Advanced Data Structures Project Report

Srija Chaturvedula
UFID: 52003934, schaturvedula@ufl.edu

INTRODUCTION

To implement the given project, gatorTaxi ride-sharing service we will be using a MinHeap and a RedBlack Tree. The MinHeap is used to store rideNumber, rideCost and tripDuration which is ordered by rideCost having the lowest ride cost as the root. If there are multiple entries with the same rideCost, the shortest tripDuration one will have the highest priority. The RedBlack Tree is used to store rideNumber, rideCost and tripDuration which is ordered by rideNumber.

Maintaining pointers between MinHeap and RedBlack Tree is necessary as these pointers will help us in locating and updating the appropriate triplets efficiently. We can accomplish this by creating a separate data structure that stores the pointers to the respective nodes in the MinHeap and RedBlack Tree. Whenever we update, insert or delete either one of the triplet in the MinHeap or RedBlack Tree, we can update the pointers in this separate data structure created.

IMPLEMENTATION

MinHeap:

‘heap_list’ and ‘curr_size’ are initialized as 0.

Heap List is used to store the elements as a complete binary tree and the first element is set to 0.

Curr size is used to store the number of elements in the heap, it can be increased or decreased as the heap gets updated.

‘insert’ method is used to insert a new element into the heap. First it appends the new element to the bottom of the heap and then increases the size of the heap by 1. Then, it calls for the ‘bubble_up’ method to move the new element in the heap until it's in the correct position.

‘**swap’** method is used to swap two elements with different indexes in the minHeap. It also updates their respective index attributes after swapping. It is necessary to have the correct index as the index attribute of MinHeapNode is used to locate the node in the heap list during various ‘bubble_up’ and ‘bubble_down’ operations.

‘get_minchild’ method is used to get the index of the minimum child of a given node at a particular index. First it checks for the left node if it is out of range, if it is then it returns the index of the right child. The method compares ride objects of the left and right child class using the ‘less_than’ method from the Ride class, it returns the index of the minimum child.

‘update_element’ method updates ‘triptime’ with a new value and then decides whether to bubble up or down to restore the heap property. If the updated value is the root of the node or if the node’s parent has a smaller ride than the updated node it calls ‘bubble_down’ to move the node down the heap, otherwise it calls ‘bubble_up’ to move it up. It also takes into consideration the ‘less_than’ method which gives true if this node’s trip time is less than the trip time of another ride object.

‘delete_element’ method deletes the element from a particular index in the heap. It swaps the last element in the heap to that particular index from where the element has been deleted and then removes the last element to ensure the heap is complete. After it’s swapped, the size of the heap is decreased by 1 and then calls ‘bubble_down’ to move it into the correct position.

‘bubble_up’ method moves the element up in the heap until it's in the correct position from a particular index. It checks whether the parent of the current node is valid, then compares the ride time of the current node and the parent node. If the ride time of the current node is less than that of its parent then it swaps using the ‘swap’ method. Updates the index to that of its parent, where if needed the next iteration starts.

'bubble_down' method moves the node down the heap until it's in the correct position from a particular index. This is done repeatedly by comparing the current node and its minimum child node and also swapping them using 'swap' if necessary, then moving to the minimum child and continuing the process.

'pop' method pops the ride which has the minimum trip duration from the heap which is the root node. Then, it swaps the root node with the last element in the heap and decreases the size of the heap by 1. It deletes the last element of the heap and calls 'bubble_down' to move the root to the correct position and then returns the root node which has the minimum trip duration.

MinHeap Node Class

This class represents a node in the min heap. It contains three attributes:

ride: The Ride object associated with this node, represents the ride associated.
rbt_node: The RBTNode object associated with this node, represents the corresponding node in the RedBlack Tree
min_heap_index: Used to keep track of the index of this node in the min heap.

It is used to maintain correspondence between RedBlack Tree and min heap. Everytime a ride is inserted into the data structure, an object is created for it which contains pointers to the corresponding node in the RedBlack tree and min heap. The pointers are used to maintain correspondence if data structures are modified.

RedBlack Tree Node Class

It defines a RBTNode object, which is a node in the RedBlack Tree.

The node has six attributes:

self.ride: stores the ride object associated with the node
self.pp: a pointer to the parent node
self.left: a pointer to the left child node
self.right: a pointer to the right child node
self.color: the color of the node
self.min_heap_node: stores the MinHeap Node object associated with the ride

Initially self.pp,self.left,self.right are initialized to none and self.color is initialized to 1 as it represents red.

