Advanced Data Structures (COP 5536)

Name: Anvesh Gupta
UFID: 8927-7005
Email Address: anvesh.gupta@ufl.edu

Problem Description

GatorTaxi is an up-and-coming ride-sharing service. They get many ride requests every day and are planning to develop new software to keep track of their pending ride requests.
A ride is identified by the following triplet:
rideNumber: unique integer identifier for each ride.
rideCost: The estimated cost (in integer dollars) for the ride.
tripDuration: the total time (in integer minutes) needed to get from pickup to destination.
In order to complete the given task, you must use a min-heap and a Red-Black Tree (RBT).

Program Structure

The entire project has 7 files Myheap.h , MyRBT.h , gatorTaxi.cpp , input.txt ,output_file.txt ,stringUtils.h and Makefile.

Flow and Structure of Program

I have firstly included a makefile that compiles the source code and produces an executable file.

```
CC = g++

# executables
gatorTaxi: gatorTaxi.cpp MyRBT.h MyHeap.h
	$(CC) MyHeap.h
	$(CC) MyRBT.h
	$(CC) gatorTaxi.cpp -o gatorTaxi


clean:
	$(RM) $(TARGET)
```

The program is then implemented in “gatorTaxi.cpp” in C++ programming language using object-oriented programming concepts. The program reads the input file, which contains a list of ride requests and their properties, and processes each request by calling the appropriate function. The program then writes the output to an output file.

The GatorTaxi program has several functions that can be called by the user to perform different operations. Some of these functions include:

void print(): This function prints the details of a ride request based on the ride number.

```
void print(RBT *rbt, Heap *heap, int *args, string &output) {       //function to print output
cout << "print: " << args[0] << endl;
output += rbt -> range(args[0], args[0]);
}
```

void printRange() : This function is used to print range.

```
void printRange(RBT *rbt, Heap *heap, int *args, string &output) {      //function to print range of output
    cout << "printRange: " << args[0] << " " << args[1] << endl;
    output += rbt -> range(args[0], args[1]);
}
```

void insertRide(RBT *rbt, Heap *heap, int* args, string &output) : This function is used to insert ride details by the user into RBTNode and HeapNode.

```
void insertRide(RBT *rbt, Heap *heap, int* args, string &output) {
   //this function is used to insert the ride using node RBTNode and HeapNode
    cout << "insert: " << args[0] << " " << args[1] << " " << args[2] << endl;
    RBTNode* rnode = new RBTNode(nullptr, nullptr, nullptr, nullptr, RED, args[0], args[1], args[2]);
    HeapNode* hnode = new HeapNode(nullptr, args[0], args[1], args[2]);
    rnode -> heapNode = hnode;
   hnode -> rbtNode = rnode;

 if(rbt -> findNode(rnode -> rideNumber, rbt -> root)) {

        output += "Duplicate RideNumber\n";
        return;
    }

    rbt -> insert(rnode);   //insert
    heap -> push(hnode);    //push operation

    rbt -> printTree();   //print tree
    heap -> printHeap();  //print heap
}
```

void getNextRide () :-This function gets the next best ride request by taking the node from the heap.

```
void getNextRide(RBT *rbt, Heap *heap, string &output) {
    cout << "getNextRide" << endl;
    if(heap -> getSize() == 0) {
        output += "No active ride requests\n";
        return;
    }
    HeapNode* deletedHeapNode = heap -> pop();
    deletedHeapNode -> printHeapNode(output);
    rbt -> deleteNode(deletedHeapNode -> rbtNode -> rideNumber);

    cout << endl;
    heap -> printHeap();
    rbt -> printTree();
}
```

cancelRide(): This function cancels a ride request by deleting the corresponding node from the red-black tree and removing the node from the heap.

```
void cancelRide(RBT *rbt, Heap *heap, int* args, string &output) {
    cout << "cancelRide: " << args[0] << endl;
    RBTNode* deletedRBTNode = rbt -> deleteNode(args[0]);
    heap -> remove(deletedRBTNode -> heapNode);
    heap -> printHeap();
    rbt -> printTree();
}
```

