Traccia 1: Sistema di Gestione delle Prenotazioni per Servizi di Car Sharing

1) **Motivazioni dell'utilizzo degli ADT:**

Auto: abbiamo scelto di implementare *l'ADT Auto* come una struttura dati personalizzata, memorizzando tutti gli attributi essenziali di un'auto a noleggio. L'ADT consente di rappresentare ogni veicolo con dati tipo: *targa, marca, modello, anno, posizione* e *tariffa oraria*, oltre a una matrice di disponibilità che gestisce il *calendario settimanale* e *orario* di ciascun veicolo. Abbiamo scelto una struttura dati personalizzata a discapito di altre perché quest'ultima permette di raggiungere *in modo ordinato tutte le informazioni* e le operazioni associate a un'auto. In questo modo si riduce il rischio di errori nella gestione dei dati e si rende il codice più chiaro, leggibile e semplice da estendere. Grazie all'utilizzo di funzioni specifiche per ogni operazione (*creazione, stampa, verifica, modifica*), si evita di accedere direttamente ai singoli elementi, migliorandone la gestione.

- **Coda_StoricoUtente**: nel progetto è stato scelto di implementare una coda (*FIFO* ovvero *First In First Out*) come ADT per gestire lo storico delle prenotazioni effettuate dagli utenti. La decisione di utilizzare una coda è motivata dalla necessità di mantenere un *ordine cronologico* delle prenotazioni: la prima prenotazione effettuata è anche la prima a essere consultata o rimossa, seguendo un approccio naturale per uno storico. Non sono stati scelti altri ADT come liste o pile, poiché questi non garantivano lo stesso comportamento semantico. Le liste, seppur flessibili, richiedono una gestione più complessa dei puntatori e dell'ordinamento se si vuole simulare un comportamento FIFO ed in più, la gestione di una lista diventa complicata se ci dobbiamo interfacciare con una grande quantità di dati. Per di più sarebbe convenuta la lista in caso noi avremmo dovuto effettuare delle *ricerche all'interno dello storico* ma non avvenendo ciò, abbiamo deciso di scegliere una coda; le pile, invece, sarebbero risultate del tutto inadeguate per uno storico, in quanto la prenotazione più recente sarebbe stata elaborata per prima, violando il principio temporale desiderato. Inoltre, l'ADT coda è efficiente nella gestione in memoria dinamica e permette operazioni

di inserimento e rimozione, risultando ideale per contesti in cui l'ordine di ingresso va preservato.

- **HashTbAuto**: Per la gestione delle automobili all'interno del sistema, è stato scelto come ADT una *tabella hash*, grazie alle sue proprietà di efficienza nell'accesso, inserimento e cancellazione. In particolare, si è scelto di implementarla utilizzando la libreria *uthash,* che permette di associare una chiave (in questo caso, *la targa* dell'auto) a un valore (*l'oggetto auto* stesso) in modo semplice e performante. Questa scelta si è rivelata particolarmente adatta poiché il sistema richiede frequenti ricerche rapide in base alla targa, cosa che una lista o un array non avrebbe garantito con la stessa efficienza. Altri ADT alternativi, come le liste concatenate o gli alberi binari, sono stati scartati principalmente per motivi di prestazioni. Le liste non garantiscono accesso diretto e richiedono ricerche lineari, mentre gli alberi binari, pur avendo prestazioni migliori, richiedono gestione di bilanciamenti e maggiore complessità implementativa. Inoltre, per la natura delle operazioni richieste (*ricerche, inserimenti e cancellazioni*) una tabella hash risulta la soluzione più giusta. La libreria uthash consente una gestione trasparente delle collisioni e offre un'interfaccia molto intuitiva per le operazioni necessarie nel contesto.

- **HashTbUtenti**: nel progetto abbiamo scelto la *tabella hash* per la gestione degli utenti, poiché grazie a essa otteniamo accessi rapidi ai dati attraverso una *chiave univoca*. In questo caso, il *codice fiscale* dell'utente (*CF*) funge da chiave, rendendo la tabella hash estremamente efficiente per operazioni di ricerca, inserimento ed eliminazione. L'accesso rapido ai dati della tabella ci ha permesso una maggiore efficienza nelle funzioni di login, registrazione e aggiornamenti ai dati degli utenti. Alternative come le liste concatenate avrebbero potuto rendere più complicata la ricerca all'interno di esse poiché, esiste il rischio che dentro questa struttura dati vengano memorizzati una grande quantità di dati. Inoltre l'uso della libreria uthash, grazie ai tipi preimpostati e alle funzioni peculiari di questa libreria, l'implementazione è risultata facilitata garantendo al contempo *affidabilità e flessibilità*. L'utilizzo di altre strutture, come array dinamici o liste, non avrebbe offerto la stessa

immediatezza di accesso tramite chiave. In aggiunta, la tabella hash permette di evitare duplicati in modo naturale, grazie al controllo sulla chiave prima.

- **List_Prenotazione**: Nel progetto si è scelto di utilizzare una *lista concatenata singola* per la gestione delle prenotazioni. Abbiamo scelto una lista grazie alla sua *gestione dinamica della memoria e alla semplicità di inserimento in testa*. Le operazioni richieste dal sistema (ad esempio: *inserimento, filtraggio, copia e stampa*) sono tutte ben supportate da una lista dinamica, che permette di gestire insiemi di dati con dimensione variabile senza dover conoscere a priori la loro dimensione. Non sono state utilizzate altre strutture come ad esempio i vettori poiché gli array richiedono una dimensione iniziale predefinita; non sapendo di preciso con quanti dati dobbiamo lavorare, questa struttura ci avrebbe causato problemi di memoria. Le liste invece consentono *un'espansione naturale e progressiva, allocando memoria solo quando necessario*. Anche altre strutture come alberi o hash table non erano giustificate, poiché le funzionalità richieste non includono ricerche complesse o ordinamenti, ma piuttosto una gestione sequenziale e filtraggio lineare dei dati. Inoltre, la lista concatenata permette una *semplice implementazione* delle copie e della distruzione dei dati, operazioni fondamentali nel contesto del progetto per garantire l'integrità delle prenotazioni durante la manipolazione.

- **Prenotazione**: Nel progetto è stato adottato un ADT denominato Prenotazione, modellato tramite una *struttura dati s_prenotazione*. La motivazione principale di questa scelta risiede nella *necessità di memorizzare in modo ordinato e coerente* tutte le informazioni relative a una prenotazione, come il codice fiscale dell'utente, la targa dell'auto e il periodo della prenotazione. L'ADT permette di nascondere la struttura interna dei dati e di offrire solo le funzioni necessarie per usarli. In questo modo, chi utilizza il sistema può lavorare in modo chiaro e sicuro senza preoccuparsi di come i dati sono gestiti internamente. Non sono state utilizzate alternative come semplici strutture globali o array statici poiché queste non avrebbero permesso un adeguato livello di incapsulamento e flessibilità. Inoltre,

approcci più primitivi avrebbero complicato la gestione della memoria dinamica, rendendo il sistema meno modulare. Usare un ADT con funzioni associate ha permesso di definire operazioni chiare come *creaPrenotazione, stampaPrenotazione e distruggiPrenotazione*, migliorando così la manutenibilità del codice.

2) Come usare e interagire con il sistema

Appena il programma viene avviato, ci ritroveremo davanti a un'interfaccia che ci permette di scegliere *3 opzioni*: puoi scegliere l'opzione "*utente*" (opzione 1), opzione "*sviluppatore*" (opzione 2) oppure "*esci*" (opzione 0: in questo modo esci programma).

Se hai scelto l'opzione 1 (utente) ti ritroverai davanti a una schermata che ti farà scegliere se effettuare il login (opzione 1: dovrai inserire le credenziali con le quali ti sei registrato in precedenza; quindi dovrai inserire il codice fiscale e la password) oppure la registrazione (opzione 2: qui potrai inserire le credenziali per essere memorizzato nel catalogo degli utenti; quindi dovrai inserire codice fiscale, nome, cognome, email, numero di telefono e la password). Se avrai scelto di registrarti dovrai poi accedere e una volta fatto, ti troverai davanti all'interfaccia utente. E potrai scegliere se:

- *Prenotazione auto*: Qui dovrai inserire la targa dell'auto che vorrai noleggiare, successivamente dovrai inserire per quanto tempo e quando vuoi noleggiare l'auto e infine, se l'auto è disponibile, la prenotazione sarà effettuata con succeso;
- *Calcolo tariffa:* per auto prenotata: Questa opzione ti permette di visualizzare i calcoli dei prezzi delle tue singole prenotazioni (con sconti annessi) e ti fa il prezzo complessivo totale;
- *Visualizza auto disponibili*: In questo modo potrai vedere tutte le auto disponibili in un determinato lasso di tempo a scelta dell'utente;
- *Visualizza prenotazioni*: Questa opzione ti permette di visualizzare le prenotazioni della settimana corrente.
- *Visualizza prenotazioni precedenti*: Permette di visualizzare le tutte le prenotazioni passate (avvenute le settimane scorse) dell'utente loggato in quel momento;

- *Esci*: Con questa opzione puoi uscire dall'interfaccia utente.

Se hai scelto l'opzione 2 (sviluppatore) invece, potrai scegliere se:

- *Aggiungere una nuova auto*: Ti verrà chiesto di inserire le nozioni generali dell'auto (targa, marca, modello, anno di immatricolazione, posizione dell'auto e infine il prezzo orario del noleggio);
- *Rimuovi auto*: Ti verrà chiesto di inserire la targa dell'auto da rimuovere.
- *Visualizza tutte le auto*: Questa opzione ci permetterà di visualizzare sullo schermo tutte le auto presenti in catalogo (disponibili o meno);
- *Visualizza tutte le prenotazioni*: In questo modo lo sviluppatore potrà vedere tutte le prenotazioni effettuate da qualsiasi utente;
- *Visualizza storico*: In questo modo lo sviluppatore potrà vedere tutte le prenotazioni effettuate e avvenute (in precedenza) da qualsiasi utente;
- *Avanza di una settimana*: In questo modo potrai re inizializzare sia lo stato di tutte le auto a "disponibile" e sia le prenotazioni. E per di più la settimana scorrerà.
- *Esci*: L'opzione che ti permetterà di uscire dall'interfaccia sviluppatore.

3) Progettazione:

- **Auto**: Il tipo astratto Auto, definito all'interno del file Auto.c, è realizzato attraversa una struttura che contiene tutti i dati necessari per rappresentare un veicolo. A questa struttura è stata aggiunta una matrice booleana a due dimensioni (giorni × ore), utile per gestire la disponibilità del veicolo durante la settimana lavorativa. L'organizzazione del progetto prevede l'utilizzo di funzioni specifiche per ogni operazione. Questo approccio rende il sistema ordinato e riduce la necessità di accedere direttamente ai singoli campi della struttura, affidando ogni operazione a una funzione apposita. In questo modo si ottiene una netta separazione tra la definizione dei dati e la logica di utilizzo. La gestione della disponibilità è uno degli aspetti centrali del progetto. Ogni veicolo ha associata una matrice settimanale che può essere

modificata tramite intervalli orari, con appositi controlli che garantiscono la validità delle richieste. È stato previsto il caso in cui l'intervallo non sia valido (ad esempio: non si può prenotare dalle 20 alle 10 dello stesso giorno; oppure non si può prenotare in orari dove il veicolo interessato è già occupato), e viene stampato un messaggio di errore per evitare prenotazioni errate. Dal punto di vista visivo, sono stati utilizzati codici colore ANSI per migliorare la leggibilità e l'estetica dei messaggi su terminale: errori evidenziati in rosso e intestazioni in ciano. Questo migliora l'esperienza dell'utente durante l'interazione, soprattutto nelle fasi di debug e verifica. Nel complesso, la progettazione ha seguito un approccio ordinato e modulare, puntando a facilitare l'utilizzo del sistema.

