COP 5536 Programming Assignment – 1

Introduction
This project – GatorTaxi – aims at creating a taxi service that can consists of a min heap and a red-black tree. This service can perform functions such as insert, cancel, update, and print which are explained later in this report.

Author
Rishabh Jaiswal
UFID: 2109-5276
rishabh.jaiswal@ufl.edu

How to run
Follow below instructions to run the program:

1. Download Jaiswal_Rishabh.zip
2. Run `*unzip Jaiswal_Rishabh.zip*`
3. Run `*make*`
4. Run `*java gatorTaxi <input filename>*`
5. Run `*cat output_file.txt*` to view the contents of the output file

```
thunder:~/ADS_Project> make
javac -g GatorMinHeap.java
javac -g GatorRedBlackTree.java
javac -g gatorTaxi.java
jar -cvmf manifest.txt gatorTaxi.jar GatorMinHeap.class GatorRedBlackTree.class GatorRide.
class gatorTaxi.class GatorTaxiMain.class GatorTreeNode.class
added manifest
adding: GatorMinHeap.class(in = 2672) (out= 1408)(deflated 47%)
adding: GatorRedBlackTree.class(in = 5287) (out= 2757)(deflated 47%)
adding: GatorRide.class(in = 735) (out= 454)(deflated 38%)
adding: gatorTaxi.class(in = 3573) (out= 2018)(deflated 43%)
adding: GatorTaxiMain.class(in = 3424) (out= 1837)(deflated 46%)
adding: GatorTreeNode.class(in = 528) (out= 352)(deflated 33%)
thunder:~/ADS_Project> java gatorTaxi ../testcase\ 2/input.txt
thunder:~/ADS_Project> ls output_file.txt
output_file.txt
thunder:~/ADS_Project>
```

Components
A ride in the gator taxi service is defined by a triplet given as

ride -> (rideNumber, rideCost, tripDuration)

where,

rideNumber is a unique identifier for a ride,
rideCost is the cost of ride,
tripDuration is the total time it will take to complete the ride.

Function descriptions

1. Print(int rideNumber)
   Prints the ride triplet for the given ride number.

   Time complexity – O(log(n)).
   Reason: to print the ride, we need to search it in the RB tree, whose time complexity if O(log(n)).

   Space complexity – O(1), or O(log(n)) if recursion stack is considered
   Reason: Since we are traversing the tree, only constant memory is taken. But, since we are using a recursive function, the recursion stack can be included in space complexity.

2. Print(int rideNumber1, rideNumber2)
   Prints all the rides with ride numbers x, where rideNumber1 <= x <= rideNumber2

   Time complexity – O(log(n)*s), where s is the number of rides in the given range
   Reason: the complexity of searching s nodes in RB tree

   Space complexity – O(s), or O(log(n)*s) if recursion stack is considered
   Reason: we are storing each node in an array. A recursive function is used to search all nodes.

3. Insert(rideNumber, rideCost, tripDuration)
   Adds a new ride request in the system.

   Time complexity – O(log(n))
   Reason: O(log(n)) to add node in min heap, O(log(n)) to add node in RB tree

   Space complexity – O(1), or O(log(n)) if recursion stack is considered
   Reason: space taken by creating a new node is constant. Recursion is used to balance tree and heapify the min heap.

4. GetNextRide()
   Returns next ride to process by prioritizing smaller rideCost and smaller tripDuration in respective order. Also deletes it from the system.

   Time complexity – O(log(n))
   Reason: O(1) to look for the next ride. O(log(n)) to delete root of min heap. O(log(n)) to delete a node from RB tree and balance it.

   Space complexity – O(1), or O(log(n)) if recursion stack is considered
   Reason: O(1) to store the deleted node. Recursive functions are used to balance and the RB tree and heapify the min heap.

5. Cancel(rideNumber)
   Deletes the ride triplet from the system.

   Time complexity – O(log(n))
   Reason: O(log(n)) to search node in the RB tree. O(log(n)) to delete the node, as discussed in the complexity of GetNextRide.

   Space complexity – O(1), or O(log(n)) if recursion stack is considered.
   Reason: Same as that for GetNextRide.

6. Update(rideNumber, tripDuration)
   Updates the given ride according to the rules:
   i. If the new trip duration is smaller than the current trip duration, update the trip duration.
   ii. If the current trip duration < new trip duration <= 2*(current trip duration), update the ride with the new trip duration and increase its cost by 10.
   iii. If the current trip duration > 2*(current trip duration), cancel the ride.

   Time complexity – O(log(n))
   Reason: worst case complexity can be with case 3 where the ride is canceled {O(log(n)) }and inserted back {O(log(n))}.

   Space complexity – O(1), or O(log(n)) if recursion stack is considered.
   Reason: O(1) is used to store a new node. The rest explanation is the same as that for insert and cancel.

Implementation

The implementation requires us to maintain a min heap that stores ride triplets ordered by ride cost and trip duration (if the ride cost is the same for 2 rides). Also, we need to maintain a Red-Black tree ordered by the ride number. We will maintain pointers between nodes in the RB tree and the min heap to achieve the required time complexities.

