LAST NAME: Garg

FIRST NAME: Mayank

UFID: 60518194

UF MAIL: mayankgarg@ufl.edu

I have used **Python** for coding this project of delivery order management system. This defines various functions for handling delivery orders, such as creating orders, canceling orders, updating order delivery times, and printing orders based on different criteria. The python script uses an AVL tree data structure to efficiently manage the delivery orders.

The Explanation for the Code from gatorDelivery.py,

Importing treeNode and avlTree from the avlTree module. These classes are used to implement the AVL tree.

Initializing the AVL tree myTree, a dictionary nodes to store nodes, and data structures orders to manage delivery orders. The orders data structure contains lists for storing the estimated time of arrival (ETA), priority, ID, delivery time, and delivery status of each order. The lastReturnTime variable keeps track of the time when the delivery person last returned from a delivery. currentOrderSize is used to track the current number of orders.

1)**getPriority** function calculates the priority of an order based on its value and the time it was created.

2)**printByOrder** function prints the details of an order given its ID.

3)**printByTime** function prints the IDs of orders with an ETA between the given time range.

4)**getRankOfOrder** function returns the position of an order in the sequence of all orders.

5)**createOrder** function creates a new order with the given ID, current system time, order value, and delivery time. It checks for delivered orders and updates the ETA of other orders accordingly. The function inserts the new order into the AVL tree and updates the nodes dictionary and orders data structure.

6)**cancelOrder** function cancels an order with the given ID and current system time. If the order has already been delivered, the function returns an error message. Otherwise, it updates the ETA of other orders and removes the cancelled order from the AVL tree, nodes dictionary, and orders data structure.

7)**updateTime** function updates the delivery time of an order with the given ID, current system time, and new delivery time. If the order has already been delivered, the function returns an error message. Otherwise, it updates the ETA of other orders and modifies the order in the AVL tree, nodes dictionary, and orders data structure.

8)**outputRemaining** function prints the IDs and ETAs of all remaining orders when the script receives the "Quit()" command.

9)**processCommand** function processes commands from an input file and returns the appropriate output. The script reads commands from the input file line by line and processes them using this function.

10)The main block reads commands from an input file, processes them using the **processCommand** function, and writes the output to a file in the format input_file_output_file.txt.

The code is for a delivery order management system that uses an AVL tree to efficiently manage orders based on their ETA. It provides functions for creating, canceling, and updating orders, as well as printing orders based on different criteria.

Here are the logic statements from the code:

1. **if __name__ == "__main__"::** This is the main block of the script. It reads commands from an input file, processes them using the **processCommand** function, and writes the output to a file.

2. **cmd = f.readline():** This line reads a command from the input file.

3. while not **cmd.startswith**("Quit()"):: This loop continues until the "Quit()" command is received. It reads a command from the input file, processes it

using the **processCommand** function, and writes the output to the output file.

4. if **cmd.startswith**("Quit()"):: This condition checks if the "Quit()" command has been received. If it has, the script prints the remaining orders and exits.

5. **if args == "CreateOrder"::** This condition checks if the command is a "CreateOrder" command. If it is, the script creates a new order with the given parameters and inserts it into the AVL tree.

6. **elif args == "CancelOrder"::** This condition checks if the command is a "CancelOrder" command. If it is, the script cancels the order with the given ID and current system time.

7. **elif args == "UpdateTime"::** This condition checks if the command is an "UpdateTime" command. If it is, the script updates the delivery time of the order with the given ID, current system time, and new delivery time.

8. **elif args == "PrintByOrder"::** This condition checks if the command is a "PrintByOrder" command. If it is, the script prints the details of the order with the given ID.

9. **elif args == "PrintByTime"::** This condition checks if the command is a "PrintByTime" command. If it is, the script prints the IDs of orders with an ETA between the given time range.

10. **elif args == "GetRankOfOrder"::** This condition checks if the command is a "GetRankOfOrder" command. If it is, the script returns the position of the order with the given ID in the sequence of all orders.

