CSE548, AMS542: Analysis of Algorithms, Fall 2012 **Date:** Oct 30

Homework #3

(Due: Nov 13)

Task 1. [15 Points] You have to do the best with what God gave you (Mrs. Gump in "Forrest Gump")

Given a mere binary heap you will have to augment it to support INSERT operations in constant amortized time.

(*a*) [**5 Points**] Consider a binary min-heap containing exactly $n = 2^{h+1} - 1$ elements for some integer $h \geq 0$. Show that for any integer $k \in [0, h+1]$ a batch of 2^k new elements can be inserted into this heap in $\mathcal{O}\left(2^k + h^2\right)$ worst-case time.

(*b*) [**10 Points**] Part (*a*) implies that a batch of 2^k new elements can be inserted into a heap (not necessarily full) of height h in $\mathcal{O}\left(2^k\right)$ worst-case time provided $2^k = \Theta\left(h^2\right)$. Use this observation to prove that a binary heap can be augmented to support INSERT operations in $\mathcal{O}(1)$ amortized time without changing the $\mathcal{O}(\log n)$ worst-case time bound for EXTRACT-MIN, DELETE and DECREASE-KEY.

Task 2. [30 Points] The Road Not Taken[1] (Robert Frost)

(*a*) [**10 Points**] We have shown in the class that Fibonacci heaps support DELETE and EXTRACT-MIN operations in $\mathcal{O}(\log n)$ amortized time each, and every other operation[2] in $\mathcal{O}(1)$ amortized time, where n is the number of items currently in the heap. Present an alternate analysis based on the potential method to show that the same Fibonacci heap implementation supports INSERT operations in $\mathcal{O}(\log n)$ amortized time and all other operations[3] (except UNION) in $\mathcal{O}(1)$ amortized time each. What bound do you get for INSERT when you make the amortized cost of UNION also $\mathcal{O}(1)$?

(*b*) [**5 Points**] Explain how to implement INCREASE-KEY($\mathcal{H}$, x, k) operations[4] efficiently in a Fibonacci Heap without changing the amortized time complexities of other operations. Argue that your bound is the best possible.

(*c*) [**15 Points**] Suppose we modify the Fibonacci Heap so that a node is cut from its parent only after losing its k^{th} child for some given integer $k \geq 2$. Analyze amortized time complexities of all heap operations on this modified version assuming $k = 3$ (recall that the version we analyzed in the class used $k = 2$). Explain how the performance of DECREASE-KEY and EXTRACT-MIN will change if k is increased.

[1] Two roads diverged in a wood, and I–
I took the one less traveled by,
And that has made all the difference.

[2] MAKE-HEAP, INSERT, UNION, MINIMUM and DECREASE-KEY

[3] MAKE-HEAP, MINIMUM, DELETE, EXTRACT-MIN and DECREASE-KEY

[4] INCREASE-KEY($\mathcal{H}$, x, k): change the key of element x of heap $\mathcal{H}$ to k assuming that $k \geq$ the current key of x

Task 3. [35 Points] Sums on a ǝǝɹ⊥[5]

Suppose you are given a full, rooted binary tree with weights on the edges, and you are asked to preprocess the tree in order to answer queries of the form: what is the 'sum' of the edge weights from a given leaf u to one of its given ancestors v in the tree? Here 'sum' is an arbitrary associative operation, e.g., regular addition, maximum/minimum, bitwise AND/OR, etc.

In this task we are interested in optimizing the time and space used by the preprocessing algorithm as well as the time required to answer each query. The query time is given by the maximum number probes into the data structure (i.e., number of times you read 'sums' or 'partial sums' from the data structure) to answer a single query.

Suppose n is the number of nodes in the tree. Then $n = 2^{h+1} - 1$ for some integer $h \geq 0$, and the tree has exactly $2^h = \frac{1}{2}(n+1)$ leaves.

(*a*) [**5 Points**] If 'sum' has an inverse (e.g., regular addition has an inverse operation called subtraction), show that the tree can be preprocessed in $\Theta(n)$ time and space to answer each query using at most 2 probes.

For parts (*b*) – (*e*) assume that 'sum' does not have an inverse operation (e.g., maximum/minimum and bitwise AND/OR).

(*b*) [**5 Points**] Show how to preprocess the tree in $\Theta(n \log n)$ time and space so that each query can be answerd using a single probe.

(*c*) [**15 Points**] Use the result from part (*b*) to show how to preprocess the tree in $\Theta(n \log^* n + n)$ time and space so that each query can be answered using at most 2 probes.

(*d*) [**5 Points**] Show that if you know how to preprprocess the tree in $\Theta\left(n \log^{[[*(k-1)]]} n + (k-1)n\right)$ time and space[6] to achieve a query time of k for some integer $k \in (0, \alpha(n))$, you can improve the preprocessing time and space to $\Theta\left(n \log^{[[*(k)]]} n + kn\right)$ at the expense of increasing the query time to $k+1$.

(*e*) [**5 Points**] Use your result from part (*d*) to show that the tree can be preprocessed in $\Theta(n\alpha(n))$ time and space to answer each query using at most $\alpha(n)$ probes.

[5]A *tree* in computer science is a *ǝǝɹʇ* in nature with the root at the top and the leaves at the bottom

[6]$\log^{[[*(k)]]} n = \log^{\overbrace{*\cdots*}^{k}} n$