Advanced Data Structures Programming Project Spring 2023

Submission Date: April 12, 2023

COP5536 - Professor Sartaj Sahni

Developed by:
Ananya Agrawal [6284-0633]
(ananya.agrawal@ufl.edu)

Table of Contents

Problem Definition

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

A ride is identified by the following triplet:

1. **rideNumber:** unique integer identifier for each ride.
2. **rideCost:** The estimated cost (in integer dollars) for the ride.
3. **tripDuration:** the total time (in integer minutes) needed to get from pickup to destination.

The needed operations are:

1. **Print(rideNumber)** prints the triplet (rideNumber, rideCost, tripDuration).
2. **Print(rideNumber1, rideNumber2)** prints all triplets (rx, rideCost, tripDuration) for which rideNumber1 <= rx <= rideNumber2.
3. **Insert (rideNumber, rideCost, tripDuration)** where rideNumber differs from existing ride numbers.
4. **GetNextRide()** When this function is invoked, the ride with the lowest rideCost (ties are broken by selecting the ride with the lowest tripDuration) is output. This ride is then deleted from the data structure.
5. **CancelRide(rideNumber)** deletes the triplet (rideNumber, rideCost, tripDuration) from the data structures, can be ignored if an entry for rideNumber doesn’t exist.
6. **UpdateTrip(rideNumber, new_tripDuration)** where the rider wishes to change the destination, in this case,

a) if the new_tripDuration <= existing tripDuration, there would be no action needed.

b) if the existing_tripDuration < new_tripDuration <= 2*(existing tripDuration), the driver will cancel the existing ride and a new ride request would be created with a penalty of 10 on existing rideCost. 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.

To complete the given task, use a min-heap and a Red-Black Tree (RBT), assuming that the number of active rides **will not exceed 2000**.

A **min heap** should be used to store (rideNumber, rideCost, tripDuration) triplets ordered by **rideCost**. If there are multiple triplets with the **same rideCost**, the one with the **shortest tripDuration** will be given higher priority (given all rideCost-tripDuration sets will be **unique**). An **RBT** should be used to store (rideNumber, rideCost, tripDuration) triplets ordered by **rideNumber** by maintaining pointers between corresponding nodes in the min-heap and RBT.

GatorTaxi can handle only one ride at a time. When it is time to select a new ride request, the ride with the lowest rideCost(ties are broken by selecting the ride with the lowest tripDuration) is selected (Root node in min heap). When no rides remain, return a message "No active ride requests".

Print(rideNumber) query should be executed in O(log(n)) and Print(rideNumber1,rideNumber2) should be executed in O(log(n)+S) where n is the number of active rides and S is the number of triplets printed. For this, your search of the RBT should enter only those subtrees that may possibly have a ride in the specified range. All other operations should take O(log(n)) time.

Input/Output Testcases

Input Format:

Insert(rideNumber, rideCost, tripDuration)
Print(rideNumber)
Print (rideNumber1,rideNumber2)
UpdateTrip(rideNumber, newTripDuration)
GetNextRide()
CancelRide(rideNumber)

INPUT 1	**OUTPUT 1**

Insert(25,98,46)
GetNextRide()
GetNextRide()
Insert(42,17,89)
Insert(9,76,31)
Insert(53,97,22)
GetNextRide()
Insert(68,40,51)
GetNextRide()
Print(1,100)
UpdateTrip(53,15)
Insert(96,28,82)
Insert(73,28,56)
UpdateTrip(9,88)
GetNextRide()
Print(9)
Insert(20,49,59)
Insert(62,7,10)
CancelRide(20)
Insert(25,49,46)
UpdateTrip(62,15)
GetNextRide()
Print(1,100)
Insert(53,28,19)
Print(1,100)
	(25,98,46)
	No active ride requests
	(42,17,89)
	(68,40,51)
	(9,76,31),(53,97,22)
	(73,28,56)
	(0,0,0)
	(62,17,15)
	(25,49,46),(53,97,15),(96,28,82)
	Duplicate RideNumber

