Provably Correct Priority Queues in Lean

Peter Khoudary

May 6, 2025

Abstract

Lean is simultaneously an automated reasoning tool and a functional programming language. This allows for proofs of correctness that naturally arise from Lean implementations of functional algorithms and data structures. I define the interface of the priority queue data structure and argue for the advantages of a leftist heap implementation. This results in an efficient Lean 4 implementation of leftist heap priority queues, complete with proofs of correctness for all interface operations [1].

[1] The repository containing the implementation and proofs of correctness can publicly accessed here.

Contents

1 Introduction

1.1 Lean

Lean is both a functional programming language and an automated reasoning tool, built upon dependent type theory [1]. This expressivity gives it multi-faceted appeal, contributing to the development of an open-source mathematical library for Lean [2]. This work focuses more specifically on extending its library for functional programming and computer science, providing an implementation of priority queues which leverages Lean's twofold capacity to give proofs of correctness for the library.

1.2 Priority Queues

Priority queues (PQs) are a data structure which are ubiquitous in algorithm design. They are at the core of Dijkstra's algorithm for shortest paths, Prim's algorithm for minimum spanning trees, and the heapsort sorting algorithm. To understand them, we first briefly review the basic queue data structure. A queue is an ordered collection of keys that support two operations, inserting to the back of the queue, or popping (removing) the first element from the queue. This is known as a first-in first-out (FIFO) data structure, since one item inserted into the queue will always leave the queue before one inserted after it. They function exactly the queues in real life, such as getting in line to buy food, where the only thing you do is get in line at the back or get your food and leave. Priority queues are an extension of the queue data structure, which each key in the queue has an associated *priority*. For the the rest of the discussion we will assume priorities are integers, but priorities can be of any type, so long as they can be totally ordered. The insert operation just inserts a key-priority pair, with no claims about where it resides in the ordering. However, the deletion operation for PQs removes the element from the PQ with the lowest (or highest, which is an implementation detail) priority from the PQ, *regardless of when it was inserted*. The real life analogy for a PQ is a hospital waiting room. Everyone is free to come in at any time, but even if the patient with a sprained ankle has been waiting for hours, someone who comes in after with life-threatening injuries is given treatment first. So every time you pop an element off a priority queue, you are guaranteed that the element it returns has the element with the lowest (or highest) priority currently in the PQ. This invariant is what allows the laid out algorithms above to function.

2 Leftist Heaps

There are several implementations of the priority queue data structure (with the most common being binary heaps), but we focus our attention on the leftist heap [3], which will argue has key advantages over the other implementations. A leftist heap is a binary tree storing key, priority pairs such that for the tree rooted at any node, it is both a min-heap and maintains a leftist shape property. A min-heap is a rooted tree such that for every node in the tree, the priority of the key stored at that node is less than or equal to the priority of all its descendants. The root of this tree must therefore always be the lowest priority element in the tree. To define this leftist shape property, we must define the concept of a "rank" for binary trees, which is just the length of its right spine. The right spine of a tree is the length of the path which begins at the root and traverse right down the tree until it hits a leaf. The formal definition of rank is as follows.

$$rank(leaf) = 0$$
$$rank(node(L, x, R)) = 1 + rank(R)$$

With this definition in hand, we are able to formally define the criteria for a leftist heap. A binary tree is a leftist heap if both predicates hold.

1. **Min-heap property**: For every node in the tree, the priority stored at the node must be less than or equal to the priorities of all its descendants. In a max-heap, the predicate is the same except priorities must be greater than or equal to.
2. **Leftist property**: For every node in the tree, the rank of its left subtree is greater than or equal to the rank of its right subtree.

Intuitively, the leftist property is a shape invariant about our tree, stating that most of the entries will exist on the left side of any given subtree. This results in a potentially extremely unbalanced tree, but that's actually the goal. This imbalance is what will allow us to implement the operations efficiently. So a leftist heap is just a min-heap which also observes this shape property.

