

Project：Real Estate Property Management System

Course：Data Structure Curriculum Design

Student Name: Peng Chuhan

Zhang Mingyang

Ma Sixi

Nie Xiaoya

Student Number： 2023310997

2023310982

2023310996

2023310785

Faculty Adviser： Lilia SFAXI

**Abstract**

**This project presents the design and implementation of an intelligent real estate agency management system using Python. The system adopts a modular, object-oriented architecture, featuring core modules for client and property management, advanced data structures, and a user-friendly graphical interface built with PyQt. A key highlight of the system is its multi-dimensional property recommendation algorithm, which evaluates and ranks properties for clients based on preferences such as location, features, and price proximity. Data persistence is achieved through CSV files, ensuring easy data management and scalability. Comprehensive unit tests are provided to guarantee the correctness and robustness of all major functionalities. The system enables efficient property search, intelligent matching, and streamlined management for both agents and clients, demonstrating the practical application of algorithmic techniques in real-world real estate scenarios.**

**Catalogue**

[I Description Of Project 5](#_Toc202522203)

[1.1 Project Overview 5](#_Toc202522204)

[1.2 Core Features 5](#_Toc202522205)

[1.3 Technical Architecture 7](#_Toc202522206)

[1.4 Data Structures and Files 7](#_Toc202522207)

[1.5. Running and Deployment 7](#_Toc202522208)

[1.6 Highlights 8](#_Toc202522209)

[1.7Application Scenarios 8](#_Toc202522210)

[II Requirement Analysis of Project 8](#_Toc202522211)

[2.1 Background and Purpose 8](#_Toc202522212)

[2.2 Functional Requirements 8](#_Toc202522213)

[2.3 Non-Functional Requirements 11](#_Toc202522214)

[2.4Data Requirements 11](#_Toc202522215)

[2.5 System Workflow (Summary) 12](#_Toc202522216)

[2.6 Future Extensions 13](#_Toc202522217)

[III. Design of Project 13](#_Toc202522218)

[3.1 System Overview 13](#_Toc202522219)

[3.2 Module Design 13](#_Toc202522220)

[3.3 Intelligent Property Matching Algorithm 14](#_Toc202522221)

[3.4 Data Flow and Interaction 15](#_Toc202522222)

[IV. Implementation of Project 16](#_Toc202522223)

[4.1 Technology Stack 16](#_Toc202522224)

[4.2 Model Implementation 16](#_Toc202522225)

[4.3 Business Logic 16](#_Toc202522226)

[4.4 Algorithm Integration 16](#_Toc202522227)

[4.5 User Interface 17](#_Toc202522228)

[4.6 Data Management 17](#_Toc202522229)

[4.7 Testing 17](#_Toc202522230)

[4.8 Extensibility 17](#_Toc202522231)

[V Running and Debugging of Project 17](#_Toc202522232)

[VI Summary 18](#_Toc202522233)

[6.1 Daily Work Schedule 18](#_Toc202522234)

[6.2 Division of Labor within the Group 19](#_Toc202522235)

[6.3 Tips 19](#_Toc202522236)

[6.4 Difficulties and Solution of Problems 21](#_Toc202522237)

## I Description Of Project

## 1.1 Project Overview

This project is a Real Estate Intelligent Management System designed for real estate agencies and property information platforms. It provides an efficient, intelligent, and extensible solution for managing and matching property and client information. The system is developed in Python 3, leveraging classic data structures (AVL tree, queue), object-oriented design, batch data import, dynamic pricing algorithms, and a graphical user interface (PyQt5) to automate and optimize the entire property-client workflow.

## 1.2 Core Features

1.2.1 Property Information Management

Properties are stored in an AVL tree for efficient CRUD operations and range queries.

Each property includes property\_ID, address, price, property\_type (e.g., HOUSE, APARTMENT, COMMERCIAL, LAND), status (AVAILABLE/SOLD), and owner.

Supports batch import from CSV files (e.g., real\_estate\_properties\_dataset.csv), automatically recognizing property status and owner.

Properties can be marked as SOLD or AVAILABLE, with owner information recorded.

1.2.2 Client Information Management

Clients are managed using a queue (ClientQueue) to ensure FIFO processing of client requests.

Each client includes client\_ID, name, contact\_info, and budget. The system can be extended to support property\_type preferences.

Supports importing client data from CSV files (e.g., client\_requests\_dataset.csv), compatible with or without the property\_type field.

