# Report sistema di gestione prenotazioni palestra

Link alla repository git-hub: https://github.com/BERNY-04/Progetto-Palestra

## 1. Motivazione della Scelta dell'ADT

L’ ADT principale scelto per questo progetto è la lista concatenata, implementata in C attraverso strutture collegate da puntatori. Abbiamo scelto questo ADT per:

la sua flessibilità, poiché permette la gestione di un numero dinamico di elementi;

la gestione della memoria, visto che con un’adeguata allocazione della memoria si evitano sprechi di memoria;

## 2. Progettazione del Sistema

Il sistema è strutturato in quattro componenti principali, ciascuno gestito da una lista concatenata. Abbiamo le strutture:

Clienti:

* Lista concatenata;
* Serve ad aggiungere i clienti, ogn’uno con i propri dati;
* Interagisce con Prenotazioni per verificare gli abbonamenti;

Lezioni:

* Lista concatenata;
* Serve ad aggiungere le lezioni, ogn’una con i propri dati;
* Collegata a Prenotazioni così da poter aggiornare i posti disponibili in ogni lezione;

Prenotazioni:

* Lista concatenata;
* Serve per permettere ai clienti di prenotare le proprie lezioni;
* Collegata a Clienti e Lezioni per i motivi precedentemente citati;

Data:

* Struttura ausiliaria;
* Permette di salvare le date relative alle Lezioni;

## 3. Specifica Sintattica e Semantica

Modulo: ***Clienti (`clienti.h/c`)***

* ***Funzione inizializzaClienti()***
  + SPECIFICA SINTATTICA: void inizializzaClienti()
  + TIPI: void
  + SPECIFICA SEMANTICA: inizializzaClienti() -> void
  + PRE-CONDIZIONI:
    - - Deve esistere il puntatore testa\_clienti che punta alla lista clienti
    - - Deve esistere la funzione liberaClienti() per la deallocazione della memoria
  + POST-CONDIZIONI:
    - - La lista dei clienti viene impostata a Null
    - - Il contatore degli Id viene resettato a 1
    - - Tutta la memoria precedentemente allocata viene liberata
  + REALIZZAZIONE:
    - 1 Chiama liberaClienti() per deallocare la memoria dedicata ad ogni cliente
    - 2 Resetta il puntatore alla lista dei clienti
    - 3 Resetta il contatore degli Id
  + CODICE:

void inizializzaClienti() {

liberaClienti();

testa\_clienti = NULL;

prossimo\_id = 1;

}

* ***Funzione aggiungiCliente()***
  + SPECIFICA SINTATTICA: int aggiungiCliente(const char\* nome, const char\* cognome, int giorni\_abbonamento)
  + TIPI: int, char
  + SPECIFICA SEMANTICA: aggiungiCliente(nome, cognome, giorni\_abbonamento) -> int
  + Pre-condizioni:
    - - Nome e Cognome devono essere stringhe valide;
    - - Nome e Cognome devono rientrare nella dimensione prestabilita;
    - - Testa\_clienti deve essere un puntatore valido, inizializzato a NUll se la lista è vuota;
  + Post-condizioni:
    - - In caso che la funzione riscontri successo un nuovo cliente viene aggiunto alla testa della lista, con il proprio Id, con la copia del nome e del cognome dati in input e con i propri giorni d'abbonamento;
    - - In caso di fallimento lal ista rimane invariata
  + REALIZZAZIONE:
    - 1 Controlla se ci sono errori inerenti alla memoria
    - 2 Aggiunge un nuovo cliente a una lista collegata di clienti.
    - 3 Alloca spazio per ogni cliente ed ad ogni cliente viene attribuito:
      * - Id univoco;
      * - Nome;
      * - Cognome;
      * - Giorni d'abbonamento;
    - 4 Aggiunge il cliente alla lista, con i propri dati e stampa un messaggio per confermare
  + CODICE:

int aggiungiCliente(const char\* nome, const char\* cognome, int giorni\_abbonamento) {

if (giorni\_abbonamento <= 0) {

printf("Errore: durata abbonamento non valida.\n");

return -1;

