Zoratti Marco, n. 5562866

# Compilazione ed Esecuzione

Decomprimere il file zip ed eseguire il seguente comando:

`g++ *.cpp -std=c++11 -Wall`

# Relazione Laboratorio Individuale

Per la ricerca e realizzazione di una struttura dati ottimale, efficiente e scalabile sono state svolte diverse analisi di complessità ed è stata scelta la miglior soluzione possibile.

## Obiettivo

Realizzare una struttura dati capace di leggere, aggiungere e modificare dati e relazioni di un database di film; in particolare, la struttura dati deve svolgere nel meno tempo possibile le seguenti funzioni:

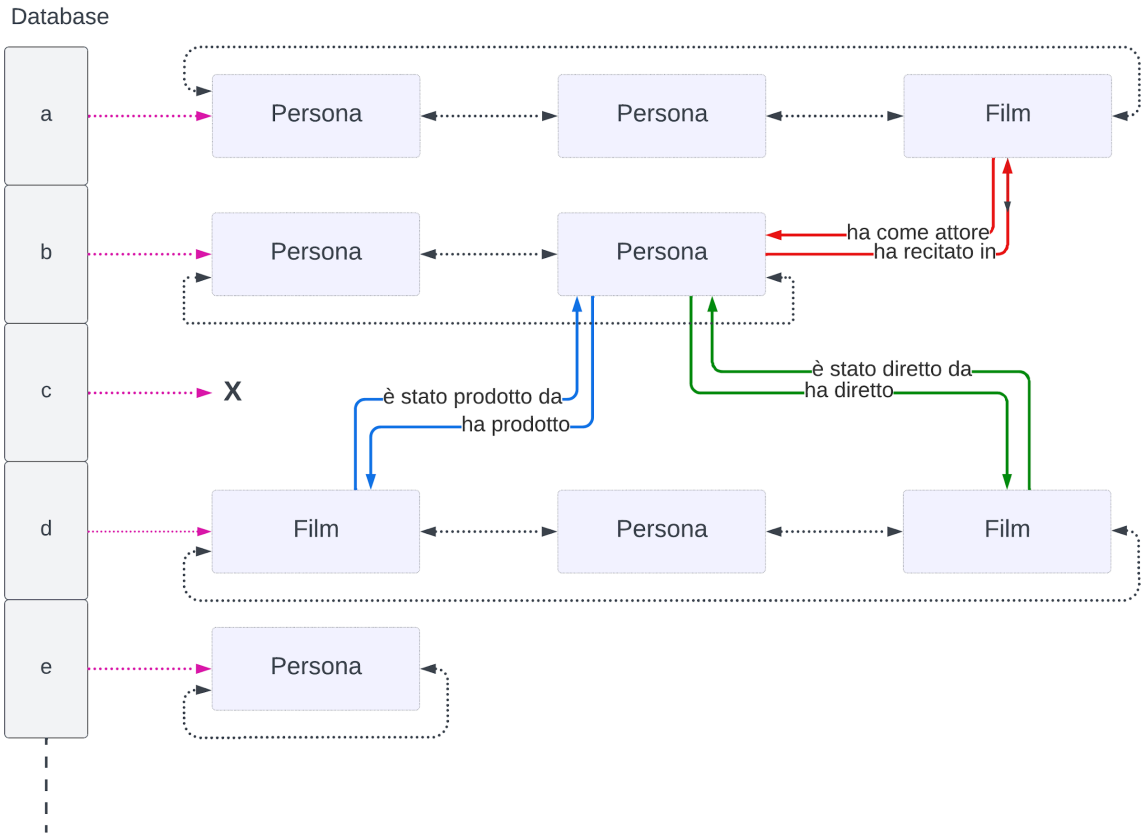
| Scrittura | Lettura |
|---|---|
| aggiungere un nodo corrispondente a una persona | stampare il contenuto del database |
| aggiungere un nodo corrispondente a un film | stampare il numero di persone/film |
| aggiungere una relazione tra due nodi | stampare il numero di relazioni della persona/del film |
| | stampare gli attori che hanno recitato in almeno un film con un attore/produttore/regista |
| | stampare il numero di Bacon di un attore |

# Struttura Dati Implementata

La SD implementata nel laboratorio è formata da una parte di Hash Table con liste di collisione circolari doppiamente collegate e da una parte di grafo;
La Hash Table suddivide i vari nodi in base all'iniziale del loro Label (a-z, numeri e caratteri speciali) e gli elementi delle liste di collisione sono simili ai nodi di un grafo.

Grazie a questa implementazione, questa struttura dati è altamente scalabile in modo da poter contenere database più grandi di quello proposto, mantenendo buone prestazioni.