The system supports automatic processing, querying, and deletion of client requests.

1.2.3 Intelligent Property Matching

Matches properties to clients based on budget, property type, and other criteria, recommending the most suitable properties.

Supports one client matching multiple properties and one property being matched to multiple clients.

Matching details are shown in the command line or GUI, including client info and matched property lists.

1.2.4 Dynamic Pricing Model

The system automatically tracks each property's views and client inquiries.

If views or inquiries exceed a threshold, the property price is increased (e.g., +5%).

If views and inquiries are below a threshold, the price is decreased (e.g., -3%).

Statistics are reset after each price adjustment.

Price adjustments can be triggered manually by an admin or automatically after processing a batch of clients.

1.2.5 Graphical User Interface (GUI)

Built with PyQt5, supporting cross-platform operation.

Main screens include client information management, property information management, matching results display, logs and operation prompts, and one-click client request processing and data refresh.

Enables efficient client processing, property purchase, and data refresh via the GUI.

1.2.6 Batch Data Import/Export

Supports batch importing of property and client data from CSV files, with strict field order matching the code.

Client data import works with or without the property\_type field; property data allows owner to be empty.

Clear error messages for missing fields or type errors, ensuring robust data import.

1.2.7 Unit Testing

Includes property management, client management, matching algorithms, dynamic pricing, and data import.

Standardized test datasets ensure reproducibility.

## 1.3 Technical Architecture

Language: Python 3.7+

Main Dependency: PyQt5

Core Modules:

real\_estate.managers: Property and client managers

real\_estate.models: Data models for property, client, etc.

real\_estate.structures: Data structures such as AVL tree and queue

real\_estate.utils: Data loading and utility functions

real\_estate.gui: Graphical interface

tests: Unit tests

## 1.4 Data Structures and Files

Property Data File (real\_estate\_properties\_dataset.csv): Contains property\_ID, address, price, property\_type, status, owner; supports batch import.

Client Data File (client\_requests\_dataset.csv): Contains client\_ID, name, contact\_info, budget; supports batch import and is compatible with or without property\_type.

Data Structures:

AVL Tree: Efficient storage and retrieval of property information, supports range queries.

Queue: Manages client requests, ensuring FIFO processing.

## 1.5. Running and Deployment

Install dependencies: pip install PyQt5

Prepare datasets (CSV format, fields must match code)

Run the main program: python main.py

Use the GUI or command line for property/client management, matching, and transactions

## 1.6 Highlights

Combines classic data structures with modern GUI technology for both performance and usability.

Dynamic pricing and intelligent matching algorithms improve property operation efficiency.

Supports batch data import for real-world business scenarios.

Clear code structure, easy to extend and maintain.

Robust error handling and unit testing ensure system stability.

## 1.7Application Scenarios

Real estate agency information management

Property information platform with intelligent recommendations

Real estate big data analysis and automated operations

# II Requirement Analysis of Project

## 2.1 Background and Purpose

With the rapid development of the real estate industry and the increasing demand for digital management, real estate agencies and property platforms require an efficient, intelligent, and user-friendly system to manage properties and clients, automate matching, and support dynamic pricing. This project aims to build a Python-based system that integrates classic data structures, batch data import, intelligent matching, dynamic pricing, and a graphical user interface to streamline real estate business operations.

## 2.2 Functional Requirements

2.2.1 Property Information Management

* Fields:

Each property record must include:

property\_ID (unique identifier, integer)

address (string)

price (float)

property\_type (enum: HOUSE, APARTMENT, COMMERCIAL, LAND)

status (enum: AVAILABLE, SOLD)

owner (string, can be empty if AVAILABLE)

* Operations:

Add, delete, update, and query property records.

Query by property\_ID, type, price range, status, or owner.

Import property data in bulk from CSV files (see real\_estate\_properties\_dataset.csv).

Store and manage property data using an AVL tree for efficient search and range queries.

Update property status and owner after a transaction.

2.2.2 Client Information Management

* Fields:

Each client record must include:

client\_ID (unique identifier, integer)

name (string)

contact\_info (string)

budget (float)

(Optional) property\_type (enum, if present in CSV)

* Operations:

Add, delete, update, and query client records.

Query by client\_ID, name, or budget.

Import client data in bulk from CSV files (see client\_requests\_dataset.csv), compatible with or without the property\_type field.

