**Data Structures**

**Assignment 01**

**Code:**

#include <iostream>

#include <fstream>

#include <chrono>

#include <ctime>

#include <string>

#include <utility> // for std::pair

using namespace std;

int get\_int() {

    int n;

    for (;;) {

        if (cin >> n) {

            return n;

        }

        cin.clear();

        cin.ignore(numeric\_limits<streamsize>::max(), '\n');

        cout << "Invalid entry. Please re-enter: ";

    }

}

string getCurrTime() {

    auto timenow = chrono::system\_clock::now();

    // Convert to time\_t to work with C-style time functions

    time\_t currentTime = chrono::system\_clock::to\_time\_t(timenow);

    string readableCurrTime = ctime(&currentTime);

    readableCurrTime.erase(readableCurrTime.length() - 1); // erase new line

    return readableCurrTime;

}

// --------------------------------------------------------------- Part 01: Tickets ---------------------------------------------------------------

class Ticket {

    private:

        int instanceTicketID; // unique id for each customer

        string customerName;

        int priority;

        string requestDescription;

        string ticketOpenTime;

        string ticketCloseTime;

        string status;

    public:

        static int numTickets; // static id counter

        Ticket () : customerName(""), priority(0), requestDescription(""), ticketOpenTime(""), ticketCloseTime(""), status("Open") {}

        Ticket (string customerName, int priority, string requestDescription) {

            instanceTicketID = ++numTickets;

            this->customerName = customerName;

            this->priority = priority;

            this->requestDescription = requestDescription;

            this->ticketOpenTime = getCurrTime();

            this->ticketCloseTime = "Active Currently";

            this->status = "Open";

        }

        // copy constructor for deep copy

        Ticket(const Ticket &other) : customerName(other.customerName), priority(other.priority), requestDescription(other.requestDescription), status(other.status),

            ticketOpenTime(other.ticketOpenTime), ticketCloseTime(other.ticketCloseTime), instanceTicketID(other.instanceTicketID) {}

        // copy assignment operator

        Ticket& operator=(const Ticket &other) {

            // Self-assignment check

            if (this == &other) {

                return \*this;

            }

            customerName = other.customerName;

            priority = other.priority;

            requestDescription = other.requestDescription;

            status = other.status;

            ticketOpenTime = other.ticketOpenTime;

            ticketCloseTime = other.ticketCloseTime;

            instanceTicketID = other.instanceTicketID;

            return \*this;

        }

        void initiateTicketID() {

            instanceTicketID = ++numTickets;

        }

        void setTicketID(int newID) {

            instanceTicketID = newID;

        }

        void setCustomerName(const std::string& name) {

            customerName = name;

        }

        void setPriority(int p) {

            priority = p;

        }

        void setStatusOpen() {

            status = "Open";

        }

        void setStatus(string newStatus) {

            status = newStatus;

        }

        void setRequestDescription(const std::string& description) {

            requestDescription = description;

        }

        void setTicketOpenTime() {

            ticketOpenTime = getCurrTime();

        }

        void setTicketCloseTime(string time) {

            ticketCloseTime = time;

        }

        int getInstanceTicketID() const {

            return instanceTicketID;

        }

        string getCustomerName() const {

            return customerName;

        }

        int getPriority() const {

            return priority;

        }

        string getRequestDescription() const {

            return requestDescription;

        }

        string getTicketOpenTime() const {

            return ticketOpenTime;

        }

        string getTicketCloseTime() const {

            return ticketCloseTime;

        }

        string getStatus() const {

            return status;

        }

        void printTicketDetails() {

            cout << "Ticket ID: " << getInstanceTicketID() << endl;

            cout << "Customer Name: " << getCustomerName()<< endl;

            cout << "Priority: " << getPriority() << endl;

            cout << "Support Request Description: " << getRequestDescription() << endl;

            cout << "Ticket Open Time: " << getTicketOpenTime()<< endl;

            cout << "Ticket Close Time: " << getTicketCloseTime() << endl;

            cout << "Status: " << getStatus() << endl << endl;

        }

};

int Ticket::numTickets = 0;

// This class stores all the tickets that were ever created. No matter if they are currently open, closed, resolved, etc.

class Node {

    public:

        Ticket ticket;

        Node \*next;

        Node () : next(NULL) {}

        Node (Ticket &newTicket) {

            ticket = newTicket;

            next = NULL;

        }

};

class TicketsList {

    public:

        Node \*head;

        Node \*tail;

        int numAllTickets;

        TicketsList () : head(NULL), tail(NULL), numAllTickets(0) {}

        void addTicket (Ticket& newTicket) {

            Node \*newnode = new Node(newTicket);

            numAllTickets++;

            if (head == NULL) {

                head = newnode;

                tail = head;

                tail->next = head;

            } else {

                Node \*temp = head;

                while (temp->next != head) {

                    temp = temp->next;

                }

                temp->next = newnode;

                tail = temp->next;

                tail->next = head;

            }

        }

        void displayTickets() {

            if (head == NULL) {

                cout<<"List is empty." << endl;

                return;

            }

            cout << "\_\_\_\_\_\_\_\_\_\_\_\_\_ Displaying All " << numAllTickets << " Tickets Created Today (Open/Closed) \_\_\_\_\_\_\_\_\_\_\_\_\_ " << endl;

            Node \*temp = head;

            do {

                temp->ticket.printTicketDetails();

                temp = temp->next;

            } while (temp != head);

        }

};

// -------------------------------------------------- Part 03 & 04: Ticket Resolution Log & Ticket Queue --------------------------------------------------

class NodeStack {

    public:

        Ticket ticket;

        NodeStack \*next;

        NodeStack (Ticket &newTicket) : next(NULL), ticket(newTicket) {}

};

class TicketStack {

    public:

        NodeStack \*top;

        int numProcessedTickets;

        TicketStack () : top(NULL), numProcessedTickets(0) {}

        void pushTicket (Ticket &incTicket) {

            numProcessedTickets++;

            if (top == NULL) {

                top = new NodeStack(incTicket);

                return;

            }

            NodeStack \*newnode = new NodeStack(incTicket);

            newnode->next = top;

            top = newnode;

