Wu Wei (wuw4)

I use DLB Trie combined with Indexable Priority Queue to ensure the efficiency of the operation.

Car.java

This is the class for car objects with general information: VIN number, make, model, color, price and mileage. In addition to those variables, there is an additional variable called index which can be used to directly access the car object in PQ.

VINdlb.java

This is the class to keep track of the VIN number of each car. There is an inner class called Node for the node object which contains four variables: char ‘c’ stores the character of VIN number, int ‘index’ which I will explain later, ‘next’ stores the sibling Node object and ‘down’ stores the child Node object. VIN number of each car is unique and the length is fixed 17 digits. Therefore, each time I want to add, update the information or remove a car, I traverse the DLB object to see if the VIN number exists. If exists, return the index value, if not, return -1 for updating or removing. Or put new VIN number to the structure, generate a new index value and return the index value for adding operation. What is index value? Since each car has a unique VIN, I generate a unique index value for each car which is the location of the object in PQ (indexable). Index value is just when the car is added (Ex: the index of the first car added to the structure is 0, the index of second car added to the structure is 1, etc.). Therefore I use a variable n to keep track the number of the VIN. What if we want to remove a car? We cannot simply reduce n by 1 since n only knows the number of the VIN. If we add a car after removing, then the collision of index value happens. To avoid the collision, I use two variables: an array of int called gaps and an int called rn. ‘gaps’ stores the index value that has been removed and ‘rn’ keeps track of the number of removed objects. Therefore when rn is greater than 0, I could assign the previously used index value to the new VIN number. What about the runtime? Since I have to traverse the dlb structure to get the index for both finding and adding, the average runtime would be O(17). Asymptotically, the runtime is O(1). What about memory? DLB structure saves a lot of memory compares to R-way Trie since the VIN has 17 characters and the lower level would be very sparse. R-way Trie would waste a lot of memory.

CarMakeModel.java

This class keeps track of the make and model combination using DLB Trie. Similar to VINdlb.java, this class build the DLB structure based on the make and model combination. I use the String (make + “\*” + model) to build the structure. “\*” is used to avoid the situation that one make has a prefix of other make. There is an inner class called CarNode which is used to create the DLB structure. CarNode class has four variables, “next” stores the sibling CarNode object, “down” stores the child CarNode object, char “c” stores the current character value and “mmpq” stores a Priority Queue object. “mmpq” objects only exist at the end-node. Each time we want to add/remove/retrieve a car, we traverse the DLB structure to find the relevant PQ object and add/remove/update it. For each car, the length of make and model combination would not be very long. Therefore the runtime to find a PQ object is O(k) where k is the length of the make model combination. Asymptotically, the runtime of finding the relevant PQ object would be O(1). About memory, since there are only countable makes and models, using R-way Trie will waste amount of memory since the lower level of the R-way Trie would be very sparse. Therefore using DLB Trie would save a lot of memory.

MinIndexPQ.java

This is a priority queue that could store Car objects. It uses min-heap that is created through an array. It saves space since I could represent a tree structure without creating many nodes. Therefore I create an array of Cars called “tree”. The root would be index 1, and left child is 2\*(parent index) and right child is 2\*(parent index) + 1. To implement indexable, I create an array of int called “pq” to store the relevant location in the “tree.” Since each car has a unique index, we can access the Car object directly through the array. Therefore the runtime of finding a Car using index is O(1). What about adding/deleting/updating? We perform the heap as a Binary tree, therefore the height of the “tree” is log(n). Each time we want to add/deleting/updating the car, we call the swim()/sink() to sort the heap, on both average and worst cases, the runtime would be the height of the tree-O(logn).

For adding procedure, I use VINdlb object to check if the VIN already exists. If yes, fail to add and give message. If not, generating relevant index value and call two MinIndexPQ objects and two CarMakeModel objects to add the car. Therefore, the runtime would be O(17) + O(logn), which is O(logn) on average.

For deleting procedure, I use VINdlb object to check if VIN exists and gets the index number. If index number is not -1 (means the car object exist), delete the VIN number in VINdlb object and delete the car objects. Therefore, the runtime would be O(17) + O(logn) for MinIndexPQ objects, O(17) + O(k) + O(logn) for CarMakeModel objects since I also need to traverse the DLB structure to find the PQ object for specific make and model. In general, it would all be O(logn) on average.

In conclusion, for updating procedure, I use VINdlb object to check if VIN exist and gets the index number. If the car exists, using index number to access the car and resort it after changing the information. Therefore, the runtime would be O(17) + O(logn) for MinIndexPQ objects, O(17) + O(k) + O(logn) for CarMakeModel objects, which would all be O(logn) on average.

For accessing the Lowest price/mileage of all cars, simply find the first car of MinIndexPQ objects, the runtime would be O(1).

For accessing the lowest price/mileage based on specific make and model, traverse the CarMakeModel objects to find the relevant MinIndexPQ object and access the first car in the heap. Therefore the runtime would be O(k) + O(1), since the length of make and model are usually not long in the real world, the runtime would be O(1) asymptotically.

The use of DLB structure would save a lot of memory since we do not waste any nodes we created. Unlike R-way Trie, the lower level would be sparse which means most of the nodes we created would be useless. In addition, DLB doesn’t limit the number of cars we can add, and we can add as many cars as we want to this CarTracker.