COP 5536 Spring 2024

Programming Project

GatorGlide Delivery Co.

Rong.Wei

UFID:13889478

Before the start of the project, I first think about the calculation of ETA according to the PDF document requirements, think about the logic, before starting to write, ETA calculation takes into account several factors:

1. current system time: the time when the order was created.
2. Delivery time: the time it takes for an order to be processed until delivery is complete.
3. the ETA and delivery time of the previous order: the system maintains a priority queue through `PriorityAVLTree`, which sorts orders according to their priority. The ETA of each new order is calculated based on the ETA of the order before it plus the delivery time of that order.

The ETA is calculated like this:

1. If the new order is the first order in the current system, or if no other order is in delivery at the time of creation, its ETA is the current system time plus its delivery time.
2. If there are other orders in delivery, the ETA of the new order is the ETA and delivery time of the previous order with higher priority plus its own delivery time.
3. If the creation of a new order affects the ETAs of existing orders, the ETAs of those orders are also updated to take into account the new situation.

In addition, when orders are canceled or delivery times are updated, the system updates the ETAs of the affected orders accordingly.

In short, the order management system ensures that orders are processed and delivered in the order and at the time they are intended to be by maintaining order prioritization and calculating ETAs. the ETA tree (EtaAVLTree) allows the system to quickly insert new orders, update the ETAs of existing orders, and locate orders that are about to arrive, thus ensuring that the entire system operates efficiently.

1.EtaAVLTree Class

1.introduction

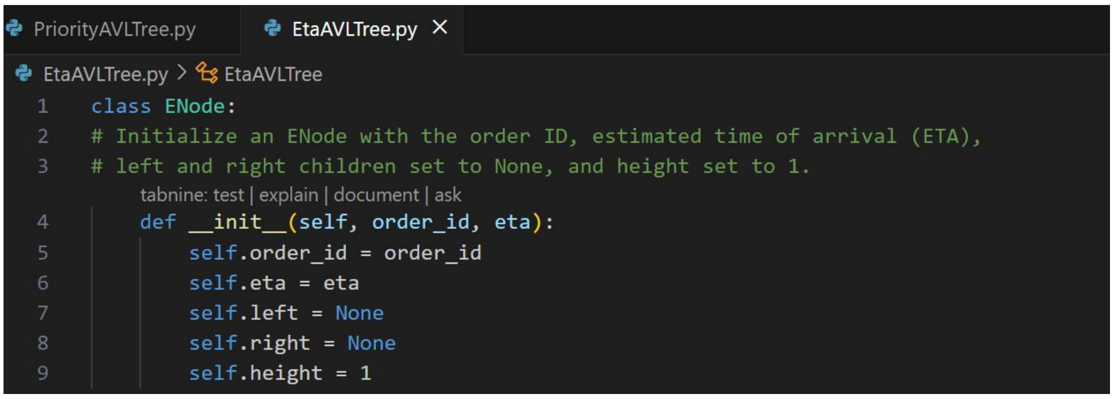
An AVL tree is a self-balancing binary search tree that is used to maintain a balanced state of a data structure and ensure the efficiency of search, insertion and deletion operations. In an order management system, the `EtaAVLTree` class utilizes the performance benefits of AVL trees to manage orders, sorting and processing them based on their estimated time of arrival (ETA).

2.EtaAVLTree Class Overview

The `EtaAVLTree` class implements an AVL tree structure based on Estimated Time of Arrival (ETA) for fast insertion, search and deletion of orders. In this way, the system can efficiently process and schedule orders to optimize the logistics and distribution process.

2.1 ENode Class

The `ENode` class is the node in the tree that contains the order information：


PriorityAVLTree.py
EtaAVLTree.py
EtaAVLTree.py > EtaAVLTree
tabnine: test | explain | document | ask

```python
class ENode:
# Initialize an ENode with the order ID, estimated time of arrival (ETA),
# left and right children set to None, and height set to 1.
    def __init__(self, order_id, eta):
        self.order_id = order_id
        self.eta = eta
        self.left = None
        self.right = None
        self.height = 1
```

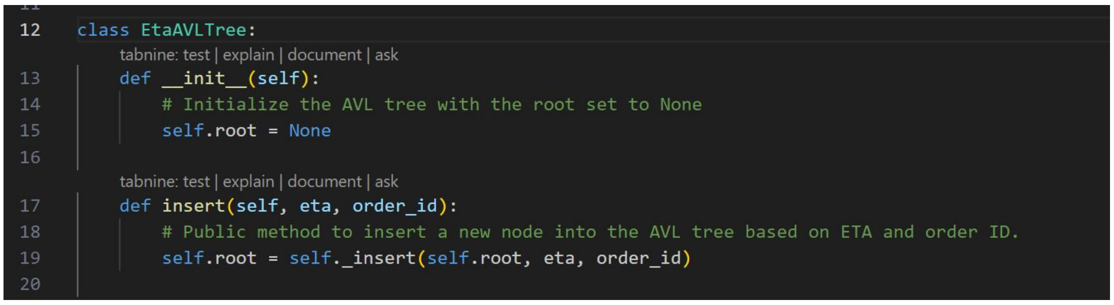
`order_id`: unique identifier of the order.
`eta`: estimated arrival time of the order.
`left`, `right`: links to left and right child nodes.
`height`: height of the node, used for AVL balancing.

3. Core functionality

3.1 Insertion operation

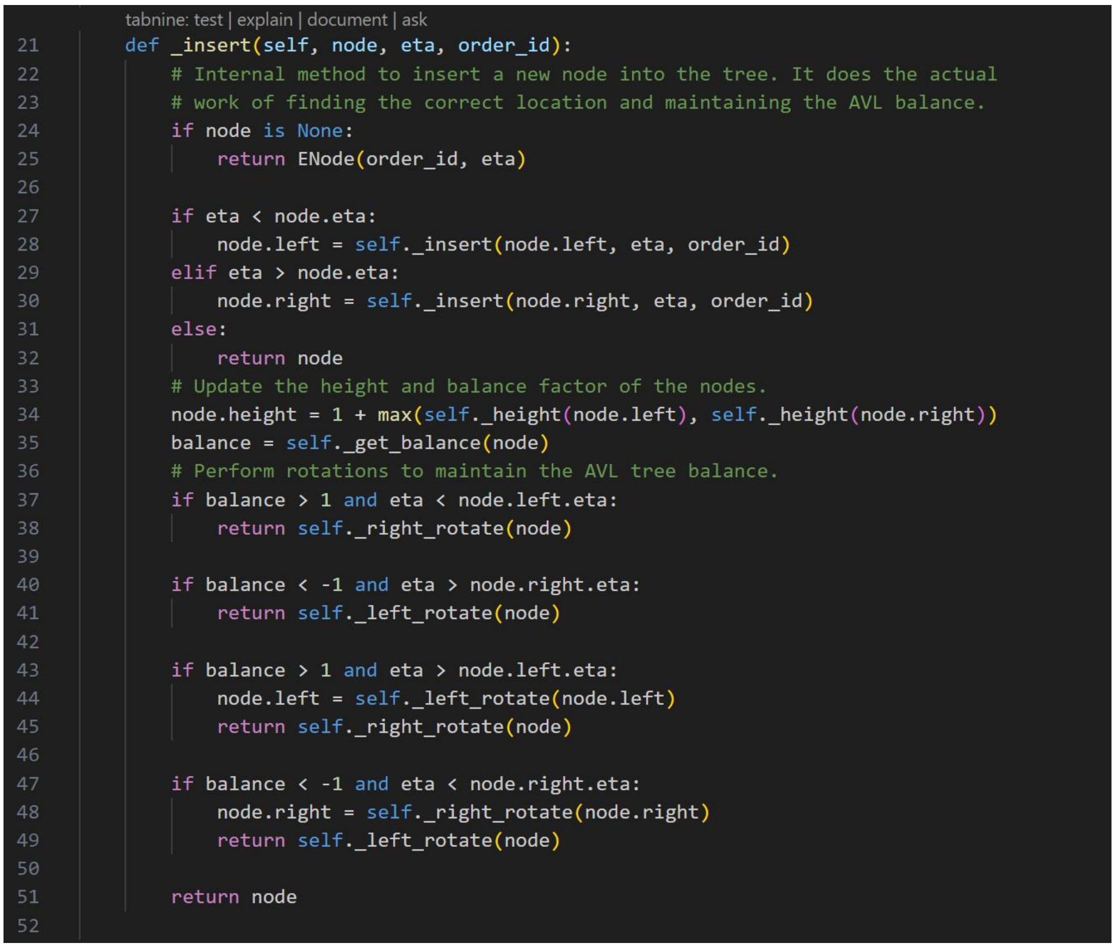
When a new order is inserted, it is placed in the appropriate position based on its expected

arrival time and the necessary rotation operations are performed to keep the tree balanced:
The insert method is the public method used to add a new node to the AVL tree.


tabnine: test | explain | document | ask
tabnine: test | explain | document | ask

```python
class EtaAVLTree:
    def __init__(self):
        # Initialize the AVL tree with the root set to None
        self.root = None

    def insert(self, eta, order_id):
        # Public method to insert a new node into the AVL tree based on ETA and order ID.
        self.root = self._insert(self.root, eta, order_id)

```

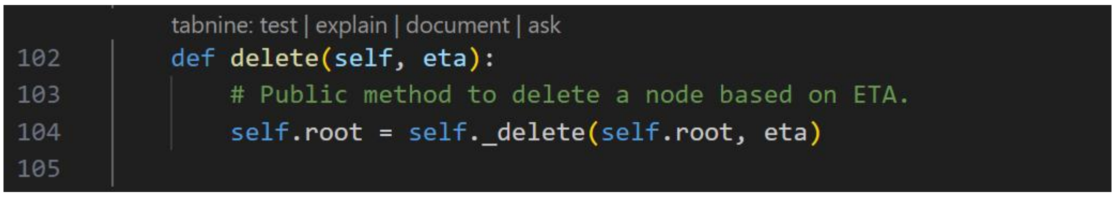
The _insert method is an internal method that is used to recursively find the correct insertion position and ensure that the tree is balanced.



```python
    def _insert(self, node, eta, order_id):
        # Internal method to insert a new node into the tree. It does the actual
        # work of finding the correct location and maintaining the AVL balance.
        if node is None:
            return ENode(order_id, eta)

        if eta < node.eta:
            node.left = self._insert(node.left, eta, order_id)
        elif eta > node.eta:
            node.right = self._insert(node.right, eta, order_id)
        else:
            return node
        # Update the height and balance factor of the nodes.
        node.height = 1 + max(self._height(node.left), self._height(node.right))
        balance = self._get_balance(node)
        # Perform rotations to maintain the AVL tree balance.
        if balance > 1 and eta < node.left.eta:
            return self._right_rotate(node)

        if balance < -1 and eta > node.right.eta:
            return self._left_rotate(node)

        if balance > 1 and eta > node.left.eta:
            node.left = self._left_rotate(node.left)
            return self._right_rotate(node)

        if balance < -1 and eta < node.right.eta:
            node.right = self._right_rotate(node.right)
            return self._left_rotate(node)

        return node

```

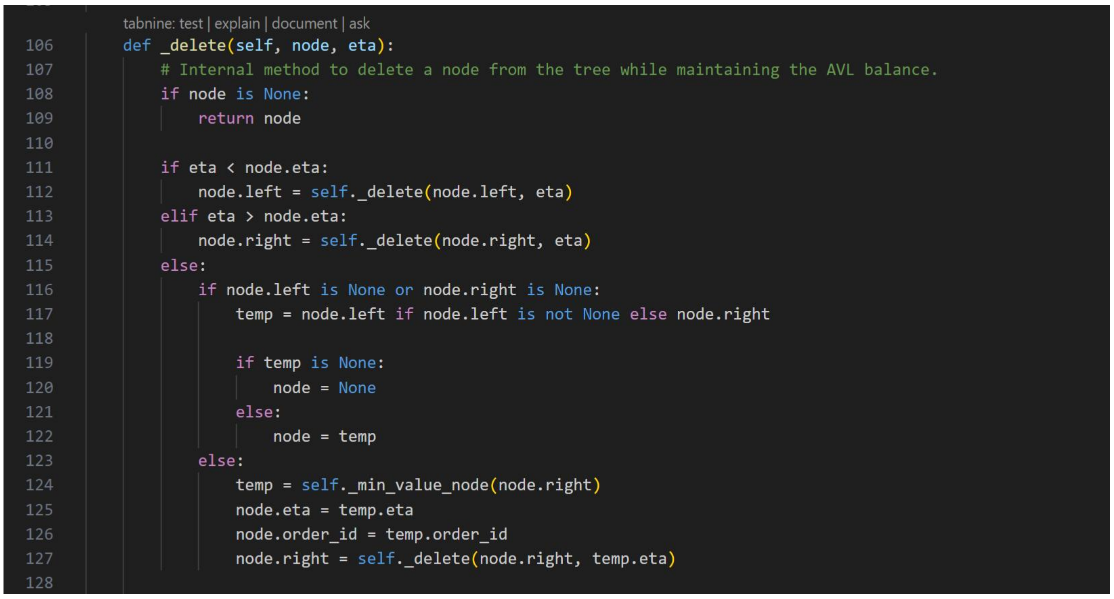
3.2 Delete operation

The delete operation allows the removal of nodes from the tree based on the ETA value of the order, while performing a balancing operation to maintain the overall efficiency of the tree structure:



```python
def delete(self, eta):
    # Public method to delete a node based on ETA.
    self.root = self._delete(self.root, eta)
```

The delete method is the public method for deleting a node with a specific ETA.

