COMP2038 Group Report

26th December 2021

Name	**Student ID**
Leong Chang Yung	20307078
Chong Hao Wei	20194465
Tan Zhun Xian	20313854
Morhaf Allababidi	20195867
Chan Yu Xuan	20283052

1. Questions on Big-O Analysis. (25 marks)

(a) Consider a polynomial function of order k given by $f(n) = a_k n^k + a_{k-1} n^{k-1} + \cdots + a_0$. Formally demonstrate that $f(n) \in O(n^k)$. Full marks for using basic definitions and concepts.

[12 marks]

Big-O Rules: Drop smaller terms

If f(n) = (1 + h(n)) with h(n) -> 0 as n -> ∞. Then, f(n) is O (1). This is because the O(h(n)) is going to be insignificant as n grows bigger.

Big-O Multiplication Rules

Suppose two equations $f_1(n)$ is $O(g_1(n))$ and $f_2(n)$ is $O(g_2(n))$. From the definition, there exists positive constants c_1, c_2, n_1 and n_2 such that $f_1(n) \leq c_1 g_1(n)$ for all $n \geq n_1$ and $f_2(n) \leq c_2 g_2(n)$ for all $n \geq n_2$. Let $n_0 = \max\{n_1, n_2\}$. Multiplying $f_1(n)$ and $f_2(n)$ gives $f_1(n)\, f_2(n) \leq c_1 c_2 g_1(n) g_2(n)$ for all $n \geq n_0$. So, $f_1(n) f_2(n)$ is $O(g_1(n) g_2(n))$.

Big-O Conventions

Use the smallest (slowest growing) 'reasonable' possible class of functions. As an example, 2n is O(n) instead of $O(n^2)$

Use the simplest expression of the class. As an example, 3n + 5 is O(n) instead of O(3n)

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

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

As n -> ∞, $(a_k + a_{k-1}/n + \cdots + a_0/n^k)$ -> $(a_k + 0 + \cdots + 0)$ -> a_k and $f(n)$ -> $n^k a_k$ by Drop smaller terms.

$O(n^k)$ is trivially $O(n^k)$ and $O(a_k)$ is O(1).

With Big-O Multiplication Rules, $O(f(n)) = O(n^k a_k) = O(n^k)\, O(a_k) = O(n^k) O(1) = O(n^k * 1) = O(n^k)$

So, $f(n)$ is $O(n^k)$ or $f(n) \in O(n^k)$.

(b) Refer to the definition of Big-O in the lecture materials. In particular, the condition for which one can state that $f(n)$ is $O(g(n))$ is defined. Briefly explain why the notation $f(n) \in O(g(n))$ is preferred compared to $f(n) = O(g(n))$. Full marks for using basic definitions and concepts and mathematical formulation.

[13 marks]

$f(n)$ is $O(g(n))$ if and only if there exists positive constants c and n_0 such that $f(n) \leq c g(n)$ for all $n \geq n_0$. In mathematical notation, $\exists c > 0, n_0$ such that $\forall n \geq n_0$, $f(n) \leq c g(n)$.

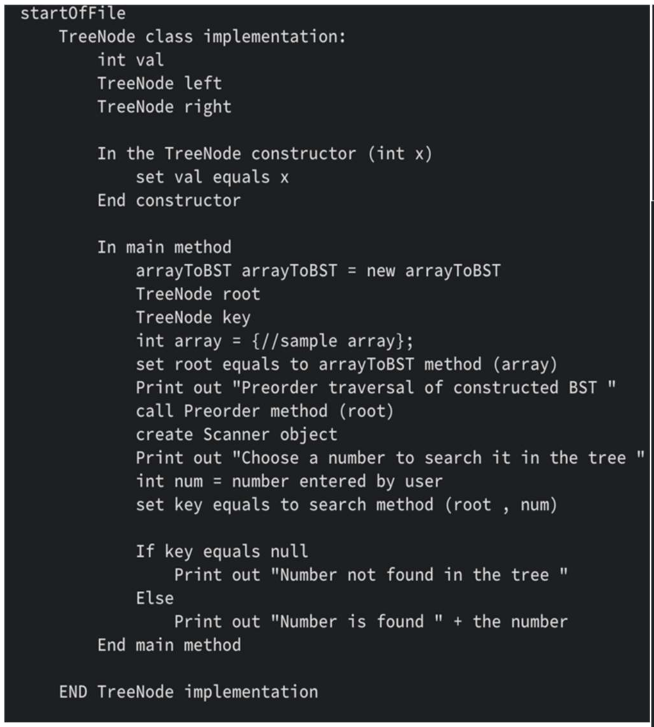
$f: N^+ \rightarrow R^+$ and $g: N^+ \rightarrow R^+$ where $N^+ = \{1,2,3, \ldots\}$ and $R^+ = \{x \in R \mid x \geq 0\}$ assuming that $f(n) \geq 0, \forall n \geq 1$. For convenience, the function is sometimes relaxed to $f(n) \geq 0, \forall n \geq N$ for some constant N.

O(n) can be thought of as the set of all functions whose growth is no worse than linear for sufficiently large n. Hence, it can be thought of as the infinite set {1, 2, …, log n, 2 log n, …, n, 2n, 3n, …, n+1, n+2, …}. So, 3n+5 is O(n) is just the statement that 3n+5 is in this set or $3n+5 \in O(n)$.

