIndex
let x = stable_list[index]; //does not compile, 'trait Index' not implemented
let x = stable_list.at(index); //valid code
```

While this limitation has design philosophy reasons behind it, there are also technical obstacles that do prevent implementing the standard library–provided indexing trait **`trait Index`** (and the mutable indexing trait **`trait IndexMut`** in similar manner, for that matter) for the present implementation of **`StableList`**.

The required method **`fn index(…)`** in the trait **`Index`**, as well as the required method **`fn index_mut(…)`** in the trait **`IndexMut`**, by design choice, always return a reference.[9] Using the indexing syntax **`type[i]`** calls these methods and dereferences the return values, effectively calling **`*type.index(i)`** or **`*type.index_mut(i)`**, so the caller is handed a non-reference value type to read or mutate.

A trivial indexing implementation, using pattern matching against the variants of **`StableCell`**, would be as follows:

```
fn index (index: StableIndex) -> &Option<&T> {
	match self.elems[index.i] {
		StableCell::Has((ref e,_)) => &Some(e),
		StableCell::Not(…) => &None,
	}
}
```

This should return a reference to an option-wrapped reference to the element **`e`** held at the index **`i`**: **`&Some(&e)`** if one exists, **`&None`** otherwise. To the caller of the indexing method this is automatically derefererenced into **`Some(&e)`** or **`None`**.

This approach, however, does not pass the compilation. The problem is that the element space in **`StableList`** stores the values as one of the variants of **`StableCell`**. The indexing method reads this value from the element space and instantiates a new option value based on them, then returning a reference to this option value. The problem is that by Rust-dictated lifetimes, the option value is now only bound to the scope of this surrounding method, as no other ownership has been declared for it – the lifetime of the value expires at the moment the method ends. The method still tries to return a reference to this value, but by the time anything

9 This is due to any read-only indexing call of the form **`container[index]`** actually being syntactic sugar for the call **`*container.index(index)`**. Thus the trait method **`fn index(&self, index: Idx) -> &Self::Output`** is required to return a reference of the output type **`Self::Output`** for the dereferencing operator **`*`** to then dereference it into an actual value. This also applies to mutably indexing, where **`container[index]`** is instead a stand-in for **`*container.index_mut(index)`** and the return type of the trait method is mutable: **`fn index_mut(&mut self, index: Idx) -> &mut Self::Output`**.

outside the method, after the method call would try to read it, there is no longer any existing value to read. Thusly the compiler prompts the following error **`E0515`** [120]:

```
cannot return reference to temporary value
```

with the additional annotation:

```
returns a reference to data owned by the current function
```

We do not want to publicly expose the inner wrapper data structure encapsulation **`StableCell`**, as it is in principle just a more convoluted and less legible custom variation of the standard option type, so writing an indexing method that returns references to existing values would require redesigning the element space architecture. If we disassemble the data in variants of **`StableCell`** into different, method-borrowable format, we end up with the following changes in the structure:

Existing, non-indexable implementation		*Indexable implementation*
`elems: Vec<StableCell<T,Id>>`	`->`	`elems: Vec<(Option<T>,IdOrNext<Id>)>`
`StableCell::Has(T,Id)`	`->`	`(Some(T),IdOrNext<Id>::Id(Id))`
`StableCell::Not(usize)`	`->`	`(None,IdOrNext<Id>::Next(usize))`

with a new enumeration of the following layout:

```
enum IdOrNext<Id> {
        Id(Id),
        Next(usize),
}
```

With this new data formatting we can now rewrite the required indexing method as follows:

```
fn index (index: StableIndex) -> &Option<T> {
        &self.elems[index.i].0
}

//...

fn index_mut (index: StableIndex) -> &mut Option<T> {
        &mut self.elems[index.i].0
}
```

The usefulness of this implementation is, however, debatable. The syntax for this newly implemented indexing method is nearly identical – even length-wise only a few characters shorter – than using the already provided accessor methods:

```
//the following line could be replaced with 'match stable_list.at(index)'
match stable_list[index] {
	Some(e) => println!("element {e} at {index}"),
	None => println!("no element at {index}"),
}
```

There is arguably no practical advantage to the new indexing method. On the contrary it has reduced maintainability, as we are breaking the data integrity and correctness guarantees of the **`StableCell`** encapsulation. While **`StableCell`**, already through syntax, always guarantees the correctness of its variants, either containing an element **`T`** with its identifier **`Id`** or a freelist index **`usize`**, the new proposed approach is not limited to combinations **`(Some(T),Id(Id))`** and **`(None,Next(usize))`** but can technically also represent invalid states **`(Some(T),Next(usize))`** or **`(None,Id(Id))`**. While the responsibility here lies on the maintainer(s) and it does not itself prove an insurmountable feat to avoid representing these illegal states in the codebase, as has been done since the dawn of software engineering and even more so as the methodologies of design by contract have evolved, it is counterproductive to forsake the safety mechanisms that Rust enumerations, as an algebraic data type, provide implicitly through their syntax in categorically preventing these errors in compile-time static analysis – especially since, as has been repeatedly stated here, this proposed indexing behavior does not offer any advantage over the existing accessor methods.

It has the be noted that the indexing could be made to differ from the accessor methods by having them return pure, non-option-wrapped element values: **`&T`** and **`&mut T`** instead of **`&Option<T>`** and **`&mut Option<T>`**. This requires a simple call to the method **`fn unwrap(self)`** that assumes an **`Option<T>`** to be of the existing type **`Some(T)`**, returning this inner **`T`**:

```
//...
	&self.elems[index.i].0.unwrap()
}

//...
	&mut self.elems[index.i].0.unwrap()
}
```

However, unwrapping a **`None`** value causes a panic, crashing the program. As a general rule for Rust that emphasizes compile time safety, unwrap should only be reserved for prototyping and test cases and such

situations where the developer knows with absolute certainty that the option can never be **`None`** – and even, just for safety, it is recommended to rather not use an unwrap.

Interestingly, the Rust programming culture seems to consider indexing as one of the operations where panicking is an allowed consequence and the responsibility of checking the validity of the index is left to the caller (and, in general, the use of indexing operation to caller's discretion). All Rust standard library collections that implement **`Index`** make a note that attempting to access an element that is not present in the collection results in a panic. The same goes for the third-party collections compared in this thesis.

`StableList` has been primarily designed for powering game loops and allowing safe and efficient cross-iteration. Deliberately providing such functionality to the user of the library that may, if used carelessly, introduce panics to the game engine is deeply against the original design philosophy. Thusly it is a design choice that when accessing individual elements within **`StableList`**, they will *always* have to be pattern-matched to see if there is an element present or not.

Leaning on these principles and all the reasons discussed in this section, **`StableList`** does not provide means of accessing its elements through indexing – and it is highly unlikely it will in future, either.

5.5 Lack of support for idiomatic iteration

As likewise briefly mentioned in beginning of *3 StableList public interface*, **`StableList<T,Id>`** does not – at least at the moment – support iteration in the manner it is usually performed in idiomatic Rust. This limitation has the concrete consequence that instances of **`StableList<T,Id>`** cannot be iterated over through the use of **`for`** loop syntax nor cannot it be iterator-chained to allow for more functional programming approach.

The following exemplary passages of code do not work with **`StableList<T,Id>`**:

```
//outputs all collisions under the radius indicated
//by the constant SIZE
for e1 in stable_list {
	for e2 in e1.others {
		if (e1.inner.x - e2.inner.x).abs() < SIZE
		&& (e1.inner.y - e2.inner.y).abs() < SIZE {
			println!(”collision occurred between {} and {}”,
			e1.id,e2.id);
		}
	}
}
```

```
//collects all enemy locations together with
//their respective access identifiers
//into a Vec
let enemy_locations: Vec<(StableIndex<Enemy>,(i32,i32))> =
stable_list.
filter(|e| e.inner.is_enemy).
map(|e| (e.id,(e.inner.x,e.inner.y))).
collect();
```

Iteration in Rust can usually be applied to custom types by implementing the **`trait Iterator`** for them. Implementing this trait automatically allows the use of both the **`for`** syntax (that is actually syntactic sugar for calling the required method **`fn Iterator::next()`** until it returns **`None`**) and all the **`trait Iterator`**–provided iterator methods, including but not limited to **`filter`**, **`map`** and **`collect`** demonstrated above. Implementing **`trait Iterator`** is fairly simple, at its basis consisting of implementing one required trait method and a type declaration for the output type of the iterator. Often it is also required to implement an auxiliary type to hold the state of the iteration, distinct from the original type that the iterator data originates in.[121]

Problems arise, however, when attempting to iterate over mutably borrowed data in a manner where the same data is borrowed multiple times over the lifetime of the same iterator. The trivial approach to implementing an iterator for **`StableList`** fails exactly because of this:

```
struct StableIter<'a,T,Id: StableId> {
	//holds iterator state
}

//holds similar data that can be accessed
//through the 'fn each(...)' and 'fn each_mut(...)'
//methods
struct StableIterStep<'a,T,Id: StableId> {
	id: StableIndex<T,Id>,
	inner: &'a mut T,
	others: StableAccess<'a,T,Id>,
}

impl<'a,T,Id: StableId> Iterator for StableIter<'a,T,Id> {
	type Item = StableIterStep<'a,T,Id: StableId>;

	fn next (&'a mut self) -> Option<Self::Item> {
		//return current iteration step,
		//advance the iterator
```

```
        }
}
```

The following two lines – the signature of the `trait Iterator` implementation for `struct StableIter`, and a definition of a so-called *associated type* that only exists in the context of traits and their implementations – are the core of the problem:

```
impl<’a,T,Id: StableId> Iterator for StableIter<’a,T,Id> {
        type Item = StableIterStep<’a,T,Id: StableId>;
```

The lifetimes as annotated above bound the lifetime of the iterator `StableIter` and any single iteration step `StableIterStep` produces by that iterator together. While we may get an error message from the compiler with a header declaring a mismatch between our method implementation not being compatible with the trait, what is at the core of this issue is that the above lifetime annotation dictates that any instantiated `StableIterStep` shall have the same lifetime as the `StableIter` it was created from. This means that already the first iteration step is guaranteed to exist as long as `StableIter` continues to exist – and as `StableIterStep` contains a mutable reference to data originating in `StableList`, we can only ever output a single iteration step from the iterator, as outputting any further mutable references to the same data would be a violation of the Rust borrow rules. This renders the iterator unusable in practice, as an iterator that can only ever take one step has very limited use cases.

Modern Rust, ever since the version 1.65 released in November 2022, however, features a concept called ”generic associated lifetimes”, also known by its acronym GAT, that was developed to used to mend these kinds of lifetime issues.[122][123][124]

GAT expands the lifetime syntax to allow it to be added also to associated types in traits. Instead of binding the lifetime `&’a mut self` to the one in the iterator `StableIter<’a,T,Id>`, it is possible to introduce a generic lifetime on the associated type, i.e. `type Item` in this case, and use a `where` clause to impose further restricting parameters on it [125]:

