Priority Queues and Leftist Heaps

1 Priority Queues

By this point, you have seen several algorithms that make use of a data structure known as a **priority queue**. Both Dijkstra's algorithm and Prim's algorithm used priority queues to consider vertices in a specific order. The central idea behind a priority queue is that it stores a set of keys and provides access to the smallest (or largest) key. Similarly to binary search trees, the key type must therefore be comparable.

Specifically, a priority queue supports two functions—an `insert` function to efficiently insert a new key, and a `deleteMin` function to retrieve and delete the element in the priority queue with the smallest (or largest) key.

In 15-122, you probably saw a data structure known as a binary heap. This data structure maintained the invariant that the smallest item was always at the root of the heap, and that the key of any node is smaller than keys of its children nodes. This is the same as the heap invariant that we used previously with treaps (and of course how treaps got their name).

In this lecture, we will show how to efficiently support a third function on priority queues, known as the `meld` operation. It combines the contents of two priority queues into a single priority queue. In doing so, we will explore various implementations for priority queues, and discuss their advantages and drawbacks.

1.1 Interface and Implementations

In its simplest form, a **priority queue** is a data structure that stores keys. The only requirement for our key type is that it must be comparable, i.e., the elements of the key type must have a total ordering.

Interface: Priority Queue

Priority queues typically support two operations:

- `insert`, which adds a new key into the priority queue,
- `deleteMin`, which returns the smallest key from the queue and removes it

One way to initialize a priority queue if given a sequence of n items is to just call do `insert` n times. However, in some cases we can build a priority queue more efficiently, so we also will look at the cost to initialize such a priority queue from a sequence of keys.

Let's consider some possible implementations for a priority queue.

Balanced BST The first one we will consider is using a balanced binary search tree as a priority queue. Since BSTs already store their keys in sorted order, `deleteMin` can be implemented by finding the leftmost node in the tree in $O(\log n)$ time, i.e., what we called `first` in our description of BST operations, then deleting that element. Insertions are also easy – they take $O(\log n)$ time, as we have seen previously. Finally, converting a sequence to a tree takes $O(n \log n)$ time.

Binary Heap If we use a binary heap, then inserting a new key into the queue takes $O(\log n)$ time, since we have to insert it in the correct position at the bottom layer, and sift it up to its proper location. Deletions take $O(\log n)$ time as well, since the lowest-priority node is at the root, and we have to restore the shape invariant after removing it. Converting a sequence to a binary heap can actually be done in $O(n)$ time (this operation is called "heapifying", making them slightly more efficient than just using a balanced tree.

There are also other types of heaps, such as **leftist heaps**, which we will explore later. These heaps have bounds similar to those of a binary heap, but also come with special properties.

Inefficient Methods There are also other, less efficient representations for priority queues. For example, we can keep all the keys in an unsorted list, which takes $O(1)$ time to insert to but $O(n)$ time for deletions. Alternatively, we could use a sorted list, which would take $O(n)$ time for the insertion operation but lower the cost of the deletions to $O(1)$. Both of these options come with trade-offs, but since either delete or insert is linear time, we typically won't consider these as options for our priority queue implementations.

1.2 The Meld Operations

Now that we have been introduced to some implementations of priority queues, we can consider a third operation on them. We would like to support a **meld** operation, which essentially acts as a union of two priority queues. If we call meld (Q_1, Q_2) and get an output of Q_3, then any key that is in either of the two input queues must be included in the output queue.

How well do our previous implementations of priority queues work with melding? Let's say I meld two queues of sizes m and n, where $m \leq n$. If these priority queues are represented using balanced binary trees, then we can use the union operation to meld them, which gives us a work bound of $O\left(m + m\log(n/m)\right)$. If we are using binary heaps, then our bound comes out to $O(m \log n)$, since we simply delete each element from the smaller priority queue and insert it into the larger one.

Neither of these efficient representations are particularly good at melding, and the inefficient ones do not fare much better. If we are using an unsorted list representation for our priority queue, then it will take $O(m)$ time to meld two queues, since we can append the shorter list to the longer one. If we instead maintain the invariant that the lists are sorted, then we will need to merge the two sorted lists which takes $O(m + n)$ work.

Ideally, we would like to have meld be logarithmic time. One data structure that supports a logarithmic-time meld is the leftist heap, which is the data structure we will be introducing in this lecture. Its bounds can be seen in the table below.

