



UNIVERSITY OF FLORIDA

COP 5536

Advanced-Data Structures

Spring 2023

Project Report

Submitted by: Ria Mahajan

UFID: 2323-4247

Mail: ria.mahajan@ufl.edu

Table of Contents

1. Introduction

- **Problem Statement**
- **Required Operations**
- **Implementation Constraints**
- **Input and Output Requirements**
 - **Input Format**
 - **Output Format**

2. Solution

- **Overview**
- **Structure**
- **Running the Project Code**
- **Source Code Breakdown**
 - **Libraries**
 - **Min Heap Class**
 - **Node Class**
 - **Red-Black Tree**
 - **GatorTaxi Class**
 - **Input/Output Handler (main) Function**

3. Code Testing

4. Time Complexity Analysis

5. Conclusion

1. Introduction

Problem Statement

GatorTaxi is an emerging ride-sharing platform that receives numerous ride requests daily. To better manage their growing volume of pending ride requests, they are in the process of creating a new software solution.

A ride can be distinguished by a combination of three attributes:

- **rideNumber:** a distinct integer identifier assigned to every individual ride.
- **rideCost:** the projected expense (in whole dollars) associated with the ride.
- **tripDuration:** the entire duration (in whole minutes) required to travel from the pickup point to the final destination.

Required Operations

The required operations include:

- **Print(rideNumber)** displays the triplet (rideNumber, rideCost, tripDuration) corresponding to the specified ride.
- **Print(rideNumber1, rideNumber2)** shows all triplets (r_x , rideCost, tripDuration) where $\text{rideNumber1} \leq r_x \leq \text{rideNumber2}$.
- **Insert(rideNumber, rideCost, tripDuration)** adds a new entry with a unique rideNumber, not matching any existing ride numbers.
- **GetNextRide()** returns and removes the ride with the lowest rideCost from the data structure, using tripDuration as a tiebreaker if necessary.
- **CancelRide(rideNumber)** eliminates the triplet (rideNumber, rideCost, tripDuration) from the data structure, disregarding non-existent rideNumber entries.

- **UpdateTrip(rideNumber, new_tripDuration)** allows riders to change their destination.

In this case:

- If $\text{new_tripDuration} \leq \text{existing_tripDuration}$, no action is needed.
- If $\text{existing_tripDuration} < \text{new_tripDuration} \leq 2 * (\text{existing_tripDuration})$, the driver cancels the existing ride, and a new ride request is created with a penalty of 10 on the existing rideCost. The data structure is updated with $(\text{rideNumber}, \text{rideCost} + 10, \text{new_tripDuration})$.
- If $\text{new_tripDuration} > 2 * (\text{existing_tripDuration})$, the ride is automatically rejected, and the entry is removed from the data structure.

Implementation Constraints

To accomplish the specified task, the project requires the utilization of a min-heap and a Red-Black Tree (RBT). The implementation of both the min-heap and RBT must be done using custom code. Additionally, it can be assumed that the total number of active rides will not surpass 2,000.

Avoid utilizing advanced data structures offered by programming languages. Instead, you are required to implement the min-heap and Red-Black Tree (RBT) data structures from scratch using fundamental components, such as pointers. Refrain from employing any Map-related libraries.

A min-heap should be employed to store $(\text{rideNumber}, \text{rideCost}, \text{tripDuration})$ triplets, sorted by rideCost. In cases where multiple triplets share the same rideCost, those with the shortest tripDuration will be given higher priority, considering that all rideCost-tripDuration combinations are unique. A Red-Black Tree (RBT) should be used to store $(\text{rideNumber}, \text{rideCost}, \text{tripDuration})$ triplets, sorted by rideNumber. Maintaining pointers between corresponding nodes in the min-heap and RBT is essential.

GatorTaxi is capable of handling only a single ride at any given moment. When it's time to choose a new ride request, the ride with the lowest rideCost (with tripDuration as a tiebreaker) is

selected, corresponding to the root node in the min-heap. If no rides are available, a message stating "No active ride requests" should be returned.

