**Report**

# **Motivazione scelte ADT**

## Clienti, Albero Binario Ordinato (BTS)

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

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.

# **Progettazione**

## Main

Il programma si apre 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.c

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.c

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.

## Prenotazioni.c

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.c

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.c

Infine, la gestione della persistenza dei dati è stata centralizzata in un apposito modulo. Qui sono implementate tutte le funzioni necessarie per:

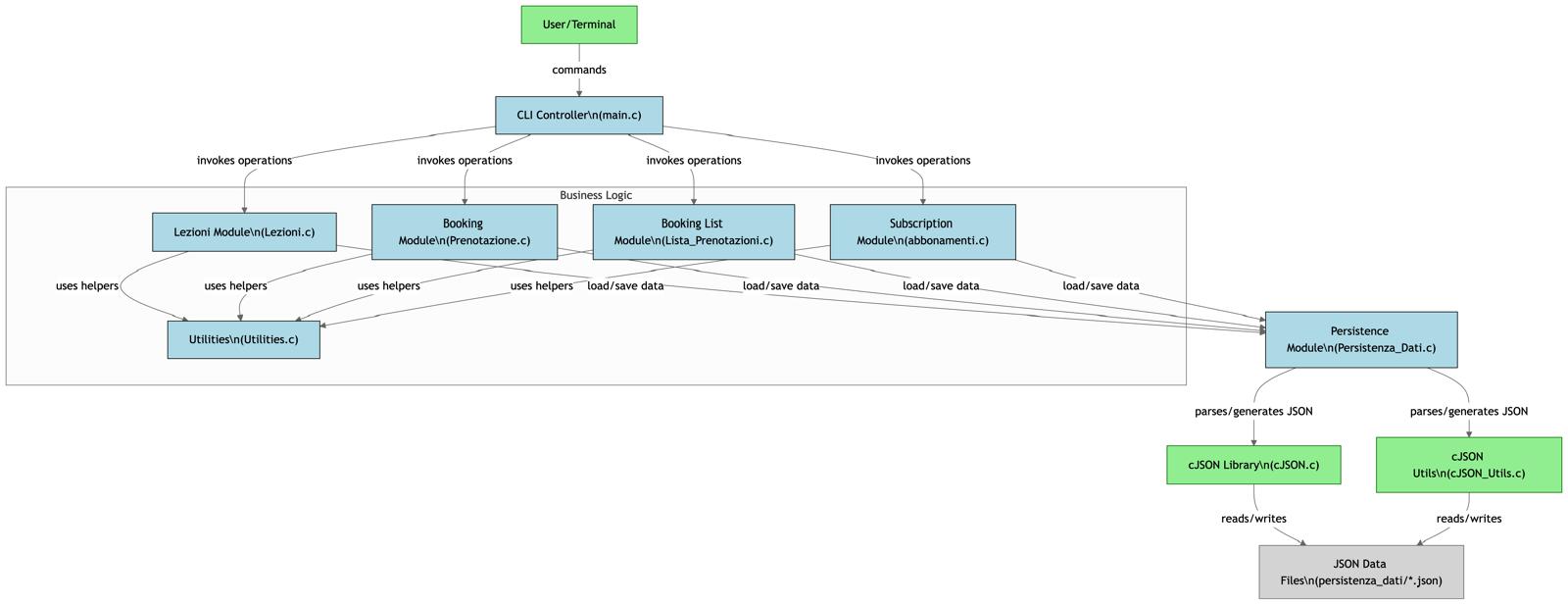
* il caricamento dei dati da file all’avvio del programma,

e il salvataggio automatico o manuale delle strutture aggiornate, assicurando che le informazioni inserite durante l’esecuzione vengano mantenute anche dopo la chiusura dell'applicazione.

## Conclusione

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.

## Rappresentazione grafica di come i vari moduli funzionano tra di loro



# **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.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.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
* Side Effect
  + Potrebbero essere salvati dei duplicati

**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 mdata 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 ssalvati 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 gia 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

**ricerca\_cliente**

Ricerca un cliente nell'albero binario di ricerca utilizzando il codice fiscale come chiave.

