Introduzione al Progetto

Il sistema di **Car Sharing** è stato sviluppato da **Nello Manuel Russo**, **Antonio Sicignano**, **Marco Visone**. L'obiettivo del progetto è offrire una piattaforma efficiente ed efficace che risponda ai seguenti requisiti fondamentali.

1 Funzionalità del Sistema

1.1 Gestione Prenotazioni

Il sistema permette agli utenti di prenotare un veicolo in base alla disponibilità in un intervallo di tempo specificato, garantendo un processo semplice e intuitivo.

1.2 Visualizzazione della Disponibilità

La piattaforma mostra in tempo reale i veicoli disponibili, evidenziando le seguenti informazioni essenziali:

- Targa del veicolo
- Modello del veicolo
- Tipologia del veicolo (auto, moto, furgone)
- Posizione geografica
- Tempo residuo di disponibilità
- Costo totale per la prenotazione

1.3 Calcolo del Costo

Il calcolo del costo avviene attraverso un algoritmo che considera molteplici fattori:

- **Tariffa base**: calcolata in base alla durata del noleggio (tariffa del veicolo al minuto moltiplicata per i minuti di utilizzo)
- **Sconti orari**: applicazione di riduzioni per fasce orarie specifiche:
  - 00:00-06:00 (notturna)
  - 06:00-08:00 (mattutina)
- **Promozioni**: inclusione di eventuali sconti per utenti abituali

1.4 Gestione dei Noleggi

Il sistema mantiene traccia di tutte le prenotazioni effettuate, fornendo le seguenti informazioni:

- Targa del veicolo
- Modello del veicolo
- Intervallo temporale della prenotazione
- Email dell'utente che ha effettuato la prenotazione
- Costo totale della prenotazione

1.5 Storico dei Noleggi

Ogni utente ha accesso a un riepilogo dettagliato delle proprie prenotazioni che include:

- Intervallo temporale della prenotazione
- Costo totale sostenuto
- Targa del veicolo utilizzato
- Modello del veicolo prenotato

2 Interfaccia

2.1 Interfaccia Utente senza accesso

2.1.1 Autenticazione

Sistema di login e registrazione per l'accesso alla piattaforma.

2.2 Interfaccia Cliente

2.2.1 Visualizzazione Disponibilità

Lista dei veicoli disponibili con relativi dettagli tecnici e commerciali.

2.2.2 Gestione Prenotazioni

- Creazione di nuove prenotazioni specificando l'intervallo temporale
- Visualizzazione delle prenotazioni attive con possibilità di cancellazione
- Accesso completo allo storico delle prenotazioni effettuate

2.3 Interfaccia Amministratore

L'interfaccia di amministrazione comprende funzionalità avanzate per la gestione del sistema:

2.3.1 Gestione Veicoli

- Aggiunta di nuovi veicoli alla tabella
- Rimozione di veicoli

2.3.2 Controllo Noleggi

- Consultazione dello storico dettagliato di ciascun veicolo
- Analisi dello storico di utilizzo di ciascun utente

3 Strutture Dati e Architettura

3.1 Gestione degli Utenti

3.1.1 ADT Utente

Per garantire un'organizzazione ordinata e strutturata dei dati. La struttura dell'ADT Utente comprende i seguenti campi:

- **Nome**: nome dell'utente
- **Cognome**: cognome dell'utente
- **Email**: indirizzo email utilizzato come identificativo univoco
- **Password**: password dell'utente memorizzata in formato hash md5 per garantire la sicurezza
- **Permessi**: livello di accesso dell'utente rappresentato da un valore binario:

- `0` = Amministratore
- `1` = Cliente

3.1.2 Struttura Dati Cliente

3.1.3 ADT Data

Per gli utenti con permessi di cliente, l'ADT Utente contiene un **ADT Data** specifico che gestisce le informazioni relative alle prenotazioni. L'ADT Data è composto da:

- **Lista prenotazioni**: storico completo delle prenotazioni effettuate dall'utente
- **Numero prenotazioni**: contatore del totale delle prenotazioni effettuate

3.1.4 Implementazione della Lista Prenotazioni

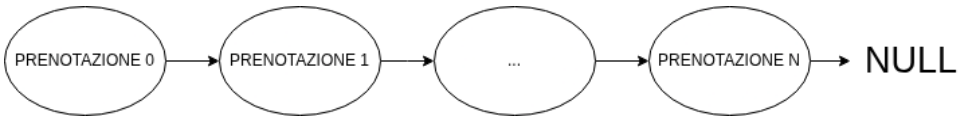

Figura 1: Rappresentazione della Lista generica.

Vedi anche (ADT Prenotazione) per ulteriori dettagli.

La lista delle prenotazioni utilizza un **ADT Lista** generico che fornisce un'interfaccia che userà ADT lista di prenotazioni **ADT ListaPre** per la gestione delle prenotazioni. La scelta della struttura dati lista è motivata da:

- **Velocità di aggiunta**: inserimento efficiente di nuove prenotazioni
- **Scalabilità**: capacità di gestire un numero variabile e crescente di prenotazioni nel tempo
- **Flessibilità**: adattabilità al numero incerto di prenotazioni per ciascun utente
- **Risparmio di Memoria**: ogni utente ha il proprio numero di prenotazioni nello storico

Complessità computazionale delle operazioni:

- **Inserimento**: O(1) - aggiunta in coda alla lista
- **Ricerca**: O(n) caso peggiore - accesso tramite confronto di intervallo temporale
- **Cancellazione**: O(n) caso peggiore - ricerca dell'elemento tramite confronto di intervallo e successiva rimozione

3.1.5 Memorizzazione degli Utenti

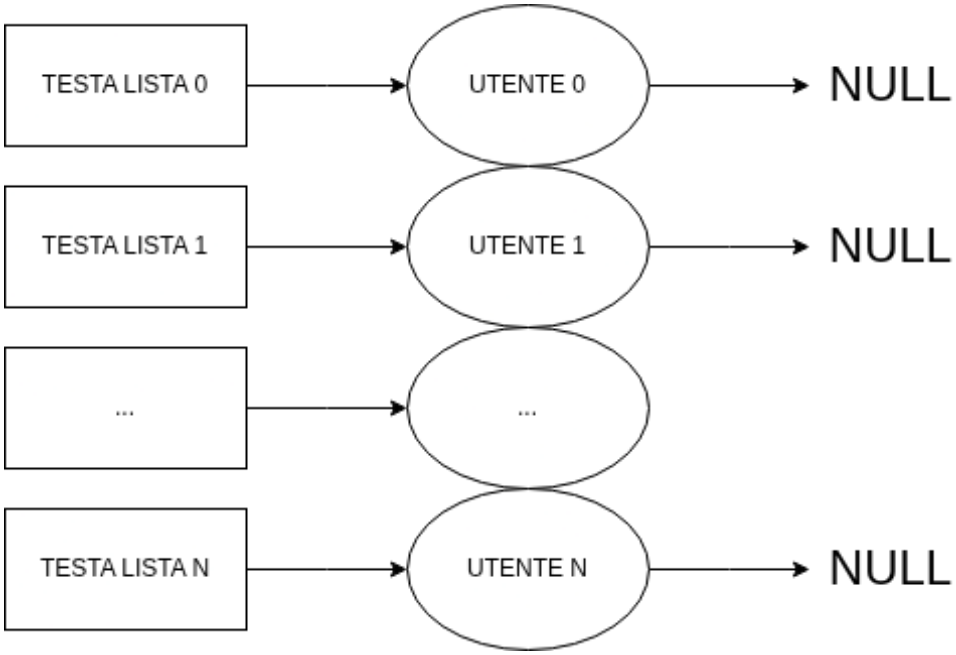

Figura 2: Rappresentazione della TabellaHash Utenti.

Gli utenti del sistema sono memorizzati nell'**ADT TabellaHash** che fornisce un'interfaccia generica utilizzata dall'**ADT TabellaHash Utenti**. La tabella hash utenti, in base alla chiave **email**, restituisce l'utente corrispondente.

3.1.6 Motivazione della Scelta

La scelta dell'**ADT TabellaHash** per la memorizzazione degli utenti è stata guidata dalle specifiche esigenze prestazionali del sistema di Car Sharing:

- **Velocità di accesso ai dati**: un sistema di Car Sharing effettua numerosi accessi ai dati degli utenti per operazioni di autenticazione, prenotazione e consultazione storico. La tabella hash garantisce accesso O(1) in caso medio, ottimale per questo tipo di applicazione ad alta frequenza di accesso
- **Efficienza nelle ricerche**: ogni login, ogni prenotazione e ogni consultazione dello storico richiede la ricerca rapida dell'utente tramite email
- **Iterazione efficiente**: quando necessario iterare su tutti gli utenti (ad esempio per operazioni amministrative), la struttura interna basata su vettore permette un'iterazione rapida e sequenziale di tutti gli elementi memorizzati
- **Scalabilità**: il ridimensionamento dinamico della tabella permette di mantenere prestazioni ottimali anche con l'aumento del numero di utenti registrati

3.1.7 Algoritmo Hash e Gestione Collisioni

La tabella hash implementa le seguenti caratteristiche tecniche:

- **Algoritmo hash**: utilizza l'algoritmo **djb2** per il calcolo del valore hash dalla chiave email
- **Gestione collisioni**: risoluzione mediante **liste concatenate** utilizzando l'ADT List precedentemente descritto
- **Ridimensionamento dinamico**: quando il numero deibucket raggiunge il 75% della dimensione massima, la tabella si ridimensiona.
- **Strategia di crescita**: al raggiungimento della soglia, la tabella raddoppia la grandezza precedente per diminuire il più possibile il numero di collisioni

Complessità computazionale delle operazioni sulla TabellaHash:

- **Inserimento**: O(1) caso medio, O(n) caso peggiore - dipende dal numero di collisioni
- **Ricerca**: O(1) caso medio, O(n) caso peggiore - ricerca tramite chiave email
- **Cancellazione**: O(1) caso medio, O(n) caso peggiore - localizzazione tramite email e rimozione
- **Ridimensionamento**: O(n) - riorganizzazione completa di tutti gli elementi nella nuova tabella
- **Iterazione completa**: O(n) - scorrimento sequenziale del vettore interno per accedere a tutti gli utenti

3.2 Gestione dei Veicoli

3.2.1 ADT Veicolo

I veicoli del sistema sono rappresentati tramite un ADT Veicolo per garantire un'organizzazione ordinata e strutturata dei dati. La struttura dell'ADT Veicolo comprende i seguenti campi:

- **Targa**: identificativo univoco del veicolo
- **Modello**: modello specifico del veicolo
- **Posizione**: localizzazione geografica corrente del veicolo

- **Tipo**: categoria del veicolo (auto, moto, furgone)
- **Tariffa al minuto**: costo per minuto di utilizzo del veicolo
- **Prenotazioni**: struttura dati che gestisce gli intervalli temporali delle prenotazioni

