**COP5536 – Advanced Data Structures**

**ProjectReport**

**Name:** Rukmini Sruthi Sundaraneedi

**Email:** r.sundaraneedi@ufl.edu

**UFID:** 9480-6285

**INTRODUCTION:**

The project aims to develop an application for Gator Taxi, a ride-sharing service, to manage their daily ride requests. The application utilizes a Red-black tree to store ride details ordered by ride number, and a Min heap to store ride details ordered by ride cost and trip duration. The application provides various operations such as printing ride details, inserting new rides, canceling rides, updating ride information, and finding the next ride with the least cost.

**Ride Definition:** A ride is defined by the following triplets:

rideNumber

rideCost

tripDuration

**Operations:**

* Printing Ride Details: This operation prints the details of a particular ride.
* Printing Ride Details Within a Range: This operation prints the details of all rides within a specific range.
* Inserting New Rides: This operation inserts new rides into the Red-black tree and Min heap while ensuring that there are no duplicate ride numbers.
* Finding the Next Ride with the Least Cost: This operation finds the next ride with the least cost by searching through the Min heap.
* Canceling Rides: This operation cancels a ride by removing it from both the Red-black tree and Min heap.
* Updating Ride Information: This operation updates ride information such as ride cost and trip duration in both the Red-black tree and Min heap.

**Data Structures**:

Red-black Tree: The Red-black tree is used to store ride details ordered by ride number while ensuring that there are no duplicates.

Min Heap: The Min heap is used to store ride details ordered by ride cost and trip duration, enabling quick retrieval of the next ride with the least cost.

**Class Definitions with Function Prototypes and Program Structure**

**Ride.java**

The Ride class represents a ride and has the following attributes: rideNumber, rideCost, and tripDuration. It stores the ride details.

**Properties:**

1. rideNumber: An integer that represents the unique identifier for a ride.
2. rideCost: An integer that represents the cost of the ride.
3. tripDuration: An integer that represents the duration of the ride.

**Functions:**

1. Constructorpublic Ride(int rideNumber, int rideCost, int tripDuration):A constructor that initializes the ride attributes (rideNumber, rideCost, and tripDuration) when a new Ride object is created.
2. Getter and Setter methods: Getter methods are used to access the values of the ride attributes, while setter methods are used to modify the values of the ride attributes.The function prototypes of the getter and setter methods are:

* public int getRideNumber()
* public int getRideCost()
* public int getTripDuration()
* public void setRideNumber(int rideNumber)
* public void setRideCost(int rideCost)
* public void setTripDuration(int tripDuration)

**HeapNode.java**

The HeapNode class stores the details of the Ride in the HeapNode and has reference to the corresponding node in RBTreeNode.

**Properties:**

1. ref: A reference to an RBTreeNode object that contains information about the ride.
2. ride: A Ride object that represents the ride information.

**Functions:**

1. Constructor public HeapNode(Ride ride,RBTreeNode ref): A constructor that takes a Ride object and an RBTreeNode object as arguments and initializes the ref and ride attributes.
2. Getter and Setter methods: Getter methods are used to access the values of the ref and ride attributes, while setter methods are used to modify the values of the ref and ride attributes. The function prototypes of the getter and setter methods are:

* public RBTreeNode getRbtReference()
* public Ride getRide()
* private void setRbtReference(RBTreeNode node)
* public void setRide(Ride ride)

**RBTreeNode.java**

RBTreeNode.java is a class representing a node in a Red-Black Tree data structure. Each node in a Red-Black Tree contains a ride object, a boolean flag to indicate whether the node is red or black, and references to its left child, right child, and parent nodes.

**Properties:**

1. ride: a Ride object representing the ride associated with this node.
2. isRed: a boolean value indicating whether the node is red or black in the Red-Black tree.
3. left: a reference to the left child node in the Red-Black tree.
4. right: a reference to the right child node in the Red-Black tree.
5. parent: a reference to the parent node in the Red-Black tree.

