ADVANCED DATA STRUCTURES

NAME: VIJAY THAKKALLAPALLY
UF ID: 93902982
UFMAIL: thakkallapally.v@ufl.edu

PROBLEM STATEMENT:

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.

Introduction:

I implemented minheap and redbalck tree to store the details of rides and to retrieve the ride details based on the requirement of functions for gatortaxi in optimal time.

MinHeap stores the details based on ride cost giving priority to the ride with minimum ride cost if two rides has same ride cost then it assigns priority to the ride with minimum trip duration.

Redblack tree is used to store details of rides with key as rideNumber. Redblack tree is used to extract the details of rides in optimal time complexity based on gatortaxi functions.

PROPERTIES OF MINHEAP:

Each parent node in a min-heap has a value that is less than or equal to that of its child nodes. Here are some properties of a min-heap:

i) **Shape property**: A complete binary tree is known as a min-heap. This suggests that every level of the tree is fully occupied, maybe except for the last level, which is occupied from left to right.

ii**) Heap property**: The parent nodes in a heap data structure have values that are smaller than or equal to the values of their child nodes. Consequently, the smallest value is always found at the root of the heap.

iii) **Insertion property**: When an element is inserted to the heap, it is added at the last level of the tree and then "bubbled up" until it satisfies the heap property.

iv) **Deletion property**: When the minimum element is removed from the heap, the last element in the heap is moved to the root and then "bubbled down" until it satisfies the heap property.

PROPERTIES OF REDBALCK TREES:

Self-balancing binary search trees with the following characteristics are known as red-black trees:

i) It is either red or black for each node.

ii) The root color is black.

iii) The leaf nodes color is all black (NIL).

iv) A node's children are black if it is red, and vice versa.

v) The count of black nodes along each path from a node to its descendent leaf nodes is constant.

These properties ensure that the tree is balanced, with the maximum height no more than twice the minimum height. Here are some additional properties of red-black trees:

i) The worst execution time of search and insert functions are O(log n).

ii) Red-black trees are used in many algorithms, such as map and set data structures.

iii) Red-black trees can be efficiently implemented using a small number of pointers per node.

iv) Red-black trees can be augmented with additional data in each node, such as the length of the subtree at that node as root, to support efficient order statistics queries.

v) The operations to maintain the red-black tree properties (e.g., color flips and rotations) are relatively simple and can be easily implemented using a few pointer manipulations.

PROJECT STRUCTURE:

1) Class : **RideDetails**

```
class RideDetails
{                                                   //class to store ride details
public:
    int rideNumber;
    int rideCost;
    int tripDuration;
    RideDetails(){}
    RideDetails(int rideNumber, int rideCost, int tripDuration)        //constructor to initialize the ride details
    {
        this->rideNumber = rideNumber;
        this->rideCost = rideCost;
        this->tripDuration = tripDuration;
    }
    int compareTo(const RideDetails& other_ride) const {               // method to compare between two ride details
        if (this->rideCost == other_ride.rideCost ) {
            return this->tripDuration - other_ride.tripDuration;
        }
        return this->rideCost - other_ride.rideCost;
    }

    bool operator==(const RideDetails &other_ride) const {                 // overloaded '==' operator to compare two ride details
        return (this->rideCost == other_ride.rideCost &&
        this->tripDuration == other_ride.tripDuration &&
        this->rideNumber == other_ride.rideNumber);
    }

};
```

This class is used to store the details of ride and it has class members as rideNumber, rideCost, tripDuration to store each value.

- Constructor of same class is used to initialize the values of all the members of that class.
- compareTo method is used to compare between two ride details based on ride cost if ride costs are equal then compares based on tripduration.
- Operator '==' it is an overloaded function to compare two rides for equality that is if all the values are equal or not.

2) Class : **RedBlackTreeNode**

This class is used to create nodes for redblack tree in order to implement it and it has class members as object of ridedetails class, objects of pointyers of same class they are parent, left, right and node-color to store color of node.

```
class RedBlackTreeNode                          // class to store redblacktree node details
{
private:
    static const bool RED = true;
    static const bool BLACK = false;

public:
    RideDetails ride_details;
    int rideNumber;
    RedBlackTreeNode *parent;
    RedBlackTreeNode *left;
    RedBlackTreeNode *right;
    bool node_colour; // black = False red = True

    RedBlackTreeNode(int rideNumber, int ridCost, int tripDuration)              // constructor to initialize the node details
    {
        this->ride_details = RideDetails(rideNumber, ridCost, tripDuration);
        this->rideNumber = rideNumber;
        this->node_colour = RED;
    }

    void set(RedBlackTreeNode *rbt_node)
    {
        this->rideNumber = rbt_node->rideNumber;
        this->ride_details = rbt_node->ride_details;
    }

    int compareTo(RedBlackTreeNode *other_node)                    // method to compare two RBT nodes
    {
        return (this->ride_details.compareTo(other_node->ride_details));
    }
};
```

- Constructor “RedBlackTreeNode” is used to initialize the node values of the tree and set method is used to assign the ride details to the nodes.