}

Cliente\* nuovo = malloc(sizeof(Cliente));

if (!nuovo) {

printf("Errore di memoria.\n");

return -1;

}

nuovo->id = prossimo\_id++;

strncpy(nuovo->nome, nome, sizeof(nuovo->nome));

strncpy(nuovo->cognome, cognome, sizeof(nuovo->cognome));

nuovo->giorni\_abbonamento = giorni\_abbonamento;

nuovo->next = testa\_clienti;

testa\_clienti = nuovo;

printf("Cliente aggiunto con ID %d.\n", nuovo->id);

return nuovo->id;

}

***Funzione verificaAbbonamenti()***

* SPECIFICA SINTATTICA: void verificaAbbonamenti()
* TIPI: void
* SPECIFICA SEMANTICA: verificaAbbonamenti() -> void
* Pre-condizioni;
  + - La lista dei clienti deve essere correttamente inizializzata;
  + - I campi dei clienti devono essere validi.
* Post-condizioni:
  + - Se la lista è vuota stampa un errore e non modifica la lista;
  + - Se la lista contiene dei clienti ne stampa i dati e non modifica la lista;
* REALIZZAZIONE:
  + 1 Verifica se all'interno della lista esiste almeno un cliente, se la lista è vuota stampa un errore
  + 2 Scorre la lista stampando rispettivamente ogni cliente con i propri dati, cioè:
  + - Nome;
  + - Cognome;
  + - Giorni d'abbonamento rimanenti;
  + Per poi passare al prossimo cliente.
* CODICE:

void verificaAbbonamenti() {

printf("\n--- Stato Abbonamenti ---\n");

Cliente\* curr = testa\_clienti;

if (!curr) {

printf("Nessun cliente registrato.\n");

return;

}

while (curr) {

printf("\nCliente %d: %s %s | Giorni restanti: %d\n",

curr->id, curr->nome, curr->cognome, curr->giorni\_abbonamento);

curr = curr->next;

}

}

***Funzione verificaValiditaAbbonamento()***

* SPECIFICA SINTATTICA: int verificaValiditaAbbonamento(int id\_cliente)
* TIPI: int
* SPECIFICA SEMANTICA: verificaValiditaAbbonamento(id\_cliente) -> int
* Pre-condizioni:
  + - Id-cliente deve essere un intero valido;
  + - la lista dei clienti può essere vuota, se non è Null ogni next deve puntare al cliente successivo.
* Post- condizioni:
  + - Restituisce 1 se il cliente esiste ed il suo abbonamento è valido;
  + - Restituisce 0 se il cliente non esiste oppure il cliente esiste ma il suo abbonamento è scaduto;
  + - Non modifica ne la lista ne i dati del cliente.
* REALIZZAZIONE:
  + Verifica se l'abbonamento di un cliente è ancora valido cercandolo tramite l'id cliente.
  + 1 Scorre la lista dei clienti e restituisce:
  + - 1 se giorni\_abbonamenti > 0;
  + - 0 se giorni\_abbonamento <=0;
* CODICE:

int verificaValiditaAbbonamento(int id\_cliente) {

Cliente\* curr = testa\_clienti;

while (curr) {

if (curr->id == id\_cliente)

return curr->giorni\_abbonamento > 0;

curr = curr->next;

}

return 0;

