PRIVATE ACCOMMODATION SYSTEM DOCUMENTATION

Contents

1.0 Introduction

The APH, Asia Pacific Home had designed and developed a renting system for Klang Valley, Malaysia. The 3 users that are in the renting system are the tenant, manager, and admin. The features tenants can sort and display property information, search property information by using property ID, save favorite properties, place rent requests from the favorite properties list, and display the renting history. The manager's role in the system can display all registered tenants' information, search for the tenants' details, delete those inactive tenants, summarize the top 10 properties saved by tenants, manage the process of renting system when requested by the tenants, and manage the payment of tenancy. While for admin, the admin will just able to add new managers and modify their status and display all the information of tenants and properties with the filter that has been designed.

1.1 Data Structure

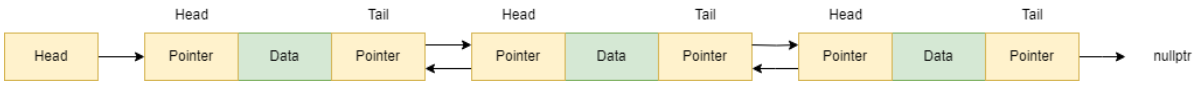


Figure 1: Data Structure

This structure has a head and the data. The data have two pointers which is head and tail. This is because this structure is a doubly linked list. It allows head pointer to tail and tail to head. The advantage of using doubly linked list is bidirectional traversal. By using doubly linked list, user able to traverse the list forward and backword, which can be useful for certain algorithms and operations.

1.2 Overall System Flowchart

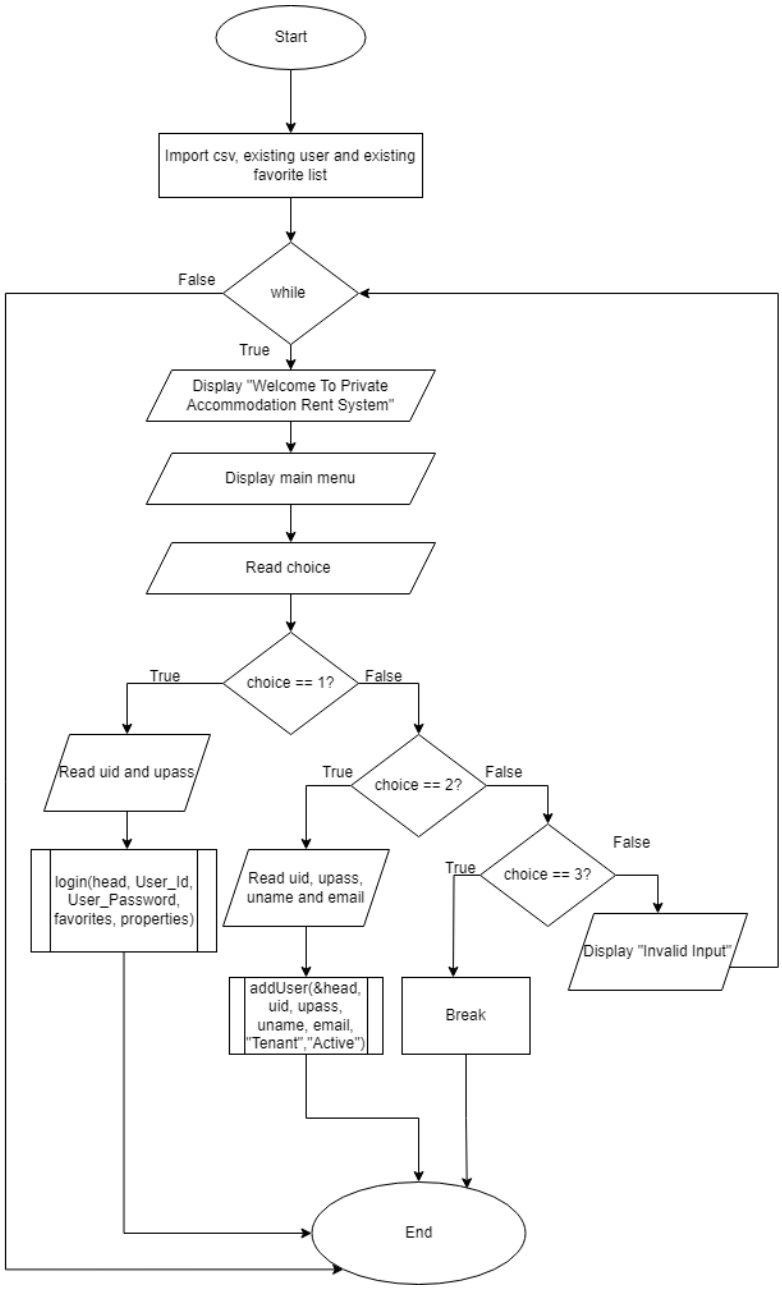


Figure 2: Overall System Flowchart

The flowchart above represents the conceptual flow of the private accommodation system. The system will import csv file related to property dataset, read existing user account and tenant favourite list. Then, the while loop is implemented to display the main menu. The user choice is read and based on their choice, it will either go to login function, register function (addUser) or break the while loop (exit the system). If the choice is more than 3 and less than 1, the system will loop the content under the while loop block.

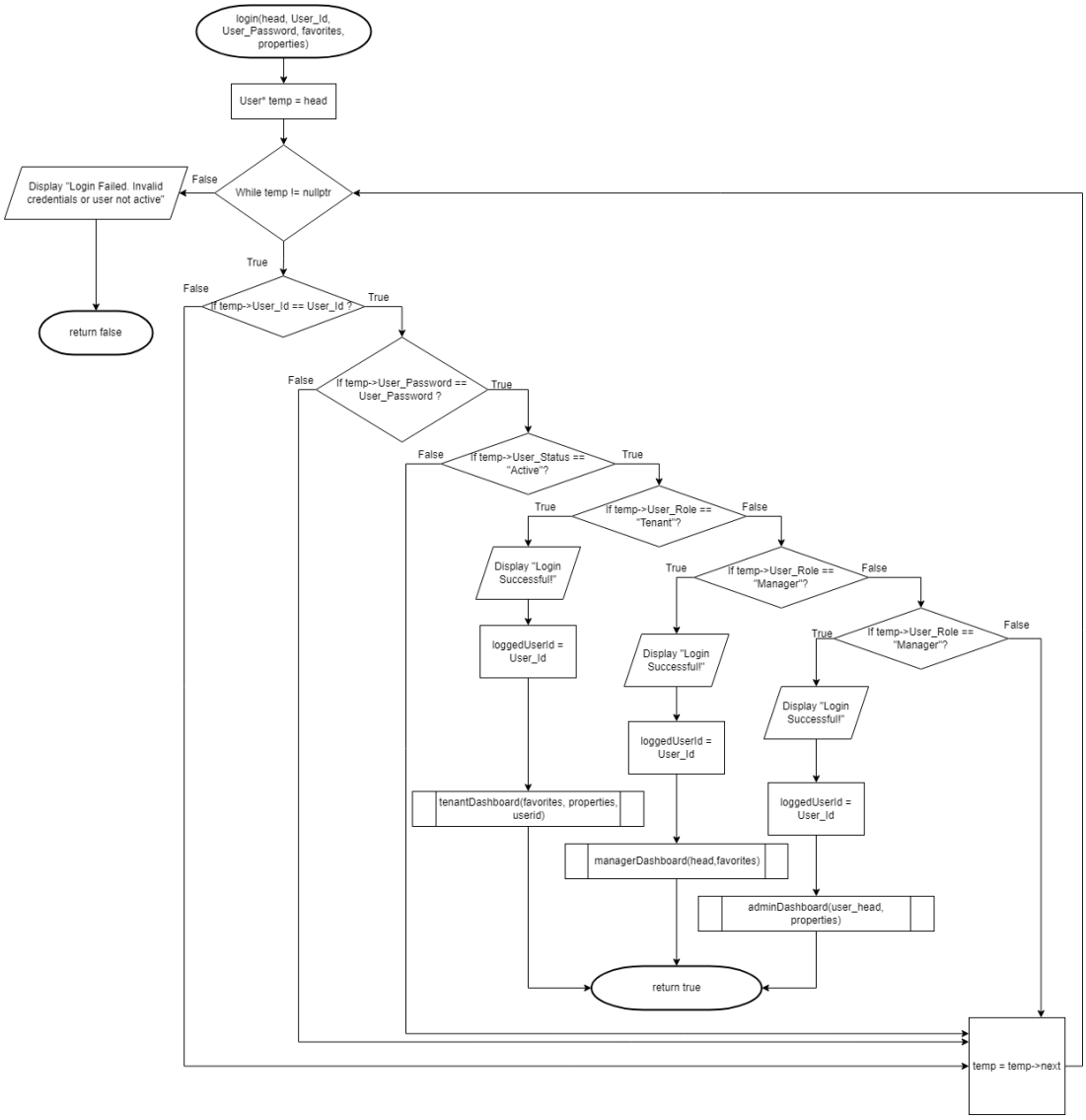


Figure 3: Login Flowchart

The flowchart above is to demonstrate the login flow of the system. This is to an extension of the overall system flow to provide a bigger picture on the overall workflow of the system. The user linked list is read and initialised as temp. Then, a while loop is implemented to check whether the user id, password, user status and role match the element of the node in the user linked list or not. If it is true, the system will navigate to the function that is related to enter specific user dashboard function. However, the login function will display a message and return false if it is false. The user linked list is traversed node by node to check the data until it reached the last node. The revised overall system flowchart is separating the main menu and each user dashboard. This is because the actual implementation of the coding for the system will be cleaner and more manageable by separating user dashboards and main menu using functions. It will be very messy if 3 users' dashboard and main menu placed under one segment of code.

1.3 Data Structure Source Code

1.3.1 Structure of Tenant Fav List

```
struct TenantFavList {
    string User_Id;
    int Ads_Id;
    string Prop_Name;
    string Size;
    string Monthly_Rent;
    string Completion_year;
    string Location;
    string Property_Type;
    string Payment_Status;
    TenantFavList* back;
    TenantFavList* next;
};
```

Figure 4: Structure of Tenant Favourite List

The structure of the tenant favorite list or been set as TenantFavList, is to store the data of user id, property name, size, monthly rent, completion year, location, property type, payment status, and the pointer of the structure. The reason for creating a new structure instead of using the same structure as the property structure is because later when doing the function of tenant I will need to pass the usable information only to this structure from the properties structure. Therefore, the TenantFavList structure had been created.

1.3.2 Structure of User

```
struct User {
    string User_Id;
    string User_Password;
    string User_Name;
    string User_Email;
    string User_Role;
    string User_Status;
    User* next;
    User* prev;
};
```

Figure 5: Structure of User

A user structure is declared. There are various member variables created under the user struct. Those member variables are created based on the common user registration account information. For example, user id, user password, username, user role and user status. These

variables are initialised in data type of string to suit the actual usage of user account information. Besides, two pointers which are next and prev are initialised, this to create a doubly linked list. Next pointer is utilised to point to the next node and previous pointer is utilised to point to previous node in later usage of user linked list. This user structure will be implemented in managing data related to users.

1.3.3 Structure of Property

```
struct Property {
    int ads_id;
    string prop_name;
    string completion_year;
    string monthly_rent;
    string location;
    string property_type;
    string rooms;
    string parking;
    string bathroom;
    string size;
    string furnished;
    vector<string> facilities;
    vector<string> additional_facilities;
    string region;
    Property* prev, * next; // add prev and next pointers

    Property* left;   // left child for BST
    Property* right;  // right child for BST

    Property() : left(nullptr), right(nullptr), prev(nullptr), next(nullptr) {} //This list initializes the pointers to nullptr whenever a new instance of Property is created.
};
```

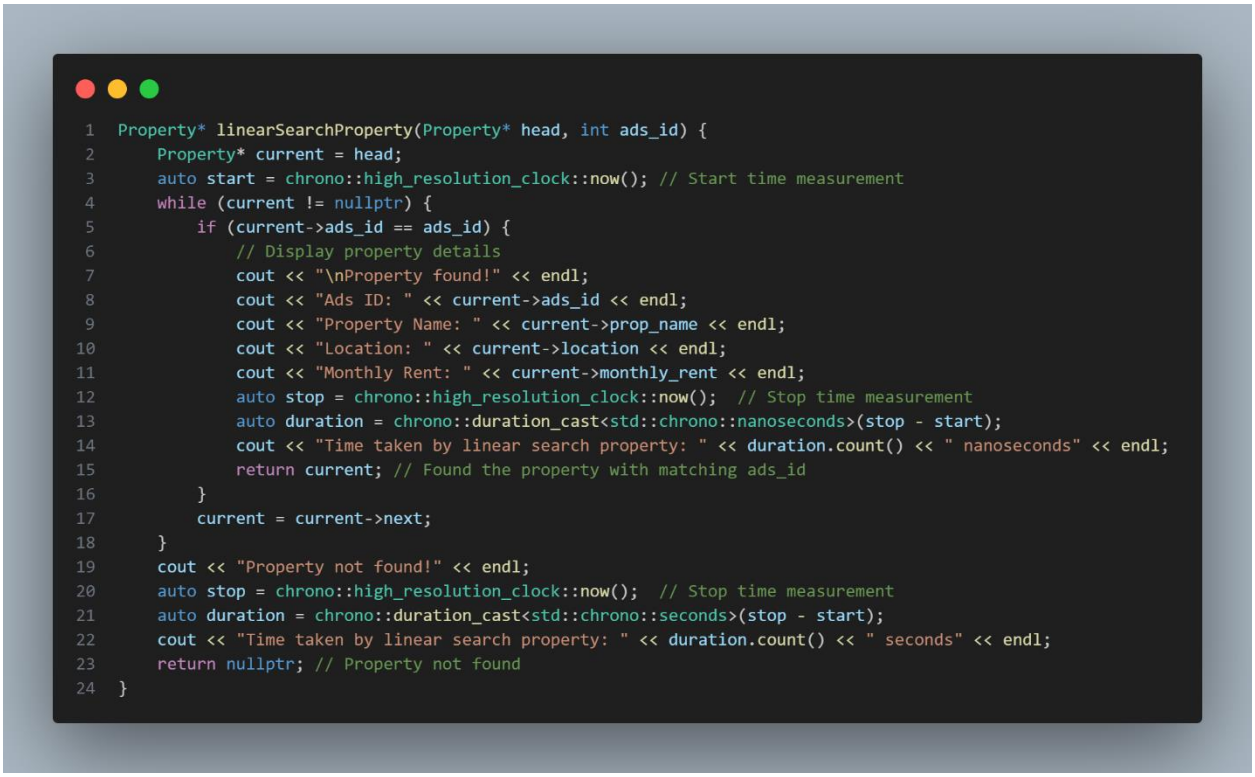
Figure 6: Structure of Property

The structure of the property had defined many variables inside such as property id, property name, year of completion, monthly rent, location, property type, number of rooms, parking, number of bathrooms, size, furnished, facilities, additional facilities, region, pointers, and the left and right. Most of the variables are using string but the property id are using integer. This is because we are using property id to doing sort or binary search tree. Besides, the facilities and additional facilities are using vector to store the data. The variable next and prev are using for the doubly linked list. The left and right is to define the binary search tree. The last line is to set the variable to nullptr. By setting these pointers to nullptr, it ensures that when a new instance of property is created, these pointers won't point to random memory location.

2.0 Implementation and Result

2.1 Algorithm Source Code

2.1.1 Linear Search



```
Property* linearSearchProperty(Property* head, int ads_id) {
    Property* current = head;
    auto start = chrono::high_resolution_clock::now(); // Start time measurement
    while (current != nullptr) {
        if (current->ads_id == ads_id) {
            // Display property details
            cout << "\nProperty found!" << endl;
            cout << "Ads ID: " << current->ads_id << endl;
            cout << "Property Name: " << current->prop_name << endl;
            cout << "Location: " << current->location << endl;
            cout << "Monthly Rent: " << current->monthly_rent << endl;
            auto stop = chrono::high_resolution_clock::now();  // Stop time measurement
            auto duration = chrono::duration_cast<std::chrono::nanoseconds>(stop - start);
            cout << "Time taken by linear search property: " << duration.count() << " nanoseconds" << endl;
            return current; // Found the property with matching ads_id
        }
        current = current->next;
    }
    cout << "Property not found!" << endl;
    auto stop = chrono::high_resolution_clock::now();  // Stop time measurement
    auto duration = chrono::duration_cast<std::chrono::seconds>(stop - start);
    cout << "Time taken by linear search property: " << duration.count() << " seconds" << endl;
    return nullptr; // Property not found
}
```

Figure 7: Linear Search

In the linear search algorithm, the parameters that need to be set are the pointer of property and the ads_id. First use “current” as the pointer of the property and the auto start is the feature of chrono library that enables the system to count the time. Then, a while loop will start if the “current” is not null. The system will first compare the current’s ads_id with the ads_id that is input by the user. If the property is found, then it will display the details of the property, the time measurement will stop and print out and return the property information to the “current”. While if the property is not found, the message “Property not found” will be prompted and the time measurement will stop and printed out.

2.1.2 Bubble Sort

```
void swap(Property* a, Property* b) {
    int temp_ads_id = a->ads_id;
    string temp_prop_name = a->prop_name;
    string temp_completion_year = a->completion_year;
    string temp_monthly_rent = a->monthly_rent;
    string temp_location = a->location;
    string temp_property_type = a->property_type;
    string temp_size = a->size;
    string temp_furnished = a->furnished;
    string temp_region = a->region;
    string temp_rooms = a->rooms;
    string temp_parking = a->parking;
    string temp_bathroom = a->bathroom;
    vector<string> temp_facilities = a->facilities;
    vector<string> temp_additional_facilities = a->additional_facilities;

    a->ads_id = b->ads_id;
    a->prop_name = b->prop_name;
    a->completion_year = b->completion_year;
    a->monthly_rent = b->monthly_rent;
    a->location = b->location;
    a->property_type = b->property_type;
    a->size = b->size;
    a->furnished = b->furnished;
    a->region = b->region;
    a->rooms = b->rooms;
    a->parking = b->parking;
    a->bathroom = b->bathroom;
    a->facilities = b->facilities;
    a->additional_facilities = b->additional_facilities;

    b->ads_id = temp_ads_id;
    b->prop_name = temp_prop_name;
    b->completion_year = temp_completion_year;
    b->monthly_rent = temp_monthly_rent;
    b->location = temp_location;
    b->property_type = temp_property_type;
    b->size = temp_size;
    b->furnished = temp_furnished;
    b->region = temp_region;
    b->rooms = temp_rooms;
    b->parking = temp_parking;
    b->bathroom = temp_bathroom;
    b->facilities = temp_facilities;
    b->additional_facilities = temp_additional_facilities;
}
```

Figure 8: Swap Function

This swap function has been created two swap the values of two Property objects and this function will be used on bubble sort later. In this function there will be two pointers created which are a and b. All the attributes of Property object a are saved into temporary variables. For example, a's ads_id is saved in temp_ads_id. Then it assigns all the attributes of Property object b to a. Finally, the function assigns the temporarily stored values of a to the attribute of b and the swap between the property object had completed.

```
void bubbleSort(Property* head) {
    if (head == nullptr);

    bool swapped;
    Property* ptr1;
    Property* lptr = nullptr;

    do {
        swapped = false;
        ptr1 = head;

        while (ptr1->next != lptr) {
            // Compare monthly rent first
            if (stoi(ptr1->monthly_rent) < stoi(ptr1->next->monthly_rent)) {
                swap(ptr1, ptr1->next);
                swapped = true;
            }
            // If monthly rent is the same, compare location
            else if (stoi(ptr1->monthly_rent) == stoi(ptr1->next->monthly_rent) &&
                stoi(ptr1->location) < stoi(ptr1->next->location)) {
                swap(ptr1, ptr1->next);
                swapped = true;
            }
            // If both monthly rent and location are the same, compare size
            else if (stoi(ptr1->monthly_rent) == stoi(ptr1->next->monthly_rent) &&
                stoi(ptr1->location) == stoi(ptr1->next->location) &&
                stoi(ptr1->size) < stoi(ptr1->next->size)) {
                swap(ptr1, ptr1->next);
                swapped = true;
            }

            ptr1 = ptr1->next;
        }
        lptr = ptr1;
    } while (swapped);
}
```

Figure 9: Bubble Sort

The parameter of bubble sort will be the pointer of property only. First, it will very the list, if the list is empty, then the system will do nothing. Else, a Boolean swap was created, ptr1 used to transverse the list and lptr is a pointer that always points to the last element that was sorted in previous. The do-while loop will keep running until there are no swaps are made to ensure the lists is fully sorted before exiting the loop. In the loop Boolean swapped is created and initialized as false statements. The system will first compare the monthly rent, if the current object has lesser monthly rent than the next object, the function swap that mentioned above

will called and the Boolean swapped will change to true. If the monthly rent values are equal, then it will compare the location. If the location of the current object is smaller in alphabetical order than the next object, swapped function call and Boolean swapped become true. Lastly, if both monthly rent and location are equal, then only the system will come to compare the size. If the current object is smaller size than the next object, swapped occur and the Boolean becomes true statement. After each pass of the inner loop, the lptr is updated to point to the last sorted element. The loop will end if the Boolean swapped remain false after all the comparisons.

