Advanced Data Structures (COP5536)

PROGRAMMING PROJECT REPORT

NAME: Popuri Haswanth

UFID: 21033164

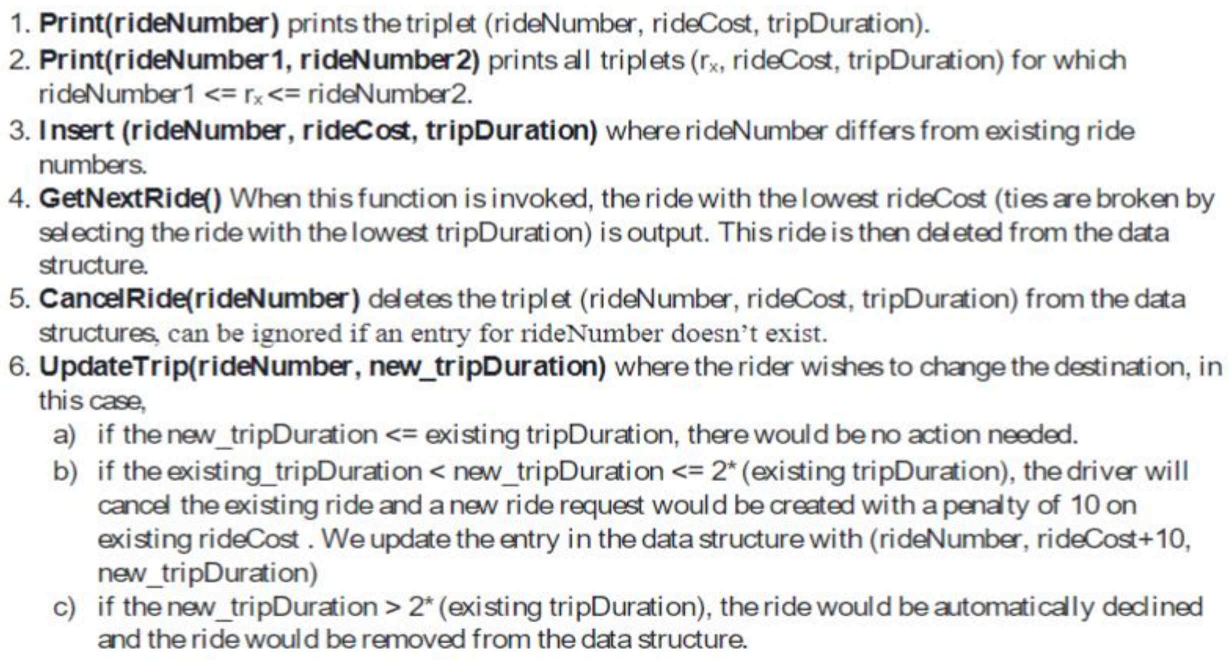
UFEmail : haswanthpopuri@ufl.edu

Problem Statement:

We are going to implement software to keep track of all the ride requests of gatorTaxi. We need to efficiently assign the rides using data structures like redblack tree and min heap.

A ride can be defined with the triplet of **(ridenumber, ridecost, tripduration)**

Required Functions to be implemented:

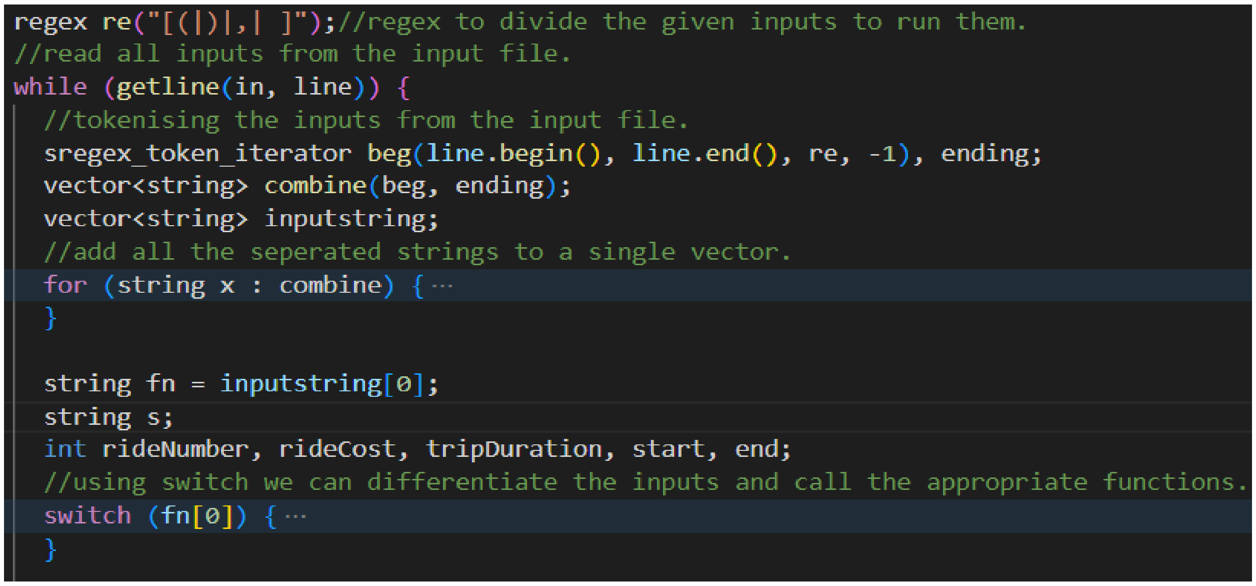


1. **Print(rideNumber)** prints the triplet (rideNumber, rideCost, tripDuration).
2. **Print(rideNumber1, rideNumber2)** prints all triplets (r_x, rideCost, tripDuration) for which rideNumber1 <= r_x <= rideNumber2.
3. **Insert (rideNumber, rideCost, tripDuration)** where rideNumber differs from existing ride numbers.
4. **GetNextRide()** When this function is invoked, the ride with the lowest rideCost (ties are broken by selecting the ride with the lowest tripDuration) is output. This ride is then deleted from the data structure.
5. **CancelRide(rideNumber)** deletes the triplet (rideNumber, rideCost, tripDuration) from the data structures, can be ignored if an entry for rideNumber doesn't exist.
6. **UpdateTrip(rideNumber, new_tripDuration)** where the rider wishes to change the destination, in this case,
   a) if the new_tripDuration <= existing tripDuration, there would be no action needed.
   b) if the existing_tripDuration < new_tripDuration <= 2* (existing tripDuration), the driver will cancel the existing ride and a new ride request would be created with a penalty of 10 on existing rideCost . We update the entry in the data structure with (rideNumber, rideCost+10, new_tripDuration)
   c) if the new_tripDuration > 2* (existing tripDuration), the ride would be automatically declined and the ride would be removed from the data structure.

Inputs: All the inputs for the problem are in the following manner:

Insert (rideNumber, rideCost, tripDuration)
Print(rideNumber)
Print (rideNumber1, rideNumber2)
UpdateTrip(rideNumber, newTripDuration)
GetNextRide()
CancelRide(rideNumber)

We need to take inputs from the input text file and read them into strings and then separate them using the regex and then adding them to the list and using switch we need to call the appropriate functions based on the string. The code for taking inputs is as below:



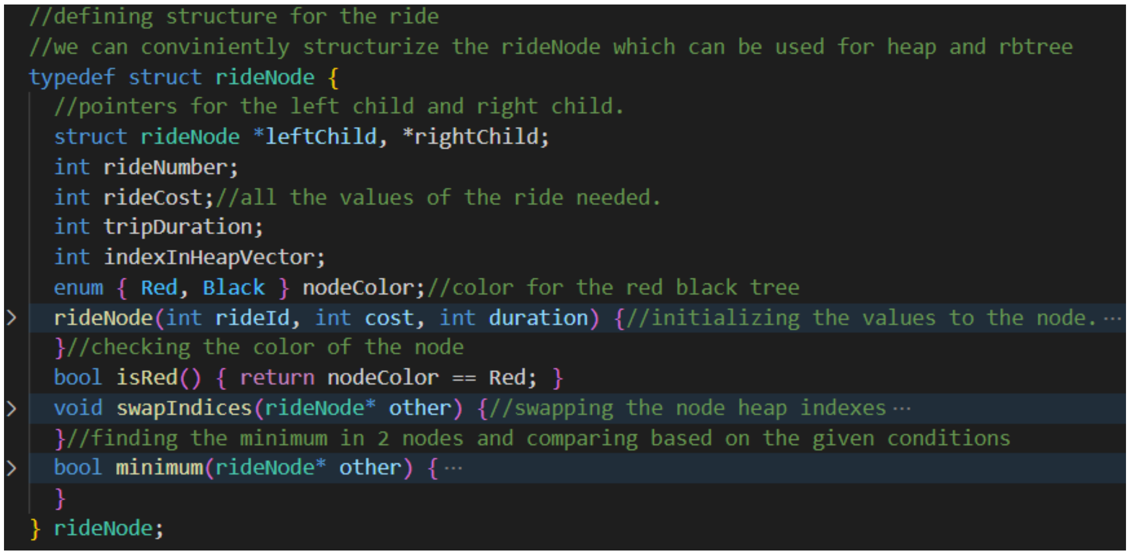
```cpp
regex re("[(|)|,| ]");//regex to divide the given inputs to run them.
//read all inputs from the input file.
while (getline(in, line)) {
    //tokenising the inputs from the input file.
    sregex_token_iterator beg(line.begin(), line.end(), re, -1), ending;
    vector<string> combine(beg, ending);
    vector<string> inputstring;
    //add all the seperated strings to a single vector.
    for (string x : combine) {···
    }

    string fn = inputstring[0];
    string s;
    int rideNumber, rideCost, tripDuration, start, end;
    //using switch we can differentiate the inputs and call the appropriate functions.
    switch (fn[0]) {···
    }
```

Defining the Node Structure:

We have defined a node structure to be convenient with both the minheap and red black tree we need to maintain the heap index so we included it to the node. We also need to keep a color for the sake of red black tree.

We also have created constructor to initialize the values and then written some functions which are further used everywhere required. The structure of the node here is important in order to delete from minheap and then also to keep track of the color and red black tree. Here we also included the comparator function in the struct itself for ease of the usage.



```cpp
//defining structure for the ride
//we can conviniently structurize the rideNode which can be used for heap and rbtree
typedef struct rideNode {
    //pointers for the left child and right child.
    struct rideNode *leftChild, *rightChild;
    int rideNumber;
    int rideCost;//all the values of the ride needed.
    int tripDuration;
    int indexInHeapVector;
    enum { Red, Black } nodeColor;//color for the red black tree
    rideNode(int rideId, int cost, int duration) {//initializing the values to the node.···
    }//checking the color of the node
    bool isRed() { return nodeColor == Red; }
    void swapIndices(rideNode* other) {//swapping the node heap indexes···
    }//finding the minimum in 2 nodes and comparing based on the given conditions
    bool minimum(rideNode* other) {···
    }
} rideNode;
```

Data structures required in the problem statement:

i) **Minheap:**
Heap is a data structure where the child nodes have more value than that of their parents and the root is the smallest of all the values in the heap. The heap looks like below:

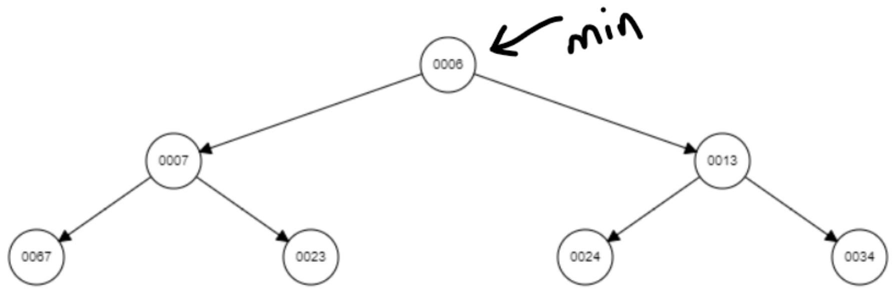
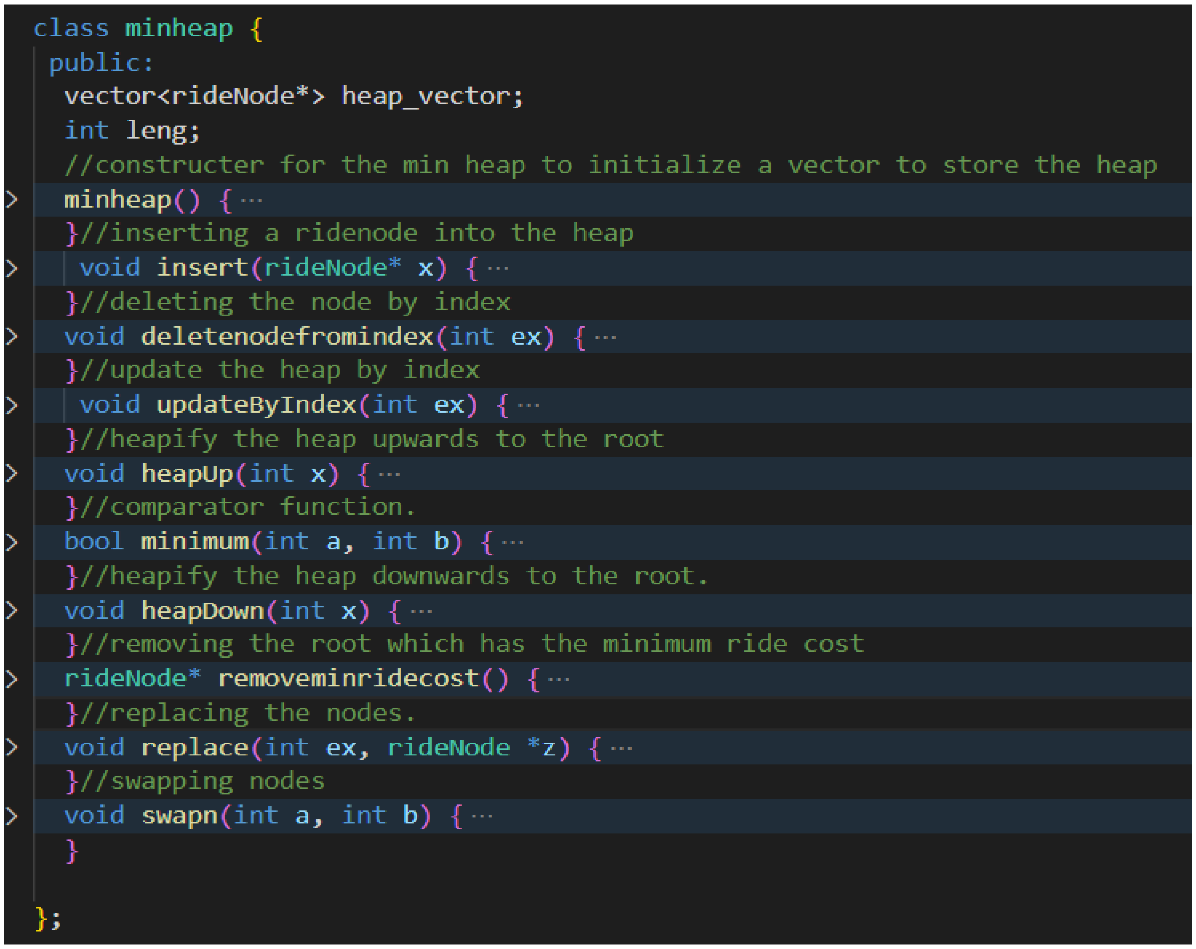


Fig: min heap structure

We must create a min heap which has the minimum ride cost and if the ride cost is equal then we need to take the one with minimum trip duration. So, we can implement the heap class as below:



```
class minheap {
 public:
  vector<rideNode*> heap_vector;
  int leng;
  //constructer for the min heap to initialize a vector to store the heap
  minheap() {…
  }//inserting a ridenode into the heap
   void insert(rideNode* x) {…
  }//deleting the node by index
  void deletenodefromindex(int ex) {…
  }//update the heap by index
   void updateByIndex(int ex) {…
  }//heapify the heap upwards to the root
  void heapUp(int x) {…
  }//comparator function.
  bool minimum(int a, int b) {…
  }//heapify the heap downwards to the root.
  void heapDown(int x) {…
  }//removing the root which has the minimum ride cost
  rideNode* removeminridecost() {…
  }//replacing the nodes.
  void replace(int ex, rideNode *z) {…
  }//swapping nodes
  void swapn(int a, int b) {…
  }

};
```

We have created a class for the minheap and then implemented the insert delete from index and all the required functions.