}

***Funzione stampaListaClienti()***

* SPECIFICA SINTATTICA: void stampaListaClienti()
* TIPI: void
* SPECIFICA SEMANTICA: stampaListaClienti() -> void
* Pre-condizioni:
  + -testa\_clienti deve essere un puntatore valido;
  + - I campi dei clienti devono essere validi.
* Post-condizioni:
  + - Se la lista è vuota stampa un messaggio e non modifica la lista;
  + - Se la lista contiene dei clienti ne stampa i dati e non modifica la lista.
* REALIZZAZIONE
  + 1 Verifica se la lista è vuota, se lo è stampa un messaggio d'errore
  + 2 Scorre tutta la lista dei clienti e stampa una lista formattata di tutti i clienti registrati, mostrando per ciascun cliente:
  + - Id;
  + - Nome;
  + - Cognome;ù
* CODICE:

void stampaListaClienti() {

Cliente\* corrente = testa\_clienti;

printf("\n--- LISTA COMPLETA CLIENTI ---\n");

if (!corrente) {

printf("Nessun cliente registrato.\n");

return;

}

printf("%-5s %-20s %-20s\n", "ID", "NOME", "COGNOME");

printf("----------------------------------------\n");

while (corrente) {

printf("%-5d %-20s %-20s\n",

corrente->id, corrente->nome, corrente->cognome);

corrente = corrente->next;

}

}

***Funzione liberaClienti()***

* SPECIFICA SINTATTICA: void liberaClienti()
* TIPI: void
* SPECIFICA SEMANTICA: liberaCLieenti() -> void
* Pre-condizioni:
  + - testa\_clienti deve essere un puntatore valido;
* Post-condizioni:
  + - Tutta la memoria inerente ai nodi è stata deallocata;
  + - testa\_clienti è impostata a Null;
  + - la funzione non restituisce nessun valore ma modifica lo stato della lista.
* REALIZZAZIONE:
  + 1 Parte dal puntatore testa\_clienti, usando un ciclo while scorre tutta la lista.
  + 2 Libera tutta la memoria allocata per la lista dei clienti e imposta testa\_clienti a Null per indicare che la lista è vuota.
* CODICE:

void liberaClienti() {

Cliente\* curr = testa\_clienti;

while (curr) {

Cliente\* temp = curr;

curr = curr->next;

free(temp);

}

testa\_clienti = NULL;

}

***Modulo: Lezioni (`lezioni.h/c`)***

***Funzione inizializzaLezioni()***

* SPECIFICA SINTATTICA: void inizializzaLezioni()
* TIPI: Void.
* SPECIFICA SEMANTICA: inizializzaLezioni()void;
* PRECONDIZIONI:
  + - Esiste la procedura “liberaLezione()”;
  + - Esistono e sono dichiarati i dati “testa\_lezioni” e “prossimo\_id\_lezione”.
* POSTCONDIZIONI: Vengono inizializzati i dati “testa\_lezioni” e “prossimo\_id\_lezione”.
* REALIZZAZIONE
  + 1 Richiamiamo la funzione “libera\_Lezioni()”;
  + 2 Inizializziamo a NULL il dato “testa\_lezioni”;
  + 3 Inizializziamo a 1 il dato “prossimo\_id\_lezione”;
* CODICE

void inizializzaLezioni() {

liberaLezioni();

testa\_lezioni = NULL;

prossimo\_id\_lezione = 1;

}

***Funzione aggiungiLezione()***

* SPECIFICA SINTATTICA: int aggiungiLezione(const char\* nome, const char\* orario, int max\_partecipanti, date data)
* TIPI: int, char, date.
* SPECIFICA SEMANTICA: aggiungiLezione(nome, orario, max\_partecipanti, data) -> void
* PRECONDIZIONI: I dati “nome”, “orario”, “max\_partecipanti” e “data” sono dati esistenti.
* POSTCONDIZIONI:
  + Se tutto va bene la lezione viene aggiunta alla lista con tutti i suoi campi
* REALIZZAZIONE
  + 1 Controlliamo se “max\_partecipanti” è minore o uguale a 0; se il controllo risulta vero, stampiamo una scritta di errore.
  + 2 Facciamo un controllo per verificare se la data inserita è una data esistente, se così non fosse, stampiamo un messaggio di errore.
  + 3 Facciamo un controllo per verificare se c’è spazio per registrare una lezione, se così non fosse, stampiamo un messaggio di errore.
  + 4 Inizializziamo la nuova lezione, e stampiamo un messaggio per dire ciò è avvenuto con successo, mostrando anche l’ID della lezione stessa.
* CODICE

int aggiungiLezione(const char\* nome, const char\* orario, int max\_partecipanti, date data) {