Store and manage client data using a queue (ClientQueue) to ensure FIFO processing of requests.

2.2.3 Intelligent Matching

* Matching Algorithm:

For each client, automatically find all AVAILABLE properties that match the client’s budget and (if specified) property\_type.

Support one-to-many and many-to-many matching scenarios.

Display matching results in the GUI and/or command line, including all relevant property and client details.

* Transaction Handling:

Allow the client to purchase a matched property, updating the property’s status to SOLD and recording the owner.

Remove the client from the queue after a successful transaction or if no match is found.

2.2.4 Dynamic Pricing

* Demand Tracking:

Track views and inquiries for each property.

If a property’s views/inquiries exceed a threshold, automatically increase its price (e.g., by 5%).

If a property’s views/inquiries are below a threshold, automatically decrease its price (e.g., by 3%).

Reset statistics after each price adjustment.

Allow manual or automatic triggering of price adjustments.

2.5 Batch Data Import and Export

* CSV Import:

Support batch import of property and client data from CSV files.

Validate data types and required fields; provide clear error messages for missing or invalid data.

Support optional fields (e.g., property\_type for clients, owner for properties).

* CSV Export (Optional):

Support exporting current property and client data to CSV for backup or reporting.

2.6 Graphical User Interface (GUI)

* Technology:

Use PyQt5 for cross-platform GUI development.

* Features:

Display and manage property and client information.

Show matching results and transaction logs.

Provide buttons for processing client requests, refreshing data, and triggering price adjustments.

Support batch data import via GUI.

Display error and status messages to the user.

2.7 Unit Testing

* Coverage:

Provide unit tests for all core modules: property management, client management, matching, dynamic pricing, and data import.

Use standard test datasets to ensure reproducibility and reliability.

## 2.3 Non-Functional Requirements

* Performance:

The system should efficiently handle thousands of property and client records.

Matching and price adjustment operations should complete within 1 second for typical datasets.

* Usability:

The GUI should be intuitive and responsive.

Error messages and logs should be clear and helpful.

* Extensibility:

The codebase should be modular and well-documented, allowing for easy addition of new features (e.g., new property types, advanced matching rules).

* Compatibility:

The system must run on Windows (Python 3.7+) and support cross-platform deployment.

* Reliability:

Robust error handling for data import, user operations, and system exceptions.

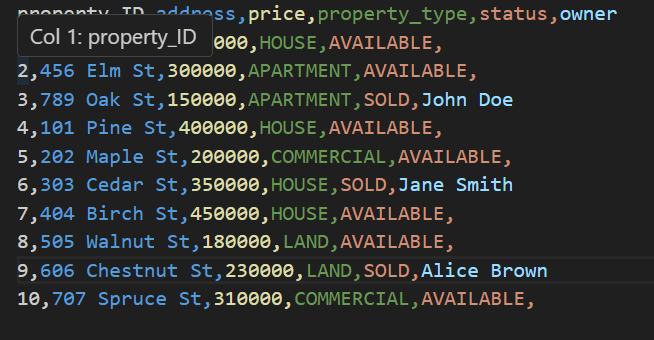
Comprehensive unit tests to ensure system stability.

## 2.4Data Requirements

Property Data:

Must be provided in CSV format with fields: property\_ID, address, price, property\_type, status, owner.

Example:



Client Data:

Must be provided in CSV format with fields: client\_ID, name, contact\_info, budget (and optionally property\_type).

Example:



## 2.5 System Workflow (Summary)

* Data Import:  
  Admin imports property and client data from CSV files.
* Client Request Processing:

The system processes client requests in FIFO order, matches suitable properties, and displays results.

* Transaction Handling:  
  If a client purchases a property, the system updates the property status and removes the client from the queue.
* Dynamic Pricing:  
  The system adjusts property prices based on demand statistics.
* GUI Interaction:  
  Users manage data, process requests, and view logs via the PyQt5 GUI.
* Testing:  
  Developers run unit tests to verify system correctness.

## 2.6 Future Extensions

Integration with External Systems: Integrate the real estate management system with external systems for enhanced functionality. For instance, implement a functionality to retrieve property details from online real estate databases or send automated notifications to clients about new listings and price changes.

# III. Design of Project

## 3.1 System Overview

The Real Estate Agency System is designed as a modular, extensible, and maintainable application using Python. The system is structured into several key components, each responsible for a specific aspect of the business logic or user interaction. The main modules include Models, Managers, Data Structures, Utilities, and a Graphical User Interface (GUI).

