Rust and Linked Lists

Martin Sivák, Lukáš Častven
Slovenská technická univerzita v Bratislave
Fakulta informatiky a informačných technológií
xcastven@stuba.sk

1 May 2023

Contents

1 Idea

Linked lists are data structures heavily used for explaining key computer science concepts to new students in this field and are very popular in academic environments. They are trivial to implement and have good properties, at least in theory.

However when you want to implement them in Rust, the triviality is quickly lost, and as we learned, in the real world, linked lists generally suck (there are few cases when they can be good, but these are rare).

2 Requirements

Learn how to implement Linked List

Based on feedback from our tutors, we learned that implementing linked lists in Rust isn't as trivial as in other languages. This is due to a fact that Rust likes all of the memory to have one clear owner [1]. With this statement we mean safe Rust likes that, of course we can always use unsafe Rust, but then we don't have the benefits that safe Rust gives us.

There is even a whole book about linked lists implementations in Rust. This book walks through different kinds of linked lists and shows the reader how to implement them in Rust, and also shows the reader how hard it can be to write these lists in safe Rust. [2]

Implement different kinds

After we understand why a basic linked list is one of the Rusts worst nightmares, we will implement different kinds of them, and those are:

1. Basic Linked List
2. Immutable Linked List
3. Thread safe immutable Linked List
4. And the king of them all **Doubly** Linked List

Benchmark them in different scenarios

After our brains have melted from different intricacies of linked lists in Rust, we will use Rusts benchmarking library Criterion [3] to see how our implementations compare in various scenarios against the mighty Vec and Rusts standard library's Linked List.

3 How it went

Learning how to implement linked list

After the call with our tutors when we decided that we will implement linked lists in Rust we still didn't believe that it can be that hard.

But we were horribly wrong. We both started reading the book mentioned before, and realized that Rusts borrow checker really likes for memory to have one clear owner.

Our learning went something like this

- Hmm, don't put everything on stack, put it in $Box < T >$
- Why can't we move the value? Oh the $Box < T >$ owns it...
- But wait, what if the element is the end of the list?
- Define an enum $Node < T >$ which can be inner element, tail, or nothing.
- Null pointer optimization? Oh we understand, so redefine $Node < T >$ to be an element or nothing.
- Did we just reimplement the $Option < T >$? Yes we did...
- Ok, now set the next as the previous head. What is this **cannot move out of borrowed content**?

- **mem::replace**? Sweet Python where are you...
- *Option.take*()? That makes sense (Dunning-Kruger Peak of Mount Stupid)
- Iterators, yeah we want to iterate over our list.
- Lifetime?! How are we supposed to know how long this thing will live?!
- Pff, easy, it's just syntactic sugar, why is it needed?
- That's why it is needed... *Fall down to Dunning-Kruger Valley of Despair*

Implementing different kinds

After few weeks of reading, testing and finally understanding that there must be only one owner of the memory, but the owners can have different ways of how to borrow their memory, we implemented these kinds of linked list and gained new knowledge:

1. Linked List: Box is a heap allocation which owns the data
2. Immutable Linked List: Rc also owns the value but, it has a primitive garbage collection principle of counting how many references are pointing at it
3. Thread safe immutable Linked List: Arc is basically the same thing as Rc, but it is atomic, which means it is safe to use in multithreaded environments
4. Doubly Linked List: RefCell, as we understood it, basically allows you to apply borrow checkers rules in runtime, however if they are broken, the program panics

4 Benchmarking

During the implementation of the linked lists we learned from various sources that linked lists, in majority of scenarios suck [2]. Vecs are faster and are easier to work with.

We have two types of benchmarks, first show us why generally linked lists suck, and the second one is when they don't suck. To determine if they suck (or don't) they will be compared to a Vec in same scenario.

Linked Lists Suck

In this table we can see how long it took to push 100000 integers into a Vec, std::collections::LinkedList, Linked List, Immutable Linked List, Thread Safe Linked List and Doubly Linked List.

