COP5536: Advanced Data Structures

Project-1: GatorTaxi

Venkata Sai Karthik Metlapalli

UFID: 65764476

vmetlapalli@ufl.edu

INTRODUCTION:

GatorTaxi consists of implementations of Red Black Tree and MinHeap data structures from scratch to create a logic for various operations in a ride sharing application. This was developed in C++ language.

DATASTRUCTURES:

The input ride object consists of 3 fields – rideNumber, rideCost and tripDuration. The input commands consist of various commands like Print, Print rides in range, get the ride of minimum cost, remove and update rides. To work on these operations, Min Heap and Red black trees are implemented from scratch. The RBTree class is used to store the rides in a Red black tree sorted by the rideNumber key. Similarly, the MinHeap class is used to store the rides in a min heap having key as the rideCost.

The ride object has been modified to have additional minHeapIndex field, which denotes the index in which this ride is present in the minheap. This is essential inorder to find the location of the ride to be removed incase of a cancelRide(rideNumber) operation in constant time. This makes the time complexity of a random ride removal from min heap in constant time.

FILE STRUCTURE:

There are many classes and files created using OOP paradigm to implement this. The various cpp files and header files are:

1. **gatorTaxi.cpp** – Main entry point for the application. Has logic to read the input from input file, parses it into an object of the IOterm which in turn is stored in the Taxi object.
2. **IOterm.h** – Consists of the actionTerm which is print, insert, update etc., and the list of parameters for each action term.
3. **Taxi.h** – Consists of class fields and method declarations of Taxi class. Fields include list of IOterms read from the input file, an instance of minheap and RBTree.

4. **Taxi.cpp** – Consists of the method definitions of the ones declared in Taxi.h and logic to write the output to the output_file.txt. These methods call the appropriate methods in the redblack tree and min heap based on input.
5. **MinHeap.h** – Consists of class fields and method declarations of MinHeap class. Fields include a vector of Ride objects.
6. **MinHeap.cpp** – Consists of method definitions of all the ones declared in MinHeap.h.
7. **RBTree.h** – Consists of 2 classes defined namely the RBNode and RTree. RBNode has fields for leftChild, rightChild, parent, color of the node and also the Ride value stored. RBTree has a root RBNode and method declarations to perform inserts and deletes from the red black tree.
8. **RBTree.cpp** - Consists of method definitions of all the ones declared in RBTree.h
9. **Ride.h** – Consists of declaration of Ride class that has the fields like rideNumber, rideCost, tripDuration and minHeapIndex.
10. **Ride.cpp** – Contains getters and setters for the above fields of Ride class and also the comparator object.

METHODS:

Taxi class:

```
/*Calls methods to insert the ride in minheap and rbtree. Prints duplicate
rideNumber error onto the output file*/
void insertRide(int rideNumber, int rideCost, int tripDuration, std::ofstream&
outFile);

/*Calls the get(rideNumber) method of rbTree class and writes the output to the
output_file*/
void printRide(int rideNumber, std::ofstream& outFile);

/*Calls the method getRIdeNumbersInRange(rideNumber1, rideNumber2) method of
rbTree class and writes the output to the output_file*/
void printRidesInRange(int rideNumber1, int rideNumber2, std::ofstream& outFile);

/*Update function -> UpdateTrip(rideNumber, new_tripDuration) where the rider
wishes to change the destination
 a) if the Ride =  getByRideNumber in RBT  and if new_tripDuration <= existing
tripDuration,  just update the tripDuration to the new_tripDuration.
b) if the existing_tripDuration < new_tripDuration <= 2 * (existing
tripDuration), then
 increment key(rideNumber, rideCost + 10,     new_tripDuration) in Minheap
c) if the new_tripDuration > 2 * (existing tripDuration), remove from bot*/
void updateTrip(int rideNumber, int newTripDuration);

/*Calls the removeByrideNumber method in rbTree and removeRandomRide in minHeap.
Removes the ride from both minHeap and rbTree*/
void cancelRide(int rideNumber);

/*Calls the removeMin in minHeap and also removes the ride with that reideNumber
from the rbTree*/
void getNextRide(std::ofstream& outFile);

/*Called to read the commands from the input file and store them as a
vector<IOterm* >(actionList) in the Taxi object*/
void readInputFromFile(std::string, std::vector<IOterm*>& actionList);
```

```
/*Called when reading each action from the actionList and to call corresponding
methods from rbTree and minheap*/
void takeAction(std::ofstream& outfile);
```