Heapifyup : we here heapify the nodes upwards to the root. The time complexity of this is **O(logn).**

Heapifydown : we here heapify the nodes downwards in the subtree of the heap. The time complexity of this is **O(logn).**

Insertion takes **O(logn)** time in the min heap.

To get the **minimum** we take only **O(1)** time since the root contains the minimum value.

To **delete** an element also takes **O(logn)** time.

These are the most important functions since we need to delete an arbitrary ride from the heap. We get the index from the red black tree.

Red Black Tree:

A red black tree is a binary search tree which is self balancing with rules such as :

i)Root must be black and also null leaves are black

ii)No 2 nodes in a path can be continuously re d.

iii)Number of black nodes from root to every leaf must be same.

The below given figure depicts a red black tree. The root 29 is black and from the root to leaf every path had 3 blacks where the last leaf will also be black.

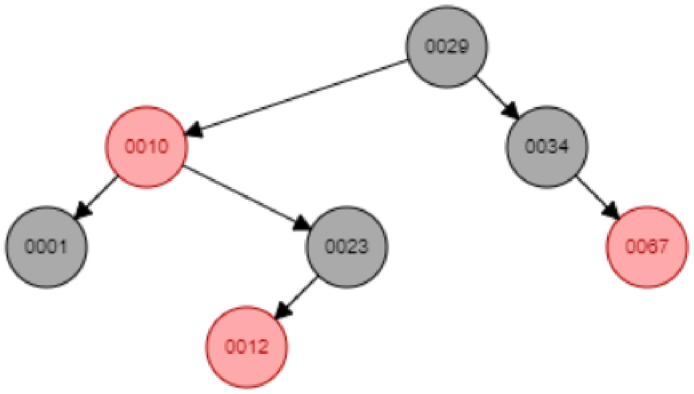
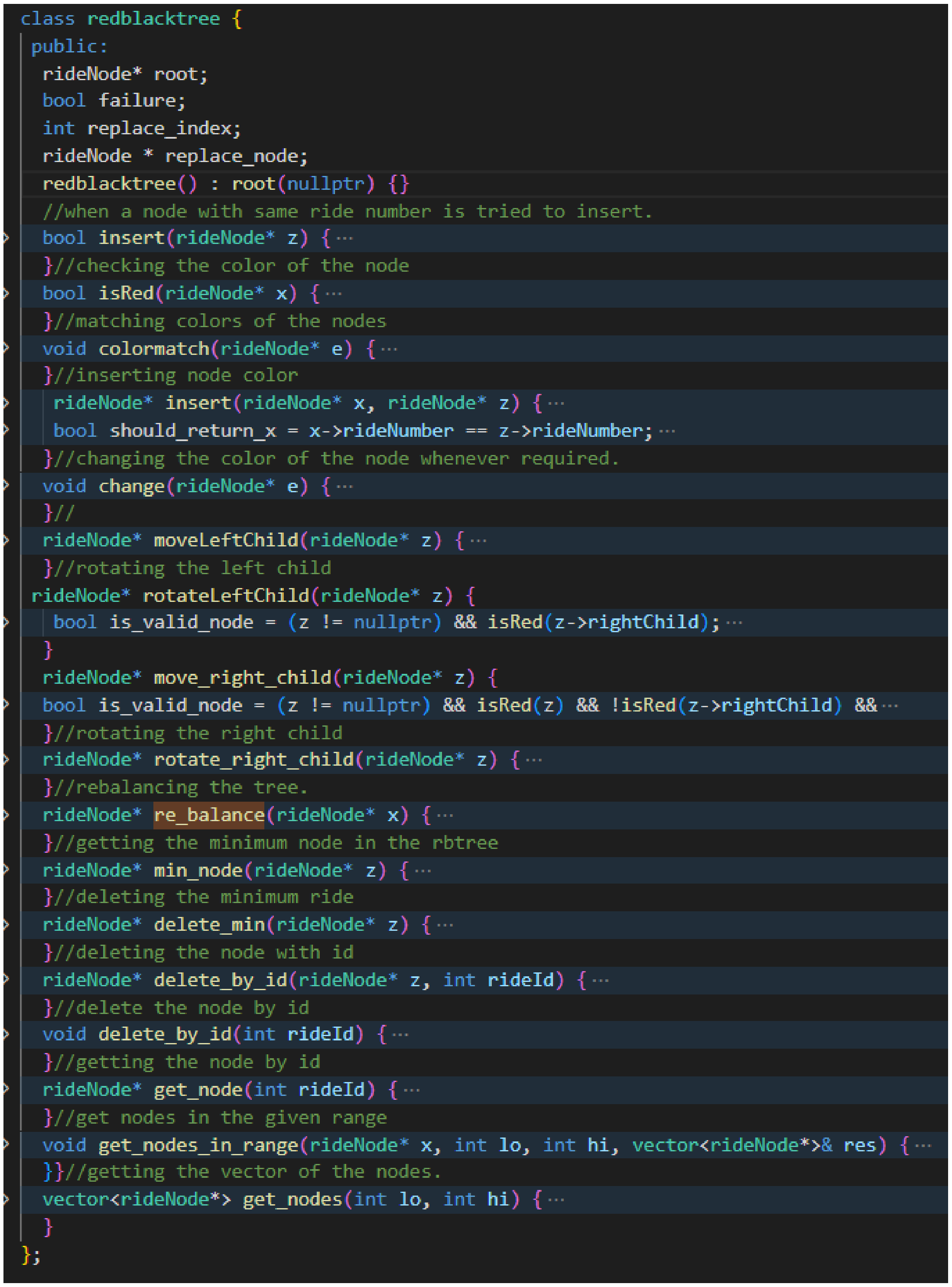