3.2.2 Gestione delle Prenotazioni dei Veicoli

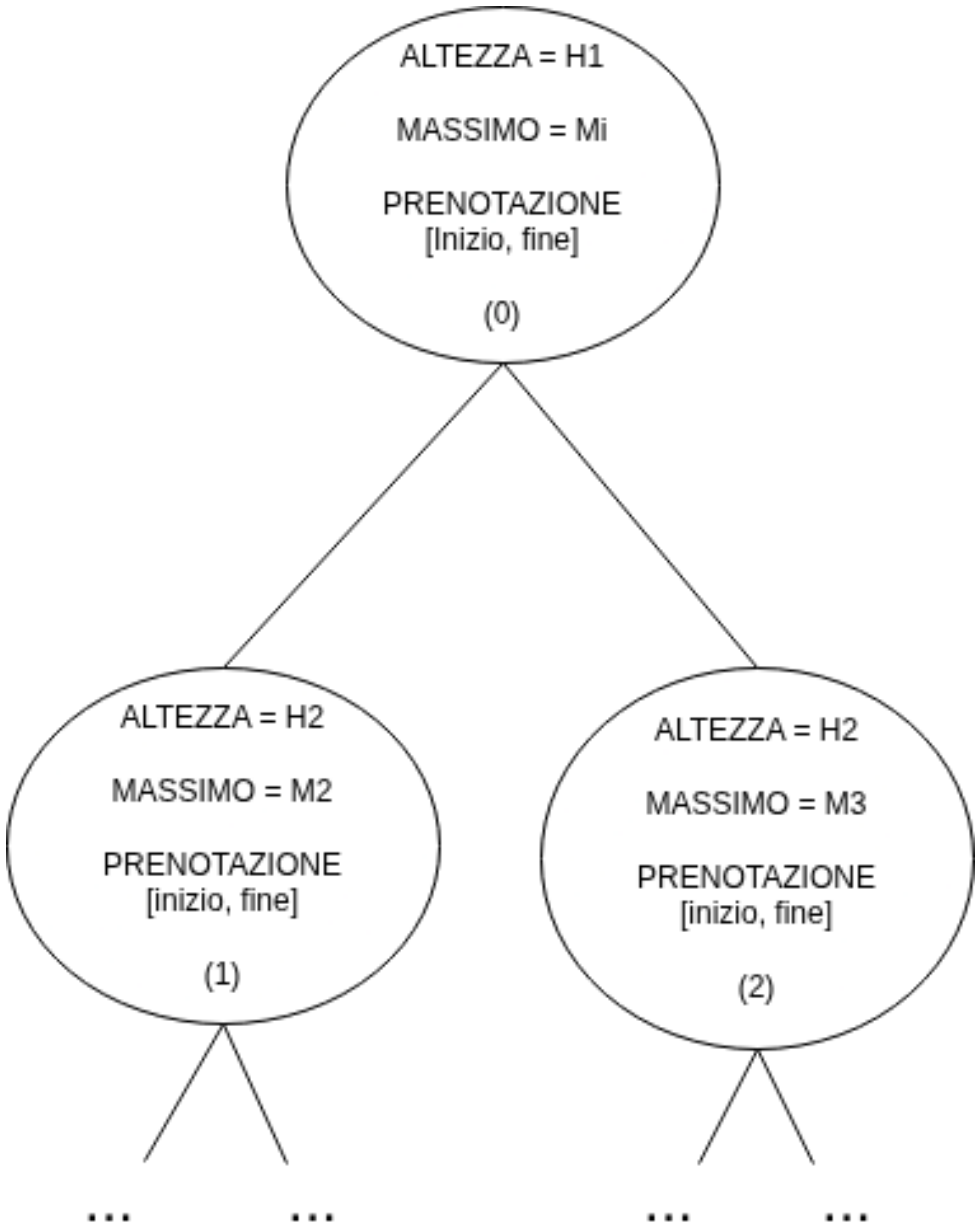

Figura 3: Rappresentazione dell'Albero AVL degli Intervalli.

Per gestire le prenotazioni di ciascun veicolo, è stata implementata una variante dell'**Interval Tree** che utilizza un **AVL Tree** come struttura sottostante. Questa scelta consente di ottimizzare il rilevamento di sovrapposizioni temporali tra le prenotazioni, garantendo al contempo un'elevata efficienza anche in presenza di grandi volumi di dati.

3.2.3 Caratteristiche e Funzionamento

L'implementazione presenta le seguenti caratteristiche e il relativo funzionamento algoritmico:

- **Bilanciamento AVL**: L'albero mantiene automaticamente il bilanciamento tramite rotazioni, assicurando un'altezza logaritmica. Questo garantisce prestazioni efficienti ($O(\log n)$) per tutte le operazioni, evitando degradi anche in presenza di numerose prenotazioni.
- **Campo 'massimo'**: Ogni nodo memorizza il valore massimo del campo 'fine' tra tutti gli intervalli presenti nel proprio sottoalbero (incluso se stesso). Questo valore è fondamentale per escludere rapidamente intere sezioni dell'albero che non possono contenere sovrapposizioni, ottimizzando la ricerca.
- **Rilevamento Sovrapposizioni**: Durante l'inserimento di una nuova prenotazione o nella ricerca di conflitti, il campo 'massimo' viene utilizzato per muoversi in modo efficiente ignorando rami non rilevanti, riducendo il numero di confronti necessari.
- **Gestione Dinamica e Manutenzione**: L'albero supporta inserimenti e cancellazioni di prenotazioni. Dopo ogni operazione, vengono eseguite le rotazioni AVL necessarie e aggiornati i campi 'massimo' dei nodi coinvolti, preservando sia la correttezza logica che le proprietà strutturali.

3.2.4 Motivazione della Scelta e Complessità

La scelta di un AVL Interval Tree è motivata dai requisiti specifici del sistema di Car Sharing:

- **Controllo Efficiente delle Sovrapposizioni**: Fondamentale per evitare prenotazioni conflittuali dello stesso veicolo, anche su orizzonti temporali molto ravvicinati.
- **Prestazioni e Scalabilità**: Il bilanciamento AVL garantisce tempi logaritmici anche con un elevato numero di intervalli, rendendo il sistema scalabile e reattivo a fronte di carichi dinamici e frequenti modifiche.

Le complessità computazionali delle principali operazioni sono:

- **Inserimento**: $O(\log n)$
- **Ricerca di Sovrapposizioni**: $O(\log n)$ nel caso medio, $O(n)$ nel peggiore (in presenza di molte sovrapposizioni)
- **Cancellazione**: $O(\log n)$

- **Aggiornamento del Campo 'massimo'**: $O(\log n)$
- **Rotazioni AVL**: Ogni singola rotazione ha costo $O(1)$, ma il numero totale di rotazioni in un'operazione può essere $O(\log n)$

3.2.5 Memorizzazione dei Veicoli

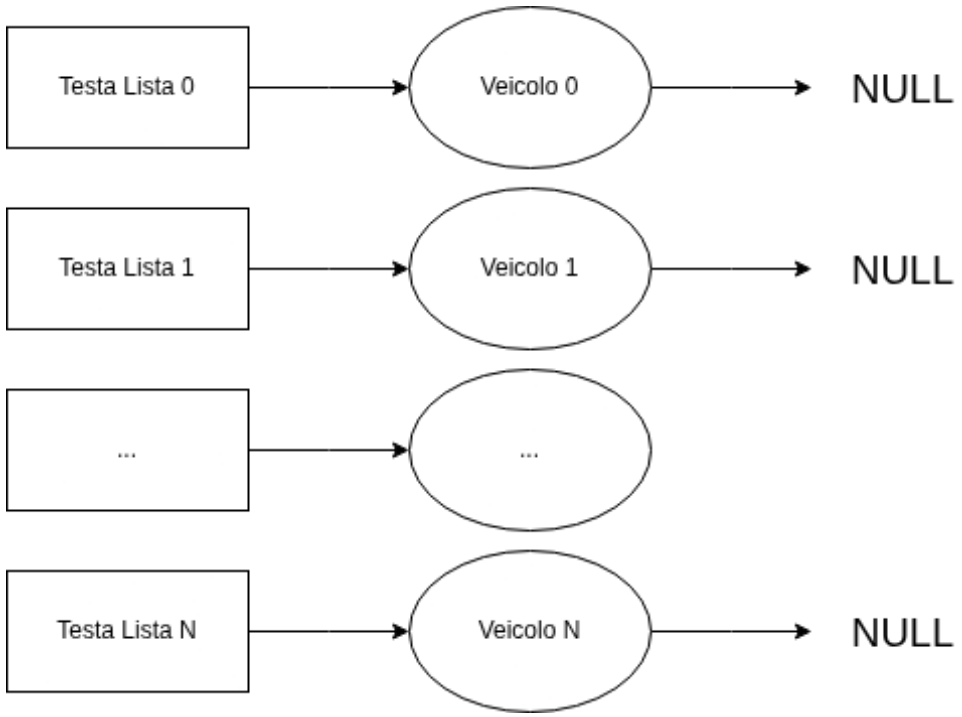

Figura 4: Rappresentazione della TabellaHash Veicoli.

I veicoli del sistema sono memorizzati nell'**ADT TabellaHash** che fornisce un'interfaccia generica utilizzata dall'**ADT TabellaHash Veicoli**. La tabella hash veicoli, in base alla chiave **targa**, restituisce il veicolo corrispondente.

3.2.6 Motivazione della Scelta

La scelta dell'**ADT TabellaHash** per la memorizzazione dei veicoli è stata guidata dalle specifiche esigenze prestazionali del sistema di Car Sharing:

- **Velocità di accesso ai dati**: un sistema di Car Sharing effettua numerosi accessi ai dati dei veicoli per operazioni di ricerca disponibilità, visualizzazione dettagli e gestione prenotazioni. La tabella hash garantisce accesso O(1) in caso medio, ottimale per questo tipo di applicazione ad alta frequenza di accesso
- **Efficienza nelle ricerche**: ogni ricerca di disponibilità, ogni prenotazione e ogni consultazione dei dettagli del veicolo richiede la ricerca rapida del veicolo tramite targa

- **Iterazione efficiente**: quando necessario iterare su tutti i veicoli (ad esempio per visualizzare la disponibilità generale o per operazioni amministrative), la struttura interna basata su vettore permette un'iterazione rapida e sequenziale di tutti gli elementi memorizzati
- **Scalabilità**: il ridimensionamento dinamico della tabella permette di mantenere prestazioni ottimali anche con l'aumento dei veicoli

3.2.7 Algoritmo Hash e Gestione Collisioni

La tabella hash dei veicoli implementa le seguenti caratteristiche tecniche:

- **Algoritmo hash**: utilizza l'algoritmo **djb2** per il calcolo del valore hash dalla chiave targa
- **Gestione collisioni**: risoluzione mediante **liste concatenate** utilizzando l'ADT List precedentemente descritto
- **Ridimensionamento dinamico**: quando il numero di bucket raggiunge il 75% della dimensione massima, la tabella si ridimensiona
- **Strategia di crescita**: al raggiungimento della soglia, la tabella raddoppia la grandezza precedente per diminuire il più possibile il numero di collisioni

Complessità computazionale delle operazioni sulla TabellaHash Veicoli:

- **Inserimento**: O(1) caso medio, O(n) caso peggiore - dipende dal numero di collisioni
- **Ricerca**: O(1) caso medio, O(n) caso peggiore - ricerca tramite chiave targa
- **Cancellazione**: O(1) caso medio, O(n) caso peggiore - localizzazione tramite targa e rimozione
- **Ridimensionamento**: O(n) - riorganizzazione completa di tutti gli elementi nella nuova tabella
- **Iterazione completa**: O(n) - scorrimento sequenziale del vettore interno per accedere a tutti i veicoli

3.3 ADT Intervallo

L'ADT **Intervallo** è stato introdotto per rappresentare un intervallo temporale mediante due valori numerici interi: `inizio` e `fine`. Questo ADT permette una rappresentazione compatta e coerente di un periodo.

Campi

- `inizio`: intero che rappresenta l'inizio dell'intervallo
- `fine`: intero che rappresenta la fine dell'intervallo

3.4 ADT Prenotazione

L'ADT **Prenotazione** è stato introdotto, analogamente agli ADT *Utente* e *Veicolo*, per modellare una prenotazione effettuata da un utente per un veicolo, includendo il periodo della prenotazione e il relativo costo.

Campi

- `costo`: valore numerico che rappresenta il costo della prenotazione
- `targaVeicolo`: stringa contenente la targa del veicolo prenotato
- `emailUtente`: stringa contenente l'email dell'utente che ha effettuato la prenotazione
- `intervallo`: istanza dell'ADT *Intervallo* che rappresenta il periodo della prenotazione

4 Implementazione delle Funzionalità con ADT

4.1 Gestione Prenotazioni

il sistema permette agli utenti di prenotare un veicolo in base alla disponibilità in un intervallo di tempo specificato, garantendo un processo semplice e intuitivo

—

4.1.1 ADT Coinvolti

- `TabellaHash Veicoli`: L'ADT primario che gestisce la collezione di tutti i veicoli.
- `AVL Interval Tree`: Ogni veicolo possiede un suo `AVL Interval Tree` per gestire le prenotazioni.
- `TabellaHash Utenti`: Utilizzata per gestire i dati degli utenti e le loro prenotazioni.

- `Intervallo`: Una struttura dati che definisce un periodo di tempo con un inizio e una fine.
- `Prenotazione`: L'oggetto che incapsula i dettagli di una specifica prenotazione.

—

4.1.2 Processo

Il flusso di gestione di una prenotazione si articola in diverse fasi sequenziali:

1. **Ricezione dell'Intervallo Richiesto**: Il sistema acquisisce l'intervallo di tempo per il quale l'utente intende effettuare la prenotazione.
2. **Preparazione dell'Elenco Veicoli Disponibili**:
   - Tutti i veicoli vengono estratti dalla `TabellaHash Veicoli` e temporaneamente copiati in un vettore.
   - Per ogni veicolo in questo vettore, si verifica la compatibilità con l'intervallo richiesto. Questa verifica implica un'interrogazione dell'`AVL Interval Tree` del veicolo per rilevare eventuali sovrapposizioni con prenotazioni esistenti.
   - Se un veicolo risulta non disponibile per l'intervallo richiesto (a causa di una sovrapposizione), il suo riferimento viene rimosso dall'elenco temporaneo, lasciando solo i veicoli realmente disponibili.
3. **Creazione e Registrazione della Nuova Prenotazione**:
   - Una volta che l'utente ha selezionato un veicolo tra quelli disponibili, viene creato un nuovo oggetto `Prenotazione`. Questo include la targa del veicolo, l'email dell'utente, l'intervallo di tempo prenotato e il costo calcolato.
   - La nuova `Prenotazione` viene inserita nell'`AVL Interval Tree` del veicolo scelto, aggiornandone la disponibilità futura.
   - Infine, la prenotazione viene associata all'utente corrispondente nella `TabellaHash Utenti`, aggiungendola anche a una lista delle prenotazioni effettuate da quell'utente.

