Report

AUTORI: Gianmarco Raimondi, Salvatore Zurino

1. MOTIVAZIONI SCELTE ADT

Clienti, Albero Binario Di Ricerca (BST)

Abbiamo deciso di utilizzare una struttura dati ad *albero binario di ricerca (BST)* per memorizzare i clienti, includendo al suo interno tutte le informazioni personali e relative ai loro abbonamenti.

La scelta del BST è stata il risultato di un'attenta analisi delle esigenze del progetto: ci serviva infatti una struttura in grado di gestire *un numero potenzialmente elevato di dati in continua crescita*, mantenendo al tempo stesso *efficienza nelle operazioni di ricerca, inserimento ed eliminazione*.

Inizialmente abbiamo escluso l'uso di un array dinamico, poiché la gestione della memoria tramite 'realloc' su grandi volumi di dati può diventare inefficiente e complessa. Il nostro dubbio era quindi tra l'uso di liste e alberi. Tuttavia, considerando che le operazioni di ricerca sarebbero state frequenti e dovevano essere eseguite nel minor tempo possibile, *le liste non si sono rivelate una scelta ottimale*, in quanto hanno tempi di accesso lineari.

Al contrario, un *albero binario bilanciato* offre tempi di accesso logaritmici, garantendo migliori prestazioni soprattutto su insiemi di dati di grandi dimensioni.

Lezioni, Array Dinamico

Il catalogo delle lezioni è caratterizzato da *un numero limitato di elementi*, i quali però devono essere *richiamati frequentemente* durante l'esecuzione del programma. Per questa ragione, abbiamo deciso di escludere l'utilizzo di una lista, in quanto il tempo di accesso sequenziale non soddisfa le nostre esigenze in termini di efficienza.

La scelta finale era quindi tra un albero e un array dinamico. Tuttavia, considerando che il numero di lezioni da gestire è ridotto e *tendenzialmente non soggetto a variazioni significative*, abbiamo optato per l'utilizzo di un *array* dinamico.

Per ottimizzare ulteriormente le prestazioni, abbiamo implementato una logica di *reallocazione anticipata*: quando l'array raggiunge la capacità massima, viene riallocato uno spazio superiore rispetto a quello strettamente necessario. Questo approccio consente di ridurre il numero di reallocazioni e mantenere elevata la velocità di accesso, contenendo al minimo lo sforzo computazionale legato alla gestione della memoria.

Prenotazioni, Lista Concatenata

La gestione delle prenotazioni dei clienti è stata progettata considerando la natura dinamica e temporale dei dati coinvolti. Le prenotazioni vengono infatti aggiunte e rimosse frequentemente, poiché ogni giorno possono esserci nuove richieste, modifiche o cancellazioni dovute anche al fatto che le prenotazioni scadute vengono archiviate.

Alla luce di queste esigenze, abbiamo scartato fin da subito l'utilizzo di un albero binario, poiché l'operazione di riordinamento e bilanciamento necessaria dopo ogni inserimento o cancellazione avrebbe reso la struttura troppo complessa da gestire e poco adatta a una sequenza di modifiche così frequente.

Abbiamo quindi preso in considerazione due alternative principali: un array fisso (eventualmente di tipo circolare) e una lista concatenata. Tuttavia, l'array, pur permettendo un accesso diretto rapido, soffre nel momento in cui sono necessarie frequenti rimozioni o inserimenti in posizioni intermedie, richiedendo spostamenti di blocchi di memoria che risultano computazionalmente costosi.

Per questo motivo, la scelta è ricaduta su una lista concatenata, una struttura che si è dimostrata particolarmente efficace per gestire dinamicamente un insieme di elementi che varia nel tempo. Essa consente inserimenti e cancellazioni efficienti, soprattutto se effettuati in testa o in coda, e si adatta perfettamente allo scenario delle prenotazioni, dove la flessibilità e l'agilità nella gestione dei dati sono più importanti dell'accesso diretto e indicizzato.

2. PROGETTAZIONE

Main

Il programma si avvia con la visualizzazione di un menù interattivo che guida l'utente nella selezione delle varie funzionalità disponibili. Per garantire un'adeguata modularità e aderire al principio dell'information hiding, le funzionalità sono state suddivise logicamente in più file sorgente, ciascuno dedicato alla gestione di uno specifico aspetto dell'applicazione.

All'interno del file main.c, oltre alla chiamata alla funzione menu() che stampa il menù principale, sono presenti due funzioni ausiliarie:

- una per pulire il buffer di input (pulisci_input) per gestire correttamente l'interazione con l'utente,
- l’altra per inserire una pausa tra le operazioni (attendi_utente), bloccando temporaneamente l'esecuzione fino alla pressione del tasto INVIO.

La logica di selezione delle opzioni nel menù è gestita tramite una struttura switch, che associa a ciascun valore numerico l’operazione corrispondente.

Tutti i dati inseriti o modificati durante l’esecuzione del programma vengono persistiti su file, così da poter essere recuperati anche nelle sessioni successive.

Abbonamenti

In questo modulo vengono implementate tutte le funzioni relative alla gestione degli abbonamenti. Gli abbonamenti sono memorizzati all’interno della struttura Cliente, che comprende sia i dati anagrafici sia due campi specifici per l’abbonamento:

- la data di inizio dell’abbonamento,
- e la durata in giorni.

Tutti i clienti vengono organizzati in un albero binario di ricerca (BST), ordinato in base al codice fiscale, scelto come chiave univoca.

Questa scelta strutturale consente accessi rapidi, inserimenti ordinati e ricerche efficienti. Sono state inoltre implementate funzioni per la cancellazione di nodi e la gestione del riordinamento dell’albero, oltre alla serializzazione dei dati in formato JSON, per facilitarne il salvataggio e la successiva lettura da file.

Lezioni

La gestione delle lezioni è affidata a un array dinamico, contenuto all’interno di una struttura CatalogoLezioni che ne memorizza anche la dimensione attuale e la capacità allocata.

Sono state implementate tutte le funzioni necessarie per:

- l’inserimento di nuove lezioni,
- la riallocazione automatica della memoria quando necessario,
- la rimozione di lezioni esistenti.

La scelta dell’array dinamico è stata motivata dal numero limitato e stabile di lezioni e dalla necessità di accessi frequenti e veloci agli elementi, rendendo questa struttura più adatta rispetto a liste o alberi.

Lista Prenotazioni

Le prenotazioni sono gestite tramite una lista dinamica, in quanto tale struttura consente di gestire in modo flessibile aggiunte e rimozioni frequenti, che sono tipiche di questo tipo di dati (prenotazioni quotidiane, cancellazioni, aggiornamenti)

È stato inoltre stabilito un limite massimo di prenotazioni per fascia oraria, e sono state sviluppate funzioni booleane per verificare la disponibilità in una determinata fascia, prima di procedere all'aggiunta.

Il modulo include anche una funzione che mostra all'utente quali fasce orarie siano attualmente disponibili e quali invece siano al completo.

Utilities

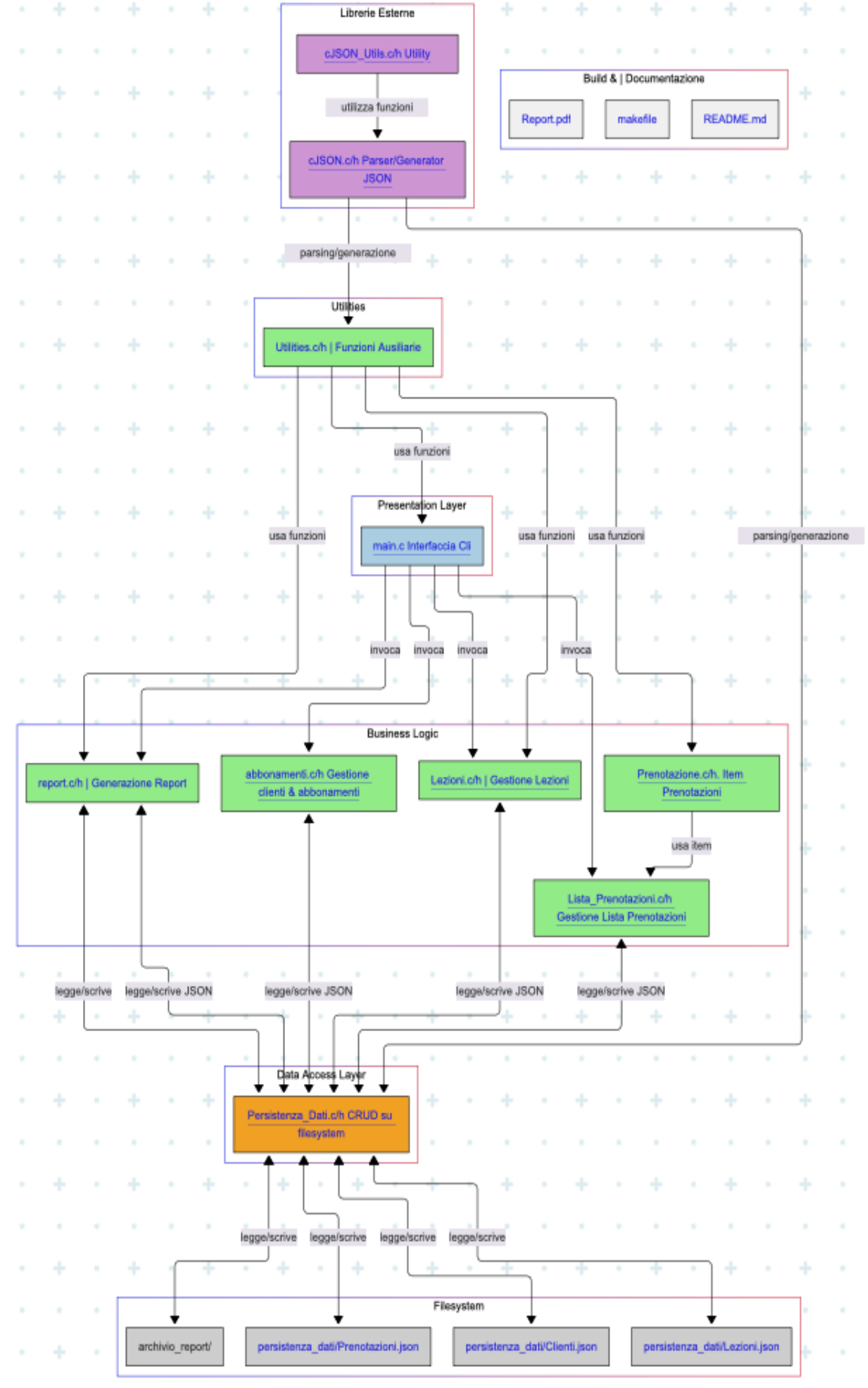
Questo file contiene una raccolta di funzioni generiche e di utilità comune, come:

- la conversione e formattazione dell'orario,
- funzioni di gestione del tempo basate sulla libreria standard time.h,
- e altre operazioni ricorrenti impiegate in più moduli.

Persistenza Dati

Infine, la gestione della persistenza dei dati è stata centralizzata in un apposito modulo. Qui sono implementate tutte le funzioni necessarie per il caricamento dei dati da file all'avvio del programma, il salvataggio automatico o manuale delle strutture aggiornate, assicurando che le informazioni inserite durante l'esecuzione vengano mantenute anche dopo la chiusura dell'applicazione.