```
impl<'a,T,Id: StableId> Iterator for StableIter<'a,T,Id> {
        type Item<’b> = StableIterStep<'b,T,Id> where Self: ’b;

        fn next<’c> (&’c mut self) -> Option<Self::Item<’c>> {
                //...
        }
}
```

This lifetime algebra unravels as follows:

The method **fn next<’c>** declares a lifetime **’c** that is common to **&’c mut self** and the return type **Option<Self::Item<’c>>**: it is guaranteed that the mutable reference to **self** shall remain valid as long as the the return type **Option<Self::Item<’c>>** exists. In other words: as only one mutable reference **&’c mut self** is allowed to exist at any one time, so is only one **Option<Self::Item<’c>>** allowed to exist at any one time. The associated type definition **type Item<’b> = StableIterStep<’b,T,Id> where Self: ’b** resonates and reinforces this relationship: any type **Self::Item<’b>** has a lifetime **’b** that is outlived by the type implementing type **Self**, i.e. the implementing type **StableIter** is required to exist at least as long as an iterator step. In other words the method’s return type **Option<Self::Item<’c>>** can be written open as **Option<StableStep<’c,T,Id>>** (imagine an ”assignment” or ”application” **’b ← ’c**) with the **’c** being forbidden from ever outliving the iterator itself.[125][126][10]

The problem, however is, that the **trait Iterator** itself does not currently support GAT: the original trait definition for has a simple declaration of **type Item** instead of a GAT-fortified **type Item<’a>**. These are not recognized by the compiler to the same form – as they shouldn’t – so one cannot add a generic associated lifetime **Item<’a>** in the implementation, as the original trait is without it. **trait Iterator** not implementing or allowing generic associated lifetimes within it is a result of the late emergence of GAT in the Rust ecosystem – as they were discussed for years and managed to be successfully implemented only far after the first version of Rust was released, when the public API of **trait Iterator** had already been stabilized.

Thusly there is simply no way to implement **trait Iterator** for **StableList** in a way that would allow cross-iterating over elements in the contemporary versions of Rust. There is discussion of allowing GAT-powered iteration in future versions of Rust [127][128] – if and when these plans are implemented, **StableList** can finally be updated to support idiomatic iteration.

Of course nothing prevents implementing **trait Iterator** for **StableList** in the manner that **for** syntax and the **trait Iterator**–provided methods could be used to single-iterate over the elements, one element at a time – i.e. iteration working to operate in the manner of **fn each(…)** or **fn each_mut(…)** instead of **fn traverse(…)**. However, it is a design choice to not implement it so. **StableList** has been designed primarily for cross-iteration in game loops and similar application domains, and single iteration is only a secondary intended feature. Thus it fits its design philosophy better to implement **trait Iterator** only when it can work for cross-iteration, not first implement it for single iteration and later make a breaking change to repurpose it for cross-iteration.

10 Granted, the algebra of lifetime annotations in Rust is nearly a branch of science of its own, and it is difficult to explain it in non-formal logic without delving excessively deep into its peculiarities. Unfortunately, enumerating over all the intricacies of lifetime annotations is not reasonable here, as at this advanced a level they only concern this one section out of all the content of this thesis.

5.6 *Improvements & alternative implementations*

The technical details, along with the rationale behind them, described in this chapter up until now are the current state of the **StableList** as an implemented library. They are, however, not necessarily intended to represent the ”final truth”, and it should be obvious that different decisions and alternative routes could have been – and could still be – made and taken to construct a data structure solution not externally dissimilar to the one presented here but internally differing in varying extents.

5.6.1 Generic index types

A preferrable future feature for **StableList** would be allowing the index type to be generic as well. Currently **StableList** supports only **usize**. As this corresponds corresponds to 64-bit unsigned integer in contemporary personal computers, compared to smaller integer types it requires more memory for each memory position allocated for the collection compared – as well as for every instantiated index stored elsewhere in the program. The broad range provided by 64-bit integers is rarely needed, as the next-narrower 32-bit unsigned integer is already able to hold more than 4 billion values, and some application domains, e.g. storing video game objects in many cases, would fare well with even narrower element spaces.

This is the case with grid-based real-time strategy game engine under development by the author [129], where each grid cell may or may not contain a **StableIndex** associated with the game object possibly occupying that cell. For the gameplay maps planned for the game, whose size may reach as high as 512×512 grid cells, this translates to 512 · 512 · 64 bits = 2 megabytes of space. However, the amount of game objects at the same time present in the gameplay map is not expected to exceed 65 536 in any gameplay situation[11], so 16-bit indices would be sufficient. This would reduce the required memory to 512 · 512 · 16 bits = 512 kilobytes.

The relative waste of space grows all the more the smaller the contained elements themselves are. For elements with size far exceeding 64 bits the difference even between 16-bit and 64-bit indices becomes non-significant, but if each element is an 8-bit collection of flags that are guaranteed to never exceed 65 536 instances, with 64-bit indices the index will consume $^{64}/_{(64\,+\,8)} \approx 88.9\%$ of the total space allocated for the collection, while with 16 bits only $^{16}/_{(16\,+\,8)} \approx 66.7\%$. Similarly, if each element is an index to some other element in the collection and there are 100 000 of them, the collection will consume ~1.5 megabytes of memory with 64-bit indices, and only 390 kilobytes bits with 16 bits.

11 This would mean a game object residing in every fourth cell even in this size of a map: $^{65\,536\text{ units}}/_{(512\,\cdot\,512\text{ cells})} = {}^{1\text{ unit}}/_{4\text{ cells}}$.

Generic index types for **`StableList`** are in great interest for its further development, but they have not been implemented so far, as there have been other optimizations that have taken higher urgency and prioritization ahead of them.

5.6.2 Priority queue–implemented freelist

One alternative implementation for **`StableList`** is to abandon the current partitioning model with partition-specific freelists and replacing it with an ordered freelist.

In other words, every time a removal takes place, the removed index would be pushed to a priority queue (that Rust implements as a binary heap [130]). For every insertion to **`StableList`** with vacated memory positions the index at the head of the priority queue is popped, and the element is inserted at that memory position.

This approach guarantees that insertions, including reinsertions, always occur at the first, lefternmost vacant memory position in the element space – a so-called *first-free insertion*. The abolition of partitions also means that truncation does not necessarily need to be tied to binary sizes, i.e. halving and doubling the size of the element space, but can be performed at a memory position resolution: the collection can now always truncate to only fit the length until the last existing element.