—

4.1.3 Efficienza Complessiva

L'efficienza del processo è determinata dalla combinazione delle operazioni svolte sui vari ADT. Consideriamo:

- N: il numero totale di veicoli presenti nel sistema.
- M: il numero massimo di prenotazioni per un singolo veicolo (che influenza l'altezza dell'`AVL Tree`).

1. **Raccolta dei veicoli dalla** `TabellaHash`: L'estrazione e copia di tutti i veicoli in un vettore richiede un tempo proporzionale al numero di elementi e bucket della tabella. Per l'implementazione fornita, questo costo è $\mathcal{O}(N)$.
2. **Filtraggio dei veicoli disponibili**: Per ogni veicolo (N veicoli), si esegue una verifica di sovrapposizione sul rispettivo `AVL Interval Tree`. Questa operazione ha un costo di $\mathcal{O}(\log M)$. Pertanto, questa fase contribuisce con $\mathcal{O}(N \cdot \log M)$.
3. **Creazione e inserimento della** `Prenotazione`: La creazione della prenotazione è $\mathcal{O}(1)$. L'inserimento della nuova prenotazione nell'`AVL Interval Tree` del veicolo selezionato è $\mathcal{O}(\log M)$. L'aggiornamento della lista delle prenotazioni dell'utente (dopo averlo trovato nella `TabellaHash Utenti`) è $\mathcal{O}(1)$.

Complessivamente, l'efficienza è dominata dalla fase di filtraggio dei veicoli:

$$\mathcal{O}(N + N \cdot \log M + \log M + 1)$$

Questo si semplifica a:

$$\mathcal{O}(N \cdot \log M)$$

Questa complessità indica che il tempo per gestire una prenotazione cresce linearmente con il numero totale di veicoli, ma solo logaritmicamente con il numero di prenotazioni per singolo veicolo, garantendo una buona scalabilità del sistema.

4.2 Visualizzazione Disponibilità

- **Obiettivo**: Questo processo ha il compito di presentare all'utente lo stato attuale della disponibilità di tutti i veicoli, calcolando per ciascuno il prossimo intervallo libero e i costi associati.
- **ADT Coinvolti**:
  - `TabellaHash Veicoli`: L'ADT che gestisce l'insieme di tutti i veicoli disponibili.
  - `AVL Interval Tree`: Per ogni veicolo, questo ADT gestisce le sue prenotazioni, permettendo di trovare efficientemente intervalli di tempo liberi.

- `Intervallo`: La struttura dati che rappresenta un periodo di tempo, utilizzata per definire le prenotazioni e gli spazi di disponibilità.

- **Processo**: Il processo di visualizzazione della disponibilità si svolge in due fasi principali:

  1. **Raccolta di tutti i Veicoli**: Il sistema recupera tutti i veicoli attualmente presenti nella `TabellaHash Veicoli` e li organizza in un elenco. Questa operazione garantisce che ogni veicolo registrato venga considerato per la disponibilità.
  2. **Analisi della Disponibilità per Ciascun Veicolo**: Per ogni veicolo raccolto, vengono eseguite le seguenti operazioni:
     - **Determinazione del prossimo intervallo libero**: Viene interrogato l' `AVL Interval Tree` associato al veicolo per identificare il primo periodo di tempo in cui il veicolo non è prenotato.
     - **Estrazione delle informazioni chiave**: Si raccolgono dati essenziali del veicolo come targa, modello, tipo e posizione.
     - **Calcolo del tempo residuo di disponibilità**: Viene calcolato quanto tempo rimane prima della prossima prenotazione.
     - **Stima del costo per l'intervallo corrente**: Si calcola il costo potenziale per un noleggio basato sull'intervallo di disponibilità attuale.
  3. **Aggregazione dei Risultati**: I dati elaborati per ciascun veicolo (intervallo disponibile, dettagli del veicolo, tempo residuo, costo) vengono raccolti per essere visualizzati all'utente.

- **Efficienza Complessiva**: L'efficienza di questo processo è determinata dalla combinazione delle operazioni svolte:

  - La fase di raccolta di tutti i veicoli dalla `TabellaHash` richiede un tempo proporzionale al numero totale di veicoli.
  - Per ogni veicolo, la ricerca del prossimo intervallo libero nell' `AVL Interval Tree` è un'operazione efficiente, che cresce logaritmicamente con il numero di prenotazioni di quel veicolo.

  Considerando n come il numero totale di veicoli e m come il numero massimo di prenotazioni per un singolo veicolo, l'efficienza complessiva è:

  $$\mathcal{O}(n \cdot \log m)$$

  Questo approccio garantisce che la visualizzazione della disponibilità sia scalabile ed efficiente, anche con un elevato numero di veicoli e prenotazioni.

4.3 Calcolo del Costo

- **ADT Coinvolti**: Prenotazione, Veicolo, Utente, Intervallo
- **Algoritmo**:
  1. Calcolo durata: durata $=$ intervallo.fine $-$ intervallo.inizio
  2. Tariffa base: base $=$ durata $\times$ veicolo.tariffa_minuto
  3. Applicazione sconti orari (sull'ADT Intervallo):
     - `00:00-06:00`: -20%
     - `06:00-08:00`: -15%
  4. Applicazione bonus fedeltà (dall'ADT Utente):

$$\text{sconto\_fedeltà} = \begin{cases} 25\% & \text{se } \texttt{numero\_prenotazioni} \bmod 5 = 0 \\ 0\% & \text{altrimenti} \end{cases}$$

  5. Costo finale: base $\times$ $(1 -$ sconti_orari$) \times (1 -$ sconto_fedeltà$)$
  6. Memorizzazione costo nell'ADT Prenotazione
- **Esempio**:
  - Utente con 7 prenotazioni: $\lfloor 7/5 \rfloor = 1 \rightarrow$ 25% di sconto
  - Utente con 12 prenotazioni: $\lfloor 12/5 \rfloor = 2 \rightarrow$ 25% di sconto
  - Utente con 3 prenotazioni: $\lfloor 3/5 \rfloor = 0 \rightarrow$ nessuno sconto

4.4 Gestione dei Noleggi

- **ADT Coinvolti**: TabellaHash Utenti, TabellaHash Veicoli, Prenotazione, Lista
- **Processo**:
  1. Ricerca utente per email (TabellaHash Utenti $\rightarrow$ O(1))
  2. Accesso alla lista prenotazioni (ADT Lista nell'ADT Utente)
  3. Iterazione lista per ottenere tutte le prenotazioni attive (O(n), n il numero di nodi della lista)
  4. Per ogni prenotazione (ADT Prenotazione):
     - Recupero veicolo tramite targa (TabellaHash Veicoli $\rightarrow$ O(1))
     - Estrazione dati: modello, intervallo, costo
  5. Aggregazione dati per visualizzazione

4.5 Storico Noleggi

- **ADT Coinvolti**: Utente, Lista, Prenotazione
- **Implementazione**:
  1. Accesso all'ADT Data dell'utente autenticato
  2. Iterazione della lista prenotazioni (ADT Lista)
  3. Per ogni elemento della lista (ADT Prenotazione):
     - Estrazione: intervallo, costo, targa, modello
     - Ordinamento cronologico inverso (data inizio intervallo)
  4. Calcolo statistiche utilizzo (tempo totale, costo medio, ecc.)
- **Ottimizzazione**:
  - Inserimento O(1) mantenendo puntatore alla coda
  - Ordinamento O(n log n) solo su richiesta (lazy sorting)

Specifica Sintattica e Semantica

Questa sezione elenca e descrive le operazioni che possono essere eseguite sugli ADT (Abstract Data Type) scelti, includendo pre-condizioni, post-condizioni, i loro input, output ed effetti collaterali.

—

4.5.1 ADT Data

L'ADT `Data` gestisce una collezione di prenotazioni, tipicamente associata a uno storico.

- **crea_data()**
  - *Descrizione:* Crea e inizializza una nuova struttura `Data`.
  - *Input:* Nessuno.
  - *Output:* Un nuovo oggetto `Data` allocato dinamicamente. `NULL` in caso di errore di allocazione.
  - *Pre-condizioni:* Nessuna.
  - *Post-condizioni:* Viene restituito un puntatore a una nuova struttura `Data` inizializzata.

- *Effetti Collaterali:* Allocazione dinamica di memoria.

- **distruggi_data(Data data)**
  - *Descrizione:* Dealloca la memoria associata a una struttura `Data`, liberando anche le eventuali liste interne di prenotazioni.
  - *Input:* `data`: puntatore alla struttura `Data` da distruggere.
  - *Output:* Nessuno.
  - *Pre-condizioni:* `data` non deve essere `NULL`.
  - *Post-condizioni:* La memoria occupata dalla struttura `Data` e dalle sue liste interne è liberata.
  - *Effetti Collaterali:* Deallocazione dinamica di memoria.

- **ottieni_storico_lista(Data data)**
  - *Descrizione:* Restituisce la lista delle prenotazioni associate alla `Data`.
  - *Input:* `data`: puntatore alla struttura `Data`.
  - *Output:* La lista delle prenotazioni (`ListaPre`).
  - *Pre-condizioni:* `data` non deve essere `NULL`.
  - *Post-condizioni:* Nessuna.
  - *Effetti Collaterali:* Nessuno.

- **aggiungi_a_storico_lista(Data data, Prenotazione prenotazione)**
  - *Descrizione:* Aggiunge una prenotazione alla lista interna della `Data`.
  - *Input:*
    - `data`: puntatore alla struttura `Data`.
    - `prenotazione`: prenotazione da aggiungere.
  - *Output:* La lista aggiornata con la prenotazione aggiunta (`Byte`).
  - *Pre-condizioni:* `data` e `prenotazione` non devono essere `NULL`.
  - *Post-condizioni:* La lista interna della `Data` è aggiornata con la nuova prenotazione.
  - *Effetti Collaterali:* Modifica la lista interna della `Data`.

- **rimuovi_da_storico_lista(Data data, Prenotazione prenotazione)**
  - *Descrizione:* Rimuove una prenotazione dalla lista interna della `Data`.
  - *Input:*
    - `data`: puntatore alla struttura `Data`.

    - `prenotazione`: prenotazione da rimuovere.
  - *Output:* La lista aggiornata dopo la rimozione (`Byte`).
  - *Pre-condizioni:* `data` e `prenotazione` non devono essere `NULL`.
  - *Post-condizioni:* La lista interna della `Data` è aggiornata dopo la rimozione.
  - *Effetti Collaterali:* Modifica la lista interna della `Data`.
- **ottieni_vettore_storico(Data data, unsigned int *dimensione)**
  - *Descrizione:* Restituisce un array contenente tutte le prenotazioni presenti nello storico.
  - *Input:*
    - `data`: struttura contenente lo storico delle prenotazioni.
    - `dimensione`: puntatore a intero dove verrà memorizzata la dimensione del vettore.
  - *Output:* Vettore di prenotazioni (`Prenotazione *`) o `NULL` in caso di errore.
  - *Pre-condizioni:* `data` deve essere una struttura valida.
  - *Post-condizioni:* Viene restituito un vettore allocato dinamicamente contenente tutte le prenotazioni, oppure `NULL` se `data` è `NULL` o se l'allocazione fallisce.
  - *Effetti Collaterali:* Allocazione dinamica di memoria per il vettore risultante.
- **ottieni_numero_prenotazioni(Data data)**
  - *Descrizione:* Restituisce il numero di prenotazioni contenute nella struttura `Data`.
  - *Input:* `data`: struttura `Data` da cui ottenere il numero di prenotazioni.
  - *Output:* Numero di prenotazioni (`unsigned int`) o -1 se `data` è `NULL`.
  - *Pre-condizioni:* `data` deve essere una struttura valida.
  - *Post-condizioni:* Nessuna.
  - *Effetti Collaterali:* Nessuno.
- **imposta_numero_prenotazioni(Data data, int numero_prenotazioni)**
  - *Descrizione:* Imposta il numero di prenotazioni nella struttura `Data`.
  - *Input:*
    - `data`: struttura `Data` su cui effettuare la modifica.
    - `numero_prenotazioni`: nuovo valore da assegnare.
  - *Output:* Nessun valore.
  - *Pre-condizioni:* `data` deve essere una struttura valida.

