COLUMBIA UNIVERSITY
IN THE CITY OF NEW YORK

RESEARCH PROJECT

COMS 4901

PROF. STEPHEN EDWARDS AND JOHN HUI

Implementation of two solutions to the Order Maintenance Problem

Alexis GADONNEIX (ag4625)

December 2023

1 Introduction

The order maintenance problem is a well-known problem in computer science. It is defined as follows: given a set of elements, we want to maintain a data structure that supports the following operations:

- `insert(x)`: insert a new element right after `x`
- `delete(x)`: delete `x`
- `compare(x, y)`: which element of `x` and `y` has higher priority?

A first very crude solution to this problem is to use a vector. But all operations are linear in the size of the vector. We can do better. A better idea would be to use a linked list. But then, the `compare` operation is linear in the size of the list...

This data structure has several applications in computer science such as process scheduling and graph algorithms. We will focus on two data structures to solve this problem: The first was proposed by Dietz & Sleator in 1987 [DS87] and the second by Bender et al. in 2002 [Ben+02]. The two solutions have very similar theoretical bounds, but the second one is more practical.

The main goals of this project are:

- Implement both solutions in Rust
- Test and debug
- Benchmark, compare, and optimize

2 Algorithms

2.1 Naive

Before diving into the two solutions, let's look at a naive solution to the problem. We can store our priorities in a binary tree that grows deeper and deeper as we insert elements (see Figure 1 for an example). To avoid having to maintain an actual tree, we can simply use the labels of the nodes:

- The first priority is labeled 0
- When we do an `insert(x)`, we relabel `x` to be $2 \times$ `label(x)` and the new element has label $2 \times$ `label(x)` $+ 1$
- The comparison between two elements is done by finding the ancestor of the deepest element that is at the same depth as the other one, and comparing these.

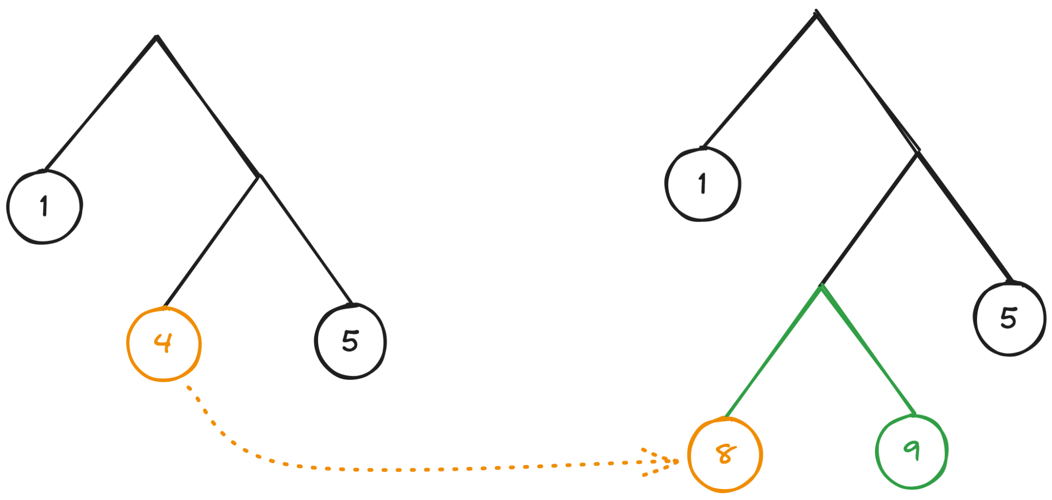

Figure 1: Example of an insertion with the "naive" data structure. We are inserting an element after the second priority (the one with label 4)

This solution is easy to implement and operations are fast, but it has a major drawback: the labels grow exponentially with the number of insertions and will quickly overflow. We implemented an alternative version that uses `BigInt`s. It prevents from the overflow issue but makes the operations much slower.

