Eric Tashji

Module 4 Priority Queues and Heaps Application Exercises

Problem 1: Exercise 5.7.29

Problem: (a) Suppose you work for a major airline and are given the job of writing the algorithm for processing upgrades into first class on various flights. Any frequent flyer can request an upgrade for his or her up-coming flight using this online system. Frequent flyers have different priorities, which are determined first by frequent flyer status (which can be, in order, silver, gold, platinum, and super) and then, if there are ties, by length of time in the waiting list. In addition, at any time prior to the flight, a frequent flyer can cancel his or her upgrade request (for instance, if he or she wants to take a different flight), using a confirmation code they got when he or she made his or her upgrade request. When it is time to determine upgrades for a flight that is about to depart, the gate agents inform the system of the number, **k**, of seats available in first class, and it needs to match those seats with the **k** highest-priority passengers on the waiting list. Describe a system that can process upgrade requests and cancellations in **O**(log(**n**)) time and can determine the **k** highest-priority flyers on the waiting list in **O**(**k** * log(**n**)) time, where **n** is the number of frequent flyers on the waiting list.

Solution: The following three algorithms will process upgrade requests, cancellations, and determining.

The first algorithm will handle upgrade requests (on next page):

Algorithm makeUpgradeRequest(**flight**, **upgradeRequest**):

> ***Input:*** The user's **flight**, which also has four parameters: silverQueue, goldQueue, platinumQueue, and superQueue, each of which tracks the upgrade requests that have already been made, based upon the frequent flyer status of those requests, and a new flyer, **upgradeRequest**, which is going to be added to one of the four queues above. (Note that the user cannot see who is in the queues, but the system will recall all of these queues for use within the algorithm). **upgradeRequest** will also have a frequent-flyer status parameter, called flyerStatus, and will be given a confirmation code, called confirmCode.
> ***Output:*** The algorithm will add **upgradeRequest** to one of **flight**'s four queues, depending on their frequent flyer status. It will also return the confirmation code to the user so that they can use their code later to cancel their upgrade request, if they choose to do so.
>
> **upgradeRequest**.confirmCode ← new randomly generated unique integer code
> **if upgradeRequest**.flyerStatus = "super" **do:**
> 	**flight**.superQueue.insert(**upgradeRequest**)
> **else if upgradeRequest**.flyerStatus = "platinum" **do:**
> 	**flight**.platinumQueue.insert(**upgradeRequest**)
> **else if upgradeRequest**.flyerStatus = "gold" **do:**

```
            flight.goldQueue.insert(upgradeRequest)
    else if upgradeRequest.flyerStatus = “silver” do:
            flight.goldQueue.insert(upgradeRequest)
    return upgradeRequest.confirmCode
```

Variables: makeUpgradeRequest() takes in two parameters as input. The first is the user’s **flight**, which includes four queues: silverQueue, goldQueue, platinumQueue, and superQueue, each one corresponding to a frequent-flyer status. The second parameter is an upgrade request, called **upgradeRequest**. It also has two other parameters: a frequent-flyer status, called flyerStatus, and a confirmation code, called confirmCode. Note that the user cannot see all the queues associated with the **flight,** but the system will be able to recall the queues associated with the user’s flight for use within the algorithm.

Explanation: The algorithm begins by assigning **upgradeRequest**’s confirmation code with a new randomly-generated integer code. It will then insert **upgradeRequest** into one of either silverQueue, goldQueue, platinumQueue, or superQueue, depending on **upgradeRequest**’s frequent-flyer status: superQueue if it is “super”, platinumQueue if it is “platinum”, goldQueue if it is “gold”, or silverQueue if it is “silver.” Finally, the algorithm will return **upgradeRequest**’s confirmation code to the user, so they can save it for later if they need to cancel their upgrade request.

Runtime: makeUpgradeRequest() has a runtime of **O**(**1**). This is because it has at most four steps, and there are no loops. Therefore, it has a constant runtime, or a runtime of **O**(**1**).

The second algorithm will handle request cancellations:

Algorithm makeCancellation(**flight, confirmCode**):

> ***Input***: **flight**, the user’s flight, which has a parameter with a balanced binary tree containing a list of cancellations that have already been made, called cancellations, and **confirmCode**, the user’s confirmation code to be entered into **flight**’s cancellations parameter. (Note that the user cannot see all the other cancellations, but the system will recall **flight**’s cancellation parameter for use with the rest of the algorithm).
> ***Output***: The algorithm doesn’t return anything, but will add **confirmCode** to **cancellations** for use in the next algorithm.

```
if flight.cancellations.root = null do:
        flight.cancellations.root ← confirmCode
        return
Let w ← flight.cancellations.root
while w != null do:
        if confirmCode > w do:
                w ← w.right
```

```
            else do:
                    w ← w.left
    w ← confirmCode
    rebalance(confirmCode, flight.cancellations)
    return
```

Variables: makeCancellations() takes two parameters as input: **flight**, the user's flight, which contains a parameter for a balanced binary tree, called cancellations, that will store the confirmation codes of all upgrade requests that have been cancelled, and **confirmCode**, the confirmation code of the new upgrade request that is to be canceled. Note that the user cannot see the elements in **flight**'s cancellations parameter, but the system will be able to store and recall the cancellations parameter, given the user's flight.

Explanation: The purpose of this algorithm is to insert the confirmation code of the upgrade request that the user wants canceled into a balanced binary tree, for use with a later algorithm. The algorithm begins by checking to see if cancellations is null. If so, it inserts **confirmCode** as cancellations' root. Otherwise, it will iterate down the list to find the correct location to insert **confirmCode**. It does so by checking to see if **confirmCode** is greater than the value of the current node. If so, it will check the current node's right node. Otherwise, it will check the current node's left node. Once the current node being checked becomes null, it will insert the node at that location. Then, to maintain balance, the rebalance method will be called on **flight**.cancellations, starting at **confirmCode**.