- *Post-condizioni:* Il campo `numero_prenotazioni` della struttura viene aggiornato.
- *Effetti Collaterali:* Modifica della struttura `Data`.

- **imposta_storico_lista(Data data, ListaPre lista_prenotazione)**
  - *Descrizione:* Imposta la lista delle prenotazioni storico nella struttura `Data`.
  - *Input:*
    - `data`: struttura `Data` da aggiornare.
    - `lista_prenotazione`: nuova lista storico da assegnare.
  - *Output:* Nessun valore.
  - *Pre-condizioni:* `data` deve essere una struttura valida.
  - *Post-condizioni:* Il campo `storico` della struttura viene aggiornato.
  - *Effetti Collaterali:* Modifica della struttura `Data`.

—

4.5.2 ADT Intervallo

L'ADT `Intervallo` rappresenta un periodo di tempo definito da un'ora di inizio e un'ora di fine.

- **crea_intervallo(time_t inizio, time_t fine)**
  - *Descrizione:* Crea un nuovo intervallo temporale con tempo di inizio e fine specificati.
  - *Input:*
    - `inizio`: timestamp di inizio dell'intervallo.
    - `fine`: timestamp di fine dell'intervallo.
  - *Output:* Un nuovo `Intervallo` allocato dinamicamente, oppure `NULL` in caso di errore o parametri non validi.
  - *Pre-condizioni:*
    - `inizio` deve essere un timestamp valido.
    - `fine` deve essere un timestamp valido.
    - `inizio` deve essere $\leq$ `fine`.
  - *Post-condizioni:* Viene restituito un puntatore a un nuovo intervallo con i tempi specificati.

  - *Effetti Collaterali:* Alloca memoria per la struttura `Intervallo`.

- **distruggi_intervallo(Intervallo i)**
  - *Descrizione:* Libera la memoria allocata per l'intervallo.
  - *Input:* `i`: puntatore all'intervallo da distruggere.
  - *Output:* Nessuno.
  - *Pre-condizioni:* `i` deve essere un puntatore valido (può essere `NULL`).
  - *Post-condizioni:* La memoria dell'intervallo viene deallocata se `i` non è `NULL`.
  - *Effetti Collaterali:* Dealloca memoria.

- **inizio_intervallo(Intervallo i)**
  - *Descrizione:* Restituisce il tempo di inizio dell'intervallo specificato.
  - *Input:* `i`: puntatore all'intervallo.
  - *Output:* Il timestamp di inizio (`time_t`) o 0 se l'intervallo è `NULL`.
  - *Pre-condizioni:* `i` deve essere un puntatore valido (può essere `NULL`).
  - *Post-condizioni:* Restituisce il timestamp di inizio.
  - *Effetti Collaterali:* Nessuno.

- **fine_intervallo(Intervallo i)**
  - *Descrizione:* Restituisce il tempo di fine dell'intervallo specificato.
  - *Input:* `i`: puntatore all'intervallo.
  - *Output:* Il timestamp di fine (`time_t`) o 0 se l'intervallo è `NULL`.
  - *Pre-condizioni:* `i` deve essere un puntatore valido (può essere `NULL`).
  - *Post-condizioni:* Restituisce il timestamp di fine.
  - *Effetti Collaterali:* Nessuno.

- **intervalli_si_sovrappongono(Intervallo interno, Intervallo esterno)**
  - *Descrizione:* Controlla se due intervalli temporali si sovrappongono.
  - *Input:*
    - `interno`: primo intervallo da controllare.
    - `esterno`: secondo intervallo da controllare.
  - *Output:* 1 (`true`) se gli intervalli si sovrappongono, 0 (`false`) altrimenti o in caso di parametri non validi (`Byte`).
  - *Pre-condizioni:*

    - `interno` non deve essere `NULL`.
    - `esterno` non deve essere `NULL`.
  - *Post-condizioni:* Restituisce un valore booleano indicante la sovrapposizione.
  - *Effetti Collaterali:* Nessuno.

- **converti_data_in_intervallo(const char *inizio, const char *fine)**
  - *Descrizione:* Converte due stringhe di data in un intervallo temporale.
  - *Input:*
    - `inizio`: stringa con data/ora di inizio (formato "dd/mm/yyyy HH:MM").
    - `fine`: stringa con data/ora di fine (formato "dd/mm/yyyy HH:MM").
  - *Output:* Un nuovo `Intervallo` allocato dinamicamente, oppure `NULL` in caso di errore o parametri non validi.
  - *Pre-condizioni:*
    - `inizio` non deve essere `NULL` e deve essere nel formato corretto.
    - `fine` non deve essere `NULL` e deve essere nel formato corretto.
  - *Post-condizioni:* Viene restituito un puntatore a un nuovo intervallo convertito dalle stringhe.
  - *Effetti Collaterali:* Alloca memoria per la struttura `Intervallo`.

- **duplica_intervallo(Intervallo i)**
  - *Descrizione:* Crea una copia dell'intervallo specificato.
  - *Input:* `i`: puntatore all'intervallo da duplicare.
  - *Output:* Una nuova copia dell'intervallo (`Intervallo`), oppure `NULL` se `i` è `NULL` o in caso di errore.
  - *Pre-condizioni:* `i` deve essere un puntatore valido (può essere `NULL`).
  - *Post-condizioni:* Viene restituita una nuova copia dell'intervallo.
  - *Effetti Collaterali:* Alloca memoria per la nuova struttura `Intervallo`.

- **compara_intervalli(Intervallo a, Intervallo b)**
  - *Descrizione:* Confronta due intervalli in base al loro tempo di inizio.
  - *Input:*
    - `a`: primo intervallo da confrontare.
    - `b`: secondo intervallo da confrontare.
  - *Output:*
    - -1 se `a` inizia prima di `b`.

    - 1 se `a` inizia dopo `b`.
    - 0 se iniziano nello stesso momento o in caso di errori (`Byte`).
  - *Pre-condizioni:*
    - `a` non deve essere `NULL`.
    - `b` non deve essere `NULL`.
  - *Post-condizioni:* Restituisce un valore che indica la relazione temporale tra gli intervalli.
  - *Effetti Collaterali:* Nessuno.

- **intervallo_in_stringa(Intervallo i)**
  - *Descrizione:* Crea una rappresentazione testuale dell'intervallo.
  - *Input:* `i`: puntatore all'intervallo da convertire.
  - *Output:* Una stringa allocata dinamicamente nel formato "dd/mm/yyyy HH:MM -> dd/mm/yyyy HH:MM" (`char *`), oppure `NULL` se `i` è `NULL` o in caso di errore.
  - *Pre-condizioni:* `i` deve essere un puntatore valido (può essere `NULL`).
  - *Post-condizioni:* Restituisce una stringa che rappresenta l'intervallo.
  - *Effetti Collaterali:* Alloca memoria per la stringa risultante.

—

4.5.3 ADT Prenotazione

L'ADT `Prenotazione` rappresenta una singola prenotazione, includendo dettagli come cliente, veicolo, intervallo temporale e costo.

- **crea_prenotazione(const char *cliente, const char *targa, Intervallo i, double costo)**
  - *Descrizione:* Crea una nuova prenotazione con i dati specificati.
  - *Input:*
    - `cliente`: nome del cliente che effettua la prenotazione.
    - `targa`: targa del veicolo associato alla prenotazione.
    - `i`: intervallo temporale della prenotazione.
    - `costo`: costo totale della prenotazione.
  - *Output:* Un puntatore a una nuova struttura `Prenotazione` se l'allocazione e l'inizializzazione hanno successo, altrimenti `NULL` (`Prenotazione`).

- *Pre-condizioni:*
  - `cliente` deve essere una stringa non nulla.
  - `targa` deve essere una stringa non nulla.
  - `i` deve essere un intervallo valido e non nullo.
- *Post-condizioni:* Viene restituita una nuova prenotazione inizializzata con i dati forniti.
- *Effetti Collaterali:* Alloca memoria dinamica per la struttura `Prenotazione` e per le stringhe `cliente` e `targa`.

- **distruggi_prenotazione(Prenotazione p)**
  - *Descrizione:* Libera la memoria allocata per una data prenotazione.
  - *Input:* `p`: puntatore alla prenotazione da distruggere.
  - *Output:* Nessun valore.
  - *Pre-condizioni:* `p` deve essere un puntatore valido a una struttura `Prenotazione`.
  - *Post-condizioni:* La memoria associata alla prenotazione `p` (inclusi i campi interni `cliente`, `targa` e `intervallo`) viene deallocata. Se `p` è `NULL`, la funzione non ha alcun effetto.
  - *Effetti Collaterali:* Dealloca memoria dinamica precedentemente allocata per la prenotazione e i suoi campi.

- **ottieni_cliente_prenotazione(Prenotazione p)**
  - *Descrizione:* Restituisce il nome del cliente associato alla prenotazione.
  - *Input:* `p`: puntatore alla prenotazione.
  - *Output:* Una stringa contenente il nome del cliente (`const char *`), o `NULL` se `p` è `NULL`.
  - *Pre-condizioni:* `p` deve essere un puntatore valido a una struttura `Prenotazione`.
  - *Post-condizioni:* Restituisce il nome del cliente.
  - *Effetti Collaterali:* Nessuno.

- **ottieni_veicolo_prenotazione(Prenotazione p)**
  - *Descrizione:* Restituisce la targa del veicolo associato alla prenotazione.
  - *Input:* `p`: puntatore alla prenotazione.
  - *Output:* Una stringa contenente la targa del veicolo (`const char *`), o `NULL` se `p` è `NULL`.
  - *Pre-condizioni:* `p` deve essere un puntatore valido a una struttura `Prenotazione`.

- *Post-condizioni:* Restituisce la targa del veicolo.
- *Effetti Collaterali:* Nessuno.

- **ottieni_intervallo_prenotazione(Prenotazione p)**
  - *Descrizione:* Restituisce l'intervallo temporale della prenotazione.
  - *Input:* `p`: puntatore alla prenotazione.
  - *Output:* Un puntatore all'intervallo temporale (`Intervallo`), o `NULL` se `p` è `NULL`.
  - *Pre-condizioni:* `p` deve essere un puntatore valido a una struttura `Prenotazione`.
  - *Post-condizioni:* Restituisce l'intervallo temporale.
  - *Effetti Collaterali:* Nessuno.

- **ottieni_costo_prenotazione(Prenotazione p)**
  - *Descrizione:* Restituisce il costo della prenotazione.
  - *Input:* `p`: puntatore alla prenotazione.
  - *Output:* Il valore del costo della prenotazione (`double`), o -1.0 se `p` è `NULL`.
  - *Pre-condizioni:* `p` deve essere un puntatore valido a una struttura `Prenotazione`.
  - *Post-condizioni:* Restituisce il costo della prenotazione.
  - *Effetti Collaterali:* Nessuno.

- **imposta_cliente_prenotazione(Prenotazione p, const char *cliente)**
  - *Descrizione:* Modifica il nome del cliente associato alla prenotazione.
  - *Input:*
    - `p`: puntatore alla prenotazione.
    - `cliente`: la nuova stringa contenente il nome del cliente.
  - *Output:* Nessun valore.
  - *Pre-condizioni:*
    - `p` deve essere un puntatore valido a una struttura `Prenotazione`.
    - `cliente` deve essere una stringa non nulla.
  - *Post-condizioni:* Il nome del cliente nella prenotazione `p` viene aggiornato.
  - *Effetti Collaterali:* Dealloca la memoria precedentemente allocata per il vecchio nome del cliente. Alloca nuova memoria dinamica per duplicare la stringa `cliente`. Modifica la memoria interna della struttura `Prenotazione`.

- **imposta_veicolo_prenotazione(Prenotazione p, const char *targa)**
  - *Descrizione:* Modifica la targa del veicolo associato alla prenotazione.

- *Input:*
  - `p`: puntatore alla prenotazione.
  - `targa`: la nuova stringa contenente la targa del veicolo.
- *Output:* Nessun valore.
- *Pre-condizioni:*
  - `p` deve essere un puntatore valido a una struttura `Prenotazione`.
  - `targa` deve essere una stringa non nulla.
- *Post-condizioni:* La targa del veicolo nella prenotazione `p` viene aggiornata.
- *Effetti Collaterali:* Dealloca la memoria precedentemente allocata per la vecchia targa del veicolo. Alloca nuova memoria dinamica per duplicare la stringa `targa`. Modifica la memoria interna della struttura `Prenotazione`.

