Manuale Tecnico

Emilio Daverio - Emilio Toli - Stefano Farina - Cristian Stinga

Università degli Studi dell'Insubria – Laurea Triennale in Informatica

Progetto Laboratorio A: Emotional Songs

Sommario

0.1 Introduzione

Emotional Songs è un progetto sviluppato nell'ambito del progetto di Laboratorio A per il corso di laurea in Informatica dell'Università degli Studi dell'Insubria.
Il progetto è crearto usando la versione 8 di java con jdk 19, ed è stato sviluppato in Windows 10 e Windows 11. Mentre per testarlo abbiamo usato divresi sistemi operativi in quanto le specifiche del progetto richiedevano che l'applicazione multipiattaforma. I sistemi orperativi uati sono i seguenti:

- Windows 10
- Windows 11
- Linux
- Mac-OS

0.1.1 Librerie Esterne Utilizzate:

Per creare questo progetto sono state utilizzate le seguenti Librerie Esterne:

- import java.io.*
- import java.io.InputStreamReader
- import java.io.IOException
- import java.io.BufferedReader
- import java.text.DecimalFormat
- import java.util.Scanner

vediamole più nel dettaglio:

1. **import java.io.***
   Questa libreria fornisce degli input e degli output di stistema tramite flussi dati; ovvero fornisce un insieme di flussi di input e un insieme di flussi di output utilizzati per leggere e scrivere dati su file o altre sorgenti di input e output.
2. **import java.io.InputStreamReader**
   Libreria che mette a disposizione metodi per ricevere in input un flusso di dati che vengono appositamente decodificati.
3. **import java.io.IOException**
   Classe che estende direttamente EXCEPTION, segnala che è stata prodotta un'eccezione generata da operazioni di input e/o di output le quali sono state interrotte o non sono adate a termine per qualche motivo.
4. **import java.io.BufferedReader**
5. **import java.text.DecimalFormat**
   è una sottoclasse di NumberFormat la quale serve per formatta i numeri decimali. Inoltre può essere usata per troncare (in base a quante cifre decimali voi dopo la virgola) i numeri periodici o con tante cifre dopo la virgola.
6. **import java.util.Scanner**
   La libreria java.util contiene il framework delle raccolte, le classi di raccolta legacy, il modello di eventi, le strutture di data e ora, l'internazionalizzazione e le classi di utilità varie (un tokenizzatore di stringhe, un generatore di numeri casuali e un array di bit). In particolare noi abbiamo utilizzato la lista java.util.Scanner per andare a leggere ciò che l'utente digita ad tastiera (input).
7. **import java.util.Locate**

0.1.2 Struttura Generale Del Sistema Di Classi

L'applicazione si basa sul concetto di classi (oggetti), le quali sono usate per gestire le varie operazioni che un utente può fare. All'interno di ogni classe ci sono dei metodi che gestiscono e controllano eventuali input che l'utente inserire dutrante l'uso dell'applicazione.

0.1.3 Classi Principali

Le classi principali che sono state sviluppate per creare e gestire l'intero progetto sono le seguenti:

- EmotionalSongs
- Utenti
- Emozioni
- Canzoni
- PlayList

Ora andremo a vederle più nel dettaglio, analizzando anche le varie complessità degli algoritmi che si trivano al loro interno.

0.2 CLASSI

0.2.1 EmotionalSongs

La classe EmotionalSongs è la classe più importante dell'applicazione, perchè qui vengono gestite tutte le operazioni e interazioni con altre classi e poi perchè è la prima parte di programma con cui un untente si "unterfccia" quando accede sull'applicazione per la prima volta o meno. EmotionalSongs è formata da vari menù con i quali l'utente si deve relazionare per svolgere le operazioni a loro concesse. Come accenato in questa classe è presente la funzione "main", che è il primo metodo utilizzato nel programma ed è la base per l'esecuzione di esso.

