A casual doubly linked list in Rust

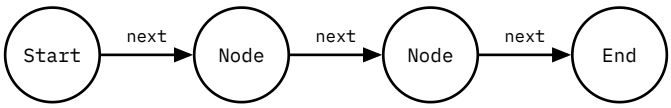
MultisampledNight

2023.05.17

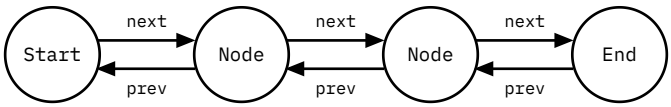
Contents

1 Introduction to doubly linked lists

First of all, let's take a look at singly linked lists, which are one of the simplest data structures to implement. The list holds a pointer to the start node, and each node refers either to the next node in the list, or to a sentinel marking the end of the list:



Doubly linked lists are just like singly linked lists, but each node additionally knows about the *previous* node:



Traditionally, there are two approaches to modeling the end/start of a linked list:

1. Setting the `prev` pointer of the **first** node and the `next` pointer of the **last** node to `NULL`. In this case, the list itself holds nullable pointers to the first and last element in the list, respectively, which are equal if the list contains only one element, and both `NULL` if the list is empty.
2. Setting the `prev` pointer of the **first** node to a sentinel `Head` element, and the `next` pointer of the first node to a sentinel `Tail` element, which both point back if the list contains at least one element, or at each other if the list is empty. In that case, the list always holds a non-nullable pointer to the sentinel `Head` and `Tail` elements.

The second approach requires either subclassing (which could be emulated with traits and vtables in Rust, see Section 2.2) or usage of enumerations, which both enable either a lot of boilerplate or a large tempting possibility for invalid state (for example, using the `Head` node as the `next` pointer for the last element). As a result, the first one was chosen instead, which essentially enables this pseudocode struct layout:

```
struct List<T> {
    start: Option<Node<T>>,
    end: Option<Node<T>>,
    len: usize,
}

struct Node<T> {
    data: T,
    prev: Option<Node<T>>,
```

```
    next: Option<Node<T>>,
}
```

2 Difficulties specific to Rust

2.1 Safe vs. unsafe

In a language like Rust, implementing something basic such as a linked list is more interesting than one might imagine, since the whole language is split into two parts: **safe** and **unsafe**.

Using **safe** Rust, it should be impossible to cause data races, segmentation faults or undefined behavior, which is ensured by banning some operations like sending non-threadsafe structs to a different thread or dereferencing raw pointers. Of course, nothing can be perfect, and Rust will likely never be able to make this "safe sandbox" fully bulletproof, as already demonstrated by the possibilities of raw memory access through appropiate files on Unixalikes or calling a safe function which uses `unsafe` incorrectly. However, sometimes there are legitimate reasons where it's necessary to do very low-level interactions like implementing a collection or performing hardware interaction, where the barricades of safe Rust get in the way.

In these cases, it is possible to switch to **unsafe** Rust by using the `unsafe` keyword in front of a block, which essentially enables superpowers for that block like dereferencing a pointer, writing global unsynchronized variables or interacting with foreign-language-interfaces — this comes at the cost that the compiler cannot check whether or not those usages are valid at all. There are tools like Miri, an interpreter for Rust which checks for undefined behavior, but ultimately it is up to the *programmer* to make sure that the unsafe operations performed in an `unsafe` block are valid and sound. For the linked list here, exactly this was necessary.

2.2 The composite pattern and its fallacies

Essentially, the composite pattern tries to combine several edge cases into a streamlined interface. Creating classes implementing that interface accordingly allows handling of these edge cases, but without having the client code (which only knows of that interface) cluttered with handling of those. In this case, let's say this abstract interface is called `Element`.

In languages like Java, which offer subclassing and an abstract model of objects, this is fairly simple to implement. Just add a parent class `Element` containing all abstractions, and let all classes handling the edge cases be a subclass of `Element`. The client code can then just have a field with the parent class as type, and depending on its needs, it can fill it with an object handling the required edge case.

In Rust however, this becomes slightly more complicated. The only mechanism in Rust which looks like subclassing are *traits*, which are basically Java's interfaces, but slightly more powerful, so let's make a trait `Element` for that common interface. Traits can only contain methods and functions, without being able to require a specific field directly. Most of the time, these traits are only compile-time concepts which are resolved *statically*, but sometimes (like here) it is necessary to resolve those *dynamically*, at runtime. For that purpose, one can create a vtable containing all function pointers alongside of the opaque original struct, specifically by prefixing the trait with `dyn`, as in `dyn Trait`.

Rust's safe references guarantee a valid, well-aligned and never-null pointer behind them, and are split into two variants: **immutable** (a bare `&`) and **mutable** (`&mut`). As long as the referenced struct hasn't been *dropped* (which mostly just means deallocated) yet, there can exist *either* any number of **immutable** references, *or* exactly one **mutable** reference. This is not

fully true, there are cases where immutable references can have a mutable effect, and references can also "stack", but for our purposes this model suffices. Since a doubly linked list requires though that each node is referenced *twice* in total, by each of its surrounding nodes, but one still needs mutable access to it, one cannot use references here. Instead, **raw pointers** are up to the table, usually denoted by `*const T` and `*mut T`, which have zero guarantees about their content. On top of that, `*mut T` is invariant, so what one actually requires here is `NonNull<T>`, which is practically `*mut T`, but covariant and with an additional guarantee of being never `NULL`.

Summing all those up, one ends up with `NonNull<dyn Element<T>>` as type for the pointer. Which is fine, actually. However, since subclassing is not really possible in Rust, now one needs to actually handle each individual edge case **twice** with the same code. `Head` and `Node` both share the same code for inserting a new node **after** the current oen, `Node` and `Tail` both share the same code for inserting a new node **before** the current one. In order to deduplicate that, either one creates a new function for that duplicated code, or one makes the `Element` trait hold only getters and setters for their data, previous and next nodes, and let the list itself perform the actual operations using those methods.

As it turns out, both solutions are quite lengthy while yielding barely any ergonomic difference. Instead, making the pointer `Option<NonNull<Node<T>>>` (note the added `Option`), where `Option::Some(node)` refers to a "real" `node` and `Option::None` to the end of the list actually produces almost the same code as in the second solution, but with all the boilerplate for the sentinel nodes and the `Element` trait removed.