2.2 Dietz & Sleator

In the solution proposed by Dietz & Sleator in 1987 [DS87], we think of the set of possible labels (e.g. 0 to 2^{64}) as a circular list that we will fill in as we insert elements. The priorities are connected together by a circular doubly-linked list. The general idea for inserting a new priority after a priority `x` is the following (see Algorithm 1 for more details):

- We iterate through the successors of `x` and compute a weight for each of them based on the distance between the elements
- We stop when we reach an element whose weight is smaller than a threshold (intuitively, this means that the element is "far enough")
- We then relabel those elements evenly
- Finally, we choose a label in between the labels of `x` its successor (in the middle) and we create a new priority with this label

Algorithm 1 List-range relabeling, Dietz & Sleator

```
Require: x: Priority
Require: base: Priority
                                                    // Relabel priorities
  j ← 1
  $v_0$ ← label(x)
  curr ← next(x)
  w ← 0
  while w ≤ $j^2$ do
      w ← (label(curr) − $v_0$) % M
      j ← j+1
  end while
  to_relabel ← next(x)
  for k ∈ 1..j do
      label(to_relabel) ← (label(x) + (k × w)/j) % M
      to_relabel ← next(to_relabel)
  end for
                                                    // Insert the new element
  y ← new Priority
  label(y) ← ((label(x) − label(base)) % M + (label(next(x)) − label(base)) % M)/2
  prec(next(x)) ← y
  next(x) ← y
```

We call this algorithm **list-range relabeling**. See Figure 2 for an example.

Note that, to avoid overflowing, we will simply loop back to 0 when we reach the maximum label (2^{64}). But this means that in order to compare two elements, we have to keep track of a "base" label starting at 0 and shifting if necessary. Then, the comparison operation is:

$$(\texttt{label(x)} - \texttt{label(base)}) \mod 2^{64} < (\texttt{label(y)} - \texttt{label(base)}) \mod 2^{64}$$

Deleting an element of the data structure is as simple as removing it from the linked list.

The paper proves the following theoretical complexity bounds (where n is the number of elements in the data structure):

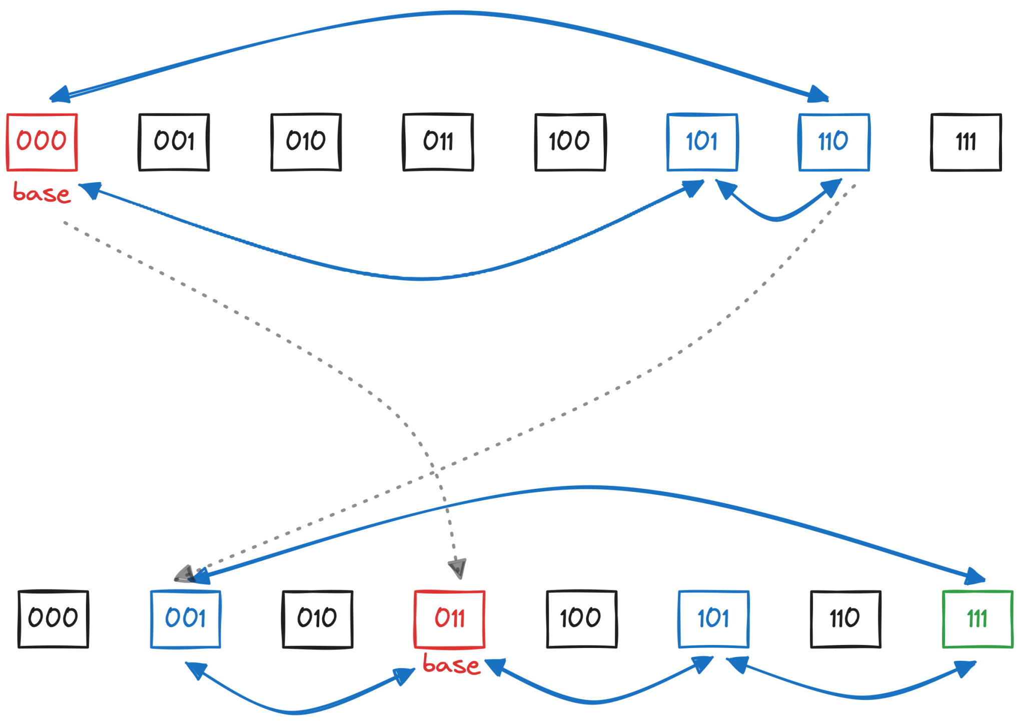

Figure 2: Example of an insertion with the list-range relabeling algorithm. We are inserting an element after the first element (with label 101). We can see that the base also has to move.

