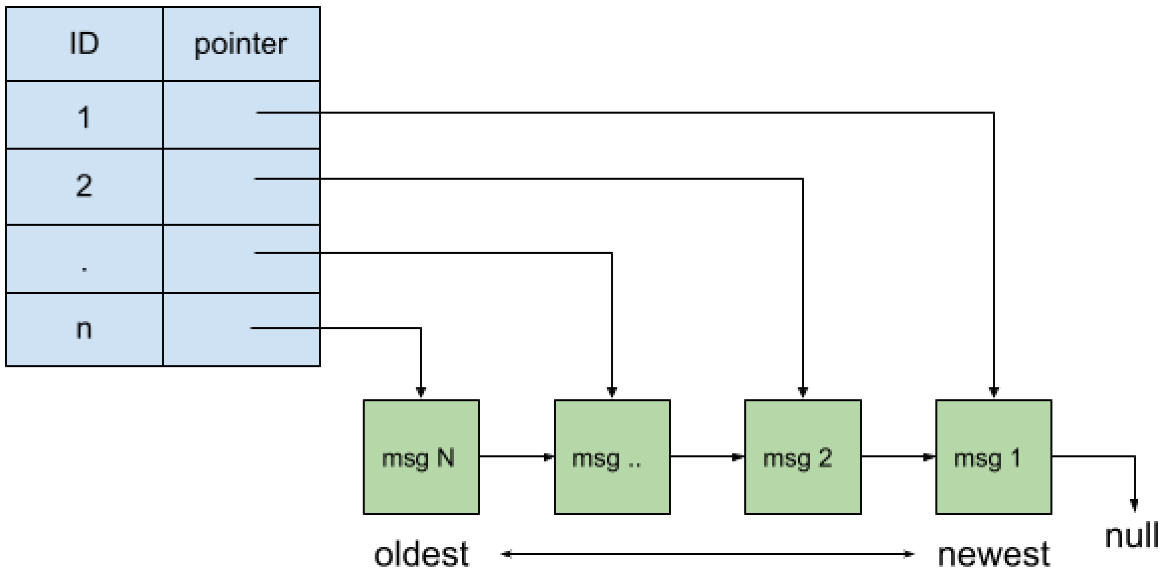
Gossamer take home assignment

Solution

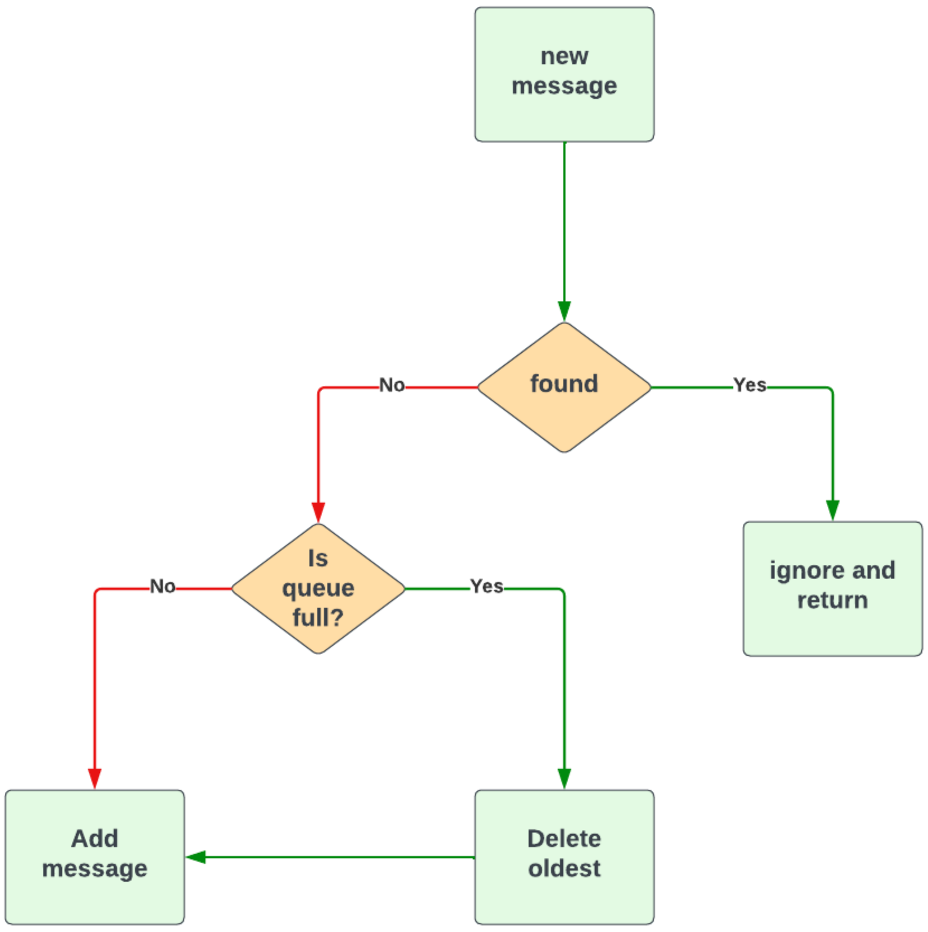
The solution consists mainly in the use of 2 data structures, since we need to store messages in a FIFO way, the most appropriate thing would be to use a queue, the issue is how to solve the searches and deletions in the best possible way.

