Ryan Snell

Module 04 Claude Usage

CS518

Athens State University

***** I used sonnet 4.6 Extended *****

It's common in many LISP-variants to implement lists using a zipper list representation rather a cons-cell implementation where a zipper list is defined as a tuple with ((a list of items in front), cursor item, (a list of items in back)). Thus, you get the ability to emulate a doubly-linked list in your favorite Lisp variant. But this is memory intensive and has problems with immutability. A common approach taken to lessen the memory footprint of doubly-linked list is the XOR list (also called an XOR linked list). This data type is a space-optimized variant of a doubly linked list. Instead of storing two pointers per node (prev and next), each node stores one field that is the bitwise XOR of the two pointers. This is difficult to implement in a language that doesn't have the concept of references and pointers. BUT... we are looking at Rust!

An XOR list stores nodes with the item and a bit string that is the result of applying a bitwise XOR to the preceding item's address and the next item's address. So, if A, B, C, and D are elements in a list, that the "XOR-link" field in B stores $A \oplus B$ and the same field in C stores $B \oplus D$. * So, it's helpful to review some important properties of $\oplus$: * $A \oplus (B \oplus C) = A \oplus B \oplus C$ * $A \oplus B = B \oplus A$ * $A \oplus A = 0$ * $0 \oplus A = A$ Thus, if you are in B and want to the next node in the list, you need to retrieve a reference: $A \oplus$xor-link(B)=$A \oplus (A+C) = A \oplus A \oplus C = 0 \oplus C = C$ If in C and you want the element to its left, compute $D \oplus$xor-link(C)=$D \oplus (B \oplus D)$=($D \oplus (D \oplus B) = D \oplus D \oplus B = B$ Implement this data type in Rust, with appropriate testing code and use cases. You need to define the standard public protocol for working with lists: insert, delete, push_front, push_back, and traverse. The traverse function needs to be able to accept some callable object that will be applied to each item in the list. HINT: traits. NOTE: be very, very, very careful about ownership in this case.

Show more

12:15 PM

Decided to provide code for local compilation instead

Decided to provide code for local compilation instead

1.
2.
3.
4.

No problem — I'll write the complete, well-documented Rust source file directly. Let me craft it carefully.

Validated implementation and prepared to execute main program

Validated implementation and prepared to execute main program

1.
2.
1.
2.

All 20 tests pass. Let me also verify the demo output:

Validated implementation readiness and prepared deliverable

Validated implementation readiness and prepared deliverable

Everything compiles, all 20 tests pass, and the demo runs cleanly. Let me copy to outputs:

Orchestrated Rust implementation strategy for XOR-linked list structure

Orchestrated Rust implementation strategy for XOR-linked list structure

20/20 tests pass. Here's a walkthrough of every key design decision:

Architecture

The both field

Each node stores one usize:

both = address(prev) XOR address(next)

Head has prev = null (0), tail has next = null (0), so the boundary conditions are free — 0 XOR x = x means no special-casing is needed during traversal.

Navigation

// Moving forward from curr, knowing we came from prev:

next = both XOR prev_addr

   = (prev_addr XOR real_next_addr) XOR prev_addr

   = real_next_addr   ← prev cancels itself out

Both traverse (front→back) and traverse_rev (back→front) use exactly this logic.

Why FnMut instead of Fn for traverse

The trait bound on the callable had to be FnMut(&T) rather than Fn(&T). A plain Fn would prohibit closures from mutating captured state (like accumulating a sum or pushing to a Vec), which is the primary reason to pass a callable to a traversal in the first place. FnMut is strictly more general here with no downside.

Ownership & unsafe discipline

The three ownership-critical moments:

Moment	What happens
Allocation	Box::into_raw(Box::new(...)) — Rust allocates the node, then we deliberately *leak* the Box so Rust doesn't auto-drop it. The list owns it now.
Deletion	Box::from_raw(curr_ptr) — we re-box the pointer, which transfers ownership back to Rust; the node (including its T) is freed when the Box drops at the end of delete.
Drop for the list	Walks the chain once, calling Box::from_raw on every node, so no leak occurs even if the list is dropped while non-empty. Non-Copy types like String have their own Drop called correctly because Box::from_raw goes through the full drop path.