It is a separate question if this manner of free truncation is feasible, as the last existing element would need to be computed as well. One possibility is to check for the last existing element during each iteration and then truncate, if it is possible, as **`StableList`** is primarily aimed for iteration-heavy applications and iteration is expected to be performed constantly.

Theoretically the combination of first-free insertion with free truncation could lead to more frequent truncations. In comparison the existing implementation of **`StableList`** only guarantees that new elements are inserted into the first, lefternmost *partition*, but the insertion inside these partitions may take place anywhere, only dictated by the positions most recently removed elements. It is still to be noted that a single occupied memory position anywhere later in the element space, no matter how much vacant memory positions precede it, is sufficient to prevent any kind of truncation, just as in the case of the existing **`StableList`** implementation, so the increase of frequency of truncations in real applications or benchmarks that simulate realistic use cases is, as well, an open question at this point.

When considering the described logical changes to functionality of **`StableList`**, its performance costs have to be taken into account. While the existing implementation of **`StableList`** performs insertions in *O(1)* time and removals in amortized *O(1)*, the priority queue employed in this described modified version for freelist

entries has an the worst-case insertion cost at *O(log(n))* and an amortized expected removal cost of *O(1)* ”over a sufficiently large number of pushes […] when pushing elements that are not already in any sorted pattern” [130], with the worst-case amortized removal cost per push at *O(log(n))*.

The modification has even greater impact on the memory requirements of **`StableList`**. For *max* capacity of **`StableList`**, a priority queue of the same *max* capacity has to be reserved. As the indices are stored as **`usize`** that defaults to 64-bit unsigned integers in modern personal computers, this requires a total of $64 \cdot max$ bits of extra space. This additional memory requirement will naturally become relatively less significant if the contained element type itself is larger in memory size, but for smaller elements and higher capacities the modified **`StableList`** may end up wasting considerable amounts of memory in comparison to the existing **`StableList`** implementation.

Priority queue–facilitated first-free insertion also allows implementing a mechanism to skip over vacant memory positions during iteration, similary to **`HopSlotMap`**. This involves repurposing the memory index entry of **`StableCell::Not(usize)`** to indicate the length of a contiguous empty region this vacant memory position is part of, instead of acting as an index to point to the next vacant memory position in the current freelist implementation.

Tracking and updating data for vacant memory regions is simple and can be performed in *O(1)* time. If an element is removed from the collection at position i and it has no adjacent vacant memory positions, the length value x is set to 1. For cases where there already exists an adjacent vacant memory positions, the length value for this newly vacated memory position will be set to $x \leftarrow x_{adjacent} + 1$, where $x_{adjacent}$ is the length value of a single adjacent memory position, be it preceding or following. In this case where the result will be $x > 1$ the other end of the region will also need to be updated. The other end resides at the memory position $i_{other} = elems[i - x]$ for preceding vacant region and at the memory position $i_{other} = elems[i + x]$ for following vacant region. The x_{other} will now be updated to the same value $x_{other} \leftarrow x$. In case the memory position is both preceded and succeeded by vacant regions, we will need to find both the preceding length and the following length at positions $i_{before} = elems[i - x]$ and $i_{after} = elems[i + x]$, sum the lengths of these regions together $x' = x_{before} + 1 + x_{after}$ and set each of the ends to correspond to this value $x_{before} \leftarrow x'$, $x_{after} \leftarrow x'$. x_i can be set to 0 or any other value, as it will never be read, as will be explained. This also explains why only the ends of each and any vacant region has to ever be updated, and the inner memory positions holding outdated length values is not a logical error.

With the priority queue approach, insertion of new elements will always be guaranteed to take place in the first vacant memory position in the collection. This means that when inserting, only the left end of any vacant region will ever be inserted at. When doing so, the collection will have to update the possible following vacant memory position to a value $x_{adjacent} \leftarrow x - 1$, where x is the vacant region length before the

insertion. The same has to be done to the other end of the vacated region to maintain the correct state of the region for future removals: i_{after} = $elems[i + x]$, $x_{after} \leftarrow x - 1$.

When iterating over the collection and encountering a vacant region, i.e. encountering their left end, the iteration will read the length x of it and increment the new iteration index by the amount of steps indicated by it. Thusly all the memory positions delimited by the vacant region length data will be skipped at once and iteration will continue at the next non-vacant position.

Zero-to-one-neighboring removal and subsequent insertion are demonstrated in Figure 5.5, while Figure 5.6 shows two-neighboring removal, also demonstrating the insertion of intervening 0 value that is never read.

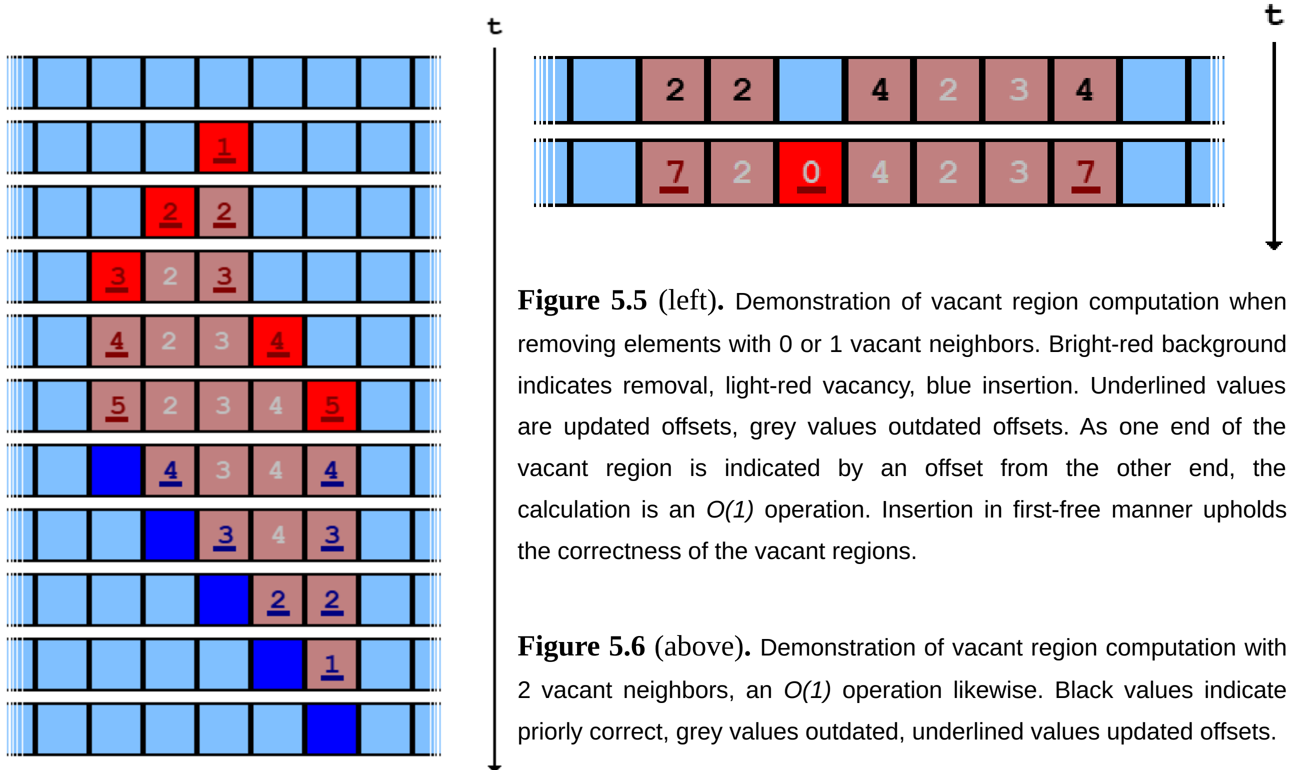

Figure 5.5 (left). Demonstration of vacant region computation when removing elements with 0 or 1 vacant neighbors. Bright-red background indicates removal, light-red vacancy, blue insertion. Underlined values are updated offsets, grey values outdated offsets. As one end of the vacant region is indicated by an offset from the other end, the calculation is an *O(1)* operation. Insertion in first-free manner upholds the correctness of the vacant regions.

Figure 5.6 (above). Demonstration of vacant region computation with 2 vacant neighbors, an *O(1)* operation likewise. Black values indicate priorly correct, grey values outdated, underlined values updated offsets.

The relative performance of the priority queue–powered `StableList`, both with and without the vacant region skipping, cannot be assessed, as these suggested alternatives have not yet been implemented and thus not benchmarked either. It is presumable, though, that they do have their use cases in cases where the collection is expected to face a lot of removals, but these removals cannot be guaranteed to trigger the spontaneous truncations for the partition-based, current `StableList`.

6 Performance comparison

In this chapter, we will analyze the benchmarking data where `StableList` has been compared against all the comparable standard-library and third-party collections in regards of the performance of their operations as per their running times.

In Section 6.1 we will analyze the time performances of i) inserting a number of elements, ii) linearly iterating over all its contained elements, iii) iterating over all its contained elements in a randomized order, and iv) removing all its contained elements in randomized order.

In Section 6.2 we will measure cross-iteration, where for every element in the collection, every other element in the collection is iterated through, and these both are possibly mutated in the process.

