Advance Data Structures -Project Assignment

Name: Kalpana Sathya Ponnada
UFID: 52461920
Email: kponnada@ufl.edu

Files included

- gatorTaxi.java
- Heap.java
- HeapRide.java
- RBT.java
- RBRide.java
- makefile

Class Definition

- **gatorTaxi** : contains implementation of Insert, Print, UpdateTrip, CancelRide, GetNextRide.
- **Heap**: contains implementation of operations on **Heap** such as insert, deleteMin, arbitraryDelete, update.
- **HeapRide**: defines the node structure that contains various attributes such as rideNumber, rideCost, tripDuration, position, and a pointer to the corresponding node in the Red Black Tree.
- **RBT**: contains implementation of operations on **Red Black Tree** such as insert, search, search in a given range, delete.
- **RBRide**: defines the node structure that contains various attributes such as rideNumber, rideCost, tripDuration, pointers to parent node, left child, right child and a corresponding node in Heap.

Overview of Methods in each class

HeapRide.java

- **public HeapRide(int rNo, int rCost, int tDur, int pos)**
  - Constructor to initialize HeapRide objects

Heap.java

- **public Heap()**
  - Constructor to initialize Heap objects
- **public int parentPos(int pos)**
  - **returns** the position of the parent node in the heap.

- **public int leftPos(int pos)**
  - **returns** the position of the left child in the heap.
- **public int rightPos(int pos)**
  - **returns** the position of the right child in the heap.
- **public void swapRides(int ride1, int ride2)**
  - Swaps rides with ride numbers ride1 and ride2 in the heap.
- **public HeapRide addHeapRide(int rNo, int rCost, int tDur)**
  - **returns** a new HeapRide object with the required parameters.
- **public void addHeapify(HeapRide ride)**
  - inserts ride into Heap datastructure
  - Performs HeapfyTop operation
- **public void heapfyTop(int pos)**
  - Adjusts the heap from the given position to the root until the heap property is satisfied
- **public HeapRide deleteMin()**
  - Swaps root node with the last node in the heap
  - Now deletes the last node in the heap
  - Performs heapfyBot from root
  - **returns** the deleted node
- **public void arbitraryDelete(int pos)**
  - Swaps the node in the index 'pos' with the last node in the heap
  - Now deletes the last node in the heap
  - Performs heapfyBot
- **public void heapfyBot(int pos)**
  - Adjusts the heap from this position till the last level until the heap property is satisfied
- **public void updateTrip(HeapRide ride, int newTrDur)**
  - updates ride's trip duration
  - performs heapfyTop operation

RBRide.java

No Methods

RBT.java

- **public RBT()**
  - Constructor to initialize RBT objects
- **public RBRide addRBNode(int rNo, int rCost, int tDur)**
  - adds and returns RBRide object
- **public void addToTree(RBRide ride)**
  - adds node to the tree by following binary search tree property

- performs insertXYz to balance the tree

❖ **public void insertXYz(RBRide ride)**
- balances the tree after insertion and performs required rotations.

❖ **public RBRide searchRide(int rNo)**
- **returns** a ride with specified ride number in the tree

❖ **public RBRide searchRBT(RBRide ride, int rNo)**
- **returns** the ride with specified ride number in the tree

❖ **public ArrayList<String> searchRide(int rNo1, int rNo2)**
- **returns** a list of rides with ride numbers in the range[rNo1, rNo2]

❖ **public ArrayList<String> searchRange(RBRide ride, int rNo1, int rNo2, ArrayList<String> listOfRides)**
- searches for rides with ride numbers in the specified range[rNo1,rNo2] and inserts them into a list
- **returns** this list

❖ **public RBRide swapWithMinimum(RBRide ride)**
- **returns** the node with minimum ride number in the tree with root 'ride'

❖ **public void deleteRide(int rNo)**
- Deletes the ride with the specified ride number

❖ **public void deleteRide(RBRide ride, int rNo)**
- Deletes the ride with the specified ride number
- Performs deleteXCn

❖ **public void deleteXCn(RBRide ride)**
- balances the tree after deletion and performs required rotations.

❖ **public void replace(RBRide ride1, RBRide ride2)**
- Replaces ride1 with ride2

❖ **public void rRotation(RBRide ride)**
- performs right rotation

❖ **public void lRotation(RBRide ride)**
- performs left rotation

gatorTaxi.java

❖ **public gatorTaxi()**
- initializes tree and heap datastructures.

❖ **public void insert(int rNo, int rCost, int tDur)**
- inserts a ride into tree and heap.
- Performs balance and heapfy operations.