        }

        void peekTicketStack () {

            if (top == NULL) {

                cout<<"No processed ticket as of now. Nothing to print." << endl;

                return;

            }

            cout << "\_\_\_\_\_\_\_\_\_\_\_\_\_ Displaying Most Recent Ticket Log \_\_\_\_\_\_\_\_\_\_\_\_\_ " << endl;

            top->ticket.printTicketDetails();

        }

        void printTicketStack () {

            if (top == NULL) {

                cout << "Cannot print ticket stack. No record in the processed tickets at the momment." << endl;

                return;

            }

            cout << "\_\_\_\_\_\_\_\_\_\_\_\_\_ Displaying " << numProcessedTickets << " Resolved Tickets \_\_\_\_\_\_\_\_\_\_\_\_\_ " << endl;

            NodeStack \*temp = top;

            while (temp != NULL) {

                temp->ticket.printTicketDetails();

                temp = temp->next;

            }

        }

};

class NodeQueue {

    public:

        Ticket ticket;

        NodeQueue \*next;

        NodeQueue (Ticket &newTicket) : ticket(newTicket), next(NULL) {}

};

class TicketQueue { // circular queue

    public:

        NodeQueue \*head;

        NodeQueue \*tail;

        int numPendingTickets;

        static int count;

        TicketQueue () : head(NULL), tail(NULL), numPendingTickets(0) {}

        void createTicket (TicketsList& TL) {

            Ticket newTicket;

            string name, req;

            int pr;

            newTicket.initiateTicketID();

            cout<<"Enter customer name: ";

            cin.ignore();

            getline(cin, name);

            newTicket.setCustomerName(name);

            cout<<"Enter priority: ";

            pr = get\_int();

            newTicket.setPriority(pr);

            cout<<"Enter Support Request Description: ";

            cin.ignore();

            getline(cin, req);

            newTicket.setRequestDescription(req);

            newTicket.setTicketOpenTime();

            newTicket.setTicketCloseTime("Active Currently");

            enqueueTicket(newTicket);

            TL.addTicket(newTicket); // adding it into all the tickets created today

        }

        void enqueueTicket (Ticket &incTicket) {

            cout << "Ticket ID " << incTicket.getInstanceTicketID() << " has been added." << endl;

            if (head == NULL) {

                head = new NodeQueue(incTicket);

                tail = head;

                tail->next = head;

            } else {

                NodeQueue \*temp = tail;

                temp->next = new NodeQueue(incTicket);

                tail = temp->next;

                tail->next = head;

            }

            numPendingTickets++;

            // Everytime a new ticket is added, make sure to put it at its correct place (sorted by priority)

            silentSortTicketQueue('p');

        }

        void dequeueTicket () {

            if (head == NULL) {

                cout<<"\nTicket's queue is already empty, no ticket can be popped." << endl;

                return;

            }

            cout << "Ticket ID " << peekTicket().getInstanceTicketID() << " has been dequeued from the Ticket Quuee." << endl;

            numPendingTickets--;

            if (head->next == head) {

                delete head;

                head = NULL;

                tail = NULL;

                return;

            }

            NodeQueue \*temp = head;

            head = head->next;

            tail->next = head;

            delete temp;

        }

        Ticket peekTicket () {

            if (head == NULL) {

                Ticket invalidTicket;

                invalidTicket.setTicketID(-1);

                return invalidTicket;

            }

            return head->ticket;

        }

        void printTicketQueue() {

            if (head == NULL) {

                cout<<"Ticket Queue is empty. Nothing to print." << endl;

                return;

            }

            NodeQueue \*temp = head;

            cout << "\_\_\_\_\_\_\_\_\_\_\_\_\_ Displaying " << numPendingTickets << " Tickets Pending Agent Assignment \_\_\_\_\_\_\_\_\_\_\_\_\_ " << endl;

            do {

                temp->ticket.printTicketDetails();

                temp = temp->next;

            } while (temp != head);

            cout<< endl;

        }

        void removeTicket(int targetID) {

            if (head == nullptr) {

                cout << "Ticket list is empty." << endl;

                return;

            }

            NodeQueue \*curr = head;

            NodeQueue \*prev = nullptr;

            bool removed = false;

            string tempName;

            do {

                if (targetID == curr->ticket.getInstanceTicketID()) {

                    removed = true;

                    tempName = curr->ticket.getCustomerName();

                    if (curr == head && curr->next == head) { // Only one element in the list

                        delete head;

                        head = NULL; // update head to nullptr as the list is now empty

                    } else if (curr == head) { // Target is at the head and there are other elements

                        // Update head to the next NodeQueue

                        NodeQueue \*tail = head;

                        while (tail->next != head) {

                            tail = tail->next;

                        }

                        head = head->next;

                        tail->next = head;

                        delete curr;

                    } else { // Target is not at the head

                        prev->next = curr->next;

                        delete curr;

                    }

                    break;

                }

                prev = curr;

                curr = curr->next;

            } while (curr != head);

            if (removed) {

                cout << "Ticket ID " << targetID << " registered under the name '" << tempName << "' has been removed from the tickets list." << endl;

            } else {

                cout << "Ticket ID " << targetID << " does not exist." << endl;

            }

        }

        void silentRemoveTicket(int targetID) {

            if (head == nullptr) {

                cout << "Ticket list is empty." << endl;

                return;

            }

            NodeQueue \*curr = head;

            NodeQueue \*prev = nullptr;

            bool removed = false;

            string tempName;

            do {

                if (targetID == curr->ticket.getInstanceTicketID()) {

                    removed = true;

                    tempName = curr->ticket.getCustomerName();

                    if (curr == head && curr->next == head) { // Only one element in the list

                        delete head;

                        head = NULL; // update head to nullptr as the list is now empty

                    } else if (curr == head) { // Target is at the head and there are other elements

                        // Update head to the next NodeQueue

                        NodeQueue \*tail = head;

                        while (tail->next != head) {

                            tail = tail->next;

                        }

                        head = head->next;

                        tail->next = head;

                        delete curr;

                    } else { // Target is not at the head

                        prev->next = curr->next;

                        delete curr;