For this I decided to use a combination of 2 structures, on one hand a dequeue using a double linked list (part of golang's container/list library) and a map to optimize searches.



Additions

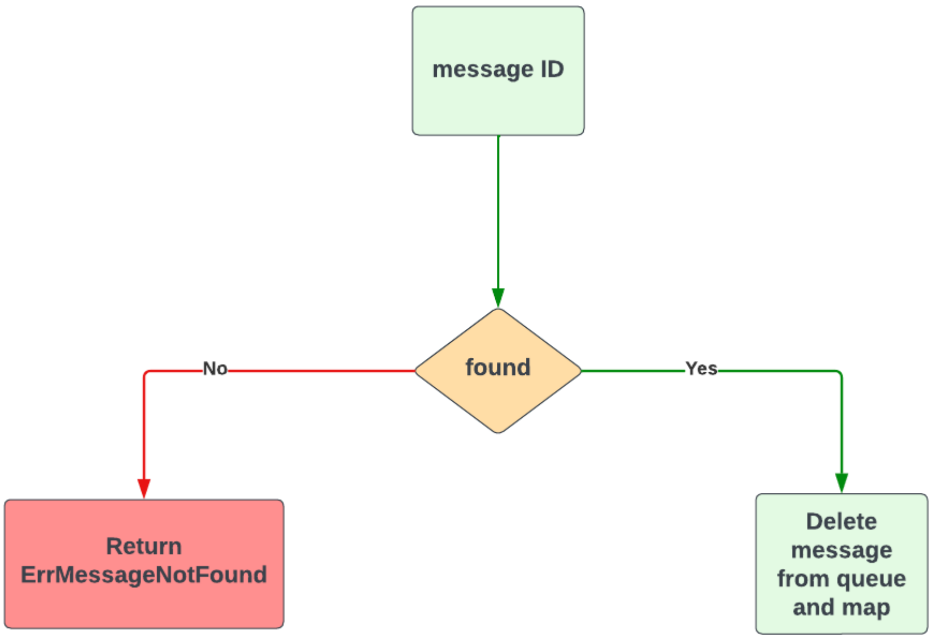
Since we must ignore duplicates, we have to be able to find them in the most optimal way possible, for that the add method consists of the following:



Since the time complexity for searches in a map is O(1) (best case) and removing an element from a linked list consists of modifying pointers, which is also O(1) this method could be considered to solve the additions in O(1) for the best case.

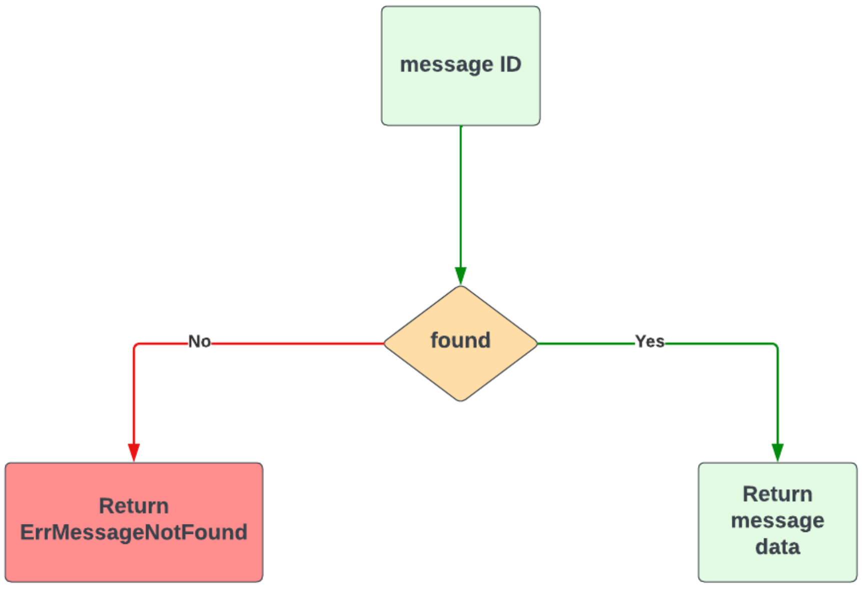
Deletions

This method is very simple, to delete messages we must first look for it in our data structure, if we don't have it we will return an error, if we find it we will delete it both from the queue and from the map.
This method is O(1)