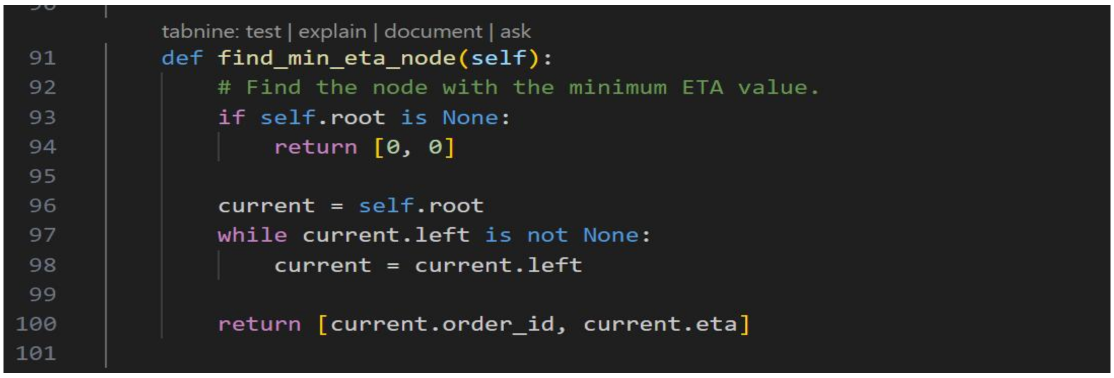


```python
def _delete(self, node, eta):
    # Internal method to delete a node from the tree while maintaining the AVL balance.
    if node is None:
        return node

    if eta < node.eta:
        node.left = self._delete(node.left, eta)
    elif eta > node.eta:
        node.right = self._delete(node.right, eta)
    else:
        if node.left is None or node.right is None:
            temp = node.left if node.left is not None else node.right

            if temp is None:
                node = None
            else:
                node = temp
        else:
            temp = self._min_value_node(node.right)
            node.eta = temp.eta
            node.order_id = temp.order_id
            node.right = self._delete(node.right, temp.eta)

```

The _delete method is an internal method that recursively deletes nodes and ensures that the tree remains balanced after deletion.

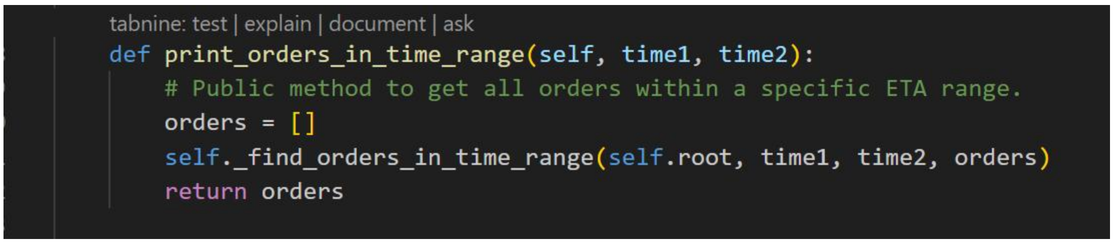
3.3 Find Minimum ETA Order

The `find_min_eta_node` method finds the order with the earliest arrival time, which is useful for determining the next order to be processed:



```python
def find_min_eta_node(self):
    # Find the node with the minimum ETA value.
    if self.root is None:
        return [0, 0]

    current = self.root
    while current.left is not None:
        current = current.left

    return [current.order_id, current.eta]
```

The find_min_eta_node method is used to find the node with the smallest ETA.

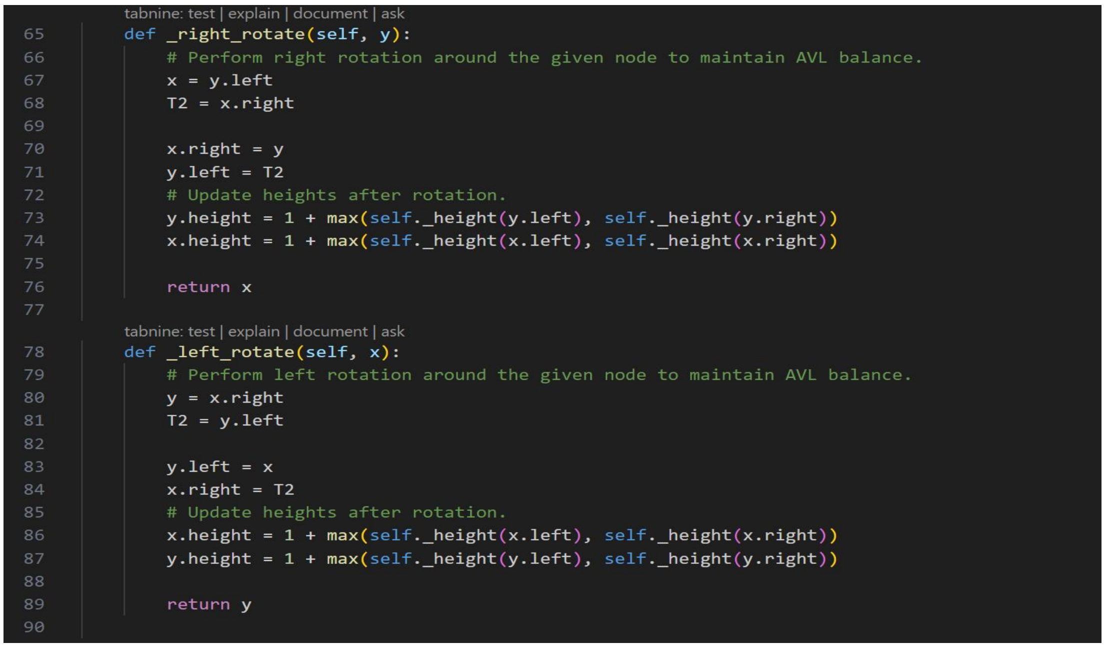


```python
def print_orders_in_time_range(self, time1, time2):
    # Public method to get all orders within a specific ETA range.
    orders = []
    self._find_orders_in_time_range(self.root, time1, time2, orders)
    return orders
```

The print_orders_in_time_range method is used to find and print all orders in a given time range.

3.4 Balancing and rotating operations

In order to keep the AVL tree balanced, `EtaAVLTree` implements left- and right-rotation operations that adjust the structure of the tree after nodes are inserted or deleted.


tabnine: test | explain | document | ask
tabnine: test | explain | document | ask

```python
def _right_rotate(self, y):
    # Perform right rotation around the given node to maintain AVL balance.
    x = y.left
    T2 = x.right

    x.right = y
    y.left = T2
    # Update heights after rotation.
    y.height = 1 + max(self._height(y.left), self._height(y.right))
    x.height = 1 + max(self._height(x.left), self._height(x.right))

    return x

def _left_rotate(self, x):
    # Perform left rotation around the given node to maintain AVL balance.
    y = x.right
    T2 = y.left

    y.left = x
    x.right = T2
    # Update heights after rotation.
    x.height = 1 + max(self._height(x.left), self._height(x.right))
    y.height = 1 + max(self._height(y.left), self._height(y.right))

    return y
```

These methods perform tree rotation operations to maintain AVL balance. Both right_rotate and left_rotate adjust the structure of the tree to ensure that the difference in height between the two subtrees of any one node is no more than one.In order to keep the AVL tree balanced, `EtaAVLTree` implements left- and right-rotation operations that adjust the structure of the tree after nodes are inserted or deleted.

4.Performance Advantages

By maintaining a balanced state, `EtaAVLTree` guarantees efficient insertion, deletion and lookup operations. For order management systems that require frequent updates and queries, this data structure provides significant performance benefits.

5. Conclusion

The `EtaAVLTree` class provides an efficient and reliable solution for order management through the self-balancing properties of AVL trees. It ensures that the system maintains an efficient response time even under frequent insertion and deletion operations. This makes `EtaAVLTree` ideal for processing time-sensitive orders, especially in the logistics and distribution areas where fast response and efficient management are required.

2. PriorityAVLTree Class

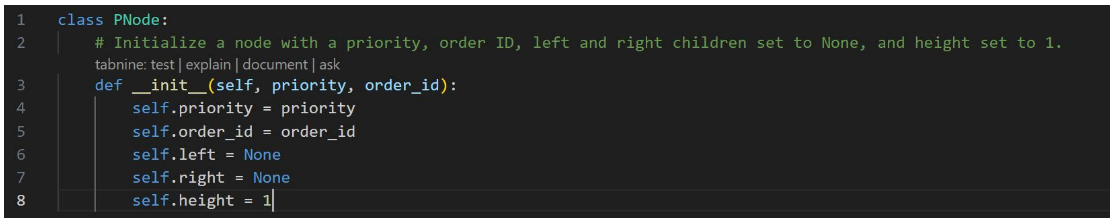
1. introduction

When it comes to efficiently managing data structures, balanced binary trees, and in particular AVL trees, are popular for their excellent time-complexity performance.An AVL tree is a self-balancing binary search tree that ensures that the height of the tree is kept at the logarithmic level, thus maintaining efficient performance on insertion, deletion, and lookup operations. An AVL tree class implemented as `PriorityAVLTree` is described in detail, designed to manage orders sorted according to priority.

2. PriorityAVLTree Class Overview

The `PriorityAVLTree` class utilizes the nature of an AVL tree to manage orders with different priorities, where each order is identified by its unique `order_id` and `priority`. The nodes in the tree (represented by the `PNode` class) are sorted according to priority and order ID to ensure that high priority orders can be accessed and processed quickly.

2.1 PNode Class



```python
class PNode:
    # Initialize a node with a priority, order ID, left and right children set to None, and height set to 1.
    def __init__(self, priority, order_id):
        self.priority = priority
        self.order_id = order_id
        self.left = None
        self.right = None
        self.height = 1
```

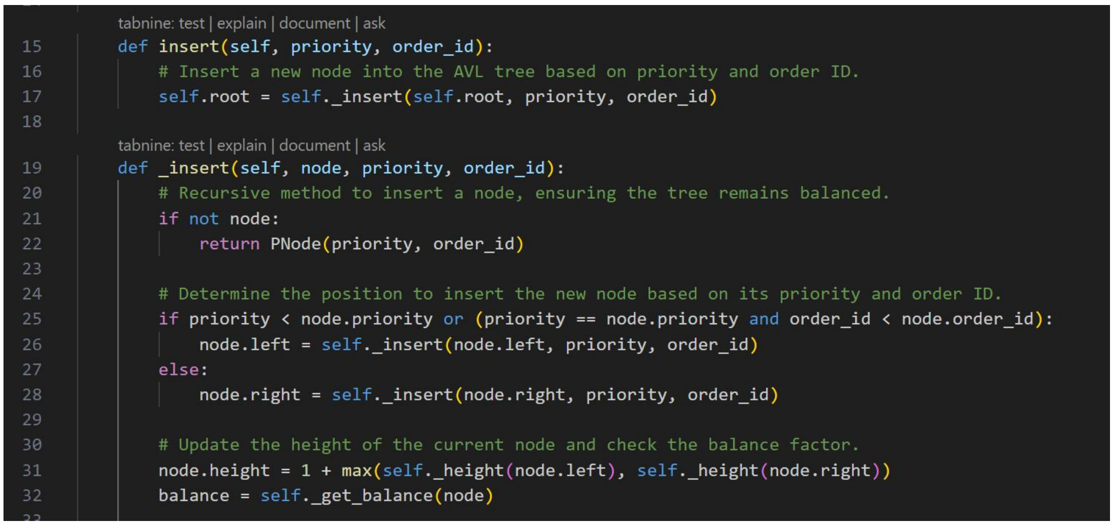
The `PNode` class is defined as follows:
`priority`: indicates the priority of the order.
`order_id`: represents the unique identifier of the order.
`left` and `right`: links to the left and right child nodes in the tree.
`height`: the height of the node in the tree, used for AVL balancing.

