Chapter 44

Priority Queues

A priority queue maintains a set of elements from a total ordering, allowing at least insertion of a new element and deleting and returning the minimum element. We used priority queues in priority-first graph search , in Dijkstra's algorithm , and in Prim's algorithm for minimum spanning trees.

In this chapter we focus on meldable priority queues which support a $meld$ function that can meld (or merge) two priority queues into one.

Data Type 44.1 (Meldable Priority Queue)**.** Given a totally ordered set $\mathbb{S}$, a Meldable Priority Queue (MPQ) is a type $\mathbb{T}$ representing subsets of $\mathbb{S}$, along with the following values and functions:

$$
\begin{array}{lcl}
\mathit{empty} & : & \mathbb{T} \\
\mathit{singleton} & : & \mathbb{S} \to \mathbb{T} \\
\mathit{findMin} & : & \mathbb{T} \to (\mathbb{S} \cup \{\bot\}) \\
\mathit{insert} & : & \mathbb{T} \times \mathbb{S} \to \mathbb{T} \\
\mathit{deleteMin} & : & \mathbb{T} \to \mathbb{T} \times (\mathbb{S} \cup \{\bot\}) \\
\mathit{meld} & : & \mathbb{T} \times \mathbb{T} \to \mathbb{T} \\
\mathit{fromSeq} & : & \mathbb{S}\ \mathit{seq} \to \mathbb{T}
\end{array}
$$

The function $singleton(e)$ creates a priority queue with just the element e. The function $findMin(Q)$ returns the minimum element, or $\bot$ (or `None`) if the queue is empty. The function $deleteMin(Q)$ removes the minimum value and returs the new queue along with the value. If the queue is empty it returns $\bot$ (or `None`). The $meld(Q_1, Q_2)$ creates a priority queue with the union of the elements in Q_1 and Q_2.

Algorithm 44.2 (Heapsort)**.** A priority queue can also be used to implement a version of selection sort, often referred to as ***heapsort***. The sort can be implemented by inserting all keys into a priority queue, and then removing them one by one, as follows.

```
sort S =
  let  q0 = Sequence.iter PQ.insert PQ.empty S
       hsort q =
         case PQ.deleteMin q of
           (_, None) ⇒ ⟨ ⟩
         | (q', Some (v)) ⇒ Seq.append ⟨ v ⟩ (hsort q')
  in hsort q0 end
```

The heapsort algorithm is completely sequential, but given $O(\log n)$ work implemations of *insert* and *deleteMin*, and using lists or tree sequences (so the append is cheap) does optimal $O(n \log n)$ work.

Priority queues have many applications beyond heapsort and priority first search in graphs, including:

- Huffman codes,
- clustering algorithms,
- event simulation, and
- kinetic algorithms for motion simulation.

Another function that is sometimes useful is *decreaseKey* that decreases the value of a key.

1 Implementing Priority Queues

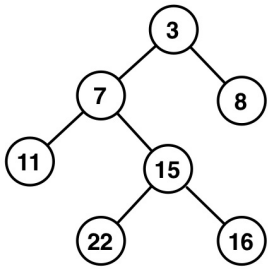
Priority queues can be implemented by using a variety of data structures.

Linked lists or Arrays. Perhaps the simplest implementation would be to use a sorted or unsorted linked list or an array. In such implementations, one of *deleteMin* and *insert* is fast and the other is slow, perhaps unacceptably so as it can take as much as $\Omega(n)$ work and span, where n is the size of the priority queue.

Balanced Trees. Another implementation is to use balanced binary search trees (e.g., treaps, or red-black trees). With balanced binary trees a *deleteMin* has to find the leftmost key and delete it. This can easily be implemented with $O(\log n)$ work and span. An *insert* is just a tree insert, and again takes $O(\lg n)$ work and span. However doing a *meld* on two heaps requires running a union on two trees, which can require $O(n)$ work if both heaps have size n.

