COP 5536 - Advanced Data Structures

PROGRAMMING PROJECT - GatorTaxi

Name - Rahul Vikram Porwal UFID - 4459-0947 email id -rahulporwal@ufl.edu

Problem Statement

GatorTaxi is a ride-sharing service that receives many ride requests each day. The aim is to develop software to manage their pending ride requests efficiently. Each ride is characterized by a unique ride number, estimated cost, and total trip duration. We want to perform operations like printing a ride, printing a range of rides, inserting a ride, get next ride with least ride cost, cancelling a ride and updating the trip duration of a given ride.

I have used a minheap and a red black tree data structure to store the rides using ride cost and ride number as the keys, respectively. The minheap maintains the rides in the heap. The red black helps get the ride to be cancelled or printed based on the ridenumber in lesser time, since it uses ridenumber as the key. Since minheap is used to store rides using ride cost as the key, we can get next ride with minimum ride cost easily using the minheap. In this way, we can get an efficient solution to manage ride requests.

Files:

1. gatorTaxi.py - This is the main driver code for the gatortaxi service. It reads instructions from an input text provided by user as a command line argument and based on the instruction, performs one of the 6 functionalities defined in the problem statement.
2. minheap.py - This code provides implementation of the required functionalities using minheap data structure. The minheap class stores the rides in minheap using ride cost as the key. It also has an instance of red black tree to store the rides using ridenumber as the key.
3. RBTree.py - This code provides an implementation of the required functionalities using red black tree data structure. The RBT class stores rides in a red black tree using ride number as the key. The RBT nodes(rides) maintain a pointer to the position(index) of the ride in the minheap.
4. filewriter.py - This code is for writing the output to output_file.txt

Instructions to run:

1. Place the input text file with the instructions in the file where the codes are placed.
2. Run the following command in terminal to run the program and generate output_file.txt:

   python gatorTaxi.py <inputfilename.txt>

3. Running this program will generate an output_file.txt file which will have the output written into it.

Function Prototypes and Program Structure

1. **gatorTaxi.py** : Driver code. Creates an instance of File class, used to write output to output_file.txt and an instance of MinHeap class called heap, to store the rides and perform all required functionalities.
    a. **def insert(heap,ridenumber,ridecost,tripduration)** : returns heap after inserting the ride. Calls insert method in MinHeap() class to insert ridetriplet
    b. **def getnextride(heap)**: If there are no rides available in heap, it prints “no active rides” and writes this message to output file. Otherwise, it calls getnextride() method from MinHeap() class to get the next ride.
    c. **def cancelride(heap,ridenumber)**: If there are no rides in the heap returns nothing. Otherwise, calls deleteride(ridenumber) method in MinHeap() class to delete the ride from heap.
    d. **def printride(heap,ridenumber)**: calls printride(ridenumber) method in MinHeap() class to print the ride and write it to output_file.txt
    e. **def printrides(heap,ridenumber1,ridenumebr2)**: calls printrides(ridenumber1,ridennumber2) method in MinHeap() to print all the rides within the range of ridenumber1 to ridenumber2 and write it to output_file.txt
    f. **def updatetrip(heap,ridenumber,new_tripduration)** : calls update(ridenumber,newtripduration) method in MinHeap() class to update the trip duration

2. **minheap.py** : creates a class called MinHeap defined to implement
    a. **def __init__()** - class constructor to create object attributes for MinHeap() class. Creates attributes heap as an empty list [this list will hold the minheap], size to track the size of heap, rbt which is an instance of RBT() class to hold the red black tree and output file which is an instance of File() class.
    b. **def parent(i)** - gets parent of node located at index i in the minheap
    c. **def leftchild(i)** - gets left child of node located at index i
    d. **def rightchild(i)** - gets right child of node located at index i

e. **def getnumberofactiverides()** - returns current size of heap that is how many rides are there in the heap.
f. **def insert(triplet)** - inserts ride in the minheap using ride cost as key (appends in heap list), updates size of heap, calls method to insert triplet in red black tree (using ridenumebr as key) and heapifies the minheap after insertion. While heapify, it calls function to update pointers in RBT() for rides whose positions get swapped during heapify. If ride already exists in the tree, then print "duplicate ridenumber" and exit the program.
g. **def getnextride()** - gets ride with minimum cost which is at index 0, returns it to gatorTaxi.py, deletes it from minheap and calls method in RBT to delete it from RBT as well
h. **def deleteride(ridenumber)** - calls find(ridenumber) method in RBT() class and gets the index(position) of the ride in heap list (minheap). Then deletes the ride from red black tree and also from heap(minheap). Then performs heapify to maintain minheap property and calls updatepointer() method in red black tree to update pointers of all rides in minheap which get swapped during heapify.
i. **def printride(ridenumber)** - searches and retrieves the ride from Red Black Tree using find(ridenumber) method of RBT() class. Then writes the ride into output_file.txt by calling filewrite() method in File() class.
j. **def printrides(ridenumber1,ridenumber2)** - calls method printrides(ridenumber1,ridenumber2) in Red black tree to print all rides between ridenumber1 and ridenumber2 and write them to output file.
k. **def update(ridenumber,tripduration)** - If new trip duration is less than old trip duration, changes trip duration to new trip duration in both rbt and minheap. If new trip duration is greater than old trip duration but less than 2 times, then calls deleteride() to delete the ride and insert(ridenumber,ridecost+10,newtripduration). If new trip duration > 2 times old trip duration, delete the ride.