## 3.2 Module Design

3.2.1 Models

* Client: Represents a customer, storing personal information, preferred neighborhoods, preferred property features, and budget. The class is designed to be easily extensible for additional preferences.
* Property: Represents a real estate property, including attributes such as ID, address, price, type, status, owner, and a list of features. The model supports flexible feature storage to accommodate various property characteristics.
* Enumerations: Property types and statuses are defined using enums for clarity and type safety.

3.2.2 Managers

* ClientManager: Handles client-related operations, including registration, updating preferences, and advanced property matching. The intelligent matching algorithm is implemented here.
* PropertyManager: Manages property CRUD operations, searching, and status updates. It ensures data integrity and supports complex search queries.

3.2.3 Data Structures

* AVL Tree: Used for efficient storage and retrieval of properties or clients, ensuring balanced search times.
* Queue: Manages client requests in a first-in, first-out manner, supporting fair and organized processing.

3.2.4 Utilities

* Loader: Handles data import and export from CSV files, enabling persistent storage and easy data manipulation.

3.2.5 GUI

* PyQt Interface: Provides an interactive and user-friendly interface for agents and clients. Users can search for properties, view details, and receive intelligent recommendations.

## 3.3 Intelligent Property Matching Algorithm

A core feature of the system is the multi-dimensional property recommendation algorithm, implemented in the ClientManager. The algorithm evaluates each property for a given client based on several weighted criteria:

* Location Match: Checks if the property's neighborhood matches any of the client's preferred neighborhoods.
* Feature Match: Counts the number of overlapping features between the property and the client's preferences.
* Price Proximity: Calculates a score based on how close the property's price is to the client's budget.
* Type Match: Optionally considers the client's preferred property type.

Each criterion is assigned a configurable weight. The total score for each property is computed as a weighted sum of all criteria. Properties are then ranked by their scores, and the top N properties are recommended to the client.

Algorithm Pseudocode:



## 3.4 Data Flow and Interaction

* Data is loaded from CSV files into model objects at startup.
* User actions in the GUI (such as searching or requesting recommendations) trigger manager methods.
* Search and recommendation results are displayed in the GUI.
* All changes can be saved back to CSV files for persistence.

# IV. Implementation of Project

## 4.1 Technology Stack

* Programming Language: Python 3
* GUI Framework: PyQt5
* Data Storage: CSV files for persistent storage of clients and properties
* Testing: Python’s unittest framework for comprehensive unit testing

## 4.2 Model Implementation

* The Client and Property classes are implemented in the models module, with support for extensible attributes and easy serialization/deserialization from CSV.
* Enumerations for property type and status ensure data consistency.

## 4.3 Business Logic

* The PropertyManager provides methods for adding, removing, updating, and searching properties. It enforces business rules, such as preventing the sale of a property without an owner.
* The ClientManager manages client data and implements the advanced property matching algorithm, allowing for personalized recommendations.

## 4.4 Algorithm Integration

* The intelligent matching algorithm is fully integrated into the ClientManager and exposed to the GUI. When a user requests recommendations, the system computes scores for all available properties and returns the best matches.
* The algorithm is designed to be easily adjustable, with weights for each criterion configurable for different business needs.

## 4.5 User Interface

* The PyQt-based GUI provides forms and tables for property and client management, as well as buttons for triggering intelligent recommendations.
* The interface is designed for ease of use, with clear navigation and real-time feedback.

## 4.6 Data Management

* Data is loaded from CSV files at startup and can be saved back at any time, ensuring that all changes are persistent.
* The loader utility module abstracts the details of file I/O, making it easy to switch to a different storage backend in the future if needed.

## 4.7 Testing

* The tests directory contains unit tests for all major modules, including models, managers, and utility functions.
* Tests cover normal operations, edge cases, and error handling, ensuring the reliability and correctness of the system.

## 4.8 Extensibility

* The modular design allows for easy addition of new features, such as more advanced matching criteria, new data sources, or integration with web services.
* The codebase follows best practices for readability and maintainability, with clear separation of concerns.

# V Running and Debugging of Project

1.All of our tests passed, regardless of whether they were python tests or unittest tests . The following is an incomplete display.