                    }

                    break;

                }

                prev = curr;

                curr = curr->next;

            } while (curr != head);

        }

        Ticket findTicket(int targetID) {

            NodeQueue \*temp = head;

            do {

                if (targetID == temp->ticket.getInstanceTicketID()) {

                    return temp->ticket;

                }

                temp = temp->next;

            } while (temp != head);

            Ticket invalidTicket;

            invalidTicket.setTicketID(-1);

            return invalidTicket;

        }

        void sortTicketQueue() {

            ifstream inputFromFile;

            inputFromFile.open("config.txt");

            if (!inputFromFile) {

                cout<< "Error opening the file." << endl;

                return;

            }

            char ch;

            cout << "\nEnter sorting criteria (p for Priority, n for Name, t for Ticket Open Time): ";

            cin >> ch;

            if (ch != 'p' && ch != 'n' && ch != 't') {

                cout << "Tickets' sorting failed. Invalid sorting selection." << endl;

                return;

            }

            string algoChoice;

            getline(inputFromFile, algoChoice);

            if (algoChoice == "bubblesort") bubbleSortTicketQueue(ch);

            else if (algoChoice == "insertionsort") insertionSortTicketQueue(ch);

            else if (algoChoice == "selectionsort") selectionSortTicketQueue(ch);

            else if (algoChoice == "quicksort" || algoChoice == "mergesort") {

                cout << "Quick Sort and Merge Sort cannot be used on Queue (with linked list). Please use another sorting algorithm." << endl;

            }

            // else if (algoChoice == "quicksort") quickSortTickets(ch);

            // else if (algoChoice == "mergesort") mergeSortTickets(tickets, 0, n-1, ch, n);

            else cout << "Invalid sorting choice in the config file." << endl;

        }

        void silentSortTicketQueue(char ch) {

            if (head == NULL || head->next == head) return;

            bool swapped;

            do {

                swapped = false;

                NodeQueue \*current = head;

                // we need to break before we return to the head node in a circular list.

                do {

                    NodeQueue \*nextNode = current->next;

                    // check if we need to swap based on the chosen criterion

                    bool condition;

                    if (ch == 'p') {

                        condition = current->ticket.getPriority() < nextNode->ticket.getPriority();

                    } else if (ch == 'n') {

                        condition = current->ticket.getCustomerName() > nextNode->ticket.getCustomerName();

                    } else {

                        condition = current->ticket.getTicketOpenTime() > nextNode->ticket.getTicketOpenTime();

                    }

                    // swap ticket data if condition is met

                    if (condition) {

                        swap(current->ticket, nextNode->ticket);

                        swapped = true;

                    }

                    current = current->next;

                } while (current != tail); // stop just before looping back to head for the current pass

            } while (swapped); // keep sorting until no swaps are made in a full pass

        }

        void bubbleSortTicketQueue(char ch) {

            if (head == NULL || head->next == head) return;

            bool swapped;

            do {

                swapped = false;

                NodeQueue \*current = head;

                // we need to break before we return to the head node in a circular list.

                do {

                    NodeQueue \*nextNode = current->next;

                    // check if we need to swap based on the chosen criterion

                    bool condition;

                    if (ch == 'p') {

                        condition = current->ticket.getPriority() < nextNode->ticket.getPriority();

                    } else if (ch == 'n') {

                        condition = current->ticket.getCustomerName() > nextNode->ticket.getCustomerName();

                    } else {

                        condition = current->ticket.getTicketOpenTime() > nextNode->ticket.getTicketOpenTime();

                    }

                    // swap ticket data if condition is met

                    if (condition) {

                        swap(current->ticket, nextNode->ticket);

                        swapped = true;

                    }

                    current = current->next;

                } while (current != tail); // stop just before looping back to head for the current pass

            } while (swapped); // keep sorting until no swaps are made in a full pass

            cout << "Tickets sorted successfully based on the chosen criterion." << endl;

        }

        // Insertion Sort: Alot of it is fixed by gpt, my code had some logic flaws.

        void insertionSortTicketQueue(char ch) {

            if (head == NULL || head->next == head) return;

            // Separate sorted portion starting from `head` itself

            NodeQueue\* sortedEnd = head;

            // Start sorting from the second node

            NodeQueue\* current = head->next;

            while (current != head) {

                Ticket temp = current->ticket; // The ticket to be inserted into the sorted portion

                NodeQueue\* sorted = head;

                NodeQueue\* prevSorted = NULL; // Track the node before the insertion point

                bool inserted = false;

                // Traverse the sorted part and find the correct position for `temp`

                while (sorted != current) {

                    bool condition;

                    if (ch == 'p') {

                        condition = sorted->ticket.getPriority() > temp.getPriority();

                    } else if (ch == 'n') {

                        condition = sorted->ticket.getCustomerName() < temp.getCustomerName();

                    } else {

                        condition = sorted->ticket.getTicketOpenTime() < temp.getTicketOpenTime();

                    }

                    if (!condition) break; // Found the insertion point

                    prevSorted = sorted;

                    sorted = sorted->next;

                }

                if (sorted == current) {

                    // Current node is already in the correct position

                    sortedEnd = current;

                    current = current->next;

                } else {

                    // Remove `current` node from its position

                    sortedEnd->next = current->next;

                    // Insert `current` node at the found position

                    if (prevSorted == NULL) {

                        // Insert at the beginning (before head)

                        NodeQueue\* tail = head;

                        while (tail->next != head) {

                            tail = tail->next;

                        }

                        tail->next = current;

                        current->next = head;

                        head = current;

                    } else {

                        // Insert between `prevSorted` and `sorted`

                        prevSorted->next = current;

                        current->next = sorted;

                    }

                    current = sortedEnd->next;

                }

            }

            cout << "Tickets sorted successfully using insertion sort based on the chosen criterion." << endl;

        }

        // Selection Sort

        void selectionSortTicketQueue(char ch) {

            if (head == NULL || head->next == head) return; // Empty or single-node list

            NodeQueue\* current = head;

            do {

                NodeQueue\* minNode = current;

                NodeQueue\* iterator = current->next;