In Section 6.3 we analyze the long-term performance when the collection is used to hold so-called ephemeral elements – elements that have some kind of an upper bound – implicit or explicit – associated with them, i.e. they are guaranteed to exist only for a certain, more or less strictly defined maximal period of time.

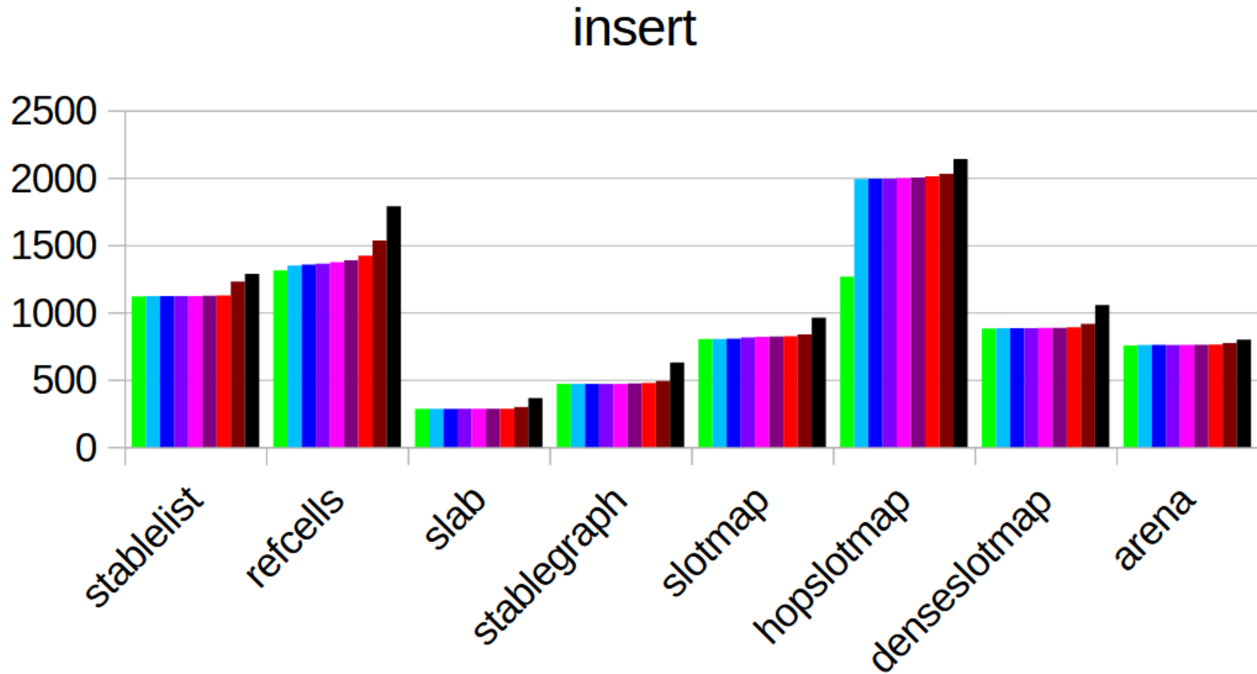

Figure 6.1. 10 000 elements inserted into each collection, repeated 100 times. Represented in the graph are minima and maxima of ranges for 100% (green, black), 95% (cyan, dark-red), 75% (blue, bright-red), 50% (violet, dark-purple) of the measurements from the median, with median value in the middle (bright magenta).

Shown in Figure 6.1, of the 8 collections measured, **`StableList`** is the third-slowest in its insertion running time. It still maintains a considerable margin over the slower solutions: a median of 1100 milliseconds compared to 1370 ms for **`RefCells`** and 2000 ms for **`HopSlotMap`**. The medium-fast solutions **`DenseSlotMap`** and **`Arena`** reside within 760 to 880 ms, while the two fastest solutions, **`Slab`** and **`StableGraph`**, take only 280 and 470 ms, respectively.

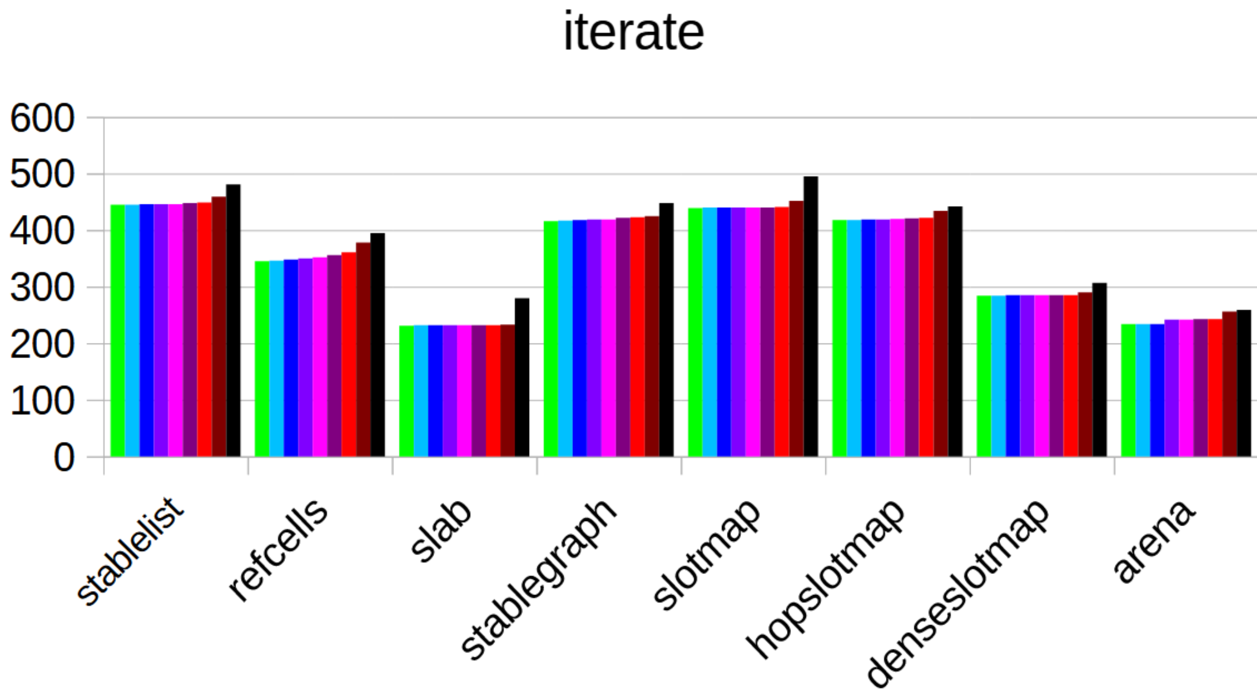

Figure 6.2. Linearly iterating over all 10 000 elements contained in each collection, repeated 100 times. Represented in the graph are minima and maxima of ranges for 100% (green, black), 95% (cyan, dark-red), 75% (blue, bright-red), 50% (violet, dark-purple) of the measurements from the median, with median value in the middle (bright magenta).

Shown in Figure 6.2, of the 8 collections measured, **`StableList`** is roughly on par with **`StableGraph`**, **`SlotMap`** and **`HopSlotMap`** as the slowest alternatives, with medians at 440, 420, 440 and 420 ms, respectively. The faster half of the collections in decreasing running time are **`RefCells`** at 350 ms, **`DenseSlotMap`** at 290 ms, and **`Slab`** and **`Arena`** at 240–230 ms.

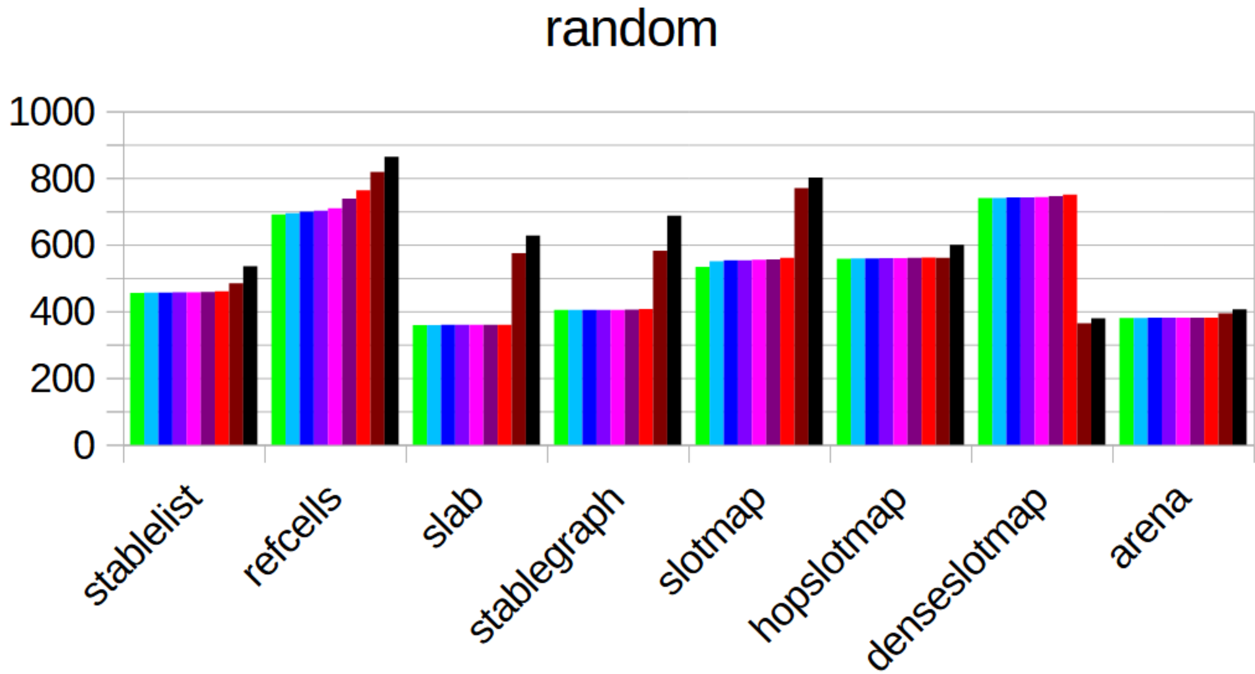

Figure 6.3. Iterating over all 10 000 elements contained in each collection in randomized order, repeated 100 times. Represented in the graph are minima and maxima of ranges for 100% (green, black), 95% (cyan, dark-red), 75% (blue, bright-red), 50% (violet, dark-purple) of the measurements from the median, with median value in the middle (bright magenta).

Shown in Figure 6.3, of the 8 collections measured, **`StableList`** is on the faster half for random-accessing individual elements. With the median of 460 ms with the remaining members of the group of **`Slab`**, **`Arena`** and **`StableGraph`**, ranging from 360 to 405 ms, it retains a noticeable margin to the slower half comprising **`SlotMap`**, **`HopSlotMap`**, **`DenseSlotMap`** and **`RefCells`** in the range of 555–745 ms.

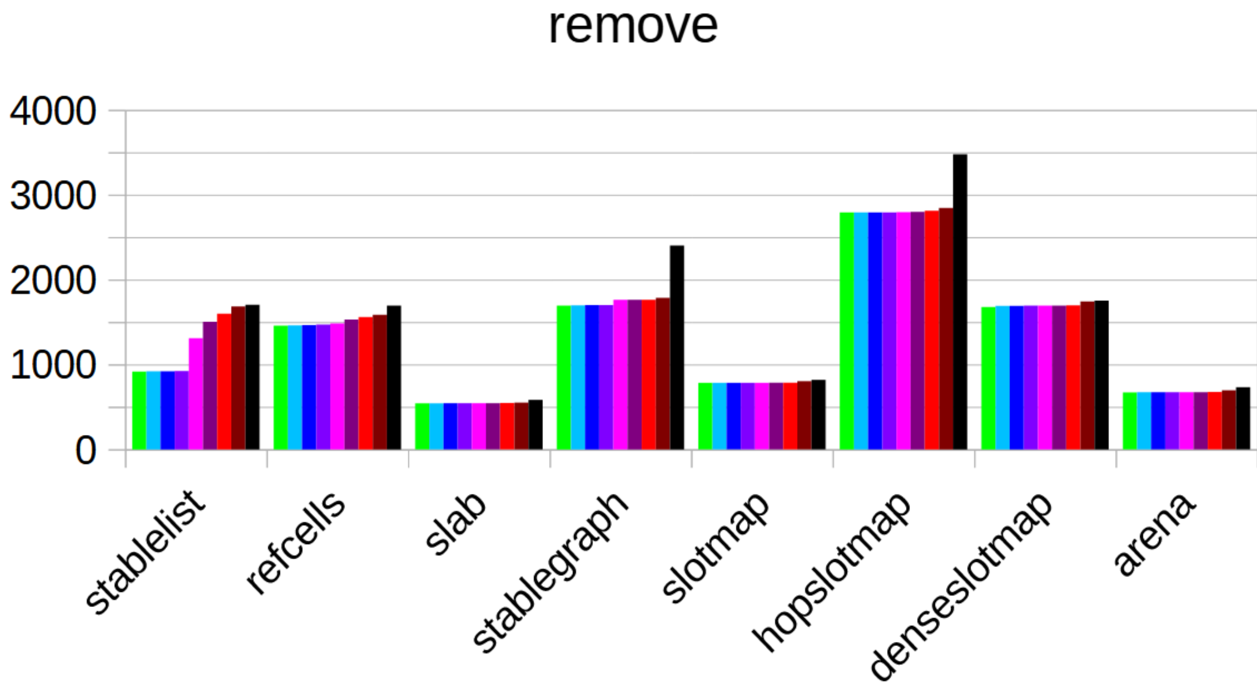

Figure 6.4. Singly removing each 10 000 contained element in each collection in randomized order, repeated 100 times. Represented in the graph are minima and maxima of ranges for 100% (green, black), 95% (cyan, dark-red), 75% (blue, bright-red), 50% (violet, dark-purple) of the measurements from the median, with median value in the middle (bright magenta).

Shown in Figure 6.4, of the 8 collections measured, **`StableList`** resides in the middle for running time for removing individual elements, although with noticeable fluctuations between the lower 50% and higher 50% of its measurements. While the subset of **`Slab`**, **`Arena`** and **`SlotMap`** is significantly faster at 545–785 ms, the mid-subset of **`StableList`**, at 1310 ms, and **`RefCells`**, at 1485 ms, is still significantly faster than the remaining slowest subset of **`HopSlotMap`**, **`DenseSlotMap`** and **`StableGraph`** at 1700–2795 ms.

In summary, **StableList** offers medium-speed insertion, random access and removal. While it is in the slower half for linear iteration, there is no significant variation among this subgroup.

Furthermore, importantly **StableList** does not have any significant outliers or peaks in all of its measurement range: even with its more uneven performance in removal operations, even there the median is rather representative of its average running time, and it does not have any noticeable slowest 5% peaks in any operations, as many of its otherwise stable competitors do. It can be regarded as rather stable for all operations, and it belongs to the category of collections for which median measurement is, in fact, a reliable reference and comparison point. Therefore, it can be summarized that **StableList** is a reliably average-runtime-performing collection.

6.2 Cross-iteration simulation

For the cross-iteration benchmark, an additional auxiliary data type is introduced. The benchmark simulates cross-iteration capabilities by simulating collisions in a two-dimensional space. The data structure **Collider** has two fields: **loc** that indicates its location within the space as a pair of 16-bit integers, and **hit** is a Boolean value that is set to **false** on instantiation and becomes **true** if a collision is computed as having occurred. The collision size d – i.e. the distance within two objects must reside from each other for the collision to be deemed as having occurred – is global in the benchmark, meaning all the **Collider** instances share the same size. In the benchmark setting that produced the data for this thesis, the size was set to $d = 2$, the number of instances generated into the two-dimensional space was 1 000, and each of their (x,y) coordinates were set to randomized values (x_i,y_i), where the values $x_i = [0,1\,000)$ and $y_i = [0,1\,000)$.

Each of the 1 000 **Collider** instances is iterated over all the remaining 999 **Collider** instances. In other words, for each $a \in E$, a is compared to $b \in E \setminus \{a\}$. If $|x_a - x_b| < d$ and $|y_a - y_b| < d|$, $has_hit_a \leftarrow true$ and $been_hit_b \leftarrow true$. This might seem superfluous at first glance, as due to the reciprocal nature of the collision computation presented, for any colliding pair (p,q), $p \neq q$, p, $q \in E$ both of the participants register the $has_hit = true$ on their own iteration turn. However, this setting allows us to enforce double-mutable cross-iteration: it allows us to see in practice if each collection truly supports mutable iteration of (at least) two of its elements at the same time – if they did not, the compiler would prompt an error of violating the rule of single mutable access at a time and not finish compiling the program. This setting also allows to verify the correctness of the setting to a degree, as after the cross-iteration has finished, for all $e \in E$, it is required that $has_hit_e = been_hit_e$.

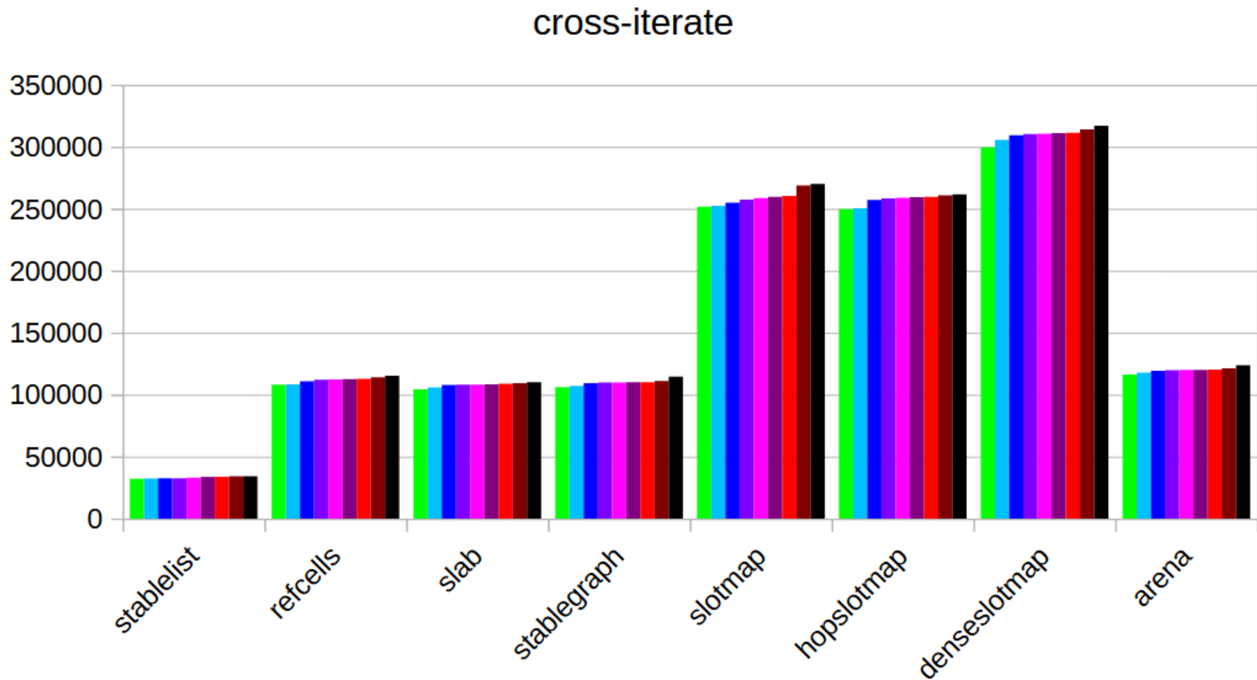

Figure 6.5. Cross-iteration benchmark measurements. Represented in the graph are minima and maxima of ranges for 100% (green, black), 95% (cyan, dark-red), 75% (blue, bright-red), 50% (violet, dark-purple) of the measurements from the median, with median value in the middle (bright magenta).

Figure 6.5 composed of the data collected in the benchmark clearly indicates that **`StableList`** offers unparalleled running time performance compared to any other collection presented here with comparable features. It has very even runtime performance, judging by the whole measurement range, whereas a number of other collections show more variation across the confidence intervals, and its median running time of ~33 000 ms is more than three times faster than the next-fastest competitor at ~110 000 ms – and more than nine times faster than the slowest competitor, at ~311 000 ms.

As a conclusion it can be said that **`StableList`**, as a collection specifically designed for cross-iteration-heavy use cases, such as video games, clearly fulfills its intended purpose successfully.

6.3 Ephemeral data simulation