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)**

Cerca un cliente nell'albero binario e ne stampa i dettagli, compresa la validità dell'abbonamento

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.

**trova\_minimo**

Trova il nodo con il valore minimo (in base al codice fiscale) in un sottoalbero binario

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

**elimina\_cliente**

Elimina un cliente dall'albero binario di ricerca in base al 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.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.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.c

**carica\_catalogo\_da\_file(const char\* filepath)**

Specifica sintattica:

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:

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.

## 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
  + Nessuno.

**void aggiungi\_prenotazione(Lista\_Prenotazioni\* lista, const Prenotazione prenotazione);**

Aggiunge una prenotazione alla lista, verificando disponibilità e conflitti.

Specifica sintattica:

void aggiungi\_prenotazione(Lista\_Prenotazioni\* lista, const Prenotazione prenotazione);

Specifica semantica:

* Input
  + Puntatore alla lista delle prenotazioni, struttura prenotazione da aggiungere.
* Output
  + Nessuno.
* 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);**

Rimuove una prenotazione da una lista, cercandola per ID e data 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);**

Stampa a video tutte le prenotazioni presenti nella 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);**

Libera tutta la memoria allocata per la lista delle prenotazioni.

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);**

Conta il numero di prenotazioni per una lezione specifica.

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);**

Verifica se una lezione ha raggiunto la capienza massima

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);**

Controlla se un partecipante ha già una prenotazione in conflitto orario

Specifica sintattica:

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);**

Cerca una prenotazione tramite data, ID, nome e cognome

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);**

Cosa fa:

Controlla se il cliente è prenotato.

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);**

Cosa fa:

Controlla se la lezione è prenotata o no.

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.

## 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
  + lEntrambi 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

# **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. Di seguito vengono descritti i principali casi di test selezionati, con i relativi obiettivi, condizioni di partenza, procedura di esecuzione e risultati attesi.

## Test di Registrazione delle Prenotazioni e Aggiornamento Disponibilità

Obiettivo:

* Verificare che la registrazione di una prenotazione aggiorni correttamente la lista delle prenotazioni e lo stato della disponibilità della lezione

Condizioni iniziali:

* Una lezione disponibile con max\_posti=10
* Nessuna prenotazione inizialmente attiva
* Cliente con abbonamento valido

Procedura:

1. Creare una prenotazione per il cliente per la lezione disponibile
2. Verificare l’ inserimento all’ interno della lista prenotazioni
3. Verificare che il numero di posti rimanenti sia aggiornato

Risultato atteso:

* La prenotazione è presente nella lista
* Il file di persistenza Prenotazioni.json è aggiornato
* Il numero di posti disponibili nella lezione è decrementato

## Test di Gestione Abbonamenti e Verifica Validità

Obiettivo:

* Verificare che il sistema riconosca correttamente lo stato di validità di un abbonamento e impedisca operazioni non consentite

Condizioni iniziali:

* Cliente A: abbonamento valido (data\_inizio = oggi, durata = 30)
* Cliente B: abbonamento scaduto (data\_inizio = 60 giorni fa, durata = 30)

Procedura:

1. Visualizzare l’elenco clienti con stampa\_clienti\_ordinati.
2. Provare a effettuare una prenotazione per entrambi i clienti

Risultato atteso:

* Il cliente A è indicato come “Valido: SI” e può prenotare.
* Il cliente B è indicato come “Valido: NO” e la prenotazione viene rifiutata

## Test di Correttezza del Report Prenotazioni

Obiettivo:

* Assicurarsi che il report delle prenotazioni fornisca informazioni complete, leggibili e coerenti con i dati memorizzati

Condizioni iniziali:

* Presenza di almeno una prenotazione registrata e persistita

Procedura:

1. Richiamare la visualizzazione delle prenotazioni dal menu principale.
2. Verificare le seguenti informazioni: nome e cognome del partecipante, codice fiscale, ID lezione, data, ID prenotazione, stato abbonamento

Risultato atteso:

* Tutti i campi vengono visualizzati correttamente.
* Nessun campo risulta vuoto o danneggiato (es. data\_nascita in formato illegibile).
* Le informazioni sono coerenti con quanto salvato in JSON.

## 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.