// Validazione input

if (max\_partecipanti <= 0) {

printf("Errore: numero massimo partecipanti non valido.\n");

return -1;

}

// Validazione data (semplificata)

if (data.aa < 2023 || data.mm < 1 || data.mm > 12 || data.gg < 1 || data.gg > 31) {

printf("Errore: data non valida.\n");

return -1;

}

LezioneFitness\* nuova = malloc(sizeof(LezioneFitness));

if (!nuova) {

printf("Errore di memoria.\n");

return -1;

}

// Inizializzazione nuova lezione

nuova->id = prossimo\_id\_lezione++;

strncpy(nuova->nome, nome, sizeof(nuova->nome));

strncpy(nuova->orario, orario, sizeof(nuova->orario));

nuova->max\_partecipanti = max\_partecipanti;

nuova->prenotati = 0;

nuova->data = data;

nuova->next = testa\_lezioni;

testa\_lezioni = nuova;

printf("Lezione aggiunta con ID %d.\n", nuova->id);

return nuova->id;

}

***Funzione visualizzaDisponibilita()***

* SPECIFICA SINTATTICA: void visualizzaDisponibilita().
* TIPI: Void.
* SPECIFICA SEMANTICA: visualizzaDisponibilita()void
* PRECONDIZIONI: Esistono i dati “curr” e “testa\_lezioni”;
* POSTCONDIZIONI: Assegniamo alla variabile “curr” il valore della variabile “testa\_lezioni”.
* REALIZZAZIONE:
  + 1 Assegniamo alla variabile “curr” il valore della variabile “testa\_lezioni”;
  + 2 Controlliamo se esiste una lezione disponibile; se esiste stampiamo le informazione inerenti alla lezione, altrimenti stampiamo un messaggio di errore.
* CODICE

void visualizzaDisponibilita() {

printf("\n--- Disponibilità Lezioni ---\n");

LezioneFitness\* curr = testa\_lezioni;

if (!curr) {

printf("Nessuna lezione disponibile.\n");

return;

}

while (curr) {

printf("ID: %d | %s alle %s | Posti: %d/%d | Data: %02d/%02d/%d\n",

curr->id, curr->nome, curr->orario, curr->prenotati,

curr->max\_partecipanti, curr->data.gg, curr->data.mm, curr->data.aa);

curr = curr->next;

}

}

***Funzione verificaPostiDisponibili()***

* SPECIFICA SINTATTICA: int verificaPostiDisponibili(int id\_lezione)
* TIPI: Int.
* SPECIFICA SEMANTICA: verificaPostiDisponibili(id\_lezione)void;
* PRECONDIZIONI: Esistono i dati “curr” e “testa\_lezioni”;
* POSTCONDIZIONI:
  + Restituisce 1 se la lezione con id\_lezione esiste ed il numero di prenotati è minore del massimo dei partecipanti
  + Restituisce 0 se la lezione non esiste o la lezione è completa
* REALIZZAZIONE:
  + 1 Scorre la lista delle lezioni
  + 2 Per ogni elemento, verifica se l'ID corrisponde a quello cercato
  + 3 Se trova corrispondenza confronta prenotati con max\_partecipanti
* CODICE

int verificaPostiDisponibili(int id\_lezione) {

LezioneFitness\* curr = testa\_lezioni;

while (curr) {

if (curr->id == id\_lezione) {

return curr->prenotati < curr->max\_partecipanti;

}

curr = curr->next;