- **imposta_intervallo_prenotazione(Prenotazione p, Intervallo i)**
  - *Descrizione:* Modifica l'intervallo temporale della prenotazione.
  - *Input:*
    - `p`: puntatore alla prenotazione.
    - `i`: il nuovo intervallo temporale.
  - *Output:* Nessun valore.
  - *Pre-condizioni:*
    - `p` deve essere un puntatore valido a una struttura `Prenotazione`.
    - `i` deve essere un puntatore valido a una struttura `Intervallo`.
  - *Post-condizioni:* L'intervallo temporale nella prenotazione `p` viene aggiornato con il nuovo intervallo `i`.
  - *Effetti Collaterali:* Dealloca la memoria precedentemente allocata per il vecchio intervallo. Modifica la memoria interna della struttura `Prenotazione`.

- **imposta_costo_prenotazione(Prenotazione p, double costo)**
  - *Descrizione:* Modifica il costo della prenotazione.
  - *Input:*
    - `p`: puntatore alla prenotazione.
    - `costo`: il nuovo valore double del costo.
  - *Output:* Nessun valore.
  - *Pre-condizioni:* `p` deve essere un puntatore valido a una struttura `Prenotazione`. *Post-condizioni:* Il costo della prenotazione `p` viene aggiornato con il nuovo valore `costo`.

- *Effetti Collaterali:* Modifica la memoria interna della struttura `Prenotazione`.

- **duplica_prenotazione(Prenotazione p)**
  - *Descrizione:* Crea una copia profonda della prenotazione data.
  - *Input:* `p`: puntatore alla prenotazione da duplicare.
  - *Output:* Un puntatore a una nuova struttura `Prenotazione` che è una copia esatta di `p` se `p` è valido (`Prenotazione`), altrimenti `NULL`.
  - *Pre-condizioni:* `p` deve essere un puntatore valido a una struttura `Prenotazione`.
  - *Post-condizioni:* Viene restituita una copia profonda della prenotazione.
  - *Effetti Collaterali:* Alloca nuova memoria dinamica per la prenotazione duplicata e per le sue stringhe e l'intervallo.

- **prenotazione_in_stringa(Prenotazione p)**
  - *Descrizione:* Genera una rappresentazione testuale formattata dei dati della prenotazione.
  - *Input:* `p`: puntatore alla prenotazione di cui si vuole ottenere la rappresentazione in stringa.
  - *Output:* Una stringa allocata dinamicamente contenente i dettagli della prenotazione (`char *`), o `NULL` se `p` è `NULL` o in caso di errore.
  - *Pre-condizioni:* `p` deve essere un puntatore valido a una struttura `Prenotazione`.
  - *Post-condizioni:* Restituisce una stringa che rappresenta la prenotazione.
  - *Effetti Collaterali:* Alloca memoria dinamica per la stringa risultante.

—

4.5.4 ADT Utente

L'ADT `Utente` gestisce le informazioni di un utente, inclusi i suoi dati personali, credenziali e storico delle prenotazioni.

- **crea_utente(const char *email, const uint8_t *password, const char *nome, const char *cognome, Byte permesso)**
  - *Descrizione:* Crea un nuovo utente con email, password, nome, cognome e permesso.
  - *Input:*
    - `email`: stringa che rappresenta l'email dell'utente.

- `password`: array di `uint8_t` che rappresenta la password dell'utente.
- `nome`: stringa che rappresenta il nome dell'utente.
- `cognome`: stringa che rappresenta il cognome dell'utente.
- `permesso`: byte che rappresenta il livello di permesso (`ADMIN` o `CLIENTE`).

– *Output:* Un nuovo utente allocato dinamicamente (`Utente`), oppure `NULL` se l'allocazione fallisce.

– *Pre-condizioni:*

- `email`, `password`, `nome`, `cognome` non devono essere `NULL` o vuoti.
- `permesso` deve essere un valore valido (`ADMIN` o `CLIENTE`).

– *Post-condizioni:* Viene restituito un nuovo utente inizializzato.

– *Effetti Collaterali:* Allocazione dinamica di memoria.

- **distruggi_utente(Utente utente)**

– *Descrizione:* Dealloca la memoria associata all'utente, inclusi eventuali dati interni.

– *Input:* `utente`: puntatore all'utente da distruggere.

– *Output:* Nessun valore.

– *Pre-condizioni:* `utente` non deve essere `NULL`.

– *Post-condizioni:* La memoria occupata dall'utente è liberata.

– *Effetti Collaterali:* Deallocazione dinamica di memoria.

- **imposta_nome(Utente utente, const char *nome)**

– *Descrizione:* Imposta il nome dell'utente.

– *Input:*

- `utente`: puntatore all'utente.
- `nome`: stringa contenente il nuovo nome.

– *Output:* Nessun valore.

– *Pre-condizioni:* `utente` e `nome` non devono essere `NULL`.

– *Post-condizioni:* Il nome dell'utente è aggiornato.

– *Effetti Collaterali:* Modifica interna dell'utente.

- **imposta_cognome(Utente utente, const char *cognome)**

– *Descrizione:* Imposta il cognome dell'utente.

– *Input:*

    - `utente`: puntatore all'utente.
    - `cognome`: stringa contenente il nuovo cognome.
  - *Output:* Nessun valore.
  - *Pre-condizioni:* `utente` e `cognome` non devono essere `NULL`.
  - *Post-condizioni:* Il cognome dell'utente è aggiornato.
  - *Effetti Collaterali:* Modifica interna dell'utente.

- **imposta_email(Utente utente, const char *email)**
  - *Descrizione:* Imposta l'email dell'utente.
  - *Input:*
    - `utente`: puntatore all'utente.
    - `email`: stringa contenente la nuova email.
  - *Output:* Nessun valore.
  - *Pre-condizioni:* `utente` e `email` non devono essere `NULL`.
  - *Post-condizioni:* L'email dell'utente è aggiornata.
  - *Effetti Collaterali:* Modifica interna dell'utente.

- **imposta_password(Utente utente, const uint8_t *password)**
  - *Descrizione:* Imposta la password dell'utente.
  - *Input:*
    - `utente`: puntatore all'utente.
    - `password`: array di `uint8_t` contenente la nuova password.
  - *Output:* Nessun valore.
  - *Pre-condizioni:* `utente` e `password` non devono essere `NULL`.
  - *Post-condizioni:* La password dell'utente è aggiornata.
  - *Effetti Collaterali:* Modifica interna dell'utente.

- **imposta_permesso(Utente utente, Byte permesso)**
  - *Descrizione:* Imposta il livello di permesso dell'utente.
  - *Input:*
    - `utente`: puntatore all'utente.
    - `permesso`: byte che rappresenta il permesso (`ADMIN` o `CLIENTE`).
  - *Output:* Nessun valore.

  - *Pre-condizioni:* `utente` non deve essere `NULL`. `permesso` deve essere un valore valido.
  - *Post-condizioni:* Il permesso dell'utente è aggiornato.
  - *Effetti Collaterali:* Modifica interna dell'utente.

- **ottieni_cognome(const Utente utente)**
  - *Descrizione:* Restituisce il cognome dell'utente.
  - *Input:* `utente`: puntatore all'utente.
  - *Output:* Una stringa contenente il cognome (`const char *`).
  - *Pre-condizioni:* `utente` non deve essere `NULL`.
  - *Post-condizioni:* Restituisce il cognome dell'utente.
  - *Effetti Collaterali:* Nessuno.

- **ottieni_nome(const Utente utente)**
  - *Descrizione:* Restituisce il nome dell'utente.
  - *Input:* `utente`: puntatore all'utente.
  - *Output:* Una stringa contenente il nome (`const char *`).
  - *Pre-condizioni:* `utente` non deve essere `NULL`.
  - *Post-condizioni:* Restituisce il nome dell'utente.
  - *Effetti Collaterali:* Nessuno.

- **ottieni_email(const Utente utente)**
  - *Descrizione:* Restituisce l'email dell'utente.
  - *Input:* `utente`: puntatore all'utente.
  - *Output:* Una stringa contenente l'email (`const char *`).
  - *Pre-condizioni:* `utente` non deve essere `NULL`.
  - *Post-condizioni:* Restituisce l'email dell'utente.
  - *Effetti Collaterali:* Nessuno.

- **ottieni_password(const Utente utente)**
  - *Descrizione:* Restituisce la password dell'utente.
  - *Input:* `utente`: puntatore all'utente.
  - *Output:* Un array di `uint8_t` contenente la password (`const uint8_t *`).
  - *Pre-condizioni:* `utente` non deve essere `NULL`.

  - *Post-condizioni:* Restituisce la password dell'utente.
  - *Effetti Collaterali:* Nessuno.

- **ottieni_permesso(Utente utente)**
  - *Descrizione:* Restituisce il livello di permesso dell'utente.
  - *Input:* `utente`: puntatore all'utente.
  - *Output:* Un `Byte` rappresentante il permesso.
  - *Pre-condizioni:* `utente` non deve essere `NULL`.
  - *Post-condizioni:* Restituisce il permesso dell'utente.
  - *Effetti Collaterali:* Nessuno.

- **ottieni_storico_utente(Utente utente)**
  - *Descrizione:* Restituisce la lista delle prenotazioni associate all'utente.
  - *Input:* `utente`: puntatore all'utente.
  - *Output:* Una lista contenente le prenotazioni (`ListaPre`).
  - *Pre-condizioni:* `utente` non deve essere `NULL`.
  - *Post-condizioni:* Restituisce la lista delle prenotazioni dell'utente.
  - *Effetti Collaterali:* Nessuno.

- **aggiungi_a_storico_utente(Utente utente, Prenotazione prenotazione)**
  - *Descrizione:* Aggiunge una prenotazione allo storico dell'utente.
  - *Input:*
    - `utente`: puntatore all'utente.
    - `prenotazione`: prenotazione da aggiungere.
  - *Output:* `Byte` (1) se l'operazione ha avuto successo, 0 in caso di fallimento.
  - *Pre-condizioni:* `utente` e `prenotazione` non devono essere `NULL`.
  - *Post-condizioni:* Aggiunge la prenotazione alla lista.
  - *Effetti Collaterali:* Modifica interna della lista storico dell'utente.

- **rimuovi_da_storico_utente(Utente utente, Prenotazione prenotazione)**
  - *Descrizione:* Rimuove una prenotazione dallo storico dell'utente.
  - *Input:*
    - `utente`: puntatore all'utente.
    - `prenotazione`: prenotazione da rimuovere.

  - *Output:* `Byte` (1) se l'operazione ha avuto successo, 0 in caso di fallimento.
  - *Pre-condizioni:* `utente` e `prenotazione` non devono essere `NULL`.
  - *Post-condizioni:* Rimuove la prenotazione dalla lista.
  - *Effetti Collaterali:* Modifica interna della lista storico dell'utente.

- **ottieni_numero_prenotazioni_utente(const Utente u)**
  - *Descrizione:* Restituisce il numero di prenotazioni contenute nella struttura `Utente`.
  - *Input:* `u`: puntatore all'utente.
  - *Output:* Numero di prenotazioni (`unsigned int`).
  - *Pre-condizioni:* `u` non deve essere `NULL`.
  - *Post-condizioni:* Restituisce il numero di prenotazioni dell'utente.
  - *Effetti Collaterali:* Nessuno.

- **utente_in_stringa(const Utente utente)**
  - *Descrizione:* Genera una rappresentazione testuale formattata dei dati dell'utente.
  - *Input:* `utente`: puntatore all'utente di cui si vuole ottenere la rappresentazione in stringa.
  - *Output:* Una stringa allocata dinamicamente contenente i dettagli dell'utente (`char *`), o `NULL` in caso di errore.
  - *Pre-condizioni:* `utente` non deve essere `NULL`.
  - *Post-condizioni:* Restituisce una stringa che rappresenta l'utente.
  - *Effetti Collaterali:* Alloca memoria dinamica per la stringa risultante.

—

4.5.5 ADT Veicolo

L'ADT `Veicolo` rappresenta un veicolo gestibile dal sistema, con attributi come targa, modello, posizione, tariffa e un albero di prenotazioni associate.