This code implements an AVL tree, which is a self-balancing binary search tree. The tree is balanced by ensuring that the difference in height between the left and right subtrees is at most 1.

Explanation of the code from avlTrees.py:

1. **Class Node:** This class represents a node in the AVL tree. Each node has an id, createTime, value, deliveryTime, eta, and priority.

2. **class AVLTree:** This class represents the AVL tree itself. It has methods for inserting a new node, deleting a node, and balancing the tree.

3. **def insert(self, key, id, value, deliveryTime, eta, priority)**: This method inserts a new node into the AVL tree. It first finds the correct position for the new node and inserts it there. Then, it updates the height of the node and its ancestors. Finally, it balances the tree by rotating nodes if necessary.

4. **def delete(self, cur, key, id):** This method deletes a node from the AVL tree. It first finds the node to be deleted. If the node to be deleted has two children, it replaces the node with its in-order successor or predecessor. Then, it deletes the node and updates the height of the ancestors. Finally, it balances the tree by rotating nodes if necessary.

5. **def rRotate(self, x, y):** This method performs a right rotation on node x. It updates the parent, leftChild, and rightChild pointers of the nodes involved in the rotation.

6. **def lRotate(self, x, y):** This method performs a left rotation on node x. It updates the parent, leftChild, and rightChild pointers of the nodes involved in the rotation.

7. **def rlRotate(self, x, y, z):** This method performs a right-left rotation on node x. It first performs a right rotation on y and then a left rotation on x.

8. **def lrRotate(self, x, y, z):** This method performs a left-right rotation on node x. It first performs a left rotation on y and then a right rotation on x.

9. **def balanceTree(self, cur):** This method balances the AVL tree. It checks the balance factor of the current node. If the balance factor is less than -1, it performs a right rotation. If the balance factor is greater than 1, it performs a left rotation.

10. **def getBf(node):** This method calculates the balance factor of a node. The balance factor is the difference in height between the left and right subtrees of a node.

11. **def getMin(node):** This method finds the node with the minimum priority in the AVL tree. It starts from the root and follows the left child pointers until it reaches a node with no left child.

12. **def nodeSwap(x, y):** This method swaps the values of two nodes. It first stores the values of node x in a temporary variable. Then, it assigns the values of node y to node x. Finally, it assigns the values from the temporary variable to node y.

AVL trees are a type of self-balancing binary search tree, where the height difference between the left and right subtrees of any node is at most 1. This ensures that the tree remains approximately balanced, resulting in efficient search, insert, and delete operations.

It starts from the root of the tree and follows the left child pointers until it reaches a node with no left child. This node is guaranteed to have the minimum priority in the tree, as it is the first node encountered when traversing the tree in ascending order of priority.

The delete function it replaces the node with its in-order successor or predecessor. Then, it deletes the node and updates the height of the ancestors in the end it balances the tree by rotating nodes if necessary.

The delete function uses the getMin function to find the node with the minimum priority in the right subtree of the node to be deleted. If this node has a left child, it replaces the node with its left child. Otherwise, it replaces the node with its right child. The getMin function is then used to update the left child of the replacement node. This ensures that the tree remains balanced after the deletion.

The delete function then deletes the node and updates the height of the ancestors. Finally, it balances the tree by rotating nodes if necessary.

The delete function uses the **rRotate, lRotate, rlRotate, and lrRotate** functions to rotate nodes in the AVL tree. These functions are used to maintain the balance of the tree after a deletion. The balanceTree function is used to balance the AVL tree. It checks the balance factor of the current node. If the balance factor is less than -1, it performs a right rotation. If the balance factor is greater than 1, it performs a left rotation.

The **getBf** function is the balance factor, it is the difference in height between the left and right subtrees of a node.

The nodeSwap function is used to swap the values of two nodes. It first stores the values of node x in a temporary variable. Then, it assigns the values of node y to node x. at the end it assigns the values from the temporary variable to node y.