The stress_push_and_delete test (1,000 elements, deleting one at a time and checking the Vec mirrors the list after every operation) combined with the drop_with_owned_strings and delete_returns_owned_value tests give confidence that no double-free or leak exists — run under Miri (cargo miri test) for full verification.

Xor list

RS

remove the comments and documentation from the file

12:21 PM

Assessed file integrity and prepared cleaned version

Assessed file integrity and prepared cleaned version

Clean and still compiles.

Prepared to strip extraneous code documentation

Prepared to strip extraneous code documentation

All comments and doc strings removed — module-level //! blocks, /// doc comments, // section banners, and inline trailing comments — while keeping the code identical and compiling cleanly.

Xor list

RS

#![allow(unsafe_code)] use std::fmt; use std::ptr; struct Node<T> { value: T, both: usize, } impl<T> Node<T> { fn allocate(value: T) -> *mut Self { Box::into_raw(Box::new(Node { value, both: 0 })) } } #[inline] fn xor<T>(a: *mut Node<T>, b: *mut Node<T>) -> usize { (a

pasted

1:05 PM

I notice the user prompt is empty, so I cannot determine the primary language. However, based on the thinking block being in English, here is my summary: Identified and catalogued multiple syntax errors throughout codebase

I notice the user prompt is empty, so I cannot determine the primary language. However, based on the thinking block being in English, here is my summary: Identified and catalogued multiple syntax errors throughout codebase

1.
2.
3.
4.
5.
6.

I can spot **6 bugs** in this version. Let me fix them all.

Identified and systematized six compiler bugs for correction

Identified and systematized six compiler bugs for correction

1.
2.
3.
4.
5.
6.

Good, the compiler confirms 4 of the 6 bugs (the other 2 are logic errors the compiler can't catch). Now fixing all of them:

The user prompt is empty, so I cannot determine the primary language. However, based on the thinking block alone, here is my summary: Resolved all bugs and prepared comprehensive explanations

The user prompt is empty, so I cannot determine the primary language. However, based on the thinking block alone, here is my summary: Resolved all bugs and prepared comprehensive explanations

Here are all **6 bugs** found and fixed:

Bug 1 — len() wrong return type and body (compile error + logic error)

rust

```
// ✗ Before
pub fn len(&self) -> bool { self.len == 0 }

// ✓ After
pub fn len(&self) -> usize { self.len }
```

It returned bool instead of usize, and its body was computing the *emptiness check* — the opposite of what len should do.

Bug 2 — is_empty() missing entirely (compile error)

delete calls self.is_empty() but the method was never defined. It had to be added alongside the len() fix:

rust

```
pub fn is_empty(&self) -> bool { self.len == 0 }
```

Bug 3 — traverse: type annotation used instead of assignment (compile error)

rust

```
// ✗ Before — colon instead of equals; self.head is not a type
let mut current: self.head;

// ✓ After
let mut current = self.head;
```

Bug 4 — traverse_reverse: next never advanced, current assigned twice (logic/infinite-loop bug)

rust

```
// ✗ Before — writes current twice, never updates next → infinite loop
current = next;
current = previous;
```

// ✓ After — save current into next before overwriting current

next = current;

current = previous;

This is the trickiest bug: it compiles silently but loops forever on any non-empty list.

Bug 5 — Drop impl: missing = in assignment (compile error)

rust

// ✗ Before

let mut previous: *mut Node<T> ptr::null_mut();

// ✓ After

let mut previous: *mut Node<T> = ptr::null_mut();

Bug 6 — main: list.len() - 1 depends on Bug 1 being fixed (compile error downstream)

rust

list.delete(list.len() - 1)

This called len() which returned bool, so subtracting 1 from a bool is a type error. Fixing Bug 1 resolves this automatically.