Heaps. Many implementations of priority queues are based on the idea of a heap. As *min-heap* is a rooted tree such that the key stored at every node is less than or equal to the keys of all its descendants. Similarly a max-heap is one in which the key at a node is greater or equal to all its descendants. Compared to binary search trees, which need to maintain a total order over the search keys, heaps need only maintain a partial ordering over the keys. There are several kinds of heaps including complete binary heaps (briefly described below), leftist heaps (described in the next section), binomial heaps, pairing heaps, and Fibonacci heaps.

Example 44.1. An example min-heap illustrated.



Binary Heaps. A (complete) binary heap is a particular implementation of a heap that maintains two invariants:

- Shape property: A complete binary tree (all the levels of the tree are completely filled except the bottom level, which is filled from the left).
- Heap property.

Because of the shape property, a binary heap can be maintained in a sequence with the root in position 0 and the following simple functions for determining the left child, right child and parent of the node at location i:

$$\begin{aligned} &\mathit{left}\ i = 2 \times i + 1 \\ &\mathit{right}\ i = 2 \times i + 2 \\ &\mathit{parent}\ i = \lceil i/2 \rceil - 1 \end{aligned}$$

If the resulting index is out of range, then there is no left child, right child, or parent, respectively.

Insertion can be implemented by adding the new key to the end of the sequence, and then traversing from that leaf to the root swapping with the parent if less than the parent. Deletion can be implemented by removing the root and replacing with the last key in the sequence, and then moving down the tree if either child of the node is less than the key at the node. Binary heaps have the same asymptotic bounds as balanced binary search trees, but are likely faster in practice if the maximum size of the priority queue is known ahead of time. If the maximum size is not known, then some form of dynamically sized array is needed.

Cost Summary. The table below summarizes the costs of different implementations of priority queues, including leftist heaps covered later in this chapter, and their costs on the four key functions. Note that, a big win for leftist heaps is in the super fast $meld$ operation—logarithmic as opposed to roughly linear in other data structures.

	$insert$	$deleteMin$	$meld$	$fromSeq$
Unsorted List	$O(1)$	$O(n)$	$O(m+n)$	$O(n)$
Sorted List	$O(n)$	$O(1)$	$O(m+n)$	$O(n \log n)$
Balanced Trees	$O(\log n)$	$O(\log n)$	$O(m \log(1+\frac{n}{m}))$	$O(n \log n)$
Binary Heaps	$O(\log n)$	$O(\log n)$	$O(m+n)$	$O(n)$
Leftist Heap	$O(\log n)$	$O(\log n)$	$O(\log m + \log n)$	$O(n)$

2 Meldable Priority Queues

This section presents an implementation of a meldable priority queue that has the same work and span costs as binary search trees or binary heaps for insertion and deleting the minimum, but also has an efficient $meld$. In particular the $meld$ function takes $O(\log n + \log m)$ work and span, where n and m are the sizes of the two priority queues to be merged.

The structure we consider is called a "leftist heap", which is a binary tree that maintains the heap property, but unlike binary heaps, it does not maintain the complete binary tree property.

There are two important properties of a min-heap:

1. The minimum is always at the root.
2. The heap only maintains a partial order on the keys (unlike a BST that maintains the keys in a total order).

The first property allows us to access the minimum quickly, and it is the second that gives us more flexibility than available in a BST.

Let us consider how to implement the three operations $deleteMin$, $insert$, and $fromSeq$ on a heap. Like $join$ for binary search trees, the $meld$ operation, makes the other operations easy to implement.

To implement $deleteMin$ we can simply remove the root and $meld$ the two subtrees rooted at the children of the root.

To implement $insert(Q, v)$, we can just create a singleton node with the value v and then meld it with the heap for Q.

With $meld$, implementing $fromSeq$ in parallel is easy using $reduce$:

$$
\begin{array}{l}
fromSeq\ S = \\
\quad Seq.reduce\ Q.meld\ Q.empty\ (Seq.map\ Q.singleton\ S)
\end{array}
$$

This is parallel, and assuming $meld$ takes logarithmic work in the input sizes, this requires only $O(n)$ work and $O(\log^2 n)$ span.