❖ **public void getNextRide()**
- performs deleteMin operation on heap
- Deletes the corresponding node in the tree
- Prints the Ride

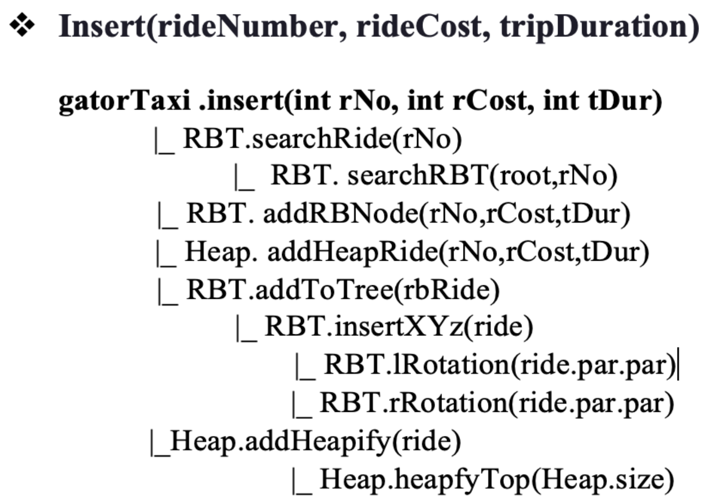
❖ **public void print(int rNo)**
- searches for the ride with specified ride number in the tree

- prints the ride

- **public void print(int rNo1, int rNo2)**
  - Searches for a list of rides in the specified range[rNo1, rNo2]
  - Iterates and prints each ride
- **public void cancelRide(int rNo)**
  - Searches for the ride with the specified ride number in the tree.
  - Deletes it from both the tree and the heap

- **public void updateTrip(int rNo, int updatedTDur)**
  - Searches for the ride with specified ride number
  - Updates the trip duration
  - Performs required operations based on the values of updatedDuration and originalDuration.
- **public static void main(String[] args)**
  - Interprets the input file
  - Each command in the file is an operation
  - Each operation writes to **output_file.txt**

Program Structure



- **Insert(rideNumber, rideCost, tripDuration)**

```
gatorTaxi .insert(int rNo, int rCost, int tDur)
        |_ RBT.searchRide(rNo)
               |_  RBT. searchRBT(root,rNo)
        |_ RBT. addRBNode(rNo,rCost,tDur)
        |_ Heap. addHeapRide(rNo,rCost,tDur)
        |_ RBT.addToTree(rbRide)
               |_ RBT.insertXYz(ride)
                    |_ RBT.lRotation(ride.par.par)
                    |_ RBT.rRotation(ride.par.par)
        |_Heap.addHeapify(ride)
                    |_ Heap.heapfyTop(Heap.size)
```

- **Print(rideNumber)**

gatorTaxi .print(int rNo)

```
        |_ RBT.searchRide(rNo)
              |_  RBT. searchRBT(root,rNo)
        |_ gatorTaxi.printRide(rNo, rCost, tDur)
```

- **Print(rideNumber1, rideNumber2)**

gatorTaxi .print(int rNo1, int rNo2)

```
        |_ RBT.searchRide(rNo1, rNo2)
              |_  RBT. searchRange(root,rNo1,rNo2,List)
```

- **GetNextRide()**

gatorTaxi.getNextRide()

```
        |_ Heap.deleteMin()
              |_  Heap.swapRides(0,Size)
              |_  Heap.heapfyBot(0)
        |_ RBT. deleteRide(rNo)
              |_ RBT.deleteRide(root,rNo)
                      |_ RBT.deleteXCn(ride)
                               |_ RBT.lRotation(ride.par.par)
                               |_ RBT.rRotation(ride.par.par)
        |_ printRide(rNo, rCost, tDur)
```

❖ **CancelRide(rideNumber)**

gatorTaxi.cancelRide(int rNo)

```
|_ RBT.searchRide(rNo)
        |_  RBT. searchRBT(root,rNo)
 |_ RBT. deleteRide(rNo)
        |_ RBT.deleteRide(root,rNo)
                 |_ RBT.deleteXCn(ride)
                           |_ RBT.lRotation(ride.par.par)
                           |_ RBT.rRotation(ride.par.par)
|_ Heap.arbitaryDelete(ride.Pos)
        |_  Heap.swapRides(pos,Size)
        |_  Heap.heapfyBot(pos)
```

❖ **UpdateTrip(rideNumber, new _tripDuration)**

gatorTaxi.updateTrip(int rNo, int updatedTDur)

```
|_ RBT.searchRide(rNo)
        |_  RBT. searchRBT(root,rNo)
//if updated duration > 2*tDur
 |_ gatorTaxi.cancelRide(rNo)
//else if tDur < updatedTDur < 2*tDur
  |_gatorTaxi.cancelRide(rNo)
  |_gatorTaxi.insert(rNo, rCost+10, updatedTDur)
// else if tDur > updatedTDur
|_ RBT.searchRide(rNo)
|_  Heap.updateTrip(rbNode.link,updatedTDur)
```

Complexity Analysis

Insert(int rNo, int rCost, int tDur)

- **Time Complexity :** O(l) + O(log n) + O(log n) + O(log n) = O(log n)
- **Space Complexity :**O(1)
- **Analysis:** Insertion into heap and red black tree requires performing operations in the worst case such as heapify and tree balancing, that take logarithmic time, O(logn). Overall, the time complexity of insert is O(logn). Space complexity, adding a single node to the heap or tree only requires constant space, so the space complexity is also O(1).