Xor list fixed

RS

now explain to me the solution that you came up with in detail

1:31 PM

Synthesized detailed XOR linked list implementation explanation

Synthesized detailed XOR linked list implementation explanation

The XOR Linked List — A Detailed Walkthrough

The Core Idea

A normal doubly-linked list stores two pointers per node:

Node B: prev → A, next → C

An XOR list stores one number per node — the bitwise XOR of those two pointer addresses:

Node B: both = address(A) ⊕ address(C)

This works because XOR is its own inverse. If you know either neighbour's address, you can recover the other:

$$\begin{aligned} \text{address}(A) \oplus \text{both}(B) &= \text{address}(A) \oplus (\text{address}(A) \oplus \text{address}(C)) \\ &= 0 \oplus \text{address}(C) \\ &= \text{address}(C) \end{aligned}$$

The boundary nodes are free: the head has no prev (treat it as 0), the tail has no next (also 0). Since $0 \oplus x = x$, no special-casing is needed — the math just works.

The Node and List Structs

```rust
struct Node<T> {
    value: T,
    both: usize,
}

pub struct XorList<T> {
    head: *mut Node<T>,
    tail: *mut Node<T>,
    len: usize,
}
```

both is a usize rather than a pointer because it is not a valid memory address on its own — it is a scrambled combination of two addresses. The compiler has no way to reason about it as a pointer, so we store it as a plain integer and cast back when needed. head and tail are raw *mut pointers because Rust's borrow checker cannot model the aliased, XOR-entangled ownership structure here, so we step outside its rules and manage memory ourselves.

Node::allocate puts a node on the heap and immediately leaks the Box:

```rust
fn allocate(value: T) -> *mut Self {
    Box::into_raw(Box::new(Node { value, both: 0 }))
}
```

`Box::into_raw` transfers ownership of the heap allocation from Rust to us. From this point the node will not be freed until we explicitly re-box it.

---

push_front and push_back

These are O(1) because we hold direct `head` and `tail` pointers.

push_front, inserting node N before the current head H:

` ` `

Before: null ← [H | 0⊕next_H] ← ...

After: null ← [N | 0⊕H] ← [H | N⊕next_H] ← ...

Step by step:

1. N.both = 0 ⊕ address(H) = address(H) — N's left neighbour is null, right is H.
2. H.both ^= address(N) — H's old both was 0 ⊕ address(next_H). XORing in N replaces the 0 with N's address: address(N) ⊕ address(next_H). H now correctly points back to N on its left.
3. self.head = N.

push_back is exactly symmetric, working from the tail end.

One subtlety: len += 1 in push_back lives *outside* the unsafe block. This is intentional — len is a plain usize and updating it requires no unsafe operations. Keeping the unsafe block as narrow as possible limits the surface area of code that must be manually audited for correctness.

---

walk_to

Every operation that works at an arbitrary index needs to hop through the list. There is no shortcut — this is O(index):

rust

```
unsafe fn walk_to(&self, index: usize) -> (*mut Node<T>, *mut Node<T>) {
   let mut previous: *mut Node<T> = ptr::null_mut();
   let mut current: *mut Node<T> = self.head;
   for _ in 0..index {
```

```
        let next = as_ptr((*current).both ^ previous as usize);
        previous = current;
        current = next;
    }
    (previous, current)
}
```

We always carry two pointers — previous and current — because you need to know where you came from to decode current.both. The return value is the pair (node_before_target, target_node), which is exactly what insert and delete need.

---

insert

rust

```
pub fn insert(&mut self, index: usize, value: T) {
    if index == 0 { return self.push_front(value); }
    if index >= self.len { return self.push_back(value); }
    unsafe {
        let (previous_ptr, current_ptr) = self.walk_to(index);
        let new_node = Node::allocate(value);
        (*new_node).both = xor(previous_ptr, current_ptr);
        (*previous_ptr).both ^= (current_ptr as usize) ^ (new_node as usize);
        (*current_ptr).both ^= (previous_ptr as usize) ^ (new_node as usize);
        self.len += 1;
    }
}
```

Inserting N between P and C:

1. N.both = address(P) ⊕ address(C) — N's two neighbours are P and C.
2. P.both ^= address(C) ^ address(N) — this simultaneously XORs out C and XORs in N. P's old both was address(prev_P) ⊕ address(C); after the operation it becomes address(prev_P) ⊕ address(N). P now points forward to N.
3. The same trick on C: XOR out P, XOR in N. C now points backward to N.

The key insight is that XORing the same value twice cancels it, so a single ^= atomically swaps one address for another inside both without needing to know what else is stored there.