- **crea_veicolo(const char *tipo, const char *targa, const char *modello, const char *posizione, double tariffa, Prenotazioni prenotazioni)**
  - *Descrizione:* Crea un nuovo veicolo con targa, modello, posizione, tariffa e prenotazioni associate.
  - *Input:*

    - `tipo`: stringa che rappresenta il tipo del veicolo.
    - `targa`: stringa che rappresenta la targa del veicolo.
    - `modello`: stringa che rappresenta il modello del veicolo.
    - `posizione`: stringa che rappresenta la posizione del veicolo.
    - `tariffa`: numero che rappresenta la tariffa al minuto in euro di un veicolo.
    - `prenotazioni`: puntatore alle prenotazioni associate a un veicolo.
  - *Output:* Un nuovo oggetto `Veicolo` se l'allocazione è andata a buon fine, altrimenti `NULL` (`Veicolo`).
  - *Pre-condizioni:*
    - `targa`: non deve essere `NULL` e deve essere di 7 caratteri.
    - `modello`: non deve essere `NULL`.
    - `posizione`: non deve essere `NULL`.
    - `tariffa`: deve essere maggiore di 0.
  - *Post-condizioni:* Restituisce un nuovo veicolo inizializzato con i dati forniti.
  - *Effetti Collaterali:* Alloca memoria dinamicamente per il veicolo.

- **distruggi_veicolo(const Veicolo v)**
  - *Descrizione:* Elimina un veicolo.
  - *Input:* `v`: puntatore a un veicolo.
  - *Output:* Nessun valore.
  - *Pre-condizioni:* `v`: non deve essere `NULL`.
  - *Post-condizioni:* Libera la memoria occupata dalla struttura `Veicolo` e dalle sue prenotazioni.
  - *Effetti Collaterali:* Libera la memoria occupata dalla struttura `Veicolo` e dalle sue prenotazioni.

- **ottieni_targa(const Veicolo v)**
  - *Descrizione:* Restituisce la targa del veicolo puntato da `v`.
  - *Input:* `v`: puntatore a un veicolo.
  - *Output:* Una stringa contenente la targa del veicolo (`const char *`).
  - *Pre-condizioni:* `v`: non deve essere `NULL`.
  - *Post-condizioni:* Restituisce la targa del veicolo.
  - *Effetti Collaterali:* Nessuno.

- **imposta_targa(Veicolo v, const char *targa)**

  - *Descrizione:* Imposta la targa nel veicolo puntato da `v`.
  - *Input:*
    - `v`: puntatore a un veicolo.
    - `targa`: stringa contenente la targa da impostare.
  - *Output:* Nessun valore.
  - *Pre-condizioni:*
    - `v`: non deve essere `NULL`.
    - `targa`: non deve essere `NULL` e deve essere di 7 caratteri.
  - *Post-condizioni:* Aggiorna il campo `targa` del veicolo `v` con il valore fornito.
  - *Effetti Collaterali:* Aggiorna il campo `targa` del veicolo `v` con il valore fornito.

- **ottieni_modello(const Veicolo v)**
  - *Descrizione:* Restituisce il modello del veicolo puntato da `v`.
  - *Input:* `v`: puntatore a un veicolo.
  - *Output:* Una stringa contenente il modello del veicolo (`const char *`).
  - *Pre-condizioni:* `v`: non deve essere `NULL`.
  - *Post-condizioni:* Restituisce il modello del veicolo. *Effetti Collaterali:* Nessuno.

- **imposta_modello(Veicolo v, const char *modello)**
  - *Descrizione:* Imposta il modello del veicolo puntato da `v`.
  - *Input:*
    - `v`: puntatore a un veicolo.
    - `modello`: stringa contenente il modello da impostare.
  - *Output:* Nessun valore.
  - *Pre-condizioni:*
    - `v`: non deve essere `NULL`.
    - `modello`: non deve essere `NULL` e deve essere lungo al massimo 50 caratteri.
  - *Post-condizioni:* Aggiorna il campo `modello` del veicolo `v` con il valore fornito.
  - *Effetti Collaterali:* Aggiorna il campo `modello` del veicolo `v` con il valore fornito.

- **ottieni_posizione(const Veicolo v)**
  - *Descrizione:* Restituisce la posizione del veicolo puntato da `v`.
  - *Input:* `v`: puntatore a un veicolo.
  - *Output:* Una stringa contenente la posizione del veicolo (`const char *`).

  - *Pre-condizioni:* `v`: non deve essere `NULL`.
  - *Post-condizioni:* Restituisce la posizione del veicolo.
  - *Effetti Collaterali:* Nessuno.

- **imposta_posizione(Veicolo v, const char *posizione)**
  - *Descrizione:* Imposta la posizione del veicolo puntato da `v`.
  - *Input:*
    - `v`: puntatore a un veicolo.
    - `posizione`: stringa contenente la posizione da impostare.
  - *Output:* Nessun valore.
  - *Pre-condizioni:*
    - `v`: non deve essere `NULL`.
    - `posizione`: non deve essere `NULL` e deve essere lungo al massimo 200 caratteri.
  - *Post-condizioni:* Aggiorna il campo `posizione` del veicolo `v` con il valore fornito.
  - *Effetti Collaterali:* Aggiorna il campo `posizione` del veicolo `v` con il valore fornito.

- **ottieni_tariffa(const Veicolo v)**
  - *Descrizione:* Restituisce la tariffa al minuto del veicolo puntato da `v`.
  - *Input:* `v`: puntatore a un veicolo.
  - *Output:* La tariffa del veicolo (`double`).
  - *Pre-condizioni:* `v`: non deve essere `NULL`.
  - *Post-condizioni:* Restituisce la tariffa del veicolo.
  - *Effetti Collaterali:* Nessuno.

- **imposta_tariffa(Veicolo v, double tariffa)**
  - *Descrizione:* Imposta la tariffa del veicolo puntato da `v`.
  - *Input:*
    - `v`: puntatore a un veicolo.
    - `tariffa`: valore double contenente la tariffa da impostare.
  - *Output:* Nessun valore.
  - *Pre-condizioni:*
    - `v`: non deve essere `NULL`.
    - `tariffa`: deve essere maggiore di 0.

  - *Post-condizioni:* Aggiorna il campo `tariffa` del veicolo `v` con il valore fornito.
  - *Effetti Collaterali:* Aggiorna il campo `tariffa` del veicolo `v` con il valore fornito.

- **ottieni_prenotazioni_veicolo(const Veicolo v)**
  - *Descrizione:* Restituisce l'albero delle prenotazioni associate al veicolo.
  - *Input:* `v`: puntatore a un veicolo.
  - *Output:* L'albero delle prenotazioni (`Prenotazioni`).
  - *Pre-condizioni:* `v`: non deve essere `NULL`.
  - *Post-condizioni:* Restituisce l'albero delle prenotazioni del veicolo.
  - *Effetti Collaterali:* Nessuno.

- **imposta_prenotazioni_veicolo(Veicolo v, Prenotazioni prenotazioni)**
  - *Descrizione:* Imposta l'albero delle prenotazioni nel veicolo puntato da `v`.
  - *Input:*
    - `v`: puntatore a un veicolo.
    - `prenotazioni`: puntatore all'albero di prenotazioni da impostare.
  - *Output:* Nessun valore.
  - *Pre-condizioni:* `v`: non deve essere `NULL`.
  - *Post-condizioni:* Aggiorna il campo `prenotazioni` del veicolo `v` con il valore fornito.
  - *Effetti Collaterali:* Aggiorna il campo `prenotazioni` del veicolo `v` con il valore fornito.

- **ottieni_tipo(const Veicolo v)**
  - *Descrizione:* Restituisce il tipo del veicolo puntato da `v`.
  - *Input:* `v`: puntatore a un veicolo.
  - *Output:* Una stringa contenente il tipo del veicolo (`const char *`).
  - *Pre-condizioni:* `v`: non deve essere `NULL`.
  - *Post-condizioni:* Restituisce il tipo del veicolo.
  - *Effetti Collaterali:* Nessuno.

- **imposta_tipo(Veicolo v, const char *tipo)**
  - *Descrizione:* Imposta il tipo del veicolo puntato da `v`.
  - *Input:*
    - `v`: puntatore a un veicolo.

    - `tipo`: stringa contenente il tipo da impostare.
  - *Output:* Nessun valore.
  - *Pre-condizioni:*
    - `v`: non deve essere `NULL`.
    - `tipo`: non deve essere `NULL` e deve essere lungo al massimo 30 caratteri.
  - *Post-condizioni:* Aggiorna il campo `tipo` del veicolo `v` con il valore fornito.
  - *Effetti Collaterali:* Aggiorna il campo `tipo` del veicolo `v` con il valore fornito.

- **veicolo_in_stringa(const Veicolo v)**
  - *Descrizione:* Genera una rappresentazione testuale formattata dei dati del veicolo.
  - *Input:* `v`: puntatore al veicolo di cui si vuole ottenere la rappresentazione in stringa.
  - *Output:* Una stringa allocata dinamicamente contenente i dettagli del veicolo (`char *`), o `NULL` in caso di errore.
  - *Pre-condizioni:* `v`: non deve essere `NULL`.
  - *Post-condizioni:* Restituisce una stringa che rappresenta il veicolo.
  - *Effetti Collaterali:* Alloca memoria dinamica per la stringa risultante.

4.5.6 ADT Coda

L'ADT `Coda` implementa una struttura dati di tipo FIFO (First In, First Out) che permette di aggiungere elementi alla fine e rimuoverli dall'inizio.

Creata per caricare le prenotazioni in un file, serviva caricarle per livello

- **crea_coda()**
  - *Descrizione:* Crea e inizializza una nuova coda vuota allocando dinamicamente la memoria necessaria.
  - *Input:* Nessun parametro.
  - *Output:* Puntatore di tipo `Coda` alla nuova struttura creata, oppure `NULL` in caso di fallimento dell'allocazione.
  - *Pre-condizioni:* Nessuna.
  - *Post-condizioni:* Una nuova struttura Coda viene allocata e i suoi puntatori interni (testa e coda) sono impostati a `NULL`, indicando una coda vuota.
  - *Effetti Collaterali:* Alloca memoria dinamica per la struttura della coda.

- **distruggi_coda(Coda coda, void (*distruttore)(void *))**
  - *Descrizione:* Distrugge una coda esistente, liberando tutta la memoria associata ai suoi nodi e alla struttura della coda stessa.
  - *Input:*
    - `coda`: puntatore alla coda da distruggere.
    - `distruttore`: puntatore a una funzione di callback che verrà applicata a ogni elemento della coda prima che il nodo venga liberato. Può essere `NULL` se gli elementi non richiedono deallocazione specifica.
  - *Output:* Nessun valore.
  - *Pre-condizioni:* `coda` deve essere un puntatore valido a una coda precedentemente creata con `crea_coda` o `NULL`.
  - *Post-condizioni:* Tutta la memoria allocata per la coda viene rilasciata e la coda diventa inutilizzabile.
  - *Effetti Collaterali:*
    - Tutta la memoria allocata per la coda (nodi ed elementi, se `distruttore` è fornito) viene rilasciata.
    - La coda diventa inutilizzabile dopo questa chiamata.
    - Se `distruttore` è fornito, applica questa funzione a ciascun elemento prima di liberare il nodo.
- **aggiungi_in_coda(void *elemento, Coda coda)**
  - *Descrizione:* Aggiunge un elemento alla fine della coda creando un nuovo nodo.
  - *Input:*
    - `elemento`: puntatore generico all'elemento da inserire nella coda.
    - `coda`: puntatore alla coda in cui inserire l'elemento.
  - *Output:* Valore intero: `0` se l'elemento viene aggiunto correttamente, `-1` in caso di errore.
  - *Pre-condizioni:*
    - `coda`: non deve essere `NULL`.
    - `elemento`: non deve essere `NULL`.
  - *Post-condizioni:* L'elemento viene aggiunto alla fine della coda se l'operazione ha successo.
  - *Effetti Collaterali:*
    - Alloca un nuovo nodo nella memoria dinamica.
    - Modifica i puntatori interni della coda (`coda->coda` e possibilmente `coda->testa`).

- **rimuovi_dalla_coda(Coda coda)**
  - *Descrizione:* Rimuove e restituisce l'elemento che si trova in testa alla coda (il primo elemento aggiunto secondo la politica FIFO).
  - *Input:*
    - `coda`: puntatore alla coda da cui rimuovere l'elemento.
  - *Output:* Puntatore generico all'elemento rimosso, oppure `NULL` se la coda è vuota o non valida.
  - *Pre-condizioni:* `coda`: non deve essere `NULL`.
  - *Post-condizioni:* L'elemento in testa alla coda viene rimosso se presente.
  - *Effetti Collaterali:*
    - L'elemento in testa alla coda viene rimosso.
    - La coda viene modificata, con il suo nuovo elemento in testa che era il secondo elemento prima della rimozione.
    - Se la coda diventa vuota, i puntatori `testa` e `coda` della struttura vengono impostati a `NULL`.
    - Libera la memoria dinamica occupata dal nodo rimosso dalla testa della coda.
- **coda_vuota(Coda coda)**
  - *Descrizione:* Verifica se la coda è vuota controllando se contiene elementi.
  - *Input:*
    - `coda`: puntatore alla coda da controllare.
  - *Output:* Valore intero: `1` se la coda è vuota o è `NULL`, `0` se contiene almeno un elemento.
  - *Pre-condizioni:* `coda`: non deve essere `NULL`.
  - *Post-condizioni:* Restituisce lo stato di vuoto/non vuoto della coda senza modificarla.
  - *Effetti Collaterali:* Nessuno, la funzione è di sola lettura.

