COP5536 ADVANCED DATA STRUCTURES

Project Report

Yaswanth Battineedi UFID 10590305
y.battineedi@ufl.edu

Introduction

GatorTaxi ride-sharing service takes ride requests from customers and then keeps track of them. The service works on 3 important things about the ride. The cost of the ride, the trip duration and the position of the ride in the list of all rides requested and currently in queue. The service has some criteria by which it chooses the next ride based on the cost of the ride. In case of ties, then the trip duration is used as a tie-breaker. The service also provides the option to change an already requested ride. Customers are able to change an already requested ride to a new destination or cancel it. When changing the destination, a penalty will be applied in the form of extra cost if the new trip duration is more than the previous duration and the ride is then updated. If the change in destination causes the ride duration to become more than twice as long as the previous one, the ride is canceled altogether. If the new trip duration is less than the previous trip duration of the ride, no penalty is applied and the ride is updated.

Implementation

The rides are stored in the form of triplet consisting of the rideNumber, rideCost and the tripDuration. These rides are stored in 2 data structures simultaneously, a Min Heap and a Red Black Tree. But they are stored according to different criteria in each of them. Min Heap stores the rides sorted by the ride cost while the ride with lowest cost being at the root of the heap. And the Red Black Tree stores the rides sorted on the basis of the ride number. We only service one ride at a time across the service, so the two data structures serve different purposes for the service.

The **Data Structures** and their functionality are briefly described below:

Ride: This class consists of the information related to the ride, i.e ride number, ride cost and the trip duration. It also has a function to compare the ride costs of 2 rides.

Heap Node: This is the node that will be inserted into the heap once it is created. It stores the ride(total information of ride) and a reference corresponding to the RedBlackTree node which is storing the same ride.

Min Heap: The heap take the heap nodes and stores them sorted by the ride cost so that the lowest cost ride is always at the top. It also keeps track of its size.

> **Insert:** This function inserts the node passed to it into the heap at the bottom and increases the size of the heap by 1. After the insertion it calls the fix_heap_bottom_up to restore the heap invariant.

Delete: This removes the element at the given position and replaces it with the last placed element into its position. And then the fix bottom up function is called to fix the heap property

Pop: This returns the top element of the heap and replaces it with the last placed element in the heap. Then the fix top down function is called to fix the heap property.

Update: This the key of the node at the given index to a new given key and then calls the fix function (bottom up or top down) depending on whether the nodes new key is less or greater than its parents key.

Swap: Helper function to swap nodes during the operation of fix functions. Swaps 2 nodes.

Fix Bottom Up: Function that starts at the given index and swaps the node with its parent if the key is less than the parent. Repeats until the condition is invalid (parent is less than child).

Fix Top Down: Counterpart to the bottom up function that starts at the given index and swaps the node with its child if its key is greater than the child's. The child with the lowest value key is considered.

Red Black Tree Node: Data structure that is placed into the red black tree. Contains the ride object and the color of the node as well as the reference corresponding to the Min Heap node.

Red Black Tree: Data structure for the red black tree. Stores the nodes of the RBT and has various functionality to operate and also preserve the tree property.

Insert: Inserts the given ride node into the tree and colors it. It traverses through the tree for an appropriate place to insert the node and then sets the parent and children. After, the colors are fixed and then the insert balance function is called to balance the tree out.

Delete: Traverses the node looking for the node to delete and handles two types of cases based on how many children the node about to be deleted has, either 1/0 or 2. Then the delete balance function is called to balance the tree out.

Get Ride: Finds the ride with the given key and returns it.

Range Search: Finds the rides between the given 2 keys and returns them.

Get minimum: traverses the leftmost branch of the tree to find the minimum node and returns it. Helper function used by other functions.

Replace Node: Replaces a node with its child in the tree. Helper function used by other functions.

Insert Balance: This function is used to balance the tree after every insertion. It performs necessary rotations and recolors the nodes to restore the property of the Red Black Tree.

Delete Balance: This function balances the tree after every deletion. It rotates and recolors where necessary to restore the property of the Red Black Tree.

Left Rotate and **Right Rotate:** Functions to rotate the branches of the tree. These are helper functions used by balancing functions.

The above are the base level data structures implemented. The higher level functions and the main function is briefly discussed below:

Insert Ride: Creates a ride with the given information and then in turn creates the min heap and red black tree objects to store the ride in. Then passes them both to be inserted into their respective data structures.

Get Next Ride: Pops the top ride in the heap, which is the ride with the lowest cost, and then deletes the corresponding node in the red black tree. In case there are no rides to pop, prints “No Active Ride Requests”.

Cancel Ride: Deletes the ride from both the red black tree as well as the min heap.

Update Ride: If the ride is to be updated with a new trip duration, it checks if the new duration is less than the old duration or if its more then old duration but less than twice or if its more than twice the old duration. Depending on the outcome of that check, it either updates the trip duration alone or updates the trip duration with a +10 penalty to the cost of the ride or outright cancels the ride respectively.

Print Ride(R): Traverses the tree using the given ride number and then returns the ride.

Print Ride(R1, R2): Traverses the tree and finds there rides which fall in between the two given rides.

Output Helper: Function which takes the messages from different functions and prints them all to an output_file.txt.

Main: Main function of the code. It takes the input from the console, opens the input file, parses the commands in it into words and numbers and then depending on parsed output says, calls the relevant functions to process the commands. Also creates the output_file.txt for the output helper function to write into.

Time and Space Complexity:

For the Print(rideNumber) operation, since the ride numbers are unique and the red-black tree is a self-balancing binary search tree, we can search for the specific ride number in O(log(n)) time complexity.

For the Print(rideNumber1, rideNumber2) operation, we optimize the search process by only entering the subtrees that may possibly have a ride in the specified range by checking if the ride number is less than or greater than the given range (R1,R2). This would reduce the search space and improve the efficiency. This will only traverse relevant subtrees to find the required rides.. The function starts at the root node and recursively traverses either the left or right subtree based on the comparison of the current node's ride number with the given range. In this way, we can minimize the number of nodes visited during the search, leading to a time complexity of O(log(n)+S) where S is the number of rides printed.

All other operations like insert, delete, and update take O(log(n)) time complexity as these are standard operations on a min heap and a red-black tree.

The space complexity of the implementation depends on the number of active rides in the system. The min heap data structure is used to store the active rides in the order of their cost, which takes up space proportional to the number of rides. The space complexity of a binary heap is O(n), where n is the number of elements in the heap.

The red-black tree data structure is used to store the rides based on their ride numbers, which takes up space proportional to the number of rides. The space complexity of a red-black tree is O(n).

The implementation also uses some additional variables and data structures to store information and process queries, but their space requirements are small and do not depend on the number of rides. Therefore, the overall space complexity of the implementation is O(n), where n is the number of active rides in the system.