updateTrip(): This function updates the details of a ride request. It has three conditions based on which it either inserts, cancels, or updates the ride request.

```
void updateTrip(RBT *rbt, Heap *heap, int* args, string &output) {
    int newTD = args[1];
    cout<< "updateTrip: " << args[0] << " " << newTD << endl;
    RBTNode* node = rbt -> findNode(args[0], rbt -> root);
    if(node) {
        int existingTD = node -> tripDuration;
        if(newTD <= existingTD) {
            node -> tripDuration = newTD;
            heap -> update(node -> heapNode , node -> rideCost, newTD);
            heap -> printHeap();
            rbt -> printTree();
        } else if(existingTD < newTD && newTD <= 2 * existingTD) {
            cancelRide(rbt, heap, args, output);
            args[1] = node -> rideCost + 10;
            args[2] = newTD;
            insertRide(rbt, heap, args, output);
        } else if(newTD > 2 * existingTD) {
            cancelRide(rbt, heap, args, output);
        }
    } else cout << "No trip to update" << endl;
}
```

parseInput(): This function reads the input file, processes it, and writes the output to an output file.

```
void parseInput(RBT *rbt, Heap *heap, int argc, char **argv) {
    fstream inputFile;
    inputFile.open(argv[1], ios::in);
    string output;

    if(inputFile.is_open()){
        string fileLine;
        while(getline(inputFile, fileLine)){
            string operationPrefix = fileLine.substr(0, 3);
            processOperation(rbt, heap, fileLine, operationPrefix, output);
        }
        inputFile.close();
    }

    ofstream MyFile("output_file.txt");
    MyFile << output;
    MyFile.close();
}
```

int main(int argc, char **argv): This function prints the details of all ride requests within a given boundary.

```
int main(int argc, char **argv) {   //main function
    RBT *rbt = new RBT();
    Heap *heap = new Heap();

    if(argc >= 2) {
        parseInput(rbt, heap, argc, argv);
        return 0;
    } else cout << "Insufficient arguments - Please provide input file" << endl;
    return 0;
    RBTNode *rnode;
    HeapNode *hnode;
    heap -> push(new HeapNode(nullptr, 10,11,12));
    heap -> push(new HeapNode(nullptr, 20,9,12));
    heap -> push(new HeapNode(nullptr, 30,9,9));

    heap -> push(new HeapNode(nullptr, 20,8,12));
    HeapNode *h = new HeapNode(nullptr, 30,9,8);
    heap -> push(h);
    heap -> printHeap();heap -> remove(h);
    heap -> printHeap();

}
```

The program uses two main classes: “MyRBT.h” and “MyHeap.h”. The “MyRBT.h” class implements the red-black tree data structure to store ride requests.

```
#include<iostream>        // to initalize header file and to insert data into a stream (output) or extract data
from a stream (input) in c++
#include<math.h>
using namespace std;    class RBTNode;            // declaration of the class RBTNode, which is used later in
the code.
class HeapNode {           //initialization of HeapNode with default values
public :
RBTNode *rbtNode;
int index, rideNumber, rideCost, tripDuration;
HeapNode() {
index = -1;
rideNumber = 0;
rideCost = 0;
tripDuration = 0;
rbtNode = nullptr;
    }
HeapNode(RBTNode *rbtNode, int rideNumber, int rideCost, int tripDuration) {
this -> index = -1;
this -> rbtNode = rbtNode;
this -> rideNumber = rideNumber;
this -> rideCost = rideCost;
this -> tripDuration = tripDuration;
    }

void HeapNodeprint() {                          // printing output of index , rideNumber
cout << "(" << this -> index << ", " << this -> rideNumber << ", " << this -> rideCost << ", " << this ->
tripDuration << ")";
}
void HeapNodeprint(string &output) {                          // for printing output in of rideNumber ,
rideCost , tripDuration
output += "(" + to_string(this -> rideNumber) + "," + to_string(this -> rideCost) + "," +  to_string(this ->
tripDuration) + ")\n";
}
};

class Heap {                // defining heap class
int back;
HeapNode* heap[2000];    // defining array of fixed size 2000 as constraint mentioned in project
public:
Heap() {          // it has a default constructor that initializes the "back" data member to 0
back = 0;
}

int size() {              //back returns the size
return back;
}

void push(HeapNode *heapNode) {        // for operation push
```

