1) a) $f(n)$ is $O(g(n))$, given (positive) functions $f(n)$ and $g(n)$, then we can say that $f(n)$ is $O(g(n))$ if and only if there exist positive constants c and n_0 such that $f(n) <= c * g(n)$ for all $n >= n_0$

To proof that $f(n) \in O(n^k)$

$f(n) = a_k n^k + a_{k-1} n^{k-1} + \ldots + a_0$

$= n^k * (a_k + a_{k-1} n^{-1} + \ldots + a_0 / n^k)$

from here n^k is $O(n^k)$

$a^k + a_{k-1} n^{-1} + \ldots + a_0 / n^k = a_k + a_{k-1}/n + \ldots + a_0 / n^k$

since the numerator is a constant and denomenator is a power. The positive powers grows faster, thus, as $n \to \infty$, $a_{k-1}/n + \ldots + a_0/n^k \to 0$ which means if there is large enough n then $a_{k-1}/n + \ldots + a_0/n^k$ will be arbitarily close to 0. Thus $a_k + a_{k-1}/n + \ldots + a_0/n^k$ is $O(1)$

Show that the function $f(a_k) = a_k$ is not $O(1)$

From the defination of Big-On

$a_k \leq c \cdot 1 \quad \forall a_k \geq n_0$

But this is impossible, thus a_k is $O(1)$

Proofing based on multiplication rule

Let $f_1(n) = n^k$ $\quad O(g_1(n)) = O(n^k)$

$f_2(n) = a_k + a_{k-1}/n + \ldots + a/n^k$ $\quad O(g_2(n)) = O(1)$

there exist positive constant c_1, c_2, n_1 and n_2 such that

$f_1(n) \leq c_1(n^k)$ for all $n \geq n_1$

$f_2(n) \leq (c_2)(1)$ for all $n \geq n_2$

Let $n_0 = \max(n_1, n_2)$, then multiplying gives

$f_1(n) f_2(n) \leq c_1 c_2 g_1(n) g_2(n)$ for all $n \geq n_0$

$f_1(n) f_2(n)$ is $O(g_1(n) g_2(n))$

$\therefore n^{k} * a_k + a_{k-1}/n + \ldots + a_0/n^k$ is $O(n^{k} * 1)$ which is $O(n^k)$

b)

Based on the definition of big Oh, f(n) is O(g(n)), given (positive) functions f(n) and g(n), then we say that f(n) is O(g(n)) if and only if there exist positive constants c and n0 such that f(n)<=c*(g(n)) for all n>=n0.

Big O represents the upper bound of time complexity which is also the worst case scenario. f(n) = O(g(n)) means that f(n) has the worst case time complexity of O(g(n)) but f(n) could also be running in faster time complexities like O(log n) and O(1). Thus, big O can be thought of as a set containing all the functions whose time complexity is no worse than g(n). Moreover, if a function belongs in the set of a faster time complexity, it also belongs in the set of a slower time complexity. For example, if a function f(n) ∈ O(1) then f(n) ∈ O(n). However, it does not produce a tight upper bound.

To proof that $f(n) \in O(1)$ and $f(n) \in O(n)$

let $f(n) = 5$

To proof that $f(n)$ is not $O(1)$

Based on the defination of Big Oh : $5 <= C * 1$, for all $n >= n0$

However, this is not possible thus $f(n)$ is $O(1)$

proof that $f(n)$ is $O(n)$ thus, $n >= 5$

$5 <= C * n$, for all $n >= n0$

let $c = 1$

Assume an infinite set [1,2,3,n,10n,2n+1,..., 3n+3], when we say 3n+3 is O(n) we mean that 3n+3 belongs in the set O(n). Therefore, writing $3n+3 \in O(n)$ is more accurate compared to 3n+3 = O(n).

Furthermore, when 3n+3 = O(n) and 3n+3 = O(n^2) it will lead to O(n) = O(n^2) which is incorrect. If the equal notation (which is an equivalence relation) is used, the relation is symmetric, reflexive and transitive. Thus, when we use f(n) = O(g(n)) and based on the symmetric property thus g(n) = O(f(n)). However, this is not true. On the contrary, $\in$ is not symmetrical so it will not have the same problem.

Proof for not symmetric

Let f(n) = 1 and g(n) = n

The desination of BigOh is $f(n) <= c*g(n)$, for all $n >= n0$

Show f(n) is O(g(n))

$1 <= c*n$, for all $n >= n0$

let c = 1

$n >= 1$, thus proven

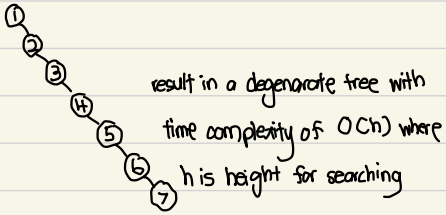
Show g(n) is O(f(n))

$n <= c*1$, for all $n >= n0$

thus, it is not possible, as we cannot get an value for all n

2a) Since the array is sorted, inserting into a regular binary search tree from either the first or last element as the root will lead to a degenerate tree. For example given sorted array [1,2,3,4,5,6,7] we will have a binary search tree that is