0.2.2 Utenti

Lo scopo principale di questa classe è gestire la registrazione di un nuovo utente; per far questo usa dei metodi di controllo, i quali verificano che i dati personali dell'utente seguano il formato richiesto. I metodi che troviamo all'interno di questa classe sono i seguenti:

- Registrazione
- toString
- ControlloFormatocf
- controlloMail
- controlloPassword
    1. controlloNonNulla
        - controlloFormato
    2. controlloPassUguale
- controlloUserEsistente
    1. esisteUtente
    2. controlloUser
- LunghezzaNome
    1. soloLettere
- LunghezzaCognome
    1. soloLettere

- controlloCAP
- controlloNumeroCivico
- ScriviFile
- Login
- controlloCOMUNEnonNull
- controlloCOMPROV
- controlloProvincianonNull

Metodo: Registrazione

Il metodo Registrazione, come dice il nome, serve per far si che un nuovo utente si possa registrare sull'applicazione. Andrà a salvare sul file: *UtentiRegistrati.dati* tutti i dati personali dell'utente, che gli sono stati passati come parametro al metodo. La sua complessità non è altro che O(1) (tempo costante) perchè deve inserire solamente dai nuovi dati nel file.
NOTA: il file *UtentiRegistrati.dati* conterrà tutti i dati di tuti gli utenti che si sono registrati sull'applicazione.

Metodo: toString

Questa funzione mi ritorna l'ordine in cui i dati devono essere salvati nel file e mi dice anche quale carattere speciale viene ustato per separe i dati l'uno dall'altro.

Metodo: ControlloFormatocf

Questo metodo serve per verificare che nel momento dell'inserimento del codice fiscale l'utente segua correttamente il formato rischiesto dall'applicazione. Visto che è solo un controllo su un singola stringa (codice fiscale appena inserito) la complessità è O(1) (costante)

Metodo: controlloMail

Questo metodo, come il prededente, serve per controllare nel monento dell'inserimento della mail che venga rispettato il formato corretto. Visto che è solo un controllo su una singola mail (quella appena inserita) la complessità è O(1)

Metodo: controlloPassword

Anche in questo caso abbiamo il metodo controlloPassword che controlla la password inserita al momento della registrazione; però a differenza dei metodi precedenti, l'operazione di controllo della password si divide in più fasi perchè devo verificare la correttezza di due password. Vediamo le vari fai:

1. controllo prima password:
    - prima fase:
      richiama il metodo controlloNonNulla che controlla che l'utente abbia inserito almeno una stringa al momento di inserire la password. Se la password è "nulla" allora uscirà il messaggio di rinserila nuovamente, altrimenti si passa alla seconda fase
    - seconda fase:
      quando arriviamo in questa fase vuol dire che siamo usciti da ciclo do-while del metodo *controlloNonNulla* che mi costringeva ad inserire almeno dei caratteri.
      Sempre il metodo controlloNonNulla richiama e cede il controllo ad un altro metodo: *controlloFormato* che controlla che la password inserita rispetti il formato richiesto; altrimenti uscirà sempre un messaggio di errore dove chiederà di reinserire nuovamente la password. Riamango in questo ciclo do-while finche il formato della password non rispetti quello richiesto.
    - terza fase:
      arrivati in questa fase il metodo controlloFormato mi ritorna la password nel formato corretto; quindi ricede il controllo al metodo controlloNonNulla il quale restituirà la password appena inserita.

- quarta fase:
  a questo punto quando ho controllato la prima password (e sono tornato nel metodo controlloPassword) passo alla conferma della password

2. Conferma Password:
   Arrivato a questo punto l'utente ha inserito la Password che desidera nel formato corretto ed ora deve digitarla nuovamente per confermalrla.

Finito di inserire anche la conferma della password, bisogna fare un ultimo controllo, ovvero quello verificare che le due password concidano perfettamente. Questo compito è affidato al metodo: *controlloPassUguale* che prende in input le due password e le confronta all'interno di un ciclo do-while. Rimango in quel ciclo fino a quando la seconda password inserita (quella di conferma) non è uguale alla prima. Una volta uscito restituisco la prima password. in tutto questo la complessità di questo algoritmo è sempre O(1).