3. Core functionality

3.1 insertion operation

The insert operation inserts a new `PNode` into the `PriorityAVLTree`. The insert logic takes into account the priority and order IDs to ensure that the sorting and balancing properties of the tree are maintained.

1. insert Role: Provides an external interface to insert new nodes into the tree.


tabnine: test | explain | document | ask
tabnine: test | explain | document | ask

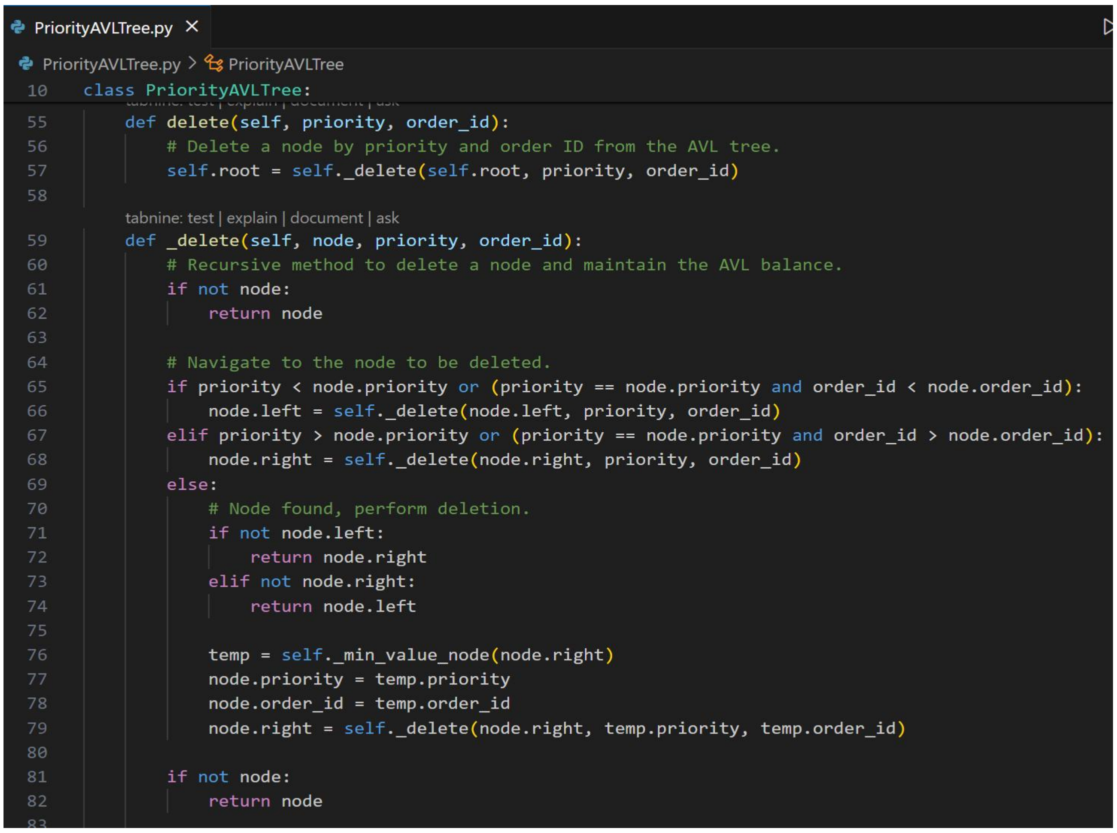
```python
    def insert(self, priority, order_id):
        # Insert a new node into the AVL tree based on priority and order ID.
        self.root = self._insert(self.root, priority, order_id)

    def _insert(self, node, priority, order_id):
        # Recursive method to insert a node, ensuring the tree remains balanced.
        if not node:
            return PNode(priority, order_id)

        # Determine the position to insert the new node based on its priority and order ID.
        if priority < node.priority or (priority == node.priority and order_id < node.order_id):
            node.left = self._insert(node.left, priority, order_id)
        else:
            node.right = self._insert(node.right, priority, order_id)

        # Update the height of the current node and check the balance factor.
        node.height = 1 + max(self._height(node.left), self._height(node.right))
        balance = self._get_balance(node)
```

2. _ insert Role: Recursively inserts new nodes and keeps the AVL tree balanced. Logic:Creates and returns a new node if the current node does not exist. Decides whether to insert into the left or right subtree based on priority and order number. Adjusts the balance of the tree after insertion (may require rotation).

3.2 Delete operation

The delete operation removes the corresponding node from the tree based on the given priority and order ID. This process is also required to maintain the balance of the AVL tree.

1. delete Role: Provides an external interface to remove a specific node from the tree.


PriorityAVLTree.py ×

PriorityAVLTree.py > PriorityAVLTree

```python
class PriorityAVLTree:
    def delete(self, priority, order_id):
        # Delete a node by priority and order ID from the AVL tree.
        self.root = self._delete(self.root, priority, order_id)

    tabnine: test | explain | document | ask
    def _delete(self, node, priority, order_id):
        # Recursive method to delete a node and maintain the AVL balance.
        if not node:
            return node

        # Navigate to the node to be deleted.
        if priority < node.priority or (priority == node.priority and order_id < node.order_id):
            node.left = self._delete(node.left, priority, order_id)
        elif priority > node.priority or (priority == node.priority and order_id > node.order_id):
            node.right = self._delete(node.right, priority, order_id)
        else:
            # Node found, perform deletion.
            if not node.left:
                return node.right
            elif not node.right:
                return node.left

            temp = self._min_value_node(node.right)
            node.priority = temp.priority
            node.order_id = temp.order_id
            node.right = self._delete(node.right, temp.priority, temp.order_id)

        if not node:
            return node
```

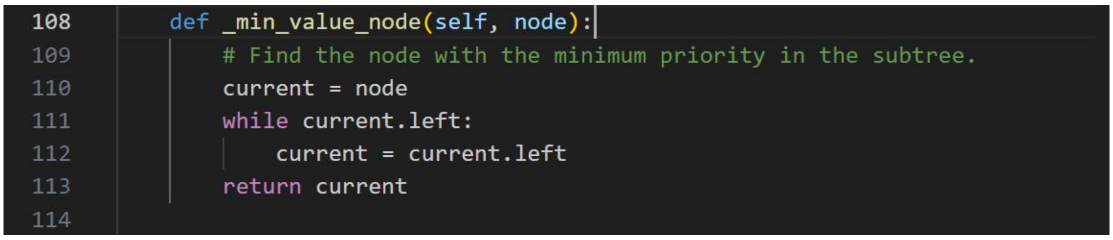
2._delete function: recursively deletes nodes and keeps the AVL tree balanced.
Logic: Find and delete the specified node. If a node has two child nodes, a successor node needs to be found to replace the deleted node. Adjust the balance of the tree after deletion (rotation may be required)

3.3 Find and Iterate

The code for these parts of the query operation can be categorized into the following major functional areas:

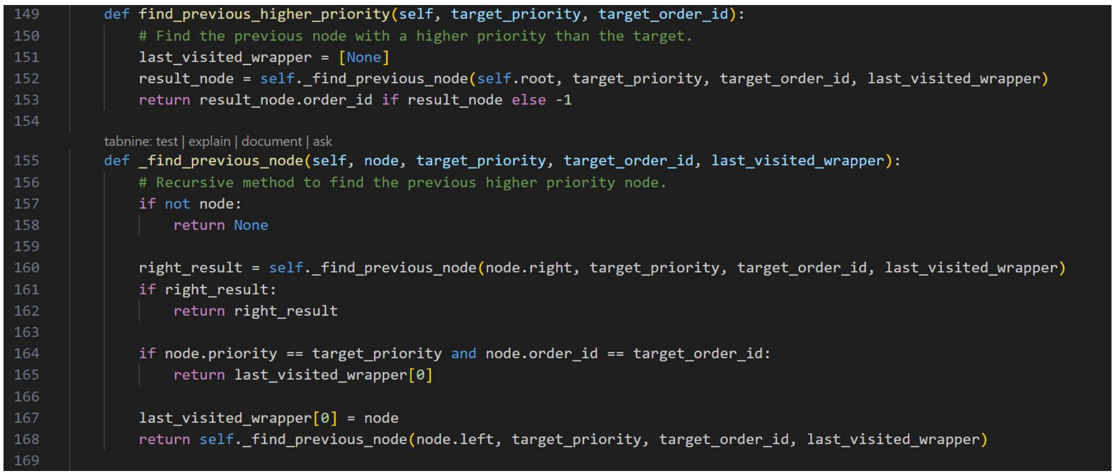
1. Query operations

The `_min_value_node` method is used to find the node with the lowest priority in the subtree, and is usually used when looking for a successor node in a delete operation.



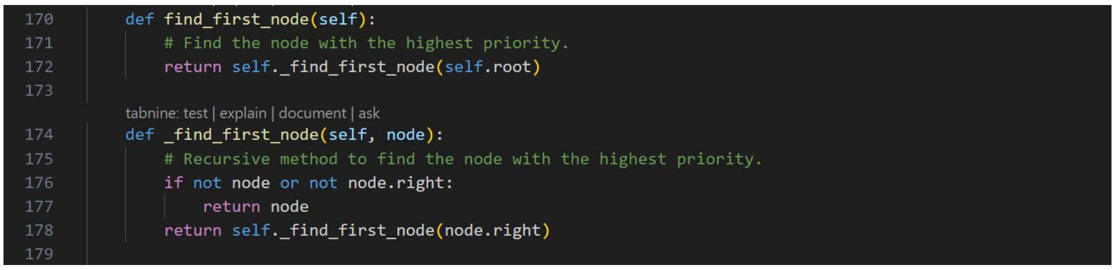
```python
    def _min_value_node(self, node):
        # Find the node with the minimum priority in the subtree.
        current = node
        while current.left:
            current = current.left
        return current

```

The `find_previous_higher_priority` method and the `_find_previous_node` method are used to find a node with a higher priority than the given target to find the next higher priority order.



```python
    def find_previous_higher_priority(self, target_priority, target_order_id):
        # Find the previous node with a higher priority than the target.
        last_visited_wrapper = [None]
        result_node = self._find_previous_node(self.root, target_priority, target_order_id, last_visited_wrapper)
        return result_node.order_id if result_node else -1

    def _find_previous_node(self, node, target_priority, target_order_id, last_visited_wrapper):
        # Recursive method to find the previous higher priority node.
        if not node:
            return None

        right_result = self._find_previous_node(node.right, target_priority, target_order_id, last_visited_wrapper)
        if right_result:
            return right_result

        if node.priority == target_priority and node.order_id == target_order_id:
            return last_visited_wrapper[0]

        last_visited_wrapper[0] = node
        return self._find_previous_node(node.left, target_priority, target_order_id, last_visited_wrapper)