The ephemeral data in this benchmark is implemented as a custom data type **`Particle`**. Each particle p_i has its remaining time-until-decay value $t(p_i)$ set to randomized positive integer value in the range *[1,50]*. The benchmark is performed over 255 time steps. On each of these time steps a new ephemeral particle is inserted into the collection, while all the existing particles in a collection are iterated over, each of them

updated in the process. On update, each particle's decay time is decremented $t(p_i) \leftarrow t(p_i) - 1$, and if $t(p_i) = 0$, the particle p_i is removed from the collection.

To simulate a sudden peak in ephemeral data – the kind of which may happen in just as well in video game engines as in worker, thread and connection pools under irregular heavy stress – at the timestep $t = 100$, a total of *10 000* additional particles are inserted into the collection at once.

As each particle's $t(p_i)$ is uniformly distributed in the range $[1,50]$, the expected value of $t(p_i)$ for each particle is $(1 + 50)/2 = 25.5$, and as one particle is created on every timestep, this stabilizes the average number of particles present at any timestep in the collection to $n = 25.5$. However, the sudden influx of particles at timestep $t = 100$ rises this momentarily to $n > 10\ 000$. As the upper bound of any particle's $t(p_i)$ is *50* timesteps, all of these mass-inserted particles will have decayed by the timestep *150*, when the average number of particles in the collection is expected to have returned back to $n = 25.5$.

The benchmark measurements, in nanoseconds, also includes the insertion and removal logic of elements, so it reflects a real use case realistically.

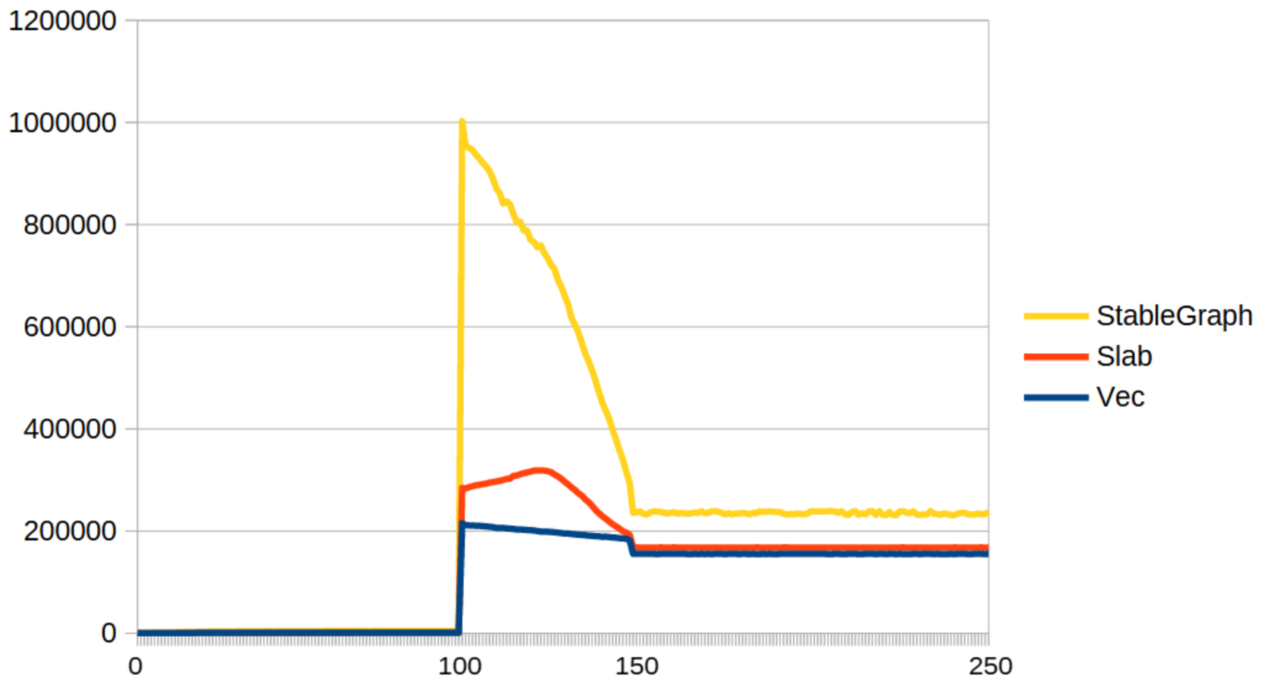

Figure 6.6. Comparison of ephemeral data simulation for Vec, Slab and StableGraph. Vec suffers from ABA problem subcase shift-N and is unsuitable for game object pools. Slab and StableGraph only suffer of ABA problem subcase remove-reinsert, so they are still suitable for game object pools with some extra guard mechanisms in place.

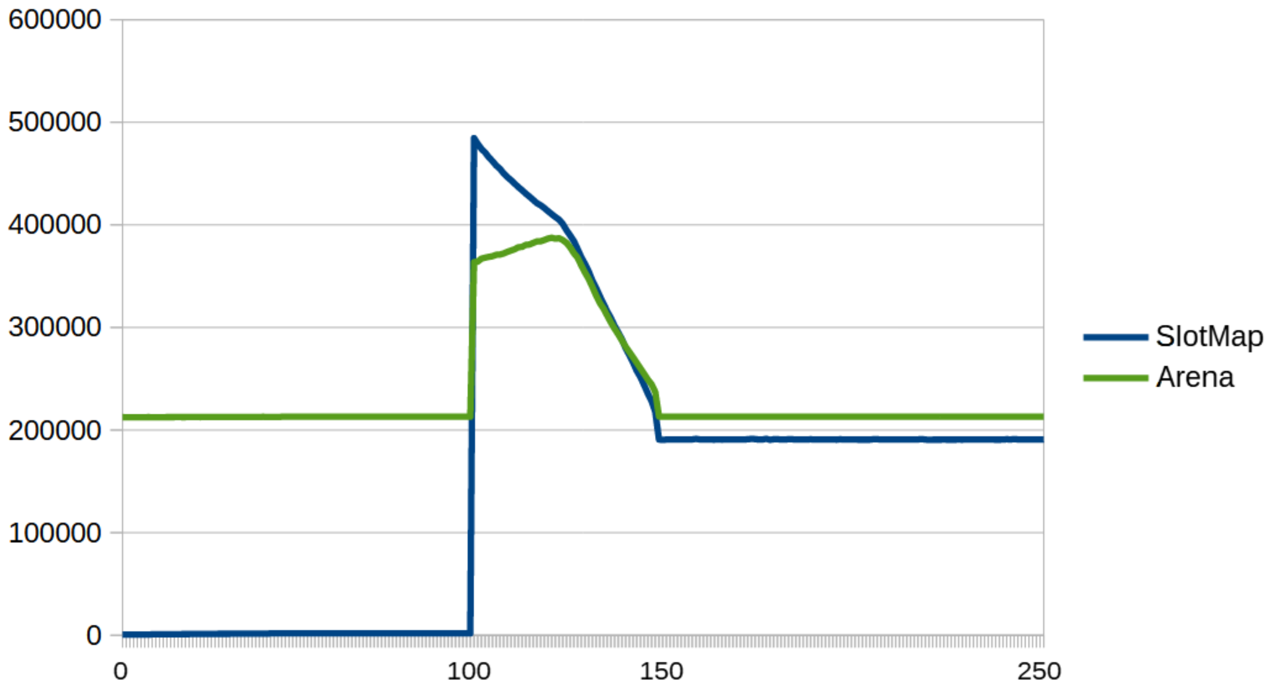

Figure 6.7. A comparison of ephemeral data simulation for SlotMap and Arena. Both of these collections are ABA-immune and are suitable for game object pools without extra guard mechanisms implemented. However, they are unable to restore to their pre-expansion iteration performance, even after the amount of contained elements is reduced. Noticeably the iteration speed of Arena is always lower-bounded by the allocated memory, not by the actual number of previously inserted elements.

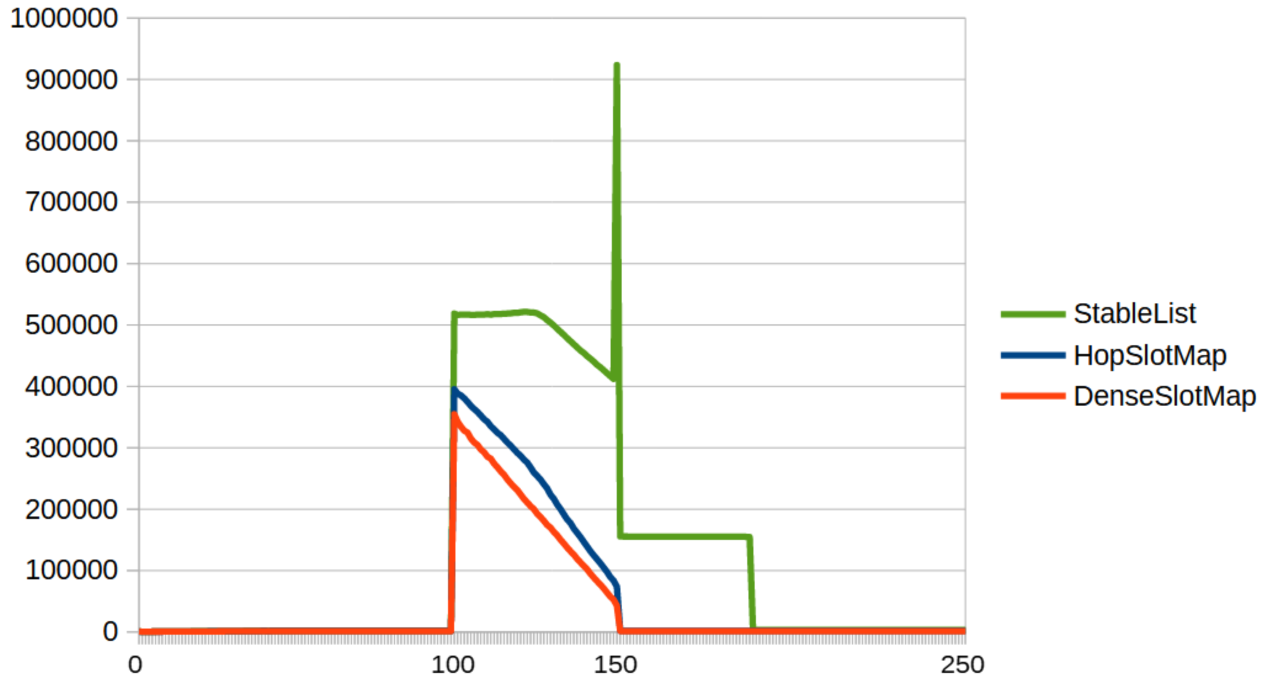

Figure 6.8. A comparison of ephemeral data simulation for StableList, HopSlotMap and DenseSlotMap. All these collections are able to effectively return back to their near-original iteration speed, even after temporarily containing an exceptionally high number of elements. StableList suffers from a temporary peak as a result of the repartitioning logic taking place as part of the element removal.

Among the collections discussed in this thesis, the only collections that are able to return to near-original time performance after the insertion peak has dissipated are `HopSlotMap`, `DenseSlotMap` and `StableList`, as shown by Figure 6.8. The other collections featured in Figures 6.6 and 6.7 suffer from significantly slower post-peak iteration times compared to pre-peak, with the exception of `Arena`, whose iteration speed is slow already before containing a massive amount of elements. This is because it inserts a ”vacant element” into all non-occupied memory positions already at allocation, forcing iteration to always visit them.[131]

`HopSlotMap` achieves this by tracking regions with vacated elements in a similar manner to the alternative `StableList` implementation suggestion described in Section 5.6.2. All vacant cells are skipped on iteration, and vacant region information is updated on every insertion and removal. This makes the insertion and removal take longer time (approximately twice as slow [99]), but it guarantees the minimal amount of cells visited when iterating.

`DenseSlotMap` operates by having two layers of indirection. The first layer is a sparse vector of keys of elements so far inserted into the collection. These cells with keys also contain the index of the payload data in the second layer – which is densely packed. Each time an element is removed from the collection, the

second layer is densely repacked so that no vacant cells remain between occupied cells. This guarantees not only the minimal amount of cells to iterate but also the minimal length of the second layer, as the second layer length always equals the true number of elements $n_{cells(2)} = n_{elements(2)}$. This is true truncation, as in the case of **`StableList`**, and allows for as fast full iteration as a standard-library **`Vec`**.[99] However, the truncation only concerns the second, payload data layer. The first layer storing the key information is never truncated and can only ever grow in size. The two layers of indirection also mean slower random access.[99]

`StableList`, however, truncates its length to half of its current length if it detects that all the cells in the latter half of the collection are vacant. This is done as part of the removal operation, and it will be done repeatedly, until the latter half in the new partitioning is non-vacant or the default or user-defined minimum length of the collection is reached.

The removal operation is of amortized ~$O(1)$ running time, but the local, individual running time is upper-bounded only by the length of the first half – thusly $O(n)$, where n is the number of elements in the collection. This is clearly visible in the **`StableList`**-featuring graph as a high spike occurring at the point of time when the running times afterwards significantly decrease.

`StableList` situating at the midsection of insertion, iteration and removal benchmarks in Section 6.1, not specifically exceling in any of them, reflects in its speed during the highly occupied iteration phase residing near the slower end of the inspected collections. While **`Vec`** performs the fastest due to its very simple architecture, peaking at mere 210 µs, **`Slab`**, **`HopSlotMap`**, **`DenseSlotMap`** and **`Arena`** have their peaks in the range of 300–400 µs. **`SlotMap`** comes close to 500 µs, **`StableList`** is under 550 µs, and **`StableGraph`** exceeds 1000 µs.

However, the margins are not completely intolerable, and as **`StableList`** is one of the only three of the analyzed collections that will return close to its pre-extended running time, it is an excellent choice for implementing pools constantly occupied by and vacated of ephemeral data. While **`HopSlotMap`** and **`DenseSlotMap`** offer better overall performance, with no isolated slowdown peaks, **`StableList`** becomes a viable option if efficient cross-iteration performance is also required of the solution, as **`StableList`** is by far faster than these alternatives in that field.

6.4 Performance analysis conclusion

It is worthy of mention that the benchmarks presented do not feature a case where ephemeral data that is cross-iterated. This was not implemented, as this thesis is a first-of-a-kind introduction of **`StableList`** as a collection, as we only focused on the simplest possible test cases to keep data as easily replicable as possible and to avoid all kinds of edge cases and ambiguous implementation dilemmas.

However, as it has been shown that **`StableList`** offers unparalleled cross-iteration performance, as well as solid ephemeral data performance only comparable to two other collections, it is presumable that **`StableList`** will also fare well – potentially even the best – out of all the collections included in the benchmark.

