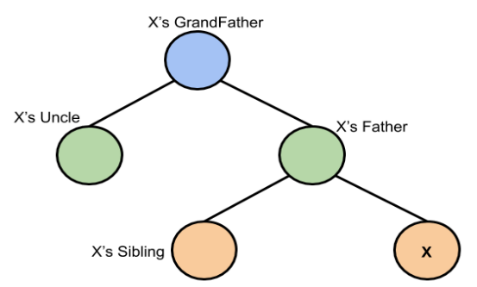
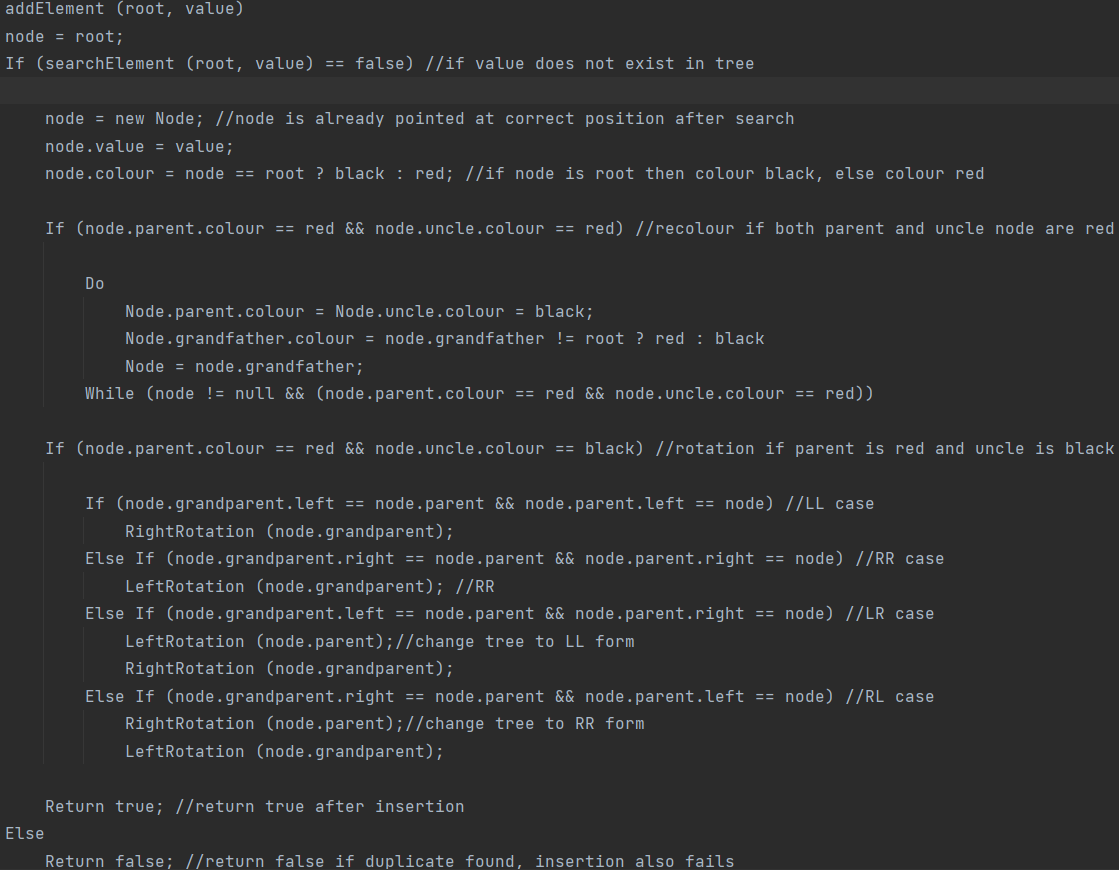
(c)

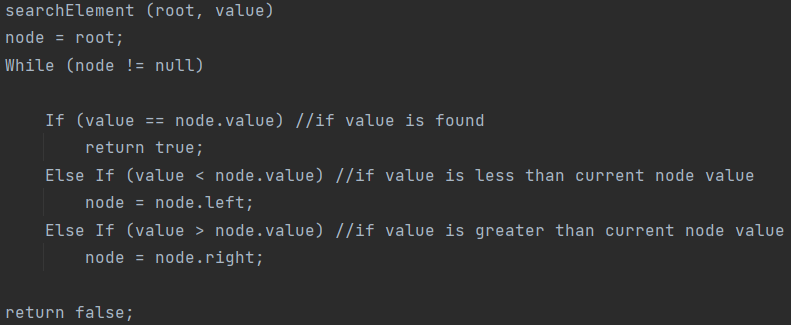
For upgrade requests and cancellation to be in O(log n) time and determine the k highest-priority flyers on the waiting list in O(k log n) time, where n is the number of frequent flyers on the waiting list, the system must traverse nodes to a depth of at most O(log n). For this to be possible, we require the use of a self-balancing binary tree which has a depth of O(log n). Two of the most popular self-balancing binary trees are AVL tree and Red-Black tree. In this instance, we decided to use a Red-Black tree. This is because of program will experience frequent insertion and deletion. Although AVL trees are more balanced, they cause more rotations during insertion and deletion. So, a Red-Black tree is preferred in this case.