}

return 0;

}

***Funzione incrementaPrenotati()***

* SPECIFICA SINTATTICA: int incrementaPrenotati(int id\_lezione).
* TIPI: Int.
* SPECIFICA SEMANTICA: incrementaPrenotati(id\_lezione) Void;ù
* PRECONDIZIONI: Esistono i dati “curr” e “testa\_lezioni”;
* POSTCONDIZIONI: Variabile prenotati viene incrementata se la funzione va a buon fine
* REALIZZAZIONE
  + 1 Facciamo un controllo per verificare se è possibile prenotarsi a lezione, se così fosse, incrementiamo di 1 i prenotati.
* CODICE

int incrementaPrenotati(int id\_lezione) {

LezioneFitness\* curr = testa\_lezioni;

while (curr) {

if (curr->id == id\_lezione && curr->prenotati < curr->max\_partecipanti) {

curr->prenotati++;

return 1;

}

curr = curr->next;

}

return 0;

}

***Funzione liberaLezioni()***

* SPECIFICA SINTATTICA: void liberaLezioni().
* TIPI: Void.
* SPECIFICA SEMANTICA: liberalezioni() -> void;
* PRECONDIZIONI: Esistono i dati “curr” e “testa\_lezioni”;
* POSTCONDIZIONI: Il dato “testa\_lezioni” viene inizializzato a NULL.
* REALIZZAZIONE:
  + 1 Scorre tutta la lista lezioni
  + 2 Salva il nodo attuale in una variabile temporanea
  + 3 Passa al prossimo nodo
  + 4 Libera la variabile temporanea, così da liberare tutta la memoria
* CODICE

void liberaLezioni() {

LezioneFitness\* curr = testa\_lezioni;

while (curr) {

LezioneFitness\* temp = curr;

curr = curr->next;

free(temp);

}

testa\_lezioni = NULL;

}

incrementaPrenotati(id\_lezione):

***Funzione incrementaPrenotati()***

* SPECIFICA SINTATTICA: int incrementaPrenotati(int id\_lezione).
* TIPI: Int.
* SPECIFICA SEMANTICA: incrementaPrenotati(id\_lezione) Void;
* PRECONDIZIONI: Esistono i dati “curr” e “testa\_lezioni”;
* POSTCONDIZIONI: Variabile prenotati viene incrementata se la funzione va a buon fin
* REALIZZAZIONE
  + 1 Facciamo un controllo per verificare se è possibile prenotarsi a lezione, se così fosse, incrementiamo di 1 i prenotati.
* CODICE

int incrementaPrenotati(int id\_lezione) {

LezioneFitness\* curr = testa\_lezioni;

while (curr) {

if (curr->id == id\_lezione && curr->prenotati < curr->max\_partecipanti) {

curr->prenotati++;

return 1;

}

curr = curr->next;

}

return 0;

}

***Modulo: Prenotazioni (`prenotazioni.h/c`)***

***Funzione inizializzaPrenotazioni()***

* SPECIFICA SINTATTICA: void inizializzaprenotazioni().
* TIPI: void.
* SPECIFICA SEMANTICA: inizializzaPrenoazioni();
* PRECONDIZIONI: la funzione “liberaPrenotazioni()” esiste.
* POSTCONDIZIONE: al dato “testa\_prenotazioni” viene assegnato il valore NULL.
* REALIZZAZIONE:
  + 1 Richiamo della funzione “liberaPrenotazioni()”.
  + 2 Inizializziamo “testa\_prenotazioni” a NULL.
* CODICE:

void inizializzaPrenotazioni() {

liberaPrenotazioni();

testa\_prenotazioni = NULL;

