Q1.

Since we use simple uniform hashing, the average number of elements that hash to the same array would be $\alpha = \frac{n}{m}$, where n is the total number of elements in the hash table, and m is the number of slots.

(a). Given that we use sorted arrays as the chaining components for the hash table:
On average, the time complexity for search would be $O(1+\log(\alpha))$, Here, $O(1)$ is the time taken to compute $h(k)$, and $O(\log(\alpha))$ is the time to search for the key inside the array at slot $h(k)$. We achieve logarithmic runtime as we could perform binary search on the sorted array, since arrays allow random access. However, if α is very small, we would not have much benefit from the usual way of using linked list as the chaining component (runtime being $O(1+\alpha)$). But in the case that $n \gg m$, then the runtime would improve.

(b). Given that we need to run a merge sort every time we insert an element:
On average, the time complexity for insertion would be $O(1+\alpha\log(\alpha))$, where $O(1)$ is the time to compute $h(k)$, and $O(\alpha\log(\alpha))$ is the time to perform the merge sort on the array at slot $h(k)$. If we do not keep track of the current number of elements in the array, then we also have an additional cost of $O(\alpha)$ to search to the end for an empty spot, otherwise it would just take $O(1)$ time to append. Yet in the end, the insertion time would be worse than the original scheme of using linked lists, as it only costs $O(1)$ time to add the key to the front of the linked list.

(c). Deletion with sorted arrays would remain the same as using linked lists, i.e. $O(1+\alpha)$. $O(1)$ is the time to compute $h(k)$, and $O(\alpha)$ is the total cost for deletion. To delete, we first need to search for the key, so it takes $O(\log(\alpha))$ to perform binary search, then $O(1)$ to remove the key, and at the end, we need to fill in the empty space by shifting all elements greater than the deleted key one slot to the left. In the worst case, the deleted key is the first element, so we need to shift all remaining elements, which is $O(\alpha)$. Since $O(\log(\alpha)) \in O(\alpha)$, we have the overall cost for search-delete-shift as $O(\alpha)$. With linked lists, deletion is also $O(\alpha)$. We need to traverse the list to search for the previous element of the target, then change its pointer to the next ($O(1)$), and free memory for the target($O(1)$). In the worst case, the target is the last element, so we require $O(\alpha)$.

Q2.

His claim is not correct for all situations. There is no guarantee that the leaves of a complete binary tree are on the same level. There could be missing right-most nodes on the last level. As illustrated in Figure 1, all internal nodes of a complete binary tree have two children, but the leaves here are not on the same level, meaning that if we color all nodes as black, the number of black nodes from root to a leaf is not the same for all paths. This violates the red black tree property:

> For each node, all paths from this node to each of its leaves contain the same number of black nodes.

Hence, it is not a red black tree in this case.

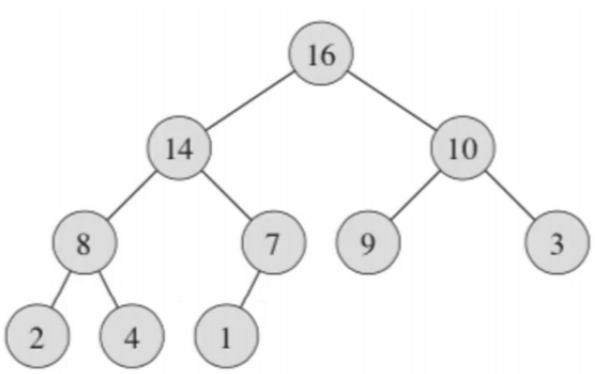

Figure 1: missing right-most nodes

However, if we have a perfect binary tree, which is a complete binary tree with all leaves on the same level (assuming leaves are nil), then it is valid red black tree:

1. the root is black
2. leaves are black nils
3. there is no consecutive red nodes along any path since there are no red nodes at all
4. all paths from a node to its leaves contain the same number of black nodes.

Q3.

An avl tree is a self balancing binary search tree that has the property that, for any node x in the tree, x has at most two children, and $x.left.key \leq x.key \leq x.right.key$. Although for an avl tree, we might need rotation for insertion and deletion operations in order to maintain the balance factor, we still guarantee that the bst property holds. So the node with the minimum key is the left-most node in the tree. We could find it by traversing down the tree from the root, and always choosing the left subtree to traverse, until we find the left subtree to be nil, then the current node contains the minimum key. Similarly, the node with the maximum key is the right-most node in the tree. Beginning from the root, we always traverse the right subtree, until the right subtree is nil, then the current node contains the maximum key.

Q5.

(a). Sub-problems: We use slightly different sub-problems. We find the longest strictly decreasing sequence(LDS) of the prefixes of the original sequence, where the LDS is ended with the last element(box size) of the prefix. We also find the length of such LDS. A solution to the original problem would be a longest sequence among all solutions to the sub-problems.
Relation between sub-problems:

- On LDS:
  Suppose $P_i = [(r_1, h_1), (r_2, h_2), \ldots, (r_i, h_i)]$ is the input sequence, then a LDS that ends with the element (r_i, h_i), is built from a LDS that ends with (r_k, h_k) for $k \in [1, i-1]$ where $r_k > r_i$ and $h_k > h_i$. There could be multiple k's that form the longest subsequence. As formula, LDS($P_i =$

longest($[(r_i, h_i)]$, $\underset{\substack{k\in[1,i-1]\\ r_k>r_i \wedge h_k>h_i}}{\text{longest}}$ (LDS(P_k) + $[(r_i, h_i)]$)). Here $'+'$ means concatenation of two sequences, and P_k is the sequence $[(r_1, h_1), (r_2, h_2), \ldots, (r_k, h_k)]$.

- On the length of LDS:

$$dp[i] = \begin{cases} 1 & i = 1 \\ \max(1, \underset{\substack{k\in[1,i-1]\\ r_k>r_i \wedge h_k>h_i}}{\max} (1 + dp[k])) & i > 1 \end{cases}$$

$dp[i]$ is the length of the LDS of P_i and the last element of the LDS is (r_i, h_i)

(b) Let P denote the input sequence $[(r_1, h_1), (r_2, h_2), \ldots, (r_n, h_n)]$. In the following psuedocode, we access the i^{th} element (box size) in P by $P[i]$, and access r_i by $P[i].r$ and h_i by $P[i].h$.

Algorithm 1 Bottom-up Longest Strictly Decreasing Sequence(P)

```
1: let dp[1..n] be an empty array          ▷ dp[i] stores the length of LDS with P[i] as the last element
2: let prev[1..n] be an empty array        ▷ prev[i] stores the index of the previous element of P[i]
3: for i = 1 to n do
4:     dp[i] = 1
5:     prev[i] = i
6:     for k = 1 to i − 1 do
7:         if P[k].r > P[i].r and P[k].h > P[i].h and 1 + dp[k] > dp[i] then
8:             dp[i] = 1 + dp[k]
9:             prev[i] = k
10:        end if
11:    end for
12: end for
13: int l = 0
14: int idx
15: for i = 1 to n do
16:    if dp[i] > l then
17:        l = dp[i]                       ▷ l will store the length of LDS(P)
18:        idx = i                         ▷ idx is the index of the last element in the LDS(P)
19:    end if
20: end for
21: let O[1..l] be the output sequence
22: while idx ≠ prev[idx] do
23:    O[l] = P[idx]
24:    idx = prev[idx]
25:    l = l − 1
26: end while
27: O[l] = P[idx]
28: return O
```

(c). Time complexity is $O(n^2)$. The computation of length of LDS is at line $3 - 12$, we have two for loops. The outer loop iterates through each element of the original sequence,