**Rules That Every Red-Black Tree Follows:**

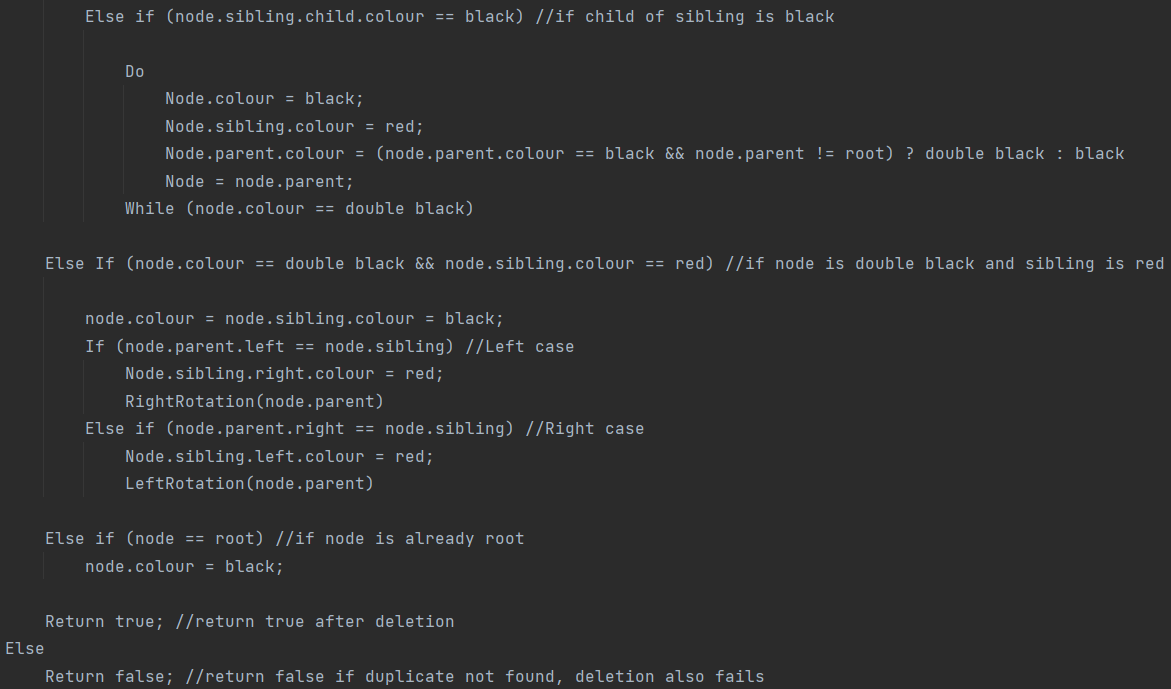
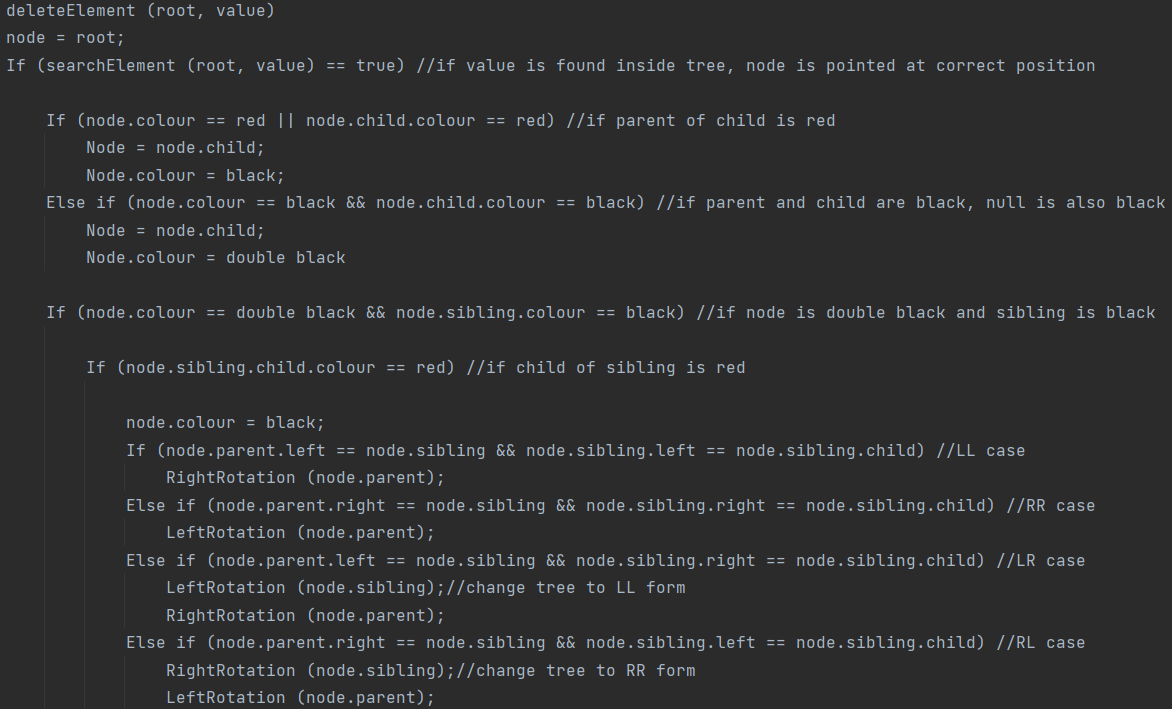
1. Every node has a color either red or black.
2. The root of the tree is always black.
3. There are no two adjacent red nodes (A red node cannot have a red parent or red child).
4. Every path from a node (including root) to any of its descendants’ NULL nodes has the same number of black nodes.
5. All leaf nodes are black nodes.