INPUT 2	OUTPUT 2

Insert(5,50,120)
Insert(4,30,60)
Insert(7,40,90)
Insert(3,20,40)
Insert(1,10,20)
Print(2)
Insert(6,35,70)
Insert(8,45,100)
Print(3)
Print(1,6)
UpdateTrip(6,75)
Insert(10,60,150)
GetNextRide()
CancelRide(5)
UpdateTrip(3,22)
Insert(9,55,110)
GetNextRide()
UpdateTrip(6,95)
Print(6)
Print(5,9)
GetNextRide()
CancelRide(7)
Print(7)
Insert(11,70,170)
GetNextRide()
Insert(12,80,200)
Print(12)
UpdateTrip(11,210)
GetNextRide()
CancelRide(14)
UpdateTrip(12,190)
Insert(13,70,220)
GetNextRide()
Insert(14,100,40)
UpdateTrip(14,100)
CancelRide(12)
Print(11,14)
GetNextRide()
Insert(15,20,35)
	(0,0,0)
	(3,20,40)
	(1,10,20),(3,20,40),(4,30,60),(5,50,120),(6,35,70)
	(1,10,20)
	(3,20,22)
	(6,55,95)
	(6,55,95),(7,40,90),(8,45,100),(9,55,110)
	(4,30,60)
	(0,0,0)
	(8,45,100)
	(12,80,200)
	(6,55,95)
	(9,55,110)
	(11,80,210),(13,70,220)
	(10,60,150)
	(0,0,0)
	(11,80,210),(13,70,220),(15,20,35)
	(15,20,35)
	(13,70,30)
	(13,70,30)
	(0,0,0)
	(17,70,25)
	(11,80,210)
	(16,70,60)
	(0,0,0)
	(12,40,30),(16,80,82),(20,16,75)
	(20,16,75)
	(0,0,0)
	(8,60,97)
	(11,80,210)
	(16,90,124)
	(15,90,85)
	(23,49,46)
	(1,56,85),(7,125,54)
	(7,125,54)
	No active ride requests
	(17,12,37)
	(0,0,0)

Print(14)
Print(10,16)
GetNextRide()
UpdateTrip(13,30)
Print(13)
GetNextRide()
Print(12)
CancelRide(19)
Insert(16,60,45)
Insert(17,70,25)
UpdateTrip(16,60)
GetNextRide()
Print(11)
Print(16,18)
Insert(18,65,130)
Insert(12,40,30)
Insert(8,60,97)
UpdateTrip(16,82)
Insert(20,16,75)
UpdateTrip(18,300)
Print(23)
Print(12,21)
CancelRide(12)
GetNextRide()
CancelRide(25)
Print(20,26)
GetNextRide()
UpdateTrip(16,124)
Insert(7,125,54)
GetNextRide()
Print(16)
Insert(22,80,85)
Insert(15,90,85)
UpdateTrip(22,195)
GetNextRide()
Insert(23,49,46)
Insert(1,56,85)
UpdateTrip(16,300)
GetNextRide()
Print(1,30)
CancelRide(1)
	Duplicate RideNumber

GetNextRide()
GetNextRide()
Insert(24,21,46)
Insert(17,12,37)
GetNextRide()
Print(16)
Insert(24,80,85)
Insert(15,90,85)
CancelRide(28)
UpdateTrip(23,450)
GetNextRide()
Print(24)
Print(22,26)
CancelRide(29)
Print(28)	