2.1 The Leftist Rank Lemma

From the leftist property holding in our tree, we are able to make a strong claim about the shape of our tree. The **Leftist Rank Lemma** states that in a leftist heap with n entries, the rank of the root node is at most $\log_2(n+1)$. Looking at the two properties that define a leftist heap, we know that if our rooted tree is a valid leftist heap, then so too are all of its subtrees. So for any subtree of our original, we have that the rank of that tree is logarithmic with respect to the number of entries in the tree. The proof details will be covered in section 4.2, so just take this as axiom for now. This powerful fact allows us to efficiently introduce another operation over our PQs.

2.2 The Meld Operation

So far, we've only considered inserting and deleting elements from the PQ. Here we introduce the *meld* operation, which takes in two PQs and merges them together into a new, valid PQ. The key advantage of leftist heaps is that they support melding heaps a, b together in $O(\log|a| + \log|b|)$ time. This outperforms other implementations [3], and is optimal for (deterministic) priority queues. Furthermore, observe that both insert and deletion can be implemented using meld as a subroutine. Inserting an element reduces to melding your current PQ with a singleton node, and deletion just removes the root element and melds its left and right subtrees together to make your new heap. So given that meld is $O(\log|a| + \log|b|)$, we get $O(\log n)$ time bounds on both insert and delete for free, which is also optimal among PQs. Being able to define both operations in terms of meld makes our analysis easier, as we get correctness of the whole interface while only having to reason about meld. After looking at our implementation of meld, we will see that the time bounds arise naturally from the Leftist Rank Lemma.

3 Implementation of the Leftist Heap Data Structure

After introducing the meld operation, let's formally define the specific PQ interface we wish to implement, with desired time bounds.

$empty$: creates a new PQ with keys of type β in $O(1)$

$insert$: inserts a key-priority pair into a PQ q with keys of type β in $O(\log|q|)$

$deleteMin$: deletes the lowest priority (key, priority) node in PQ q and constructs resulting PQ q' in $O(\log|q|)$

$meld$: combines PQs a, b with key types β into one PQ in $O(\log|a| + \log|b|)$

To start, we provide the inductive definition of a leftist heap.

```
inductive leftistHeap (β: Type) where
| leaf
| node (left: leftistHeap β) (key: β) (priority: Int) (rank: Nat) (right: leftistHeap β)
```

A key implementation detail is that at every node, we also store its rank. Even though the definition of rank is recursive as laid out above, in order to get desired time bounds on meld we need $O(1)$ access to the rank of a given node. Now for the most important part, melding.

```
def rank {β : Type} : leftistHeap β → Nat
  | leaf => 0
  | node _ _ _ r _ => r

def mkLeftistNode {β : Type} (left : leftistHeap β) (key : β) (priority : Int) (right :
    leftistHeap β) : leftistHeap β :=
  if rank left < rank right
  then node right key priority (rank left + 1) left
  else node left key priority (rank right + 1) right

def meld {β : Type} : leftistHeap β → leftistHeap β → leftistHeap β
  | leaf, b => b
  | a, leaf => a
  | node la ka pa rka ra, node lb kb pb rkb rb =>
    if pa < pb
    then mkLeftistNode la ka pa (meld ra (node lb kb pb rkb rb))
    else mkLeftistNode lb kb pb (meld (node la ka pa rka ra) rb)
```

Firstly, we define the rank helper function to just merely extract the rank of a given node. Let's break down the meld operation. Trivially, if you meld a heap with a leaf, the heap just stays the same. In the recursive case, we case on which heap has the lower priority root node. In order to maintain the minheap property, we need to set that key-priority pair as the root of our new tree. We then keep the left subheap of the node with the lower priority, and recursively meld along the right subheap and the entire other heap. More simply, given heaps a, b where the root of a has lower priority, our new heap is going to have L_a as one child, and $meld(B, R_a)$ as the other.

Finally, given that those two are going to be our children, we send the children and key-priority pair to the $mkLeftistNode$ function to actually create our new node. We need this extra conditional to maintain the leftist invariant. By the leftist property we know the rank of L_a is at least as large as the rank of R_a. But in the process of melding B with R_a its possible that the resulting heap has rank larger than L_a. So in order to maintain the leftist property at every node, $mkLeftistNode$ just sets the child with larger rank (where your two options in this case are L_a and $meld(R_A, B)$) to be the left child and the other as the right. Given this, we can be sure that the node formed at every subheap of the newly melded PQ satisfies the leftist property.