                // Find the node with the highest priority or earliest open time in the unsorted portion

                while (iterator != head) {

                    bool condition;

                    if (ch == 'p') {

                        condition = iterator->ticket.getPriority() > minNode->ticket.getPriority();

                    } else if (ch == 'n') {

                        condition = iterator->ticket.getCustomerName() < minNode->ticket.getCustomerName();

                    } else {

                        condition = iterator->ticket.getTicketOpenTime() < minNode->ticket.getTicketOpenTime();

                    }

                    if (condition) {

                        minNode = iterator;

                    }

                    iterator = iterator->next;

                }

                // Swap the tickets if a smaller (or higher priority) ticket is found

                if (minNode != current) {

                    Ticket temp = current->ticket;

                    current->ticket = minNode->ticket;

                    minNode->ticket = temp;

                }

                current = current->next;

            } while (current != head);

            cout << "Tickets sorted successfully using selection sort based on the chosen criterion." << endl;

        }

        void searchTicket () {

            ifstream inputFromFile;

            inputFromFile.open("config.txt");

            if (!inputFromFile) {

                cout<< "Error opening the file." << endl;

                return;

            }

            string algoChoice;

            inputFromFile >> algoChoice;

            inputFromFile >> algoChoice;

            // getline(inputFromFile, algoChoice); // ignore the first line because it is for sorting algorithm

            // getline(inputFromFile, algoChoice); // fetch algo name from 2nd line

            if (algoChoice != "binarysearch" && algoChoice != "interpolationsearch") {

                cout << "Invalid searching algorithm in the config file." << endl;

                return;

            }

            int choice;

            cout << "How do you want to search for the ticket? " << endl;

            cout << "1. Search by ID\n2. Search by Customer Name" << endl;

            choice = get\_int();

            if (choice != 1 && choice != 2) {

                cout<<"Invalid choice." << endl;

                return;

            }

            if (choice == 1) {

                int targetID;

                cout<<"Enter the ticket ID that you wanna search: ";

                targetID = get\_int();

                // Sort the ticket queue by time before searching

                // bubbleSortTicketQueue('t');

                if (algoChoice == "binarysearch") binarySearchTicketByID(targetID);

                else interpolationSearchTicketByID(targetID);

            }

            else {

                string targetName;

                cin.ignore();

                cout<<"Enter the customer name that you wanna search: ";

                getline(cin, targetName);

                // Sort the ticket queue by name before searching

                bubbleSortTicketQueue('n');

                if (algoChoice == "binarysearch") binarySearchTicketByName(targetName);

                else linearSearchTicketByName(targetName);

            }

        }

        Ticket binarySearchTicketByID(int targetID) {

            if (!head) {

                cout << "The ticket queue is empty." << endl;

                Ticket invalidTicket;

                invalidTicket.setTicketID(-1);

                return invalidTicket;

            }

            // Initialize start and end pointers for binary search

            NodeQueue\* start = head;

            NodeQueue\* end = tail->next;

            do {

                // Find the middle of the current range (start to end)

                NodeQueue\* slow = start;

                NodeQueue\* fast = start;

                // using two pointer technique to find the middle node

                while (fast != end && fast->next != end) {

                    fast = fast->next->next;

                    slow = slow->next;

                }

                if (slow->ticket.getInstanceTicketID() == targetID) {

                    cout << "Match Found! Customer Details:\n";

                    slow->ticket.printTicketDetails();

                    return slow->ticket;

                }

                else if (slow->ticket.getInstanceTicketID() < targetID) {

                    start = slow->next;

                }

                else {

                    end = slow;

                }

            } while (start != end);

            cout << "Ticket ID " << targetID << " does not exist and is not found in the database." << endl;

            Ticket invalidTicket;

            invalidTicket.setTicketID(-1);

            return invalidTicket;

        }

        Ticket binarySearchTicketByName(string targetName) {

            if (!head) {

                cout << "The ticket queue is empty." << endl;

                Ticket invalidTicket;

                invalidTicket.setTicketID(-1);

                return invalidTicket;

            }

            // Initialize start and end pointers for binary search

            NodeQueue\* start = head;

            NodeQueue\* end = tail->next;

            do {

                // find the middle of the current range (start to end)

                NodeQueue\* slow = start;

                NodeQueue\* fast = start;

                // using two pointer technique to find the middle node

                while (fast != end && fast->next != end) {

                    fast = fast->next->next;

                    slow = slow->next;

                }

                if (slow->ticket.getCustomerName() == targetName) {

                    cout << "Match Found! Customer Details:\n";

                    slow->ticket.printTicketDetails();

                    return slow->ticket;

                }

                else if (slow->ticket.getCustomerName() < targetName) {

                    start = slow->next;

                }

                else {

                    end = slow;

                }

            } while (start != end);

            cout << "Custoemr '" << targetName << "' does not exist and is not found in the database." << endl;

            Ticket invalidTicket;

            invalidTicket.setTicketID(-1);

            return invalidTicket;

        }

        Ticket interpolationSearchTicketByID(int targetID) {

            if (!head) {

                cout << "The ticket queue is empty." << endl;

                Ticket invalidTicket;

                invalidTicket.setTicketID(-1);

                return invalidTicket;

            }

            NodeQueue\* start = head;

            NodeQueue\* end = tail->next; // end points to head due to circular nature

            do {

                int startID = start->ticket.getInstanceTicketID();

                int endID = (end == head ? tail->ticket.getInstanceTicketID() : end->ticket.getInstanceTicketID());

                if (startID == targetID) {

                    cout << "Match Found! Customer Details:\n";

                    start->ticket.printTicketDetails();

                    return start->ticket;

                }

                if (endID == targetID) {

                    cout << "Match Found! Customer Details:\n";

                    tail->ticket.printTicketDetails();

                    return tail->ticket;

                }

                if (startID == endID || targetID < startID || targetID > endID) {

                    break; // Target out of bounds

                }

                // Estimate position using interpolation formula

                int count = countNodesBetween(start, end);

                int pos = (targetID - startID) \* count / (endID - startID);

                NodeQueue\* mid = moveForward(start, pos);

                if (mid->ticket.getInstanceTicketID() == targetID) {

                    cout << "Match Found! Customer Details:\n";

                    mid->ticket.printTicketDetails();

                    return mid->ticket;

