

Ride-sharing Taxi Service

Aim:

The aim of the project is to develop a ride-sharing taxi service.

Language Used:

Java

Input:

The input to the program is an “input.txt” file which consists of the operations and the ride numbers, ride cost and trip duration.

The input data will be given in the following format:

Insert(rideNumber, rideCost, tripDuration)

Print(rideNumber)

Print (rideNumber1,rideNumber2)

UpdateTrip(rideNumber, newTripDuration)

GetNextRide()

CancelRide(rideNumber)

How to run:

Step 1: Unzip the file

Step 2: Open the command prompt

Step 3: Enter into the file using cd

Step 4: Execute the following command:

make

Step 5: After the compilation is done, run the following command:

java gatorTaxi input.txt

Working Directory Structure:

The submitted file after unzipping contains the following files:

- gatorTaxi.java
- MinHeap.java
- RedBlackTree.java
- Node.java
- makefile

Class and Class Definitions:

There are 4 main classes as below:

1. **Node:** This class is used to store the node elements which is used by the red black tree and min heap. It consists of the following variables-
 - a. rideNum- to store the number of the ride

- b. rideCost- to store the cost of the ride
- c. rideDur- to store the duration of the ride
- d. color- to store the color of the node which is 0 for black and 1 for red
- e. indexOfHeap- to store the index of the heap
- f. left- to store the left child of the node
- g. right- to store the right child of the node
- h. parent- to store the parent of the node

```
J Node.java > ...
1  public class Node {
2      int rideNum;
3      int rideCost;
4      int rideDur;
5      Node left;
6      Node right;
7      int color;
8      Node parent;
9      int indexOfHeap;
10 }
```

2. **gatorTaxi:** This class reads the input file line by line to perform the operations like Insert, GetNextRide, Print, UpdateTrip and CancelRide by calling their respective functions from their respective class. It stores the output onto another output file.

```
J gatorTaxi.java > ...
1  import java.util.ArrayList;
2  import java.io.BufferedReader;
3  import java.io.IOException;
4  import java.io.FileReader;
5  import java.io.FileWriter;
6
7  public class gatorTaxi {
8      Run | Debug
9  >  public static void main(String[] args) { ...
154 }
```

3. **MinHeap:** This class contains the min heap implementation. It contains functions to insert a ride, delete a node and getting the next ride.

```

J MinHeap.java > ...
1  public class MinHeap {
2      Node heap_array[] = null;
3      int heap_capacity;
4      int heap_limit;
5
6  >  MinHeap(int limit) { ...
11
12 >  void swap(Node a, Node b) { ...
19
20 >  boolean checkHeapCondition(int i, int j) { ...
26
27 >  void heapifyAboveRoot(int index) { ...
33
34 >  boolean insertNode(Node node) { ...
47
48 >  void heapifyFromRoot(Node node) { ...
62
63 >  void deleteRide(int index) { ...
69
70 >  int getNextRide() { ...
86 }

```

4. **RedBlackTree:** This class contains the red black tree implementation. It contains functions to insert a ride, delete a ride and search for a ride.

```

J RedBlackTree.java > ...
1  import java.util.ArrayList;
2
3  public class RedBlackTree {
4
5      public Node root, dummy;
6
7  >  public RedBlackTree() { ...
14
15 >  public Node getLeafNode() { ...
18
19 >  public void leftRotate(Node leftNode) { ...
38
39 >  public void rightRotate(Node rightNode) { ...
58
59 >  public Node assignNodeElements(int num, int cost, int dur) { ...
70
71 >  public Node insertRide(int num, int cost, int dur) { ...
149
150 >  public void replace(Node temp1, Node temp2) { ...
163
164 >  public void deleteRide(int taxiNum) { ...
274
275 >  public Node lookIntoRide(Node taxiNode, int taxiNum) { ...
281
282 >  public Node lookForRide(int taxiNum) { ...
285
286 >  public ArrayList<String> lookIntoRide(Node taxiNode, int ride1, int ride2) { ...
300
301 >  public ArrayList<String> lookForRide(int ride1, int ride2) { ...
304 }

```

Function Prototypes:

1. **MinHeap.java** file contains the following functions:

- a. MinHeap(int limit)
 - It is a constructor function for the class where heap_limit is assigned to the given variable limit and heap_array is declared which the size of limit.
 - This function returns nothing.
 - b. swap(Node a, Node b)
 - It is a helper function for the class to swap the nodes a and b.
 - This function returns nothing.
 - c. checkHeapCondition(int I, int j)
 - It is a function which checks if the min heap property is violated or not.
 - This function returns Boolean value of true if the heap property is violated or false if the heap property is satisfied.
 - d. HeapifyAboveRoot(int index)
 - It is a function which performs heapify operation from the index until the root the heap.
 - This function returns nothing.
 - e. insertNode(Node node)
 - It is a function to insert the node into the heap at the end of heap array.
 - This function returns Boolean value of true if the heap insert is successful or false if the heap insert did not happen.
 - f. heapifyFromRoot(Node node)
 - It is a function which performs heapify operation from the root using recursion.
 - This function returns nothing.
 - g. deleteRide(int index)
 - It is a function which deletes the ride at the given index in the heap array and the last node is put in place of the deleted node.
 - This function returns nothing.
 - h. getNextRide()
 - It is a function which gets the next minimum ride in the heap which is always the root since it is a min heap. This deleted the minimum element.
 - This function returns the ride number.
2. **RedBlackTree.java** file contains the following functions:
- a. RedBlackTree()
 - It is a constructor function for the class where a new dummy node is initialized, left and right children are assigned to null, dummy is made as root and the color is assigned as black.
 - This function returns nothing.
 - b. getLeafNode()
 - It is a function to get the dummy leaf node.
 - This function returns Node.
 - c. leftRotate(Node leftNode)
 - It is a helper function for the red black tree to perform left rotation operation when the RBT properties are violated after insertion and deletion.
 - This function returns nothing.

- d. `rightRotate(Node rightNode)`
 - It is a helper function for the red black tree to perform right rotation operation when the RBT properties are violated after insertion and deletion.
 - This function returns nothing.
 - e. `assignedNodeElements(int num, int cost, int dur)`
 - It is a helper function which assigns the node elements to their values.
 - This function returns the Node after assigning values.
 - f. `insertRide(int num, int cost, int dur)`
 - It is a function to insert the node into the red black tree and balances the tree using `leftRotate` and `rightRotate` functions after insertion.
 - This function returns the inserted node.
 - g. `replace(Node temp1, Node temp2)`
 - It is a helper function to replace the nodes `temp1` with `temp2`.
 - This function returns nothing.
 - h. `deleteRide(int taxiNum)`
 - It is a function which deletes the ride with the given `taxiNum` in the red black tree and balances the tree using `leftRotate` and `rightRotate` functions after deletion.
 - This function returns the deleted node.
 - i. `lookIntoRide(Node taxiNode, int taxiNum)`
 - It is a function to search for the ride using the `taxiNum`.
 - This function returns the node after finding it.
 - j. `lookForRide(int taxiNum)`
 - It is a function which calls the `lookIntoRide` function passing the root node and the `taxiNum` to search for the ride.
 - This function returns the node after finding it.
 - k. `lookIntoRide(Node taxiNode, int ride1, int ride2)`
 - It is a function which gives the list of rides between the range of `ride1` and `ride2`.
 - This function returns a `ArrayList` of string.
 - l. `lookForRide(int ride1, int ride2)`
 - It is a function which calls the `lookIntoRide` function passing the root node and ride numbers as `ride 1`, `ride2` to search for the rides between the range of `ride1` until `ride2`.
 - This function returns a `ArrayList` of string.
3. **gatorTaxi.java** file contains the following function:
- a. `public static void main(String[] args)`
 - It is a function which reads the input file line by line and performs the operations like `insert`, `getnextride`, `print`, `updatetrip`, `cancelride`.
 - This function returns nothing.

Implementation:

The following operations are implemented in the code:

1. **Insert Operation:** Insert(rideNumber, rideCost, tripDuration):
This operation will insert the given ride. The gatorTaxi main function will call the RedBlackTree.insertRide function and MinHeap.insertNode function.
2. **GetNextRide Operation:** GetNextRide():
This operation will get the next minimum ride and delete it. The gatorTaxi main function will call the MinHeap.getNextRide function and RedBlackTree.lookForRide function to search of the ride using the ride number and RedBlackTree.deleteRide function to delete the ride.
3. **Print Operation with single ride number:** Print(Ride Number):
This operation will print the ride with the given ride number. The gatorTaxi main function will call the RedBlackTree.lookForRide function to search for the ride.
4. **Print Operation with two ride numbers:** Print(Ride1, Ride2):
This operation will print the list of rides with the range between the given ride numbers. The gatorTaxi main function will call the RedBlackTree.lookForRide function to search for the rides between the two given ride numbers.
5. **UpdateTrip Operation:** UpdateTrip(rideNumber,new_TripDuration):
This operation will update the given ride. The gatorTaxi main function will call the RedBlackTree.lookForRide function and checks if new trip duration is greater than the existing trip duration. If yes, then RedBlackTree.deleteRide and MinHeap.deleteRide functions are called. If new trip duration is less than twice the old trip duration, then RedBlackTree.insertRide and MinHeap.insertNode functions are called.
6. **CancelRide Operation:** CancelRide(rideNumber):
This operation will search for the given ride number and delete it. The gatorTaxi main function will call the RedBlackTree.lookForRide function to search of the ride using the ride number. If the ride exists, then MinHeap.deleteRide and RedBlackTree.deleteRide functions are called to delete the ride.

