# COP 5536 Spring 2023

### COP5536 – Advanced Data Structures

## Project Report

Venkata Satya Abhiram Palika
(UFID: 9956-1573)
palikav@ufl.edu

### Implementation

- Language – Python
- Usage - Min Heap, Red-Black Tree
- Concept applied – pointers

### Dependencies

- Python >= 2

### Execution

```
○ abhiram@Abhirams-MacBook-Air Gator Taxi % python3 gatorTaxi.py abhiram_input.txt
```

### Required Operations

```
1.  Print(rideNumber) – (rideNumber, rideCost, tripDuration)

2.  Print (rideNumber1, rideNumber2) – prints all triplets (rx,
rideCost, tripDuration)

3.  Insert (rideNumber, rideCost, tripDuration)

4.  GetNextRide() –
When this function is invoked, the ride with the lowest rideCost
structure.

5.  CancelRide(rideNumber) – structures, can be ignored if an entry for
rideNumber doesn't exist.

6.  UpdateTrip(rideNumber, new_tripDuration)
```

## File structure

- **gatorTaxi.py –**
  This code defines several functions for managing ride requests in a taxi service. The Ride class defines the properties of a ride and has a method for comparing two rides. Here red-black tree and min heap are imported and used.

```python
import sys
from minHeap import MinHeap
from minHeap import MinHeapNode
from rbTree import RBTree
from rbTree import RBTreeNode

class Ride:
    def __init__(self, ride_number, rideCost, tripDuration):
        #  ride_number: unique number for each ride
        self.ride_number = ride_number
        self.rideCost = rideCost
        self.tripDuration = tripDuration

    def is_less_than(self, other_ride):
        # compare two rides
        if self.rideCost < other_ride.rideCost:
            return True
        elif self.rideCost == other_ride.rideCost:
            if self.tripDuration > other_ride.tripDuration:
                return False
            else:
                return True
        elif self.rideCost > other_ride.rideCost:
            return False

def cancel(ride_number, heap, rbtree):
    # remove ride from heap and rbtree
    heap_node = rbtree.remove(ride_number)
    if heap_node is not None:
        heap.remove_element(heap_node.min_heap_index)
```

- **minheap.py –**
  The code implements a binary min heap data structure to keep track of rides sorted by their trip duration. The heap elements are objects of the MinHeapNode class. It contains the ride object, object of the red-black tree and the position of node (index).

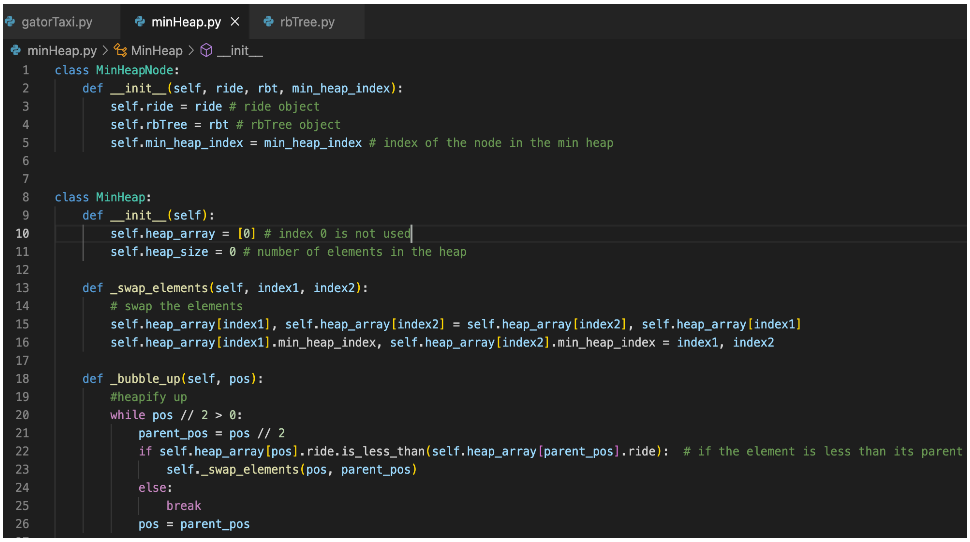
  Time Complexity –
  The time complexity of the complete code is O(log n) for each operation because it involves at most heapify up or down operation.

  i. For insertion, it is O(log n) because of the height of min heap is log n (In worst case, it takes log n swaps).

  ii. For deletion, it is O(log n) because replace, heapify operations on root and the height of binary heap (worst case – bottom element).

  iii. For Search, it is O(log n) because of the operation on the array (linear).

