CSC263, Problem Set 1

Youngsin Ryoo

Question 1

a) Let $n = len(L)$. All elements in L are equal to k. Then, we have 1 comparison in line 12 for n times. Therefore, we have $\theta(n)$.

b) Let $n = len(L)$. Let $m = len(M)$. In the most case, we can do at most $n(1+m)$ comparisons: we'll compare first in line 12, which takes 1 comparison, and then jump to line 14 where $L[i] \neq k$ and in line 15, find_position will have have 1 comparison in line 4 for m iterations. The outer loop then will iterate n times, which gives $n(1+m)$ comparisons.
Let L be a list of length n where no elements are k. Let M be a list of length m where no elements of it are equal to elements in L. Then, for every n iterations, we'll have 1 comparison in line 12 and m comparisons in line 4. Therefore, we have $n(1+m) = n + nm$. Since $len(M) \geq len(L)$ by precondition, we have $\theta(m^2)$.

c) We have: Pr(element in L is k) $= \frac{1}{10}$
Pr(element in L is not k) $= \frac{9}{10}$
Note: value is chosen independently for the lists, which means overlap is possible in the lists. We always have one comparison at line 12 whether $L[i]$ is equal to k or not. Then, what we should consider is the case where the loop stops at its target at, let's say, j position in M in find_position. We have:

$$\begin{aligned}
\text{Expected running time} &= n[\frac{1}{10} \cdot 1 + \frac{9}{10} \sum_{j=1}^{m} 1(\frac{9}{10}^{j-1} \frac{1}{10} j)] \\
&= \frac{n}{10} + \frac{n}{10}(\frac{(\frac{9}{10})^{(m+1)}(m(\frac{9}{10}-1)-1)+\frac{9}{10}}{(1-\frac{9}{10})^2}) \\
&= \frac{n}{10} + \frac{n}{10}(\frac{(\frac{9}{10})^{m+1}(-\frac{m}{10}-1)+\frac{9}{10}}{\frac{1}{100}}) \\
&= \frac{n}{10} + 10n\left((\frac{9}{10})^{m+1}(\frac{-m-10}{10}) + \frac{9}{10}\right) \\
&= \frac{n}{10} + n\left((-m-10)((\frac{9}{10})^{m+1} + 9\right)
\end{aligned}$$

d) We have: $\Pr(k = n) = \frac{1}{n+1}$
$\Pr(k \neq n) = \frac{n}{n+1}$
Note: values assigned dependently because no overlap is possible.
That is, k is in L and M at most once. Also, in find_position, there is at least an element in A (which is M) that matches the target ($L[i]$). We have two cases:
1. There is no k in L and $L[i]$ matches an element in M.
2. There is one k in L and $L[i]$ matches an element in M.
For the first case, because there is no k, although we will have 1 comparison at line 12 for every iteration, we'll always enter the loop in the find_position. Since there is always one matching pair in L and M for each iteration, the total comparisons that account the else block of line 14 for n iterations (outer loop) is $\sum_{m=1}^{n} m = \frac{n(n+1)}{2}$. So, we have: $\frac{1}{n+1}(n + \frac{n(n+1)}{2})$.
For the second case, it is still the same that we have 1 comparison at line 12 for every iteration. However, now that there is one k in the L and M, and it's dependent, we have to account the probability that changes in the find_position loop depending on where k is located. Since k is in L and M, we have $\frac{1}{n}$ probability for each iteration in find_position. Then, for the second case, we have $\frac{1}{n}\sum_{j=1}^{n}(\frac{n(n+1)}{2} - j)$. Then, we have: $\frac{n}{n+1}\left(n + \frac{1}{n}\sum_{j=1}^{n}(\frac{n(n+1)}{2} - j)\right)$
First case and second case put together, which is our answer, is:

Expected running time

$$
\begin{aligned}
&= \frac{1}{n+1}(n + \frac{n(n+1)}{2}) + \frac{n}{n+1}\left(n + \frac{1}{n}\sum_{j=1}^{n}(\frac{n(n+1)}{2} - j)\right) \\
&= \frac{1}{n+1}(n + \frac{n(n+1)}{2}) + \frac{n}{n+1}\left(n + \frac{1}{n}(\frac{1}{2}\sum_{j=1}^{n}(n^2+n) - \sum_{j=1}^{n} j)\right) \\
&= \frac{1}{n+1}(n + \frac{n(n+1)}{2}) + \frac{n}{n+1}\left(n + \frac{1}{n}(\frac{1}{2}(n^3+n^2) - \frac{n(n+1)}{2})\right) \\
&= \frac{1}{n+1}(n + \frac{n(n+1)}{2}) + \frac{n}{n+1}\left(n + \frac{n^3-n}{2n}\right) \\
&= \frac{1}{n+1}(n + \frac{n(n+1)}{2}) + \frac{n}{n+1}\left(n + \frac{n^2-1}{2}\right)
\end{aligned}
$$

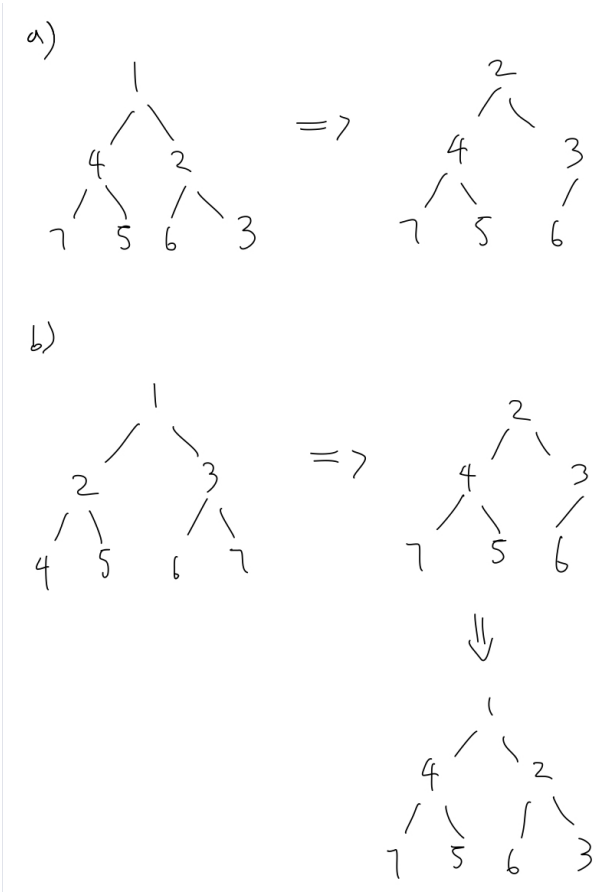
Question 2

a) A collection of objects where each object stores a surplus of dosages or shortage as a priority and a centre name as its value.

b) We will implement two max heaps. One max heap will be referred to Surplus-Max heap and the other will be referred to Shortage-Max-heap. According to the question,

a centre files its report as either a surplus or shortage. If it is a surplus of dosages, the staff member will put it in the Surplus-Max heap. If it is a shortage of dosages, the staff member will put it in the Shortage-Max-heap. This insertion of the center's status to each heap could be done with an operation insert(PQ, status, centre_name), where status and centre_name together forms an object in the array. This insert operation will take $O(logn)$, where n is the height of the tree, because it will be bubbled up with at most $logn$ swaps. To find the centres with the largest surplus and the largest shortage, we simply need to take out the first node in each Surplus-Max heap and Shortage-Max heap. We could use Extract_Max(PQ) operation for this process. Extract_Max(PQ) will find the largest surplus from Surplus-Max heap and extract it after; it will find the largest shortage from Shortage-Max heap and extract it as well. Extract_Max(PQ) will also take $O(logn)$, because bubbling down takes at most $logn$ swaps and extracting the max takes constant time.