                }

                else if (mid->ticket.getInstanceTicketID() < targetID) {

                    start = mid->next;  // Move start forward, bypassing the interpolated node

                }

                else {

                    end = mid;  // Narrow search to start and mid

                }

            } while (start != end);

            cout << "Ticket ID " << targetID << " does not exist and is not found in the database." << endl;

            Ticket invalidTicket;

            invalidTicket.setTicketID(-1);

            return invalidTicket;

        }

        // Helper function to count nodes between start and end

        int countNodesBetween(NodeQueue\* start, NodeQueue\* end) {

            int count = 0;

            NodeQueue\* temp = start;

            while (temp != end) {

                count++;

                temp = temp->next;

                if (temp == start) break; // loop detected in circular list

            }

            return count;

        }

        // Helper function to move forward by `pos` steps in a circular list

        NodeQueue\* moveForward(NodeQueue\* start, int pos) {

            NodeQueue\* current = start;

            while (pos-- > 0 && current != nullptr) {

                current = current->next;

                if (current == start) break; // Avoid infinite loop in circular list

            }

            return current;

        }

        Ticket linearSearchTicketByName(string targetName) {

            if (!head) {

                cout << "The ticket queue is empty." << endl;

                Ticket invalidTicket;

                invalidTicket.setTicketID(-1);

                return invalidTicket;

            }

            NodeQueue\* current = head;

            do {

                if (current->ticket.getCustomerName() == targetName) {

                    cout << "Match Found! Customer Details:\n";

                    current->ticket.printTicketDetails();

                    return current->ticket;

                }

                current = current->next;

            } while (current != head);

            cout << "Customer '" << targetName << "' does not exist and is not found in the database." << endl;

            Ticket invalidTicket;

            invalidTicket.setTicketID(-1);

            return invalidTicket;

        }

};

int TicketQueue::count = 0;

// --------------------------------------------------------------- Part 02: Agents ---------------------------------------------------------------

class Agent {

    private:

        int agentID;

        string agentName;

        bool availability;

        string status;

        int maxCapacity;

        int numAssignedTickets;

        int numResolvedTickets;

    public:

        Ticket\* assignedTickets; // tickets that have been assigned to an agent

        Ticket\* resolvedTickets; // tickets that have been resolved by an agent

        static int numAgents;

        // Agent () : agentID(++numAgents),  agentName(""), availability(1), status("Available"), numAssignedTickets(0), maxCapacity(5) {}

        Agent() : agentID(++numAgents), agentName(""), availability(1), status("Available"), numAssignedTickets(0), maxCapacity(5), numResolvedTickets(0) {

            assignedTickets = new Ticket[maxCapacity];

            resolvedTickets = new Ticket[maxCapacity];

        }

        Agent (string newAgentName) : agentID(++numAgents),  agentName(newAgentName), availability(1), status("Available"), numAssignedTickets(0), maxCapacity(5), numResolvedTickets(0) {

            assignedTickets = new Ticket[maxCapacity];

            resolvedTickets = new Ticket[maxCapacity];

        }

        void setAgentName (string name) {

            agentName = name;

        }

        int getAgentID() const {

            return agentID;

        }

        string getAgentName() const {

            return agentName;

        }

        bool isAvailable() const {

            return availability;

        }

        string getStatus() const {

            return status;

        }

        int getAssignedTicketCount() const {

            return numAssignedTickets;

        }

        void markUnavailable() {

            availability = false;

            status = "Unavailable";

            cout << "Agent " << agentName << " is now unavailable (Full capacity)." << endl;

        }

        void decNumAssignedTickets () {

            numAssignedTickets--;

        }

        void setAvailability(bool newAvailability) {

            availability = newAvailability;

            status = newAvailability ? "Available" : "Unavailable";

        }

        void setStatus(const string& newStatus) {

            status = newStatus;

        }

        int getMaxCapacity() const {

            return maxCapacity;

        }

        void incNumResolvedTickets () {

            numResolvedTickets++;

        }

        int getNumResolvedTickets () {

            return numResolvedTickets;

        }

        void assignTicket(Ticket ticket, TicketQueue &ticketQueue) {

            if (ticketQueue.numPendingTickets == 0) {

                cout << "There is no pending ticket to be assigned at the moment." << endl;

                return;

            }

            if (numAssignedTickets == maxCapacity) {

                cout << "Agent " << agentName << " has reached maximum ticket capacity." << endl;

                // markUnavailable();

            } else {

                assignedTickets[numAssignedTickets] = ticket;

                cout << "Ticket " << ticket.getInstanceTicketID() << " assigned to Agent " << agentName << endl;

                numAssignedTickets++;

                ticketQueue.dequeueTicket();

                if (numAssignedTickets == maxCapacity) { // set agent as unavailable if he has 5 tickets

                    markUnavailable();

                }

            }

        }

        void displayAgentsTickets () {

            cout<<"\_\_\_\_\_\_\_\_\_\_\_\_\_ " << numAssignedTickets << " Ticket(s) Assigned to Agent '" << agentName << "' \_\_\_\_\_\_\_\_\_\_\_\_\_" << endl;

            for (int i=0; i<numAssignedTickets; i++) {

                assignedTickets[i].printTicketDetails();

            }

            cout<<endl;

        }

        // In Agent class, add a method to get assigned ticket for index

        Ticket getAssignedTicket(int index) {

            if (index >= 0 && index < numAssignedTickets) {

                return assignedTickets[index];

            }

            throw std::out\_of\_range("Invalid ticket index");

        }

        // In Agent class, add a method to check if the agent has a specific ticket

        bool hasTicket(Ticket &ticket) {

            for (int i = 0; i < numAssignedTickets; i++) {

                if (assignedTickets[i].getInstanceTicketID() == ticket.getInstanceTicketID()) {

                    return true;

                }

            }

            return false;

        }

        void displayDetails () {

            cout << "Agent ID: " << agentID << endl;

            cout << "Agent Name: " << agentName << endl;

            cout << "Status: " << status << endl;

            cout << "Availability: " << availability << endl;

            cout << "Assigned Tickets: " << numAssignedTickets << endl;

            cout << "Resolved Tickets: " << numResolvedTickets << endl << endl;