Metodo: controlloUserEsistente

Questo controllo serve a verificare che lo UserId inserito dall'utente al momento della registrazione sia già presente o meno nel file *UtentiRegistrati.dati*. Questo controllo lo si fa richiamando il metodo *esisteUtente*; se l'Id scerlto si trova all'interno del file allora verrà restituito il valore booleano **true** e di conseguenza comparirà a video un messagio di errore, nel quale verrà richiesto di inserire un nomeUtente diverso. Se invece il valore ristituito è **false** significa che il nome scelto va bene (non si trova nel file), e di conseguenza verrà "attivato" un altro metodo di controllo: *controlloUser* che controlla se il nomeUtente rispetti il formato richiesto.
In questo caso la complessità è $\Theta(n)$ in quanto la ricerca del nomeUtente richiede di scandire tutto il file per verificare se è già presente o meno.

Metodo: LunghezzaNome

Con questo metodo viene fatto un controllo sul nome dell'utente; prima di tutto viene controllato che non sia nullo e che sia almeno lungo 3 caratteri. Dopo di che viene richiamato il metodo *soloLettere* che mi consente di verificare se il nome inserito sia composto da solo lettere, altrimenti verrà generato un messaggio di errore dove si dice che deve contenere solo caratteri AlfaNumerici.
La complessità è sempre O(1)

Metodo: LunghezzaCognome

Con questo metodo viene fatto un controllo sul cognome dell'utente; prima di tutto viene controllato che non sia nullo e che sia almeno lungo 3 caratteri. Dopo di che viene richiamato il metodo *soloLettere* che mi consente di verificare se il cognome inserito sia composto da solo lettere, altrimenti verrà generato un messaggio di errore dove si dice che deve contenere solo caratteri AlfaNumerici.
La complessità è sempre O(1)

Metodo:controlloCAP

In questo caso, come dice il nome, verrà fatto il controllo sul cap, il quale deve essere complreso nell'intervallo tra [10, 97100]; altrimenti comparirà un messaggio di errore dove verrà richiestodi rinserirlo. Dopo di che viene controllata la sua lunghezza:

- se la lunghezza è uguale 5
  allora verrà accetta come input

- se la lunghezza è nimore di 5
  vengono aggiunti tanti zero, davanti al numero, quanti me ne servono per arrivare a 5.

La complessità è sempre O(1) (costante) in quanto deve solo confrontare la stringa appena inserita.

Metodo: controlloNumeroCivico

Controlla che il numero civico inserito rispetti i formati richiesti altrimenti si rimmarà ne ciclo while fino a quando si inserisce il formato richiesto.
La complessità è sempre O(1)

Metodo: ScriviFile

Serve semplicemente per salvare sul file i dati dell'utente nell'ordine corretto. La complessità è sempre O(1)

Metodo: Login

Come dice il nome del metodo, questo controllo serve per effettuare il login sull'applicazione.
Innanzitutto viene chiesto di inserire NomeUtente e Password (scelti al momento della registrazione), dopo di che si prende il file *UtentiRegistrati.dati* e si inizia a scandirlo. Il metodo si ferma quando o ha letto tutto il file o quando trova l'Id e la password corrispondenti. Qullo che fa questo metodo e semplicemente confrontare il nomeUtente e la password inseriti dall'utente con quelli presenti nel file. In questo caso ho tre possibili risultati di output:

1. login effetuato con successo
   vuol dire che ti eri regitrato in precedenza ed ora puoi svolgere ulteriori attività.
2. login errato
   perche la password e/o il NomeUtente inseriri sono sbagliati
3. login errato
   perchè non si è ancora effettuata la registrazione.

La complessità di questo metodo è $O(n)$, perchè il metodo si può fermare anche prima di aver letto tutto il file. Nel caso peggiore la complessità sarà $\theta(n)$ perchè deve scendire tutto il file. NOTA: la Complessità nel caso peggiore si verifa quando un utente inserisce scorrettamente i dati o prova a loggarsi anche se non si è registrato .

