COP 5536 Project Report

Name: Yuyang Zhang

UFID: 13371560

UF Email: [yuyangzhang@ufl.edu](mailto:yuyangzhang@ufl.edu)

**Description:**

The project is designed to satisfied the requirement of GatorGlide Delivery Co. to elevate its software infrastructure to meet the growing demands of customers. The program is created by python language. It receives a text file as input that lists the operations of the system, and output a text file that lists the corresponding message of input operations. This report introduces details, function prototypes, and explanations of each class with their attributes and methods. At the end, a diagram is displayed to introduce the structure of the entire program.

**Instructions to Run:**

1. Navigate to the project directory, make sure the gatorDelivery.py and input text file are under this directory. Then run the following set of commands:

python gatorDelivery.py <input text file>

2. The output for the program will be written to the filename called ‘input\_filename’\_output\_file.txt’. For example, if you apply the default name ‘test1.txt’, the output file name is ‘test1\_output\_file.txt’.

**Function prototypes and explanations:**

OrderNode Class:

**Attributes:**

1. ‘priority’: Priority of the order.

2. ‘order\_id’: Unique identifier for the order.

3. ‘order\_creation\_time’: Time when the order was created.

4. ‘order\_value’: Value of the order.

5. ‘delivery\_time’: Time taken for delivery of the order.

6. ‘eta’: Estimated time of arrival for the order.

7. ‘left’: Pointer to the left child node.

8. ‘right’: Pointer to the right child node.

9. ‘height’: Height of the node in the AVL tree.

OrderTree Class:

**Attributes:**

1. ‘root’: Root node of the AVL tree representing orders.

**Methods:**

**init(self):**

* Purpose: Initializes an empty binary search tree with no root.
* Algorithm: Initializes the root node to None.

**insert(self, priority, order\_id, order\_creation\_time, order\_value, delivery\_time, eta):**

* Purpose: Inserts a new node into the tree.
* Algorithm: Calls the \_insert helper function with the provided parameters. Handles rotation and balance checks to maintain AVL property.

**\_insert(self, node, priority, order\_id, order\_creation\_time, order\_value, delivery\_time, eta):**

* Purpose: Helper function to recursively insert a new node into the tree.
* Algorithm: Recursively traverses the tree to find the appropriate position for insertion. Inserts the new node at the correct position. Updates the height of the current node and performs rotations if necessary to maintain balance.

**\_get\_height(self, node):**

* Purpose: Gets the height of a node.
* Algorithm: Returns the height of the node if it exists, otherwise returns 0.

**\_get\_balance(self, node):**

* Purpose: Gets the balance factor of a node.
* Algorithm: Calculates the balance factor by subtracting the height of the right subtree from the height of the left subtree.

**\_rotate\_left(self, z):**

* Purpose: Performs a left rotation to balance the tree.
* Algorithm: Rotates the given node 'z' towards the left.

**\_rotate\_right(self, z):**

* Purpose: Performs a right rotation to balance the tree.
* Algorithm: Rotates the given node 'z' towards the right.

**search(self, order\_id):**

* Purpose: Searches for an order with a given order ID in the tree.
* Algorithm: Calls the \_search helper function with the root node and the order ID.

**\_search(self, node, order\_id):**

* Purpose: Helper function to recursively search for an order in the tree.
* Algorithm: Recursively traverses the tree to find the node with the given order ID.

**find\_predecessor(self, priority):**

* Purpose: Finds the predecessor node of a given order priority.
* Algorithm: Starts traversing the tree from the root and moves left or right based on the priority. If the priority is less than the current node's priority, moves right and updates the predecessor. If the priority is greater, moves left. If the priority matches, finds the maximum value node in the left subtree.

**\_find\_max(self, node):**

* Purpose: Helper function to find the maximum value node in a subtree.
* Algorithm: Starts from the given node and traverses as far right as possible in the subtree. Continuously moves to the right child until reaching the rightmost leaf node.

**find\_successor(self, priority):**

* Purpose:Finds the successor node of a given order priority in the AVL tree.
* Algorithm: Start from the root node and traverse down the tree. If the priority of the current node is greater than the given priority, move to the left subtree. If the priority of the current node is less than the given priority, move to the right subtree. If the priority of the current node is equal to the given priority, find the minimum value node in the right subtree, which will be the successor.

**\_find\_min(self, node):**

* Purpose: Helper function to find the minimum value node in a subtree.
* Algorithm: Traverse to the leftmost node of the subtree by following the left child pointers recursively until reaching a leaf node.

**delete(self, order\_id):**

* Purpose: Deletes an order with a given order ID from the AVL tree.
* Algorithm:Find the node corresponding to the given order ID in the AVL tree.Perform deletion based on three cases:

1. Node with no child or only one child: Simply remove the node.

2.Node with two children: Find the inorder successor.Replace the node's values with the successor's values.Delete the successor node.

After deletion, update the height and balance factor of the nodes and perform rotations if necessary to maintain balance.

**\_delete(self, node, priority):**

* Purpose: Helper function to recursively delete a node from the tree.
* Algorithm: If the current node is None, return None. Find the node to be deleted based on its priority. Implement deletion based on the three cases mentioned in the delete method.