Rappresentazione grafica di come i vari moduli interagiscono tra di loro



3. FILE CON SPECIFICHE

main.c

void attendi_utente()

Specifica sintattica:

```
void attendi_utente();
```

Specifica semantica:

- Input
  - Nessun parametro in ingresso.
- Output
  - Nessun valore restituito
- Pre Condizione
  - Nessuna condizione specifica richiesta. La funzione può essere chiamata in qualsiasi momento del flusso del programma.
- Post Condizione
  - Il programma si mette in pausa finché l'utente non preme il tasto INVIO. Dopo l'interazione, l'esecuzione riprende normalmente
- Side Effect
  - Viene stampato un messaggio sul terminale che invita l'utente a premere INVIO, e si attende un input da tastiera (tramite getchar()), che può influenzare temporaneamente il flusso di esecuzione del programma

void pulisci_input()

Specifica sintattica:

```
void pulisci_input();
```

Specifica semantica:

- Input
  - Nessun parametro in ingresso.
- Output
  - Nessun valore restituito
- Pre Condizione
  - La funzione deve essere chiamata dopo una lettura da input che potrebbe lasciare caratteri residui nel buffer, come ad esempio scanf

- Post Condizione
    - Tutti i caratteri residui nel buffer di input standard (stdin), fino al primo newline (\n) o EOF, vengono eliminati
- Side Effect
    - Pulizia del buffer di input standard, il che previene comportamenti indesiderati nelle letture successive da tastiera

void menu()

Specifica sintattica:

```
void menu();
```

Specifica semantica:

- Input
    - Nessun parametro in ingresso.
- Output
    - Nessun valore restituito
- Pre Condizione
    - Nessuna condizione specifica richiesta. La funzione può essere chiamata in qualsiasi momento del flusso del programma.
- Post Condizione
    - Sul terminale viene stampato un menu testuale che mostra tutte le funzionalità disponibili del programma, numerate da 0 a 11, e viene richiesto all'utente di selezionare un'opzione.
- Side Effect
    - Output a schermo tramite printf. L'interfaccia utente viene aggiornata per guidare l'utente nella selezione dell'operazione da compiere.

int main();

Specifica sintattica:

```
int main();
```

Specifica semantica:

- Input
    - Nessuno
- Output
    - Restituisce 0 a fine esecuzione per indicare la terminazione corretta del programma
- Pre Condizione
    - Nessuna
- Post Condizione

    - I file vengono letti (se esistono) e le strutture dati vengono popolate.
    - L'utente può interagire con il sistema tramite il menu per:
        - Inserire un cliente.
        - Visualizzare clienti ordinati.
        - Verificare la validità di un abbonamento.
        - Prenotare una lezione.
        - Visualizzare le prenotazioni.
        - Inserire nuove lezioni.salvato
- Side Effect
    - Lettura/scrittura su file JSON.
    - Allocazione e deallocazione di memoria dinamica.
    - Stampa su schermo.
    - Interazione con l'utente via scanf, fgets e printf.

Prenotazioni.h

Specifica Sintattica:

Operatori definiti sul tipo Prenotazione:

- creaPrenotazione(unsigned int, Lezione, Cliente) → Prenotazione
- getID(Prenotazione) → unsigned int
- getLezione(Prenotazione) → Lezione
- getPartecipante(Prenotazione) → Cliente
- visualizza_prenotazione(Prenotazione) → void

Specifica Semantica:

creaPrenotazione(id, lez, part) = p

- Post: p = (id, lez, part)

getID(p) = id

- Post: p = (id, lez, part)

getLezione(p) = lez

- Post: p = (id, lez, part)

getPartecipante(p) = part

- Post: p = (id, lez, part)

visualizza_prenotazione(p)

- Pre: p è una prenotazione valida
- Post: stampa a schermo i dati contenuti nella prenotazione p
- Side effect: scrive su stdout

prenotazioni.c

void visualizza_prenotazione(Prenotazione prenotazione)

Specifica sintattica:

void visualizza_prenotazione(Prenotazione prenotazione);

Specifica semantica:

- Input
  - Una struttura Prenotazione, che contiene informazioni su:
    - ID della prenotazione,
    - Cliente (nome e cognome),
    - Lezione (nome e data).
- Output
  - Nessuno
- Pre Condizione
  - La struttura Prenotazione passata deve essere correttamente inizializzata, inclusi i campi partecipante, lezione e lezione.data.
- Post Condizione
  - Sul terminale verranno mostrati:
    - ID della prenotazione,
    - Nome e cognome del cliente,
    - Nome della lezione,
    - Data e ora della lezione in formato gg/mm/aaaa--hh:mm. salvato
- Side Effect
  - Output a schermo tramite printf

Abbonamenti.h

Tipo di dato astratto cliente

Specifica Sintattica:

- creaCliente(stringa, stringa, stringa, stringa, intero, time_t, intero) → Cliente
- getNome(Cliente) → stringa
- getCognome(Cliente) → stringa
- getCodiceFiscale(Cliente) → stringa
- getDataNascita(Cliente) → stringa
- getDurata(Cliente) → intero
- getDataInizio(Cliente) → time_t
- getIDAbbonamento(Cliente) → intero
- abbonamento_valido(Cliente) → intero (booleano)

Specifica Semantica:

creaCliente(n, c, cf, dn, dur, inizio, id) = cl

- Post: cl = (n, c, cf, dn, dur, inizio, id)

getNome(cl) = n

- Post: cl = (n, c, cf, dn, dur, inizio, id)

getCognome(cl) = c

- Post: cl = (n, c, cf, dn, dur, inizio, id)

getCodiceFiscale(cl) = cf

- Post: cl = (n, c, cf, dn, dur, inizio, id)

getDataNascita(cl) = dn

- Post: cl = (n, c, cf, dn, dur, inizio, id)

getDurata(cl) = dur

- Post: cl = (n, c, cf, dn, dur, inizio, id)

getDataInizio(cl) = inizio

- Post: cl = (n, c, cf, dn, dur, inizio, id)

getIDAbbonamento(cl) = id

- Post: cl = (n, c, cf, dn, dur, inizio, id)

abbonamento_valido(cl) = valido

- Pre cl contiene data_inizio valida
- Post: restituisce 1 se l'abbonamento non è scaduto, 0 altrimenti

Tipo di dato astratto nodo albero

Specifica Sintattica:

- inserisci_cliente(NodoAlbero, Cliente) → NodoAlbero
- stampa_clienti_ordinati(NodoAlbero) → void
- libera_clienti(NodoAlbero) → void
- ricerca_cliente(NodoAlbero, stringa) → NodoAlbero
- ricerca_e_verifica_cliente(NodoAlbero, stringa) → void
- elimina_cliente(NodoAlbero, stringa) → NodoAlbero

Specifica Semantica:

inserisci_cliente(radice, c) = nuova_radice

- creaCliente(n, c, cf, dn, dur, inizio, id) = cl

stampa_clienti_ordinati(radice)

- Pre: radice può essere NULL o un puntatore valido
- Pre: radice può essere NULL o un puntatore valido

libera_clienti(radice)

- Post: libera tutta la memoria allocata per l'albero
- Post: libera tutta la memoria allocata per l'albero

ricerca_cliente(radice, cf) = nodo

- Post: restituisce il nodo che contiene il cliente con codice fiscale cf, NULL se non trovato

ricerca_e_verifica_cliente(radice, cf)

- Post: stampa a schermo se il cliente cf esiste e se l'abbonamento è valido

elimina_cliente(radice, cf) = nuova_radice

- Post: nuova_radice è l'albero con il cliente cf rimosso, se esiste

abbonamenti.c

Nodo* crea_nodo(Cliente c)

Specifica sintattica:

```
Nodo* crea_nodo(Cliente c);
```

Specifica semantica:

- Input
  - Cliente c – la struttura con i dati del nuovo cliente che dovremmo aggiungere nel nodo che creeremo
- Output
  - Puntatore al nodo appena creato con all' interno i dati del cliente c ed i puntatori ai nodi sx e dx
- Pre Condizione
  - Nessuna
- Post Condizione
  - Viene restituito un puntatore ad un nuovo nodo che viene allocato dinamicamente con i puntatori a dx e sx vuoti ed il cliente nuovo salvato
- Side Effect

- Viene occupato uno spazio di memoria con la chiamata del malloc quindi dovremmo ricordarci di effettuare una free successivamente

Nodo* inserisci_cliente(Nodo* radice, Cliente c)

Specifica sintattica:

Nodo* inserisci_cliente(Nodo* radice, Cliente c);

Specifica semantica:

- Input
  - Nodo* radice: il puntatore alla radice dell' albero che avviamo creato(può essere anche NULL se si tratta del primo inserimento)
  - Cliente nuovo: i dati del cliente che andremo ad inserire nell' albero
- Output
  - Puntatore alla radice aggiornata dell' albero che include anche i dati del nuovo cliente
- Pre Condizione
  - L'albero deve essere binario e si deve basare sull' id per la ricerca
- Post Condizione
  - Se l' id non esiste già viene inserito correttamente rispettando l' ordine dell' albero
  - Se è gia presente, non ci sono modifiche è viene stampato un errore

Int abbonamento_valido(Cliente c);

Specifica sintattica:

Int abbonamento_valido(Cliente c);

Specifica semantica:

- Input
  - Cliente c – la struttura con i dati del cliente
- Output
  - Restituisce 1 se l'abbonamento è ancora valido
  - 0 altrimenti
- Pre Condizione
  - Il tempo di attivazione deve essere stato inizializzato correttamente
- Post Condizione
  - La funzione calcola il tempo trascorso dalla data d'attivazione alla data odierna e lo confronta con la durata dell'abbonamento (confrontando i giorni)
- Side Effect

- Utilizzo della funzione time() per ottenere l'ora corrente
- Utilizzo della funzione difftime() per il calcolo della differenza temporale
- Stampa su stderr in caso di data di inizio futura (errore logico)

Void stampa_clienti_ordinati(Nodo* radice)

Specifica sintattica:

```
void stampa_clienti_ordinati(Nodo* radice);
```

Specifica semantica:

- Input
  - Puntatore alla radice dell' albero binario dove sono salvati i clienti
- Output
  - Nessuno la funzione è void
- Pre Condizione
  - L'albero deve essere già ordinato
- Post Condizione
  - Tutti i clienti salvati nell' albero vengono stampati in ordine crescente di codice fiscale, con i dati riguardanti i loro abbonamenti tramite una funzione ricorsiva
- Side Effect
  - Viene stampato un output a schermo

Void libera_clienti(Nodo* radice)

Specifica sintattica:

```
void libera_clienti(Nodo* radice);
```

Specifica semantica:

- Input
  - Puntatore alla radice dell' albero binario dove sono salvati i clienti
- Output
  - Nessuno la funzione è void
- Pre Condizione
  - L'albero deve essere stato **già** allocato correttamente
- Post Condizione
  - Tutta la memoria allocata per l' albero dei clienti viene liberata
- Side Effect
  - Dopo l' esecuzione il puntatore a radice e tutti i nodi non potreanno più essere usati

NodoAlbero* ricerca_cliente(NodoAlbero* radice, const char* codice_fiscale)

Specifica sintattica:

NodoAlbero* ricerca_cliente(NodoAlbero* radice, const char* codice_fiscale);

Specifica semantica:

- Input
  - Puntatore alla radice dell'albero binario.
  - stringa rappresentante il codice fiscale del cliente da cercare
- Output
  - Restituisce il puntatore al nodo contenente il cliente cercato, oppure NULL se non trovato
- Pre Condizione
  - Nessuna
- Post Condizione
  - Se il cliente è presente nell'albero, viene restituito il nodo corrispondente; altrimenti, NULL
- Side Effect
  - Nessuno.

void ricerca_e_verifica_cliente(NodoAlbero* radice, const char* codice_fiscale)

Specifica sintattica:

void ricerca_e_verifica_cliente(NodoAlbero* radice, const char* codice_fiscale);

Specifica semantica:

- Input
  - puntatore alla radice dell'albero binario.
  - stringa contenente il codice fiscale del cliente da cercare.
- Output
  - Nessun valore restituito; stampa i dati del cliente su stdout.
- Pre Condizione
  - radice deve puntare a un albero valido (può essere anche NULL se vuoto).
- Post Condizione
  - Se il cliente è trovato, stampa i dettagli; altrimenti, stampa un messaggio di errore.
- Side Effect
  - Stampa a video.

static void trova_minimo(NodoAlbero* nodo)

Specifica sintattica:

NodoAlbero* trova_minimo(NodoAlbero* nodo);

Specifica semantica:

- Input
  - puntatore alla radice del sottoalbero.
- Output
  - Restituisce il puntatore al nodo con il valore minimo (più a sinistra).
- Pre Condizione
  - nodo non deve essere NULL
- Post Condizione
  - Viene restituito il nodo con codice fiscale minimo nel sottoalbero
- Side Effect
  - Nessuno

NodoAlbero* elimina_cliente(NodoAlbero* radice, const char* codice_fiscale)

Specifica sintattica:

NodoAlbero* elimina_cliente(NodoAlbero* radice, const char* codice_fiscale);

Specifica semantica:

