22C:44 Homework 4 Solutions

1. The best case running time for the n insertion operations is $\Theta(n)$. This happens when each insertion takes $\Theta(1)$ time.

   The worst case running time for the n insertion operations is $\Theta(n \lg n)$. First note that the worst case running time is $O(n \lg n)$. This is because each insertion is on a heap with at most n elements and inserting into a heap with k elements takes $\Theta(\lg k)$ in the worst case. To see that this upper bound is tight, that is, in the worst case the running time is indeed $\Theta(n \lg n)$, consider a sequence of n numbers in increasing order. The $(k+1)$st number is inserted into a heap with k numbers and has to be moved all the way to the root of the heap. This takes $\Theta(\lg k)$ time. Summing this over all insertions we get

   $$\sum_{k=1}^{n-1} \Theta(\lg k) = \Theta(\sum_{k=1}^{n-1} \lg k = \Theta(\lg((n-1)!)) = \Theta(n \lg n).$$

   The last inequality follows from *Stirling's Formula* given as equation 2.11 on Page 35 on the textbook.

   We now separately calculate the time it takes to allocate memory using each of schemes described in the problem.

   (a) For every integer k, $1 \leq k < n$, after k insertions, H contains k elements and is full. For the $(k+1)$st insertion, memory allocation takes time $\Theta(k + (k+1)) = \Theta(k)$. Summing this over all insertions, the total time for memory allocation is

   $$\sum_{k=1}^{n-1} \Theta(k) = \Theta(\sum_{k=1}^{n-1} k) = \Theta(n^2).$$

   Therefore, the total amount of time, for insertion plus memory allocation is $\Theta(n^2)$ in the best as well as in the worst case.

   (b) Using the second scheme, memory allocation takes place when we have 2^k elements in the heap and the heap grows to size 2^{k+1}. This takes time $\Theta(2^k + 2^{k+1}) = \Theta(2^k)$. This takes place for all integers $k = 0$ through m where m is the largest integer satisfying $2^m < n$. In other words, this happens for all integers $k = 0, 1, \ldots, \lfloor \lg n \rfloor$. Summing $\Theta(2^k)$ over all possible k we obtain the total time for memory allocation as

   $$\sum_{k=0}^{\lfloor \lg n \rfloor} 2^k = \Theta(\sum_{k=0}^{\lg n} 2^k) = \Theta(2^{k+1} - 1) = \Theta(n).$$

2. (a) For each i, the "children" of node i are nodes $3i-1$, $3i$, $3i+1$. Using this we deduce that the "parent" of node i is node $\lfloor (i+1)/3 \rfloor$. So define children$(i) = \{3i-1, 3i, 3i+1\}$. The heap property can be stated as, for each i, $A[i] \geq A[j]$ is $j \in$ children(i).

   (b) The largest 3-ary tree with height h has all $(h+1)$ levels full and the smallest 3-ary tree with height h has the first h levels full and level $(h+1)$ containing 1 node. This observation implies that if a 3-ary tree with height h has n nodes,

   $$\frac{3^h - 1}{2} + 1 \leq n \leq \frac{3^{h+1} - 1}{2}.$$

   This can be rearranged to

   $$\frac{2n+1}{3} \leq 3^h \leq 2n - 1$$

and by taking the logarithm to the base 3 of all terms and simplifying we get

$$\log_3(2n+1) - 1 \le h \le \log_3(2n-1).$$

This tells that for $n = 1$, $h = 0$. This is of course, no surprise! For any $n > 1$, it is easy to see that $|\log_3(2n-1) - (\log_3(2n+1) - 1)| < 1$. This implies that there is a unique integer in the range $[\log_3(2n+1) - 1..\log_3(2n-1)]$. This implies that $h = \lfloor \log_3(2n-1) \rfloor$.

(c) The number of nodes n and the number of leaves ℓ in 3-ary heaps for all n, $1 \le n \le 10$ are shown in the table below.

n	ℓ
1	1
2	1
3	2
4	3
5	3
6	4
7	5
8	5
9	6
10	7

It is clear from the table that for every 3 nodes added, 2 node-additions cause an increase in the number of leaves. This implies that the number of leaves is roughly 2/3rds the number of nodes. From the table, we get the more precise formula $\lfloor (2n+1)/3 \rfloor$.

3. (a) Procedures `BUILD-HEAP` and `BUILD-HEAP'` do not create the same heap. The smallest example has 3 elements in it. Start with the heap $2, 3, 8$. Calling `BUILD-HEAP` on this will call `HEAPIFY` at 2 and the result is $8, 3, 2$. Calling `BUILD-HEAP'` on this will insert 2 first and then 3 into the heap. After inserting 3, we have the heap $3, 2$. If we insert 8 into this heap we get the heap $8, 2, 3$.

   (b) This is shown in Problem 1.

4. 8.1-1 is skipped (too easy!). The solution 8.1-2 is that `PARTITION` returns $\lfloor (p+r)/2 \rfloor$.

5.
```
BETTER-PARTITION(A, p, s){
        q ← p - 1; r ← p;
        for j ← p to s-1 do{
                if (A[j+1] == A[q+1]) then{
                        swap(A, r+1, j+1);
                        r++;
                }
                else if (A[j+1] < A[q+1]) then{
                        t ← A[j+1];
                        A[j+1] ← A[r+1];
                        A[r+1] ← A[q+1];
                        A[q+1] ← t;
                        q++; r++;
                }
        }
}
```