Program Structure

```
Project Root:
├── ride.py
    └── Ride
        ├── __init__(rn, rc, td) -> None
        └── __str__() -> str
├── rbtree.py
    └── Node
        ├── __init__(Ride) -> None
        ├── __eq__(other) -> bool
        ├── __repr__() -> str
        ├── GetColor() -> str
        ├── SetColor(color) -> None
        ├── RedNode() -> bool
        ├── BlackNode() -> bool
        └── IsNull() -> bool
    └── RedBlackTree
        ├── __init__() -> None
        ├── RideRangeSearcher(node, rangeStart, rangeEnd) -> None
        ├── TreeSearcher(node, key) -> Node
        ├── FixDelete(x) -> None
        ├── RedBlackSwitch(u, v) -> None
        ├── NodeDeleter(node, key) -> None
        ├── InsertFix(k) -> None
        ├── RangeSearch(s, e) -> str
        ├── Search(rideNumber) -> Node
        ├── Minimum(node) -> Node
        ├── RotateLeft(x) -> None
        ├── RotateRight(x) -> None
        ├── Insert(ride) -> None
        ├── Delete(item) -> None
        ├── __getitem__(key) -> int
        └── __setitem__(key, value) -> None
├── minheap.py
    └── Minheap
        ├── __init__(heap) -> None
        ├── CompareRide(ride_1, ride_2) -> bool
        ├── AdjustDown(start, end) -> None
        ├── AdjustUp(pos) -> None
        ├── PushHeap(item) -> None
        ├── PopHeap() -> Ride
        └── Heapify() -> None
├── data_structs.py
    └── CommonDataHandler
        ├── __init__() -> None
        ├── Print(rideNumber) -> Ride
        ├── PrintRange(ride_number_start, ride_number_end) -> str
        ├── Insert(ride) -> None
        ├── GetNextRide() -> Ride
        ├── CancelRide(rideNumber) -> None
        └── UpdateTrip(rideNumber, new_duration) -> None
    └── RBT
        ├── __init__() -> None
        ├── Insert(ride) -> None
        ├── Remove(ride) -> None
        ├── SearchRide(rideNumber) -> Ride
        └── SearchRideRange(start, end) -> str
    └── MinHeap
        ├── __init__() -> None
        ├── Insert(ride) -> None
        ├── Remove() -> Ride
        └── Delete(rideNumber) -> None
├── operation_type.py
    └── OperationType
├── file_parser.py
    └── FileParser
        ├── __init__(filename) -> None
        ├── ParseParam(line) -> list
        ├── ParseLine(line) -> OperationType, list
        ├── ParseFile() -> list
        ├── AppendToOutput(line) -> None
        └── CloseAllFiles() -> None
├── gatorTaxi
└── Makefile
```

File 1: ride.py

Class Ride:

Data class to hold the ride triplet value.

__init__(rn, rc, td) → None

Initializes the ride triplet, rideNumber as rn, rideCost as rc, tripDuration as td.

__str__() → str

Python inbuilt function to convert the ride object datatype to a string datatype

File 2: rbtree.py

Class Node:

Class to create a node as a ride object in a Red Black Tree

__init__(Ride) → None

Initializes the node, its index, parent of the node, left child of the node, right child of the node, color of the node, and value of the node as None, and the id of the node itself as 1.

__eq__(other) → bool

Python inbuilt function to compare the node id with id of other node and return a boolean value.

__repr__() → str

Python inbuilt function to return a printable representational string of the ride object as the node’s id along with its value.

GetColor() → str

Function to return the color of the node as Black in string datatype if the node is assigned 0 in integer datatype and as Red, otherwise

SetColor(color) → None

Function to assign the color of a node as 0, if its value is Black and 1 if the value is Red. In all other cases, it raises an exception for Unknown color.

RedNode() → bool

Function to return True if the color of the node is assigned 1 corresponding to Red color.

BlackNode() → bool
Function to return True if the color of the node is assigned 0 corresponding to Black color.

IsNull() → bool
Function to return a boolean value if the node is not present in the Red Black Tree by checking its id assignment as -1.

Class RedBlackTree:

Class to initialize and create a Red Black Tree using the Node class. It is a base class that stores the ride objects and orders them by ride numbers.

__init__() → None
Python constructor to initialize the null node as a Ride object having -1 as the triplet value, along with an id of -1 and color as black. This ride object is used as the initial root of the tree while setting the tree size parameter as 0.