- `insert(x)`: $O(\log n)$ (amortized)
- `delete(x)`: $O(1)$
- `compare(x, y)`: $O(1)$

It is possible to get an $O(1)$ amortized complexity for `insert(x)` by adding a level of indirection using a technique by Tsakalidis [Tsa82]. We chose not to implement this extra-layer for the sake of simplicity and because we thought that the complexity gains would be too small in comparison to the cost of the extra look-up.

This data structure theoretically only supports $\sqrt{M}$ insertions where M is the number of possible labels (2^{32} or 2^{64}). It is a significant limitation on a 32-bits machine as you can insert at most 65K priorities.

2.3 Bender et al.

In 2002, Bender et al. proposed a new solution to the problem in [Ben+02]. The main goal of this approach is not to provide better bounds, but to get an algorithm and a proof that are more intuitive. In practice, we will see that our implementation is in fact faster than the first algorithm for some operations.

This time, we look at the set of possible labels (0 to 2^{64}) as the leaves of a binary tree whose root is the empty string, and each node n has two children $n : 0 = 2 \times n$ and $n : 1 = 2 \times n + 1$ (where : is the concatenation to the binary representation). We don't have to maintain an explicit tree because we can easily compute the parent or the children of a node with bit operations. And once again, the priorities (which are leaves of the implicit tree) are connected together by a doubly-linked list (this one doesn't need to be circular).

The algorithm for `insert(x)` is pretty straightforward (See Algorithm 2), and the key steps are:

- if there is space (`label(next(x))` $>$ `label(x)` $+ 1$), then just insert the new element somewhere in between.
- if there is no space available, we have to relabel (we call this algorithm **tag-range relabeling**).
  - To do so, we simply have to go up the tree and find the first sub-tree containing `x` that is not "too dense".
  - Once we find this sub-tree, we evenly spread all the elements it contains over the range of labels of the sub-tree. Note that once again, all these operations can be done with simple bit manipulations over the labels.

See Figure 3 for an example.

The paper proves that this algorithm has the same complexity bounds as the previous one:

- `insert(x)`: $O(\log n)$
- `delete(x)`: $O(1)$
- `compare(x, y)`: $O(1)$

The same indirection trick can be applied to reduce the insertion's complexity, but we chose not to implement it for the same reasons as before.

Deleting an element of the data structure is as simple as removing it from the linked list. And a small but important difference is that, in order to compare two elements, we simply have to compare their labels (and we don't have to compute their position relative to a base as in the previous solution).

The previous data structure was limited to $\sqrt{M}$ insertions. Here, the maximum number of elements you can insert depends on the parameter T. If T is close to 2, the insertion will be faster but you can only insert a small amount of priorities before the root overflows. If T is close to 1, the algorithm is slower but you can insert more priorities (close to M). The issue is that it's difficult to find a good trade-off as the user doesn't usually know up-front how many priorities he will need. But the good news is that T doesn't have to be fixed and you can compute the best T at each insertion. This computation is not free though, and we will see later (section 3.3) how we made this process faster.

Algorithm 2 Tag-range relabeling, Bender et al.

```
Require: x:  Priority
Require: T: float (in ]1, 2[)
  if label(x) $+ 1 ==$ label(next(x)) then                // Relabeling (if needed)
                                    // Find the smallest subtree that is sparse enough
    begin $\leftarrow$ x
    end $\leftarrow$ x
    count $\leftarrow 1$
    layer $\leftarrow 0$
    threshold $\leftarrow 0$
    internal_node $\leftarrow$ label(x)
    min_label_in_subtree $\leftarrow$ label(x)
    max_label_in_subtree $\leftarrow$ label(x)
    while layer $< 64$ && threshold $<$ count$/2^{\text{layer}}$ do
      while label(prec(begin)) $\geq$ min_label_in_subtree do
        begin $\leftarrow$ prec(begin)
        count $\leftarrow$ count $+ 1$
      end while
      while label(next(end)) $\leq$ max_label_in_subtree do
        end $\leftarrow$ next(end)
        count $\leftarrow$ count $+ 1$
      end while
      layer $\leftarrow$ layer $+ 1$
      threshold $\leftarrow$ threshold/T
      internal_node $\leftarrow$ internal_node$/2$                    // Get the parent
      min_label_in_subtree $\leftarrow$ internal_node $* 2^{\text{layer}}$      // Get the left-most
  leaf
      max_label_in_subtree $\leftarrow$ internal_node $* 2^{\text{layer+1}} - 1$        // Get the
  right-most leaf
    end while
                                                         // Relabel priorities
    gap $\leftarrow 2^{\text{layer}}/$count
    curr $\leftarrow$ begin
    curr_label $\leftarrow$ min_label_in_subtree
    while curr $==$ next(end) do
      label(curr) $\leftarrow$ curr_label
      curr $\leftarrow$ next(curr)
      curr_label $\leftarrow$ curr_label $+$ gap
    end while
  end if
                                                      // Insert the new element
  y $\leftarrow$ new Priority
  label(y) $\leftarrow$ (label(x) $+$ label(next(x)))$/2$
  prec(next(x)) $\leftarrow$ y
  next(x) $\leftarrow$ y
```

