Recitation Problems - Com S 311

Week of Feb 26^{th} - Mar 2^{nd}

1. What are the minimum and maximum number of elements in a heap of height h?
If the heap is complete, and its height is h, the maximum number of nodes is $2^{h+1} - 1$. If the lowest level has just one node, and the other levels are complete, the minimum number of nodes is $2^h - 1 + 1 = 2^h$ elements.

2. Show that an n-element heap has height $\lfloor \log n \rfloor$.
If a tree is complete, then there are $2^{h+1} - 1$ nodes (assuming a tree with just one node has height 0). If there are n nodes in the heap and the height of the tree is h, then $2^h \leq n$.

If the tree has height h, it must contain at least one more than $2^h - 1$ nodes. We have $n \leq 2^{h+1} - 1$ because the tree must contain at most $2^{h+1} - 1$ nodes. So

$$\begin{aligned} 2^h &\leq n < 2^{h+1} \\ h &\leq \log_2(n) < h + 1 \end{aligned} \tag{1}$$

Since height h is an integer, $h = \lfloor \log_2(n) \rfloor$.

3. How can we efficiently search in $O(\log n)$ time for a particular key element in a heap?
We **CANNOT**. A heap is not a binary search tree. We know almost nothing about the relative order of the $\frac{n}{2}$ leaf elements in a heap, certainly nothing that lets us avoid doing a linear-search on them. All we know is that the minimum element is at the root node in a min-heap; the maximum element is at the root node in a max-heap.

4. Consider a max-heap T, with distinct elements
   (a) Where in T does the maximum element reside?
   The maximum element in a max-heap resides at the root node.
   (b) Where in T might the smallest element reside?
   The minimum element in a max-heap is one of the leaf nodes but we don't know which leaf node.

5. Is an array that is sorted in increasing order a min-heap?
Yes. The min-heap property that the parent node value is less than or equal to its child nodes $(2i)$ and $(2i+1)$ holds for all nodes that have children.

6. Is the array with values [23, 17, 14, 6, 13, 10, 1, 5, 7, 12] a max-heap? Justify your answer with a proper explanation. Not a max-heap. Child with value 7 (index $9 = 2 \times 4 + 1$) is greater than its parent with value 6 (index 4).

7. Given k arrays, each of them containing n integers in increasing order, write as efficient an algorithm as possible to output an array containing all the kn integers sorted in ascending order.
Brute force: Merge the arrays in sequence. The sizes of the arrays after merging will increase. Time to merge is $O(\text{array size})$.

$$n(1 + 2 + 3 + ... + (k-1)) = \frac{n \cdot (k-1)k}{2} \in O(nk^2) \tag{2}$$

Heap Idea:
(Step 1) Pick the first element from each array and create a min-heap. The size of the heap is k because we have k arrays. Each node in the min-heap is a triple (v, i, j), where v indicates the value of the node and i, j indicates the fact that the value is obtained from the j^{th} index of the i^{th} array. That is $A_i[j] = v$. The runtime for building this heap is $O(k)$.

(Step 2) Add the minimum element from the heap to the output array. Note that the root of the min-heap initially contains the minimal element among the $k \cdot n$ elements as the arrays are already sorted and we have picked the first element from each array to form the min-heap.

(Step 3) If the value that is added in Step 2 is the j^{th} element from the i^{th} array and $j < n$, then replace the root of the heap with $A_i[j+1]$. Otherwise, replace the first with the last element, and reduce the size of the heap by 1. To maintain the heap, call Heapify-Down. This takes $O(\log k)$ time.

(Step 4) Repeat Step 2 until the heap is empty. Total number of repeat-until operations is $k \cdot n$ as there are $k \cdot n$ elements. In each iteration, Heapify-Down takes $O(\log k)$ time. Total runtime is $O(k \cdot n \log k)$.

Note that in the below pseudocode, each element in array representation of heap is $< value, i, j >$

- $val(H[index]) = value$
- $arr(H[index]) = i$
- $loc(H[index]) = j$

Algorithm 1 Pseudocode (given $A_1, A_2, ..., A_k$, output B)

make a min-heap with $A_1[1], A_2[1], ..., A_k[1]$;
$size = k$;
for iteration = 1 to $k \cdot n$ **do**
    add $val(H[1])$ to B[iteration]
    $j = loc(H[1])$
    if $j < n$ **then**
        $i = arr(H[1])$;
        $H[1] =< A_i[j+1], i, j+1 >$;
    else
        $H[1] = H[size]$;
        $size - -$;
    Heapify-Down($H, 1, size$);
Output B;