Lecture 20 — Leftist Heaps and Sorting Lower Bounds

Parallel and Sequential Data Structures and Algorithms, 15-210 (Spring 2012)

Lectured by Margaret Reid-Miller — 29 March 2012

Today:

- Priority Queues and Leftists Heaps
- Sorting Lower Bounds

1 Priority Queues

We have already discussed and used priority queues in a few places in this class. We used them as an example of an abstract data type. We also used them in the priority-first graph search to implement Dijkstra's algorithm, and Prim's algorithm for minimum spanning trees.

As you might have seen in other classes, a priority queue can also be used to implement an $O(n \log n)$ work (time) version of selection sort, often referred to as heapsort. The sort can be implemented as:

```
fun sort(S) =
let
  (* Insert all keys into priority queue *)
  val pq = Seq.iter Q.insert Q.empty S

  (* remove keys one by one *)
  fun sort' pq =
    case (PQ.deleteMin pq) of
      NONE => nil
    | SOME(v,pq') => cons(v,sort'(pq'))
in
  Seq.fromList (sort' pq)
end
```

Priority queues also have applications elsewhere, including

- Huffman Codes
- Clustering algorithms
- Event simulation
- Kinetic algorithms

†Lecture notes by Guy E Blelloch, Margaret Reid-Miller, and Kanat Tangwongsan.

What are some possible implementations of a priority queue?

With sorted and unsorted linked lists (or arrays), one of `deleteMin` and `insert` is fast and the other is slow. On the other hand balanced binary search trees (e.g., treaps) and binary heaps implemented with (mutable) arrays have $O(\log n)$ span for both operations. But why would you choose to use binary heaps over balanced binary search trees? For one, binary heaps provide a `findMin` operation that is $O(1)$ whereas for BSTs it is $O(\log n)$. Let's consider how you would build a priority queue from a sequence.

But first, let's review the heaps and search trees. A *min-heap* is a rooted tree such that the key stored at every node is less 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. A *search-tree* is a rooted tree such that the key sorted at every node is greater than (or equal to) all the keys in its left subtree and less than all the keys in its right subtree. Heaps maintain only a partial ordering, whereas search trees maintain a total ordering.

A binary heap is a particular implementation that maintains two invariances:

- 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 an array, and the index of the a parent or child node is a simple computation. Recall that operations on a binary heap first restore the shape property, and then the heap property.

To build a priority queue, we can insert one element at a time into the priority queue as we did in heap sort above. With both balanced binary search trees and binary heaps, the cost is $O(n \log n)$. Can we do better? For heaps, yes, build the heap recursively. If the left and right children are already heaps, we can just "shift down" the element at the root:

```
1  fun sequentialFromSeqS =
2  let
3    fun heapify(S, i) =
4      if (i >= |S|/2) then S
5      else
6        val S' = heapify(S, 2*i + 1)
7        val S'' = heapify(S', 2*i + 2)
8        shiftDown(S'', i)
9  in heapify(S, 0) end
```

With ST-sequences, `shiftDown` does $O(\log n)$ work on a subtree of size n. Therefore, `sequentialFromSeq` has work

$$W(n) = 2W(n/2) + O(\log n) = O(n)$$

We can build a binary heap in parallel with ST-sequences. If you consider S as a complete binary tree, the leaves are already heaps. The next level up of this tree, the roots of the subtrees violate

the heap property and need to be shifted down. Since the two children of each root are heaps, the result of shift down is a heap. That is, on each level of the complete tree, fix the heaps at that level by shifting down the elements at that level. The code below, for simplicity, assumes $|S| = 2^k - 1$ for some k:

```
1  fun fromSeq S : 'a seq =
2  let
3    fun heapify (S, d) =
4    let
5      val S' = shiftDown (S, ⟨2^d − 1, ..., 2^(d+1) − 2⟩, d)
6    in
7      if (d = 0) then S'
8      else heapify (S', d − 1)
9  in heapify (S, log2 n − 1) end
```

There is a subtly with this parallel `shiftDown`. It too needs to work one layer of the binary tree at a time. That is, it takes a sequence of indices corresponding to elements at level d and determines if the those elements need to swap with elements at level $d+1$. It does the swaps using `inject`. Then it calls `shiftDown` recursively using the indices to where the elements at d moved to in level $d+1$, if indeed they moved down. When it reaches the leaves it returns the updated ST-sequence.

This parallel version does the same work as the sequential version. But now span is $O(\log n)$ at each of the $O(\log n)$ layers of the tree:

$$S(n) = S(n/2) + O(\log n) = O(\log^2 n).$$

In summary, the table below shows that a binary heap is an improvement over more general purpose structures used for implementing priority queues. The shape property of a binary heap,

Implementation	`findMin`	`deleteMin`	`insert`	`fromSeq`
sorted linked list	$O(1)$	$O(1)$	$O(n)$	$O(n \log n)$
unsorted linked list	$O(n)$	$O(n)$	$O(1)$	$O(n)$
balanced search tree	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(n \log n)$
binary heap	$O(1)$	$O(\log n)$	$O(\log n)$	$O(n)$

though, limits its ability to implement other useful priority queue operations efficiently. Next, we will a more general priority queue, meldable ones.