- **Coda_StorcoUtente**: L'implementazione dell'ADT Coda si basa su una struttura dati dinamica composta da nodi concatenati singolarmente. Ogni nodo contiene un campo Prenotazione e un puntatore al nodo successivo. Il tipo struttura c_coda contiene invece i riferimenti alla testa e alla coda della sequenza, oltre a un contatore degli elementi. Anche qui ogni operazione fondamentale è incapsulata in una funzione apposita, garantendo l'indipendenza tra le componenti. L'inserimento di nuovi elementi avviene esclusivamente in fondo alla coda, mentre la rimozione è limitata alla testa, seguendo il comportamento tipico delle strutture FIFO. Queste operazioni sono progettate per avere complessità costante, poiché non è necessario scorrere l'intera struttura per effettuare l'aggiunta o l'estrazione di un elemento. Il numero di elementi indicato dal contatore numel deve essere coerente con la reale struttura dei nodi, e i puntatori testa e coda devono riflettere correttamente lo stato della sequenza. Per evitare eventuali sprechi di memoria, ogni nodo rimosso viene liberato tramite free() al momento del prelievo, evitando perdite di memoria. Nelle funzioni che operano su tutte le prenotazioni, come stampaStorico, viene impiegata una coda temporanea per garantire che i dati originali non vengano alterati, ripristinando l'ordine iniziale al termine della procedura.

- **Controlli**: Il progetto è stato sviluppato seguendo un approccio modulare. Ogni funzione si occupa di un singolo tipo di controllo e segue un pattern

uniforme: analizza i caratteri uno per uno, verifica la loro validità secondo regole stabilite (es. lunghezza minima, presenza di caratteri speciali, sequenze numeriche o alfabetiche) e restituisce un risultato binario. Le funzioni principali includono: validaPassword verifica che una password abbia almeno 8 caratteri, una lettera maiuscola, una minuscola, un numero e un carattere speciale; validaEmail controlla la presenza di un @ e di un . successivo, assicurandosi che siano nel giusto ordine; validaTarga assicura che la targa italiana sia composta da 7 caratteri nell'rodine lettera-lettera-numero-numero-lettera-lettera; validaCodiceFiscale: verifica che il codice fiscale sia lungo 16 caratteri e rispetti la sequenza di lettere e numeri corretta; validaTelefono richiede esattamente 10 cifre che iniziano per 3, per rispettare il formato dei numeri mobili italiani; validaViaStradale controlla che l'indirizzo cominci con un prefisso valido (Via, Viale, ecc.) e termini con un numero civico; Oltre alle validazioni, sono presenti funzioni di supporto come confrontaCaseInsensitive e capitalizza, utili a migliorare l'accuratezza e l'uniformità dei dati. La progettazione ha tenuto conto anche della portabilità: per esempio, la funzione pulisciConsole usa direttive di compilazione per distinguere tra Windows e Unix.

- **HashTbAuto**: La progettazione del sistema è stata orientata alla modularità e alla chiarezza. La struttura dati AutoRecord (definita nel file), associa ogni oggetto auto alla sua targa mediante il campo char* targa, e integra il gestore UT_hash_handle necessario per il funzionamento di uthash. La struttura AutoHashTB è un alias per un puntatore a AutoRecord, rendendo l'utilizzo della tabella flessibile e intuitivo. Tutte le operazioni su AutoHashTB sono implementate in modo da garantire coerenza interna: l'inserimento controlla che la targa non sia già presente per evitare duplicati; la rimozione libera tutta la memoria associata; la distruzione totale della tabella elimina ricorsivamente ogni record e l'oggetto auto relativo, assicurando l'assenza di sprechi di memoria. È stata implementata la stampa condizionata alla disponibilità temporale delle auto, funzione che dimostra la scalabilità del sistema verso scenari reali. Sono stati utilizzati colori ANSI nelle stampe su schermo per migliorare l'estetica, l'esperienza utente, facilitando l'identificazione delle informazioni salienti.

- **HashTbUtente**: HashTbUtenti.c è stato progettato seguendo un'architettura modulare e chiara. L'elemento centrale è la struttura HashRecord, che associa una chiave (cf, codice fiscale) a un valore di tipo Utente. Ogni HashRecord viene gestito tramite la libreria uthash, che consente di creare e manipolare la tabella hash senza dover implementare manualmente le funzioni di hashing. Le operazioni principali implementate sono: inserisciUtente aggiunge un utente solo se non è già presente nella tabella; cercaUtente restituisce l'utente associato a un CF specifico; eliminaUtente rimuove l'utente dalla tabella, restituendone i dati; distruggiHashTBUtenti libera completamente la memoria occupata dalla tabella hash; loginRegistrazioneUtente gestisce il flusso di autenticazione e inserimento dati da parte dell'utente; aggiungiPrenotazioniAStoricoUtenti associa prenotazioni a utenti esistenti; stampaStoricoTuttiUtenti stampa lo storico di prenotazioni per ogni utente. Anche qui per evitare evenutale perdita di memoria, sono stati inseriti free quando dovevamo eliminare degli elementi.

- **List_Prenotazione**: La progettazione del sistema si basa sull'utilizzo di una lista concatenata singola, una struttura dati dinamica in cui ogni elemento (nodo) contiene una prenotazione e un puntatore al nodo successivo. La lista viene rappresentata come un puntatore al primo nodo e viene gestita attraverso una serie di funzioni che permettono di costruirla, modificarla e consultarla. Le funzioni fondamentali di List_Prenotazione includono: nuovaLista() per creare una lista vuota, consLista() per aggiungere una prenotazione in testa, e ListaVuota() per verificare se la lista contiene elementi. Per far scorrere la lista, si utilizzano ottieniPrimo() per accedere al primo elemento e codaLista() per ottenere la lista senza il primo nodo. Abbiamo sviluppato delle funzioni apposta per garantire sia il filtraggio delle prenotazioni o la stampa dei dati e sia per evitare sprechi di memoria ovvero: la funzione distruggiLista() si occupa di deallocare correttamente tutta la memoria utilizzata dalla lista, evitando sprechi di risorse; copiaProfondaLista(), al contrario, genera una replica indipendente della lista originale, particolarmente utile per eseguire operazioni in sicurezza

senza modificare i dati di partenza; le funzioni come stampaListaPrenotazioni, visPrenotazioniPerUtente e filtraPrenotazioniPerCF sono progettate per visualizzare le informazioni a schermo o per estrarre solo determinati dati. Il sistema utilizza codici di colore ANSI per rendere l'output a terminale più chiaro e leggibile.

- **Prenotazione**: L'ADT Prenotazione è definito come un puntatore a una struttura s_prenotazione, la cui definizione rimane interna al file sorgente per favorire l'incapsulamento. Questo approccio impedisce l'accesso diretto ai membri della struttura al di fuori delle funzioni definite, garantendo l'integrità dei dati. La struttura s_prenotazione contiene quattro campi principali: un identificativo univoco (ID_prenotazione), il Codice Fiscale dell'utente (CF), la targa del veicolo (targa) e una struttura Periodo che racchiude il giorno e l'ora di inizio e fine. Questo consente una rappresentazione chiara del tempo di prenotazione. Anche qui le funzioni sono organizzate in modo modulare, ciascuna con un singolo compito specifico. Le stringhe vengono copiate usando strncpy per evitare problemi di buffer overflow, e ogni funzione che alloca memoria dinamica controlla che l'allocazione sia andata a buon fine, terminando il programma in caso contrario con un messaggio di errore colorato in rosso (con i codici del colore ANSI) per facilitarne la rilevazione. Per rendere più leggibili le informazioni chiave stampate a schermo abbiamo usato vari colori nella stampa (rosso, bl, giallo e ciano).

- **Utente**: Utente è stata sviluppata secondo una programmazione modulare. La struttura s_utente incapsula tutti i dati relativi a un singolo utente. In questo file viene sfruttato anche un ADT introdotto in Coda_StoricoUtente, cioè una Coda, progettata per mantenere traccia cronologica delle attività dell'utente, come accessi, modifiche o operazioni effettuate. In Utente sono state implementa tutte le funzionalità fondamentali: creaUtente che serve per creare un nuovo utente, ottieniX che serve per l'accesso ai dati, stampaUtente per stampare e distruggiUtente che tramite la funzione free elimina un utente senza creare sprechi in memoria. Questo garantisce un'interfaccia semplice, chiara e sicura. Qui vengono memorizzati dati di tipo char[] per memorizzare stringhe come CF, nome, email, ecc., con lunghezze

massime definite per evitare problemi di overflow. È presente un controllo esplicito per assicurare che le stringhe siano terminate correttamente. Inoltre, l'uso di malloc permette una gestione dinamica della memoria, utile in scenari in cui il numero di utenti non è noto a priori.

- **Utile_DevMenu**: Il file Utile_DevMenu.c gestisce l'interfaccia dedicata allo sviluppatore e funge da punto di interazione con gli altri moduli del sistema (come Auto.h, HashTbAuto.h, HashTbUtenti.h, Controlli.h e i rispettivi file .c) i quali si occupano della logica di gestione delle auto, degli utenti e dei controlli. Attraverso questa collaborazione tra moduli, il sistema garantisce un funzionamento coerente e modulare. A livello strutturale, per la gestione dei dati è stato utilizzato il tipo Auto, che rappresenta ogni veicolo con le sue proprietà (targa, marca, modello, anno, posizione, prezzo). Questi oggetti sono poi gestiti tramite una tabella hash dinamica (definita nel file HashTbAuto). Per rappresentare marche e modelli, è stata usata una struttura MarcaModelli che mappa ogni marca a un numero fisso di modelli validi. Questo approccio consente una verifica guidata e immediata della correttezza degli input forniti dagli sviluppatori durante l'aggiunta di nuove auto. Le funzioni di interazione (per esempio: aggiungiAutoInterattivo, rimuoviAuto, stampaAutoHashTB), verificano sempre la correttezza dei dati prima di procedere con modifiche al sistema. Il sistema implementa anche un meccanismo di avanzamento temporale per simulare il passaggio di una settimana, aggiornando lo stato delle auto e lo storico delle prenotazioni.

- **Utile_UtenteMenu**: anche in Utile_UtenteMenu la progettazione del codice viene scritta con approccio modulare. Questo file gestisce le varie opzioni del menù utenti suddividendo le funzionalità in diversi file .c e .h. Ogni modulo gestisce un aspetto specifico: List_Prenotazione.c/h gestisce la lista di prenotazioni; HashTbAuto.c/h gestisce la tabella hash per le auto; Utente.h / Auto.h / prenotazione.h definiscono le strutture dati principali; Utile_UtenteMenu.c gestisce del menù e delle interazioni utente; Controlli.c/h contiene funzioni di validazione (es. formati targa, input

numerico). Ogni struttura dati è incapsulata e acceduta tramite funzioni specifiche, migliorando la coesione e riducendo il coupling tra moduli. L'interfaccia utente è separata dalla logica del sistema. L'uso di macro per la colorazione del testo migliora l'usabilità (codice colore ANSI), rendendo chiari gli errori e i messaggi di successo.

- **Test**: Il file Test.c è stato progettato per gestire e automatizzare i test delle funzionalità principali del sistema. Contiene funzioni per caricare dati fittizi, eseguire simulazioni di prenotazioni, calcolare costi, visualizzare disponibilità e stampare storici. Il cuore del file è la funzione eseguiCasoDiTest, che legge un test da file, esegue l'operazione richiesta e confronta l'output ottenuto con l'output atteso (oracolo), scrivendo i risultati. La funzione esegui_test_suite permette di automatizzare più test in sequenza. Le prenotazioni vengono gestite con liste e simulate in modo controllato per verificare i diversi comportamenti del sistema. Alla fine dei test, tutti i dati fittizi vengono deallocati per evitare degli sprechi di memoria.

- **Main**: Il file main.c gestisce il ciclo principale del programma di car sharing. All'avvio, inizializza tre strutture dati fondamentali: una tabella hash per gli utenti, una per le auto e una lista per le prenotazioni. Queste strutture vengono poi usate per tutta la durata dell'esecuzione. Il programma entra in un ciclo in cui chiede all'utente se vuole accedere come **utente** o come **sviluppatore**. Se si sceglie il ruolo di utente, si accede alla fase di login o registrazione. Dopo l'accesso, viene mostrato un menù interattivo che consente di prenotare auto, visualizzare le proprie prenotazioni, ecc. Le scelte vengono gestite tramite switchUtente. Se si accede come sviluppatore, viene mostrato un menù diverso che permette operazioni come l'aggiunta o rimozione di auto, o la visualizzazione delle prenotazioni. Anche in questo caso, ogni scelta viene gestita da una funzione apposita. Alla chiusura del programma, tutte le strutture dati vengono distrutte per liberare la memoria.

4) Specifiche

•**Auto**:

Auto creaAuto(char * targa, char * marca, char * modello, char* posizione, float prezzoxora, anno)

Specifica sintattica

creaAuto(char *, char *, char *, char*, int, float) → Auto

Specifica semantica

creaAuto(targa, marca, modello, posizione, prezzoxora, anno) → a

La funzione crea un'auto, che successivamente può essere aggiunta al catalogo, e vengono passati i dati dell'auto come parametro, in modo da inizializzarli.

