ECE608 CHAPTER 6 PROBLEMS

1) 6.1-6
2) 6.1-7
3) 6.2-6
4) 6.3-3
5) 6.4-2
6) 6.5-8
7) 6-3

ECE 608 - Chapter 6 answers

(1) CLR 6.1-6

No, $\langle 23, 17, 14, 6, 13, 10, 1, 5, 7, 12\rangle$ is not a heap because the heap property does not hold between the 4th element and its second child, the 9th element (i.e., $6 < 7$).

(2) CLR 6.1-7

Let i represent the index of a node in a heap.

PROOF:

Since a leaf in a heap is a node with no left son, for every leaf we should have; $2i > n$. That is; $i = \lfloor n/2 \rfloor + 1, \lfloor n/2 \rfloor + 2, ..., n$.

(3) CLR 6.2-6

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 $\geq$ 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, which is the number of edges in the longest path 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 worst-case running time is $\Omega(\lg n)$.

(4) CLR 6.3-3

Let H be the height of the heap.

Two subtleties to beware of:

- Be careful not to confuse the height of a node (longest distance from a leaf) with its depth (distance from the root).
- If the heap is not a complete binary tree (bottom level is not full), then the nodes at a given level (depth) don't all have the same height. For example, although all nodes at depth H have height 0, nodes at depth $H-1$ can have either height 0 or height 1.

For a complete binary tree, it's easy to show that there are $\left\lceil \frac{n}{2^{h+1}} \right\rceil$ nodes of height h. But the proof for an incomplete tree is tricky, and is not derived from the proof for a complete tree.

The proof is by induction on h.

- Base case: Show that it's true for $h = 0$
  (i.e., that # leaves$\leq \left\lceil \frac{n}{2^{h+1}} \right\rceil = \left\lceil \frac{n}{2} \right\rceil$).

  The nodes at height 0 (i.e., the tree leaves) are at depths H and $H-1$. They consist of

  - all nodes at depth H
  - the nodes at depth $H-1$ that are not parents of depth-H nodes

  Let x be the number of nodes at depth H — that is, the number of nodes in the bottom (possibly incomplete) level.

  Note that $n-x$ is odd, because the $n-x$ nodes above the bottom level form a complete binary tree, and a complete binary tree has an odd number of nodes (1 less than a power of 2). Thus if n is odd, x is even, and if n is even, x is odd.

  To prove the base case, we must consider separately the case in which n is even (x is odd) and the case in which n is odd (x is even). Here are two ways to do this: The first requires more cleverness, and the second requires more algebraic manipulation.

  - First method of proving base case:

  * If n is odd, then x is even, so all nodes have siblings — i.e., all internal nodes have 2 children. Thus #internal nodes = #leaves − 1 so n = #nodes = #leaves + #internal nodes = $2 \cdot$ #leaves − 1. Thus #leaves = $\frac{(n+1)}{2} = \left\lceil \frac{n}{2} \right\rceil$ (last step true because n is odd).
  * If n is even, then x is odd, and some leaf doesn't have a sibling. If we gave it a sibling, we would have $n+1$ nodes, where $n+1$ is odd, so the case we analyzed above would apply. But we would also increase the number of leaves by 1, since the new node's parent already has a child, so it was not a leaf (i.e., we added a leaf without removing one). By the odd-node case above, #leaves + 1 = $\left\lceil \frac{(n+1)}{2} \right\rceil = \left\lceil \frac{n}{2} \right\rceil + 1$ (last step true because n is even); hence, the number of leaves is $\left\lceil \frac{n}{2} \right\rceil$.

– Second method of proving base case:

Note that at any depth $d < H$ there are 2^d nodes, because all such tree levels are complete.

  * If x is even, there are $\frac{x}{2}$ nodes at depth $H-1$ that are parents of depth-H nodes, hence $2^{H-1} - \frac{x}{2}$ nodes at depth $H-1$ that are not parents of depth-H nodes. Thus the total number of height-0 nodes is:

$$\begin{aligned} x + 2^{H-1} - \frac{x}{2} &= 2^{H-1} + \frac{x}{2} \\ &= \frac{(2^H + x)}{2} \\ &= \left\lceil \frac{(2^H + x - 1)}{2} \right\rceil \text{ (because } x \text{ is even)} \\ &= \left\lceil \frac{n}{2} \right\rceil \end{aligned}$$

($n = 2^H + x - 1$ because the complete tree down to depth $H-1$ has $2^H - 1$ nodes and depth H has x nodes.)

  * If x is odd, by an argument similar to the even case we see that the number of height-0 nodes is:

$$\begin{aligned} x + 2^{H-1} - \frac{(x+1)}{2} &= 2^{H-1} + \frac{(x-1)}{2} \\ &= \frac{(2^H + x - 1)}{2} \end{aligned}$$

$$= \frac{n}{2}$$
$$= \left\lceil \frac{n}{2} \right\rceil \text{ (because } x \text{ odd} \Rightarrow n \text{ even)}$$

- Inductive step: Show that if it's true for height $h-1$, it's true for h.

