Recitation 15

Priority Queues

15.1 Announcements

- *DPLab* has been released, and is due **this Friday**.

15.2 Leftist Heaps

Task 15.1. *Identify the defining properties of a leftist heap.*

A leftist heap is a binary tree given by

datatype `tree = Leaf | Node` **of** `key × tree × tree`

which satisfies

(a) the *heap property*, requiring that the key stored at each node is smaller[1] than any descendent key, and

(b) the *leftist property*, requiring that for every $\texttt{Node}(\_, L, R)$, we have $\text{rank}(L) \geq \text{rank}(R)$. We define the *rank* of a heap to be the number of nodes in its right spine, i.e.,

$$\text{rank}(\texttt{Leaf}) = 0$$
$$\text{rank}(\texttt{Node}(\_, L, R)) = 1 + \text{rank}(R)$$

Task 15.2. *What is an upper bound on the rank of the root of a leftist heap?*

For a leftist heap containing n entries, the rank of the root is at most $\log_2(n+1)$.

[1] We assume a min-heap. In a max-heap, each key is larger than its descendents.

15.2.1 Building A Leftist Heap

Consider the following pseudo-SML code implementing leftist heaps.

Data Structure 15.3. *Leftist Heap*

```
datatype PQ = Leaf | Node of int × key × PQ × PQ

fun rank Q =
  case Q of
    Leaf ⇒ 0
  | Node (r,_,_,_) ⇒ r

fun makeLeftistNode (k, A, B) =
  if rank A < rank B
  then Node (1 + rank A, k, B, A)
  else Node (1 + rank B, k, A, B)

fun 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))

fun singleton k = Node (1,k,Leaf,Leaf)

fun insert (Q, k) = meld (Q, singleton k)

fun fromSeq S = Seq.reduce meld Leaf (Seq.map singleton S)

fun deleteMin Q =
  case Q of
    Leaf ⇒ (NONE, Q)
  | Node (_,k,L,R) ⇒ (SOME k, meld (L,R))
```

Task 15.4. *Diagram the process of executing the code*

$\texttt{fromSeq}\ \langle 3, 5, 2, 1, 4, 6, 7, 8 \rangle$

```
3 5 2 1 4 6 7 8

  3   1   4   7
 /   /   /   /
5   2   6   8

   1       4
  / \     / \
 2   3   6   7
    /       /
   5       8

        1
       / \
      3   2
     / \
    4   5
   / \
  6   7
     /
    8
```

Task 15.5. *What are the work and span of* $(\texttt{fromSeq}\ S)$ *in terms of* $|S| = n$*?*

Notice that `meld` only traverses the right spines of its arguments, each of which are logarithmic in length, and therefore $\texttt{meld}(A, B)$ requires $O(\log |A| + \log |B|)$ work and span and returns a heap of size $|A| + |B|$. This suggests the recurrences

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

both of which we have seen before; they solve to $O(n)$ work and $O(\log^2 n)$ span, respectively.

15.2.2 Dynamic Median

Task 15.6. *Design a data structure which supports the following operations:*

	Work	*Span*	*Description*
`fromSeq` S	$O(\|S\|)$	$O(\log^2 \|S\|)$	*Constructs a dynamic median data structure from the collection of keys in* S
`median` M	$O(1)$	$O(1)$	*Returns the median of all keys stored in* M
`insert` (M, k)	$O(\log \|M\|)$	$O(\log \|M\|)$	*Inserts* k *into* M

For simplicity, you may assume that all elements inserted into such a structure are distinct.

Our data structure will be a triple (L, m, G), where L is a max-heap, m is the median, and G is a min-heap. We maintain the invariant that L contains all items less than m, and symmetrically G contains all items greater than m.

To implement `fromSeq`, we use a selection algorithm (i.e. quickselect) to select the median of the sequence using linear work and log-squared span. We filter twice to create a left and right half containing all items less than and greater than the median, respectively. Perform `MaxPQ.fromSeq` and `MinPQ.fromSeq` on these halves to construct L and G.

To implement `insert`, check if $k \geq m$. If so, insert k into G. If this results in $|L|+2 = |G|$, then insert m into L, delete the minimum from G, and set it to be the new median. We do the obvious symmetric thing for the case $k < m$.

We implement `median` by simply returning m.

15.3 Additional Exercises

Exercise 15.7. *Prove a lower bound of* $\Omega(\log n)$ *for* `deleteMin` *in comparison-based meldable priority queues. That is, prove that any meldable priority queue implementation which has a logarithmic* `meld` *cannot support* `deleteMin` *in faster than logarithmic time.*