- compareTo method is used to compare between two tree nodes to find the relation of equal or less than or greater than.

3) class : **MinHeap**

This class is used to implement minheap functionality and here I used templates to generalize the implementation to any data type. It has the class member called ‘heap_values’ It is a vector used to store the heap values and used to perform all the heap operations. This class also has methods they are insert, getMin, removeMin, isEmpty, deleteNode.

i) **insert(T rbt_node):**

This method inserts a new element into the heap. It adds the new element to the end of the vector and then reorders the heap to maintain the heap property (i.e., the parent node must be smaller than its children in a min-heap). It does this by repeatedly swapping the new

element with its parent until the heap property is satisfied. The running time of this method is O(log n).

ii) **getMin():**

This method returns the minimum element in the heap (i.e., the root of the heap). It simply returns the first element in the vector, which should be the minimum element if the heap is maintained properly. The running time of this method is O(1).

iii) **removeMin():**

This method removes the minimum element from the heap (i.e., the root of the heap). It does this by swapping the root with the last element in the vector, removing the last element, and then reordering the heap to maintain the heap property. It does this by repeatedly swapping the new root with its smallest child until the heap property is satisfied. The running time of this method is O(log n).

iv) **isEmpty():**

This method returns a boolean indicating whether the heap is empty or not. It simply checks if the vector storing the heap values is empty or not. Running time of this method is O(1).

v) **deleteNode(T rbt_node**):

This method deletes a given node from the heap. It first finds the index of the node to be deleted in the vector and then swaps it with the last element in the vector, removes the last element, and reorders the heap to maintain the heap property. It does this by repeatedly swapping the new element with its smallest child until the heap property is satisfied. If the new element is smaller than its parent, it swaps with its parent until the heap property is satisfied again. Running complexity of this method is O(log n).

vi**) heapify(int index):**

The heapify function is used to maintain the heap property after an element is removed or added. The heapify function takes a heap and an index as input, and recursively performs a swap operation with the smallest child node until the heap property is satisfied. After the minimum element is removed, the last element in the heap is placed at the root position, and the minheap property may be violated. The heapify function is called on the root node to restore the heap property. The heapify function works by comparing the value of the root with its right and left child nodes. If the value of the left child node is lesser than the root node, a swap is performed between the root and the left child. If value of right child node is smaller than root node, a swap is done between the root and the right child. The running complexity of the heapify function is O(log n). This is because the heapify function performs swaps and recursive calls at most log n times, where each operation takes constant time.

4) Class **: RedBlackTree**

The RedBlackTree class has a RedBlackTreeNode pointer for the root node of the redblack tree and a null_node, which represents a node with a value of 0, and it is used for boundary conditions. It also has two constant boolean variables, RED and BLACK, representing the color of the nodes.

i) RedBlackTree() :

This is the constructor for the RedBlackTree class. It initializes the null_node of the tree with a RedBlackTreeNode object that has a value of 0, a key of 0, and a node color of BLACK. It also sets the root of the tree to be equal to null_node.

ii) rotateLeft(RedBlackTreeNode *rbt_node) :

To balance the Red-Black Tree, this function rotates the node passed as an argument to the left. It begins by determining whether the provided node is nullptr or null node. If it is, it just goes back. If not, it makes the passed node's right child the rightNode. After that, it makes the passed node's left child equal to the right child of the rightNode. If the rightNode's left child is not null node, the passed node is set as the child's parent. The passed node's parent is then set to be the rightNode's parent The rightNode is set as the tree's root if the parent of the given node is nullptr. The left child of the parent is set as the rightNode if the passed node is the left child of its parent. If not, the rightNode is set to be the parent's right child. Lastly, the passed node is set as the left child of the rightNode, and the rightNode is set as the passed node's parent.

iii) rotateRight(RedBlackTreeNode *rbt_node) :