Essa può essere rappresentata con il seguente schema:




Come si può notare, un array chiamato “*database*” contiene una cella per ogni lettera dell’alfabeto (ovvero 26 caratteri) e ognuna di esse punta ad una lista circolare doppiamente collegata contenente i nodi.
Oltre ai 26 possibili caratteri dell’alfabeto latino, si trova anche una cella per i numeri e una per i caratteri che non rientrano nei casi precedenti.

In questo modo possiamo suddividere le persone e i film in base alla loro iniziale, ottimizzando la ricerca di un nodo e mantenendo una complessità costante nelle operazioni di inserimento.
Inoltre sarà più facile stampare l'intero database seguendo un ordine, ovvero quello alfabetico.

Dal punto di vista dell'occupazione di memoria, questa struttura dati si basa su un array di dimensione costante e non vuoto e puntatori creati su richiesta, quindi non sono previsti sprechi di memoria.

## Hash Table con liste di collisione circolari e doppiamente collegate

La funzione di Hash ritornerà il numero 0 per il carattere 'a', 1 per 'b', 2 per 'c' …
…25 per 'z', 26 per tutti i film che iniziano con un numero (es. "*300*" diretto da *Zack Snyder*) e 27 per tutti i film/persone il cui nome inizia con un carattere diverso dai casi precedenti (es. "*È stata la mano di Dio*" diretto da *Paolo Sorrentino*).

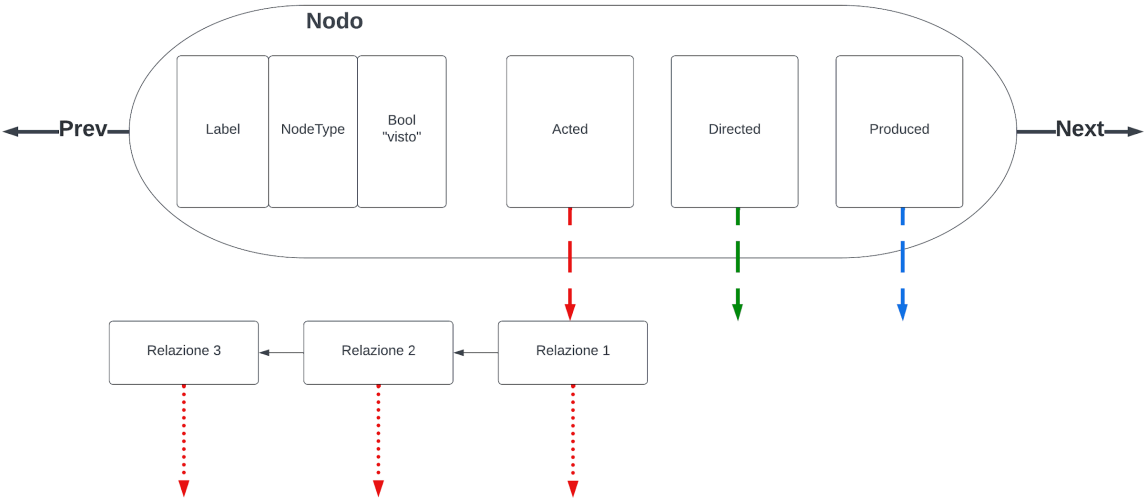
Grazie alla conversione *char* ↔ *int* della tavola ASCII, la funzione di Hash ha complessità costante in **Θ(1)** in quanto se il carattere ha un valore compreso:

- tra 48 e 57, è un numero;
- tra 65 e 90, è una lettera maiuscola;
- tra 97 e 122, è una lettera minuscola;
- se non rientra negli altri casi, è un carattere speciale.

In questo modo, l'array avrà una dimensione costante (28 celle di puntatori) che non varierà con l'aggiunta o la rimozione di uno o più nodi nel database.

Grazie alla lista circolare doppiamente collegata possiamo scandire tutti gli elementi partendo contemporaneamente dalla testa e dalla coda verso il centro della lista, dimezzando i tempi di ricerca di un nodo.
Questo fattore diventa fondamentale quando il database contiene una quantità di dati elevata, per questo non è stato trascurato nelle analisi di complessità delle funzioni implementate.

### I nodi delle liste di collisione




Ogni nodo delle liste di collisione è una struct composta da diversi elementi:

- un puntatore al nodo precedente ed uno al successivo;
- un *NodeType* che specifica di che tipo è il nodo (*PERSON* o *MOVIE*)
- un *Label* contenente il nome della persona o del film;
- un valore booleano chiamato *Visto* che serve a “marcare” il nodo nelle funzioni di ricerca;
- una lista semplice per ogni tipo di relazione possibile che contenga i puntatori ai nodi con cui si ha una relazione.