**Interesting points about Red-Black Tree:**

1. Black height of the red-black tree is the number of black nodes on a path from the root node to a leaf node. Leaf nodes are also counted as black nodes. So, a red-black tree of height h has black height >= h/2.
2. Height of a red-black tree with n nodes is h<= 2 log2(n + 1).
3. All leaves (NIL) are black.
4. The black depth of a node is defined as the number of black nodes from the root to that node.



This is the pseudocode for the default search element and add element methods of a red-black tree.



This is the pseudocode for the default delete element method of a red-black tree.

For a program using a red-black tree, upgrade and cancellations thus will take O(log n) time. This is because the program will have to traverse the tree to find the correct position and insert or delete a node. This process takes O(log n) time because the red-black tree has a depth of O(log n). To determine the *k* highest-priority flyers on the waiting list, the process takes O(k log n) time. This is because the red-black tree has a depth of O(log n) and the program has to traverse the red-black tree k time. So, O(k \* log n) = O(k log n). A Java program has been created below using TreeSet which is implemented using a Red-Black Tree. The comparator of the program is designed to move the highest priority passengers to the left of the tree. To get the *k* highest-priority flyers on the waiting list, we get the leftmost passenger in the tree first.

Text

Description automatically generated

This is the public class for our program named FrequentFlyerProgram. The constructor initializes a new TreeSet using the Passenger comparator. TreeSet uses Red-Black Tree in its implementation. The referenceID is also set to 0.

Text

Description automatically generated

This is the Passenger class and it holds information about the passenger.

Passenger() – Used to initialize the comparator in the TreeSet

The constructor with the 6 parameters creates a new Passenger object with the given parameters which is saved inside the TreeSet.

Text

Description automatically generated

This is the comparator for the Passenger class. It is used to compare 2 passenger objects inside the TreeSet. It is used in the search, insert and delete functions of the TreeSet. First, we compare rank, higher rank goes left. If ranks are equal, we compare time, higher time goes left. If rank and time are equal, we compare ref number, lower ref number goes left.

Text

Description automatically generated

This is the getPassengerList method which returns k passengers with the highest priority.

This method returns the current referenceID and increments it by 1 so every passenger has a unique ID. This is to allow passengers with the same rank and waiting time to exist together in the Red-Black Tree.