}

***Funzione aggiungiPrenotazione()***

* SPECIFICA SINTATTICA: int aggiungiPrenotazione(int, int ).
* TIPI: int.
* SPECIFICA SEMANTICA: aggiungiPrenotazione(id\_cliente, id\_lezione)
* PRECONDIZIONI: il cliente deve esistere e deve possedere l’abbonamento;
* POSTCONDIZIONI: Viene prenotata una lezione.
* REALIZZAZIONE
  + 1 Facciamo un controllo per verificare la validità dell’abbonamento,
  + 2 Controlliamo se la lezione selezionata è al completo,
  + 3 Troviamo in cliente e decrementiamo l’abbonamento,
  + 4 Incrementa contatore prenotati nella lezione,
  + 5 Crea una nuova prenotazione.
* CODICE:

int aggiungiPrenotazione(int id\_cliente, int id\_lezione) {

// Verifica validità abbonamento cliente

if (!verificaValiditaAbbonamento(id\_cliente)) {

printf("Abbonamento non valido per il cliente %d.\n", id\_cliente);

return 0;

}

// Verifica disponibilità posti

if (!verificaPostiDisponibili(id\_lezione)) {

printf("La lezione %d è al completo.\n", id\_lezione);

return 0;

}

// Trova il cliente e decrementa l'abbonamento

Cliente\* c = testa\_clienti;

while (c) {

if (c->id == id\_cliente) {

if (c->giorni\_abbonamento > 0) {

c->giorni\_abbonamento--; // DECREMENTO IMMEDIATO QUI

break;

} else {

printf("Errore: abbonamento esaurito per il cliente %d.\n", id\_cliente);

return 0;

}

}

c = c->next;

}

// Incrementa contatore prenotati nella lezione

if (!incrementaPrenotati(id\_lezione)) {

printf("Errore durante la prenotazione.\n");

return 0;

}

// Crea nuova prenotazione

Prenotazione\* nuova = malloc(sizeof(Prenotazione));

if (!nuova) {

printf("Errore di memoria.\n");

return 0;

}

nuova->id\_cliente = id\_cliente;

nuova->id\_lezione = id\_lezione;

nuova->next = testa\_prenotazioni;

testa\_prenotazioni = nuova;

printf("Prenotazione registrata per cliente %d alla lezione %d.\n", id\_cliente, id\_lezione);

return 1;

}

***Funzione stampaPrenotazioniCliente()***

* SPECIFICA SINTATTICA: void stampaPrenotazioniCliente(int)
* TIPI: Void, Int.
* SPECIFICA SEMANTICA: stampaPrenotazioniCliente(id\_cliente).
* PRECONDIZIONI: Il cliente deve esistere, e deve aver prenotato almeno una lezione esistente.
* REALIZZAZIONE:
  + 1 In base all’ID di un cliente, vengono stampate tutte le lezioni prenotate dal cliente stesso.
* CODICE:

void stampaPrenotazioniCliente(int id\_cliente) {

Prenotazione\* curr = testa\_prenotazioni;

int trovato = 0;

printf("\n--- Lezioni prenotate dal cliente %d ---\n", id\_cliente);

while (curr) {

if (curr->id\_cliente == id\_cliente) {

trovato = 1;

printf(" - Lezione ID: %d\n", curr->id\_lezione);

}

curr = curr->next;

}

if (!trovato) {

printf("Nessuna prenotazione trovata per questo cliente.\n");

}

}

***Funzione generaReport()***

* SPECIFICA SINTATTICA: void generaReport();
* TIPI: void.
* SPECIFICA SEMANTICA: generaReport();
* PRECONDIZIONI: Devono esistere delle prenotazioni e delle lezioni.
* POSTCONDIZIONI: Viene incrementato il contatore corrispondente all’id della lezione.
* REALIZZAZIONE:ù
  + 1 Scorre la lista delle prenotazioni e incrementa il contatore corrispondente all’id della lezione
  + 2 Scorre l’array dei contatori e stampa le lezioni con almeno una prenotazione
* CODICE:

void generaReport() {

int frequenza[100] = {0}; // Array per contare le prenotazioni per lezione

Prenotazione\* curr = testa\_prenotazioni;