Input and Output Requirements

The program should be executed using the following commands depending on the programming language:

For Python:

```
$ python gatorTaxi.py file_name
```

In these commands, **file_name** refers to the name of the file containing the input test data.

Input Format

The input test data will be provided in the following structure:

- Insert(rideNumber, rideCost, tripDuration)
- Print(rideNumber)
- Print(rideNumber1, rideNumber2)
- UpdateTrip(rideNumber, newTripDuration)
- GetNextRide()
- CancelRide(rideNumber)

This format represents the various operations and their corresponding parameters that will be used throughout the program.

Output Format

- **Insert(rideNumber, rideCost, total_time)** should not generate any output. However, if rideNumber is a duplicate, an error should be displayed and the program should stop.
- **Print(rideNumber)** will output the (rideNumber, rideCost, tripDuration) triplet if the rideNumber exists; otherwise, it should print (0, 0, 0).
- **Print(rideNumber1, rideNumber2)** will display all (rideNumber, rideCost, tripDuration) triplets within the range of rideNumber1 and rideNumber2, inclusive, in a single line separated by commas. If there are no rides in the specified range, the output should be (0, 0, 0). No additional comma should be printed at the end of the line. Other output includes a string message indicating no rides are available.
- **UpdateTrip(rideNumber, newTripDuration)** will not generate any output.
- **CancelRide(rideNumber)** will not produce any output.
- **GetNextRide()** will display the ride with the lowest rideCost, prioritizing the one with the shortest tripDuration in cases of multiple triplets with the same rideCost.
- All output should be directed to a file named "**output_file.txt**".

2. Solution

Overview

GatorTaxi service is designed to offer optimal ride management by employing two key data structures, Min Heap and Red-Black Tree. The choice of these data structures enables the system to quickly find the next ride to process, maintain an ordered set of rides, and perform search and range queries efficiently.

Min Heap: The Min Heap is a crucial component of the GatorTaxi service, as it ensures the efficient management of ride requests. This binary tree data structure has the property that each parent node has a value less than or equal to its children, allowing the system to swiftly locate the minimum element. In the context of GatorTaxi, this minimum element represents the ride with the lowest cost and trip duration. The Min Heap's efficiency in insertion, deletion, and updates makes it ideal for handling ride requests and determining the next ride to process.

Red-Black Tree: The Red-Black Tree plays a vital role in the GatorTaxi service by enabling fast search and range query operations. This self-balancing binary search tree maintains a roughly balanced structure during insertions, deletions, and updates, ensuring that the tree height remains logarithmic. Consequently, the time complexity for operations such as search, insertion, and deletion is optimized. In the GatorTaxi service, the Red-Black Tree is utilized to store rides with unique ride numbers.

Overall, the GatorTaxi service relies on the combined strengths of the Min Heap and Red-Black Tree data structures to efficiently manage ride requests. By using a Min Heap to quickly locate the next ride to process and a Red-Black Tree to maintain an ordered set of rides, the system ensures optimal ride management and processing of various commands. This innovative approach to ride management demonstrates the importance of selecting appropriate data structures in the design of efficient systems.

Structure

The project implementation consists of 2 files:

- `gatorTaxi.py` - This source code file encompasses all necessary implementations for the Min Heap, Red-Black Tree, and the primary `GatorTaxi` class.
- `Input.txt` - An example input file is provided, showcasing the prescribed input format.