4.5.7 ADT Lista

L'ADT `Lista` rappresenta una lista collegata singolarmente che permette di gestire una sequenza di elementi di tipo generico attraverso nodi collegati.

- **crea_lista()**
  - *Descrizione:* Crea e restituisce una lista vuota.

- *Input:* Nessuno.
- *Output:* Un valore `NULL` che rappresenta una lista vuota (`Nodo`).
- *Pre-condizioni:* Nessuna.
- *Post-condizioni:* Viene restituito un valore `NULL` che rappresenta una lista vuota.
- *Effetti Collaterali:* Nessuno.

- **distruggi_nodo(Nodo nodo, void (*funzione_distruggi_item)(void *))**
  - *Descrizione:* Libera la memoria occupata da un nodo, distruggendo anche l'item se indicato.
  - *Input:*
    - `nodo`: il nodo da distruggere.
    - `funzione_distruggi_item`: puntatore a funzione che distrugge il contenuto `item`.
  - *Output:* Nessuno (`void`).
  - *Pre-condizioni:*
    - `nodo` non deve essere `NULL`.
  - *Post-condizioni:* Il nodo viene deallocato.
  - *Effetti Collaterali:* Il nodo viene deallocato. Se la funzione è fornita, viene chiamata sull'item.

- **aggiungi_nodo(Item item, Nodo nodo)**
  - *Descrizione:* Aggiunge un nuovo nodo in testa alla lista, contenente l'item fornito.
  - *Input:*
    - `item`: puntatore all'oggetto da inserire nel nodo.
    - `nodo`: nodo a cui collegare il nuovo nodo in testa.
  - *Output:* La lista con un nuovo nodo, in caso di errore restituisce la lista iniziale (`Nodo`).
  - *Pre-condizioni:*
    - `item` non deve essere `NULL`.
    - `nodo` non deve essere `NULL`.
  - *Post-condizioni:* Viene restituita la lista con un nuovo nodo in testa.
  - *Effetti Collaterali:* Alloca memoria dinamica per il nuovo nodo.

- **ottieni_prossimo(Nodo nodo)**

  - *Descrizione:* Restituisce il puntatore al nodo successivo nella lista.
  - *Input:*
    - `nodo`: nodo da cui ottenere il successivo.
  - *Output:* Un puntatore al nodo successivo nella lista, in caso di errore restituisce `NULL` (`Nodo`).
  - *Pre-condizioni:*
    - `nodo` non deve essere `NULL`.
  - *Post-condizioni:* Viene restituito il puntatore al nodo successivo.
  - *Effetti Collaterali:* Nessuno.

- **ottieni_item(Nodo nodo)**
  - *Descrizione:* Restituisce il contenuto `item` di un nodo.
  - *Input:*
    - `nodo`: nodo da cui ottenere l'item.
  - *Output:* Il contenuto del nodo (`Item`).
  - *Pre-condizioni:*
    - `nodo` non deve essere `NULL`.
  - *Post-condizioni:* Viene restituito il contenuto del nodo.
  - *Effetti Collaterali:* Nessuno.

- **imposta_prossimo(Nodo nodo, Nodo prossimo)**
  - *Descrizione:* Imposta il nodo successivo per un nodo corrente.
  - *Input:*
    - `nodo`: nodo da modificare.
    - `prossimo`: nodo da impostare come successivo.
  - *Output:* Nessuno (`void`).
  - *Pre-condizioni:*
    - `nodo` non deve essere `NULL`.
    - `prossimo` non deve essere `NULL`.
  - *Post-condizioni:* Il campo `next` del nodo viene aggiornato.
  - *Effetti Collaterali:* Il campo `next` del nodo viene aggiornato con il nuovo valore.

- **inverti_lista(Nodo nodo)**
  - *Descrizione:* Inverte l'ordine degli elementi di una lista collegata singolarmente.

- *Input:*
  - `nodo`: puntatore al primo nodo della lista originale.
- *Output:* La lista invertita (`Nodo`).
- *Pre-condizioni:*
  - `nodo` non deve essere `NULL`.
- *Post-condizioni:* Viene restituita la lista con l'ordine degli elementi invertito.
- *Effetti Collaterali:* La lista è invertita modificando i puntatori `next` di ciascun nodo.

- **lista_vuota(Nodo lista)**
  - *Descrizione:* Verifica se una lista è vuota.
  - *Input:*
    - `lista`: puntatore al nodo iniziale della lista.
  - *Output:* Un valore di tipo `Byte`: 1 se la lista è vuota, altrimenti 0 (`Byte`).
  - *Pre-condizioni:* Nessuna.
  - *Post-condizioni:* Viene restituito 1 se la lista è vuota, altrimenti 0.
  - *Effetti Collaterali:* Nessuno.

4.5.8 ADT Lista Prenotazione

L'ADT `Lista Prenotazione` implementa una struttura dati di tipo lista concatenata per la gestione di prenotazioni. Permette di aggiungere, rimuovere e accedere alle prenotazioni in modo dinamico, mantenendo un riferimento agli intervalli temporali.

- **distruggi_lista_prenotazione(ListaPre l)**
  - *Descrizione:* Libera tutta la memoria occupata da una lista di prenotazioni, inclusi tutti i nodi e le prenotazioni contenute.
  - *Input:* `l` - lista di prenotazioni da distruggere.
  - *Output:* Nessun valore di ritorno (`void`).
  - *Pre-condizioni:* `l` non deve essere `NULL`.
  - *Post-condizioni:* Tutta la memoria della lista viene liberata.
  - *Effetti Collaterali:* Dealloca la memoria di tutti i nodi e le prenotazioni della lista.

- **aggiungi_prenotazione_lista(ListaPre l, Prenotazione p)**
  - *Descrizione:* Aggiunge una nuova prenotazione all'inizio della lista.

- *Input:* `l` - testa corrente della lista (può essere `NULL`), `p` - prenotazione da aggiungere.
- *Output:* Puntatore di tipo `ListaPre` alla nuova testa della lista, oppure `NULL` in caso di errore di allocazione.
- *Pre-condizioni:* `p` non deve essere `NULL`, `l` non deve essere `NULL`.
- *Post-condizioni:* La prenotazione viene aggiunta all'inizio della lista.
- *Effetti Collaterali:* Alloca memoria per un nuovo nodo e modifica la struttura della lista.

- **rimuovi_prenotazione_lista(ListaPre l, Prenotazione p)**
  - *Descrizione:* Rimuove dalla lista la prima prenotazione con intervallo temporale uguale a quello della prenotazione fornita.
  - *Input:* `l` - testa corrente della lista, `p` - prenotazione di riferimento per l'intervallo da cercare.
  - *Output:* Puntatore di tipo `ListaPre` alla testa della lista (potenzialmente modificata).
  - *Pre-condizioni:* `p` non deve essere `NULL`, `l` non deve essere `NULL`.
  - *Post-condizioni:* Se trovata, la prenotazione viene rimossa e la memoria liberata.
  - *Effetti Collaterali:* Potrebbe deallocare memoria del nodo e della prenotazione, modifica la struttura della lista.

- **ottieni_prenotazione_lista(ListaPre l)**
  - *Descrizione:* Restituisce la prenotazione contenuta nel nodo corrente.
  - *Input:* `l` - nodo della lista.
  - *Output:* Puntatore di tipo `Prenotazione` alla prenotazione contenuta nel nodo, oppure `NULL` se `l` è `NULL`.
  - *Pre-condizioni:* `l` non deve essere `NULL`.
  - *Post-condizioni:* Restituisce la prenotazione senza modificare la lista.
  - *Effetti Collaterali:* Nessuno.

- **duplica_lista_prenotazioni(ListaPre l)**
  - *Descrizione:* Crea una copia duplicata di una lista di prenotazioni, preservandone l'ordine originale.
  - *Input:* `l` - lista di prenotazioni da duplicare.
  - *Output:* Puntatore di tipo `ListaPre` al duplicato della lista originale.

- *Pre-condizioni:* `l` non deve essere `NULL`.
- *Post-condizioni:* Viene creato un duplicato completo della lista originale.
- *Effetti Collaterali:* Alloca memoria per la nuova lista e i suoi nodi. La memoria deve essere gestita e liberata dal chiamante.

4.5.9 ADT Prenotazioni

L'ADT `Prenotazioni` implementa una struttura dati basata su un albero AVL per la gestione efficiente delle prenotazioni, con supporto a operazioni di inserimento, cancellazione e ricerca.

- **crea_prenotazioni()**
  - *Descrizione:* Crea e inizializza una nuova struttura `Prenotazioni` (un albero AVL vuoto).
  - *Input:* Nessuno.
  - *Output:* Puntatore a `Prenotazioni` se l'allocazione ha successo, `NULL` altrimenti.
  - *Pre-condizioni:* Nessuna.
  - *Post-condizioni:* Un albero vuoto è inizializzato e pronto all'uso.
  - *Effetti Collaterali:* Alloca memoria dinamicamente.
- **distruggi_prenotazioni(Prenotazioni prenotazioni)**
  - *Descrizione:* Dealloca tutta la memoria associata a una struttura `Prenotazioni`.
  - *Input:* `prenotazioni` - struttura da distruggere.
  - *Output:* Nessuno.
  - *Pre-condizioni:* Può essere `NULL`.
  - *Post-condizioni:* Tutta la memoria viene liberata.
  - *Effetti Collaterali:* Deallocazione della memoria.
- **aggiungi_prenotazione(Prenotazioni prenotazioni, Prenotazione p)**
  - *Descrizione:* Aggiunge una prenotazione all'albero.
  - *Input:* `prenotazioni` - albero in cui inserire, `p` - prenotazione da inserire.
  - *Output:* `OK` se l'inserimento ha successo, `OCCUPATO` se si verifica sovrapposizione, 0 in caso di errore.
  - *Pre-condizioni:* `prenotazioni` e `p` non devono essere `NULL`.
  - *Post-condizioni:* La prenotazione viene aggiunta all'albero se non ci sono conflitti.

  - *Effetti Collaterali:* Allocazione memoria, modifica della struttura dell'albero.

- **controlla_prenotazione(Prenotazioni prenotazioni, Intervallo i)**
  - *Descrizione:* Verifica se un intervallo è disponibile.
  - *Input:* `prenotazioni` - albero da interrogare, `i` - intervallo da verificare.
  - *Output:* `OCCUPATO` se l'intervallo è sovrapposto, `OK` altrimenti.
  - *Pre-condizioni:* `prenotazioni` e `i` non devono essere `NULL`.
  - *Post-condizioni:* Nessuna modifica.
  - *Effetti Collaterali:* Nessuno.

- **cancella_prenotazione(Prenotazioni prenotazioni, Intervallo i)**
  - *Descrizione:* Rimuove la prenotazione corrispondente all'intervallo.
  - *Input:* `prenotazioni` - albero da modificare, `i` - intervallo della prenotazione da rimuovere.
  - *Output:* 1 se la cancellazione ha successo, 0 altrimenti.
  - *Pre-condizioni:* `prenotazioni` e `i` non devono essere `NULL`.
  - *Post-condizioni:* La prenotazione viene rimossa se trovata.
  - *Effetti Collaterali:* Deallocazione memoria, modifica della struttura dell'albero.

- **ottieni_vettore_prenotazioni_ordinate(Prenotazioni prenotazioni, unsigned int *size)**
  - *Descrizione:* Estrae le prenotazioni ordinate per data di inizio.
  - *Input:* `prenotazioni` - albero da analizzare, `size` - puntatore a dimensione del risultato.
  - *Output:* Array dinamico di `Prenotazione`, ordinato cronologicamente.
  - *Pre-condizioni:* `prenotazioni` e `size` non devono essere `NULL`.
  - *Post-condizioni:* `*size` contiene il numero di elementi.
  - *Effetti Collaterali:* Alloca memoria da liberare dal chiamante.

