COP 5536 – Advanced Data Structures

SPRING 2023

Programming Project

GATOR TAXI

PROJECT REPORT

Name: Adithya Koti Narasimhachar Guruachar
UFID: 9342-2539
Email: adithyakotinaras@ufl.edu

Problem statement:

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:

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.

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 <= r_x <= 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, update the ride with the new trip duration.

    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.

Programming Environment:

GatorTaxi is implemented in java programming language. All java classes are in the same directory. Input file should be in the same directory as the java classes. A makefile is present whose default function is to create .class files of all java classes. Output will be written to output_file.txt which is present in the same directory as the input file.
The program can be executed in the following order.

1. make
2. java gatorTaxi input_file_name.txt

Implementation:

To implement gatorTaxi we make use of a min heap and red-black tree (RBT). The min heap priorities rides with lower ride cost. If two or more rides have the same ride cost, then the ride with minimum trip duration is given more priority. The RBT is implemented with ride number as the search key.

1. To implement **Insert (rideNumber, rideCost, tripDuration)** method.

    To implement Insert method, we first check if the given ride number is already present in the RBT, if yes then we quit. If the ride with the given ride number is not present in the RBT, then we create a new min heap node and a new RBT node. We establish a link between these two new corresponding nodes, by adding a min heap pointer into RBT node and a RBT node pointer into min heap node. This is required to achieve cancelRide method in O(log(N)) complexity.

2. To implement **Print (rideNumber)** method.

    To implement Print method, we traverse through the RBT tree by comparing at each node, its ride number with the given ride number and either traverse through the left subtree or the right subtree until we find the node with required ride number. If we don't find the node with the required ride number we print (0,0,0) or else print the ride details.

3. To implement **Print(rideNumber1, rideNumber2)**

    To implement Print method between a given two ride numbers , we perform an recursive in-order traversal of the RBT tree to find and print the details of the rides whose ride numbers lies between the rideNumber1 and rideNumber2 (both included).

4. To implement **GetNextRide()** method.

    To implement GetNextRide method we perform removeMin() functionality of the min heap to get the node with the least ride cost. If two nodes have the same least ride

cost, then the node with lowest trip duration is returned by the removeMin() function. Once the ride is removed from the min heap, using the ride number of the ride, the ride is also deleted from the RBT tree.

5. To implement **CancelRide(rideNumber)** method.

   To implement CancelRide method, we perform a delete operation in the RBT tree and delete the node with the given ride number. Next, using the min heap pointer of the deleted RBT node, we get the corresponding min heap node to be deleted. We delete this min heap node and either heapify up or heapify down. For performing a remove operation at arbitrary location in min heap we store the index in the min heap node and after removing we heapify in the upward or downward direction.

6. To implement **UpdateTrip(rideNumber, new_tripDuration)**

   To implement updateTrip method, we have three conditions.
   If the new_tripDuration <= existing tripDuration, then we update the trip duration of the ride with the given ride number in both min heap and RBT tree. If required, we heapify the min heap either in the upward or downward direction.
   Second, if the existing_tripDuration < new_tripDuration <= 2*(existing tripDuration), then we perform cancelRide(rideNumber) and perform Insert(rideNumber, rideCost+10, new_tripDuration).
   If the new_tripDuration > 2*(existing tripDuration), then we simply perform cancelRide(rideNumber).

Operation on rideNumber such as Print(rideNumber), Print (rideNumber1, printNumber2) are done on the RBT tree while the other operations are done by modifying both the RBT tree and min heap.

The input file name is read as a command line argument and we loop across each statement in the input file, perform the required operation and write the result to the output file named as output_file.txt.

Project Structure:

The project consists of five classes and a makefile.
1. makefile
2. minHeapNode.java
3. RBTNode.java
4. MinHeap.java
5. RBT.java
6. gatorTaxi.java

Makefile:

The make file is used to create .class files of the above mentioned five classes. It has a default argument which complies all the classes.

```
default:
   javac minHeapNode.java
   javac RBTNode.java
   javac MinHeap.java
   javac RBT.java
   javac gatorTaxi.java
```

Class Definitions and function prototypes:

1. minHeapNode.java

This class is used to create objects for rides of gatorTaxi which is pushed into min heap. It contains ride number, ride cost, trip duration, index, and pointer to corresponding RBT Node.