  Let n_h be the number of nodes at height h in the n-node tree T.

  Consider the tree T' formed by removing the leaves of T. It has $n' = n - n_0$ nodes. We know from the base case that $n_0 = \left\lceil \frac{n}{2} \right\rceil$, so $n' = n - n_0 = n - \left\lceil \frac{n}{2} \right\rceil = \left\lfloor \frac{n}{2} \right\rfloor$. Note that the nodes at height h in T would be at height $h-1$ if the leaves of the tree were removed — that is, they are at height $h-1$ in T':

  $n_h = n'_{h-1}$

  By induction we can bound n'_{h-1}:

  $n_h = n'_{h-1} \le \left\lceil \frac{n'}{2^h} \right\rceil = \left\lceil \frac{\lfloor \frac{n}{2} \rfloor}{2^h} \right\rceil \le \left\lceil \frac{\frac{n}{2}}{2^h} \right\rceil = \left\lceil \frac{n}{2^{h+1}} \right\rceil$

(5) CLR 6.4-2

Initialization: At the beginning of the first iteration $i = \mathit{length}[A] = n$. The loop invariant is true because $A[1..i]$ contains $i = n$ elements and these are infact the n smallest elements of the array. At the end of the first iteration, the largest element has been placed in $A[n]$, and so $A[1..n-1]$ contains the $n-1$ smallest elements.

Maintenence: In each iteration we find the largest element in $A[1..i]$ and move it to $A[i]$. After the swap no element in $A[1..i-1]$ can be larger than the element in $A[i]$. And the same argument for the previous iteration guarantees that the element in $A[i]$ cannot be larger than the element in $A[i+1]$, which in turn cannot be larger than the element in $A[i+2]$ by the same argument. So at the beginning of the next iteration i will decremented and $A[1..i]$ must contain elements that are no greater than any of the elements in the $A[i+1..n]$ i.e., $A[1...i]$ contains the i smallest elements. At the beginning of each iteration we also have $A[i+1] \le A[i+2] \le A[i+3] \le ..A[n]$. The element chosen to be placed in $A[i]$ is guaranteed to be no greater than $A[i+1]$. So each iteration places a new element into $A[i+1...n]$ in sorted order, and the element is so chosen that $A[1...i]$ contains the i smallest elements.

Termination: The alrogithm terminates at $i = 2$. At the end of this iteration $A[1]$ contains the smallest element, and $A[2..n]$ contains the remaining $n1$ elements in non-decreasing sorted order. Thus $A[1..n]$ contains the n elements in sorted order.

(6) CLR 6.5-8

Without loss of generality, assume that the k lists are sorted in descending order. Assume that the function GET-NEXT(i) returns the next element from the sorted list i, initially starting with the first element. If list i is empty it returns $NULL$.

When merging several sorted lists, the basic operation to be performed at each step is to get the maximum of the current front-most elements of the k lists and output it. Then consider the next element in the list (if it exists) from which the maximum was output. We can perform this operation efficiently using a heap. We maintain a heap of the current largest elements from the k lists. We also store its list number along with its value. So each element in the heap is represented by an ordered pair $\langle x, y\rangle$, where x is the value and y is the list number.

The algorithm is a straightforward implementation of the description above. We store the heap in the array A, and we store the output array in B. Lines 1-3 perform the initialization of the heap. The *for* loop in line 4 extracts the maximum from the heap and inserts the next element from the list where the maximum came from.

```
MERGE-K-LISTS
1. for $i \leftarrow 1$ to $k$
2.       do $A[i] \leftarrow \langle$GET-NEXT($i$), $i\rangle$
3. BUILD-HEAP($A$)
4. for $i \leftarrow n$ downto 1
5.       do $\langle x, y\rangle \leftarrow$ HEAP-EXTRACT-MAX($A$)
6.             $B[i] \leftarrow x$
7.             $z \leftarrow$ GET-NEXT($y$)
8.             if $z \neq NULL$
```

```
9.            then HEAP-INSERT($A, \langle z, y\rangle$)
10.           else heap-size($A$) $\leftarrow$ heap-size($A$) - 1
```

The first loop in lines 1-2 clearly takes $O(k)$ time. The call to BUILD-HEAP in line 3 also takes $O(k)$ time. The call to HEAP-EXTRACT-MAX is $O(\lg k)$, as is the call to HEAP-INSERT. The rest of the statements in the *for* loop are all $\Theta(1)$. Since the loop is repeated n times, the entire loop is $O(n \lg k)$, and thus the entire algorithm is $O(n \lg k)$.

(7) CLR 6-3

(a)

2	3	8	12
4	5	9	14
16	∞	∞	∞
∞	∞	∞	∞

Figure 1: Table for Problem CLR 6-3.