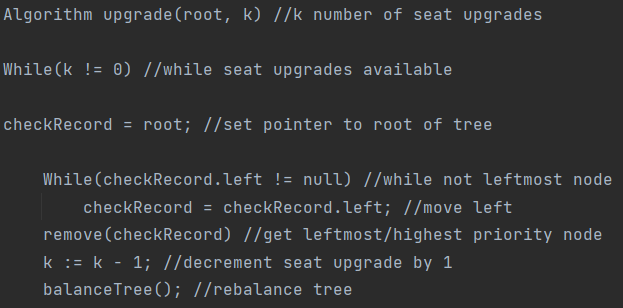
Text

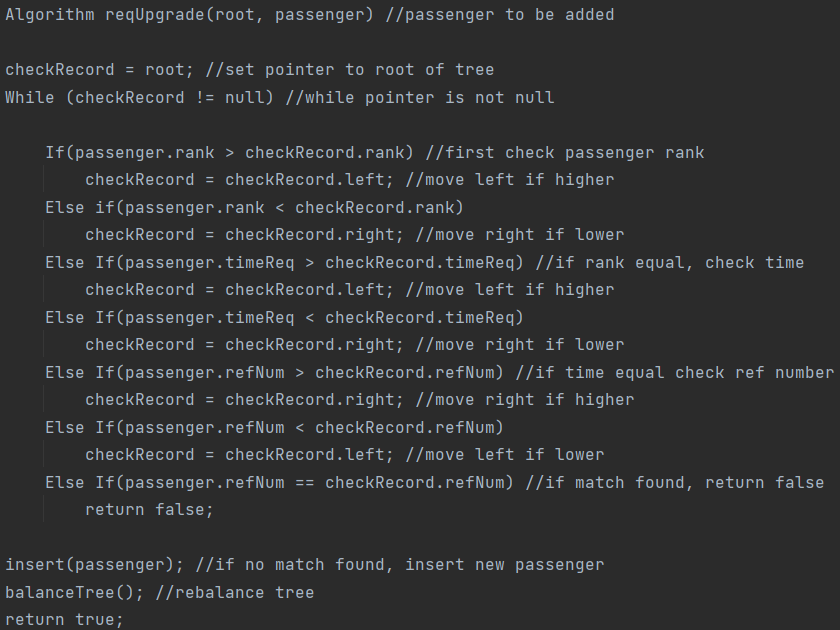
Description automatically generated

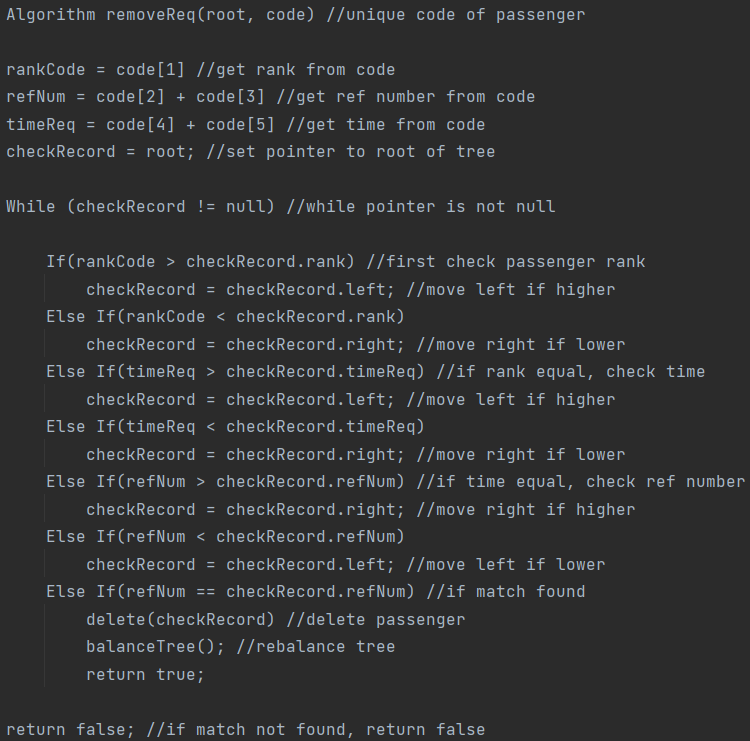
This is the addPassenger method which creates new Passenger object and inserts it into the TreeSet. A unique comfirmation code is also generated for each passenger. As an example, S20110 means the passenger has platinum rank, referenceID of 01 and waitingTime of 10. The new passenger is only inserted if no duplicates exist in the TreeSet.

Text

Description automatically generated

This is the removePassenger method which removes a passenger object from the TreeSet. It converts the confirmationCode of the passenger into rank, referenceID and waitingTime which is used to create a temporary passenger object. The passenger is deleted from the TreeSet if a duplicate is found.





These are the pseudocode for the upgrade, cancellation and get k highest priority flyer methods of the program.