Precondizioni:

- targa è una stringa composta da 7 caratteri e deve essere composta da 2 lettere, 3 numeri e altre 2 lettere;
- marca è una stringa non vuota che contiene un nome valido di casa automobilistaica (le marche sono predefinite nel programma)
- modello è una stringa non vuota; anno è un valore intero che contiene l'anno di uscita dell'auto;
- posizione è una stringa non vuota da massimo 35 caratteri.

Postcondizione:

- a è un puntatore a una struttura (Auto) inizializzata con i valori forniti in input.

Side effect:

- la variabile struttura a viene allocata dinamicamente in memoria.

char* ottieniTarga(Auto a)

Specifica sintattica

ottieniTarga(Auto) → char*

Specifica semantica

ottieniTarga(a) → a->targa

La funzione accede al campo targa della struttura data in input e lo stampa.

Precondizioni:

- a è un puntatore a una struttura (Auto) correttamente allocata in memoria.

Postcondizioni:

- Il campo targa, appartiene alla struttura Auto puntata da a, è di tipo stringa (7 caratteri) e viene stampato su standard output.

Side effect:

- Nessuno.

char* ottieniTarga(Auto a)

Specifica sintattica

ottieniMarca(Auto) → char*

Specifica semantica

ottieniMarca(a) → a->marca

La funzione accede al campo marca della struttura data in input e lo stampa.

Precondizioni:

- a è un puntatore a una struttura (Auto) correttamente allocata in memoria.

Postcondizioni:

- Il campo marca, appartiene alla struttura Auto puntata da a, è di tipo stringa e viene stampato su standard output.

Side effect:

- nessuno.

char* ottieniModello(Auto A)

Specifica sintattica

ottieniModello(Auto) → char*

Specifica semantica

ottieniModello(a) → a->modello

La funzione accede al campo modello della struttura data in input e lo stampa.

Precondizioni:

- a è un puntatore a una struttura (Auto) correttamente allocata in memoria.

Postcondizioni:

- Il campo modello, appartiene alla struttura Auto puntata da a, è di tipo stringa e viene stampato su standard output.

*Side effec*t:

- nessuno.

int ottieniAnno(Auto a)

Specifica sintattica

ottieniAnno(Auto) → int

Specifica semantica

ottieniAnno(a) → a

La funzione accede al campo anno della struttura data in input e lo stampa.

Precondizioni:

- a è un puntatore a una struttura (Auto) correttamente allocata in memoria.

Postcondizioni:

- Il campo targa, appartiene alla struttura Auto puntata da a, è di tipo intero e viene stampato su standard output.

Side effect:

- nessuno.

float ottieniPrezzo(Auto a)

Specifica sintattica

ottieniPrezzo(Auto) → float

Specifica semantica

ottieniPrezzo(a) → a

La funzione accede al campo prezzo della struttura data in input e lo stampa.

Precondizioni:

- a è un puntatore a una struttura (Auto) correttamente allocata in memoria.

Postcondizioni:

- Il campo targa, appartiene alla struttura Auto puntata da a, è di tipo float e viene stampato su standard output.

*Side effec*t:

- nessuno.

char* ottieniPosizione(Auto a)

Specifica sintattica

ottieniPosizione(Auto) → char*

Specifica semantica

ottieniPosizione(a) → a

La funzione accede al campo posizione della struttura data in input e lo stampa.

Precondizioni:

- a è un puntatore a una struttura (Auto) correttamente allocata in memoria.

Postcondizioni:

- Il campo posizione, appartiene alla struttura Auto puntata da a, è una stringa da massimo 35 caratteri e viene stampato su standard output.

Side effect:

- nessuno.

void ottieniAuto(Auto a)

Specifica sintattica

distruggiAuto(Auto) → void

Specifica semantica

distruggiAuto(a) → void

La funzione, quando viene chiamata, libera la partizione di memoria dell'auto passata come parametro eliminandola.

Precondizioni:

- a è un puntatore a una struttura (Auto) correttamente allocata in memoria.

*Postcondizion*i:

- la funzione non presenta post condizioni.

Side effect:

- viene liberato dello spazio in memoria in maniera dinamica.

void distruggiAuto(Auto a)

Specifica sintattica

distruggiAuto(Auto) → void

Specifica semantica

distruggiAuto(a) → void

La funzione, quando viene chiamata, libera la partizione di memoria dell'auto passata come parametro eliminandola.

Precondizioni:

- a è un puntatore a una struttura (Auto) correttamente allocata in memoria.

Postcondizioni:

- la funzione non presenta post condizioni.

Side effect:

- viene liberato dello spazio in memoria in maniera dinamica.

void impostaDisponibile(Auto a, int giornoInizio, intgiornoFine, int oraInizio, int oraFine, bool stato)

Specifica sintattica

impostaDisponibile(Auto, int, int, int, int, bool) → void

Specifica semantica

distruggiAuto(a) → void

impostaDisponibile(a, giornoInizio, giornoFine, oraInizio, oraFine, stato) -> void

Vengono passate per parametro alla funzione un puntatore alla struttura Auto e un intervallo di tempo. La funzione deve stabilire se l'auto passata per parametro è disponibile in quel lasso dato in input.

Precondzione:

- a è un puntatore a una struttura (Auto) correttamente allocata in memoria; giornoInizio è una variabile intera che contiene un valore maggiore o uguale a 0;
- giornoFine è una variabile intera che contiene un valore maggiore o uguale a 0;
- oraInizio è una variabile intera che contiene un numero maggiore o uguale a 0 e allo stesso tempo minore o uguale a 24;
- oraFine è una variabile intera che contiene un numero maggiore o uguale a 0 e allo stesso tempo minore o uguale a 24;
- stato è una variabile booleana che se è true, la funzione imposta la matrice a true per indicare che l'auto è disponibile nell'intervallo specificato, se è false imposta la matrice a false per indicare che l'auto non è disponibile in quell'intervallo;
- giornoInizio è sempre minore di giornoFIne.

Postcondizione:

- la funzione non presenta post condizioni.

Side effect:

- la funzione non presenta side effect.

int verificaDisponibilita(Auto a, int giornoInizio, intgiornoFine, int oraInizio, int oraFine)

Specifica sintattica

verificaDisponibilita(Auto, int, int, int, int) → int

Specifica semantica

verificaDisponibilita(a, giornoInizio, giornoFine, oraInizio, oraFine) → 0 /

La funzione controlla che nel lasso di tempo passato in input sia disponibile l'auto memorizzate nel puntatore a struttura Auto.

Precondizione:

- a è un puntatore a una struttura (Auto) correttamente allocata in memoria;
- giornoInizio è una variabile intera che contiene un valore maggiore o uguale a 0;
- giornoFine è una variabile intera che contiene un valore maggiore o uguale a 0;
- oraInizio è una variabile intera che contiene un numero maggiore o uguale a 0 e allo stesso tempo minore o uguale a 24;
- oraFine è una variabile intera che contiene un numero maggiore o uguale a 0 e allo stesso tempo minore o uguale a 24;
- giornoInizio è sempre minore di giornoFIne.

Postcondizione:

- stato è una variabile booleana che se è true, la funzione imposta la matrice a true per indicare che l'auto è disponibile nell'intervallo specificato, se è false imposta la matrice a false per indicare che l'auto non è disponibile in quell'intervallo.

Side Effect:

- la funzione non presenta side effect.

void stampaAuto(Auto a)

Specifica sintattica

stampaAuto(Auto) → void

Specifica semantica

stampaAuto(a) → void

Viene passato per parametro un puntatore alla struttura Auto e la funzione deve stampare a schermo tutti i parametri della struttura

Precondizioni:

- a è un puntatore a una struttura (Auto) correttamente allocata in memoria.

P*ostcondizione*:

- la funzione non presenza post condizioni.

Side Effect:

- la funzione non presenza side effect.

void reimpostaDisponibilitaAuto(Auto a)

Specifica sintattica

reimpostaDisponibilitaAuto(Auto) → void

Specifica semantica

reimpostaDisponibilitaAuto(a) → void

Questa funzione azzera il campo stato del puntatore alla struttura Auto. Lo stato di tutte le auto memorizzate in catalogo alla fine di questa funzione sarà impostato a false.

Precondizione:

- a è un puntatore a una struttura (Auto) correttamente allocata in memoria.

Postcondizione:

- la funzione non presenza postcondizioni.

Side Effect:

- il campo stato viene impostato a false a qualsiasi veicolo del catalogo.

•**HashTbAuto:**

AutoHashTB creaHashTBAuto(void)

Specifica Sintattica

creaHashTBAuto(void) -> AutoHashTB

Specifica Semantica

creaHashTBAuto(void) -> NULL

La funzione crea una tabella di hash inizializzata a NULL

Precondizione:

- la funzione non presenza precondizioni.

Postconidizone:

- NULL è una tabella di hash non inizializzata.

Side effect:

- la funzione non presenta side effect.

Int inserisciAuto(AutoHashTB *ht, Auto a)

Specifica Sintattica

inserisciAuto(AutoHashTB *, Auto) -> int

Specifica Semantica

inserisciAuto(ht, a) -> 0 / 1

La funzione inserisce un nuovo AutoEntry all'interno della tabella di hash passata in input. La funzione restituirà 1 se l'inserimento avrà avuto successo; invece restituirà 0 in caso contrario.

Precondizione:

- ht è una tabella di hash correttamente allocata in memoria, a è un puntatore a una struttura (Auto) correttamente allocata in memoria.

Postcondizione:

- se ritorna 1, allora l'auto è stata aggiunta con successo alla tabella di hash; se ritorna 0, l'auto non è stata aggiunta (errore o auto già presente in catalogo).

Side effect:

- Viene allocato lo spazio in memoria per l'auto aggiunta al catalogo; viene modificata la tabella di hash

Auto cercaAuto(AutoHashTB ht, const char *targa)

Specifica Sintattica

cercaAuto(AutoHashTB, const char *) -> Auto

Specifica Semantica

cercaAuto(ht, targa) -> entry / NULL

La funzione cerca un'auto all'interno della hash table tramite la targa. Se targa è valida e presente nella tabella di hash, restituisce il puntatore all'auto. Altrimenti la funzione restituisce NULL.

Precondizione:

- ht è una tabella di hash correttamente allocata in memoria
- targa è una costante di tipo stringa composta da 7 caratteri e deve essere composta da 2 lettere, 3 numeri e altre 2 lettere.

Postcondizione

- entry è il puntatore dell'auto aggiunta alla tabella di hash.

Side effect:

- La funzione non ha side effect.

Auto rimuoviAuto(AutoHashTB *ht, const char *targa)

Specifica Sintattica

rimuoviAuto(AutoHashTB *, const char *) -> Auto

Specifica Semantica

rimuoviAuto(ht, targa) -> a

La funzione cerca un'auto all'interno della hash table passata in input per eliminarla.

Precondizione:

- ht è una tabella di hash correttamente allocata in memoria,
- targa è una costante di tipo stringa composta da 7 caratteri e deve essere composta da 2 lettere, 3 numeri e altre 2 lettere.

Postconidizione:

- a è un puntatore a una struttura (Auto) correttamente allocata in memoria.

Side effect:

- Viene liberato spazio in memoria.

void distruggiHashTBAuto(AutoHashTB *ht)

Specifica Sintattica

distruggiHashTBAuto(AutoHashTB *) -> void

Specifica Semantica

distruggiHashTBAuto(ht) -> void

La funzione elimina l'intera tabella di hash e le auto contenute.

Precondizione:

- ht è una tabella di hash correttamente allocata in memoria.

Postcondizione:

- la funzione non presenza postconizioni.

Side effect:

- vengono eliminate le auto assieme alla tabella di hash date in input; viene liberato spazio in memoria.

void stampaHashTBAuto(AutoHashTB ht)

Specifica Sintattica

stampaHashTBAuto(AutoHashTB) -> void

Specifica Semantica

stampaHashTBAuto(ht) -> void

La funzione fa scorrere tutta la tabella per stampare tutte le auto nella tabella.

Precondizione:

- ht è una tabella di hash correttamente allocata in memoria.

Postcondizione:

- la funzione non presenta postcondizione.

