CS 522: Data Structures and Algorithms II
Homework 1

Dustin Ingram

January 29, 2013

1. **Solution:** If both INCREMENT and DECREMENT operations were included in the k-bit counter, an amortized analysis of the cost of n operations would cost as much as $\Theta(nk)$ time because we would no longer be able to consider each operation as a consecutive INCREMENT, but rather as any combination of INCREMENTS and DECREMENTS. Thus, in a worse-case scenario, it would be possible to alternate n times between two operations which cost $O(k)$ each, resulting in a total cost of $\Theta(nk)$.

2. **Solution:** To show that the amortized cost of TABLE-DELETE under this strategy is bounded above by a constant, we will consider two cases. We will use the potential function:

$$\Phi(T) = |2 \cdot T.num - T.size|$$

The first case is the one in which the table does not contract, and thus $num_i = num_{i-1} - 1$, $size_i = size_{i-1}$, and $c_i = 1$:

$$\begin{aligned}
\hat{c_i} &= c_i + \Phi_i - \Phi_{i-1} \\
\hat{c_i} &= 1 + |2 \cdot num_i - size_i| - |2 \cdot num_{i-1} - size_{i-1}| \\
\hat{c_i} &= 1 + |2 \cdot (num_{i-1} - 1) - size_{i-1}| - |2 \cdot num_{i-1} - size_{i-1}| \\
\hat{c_i} &= 1 + |-2| \\
\hat{c_i} &= 3
\end{aligned}$$

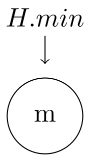
The second case is the one in which the table does contract, and thus $size_i = \frac{2}{3} size_{i-1}$, $num_{i-1} = \frac{1}{3} size_{i-1}$, and $c_i = num_i + 1$:

$$
\begin{aligned}
\hat{c_i} &= c_i + \Phi_i - \Phi_{i-1} \\
\hat{c_i} &= (num_i + 1) + |2 \cdot num_i - size_i| - |2 \cdot num_{i-1} - size_{i-1}| \\
\hat{c_i} &= ((num_{i-1} - 1) + 1) + |2 \cdot (num_{i-1} - 1) - \frac{2}{3} size_{i-1}| - |2 \cdot num_{i-1} - size_{i-1}| \\
\hat{c_i} &= (num_{i-1}) + | - 2 + \frac{1}{3} size_{i-1}| \\
\hat{c_i} &= 2
\end{aligned}
$$

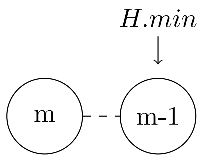
Thus we see that the amortized cost of TABLE-DELETE is at most 3 and is thus bounded.

3. **Solution:** One can use the following sequence to produce a Fibonacci heap that is only a linear chain of n nodes. Since we start at the base case (where the heap is empty) and create a chain of a single node, followed by a chain of two nodes from a chain of one node, and then a chain of three nodes from a chain of two nodes, we can repeat the process n times to produce a chain of total length n.

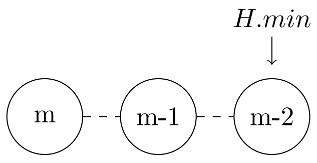
   FIB-HEAP-INSERT(H, m):



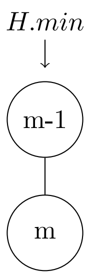
   FIB-HEAP-INSERT(H, $m-1$):



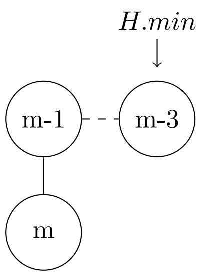
   FIB-HEAP-INSERT(H, $m-2$):



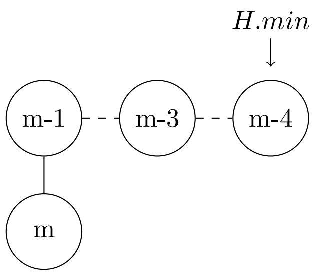
   FIB-HEAP-EXTRACT-MIN(H):



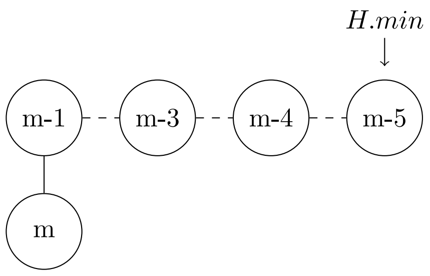
FIB-HEAP-INSERT(H, $m-3$):