Print(int rNo)

- **Time Complexity :** O(log n)
- **Space Complexity :**O(1)
- **Analysis:** Searches for a ride with the specified ride number (rNo) and then prints it. Searching in the RBTree takes O(logn) time, which makes the overall time complexity O(logn). Searching does not require any extra space, so the space complexity is O(1).

Print(int rNo1, int rNo2)

- **Time Complexity :** O(log n+S) (where S is the number of rides in the heap whose ride numbers are in the range [rNo1, rNo2]).
- **Space Complexity:** O(S) (where S is the number of rides in the heap whose ride numbers are in the range [rNo1, rNo2]).
- **Analysis:** Searches for rides with ride numbers in the range of [rNo1, rNo2], and then prints all of these rides. The search operation in the RBTree takes O(logn) time, and the resulting rides are stored in a list. Then, the program iterates through this list and prints each ride. The overall time complexity of this operation is O(logn + S), where S is the number of rides within the specified range. The space complexity of this operation is O(S), since we store the resulting rides in a list.

UpdateTrip(int rNo, int updatedTDur)

- **Time Complexity :** O(logn)
- **Space Complexity:** O(1)
- **Analysis:** Searches for a ride with a specified ride number, and then updates its trip duration with the given value. Searching for a ride in the RBTree takes O(logn). In addition to search, we have three possible operations
    - If the updated trip duration is greater than 2*original duration, we perform cancelRide and that takes O(logn).
    - If the updated trip duration is between the original duration and 2*original duration, we perform cancelRide which takes O(logn) and insert a new ride, which takes O(logn)
    - If the updated trip duration is less than the original duration, we update the duration of the ride in both the heap and tree and that takes O(logn).

In all three cases, the overall time complexity is O(logn). Requires a constant space, so the space complexity is O(1).

CancelRide(int rNo)

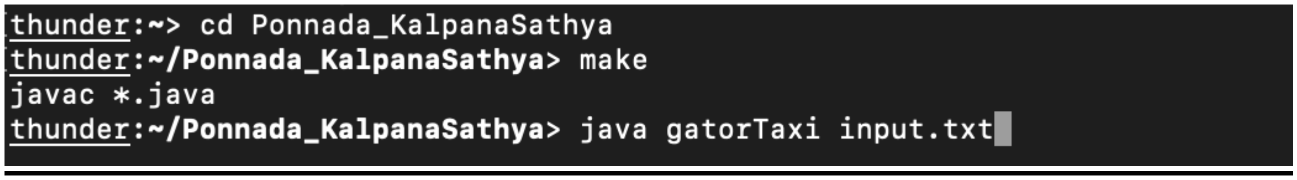
- **Time Complexity :** O(logn) + O(logn) + O(logn) = O(logn)
- **Space Complexity:** O(1)
- **Analysis:** Searches for a specific ride with a given ride number in the tree, which takes O(logn) time and it is deleted from the tree along with rebalancing in O(logn) time. Then, the corresponding heap node is deleted from the heap in O(logn) time. The overall time complexity is O(logn). Requires constant space, so the space complexity is O(1).

GetNextRide()

- **Time Complexity :** O(logn) + O(logn) = O(logn)
- **Space Complexity:** O(1)
- **Analysis:** Deletes the root from the heap and performs heapifyBot, which takes O(logn) time. Then, the corresponding node in the tree is found using the deleted node and is deleted from the tree, which also takes O(logn) time. Thus, the overall time complexity is O(logn). The operation does not require additional space and hence the space complexity is O(1).

How to run?

Once the zip file is extracted, run the below commands



```
thunder:~> cd Ponnada_KalpanaSathya
thunder:~/Ponnada_KalpanaSathya> make
javac *.java
thunder:~/Ponnada_KalpanaSathya> java gatorTaxi input.txt
```

Conclusion

Min Heap is a binary tree where each node's value is less than or equal to children values and all operations take O(logn) time. Red Black Tree is a balanced binary search tree where each node has either black or red colour and every operation takes O(logn) time.
In this problem we use these two data structures to perform the required operations. Rides are stored in the Min Heap based on the ride cost, while in the Red-Black Tree, rides are stored based on their ride numbers.
Corresponding nodes in the heap and tree are linked to each other. For each operation, we utilize both data structures but to minimize the complexity, we prioritize the data structure that is most appropriate for a specific operation. After performing the operation on the first data structure, we use the 'link' pointer to execute the same operation on the other data structure, thereby reducing the overall complexity.
The results are written to '**output_file.txt**'.