RideRangeSearcher(node, rangeStart, rangeEnd) → None
Function to perform selective inorder traversal by scanning the tree to find rides within a given range of ride numbers.

TreeSearcher(node, key) → Node
Function to search a node in the Tree, and return it when found. The node variable is used to maintain the search hierarchy and the key denotes the ride number which we are searching for. The search operation takes place like in a Binary Search Tree, if the key is smaller than the root, then the search is done in the left subtree, or else the search is done in the right subtree. This comparison occurs for each parent node and thus the search time is halved. If the node is found at the root of the tree, then that node is returned. If the node is not found in the tree, then the search fails and returns the null node.

FixDelete(x) → None
Helper function to re-balance the tree after deletion. As long as the deleted node is not at the root of the tree and is Black colored, the tree rebalances. After a node is deleted from the tree, this function helps to update the node color value and rebalance the tree to maintain the Red Black Tree properties.

RedBlackSwitch(u, v) → None
This helper function takes two input nodes and helps to switch their positions to maintain the Red Black Tree properties.

NodeDeleter(node, key) → None

Function to first search for a node as in a Binary Search Tree manner and remove it from the tree while maintaining the Red Black Tree properties for node color reassignment and rebalancing.

InsertFix(k) → None
Helper function to restructure the tree after inserting a new node into the Red Black Tree. After a node is inserted this function helps to update the color value of the node and rebalances the tree to maintain the Red Black Tree properties.

RangeSearch(s, e) → str
Helper function which accepts the start and end of the range and calls the RideRangeSearcher function to search for ride nodes within the range if any, else returns a string with 0, 0, 0 for found triplets. It returns the string representation of the search result.

Search(rideNumber) → Node
Helper function that searches for a ride node with respect to an input rideNumber and calls the TreeSearcher function to perform the actual search.

Minimum(node) → Node
Function to find the node with the minimum value by traversing left starting at the root, as in a Binary Search Tree until we reach the leaf node.

RotateLeft(x) → None
Function to perform a left rotation operation as in a Binary Search Tree.

RotateRight(x) → None
Function to perform a right rotation operation as in a Binary Search Tree.

Insert(ride) → None
Helper function to create a node of the ride as insert it into the Red Black Tree using the InsertFix function.

Delete(item) → None
Helper function to delete a node from the Red Black Tree by using the NodeDeleter function.

__getitem__(key) → int, __setitem__(key, value) → None
Python inbuilt getter and setter function to get the value based on the key input and set the value with respect to the key within the tree.

File 3: minheap.py

Class Minheap:

Class to implement a minheap using a list. It is a base class that stores the ride objects and orders them by ride cost and trip duration.

__init__(heap) → None

Constructor to initialize the heap using a list.

CompareRide(ride_1, ride_2) → bool

Helper function to compare the ride costs and in case of a tie, compare the rides based on their trip duration. This serves as the basis of the structure of the heap as a min heap by comparing the properties of rides.

AdjustDown(start, end) → None

Function to restructure the heap after an insertion or deletion by comparing the child with their parent and swapping if necessary to maintain the minheap condition.

AdjustUp(pos) → None

Function to adjust the item at a given position and correct its position in the following subtree recursively.

PushHeap(item) → None

Function to append the item into the minheap and adjust its position to maintain minheap properties.

PopHeap() → Ride

Function to remove an item from the minheap, usually the minimum value is at the root of the minheap, so this function removes the root and swaps the root with the last item. Then the heap is restructured to maintain the minheap property.

Heapify() → None

Function to create a priority queue from the minheap by traversing the heap and appending values from root to last item and restructuring the heap for every remove operation after appending the item to the list.

File 4: data_structs.py

Class CommonDataHandler:

This class maintains the parity between the Red Black Tree and the MinHeap for simultaneous operations.

__init__() → None

Constructor to initialize the minheap and the red-black tree.

Print(rideNumber) → Ride

Function to define the 1st operation for printing the ride triplet after searching for it in the red-black tree with respect to a given ride number.

