COP 5536 Spring 2023

Programming Project

Raj Shukla (UFID: 86317877)

raj.shukla@ufl.edu

Project Description

GatorTaxi is a ride-sharing service. The main objective is to provide a new software solution to keep track of the pending ride requests.

A ride is identified by the following triplet:

rideNumber: unique integer ID for each ride.
rideCost: The estimated cost for the ride.
tripDuration: the total time (in integer minutes) needed to get from pickup to destination.

These ase the key components utilized to designing our abstract Data structures which are later utilized to serve certain application functionalities.

The needed Application functions are:

1. **Print(rideNumber)** prints the triplet (rideNumber, rideCost, tripDuration). If the mentioned triplet is not present we need to print "(0,0,0)".

2. **Print(rideNumber1, rideNumber2)** prints all triplets (rx, rideCost, tripDuration) for which rideNumber1 <= rx <= rideNumber2 appropriately seperated by "," except after the last triplet. If the mentioned triplet is not present we need to print "(0,0,0)".

3. **Insert (rideNumber, rideCost, tripDuration)** where rideNumber differs from existing ride numbers. This produce no output unless rideNumber is a duplicate in which case application outputs an error and stop.

4. **GetNextRide()** When this function is invoked, the ride with the lowest rideCost (A tie are broken by selecting the ride with the lowest tripDuration) is output. This ride is then deleted from the data structure after displaing the required triplet.

5. **CancelRide(rideNumber)** deletes the triplet (rideNumber, rideCost, tripDuration) from the data structures, can be ignored if an entry for rideNumber doesn't exist. This will not print anything.

6. **UpdateTrip(rideNumber, new_tripDuration)** where the rider wishes to change the destination, in this case,

   1. If the new_tripDuration <= existing tripDuration, must update the new_tripDuration for the specified rideNumber, but it will not update the rideNumber or rideCost.
   2. 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)

3. If the new_tripDuration > 2*(existing tripDuration), the ride would be automatically declined and the ride would be removed from the data structure.

The UpdateTrip(rideNumber, new tripDuration) will not output anything.

Abstract Data Structure Requirement:

As per the use case the application is implemented utilizing a min-heap and a Red-Black Tree (RBT). (Assuming that the number of active rides will not exceed 2000)

A min heap should be used to store (rideNumber, rideCost, tripDuration) triplets ordered by rideCost. If there are multiple triplets with the same rideCost, the one with the shortest tripDuration will be given higher priority (given all rideCost-tripDuration sets will be unique). An RBT should be used to store (rideNumber, rideCost, tripDuration) triplets ordered by rideNumber. The application maintain pointers between corresponding RBT. Min heap is build on top of dynamic array. C++17 is utilized. (MinGw compiler opted)
Header files used:

- Iostream : process standard output
- Fstream : process input output file (file handling)
- String : processing parsed input file.
- Vector : utilized to build heap on top of dynamic array.

I/O requirement:

