Assignment 1 - CSC265

David Knott
Student #999817685

September 30, 2013

1. a)

   b) Note that a heap of height n contains at most $2^n - 1$ nodes and at least 2^{n-1} nodes. If the heap contains the maximum amount of nodes, or is "complete", then it's easy to turn into a pennant. If we take the root node drop it's left subtree we get a pennant of height n (the root node + it's right subtree) and a heap of height $n - 1$ (the root's left subtree). Since the left subtree is a complete heap, we can simply repeat this process to obtain a pennant forest with n pennants, where pennant p_n is of height n. This process only takes $\log(n)$ time, since the most we're doing is traversing down the left side of the tree and copying pointers as we go.

   For an incomplete heap we use a recursive algorithm that, given a node in the heap, outputs a pennant forest made of the heap of height n rooted at the input node. Note that for any node in a heap only one of the four conditions appy:

   i. The node is the root of a complete heap.
   ii. The node's left and right subtrees are complete heaps, but the left subtree is higher than the right by one level.
   iii. The node's right subtree is a complete heap, but the left subtree is higher than the right by one level and is incomplete.
   iv. The node's left subtree is a complete heap, but the right subtree is incomplete. The two subtrees have the same height.

   So for any node in the tree, our algorithm will do one the following for each of the conditions described above:

   i. The function simply turns the node and it's children into a pennant forest using the process described in the first paragraph.
   ii. The function calls itself on the left subtree, yielding the pennant forest P. Since P's largest pennant is of size $n - 2$ and has only one pennant per height, the function appends the pennant made from the input node and the right subtree, which will be of height $n - 2$. The function then returns this forest after ordering the last two pennants.

iii. The function does the same as above. The largest pennant in the forest returned is less than or equal to $n-2$, and there is strictly one or less pennants of size $n-2$. This means the function can append the pennant made out of the input node and it's right subtree while still maintaining the pennant forest property.

iv. The function calls itself on the right subtree, which will yield a pennant forest who's maximum pennant is at most height $n-2$. The function adds the pennant made from the input node and it's left child, which will be of height $n-1$.

This function also runs in $\log(n)$ time, since for each level of the tree the function is called at most once, since the function calls itself on only one subtree.

c)

d) Take the root of the pennant (a) and the root of the pennant's binary tree (b) as the roots of two new pennants. Take off the left subtree of b and attach it to a. Since $a \geq b \geq$ any child of b, the two pennants will satisfy the priority property. Since we only do a constant amount of pointer swapping, this algorithm takes constant time.

e) Suppose these two pennants are called X and Y. The root of X will be called a and the child of a will be called b. The root of Y will be called c and the child of c will be called d. Assume that $a \geq c$ (if this isn't true simply swap X with Y). There are three possibilities:

i. $a \geq b \geq c \geq d$

ii. $a \geq c \geq b \geq d$

iii. $a \geq c \geq d \geq b$

And for any of these conditions, the algorithm will behave as such:

i. The new pendant will take a as it's root and b as it's child. The right subtree of b will be the one rooted at d. The left subtree will be the one rooted at b, but with it's root replaced with c. In $\log(n)$ time, c will bubble down the heap it has become the root of. This will yield a pennant of size n in logarithmic time.

ii. The new pendant will take a as it's root and c as it's child. The right subtree of c will be the one rooted at d. The left subtree will be the one rooted at b. This will produce a pennant of size n in constant time.

iii. Works the same as above.

The worst case running time of this algorithm is logarithmic in the height of the input pennants.

f)

2. a) A d-ary heap would be represented in an array much in the same way a binary heap is, only instead of 2^l array entries for the level l, one would need d^l entries. When it comes to traversal, the node at array entry n's rth child would be referenced with the function $dn+r$, where r is in the range $[1, d]$. Traversing to the parent node from n is done with the formula $\lfloor n/d \rfloor$.

   b) A d-ary heap with n elements has a height of $\lceil \log_d(n) \rceil$

   c) Just like in a binary heap, ExtractMax removes and returns the root node of the heap, replacing it with the last element in the last layer. This new element then bubbles down the heap by recursively comparing itself with it's children then swapping itself with the maximum element that's greater than it (if there's no such element then the algorithm ends.) Because this bubbling takes time proportional to the height of the tree and at each level must do d comparisons (finding the maximum child takes $d-1$ comparisons, and comparing the max with the current parent takes 1 comparison), the amount of comparisons done in the worst case will be about $d\lceil \log_d(n) \rceil$, which is in $O(\log n)$ time.

   d) Insert will put the element in the next available spot on the heap, then bubble it up recursively. The rules for this bubble up procedure are: if the parent is smaller than the inserted element their positions are swapped and the algorithm continues, if the parent is larger then the heap property has been restored and the algorithm ends. The worst case in this scenario would be the inserted element has the largest value out of any of the elements in the heap, so we must preform $\lceil \log_d(n) \rceil$ swaps with 1 comparison per swap. Hence this algorithm runs in $O(\log n)$ time.

   e) IncreaseKey works much the same as the above algorithm. We simply change the element's value and bubble it up through the heap, taking at most $\lceil \log_d(n) \rceil$ comparisons, making it in $O(\log n)$ time.

3. a) For this problem we convert each list into a binomial min-heap. This is easy since we know the length of each list and hence the structure of it's corresponding binomial heap. This is because $n/k = \sum_k^{\lceil log_2 n/k \rceil} x_k 2^k$ for some bit vector x (the binary representation of n/k) and the set $S = \{k \in \mathbb{N} \mid x_k = 1\}$ is the sizes of the binomial trees we must use in the heap. So to construct the heap from a list L of size n/k we simply iterate over $k \in S$, popping 2^k elements from the list to produce a binomial tree of height k.

   Once we do this for all input lists, we have a series of binomial heaps $B_1, \ldots, B_k$. This takes $c\frac{n}{k}k = cn$ operations, where c is the constant satisfying $c\frac{n}{k} = f(\frac{n}{k})$ where $f(x)$ is the amount of operations it takes to binomialize a sorted list of length x. Merging two binomial heaps takes $O(\log(n))$ time, where n is the number of nodes in the largest heap.

   To merge all these heaps together, we do a tournament style system where we merge every odd numbered heap to the even numbered heap next to it. We repeat this around $\log(k)$ times to yield one binomial heap with n nodes. We then call ExtractMin on this heap n times to produce the sorted list.