Running the Project Code

First, open the terminal and navigate to the directory containing both `gatorTaxi.py` and `input.txt`. Then, execute the command, replacing "`file_name`" with the actual name of your input file. A new file named `output_file.txt` will be created in the same directory, containing all the resulting triplets in the required format.



```
1 python gatorTaxi.py file_name
```


Breakdown

Source Code can be broken down into the following major components:

- **Libraries**
- **Min Heap Class**
- **Node Class**
- **Red-Black Tree**
- **GatorTaxi Class**
- **Input/Output Handler (main) Function**

Libraries

The import sys statement at the beginning of the code is used to import the Python modules:

1. The **deque** class is a built-in collection in Python that provides a double-ended queue data structure. It allows for efficient appending and popping of elements from either end of the queue.
2. The **sys** module provides access to some variables used or maintained by the Python interpreter and to functions that interact strongly with the interpreter. It is typically used to access command-line arguments, the standard input/output streams, and the exit() function.
3. The **re-module** provides regular expression matching operations. It allows for searching and manipulating strings based on patterns, making it useful for text processing and parsing.

Min-Heap Class

```
def __init__(self) -> None:
    pass

def insert(self, rideNumber: int, rideCost: int, tripDuration: int, rbnode: int) -> None:
    pass

def update(self, updatenode: int, newtripDuration: int) -> None:
    pass

def cancel(self, currentNode: int) -> None:
    pass

def upHeapify(self, i: int) -> None:
    pass

def GetNextRide(self) -> Union[List[int], None]:
    pass

def downHeapify(self, i: int) -> None:
    pass
```

Function explanations and time complexity:

- **__init__(self)** -> None: Initializes the min-heap with an empty list for the heap, a dictionary for the index mapping, and a size of 0. Time complexity: $O(1)$
- **insert(self, rideNumber, rideCost, tripDuration, rbnode)** -> None: Inserts a new ride into the heap and maintains the heap property. Time complexity: $O(\log n)$, where n is the number of elements in the heap.

- **update(self, updatenode, newtripDuration)** -> None: Updates the trip duration of an existing ride, adjusts the ride cost if necessary, and maintains the heap property. Time complexity: $O(\log n)$, where n is the number of elements in the heap.
- **cancel(self, currentNode)** -> None: Cancels a ride by removing it from the heap and maintaining the heap property. Time complexity: $O(\log n)$, where n is the number of elements in the heap.
- **upHeapify(self, i)** -> None: Moves a ride up the heap to maintain the heap property. Time complexity: $O(\log n)$, where n is the number of elements in the heap.
- **GetNextRide(self)** -> Union[List[int], None]: Retrieves the next ride (with minimum cost and trip duration) from the heap and maintains the heap property. Time complexity: $O(\log n)$, where n is the number of elements in the heap.
- **downHeapify(self, i)** -> None: Moves a ride down the heap to maintain the heap property. Time complexity: $O(\log n)$, where n is the number of elements in the heap.

Node Class

```
def __init__(self, rideNumber: int, rideCost: float, tripDuration: float, parent=None, color=1, left=None, right=None) -> None:
    pass

def __repr__(self) -> str:
    pass
```

Function explanations and time complexity:

1. **__init__(self, rideNumber, rideCost, tripDuration, parent=None, color=1, left=None, right=None) -> None:** Initializes a Node object with the specified ride number, cost, and trip duration, along with parent, color, left child, and right child nodes. Time complexity: $O(1)$
2. **__repr__(self) -> str:** Returns a string representation of the color of the Node object. Time complexity: $O(1)$

Red-Black Tree Class

```
from typing import List, Tuple, Optional

class RedBlackTree:
    def __init__(self) -> None:
        # ...

    def insert(self, rideNumber: int, rideCost: float, tripDuration: float) -> None:
        # ...

    def insert_fix(self, new_node: TreeNode) -> None:
        # ...

    def rotate_left(self, currNode: TreeNode) -> None:
        # ...

    def rotate_right(self, currNode: TreeNode) -> None:
        # ...

    def delete(self, currentNode: TreeNode) -> None:
        # ...

    def delete_fix(self, fix: TreeNode) -> None:
        # ...

    def replace(self, u: TreeNode, v: TreeNode) -> None:
        # ...

    def minimum(self, target: TreeNode) -> TreeNode:
        # ...

    def search(self, rideNumBerval: int) -> Optional[TreeNode]:
        # ...

    def Print(self, start_ride: int, end_ride: int) -> List[Tuple[int, float, float]]:
        # ...

    def collect_rides(self, rides_list: List[Tuple[int, float, float]], node: TreeNode, start_ride: int, end_ride: int) -> None:
        # ...

    def update(self, node: TreeNode, new_duration: float) -> None:
        # ...

    def print_tree(self, print_color: bool = True) -> None:
        # ...
```

Function explanations and time complexity:

1. **__init__(self)** -> None: Constructor to initialize the tree. It creates an empty tree with a sentinel NIL node. Time complexity: $O(1)$.