In our defintion of $meld$, we only ever recurse on the right spine of either heap. That is, the recursive call in $meld(A, B)$ is always either $meld(B, R_a)$ or $meld(A, R_b)$. Herein lies the beauty of the Leftist Rank Lemma. Since each call to $meld$ does constant work (one comparison and a call to $mkLeftistNode$ which is constant work because we can extract the rank of a root in constant time), and we only traverse down the right spine of either tree, we have that the overall work of $meld(A, B)$ is $O(\log |A| + \log |B|)$ since the Leftist Rank Lemma dictates that the length of the right spine of any leftist heap h is $O(\log |h|)$.

4 Proofs of Correctness

We defined a valid leftist heap as binary tree which satisfies both the minheap and leftist priority. Given our inductive defintion above, nothing about it actually enforces that the tree is corresponds is indeed a valid leftist heap, so we define the following predicate (which is dependent on two other predicates) to qualify an arbitrary instance of a leftist heap as being valid.

```
def validLeftistHeap {β : Type} (h : leftistHeap β) := minHeap h ∧ leftistProperty h
```

Recall that these two properties of valid leftist heaps do two very different things. The minheap property is what actually enforces that our heap functions correctly, with the minimum priority node at the root of any subtree. The only thing the leftist property does is enforce a shape invariant which we argued above gives us the desired time bounds on our operations.

We've already argued that given a valid leftist heap (one which would satisfy this predicate), we get correct PQ operations and desired time bounds. So in order to prove our implementation correct, it suffices to show all of our interface operations maintain this valid leftist heap property. That is, all interface functions, when given a valid leftist heap as argument, return a valid leftist heap. Taking this one step further, we know that both *insert* and *deleteMin* both are basic operations wrapped around the *meld* operation, meaning proving their correctness follows easily from proving correctness of *meld*. So every heap operated on by our implementation is either an empty heap (which is trivially valid), or the result of two valid leftist heaps melded together. Thus we can prove correctness by proving that *meld* respects the validity of leftist heaps. This is formalized below.

```
theorem meld_valid {β : Type} : ∀ (a b : leftistHeap β),
  validLeftistHeap a → validLeftistHeap b → validLeftistHeap (meld a b) := by ...
```

Since the validity predicate is just asserting that both the minheap and leftist property we address the proofs of each these separately.

4.1 Proof Structure for Priority Queue Specification

Proving that our PQs are correct just means we enforce that all heaps we operate on enforce the minheap property, which is defined below.

```
inductive forAllHeap {β : Type} (p : β → Int → Prop) : leftistHeap β → Prop
| leaf : forAllHeap p leaf
| node left key priority rank right :
    p key priority →
    forAllHeap p left →
    forAllHeap p right →
    forAllHeap p (node left key priority rank right)

inductive minHeap {β : Type} : leftistHeap β → Prop
| leaf : minHeap leaf
| node left key priority rank right:
    forAllHeap (fun _ pL => priority ≤ pL) left →
    forAllHeap (fun _ pR => priority ≤ pR) right →
    minHeap left →
    minHeap right →
    minHeap (node left key priority rank right)
```

The *forAllHeap* predicate merely enforces that some predicate over the key-priority pairs holds at every node in a heap, and the *minHeap* predicate leverages it to enforce that all descendants of a node have higher priority and themselves are valid minheaps. Proving correctness of our PQs resolves to proving the empty node is a minHeap, and that melding two valid leftist heaps results in a minheap. This is formalized below.

```
theorem meld_minHeap {β : Type} : ∀ (a b : leftistHeap β),
  validLeftistHeap a → validLeftistHeap b → minHeap (meld a b) := by ...
```

The proof proceeds by nested induction, inducting first over a then again over b within the inductive case of a. Thus we have shown that all heaps used in our implementation are indeed minheaps, and therefore correct PQs.