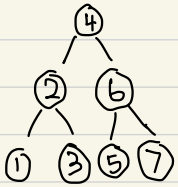
When converting array to binary search tree



result in a degenerate tree with time complexity of O(n) where h is height for searching

The time complexity of binary search trees depends on the height of the tree, which can be represented as O(h) where h is the height of the tree. Since a degenerate search tree has internal nodes that can only have one child, it means the time complexity of the degenerate search tree is O(n) where n is the number of elements. Therefore to have the optimal time complexity of O(log n), an AVL tree can be used since an AVL tree is a balanced binary search tree. Firstly, we can find the size of the array using built-in functions such as "sizeof()"in C language and "array.length" in Java which will be constant O(1) time. Then, from the size, we can find the midpoint (middle index) of the array by taking array size minus 1 and divide 2 [(array_size-1)/2], this is also O(1). The middle element can then be used as the root of the AVL tree. Then, we can repeat the process with the left subtree and the right subtree, making the middle element of the left half of the array the left child of the root,and making the middle element of the right half of the array the right child of the root and repeat until the array cannot be split in half. This way, the left subtree will contain elements that are smaller than the root and the right subtree will contain elements bigger than the root. Using the middle element as the root will help improve the efficiency of the conversion as we will not need to restructure the binary search tree (which has a time complexity of O(log n)) to maintain height. Thus, when doing a search, only half the total elements need to be searched leading to a time complexity of O(log n). Overall, this algorithm will take O(n) time as it only needs to iterate through the sorted array and insert into the tree. As an example

sorted array [1,2,3,4,5,6,7]



by converting to a balance search tree which is AVL tree that have a time complexity of O(n) since the array has already been sorted.

① | 1 | 2 | 3 | 4 | 5 | 6 | 7 |

↑

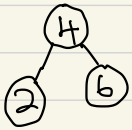
(size-1)/2 to get the middle index

(7-1)/2 = 3 array[3] = 4 = middle index

④

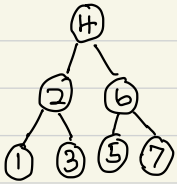
② | 1 | 2 | 3 | | 5 | 6 | 7 |

(3-1)/2 = 1 array[1] = 2 and 6 is middle index



③ [1] [3] [5] [7]

(1-1)/2 = 0 array[0] = 1, 3, 5, 7 = middle index



b)

For our software, we will use an AVL Tree structure to handle these software requirements.

For the insertion and deletion of the employees in set E, the AVL tree structure will allow this algorithm to have a runtime of O(log n) which will fulfill the specification set by the company. This is because the AVL tree is a balanced binary tree meaning that only half the tree needs to be searched as the left subtree would contain employees with early alphabetical letters in their name and the right subtree with the later alphabetical letters which is O(log n). Moreover, restructuring of the AVL tree is also O(log n). For the alphabetical sort of the employees in set E, we will use the tree sort algorithm to perform an in-order traversal on the binary search tree to get the alphabetical order of the employees in set E. This means that our AVL Tree will sort by earliest alphabetical letters as the left children and the later letters as right children. By implementing the in-order traversal method, we will be able to make the time complexity of this process to be O(n) since we need to traverse all the nodes. This will fulfill the requirements.

For the yearly promise aspect, we will store a global variable into the algorithm that will contain the x amount of stock based on the yearly promise given by the CEO each year so that each node will be able to point to it and be able to add on to their value of stocks every year. This method allows us to have a runtime of O(1) as it will allow us to refer to the amount of x stocks which will be the same for everyone in that year.

In conclusion, we chose this AVL tree because inserting and removing in O(log n) time was a requirement. Thus, this AVL Tree method will be the most suitable choice for the requirements given to us.

c)

In order to implement the system, we are going to use a heap data structure. Min heap will be a more suitable choice as we have to get the k highest priority flyers in the waiting list. Therefore, using a heap sort in the min heap will give us the k highest priority flyers with the time complexity of O(k log n) as heap sort using min heap will sort values in descending order. The sorting time complexity, O(k log n), does fulfill this requirement for searching k highest priorities flyers. Also, since we are using a heap data structure, inserting a new value will have a time complexity of O(1) and it will have a total of O(log n) including the upheap process. Deletion in a heap normally happens in the root of the data structure. Downheap, which has a time complexity of O(log n), should take place. However, in this question, we are asked to delete the node based on the customer's request meaning that we can cancel or remove the customer's request whenever they want which means that we need to remove any nodes in the heap data structure. Therefore, we have to use a locator which is a mechanism that keeps track of an element within its current position in the data structure (Goodrich & Tamassia, 2015). This will allow us to delete any nodes without searching through the whole heap data structure (O(n)). With locators, we will delete nodes that are in the heap data structure which now leads to a time complexity of O(1). After that, the removed node will be replaced by the last leaf node and downheap might take place which leads to O(log n). Overall, it will have a time complexity of O(log n).

References

Goodrich, M. T., & Tamassia, R. (2015). Algorithm design and applications. Wiley.