PrintRange(ride_number_start, ride_number_end) → str

Function to define the 2nd operation for printing all rides within a given input range by using the SearchRideRange helper function defined in the red-black tree for scanning the tree using selective inorder traversal.

Insert(ride) → None

Function to define the 3rd operation for inserting a new ride in the red-black tree as well in the minheap simultaneously. In case a ride exists already, it raises an exception of Duplicate RideNumber and the program halts its execution.

GetNextRide() → Ride

Function to define the 4th operation for getting the next ride with the lowest ride cost, in case of a tie, the trip duration of the respective rides are compared to find the next ride. This ride is then deleted simultaneously from the min heap and the red-black tree.

CancelRide(rideNumber) → None

Function to define the 5th operation for deleting a ride entry from the minheap and the red-black tree simultaneously after searching for the given ride in the tree using its ride number and checking if it originally exists in the tree before removal.

UpdateTrip(rideNumber, new_duration) → None

Function to define the 6th operation for updating a trip based on the following conditions:

(1) If the duration of the new trip is lesser than the original trip, then the ride is updated with the existing ride number and ride cost and new duration.

(2) If the new duration is more than the existing duration but lesser than twice the original trip duration, then the original ride is cancelled and a new ride is inserted with an updated cost with a penalty of 10 on the existing ride cost. The ride number remains the same as the original, the trip duration is updated to the new trip duration.
(3) If the new duration is more than twice the existing duration, then the ride is cancelled.

Class RBT:

This class initializes the base Red Black Tree class for operations such as insertion, removal, searching a ride and searching a ride within a range.

__init__() → None

Constructor to initialize the red-black tree.

Insert(ride) → None

Function to insert a ride into the red-black tree.

Remove(ride) → None

Function to delete the ride from the red-black tree.

SearchRide(rideNumber) → Ride

Function to search for a given ride based on an input ride number in the tree.

SearchRideRange(start, end) → str

Function to search for all rides in the given range in the red-black tree.

Class MinHeap:

This class initialized the base Minheap class for operations such as insertion, deletion, push, pop and heapify.

__init__() → None

Constructor to initialize the minheap.

Insert(ride) → None

Function to push a ride into the heap and restructure the heap.

Remove() → Ride

Function to pop a ride from the root node of the min heap and restructure the heap. In case the heap is empty, a ride object with all values as -1 is returned.

Delete(rideNumber) → None
Function to delete the ride from the minheap on the basis of the input ride number and restructure the heap.

File 5: operation_type.py

Class OperationType:
This class uses the Enum library in Python to create enumerations for the listed operations such as Print, Insert, GetNextRide, etcetera. These enumerations are a set of symbolic names that are unique and constant.

File 6: file_parser.py

Class FileParser:
This class handles the file handling for reading from an input file and writing to an output file.

__init__(filename) → None
Constructor to open an input file in read mode and an output file in writing and creating mode.

ParseParam(line) → list
Function to find the parameters from an operation string by using substring and returning it as a list of integers.

ParseLine(line) → OperationType, list
Function to parse each line from an input file and separate the functions by matching keywords with respect to the listed operations.

ParseFile() → list
Function to parse the input file and return a list of each line present.

AppendToOutput(line) → None
Utility function to write a line to the output file.

CloseAllFiles() → None
Utility function to close input and output file buffers after using them.

File 7: gatorTaxi

Main:

In this file, we read the command line arguments and check if the user has provided an input file. If yes, we read from the input file and all the previously defined functions from various files are used to handle the ride requests and perform operations as given in the input file.

File 8: Makefile

Default:

Makefile operation to set the proper executable permission for the gatorTaxi python script.

Space & Time Complexities

Red Black Tree:

Overview:

OPERATION	AVERAGE CASE	WORST CASE
Search	O(log n)	O(log n)
Insert	O(log n)	O(log n)
Delete	O(log n)	O(log n)

Insert: The Insert operation takes a ride as an input and converts it to a Node object. This ride node is initially set as a Black colored node in the red-black tree and becomes the root of the tree. Further inserts require restructuring of the tree and changing the colors of the nodes to align with the properties of a red-black tree.