Theorem: Priority Queue Cost Reference

For insert, deleteMin, and fromSeq, n is the size of the priority queue. For meld, m is the size of the smaller queue, and n is the size of the larger queue being melded.

Representation	**insert**	**deleteMin**	**fromSeq**	**meld**
Unsorted list	$O(1)$	$O(n)$	$O(n)$	$O(m)$
Sorted list	$O(n)$	$O(1)$	$O(n \log n)$	$O(m+n)$
Binary search tree	$O(\log n)$	$O(\log n)$	$O(n \log n)$	$O\left(m + m\log(n/m)\right)$
Binary heap	$O(\log n)$	$O(\log n)$	$O(n)$	$O(m \log n)$
Leftist heap	$O(\log n)$	$O(\log n)$	$O(n)$	$O(\log(m+n))$

Leftist heaps are not the only other type of heap that we can use to implement a priority queue. There are also Fibonacci heaps and brodal heaps, which also can be used for priority queues.

2 Uses of meld

As it turns out, once we have implemented the meld operation, we can use it to implement many other common operations! Let's take a look at how we might be able to do this on a generic heap data structure, which requires the preservation of the heap invariant. We define our type of priority queues, PQ, as follows:

Definition: PQ Type

```
type PQ<K: Ordered> =
    Empty
  | Node { left: PQ,
           key: K,
           right: PQ  }
```

That is, a priority queue is either empty, or consists of a root node with left and right subheaps. This is similar to the heaps that you are familiar with, just written in a functional style.

Creating a heap from a single key k just means creating a node with two Empty children:

```
fun singleton(k : K): return Node(Empty, k, Empty)
```

Once we are able to create singletons, we can insert a new key into an existing priority queue easily, by creating a singleton and melding it in:

```
fun insert(Q : PQ<K>, k : K) -> PQ<K>:
  return meld(Q, singleton(k))
```

`deleteMin` is a bit more complicated, particularly in a functional style. It relies on the fact that the minimum-priority element is currently stored at the root of the priority queue Q. Additionally, because we want a functional/immutable implementation, `deleteMin` returns a pair of things—the first is the minimum-priority key (or NONE if the queue is empty), and the second is the remainder of the queue after the deletion. The implementation of `deleteMin` looks like:

```
fun deleteMin(Q : PQ<K>) -> (option<K>, PQ<K>):
```

```
  match Q with:
    case Empty: return (NONE, Empty)
    case Node(L, k, R):
      return (SOME(k), meld(L, R))
```

Given a `meld` operation, `fromSeq` is actually also very easy to implement! All we need to do is turn each of the elements into a singleton priority queue, and then meld the small queues together into one big priority queue. We can implement this using the map-reduce pattern:

```
fun fromSeq(S : sequence<K>) -> PQ<K>:
  return reduce(meld, Empty, map(singleton, S))
```

Having a meld function that preserves the heap invariant can be very powerful, as you can see from these examples! If we assume that we have an $O(\log(n+m))$ implementation of meld, then the `insert` function will have work $O(\log(n+1)) = O(\log n)$. The work of `deleteMin` will be $O\big(\log(n/2+n/2)\big) = O(\log n)$.

Finally, the work of `fromSeq` can be found via a recurrence. If we think about the recursive call pattern of the reduce function (divide-and-conquer), we notice that the root will be melding the two priority queues created from each half of the sequence, which costs $O(\log n)$. It also makes two recursive calls of half the size, so we get:

$$W(n) = 2W\left(\frac{n}{2}\right) + O(\log n)$$

This recurrence is leaf-dominated. Each leaf takes constant time, and there are $O(n)$ leaves, so our overall bound by the brick method is $O(n)$. This explains how leftist heaps are able to meet the bounds stated previously!

Moreover, this fromSeq implementation is parallelizable, since it uses the reduce function. If we assume that the span of meld is the same as its work, we get the recurrence:

$$S(n) = S\left(\frac{n}{2}\right) + O(\log n)$$

This is a balanced recurrence. There are $O(\log n)$ levels, and the cost per level is also $O(\log n)$, so the overall span is $O(\log^2 n)$.

