Mohammad Uzair Fasih
6282-1020
mfasih@ufl.edu

[COP5536] Advanced Data Structures

Project Assignment Report

Project Structure

```
.
├── GatorTaxi.hpp       // GatorTaxi implementation
├── MinHeap.hpp         // Min-Heap implementation
├── Nodes.hpp           // Nodes for Red-Black tree and Min-Heap
├── RedBlackTree.hpp    // Red-Black tree implementation
├── main.cpp            // Entrypoint for the project
├── makefile            // Makefile for the project
└── Report.pdf          // This file
```

Function Prototypes and Time Complexity

MinHeap.hpp

The MinHeap.hpp header file contains the implementation for a min-heap. The implementation is pointer based where each node consists of a left child pointer, a right child pointer, a parent pointer and an external pointer to point to the node's Red-Black Tree counterpart. Apart from this metadata, the min-heap also stores the Ride information.

The function prototypes for the **MinHeap** class is as follows:

```
class MinHeap {
private:
  MinHeapNode *root = nullptr;

  MinHeapNode *getLastNode();
  MinHeapNode *getNextNode(MinHeapNode *node);
  void heapify(MinHeapNode *node);

public:
  MinHeapNode *insert(Ride ride);
  MinHeapNode *getMin();
  void remove(MinHeapNode *node);
};
```

A description of the operations of the min-heap are as follows:

`MinHeapNode *insert(Ride ride):` Inserts a new ride in the min heap. The time complexity of this operation is **O(log n)**, where n is the number of elements in the heap. This method inserts a new node into the heap, and then performs a bubble-up operation to maintain the heap property. The time complexity of this operation is O(log n), where n is the number of nodes in the heap. This is because in the worst case, the node may need to be bubbled up to the root of the heap, which is at a distance of log n from the node. The space complexity is **O(1)** for best-case scenario (when the tree is empty) and **O(log n)** for the worst-case scenario.

`MinHeapNode *getMin():` Returns the minimum ride while also deleting it from the heap. The time complexity of this operation is **O(log n)**, where n is the number of elements in the heap. This method retrieves the minimum node in the heap, deletes it from the heap, and then performs a heapify operation to maintain the heap property. The time complexity of this operation is O(log n), where n is the number of nodes in the heap. This is because in the worst case, the node that replaces the deleted minimum node may need to be bubbled down to the leaves of the heap, which is at a distance of log n from the node. The space complexity is **O(log n)** for both the best-case and the worst-case scenario.

`void remove(MinHeapNode *node):` Removes the specified node from the heap. The time complexity of this operation is **O(log n)**, where n is the number of elements in the heap. This method removes the specified node from the heap, and then performs a heapify operation to maintain the heap property. The time complexity of this operation is O(log n), where n is the number of nodes in the heap. This is because in the worst case, the node that replaces the removed node may need to be bubbled down to the leaves of the heap, which is at a distance of log n from the node. Note that the time complexity of this operation may be higher if the position of the node to be removed is not known in advance, as finding the node in the heap may take up to O(n) time in the worst case. The space complexity is **O(log n)** for both the best-case and the worst-case scenario.

RedBlackTree.hpp

The RedBlackTree.hpp header file contains the implementation for a red-black tree. The implementation is pointer based where each node consists of a left child pointer, a right child pointer, a parent pointer and an external pointer to point to the node's Red-Black Tree counterpart. Apart from this metadata, the min-heap also stores the Ride information as well as a color property to indicate the color of the current node.

The function prototypes for the **RedBlackTree** class is as follows:

```
class RedBlackTree {
private:
  RBTNode *root = nullptr;
  void leftRotate(RBTNode *node);
  void rightRotate(RBTNode *node);
  void fixInsert(RBTNode *node);
  void fixDelete(RBTNode *node);

public:
  RBTNode *search(int rideNumber);
  RBTNode *insert(Ride ride);
  void remove(RBTNode *node);
  void getRange(int start, int end, std::vector<Ride> &rides);
  void getRange(RBTNode *node, int start, int end, std::vector<Ride> &rides)
};
```

A description of the operations of the red-black tree are as follows:

`RBTNode *search(int rideNumber):` This method searches for a node identified by the rideNumber in the red-black tree. It returns a pointer to the found RBTNode or a null pointer if no node is found. The time complexity of this method is **O(log n)**. The space complexity is **O(1)**.

