Assignment 1

Sample Solutions

CSCI 3110 — Summer 2014

Question 1

A worst-case example. Consider an instance consisting of \( n \) men \( m_0, m_1, \ldots, m_{n-1} \) and \( n \) women \( w_0, w_1, \ldots, w_{n-1} \). The preference list of \( m_i \), for \( 0 \leq i \leq n - 2 \), is \( \langle w_i, w_{i+1}, \ldots, w_{n-2}, w_0, w_1, \ldots, w_{i-1}, w_{n-1} \rangle \). Man \( m_{n-1} \) has the same preference list as \( m_0 \). For \( 0 \leq i \leq n - 2 \), the preference list of woman \( w_i \) is \( \langle m_{i+1}, m_{i+2}, \ldots, m_{n-1}, m_0, m_1, \ldots, m_i \rangle \). The preference list of woman \( w_n \) can be chosen arbitrarily.

We divide the proposals into \( n \) rounds. Each of the first \( n - 1 \) rounds consists of \( n \) proposals, one per man. The final round consists of a single proposal. Consider the first round. For \( 0 \leq i \leq n - 2 \), man \( m_i \) proposes to woman \( w_i \) and gets married to her. Then \( m_{n-1} \) proposes to \( w_0 \). Since \( w_0 \) prefers \( m_{n-1} \) over \( m_0 \), \( w_0 \) divorces \( m_0 \) and marries \( m_{n-1} \). Thus, after the first round, \( m_0 \) is unmarried and every other man is married to the first woman on his list. Also, \( m_0 \) has proposed to one woman so far. In general, we claim that after the \( k \)th round, where \( 1 \leq k \leq n - 1 \), \( m_0 \) is unmarried, every other man \( m_i \) is married to the \( k \)th woman on his list, \( w_{(i+k-1) \mod (n-1)} \), and \( m_0 \) has proposed to \( k \) women so far. Indeed, we just observed that this is true for \( k = 1 \). For \( k > 1 \), assume the claim holds for \( k - 1 \). Then, for \( 0 \leq i \leq n - 2 \), \( m_i \) proposes to the \( k \)th woman on his list, \( w_{(i+k-1) \mod (n-1)} \). By the induction hypothesis, \( w_{(i+k-1) \mod (n-1)} \) is married to \( m_{i+1} \) at the end of the \( (k - 1) \)st round. Every woman other than \( w_i \neq w_{(i+k-1) \mod (n-1)} \) prefers \( m_i \) over \( m_{i+1} \), so \( w_{(i+k-1) \mod (n-1)} \) divorces \( m_{i+1} \) and marries \( m_i \), thereby making \( m_{i+1} \) the next unmarried man to propose in this round. Once we reach the situation when \( w_{n-1} \) is unmarried, \( w_{n-1} \) proposed to \( m_0 \)’s current partner \( w_{k-1} \). Since every woman other than \( w_{n-1} \) prefers \( w_{n-1} \) over \( w_0 \), \( w_{k-1} \) divorces \( m_0 \) and gets married to \( m_{n-1} \), so the invariant holds at the end of the \( k \)th round.

After \( n - 1 \) rounds, \( m_0 \) can no longer avoid proposing to \( w_{n-1} \) because she’s the last woman on his list. \( w_{n-1} \), being unmarried, accepts the proposal. Since \( m_0 \) was the only unmarried man before this proposal, all men are married after \( m_0 \) and \( w_{n-1} \) get married and the algorithm terminates. Since there are \( n - 1 \) rounds with \( n \) proposals each and a single final round with a single proposal, the total number of proposals is \( (n - 1)n + 1 = n^2 - n + 1 \).

\( n^2 - n + 1 \) is worst possible. Consider an arbitrary input and a run of the algorithm on this input. Let \( w_i \) be the last woman to get married for the first time, and assume this happens at time \( t \). Since a woman, once married, stays married forever, all women, and thus all men, are married after time \( t \).
Since the algorithm terminates once all men are married, this means that there are exactly \( t \) proposals. By time \( t \), every woman other than \( w_1 \) can have received at most \( n \) proposals, while, by definition, the proposal \( w_1 \) receives at time \( t \) is the first proposal she receives. This shows that the number of proposals is \( t \leq (n - 1)n + 1 = n^2 - n + 1 \).

**Question 2**

The priority queue is implemented as an array \( Q \) of size \( u \). Each array entry \( Q[i] \) stores a pointer to the head of a doubly-linked list, which contains all elements with priority \( i \). Using this data structure, we can implement the different priority queue operations as follows:

- **Insert(\( Q, x, p \))**: We add element \( x \) to the head of list \( Q[i] \). This clearly takes constant time, as it requires a single index access into \( Q \), followed by an insertion into a doubly-linked list.

- **Delete(\( Q, x \))**: Assuming we have a pointer to the priority queue node (that is, the doubly-linked list node) storing element \( x \), this is a simple matter of removing this node from its doubly-linked list, which takes constant time.

- **DecreaseKey(\( Q, x, p \))**: First we compare \( p \) to the current priority \( p' \) of \( x \). If \( p' \leq p \), there is nothing to do. Otherwise, we first delete \( x \) from \( Q \) and then re-insert it with priority \( p \). Since insertions and deletions take constant time, so does this operation. (Alternately, we can directly move \( x \)'s node from the list \( Q[p'] \) to the list \( Q[p] \). This is faster in practice.)

- **DeleteMin**: Deleting and returning the minimum element, once we have found it, takes constant time because it is the same as a Delete operation. Thus, the hard part is to find an element with minimum priority. To do this, we scan \( Q \), inspecting slots \( Q[1], Q[2], \ldots \), until we find the first slot \( Q[i] \) that is not empty, that is, does not point to the empty list. Clearly, \( i \) is the minimum priority in \( Q \), so deleting and returning any of the elements in list \( Q[i] \) is correct. The cost of inspecting slot \( Q[j] \), for \( 1 \leq j \leq i \), is constant because it is a simple test whether \( Q[j] \) points to the empty list or not. Thus, the total cost is proportional to \( 1 + i \).