The insertion in a red-black tree is like the insertion in a binary search tree. As the red-black tree is a height-balanced tree, thus the insert operation will take O(height of tree) time which is equal to O(log n) time. The color-changing step requires O(1) time.

In the worst case, if an insertion is done such that, two red nodes wind up consecutively in a path from the root to leaf, then the recoloring has to be performed for the entire path from the leaf to root. Thus the insertion will get completed in O(log n) time with rebalancing, recoloring, and rotation operations.
Thus the overall time complexity for an insert operation in a red-black tree is O(log n).

Search: The Search operation takes an input ride number or a range of ride numbers and searches for the respective ride in the red-black tree. The search is done by recursively searching in either the left subtree or the right subtree and comparing the key value with the ride number.

As this operation reduces the number of comparisons made at each step by half, thus for a tree of height h, containing, n = 2^h nodes, the total time complexity becomes O(log n) in the worst case where the node is found at the leaf of the tree and O(1) in the best case where the node is found at the root.

For doing a range search, a start and an end are provided and the tree is searched for only those nodes that fall into this range. This selective inorder traversal reduces the number of comparisons to be made for a range search and then each node that falls into that range is searched like in a Binary Search Tree. Therefore the time to search for each node that falls into the range selectively is the same as for a regular search operation O(log n) and an additional time to select only those nodes for searching the tree that have a ride in the specified range takes S time. So the overall time complexity for a range search becomes O(log (n) + S). Here, S is the number of ride triplets in the given range.

Delete: The Delete operation in a red-black tree deletes that ride from the tree that is cancelled and removes the node. After the removal of a node in the red-black tree, the tree is restructured and re-colored using necessary rotations to fulfil the red-black tree properties.

The delete operation can take O(log n) time in the worst case, where the node deleted can be a leaf node, and the restructuring is followed up to the path to the root from the deleted leaf node. In the best case, where the root is the node to be removed, it can take O(1) time, as no restructuring will be required, and color swaps might take O(1) time as well.

Space Complexity of a Red-Black Tree: The space complexity of a red-black tree can be O(n) where the tree can be output to a list using an inorder traversal (left subtree, root, right subtree). It is the same as that of a Binary Search Tree. There will be no duplicate data in the tree therefore no extra space is required. Each node takes up to O(1) space thus for n total nodes, O(n) is required. The coloring information can be stored with the node in O(1) space as well using binary notation, 0 for black and 1 for red.

Min Heap:

Overview:

OPERATION	AVERAGE CASE	WORST CASE
Search	O(n)	O(n)

Insert	O(log n)	O(log n)
Delete	O(log n)	O(log n)

Insert: The Insert operation in a minheap inserts a ride object into the heap. The insertion takes place at the leaf position of the heap and subsequent inserts require the heap to restructure after each insert to align with the properties of a minheap.

In the worst case, for every node insertion at the leaf node of the heap, the heap has to restructure as the inserted node is smaller than its parent node. This takes O(log n) time where the inserted value has to shift up to the root from the leaf as it is the smallest element in that heap.

To reach the root, following a binary order traversal, we have to move up one parent each time for each level, as there are log n levels thus the time complexity becomes O(log n). In the best case, the inserted node is the maximum value thus it remains at the leaf and no restructuring is required, thus time complexity becomes O(1).

Search: The Search operation in a minheap is done linearly after the heapify operation, where all the elements of the heap are appended to a priority queue, in the case of a minheap, a min priority queue is formed by traversing from the root of the minheap, swapping the root with the last leaf element and restructuring at each step until all root elements are appended to the min priority queue in ascending order.

As the heap contains n elements that are appended to a priority queue, the linear search is done on the priority queue containing n elements in O(n) time.