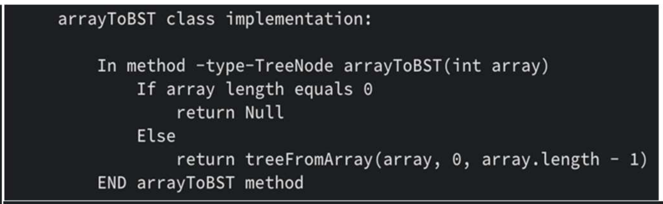
We should avoid writing big-O notations in the form $f(n) = O(g(n))$. As an example, if $f(n)$ is O(n), then f(n) is also $O(n^2)$ because if f(n) grows no worse than linear for sufficiently large n, then f(n) must also grow no worse than quadratic for sufficiently large n. If we write $f(n) = O(n)$ and $f(n) = O(n^2)$. Then this implies that $O(n) = O(n^2)$ which is incorrect as $O(n) \subset O(n^2)$. Instead, we should write $f(n) \in O(n)$ and $f(n) \in O(n^2)$.

2. Questions on Binary Search Tree, Heap, Balanced Binary Search Tree, Basic Data Structures (stack/queue)

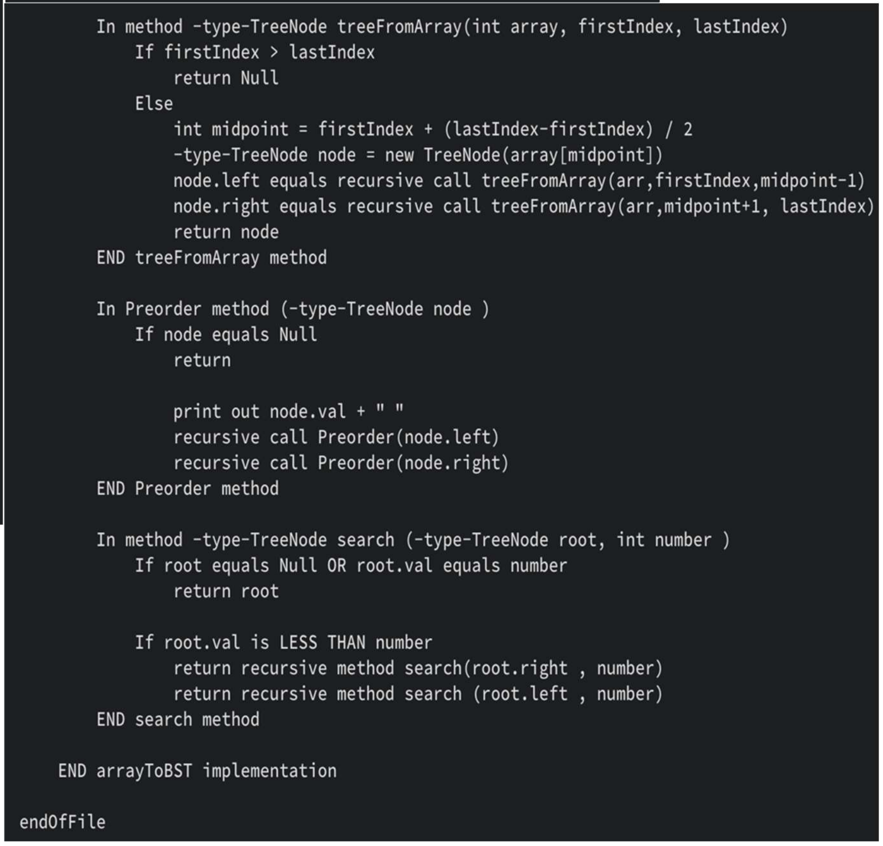
(a)



```
startOfFile
    TreeNode class implementation:
        int val
        TreeNode left
        TreeNode right

        In the TreeNode constructor (int x)
            set val equals x
        End constructor

        In main method
            arrayToBST arrayToBST = new arrayToBST
            TreeNode root
            TreeNode key
            int array = {//sample array};
            set root equals to arrayToBST method (array)
            Print out "Preorder traversal of constructed BST "
            call Preorder method (root)
            create Scanner object
            Print out "Choose a number to search it in the tree "
            int num = number entered by user
            set key equals to search method (root , num)

            If key equals null
                Print out "Number not found in the tree "
            Else
                Print out "Number is found " + the number
        End main method

    END TreeNode implementation
```



```
    arrayToBST class implementation:

        In method -type-TreeNode arrayToBST(int array)
            If array length equals 0
                return Null
            Else
                return treeFromArray(array, 0, array.length - 1)
        END arrayToBST method
```



```
        In method -type-TreeNode treeFromArray(int array, firstIndex, lastIndex)
            If firstIndex > lastIndex
                return Null
            Else
                int midpoint = firstIndex + (lastIndex-firstIndex) / 2
                -type-TreeNode node = new TreeNode(array[midpoint])
                node.left equals recursive call treeFromArray(arr,firstIndex,midpoint-1)
                node.right equals recursive call treeFromArray(arr,midpoint+1, lastIndex)
                return node
        END treeFromArray method

        In Preorder method (-type-TreeNode node )
            If node equals Null
                return

                print out node.val + " "
                recursive call Preorder(node.left)
                recursive call Preorder(node.right)
        END Preorder method

        In method -type-TreeNode search (-type-TreeNode root, int number )
            If root equals Null OR root.val equals number
                return root

            If root.val is LESS THAN number
                return recursive method search(root.right , number)
                return recursive method search (root.left , number)
        END search method

    END arrayToBST implementation

endOfFile
```

The pseudocode above represents the following steps to convert a sorted array to a Binary Search Tree (BST):

1. Get the middle element of the array and make it as the root (this splits the array into left and right halves)
2. Get the middle Element of the left half of the array and make it child of root
3. Get the middle Element of the right half of the array and make it child of root
4. Recursively repeat 2-3 (inserting child elements)

By successfully constructing a Binary Search Tree from a sorted array, We successfully get O(log n) as time complexity when searching up any element in the tree. That is because every time we search in the tree, we are halving the number of inputs in each search iteration (recursion).