        }

};

int Agent::numAgents = 0;

class AgentsList {

    int size;

    static int totalSize;

    Agent\* agents;

    public:

        AgentsList () : size(0) {}

        ~AgentsList() {

            delete[] agents;

        }

        void createAgent() {

            Agent newAgent;

            string name, status;

            bool availability;

            cout<<"Enter Agent Name: ";

            cin.ignore();

            getline(cin, name);

            newAgent.setAgentName(name);

            addAgent(newAgent);

        }

        void addAgent(Agent &newAgent) {

            cout << "Agent '" << newAgent.getAgentName() << "' has been added to the database." << endl;

            if (size == 0) {

                size = 1;

                agents = new Agent[size];

                agents[0] = newAgent;

                return;

            }

            Agent\* tempAgents = new Agent[size];

            for (int i = 0; i < size; i++) {

                tempAgents[i] = agents[i];

            }

            delete[] agents;

            agents = new Agent[size + 1];

            for (int i = 0; i < size; i++) {

                agents[i] = tempAgents[i];

            }

            agents[size] = newAgent;

            size++;

            delete[] tempAgents;

        }

        void assignTicketToAgent(TicketQueue &ticketQueue) {

            int minTickets = 6; // Start higher than max allowed tickets

            Agent\* selectedAgent = NULL;

            // Find an available agent with the least number of tickets

            for (int i = 0; i < size; i++) {

                if ((agents[i].isAvailable()) && (agents[i].getAssignedTicketCount() < minTickets) && (agents[i].getAssignedTicketCount() >=0 && agents[i].getAssignedTicketCount() < 5)) {

                    minTickets = agents[i].getAssignedTicketCount();

                    selectedAgent = &agents[i];

                }

            }

            if (selectedAgent != NULL) {

                selectedAgent->assignTicket(ticketQueue.peekTicket(), ticketQueue);

            } else {

                cout << "No available agents to assign the ticket!" << endl;

            }

        }

        void resolveTicket(TicketQueue &ticketQueue, TicketStack &ticketStack) {

            // cout << "Resolving the highest priority ticket..." << endl;

            Ticket highestPriorityTicket;

            highestPriorityTicket.setPriority(0);

            Agent\* selectedAgent = NULL;

            bool ticketFound = false;

            // Loop through all agents to find the highest priority ticket

            for (int k = 0; k < size; k++) {

                for (int i = 0; i < agents[k].getAssignedTicketCount(); i++) {

                    Ticket currentTicket = agents[k].assignedTickets[i];

                    cout << "test resolve " << endl;

                    // Check if this is the highest priority ticket found so far

                    if (!ticketFound || currentTicket.getPriority() > highestPriorityTicket.getPriority()) {

                        highestPriorityTicket = currentTicket;

                        selectedAgent = &agents[k];

                        ticketFound = true;

                    }

                }

            }

            // If no ticket was found, print a message and return

            if (!ticketFound) {

                cout << "No tickets or No Agents available to resolve!" << endl;

                cout << "You must add a ticket and an agent to continue, also make sure the ticket is first assigned to an agent." << endl;

                return;

            }

            cout << "Resolving Ticket ID " << highestPriorityTicket.getInstanceTicketID() << "..." << endl;

            Ticket ticket;

            // Remove the ticket from the agent's assignedTickets array

            for (int i = 0; i < selectedAgent->getAssignedTicketCount(); i++) {

                if (selectedAgent->assignedTickets[i].getInstanceTicketID() == highestPriorityTicket.getInstanceTicketID()) {

                    ticket = selectedAgent->assignedTickets[i];

                    // Shift the tickets to remove the resolved one

                    for (int j = i; j < selectedAgent->getAssignedTicketCount() - 1; j++) {

                        selectedAgent->assignedTickets[j] = selectedAgent->assignedTickets[j + 1];

                    }

                    selectedAgent->decNumAssignedTickets(); // decrease assigned tickets count

                    break;

                }

            }

            if (ticket.getInstanceTicketID() != -1) {

                // Set ticket status to closed and update ticketCloseTime

                ticket.setStatus("Closed");

                ticket.setTicketCloseTime(getCurrTime());

                // Change agent status to available if max capacity was reached

                if (selectedAgent->getAssignedTicketCount() < selectedAgent->getMaxCapacity()) {

                    selectedAgent->setStatus("Available");

                    selectedAgent->setAvailability(true);

                }

                // Push the resolved ticket into TicketStack

                ticketStack.pushTicket(ticket);

                cout << "Ticket ID " << ticket.getInstanceTicketID() << " has been resolved and logged." << endl;

            } else {

                cout << "Ticket not found!" << endl;

            }

        }

        // Function to log the resolved ticket in the stack

        void logResolvedTicket(Ticket &ticket, TicketStack &ticketStack) {

            ticketStack.pushTicket(ticket);

        }

        void sortAgents() {

            if (size == 0) {

                cout << "No agents added right now. Sorting cannot be done." << endl;

                return;

            }

            cout << "Starting the sort." << endl;

            ifstream inputFromFile;

            inputFromFile.open("config.txt");

            if (!inputFromFile) {

                cout<< "Error opening the file." << endl;

                return;

            }

            string algoChoice;

            getline(inputFromFile, algoChoice);

            if (algoChoice == "bubblesort") bubbleSortAgents();

            else if (algoChoice == "insertionsort") insertionSortAgents();

            else if (algoChoice == "selectionsort") selectionSortAgents();

            else if (algoChoice == "quicksort") quickSortAgents(agents, 0, size-1);

            else if (algoChoice == "mergesort") mergeSortAgents(agents, 0, size-1);

            else cout << "Invalid sorting choice in the config file." << endl;

        }

        // Bubble Sort

        void bubbleSortAgents() {

            for (int i = 0; i < size; i++) {

                bool flagSwap = false;

                for (int j = 0; j < size - i - 1; j++) {

                    if (agents[j].getAssignedTicketCount() < agents[j + 1].getAssignedTicketCount()) {