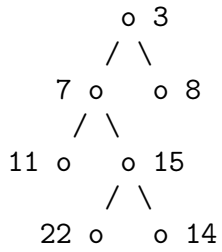
1.1 Meldable Priority Queues

Recall that, much earlier in the course, we introduced a meldable priority queue as an example of an abstract data type. It includes the `meld` operation, which is analogous to `merge` for binary search trees; It takes two meldable priority queues and returns a meldable priority queue that contains all the elements of the two input queues.

Today we will discuss one implementation of a meldable priority queue, which has the same work and span costs as binary heaps, but also has an efficient operation `meld`. This operation has work and span of $O(\log n + \log m)$, where n and m are the sizes of the two priority queues to be merged.

The structure we will consider is a 'leftist heap, which is a binary tree that maintains the heap property, but unlike binary heaps, it not does maintain the complete binary tree property. The goal is to make the `meld` fast, and in particular run in $O(\log n)$ work. First, let's consider how we could use `meld` and what might be an implementation of `meld` on a heap.

Consider the following a min-heap



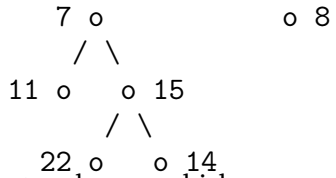
There are two important properties of a min-heap:

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's consider how to implement the three operations `deleteMin`, `insert`, and `fromSeq` on a heap. Like `join` for treaps, the `meld` operation, makes the other operations easy to implement.

To implement `deleteMin` we can simply remove the root. This would leave:



This is simply two heaps, which we can use `meld` to join.

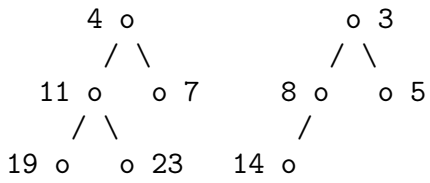
To implement `insert`(Q, v), we can just create a singleton node with the value v and then meld it with the heap for Q.

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

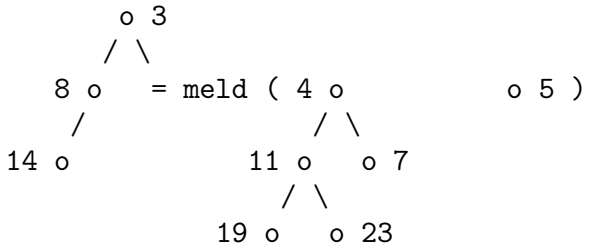
```
(* Insert all keys into priority queue *)
val pq = Seq.reduce Q.meld Q.empty (Seq.map Q.singleton S)
```

In this way, we can insert multiple keys into a heap in parallel: Simply build a heap as above and then meld the two heaps. There is no real way, however, to remove keys in parallel unless we use something more powerful than a heap.

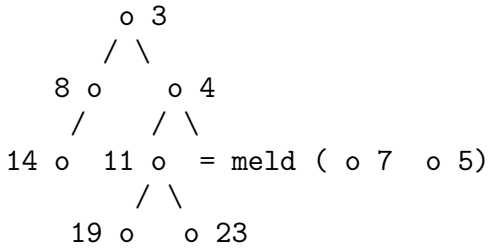
The only operation we need to care about, therefore, is the `meld` operation. Let's consider the following two heaps



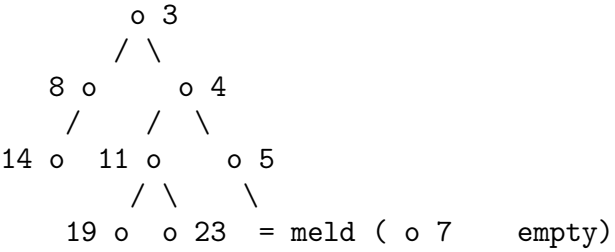
If we meld these two min-heaps, which value should be at the root? Well it has to be 3 since it is the minimum value. So what we can do is select the tree with the smaller root and then recursively meld the other tree with one of its children. In our case let's meld with the right child. So this would give us:



If we apply this again we get



and one more time gives:



Clearly if we are melding a heap A with an empty heap we can just use A. This algorithm leads to the following code

```
1  datatype PQ = Leaf | Node of (key, PQ, PQ)

2  fun meld(A, B) =
3    case (A, B) of
4      (_, Leaf) ⇒ A
5    | (Leaf, _) ⇒ B
6    | (Node(k_a, L_a, R_a), Node(k_b, L_b, R_b)) ⇒
7        case Key.compare (k_a, k_b) of
8          LESS ⇒ Node(k_a, L_a, meld(R_a, B))
9        | _ ⇒ Node(k_b, L_b, meld(A, R_b))
```

This code 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.