Fig: red black tree

To **insert** into a red black tree it takes **O(logn)** time

To **delete** from the red black tree it takes **O(logn)** time

To **search** an element in red black tree it takes **O(logn)** time

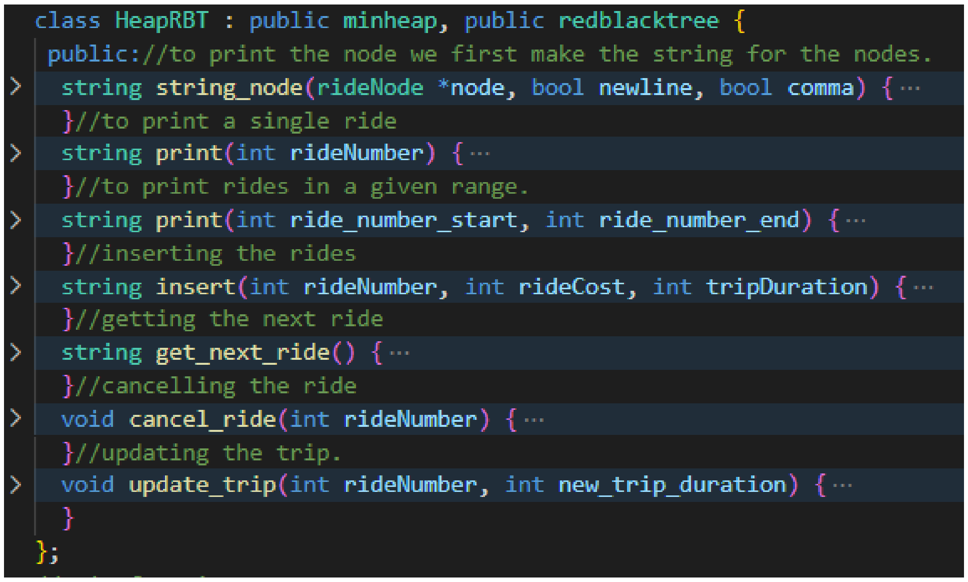


```
class redblacktree {
 public:
  rideNode* root;
  bool failure;
  int replace_index;
  rideNode * replace_node;
  redblacktree() : root(nullptr) {}
  //when a node with same ride number is tried to insert.
  bool insert(rideNode* z) {···
  }//checking the color of the node
  bool isRed(rideNode* x) {···
  }//matching colors of the nodes
  void colormatch(rideNode* e) {···
  }//inserting node color
   rideNode* insert(rideNode* x, rideNode* z) {···
   bool should_return_x = x->rideNumber == z->rideNumber;···
  }//changing the color of the node whenever required.
  void change(rideNode* e) {···
  }//
  rideNode* moveLeftChild(rideNode* z) {···
  }//rotating the left child
 rideNode* rotateLeftChild(rideNode* z) {
   bool is_valid_node = (z != nullptr) && isRed(z->rightChild);···
  }
  rideNode* move_right_child(rideNode* z) {
  bool is_valid_node = (z != nullptr) && isRed(z) && !isRed(z->rightChild) &&···
  }//rotating the right child
  rideNode* rotate_right_child(rideNode* z) {···
  }//rebalancing the tree.
  rideNode* re_balance(rideNode* x) {···
  }//getting the minimum node in the rbtree
  rideNode* min_node(rideNode* z) {···
  }//deleting the minimum ride
  rideNode* delete_min(rideNode* z) {···
  }//deleting the node with id
  rideNode* delete_by_id(rideNode* z, int rideId) {···
  }//delete the node by id
  void delete_by_id(int rideId) {···
  }//getting the node by id
  rideNode* get_node(int rideId) {···
  }//get nodes in the given range
  void get_nodes_in_range(rideNode* x, int lo, int hi, vector<rideNode*>& res) {···
  }}//getting the vector of the nodes.
  vector<rideNode*> get_nodes(int lo, int hi) {···
  }
};
```