Side effect:

- viene stampata tutta la tabella di hash data in input

void stampaHashTBPerDisp(AutoHashTB ht, int giornoInizio, int giornoFine, int oraInizio, int oraFine)

Specifica Sintattica

stampaHashTBPerDisp(AutoHashTB, int, int, int, int) -> void

Specifica Semantica

stampaHashTBPerDisp(ht, giornoInizio, giornoFine, oraInizio, oraFine) -> void

La funzione accede alla tabella di hash data in input e stampa sono le auto disponibili nel lasso di tempo dato in input.

Precondizione:

- ht è una tabella di hash correttamente allocata in memoria;
- giornoInizio è una variabile intera che contiene un valore maggiore o uguale a 0;

- giornoFine è una variabile intera che contiene un valore maggiore o uguale a 0;
- oraInizio è una variabile intera che contiene un numero maggiore o uguale a 0 e allo stesso tempo minore o uguale a 24;
- oraFine è una variabile intera che contiene un numero maggiore o uguale a 0 e allo stesso tempo minore o uguale a 24;
- giornoInizio è sempre minore di giornoFIne.

Postcondizione:

- la funzione non presenta postcondizioni.

Side effect:

- viene fatta scorrere tutta la tabella hash con un ciclo for e successivamente vengono stampate tutte le auto disponibili.

void reimpostaDisponibilitaTutteLeAuto(AutoHashTB)

Specifica Sintattica

reimpostaDisponibilitaTutteLeAuto(AutoHashTB) -> void

Specifica Semantica

reimpostaDisponibilitaTutteLeAuto(ht) -> void

la funzione rende disponibile tutte le auto presenti nella tabella di hash passata per parametro.

Precondizione:

- ht è una tabella di hash correttamente allocata in memoria.

Postcondizione:

- la funzione non presenza postconidizoni.

Side effect:

- La disponibilità delle auto presenti nel catalogo viene impostata a “disponibile”.

•HashTbUtenti:

int insertUtente(HashTB *h, Utente u)

Specifica Sintattica

insertUtente(HashTB *, Utente) -> int

Specifica Semantica

insertUtente(h, u) -> 1/0

La funzione crea un nuovo utente e successivamente lo implementa nella tabella hash data in input allocando uno spazio in memoria apposito. La funzione restituisce 1 o 0 in caso di successo oppure di insuccesso.

Precondizione:

- h è una tabella di hash correttamente allocata in memoria;
- u è un puntatore a una struttura (Utente) correttamente allocata in memoria.

Postconidzione:

- La funzione restituisce 1 in caso di successo e restituisce 0 in caso di fallimento (utente già presente oppure in caso di errore).

Side effect:

- viene allocato dello spazio in memoria;
- viene creato un utente; viene aggiunto un utente alla tabella di hash data in input.

Utente cercaUtente(HashTB, const char *)

Specifica Sintattica

cercaUtente(HashTB, const char *) -> Utente

Specifica Semantica

cercaUtente(h, CF) -> entry

La funzione farà scorrere l'intera tabella di hash per cercare un determinato utente tramite il suo CF.

Precondizione:

- h è una tabella di hash correttamente allocata in memoria e non nulla;
- Il campo CF, appartiene alla struttura Utente puntata da u ed è una stringa di 16 caratteri.

Postconidzione:

- entry è un puntatore alla struttura utente.

Side effect:

- la funzione non presenta side effect.

Utente eliminaUtente(HashTB *, const char *)

Specifica Sintattica

eliminaUtente(HashTB *, const char *) ->Utente

Specifica Semantica

eliminaUtente(h, CF) -> u

La funzione cerca (nella tabella di hash passata in input) l'utente associato al CF passato per parametro per poi successivamente restituirlo al chiamante. Se la memoria non viene liberata, la struttura restituirà NULL.

Precondizione: h è una tabella di hash correttamente allocata in memoria e non nulla; Il campo CF, appartiene alla struttura Utente puntata da u ed è una stringa di 16 caratteri.

Postconidzione: u è un puntatore alla struttura Utente.

Side effect: viene rimossa dalla tabella di hash l'utente associato al CF passato per prametro.

void distruggiHashTBUtenti(HashTB *)

Specifica Sintattica

distruggiHashTBUtenti(HashTB *) -> void

Specifica Semantica

distruggiHashTBUtenti(h) -> void

La funzione elimina tutti gli utenti registrati nella tabella di hash, per poi liberare la memoria associata alla tabella di hash passata in input.

Precondizione: h è una tabella di hash correttamente allocata in memoria.

Postconidzione: la funzione non presenta postcondizioni.

Side effect: *h sarà impostato a null; la memoria associata agli utenti e alla hash table verrà liberata.

void stampaHashTBUtenti(HashTB)

Specifica Sintattica

stampaHashTBUtenti(HashTB) -> void

Specifica Semantica

stampaHashTBUtenti(h) -> void

La funzione farà scorrere la tabella di hash in modo tale da stampare a schermo il codice fiscale e i dettagli di ogni utente.

Precondizione: h è una tabella di hash correttamente allocata in memoria.

Postconidzione: la funzione non presenta postcondizioni.

Side effect: il codice fiscale e i dettagli di ogni utente vengono stampati a schermo.

Utente loginRegisterUtente(HashTB *h)

Specifica Sintattica

loginRegisterUtente(HashTB *) -> Utente

Specifica Semantica

loginRegisterUtente(h) -> Utente

La funzione gestisce il l'accesso e la registrazione dell'utente. Se l'utente sta effettuando l'accesso con un profilo già esistente verrà effettuata l'autenticazione dell'utente richiedendo le credenziali d'accesso; se l'utente sta effettuando la registrazione, allora la funzione raccoglierà i dati dell'utente e successivamente quest'ultimi verranno memorizzati nella tabella di hash.

Precondizione: h è una tabella di hash correttamente allocata in memoria

Postconidzione: nuovoUtente (rappresenta l'utente appena registrato) è un puntatore alla struttura Utente. u (rappresenta l'utente già registrato) è un puntatore alla struttura Utente.

Side effect: in caso di una nuova registrazione: viene allocata della nuova memoria all'interno della tabella di hash e per tanto, la tabella viene modificata; Per verificare le credenziali e i dati degli utenti (sia quelli registrati che non) viene fatta scorrere le tabelle di hash).

void aggiungiPrenotazioniAStoricoUtenti(HashTB h, Lista listaaPrenotazioni)

Specifica Sintattica

aggiungiPrenotazioniAStoricoUtenti(HashTB, Lista) -> void

Specifica Semantica

aggiungiPrenotazioniAStoricoUtenti(h,listaaPrenotazioni) -> void

La funzione prende ogni prenotazione da listaaPrenotazioni e la inserisce nello storico dell'utente a cui appartiene (usando il codice fiscale per trovare l'utente nella tabella hash h). Se l'utente viene trovato, la prenotazione viene aggiunta al suo storico; altrimenti viene ignorata

Precondizione: h è una tabella di hash correttamente allocata in memoria; listaaPrenotazioni è una listaa correttamente formata che contiene elementi di tipo Prenotazione

Postconidzione: la funzione non presenta postcondizioni.

Side effect: allocazione di memoria per la copia delle prenotazioni; modifica delle strutture Utente presenti nella tabella h.

void stampaStoricoTuttiUtenti(HashTB h)

Specifica Sintattica

stampaStoricoTuttiUtenti(HashTB) -> void

Specifica Semantica

stampaStoricoTuttiUtenti(h) -> void

La funzione stampa lo storico delle prenotazioni per ciascun utente presente nella tabella hash data in input.

Precondizione: h è una tabella hash correttamente allocata contenente puntatori validi alla struttura Utente.

Postconidzione: la funzione non presenta postcondizioni.

Side effect: la funzione effettua una copia temporanea delle code (storici); la funzione può allocare, se necessario, temporaneamente la memoria per la copia delle code.

•**Prenotazione**:

PrenotazionecreaPrenotazione(char *CF, char *targa, int giornoInizio, int giornoFine, oraInizio, int oraFine)

Specifica sintattica

creaPrenotazione(char *, char *, int, int, int, int) → Prenotazione

Specifica semantica

creaPrenotazione(CF, targa, giornoInizio, giornoFine, oraInizio, oraFine) → nuovaPrenotazione

La funzione crea una nuova prenotazione, allocando dinamicamente memoria per una struttura di tipo Prenotazione, inizializzandola con i dati passati come argomenti, e restituisce un puntatore a questa nuova prenotazione.

Precondizioni: CF è una stringa composta da 16 caratteri alfanumerici; targa è una stringa valida di targa auto, composta da 7 caratteri (es. 2 lettere, 3 numeri, 2 lettere); giornoInizio e giornoFine sono interi validi che indicano il giorno di inizio e fine della prenotazione; oraInizio e oraFine sono interi validi che indicano l'ora di inizio e fine della prenotazione; giornoFine >= giornoInizio, oraFine >= oraInizio.

Postcondizioni: p è un puntatore a una struttura di tipo Prenotazione allocata dinamicamente e inizializzata con i valori forniti in input.

Side effect: la struttura Prenotazione viene allocata dinamicamente in memoria; la variabile statica id_contatore viene incrementata per garantire ID univoci.

char* ottieniCF(Utente utente)

Specifica Sintattica

ottieniCF(Utente) → char *

Specifica Semantica

ottieniCF(utente) → cf

La funzione restituisce la stringa che rappresenta il codice fiscale (CF) dell'utente passato come parametro.

Precondizioni: utente è un puntatore valido a una struttura Utente inizializzata; utente->CF è una stringa non nulla e correttamente terminata.

Postcondizioni: restituisce un puntatore alla stringa CF dell'utente.

Side effect: la funzione non presenta side effect.

char *ottieniNome(Utente utente)

Specifica Sintattica:

ottieniNome(Utente) → char *

Specifica Semantica:

ottieniNome(utente) → nome

La funzione restituisce la stringa che rappresenta il nome dell'utente.

Preconidizioni: utente è un puntatore valido a una struttura Utente inizializzata; utente->nome è una stringa non nulla.

Postcondizioni: nome è un puntatore stringa alla struttura utente.

Safe effect: la funzione non presenta side effect

char *ottieniCognome(Utente utente)

Specifica Sintattica

ottieniCognome(Utente) → char *

Specifica Semantica

ottieniCognome(utente) → cognome

La funzione restituisce la stringa che rappresenta il cognome dell'utente.

Precondizione: utente è un puntatore valido a una struttura Utente inizializzata; utente->cognome è una stringa non nulla.

Postcondizione: cognome è un puntatore stringa alla struttura utente.

Side effect: la funzione non presenta side effect

char *ottieniEmail(Utente utente)

Specifica Sintattica

ottieniEmail(Utente) → char *

Specifica Semantica

ottieniEmail(utente) → email

La funzione restituisce la stringa che rappresenta l'email dell'utente.

Precondizione: utente è un puntatore valido a una struttura Utente inizializzata; utente->email è una stringa non nulla.

Postcondizione: email è un puntatore stringa alla struttura utente.

Side effect: la funzione non presenta side effect

char *ottieniTelefono(Utente utente)

Specifica Sintattica

ottieniTelefono(Utente) → char *

Specifica Semantica

ottieniTelefono(utente) → telefono

La funzione restituisce la stringa che rappresenta il numero di telefono dell'utente.

Precondizione: utente è un puntatore valido a una struttura Utente inizializzata; utente->telefono è una stringa non nulla.

Postcondizione: telefono è un puntatore stringa alla struttura utente.

Side effect: la funzione non presenta side effect

char *ottieniPassword(Utente utente)

Specifica Sintattica

ottieniPassord (Utente) → char *

Specifica Semantica

ottieniPassword(utente) → password

La funzione restituisce la stringa che rappresenta la password dell'utente.

Precondizione: utente è un puntatore valido a una struttura Utente inizializzata; utente->password è una stringa non nulla.

Postcondizione: password è un puntatore stringa alla struttura utente.

Side effect: la funzione non presenta side effect

Coda ottieniStorico(Utente u)

Specifica Sintattica

ottieniStorico(Utente) → Coda

Specifica Semantica

ottieniStorico(utente) → storico

La funzione restituisce la coda storico associata all’utente, che contiene la lista delle prenotazioni o attività svolte in passato.

Precondizione: u è un puntatore valido a una struttura Utente inizializzata; u->storico è una coda correttamente allocata in memoria.

Postcondizione: storico è una coda associata all’utente dato in input.

Side effect: la funzione non presenta side effect

void distruggiUtente(Utente u)

Specifica Sintattica

distruggiUtente(Utente) → void

Specifica Semantica

distruggiUtente(u) → void

La funzione libera la memoria dinamica occupata dalla struttura Utente passata come parametro.