`RBTNode *insert(Ride ride):` This method inserts a new RBTNode with the given ride into the red-black tree. It returns a pointer to the inserted RBTNode. The time complexity of this method is **O(log n)**. The space complexity is **O(1)** ignoring the space used to create new nodes.

`void remove(RBTNode *node):` This method removes a RBTNode with the given node from the red-black tree. If the node doesn't exist, the method simply returns. The time complexity of this method is **O(log n)**. The space complexity is **O(1)**.

`void getRange(int start, int end, std::vector<Ride> &rides):` This method gets the rides within the range specified by start and end. The rides are added to the rides vector passed by reference. The time complexity of using inorder traversal to get this range is **O(k + log(n))**, where k is the number of nodes in the range. The space complexity is also **O(k + log(n))**.

`void getRange(RBTNode *node, int start, int end, std::vector<Ride> &rides):` This method is a helper method for the previous getRange() method. It performs an inorder traversal of the red-black tree rooted at node and adds the rides within the specified range to the rides vector passed by reference. If the node is null, the method simply returns.

The time complexities of insert, remove, and search functions are O(log n) because the Red-Black Tree maintains a balanced binary search tree property which guarantees the height of the tree is at most log n where n is the number of nodes in the tree.

GatorTaxi.hpp

The GatorTaxi.hpp header file contains the implementation for a GatorTaxi service as per the description of the project. The implementation uses the red-black tree and min-heap implementation listed above.

The function prototypes for the **GatorTaxi** class is as follows:

```
class GatorTaxi {
public:
  void Print(int rideNumber);
  void Print(int rideNumber1, int rideNumber2);
  bool Insert(int rideNumber, int rideCost, int rideDescription);
  void GetNextRide();
  void CancelRide(int rideNumber);
  void UpdateTrip(int rideNumber, int new_tripDuration);
};
```

A brief description of its methods are as follows:

`void Print(int rideNumber):` This method prints the triplet (rideNumber, rideCost, tripDuration) identified by the given rideNumber if it exists in the RedBlackTree data structure, and prints (0,0,0) otherwise. This query is executed in **O(log(n))** time. The space complexity is **O(1)**.

`void Print(int rideNumber1, int rideNumber2):` This method prints all the rides between rideNumber1 and rideNumber2 (inclusive) if they exist in the RedBlackTree data structure, and prints (0,0,0) otherwise. This method is executed in **O(log(n) + k)** time, where k is the number of rides found. The space complexity is also **O(k + log(n))**.

`bool Insert(int rideNumber, int rideCost, int rideDescription):` This method inserts a new ride with the given rideNumber, rideCost, and rideDescription into both the MinHeap and RedBlackTree data structures. If the operation succeeds, it returns false; otherwise, it returns true. This operation is performed in **O(log(n))** time. The space complexity is **O(1)** for best-case scenario (when the tree is empty) and **O(log n)** for the worst-case scenario.

`void GetNextRide():` This method retrieves and removes the ride with the lowest rideCost from the MinHeap data structure. If no ride exists, it prints "No active ride requests". This operation takes **O(log(n))** time. The space complexity is **O(1).**

`void CancelRide(int rideNumber):` This method removes the ride with the given rideNumber from both the MinHeap and RedBlackTree data structures if it exists. If it does not exist, it does nothing. This operation works in **O(log(n))** time. The space complexity is **O(log n).**

`void UpdateTrip(int rideNumber, int new_tripDuration):` This method updates the tripDuration of the ride with the given rideNumber to the new value new_tripDuration if it exists in the RedBlackTree data structure. This operation works in **O(log(n))** time. The space complexity is **O(log n).**