(b)

The conditions given can be satisfied by using an AVL Search Tree structure for the software. When doing insertion and Deletion, the process time for an AVL Search Tree will always be O(log n), the procedure for Insertion such as follows:

1. Search through the tree for empty node (Search right node if current node alphabetical order is smaller than insert value alphabetical order, else search left node)
2. When empty node is found, insert the new value
3. Check the balance factor of every single node
4. Case 1: If Balance factor is >=-1 and <=1, the insert operation ends
5. Case 2: If Balance factor doesn't satisfy the condition in Case 1, rebalance the tree using rotation

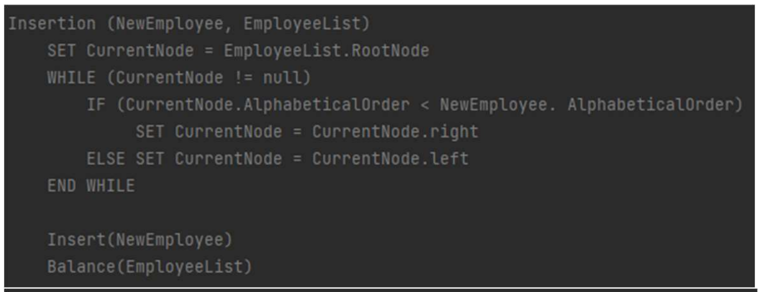
The procedure for deletion is similar to Insertion, which is as follows:

1. Search through the tree for the value to be deleted
2. When the value to be deleted is found, set the node to null and replace with suitable node
3. Check the balance factor of every single node
4. Case 1: If Balance factor is >=-1 and <=1, the insert operation ends
5. Case 2: If Balance factor doesn't satisfy the condition in Case 1, rebalance the tree using rotation

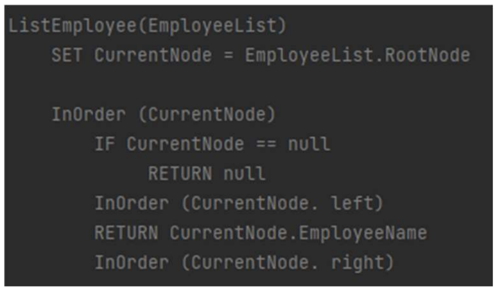
Since the AVL tree for employee names is constructed in alphabetical order, the software can finish the request in O(n) time.

Lastly, the promises can be declared as a global variable which connect to each node so the software is able to process it in O(1) time and everyone will be getting the same value.

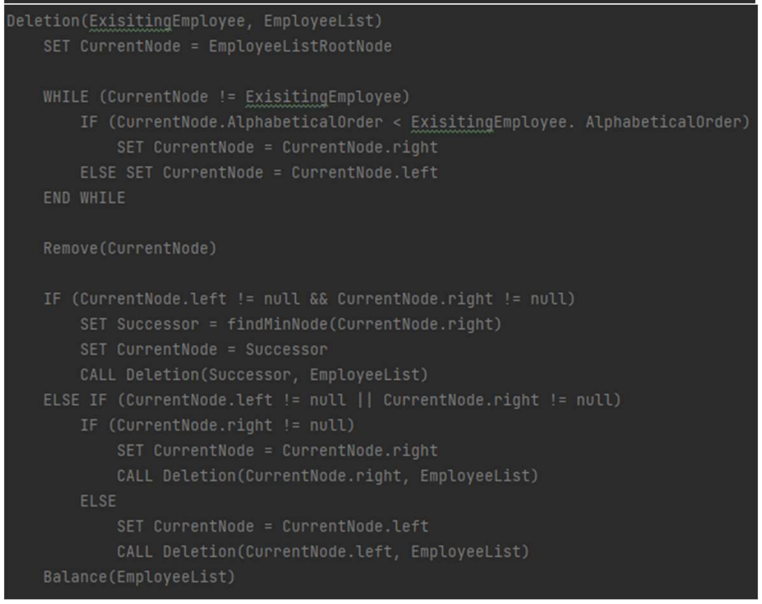
Pseudocode:



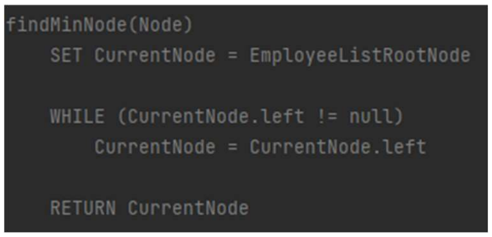
```
Insertion (NewEmployee, EmployeeList)
    SET CurrentNode = EmployeeList.RootNode
    WHILE (CurrentNode != null)
        IF (CurrentNode.AlphabeticalOrder < NewEmployee. AlphabeticalOrder)
             SET CurrentNode = CurrentNode.right
        ELSE SET CurrentNode = CurrentNode.left
    END WHILE

    Insert(NewEmployee)
    Balance(EmployeeList)
```



```
ListEmployee(EmployeeList)
    SET CurrentNode = EmployeeList.RootNode

    InOrder (CurrentNode)
        IF CurrentNode == null
             RETURN null
        InOrder (CurrentNode. left)
        RETURN CurrentNode.EmployeeName
        InOrder (CurrentNode. right)
```



```
Deletion(ExisitingEmployee, EmployeeList)
    SET CurrentNode = EmployeeListRootNode

    WHILE (CurrentNode != ExisitingEmployee)
        IF (CurrentNode.AlphabeticalOrder < ExisitingEmployee. AlphabeticalOrder)
            SET CurrentNode = CurrentNode.right
        ELSE SET CurrentNode = CurrentNode.left
    END WHILE

    Remove(CurrentNode)

    IF (CurrentNode.left != null && CurrentNode.right != null)
        SET Successor = findMinNode(CurrentNode.right)
        SET CurrentNode = Successor
        CALL Deletion(Successor, EmployeeList)
    ELSE IF (CurrentNode.left != null || CurrentNode.right != null)
        IF (CurrentNode.right != null)
            SET CurrentNode = CurrentNode.right
            CALL Deletion(CurrentNode.right, EmployeeList)
        ELSE
            SET CurrentNode = CurrentNode.left
            CALL Deletion(CurrentNode.left, EmployeeList)
    Balance(EmployeeList)
```



```
findMinNode(Node)
    SET CurrentNode = EmployeeListRootNode

    WHILE (CurrentNode.left != null)
        CurrentNode = CurrentNode.left

    RETURN CurrentNode
```

(c)

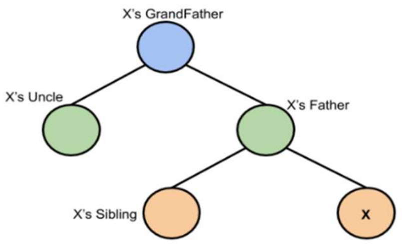
For upgrade requests and cancellation to be in O(log n) time and determine the k highest-priority flyers on the waiting list in O(k log n) time, where n is the number of frequent flyers on the waiting list, the system must traverse nodes to a depth of at most O(log n). For this to be possible, we require 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.

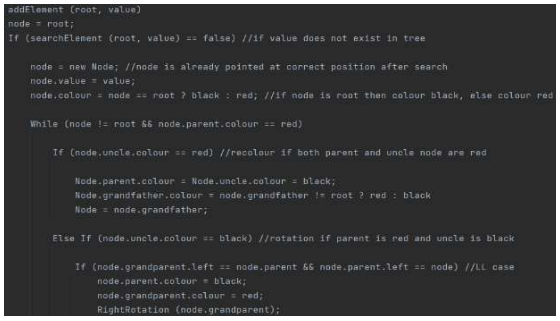
Rules That Every Red-Black Tree Follows:

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.

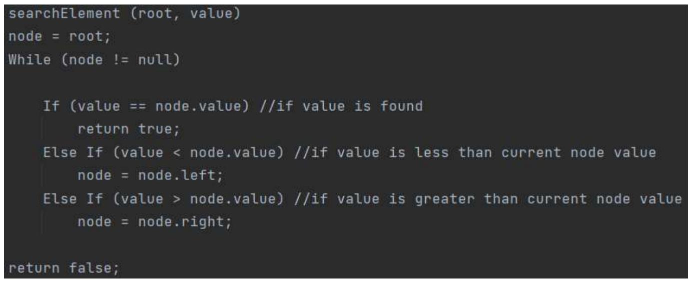
Interesting points about Red-Black Tree:

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 $\log_2(n + 1)$.
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.

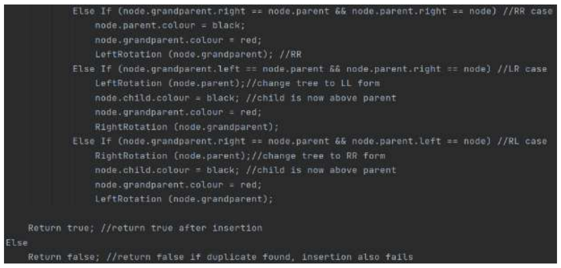




```
addElement (root, value)
node = root;
If (searchElement (root, value) == false) //if value does not exist in tree

    node = new Node; //node is already pointed at correct position after search
    node.value = value;
    node.colour = node == root ? black : red; //if node is root then colour black, else colour red

    While (node != root && node.parent.colour == red)

        If (node.uncle.colour == red) //recolour if both parent and uncle node are red

            Node.parent.colour = Node.uncle.colour = black;
            Node.grandfather.colour = node.grandfather != root ? red : black
            Node = node.grandfather;

        Else If (node.uncle.colour == black) //rotation if parent is red and uncle is black

            If (node.grandparent.left == node.parent && node.parent.left == node) //LL case
                node.parent.colour = black;
                node.grandparent.colour = red;
                RightRotation (node.grandparent);
```