Precondizione: u è un puntatore alla struttura Utente.

Postcondizione: la funzione non presenta postcondizione

Side effect: la memoria di u è stata liberata.

void stampaUtente(Utente u)

Specifica Sintattica

stampaUtente(Utente) → void

Specifica Semantica

stampaUtente(u) → void

La funzione stampa a schermo le informazioni dell'utente u

Precondizione: u è un puntatore alla struttura Utente; I campi CF, nome, cognome, email e telefono di u sono stringhe valide e non nulle.

Postcondizione: la funzione non presenta postcondizione

Side effect: vengono stampati a schermo i dati dell'utente.

•**Coda_StoricoUtente**:

Coda nuovaCoda(void)

Specifica sintattica

nuovaCoda(void) → Coda

Specifica semantica

nuovaCoda(void) → q

la funzione crea una nuova struttura dati coda vuota, allocandola dinamicamente.

Precondizioni: la funzione non presenta precondizioni.

Postcondizioni: q è un puntatore a una coda vuota.

Side effect: q viene allocata dinamicamente.

int codaVuota(Coda q)

Specifica sintattica

codaVuota(Coda) → int

Specifica semantica

codaVuota(q) → -1 /1/ 0

la funzione verifica se la coda q è vuota.

Precondizioni: q è un puntatore a una coda correttamente allocata in memoria.

Postcondizioni: se q= NULL restituisce -1 (errore); se q esiste, restituisce 1 se la coda è vuota (numel = 0), altrimenti 0.

Side effect: la funzione non presenta side effect.

int inserisciCoda(Prenotazione val, Coda q)

Specifica sintattica

inserisciCoda(Prenotazione, Coda) → int

Specifica semantica

inserisciCoda(val, q)

La funzione te.nta di inserire val in coda alla struttura q.

Preconidizioni: q è un puntatore a una coda valida; val è una prenotazione valida (già allocata e inizializzata).

Postcondizioni: Se la coda è inesistente (NULL) o manca memoria, ritorna un codice di errore. Altrimenti, aggiunge l'elemento in fondo e restituisce 1 a indicare successo.

Side effect: Viene allocata la memoria per un nuovo nodo.

Prenotazione prelevaCoda(Coda q)

Specifica sintattica

prelevaCoda(Coda) → Prenotazione

Specifica semantica

prelevaCoda(q)

rimuove e restituisce l'elemento in testa alla coda q.

Preconidizioni: q (puntatore a una coda) non è vuota (numel > 0).

Postcondizioni: result è l'elemento che era in testa alla coda all'inizio della funzione.

Side effect: Viene deallocata la memoria del nodo rimosso.

void stampaStorico(Coda q)

Specifica Sintattica

stampaStorico(Coda) → void

Specifica semantica

stampaStorico(q) → void

La funzione stampa tutte le prenotazioni contenute nella coda q. Dovendo stampare il contenuto di una coda, creiamo una seconda coda che andrà a memorizzare gli elementi che stamperemo a schermo. Man mano che stampiamo gli elementi dalla coda originale, questa diventerà vuota. Successivamente, deallochiamo la coda.

Preconidizioni: q è un puntatore a una coda valida o NULL.

Postcondizioni: la funzione non ha postcondizioni.

Side effect: una coda viene visualizzata a schermo e de allocata

- **Controlli**:

int stringaValida(const char *str)

Specifica sintattica

stringaValida(const char *) → int

Specifica semantica

stringaValida(str) → 1/0

La funzione verifica se la stringa str è composta solo da caratteri alfabetici, numerici o spazi.

Precondizioni: str è una stringa terminata da '\0'.

Postcondizioni: la funzione restituisce 1 se tutti i caratteri in str sono alfabetici, numerici o spazi e in caso contrario restituirà 0.

Side effect: la funzione non presenta side effect.

int confrontaCaseInsensitive(const char *a, const char *b)

Specifica sintattica

confrontaCaseInsensitive(const char *, const char *) → int

Specifica semantica

confrontaCaseInsensitive(a, b) → 1/0

La funzione confronta le due stringhe a e b cercando di capire se sono uguali (non tenendo conto di sé le lettere sono maiuscole o minuscole).

Precondizioni: a e b sono due stringhe terminate da '\0'.

Postcondizioni: la funzione restituisce 1 se a e b sono uguali in caso contrario restituisce 0.

Side effect: la funzione non presenta side effect.

int validaPassword(const char *pwd)

Specifica sintattica

validaPassword(const char *) → int

Specifica semantica

validaPassword(pwd) → 1/0

La funzione verifica se pwd rispetta dei criteri ovvero se: è una stringa da almeno 8 caratteri, se contiene almeno una lettera maiuscola, una minuscola, una cifra e un carattere speciale (non alfanumerico).

Precondizioni: pwd è una stringa terminata da '\0'.

Postcondizioni: la funzione restituisce 1 se pwd rispetta i criteri prestabiliti. La funzione restituisce 0 in caso contrario.

Side effect: la funzione non presenta side effect.

int validaTarga(const char *targa)

Specifica sintattica

validaTarga(const char *) → int

Specifica semantica

validaTarga(targa) →1/0

La funzione verifica se targa rispetta dei criteri ovvero se: è una stringa lunga 7 caratteri, se i primi 2 e gli ultimi 2 caratteri sono lettere maiuscole, se i 3 caratteri centrali sono cifre.

Precondizioni: targa è una stringa terminata da '\0'.

Postcondizioni: la funzione restituisce 1 se targa rispetta i criteri prestabiliti. La funzione restituisce 0 in caso contrario.

Side effect: la funzione non presenta side effect.

int validaCognome(const char *cognome)

Specifica sintattica

validaCognome(const char *) → int

Specifica semantica

validaCognome(cognome) →1/0

La funzione verifica se cognome è una stringa non vuota contiene solo lettere e spazi.

Precondizioni: cognome è una stringa terminata da '\0'.

Postcondizioni: la funzione restituisce 1 se cognome contiene solo lettere e spazi. La funzione restituisce 0 in caso contrario.

Side effect: la funzione non presenta side effect.

int validaEmail(const char *email)

Specifica sintattica

validaEmail(const char *) → int

Specifica semantica

validaEmail(email) → 1/0

La funzione verifica se email rispetta dei criteri ovvero se: contiene almeno un @ e un punto dopo la chiocciola, con almeno un carattere tra i due e dopo il punto.

Precondizioni: email è una stringa terminata da '\0'.

Postcondizioni: la funzione restituisce 1 se email rispetta i criteri sopra citati. La funzione restituisce 0 in caso contrario.

Side effect: la funzione non presenta side effect.

int validaCodiceFiscale(const char *cf)

Specifica sintattica

validaCodiceFiscale(const char *) → int

Specifica semantica

validaCodiceFiscale(cf) → 1/0

La funzione verifica se cf rispetta dei criteri ovvero se: è lungo esattamente 16 caratteri, se contiene lettere maiuscole in posizioni specifiche (0-5, 8, 11, 15) e cifre negli altri.

Precondizioni: cf è una stringa terminata da '\0'.

Postcondizioni: la funzione restituisce 1 se cf rispetta i criteri sopra citati. La funzione restituisce 0 in caso contrario.

Side effect: la funzione non presenta side effect.

int validaTelefono(const char *numero)

Specifica sintattica

validaTelefono(const char *) → int

Specifica semantica

validaTelefono(numero) → 1/0

La funzione verifica se numero è una stringa di 10 cifre e che inizia col numero 3.

Precondizioni: numero è una stringa composta solo da numeri terminata da '\0'.

Postcondizioni: la funzione restituisce 1 se nuemero è composto da 10 cifre e che inizia col 3.

Side effect: la funzione non presenta side effect.

int confrontaPrefisso(const char* input, const char* prefisso)

Specifica sintattica

confrontaPrefisso(const char *, const char *) → int

Specifica semantica

confrontaPrefisso(input, prefisso) → 1/0

La funzione verifica se la stringa input inizia con la stringa “prefisso” seguita da uno spazio.

Precondizioni: input e prefisso sono stringhe terminate da '\0'.

Postcondizioni: la funzione restituisce 1 se input inizia con prefisso e il carattere successivo è uno spazio altrimenti la funzione restituisce 0.

Side effect: la funzione non presenta side effect.

int isNumeroCivico(const char* token)

Specifica sintattica

isNumeroCivico(const char *) → int

Specifica semantica

isNumeroCivico(token) → 1/0

La funzione verifica se la stringa token è composta solamente da cifre.

Precondizioni: token è una stringa terminata da '\0'.

Postcondizioni: la funzione restituisce 1 se token è una stringa numerica altrimenti la funzione restituisce 0.

Side effect: la funzione non presenta side effect.

int validaViaStradale(const char* input)

Specifica sintattica

validaViaStradale(const char *) → int

Specifica semantica

validaViaStradale(input) → 1/0

La funzione verifica se input rappresenta un indirizzo stradale valido. E per fare ciò deve rispettare dei requisiti ovvero: se c'è la presenza di prefisso valido all'inizio, se il prefisso è seguito dal nome della via e dal numero civico, la stringa deve contenere solo lettere oppure svariati caratteri speciali.

Precondizioni: input è una stringa terminata da '\0'.

Postcondizioni: la funzione restituisce 1 se input rispetta i requisiti richiesti in alternativa la funzione restituisce 0.

Side effect: la funzione non presenta side effect.

void capitalizza(char* nome)

Specifica sintattica

capitalizza(char *) → void

Specifica semantica

capitalizza(nome) → void

La funzione modifica la stringa nome data in input mettendo in maiuscolo la prima lettera di ogni parola e in minuscolo le altre.

Precondizioni: nome è una stringa terminata da '\0'.

Postcondizioni: la funzione non presenta postcondizioni.

Side effect: nome viene modifiata.

- **List_Prenotazione**:

Lista nuovaLista(void)

Specifica sintattica

nuovaLista(void) → Lista

Specifica semantica

nuovaLista(void) → l

La funzione crea una nuova lista vuota.

Precondizioni: la funzione non presenta precondizioni.

Postcondizioni: l è una lista vuota.

Side effect: la funzione non presenta side effect.

Lista ListaVuota(Lista l)

Specifica sintattica

ListaVuota(Lista) → int

Specifica semantica

ListaVuota(l) → 1/0

La funzione verifica se una lista è vuota.

Precondizioni: l è una lista correttamente allocata in memoria.

Postcondizioni: la funzione restituisce 1 se la Lista è vuota, altrimenti restituisce 0.

Side effect: la funzione non presenta side effect.

Lista consLista(Prenotazione val, Lista l)

Specifica sintattica

consLista(Prenotazione, Lista) → Lista

Specifica semantica

consLista(val, l) → nuovaLista

La funzione aggiunge una prenotazione (val) in cima alla lista l.

Precondizioni: val è un oggetto Prenotazione; l è una lista correttamente allocata in memoria.

Postcondizioni: l è una lista che ha come primo nodo un elemento contenente vale la coda di l è la lista originaria l.

Side effect: viene allocata memoria dinamica per il nuovo nodo.

Lista codaLista(Lista l)

Specifica sintattica

codaLista(Lista) → Lista

Specifica semantica

codaLista(l) → temp

La funzione ti permette di ottenere la lista senza il primo elemento, ovvero la coda di l. Se l è NULL anche temp sarà NULL.

Precondizioni: l è una lista correttamente allocata in memoria.

Postcondizione: temp è la lista data in input senza il primo elemento.

Side effect: la funzione non presenta side effect.

Prenotazione ottieniPrima(Lista l)

Specifica sintattica

ottieniPrimo (Lista) → Prenotazione

Specifica semantica

ottieniPrimo (l) → e

La funzione ti permette di leggere il primo elemento della lista data in input.

Precondizioni: l è una lista correttamente allocata in memoria.

Postcondizione: e è il primo elemento di l oppure NULL in caso l è vuota.

Side effect: la funzione non presenta side effect.

Lista copiaProfondaLista(Lista l)

Specifica sintattica

copiaProfondaLista(Lista) → Lista

Specifica semantica

copiaProfondaLista(l) → nuova

La funzione fa una copia (nuova) della lista data in input (l).

Precondizioni: l è una lista correttamente allocata in memoria.

Postcondizione: nuova è una lista correttamente allocata in memoria.