Question 3



c) In order for a heap to be an EIS heap, every ancestor of the last leaf must be smaller than its sibling and the last leaf must be smaller than its sibling if it has one.

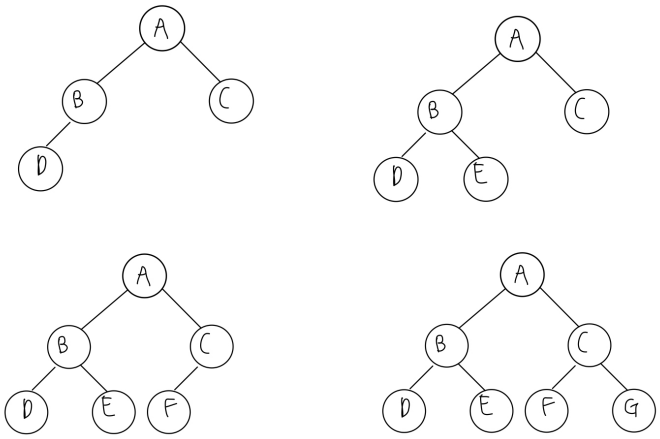


Figure 1: An image of for (d)

d) We want to show that the route from A to the last node in the original heap must stay the same for EIS heaps but not for non-EIS heaps. That is, the route that gets bubbled down during Extract_Min is the same as the route that gets bubbled up later during Insert.

Assume every ancestor of the last node is smaller than each of their siblings. Then, we extract the min and put the last node in place of the root. Since all the nodes in the route from the root to the original last leaf are smaller than its sibling, when the newly inserted (once was the last node) value will be swapped with the number that is smaller than itself and its other child, it will strictly follow down the route from the root to the original last leaf. For example, in the Figure 1, the last node D will be inserted in place of A and gets bubbled down with B because B is smaller than C. When the last node is E, E will be inserted in place of A and gets bubbled down with B because B is smaller than C. When the last node is F, F will be inserted in place of A and gets bubbled down with C because C is smaller than B. Note that bubble down is strictly happening on the path where the original last leaf's ancestors are. Finally, when the last node is G, G will be inserted in place of A and gets bubbled down with C because C is smaller than B. So, when we insert the minimum value (the original root) again to the heap, since it is the minimum value, it will be bubbled up all the way through its ancestors to the root. Since no other parts of the tree got altered during the bubble down, the original heap will be restored, which is the EIS heap.

Assume not all ancestors of the last node are smaller than it sibling. This means that when it bubbles down during the Extract_Min, the nodes other than the ancestors of the original last node will be changed. So, when we bubble up during the insertion, the tree would not be able to restore the original heap since bubble up can only happen through the route from the last node to the root. For example, in the first tree in Figure 1, if B was bigger than C, when we bubble down, C will be swapped with D. This cannot be changed to the original node when we bubble up later because bubble up can only happen on the path from the position of the original last node to the root. ■

Question 4

a) The best case number of nodes that must be examined to find the data associated with v is found when v is in the root. So, we will examine 1 node.

b) The worst case number of nodes that must be examined to find the data associated with v is $k+1$ nodes.
Proof:
We have $2^k - 1$ nodes. Since we are considering a complete binary tree, even if we delete 1 node from the complete binary tree, the tree will still be nearly complete and the height would remain the same. So, if we delete one node, we have 2^k nodes. This means that the height of the tree is $\lfloor log 2^k \rfloor = k$. The worst case would happen when the longest path from the root to the last node is examined. In this case, we will have $k+1$ nodes. Therefore, the worst case number of nodes which must be examined to find the data associated with v is $k+1$ nodes.

c)