2.1 A Slow Meld

Let's say we have two heaps, A and B, which we are trying to meld. We don't know anything about them other than that they have type PQ, as before, and that the key at the root of a heap is always smaller than any key in one of its subheaps (i.e. the standard heap invariant). We can implement meld in a way that does not meet the time bound, as follows:

Algorithm: Melding Two Heaps

```
fun meld (A: PQ<P>, B: PQ<P>)-> PQ<P>:
  match A, B with:
    case Empty, _: return B
    case _, Empty: return A
```

```
    case Node(L_a, k_a, R_a), Node(L_b, k_b, R_b):
      if (k_a < k_b):
        return makeNode(L_a, k_a, meld(R_a, B))
      else:
        return makeNode(L_b, k_b, meld(R_b, A))
```

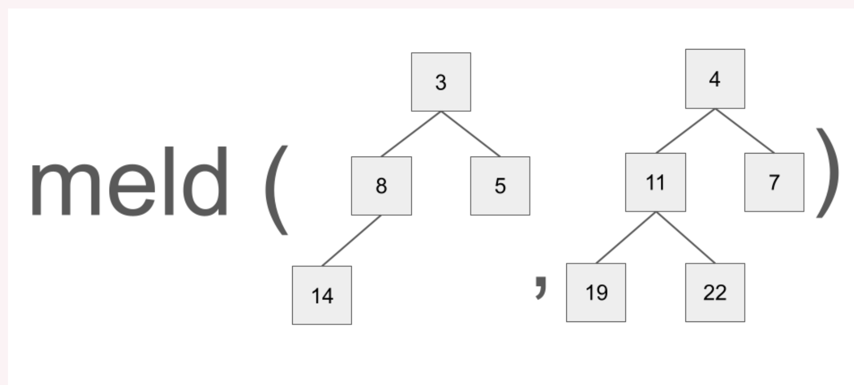
This algorithm has two simple base cases, which come when one of the two heaps is empty – in which case we return the other one. Otherwise, we have two nodes, each with their own key. Out of these two keys, we choose the one with the lower priority to be the new root key. We then recursively meld the other heap into the right branch of the chosen heap.

Let's take a look at what this meld is doing on some actual heaps.

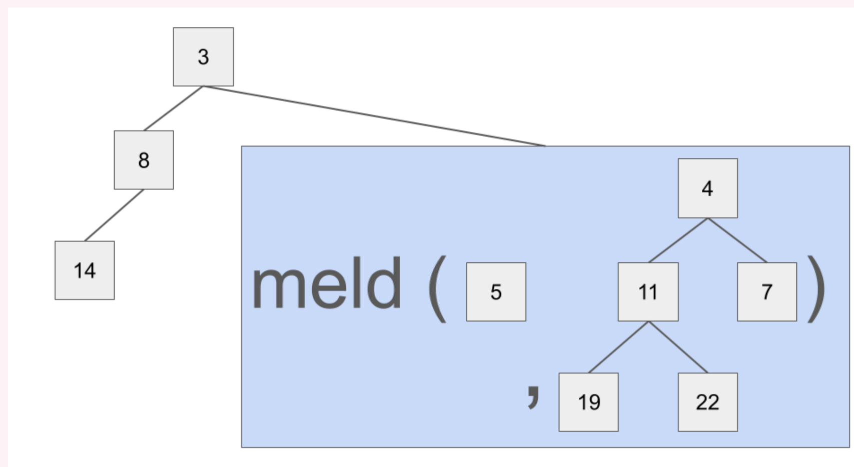
Example: Melding Heaps

Consider running the meld operation on these two priority queues storing integer keys.

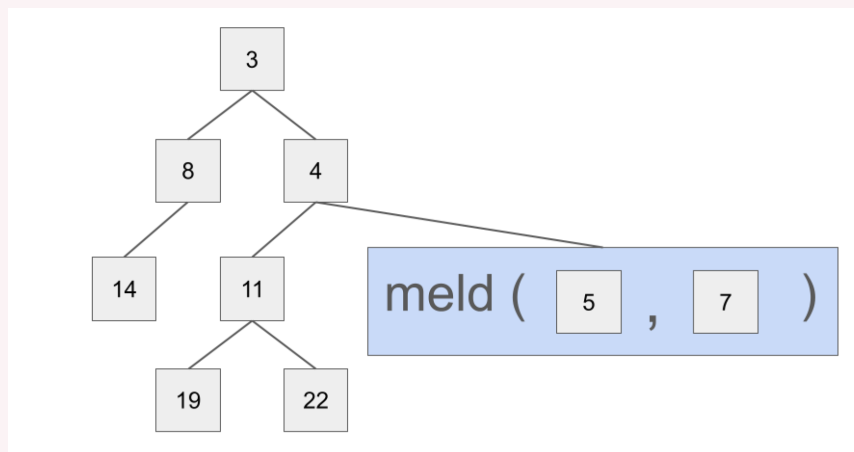
Our initial call to meld is as follows:



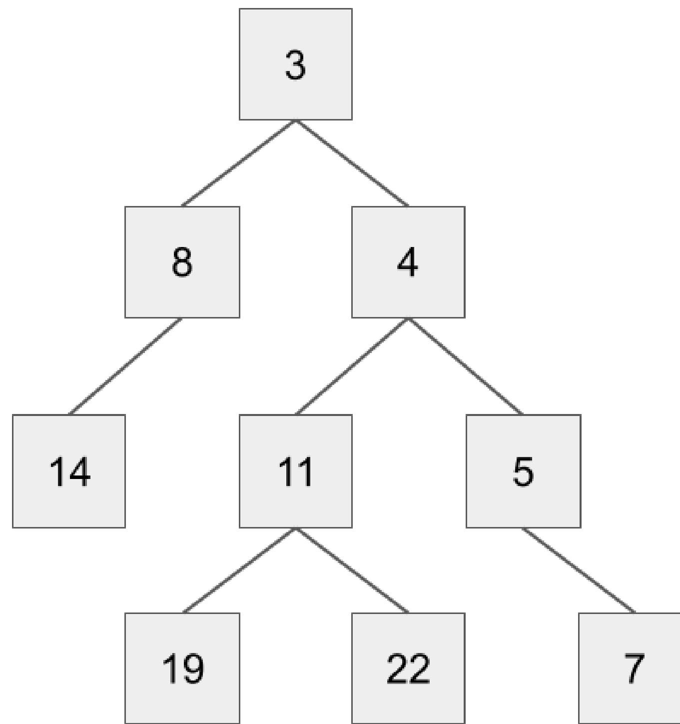
At this point, we compare the keys at the root of each priority queue. 3 is less than 4, so the result will have 3 at the root, and the left subtree is the same as 3's left subtree. The right subtree is the result of recursively melding 3's right subtree (which is only 5) and the entire other heap. This recursive call is represented by the blue box.



To execute the recursive call to meld, we once again compare the priorities of the keys at the roots. 4 is less than 5, so we choose 4 to be the root of this result. 4's left subtree then becomes the left subtree of the result. The right subtree comes from merging the right subtree of 7 with the other heap, which is 5.



Now, we need to meld 5 and 7 together. 5 is the smaller of the two, so we make it the root of the result. The right subtree is the empty heap melded with 7, which is just 7. This gives us our final answer of:



Indeed, we can verify that this result satisfies the heap invariant, since every node has a priority that is smaller than that of any of its descendants.

Unfortunately, this meld can be expensive, since it makes recursive calls all the way down the right spines of both trees. The example we gave was fairly balanced, but if we were given two priority queues that were both long right spines, then our meld operation could take linear time – which is much worse than the logarithmic-time meld that we were looking for.

3 Leftist Heaps

The **leftist heap** data structure is similar to a binary heap because it arranges its nodes in a manner that satisfy the heap invariant—there will never be a key at any node that is greater than one of its descendants. Leftist heaps aim to support a fast meld operation that is implemented identically to the one we just saw. It does this without any balance invariants or height limitations on our heap as well!

Our leftist heaps are inspired by the meld function that we wrote earlier, and how it behaves. When we wrote meld, we made the choice to always make recursive calls on the right subtrees, rather than the left ones. This meant that if k_a was smaller, then we would meld B with R_a, and likewise, we melded A with R_b in the symmetric case. We never made a recursive call to meld on L_a or L_b. You can see this in the example as well—meld always operated on what we call

the **right spine** of the two trees.

This allows us to keep the cost of meld low, as long as we are able to somehow limit the lengths of the right spines. For example, we could have two very large trees A and B which are each structured as one long left spine—the right child of every node is empty. Melding these two heaps together costs constant time, since we compare the root priorities and then make a singular recursive call, since meld is linear in the total length of the two right spines.

Thus, our heap only needs to maintain the property that its right spine is relatively short—any other parts of the tree can be as large as they want. To accomplish this, we define a couple of key terms.