Table 1: Push elements

Collection type	Time
Vec	104.36 µs
Std LL	1.3043 ms
LL	1.2673 ms
Immutable LL	1.4542 ms
Thread Safe LL	3.6213 ms
DLL	2.5458 ms

In this table we can see how long it took to pop 100000 integers from a Vec, std::collections::LinkedList, Linked List, Immutable Linked List, Thread Safe Linked List and Doubly Linked List.

Table 2: Pop elements

Collection type	Time
Vec	17.773 µs
Std LL	693.14 µs
LL	663.92 µs
Immutable LL	794.60 µs
Thread Safe LL	2.7304 ms
DLL	1.5427 ms

In this table we can see how long it took to get element in the middle of a Vec, Linked List, Immutable Linked List, Thread Safe Linked List with 100000 elements. Std::collections::LinkedList and our Doubly Linked List don't support this operation

Table 3: Pop elements

Collection type	Time
Vec	190.43 ps
LL	102.73 µs
Immutable LL	156.59 µs
Thread Safe LL	149.29 µs

When Linked Lists don't suck

In this table we can see how long it took to push 100 $[u64; 10000]$ into a Vec, std::collections::LinkedList, Linked List, Immutable Linked List, Thread Safe Linked List and Doubly Linked List. [1]

Table 4: Pop elements

Collection type	Time
Vec	537.76 µs
Std LL	464.61 µs
LL	474.34 µs
Immutable LL	627.81 µs
Thread Safe LL	782.17 µs
DLL	1.1396 ms

In this table we can see how long it took to pop 100 $[u64; 10000]$ from a Vec, std::collections::LinkedList, Linked List, Immutable Linked List, Thread Safe Linked List and Doubly Linked List.

Table 5: Pop elements

Collection type	Time
Vec	656.30 µs
Std LL	469.21 µs
LL	455.28 µs
Immutable LL	2.7239 µs
Thread Safe LL	320.21 µs
DLL	917.19 µs

In this table we can see how long it took to get element in the middle of a Vec, Linked List, Immutable Linked List, Thread Safe Linked List with 100 $[u64; 10000]$ elements. Std::collections::LinkedList and our Doubly Linked List don't support this operation

Table 6: Pop elements

Collection type	Time
Vec	184.92 ps
LL	27.667 ns
Immutable LL	70.145 ns
Thread Safe LL	70.208 ns

[1] All of these times also include the time of cloning the whole array, because when we wanted to create all to be inserted arrays, we got stack overflow error

5 Conclusion

When we were choosing what to do for our project, we laughed at the suggestion of implementing a linked list. After a call with our tutors, in which they explained to us why linked lists in Rust aren't trivial we choose this topic thinking that they are just fooling us. Well after couple of weekends fighting with the borrow checker, we now understand why linked lists and Rusts safety don't mix well together.

As we learned during the implementation, linked lists aren't that useful (except in some rare scenarios). And we showed benchmarks proving that.

Thanks to this we learned, in more depth, how we can differently share references in more complex scenarios. Such as in doubly linked list, which contains many small cycles. Also we learned how to use criterion with more complicated benchmarks, and how to tune the benchmark environment for more accurate results.

References

[1] "Why writing a linked list in safe Rust is so damned hard — Hacker News — news.ycombinator.com." https://news.ycombinator.com/item?id=16442743#:~:text=Yeah%2C%20a%20doubly%2Dlinked%20list,first%20program%20you%20might%20choose. [Accessed 1-May-2023].

[2] "Introduction - Learning Rust With Entirely Too Many Linked Lists — rust-unofficial.github.io." https://rust-unofficial.github.io/too-many-lists/index.html. [Accessed 1-May-2023].

[3] "criterion - Rust — docs.rs." https://docs.rs/criterion/latest/criterion/. [Accessed 1-May-2023].