**文本

AI 生成的内容可能不正确。日程表

AI 生成的内容可能不正确。**

* Environment Dependency Confirmation

We make sure to install the required libraries such as Python 3.7+, PyQt5, matplotlib, etc. Install via pip install pyqt5 matplotlib.

* Debugging of core modules

We mainly test whether the structure of avltree is correct, adding many edge tests to it to check if it is balanced, etc. Secondly, we mainly test the Property class methods and the price status of real estate, etc.

2. Test Code Coverage

图形用户界面

AI 生成的内容可能不正确。

Almost 95% code coverage

# VI Summary

## 6.1 Daily Work Schedule

From Monday to Friday 14：00-20：00

1.Week1

Day 1 – Project Initialization & Core Models

* Designed the overall project structure and module breakdown (models, structures, managers, utils, gui, tests)
* Implemented core data models:
  + Client class with attributes like client\_ID, name, contact info, budget, and property preferences
  + Property class with enums for PropertyType and PropertyStatus, and methods like \_\_repr\_\_, \_\_eq\_\_, and \_\_lt\_\_ for AVL Tree comparison
* Wrote unit tests to verify basic instantiation and behavior

Day 2 – Data Structures Implementation & Visualization

* Built the AVLTree class with support for insertion, deletion, balancing, and searching
* Implemented ClientQueue as a linked list supporting enqueue and dequeue in O(1) time
* Started visualizing AVL tree nodes using TreeNodeItem and integrated it into a PyQt5 graphics view
* Added unit tests for AVL and Queue operations

Day 3 – Business Logic Managers

* Developed PropertyManager and ClientManager:
  + Handled operations like adding/removing properties, matching clients, and buying properties
  + Ensured data consistency via centralized management
* Wrote tests to validate core business logic (matching, ownership assignment, etc.)

Day 4 – Dataset Integration

* Built the load\_dataset utility function to load property and client data from CSV files
* Used the loader to initialize the AVL Tree and Client Queue with real data
* Verified data loading with additional unit tests

Day 5 – Debugging, Testing & Midterm Review

* Integrated and debugged all modules together
* Enhanced test coverage by adding edge case scenarios (duplicate IDs, empty data, invalid inputs)
* Prepared materials for midterm checkpoint presentation

2.week2

Day 6 – Refactoring Based on Feedback

* Addressed instructor feedback by:
  + Eliminating redundant data in PropertyManager (removed internal dictionary)
  + Rewriting client queue search/removal logic to avoid inefficient dequeue-enqueue cycles
  + Centralizing all interactions via manager classes for better modularity

Day 7 – Advanced Algorithm Implementation

* Added boundary and stress tests for edge cases
* Implemented Part 2 core logic:
  + Dynamic Pricing: Adjusts prices based on property view counts
  + Optimized Matching: Uses scoring algorithm to rank and match properties based on budget and type

Day 8 – GUI Development & Enhancements

* Designed a fully functional GUI using PyQt5:
  + Included client queue, property table, logs, search, and matching buttons
  + Visualized the AVL Tree dynamically based on current property state
* Improved user experience and interaction design
* Added analytic dashboards using Matplotlib (e.g., property type distribution, transaction rates)

Day 9 – Documentation & Final Polish

* Wrote the final project report with sections on:
  + Use cases
  + Data structure selection and justification
  + Core algorithm descriptions
  + Testing and validation
* Finalized PowerPoint slides and generated test coverage reports
* Polished code, refactored unclear parts, and added comments where needed

Day 10 – Final Presentation

* Performed a full demo of the application
* Showcased:
  + GUI features
  + Core algorithm flows
  + Code optimizations
  + Test coverage and results
* Delivered a well-organized final presentation and responded to Q&A

## 6.2 Division of Labor within the Group

Pengchuhan：

1.Design and implement core data structures such as AVL trees and queues (real\_estate.structures)

2. Realize functions such as adding, deleting, querying, modifying properties and customers, as well as queue management

3.Responsible for PowerPoint writing, requirements analysis and use case analysis

4.Responsible for organizing unit tests and writing the main test codes (tests)

5.Be responsible for the overall integration, deployment and final debugging of the project

Zhangmingyang：

1.Design and implement data models for real estate, customers, etc. (real\_estate.models)

2.Responsible for the batch import and export functions of CSV data (real\_estate.utils.loader)

3.Realize functions such as data verification, exception handling, and log output