Instance variables:

- public int rideNumber – Every ride has a unique ride number and is stored in this variable.
- public int rideCost - Ride cost of a ride is stored in this variable.
- public int tripDuration - The trip duration of a ride is stored in this variable.
- public int index - The index of the current node in the minheap array is stored in this variable.
- public RBTNode rbtPointer - This is a pointer to the corresponding node in RBT data structure.

Constructor:

- minHeapNode(int rideNumber, int rideCost, int tripDuration) – This constructor is used to initialize the instance variables.

Instance methods:

- public String toString() - This methods overrides default toString method of Object class and returns the all the instance variables with primitive data types as a string concatenated by a space. The time and space complexity if O(1).

2. RBTNode.java

This class is used to create objects for rides of gatorTaxi like minHeapNode and is used for pushing into RBT. It contains ride number, ride cost, trip duration and pointer to corresponding minheap node.

Class variables:

- final static int black = 1 – A final variable used for referring to black node color.
- final static int red = 0 – A final variable used for referring to red node color.

Instance variables:

- public int rideNumber - Every ride has a unique ride number and is stored in this variable.
- public int rideCost - Ride cost of a ride is stored in this variable.
- public int tripDuration - The trip duration of a ride is stored in this variable.
- public minHeapNode heapNode - This is a pointer to the corresponding node in min heap data structure.
- public int colour – This is used to store the colour of the RBT node.
- public RBTNode left – This is used to store the reference to the left child.
- public RBTNode right – This is used to store the reference to the right child.
- public RBTNode parent – This is used to store the reference to the right child.

Constructor:

- RBTNode() – A default constructor used for creating of external node. It initializes the left, right and parent variables to null and colour variable to black.
- RBTNode(int rideNumber, int rideCost, int tripDuration, int colour, RBTNode externalNode) – This constructor is used to create RBT Nodes and initializes the instance variables. left, parent and right variables are initialized to externalNode.

Instance methods:

- public String toString() - This methods overrides default toString method of Object class and returns the all the instance variables with primitive data types as a string concatenated by a space. The time and space complexity is O(1).

3. MinHeap.java

This class contains the implementation of min heap. The min heap is ordered by minimum ride cost and if there are multiple rides with the same ride cost, then the min heap is ordered by trip duration.

Instance variables:

- public minHeapNode minHeap[] – Array for storing minHeapNodes.
- public int size – Integer which stores number of node present in the minHeap.

Constructor:

- MinHeap() – This constructor initializes the minHeap array and size variable to 0.

Instance methods:

- public minHeapNode **getParentNode(int index)** – Method that takes the index of the minHeapNode as the input and returns its parent node. The time and space complexity of this method is O(1).
- public minHeapNode **getLeftChildNode(int index)** – Method that takes the index of the minHeapNode and returns its left child. The time and space complexity of this method is O(1).
- public minHeapNode **getRightChildNode(int index)** – Method that takes the index of the minHeapNode and returns its right child. The time and space complexity of this method is O(1).
- public int **getParentIndex(int index)** – Method that takes the index of the minHeapNode and returns the index of the parent. The time and space complexity of this method is O(1).
- public int **getLeftChildIndex(int index)** – Method that takes the index of the minHeapNode and returns the index of the left child. The time and space complexity of this method is O(1).
- public int **getRightChildIndex(int index)** – Method that takes the index of the minHeapNode and returns the index of the right child. The time and space complexity of this method is O(1).
- public boolean **isNodeALeaf(int index)** – Method that takes the index of the minHeapNode and returns if the node at that index is a leaf node or not a leaf node. The time and space complexity of this method is O(1).
- public boolean **hasLeftChild(int index)** – Method that takes the index of the minHeapNode and checks if the node at this index has a left child. The time and space complexity of this method is O(1).
- public boolean **hasRightChild(int index)** – Method that takes the index of the minHeapNode and checks if the node at this index has a right child. The time and space complexity of this method is O(1).
- public void **swapNodes(int index1, int index2)** – Utility function that is used to swap two nodes present at indexes index1 and index2. The time and space complexity of this method is O(1).
- public **void insert(minHeapNode newNode**) – Method to insert a new minHeapNode into the min heap. This method calls heapifyUp. The time complexity of insert is O(log(N)) as heapify takes O(log(N)) time. The space complexity is O(1).
- public void **heapifyUp(int currentIndex)** – Method to heapify the min heap in the upward direction form the node at currentIndex. The time complexity is heapifyUp is order of height of the tree and is O(log(N)). The space complexity is O(1).
- public void **heapifyDown(int index)** – Method to heapify the min heap in the downward direction from the node at index. The time complexity is heapifyDown is order of height of the tree and is O(log(N)). The space complexity is O(1).
- public minHeapNode **removeMin()** – Method used to remove and return the root of the min heap. The time complexity of this method is O(log(N)) as it internally calls