**Functions:**

1. Constructor: RBTreeNode(Ride ride, boolean isRed): creates a new node with a given ride object and color.
2. public Ride getRide(): returns the ride associated with this node.
3. public void setRide(Ride ride)): sets the ride associated with this node.
4. public boolean isRed(): returns true if the node is red, false if it is black.
5. public void setRed(boolean isRed): sets the color of the node to red or black.
6. public RBTreeNode getLeft(): returns the left child node.
7. public void setLeft(RBTreeNode left): sets the left child node.
8. public RBTreeNode getRight(): returns the right child node.
9. public void setRight(RBTreeNode right): sets the right child node.
10. public RBTreeNode getParent(): returns the parent node.
11. public void setParent(RBTreeNode parent): sets the parent node.
12. public int getColor(): returns an integer value representing the color of the node (0 for red, 1 for black).
13. public void setColor(int color): sets the color of the node based on an integer value (0 for red, 1 for black).
14. public int getRideNumber(): returns the ride number associated with this node's ride.

**RBTree.java**

This is a Java class implementing a Red-Black Tree data structure, which is a self-balancing binary search tree with the addition of a color bit in each node that indicates whether the node is red or black. The purpose of this color bit is to ensure that no path from the root node to a leaf node is more than twice as long as any other path.

The RBTree class contains methods for rotating nodes, inserting new nodes while maintaining the Red-Black properties, and balancing the tree after insertions. It also defines constants for the Red and Black colors of the nodes.

**Properties:**

rootNode:is a private static variable which stores the root node of the Red Black Tree.

COLOR\_RED: it is a constant which has value 0 when red.

COLOR\_BLACK: it represents a black node when value is 1

public RBTreeNode getRootNode()

public void setRootNode(RBTreeNode rootNode)

public void rotateRight(RBTreeNode node)

public void rotateLeft(RBTreeNode node)

public void insert(RBTreeNode node)

public void balanceRBLInsert(RBTreeNode node)

public List<RBTreeNode> inorderTraverse()

public RBTreeNode searchNode(int rideNumber)

public boolean deleteNode(int rideNumber)

**Functions:**

1. public RBTreeNode getRootNode(): This is a getter function that returns the root node of the tree
2. public void setRootNode(RBTreeNode rootNode): This is a setter function that sets the root node of the tree.
3. public void rotateRight(RBTreeNode node): This method performs a right rotation of the given node around which the rotation is to be performed
4. public void rotateLeft(RBTreeNode node): This method performs a left rotation of the given node around which the rotation is to be performed.
5. public void insert(RBTreeNode node): This method inserts a new node into the Red-Black tree with no regards for the maintenance of the Red-Black property. It contains a call to the balanceRBLInsert() method which is responsible for maintaining the property.
6. public void balanceRBLInsert(RBTreeNode node): This method is responsible for maintaining the Red-Black property of the tree after insertion of a new node. It performs rotations based on the color of the parent, grandparent, and uncle of the affected node.
7. Public void rebalanceRBLDelete(RBTreeNode node) : This method is responsible for maintaining the Red-Black property of the tree after deletion of a node.
8. Public void delete(int rideNumber) : This method deletes a node from the Red-Black tree and calls the balanceRBLDelete() method to maintain the Red-Black property of the tree.
9. Public RBTreeNode search(int rideNumber): This function searches for a particular node with the given ride number in the red-black tree. It takes an integer rideNumber as input and returns a RBTreeNode corresponding to the node with that ride number, or null if it is not found.
10. Public RBTreeNode Print(RBTreeNode rNode, int rideNumber): This function prints the ride details of a particular ride number. It takes a RBTreeNode rNode and an integer rideNumber as inputs, and recursively traverses the tree until it finds the node with the given ride number. It then returns the corresponding RBTreeNode so that the ride details can be printed.
11. public List<RBTreeNode> Print(RBTreeNode rtNode, int rideNumber1, int rideNumber2, List<RBTreeNode> triplets): This function prints the details of ride numbers within a range. It takes a RBTreeNode rtNode, two integers rideNumber1 and rideNumber2 representing the range, and a List<RBTreeNode> triplets to store the RBTreeNodes whose ride numbers are within the range. It recursively traverses the tree and adds the RBTreeNodes whose ride numbers are within the range to the triplets list.
12. public int UpdateTrip(RBTreeNode rNode, int rideNumber,int new\_tripDuration ): This function updates the trip details for a particular ride. It takes a RBTreeNode rNode, an integer rideNumber representing the ride to be updated, and an integer new\_tripDuration representing the new trip duration. It first finds the RBTreeNode with the given ride number in the tree. If the new trip duration is less than or equal to the current trip duration, it returns 1. If the new trip duration is between the current trip duration and twice the current trip duration, it returns 2. If the new trip duration is greater than twice the current trip duration, it returns 3. If the ride number is not found in the tree, it returns 0.