- Input
  - puntatore alla radice dell'albero binario
  - codice fiscale del cliente da eliminare
- Output
  - Restituisce il nuovo puntatore alla radice dell'albero dopo l'eliminazione
- Pre Condizione
  - L'albero deve essere strutturato come un albero binario di ricerca
- Post Condizione
  - Se esiste un nodo con il codice fiscale specificato, viene eliminato mantenendo la struttura dell'albero
- Side Effect
  - Deallocazione della memoria per il nodo rimosso

lezioni.h

Tipo di dato astratto Lezione

Specifica Sintattica:

- creaLezione(intero, stringa, intero, time_t) → Lezione
- getID(Lezione) → intero
- getNome(Lezione) → stringa
- getNome(Lezione) → stringa
- getNome(Lezione) → stringa

Specifica Semantica:

creaLezione(id, nome, max, data) = lz

- Post: lz = (id, nome, max, data)

getID(lz) = id

- Post: lz = (id, nome, max, data)

getNome(lz) = nome

- Post: lz = (id, nome, max, data)

Post: lz = (id, nome, max, data)

- Post: lz = (id, nome, max, data)

getData(lz) = data

- Post: lz = (id, nome, max, data)

Tipo di dato astratto Catalogo_Lezioni

Specifica Sintattica:

- inizializza_catalogo(Catalogo_Lezioni*) → void
- aggiungi_lezione(Catalogo_Lezioni*, Lezione) → void
- elimina_lezione(Catalogo_Lezioni*, Lezione) → void
- elimina_catalogo(Catalogo_Lezioni*) → void
- mostra_lezioni(Catalogo_Lezioni) → void
- trova_lezione(Catalogo_Lezioni*, intero) → Lezione*
- conflitto_orario_lezione(Catalogo_Lezioni*, time_t) → booleano

Specifica semantica

inizializza_catalogo(c)

- Post: c.numero_lezioni = 0 ∧ c.capacità = CAPACITÀ_INIZIALE ∧ c.lezione è un array allocato di dimensione c.capacità

aggiungi_lezione(c, lz)

- Pre: c è inizializzato
- Post: lz è aggiunta all'array di c; se pieno, l'array è riallocato con capacità maggiore

elimina_lezione(c, lz)

- Post: la lezione con lo stesso ID di lz è rimossa da c, gli elementi successivi sono spostati per mantenere la continuità

elimina_catalogo(c)

- Post: libera la memoria allocata per c.lezione e pone c.numero_lezioni = 0

mostra_lezioni(c)

- Post: stampa a schermo tutte le lezioni presenti in c (id, nome, posti, data)

trova_lezione(c, id) = ptr

- Post: se esiste una lezione in c con ID == id, allora ptr punta a essa; altrimenti ptr = NULL

conflitto_orario_lezione(c, orario) = conf

- Post: conf = true se esiste in c una lezione con data == orario, false altrimenti

lezioni.c

void inizializza_catalogo(Catalogo_Lezioni* catalogo);

Specifica sintattica:

void inizializza_catalogo(Catalogo_Lezioni* catalogo);

Specifica semantica:

- Input
  - Puntatore alla struttura Catalogo_Lezioni.
- Output
  - Nessuno
- Pre Condizione
  - catalogo deve essere un puntatore valido
- Post Condizione
  - La struttura viene inizializzata con:
    - capacità = CAPACITÀ_INIZIALE (es. 4),
    - numero_lezioni = 0,
    - memoria allocata per l'array di lezioni.
- Side Effect
  - Allocazione dinamica di memoria

void aggiungi_lezione(Catalogo_Lezioni* catalogo, const Lezione nuova_lezione);

Specifica sintattica:

void aggiungi_lezione(Catalogo_Lezioni* catalogo, const Lezione nuova_lezione);

Specifica semantica:

- Input
  - Puntatore a Catalogo_Lezioni, struttura Lezione da inserire.
- Output
  - Nessuno
- Pre Condizione
  - catalogo deve essere un puntatore valido e inizializzato
- Post Condizione
  - La lezione viene aggiunta in coda all'array. Se necessario, la memoria viene raddoppiata
- Side Effect
  - Possibile riallocazione di memoria tramite realloc

void elimina_lezione(Catalogo_Lezioni* catalogo, const Lezione lezione_da_eliminare);

Specifica sintattica:

void elimina_lezione(Catalogo_Lezioni* catalogo, const Lezione lezione_da_eliminare);

Specifica semantica:

- Input
  - Puntatore al catalogo e lezione da eliminare (identificata da ID e data)
- Output
  - Nessuno
- Pre Condizione
  - catalogo deve essere inizializzato e deve contenere almeno una lezione
- Post Condizione
  - La lezione viene rimossa, gli elementi successivi vengono shiftati, e se la capacità supera di 4 volte il numero di lezioni, viene ridotta a metà.
- Side Effect
  - Possibile riallocazione di memoria tramite realloc.
  - Stampa di errori se necessario

void elimina_catalogo(Catalogo_Lezioni* catalogo);

Specifica sintattica:

void elimina_catalogo(Catalogo_Lezioni* catalogo);

Specifica semantica:

- Input
  - Puntatore alla struttura Catalogo_Lezioni
- Output
  - Nessuno
- Pre Condizione
  - Il campo lezione deve essere stato allocato
- Post Condizione
  - Memoria deallocata, valori azzerati
- Side Effect
  - free() sulla memoria allocata.

void mostra_lezioni(const Catalogo_Lezioni catalogo);

Specifica sintattica:

void mostra_lezioni(const Catalogo_Lezioni catalogo)**;**

Specifica semantica:

- Input
  - Copia del catalogo
- Output
  - Stampa su schermo
- Pre Condizione
  - Il catalogo deve contenere lezioni
- Post Condizione
  - Ogni lezione viene stampata a video (ID, nome, posti, data/ora). Se vuoto, viene mostrato un errore
- Side Effect
  - Output su stdout con printf

const Lezione* trova_lezione(const Catalogo_Lezioni* catalogo, const unsigned int id);

Specifica sintattica:

const Lezione* trova_lezione(const Catalogo_Lezioni* catalogo, const unsigned int id);

Specifica semantica:

- Input
  - Puntatore al catalogo e ID della lezione da cercare
- Output
  - Puntatore alla lezione trovata o NULL se non esiste
- Pre Condizione
  - Catalogo valido e inizializzato

- Post Condizione
  - Puntatore alla lezione desiderata (se presente)
- Side Effect
  - Nessuno

bool conflitto_orario_lezione(const Catalogo_Lezioni* catalogo, time_t orario)

Specifica sintattica:

bool conflitto_orario_lezione(const Catalogo_Lezioni* catalogo, time_t orario);

Specifica semantica:

- Input
  - Puntatore al catalogo, orario da verificare.
- Output
  - true se esiste già una lezione con lo stesso orario, altrimenti false
- Pre Condizione
  - Catalogo valido e inizializzato
- Post Condizione
  - Valutazione della presenza di conflitto temporale
- Side Effect
  - Nessuno

utilities.h

Tipo di dato astratto Orario

Specifica Sintattica:

Operatori sul tipo Cliente:

- converti_orario_in_struct_tm(time_t) → Orario_Tm*
- converti_orario_in_time_t(Orario_Tm*, intero, intero, intero, intero, intero) → time_t
- genera_id_univoco(stringa) → intero
- file_vuoto(stringa) → booleano

Specifica semantica

converti_orario_in_struct_tm(orario) = tm

- Post: tm è un puntatore a una struttura Orario_Tm che rappresenta orario (espresso come time_t) scomposto nei suoi componenti (anno, mese, giorno, ora, minuto, ecc.)

converti_orario_in_time_t(tm, g, m, a, h, min) = t

- Post: t è un valore time_t ottenuto componendo l'orario a partire dai valori g, m, a, h, min memorizzati in tm

genera_id_univoco(filepath) = id

- Post: id è un intero positivo che rappresenta un identificativo univoco generato a partire dal contenuto del file filepath (es. incremento ultimo ID trovato o file vuoto)

file_vuoto(filepath) = vuoto

- Post: vuoto = true se il file indicato da filepath è vuoto o non contiene dati rilevanti, false altrimenti

utilities.c

Orario_Tm* converti_orario_in_struct_tm(time_t orario)

Specifica sintattica:

Orario_Tm* converti_orario_in_struct_tm(time_t orario)**;**

Specifica semantica:

- Input
  - valore temporale da convertire
- Output
  - Puntatore a una struttura Orario_Tm corrispondente all'orario specificato
- Pre Condizione
  - orario deve essere un valore valido ed accettabile
- Post Condizione
  - Restituisce la struttura compilata correttamente per visualizzazione
- Side Effect
  - La struttura restituita punta a memoria gestita da localtime, quindi verrá deallocata automaticamente

time_t converti_orario_in_time_t(Orario_Tm* tm_orario, int giorno, int mese, int anno, int ora, int minuto)

Specifica sintattica:

time_t converti_orario_in_time_t(Orario_Tm* tm_orario, int giorno, int mese, int anno, int ora, int minuto);

Specifica semantica:

- Input
  - struttura da compilare
  - valore temporale da convertire
- Output
  - Valore corrispondente alla data e ora indicata
- Pre Condizione
  - orario deve essere un valore valido ed accettabile

- Post Condizione
  - Restituisce un null o altri valori coerenti
- Side Effect
  - La struttura tm_orario viene sovrascritta

unsigned int genera_id_univoco(const char filepath)

Specifica sintattica:

unsigned int genera_id_univoco(const char filepath);

Specifica semantica:

- Input
  - percorso al file JSON in cui cercare ID esistenti
- Output
  - ID univoco non presente nel file
- Pre Condizione
  - filepath valido; se il file non esiste, viene considerato vuoto
- Post Condizione
  - Restituito un ID numerico unico rispetto agli altri ID già presenti nel JSON
- Side Effect
  - Allocazione dinamica di memoria temporanea (poi liberata); apertura e lettura del file JSON

bool file_vuoto(char filepath)

Specifica sintattica:

bool file_vuoto(char* filepath);

Specifica semantica:

- Input
  - percorso al file da controllare
- Output
  - restituisce true se il file non è apribile
  - false altrimenti
- Pre Condizione
  - filepath valido (anche se il file non esiste)
- Post Condizione
  - Se il file non esiste o non è accessibile, la funzione restituisce true
  - Se il file esiste, restituisce sempre false, anche se il file è vuoto
- Side Effect
  - Apertura e chiusura del file specificato

Persistenza_dati.h

Tipo di dato astratto Persistenza dati

Specifica Sintattica:

- carica_catalogo_da_file(stringa) → Catalogo_Lezioni*
- carica_prenotazioni_da_file(stringa, Lista_Prenotazioni*) → void
- carica_abbonamenti_da_file(stringa, NodoAlbero**) → void
- salva_lezioni_su_file(Catalogo_Lezioni*, stringa) → booleano
- salva_lezioni_su_file(Catalogo_Lezioni*, stringa) → booleano
- salva_abbonamenti_su_file(NodoAlbero*, stringa) → booleano
- elimina_elem_da_persistenza(stringa, intero) → void

Specifica semantica

carica_catalogo_da_file(fp) = cat

- Post: cat è un puntatore a Catalogo_Lezioni creato leggendo i dati da file JSON nel percorso fp

carica_prenotazioni_da_file(fp, lista)

- Post: lista viene popolata con i dati delle prenotazioni presenti nel file fp

carica_abbonamenti_da_file(fp, radice)

- Post: l'albero binario puntato da *radice viene popolato con i dati dei clienti presenti nel file fp

salva_lezioni_su_file(cat, fp) = stato

- Post: stato = true se i dati del catalogo cat sono stati salvati correttamente nel file fp, false altrimenti

salva_prenotazioni_su_file(lista, fp) = stato

- Post: stato = true se le prenotazioni della lista lista sono state salvate correttamente nel file fp, false altrimenti

salva_abbonamenti_su_file(nodo, fp) = stato

- Post: stato = true se i dati dell'albero binario nodo sono stati salvati nel file fp, false altrimenti

elimina_elem_da_persistenza(tipo, id)

- Post: elimina dal file JSON relativo a tipo l'elemento con identificativo id. tipo ∈ { "cliente", "lezione", "prenotazione" }

Persistenza_dati.c

Catalogo_Lezioni* carica_catalogo_da_file(const char* filepath)

