COP 5536
Project Assignment Report

Name: Ronit Singh || UFID: 4755-7797 || UF-Email: ronit.singh@ufl.edu

Usage

Dependencies

- GCC $\geqslant$ 9.3.0

Compile

- `$ make`

Execute

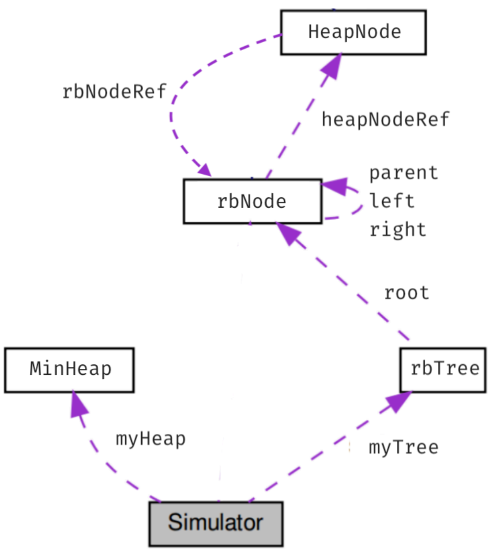
- `$ ./gatorTaxi <inputfile>`

  <inputfile>: path to input file or input file name

Overall Design

Architectural overview

The architectural overview of the implementation is illustrated via the following class diagram.



There are 4 main classes in this program:

- `HeapNode` : this class represents the node in the min heap. It implements some of the necessary methods for the heap structure.
- `rbNode` : this class represents the node in the red-black tree. It stores some additional attributes like color, parent, left child, and right child. It also stores a pointer to the corresponding node of the min heap. Some of the necessary methods for the red- black tree are also implemented.
- `MinHeap` : this class implements the min heap structure using `HeapNode` as its node. The heap is implemented as array-based.

- `rbTree` : this class implements the red-black tree using `rbNode` as its node. All the operations are implemented in this class.

⟹ Additionally, there is a main.cpp (Simulator) which acts as a simulator and reads input from a file. It has a min heap and red-black tree, in which the commands read from the input files are executed and the required output is written into the output file.

Program Structure

The source code contains 4 classes and a main file which provides the required abstraction.

Class: heapNode

```
class heapNode {
    private:
        int rideNumber,
          rideCost, tripDuration; // Data values held by the node.
        rbNode *rbNodeRef; // A pointer to the red-black tree node
    public:
        int pos = 0; // position of this heap node in heap array.
};
```

⟹ Methods Overview of this class:

- `bool operator<(const heapNode &other) const;`
  : Less-than operator overload for heapNode class.
- `rbNode *getrbNodeRef() const;`
  : Getter for heap node reference.
- `void setrbNodeRef(rbNode *newHeapNodeRef);`
  : Setter for heap node reference.
- `friend std::ostream &operator<<(std::ostream &os, const heapNode &node);`
  : Overloaded output operator to print out the heap node.

Class: minHeap

```
class minHeap {
  // the underlying vector that stores the elements of the heap
  std::vector<heapNode> heap;
  // the current size of the heap (initialized to 1 because the root of the heap is initially empty)
  int size = 1;
};
```

⟹ Methods Overview of this class:

- `bool isEmpty();`
  : check if the heap is empty.

- `int getParent(int index);`
  : get the index of the parent of a given node
- `int getLeftChild(int index);`
  : get the index of the left child of a given node
- `bool isValidIndex(int index);`
  : check if a given index is a valid index in the heap
- `void swap(int index1, int index2);`
  : swap the elements at two given indices in the heap
- `void heapifyUp(int position);`
  : perform the "heapify up" operation at a given position in the heap
- `void heapifyDown(int position);`
  : perform the "heapify down" operation at a given position in the heap
- `int getRightChild(int index);`
  : get the index of the right child of a given node
- `void insert(heapNode node);`
  : insert a new element into the heap
- `heapNode removeMin();`
  : remove and return the minimum element from the heap
- `void remove(int index);`
  : remove the element at a given index from the heap

Class: rbNode

```
class rbNode {
private:
  rbNode *left, *right, *parent; // Pointers to the left child, right child, and parent of the node.
  heapNode *heapNodeRef; // Pointer to the corresponding node in the heap.
  nodeColor color;       // Color of the node.
public:
  // Data values held by the node.
  int rideNumber, rideCost, tripDuration;
};
```

⟹ Methods Overview of this class:

- `nodeColor getColor() const;`
  : Getter for the colour of red-black node.
- `void setColor(nodeColor newColor);`
  : Setter for the colour of red-black node

- `heapNode *setHeapNodeRef() const;`
  : Setter for the pointer to the heap node for a specific red-black node.
- `heapNode *getHeapNodeRef() const;`
  : Getter for the pointer to the heap node for a specific red-black node.
- `void setParent(rbNode *newParent);`
  : Setter for the parent child pointer of red-black node.
- `rbNode *getLeft() const;`
  : Getter for the left child pointer of red-black node.
- `void setLeft(rbNode *newLeft);`
  : Setter for the left child pointer of red-black node.
- `rbNode *getRight() const;`
  : Getter for the right child pointer of red-black node.
- `void setRight(rbNode *newRight);`
  : Setter for the right child pointer of red-black node.
- `friend std::ostream &operator<<(std::ostream &os, const rbNode &node);`
  : Overloaded output operator to print out the node.