2.1.3 Merge Sort

```
699  Property* mergeSort(Property* head) {
700      // Base case: If the list is empty or has only one element, it's already sorted
701      if (!head || !head->next)
702          return head;
703      auto start = chrono::high_resolution_clock::now(); // Start time measurement
704      // Calculate the length of the linked list
705      int length = 0;
706      Property* current = head;
707      while (current) {
708          length++;
709          current = current->next;
710      }
711
712      // Create a dummy node to simplify merging
713      Property dummy;
714      dummy.next = head;
715
716      // Iterate over the list in multiple steps, merging sublists of size step
717      for (int step = 1; step < length; step *= 2) {
718          Property* prevTail = &dummy; // Pointer to the tail of the merged list
719          Property* curr = dummy.next; // Pointer to the current sublist
720
721          // Merge pairs of sublists of size 'step'
722          while (curr) {
723              Property* left = curr; // Left sublist
724              Property* right = split(left, step); // Right sublist
725              curr = split(right, step); // Move the current pointer to the next sublist
726
727              // Merge the left and right sublists, updating the prevTail pointer
728              prevTail = merge(prevTail, left, right);
729          }
730      }
731      auto stop = chrono::high_resolution_clock::now();  // Stop time measurement
732      auto duration = chrono::duration_cast<std::chrono::seconds>(stop - start);
733      cout << "Time taken by merge sort: " << duration.count() << " seconds" << endl;
734
735      return dummy.next; // Return the sorted list
736  }
```

Figure 9: Merge Sort

The merge sort function accepts a pointer to the head of the property linked list. For line 701 to 702, the head pointer is checked. If the head pointer is empty or there is only one element in the linked list, it will exit the function. The linked list referred to the property linked list passed

to the head pointer. For line 705 to 710, the length variable equals to zero and current pointer refers to head are initialised. The while loop is applied to count the length of the linked list iterative. The while loop break if the header reached the last node of the linked list. For 713 to 714, the dummy pointer is initialised for the merging process. The next pointer of dummy is set to the starting point of the property linked list. For line 722 to 730, there are nested loops applied. The outer loop double up the size of the sub-lists in each iteration, this is to control size of sub-lists to be merged. Then, prevTail pointer is initialised and point to the dummy node, this is to keep track of the last node of merged sub-lists. The curr is initialised and point to the next node of the linked list. Curr pointer is utilised to traverse the linked list to be merged later in the while loop. In the inner loop, it iterates through the linked list for merging sub-lists of current size until reached the end of the node in the linked list. Next, left pointer is assigned to the curr, this is to mark the beginning of the left sub-list. For the right pointer, the split function is called to split starting from the left pointer into 2 parts which are left part (assigned previously) and right part from the left pointer based on given size (step). The split function is called again on the right sub-list with the same step size to split it further. The result is assigned to curr, which will be used in the next iteration of the loop. Then, the merge function is called and accepts left sub-list, right sub-list and prevTail pointer. The prevTail pointer is to point the last node of the merged linked list. The left and right sub-list are sorted in order and append the merged result to the sorted portion of the list. The prevTail is updated. After the nested loop ended, the sorted linked list is returned as the result.

```
738     // Function to split a linked list into two parts, given a step size
739     Property* split(Property* head, int step) {
740         for (int i = 1; head && i < step; i++)
741             head = head->next;
742 
743         if (!head)
744             return nullptr;
745 
746         Property* rest = head->next; // Store the second part of the split
747         head->next = nullptr; // Disconnect the first part
748         return rest; // Return the second part
749     }
```

Figure 10: Split Function

The split function above accepts a pointer to the head of property object and step in integer. The for loop iterative if the step given from the mergeSort function is less than i and the nodes in the sub-list passed into the function not equal to null. The head linked list will be navigated to next node. The head refers to the sub-list passed from mergeSort. Then, if the sub-list is null, it will return nulltpr. The rest pointer is initialised to store the second part of the sub-list. It is pointed to the next node of the current node of the head. Then, the first part is disconnected by

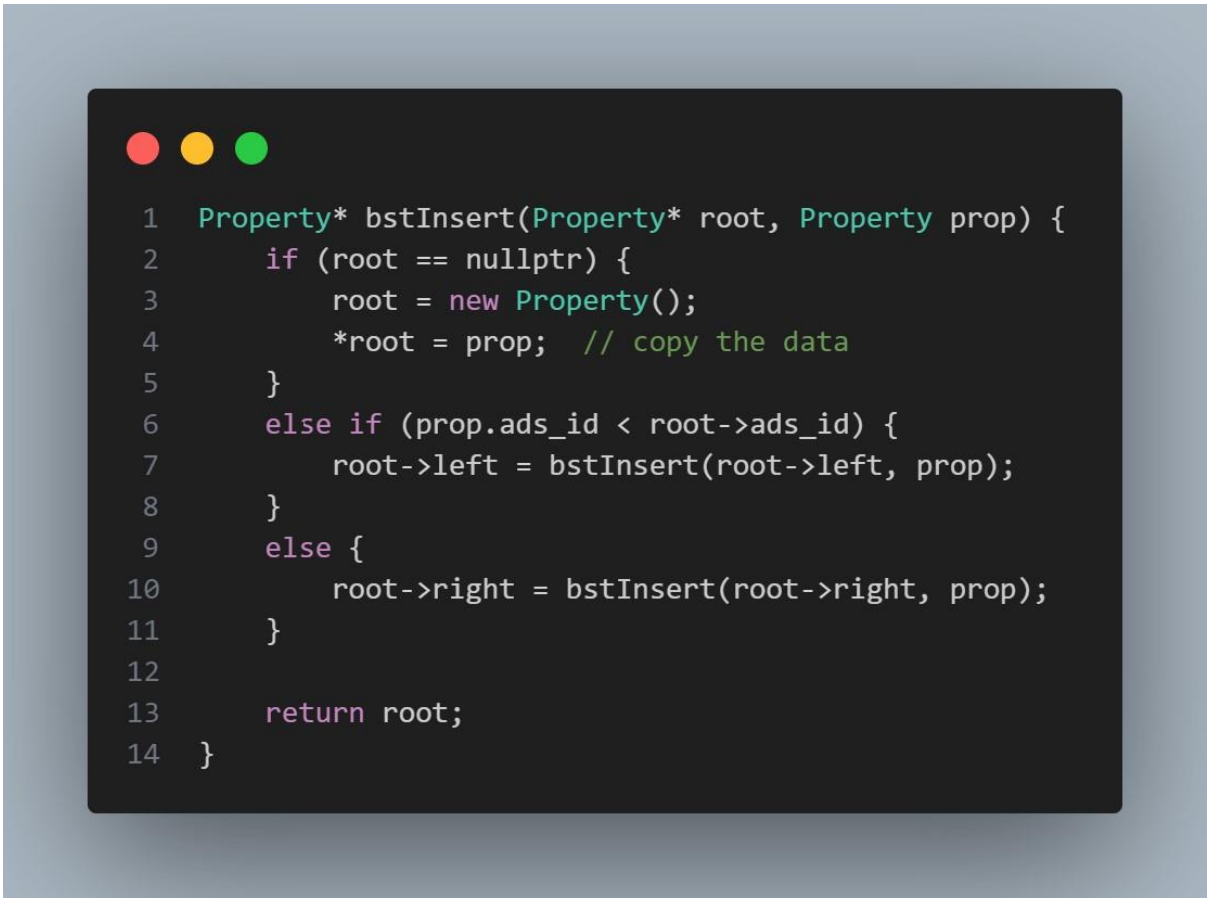
assigning the next node of the head pointing to null. Lastly, the second part of the sub-list is returned.

```
752  Property* merge(Property* tail, Property* left, Property* right) {
753      tail->next = nullptr; // Initialize the merged list with a null terminator
754
755      // Iterate while both left and right sublists have elements
756      while (left && right) {
757          // Compare and merge based on rent, location, and size criteria
758          if (extractRentValue(left->monthly_rent) > extractRentValue(right->monthly_rent) ||
759              (extractRentValue(left->monthly_rent) == extractRentValue(right->monthly_rent) && left->location > right->location) ||
760              (extractRentValue(left->monthly_rent) == extractRentValue(right->monthly_rent) &&
761                  left->location == right->location && extractSizeValue(left->size) > extractSizeValue(right->size))) {
762              tail->next = left; // Append left element to the merged list
763              left = left->next; // Move to the next element in the left sublist
764          }
765          else {
766              tail->next = right; // Append right element to the merged list
767              right = right->next; // Move to the next element in the right sublist
768          }
769          tail = tail->next; // Move the tail pointer
770      }
771
772      tail->next = left ? left : right; // Append any remaining elements from either sublist
773
774      while (tail->next)
775          tail = tail->next; // Move the tail pointer to the end of the merged list
776
777      return tail; // Return the tail of the merged list
778  }
```

Figure 11: Merge Function

The merge function above accepts 3 pointers, which are the merged linked list, left sub-list and right sub-list. Firstly, the merged linked list is initialised by setting the next node to null through tail. For 756 to 770 lines, the left sub-list and right sub-list is compared between each other on 3 criteria which is monthly rent, location and size. The comparison follows monthly rent > location > size. If left sub-list is greater, the left sub-list's node is appended into the merged linked list and navigate to the next node of the left sub-list. However, the right sub-list's node is appended into the merged linked list and navigate to the next node of the right sub-list if it is vice versa. Then, the merged linked list pointer is navigated to the next node. After the merging process, nodes left either in left sub-list or right sub-list are appended into the merged linked list. Next, the merged linked list's pointer is moved to the last node. Lastly, the merged linked list is returned.

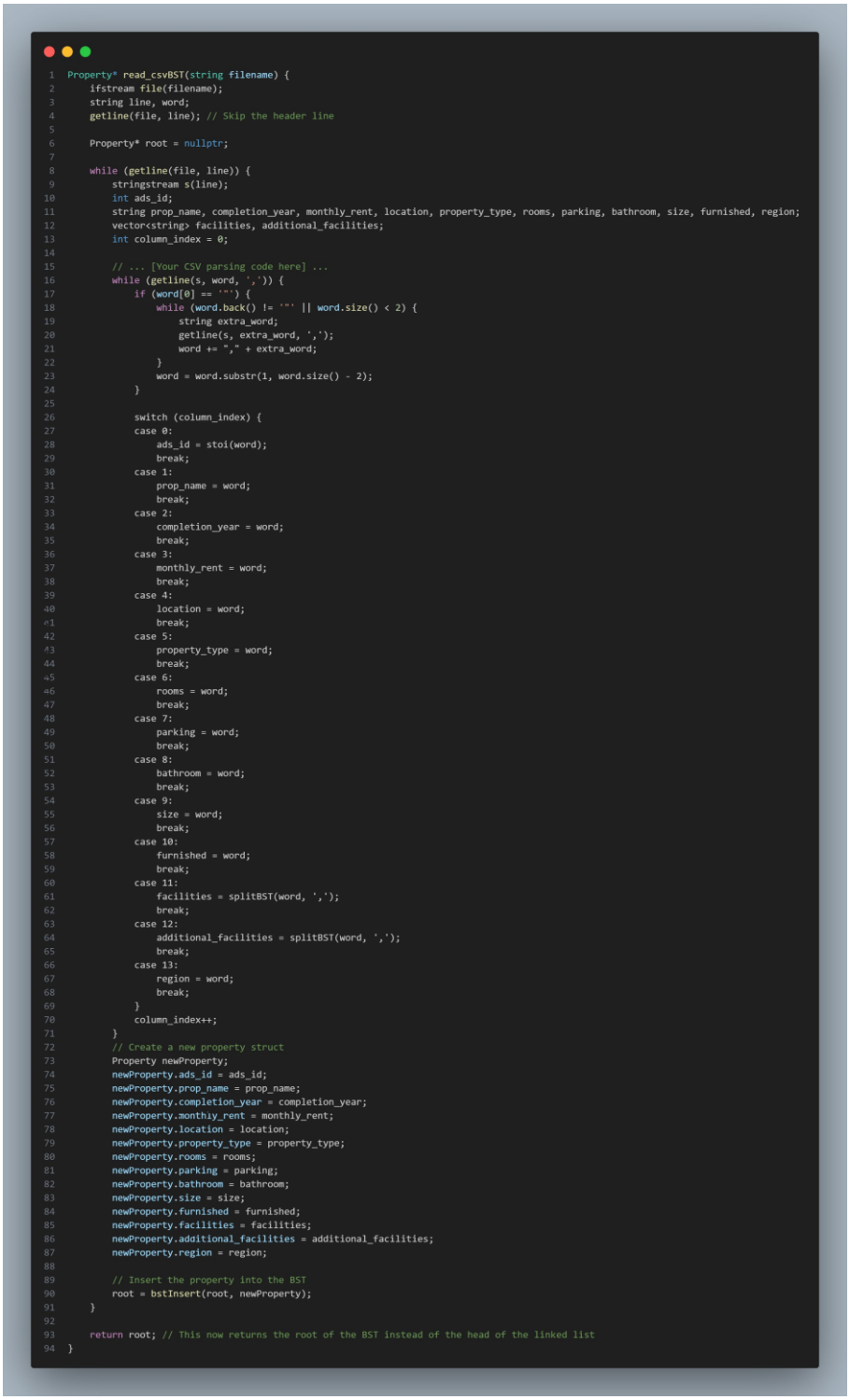
2.1.4 Binary Search Tree



```
1  Property* bstInsert(Property* root, Property prop) {
2      if (root == nullptr) {
3          root = new Property();
4          *root = prop;  // copy the data
5      }
6      else if (prop.ads_id < root->ads_id) {
7          root->left = bstInsert(root->left, prop);
8      }
9      else {
10         root->right = bstInsert(root->right, prop);
11     }
12
13     return root;
14 }
```

Figure 12: Binary Search Tree Insert Function

This is the binary search tree insert function which also known as the add function. This function has 2 parameters which is root and prop. The binary search tree add function is different with the normal doubly add function. On the line 2 is a basic scanning first linked list function, if the “root” is empty then copy the “prop” into the root. Then start from line 6 to 11 is to create a tree. It compare the “prop” properties id with the “root” properties id. If it is less than, it will go to the left side, else it will go to right side. The reason of doing one time is because this function will use in the read csv function which have a while loop there.



```
1  Property* read_csvBST(string filename) {
2      ifstream file(filename);
3      string line, word;
4      getline(file, line); // Skip the header line
5
6      Property* root = nullptr;
7
8      while (getline(file, line)) {
9          stringstream s(line);
10         int ads_id;
11         string prop_name, completion_year, monthly_rent, location, property_type, rooms, parking, bathroom, size, furnished, region;
12         vector<string> facilities, additional_facilities;
13         int column_index = 0;
14
15         // ... [Your CSV parsing code here] ...
16         while (getline(s, word, ',')) {
17             if (word[0] == '"') {
18                 while (word.back() != '"' || word.size() < 2) {
19                     string extra_word;
20                     getline(s, extra_word, ',');
21                     word += "," + extra_word;
22                 }
23                 word = word.substr(1, word.size() - 2);
24             }
25
26             switch (column_index) {
27             case 0:
28                 ads_id = stoi(word);
29                 break;
30             case 1:
31                 prop_name = word;
32                 break;
33             case 2:
34                 completion_year = word;
35                 break;
36             case 3:
37                 monthly_rent = word;
38                 break;
39             case 4:
40                 location = word;
41                 break;
42             case 5:
43                 property_type = word;
44                 break;
45             case 6:
46                 rooms = word;
47                 break;
48             case 7:
49                 parking = word;
50                 break;
51             case 8:
52                 bathroom = word;
53                 break;
54             case 9:
55                 size = word;
56                 break;
57             case 10:
58                 furnished = word;
59                 break;
60             case 11:
61                 facilities = splitBST(word, ',');
62                 break;
63             case 12:
64                 additional_facilities = splitBST(word, ',');
65                 break;
66             case 13:
67                 region = word;
68                 break;
69             }
70             column_index++;
71         }
72         // Create a new property struct
73         Property newProperty;
74         newProperty.ads_id = ads_id;
75         newProperty.prop_name = prop_name;
76         newProperty.completion_year = completion_year;
77         newProperty.monthly_rent = monthly_rent;
78         newProperty.location = location;
79         newProperty.property_type = property_type;
80         newProperty.rooms = rooms;
81         newProperty.parking = parking;
82         newProperty.bathroom = bathroom;
83         newProperty.size = size;
84         newProperty.furnished = furnished;
85         newProperty.facilities = facilities;
86         newProperty.additional_facilities = additional_facilities;
87         newProperty.region = region;
88
89         // Insert the property into the BST
90         root = bstInsert(root, newProperty);
91     }
92
93     return root; // This now returns the root of the BST instead of the head of the linked list
94 }
```

Figure 13: Binary Search Tree Read CSV Function

The read_csvBST function is most likely the same with read_csv function. The difference is only the add data function which is on line 89. It will read all the data and use the bstInsert function and pass the parameters into the function.

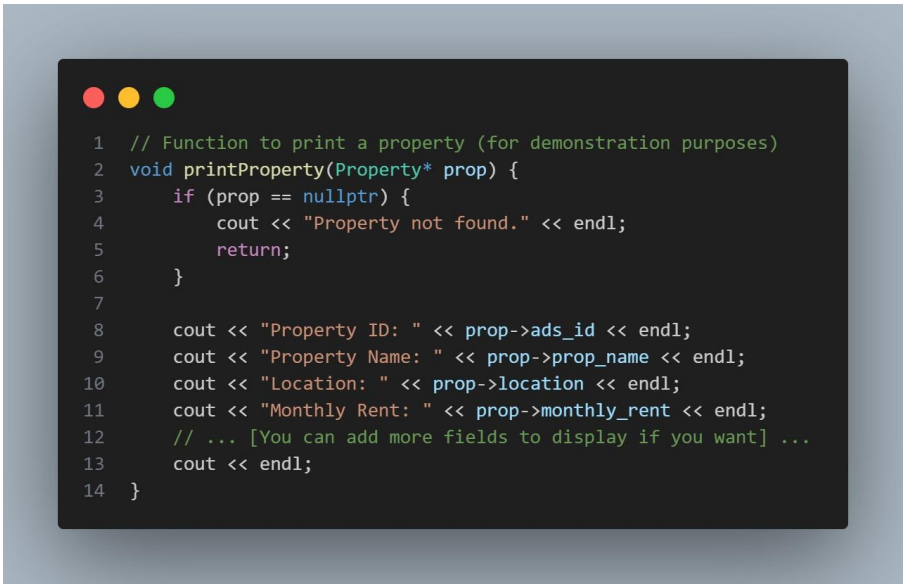
```
Property* bstSearch(Property* root, int ads_id) {

    if (root == nullptr || root->ads_id == ads_id) {
        return root;
    }

    if (root->ads_id < ads_id) {
        return bstSearch(root->right, ads_id);
    }
    return bstSearch(root->left, ads_id);
}
```

Figure 14: Binary Search Tree Search Function

This is a binary search tree search function with two parameter which is root and the property id. If the root is empty, it will directly return the root. If the root property id is smaller than property id input at the parameter, it will return the binary search again with the right side. Else it will return the left side.



```
// Function to print a property (for demonstration purposes)
void printProperty(Property* prop) {
    if (prop == nullptr) {
        cout << "Property not found." << endl;
        return;
    }

    cout << "Property ID: " << prop->ads_id << endl;
    cout << "Property Name: " << prop->prop_name << endl;
    cout << "Location: " << prop->location << endl;
    cout << "Monthly Rent: " << prop->monthly_rent << endl;
    // ... [You can add more fields to display if you want] ...
    cout << endl;
}
```

Figure 15: Binary Search Tree Print Property Function

This is a basic display property function but will used in binary search tree which mixed with binary search tree search function. If the search function return nullptr, it will display the property not found, else it will display the property information.

```
// Recursive function to print BST pre-order (for demonstration purposes)
void preOrder(Property* root) {
    if (root == nullptr) return;

    printProperty(root);
    inOrder(root->left);
    inOrder(root->right);
}

// Recursive function to print BST in-order (for demonstration purposes)
void inOrder(Property* root) {
    if (root == nullptr) return;

    inOrder(root->left);
    printProperty(root);
    inOrder(root->right);
}

// Recursive function to print BST post-order (for demonstration purposes)
void postOrder(Property* root) {
    if (root == nullptr) return;

    inOrder(root->left);
    inOrder(root->right);
    printProperty(root);
}
```

Figure 16: Binary Search Tree Print Order Function

These three functions are to print the order of binary search tree. It has preorder, inorder and postorder. All these functions mostly same, it just displays at different place.