In questo modo, i nodi tra loro collegati da una relazione potranno “raggiungersi” immediatamente tramite un puntatore: grazie a questa implementazione, molte delle funzioni di ispezione come “stampare gli attori che hanno recitato in almeno un film con un attore/produttore/regista” avranno una complessità ottimizzata.

# Implementazione e Complessità

Con la struttura dati presentata, l’analisi di complessità delle funzioni implementate è così composta:

1. **Creazione di un nuovo database: Θ(N) dove N è la dimensione della Hash Table**
   Come *spiegato precedentemente*, la funzione di creazione del database alloca e inizializza in memoria 28 puntatori a liste di collisione, uno per ogni possibile carattere iniziale del Label.

2. **Aggiunta di un nuovo nodo: Θ(1) nel caso migliore, Θ(m/2) nel caso peggiore dove m è il numero di nodi nella lista di collisione**
   L'aggiunta in testa alla lista di collisione di un nuovo nodo è istantanea se questa risulta vuota;
   In caso contrario, la funzione ausiliaria *find* (con complessità Θ(m/2) dove m è la dimensione della lista di collisione) controllerà che non esista già il nuovo nodo per poi inserirlo in testa.
   Possiamo quindi affermare che l'aggiunta di un nuovo nodo ha complessità Θ(1) nel caso migliore, Θ(m/2) nel caso peggiore con m = numero di nodi nella lista di collisione.

3. **Aggiunta di un arco (o *relazione*) tra due nodi: Θ(m/2 + r) dove m è la dimensione della lista di collisione ed r il numero di relazioni del nodo in questione.**
   Dopo aver trovato i due nodi interessati tramite *find* (quindi Θ(m/2)), la funzione controlla che la relazione non esista già tramite la funzione ausiliaria *contains* (con complessità Θ(r) con r = numero di relazioni del nodo) per poi inserire in testa l'arco nella lista di puntatori apposita di entrambi i nodi (quindi in Θ(1)).
   Quindi la complessità totale è: Θ(m/2 + r) con "m" = numero di nodi nella lista di collisione ed "r" = il numero di relazioni del nodo.

4. **Numero di nodi nel database: Θ(TOT/2) dove TOT è il numero di nodi nel database**
   Questa funzione scandisce tutte le 28 liste di collisione dell'array e conta il numero di nodi di tipo PERSON o MOVIE.
   Come per la funzione *find*, la scansione avviene sia dalla testa sia dalla coda della lista verso il centro, in modo tale da dimezzare il tempo di esecuzione della funzione; per questo la complessità è Θ(TOT/2) con TOT = il numero di nodi nel database.

5. **Numero di archi di tipo T: Θ(TOT/2 * r) dove TOT è il numero totale di nodi nel database ed r è il numero di relazioni di ogni singolo nodo**.
   Dato che le relazioni tra due nodi sono bidirezionali e possono esistere solo archi di tipo PERSON-MOVIE, si possono contare solamente le relazioni di tutti i nodi di tipo MOVIE.

Come per la funzione *find*, vengono scansionate tutte le liste di collisione tramite due puntatori dimezzando il tempo di ricerca.
A questo punto, ogni qualvolta venga trovato un nodo di tipo MOVIE si contano tutte le sue relazioni di tipo T scandendo la sua lista di archi di dimensione r.
Quindi la complessità totale della funzione è Θ(TOT/2 * r) con TOT = numero di nodi nel database ed "r" = numero di relazioni di ogni singolo nodo.

6. **Lista di attori con una relazione di tipo R con una persona P: Θ(m/2 + r * a)**
   Grazie ad una funzione ausiliaria chiamata "*attoriColleghiDi*" si può ottenere una lista di tutti gli attori che hanno una relazione di tipo R con una persona P.

   Per fare ciò, si cerca il nodo P tramite *find* (Θ(m/2) con m = numero di nodi nella lista di collisione) e si guardano tutti gli archi di tipo R che P ha nella sua lista: Θ(r) dove r è il numero di relazioni di tipo R di P.
   A questo punto, per ogni MOVIE con cui P ha una relazione di tipo R, si leggono tutti gli attori di quel film evitando le ripetizioni grazie al flag booleano *"visto"*.

   A questo punto la funzione ritorna una lista contenente tutti gli attori che hanno recitato in film prodotti/diretti/interpretati dalla persona P.
   La complessità totale è quindi: Θ(m/2 + r * a) con:
   - m = numero di nodi nella lista di collisione;
   - r = numero di relazioni di tipo R di P;
   - a = numero di attori di un film in relazione con P;

   *Nota*: alla fine della ricorsione, questa funzione resetta tutti i "visto" su false tramite l'apposita funzione ausiliaria *ResetVisto* in modo tale da evitare errori nelle chiamate successive.