Specifica sintattica:

Catalogo_Lezioni* carica_catalogo_da_file(const char* filepath);

Specifica semantica:

- Input
  - percorso del file JSON contenente il catalogo delle lezioni
- Output
  - Restituisce un puntatore a una struttura Catalogo_Lezioni popolata con i dati letti dal file
  - In caso di errore durante la lettura o il parsing del file, restituisce comunque un catalogo vuoto inizializzato
- Pre Condizione
  - Il parametro 'filepath' deve essere un puntatore non nullo e il file deve essere accessibile.
  - È necessario che la libreria cJSON sia correttamente inclusa e collegata al progetto
- Post Condizione
  - Se il file esiste e contiene un array JSON valido, il catalogo viene correttamente popolato con tutte le lezioni.
  - In caso contrario, viene restituito un catalogo vuoto.
- Side Effect
  - Allocazione dinamica temporanea per la lettura del file, liberata prima del termine.
  - Eventuali errori di lettura vengono stampati su stderr.
  - Uso della funzione inizializza_catalogo() per inizializzare la struttura.
  - Parsing del contenuto JSON tramite cJSON_Parse.
  - Eventuale allocazione temporanea per l'array JSON, liberata con cJSON_Delete

void carica_prenotazioni_da_file(const char* filepath, Lista_Prenotazioni* lista);

Specifica sintattica:

void carica_prenotazioni_da_file(const char* filepath, Lista_Prenotazioni* lista);

Specifica semantica:

- Input
  - percorso del file JSON da cui leggere i dati
  - puntatore a una lista inizializzata delle prenotazioni.
- Output
  - Nessun valore di ritorno. La lista passata viene popolata con i dati letti dal file.
- Pre Condizione
  - 'filepath' e 'lista' devono essere puntatori validi e non nulli. lista deve essere inizializzata a NULL.
- Post Condizione
  - La lista viene popolata con tutte le prenotazioni presenti nel file, se il file è valido.

- Side Effect
    - Allocazioni dinamiche temporanee per la lettura e parsing del file JSON. Eventuali errori di lettura stampati su stderr.

void carica_abbonamenti_da_file(const char* filepath, NodoAlbero radice_BST)**

Specifica sintattica:

void carica_abbonamenti_da_file(const char* filepath, NodoAlbero** radice_BST);

Specifica semantica:

- Input
    - percorso del file JSON contenente gli abbonamenti.
    - doppio puntatore alla radice dell'albero BST da aggiornare.
- Output
    - Nessun valore di ritorno. L'albero viene aggiornato in loco.
- Pre Condizione
    - 'filepath' e 'radice_BST' devono essere validi. L'albero puntato da 'radice_BST' deve essere inizializzato a NULL.
- Post Condizione
    - Tutti gli abbonamenti presenti nel file vengono inseriti nell'albero binario.
- Side Effect
    - Analisi del file JSON con allocazioni dinamiche temporanee.
    - Output su stderr in caso di errore.

bool salva_lezioni_su_file(const Catalogo_Lezioni* catalogo, const char* filepath);

Specifica sintattica:

bool salva_lezioni_su_file(const Catalogo_Lezioni* catalogo, const char* filepath);

Specifica semantica:

- Input
    - puntatore alla struttura contenente le lezioni da salvare.
    - percorso del file JSON di destinazione.
- Output
    - true se il salvataggio è avvenuto correttamente
    - false in caso contrario.
- Pre Condizione
    - catalogo e filepath devono essere puntatori validi.
- Post Condizione
    - Il contenuto del catalogo viene salvato in formato JSON nel file specificato.
- Side Effect

    - Scrittura su file e allocazione temporanea di stringhe JSON.
    - Stampa di errori su stderr.

bool salva_prenotazioni_su_file(const Lista_Prenotazioni lista, const char* filepath);

Specifica sintattica:

bool salva_prenotazioni_su_file(const Lista_Prenotazioni lista, const char* filepath);

Specifica semantica:

- Input
    - puntatore alla struttura contenente le lezioni da salvare.
    - percorso del file JSON di destinazione.
- Output
    - true se il salvataggio è avvenuto correttamente
    - false in caso contrario.
- Pre Condizione
    - catalogo e filepath devono essere puntatori validi.
- Post Condizione
    - Il contenuto del catalogo viene salvato in formato JSON nel file specificato.
- Side Effect
    - Scrittura su file e allocazione temporanea di stringhe JSON.
    - Stampa di errori su stderr.

bool salva_abbonamenti_su_file(const NodoAlbero* nodo, const char* filepath);

Specifica sintattica:

bool salva_abbonamenti_su_file(const NodoAlbero* nodo, const char* filepath);

Specifica semantica:

- Input
    - radice dell'albero binario bilanciato contenente i clienti abbonati.
    - percorso del file JSON di destinazione.
- Output
    - true se il salvataggio è avvenuto correttamente
    - false in caso contrario.
- Pre Condizione
    - nodo e filepath devono essere puntatori validi.
- Post Condizione

    - I dati dell’albero dei clienti vengono salvati in formato JSON nel file specificato.
- Side Effect
    - Scrittura su file e allocazione temporanea di stringhe JSON.
    - Stampa di errori su stderr.

void elimina_elem_da_persistenza(const char* tipo, const unsigned int id);

Specifica sintattica:

void elimina_elem_da_persistenza(const char* tipo, const unsigned int id);

Specifica semantica:

- Input
    - tipo di elemento da rimuovere ("cliente", "lezione", "prenotazione").
    - identificativo numerico dell’elemento da eliminare.
- Output
    - Nessun valore di ritorno.
- Pre Condizione
    - ‘tipo’ deve essere una stringa valida.
    - ‘id’ deve corrispondere a un valore valido.
- Post Condizione
    - Se esiste un elemento nel file corrispondente al tipo e all’ID forniti, esso viene rimosso.
- Side Effect
    - Lettura, modifica e riscrittura del file JSON specificato.
    - Stampa di messaggi informativi o di errore su stdout/stderr.

static void aggiung_clienti_array_json(const NodoAlbero* nodo, cJSON* array_json);

Specifica sintattica:

static void aggiung_clienti_array_json(const NodoAlbero* nodo, cJSON* array_json);

Specifica semantica:

- Input
    - puntatore alla radice o a un sottoalbero dell’albero binario dei clienti.
    - array JSON su cui serializzare i clienti.
- Output
    - Nessun valore di ritorno.
- Pre Condizione
    - array_json deve essere un oggetto JSON valido.
- Post Condizione

    - Tutti i clienti dell'albero vengono aggiunti all'array JSON in ordine simmetrico (in-order).
- Side Effect
    - Allocazione dinamica di oggetti JSON per ogni cliente.

bool salva_report_su_file(const Catalogo_Lezioni* catalogo, const int* conteggi, int num_prenotazioni, const char* path, Orario_Tm* orario)

Specifica sintattica:

bool salva_report_su_file(const Catalogo_Lezioni* catalogo, const int* conteggi, int num_prenotazioni, const char* path, Orario_Tm* orario);

Specifica semantica:

Input

- catalogo: puntatore a una struttura contenente l'elenco delle lezioni disponibili.
- conteggi: array di interi che rappresenta il numero di prenotazioni per ciascuna lezione.
- num_prenotazioni: numero totale di prenotazioni effettuate nel mese.
- path: percorso del file su cui salvare il report in formato JSON.
- orario: puntatore a una struttura Orario_Tm che contiene il mese e l'anno di riferimento.

Output

- Restituisce true se il file viene generato e scritto correttamente, false in caso di errore (es. memoria insufficiente, file non scrivibile, errori di serializzazione JSON).

Pre-condizioni

- Tutti i puntatori (catalogo, conteggi, path, orario) devono essere validi (≠ NULL).
- catalogo->lezione deve contenere catalogo->numero_lezioni lezioni allocate correttamente.

Post-condizioni

- Viene creato un file nel percorso path contenente un oggetto JSON strutturato come segue:
    - anno, mese: anno e mese del report.
    - totale_prenotazioni: numero totale di prenotazioni effettuate.
    - lezioni_piu_frequentate: array contenente le lezioni con la frequenza massima, ciascuna con:
        - id_lezione: identificativo della lezione.
        - nome: nome della lezione.
        - frequenza: numero di partecipazioni.

Side Effect

- Scrittura su disco nel file JSON.
- Allocazione temporanea di memoria dinamica per la costruzione della struttura JSON (rilasciata prima del termine della funzione).
- In caso di errore, stampa nulla ma restituisce false.

bool report_esistente(Orario_Tm *orario)

Specifica sintattica:

bool report_esistente(Orario_Tm *orario);

Specifica semantica:

Input

- orario: puntatore a una struttura Orario_Tm contenente i campi tm_year (anno) e tm_mon (mese), utilizzati per comporre il nome del file del report.

Output

- Restituisce true se esiste un file di report con nome conforme al pattern Report_<anno>_<mese>.json nella directory archivio_report/.
- Restituisce false se il file non è presente.

Pre-condizioni

- Il puntatore orario deve essere valido e inizializzato correttamente.
- I campi tm_year e tm_mon devono contenere valori compatibili con la formattazione della stringa del nome file.

Post-condizioni

- Se esiste un file nel percorso archivio_report/Report_<anno>_<mese>.json, la funzione restituisce true.
- In caso contrario, la funzione restituisce false.

Side Effect

- Tentativo di apertura in lettura ("r") del file di report specifico.
- Nessun effetto collaterale persistente: il file, se trovato, viene semplicemente aperto e immediatamente chiuso.

Lista_Prenotazioni.h

Tipo di dato astratto Lista

Specifica Sintattica:

- crea_lista_prenotazioni() → Lista_Prenotazioni*
- aggiungi_prenotazione(Lista_Prenotazioni*, Prenotazione) → bool
- disdici_prenotazione(Lista_Prenotazioni*, Lezione*) → bool
- visualizza_prenotazioni(const Lista_Prenotazioni*) → void
- libera_lista_prenotazioni(Lista_Prenotazioni*) → void
- conteggia_prenotazioni(const Lista_Prenotazioni*, const Lezione*) → int
- lezione_piena(Lista_Prenotazioni, Lezione) → bool
- controllo_conflitto_orario(Lista_Prenotazioni, Lezione, Cliente) → bool
- trova_prenotazione(Lista_Prenotazioni, Lezione, Cliente) → Prenotazione*
- cliente_prenotato(Lista_Prenotazioni, Cliente) → bool
- lezione_prenotata(Lista_Prenotazioni, Lezione) → bool

Specifica semantica

crea_lista_prenotazioni() = lista

- Post: restituisce un puntatore a una lista di prenotazioni vuota.

aggiungi_prenotazione(lista, prenotazione) = successo

- Post: se successo = true allora prenotazione è stata aggiunta alla lista; altrimenti no

disdici_prenotazione(lista, lezione) = successo

- Post: se successo = true allora una prenotazione associata a lezione è stata rimossa dalla lista.

visualizza_prenotazioni(lista)

- Post: stampa a video tutte le prenotazioni contenute in lista.

libera_lista_prenotazioni(lista)

- Post: libera la memoria occupata dalla lista, rendendo lista vuota.

conteggia_prenotazioni(lista, lezione) = n

- Post: n è il numero di prenotazioni presenti in lista relative a lezione.

lezione_piena(lista, lezione) = bool

- Post: ritorna true se il numero di prenotazioni per lezione ha raggiunto la sua capacità massima, false altrimenti

controllo_conflitto_orario(lista, lezione, partecipante) = bool

- Post: ritorna true se partecipante ha una prenotazione in conflitto con l'orario di lezione, false altrimenti.

trova_prenotazione(lista, lezione, partecipante) = prenotazione*

- Post: ritorna un puntatore alla prenotazione di partecipante per lezione, oppure NULL se non esiste.

cliente_prenotato(lista, partecipante) = bool

- Post: ritorna true se partecipante ha almeno una prenotazione nella lista, false altrimenti.

lezione_prenotata(lista, lezione) = bool

- Post: ritorna true se ci sono prenotazioni per lezione nella lista, false altrimenti.

Lista_Prenotazioni.c

Lista_Prenotazioni* crea_lista_prenotazioni();

Cosa fa:

Crea una nuova lista di prenotazioni vuota.