2.2 Important Functions Source Code

Save Favorite Property Function

```
void saveFavoriteProperty(Property* properties, TenantFavList** head, TenantFavList* favorites, int ads_id, string userid) {
    TenantFavList* fav = new TenantFavList; //append to existing favorites list
    Property* temp = properties;
    TenantFavList* current = favorites; //exisiting favorites list
    bool propertyFound = false;
    bool alreadyFavorited = false;
    while (temp != nullptr) {
        if (temp->ads_id == ads_id ) {
            propertyFound = true;
            while (current != nullptr) {
                if (current->Ads_Id == ads_id) {
                    if (current->User_Id == userid) {
                        alreadyFavorited = true;
                        break;
                    }
                }
                current = current->next;
            }

            if (!alreadyFavorited) {
                fav->User_Id = userid;
                fav->Ads_Id = ads_id;
                fav->Prop_Name = temp->prop_name;
                fav->Completion_year = temp->completion_year;
                fav->Monthly_Rent = temp->monthly_rent;
                fav->Location = temp->location;
                fav->Property_Type = temp->property_type;
                fav->Size = temp->size;
                fav->next = nullptr;
                fav->back = nullptr;

                // Check if the list is empty
                if (*head == nullptr) {
                    *head = fav;
                    fav->back = nullptr;
                }
                else {
                    TenantFavList* current = *head;
                    while (current->next != nullptr) {
                        current = current->next;
                    }
                    current->next = fav;
                    fav->back = current;
                }
                cout << "Property with Ads_Id " << ads_id << " Added" << endl << endl;
                cout << "Favorite property saved: " << endl;
                cout << "Ads Id: " << fav->Ads_Id << endl;
                cout << "Property Name: " << fav->Prop_Name << endl;
                cout << "Completion Year: " << fav->Property_Type << endl;
                cout << "Monthly Rent: " << fav->Monthly_Rent << endl;
                cout << "Location: " << fav->Location << endl;
                cout << "Property Type: " << fav->Property_Type << endl;
                cout << "Size :" << fav->Size << endl << endl;
                //break;
            }
        }
        temp = temp->next;
    }

    if (!propertyFound) {
        cout << "Property with Ads_Id " << ads_id << " not found." << endl << endl;
    }

    else if (alreadyFavorited) {
        cout << "Property with Ads_Id " << ads_id << " is favourited." << endl << endl;
    }
}
```

Figure 17: Save Favourite Property Function

In this code, there will be five parameters which are the pointer to the linked list of properties, a double pointer to the head of the linked list in favorite properties, a pointer to the current list

of favorite properties, the ID of the property that inserted by the user and the user ID. First, a new node that is named “fav” will be created for the favorite linked list and there will be a pointer “temp” to point at the “properties” list and a “current” point at the favorite list. Besides, two booleans named propertyFound and alreadyFavorited will be created and set with the false condition. The validation process and the process of saving favorite properties will function in a while loop. First, the system will check for the properties data to check if is there any data inside if yes then will proceed to check for the properties’ ads_id to check whether the property that has been inserted is in the CSV file. If yes, then the propertyFound Boolean will become true and move on with the process or else it will print that the property with the ads_id inserted is not found in the CSV. If the property has been found in the CSV it will proceed with checking the ads_id and user_id whether the user already favorited this property or not. If the user already favorited this property, the alreadyFavorited Boolean will become true and print out the message that the property with this ads_id already had been saved by the user. If the user did not save this property before, it will save this property into the user's favorite list and display the property’s details to the user.

Add To Rent from Fav Function

```
void addToRentFromFav(TenantFavList* favorites, string userid) {
    if (favorites == nullptr) {
        cout << "There are no favorite properties in your list " << endl;
    }
    else {
        bool checkRequest = false;
        int adsIdToRent;
        cout << "Enter Ads Id of the property from favorite list to rent: ";
        cin >> adsIdToRent;
        TenantFavList* current = favorites;
        while (current != nullptr) {
            if (current->Ads_Id == adsIdToRent && current->User_Id == userid && current->Payment_Status == "") {
                checkRequest = true;
                current->Payment_Status = "Requested";
                cout << "Rent request has been placed for property: " << endl;
                cout << "Ads Id: " << current->Ads_Id << endl;
                cout << "Prop Name: " << current->Prop_Name << endl;
                cout << "Completion year: " << current->Property_Type << endl;
                cout << "Monthly Rent: " << current->Monthly_Rent << endl;
                cout << "Location: " << current->Location << endl;
                cout << "Property Type: " << current->Property_Type << endl;
                cout << "Size :" << current->Size << endl;
                cout << "Payment Status: " << current->Payment_Status << endl << endl;
                return;
            }
            current = current->next;
        }
        if (!checkRequest) {
           cout << "Property with Ads_Id " << adsIdToRent << " is already in tenancy process" << endl << endl;
        }
        else {
            cout << "Property with Ads_Id " << adsIdToRent << " not found in your favorites list." << endl << endl;
        }
    }
}
```

Figure 18: Add Rent Request Function

In this function, the parameter will be the pointer of the TenantFavList and the user id. First, it will validate whether there are any favorite properties in the user favorite list. If there is not any property it will prompt out a message that shows the user has no favorite property yet, else it will continue the process. A Boolean call checkRequest with conditional false will created and new int of adsIdToRent created. Then, it will require the user to input the property id that user wants to rent and point of "current" to point at the favorite's properties. Then, a while loop will start to compare the current's ads_id with adsIdToRent, current's user_id with userId and the current's payment_status must be empty then only the checkRequest will be true. Then a new payment status will be displayed along with the property's details. At the end of the process if the checkRequest is still in false condition, it will prompt a message that the property with this ads_id is already in tenancy process.

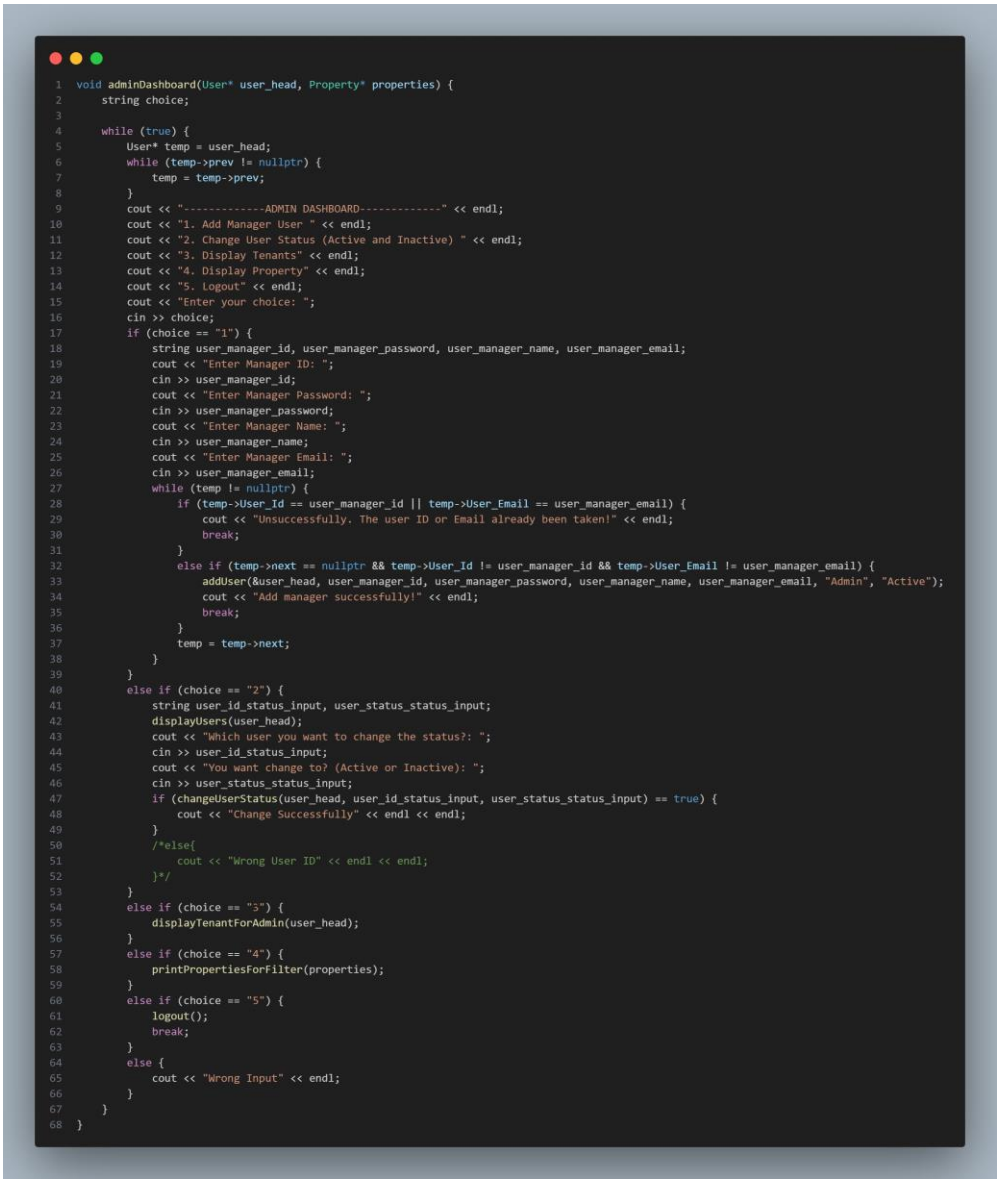
Display Rent History Function

```
void displayRentHistory(TenantFavList* favorites, string User_Id) {
    cout << "Renting History" << endl;
    if (favorites == nullptr) {
        cout << "No renting history found" << endl;
    }
    else {
        TenantFavList* current = favorites;
        bool found = false;
        while (current != nullptr) {
            if (current->User_Id == User_Id && current->Payment_Status != "") {
                cout << "User Id: " << current->User_Id << endl;
                cout << "Ads Id: " << current->Ads_Id << endl;
                cout << "Property Name: " << current->Prop_Name << endl;
                cout << "Completion year: " << current->Property_Type << endl;
                cout << "Monthly Rent: " << current->Monthly_Rent << endl;
                cout << "Location: " << current->Location << endl;
                cout << "Property Type: " << current->Property_Type << endl;
                cout << "Size :" << current->Size << endl;
                cout << "Payment Status: " << current->Payment_Status << endl << endl;
                found = true;
            }
            current = current->next;
        }
        if (!found) {
            cout << "No renting history found for User Id: " << User_Id << endl;
        }
    }
}
```

Figure 19: Display Rent History Function

In the display rent history function the parameters are pointer of tenant favorite list and the user id. First the system will validate do the user has any favorite list, if yes then proceed if not then it will show the message that the user does not have any renting history found. Then, a “current” point will be created to point at the favorites in the tenant favorite list and Boolean of found will set as false. If the current user_id same as the user_id inserted and the current payment_status not equal to empty, the data of the properties will be displayed, and the Boolean found will change to true status. At the end of process if Boolean found still remains false, it will prompt a message that shown there is no renting history found for this user id.

Admin Dashboard Function



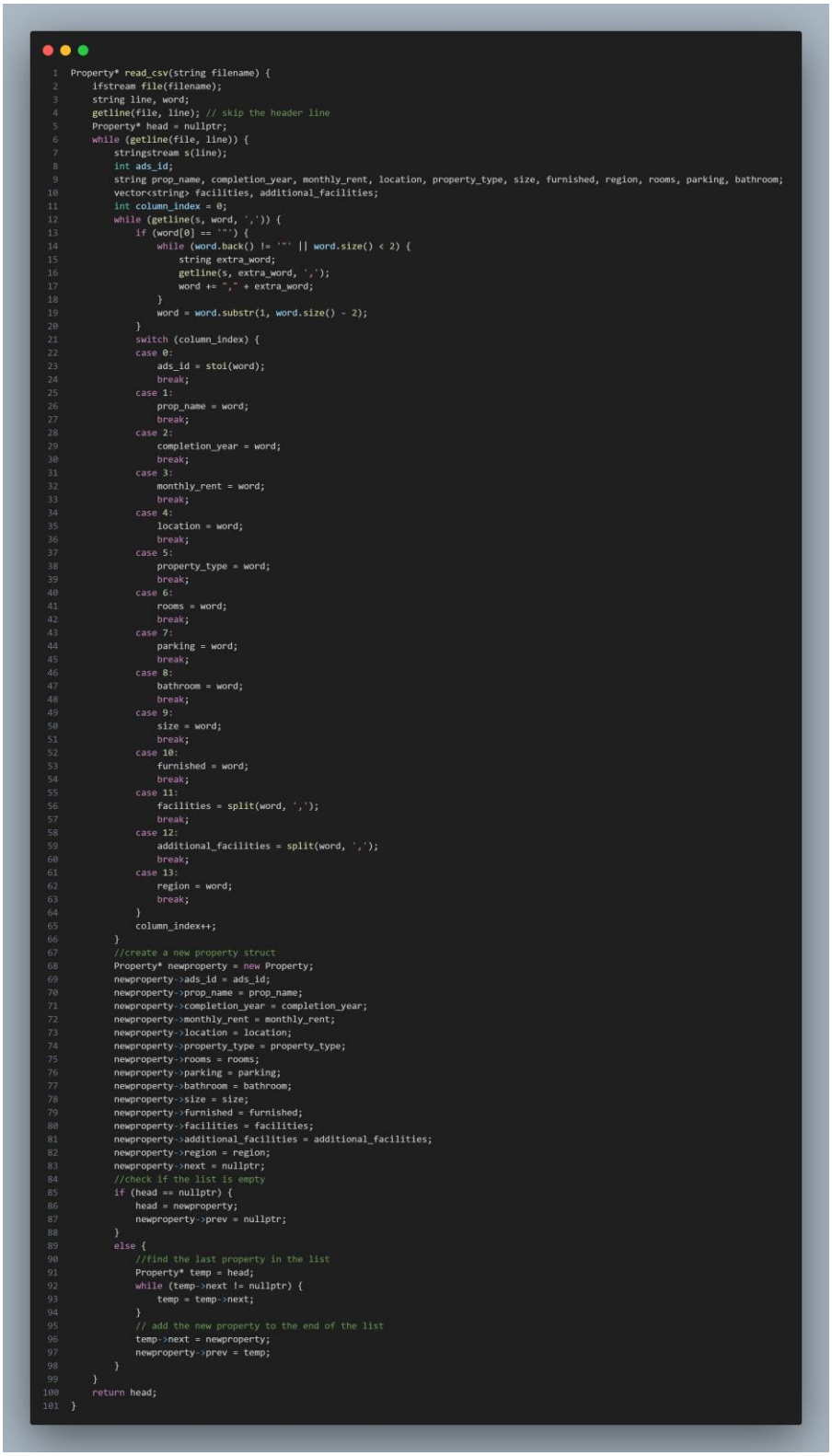
```
1  void adminDashboard(User* user_head, Property* properties) {
2      string choice;
3
4      while (true) {
5          User* temp = user_head;
6          while (temp->prev != nullptr) {
7              temp = temp->prev;
8          }
9          cout << "-------------ADMIN DASHBOARD-------------" << endl;
10         cout << "1. Add Manager User " << endl;
11         cout << "2. Change User Status (Active and Inactive) " << endl;
12         cout << "3. Display Tenants" << endl;
13         cout << "4. Display Property" << endl;
14         cout << "5. Logout" << endl;
15         cout << "Enter your choice: ";
16         cin >> choice;
17         if (choice == "1") {
18             string user_manager_id, user_manager_password, user_manager_name, user_manager_email;
19             cout << "Enter Manager ID: ";
20             cin >> user_manager_id;
21             cout << "Enter Manager Password: ";
22             cin >> user_manager_password;
23             cout << "Enter Manager Name: ";
24             cin >> user_manager_name;
25             cout << "Enter Manager Email: ";
26             cin >> user_manager_email;
27             while (temp != nullptr) {
28                 if (temp->User_Id == user_manager_id || temp->User_Email == user_manager_email) {
29                     cout << "Unsuccessfully. The user ID or Email already been taken!" << endl;
30                     break;
31                 }
32                 else if (temp->next == nullptr && temp->User_Id != user_manager_id && temp->User_Email != user_manager_email) {
33                     addUser(&user_head, user_manager_id, user_manager_password, user_manager_name, user_manager_email, "Admin", "Active");
34                     cout << "Add manager successfully!" << endl;
35                     break;
36                 }
37                 temp = temp->next;
38             }
39         }
40         else if (choice == "2") {
41             string user_id_status_input, user_status_status_input;
42             displayUsers(user_head);
43             cout << "Which user you want to change the status?: ";
44             cin >> user_id_status_input;
45             cout << "You want change to? (Active or Inactive): ";
46             cin >> user_status_status_input;
47             if (changeUserStatus(user_head, user_id_status_input, user_status_status_input) == true) {
48                 cout << "Change Successfully" << endl << endl;
49             }
50             /*else{
51                 cout << "Wrong User ID" << endl << endl;
52             }*/
53         }
54         else if (choice == "3") {
55             displayTenantForAdmin(user_head);
56         }
57         else if (choice == "4") {
58             printPropertiesForFilter(properties);
59         }
60         else if (choice == "5") {
61             logout();
62             break;
63         }
64         else {
65             cout << "Wrong Input" << endl;
66         }
67     }
68 }
```

Figure 20: Admin Dashboard Function

This is the admin dashboard function which display a main menu for admin. On the line 2, it opens a string call choice, this string is to let user input their choice and validate on the if else function. Start from the line 4, it creates a while true function. This is for keep looping until it false or break. At the line 5, I set the temp to the user_head which is the parameter inside the function. Then line 6 to 8 is to ensure that the linked-list is at the first linked-list. Then is the cout function and the cin function, the cin function is where user input the choice. The from line 17, it validates the choice, if it is 1 it will go to the add manager function. The choice 2, 3,

4, and 5 had this same technique just different function. If user input other than 1, 2, 3, 4, and 5, it will show the admin dashboard again since it is using while true function.

Read CSV Function



```
Property* read_csv(string filename) {
    ifstream file(filename);
    string line, word;
    getline(file, line); // skip the header line
    Property* head = nullptr;
    while (getline(file, line)) {
        stringstream s(line);
        int ads_id;
        string prop_name, completion_year, monthly_rent, location, property_type, size, furnished, region, rooms, parking, bathroom;
        vector<string> facilities, additional_facilities;
        int column_index = 0;
        while (getline(s, word, ',')) {
            if (word[0] == '"') {
                while (word.back() != '"' || word.size() < 2) {
                    string extra_word;
                    getline(s, extra_word, ',');
                    word += "," + extra_word;
                }
                word = word.substr(1, word.size() - 2);
            }
            switch (column_index) {
            case 0:
                ads_id = stoi(word);
                break;
            case 1:
                prop_name = word;
                break;
            case 2:
                completion_year = word;
                break;
            case 3:
                monthly_rent = word;
                break;
            case 4:
                location = word;
                break;
            case 5:
                property_type = word;
                break;
            case 6:
                rooms = word;
                break;
            case 7:
                parking = word;
                break;
            case 8:
                bathroom = word;
                break;
            case 9:
                size = word;
                break;
            case 10:
                furnished = word;
                break;
            case 11:
                facilities = split(word, ',');
                break;
            case 12:
                additional_facilities = split(word, ',');
                break;
            case 13:
                region = word;
                break;
            }
            column_index++;
        }
        //create a new property struct
        Property* newproperty = new Property;
        newproperty->ads_id = ads_id;
        newproperty->prop_name = prop_name;
        newproperty->completion_year = completion_year;
        newproperty->monthly_rent = monthly_rent;
        newproperty->location = location;
        newproperty->property_type = property_type;
        newproperty->rooms = rooms;
        newproperty->parking = parking;
        newproperty->bathroom = bathroom;
        newproperty->size = size;
        newproperty->furnished = furnished;
        newproperty->facilities = facilities;
        newproperty->additional_facilities = additional_facilities;
        newproperty->region = region;
        newproperty->next = nullptr;
        //check if the list is empty
        if (head == nullptr) {
            head = newproperty;
            newproperty->prev = nullptr;
        }
        else {
            //find the last property in the list
            Property* temp = head;
            while (temp->next != nullptr) {
                temp = temp->next;
            }
            // add the new property to the end of the list
            temp->next = newproperty;
            newproperty->prev = temp;
        }
    }
    return head;
}
```

Figure 21: Read CSV Function

The read csv function is for read the csv that provided by Asia Pacific University and add it into the linked list. Besides, it also requires some extra library to use the function which is fstream and sstream. These two libraries are a library to read the csv. At the line 2 is using the ifstream function to open or import the csv file. Then the getline function is to read and discards the header line of the csv. After that, open a while loop function for looping the data, which mean if the csv has 1000 lines of data, it will run 1000 times. Inside the while loop have set a column index variable for counting the column, it will use it at the bottom of the code. Then open another while loop function for the read all the column and extract the data. After getting the data, add inside the property linked list and set the pointer. Lastly return the property head which is the linked list of the property.

Login Function

```
bool login(User* head, string User_Id, string User_Password, TenantFavList* favorites, Property* properties) {
    User* temp = head;
    while (temp != nullptr) {
        if (temp->User_Id == User_Id && temp->User_Password == User_Password && temp->User_Status == "Active" && temp->User_Role == "Tenant") {
            cout << "Login Successful!" << endl;
            loggedUserId = User_Id; // Set the logged-in user's User_Id
            tenantDashboard(favorites,properties, loggedUserId);
            return true;
        }
        else if (temp->User_Id == User_Id && temp->User_Password == User_Password && temp->User_Status == "Active" && temp->User_Role == "Manager") {
            cout << "Login Successful!" << endl;
            loggedUserId = User_Id; // Set the logged-in user's User_Id
            managerDashboard(head,favorites);
            return true;
        }
        else if (temp->User_Id == User_Id && temp->User_Password == User_Password && temp->User_Status == "Active" && temp->User_Role == "Admin") {
            cout << "Login Successful!" << endl;
            loggedUserId = User_Id; // Set the logged-in user's User_Id
            //reset(temp);
            adminDashboard(head,properties);
            return true;
        }
        temp = temp->next;
    }

    cout << "Login Failed. Invalid credentials or user not active." << endl;
    return false;
}
```

Figure 22: Login Function

This is a login function with using bool which return true or false. Besides, this function will insert 5 parameters which is user, tenant favourite list, property, user id, and user password. Firstly, import the user linked list. Then open a while loop when that linked list not empty, it will run the code. Besides, this while loop will run all the linked list and detect the user id, user password, user status, and user role to login the specific role. Additionally, it will set the "loggedUserId" to the user id.

Logout Function

```
void logout() {
    if (loggedUserId.empty()) {
        cout << "No user is currently logged in." << endl;
    }
    else {
        cout << "Logged out user: " << loggedUserId << endl;
        loggedUserId = ""; // Reset the logged-in user's User_Id
    }
}
```

Figure 23: Logout Function

This logout function is just only to set the “loggedUserId” to empty.