To balance the Red-Black Tree, this function rotates the node passed as a parameter to the right. It begins by determining whether the provided node is nullptr or null node. If it is, it just goes back. If not, it configures the leftNode to be the given node's left child. After that, it makes the passed node's right child equal to the left child of the leftNode. The passed node is set as the right child of the leftNode if the right child is not null node. The passed node's parent is then set to be the leftNode's parent. The leftNode is set as the tree's root if the parent of the provided node is nullptr. The right child of the parent is set to be the leftNode if the passed node is the right child of its parent. If not, the leftNode is set to be the parent's left child. Last but not least, the passed node is set as the parent of the passed node and as the right child of the leftNode.

iv) insertFixUp(RedBlackTreeNode *rbt_node) :

This method is called after inserting a new node into the tree in order to balance it. It takes a pointer to the inserted node as a parameter. The method starts by initializing a pointer to the uncle of the inserted node. It then enters a loop that continues as long as the parent of the inserted node has a node color of RED. If the parent of the inserted node is the right child of its parent, then the uncle of the inserted node is the left child of the grandparent of the inserted node. If the color of the uncle of the inserted node is RED, then the colors of the parent and uncle of the inserted node are set to BLACK, and the color.

V**) transplant (RedBlackTreeNode *replacingNode, RedBlackTreeNode *newReplacement) :**

The "transplant" method replaces the subtree rooted at node u with the subtree rooted at node v. It is a helper method used in the "delete" method. The method updates the parent pointers and links of the nodes in the tree correctly to ensure that the replacement is successful. This method is typically used when deleting a node with one or no children.

vi) deleteNode(RedBlackTreeNode *rbt_node):

This method deletes a given node from the Red-Black Tree. If the deleted node is a leaf node, the tree simply removes it. If the node only has one child, the removed node is replaced with that child. If the node has two children, its successor is found and replaces the deleted node. If the successor is a child of the deleted node, it is replaced by its own child, otherwise, the successor is removed from its original position and replaces the deleted node. This method also calls the deleteFixUp method to ensure that the properties of the Red-Black Tree are maintained after the deletion.

vii) deleteFixUp(RedBlackTreeNode *rbt_node) :

This method in the provided code is used to keep the properties of the Red-Black Tree after a node has been deleted. When a node is deleted, the tree may violate properties of the Red-Black Trees, specifically the properties of balance and color. It takes one argument, node, which is the node that was deleted from the tree. It starts by checking if the node is null, which would indicate that there are no more nodes to fix up. If the node is not null, it enters a loop that continues until the node is the root of the tree or is black.

viii) minimum (RedBlackTreeNode *rbt_node)**:**

This method finds the node with the lowest value in a tree's sub-section. It achieves this by continuously moving to the left child of each node until it reaches the bottommost level of the tree.

ix) search(key): This method searches in the Red-Black Tree with the given key. It does so by starting at the root and traversing the tree until it finds the node with the given key or reaches a leaf node. If the key is not found, it returns None.

5) class: **gatorTaxi**

This class is used to implement the actual functionality of the gator taxi given in the problem statement. The gatorTaxi class represents a ride-sharing service that provides functionality for managing ride requests. It uses two data structures - a min heap and a red-black tree to store and manage ride details.

```
bool insert(int rideNumber, int rideCost, int tripDuration)
{
    RedBlackTreeNode *rbt_node = new RedBlackTreeNode(rideNumber, rideCost, tripDuration);
    if (redblacktree.search(rbt_node->ride_details.rideNumber) == redblacktree.null_node)
    {
        min_Heap.insert(rbt_node->ride_details);
        redblacktree.insert(rbt_node);
        return true;
    }
    else
    {
        return false;
    }
}
```

Time complexity: O (log n)

The insert method is used to add a new ride request to the data structures. It takes three integer parameters rideNumber, rideCost, and tripDuration, and creates a new RedBlackTreeNode object to store the ride details. It then inserts the ride details into both the min heap and the red-black tree. If the ride already exists, it returns false, otherwise it returns true.

```
RideDetails getNextRide()
{
    RideDetails ride_details = min_Heap.removeMin();                    // method to get the next available ride
    RedBlackTreeNode *rbt_node;
    if (ride_details.rideNumber !=0 && ride_details.rideCost!=0 && ride_details.tripDuration !=0)
    {
        rbt_node = redblacktree.search(ride_details.rideNumber);
        redblacktree.deleteNode(rbt_node->ride_details.rideNumber);
        return ride_details;
    }
    else
    {
        // handle if rbt_node == NULL => "No Active ride requests" should be printed
        return RideDetails(0,0,0);
    }
}
```

Time complexity: O (log n)