Expected running time

$$
\begin{aligned}
&= \frac{1}{2^k - 1} \sum_{i=1}^{k} i \cdot 2^{i-1} \\
&= \frac{1}{2^k - 1} \cdot \frac{1}{2} \sum_{i=1}^{k} i 2^i \\
&= \frac{1}{2^{k+1} - 2} \left(\frac{2^{k+1}(k(2-1)-1)+2}{(1-2)^2} \right) \\
&= \frac{1}{2^{k+1} - 2} (k 2^{k+1} - 2^{k+1} + 2) \\
&= \frac{2(k 2^k - 2^k + 1)}{2(2^k - 1)} \\
&= \frac{k 2^k - 2^k + 1}{2^k - 1}
\end{aligned}
$$

d) $\Pr(v \text{ at the root}) = 0.2$
$\Pr(v \text{ not in the tree}) = 0.4$

Pr(neither the first or the second case) = 0.4

$$\text{Expected running time} = 0.2(1 \cdot 1) + 0.4k + 0.4(\frac{1}{2^k - 2} \sum_{i=2}^{k} i2^{i-1})$$

(first term when v at the root, second term when v not in the tree, and third term when neither.)

(in the third case, we do $\frac{1}{2^k-2}$ because v is at non-root position always.)

$$\begin{aligned}
&= \frac{1}{5} + \frac{2k}{5} + \frac{2}{5}\left(\frac{1}{2^k - 2} \cdot \frac{1}{2} \sum_{i=2}^{k} i2^i\right) \\
&= \frac{1}{5} + \frac{2k}{5} + \frac{1}{5(2^k - 2)}\left(\frac{2^{k+1}(k(2-1)-1)+2}{(1-2)^2}\right) \\
&= \frac{1}{5} + \frac{2k}{5} + \frac{1}{5(2^k - 2)}\left(\frac{2^{k+1}(k-1)+2}{1}\right) \\
&= \frac{1}{5}\left(1 + 2k + \frac{k2^{k+1} - 2^{k+1} + 2}{2^k - 2}\right)
\end{aligned}$$

Question 5

Since it is a max heap, n is the largest number and therefore the root. Consider a number that is one less than n. It cannot be in the root but cannot go under any other number other than n. So, this number is always under the root, n, therefore, it is the largest number that y can be. This means that $n = y + 1$. So, the lower bound of n is $y + 1$.

Let's consider the upper bound. For complete trees, we always have $h_R = \lfloor logy \rfloor$ and $h_L = \lfloor logy \rfloor + 1$. Consider the nearly complete tree. In this case, the height of the right subtree (h_R) must be equal to or one less than the height of the left subtree(h_L). The smallest number that y can be in this case is the consecutive numbers from 1 to y. Since h_L could be either $h_R + 1$ or h_R, the number of nodes in h_R must be smaller than or equal to the number of nodes of h_L. So, we must put y in the right child of the root to account the difference in number of nodes that is made by the heights of H_L.

The total number of nodes we have for the right subtree is $y = 2^k - 1$ (y is from 1 to y consecutively). Then, for complete binary trees, we have:

$$\begin{aligned}
y &= 2^k - 1 \\
y + 1 &= 2^k \\
log(y + 1) &= log(2^k) = k
\end{aligned}$$

Then,

$$\begin{aligned}
h_R &= k - 1 \\
h_L &= h_R + 1 = k - 1 + 1 = k
\end{aligned}$$

Therefore, in nearly complete tree, we have:

$$\begin{aligned} 2^{k-1} - 1 < y < 2^k - 1 \\ 2^{k-1} < y + 1 < 2^k \\ log2^{k-1} < log(y+1) < log2^k \\ k - 1 < log(y+1) < k \\ k - 1 \leq \lfloor log(y+1) \rfloor < k \end{aligned}$$

That is, $h_L = \lfloor log(y+1) \rfloor = k - 1 = h_R$
Since we have to add 1 to the height to get the number of nodes, we then have $2^{\lfloor log(y+1) \rfloor + 1}$ nodes. We also add y because we know we have y nodes in the right subtree. Therefore, $n = y + 2^{\lfloor log(y+1) \rfloor + 1}$.