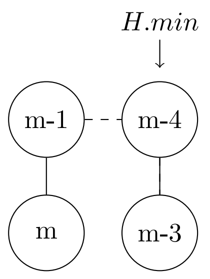
FIB-HEAP-INSERT(H, $m-4$):



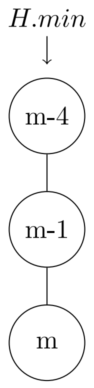
FIB-HEAP-INSERT(H, $m-5$):



FIB-HEAP-EXTRACT-MIN(H):



Fib-Heap-Delete(H, $m-3$):



4. **Solution:**
   a. The issue with the professor's claim is that while we can in fact add x's child list to the root list of H in constant time, this is not "paying forward" to the total amortized cost that it will take to do the eventual Cascading-Cuts that are necessary.

   b. The upper bound would be as bad as:

$$O(x.degree \cdot c) = O(\lg n)$$

   c. The potential of H might be:

$$\phi = x.degree \cdot c + \frac{t(H)}{m(H)}$$

   d. Thus, the amortized time for Pisano-Delete is no better than for Fib-Heap-Delete.

5. **Solution:** We will use the stack, which we will call `stack`, to hold the actual element being inserted, and the "huge" array `dict` to hold references to where the element exists in `stack`. We will also need a "mirrored"

stack, `mirrored`, which will hold the actual values of the elements. Since all actions are direct-access and of a constant number for each function, all functions are $O(1)$ running time.

Initialize: At any given time, `stack` will contain exactly the number of elements that are currently stored in `dict`. Thus, at initialization, the size of `stack` is 0, and requires no initialization. Since `dict` may contain garbage, and will only be overwritten, it too requires no initialization.

Insert: When inserting a new element, we add the element to the end of `stack` as such:

$$\texttt{stack}[\texttt{stack}.length] = \texttt{element}.key$$

We also add the value of the element to the mirrored stack:

$$\texttt{mirror}[\texttt{stack}.length] = \texttt{element}.value$$

Since this increases the length of `stack` by 1, the index of `element` is now at `stack`.*length* $- 1$. We add this value to `dict`, overwriting whatever might be there.

$$\texttt{dict}[\texttt{element}.key] = \texttt{stack}.length - 1$$

Search: When searching for an `element`, we must satisfy two conditions to know that we will return the correct element in constant time. First, we must make sure that we have previously inserted this element and that the reference data in `dict` is not garbage:

$$0 \leq \texttt{dict}[key] \leq \texttt{stack}.length - 1$$

This alone, however, does not prove that Search is returning the correct element. We must also compare the results. The result is only valid if:

$$\texttt{stack}[\texttt{dict}[key]] = \texttt{key}$$

If this is true, then we can return the value from the mirrored stack:

$$\texttt{mirrored}[\texttt{stack}[\texttt{dict}[key]]]$$

Delete: Deletion can be done simply by turning the key reference in `dict` back into garbage, e.g. to a value that will never be a valid key and will fail the first test:

$$\texttt{dict}[key] = -1$$

This will leave a gap in `stack` (and in `mirror`). If it is desirable to maintain the lengths of these stacks to match the actual number of keys, we could maintain a separate stack which would keep track of deleted keys and re-use them when new elements are inserted.

6. **Solution:**

   Assuming uniform hashing, given two elements k and l and a table of size m the following probability is true for two elements, k and l, where $k \neq l$ and $h(k) = h(l)$:

   $$P_{individual-collision} = 1/m$$

   Thus the probability of not having a collision between k and l is:

   $$P_{no-individual-collision} = 1 - P_{individual-collision}$$

   Therefore, the probability that there will be no collisions for a single element with any of the $n-1$ other elements is:

   $$P_{never-collisions} = {P_{no-individual-collision}}^{(n-1)}$$

   Thus the probability that there will be at least one collision for a single element with any of the $n-1$ other elements is:

   $$P_{collision} = 1 - P_{never-collisions}$$

   Therefore the expected number of collisions after inserting all n nodes is:

   $$n \cdot P_{collision} = n\left(1 - \left(1 - \frac{1}{m}\right)^{(n-1)}\right)$$

7. **Solution:** For each set in $\mathscr{H}$, the probability of there being a collision is $P_{collision}$ as was found in the previous question. The set U represents the number of available elements while the set B represents the available locations without collisions. Thus the probability of there being a collision for any of the elements is:

   $$P_{collision} \cdot (|B| - |U|)$$

   I'm not sure how to continue from there.