4.Participate in the supplementation and improvement of unit test cases

5.Responsible for writing reports

Niexiaoya：

1.Implement the property manager and customer manager (real\_estate.managers)

2.Design and implement intelligent matching and dynamic pricing algorithms

3.Assist the team leader in implementing some core algorithms

4.Participate in system integration, joint debugging and BUG fixing

5.Responsible for writing reports

Masixi：

1.Design and implement the graphical interface based on PyQt5 (real\_estate.gui)

2.Realize the visual display of information such as real estate, customers, and matching results

3.Realize the interaction logic between the interface and the back-end data

4.Be responsible for the optimization of interface operation processes and user experience

5.Responsible for writing reports

## 6.3 Tips

**Pengchuhan:** As the team leader for this real estate management system, I have gained valuable hands-on experience throughout the entire software development life cycle. My primary responsibilities included designing and implementing the core data structures (such as AVL trees and queues), developing intelligent algorithms for property-client matching and dynamic pricing, and coordinating comprehensive unit testing. Additionally, I was in charge of requirement analysis, and integrating all modules into a cohesive and robust application.

Leading this project not only improved my technical skills in data structures and algorithms, but also honed my abilities in project management, communication, and problem-solving. I learned the importance of clear architectural planning, rigorous testing, and seamless collaboration among team members. Facing real-world constraints such as efficiency, maintainability, and scalability, I became more adept at making design trade-offs and optimizing both performance and code quality.

This course project has deepened my understanding of software engineering practices and reinforced my confidence in leading technical teams to deliver high-quality solutions.

**Zhangmingyang**：In this course, I used the vscode app for the first time, learned how to use it, and gained a clearer understanding of the program structure. While writing code, I continuously deepened my understanding of the knowledge point of binary trees.In the process of designing and implementing data models for real estate, customers, etc., I deeply realized the importance of data structure. A good data model can effectively support the stability and scalability of the entire system, especially when dealing with complex business logic. A clear and efficient model design is the key to success. In terms of the functional implementation of data validation, exception handling and log output, I realized that a robust error handling mechanism is crucial for the stability of the system. Through reasonable anomaly capture and logging, problems can be quickly located and timely responses made, effectively enhancing the system's fault tolerance. Finally, participating in the process of supplementing and improving the unit test cases made me deeply realize the importance of testing for ensuring the quality of the system. Unit tests not only help catch potential bugs, but also enhance the maintainability and scalability of the code. During this process, gradually improving test coverage and ensuring the comprehensiveness of test cases have also made me pay more attention to the quality of the code.

Overall, these tasks have helped me deepen my understanding of data processing, exception management, and test assurance in practice, enhanced my development skills, and made me more familiar with the complex requirements and technical challenges in actual projects

**Niexiaoya**：During the development of the Real Estate Intelligent Management System, I was mainly responsible for the implementation of the property and client manager functionalities, and assisted the team leader with system integration and debugging. My specific tasks included developing the property manager and client manager modules under real\_estate.managers, improving the core featus such as CRUD operations and queue management for properties and clients, participating in the implementation of some core algorithms, as well as system integration, joint debugging, and bug fixing. Through this project, I gained a deep understanding and practical experience of object-oriented design principles. By modularizing the management logic for properties and clients, I greatly improved the maintainability and scalability of the code. While implementing the CRUD functionalities, I realized the efficiency and importance of data structures such as AVL trees and queues in real-world business scenarios. The queue ensures orderly processing of client requests, while the AVL tree enables efficient and reliable retrieval and range queries for property information. In the process of assisting the team leader with system integration and debugging, I learned how to collaborate with team members, how to seamlessly connect the modules each person was responsible for, and how to promptly identify and fix various bugs that arose during integration. Through continuous joint debugging and testing, I gained a more comprehensive understanding of the overall system architecture and data flow, and improved my ability to analyze and solve complex problems. Additionally, participating in the implementation of some core algorithms made me appreciate the importance of combining algorithm design with business requirements. How to improve system performance and user experience while ensuring business logic correctness was a goal I constantly pursued and practiced throughout this project. Overall, this project not only enhanced my programming and engineering skills, but also made me realize the crucial role of teamwork, module integration, and system debugging in real-world software development. I believe these valuable experiences will lay a solid foundation for my future studies and career.

