COP5536
Programming Project
Assignment-1

Name: Shanmukha Anurag
UFID: 58715595
UF email: shanmukhaa.sajja@ufl.edu
Implementation Language: Python

PROJECT DESCRIPTION: GATOR TAXI

GatorTaxi is an up-and-coming ride-sharing service. They get many ride requests daily and plan to develop new software to keep track of their pending ride requests.

The following triplet identifies a ride:
rideNumber: unique integer identifier for each ride.
rideCost: The estimated cost (in integer dollars) for the ride.
tripDuration: the total time (in integer minutes) needed to get from pickup to destination.

In this project, we are asked to develop a min heap and red-black tree, using which we have to perform the following request for the GatorTaxi.

1.**Print(rideNumber)**: Prints the rideNumber, rideCost, and tripDuration of the corresponding rideNumber.
2.**Print(rideNumber1, rideNumber2)**: Prints the rideNumber, rideCost, and tripDuration of all the rides with rideNumber between rideNumber1 and rideNumber2.
3.**Insert (rideNumber, rideCost, tripDuration)**: Insert the Ride with the corresponding details given in the parameters into the Min Heap and the Red Black Tree according to the corresponding rules mentioned below. If there is already a ride in the Min Heap / the Red Black Tree, then output an error message "Duplicate RideNumber" and terminate the program.
4.**GetNextRide():** Prints the details of the ride with the smallest rideCost available in a heap (extract_min) and deletes the corresponding nodes from the Min Heap and the Red Black Node.
5.**CancelRide(rideNumber)**: Deletes the ride with the corresponding node with the given rideNumber from the Min Heap and Red Black Tree
6.**UpdateTrip(rideNumber, new_tripDuration)**: Update the tripDuration of the corresponding rideNumber based on the following rules.

a) if the new_tripDuration <= existing tripDuration, update the tripDuration.
b)if the existing_tripDuration < new_tripDuration <= 2*(existing tripDuration), the driver will cancel the existing ride, and a new ride request will be created with a penalty of 10 on the existing ride cost. We update the entry in the data structure with (rideNumber, rideCost+10, new_tripDuration)
c) if the new_tripDuration > 2*(existing tripDuration), the ride would be automatically declined, and the ride would be removed from the data structure.

FUNCTION PROTOTYPES AND MODULES

1. Ride:

These objects consist of the triplets rideNumber, rideCost, tripDuration.

Ride Function Prototypes:

```
1. def __init__(self, rideNumber, rideCost, tripDuration):
```

2. Min Heap:

The execution of Min Heap is standard. But the implementation suggests that the Rides with the information triplets rideNumber, rideCost, and tripDuration be heapified as per the rideCost and, in case of a tie, then heapified according to tripDuration.

Min Heap Function Prototypes:

```
1. def __init__(self):
2. def insert(self, ride):
3. def extract_min(self):
4. def delete(self, ride):
5. def heapifyUp(self, index):
6. def heapifyDown(self, index):
7. def swap(self, i, j):
```

3. Red Black Tree:

The Red Black Tree is used to store and sort the rideNumber. We must ensure that pointers exist between the Red Black Tree and the Min Heap. The Binary Search Tree is built to balance and avoid left-leaning and right-leaning trees to the maximum extent possible.

Red Black Tree Function Prototypes:

```
1. def __init__(self):
2. def searchNode(self, node, key):
3. def deleteRotationHelper(self, currNode):
4. def swapNodes(self, one, two):
5. def deleteNode(self, node, key):
6. def insertRotationHelper(self, key):
7. def searchTree(self, key):
8. def minimum(self, node):
9. def lrRotation(self, x):
10. def rlRotate(self, x):
11. def insert(self, key):
12. def get_root(self):
13. def delete_node(self, data):
14. def pretty_print(self):
```

4. GatorTaxi

The GatorTaxi has to perform the functions described above in the Project Description.
PrintRide(RideNumber),
PrintRide(RideNumber1,RideNumber2),
InsertRide(RideNumber,RideCost,TripDuration),
UpdateRide(RideNumber, TripDuration),
CancelRide(RideNumber),
GetNextRide()

GatorTaxi Function Prototype:

```
1. if __name__ == "__main__":
2. def insertRide(redBlackTree,minHeap,keyValueDict,rideNumber,rideCost,tripDuration):
3. def printRide(redBlackTree,minHeap,keyValueDict,rideNumber):
4. def printRides(root,minHeap,keyValueDict,x,rideNumber1,rideNumber2):
5. def getNextRide(redBlackTree, minHeap, keyValueDict):
6. def cancelRide(redBlackTree,minHeap,keyValueDict,rideNumber):
7. def updateRide(redBlackTree,minHeap, keyValueDict, rideNumber, newTD):
```

STRUCTURE OF THE DIRECTORY

Sajja_Shanmukha Anurag(unzipped) – --
|__ gatorTaxi.py
|__ input.txt (included or can be added)

Command to run the file:
<Directory in which the files are present/>python3 gatorTaxi.py <fileName>

Here in this case
Sajja_Shanmukha\ Anurag/> python3 gatorTaxi.py input.txt

CODE IMPLEMENTATION AND STRUCTURE

Ride Structure :

```
class Ride:

    def __init__(self, rideNumber, rideCost, tripDuration):

        self.rideNumber = rideNumber
        self.rideCost = rideCost
        self.tripDuration = tripDuration
```

The code represents the structure and implementation of the class Ride with the elements rideNumber, rideCost, and tripDuration.