That said it is still advisable that cross-iterated ephemeral data benchmarks be formed and data be collected in future research to further analyze the relative performances on basis of empirical, concrete data.

It would also serve to produce interesting data to include pure, raw memory pointers into the benchmark. This was not done for this thesis for the same, aforementioned scoping reasons, and it is also worth noting – in the case this kind of benchmarking will eventually be performed – that using pure raw memory pointers abandons all existing runtime safety guarantees that idiomatic Rust is iconic and appreciated for.

7 Conclusion

In this thesis we introduced `StableList` – a novel collection data type implemented in safe Rust that can be employed as a pool of game objects in a video game engine. Compared to other equivalent Rust solutions, standard-library or third-party, `StableList` provides average or above-average performance on its insertion and remove operations, as well as in cases of singly iterating through the collection or random-accessing its elements. Where `StableList` really shines is its efficient cross-iteration over all of its elements where, judging by the benchmark results, it is unrivaled by any of the other alternative solutions.

Aside from video games, `StableList` is likely to find use in other application domains as well, as its capability to implicitly truncate on removal and its tendency on insertion to reoccupy vacated memory positions in such a manner that the internal memory structure proactively favors future truncation. These features make `StableList` especially suitable for cases where a high number of ephemeral elements are managed, i.e. elements that are being constantly spawned and destroyed. Even for cases where the memory space is hugely extended due to a temporarily high number of elements inserted, `StableList` is usually able to restore itself back to a faster iteration speed after these elements have expired, as due to spontaneous truncations likely occurring there are less vacant memory positions that the collection would waste computation on visiting.

Suggestions for future improvements over current `StableList` include generic index types: the current default `usize` type corresponds to 64-bit unsigned integer in contemporary personal computers. Compared to narrower integer representations, this comes at the cost of more bits required not only per each element in the collection but also for every instantiated index stored elsewhere in the program. Such range is rarely needed, as the next-narrowed 32-bit unsigned integer is already able to hold more than 4 billion values, and some application domains, e.g. storing video game objects in many cases, would fare well with mere 16-bit indices, supporting a maximum of 65 536 elements at any time.

The lack of idiomatic iteration is an issue that sets `StableList` apart from many other collections and makes it more difficult to intuitively use at first. However, implementing such iteration for `StableList` in such a fashion that cross-iteration – the intended original purpose of `StableList` – would also be supported is currently impossible, as the Rust approach for implementing idiomatic iterators does not support so-called generic associated lifetimes, at least as of yet. If and when this support is added to the language, implementing idiomatic iteration support for `StableList` is a simple and effortless task.

Section 5.6 introduced us a few possible variations of **`StableList`** internal logic – namely one that replaces element space binary partitioning with a priority queue–supported ascending-order reinsertion, and a further development of this that supports ignoring vacant memory positions during iteration. These are not improvements of **`StableList`**, per se, but rather proposals for alternative implementations. It would serve well to implement these in the future, possibly including them as alternative data types in the same library crate *stablelist*, and collect and assess benchmarking data about their performance.

Assessing the performances of **`StableList`** and all the other comparable collections discussed in the course of this thesis through cross-iterated ephemeral data benchmarks would provide interesting data, as benchmarks so far have concerned cross-iteration and ephemeral data only separately but not in the same context. Similarly, not including raw pointers in the benchmarks was a conscious research decision, as this thesis favors, encourages and attempts to showcase the use of safe Rust–heavy data types, raw pointers conversely being an archetypical example of unsafe Rust. That said, in future research it would likewise be useful to include raw pointers in benchmarks of use cases demonstrated in this thesis – not so much to assess their performance compared to safer alternatives, but to assess the performance of these safer alternatives to them.

References

1. ”Announcing Rust 1.0”. Rust Blog. Accessed: February 5, 2025. <https://blog.rust-lang.org/2015/05/15/Rust-1.0.html>.

2. ”RELEASES.md”. GitHub - rust-lang/rust. Accessed: February 5, 2025. <https://github.com/rust-lang/rust/blob/master/RELEASES.md#version-100-2015-05-15>.

3. J. Blandy & J. Orendorff, *Programming Rust. Fast, Safe Systems Development*. O’Reilly Media, 2017. pp. 1–5.

4. ”Memory Management Glossary: M”. Memory Management Reference. Accessed: February 5, 2025. <https://www.memorymanagement.org/glossary/m.html#term-manual-memory-management>.

5. S. Prata, *C Primer Plus. Sixth Edition*. Addison–Wesley, 2013. pp. 543–551.

6. B. Stroustrup, *Programming Principles and Practice Using C++. Third Edition*. Addison–Wesley, 2024. pp. 439–451.

7. B. Stroustrup, *Tour of C++. Third Edition*. Addison–Wesley, 2022. pp. 78–80.

8. D. Gay, R. Ennals & E. Brewer, ”Safe Manual Memory Management”, presented at the ISMM 2007, October 2007.

9. ”Memory Management Glossary: G”. Memory Management Reference. Accessed: February 5, 2025. <https://www.memorymanagement.org/glossary/g.html#term-garbage-collection>.

10. ”Java Garbage Collection Basics”. Oracle.com. Accessed: February 5, 2025. <https://www.oracle.com/webfolder/technetwork/Tutorials/obe/java/gc01/index.html>.

11. ”Garbage collection”. Microsoft Learn. Accessed: February 5, 2025. <https://learn.microsoft.com/en-us/dotnet/standard/garbage-collection/>.

12. S. Klabnik & C. Nichols, *The Rust Programming Language. Second Edition*. No Starch Press, 2022. pp. 59–83. Digital version: <https://doc.rust-lang.org/book/ch04-01-what-is-ownership.html>.

13. J. Blandy & J. Orendorff, *Programming Rust. Fast, Safe Systems Development*. O’Reilly Media, 2017. pp. 71–89.

14. J. Gregory, *Game Engine Architecture. Third Edition*. CRC Press, 2018. pp. 430–431.

15. B. Stroustrup, *Tour of C++. Third Edition*. Addison–Wesley, 2022. pp. 167–168.

16. B. Kenwright, ”Fast Efficient Fixed-Size Memory Pool”, presented at the Computation Tools 2012, July 2012.

17. F. Ahmed, M. Zia, H. Mahmood & S. Al Kobaisi, ”Open source computer game application. An empirical analysis of quality concerns”. *Entertainment Computing*, vol. 21, pp. 1–10, 2017. <https://www.sciencedirect.com/science/article/abs/pii/S1875952117300290>.

18. M. Overmars, ”Game Maker Tutorial: Designing Good Games”, YoYo Games, 2007. <https://web.cs.wpi.edu/~gogo/courses/imgd1001_2012a/readings/Overmars_GoodGames.pdf>.

19. J. Gregory, *Game Engine Architecture. Third Edition*. CRC Press, 2018. pp. 8–10.

20. S. Madhav, *Game Programming Algorithms and Techniques. A Platform-Agnostic Approach*. Addison-Wesley, 2013. p. 93.

21. J. Gregory, *Game Engine Architecture. Third Edition*. CRC Press, 2018. CRC Press. pp. 11–12.

22. J. Gregory, *Game Engine Architecture. Third Edition*. CRC Press, 2018. CRC Press. pp. 525–528.

23. S. Madhav, *Game Programming Algorithms and Techniques. A Platform-Agnostic Approach*. Addison-Wesley, 2013. p. 4–12.