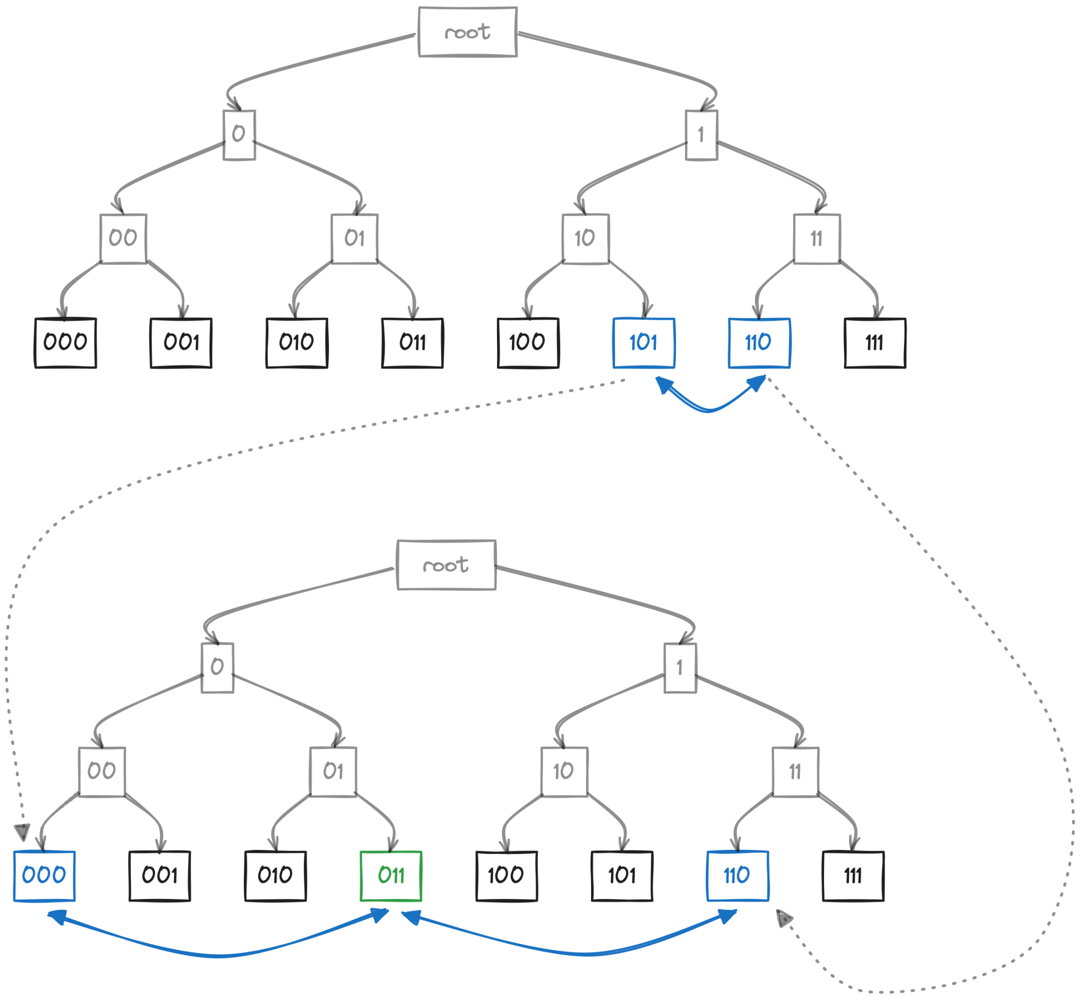

Figure 3: Example of an insertion with the tag-range relabeling algorithm. We are inserting an element after the first element (with label 101). The first sub-tree that was sparse enough is the whole tree (assuming we chose `T` accordingly).