Display All Tenant Function

```
void displayAllTenant(User* head) {
    cout << "Tenant Information" << endl;
    User* temp = head;
    while (temp != nullptr) {
        if (temp->User_Role == "Tenant") {
            cout << "----------------------" << endl;
            cout << "User Id: " << temp->User_Id << endl;
            cout << "Name: " << temp->User_Name << endl;
            cout << "Email: " << temp->User_Email << endl;
            cout << "Role: " << temp->User_Role << endl;
            cout << "Status: " << temp->User_Status << endl;
            cout << "----------------------" << endl;
        }
        temp = temp->next;
    }
}
```

Figure 24: Display All Tenant Function

This function is to display all the tenant. It open a while loop to loop the linked list and display the information.

Verify Existing User Function

```
bool verifyExistingUser(User* head, string User_Id) {
    User* temp = head;
    while (temp != nullptr) {
        if (temp->User_Id == User_Id) {
            return true;
        }
        temp = temp->next;
    }
    return false;
}
```

Figure 25: Verify Existing User Function

This function is to find the user id in the linked list or not. It creates a while loop to matching the user id. If it didn't find the user id, it will go to the next linked list data and detect again.

Add User Function

```
void addUser(User** head, string User_Id, string User_Password, string User_Name, string User_Email, string User_Role, string User_Status) {
    // Create a new user struct
    User* newUser = new User;
    newUser->User_Id = User_Id;
    newUser->User_Password = User_Password;
    newUser->User_Name = User_Name;
    newUser->User_Email = User_Email;
    newUser->User_Role = User_Role;
    newUser->User_Status = User_Status;
    newUser->next = nullptr;
    newUser->prev = nullptr;

    // Check if the list is empty
    if (*head == nullptr) {
        *head = newUser;
        newUser->prev = nullptr;
    }
    else {
        User* current = *head;
        while (current->next != nullptr) {
            current = current->next;
        }
        current->next = newUser;
        newUser->prev = current;
    }
}
```

Figure 26: Add User Function

This adds user function require user input user id, user password, username, user email, user role, and user status. Then it set the next and prev pointer for nullptr at the first. Then at the if else function is to verify the linked list is empty or not. If it is empty it will add in and set the prev pointer to nullptr, else it will add the user data at the end of the linked list.

Change User Status Function

```
bool changeUserStatus(User* head, string User_Id, string User_Status) {
    User* temp = head;
    while (temp != nullptr) {
        if (temp->User_Id == User_Id) {
            temp->User_Status = User_Status;
            cout << "User Status Changed Successfully." << endl;
            return true;
        }
        temp = temp->next;
    }

    cout << "User with ID " << User_Id << " not found." << endl;
    return false;
}

```

Figure 27: Change User Status Function

This function using a bool function and able to change the user status such as active and inactive. After user input the user id, the while loop will loop the linked list and find the correct user id. After the while loop found, it will replace the user status with the user input. If the while loop cannot found the user input, it will show user not found.

Generate Top 10 Favourite Property Function

```
void summarizeTopFavoriteProperties(TenantFavList* favorites) {
    cout << "Top 10 Favorite Properties Summary:" << endl;

    unordered_map<int, int> propertyCountMap;
    TenantFavList* current = favorites;

    // Count the occurrences of each property
    while (current != nullptr) {
        propertyCountMap[current->Ads_Id]++;
        current = current->next;
    }

    // Create a vector of pairs for sorting
    vector<pair<int, int>> propertyCountPairs;
    for (const auto& entry : propertyCountMap) {
        propertyCountPairs.push_back(make_pair(entry.first, entry.second));
    }

    // Sort the vector in descending order based on property count
    sort(propertyCountPairs.begin(), propertyCountPairs.end(),
        [](pair<int, int>& a, pair<int, int>& b) {
            return a.second > b.second;
        });
```

Figure 28: Generate Top 10 Favourite Property Function Part 1

The summariseTopFavoriteProperties accepts one pointer to the favourites from TenantFavList linked list. An underorder map is declared as propertyCountMap by taking ads id and count. The ads id map to the count of times in the propertyCountMap vector. The current pointer is assigned to the TenantFavList linked list to traverse it. The occurrence of each property is counted based on the ads id in the while loop. Then, a vector pairs is initialised as propertyCountMap. The propertyCountPairs is traversing using the for loop while adding the entries (ads id and count) into the vector. Lastly, the entries in the propertyCountPairs is sorted based on descending orders implementing sort library in C++. The a and b represents each pair of elements in the propertyCountPairs. The second entry which is count is compared between two pairs, if the condition a is greater than b, the lambda returns true, a comes before b and vice versa.

```
int count = 0;
for (auto& pair : propertyCountPairs) {
    if (count >= 10) {
        break;
    }

    cout << "TOP " << count + 1 << endl;

    // Find and display property details
    TenantFavList* fav = favorites;
    while (fav != nullptr) {
        if (fav->Ads_Id == pair.first) {
            cout << "Total Favourite: " << pair.second << endl;
            cout << "Ads Id: " << fav->Ads_Id << endl;
            cout << "Property Name: " << fav->Prop_Name << endl;
            cout << "Completion year: " << fav->Property_Type << endl;
            cout << "Monthly Rent: " << fav->Monthly_Rent << endl;
            cout << "Location: " << fav->Location << endl;
            cout << "Property Type: " << fav->Property_Type << endl;
            cout << "Size :" << fav->Size << endl;
            cout << "----------------------" << endl << endl;
            break;
        }
        fav = fav->next;
    }

    count++;
}

if (count == 0) {
    cout << "No favorite properties found." << endl;
}
```

Figure 29: Generate Top 10 Favourite Property Function Part 2

The outer for loop is to keep track how many properties need to be display, as the condition is to break the loop once the count reach 10. Hence, it will only display top 10 properties. In the while loop, the ads id is matched in TenantFavList and pair (propertyCountPairs), the property information will be displayed. The iteration continues until the count reach 10. If the count is equal to zero, there is no favorite property found.

Manage Tenancy Process

```
123  void manageTenancyProcess(TenantFavList* favorites) {
124          cout << "Properties with requested status:" << endl;
125          TenantFavList* fav = favorites; // Use for find requested payment status
126          TenantFavList* temp = favorites; // Use for find the given ads_id
127          bool pendingPropertiesExist = false;
128          // Display property info that is only with payment status requested
129          while (fav != nullptr) {
130              if (fav->Payment_Status == "Requested") {
131                  pendingPropertiesExist = true;
132                  cout << "----------------------" << endl;
133                  cout << "User Id: " << fav->User_Id << endl;
134                  cout << "Ads Id: " << fav->Ads_Id << endl;
135                  cout << "Prop Name: " << fav->Prop_Name << endl;
136                  cout << "Completion year: " << fav->Property_Type << endl;
137                  cout << "Monthly Rent: " << fav->Monthly_Rent << endl;
138                  cout << "Location: " << fav->Location << endl;
139                  cout << "Property Type: " << fav->Property_Type << endl;
140                  cout << "Size :" << fav->Size << endl << endl;
141                  cout << "Payment Status: " << fav->Payment_Status << endl;
142                  cout << "----------------------" << endl << endl;
143              }
144              fav = fav->next ;
145          }
```

Figure 30: Manage Tenancy Function Part 1

The manageTenancyProcess accepts one pointer to the favorites from TenantFavList linked list. The TenantFavList linked list is passed to two different pointer which are fav and temp. Fav will be used for finding the requested payment status and temp for finding the given ads id. A Boolean of pendingPropertiesExist is initialised as false. Then, a while loop is implemented to find the node in the fav linked list for the payment status is equal to requested and display to the manager. The pendingPropertiesExist will be updated to true if found any property info related to payment status equal to requested.

```
148         if (!pendingPropertiesExist) {
149             cout << "No properties are currently in requested status." << endl;
150             return;
151         }
152
153         //Select the property and user
154         int adsIdToProcess;
155         cout << "Enter Ads Id to process rent request: ";
156         cin >> adsIdToProcess;
157
158         string userid;
159         cout << "Enter User Id to process rent request: ";
160         cin >> userid;
161
162         bool propertyApproved = false;
163         while (temp != nullptr) {
164             if (temp->Ads_Id == adsIdToProcess && temp->Payment_Status == "Approved") {
165                 propertyApproved = true;
166                 break;
167             }
168             temp = temp->next;
169         }
```

Figure 31: Manage Tenancy Function Part 2

If the pendingPropertiesExist is equal to false, a message is displayed to the manager and exit the function. If the pendingPropertiesExist is true, the manager need to enter adsIdProcess and userId to update the payment status for the specific property. The Boolean of propertyApproved is initialised as false. Then, by using temp linked list, if the adsIdProcess and userId match any nodes in the temp, the propertyApproved boolean is updated to true.

```
temp = favorites; // reset pointer
bool propertyFound = false;
while (temp != nullptr) {
    if (temp->Ads_Id == adsIdToProcess && temp->User_Id == userid && temp->Payment_Status == "Requested") {
        cout << "Rent request received for property: " << endl;
        cout << "Ads Id: " << temp->Ads_Id << endl;
        cout << "Prop Name: " << temp->Prop_Name << endl;
        cout << "Completion year: " << temp->Property_Type << endl;
        cout << "Monthly Rent: " << temp->Monthly_Rent << endl;
        cout << "Location: " << temp->Location << endl;
        cout << "Property Type: " << temp->Property_Type << endl;
        cout << "Size :" << temp->Size << endl << endl;
        cout << "Payment Status: " << temp->Payment_Status << endl << endl;

        cout << "Do you want to approve or deny the request? (Enter 'approve' or 'deny'): ";
        string response;
        cin >> response;

        if (response == "approve" && propertyApproved == false) {
            temp->Payment_Status = "Approved";
            cout << "Rent request approved." << endl;
            cout << "New payment status: " << temp->Payment_Status << endl << endl;
        }
        else if (response == "deny") {
            temp->Payment_Status = "Denied";
            cout << "Rent request denied." << endl;
            cout << "New payment status: " << temp->Payment_Status << endl << endl;
        }
        else if (propertyApproved == true){
            cout << "The property is appointed by other tenant." << endl;
        }
        else {
            cout << "Invalid response. Please try again." << endl;
        }
        propertyFound = true;
        break;
    }
    temp = temp->next;
}
```

Figure 32: Manage Tenancy Function Part 3

The temp pointer is reset. The Boolean of propertyFound is initialised as false. The property information will be displayed if the adsIdToProcess and userid matched the manager entered and payment status is equal to requested in any of the nodes of the TenantFavList. Then, a message to let manager update the payment status is displayed. Based on the manager response, if it is approve and the boolean of propertyApproved is false, then the payment status is updated. If it is deny, the payment status is updated to denied. If the system found the propertyApproved is true, an error message is displayed else invalid response error message is prompted to the manager.

```
    //No property found in the fav list of user
    if (!propertyFound) {
        cout << "Property with Ads_Id " << adsIdToProcess << " of user id " << userid << " not found or has no rent request." << endl << endl;
    }
```

Figure 33: Manage Tenancy Function Part 4

Lastly, if the propertyFound is false, the error message regarding the manager entered adsIdToProcess and userid is prompted.

2.3 System Input/Output Screen

2.3.1 Login Page

```
Welcome To Private Accommodation Rent System
1. Login
2. Register Tenant
3. Exit
Please enter your choice: 1
User ID: u456
Password: securepass
Login Successful!
```

Figure 34: Login Correct Output

This is the login menu, which allows users to login, register, or exit the program. Press 1 to login into the system. Then input the correct user id and password to login.

```
Welcome To Private Accommodation Rent System
1. Login
2. Register Tenant
3. Exit
Please enter your choice: 1
User ID: testing
Password: testing
Login Failed. Invalid credentials or user not active.
```

Figure 35: Login Incorrect Output

This is an example of insert wrong user id or password.

```
Welcome To Private Accommodation Rent System
1. Login
2. Register Tenant
3. Exit
Please enter your choice: 2
User ID: tenant123
Password: tenant123
Username: Anson
Email: anson@example.com
Register Successfully.
```

Figure 36: Register Successfully Output

This is the register function which allow users to create an account. But this function is already set to register for tenant only. So, it can’t be register as admin or manager.

2.3.2 Sample Output (Tenant)

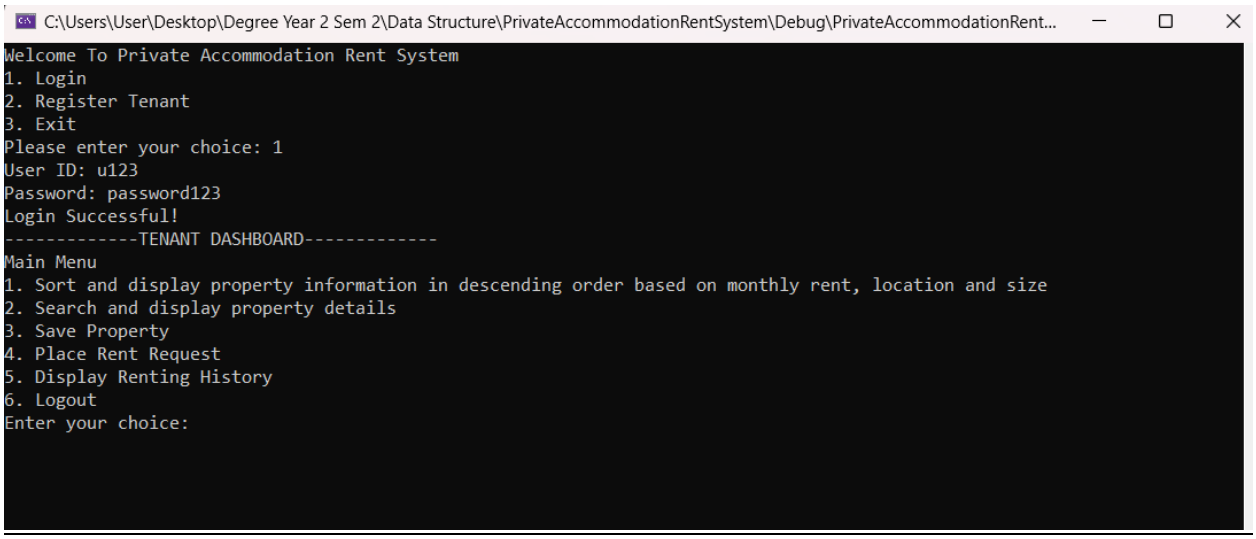


Figure 37: Tenant Dashboard Output

After the tenant logs in successfully, the tenant dashboard will show. 5 functions designed for the tenants, which are sorting properties by descending order based on monthly rent, location, and size, searching and displaying property details, saving favorite properties, placing rent requests from the favorites lists, and displaying renting history.

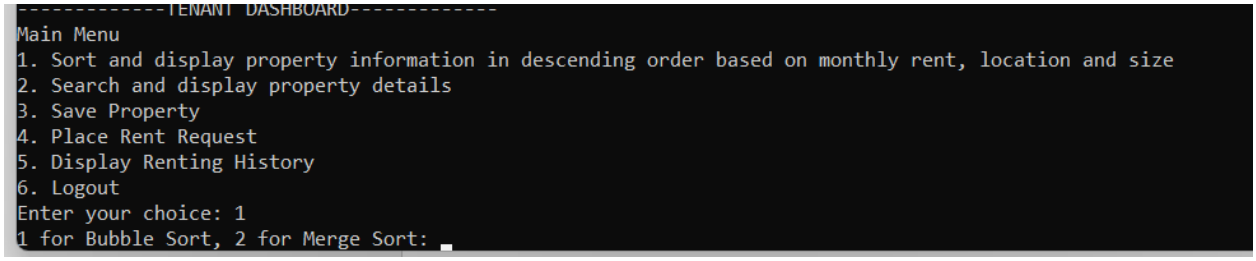


Figure 38: Sort Function Menu Output

When the tenants select the sort of function, there will be two sort functions for them to choose from. Both bubble sort and merge sort can be used. The reason for building these two features is to compare the completion time for both features and make some analysis.

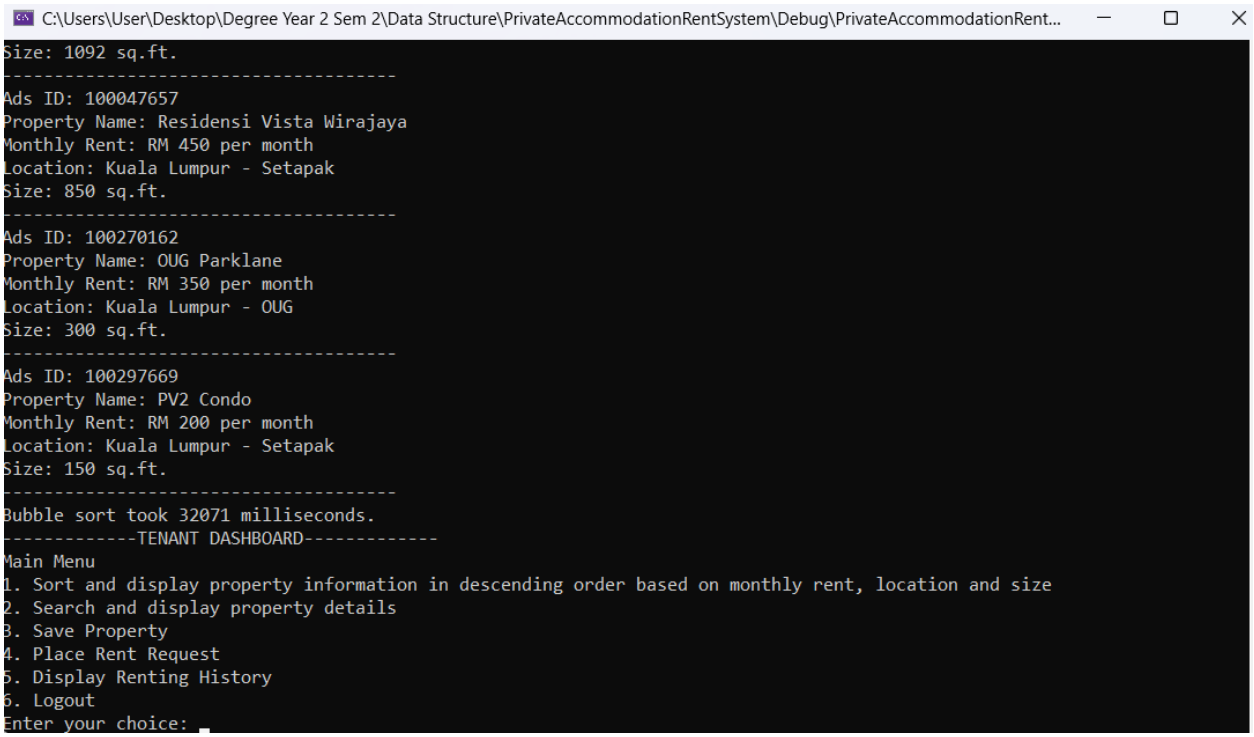


Figure 39: Bubble Sort Output

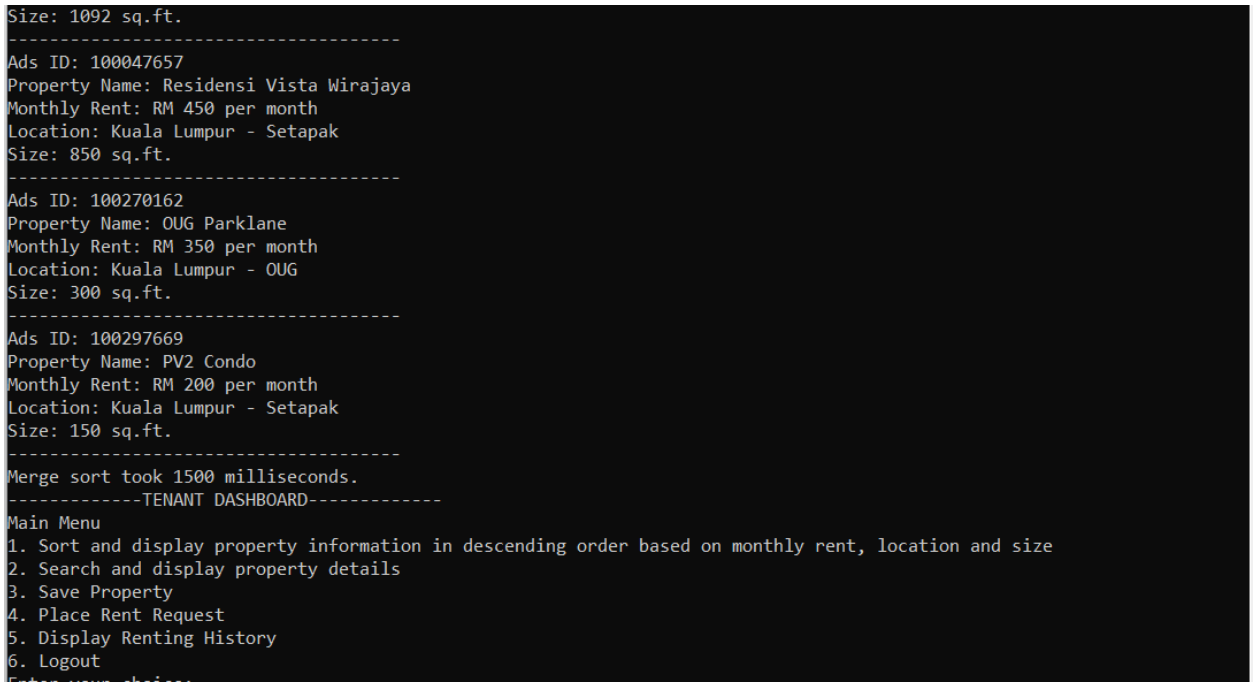


Figure 40: Merge Sort Output

In this sort of function, I am just taking 1000 data for sorting because the original data files contain too large of data which around 20000 data and using bubble sort function it is too slow. To prove this, I had used ChatGPT to calculate the estimated time to completing sorting 20000 data, and the time that calculated by the AI is around 111 hours. The time taken to complete

the bubble sort is 32071 milliseconds while in merge sort it takes only 1500 milliseconds to complete the sorting of 1000 data.

```
-------------TENANT DASHBOARD-------------
Main Menu
1. Sort and display property information in descending order based on monthly rent, location and size
2. Search and display property details
3. Save Property
4. Place Rent Request
5. Display Renting History
6. Logout
Enter your choice: 2
1 for Linear Search, 2 for Binary Search Tree:
```