**RideMinHeap.java**

This class implements a min-heap data structure for storing rides based on their ride cost and trip duration.

**Properties:**

1. heap: an array to store the rides
2. capacity: maximum number of rides that can be stored in the heap
3. size: current number of rides in the heap

**Functions:**

1. public RideMinHeap(): constructor function to initialize the heap with a capacity of 2000
2. public int getsize(): returns the current number of rides in the heap
3. public int getParent(int index): returns the index of the parent node
4. public int getLeftChild(int index): returns the index of the left child node
5. public int getRightChild(int index): returns the index of the right child node
6. public void swap(int index1, int index2): swaps the rides at the two specified indexes in the heap array
7. public Boolean isLeaf(int index): checks if the node at the specified index is a leaf node
8. public void heapifyUp(int index): moves the ride at the specified index up the heap until its ride cost is greater than or equal to its parent's ride cost
9. public void heapifyDown(int index): moves the ride at the specified index down the heap until its ride cost is less than or equal to its children's ride costs
10. public void insert(HeapNode ride): function to insert a ride into the heap
11. public Ride peek(): function to return the ride with the lowest ride cost without removing it from the heap
12. public void delete(int index): function to delete a ride of particular index from the heap
13. public HeapNode GetNextRide(): function to get the next ride (with the lowest ride cost) from the heap and remove it from the heap
14. public Boolean isEmpty(): function to check if the heap is empty

**gatortaxi.java**

The gatorTaxi class is the main class which contains the main() method that reads input from a file and processes the commands in the file. In this class we initilazie a Scanner to read input and, PrintWriter to write output. The class has a RBTree object to hold all the rides ordered by ridenumber and a RideMinHeap object to hold rides ordered by ride cost and trip duration.

Then, it reads input line by line and splits the line into an array of words using whitespace as a delimiter. The first word in the line indicates the type of command that needs to be executed. Depending on the command, the code performs different operations.

Each command is of the form Command arg1 arg2 arg3, where Command is one of the following commands:

* For the Insert command, it checks if the ride number already exists in the RBTree. If it does, it returns an error message. If not, it creates a new Ride object with the given ride number, ride cost and trip duration and inserts it into the RBTree. It also creates a new HeapNode object with the Ride object and the RBTreeNode object and inserts it into the RideMinHeap.
* For the Print command with one argument, the program searches the RBTree for the ride with the given ride number and prints its ride number, ride cost and trip duration. If it is not found, it prints 0,0,0.
* For the Print command with two arguments, the program prints all rides whose ride numbers are within the range num1 and num2, inclusive. If there are no such rides, it prints 0,0,0.
* For the GetNextRide command, the program removes the ride with the minimum trip duration from the RideMinHeap. It then deletes the RBTreeNode object corresponding to the ride from the RBTree. It prints the ride number, ride cost and trip duration of the ride. If there are no rides in the RideMinHeap, it prints nothing.
* For the command is "CancelRide", the ride number is extracted from the input, and the corresponding ride is searched for in the RBT. If the ride is found, it is deleted from both the RBT and the min-heap.
* For the command is "UpdateTrip", the ride number and the new trip duration are extracted from the input. The code then searches for the ride with the given ride number in the RBT and updates its trip duration. If the new trip duration is less than or equal to the existing trip duration, the code deletes the old ride from both data structures and inserts a new ride with the updated duration. If the new trip duration is greater than the existing duration but less than or equal to two times the existing duration, the ride cost is updated, and a new ride with the updated cost is inserted. If the new trip duration is more than two times the existing duration, the ride is deleted from both data structures.

Finally, we close the output stream and the scanner.