// Conta le prenotazioni per ogni lezione

while (curr) {

if (curr->id\_lezione > 0 && curr->id\_lezione < 100) {

frequenza[curr->id\_lezione]++;

}

curr = curr->next;

}

// Stampa il report

printf("\n--- Report Utilizzo Lezioni ---\n");

for (int i = 1; i < 100; i++) {

if (frequenza[i] > 0) {

printf("Lezione ID %d: %d prenotazioni\n", i, frequenza[i]);

}

}

}

***Funzione liberaPrenotazioni()***

* SPECIFICA SINTATTICA: void liberaPrenotazioni().
* TIPI: void.
* SPECIFICA SEMANTICA: liberaPrenotazioni().
* PRECONDIZIONI: deve esistere una lista prenotazioni.
* POSTCONDIZIONI: viene liberato lo spazio in memoria in cui era registrata la lista delle prenotazioni.
* REALIZZAZIONE:
  + 1 Liberiamo lo spazio in memoria in cui era registrata la lista delle prenotazioni.
* CODICE:

void liberaPrenotazioni() {

Prenotazione\* curr = testa\_prenotazioni;

while (curr) {

Prenotazione\* temp = curr;

curr = curr->next;

free(temp);

}

testa\_prenotazioni = NULL;

}

***Funzione generaReport():***

* SPECIFICA SINTATTICA: void generaReport()
* TIPI: void.
* SPECIFICA SEMANTICA: generaReport()
* PRECONDIZIONI: Devono esistere delle prenotazioni e delle lezioni.
* POSTCONDIZIONI: Viene incrementato il contatore corrispondente all’id della lezione.
* REALIZZAZIONE
  + 1 Scorre la lista delle prenotazioni e incrementa il contatore corrispondente all’id della lezione
  + 2 Scorre l’array dei contatori e stampa le lezioni con almeno una prenotazione
* CODICE:

void generaReport() {

int frequenza[100] = {0}; // Array per contare le prenotazioni per lezione

Prenotazione\* curr = testa\_prenotazioni;

// Conta le prenotazioni per ogni lezione

while (curr) {

if (curr->id\_lezione > 0 && curr->id\_lezione < 100) {

frequenza[curr->id\_lezione]++;

}

curr = curr->next;

}

// Stampa il report

printf("\n--- Report Utilizzo Lezioni ---\n");

for (int i = 1; i < 100; i++) {

if (frequenza[i] > 0) {

printf("Lezione ID %d: %d prenotazioni\n", i, frequenza[i]);

}

}

}

## 4. Razionale dei Casi di Test

I test sono progettati per verificare:

* Il corretto funzionamento delle funzioni principali del programma, così da verificare se vengono inseriti correttamente i clienti con i propri campi;
* Stessa cosa vale per le lezioni, si verifica il corretto inserimento dei campi;

testPrenotazioniEDisponibilita: Verifica che prenotazioni valide aggiornino correttamente tutti gli ADT.

testAbbonamenti: Verifica se l’abbonamento viene aggiunto correttamente e ne verifica la validità.

Integrità del Sistema:

testReport: Verifica che il report rifletta accuratamente lo stato delle prenotazioni

Priorità alle Operazioni principali: Test approfonditi su aggiungiPrenotazione (cuore del sistema).

Controllo Errori: Test su input invalidi (es. clienti inesistenti, lezioni piene).

Consistenza: Verifica che le modifiche a un ADT si riflettano sugli altri (es. decremento abbonamento → aggiornamento report).

! Per eseguire il make per il Makefile\_test usare: make -f Makefile\_test test !

## Conclusioni

L’uso di liste concatenate si è dimostrato adeguato per:

- Gestire dati dinamici con operazioni semplici.

- Mantenere un basso consumo di memoria.

- Facilitare la manutenzione e l’estensione del codice.