Get message

In order to get the message we look for it in our data structure, if we don't find it we will return an error, if we find it we return the message data.
This method is also O(1)

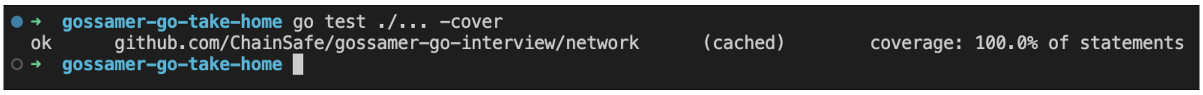


Get all messages FIFO

This method is simple but is the more expensive one. We have to traverse the queue and build an array with the elements in FIFO order.
This method is O(n) since we have to traverse the full queue.

Testing

In terms of testing the project comes with tests and this implementation pass all of them with a coverage of 100%

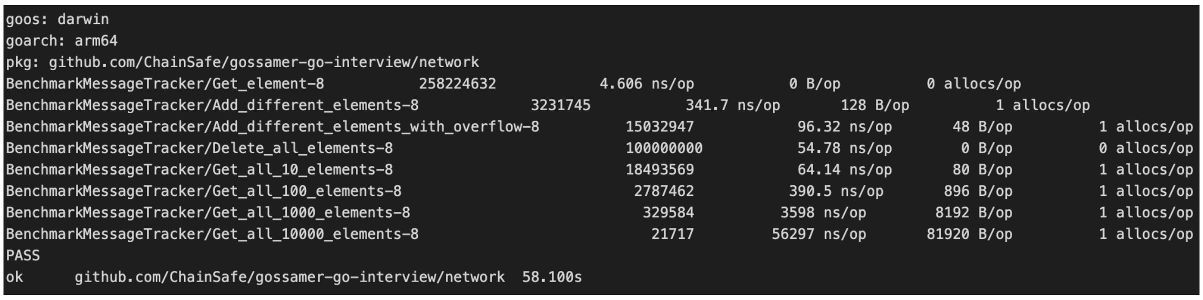


```
● ➜  gossamer-go-take-home go test ./... -cover
  ok      github.com/ChainSafe/gossamer-go-interview/network      (cached)        coverage: 100.0% of statements
○ ➜  gossamer-go-take-home
```

Benchmarking

I added some benchmarks in golang to bench the solution in terms of throughput (ns/op), and memory consumption (bytes/op and allocations/op)
***Note:** these benchmarks could take some time to execute since it requires an initial setup*

These are the results:



```
goos: darwin
goarch: arm64
pkg: github.com/ChainSafe/gossamer-go-interview/network
BenchmarkMessageTracker/Get_element-8           258224632            4.606 ns/op           0 B/op          0 allocs/op
BenchmarkMessageTracker/Add_different_elements-8             3231745           341.7 ns/op       128 B/op          1 allocs/op
BenchmarkMessageTracker/Add_different_elements_with_overflow-8          15032947            96.32 ns/op       48 B/op          1 allocs/op
BenchmarkMessageTracker/Delete_all_elements-8                          100000000            54.78 ns/op        0 B/op          0 allocs/op
BenchmarkMessageTracker/Get_all_10_elements-8                           18493569            64.14 ns/op       80 B/op          1 allocs/op
BenchmarkMessageTracker/Get_all_100_elements-8                           2787462           390.5 ns/op       896 B/op          1 allocs/op
BenchmarkMessageTracker/Get_all_1000_elements-8                           329584          3598 ns/op        8192 B/op          1 allocs/op
BenchmarkMessageTracker/Get_all_10000_elements-8                           21717         56297 ns/op       81920 B/op          1 allocs/op
PASS
ok      github.com/ChainSafe/gossamer-go-interview/network  58.100s
```

We can see that all methods has very good stats, the only one which worst results is the get all elements since it is O(n) in terms of time and memory

Another possible solution

Using an array for the queue

We could use an array for the queue but deletions would cause empty spaces at the beginning of the array and we couldn't use them unless we shift all the elements or the queue was completely emptied. This does not match our requirements and that is why I discarded this option

Using a circular queue

An improvement to using an array for the queue is to use a circular queue, this avoids the empty spaces when executing the dequeue operation (pop) but would still cause empty spaces that would force us to shift all the elements, which is very expensive.

Considerations

- The add and removes are not atomic, a corner case could happen where a message is added or deleted deleted from the queue and not from the map (for example if the application panic while executing this methods)

- The methods are not thread safe, it was not required in the problem description, but it would be easily solved by adding a mutex before each critical section.
  The only tradeoff is that the benchmarks will be affected and probably the throughput will decrease