Specifica sintattica:

Lista_Prenotazioni* crea_lista_prenotazioni();

Specifica semantica:

- Input
  - Nessuno
- Output
  - Puntatore a una lista vuota.
- Pre Condizione
  - Nessuna.
- Post Condizione
  - Viene restituito un puntatore a una lista vuota.
- Side Effect
  - Struttura Lista_Prenotazioni allocata in memoria..

bool aggiungi_prenotazione(Lista_Prenotazioni* lista, const Prenotazione prenotazione);

Specifica sintattica:

bool aggiungi_prenotazione(Lista_Prenotazioni* lista, const Prenotazione prenotazione);

Specifica semantica:

- Input
  - Puntatore alla lista delle prenotazioni, struttura prenotazione da aggiungere.
- Output
  - True se la prenotazione è stata aggiunta, false altrimenti..
- Pre Condizione
  - lista deve essere un puntatore valido.
- Post Condizione
  - La prenotazione viene aggiunta in fondo alla lista se non ci sono conflitti.
- Side Effect
  - Allocazione dinamica di memoria e scrittura su stderr in caso di errore o conflitto.

bool disdici_prenotazione(Lista_Prenotazioni* lista, const Lezione* 1 lezione);

Specifica sintattica:

bool disdici_prenotazione(Lista_Prenotazioni* lista, const Lezione* lezione);

Specifica semantica:

- Input
  - Puntatore alla lista delle prenotazioni, puntatore alla lezione da disdire.
- Output
  - true se la prenotazione è stata trovata e rimossa
  - false altrimenti
- Pre Condizione
  - lista e lezione devono essere puntatori validi.
- Post Condizione
  - La prenotazione corrispondente alla lezione viene rimossa se trovata
- Side Effect
  - Scrittura su stderr e deallocazione della memoria del nodo rimosso

void visualizza_prenotazioni(const Lista_Prenotazioni lista);

Specifica sintattica:

void visualizza_prenotazioni(const Lista_Prenotazioni lista);

Specifica semantica:

- Input
  - Lista delle prenotazioni.
- Output
  - Nessuno.
- Pre Condizione
  - Nessuna.
- Post Condizione
  - Le prenotazioni vengono stampate a video.
- Side Effect
  - Scrittura su stdout o stderr.

void libera_lista_prenotazioni(Lista_Prenotazioni* lista);

Specifica sintattica:

void libera_lista_prenotazioni(Lista_Prenotazioni* lista);

Specifica semantica:

- Input
  - Puntatore alla lista delle prenotazioni.
- Output
  - Nessuno.
- Pre Condizione
  - lista deve essere un puntatore valido.
- Post Condizione
  - Tutti i nodi della lista vengono deallocati.
- Side Effect
  - Deallocazione di memoria.

int conteggia_prenotazioni(const Lista_Prenotazioni* lista, const Lezione* lezione);

Specifica sintattica:

int conteggia_prenotazioni(const Lista_Prenotazioni* lista, const Lezione* lezione);

Specifica semantica:

- Input
  - Puntatore alla lista delle prenotazioni e puntatore alla lezione da cercare.
- Output
  - Numero intero di prenotazioni trovate, -1 in caso di errore
- Pre Condizione
  - lista e lezione devono essere puntatori validi

- Post Condizione
    - Viene restituito il numero di prenotazioni per la lezione specificata
- Side Effect
    - Scrittura su stderr in caso di errore

bool lezione_piena(const Lista_Prenotazioni lista, const Lezione lezione_da_analizzare);

Specifica sintattica:

bool lezione_piena(const Lista_Prenotazioni lista, const Lezione lezione_da_analizzare);

Specifica semantica:

- Input
    - Lista delle prenotazioni e struttura lezione
- Output
    - true se la lezione è piena
    - false altrimenti
- Pre Condizione
    - lista deve essere valido
- Post Condizione
    - Viene restituito il risultato del controllo di capienza
- Side Effect
    - Nessuno (usa conteggia_prenotazioni, che può stampare su stderr)

bool controllo_conflitto_orario(const Lista_Prenotazioni lista, const Lezione lezione, const Cliente partecipante);

Specifica sintattica:

bool controllo_conflitto_orario(const Lista_Prenotazioni lista, const Lezione lezione, const Cliente partecipante);

Specifica semantica:

- Input
    - Lista delle prenotazioni, struttura lezione, struttura cliente
- Output
    - true se esiste un conflitto
    - false altrimenti
- Pre Condizione
    - lista, lezione e partecipante devono essere validi
- Post Condizione
    - Viene restituito se esiste un conflitto di orario
- Side Effect
    - Nessuno

Prenotazione* trova_prenotazione(const Lista_Prenotazioni lista, const Lezione lezione, const Cliente partecipante);

Specifica sintattica:

Prenotazione* trova_prenotazione(const Lista_Prenotazioni lista, const Lezione lezione, const Cliente partecipante);

Specifica semantica:

- Input
  - Lista delle prenotazioni, struttura lezione, struttura cliente
- Output
  - Puntatore alla prenotazione trovata o NULL se non trovata
- Pre Condizione
  - lista, lezione e partecipante devono essere validi
- Post Condizione
  - Viene restituito un puntatore alla prenotazione, se presente
- Side Effect
  - Scrittura su stderr in caso di errore
  - Prenotazione non trovata

bool cliente_prenotato(const Lista_Prenotazioni lista, const Cliente partecipante);

Specifica sintattica:

bool cliente_prenotato(const Lista_Prenotazioni lista, const Cliente partecipante);

Specifica semantica:

- Input
  - una lista allocata contenente prenotazioni (può anche essere vuota)
  - un cliente con un codice fiscale valido
- Output
  - true se esiste almeno una prenotazione in cui il codice fiscale del partecipante corrisponde a quello di una prenotazione nella lista.
  - false altrimenti
- Pre Condizione
  - La lista lista è allocata correttamente.
  - L'oggetto partecipante ha un campo codice_fiscale valido
- Post Condizione
  - Viene restituito true se il cliente è presente almeno una volta come partecipante nella lista delle prenotazioni.
  - Viene restituito false se non è presente alcuna prenotazione legata al cliente
- Side Effect

- Nessuno.

bool lezione_prenotata(const Lista_Prenotazioni lista, const Lezione lezione);

Specifica sintattica:

bool lezione_prenotata(const Lista_Prenotazioni lista, const Lezione lezione);

Specifica semantica:

- Input
  - una lista allocata contenente prenotazioni (può anche essere vuota)
  - una struttura Lezione con campo ID valido da confrontare
- Output
  - true se esiste almeno una prenotazione nella lista è associata alla lezione indicata
  - false altrimenti
- Pre Condizione
  - La lista lista è allocata correttamente.
  - La lezione passata come parametro ha un campo ID valido
- Post Condizione
  - Viene restituito true se è presente almeno una prenotazione associata alla lezione specificatai.
  - Viene restituito false se non vi è alcuna corrispondenza nella lista
- Side Effect
  - Nessuno.

bool codice_fiscale_valido(const char* cf)

Specifica Sintattica:

bool codice_fiscale_valido(const char* cf);

Specifica Semantica:

- Input
  cf: puntatore a una stringa C contenente il codice fiscale da verificare.
- Output
  Restituisce true se il codice fiscale è valido, false altrimenti.
- Precondizione
  cf deve essere un puntatore valido a una stringa terminata da \0.
- Postcondizione
  Viene restituito true solo se:
  - la stringa ha esattamente 16 caratteri;
  - ogni carattere è alfanumerico (A-Z, 0-9) e non minuscolo.
- Side Effect
  Nessuno: la funzione non altera lo stato esterno né modifica l'input.

Report.h

Tipo di dato astratto Report

Specifica Sintattica:

- genera_report_mensile(Prenotazione[], intero) → void

Specifica semantica

genera_report_mensile(prenotazioni, n)

- Post: Viene generato un report mensile analizzando i dati contenuti nell'array prenotazioni di dimensione n.
- Il report viene salvato in un file (presumibilmente in formato testo o JSON) contenente informazioni aggregate sulle prenotazioni effettuate nel mese

Report.c

void genera_report_mensile(Lista_Prenotazioni* lista, Catalogo_Lezioni* catalogo);

Genera un report che viene stampato a schermo con i dati relativi all' ultimo mese di corso

Specifica sintattica:

void genera_report_mensile(Lista_Prenotazioni* lista, Catalogo_Lezioni* catalogo);

Specifica semantica:

- Input
  - puntatore a una struttura Lista_Prenotazioni, contenente tutte le prenotazioni effettuate
  - puntatore a una struttura Catalogo_Lezioni, che rappresenta tutte le lezioni disponibili nel sistema.
- Output
  - Nessun valore restituito
  - Output su terminale e su file esterno
- Pre Condizione
  - Entrambi i puntatori lista e catalogo devono essere correttamente inizializzati (non NULL)
  - Le strutture devono contenere dati validi coerenti con il dominio applicativo
- Post Condizione
  - Vengono stampate a schermo:

        - Il numero totale di prenotazioni.
        - L'elenco delle lezioni disponibili (ID, nome, posti massimi).
        - L'elenco delle prenotazioni effettuate (ID, lezione, nome e cognome del partecipante).
    - Viene generato e salvato un file contenente lo stesso report
- Side Effect
    - Output a schermo tramite printf.
    - Scrittura su file esterno tramite la funzione salva_report_su_file

test_clienti.h

Tipo di dato astratto Report

Specifica Sintattica:

- test_clienti(void) → void

Specifica semantica

- test_clienti()
- • Pre: Nessuna precondizione.
- • Post: Esegue una serie di test automatizzati sulla funzionalità relativa alla gestione dei clienti.
- • In particolare:
- - Testa l'inserimento dei clienti nell'albero (NodoAlbero).
- - Verifica il corretto salvataggio su file degli abbonamenti.
- - Esegue test di ricerca clienti tramite codice fiscale.
- • Confronta l'output effettivo con l'oracolo previsto per validare il comportamento del sistema.
- • I risultati dei test vengono stampati su console con messaggi "ok" o "fail".

Test_clienti.c

static void esegui_test_cliente(int test_num, const char* input_path, const char* esito_path, const char* oracolo_path, NodoAlbero radice, const char* messaggio_successo, const char* messaggio_fallimento)**

Specifica Sintattica:

static void esegui_test_cliente(int test_num, const char* input_path, const char* esito_path, const char* oracolo_path, NodoAlbero** radice, const char* messaggio_successo, const char* messaggio_fallimento);

Specifica Semantica:

- Input:
    - test_num: Intero che identifica il numero del test in esecuzione.
    - input_path: Percorso al file di input contenente i dati del cliente.
    - esito_path: Percorso del file di output dove salvare l'esito effettivo del test.
    - oracolo_path: Percorso del file oracolo contenente l'output atteso.
    - radice: Puntatore alla radice dell'albero binario dei clienti.
    - messaggio_successo: Messaggio da scrivere nel file di esito in caso di test superato.
    - messaggio_fallimento: Messaggio da scrivere nel file in caso di test fallito.
- Output:
    - Nessun valore restituito.
    - L'esito del test viene stampato su console e scritto in un file log.
- Pre-condizioni:
    - input_path, oracolo_path, radice devono essere puntatori validi e non nulli.
    - I file di input e oracolo devono essere accessibili.
- Post-condizioni:
    - I dati del cliente vengono letti dal file input_path, convertiti in struttura Cliente.
    - Se l'abbonamento è valido e il cliente non è già presente, viene tentato l'inserimento nell'albero.
    - L'esito viene scritto su esito_path e confrontato con il contenuto dell'oracolo_path.
    - Viene stampato a video un messaggio "ok" o "fail" in base all'esito.
- Side Effects:
    - Lettura e scrittura da/per file (fopen, fgets, fputs, fclose).
    - Allocazione e deallocazione dinamica di memoria (malloc, free).
    - Stampa su console (printf).
    - Possibile aggiornamento della struttura dati globale (radice).

static void test_cliente_valido(NodoAlbero radice)**

Specifica Sintattica:

static void test_cliente_valido(NodoAlbero** radice);

Specifica Semantica:

- Input:
    - radice: Puntatore alla radice dell'albero binario dei clienti.
- Output:
    - Nessun valore restituito.
    - Stampa a console l'esito del test e scrive un log su file.
- Pre-condizioni:
    - radice deve essere un puntatore valido e non nullo.
- Post-condizioni:
    - Viene eseguito un test di inserimento di un cliente con dati validi, leggendo i dati dal file PATH_INPUT_CLIENTE_VALIDO.
    - L'esito del test viene confrontato con l'oracolo PATH_ORACOLO_CLIENTE_VALIDO.
    - In caso di successo, scrive il messaggio "Cliente registrato correttamente" nel file di esito.
    - In caso di fallimento, scrive il messaggio "Inserimento cliente fallito".