2. **insert(self, rideNumber: int, rideCost: float, tripDuration: float) -> None:** Inserts a new node with the given ride number, ride cost, and trip duration into the tree, maintaining the red-black tree properties. Time complexity: $O(\log n)$.
3. **insert_fix(self, new_node: TreeNode) -> None:** Fixes the tree after insertion to ensure it follows the red-black tree properties. Time complexity: $O(\log n)$.
4. **rotate_left(self, currNode: TreeNode) -> None:** Performs a left rotation on the given node. Time complexity: $O(1)$.
5. **rotate_right(self, currNode: TreeNode) -> None:** Performs a right rotation on the given node. Time complexity: $O(1)$.
6. **delete(self, currentNode: TreeNode) -> None:** Deletes the given node from the tree and maintains the red-black tree properties. Time complexity: $O(\log n)$.
7. **delete_fix(self, fix: TreeNode) -> None:** Fixes the tree after deletion to ensure it follows the red-black tree properties. Time complexity: $O(\log n)$.
8. **replace(self, u: TreeNode, v: TreeNode) -> None:** Replaces the node u with the node v. Time complexity: $O(1)$.
9. **minimum(self, target: TreeNode) -> TreeNode:** Finds the minimum node in the subtree rooted at the target node. Time complexity: $O(\log n)$.
10. **search(self, rideNumber: int) -> Optional[TreeNode]:** Searches for a node with the given ride number in the tree. Time complexity: $O(\log n)$.
11. **Print(self, start_ride: int, end_ride: int) -> List[Tuple[int, float, float]]:** Returns a list of rides with ride numbers in the given range. Time complexity: $O(m + \log n)$, where m is the number of rides in the output list.
12. **collect_rides(self, rides_list: List[Tuple[int, float, float]], node: TreeNode, start_ride: int, end_ride: int) -> None:** A helper function for the Print method, traverses the tree in an in-order manner and appends the rides in the given range to the rides_list. Time complexity: $O(m + \log n)$, where m is the number of rides in the output list.
13. **update(self, node: TreeNode, new_duration: float) -> None:** Updates the trip duration of a given node in the tree. Time complexity: $O(1)$.
14. **print_tree(self, print_color: bool = True) -> None:** Prints the tree in a readable format, optionally including the color of each node. Time complexity: $O(n)$, where n is the number of nodes in the tree.

GatorTaxi Class

```
class Gatortaxi:
    def __init__(self):
        self.rbt = RedBlackTree()
        self.heap = minHeap()

    def run(self, input_file: str, output_file: str) -> None:
        """
        Runs the Gatortaxi program with the given input file and writes the output to the given output file.
```

Function explanations and time complexity:

- **__init__(self):** Constructor that initializes an empty Red-Black Tree (self.rbt) and a Min Heap (self.heap).
- **run(self, input_file, output_file):** This method reads from an input file and writes the output to an output file. It processes various ride operations such as insertion, updating, cancellation, and retrieval of the next ride

Here's a brief explanation of how the run method works:


- It reads the input file line by line and identifies the ride operation mentioned in each line.
- For each operation, it extracts the arguments needed for the operation and performs the corresponding action using the Red-Black Tree and Min Heap instances.
- The output of each operation is written to the output file.