```

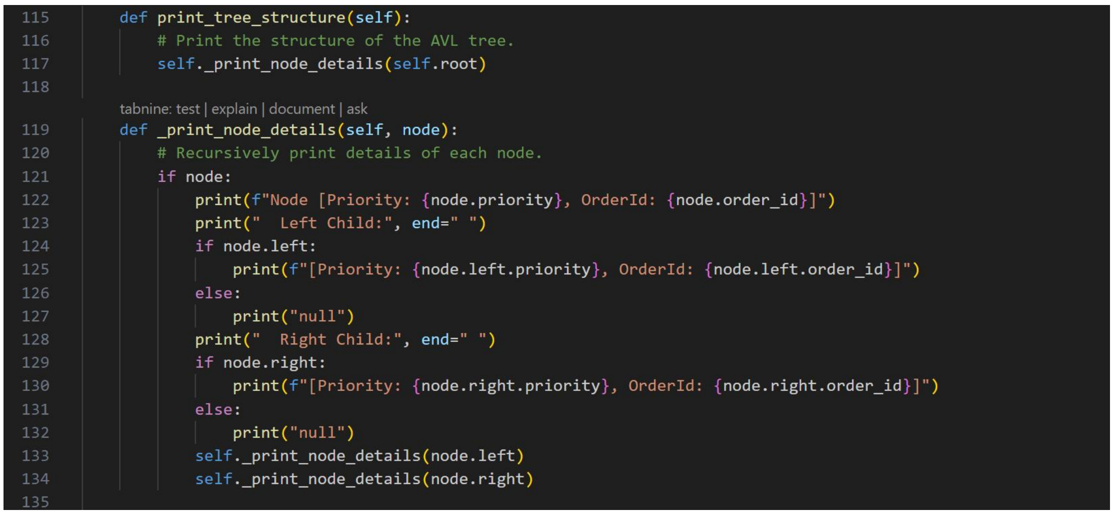
- The `find_first_node` method and the `_find_first_node` method are used to find the node with the highest priority, which is useful for determining the next processed item in a queue or list.



```python
    def find_first_node(self):
        # Find the node with the highest priority.
        return self._find_first_node(self.root)

    def _find_first_node(self, node):
        # Recursive method to find the node with the highest priority.
        if not node or not node.right:
            return node
        return self._find_first_node(node.right)

```

2. Iteration and printing

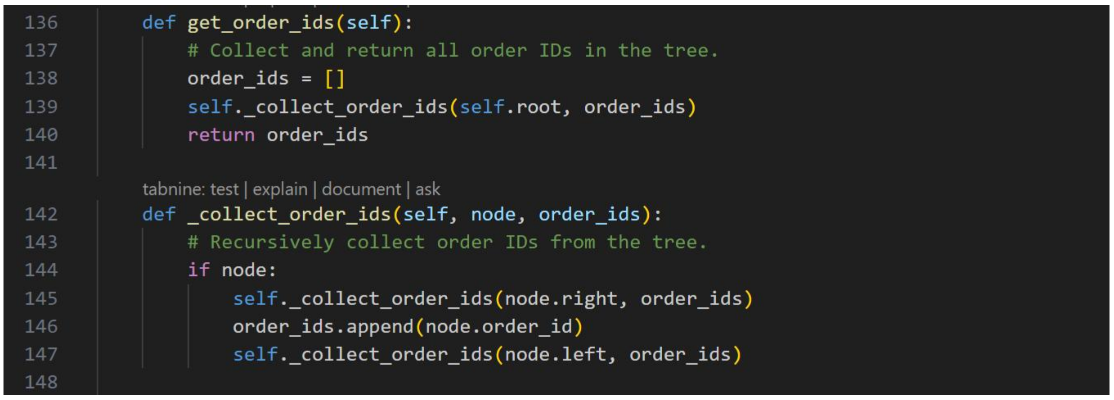
The `_print_tree_structure` method and the `_print_node_details` method are used to recursively traverse the tree and print the details of each node. These methods help visualize the structure of the tree and are useful for debugging and verifying the state of the tree.



```python
    def print_tree_structure(self):
        # Print the structure of the AVL tree.
        self._print_node_details(self.root)

    def _print_node_details(self, node):
        # Recursively print details of each node.
        if node:
            print(f"Node [Priority: {node.priority}, OrderId: {node.order_id}]")
            print("  Left Child:", end=" ")
            if node.left:
                print(f"[Priority: {node.left.priority}, OrderId: {node.left.order_id}]")
            else:
                print("null")
            print("  Right Child:", end=" ")
            if node.right:
                print(f"[Priority: {node.right.priority}, OrderId: {node.right.order_id}]")
            else:
                print("null")
            self._print_node_details(node.left)
            self._print_node_details(node.right)
```

3. Data collection

- The `get_order_ids` method and the `_collect_order_ids` method are used to collect and return all order numbers from the AVL tree. These methods can be used to generate a list of order numbers that may be used for reporting or further processing.

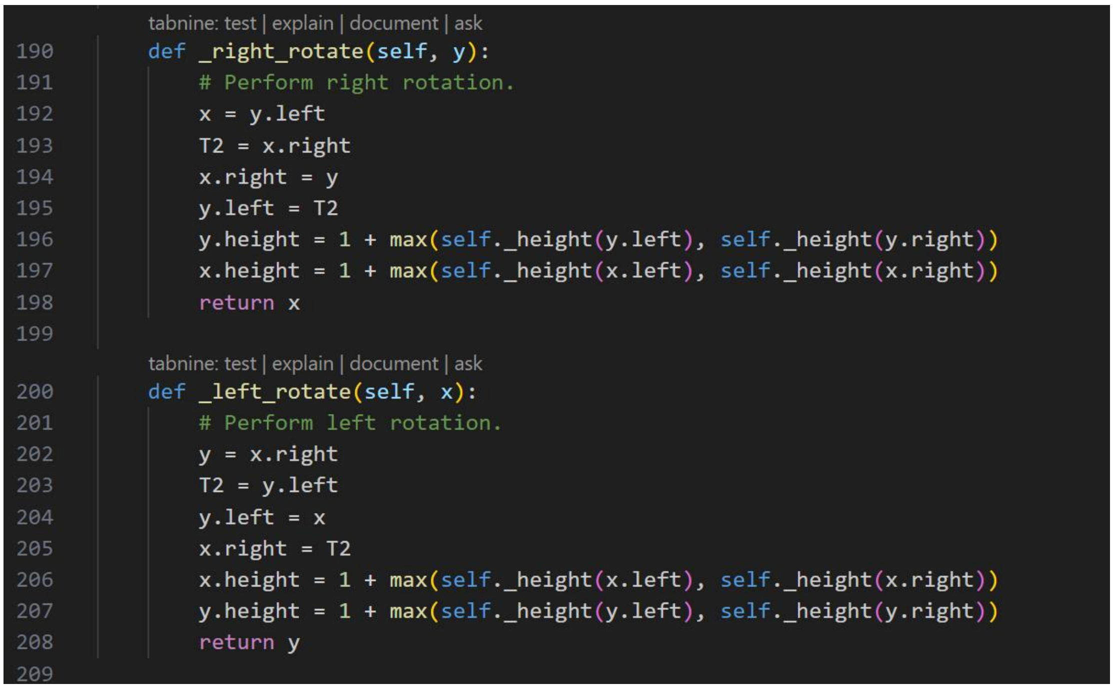


```python
    def get_order_ids(self):
        # Collect and return all order IDs in the tree.
        order_ids = []
        self._collect_order_ids(self.root, order_ids)
        return order_ids

    def _collect_order_ids(self, node, order_ids):
        # Recursively collect order IDs from the tree.
        if node:
            self._collect_order_ids(node.right, order_ids)
            order_ids.append(node.order_id)
            self._collect_order_ids(node.left, order_ids)
```

Such a classification helps to understand the organization of the code and the responsibilities of each part, which makes it easier to maintain and extend. Query operations focus on finding and accessing data in the tree, traversal and printing focus on displaying the tree structure, and data collection focuses on extracting information from the tree.

3.4 AVL tree balancing operations

To maintain the balance of the tree, the class implements left rotation (`_left_rotate`) and right rotation (`_right_rotate`) methods. These rotation operations adjust the structure of the tree according to the tree's balance factor when nodes are inserted or deleted.

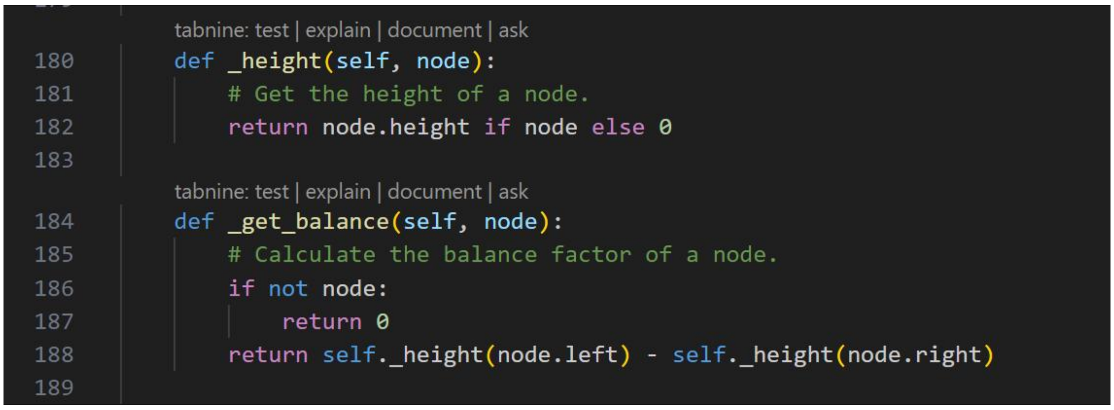


tabnine: test | explain | document | ask

```python
def _right_rotate(self, y):
    # Perform right rotation.
    x = y.left
    T2 = x.right
    x.right = y
    y.left = T2
    y.height = 1 + max(self._height(y.left), self._height(y.right))
    x.height = 1 + max(self._height(x.left), self._height(x.right))
    return x

def _left_rotate(self, x):
    # Perform left rotation.
    y = x.right
    T2 = y.left
    y.left = x
    x.right = T2
    x.height = 1 + max(self._height(x.left), self._height(x.right))
    y.height = 1 + max(self._height(y.left), self._height(y.right))
    return y
```

tabnine: test | explain | document | ask

Ancillary methods for trees



tabnine: test | explain | document | ask

```python
def _height(self, node):
    # Get the height of a node.
    return node.height if node else 0

def _get_balance(self, node):
    # Calculate the balance factor of a node.
    if not node:
        return 0
    return self._height(node.left) - self._height(node.right)
```

tabnine: test | explain | document | ask

Height: Get the height of the node.
Balance: Calculate the balance factor of the node, which is used to determine the rotation operation.

4. Performance Advantages

Because `PriorityAVLTree` maintains the balanced nature of the AVL tree, it is able to complete insertion, deletion, and lookup operations in logarithmic time, which makes it excellent for working with large amounts of data. This structure is especially important for systems that need to access and update prioritization information quickly, such as real-time order processing systems.

5. Conclusion

The `PriorityAVLTree` class takes full advantage of the self-balancing properties of AVL trees to provide an efficient way to manage and query priority-based orders. This not only ensures the efficiency of the operation, but also simplifies the order management logic. Therefore, `PriorityAVLTree` is ideal for systems that require efficient prioritization and retrieval functionality.

3.OrderManagementSystem Class

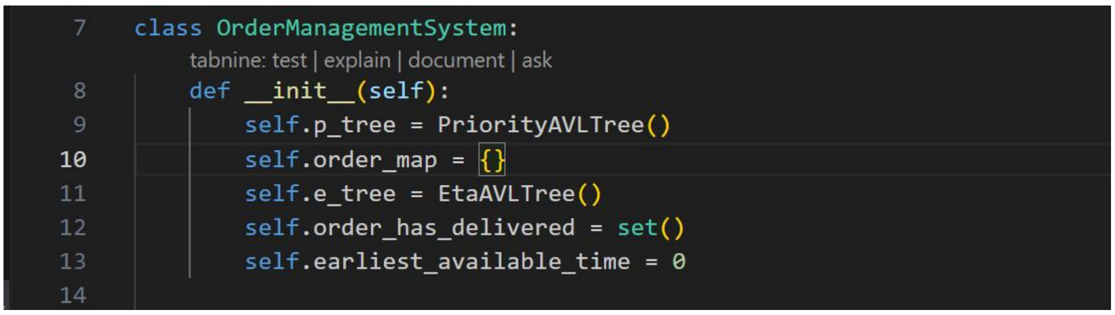
1. introduction

The `OrderManagementSystem` class uses the `PriorityAVLTree` and `EtaAVLTree` data structures to optimize order processing and ensure that orders are managed efficiently according to estimated time of arrival (ETA) and priority.