Min Heap Structure :

```
class MinHeap:
    def __init__(self):
        self.heap = []
        self.positions = {}

    def insert(self, ride):

        self.heap.append(ride)

        index = len(self.heap)-1
        self.positions[ride.rideNumber] = index

        self.heapifyUp(index)

    def extract_min(self):

        if len(self.heap) == 0:
            return None

        if len(self.heap) == 1:
            ride = self.heap.pop()
            self.positions.pop(ride.rideNumber)
            return ride

        min_val = self.heap[0]

        self.heap[0] = self.heap.pop()
```

```
        self.positions.pop(min_val.rideNumber)
        self.positions[self.heap[0].rideNumber] = 0

        self.heapifyDown(0)

        return min_val

    def delete(self, ride):

        if len(self.heap) == 0:
            return False

        if ride.rideNumber not in self.positions:
            return False

        index = self.positions[ride.rideNumber]

        self.heap[index] = self.heap[-1]

        self.positions[self.heap[-1].rideNumber] = index

        self.heap.pop()

        del self.positions[ride.rideNumber]

        if index == len(self.heap):
            return True
        self.heapifyUp(index)
        self.heapifyDown(index)
        return True

    def heapifyUp(self, index):

        prtNode = (index - 1) // 2

        if prtNode >= 0 and (self.heap[prtNode].rideCost > self.heap[index].rideCost or
(self.heap[prtNode].rideCost == self.heap[index].rideCost and
self.heap[prtNode].tripDuration > self.heap[index].tripDuration)):
            self.swap(index, prtNode)
            self.heapifyUp(prtNode)

    def heapifyDown(self, index):

        lchild = index * 2 + 1
        rchild = index * 2 + 2

        minimum = index

        if (lchild < len(self.heap) and
            (self.heap[lchild].rideCost < self.heap[minimum].rideCost or
            (self.heap[lchild].rideCost == self.heap[minimum].rideCost and
```

```
            self.heap[lchild].tripDuration < self.heap[minimum].tripDuration))):
            minimum = lchild

        if (rchild < len(self.heap) and
            (self.heap[rchild].rideCost < self.heap[minimum].rideCost or
            (self.heap[rchild].rideCost == self.heap[minimum].rideCost and
            self.heap[rchild].tripDuration < self.heap[minimum].tripDuration))):
            minimum = rchild

        if minimum != index:
            self.swap(index, minimum)
            self.heapifyDown(minimum)

    def swap(self, i, j):

        self.positions[self.heap[i].rideNumber] = j
        self.positions[self.heap[j].rideNumber] = i

        self.heap[i], self.heap[j] = self.heap[j], self.heap[i]
```

The Min Heap implements the structure by holding the ride elements from the Ride class.

The __init__ function is the constructor that helps initialize the Heap and positions, acting as a dictionary and pointer for the Red Black Tree.

The Insert function adds the element to the end of the array and works it way up to maintain the structure of the heap(heapifyUp). Update the element's position in the dictionary to maintain pointers from the red-black tree.

The extract_min function utilizes the feature of the heap that the topmost element in the array representation of the heap is the minimum and removes and returns it. The Last Ride element in the heap array is sent to its place and works its way down to maintain the heap's structure (heapifyDown).

The delete function usually takes place in O(n) time duration. But the positions dictionary would help delete the element from the array in O(1) time, and then to maintain the structure of the heap, the array is heapified in both directions (heapifyUp), (heapifyDown). This heapify operation would happen in the time of O(logn)

The heapifyUp function checks for the rideCost of the parent node and the current node and swaps them if the parent node's rideCost is higher. If both the rideCosts are equal, then the tripDuration of the parent and current node are compared similarly. And this process continued from the current parent node index.

The heapifyDown function checks the rideCost of the current node and its left and right child rideCosts and swaps with whichever is smaller. If both the rideCosts of the current node and one of its children are equal, then the tripDurations of the nodes are compared similarly to the rideCosts. This process continues to the end of the array representation of the heap to the swapped children index.

The swap function is introduced to update the element's index positions in the array representation of the heap whenever efforts are made by the heapifyUp and heapifyDown functions to maintain the heap's structure.

The **space complexity** of the data structure would be O(N)