DeliveryNode Class:

**Attributes:**

1. ‘eta’: The estimated time of arrival for the delivery.

2. ‘order\_id’: The ID of the order associated with the delivery.

3. ‘left’: Pointer to the left child node.

4. ‘right’: Pointer to the right child node.

5. ‘height’: The height of the node in the AVL tree, initially set to 1.

**Methods:**

**init(self):**

* Purpose: Initializes an empty binary search tree with no root.
* Algorithm: Initializes the root node to None.

**insert(self, eta, order\_id):**

* Purpose: Inserts a new delivery into the tree.
* Algorithm: Calls the \_insert helper function with the provided parameters. Handles rotation and balance checks to maintain AVL property.

**\_insert(self, node, eta, order\_id):**

* Purpose: Helper function to recursively insert a new node into the tree.
* Algorithm: Recursively traverses the tree to find the appropriate position for insertion. Inserts the new node at the correct position. Updates the height of the current node and performs rotations if necessary to maintain balance.

**\_get\_height(self, node):**

* Purpose: Get the height of a node in the AVL tree.
* Algorithm: Recursively calculates the height of the subtree rooted at the given node.

**\_get\_balance(self, node):**

* Purpose: Get the balance factor of a node in the AVL tree.
* Algorithm: Calculates the balance factor by subtracting the height of the right subtree from the height of the left subtree.

**\_rotate\_left(self, z):**

* Purpose: Perform a left rotation to balance the AVL tree.
* Algorithm: Performs a left rotation operation on the given node to maintain AVL balance. Updates the heights of the affected nodes.

**\_rotate\_right(self, z):**

* Purpose: Perform a right rotation to balance the AVL tree.
* Algorithm: Performs a right rotation operation on the given node to maintain AVL balance. Updates the heights of the affected nodes.

**search(self, order\_id):**

* Purpose: Search for an order with a given order ID in the tree.
* Algorithm: Recursively searches for the order ID starting from the root node. Returns the node containing the order if found, otherwise returns None.

**\_search(self, node, order\_id):**

* Purpose: Helper function to recursively search for an order in the tree.
* Algorithm: Recursively traverses the tree to find the node with the given order ID. Returns the node containing the order if found, otherwise returns None.

**\_find\_max(self, node):**

* Purpose: Helper function to find the maximum value (node with the largest ETA) in a subtree.
* Algorithm: Traverses to the rightmost node of the subtree to find the maximum value.

**\_find\_min(self, node):**

* Purpose: Helper function to find the minimum value (node with the smallest ETA) in a subtree.
* Algorithm: Traverses to the leftmost node of the subtree to find the minimum value.

**delete(self, order\_id):**

* Purpose: Delete an order with a given order ID from the tree.
* Algorithm: Finds the node with the given order ID, deletes it, and maintains AVL balance by performing rotations if necessary.

**search\_range(self, time1, time2):**

* Purpose: Search for orders within a specified time range based on their estimated time of arrival (ETA).
* Algorithm: Calls a helper function to perform a recursive search starting from the root node. The helper function traverses the tree and adds the order IDs of nodes with ETAs within the specified range to the result list.

**\_search\_range\_helper(self, node, time1, time2, result):**

* Purpose: Helper function to recursively search for orders within a time range.
* Algorithm: Recursively traverses the tree and checks if the node's ETA falls within the specified time range. If it does, the order ID is added to the result list. The search continues recursively in the left subtree if the node's ETA is greater than or equal to the lower bound of the time range, and in the right subtree if the node's ETA is less than or equal to the upper bound of the time range.

**inorder\_traversal(self):**

* Purpose: Perform an inorder traversal of the tree to retrieve all orders sorted by their ETA.
* Algorithm: Calls a helper function to perform an inorder traversal starting from the root node. The helper function recursively traverses the tree in an inorder manner, appending the ETAs and order IDs of each node to separate arrays.

**\_inorder\_traversal\_helper(self, node, eta\_array, order\_id\_array):**

* Purpose: Helper function to perform inorder traversal recursively.
* Algorithm: Recursively traverses the tree in an inorder manner, processing each node by appending its ETA and order ID to the respective arrays. Traversal starts from the left subtree, then processes the current node, and finally continues to the right subtree.

OMS Class:

**Attributes:**

1. 'current\_time': Class variable to track the current time.

2. 'order\_tree': Instance of OrderTree to manage orders.

3. 'delivery\_tree': Instance of DeliveryTree to manage deliveries.

**Methods:**

**init(self):**

* Purpose: Initializes an OrderTree and a DeliveryTree.
* Algorithm: Initializes instances of OrderTree and DeliveryTree.

**create\_order(self, order\_id, order\_creation\_time, order\_value, delivery\_time):**

* Purpose: Creates a new order and manages its insertion into the order tree and delivery tree.
* Algorithm: Calculates the order's priority based on creation time and value.Inserts the order into the order tree.Adjusts the order's position in the order tree if necessary.Calculates the estimated time of arrival (ETA) for the order. Writes creation message to a text file.Inserts the order into the delivery tree.Updates ETA for the order.Checks for any delivered orders.