For the above-discussed requirements, the below implementation is adopted:

Custom data structures

1. GatorRide

```
GatorRide(int rideNumber, int rideCost, int tripDuration) {
    this.rideNumber = rideNumber;
    this.rideCost = rideCost;
    this.tripDuration = tripDuration;
```

```
}
```

2. GatorTreeNode

```
GatorTreeNode(GatorRide ride) {
    this.ride = ride;
    this.indexInHeap = -1;
    // 1 -> RED, 0-> BLACK
    this.color = 1;
    this.left = null;
    this.right = null;
    this.parent = null;
}
```

The *GatorTreeNode* contains *GatorRide* as its data. Also, the same node will be used in our min heap as well. This establishes the pointer from the min heap to the RB tree.

Further, *GatorTreeNode* is also storing *indexInHeap* variable that defines the pointer from the RB tree to the min heap.

3. GatorMinHeap
   Min heap for the Gator taxi service. It is implemented as an array of size 100 (the largest number of rides that can be stored in the service).

```
GatorMinHeap(int maxSize) {
    // initiating min heap with max size
    heap = new GatorTreeNode[maxSize];
    // current size of the min heap
    size = 0;
}
```

4. GatorRedBlackTree
   Red-Black tree for the gator taxi service. It is implemented with pointers left, right, and parent.

```
public GatorRedBlackTree() {
  TNULL = new GatorTreeNode(null);
  TNULL.color = 0;
  TNULL.left = null;
  TNULL.right = null;
```

```
    root = TNULL;
}
```

Here, TNULL is the terminal null node, which is always black in color and contains no value.

5. GatorTaxiMain
   It represents the gator taxi system discussed in the introduction.

```
GatorTaxiMain() {
       rideHeap = new GatorMinHeap(100);
       rideTree = new GatorRedBlackTree();
}
```

Classes & functions

1. gatorTaxi
   This is the main class (entry point) of the program. It reads the given input file, perform corresponding operations by calling appropriate functions, and writes the output to the *output_file.txt* file.

2. GatorTaxiMain
   Represents the gator taxi system. Implements below functions (explained above):
   - String **Print**(int rideNumber)
   - String **Print**(int rideNumber1, int rideNumber2)
   - boolean **Insert**(int rideNumber, int rideCost, int tripDuration)
   - String **GetNextRide**()
   - void **CancelRide**(int rideNumber)
   - void **UpdateTrip**(int rideNumber, int new_tripDuration)

3. GatorMinHeap
   Represents the min heap for the Gator taxi. Implements below functions:
   - void **swap**(int i, int j) -> swap two nodes in the min heap. Also updates the indexInHeap pointers.
   - void **insert**(**GatorTreeNode** rideNode) -> inserts a GatorTreeNode instance in the min heap.
   - **GatorTreeNode remove**() -> removes a node from the top of the min heap, and returns it.
   - void **removeNode**(int rideNumber) -> removes a node located at a certain index.
   - int **peakNextRideNumber**() -> returns the rideNumber for the ride present at the root of the min heap.

HELPER FUNCTIONS

- boolean **isEmpty**() -> return if min heap is empty or not
- void **minHeapify**(int index) -> maintains validity of min heap from the given index

4. GatorRedBlackTree
   Represents the Red-Black tree for the Gator taxi. Implements the below functions:
   - GatorTreeNode **getTNull**() -> returns the TNULL object. This is used by other classes to compare a node with TNULL (to check if it's null).
   - GatorRide[] **getRidesInInterval**(int ride1, int ride2) -> returns a list of nodes that lie between ride1 and ride2 (inclusive).
   - void **insert**(GatorTreeNode rideNode) -> inserts given node to the RB tree.
   - int **deleteNode**(int rideNumber) -> deletes the node with the given ride number. Returns the corresponding index of the deleted node in heap, or -1 if node not found.
   - GatorTreeNode **searchRide**(int rideNumber) -> searches for the node with the given ride number. Returns TNULL if node not found.

   HELPER FUNCTIONS

   - void **visitInorder**(**GatorTreeNode** node, int min, int max, **List**<**GatorRide**> rideList) -> function for inorder traversal of nodes in the interval [min, max].
   - void **fixDelete**(**GatorTreeNode** nodeToFix) -> fix RB tree properties after a node is deleted.
   - void **replace**(**GatorTreeNode** oldNode, **GatorTreeNode** newNode) -> function to replace a node with another.
   - int **deleteNodeHelper**(**GatorTreeNode** root, int rideNumber) -> function to delete a node with a given ride number from tree with a given root
   - void **fixInsert**(**GatorTreeNode** rideNode) -> function to fix properties of RB tree after insertion of a node.
   - **GatorTreeNode inorderSuccessorOf**(**GatorTreeNode** node) -> function to find inorder successor of given node.
   - void **leftRotate**(**GatorTreeNode** rideNode) -> function to left rotate tree with respect to given ride node.
   - void **rightRotate**(**GatorTreeNode** rideNode) -> function to right rotate tree with respect to given ride node.