Metodo: controlloCOMUNEnonNull:

Questo metodo di controllo è usato per verificare che inizialmente il comune inserito dall' utente non sia Null. Superato questo controllo, viene invocato il metodo privato *controlloCOMPROV* che mi controlla che la stringa appena inserita contenga solo caratteri alfanumerici. Una volta effettuati tutti i controlli, quello che viene restituito è il comune che rispetti tutti i parametri.
La complessità è semplicemente O(1)

Metodo: controlloProvincianonNull:

Questo metodo di controllo è usato per verificare che inizialmente la provincia inserita dall' utente non sia Null. Superato questo controllo, viene invocato il metodo privato *controlloCOMPROV* che mi controlla che la stringa appena inserita contenga solo caratteri alfanumerici. Una volta effettuati tutti i controlli, quello che viene restituito è la provincia che rispetti tutti i parametri.
La complessità è semplicemente O(1)

0.2.3 Emozioni

Questa classe viene usata per gestire e visualizzare le emozioni legate alle varie canzoni; per far questo sono stati creati due metodi per:

- Inserire nuove Emozioni
- Visualizzare le Emozioni

Metodo: inserisciEmozioni

Per Inserire delle nuove emozioni viene usato il metodo statico *inserisciEmozioni* il quale prende in input come paramero il titolo (sia che la canzone venga ricercata per titolo o per autore e anno) della canzone sulla quale si vuole inserire una nuova emozione. Dopo aver inserito il titolo, tramite un ciclo do-while verrà visualizzato un menù con un lista delle emozioni che l'utente può aver provato mentre ascoltava la canzone[1]. Una volta scelta l'emozione, si deve inserire una breve descrizione del perchè si è scelta quell' emozione (al massimo 50 caratteri). In seguito verrà chiesto di dare un punteggio (tra 1 e 5) dell'emozione, ovvero quanto forte hai provato qull'emozione, ed infine l'utente

[1] come lista delle emozioni usiamo una scala standard: GEMS (Geneva Emotional Music Scale)

dovrà insrire delle note dove potrà esprimere il perchè ha scelto proprio quell'emozione e il perchè del punteggio inserito (max 250 caratteri). Una volta completata la procedura di inserimento della nuova emozione, tutti i dati inseriti saranno salvati sul file Emozioni.dati

In questo caso la complesità di questo algoritmo è O(1) (costante) perchè semplicemente deve inserire la nuova emozione nel file.

Metodo: visualizzaEmozioni

Con questo metoto l'utente dopo che ha cercato le canzoni (per titolo, autore e anno) può decidere di visualizzare o meno la percentuale delle varie emozioni assocate a quella canzone. Il risultato dell'operazone sarà:

- percentuale di tutte le emozioni
- 0 %
  se un emozione (associata a quella canzone) non è ancora stata "inserita" da nessun utente.
- NAN (Not-A-Number)
  compare quando quella canzone non è presente nel file Emozioni.dati in quanto nesssun utente ha ancora inserito delle emozioni associate a quella canzone.

In questo caso l'algoritmo per visualizzare le emozioni deve scandire tutto il file Emozioni.dati e verificare se effettivamente quella canzone è presente o meno nel file; se é presente, allora deve tenere traccia di vari aspetti che poi serviranno per calcolare la percentuale delle emozioni. Gli aspetti da tener conto sono:

- Quante volte compare il titolo della canzone
- Quante volte compare la stessa emozione associata a quella canzone.

Per fare tutto questo vengono usate delle variabili count appropriate. Tutto questo ci porta ad una complessità che dipende strettamente dalla dimensione del file. Comunque la sua complessità è $\theta(n)$, perche deve scandire tutto il file.
Per quanto riguarda l'inserimento di una nuova emozione il tempo di acceso al file è costatnte, qundi la complessità dell'algoritmo è O(1) per inserire una nuova emozione nel file.

0.2.4 Canzioni

Questa classe gestisce tutto quello che riguarda la partre delle canzoni. Ovvero gestisce la ricerca e la visualizzazione delle canzoni. I metodi che sono stati creati sono i seguenti:

- cercaBranoMusicaleTitolo
  1. visualizzaEmozioni
  2. inserisciEmozioni
- cercaBranoMusicaleAutoreAnno
  1. visualizzaEmozioni
  2. inserisciEmozioni
- controlloCanzoneEsistente
- ricercaCanzoni
- numeroTotaleCanzoni