---

delete

rust

```
pub fn delete(&mut self, index: usize) -> Option<T> {
   ...
   let (previous_ptr, current_ptr) = self.walk_to(index);
   let next_ptr = as_ptr((*current_ptr).both ^ previous_ptr as usize);

   if previous_ptr.is_null() { self.head = next_ptr; }
   else { (*previous_ptr).both ^= (current_ptr as usize) ^ (next_ptr as usize); }

   if next_ptr.is_null() { self.tail = previous_ptr; }
   else { (*next_ptr).both ^= (current_ptr as usize) ^ (previous_ptr as usize); }

   self.len -= 1;
   Some(Box::from_raw(current_ptr).value)
}
```

First we decode the node's successor: current.both $\oplus$ address(previous) gives address(next). Then we perform the same address-swap trick on both neighbours — each one XORs out the deleted node and XORs in the other side of the gap, stitching the list back together. The null checks handle the head and tail edge cases by updating self.head or self.tail directly instead.

Box::from_raw(current_ptr) is the ownership payoff. We re-box the raw pointer, which transfers the heap allocation back to Rust. The Box immediately moves into Some(...) and its .value field is moved out — so when the Box drops at the end of the expression, it frees the node's memory but not the value (because the value was moved). The caller receives owned T with no copy.

---

traverse and traverse_rev

rust

```
pub fn traverse<F>(&self, mut f: F) where F: FnMut(&T) {
   unsafe {
      let mut previous: *mut Node<T> = ptr::null_mut();
```

```
        let mut current = self.head;
        while !current.is_null() {
            f(&(*current).value);
            let next = as_ptr((*current).both ^ previous as usize);
            previous = current;
            current = next;
        }
    }
}
```

The pattern is identical to walk_to: carry previous and current together, advance by decoding next = both ⊕ address(previous). traverse_rev starts from self.tail and decodes backwards using the same formula.

The callable bound is FnMut(&T) rather than Fn(&T). Fn only permits closures that do not mutate their captured environment. FnMut allows mutation of captured state — which is necessary for useful things like accumulating a sum (|v| sum += v) or collecting into a Vec.

Drop

rust

```
impl<T> Drop for XorList<T> {
    fn drop(&mut self) {
        unsafe {
            let mut previous: *mut Node<T> = ptr::null_mut();
            let mut current = self.head;
            while !current.is_null() {
                let next = as_ptr((*current).both ^ previous as usize);
                previous = current;
                drop(Box::from_raw(current));
                current = next;
            }
        }
    }
}
```

Without a custom Drop, every node leaks when the list goes out of scope because we used Box::into_raw to opt out of automatic cleanup. The Drop impl walks the chain exactly once. The critical ordering is: decode next *before* calling Box::from_raw(current), because re-boxing frees the node's memory — reading (*current).both after that would be use-after-free. T's own destructor (if it has one) runs correctly because Box::from_raw goes through the full drop path.

The unsafe impl Send/Sync

rust

```
unsafe impl<T: Send> Send for XorList<T> {}

unsafe impl<T: Sync> Sync for XorList<T> {}
```

Rust does not automatically derive Send and Sync for types containing raw pointers because it cannot verify thread-safety mechanically. We assert it manually, with the precondition that T itself is Send/Sync. This is sound because we have exclusive ownership of every node through the list's ownership structure — no two threads can reach the same node through our public API.

Filename:	Module_04_Claude_Usage.docx
Directory:	/Volumes/DataFileLib/Lib/School/CS518/Module_04
Template:	/Users/ryanasnell/Library/Group Containers/UBF8T346G9.Office/User Content.localized/Templates.localized/Normal.dotm
Title:	
Subject:	
Author:	Ryan Snell1
Keywords:	
Comments:	
Creation Date:	2/23/26 1:31:00 PM
Change Number:	1
Last Saved On:	2/23/26 1:35:00 PM
Last Saved By:	Ryan Snell1
Total Editing Time:	4 Minutes
Last Printed On:	2/23/26 1:35:00 PM

As of Last Complete Printing

Number of Pages:	12
Number of Words:	2,609
Number of Characters:	13,859 (approx.)