Recitation 13

Priority Queues and Hashing

13.1 Announcements

- PASLLab is due this **Friday, May 5**.
- The final exam is on **Friday, May 13**.
- A review session for the final is upcoming. Stay tuned!
- A practice final and its solutions will be released soon on the course website.

13.2 Leftist Heaps

Task 13.1. *Identify the defining properties of a leftist heap.*

A leftist heap is a binary tree given by

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 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.,

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

Task 13.2. *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.

13.2.1 Building A Leftist Heap

Consider the following pseudo-SML code implementing leftist heaps.

Data Structure 13.3. *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 13.4. *Diagram the process of executing the code*

`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 13.5. *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 `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.

13.3 Removing Duplicates

Removing duplicates is a crucial substep of many interesting algorithms. For example, in BFS, consider the step where we construct a new frontier. One viable method would to be to generate the sequence of all out-neighbors, and then remove duplicates:

$$F' = \texttt{removeDuplicates}\ \left\langle v : u \in F, v \in N_G^+(u) \right\rangle$$

So, how fast is it to remove duplicates? Can we do it in parallel?

13.3.1 Sequential

Before we think about parallelism, we should acquaint ourselves with a good sequential algorithm solving the same problem. This way, we know what to shoot for in terms of work bounds, since we want our parallel algorithm to be asymptotically work-efficient.

Task 13.6. *Describe a sequential algorithm which performs expected $O(n)$ work to remove duplicates from a sequence of length n. Also argue that $\Omega(n)$ work is necessary in order to solve this problem, and conclude that your algorithm is asymptotically optimal.*

Hint: try hashing elements one at a time.

We can iterate left-to-right across the sequence, maintaining a set of all elements seen so far. At each element, we check to see if it's present in the set. If it is, we ignore it. If it isn't, we insert it into the set and also write it to the output. Using a hash set, we can check for membership and do insertions both in expected constant time. So, this algorithm has expected $O(n)$ work.

Removing duplicates requires at least $\Omega(n)$ work because we have to inspect every element. Here's a sketch of a proof by contradiction: suppose there was an algorithm for removing duplicates which used only $o(n)$ work. Then there must be at least one element of the input which the algorithm did not inspect. If the algorithm would blindly include this element in the output, then we can adversarily choose that element to be a duplicate. If instead the algorithm blindly excludes the element from the output, then we can adversarily choose the element to be distinct. Thus clearly the given algorithm does not properly remove duplicates.

Since $\Omega(n)$ work is necessary and our algorithm has $O(n)$ work, we know it is asymptotically optimal in terms of runtime.

13.3.2 Parallel

Task 13.7. *Implement a function*

```
val removeDuplicates : (α × int → int) → α Seq.t → α Seq.t
```

where (`removeDuplicates h S`) *retuns a sequence of all unique elements of* S*, given that* $h(e, m)$ *hashes the element* e *to a uniform random integer in the range* $[0, m)$ *(thus the probability of collision for any two distinct elements is* $1/m$*).*

Hint: as a first attempt, try simultaneously hashing as many elements as possible all at the same time. What do you do when elements collide?

We can use `inject` to simultaneously insert as many keys as possible into an initially empty hash table T of some size $m > n$ (we'll decide on a value for m later). Specifically, for every $0 \leq i < |S|$, we attempt to insert the pair $(i, S[i])$ at the location $T[h(S[i], m)]$.

Every pair $(i, S[i])$ then compares itself against the value $(j, S[j])$ stored at $T[h(S[i], m)]$. If $i = j$, then $S[i]$ is "accepted," and will be included in the output. Otherwise, if $S[i] \neq S[j]$, then $S[i]$ is passed on to be retried in the next round. We continue retrying until no elements remain.

Why is this algorithm correct? Consider some key k; we never discard k until k is accepted, so we only need to argue that we never accept the same key twice. Consider some round and two indices $i \neq j$ such that $S[i] = S[j]$. If $S[i]$ is accepted then $S[j]$ won't be (since $i \neq j$), and furthermore it won't be retried on the next round. Therefore $S[j]$ will never be accepted.

How many elements are retried each round? Consider:

$$\begin{aligned}
\textbf{Pr}\left[S[i] \text{ is retried}\right] &= \textbf{Pr}\left[\exists j. S[i] \neq S[j] \wedge h(S[i], m) = h(S[j], m)\right] \\
&\leq \sum_{S[i] \neq S[j]} \textbf{Pr}\left[h(S[i], m) = h(S[j], m)\right] \\
&\leq \frac{|S|}{m}
\end{aligned}$$

If we chose $m = 3|S|/2$, then $|S|/m = 2/3$, and by linearity of expectation, we have that the number of retried elements is at most $2|S|/3$ in expectation.

Since we need $O(|S|)$ work and $O(\log |S|)$ span on each round, we expect a logarithmic number of rounds with a geometrically decreasing input. We've seen such recurrences before; they solve to expected linear work and log-squared span.

Algorithm 13.8. *Removing duplicates with hashing.*

```
fun removeDuplicates S =
  if |S| = 0 then ⟨⟩ else
  let
    val E = Seq.enum S

    val m = $\frac{3|S|}{2}$
    val base = $\langle NONE : 0 \le i < m \rangle$
    val updates = $\langle (h(k,m),\ SOME(i,k)) : (i,k) \in E \rangle$
    val T = Seq.inject (base, updates)

    fun accept (i,k) = $\big(T[h(k,m)] = SOME\ (i,k)\big)$
    val A = $\langle k : (i,k) \in E \mid accept(i,k) \rangle$

    fun retry k = let val SOME(_, k') = T[h(k,m)] in $k \ne k'$ end
  in
    Seq.append (A, removeDuplicates $\langle k \in S \mid retry(k) \rangle$)
  end
```

13.4 Additional Exercises

Exercise 13.9.

Task 13.10. *Design a data structure which supports the following operations:*

	Work	*Span*	*Description*
`fromSeq` S	$O(\lvert S \rvert)$	$O(\log^2 \lvert S \rvert)$	*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)	$O(\log \lvert M \rvert)$	$O(\log \lvert M \rvert)$	*Inserts k into M*

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

Exercise 13.11. *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.*