Ride operations:

1. **Insert:** Inserts a new ride request into the Red-Black Tree and Min Heap. If the ride number is already in the Red-Black Tree, it writes "Duplicate RideNumber" to the output file.
2. **GetNextRide:** Retrieves the next ride with the lowest cost and shortest duration from the Min Heap. It then deletes the corresponding node from the Red-Black Tree and writes the ride information to the output file.
3. **UpdateTrip:** Updates the trip duration of a ride request. It searches for the ride in the Red-Black Tree, updates the duration in the Min Heap and Red-Black Tree, and writes the updated ride information to the output file.
4. **CancelRide:** Cancels a ride request. It searches for the ride in the Red-Black Tree, deletes the corresponding node from the Min Heap and Red-Black Tree, and writes the cancellation to the output file.
5. **Print:** Prints the ride information of one or multiple rides. If given one ride number, it searches for the ride in the Red-Black Tree and writes the information to the output file. If given a range, it writes the information of all rides within the range to the output file. If no ride is found, it writes "(0, 0, 0)" to the output file.

The overall time complexity of `process_input` depends on the number of commands being executed and the type of commands.

Input/Output Handler (main) Function

A code block with a light blue background and a white rounded rectangle containing Python code. The code is color-coded: 'def', 'if', 'print', 'sys.exit', 'input_file', 'output_file', 'Gatortaxi', and 'run' are in blue; 'len', 'sys.argv', '1', '2', 'input.txt', and 'output.txt' are in green; and '1' and '2' are in red. The code defines a 'main' function that checks for command-line arguments, prints a usage message if there are not exactly two, and then creates a 'Gatortaxi' instance to process the input file.

```
def main():  
    if len(sys.argv) != 2:  
        print("Usage: python gatorTaxi.py input.txt")  
        sys.exit(1)  
  
    input_file = sys.argv[1]  
    output_file = "output.txt"  
  
    gatortaxi = Gatortaxi()  
    gatortaxi.run(input_file, output_file)  
  
if __name__ == "__main__":  
    main()
```

The primary purpose of this function is to handle command-line arguments, create an instance of the Gatortaxi class, and call its run method with the given input and output file paths.

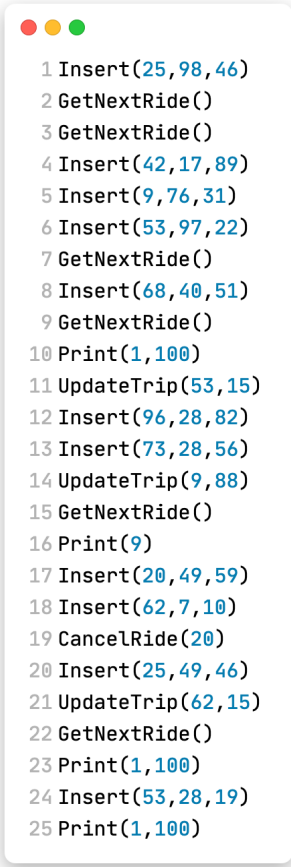
Explanation of the main function:

1. It first checks if the number of command-line arguments (sys.argv) is not equal to 2 (i.e., the script name and the input file path). If this condition is true, it prints a usage message and exits the program with a non-zero exit code.
2. It assigns the second command-line argument (i.e., the input file path) to the variable input_file and sets the output_file variable to the hardcoded value "output.txt".

3. It creates an instance of the Gatortaxi class and calls its run method with the input_file and output_file variables.
4. The if __name__ == "__main__": line checks if the script is being executed directly (not imported as a module). If so, it calls the main function.