Metodo: cercaBranoMusicaleTitolo

Questo metodo mi permette di ricercare una canzone all' interno del file *Canzoni.dati* passandogli come parametro il Titolo della canzone desiderata.
Qundi quello che fa, è prendere il file e ricercare la canzone, leggendo tutto il file. Se la canzone non viene trovata allora comparirà a video un messagio di errore con scritto *Canzone non trovata*. Se invece la canzone viene trovata, allora verranno stampate a video tutte le informazioni relative alla canzone (titolo, autore, anno). Dopo di che comparirà un menù con le operazioni che posso

essere fatte una volta trovata la canzone, tenendo conto del valore booleano che il parametro loggato, passato come secondo parametro alla funzione cercaBranoMusicaleTitolo, assume in quel momento; perchè se é **False** (non sei loggato) allora l'utente può decidere di visulizzare le emozioni associate a quella canzone attraverso il metodo *visualizzaEmozioni* e non può far nient'altro; mentre se il valore è **True** (sei loggato) allora oltre che a visualizzare le emozioni l'utente può inserire una nuova emozione attraverso il metodo *inserisciEmozioni*.
In questo caso la complessità dell'algoritmo è data da più complessità perchè:

- la ricerca della canzone:
  richiede una complessità di O(n) perchè mi fermo solo quando trovo la canzone, la quale può trovarsi in una qualunque riga del file, oppure quando ho finito di leggere tutto il file.
- la visualizzazione delle Emozioni:
  richiede $\Theta(m)$ come complessità perchè deve obbigatoriamente leggere tutto il file *Emozioni.dati*
- Inserimento di una nuova emozione:
  richiede semplicemente tempo costante O(1)

Qunidi complessivamente la complessità di questo metodo è $O(n \cdot m)$

Metodo: cercaBranoMusicaleAutoreAnno

Questo metodo mi permette di ricercare una canzone all' interno del file *Canzoni.dati* passandogli come parametro il nome dell'autore e l'anno di uscita della canzone.
Qundi quello che fa è prendere i parametri che gli sono stati passati e confrontarli con quelli scritti sul file. Se la canzone non viene trovata allora comparirà a video un messagio di errore con scritto *Canzone non trovata*. Se invece la canzone viene trovata, allora verranno stampate a video tutte le informazioni relative alla canzone. Dopo di che comparirà un menù con le operazioni che posso essre fatte una volta trovata la canzone tenendo conto del valore booleano che che il parametro loggato, passato come secondo parametro alla funzione cercaBranoMusicaleAutoreAnno, assume in quel momento; perchè se é **False** allora l'utente può visulizzare solo le emozioni associate a quella canzone attraverso il metodo *visualizzaEmozioni*; mentre se il valore è **True** allora oltre che a visualizzare le emozioni l'utente può anche inserire una nuova emozione legata a quella canzone attraverso il metodo *inserisciEmozioni*.
Anche in questo caso la complessità dell'algoritmo è data da più complessità perchè:

- la ricerca della canzone:
  richiede una complessità di O(n) perchè mi fermo solo quando trovo la canzone, la quale può trovarsi in una qualunque riga del file, oppure quando ho finito di leggere tutto il file.
- la visualizzazione delle Emozioni:
  richiede $\Theta(m)$ come complessità perchè deve obbigatoriamente leggere tutto il file *Emozioni.dati*
- Inserimento di una nuova emozione:
  richiede semplicemente tempo costante O(1)

Qunidi complessivamente la complessità di questo metodo è $O(n \cdot m)$

Metodo: controlloCanzoneEsistente

Questo metodo non fa altro che restituirmi un valore booleano (true, false) in base se ha trovato o meno la canzone nel file *Canzoni.dati* . Quindi mi serve solo per cercare se una canzone è presente o meno nel file, ed è stato creato per semplificare le operazioni di ricerca al momento di cancellare una canzone dalla plyalist.
Di conseguenza la sua complessità è O(n) perchè devo nel caso peggiore leggere tutto il file, mentre mi fermo prima se la trovo.

Metodo: ricercaCanzoni

Mi serve semplicemente per ricervare una canzone nel file. E' stato creato per semplificare l'operazione di aggiunta di una canzone nella playlist. In questo caso quando trovo un canzone restituisco il numero di riga della canzone, in che posizione si trova all'interno del file. Altrimenti restituisco che non è stata trovata.
Di conseguenza la sua complessità è sempre O(n)

Metodo: numeroTotaleCanzoni

Non fa altro che contarmi il numero totale di canzoni presente nel file *Canzoni.dati*
Qundi la sua complesità non è altro che O(n) perchè deve leggere tutto il file.

