1.
Solution:

For the delete operation, we need to define some helper methods and follow the below steps:

1. For element at A[i], change its priority to some large value (larger than the current maximum) :takes O(1)time
2. Then shift that A[i] up, to maintain the heap property: takes O(log n) time
3. Now extract it and remove it using getMax(): takes O(1) time
4. Finally, heapify the structure again: takes O(log n) time

So in total time complexity is O(1) + O(log n) + O(log n) + O(1), which is equivalent to O(log n).

2. Solution:
Pseudocode:

Let a be the array in which elements of the max heap are stored.
The number of nodes in a binary tree of height h = 2h+1-1
We know that in case of the max heap the k largest elements exist between the root and the maximum height k-1 in the binary max heap.
Number of nodes in the binary tree of height k-1 = 2h+1-1 = 2k-1+1-1 = 2k - 1
So the k largest elements exist among the front 2k - 1 elements in the array.
So we can search the array from index 0 to index 2k - 2.
The required algorithm is shown below:
get_k_Biggest_Elements(Array_a)
integer m = 2k - 1
Create a new array b of size m.
The elements from index 0 to index 2k - 2 in array a are copied to array b.
  BUILD-MAX-HEAP(b)
  for i:= 1 to k
Heap-Extract-Max(b)
  MAX-HEAPIFY //This step results in O(log k) complexity
end for

Time Complexity Analysis:

Construct a new max-heap using the front 2k-1elements from the given array(which is a max heap). Note the height of the new max heap is k. The maximum element is present in the root node. We extract the maximum element from the heap and convert it back to the max heap. This step has a complexity O(log k). Since the previous step is performed k times, hence total time complexity = k * O(log k) = O(klogk).

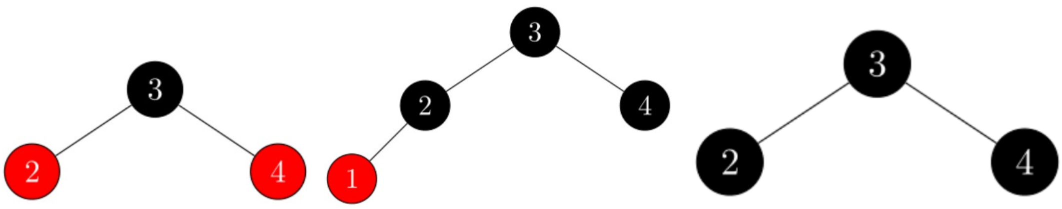
3. Solution:

Denote for a node j having two children its predecessor by p and its successor by s. First, we show by contradiction that the successor of j doesn't have left child. If s has a left child, then the key of s is greater than the key of left[s]. Also the key of s is also

larger than that of j, and since s has one left child, the key of left[s] is larger than that of j. Thus key[s] ≥ key[left[s]] ≥ key[j],which is a contradiction since s is the successor of j. So that successor of j has no left child.
Also we can show by contradiction that the predecessor of j doesn't have right child. Suppose p has a right child. Then we can see that the key of p is less than that of the right[p]. Also the key of p is less than that of j, and since p has a right child, so the key of right[p] is less than that of j. Thus key[p] ≤ key[right[p]] ≤ key[j], which is a contradiction, since p is the predecessor of j. Hence the predecessor of j has no right child.

4. Solution:

In a red-black tree, the restaurant tree after performing insertion and deletion operations may not be the same as the initial red-black tree. Because after insertion or deletion operations, the resulting tree may violate the property of the red-black tree.
For example:
Initial: insert 1: delete 1:



5. Solution:

Pseudocode:

initialize i=1 and j=1 and a sequence C
while i <= n and j<=n
if A[i] = A[j], do i++ and j++
else if A[i] > A[j], append A[j] to C and do j++
else if A[j] > A[i], append A[i[to C and do i++.
if i==n (B is remaining with some elements)
append rest elements of B in C
else if j == n (A is remaining with some elements)
append rest elements of A in C
C will contain the difference of sequence A and B.

Time Complexity :

O(n)