Side effect: nuova viene allocata dinamicamente.

void distruggiLista(Lista l)

Specifica sintattica

distruggiLista(Lista) → void

Specifica semantica

distruggiLista(l) → void

La funzione libera tutta la memoria occuopata da l e dalle sue prenotazioni.

Precondizioni: l è una lista correttamente allocata in memoria.

Postcondizione: la funzione non presenta postcondizioni.

Side effect: l viene deallocata dinamicamente.

void visPrenotazioniPerUtente(Lista l, const char *CF)

Specifica sintattica

visPrenotazioniPerUtente(Lista, const char *) → void

Specifica semantica

visPrenotazioniPerUtente(l, CF) → void

La funzione stampa tutte le prenotazioni presenti in l corrispondenti a CF.

Precondizioni: l è una lista correttamente allocata in memoria, CF è una stringa composta da 16 caratteri alfanumerici.

Postcondizione: la funzione non presenta postcondizioni.

Side effect: vengono stampati dei dati di l a schermo.

void stampaListaPrenotazioni(Lista l)

Specifica sintattica

stampaListaPrenotazioni(Lista) → void

Specifica semantica

stampaListaPrenotazioni(l) → void

La funzione stampa tutte le prenotazioni presenti in l.

Precondizioni: l è una lista correttamente allocata in memoria.

Postcondizione: la funzione non presenta postcondizioni.

Side effect: vengono stampati tutti i dati di l a schermo.

Lista filtraPrenotazioniPerCF(Lista l, const char *CF)

Specifica sintattica

filtraPrenotazioniPerCF(Lista, const char *) → Lista

Specifica semantica

filtraPrenotazioniPerCF(l, CF) → risultato

La funzione restituisce una nuova lista filtrata contenente solo le prenotazioni (prese da l) corrispondenti a CF.

Precondizioni: l è una lista correttamente allocata in memoria, CF è una stringa composta da 16 caratteri alfanumerici.

Postcondizione: risultato è una lista correttamente allocata in memoria.

Side effect: risultato viene allocata dinamicamente; viene fatta scorrere l senza modificarla.

- **Utile_DevMenu:**

int trovaMarca(const char *marca)

Specifica sintattica

trovaMarca(const char *) → int

Specifica semantica

trovaMarca(marca) → i

La funzione serve per trovare la posizione di una determinata marca (data in input) cercandola all'interno di un database delle marche. Viene restituito l'indice i (i ≥ 0) se viene trovata la marca (l'indice è uguale alla posizione della marca all'interno del database). In caso contrario viene stampato -1.

Precondizioni: marca è una stringa terminata da '\0' non vuota, databaseMarche è un array inizializzato di strutture MarcaModelli.

Postcondizione: i è una variabile intera ≥ 0.

Side effect: la funzione non presenta side effect.

int modelloPresente(const char *modello, const MarcaModelli *marca)

Specifica sintattica

modelloPresente(const char *, const MarcaModelli *) → int

Specifica semantica

modelloPresente(modello, marca) → 1/0

La funzion controlla se modello (dato in input) esiste nella lista di modelli associati alla marca passata in input.

Precondizioni: marca è una stringa terminata da '\0' non vuota, modello è una stringa non vuota che termina '\0'.

Postcondizione: la funzione restituisce 1 se il modello esiste nella lista di modelli e restituisce 0 in caso contrario.

Side effect: la funzione non presenta side effect.

void inserisciMarcaModello(char *marca, char *modello)

Specifica sintattica

inserisciMarcaModello(char *, char *) → void

Specifica semantica

inserisciMarcaModello(marca, modello) → void

La funzione fa selezionare marca e modello all'utente da una lista. Verificando successivamente che la marca e modello esistono.

Precondizioni: marca è una stringa terminata da '\0' non vuota e puntatore a buffer di dimensione sufficiente (>= MAX_LUNGHEZZA); modello è una stringa non vuota che termina '\0' e puntatore a buffer di dimensione sufficiente (>= MAX_LUNGHEZZA); databaseMarche contiene un elenco predefinito di marche e modelli

Postcondizione: la funzione non presenta postcondizioni.

Side effect: marca e modello vengono salvati nei parametri passati per riferimento.

void aggiungiAutoInterattivo(AutoHashTB *ht)

Specifica sintattica

aggiungiAutoInterattivo(AutoHashTB *ht) → void

Specifica semantica

aggiungiAutoInterattivo(ht) → void

La funzione permette allo sviluppatore di inserire di una nuova auto nel sistema con validazione di tutti i campi.

Precondizioni: ht è una tabella hash valida e inizializzata

Postcondizione: la funzione non presenta postcondizioni.

Side effect: Viene aggiunta una nuova auto alla tabella hash e ne vengono validati tutti i campi.

Lista gestisciMenuSviluppatore(int scelta, AutoHashTB *ht, Lista l, UtentiHashTB tabUtenti)

Specifica sintattica

gestisciMenuSviluppatore(int, AutoHashTB *, Lista, UtentiHashTB) → Lista

Specifica semantica

gestisciMenuSviluppatore(scelta, ht, l, tabUtenti) → nuovaLista
Gestisce le varie operazioni possibili nel menu sviluppatore a seconda della scelta effettuata.

Precondizioni: ht, l, tabUtenti sono strutture dati inizializzate correttamente in memoria; scelta è un valore intero che può assumere un valore fra 1 e 7.

Postcondizione: l è una lista correttamente allocata in memoria.

Side effect: la funzione ha diverse funzionalità, e può: aggiungere un'auto al catalogo, eliminare un'auto dal catalogo, stampare tutte le auto del catalogo, visualizzare lo storico prenotazioni oppure avanzare il tempo di una settimana.

int mostraMenuSviluppatore(void)

Specifica sintattica

mostraMenuSviluppatore(void) → int

Specifica semantica

mostraMenuSviluppatore(void) → scelta
La funzione stampa a schermo il menu di scelta degli sviluppatori.

Precondizioni: la funzione non presenta precondizioni.

Postcondizione: scelta è un numero intero compreso fra 1 e 7.

Side effect: viene stampato a schermo il menu sviluppatore e ritorna la scelta dell'utente. MODIFICAMI

int selezionaRuolo (void)

Specifica sintattica

selezionaRuolo (void) → int

Specifica semantica

selezionaRuolo (void) → ruolo

La funzione permette a chi manda in esecuzione il programma di scegliere se è un utente oppure uno sviluppatore.

Precondizioni: la funzione non presenta precondizioni.

Postcondizione: ruolo è un numero intero compreso fra 0 e 3.

Side effect: viene stampato a schermo il menu sviluppatore e ritorna la scelta dell'utente. MODIFICAMI

- **Utile_UtenteMenu:**

Lista switchUtente(int scelta, Utente u, Lista l, AutoHashTB tabAuto)

Specifica sintattica

switchUtente(int, Utente, Lista, AutoHashTB) → Lista

Specifica semantica

switchUtente(scelta, u, l, tabAuto) → l

La funzione gestisce la scelta dell'utente.

Precondizioni: u è un puntatore a una struttura (Utente) correttamente allocata in memoria; l è una lista correttamente allocata in memoria; tabAuto è una tabella di hash; scelta è un intero.

Postcondizione: l è una lista correttamente allocata in memoria.

Side effect: la funzione in base alla scelta dell'utente può svolgere più azioni a infatti può: può aggiungere prenotazioni, cancellarle o semplicemente mostrare informazioni.

int menuUtente(void)

Specifica sintattica

menuUtente(void) → int

Specifica semantica

menuUtente(void) → scelta

La funzione stampa a schermo il menu di scelta degli utenti e acquisisce la scelta dell'utente.

Precondizioni: la funzione non presenta precondizioni.

Postcondizione: scelta è una variabile intera che può valere da 1 a 6.

Side effect: la funzione stampa a schermo il menu sci scelta.

Lista prenotazioneAuto(Lista l, Utente u, AutoHashTB tabAuto)

Specifica sintattica

prenotazioneAuto(Lista, Utente, AutoHashTB) → Lista

Specifica semantica

prenotazioneAuto(l, u, tabAuto) → l

La funzione permette all'utente di effettuare una prenotazione, validando dati e disponibilità

Precondizioni: u è un puntatore a una struttura (Utente) correttamente allocata in memoria; l è una lista correttamente allocata in memoria; tabAuto è una tabella di hash; scelta è un intero.

Postcondizione: l è una lista correttamente allocata in memoria.

Side effect: la lista viene aggiornata.

void visualizzaAutoDisponibili(AutoHashTB ht)

Specifica sintattica

visualizzaAutoDisponibili(AutoHashTB) → void

Specifica semantica

visualizzaAutoDisponibili(ht) → void

La funzione permette all'utente di visualizzare tutte le auto disponibili in un determinato intervallo di tempo.

Precondizioni: ht è una tabella hash allocata dinamicamente.

Postcondizione: l è una lista correttamente allocata in memoria.

Side effect: vengono stampati dei messaggi a schermo.

int calcolaOreTotali(int giornoInizio, int giornoFine, int oraInizio, int oraFine)

Specifica sintattica

calcolaOreTotali(int, int, int, int) → int

Specifica semantica

calcolaOreTotali(giornoInizio, giornoFine, oraInizio, oraFine) → prezzoOrario * oreTotali

Calcola la durata totale della prenotazione in ore basandosi su giorni e ore fornite.

Precondizioni: giornoInizio e giornoFine sono interi validi che indicano il giorno di inizio e fine della prenotazione; oraInizio e oraFine sono interi validi che indicano l'ora di inizio e fine della prenotazione; giornoFine >= giornoInizio, oraFine >= oraInizio.

Postcondizione: prezzoOrario è un numero float ≥0; oreTotali è un numero intero (int) ≥ 0.

Side effect: la funzione non presenta side effect.

float calcolaPrezzo(Auto a, int giornoInizio, int giornoFine, int oraInizio, int oraFine)

Specifica sintattica

calcolaPrezzo(Auto, int, int, int, int) → float

Specifica semantica

calcolaPrezzo(a, giornoInizio, giornoFine, oraInizio, oraFine) → prezzoOrario * oreTotali

La funzione calcola quanto costa una prenotazione (senza sconti). Viene calcolato creando due variabili locali e facendo: prezzo orario (variabile float) dell'auto per ore totali (variabile intera).

Precondizioni: a è un puntatore a un oggetto Auto inizializzato; giornoInizio e giornoFine sono due campi (interi) della struttura periodo e hanno una dimensione di minimo 0 e massimo 6; oraInizio e oraFine sono due campi (interi) della struttura periodo e hanno una dimensione di minimo 0 e massimo 23 (l'intervallo rappresenta un periodo valido, se giornoInizio = giornoFine, allora oraInizio < oraFine).

Postcondizione: prezzo orario è una variabile locale di tipo float; oreTotali è una variabile locale di tipo intero.

Side effect: la funzione non presenta side effect.

float calcolaSconto(float prezzo, int giornoInizio, int giornoFine)

Specifica sintattica

calcolaSconto(float, int, int) → float

Specifica semantica

calcolaSconto(prezzo, giornoInizio, giornoFine) → prezzo

La funzione restituisce prezzo scontato del 10% ammesso che sia l'inizio che giornoInizio che giornoFine avvengano di sabato o domenica. In caso contrario, la restituisce prezzo in maniera invariata.

Precondizioni: giornoInizio e giornoFine sono variabili intere e il loro valore può essere minimo 0 e massimo 6; prezzo è un valore float positivo.

Postcondizione: prezzo è un valore float positivo (passato per parametro). Durante lo svolgimento della funzione il contenuto di prezzo può mutare.

Side effect: Questa funzione controlla se una prenotazione cade sabato o domenica.

float calcolaScontoOre(float prezzo, int giornoInizio, int giornoFine, int oraInizio, int oraFine)

Specifica sintattica

calcolaScontoOre(float, int, int, int, int) → float

Specifica semantica

calcolaScontoOre(prezzo, giornoInizio, giornoFine, oraInizio, oraFine) → sconto

La funzione restituisce prezzo scontato del 20% se la durata della prenotazione è maggiore o uguale a 10 ore. In alternativa, la funzione restituisce il prezzo invariato. Se succede prezzo viene scontato del 10%.

Precondizioni: giornoInizio e giornoFine sono variabili intere e il loro valore può essere minimo 0 e massimo 6; oraInizio e oraFine sono due campi (interi) della struttura periodo e hanno una dimensione di minimo 0 e massimo 23; prezzo è un valore float positivo.