Space Complexity –
The space complexity of the complete code is O(n) because it contains an array of n elements (n nodes).


gatorTaxi.py   minHeap.py   rbTree.py
minHeap.py > MinHeap > __init__

```
class MinHeapNode:
    def __init__(self, ride, rbt, min_heap_index):
        self.ride = ride # ride object
        self.rbTree = rbt # rbTree object
        self.min_heap_index = min_heap_index # index of the node in the min heap


class MinHeap:
    def __init__(self):
        self.heap_array = [0] # index 0 is not used
        self.heap_size = 0 # number of elements in the heap

    def _swap_elements(self, index1, index2):
        # swap the elements
        self.heap_array[index1], self.heap_array[index2] = self.heap_array[index2], self.heap_array[index1]
        self.heap_array[index1].min_heap_index, self.heap_array[index2].min_heap_index = index1, index2

    def _bubble_up(self, pos):
        #heapify up
        while pos // 2 > 0:
            parent_pos = pos // 2
            if self.heap_array[pos].ride.is_less_than(self.heap_array[parent_pos].ride):  # if the element is less than its parent
                self._swap_elements(pos, parent_pos)
            else:
                break
            pos = parent_pos
```

- **rbTree.py –**
In this code, a Red-Black Tree is implemented with Attributes: color, left child, right child, parent. The code provides the class RBTreeNode that defines a node of the Red-Black Tree. In order to preserve the Red-Black Tree qualities, it offers methods for finding, adding, and deleting nodes from the tree, as well as rotation and rebalancing techniques.

Time complexity –
The time complexity of the complete code is O(log n) for all operations except rebalancing. The rebalancing operation has a worst-case time complexity of O(log n).

i. For insertion, it is O(log n) because of the height of the tree. For color it is O(1) for each node.

ii. For deletion, it is O(log n) because every rotation takes same amount of time and it is proportional to the height.

iii. For Search, it is O(log n) because it is similar to binary tree.

Space complexity –
The space complexity of the complete code is O(n) because the code uses a RBTreeNode class to represent each node in the tree, and each node stores a ride object and a min heap node object.

gatorTaxi.py minHeap.py rbTree.py

rbTree.py > ...

```python
L = [0, 1]
# L[1] = red , L[0] = black

class RBTreeNode:
    def __init__(self, ride, min_heap_node):
        self.ride = ride # ride object
        self.parent = n  # parent node
        self.left = n  # left node
        self.right = n  # right node
        self.color = L[1]
        self.min_heap_node = min_heap_node # min heap node

n = None
class RBTree:
    def __init__(self):
        # initialize the tree
        self.nil = RBTreeNode(None, None) # nil node
        self.size = 0
        self.nil.left = n
        self.nil.right = n
        self.tree = n
        self.nil.color = L[0]
        self.root = self.nil

    def _find_min(self, n):
        # find the minimum node
        if n.left == self.nil:
            return n
        else:
            return self._find_min(n.left)
```

## List of Methods

### 1. gatorTaxi.py –

- is_less_than (self, other_ride) –
  It compares two rides based on the rideCost and tripDuration.

| |
|---|
| Arguments - ride cost, trip duration |

- cancel (ride_number, heap, rbtree) –
  It removes a ride from the heap and red-black tree.

| |
|---|
| Arguments - ride number, min heap, and red-black tree. |

- insert (ride, heap, rbtree) –
  Insertion of ride into red-black tree and min heap

| |
|---|
| Arguments - ride object, min heap and red-black tree. |

- update (ride_number, new_duration, heap, rbtree) –
  Role of it is - Updating of ride in red-black tree and min heap based on its latest ride duration.

| Arguments - new duration, in red-black tree and min heap |
|---|

- add_out (ride, msg, list) –
  It writes output to the output file. Here a Boolean flag is used indicating whether the ride is a list or not.

| Arguments – ride object, a message, and a boolean flag. |
|---|

- nxt_ride(heap, rbtree):
  A function that gets the next ride from the heap and red-black tree.

| Arguments – heap, red-black tree |
|---|

- ride_out(ride_number, rbtree): A function that gets a ride from the red-black tree based on its ride number.

| Arguments – ride number, red-black tree |
|---|

- rides_out(l, h, rbtree): A function that gets rides from the red-black tree based on their cost range.

| Arguments – lower and upper bounds of cost range, red-black tree |
|---|

- parse_numbers(s): It parses a string containing numbers and returns them as a list.

