Algorithms I

Tutorial 3 Solution Hints

August 26, 2016

Problem 1

For a FIFO queue, we store a pair (i, x), where x is the value and i is a counter which is equal to the number of elements inserted before x. So, for the first element that is enqueued $i = 0$. Each time we insert a new element, we increment i. We'll use a min heap which is ordered by i.

For a stack, we store the same pair (i, x) in the same fashion. However, we use a max heap instead of min heap (ordered by i).

Problem 2

We start from the root of the heap. We check if the value at root is less than m. If not, we stop. Otherwise, we add 1 to the answer, and repeat the same procedure for its left and right child.

Problem 3

We'll use a max heap of the pairs (x_i, v_i) (ordered by v_i). We maintain a *position* array such that $position[x]$ represents the index of the pair (x, v) in the heap. We maintain this information while all the operations. i.e. whenever we swap heap values at i and j, we also swap $position[i]$ and $position[j]$. So, given (x, v), we can find its index in the heap. $getMax()$ is trivial. If $decreaseKey(x, v)$ is called, we update the value at x (we already know its index in the heap). Now, we need to call $heapify$ at this index because heap property may have been vioated after the update.

Problem 4

We'll use two hash tables H_A and H_B for A and B respectively. We insert all the elements from A in H_A and all the elements in B in H_B. Now, we check if $A \subseteq B$ and $B \subseteq A$. For this, we check if for every $1 \leq i \leq |A|$, $A[i]$ is present in H_B. And similarly, check if for every $1 \leq i \leq |B|$, $B[i]$ is present in H_A. If all of the conditions satisfy, this means A and B are equal as sets otherwise not.

Problem 5

4, 5, 9

Problem 6

We use the same idea that is used in the selection algorithm (k^{th} order statistics). We partition the elements of the array using a pivot element. If the left part contains $\leq k$ elements, we recursively search in that part. Otherwise, we subtract the size of left part from k and search in the right part.

Problem 7

If the array contains only one element, that element is the majority element. Otherwise, we divide the array into two equal parts. Let's call them left and right. We recursively check if the left and right part have a majority element. If the current array has a majority element, either left or right (or both) must have that element as the majority element. Also if both left and right part have a majority element, they must be equal. So, there is at most one candidate for the current array. We check if the candidate is actually majority element. Running time: $T(n) = 2 \cdot T(n/2) + O(n)$. $O(n)$ algorithm was discussed in the class.

Problem 8

This can be solved using a modification of merge sort. If the size of the array is 1, the number of such pairs is 0. Otherwise, we divide the array into two halves $left$ and $right$. All the elements in $left$ have index less than all the elements in $right$. We recursively calculate the number of such pairs for $left$ and $right$. Now, we will have to add the number of pairs (i, j) such that $i \in left$ and $j \in right$ and $a[i] > a[j]$. Such pairs can be counted while merging the two halves. We maintain a variable *count* (initialized by 0). While merging $left$ and $right$ whenever we encounter (i, j) such that $right[j] < left[i]$, we add $(|left| - i)$ to *count* (assuming i is 0 based). This is because for all k such that $k \geq i$, (k, j) is a pair that satisfies the required conditions. So, we return the sum of the count of such pairs in $left$, in $right$ and *count*.

Problem 9

We sort A_2. Now, for each element of y of A_1, we check if $(x - y)$ exists in A_1. This can be done using a binary search.