1.2 Leftist Heaps

It turns out there is a relatively easy fix to this imbalance problem. The idea is to keep the trees so that the trees are always deeper on the left than the right. In particular, we define the *rank* of a node x as

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

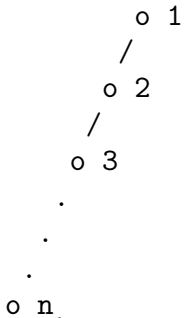
and more formally:

$$\mathsf{rank}(\mathsf{leaf}) = 0$$
$$\mathsf{rank}(\mathsf{node}(\_, \_, R) = 1 + \mathsf{rank}(R)$$

Now we require that all nodes of a leftist heap have the "leftist property". That is, if $L(x)$ and $R(x)$ are the left and child children of x, then we have:

Leftist Property: For all node x in a leftist heap, $\mathsf{rank}(L(x)) \geq \mathsf{rank}(R(x))$

This is why the tree is called leftist: for each node in the heap, the rank of the left child must be at least the rank of the right child. Note that this definition allows the following unbalanced tree.



This is OK since we only ever traverse the right spine of a tree, which in this case has length 1.

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. In this way, all update operations we care about can be supported efficiently. We'll make this idea precise in the following lemma which will prove later; we'll see how we can take advantage of this fact to support fast meld operations.

Lemma 1.1. *In a leftist heap with n entries, the rank of the root node is at most* $\log_2(n+1)$.

In words, this lemma says *leftist heaps have a short right spine,* about $\log n$ in length. To get good effiency, we should take advantage of it. Notice that unlike the binary search tree property, the heap property gives us a lot of freedom in working with left and right child of a node (in particular, they don't need to be ordered in any specific way). Since the right spine is short, our meld algorithm should, when possible, try to work down the right spine. With this rough idea, if the number of steps required to meld is proportional to the length of the right spine, we have an efficient algorithm that runs in about $O(\log n)$ work.

To make use of ranks we add a rank field to every node and make a small change to our code to maintain the leftist property: the meld algorithm below effectively traverses the right spines of the heaps A and B. (Note how the recursive call to `meld` are only with either (R_a, B) or (A, R_b).)

```
1  datatype PQ = Leaf | Node of (int, key, PQ, PQ)

2  fun rank Leaf = 0
3    | rank (Node(r, _, _, _)) = r

4  fun makeLeftistNode (v, L, R) =
5    if (rank(L) < rank(R))
6    then Node(1 + rank(L), v, R, L)
7    else Node(1 + rank(R), v, L, R)

8  fun meld (A, B) =
9    case (A, B) of
10     (_, Leaf) ⇒ A
11   | (Leaf, _) ⇒ B
12   | (Node(_, k_a, L_a, R_a), Node(_, k_b, L_b, R_b)) ⇒
13       case Key.compare(k_a, k_b) of
14         LESS ⇒ makeLeftistNode (k_a, L_a, meld(R_a, B))
15       | _ ⇒ makeLeftistNode (k_b, L_b, meld(A, R_b))
```

Note that the only real difference is that we now use `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. It also maintains the rank value on each node.

Theorem 1.2. *If A and B are leftists heaps then the* `meld`(A,B) *algorithm runs in* $O(\log(|A|) + \log(|B|)$ *work and returns a leftist heap containing the union of A and 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 $\mathsf{rank}(A) + \mathsf{rank}(B)$ steps, and each step does constant

work. Since both trees are leftist, by Lemma 1.1, 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 `makeLeftistNode`. This routine guarantees that the rank of the left branch is at least as great as the rank of the right branch. □

Before proving Lemma 1.1 we will 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'll 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 $\mathsf{rank}(L(x)) \geq \mathsf{rank}(R(x)) = r-1$. As the size of the tree rooted x is $1 + |L(x)| + |R(x)|$, the smallest size this tree can be is

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

Unfolding the recurrence, we get $n(r) \geq 2^r - 1$, which proves the claim.

Proof of Lemma 1.1. To prove that 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 $r \leq \log_2(n+1)$. □

1.3 Summary of Priority Queues

Already, we have seen a handful of data structures that can be used to implement a priority queue. Let's look at the performance guarantees they offer.

Implementation	`insert`	`findMin`	`deleteMin`	`meld`
(Unsorted) Sequence	$O(n)$	$O(n)$	$O(n)$	$O(m+n)$
Sorted Sequence	$O(n)$	$O(1)$	$O(n)$	$O(m+n)$
Balanced Trees (e.g. Treaps)	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(m \log(1 + \frac{n}{m}))$
Leftist Heap	$O(\log n)$	$O(1)$	$O(\log n)$	$O(\log m + \log n)$

Indeed, a big win for leftist heap is in the super fast `meld` operation—logarithmic as opposed to roughly linear in other data structures.