2.OrderManagementSystem Class Overview

`OrderManagementSystem` is an integrated order management system that uses AVL trees to process orders efficiently. Orders in the system are managed according to priority and estimated time of arrival (ETA) to ensure timely processing and delivery of orders.

2. Core functionality

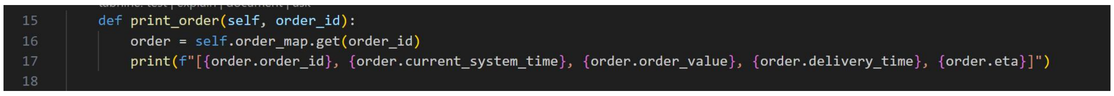


```python
class OrderManagementSystem:
    tabnine: test | explain | document | ask
    def __init__(self):
        self.p_tree = PriorityAVLTree()
        self.order_map = {}
        self.e_tree = EtaAVLTree()
        self.order_has_delivered = set()
        self.earliest_available_time = 0
```

`p_tree` (`PriorityAVLTree`): Manages the priority of an order.
`e_tree` (`EtaAVLTree`): Manages orders based on their estimated arrival time.
`order_map`: Stores a hash table of order IDs and their corresponding order objects.
`order_has_delivered`: Stores a collection of delivered orders.
`earliest_available_time`: Records the earliest available time in the system for processing order deliveries.

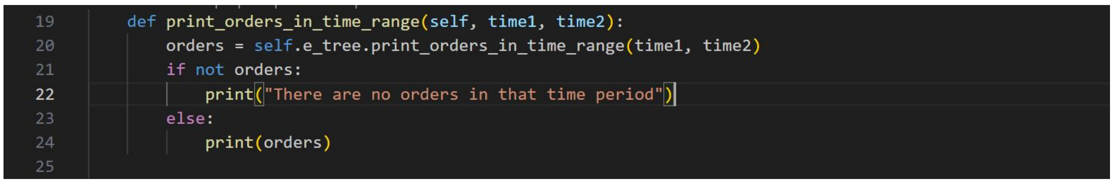
3. Functional approach

1. print_order(order_id)-Prints the details of the specified order ID.



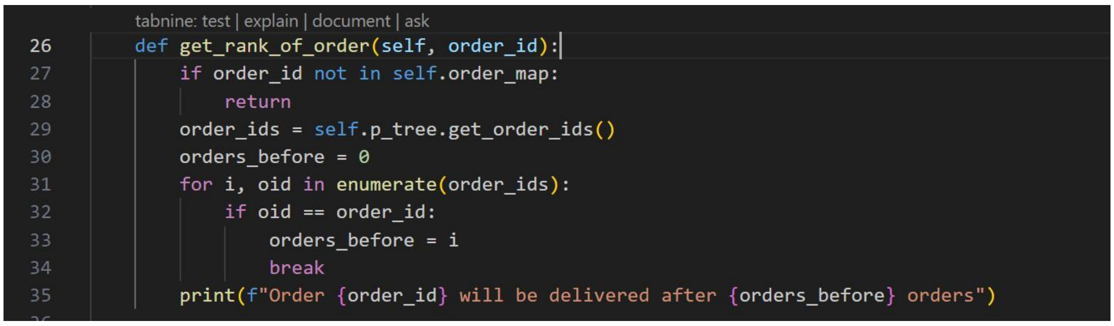
```python
def print_order(self, order_id):
    order = self.order_map.get(order_id)
    print(f"[{order.order_id}, {order.current_system_time}, {order.order_value}, {order.delivery_time}, {order.eta}]")
```

2. print_orders_in_time_range(time1, time2)-Prints all orders within the specified time range.



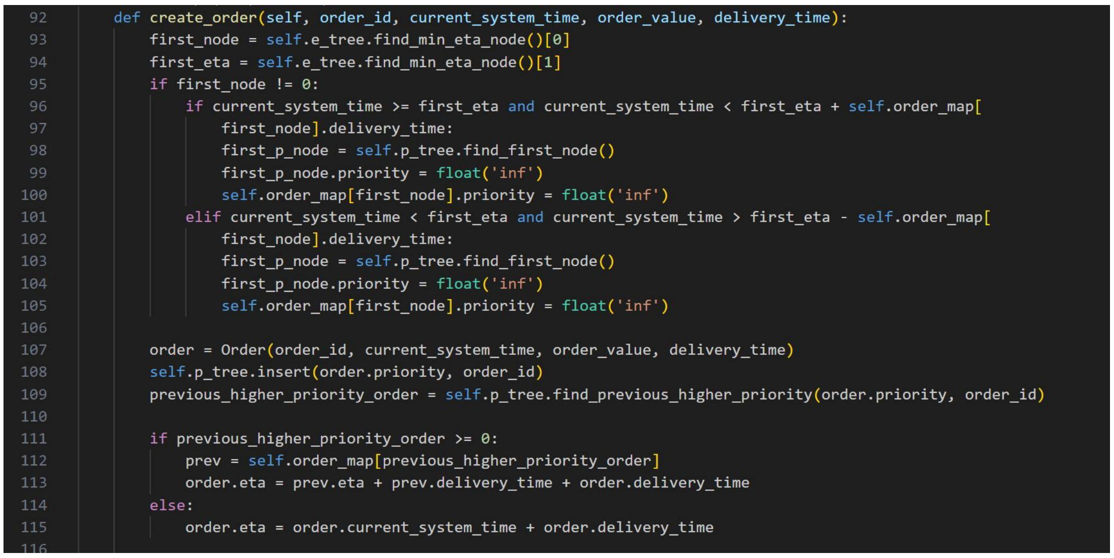
```python
def print_orders_in_time_range(self, time1, time2):
    orders = self.e_tree.print_orders_in_time_range(time1, time2)
    if not orders:
        print("There are no orders in that time period")
    else:
        print(orders)
```

3. get_rank_of_order(order_id)-Gets the rank of the specified order in the priority queue.



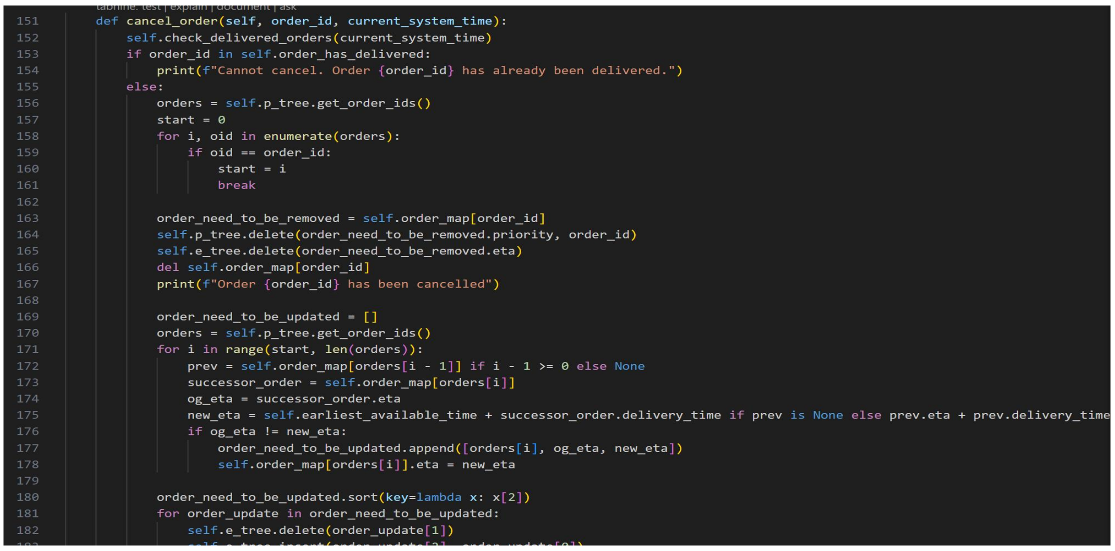
```python
tabnine: test | explain | document | ask
def get_rank_of_order(self, order_id):
    if order_id not in self.order_map:
        return
    order_ids = self.p_tree.get_order_ids()
    orders_before = 0
    for i, oid in enumerate(order_ids):
        if oid == order_id:
            orders_before = i
            break
    print(f"Order {order_id} will be delivered after {orders_before} orders")
```

4. create_order(order_id, current_system_time, order_value, delivery_time)-Creates a new order and inserts it into the Priority and ETA management tree.



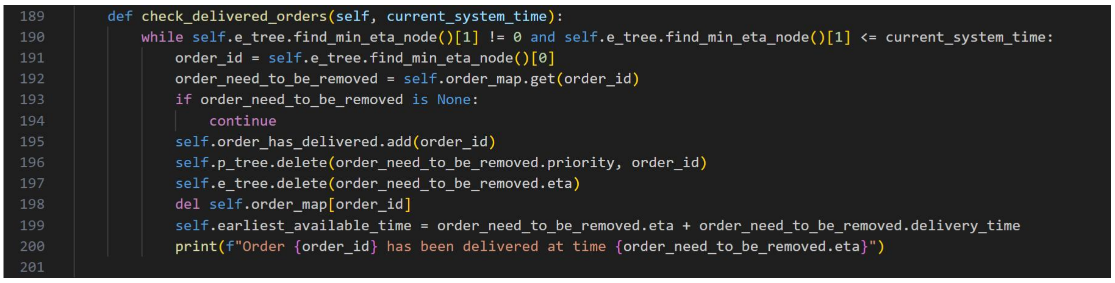
```python
def create_order(self, order_id, current_system_time, order_value, delivery_time):
    first_node = self.e_tree.find_min_eta_node()[0]
    first_eta = self.e_tree.find_min_eta_node()[1]
    if first_node != 0:
        if current_system_time >= first_eta and current_system_time < first_eta + self.order_map[
            first_node].delivery_time:
            first_p_node = self.p_tree.find_first_node()
            first_p_node.priority = float('inf')
            self.order_map[first_node].priority = float('inf')
        elif current_system_time < first_eta and current_system_time > first_eta - self.order_map[
            first_node].delivery_time:
            first_p_node = self.p_tree.find_first_node()
            first_p_node.priority = float('inf')
            self.order_map[first_node].priority = float('inf')

    order = Order(order_id, current_system_time, order_value, delivery_time)
    self.p_tree.insert(order.priority, order_id)
    previous_higher_priority_order = self.p_tree.find_previous_higher_priority(order.priority, order_id)

    if previous_higher_priority_order >= 0:
        prev = self.order_map[previous_higher_priority_order]
        order.eta = prev.eta + prev.delivery_time + order.delivery_time
    else:
        order.eta = order.current_system_time + order.delivery_time
```

5. cancel_order(order_id, current_system_time)-Cancels the specified order and updates the related data structure.