4.2 Proof Structure for Asymptotic Complexity

The idea is much the same here as in the previous section, except we're proving that *meld* preserves the leftist property.

```
inductive leftistProperty {β : Type} : leftistHeap β → Prop
| leaf : leftistProperty leaf
| node left key priority rk right :
    rank right ≤ rank left →
    leftistProperty left →
    leftistProperty right →
    leftistProperty (node left key priority rk right)
```

The predicate just asserts that for every node, its left subtree has rank at least the rank of the right subtree. In terms of actual proof structure, it is nearly identical to the proof of `meld_minHeap` above. We merely show that melding preserves both the minheap and leftist property, and therefore preserves validity. Given this, now we know all of our heaps are valid. The only thing that remains is show that the Leftist Rank Lemma holds in our implementation of a valid leftist heap, and we have officially proven correctness.

Firstly, we'll provide the English proof of Leftist Rank Lemma and then show its formalization in our implementation. To prove the lemma, we first prove another lemma relating the size of a valid leftist heap to its rank. Recall that the size of a binary tree / leftist heap is 0 if its a leaf, or 1 plus the sum of the sizes of its children.

Lemma. *Given a valid leftist heap h, we have that $size(h) \geq 2^{rank(h)} - 1$*

Proof. We proceed by induction on h. If h is a leaf, then its rank and its size are zero, and indeed $0 \geq 2^0 - 1 = 0$. In the inductive case, $h = node(L, \_, R)$, and the inequality is shown as follows.

$$
\begin{aligned}
size(h) &= 1 + size(L) + size(R) && \text{(by definition of size)} \\
&\geq 1 + 2^{rank(L)} - 1 + 2^{rank(R)} - 1 && \text{(by IH)} \\
&= 2^{rank(L)} + 2^{rank(R)} - 1 \\
&\geq 2^{rank(R)} + 2^{rank(R)} - 1 && \text{(by the leftist property, } rank(L) \geq rank(R)\text{)} \\
&= 2^{1+rank(R)} - 1 = 2^{rank(h)} - 1 && \text{(by definition of rank)}
\end{aligned}
$$

□

The existence of this lemma makes proving the Leftist Rank Lemma simple and clean.

Theorem. *Given a valid leftist heap h of size n, we have that* $rank(h) \leq \log_2(n+1)$

Proof. We merely rewrite our equation using the lemma above. Then $size(h) = n \geq 2^{rank(h)} - 1 \implies n + 1 \geq 2^{rank(h)} \implies rank(h) \leq \log_2(n+1)$. □

Our argument in section 3 for the time bounds of *meld* was based entirely on the Leftist Rank Lemma. So if we can show that the Lemma holds in our implementation, then we have shown that our implementation meets the desired time bounds.

Our implementation does not use the recursive definition of rank required to prove this lemma, since we were storing at the node. However, we define an auxiliary rank function which uses the recursive definition, and show these two are equivalent. From there, we are free to write our proofs.

```
def recursiveRank {β : Type} : leftistHeap β → ℕ
  | leaf => 0
  | node _ _ _ _ right => 1 + recursiveRank right

lemma rankEntries {β : Type} : ∀ (a : leftistHeap β),
  validLeftistHeap a → size a ≥ 2 ^ (rank a) - 1 := by ...

theorem leftistRank {β : Type} : ∀ (a : leftistHeap β),
  validLeftistHeap a → rank a ≤ Nat.log2 (size a + 1) := by ...
```

Therefore by our argument in section 3, we have shown that *meld* is $O(\log|a| + \log|b|)$, and therefore all operations have desired complexity.

5 Conclusion

I provided the definition and motivation of priority queues and argued for the merits of the leftist heap implementation. Leftist heaps are desirable because of their efficient support for the *meld* operation, which allows us to implement other operations in logarithmic time bounds with minimal effort. After defining the leftist heap data structure in Lean, I provided proofs of correctness for both priority queue operations and asymptotic complexity.

References

[1] De Moura, L., Kong, S., Avigad, J., Van Doorn, F., & von Raumer, J. (2015). The Lean theorem prover (system description). *In Automated Deduction-CADE-25: 25th International Conference on Automated Deduction, Berlin, Germany, August 1-7, 2015, Proceedings 25* (pp. 378-388). Springer International Publishing.

[2] Lean Community. Lean and its Mathematical Library. https://leanprover-community.github.io/

[3] Acar, U. A., & Blelloch, G. E. (2022). Chapter XI: Priority Queues. In Algorithms: Parallel and Sequential. Retrieved from https://www.cs.cmu.edu/∼15210/docs/book.pdf.