**2. minheap.py –**

- swap_elements(self, index1, index2): Swaps the elements at positions index1 and index2 in the heap array, and updates the min heap indices of the swapped nodes.

| Arguments – Index of the 2 elements (index1, index2) |
|---|

- _bubble_up(self, pos): Heapify up the element at position pos in the heap array to its correct position to maintain the heap property.

| Arguments – Position of the element in the heap array (pos) |
|---|

- _find_min_child(self, pos): It finds the location of the element's min child (index) at position in the min heap.

| Arguments – Position of the element in the heap array (pos) |
|---|

- _bubble_down(self, pos): Heapify down the element at position pos in the heap array to its correct position to maintain the heap property.

| Arguments – position of the element in the heap array (pos) |
|---|

- modify_element(self, pos, updated_key): Modifies the trip duration of the ride at position pos in the heap array to updated_key, and bubbles up or down the node to maintain the heap property depending on whether the updated key is greater or less than its parent.

| Arguments – Position of the element in the heap array (pos), updated_key |
|---|

- add (self, element): Adds a new MinHeapNode object to the heap array, increments the heap size, and heapify up the new node to its correct position.

| Arguments – New object (element) |
|---|

- extract_min(self): Removes and returns the minimum element from the heap array, which is always the element at position 1, and bubbles down the last element to its correct position.

- remove_element(self, pos): Removes the element at position pos from the heap array, and bubbles down the last element to its correct position.

| Arguments – Position of the element in the heap array (pos) |
|---|

3. **rbTree.py –**

- _find_min(self, n): Recursively finds the minimum node starting from the given node n and returns it.

| Arguments – Given node (n) |
|---|

- _left_rotate(self, x): Performs a left rotation on the node x and its right child y. Updates the parent of y and x, and x's right child.

| Arguments – Node(x) |
|---|

- _update_parent(self, x, y): The function sets the parent of node y to the parent of node x. If node x is the root node, then the function updates the root to be node y. In case node x is the left child of its parent, the function updates the left child of its parent to be node y. If node x is the right child of its parent, the function updates the right child of its parent to be node y.

| Arguments – x, y (elements) |
|---|

- _right_rotate(self, x): Performs a right rotation on the node x and its left child y. Updates the parent of y and x, and x's left child.

| Arguments – Node (x) |
|---|

- remove_node(self, item, key): Removes the node with given key from the tree rooted at item. Returns the min_heap_node attribute of the removed node. If the removed node has no left or right child, replaces it with its right or left child. If it has both left and right child, replaces it with its successor y, which has no left child, and moves y's right child to y's position in the tree. Then, rebalances the tree as needed.

| Arguments – item, key |
|---|

- search_key(self, key): Searches for the node with the given key in the tree and returns it. If it is not found, returns None.

| Arguments – key |
|---|

- _rb_replace(self, item, child_item): Replaces item with child_item in the tree. Updates the parent of child_item.

| Arguments – item, child_item |
|---|

- remove (self, ride_number): Removes the the node of min_heap attribute of the removed node by calling remove_node method with root node and given ride_number.

| Arguments – ride_number |
|---|

- locate_rides_in_range(self, n, low, high, result): Recursive function that locates all the rides in a given range using the binary search tree property.

| Arguments – Node (n), lower, higher bound of the given range, result |
|---|

- _rebalance_post_insert(self, node): Balances the tree after inserting a node by performing rotations and recoloring nodes.

| Arguments – node |
|---|

- add (self, ride, min_heap): Adds a node with the ride object and min heap node to the tree, then rebalances the tree using _rebalance_post_insert.

| Arguments – ride object, min_heap |
|---|

- rides_in_range(self, low, high): Using _locate_rides_in_range, it gives a list of all the rides with ride numbers between [low, high].

| Arguments – lower, higher bounds of the ride numbers. |
|---|