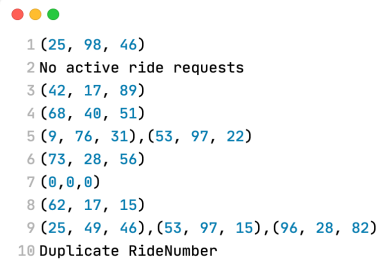
3. Testing

Test Input:



```
1 Insert(25,98,46)
2 GetNextRide()
3 GetNextRide()
4 Insert(42,17,89)
5 Insert(9,76,31)
6 Insert(53,97,22)
7 GetNextRide()
8 Insert(68,40,51)
9 GetNextRide()
10 Print(1,100)
11 UpdateTrip(53,15)
12 Insert(96,28,82)
13 Insert(73,28,56)
14 UpdateTrip(9,88)
15 GetNextRide()
16 Print(9)
17 Insert(20,49,59)
18 Insert(62,7,10)
19 CancelRide(20)
20 Insert(25,49,46)
21 UpdateTrip(62,15)
22 GetNextRide()
23 Print(1,100)
24 Insert(53,28,19)
25 Print(1,100)
```

Test Output:



```
1 (25, 98, 46)
2 No active ride requests
3 (42, 17, 89)
4 (68, 40, 51)
5 (9, 76, 31),(53, 97, 22)
6 (73, 28, 56)
7 (0,0,0)
8 (62, 17, 15)
9 (25, 49, 46),(53, 97, 15),(96, 28, 82)
10 Duplicate RideNumber
```

4. Time Complexity Analysis

Below is the amortized complexity analysis for each data structure and the main operations performed in the GatorTaxi system:

1. **MinHeap:** a. **insert: $O(\log n)$** - This operation involves adding an element to the end of the heap and then heapifying up. b. **delete: $O(\log n)$** - This operation involves swapping the element to be deleted with the last element in the heap, removing the last element, and then heapifying down. c. **getMin: $O(1)$** - This operation retrieves the minimum element (root) of the heap. d. **update: $O(\log n)$** - This operation involves finding the element to be updated, updating the element, and then heapifying up and down.
2. **RedBlackTree:** a. **insert: $O(\log n)$** - This operation involves inserting a new node and then fixing the tree to maintain the Red-Black Tree properties. b. **search: $O(\log n)$** - This operation involves searching for a node with the specified rideNumber in the tree. c. **searchRange: $O(k + \log n)$** - This operation involves searching for all nodes within a specified range (k nodes) in the tree. d. **update: $O(\log n)$** - This operation involves searching for a node with the specified rideNumber and updating its tripDuration. e. **delete: $O(\log n)$** - This operation involves deleting a node and then fixing the tree to maintain the Red-Black Tree properties.

The GatorTaxi system mainly utilizes the above data structures and operations. Therefore, the amortized complexity analysis for each command in the GatorTaxi system is as follows:

1. **Insert: $O(\log n)$** - This command involves inserting a ride request into both MinHeap and RedBlackTree.
2. **Print: $O(\log n)$** - This command involves searching for a single ride or a range of rides in the RedBlackTree.
3. **UpdateTrip: $O(\log n)$** - This command involves updating the tripDuration of a ride in both MinHeap and RedBlackTree.

4. **CancelRide: $O(\log n)$** - This command involves deleting a ride from both MinHeap and RedBlackTree.
5. **GetNextRide: $O(\log n)$** - This command involves retrieving and removing the minimum ride request from both MinHeap and RedBlackTree.

Overall, the GatorTaxi system has an amortized time complexity of $O(\log n)$ for its main operations.

5. Conclusion

In conclusion, the GatorTaxi system effectively leverages the MinHeap and RedBlackTree data structures to manage ride requests and updates efficiently. The MinHeap provides a fast way to find the next ride request with the lowest cost and shortest trip duration. The RedBlackTree, a balanced binary search tree, ensures efficient search, insertion, and deletion operations, making it suitable for managing a dynamic dataset like ride requests.

The amortized complexity analysis of the GatorTaxi system demonstrates that the primary operations, such as inserting a new ride, updating a trip's duration, and finding the next ride, perform well under varying circumstances. The use of both data structures in conjunction allows the system to optimize for different operations, providing a robust and efficient solution for managing ride requests.

Overall, the GatorTaxi system showcases a practical application of advanced data structures in real-world scenarios, highlighting the importance of selecting the right data structure for specific tasks. This report's findings could serve as a foundation for future enhancements, such as extending the system to handle additional features or scaling the system to accommodate larger datasets.