**prints(self, order\_id):**

* Purpose: Retrieves and prints details of a specific order.
* Algorithm: Searches for the order in the order tree.If found, extracts order details and writes them to a text file.If not found, writes a message indicating the absence of orders with the given ID.

**print(self, time1, time2):**

* Purpose: Prints orders within a specified time range.
* Algorithm: Searches for orders with ETA between 'time1' and 'time2' in the delivery tree.Writes the result to a text file.

**calculate\_priority(self, order\_creation\_time, order\_value):**

* Purpose: Calculates the priority of an order based on its creation time and value.
* Algorithm: Calculates order priority using a weighted formula.

**calculate\_eta(self, order\_id):**

* Purpose: Calculates the estimated time of arrival (ETA) for a given order.
* Algorithm: Retrieves the order from the order tree.Calculates ETA based on delivery time, creation time, and successor's attributes.Assigns the calculated ETA to the order's attribute.

**deliver\_orders(self, current\_time):**

* Purpose: Delivers orders with an ETA less than or equal to the current time.
* Algorithm:Performs an inorder traversal of the delivery tree.Checks if the ETA of each node is less than or equal to the current time.If so, deletes the node from the delivery tree, writes a delivery message to a text file, and continues traversal.

**cancel\_order(self, order\_id, current\_system\_time):**

* Purpose: Cancels an order if it exists and has not been delivered.
* Algorithm:Checks if any orders are delivered at the current time.Searches for the order in the delivery tree.If found and not delivered, deletes the order from both the order tree and delivery tree, writes a cancellation message to a text file, and updates the ETA.If not found or already delivered, writes an error message to a text file.

**update\_time(self, order\_id, current\_system\_time, new\_delivery\_time):**

* Purpose: Updates the delivery time of an order if it exists and has not been delivered.
* Algorithm:Renews the current time.Updates the delivery time of the order.Checks if any orders are delivered at the current time.Searches for the order in the delivery tree.If found, updates all affected ETAs and delivers the orders.If not found, writes an error message to a text file.

**update\_eta(self, order\_id):**

* Purpose: Updates the estimated time of arrival (ETA) for all orders affected by a change in delivery time.
* Algorithm:Finds all affected orders by iterating through predecessors of the given order.Calculates and updates the ETA for each affected order.Writes the updated ETAs to a text file.Updates the ETA in the delivery tree.

**get\_rank\_of\_order(self, order\_id):**

* Purpose: Determines the position of an order in the delivery sequence.
* Algorithm:Performs an inorder traversal of the delivery tree to collect order IDs.Compares the given order ID with collected order IDs to determine its rank.Writes the order's rank to a text file.

Main

**process\_input(input\_file):**

* Purpose: Process input commands from a file and execute corresponding operations on the Order Management System.
* Algorithm:

1. Read input from the specified file.

2. Initialize an instance of the Order Management System (OMS).

3. Split the input string into lines and remove leading/trailing whitespace.

4. Iterate over each line in the input:

a. Split the line into operation and parameters.

b. Determine the operation and extract parameters accordingly.

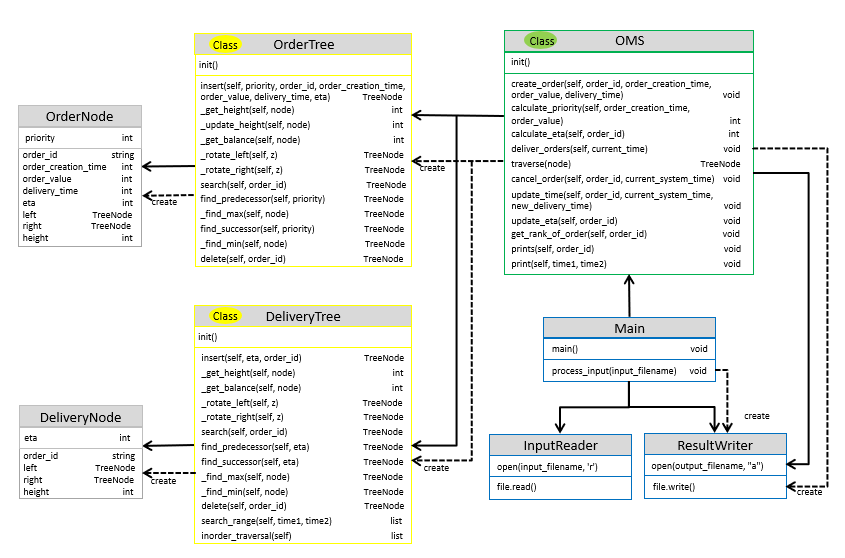
c. Execute the corresponding operation on the OMS instance.

5. Return when all input commands have been processed.

**main():**

* Purpose: Entry point of the program to process input and execute operations.
* Algorithm: Call process\_input() function to handle input processing and execute operations. Print a message indicating that the output has been written to a file.

Note: The structure of program is on next page

**Structure of Program:**