CMPSC 623 Problem Set 3
by Prof. Honggang Zhang

Out: October 2, 2007
Due: October 9, 2007

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 $\lfloor\ \rfloor$).

Problem 2. Exercise 6.1-7 on page 130. **Solution**: No.

Problem 3. Exercise 6.2-5 on page 132.

Problem 4. Exercise 6.3-2 on page 135.

Problem 5. Problem 7-4 on page 162. **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 $A, p, q-1$. $QUICKSORT$ 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 $q = r$. The sequence of recursive calls in this scenario is:

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

You can construct a specific array that causes this behavior. For example, if you always choose the first element as the pivot, 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. If you always choose the last element as the pivot, a completely sorted input array gives you this behavior.

c).

The problem demonstrated by the scenario in (b) is that each invocation of $QUICKSORT'$ calls $QUICKSORT'$ again with almost the same range. 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 returned from $PARTITION$ is smaller and recurses on the smaller subarray, which is at most half the size of the current array. Since the 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''(A, p, r)
 1.  while p < r
 2.    do // Partition and sort the small subarray first
 3.         q ← PARTITION(A, p, r)
 4.         if q − p < r − q
 5.              then QUICKSORT''(A, p, q)
 6.                   p ← q + 1
 7.              else QUICKSORT''(A, q + 1, r)
 8.                   r ← 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.

Problem 6. Exercise 8.1-3 on page 168.

Problem 7. Exercise 8.2-4 on page 170.

Problem 8. Exercise 8.3-4 on page 173.