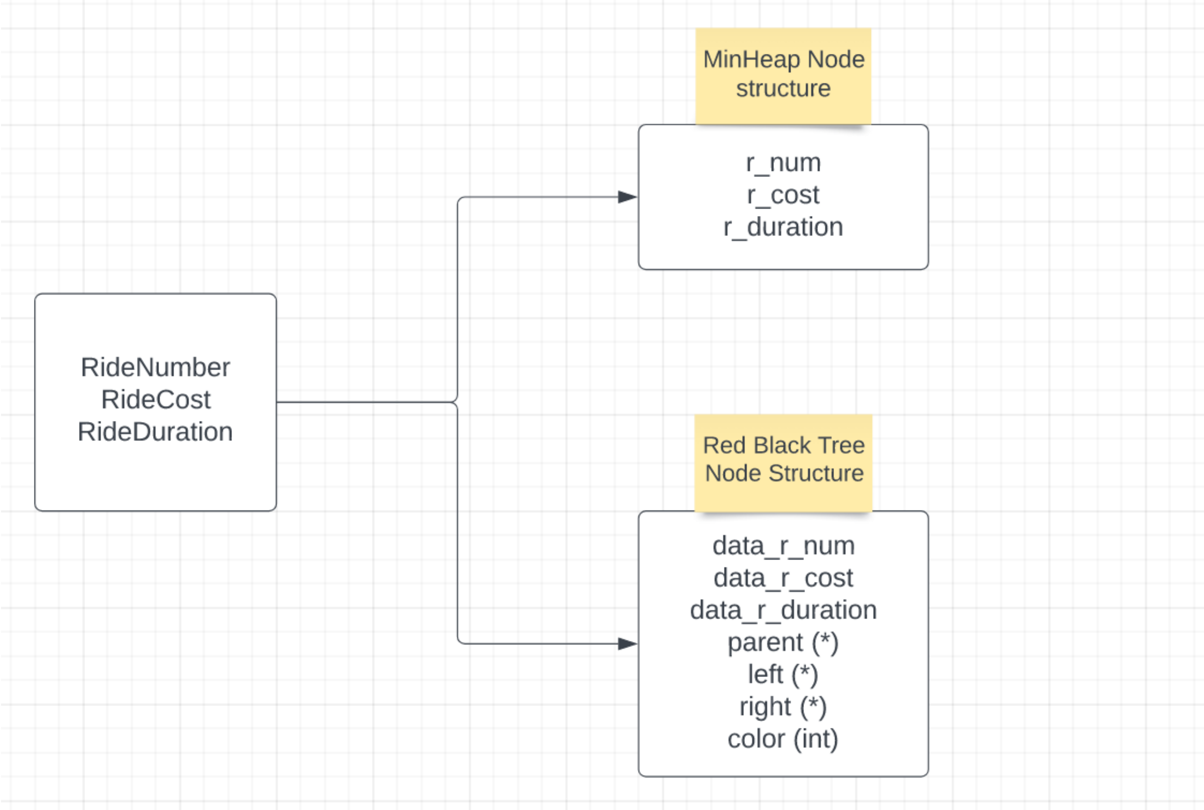
Execution is done post making executable on the test system the following way:
$ ./ gatorTaxi file_name

Here <file_name> is the inut text file provided. Input file can contain commands from:

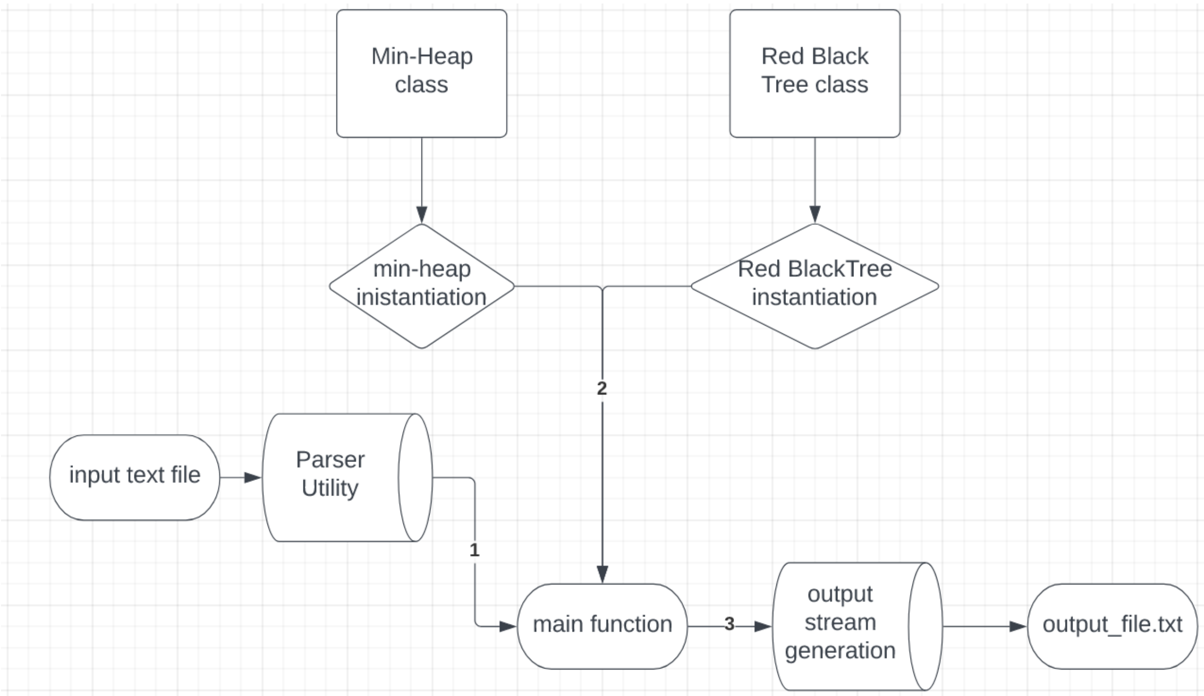
Insert(rideNumber, rideCost, tripDuration)
Print(rideNumber)
Print (rideNumber1,rideNumber2)
UpdateTrip(rideNumber, newTripDuration)
GetNextRide()
CancelRide(rideNumber)

Output requirement: Each command in the input must follow the functionality output requiement as mentioned in the application functionality need. All output goes to a file named "**output_file.txt**".