RBTree class:

```
/*To insert a new ride in RBTree*/
int insertRideInRBTree(Ride* ride);

/*Get rides in the given range of rideNumbers */
std::vector<Ride*> getRidesInRangeFromRBTree(int rideNumber1, int rideNumber2);

//Get by RideNumber
RBNode* get(int rideNumber);

//Remove Ride from RBTree
void removeRideFromRBTree(int rideNumber);
```

These methods include internal calls to insertInBSTree, fixViolations to fix the violations caused by inserts and deletes. They are mentioned below:

UTILITY/ HELPER FUNCTIONS:

```
/*Returns the sibling node of the current node*/
RBNode* RBNode::getSibling();


/*Recursive function to do inorder traversal of subtrees within the range*/
/* append all elements in [rideNumber1, rideNumber2] in the subtree rooted at
root to rangeRides */
void getRangeSubtree(RBNode* RBTreeRoot, int rideNumber1, int rideNumber2,
std::vector<Ride*>& rangeRides);


/*To insert a node in the RBTree., which is a BST. After this, we need to fix the
violations caused to RBTree properties*/
int BSTInsert(RBNode*& RBTreeRoot, RBNode* newNode);

/*Insert / Delete Utility functions. Helps in performing rotations and color
changes*/
RBNode* removeDegree1(RBNode* toremove, RBNode* toraise, RBTree* tree);


/* adjust if p and pp are both red */
void adjustTwoRed(RBNode* p, RBTree* tree);


/* Called as part of the remove operation to adjust if py is the parent of a
deficient node v is the sibling of the deficient node */
void adjustDeficient(RBNode* py, RBNode* v, RBTree* tree);


/* Method to find the rightmost leaf in the left subtree and replace assuming
toreplace has both left and right child return the leaf node */
RBNode* replace(RBNode* toreplace);
```

MinHeap class:

```
/* inserts new node in the MinHeap and this includes the heapify_up to adjust
heap property */
void insert(Ride*);

/*Called incase of update Function*/
void increasekey(Ride* ride,int additionalCost, int newTripDuration);

/* getNextRide performs the Remove min operation - pops topmost Ride Node and
returns it. This also includes a heapify_down step to place the last leaf at root
node and then heapify*/
Ride* getNextRide();

/* Called after an insert method, to check if the newly inserted node has to
bubble up to the root*/
void heapifyUp(int index);

/* Called after remove min operation. The last leaf is placed in root after
deleting the old root(min element).Then heapify is done downwards from root */
void heapifyDown(int index);

/*Used to remove the ride from MinHeap. Called incase of random Remove operation
when performing Cancel Ride(rideNumber)*/
void remove(Ride* toRemove);
```

Ride.class:

```
// return true if r1.cost < r2.cost. If same cost, then the trip duration is used
as a tie breaker
//Used as a means extract min element from the min-heap
bool Ride::compareCostsOfRides(Ride* r1, Ride* r2);
```

DATA FIELDS:

Ride class:

```
// Unique integer identifier for each ride.
int rideNumber;
// The estimated cost(in integer dollars) for the ride.
int rideCost;
// the total time(in integer minutes) needed to get from pickup to destination
int tripDuration;
//Integer field to store the index in Minheap
int minHeapIndex;
```

MinHeap class:

```
//vector field to store the rides
std::vector<Ride*> minHeapRides;
```

RBNode class:

```
// Fields of the RBNode
```

```
//Ride object
Ride* ride;
//Pointers to left child, right child and parent nodes
RBNode* left;
RBNode* right;
RBNode* parent;
//color -> RED, BLACK stored in an enumerated list
int color;
```