3 Implementation

In this section, I will present the main data structures and design choices of our implementations, as well as some tricks we used to improve the speed.

3.1 Interface

The project was built as a Rust library (a `cargo crate`). In order to have a common interface for multiple implementations, as well as shared tests and benchmarks, we leveraged Rust's `Trait` system. The interface for a priority is minimal:

- `new: () -> Priority`: create a new priority, unrelated to any other priority

- `insert: Priority -> Priority`: insert a new priority right after the argument

The comparison is made possible through the implementation of the `PartialCmp` trait which lets us use the common comparison operators (`>=`, `<`, etc.).

The deletion is implemented through the `Drop` trait, which acts as a destructor. A priority is deleted from the data structure when it goes out of scope.

3.2 Common data structures

I won't go too much into the technical details, but we used a couple of data structures to implement both solutions ([Ben+02] and [DS87]).

- `PriorityInner` is a node of a circular doubly-linked list. It can be seen as the private interface of a `Priority`. It stores the following fields: `prev`, `next`, `label`, `ref_count`. The first two are (smart) pointers to the neighbors, the third is the label as a `usize = u64` (on a 64-bit machine), and the last one is used for the manual reference counting. Those fields are actually wrapped in smart pointers called `RefCell`, mutable memory locations. It is necessary to abide by Rust's borrow checker when trying to mutate a field of an object that is not owned or borrowed as mutable (this is called the interior mutability pattern).
- `Arena` is the internal store of priorities containing all the `PriorityInner`. In our case, it's a `Slab`. It is indexed by `PriorityKey`, an alias for `usize`. The implementation of the `Arena` contains all the methods in charge of creating, deleting, and connecting the `PriorityInner`, the nodes of the linked list.

The end-user has access to a `Priority`, which contains (privately) its `PriorityKey` and a reference to the `Arena`.

3.3 Pre-computing the density thresholds

When looking at Algorithm 2, you might notice some potentially expensive operations. First, we have many multiplications and divisions by powers of 2 used to traverse the tree. Those are not an issue since we can simply use the efficient right- and left-shifts operations (», «).

The more annoying ones are the floating point operations needed to compute the density threshold. `T` is a parameter between 1 and 2 and the maximum density of a subtree whose root has a height h (0 being the height of a leaf) is $\mathtt{T}^{-h} = 1/\mathtt{T}^h$. It gets even worse if we compute the best value for T at each insertion to ensure the algorithm terminates without compromising its speed.

The solution we chose was to pre-compute some values at compile time that the algorithm will simply have to look-up. The values that we need are

$$[\mathtt{T}^{-i}\ \mathtt{for\ i\ in\ 0..64\ for\ T\ in\ list\_of\_Ts}]$$

where `list_of_Ts` is a list of numbers between 1 and 2. In our case we chose 20 values evenly distributed between 1.1 and 1.9. However, there are still two issues remaining: We still have to compute the densities of the subtrees (float) at run-time, and we don't have an easy way of finding which value of `T` we should use.

To solve these, instead of pre-computing the density thresholds, we chose to calculate the (integer) capacities of the subtree for different heights h and values of `T`:

$$\left\lfloor \frac{2^h}{T^h} \right\rfloor$$

This value can simply be compared to the number of priorities contained in the subtree. Moreover, it's now trivial to find the best `T` by finding the capacity that is closest to (and larger than) the total number of priorities in the `Arena`.

4 Testing, Benchmarking, and results

4.1 Unit tests, Integration tests, and Quickcheck

We implemented a pretty large test suite in order to ensure the correctness of the implementations. It was a good idea given the number of bugs we were able to find thanks to these tests.

- Unit tests: We implemented to test the various internal data structures, and in particular make sure we were not leaking any memory in the process.
- Integration tests: We implemented multiple scenarii using the public methods of a `Priority`. The interface is really simple (insertion, deletion, comparison), making the testing process straightforward.
- Quickcheck: The simplicity of the interface encouraged us to try a random and automated property-based testing tool called Quickcheck (it's the Rust port of a Haskell library). The general idea is that we simply have to implement the `Arbitrary` trait for our data structure and define some properties that must be true for any instance of the data structure.