The above class is implemented to get the functions like printing nodes in a given range. Searching the rides whenever required. As heap keeps track of the rides based on the minimum cost. We just need to recursively iterate through the red black tree like an **inorder** traversal and get all the rides in the given range in **(log(n) + S)** time.

Now the main interlinking class HeapRBT :

We need to interlink the classes of Heap and RBT by extending the min heap and red black tree. We need to use both the data structures combinedly to get an efficient usage of the rides based on the given conditions.



```
class HeapRBT : public minheap, public redblacktree {
 public://to print the node we first make the string for the nodes.
  string string_node(rideNode *node, bool newline, bool comma) {···
  }//to print a single ride
  string print(int rideNumber) {···
  }//to print rides in a given range.
  string print(int ride_number_start, int ride_number_end) {···
  }//inserting the rides
  string insert(int rideNumber, int rideCost, int tripDuration) {···
  }//getting the next ride
  string get_next_ride() {···
  }//cancelling the ride
  void cancel_ride(int rideNumber) {···
  }//updating the trip.
  void update_trip(int rideNumber, int new_trip_duration) {···
  }
};
```

Print rideNumber : we need to traverse through the red black tree and find and get the ride from the tree so this takes **O(logn)** time.

Print ride_number1, ride_number2: We traverse through the red black tree inorder traversal and then append all the nodes in the range to a vector and then print the rides in the vector. This takes **O(logn)** for traversing and S time for printing the vector S is the number of triplets in the range given. So the time complexity is **O(logn + S)**

Insert(rideNumber, rideCost, tripDuration): To insert a ride we need to insert it into the heap and red black tree. Where insertion in both takes **O(logn) .**

GetNextRide(): Since we get the minimum cost ride from the heap in **O(1).**But we need to delete the ride from the red black tree and the heap which both takes **O(logn)** time.

CancelRide rideNumber: We need to first search the ride from the red black tree which takes **O(logn)** time then we can get the index of it in the heap in **O(1)** then we need to delete the ride from both the red black tree and the min heap which takes **O(logn).** So the this operation takes **O(logn)**

UpdateTrip rideNumber, tripDuration : We first need to search for that ride in the red black tree which is **O(logn)** time. Then we can get the index of it in the heap in **O(1)** and then we can update it in the heap in **O(logn).**

Conclusion : The given requirements are satisfied,implemented all the datastructures needed in required time complexities. Learnt the implementation of RedBlackTree and Minheap