0.2.5 PlayList

Questa classe serve per gestire tutto l'aspetto riguardante l'inserimento, la visualizzazione, la cancellazione, la creazione di una plylist e gestisce anche l'aspetto che riguarda l'inserire, visualizzare, calcellare delle canzoni da una playlist. Per fare tutto questo sono stati creati i seguenti metodi:

- registraPlaylist
  1. controlloPlaylistEsistente
  2. aggiuntaCanzoniServ
  3. insertByLine
- aggiuntaCanzoniServ
  1. ricercaCanzoni
  2. numeroTotaleCanzoni
- aggiungiDopoInPlaylist
  1. controlloPlaylistEsistente
  2. numTotPlaylist
  3. insertByLine
- eliminaCanzoneDaPlaylist
  1. controlloCanzoneEsistente
- visualizzaPlaylistUtente
- cancellaPlaylist
  1. controlloPlaylistEsistente
- numTotPlaylist
- visualizzaCanzoniPlaylist

Metodo: registraPlaylist

Questo metodo mi permette di creare una nuova playlist; in base all'idUtente e al nome della playlist che gli vengono passati come parametri.
Quindi come prima cosa il metodo guarda se il nome della nuova playlist è gia presente all'interno del file *Playlist.dati* attraverso il metodo *controlloPlaylistEsistente* che mi restituisce due valori: True vuol dire che quel nome è gia stato usato da quell'utente per crare un'altra playlist; False: significa che il nome non è stato usato e qiundi lo può usare tranquillamente. Se il valore restituito è TRUE allora comparirà un messaggio di errore dove verà chiesto di inserire nuovamente il nome della nuova playlist da creare; mentre se il valore restituito è FALSE allora il nome inserito va bene. Una volta superaro questo controllo partirà un menù dove verrà stampato a video il suguente messagio: Vuoi aggiungere canzoni alla Playlist? Digitare si o no

- se digito SI:
  verrà ceduto il controllo al metodo *aggiuntaCanzoniServ* il quale richiede di inserire in input il nome della canzone che voglio aggiungere nella nuova playlist. Tramite il metodo *ricercaCanzoni* viene cerca la canzone richiesta nel file, "Cnazoni.dati". Il risulato sarà il numero di linea che identifica univocamente quella canzone, se essa è presente nel file; altrimenti uscirà un messaggio di errore (nessuna canzone trovata). A questo punto restituisco il controllo al menù di prima; il quale l'azione successiva che farà sarà chiamare il metodo *insertByLine* e in base al numero della riga va a prendere il nome della canzone (titolo) nel file Canzoni.dati restituendolo al menù. Come ultima operazione sarà quella di inserire e salvare nel file Plailist.dati il nome della nuova plylist e tutte le canzoni che abbiamo voluto inserire. NOTA: in questa fase si possono aggiungere tutte le canzoni che si vogliono aggiungere. Si ternima quando viene digitata la parola NO.

- se digiti NO:
  quello che fa è semplicemente crera e salvare la playlisy nel file. NOTA: si possono creare playlist che non contengono canzoni.

la complessità di questo metodo è O($n \cdot m \cdot y$) perchè dipende dalle seguenti comlessità:

1. Complessità della ricerca del nome della playlist:
   è Θ(n) perchè deve scandire tutto il file
2. Complessità metodo aggiuntaCanzoniServ:
   è O(m) perchè deve ricercare il titolo della canzone nel file.
3. Complessità metodo insertByLine:
   è O(y) perchè scandisce il file Canzoni fino a quando non trova la riga corrisponddente.
4. Complessità di aggiungere il tutto nel file Playlist:
   è O(1)

Metodo: esisteCanzone

Questo controllo non fa altro che, preso come paramtero un titolo, cercare la canzone nel file Canzoni.dati restituendo True se la trova o False se non la trova. la complessità è O(n)