Complexity Analysis:

The time and space complexity for the above operations are as follows:

1. **Insert(rideNumber, rideCost, tripDuration):**

Time Complexity: $O(\log n)$

During insertion of a node into red black tree, the ride number is first searched in the tree which consumes $O(\log n)$ time. During insert, the while loop takes $O(\log n)$ time in the worst case scenario while searching for a parent for this node. After insertion, if the tree is not balanced, then to balance it, the function takes $O(\log n)$ time.

During insertion of a node into a min heap, the ride number is first added to the heap array at the last consuming $O(1)$ time. Then, for checking if imbalance occurred and to perform heapify, the function takes $O(\log n)$ time.

Thus, time complexity = $O(\log n) + O(\log n) + O(1) + O(\log n) + O(1) + O(\log n)$
= $O(\log n)$

Space Complexity: $O(1)$

For insertion, we always take constant space for node.

2. **GetNextRide():**

Time Complexity: $O(\log n)$

In the min heap, deleting the minimum node and replacing it with the last node consumes $O(1)$ time. After deletion, to check and perform heapify operation if min heap property is violated, the function takes $O(\log n)$ time.

In red black tree, the delete function takes $O(\log n)$ time. After deletion, if the tree is not balanced, then to balance it, the function takes $O(\log n)$ time.

Thus, time complexity = $O(1) + O(\log n) + O(\log n) = O(\log n)$

Space Complexity: $O(1)$

This operation performs only deletion and does not require further external memory.

3. **Print(Ride Number):**

Time Complexity: $O(\log n)$

In the red black tree, the search function is called where in the worst case scenario, all the nodes in the tree are traversed, thus consuming $O(\log n)$ time.

Space Complexity: $O(1)$

This operation only prints the triplet and does not require further external memory.

4. **Print(Ride1, Ride2):**

Time Complexity: $O(\log n + S)$ where S is the number of the triplets which are printed

In the red black tree, the search function is called where in the worst case scenario, all the nodes in the tree are traversed, thus consuming $O(\log n)$ time. While printing the rides, there is a for loop which iterates S times which is the number of triplets within the range of the given Ride1 and Ride2, taking $O(S)$ time.

Thus, time complexity = $O(\log n) + O(S) = O(\log n + S)$

Space Complexity: $O(S)$ where S is the number of the triplets which are printed

The function stores the rides between the Ride1 and Ride2 in a ArrayList of string where the size is same as the number of elements in the range of the two rides.

5. **UpdateTrip(rideNumber,new_TripDuration):**

Time Complexity: $O(\log n)$

To search for the given ride in the red black tree consumes $O(\log n)$ time.

In the first scenario, after the new duration is updated, the function to perform heapify

operation takes $O(\log n)$ time.

In the second scenario, to perform delete operation in both min heap and red black tree after insert operation consumes $O(\log n)$ time.

In the third scenario, to perform delete operation in both min heap and red black tree takes $O(\log n)$ time.

Thus, time complexity = $O(\log n) + O(1) + O(\log n) + O(\log n) + O(\log n) = O(\log n)$

Space Complexity: $O(1)$

6. **CancelRide(rideNumber):**

Time Complexity: $O(\log n)$

To search for the given ride in the red black tree consumes $O(\log n)$ time.

In the min heap, deleting the minimum node and replacing it with the last node consumes $O(1)$ time. After deletion, to check and perform heapify operation if min heap property is violated, the function takes $O(\log n)$ time.

In red black tree, the delete function takes $O(\log n)$ time. After deletion, if the tree is not balanced, then to balance it, the function takes $O(\log n)$ time.

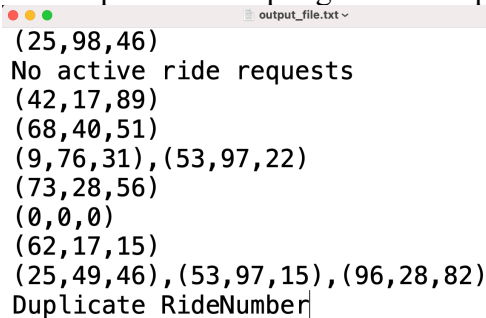
Thus, time complexity = $O(\log n) + O(1) + O(\log n) + O(\log n) + O(\log n) = O(\log n)$

Space Complexity: $O(1)$

This operation needs constant amount of space.

Output: The output will contain the rides and comments in the “output_file.txt” file.

The output for the input given in the project file is as follows:



```
output_file.txt
(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
```

Conclusion: The GatorTaxi ride-sharing taxi service is implemented successfully using Java. Inserting a ride is done and if a duplicate ride number is entered, the comment is outputted. Printing a ride number will print the triplet of ride number, ride cost and trip duration, and if the ride number does not exist then zeroes are printed. Printing with two ride numbers will print the ride numbers in the range between those two ride numbers. Update trip will update the ride and does not print anything. Cancel ride will cancel the ride and does not print anything. GetNextRide will output the ride which has the least cost.