Definition: Rank of a Heap

We say that the **rank** of a heap is equal to the number of nodes in its right spine.

Once we have defined ranks, we can define the leftist property:

Definition: Leftist Property

We say that a heap is **leftist** if for every node (L, k, R) in the heap, it holds that the rank of L is greater than or equal to the rank of R.

It follows that a leftist heap is a heap for which the leftist property holds. Now that we have defined leftist heaps at a conceptual level, we can begin to implement them. Our datatype will look similar to the one for priority queues that we defined previously, but we will add an additional field to store the rank as an integer.

Definition: Leftist Heap PQ Type

```
type PQ<K: Ordered> =
    Empty
  | Node { left: PQ,
           key: K,
           rank: int,
           right: PQ  }
```

Now, we need a function to create a leftist heap's node. Unfortunately, we cannot use the `Node` constructor directly, since this can lead to us creating heaps that violate the leftist property. Thus, we write an auxiliary function `makeNode`, which takes two leftist heaps A and B as well as a priority p, which must be less than any priority in either A or B. This function will return a leftist heap with p at the root, and A and B as subtrees – it is not guaranteed which of these two will be the right subtree.

Algorithm: Node Construction for Leftist Heaps

```
fun makeNode(A: PQ<K>, k: K, B: PQ<K>) -> PQ<K>:
  (_, _, r_a, _) = A
  (_, _, r_b, _) = B
```

```
if (r_b < r_a):
  return Node(A, k, r_b + 1, B)
else:
  return Node(B, k, r_a + 1, A)
```

This algorithm check the ranks of the two heaps, and position them appropriately. By the leftist property, the right subtree must be the one with smaller rank.

We can then write a version of meld on leftist heaps that will take two leftist heaps and return a leftist heap as well! All we would have to do is convert our old meld code to work with the new datatype, and have it call node' instead of Node every time it constructs a new node.

3.1 Proving the meld bound

Now, we need to formally show that meld takes logarithmic work. Remember, our meld had work linear in the lengths of the right spines, thus, we only need to bound the lengths of the right spines of leftist heaps. To do this, we will prove that the length of the right spine of a leftist heap with n nodes is at most $\log_2(n+1)$. The way we will prove this is by considering the minimum size of a leftist heap with a given rank r. This will allow us to bound the rank of a leftist heap.

Lemma: Minimum Size of Leftist Heaps

Define a function $m(r)$ to be equal to the minimum number of nodes in a leftist heap with rank r. Our claim is that $m(r) = 2^r - 1$.

Proof. We show this by induction on r.

Base case: If r is 0, then the tree has no right spine, and therefore cannot even have a root node. Thus, any rank-0 tree has 0 nodes. Indeed, $2^0 - 1 = 1 - 1 = 0$, so the base case of $r = 0$ is proven.

Inductive case: Let's assume that $m(r-1) = 2^{r-1} - 1$ as our inductive hypothesis. For us to have a tree of rank r, we need to have a root node, as well as a left and right subtree. To achieve a rank of r, we need the right subtree to have rank $r - 1$. Thus, the right side must have at least $m(r-1)$ nodes. By the leftist property, the left subtree must have rank at least $r-1$. Since $m(r)$ is an increasing function, the left subtree's minimum size is also $m(r-1)$. Thus, we get that:

$$\begin{aligned} m(r) &= 1 + m(r-1) + m(r-1) \\ &= 1 + 2^{r-1} - 1 + 2^{r-1} - 1 \\ &= 2 \cdot 2^{r-1} - 1 \\ &= 2^r - 1 \end{aligned}$$

which completes the inductive step.

Corollary: $\text{rank}(Q) \leq \log_2(|Q| + 1)$ for any priority queue Q.

We have shown previously that $|Q| \geq 2^{\text{rank}(Q)} - 1$, since $m(\text{rank}(Q))$ is the minimum size of a heap with rank $\text{rank}(Q)$. Therefore, $|Q| + 1 \geq 2^{\text{rank}(Q)}$. Taking logarithms, we get that $\log_2(|Q| + 1) \geq \text{rank}(Q)$ as desired. $\square$

It follows that the length of the right spine of a leftist heap with n elements is $O(\log n)$. As a result, melding two priority queues of m and n elements should take $O(\log m + \log n)$ work, which is the same bound as $O(\log(m + n))$, giving us the logarithmic time merge that we have wanted.