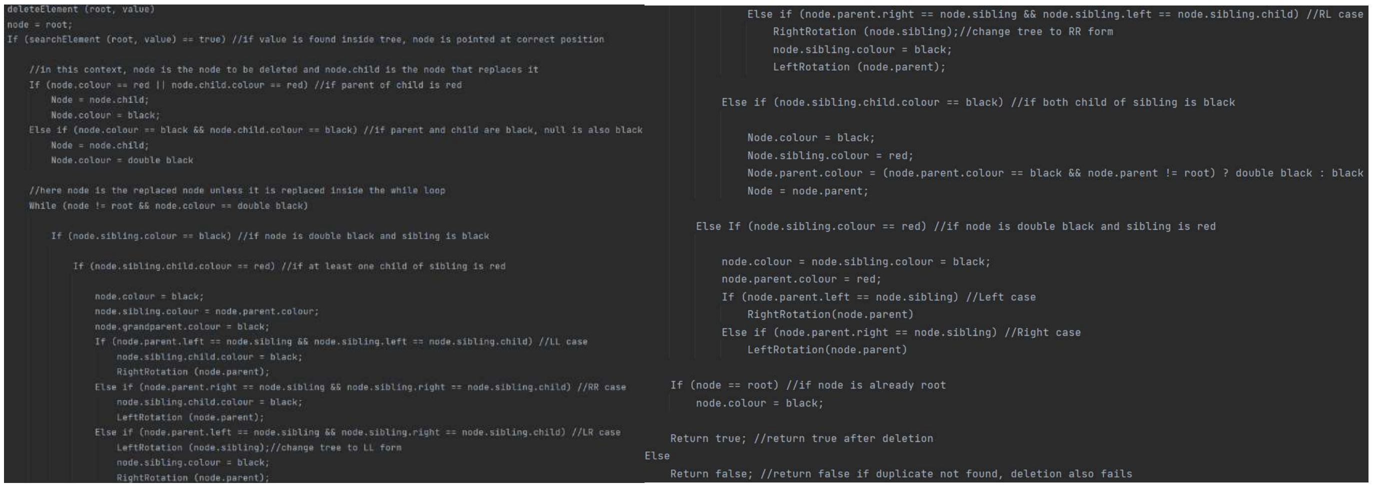


```
searchElement (root, value)
node = root;
While (node != null)

    If (value == node.value) //if value is found
        return true;
    Else If (value < node.value) //if value is less than current node value
        node = node.left;
    Else If (value > node.value) //if value is greater than current node value
        node = node.right;

return false;
```



```
            Else If (node.grandparent.right == node.parent && node.parent.right == node) //RR case
                node.parent.colour = black;
                node.grandparent.colour = red;
                LeftRotation (node.grandparent); //RR
            Else If (node.grandparent.left == node.parent && node.parent.right == node) //LR case
                LeftRotation (node.parent);//change tree to LL form
                node.child.colour = black; //child is now above parent
                node.grandparent.colour = red;
                RightRotation (node.grandparent);
            Else If (node.grandparent.right == node.parent && node.parent.left == node) //RL case
                RightRotation (node.parent);//change tree to RR form
                node.child.colour = black; //child is now above parent
                node.grandparent.colour = red;
                LeftRotation (node.grandparent);

    Return true; //return true after insertion
Else
    Return false; //return false if duplicate found, insertion also fails
```

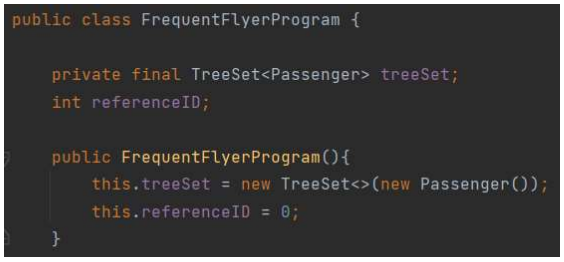
This is the pseudocode for the default search element and add element methods of a red-black tree.



```
deleteElement (root, value)
node = root;
If (searchElement (root, value) == true) //if value is found inside tree, node is pointed at correct position

    //in this context, node is the node to be deleted and node.child is the node that replaces it
    If (node.colour == red || node.child.colour == red) //if parent of child is red
        Node = node.child;
        Node.colour = black;
    Else if (node.colour == black && node.child.colour == black) //if parent and child are black, null is also black
        Node = node.child;
        Node.colour = double black

    //here node is the replaced node unless it is replaced inside the while loop
    While (node != root && node.colour == double black)

        If (node.sibling.colour == black) //if node is double black and sibling is black

            If (node.sibling.child.colour == red) //if at least one child of sibling is red

                node.colour = black;
                node.sibling.colour = node.parent.colour;
                node.grandparent.colour = black;
                If (node.parent.left == node.sibling && node.sibling.left == node.sibling.child) //LL case
                    node.sibling.child.colour = black;
                    RightRotation (node.parent);
                Else if (node.parent.right == node.sibling && node.sibling.right == node.sibling.child) //RR case
                    node.sibling.child.colour = black;
                    LeftRotation (node.parent);
                Else if (node.parent.left == node.sibling && node.sibling.right == node.sibling.child) //LR case
                    LeftRotation (node.sibling);//change tree to LL form
                    node.sibling.colour = black;
                    RightRotation (node.parent);
                Else if (node.parent.right == node.sibling && node.sibling.left == node.sibling.child) //RL case
                    RightRotation (node.sibling);//change tree to RR form
                    node.sibling.colour = black;
                    LeftRotation (node.parent);

            Else if (node.sibling.child.colour == black) //if both child of sibling is black

                Node.colour = black;
                Node.sibling.colour = red;
                Node.parent.colour = (node.parent.colour == black && node.parent != root) ? double black : black
                Node = node.parent;

        Else If (node.sibling.colour == red) //if node is double black and sibling is red

            node.colour = node.sibling.colour = black;
            node.parent.colour = red;
            If (node.parent.left == node.sibling) //Left case
                RightRotation(node.parent)
            Else if (node.parent.right == node.sibling) //Right case
                LeftRotation(node.parent)

    If (node == root) //if node is already root
        node.colour = black;

    Return true; //return true after deletion
Else
    Return false; //return false if duplicate not found, deletion also fails
```