(b) Each row and each column is sorted in non-decreasing order left-to-right and top-to-bottom, respectively. So if $Y[1;1]$ = inf then all elements in the first row and first column must be no less than $Y[1;1]$ i.e., they must all be ∞. This makes $Y[2;1] = Y[3;1]Y[4;1] =:::= Y[m;1] = \infty$. Once the left-most element in each row is 1 all other elemnts in that row must also be ∞. Thus the entire tableau is empty (each element is ∞). If $Y[m;n] < \infty$ then all elements in the bottom row and right-most column are also $< \infty$. Once the right-most element in each row is finite, all other elements in each row must also be finite. So every element in the tableau is $< \infty$ and the tableau if full.

(c) We wish to formulate an algorithm that removes the smallest element from a Young Tableau, and adjust the remaining elements so that the matrix still remains a Tableau. We know that $Y[1;1]$ contains the smallest element. We simply remove this element and replace it by the second smallest element. That would either be $Y[1;2]$ or $Y[2;1]$. The element that is swapped in to replace $Y[1;1]$ leaves a vacant slot that must be filled in by the correct successor. The same method used for replacing $Y[1;1]$ can be applied recursively to repeatedly find the next element to swap in. This recursion continues until we either reach $Y[m;n]$ or an ∞ element, at that point we simply generate an infinity to swap in for that element.

The correctness is guaranteed by the fact that on each recursive call we remove an element and replace it by the smaller one of the two neighbors (below and right). Thus on each step we can potentially disturb either the column sorted-order or the row sorted-order for that particular element. Using the smaller one of the two neighbors ensures that the sorted ordering is preserved within the row and column of the replaced element. For example, if the one below is smaller then using that as a replacement keeps the row in sorted order, and if the one on the right is smaller then using that as the replacement keeps the column in sorted order. Thus on each step we maintain the ordering property, so even at termination of the algorithm the matrix will be a Tableau.

The running time of the algorithm can be given as $T(p)$, where $p = m + n$. We start by searching for the smallest element in the m x n matrix. At each step we perform one comparison and one copy, which can be considered $\Theta(1)$ work. In the next step we search for the smallest element in either a mxn-1 or $m - 1$xn matrix, and so on. At each step we reduce the matrix being worked on by one column or one row. So $T(p)$ can be written as:

$$T(p) = T(p - 1) + \Theta(1)$$

The algorithm terminates when $p = 1$ so there will be p recursive calls each performing $\Theta(1)$ amount of work. So the running time is bound by $O(p) = O(m + n)$.

(d) We can insert a new element into the Tableau at $Y[m;n]$, and then recursively compare it against its left and top neighbors to move it up the matrix. We begin be removing the infinity in $Y[m;n]$ and replacing it with the new element. Then we compare this new element agianst $Y[m;n-1]$ and $Y[m-1;n]$, if one of them is greater then the new element is swapped with that one, if both of them are greater, then swap with the one that is greater between the two, and if both are smaller then we need to do nothing. The same procedure can be recursively applied to repeatedly swap the new element until it either reaches $Y[1;1]$ or both its left and top neighbors are greater. The running time for this algorithm is similar to that for EXTRACT-MIN, we start with a matrix of size m x n and on each recursive call we reduce the matrix being worked on by either one column or one row. The algorithm must terminate after $m+n$ recursive calls at the most. The recurrance $T(p)$ for INSERT is the same as the one given for EXTRACT-MIN in part (c) and the running time is also $O(m+n)$.

(e) 1. Initialize the matrix with all elements as infinity: $O(n^2)$ time 2. Insert each element one by one, using the INSERT procedure: $O(n)$ for each of the n^2 elements i.e., $O(n^3)$ time. 3. Call EXTRACT-MIN n^2 times to obtain the numbers in non-decreasing order: $O(n)$ for each of the n^2 elements i.e., $O(n^3)$ time. Thus we can sort the given n^2 numbers in $O(n^3)$ time.

(f) We can do this by recursively comparing the given number to elements of the Tableau. Start by comparing against $Y[1;n]$ i.e., the element in the upper right corner. If $Y[1;n]$ is greater than the given number then we know that all elements in the n'th column of the tableau will be greater also. And so in this case we may eliminate the n'th column and recursively call this search procedure on a sub-matrix with $Y[1;n-1]$ as the upper-right corner. However if $Y[1;n]$ was less than the given number then we know that all elements in the first row are also less than the given number. Thus we may eliminate the firist row and recursively call this search procedure on the subm-matrix with $Y[2;n]$ as the upper-right corner. Thus we see that with one comparison we can eliminate one row or one column and arrive at a smaller problem. The recursion equation is:

$$T(p) = T(p-1) + \Theta(1)$$

The algorithm terminates when $p = 1$ so there will be p recursive calls each performing

$\Theta(1)$ amount of work. So the running time is bound by $O(p) = O(m + n)$. At the end of $m + n$ calls we will either have found the element or concluded that it is not present in the tableau.