Implementing Meld. The only operation we need to care about, therefore, is the $meld$ operation. Suppose that we are given twe heaps to meld. By inspecting the the roots of the heaps, we can determine that the smaller one will be the root of the new melded heap. Thus, all we have to do now is construct the left and the right subtrees of the root. At this point, we have three trees to consider—the left-subtree and the right-subtree of the chosen root, and the other tree. Let us keep the left subtree in its place—as the left-subtree of the new root—and construct the right subtree by melding the two remaining trees. We can then construct the right subtree by a recursive application of the algorithm, until we encounter trivial trees such as an empty tree. In summary, to meld two heaps, we choose the heap with the smaller root and meld the other heap with its right subtree.

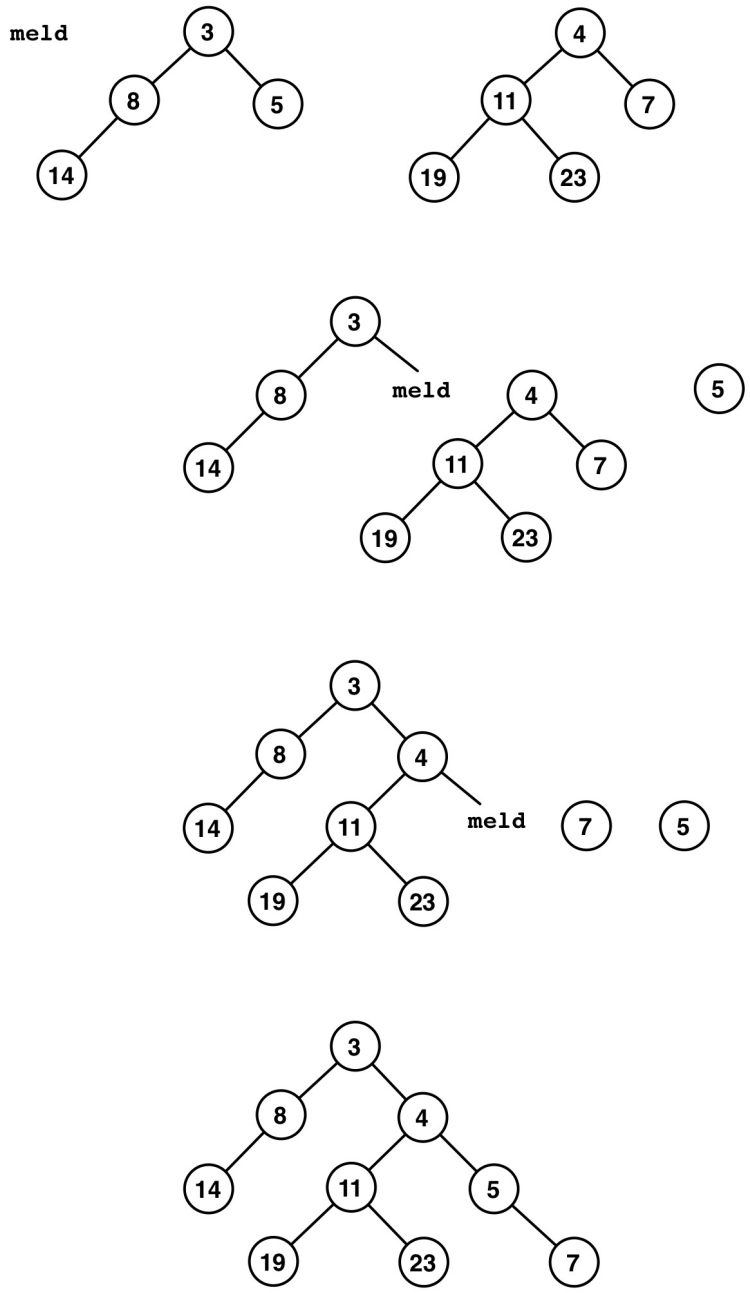
This idea leads to the following algorithm.