Figure 41: Search Function Menu Output

Inside the search function, there will be two algorithms to choose from for searching. 1 is for linear search while 2 is for binary search tree algorithm.

```
-------------TENANT DASHBOARD-------------
Main Menu
1. Sort and display property information in descending order based on monthly rent, location and size
2. Search and display property details
3. Save Property
4. Place Rent Request
5. Display Renting History
6. Logout
Enter your choice: 2
1 for Linear Search, 2 for Binary Search Tree: 1
Enter property ID to search: 100741868

Property found!
Ads ID: 100741868
Property Name: LakeFront Residence
Location: Selangor - Cyberjaya
Monthly Rent: RM 1 800 per month
Time taken by linear search property: 829200 nanoseconds
-------------TENANT DASHBOARD-------------
```

Figure 42: Linear Search Output

```
-------------TENANT DASHBOARD-------------
Main Menu
1. Sort and display property information in descending order based on monthly rent, location and size
2. Search and display property details
3. Save Property
4. Place Rent Request
5. Display Renting History
6. Logout
Enter your choice: 2
1 for Linear Search, 2 for Binary Search Tree: 2
Enter property ID to search: 100741868
Time taken by binary search tree property: 200 nanoseconds
Time taken by binary search tree property: 300 nanoseconds
Time taken by binary search tree property: 200 nanoseconds
Time taken by binary search tree property: 300 nanoseconds
Time taken by binary search tree property: 400 nanoseconds
Property ID: 100741868
Property Name: LakeFront Residence
Location: Selangor - Cyberjaya
Monthly Rent: RM 1 800 per month
```

Figure ~~41~~43: Binary Search Tree Output

As can see, searching for the same result took two different times. In linear search, it took 829200 nanoseconds while in binary search tree, it only took around 300 nanoseconds to find the data.

```
-------------TENANT DASHBOARD-------------
Main Menu
1. Sort and display property information in descending order based on monthly rent, location and size
2. Search and display property details
3. Save Property
4. Place Rent Request
5. Display Renting History
6. Logout
Enter your choice: 3
Enter property ID to add into favourite list: 100827372
Property with Ads_Id 100827372 Added

Favorite property saved:
Ads Id: 100827372
Property Name: Canopy Hills
Completion Year: Condominium
Monthly Rent: RM 1 300 per month
Location: Selangor - Kajang
Property Type: Condominium
Size :600 sq.ft.
```

Figure 44: Save Favourite Property Output

When the tenants select 3, the system will require tenants to enter their favorite property. After inserting the property, it will show that the property with its id had been added and the property's details will show.

```
Ads Id: 100321976
Prop Name: Putra Villa
Completion year: Condominium
Monthly Rent: RM 1 800 per month
Location: Kuala Lumpur - Setapak
Property Type: Condominium
Size :1066 sq.ft.
Payment Status:

Ads Id: 100827372
Prop Name: Canopy Hills
Completion year: Condominium
Monthly Rent: RM 1 300 per month
Location: Selangor - Kajang
Property Type: Condominium
Size :600 sq.ft.
Payment Status:

Enter Ads Id of the property from favorite list to rent:
```

Figure 45: Place Rent Request Output

After the tenants select 4 to place rent requests, the properties that had been saved by the tenants will be displayed. From the lists displayed, tenants just need to enter the property ID to make a rent request.

```
Enter Ads Id of the property from favorite list to rent: 100321976
Rent request has been placed for property:
Ads Id: 100321976
Prop Name: Putra Villa
Completion year: Condominium
Monthly Rent: RM 1 800 per month
Location: Kuala Lumpur - Setapak
Property Type: Condominium
Size :1066 sq.ft.
Payment Status: Requested
```

Figure 46: Payment Status Changed Output

After the tenants inserted the property ID, the payment status will be updated from blank to the word “Requested” which needs the manager to approve this payment request.

```
------------TENANT DASHBOARD------------
Main Menu
1. Sort and display property information in descending order based on monthly rent, location and size
2. Search and display property details
3. Save Property
4. Place Rent Request
5. Display Renting History
6. Logout
Enter your choice: 5
Renting History
User Id: u123
Ads Id: 100321976
Property Name: Putra Villa
Completion year: Condominium
Monthly Rent: RM 1 800 per month
Location: Kuala Lumpur - Setapak
Property Type: Condominium
Size :1066 sq.ft.
Payment Status: Requested
```

Figure 47: Display Renting History Output

Tenants can view their renting history by selecting 5. The rental history of the tenants will show. In this section, only payments with status will be displayed.

2.3.3 Sample Output (Manager)

Display Registered Tenant's Detail

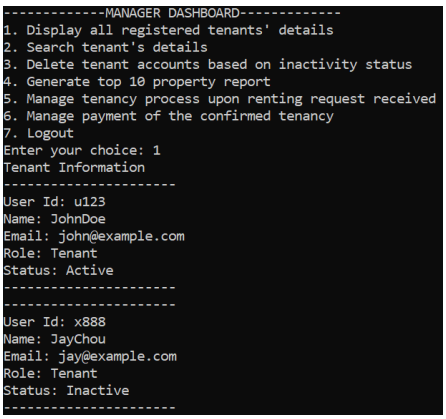


Figure 48: Display Registered Tenant Details Output

By entering 1, the manager able to view all the details of registered tenants. The tenant information is displayed as shown in the figure above.

Search Tenant's Details

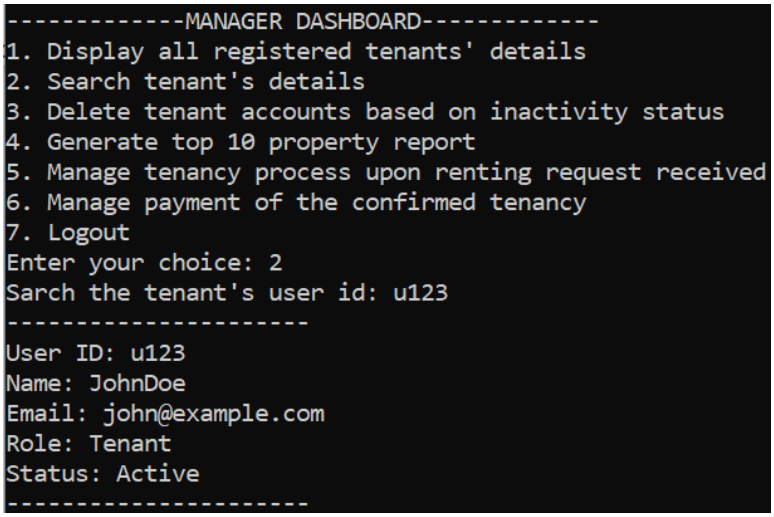


Figure 49: Search Tenant Details Correct Output

By entering 2, the manager is prompted to search the tenant information using user ID. If the user id searched is matched, the tenant information is displayed.

```
-------------MANAGER DASHBOARD-------------
1. Display all registered tenants' details
2. Search tenant's details
3. Delete tenant accounts based on inactivity status
4. Generate top 10 property report
5. Manage tenancy process upon renting request received
6. Manage payment of the confirmed tenancy
7. Logout
Enter your choice: 2
Sarch the tenant's user id: w
Record not found!
```

Figure 50: Search Tenant Details Incorrect Output

If the manager searches the user id that doesn't exist, the record not found is displayed to the manager.

Delete Tenant Accounts Based on Inactivity Status

```
------------MANAGER DASHBOARD------------
1. Display all registered tenants' details
2. Search tenant's details
3. Delete tenant accounts based on inactivity status
4. Generate top 10 property report
5. Manage tenancy process upon renting request received
6. Manage payment of the confirmed tenancy
7. Logout
Enter your choice: 3
Tenant Inactive Account
1. user id: x888
Delete the tenant's inactive account using user id: x888
Account is deleted successfully!
```

Figure 51: Delete Tenant Accounts Output

By entering 3, the manager is prompted to enter the user id of the tenant account which is inactive, he/she wishes to delete. A message "Account is deleted successfully!" is displayed upon successful deletion.

```
-------------MANAGER DASHBOARD-------------
1. Display all registered tenants' details
2. Search tenant's details
3. Delete tenant accounts based on inactivity status
4. Generate top 10 property report
5. Manage tenancy process upon renting request received
6. Manage payment of the confirmed tenancy
7. Logout
Enter your choice: 3
Tenant Inactive Account
No Inactive Tenant Account
```

Figure 52: Delete Tenant Accounts Fail Output

If there is no inactive tenant account, the message "No Inactive Tenant Account" is displayed.

Generate Top 10 Favourite Property Report

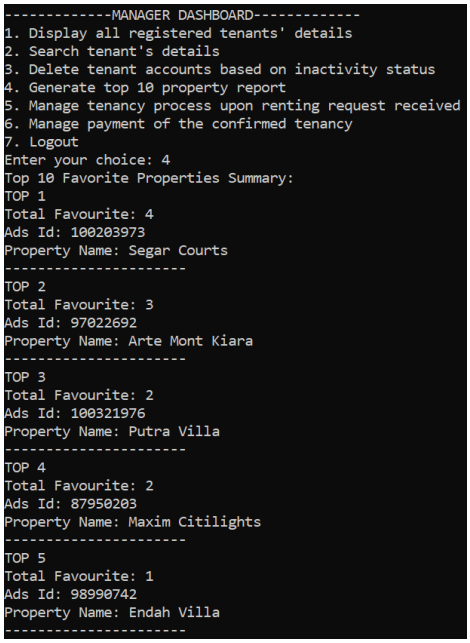


Figure 53: Generate Top 10 Favourite Property Output Part 1

Figure 54: Generate Top 10 Favourite Property Output Part 2

By entering 4, the manager able to view the top 10 most favourite property among the tenants. The top 10 property report is generated as the figure above.

Manage Tenancy Process

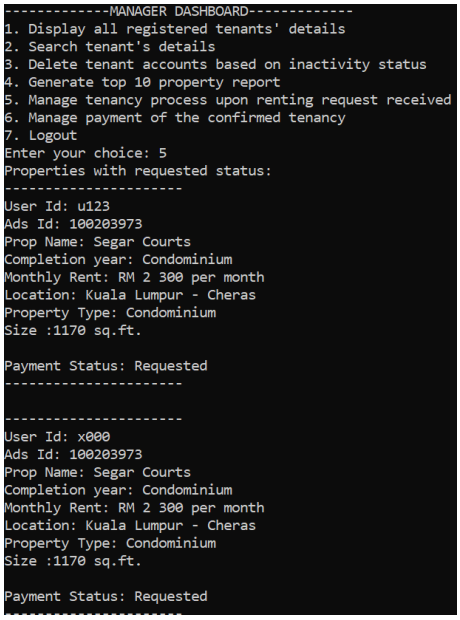


Figure 55: Manage Tenancy Output Part 1

By entering 5, the manager able to view the property which its payment status is requested from the tenants.

```
Enter Ads Id to process rent request: 100203973
Enter User Id to process rent request: x000
Rent request received for property:
Ads Id: 100203973
Prop Name: Segar Courts
Completion year: Condominium
Monthly Rent: RM 2 300 per month
Location: Kuala Lumpur - Cheras
Property Type: Condominium
Size :1170 sq.ft.

Payment Status: Requested

Do you want to approve or deny the request? (Enter 'approve' or 'deny'): approve
Rent request approved.
New payment status: Approved
```

Figure 56: Manage Tenancy Output Part 2

Then, the manager is required to enter the ads id and user id to manage the tenancy process of the property. After that, the manager can enter approve or deny for the request. In the figure above, the manager approve the property, the payment status of the property changed to approved.

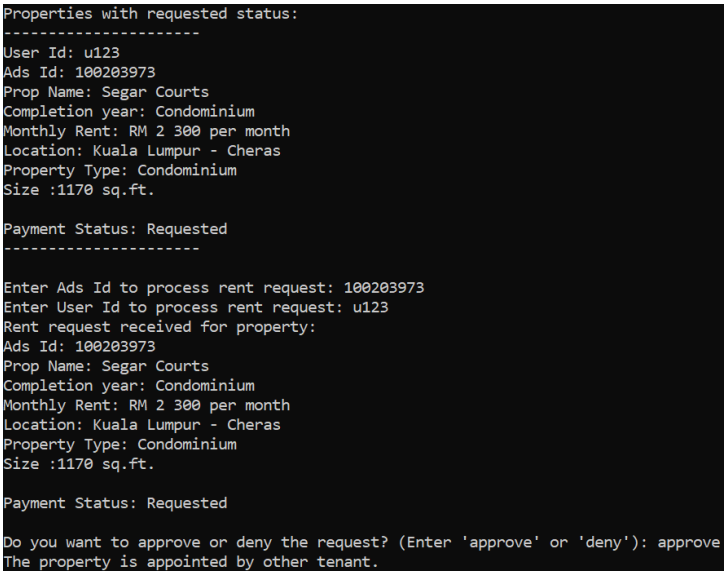


Figure 57: Manage Tenancy Output Part 3

For the property had been approved, the second request for the same property is only can be denied. If the manager update the payment status of the property which is approved previously for other tenant, the message “The property is appointed by other tenant” is displayed. The manager only can enter deny in this scenario.

Manage Payment For Confirmed Tenancy

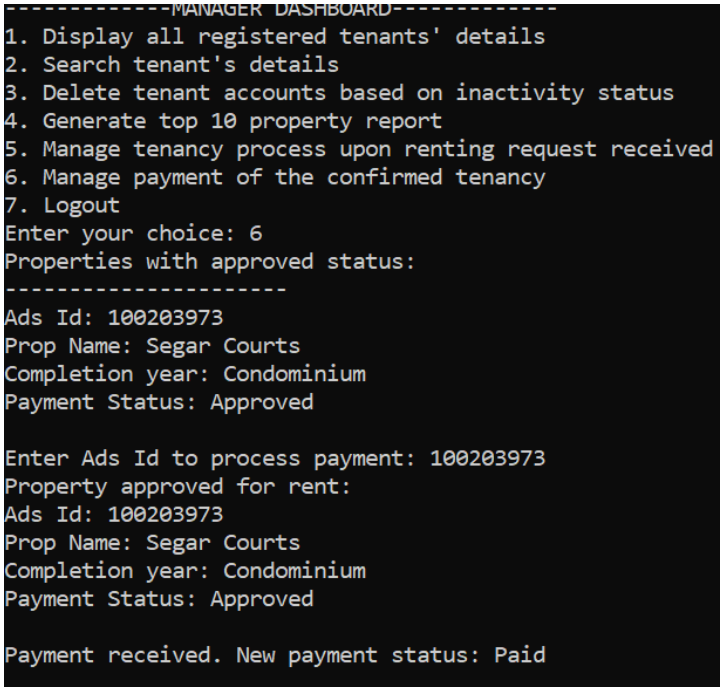


Figure 58: Manage Payment Output Part 1

By entering 6, the manager able to update the payment status to paid after received payment from tenant. The property information of payment status with approved is displayed. The manager need to enter the Ads Id to update the payment status to paid as shown in the figure above.

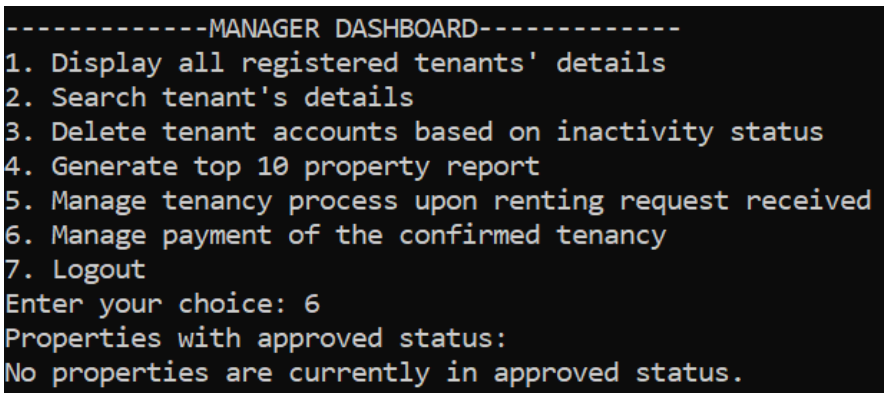


Figure 59: Manage Payment Output Part 2

If there is no property in approved status, the message “No properties are currently in approved status is displayed to the manager.

Logout

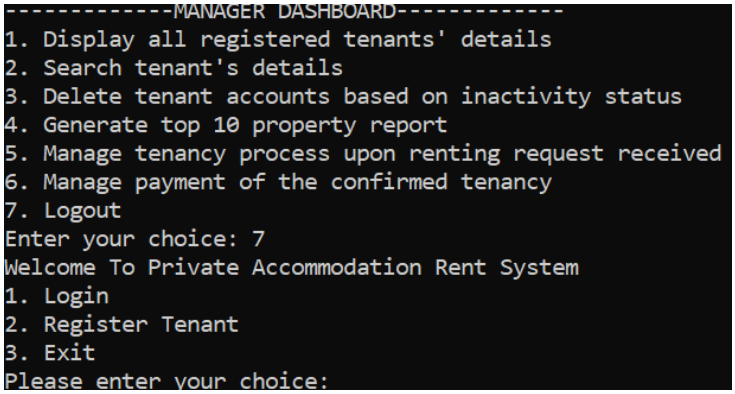


Figure 60: Logout Output

By entering 7, the manager able to logout to the main menu of the system as shown in the figure above.

2.3.3 Sample Output (Admin)

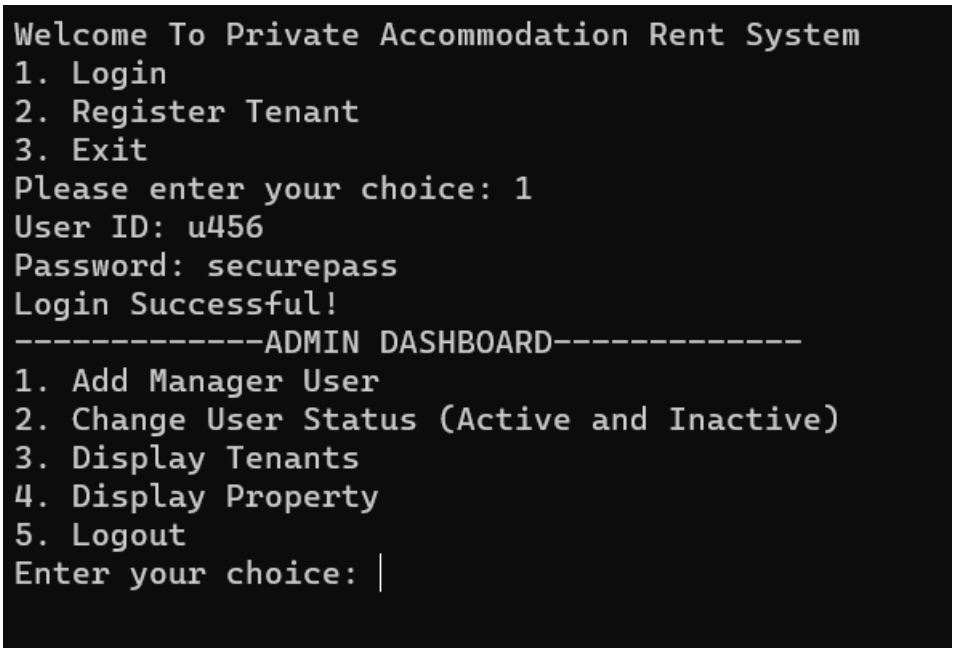


Figure 61: Admin Dashboard Output

This is the admin dashboard, it shows the main menu of admin. This admin dashboard lets user type in a choice to choose which function want to use. There have 5 options to choose which is add manager, change user status, display tenants, display property, and logout.