  Our first approach was to generate an arbitrary `Vec<Priority>` by inserting and deleting priorities and maintaining the order of the vector. The only property we have to check is: "Is the order of the vector the same as the order of our data structure?". One nice feature of Quickcheck is the shrinking, meaning that it will try to find a minimal example by shrinking the failing input. But we can't "reverse" an insertion or a deletion, the internal state will be different.

  The solution is to generate arbitrary vectors of decisions that can be either `Insert(i)` or `Delete(i)` for an index `i`. We turn this vector of decisions into a vector of priorities only when the property is tested. Shrinking the vector of decision can be done by removing the decisions from the end until we find the one that caused the failure.

4.2 Bugs found and fixed

We found multiple bugs thanks to our tests. Most of them were found by Quickcheck, giving confirmation that human programmers are pretty bad at testing edge cases. In particular we fixed two bugs related to integer overflow. In both implementations, we used word-sized labels to optimize memory and speed. When manipulating the labels, we thus had to be careful not make mistakes (in particular in [DS87] which uses modular arithmetic).

The first "interesting" bug was found in the implementation of [Ben+02], when trying to compute the minimum and maximum label of a subtree. Given an internal node with label x at height i, the minimum is $x \cdot 2^i = x >> i$ and the maximum is $x \cdot 2^{i+1} - 1 = (x+1) >> i$. This calculation overflows if the subtree is the right-most one. Our solution was to notice that the maximum was simply the label of internal node with i "1" at the end. This can be written $!((!x) >> i)$.

The second bug happened in the implementation of [DS87]. When relabeling the nodes, the paper simply suggests to compute the value

$$\left\lfloor \frac{k \cdot w}{j} \right\rfloor \% M$$

The problem is that $k \cdot w$ might overflow. Initially, we were computing

$$\left\lfloor \frac{(k \cdot w)\% M}{j} \right\rfloor$$

At first, we didn't realize that they were different. It turns out that division in modular arithmetic is a bit trickier than we thought

$$(a/b)\%M = (a\%M \cdot b^{-1}\%M)\%M$$

where b^{-1} is the modular inverse of b (such that $(b \cdot b^{-1})\%M = 1$). b^{-1} can be computed with the extended Euclidean algorithm in $O(\log^2(M))$. In practice and for the sake of simplicity, we instead chose to locally cast the numbers to 2-words integers and perform the overflowing calculation, before casting it back to `usize`. Our benchmarks indicated it didn't have any measurable impact on speed.

4.3 Benchmarks, Profiling, and Optimization

Our benchmarks were mainly aimed at comparing the implementations on two tasks: insertion and comparison. We used the crate `Criterion` to write and execute the benchmarks (including precise measurements).

Benchmarking the insertion gave us multiple insights:

#insertions	list-range	tag-range	naive	naive-bigint
10	60	63	75	150
1_000	31	37		122
100_000	47	46		140
1_000_000	156	155		
5_000_000	188	185		

Table 1: Time per insertion (nanoseconds) depending on the total number of insertions

When inserting at random, our implementations of [DS87] and [Ben+02] have the same performance (see Table 1). This is in line with the results presented in [Ben+02]. However, the list-range relabeling is a bit faster for some common edge cases such as "always insert after the last element".

It is also important to mention that the list-range relabeling has a strict limit in terms of how many priorities you can insert whereas the tag-range relabeling (with the variable threshold discussed above) can use almost every possible label for a given word size.

