Name: Georges Hatem

CS 549 Programming Assignment

Due Date: February 27, 2022

## Describe an algorithm for the New Unknown Airline (NUA) Upgrade System:

An Algorithm for the New Unkown Airline (NUA) Upgrade System would be based on Heap Tree Algorithm since we need O(log(n)) for insertion of an element and O(log(n)) for removal of an element.

A Heap Tree has 2 basic rules:

1) The Heap Tree must be in order from decreasing to increasing (from the root to the external nodes)
2) The Heap Tree must be a Complete Binary Tree, meaning that the tree is filled from left to right.

These 2 rules have to always exist whenever we are inserting or removing elements to keep the Heap Tree.

So, let’s start our insertion process and explain why that would take a running time of O(log(n)). First off, when inserting an element into the Heap Tree, it has to be inserted at first at the left end (e.g. if we have n elements counting from 1 to n, the element inserted would be inserted at n+1 and the tree is filled from left to right.

Once we insert the element at the left end of the Heap Tree (at n+1), we need to heapify the tree so that it follows the Heap-order property. So, if the key of the added element is smaller than the key of the added element’s parent, then we have to swap the node of the added element and the node of the added element’s parent. We perform this procedure a lot of times until we receive a conclusion that the insertion process was submitted successfully.

The Algorithm of the insertion process of an element into the Heap Tree is as follows:

Algorithm HeapInsert(k,e):
Input: k represents the key, which is in this case the priority flyer status and e represents the element, which is the name of the frequent flyer in this case.
Output: A node (k,e) will be inserted in the Heap Tree

n ← n + 1
A[n] ← (k,e)
i ← n

while i > 1 and A[[i/2]] > A[i] do
    Swap A[[i/2]] and A[i]
    i ← [i/2]

For the removal process, we do the following for the Heap Tree. We can remove any nodes from the Heap Tree. When we remove an element from the Heap Tree, we replace the last element in the Heap Tree (the element at i = n) with that element that was removed. We do this to complete the binary tree. Now, it is time to work on the Heap-order property of the Heap Tree. We do that by making sure that the key placed at the parent's node has a lower value than the key placed at its children's node.

The Algorithm for the removal process of the Heap Tree Remove Min is as follows:

**Algorithm** HeapRemoveMin():

***Input:*** None

***Output:*** An update of the array, $A$, of $n$ elements, for a heap, to remove and return an item with smallest key

$temp \leftarrow A[1]$
$A[1] \leftarrow A[n]$
$n \leftarrow n - 1$
$i \leftarrow 1$
**while** $i < n$ **do**
    **if** $2i + 1 \leq n$ **then** // this node has two internal children
        **if** $A[i] \leq A[2i]$ and $A[i] \leq A[2i + 1]$ **then**
            **return** $temp$ // we have restored the heap-order property
        **else**
            Let $j$ be the index of the smaller of $A[2i]$ and $A[2i + 1]$
            Swap $A[i]$ and $A[j]$
            $i \leftarrow j$
    **else** // this node has zero or one internal child
        **if** $2i \leq n$ **then** // this node has one internal child (the last node)
            **if** $A[i] > A[2i]$ **then**
                Swap $A[i]$ and $A[2i]$
        **return** $temp$ // we have restored the heap-order property
**return** $temp$ // we reached the last node or an external node

This was taken from Zybook. The Heap Tree holds its minimum key value at the top of the Heap Tree since the Heap Tree is structured from descending to ascending values. To remove the Heap Tree Minimum key value, you need to replace the value at the element n in the Heap Tree with the value you want to remove. This way your Heap Tree is a complete tree. Also, you have to account for the nodes' parents being always at lower values than its children's coming. The same analogy applies for any node in the Heap Tree as well.

Now, let's move on into the process of searching for all frequent flyers with the highest priority. In order to do that, we need to search through the whole Heap Tree since the Heap Tree stores values from descending to ascending order. This means that the search is O(log(n)). We have to do that for k highest flyers so the running time would be O(klog(n)).