Red Black Tree Structure:

```
class Node():
    def __init__(self, data):

        self.data = data

        self.prt = None
        self.lft = None
        self.rgt = None

        self.clr = 1

class RedBlackTree():
    def __init__(self):

        self.tnull = Node(0)
        self.tnull.clr = 0
        self.tnull.lft = None
        self.tnull.rgt = None
        self.root = self.tnull

    def searchNode(self, nd, rideNo):

        if nd == self.tnull or rideNo == nd.data:
            return nd
        if rideNo < nd.data:
            return self.searchNode(nd.lft, rideNo)
        return self.searchNode(nd.rgt, rideNo)

    def deleteRotationHelper(self, currNode):

        while currNode != self.root and currNode.clr == 0:

            if currNode == currNode.prt.lft:

                siblingNode = currNode.prt.rgt

                if siblingNode.clr == 1:
                    siblingNode.clr = 0
                    currNode.prt.clr = 1
                    self.lrRotation(currNode.prt)
                    siblingNode = currNode.prt.rgt

                if siblingNode.lft.clr == 0 and siblingNode.rgt.clr == 0:
                    siblingNode.clr = 1
                    currNode = currNode.prt
                else:

                    if siblingNode.rgt.clr == 0:
                        siblingNode.lft.clr = 0
                        siblingNode.clr = 1
```

```
                    self.rlRotate(siblingNode)
                    siblingNode = currNode.prt.rgt

                siblingNode.clr = currNode.prt.clr
                currNode.prt.clr = 0
                siblingNode.rgt.clr = 0
                self.lrRotation(currNode.prt)
                currNode = self.root

        else:

            siblingNode = currNode.prt.lft

            if siblingNode.clr == 1:
                siblingNode.clr = 0
                currNode.prt.clr = 1
                self.rlRotate(currNode.prt)
                siblingNode = currNode.prt.lft

            if siblingNode.lft.clr == 0 and siblingNode.rgt.clr == 0:
                siblingNode.clr = 1
                currNode = currNode.prt
            else:

                if siblingNode.lft.clr == 0:
                    siblingNode.rgt.clr = 0
                    siblingNode.clr = 1
                    self.lrRotation(siblingNode)
                    siblingNode = currNode.prt.lft

                siblingNode.clr = currNode.prt.clr
                currNode.prt.clr = 0
                siblingNode.lft.clr = 0
                self.rlRotate(currNode.prt)
                currNode = self.root

    currNode.clr = 0

def swapNodes(self, one, two):
    if one.prt == None:
        self.root = two
    elif one == one.prt.lft:
        one.prt.lft = two
    else:
        one.prt.rgt = two
    two.prt = one.prt

def deleteNode(self, nd, rideNo):
    z = self.tnull
    while nd != self.tnull:
        if nd.data == rideNo:
```

```
            z = nd
        if nd.data <= rideNo:
            nd = nd.rgt
        else:
            nd = nd.lft
    if z == self.tnull:
        return
    copy = z
    copyColor = copy.clr
    if z.lft == self.tnull:
        childNode = z.rgt
        self.swapNodes(z, z.rgt)
    elif (z.rgt == self.tnull):
        childNode = z.lft
        self.swapNodes(z, z.lft)
    else:
        copy = self.minimum(z.rgt)
        copyColor = copy.clr
        childNode = copy.rgt
        if copy.prt == z:
            childNode.prt = copy
        else:
            self.swapNodes(copy, copy.rgt)
            copy.rgt = z.rgt
            copy.rgt.prt = copy
        self.swapNodes(z, copy)
        copy.lft = z.lft
        copy.lft.prt = copy
        copy.clr = z.clr

    if copyColor == 0:
        self.deleteRotationHelper(childNode)

def insertRotationHelper(self, rideNo):

    while rideNo.prt.clr == 1:

        if rideNo.prt == rideNo.prt.prt.rgt:
            uncle = rideNo.prt.prt.lft
            if uncle.clr == 1:

                uncle.clr = 0
                rideNo.prt.clr = 0
                rideNo.prt.prt.clr = 1
                rideNo = rideNo.prt.prt
            else:
                if rideNo == rideNo.prt.lft:

                    rideNo = rideNo.prt
                    self.rlRotate(rideNo)
```

```
                rideNo.prt.clr = 0
                rideNo.prt.prt.clr = 1
                self.lrRotation(rideNo.prt.prt)
        else:
            uncle = rideNo.prt.prt.rgt
            if uncle.clr == 1:

                uncle.clr = 0
                rideNo.prt.clr = 0
                rideNo.prt.prt.clr = 1
                rideNo = rideNo.prt.prt
            else:
                if rideNo == rideNo.prt.rgt:

                    rideNo = rideNo.prt
                    self.lrRotation(rideNo)

                rideNo.prt.clr = 0
                rideNo.prt.prt.clr = 1
                self.rlRotate(rideNo.prt.prt)
        if rideNo == self.root:
            break
    self.root.clr = 0

def searchTree(self, rideNo):
    return self.searchNode(self.root, rideNo)

def minimum(self, nd):
    while nd.lft != self.tnull:
        nd = nd.lft
    return nd

def lrRotation(self, x):
    y = x.rgt
    x.rgt = y.lft

    if y.lft != self.tnull:
        y.lft.prt = x

    y.prt = x.prt

    if x.prt == None:
        self.root = y
    elif x == x.prt.lft:
        x.prt.lft = y
    else:
        x.prt.rgt = y

    y.lft = x
    x.prt = y
```

```
def rlRotate(self, x):
    y = x.lft
    x.lft = y.rgt

    if y.rgt != self.tnull:
        y.rgt.prt = x

    y.prt = x.prt

    if x.prt == None:
        self.root = y
    elif x == x.prt.rgt:
        x.prt.rgt = y
    else:
        x.prt.lft = y

    y.rgt = x
    x.prt = y

def insert(self, rideNo):

    nd = Node(rideNo)
    nd.prt = None
    nd.data = rideNo
    nd.lft = self.tnull
    nd.rgt = self.tnull
    nd.clr = 1

    y = None
    x = self.root
    while x != self.tnull:
        y = x
        if nd.data < x.data:
            x = x.lft
        elif nd.data > x.data:
            x = x.rgt
        else:
            pass

    nd.prt = y
    if y == None:
        self.root = nd
    elif nd.data < y.data:
        y.lft = nd
    else:
        y.rgt = nd

    if nd.prt == None:
        nd.clr = 0
        return
```

```
        if nd.prt.prt == None:
            return

        self.insertRotationHelper(nd)

    def get_root(self):
        return self.root

    def delete_node(self, data):
        self.deleteNode(self.root, data)

    def pretty_print(self):
        self.__print_helper(self.root, "", True)
```

The functions in the Red-Black Trees are divided into helper functions which help achieve the essential functions' tasks. The init function acted as the constructor and initialized the node to be red with the assigned data.