Class: rbTree

```
class rbTree {
private:
  rbNode *root, *nil;  // Pointer to root of tree and a nil pointer
which represent the black empty nodes of red-black tree
};
```

$\Longrightarrow$ Methods Overview of this class:

- `bool isLeftChild(rbNode *node);`
  : Checks if the node is a left child of its parent.
- `bool isRightChild(rbNode *node);`
  : Checks if the node is a right child of its parent.
- `void UpdateParentChildLink(rbNode *parent, rbNode *oldChild,rbNode *newChild);`
  : Setter for the pointer to the heap node for a specific red-black node.
- `void rotateLeft(rbNode *node);`
  : Performs a left rotation on the given node.
- `void rotateRight(rbNode *node);`
  : Performs a right rotation on the given node.

- `rbNode *getMinimumNode(rbNode *node);`
  : Returns the node with the minimum ride number in the subtree rooted at node.
- `void insertionRebalance(rbNode *node);`
  : Rebalances the tree after inserting a new node.
- `void DeletionRebalance(rbNode *node);`
  : Rebalances the tree after deleting a node.
- `rbNode *searchRecursive(rbNode *root, int rideNumber);`
  : Searches for a node with the given ride number recursively starting from the given root node.
- `void searchInRangeRecursive(rbNode *root, int rideNumber1, int rideNumber2, std::vector<rbNode> &vec);`
  : Searches for all nodes with ride numbers in the given range recursively starting from the given root node.
- `void insert(rbNode *node);`
  : Inserts the given node into the tree.
- `void deleteNode(rbNode *node);`
  : Deletes the given node from the tree.
- `rbNode *search(int rideNumber);`
  : Searches for a node with the given ride number in the tree.
- `std::vector<rbNode> searchInRange(int rideNumber1, int rideNumber2);`
  : Searches for all nodes with ride numbers in the given range.

$\Longrightarrow$ The main.cpp file reads input and writes output. The following is overview of the function implemented in this file.

- `void Insert(int rideNumber, int rideCost, int tripDuration, std::ofstream &out);`
  : Inserts the ride information into both the red black tree and minheap.
- `void GetNextRide(std::ofstream &out);`
  : Retrieves the next ride from a heap data structure, deletes it from the red black tree as well as the heap, and writes it to an output file stream object.
- `void Print(int rideNumber, std::ofstream &out);`
  : Prints the details of the ride with given rideNumber.

- `void Print(int rideNumber1, int rideNumer2, std::ofstream &out);`
  : Prints the details of all rides with ride numbers in the given range.

- `void CancelRide(int rideNumber);`
  : Cancels the ride with given ride number. Removes the ride from the red-black tree and the heap.

- `void UpdateTrip(int rideNumber, int newTripDuration);`
  : Updates the trip duration of a ride if the new duration is not more than twice the current duration. Otherwise removes it from both data structure.

- `std::vector<std::string> process_string(std::string s);`
  : A utility function to separate information from given string. Essentially to process data from the input file.

Complexity Analysis

The needed operations were:

1. Print(rideNumber)
   It searches through the red-black tree for rideNumber which gives us complexity of O(log n), where n is the number of nodes in tree.
2. Print(rideNumber1, rideNumber2)
   It searches through the red-black tree for rideNumber within range in binary search fashion which gives us the complexity of O(log n + S), where n is the number of nodes in tree and S is the number of triplets
3. Insert (rideNumber, rideCost, tripDuration)
   Inserting into the red-black tree & min heap would take O(log n).
4. GetNextRide()
   To get next ride we remove min from min heap which has complexity of O(log n) and then to delete that node from red-black tree would take O(log n).
5. CancelRide(rideNumber)
   Deletion from both the red-black tree and min heap would take O(log n) as from min heap we delete the node in O(log n) and from this node we get pointer to node in min heap and then we delete that node in min heap in log n.
6. UpdateTrip(rideNumber, new_tripDuration)
   Searching and deleting node from red-black tree & min heap takes O(log n) as explained in CancelRide() and then adding another node according to condition will be also in O(log n) for both structures. So, overall O(log n).