The getNextRide method is used to retrieve the next available ride request. It removes the minimum element from the min heap using the removeMin method of the MinHeap class and returns the corresponding RideDetails object. It also removes the corresponding node from the red-black tree using the deleteNode method of the RedBlackTree class because this ride is used and it should no longer available.

```
void cancelRide(int rideNumber)
{
    RedBlackTreeNode *rbt_node = redblacktree.search(rideNumber);
    if (rbt_node != NULL)
    {
        min_Heap.deleteNode(rbt_node->ride_details);
        redblacktree.deleteNode(rideNumber);
    }
}
```

Time Complexity: O (log n)

The cancelRide method is used to cancel a ride request. It takes a single integer parameter rideNumber and removes the corresponding ride details from both the min heap and the red-black tree using the deleteNode method of the MinHeap and RedBlackTree classes, respectively because this ride is cancled.

```
    void updateTrip(int rideNumber, int newTripDuration)
    {
        RedBlackTreeNode *rbt_node = redblacktree.search(rideNumber);
        if (rbt_node == NULL)
        {
            return;
        }
        min_Heap.deleteNode(rbt_node->ride_details);
        if (rbt_node->ride_details.tripDuration >= newTripDuration)
        {
            rbt_node->ride_details.tripDuration = newTripDuration;
            min_Heap.insert(rbt_node->ride_details);
        }
        else if (newTripDuration < rbt_node->ride_details.tripDuration * 2)
        {
            rbt_node->ride_details.tripDuration = newTripDuration;
            rbt_node->ride_details.rideCost += 10;
            min_Heap.insert(rbt_node->ride_details);
        }
        else
        {
            redblacktree.deleteNode(rideNumber);
        }
    }
};
```

Time Complexity : O(log n)

The updateTrip method is used to update the details of a ride request. It takes two integer parameters rideNumber and newTripDuration, and updates the trip duration of the corresponding ride request. If the new trip duration is less than or equal to the original trip duration, it updates the ride details in the min heap and the red-black tree using the deleteNode and insert methods of the MinHeap and RedBlackTree classes, respectively. If the new trip duration is less than twice the original trip duration, it updates the ride details in the min heap and the red-black tree and increases the ride cost by 10. If the new trip duration is greater than or equal to twice the original trip duration, it removes the corresponding ride details from the red-black tree using the deleteNode method of the RedBlackTree class.

```
RedBlackTreeNode *print(int rideNumber)
{
    RedBlackTreeNode *rbt_node = redblacktree.search(rideNumber);
    if (rbt_node == NULL || rbt_node == redblacktree.null_node)
    {
        rbt_node = new RedBlackTreeNode(0, 0, 0);
    }
    return rbt_node;
}
```

Time Complexity: O(log n)

The print method is used to retrieve information about a ride request. It takes a single integer parameter rideNumber and returns the corresponding RedBlackTreeNode object containing details about the ride. If no such ride exists, it returns a new RedBlackTreeNode object with default values.

```
vector<RedBlackTreeNode *> print(int rideNumber1, int rideNumber2)
{
    vector<RedBlackTreeNode *> arr = redblacktree.rangePrint(rideNumber1, rideNumber2);
    if (arr.empty())
    {
        RedBlackTreeNode *rbt_node = new RedBlackTreeNode(0, 0, 0);
        arr.push_back(rbt_node);
    }
    return arr;
}
```

Time Complexity : O(log n + S)

The print method will accept two integer parameters rideNumber1 and rideNumber2 and returns a vector of RedBlackTreeNode objects containing ride details in the given range. If no such rides exist, it returns a vector containing a single RedBlackTreeNode object with default values.

main function :(int main (int argc, char *argv[]))

The program takes arguments from command-line argc and argv, which are passed to the function. The first argument (argv[0]) is the program name, and the next argument (argv[1]) is input file name that the program should read from. If argc is 1, then it prints a message and stops.

The program constructs an object of the gatorTaxi class, that is a red-black tree data structure that stores RideDetails objects. The program then opens the input file to read the input and an output file (output_file.txt) to write the output to the file.

The program then enters a loop that reads lines from the input file until there are no more lines to read. Each line of the input file represents a command to be performed on the gatorTaxi. The program parses each command and calls the appropriate method on the gatorTaxi object. The commands that the program can handle are:

i) Insert (rideNumber, rideCost, tripDuration)
ii) Print(rideNumber)
iii) Print (rideNumber1, rideNumber2)
iv) UpdateTrip (rideNumber, newTripDuration)
v) GetNextRide ()
vi) CancelRide(rideNumber)

For each command that modifies the red-black tree, the program calls the appropriate method on the gatorTaxi object. For commands that print information to the output file, the program formats the output as a string and writes it to the output file.

Once the program has finished processing all commands in the input file, it closes both the input and output files and returns 0 to indicate successful completion.