RBTree class:

```
// Root Node of the tree
RBNode* RBTreeRoot;
```

IOterm class:

```
//Operation to be performed (Eg print, insert, getNextride etc.,)
int action;
//Parameters of that action
std::vector<int> parameters;

action is one of
enum actions {
	Print,
	Insert,
	GetNextRide,
	CancelRide,
	UpdateTrip
};
```

Taxi class:

```
//List of actions parsed from the input file
std::vector<IOterm*> actionList;
//reference to minHeap
MinHeap* minHeap;
//reference to rbTree
RBTree* rbTree;
```

TIME COMPLEXITY:

All the operations take O(log(n)) complexity, except for the print(rideNumbr1, rideNumbre2) that takes O(log(n)+K) where n is the number of active rides and K is the number of triplets printed.
Here, n = total number of nodes in the red black tree.

1.Print (int rideNumber1, int rideNumber2):
For printing range of buildings, we go into the left child if root value is greater than left boundary and similarly for right child. Hence this adds a complexity of O(k) where k is the number of nodes (rides) in the given range.

2. Print (int rideNumber):
Complexity for search in a Redblack tree is O(log(n)), because it is just like search for a key in a Binary Search Tree (BST).

Complexity for heap insertion, increase Key and extracting min is O(log(n)) as we recursively compare up to the root(heapifyUp) or go down a branch(heapifyDown) respectively.

3. Insert (int rideNumber, int rideCost, int tripDuration):
Complexity for heap insertion is O(log(n)) as we insert as a new leaf and recursively compare up to the root(heapifyUp) which takes log(n) comparisons along that path from leaf to the root.
Similarly, for RBTree insert, it takes O(log(n)) time complexity. Since insert in BST is O(log(n)) and then rotations and actions to fix red black violations also take O(log(n)).

So, insert time complexity is O(log(n))

The complexity for random node remove from min heap is also log(n) as we are already storing the index of the node in heap. Hence, we can access the node in constant time and then heapifyDown in log(n) time.

4. getNextRide():
This is the remove min operation from heap, and we also remove the corresponding ride in the ride black tree.
Complexity for extracting min is O(log(n)) as we swap the root with last leaf and recursively go down a branch(heapifyDown) which takes log(n) comparisons along that path from root to a leaf.

Complexity for red black tree delete is also O(log(n)) This delete involves first a get operation to find the ride and then delete it, then fixing the violations each of which takes O(log(n)) time.

So, removeMin time complexity is O(log(n))

5.cancelRide (int rideNumber):
cancelRide is nothing but remove the node with given key from RBtree. Correspondingly, we also make use of the minheapIndex field to remove the node from min Heap.

Complexity for red black tree delete is O(log(n)) This delete involves first a get operation to find the ride and then delete it, then fixing the violations each of which takes O(log(n)) time.

In constant O(1) time, we get to the node present at minHeapIndex and then perform remove operation from minheap. This remove takes O(log(n)) as we swap the minHeapIndex node with last leaf and recursively go down a branch(heapifyDown) which takes log(n) comparisons along that path from minHeapIndex node to a leaf.

So, cancelRide time complexity is O(log(n))

6. updateTrip(int rideNumber, int newTripDuration):
Update function just performs increment key and remove operations. Hence O(log(n)) time complexity.

SPACE COMPLEXITY:

1.Print (int rideNumber1, int rideNumber2):
O(k), where k is the number of rides in that range of rideNumbers. Since, we need to iterate over the RB tree recursively and store the rides in a vector whose size would be k.

2. Print (int rideNumber):
O(1) time because we just need to print the one ride with the given rideNumber and rideNumbers are unique.

3. Insert (int rideNumber, int rideCost, int tripDuration):
O(1), we just add one more element to the minheap and Redblack tree and just do fixes and rearrangements after adding this node.

4. getNextRide():
O(1) time because we just need to return the one ride with the minimum cost.

5.cancelRide (int rideNumber):
O(1) time because we just need to return the one ride with the given rideNumber and rideNumbers are unique.

6. updateTrip(int rideNumber, int newTripDuration):
O(1) since we just update the cost and tripDuration fields, or perform a remove operation.