## **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 \mid Node of key \times tree \times tree
```

which satisfies

- (a) the *heap property*, requiring that the key stored at each node is smaller<sup>1</sup> than any descendent key, and
- (b) the *leftist property*, requiring that for every  $Node(\underline{\ }, L, R)$ , we have  $rank(L) \geq rank(R)$ . We define the *rank* of a heap to be the number of nodes in its right spine, i.e.,

$$\begin{aligned} & \operatorname{rank}(\operatorname{Leaf}) = 0 \\ & \operatorname{rank}(\operatorname{Node}(\_, L, R)) = 1 + \operatorname{rank}(R) \end{aligned}$$

**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  $\log_2(n+1)$ .

<sup>&</sup>lt;sup>1</sup>We assume a min-heap. In a max-heap, each key is larger than its descendents.

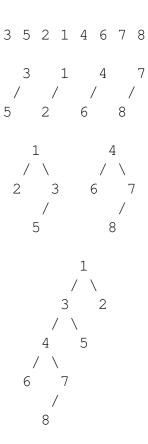
#### 15.2.1 Building A Leftist Heap

Consider the following pseudo-SML code implementing leftist heaps.

```
Data Structure 15.3. Leftist Heap
  1 datatype PQ = Leaf \mid Node \ of \ int \times key \times PQ \times PQ
 2
 3 fun rank Q =
 4
      case Q of
         Leaf \Rightarrow 0
 5
       | Node (r, \_, \_, \_) \Rightarrow r
 8 fun makeLeftistNode (k, A, B) =
 9
       if rank A < rank B
10
       then Node (1 + rank A, k, B, A)
11
       else Node (1 + rank B, k, A, B)
12
13 fun meld (A,B) =
14
       case (A, B) of
15
         (\_, Leaf) \Rightarrow A
       | (Leaf, \_) \Rightarrow B
16
17
       | (Node (\_, k_a, L_a, R_a), Node (\_, k_b, L_b, R_b)) \Rightarrow
18
            if k_a < k_b
19
            then makeLeftistNode (k_a, L_a, meld (R_a, B))
20
            else makeLeftistNode (k_b, L_b, meld (A, R_b))
21
22 fun singleton k = Node (1, k, Leaf, Leaf)
23
24 fun insert (Q, k) = meld(Q, singleton k)
25
26 fun from Seq S = Seq. reduce meld Leaf (Seq. map singleton S)
27
28 fun deleteMin Q =
29
      case Q of
30
         Leaf \Rightarrow (NONE, Q)
31
       | Node (\_, k, L, R) \Rightarrow (SOME \ k, meld \ (L, R))
```

Task 15.4. Diagram the process of executing the code

fromSeq 
$$(3, 5, 2, 1, 4, 6, 7, 8)$$



**Task 15.5.** What are the work and span of  $(fromSeq\ S)$  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  $\operatorname{meld}(A,B)$  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 me-
			dian 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  $k \ge m$ . 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.