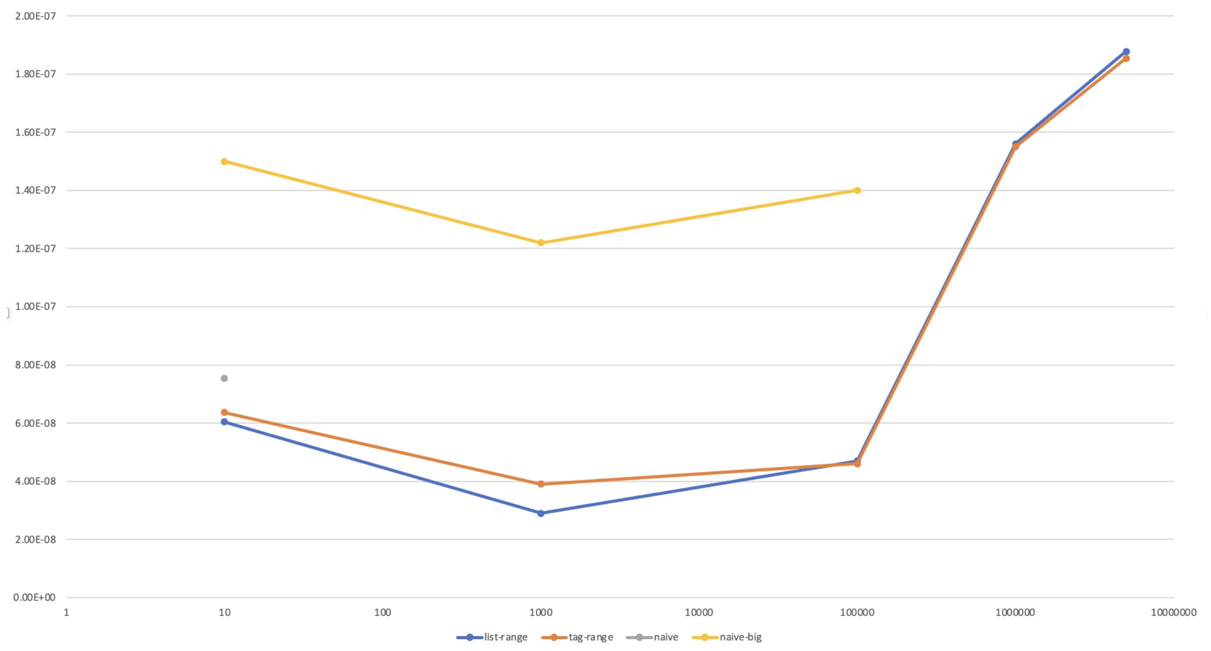

Figure 4: Time per insertion (nanoseconds) depending on the total number of insertions

We were also able to measure the average insertion time for different total number of insertions (see Figure 4). [Ben+02] showed that the average number of relabeling was increasing with the total number of elements inserted, and this is in line with the speed we measured.

The naive algorithm using word-sized labels has approximately the same speed as our other implementations but it is limited to a very small total number of insertions. We also tried to extend this naive approach using `BigInt`s. As visible on the graph, this is a lot slower.

We also created two benchmarks to measure the comparison speed. One that creates priorities (with insertions and deletions to "shuffle" the slab and avoid too much gains due to caching) and then compares two random priorities, and another one that also creates a random vector of 200 priorities but this time sorts it (see Table 2).

benchmark	list-range	tag-range	naive	naive-bigint
compare2	8.3	5.2	11	200
sort	3221	2645		43672

Table 2: Time per comparison or sort (nanoseconds)

We notice that the naive implementation is slower than the others, which is great since that means we are not making adding unnecessary if the user only needs a small amount of priorities.

Another important take-away is that the tag-range algorithm ([Ben+02]) is faster than the other in terms of comparison speed. This makes sense: the comparison in the tag-range version is 2 look-ups and 1 comparison whereas in the list-range version, you need 3 look-ups (you need the label of the base), 2 wrapping substraction, and 1 comparison.

In order to find bottlenecks in our code, we performed some profiling using the `cargo-flamegraph` crate. This library is really easy to use, can be plugged to benchmarks, and produces useful graphs at a "function granularity" (e.g. Figure 5). We sometimes had to artificially refactor our code into very small functions to get a more detailed output.