    - Viene stampato a video “ok” o “fail” in base al risultato del confronto.
- Side Effects:
    - Lettura e scrittura da/per file.
    - Output su console.
    - Possibile modifica della struttura dati globale radice con il nuovo cliente inserito.

static void test_cliente_dati_mancanti(NodoAlbero radice)**

Specifica Sintattica:

static void test_cliente_dati_mancanti(NodoAlbero** radice);

Specifica Semantica:

- Input:
    - radice: Puntatore alla radice dell’albero binario dei clienti.
- Output:
    - Nessun valore restituito.
    - Stampa a console l’esito del test e scrive un log su file.
- Pre-condizioni:
    - radice deve essere un puntatore valido e non nullo.
- Post-condizioni:
    - Viene eseguito un test di inserimento cliente con dati incompleti, leggendo i dati dal file PATH_INPUT_CLIENTE_DATI_MANCANTI.
    - L’esito del test viene confrontato con l’oracolo PATH_ORACOLO_CLIENTE_DATI_MANCANTI.
    - In caso di successo (che non è previsto), scrive “Cliente registrato correttamente” nel file di esito.
    - In caso di fallimento (atteso), scrive “Campo mancante nel file input”.
    - Viene stampato a video “ok” o “fail” in base al risultato del confronto.
- Side Effects:
    - Lettura e scrittura da/per file.
    - Output su console.
    - Possibile modifica della struttura dati globale radice se l’inserimento fosse riuscito (ma in questo test non dovrebbe accadere).

static void test_abbonamento_non_valido(NodoAlbero radice)**

Specifica Sintattica:

static void test_abbonamento_non_valido(NodoAlbero** radice);

Specifica Semantica:

- Input:
    - radice: Puntatore alla radice dell’albero binario dei clienti.
- Output:
    - Nessun valore restituito.
    - Stampa a console l’esito del test e scrive un log su file.

- Pre-condizioni:
    - radice deve essere un puntatore valido e non nullo.
- Post-condizioni:
    - Viene eseguito un test di inserimento cliente con abbonamento non valido, leggendo i dati dal file PATH_INPUT_CLIENTE_ABBONAMENTO_NON_VALIDO.
    - L'esito del test viene confrontato con l'oracolo PATH_ORACOLO_CLIENTE_ABBONAMENTO_NON_VALIDO.
    - In caso di inserimento riuscito (non atteso), scrive "Cliente registrato correttamente" nel file di esito.
    - In caso di fallimento (atteso), scrive "Durata abbonamento non valido".
    - Viene stampato a video "ok" o "fail" in base al risultato del confronto.
- Side Effects:
    - Lettura e scrittura da/per file.
    - Output su console.
    - Possibile modifica della struttura dati globale radice solo se il test fallisse (ma non dovrebbe).

static void test_cliente_duplicato(NodoAlbero radice)**

Specifica Sintattica:

static void test_cliente_duplicato(NodoAlbero** radice);

Specifica Semantica:

- Input:
    - radice: Puntatore alla radice dell'albero binario dei clienti.
- Output:
    - Nessun valore restituito.
    - Stampa a console l'esito del test e scrive log su file.
- Pre-condizioni:
    - radice deve essere un puntatore valido e non nullo.
- Post-condizioni:
    - Viene eseguito un test di inserimento cliente con dati che risultano duplicati rispetto a clienti già presenti nell'albero.
    - L'esito viene confrontato con l'oracolo PATH_ORACOLO_CLIENTE_DUPLICATO.
    - In caso di fallimento per duplicato, scrive nel file esito "Cliente già esistente".
    - Se per errore l'inserimento avviene, scrive "Cliente registrato correttamente".
    - Viene stampato a video "ok" o "fail" in base al confronto.
- Side Effects:
    - Lettura e scrittura da/per file.
    - Output su console.
    - Modifica della struttura dati radice solo se il test dovesse erroneamente permettere l'inserimento duplicato.

void avvia_test_clienti(NodoAlbero radice)**

Specifica Sintattica:

void avvia_test_clienti(NodoAlbero** radice);

Specifica Semantica:

- Input:
  - radice: Puntatore alla radice dell'albero binario dei clienti.
- Output:
  - Nessun valore restituito.
  - Stampa a console i risultati di tutti i test relativi alla gestione clienti.
- Pre-condizioni:
  - radice deve essere un puntatore valido e non nullo.
- Post-condizioni:
  - Vengono eseguiti in sequenza i seguenti test:
    - Inserimento di cliente valido.
    - Inserimento di cliente con dati mancanti.
    - Verifica di abbonamento non valido.
    - Inserimento di cliente duplicato.
  - Per ciascun test viene mostrato a video l'esito (ok/fail).
- Side Effects:
  - Output a schermo tramite printf.
  - Eventuale scrittura su file log da parte dei singoli test.

Test_prenotazioni.h

Tipo di dato astratto Report

Specifica Sintattica:

- avvia_test_prenotazioni(Lista_Prenotazioni*, NodoAlbero*, Catalogo_Lezioni*) → void

Specifica semantica

- Precondizioni:
  - lista è un puntatore valido a una struttura Lista_Prenotazioni, contenente le prenotazioni attualmente effettuate.

- radice è un puntatore valido alla radice di un albero binario di ricerca che rappresenta i clienti registrati.
- catalogo è un puntatore valido a una struttura Catalogo_Lezioni, contenente tutte le lezioni disponibili nel sistema.

Test_prenotazionni.c

```
static void esegui_test_prenotazione(
    int test_num,
    const char* input_path,
    const char* esito_path,
    const char* oracolo_path,
    Lista_Prenotazioni* lista,
    NodoAlbero* radice,
    Catalogo_Lezioni* catalogo,
    const char* messaggio_successo,
    const char* messaggio_fallimento
);
```

Specifica sintattica:

```
static void esegui_test_prenotazione(
    int test_num,
    const char* input_path,
    const char* esito_path,
    const char* oracolo_path,
    Lista_Prenotazioni* lista,
    NodoAlbero* radice,
    Catalogo_Lezioni* catalogo,
    const char* messaggio_successo,
```

```
    const char* messaggio_fallimento

);
```

Specifica semantica:

- Input
  - test_num: numero identificativo del test.
  - input_path: percorso del file contenente i dati della prenotazione da testare.
  - esito_path: percorso del file dove verrà salvato l'esito del test.
  - oracolo_path: percorso del file contenente l'output atteso (oracolo).
  - lista: puntatore a una struttura Lista_Prenotazioni già inizializzata, in cui vengono inserite le nuove prenotazioni.
  - radice: puntatore alla radice di un albero contenente i clienti registrati.
  - catalogo: puntatore alla struttura Catalogo_Lezioni, contenente tutte le lezioni disponibili.
  - messaggio_successo: messaggio da scrivere su file in caso di test superato.
  - messaggio_fallimento: messaggio da usare nel log in caso di fallimento.
- Output
  - Nessun valore restituito.
  - Scrittura di log dettagliato nel file esito_path.
  - Stampa su terminale del risultato del test (ok/fail).
  - Verifica automatica tra l'output prodotto (esito_path) e l'oracolo (oracolo_path)
- Pre Condizione
  - Tutti i puntatori devono essere validi e correttamente inizializzati.
  - I file input_path e oracolo_path devono esistere e contenere dati coerenti con il formato previsto.
  - Le funzioni chiamate (ricerca_cliente, trova_lezione, codice_fiscale_valido, aggiungi_prenotazione, ecc.) devono essere correttamente implementate
- Post Condizione
  - Viene tentata l’inserzione della prenotazione secondo i dati forniti nel file input_path.
  -
  - Il file esito_path contiene il risultato dell’operazione, che può includere:
    - Errore nei dati
    - Cliente non trovato
    - Lezione inesistente
    - Lezione piena

        - Prenotazione già esistente
        - Prenotazione registrata correttamente
    - Il risultato ottenuto viene confrontato con l'oracolo.
    - Il test è considerato superato solo se l'output coincide con l'oracolo
- Side Effect
    - Lettura da file (input_path, oracolo_path).
    - Scrittura su file (esito_path).
    - Allocazione e deallocazione dinamica della memoria per i campi della prenotazione.
    - Output su terminale (printf, stampa_ok, stampa_fail)

static void test_prenotazione_id_lezione_non_valido(Lista_Prenotazioni* lista, NodoAlbero* radice, Catalogo_Lezioni* catalogo);

Specifica sintattica:

static void test_prenotazione_id_lezione_non_valido(Lista_Prenotazioni* lista, NodoAlbero* radice, Catalogo_Lezioni* catalogo);

Specifica semantica:

- Input
    - Puntatore a Lista_Prenotazioni, contenente le prenotazioni effettuate finora
    - Puntatore alla radice dell'albero binario di ricerca NodoAlbero, contenente i clienti
    - Puntatore a Catalogo_Lezioni, con l'elenco delle lezioni disponibili
- Output
    - Nessun valore restituito.
    - Output su terminale sotto forma di messaggi ok / fail per ogni test case
- Pre Condizione
    - Tutti i puntatori devono essere correttamente inizializzati e non NULL
    - I file di input, esito e oracolo devono essere presenti e accessibili.
- Post Condizione
    - Il test verifica la corretta gestione di una prenotazione con ID lezione non valido
    - Se l'ID lezione non esiste, il sistema deve scrivere "ID lezione non trovato" nel file di esito
    - Il contenuto del file di esito viene confrontato con l'oracolo. Se combaciano, stampa ok, altrimenti stampa fail
- Side Effect
    - Lettura da file di input
    - Scrittura su file di esito

- Stampa a schermo tramite stampa_ok o stampa_fail

static void test_prenotazione_duplicata(Lista_Prenotazioni* lista, NodoAlbero* radice, Catalogo_Lezioni* catalogo);

Specifica sintattica:

static void test_prenotazione_duplicata(Lista_Prenotazioni* lista, NodoAlbero* radice, Catalogo_Lezioni* catalogo);

Specifica semantica:

- Input
  - Puntatore a Lista_Prenotazioni, contenente le prenotazioni effettuate finora
  - Puntatore alla radice dell'albero binario di ricerca NodoAlbero, contenente i clienti
  - Puntatore a Catalogo_Lezioni, con l'elenco delle lezioni disponibili
- Output
  - Nessun valore restituito.
  - Output su terminale sotto forma di messaggi ok / fail per ogni test case
- Pre Condizione
  - Tutti i puntatori devono essere correttamente inizializzati e non NULL
  - I file di input, esito e oracolo devono essere presenti e accessibili.
- Post Condizione
  - Il test verifica il comportamento del sistema quando si tenta di effettuare una prenotazione già esistente
  - In caso di duplicato, il messaggio atteso è "Prenotazione già registrata"
  - Il risultato del test dipende dal confronto tra il file di esito e il file oracolo.
- Side Effect
  - Lettura da file di input
  - Scrittura su file di esito
  - Stampa a schermo tramite stampa_ok o stampa_file

static void test_prenotazione_codice_fiscale_malformato(Lista_Prenotazioni* lista, NodoAlbero* radice, Catalogo_Lezioni* catalogo);

Specifica sintattica:

static void
test_prenotazione_codice_fiscale_malformato(Lista_Prenotazioni* lista, NodoAlbero* radice, Catalogo_Lezioni* catalogo);

Specifica semantica:

- Input
  - Puntatore a Lista_Prenotazioni, contenente le prenotazioni effettuate finora
  - Puntatore alla radice dell'albero binario di ricerca NodoAlbero, contenente i clienti
  - Puntatore a Catalogo_Lezioni, con l'elenco delle lezioni disponibili
- Output
  - Nessun valore restituito.
  - Output su terminale sotto forma di messaggi ok / fail per ogni test case
- Pre Condizione
  - Tutti i puntatori devono essere correttamente inizializzati e non NULL
  - I file di input, esito e oracolo devono essere presenti e accessibili.
- Post Condizione
  - Il test verifica la gestione di una prenotazione con codice fiscale non valido
  - Se il codice fiscale è malformato, il sistema deve scrivere "Codice fiscale non valido"
  - Il confronto tra file di esito e oracolo determina l'esito del test.
- Side Effect
  - Lettura da file di input
  - Scrittura su file di esito
  - Stampa a schermo tramite stampa_ok o stampa_file.