- **ottieni_vettore_prenotazioni_per_file(Prenotazioni prenotazioni, unsigned int *size)**
  - *Descrizione:* Estrae le prenotazioni in ordine di livello (BFS).
  - *Input:* `prenotazioni` - albero, `size` - puntatore alla dimensione.
  - *Output:* Array dinamico in ordine di livello.
  - *Pre-condizioni:* `prenotazioni` e `size` non devono essere `NULL`.

- *Post-condizioni:* `*size` contiene la dimensione dell'array.
- *Effetti Collaterali:* Alloca memoria dinamica da gestire esternamente.

- **ottieni_intervallo_disponibile(Prenotazioni prenotazioni, Intervallo i)**
  - *Descrizione:* Verifica la disponibilità di un intervallo e ne restituisce uno valido se presente.
  - *Input:* `prenotazioni` - struttura contenente l'albero, `i` - intervallo da analizzare.
  - *Output:* Puntatore a `Intervallo` disponibile, oppure `NULL` se non trovato.
  - *Pre-condizioni:* `prenotazioni` e `i` devono essere validi.
  - *Post-condizioni:* Se disponibile, restituisce un intervallo compatibile.
  - *Effetti Collaterali:* Possibile allocazione di memoria da gestire dal chiamante.

4.5.10 ADT Tabella Hash

L'ADT `Tabella Hash` implementa una struttura dati di tipo hash table per la memorizzazione efficiente di coppie chiave-valore. Viene utilizzata nel sistema per implementare la **tabella veicoli** e la **tabella utenti**, garantendo accesso rapido ai dati attraverso funzioni di hash.

- **nuova_tabella_hash(const unsigned int grandezza)**
  - *Descrizione:* Crea una nuova tabella hash per la memorizzazione degli oggetti con la dimensione specificata.
  - *Input:* `grandezza` - dimensione iniziale della tabella hash.
  - *Output:* Un puntatore a una nuova struttura `TabellaHash` o `NULL`.
  - *Pre-condizioni:* `grandezza` deve essere maggiore di 0.
  - *Post-condizioni:* Restituisce una nuova `TabellaHash` se l'allocazione è riuscita, altrimenti restituisce `NULL`.
  - *Effetti Collaterali:* Alloca memoria dinamicamente per la tabella hash e i suoi bucket.

- **distruggi_tabella(TabellaHash tabella_hash, void (*funzione_distruggi_valore)(void *))**
  - *Descrizione:* Libera la memoria allocata per una tabella hash, inclusi tutti gli elementi memorizzati.
  - *Input:* `tabella_hash` - puntatore alla tabella hash da eliminare; `funzione_distruggi_valore` - funzione da applicare ad ogni valore per liberare la memoria associata.

- *Output:* Nessun valore di ritorno (`void`).
- *Pre-condizioni:* `tabella_hash` non deve essere `NULL`.
- *Post-condizioni:* Tutta la memoria della tabella hash viene liberata.
- *Effetti Collaterali:* Libera la memoria associata alla struttura `TabellaHash` e ai suoi elementi.

- **inserisci_in_tabella(TabellaHash tabella_hash, const char *chiave, void *valore)**
  - *Descrizione:* Inserisce un elemento nella tabella hash associando una chiave a un valore.
  - *Input:* `tabella_hash` - puntatore alla tabella hash; `chiave` - stringa costante contenente la chiave dell'elemento; `valore` - puntatore al valore da associare alla chiave.
  - *Output:* Un valore di tipo `Byte` (1 oppure 0).
  - *Pre-condizioni:* `tabella_hash`, `chiave` e `valore` non devono essere `NULL`.
  - *Post-condizioni:* Restituisce 1 se l'inserimento è avvenuto con successo, 0 in caso di errore.
  - *Effetti Collaterali:* Modifica la tabella hash aggiungendo un nuovo elemento e ridimensiona la tabella se necessario.

- **cancella_dalla_tabella(const TabellaHash tabella_hash, const char *chiave, void (*funzione_distruggi_valore)(void *))**
  - *Descrizione:* Rimuove un oggetto dalla tabella hash utilizzando la chiave fornita.
  - *Input:* `tabella_hash` - puntatore alla tabella hash; `chiave` - stringa costante contenente la chiave dell'elemento da rimuovere; `funzione_distruggi_valore` - funzione da applicare al valore per liberare la memoria associata.
  - *Output:* Un valore di tipo `Byte` (1 oppure 0).
  - *Pre-condizioni:* `tabella_hash` e `chiave` non devono essere `NULL`.
  - *Post-condizioni:* Restituisce 1 se la rimozione ha avuto successo, 0 se la chiave non è presente o se si verifica un errore.
  - *Effetti Collaterali:* Modifica la tabella hash rimuovendo l'elemento associato alla chiave; libera memoria dinamicamente associata al nodo e al valore.

- **cerca_in_tabella(TabellaHash tabella_hash, const char *chiave)**
  - *Descrizione:* Cerca un elemento nella tabella hash tramite la chiave specificata.
  - *Input:* `tabella_hash` - puntatore alla tabella hash; `chiave` - stringa costante contenente la chiave da cercare.

  - *Output:* Un puntatore generico o `NULL`.
  - *Pre-condizioni:* `tabella_hash` e `chiave` non devono essere `NULL`.
  - *Post-condizioni:* Se la chiave è presente, restituisce il puntatore al valore associato; altrimenti restituisce `NULL`.
  - *Effetti Collaterali:* Nessuno.

- **ottieni_vettore(const TabellaHash tabella_hash, unsigned int *dimensione)**
  - *Descrizione:* Estrae tutti i valori contenuti nella tabella hash e li restituisce in un array.
  - *Input:* `tabella_hash` - puntatore alla tabella hash; `dimensione` - puntatore ad un intero dove verrà memorizzata la dimensione del vettore restituito.
  - *Output:* Un array di puntatori generico o `NULL`.
  - *Pre-condizioni:* `tabella_hash` e `dimensione` non devono essere `NULL`.
  - *Post-condizioni:* Restituisce un array di puntatori ai valori presenti nella tabella hash, oppure `NULL` se ci sono errori o la tabella è vuota.
  - *Effetti Collaterali:* Alloca dinamicamente memoria per il vettore risultante.

5 Razionale dei Casi di Test

5.1 Introduzione

I casi di test sono stati implementati seguendo un approccio strutturato basato su file di input, oracle e output per garantire la verificabilità automatica delle funzionalità del sistema di car sharing. Ogni test case è organizzato in una cartella dedicata contenente tre componenti fondamentali: dati di input, risultato atteso (oracle) e risultato effettivo (output).

5.2 Architettura del Sistema di Test

Il sistema di test utilizza i seguenti file di riferimento:

- **veicoli.txt**: Database dei veicoli disponibili con formato: `Targa;Modello;Posizione;Tariffa;Cat`
- **utenti.txt**: Database degli utenti registrati con formato: `Nome;Cognome;Email;Password`
- **test_suite.txt**: Lista dei test cases da eseguire

5.3 Descrizione Dettagliata dei Test Cases

5.3.1 TC1 - Gestione Prenotazioni e Calcolo Costi

Obiettivo: Verificare la corretta creazione delle prenotazioni, l'aggiornamento della disponibilità dei veicoli e il calcolo accurato dei costi di noleggio.

Input (input.txt):

```
mario.rossi@email.com;AB123CD;10/06/25 09:00;10/06/25 10:30
anna.verdi@email.com;AB123CD;10/06/25 10:00;10/06/25 11:00
luca.bianchi@email.com;AB123CD;10/06/25 13:00;10/06/25 14:00
```

Risultato Atteso (oracle.txt):

```
OK mario.rossi@email.com AB123CD 10/06/25 09:00 10/06/25 10:30 Costo=1.35
OCCUPATO anna.verdi@email.com AB123CD 10/06/25 10:00 10/06/25 11:00
OK luca.bianchi@email.com AB123CD 10/06/25 13:00 10/06/25 14:00 Costo=0.90
```

Razionale: Questo test verifica tre scenari critici:

1. **Prenotazione valida**: Mario prenota il veicolo AB123CD per 1.5 ore (09:00-10:30) con costo corretto di 1.35€
2. **Conflitto temporale**: Anna tenta di prenotare lo stesso veicolo in sovrapposizione (10:00-11:00) e riceve status "OCCUPATO"
3. **Prenotazione successiva**: Luca prenota lo stesso veicolo dopo la liberazione (13:00-14:00) con costo di 0.90€ per 1 ora

Il calcolo dei costi è basato sulla tariffa del veicolo AB123CD (Fiat Panda: 0.015€/minuto) moltiplicata per la durata del noleggio.

5.3.2 TC2 - Visualizzazione Disponibilità Veicoli

Obiettivo: Verificare la corretta visualizzazione dei veicoli disponibili per intervalli temporali specificati.

Input (input.txt):

```
10/06/25 9:00;10/06/25 14:00
10/06/25 10:30;10/06/25 11:00
11/06/25 13:00;11/06/25 14:00
```

Risultato Atteso (oracle.txt):

```
Veicoli disponibili per il 10/06/25 9:00 - 10/06/25 14:00:
'CD456EF' 'ST987UV' 'KL321MN' 'GH789IJ' 'OP654QR'
Veicoli disponibili per il 10/06/25 10:30 - 10/06/25 11:00:
'CD456EF' 'ST987UV' 'KL321MN' 'GH789IJ' 'OP654QR' 'AB123CD'
Veicoli disponibili per il 11/06/25 13:00 - 11/06/25 14:00:
'CD456EF' 'ST987UV' 'KL321MN' 'GH789IJ' 'OP654QR' 'AB123CD'
```

Razionale: Il test verifica tre scenari temporali:

1. **Periodo ampio**: Durante 9:00-14:00 del 10/06, AB123CD non è disponibile a causa delle prenotazioni di Mario e Luca
2. **Finestra intermedia**: Durante 10:30-11:00, AB123CD diventa disponibile tra le due prenotazioni
3. **Giorno successivo**: L'11/06 tutti i veicoli sono disponibili, incluso AB123CD

5.3.3 TC3 - Storico Prenotazioni Utente

Obiettivo: Verificare la corretta visualizzazione dello storico delle prenotazioni per ciascun utente registrato.

Input (input.txt):

```
mario.rossi@email.com
anna.verdi@email.com
luca.bianchi@email.com
```

Risultato Atteso (oracle.txt):

```
Prenotazioni di mario.rossi@email.com:
    - Veicolo: Fiat Panda AB123CD
        - Data: 10/06/25 09:00 -> 10/06/25 10:30
        - Costo: 1.35€
Prenotazioni di anna.verdi@email.com:
Prenotazioni di luca.bianchi@email.com:
    - Veicolo: Fiat Panda AB123CD
        - Data: 10/06/25 13:00 -> 10/06/25 14:00
        - Costo: 0.90€
```

Razionale: Il test verifica:

1. **Storico con prenotazioni**: Mario e Luca mostrano le loro prenotazioni successful con dettagli completi
2. **Storico vuoto**: Anna non ha prenotazioni successful (la sua è stata rifiutata per conflitto)
3. **Completezza informazioni**: Ogni prenotazione include modello veicolo, targa, intervallo temporale e costo

5.4 Metodologia di Esecuzione

5.4.1 Processo di Test Automatizzato

Il sistema di test utilizza uno script eseguibile `./test` che:

1. Legge la lista dei test cases da `test_suite.txt`
2. Per ogni test case, carica l'input dalla cartella corrispondente
3. Esegue la funzionalità e genera l'output
4. Confronta l'output con l'oracle atteso
5. Registra il risultato in `result.txt`

5.4.2 Validazione della Qualità del Codice

Il sistema è stato testato con Valgrind per garantire:

- **Assenza di memory leak**: 152 allocazioni, 152 deallocazioni
- **Gestione corretta della memoria**: 0 bytes in uso all'uscita
- **Assenza di errori**: 0 errori rilevati

Comando utilizzato:

```
valgrind --leak-check=full --show-leak-kinds=all
         --track-origins=yes ./test test_suite.txt utenti.txt veicoli.txt
```

5.5 Copertura dei Requisiti

5.6 Risultati

Tutti i test cases hanno superato la verifica con successo:

Requisito	Test Case
Gestione Prenotazioni	TC1
Calcolo del Costo	TC1
Visualizzazione Disponibilità	TC2
Gestione Noleggi	TC1, TC3
Storico Noleggi	TC3

Tabella 1: Mappatura Requisiti-Test Cases

- **TC1**: HA SUPERATO IL TEST
- **TC2**: HA SUPERATO IL TEST
- **TC3**: HA SUPERATO IL TEST

La suite di test dimostra la correttezza funzionale del sistema e la sua robustezza nella gestione della memoria.