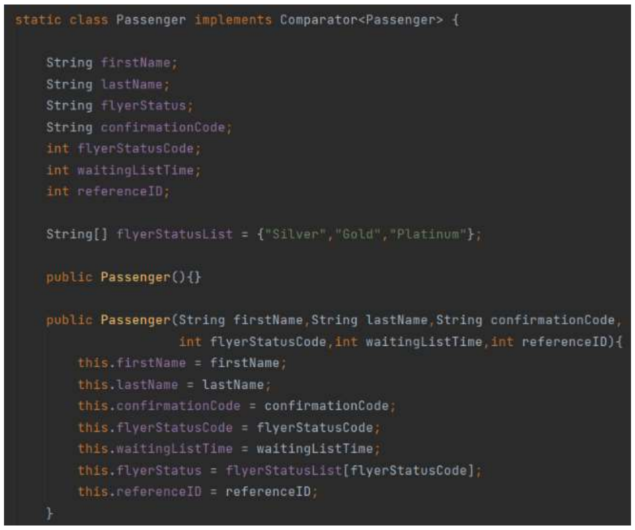
This is the pseudocode for the default delete element method of a red-black tree.

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.



```java
public class FrequentFlyerProgram {

    private final TreeSet<Passenger> treeSet;
    int referenceID;

    public FrequentFlyerProgram(){
        this.treeSet = new TreeSet<>(new Passenger());
        this.referenceID = 0;
    }
```

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.

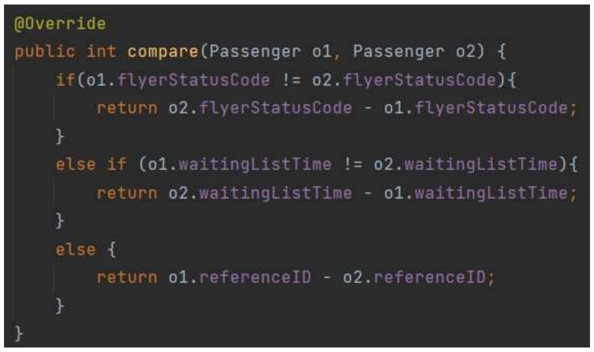


```java
static class Passenger implements Comparator<Passenger> {

    String firstName;
    String lastName;
    String flyerStatus;
    String confirmationCode;
    int flyerStatusCode;
    int waitingListTime;
    int referenceID;

    String[] flyerStatusList = {"Silver","Gold","Platinum"};

    public Passenger(){}

    public Passenger(String firstName,String lastName,String confirmationCode,
                     int flyerStatusCode,int waitingListTime,int referenceID){
        this.firstName = firstName;
        this.lastName = lastName;
        this.confirmationCode = confirmationCode;
        this.flyerStatusCode = flyerStatusCode;
        this.waitingListTime = waitingListTime;
        this.flyerStatus = flyerStatusList[flyerStatusCode];
        this.referenceID = referenceID;
    }
```

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

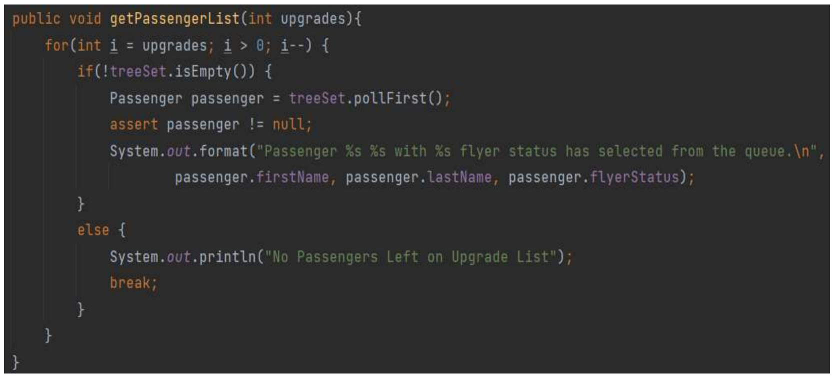
Passenger() – Used to initialize the comparator in the TreeSet

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



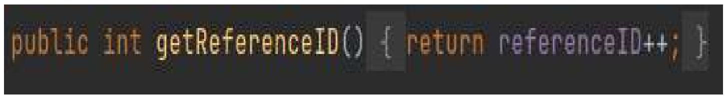
```java
@Override
public int compare(Passenger o1, Passenger o2) {
    if(o1.flyerStatusCode != o2.flyerStatusCode){
        return o2.flyerStatusCode - o1.flyerStatusCode;
    }
    else if (o1.waitingListTime != o2.waitingListTime){
        return o2.waitingListTime - o1.waitingListTime;
    }
    else {
        return o1.referenceID - o2.referenceID;
    }
}
```

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.