**Program Structure:**

The structure of the program for the above code is as follows:

1. Import necessary packages
2. Define the class Ride
3. Define the class HeapNode
4. Define the class MinHeap
5. Define the class RBTreeNode
6. Define the class RedBlackTree
7. Define the class gatorTaxi
8. Define the main() method in the gatorTaxi class
9. Create objects of the necessary classes
10. Read input from a file
11. Process the input commands and update the data structures accordingly
12. Write the output to a file
13. Close the output file and end the program

The main logic of the program is in the gatorTaxi class, where input commands are processed and data structures are updated accordingly. The MinHeap and RedBlackTree classes are used to implement the data structures for the ride booking system in gatorTaxi class. The Ride class and HeapNode class are helper classes for these data structures. The RBTreeNode class is used to store nodes in the Red-Black Tree.

**Program Flow:**

The complete program flow for Gator Taxi can be described as follows:

1. Initialize the Ride class with the following attributes:
   * + Ride number
     + Ride cost
     + Ride duration
2. Initialize the RBTreeNode class with the following attributes:

* Ride object
* Color (RED or BLACK)
* Parent node
* Left child node
* Right child node

1. Initialize the RBTree class with the following methods:

* Search: searches for a node in the tree based on the ride number
* Insert: inserts a node into the tree based on the ride number
* Delete: deletes a node from the tree based on the ride number
* UpdateTrip: updates the ride duration for a given ride number

1. Initialize the HeapNode class with the following attributes:

* Ride object
* Reference to the corresponding RBTreeNode object

1. Initialize the MinHeap class with the following methods:

* Insert: inserts a new HeapNode object into the heap
* ExtractMin: removes and returns the HeapNode object with the minimum ride cost
* GetSize: returns the current size of the heap

1. Initialize the GatorTaxi class with the following methods:

* Main: reads commands from a file, performs the corresponding operations on the RBTree and MinHeap, and writes output to a file
* ProcessCommand: parses a command string and performs the corresponding operation on the RBTree and MinHeap
* GetRideInfo: retrieves the ride information from a string
* Print: writes the result of a command to a file

1. The main method of the GatorTaxi class performs the following operations:

* Creates an instance of the RBTree class
* Creates an instance of the MinHeap class
* Reads commands from an input file
* Parses each command
* Performs the corresponding operation on the RBTree and MinHeap
* Writes the output to an output file

1. To Process a Command we do the following:

* Parses a command string into its component parts
* Calls the corresponding method in the RBTree or MinHeap class to perform the operation

Overall, the program flow involves reading commands from a file, processing each command by performing the corresponding operations on the RBTree and MinHeap, and writing the result to an output file. The RBTree and MinHeap classes provide efficient data structures for searching, inserting, and deleting ride information, while the GatorTaxi class provides a framework for executing the program logic and managing input and output.

**Time and Space Complexity:**

The time complexity of the above program depends on the operations being performed. Here are the time complexities of some of the main operations:

1. Inserting a ride into the RBTree: O(log n) where n is the number of nodes in the tree.
2. Inserting a ride into the min-heap: O(log n) where n is the number of nodes in the heap.
3. Deleting a ride from the RBTree: O(log n) where n is the number of nodes in the tree.
4. Deleting a ride from the min-heap: O(log n) where n is the number of nodes in the heap.
5. Searching for a ride in the RBTree: O(log n) where n is the number of nodes in the tree.
6. Updating a ride in the RBTree: O(log n) where n is the number of nodes in the tree.
7. Updating a ride from the min-heap: O(log n) where n is the number of nodes in the heap.

The time complexity of the overall program would depend on the number of operations being performed and the size of the input file. Assuming there are m operations and n rides in the input file, the worst-case time complexity of the program would be O(m log n) since each operation may involve a log n operation on the RBTree or the min-heap.

As for space complexity, the program creates an RBTree and a min-heap to store the rides. The space complexity of the RBTree would be O(n) where n is the number of nodes in the tree, while the space complexity of the min-heap would be O(n) where n is the number of nodes in the heap. The program also creates several temporary variables and arrays to hold the input data and perform the operations, but their space requirements are negligible compared to the data structures used to store the rides. Therefore, the overall space complexity of the program would be O(n).

**Conclusion:**

The Gator Taxi's application uses a Red-black tree and a Min heap to manage their daily ride requests efficiently. With features such as printing ride details, inserting new rides, canceling rides, updating ride information, and finding the next ride with the least cost, Gator Taxi can streamline their daily operations and improve customer satisfaction. The complexity in the worst case will be O(m log n) if there are n active rides and m operations are made. Hence, I got a clear understanding of how to implement Red Black Tree and Min heap.