A Linked List Implementation for Ethereum

Coding data structures in Solidity is weird and beautiful.

Alberto Cuesta Cañada

Follow

Coinmonks

--

Listen

Share

A few months ago I implemented a Linked List in Solidity for a client. More recently I decided that I might take on implementing a Fast Limit Order Book in Solidity as a pet project (aren't nerds fun!) and Linked Lists appeared again.

As I recently wrote in an article about when to use different data structures:

Linked Lists are your data structure of choice when you need to preserve the insertion order, and also when you want to insert in arbitrary positions.

I love coding basic building blocks and no one seemed to have done this one*, so I took to it happily.

In this article, I'll introduce an implementation for Singly and Doubly Linked Lists, which you can reuse or modify for your own purposes. All the code is available in GitHub or as an npm package.

*Disclaimer: While writing this article, and after having coded the contracts, I found this earlier implementation from chriseth. Like him, I also considered using an array. Compared to using a mapping, it simplifies the creation of new items but also makes deletion more difficult.

Implementation

For this article I'm going to ignore that Solidity is an Object Oriented Programming Language and code the lists in a single contract. Doing that will allow me to focus on the basics such as data usage. An OOP implementation is possible, but the trade offs deserve an article of its own.

Implementing a Linked List in a single contract Solidity was not an obvious thing to do. This code would be very convenient, but not doable in Solidity since you can't have recursive structs

.

The only dynamic contract variable that exists in Solidity are mappings. Even arrays are mappings under the hood. Given that constraint the best implementation for a Linked List in Solidity that I could come up is based on this:

The Linked List is made of Item

. The Item

has a unique id, a member for the id of another Item

, and an address which is the data payload. Then all items created are stored in a mapping indexed by Item

id.

You can retrieve any Item

at cost O(1) if you know its id, just by looking it up in the mapping. If you are looking at an Item

in the List and want to proceed to the next one you have to retrieve item.next which is an Item

id, and then look up that next Item

in the mapping.

If this gets you confused, don't feel bad. I also got very confused. My first question was "is there any point to a Linked List if you can just arbitrarily retrieve any Item

from a list?”.

The question is that yes, there is a point, but with a very limited scope. Linked Lists are your data structure of choice when you need to preserve the insertion order, and also when you want to insert in arbitrary positions. The fact that you can iterate over the list is useful to a point, if you have to do that in a transaction the list cannot grow indefinitely.

You can use this data structure when a contract needs to frequently use several items in an ordered list that you can assume will be of limited size.

For example if you need a contract to always know the 100 largest holders of a token to give them some perks.

Unlike in previous articles, this time I’m not going to paste the whole code here. Instead, I will direct you to the full implementation. There is an implementation for a Singly Linked List and another one for a Doubly Linked List, with each one being about 200 lines of code, which I’ve crafted carefully for maximum clarity.

In this case, I think it is more important to discuss the trade-offs between a Singly Linked List and Doubly Linked List, in particular given that the Ethereum blockchain is quite limited in the algorithms that you can execute safely.

Usage

When I started the implementation of linked lists, I thought that doing a Doubly Linked List would be more complex than a Singly Linked List. Interestingly enough, it is slightly easier to implement the former. Adding in each item a link for the previous one allows you to remove this inefficient method:

That while

statement is evil.

You would use this method anytime that you know of an item and you want to insert another before it. Quite a common use case.

A gas comparison between LinkedList.sol and DoubleLinkedList.sol sheds some more light on the issue. For these tests, I used a list with 100 items.

addHead

and insertAfter

operations with LinkedList are O(1) and cost about 100K gas. Data retrieval is not depicted but given that we use mapping the cost will be O(1).

The issue is when we need to loop over the list. Every item we loop through seems to cost about 1K gas as seen in findTailIdWithGas

(which is a mock function that encloses findTailId

in a transaction, wasting some gas).

Maybe we can make without adding items at the tail or inserting before a known item, but the remove

function is more of an issue. In a LinkedList you have to loop through the list from the head to remove items. In smart contracts a method with a cost of O(N) needs to be approached very carefully, or better even, avoided.

A Doubly Linked List is easier to implement and more practical, even if a bit more expensive to use.

In this specific case, and with a gas block limit of about ten million, it means that you can’t remove items that are more than 10,000 positions away from the head. That can be quite dangerous.

For DoubleLinkedList, on the other hand, all the methods are O(1). addHead

and insertAfter

are more costly than in LinkedList because we need to update an extra pointer. If you need to insert at the end of the list, find neighbouring items in both directions, or remove items, you’ll benefit from O(1) cost. I haven’t included gas costs for looping the list but they should be identical to LinkedList.

And as I said before, it’s interesting that for the same functionality, DoubleLinkedList costs less to deploy than LinkedList. Not that important but interesting.

As with anything, your mileage will vary. Maybe you can do with a Singly Linked List, maybe you need a Doubly Linked List. Maybe you should use an array. At least now you know them all.

Other Implementations

The implementations discussed above are not the only ones, I just thought they would be easy to understand. There are other interesting implementations that might fit your use case better:

- You want to save some gas, then remove the id

field from the Item

struct. You don't actually need it, funny enough.

- You don't like the structs, them replace each struct variable with a mapping as a state variable, it will work exactly the same.
- You are happy with just appending items at the tail, and maybe those items expire after a set period of time: You can use RenounceableQueue.sol. It would work great for a traditional fast limit order book.
- If all the items in your list are unique, then you can use OrderedSet.sol. I like the compactness and elegance of it.
- If all the items in your list are unique, but you don't care about the order, then you are after a canonical Set, get it from OpenZeppelin.

Conclusion

Linked Lists are the first complex data structure that is considered in smart contracts. Given the constraints in smart contracts that force us to code as simply as possible it is necessary to know the trade offs between different linked list implementations.

In this article I've shown both Singly Linked Lists and Doubly Linked Lists, pointing to code ready to be reused. An analysis of gas costs has also been provided, along with guidance for safe use.

I feel quite privileged to have the opportunity to code basic data structures. Sometimes coding smart contracts feels like a trip many years back when programming meant being very close to the hardware and was very close to mathematics. I like that.

If you are considering using this code in a project, want to contribute, or have ideas to explore, please drop me a line! Talking to those that read me is always a pleasure :)