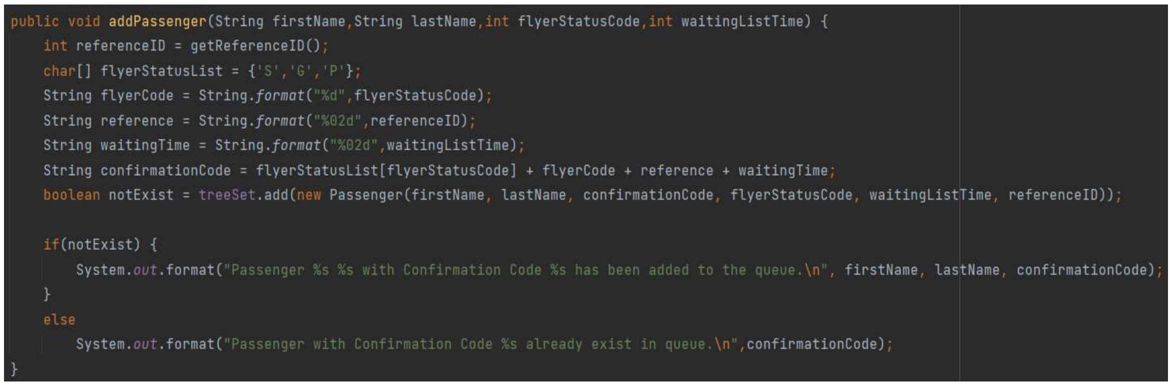
```java
public void getPassengerList(int upgrades){
    for(int i = upgrades; i > 0; i--) {
        if(!treeSet.isEmpty()) {
            Passenger passenger = treeSet.pollFirst();
            assert passenger != null;
            System.out.format("Passenger %s %s with %s flyer status has selected from the queue.\n",
                    passenger.firstName, passenger.lastName, passenger.flyerStatus);
        }
        else {
            System.out.println("No Passengers Left on Upgrade List");
            break;
        }
    }
}
```

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



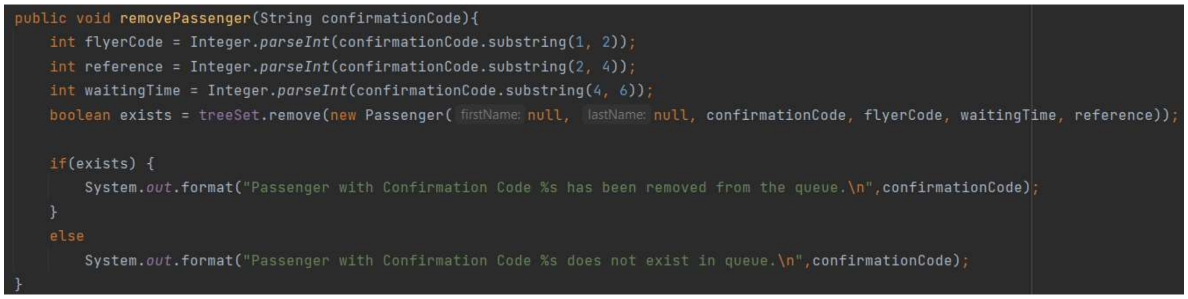
```java
public int getReferenceID() { return referenceID++; }
```

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.



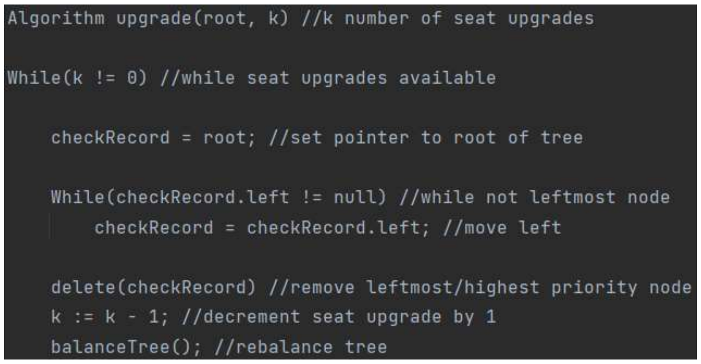
```java
public void addPassenger(String firstName,String lastName,int flyerStatusCode,int waitingListTime) {
    int referenceID = getReferenceID();
    char[] flyerStatusList = {'S','G','P'};
    String flyerCode = String.format("%d",flyerStatusCode);
    String reference = String.format("%02d",referenceID);
    String waitingTime = String.format("%02d",waitingListTime);
    String confirmationCode = flyerStatusList[flyerStatusCode] + flyerCode + reference + waitingTime;
    boolean notExist = treeSet.add(new Passenger(firstName, lastName, confirmationCode, flyerStatusCode, waitingListTime, referenceID));

    if(notExist) {
        System.out.format("Passenger %s %s with Confirmation Code %s has been added to the queue.\n", firstName, lastName, confirmationCode);
    }
    else
        System.out.format("Passenger with Confirmation Code %s already exist in queue.\n",confirmationCode);
}
```

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 Silver rank, referenceID of 01 and waitingTime of 10. The new passenger is only inserted if no duplicates exist in the TreeSet.

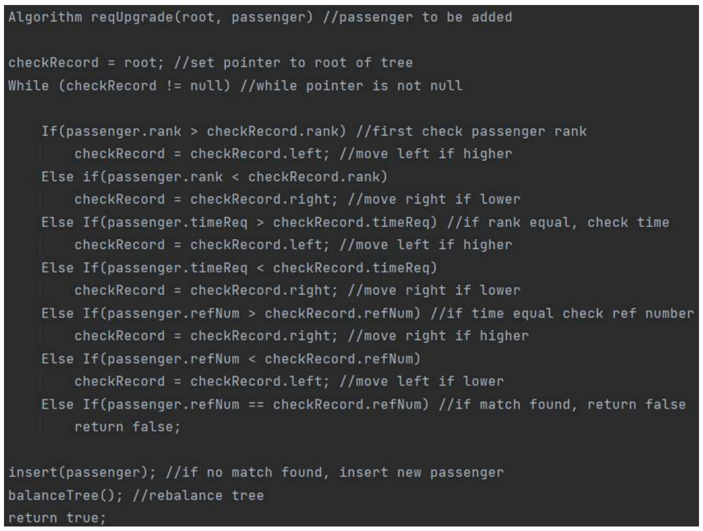


```
public void removePassenger(String confirmationCode){
    int flyerCode = Integer.parseInt(confirmationCode.substring(1, 2));
    int reference = Integer.parseInt(confirmationCode.substring(2, 4));
    int waitingTime = Integer.parseInt(confirmationCode.substring(4, 6));
    boolean exists = treeSet.remove(new Passenger( firstName: null,  lastName: null, confirmationCode, flyerCode, waitingTime, reference));

    if(exists) {
        System.out.format("Passenger with Confirmation Code %s has been removed from the queue.\n",confirmationCode);
    }
    else
        System.out.format("Passenger with Confirmation Code %s does not exist in queue.\n",confirmationCode);
}
```