Data Structure 44.3 (Naïve Meldable Binary Heap)**.**

```
datatype PQ = Leaf
            | Node of (key × PQ × PQ)
meld(A, B) =
  case(A, B)
    (_, Leaf) ⇒ A
  | (Leaf, _) ⇒ B
  | (Node(k_a, L_a, R_a), Node(k_b, L_b, R_b)) ⇒
      if k_a < k_b
      then Node(k_a, L_a, meld(R_a, B))
      else Node(k_b, L_b, meld(A, R_b))
empty = Leaf
singleton(k) = Node(k, Leaf, Leaf)
insert(k, Q) = meld(singleton k, Q)
deleteMin(Q) =
  case Q of
    Leaf ⇒ (Q, None)
  | Node(k, L, R) ⇒ (meld(L, R), Some k)
fromSeq S =
  Seq.reduce meld empty (Seq.map singleton S)
```

Cost of Naïve Meld. The $meld$ algorithm traverses the right spine of each tree (recall that the right spine of a binary tree is the path from the root to the rightmost node). The problem is that the tree could be very imbalanced, and in general, we can not put any useful bound on the length of these spines—in the worst case all nodes could be on the right spine. In this case the $meld$ function could take $\Theta(|A| + |B|)$ work.

Example 44.2. An example $meld$ operations on two heaps illustrated.



2.1 Leftist Heaps

Fixing the Imbalance Problem. It turns out there is a relatively easy fix to the imbalance problem . The idea is to keep the trees so that the trees are always deeper on the left than the right. To implement this idea, we associate a "rank" with each node in the binary tree

and ensure during a meld operation that the tree is deeper on the left.

Definition 44.4 (Rank)**.** The *rank* of a node x is

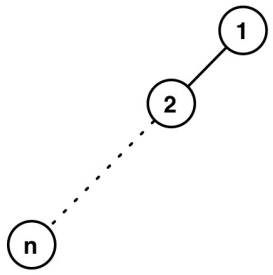
$$rank(x) = \#\text{ of nodes on the right spine of the subtree rooted at } x,$$

and more formally:

$$\begin{array}{lcl} rank(Leaf) & = & 0 \\ rank(node(\_, \_, R) & = & 1 + rank(R) \end{array}$$

Definition 44.5 (Leftist Property and Leftist Heaps)**.** A leftist heap is a heap where the ***leftist property*** holds: for any node x in the heap, $rank(L(x)) \geq rank(R(x))$, where $L(x)$ and $R(x)$ are the left and the right child of x respectively.

Example 44.3. An example leftist heap.



Note. At an intuitive level, the leftist property implies that most of entries (mass) will pile up to the left, making the right spine of such a heap relatively short. Leftist heaps can therefore be extremely unbalanced. This is fine, however, because the data structure will always work on the shorter paths in the tree. We will make this idea precise in the following lemma which will prove later; we will see how we can take advantage of this fact to support fast meld operations.

Data Structure 44.6 (Leftist Heap)**.**

```
datatype PQ  =  Leaf
             |  Node of (int × key × PQ × PQ)
rank A =  case A of
            Leaf = 0
          | Node(r, _, _, _) = r
makeLeftistNode (v, L, R) =
  if (rank L < rank R)
  then Node(1 + rank L, v, R, L)
  else Node(1 + rank R, v, L, R)
meld (A, B) =
  case (A, B) of
    (_, Leaf) ⇒ A
  | (Leaf, _) ⇒ B
  | (Node(_, k_a, L_a, R_a), Node(_, k_b, L_b, R_b)) ⇒
      if k_a < k_b then
        makeLeftistNode(k_a, L_a, meld(R_a, B))
      else
        makeLeftistNode(k_b, L_b, meld(A, R_b))
```

Leftist heap data structure extends the naive data structure in small but important ways to ensure efficiency.