**Masixi**：The process of developing a graphical user interface for our real estate agency system presents certain challenges. My main job is to design and build the PyQt5 interface, ensuring that users can easily view and manage properties, clients, and matching results, and smoothly connect all content with the backend. At first, I mainly thought about how to make the interface clear and easy to use, hoping that users could quickly find what they needed, so I carefully organized the layout and used tables and dialog boxes to display information in a friendly way. The simplicity and aesthetics of the system are important to me, so I pay attention to details such as button placement, color, and feedback messages. One of the most challenging parts is to enable real-time communication between the GUI and the backend. I must ensure that when users make changes, the data is updated immediately and any errors are handled properly. I also try to make every step in the user process as smooth as possible, reduce unnecessary clicks, and add useful prompts where needed. Through this project, I have learned a lot about PyQt5 and Python, but more importantly, how important it is to think from the user's perspective.

## 6.4 Difficulties and Solution of Problems

1. AVL Tree Visualization Complexity

Challenge:  
Designing a clear and interactive GUI representation of the self-balancing AVL Tree was non-trivial, especially when handling node positioning, edge drawing, and dynamic updates after property operations (insert/delete). The tree's recursive and rebalancing nature required precise rendering logic to preserve readability and performance.

Solution:  
We designed a custom TreeNodeItem class with hover, selection, and context menu support, enabling real-time interaction with the tree nodes. The visualization now includes:

* Dynamic re-rendering upon data updates
* Readable labeling of property IDs and prices
* Smooth layout using coordinate offsets to prevent overlap

This not only improved usability but also made structural relationships within the AVL Tree intuitively visible for users.

2. Redundant Property Storage in PropertyManager

Challenge:  
The original implementation maintained two separate structures in PropertyManager: an AVL Tree (self.tree) and a dictionary (self.properties). Both stored the entire property dataset, leading to:

* Space redundancy (O(2N) memory usage)
* Consistency risks due to potential data desynchronization
* Inefficient operations relying more on the dictionary instead of the AVL Tree

Solution:  
We removed the dictionary entirely and standardized all property operations (add, delete, search, match) to use the AVL Tree exclusively. As a result:

* Memory usage is optimized to O(N)
* Query and update operations are log-time efficient (O(log N))
* Data integrity is preserved by maintaining a single source of truth

This change aligns with best practices in data structure design, ensuring both efficiency and maintainability.

3. ClientQueue Traversal and Removal Logic

Challenge:  
Earlier versions of ClientManager used inefficient dequeue-enqueue loops to traverse or modify the client queue. This approach:

* Introduced unnecessary data movement
* Complicated the logic for operations like find\_client\_by\_id and remove\_client

Solution:  
We refactored the queue traversal to a single-pass linked list traversal, maintaining references to both current and previous nodes. This allowed in-place modifications and simplified logic for:

* Finding a client by ID
* Removing a specific client
* Maintaining FIFO ordering without structural disturbance

This made queue operations more intuitive, performant (O(N)), and readable.

4. Improved Matching and Purchasing Logic

Challenge:  
Originally, property matching returned arbitrary or the first available result, ignoring pricing or client preferences.

Solution:  
We refined the matching algorithm to:

* Score candidate properties using weighted criteria (type, location, price range)
* Select the most affordable suitable match
* Respect client constraints strictly

This enhanced the realism and intelligence of the property-client matching process.

5. GUI Functional & Visual Enhancements

Challenge:  
The initial system lacked a graphical interface, offering only raw console output, which was hard to interpret. Early console outputs exposed unnecessary technical details (e.g., OrderedDict, file paths) and lacked structured formatting.

Solution:  
We developed a full-featured PyQt5 GUI with the following improvements:

* Interactive AVL Tree View with tooltips and dynamic updates
* Client Queue and Property List displayed in structured widgets
* Action Buttons for adding, removing, viewing, and purchasing
* Analytical Dashboards showing charts (e.g., transaction rate, price trends)

This transformation greatly improved user experience, clarity, and project presentation quality.

6. Testing Coverage and Edge Case Validation

Challenge:  
Earlier test coverage lacked consideration for edge cases such as:

* Adding duplicate clients or properties
* Attempting invalid operations (e.g., purchase without budget)

Solution:  
We extended the unittest coverage to include:

* Validation of uniqueness constraints
* Error handling for invalid transactions
* Assertions for queue integrity and AVL Tree balance

This ensured robustness, bug resistance, and confidence in system behavior under edge scenarios.

窗体顶端

窗体底端