Runtime: This algorithm has a runtime of **O**(log(**n**)). This is because the process of initially inserting the new element into **flight**.cancellations has a worst-case runtime of **O**(log(**n**)), as the runtime to search a binary tree for the correct location for insertion is **O**(log(**n**)). Furthermore, the rebalance method also has a worst-case runtime of **O**(log(**n**)). This results in an algorithm with a worst-case runtime of **O**(**2** * log(**n**)), which can be simplified to **O**(log(**n**)).

The final algorithm will determine which upgrade requests get filled, based on how many seats are available in first class (k) (on next page).

Eric Tashji

Algorithm fillRequests(**k**, **flight**):

Input: **k**, the number of first-class seats available, and **flight**, the flight that is having its requests fulfilled. **flight** also has several parameters associated with it, including the following four queues: silverQueue, goldQueue, platinumQueue, and superQueue, which store the queues of passengers requesting first-class seats, based upon their frequent-flyer status. **flight** has an additional parameter, called cancellations, which is a balanced binary tree containing all of the cancellation requests that users had made.
Output: **firstClass**, a list containing all of the passengers that will be flying first-class on the given flight.

```
i ← 0
firstClass ← empty list
canceled ← false
while i < k do:
        if flight.superQueue.peak() != null do:
                flyer ← flight.superQueue.dequeue()
        else if flight.platinumQueue.peak() != null do:
                flyer ← flight.platinumQueue.dequeue()
        else if flight.goldQueue.peak() != null do:
                flyer ← flight.goldQueue.dequeue()
        else if flight.silverQueue.peak() != null do:
                flyer ← flight.silverQueue.dequeue()
        else do:
                return firstClass
        w ← flight.cancellations.root
        while w != null do:
                if flyer.confirmCode = w do:
                        canceled ← true
                        break
                else if flyer.confirmCode > w do:
                        w ← w.right
                else do:
                        w ← w.left
        if canceled ← false do:
                firstClass.add(flyer)
                i ← i + 1
        else do:
                canceled ← false
return firstClass
```

(Explanation on next page)

Eric Tashji

Variables: fillRequests() takes two variables as input: **k**, the number of first-class seats available, and **flight**, the current flight. **flight** also has five parameters associated with it: the four queues: superQueue, platinumQueue, goldQueue, and silverQueue, which store the requests of passengers for first-class seats, based on their frequent-flyer status. **flight**'s additional parameter is cancellations, which is a balanced binary tree containing all of the first-class seat requests that have been canceled by the customers. fillRequests() also establishes four other variables: **i**, a counter to determine when we have reached **k** seats, **firstClass**, the list of passengers being placed in first-class seating, **flyer**, the current flight request in question, **canceled**, a boolean used to determine whether or not the passenger in question canceled their first-class seating request, and **w**, the current node in the **flight**.cancellations balanced binary tree. **w** will be compared with **flyer** to determine whether or not their first-class flight request has been canceled.

Explanation: fillRequests() begins by entering into a while loop that will run until all **k** seats are filled. Within the while loop, the four queues of passenger requests will be checked to determine if there is a flyer making a first-class seating request within each. Note that they are checked in order of first-class flyer status: superQueue, then platinumQueue, then goldQueue, then silverQueue. This ensures that the frequent-flyer with the highest status gets granted a first-class seat before those with a lower status. Furthermore, a queue is being used because it will prioritize those that applied for first-class seating earlier over those that applied later. The chosen queue will then be dequeued, and the dequeued element will be saved into the **flyer** variable.

The confirmation code of the **flyer** variable will then be checked against **w** as it iterates through the **flight**.cancellations balanced binary tree. If **flyer**'s confirmation code is equal to **w**, then **canceled** will be set to true and the algorithm will break the while loop, which will stop iteration. Otherwise, **w** will move to its right node if **flyer**'s confirmation code is greater than **w**, or **w** will move to the left if **flyer**'s confirmation code is less than **w**. The iteration will continue until **w** is null, indicating that **flyer**'s confirmation code is not in the list. If that is the case, then we know that the **flyer** has not canceled his first-class upgrade request. This means that we can then add **flyer** to **firstClass** and increment **i** because we have filled one seat of **k**. If the request was canceled, the algorithm will not add **flyer** to the **firstClass** list or increment **i** because it was cancelled, and **flyer** no longer wants a first-class seat. This process will then repeat until all seats of **k** have been filled.

Runtime: This algorithm has a runtime of **O**(**k** * log(**n**)). It has a runtime of **k** because the algorithm will run until all **k** seats are filled. Furthermore, it has a runtime of **O**(log(**n**) because everytime a flyer is inspected to determine if he can be placed on the plane, the algorithm must determine if the flyer has already canceled his flight. This requires a search of the balanced binary tree **flight**.cancellations to determine whether or not the passenger had canceled their flight. This search has a runtime of **O**(log(**n**)). Therefore, it has a runtime of **O**(**k** * log(**n**)).

(Notes on the program on next page.)

Eric Tashji

Notes on the program: This program is written in Java. It contains two classes: a Flight class and a Main class. The Flight class contains all operations involving the 4 queues and the balanced binary tree for cancellations. The Main class contains a menu for easily navigating the different options of “adding,” “removing,” or “filling” the first-class seats. Note that the cancellation method will not cancel an upgrade request if the code is entered incorrectly, and it will not inform the user if their code was entered incorrectly as long as the code was six digits. Also, the program will quit once the “filled” method is called because all the queues will be emptied by the program. Finally, while the program may not be exactly the same as the pseudocode, it resembles it as closely as possible within the Java language.