Postcondizione: prezzo è un valore float positivo (passato per parametro). Durante lo svolgimento della funzione il contenuto di prezzo può mutare.

Side effect: Questa funzione controlla se una prenotazione dura per 10 ore o per di più. Se succede prezzo viene scontato del 20%.

float calcolaScontoFasciaOraria(float prezzo, int oraInizio, int oraFine)

Specifica sintattica

calcolaScontoFasciaOraria(float, int, int) → float

Specifica semantica

calcolaScontoFasciaOraria(prezzo, oraInizio, oraFine) → sconto

La funzione restituisce prezzo (dato precedentemente in input) scontato del 15% se la prenotazione ricade anche solo in parte nella fascia oraria notturna (ovvero dalle 20 alle 8 del mattino).

Precondizioni: oraInizio e oraFine sono due campi (interi) della struttura periodo e hanno una dimensione di minimo 0 e massimo 23; prezzo è un valore float positivo.

Postcondizione: prezzo è un valore float positivo (passato per parametro). Durante lo svolgimento della funzione il contenuto di prezzo può mutare.

Side effect: La funzione controlla se oraInizio è maggiore o uguale a 20; La funzione controlla che oraFine è minore o uguale a 8. Se succede prezzo viene scontato del 15%.

float calcolaPrezzoFinale(Auto a, int giornoInizio, int giornoFine, int oraInizio, int oraFine)

Specifica sintattica

calcolaPrezzoFinale(Auto, int, int, int, int) → float

Specifica semantica

calcolaPrezzoFinale(a, giornoInizio, giornoFine, oraInizio, oraFine) → prezzoFinal

Calcola il prezzo totale della prenotazione applicando eventuali sconti: quello per la fascia oraria notturna, quello per una durata di 10 o più ore, e quello previsto se la prenotazione avviene durante il sabato o la domenica.

Precondizioni: a è un puntatore alla struttura auto; giornoInizio e giornoFine sono variabili intere e il loro valore può essere minimo 0 e massimo 6; oraInizio e oraFine sono due campi (interi) della struttura periodo e hanno una dimensione di minimo 0 e massimo 23.

Postcondizione: prezzo è un valore float positivo.

Side effect: La funzione non presenta side effect.

float calcolaPrezziPrenotazioni(Lista prenotazioni, Utente u, AutoHashTB ht)

Specifica sintattica

calcolaPrezziPrenotazioni(Lista, Utente, AutoHashTB) → float

Specifica semantica

calcolaPrezziPrenotazioni(prenotazioni, u, ht) → totale /-1

Questa funzione stampa tutti i prezzi delle prenotazioni fatte da un utente e li somma, tenendo conto di tutti gli sconti. Se non ci sono prenotazioni, la funzione restituisce il valore -1.

Precondizioni: u è un puntatore a una struttura (Utente) correttamente allocata in memoria; prenotazioni è una lista non vuota; ht è una tabella correttamente allocata in memoria e non vuota;

Postcondizione: prezzo è una variabile di tipo float.

Side effect: vengono stampati dei messaggi a schermo, viene de allocata la lista listaUtente; La struttura listaUtente potrebbe essere modificata.

- **Main:**

int main(void)

Specifica sintattica

main(void) → int

Specifica semantica

main(void) → 0

È la funzione principale del programma. Inizializza le strutture dati e gestisce il flusso del programma tramite un menu interattivo che permette all'utente di accedere come utente o sviluppatore.

Precondizioni: la funzione non presenta precondizioni.

Postcondizione: 0 è un valore intero.

Side effect: vengono effettuate operazioni di input/output (per esempio: printf, scanf); alla fine vengono liberate tutte le strutture dati (uso di distruggiHashTBUtenti, distruggiAutoHashTB, distruggiLista); viene pulita la console (pulisciConsole()), viene messa in pausa l'esecuzione (pausaConsole()), e vengono visualizzati messaggi di sistema.

- **Test:**

void caricaUtentiTest(UtentiHashTB *tab)

Specifica sintattica

caricaUtentiTest(UtentiHashTB*) → void

Specifica semantica

caricaUtentiTest(tab) → void

La funzione, con lo scopo di effettuare delle operazioni di test automatici, popola una tabella hash di utenti (tab) con un insieme di strutture Utente con dati fittizi (CF, nome, cognome).

Precondizioni:

- tab è un puntatore a una struttura UtentiHashTB inizializzata correttamente.

Postcondizioni:

- la funzione non presenta postcondizioni

Side effect:

- vengono creati e inseriti 5 utenti nella hash table.

void caricaAutoTest(AutoHashTB *tab)

Specifica sintattica

caricaAutoTest(AutoHashTB*) → void

Specifica semantica

caricaAutoTest(tab) → void

La funzione, con lo scopo di effettuare delle operazioni di test automatici, popola una tabella hash di auto (tab) con un insieme di strutture Utente con dati fittizi (targa, modello, marca, categoria, costo orario).

Precondizioni:

- tab è un puntatore a una struttura AutoHashTB inizializzata correttamente.

Postcondizioni:

- la funzione non presenta postcondizioni

Side effect:

- vengono create e inserite 5 auto nella hash table.

Lista caricaPrenotazioni(Lista prenotazioni, int g_i, int g_f, int o_i, int o_f, char * CF, char *targa)

Specifica sintattica

caricaPrenotazioni(Lista, int, int, int, int, char*, char*) → Lista

Specifica semantica

caricaPrenotazioni(prenotazioni, g_i, g_f, o_i, o_f, CF, targa)→ prenotazioni

La funzione crea una nuova prenotazione con i parametri indicati e la aggiunge in testa alla lista di prenotazioni.

Precondizioni:

- La lista prenotazioni è valida (può anche essere vuota).
- CF è un codice fiscale valido.
- targa è una targa valida.
- Le date e ore sono coerenti (g_i ≤ g_f, o_i ≤ o_f).

Postcondizioni:

- prenotazioni è una lista correttamente allocata in memoria

Side effect:

- viene allocata della memoria.

void eseguiPrenotazioneSimulata(Lista prenotazioni, UtentiHashTB tabUtenti, AutoHashTB tabAuto, int g_inizio, int g_fine, int ora_inizio, int ora_fine, char*targa, char*CF, FILE *output_fp)

Specifica sintattica

eseguiPrenotazioneSimulata(Lista, UtentiHashTB, AutoHashTB, int, int, int, int, char*, char*, FILE*) → void

Specifica semantica

eseguiPrenotazioneSimulata(prenotazioni, tabUtenti, tabAuto, g_inizio, g_fine, ora_inizio, ora_fine, targa, CF, output_fp)→ void

La funzione esegue una prenotazione simulata tra un utente e un'auto, se entrambi sono presenti. Scrive l'esito su file.

Precondizioni:

- hash table tabUtenti e tabAuto devono essere inizializzati.
- CF e targa sono due stringhe valide.

- prenotazioni è una lista inizializzata.
- output_fp deve essere aperto in scrittura.

Postcondizioni:

- La funzione non presenta postcondizioni.

Side effect:

- La funzione apre in scrittura, modifica e chiude sul file dato come parametro.
- La funzione potrebbe modificare la lista prenotazioni.
- La funzione scrive il risultato del test su output_fp

void eseguiCalcoloCostoSimulato(UtentiHashTB tabUtenti, AutoHashTB tabAuto, Lista prenotazioni, FILE *output_fp)

Specifica sintattica

eseguiCalcoloCostoSimulato(UtentiHashTB, AutoHashTB, Lista, FILE*) → void

Specifica semantica

eseguiCalcoloCostoSimulato(tabUtenti, tabAuto, prenotazioni, output_fp) → void

La funzione esegue una prenotazione simulata tra un utente e un'auto, se entrambi sono presenti. Scrive l'esito su file.

Precondizioni:

- hash table tabUtenti e tabAuto devono essere inizializzati.
- CF e targa sono due stringhe valide.
- prenotazioni è una lista inizializzata.
- output_fp deve essere aperto in scrittura.

Postcondizioni:

- La funzione non presenta postcondizioni.

Side effect:

- La funzione apre in scrittura, modifica e chiude sul file dato come parametro.
- La funzione potrebbe modificare la lista prenotazioni.
- La funzione scrive il risultato del test su output_fp.

int eseguiCasoDiTest(char *tc_id, char *test_tipo)

Specifica sintattica

eseguiCasoDiTest(char*, char*)→ int

Specifica semantica

eseguiCasoDiTest (tc_id, test_tipo) → pass

La funzione esegue un caso di test specificato da tc_id e test_tipo. Elabora il file di input, genera il file di output e lo confronta con il file oracolo. Restituisce 1 se i file coincidono, 0 altrimenti o in caso di errore.

Precondizioni:

- tc_id è una stringa valida che identifica il nome della cartella contenente i file del test case (es. "TC01").
- tipo_di_test è una stringa che può assumere i valori "PRENOTA", "CALCOLO", "VISUALIZZA" o "STORICO", ognuno dei quali attiva una specifica logica di elaborazione nel corpo della funzione.
- Nella cartella test/tc_id/ devono esistere i file: tc_id_input.txt /contiene le istruzioni di input per il test) e tc_id_oracle.txt (contiene l'output atteso).
- Le funzioni esterne per gestione utenti, auto e prenotazioni devono essere disponibili e funzionanti.

Postcondizioni:

- La funzione restituisce pass, un intero che vale 1 se l'output del test corrisponde all'oracolo, 0 altrimenti.

Side effect:

- La funzione apre in scrittura, modifica e chiude sul file dato come parametro.
- La funzione potrebbe modificare la lista prenotazioni.

void esegui_test_suite(const char *suite_file, const char *result_file)

Specifica sintattica

esegui_test_suite(suite_file, result_file)→ void

Specifica semantica

void esegui_test_suite(suite_file, result_file) → void

La funzione si occupa di gestire l'esecuzione dei test automatizzati. Prima di tutto, costruisce i percorsi completi dei file aggiungendo la cartella "test/" ai nomi dei file della suite e dei risultati. Apre poi il file della

suite in modalità lettura e quello dei risultati in modalità scrittura. Per ogni riga presente nel file della suite) che deve contenere un identificativo del test e il tipo di test (la funzione richiama eseguiCasoDiTest, che esegue il caso di test specificato. L'esito del test, che può essere "PASS" (superato) o "FAIL" (fallito), viene sia stampato sullo schermo che registrato nel file dei risultati.

Precondizioni:

- suite_file è il file contenente l'elenco dei test case da eseguire.
- result_file è il file su cui verranno scritti i risultati (PASS/FAIL) di ogni test.
- I file "test/<suite_file>" e "test/<result_file>" devono essere nomi validi.
- Il file suite deve esistere e contenere righe nel formato: <tc_id> <tipo_di_test>.

Postcondizioni:

- La funzione non restituisce nulla.

Side effect:

- La funzione segue in sequenza tutti i casi di test indicati nel file di suite.
- La funzione stampa a schermo l'esito di ciascun test.
- La funzione scrive su result_file l'esito dettagliato dei test.

int main(int argc, char *argv[])

Specifica sintattica

main(int, char *) → int

Specifica semantica

main(argc, argv[]) → int

main chiama la funzione esegui_test_suite in modo tale da far partire tutti i casi di test.

Precondizioni:

- argc è una variabile intera che contiene il numero dei parametri passati alla funzione e deve essere diverso da 3.
- argv[] deve contenere 3 stringhe: nome del programma, i nomi dei file (risultati.txt e test_suite.txt).

Postcondizioni:

- La funzione restituisce il valore 0 se è andato tutto a buon fine.

Side effect:

- La funzione non presenta side effect.

5) RAZIONALE CASI DI TEST

Nel progetto è implementato un sistema di testing, che si trova in Test.c e che mediante le funzioni dichiarate permette di eseguire una serie di tipi di test automatici.

Tipi di test in Test.c

Panoramica dei tipi di test:

- PRENOTA: questo test permette di vedere la corretta creazione della prenotazione e aggiorna la disponibilità dell'auto di conseguenza.
- CALCOLO: questo test permette di calcolare il costo del noleggio in base al tempo di utilizzo.
- VISUALIZZA: questo test permette di verificare la corretta visualizzazione delle auto disponibili in base alla disponibilità.
- STORICO: questo test permette di simulare il comportamento dello storico e quindi la sua corretta visualizzazione.