Abstract Data structure Node structure: The following is how a node in min-heap and RBT in our application looks like:

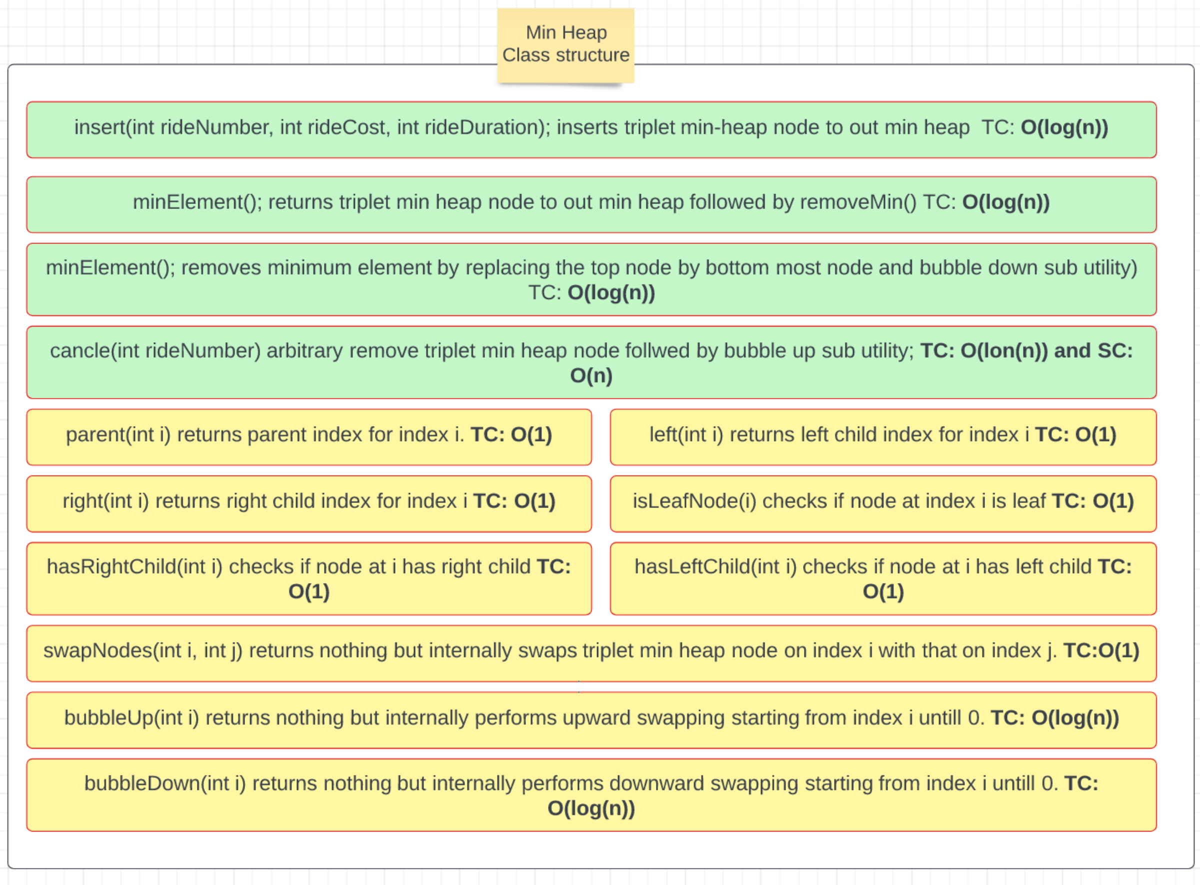


Application component Architecture diagram: (Helps in analyzing the component connections)
Program flow structure



Following is the class structure with all function prototypes showing the structure of the programs. Each function is described with its use and complexities.