The insert function helps to enter the data to the node initialized with the color red and follows to add to the node according to the value of the rideNumber in the BinarySearchTree. But the Red Black Tree rules ensure that the root node is always red and the number of black nodes is balanced on every node's left and right subtrees. Along with this, all the leaf nodes are considered null nodes, and these null nodes are observed as black. We have to perform the appropriate left and right rotations to achieve the BinarySearchTree along with following the rules of the RedBlackTree.

Like the insert function, the delete function deletes the respective node specified and replaces it with the in-order successor of the deleted node. But this can collapse the structure of the RedBlackTree by not following the minimum requirements of the RedBlackTrees. This function takes the help of several helper and rotation functions to follow the rules of the RedBlackTree.

All the functions other than the insert delete functions are helper functions that handle various variations to follow the red-black trees.

The **space complexity** of the data structure would be O(N)

GatorTaxi Structure:

```
def insertRide(redBlackTree,minHeap,keyValueDict,rideNumber,rideCost,tripDuration):
    try:

        x=keyValueDict[rideNumber]
        return 0
    except:
        keyValueDict[rideNumber]=[rideCost,tripDuration]
        redBlackTree.insert(rideNumber)
        minHeap.insert(Ride(rideNumber,rideCost,tripDuration))
        return 1
def printRide(redBlackTree,minHeap,keyValueDict,rideNumber):
    try:

        return [rideNumber,keyValueDict[rideNumber][0],keyValueDict[rideNumber][1]]
    except:
        return [0,0,0]
def printRides(root,minHeap,keyValueDict,x,rideNumber1,rideNumber2):
    try:
        printRides(root.lft,minHeap,keyValueDict,x,rideNumber1,rideNumber2)
        if root.data >= rideNumber1 and root.data <= rideNumber2:

x.append([root.data,keyValueDict[root.data][0],keyValueDict[root.data][1]])
        else:
            pass
        printRides(root.rgt,minHeap,keyValueDict,x,rideNumber1,rideNumber2)
    except:
        pass
def getNextRide(redBlackTree, minHeap, keyValueDict):

    x = minHeap.extract_min()
    try:
        p = x.rideNumber
        redBlackTree.delete_node(p)
        keyValueDict.pop(p)
        return [x.rideNumber,x.rideCost,x.tripDuration]
    except:
        return "No active ride requests"

def cancelRide(redBlackTree,minHeap,keyValueDict,rideNumber):

    try:
```

```
        redBlackTree.delete_node(rideNumber)

minHeap.delete(Ride(rideNumber,keyValueDict[rideNumber][0],keyValueDict[rideNumber]
[1]))
        keyValueDict.pop(rideNumber)
    except:
        pass

def updateRide(redBlackTree,minHeap, keyValueDict, rideNumber, newTD):
    try:
        x = keyValueDict[rideNumber][1]
        if newTD < x:

minHeap.delete(Ride(rideNumber,keyValueDict[rideNumber][0],keyValueDict[rideNumber]
[1]))
            minHeap.insert(Ride(rideNumber,keyValueDict[rideNumber][0],newTD))
            keyValueDict[rideNumber][1]=newTD
        elif newTD > x and newTD < 2* x:

minHeap.delete(Ride(rideNumber,keyValueDict[rideNumber][0],keyValueDict[rideNumber]
[1]))
            minHeap.insert(Ride(rideNumber,keyValueDict[rideNumber][0]+10,newTD))
            keyValueDict[rideNumber][0]=keyValueDict[rideNumber][0]+10
            keyValueDict[rideNumber][1]=newTD
        elif newTD>2*x:

minHeap.delete(Ride(rideNumber,keyValueDict[rideNumber][0],keyValueDict[rideNumber]
[1]))
            redBlackTree.delete_node(rideNumber)
            keyValueDict.pop(rideNumber)
        else:
            pass
    except:
        pass
if __name__ == "__main__":
    redBlackTree = RedBlackTree()
    minHeap = MinHeap()
    keyValueDict=dict()
    fread = open(sys.argv[1],'r')
    fwrite = open("output_file.txt", 'w')
    flag = 1
    while flag:

        line = fread.readline()
```

```
        if "Insert" in line:
            cmdArgs = line[7:-2:]
            argList = cmdArgs.split(",")
            rideNumber=int(argList[0])
            rideCost = int(argList[1])
            tripDuration = int(argList[2])
            flag =
insertRide(redBlackTree,minHeap,keyValueDict,rideNumber,rideCost,tripDuration)
            if flag==0:
                fwrite.write("Duplicate RideNumber\n")

        if "GetNextRide" in line:

            l = getNextRide(redBlackTree,minHeap,keyValueDict)

            if "No active ride requests"==l:
                fwrite.write(l + "\n")
            else:
                fwrite.write("("+str(l[0])+","+str(l[1])+","+str(l[2])+")\n")

        if "Print" in line:
            cmdArgs = line[6:-2:]
            argList = cmdArgs.split(",")

            if len(argList)==1:
                l=printRide(redBlackTree,minHeap,keyValueDict,int(argList[0]))
                fwrite.write("("+str(l[0])+","+str(l[1])+","+str(l[2])+")\n")

            else:

                p = []

printRides(redBlackTree.root,minHeap,keyValueDict,p,int(argList[0]),int(argList[1])
)
                string = ""

                if len(p)==0:
                    fwrite.write("(0,0,0)\n")

                else:

                    for i in p:
                        string += "("+str(i[0])+","+str(i[1])+","+str(i[2])+")" +
","
                    fwrite.write(string[:-1] + "\n")
```

```
        if "UpdateTrip" in line:
            cmdArgs = line[11:-2:]
            argList = cmdArgs.split(",")

updateRide(redBlackTree,minHeap,keyValueDict,int(argList[0]),int(argList[1]))

        if "Cancel" in line:
            cmdArgs = line[11:-2:]
            argList = cmdArgs.split(',')
            cancelRide(redBlackTree,minHeap,keyValueDict,int(argList[0]))

        if line == "":
            flag = 0

    fwrite.close()
    fread.close()
```

The main function opens the input file given in the argument in the command line. It initializes a RedBlackTree and a min heap along with a dictionary to store the values of the Rides and perform the required operations.