In the worst case all n elements have to be searched if the element we are searching for is at the end of the priority queue, thus taking O(n) time, whereas in the best case, the element we are searching for is present at the root of the minheap, thus at the beginning of the queue, hence time complexity becomes O(1).

Delete: The Delete operation will remove the ride with the lowest ride cost, given that the minheap is structured on the basis of ride cost and trip duration, in case of a tie. As the node with the lowest ride cost will be situated at the root of the heap, therefore a pop operation will remove the root node from the heap and the leaf is swapped with the root, thus restructuring the minheap again.

In the worst case, the heap takes log n moves to restructure after popping the root and swapping it with the leaf, thus taking O(log n) time. In the best case, the heap only has 1 element which is the root and popping the root will result in no restructuring required, thus taking O(1) time.

Space Complexity of a Min Heap: The minheap when heapified, appends itself into a min priority queue containing n elements, therefore the space required is O(n). While pushing elements into the heap, a list of elements are required to insert into the leaf of the heap followed by a restructuring of the heap to align with the properties of a minheap, thus the overall space complexity is O(n) for a minheap.

Operations:

1. **Print:** The print operation first searches for the ride node in a red-black tree on the basis of an input ride number, as the red-black tree is ordered on the basis of the ride number. Since the search operation in a red-black tree takes O(log n) time, thus the print operation is executed in O(log n) time as well.

2. **PrintRange:** The print range operation does a selective search for the nodes present within an input range of ride numbers. This is done by scanning the tree using a selective inorder traversal in a red-black tree. As the range search operation first decides which leaf to explore and later traverses to find all the rides that may be present in the input range on the red-black tree, therefore, the overall time required to execute the print range operation is O(log (n) + S), where S is the number of ride triplets that are present within the specified range and n is the number of active rides.

3. **Insert:** The insert operation inserts the ride object into the minheap and the red-black tree simultaneously after checking that a duplicate ride number does not already exists in the tree or the heap. Since the insert operation in both the minheap and the red-black tree takes O(log n) time, therefore, the insert operation takes O(log n) time. For checking if a ride does not previously exist in the red-black tree we do a search operation in the red-black tree that also takes O(log n) time, thus collectively the time complexity for the insert operation remains O(log n).

4. **GetNextRide:** The get next ride operation, outputs the next ride from the minheap where the ride cost is the lowest given that the minheap is ordered by ride cost. In the case of a tie, the trip durations are compared, and the ride with the lowest trip duration is output. The ride is then deleted from the minheap and simultaneously from the red-black tree as well. Since the deletion operation in both the minheap and the red-black tree takes O(log n) time, thus the time complexity for the get next ride operation takes O(log n) time.

5. **CancelRide:** The cancel ride operation deletes the ride triplet from the min heap and the red-black tree simultaneously. As the deletion operation in both the minheap and the

red-black tree takes O(log n) time, therefore the cancel ride operation also takes O(log n) time.

6. **UpdateTrip:** The update trip operation is subdivided into 3 cases:
   (a) When the new trip duration is smaller than the existing trip duration, then the previous ride is cancelled thus invoking a delete operation from both minheap and red-black tree and a new ride node is inserted into both the minheap and the red-black tree with the original ride number and ride cost and the new trip duration. Both insertion and deletion operations in both the data structures take O(log n) time, thus this operation also takes O(log n) time.
   (b) When the new trip duration is greater than the existing trip duration but smaller than twice the existing trip duration, then the existing ride is cancelled thus invoking a delete operation in both the min heap and the red-black tree followed by insertion of a new ride node with existing ride number, a penalty of 10 on the existing ride cost and the new trip duration. Both insertion and deletion operations in both the data structures take O(log n) time, thus this operation also takes O(log n) time.
   (c) When the new trip duration is greater than twice the existing trip duration, the ride is cancelled and the ride object is removed from both the data structures, as the deletion operation in both the min heap and the red-black tree takes O(log n) time, thus this operation also takes O(log n) time.

   Therefore, all the suboperations in the update trip operation take O(log n) time, therefore the update trip operation takes a total of O(log n) time.