                        swap(agents[j], agents[j + 1]);

                        flagSwap = true;

                    }

                }

                if (!flagSwap) break;

            }

        }

        // Insertion Sort

        void insertionSortAgents() {

            for (int i = 1; i < size; i++) {

                Agent temp = agents[i];

                int j = i - 1;

                while (j >= 0 && agents[j].getAssignedTicketCount() < temp.getAssignedTicketCount()) {

                    agents[j + 1] = agents[j];

                    j--;

                }

                agents[j + 1] = temp;

            }

        }

        // Selection sort

        void selectionSortAgents() {

            for (int i = 0; i < size; i++) {

                int minIndex = i;

                for (int j = i + 1; j < size; j++) {

                    if (agents[j].getAssignedTicketCount() > agents[minIndex].getAssignedTicketCount()) {

                        minIndex = j;

                    }

                }

                if (minIndex != i) swap(agents[i], agents[minIndex]);

            }

        }

        // Quick Sort

        int partitionAgents(Agent agents[], int low, int high) {

            Agent pivot = agents[low];

            int start = low, end = high;

            while (start < end) {

                // move start pointer to the right until we find an element less than pivot

                while (start <= high && agents[start].getAssignedTicketCount() >= pivot.getAssignedTicketCount()) {

                    start++;

                }

                // move end pointer to the left until we find an element greater than pivot

                while (end >= low && agents[end].getAssignedTicketCount() < pivot.getAssignedTicketCount()) {

                    end--;

                }

                // if start is still less than end, swap elements at start and end

                if (start < end) {

                    swap(agents[start], agents[end]);

                }

            }

            // place pivot element in the correct position

            swap(agents[low], agents[end]);

            return end;

        }

        void quickSortAgents(Agent agents[], int low, int high) {

            if (low < high) {

                int pIndex = partitionAgents(agents, low, high);

                quickSortAgents(agents, low, pIndex - 1);

                quickSortAgents(agents, pIndex + 1, high);

            }

        }

        // Merge Sort

        void mergeAgents(Agent agents[], int left, int mid, int right) {

            int n1 = mid - left + 1;

            int n2 = right - mid;

            Agent\* leftArr = new Agent[n1];

            Agent\* rightArr = new Agent[n2];

            for (int i = 0; i < n1; i++) leftArr[i] = agents[left + i];

            for (int j = 0; j < n2; j++) rightArr[j] = agents[mid + 1 + j];

            int i = 0, j = 0, k = left;

            while (i < n1 && j < n2) {

                // Sorting by descending ticket count

                if (leftArr[i].getAssignedTicketCount() > rightArr[j].getAssignedTicketCount()) {

                    agents[k++] = leftArr[i++];

                } else if (leftArr[i].getAssignedTicketCount() == rightArr[j].getAssignedTicketCount()) {

                    // Secondary sort by name length if ticket counts are equal

                    if (leftArr[i].getAgentName().length() >= rightArr[j].getAgentName().length()) {

                        agents[k++] = leftArr[i++];

                    } else {

                        agents[k++] = rightArr[j++];

                    }

                } else {

                    agents[k++] = rightArr[j++];

                }

            }

            while (i < n1) agents[k++] = leftArr[i++];

            while (j < n2) agents[k++] = rightArr[j++];

            delete[] leftArr;

            delete[] rightArr;

        }

        void mergeSortAgents(Agent agents[], int left, int right) {

            if (left < right) {

                int mid = left + (right - left) / 2;

                mergeSortAgents(agents, left, mid);

                mergeSortAgents(agents, mid + 1, right);

                mergeAgents(agents, left, mid, right);

            }

        }

        void displayAgents () {

            if (size == 0) {

                cout << "No Agents added in the database at the moment." << endl;

                return;

            }

            cout<<"\_\_\_\_\_\_\_\_\_\_\_\_\_ Displaying Agents: " << size << " \_\_\_\_\_\_\_\_\_\_\_\_\_" << endl;

            for (int i=0; i<size; i++) {

                agents[i].displayDetails();

            }

        }

        void displaySpecificAgentTicket(int ID) {

            if (ID <= 0 || ID > size) {

                cout<<"Invalid Agent ID. Such Agent does not exist." << endl;

                return;

            }

            agents[ID-1].displayAgentsTickets();

        }

        void displayAllAssignedTickets () {

            for (int i = 0; i < size; i++) {

                if (agents[i].getAssignedTicketCount() != 0) {

                    agents[i].displayAgentsTickets();

                }

            }

        }

};

int AgentsList::totalSize = 0;

int main() {

    int choice;

    bool again = true;

    // Initialize system objects here

    AgentsList AL;

    TicketQueue TQ;

    TicketStack TS;

    TicketsList TL;

    while (again) {

        cout << "\n------------------ Muhammad Hammad || 23K-2005 -------------------";

        cout << "\n---------------------- One-Stop System Menu ----------------------\n";

        cout << "1. Add Ticket\n";

        cout << "2. Remove Ticket\n";

        cout << "3. Search for Ticket\n";

        cout << "4. Sort Tickets\n";

        cout << "5. Display Pending Tickets\n";

        cout << "6. Display All Tickets created today\n";

        cout << "7. Add Agent\n";

        cout << "8. Sort Agent by number of Tickets Assigned\n";

        cout << "9. Display all Agents\n";

        cout << "10. Assign Ticket to Agent\n";

        cout << "11. Resolve Ticket\n";

        cout << "12. Show Recent Ticket Log\n";

        cout << "13. Show All Ticket Logs\n";

        cout << "14. Exit\n";

        cout << "Choose an option: ";

        choice = get\_int();;

        switch (choice) {

            case 1: {

                // Create a ticket

                TQ.createTicket(TL);

                break;

            }

            case 2: {

                // Remove a ticket

                int ticketID;

                cout << "Enter Ticket ID to remove: ";

                ticketID = get\_int();;

                TQ.removeTicket(ticketID);

                break;

            }

            case 3: {

                // Search for a ticket

                TQ.searchTicket();

                break;

            }

            case 4: {

                // Sort the Tickets Queue

                TQ.sortTicketQueue();

                break;