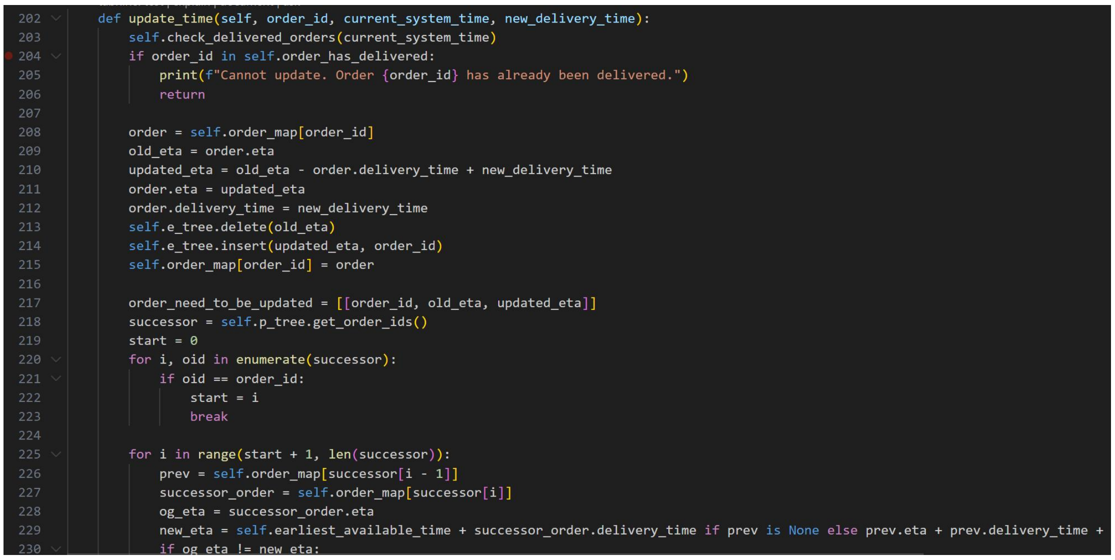
```python
def cancel_order(self, order_id, current_system_time):
    self.check_delivered_orders(current_system_time)
    if order_id in self.order_has_delivered:
        print(f"Cannot cancel. Order {order_id} has already been delivered.")
    else:
        orders = self.p_tree.get_order_ids()
        start = 0
        for i, oid in enumerate(orders):
            if oid == order_id:
                start = i
                break

        order_need_to_be_removed = self.order_map[order_id]
        self.p_tree.delete(order_need_to_be_removed.priority, order_id)
        self.e_tree.delete(order_need_to_be_removed.eta)
        del self.order_map[order_id]
        print(f"Order {order_id} has been cancelled")

        order_need_to_be_updated = []
        orders = self.p_tree.get_order_ids()
        for i in range(start, len(orders)):
            prev = self.order_map[orders[i - 1]] if i - 1 >= 0 else None
            successor_order = self.order_map[orders[i]]
            og_eta = successor_order.eta
            new_eta = self.earliest_available_time + successor_order.delivery_time if prev is None else prev.eta + prev.delivery_time
            if og_eta != new_eta:
                order_need_to_be_updated.append([orders[i], og_eta, new_eta])
                self.order_map[orders[i]].eta = new_eta

        order_need_to_be_updated.sort(key=lambda x: x[2])
        for order_update in order_need_to_be_updated:
            self.e_tree.delete(order_update[1])
            self.e_tree.insert(order_update[2], order_update[0])
```

6. check_delivered_orders(current_system_time)-Checks and processes orders that are due to be delivered before the current system time.



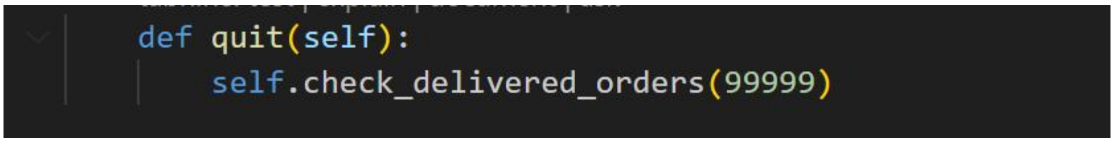
```python
def check_delivered_orders(self, current_system_time):
    while self.e_tree.find_min_eta_node()[1] != 0 and self.e_tree.find_min_eta_node()[1] <= current_system_time:
        order_id = self.e_tree.find_min_eta_node()[0]
        order_need_to_be_removed = self.order_map.get(order_id)
        if order_need_to_be_removed is None:
            continue
        self.order_has_delivered.add(order_id)
        self.p_tree.delete(order_need_to_be_removed.priority, order_id)
        self.e_tree.delete(order_need_to_be_removed.eta)
        del self.order_map[order_id]
        self.earliest_available_time = order_need_to_be_removed.eta + order_need_to_be_removed.delivery_time
        print(f"Order {order_id} has been delivered at time {order_need_to_be_removed.eta}")

```

7. update_time(order_id, current_system_time, new_delivery_time)-Updates the delivery time of the specified order and adjusts the order's ETA.



```python
def update_time(self, order_id, current_system_time, new_delivery_time):
    self.check_delivered_orders(current_system_time)
    if order_id in self.order_has_delivered:
        print(f"Cannot update. Order {order_id} has already been delivered.")
        return

    order = self.order_map[order_id]
    old_eta = order.eta
    updated_eta = old_eta - order.delivery_time + new_delivery_time
    order.eta = updated_eta
    order.delivery_time = new_delivery_time
    self.e_tree.delete(old_eta)
    self.e_tree.insert(updated_eta, order_id)
    self.order_map[order_id] = order

    order_need_to_be_updated = [[order_id, old_eta, updated_eta]]
    successor = self.p_tree.get_order_ids()
    start = 0
    for i, oid in enumerate(successor):
        if oid == order_id:
            start = i
            break

    for i in range(start + 1, len(successor)):
        prev = self.order_map[successor[i - 1]]
        successor_order = self.order_map[successor[i]]
        og_eta = successor_order.eta
        new_eta = self.earliest_available_time + successor_order.delivery_time if prev is None else prev.eta + prev.delivery_time +
        if og_eta != new_eta:
```

8.quit()-end order processing, usually used for system shutdown or end operation call.



```
def quit(self):
    self.check_delivered_orders(99999)
```

5. Performance Advantages

The use of AVL tree structures ensures efficient operations because they provide logarithmic time complexity in add, delete, and search operations. This efficiency is critical for order management systems that require fast response times, especially under high load.

6. Conclusion

The `OrderManagementSystem` class provides a powerful tool for managing and scheduling orders through its efficient data structures and methods. It ensures that order processing remains efficient and accurate even in complex and dynamically changing environments. This system is suitable for business scenarios that require precise control over order prioritization and delivery time

4.Order Class

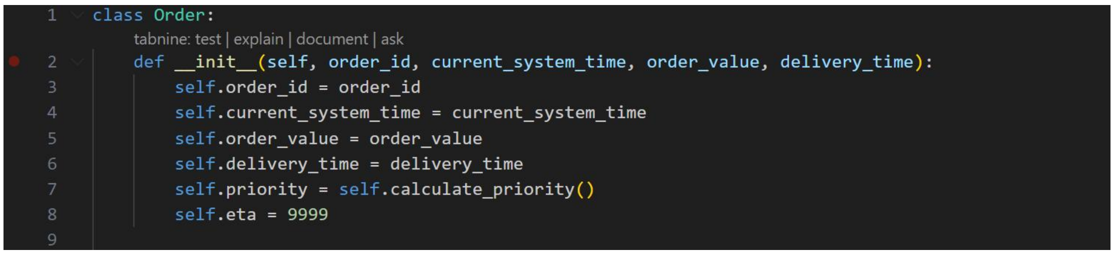
1.introduction

In an order management system, the processing and scheduling of each order often depends on its attributes such as priority and estimated time of arrival (ETA). The `Order` class plays a key role in this context by not only storing basic information about an order, but also calculating its priority and assigning an ETA to it.

2. Order Class Overview

The `Order` class provides a structured representation of each order in the order management system. It contains the order's identifier, timestamp, value, delivery time, and calculated priority and estimated arrival time.

3. Causality

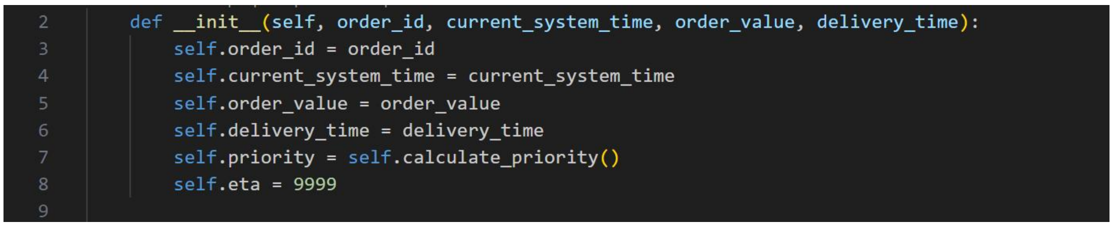


```python
class Order:
    def __init__(self, order_id, current_system_time, order_value, delivery_time):
        self.order_id = order_id
        self.current_system_time = current_system_time
        self.order_value = order_value
        self.delivery_time = delivery_time
        self.priority = self.calculate_priority()
        self.eta = 9999

```

`order_id`: The unique identifier of the order.
`current_system_time`: The system time when the order was created.
`order_value`: The value or amount of the order.
`delivery_time`: The estimated delivery time of the order.
`priority`: The priority calculated based on the order attributes.
`eta`: The estimated arrival time of the order.

4.Core functionality

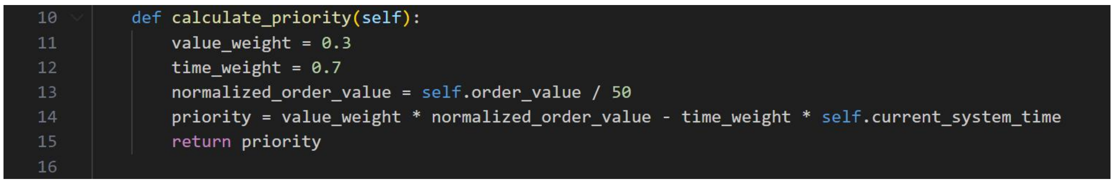
4.1 __init__(order_id, current_system_time, order_value, delivery_time)



```python
    def __init__(self, order_id, current_system_time, order_value, delivery_time):
        self.order_id = order_id
        self.current_system_time = current_system_time
        self.order_value = order_value
        self.delivery_time = delivery_time
        self.priority = self.calculate_priority()
        self.eta = 9999

```

The constructor initializes the order object, calculates and sets the order priority, with the initial ETA set to a high value (e.g., 9999) to indicate that no specific time has been assigned.

4.2 calculate_priority()

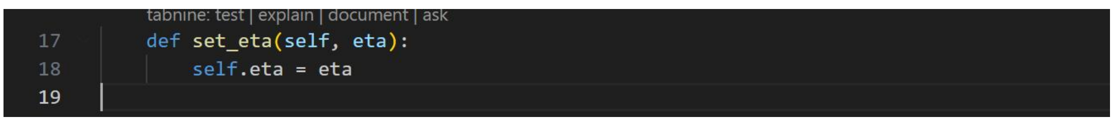


```python
    def calculate_priority(self):
        value_weight = 0.3
        time_weight = 0.7
        normalized_order_value = self.order_value / 50
        priority = value_weight * normalized_order_value - time_weight * self.current_system_time
        return priority

```

Calculates and returns the priority of an order. The priority is a weighted function based on the value of the order and the current system time, where the weights of value and time can be adjusted as needed to reflect different business logic.

4.3 set_eta(eta)

Set the Estimated Time of Arrival (ETA) for the order. This is usually done after the order has been inserted into the management system and the specific arrival time is determined based on the system logic.



```python
    def set_eta(self, eta):
        self.eta = eta

```

5.logic

The `Order` class is designed to take into account the business value and time sensitivity of orders. The method of prioritization reflects the basic principle that orders of high value and/or urgency should have a higher priority. This approach ensures that important orders are prioritized.

6. Conclusion

The `Order` class is the core component of the order management system, providing a structured representation of order data and priority management. It makes the order processing process both reliable and efficient, ensuring that each order is dispatched correctly according to business needs.

5.Test_case Class

1.Introduction

In order to ensure that the Order Management System (OMS) is working as expected, a system is needed that is capable of executing multiple test cases. This document describes a script for running test cases that simulates the operation of the Order Management System to verify its functionality and performance.

2.Class Overview

The test case execution system simulates operations on the order management system by reading an input file containing a series of commands. These commands include creating an order, canceling an order, updating the time of an order, getting the order ranking, and so on. The results of the script execution are written to the specified output file.

3.Core functionality

Read test case: reads a series of predefined commands and parameters from a text

file.
Execute commands: Execute the corresponding order management system operations according to the read commands and parameters.
Output results: Output the execution results to a specified file for result verification and analysis.

4.Testing process

1. Initialization: When the script starts execution, a new instance of `OrderManagementSystem` is created.
2. Command processing: the script reads the commands from the input file line by line and performs the corresponding actions.
3. result redirection: the script redirects the standard output to the output file, ensuring that all console output is captured.
4. Termination: When the `Quit` command is encountered, the script stops processing subsequent commands and exits.