24. ”ECS”. Are we game yet? Accessed: February 5, 2025. <https://arewegameyet.rs/ecosystem/ecs/>.

25. ”Rust”. Rust Programming Language. Accessed: February 5, 2025. <https://www.rust-lang.org/>.

26. W. Budgen & A. Alahmar, ”Rust: The Programming Language for Safety and Performance”, presnted at the IGSCONG’22, June 2022. <https://arxiv.org/pdf/2206.05503>.

27. V. Romeo, ”Analysis of entity encoding techniques, design and implementation of a multithreaded compile-time Entity-Component-System C++14 library”, M.Sc. thesis, MIFT, UniME, Messina, Italy, 2016.

28. S. Madhav, *Game Programming Algorithms and Techniques. A Platform-Agnostic Approach*. Addison-Wesley, 2013. p. 13–15.

29. ”Struct Vec”. The Rust Standard Library. <https://doc.rust-lang.org/std/vec/struct.Vec.html>.

30. ”Vectors”. Rust by Example. Accessed: February 5, 2025. <https://doc.rust-lang.org/rust-by-example/std/vec.html>.

31. S. Klabnik & C. Nichols, *The Rust Programming Language. Second Edition*. No Starch Press, 2022. pp. 142–147. Digital version: <https://doc.rust-lang.org/book/ch08-01-vectors.html>.

32. J. Blandy & J. Orendorff, *Programming Rust. Fast, Safe Systems Development*. O’Reilly Media, 2017. pp. 360–374.

33. ”Module collections”. The Rust Standard Library. Accessed: February 5, 2025. <https://doc.rust-lang.org/std/collections>.

34. J. Blandy & J. Orendorff, *Programming Rust. Fast, Safe Systems Development*. O’Reilly Media, 2017. p. 60.

35. ”Learn Rust With Entirely Too Many Linked Lists”. https://rust-unofficial.github.io. Accessed: February 5, 2025. <https://rust-unofficial.github.io/too-many-lists/#an-obligatory-public-service-announcement>.

36. ”std::vector”. C++ Reference. Accessed: February 5, 2025. <https://en.cppreference.com/w/cpp/container/vector>.

37. ”Class ArrayList<E>”. Java® Platform, Standard Edition & Java Development Kit Version 23 API Specification. Accessed: February 5, 2025. <https://docs.oracle.com/en/java/javase/23/docs/api/java.base/java/util/ArrayList.html>.

38. ”List<T> Class”. Microsoft Learn. Accessed: February 5, 2025. <https://learn.microsoft.com/en-us/dotnet/api/system.collections.generic.list-1>.

39. S. Klabnik & C. Nichols, *The Rust Programming Language. Second Edition*. No Starch Press, 2022. p. 120. Digital version: <https://doc.rust-lang.org/book/ch07-01-packages-and-crates.html>.

40. ”Crates”. Rust by Example. Accessed: February 5, 2025. <https://doc.rust-lang.org/rust-by-example/crates.html>.

41. J. Blandy & J. Orendorff, *Programming Rust. Fast, Safe Systems Development*. O’Reilly Media, 2017. p. 161.

42. ”The GNU C Reference Manual”. GNU Operating System. Accessed: February 5, 2025. <https://www.gnu.org/software/gnu-c-manual/gnu-c-manual.html#Pointers>.

43. ”Pointer declaration”. C++ Reference. Accessed: February 5, 2025. <https://en.cppreference.com/w/cpp/language/pointer>.

44. V. van der Veen et al, ”Memory Errors: The Past, the Present, and the Future”, The Network Institute, VU University, Amsterdam, Netherlands & Royal Holloway, University of London, London, UK, Tech. Rep. IR-CS-73, 2012. <https://www.isg.rhul.ac.uk/sullivan/pubs/tr/technicalreport-ir-cs-73.pdf>.

45. S. Nagarakatte, M. M. K. Martin & S. Zdancewic, ”Watchdog: Hardware for Safe and Secure Manual Memory Management and Full Memory Safety”, presented at ISCA 2012, June 2012. <https://www.cis.upenn.edu/~stevez/papers/NMZ12.pdf>.

46. C. Cimpanu, ”Microsoft: 70 percent of all security bugs are memory safety issues”. ZDNET. Accessed: February 5, 2025. <https://www.zdnet.com/article/microsoft-70-percent-of-all-security-bugs-are-memory-safety-issues>.

47. S. Klabnik & C. Nichols, *The Rust Programming Language. Second Edition*. No Starch Press, 2022. p. 118. <https://doc.rust-lang.org/book/ch06-03-if-let.html#summary>.

48. K. Ferdowsi, Kasra, ”The Usability of Advanced Type Systems: Rust as a Case Study”, UC San Diego, USA, 2023. <https://arxiv.org/pdf/2301.02308>.

49. ”RAII”. C++ Reference. Accessed: February 5, 2025. <https://en.cppreference.com/w/cpp/language/raii>.

50. B. Stroustrup, *C++ Style and Technique FAQ*. <https://stroustrup.com/bs_faq2.html#finally>.

51. S. Klabnik & C. Nichols, *The Rust Programming Language. Second Edition*. No Starch Press, 2022. pp. 201–213. Digital version: <https://doc.rust-lang.org/book/ch10-03-lifetime-syntax.html>.

52. J. Blandy & J. Orendorff, *Programming Rust. Fast, Safe Systems Development*. O’Reilly Media, 2017. pp. 93–122.

53. J. Blandy & J. Orendorff, *Programming Rust. Fast, Safe Systems Development*. O’Reilly Media, 2017. p. 211.

54. S. Klabnik & C. Nichols, *The Rust Programming Language. Second Edition*. No Starch Press, 2022. pp. 108–110. Digital version: <https://doc.rust-lang.org/book/ch06-01-defining-an-enum.html#the-option-enum-and-its-advantages-over-null-values>.

55. J. Blandy & J. Orendorff, *Programming Rust. Fast, Safe Systems Development*. O’Reilly Media, 2017. pp. 22, 27, 43, 56, 359.

56. InfoQ, QCon, London, UK. *Null References: The Billion Dollar Mistake*. (August 25, 2009). Accessed: February 5, 2025. <https://www.infoq.com/presentations/Null-References-The-Billion-Dollar-Mistake-Tony-Hoare/>.

57. J. Blandy & J. Orendorff, *Programming Rust. Fast, Safe Systems Development*. O’Reilly Media, 2017. pp. 56, 72.

58. S. Klabnik & C. Nichols, *The Rust Programming Language. Second Edition*. No Starch Press, 2022. pp. 420–423. Digital version: <https://doc.rust-lang.org/book/ch19-01-unsafe-rust.html#unsafe-rust>.

59. J. Blandy & J. Orendorff, *Programming Rust. Fast, Safe Systems Development*. O’Reilly Media, 2017. pp. 525–528, 538–557.

60. V. Lepetic, *Principles of Mathematics. A Primer.* John Wiley & Sons, 2016. pp. 44–46.

61. S. Madhav, *Game Programming Algorithms and Techniques. A Platform-Agnostic Approach*. Addison-Wesley, 2013. p. 147–148.

62. C. Ericson, *Real-time Collision Detection*. Elsevier, 2005. p. 14.

63. D. Dechev, P. Pirkelbauer & B. Stroustrup, ”Understanding and Effectively Preventing the ABA Problem in Descriptor-based Lock-free Designs”, presented at 2010 ISORCW, May 2010. <https://www.stroustrup.com/isorc2010.pdf>.

64. ”Crate slotmap § Why not index a `Vec`, or use `slab`, `stable-vec`, etc?”. Crate slotmap. Accessed: February 5, 2025. <https://docs.rs/slotmap/latest/slotmap/#why-not-index-a-vec-or-use-slab-stable-vec-etc>.

65. ”Crate generational_arena”. Crate generational_arena. Accessed: February 5, 2025. <https://docs.rs/generational-arena/latest/generational_arena>.

66. J. Gregory, *Game Engine Architecture. Third Edition*. CRC Press, 2018. p. 711.

67. Konami. *Gradius*. (1986). Konami. Video game for NES console.

68. Capcom. *Gun.Smoke*. (1988). Capcom. Video game for NES console.

69. Technōs Japan. *River City Ransom*. (1989). Technōs Japan. Video game for NES console.

70. Konami. *Batman Returns*. (1993). Konami. Video game for NES console.

71. Rare. *Battletoads*. (1991). Tradewest. Video game for NES console.

72. Luca "poncle" Galante. *Vampire Survivors*. (2022). Video game for Windows, among other platforms.

73. J. Bolding. ”Here's a free shooter for fans of Vampire Survivors and Brotato”. PC Gamer. Accessed: February 5, 2025. <https://www.pcgamer.com/heres-a-free-shooter-for-fans-of-vampire-survivors-and-brotato/>.

74. S. Madhav, *Game Programming Algorithms and Techniques. A Platform-Agnostic Approach*. Addison-Wesley, 2013. pp. 9–12.

75. J. Gregory, *Game Engine Architecture. Third Edition*. CRC Press, 2018. pp. 534–538.

76. D. R. Hanson, ”Fast allocation and deallocation of memory based on object lifetimes”, *Software: Practice and Experience*, vol. 20, no. 1, pp. 5–12, 1990.

77. ”Crate typed_arena”. Crate typed_arena. <https://docs.rs/typed-arena/latest/typed_arena/>.