            }

            case 5: {

                // Display pending tickets (ticket queue)

                TQ.printTicketQueue();

                break;

            }

            case 6: {

                // Display all tickets created today (tickets list)

                TL.displayTickets();

                break;

            }

            case 7: {

                AL.createAgent();

                break;

            }

            case 8: {

                // Sort Agents by Numer of Tickets Assigned

                AL.sortAgents();

                break;

            }

            case 9: {

                // Display Agents

                AL.displayAgents();

                break;

            }

            case 10: {

                // Assign a ticket to an agent

                AL.assignTicketToAgent(TQ);

                break;

            }

            case 11: {

                // Resolve a ticket

                AL.resolveTicket(TQ, TS);

                break;

            }

            case 12: {

                // Show recent ticket log (top of the stack)

                TS.peekTicketStack();

                break;

            }

            case 13: {

                // Show all ticket logs (entire stack)

                TS.printTicketStack();

                break;

            }

            case 14: {

                // Exit the program

                again = false;

                cout << "Exiting program. Ba-bye!\n";

                break;

            }

            default:

                cout << "Invalid option. Please try again.\n";

        }

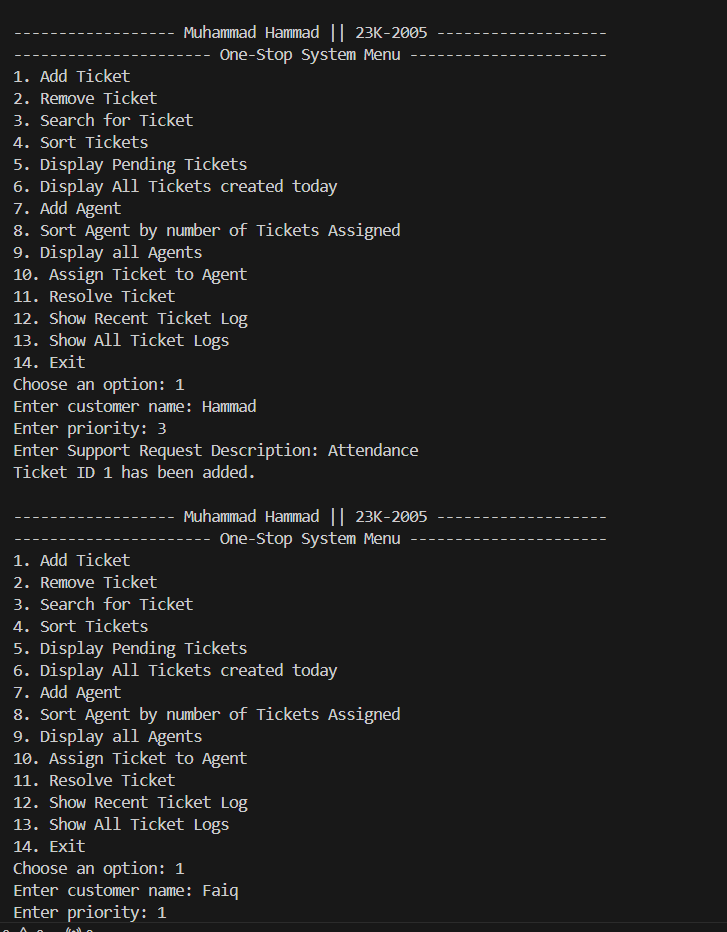
    }

    return 0;

}

**Output:**

**Creating Tickets**



A screenshot of a computer program

Description automatically generated

**Sort Tickets by Name and Display**

A screenshot of a computer program

Description automatically generated

A screenshot of a computer program

Description automatically generated

**Sort Tickets by Name and Display**

A screenshot of a computer program

Description automatically generated

A screenshot of a computer program

Description automatically generated

**Sort Tickets by Time and Display**

A screenshot of a computer program

Description automatically generated

A screenshot of a computer

Description automatically generated

**Removing a Ticket from the Queue**

A screenshot of a computer program

Description automatically generated

**Searching a Ticket by ID**

A screenshot of a computer program

Description automatically generated

**Searching a Ticket by Name**A screenshot of a computer

Description automatically generated

**Display Pending Tickets, Waiting to be Assigned**

A screenshot of a computer screen

Description automatically generated

**Adding Agents**

A screenshot of a computer screen

Description automatically generated

**Display All Agents**

A screenshot of a computer screen

Description automatically generated

**Assign Tickets To Agents**

A screenshot of a computer screen

Description automatically generated

A screen shot of a computer

Description automatically generated

**Resolving Tickets**

A screenshot of a computer

Description automatically generated

A screen shot of a computer

Description automatically generated

A screen shot of a computer

Description automatically generated

A screen shot of a computer

Description automatically generated

**Display Most Recent Log (Latest Ticket Processed)**

A screenshot of a computer program

Description automatically generated

**Display All Resolution Logs**

A screenshot of a computer program

Description automatically generated

**Display All the Tickets Created Today**

A screenshot of a computer program

Description automatically generated

A screenshot of a computer program

Description automatically generated

**Exiting Program**

A screenshot of a computer

Description automatically generated

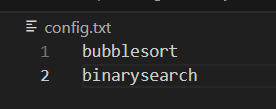
**Code Logic:**

**Structure:**

* Ticket class holds the attributes of the ticket.
* TicketsList class is there to hold all the tickets that were created in a day, both open and resolved tickets.
* TicketQueue class manages the tickets, holds pending tickets, and is responsible for the searching, sorting and removing of tickets.
* TicketStack class holds the logs, when a ticket is processed it is pushed into the TicketStack class.
* Agent class holds the attributes of agents.
* AgentList class is responsible for managing the Agent class. Adding, searching and sorting of agents is done here.

**Features:**

* Adding, removing, searching, sorting, printing tickets.
* Adding, searching, sorting of agents, printing agents and their assigned tickets’ details.
* Maintaining logs of resolved tickets.
* The pending tickets are managed in a queue.
* Configuration file where you can choose the sorting and searching algorithm.



**Note:**

* Some of the logics were fixed by GPT.
* I have added a few more features and functions, but word file only contains the simulation of program that were required in the program.