RedBlack Tree

‘get_ride’ method is used to search for a ride with the specific ride number in the RBT, it checks from the root of the tree and iterates until it finds the ride and returns it.

'findrange' method is used to find all the rides with a given range of ride numbers given. If the current node is greater than 'low' it traverses the left subtree, if its less than 'high' then it traverses the right subtree. If it's in the range of [low,high] appends it to the list.

‘getrange’ method takes in low and high values and initializes a list to store all the elements to store all the rides that fall in the range. It returns the list.

'repnode' method is used for replacing a node in the RBT with a new node. It traverses the tree from the root and accordingly to find the node to be replaced and then replaces it with the new node.

'deletenodehelp' method is used to delete a node from RBT. It checks for the node based on the given key and determines whether to delete the node or to delete the successor if the node has children while also maintaining the RBT properties.

'post_delete_fix' method is called when a node is deleted in order to ensure that all the RBT properties are maintained. It recolors and rotates the tree accordingly.

'insert' method is used to insert a new node into the RBT, it creates a new node and sets its parent, childs and color and then traverses the tree to find the correct position to be inserted.

'post_insert_fix' method is called after an insert to ensure that all the RBT properties are maintained.

'minimum' method is used to get the node with the minimum value at the given node subtree. It traverses through and returns the leftmost node which has the minimum value.

'leftro' method is used to implement left rotation at a given node. To get the right child of the given node to be the root, restructure of the binary tree is required. Hence, left rotation.

'rightro' method is used to perform right rotation on the tree in order to maintain the balance of the tree when a particular node is inserted or deleted.

RIDE

'less_than' method is used to compare two ride objects based on their cost and trip duration.

'insert_ride' method is used to insert a ride into the min heap and RBT. It checks if the ride is already present based on the number if yes, returns an error and if no, it creates a node for both of them respectively and inserts the minheap node into the minheap tree and same for RBT.

'print_ride' method retrieves the ride from the given ride number from a RBT and prints it, otherwise it prints that the ride is not found.

'print_rides' method retrieves all the rides from the given range of ride numbers from a RBT and prints it in the ascending order.

'get_next_ride' method pops the top element from the min heap node and deletes the RBT node which corresponds to it and prints it. Otherwise, prints that there are no active rides.

'cancel_ride' method deletes the particular ride from the RBT and min heap.

'update_ride' method is used to update the duration of a ride and adjusts the cost of the ride with the given ride number and updated trip duration respectively.

'write_to_output' method gives the message string to the output_file.txt. It converts the list of ride objects into a comma separated string and writes it to the output file.

Main Function

It reads the input from the input file and writes output to the output_file.txt. MinHeap and RedBlackTree are created to store all the rides. It reads the command from the input file and processes them into the command word and the ride details. And based on the command word, the main function calls the appropriate function to execute the input. Once all the commands are processed, the output file is closed and the program exits.

Time and Space Complexity

The search for a single ride using print_ride method should be executed in O(log(n)) time complexity, because the search is performed on the implemented Red-Black Tree data structure that ensures O(log(n)) search time complexity because of the binary search tree property.

Similarly, the print_rides method should be executed in O(log(n) + S) time complexity, which is achieved by searching only those subtrees that may possibly have a ride in the specified range.

For the remaining methods, the time complexity is O(log(n)) for insert, delete, and update operations, because they are operating on a balanced binary tree. Some other operations might take some time but it will be negligible compared to n because they are more or less constant and don't depend on n.

The total number of currently present rides is the main variable that affects space complexity. The min heap and RBT are used to store all the rides, so the space complexity of these data structures is proportional to the number of rides.

For the min heap, the space complexity is O(n) where n is the number of active rides. This is because the heap is a complete binary tree and the number of nodes in a complete binary tree of height h is 2^(h+1) - 1. In this case, the height of the tree is log(n) and the number of nodes is n, so the space complexity is O(n).

For the RBT, the space complexity is also O(n) in the worst case, where n is the number of active rides. This is because the RBT maintains a balanced binary search tree, which has a height of log(n) in the worst case, and each node in the tree stores a constant amount of information.

Therefore, the overall space complexity of the given code is O(n), where n is the number of active rides.