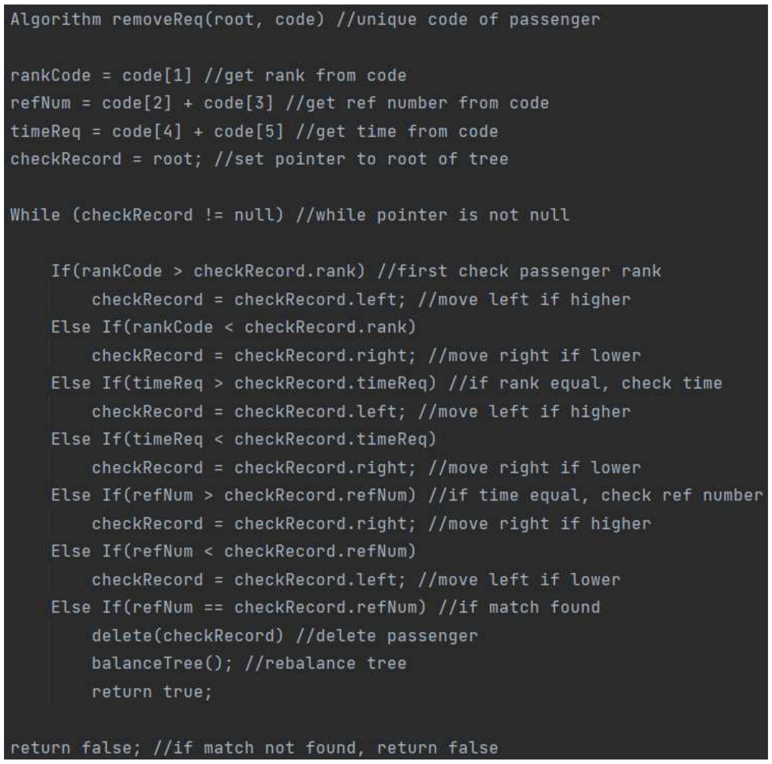
This is the removePassenger method which removes a passenger object from the TreeSet. It converts the confirmationCode of the passenger into rank, referenceID and waitingTime which is used to create a temporary passenger object. The passenger is deleted from the TreeSet if a duplicate is found.



```
Algorithm upgrade(root, k) //k number of seat upgrades

While(k != 0) //while seat upgrades available

    checkRecord = root; //set pointer to root of tree

    While(checkRecord.left != null) //while not leftmost node
        checkRecord = checkRecord.left; //move left

    delete(checkRecord) //remove leftmost/highest priority node
    k := k - 1; //decrement seat upgrade by 1
    balanceTree(); //rebalance tree
```



```
Algorithm reqUpgrade(root, passenger) //passenger to be added

checkRecord = root; //set pointer to root of tree
While (checkRecord != null) //while pointer is not null

    If(passenger.rank > checkRecord.rank) //first check passenger rank
        checkRecord = checkRecord.left; //move left if higher
    Else if(passenger.rank < checkRecord.rank)
        checkRecord = checkRecord.right; //move right if lower
    Else If(passenger.timeReq > checkRecord.timeReq) //if rank equal, check time
        checkRecord = checkRecord.left; //move left if higher
    Else If(passenger.timeReq < checkRecord.timeReq)
        checkRecord = checkRecord.right; //move right if lower
    Else If(passenger.refNum > checkRecord.refNum) //if time equal check ref number
        checkRecord = checkRecord.right; //move right if higher
    Else If(passenger.refNum < checkRecord.refNum)
        checkRecord = checkRecord.left; //move left if lower
    Else If(passenger.refNum == checkRecord.refNum) //if match found, return false
        return false;

insert(passenger); //if no match found, insert new passenger
balanceTree(); //rebalance tree
return true;
```



```
Algorithm removeReq(root, code) //unique code of passenger

rankCode = code[1] //get rank from code
refNum = code[2] + code[3] //get ref number from code
timeReq = code[4] + code[5] //get time from code
checkRecord = root; //set pointer to root of tree

While (checkRecord != null) //while pointer is not null

    If(rankCode > checkRecord.rank) //first check passenger rank
        checkRecord = checkRecord.left; //move left if higher
    Else If(rankCode < checkRecord.rank)
        checkRecord = checkRecord.right; //move right if lower
    Else If(timeReq > checkRecord.timeReq) //if rank equal, check time
        checkRecord = checkRecord.left; //move left if higher
    Else If(timeReq < checkRecord.timeReq)
        checkRecord = checkRecord.right; //move right if lower
    Else If(refNum > checkRecord.refNum) //if time equal, check ref number
        checkRecord = checkRecord.right; //move right if higher
    Else If(refNum < checkRecord.refNum)
        checkRecord = checkRecord.left; //move left if lower
    Else If(refNum == checkRecord.refNum) //if match found
        delete(checkRecord) //delete passenger
        balanceTree(); //rebalance tree
        return true;

return false; //if match not found, return false
```

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