Funzionamento base dei casi di test

- Viene letto il file testsuite.txt, che contiene il TCID (un identificativo del test case) e il tipo di test (che sarà nel dominio PRENOTA, CALCOLO, VISUALIZZA, STORICO).
- Vengono caricati i file TCID_input.txt per prendere i campi di input da caricare, poi viene letto TCID_oracle.txt che contiene l'output corretto che il programma dovrà avere nel caso in cui vada tutto bene. Poi alla fine ci sarà TCID_output.txt che è il file che conterrà l'output del programma.

- Se l'oracolo e l'output combaciano il programma scriverà nel file risultati.txt che il test case ha avuto successo (PASS), altrimenti che non ha avuto successo (FAIL) nel formato "TCID – TipoDiDato – PASS/FAIL".
- Passa al caso di test successivo ripetendo questo iter fino alla fine.

Test per la corretta creazione prenotazione e aggiornamento della disponibilità dell'auto (singolo utente e singola macchina)

Il file di input contiene le informazioni della prenotazione e sono formattate nel seguente modo:

g_inizio g_fine o_inizio o_fine codiceFiscale targa

dove g_inizio e g_fine è rispettivamente giorno di inizio e giorno di fine. Mentre o_inizio e o_fine è rispettivamente ora di inizio e ora di fine.

Esempio: 1 3 12 22 DNTCRL65S67M126L AB123CD

L'oracolo invece sarà la stampa della prenotazione in modo corretto, avrà come formato il seguente:

ID Prenotazione: 0

CF: DNTCRL65S67M126L

Targa: AB123CD

Periodo: dal Lunedi' ore 12:00 al Mercoledi' ore 22:00

In questi casi di test verifichiamo con un singolo utente e una singola macchina.

Casi di test PRENOTAZIONE

- **TC01:** testiamo il caso in cui è presente il giorno di inizio e fine e l'ora di inizio e fine della prenotazione con codice fiscale e targa giusti.
- **TC02:** testiamo il caso in cui il giorno di inizio è maggiore del giorno di fine, orario regolare e codice fiscale e targa giusti. Il programma non permetterà di registrare la prenotazione.
- **TC03:** testiamo il caso in cui l'orario di inizio è maggiore dell'orario di fine e il giorno di inizio è lo stesso (Lunedì). Il programma non permetterà di registrare la prenotazione.
- **TC04:** testiamo il caso in cui alcuni campi siano delle lettere e non dei numeri. Il programma non accetterà l'input.

- **TC05:** testiamo il caso in cui occupiamo tutta la settimana facendo prenotazioni multiple, andando a occupare tutte le ore del giorno. Alla fine proviamo a fare un'altra prenotazione nella stessa settimana, ma non riuscirà e il programma ci stampa che la "Prenotazione non è riuscita".
- **TC06:** testiamo il caso in cui il giorno di inizio e fine va oltre il dominio dei valori accettabili (vanno da 1 a 7), poi proviamo anche il caso in cui i valori sono negativi e quindi non accettabili (sempre fuori dal dominio).

Test per il calcolo del costo del noleggio in base al tempo di utilizzo (singolo utente e singola macchina)

Il file di input contiene le informazioni della prenotazione e sono formattate nel seguente modo:

> g_inizio g_fine o_inizio o_fine codiceFiscale targa
> dove g_inizio e g_fine è rispettivamente giorno di inizio e giorno di fine. Mentre o_inizio e o_fine è rispettivamente ora di inizio e ora di fine.

Esempio: 1 3 12 22 DNTCRL65S67M126L AB123CD

L'oracolo invece sarà la stampa del calcolo del costo in base agli sconti e al tempo di utilizzo:

Costo totale per Carla Danti (DNTCRL65S67M126L): 45.00 €

Vi sono tre sconti che si attivano in base al tempo di utilizzo, il primo è lo sconto notturno che se l'auto viene prenotata nella fascia oraria che va dalle 20 di sera fino alle 8 di mattina ha diritto a 15% di sconto, il secondo è lo sconto inerente alle ore di utilizzo che se arriva a 10 ore o più si ha diritto al 20% e il terzo è lo sconto del weekend, se si fanno prenotazioni nel weekend si ha diritto al 10% di sconto.

In questi casi di test verifichiamo con un singolo utente e una singola macchina.

Casi di test CALCOLO

- **TC07:** testiamo il caso in cui calcoliamo il costo del noleggio senza alcuno sconto. Il programma scriverà nel file di output il costo totale che dovrà pagare l'utente senza nessuno sconto.

- **TC08:** testiamo il caso in cui calcoliamo il costo del noleggio con lo sconto delle prenotazioni nel weekend. Il programma scriverà nel file di output il costo totale scontato del 10%.
- **TC09:** testiamo il caso in cui calcoliamo il costo del noleggio con lo sconto delle prenotazioni notturne. Il programma scriverà nel file di output il costo totale scontato del 15%.
- **TC10:** testiamo il caso in cui calcoliamo il costo del noleggio con lo sconto delle prenotazioni per 10 ore o più. Il programma scriverà nel file di output il costo totale scontato del 20%.
- **TC11:** testiamo il caso in cui calcoliamo il costo del noleggio con tutti e tre gli sconti applicati. Il programma scriverà nel file di output il costo totale scontato del 10%, del 15% e del 20%.
- **TC12:** testiamo il caso in cui calcoliamo il costo del noleggio con più prenotazioni per applicare più volte gli sconti. Il programma stamperà il costo totale sommando i costi parziali, già scontati, delle prenotazioni.

Test per la corretta visualizzazione delle auto disponibili in base alla disponibilità

Il file di input contiene le informazioni della prenotazione e sono formattate nel seguente modo:

g_inizio g_fine o_inizio o_fine
dove g_inizio e g_fine è rispettivamente giorno di inizio e giorno di fine. Mentre o_inizio e o_fine è rispettivamente ora di inizio e ora di fine.

Esempio: 3 3 10 20

L'oracolo sarà la stampa delle auto disponibili nella fascia giorno/ora impostata. Sarà salvato nel file nel seguente modo:
Auto disponibili nella fascia selezionata:

Targa: AB123CD
Marca: Fiat
Modello: Panda

Indirizzo: Via Roma 10
Anno: 2000
Costo orario: 15.00 €

Targa: AA555BB
Marca: Toyota
Modello: Corolla
Indirizzo: Viale Salerno 23
Anno: 2015
Costo orario: 25.00 €

In questo caso proveremo con 5 auto di test, già caricate nella tabella di Hash auto.

Casi di test VISUALIZZA

- **TC13:** testiamo il caso in cui non ci sono prenotazioni, quindi vedremo tutte le macchine disponibili. Il programma scriverà nel file dell'output scriverà le auto disponibili.
- **TC14:** testiamo il caso in cui diamo dei valori scorretti e che non rispettano il dominio dei possibili valori. Il programma scriverà nel file di output tre messaggi di errore.
- **TC15:** testiamo il caso in cui tutte le auto sono state prenotate per la fascia giorno/ora passata dal file di input. Il programma scriverà nel file di output un messaggio che recita che nessuna auto è disponibile per la fascia giorno/ora selezionata.

Test per la simulazione dello storico delle prenotazioni (singolo utente e singola macchina)

Il file di input contiene le informazioni della prenotazione e sono formattate nel seguente modo:

g_inizio g_fine o_inizio o_fine codiceFiscale targa

dove g_inizio e g_fine è rispettivamente giorno di inizio e giorno di fine. Mentre o_inizio e o_fine è rispettivamente ora di inizio e ora di fine.

Esempio: 1 3 12 22 DNTCRL65S67M126L AB123CD

L’oracolo sarà la stampa dello storico, avranno una visualizzazione simile alla stampa delle prenotazioni. Sarà salvato nel file in questo modo:

Utente: Carla Danti (CF: DNTCRL65S67M126L)
Prenotazione #1:
ID Prenotazione: 20
CF: DNTCRL65S67M126L
Targa: AB123CD
Periodo: dal Lunedi' ore 10:00 al Martedi' ore 19:00

In questi casi di test verifichiamo con un singolo utente e una singola macchina.

Casi di test STORICO

- **TC16:** testiamo il caso in cui non passiamo il special case “nessun dato” in input e proviamo a far stampare lo storico. Il programma scriverà nel file di output che nessuna prenotazione è presente nello storico.
- **TC17:** testiamo il caso in cui passiamo dei valori validi e in cui abbiamo una prenotazione che saranno caricate nello storico. Il programma scriverà nel file di output lo storico dell’utente che ha prenotato.
- **TC18:** testiamo il caso in cui passiamo più prenotazioni che saranno caricate nello storico. Il programma scriverà nel file di output lo storico dell’utente con le prenotazioni.

Test con più utenti e più macchine

Adesso testeremo le funzioni con più utenti e più macchine, questi ultimi sono già stati caricati dal Test.c. Faremo quindi altri test per i tipi di test STORICO, CALCOLO, PRENOTA (già abbiamo fatto i casi di test di VISUALIZZA).

Casi di test PRENOTA

- **TC19:** testiamo il caso in cui più utenti prenotino lo stesso veicolo, le prenotazioni non saranno sovrapposte garantendo la disponibilità. Il programma scriverà nel file di output la stampa della prenotazione, che quindi sarà andata a buon fine.

- **TC20:** testiamo il caso in cui abbiamo tre prenotazioni, ma una di queste si sovrappone a una delle precedenti. Il programma scriverà nel file di output la stampa delle prenotazioni e che una prenotazione non è riuscita.
- **TC21:** testiamo il caso in cui proviamo a prenotare più macchine, con più utenti diversi. Il programma scriverà nel file di output la stampa delle prenotazioni avvenute con successo.
- **TC22:** testiamo il caso in cui l'utente prenota nella stessa fascia giorno/ora due veicoli diversi. Il programma scriverà nel file di output la stampa delle prenotazioni.
- **TC23:** testiamo il caso in cui abbiamo più prenotazioni e due di queste ha valori del giorno/ora sbagliati. Il programma scriverà nel file di output la stampa delle prenotazioni e scriverà che due prenotazioni non sono riuscite.

Casi di test CALCOLO

- **TC24:** testiamo il caso in cui due utenti prenotano la stessa macchina per la fascia giorno/ora diversa. Il programma scriverà nel file di output i costi totali che dovranno pagare ciascun utente. È presente lo sconto per 10 o più ore di prenotazione.
- **TC25:** testiamo il caso in un utente prenota per più auto per fascia giorno/ora diversa. Il programma scriverà nel file di output i costi totali che dovranno pagare ciascun utente. È presento lo sconto per 10 o più ore di prenotazione e prenotazioni notturne.
- **TC26:** testiamo il caso in cui più utenti prenotano più macchine per fascia giorno/ora diversa. Il programma scriverà nel file di output i costi totali che dovranno pagare ciascun utente. È presente lo sconto per 10 o più ore di prenotazione, prenotazioni weekend e prenotazioni notturne.
- **TC27:** testiamo il caso in cui proviamo a fare lo sconto del weekend con più utenti e più macchine. Il programma scriverà nel file di output i costi totali che dovranno pagare ciascun utente.
- **TC28:** testiamo il caso in cui proviamo a fare lo sconto di 10 o più ore di prenotazione con più utenti e più macchine. Il programma scriverà nel file di output i costi totali che dovranno pagare ciascun utente.

- **TC29:** testiamo il caso in cui proviamo a fare lo sconto notturno con più utenti e più macchine. Il programma scriverà nel file di output i costi totali che dovranno pagare ciascun utente.

Casi di test STORICO

- **TC30:** testiamo il caso in cui abbiamo più utenti con la stessa macchina e non hanno prenotato tutti gli utenti. Il programma scriverà nel file di output lo storico di tutti gli utenti.
- **TC31:** testiamo il caso in cui abbiamo più utenti con macchine diverse e tutti hanno almeno una prenotazione. Il programma scriverà nel file di output lo storico di tutti gli utenti.

Casi di test Codice fiscale

In questo caso non ci interessa che tipo di test fa, dobbiamo solo verificare che i controlli del codice fiscale e della ricerca dell'utente funzionino.

- **TC32:** testiamo il caso in cui abbiamo varie prenotazioni in cui il codice fiscale non rispetta l'algoritmo. Il programma scriverà nel file di output che l'utente non è stato trovato.

Casi di test Targa

In questo caso non ci interessa che tipo di test fa, dobbiamo solo verificare che controlli della targa e della ricerca dell'auto funzionino.

- **TC33:** testiamo il caso in cui abbiamo varie prenotazioni in cui le targhe non rispettano l'algoritmo. Il programma scriverà nel file di output che l'auto non è stata trovata e che la prenotazione non è riuscita.