static void test_prenotazione_lezione_passata(Lista_Prenotazioni* lista, NodoAlbero* radice, Catalogo_Lezioni* catalogo);

Specifica sintattica:

static void test_prenotazione_lezione_passata(Lista_Prenotazioni* lista, NodoAlbero* radice, Catalogo_Lezioni* catalogo);

Specifica semantica:

- Input
  - Puntatore a una struttura Lista_Prenotazioni, contenente tutte le prenotazioni effettuate.
  - Puntatore a una struttura NodoAlbero, radice dell'albero dei clienti registrati.
  - Puntatore a una struttura Catalogo_Lezioni, che rappresenta tutte le lezioni disponibili nel sistema.
- Output

        - Nessun valore restituito.
        - Output su terminale sotto forma di messaggi ok / fail per ogni test case
    - Pre Condizione
        - Tutti i puntatori (lista, radice, catalogo) devono essere inizializzati correttamente e non essere NULL.
        - I file di input (PATH_INPUT_LEZIONE_PASSATA), esito (PATH_ESITO_LEZIONE_PASSATA) e oracolo (PATH_ORACOLO_LEZIONE_PASSATA) devono esistere e contenere dati coerenti con il test.
    - Post Condizione
        - Il test simula una prenotazione per una lezione con data già passata.
        - Il sistema deve rilevare che la lezione non è più disponibile e scrivere "Lezione non disponibile (data scaduta)" nel file di esito.
        - Il file di esito viene confrontato con l'oracolo: se coincidono, il test è superato e viene stampato "ok", altrimenti "fail".
    - Side Effect
        - Lettura da file di input.
        - Scrittura su file di esito.
        - Confronto con file oracolo.
        - Stampa a terminale dell'esito del test

Test_Report.h

Tipo di dato astratto: Test Report

test_report_standard(Catalogo_Lezioni*, Lista_Prenotazioni*) → void

test_report_esistente(Catalogo_Lezioni*, Lista_Prenotazioni*) → void

avvia_test_report(Catalogo_Lezioni*, Lista_Prenotazioni*) → void

Specifica Semantica

- test_report_standard(catalogo, lista)
  Post: Esegue un test sulla generazione di un report mensile standard.
  Scrive l'esito del test su file e stampa "ok" o "fail" a seconda del confronto con l'oracolo.
- test_report_esistente(catalogo, lista)
  Post: Verifica se il sistema rileva correttamente la presenza di un report mensile già generato.
  Se il report esiste, stampa un messaggio coerente e confronta con l'oracolo.
- avvia_test_report(catalogo, lista)
  Post:
    - Avvia una sessione di test stampando a schermo un'intestazione ("— TEST REPORT —").
    - Esegue test_report_standard e test_report_esistente in sequenza.
    - Mostra a console l'esito dei test e genera log nei file di esito indicati.

Test_Report.c

static void esegui_test_report(int test_num, const char* esito_path, const char* oracolo_path, Catalogo_Lezioni* catalogo, Lista_Prenotazioni* lista, const char* messaggio_successo, const char* messaggio_fallimento)

Specifica Sintattica:

static void esegui_test_report(int test_num, const char* esito_path, const char* oracolo_path, Catalogo_Lezioni* catalogo, Lista_Prenotazioni* lista, const char* messaggio_successo, const char* messaggio_fallimento);

Specifica Semantica:

- Input:
  - test_num: Identificativo numerico del test.
  - esito_path: Percorso al file dove viene scritto l'esito del test.
  - oracolo_path: Percorso al file contenente l'output atteso.
  - catalogo: Puntatore al catalogo delle lezioni.
  - lista: Puntatore alla lista delle prenotazioni.
  - messaggio_successo: Messaggio da scrivere in caso di successo.
  - messaggio_fallimento: Messaggio da scrivere in caso di fallimento.
- Output:
  - Nessun valore restituito.
  - Scrive l'esito del test su file.
  - Stampa a console "ok" o "fail" in base al confronto con l'oracolo.
- Pre-condizioni:
  - catalogo e lista devono essere puntatori validi e inizializzati.
  - I file esito_path e oracolo_path devono essere accessibili.
- Post-condizioni:
  - Viene generato il report mensile tramite genera_report_mensile.
  - L'output del report viene scritto nel file esito_path.
  - Il contenuto del file di esito viene confrontato con l'oracolo.
  - Viene stampato a video l'esito del test.
- Side Effects:
  - Scrittura su file.
  - Reindirizzamento temporaneo di stdout e stderr.
  - Stampa a console tramite funzioni stampa_ok o stampa_fail.

static void test_report_standard(Catalogo_Lezioni* catalogo, Lista_Prenotazioni* lista)

Specifica Sintattica:

static void test_report_standard(Catalogo_Lezioni* catalogo, Lista_Prenotazioni* lista);

Specifica Semantica:

- Input:

    - catalogo: Puntatore alla struttura Catalogo_Lezioni contenente tutte le lezioni disponibili.
    - lista: Puntatore alla struttura Lista_Prenotazioni contenente le prenotazioni effettuate.
- Output:
    - Nessun valore restituito.
    - Stampa su console dell'esito del test (OK / FAIL).
    - Scrittura dell'esito effettivo su file log.
- Pre-condizioni:
    - I puntatori catalogo e lista devono essere validi e correttamente inizializzati.
    - I file specificati tramite le costanti PATH_ESITO_REPORT_STANDARD e PATH_ORACOLO_REPORT_STANDARD devono esistere e essere accessibili in lettura/scrittura.
- Post-condizioni:
    - Viene eseguita la funzione esegui_test_report con i parametri predefiniti per testare la generazione standard del report.
    - L'output effettivo del report viene confrontato con l'oracolo.
    - Viene stampato su console un messaggio indicante il successo o il fallimento del test.
- Side Effects:
    - Scrittura su file di log specificato.
    - Possibile generazione o sovrascrittura del file di report JSON.
    - Stampa a console tramite printf, stampa_ok, stampa_fail.

static void test_report_esistente(Catalogo_Lezioni* catalogo, Lista_Prenotazioni* lista)

Specifica Sintattica:

static void test_report_esistente(Catalogo_Lezioni* catalogo, Lista_Prenotazioni* lista);

Specifica Semantica:

- Input:
    - catalogo: Puntatore alla struttura Catalogo_Lezioni contenente tutte le lezioni disponibili.
    - lista: Puntatore alla struttura Lista_Prenotazioni contenente le prenotazioni effettuate.
- Output:
    - Nessun valore restituito.
    - L'esito del test viene stampato su console (OK / FAIL) e salvato in un file di log.
- Pre-condizioni:
    - catalogo e lista devono essere validi e correttamente inizializzati.
    - Il file PATH_ORACOLO_REPORT_ESISTENTE deve contenere l'output atteso.
    - Deve essere già presente un file di report relativo al mese corrente per simulare correttamente il caso di duplicazione.
- Post-condizioni:
    - Il test verifica se il sistema riconosce la presenza di un report già generato per il mese corrente.
    - L'esito effettivo viene salvato su PATH_ESITO_REPORT_ESISTENTE e confrontato con l'oracolo.
    - In caso di corrispondenza, stampa "OK"; altrimenti "FAIL".
- Side Effects:

    - Lettura da file di input e oracolo.
    - Scrittura su file di log.
    - Output a console tramite printf, stampa_ok, stampa_fail.

void avvia_test_report(Catalogo_Lezioni* catalogo, Lista_Prenotazioni* lista)

Specifica Sintattica:

void avvia_test_report(Catalogo_Lezioni* catalogo, Lista_Prenotazioni* lista);

Specifica Semantica:

- Input:
    - catalogo: Puntatore a una struttura Catalogo_Lezioni contenente le lezioni registrate.
    - lista: Puntatore a una struttura Lista_Prenotazioni con tutte le prenotazioni correnti.
- Output:
    - Nessun valore restituito.
    - Viene stampato a schermo l'esito di ogni test eseguito (OK / FAIL).
- Pre-condizioni:
    - catalogo e lista devono essere stati inizializzati e contenere dati coerenti con i test da eseguire.
- Post-condizioni:
    - Esegue in sequenza due test:
        1. test_report_standard → verifica la generazione corretta di un report.
        2. test_report_esistente → verifica il comportamento del sistema quando il report del mese corrente è già presente.
    - I risultati vengono mostrati a terminale e scritti su file (esito vs oracolo).
- Side Effects:
    - Output a console (printf, stampa_ok, stampa_fail).
    - Lettura e scrittura da/per file nei test interni.
    - Eventuale accesso alla cartella dei report per verificare la presenza di file preesistenti.

Test_Utilities.h

Tipo di dato astratto TestUtilities

Specifica Sintattica:

- confronta_output(const char*, const char*) → bool
- scrivi_log(const char*, const char*) → void
- stampa_ok(int) → void
- stampa_fail(int, const char*, const char*) → void
- riga_oracolo_presente(FILE*, int) → bool
- blocco_valido(char**, int) → bool
- duplica_stringa(const char*) → char*

Specifica semantica

- Precondizioni:

- Tutti i puntatori passati come argomenti devono essere validi e non NULL, salvo ove specificato diversamente (es. duplica_stringa accetta NULL).
- I file devono essere accessibili in lettura o scrittura a seconda del contesto.
- I numeri di test devono essere valori interi non negativi.

Test_Utilities.c

bool confronta_output(const char *file_output, const char *file_oracolo)

Specifica sintattica:

bool confronta_output(const char *file_output, const char *file_oracolo);

Specifica semantica:

- Input
  - file_output: percorso del file generato dal programma da confrontare.
  - file_oracolo: percorso del file oracolo contenente l'output atteso.
- Output
  - Restituisce true se i due file sono identici riga per riga (sia nel contenuto che nel numero di righe), false altrimenti.
- Pre Condizione
  - Entrambi i file devono essere accessibili e apribili in lettura.
  - I percorsi file_output e file_oracolo devono essere stringhe valide
- Post Condizione
  - Viene restituito true solo se tutte le righe dei file corrispondono esattamente (esclusi i caratteri di newline finali).
  - In caso contrario viene restituito false, e un messaggio di errore è eventualmente stampato su stderr.
- Side Effect
  - Lettura dei file da disco.
  - Eventuale stampa di messaggi d'errore su stderr

void scrivi_log(const char* path, const char* messaggio)

Specifica sintattica:

void scrivi_log(const char* path, const char* messaggio);

Specifica semantica:

- Input
    - path: percorso del file di log su cui scrivere.
    - messaggio: stringa da scrivere nel file.
- Output
    - Nessun valore restituito.
- Pre Condizione
    - Il parametro path deve essere una stringa valida.
    - messaggio deve essere una stringa non nulla)
- Post Condizione
    - Il file specificato da path viene creato o sovrascritto, contenendo messaggio..
- Side Effect
    - Scrittura su file esterno.
    - In caso di errore, viene stampato un messaggio su stderr

void stampa_ok(int test_num)

Specifica sintattica:

```
void stampa_ok(int test_num);
```

Specifica semantica:

- Input
    - test_num: numero identificativo del test eseguito.
- Output
    - Nessun valore restituito.
    - Output testuale su terminale.
- Pre Condizione
    - test_num deve essere un numero intero non negativo.
- Post Condizione
    - Viene stampato a schermo il messaggio di test superato.
    - Le variabili globali test_ok e test_totali vengono incrementate di 1.
- Side Effect
    - Output a terminale tramite printf.
    - Aggiornamento di variabili globali di stato (test_ok, test_totali).

void stampa_fail(int test_num, const char *atteso, const char *ottenuto)

Specifica sintattica:

```
void stampa_fail(int test_num, const char *atteso, const char *ottenuto);
```

Specifica semantica:

- Input
    - test_num: numero del test fallito.
    - atteso: messaggio o esito atteso.