Once the input file is opened, the pointer scans each line from the file and analyzes the operation based on the values in the string.

The required operations are performed by using the corresponding functions as mentioned below.

1. **PrintRide(RideNumber) :**
   `def printRide(redBlackTree,minHeap,keyValueDict,rideNumber):`
2. **PrintRide(RideNumber1,RideNumber2) :**
   `def printRides(root,minHeap,keyValueDict,x,rideNumber1,rideNumber2):`
3. **InsertRide(RideNumber,RideCost,TripDuration) :**
   `def insertRide(redBlackTree,minHeap,keyValueDict,rideNumber,rideCost,tripDuration):`
4. **UpdateRide(RideNumber, TripDuration) :**
   `def updateRide(redBlackTree,minHeap, keyValueDict, rideNumber, newTD):`
5. **CancelRide(RideNumber) :**
   `def cancelRide(redBlackTree,minHeap,keyValueDict,rideNumber):`
6. **GetNextRide() :**
   `def getNextRide(redBlackTree, minHeap, keyValueDict):`

TIME AND SPACE COMPLEXITY

1. PrintRide(RideNumber) :

```
def printRide(redBlackTree,minHeap,keyValueDict,rideNumber):
```

The print ride uses the rideNumber as an argument to find the corresponding rideNumber in the Red Black Tree, an O(logn) operation where n is the number of active rides (elements in the red-black tree). Once we find the red-black tree, the keyValueDict dictionary returns the corresponding rideCost and the tripDuration in O(1) time.

Hence, the time complexity of the **PrintRide (RideNumber)** is **O(logn)**.

The **space complexity** would be **0(1)**

2. PrintRide(RideNumber1,RideNumber2) :

```
def printRides(root,minHeap,keyValueDict,x,rideNumber1,rideNumber2):
```

The printRides function receives two rideNumbers (rideNumber1, rideNumber2). We perform a similar to - inorder traversal on the red-black tree. To find the values of rideNumbers between these two rideNumbers.

These values are collected in logN traversal to find rideNumber1 and back height S to travel the way upward in order traversal does and travel downward to the rideNumber2 another O(N) to find the rideNumber2.

The corresponding rideCost and tripDuration are returned from the keyValueDict.The total time complexity of this operation would take logn + logn + traversal to each element in between.

Hence the operation costs 2*logn + S, where S is the number of rides to be printed. The **time complexity** of **PrintRide(RideNumber1, RideNumber2)** operation would be **O(logN + S)** where 0 <= S <= N where n is the number of active rides.

The **space complexity** of the same is **O(S)**

3. InsertRide(RideNumber,RideCost,TripDuration) :

```
def insertRide(redBlackTree,minHeap,keyValueDict,rideNumber,rideCost,tripDuration):
```

The insert operation takes the triplet rideNumber, rideCost, and tripDuration, initializes a ride object from the class Ride, and adds the corresponding ride to the min heap. Also, the keyValueDict dictionary is updated for the corresponding {rideNumber -> [rideCost, tripDuration]}. The positions dictionary in the Min Heap acts as the pointer between the RedBlackTree and the Min Heap. The positions dictionary acts with the {rideNumber -> [index in the Min Heap]}

Adding the rideNumber to the keyValueDict dictionary would help identify any duplicates in rideNumbers and quickly return “Duplicate RideNumber.”

The insertion into the Min Heap is done by appending the Ride element to the end of the array representation of the heap O(1) and performing the heapify up operation O(logn).

The insertion of rideNumber into the RedBlackTree would take O(logn) along with the additional cost of the rotations performed, which is negligible.

Thus the total operation would cost 1*logn + logn + C. The **time complexity** of the **InsertRide** operation would be **O(logn)**.

The **space complexity** would be **0(1)** for the operation.

The **space complexity** for the total **Min Heap** would be **O(N)**
The **space complexity** for the total **Red-Black Tree** would be **O(N)**

4. UpdateRide(RideNumber, TripDuration) :

def updateRide(redBlackTree,minHeap, keyValueDict, rideNumber, newTD):

The updateRide operation finds the corresponding ride with the given rideNumber in the Min Heap using the positions dictionary.

If the new_tripDuration given is less than the existing_tripDuration, then the heapify operation is performed after updating the tripDuration of the ride in the Min Heap. While this operation is performed, all the key Value pairs in the positions dictionary are updated according to the new indexes of the rides in the Min Heap.

If the new_tripDuration is between the existing tripDuration and 2* existing_tripDuration, then the corresponding ride with the given rideNumber is found in the Min Heap. The rideCost is updated to rideCost + 10, and the new_tripDuration is also updated. Then the heapify operation is performed, and the positions dictionary is updated accordingly..

If the new_tripDuration is greater than the 2*existing_tripDuration, the ride is found in the Min Heap using the positions dictionary and deleted. And the heapify operation is performed, the positions dictionary is updated accordingly, and the key with the given rideNumber is deleted from the dictionary. In this case, we must also update the RedBlackTree since the ride is deleted. The other cases wouldn't require this as the RedBlackTree only stores the rideNumber.