78. S. Klabnik & C. Nichols, *The Rust Programming Language. Second Edition*. No Starch Press, 2022. p. 472. Digital version: <https://doc.rust-lang.org/stable/book/ch20-02-multithreaded.html#improving-throughput-with-a-thread-pool>.

79. B. Christudas. ”Query by Slice, Parallel Execute, and Join: A Thread Pool Pattern in Java”. java.net. Accessed: February 5, 2025. <https://web.archive.org/web/20080207124322/http://today.java.net/pub/a/today/2008/01/31/query-by-slice-parallel-execute-join-thread-pool-pattern.html>.

80. D. Carmona. ”Programming the Thread Pool in the .NET Framework”. Microsoft Learn. Accessed: February 5, 2025. https://learn.microsoft.com/en-us/previous-versions/dotnet/articles/ms973903(v=msdn.10)>.

81. ”stable-vec”. crates.io: Rust package registry. Accessed: February 5, 2025. <https://crates.io/crates/stable-vec>.

82. ”slab”. crates.io: Rust package registry. Accessed: February 5, 2025. <https://crates.io/crates/slab>.

83. ”slotmap”. crates.io: Rust package registry. Accessed: February 5, 2025. <https://crates.io/crates/slotmap>.

84. ”generational-arena”. crates.io: Rust package registry. Accessed: February 5, 2025. <https://crates.io/crates/generational-arena>.

85. ”petgraph”. crates.io: Rust package registry. Accessed: February 5, 2025. <https://crates.io/crates/petgraph>.

86. GitHub - LukasKalbertodt/stable-vec. Accessed: February 5, 2025. <https://github.com/LukasKalbertodt/stable-vec>.

87. GitHub - tokio-rs/slab. Accessed: February 5, 2025. <https://github.com/tokio-rs/slab>.

88. GitHub - orlp/slotmap. Accessed: February 5, 2025. <https://github.com/orlp/slotmap>.

89. GitHub - fitzgen/generational-arena. Accessed: February 5, 2025.<https://github.com/fitzgen/generational-arena>.

90. GitHub - petgraph/petgraph. Accessed: February 5, 2025.<https://github.com/petgraph/petgraph>.

91. ”Memory Management Glossary: F”. Memory Management Reference. Accessed: February 5, 2025. <https://www.memorymanagement.org/glossary/f.html#free.list>.

92. ”Struct Vec § Guarantees ”. The Rust Standard Library.<https://doc.rust-lang.org/std/vec/struct.Vec.html#guarantees>.

93. S. Klabnik & C. Nichols, *The Rust Programming Language. Second Edition*. No Starch Press, 2022. pp. 330–343. Digital version: <https://doc.rust-lang.org/book/ch15-05-interior-mutability.html>.

94. J. Blandy & J. Orendorff, *Programming Rust. Fast, Safe Systems Development*. O’Reilly Media, 2017. pp. 205–209.

95. J. Blandy & J. Orendorff, *Programming Rust. Fast, Safe Systems Development*. O’Reilly Media, 2017. pp. 90–92.

96. ”Crate stable_vec”. Crate stable_vec. Accessed: February 5, 2025. <https://docs.rs/stable-vec/latest/stable_vec>.

97. ”Crate slab”. Crate slab. Accessed: February 5, 2025. <https://docs.rs/slab/latest/slab>.

98. ”Crate petgraph”. Crate petgraph. Accessed: February 5, 2025. <https://docs.rs/petgraph/latest/petgraph/index.html>.

99. ”Crate slotmap”. Crate slotmap. Accessed: February 5, 2025. <https://docs.rs/slotmap/latest/slotmap/>.

100. ”hop.rs - source”. Crate slotmap. Accessed: February 5, 2025. <https://docs.rs/slotmap/latest/src/slotmap/hop.rs.html#485-552>.

101. ”dense.rs - source”. Crate slotmap. Accessed: February 5, 2025. <https://docs.rs/slotmap/latest/src/slotmap/dense.rs.html#351-367>.

102. ”Module rc”. The Rust Standard Library. Accessed: February 5, 2025. <https://doc.rust-lang.org/std/rc/index.html>.

103. ”Trait Index”. The Rust Standard Library. Accessed: February 5, 2025. <https://doc.rust-lang.org/std/ops/trait.Index.html>.

104. ”Trait Iterator”. The Rust Standard Library. Accessed: February 5, 2025. <https://doc.rust-lang.org/std/iter/trait.Iterator.html>.

105. ”std::map”. C++ Reference. Accessed: February 5, 2025. <https://en.cppreference.com/w/cpp/container/map>.

106. ”Struct HashMap”. The Rust Standard Library. Accessed: February 5, 2025. <https://doc.rust-lang.org/std/collections/struct.HashMap.html>.

107. S. Klabnik & C. Nichols, *The Rust Programming Language. Second Edition*. No Starch Press, 2022. pp. 192–201. Digital version: <https://doc.rust-lang.org/book/ch10-02-traits.html#trait-bound-syntax>.

108. ”Bounds”. Rust by Example. Accessed: February 5, 2025. <https://doc.rust-lang.org/rust-by-example/generics/bounds.html>.

109. J. Blandy & J. Orendorff, *Programming Rust. Fast, Safe Systems Development*. O’Reilly Media, 2017. pp. 303–304.

110. ”Struct PhantomData”. The Rust Standard Library. Accessed: February 5, 2025. <https://doc.rust-lang.org/std/marker/struct.PhantomData.html>.

111. ”Trait Copy § When can’t my type be Copy?”. The Rust Standard Library. Accessed: February 5, 2025. <https://doc.rust-lang.org/std/marker/trait.Copy.html#when-cant-my-type-be-copy>.

112. K. Davis, B. Peabody & P. Leach, ”RFC 9562 - Universally Unique IDentifiers (UUIDs)”, Internet Engineering Task Force, 2024. <https://datatracker.ietf.org/doc/html/rfc9562>.

113. S. Klabnik & C. Nichols, *The Rust Programming Language. Second Edition*. No Starch Press, 2022. pp. 103–107. Digital version: <https://doc.rust-lang.org/book/ch06-01-defining-an-enum.html>.

114. ”Enums”. Rust by Example. Accessed: February 5, 2025. <https://doc.rust-lang.org/rust-by-example/custom_types/enum.html>.

115. J. Blandy & J. Orendorff, *Programming Rust. Fast, Safe Systems Development*. O’Reilly Media, 2017. pp. 211–215.

116. K. Knopp, *Theory and Application of Infinite Series*. Blackie & Son, 1954. pp. 111–112.

117. S. Klabnik & C. Nichols, *The Rust Programming Language. Second Edition*. No Starch Press, 2022. pp. 77–83. Digital version: <https://doc.rust-lang.org/book/ch04-03-slices.html>.

118. ”Module slice”. The Rust Standard Library. Accessed: February 5, 2025. <https://doc.rust-lang.org/std/slice/>.

119. ”Struct Vec § pub fn split_at_mut(&mut self, mid: usize) -> (&mut [T], &mut [T])”. The Rust Standard Library. Accessed: February 5, 2025. <https://doc.rust-lang.org/std/vec/struct.Vec.html#method.split_at_mut>.

120. ”Error code E0515”. Rust error codes index. Accessed: February 5, 2025. <https://doc.rust-lang.org/error_codes/E0515.html>.

121. ”Module iter § Implementing Iterator”. The Rust Standard Library. Accessed: February 5, 2025. <https://doc.rust-lang.org/std/iter/index.html#implementing-iterator>.

122. ”What's in 1.65.0 stable § Generic associated types (GATs)”. Rust Blog. Accessed: February 5, 2025. <https://blog.rust-lang.org/2022/11/03/Rust-1.65.0.html#generic-associated-types-gats>.

123. ”The push for GATs stabilization”. Rust Blog. Accessed: February 5, 2025. <https://blog.rust-lang.org/2021/08/03/GATs-stabilization-push.html>.

124. ”Feature Name: generic_associated_types”. The Rust RFC Book. Accessed: February 5, 2025. <https://rust-lang.github.io/rfcs/1598-generic_associated_types.html>.

125. ”Required Bounds”. Generic Associated Types Initiative. Accessed: February 5, 2025. <https://rust-lang.github.io/generic-associated-types-initiative/explainer/required_bounds.html>.

126. ”GATs: Decide whether to have defaults for where Self: 'a”. GitHub - rustlang/rust. Accessed: February 5, 2025. <https://github.com/rust-lang/rust/issues/87479>.

127. ”Why GATs?”. Generic Associated Types Initiative. Accessed: February 5, 2025. <https://rust-lang.github.io/generic-associated-types-initiative/explainer/motivation.html>.

128. ”Defining and implementing the Iterable trait with GATs”. Generic Associated Types Initiative. Accessed: February 5, 2025. <https://rust-lang.github.io/generic-associated-types-initiative/shiny_future/iterable.html>.

129. Wolvenpit Studio. *Far Dawn*. (unpublished). Video game for Windows and Linux under development. <https://wolvenpit.com/fardawn>.

130. ”Struct BinaryHeap”. The Rust Standard Library. Accessed: February 5, 2025. <https://doc.rust-lang.org/std/collections/struct.BinaryHeap.html>.

131. ”lib.rs - source”. Crate generational_arena. Accessed: February 5, 2025. <https://docs.rs/generational-arena/latest/src/generational_arena/lib.rs.html#799-832>.