        - ottenuto: messaggio o esito ottenuto.
- Output
    - Nessun valore restituito.
    - Output su terminale sotto forma di messaggi ok / fail per ogni test case
- Pre Condizione
    - atteso e ottenuto devono essere stringhe valide.
    - test_num deve essere >= 0.
- Post Condizione
    - Viene stampato un messaggio dettagliato con esito atteso e ottenuto.
    - Le variabili globali test_fail e test_totali vengono incrementate di 1.
- Side Effect
    - Output a terminale tramite printf.
    - Aggiornamento di variabili globali (test_fail, test_totali.

void stampa_fail(int test_num, const char *atteso, const char *ottenuto)

Specifica sintattica:

```
void stampa_fail(int test_num, const char *atteso, const char *ottenuto);
```

Specifica semantica:

- Input
    - test_num: numero del test fallito.
    - atteso: messaggio o esito atteso.
    - ottenuto: messaggio o esito ottenuto.
- Output
    - Nessun valore restituito.
    - Output su terminale sotto forma di messaggi ok / fail per ogni test case
- Pre Condizione
    - atteso e ottenuto devono essere stringhe valide.
    - test_num deve essere >= 0.
- Post Condizione
    - Viene stampato un messaggio dettagliato con esito atteso e ottenuto.
    - Le variabili globali test_fail e test_totali vengono incrementate di 1.
- Side Effect
    - Output a terminale tramite printf.
    - Aggiornamento di variabili globali (test_fail, test_totali.

bool riga_oracolo_presente(FILE *oracolo, int test_num)

Specifica sintattica:

```
bool riga_oracolo_presente(FILE *oracolo, int test_num);
```

Specifica semantica:

- Input
    - oracolo: puntatore a un file aperto in lettura, contenente l'output atteso.

    - test_num: numero identificativo del test in esecuzione.
- Output
    - Restituisce true se una riga è presente nel file oracolo.
    - Restituisce false se il file è terminato (EOF) prima del previsto
- Pre Condizione
    - Il puntatore oracolo deve essere valido e riferirsi a un file aperto correttamente.
    - test_num deve essere >= 0.
- Post Condizione
    - Se una riga è presente, la funzione restituisce true.
    - Se il file non contiene una riga da leggere, viene stampato un messaggio d'errore su stderr e restituito false.
- Side Effect
    - Lettura da file.
    - Eventuale stampa su stderr in caso di errore.

bool blocco_valido(char **campi, int numero_campi)

Specifica sintattica:

```
bool blocco_valido(char **campi, int numero_campi);
```

Specifica semantica:

- Input
    - campi: array di puntatori a stringa, rappresentante un blocco di dati testuali da validare.
    - numero_campi: numero di campi previsti all'interno del blocco.
- Output
    - Restituisce true se tutti i campi sono non NULL e non vuoti.
    - Restituisce false altrimenti
- Pre Condizione
    - campi deve essere un array valido di puntatori a stringa.
    - numero_campi deve essere maggiore di 0.
- Post Condizione
    - Se ogni elemento è valido (non NULL e non vuoto), la funzione restituisce true.
    - In caso contrario, stampa un messaggio di errore con l'indice del campo errato e restituisce false.
- Side Effect
    - Eventuale stampa su stderr in caso di errore.

char* duplica_stringa(const char* stringa)

Specifica sintattica:

```
char* duplica_stringa(const char* stringa);
```

Specifica semantica:

- Input
    - stringa: puntatore a una stringa costante da duplicare.
- Output

        - Restituisce un nuovo puntatore a una copia della stringa (malloc), oppure NULL se stringa è NULL o se l'allocazione fallisce
- Pre Condizione
    - stringa può essere NULL, ma in tal caso il comportamento è definito (ritorna NULL).
- Post Condizione
    - Se la duplicazione riesce, viene restituito un puntatore a una copia della stringa, che deve essere deallocata dal chiamante.
    - In caso di errore (es. memoria insufficiente), restituisce NULL.
- Side Effect
    - Allocazione dinamica di memoria (malloc).

Test_main.c

int main(void)

Specifica sintattica:

```
int main(void);
```

Specifica semantica:

- Input
    - Nessun parametro in ingresso.
- Output
    - Ritorna 0 se tutti i test sono stati superati.
    - Ritorna 1 se almeno un test è fallito.
    - Output testuale dettagliato sul terminale con i risultati di ogni fase di test.
- Pre Condizione
    - Devono essere disponibili e correttamente formattati i seguenti file:
        - PATH_FILE_LEZIONI (file contenente il catalogo delle lezioni).
        - PATH_FILE_PRENOTAZIONI (file contenente le prenotazioni).
        - PATH_FILE_ABBONAMENTI (file contenente i dati degli abbonamenti/clienti).
    - Devono essere definite le funzioni di caricamento dati e test (carica_*, avvia_test_*)
- Post Condizione
    - Vengono caricati i dati iniziali necessari per i test.
    - Vengono eseguiti test automatici sulle funzionalità di:
        - Gestione clienti.
        - Gestione prenotazioni.
        - Generazione report.
    - Viene liberata correttamente la memoria occupata dalle strutture dati.
    - Viene stampato un riepilogo finale con il numero totale di test, quelli superati e quelli falliti.
    - Il programma termina restituendo un codice coerente con l'esito della suite.
- Side Effect
    - Output a schermo tramite printf.
    - Lettura da file tramite funzioni carica_*.

- Eventuale stampa di errori o messaggi di stato durante i test.
- Allocazione e deallocazione di strutture dati dinamiche (liste, alberi, cataloghi)

4. RAZIONALE DEI CASI DI TEST

L'attività di testing è stata pianificata per verificare la coerenza funzionale del sistema rispetto ai requisiti individuati durante la fase di analisi e progettazione. Questo documento descrive il razionale, gli obiettivi e le modalità di esecuzione dei test sviluppati per verificare il corretto funzionamento delle funzionalità implementate nel progetto SGP. Ogni sezione è dedicata a un insieme di test relativi a specifici moduli.

1. Test Clienti

I test clienti hanno l'obiettivo di validare le operazioni relative alla gestione degli utenti, tra cui l'inserimento, la validazione dei dati, il controllo di duplicati e la verifica della validità dell'abbonamento.

1. Test cliente valido

Verifica che un cliente con tutti i campi validi venga correttamente inserito nell'albero binario.

1. Caricamento dati da file input.
2. Inserimento cliente tramite esegui_test_cliente.
3. Scrittura su file esito e confronto con file oracolo.
4. Stampa dell'esito su console.

2. Test cliente con dati mancanti

Simula l'inserimento di un cliente con un campo mancante. Il sistema deve segnalare l'errore nel file log.

1. Caricamento dati da file input.
2. Inserimento cliente tramite esegui_test_cliente.
3. Scrittura su file esito e confronto con file oracolo.
4. Stampa dell'esito su console.

3. Test cliente con abbonamento non valido

Verifica che un abbonamento con durata nulla o negativa venga considerato non valido.

1. Caricamento dati da file input.
2. Inserimento cliente tramite esegui_test_cliente.
3. Scrittura su file esito e confronto con file oracolo.
4. Stampa dell'esito su console.

4. Test cliente duplicato
Controlla il comportamento del sistema quando si tenta di inserire un cliente già esistente (stesso codice fiscale).

1. Caricamento dati da file input.
2. Inserimento cliente tramite esegui_test_cliente.
3. Scrittura su file esito e confronto con file oracolo.
4. Stampa dell'esito su console.

2. Test Report

I test sul modulo report hanno lo scopo di verificare la generazione del report mensile e la corretta gestione dei casi in cui un report è già stato prodotto per un determinato mese.

1. Test report standard
Verifica la corretta generazione del report nel caso in cui non esista ancora un file per il mese corrente.

1. Reindirizzamento di stdout e stderr verso il file esito.
2. Chiamata alla funzione genera_report_mensile.
3. Scrittura dell'esito e confronto con l'oracolo.
4. Ripristino output standard e stampa del risultato.

2. Test report esistente
Simula la presenza di un report già salvato e controlla che il sistema non ne generi uno nuovo.

1. Reindirizzamento di stdout e stderr verso il file esito.
2. Chiamata alla funzione genera_report_mensile.
3. Scrittura dell'esito e confronto con l'oracolo.
4. Ripristino output standard e stampa del risultato.

3. Test Prenotazioni

I test relativi alla gestione delle prenotazioni sono stati progettati per verificare la robustezza e l'affidabilità del modulo che consente ai clienti di prenotare lezioni. Il sistema deve gestire correttamente varie condizioni, tra cui prenotazioni corrette, errori di input, conflitti di orario e condizioni limite.

1. Test prenotazione con dati corretti
Verifica che una prenotazione venga correttamente registrata quando tutti i dati sono validi.

1. Viene creato un file di input con un codice fiscale valido e un ID lezione esistente.
2. Il sistema tenta l'inserimento della prenotazione nella lista.
3. L'esito atteso è "Prenotazione registrata correttamente".
4. L'output viene confrontato con l'oracolo per verificarne la correttezza.

2. Test prenotazione con cliente inesistente
Scopo del test è verificare che il sistema rifiuti una prenotazione associata a un codice fiscale non presente nell'albero dei clienti.

1. Si fornisce un file di input con un codice fiscale inesistente.
2. Il sistema ricerca il cliente nell'albero e tenta la prenotazione.
3. L'esito atteso è "Cliente non trovato".
4. L'output viene confrontato con l'oracolo.

3. Test prenotazione con lezione piena
Scopo del test è verificare che non sia possibile prenotare una lezione che ha raggiunto la capienza massima.

1. Viene preparata una lezione già prenotata da un numero massimo di clienti.
2. Il test tenta di aggiungere un'ulteriore prenotazione.
3. L'esito atteso è "Lezione piena".
4. L'output prodotto viene confrontato con l'oracolo.

4. Test prenotazione con orari sovrapposti
Scopo del test è accertarsi che un cliente non possa prenotarsi a più lezioni con lo stesso orario.

1. Si inserisce nel sistema una prenotazione già esistente per un certo orario.
2. Si tenta di aggiungerne una seconda con orario coincidente.
3. L'esito atteso è "Conflitto orario con altra prenotazione".
4. L'output viene confrontato con l'oracolo.

5. Test prenotazione con ID Lezione non valido
Scopo del test è verificare che il sistema identifichi correttamente una richiesta di prenotazione associata a una lezione inesistente.

1. Viene creato un file input con un ID lezione non presente nel catalogo.
2. Il test tenta l'inserimento della prenotazione.
3. L'esito atteso è un messaggio "ID lezione non trovato".
4. L'output viene confrontato con l'oracolo per verificarne la correttezza.

6. Test prenotazione Duplicata
Verifica che il sistema blocchi le prenotazioni duplicate dello stesso cliente alla stessa lezione.

1. Una prenotazione valida viene inserita.

2. Viene eseguita nuovamente una prenotazione con gli stessi dati.
3. L'esito atteso è "Prenotazione già registrata".
4. L'output viene confrontato con l'oracolo per verificarne la correttezza.

7. Test prenotazione con Codice Fiscale Malformato
Controlla che un codice fiscale non conforme venga rilevato come errore sintattico.

1. Il file di input contiene un codice fiscale troppo corto o con caratteri non validi.
2. Il sistema deve rispondere con un messaggio "Codice fiscale non valido".
3. Il risultato viene confrontato con l'oracolo.

8. Test prenotazione per Lezione Passata
Verifica che non sia possibile effettuare prenotazioni per lezioni con data già trascorsa.

1. Il file input descrive una lezione la cui data è nel passato rispetto alla data corrente.
2. Il sistema risponde con "Lezione non disponibile (data scaduta)".
3. Il risultato viene confrontato con l'oracolo.

Note finali

Tutti i casi di test sono stati eseguiti manualmente tramite interazione con l'interfaccia testuale del programma. Durante il test è stato utilizzato il debugger gdb per l'analisi di eventuali segmentazioni della memoria. I dati di test sono stati creati ad hoc e validati con confronti diretti nel file system.

CONCLUSIONI

Il programma è stato progettato con una forte attenzione alla modularità, alla chiarezza e alla manutenibilità del codice, rispettando i principi fondamentali della programmazione strutturata. La scelta delle strutture dati è stata effettuata in base alle caratteristiche specifiche di ciascun insieme di informazioni, con particolare attenzione alle operazioni più frequenti e ai vincoli funzionali del sistema.