Finding the Ride element in the Min Heap can be done in O(1) using the positions dictionary. The heapify operation in every case can be done in O(logn). The deletion in the RedBlackTree can also be done in O(logn) time.

The following changes are also updated in the keyValueDict.

Thus, the total operation would cost logn (best case) and logn + logn (worst case). The **time complexity** of the **UpdateRide** operation would be **O(logn)**.

The **space complexity** of the operation would be **O(1)**

5. CancelRide(RideNumber) :

def cancelRide(redBlackTree,minHeap,keyValueDict,rideNumber):

The cancelRide function finds the corresponding rideNumber in the RedBlackTree, deletes the node, and performs the required rotations at an additional cost. The operation then finds the corresponding Ride element to the given rideNumber and finds the position of the Ride in the Min Heap using the positions dictionary. And then, the heapify operation is performed, and key-value pairs in the positions dictionary are updated.

The changes are also updated in the keyValueDict.

Thus, the total operation would cost logn + 1*logn time. The **time complexity** of the **CancelRide** operation would be **O(logn)**.

The **space complexity** of the operation would be **O(1)**

6. GetNextRide() :

def getNextRide(redBlackTree, minHeap, keyValueDict):

The getNextRide function is the extract_min operation in the Min Heap. This is the topmost element of the array representation of the Min Heap. This ride is returned from the extract_min function, and then the heapify operation is performed over the Min Heap, and the positions dictionary is updated accordingly. The returned ride element gives the required rideNumber, and the corresponding rideNumber is deleted from the RedBlackTree.

The corresponding key-value pair with the rideNumber are also deleted from the keyValueDict.

Thus, the total operation costs are logn + logn time. The **time complexity** of the GetNextRide operation would be **O(logn)**.