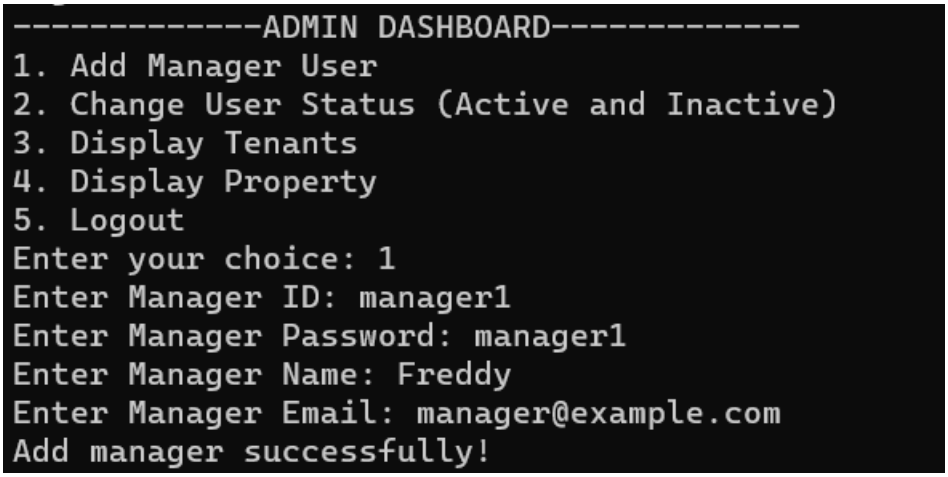


Figure 62: Add Manager Success Output

The is the add manager user function which used in manager role. After user input 1 at the admin dashboard, the system will require user to input the manager id, password, name, and email address. After that it will add the manager account into the linked list.

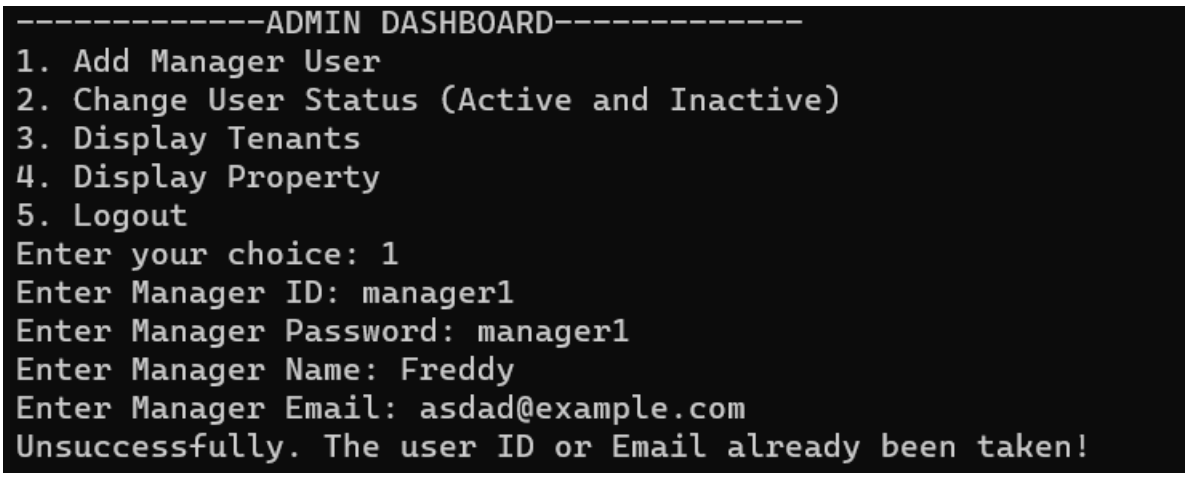


Figure 63: Add Manager Fail Output

This is also the add manager function but insert with existing manager id. If user input existing manager id, it will display a error message told user that user id or email have been taken.

```
-------------ADMIN DASHBOARD-------------
1. Add Manager User
2. Change User Status (Active and Inactive)
3. Display Tenants
4. Display Property
5. Logout
Enter your choice: 2
----------------------
User ID: u123
Name: JohnDoe
Email: john@example.com
Role: Tenant
Status: Active
----------------------
----------------------
User ID: u456
Name: JaneSmith
Email: june@example.com
Role: Admin
Status: Active
----------------------
----------------------
User ID: u789
Name: BobJohnson
Email: bob@example.com
Role: Manager
Status: Active
----------------------
----------------------
```

Figure 64: Change User Status Output Part 1

Back to admin menu, choosing 2 for using change user status. Then it will show all the user information such as user id, name, email, role, and status.

```
User ID: manager1
Name: Freddy
Email: manager@example.com
Role: Admin
Status: Active
----------------------
Which user you want to change the status?: manager1
You want change to? (Active or Inactive): Inactive
User Status Changed Successfully.
Change Successfully
```

Figure 65: Change User Status Output Part 2

This is when admin change the user status. Admins need to key in the user id and update the user status.

```
-------------ADMIN DASHBOARD-------------
1. Add Manager User
2. Change User Status (Active and Inactive)
3. Display Tenants
4. Display Property
5. Logout
Enter your choice: 3
Press 1 for view ACTIVE tenant, Press 2 for view INACTIVE tenant : 1
User ID: u123
Name: JohnDoe
Email: john@example.com
Role: Tenant
Status: Active
----------------------
User ID: u999
Name: Natalie
Email: natalie@example.com
Role: Tenant
Status: Active
----------------------
User ID: x000
Name: Nash
Email: nash@example.com
Role: Tenant
Status: Active
----------------------
```

Figure 66: Display Active Tenant Output

```
-------------ADMIN DASHBOARD-------------
1. Add Manager User
2. Change User Status (Active and Inactive)
3. Display Tenants
4. Display Property
5. Logout
Enter your choice: 3
Press 1 for view ACTIVE tenant, Press 2 for view INACTIVE tenant : 2
User ID: x888
Name: JayChou
Email: jay@example.com
Role: Tenant
Status: Inactive
----------------------
```

Figure 67: Display Inactive Tenant Output

At the admin dashboard input 3 for using the display tenant function. This function using filter function to filter user status. Press 1 for the active user and press 2 for the inactive user.

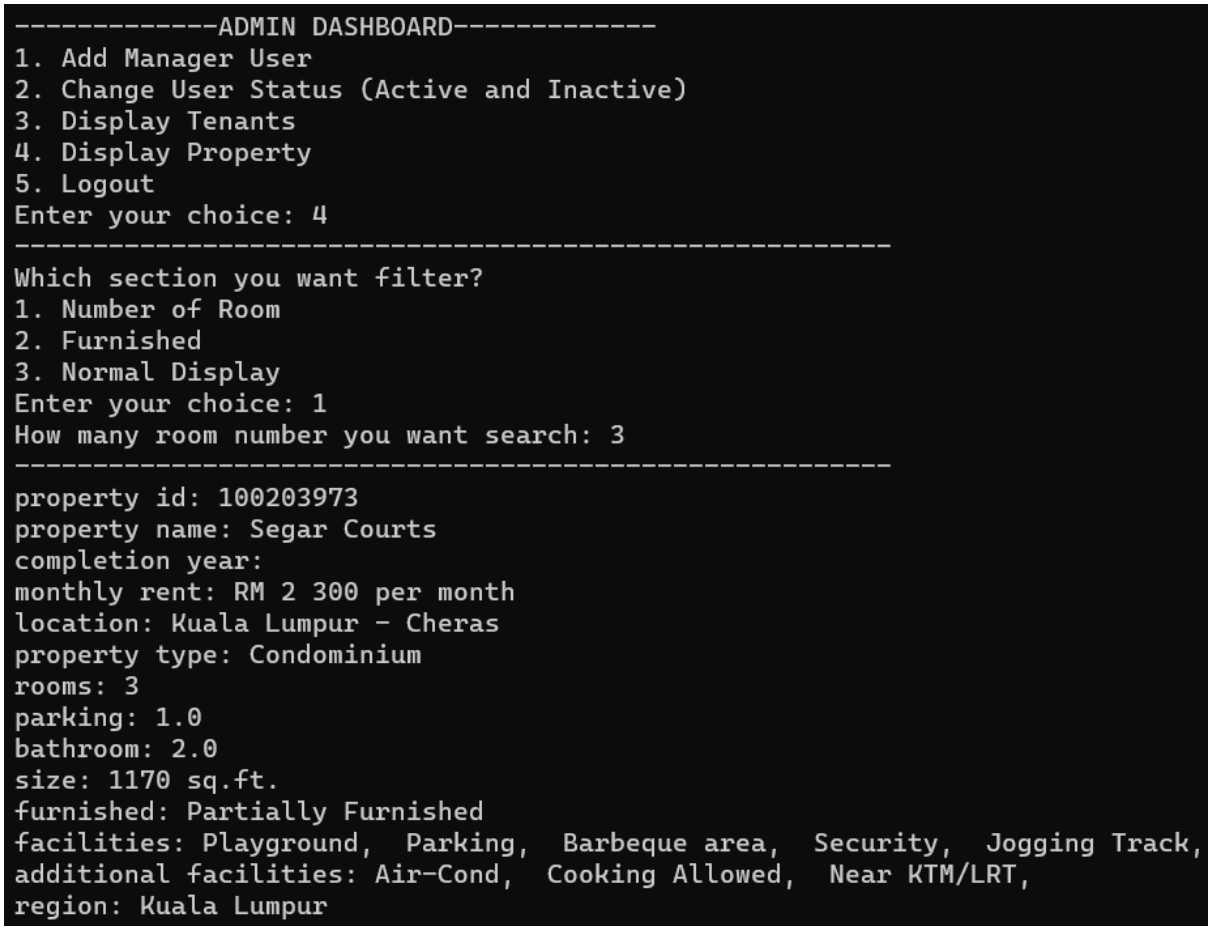


Figure 68: Display Number of Room Output

This is the display properties function, when user press 4 on the admin dashboard, it will lead admin to display properties function which able to view by different criteria. Admin may view by room number like example inside picture. This function allow the system to display 5 records of property.

Figure 69: Display Number of Room Selection Output

At the end of the display function, the system allow user to view the next 5 records, previous 5 records, or exit this function.

```
Which section you want filter?
1. Number of Room
2. Furnished
3. Normal Display
Enter your choice: 2
Which furnished you want to search? (1. Fully, 2. Partially, or 3. Not Furnished): 1
-----------------------------------------------------
property id: 100323185
property name: The Hipster @ Taman Desa
completion year: 2022.0
monthly rent: RM 4 200 per month
location: Kuala Lumpur - Taman Desa
property type: Condominium
rooms: 5
parking: 2.0
bathroom: 6.0
size: 1842 sq.ft.
furnished: Fully Furnished
facilities: Minimart,  Gymnasium,  Security,  Playground,  Swimming Pool,  Parking,  Lift,
 hall,  Jogging Track,
additional facilities: Air-Cond,  Cooking Allowed,  Washing Machine,
region: Kuala Lumpur
-----------------------------------------------------
```

Figure 70: Display Furnished Output

This function is same with the last one but this one is to display furnished. This function has 3 choices to choose which is fully furnished, partially furnished, and not furnished. This example is press 1 so it will only print fully furnished property.

```
Which section you want filter?
1. Number of Room
2. Furnished
3. Normal Display
Enter your choice: 3
-----------------------------------------------------
property id: 100323185
property name: The Hipster @ Taman Desa
completion year: 2022.0
monthly rent: RM 4 200 per month
location: Kuala Lumpur - Taman Desa
property type: Condominium
rooms: 5
parking: 2.0
bathroom: 6.0
size: 1842 sq.ft.
furnished: Fully Furnished
facilities: Minimart,  Gymnasium,  Security,  Playground,  Swimming Pool,  Parking,  L
 hall,  Jogging Track,
additional facilities: Air-Cond,  Cooking Allowed,  Washing Machine,
region: Kuala Lumpur
-----------------------------------------------------
```

Figure 70: Display Norma Display Output

This figure shown a normal display property function which means that not apply any filter on it.

2.4 Execution time between linear search and binary search tree

	Linear Search (nanoseconds)	Binary Search Tree (nanoseconds)
First time	3330100	1809200
Second time	4659900	2581000
Third time	2962000	2078700
Fourth time	3666000	2780700
Fifth time	4199300	4532500

Table 1: Execution Time for First User

	Linear Search (nanoseconds)	Binary Search Tree (nanoseconds)
First time	3814700	2315400
Second time	5562800	1510800
Third time	6608800	937100
Fourth time	2666900	1591800
Fifth time	6354000	1449000

Table 2: Execution Time for Second User

In this algorithm, the first user is using 16GB ram and using intel 7 10th gen as the computer system while the second user is using 16GB ram and intel 7 12th gen. In this case, we are trying to analyse whether will the algorithm affect the searching speed even using the same memory but different processors. The average execution time for first user in linear search is 3763460 nanoseconds which is around 3.73 milliseconds while the average time for second user is 5001440 nanoseconds which around 5 milliseconds. For the binary search tree, the average execution time for first user is 2756420 nanoseconds which around 2.8 milliseconds, the average time for second user is 1560820 nanoseconds which is 1.56 milliseconds. From the above data, it is obvious that the algorithms that implemented will affect the speed performance of searching. This is because the average case complexity of linear search is using the formula O(n) (Ravikiran, 2023) while the formula of binary search tree in average case complexity is O(logN) (Shukla, n.d.).

2.5 Execution time between bubble sort and merge sort

	Bubble Sort (milliseconds)	Merge Sort (milliseconds)
First time	30292	2343
Second time	30310	1430
Third time	34578	547
Fourth time	31657	646
Fifth time	29944	559

Table 3: Execution Time for First User

	Bubble Sort (Milliseconds)	Merge Sort (Milliseconds)
First Time	31195	3693
Second Time	27290	3786
Third Time	26869	3654
Fourth Time	25559	3782
Fifth Time	30560	3550

Table 4: Execution Time for Second User

In the first user, the computer hardware that been used is 16GB ram and using intel 7 10th gen and the average time is around 31356.2 milliseconds, and the average time of merge sort is 1105 milliseconds. While for the second user is using 8GB ram and using intel 5 10th gen while the average time of bubble sort is around 28294.6 milliseconds and merge sort are 3693 milliseconds. Based on the above comparison, it is obvious that merge sort algorithm is faster than the bubble sort algorithm as both users having the result 1105 milliseconds and 3693 milliseconds which quicker than bubble sort algorithm that around 31356.2 milliseconds and 28294.6 milliseconds. In addition, in this sort algorithm, the data that been used to analyse is around 1000 data only as the bubble sort will take much longer time that around 100 hours to do the sorting for 20000 data that provided and the calculation was calculated by the AI model, ChatGPT. By this a conclusion can made, the sorting algorithm will be affected by the users' ram and processors (Amini, 2021).

	Merge Sort (Milliseconds)
First Time	16882
Second Time	12499
Third Time	35698
Fourth Time	22408
Fifth Time	24952

Table 5: Execution Time for Merge Sort of 20000 Data

As the previous bubble sort algorithm result shown average around 28000 milliseconds to 31300 milliseconds needed just to sort 1000 data. By converting it to seconds, it utilised 28 seconds to 31.3 seconds to sort 1000 data. However, the merge sort algorithm just utilised 1 second to 3 seconds in sorting 1000 data. This clearly shown that the bubble sort algorithm is not efficient in sorting the larger data and the difference of sorting speed can be 10x times slower than merge sort.

To prove that the merge sort algorithm is capable in managing larger data. An experiment to sort around 20000 data using merge sort algorithm has been carried out. The device specification and criteria to sort is same as second user in the previous discussion. The result stated that the merge sort algorithm average runtime to sort is 22488 milliseconds which is 22.5 seconds in converting to seconds. This clearly proved that the merge sort is more efficient compared bubble sort algorithm. The result in the previous part for merge sort is proved as the best case of sorting time of merge sort algorithm in sorting 20 multiple larger data which is 1*20 = 20 seconds, which the result is almost the same as the result below. Merge sort is one of the most efficient sorting algorithms due to its work with the principle of divide and conquer. This principle is to break down the list into multiple sub-lists until each sub-lists contain of a single element to sort them based on the criteria (Pankaj, 2022). The time complexity of merge sort can be calculated by O(n log n) and bubble sort can be calculated by O(n^2). By comparing this 2 algorithm's time complexity, the merge sort is more favourable as the efficient sorting algorithm than bubble sort.

3.0 Conclusion

3.1 Limitation

The rent system currently has several limitations that could be affect user experience or the efficiency. The first limitation is the input handling vulnerability which is the system struggles with strings containing spaces. For example, when users input names like “Anson Lim”, the space is mistakenly interpreted as an instruction to move to the next input line. This limitation only allows users input one word in every input function, this may lead challenges for users who need to input multiple word in a string or sentences. The second limitation is the efficiency of the CSV loading. The system's data loading process, specifically when reading from a CSV file, exhibits noticeable delays even with a dataset of 19,991 entries. In scenarios where companies or users handling millions of data, it could need to wait a huge amount of time to startup or data retrieval. The third limitation is the int data type that let user input may entered infinite loop even the error handling using if-else statement is made. This is due to the input stream goes into the failed state and not properly cleared. The last limitation is the user interface (UI). It being limited due to using a command-line format, and not meet modern user expectations. Lack of graphical elements and interactive prompts could make the system less user friendly. It should implement the graphical user interface for enhancing the user experience.

3.2 Future Work based on System Limitations

The limitation related to strings spacing input can be resolved by using getline along with the cin, this is because the getline function able to read the entire line including spaces but the cin will stop after a space is entered. Next, the loading time for reading csv data can be reduced by trying to read the data chunks by chunks instead of line by line. Some studies can be carried out to learn any method to read data concurrently in C++ in the future. For the user interface limitation, the UI can be enhanced in C++ using GUI libraries in the future. For example, Qt or GTK libraries. These libraries provide extensive tool to create graphical windows, buttons, text field and etc. This allow users to interact with the application visually. Finally, the int infinite loop issue can be resolved by implementing the string instead of int and convert them into int for the system to perform any necessary action.

3.3 Experience and Feedback

After using C++ to code a data structure program, we only understand that the algorithms that been implemented in the system will affect the time taken to show the results. This means that if the system used a bad algorithms in the system, it could take much of time for it to display the output. Besides, C++ is able to direct access to the memory with using pointers. Pointers is a feature that use in C++ that allow for creation and management of dynamic data structures, for example linked list and trees. Moreover, there are many libraries that pre-built for the data structures for example chrono. Chrono is a library that we used in this system to count the execution time and display it at the sort and search functions.

3.4 Assumption

1. Each property that payment status is approved, other tenant requested the same property, the manager only can update them to deny.
2. All tenants are able to save the same property.
3. The tenants' favourites properties list will only be display when the tenants had saved the property.
4. Tenants can only request for the payment status from their favourite properties list only.
5. The admin able to update all the user status.
6. Only admin can be adding manager account.
7. Admin account cannot be add, it already has set inside the system.
8. Managers update the property payment status to paid after received tenant transaction from other medium.
9. User id is the unique attribute for user account information.
10. Top 10 properties report is generated for manager based on the number of properties in favourite list among tenants.

4.0 References

Amini, P. (5 August, 2021). *5 Factors to Consider Before Choosing a Sorting Algorithm*. Retrieved from Medium: https://towardsdatascience.com/5-factors-to-consider-before-choosing-a-sorting-algorithm-5b079db7912c

Pankaj. (4 August, 2022). *DigitalOcean*. Retrieved from Merge Sort Algorithm - Java, C, and Python Implementation: https://www.digitalocean.com/community/tutorials/merge-sort-algorithm-java-c-python

Ravikiran, A. S. (27 July, 2023). *What is Linear Search Algorithm | Time Complexity*. Retrieved from Simplilearn: https://www.simplilearn.com/tutorials/data-structure-tutorial/linear-search-algorithm#what_is_a_linear_search_algorithm

Shukla, S. (n.d.). *Time and Space complexity of Binary Search Tree (BST)*. Retrieved from OpenGenus: https://iq.opengenus.org/time-and-space-complexity-of-binary-search-tree/