Leftist heaps maintain a rank field for every node and maintain the leftist property by "piling" trees with larger ranks to the left via the $makeLeftistNode$ function. The $meld$ algorithm takes advantage of this by recurring only into the right subtree of a node, which is guaranteed to have smaller rank. Other operations can be implemented in terms of the $meld$ operation as with the naive data structure .

Note. The only real difference between the naive data structure and leftist heaps is that the latter uses $makeLeftistNode$ to create a node and ensure that the resulting heap satisfies the leftist property (assuming the two input heaps L and R did). It makes sure that the rank of the left child is at least as large as the rank of the right child by switching the two children if necessary. To implement $makeLeftistNode$ efficiently, the data structure also maintains the rank value on each node.

To analyze the cost of leftist heaps, we start by establishing a key lemma that states that leftist heaps have a short right spine, about $\log n$ in length. We then prove the main theorem that shows that the $meld$ operation requires logarithmic work in the size of two heaps being melded.

Lemma 44.1 (Leftist Rank)**.** In a leftist heap with n entries, the rank of the root node is at most $\log_2(n+1)$.

Proof. To prove the lemma, we first prove a claim that relates the number of nodes in a leftist heap to the rank of the heap.

Claim: If a heap has rank r, it contains at least $2^r - 1$ entries.

To prove this claim, let $n(r)$ denote the number of nodes in the smallest leftist heap with rank r. It is not hard to convince ourselves that $n(r)$ is a monotone function; that is, if $r' \geq r$, then $n(r') \geq n(r)$. With that, we will establish a recurrence for $n(r)$. By definition, a rank-0 heap has 0 nodes. We can establish a recurrence for $n(r)$ as follows. Consider the heap with root note x that has rank r. It must be the case that the right child of x has rank $r-1$, by the definition of rank. Moreover, by the leftist property, the rank of the left child of x must be at least the rank of the right child of x, which in turn means that $\mathit{rank}(L(x)) \geq \mathit{rank}(R(x)) = r-1$. As the size of the tree rooted x is $1 + |L(x)| + |R(x)|$, the smallest size this tree can be is

$$\begin{aligned} n(r) &= 1 + n(\mathit{rank}(L(x))) + n(\mathit{rank}(R(x))) \\ &\geq 1 + n(r-1) + n(r-1) = 1 + 2 \cdot n(r-1). \end{aligned}$$

Solving the recurrence (a full binary tree of depth r), we get $n(r) \geq 2^r - 1$, which proves the claim.

To prove our lemma, i.e., the rank of the leftist heap with n nodes is at most $\log(n+1)$, we simply apply the claim. Consider a leftist heap with n nodes and suppose it has rank r. By the claim it must be the case that $n \geq n(r)$, because $n(r)$ is the fewest possible number of nodes in a heap with rank r. But then, by the claim above, we know that $n(r) \geq 2^r - 1$, so

$$n \geq n(r) \geq 2^r - 1 \implies 2^r \leq n+1 \implies r \leq \log_2(n+1).$$

This concludes the proof that the rank of a leftist heap is $r \leq \log_2(n+1)$. □

Theorem 44.2 (Leftist Heap Work)**.** If A and B are leftists heaps then the $\mathit{meld}(A, B)$ algorithm runs in $O(\log(|A|) + \log(|B|))$ work and returns a leftist heap containing all elements from A and from B.

Proof. The code for meld only traverses the right spines of A and B, advancing by one node in one of the heaps. Therefore, the process takes at most $\mathit{rank}(A) + \mathit{rank}(B)$ steps, and each step does constant work. Since both trees are leftist, by the main lemma , the work is bounded by $O(\log(|A|) + \log(|B|))$. To prove that the result is leftist we note that the only way to create a node in the code is with $\mathit{makeLeftistNode}$. This function guarantees that the rank of the left branch is at least as great as the rank of the right branch. □