heapifyDown and heapifyDown takes O(log(N)) time. The space complexity of this method is O(1).

- public void **removeNode(minHeapNode node)** – Method used to remove the node passed as a parameter from the min heap. The time complexity of this method is O(log(N)) as it internally calls heapifyDown or heapifyUp and it takes O(log(N)) time. The space complexity of this method is O(1).

4. RBT.java

This class contains the implementation of Red Black Tree. The search key in RBT is the ride number.

Class variables:

- static RBTNode externalNode – External node of the RBT

Instance variables:

- RBTNode root – Root node of the RBT

Constructor:

- RBT() – A default constructor that initializes externalNode and assigns the externalNode to the root

Instance methods:

- public void **insert(RBTNode newNode**) – Method to insert a new RBTNode inside the red black tree. The insert is similar to insertion in binary search tree, and the time complexity is order of height of the tree which is O(log(N)).
- public void **restoreRbtOnInsert(RBTNode node)** – Method to restore the RBT properties after insertion of a new node. This operation involves colour changes and rotations, each takes constant amount of time. The overall time complexity is again order of height of the tree which is O(log(N)). The space complexity is O(1) as no extra space is used.
- public void **rotateRight(RBTNode node)** – Method to rotate the given node to the right. The time complexity of this operation is O(1) as it involves only constant number of pointer changes and colour changes. The space complexity is O(1).
- public void **rotateLeft(RBTNode node)** – Method to rotate the given node to the left. The time complexity of this operation is O(1) as it involves only constant number of pointer changes and colour changes. The space complexity is O(1).
- public RBTNode **getUncleNode(RBTNode parent)** – Method to get the uncle node of the given parent node. The time complexity of this operation is O(1) as it involves just involves pointer access. The space complexity is O(1).
- public RBTNode **inOrderSuccessor(RBTNode node)** – Method to get the in-order successor of a given node. The time complexity of this function is order or height of the tree as we need to traverse the entire height of the tree in the worst case to find the in-order successor. The time complexity is therefore O(log(N)) and has a constant space complexity.

- public void **deleteRBTNode(int rideNumber)** – Method to delete a RBTNode with the given ride number. The time complexity of this function is order of height of the tree (same as in BST) and is O(log(N)). The operation doesn't use any extra space so the space complexity of O(1).
- public void **restoreRbtOnDelete(RBTNode fNode)** – Method to restore the RBT properties after a node is deleted. The node which violates the properties of the RBT is passed as a parameter. The time complexity of this function is O(log(N)) as it involves constant amount of rotations and colour changes. No extra space is used so the space complexity is O(1).
- public boolean **checkRideNumber(int rideNumber)** – Method to check whether there exists any node in RBT with the given ride number. This function takes O(log(N)) time as it involves traversing the entire tree height in the worst case.
  The space complexity is O(1).
- public **RBTNode getNodeFromRideNumber(int rideNumber**) – Method to get the node from RBT with the given ride number. It returns null if the node with the given rideNumber is not present. This function takes O(log(N)) time as it involves traversing the entire tree height in the worst case.
  The space complexity is O(1).