The **space complexity** of the operation would be **O(1).**

COMPLETE CODE + IMPLEMENTATION

```
import sys

class Ride:

    def __init__(self, rideNumber, rideCost, tripDuration):

        self.rideNumber = rideNumber
        self.rideCost = rideCost
        self.tripDuration = tripDuration

class MinHeap:
    def __init__(self):
        self.heap = []
        self.positions = {}

    def insert(self, ride):

        self.heap.append(ride)

        index = len(self.heap)-1
        self.positions[ride.rideNumber] = index

        self.heapifyUp(index)

    def extract_min(self):

        if len(self.heap) == 0:
            return None

        if len(self.heap) == 1:
            ride = self.heap.pop()
            self.positions.pop(ride.rideNumber)
            return ride

        min_val = self.heap[0]

        self.heap[0] = self.heap.pop()

        self.positions.pop(min_val.rideNumber)
        self.positions[self.heap[0].rideNumber] = 0

        self.heapifyDown(0)

        return min_val

    def delete(self, ride):

        if len(self.heap) == 0:
```

```
            return False

        if ride.rideNumber not in self.positions:
            return False

        index = self.positions[ride.rideNumber]

        self.heap[index] = self.heap[-1]

        self.positions[self.heap[-1].rideNumber] = index

        self.heap.pop()

        del self.positions[ride.rideNumber]

        if index == len(self.heap):
            return True
        self.heapifyUp(index)
        self.heapifyDown(index)
        return True

    def heapifyUp(self, index):

        prtNode = (index - 1) // 2

        if prtNode >= 0 and (self.heap[prtNode].rideCost > self.heap[index].rideCost or
(self.heap[prtNode].rideCost == self.heap[index].rideCost and
self.heap[prtNode].tripDuration > self.heap[index].tripDuration)):
            self.swap(index, prtNode)
            self.heapifyUp(prtNode)

    def heapifyDown(self, index):

        lchild = index * 2 + 1
        rchild = index * 2 + 2

        minimum = index

        if (lchild < len(self.heap) and
            (self.heap[lchild].rideCost < self.heap[minimum].rideCost or
            (self.heap[lchild].rideCost == self.heap[minimum].rideCost and
            self.heap[lchild].tripDuration < self.heap[minimum].tripDuration))):
            minimum = lchild

        if (rchild < len(self.heap) and
            (self.heap[rchild].rideCost < self.heap[minimum].rideCost or
            (self.heap[rchild].rideCost == self.heap[minimum].rideCost and
            self.heap[rchild].tripDuration < self.heap[minimum].tripDuration))):
            minimum = rchild

        if minimum != index:
```

```
            self.swap(index, minimum)
            self.heapifyDown(minimum)

    def swap(self, i, j):

        self.positions[self.heap[i].rideNumber] = j
        self.positions[self.heap[j].rideNumber] = i

        self.heap[i], self.heap[j] = self.heap[j], self.heap[i]

class Node():
    def __init__(self, data):

        self.data = data

        self.prt = None
        self.lft = None
        self.rgt = None

        self.clr = 1

class RedBlackTree():
    def __init__(self):

        self.tnull = Node(0)
        self.tnull.clr = 0
        self.tnull.lft = None
        self.tnull.rgt = None
        self.root = self.tnull

    def searchNode(self, nd, rideNo):

        if nd == self.tnull or rideNo == nd.data:
            return nd
        if rideNo < nd.data:
            return self.searchNode(nd.lft, rideNo)
        return self.searchNode(nd.rgt, rideNo)

    def deleteRotationHelper(self, currNode):

        while currNode != self.root and currNode.clr == 0:

            if currNode == currNode.prt.lft:

                siblingNode = currNode.prt.rgt

                if siblingNode.clr == 1:
                    siblingNode.clr = 0
                    currNode.prt.clr = 1
                    self.lrRotation(currNode.prt)
                    siblingNode = currNode.prt.rgt
```

```
            if siblingNode.lft.clr == 0 and siblingNode.rgt.clr == 0:
                siblingNode.clr = 1
                currNode = currNode.prt
            else:

                if siblingNode.rgt.clr == 0:
                    siblingNode.lft.clr = 0
                    siblingNode.clr = 1
                    self.rlRotate(siblingNode)
                    siblingNode = currNode.prt.rgt

                siblingNode.clr = currNode.prt.clr
                currNode.prt.clr = 0
                siblingNode.rgt.clr = 0
                self.lrRotation(currNode.prt)
                currNode = self.root

        else:

            siblingNode = currNode.prt.lft

            if siblingNode.clr == 1:
                siblingNode.clr = 0
                currNode.prt.clr = 1
                self.rlRotate(currNode.prt)
                siblingNode = currNode.prt.lft

            if siblingNode.lft.clr == 0 and siblingNode.rgt.clr == 0:
                siblingNode.clr = 1
                currNode = currNode.prt
            else:

                if siblingNode.lft.clr == 0:
                    siblingNode.rgt.clr = 0
                    siblingNode.clr = 1
                    self.lrRotation(siblingNode)
                    siblingNode = currNode.prt.lft

                siblingNode.clr = currNode.prt.clr
                currNode.prt.clr = 0
                siblingNode.lft.clr = 0
                self.rlRotate(currNode.prt)
                currNode = self.root

    currNode.clr = 0

def swapNodes(self, one, two):
    if one.prt == None:
        self.root = two
    elif one == one.prt.lft:
```

```
            one.prt.lft = two
        else:
            one.prt.rgt = two
        two.prt = one.prt

    def deleteNode(self, nd, rideNo):
        z = self.tnull
        while nd != self.tnull:
            if nd.data == rideNo:
                z = nd
            if nd.data <= rideNo:
                nd = nd.rgt
            else:
                nd = nd.lft
        if z == self.tnull:
            return
        copy = z
        copyColor = copy.clr
        if z.lft == self.tnull:
            childNode = z.rgt
            self.swapNodes(z, z.rgt)
        elif (z.rgt == self.tnull):
            childNode = z.lft
            self.swapNodes(z, z.lft)
        else:
            copy = self.minimum(z.rgt)
            copyColor = copy.clr
            childNode = copy.rgt
            if copy.prt == z:
                childNode.prt = copy
            else:
                self.swapNodes(copy, copy.rgt)
                copy.rgt = z.rgt
                copy.rgt.prt = copy
            self.swapNodes(z, copy)
            copy.lft = z.lft
            copy.lft.prt = copy
            copy.clr = z.clr

        if copyColor == 0:
            self.deleteRotationHelper(childNode)

    def insertRotationHelper(self, rideNo):

        while rideNo.prt.clr == 1:

            if rideNo.prt == rideNo.prt.prt.rgt:
                uncle = rideNo.prt.prt.lft
                if uncle.clr == 1:

                    uncle.clr = 0
```

```
                rideNo.prt.clr = 0
                rideNo.prt.prt.clr = 1
                rideNo = rideNo.prt.prt
            else:
                if rideNo == rideNo.prt.lft:

                    rideNo = rideNo.prt
                    self.rlRotate(rideNo)

                rideNo.prt.clr = 0
                rideNo.prt.prt.clr = 1
                self.lrRotation(rideNo.prt.prt)
        else:
            uncle = rideNo.prt.prt.rgt
            if uncle.clr == 1:

                uncle.clr = 0
                rideNo.prt.clr = 0
                rideNo.prt.prt.clr = 1
                rideNo = rideNo.prt.prt
            else:
                if rideNo == rideNo.prt.rgt:

                    rideNo = rideNo.prt
                    self.lrRotation(rideNo)

                rideNo.prt.clr = 0
                rideNo.prt.prt.clr = 1
                self.rlRotate(rideNo.prt.prt)
        if rideNo == self.root:
            break
    self.root.clr = 0

def searchTree(self, rideNo):
    return self.searchNode(self.root, rideNo)

def minimum(self, nd):
    while nd.lft != self.tnull:
        nd = nd.lft
    return nd

def lrRotation(self, x):
    y = x.rgt
    x.rgt = y.lft

    if y.lft != self.tnull:
        y.lft.prt = x

    y.prt = x.prt

    if x.prt == None:
```

```
            self.root = y
        elif x == x.prt.lft:
            x.prt.lft = y
        else:
            x.prt.rgt = y

        y.lft = x
        x.prt = y

    def rlRotate(self, x):
        y = x.lft
        x.lft = y.rgt

        if y.rgt != self.tnull:
            y.rgt.prt = x

        y.prt = x.prt

        if x.prt == None:
            self.root = y
        elif x == x.prt.rgt:
            x.prt.rgt = y
        else:
            x.prt.lft = y

        y.rgt = x
        x.prt = y

    def insert(self, rideNo):

        nd = Node(rideNo)
        nd.prt = None
        nd.data = rideNo
        nd.lft = self.tnull
        nd.rgt = self.tnull
        nd.clr = 1

        y = None
        x = self.root
        while x != self.tnull:
            y = x
            if nd.data < x.data:
                x = x.lft
            elif nd.data > x.data:
                x = x.rgt
            else:
                pass

        nd.prt = y
        if y == None:
            self.root = nd
```

```
        elif nd.data < y.data:
            y.lft = nd
        else:
            y.rgt = nd

        if nd.prt == None:
            nd.clr = 0
            return

        if nd.prt.prt == None:
            return

        self.insertRotationHelper(nd)

    def get_root(self):
        return self.root

    def delete_node(self, data):
        self.deleteNode(self.root, data)

    def pretty_print(self):
        self.__print_helper(self.root, "", True)

def insertRide(redBlackTree,minHeap,keyValueDict,rideNumber,rideCost,tripDuration):
    try:

        x=keyValueDict[rideNumber]
        return 0
    except:
        keyValueDict[rideNumber]=[rideCost,tripDuration]
        redBlackTree.insert(rideNumber)
        minHeap.insert(Ride(rideNumber,rideCost,tripDuration))
        return 1
def printRide(redBlackTree,minHeap,keyValueDict,rideNumber):
    try:

        return [rideNumber,keyValueDict[rideNumber][0],keyValueDict[rideNumber][1]]
    except:
        return [0,0,0]
def printRides(root,minHeap,keyValueDict,x,rideNumber1,rideNumber2):
    try:
        printRides(root.lft,minHeap,keyValueDict,x,rideNumber1,rideNumber2)
        if root.data >= rideNumber1 and root.data <= rideNumber2:
            x.append([root.data,keyValueDict[root.data][0],keyValueDict[root.data][1]])
        else:
            pass
        printRides(root.rgt,minHeap,keyValueDict,x,rideNumber1,rideNumber2)
    except:
        pass
def getNextRide(redBlackTree, minHeap, keyValueDict):
```

```
    x = minHeap.extract_min()
    try:
        p = x.rideNumber
        redBlackTree.delete_node(p)
        keyValueDict.pop(p)
        return [x.rideNumber,x.rideCost,x.tripDuration]
    except:
        return "No active ride requests"

def cancelRide(redBlackTree,minHeap,keyValueDict,rideNumber):

    try:

        redBlackTree.delete_node(rideNumber)

minHeap.delete(Ride(rideNumber,keyValueDict[rideNumber][0],keyValueDict[rideNumber][1]))
        keyValueDict.pop(rideNumber)
    except:
        pass

def updateRide(redBlackTree,minHeap, keyValueDict, rideNumber, newTD):
    try:
        x = keyValueDict[rideNumber][1]
        if newTD < x:

minHeap.delete(Ride(rideNumber,keyValueDict[rideNumber][0],keyValueDict[rideNumber][1]))
            minHeap.insert(Ride(rideNumber,keyValueDict[rideNumber][0],newTD))
            keyValueDict[rideNumber][1]=newTD
        elif newTD > x and newTD < 2* x:

minHeap.delete(Ride(rideNumber,keyValueDict[rideNumber][0],keyValueDict[rideNumber][1]))
            minHeap.insert(Ride(rideNumber,keyValueDict[rideNumber][0]+10,newTD))
            keyValueDict[rideNumber][0]=keyValueDict[rideNumber][0]+10
            keyValueDict[rideNumber][1]=newTD
        elif newTD>2*x:

minHeap.delete(Ride(rideNumber,keyValueDict[rideNumber][0],keyValueDict[rideNumber][1]))
            redBlackTree.delete_node(rideNumber)
            keyValueDict.pop(rideNumber)
        else:
            pass
    except:
        pass
if __name__ == "__main__":
    redBlackTree = RedBlackTree()
    minHeap = MinHeap()
    keyValueDict=dict()
    fread = open(sys.argv[1],'r')
    fwrite = open("output_file.txt", 'w')
    flag = 1
    while flag:
```

```
        line = fread.readline()

        if "Insert" in line:
            cmdArgs = line[7:-2:]
            argList = cmdArgs.split(",")
            rideNumber=int(argList[0])
            rideCost = int(argList[1])
            tripDuration = int(argList[2])
            flag =
insertRide(redBlackTree,minHeap,keyValueDict,rideNumber,rideCost,tripDuration)
            if flag==0:
                fwrite.write("Duplicate RideNumber\n")

        if "GetNextRide" in line:

            l = getNextRide(redBlackTree,minHeap,keyValueDict)

            if "No active ride requests"==l:
                fwrite.write(l + "\n")
            else:
                fwrite.write("("+str(l[0])+","+str(l[1])+","+str(l[2])+")\n")

        if "Print" in line:
            cmdArgs = line[6:-2:]
            argList = cmdArgs.split(",")

            if len(argList)==1:
                l=printRide(redBlackTree,minHeap,keyValueDict,int(argList[0]))
                fwrite.write("("+str(l[0])+","+str(l[1])+","+str(l[2])+")\n")

            else:

                p = []

printRides(redBlackTree.root,minHeap,keyValueDict,p,int(argList[0]),int(argList[1]))
                string = ""

                if len(p)==0:
                    fwrite.write("(0,0,0)\n")

                else:

                    for i in p:
                        string += "("+str(i[0])+","+str(i[1])+","+str(i[2])+")" + ","
                    fwrite.write(string[:-1] + "\n")

        if "UpdateTrip" in line:
            cmdArgs = line[11:-2:]
            argList = cmdArgs.split(",")
            updateRide(redBlackTree,minHeap,keyValueDict,int(argList[0]),int(argList[1]))
```

```
    if "Cancel" in line:
        cmdArgs = line[11:-2:]
        argList = cmdArgs.split(',')
        cancelRide(redBlackTree,minHeap,keyValueDict,int(argList[0]))

    if line == "":
        flag = 0

fwrite.close()
fread.close()
```