5.Usage

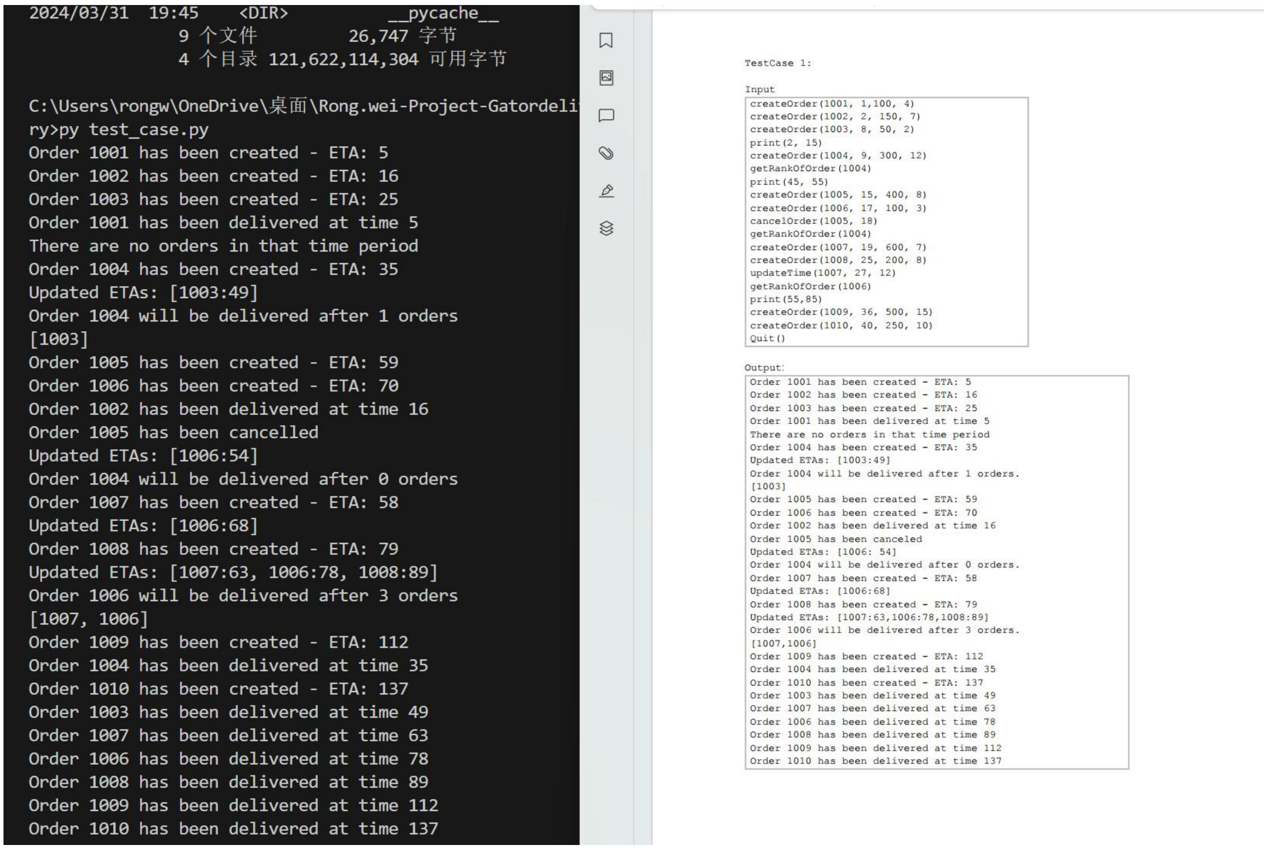
The script receives two parameters via the command line: an input filename and an output filename. Run the script using the following command:

```
python test_case.py input_filename.txt output_filename.txt
```

`input_filename.txt` should contain a series of test commands.
The `output_filename.txt` will contain the output of the executed commands.

6.Conclusion

Test1



```
2024/03/31  19:45    <DIR>          __pycache__
               9 个文件         26,747 字节
               4 个目录 121,622,114,304 可用字节

C:\Users\rongw\OneDrive\桌面\Rong.wei-Project-Gatordeli
ry>py test_case.py
Order 1001 has been created - ETA: 5
Order 1002 has been created - ETA: 16
Order 1003 has been created - ETA: 25
Order 1001 has been delivered at time 5
There are no orders in that time period
Order 1004 has been created - ETA: 35
Updated ETAs: [1003:49]
Order 1004 will be delivered after 1 orders
[1003]
Order 1005 has been created - ETA: 59
Order 1006 has been created - ETA: 70
Order 1002 has been delivered at time 16
Order 1005 has been cancelled
Updated ETAs: [1006:54]
Order 1004 will be delivered after 0 orders
Order 1007 has been created - ETA: 58
Updated ETAs: [1006:68]
Order 1008 has been created - ETA: 79
Updated ETAs: [1007:63, 1006:78, 1008:89]
Order 1006 will be delivered after 3 orders
[1007, 1006]
Order 1009 has been created - ETA: 112
Order 1004 has been delivered at time 35
Order 1010 has been created - ETA: 137
Order 1003 has been delivered at time 49
Order 1007 has been delivered at time 63
Order 1006 has been delivered at time 78
Order 1008 has been delivered at time 89
Order 1009 has been delivered at time 112
Order 1010 has been delivered at time 137
```

TestCase 1:

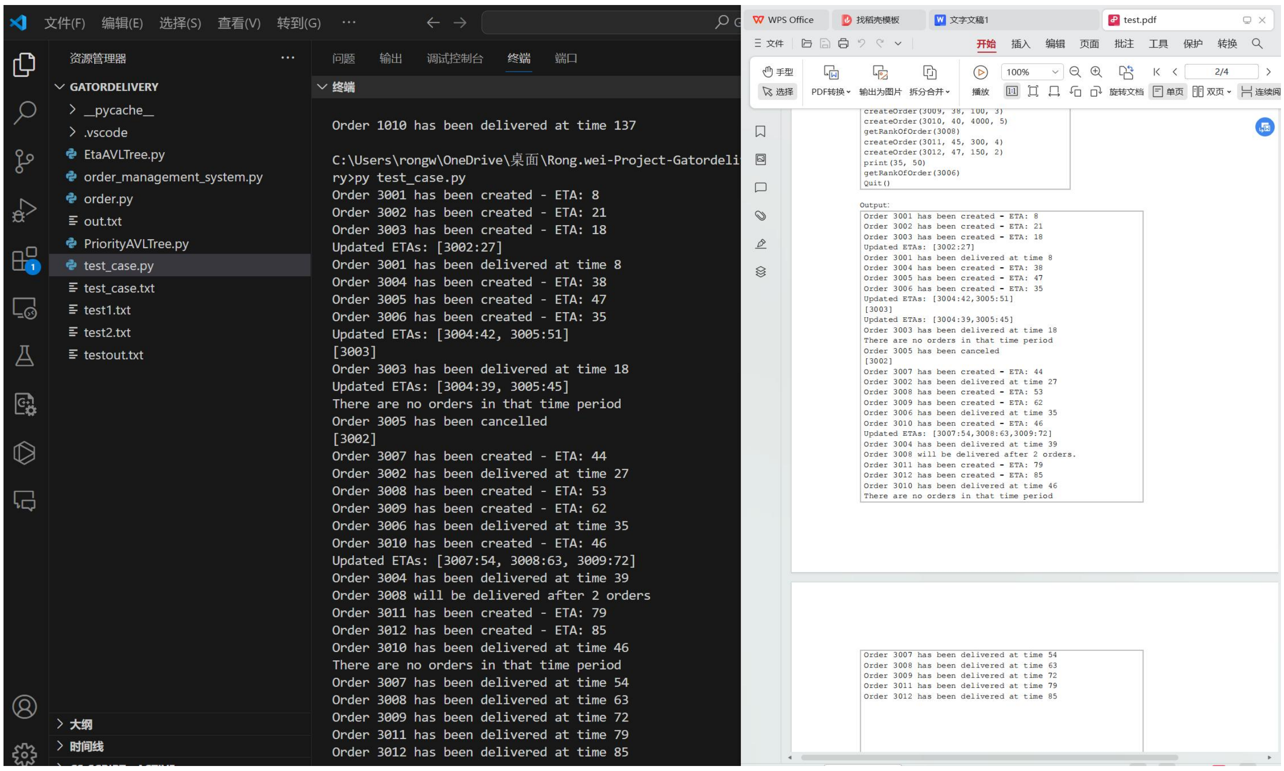
Input

```
createOrder(1001, 1,100, 4)
createOrder(1002, 2, 150, 7)
createOrder(1003, 8, 50, 2)
print(2, 15)
createOrder(1004, 9, 300, 12)
getRankOfOrder(1004)
print(45, 55)
createOrder(1005, 15, 400, 8)
createOrder(1006, 17, 100, 3)
cancelOrder(1005, 18)
getRankOfOrder(1004)
createOrder(1007, 19, 600, 7)
createOrder(1008, 25, 200, 8)
updateTime(1007, 27, 12)
getRankOfOrder(1006)
print(55,85)
createOrder(1009, 36, 500, 15)
createOrder(1010, 40, 250, 10)
Quit()
```

Output:

```
Order 1001 has been created - ETA: 5
Order 1002 has been created - ETA: 16
Order 1003 has been created - ETA: 25
Order 1001 has been delivered at time 5
There are no orders in that time period
Order 1004 has been created - ETA: 35
Updated ETAs: [1003:49]
Order 1004 will be delivered after 1 orders.
[1003]
Order 1005 has been created - ETA: 59
Order 1006 has been created - ETA: 70
Order 1002 has been delivered at time 16
Order 1005 has been canceled
Updated ETAs: [1006: 54]
Order 1004 will be delivered after 0 orders.
Order 1007 has been created - ETA: 58
Updated ETAs: [1006:68]
Order 1008 has been created - ETA: 79
Updated ETAs: [1007:63,1006:78,1008:89]
Order 1006 will be delivered after 3 orders.
[1007,1006]
Order 1009 has been created - ETA: 112
Order 1004 has been delivered at time 35
Order 1010 has been created - ETA: 137
Order 1003 has been delivered at time 49
Order 1007 has been delivered at time 63
Order 1006 has been delivered at time 78
Order 1008 has been delivered at time 89
Order 1009 has been delivered at time 112
Order 1010 has been delivered at time 137
```

Test2


文件(F) 编辑(E) 选择(S) 查看(V) 转到(G) ···
资源管理器
GATORDELIVERY
__pycache__
.vscode
EtaAVLTree.py
order_management_system.py
order.py
out.txt
PriorityAVLTree.py
test_case.py
test_case.txt
test1.txt
test2.txt
testout.txt
大纲
时间线
问题 输出 调试控制台 终端 端口
终端
WPS Office
找稻壳模板
文字文稿1
test.pdf
文件
开始 插入 编辑 页面 批注 工具 保护 转换
手型 选择 PDF转换 输出为图片 拆分合并 播放 100% 旋转文档 单页 双页 连续阅
2/4

```
Order 1010 has been delivered at time 137

C:\Users\rongw\OneDrive\桌面\Rong.wei-Project-Gatordeli
ry>py test_case.py
Order 3001 has been created - ETA: 8
Order 3002 has been created - ETA: 21
Order 3003 has been created - ETA: 18
Updated ETAs: [3002:27]
Order 3001 has been delivered at time 8
Order 3004 has been created - ETA: 38
Order 3005 has been created - ETA: 47
Order 3006 has been created - ETA: 35
Updated ETAs: [3004:42, 3005:51]
[3003]
Order 3003 has been delivered at time 18
Updated ETAs: [3004:39, 3005:45]
There are no orders in that time period
Order 3005 has been cancelled
[3002]
Order 3007 has been created - ETA: 44
Order 3002 has been delivered at time 27
Order 3008 has been created - ETA: 53
Order 3009 has been created - ETA: 62
Order 3006 has been delivered at time 35
Order 3010 has been created - ETA: 46
Updated ETAs: [3007:54, 3008:63, 3009:72]
Order 3004 has been delivered at time 39
Order 3008 will be delivered after 2 orders
Order 3011 has been created - ETA: 79
Order 3012 has been created - ETA: 85
Order 3010 has been delivered at time 46
There are no orders in that time period
Order 3007 has been delivered at time 54
Order 3008 has been delivered at time 63
Order 3009 has been delivered at time 72
Order 3011 has been delivered at time 79
Order 3012 has been delivered at time 85
```