7. **Calcolo del numero di Bacon di un attore A**
   Per il calcolo del numero di Bacon è stata implementata una funzione che parte dall'attore *Kevin Bacon* e calcola tutti gli attori che hanno recitato con lui;
   Dopodiché controlla se tra questi è presente l'attore A e in caso affermativo ritorna 1; In caso negativo, per tutti gli attori "colleghi" di Bacon viene ritornato 1 + il valore della funzione eseguita ricorsivamente.
   In questo modo, gli attori che hanno collaborato "direttamente" con *Kevin Bacon* avranno numero di Bacon = 1, i colleghi di essi avranno numero di Bacon = 2 e così via.

L'implementazione prevede la ricerca tramite *find* (Θ(m/2) con m = numero nodi della lista di collisione) dell'attore di cui si vuole sapere il numero di Bacon, per poi cercare ricorsivamente la sua relazione con il famoso interprete tramite la funzione ausiliaria *baconAux*:

```
int baconAux(
      const MovieDB &mdb,
      const list::List& listaAttori,
      const Label L);
```

Essa controlla se in *listaAttori* sia presente l'attore A (con Label L) e quindi farà N confronti dove N è il numero di attori in *listaAttori*:

- in caso affermativo ritorna 1;
- in caso negativo, cerca (tramite *find*) il nodo dell'attore in questione e, se non è già stato marcato come *"visto"*, aggiungi i suoi colleghi (tramite *coActors*) alla lista di attori che verrà passata al prossimo ciclo ricorsivo e rendi il valore booleano *"visto"* del nodo uguale a *true*, per evitare ulteriori scansioni inutili.

Se la lista di attori risulta vuota, vuol dire che sono stati già visitati tutti i nodi correlati alla persona ricercata, quindi ritorna il valore d'errore *NO_BACON_NUMBER.*

Infine, dato *n* = numero di ricorsioni effettuate, la complessità finale della funzione che ritorna il calcolo del numero di Bacon di un attore A è

**T(*n*): {**
**Θ(1)** ***se n = 0;***
**Θ(m/2 + (coActors * N))** ***se n = 1;***
**T(n-1) + Θ(coActors * N)** ***se n >= 2;***
**})**

dove:

- m è il numero di nodi nella lista di collisione;
- coActors è la complessità della funzione omonima;
- N è il numero di attori "colleghi" con l'attore del livello n-1.

*Nota*: alla fine della ricorsione, *baconAux* resetta tutti i "visto" su false tramite l'apposita funzione ausiliaria *ResetVisto* in modo tale da evitare errori nelle chiamate successive.

## Funzioni Ausiliarie:

8. **Find: Θ(m/2) dove m è il numero di nodi nella lista di collisione**
   La funzione *find* cerca il nodo con label L nella sua lista di collisione. Essa legge la lista sia dalla testa sia dalla coda verso il centro, quindi la sua complessità sarà nel caso peggiore Θ(m/2).
   Il caso migliore con complessità Θ(1) si verifica quando il nodo cercato è il primo o l'ultimo della lista.

9. **Contains: Θ(r) dove r è il numero di relazioni del nodo**
   La funzione *contains* scandisce tutta la lista di relazioni di un nodo e controlla se è già presente uno specifico arco o meno.
   Per questo la sua complessità dipende dalla lunghezza della lista di relazioni r: Θ(r) nel caso peggiore in cui l'arco cercato non esiste e Θ(1) nel caso in cui l'arco cercato sia il primo della lista.

10. **ResetVisto: Θ(TOT/2) dove TOT è il numero totale di nodi nel database**
    Questa funzione ausiliaria scandisce tutto il database tramite due puntatori alle liste di adiacenza e imposta tutti i flag booleani *"visto"* su *false.*
    Dato che deve scandire tutta la struttura dati, la sua complessità è Θ(TOT/2) con TOT = numero totale di nodi nel database.

# Altro

All'inizio del file movies.cpp, sono state implementate un paio di funzioni e struct per definire e lavorare con le liste di puntatori più comodamente.

Purtroppo, per questioni di *include* non si è potuto creare sottocartelle in modo da dividere e ordinare meglio il codice.
In particolare, le liste di puntatori dovevano puntare a VertexNode, che erano definiti in *"movies.h"* ma allo stesso tempo movies.h doveva importare le liste di puntatori, creando un "*loop di include"* con errori del tipo: "`redefinition of list::List twice`"oppure "`redefinition of movies::VertexNode`".

Questa relazione è accessibile e commentabile su Google Docs all'indirizzo:
[Relazione Labo 10 - Documenti Google]

Se si hanno problemi ad aprire il link soprascritto, copiare ed incollare il seguente link:

https://docs.google.com/document/d/1mwiqJ3No_HdR-3Uxe7SwCjFwH7bkS13iQcSRiZWzgy0/edit#