and the “MyHeap.h” class implements the heap data structure to keep track of the best ride request available at a given time.(providing a sample of code as it is too long)

```
#include "MyHeap.h"    // including the file MyHeap.h that is included in this file
#include <queue>       // initializing queue header
#define RED 0         // Initalizing red black tree variables and giving them initial values
#define BLACK 1
#define L 0
#define R 1

class RBTNode {                   //defining class RBTNode which will give initialize the default variables and
will initalize them for this class with keyword "this"
public :
    RBTNode *leftChild, *rightChild, *parent;
    HeapNode *heapNode;
    bool color;
    int rideNumber, rideCost, tripDuration;

    // Initializes external nodes
    RBTNode() {
        this -> leftChild = nullptr;
        this -> rightChild = nullptr;
        this -> parent = nullptr;
        this -> heapNode = nullptr;
        this -> color = BLACK;
        this -> rideNumber = 0;
        this -> rideCost = 0;
        this -> tripDuration = 0;
    }
    //This function initializes the  non-external RBT nodes
    RBTNode(RBTNode *leftChild, RBTNode *rightChild, RBTNode *parent, HeapNode* heapNode,bool color, int
rideNumber, int rideCost, int tripDuration) {
        this -> leftChild = leftChild != nullptr ?        // terneray operator is used to assign values to
leftChild and rightChild
        leftChild : new RBTNode();
        this -> rightChild = rightChild != nullptr ?
        rightChild : new RBTNode();
        this -> parent = parent;
        this -> heapNode = heapNode;
        this -> color = color;
        this -> rideNumber = rideNumber;
        this -> rideCost = rideCost;
        this -> tripDuration = tripDuration;
    }

    // This function initializes the non-external RBT nodes from an existing node
    RBTNode(RBTNode *node) {
        this -> leftChild = node -> leftChild != nullptr ?
            node -> leftChild : new RBTNode();
        this -> rightChild = node -> rightChild != nullptr ?
            node -> rightChild : new RBTNode();
```

```
void RB0(RBTNode *y, RBTNode *py) {  //defines a function RB0 that takes in two pointers to RBTNodes y and py as
arguments and does not return anything. The function is a part of the process of adjusting a Red-Black Tree
(RBT) data structure to maintain its properties after deleting a node.
        if(y == root) {   //checks if y is equal to the root of the tree. If it is, the color of y is set to
BLACK and the function returns.
            y -> color = BLACK;
            return;
        }

        RBTNode *v = py -> leftChild;  //If y is not equal to the root of the tree, the function declares a
pointer to an RBTNode v that points to the left child of py.

        if(py -> color == BLACK) {    //The function then checks if the color of py is BLACK. If it is, the
color of v is set to RED. If it is not, the color of py is set to BLACK and the color of v is set to RED. The
function then returns.
            v -> color = RED;
        } else {
            py -> color = BLACK;
            v -> color = RED;
            return;
        }
        adjustRBTAfterDelete(py, py -> parent);  //If the color of py was initially BLACK, the function calls
itself recursively with arguments py and the parent of py. This continues the process of adjusting the tree
after deleting a node.
    }

    void RB1Cases(RBTNode *y, RBTNode *py, RBTNode *a) {   //defines a function RB1Cases that takes in three
pointers to RBTNodes y, py, and a as arguments and does not return anything. The function is part of the process
of adjusting a Red-Black Tree (RBT) data structure to maintain its properties after deleting a node.
        if(a == RED) {   //The function checks if the color of a is RED. If it is, the function calls another
function named RB11 with arguments y and py.
            cout << "RB11 case" << endl;
            RB11(y, py);
        } else {   //If it is not, the function calls another function named RB12 with the same arguments.
            cout << "RB12 case" << endl;
            RB12(y, py);
        }
    }

    void RB11(RBTNode *y, RBTNode *py) {   //function declares two pointers to RBTNodes v and a. v points to the
left child of py and a points to the left child of v.
        RBTNode *v = py -> leftChild, *a = v -> leftChild;  //The function then sets the color of a to BLACK,
the color of v to the color of py, and the color of py to BLACK.
        a -> color = BLACK;
        v -> color = py -> color ;
        py -> color = BLACK;
        LLRotation(py);
    }
```

Conclusion:

GatorTaxi is a ride-hailing service that utilizes a red-black tree to organize ride requests and a heap to track the most favorable request at any given moment. The program, written in C++, employs object-oriented programming principles and includes various functions that can be invoked by the user to execute different tasks. It reads an input file, processes the data, and outputs the results to an output file.