5.0 Appendix

5.1 Runtime Result Screenshots

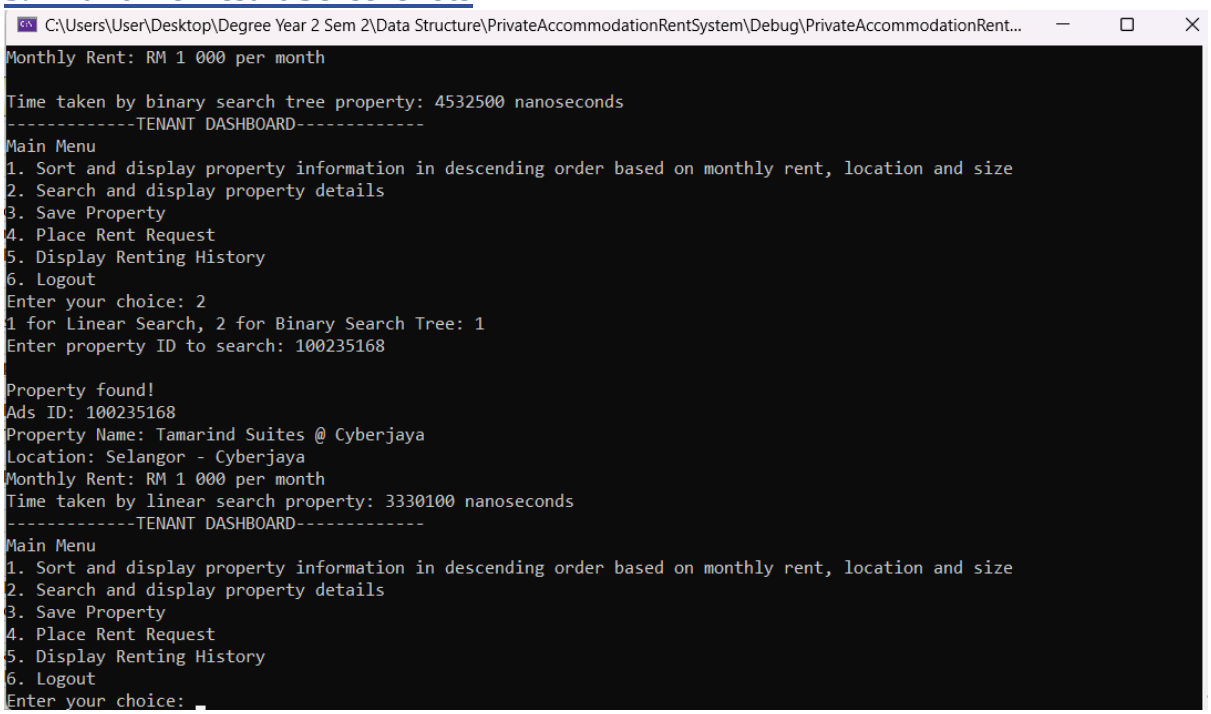


Figure 71: Linear Search 1st Attempt

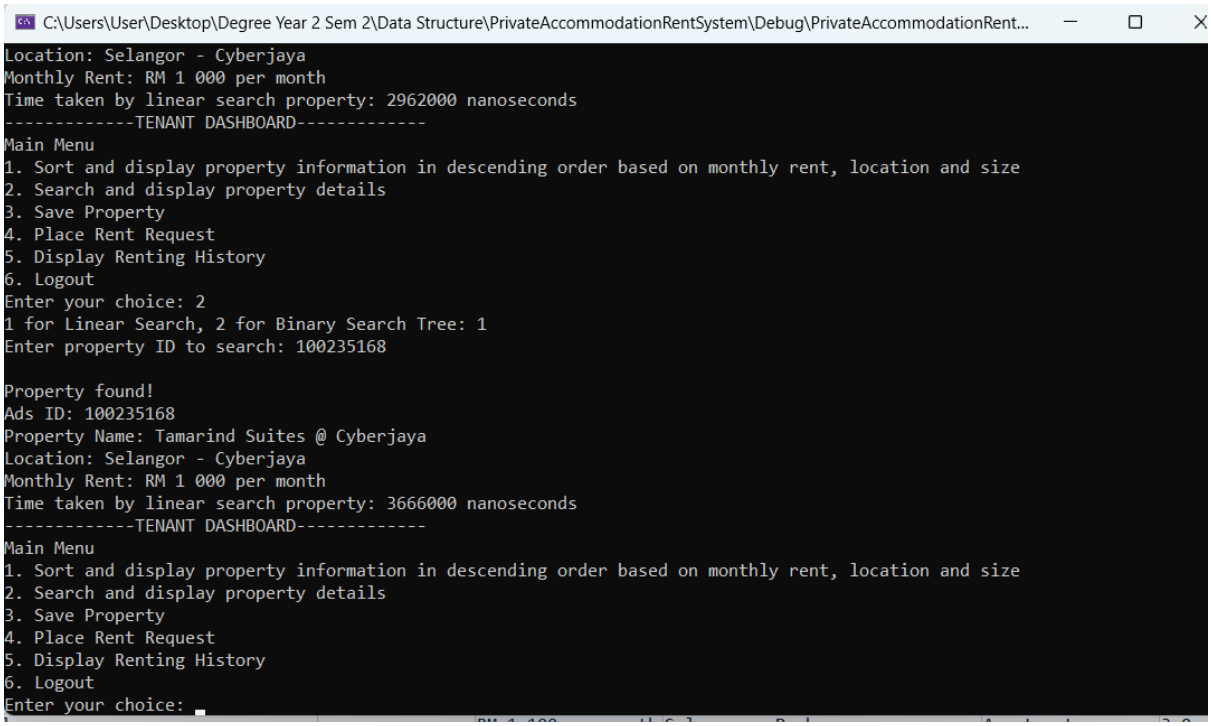


Figure 72: Linear Search 2nd Attempt

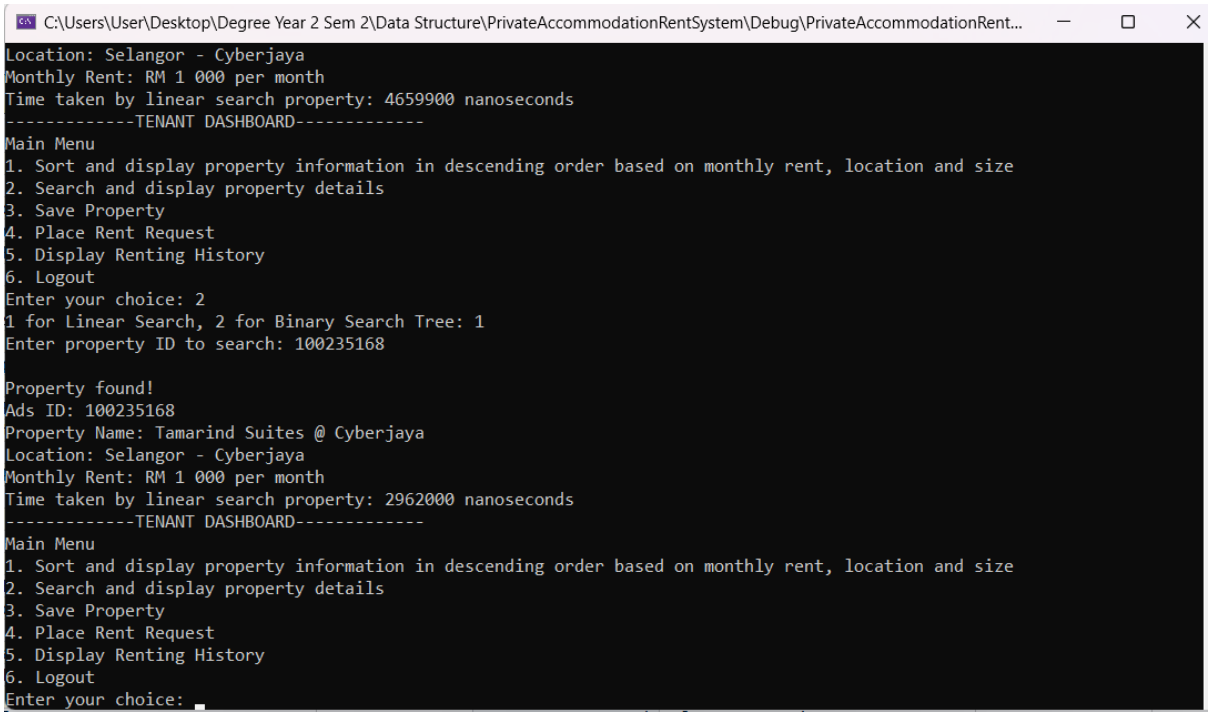


Figure 73: Linear Search 3rd Attempt

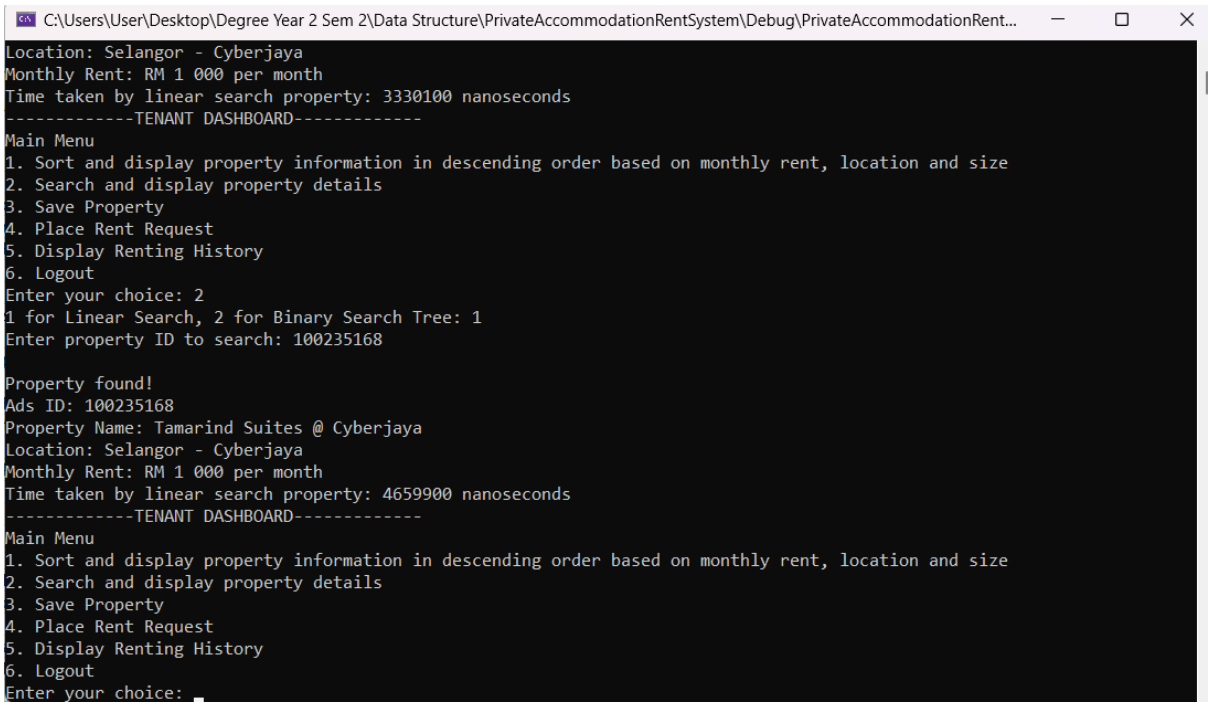


Figure 74: Linear Search 4th Attempt

```
Location: Selangor - Cyberjaya
Monthly Rent: RM 1 000 per month
Time taken by linear search property: 1593600 nanoseconds
------------TENANT DASHBOARD-------------
Main Menu
1. Sort and display property information in descending order based on monthly rent, location and size
2. Search and display property details
3. Save Property
4. Place Rent Request
5. Display Renting History
6. Logout
Enter your choice: 2
1 for Linear Search, 2 for Binary Search Tree: 1
Enter property ID to search: 100235168

Property found!
Ads ID: 100235168
Property Name: Tamarind Suites @ Cyberjaya
Location: Selangor - Cyberjaya
Monthly Rent: RM 1 000 per month
Time taken by linear search property: 4199300 nanoseconds
------------TENANT DASHBOARD-------------
Main Menu
1. Sort and display property information in descending order based on monthly rent, location and size
2. Search and display property details
3. Save Property
4. Place Rent Request
5. Display Renting History
6. Logout
Enter your choice:
```

Figure 75: Linear Search 5th Attempt

```
------------TENANT DASHBOARD-------------
Main Menu
1. Sort and display property information in descending order based on monthly rent, location and size
2. Search and display property details
3. Save Property
4. Place Rent Request
5. Display Renting History
6. Logout
Enter your choice: 2
1 for Linear Search, 2 for Binary Search Tree: 2
Enter property ID to search: 100235168
Property ID: 100235168
Property Name: Tamarind Suites @ Cyberjaya
Location: Selangor - Cyberjaya
Monthly Rent: RM 1 000 per month

Time taken by binary search tree property: 1809200 nanoseconds
------------TENANT DASHBOARD-------------
Main Menu
1. Sort and display property information in descending order based on monthly rent, location and size
2. Search and display property details
3. Save Property
4. Place Rent Request
5. Display Renting History
6. Logout
Enter your choice:
```

Figure 76: Binary Search Tree 1st Attempt

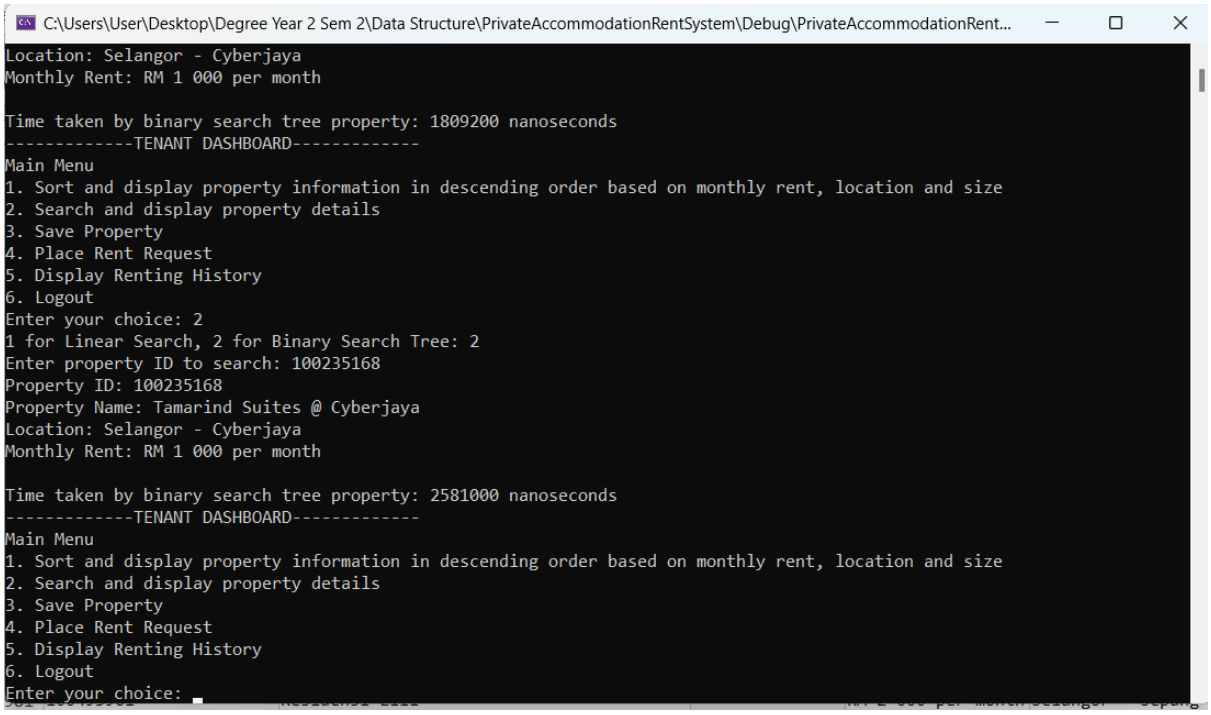


Figure 77: Binary Search Tree 2nd Attempt

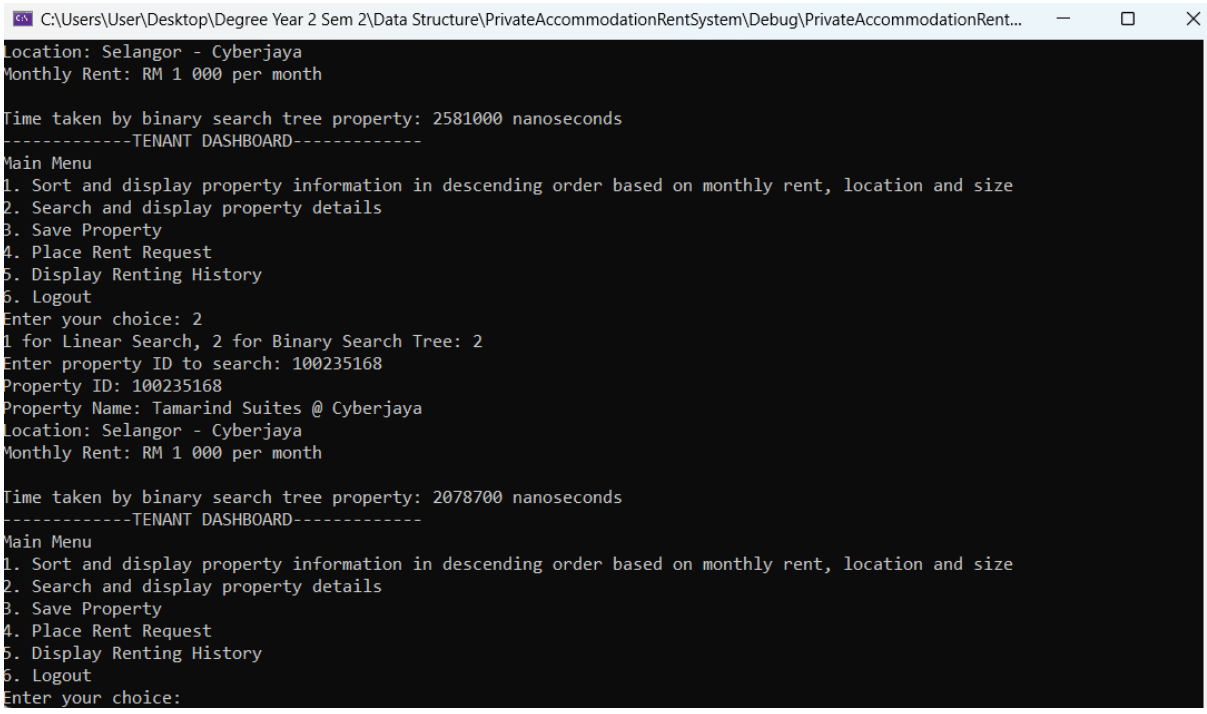


Figure 78: Binary Search Tree 3rd Attempt

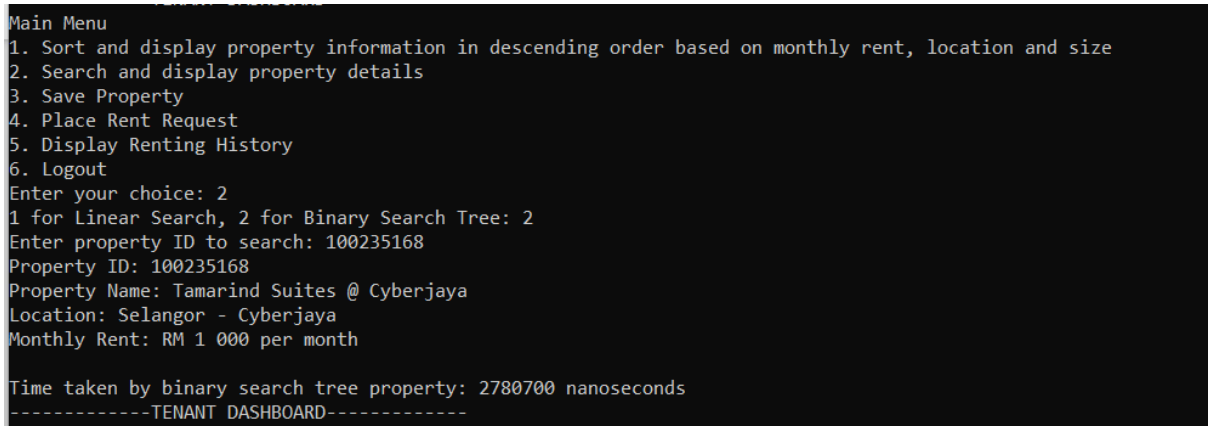


Figure 79: Binary Search Tree 4th Attempt

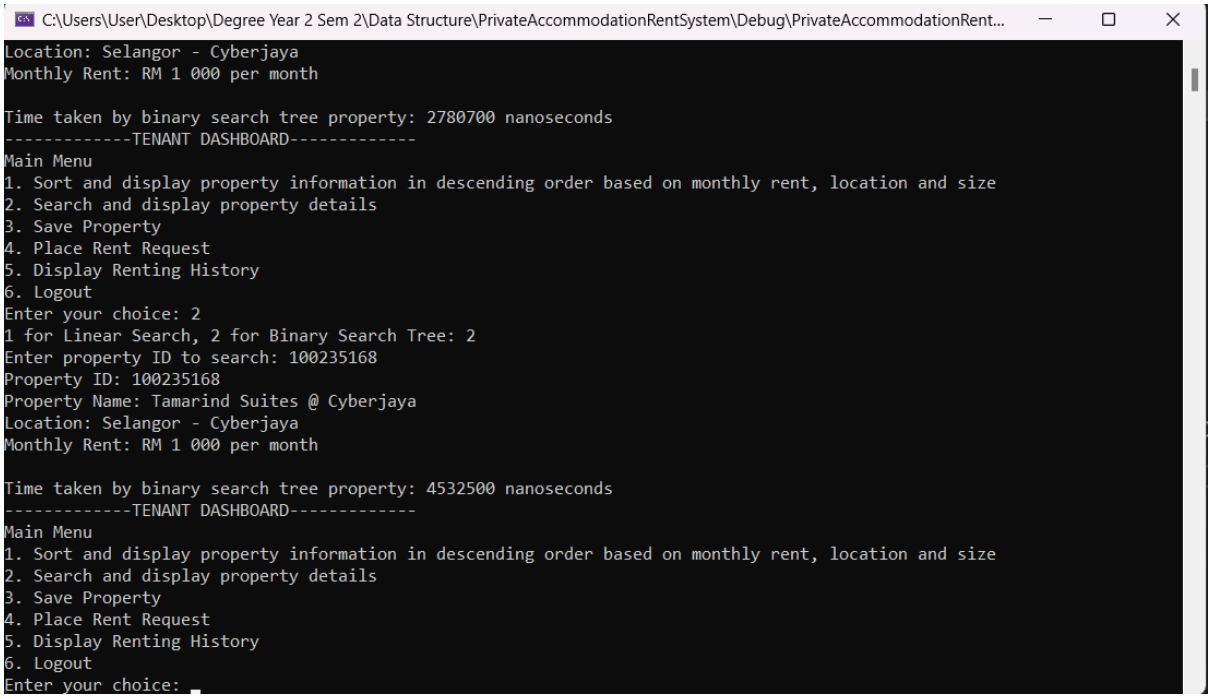


Figure 80: Binary Search Tree 5th Attempt

```
-------------------------------------
Ads ID: 100297669
Property Name: PV2 Condo
Monthly Rent: RM 200 per month
Location: Kuala Lumpur - Setapak
Size: 150 sq.ft.
-------------------------------------
Bubble sort took 30292 milliseconds.
-------------TENANT DASHBOARD-------------
Main Menu
1. Sort and display property information in descending order based on monthly rent, location and size
2. Search and display property details
3. Save Property
4. Place Rent Request
5. Display Renting History
6. Logout
Enter your choice:
```

