CS 670 Spring 2015 - Solutions to HW 4

Problem 17.1-3

Let $c(i)$ denote the cost of the i^{th} operation.

$$\sum_{i=1}^{n} c(i) = \sum_{j=0}^{\lfloor \log_2(n) \rfloor} 2^j + n - \lfloor \log_2(n) \rfloor$$

The first term on the right counts the cost of the 2^{th} power operations. The term $n - \lfloor \log_2(n) \rfloor$ counts the cost of the rest of the operations.

$$\Rightarrow \sum_{i=1}^{n} c(i) \leq \frac{2^{\lfloor \log_2(n) \rfloor + 1} - 1}{2 - 1} + n - \lfloor \log_2(n) \rfloor \leq 3n$$

The amortized cost is thus bounded by

$$\frac{\sum_{i=1}^{n} c(i)}{n} \leq 3$$

Problem 17.2-1

Let the amortized cost of $PUSH$ be 3, and that of POP and $MULTIPOP$ be 1 and $COPY$ be 0. The idea is that whenever we made push we pay 1 unit for pushing, and 1 unit for any subsequent popping. We also pay another 1 unit for any copying that would be done after k operations. Similarly, POP and $MULTIPOP$ need to pay 1 unit for copying after k operations. Hence, the order for n operations is $O(n)$.

Problem 17.2-2

Let the amortized cost of each operation be 3. Each operation pays 1 unit for itself and 2 units for the next op which is a power of 2.

When i is power of 2, say $i = 2^r$, the extra units earned from the operations 2^{r-1} to 2^r is $2 * (2^r - 2^{r-1}) = 2^r$, which can be used to pay for the operation i.

For example:

1st op pays 1 unit for itself and 2 units for the 2nd op. So the actual costs of ops 1 and 2 are covered

2nd and 3rd op pay 1 unit each for themselves and 2 units each for the 4th op. So the actual costs of ops 3 and 4 are covered

4th, 5th, 6th and 7th op pay 1 unit each for themselves and 2 units each for the 8th op. So the actual cost of ops 5, 6, 7 and 8 are covered.

Problem 19-3

a. The FIB-HEAP-CHANGE-KEY(H, x, k) operation in the case in which

- $k < key[x]$ will invoke the FIB-HEAP-DECREASE-KEY(H, x, k) operation which has an amortized cost of $O(1)$.
- $k = key[x]$ will just return after the comparison and hence has an amortized cost of $O(1)$ (due to the cost of comparison).
- $k > key[x]$ will invoke CUT$(H, y, p[y])$ for each child of x (i.e., $p[y] = x$) and then increase the key of x to k ($key[x] \leftarrow k$) and then invoke CUT$(H, x, p[x])$. Since the maximum degree of node in an n-node Fibonacci heap is $O(\log n)$ we have the amortized cost of the operation in this case is $O(\log n)$. Alternatively, we could $key[x] \leftarrow k$ and push x down the tree until there is no heap property violation by swapping x with a child with the minimum key at each level. The amortized cost is $O(\log n)$ if we maintain for each node a linked list for its children with a pointer to the child with a minimum key. This would incur an $O(1)$ extra cost for all other operations, but won't change the big-O cost. $O(\log n)$ and since the height of a tree is $O(\log n)$ the total cost is $O(\log^2 n)$.

b. Deleting a node has an amortized cost of $O(\log n)$. If we were to delete r particular nodes the amortized cost would be $O(r \log n)$ but since the question says we could delete arbitrary nodes, the time cost could be less. What we can do is to cut r leaf nodes so that each one of them becomes a singleton tree, and then we remove each one of them. In order to facilitate this kind of operation we maintain for each tree a linked list of the leaf nodes and a pointer from the root to a leaf node. In this process of pruning there may be cascade-cuts triggered as a result of enforcing the cut policy. The amortized cost of pruning r nodes is $O(r)$, nevertheless, since it requires r actual cuts.