3. **RBTree.py**: creates a class called RBT defined to implement all the required functionalities of red black tree
    a. A nested class called **Node** is created within the red black tree which defines the nodes of the red black tree. Each Node has a ridenumber, ridetriplet, left,

right, color, and pointer attribute. The ridenumber is an integer that serves as the node's key in the tree. The ridetriplet is a tuple of three elements containing ride information. The left and right attributes are pointers to the node's left and right children. The color attribute can be either 'R' or 'B' representing red and black nodes, respectively. The pointer attribute is used to store a pointer to a node in a min-heap data structure.

b. **def __init__()** - class constructor for RBT(). Here, the object attributes stored will be the root of the Red Black Tree and an instance of File() class to write output into output_file.txt.
c. **def left_rotate(node)**-performs a left rotate at the given node in the tree.
d. **def right_rotate(node)**- performs a right rotate at the given node in the tree
e. **def flipcolors(node)** - flips the colors of the node and its children
f. **def insert_helper(node,ridenumber,ridetriplet)** - Starting with node, go through the tree to find the position of the external null node where it has to be placed, recursively. If the ride already exists, then exit the program. Otherwise perform red black tree insertion dealing with all insert cases.
g. **def insert(ridenumber,ridetriplet)** - call insert_helper(root,ridenumber,ridetriplet) to start searching for external node from the root of the tree. Insert helper will return the root of the tree after performing red black tree insertion. Set color of root to 'B'.
h. **def update_pointer(ridenumber,pointer_to_minheap)**- The update_pointer method updates the pointer of a node with a given ride number.
i. **def minimum(node)**- The minimum method finds the node with the minimum key in the tree rooted at node.
j. **def delete_node_helper(node,ridenumber)** - Starting with node passed as a parameter, search for node to delete in the tree rooted at node recursively. When node is found, delete it according to the rules of Red Black Tree and return root of the resultant tree.
k. **def delete(ridenumber)**-calls the delete_node_helper(root,ridenumber) function to pass the root of the tree as the starting point to start searching for node to delete. It will get the root of the tree resulting from delete operation.

l. **def find(ridenumber)** - searches for ride given ridenumber in RB tree returns the ride triplet and pointer_to_minheap if ride is found
m. **def update_tripduration(ridenumber,newtripduration)** - This function handles the case for rbt when new trip duration is less than old trip duration. The other 2 cases are handled by the delete and insert functions based on the update conditions.
n. **def get_inorder(node,res,ridenumber1,ridenumber2)** - Recursive inorder traversal to get nodes within range [ridenumber1,ridenumber2] and stores it in res list to return to printrides() function
o. **def printrides(ridenumber1,ridenumber2)** - calls getinorder method to get the list of all rides to print. Then prints the list. If list is empty, prints (0,0,0).

4. **filewriter.py**: Class File is defined to write output to "output_file.txt"
    a. **def __init__()**: opens output_file.txt and deletes any old content in the file
    b. **def filewrite(data)**: appends data to new line in output_file.txt

Time and Space Complexity Analysis

1. Time Complexity Analysis:
    a. Insert(Ridetriplet)
        i. Overall Time Complexity is O(log n), n is the number of rides. This includes time complexity of both insertion in minheap and red black tree.
        ii. Insertion in Minheap includes steps like appending to heap list and then heapify. Appending takes O(1) time. Heapify takes O(log n) since binary tree has height log(n).
        iii. Insertion in Red black tree will also take O(log n) time. So, totally it will take O(log n) time.
    b. CancelRide(Ridenumber)
        i. Overall Time Complexity - O(log n). This includes time complexity of deleting in both minheap and red black tree
        ii. Since we want to delete based on ridenumbers, we search the node to delete in the red black tree, which uses ridenumbers as the key to

store the rides. Then we delete it from red black tree and return the pointer_to_minheap of the node to the delete method in minheap. This deletion in red black tree takes O(log n) time.

iii. We get the index at which the ride is present in minheap from the above operation of searching and deleting in red black tree. So, we directly delete that ride from minheap list. This takes O(1) time. But, then we have to perform heapify to maintain minheap property. This takes O(log n) time. So, overall time complexity is O(log n)

c. Printride(ridenumber)
    i. Time complexity is O(log n). Here, finding the ride in red black tree and returning the index in minheap takes O(log n) time. O(log n) is the tree traversal time for balanced red black tree. Then saving in output and printing it takes constant time. So, overall time complexity is O(log n)

d. Printrides(ridenumber1,ridenumber2)
    i. Takes O(s + log(n)) time, s is number of rides available between ridenumber1 and ridenumber2.
    ii. Takes O(log n) time to search ridenumber1 in red black tree (tree traversal in balanced red black tree). Then once we find ride with ridenumber1, we have to check all rides in the right child subtree of the node of ridenumber1, which are less than or equal to ridenumber2. We do this by using inorder traversal. So, it takes O(s) time to check for s rides.
    iii. Hence, overall time complexity is O(log(n) + s).

e. getnextride()
    i. This also takes O(log n) time.
    ii. Extracting the first element that is minheap[0] takes O(1) time.
    iii. Then delete operation will take O(log n). [Analysis of cancelride function]

f. UpdateTrip(ridenumber,newtripduration)
    i. This operation also takes O(log n)

ii. In the first condition, updating old trip duration to new trip duration takes O(1). after updating, we need to check if the parent of updated node has same ride cost but greater trip duration. If it does, we need to swap. This will also take O(log n) since this operation needs to be done till we reach either root or we reach a node whose parent's ride cost is either less or if it is equal then trip duration is also less.

iii. So, overall time complexity will still be O(log n).

2. Space Complexity Analysis
    a. Red Black Trees - O(n), n is the number of rides. We construct a tree out of the n nodes(rides). This requires a space of O(n). All other variables we use for processing require constant space. So, overall space complexity is O(n).
    b. Minheap - O(n), n is the number of rides. We are using a 1D list to store minheap. Apart from this, all other variables we use take up constant space. So, overall space complexity will be O(n).