Figure 81: Bubble Sort 1st Attempt

```
-------------------------------------
Ads ID: 100297669
Property Name: PV2 Condo
Monthly Rent: RM 200 per month
Location: Kuala Lumpur - Setapak
Size: 150 sq.ft.
-------------------------------------
Bubble sort took 30130 milliseconds.
-------------TENANT DASHBOARD-------------
Main Menu
1. Sort and display property information in descending order based on monthly rent, location and size
2. Search and display property details
3. Save Property
4. Place Rent Request
5. Display Renting History
6. Logout
Enter your choice:
```

Figure 82: Bubble Sort 2nd Attempt

```
-------------------------------------
Ads ID: 100297669
Property Name: PV2 Condo
Monthly Rent: RM 200 per month
Location: Kuala Lumpur - Setapak
Size: 150 sq.ft.
-------------------------------------
Bubble sort took 34578 milliseconds.
-------------TENANT DASHBOARD-------------
Main Menu
1. Sort and display property information in descending order based on monthly rent, location and size
2. Search and display property details
3. Save Property
4. Place Rent Request
5. Display Renting History
6. Logout
Enter your choice:
```

Figure 83: Bubble Sort 3rd Attempt

```
Size: 150 sq.ft.
-------------------------------------
Bubble sort took 31657 milliseconds.
-------------TENANT DASHBOARD-------------
Main Menu
1. Sort and display property information in descending order based on monthly rent, location and size
2. Search and display property details
3. Save Property
4. Place Rent Request
5. Display Renting History
6. Logout
Enter your choice:
```

Figure 84: Bubble Sort 4th Attempt

```
-------------------------------------
Bubble sort took 29944 milliseconds.
-------------TENANT DASHBOARD-------------
Main Menu
1. Sort and display property information in descending order based on monthly rent, location and size
2. Search and display property details
3. Save Property
4. Place Rent Request
5. Display Renting History
6. Logout
Enter your choice:
```

Figure 85: Bubble Sort 5th Attempt

```
-------------------------------------
Ads ID: 100323128
Property Name: Pangsapuri Teratak Muhibbah 2
Monthly Rent: RM 1 000 per month
Location: Kuala Lumpur - Taman Desa
Size: 650 sq.ft.
-------------------------------------
Merge sort took 2343 milliseconds.
-------------TENANT DASHBOARD-------------
Main Menu
1. Sort and display property information in descending order based on monthly rent, location and size
2. Search and display property details
3. Save Property
4. Place Rent Request
5. Display Renting History
6. Logout
Enter your choice:
```

Figure 86: Merge Sort 1st Attempt

```
--------------------------------------
Ads ID: 100323128
Property Name: Pangsapuri Teratak Muhibbah 2
Monthly Rent: RM 1 000 per month
Location: Kuala Lumpur - Taman Desa
Size: 650 sq.ft.
--------------------------------------
Merge sort took 1430 milliseconds.
-------------TENANT DASHBOARD-------------
Main Menu
1. Sort and display property information in descending order based on monthly rent, location and size
2. Search and display property details
3. Save Property
4. Place Rent Request
5. Display Renting History
6. Logout
Enter your choice:
```

Figure 87: Merge Sort 2nd Attempt

```
--------------------------------------
Merge sort took 547 milliseconds.
-------------TENANT DASHBOARD-------------
Main Menu
1. Sort and display property information in descending order based on monthly rent, location and size
2. Search and display property details
3. Save Property
4. Place Rent Request
5. Display Renting History
6. Logout
Enter your choice:
```

Figure 88: Merge Sort 3rd Attempt

```
--------------------------------------
Merge sort took 646 milliseconds.
-------------TENANT DASHBOARD-------------
Main Menu
1. Sort and display property information in descending order based on monthly rent, location and size
2. Search and display property details
3. Save Property
4. Place Rent Request
5. Display Renting History
6. Logout
Enter your choice:
```

Figure 89: Merge Sort 4th Attempt

```
--------------------------------------
Merge sort took 559 milliseconds.
-------------TENANT DASHBOARD-------------
Main Menu
1. Sort and display property information in descending order based on monthly rent, location and size
2. Search and display property details
3. Save Property
4. Place Rent Request
5. Display Renting History
6. Logout
Enter your choice:
```

Figure 90: Merge Sort 5th Attempt

```
User ID: u123
Password: password123
Login Successful!
-------------TENANT DASHBOARD-------------
Main Menu
1. Sort and display property information in descending order based on monthly rent
2. Search and display property details
3. Save Property
4. Place Rent Request
5. Display Renting History
6. Logout
Enter your choice: 2
1 for Linear Search, 2 for Binary Search Tree: 1
Enter property ID to search: 100235168

Property found!
Ads ID: 100235168
Property Name: Tamarind Suites @ Cyberjaya
Location: Selangor - Cyberjaya
Monthly Rent: RM 1 000 per month
Time taken by linear search property: 3814700 nanoseconds
```

Figure 91: Linear Search 1st Attempt

```
Welcome To Private Accommodation Rent System
1. Login
2. Register Tenant
3. Exit
Please enter your choice: 1
User ID: u123
Password: password123
Login Successful!
-------------TENANT DASHBOARD-------------
Main Menu
1. Sort and display property information in descending order based on monthly rent, locati
2. Search and display property details
3. Save Property
4. Place Rent Request
5. Display Renting History
6. Logout
Enter your choice: 2
1 for Linear Search, 2 for Binary Search Tree: 1
Enter property ID to search: 100235168

Property found!
Ads ID: 100235168
Property Name: Tamarind Suites @ Cyberjaya
Location: Selangor - Cyberjaya
Monthly Rent: RM 1 000 per month
Time taken by linear search property: 5562800 nanoseconds
```

Figure 92: Linear Search 2nd Attempt

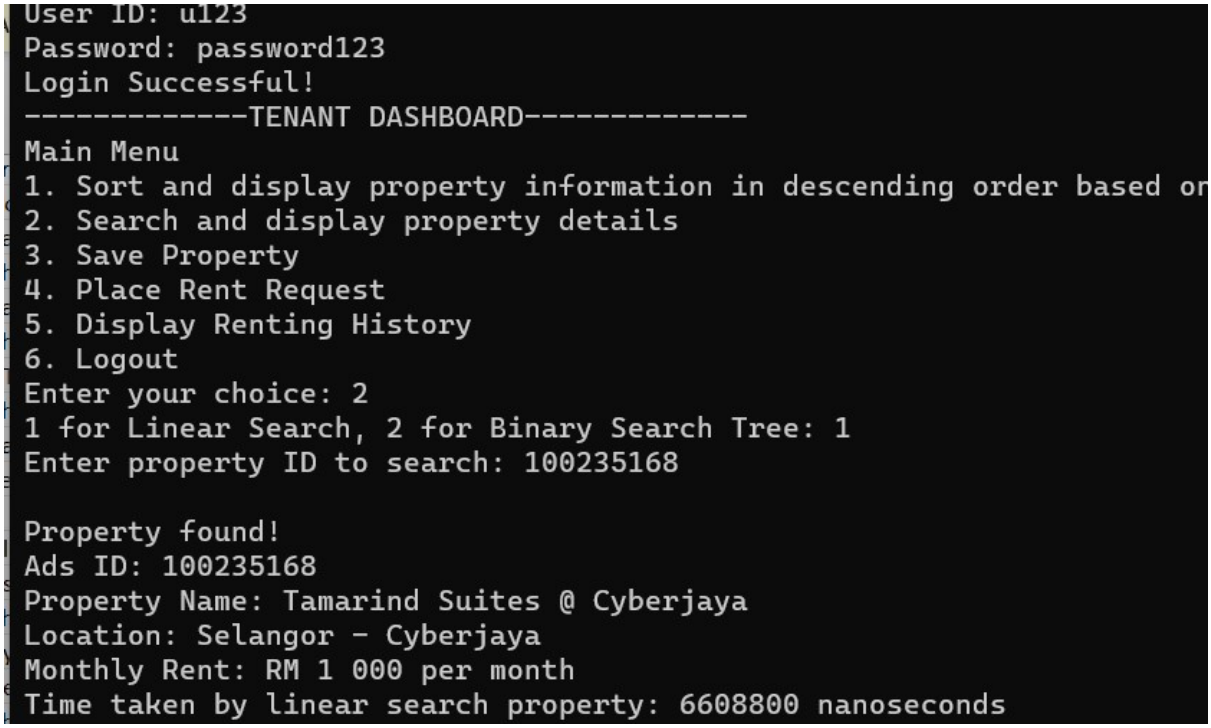


Figure 93: Linear Search 3rd Attempt

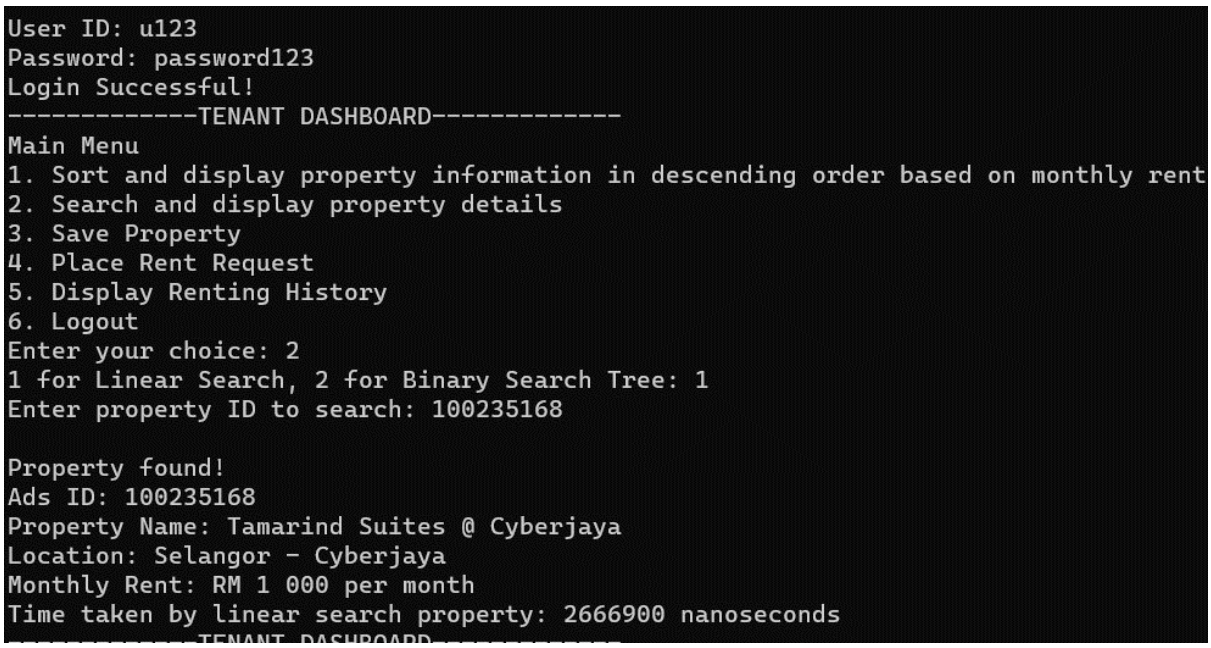


Figure 94: Linear Search 4th Attempt

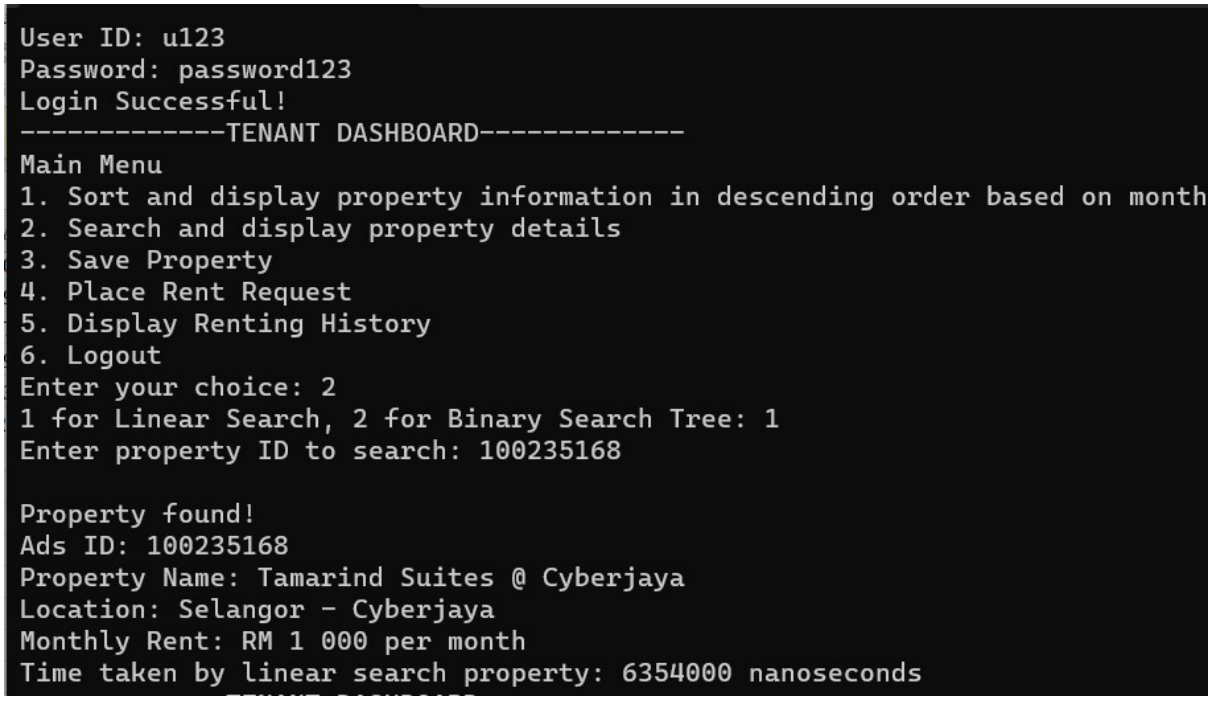


Figure 95: Linear Search 5th Attempt

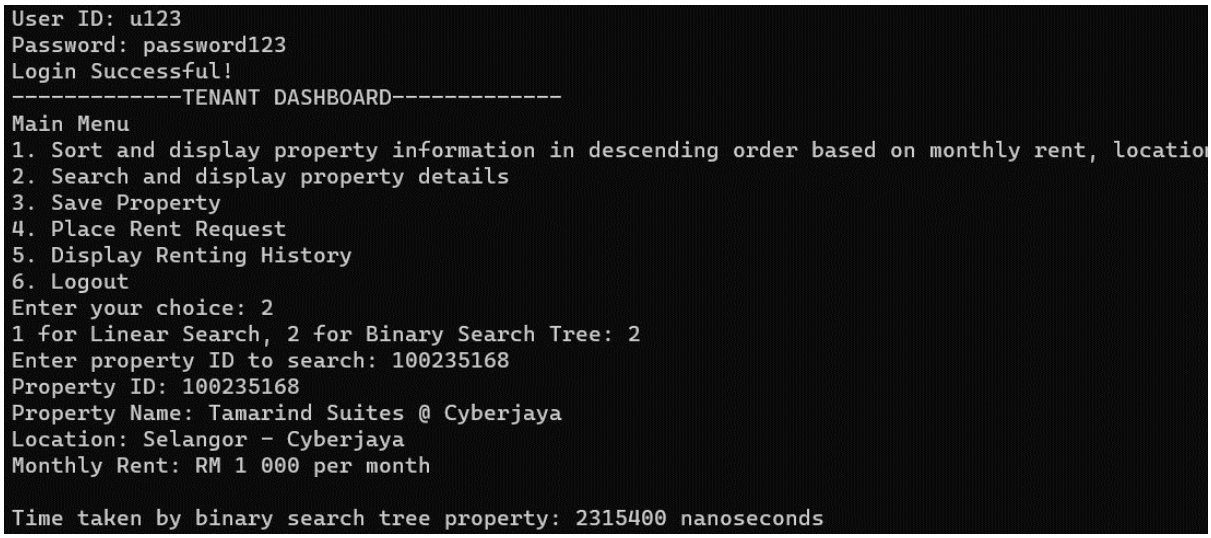


Figure 96: Binary Search Tree 1st Attempt

```
Please enter your choice: 1
User ID: u123
Password: password123
Login Successful!
-------------TENANT DASHBOARD-------------
Main Menu
1. Sort and display property information in descending order based on monthly rent, location and
2. Search and display property details
3. Save Property
4. Place Rent Request
5. Display Renting History
6. Logout
Enter your choice: 2
1 for Linear Search, 2 for Binary Search Tree: 2
Enter property ID to search: 100235168
Property ID: 100235168
Property Name: Tamarind Suites @ Cyberjaya
Location: Selangor - Cyberjaya
Monthly Rent: RM 1 000 per month

Time taken by binary search tree property: 1510800 nanoseconds
```

Figure 97: Binary Search Tree 2nd Attempt

```
Please enter your choice: 1
User ID: u123
Password: password123
Login Successful!
-------------TENANT DASHBOARD-------------
Main Menu
1. Sort and display property information in descending order based on monthly rent, location
2. Search and display property details
3. Save Property
4. Place Rent Request
5. Display Renting History
6. Logout
Enter your choice: 2
1 for Linear Search, 2 for Binary Search Tree: 2
Enter property ID to search: 100235168
Property ID: 100235168
Property Name: Tamarind Suites @ Cyberjaya
Location: Selangor - Cyberjaya
Monthly Rent: RM 1 000 per month

Time taken by binary search tree property: 937100 nanoseconds
```

Figure 98: Binary Search Tree 3rd Attempt

```
Please enter your choice: 1
User ID: u123
Password: password123
Login Successful!
-------------TENANT DASHBOARD-------------
Main Menu
1. Sort and display property information in descending order based on monthly rent, location
2. Search and display property details
3. Save Property
4. Place Rent Request
5. Display Renting History
6. Logout
Enter your choice: 2
1 for Linear Search, 2 for Binary Search Tree: 2
Enter property ID to search: 100235168
Property ID: 100235168
Property Name: Tamarind Suites @ Cyberjaya
Location: Selangor - Cyberjaya
Monthly Rent: RM 1 000 per month

Time taken by binary search tree property: 1591800 nanoseconds
```

Figure 99: Binary Search Tree 4th Attempt

```
Please enter your choice: 1
User ID: u123
Password: password123
Login Successful!
-------------TENANT DASHBOARD-------------
Main Menu
1. Sort and display property information in descending order based on monthly rent, location and si
2. Search and display property details
3. Save Property
4. Place Rent Request
5. Display Renting History
6. Logout
Enter your choice: 2
1 for Linear Search, 2 for Binary Search Tree: 2
Enter property ID to search: 100235168
Property ID: 100235168
Property Name: Tamarind Suites @ Cyberjaya
Location: Selangor - Cyberjaya
Monthly Rent: RM 1 000 per month

Time taken by binary search tree property: 1449000 nanoseconds
```

Figure 100: Binary Search Tree 5th Attempt

```
Bubble sort took 31195 milliseconds.
```

Figure 101: Bubble Sort 1st Attempt

```
Bubble sort took 27290 milliseconds.
```

Figure 102: Bubble Sort 2nd Attempt

```
Bubble sort took 26869 milliseconds.
```

Figure 103: Bubble Sort 3rd Attempt

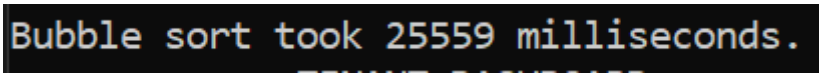


Figure 104: Bubble Sort 4th Attempt

Figure 105: Bubble Sort 5th Attempt

Figure 106: Merge Sort 1st Attempt

Figure 107: Merge Sort 2nd Attempt

Figure 108: Merge Sort 3rd Attempt

Figure 109: Merge Sort 4th Attempt

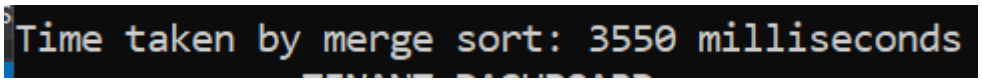


Figure 110: Merge Sort 5th Attempt

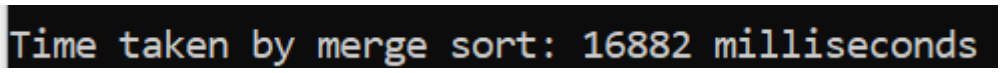


Figure 111: Merge Sort 1st Attempt (20k data)

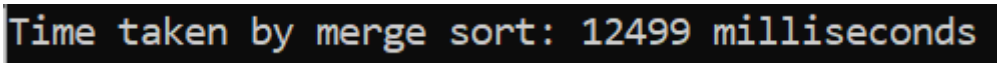


Figure 112: Merge Sort 2nd Attempt (20k data)

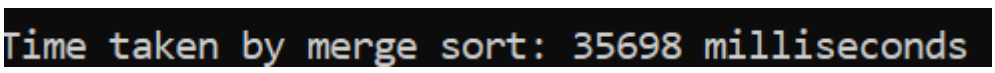


Figure 113: Merge Sort 3rd Attempt (20k data)

Figure 114: Merge Sort 4th Attempt (20k data)

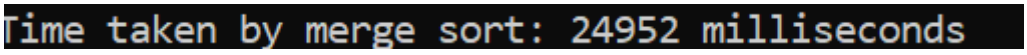


Figure 115: Merge Sort 5th Attempt (20k data)