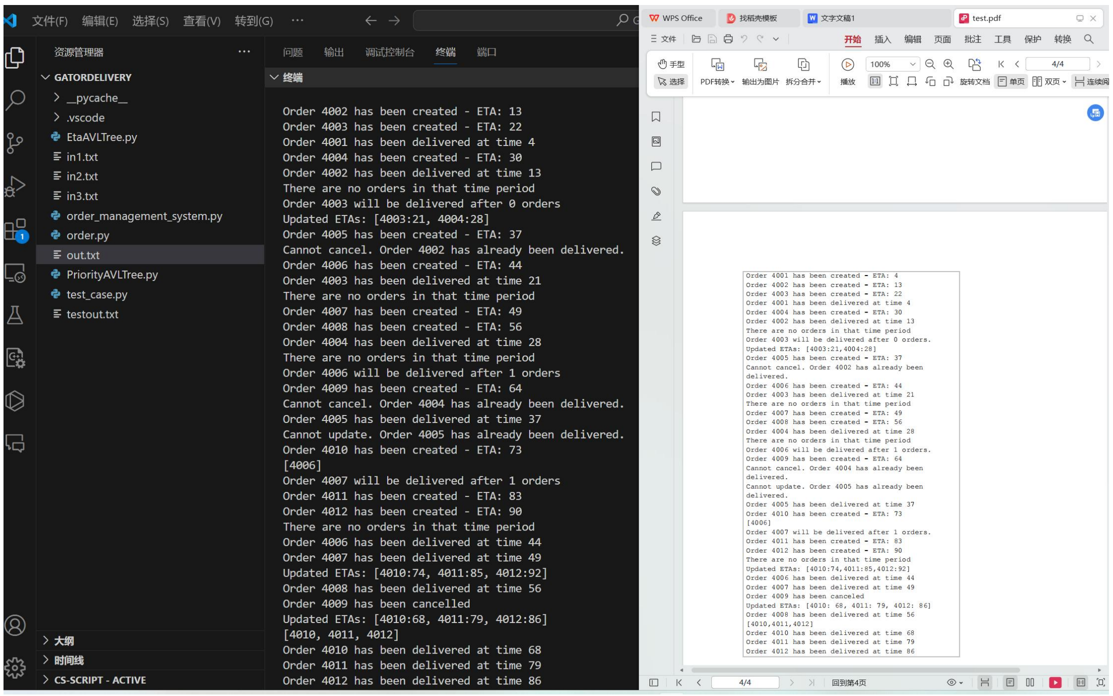
```
createOrder(3009, 38, 100, 3)
createOrder(3010, 40, 4000, 5)
getRankOfOrder(3008)
createOrder(3011, 45, 300, 4)
createOrder(3012, 47, 150, 2)
print(35, 50)
getRankOfOrder(3006)
Quit()
```

Output:

```
Order 3001 has been created - ETA: 8
Order 3002 has been created - ETA: 21
Order 3003 has been created - ETA: 18
Updated ETAs: [3002:27]
Order 3001 has been delivered at time 8
Order 3004 has been created - ETA: 38
Order 3005 has been created - ETA: 47
Order 3006 has been created - ETA: 35
Updated ETAs: [3004:42,3005:51]
[3003]
Updated ETAs: [3004:39,3005:45]
Order 3003 has been delivered at time 18
There are no orders in that time period
Order 3005 has been canceled
[3002]
Order 3007 has been created - ETA: 44
Order 3002 has been delivered at time 27
Order 3008 has been created - ETA: 53
Order 3009 has been created - ETA: 62
Order 3006 has been delivered at time 35
Order 3010 has been created - ETA: 46
Updated ETAs: [3007:54,3008:63,3009:72]
Order 3004 has been delivered at time 39
Order 3008 will be delivered after 2 orders.
Order 3011 has been created - ETA: 79
Order 3012 has been created - ETA: 85
Order 3010 has been delivered at time 46
There are no orders in that time period
```

```
Order 3007 has been delivered at time 54
Order 3008 has been delivered at time 63
Order 3009 has been delivered at time 72
Order 3011 has been delivered at time 79
Order 3012 has been delivered at time 85
```

Test3

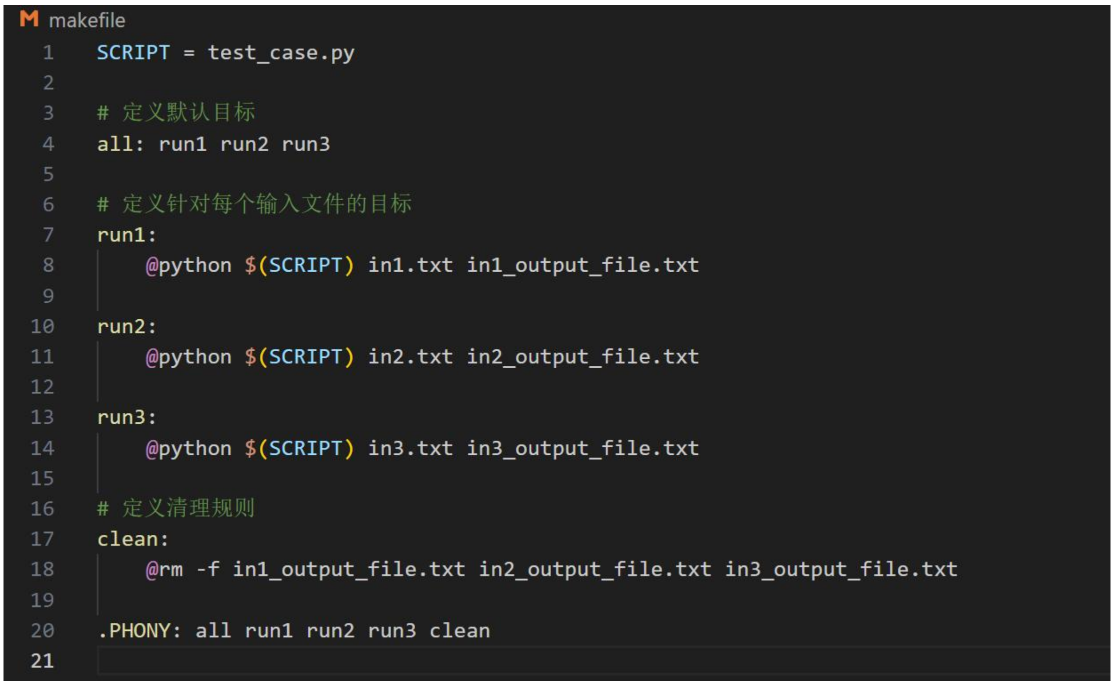


```
Order 4002 has been created - ETA: 13
Order 4003 has been created - ETA: 22
Order 4001 has been delivered at time 4
Order 4004 has been created - ETA: 30
Order 4002 has been delivered at time 13
There are no orders in that time period
Order 4003 will be delivered after 0 orders
Updated ETAs: [4003:21, 4004:28]
Order 4005 has been created - ETA: 37
Cannot cancel. Order 4002 has already been delivered.
Order 4006 has been created - ETA: 44
Order 4003 has been delivered at time 21
There are no orders in that time period
Order 4007 has been created - ETA: 49
Order 4008 has been created - ETA: 56
Order 4004 has been delivered at time 28
There are no orders in that time period
Order 4006 will be delivered after 1 orders
Order 4009 has been created - ETA: 64
Cannot cancel. Order 4004 has already been delivered.
Order 4005 has been delivered at time 37
Cannot update. Order 4005 has already been delivered.
Order 4010 has been created - ETA: 73
[4006]
Order 4007 will be delivered after 1 orders
Order 4011 has been created - ETA: 83
Order 4012 has been created - ETA: 90
There are no orders in that time period
Order 4006 has been delivered at time 44
Order 4007 has been delivered at time 49
Updated ETAs: [4010:74, 4011:85, 4012:92]
Order 4008 has been delivered at time 56
Order 4009 has been cancelled
Updated ETAs: [4010:68, 4011:79, 4012:86]
[4010, 4011, 4012]
Order 4010 has been delivered at time 68
Order 4011 has been delivered at time 79
Order 4012 has been delivered at time 86
```

```
Order 4001 has been created - ETA: 4
Order 4002 has been created - ETA: 13
Order 4003 has been created - ETA: 22
Order 4001 has been delivered at time 4
Order 4004 has been created - ETA: 30
Order 4002 has been delivered at time 13
There are no orders in that time period
Order 4003 will be delivered after 0 orders.
Updated ETAs: [4003:21,4004:28]
Order 4005 has been created - ETA: 37
Cannot cancel. Order 4002 has already been
delivered.
Order 4006 has been created - ETA: 44
Order 4003 has been delivered at time 21
There are no orders in that time period
Order 4007 has been created - ETA: 49
Order 4008 has been created - ETA: 56
Order 4004 has been delivered at time 28
There are no orders in that time period
Order 4006 will be delivered after 1 orders.
Order 4009 has been created - ETA: 64
Cannot cancel. Order 4004 has already been
delivered.
Cannot update. Order 4005 has already been
delivered.
Order 4005 has been delivered at time 37
Order 4010 has been created - ETA: 73
[4006]
Order 4007 will be delivered after 1 orders.
Order 4011 has been created - ETA: 83
Order 4012 has been created - ETA: 90
There are no orders in that time period
Updated ETAs: [4010:74,4011:85,4012:92]
Order 4006 has been delivered at time 44
Order 4007 has been delivered at time 49
Order 4009 has been canceled
Updated ETAs: [4010: 68, 4011: 79, 4012: 86]
Order 4008 has been delivered at time 56
[4010,4011,4012]
Order 4010 has been delivered at time 68
Order 4011 has been delivered at time 79
Order 4012 has been delivered at time 86
```

The Test Case Execution System provides a reliable testing platform for the order management system. It can simulate complex operation scenarios, verify system functionality, and ensure that the system processes orders correctly according to business logic. Through the automated testing process, potential problems can be effectively identified and fixed, thus improving the stability and reliability of the system.

6.Makefile



makefile

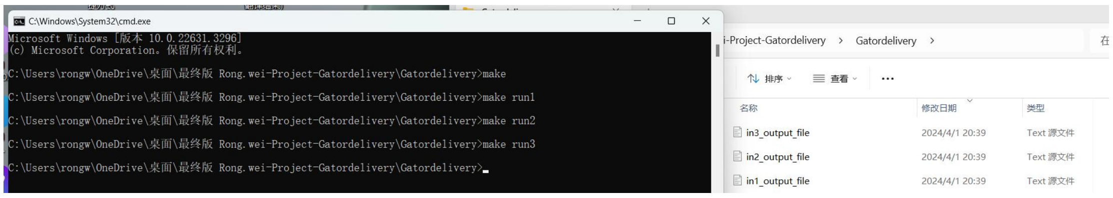
```makefile
SCRIPT = test_case.py

# 定义默认目标
all: run1 run2 run3

# 定义针对每个输入文件的目标
run1:
	@python $(SCRIPT) in1.txt in1_output_file.txt

run2:
	@python $(SCRIPT) in2.txt in2_output_file.txt

run3:
	@python $(SCRIPT) in3.txt in3_output_file.txt

# 定义清理规则
clean:
	@rm -f in1_output_file.txt in2_output_file.txt in3_output_file.txt

.PHONY: all run1 run2 run3 clean
```

Using make

Use make run1 to get result 1.

Use make run2 to get result 2.

Use make run3 to get result 2.



C:\Windows\System32\cmd.exe

```
Microsoft Windows [版本 10.0.22631.3296]
(c) Microsoft Corporation。保留所有权利。

C:\Users\rongw\OneDrive\桌面\最终版 Rong.wei-Project-Gatordelivery\Gatordelivery>make

C:\Users\rongw\OneDrive\桌面\最终版 Rong.wei-Project-Gatordelivery\Gatordelivery>make run1

C:\Users\rongw\OneDrive\桌面\最终版 Rong.wei-Project-Gatordelivery\Gatordelivery>make run2

C:\Users\rongw\OneDrive\桌面\最终版 Rong.wei-Project-Gatordelivery\Gatordelivery>make run3

C:\Users\rongw\OneDrive\桌面\最终版 Rong.wei-Project-Gatordelivery\Gatordelivery>
```

i-Project-Gatordelivery > Gatordelivery >

名称	修改日期	类型
in3_output_file	2024/4/1 20:39	Text 源文件
in2_output_file	2024/4/1 20:39	Text 源文件
in1_output_file	2024/4/1 20:39	Text 源文件