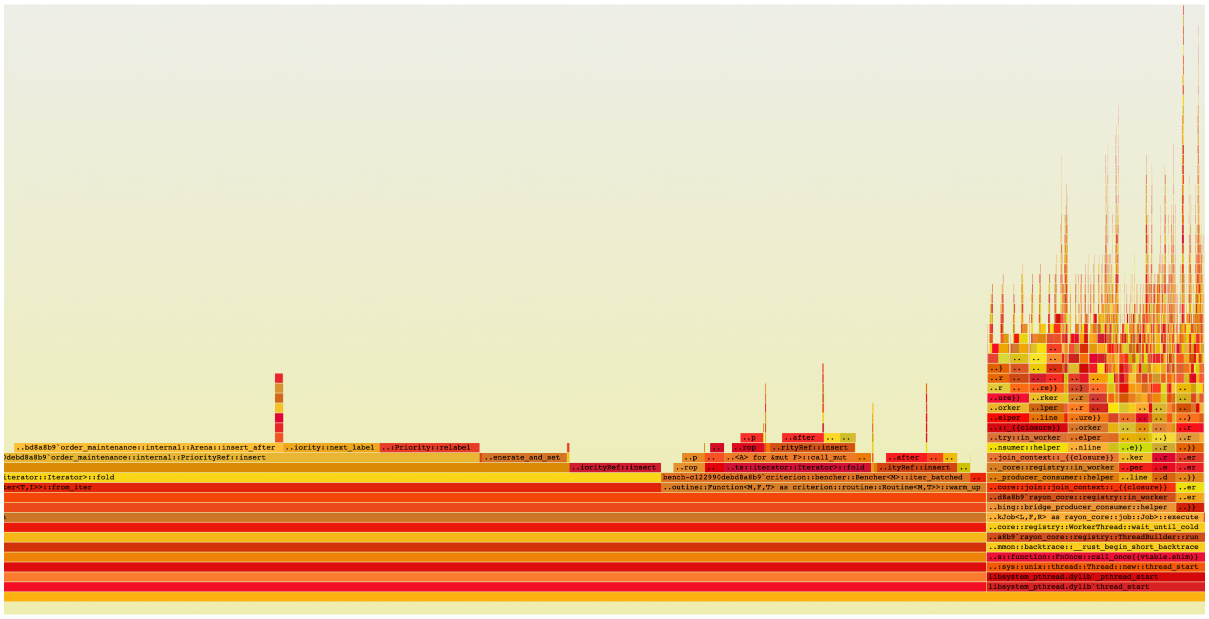

Figure 5: Example of a flamegraph produced by the crate

Thanks to these flamegraphs, we were able to optimize some parts of the implementations. For example:

- Using a `Slab` to store the priorities was approximately 20% faster (on our benchmarks) than using a `SlotMap` that we initially used.
- To compute the mid-point between two labels, it's 15% faster (on my laptop) to use integers operations (`x + (y-x)/2`) rather than using bit manipulation (`(x^y) » 1 + x&y`). This one was unexpected, we thought the two would be equivalent after the compiler optimizations.

5 Conclusion and Future Work

This project led to the development of an (almost) production-ready Rust library. We were able to verify or refine the theoretical claims of the papers on real-world use-cases.

The current conclusion is that we should use the implementation of the algorithm from [Ben+02], as it is simple to understand and maintain, and provides better or similar results on every benchmark.

There are still some areas to explore in the future. Among them:

- Testing and benchmarking on a microcontroller, since it was one of the initial motivations of this project. Our hope is that we will see similar trends on different hardware.
- Trying out different allocation strategies. We already experimented with multiple stores, but this analysis could be refined. It might also be worth trying using `UnsafeCell` instead of `RefCell` now that we are more confident about the correctness of the code.
- Adding a level of indirection. Both papers reduce the theoretical complexity by adding a level of indirection. Our guess is that it wouldn't improve the speed because of the cost of the additional look-ups, but it might be worth trying.

I also wanted to address special thanks to John who has been extremely patient and helpful this semester!

References

[Tsa82] Athanasios K. Tsakalidis. “Maintaining order in a generalized linked list”. In: *Theoretical Computer Science*. Ed. by Armin B. Cremers and Hans-Peter Kriegel. Berlin, Heidelberg: Springer Berlin Heidelberg, 1982, pp. 343–352. ISBN: 978-3-540-39421-1.

[DS87] P. Dietz and D. Sleator. “Two Algorithms for Maintaining Order in a List”. In: *Proceedings of the Nineteenth Annual ACM Symposium on Theory of Computing*. STOC ’87. New York, New York, USA: Association for Computing Machinery, 1987, pp. 365–372. ISBN: 0897912217. DOI: 10.1145/28395.28434. URL: https://doi.org/10.1145/28395.28434.

[Ben+02] Michael A. Bender et al. “Two Simplified Algorithms for Maintaining Order in a List”. In: *Proceedings of the 10th Annual European Symposium on Algorithms*. ESA ’02. Berlin, Heidelberg: Springer-Verlag, 2002, pp. 152–164. ISBN: 3540441808.