Min-Heap



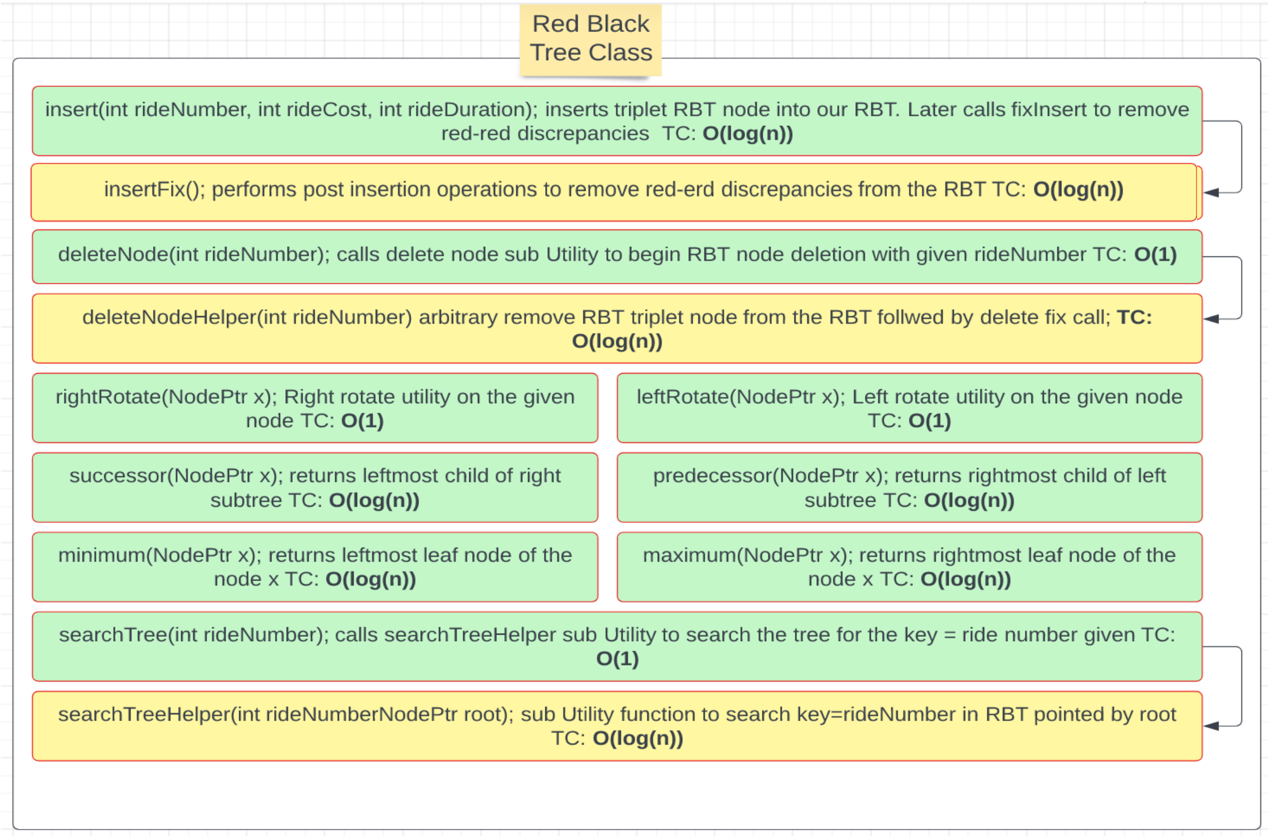
Please note that the Space complexities in each will be constant if not mentioned otherwise. Note that the diagram fully establishes the connection between class structure and the provided code. Indicators:

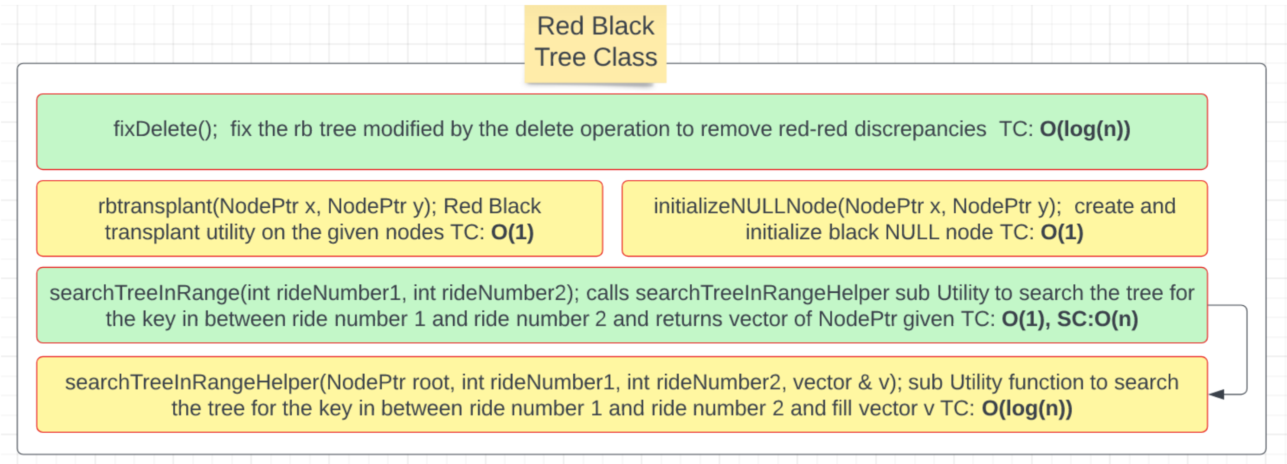
TC: Time complexity

SC: Space complexity

Green is the public class methods and yellow are the sub utilities which are private in nature to control class data abstraction and data hiding.

Red Black Tree





Please note that the Space complexities in each will be constant if not mentioned otherwise. Note that the diagram fully establishes the connection between class structure and the provided code. Indicators:
TC: Time complexity
SC: Space complexity
Green is the public class methods and yellow are the sub utilities which are private in nature to control class data abstraction and data hiding.
NodePtr: RBTNode pointer

Appendix to refer insertion and deletion cases in Red Black Tree: . (Assuming K is the target node, S is sibling Node, P is parent node, U is uncle node and G is grandparent node for the provided RBT)

Insertion of node K:

To insert a node K into a red-black tree T, we do the following. (Assuming S is sibling Node, P is parent node, U is uncle node and G is grandparent node for the provided RBT)

1. We insert K using an ordinary BST insertion operation.
2. We color K node red.
3. We check if the insertion violated the red-black tree properties. If it did, we fix it.

Cases:

Case 1: Tree is empty.

Case 2: P (parent) is black.

Case 3: P (parent) is red.

Case 3.1: P(parent node) is red and U is red too.

Case 3.2: P(parent) is red and U is black (or NULL)

Case 3.2.1: P(parent node) is right child of G and curent K is right child of P

Case 3.2.2: P(parent node) is right child of G and current K is left child of P

Case 3.2.3: P(parent) is left child of G and current K is left child of P

Case 3.2.4: P (parent) node is left child of G and current K is right child of P.

Pseudo code to resolve all cases:

```
RB-INSERT(T, k)
   BST-INSERT(T, k) //normal BST insertion
   while k.parent.color == RED
     if k.parent == k.parent.parent.right
       u = k.parent.parent.left //uncle
       if u.color == RED // case 3.1
         u.color = BLACK
         k.parent.color = BLACK
         k.parent.parent.color = RED
         k = k.parent.parent
       else if k == k.parent.left // case 3.3.1 and 3.3.2
           k = k.parent
           LEFT-ROTATE(T, k)
         k.parent.color = BLACK
         k.parent.parent.color = RED
         RIGHT-ROTATE(T, k.parent.parent)
     else (same as then clause with "left" and "right" exchanged)
   T.root.color = BLACK
```

Deletion of node K:

Case 1: K(target node) is a red node

Case 2: K(target node) has a red child

Case 3: K(Target node) is a black node

Case 3.1: K's sibling S is red

Case 3.2: K's sibling S is black, and both of S's children are black.

Case 3.3: K's sibling S is black, S's left child is red, and S's right child is black.

Case 3.4: K's sibling S is black, and S's right child is red.

Pseudo code:

```
RB-DELETE(T, x)
  BST-DELETE(T, x)
  while x ≠ T.root and x.color == BLACK
    if x == x.parent.left
      s = x.parent.right
      if s.color == RED
        s.color = BLACK // case 3.1
        x.parent.color = RED // case 3.1
        LEFT-ROTATE(T, x.parent) // case 3.1
        s = x.parent.right // case 3.1
      if s.left.color == BLACK and s.right.color == BLACK
        s.color = RED // case 3.2
        x = x.parent //case 3.2
      else if s.right.color == BLACK
          s.left.color = BLACK // case 3.3
          s.color = RED //case 3.3
          RIGHT-ROTATE(T, s) // case 3.3
          s = x.parent.right // case 3.3
        s.color = x.parent.right // case 3.4
        x.parent.color = BLACK // case 3.4
        s.right.color = BLACK // case 3.4
        LEFT-ROTATE(T, x.parent) // case 3.4
        x = T.root
    else (same as then close with "right" and "left" exchanged)
  x.color = BLACK
```

BST-DELETE and BST-INSERT are normal binary search tree insert and delete operation.