CMPSC 623 Problem Set 3
by Prof. Honggang Zhang

Out: October 5, 2006
Due: October 12, 2006, before class.

Problem 1. Exercise 6.1-2 on page 129.

Solution: Let h be the height of the heap. Since a heap is an almost-complete binary tree (complete at all levels except possibly the lowest), it has at most $2^{h+1} - 1$ elements (if it is complete) and at least $2^h - 1 + 1 = 2^h$ elements (if the lowest level has just 1 element and the other levels are complete). That is, we have

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

Thus, $h \leq \lg n < h + 1$. Since h is an integer, $h = \lfloor \lg n \rfloor$ (by definition of $\lfloor\ \rfloor$).

Exercise 6.1-6 on Page 130.

Solution: No.

Problem 2. Page 132, 6.2-6.

Solution: If you put a value at the root that is less than every value in the left and right subtrees, then MAX-HEAPIFY will be called recursively until a leaf is reached. To make the recursive calls traverse the longest path to a leaf, choose values that make MAX-HEAPIFY always recurse on the left child. It follows the left branch when the left child is larger than or equal to the right child, so putting 0 at the root and 1 at all the other nodes, for example, will accomplish that. With such values, MAX-HEAPIFY will be called h times (where h is the heap height = the longest path length from the root to a leaf), so its running time will be $\Theta(h)$ (since each call does $\Theta(1)$ work), which is $\Theta(\lg n)$. Since we have a case in which MAX-HEAPIFY's running time is $\Theta(\lg n)$, its worse-case running time is $\Omega(\lg n)$.

Problem 3. Page 136, 6.4-1.

Solution: Straightforward.

Problem 4. Page 154, 7.3-1.

Solution: We may be interested in the worse case performance, but in that case the randomization is irrelevant: it will not improve the worst case. What randomization can do is to make the chances of encountering a worst-case scenario small.

Problem 5. Page 162, problem 7-4.

Solution:

a).

QUICKSORT' does exactly what QUICKSORT does, hence it sorts correctly.

QUICKSORT' and QUICKSORT do the same partitioning, and then each calls itself with arguments A, p, q. QUICKSORT then calls itself again, with arguments $A, q+1, r$. QUICKSORT' instead set $p = q+1$ and re-executes itself. This executes the same operations as calling itself with $A, q+1, r$, because in both cases the first and third arguments (A, r) have the same values as before and p has the old value of $q+1$.

b).

The stack depth of QUICKSORT' will be $\Theta(n)$ on an n-element input array if there are $\Theta(n)$ recursive calls to QUICKSORT'. This happens if every call to $PARTITION(A, p, r)$ returns $q = r-1$. The sequence of recursive calls in this scenario is:

$QUICKSORT'(A, 1, n)$, $QUICKSORT'(A, 1, n-1)$, $QUICKSORT'(A, 1, n-2)$, $...QUICKSORT'(A, 1,$

You can construct a specific array that causes this behavior: The PATTITION shown in the book will behave this way, for example, on the array $< n, 1, 2, ..., n-1 >$, which it partitions into $< n-1, 1, 2, ..., n-2 >$ and $< n >$. Each time PARTITION is given an array with the largest element at the front and the rest of the elements sorted in creasing order, the high side of the resulting partition has just the largest element and the low side has the rest of the elements, again with the largest at the front and the rest in increasing order.

c).

The problem demonstrated by the scenario in (b) is that each invocation of QUICKSORT' calls QUICKSORT' again with almost the same reange. To avoid such behavior, we must change QUICKSORT' so that recursive call is on a smaller interval of the array. THe following variation of QUICKSORT' checks which of the two subarrays retuned from PARTITION is smaller and recurses on the smaller subarray, which is at most half the size of the current array. Sinc ehte array size is reduced by at least half on each recursive call, the number of recursive calls, and hence the stack depth, is $\Theta(\lg n)$ in the worst case. Note that this works no matter what PARTITION algorithms we use.

```
QUICKSORT''

while $p<r$

    do // Partition and sort the small subarray first

    $q\gets$ PARTITION(A,p,r)

    if $q\gets p+1 < r-q$

        then QUICKSORT''(A,p,q)

            $p\gets q+1

        else QUICKSORT''(A,q+1, r)
```

```
$r\gets q$
```

The expected running time is not affected, because exactly the same work is done as before: The same partitions are produced, and the same subarrays are sorted.