5. gatorTaxi.java

This is the class with the main method and it contains the code for taking input from the file, performing insert, delete and update operations on both the min heap and RBT and outputting the results to the file.

Class variables:

- static int **printCount** – Print count stores the already printed rides in print operation. It is used to separate multiple rides by comma in Print(rideNumber1, rideNumber2) method.

Class methods:

- public static boolean **insert(int rideNumber, int rideCost, int tripDuration, MinHeap mHeap, RBT rbt)** –

  This method implements **the Insert (rideNumber, rideCost, tripDuration)** requirement.
  This method is used for insertion into min heap and RBT. First, it checks whether a ride is already present in the RBT and if it's not present then it inserts a new Node with given ride details into both min Heap and RBT. The time complexity is O(log(N)) for checking if the ride already exists and O(log(N)) for insertion into min heap and O(log(N)) for insertion into RBT. So, the total time complexity is **O(log(N))**. On successful insertion it returns true, or else false.

- public static String **printRideNumber(int rideNumber, RBT rbt)** –

  This method implements **Print(rideNumber)** requirement.

This method is used for returning the node with the given ride number. The RBTNode from the given ride number is obtained in the complexity of order of height of the RBT which is **O(log(N))**. If the node with the given ride number doesn't exist it returns (0,0,0).

- public static int **printRideNumberWithInRange(int min, int max,RBTNode rbt, FileWriter outputWriter)** –

  This method implements **Print(rideNumber1, rideNumber2)** requirement.
  This method is used for printing nodes whose ride number ranges in between min and max both included. We perform an in-order traversal recursively in the RBT and print the nodes that are in the required range.
  The time complexity of this operation is **O(log(N) + S)**. O(log(N)) to find the node with ride number greater than or equal to the minimum ride number and S for getting next S successors.
  This function returns the number of nodes within the range along with printing the node details.

- public static String **getNextRide(MinHeap mHeap, RBT rbt)** –

  This method implements **GetNextRide()** requirement.
  This method checks if there are any active rides present from the min heap and if it's present then performs removeMin from min heap and deleteRBTNode from RBT tree and returns the node details. If there are no active rides, then it returns "No active ride requests".
  The time complexity of removing minimum element from min heap is O(log(N)) and removing the corresponding node from the RBT also take O(log(N)) time. So, the overall complexity if **O(log(N)).**

- public static void **cancelRide(int rideNumber, MinHeap mHeap, RBT rbt)**

  This method implements **CancelRide(rideNumber)** requirement.
  This method checks if there is a node with given ride number in the RBT. If yes, then removes this node from the RBT. Using the heapNode pointer present in the RBT node, we remove the corresponding node from the min heap and perform heapify.
  The complexity of checking and removing a node from RBT is O(log(N)) and removing a node from min heap is O(log(N)). We store the index of the min heap node, to achieve O(log(N)) complexity for removal of an arbitrary node. So, the overall complexity of cancelRide is **O(log(N)).**

- public static void **updateTripDuration(int rideNumber, int modifiedTripDuration, RBT rbt, MinHeap mHeap)**

  This method implements **UpdateTrip(rideNumber,newTripDuration)** requirement.
  This method is used for updating the trip duration of a ride with the given ride number. First we check and get the RBT node from the RBT tree with the given ride number. This takes O(log(N)) complexity.

Next, if the current trip duration is greater than or equal to the new modified trip duration, then we simply update the trip duration of the node in both RBT and minheap and heapify min heap if required. This takes O(log(N)) complexity to heapify the min heap.
If the existing trip duration is less than the new modified trip Duration and the modified trip Duration is less than or equal to 2 times the existing trip duration then cancel the ride and create a new ride with cost penalty of 10. This takes the time complexity of O(log(N)). In all other conditions we simply cancel the ride. This takes the time complexity of O(log(N)). So the overall cost of this operation is **O(log(N)).**

All operations mentioned in the problem statement are implemented in O(log(N)) complexity.