Metodo: aggiungiDopoInPlaylist

Questo metodo mi serve per far si che un utente possa inserire una canzone in un secondo momento in una delle playlist a lui associate.
I parametri che li vengono passati sono il nome della playlist e lo UserId. Come prima cosa viene controllato che il nome della playlist, associata a quell'utente, esista e/o sia corretto il nome che li viene passato. Tutto questo lo si fa attraverso il metodo *controlloPlaylistEsistente* il quale scandisce tutto il file Playlist.dati e controlla se quel nome è presente o meno restituendo: **True** se ha trovato la playlist, **False** se non la trovata, stampando a video un messaggio di errore. Una volta che ritorno il valore, restituisco il controllo al metodo aggiungiDopoInPlaylist che confronta il risultato, se quest'ultimo è positivo (true), ovvero è stata trovata quella playlist, si entrerà in un ciclo do-while il quale come prima cosa usa il metodo *aggiuntaCanzoniServ* per ricercare la canzone che voglio inserire nella playlist salvado il numero della linea (valore restituito da questo metodo) in una varibile; dopo di che inserisco la canzone nella playlist corretta in ultima posizione (in coda alle altre canzoni presenti all'interno di quella playlist). In fine chiedo all'utente se desidera inserire altre canzoni, oltre a quella appena aggiunta, tramite un menù; se la risposta è positiva (si) allora rifaccio la stessa procedura vista prima (ricerca la canzone e la aggiungo).[2] Se la risposta è negativa (non si vuole aggiungere ulteriori canzoni) allora quello che seguirà sarà una procedera di "Riscrittuara" del file, e al termine dell'operazione sarà/saranno aggiunte tutte le nuove canzoni.

La complessità di questo metodo è O(n)

Metodo: eliminaCanzoneDaPlaylist

Molto seplicemente questo metodo mi serve per cancellare una canzone da una plylist. Per far questo devo passare al metodo come parametri lo UserId, il nome della playlist dalla quale volglio eliminare una canzone e il titolo della canzone che voglio togliere. Naturalmente per far questo devo leggere il file *Playlist.dati* e controllare che il nome della playlist e la canzone siano presenti. Se i controlli sono positivi allora l'operazione successiva sarà aggiornare il file, ovvero riscriverlo togliendo la/le canzone/canzoni che l'utente ha deciso di eliminare. Altrimenti comparià un messaggio di errore che la palylist e/o la canzone non sono presenti nel file.

La complessità di questo metodo è O(n)

Metodo: visualizzaPlaylistUtente

In questo caso l'utente ha la possibilità di visualizzare tutte le playlist che ha creato.
NOTA: questo metodo visualizza solo i nomi delle playlist e non le canzoni che contengono.

La sua complessità è obbligatoriamente $\Theta(n)$ perchè deve scandire tutto il file delle playlist

[2] In questo caso l'inserimento di una nuova canzone verrà fatto sulla stessa playlist. Se si volesse aggiungere una canzone in un altra playlist bisogna uscrire e rifare tutta la procedura.

Metodo: cancellaPlaylist

Anche questo medoto, come quello descritto precedententemete, serve per cancellare qualcosa; ma a differenza di prima che cancellavo una canzone, ora invece cancello proprio tutta una playlist e di conseguenza anche tutte le canzoni ad essa associate. I parametri che dovranno esssere passati sono UserId e il nome della playlist che l'utente ha intenzione di eliminare. Naturalmente verrà fatto un controllo per vedere se la playlist che si vuole eliminare è presente o meno nel file attarverso il metodo *controlloPlaylistEsistente*, se la risposta è positiva allora l'operazione successiva sarà quella di cancellare dal file Playlist.dati quella playlis, altrimenti comaparirà un messaggio di errore (La playlist che hai inserito non esiste!) La complessità è O(n).

Metodo: numTotPlaylist

Mi conta quante playlist sono associate a quell'utente.

La complessità è $\Theta(n)$perchè deve scandire tutto il file

Metodo: visualizzaCanzoniPlaylist

In questo caso l'utente ha la possibilità di visualizzare tutte le canzoni che si trovano in una determimata playlist.
NOTA: questo metodo visualizza solo i nomi delle canzoni in base al nome della playlist passato come parametro al metodo.

La sua complessità è O(n).