Recitation 14

Priority Queues

14.1 Announcements

- PASLLab has been released, and is due next Friday (April 29 – or is that next next Friday?). PASLLab worth 175 points.
14.2 Leftist Heaps

**Task 14.1.** Identify the defining properties of a leftist heap.

A leftist heap is a binary tree given by

\[
\text{datatype tree = Leaf | Node of key \times tree \times tree}
\]

which satisfies

(a) the *heap property*, requiring that the key stored at each node is smaller\(^1\) than any descendant key, and

(b) the *leftist property*, requiring that for every \(\text{Node(_, L, R)}\), 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.,

\[
\begin{align*}
\text{rank}(\text{Leaf}) &= 0 \\
\text{rank}(\text{Node(_, L, R)}) &= 1 + \text{rank}(R)
\end{align*}
\]

**Task 14.2.** 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.
14.2.1 Building A Leftist Heap

Consider the following pseudo-SML code implementing leftist heaps.

```
Data Structure 14.3. 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 14.4. **Diagram the process of executing the code**

\[ \text{fromSeq } (3, 5, 2, 1, 4, 6, 7, 8) \]

Task 14.5. **What are the work and span of \((\text{fromSeq } S)\) in terms of \(|S| = n|?**

Notice that \texttt{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.
### Dynamic Median

**Task 14.6.** Design a data structure which supports the following operations:

<table>
<thead>
<tr>
<th>Operation</th>
<th>Work</th>
<th>Span</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>fromSeq S</code></td>
<td>$O(</td>
<td>S</td>
<td>)$</td>
</tr>
<tr>
<td><code>median M</code></td>
<td>$O(1)$</td>
<td>$O(1)$</td>
<td>Returns the median of all keys stored in $M$</td>
</tr>
<tr>
<td><code>insert (M, k)</code></td>
<td>$O(\log</td>
<td>M</td>
<td>)$</td>
</tr>
</tbody>
</table>

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 $m$. 
14.3 Additional Exercises

**Exercise 14.7.** 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.